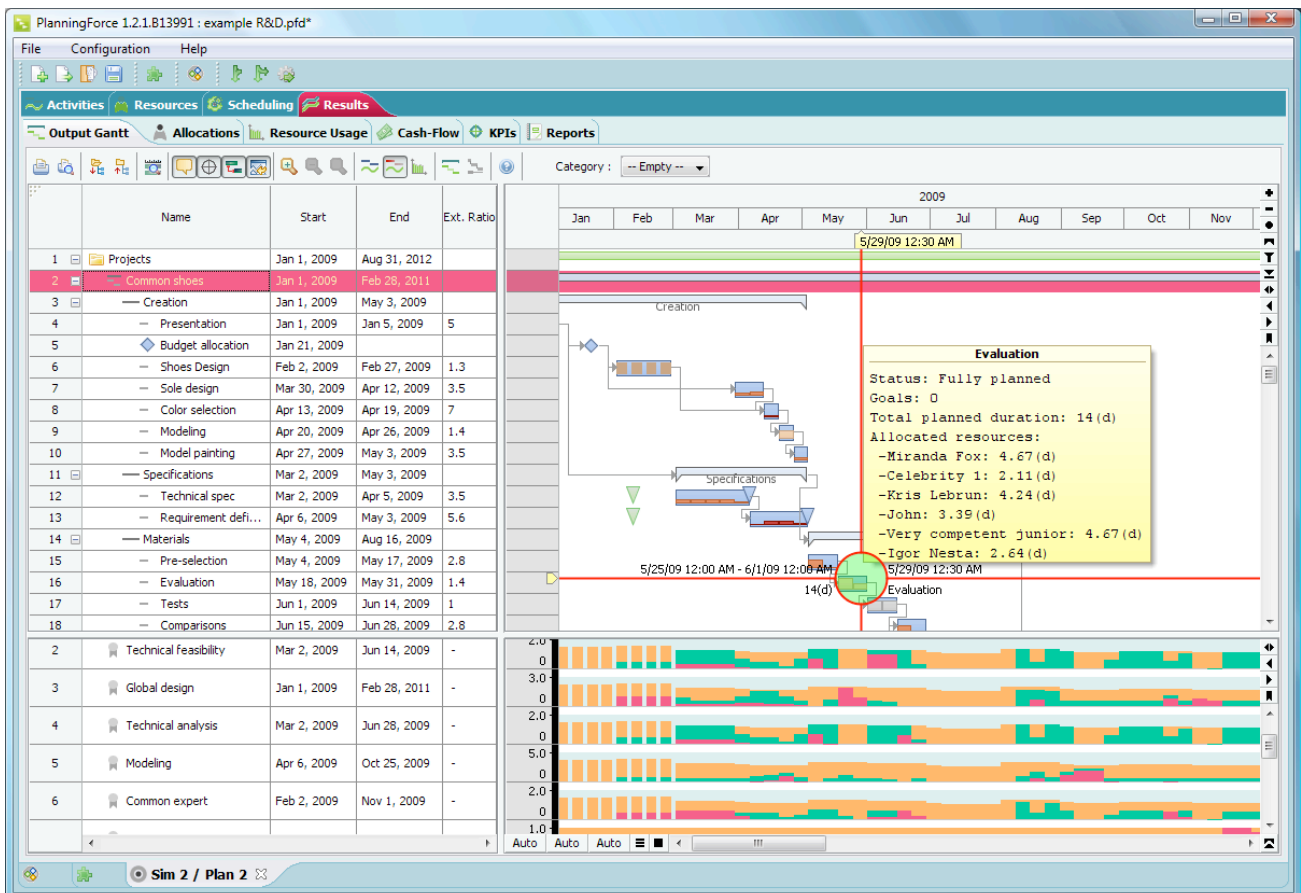


FlexGantt - Release 1

Layers, Layer Container, Layer Factory



Dirk Lemmermann Software & Consulting
 Asylweg 28
 8134 Adliswil
 Switzerland
www.dlsc.com

All rights reserved.
 Java is a trademark registered ® to Sun Microsystems
<http://java.sun.com>

The image on the title page shows the application „PlanningForce“, which makes heavy use of FlexGantt. For more information on this product go to <http://www.planningforce.com>

FlexGantt Release 1.1.2

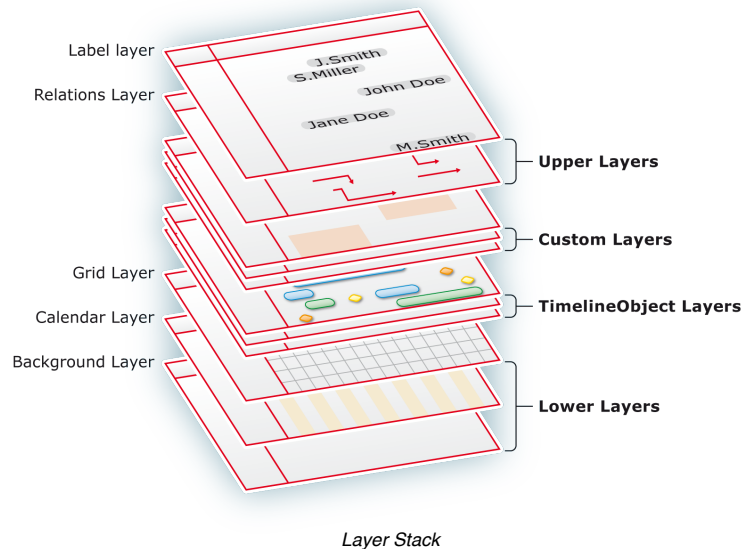
Document last updated on October 1st, 2008

Table of Content

Layer Container	1
Model vs. UI Layer	1
Layer Ordering	1
Layer Lookup	2
Layer Selector Support	2
Overview / Radar Selector Support	2
Highlighting	3
Selection Models	4
Layer Factory	4
Layer Policies	5
Layer Features	5
System Layers	7
Background Layer	8
Calendar Layer	8
Crosshair Layer	9
DatelineLayer	9
DragLayer	9
<i>Cursors</i>	10
<i>Edit Modes / Edit Mode Controller</i>	12
EditingLayer	13
EventlineLayer	13
GridLayer	14
LabelLayer	14
LassoLayer	15
<i>Lasso Modes</i>	15
<i>Cursors</i>	16
<i>Selection Behaviour</i>	17
PopupLayer	18
RelationshipLayer	18
RowLayer	19
SelectionLayer	19
TimeNowLayer	19
Timeline Object Layers	20
Custom Layers	21
Frequently Asked Questions (FAQ)	24
How can I change the transparency of a layer?	24
How can the layer transparency be updated in real-time when using the slider in the layer selector?	24
How can I use different colors for different layers?	24
Is it possible to remove the layer UI / layer palette / layer selector?	25
How many layers can a Gantt chart display?	25

Layer Container

The right-hand side of every Gantt chart is a layer container. This container gets created by the component factory of the Gantt chart. The following image depicts the structure of the layer container and shows how the container is composed of a stack of layers. The layer container manages three different types of layers that are supported by the framework: system layers, timeline object layers, and custom layers.



System layers are non-optional layers required by the framework. They provide the various features supported by the Gantt chart. Timeline object layers are responsible for rendering activities and events (the data). Custom layers are used to extend the feature list to meet application-specific requirements.

Model vs. UI Layer

When reading about layers and the layer container it is important to understand the difference between model layers and user interface (UI) layers. Model layers are provided by the Gantt chart model. The *IGanttChartModel* interface defines a method called *getLayers()*. This method returns an iterator over *ILayer* instances. Each one of these *ILayer* instances will result in the creation of a timeline object layer, which is a UI layer that will be added to the layer container. UI layers have graphical attributes while model layers only carry data and features requests.

Layer Ordering

The layers approach found in **FlexGantt** is very similar to the one used in graphics applications. These applications have many things in common but the one thing that can always be found is different ways of changing the order of layers. It is only logical that **FlexGantt** supports the same feature. The following table lists the available operations:

Method	Description
<code>void moveBack(ILayer)</code>	Moves the given layer behind the layer that is currently shown below it.
<code>void moveForward(ILayer)</code>	Moves the given layer in front of the layer that is currently shown above it.
<code>void moveToBack(ILayer)</code>	Moves the given layer all the way to the back. All other layers will be drawn on top.
<code>void moveToFront(ILayer)</code>	Moves the given layer all the way to the front. All other layers will be drawn below.
<code>void hideLayer(ILayer)</code>	Hides the given layer. The layer becomes invisible.
<code>void showLayer(ILayer)</code>	Shows the given layer. The layer becomes visible.

It should be noted that these operations are only available for timeline object and for custom layers. System layers appear in a fixed order, which gets determined by two protected methods of *LayerContainer*. This default order can be

changed by overriding these methods. For more information on this topic, please refer to the chapter called „System Layers“.

Layer Lookup

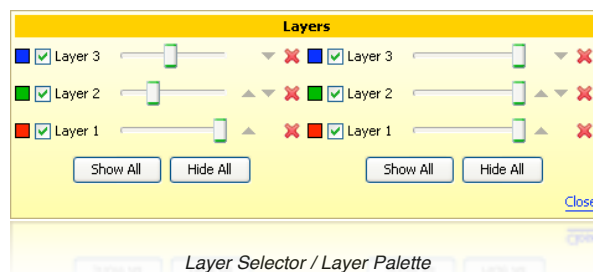
Layers are not „manually“ added to or removed from the layer container. Instead the *LayerContainer* class manages the life-cycle of layers. For the layer creation it delegates to a layer factory. After the layers have been created the container owns them. The following methods are available for layer access:

Method	Description
<code><T extends AbstractSystemLayer> T getSystemLayer(Class<T> layerType)</code>	Returns the system layer instance that was created for the given system layer type.
<code>TimelineObjectLayer getTimelineObjectLayer(ILayer)</code>	Returns the timeline object layer instance that was created for the given model layer instance.
<code>AbstractCustomLayer getCustomLayer(ILayer)</code>	Returns the custom layer instance that was created for the given model layer instance. This method will only work if the model layer is a custom model layer. <code>boolean ILayer.isCustomLayer()</code>

It is important to know that the *LayerContainer* recreates its layer stack after certain events have occurred on the Gantt chart model¹. Hence, applications should not keep references to the layers for a longer period of time.

Layer Selector Support

The layer container stores an icon map used to retrieve icons for layers. These icons can then be used in other **Flex-Gantt** components like the *LayerPalette* / *LayerSelector*. The *ILayer* class can not provide an icon by itself, since the model and the UI are supposed to be in different classes².



Method	Description
<code>Icon getLayerIcon(ILayer) void setLayerIcon(ILayer, Icon)</code>	Sets / gets an icon for the given layer. FlexGantt registers a default icon, which gets retrieved from the <i>IconRegistry</i> via the key <i>IconID.LAYER</i> .

Overview / Radar Selector Support

The overview selector implements a radar-like feature, which provides an overview over the entire data currently stored in the Gantt chart model. Each timeline object gets shown as a thin line or a small dot. To add a little bit more meaning to this kind of representation the application developer can map different colors and icons to so-called „status“ objects. A good example for a status object is the value of an enumerator, which lists the different states that a timeline object can be in.

¹ See *GanttChartModelEvent.ID.LAYER_ADDED* or *GanttChartModelEvent.ID.LAYER_REMOVED*.

² Model View Controller concept (MVC) / Separation of Concerns (SoC)

```
public enum Status {
    NEW,
    PLANNED,
    EXECUTED,
    ARCHIVED
}
```

The values of this enumerator can then be used to register the colors and icons:

```
layerContainer.setTimelineObjectStatusColor(Status.NEW, Color.WHITE);
layerContainer.setTimelineObjectStatusColor(Status.PLANNED, Color.RED);
layerContainer.setTimelineObjectStatusColor(Status.EXECUTED, Color.GREEN);
layerContainer.setTimelineObjectStatusColor(Status.ARCHIVED, Color.GRAY);

//
// Only display an icon for new timeline objects.
//
layerContainer.setTimelineObjectStatusColor(Status.NEW, new NewIcon());
```

The status objects can then be set on the timeline objects:

```
DefaultTimelineObject tlo = new DefaultTimelineObject();
tlo.setStatusObject(Status.NEW);
```

These settings will cause the overview selector to display the timeline objects in four different colors depending on their state. Those timeline objects that are new will also show an icon, which will make it easy for the user to navigate to them and to perform some kind of scheduling action on them.

Method	Description
Icon <code>getTimelineObjectStatusIcon(Object)</code> void <code>setTimelineObjectStatusIcon(Object, Icon)</code>	Gets or sets an icon for the given status object.
Color <code>getTimelineObjectStatusColor(Object)</code> void <code>setTimelineObjectStatusColor(Object)</code>	Gets or sets a color for the given status object.

Highlighting

An extremely helpful feature provided by the *LayerContainer* is its ability to highlight timeline objects. Highlighting means that one or more timeline objects will be drawn in two different ways, which will cause them to blink. Making them blink will force the user's attention on them. Highlighting is implemented as a specialized thread (inner class *HighlightThread*), which periodically puts the layer container into two different states and then calls its *repaint()* method. The two states are passed to the timeline object renderers. This way the renderers can draw the highlight feedback. The default renderer implementations simply use different colors for the two states. This is sufficient to make them blink. The following code fragment is taken from the *DefaultTimelineObjectRenderer* class. The method paints the content of an activity (bar):

```
/**
 * Renders the timeline object if it is an activity (different start and end
 * time = duration).
 */
protected void paintActivityContent(Graphics g) {
    int w = getWidth() - 1;
    int h = getHeight() - 1;
    Graphics2D g2d = (Graphics2D) g;
    if (highlighted) {
        g2d.setPaint(new GradientPaint(0, 0, highlightFillColor1, 0, h / 2,
            highlightFillColor2));
    } else if (selected) {
        g2d.setPaint(new GradientPaint(0, 0, selectionFillColor1, 0, h / 2,
            selectionFillColor2));
    } else {
        g2d.setPaint(new GradientPaint(0, 0, activityFillColor1, 0, h / 2,
            activityFillColor2));
    }
    ... more drawing going on here
}
```

The „highlighted“ flag was set in the *getTimelineObjectRendererComponent()* method of the renderer. The following table lists the methods related to highlighting:

Method	Description
<code>void addHighlightedObject(TimelineObjectPath path)</code>	Adds a timeline object path to the list of highlighted objects. The layer container will automatically start the highlighting thread.
<code>void addHighlightedObjects(Collection<TimelineObjectPath> paths)</code>	Adds several timeline object paths at the same time. The layer container will automatically start the highlighting thread.
<code>void removeHighlightedObjects(Collection<TimelineObjectPath> paths)</code>	Removes a timeline object path from the list of highlighted objects. The layer container automatically stops the highlighting thread if the list is now empty.
<code>void clearHighlightedObjects()</code>	Clears the list of highlighted timeline objects. The highlighting thread will be stopped automatically.
<code>boolean isHighlighted(TimelineObjectPath path)</code>	Determines if the given timeline object is a member of the list of highlighted objects.
<code>void setHighlighting(boolean b)</code>	Turns highlighting on and off. This method gets called by the highlight thread with the alternating values „true“ and „false“.
<code>boolean isHighlighting()</code>	Determines if the layer container is currently in the second mode „highlight“.
<code>long getHighlightingDelay(); setHighlightingDelay(long delay)</code>	Gets / sets the delay between two calls of the highlighting thread to <i>setHighlighting(boolean)</i> . The lower this value, the higher the blink frequency.

Some notes:

- The layer container automatically starts or stops the highlight thread whenever a timeline object path gets added to or removed from the list of highlighted paths.
- The highlight feature is used by the Gantt chart when it tries to show a „message context“. If the message is of type *TimelineObjectPathMessage* then the timeline object path stored in the message object will be passed to the layer container's *addHighlightedObject(TimelineObjectPath)* method.
- Highlighting will only be visible if the renderers support it. Please make sure that your renderers make use of the „highlighted“ flag passed to them.

Selection Models

The layer container manages the selection models used for the timeline object layers. This is due to the fact that the UI layers for the model layers are very volatile. They can be created and deleted after model changes but the selections should remain the same. This is why the layer container manages a map of selection models where the model layers are used as the key.

Method	Description
<code>ITimelineObjectLayerSelectionModel getSelectionModel(ILayer layer) void setSelectionModel(ILayer, ITimelineObjectLayerSelectionModel)</code>	Gets / sets the selection model for the given layer.

Layer Factory

The layer container delegates the creation of layers to a layer factory, which needs to implement the *ILayerFactory* interface. A default factory (*DefaultLayerFactory*) is included and will be sufficient for most applications. However, there are reasons why an application might choose to provide its own factory: to have a single point in the application where layers get configured, to use subclasses of the built-in layers or to use custom layers. The following code shows the factory definition:

```

/**
 * Layer factories are used by layer containers in order to create the actual
 * user interface components for the various system layers, object layers,
 * and custom layers.
 */
public interface ILayerFactory {

    /**
     * Creates a new system layer. The creation of all system layer types must
     * be supported (all subclasses of AbstractSystemLayer).
     */
    <T extends AbstractSystemLayer> T createSystemLayer(LayerContainer lc,
        Class<T> layerType);

    /**
     * Creates a new timeline object layer to be used for rendering timeline
     * objects.
     */
    TimelineObjectLayer createTimelineLayer(LayerContainer lc, ILayer layer);

    /**
     * Creates a new custom layer to be used for rendering custom information.
     */
    AbstractCustomLayer createCustomLayer(LayerContainer lc, ILayer layer);
}

```

The method, which creates the system layers is rather restricted. The method has to guarantee that the layer created by it is of the type passed to it. The method is not allowed to return null. The method, which creates custom layers will only be called if *ILayer.isCustomLayer()* returns „true“.

Applications that want to replace the default layer factory need to pass their own factory to the constructor of the *LayerContainer* class. The layer container itself gets created by the component factory of the Gantt chart, hence replacing the default layer factory also requires replacing the default component factory. The following code fragment shows a custom component factory using a custom layer factory:

```

public class CustomComponentFactory extends DefaultComponentFactory {

    /**
     * Creates a layer container that uses a custom layer factory.
     */
    @Override
    public LayerContainer createLayerContainer(AbstractGanttChart gc,
        TreeTable table, IGanttChartModel model) {
        return new LayerContainer(gc, model, table, new CustomLayerFactory());
    }
}

```

This custom component factory can then be passed to the constructor of the Gantt chart.

```
GanttChart gc = new GanttChart(new CustomComponentFactory());
```

Layer Policies

The layer container provides all policies required by its layers. The container owns a policy provider instance (*IPolicyProvider*, see „FlexGantt - Policies & Commands“) that will be used for storing and retrieving policies based on policy interfaces. Many layers use these policies to control their behaviour and / or appearance. The popup layer for example looks up a policy of type *IPopupPolicy* and then looks up a popup value object from the policy for a given timeline object. The type of this value object is used to lookup a popup renderer that will ultimately perform the rendering of the popup window.

Layer Features

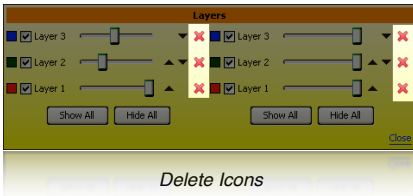
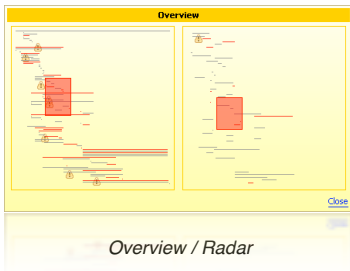
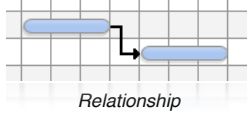
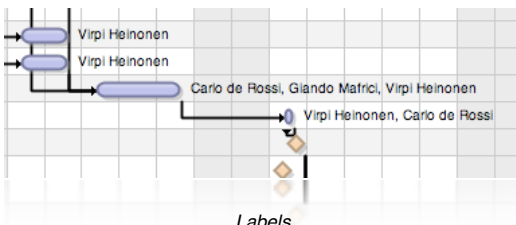
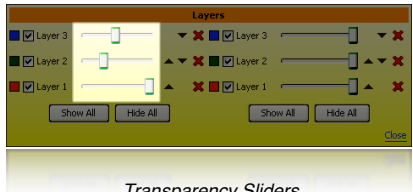
The behaviour of several **FlexGantt** components can be controlled by the model layer instances themselves. The *ILayer* interface defines a method called *isFeatureEnabled(Feature)*. The available features are defined in the enumerator *Feature*. The default *ILayer* implementation *Layer* defines methods to add and remove features to and from the layer.

```

public void Layer.addFeature(Feature);
public void Layer.removeFeature(Feature);

```


The following table lists the *Feature* enumerator values:

Feature	Description
DELETION	<p>If enabled allows the user to delete the layer from the Gantt chart. A „delete“ icon will be visible in the layer selector.</p>  <p><i>Delete Icons</i></p>
OVERVIEW	<p>If enabled considers the layer for the overview / radar selector. The timeline objects shown in the layer will be visible in the radar.</p>  <p><i>Overview / Radar</i></p>
RELATIONSHIPS	<p>If enabled will cause the <i>RelationshipLayer</i> to draw the relationships defined in the Gantt chart model for the timeline objects shown on this layer.</p>  <p><i>Relationship</i></p>
SHOW_IN_PALETTE	<p>If enabled will cause the layer selector to list the layer. See screenshot for feature DELETION.</p>
TIMELINE_OBJECT_CREATION	<p>The <i>LassoLayer</i> will consider the layer for timeline object creation. The <i>LassoLayer</i> will show a layer selection dialog when the user creates a new timeline object only if several timeline object layers enable this feature. If only one layer has this feature enabled the timeline object will automatically be created on that layer.</p>
TIMELINE_OBJECT_DESCRIPTIONS	<p>The <i>LabelLayer</i> will only attempt to draw labels next to a timeline object if its layer enabled this feature.</p>  <p><i>Labels</i></p>
TRANSPARENCY	<p>If this feature is enabled it will cause the <i>LayerSelector</i> to display a transparency slider next to the layer's name.</p>  <p><i>Transparency Sliders</i></p>

System Layers

System layers are non-optional layers that are required by the Gantt chart in order to work properly. The number of system layers is fixed. Each one of these layers implements a small set of features that the framework guarantees to be available. The *TimeNowLayer* for example draws a vertical line at the location of the current system time. The layer container manages an upper and a lower stack of system layers. Timeline object layers and custom layers are placed between these two stacks. The stacks and the ordering of the system layers within the stacks are controlled by two methods of the layer container:

```
protected List<Class<? extends AbstractSystemLayer>> getTypesOfUpperSystemLayers();
protected List<Class<? extends AbstractSystemLayer>> getTypesOfLowerSystemLayers();
```

The following is a list of all system layers. Some of them can be turned on or off, depending on whether the feature that they support is currently enabled or not.

Layer	Description
BackgroundLayer	Fills the background of the layer container with a solid color or a texture (optional alternating row colors).
CalendarLayer	Draws the information returned from the calendar model (weekends, holidays) via the help of <i>ICalendarEntryRenderer</i> instances. <code>public void AbstractGanttChart.setCalendarVisible(boolean)</code>
CrosshairLayer	Draws a crosshair that can be used in training sessions. <code>public void AbstractGanttChart.setCrosshairVisible(boolean)</code>
DatelineLayer	Fills the currently focused time span with a transparent color. Also visualizes the bounds of the horizon of the timeline. These bounds will only become visible after some zooming operations (e.g. user scrolled to the end of the horizon and then zoomed out).
DragLayer	Performs anything related to drag & drop operations. Renders the dragged timeline object. Fills the background of all rows via the <i>IDragRowRenderer</i> instances.
EditingLayer	Handles in-place editing of timeline objects.
EventlineLayer	Draws the time spans and time points of eventline activities / events inside the layer container.
GridLayer	Draws the vertical grid lines. <code>public void AbstractGanttChart.setGridLineMode(GridLineMode)</code> <code>public void AbstractGanttChart.setVerticalLinesOnTop(boolean)</code>
LabelLayer	Draws the text labels next to the timeline objects. <code>public void AbstractGanttChart.setLabelsVisible(boolean)</code>
LassoLayer	Shows the lasso when the user performs a selection with the control key down.
PopupLayer	Renders little popup windows with detailed information about a timeline object.
RelationshipLayer	Draws lines connecting two timeline objects if a relationship exists between them.
RowLayer	Renders the background of individual rows via the help of <i>IRowRenderer</i> instances.
SelectionLayer	Fills selected time spans with a solid color.
TimeNowLayer	Draws a vertical line at the location of the current system time. <code>public void AbstractGanttChart.setTimeNowVisible(boolean)</code>

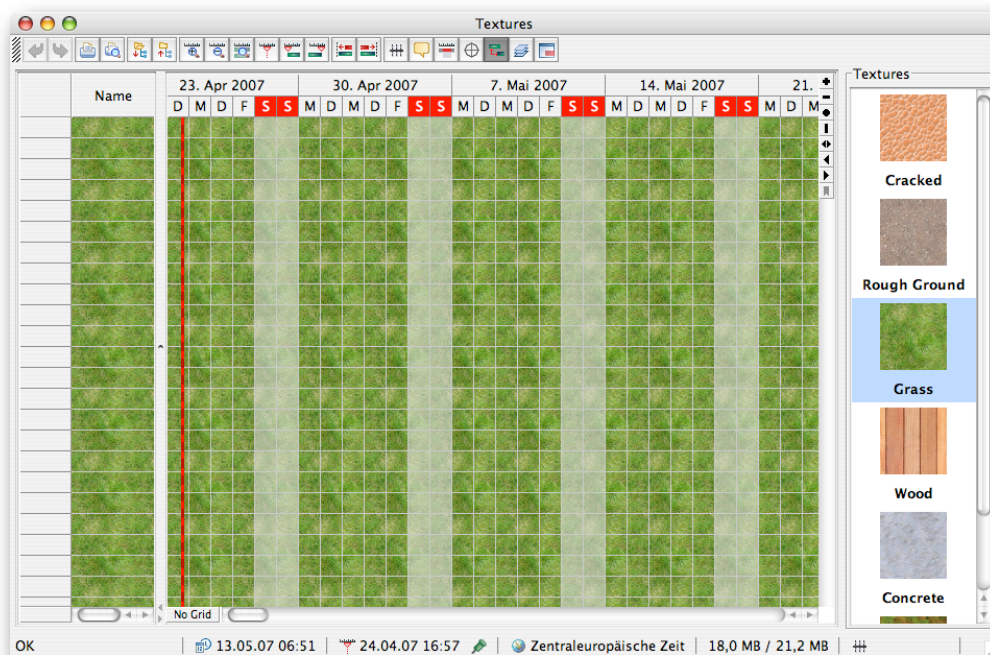
Background Layer

As its name already indicates the background layer is responsible for drawing the background of the layer container. In many situations this will simply be a rectangle filled with the background color of the layer container itself. If an application also specifies an alternating background color on the layer container then the layer will fill a rectangle with this color in every odd row. This greatly increases the readability of the Gantt chart. The transparency of the alternating background color can be controlled by the layer's alpha value.

For even higher customization the background layer will also accept a texture image that will be drawn as tiles until it fills the entire background. This is a nice feature for creating a certain them (e.g. a soccer scheduling application might show grass in the background). The tree table matches several of the methods of the background layer so that the left-hand side and the right-hand side can be drawn consistently.

The following image shows a snapshot of the "Textures" demo application. In this demo the user can select a texture from a palette of available textures. The selected texture image gets assigned to the background layer via a call to:

```
void BackgroundLayer.setTexture(Image img);
```



Texture Demo Application

Related API:

```
void LayerContainer.setBackground(Color col)
void LayerContainer.setAlternatingBackground(Color col)
void TreeTable.setAlternatingBackground(Color col)
void TreeTable.setTexture(Image img)
void AbstractLayer.setAlpha(float alpha)
```

Calendar Layer

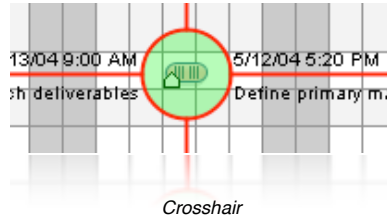
Every Gantt chart can have a calendar model attached to it. This model returns calendar entries, which represent things like weekends or holidays. These entries are visualized by the calendar layer via the help of renderers, which have to implement the *ICalendarEntryRenderer* interface. A renderer component will always reach from the very top of the layer container to the very bottom. Its start location (x-coordinate) and its width are dependent on the start and end time of the calendar entry that they represent. The layer has been setup to gray out Saturdays and Sundays (weekends).

Related API:

```
void CalendarLayer.setCalendarEntryRenderer(Class entryType, ICalendarEntryRenderer renderer);
ICalendarModel AbstractGanttChart.getCalendarModel();
TimeGranularityCalendarModel
WeekendCalendarEntry
HolidayCalendarEntry
```

Crosshair Layer

The crosshair layer draws a crosshair at the current mouse location. This feature is especially useful when giving a demonstration of the Gantt chart. The crosshair will only be visible if it has been turned on. A pre-defined action called *CrosshairAction* is available for toggling the visibility of the crosshair. The layer uses the *ICrosshairPolicy* to lookup four different labels that will be shown in the four corners of the crosshair. Several methods are available on the layer that can be used to customize the appearance of the crosshair (radius, fill color, line color, text color).

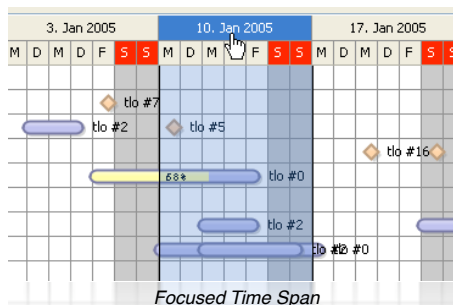


Related API:

```
void AbstractGanttChart.setCrosshairVisible(boolean);
ICrosshairPolicy
CrosshairAction
```

DatelineLayer

The dateline layer visualizes the time span that currently has the focus in the dateline so that the user can more easily identify the timeline objects that intersect with this span. The focused time span is the time span between two major or minor grid locations, depending on whether the mouse cursor hovers over the upper or the lower part of the dateline. The dateline layer will draw a semi-transparent rectangle from the top to the bottom of the layer container. The transparency of the rectangle can be controlled with the layer's alpha value. The layer will draw the same rectangle when the user performs a time span selection in the dateline.



Related API:

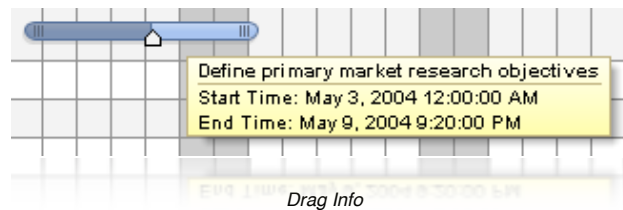
```
ITimeSpan Dateline.getFocusedTimeSpan();
void DatelineLayer.setFocusedTimeSpanFillColor(Color);
void DatelineLayer.setFocusedTimeSpanLineColor(Color);
void DatelineLayer.setFocusedTimeSpanVisible(boolean);
void AbstractLayer.setAlpha(float);
```

DragLayer

The drag layer handles all interaction and visualization required in order to support drag & drop operations on timeline objects. The layer uses various policies to determine ...

- if an object can be dragged from one row to another (*IDragAndDropPolicy*)
- if the duration of an object can be changed (*IEditTimelineObjectPolicy*)
- if the start time of an object can be changed (*IEditTimelineObjectPolicy*)
- if the „percentage complete“ value of an activity object can be edited (*IEditActivityObjectPolicy*)
- if the „capacity used“ value of a capacity object can be edited (*IEditCapacityObjectPolicy*)

Additionally the *IDragInfoPolicy* is used to lookup a drag info object that contains the information that shall be displayed in the drag info popup. An *IDragInfoRenderer* instance gets used to visualize this information.






Another rather advanced feature provided by the drag layer is the ability to change the background of rows when the user performs a drag and drop operation. Renderers of type *IDragRowRenderer* are invoked only during a drag. They can make their own contribution to the row background. The renderer *DefaultDragRowRenderer* for example, covers the regular background with a semi-transparent gray color to indicate that a drop is not valid at the current mouse location. Many applications use this feature to provide feedback to the user about the quality of a row as a drop location by filling the background with different colors (red, yellow, green).



Related API:

```
IDragAndDropPolicy
IDragInfoPolicy
IDragInfoRenderer
IDragRowRenderer
DefaultDragInfoRenderer
DefaultDragRowRenderer
IEditTimelineObjectPolicy
IEditActivityObjectPolicy
IEditCapacityObjectPolicy
```

Cursors

The drag layer uses different cursor shapes to indicate the available editing options at the current mouse cursor location. The following table lists the icons and their meaning:

Icon	Icon ID	Description
	CURSOR_CHANGE_START_TIME	Shown when the mouse cursor hovers over the left edge of the timeline object and the user can change the start time of the object. Related API: <pre>boolean IEditTimelineObjectPolicy .isStartTimeChangeable (TimelineObjectPath, IGanttChartModel); ITimelineObject.isStartTimeChangeable ();</pre>
	CURSOR_CHANGE_END_TIME	Shown when the mouse cursor hovers over the right edge of the timeline object and the user can change the duration of the object. Related API: <pre>boolean IEditTimelineObjectPolicy .isDurationChangeable (TimelineObjectPath, IGanttChartModel); ITimelineObject.isDurationChangeable ();</pre>
	CURSOR_MOVE	Shown when mouse cursor hovers over the center of the timeline object and the user can move the timeline object left and right and from one row to another. How the layer determines this option is described above for the MOVE_VERTICAL / HORIZONTAL cursors.

Icon	Icon ID	Description
	CURSOR_EDIT_CAPACITY	<p>Shown when the mouse cursor hovers over the top edge of a capacity object and the user can change the „capacity used“ value of the object.</p> <p>Related API:</p> <pre>boolean IEditCapacityObjectPolicy .isCapacityChangeable(TimelineObjectPath, IGanttChartModel); boolean ICapacityObject.isCapacityChangeable();</pre>
	CURSOR_EDIT_PERCENTAGE	<p>Shown when the mouse cursor hovers over the left edge of an activity object and the user holds down the SHIFT key and changing the „percentage used“ value of the object is allowed.</p> <p>Related API:</p> <pre>boolean IEditActivityObjectPolicy .isPercentageChangeable(TimelineObjectPath, IGanttChartModel); boolean IActivityObject.isPercentageChangeable();</pre>

It should be noted that the behaviour described in the table is based on the use of the default edit mode controllers for timeline objects, capacity objects, and activity objects. More information on the edit mode controller is provided later in the chapter „Edit Mode Management“.

The drag layer retrieves all cursor icons from the *IconRegistry* via the given ID. The easiest way to replace these cursors is to register new icons by calling *IconRegistry.setIcon(IconID, Icon)*. Another option is to subclass the *DragLayer* class and override the *createEditModeCursor(EditMode)* method. The default implementation of this method is shown here:

```
/**
 * Factory method for the cursors of the various edit modes. The method
 * looks up the cursors from the {@link IconRegistry} via the following
 * identifiers:
 * <ul>
 * <li> {@link IconId#CURSOR_EDIT_CAPACITY }</li>
 * <li> {@link IconId#CURSOR_EDIT_PERCENTAGE }</li>
 * <li> {@link IconId#CURSOR_CHANGE_START_TIME }</li>
 * <li> {@link IconId#CURSOR_CHANGE_END_TIME }</li>
 * <li> {@link IconId#CURSOR_MOVE_HORIZONTAL }</li>
 * <li> {@link IconId#CURSOR_MOVE_VERTICAL }</li>
 * <li> {@link IconId#CURSOR_MOVE }</li>
 * </ul>
 * The hotspot for the cursors is always (16, 16) except for the percentage
 * complete cursor (31, 15).
 *
 * @param mode
 *         the edit mode for which to create a cursor
 * @return a cursor for the given edit mode
 * @since 1.0
 */
protected Cursor createEditModeCursor(EditMode mode) {
    IconId id = null;
    Point hotSpot = new Point(16, 16);
    switch (mode) {
        case CHANGE_CAPACITY:
            id = IconId.CURSOR_EDIT_CAPACITY;
            break;
        case CHANGE_PERCENTAGE_COMPLETE:
            id = IconId.CURSOR_EDIT_PERCENTAGE;
            hotSpot = new Point(31, 15);
            break;
        case CHANGE_START_TIME:
            id = IconId.CURSOR_CHANGE_START_TIME;
            break;
        case CHANGE_END_TIME:
            id = IconId.CURSOR_CHANGE_END_TIME;
            break;
        case CHANGE_TIME_SPAN:
            id = IconId.CURSOR_MOVE_HORIZONTAL;
            break;
        case CHANGE_NODE:
    }
```

```

        id = IconId.CURSOR_MOVE_VERTICAL;
        break;
    case CHANGE_NODE_AND_TIME_SPAN:
        id = IconId.CURSOR_MOVE;
        break;
    case NONE:
    default:
        return Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR);
    }
    ImageIcon icon = (ImageIcon) IconRegistry.getIcon(id);
    return Toolkit.getDefaultToolkit().createCustomCursor(icon.getImage(),
        hotSpot, mode.toString());
}

```

Edit Modes / Edit Mode Controller

The table above describes the various cursors shown when the mouse cursor hovers over certain parts of a timeline object. It is important to know that this behaviour is based on the default editor mode controllers registered on the drag layer. Application developers are free to provide their own edit mode controllers to fine-tune the behaviour of their application. Edit mode controllers have to implement the `IEditModeController` interface, which primarily consists of just one method:

```
EditMode getEditMode(DragLayer layer, ObjectBounds bounds, MouseEvent evt);
```

It is up to this method to determine the mode into which the layer will be put. The given object bounds are a complete description of the location of a timeline object. The mouse event is the event that triggered the method call.

FlexGantt maps several default edit mode controllers to default model classes. They are listed in the following table:

Model Class	Edit Mode Controller
Object	DefaultEditModeController
DefaultCapacityObject	CapacityObjectEditModeController
DefaultActivityObject	ActivityObjectEditModeController

Possible edit modes are listed in the *EditMode* enumerator:

Edit Mode	Description
CHANGE_CAPACITY	Only used by the <i>CapacityObjectEditModeController</i> . This feature is useful if the user wants to interactively change the „capacity used“ value of a capacity object.
CHANGE_END_TIME	Allows the user to change the duration of a timeline object. This will change the duration / the time span of the object.
CHANGE_NODE	Allows the user to move the timeline object between rows. This is considered a drag and drop operation and will result in calls to the <i>IDragAndDropPolicy</i> .
CHANGE_NODE_AND_TIME_SPAN	Allows the user to move the timeline object from one row to another and also to the left or the right. In this edit mode the user can drop the timeline object anywhere.
CHANGE_PERCENTAGE_COMPLETE	Only used by the <i>ActivityObjectEditModeController</i> . This feature is useful if the user wants to interactively change the „percentage complete“ value of an activity object.
CHANGE_START_TIME	Allows the user to change the start time of a timeline object. This will change the duration / the time span of the object.
CHANGE_TIME_SPAN	Allows the user to move the timeline object to the left and the right. This mode does not affect the duration of the timeline object.
NONE	This value indicates that no editing operation is available at the given mouse location

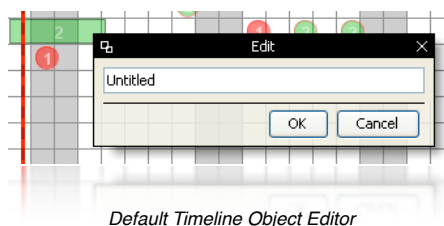
EditingLayer

The editing layer is responsible for displaying editors for in-place editing of timeline objects. If an editor is registered for a certain type of timeline object then this editor will be added to the layer when the user performs a double click on the object.

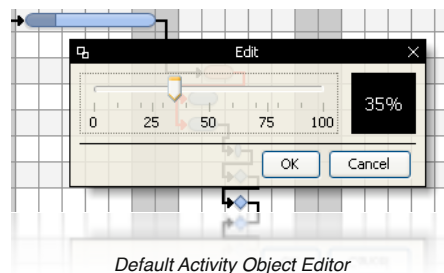
Editors are registered on the LayerContainer class and not on the layer itself. This is due to the fact that system layers are sometimes recreated after certain changes in the Gantt chart model:

```
void LayerContainer.setTimelineObjectEditor(Class objectType, ITimelineObjectEditor editor);
ITimelineObjectEditor LayerContainer.getTimelineObjectEditor(Class objectType);
```

FlexGantt ships with two built-in editors, one for timeline objects of type *DefaultTimelineObject* and one for objects of type *DefaultActivityObject*. Both of them subclass *AbstractTimelineObjectEditor*. It is highly recommended that application developers do the same if they have to implement their own editors.



Default Timeline Object Editor



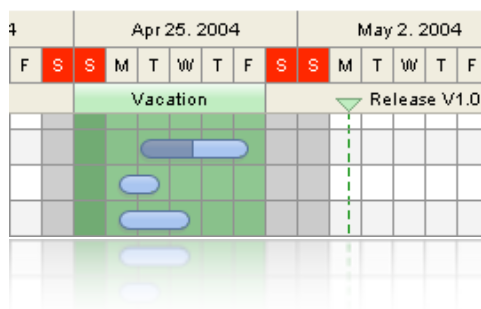
Default Activity Object Editor

Related API:

```
ITimelineObjectEditor
AbstractTimelineObjectEditor
DefaultTimelineObjectEditor
DefaultActivityObjectEditor
void EditingLayer.setTimelineObjectEditor(ITimelineObjectEditor editor);
```

EventlineLayer

The eventline layer picks up eventline activities and events from the eventline (model) and visualizes them in the layer container. Activities are shown as filled rectangles and events as vertical lines, both of them reaching from the top to the bottom of the layer container. Different strokes and paints can be specified for different eventline objects.



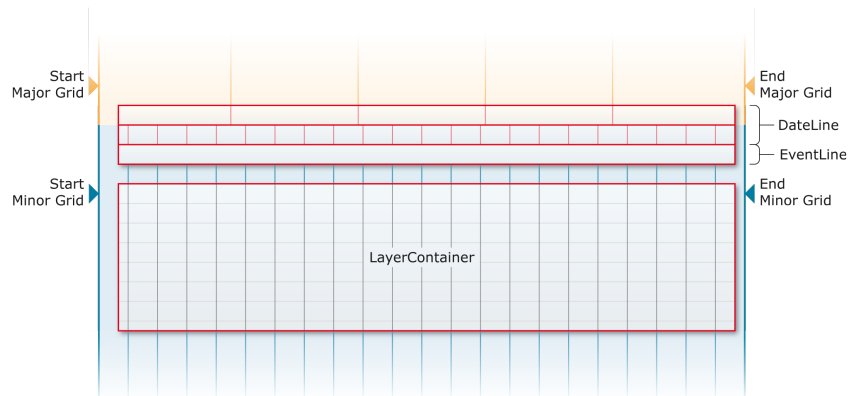
Related API:

```
void EventlineLayer.setPaint(Class eventlineObjectType, Paint paint);
void EventlineLayer.setStroke(Class eventlineObjectType, Paint paint);
```


GridLayer

The grid layer is responsible for drawing the vertical and horizontal grid lines. The locations of the horizontal grid lines are calculated based on the location of the rows and their individual heights. The locations of the vertical grid lines can be retrieved from the dateline model.

The grid layer can display the grid in different modes: no grid, minor grid, major grid, combined grid. In most cases the minor and the major grid lines can be shown at the same time because the major grid lines are usually on top of a minor grid line. But there are situations where this is not the case, for example when the timeline shows the major granularity 'months' and the minor granularity 'weeks'. The appearance of the grid lines would be visually unpleasant with major grid lines between two minor grid lines, sometimes closer to the left or the right line. To avoid this the grid layer uses a special policy called *IGridLinePolicy*. The layer queries this policy for the visibility of the major and minor grid lines. The currently used dateline model gets passed to the policy's methods as the results are dependent on it.



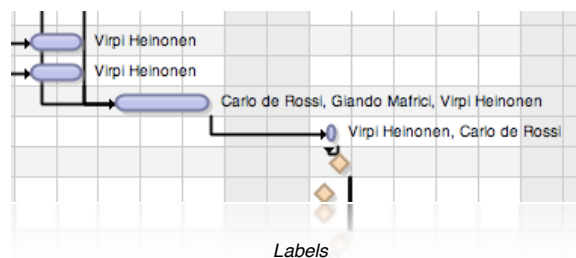
Major and Minor Grid Lines

Related API:

```
GridAction
GridLineMode
List<GridLine> IDatelineModel.getGrid(int x1, int x2, boolean major);
void AbstractGanttChart.setGridLineMode(GridLineMode);
IGridLinePolicy
```

LabelLayer

The label layer's only purpose is to draw descriptions next to timeline objects. A description text will only show up if the label policy returns one for the timeline object, if the timeline object layer on which the timeline object is located has the description feature turned on, and if the Gantt chart has labels turned on.



Example:

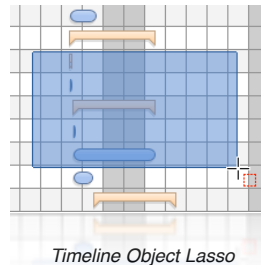
```
DefaultTimelineObject tlo = new DefaultTimelineObject();
tlo.setLabel(LabelType.DESRIPTION, "Steve Smith, Dirk Lemmermann")
```

Related API:

```
ILabelPolicy
String ITimelineObject.getLabel(LabelType);
void DefaultTimelineObject.setLabel(LabelType, String);
boolean ILayer.isFeatureSupported(Feature);
void AbstractGanttChart.setLabelsVisible(boolean);
```

LassoLayer

The lasso layer provides the necessary infrastructure to display a selection lasso, which is basically a rectangle that gets created when the user performs a mouse drag operation (pressing the SHIFT or the CTRL key while dragging allows for multiple selections). The layer can be configured to either select the timeline objects with which the lasso intersects or the time span represented by the start and width of the rectangle. Selected timeline objects are added to the selection model³ of the timeline object layer on which they are displayed⁴. Selected time spans are added to the selection model⁵ of the layer container.



Lasso Modes

The lasso layer can be used in four different modes, which are listed in the enumerator *LassoMode*:

Lasso Mode	Description
<code>SELECT_TIMELINE_OBJECTS</code>	The user can use the lasso to select timeline objects. The lasso layer supports several different selection behaviours. Read the chapter „Selection Behaviour“ for more information on this topic. Selected timeline objects are added to the selection model of the layer (<i>ITimelineObjectLayerSelectionModel</i>).
<code>SELECT_TIME_SPANS</code>	The user can use the lasso to select time spans. Selected time spans are added to the selection model of the layer container (<i>ILayerContainerSelectionModel</i>). The model stores the selected time spans for each row / node.
<code>CREATE_TIMELINE_OBJECTS</code>	The user specifies a time interval via the help of the lasso. When releasing the mouse button the lasso layer invokes the <i>IEditTimelineObjectPolicy</i> to determine whether a timeline object can be created at the given location and which command to use to create the object. The lasso layer even supports the creation of timeline objects on several rows at the same time. This behaviour can be controlled with the <i>setSingleRowObjectCreation(boolean)</i> method of <i>LassoLayer</i> .
<code>CREATE_RELATIONSHIP</code>	<p>The user can create a relationship between two timeline objects. The focused timeline object will be decorated by the layer with four red arrows around its edges. The user can then click on the focused object and drag to a second one. A line with an arrow head will be drawn. The line starts at the source timeline object and reaches to the current mouse cursor location.</p> <p>The lasso layer uses the <i>IRelationshipPolicy</i> to determine, which objects can be the source or the target of a relationship. The policy also returns the command needed to create the relationship and add it to the Gantt chart model.</p>

³ This selection model implements the *ITimelineObjectLayerSelectionModel* interface.

⁴ The selection models of the *TimelineObjectLayer* instances are also managed by the *LayerContainer* class. See *LayerContainer.getSelectionModel(ILayer)*

⁵ This selection model implements the *ILayerContainerSelectionModel* interface.







Normally the lasso mode gets set explicitly by calling:

```
public void LassoLayer.setLassoMode(LassoMode);
```

However, the `CREATE_TIMELINE_OBJECTS` mode can be entered by pressing the ALT key.

Cursors

The following table lists the cursors shown by the lasso layer. These cursors always only show up when the mouse cursor hovers over the background of a row but and not over a timeline object.

Icon	Icon ID	Description
	<code>CURSOR_CREATE_RELATIONSHIPS</code>	Shows up when the mode of the lasso layer was set to: <code>LassoMode.CREATE_TIMELINE_OBJECTS</code>
	<code>CURSOR_CREATE_TIMELINE_OBJECTS</code>	Shows up when the mode of the lasso layer was set to: <code>LassoMode.CREATE_RELATIONSHIP</code>
	<code>CURSOR_SELECT_TIMELINE_OBJECTS</code>	Shows up when the mode of the lasso layer was set to: <code>LassoMode.SELECT_TIMELINE_OBJECTS</code>
	<code>CURSOR_SELECT_TIMELINE_OBJECTS_MULTI</code>	Shows up when the mode of the lasso layer was set to: <code>LassoMode.SELECT_TIMELINE_OBJECTS</code> and the SHIFT or the CTRL key is pressed.
	<code>CURSOR_SELECT_TIME_SPANS</code>	Shows up when the mode of the lasso layer was set to: <code>LassoMode.SELECT_TIME_SPANS.</code>
	<code>CURSOR_SELECT_TIME_SPANS_MULTI</code>	Shows up when the mode of the lasso layer was set to: <code>LassoMode.SELECT_TIME_SPANS</code> and the SHIFT or CTRL key is pressed.

The lasso layer retrieves all cursor icons from the *IconRegistry* via the given ID. The easiest way to replace these cursors is to register new icons by calling *IconRegistry.setIcon(IconID, Icon)*. Another option is to subclass the *LassoLayer* class and override the *createLassoCursor(LassoMode, boolean)* method. The default implementation of this method is shown here:

```
/**
 * Creates a cursor that will be used for the given lasso mode. This method
 * looks up its cursor from the {@link IconRegistry} with the following IDs:
 * <ul>
 * <li> {@link IconId#CURSOR_SELECT_TIMELINE_OBJECTS}</li>
 * <li> {@link IconId#CURSOR_SELECT_TIMELINE_OBJECTS_MULTI}</li>
 * <li> {@link IconId#CURSOR_SELECT_TIME_SPANS}</li>
 * <li> {@link IconId#CURSOR_SELECT_TIME_SPANS_MULTI}</li>
 * <li> {@link IconId#CURSOR_CREATE_RELATIONSHIPS}</li>
 * <li> {@link IconId#CURSOR_CREATE_TIMELINE_OBJECTS}</li>
 * </ul>
 *
 * @param mode
 *         the lasso mode for which to create a cursor
 * @param multi
 *         determines if the cursor is needed for a 'multi' operation
 *         (multi timeline object selection, multi time span selection)
 * @return a cursor for the given lasso mode
 * @since 1.0
 */
protected Cursor createLassoCursor(LassoMode mode, boolean multi) {
    IconId id = IconId.CURSOR_SELECT_TIMELINE_OBJECTS;
    switch (mode) {
        case SELECT_TIMELINE_OBJECTS:
```

```

        if (multi) {
            id = IconId.CURSOR_SELECT_TIMELINE_OBJECTS_MULTI;
        } else {
            id = IconId.CURSOR_SELECT_TIMELINE_OBJECTS;
        }
        break;
    case CREATE_RELATIONSHIP:
        id = IconId.CURSOR_CREATE_RELATIONSHIPS;
        break;
    case CREATE_TIMELINE_OBJECTS:
        id = IconId.CURSOR_CREATE_TIMELINE_OBJECTS;
        break;
    case SELECT_TIME_SPANS:
        if (multi) {
            id = IconId.CURSOR_SELECT_TIME_SPANS_MULTI;
        } else {
            id = IconId.CURSOR_SELECT_TIME_SPANS;
        }
        break;
    }
    ImageIcon icon = (ImageIcon) IconRegistry.getIcon(id);
    return Toolkit.getDefaultToolkit().createCustomCursor(icon.getImage(),
        new Point(15, 15), "lasso_cursor"); //$NON-NLS-1$
}

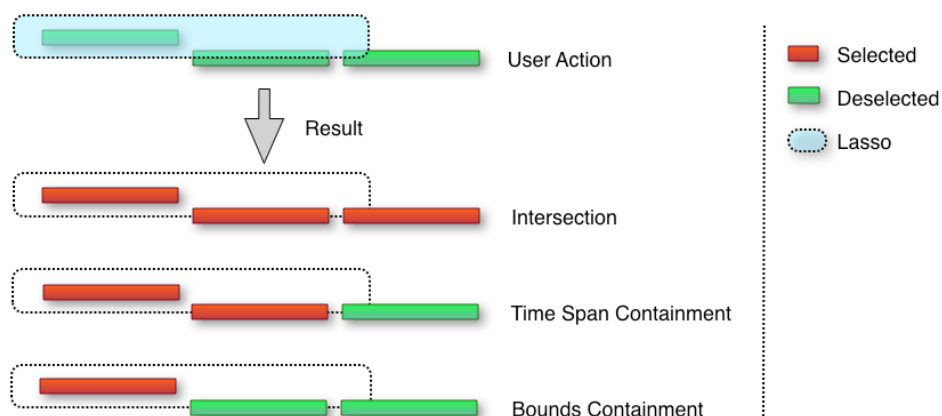
```

Selection Behaviour

The way selection works can be highly application-specific. Application developers have to ask the question: „when do I consider a timeline object to be selected by the lasso?“. **FlexGantt** assumes that there are three different answers to this question and lists them in the *SelectionBehaviour* enumerator:

1. **INTERSECTION**: a value indicating to the lasso layer that a simple intersection of the bounds of a timeline object with the bounds of the lasso is sufficient for the selection of the timeline object.
2. **TIME_SPAN_CONTAINMENT**: a value indicating to the lasso layer that the time span of a timeline object needs to be completely contained within the time span defined by the lasso in order for the timeline object to become selected.
3. **BOUNDS_CONTAINMENT**: a value indicating to the lasso layer that the bounds of a timeline object need to be completely contained within the bounds of the lasso in order for the timeline object to become selected.

The following diagram illustrates the different selection behaviours. The first row shows the lasso used by the user. The next three rows show the resulting selections as they would occur for the various selection behaviours.



Selection Behaviour

Related API:

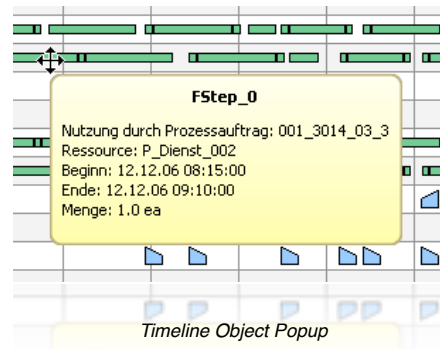
```

void LassoLayer.setSelectionBehaviour(SelectionBehaviour);
SelectionModeAction
ILayerContainerSelectionModel

```

PopupLayer

A bar underneath a timeline is a nice representation for an activity but it can only carry a limited amount of information due to space restrictions. Popups / annotations / tooltips that show up when the mouse cursor hovers over such a bar is an excellent feature that can display a wealth of additional information that the planner might need to make a scheduling decision. **FlexGantt**'s popups can be customized in the same way that any Swing component can be customized. Different renderers can be mapped to different types of popup objects. The popup objects are looked up from the *IPopupPolicy*. The image below shows the default popup renderer (*DefaultPopupRenderer*) that displays popup objects in their serialized form⁶.



The popup layer uses the *IPopupPolicy* to lookup the information shown in the popup. The following two policy methods are used for this purpose:

```
Object getPopupValue(TimelineObjectPath path, IGanttChartModel model, boolean extended);
Object getPopupTitleValue(TimelineObjectPath path, IGanttChartModel model, boolean extended);
```

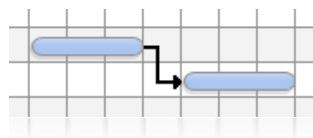
The idea behind the „extended“ flag is that the application can provide two different types of popups for the same timeline object. For example one with a higher aggregation level and one with a more detailed view on the data. The user can toggle between the two popups by pressing the SHIFT key while the popup is shown.

Related API:

```
void PopupLayer.setPopupRenderer(IPopupRenderer renderer);
IPopupRenderer
DefaultPopupRenderer
Object IPopupPolicy.getPopupValue(Object node, Object timelineObject, IGanttChartModel model);
```

RelationshipLayer

It is quite common that timeline objects have some kind of relationship with each other. Project planning software for example often defines constraints between them. Some examples for constraints are: 'start after', 'finish before', 'same start', 'same end'. **FlexGantt** can visualize them by drawing lines between them. Each application has its own way of visualizing them (different colors and / or line styles for different constraints). By plugging in a custom relationship renderer it is possible to implement any kind of relationship rendering.



A relationship between two timeline objects

Related API:

```
Collection<IRelationship> IGanttChartModel.getRelationships();
Collection<IRelationship> IGanttChartModel.getRelationships(Object timelineObject);
IRelationshipRenderer
IRelationship
IRelationshipPolicy
DefaultRelationship
```

⁶ The renderer simply calls *Object.toString()* on the popup value object and displays the returned text in a *JTextArea*.

```
DefaultRelationshipPolicy
DefaultCreateRelationshipCommand
void RelationshipLayer.setLookupStrategy(LookupStrategy strategy);
void RelationshipLayer.setRelationshipRenderer(IRelationshipRenderer);
```

RowLayer

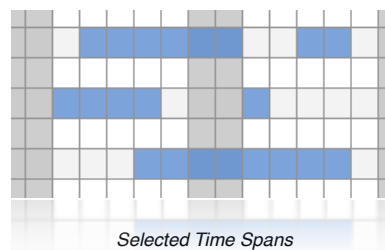
The row layer can be used to display additional information in each row by registering specialized row renderers. The renderer lookup is based on the type of the hierarchy / tree node. Row renderers accept a flag, which signals them whether the row that they are rendering is the current focus owner. This flag will only be supported by the row layer if the focus support feature is enabled.

Related API:

```
void RowLayer.setPaintingFocus(boolean);
void RowLayer.setRowRenderer(Object, IRowRenderer);
IRowRenderer
```

SelectionLayer

The selection layer is responsible for visualizing the currently selected time spans. The selection is managed by the layer container selection model (*ILayerContainerSelectionModel*). Selections are drawn as rectangles filled with a paint that can be specified. Time spans can be selected with the lasso of the *LassoLayer* when the layer's selection mode has been set to *SelectionMode.SELECT_TIME_SPANS*.

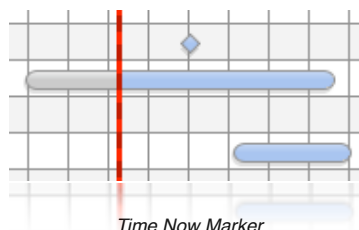


Related API:

```
void SelectionLayer.setSelectionPaint(Paint paint);
void LassoLayer.setSelectionMode(SelectionMode mode);
ILayerContainerSelectionModel LayerContainer.getSelectionModel();
```

TimeNowLayer

The time now layer draws a vertical line at the location of the 'time now', which is usually the current system time. However, a Gantt chart can choose to have its own current time. The Gantt chart can choose whether it wants to display the time now or not. An auto-scroll feature is also available, which will ensure that the time now cursor (vertical line) will always stay within the visible area, which causes an automatic scrolling of the timeline.



The „time now“ will not be updated automatically. It is up to the application to update it. **FlexGantt** provides a convenience class called *TimeNowThread*, which can be used for this purpose. The thread uses a configurable delay to periodically call *Eventline.setTimeNow(long)*. Applications need to make sure that the thread gets terminated when the Gantt chart for which it is used gets closed.

The following code fragments shows the implementation of the thread's *run()* method:

```
public void run() {
    while (running) {
        try {
            Thread.sleep(delay);
        } catch (InterruptedException e) {
            LOGGER.log(Level.WARNING, "problem in update thread", e);
        }
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                TimeZone zone = eventline.getDateline().getTimeZone();
                long millis = Calendar.getInstance(zone).getTimeInMillis();
                eventline.setTimeNow(millis);
            }
        });
    }
}
```

Related API

```
TimeNowThread
void AbstractGanttChart.setTimeNowVisible(boolean visible);
void AbstractGanttChart.setTimeNowScrolling(boolean scroll);
```

Timeline Object Layers


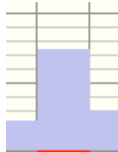
Timeline object layers are variable in number. How many of them are created depends on the Gantt chart model. An application is free to add any number of timeline object layers to the layer container. Timeline object layers are a grouping mechanism for timeline objects. An application that supports before and after analysis might split its timeline objects into two groups. The first group represents the „before“ situation, while the other group represents the „after“ situation. Putting these objects onto different layers makes it possible to show or hide them as desired. Timeline object layers support transparency. This way all objects are recognizable at the same time even though some of them might overlap others.


Timeline object layers are volatile objects. They can be created and deleted and recreated whenever changes in the Gantt chart model happen. This is the reason why the renderers and the selection models are owned by the layer container and not by the layer itself:

```
ITimelineObjectRenderer LayerContainer.getTimelineObjectRenderer(Class cl);
void LayerContainer.setTimelineObjectRenderer(Class cl, ITimelineObjectRenderer renderer);
ITimelineObjectLayerSelectionModel LayerContainer.getSelectionModel(ILayer layer);
void LayerContainer.setSelectionModel(ILayer layer, ITimelineObjectLayerSelectionModel model);
```

The primary purpose of the timeline object layer class is to draw the timeline objects. The layer class delegates this task to renderers of type *ITimelineObjectRenderer*. Implementations of this renderer interface need to return a component, which will be used like a brush to visualize one timeline object after another. Renderer components are only added temporarily to the layer **container**. They will be removed after their paint methods have been called.

FlexGantt ships with several default renderers, which are mapped to default model classes. The following table lists the renderer / model class mappings as defined by the *LayerContainer* class.

Model Class	Renderer
Object.class	DefaultTimelineObjectRenderer  <i>Timeline Object (Renderer)</i>
DefaultCapacityObject	DefaultCapacityObjectRenderer  <i>Capacity Object (Renderer)</i>

Model Class	Renderer
DefaultActivityObject	DefaultActivityObjectRenderer This renderer provides additional feedback for the „percentage complete“ value stored on the activity object.  <i>Activity Object (Renderer)</i>
DefaultEventObject	DefaultEventObjectRenderer This renderer is specialized on rendering events. An „event“ is a timeline object, which does not have a duration (end time equals start time). The renderer uses icons to represent these events.

Custom Layers

Custom layers have no predefined purpose at all. They are an extension point that application developers can use to add arbitrary visual information to the Gantt chart. A custom layer might for example draw a rectangle around timeline objects that are somehow related to each other.

A custom layer factory and a custom component factory are needed in order to add a custom layer to the layer stack of a layer container. This means that the following steps are needed:

1. Implement the *ILayerFactory* interface or subclass *DefaultLayerFactory* (highly recommended).
2. Implement the *IComponentFactory* interface or subclass *DefaultComponentFactory* (highly recommended). This component factory needs to pass the custom layer factory to the constructor of the layer container.
3. Create a Gantt chart model that adds one or more custom layers. A layer is considered a custom layer if *ILayer.isCustomLayer()* returns true.
4. Create your Gantt chart using the custom component factory. This is done by passing the factory to the constructor of the Gantt chart. The factory can not be set at a later time because it is already needed when the Gantt chart constructs its children components (timeline, tree table, layer container, ...).

The following code examples add a custom layer. The purpose of this layer is to add a watermark to the Gantt chart. Such a watermark might be used to indicate to the users that they are using a demo / trial version of a software product.

```
/**
 * Copyright 2006, 2007
 * Dirk Lemmermann Software & Consulting
 * http://www.dlsc.com
 */
package com.dlsc.flexgantt.manual;

import com.dlsc.flexgantt.model.gantt.ILayer;
import com.dlsc.flexgantt.swing.layer.AbstractCustomLayer;
import com.dlsc.flexgantt.swing.layer.DefaultLayerFactory;
import com.dlsc.flexgantt.swing.layer.LayerContainer;

/**
 * A custom layer factory that gets used instead of the default layer factory so
 * that a custom layer gets created when it finds a model layer named 'watermark'.
 */
public class WatermarkLayerFactory extends DefaultLayerFactory {

    /**
     * The singleton instance.
     */
    private static WatermarkLayerFactory instance;

    /**
     * Private constructor as part of singleton pattern implementation.
     */
}
```



```

    */
    private WatermarkLayerFactory() {
    }

    public static synchronized WatermarkLayerFactory getInstance() {
        if (instance == null) {
            instance = new WatermarkLayerFactory();
        }
        return instance;
    }

    /**
     * The default implementation of the factory method for creating custom
     * layers does nothing. In order to add custom layers it is necessary to
     * create a custom factory and then return custom layer implementations
     * whenever a model layer's 'custom layer' flag indicates that the layer
     * requires it.
     */
    public AbstractCustomLayer createCustomLayer(LayerContainer lc, ILayer layer) {
        if (layer.getName().equals("watermark")) {
            return new WatermarkLayer(lc, layer);
        } else {
            return super.createCustomLayer(lc, layer);
        }
    }
}

```

The following code is the implementation of the custom watermark layer.

```

/**
 * Copyright 2006, 2007
 * Dirk Lemmermann Software & Consulting
 * http://www.dlsc.com
 */
package com.dlsc.flexgantt.manual;

import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.Rectangle;
import java.awt.TexturePaint;
import java.awt.image.BufferedImage;
import java.net.URL;

import javax.swing.ImageIcon;

import com.dlsc.flexgantt.model.gantt.ILayer;
import com.dlsc.flexgantt.swing.layer.AbstractCustomLayer;
import com.dlsc.flexgantt.swing.layer.LayerContainer;

/**
 * The watermark layer is a custom layer implementation. It will be created and
 * added to a layer container if one of the model layers is a custom layer and
 * the layer's name equals 'watermark'.
 *
 * @author Dirk Lemmermann
 */
public class WatermarkLayer extends AbstractCustomLayer {

    /**
     * Stores the texture paint that will be drawn as tiles in the background
     * of the layer (container).
     */
    private TexturePaint texturePaint;

    /**
     * Constructs a new watermark layer.
     *
     * @param lc the layer container to which the layer will belong
     */
    public WatermarkLayer(LayerContainer lc, ILayer layer) {
        super(lc, layer);

        //
        // Load an image and create a texture paint
        // object with it. This paint object can then
        // be used in the paintLayer() method to fill
        // the background with the watermark image.
        //
        URL url = getClass().getResource("watermark.png");
    }
}

```

```

        ImageIcon icon = new ImageIcon(url);
        Image texture = icon.getImage();
        BufferedImage buffer = new BufferedImage(texture
            .getWidth(layerContainer), texture
            .getHeight(layerContainer), BufferedImage.TYPE_INT_RGB);
        Graphics2D bg = buffer.createGraphics();
        bg.drawImage(texture, 0, 0, layerContainer);
        texturePaint = new TexturePaint(buffer, new Rectangle(0, 0, buffer
            .getWidth(), buffer.getHeight()));
    }

    protected void paintLayer(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        Rectangle clip = g.getClipBounds();
        g2d.setPaint(texturePaint);
        g2d.fillRect(clip.x, clip.y, clip.width, clip.height);
    }
}

```

A Gantt chart model that wants the watermark layer to be shown then needs to add a custom model layer with the name 'watermark'.

```

/**
 * Copyright 2006, 2007
 * Dirk Lemmermann Software & Consulting
 * http://www.dlsc.com
 */
package com.dlsc.flexgantt.manual;

import com.dlsc.flexgantt.model.gantt.DefaultGanttChartModel;
import com.dlsc.flexgantt.model.gantt.DefaultGanttChartNode;
import com.dlsc.flexgantt.model.gantt.Layer;

/**
 * A specialization of the default Gantt chart model. It adds a custom layer
 * named 'watermark'. This model needs to get used in combination with a layer
 * factory that knows how to create a watermark layer.
 *
 * @author Dirk Lemmermann
 */
public class WatermarkGanttChartModel extends DefaultGanttChartModel {

    /**
     * Constructs a new Gantt chart model.
     */
    public WatermarkGanttChartModel() {
        super(new DefaultGanttChartNode());
        Layer layer = new Layer("watermark");
        layer.setCustomLayer(true);
        addLayer(layer);
    }
}

```

The last thing needed is a component factory that creates a layer container with the watermark layer factory. This company factory then needs to be passed to the constructor of the Gantt chart.

```

/**
 * Copyright 2006, 2007
 * Dirk Lemmermann Software & Consulting
 * http://www.dlsc.com
 */
package com.dlsc.flexgantt.manual;

import com.dlsc.flexgantt.model.gantt.IGanttChartModel;
import com.dlsc.flexgantt.swing.AbstractGanttChart;
import com.dlsc.flexgantt.swing.DefaultComponentFactory;
import com.dlsc.flexgantt.swing.layer.LayerContainer;
import com.dlsc.flexgantt.swing.treetable.TreeTable;

public class WatermarkComponentFactory extends DefaultComponentFactory {

    /**
     * Creates a layer container that uses the watermark layer factory.
     */
    public LayerContainer createLayerContainer(AbstractGanttChart gc,
        TreeTable table, IGanttChartModel model) {
        return new LayerContainer(gc, model, table, WatermarkLayerFactory.getInstance());
    }
}

```

Frequently Asked Questions (FAQ)

How can I change the transparency of a layer?

All layer types (system, custom, timeline object) are extensions of the *AbstractLayer* class. This class defines a method called *setAlpha(float)*. It is up to the layer implementation to decide how to use this alpha value but most actually do apply the value to all objects drawn by them. One exception is the popup layer in combination with the default popup renderer class. This renderer will only apply the alpha value on the background of the popup. The text will not use transparency. This behavior can of course be changed by registering a different renderer on the layer.

How can the layer transparency be updated in real-time when using the slider in the layer selector?

The transparency slider in the layer selector does not apply its current value immediately to the layer's alpha attribute. This was done for performance reasons. Some layers show a lot of data. If they are repainted every single time the slider's value changes then the user would notice a long delay between the old and the new position of the slider thumb. The slider would become very hard to use. However, if the amount of data is small then it makes perfect sense to change this behavior. **FlexGantt** provides the following method in the class *LayerPalette*.

```
void LayerPalette.setRepaintingImmediately(boolean).
```

The default is false. If set to true a repaint will occur on the layers immediately when the slider value changes.

In order to call this method one will need to get access to the palette first. This is somewhat complicated. The palette is part of the *LayerSelector*, which gets created by an instance of *ISelectorFactory*. What you need to do is to subclass *TimeGranularitySelectorFactory* and override the following method:

```
createSelector(AbstractGanttChart gc, JComponent parentComponent, SelectorID id)
```

Inside this method one needs to call *super.createSelector(...)*. When the event ID equals *LAYERS* then the following can be called to retrieve the content of the selector:

```
JComponent content = Selector.getContentComponent()
```

This method returns a panel. Each child in this returned panel is an instance of *LayerPalette* (remember the *DualGanttChart* needs two layer palettes).

To make your Gantt chart use your selector factory override the following protected method:

```
ISelectorFactory AbstractGanttChart.getSelectorFactory()
```

How can I use different colors for different layers?

Colors are not used by or assigned to layers. Colors come in play when writing timeline object renderers (*ITimelineObjectRenderer*). One has to implement custom renderers that uses some kind of attribute stored on the application's Gantt chart model to select a color.

```
// Pseudo Code
public class MyRenderer extends JPanel implements ITimelineObjectRenderer {

    Component getTimelineObjectRendererComponent(TimelineObjectLayer layer,
        Object treeNode, Object timelineObject, boolean selected,
        boolean focus, boolean highlighted, int row) {
        MyModelObject model = (MyModelObject) timelineObject;
        Object attribute = model.getAttribute();
        if (attribute.equals(...)) {
            setBackground(Color.ORANGE);
        } else if (attribute.equals(...)) {
            setBackground(Color.BLUE);
        }
        return this;
    }
}
```

Your renderer then needs to be registered on the layer container:

```
myGantt.getLayerContainer().setTimelineObjectRenderer(MyModelObject.class, new MyRenderer());
```

Is it possible to remove the layer UI / layer palette / layer selector?

The control on which the user clicks to bring up the layer selector is simply a *JLabel* instance inside a panel. This panel is of type *UtilityControlPanel*. This panel is shown in the lower left corner of the scrollpane that wraps the layer container. This scrollpane is of type *LayerContainerScrollPane*. The following code fragment sets the label to invisible so that the user can no longer click on it:

```
LayerContainerScrollPane pane = ganttChart.getLayerContainerScrollPane();
UtilityControlPanel ucp = (UtilityControlPanel) pane.getCorner(JScrollPane.LOWER_LEFT);
JLabel layerLabel = ucp.getLayerLabel();
layerLabel.setVisible(false);
```

How many layers can a Gantt chart display?

There is no restriction on the number of layers that can be added to a Gantt chart model. However, the number of layers has a direct impact on the drawing performance. The layer container will always loop over all layers and perform at least one check on them to see if they are visible (*AbstractLayer.isVisible()*). If a layer is in fact visible, then the layer container will call the *paintLayer(Graphics g)* method on it. It depends on the amount of work done in these methods whether the Gantt chart's performance is still acceptable.