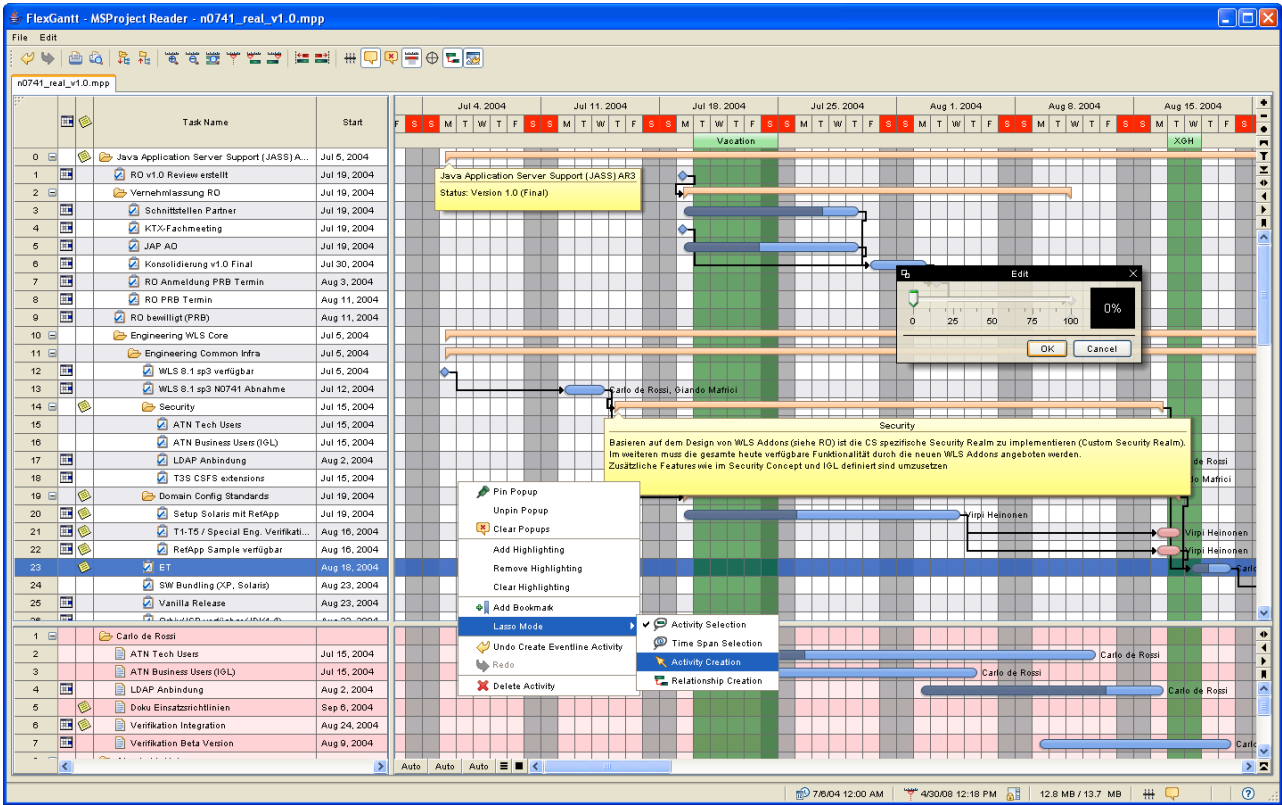


FlexGantt Release 1.x

Komponenten



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

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

Inhaltsverzeichnis

Einleitung	1
Component Factory	1
AbstractGanttChart	1
Modelle	2
Tabellenspalten	2
Systemmeldungen	3
Statusmeldungen	4
Toolbar	4
Commands	4
Gitterlinien	5
Drucken	5
Visuelle Attribute	6
Komponenten	7
Navigation	8
Hilfsfunktionen	8
GanttChart	9
DualGanttChart	10
SimpleGanttChart	11
TreeTable	11
Selektieren von Zeilen	11
Editieren von Werten in Zellen	12
Einrücken von Knoten	13
Knoten Erzeugen und Löschen	13
Zeilenhöhen	14
Tabellenspalten	14
Listener und Events	15
Drag & Drop	15
Fokus	16
Baumknoten und Tree Paths	16
Visuelle Attribute	16
Zeichnen	17
Hilfsmethoden	19
Policies	20
Interne Datenstrukturen	21
TreeTableHeader	21
Model	21
Renderer	21
Zugriff auf Tabellenspalten	22
Sortierung von Tabellenspalten	22
Anpassen von Tabellenspalten	22
Visuelle Attribute	23
Popup Menü	23
Timeline	23
Dateline	24
Model	25
Renderer	26
Zoom In / Out	26
Policies	28
Navigation	28
Visuelle Attribute	28
Popup Menü	28
Eventline	29
Model	29
Selection Model	29
Eventline Objekte Löschen	29
Marker	29
Policies	29
Popup Menü	29
LayerContainer	31

Layer Factory	31
System Layers	31
Timeline Object Layers	36
Custom Layers	36
Selection Models	37
Timeline Objects Hervorheben (Highlighting)	37
Timeline Object Status	37
Popup Menü	37
Policies	37
Hilfsmethoden	37
NavigationControlPanel	37
Ausblenden von Controls	37
Änderung des Verhaltens einer Control	37
UtilityControlPanel	38
Statusbar	38
GanttChartToolBar	40
Row Headers	41
Menü Provider	42
Grid Components	43
Grid Control	43
Automatisches Gitter	44
Policies	44
Listener und Events	44
Zugriff	45
MultiGanttChartContainer	45

1. Einleitung

Dieses Dokument beschäftigt sich ausführlich mit den wichtigsten Komponenten aus denen sich ein **FlexGantt**-basierter Gantt Chart zusammensetzt. Grob gesprochen besteht ein Gantt Chart aus einer linken und einer rechten Seite, welche für unterschiedliche Aufgaben genutzt werden. Die linke Seite zeigt Informationen tabellarisch an, während die rechte Seite diese graphisch darstellt. Beide Seiten bestehen aus einer ganzen Reihe von Swing Komponenten. Hierzu gehören die Tabelle, die Zeitleiste, der Ebenen Container, usw. Wie diese Komponenten erzeugt und individuell konfiguriert werden können wird in diesem Dokument detailliert besprochen.

2. Component Factory

In einem Framework ist es wichtig, dass dem Entwickler grösstmögliche Flexibilität geboten wird, wenn es darum geht die Objekte und das Verhalten des Frameworks zu erweitern oder zu verändern. Bei Frameworks für Benutzeroberflächen, welche komplexe Komponenten beinhalten, ist es daher absolut notwendig, daß der Entwickler Kontrolle darüber bekommt wie diese komplexen Komponenten ihre Einzelteile erzeugen. Im allgemeinen wird diese Anforderung durch das so genannte *Factory Pattern* umgesetzt. Auch **FlexGantt** macht hierbei keine Ausnahme und bedient sich eines *IComponentFactory* genannten Interfaces, um solche Dinge wie die Tabelle und die Zeitleiste zu erzeugen. Eine Default Implementierung namens *DefaultComponentFactory* ist bereits Teil des Frameworks. Beim Erzeugen einer Gantt Chart Instanz muss dem Konstruktor immer eine Komponentenfabrik mit übergeben werden. Es gibt mehrere Gantt Chart Konstruktoren, welche keine solche Komponentenfabrik erwarten, weil sie einfach die Default Implementierung nutzen.

Das Factory Interface sieht wie folgt aus:

```
public interface IComponentFactory {

    Timeline createTimeline(AbstractGanttChart gc);
    Dateline createDateline(Timeline timeline);
    Eventline createEventline(Timeline timeline, Dateline dateline);
    TreeTable createTreeTable(AbstractGanttChart gc, ITreeTableModel model);
    TreeTableHeader createTreeTableHeader(AbstractGanttChart gc);
    TreeTableRowHeader createTreeTableRowHeader(TreeTable table);
    LayerContainer createLayerContainer(AbstractGanttChart gc,
        TreeTable table, IGanttChartModel model);
    LayerContainerRowHeader createLayerContainerRowHeader(LayerContainer lc);

    JComponent createTreeTableCorner(TreeTable table, String corner);
    JComponent createLayerContainerCorner(LayerContainer lc, String corner);
}
```

Man kann leicht erkennen, daß die meisten Factorymethoden sehr konkret sind, d.h. sie erwarten einen ganz bestimmten Komponententypen als Rückgabewert. Hierzu gehören: Timeline, Dateline, Eventline, TreeTable, TreeTableHeader, TreeTableRowHeader, LayerContainer, LayerContainerRowHeader

Zusätzlich gibt es noch die beiden Methoden *createTreeTableCorner()* und *createLayerContainerCorner()*, die ein wenig abstrakter sind und beliebige *JComponent* Instanzen erzeugen. Diese Komponenten werden benötigt, um die verschiedenen „Ecken“ im Gantt Chart zu füllen. Auf der linken Gantt Chart Seite gibt es eine Ecke in den linken oberen und unteren Ecken der *TreeTable*. Auf der rechten Seite gibt es Ecken oben und unten rechts und links vom *LayerContainer*:

- *TreeTable* oben links: ein Panel, auf das der User klicken kann, um einen Selektor zu öffnen. Dieser Selektor erlaubt es dem Benutzer Tabellenspalten ein- und auszublenden bzw. deren Reihenfolge zu ändern.
- *TreeTable* unten links: ein leeres ungenutztes Panel
- *LayerContainer* oben links: diese Ecke ist nur sichtbar, wenn der LayerContainer einen LayerContainerRowHeader anzeigt. In der Ecke wird ein Füllobjekt angezeigt, welches keine weitere Funktionalität hat.
- *LayerContainer* oben rechts: das sogenannte *NavigationControlPanel*, welches dem Benutzer verschiedene kleine Kontrollelemente zur Navigation innerhalb des Gantt Charts zur Verfügung stellt.
- *LayerContainer* unten rechts: ein Kontrollelement, welches es dem Benutzer erlaubt den Gantt Chart zu splitten (nur beim *DualGanttChart*).
- *LayerContainer* unten links: eine Sammlung von sehr unterschiedlichen Selektoren. Diese dienen dazu das Drag & Drop Gitter einzustellen (*GridSelector*), Ebenen ein- und auszublenden (*LayerSelector*), oder zur Navigation, um schnell zu einem ganz genauen Punkt innerhalb des Gantt charts zu gelangen (*OverviewSelector*).

3. AbstractGanttChart

FlexGantt wird mit mehreren Gantt Chart Klassen ausgeliefert. Allen gemein ist die Tatsache, dass sie Unterklassen von *AbstractGanttChart* sind. Zu den Aufgaben dieser Klasse gehört:

- Verwaltung von Datenmodellen
- Verwaltung von Tabellenspalten
- Anzeige von System- und Statusmeldungen
- Ausführung von Kommandos

- Drucken von Plänen
- Verwaltung einer grossen Anzahl an visuellen Attributen
- Unterstützung des Users bei der Navigation
- Ermöglichen des Zugriffs auf Teilkomponenten
- Unterstützung der mitgelieferten Toolbar
- Bereitstellen einer Vielzahl an Hilfsfunktionen

Modelle

Innerhalb eines mit **FlexGantt** erstellten Gantt Charts kommen eine Vielzahl an Modellen zum Einsatz. Vier dieser Modelle findet man in der hier beschriebenen Klasse **AbstractGanttChart**. Hierzu gehört das primäre Gantt Chart Model, welches die eigentlichen Gantt Chart Daten zur Verfügung stellt, das Paging Model mit dessen Hilfe man zwischen verschiedenen Horizonten hin- und herspringen kann, das Column Model, welches die Spalten der Tabelle definiert und das Calendar Model, mit dessen Hilfe man Wochenenden und Feiertage festlegt.

Model	Beschreibung
IGanttChartModel	Das Gantt Chart Model ist eine Erweiterung des <i>ITreeTableModel</i> und liefert sowohl die Daten für die Tabelle, als auch die Daten für die zeitbezogenen Aktivitäten, welche im rechten Teil des Gantt Charts angezeigt werden.
IPagingModel	Mithilfe des Paging Models kann eine Anwendung eine Menge an Zeiträumen definieren aus welcher der User abwechselnd einen auswählen kann. Dieser Zeitraum wird dann als Horizont auf der Zeitleiste gesetzt. Das Paging Model ermöglicht somit ein Zerstückeln des gesamten Zeitraums und erlaubt so auch die Handhabung von sehr grossen Datenmengen, da nun immer nur ein Bruchteil davon angezeigt werden muss.
IColumnModel	Das Column Model dient der Tabelle auf der linken Seite des Gantt Charts als Informationsquelle, um herauszufinden welche Spalten es darstellen soll. Alle Spalten im Model sind immer sichtbar. Um optionale Spalten zu definieren, müssen diese dem Gantt Chart direkt hinzugefügt werden.
ICalendarModel	Kalender Modelle werden für zwei Zwecke genutzt. Zum einen kann man mit ihrer Hilfe definieren an welchen Tagen Wochenenden und / oder Ferien liegen. Zum anderen dienen sie dazu solche Dinge wie Schichtpläne zu berechnen. Kalender Modelle müssen sehr effizient implementiert sein, damit sie die Rendering Performance nicht unnötig strapazieren.

Eine genaue Beschreibung der verschiedenen Modelle, ihrer Aufgaben, und wie man sie am besten einsetzt befindet sich in dem Dokument „FlexGantt - Modelle“.

Die für die Modelle relevanten Methoden heissen:

```
void AbstractGanttChart.setModel(IGanttChartModel)
IGanttChartModel AbstractGanttChart.getModel()
void AbstractGanttChart.setPagingModel(IPagingModel)
IPagingModel AbstractGanttChart.getPagingModel()
void AbstractGanttChart.setColumnModel(IColumnModel)
IColumnModel AbstractGanttChart.getColumnModel()
void AbstractGanttChart.setCalendarModel(ICalendarModel)
ICalendarModel AbstractGanttChart.getCalendarModel()
```

Tabellenspalten

Die linke Seite eines Gantt Charts besteht aus einer sogenannten TreeTable, d.h. aus der Kombination eines Baumes mit einer Tabelle. Eine Besonderheit von **FlexGantt** ist die starke Unterscheidung zwischen der sogenannten „Key“ Spalte (KeyColumn) und den restlichen Spalten (TreeTableColumn). Die Key Spalte wird benutzt, um die hierarchische Struktur des Gantt Chart Models darzustellen, während die restlichen Spalten die einzelnen Werte der Baumknoten anzeigen. Die Klasse **AbstractGanttChart** bietet mehrere Methoden an, die das einfache Arbeiten mit der Key Spalte ermöglichen sollen. Dies sind:

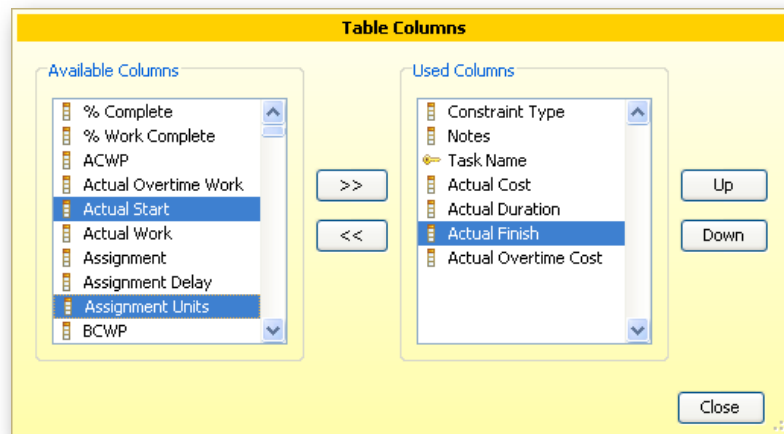
```
KeyColumn AbstractGanttChart.getKeyColumn()
void AbstractGanttChart.setKeyColumnPosition(int)
int AbstractGanttChart.getKeyColumnPosition()
```

Es ist wichtig anzumerken, daß die Methode `getKeyColumn()` immer funktioniert, unabhängig davon, ob die Key Spalte gerade sichtbar oder unsichtbar ist. Dies ist eine extrem wichtige Voraussetzung für viele Funktionen innerhalb des Gantt Charts. So benutzt zum Beispiel das Drag & Drop der Tabelle die Key Spalte, um eine Grafik zu erzeugen, die während des Drags angezeigt wird.

AbstractGanttChart bietet auch Methoden an, um auf alle Spalten zuzugreifen. Diese lauten:

```
int AbstractGanttChart.getColumnCount()
TreeTableColumn AbstractGanttChart.getColumn(int)
```

Viele Anwendungen benötigen Spalten, die sich links von der Key Spalte befinden. Zu diesem Zweck kann die Methode *setKeyColumnPosition(int)* aufgerufen werden. Wird die Position auf 2 gesetzt sorgt dies dafür, daß zwei andere Spalten links von der Key Spalte platziert werden. Zur Laufzeit hat der Benutzer selbst die Möglichkeit diese Position zu definieren. Dies geschieht im Selektor für Tabellenspalten, welcher durch einen Klick auf die linke obere Ecke der Tabelle angezeigt wird.



Selektor für Tabellenspalten

Im gleichen Selektor hat der Benutzer auch die Möglichkeit Spalten aus der Tabelle zu entfernen oder Spalten der Tabelle hinzuzufügen. Die „verfügbaren“ Spalten sind nicht Teil des Column Models sondern werden separat verwaltet von *AbstractGanttChart*. Die entsprechenden Methoden lauten:

```
void AbstractGanttChart.setAvailableColumns(Collection<TreeTableColumn>)
void AbstractGanttChart.addAvailableColumn(TreeTableColumn)
void AbstractGanttChart.removeAvailableColumn(TreeTableColumn)
Collection<TreeTableColumn> AbstractGanttChart.getAvailableColumns()
```

Ein weiteres Feature der Tabellenspalten in **FlexGantt** ist ihre Fähigkeit zum sortieren. Der Benutzer kann durch klicken auf Spaltenköpfe die Werte in der Tabelle nach einer oder mehrerer Spalten sortieren lassen (mittels der STRG / CTRL Taste). Applikationen, die ein Interesse daran haben informiert zu werden, wenn dies geschieht, können sich als Listener an den Gantt Chart anhängen. Die folgenden Methoden in *AbstractGanttChart* sind relevant bezüglich der Sortierfunktionalität:

```
void AbstractGanttChart.addSortingListener(ISortingListener)
void AbstractGanttChart.removeSortingListener(ISortingListener)
void AbstractGanttChart.sortTables(int[], boolean[])
```

Systemmeldungen

Die Anzeige von Meldungen ist eine Standardanforderung vieler Applikationen. Für gewöhnlich unterscheidet man hier zwischen Fehler-, Warnungs-, und Informationsmeldungen, die die Anwendung dem Benutzer anzeigen möchte. In **FlexGantt** werden alle Meldungen an Methoden in *AbstractGanttChart* delegiert. Dies ermöglicht es auf einfache Art und Weise die Darstellung der Meldungen zu ändern. **FlexGantt** nutzt die *JOptionPane* Klasse, um Meldungen anzuzeigen. Anwendungen, die einen **FlexGantt** Gantt Chart in einer Umgebung einbetten (zum Beispiel in einer Eclipse RCP oder in Netbeans), lenken die Meldungen oft um, so dass das Erscheinungsbild der Meldung konsistent ist mit dem Rest der Anwendung. Die für Meldungen relevanten Methoden lauten:

```
showMessage(String)
showMessage(String, MessageTypeId)
showMessage(String, MessageTypeId, Object)
```

Der Enumerator *MessageTypeId* dient dabei der Unterscheidung zwischen den verschiedenen Arten von Meldungen. Das generische Objekt, welches der letzten Methode als drittes Argument übergeben wird, beinhaltet für gewöhnlich eine Exception. Die Default Implementierung dieser Methode macht bewusst keinen Gebrauch von diesem Object, d.h. es wird nicht angezeigt. Exceptions sind sehr technisch und für den normalen User nicht nutzbar.

Statusmeldungen

Der von einem Gantt Chart angezeigte Plan beinhaltet häufig Inkonsistenzen, d.h. bestimmte Bedingungen, welche vom Planer zu berücksichtigen sind, werden nicht eingehalten. Solche „Constraint Violations“ werden häufig durch das Einfärben von Balken oder Linien sichtbar gemacht. Das Problem hierbei ist, dass nicht immer der gesamte Plan im sichtbaren Bereich ist, d.h. der Benutzer bekommt die Probleme erst gar nicht angezeigt. Abhilfe schaffen hier die Status Meldungen. Dies sind Meldungen, die auf einen Baumknoten oder einen Balken verweisen können. In der Benutzeroberfläche kann dieser Verweis genutzt werden, um den entsprechenden Knoten oder Balken sichtbar zu machen. Status Meldungen müssen das `IMessage` Interface implementieren. Bereits mitgeliefert werden die Klassen `Message`, `TimelineObjectPathMessage`, `TreePathMessage`.

`AbstractGanttChart` definiert mehrere Methoden, um Status Meldungen zu verwalten (anhängen, entfernen, abfragen).

```
addMessage (IMessage)
removeMessage (IMessage)
clearMessages ()
getMessages ()
```

Um alle Meldungen sichtbar zu machen kann die folgende Methode aufgerufen werden. Diese zeigt dann einen Dialog vom Typ `MessageDialog` mit sämtlichen Meldungen an. Die Meldungen werden in einer speziellen Tabelle vom Typ `MessageTable` aufgelistet.

```
showMessages ()
```

Wie bereits erwähnt können Meldungen auf Knoten oder Balken verweisen. Ist dies der Fall, dann kann durch den Aufruf der nächsten Methode der Benutzer zum Ursprung der Meldung gelangen. Handelt es sich um eine `TreePathMessage`, dann wird sichergestellt, dass der entsprechende Knoten im Baum sichtbar wird. Hierzu werden gegebenenfalls alle Elternknoten geöffnet und die Tabelle vertikal so verschoben, dass die Zeile in der sich der Knoten befindet sichtbar wird. Ist die Meldung vom Typ `TimelineObjectPathMessage`, dann wird zusätzlich auch noch der grafische Teil des Gantt Charts verschoben, damit der Balken auf den die Meldung verweist sichtbar wird.

```
showMessageContext (IMessage)
```

Applikationen, die sich dafür interessieren, ob Meldungen hinzugefügt wurden oder wieder entfernt wurden, können sich als `IMessageListener` an den Gantt Chart anhängen. Sie erhalten dann die entsprechenden Events und können darauf reagieren.

```
addMessageListener (IMessageListener)
removeMessageListener (IMessageListener)
```

Toolbar

In den allermeisten Fällen werden Gantt Charts in Verbindung mit einem oder mehreren Toolbars eingesetzt. Solche Toolbars sind häufig sehr komplex und spezifisch für die jeweilige Anwendung. Da dies den Rahmen des Frameworks sprengen würde, bietet **FlexGantt** nur wenig direkte Unterstützung für Toolbars. Die mitgelieferte Klasse `GanttChartToolBar` dient lediglich dem Zweck möglichst schnell zu einem Prototypen zu kommen und definiert dafür einen festen Satz an Buttons, aus welchen eine Applikation die für sie wichtigen auswählen kann. Wie dies gemacht wird kann im Kapitel über die Toolbar nachgelesen werden.



Toolbar

Zusätzlich können aber auch noch beliebige Toolbar Actions im Gantt Chart registriert werden. Diese werden von `GanttChartToolBar` abgerufen und dynamisch hinzugefügt. Umgesetzt werden diese Actions immer als Push Button (nie als Toggle Button).

```
addToolBarAction (Action)
removeToolBarAction (Action)
clearToolBarActions ()
getToolBarActions ()
```

Commands

Wenn man ein wenig abstrahiert, dann kann man einen Gantt Chart als einen Editor ansehen. Der Plan ist das Dokument, in welchem Aktivitäten angelegt, verändert, und auch gelöscht werden. Editoren unterstützen dabei in der Regel automatisch Funktionen wie Cut, Copy, und Paste. Diese Funktionen sind für gewöhnlich als sogenannte Commands implementiert. Ein Command kann ausgeführt und rückgängig gemacht werden (Undo). Dieses für Editoren typische Verhalten wird auch von **FlexGantt** unterstützt: Commands werden von einem CommandStack ausgeführt (execute) und

rückgängig gemacht (undo). Sogenannte Command Interceptors können eingesetzt werden, um Commands vor ihrer Ausführung „abzufangen“. Mehr Informationen zum Thema Commands können im Dokument „FlexGantt - Commands und Policies“ gefunden werden.

AbstractGanttChart definiert die folgenden drei Methoden, um Commands auszuführen bzw. rückgängig zu machen. Nach ihrem Aufruf delegieren alle drei Methoden zum Command Stack.

```
commandExecute (ICommand)
commandRedo ()
commandUndo ()
```

Jeder Gantt Chart besitzt seinen eigenen Command Stack. Dies ist in der Regel eine Instanz vom Typ *DefaultCommandStack*. Falls nötig kann der Stack aber auch durch eine eigene Implementierung ausgetauscht werden.

```
setCommandStack (ICommandStack)
getCommandStack ()
```

Je nach Art des auszuführenden Commands kann es nötig sein, dass die Applikation vom Benutzer eine Bestätigung benötigt. Zum Beispiel dann wenn ein Command eine Aktivität löschen möchte. In diesem Fall würde man erwarten, dass ein Dialog auftaucht in dem die Sicherheitsabfrage „Wollen Sie diese Aktivität wirklich löschen“ gestellt wird. Erst wenn der Benutzer bestätigt soll das Command wirklich ausgeführt werden. Diese Art Funktionalität ist mit den Command Interceptors umsetzbar. Registriert und abgerufen werden diese beim AbstractGanttChart mittels der folgenden Methoden:

```
setCommandInterceptor (Class<? extends ICommand>, ICommandInterceptor)
getCommandInterceptor (Class)
```

Commands haben es an sich, daß sie je nach Einsatzgebiet mehr oder lange laufen können. Commands können zum Beispiel dafür eingesetzt werden serverseitige Informationen in einer Datenbank oder einem Application Server abzugleichen mit den Änderungen, die vom Benutzer im Gantt Chart gemacht wurden. Je nach Erreichbarkeit und Belastung kann das Command dann in Bruchteilen von Sekunden oder erst nach Minuten beendet werden. Von *AbstractGanttChart* wird eine Fortschrittsanzeige („Progress Bar“) zur Verfügung gestellt, um dem Benutzer ein Gefühl dafür zu geben wie weit ein Command schon fortgeschritten ist bzw. wie lange es noch laufen wird. Sichtbar gemacht wird die Fortschrittsanzeige allerdings erst dann wenn am bisherigen Fortschritt erkennbar wird, dass es sich überhaupt lohnt die Anzeige sichtbar zu machen. Die hierfür notwendige Logik wird von *IProgressMonitor* zur Verfügung gestellt. AbstractGanttChart benutzt eine Factory, um diese Monitore zu erzeugen. Genau wie der Command Stack ist auch die Factory austauschbar.

```
setProgressMonitorFactory (IProgressMonitorFactory)
getProgressMonitorFactory ()
```

Die Default Factory erzeugt Instanzen vom Typ *GanttChartProgressMonitor*, welche als Wrapper funktionieren und ihre Aufrufe an Instanzen vom Typ *javax.swing.ProgressMonitor* delegieren.

Gitterlinien

Zur besseren Orientierung innerhalb eines Gantt Charts sind Gitterlinien unterhalb der Zeitleiste unbedingt notwendig. **FlexGantt** unterstützt dabei verschiedene Modi:

1. Keine Gitterlinien (GridLineMode.NO_GRID_LINES)
2. Grobes Gitter (GridLineMode.MAJOR_GRID_LINES)
3. Feines Gitter (GridLineMode.MINOR_GRID_LINES)
4. Kombiniertes Gitter, feine und grobe Linien gleichzeitig (GridLineMode.COMBINED_GRID_LINES)

Gesetzt werden können die unterschiedlichen Modi direkt auf AbstractGanttChart durch folgende Methoden:

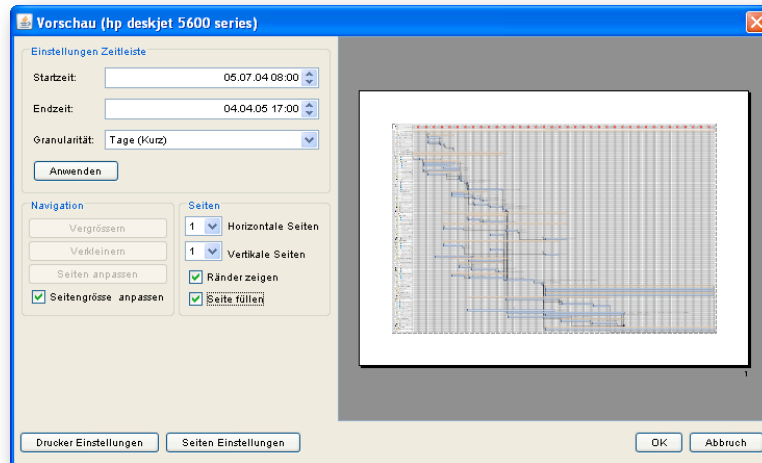
```
setGridLineMode (GridLineMode mode);
getGridLineMode ()
```

Drucken

Das Drucken von Plänen liegt ebenfalls im Aufgabenbereich von AbstractGanttChart. Um einen Druckauftrag zu starten ist der Aufruf von lediglich einer der beiden Methoden unten notwendig.

```
print (boolean)
print (boolean, ITimeSpan)
```

Der boolean Parameter bestimmt dabei, ob eine Druckvorschau angezeigt wird. Die Zeitspanne bestimmt welcher zeitliche Bereich zum Ausdruck kommt.



Druckvorschau

Visuelle Attribute

Von *AbstractGanttChart* werden eine Vielzahl von visuellen Attributen definiert, obwohl diese Attribute von Unterkomponenten wie z.B. der *TreeTable* oder dem *LayerContainer* benutzt werden. Dies liegt daran, daß ein Gantt Chart manchmal aus mehreren Tabellen und mehreren Layer Containern bestehen kann. In diesem Fall sollen manche Attribute möglichst gemeinsam gesetzt werden. Ein gutes Beispiel ist das Crosshair Feature. Ist dieses eingeschaltet bekommt der Benutzer ein Fadenkreuz im *LayerContainer* angezeigt. Im *DualGanttChart* soll dies natürlich sowohl für den oberen als auch den unteren Gantt Chart passieren.

Die folgende Tabelle listet jene Methoden auf, die im Zusammenhang mit visuellen Attributen genutzt werden können. Die Setter Methoden (setXYZ) lösen alle ein *java.beans.PropertyChangeEvent* aus. Die Property Namen sind allesamt als Konstanten in *AbstractGanttChart* definiert. Zum Beispiel:

```
/**
 * Constant used for those property change events that get fired when the
 * Gantt chart model changes.
 *
 * @see #setModel(IGanttChartModel)
 * @since 1.0
 */
public static final String PROPERTY_MODEL = "model";
```

Methode	Beschreibung
setControlsBackground(Color) getControlsBackground() setControlsForeground(Color) getControlsForeground()	Erlaubt das einfache Setzen der Vorder- und Hintergrundfarbe jener Unterkomponenten, welche den „Rahmen“ des Gantt Charts bilden. Hierzu gehören der <i>TreeTableHeader</i> , der <i>TreeTableRowHeader</i> , die <i>Dateline</i> , die <i>Eventline</i> , der <i>LayerContainerRowHeader</i> , und die verschiedenen kleineren Elemente zur Navigation, für das Paging, für die Ebeneneinstellungen, etc....
setTreeTableRowHeaderWidth(int) getTreeTableRowHeaderWidth() ; setLayerContainerRowHeaderWidth(int) getLayerContainerRowHeaderWidth()	Setzt die Breite der beiden verschiedenen Row Header. Sowohl die <i>TreeTable</i> als auch der <i>LayerContainer</i> besitzen einen solchen Header auf ihrer linken Seite. Die Tabelle nutzt diesen Header, um Zeilennummern und Expand / Collapse Icons darzustellen. Der <i>LayerContainer</i> nutzt den Header, um eine Legende für die von Ressourcen benötigten Kapazitätslinien anzuzeigen.
setCalendarVisible(boolean) isCalendarVisible()	Bestimmt, ob der <i>CalendarLayer</i> sichtbar ist oder nicht. Dieser Layer zeichnet u.a. graue Hintergründe für Wochenenden.

Methode	Beschreibung
<code>isCrosshairVisible()</code> <code>setCrosshairVisible(boolean)</code>	Bestimmt, ob der <i>CrosshairLayer</i> sichtbar ist oder nicht. Dieser Layer zeichnet ein Fadenkreuz mit Zusatzinformationen an der aktuellen Mouse Cursor Position.
<code>setLabelsVisible(boolean)</code> <code>isLabelsVisible()</code>	Bestimmt, ob der <i>LabelLayer</i> sichtbar ist oder nicht. Dieser Layer schreibt Texte an der rechten Seite von den Balken.
<code>setPopupVisible(boolean)</code> <code>isPopupVisible()</code>	Bestimmt, ob der <i>PopupLayer</i> sichtbar ist oder nicht. Dieser Layer zeigt kleine gelbe Popup Fenster mit detaillierten Informationen über die Balken an der aktuellen Mouse Cursor Position an.
<code>setRelationsVisible(boolean)</code> <code>isRelationsVisible()</code>	Bestimmt, ob der <i>RelationshipLayer</i> sichtbar ist oder nicht. Dieser Layer zeichnet die Linien zwischen Balken, welche Beziehungen ausdrücken sollen.
<code>setRowLayerVisible(boolean)</code> <code>isRowLayerVisible()</code>	Bestimmt, ob der <i>RowLayer</i> sichtbar ist oder nicht. Dieser Layer kann beliebige Dinge im Hintergrund der einzelnen Zeilen zeichnen.
<code>setTimeNowScrolling(boolean)</code> <code>isTimeNowScrolling()</code>	Bestimmt, ob der Gantt Chart automatisch mit der aktuellen Zeit mitscrollen soll. Falls dieses Feature eingeschaltet ist, wird der horizontale Scrollbalken ausgeschaltet und die „Time Now“ Linie bleibt zentriert in der Mitte des Gantt Charts stehen. Der Gantt scrollt automatisch nach rechts.
<code>setTimeNowVisible(boolean)</code> <code>isTimeNowVisible()</code>	Bestimmt, ob der <i>TimeNowLayer</i> sichtbar ist oder nicht. Dieser Layer hat nur eine Aufgabe, nämlich das Zeichnen einer vertikalen Linie an der Position der aktuellen Zeit („Time Now“).
<code>setVerticalLinesOnTop(boolean)</code> <code>isVerticalLinesOnTop()</code>	Bestimmt, ob der <i>GridLayer</i> oberhalb oder unterhalb der Balken gezeichnet wird.

Komponenten

Wie bereits mehrfach erwähnt besteht ein Gantt Chart aus einer Vielzahl von Unterkomponenten („children components“). Um auf diese zuzugreifen bietet *AbstractGanttChart* eine Vielzahl an Methoden, welche in der folgenden Tabelle beschrieben werden.

Methode	Beschreibung
<code>getDateline()</code>	Liefert die <i>Dateline</i> zurück. Diese wird häufig dann benötigt, um die von ihr angezeigte Zeitspanne („der Horizont“) oder Granularität zu ändern. Für diese beiden Anwendungsfälle werden allerdings auch Hilfsmethoden („convenience methods“) von <i>AbstractGanttChart</i> zur Verfügung gestellt: <pre>AbstractGanttChart.setTimeSpan(ITimeSpan) AbstractGanttChart.setGranularity(IGranularity)</pre>
<code>getEventline()</code>	Liefert die <i>Eventline</i> zurück, welche für die Anzeige von solchen Aktivitäten und Meilensteinen zuständig ist, welche für alle Zeilen im Gantt Chart relevant sind.
<code>getGridComponents()</code>	Liefert alle jene Unterkomponenten zurück, welche das <i>IGridComponent</i> Interface implementieren. Dies sind die <i>Eventline</i> und der <i>LayerContainer</i> , wobei letzterer mehrfach vorkommen kann (siehe <i>DualGanttChart</i>). Diese Abfrage ist zum Beispiel für das <i>GridControlPanel</i> wichtig, damit dieses eine <i>GridControl</i> für jede <i>IGridComponent</i> Instanz anzeigen kann. Mithilfe der <i>GridControl</i> kann ein virtuelles Gitter kontrolliert werden, mit welchem man die Start- und Endzeiten von Balken einfacher editieren kann.
<code>getLayerContainers()</code>	Liefert alle <i>LayerContainer</i> Instanzen zurück. Je nach Gantt Chart Typ kann dies einer (<i>GanttChart</i>) oder mehrere sein (<i>DualGanttChart</i>).

Methode	Beschreibung
<code>getLayerContainerScrollPanes()</code>	Liefert alle <i>LayerContainerScrollPane</i> Instanzen zurück. Je nach Gantt Chart Typ kann dies eine (<i>GanttChart</i>) oder mehrere sein (<i>DualGanttChart</i>). Eine solche Pane ist ein Wrapper um einen <i>LayerContainer</i> herum und implementiert die Scrolling Funktionalität. Weiterhin beinhaltet sie die linke und rechte obere und untere Ecke, in welche beliebige weitere Komponenten platziert werden können.
<code>setRowResizePosition(int) getRowResizePosition()</code>	Im <i>AbstractGanttChart</i> wird eine horizontale schwarze dicke Linie gezeichnet wenn der Benutzer die Höhe einer Zeile verändert. Die Position dieser Zeile wird mit diesen Methoden gesetzt bzw. abgefragt. Für gewöhnlich werden sie nur intern verwendet.
<code>getTimeline()</code>	Liefert die <i>Timeline</i> zurück. Dies ist der Container, welcher die <i>Dateline</i> und die <i>Eventline</i> beinhaltet.
<code>getTreeTableHeader()</code>	Liefert den Header der <i>TreeTable</i> zurück. Der Header stellt die Spaltenköpfe dar. Obwohl ein Gantt Chart mehrere Tabellen haben kann gibt es immer nur genau einen <i>TreeTableHeader</i> . Dieser wird gemeinsam genutzt.
<code>getTreeTables()</code>	Liefert alle <i>TreeTable</i> Instanzen zurück. Je nach Gantt Chart Typ kann dies einer (<i>GanttChart</i>) oder mehrere sein (<i>DualGanttChart</i>).
<code>getTreeTableScrollPanes()</code>	Liefert alle <i>TreeTableScrollPane</i> Instanzen zurück. Je nach Gantt Chart Typ kann dies eine (<i>GanttChart</i>) oder mehrere sein (<i>DualGanttChart</i>). Eine solche Pane ist ein Wrapper um eine <i>TreeTable</i> herum und implementiert die Scrolling Funktionalität. Weiterhin beinhaltet sie die linke obere und untere Ecke, in welche beliebige weitere Komponenten platziert werden können.

Navigation

Die Suche nach Informationen in einem Gantt Chart gestaltet sich unter Umständen recht schwierig. Abhängig ist sie von der Dauer des abgedeckten Zeitraums („Horizont“), der benutzten Granularität (Tage, Stunden, Minuten), der Menge und der Feinheit der Daten. Daher sind Navigationshilfen im Gantt Chart unabdingbar für eine effiziente Handhabung. *AbstractGanttChart* leistet seinen Beitrag und definiert die folgenden Methoden:

Methode	Beschreibung
<code>showAllObjects()</code>	Stellt sicher, daß alle Balken im sichtbaren Bereich des Gantt Charts erscheinen. Erreicht wird dies durch entsprechende Anpassungen in der Zeitleiste. Der notwendige sichtbare Zeitbereich wird berechnet basierend auf der frühesten Startzeit und der spätesten Endzeit aller Balken. Änderungen am ersten sichtbaren Zeitpunkt, an der Granularität und am Zoom Faktor bewirken dann, daß alle Balken für den Benutzer zu sehen sind.
<code>showEarliestObjects() showLatestObject();</code>	Ändert den ersten bzw. letzten sichtbaren Zeitpunkt so, daß die am frühesten beginnenden bzw. am spätesten endenden Balken sichtbar werden. Diese Balken können sich unter Umständen in einer Zeile befinden, welche momentan nicht sichtbar ist. Eventuell muss daher nach unten oder oben gescrollt werden.
<code>showTime(long, boolean)</code>	Macht den angegebenen Zeitpunkt sichtbar. Entweder als frühester Zeitpunkt des sichtbaren Abschnitts, oder zentriert in der Mitte.
<code>showTimeNow(boolean)</code>	Macht den aktuellen Zeitpunkt („Time Now“) sichtbar. Entweder als frühester Zeitpunkt des sichtbaren Abschnitts, oder zentriert in der Mitte. Die vertikale „Time Now“ Linie wird hierdurch sichtbar.

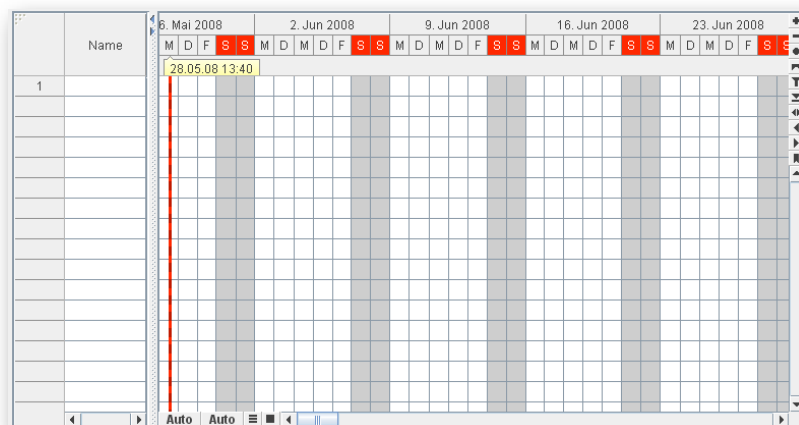
Hilfsfunktionen

Mehrere Hilfsfunktionen sind ebenfalls auf *AbstractGanttChart* definiert.

Methode	Beschreibung
<code>expandAll()</code> <code>collapseAll()</code>	Öffnet bzw. schliesst sämtliche Baumknoten in allen Tabellen (Erinnerung: <i>DualGanttChart</i> enthält 2 Tabellen).
<code>alignEndTimes()</code> <code>alignStartTimes()</code>	Ändert die Start- bzw. Endzeiten der momentan selektierten Balken so ab, daß alle Balken die gleiche früheste Start- bzw. späteste Endzeit haben.
<code>resetToPreferredSizes()</code>	Stellt den vertikalen Splitter im Gantt Chart so ein, daß die Tabelle vollständig sichtbar wird.
<code>getDatelineModel()</code>	Liefert das Model der <i>Dateline</i> zurück.
<code>setTimeSpan(ITimeSpan)</code> <code>getTimeSpan()</code>	Setzt bzw. liefert die von der Dateline abgedeckte Zeitspanne („Horizont“).
<code>setGranularity(IGranularity)</code> <code>getGranularity()</code>	Setzt bzw. liefert die von der Dateline angezeigte Granularität („Tage“, „Stunden“, „Minuten“, ...).

4. GanttChart

Die *GanttChart* Klasse ist eine direkte Unterklasse von *AbstractGanttChart*. Sie implementiert einen „einfachen“ Gantt Chart, d.h. nur eine linke und eine rechte Seite (im Gegensatz zum *DualGanttChart*, welcher jeweils zwei linke und rechte Seiten hat).



GanttChart

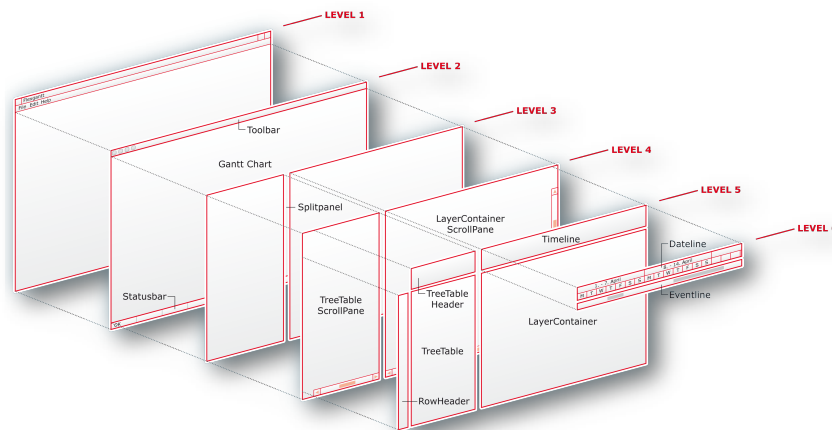
Entsprechend gibt es auch nur die folgenden Methoden, um auf die einzelnen Unterkomponenten zuzugreifen:

```

getTreeTable();
getTreeTableRowHeader();
getTreeTableScrollPane();
getLayerContainer();
getLayerContainerRowHeader();
getLayerContainerScrollPane();

```

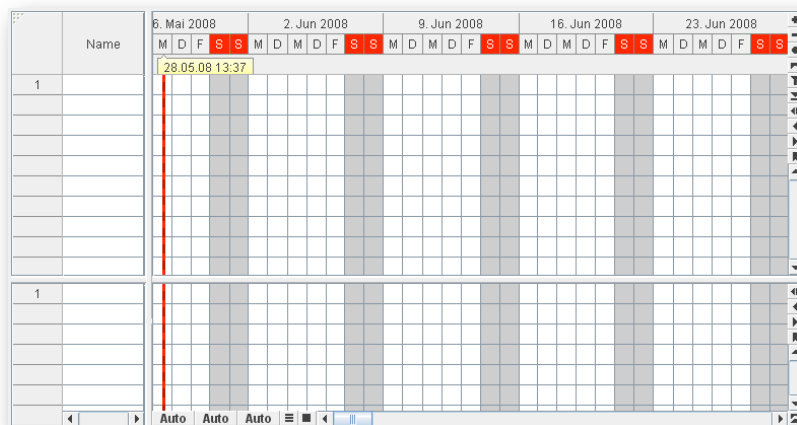
Es gibt zwar auch die Methoden, um auf „alle“ Instanzen von *TreeTable*, *LayerContainer*, etc... zuzugreifen, allerdings liefern diese immer nur genau eine Instanz zurück.



Aufbau des GanttChart Containers

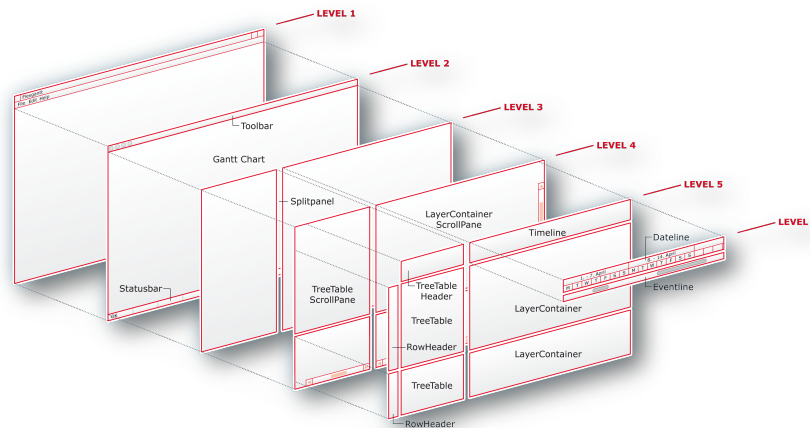
5. DualGanttChart

Der *DualGanttChart* implementiert ein Feature, welches von vielen Applikationen benötigt wird, nämlich das synchrone Scrollen von zwei unterschiedlichen Abschnitten des gleichen Planes oder von zwei völlig verschiedenen Plänen. Dies erreicht der *DualGanttChart* indem er jeweils zwei linke und rechte Seiten hat, welche jeweils untereinander angeordnet sind.



DualGanttChart

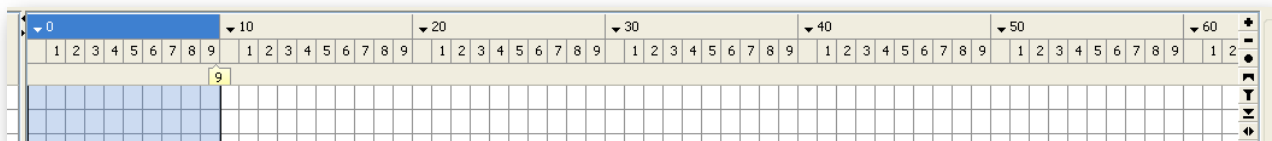
Der LayoutManager von DualGanttChart garantiert, dass die entsprechenden Komponenten immer genau die gleiche Breite aufweisen. Die beiden TreeTable Instanzen teilen sich die gleiche TreeTableHeader Instanz und die beiden LayerContainer Instanzen teilen sich die gleiche Timeline Instanz.



Aufbau des DualGanttChart Containers

6.SimpleGanttChart

Die mit Abstand meisten Gantt Charts zeigen innerhalb der Zeitleiste einen Kalender. Dieser Kalender kann dann von Jahren bis Stunden alles anzeigen. Der von **FlexGantt** eingesetzte Kalender kann sogar alles von Jahrtausenden bis Millisekunden anzeigen. Es gibt aber auch Ausnahmen, welche sich hauptsächlich im Universitäts- und Forschungsumfeld finden. Hier werden häufig Zeiteinheiten („Units“) benutzt, um Planungsergebnisse schneller erfassen und miteinander vergleichen zu können. Diese Units sind für gewöhnlich einfache ganze Zahlen (1, 2, 3, ..., 100, 101, ..., 1000, ...). **FlexGantt** bietet hierfür Unterstützung in Form des *SimpleGanttChart*. Dieser Gantt Chart benutzt die *SimpleGranularity*, das *SimpleDateLineModel*, und den *SimpleDateLineRenderer*.



Die Timeline des SimpleGanttChart

7.TreeTable

Der grösste Teil der linken Seite eines Gantt Charts ist implementiert in der *TreeTable* Klasse. Diese Klasse hat eine hohe Komplexität und eine grosse Anzahl an Methoden. Die folgenden Abschnitte versuchen die Methoden in Gruppen einzuteilen. Nicht jede Methode wird hierbei im Detail beschrieben. Wichtig ist nur zu verstehen in welchem Zusammenhang sie benutzt werden. Es ist wichtig zu wissen, daß die Spaltenköpfe der Tabelle nicht zur *TreeTable* gehören, sondern zur *TreeTableHeader* Komponente.

Selektieren von Zeilen

Gantt Charts müssen dem Benutzer die Möglichkeit bieten eine oder mehrere Zeilen zu selektieren, um anschliessend auf diesen Aktionen / Kommandos ausführen zu können. *TreeTable* verwendet hierfür ein eigenes Selektionsmodell, welches das *ITreeTableSelectionModel* Interface implementiert. Per Default verwendet *TreeTable* eine Instanz vom Typ *DefaultTreeTableSelectionModel*. Es stehen Methoden zur Verfügung, um dieses Model abzufragen bzw. mit einer anderen Implementierung zu ersetzen.

```
getSelectionModel()
setSelectionModel(ITreeTableSelectionModel)
```

Wie für Selektionsmodelle in Swing üblich, können sich interessierte Parteien als Listener an das Model anhängen. Diese Listener werden dann stets informiert, wenn sich der Selektionszustand innerhalb der Tabelle geändert hat, d.h. wenn Zeilen selektiert oder deselektiert werden.

```
addTreeSelectionListener(TreeSelectionListener)
removeTreeSelectionListener(TreeSelectionListener)
```

Je nach Anwendungstyp oder aktuellen Zustand einer Anwendung kann es sein, dass eine Tabelle unterschiedliche Selektionsverfahren zulässt. So kann es sein, dass nur jeweils eine Zeile ausgewählt werden darf, oder aber nur hintereinander folgende Zeilen, oder aber auch beliebige und beliebig viele Zeilen. Dieses Verhalten kann durch das Setzen eines Selektionsmodus gesteuert werden.


```
setSelectionMode(int)
getSelectionMode()
```

Die erlaubten Werte für diese Methoden sind definiert im Interface `TreeSelectionModel`. Es sind :

- `SINGLE_TREE_SELECTION`
- `CONTIGUOUS_TREE_SELECTION`
- `DISCONTIGUOUS_TREE_SELECTION`.

Gleich eine ganze Reihe an Methoden stehen dann zur Verfügung, um Selektionen zu erzeugen, zu entfernen, oder aber abzufragen. Die meisten dieser Methoden delegieren zum Selektionsmodel.

```
selectAll()
clearSelection()
setSelectionPath(TreePath)
setSelectionPaths(TreePath[])
addSelectionPath(TreePath)
removeSelectionPath(TreePath)
addSelectionPaths(TreePath[])
removeSelectionPaths(TreePath[])
addSelectionRow(int)
addSelectionRows(int[])
setSelectionRow(int)
setSelectionRows(int[])
getSelectionPath()
getSelectionPaths()
setSelectionInterval(int, int)
addSelectionInterval(int, int)
removeSelectionInterval(int, int)
isPathSelected(TreePath)
isRowSelected(int)
getSelectionCount()
```

Editieren von Werten in Zellen

Die `TreeTable` Klasse erlaubt es dem Benutzer die Werte in jeder ihrer Zellen zu verändern. Hierfür werden Editoren verwendet, welche wie in Swing üblich, auf Typen gemapped werden. Möchte der Benutzer eine bestimmte Zelle editieren, dann wird zuerst einmal der Typ des aktuellen Wertes ermittelt¹. Für diesen Typ wird dann der zugehörige Editor bestimmt und von der Tabelle in der entsprechenden Zelle platziert. `TreeTable` definiert bereits mehrere Editoren. In der folgenden Tabelle sieht man das Mapping von Object Type auf Editor Type.

Object Type	Editor Type
Object	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$GenericEditor</code>
Boolean	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$BooleanEditor</code>
Date	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$DateEditor</code>
Calendar	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$CalendarEditor</code>
Number	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$NumberEditor</code>
Color	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$ColorEditor</code>
Enum	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$EnumEditor</code>

Wie man sehen kann sind sämtliche vordefinierten Editoren als innere Klassen von `TreeTable` definiert. Um eigene Editoren zu benutzen, muss man diese bei der `TreeTable` registrieren.

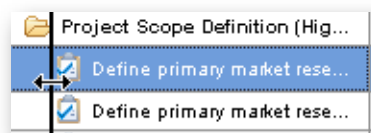
```
setCellEditor(Class, ITreeTableCellEditor)
getCellEditor(int, int)
getCellEditor(Class)
```

¹ Dies ist entweder der Typ des Objektes in der Zelle oder aber, wenn kein Wert vorhanden ist, der von der `TreeTableColumn` definierte Spaltentyp.

Methode	Beschreibung
<code>isEditingEnabled()</code> <code>setEditingEnabled(boolean)</code>	Bestimmt oder verändert die Möglichkeit zum Editieren.
<code>isCellEditable(int, int)</code>	Bestimmt, ob die angegebene Zelle editierbar ist.
<code>editCellAt(int, int)</code> <code>editCellAt(int, int, EventObject)</code>	Startet das Editieren der angegebenen Zelle.
<code>editCellFocused(InputEvent)</code>	Startet das Editieren der momentan fokussierten Zelle.
<code>isEditing()</code>	Bestimmt, ob der Benutzer gerade eine Zelle editiert.
<code>getCellEditor()</code>	Liefert den aktuell verwendeten Zelleditor zurück.

Einrücken von Knoten

Das Einrücken von Knoten wird ebenfalls von `TreeTable` unterstützt. Beim Einrücken wird einem Baumknoten ein Elternknoten zugewiesen der eine Ebene tiefer liegt. Dadurch verschiebt sich der Knoten ein wenig nach rechts.



Knoten Einrücken

Um überhaupt Knoten einrücken zu können, muss das entsprechende Feature bei der Tabelle eingeschaltet werden.

```
setIndentEnabled(boolean)
isIndentEnabled()
```

Anschließend kann der Benutzer entweder interaktiv Knoten einrücken, indem er den Mauszeiger an die linke Kante eines Wertes in der `KeyColumn` heranführt, oder aber programmatisch, indem die entsprechenden von der `TreeTable` Klasse zur Verfügung gestellten Methoden aufgerufen werden².

```
indentNodes(TreePath[])
outdentNodes(TreePath[])
```

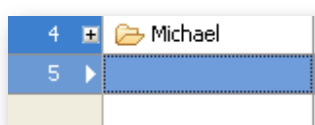
Diese Methoden erzeugen am Ende *Commands*, die vom *CommandStack* des Gantt Charts in einem separaten Thread ausgeführt werden.

Knoten Erzeugen und Löschen

In der *TreeTable* können neue Knoten eingefügt und gelöscht werden, wenn das entsprechende Feature eingeschaltet wurde. Dies geschieht mit den folgenden Methoden:

```
setCreationEnabled(boolean)
isCreationEnabled()
setDeletionEnabled(boolean)
isDeletionEnabled()
```

Ob das „Creation“ Feature eingeschaltet ist oder nicht, erkennt man an einem Pfeil in der Zeile nach der letzten Zeile. Zum Anlegen eines neuen Knotens muss der Benutzer in dieser Zeile lediglich einen Namen eintippen und diesen mit der Eingabetaste abschliessen.



² Es sollte angemerkt werden, daß das Wort „outdent“ so im Englischen nicht existiert. Aus Symmetriegründen erschien es aber doch als die richtige Wahl für den Namen der Umkehrmethode.

Knoten Hinzufügen

Die eigentliche Arbeit, um den neuen Knoten zu erzeugen und ihn an das Model anzufügen passiert in einer Policy und in dem von der Policy zurück gelieferten *Command*.

Eine weitere Möglichkeit zum Erzeugen neuer Knoten ist das Einfügen eines Knotens in einer beliebigen Zeile in der Tabelle. Hierfür gibt es eine weitere Methode in der *TreeTable*:

```
insertNode(int)
```

Zum Löschen eines Knotens kann die folgende Methode verwendet werden:

```
deleteSelectedNodes()
```

Zeilenhöhen

Jede Zeile in einer *TreeTable* kann ihre eigene Höhe haben. Diese Höhe ist auch vom Benutzer interaktiv einstellbar. Er muss nur den Mouse Cursor auf eine der horizontalen Linien im *TreeTableRowHeader* bewegen. Dann verändert sich der Cursor zum einem Resize Cursor. Durch eine Drag Bewegung kann dann die Höhe der Zeile verändert werden. Auch dieses Feature lässt sich mittels der nachstehenden Methoden ein- und ausschalten.



```
setResizingEnabled(boolean)
isResizingEnabled()
```

Tabellenspalten

Eigentlich ist die Klasse *AbstractGanttChart* zuständig für alles was mit Tabellenspalten zu tun hat. Es gibt aber auch bei der *TreeTable* eine Reihe von Methoden, die für Tabellenspalten relevant sind. Die meisten von ihnen delegieren allerdings weiter. Entweder an das Spaltenmodell, an *AbstractGanttChart* oder an *TreeTableHeader*. Der Header ist der einzige, der weiss an welcher Position und wie breit die Spalten gerade angezeigt werden.

Methode	Beschreibung
<code>setColumnModel(IColumnModel)</code>	Setzt ein neues Model für die Tabellenspalten. Diese Methode darf nur intern vom Framework verwendet werden. Der eigentliche Ort für die Anwendung um dieses Model zu setzen ist in <i>AbstractGanttChart</i> . Von dort wird diese Methode bei allen im Gantt Chart vorhandenen Tabellen aufgerufen.
<code>getColumnModel()</code>	Liefert das aktuell verwendete Model für Spalten zurück.
<code>getKeyColumn()</code>	Liefert die KeyColumn zurück. Delegiert nach <i>AbstractGanttChart</i> .
<code>getColumn(int)</code>	Liefert die Spalte für den gegebenen Index zurück. Delegiert nach <i>TreeTableHeader</i> .
<code>getColumnIndex(TreeTableColumn)</code>	Liefert den Index der gegebenen Spalte zurück. Delegiert zum Spaltenmodell.
<code>getColumnCount()</code>	Liefert die Anzahl der Spalten zurück. Delegiert zum Spaltenmodell.
<code>getKeyColumnPosition()</code>	Liefert die Position der KeyColumn zurück. Delegiert zum <i>AbstractGanttChart</i> .
<code>getColumnIndexAt(int)</code>	Liefert den Index der Spalte an der gegebenen X-Koordinate zurück. Delegiert zum <i>TreeTableHeader</i> .
<code>getColumnAt(int)</code>	Liefert die Spalte an der gegebenen X-Koordinate zurück. Delegiert zum <i>TreeTableHeader</i> .

Tabellenspalten sind nicht nur wichtig für das Layout der Tabelle, sondern auch für das Editerverhalten. Jede Spalte definiert nämlich einen Objekttypen, welcher benutzt wird, um den richtigen Editor für die Zellen in der Spalte zu bestimmen. Allerdings nur dann, wenn eine Zelle nicht bereits einen Wert besitzt. In diesem Fall wird der Typ des Wertes für die Bestimmung des Editors verwendet.

Listener und Events

An die *TreeTable* können Listener angehängt werden, welche darüber informiert werden, ob Knoten im Baum geöffnet oder geschlossen werden. Hierzu müssen die Listener das *TreeExpansionListener* Interface implementieren. Danach können sie sich mittels der folgenden Methoden an die *TreeTable* anhängen oder wieder entfernen.

```
addTreeExpansionListener(TreeExpansionListener)
removeTreeExpansionListener(TreeExpansionListener)
```

Ein sehr ähnlicher Listener ist das *TreeWillExpandListener* Interface. Objekte, die dieses Interface implementieren, werden über das Öffnen und Schliessen von Knoten unterrichtet bevor es geschieht, d.h. nach dem Klick des Benutzers, aber noch bevor der Knoten offen ist und seine Kinder sichtbar sind. Dies bietet die Möglichkeit ein sogenanntes „Lazy Loading“ zu implementieren.

Beim Lazy Loading werden die notwendigen Datenstrukturen erst dann erzeugt, wenn sie auch wirklich benötigt werden. Dieser Strategie kann beim initialen Erzeugen der *TreeTable* sehr viel Zeit sparen, da hier dann nur Daten für die erste Ebene der Baumknoten aus einer Datenbank oder einem Application Server gelesen werden müssen.

Die folgenden Methoden von *TreeTable* können benutzt werden, um *TreeWillExpandListener* an die Tabelle anzuhängen oder zu entfernen.

```
addTreeWillExpandListener(TreeWillExpandListener)
removeTreeWillExpandListener(TreeWillExpandListener)
```

Wichtig zu erwähnen ist bei dieser Art Listener auch noch, dass die Listener Exceptions von Typ *ExpandVetoException* werfen können. In diesem Fall wird das Öffnen oder Schliessen des Baumknotens abgebrochen und der Knoten bleibt in seinem ursprünglichen Zustand.

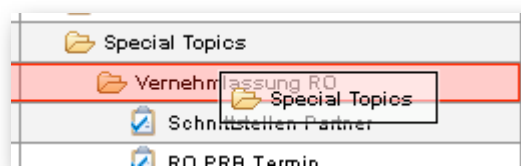
Drag & Drop

Die *TreeTable* unterstützt direkt Drag & Drop (DnD) Funktionalität. Allerdings ist der Code hierfür ausgelagert und befindet sich in der *TreeTableDragAndDropManager* Klasse. Entsprechend klein ist auch die Anzahl jener Methoden in *TreeTable*, die einen Bezug zu DnD haben. In der *TreeTable* kann man lediglich das gesamte DnD ein- und ausschalten. Hierfür existieren die folgenden Methoden.

```
setDraggingEnabled(boolean)
isDraggingEnabled()
```

Per Default zeigt die *TreeTable* „gute“ und „schlechte“ Drops mit zwei unterschiedlichen Farben an. Diese können mit den folgenden Methoden kontrolliert werden.

```
getDropColorValid()
setDropColorValid(Color)
getDropColorInvalid()
setDropColorInvalid(Color)
```



Invalid Drop Color

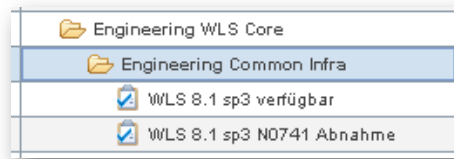


Valid Drop Color

Wie bereits erwähnt befindet sich die meiste DnD Funktionalität in der *TreeTableDragAndDropManager* Klasse. Diese Klasse wird nur intern verwendet und stellt keine offene API dar. Applikationen steuern das DnD Verhalten der *TreeTable* über die bei ihr registrierten Policies. Mehr Informationen zu diesem Thema können im Dokument „FlexGantt - Policies“ gefunden werden.

Fokus

In der *TreeTable* hat immer genau eine Zelle den Fokus. Sichtbar wird dies durch einen entsprechenden Rand um die Zelle herum.



Fokusierte Zelle

Verwaltet wird der Fokus von der Klasse *CellFocusManager*. Diese kennt immer die aktuell fokusierte Spalte und Zeile. Weiterhin bietet sie Methoden an, um den Fokus auf die vorherige, die nächste, die obere, die untere, die erste, oder die letzte Zelle zu setzen. Bezüglich Fokus bietet die *TreeTable* nur Methoden an, die den Manager zurück liefern oder an diesen delegieren.

```
getCellFocusManager()
getFocusedRow()
getFocusedColumn()
```

Baumknoten und Tree Paths

Eine *TreePath* Instanz repräsentiert einen eindeutigen Pfad zu einem Knoten im Baum. Während der gleiche Baumknoten mehrfach in einem Baum vorkommen kann, ist ein *TreePath* eindeutig. Entsprechend nützlich ist diese Klasse und entsprechend häufig wird sie im Zusammenhang mit der *TreeTable* verwendet. Ein Pfad kann zum Beispiel dafür genutzt werden, um programmatisch einen Baumknoten zu öffnen oder zu schließen.

```
isTreePathExpanded(TreePath)

expandAll()
expandPath(TreePath)
expandRow(int)

collapseAll()
collapsePath(TreePath)
collapseRow(int)
```

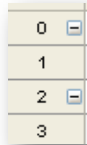
Es gibt noch eine ganze Reihe an weiteren Methoden, welche einen *TreePath* zurück liefern oder diesen als ein Argument benötigen.

```
containsPath(TreePath)
getPathsBetweenRows(int, int)
getTreePath(int)
getTreePathAt(int)
getTreePaths(int[])
getRowsForPaths(TreePath[])
getRowForPath(TreePath)
```

Visuelle Attribute

Wie alle Komponenten verfügt auch die *TreeTable* über einfache visuelle Attribute, welche mit Getter- und Setter-Methoden abgefragt bzw. gesetzt werden können. Die folgende Tabelle listet diese Methoden auf und gibt kurze Erklärungen pro Methode. Die meisten Setter-Methoden erzeugen *PropertyChangeEvents*, welche von der Tabelle an ihre Listener geschickt werden. Die Namen der Events sind als Konstanten (public static final) in der *TreeTable* Klasse definiert.

Methode	Beschreibung
setInset(int) getInset()	Spezifiziert eine Anzahl an Pixel um die Baumknoten eingerückt erscheinen, wenn sie sich auf einer unteren Ebene befinden. Die Anzahl an Pixeln, um die der Knoten letztlich eingerückt wird hängt von der Ebenentiefe ab. Diese Tiefe wird mit dem Inset multipliziert. Beispiel: der Knoten befindet sich zwei Ebenen tief im Baum, dann wird er bei einem Inset von 10 um 20 Pixel eingerückt ($2 * inset = 2 * 10 = 20$)
setGridColor(Color)	Eine Convenience Methode, die gleichzeitig die Farbe der vertikalen und der horizontalen Gitterlinien ändert.

Methode	Beschreibung
<code>getVerticalGridColor()</code> <code>setVerticalGridColor(Color)</code>	Liefert / setzt die Farbe für die vertikalen Gitterlinien.
<code>getHorizontalGridColor()</code> <code>setHorizontalGridColor(Color)</code>	Liefert / setzt die Farbe für die horizontalen Gitterlinien.
<code>isHorizontalLinesVisible()</code> <code>setHorizontalLinesVisible(boolean)</code>	Bestimmt, ob die horizontalen Gitterlinien angezeigt werden.
<code>isRowNumbersVisible()</code> <code>setRowNumbersVisible(boolean)</code>	Bestimmt, ob die Zeilennummern im <i>TreeTableRowHeader</i> angezeigt werden oder nicht.  <i>Zeilennummern</i>
<code>isRootVisible()</code> <code>setRootVisible(boolean)</code>	Legt fest, ob die Baumwurzel (Root) sichtbar ist oder nicht.
<code>isVerticalLinesVisible()</code> <code>setVerticalLinesVisible(boolean)</code>	Bestimmt, ob die vertikalen Gitterlinien angezeigt werden.
<code>getAlternatingForeground()</code> <code>setAlternatingForeground(Color)</code>	Liefert / setzt eine „alternierende“ Farbe für den Vordergrund (Textfarbe) von Tabellenzeilen. Diese Farbe wird immer in jeder zweiten Zeile verwendet.
<code>getAlternatingBackground()</code> <code>setAlternatingBackground(Color)</code>	Liefert / setzt eine „alternierende“ Farbe für den Hintergrund einer Tabellenzeile. Diese Farbe wird immer in jeder zweiten Zeile verwendet.
<code>getSelectionBackground()</code> <code>setSelectionBackground(Color)</code>	Liefert / setzt eine Farbe für den Vordergrund (Textfarbe) von selektierten Tabellenzeilen.
<code>getSelectionForeground()</code> <code>setSelectionForegroundColor(Color)</code>	Liefert / setzt eine Farbe für den Hintergrund von selektierten Tabellenzeilen.
<code>getTexture()</code> <code>setTexture(Image)</code>	Liefert / setzt ein Image, welches als Texture für den Hintergrund der Tabelle verwendet wird. Das Image dient als „Tile“, d.h. es wird im Hintergrund immer wieder horizontal und vertikal wiederholt.
<code>getAlpha()</code> <code>setAlpha(float)</code>	Liefert / setzt einen Alpha Wert für Transparenz. Ist dieser Wert kleiner als 1 werden die Zeilen transparent und Texturen (z.B.) im Hintergrund bleiben sichtbar. Selbst dann wenn alternierende Zeilenfarben verwendet werden oder wenn eine Zeile selektiert ist.
<code>getDefaultRowHeight()</code> <code>setDefaultRowHeight(int)</code>	Liefert / setzt die Standard Zeilenhöhe. Diese wird für leere Zeilen verwendet. Benutzte Zeilen erhalten Ihre Höhe von einer Policy. Die Default Implementierung der dafür zuständigen Policy delegiert an den Knoten und fragt diesen nach der Höhe.

Zeichnen

Die *TreeTable* verwendet zum Zeichnen das bereits von Swing bekannte Konzept der *Renderer*. Dies sind Objekte, welche UI Komponenten zur Verfügung stellen. Diese Komponenten werden temporär einem Container hinzugefügt, um sich an einer ganz bestimmten Stelle zu zeichnen. *Renderer* werden Objekt Typen zugewiesen, d.h. unterschiedliche Objekte können von unterschiedlichen *Renderern* gezeichnet werden. *TreeTable* definiert bereits eine ganze Reihe von *Renderern*. Diese können der folgenden Tabelle entnommen werden.

Object Type	Renderer Type
Object	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$DefaultTreeTableCellRenderer</code>
Boolean	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$BooleanRenderer</code>
Date	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$DateRenderer</code>
Calendar	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$CalendarRenderer</code>
Number	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$NumberRenderer</code>
Double	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$DoubleRenderer</code>
ImageIcon	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$IconRenderer</code>
Color	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$ColorRenderer</code>
Enum	<code>com.dlsc.flexgantt.swing.treetable.TreeTable\$EnumRenderer</code>


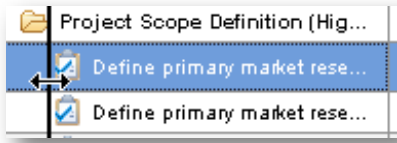
Wie man sehen kann sind sämtliche vordefinierten Renderer als innere Klassen von *TreeTable* definiert. Um eigene Renderer zu benutzen, muss man diese bei der *TreeTable* registrieren.

```
setCellRenderer(Class, ITreeTableCellRenderer)
getCellRenderer(Class)
```

Das eigentliche Zeichnen der *TreeTable* verteilt sich über mehrere Methoden.

Methode	Beschreibung
<code>paintComponent()</code>	Zeichnet die gesamte Komponente indem sie erst den Hintergrund zeichnen lässt und dann die einzelnen Spalten. Findet gerade eine Drag & Drop Operation statt oder wird eine Zeilenhöhe verändert, so werden auch diese Vorgänge durch den Aufruf der entsprechenden Zeichenroutinen visualisiert.
<code>paintBackground()</code>	Zeichnet den Hintergrund. Entweder eine einfache Farbe oder aber eine Textur.
<code>paintColumns()</code>	Zeichnet sämtliche Spalten. Für jede Spalte wird <code>paintColumn()</code> aufgerufen.
<code>paintColumn()</code>	Zeichnet eine einzelne Spalte. Für jede Zelle in der Spalte wird <code>paintCell()</code> aufgerufen.
<code>paintCell()</code>	Zeichnet eine einzelne Zelle. Der Objekttyp der Zelle und der dazu passende Renderer werden bestimmt. Vom Renderer wird die Renderer Komponente abgefragt und dem <i>TreeTable</i> Container hinzugefügt. Anschliessend wird die <code>paintComponent()</code> Methode der Komponente aufgerufen.

Optional werden zwei weitere Methoden beim Zeichnen involviert:

Methode	Beschreibung
<code>paintDragAndDrop()</code>	<p>Zeichnet das Drag & Drop Feedback. Wird eine Zeile verschoben, so wird die Zelle in der <i>KeyColumn</i> an der Position des Mouse Cursors gezeichnet. Diese Zelle wird auch dann benutzt wenn die <i>KeyColumn</i> gerade nicht sichtbar sein sollte.</p>  <p><i>Drag & Drop Feedback</i></p>
<code>paintIndentationLine()</code>	<p>Zeichnet eine einfache vertikale Linie an der aktuellen Mouse Cursor Position während der Benutzer einen Knoten einrückt.</p>  <p><i>Einrück-Linie</i></p>

Die Gantt Charts von **FlexGantt** haben eine eingebaute Unterstützung für alternierende Vorder- und Hintergrundfarben für die Zeilen. In der *TreeTable* Klasse befinden sich entsprechende Methoden, um die Farben abhängig vom Index einer Zeile zu bestimmen.

Methode	Beschreibung
<code>getForeground(int, boolean)</code>	Liefert die Vordergrundfarbe für die angegebene Zeile zurück. Die Vordergrundfarbe wird für gewöhnlich für den Text in den Zellen benutzt. Optional kann die Methode auch berücksichtigen, ob die Zeile gerade selektiert ist oder nicht. Ist sie selektiert, so liefert die Methode die Vordergrundfarbe für selektierte Zeilen zurück.
<code>getForeground(int)</code>	Liefert die Vordergrundfarbe für die angegebene Zeile zurück. Die Vordergrundfarbe wird für gewöhnlich für den Text in den Zellen benutzt.
<code>getBackground(int, boolean)</code>	Liefert die Hintergrundfarbe für die angegebene Zeile zurück. Die Hintergrundfarbe wird für gewöhnlich für den Hintergrund in den Zellen benutzt. Optional kann die Methode auch berücksichtigen, ob die Zeile gerade selektiert ist oder nicht. Ist sie selektiert, so liefert die Methode die Hintergrundfarbe für selektierte Zeilen zurück.
<code>getBackground(int)</code>	Liefert die Hintergrundfarbe für die angegebene Zeile zurück. Die Hintergrundfarbe wird für gewöhnlich für den Text in den Zellen benutzt.

Hilfsmethoden

Methode	Beschreibung
<code>getGanttChart()</code>	Liefert den Gantt Chart zurück zu dem die Tabelle gehört (parent container).
<code>getModel()</code>	Liefert das Model zurück. Die TreeTable benötigt ein <i>ITreeTableModel</i> , welches eine Oberklasse ist von <i>IGanttChartModel</i> .

Methode	Beschreibung
<code>getLayerContainer()</code>	Liefert den LayerContainer rechts von der Tabelle zurück. Die TreeTable und der LayerContainer arbeiten eng zusammen. Die von der Tabelle angezeigten Zeilen sind der Input für den LayerContainer.
<code>getCellRect(int, int)</code>	Liefert das Rechteck für die angegebene Zelle zurück. Das Rechteck wird für Editoren und Renderer benötigt, um sie an der richtigen Stelle zu platzieren.
<code>getTreeTableHeader()</code>	Liefert den Header der Tabelle zurück. Es ist wichtig zu verstehen, daß nicht die Tabelle der Besitzer des Headers ist, sondern der Gantt Chart. Dies liegt daran, daß ein Gantt Chart mehrere Tabellen beinhalten kann und sich diese alle den gleichen Header teilen müssen.
<code>getFirstVisibleRow()</code>	Liefert den Index der ersten sichtbaren Zeile zurück.
<code>getLastVisibleRow()</code>	Liefert den Index der letzten sichtbaren Zeile zurück.
<code>getRowCount()</code>	Liefert die Anzahl aller Zeilen zurück. Dieser Wert ist abhängig davon, welche Knoten im Baum geöffnet sind.
<code>getVisibleRowCount()</code>	Liefert die Anzahl der gerade sichtbaren Zeilen zurück. Dies sind jene Zeilen, welche gerade im Viewport der JScrollPane sichtbar sind.
<code>getTreeTableNodes()</code>	Liefert die interne Datenstruktur der Tabelle zurück.
<code>getRowHeight(int)</code>	Liefert die Höhe der angegebenen Zeile zurück. Jede Zeile in der Tabelle kann ihre eigene Höhe haben.
<code>scrollTo(int, boolean)</code>	Fordert die Tabelle auf die angegebene Zeile sichtbar zu machen. Optional kann angegeben werden, ob die Zeile am oberen Rand des sichtbaren Bereichs angezeigt werden soll. Dies ist dann wichtig, wenn auch möglichst viele Kinder des in der Zeile angezeigten Knotens sichtbar werden sollen.
<code>scrollTo(int)</code>	Fordert die Tabelle auf die angegebene Zeile sichtbar zu machen. Die Zeile kann entweder ganz unten oder ganz oben im sichtbaren Bereich angezeigt werden.
<code>scrollTo(TreePath, boolean)</code>	Fordert die Tabelle auf die durch den Pfad bestimmte Zeile sichtbar zu machen. Optional kann angegeben werden, ob die Zeile am oberen Rand des sichtbaren Bereichs angezeigt werden soll. Dies ist dann wichtig, wenn auch möglichst viele Kinder des in der Zeile angezeigten Knotens sichtbar werden sollen.
<code>scrollTo(TreePath)</code>	Fordert die Tabelle auf die durch den Pfad angegebene Zeile sichtbar zu machen. Die Zeile kann entweder ganz unten oder ganz oben im sichtbaren Bereich angezeigt werden.
<code>scrollFocusToVisible()</code>	Fordert die Tabelle auf jene Zeile sichtbar zu machen in der sich die fokussierte Zelle befindet.
<code>getRowAt(int)</code>	Bestimmt den Index der Zeile an der angegebenen Y-Koordinate.
<code>getColumnClass(int)</code>	Liefert den Typ jener Objekte zurück, welche in der angegebenen Spalte angezeigt werden (Boolean, Integer, String,).
<code>getRootNode()</code>	Liefert die Wurzel der von der Tabelle angezeigten Baumstruktur zurück.

Policies

Das Verhalten der *TreeTable* wird durch Policies kontrolliert. Zum Einsatz kommen sie bei der Tabelle beim Drag & Drop, beim Editieren von Zellen, bei der Bestimmung der Zeilenhöhe, und beim Einrücken von Baumknoten. Die Interfaces dieser Policies heissen:

- *INodeDragAndDropPolicy*
- *INodeEditPolicy*
- *IRowPolicy*
- *INodeIndentationPolicy*

Die Implementierungen dieser Policies müssen beim Policy Provider der Tabelle registriert werden. Zugriff auf den Provider bekommt man mittels der folgenden *TreeTable* Methoden.

```
getPolicyProvider()
setPolicyProvider(IPolicyProvider)
```

Eine genauere Beschreibung der Policies befindet sich im Dokument „FlexGantt - Policies“.

Interne Datenstrukturen

Der *TreeTable* liegt ein Model zugrunde, welches eine Baumstruktur definiert. Intern jedoch verwaltet die *TreeTable* eine Liste mit Instanzen vom Typ *TreeTableNode*. Jede *TreeTableNode* Instanz repräsentiert genau eine Zeile in der Tabelle. Das erste Element in dieser Liste repräsentiert die erste Zeile, das zweite die zweite Zeile, und das letzte die letzte Zeile.

Immer wenn eine Änderung am Model vorgenommen wird, muss die *TreeTable* die Liste mit den *TreeTableNodes* aktualisieren. Hierzu wird intern die folgende Methode aufgerufen.


```
updateNodes();
```

Zugriff auf die Liste bekommt man mit diesen Methoden.

```
getTreeTableNodeAt(int)
getTreeTableNodesBetween(int, int)
```

8.TreeTableHeader

Die *TreeTableHeader* Komponente ist für die Darstellung der Spaltenüberschriften oberhalb der *TreeTable* zuständig. Diese Überschriften sind für sämtliche Tabellen innerhalb des gleichen Gantt Chart gültig. Das heisst zum Beispiel, daß eine vom Benutzer ausgelöste Sortierung überall gilt. Im Falle des *DualGanttChart* werden sowohl die obere als auch die untere Tabelle sortiert.



Task Name	Actual Cost	Actual Duration	Actual Finish	Actual Overtime Cost
-----------	-------------	-----------------	---------------	----------------------

Spaltenköpfe

Model

Der *TreeTableHeader* benutzt als Model das *IColumnModel*, welches auf *AbstractGanttChart* definiert ist. Gesetzt wird das Model aber auch auf dem Header. Die Methoden hierfür werden nur intern gebraucht.

```
setModel(IColumnModel)
getModel()
```

Renderer

Wie fast alle FlexGantt Komponenten, benutzt auch der *TreeTableHeader* das Renderer Konzept. Für das Mapping von Object Typ auf Renderer wird der „Header Value“ der Spalte verwendet. Definiert wird der Header Value durch den Aufruf der folgenden Methode auf *TreeTableColumn*;

```
TreeTableColumn.setHeaderValue(Object)
```

Ist kein Header Value gesetzt, so wird der Name der Spalte verwendet. Da der Name immer vom Typ String ist, müssen Renderer in diesem Fall auf String gemapped werden.

```
setColumnHeaderRenderer(Class, IColumnHeaderRenderer)
getColumnHeaderRenderer(Class)
```

Eingesetzt werden die Renderer von den paint...() Methoden des Headers.

```
paintComponent(Graphics)
paintColumnHeader(Graphics, TreeTableColumn, int, int, int, int)
paintColumnHeaders(Graphics)
```

Zugriff auf Tabellenspalten

Verschiedene Methoden existieren, um auf die Tabellenspalten zuzugreifen.

Methode	Beschreibung
<code>getColumn(int)</code>	Liefert die Spalte für den angegebenen Index zurück.
<code>getColumnAt(int)</code>	Liefert die Spalte für die angegebene X-Koordinate zurück.
<code>getColumnBounds(int)</code>	Liefert das Rechteck für den Spaltenkopf der Spalte mit dem angegebenen Index zurück.
<code>getColumnBounds(TreeTableColumn)</code>	Liefert das Rechteck für den Spaltenkopf der angegebenen Spalte zurück.
<code>getColumnIndexAt(int)</code>	Liefert den Index jener Spalte zurück, die sich an der angegebenen X-Koordinate befindet.
<code>getColumnLocation(TreeTableColumn)</code>	Liefert die X-Koordinate für die angegebene Spalte zurück.
<code>getDraggedColumn()</code>	Liefert jene Spalte zurück, die im Moment des Methodenaufrufs vom Benutzer verändert wird (Breitenänderung).
<code>getFillerColumn()</code>	Liefert die zusätzliche Spalte zurück, die zum Auffüllen des freien Raumes rechts von der Tabelle benutzt wird.

Sortierung von Tabellenspalten

Durch einen Klick auf die Spaltenköpfe im TreeTableHeader kann der Benutzer Sortierungen auslösen. Wird dabei auch noch die STRG Taste gedrückt, können auch mehrere Spalten gleichzeitig sortiert werden. Pfeile mit Nummern rechts vom Namen stellen klar welche Spalte gerade sortiert ist und an welcher Stelle in der Sortierreihenfolge sie vorkommen (bei Mehrspaltensortierung).

Task Name ▲ 3	Actual Cost ▲ 1	Actual Duration ▼ 2
------------------	--------------------	------------------------

Mehrfachsartierung

Methode	Beschreibung
<code>getSortCount()</code>	
<code>getSortPosition(int)</code>	
<code>sortColumn(TreeTableColumn, boolean)</code>	
<code>sorting(SortingEvent)</code>	

Anpassen von Tabellenspalten

Die Grösse einer Spalte kann vom Benutzer angepasst werden. Hierzu muss der Mouse Cursor dicht an den rechten Rand eines Spaltenkopfes bewegt werden. Wenn sich der Mouse Cursor in den „Resize“ Cursor verändert, kann durch einen Mouse Drag die Breite verändert werden.

Actual Cost	Actual Duration	Actual Finish
-------------	-----------------	---------------

Resize Cursor

FlexGantt benutzt im Zusammenhang mit der Spaltenbreite auch den Begriff „optimale Breite“. Diese ist dann erreicht, wenn der Inhalt aller Zellen in einer Spalte vollständig sichtbar ist. Die Berechnung dieser Breite kann die Breite der für das Editieren verwendeten Komponenten, der Editoren, mit berücksichtigen. Definiert sind die Methoden zum Optimieren auf `AbstractGanttChart`. Mit einem Doppelklick während der „Resize“ Cursor angezeigt wird, kann der Benutzer diese Methoden aufrufen. Ein- und ausgeschaltet werden kann diese Möglichkeit mit den folgenden Methoden:

```
setDoubleClickResizeEnabled(boolean)
isDoubleClickResizeEnabled()
```

Visuelle Attribute

Methode	Beschreibung
<code>setGridColor(Color)</code> <code>getGridColor()</code>	Setzt / liefert jene Farbe zurück, die für die vertikalen Linien zwischen den Spaltenköpfen verwendet wird. Diese Linien definieren nicht wirklich ein Gitter, aus Konsistenzgründen wurde aber auch hier der Begriff Gitter verwendet.
<code>setHeaderHeight(int)</code> <code>getHeaderHeight()</code>	Setzt / liefert die Höhe des Headers. Diese Methoden werden hauptsächlich intern verwendet, denn die Höhe des Headers ist sowieso immer gleich der Höhe der Timeline. Der Header ist als <code>PropertyChangeListener</code> an der Timeline angehängt und bekommt auf diese Weise jede Größenänderung mit.
<code>setShowingFillerColumn(boolean)</code> <code>isShowingFillerColumn()</code>	Bestimmt, ob der Header und die Tabelle eine zusätzliche Spalte zeichnen, um den freien Raum zu füllen, der in beiden Komponenten am rechten Rand vorkommen kann. Dies ist dann der Fall, wenn der Benutzer den vertikalen Splitter so weit nach rechts verschiebt, daß der zur Verfügung stehende Platz auf der linken Seite größer ist als die Tabelle.

Popup Menü

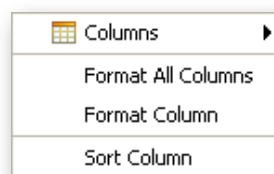
Der `TreeTableHeader` benutzt eine Implementierung des `ITreeTableHeaderMenuProvider` Interfaces um Popup Menüs dynamisch zu erzeugen.

```
JPopupMenu getMenu(TreeTableHeader header, MouseEvent e, TreeTableColumn column);
```

Auf `TreeTableHeader` sind die folgenden Methoden definiert um den Menü Provider abzufragen bzw. zu setzen:

```
getMenuProvider()
setMenuProvider(ITreeTableHeaderMenuProvider)
```

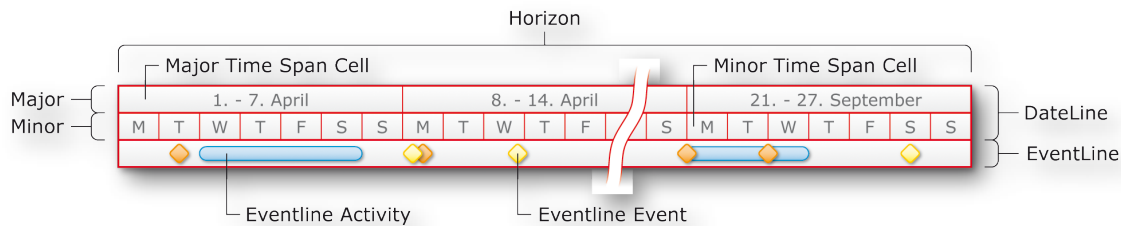
Das folgende Bild zeigt das Menü des `TreeTableHeader` wie es vom `DefaultTreeTableHeaderMenuProvider` erzeugt wird.



Default Menü vom `TreeTableHeader`

9. Timeline

Die `Timeline` Komponente dient hauptsächlich als Container für die `Dateline` und die `Eventline`. Weiterhin stellt sie zentral für beide Funktionalität für Mouse Motion Events und virtuelle Gitter zur Verfügung. Die folgende Abbildung zeigt den Aufbau der `Timeline`.



Aufbau der Timeline Komponente und Unterkomponenten

Im oberen Bereich befindet sich die *Dateline*, die für die Darstellung der Zeit verantwortlich ist. Im unteren Bereich befindet sich die *Eventline*. Diese kann benutzt werden, um globale Ereignisse und Aktivitäten anschaulich darzustellen. „Global“ bedeutet hierbei, dass es sich um Vorgänge handelt, die nicht direkt einem Objekt aus dem Baum zugeordnet werden können (z.B. die Aktivität „Betriebsferien“, die für alle Personen in einer Organisation gelten). Die folgende Tabelle zeigt die von der *Timeline* Klasse definierten Methoden.

Methode	Beschreibung
<code>getDateline()</code>	Liefert die <i>Dateline</i> Komponente zurück.
<code>getEventline()</code>	Liefert die <i>Eventline</i> Komponente zurück.
<code>getCursorLocation()</code>	Die Timeline ist als <i>MouseMotionListener</i> an der <i>Dateline</i> , der <i>Eventline</i> , und allen <i>LayerContainer</i> Instanzen registriert und erhält dadurch von verschiedenen Quellen die Position des Mouse Cursors. Von dieser Methode werden die Koordinaten des Events umgerechnet in das Koordinatensystem der Timeline zurück geliefert.
<code>getCursorGranularity()</code>	Liefert die Gitter Granularität jener Komponente zurück, von der das letzte <i>MouseEvent</i> empfangen wurde. Dies ist wichtig, da die Eventline und die LayerContainer Instanzen alle unterschiedliche Granularitäten für ihre Gitter verwenden können. Die Zeitanzeige in der Eventline muss dies berücksichtigen.
<code>getCursorGridPolicy()</code>	Liefert die verwendete <i>IGridPolicy</i> Implementierung jener Komponente zurück, von der das letzte <i>MouseEvent</i> empfangen wurde. Dies ist wichtig, da die Eventline und die LayerContainer Instanzen alle unterschiedliche Policies verwenden können. Die Zeitanzeige in der Eventline muss dies berücksichtigen.
<code>scrollLeft(boolean)</code>	Verschiebt den für den Benutzer sichtbaren Zeitbereich nach links, d.h. frühere Aktivitäten werden sichtbar. Das Verschieben kann langsam oder schnell geschehen. Bei einem langsamen Scrollen wird die Zeitleiste jedesmal um die Dauer der kleinen Zeiteinheit verschoben, ansonsten um die Dauer der grossen Einheit. Zeigt die Zeitleiste zum Beispiel Monate mit Tagen an, dann wird beim langsamen Scrollen immer um einen Tag nach links verschoben und beim schnellen Scrollen um einen ganzen Monat.
<code>scrollRight(boolean)</code>	Verschiebt den für den Benutzer sichtbaren Zeitbereich nach rechts, d.h. spätere Aktivitäten werden sichtbar. Das Verschieben kann langsam oder schnell geschehen. Bei einem langsamen Scrollen wird die Zeitleiste jedesmal um die Dauer der kleinen Zeiteinheit verschoben, ansonsten um die Dauer der grossen Einheit. Zeigt die Zeitleiste zum Beispiel Monate mit Tagen an, dann wird beim langsamen Scrollen immer um einen Tag nach rechts verschoben und beim schnellen Scrollen um einen ganzen Monat.

10.Dateline

Die *Dateline* Komponente ist verantwortlich für die Darstellung der Zeit innerhalb der *Timeline*. Der hierbei verwendete Ansatz teilt die *Dateline* in einen oberen und einen unteren Bereich auf. Im oberen Bereich wird die „Major“ und im unteren Bereich die „Minor“ Granularität dargestellt. Werden zum Beispiel im unteren Bereich Minuten dargestellt, so erscheinen im oberen Bereich Stunden. Werden unten Tage angezeigt, so erscheinen oben Tage.

Model

Die Dateline verwaltet ihr Model selbst, das heisst sie ist der Besitzer des Models und nicht der Gantt Chart. Per Default verwendet die Dateline das TimeGranularityDatelineModel. Dies ist eine Implementierung des IDatelineModel Interfaces. Eine weitere Implementierung, die bereits mit FlexGantt ausgeliefert wird, ist das SimpleGranularityDatelineModel. Der Unterschied zwischen diesen beiden Modellen liegt in dem verwendeten Granularitätstypen. Zwei Typen sind bereits in FlexGantt enthalten: TimeGranularity und SimpleGranularity. Beide Typen sind Enumeratoren und implementieren das IGranularity Interface. Während TimeGranularity normale Zeitaufösungen wie Stunden, Minuten, und Sekunden definiert, beschränkt sich SimpleGranularity auf die Definition von Mengen an Zeiteinheiten (Eins, Zehn, Hundert, Tausend). SimpleGranularity wird für gewöhnlich mehr im Forschungsumfeld eingesetzt, wo Planungsalgorithmen und -ergebnisse besser miteinander verglichen werden können, wenn sie diskrete Zeiteinheiten verwenden.

Methode	Beschreibung
<code>setModel (IDatelineModel)</code> <code>getModel ()</code>	Setzt / liefert das von der Dateline verwendete Model.

Dateline hängt sich selbst als *IDatelineModelListener* an das von ihr verwendete Model an. Hierdurch erfährt es von allen Änderungen am Model und kann entsprechend reagieren. Für gewöhnlich bedeutet „reagieren“ hier, daß sich die Dateline neu zeichnet.

Mehrere Methoden der Dateline delegieren ihre Aufrufe direkt an das Model. Hierzu gehören Methoden zum Setzen des insgesamt abgedeckten Planungshorizonts, der angezeigten (Minor) Granularität, zum Anzeigen oder zur Selektion eines bestimmten Zeitabschnitts, etc....:

Methode	Beschreibung
<code>getTimeZone ()</code> <code>setTimeZone (TimeZone)</code>	Liefert / setzt die Zeitzone, für die die Dateline Gültigkeit hat. Die Anfangs- und Endzeiten der im Gantt Chart dargestellten Aktivitäten sind angegeben in Millisekunden basierend auf GMT, Greenwich Middle Time. Das bedeutet, wenn eine andere Zeitzone als GMT von der Dateline angezeigt wird, die Aktivitätsbalken ein wenig verschoben dargestellt werden.
<code>setSelectedTimeSpan (ITimeSpan)</code> <code>getSelectedTimeSpan ()</code>	Selektiert eine Zeitspanne in der Dateline. Diese Selektion ist nur temporär und dient lediglich dazu dem Benutzer zu gestatten eine Zeitspanne für eine darauf folgende Aktion zu definieren. Per Default ist die Aktion immer ein Request, um in die ausgewählte Spanne hinein zu zoomen.
<code>getTimeLocation (long)</code>	Liefert die X-Koordinate für den angegebenen Zeitpunkt zurück. Die Berechnung erfolgt im Model und ist abhängig von der Breite der Dateline und der Anfangs- und Endzeit.
<code>setTimeSpan (ITimeSpan)</code> <code>getTimeSpan ()</code>	Liefert / setzt den von der Dateline abgedeckten Planungshorizont. Der Horizont definiert den ersten und den letzten von der Dateline angezeigten Zeitpunkt.
<code>setGranularity (IGranularity)</code> <code>getGranularity ()</code>	Liefert / setzt die von der Dateline im unteren Bereich angezeigte Granularität / Zeitauflösung (Stunden, Minuten, Sekunden,).
<code>getTimeAt (int)</code>	Liefert den Zeitpunkt an der gegebenen X-Koordinate zurück.
<code>getTimeSpanAt (Point)</code>	Liefert die Zeitspanne an der gegebenen Koordinate zurück. Hierbei ist es wichtig zu wissen, ob die Koordinate einen Punkt in der oberen oder der unteren Hälfte spezifiziert, denn die im oberen Bereich angezeigten Zeitspannen sind viel grösser als die des unteren Bereichs.
<code>getTimeSpanAt (int, boolean)</code>	Liefert die Zeitspanne an der gegebenen X-Koordinate zurück. Das mit angegebene Flag bestimmt, ob die Zeitspanne der oberen oder unteren Hälfte verlangt wird.

Methode	Beschreibung
<code>requestVisibleTimeSpan (ITimeSpan)</code>	Beantragt beim Model, daß die angegebene Zeitspanne sichtbar wird. Diese Methode delegiert an das Model, in dem dann recht komplexe Berechnungen angestellt werden müssen, um zu bestimmen welche Granularität und welcher Zoom Faktor nötig sind, um dem Request gerecht zu werden.

Renderer

Auch die Dateline verwendet das Konzept des Renderer Mappings. In diesem Fall wird der Renderer gemapped auf den Typen eines Models, also *IDatelineRenderer* Instanzen auf Unterklassen von *IDatelineModel*. Unterschiedliche Modelle können so unterschiedlich gezeichnet werden. Das *TimeGranularityDatelineModel* wird von *TimeGranularityDatelineRenderer* und *SimpleGranularityDatelineModel* wird vom *SimpleGranularityDatelineRenderer* gezeichnet. Beide Renderer sind hierbei Unterklassen von *AbstractDatelineRenderer*. In dieser abstrakten Oberklasse befindet sich sehr viel Logik zum bestimmen der richtigen Vorder- und Hintergrundfarbe für eine Zeitspanne und ihrer Zelle in der Dateline.

Methode	Beschreibung
<code>setDatelineRenderer (Class<T>, IDatelineRenderer<T>)</code> <code>getDatelineRenderer (Class<T>)</code>	Setzt / liefert einen Renderer für einen Model Typen. Die Methode stellt sicher, daß der Renderer zum angegebenen Model passt.
<code>getDatelineRenderer ()</code>	Liefert den Renderer für das aktuell von der Dateline verwendete Model zurück.
<code>paintComponent (Graphics)</code>	Standard Zeichenmethode von Swing. Ruft direkt oder indirekt die verschiedenen Zeichenmethoden der Dateline auf.
<code>paintGrid (Graphics)</code>	Zeichnet die Gitterlinien in der Dateline. Horizontal gibt es nur eine Linie zu zeichnen und zwar auf halber Höhe der Dateline. Dies trennt den oberen vom unteren Bereich, wobei der obere Bereich für die „major“ und der untere Bereich für die „minor“ Zeitspannen genutzt wird. Die X Koordinaten für die vertikalen Linien werden vom Model berechnet.
<code>paintSelectedTimeSpan (Graphics)</code>	Füllt den Hintergrund des ausgewählten Bereichs in der Zeitleiste mit der Selektionsfarbe. Der Benutzer kann einen Bereich auswählen, indem er die linke Maustaste drückt und dann eine Drag Bewegung nach links oder rechts ausführt.
<code>paintGrid (Graphics, int, int, List<GridLine>, boolean)</code>	Zeichnet die übergebenen Gitterlinien.
<code>paintCell (Graphics, ITimeSpan, int, int, int, int, boolean, boolean)</code>	Zeichnet eine Zelle innerhalb der Dateline. Die Zelle repräsentiert die übergebene Zeitspanne.
<code>paintZoomRectangle (Graphics, Rectangle)</code>	Zeichnet das Feedback während eines manuellen, vom Benutzer initiierten, Zooms.

Zoom In / Out

Das Hinein- und Herauszoomen in die Dateline bzw. aus der Dateline heraus, ist eine der wichtigsten Funktionen für den Umgang des Benutzers mit einem Planungs- und Steuerungssystems. Häufig haben die von einem Gantt Chart angezeigten Aktivitäten sehr unterschiedliche Längen, so daß ein vernünftiges Arbeiten mit den entsprechenden Balken erst bei einem Wechsel der angezeigten Granularität möglich ist.

Methode	Beschreibung
<code>zoomIn ()</code>	Zoomt in die Dateline hinein. Hierzu wird der momentan sichtbare Bereich halbiert und ein Request ausgelöst diese Hälfte sichtbar zu machen. Ein Hineinzoomen kann einen Wechsel der angezeigten Granularität zur Folge haben. Weiterhin kann sich der Zoom Faktor ändern.

Methode	Beschreibung
<code>zoomOut()</code>	Zoomt aus der Dateline heraus. Hierzu wird der momentan sichtbare Bereich verdoppelt und ein Request ausgelöst diese größere Zeitspanne sichtbar zu machen. Ein Herauszoomen kann einen Wechsel der angezeigten Granularität zur Folge haben. Weiterhin kann sich der Zoom Faktor ändern.
<code>requestVisibleTimeSpan (ITimeSpan)</code>	Fordert die <i>Dateline</i> auf eine ganz bestimmte Zeitspanne sichtbar zu machen. Der Aufruf dieser Methode hat große Auswirkungen auf den Zoomfaktor und die angezeigte Granularität.

Die Dateline Klasse bietet für das Zooming verschiedene Optionen an. So ist es z.B. möglich einen Zoom Vorgang animiert ablaufen zu lassen. Dies ist das Default Verhalten. Hierbei wird ein neuer Thread gestartet, welcher versucht, den vom Benutzer angeforderten Zeitbereich in mehreren Schritten innerhalb einer frei definierbaren Dauer (in Millisekunden) zu erreichen. Je nach Datenmenge kann dies mehr oder weniger viele Repaints bedeuten, was wiederum direkte Auswirkungen auf die Qualität der Animation hat.

Methode	Beschreibung
<code>setAnimationDuration(int)</code> <code>getAnimationDuration()</code>	Bestimmt die Länge der Animation.
<code>setAnimatingZoom(boolean)</code> <code>isAnimatingZoom()</code>	Bestimmt, ob das Zoomen animiert werden soll oder nicht, d.h. ob in mehreren Schritten oder nur in einem Schritt zu einer neuen Zeitspanne gesprungen werden soll.
<code>isAnimationThreadRunning()</code>	Bestimmt, ob in diesem Moment der Thread zum animierten Zoomen läuft.

Unterschiedliche Anwendungen haben unterschiedliche Vorstellungen darüber wie sich eine Zoom Operation auswirken soll. Manche Anwendungen bevorzugen es, wenn sich beim Zoomen der erste sichtbare Zeitpunkt nicht ändert. Andere wiederum, wollen immer in die Mitte hinein zoomen, d.h. der erste und letzte sichtbare Zeitpunkt werden verändert.³

Methode	Beschreibung
<code>setZoomStrategy (ZoomStrategy)</code> <code>getZoomStrategy()</code>	Setzt / liefert die von der <i>Dateline</i> für Zoom Vorgänge verfolgte Strategie. Entweder ändert sich beim Zoom nur die Endzeit, oder aber die Start- und die Endzeit. Die folgenden Werte sind auf <i>ZoomStrategy</i> definiert: <div style="background-color: #ffffcc; padding: 5px;"> <code>ZoomStrategy.CHANGE_VISIBLE_END_TIME</code> <code>ZoomStrategy.CHANGE_VISIBLE_START_AND_END_TIME</code> </div>

Um dem Benutzer noch mehr Kontrolle über das Zoomen zu geben, bietet die Dateline auch noch manuelles Zoomen an. Hierbei kann der Benutzer mittels eines Mouse Drags die Breite der Zellen in der Dateline verändern. Dies funktioniert allerdings nur für die oberen („major“) Zeitspannen. Um einen manuellen Zoom zu starten, muss der Benutzer den Mouse Cursor dicht an den rechten Rand einer Zelle in der Dateline bewegen. Der Mouse Cursor verändert dann seine Form und wird zum „Resize“ Cursor. Jetzt kann die linke Maustaste gedrückt und die Maus nach links oder rechts bewegt werden. Hierbei erscheint ein Rechteck, das zuerst die Grösse der Zelle hat und dann entsprechend der Mausbewegungen größer oder kleiner wird.



Manual Zoom

³ Bei einem Zoom-In werden die beiden Zeitpunkte nach aussen weggedrückt, während sie bei einem Zoom-Out nach innen wandern.

Methode	Beschreibung
<code>setManualZoomColor(Color)</code> <code>getManualZoomColor()</code>	Setzt / liefert die Farbe vom Rechteck für manuelles Zoomen zurück.
<code>setManualZoomEnabled(boolean)</code> <code>isManualZoomEnabled()</code>	Bestimmt, ob der Benutzer manuell Zoomen darf oder nicht.

Policies

Die Dateline verwendet nur eine einzige Policy namens *IZoomPolicy*. Diese wird verwendet, um zu bestimmen, welche Granularitäten für Zoom Vorgänge zur Verfügung stehen.

Methode	Beschreibung
<code>setPolicyProvider(IPolicyProvider)</code> <code>getPolicyProvider()</code>	Setzt / liefert den von der Dateline verwendeten Policy Provider.

Navigation

Methode	Beschreibung
<code>showTimeNow(boolean)</code>	Stellt sicher, daß die aktuelle Systemzeit im sichtbaren Bereich angezeigt wird. Delegiert weiter an AbstractGanttChart.
<code>showTime(long, boolean)</code>	Stellt sicher, daß der angegebene Zeitpunkt im sichtbaren Bereich angezeigt wird. Delegiert weiter an AbstractGanttChart.

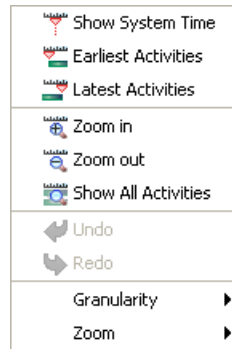
Visuelle Attribute

Auch auf der Dateline können verschiedene Attribute gesetzt werden, die das Erscheinungsbild verändern. Es sollte allerdings angemerkt werden, daß der verwendete Renderer den größten Einfluss auf das Zeichenverhalten hat.

Methode	Beschreibung
<code>setSelectionForeground(Color)</code> <code>getSelectionForeground()</code>	Setzt / liefert die Vordergrundfarbe, die von der <i>Dateline</i> verwendet wird, um den Bereich hervorzuheben, den der User per Mouse Drag selektiert hat.
<code>setSelectionBackground(Color)</code> <code>getSelectionBackground()</code>	Setzt / liefert die Hintergrundfarbe, die von der <i>Dateline</i> verwendet wird, um den Bereich hervorzuheben, den der User per Mouse Drag selektiert hat.
<code>setGridColor(Color)</code> <code>getGridColor()</code>	Setzt / liefert die Farbe, die für die Gitterlinien verwendet wird.
<code>setFocusedTimeSpan(ITimeSpan)</code> <code>getFocusedTimeSpan()</code>	Setzt / liefert die Farbe, die für den Hintergrund jener Zelle in der Dateline verwendet wird, über die sich der Mouse Cursor befindet.

Popup Menü

Die *Dateline* unterstützt das Menu Provider Konzept und benutzt per Default den *DefaultDatelineMenuProvider*.



Default Dateline Menu Provider

Methode	Beschreibung
<code>setMenuProvider (IDatelineMenuProvider)</code> <code>getMenuProvider ()</code>	Setzt / liefert den Menu Provider.

11.Eventline

Die Eventline Klasse implementiert die Leiste unterhalb der Dateline. In dieser Leiste werden zum einen verschiedene Zeit Cursor angezeigt und zum anderen sogenannte „globale“ Aktivitäten und Ereignisse. Hierunter versteht man planungsrelevante Informationen, die nicht einem bestimmten Knoten im Baum bzw. einer bestimmten Zeile in der Tabelle zugeordnet werden können. Ein typisches Beispiel sind Betriebsferien.

Model

Selection Model

Eventline Objekte Löschen

Marker

Policies

Popup Menü

```

deleteSelectedEventlineObjects()

getDateline()

getEditMode()

getEventlineObjectRenderer(Class)

getEventlineObjectsAt(int, int)

getGridColor()

getLookAheadSize()

getMarkerFillColor1()

getMarkerFillColor2()

getMenuProvider()

getModel()

getPolicyProvider()

```

getResizeHandleSize()
getSelection()
getSelectionColor()
getSelectionModel()
getTimeline()
getTimeNow()
getTopMostEventlineObjectAt(int, int)
isCreateEventlineObjectsEnabled()
isGridAdjustedMouseCursorTime()
isGridAutomatic()
isGridControlVisible()
isShowingDSTMarkers()
isTimeNowOnTop()
isTimeNowVisible()
paintComponent(Graphics)
paintDSTMarker(Graphics, GridLine)
paintEventlineObject(Graphics, Object, ITimeSpan)
paintEventlineObjects(Graphics)
paintMarker(Graphics, int, String, boolean)
selectAllEventlineObjects()
setCreateEventlineObjectsEnabled(boolean)
setEventlineObjectRenderer(Class, IEventlineObjectRenderer)
setGridAdjustedMouseCursorTime(boolean)
setGridAutomatic(boolean)
setGridColor(Color)
setGridControlVisible(boolean)
setGridGranularity(IGranularity)
setLookAheadSize(int)
setMarkerFillColor1(Color)
setMarkerFillColor2(Color)
setMenuProvider(IEventlineMenuProvider)
setModel(IEventlineModel)

setPolicyProvider(IPolicyProvider)

setResizeHandleSize(int)

setSelectionColor(Color)

setSelectionModel(IEventlineSelectionModel)

setShowingDSTMarkers(boolean)

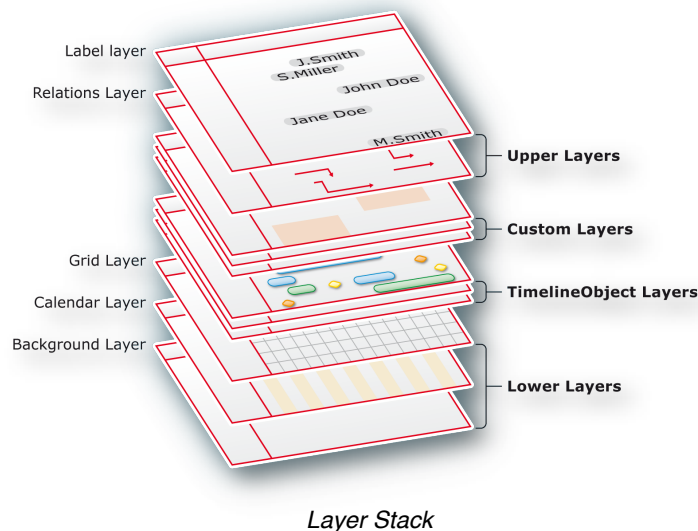
setTimeNow(long)

setTimeNowOnTop(boolean)

setTimeNowVisible(boolean)

12.LayerContainer

Die rechte Seite eines **FlexGantt** Gantt Charts besteht hauptsächlich aus einem *LayerContainer*. Dieser Container beinhaltet eine ganze Reihe von sogenannten Layers (Ebenen). Es gibt insgesamt drei Arten von Layern: die System Layers, die Timeline Object Layers, und die Custom Layers. Alle zusammen bilden einen Layer Stack. Die Timeline Object Layer sind dafür zuständig die Balken für die Aktivitäten zu zeichnen. In die Zange genommen werden sie dabei von verschiedenen System Layers. Diese zeichnen die unterschiedlichsten Dinge, die für den Benutzer beim Umgang mit dem Gantt Chart relevant sind. Hierzu gehört die aktuelle Systemzeit, Gitterlinien, etc. Custom Layers erlauben dem Anwendungsentwickler beliebige Informationen hinzuzufügen. Informationen, die sehr Applikationsspezifisch sind, und die bei der Entwicklung des Frameworks nicht vorher gesehen wurden.



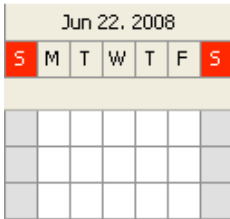


Layer Stack


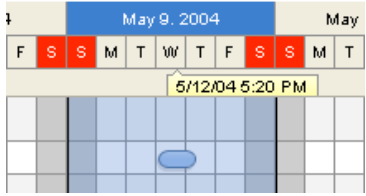
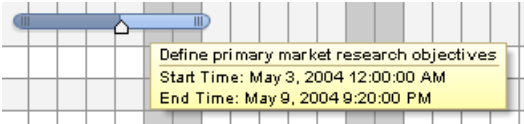
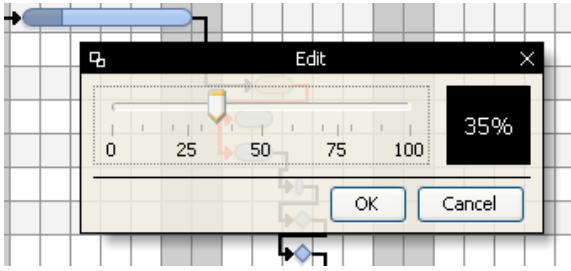
Layer Factory

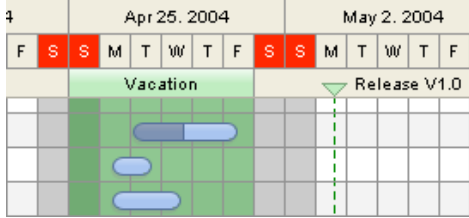
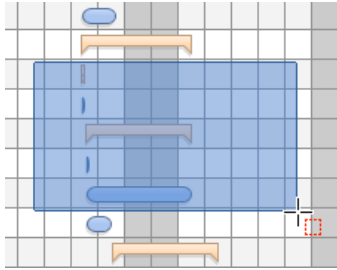
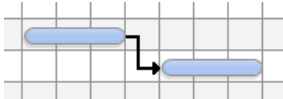
Der Konstruktor von *LayerContainer* erwartet eine *ILayerFactory* Instanz. Diese ist dafür zuständig, die einzelnen Ebenen zu erzeugen. Per Default benutzt *LayerContainer* die *DefaultLayerFactory*. Applikationen können ihre eigene Factory verwenden, um entweder Erweiterungen der Systemebenen und der Timeline Object Ebenen, oder aber um applikationsspezifische Ebenen einzufügen.

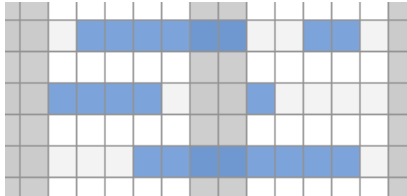
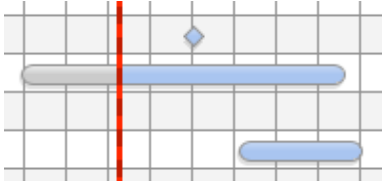
System Layers

In einem Gantt Chart gibt es Informationen, die eigentlich immer benötigt werden, damit der Benutzer vernünftig mit der Komponente arbeiten kann. Diese Informationen werden von den Systemebenen angezeigt. Die folgende Tabelle listet sämtliche Unterklassen von *AbstractSystemLayer* und beschreibt ihre Funktion. Weiterhin kann der Tabelle entnommen werden, ob sich eine Ebene oberhalb oder unterhalb der Timeline Object Ebenen befindet.

Layer / Ebene	Plazierung	Beschreibung
BackgroundLayer	Unterhalb	<p>Füllt den Hintergrund vom <i>LayerContainer</i> mit der Hintergrundfarbe. Ist eine alternierende Farbe spezifiziert, dann wird der Hintergrund von jeder zweiten Zeile mit dieser Farbe gefüllt. Dies erhöht die Lesbarkeit des Gantt Charts.</p> <pre>void LayerContainer.setBackground(Color); Color LayerContainer.getBackground(); void LayerContainer.setAlternatingBackground(Color); Color LayerContainer.getAlternatingBackground();</pre> <p>Der Layer unterstützt auch Textures. Dies sind Images, die als „Kacheln“ in den Hintergrund gezeichnet werden.</p> <pre>void BackgroundLayer.setTexture(Image img); Image BackgroundLayer.getTexture();</pre> <div>  <p><i>Normaler Hintergrund</i></p> </div> <div>  <p><i>Hintergrund mit Steinen</i></p> </div> <div>  <p><i>Hintergrund mit Grass</i></p> </div>
CalendarLayer	Unterhalb	<p>Zeichnet die Informationen aus dem Calendar Model. Wochenenden werden grau und Feiertage orange gefüllt. Der Layer benutzt Renderer für das eigentliche Zeichnen. Die Renderer werden gemapped auf den Typ der Einträge im Kalender.</p>

Layer / Ebene	Plazierung	Beschreibung
CrosshairLayer	Oberhalb	<p>Zeichnet ein Fadenkreuz an der Stelle der aktuellen Mouse Cursor Position. Das Fadenkreuz liefert zusätzliche Informationen in den vier Ecken des Kreuzes.</p>  <p><i>Fadenkreuz</i></p>
DatelineLayer	Oberhalb	<p>Zeichnet Feedback für den gerade fokusierte Zeitabschnitt.</p>  <p><i>Fokusierte Zeitspanne (Blau)</i></p>
DragLayer	Oberhalb	<p>Zeichnet Feedback für Drag & Drop Operationen. Zum einen erscheint ein kleines Fenster mit der aktuellen Drop Position, zum anderen können ganze Zeilen mit Farben gefüllt werden, die ausdrücken sollen, ob dort ein Drop möglich ist oder nicht.</p>  <p><i>Drag Info Renderer</i></p>
EditingLayer	Oberhalb	<p>Managed die sogenannten In-Place Editoren. Dies sind Editoren, die erscheinen, wenn der Benutzer einen Doppelklick auf einer Aktivität macht.</p>  <p><i>Timeline Object Editor</i></p>

Layer / Ebene	Plazierung	Beschreibung
EventlineLayer	Unterhalb	<p>Markiert die Zeitspannen und Zeitpunkte, die von globalen Aktivitäten und Meilensteinen in der Eventline belegt sind. Bei Aktivitäten geschieht dies durch das Füllen der Fläche unterhalb der Aktivität. Bei Meilensteinen wird eine vertikale gestrichelte Linie gezeichnet.</p>  <p><i>Eventline Layer</i></p>
GridLayer	Optional	Zeichnet die vertikalen Gitterlinien. Der Layer beachtet dabei den gerade eingeschalteten <i>GridLineMode</i> (keine, feine, grobe, oder kombinierte Linien).
LabelLayer	Oberhalb	Schreibt Textinformationen rechts neben Aktivitäten. Häufig benennt dieser Text die Resource, die für die Aktivität zuständig ist.
LassoLayer	Oberhalb	<p>Zeichnet das visuelle Feedback für Lasso Operationen bei denen der Benutzer versucht ein oder mehrere Aktivitäten gleichzeitig zu selektieren. Der Layer unterstützt dabei mehrere unterschiedliche Strategien, um zu entscheiden wann eine Aktivität zur Selektion gehört und wann nicht.</p>  <p><i>Lasso</i></p>
PopupLayer	Oberhalb	Zeichnet kleine gelbe Notizzettel mit Zusatzinformationen über Aktivitäten. Dies geschieht immer dann, wenn sich der Mouse Cursor über einem Objekt befindet, daß ein Popup Objekt mit sich assoziiert hat.
RelationshipLayer	Unterhalb	<p>Zeichnet die Linien zwischen Aktivitäten, die in irgendeiner Beziehung zueinander stehen.</p>  <p><i>Relationship</i></p> <pre>Collection<IRelationship> IGanttChartModel.getRelationships();</pre>
RowLayer	Unterhalb	Dieser Layer benutzt Renderer vom Typ <i>IRowRenderer</i> , um den Hintergrund einzelner Zeilen individuell anzupassen.

Layer / Ebene	Plazierung	Beschreibung
SelectionLayer	Unterhalb	<p>In einem FlexGantt Gantt Chart können sowohl Aktivitäten als auch Zeitspannen in Zeilen selektiert werden. Wenn Zeitspannen selektiert sind, dann ist diese Ebene dafür zuständig dies zu visualisieren.</p>  <p style="text-align: center;"><i>Zeitspannen Selektionen</i></p> <p>Selektionen dieser Art werden in einem separaten Selektionsmodell vom Typ <i>ILayerContainerSelectionModel</i> verwaltet.</p>
TimeNowLayer	Oberhalb	<p>Zeichnet die rot / dunkelrot gestrichelte vertikale Linie an der Stelle der aktuellen Systemzeit.</p>  <p style="text-align: center;"><i>Aktuelle Systemzeit</i></p>

Die Platzierung der Systemebenen wird durch zwei protected Methoden der *LayerContainer* Klasse bestimmt. Diese Methoden können überschrieben werden, um auf die Platzierung Einfluss zu nehmen. Ihre Default Implementierung sieht wie folgt aus:

```
/**
 * Returns those system layers that will be placed above the timeline
 * object layers and the customer layers. The system layers will be
 * added to the layer container in the order found in the returned list.
 */
protected List<Class<? extends AbstractSystemLayer>> getTypesOfUpperSystemLayers() {
    List<Class<? extends AbstractSystemLayer>> result =
        new ArrayList<Class<? extends AbstractSystemLayer>>();

    result.add(DragLayer.class);
    result.add(LassoLayer.class);
    result.add(DatelineLayer.class);
    result.add(EditingLayer.class);
    result.add(CrosshairLayer.class);
    result.add(PopupLayer.class);
    result.add(TimeNowLayer.class);
    result.add(LabelLayer.class);
    if (gantttChart.isVerticalLinesOnTop()) {
        result.add(GridLayer.class);
    }
    return result;
}

/**
 * Returns those system layers that will be placed below the timeline
 * object layers and the customer layers. The system layers will be
 * added to the layer container in the order found in the returned list.
 */
protected List<Class<? extends AbstractSystemLayer>> getTypesOfLowerSystemLayers() {
    List<Class<? extends AbstractSystemLayer>> result =
        new ArrayList<Class<? extends AbstractSystemLayer>>();
}
```




```

result.add(RelationshipLayer.class);
if (!ganttChart.isVerticalLinesOnTop()) {
    result.add(GridLayer.class);
}
result.add(EventlineLayer.class);
result.add(SelectionLayer.class);
result.add(RowLayer.class);
result.add(CalendarLayer.class);
result.add(BackgroundLayer.class);
return result;
}

```

Timeline Object Layers

Die Darstellung der Aktivitäten als Balken ist die Aufgabe der *TimelineObjectLayer* Klasse. Diese verwendet Renderer, die das eigentliche Zeichnen übernehmen. Wie üblich werden auch diese Renderer einem bestimmten Typ zugewiesen. In diesem Fall geschieht das Mapping über den Typ der Aktivitäten. Per Default sind bereits die folgenden Renderer definiert:

Objekt Typ	Timeline Object Renderer (ITimelineObjectRenderer)
Object	<p><code>com.dlsc.flexgantt.swing.layer.timeline.DefaultTimelineObjectRenderer</code></p>  <p>Dieser Renderer zeichnet einen einfachen Balken mit oder ohne abgerundeten Ecken und einem Farbverlauf von hellblau nach blau. Weiterhin wird vom Renderer ein Schatten geworfen.</p>
DefaultActivityObject	<p><code>com.dlsc.flexgantt.swing.layer.timeline.DefaultActivityObjectRenderer</code></p>  <p>Dieser Renderer ist eine Unterklasse von <code>DefaultTimelineObjectRenderer</code> und erweitert diesen mit der Fähigkeit den „percentage complete“ Wert zu visualisieren.</p>
DefaultEventObject	<p><code>com.dlsc.flexgantt.swing.layer.timeline.DefaultEventObjectRenderer</code></p>
DefaultCapacityObject	<p><code>com.dlsc.flexgantt.swing.layer.timeline.DefaultCapacityObjectRenderer</code></p>  <p>Ein ganz einfacher Renderer, der einen gefüllten vertikalen Balken zeichnet. Die Höhe des Balkens ist abhängig vom „capacity used“ Wert des Objektes.</p>

Registriert werden die Renderer auf der Klasse `LayerContainer`. Dort existieren die folgenden Methoden:

```

public void setTimelineObjectRenderer(Class objectType, ITimelineObjectRenderer renderer);
public ITimelineObjectRenderer getTimelineObjectRenderer(Class cl);

```

Eine genaue Beschreibung der Funktionsweise der Renderer für Aktivitäten befindet sich in dem Handbuch „FlexGantt - Renderer“.

Custom Layers

Selection Models

Timeline Objects Hervorheben (Highlighting)

Timeline Object Status











Popup Menü

Policies

Hilfsmethoden

13.NavigationControlPanel

Der Benutzer kann zum Navigieren im Plan das *NavigationControlPanel* benutzen. In diesem Panel befinden sich mehrere Instanzen vom Typ *NavigationControl*. Jede dieser Instanzen hat eine eigene Funktion, die der nachstehenden Tabelle entnommen werden kann.

NavigationControlType	Icon	Beschreibung
ALL_OBJECTS		Ändere den sichtbaren Bereich im Gantt Chart so ab, daß sämtliche Aktivitäten sichtbar werden. Dies kann zu einer Änderung der gerade angezeigten Granularität führen, z. B. ein Wechsel von Tagen auf Minuten.
BOOKMARKS		Zeige einen Selektor mit den von der Applikation oder dem Benutzer angelegten Lesezeichen. Die Auswahl eines Lesezeichens bewirkt eine Änderung des sichtbaren Bereichs.
EARLIEST_OBJECT		Verschiebe die Zeitleiste so, daß die am frühesten beginnenden Aktivitäten sichtbar werden.
GOTO		Springe zu einem vom Benutzer anzugebenden Zeitsprung.
GRANULARITY		Zeige einen Selektor mit den zur Verfügung stehenden Granularitäten an („Stunden“, „Minuten“,). Nachdem der Benutzer eine Granularität ausgewählt hat, wird diese von der Zeitleiste verwendet.
LATEST_OBJECT		Verschiebe die Zeitleiste so, daß die am spätesten endenden Aktivitäten sichtbar werden.
TIME_NOW		Verschiebe die Zeitleiste so, daß die aktuelle Systemzeit sichtbar wird.
TIME_SPAN		Zeige einen Selektor an, und benutze die vom Benutzer dort eingegebene Zeitspanne als neuen Horizont für die Zeitleiste.
ZOOM_IN		Zeige eine um 50% kleinere Zeitspanne im gleichen sichtbaren Bereich an. Dies führt zu einer Vergrößerung der Aktivitätsbalken.
ZOOM_OUT		Zeige eine um 50% größere Zeitspanne im gleichen sichtbaren Bereich an. Dies führt zu einer Verkleinerung der Aktivitätsbalken.

Das *NavigationControlPanel* verwendet die von *AbstractGanttChart* zur Verfügung gestellte *ISelectorFactory*, um die Selektoren zu erzeugen. Sollen andere Selektoren angezeigt werden, so muss eine Unterklasse von *DefaultSelectorFactory* angelegt und die entsprechenden *create...()* Methoden überschrieben werden. Abschliessend muss ebenfalls eine Unterklasse von *GanttChart* oder *DualGanttChart* angelegt und die *getSelectorFactory()* überschrieben werden.

Ausblenden von Controls

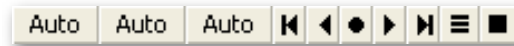
TODO

Änderung des Verhaltens einer Control

Wenn nötig kann des Standardverhalten der *NavigationControl* Instanzen verändert werden. Hierzu müssen die *ActionListener* von den Controls entfernt und neue wieder angehängt werden.

14. UtilityControlPanel

Die linke untere Ecke der *LayerContainerScrollPane*, also der rechten Seite des Gantt Charts, wird belegt vom *UtilityControlPanel*. Dieses Panel hat keine eigene Aufgabe und dient lediglich dazu verschiedene Komponenten zusammen zu fassen, die für diverse Einstellungen des Gantt Charts verwendet werden. Hierzu gehören das *GridControlPanel*, das *PagingControlPanel*, und zwei Knöpfe für den *LayerSelector* und den *OverviewSelector*. Erzeugt wird das *UtilityControlPanel* von der *DefaultComponentFactory* in der Methode *createLayerContainerCorner()*.



UtilityControlPanel

15. Statusbar

Die in **FlexGantt** enthaltene Klasse *GanttChartStatusBar* liefert in knapper Form wichtige Informationen über den Zustand eines Gantt Charts. Einige wichtige Methoden dieser Klasse sind definiert in der Oberklasse *StatusBar*. Die folgende Tabelle enthält Beschreibungen dieser Methoden.












Methode	Beschreibung
<code>addItem(JComponent)</code>	Fügt eine Komponente hinzu. Die Komponente wird allerdings nicht direkt der <i>StatusBar</i> Instanz hinzugefügt, sondern einer Unterkomponente. Hierbei handelt es sich um ein <i>JPanel</i> , welches ihre Kinder horizontal von links nach rechts anordnet. Per Default sind die meisten hinzugefügten Items <i>JLabel</i> Instanzen, die zumeist nur ein Icon anzeigen.
<code>addSeparator()</code>	Fügt einen Separator hinzu. Für gewöhnlich ist dies ein einfacher vertikaler Strich.
<code>removeAllItems()</code>	Entfernt sämtliche Items aus der <i>StatusBar</i> .
<code>getItemPanel()</code>	Liefert das <i>JPanel</i> zurück, das die Items enthält.
<code>getResizeIconLabel()</code>	In der rechten unteren Ecke der Statusbar befindet sich ein <i>JLabel</i> , mit dessen Hilfe man das Fenster in dem sich der Gantt Chart befindet vergrößern oder verkleinern kann. In diesem <i>JLabel</i> wird das von Windows bekannte Resize Icon angezeigt.
<code>setStatus(String)</code>	Setzt den Status Text in der Statuszeile. Der Text wird immer auf der linken Seite in einem <i>JLabel</i> angezeigt.
<code>getStatus()</code>	Liefert den Status Text zurück.
<code>getStatusLabel()</code>	Liefert die <i>JLabel</i> Instanz zurück, die für das Anzeigen des Status Textes zuständig ist.

Diese Methoden sind sehr generisch. Gantt Chart spezifische Methoden befinden sich in der Unterklasse *GanttChartStatusBar*.

Methode	Beschreibung
<code>getActionSelectorLabel()</code>	Liefert das <i>JLabel</i> zurück, auf das der Benutzer klicken kann, um den <i>ActionSelector</i> anzeigen zu lassen. Der <i>Selector</i> enthält eine Auflistung aller beim Gantt Chart registrierten Actions / Key Strokes.
<code>getCrosshairLabel()</code>	Liefert das <i>JLabel</i> zurück, das anzeigt, ob das Fadenkreuz eingeschaltet ist.
<code>getGanttChart()</code>	Liefert den Gantt Chart zurück für den der Statusbar gerade benutzt wird. Der gleiche Statusbar kann nämlich für mehrere Gantt Charts verwendet werden, aber immer nur einen zur gleichen Zeit.
<code>getGridLabel()</code>	Liefert das <i>JLabel</i> zurück, das anzeigt, ob Gitterlinien eingeschaltet sind.

Methode	Beschreibung
<code>getMemoryLabel()</code>	Liefert das JLabel zurück, das den aktuellen Speicherverbrauch anzeigt. Der Benutzer kann durch einen Klick auf das JLabel explizit eine Garbage Collection auslösen. Für die Dauer der Garbage Collection wird ein anderer Text angezeigt.
<code>getMessagesLabel()</code>	Liefert das JLabel zurück, das anzeigt, ob Statusmeldungen vorliegen (Informationen, Warnungen, Fehler).
<code>getPopupLabel()</code>	Liefert das JLabel zurück, das anzeigt, ob Popups eingeschaltet sind.
<code>getTimeLabel()</code>	Liefert das JLabel zurück, das die Uhrzeit an der aktuellen Mouseposition anzeigt.
<code>getTimeNowLabel()</code>	Liefert das JLabel zurück, das die aktuelle Systemzeit anzeigt.
<code>getTimeZoneLabel()</code>	Liefert das JLabel zurück, das die von der Dateline verwendete Zeitzone anzeigt.

Der *GanttChartStatusBar* implementiert das *IMultiGanttChartContainerListener* Interface. Das bedeutet, daß der Statusbar sowohl für einen einzelnen als auch für mehrere Gantt Charts gleichzeitig verwendet werden kann. Dies allerdings nur dann, wenn sie sich in einem *IMultiGanttChartContainer* befinden. Wann immer der Benutzer den Gantt Chart wechselt, passt sich der *StatusBar* an, indem er die *IStatusBarPolicy* von neuen Gantt Chart abfragt und seinen Inhalt entsprechend dieser Policy aufbaut (welche Felder im *StatusBar* angezeigt werden bestimmt die Policy). Die folgende Tabelle liefert eine Übersicht, über die vom *GanttChartStatusBar* unterstützten Felder. Definiert sind die Felder im Enumerator *StatusBarField*.




StatusBarField	Icon	Beschreibung
CROSSHAIR		Zeig den Status des Fadenkreuzes an.
GRID		Zeig den Status der Gitterlinien an. Der Tooltip auf diesem Feld zeigt zusätzlich an, ob das grobe oder das feine Gitter eingeschaltet ist.
KEY_STROKES		Durch einen Klick auf dieses Feld wird ein Selektor sichtbar, der eine Übersicht über die am Gantt Chart registrierten Key Strokes / Actions liefert.
MEMORY		Zeig den aktuellen Speicherverbrauch an. Ein Klick auf das Feld löst eine Garbage Collection aus.
MESSAGE		Zeig an, ob Statusmeldungen vorliegen. Ist mindestens eine der Meldungen eine Fehlermeldung, dann wird das Fehler Icon angezeigt. Liegt keine Fehlermeldung vor aber mindestens eine der Meldungen ist eine Warnung, dann wird das Warnung Icon angezeigt. Liegen keine Fehler und keine Warnungen vor, dann wird das Informations Icon angezeigt.
POPUP		Popups an / aus.
PROGRESS_BAR		Zeig einen Progress Bar an, der u.a. dafür verwendet werden kann den Fortschritt von Commands anzuzeigen.
TIME		Zeig den Zeitpunkt an der aktuellen Cursorposition an.
TIME_NOW		Zeig die aktuelle Systemzeit an.
TIME_NOW_LOCK		Gib dem Benutzer die Möglichkeit automatisches Scrollen ein- / und auszu-schalten.
TIME_ZONE		Gib dem Benutzer die Möglichkeit die Zeitzone zu ändern.

16. GanttChartToolBar

Die in **FlexGantt** enthaltene *GanttChartToolBar* Komponente bietet die Möglichkeit auf schnelle und einfache Weise dem Gantt Chart eine Buttonleiste mit grundlegenden Funktionen zur Steuerung hinzuzufügen. Allerdings bedeutet dies nicht, daß diese Buttonleiste allen Ansprüchen gerecht wird. Dazu sind die Anforderungen an Toolbars von Applikation zu Applikation viel zu unterschiedlich. Es gibt ganze Frameworks, die sich nur damit beschäftigen. Die *GanttChartToolBar* Klasse soll lediglich dazu dienen die Anfangsphase bei der Applikationsentwicklung zu überbrücken, anschliessend muss entweder in weitere Frameworks oder in Eigenentwicklungen investiert werden.

GanttChartToolBar erlaubt es ähnlich wie *GanttChartStatusBar* zu definieren, welche Element in ihr angezeigt werden sollen. Bei der Toolbar geschieht dies allerdings nicht mittels einer Policy, sondern zur Konstruktionszeit. Dem Konstruktor muss eine *Collection* von Kontrollelementen übergeben werden. Diese sind im *Control* Enumerator definiert. Die folgende Tabelle listet auf, welche Elemente zur Verfügung stehen, welche Icons in den für sie erzeugten Buttons sichtbar sind, und welche der Buttons sogenannte Toggles sind. Toggle Buttons bleiben eingedrückt, nachdem der Benutzer auf sie geklickt hat.

Control	Icon	Toggle	Beschreibung
ALIGN_END_TIMES		Nein	Setzt die Endzeit der aktuell selektierten Aktivitäten auf den spätesten benutzten Zeitpunkt der Aktivitäten.
ALIGN_START_TIMES		Nein	Setzt die Startzeit der aktuell selektierten Aktivitäten auf den frühesten benutzten Zeitpunkt der Aktivitäten.
CROSSHAIR		Ja	Schaltet das Fadenkreuz ein oder aus.
EARLIEST_OBJECTS		Nein	Verändert den sichtbaren Zeitbereich so ab, daß die am frühesten beginnenden Aktivitäten sichtbar werden.
EVENTLINE		Ja	Schaltet die Eventline ein oder aus.
GRID		Ja	Schaltet das Gitter ein oder aus.
LATEST_OBJECTS		Nein	Verändert den sichtbaren Zeitbereich so ab, daß die am spätesten endenden Aktivitäten sichtbar werden.
POPUP		Ja	Schaltet Notizen ein oder aus.
POPUP_CLEAR		Nein	Entfernt alle fixierten Notizen.
PRINT		Nein	Druckt den Gantt Chart.
PRINT_PREVIEW		Nein	Zeigt eine Druckvorschau.
REDO		Nein	Wiederholt den zuletzt rückgängig gemachten Befehl.
RELATIONS		Ja	Zeigt oder verbirgt Beziehungslinien.
SHOW_ALL		Nein	Verändert den sichtbaren Zeitbereich so ab, daß alle Aktivitäten sichtbar werden.
SPLIT		Ja	Splittet den Gantt Chart in zwei Teile (nur beim DualGanttChart).
TIME_NOW		Nein	Verschiebt den sichtbaren Zeitbereich, so daß der aktuelle Systemzeitpunkt sichtbar wird.
TREE_COLLAPSE_ALL		Nein	Schliesst sämtliche Baumknoten.
TREE_EXPAND_ALL		Nein	Öffnet sämtliche Baumknoten.

Control	Icon	Toggle	Beschreibung
UNDO		Nein	Macht den zuletzt ausgeführten Befehl rückgängig.
ZOOM_IN		Nein	Zeigt einen kleineren Zeitbereich an. Dies hat zur Folge, daß die sichtbaren Aktivitäten größer werden.
ZOOM_OUT		Nein	Zeigt einen größeren Zeitbereich an. Dies hat zur Folge, daß die sichtbaren Aktivitäten kleiner werden.

Die Klasse *GanttChartFrame* erzeugt automatisch eine Instanz vom Typ *GanttChartToolBar*. Dieser Toolbar hat dann sämtliche Kontrollelemente in sich, die es gibt. Um dieses Verhalten zu verändern, muß lediglich die Methode *createToolBar()* in einer Unterklasse von *GanttChartFrame* überschrieben werden.

Genau wie die Statuszeile kann auch die Toolbar mit mehreren Gantt Charts gleichzeitig verwendet werden, falls sich diese in einem Container befinden, der das *IMultiGanttChartContainer* Interface implementiert. Ist dies der Fall wird die Toolbar immer dann informiert, wenn der Benutzer von einem Gantt Chart zu einem anderen wechselt, z. B. durch einen Klick auf einen Reiter in einer *JTabbedPane*⁴.

17.Row Headers

Die Klasse *JScrollPane* erlaubt es einen sogenannten „Row Header“ links von der zu scrollenden Komponente zu platzieren. In **FlexGantt** wird dieses Feature benutzt, um sowohl neben der *TreeTable* als auch neben dem *LayerContainer* Komponenten zu zeigen, die das Arbeiten mit der jeweiligen Komponente möglichst sinnvoll unterstützen. Erzeugt werden diese beiden Header von den Methoden *createTreeTableRowHeader()* und *createLayerContainerRowHeader()* in der Component Factory.

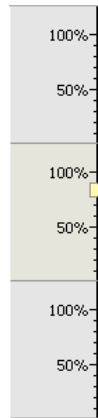
Neben der *TreeTable* befindet sich eine Instanz vom Typ *TreeTableRowHeader*, die Zeilennummern und Icons zum öffnen und schliessen von Baumknoten darstellt. Gezeichnet werden diese von der Klasse *DefaultTreeTableRowHeaderRenderer*, die das *ITreeTableRowHeaderRenderer* Interface implementiert.

0	[-]
1	[-]
2	[+]
22	[-]
23	[-]
24	[-]
25	
26	
27	
28	[-]

TreeTableRowHeader

Neben dem *LayerContainer* befindet sich der *LayerContainerRowHeader*. Dieser zeigt Kapazitätslinien und deren Beschriftung an, welche von der Klasse *DefaultLayerContainerRowHeaderRenderer* gezeichnet werden. Dieser Renderer implementiert das *ILayerContainerRowHeaderRenderer* Interface.

⁴ Dies ist der Fall bei der Klasse *MultiGanttChartContainer*, die eine *JTabbedPane* verwendet, um mehrere Gantt Charts anzuzeigen.



LayerContainerRowHeader

Beide Row Header Klassen benutzen den Typ des „Row Header Value“, um den richtigen Renderer zum Zeichnen zu bestimmen. Geliefert wird dieser Wert für jeden Baumknoten vom Model. Hierzu existiert auf *ITreeTableModel* die folgende Methode:

```
Object getRowHeaderValue(T node);
```

Für gewöhnlich delegiert die Implementierung diese Methode weiter auf die eigentliche Knotenklasse, welche meistens eine Unterklasse von *DefaultMutableTreeTableNode* ist. Dort wird die Methode *getRowHeaderValue()* aufgerufen, die wie folgt implementiert ist:

```
public Object getRowHeaderValue() {
    // This method is not allowed to return NULL.
    if (rowHeaderValue != null) {
        return rowHeaderValue;
    }
    return this;
}
```

Wie man sieht ist ein separater „Row Header Value“ nicht unbedingt erforderlich. Notfalls liefert sich der Knoten selbst zurück. Da die beiden Renderer *DefaultTreeTableRowHeaderRenderer* und *DefaultLayerContainerRowHeaderRenderer* auf *Object* gemapped sind, werden diese dann zum Zeichnen verwendet.

Ein Beispiel für den praktischen Einsatz eines Row Header Values ist die Anzeige einer Tasknummer im Row Header der *TreeTable*. Tasknummern erscheinen nicht immer durchgängig, sondern sind davon abhängig, welche Knoten gerade geschlossen sind und welche nicht. So kann es zum Beispiel vorkommen, daß in der Zeile 20 ein Task mit der Nummer 85 angezeigt wird. Ein Integer mit dem Wert 85 wäre ein möglicher Rückgabewert der *getRowHeaderValue()* Methode und eine Implementierung des *ITreeTableRowHeaderRenderer* Interfaces müßte auf den Typ *Integer* gemapped werden.

Weitere Informationen zum Thema Renderer befinden sich im Handbuch „FlexGantt - Renderer“.

18. Menü Provider

Menü Provider sind Objekte, welche eingesetzt werden, um dynamisch Popup Menüs für Komponenten zu erzeugen. Dynamisch heisst, daß der Provider jedes mal dann aufgerufen wird, wenn der Benutzer den „Popup Trigger“ auf der Komponente anwendet. In der Regel ist dies ein Klick mit der rechten Maustaste.

Popup Menüs dynamisch zu erzeugen hat den Vorteil, daß der aktuelle Zustand der Komponente mit berücksichtigt werden kann. Ist zum Beispiel in einer *TreeTable* der Baumknoten in der angeklickten Zeile geöffnet, dann macht es keinen Sinn den Menüeintrag zum Öffnen des Knotens aktiv zu haben.

Beim Aufruf eines Menü Providers werden bereits die wichtigsten Parameter mit übergeben. Hierzu gehört immer die Komponente zu der das Popup Menü gehört und das Mouse Event, das als „Popup Trigger“ gedient hat. Die weiteren Argumente sind von der jeweiligen Komponente abhängig.

Als Beispiel ist hier das *ITreeTableMenuProvider* Interface angegeben. Es verlangt die *TreeTable*, das Mouse Event, einen *TreePath*, und eine *TreeTableColumn*. Der *TreePath* zeigt auf die Zeile auf die der User geklickt hat und die *TreeTableColumn* auf die Spalte.

```
JPopupMenu getPopupMenu(TreeTable table, MouseEvent e, TreePath treePath, TreeTableColumn column);
```

Auf der *TreeTable* sind die folgenden Methoden definiert um den Menü Provider abzufragen bzw. zu setzen:

```
getMenuProvider()
setMenuProvider(ITreeTableMenuProvider)
```

In der Praxis hat es sich als vorteilhaft erwiesen, wenn der Menü Provider eine Unterklasse von *JPopupMenu* ist. Diese Unterklasse muss dann nur die einzelne Interfacemethode implementieren und sich selbst als Ergebnis zurück liefern. Als Beispiel hier ein Ausschnitt aus dem Standard Menü Provider für die *TreeTable*:

```
public class DefaultTreeTableMenuProvider extends JPopupMenu implements
    ITreeTableMenuProvider {

    public JPopupMenu getPopupMenu(TreeTable table, MouseEvent e,
        TreePath treePath, TreeTableColumn column) {
        removeAll();
        add(new SelectAllNodesAction(table));
        add(new ClearSelectionAction(table));
        add(new JSeparator());
        add(new ExpandNodeAction(table));
        add(new CollapseNodeAction(table));
        add(new JSeparator());
        add(new IndentNodeAction(table));
        add(new OutdentNodeAction(table));
        add(new JSeparator());
        add(new UndoActionLabeled<AbstractGanttChart>(table.getGanttChart()));
        add(new RedoActionLabeled<AbstractGanttChart>(table.getGanttChart()));
        add(new JSeparator());
        int row = table.getRowForPath(treePath);
        if (row == -1) {
            row = Math.max(0, table.getRowCount() - 1);
        }
        add(new InsertNodeAction(table, row));
        add(new DeleteNodeAction(table));
        return this;
    }
}
```

Menü Provider haben sich in der Praxis als äußerst praktische Objekte erwiesen, die den Umgang mit Popup Menüs stark vereinfachen.

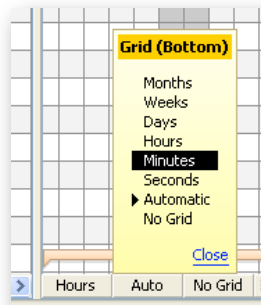
19.Grid Components

Beim bearbeiten von Plänen in einem Gantt Chart ist es hilfreich, wenn der Benutzer ein virtuelles Gitter benutzen kann, an dem er Aktivitäten ausrichtet. In **FlexGantt** findet man dieses Feature bei der *Eventline* und beim *LayerContainer*. Beide Klassen implementieren das *IGridComponent* Interface. Dieses Interface macht es möglich eine Granularität zu setzen, die die Grundlage für das Gitter bildet. Wird zum Beispiel die Granularität auf *TimeGranularity.WEEK* gesetzt, dann fangen Aktivitäten nach einer Verschiebeoperation immer am Anfang einer Woche an. Wird nur die Endzeit verändert, so wird diese immer auf das Ende einer Woche fallen.

Methode	Beschreibung
<code>getGridGranularity()</code> <code>setGridGranularity(IGranularity)</code>	Setzt / liefert die von der Komponente verwendete Gitter Granularität. Das Setzen erzeugt ein Event, das an die entsprechenden Listener gesendet wird.

Grid Control

Für jede *IGridComponent* Instanz in einem Gantt Chart wird eine *GridControl* erzeugt und dem *UtilityControlPanel* hinzugefügt. Die *GridControl* ermöglicht es dem Benutzer die Granularität des Gitters festzulegen. Welche Granularitäten zur Auswahl stehen, kann der jeweiligen *IGridPolicy* entnommen werden. Jede Grid Component hat ihre eigene Policy. Die folgende Abbildung zeigt die drei GridControls des *DualGanttChart*. Die erste Control hat die Gitter Granularität auf Stunden gesetzt, die zweite verwendet den „Auto“ Modus, die dritte benutzt gar kein Gitter. Beim Klicken auf die zweite Control wurde der entsprechende Selector geöffnet, so daß der Benutzer eine neue Granularität auswählen kann.



Grid Control

Methode	Beschreibung
<code>getGridComponentName()</code>	Liefert den Namen der Grid Komponente („Eventline“, „Layer Container 1“, „Layer Container 2“). Der Name wird als Titel in der Grid Control angezeigt.
<code>isGridControlVisible()</code>	Bestimmt, ob die Grid Control für die Grid Component angezeigt werden soll oder nicht. Dies ist abhängig davon, ob die Applikation dem Benutzer das Setzen des Gitters erlaubt oder nicht.

Automatisches Gitter

Auch ein automatisches Gitter wird von FlexGantt unterstützt. Ist dieses eingeschaltet, so wird immer die gerade angezeigte Granularität der Dateline für das virtuelle Gitter verwendet. Werden zum Beispiel gerade Tage angezeigt, so fangen Aktivitäten nach einer Verschiebung immer am Anfang eines Tages an. Wenn das automatische Gitter aktiv ist, wird die Granularität, die von der `getGridGranularity()` Methode zurück geliefert wird, vollständig ignoriert.

Methode	Beschreibung
<code>setGridAutomatic(boolean)</code> <code>isGridAutomatic()</code>	Schaltet das automatische Gitter ein oder aus.

Policies

Jede `IGridComponent` Instanz kann ihre eigene `IGridPolicy` verwenden. Neben anderen Dingen kann durch die Policy genau gesteuert werden, welche Granularitäten der Benutzer zur Auswahl bekommt.

Methode	Beschreibung
<code>getGridPolicy()</code>	Liefert die <code>IGridPolicy</code> zurück.

Listener und Events

Wann immer der Benutzer die Einstellungen der `IGridComponent` Instanzen ändert, ist es wichtig, daß interessierte Parteien davon erfahren. Hierfür existiert das `IGridComponentListener` Interface. Objekte, die dieses Interface implementieren erhalten ein Event bei jeder Änderung am Gitter der Komponente. Das Interface selber ist extrem einfach und besteht nur aus einer Methode:

```
/**
 * Callback method that gets invoked when the grid granularity used by a
 * grid component changes.
 *
 * @since 1.0
 */
void gridComponentChanged();
```

Methode	Beschreibung
<code>addGridComponentListener(IGridComponentListener)</code> <code>removeGridComponentListener(IGridComponentListener)</code>	Fügt einen Listener hinzu bzw. entfernt ihn wieder.

Zugriff

Sämtliche *GridControl* Instanzen werden erzeugt von und befinden sich im *GridControlPanel*. Zur Konstruktionszeit des Panels ruft dieses die Methode *AbstractGanttChart.getGridComponents()* auf, um herauszufinden, für wieviele *IGridComponent* Instanzen es *GridControl* Instanzen erzeugen muss. Das *GridControlPanel* selber ist eine Unterkomponente des *UtilityControlPanel*. Letzteres wird von der Component Factory des Gantt Charts als „linke untere Ecke“ für die *LayerContainerScrollPane* erzeugt.

Das bedeutet, daß eine Möglichkeit des Zugriffs auf eine *GridControl* der Moment ist, wo diese als Teil des *UtilityControlPanel* in der Factory erzeugt wird. Dazu muss die Methode *createLayerContainerCorner()* in einer Unterklasse der *DefaultComponentFactory* überschrieben werden. Anschliessend muss die Unterklasse dem Konstruktor des Gantt Charts übergeben werden.

Eine weitere Möglichkeit existiert, nachdem der Gantt Chart erzeugt wurde. Dann kann von der *LayerContainerScrollPane* die linke untere Ecke abgefragt werden. Das zurück gelieferte Objekt muss dann allerdings nach *UtilityControlPanel* gecasted werden. Anschliessend kann man von dem Panel das *GridControlPanel* abfragen und von diesem wiederum die einzelnen *GridControl* Instanzen.

```
LayerContainerScrollPane pane = myGanttChart.getLayerContainerScrollPane();
UtilityControlPanel utility = (UtilityControlPanel) pane.getCorner(JScrollPane.LOWER_LEFT_CORNER);
GridControlPanel controlPanel = utilityPanel.getGridControlPanel();
GridControl gridControl = controlPanel.getGridControl(myEventline);
```

20.MultiGanttChartContainer

Viele Planungs- und Steuerungssysteme arbeiten mit mehreren Plänen gleichzeitig, wobei es sehr unterschiedliche Kriterien geben kann, die die Aufteilung bestimmen. Kriterien können sein: Standort, Produktreihe, Team, usw. Häufig ist aber die gleiche Person zuständig für die Planung. Um diesem Planer Zugriff auf alle Pläne gleichzeitig zu ermöglichen kann die *MultiGanttChartContainer* Klasse verwendet werden. Diese Klasse ist eine Unterklasse von *JPanel* und benutzt intern ein *JTabbedPane*. Für jeden Gantt Chart der diesem Container hinzugefügt wird, zeigt das *JTabbedPane* einen eigenen Reiter an.

MultiGanttChartContainer implementiert das *IMultiGanttChartContainer* Interface. Hierdurch ist es möglich, daß nur eine *GanttChartToolBar* und eine *GanttChartStatusBar* Instanz nötig sind, um mit allen Gantt Charts zu arbeiten.

Damit Applikationen herausfinden können, welcher Gantt Chart gerade aktiv ist, können Listener vom Typ *IMultiGanttChartContainerListener* an den Container angehängt werden. Diese Listener erhalten Events sobald ein Gantt Chart hinzugefügt, eingefügt, entfernt, oder aktiviert wird.