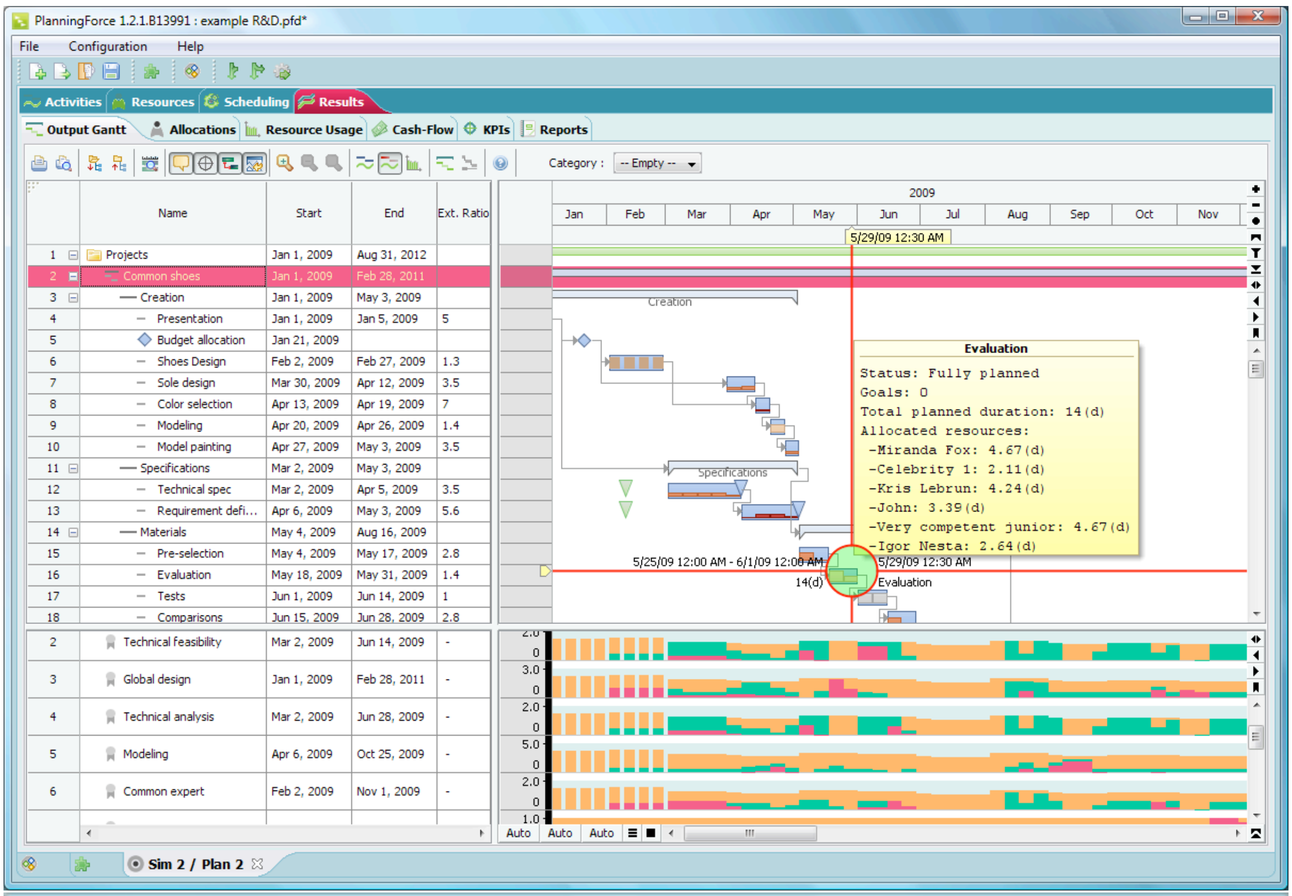
FlexGantt - Release 1

Policies & Commands



	Name	Start	End	Ext. Ratio
1	Projects	Jan 1, 2009	Aug 31, 2012	
2	Common shoes	Jan 1, 2009	Feb 28, 2011	
3	Creation	Jan 1, 2009	May 3, 2009	
4	Presentation	Jan 1, 2009	Jan 5, 2009	5
5	Budget allocation	Jan 21, 2009		
6	Shoes Design	Feb 2, 2009	Feb 27, 2009	1.3
7	Sole design	Mar 30, 2009	Apr 12, 2009	3.5
8	Color selection	Apr 13, 2009	Apr 19, 2009	7
9	Modeling	Apr 20, 2009	Apr 26, 2009	1.4
10	Model painting	Apr 27, 2009	May 3, 2009	3.5
11	Specifications	Mar 2, 2009	May 3, 2009	
12	Technical spec	Mar 2, 2009	Apr 5, 2009	3.5
13	Requirement defi...	Apr 6, 2009	May 3, 2009	5.6
14	Materials	May 4, 2009	Aug 16, 2009	
15	Pre-selection	May 4, 2009	May 17, 2009	2.8
16	Evaluation	May 18, 2009	May 31, 2009	1.4
17	Tests	Jun 1, 2009	Jun 14, 2009	1
18	Comparisons	Jun 15, 2009	Jun 28, 2009	2.8

2	Technical feasibility	Mar 2, 2009	Jun 14, 2009	-
3	Global design	Jan 1, 2009	Feb 28, 2011	-
4	Technical analysis	Mar 2, 2009	Jun 28, 2009	-
5	Modeling	Apr 6, 2009	Oct 25, 2009	-
6	Common expert	Feb 2, 2009	Nov 1, 2009	-

```
Evaluation
Status: Fully planned
Goals: 0
Total planned duration: 14(d)
Allocated resources:
 -Miranda Fox: 4.67(d)
 -Celebrity 1: 2.11(d)
 -Kris Lebrun: 4.24(d)
 -John: 3.39(d)
 -Very competent junior: 4.67(d)
 -Igor Nesta: 2.64(d)
```

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

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

FlexGantt Release 1.1.2

Document last updated on October 1st, 2008

Table of Content

Policies

The primary purpose of policies is to help components find out how to behave in certain situations. For example: the user wants to select an activity. The *TimelineObjectLayer*, which handles the display and the selection of activities, can now invoke methods on a specialized policy in order to find out, whether the activity is selectable at all. The policy helps the layer to make a decision. That‘s what policies are: they are lightweight helper classes in the decision making process and control flow of a component.

Policies are also used for looking up data that is not available in a model. The *PopupLayer*, for example, queries the popup text for a timeline object from a policy of type *IPopupPolicy*. The popup information could have been added to a model, but would have resulted in a bloated model API.

Another important use for policies is the lookup of commands. Commands are used to modify models. In most cases simply updating the model is not enough. Applications often need to make calls to a backend to update the data source (e.g. a database). Application developers can customize policies and return their own commands. These commands are often subclasses of the default commands and add the required action to update the data source.

Lookup & Registration

Policies can be retrieved from policy providers. These are objects, which implement the *IPolicyProvider* interface. The default implementation of this interface is the *PolicyProvider* class, which gets used by all policy-controlled components in ***FlexGantt***. The interface looks like this:

```
<T extends IPolicy> void IPolicyProvider.setPolicy(Class<T> policyType, T policyImpl);
<T extends IPolicy> T IPolicyProvider.getPolicy(Class<T> policyType);
void IPolicyProvider.addPolicyProviderListener(IPolicyProviderListener l);
void IPolicyProvider.removePolicyProviderListener(IPolicyProviderListener l);
```

The provider uses the policy interface class as the key for retrieving and specifying policy implementations. This approach guarantees that the provider will always return an object, which actually implements the required policy interface. The policy provider of each component gets configured in the component‘s constructor. We find the following lines of code in the constructor of the *LayerContainer* class.

```
policyProvider.setPolicy(IDragAndDropPolicy.class, new DefaultDragAndDropPolicy());
policyProvider.setPolicy(IEditTimelineObjectPolicy.class, new DefaultEditTimelineObjectPolicy());
policyProvider.setPolicy(IEditActivityObjectPolicy.class, new DefaultEditActivityObjectPolicy());
policyProvider.setPolicy(IEditCapacityObjectPolicy.class, new DefaultEditCapacityObjectPolicy());
policyProvider.setPolicy(ILabelPolicy.class, new DefaultLabelPolicy());
policyProvider.setPolicy(IRelationshipPolicy.class, new DefaultRelationshipPolicy());
policyProvider.setPolicy(IPopupPolicy.class, new DefaultPopupPolicy());
policyProvider.setPolicy(ISelectionPolicy.class, new DefaultSelectionPolicy());
policyProvider.setPolicy(ICrosshairPolicy.class, new DefaultCrosshairPolicy());
policyProvider.setPolicy(IOverviewPolicy.class, new DefaultOverviewPolicy());
policyProvider.setPolicy(ILinePolicy.class, new DefaultLinePolicy());
policyProvider.setPolicy(IEditLayerPolicy.class, new DefaultEditLayerPolicy());
policyProvider.setPolicy(IGridPolicy.class, new TimeGranularityGridPolicy());
policyProvider.setPolicy(IGridLinePolicy.class, new TimeGranularityGridLinePolicy());
policyProvider.setPolicy(IDragInfoPolicy.class, new TimeGranularityDragInfoPolicy());
```

The following code can be used to replace one of these policies:

```
LayerContainer lc = myGantt.getLayerContainer(); // myGantt is an instance of GanttChart
IPolicyProvider pp = lc.getPolicyProvider();
pp.setPolicy(IPopupPolicy.class, new MyPopupPolicy());
```

The policy interfaces and their default implementations are located in packages that map to the components where they are being used:

```
com.dlsc.flexgantt.policy.dateline          // Dateline policies
com.dlsc.flexgantt.policy.eventline         // Eventline policies
com.dlsc.flexgantt.policy.gantt             // AbstractGanttChart policies
com.dlsc.flexgantt.policy.layer             // LayerContainer policies
com.dlsc.flexgantt.policy.treetable         // TreeTable policies
```

Creating New Policies

How much a developer gets into contact with policies can be very different from one application to another. Some applications have no need for modifying the already existing default policy implementations. Some need to subclass one or more of the default policies in order to fine-tune them to their needs. And yet another class of applications might even introduce their own policies. These new policies have to implement the *IPolicy* interface, which is an empty interface.

New policies should always subclass the *AbstractPolicy* class. This class provides its own assert functionality, which is very much needed when implementing policies. The reason for this are the very generic signatures of policy methods. In many cases the arguments are of type *java.lang.Object*. The *assertClass()* method of *AbstractPolicy* can be used to check whether an object is of a more specific type. The method looks like this:

```
	public void assertClass(String methodName, String argName, Class<?> cl,
			Object obj) {
		if (methodName == null) {
			throw new IllegalArgumentException("method name can not be NULL");
		}
		if (argName == null) {
			throw new IllegalArgumentException("argument name can not be NULL");
		}
		if (cl == null) {
			throw new IllegalArgumentException("class can not be NULL");
		}
		if (obj == null) {
			throw new IllegalArgumentException("object can not be NULL");
		}
		if (!cl.isAssignableFrom(obj.getClass())) {
			String policyName = getClass().getName();
			throw new IllegalArgumentException("the policy of type "
					+ policyName + " expected the argument '" + argName
					+ "' of method '" + methodName + "' to be of type "
					+ cl.getName() + " but it was of type "
					+ obj.getClass().getName() + "!");
		}
	}
```

The method tries to make sure that the object „obj“ passed to the method „methodName“ for the argument „argName“ is of type „cl“. If it isn‘t the method throws an exception.

The following is the implementation of the *ILinePolicy*[1] used by the *LayerContainer*. It shows the *assertClass()* method in action. One can see, that the policy methods accept *java.lang.Object* as the type for the tree table node argument. The *assertClass()* method ensures that the node implements the *IGanttChartNode* interface, in which case the policy can delegate to the matching methods of the node.

```
public class DefaultLinePolicy extends AbstractPolicy implements ILinePolicy {

	public int getLineCount(Object node, ITreeTableModel model) {
		assertClass("getLineCount", "node", IGanttChartNode.class, node);
		return ((IGanttChartNode) node).getLineCount();
	}

	public int getLineLocation(Object node, ITreeTableModel model,
			int lineIndex, int rowHeight) {
		assertClass("getLineLocation", "node", IGanttChartNode.class, node);
		return ((IGanttChartNode) node).getLineLocation(lineIndex, rowHeight);
	}

	public int getLineHeight(Object node, ITreeTableModel model, int lineIndex,
			int rowHeight) {
		assertClass("getLineHeight", "node", IGanttChartNode.class, node);
		return ((IGanttChartNode) node).getLineHeight(lineIndex, rowHeight);
	}

	public int getLineIndex(Object node, ITreeTableModel model,
			Object timelineObject) {
		assertClass("getLineIndex", "timelineObject", ITimelineObject.class,
				timelineObject);
		return ((ITimelineObject) timelineObject).getLineIndex();
	}

	public boolean isLineVisible(Object node, ITreeTableModel model,
			int lineIndex) {
		return true;
	}
}
```

[1] This policy is responsible for managing the „inner lines“ shown within a row. This feature can be used to display over-lapping timeline objects.

Commands

Commands are used by the framework to modify models. They are looked up from policies when the user performs an action. Most frameworks modify the model directly and then send an event to listeners, which perform the additonal work needed to update the data source. However, using commands as an indirection to modify the data has the advantage that undo / redo functionality and progress monitoring can be more easily added to an application.

Commands need to implement the *ICommand* interface. This interface defines the methods for executing, undoing, or redoing a command:

```
public interface ICommand extends Serializable {

	/**
	 * Executes the command.
	 * 
	 * @param monitor
	 *            a progress monitor
	 * @since 1.0
	 */
	void executeCommand(IProgressMonitor monitor);

	/**
	 * Undos the command.
	 * 
	 * @param monitor
	 *            a progress monitor
	 * @since 1.0
	 */
	void undoCommand(IProgressMonitor monitor);

	/**
	 * Redos the command.
	 * 
	 * @param monitor
	 *            a progress monitor
	 * @since 1.0
	 */
	void redoCommand(IProgressMonitor monitor);

	/**
	 * Determines whether the command is relevant for undo / redo operations.
	 * Commands that are not conisdered to be relevant will not be added to the
	 * list of commands that can be undone but they also do not invalidate that
	 * list and the command stack remains undoable if it was undoable before.
	 * 
	 * @return TRUE if the command is relevant
	 * @since 1.0
	 */
	boolean isRelevant();

	/**
	 * Returns true if the command can be undone.
	 * 
	 * @return true if the command is undoable
	 * @since 1.0
	 */
	boolean isUndoable();

	/**
	 * Returns true if the command can be redone.
	 * 
	 * @return true if the command is redoable
	 * @since 1.0
	 */
	boolean isRedoable();

	/**
	 * The name of the command.
	 * 
	 * @return the command's name
	 * @since 1.0
	 */
	String getName();
}
```

The interface also contains methods needed for operations performed by the command stack. These methods determine whether a command is undoable or redoable and also whether it is a „relevant“ method. Only relevant methods are added to the command stack and can be undone or redone.

Command Stack

Each Gantt chart manages its own command stack. It is called „stack“ because it internally uses a list to manage a stack of executed commands. It allows the user to pop and push commands via execute, undo, and redo operations. The availability of these operations can of course be limited based on the undo and redo support of the individual commands. If a command is not undoable then the user will not be able to step back to any command before it. The command stack can be retrieved from the Gantt chart by calling the following method on *AbstractGanttChart*:

```
public ICommandStack AbstractGanttChart.getCommandStack();
```

However, in most situations there is no need to have direct access to the command stack. When an application wants to execute a command it simply calls the command execution methods on *AbstractGanttChart*:

```
public void AbstractGanttChart.commandExecute(ICommand cmd);
public void AbstractGanttChart.commandUndo();
public void AbstractGanttChart.commandRedo();
```

These methods create progress monitors and then delegate the call to the Gantt chart‘s command stack, which will perform the actual command execution in a specialized command execution thread. Any object can be a command stack as long as it implements the *ICommandStack* interface.

```
public interface ICommandStack {
        void execute(ICommand cmd, IProgressMonitor monitor);
        void undo(IProgressMonitor monitor);
        void redo(IProgressMonitor monitor);
        void clear();
        boolean isUndoable();
        boolean isRedoable();
        ICommand getUndoableCommand();
        ICommand getRedoableCommand();
        void addCommandStackListener(ICommandStackListener l);
        void removeCommandStackListener(ICommandStackListener l);
}
```

Application developers can choose to use the same command stack for all Gantt charts by creating a single instance and then setting it on each Gantt chart by calling the following method.

```
public void AbstractGanttChart.setCommandStack(ICommandStack);
```

Another option is to reuse an already existing command stack when the Gantt charts gets added to an already existing project. Then the „old“ command stack needs to get extended with the „new“ *ICommandStack* interface. Last but not least the command stack of the Gantt chart can also be used as the command stack for the entire application. All of these options depend on the type and architecture of the application for which the Gantt chart will be used.

FlexGantt uses the *DefaultCommandStack* class for its Gantt charts. This command stack has a „size“ attribute, which defines the maximum number of commands that remain on the stack. Commands will be removed from the bottom of the stack if this number gets reached. This behaviour is necessary to avoid running out of memory. The size of the command stack can be changed by the application developer. The fact that commands will not be garbage collected also means that commands need to be implemented in a lightweight manner. The less memory a command requires to keep its state[2] the better.

Command Stack Listener

The command stack executes commands in a separate thread of type *CommandExecutionThread*. This class does not add any functionality to the base class *java.lang.Thread*. It currently only exists to make it easier to identify the thread (e.g. when debugging). The *run()* method of the thread performs the actual command execution and fires events of type *CommandStackEvent*, which can be received by listeners attached to the command stack. These listeners need to implement the *ICommandStackListener* interface. The event object contains an ID field that can be used to distinguish between different types of events. Possible events are:

- COMMAND_CANCELED - the user must have clicked on the „cancel“ button in the progress dialog
- COMMAND_EXECUTED - the command was successfully executed
- COMMAND_FAILED - the command execution failed
- COMMAND_STARTED - the command has started its execution
- COMMAND_UNDONE - the command was undone

[2] „State“ in this context means the information needed to undo the command

One command stack listener that is already included in ***FlexGantt*** is the class *GanttChartGlassPane*. It uses the events that it receives in order to block user input during the time when a command gets executed.

Progress Monitor

Before delegating commands to the command stack, the Gantt chart creates an instance of *IProgressMonitor* and passes it along with the command to the stack. This monitor allows long running tasks to give feedback to the user and report how much work has been completed. ***FlexGantt*** uses the same progress monitor interface used by Eclipse as it is completely independent of the UI technology used (Swing, SWT). The interface looks like this:

```
IProgressMonitor.beginTask(String, int)
IProgressMonitor.done()
IProgressMonitor.internalWorked(double)
IProgressMonitor.isCanceled()
IProgressMonitor.setCanceled(boolean)
IProgressMonitor.setTaskName(String)
IProgressMonitor.subTask(String)
IProgressMonitor.worked(int)
```

An in-depth explanation on how to use this interface is given in the appendix „How to Correctly and Uniformly Use Progress Monitors“. This article was copied from the Eclipse website.

Compound Commands

In ***FlexGantt*** users can create, delete, or modify multiple objects at the same time. The row height, for example, can be modified for a single row with a simple drag operation but also for several rows at once if the user holds down the SHIFT key while dragging. It is the *DefaultRowResizeCommand* that is performing the actual change of the height for a single row. Multiple instances of this command would be needed in order to resize several rows at the same time but this approach would make the command stack less useful. The stack would be overloaded with commands and undo / redo operations would be tiresome. This is where compound commands come in. They allow several commands to be executed together. This way only a single command gets added to the command stack and can be undone or redone with ease. The following table lists the compound commands defined by ***FlexGantt***.

Compound Command	Nested Command
DefaultCreateMultipleTimelineObjectsCommand	**DefaultCreateTimelineObjectCommand**
DefaultChangeMultipleTimelineObjectsTimeSpanCommand	**DefaultChangeTimelineObjectTimeSpanCommand**
DefaultDeleteMultipleEventlineObjectsCommand	**DefaultDeleteEventlineObjectCommand**
DefaultDeleteMultipleNodesCommand	**DefaultDeleteNodeCommand**
DefaultDeleteMultipleTimelineObjectsCommand	**DefaultDeleteTimelineObjectCommand**

Command Interceptors

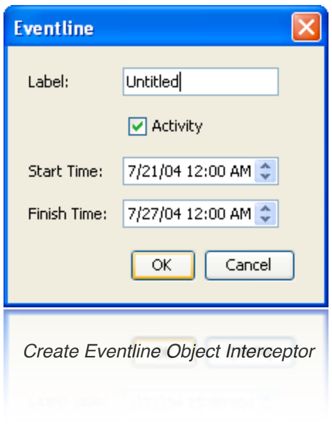
Many applications require that the user confirms and refines changes that are made to models. A typical scenario is a dialog prompting the user with a message like „Are you sure you want to assign this task to this resource?“. Behaviour like this can be implemented in ***FlexGantt*** by using so-called command interceptors. Interceptors need to implement the *ICommandInterceptor* interface. This interface has only one method, which gets invoked before a command gets executed.

```
boolean ICommandInterceptor.intercept(T gc, ICommand cmd);
```

Interceptors are the „glue“ between commands and the view. They are needed because policies and commands do not have any backpointer to the view, which is why they can not prompt the user themselves. A key architectural decision in ***FlexGantt*** in the decoupling of policies, models, and commands from the view. This allows them to be reused for other UI technologies. Currently only the Swing framework is supported by ***FlexGantt*** but other UI frameworks might be supported in the future. If, for example, an SWT view gets added then the policies and commands could be reused. Interceptors need to be registered with the Gantt chart for which they are used. This is done by mapping the command type to the interceptor instance by invoking the following method on *AbstractGanttChart*:

```
AbstractGanttChart.setCommandInterceptor(Class<? extends ICommand> cmdClass,
                                                             ICommandInterceptor inter);
```

Only one interceptor is registered by default. It is the one used for *DefaultCreateEventlineObjectCommand.class*. This interceptor shows the following dialog when the user wants to create a new object in the eventline.



Create Eventline Object Interceptor

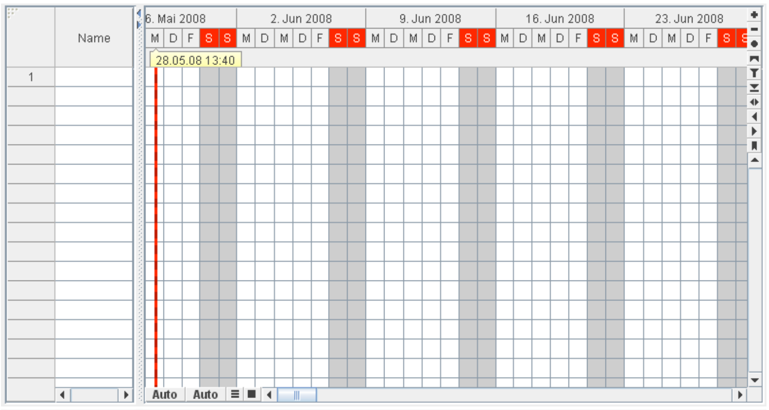
Threading Issues

We already know that commands in ***FlexGantt*** are executed in their own thread (*CommandExecutionThread*). We also know that commands perform changes on the models (e.g. *IGanttChartModel*). These changes cause events to be fired from the models to their listeners (e.g. *IGanttChartModelListener*). Some of the listeners are Swing components, which now have to update themselves in various ways. Simple repaints cause no problems, because the *repaint()* method of these components is thread-safe. However, some of the ***FlexGantt*** components perform more elaborate updates of their internal state, which can cause problems like concurrent modifications of collections.

To avoid this problem the ***FlexGantt*** components call the static *invokeLater(Runnable)* or the *invokeAndWait(Runnable)* methods of the *SwingUtilities* class. These methods ensure that the Runnable passed to them will be executed on the Event Dispatch Thread (EDT). Application developers need to be aware of this problematic whenever they add listeners to models. Their listeners, too, will be called within the command thread. This fact might be relevant, depending on the purpose of the listeners.

Policies of AbstractGanttChart

The superclass of all ***FlexGantt*** Gantt charts is basically an empty container and it is the responsibility of the subclasses to add components to it. The simple *GanttChart* class adds a single tree table and a single layer container to it. The more advanced *DualGanttChart* adds two of each. Even though the abstract superclass does not define any children components, it still provides policies: The *IOverviewPolicy* and the *IStatusBarPolicy*.

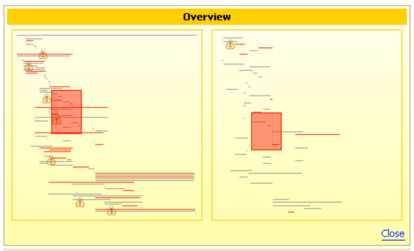


Gantt Chart

IOverviewPolicy

```
Implementation: DefaultOverviewPolicy
      Commands: N/A
       Used By: OverviewPalette
```

The overview policy is used by the overview selector, which provides „radar" functionality. In the selector all timeline objects are visible all the time time. The colors used for these timeline objects and the icons shown on top of them can be controlled via the policy. The following snapshot shows an overview selector, which uses two different colors for the timeline objects (red and black). The meaning of the red color could be that the task represented by the timeline object is delayed. The yellow warning icons might indicate that a task does not have a resource assigned to it, yet.



Overview Selector

Policy Method	Purpose

`Object getOverviewStatus( Object node, Object timelineObject, IGanttChartModel model);`	Returns an object, which represents the state of a timeline object. This object is used as a key to lookup the color and / or icon from the LayerContainer.
	
	`public void setTimelineObjectStatusColor( Object status, Color color)`
	`public Color getTimelineObjectStatusColor( Object status)`
	
	`public void setTimelineObjectStatusIcon( Object status, Icon icon)`
	`public Icon getTimelineObjectStatusIcon( Object status)`

The default implementation of this policy is called *DefaultOverviewPolicy, which* assumes that the timeline object passed to it implements the ITimelineObject interface. It then performs a cast and delegates to the interface.

```
public Object getOverviewStatus(Object node, Object timelineObject,
               IGanttChartModel model) {
       assertClass("getOverviewStatus", "timelineObject", ITimelineObject.class, timelineObject);
       return ((ITimelineObject) timelineObject).getStatus();
}
```

Please note how the *assertClass()* method gets used to verify that the timeline object really does implement the ITimelineObject interface. If it doesn‘t then the assert method will print out a nicely formatted warning message to the standard system out stream.

IStatusBarPolicy

```
Implementation: AbstractStatusBarPolicy
                TimeGranularityStatusBarPolicy
                SimpleGranularityStatusBarPolicy
      Commands: N/A
       Used By: GanttChartStatusBar
```

The statusbar policy is used to control which information will be shown in the status bar and how this information will be formatted. Several fields are shown in the statusbar. Some of these fields display time points or time spans. Depending on the currently used dateline model these time points and time spans need to be rendered very differently. The model class *SimpleGranularityDatelineModel* requires the statusbar to show time points as numbers (1, 2, 3, ...) and time spans as pairs of numbers (1 - 10, 10 - 20, ...). The *TimeGranularityDatelineModel* needs the statusbar to display standard time information such as "Mon. Jan 5th". Accordingly there are two different statusbar policies: *SimpleGranularityStatusBarPolicy* and *TimeGranularityStatusBarPolicy*. By default the ***FlexGantt*** Gantt chart classes *GanttChart* and *DualGanttChart* use the *TimeGranularityDatelineModel* together with *TimeGranularityStatusBarPolicy*. *SimpleGranularityDualGanttChart* and *SimpleGranularityGanttChart* use *SimpleGranularityDatelineModel* together with the *SimpleGranularityStatusBartPolicy*.

The following table lists the methods of *IStatusBarPolicy* and describes their purpose:

Policy Method	Purpose
`String getTimeNowString( IDatelineModel, long)`	Returns a formated text representing the given time point.
`String getTimeSpanString( IDatelineModel, ITimeSpan)`	Returns a formated text representing the given time span.
`String getTimeString( IDatelineModel, long)`	Returns a formated text representing the given time point.
`boolean isStatusBarFieldVisible( StatusBarField)`	Determines whether the given status bar field will be shown in the status bar or not. The constructors of *IStatusBarPolicy* implementations usually accept a collection of fields and then check in this method whether the field is included in the collection or not.

The Enumerator *StatusBarField* contains several values, used to distinguish between the different status bar fields:

StatusbarField	Purpose
`CROSSHAIR`	A field used as a visual indicator for the crosshair feature (is the crosshair turned on or off).
`GRID`	A field used as a visual indicator for the current mode of the grid (major grid, minor grid, off).
`KEY_STROKES`	This field brings up a selector, which lists all currently registered actions and the key strokes used to trigger them.
`MEMORY`	A field, which displays the available and the currently used memory. A hidden feature is the garbage collection that can be triggered by clicking on the field.

StatusbarField	Purpose
`MESSAGE`	A field used as a visual indicator for the presence of status messages (error, warning, information messages). The field displays different icons based on the severity level of the messages. A click on the field brings up a new window, which lists all messages currently attached to the Gantt chart.
`POPUP`	A field used as a visual indicator for the popup feature (are popups turned on or off)
`PROGRESS_BAR`	A field used to display a progress bar. The status bar implements the IProgressMonitor interface and can be used as a progress monitor when executing commands.
`TIME`	A field, which displays the time at the current location of the mouse cursor.
`TIME_NOW`	A field that is used to display the „time now“ (in most cases the system time / the computer‘s time).
`TIME_NOW_LOCK`	A field that can be used to lock the horizontal scrolling behaviour to the „time now“. The visible area of the Gantt chart automatically follows the vertical line, which (normally) displays the current system time.
`TIME_ZONE`	A field that shows the current time zone used by the dateline. By clicking on the field the user can bring up a dialog to change the timezone.

The class *AbstractStatusBarPolicy* adds all fields to itself, except for the TIME_ZONE field. This is due to the fact that most planning and scheduling applications don‘t need it.

It should be noted that the *GanttChartStatusBar* class is just one possible implementation of a statusbar for a ***FlexGantt*** Gantt chart. The superclass of *GanttChartStatusBar* is called *StatusBar*. It is completely independent of any Gantt chart classes. Developers are free to implement their own statusbar by subclassing this class (or any other class for that matter). For more information on this topic, please consult the document „FlexGantt - Components“.

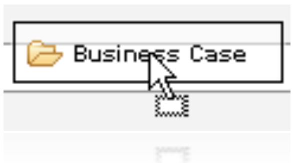
Policies of TreeTable

Most of the policies used for the tree table deal with its editing behaviour. This makes sense, since the table‘s primary purpose is data entry and display. The data display gets controlled by the tree table cell renderers, so there is no need for policies in that area.

INodeDragAndDropPolicy

```
Implementation: DefaultNodeDragAndDropPolicy
      Commands: DefaultNodeDragAndDropCommand
       Used By: TreeTableDragAndDropManager
```

The *INodeDragAndDropPolicy* gets used to control the drag & drop (DnD) operations within the tree table. The methods within this policy determine whether a node is draggable and where it can be dropped. Once a drag and drop operation is complete the policy also returns the command that shall be used to perform the necessary model modifications.



TreeTable Drag & Drop

FlexGantt uses the standard drag and drop concepts that are built into AWT. It is important to understand these concepts in order to completely understand how the *INodeDragAndDropPolicy* works. Please visit the following site to learn more about drag and drop: http://java.sun.com/docs/books/tutorial/dnd/

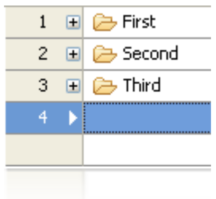
Policy Method	Purpose
`int getDragActions(Object node, ITreeTableModel model);`	Returns the drag actions that are allowed for the given timeline object object. The actions are defined as integer constants in the *java.awt.dnd.DnDConstants* class: • ACTION_COPY • ACTION_COPY_OR_MOVE • ACTION_LINK • ACTION_MOVE • ACTION_NONE • ACTION_REFERENCE ***FlexGantt*** only supports „none“ and the copying and / or moving of timeline objects. The drag and drop behaviour is platform dependent, which means that different operating systems use different modifier keys to distinguish between a copy and a move (on Windows use the CTRL key to perform a copy).
`int getDropActions(Object draggedNode, Object draggedNodeParent, ITreeTableModel draggedNodeModel, Object newParentNode, ITreeTableModel newModel);`	Returns the possible drop actions for the given dragged timeline object on the given target node. The user will only be able to drop the timeline object if the drag action matches one of the drop actions.
`ICommand getDragAndDropCommand( Object droppedNode, Object oldParent, ITreeTableModel oldModel, int oldChildIndex, Object newParent, ITreeTableModel newModel, int newChildIndex, int dropAction);`	Returns the command object that will perform the actual changes required in order to detach the node from its current parent and attach it to its new parent.

INodeEditPolicy

```
Implementation: DefaultNodeEditPolicy
      Commands: DefaultChangeKeyCommand
                DefaultChangeValueCommand
                DefaultCreateNodeCommand
                DefaultDeleteMultipleNodesCommand
                DefaultDeleteNodeCommand
                DefaultInsertNodeCommand
       Used By: TreeTable
```

The tree table is an editable component, which means that the user can change the values in the tree cells, create new nodes / rows, or delete nodes / rows. This policy is used to control which editing features are available and how they are being executed (commands).

,

Policy Method	Purpose
`boolean isKeyEditable( Object node, ITreeTableModel model);`	Returns true if the key value of a node in the tree table can be edited by the user. This is the value shown in the key column. The key column is the column, which displays the hierarchical tree structure of the Gantt chart model.
`boolean isValueEditable( Object node, ITreeTableModel model, int modelIndex);`	Returns true if the value in the column with the given model index can be edited by the user.
`boolean isSelectable( Object node, ITreeTableModel model);`	Returns true if the given node is selectable by the user.
`boolean isDeletable( Object node, ITreeTableModel model);`	Returns true if the given node is deletable by the user.
`boolean isCreateEnabled( ITreeTableModel model);`	Returns true if the table allows the user to interactively create new rows / nodes. If this is the case then a little arrow becomes visible to the left of the row below the last used row. This unused row can be used to create new nodes.  *New Row Marker*
`ICommand getDeleteNodeCommand( Object node, ITreeTableModel model);`	Returns the command that will be used to delete the given node.
`ICommand getDeleteNodesCommand( List<Object> node, ITreeTableModel model);`	Returns the command that will be used to delete the given nodes (several at the same time).
`ICommand getCreateNodeCommand( Object parentNode, ITreeTableModel model, Object key, Object[] values);`	Returns the command that will be used to create a new node.
`ICommand getInsertNodeCommand( Object parentNode, int childIndex, ITreeTableModel model);`	Returns the command that will be used to insert a new row between two already existing rows.

Policy Method	Purpose
`ICommand getChangeKeyCommand( Object node, ITreeTableModel model, Object key);`	Returns the command that will be used to change the „key“ value of a tree table node.
`ICommand getChangeValueCommand( Object node, ITreeTableModel model, Object value, int index);`	Returns the command that will be used to change the column value for the given model index of the given node.

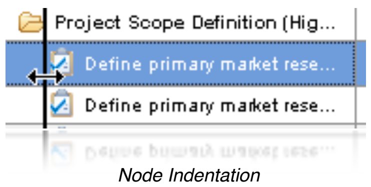
The default implementation of this policy delegates most of its work to the *ITreeTableNode* interface.

```
public boolean isKeyEditable(Object node, ITreeTableModel model) {
       return ((ITreeTableNode) node).isKeyEditable();
}

public boolean isValueEditable(Object node, ITreeTableModel model,
               int modelIndex) {
       return ((ITreeTableNode) node).isValueEditable(modelIndex);
}

public boolean isDeletable(Object node, ITreeTableModel model) {
       return ((ITreeTableNode) node).isDeletable();
}

public boolean isSelectable(Object node, ITreeTableModel model) {
       return ((ITreeTableNode) node).isSelectable();
}
```

INodeIndentationPolicy

```
Implementation: DefaultNodeIndentationPolicy
      Commands: DefaultIndentNodesCommand
                DefaultOutdentNodesCommand
       Used By: TreeTable
```

The tree table allows the user to interactively indent nodes. This means, that a node gets reassigned to a new parent node and the hierarchy shown in the table changes. One use case for this behaviour is the creation of the structure of an organization (company, departments, people). The following snapshot shows the visual feedback shown to the user. The mouse cursor changes its shape and a black vertical line appears.


Project Scope Definition (Hig...
Define primary market rese...
Define primary market rese...
Node Indentation

FlexGantt uses the term „outdent“ for the inverse of the indentation operation. It should be noted that this word does not exist in the English language. It was used for symmetrie.

Policy Method	Purpose
`boolean isIndentable( Object node, ITreeTableModel model);`	Returns true if the user is allowed to indent the given tree node. Indentation means that the node gets reassigned as a child to the node that is currently located above it.
`boolean isOutdentable( Object node, ITreeTableModel model);`	Returns true if the user is allowed to outdent the given tree node. Outdentation is the opposite of indentation. A node gets reassigned as a child to the parent of its current parent.

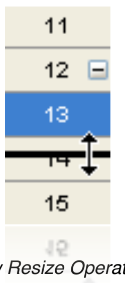
Policy Method	Purpose
`ICommand getIndentationCommand( Object[] nodes, Object oldParent, int[] oldChildIndices, Object newParent, int[] newChildIndices, ITreeTableModel model);`	Returns the command used for indenting a node.
`ICommand getOutdentationCommand( Object[] nodes, Object oldParent, int[] oldChildIndices, Object newParent, int[] newChildIndices, ITreeTableModel model);`	Returns the command used for outdenting a node.

The default implementation of this policy uses the two commands *DefaultIndentNodesCommand* and *DefaultOutdent-NodesCommand.* Both of these commands subclass the *DefaultReassignNodesCommand*. Custom indentation commands are free to subclass this class.

IRowPolicy

```
Implementation: DefaultRowPolicy
      Commands: DefaultRowResizeCommand
       Used By: AbstractRowHeader
                TreeTable
                TreeTableNode
                GridLayer
                OverviewPalette
```

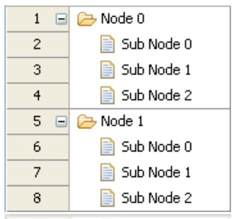
The behaviour and apperance of rows inside the Gantt chart can be controlled via this policy. The policy defines the current height of a row, its minimum height, and its maximum height. The policy also determines whether a row is resizable or not, whether its horizontal line gets drawn, and the content of the tooltip that will be shown. The following snapshot shows the visual feedback shown to the user during a resize operation: the mouse cursor will change its shape and a short horizontal line becomes visible.



Row Resize Operation

Policy Method	Purpose
`int getRowHeight( Object node, ITreeTableModel model);`	Returns the current height of the row for the given node.
`int getRowHeightMinimum( Object node, ITreeTableModel model);`	Returns the minimum height that a row can have. This is relevant when the user changes the row height via the mouse.
`int getRowHeightMaximum( Object node, ITreeTableModel model);`	Returns the maximum height that a row can have. This is relevant when the user changes the row height via the mouse.
`boolean isRowResizable( Object node, ITreeTableModel model);`	Returns true if the row where the given node is located is resizable.

Policy Method	Purpose

`boolean isRowLineVisible(`
`        Object node,`
`        boolean expanded,`
`        ITreeTableModel model);`
	Returns true if a horizontal line shall be drawn at the bottom of the row that displays the given node. Not showing this line allows an applicati-on to group rows together.
	
	 1 Node 0
	2 Sub Node 0
	3 Sub Node 1
	4 Sub Node 2
	5 Node 1
	6 Sub Node 0
	7 Sub Node 1
	8 Sub Node 2
	Grouping of Rows

`String getRowToolTip(`
`        Object node,`
`        ITreeTableModel model,`
`        TreeTableColumn column);`
	Returns a text that will be displayed as the tooltip when the mouse cursor hovers over the given node. The additional „column“ parameter can be used to return tooltips for individual cells.

`ICommand getRowResizeCommand(`
`        Object node,`
`        ITreeTableModel model,`
`        int rowHeight);`
	Returns the command that will be used for resizing the row where the given node is located.

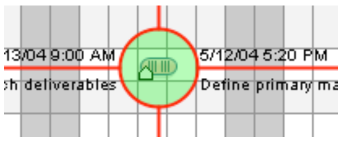
Policies of LayerContainer

The layer container is the component with the largest number of policies. This is due to the fact that it is a very „rich“ component in respect to visualization and editing capabilities. The more features a component has, the more customization potential is available.

ICrosshairPolicy

```
Implementation: DefaultCrosshairPolicy
      Commands: N/A
       Used By: CrosshairLayer
```

The crosshair feature is a very good instrument for training classes and product demonstrations. It allows the presenter to highlight a timeline object. The crosshair can display up to four labels in the four corners around the center. The content of these labels can be configured by the crosshair policy.



Crosshair Feature

Policy Method	Purpose
`boolean isLabelVisible( LabelPosition position);`	Determines if the label at the given position is visible or not. The enumerator LabelPosition is defined inside the interface ICrosshairPolicy. Possible values are UPPER_LEFT, UPPER_RIGHT, LOWER_LEFT, and LOWER_RIGHT.
`String getLabel( Object node, TimelineObjectPath path, IGanttChartModel model, long time, LabelPosition position);`	Returns the label for the given position (see above). The method receives the node (equals the row), the path to the timeline object, the model, and the time at the current mouse position as input.

The default implementation can be configured with different formatters for the dates shown. One formatter is used for the time at the current mouse location, the other for the start and end times of the timeline object.

```
public void setTimelineDateFormat(DateFormat format) {
public void setTimelineObjectDateFormat(DateFormat format)
```

The default implementation displays the following information in the four positions:

Position	Text
UPPER_LEFT	The time span of the timeline object on which the crosshair cursor is hovering (if there is one, blank otherwise).
UPPER_RIGHT	The time at the current crosshair cursor position.
LOWER_LEFT	The name of the timeline object on which the crosshair cursor is hovering (if there is one, blank otherwise).
LOWER_RIGHT	The name of the hierarchy node that belongs to the row over which the crosshair cursor is hovering.

IDragAndDropPolicy

```
Implementation: DefaultDragAndDropPolicy
      Commands: DefaultDragAndDropCommand
                DefaultMultiDragAndDropCommand
       Used By: DragLayer
                LassoLayer
                DefaultDragRowRenderer
                DefaultEditModeController
```

The drag and drop policy gets invoked whenever a drag gesture gets recognized by the layer container. This is the case when the mouse cursor hovers over a timeline object and the user presses and drags the mouse. At the beginning of the drag operation the *DragLayer* calls upon the policy to check which drag actions (copy and / or move) are supported by the timeline object.

While the drag operation is going on it checks which actions are allowed at the current drop location. A drop is only possible if the **allowed** drop actions contain the **user-requested** drop action. If the drop is valid the drag layer looks up the drag and drop command from the policy, which will then perform the actual model modification. In a first step the default commands remove the dragged timeline object from its current parent and attaches it to the new parent. If the timeline object happens to be the source or the target of one or more relationships then the command will fix these in a second step.

The drag and drop actions are defined in the class *java.awt.dnd.DnDConstants*.

```
    /**
     * An int representing no action.
     */
    public static final int ACTION_NONE             = 0x0;

    /**
     * An int representing a copy action.
     */
    public static final int ACTION_COPY             = 0x1;

    /**
     * An int representing a move action.
     */
    public static final int ACTION_MOVE             = 0x2;

    /**
     * An int representing a copy or move actions.
     */
    public static final int ACTION_COPY_OR_MOVE     = ACTION_COPY | ACTION_MOVE;

    /**
     * An int representing a link action.
     */
    public static final int ACTION_LINK              = 0x40000000;

    /**
     * An int representing a reference action (synonym for ACTION_LINK).
     */
    public static final int ACTION_REFERENCE    = ACTION_LINK;
```

Please note that ***FlexGantt*** does not use the „verbs“ LINK and REFERENCE. The user can only copy or move timeline objects. Do not mistake relationships as links. Creating relationships is done by the *LassoLayer* via standard mouse listeners and not via the drag and drop API.

Policy Method	Purpose
`int getDragActions( TimelineObjectPath path, IGanttChartModel model);`	Returns the drag actions that are **supported** by the timeline object to which the path points. The timeline object is a member of the given model. This method basically asks the questions „What is it that I can do with you? Can I move you somewhere else? Can I copy you?“.
`int getDropActions( TimelineObjectPath path, IGanttChartModel model, Object newNode, IGanttChartModel newModel, long newStartTime);`	Returns the drop actions that are **allowed** for the given arguments. This method answers the question „What exactly is the user allowed to do with this timeline object on this row (node)? Can the object be moved here? Can it be copied?“.

Policy Method	Purpose

**`ICommand getDragAndDropCommand(`
`        TimelineObjectPath path,`
`        IGanttChartModel model,`
`        Object newNode,`
`        IGanttChartModel newModel,`
`        long newStartTime,`
`        Object[] timelineObjects,`
`        ILayer layer,`
`        int dropAction);`**
	Returns the command that will be used to execute the user-requested drop action. The command can use the *dropAction* value to determine whether the user wants to copy or move the timeline object. The command returned by this method has to ensure that those relationships where the timeline object is a source or a target will be fixed. Relationships contain paths to timeline objects. These paths will be most likely different after the drag and drop. They remain intact if the timeline object gets dropped on the same row.

The default implementation *DefaultDragAndDropPolicy* delegates to the following two methods that are defined on the *ITimelineObject* and *IGanttChartNode* interfaces:

```
int ITimelineObject.getDragActions();
int IGanttChartNode.getDropActions(Object timelineObject, long timeAtDropLocation);
```

Using the drag and drop policy seems complicated but for the most cases it is completely sufficient to focus on the values returned by these interface methods. The *getDragActions()* method needs to return an integer, which represents what the user can do. The value of this integer is either NONE, MOVE, COPY, or COPY_OR_MOVE. The node method *getDropActions()* has to consider the arguments passed to it and base its return value on them: „Do I accept this type of timeline object? If I do, can the user only move the timeline object to me (the node) or can he also create a copy? Do I care about the time point at the drop location? Should I accept a timeline object that was dropped in the past?“ And so on and so on.

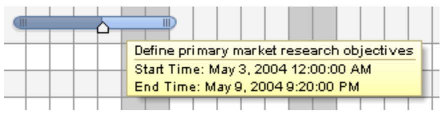
The class *DefaultGanttChartNode* accepts all timeline objects that are being „moved“ to it. It rejects all timeline objects when the user tries to perform a „copy“, because the *DefaultDragAndDropCommand* doesn‘t know how to execute a copy. However, the timeline object class *DefaultTimelineObject* allows copies and moves. Application developers need to implement their own commands if they want to copy timeline objects. *AbstractDragAndDropCommand* can be used as the superclass for these custom commands. It provides the *fixRelationships()* method for repairing the relationships of the dragged timeline objects.

```
protected void fixRelationships(ITimelineObject object,
               IGanttChartNode oldOwner, IGanttChartNode newOwner,
               DefaultGanttChartModel oldModel, DefaultGanttChartModel newModel) {
       Iterator<IRelationship> iter = oldModel.getRelationships();
       while (iter.hasNext()) {
               IRelationship rel = iter.next();
               TimelineObjectPath newPath = newModel.getTimelineObjectPath(
                              newOwner, object, layer);
               if (rel.getSourcePath().getTimelineObject().equals(object)) {
                       // adjust the source path
                       rel.setSourcePath(newPath);
               } else if (rel.getTargetPath().getTimelineObject().equals(object)) {
                       // adjust the target path
                       rel.setTargetPath(newPath);
               }
       }
}
```

IDragInfoPolicy

```
Implementations: DefaultDragInfoPolicy
                 TimeGranularityDragInfoPolicy
       Commands: N/A
        Used By: DragLayer
                 DefaultDragInfoRenderer
```

It is very important that the user receives feedback during drag and drop operations, so that he knows what the result of the operation will be. This information is provided by the *IDragInfoPolicy*. It returns an object, which represents / contains the drag info data. The *DragLayer* and *IDragInfoRenderer* instances are then responsible for visualizing the data.


Define primary market research objectives
Start Time: May 3, 2004 12:00:00 AM
End Time: May 9, 2004 9:20:00 PM

Drag Info

Policy Method	Purpose
`Object getDragInfo( TimelineObjectPath path, IGanttChartModel model, Object dropNode, ITimeSpan dropSpan);`	Returns the drag information needed when the user drags a timeline object. Drags also occure when the user simply changes the start or the end time of the timeline object. Drag and drop does not mean that the timeline object gets placed on a different row.
`Object getDragInfo( TimelineObjectPath path, IGanttChartModel model, long timePoint, double percentageComplete);`	Returns the drag information needed when the user changes the percentage complete value of an activity object. Activity objects extend timeline objects with an additional field that stores the „percentage complete“ value.
`Object getDragInfo( TimelineObjectPath path, IGanttChartModel model, int rowHeight, int y, double capacityUsed);`	Returns the drag information needed when the user changes the capacity value of a capacity object. Capacity objects extend timeline objects with an additional field that stores the „capacity used“ value.

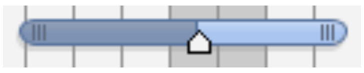
The type of the drag info objects can be used to register different renderers on the *DragLayer*.

```
public void DragLayer.setDragInfoRenderer(Class cl, IDragInfoRenderer renderer);
```

IEditActivityObjectPolicy

```
Implementation: DefaultEditActivityObjectPolicy
      Commands: DefaultChangePercentageCommand
       Used By: DragLayer
                ActivityObjectEditModeController
```

The *IActivityObject* interface is an extension of the *ITimelineObject* interface. It adds a single new field to timeline objects called „percentage complete“. This value is often used in project planning applications to express how much of the necessary work for a task has been completed. In ***FlexGantt*** the user can interactively modify this value. When the mouse hovers on top of such an activity object the user will be presented with a little slider at the bottom of the activity. The default settings of *ActivityObjectEditModeController* now allow the user to move the slider by holding down the SHIFT key and dragging the mouse to the left or right. This of course will only be possible if the *IEditActivityObjectPolicy* allows it and if also provides a command, which will perform the necessary model modification.



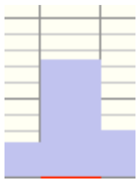
Activitiy Object with Percentage Slider

Policy Method	Purpose
`boolean isPercentageChangeable( TimelineObjectPath path, IGanttChartModel model);`	Returns true if the user is allowed to change the „percentage complete“ value.
`ICommand getChangePercentageCommand( TimelineObjectPath path, IGanttChartModel model, double percentage);`	Returns a command that will modify the model so that the given value will be used as the new „percentage complete“ value for the activity object specified by the given timeline object path.

IEditCapacityObjectPolicy

```
Implementation: DefaultEditCapacityObjectPolicy
      Commands: DefaultChangeCapacityCommand
       Used By: DragLayer
                CapacityObjectEditModeController
```

The *ICapacityObject* interface is an extension of the *ITimelineObject* interface. It adds a single new field to timeline objects called „capacity used“. This value can be used to implement capacity profiles for limited-capacity resources. In ***FlexGantt*** the user can interactively modify this value. When the mouse hovers over the top edge of such a capacity object the mouse cursor will change. The default settings of *CapacityObjectEditModeController* now allow the user to change the height of the object by holding down the SHIFT key and dragging the mouse up or down. This of course will only be possible if the *IEditCapacityObjectPolicy* allows it and if also provides a command, which will perform the necessary model modification.



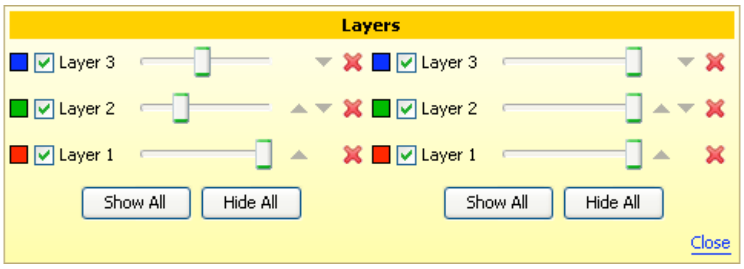
Capacity Object

Policy Method	Purpose
`boolean isCapacityChangeable( TimelineObjectPath path, IGanttChartModel model);`	Returns true if the user is allowed to change the „capacity used“ value.
`ICommand getChangeCapacityCommand( TimelineObjectPath path, IGanttChartModel model, double newCapacityUsed);`	Returns a command that will modify the model so that the given value will be used as the new „capacity used“ value for the capacity object specified by the given timeline object path.

IEditLayerPolicy

```
Implementation: DefaultEditLayerPolicy
      Commands: DefaultAddLayerCommand
                DefaultRemoveLayerCommand
       Used By: LayerPalette
```

Layers can be added and removed from a Gantt chart model. The *IEditLayerPolicy* returns the commands that are needed for the model changes. The following image shows the layer selector with red „delete“ icons.



Layer Selector

Policy Method	Purpose
`ICommand getAddLayerCommand( ILayer layer, IGanttChartModel model);`	Returns the command used for adding the given layer to the given model.
`ICommand getRemoveLayerCommand( ILayer layer, IGanttChartModel model);`	Returns the command used for removing the given layer from the given model.

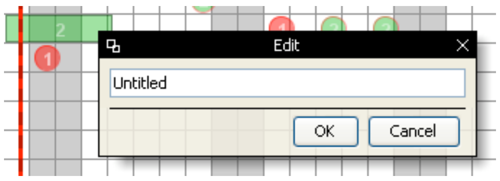
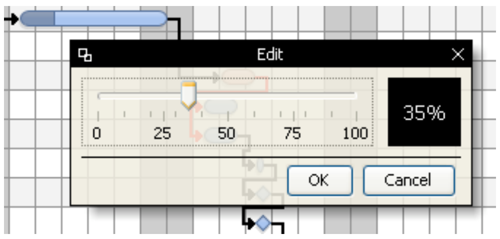
Note: layers can only be removed from the model via the UI if the feature DELETION was added to the layer:

```
boolean ILayer.isFeatureEnabled(ILayer.Feature feature); // interface

public void Layer.addFeature(ILayer.Feature feature);    // implementation
public void Layer.removeFeature(ILayer.Feature feature);
```

IEditTimelineObjectPolicy

```
Implementation: DefaultEditTimelineObjectPolicy
      Commands: DefaultChangeMultipleTimelineObjectsTimeSpanCommand
                DefaultChangeTimelineObjectTimeSpanCommand
                DefaultCreateTimelineObjectCommand
                DefaultDeleteMultipleTimelineObjectsCommand
       Used By: DragLayer
                EditingLayer
                LassoLayer
```

The *IEditTimelineObjectPolicy* defines which editing operations are allowed for timeline objects. Possible operations are: changing the start time, changing the duration, bringing up an in-place editor, creating new timeline objects, deleting existing ones.

Policy Method	Purpose
`boolean isInPlaceEditable( TimelineObjectPath path, IGanttChartModel model);`	Returns true if the user can bring up an „in-place“ editor by double-clicking on a timeline object. These editors are somewhat special as they are not shown in a new window or dialog. Instead they are directly added to the layer container. The editors have have to implement the *ITimelineObjectEditor* interface. In most cases they extend the *AbstractTimelineObjectEditor* class. New editors have to be registered on the layer container: `public void LayerContainer.setTimelineObjectEditor( Class tloCass, ITimelineObjectEditor editor);` ***FlexGantt*** ships with two built-in editors. One for the standard timeline objects (*DefaultTimelineObject*) and one for activity objects (*DefaultActivityObject*).  Untitled *Timeline Object Editor*  *Activity Object Editor*

Policy Method	Purpose
`boolean isStartTimeChangeable( TimelineObjectPath path, IGanttChartModel model);`	Returns true if the user can change the start time of the given timeline object. Changing the start time can mean two different things, depending on whether the user is also allowed to change the duration. • If the duration **can not** be changed, then the permission to change the start time means that the timeline object can be moved left or right. • If the duration **can** be changed, then the permission to change the start time means that the timeline object can start earlier and the overall duration of the timeline object changes.
`boolean isDurationChangeable( TimelineObjectPath path, IGanttChartModel model);`	Returns true if the user is allowed to drag the right edge of a timeline object in order to change the object‘s duration (time span).
`boolean isDeletable( TimelineObjectPath path, IGanttChartModel model);`	Returns true if the given timeline object may be deleted by the user.
`boolean isCreatable( Object node, IGanttChartModel model, ILayer layer, ITimeSpan span);`	Returns true if the user is allowed to create new timeline objects on the given layer for the given time span and node.
`ICommand getChangeTimeSpanCommand( TimelineObjectPath path, IGanttChartModel model, ITimeSpan span, int dropAction);`	Returns the command that will be used to change the time span of the given timeline object. The „dropAction“ argument originates from Java‘s drag and drop API. It is usually used to distinguish between a MOVE and a COPY. This, of course, doesn‘t make sense when changing the duration of a timeline object but it might be useful as a flag for something else, e.g. to distinguish between a „change duration if no constraints are violated“ and a „change duration even if it causes an inconsistent schedule“. The values for the „dropAction“ argument are defined in the class *java.awt.dnd.DnDConstants.*
`ICommand getChangeTimeSpansCommand( IGanttChartModel model, List<TimelineObjectPath> paths, List<ITimeSpan> timeSpans, int dropAction);`	Returns the command that will be used to change the time spans of the given timeline objects. The two lists containing the time spans and the timeline objects are in sync, which means that the first time span shall be applied to the first timeline object and the second time span shall be applied to the second timeline object, and so on. This method gets invoked by the layer container, when the user wants to perform an an “alignment“ of several timeline objects (same start time, same end time). The „dropAction“ argument originates from Java‘s drag and drop API. It is usually used to distinguish between a MOVE and a COPY. This, of course, doesn‘t make sense when changing the duration of a timeline object but it might be useful as a flag for something else, e.g. to distinguish between a „change duration if no constraints are violated“ and a „change duration even if it causes an inconsistent schedule“. The values for the „dropAction“ argument are defined in the class *java.awt.dnd.DnDConstants.*

Policy Method	Purpose
`ICommand getCreateCommand( Object node, IGanttChartModel model, ILayer layer, ITimeSpan span, int lineIndex);`	Returns the command that will be used to create a new timeline object on the given node, model, layer, time span, and line index.
`ICommand getDeleteCommand( IGanttChartModel model, Collection<TimelineObjectPath> paths);`	Returns the command that will be used to delete the given timeline objects.

The default implementation of this policy delegates to the *ITimelineObject* interface for checking the editing features of timeline objects.

IGridLinePolicy

```
Implementations: DefaultGridLinePolicy
                 TimeGranularityGridLinePolicy
       Commands: N/A
        Used By: GridLayer
```

The grid line policy (do not confuse with grid policy) controls the behaviour of the major and minor grid lines. In general the *GridLayer* is capable of displaying either the minor grid lines, or the major grid lines or both at the same time. Usually the major grid lines are located on top of a minor grid lines. Depending on the dateline model there might be situations where this is not the case (e.g. the *TimeGranularityDatelineModel* showing major grid lines for months and minor grid lines for weeks). In these cases it is favourable not to show the major grid lines in the „combined“ grid line mode because they are not in synch with the minor grid lines. This kind of logic can be expressed by the grid line policy.

Policy Method	Purpose
`boolean isMajorGridLinesVisible( IDatelineModel model, GridLineMode mode);`	Returns true if the grid lines for the currently shown major time granularity are visible when the given grid line mode was chosen by the user.
`boolean isMinorGridLinesVisible( IDatelineModel model, GridLineMode mode);`	Returns true if the grid lines for the currently shown minor time granularity are visible when the given grid line mode was chosen by the user.

The following are the default implementations of the two grid line policy methods. They are very straight forward.

```
public boolean isMajorGridLinesVisible(IDatelineModel datelineModel,
               GridLineMode mode) {
       switch (mode) {
       case COMBINED_GRID_LINES:
       case MAJOR_GRID_LINES:
               return true;
       case MINOR_GRID_LINES:
       case NO_GRID_LINES:
               return false;
       }
       return false;
}

public boolean isMinorGridLinesVisible(IDatelineModel datelineModel,
               GridLineMode mode) {
       switch (mode) {
       case COMBINED_GRID_LINES:
       case MINOR_GRID_LINES:
               return true;
       case MAJOR_GRID_LINES:
       case NO_GRID_LINES:
               return false;
       }
       return false;
}
```

The specialization *TimeGranularityGridLinePolicy* covers the case described above. The implementation ensures that major grid lines are not visible if the minor granularity displayed in the dateline is „weeks“:

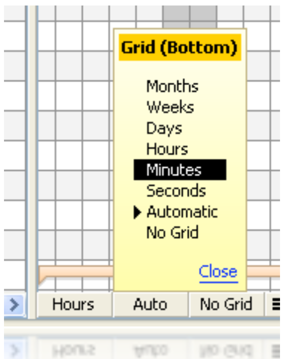
```
public boolean isMajorGridLinesVisible(IDatelineModel datelineModel,
               GridLineMode mode) {
       assertClass("isMajorGridLinesVisible", "datelineModel",
                      TimeGranularityDatelineModel.class, datelineModel);
       switch (mode) {
       case MAJOR_GRID_LINES:
               return true;
       case COMBINED_GRID_LINES:
               TimeGranularityDatelineModel tgModel = (TimeGranularityDatelineModel) datelineModel;
               TimeGranularity tg = tgModel.getGranularity();

               /*
                * The dateline model says that it is displaying weeks. In this case
                * we don't want to see major grid lines.
                */
               if (tg.equals(TimeGranularity.WEEK)) {
                      return false;
               }
               return true;
       case MINOR_GRID_LINES:
               return false;
       case NO_GRID_LINES:
               return false;
       }
       return false;
}
```

IGridPolicy

```
Implementations: AbstractGridPolicy
                 TimeGranularityGridPolicy
                 SimpleGranularityGridPolicy
       Commands: N/A
        Used By: GanttChartStatusBar
                 Timeline
                 Eventline
                 GridSelector
                 DragLayer
                 LassoLayer
```

The grid policy (do not confuse with grid line policy) is used to realize a virtual and invisible grid on the *Eventline* and the *LayerContainer* component. The grid is used to easily position and align timeline objects during editing operations (changing the start time, the duration, or the node / row). The dragged timeline object snaps to the grid lines when the user releases the mouse button.

The Gantt chart returns an array of those components that implement the *IGridComponent* interface. Normally these are a single *Eventline* instance and one or more *LayerContainer* instances. The *GridControlPanel* then creates a *GridControl* for each one of these grid-supporting controls. When clicked the *GridSelector* appears and the user can select the desired grid settings.



Grid Selector

The grid line policy provides several methods for calculating grid-adjusted time points. One for the start time, one for the end time, and one for a time span. When implemented each one of these methods need to respect the „autoGrid“ flag passed to them. This flag controls whether the granularity used for the grid will be the one passed to the method or the one currently shown in the dateline.

Policy Method	Purpose
`IGranularity[] getGridGranularities();`	Returns an array of all granularities that are supported by the grid. The user gets the option to select one of these granularities for the next editing operation. The *GridSelector* class lists the granularities.
`long getGridAdjustedStartTime( IGranularity granularity, long unadjustedStartTime, IDatelineModel<IGranularity> model, boolean autoGrid);`	Calculates a start time (e.g. for a timeline object) based on a granularity and time point. The method will use the currently displayed minor time granularity if the „autoGrid“ flag is true and the passed granularity otherwise.
`long getGridAdjustedEndTime( IGranularity granularity, long unadjustedEndTime, IDatelineModel<IGranularity> model, boolean autoGrid);`	Calculates an end time (e.g. for a timeline object) based on a granularity and time point. The method will use the currently displayed minor time granularity if the „autoGrid“ flag is true and the passed granularity otherwise.
`ITimeSpan getGridAdjustedTimeSpan( IGranularity granularity, ITimeSpan unadjustedTimeSpan, IDatelineModel<IGranularity> model, boolean autoGrid);`	Calculates a time span (e.g. for a timeline object) based on a granularity and time point. The method will use the currently displayed minor time granularity if the „autoGrid“ flag is true and the passed granularity otherwise.

Internally the policy implementation *TimeGranularityGridPolicy* uses the same calculation for the start and the end times:

```
private long getGridAdjustedTime(TimeGranularity tg, long time,
               IDatelineModel<TimeGranularity> datelineModel, boolean autoGrid) {

       /*
        * If the automatic grid is enabled then we use the granularity that is
        * currently used by the dateline.
        */
       if (autoGrid) {
               tg = datelineModel.getGranularity();
       }

       /*
        * No granularity whatsoever means no work, simply return the original
        * time point.
        */
       if (tg == null) {
               return time;
       }

       /*
        * Calculate the time point to the left (earlier) of the given time.
        */
       Date d1 = new Date(time);
       tg.adjustDate(d1);
       long t1 = d1.getTime();

       /*
        * Calculate the time point to the right (later) of the given time.
        */
       Date d2 = new Date(time);
       tg.adjustDate(d2);
       tg.increment(d2);
       long t2 = d2.getTime();

       /*
        * Return the time point that is 'closer' to the given time.
        */
       if (Math.abs(time - t1) < Math.abs(time - t2)) {
               return t1;
       }
       return t2;
}
```

This is how *SimpleGranularityGridPolicy* calculates the adjusted time point:

```
private long getGridAdjustedTime(SimpleGranularity sg, long time,
               IDatelineModel<SimpleGranularity> model, boolean autoGrid) {
       if (autoGrid) {
               sg = model.getGranularity();
```

```
        }
        long result = sg.adjust(time);
        return result;
}
```

ILabelPolicy

```
Implementation: DefaultLabelPolicy
      Commands: N/A
       Used By: LabelLayer
                GanttChartStatusBar
                DefaultDragInfoRenderer
```

The label policy servers a very simple purpose. It is used to lookup various labels for timeline objects and relationships. These labels can then be displayed in the statusbar or in the label layer (e.g. the text to the right of a timeline object).

Policy Method	Purpose

`boolean isLabelTypeVisible(`
`TimelineObjectPath path,`
`IGanttChartModel model,`
`LabelType type);`
	Checks whether the given label type is visible at all. This is, for example, useful when some timeline objects want their status to be shown in the status bar and others don‘t (because they are not relevant for scheduling activities).

`String getLabel(`
`TimelineObjectPath path,`
`IGanttChartModel model,`
`LabelType type);`
	Returns a piece of text for the given label type and timeline object.

`String getLabel(`
`IRelationship relationship,`
`IGanttChartModel model);`
	Returns a piece of text for the given label type and relationship.

ILinePolicy

```
Implementation: DefaultLinePolicy
      Commands: N/A
       Used By: OverviewPalette
                TimelineObjectLayer
                LassoLayer
                DefaultRowRenderer
```

The line policy is used for the „multi-line“ feature, which allows applications to place timeline objects on different lines within the same row. This feature is very useful when timeline objects can overlap each other. The line policy is often used in combination with the row policy (*IRowPolicy*) because in most cases the row height has to be adjusted when changing the line count. More lines usually need more space.

The default behaviour of the policy is to equally distribute the lines across the height of the row. If, for example, the line count of a row is 5 and the row height is 100, then the rows will be located at y = 0, 20, 40, 60 and 80. Each line will have a height of 100 / 5 = 20. This, however, can be altered by custom implementations of this policy. Lines can be located anywhere, they can overlap, and they can all have their own height.

Policy Method	Purpose

`int getLineCount(`
`Object node,`
`ITreeTableModel model);`
	Returns the number of lines inside the row of the given node.

`int getLineIndex(`
`Object node,`
`ITreeTableModel model,`
`Object timelineObject);`
	Returns the line index of the given timeline object. This index is used to place the timeline object on its line. The index range always starts with 0 and ends with line count minus 1. A line index of -1 will place a timeline object on the entire row. An exception will be thrown if a timeline object returns an invalid index.

Policy Method	Purpose
`int getLineLocation( Object node, ITreeTableModel model, int lineIndex, int rowHeight);`	Returns the y-coordinate of the given line. Lines can be located anywhere within the row. Lines may even overlap.
`int getLineHeight( Object node, ITreeTableModel model, int lineIndex, int rowHeight);`	Returns the height of the given line. Each line can have its own height.
`boolean isLineVisible( Object node, ITreeTableModel model, int lineIndex);`	Determines whether the line will be visible or not. The timeline objects on the line will always be shown even if this method returns false. This flag only controls whether a horizontal gray line will be visible inside the row, which helps to distinguish between the lines.

The default implementation of this policy delegates to the interfaces *IGanttChartNode* and *ITimelineObject*.

```
int IGanttChartNode.getLineCount()
int IGanttChartNode.getLineLocation(int lineIndex, int rowHeight);
int IGanttChartNode.getLineHeight(int lineIndex, int rowHeight);

int ITimelineObject.getLineIndex();
```

The default implementations of the *IGanttChartNode* methods are shown here:

```
/**
 * Specifies how many lines will be shown within the node's row. Timeline
 * objects can be placed on any one of these lines.
 *
 * @param count
 *           the numer of lines shown within the node's row
 */
public void setLineCount(int count) {
       if (count < 0) {
              throw new IllegalArgumentException("illegal line count " + count
                            + " (must be larger than or equal to 0)");
       }
       this.lineCount = count;
}

/*
 * @see com.dlsc.flexgantt.model.gantt.IGanttChartNode#getLineCount()
 */
public int getLineCount() {
       return lineCount;
}

/*
 * @see com.dlsc.flexgantt.model.gantt.IGanttChartNode#getLineLocation(int,
 *      int)
 */
public int getLineLocation(int lineIndex, int rowHeight) {
       int count = getLineCount();
       if (lineIndex >= count) {
              throw new IllegalArgumentException("illegal line index "
                            + lineIndex + " (line count = " + count + ")");
              }
       if (lineIndex >= 0) {
              double h = (double) rowHeight / (double) count;
              return (int) (lineIndex * h);
       }
       return 0;
}

/*
 * @see com.dlsc.flexgantt.model.gantt.IGanttChartNode#getLineHeight(int,
 *      int)
 */
public int getLineHeight(int lineIndex, int rowHeight) {
       int count = getLineCount();
       if (lineIndex >= count) {
              throw new IllegalArgumentException("illegal line index "
                                + lineIndex + " (line count = " + count + ")");
        }
        if (lineIndex >= 0) {
                if (lineIndex < count - 1) {
                        return getLineLocation(lineIndex + 1, rowHeight)
                                        - getLineLocation(lineIndex, rowHeight);
                }
                return rowHeight - getLineLocation(count - 1, rowHeight) - 1;
        }
        return rowHeight;
}
```

IPopupPolicy

```
Implementation: DefaultPopupPolicy
      Commands: N/A
       Used By: PopupLayer
                TreeTable
```

Popups are a very useful feature for providing detailed information about a tree node or timeline object and the business object that either one represent. Popups show up when the mouse cursor hovers over a tree node or timeline object that has a popup value associated with it. A popup value can be any kind of object. It is up to the popup renderers to visualize the value. The *PopupLayer* class is responsible for drawing the popups. Each timeline object can carry a standard and an extended popup value. The user can toggle between these two by pressing or releasing the SHIFT key.

Policy Method	Purpose
`Object getPopupValue( TimelineObjectPath path, IGanttChartModel model, boolean extended);`	Returns the input for the popup for the given timeline object. The „extended“ attribute is used to distinguish between the standard and the extended popup values.
`Object getPopupTitleValue( TimelineObjectPath path, IGanttChartModel model);`	Returns the input for creating a title for the popup of the given timeline object.
`Object getPopupValue( TreePath path, IGanttChartModel model, boolean extended);`	Returns the input for the popup for the given tree node. The „extended“ attribute is used to distinguish between the standard and the extended popup values.
`Object getPopupTitleValue( TreePath path, IGanttChartModel model);`	Returns the input for creating a title for the popup of the given tree ode.

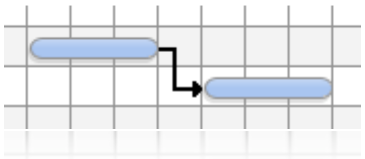
The default implementation of this policy delegates to the *ITimelineObject* and the *IGanttChartNode* interfaces:

```
public class DefaultPopupPolicy extends AbstractPolicy implements IPopupPolicy {

        public Object getPopupValue(TimelineObjectPath path,
                        IGanttChartModel model, boolean extended) {
                assertClass("getPopupValue", "path.getTimelineObject()", ITimelineObject.class,
                                path.getTimelineObject());
                return ((ITimelineObject) path.getTimelineObject())
                                .getPopupObject(extended);
        }

        public Object getPopupTitleValue(TimelineObjectPath path,
                        IGanttChartModel model) {
                assertClass("getPopupValue", "path.getTimelineObject()", ITimelineObject.class,
                                path.getTimelineObject());
                return ((ITimelineObject) path.getTimelineObject())
                                .getPopupTitleObject();
        }

        public Object getPopupValue(TreePath path, IGanttChartModel model,
                        boolean extended) {
                assertClass("getPopupValue", "path.getLastPathComponent()", ITreeTableNode.class,
                                path.getLastPathComponent());
                return ((ITreeTableNode) path.getLastPathComponent())
                                .getPopupObject(extended);
        }

        public Object getPopupTitleValue(TreePath path, IGanttChartModel model) {
                assertClass("getPopupValue", "path.getLastPathComponent()", ITreeTableNode.class,
                                path.getLastPathComponent());
                return ((ITreeTableNode) path.getLastPathComponent())
                                .getPopupTitleObject();
        }
}
```

IRelationshipPolicy

```
Implementation: DefaultRelationshipPolicy
      Commands: DefaultCreateRelationshipCommand
       Used By: LassoLayer
```

FlexGantt uses the very abstract term „relationship“ in order to model such things like predecessors, successors, or constraints (finish-before, start-after, same-end, same-beginning). The *LassoLayer* is used as a controller for letting the user interactively create relationships between timeline objects. In order to do so the layer needs the answer to one question: can the timeline object be used as the source or the target of a relationship? It is the purpose of the relationship policy to answer this question.


Relationship Line

Policy Method	Purpose
`boolean isRelationshipSource( TimelineObjectPath path, IGanttChartModel model);`	Returns true if the given timeline object can be used as the source of a relationship.
`boolean isRelationshipTarget( TimelineObjectPath sourcePath, TimelineObjectPath targetPath, IGanttChartModel model);`	Returns true if the given timeline object can be used as the target of a relationship. The default relationship renderer draws an arrow at the target location.
`ICommand getCreateRelationshipCommand( TimelineObjectPath sourcePath, TimelineObjectPath targetPath, IGanttChartModel model)`	Returns the command that will create the new relationship object and add it to the model.

The default implementation of this policy looks like this:

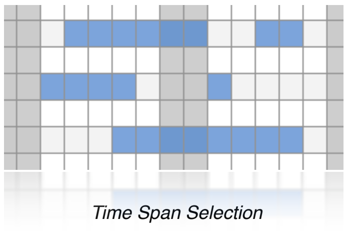
```
public class DefaultRelationshipPolicy extends AbstractPolicy implements IRelationshipPolicy {

        public ICommand getCreateRelationshipCommand(TimelineObjectPath sourcePath,
                        TimelineObjectPath targetPath, IGanttChartModel model) {
                assertClass("getRelationshipCommand", "model", DefaultGanttChartModel.class, model);
                return new DefaultCreateRelationshipCommand(sourcePath, targetPath,
                                (DefaultGanttChartModel) model);
        }

        public boolean isRelationshipSource(TimelineObjectPath path,
                        IGanttChartModel model) {
                return true;
        }

        public boolean isRelationshipTarget(TimelineObjectPath sourcePath,
                        TimelineObjectPath targetPath, IGanttChartModel model) {
                return true;
        }
}
```

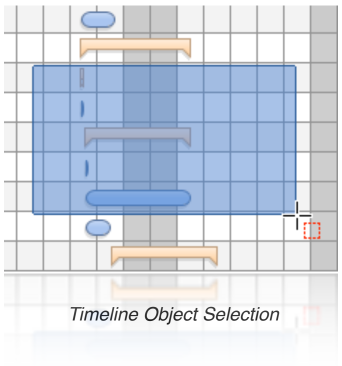
According to this implementation any timeline object can be the source or the target of a new relationship.

ISelectionPolicy

```
Implementation: DefaultSelectionPolicy
      Commands: N/A
       Used By: TimelineObjectLayer
                LassoLayer
```

The selection policy controls two different types of selections that the user can perform: the user can select timeline objects and time spans . Selected time spans are added to the selection model of the LayerContainer[3] as they are independent of the layers currently shown. Timeline objects, however, are added to the selection model of the layer[4] on which they are shown.


Time Span Selection


Timeline Object Selection

Policy Method	Purpose
`boolean isSelectable( Object node, ITimeSpan span, IGanttChartModel model);`	Returns true if the given time span on the given node (row) can be selected by the user. One example where this method is useful is the disabling of selections where the time spans are already in the past.
`boolean isSelectable( TimelineObjectPath path, IGanttChartModel model);`	Returns true if the timeline object given by the path is selectable. Scheduling applications often add timeline objects for decoration purposes only. These objects should not be selectable.

The default implementation of this policy delegates to the *ITimelineObject* interface in order to find out whether an object is selectable or not.

```
boolean ITimelineObject.isSelectable();
```

For time spans the policy delegates to the *IGanttChartNode* interface:

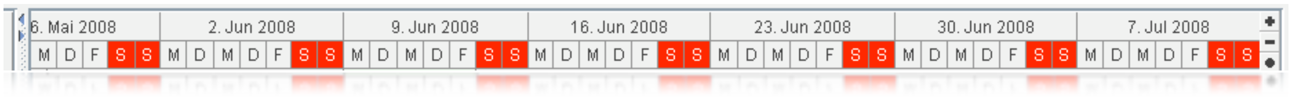
```
boolean IGanttChartNode.isSelectable(ITimeSpan);
```

[3] An instance of *ILayerContainerSelectionModel.* This model only manages time span selections.

[4] Each layer has its own instance of *ITimelineObjectLayerSelectionModel*. This model only manages timeline object selections. The selection model for each layer can be looked up from the *LayerContainer* (*getSelectionModel(ILayer)*).

Policies of Dateline

The dateline currently only supports a single policy type, which is used for zooming operations. However, this might change in future releases as new features are constantly added to the entire framework.


6. Mai 2008
2. Jun 2008
9. Jun 2008
16. Jun 2008
23. Jun 2008
30. Jun 2008
7. Jul 2008
M D F S S M D M D F S S M D M D F S S M D M D F S S M D M D F S S M D M D F S S M D M D F S S

Dateline

IZoomPolicy

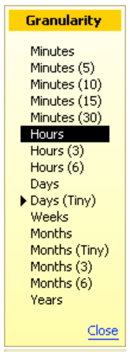
```
Implementations: AbstractZoomPolicy
                 SimpleGranularityZoomPolicy
                 TimeGranularityZoomPolicy
       Commands: N/A
        Used By: TimeGranularityTimelineControlPanel
                 TimeGranularityDatelineModel
                 SimpleGranularityDatelineModel
                 SimpleDateline
                 DefaultDatelineMenuProvider
                 GranularitySelector
```

The dateline supports several zooming operations: **zoom in** - shows only half of the currently visible time span, **zoom out** - makes a time span visible that is twice as long as the current one, **zoom into** - the user can request a specific time span to be made visible, **granularity change** - the user can request a specific time granularity to be shown (weeks, months, ...).

The zoom policy supports these operations by listing all the granularities that are supported by the *Dateline*. The policy implementation *TimeGranularityZoomPolicy* lists only a subset of the values defined by the *TimeGranularity* enumerator (the range from MINUTE to YEAR). This is done by the default constructor, which looks like this:

```
/**
 * Constructs a new policy with a time granularity range of [MINUTE,  YEAR].
 */
public TimeGranularityZoomPolicy() {
       this(TimeGranularity.getRange(TimeGranularity.MINUTE, TimeGranularity.YEAR));
}
```

The abstract super class *AbstractZoomPolicy* ensures that the granularity values are in sorted order. It can do this by invoking the *isSmaller(IGranularity)* and *isLarger(IGranularity)* methods of the *IGranularity* interface. The following snap-shot shows the granularity selector with the default granularity set in place.

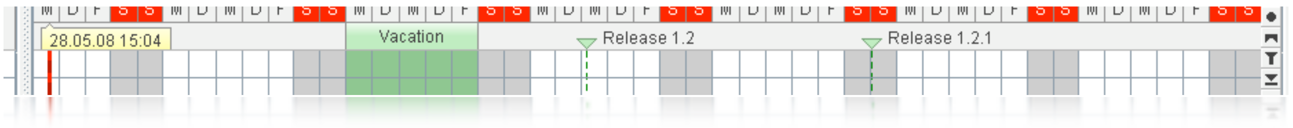


Granularity Selector

Policy Method	Purpose
`public int getGranularityCount();`	Returns the total number of granularities supported by the dateline.
`public IGranularity getGranularity( int index);`	Returns the granularity for the given index.
`public int getGranularityIndex( IGranularity granularity);`	Returns the index for the given granularity.

Policies of Eventline

The eventline defines three policies, each one of them covering a different aspect. One deals with the eventline‘s editing behaviour, one with its selection behaviour, and one with label decoration. The eventline is located underneath the dateline and it is responsible for displaying events and activities that can not be associated with any particular row (e.g. company holidays, milestones, ...).



Eventline

IEditEventlineObjectPolicy

```
Implementation: DefaultEditEventlineObjectPolicy
      Commands: DefaultChangeEventlineObjectTimeSpanCommand
                DefaultCreateEventlineObjectCommand
                DefaultDeleteMultipleEventlineObjectsCommand
       Used By: Eventline
                DefaultEventlineObjectRenderer
```

The objects shown in the *Eventline* component can be edited in a very similar manner to the way timeline objects can be edited in the *LayerContainer* component. Which modifications are allowed gets determined by this policy (start time, duration, deletion, creation). If a modification is allowed by the policy then the appropriate command for performing the actual model change can be retrieved from the policy as well.

Policy Method	Purpose
`boolean isStartTimeChangeable( Object eventlineObject, IEventlineModel model);`	Returns true if the start time of the given eventline object can be changed.
`boolean isDurationChangeable( Object eventlineObject, IEventlineModel model);`	Returns true if the duration of the given eventline object can be changed.
`boolean isDeletable( Object eventlineObject, IEventlineModel model);`	Returns true if the given eventline object can be deleted.
`boolean isCreatable( IEventlineModel model, ITimeSpan span);`	Returns true if the user is allowed to interactively create a new eventline object for the given time span.
`ICommand getCreateCommand( IEventlineModel model, ITimeSpan span);`	Returns the command that shall be used to create a new eventline object for the given model and time span.
`ICommand getChangeTimeSpanCommand( Object eventlineObject, IEventlineModel model, ITimeSpan span);`	Returns the command that shall be used to modify the time span of the given eventline object.
`ICommand getDeleteCommand( Collection<Object> eventlineObjects, IEventlineModel model);`	Returns the command that shall be used to delete the given eventline objects. The command needs to be able to delete several eventline objects at the same time.

The default implementation of this policy delegates all boolean methods to the IEventlineObject interface.

IEventlineLabelPolicy

```
Implementation: DefaultEventlineLabelPolicy
      Commands: N/A
       Used By: Eventline
                GanttChartStatusBar
                DefaultEventlineObjectRenderer
```

This label policy can be used to lookup various pieces of text that can be used to decorate an eventline object.

Policy Method	Purpose
`String getLabel( Object eventlineObject, IEventlineModel model, LabelType type);`	Returns a text for the given label type. The *LabelType* enumerator lists various possible types of text (NAME, TOOLTIP, POPUP_TITLE, ...). Currently only the type NAME is used by the *Eventline*. Future releases might make use of additional types. However, if no text is specified for a given type, then by default the NAME text will be used.

The following code fragment shows the default implementation of this policy and how it delegates to the *IEventlineObject* interface.

```
public String getLabel(Object eventlineObject, IEventlineModel model,
               LabelType type) {
       assertClass("getLabel", "eventlineObject", IEventlineObject.class, eventlineObject);
       return ((IEventlineObject) eventlineObject).getLabel(type);
}
```

IEventlineSelectionPolicy

```
Implementation: DefaultEventlineSelectionPolicy
      Commands: N/A
       Used By: Eventline
```

The only purpose of this policy is to determine whether a given eventline object currently displayed in the Eventline can be selected by the user or not.

Policy Method	Purpose
`boolean isSelectable( Object eventlineObject, IEventlineModel model);`	If this method returns true then the user will be able to select the given eventline object.

This policy, too, delegates to the IEventlineObject interface:

```
public boolean isSelectable(Object eventlineObject, IEventlineModel model) {
       assertClass("isSelectable()", "eventlineObject",  IEventlineObject.class, eventlineObject);
       return ((IEventlineObject) eventlineObject).isSelectable();
}
```

Appendix: „How to Correctly and Uniformly Use Progress Monitors“

Handling a progress monitor instance is deceptively simple. It seems to be straightforward but it is easy to make a mistake when using them. And, depending on numerous factors such as the underlying implementation, how it is displayed, if it’s set to use a fixed number of work items or ‘unknown’, if used through a SubProgressMonitor wrapper etc., the result can range from completely ok, mildly confusing or outright silliness.

In this article I hope I can lay down a few ground rules that will help anyone use progress monitors in a way that will work with the explicit and implicit contract of IProgressMonitor. Also, understanding the usage side makes it easier to understand how to implement a monitor.

By Kenneth Ölwing, BEA JRPG
January 18, 2006

Using a progress monitor - what's up with that?

It all really comes down to a few, not too complex, rules. A common theme is 'know what you know - but only that'. This means that you shouldn't assume you know things you really don't know, and this includes the common mistake of only considering progress monitors you have seen, i.e. typically the graphical ones when using the IDE. Another thing to watch out for is the fact that commonly you design a number of tasks that may call each other using sub progress monitors, and while doing that make assumptions based on your knowledge that they will be called in this manner - never forget that sometime maybe your separate subtasks may be called from not-yet-written routines. It's then vitally important that your subtasks act exactly in a 'neutral' manner, i.e. with no 'implicit assumptions' on what happened before or what will happen after.

One of the motivations for this article is when I tried my hand at implementing a progress monitor intended for headless/console use - and realised that code using it could make it look really wacky when the monitor was wrongly used, and this was issues that were not as readily apparent with a graphical monitor. Also, code (including my own) frequently abuses the explicit and implicit (which admittedly are my interpretation of reasonable behavior) contract that the IProgressMonitor interface states, and this makes for dicey decisions for a monitor implementor - should it complain (and how) when it gets conflicting orders? If not, how should it then behave to make for a reasonable and intuitive user experience?

The protocol of IProgressMonitor

Generally, all interaction with a progress monitor is through the interface IProgressMonitor and this interface defines the protocol behavior expected. It does leave some things up in the air though; for example, the description states some things that should be true, but the methods have no throws clause that helps enforce some invariants. I have chosen to interpret the descriptions ‘hard’, even to the point of saying it’s valid to throw an (unchecked) exception if a described rule is violated (this is somewhat controversial of course - if you implement a monitor doing this you should probably provide a way to turn off 'strictness'). Hopefully we could eventually see a new interface that deprecates the old methods and provides new ones that better reflect the contract. The discussion below is based on the assumption that the reader is familiar with the general API; review it in the Eclipse help.

The first important consideration is the realization that a monitor (contract wise) can be in basically four states. Any given implementation may or may not track those state changes and may or may not do anything about them, which is part of the reason that misbehaving users of a monitor sometimes gets away with it. Only one of these states are readily testable using the interface however (if the monitor is canceled); the other states are just a given from correct use of the interface.

Essentially, the state changes are governed by the methods beginTask(), done() and setCanceled(), plus the implicit initial state of a new instance. Note that for the purposes discussed here the perceived ‘changes in state’ occurring as a result from calling worked() is not relevant. A separate discussion below details how to deal with worked() calls.

Please note that the states described here are not any ‘officialese’ that can be found as constants or anything like that; they’re only here to serve so they can be used for discussion.

- PRISTINE - This is the initial state of a newly created instance of an IProgressMonitor implementation, i.e. before beginTask() has been called. In principle a given implementation may handle a single instance such that it is reusable and reverted back to the PRISTINE state after a done() call, but that is opaque from the point of the contract. In this state it should be essentially correct and possible to go to any of the other states, but the typical and expected transition should be from PRISTINE to IN_USE as a result from a successful beginTask() call. The transition to FINISHED should result only in a very particular situation, see more below.

- IN_USE - This is the state the monitor after the first and only call to beginTask(). This is one of those things that are very easy to get wrong; contract wise, beginTask() can and should only be called at most once for a given instance. A more detailed discussion on the code pattern required to deal with this obligation can be found below.
- FINISHED - The transition to this state is achieved by calling done(). As with beginTask(), done() should only be called once and should always be called on a monitor when beginTask() has been called (i.e. it is ok to not call done() only if the monitor is still in the PRISTINE state). Again, the discussion below is more detailed on how to ensure proper protocol.
- CANCELED - Actually, this state is slightly murky; it's possible that canceled/not canceled should be tracked separately from the others. But, contract wise it should be adequate if this state is either achieved directly from PRISTINE and just left that way, or if done() is called (likely as a result of detecting the canceled status), it is cleared and the monitor then transitions to FINISHED.

Now, one contract pattern described above is that if beginTask() is ever called, done() MUST be called. This is achieved by always following this code pattern (all code is simplified):

```
monitor = … // somehow get a new progress monitor which is in a pristine state
// figure some things out such as number of items to process etc…
try
    {
    monitor.beginTask(…)
    // do stuff and call worked() for each item worked on, and check for cancellation
    }
finally
    {
    monitor.done()
    }
```

The important thing here then is to ensure that done() is always called (by virtue of being in the finally clause) but (normally) only if beginTask() has been successfully called (by virtue of being the first thing called in the try clause). There is a small loophole that could cause done() to be called without the monitor actually transitioning from PRISTINE to IN_USE. This loophole can with this pattern only happen if a particular beginTask() implementation throws an unchecked exception (The interface itself declares no throws clause) before it internally makes a note of the state change (if the specific implementation even tracks state in this manner and/or is too loose in its treatment of the interface contract).

Arguably, you should always strive for calling beginTask()/done(). The reasons for this are buried in the fact that you in principle never know when you are being called as a subtask. If you don't 'complete' the monitor, the parent can end up with an incorrect count for its own task. The full rationale is covered more below, in the section "Ensure to always complete your monitor!".

Delegating use of a progress monitor to subtasks

Above for the IN_USE state I mentioned that it's very easy to get things wrong; beginTask() should never be called more than once. This frequently happens in code that doesn't correctly understand the implications of the contract. Specifically, such code pass on the same instance it has been given to subtasks, and those subtasks; not aware that the caller already has begun following the contract, also tries following the contract in the expected manner – i.e. they start by doing a beginTask().

Thus, passing on a monitor instance is almost always wrong unless the code knows exactly what the implications are. So the rule becomes: In the general case, a piece of code that has received a progress monitor from a caller should always assume that the instance they are given is theirs and thus completely follow the beginTask()/done() protocol, and if it has subtasks that also needs a progress monitor, they should be given their own monitor instances through further use of the SubProgressMonitor implementation that wraps the 'top-level' monitor and correctly passes on worked() calls etc (more on this below).

Sample code to illustrate this:

```
monitor = … // somehow get a new progress monitor which is in a pristine state

// figure some things out such as number of items to process etc…
try {
       monitor.beginTask(…)
       // do stuff and call worked() for each item processed, and check for cancellation
        …
       // farm out a piece of the work that is logically done by 'me' to something else
       someThing.doWork(new SubProgressMonitor(monitor,…))
       // farm out another piece of the work that is logically done by 'me' to something else
       anotherThing.doWork(new SubProgressMonitor(monitor,…))
} finally {
        monitor.done()
}
```

Note that each doWork() call gets a new instance of a SubProgressMonitor; such instances can and should not be re-used for all the protocol reasons already discussed.

The only time a single instance of a monitor passed to, or retrieved by, a certain piece code can be reused in multiple places (e.g. typically methods called by the original receiver), is when the code in such methods is so intimately coupled so that they in effect constitute a single try/finally block. Also, for this to work each method must know exactly who does beginTask()/done() calls, and also (don't forget this) how many work items they represent of the total reported to beginTask() so that they can make the correct reports. Personally, I believe this is generally more trouble than it's worth – always follow the regular pattern of one receiver, one unique monitor instead and the code as a whole is more maintainable.

Managing the item count

This section is about how to do the initial beginTask() call and report the amount of total work expected, and then ideally report exactly that many items to the monitor. It is ok to end up not reporting all items in one particular case: when the job is aborted (due to cancellation by user, an exception thrown and so on) – this is normal and expected behavior and we will wind up in the finally clause where done() is called.

It is however sloppy technique to 'just pick a number' for the total and then call worked(), reporting a number and hope that the total is never exceeded. Either way this can cause very erratic behavior of the absolute top level and user visible progress bar (it is for a human we're doing this after all) – if the total is too big compared to the actual items reported, a progress bar will move slowly, perhaps not at all due to scaling and then suddenly (at the done() call) jump directly to completed. If the total is too small, the bar will quickly reach '100%' or very close to it and then stay there 'forever'.

So, first and foremost: do not guess on the number of work items. It's a simple binary answer: either you know exactly how many things that will be processed…or you don't know. It IS ok to not know! If you don't know, just report IProgressMonitor.UNKNOWN as the total number, call worked() to your hearts content and a clever progress monitor implementation will still do something useful with it. Note that each (sub)task can and should make its own decision on what it knows or not. If all are following the protocol it will ensure proper behavior at the outer, human visible end. A heads up though: never call the SubProgressMonitor(parentMonitor, subticks) constructor using IProgressMonitor.UNKNOWN for subticks - this is wrong! More on this later.

How to call beginTask() and worked()

There are typically two basic patterns where you know how many items you want to process: either you are going to call several different methods to achieve the full result, or you are going to call one method for each instance in a collection of some sort. Either way you know the total item count to process (the number of methods or the size of the collection). Variations of this are obviously combinations of these basic patterns so just multiply and sum it all up.

There is sometimes a benefit of scaling your total a bit. So, instead of reporting '3' as the total (and do worked(1) for each item) you may consider scaling with, say 1000, and reporting '3000' instead (and do worked(1000) for each item). The benefit comes up when you are farming out work to subtasks through a SubProgressMonitor; since they may internally have a very different total, especially one that is much bigger than your total, you give them (and the monitor instance) some 'room' to more smoothly consume and display the allotment you've given them (more explanations below on how to mix worked() and SubProgressMonitor work below). Consider that you say 'my total is 3' and you then give a subtask '1' of these to consume. If the subtask now will report several thousand worked() calls, and assuming that the actual human visible progress bar also has the room, the internal protocol between a SubProgressMonitor and it's wrapped monitor will scale better and give more smooth movement if you instead would have given it '1000' out of '3000'. Or not - the point is really that you don't know what monitor implementation will be active, all you can do is give some information. How it's then displayed in reality is a matter of how nifty the progress monitor implementation is.

A sample of simple calls:

```
monitor = … // somehow get a new progress monitor which is in a pristine state
int total = 3 // hardcoded and known
try
    {
    monitor.beginTask(total)

    // item 1
    this.doPart1()
    monitor.worked(1)
    // item 2
    this.doPart2()
    monitor.worked(1)

    // item 3
    this.doPart3()
    monitor.worked(1)
    }
finally
    {
    monitor.done()
    }
```

No reason to scale and no collection to dynamically compute.

A more elaborate sample:

```
monitor = … // somehow get a new progress monitor which is in a pristine state
int total = thingyList.size() * 3 + 2
try {
    monitor.beginTask(total)

    // item 1
    this.doBeforeAllThingies()
    monitor.worked(1)

    // items 2 to total-1
    for (Thingy t : thingyList) {
        t.doThisFirst()
        monitor.worked(1)
        t.thenDoThat()
        monitor.worked(1)
        t.lastlyDoThis()
        monitor.worked(1)
        }

    // final item
    this.doAfterAllThingies()
    monitor.worked(1)
    } finally {
           monitor.done()
    }
```

Mixing straightforward calls with subtasks

I was initially confused by how to report progress when I farmed out work to subtasks. I experienced 'reporting too much work' since I didn't understand when to call and when to not call worked(). Once I caught on, the rule is very simple. However: calling a subtask with a SubProgressMonitor is basically an implicit call to worked() with the amount allotted to the subtask. So instead of this:

```
monitor = … // somehow get a new progress monitor which is in a pristine state
int scale = 1000
int total = 3 // hardcoded and known
try
    {
    monitor.beginTask(total * scale)

    // item 1
    this.doPart1()
    monitor.worked(1 * scale)

    // item 2
    this.doPart2(new SubProgressMonitor(monitor, 1 * scale)) // allot 1 item
    monitor.worked(1 * scale) // WRONG! Not needed, already managed by the SubProgressMonitor

    // item 3
    this.doPart3()
    monitor.worked(1 * scale)
    } finally  {
          monitor.done()
    }
```

You should just leave out the second call to worked().

Never pass IProgressMonitor.UNKNOWN (or any other negative value) when creating a SubProgressMonitor() wrapper!

A situation I just the other day experienced was when doing an IProgressMonitor.UNKNOWN number of things - I needed to call a subtask, and hence I set up to call it using a SubProgressMonitor(parent, subticks) but I realized that I hadn't ever considered how the sub monitor should be created - how many subticks it should be given - in the unknown case. I figured it should be ok to pass IProgressMonitor.UNKNOWN there also. However, when later trying my code I saw to my horror that my progress bar went backwards! Not the effect I figured on...

As it turns out, this is because the implementation (as of Eclipse 3.2M3) blindly uses the incoming ticks as a scaling factor. However, it goes haywire when it receives a negative value (and IProgressMonitor.UNKNOWN happens to have a value of -1). It does computations with it, and it ends up calling worked() with negative values which my monitor tried to process...that code is now fixed to be more resilient in such cases. I've filed bug #119018 to request that SubProgressMonitor handles it better and/or document that negative values is a bad idea for the constructor call.

Whatever, passing IProgressMonitor.UNKNOWN is incorrect in any case. If you have called beginTask() using IProgressMonitor.UNKNOWN you can gladly pass in any reasonable tick value to a SubProgressMonitor, it will give the correct result.

Ensure to always complete your monitor!

Consider the concept described in the previous section: the important thing here is that basically, you say that you have three distinct and logical things to do, and then you tick them off - but one of the ticks is actually farmed out to a subtask through a SubProgressMonitor. You don't really know how many distinct and logical things the subtask has to do, nor should you care. The mechanics of using a SubProgressMonitor makes the advancement of one of your ticks happen in the correct way. So, the end expectation is that once you reach the end of your three things, the monitor you have, have actually fulfilled the count you intended - the internal state of it should reflect this: "the user said three things should happen and my work count is now indeed '3'".

But, as I recently found out, this can fail. Specifically, I blindly invoked IProject.build() on a project which had no builders configured. To this method I sent in a SubProgressMonitor and allotted it one 'tick' of mine. But, as it turned out, internally it never used the monitor it got, presumably because there was no work to perform - not very unreasonable in a sense. However, this did have the effect that one of my ticks never got, well, 'tocked' :-). I could solve this specific problem by simply checking if there was any builders configured, and if there were none, I simply advanced the tick by worked(1) instead. But, it requires me, the caller, to make assumptions on the internal workings of the subtask, which is never good.

This is not a huge problem of course. But, I think it would make sense to always act the same. The code resulting from IProject.build() could just call beginTask("", countOfBuilders) regardless of if countOfBuilders was 0, iterate over the empty array or whatever, and then call done(). This would correctly advance my tick.

Cancellation

The sample code above does not show cancellation checks. However, it is obviously recommended that users of a progress monitor actively check for cancellation to timely break out of the operation. The more (potentially) long-running, the more important of course. And remember: you don't know if the operation is running in a context that allows it to be canceled or not - so you just have to code defensively. A sample of how it should look could be this:

```
monitor = … // somehow get a new progress monitor which is in a pristine state
try
    {
    monitor.beginTask(thingyList.size())

    for (Thingy t : thingyList)
        {
        if(monitor.isCanceled())
            throw new OperationCanceledException();
        t.doSomething()
        monitor.worked(1)
        }
    } finally {
    monitor.done()
}
```

The NullProgressMonitor

A common pattern is to allow callers to skip sending a monitor, i.e. sending ‘null’. A simple and convenient way to deal with such calls is this:

```
public void doIt(IProgressMonitor monitor) {
    // ensure there is a monitor of some sort
    if(monitor == null)
        monitor = new NullProgressMonitor();

    try {
        monitor.beginTask(thingyList.size())

        for (Thingy t : thingyList)
            {
            if(monitor.isCanceled())
                throw new OperationCanceledException();
            t.doSomething()
            monitor.worked(1)
            }
    } finally {
        monitor.done()
    }
}
```

Conclusion

I believe that by diligently following these rules and patterns, you will never have a problem in using the progress monitor mechanism. Obviously, it requires implementations to follow the contract as well. But remember, if you mistreat the protocol you will sooner or later end up talking to a progress monitor implementation that is stern and will simply throw an exception or give strange visual effects if you call it's beginTask() one time too many. It's essentially valid if the IProgressMonitor interface description is to be believed – and you will get blamed by your customer…