IBM ILOG JViews TGO V8.6

Graphic components

Table of contents

Introducing graphic components

Introduces each graphic component provided by IBM® ILOG® JViews TGO. These components are ready-to-use graphic components that let you represent business objects and data as a network of nodes and links (the network component),as items of equipment composed of cards, ports, and LEDs (the equipment component), as a tree structure (the tree component), or in a two-dimensional table format (the table component).

In this section

The network component
Introduces the network component, which shows network nodes interconnected by links.

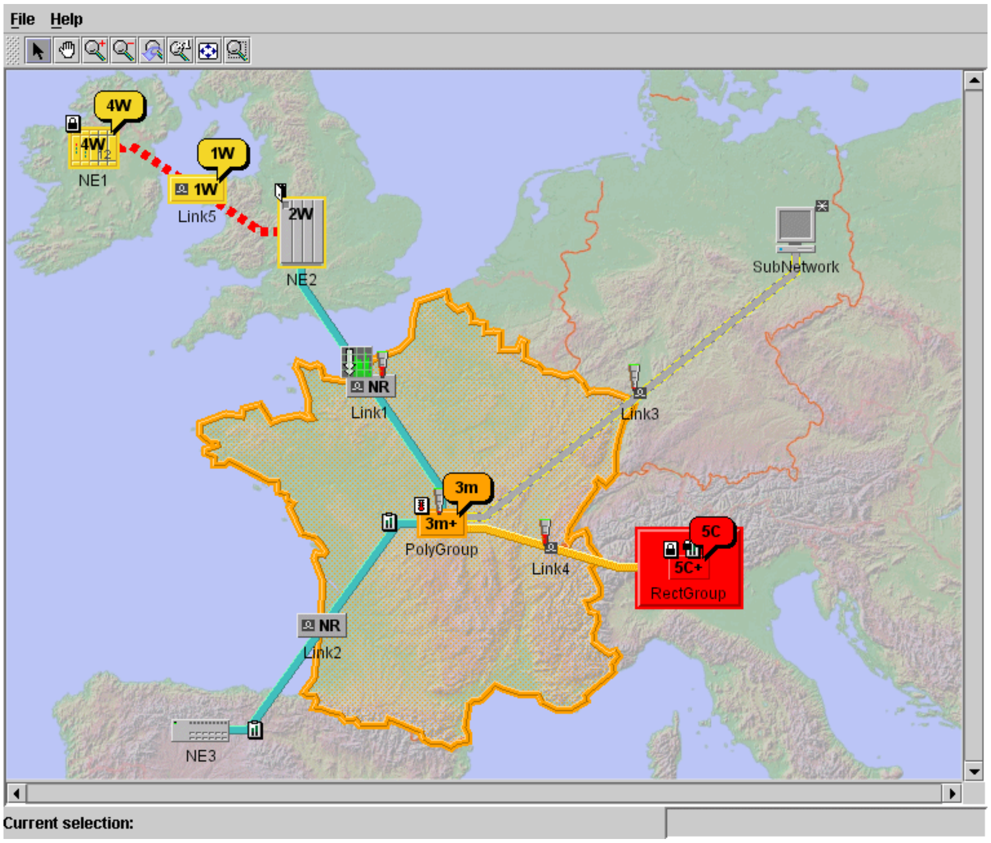
The equipment component
Introduces the equipment component, which displays items of equipment such as cards, shelves, ports, and LEDs.

The table component
Introduces the table component, which displays data in a two-dimensional table format.

The tree component
Introduces the tree component, which displays data in a hierarchical representation.

The network component

The network component is based on the IBM® ILOG® JViews grapher. It shows network nodes interconnected by links. The network component has support for editing the network, navigation, automatic layout of nodes and links, and background maps. It also supports pop-up menus and tooltips.

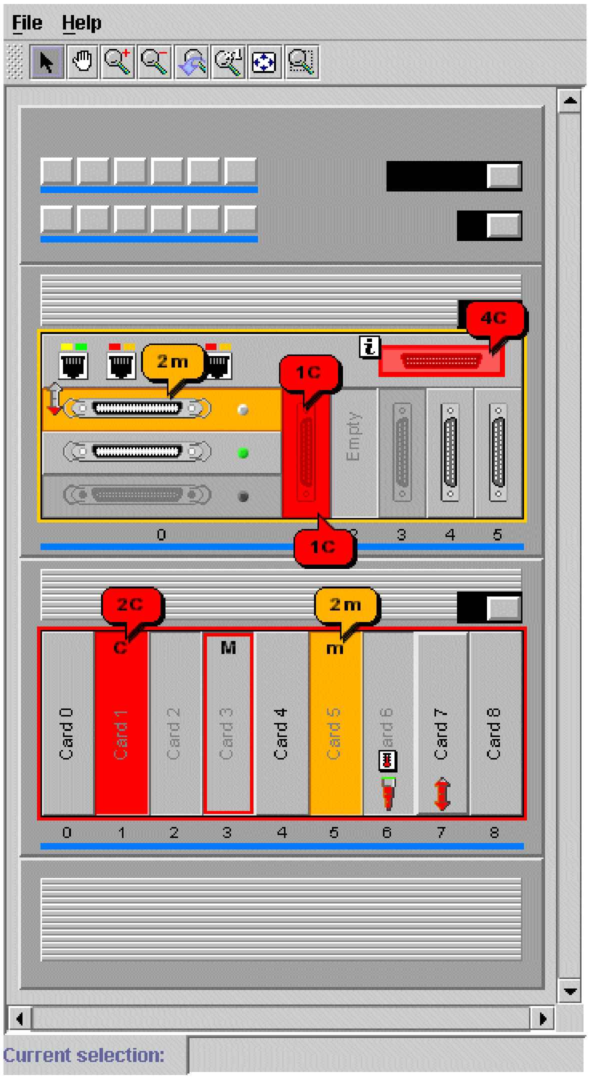


Network component

The network component can be configured either through a CSS file, where you define all its associated settings (map displayed in the background, display of toolbar or overview window, zoom policy, and so on), or through an API.

The equipment component

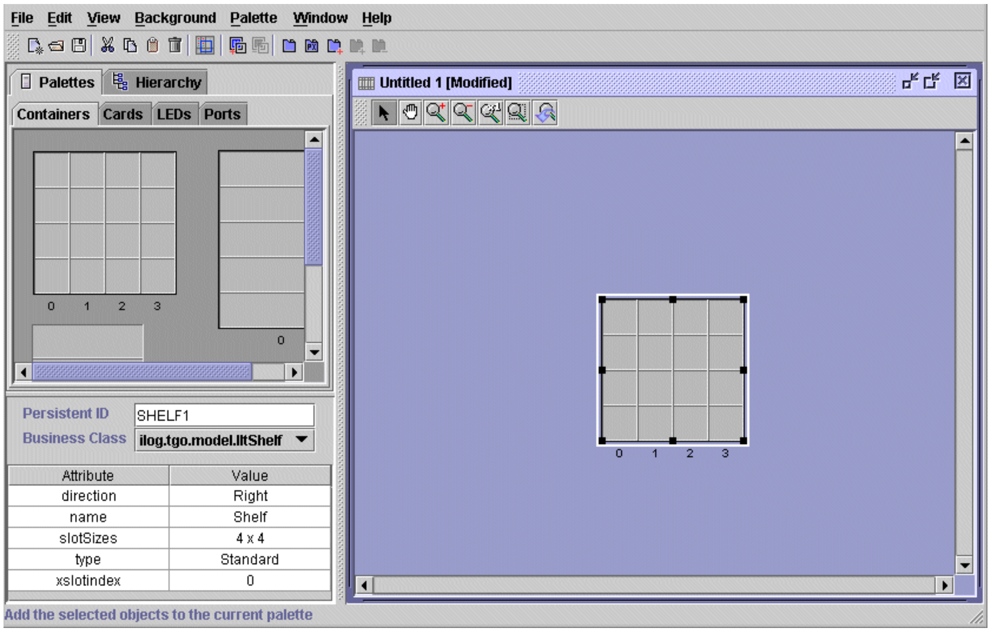
Like the network component, the equipment component is based on IBM® ILOG® JViews. It allows you to display items of equipment such as cards, shelves, ports, and LEDs. It also supports pop-up menus and tooltips.



Equipment component

Like the network component, the equipment component can be configured either through a CSS file, where you define all its associated settings (map displayed in the background, toolbar, and so on), or through an API.

JViews TGO includes an equipment editor, a graphic user interface, that allows you to build an equipment component in a very easy and user-friendly manner.



Persistent ID	SHELF1
Business Class	ilog.tgo.model.IltShelf

Attribute	Value
direction	Right
name	Shelf
slotSizes	4 x 4
type	Standard
xslotindex	0

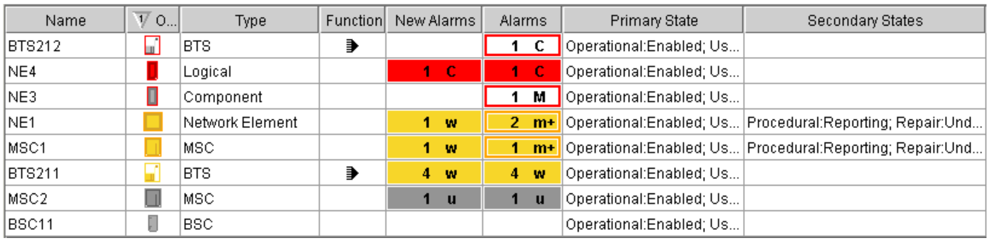
Add the selected objects to the current palette

The equipment editor

The table component

The table component is based on the Swing table component. It allows you to display data in a two-dimensional table format. Business objects are displayed in table rows, while their associated properties appear in separate columns.

The table component features smart resizing modes, multiple selection, and sorting, as well as filtering and searching capabilities. It also supports pop-up menus and tooltips.

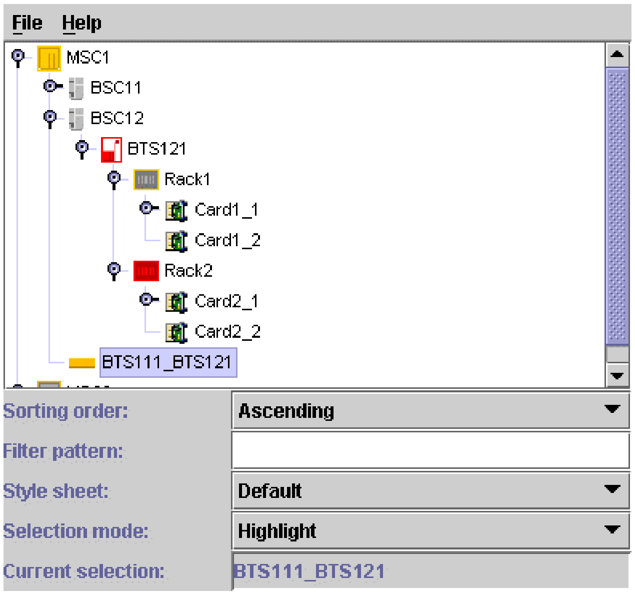


Name	O...	Type	Function	New Alarms	Alarms	Primary State	Secondary States
BTS212	■	BTS	➧		1 C	Operational:Enabled; Us...	
NE4	■	Logical		1 C	1 C	Operational:Enabled; Us...	
NE3	■	Component			1 M	Operational:Enabled; Us...	
NE1	■	Network Element		1 w	2 m+	Operational:Enabled; Us...	Procedural:Reporting; Repair:Und...
MSC1	■	MSC		1 w	1 m+	Operational:Enabled; Us...	Procedural:Reporting; Repair:Und...
BTS211	■	BTS	➧	4 w	4 w	Operational:Enabled; Us...	
MSC2	■	MSC		1 u	1 u	Operational:Enabled; Us...	
BSC11	■	BSC				Operational:Enabled; Us...	

Table component

The tree component

The tree component is based on the Swing tree component. It allows you to display data in a hierarchical representation. It features an efficient tiny look and feel, smart selection modes, sorting capabilities, as well as load-on-demand.



Sorting order: Ascending
Filter pattern:
Style sheet: Default
Selection mode: Highlight
Current selection: BTS111_BTS121

Tree component

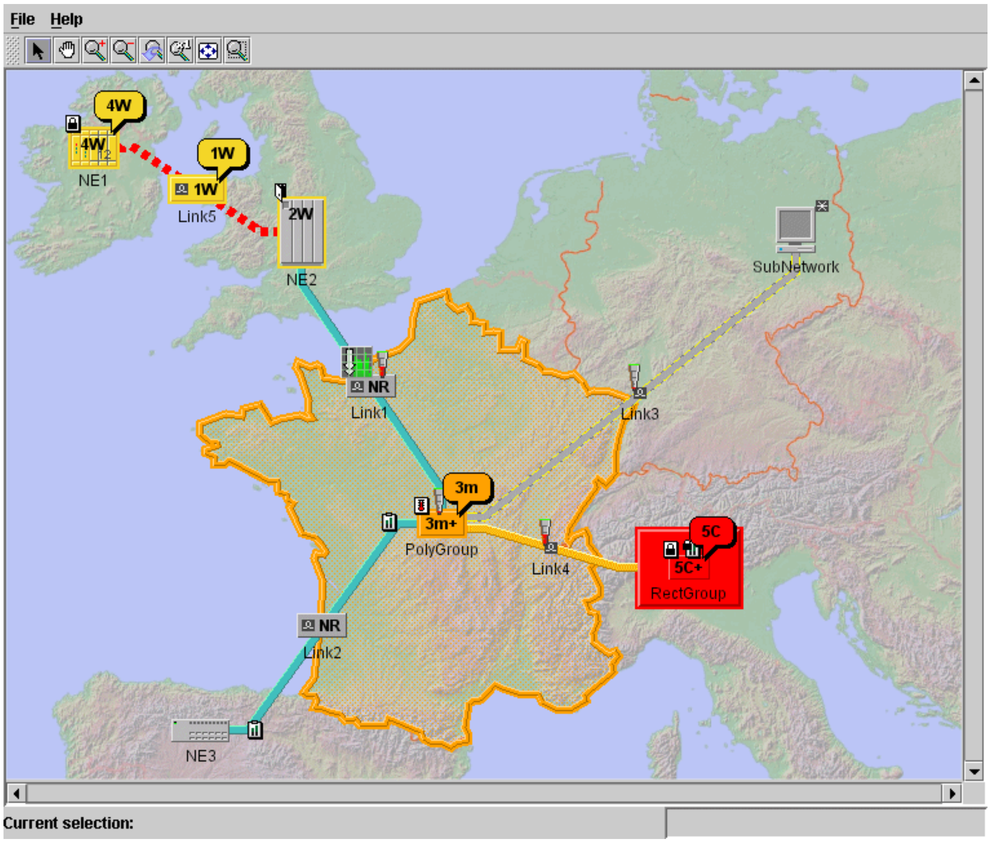
Network component

Describes the network component, which is one of the four graphic components supplied with IBM® ILOG® JViews TGO (JTGO). This component displays telecommunication networks in the form of topological or geographical views, depending on how you want to view them.

In this section

Introducing the network component

The network component is based on the Swing network component. It allows you to display data in the form of a graph representing nodes connected by links on top of a background map. The nodes may appear collapsed or expanded to reveal their child objects.



The network component supports editing, navigation, automatic layout of nodes and links, and background maps (for geographical displays). It can display all kinds of JViews TGO objects: network elements, links, groups, polylines, off-page connectors, cards, card carriers, shelves, ports, LEDs.

The network component is connected to a data source, from which it obtains the business objects to be displayed. By default, the network displays all the objects contained in the data source. However, it is also possible to restrict the contents displayed by:

- selecting the root nodes to be shown,
- applying a filter,
- specifying the business classes to be accepted or excluded by the component,

- specifying whether nodes are expandable or not (load on demand).

Objects that do not have a parent are displayed as root nodes, while the others are displayed under their parent.

The network component offers the following notable features:

- Filtering capabilities

  The network component allows you to filter the nodes that are displayed. That is, the business objects present in the attached data source are only displayed if they are accepted by the current filter.

- Interaction support

  You can interact with the network view as a whole as well as with individual objects.

- Load on demand

  The network component supports load on demand for the business objects to be displayed. This means that the graphic representation of a given business object is only created when its parent object is expanded through code or through user interaction. By default, load on demand is customized through the CSS property `expansion` (see Customizing the expansion of business objects). More advanced customization can be performed at the adapter level (see *Expansion strategy*).

- Layout capabilities

  You can perform node, link and label layouts.

- Zooming capabilities

  There are three zoom policies: physical, logical, and mixed.

- Background support

  The network component allows you to display maps in the background.

The network component is implemented by the class `IlpNetwork`, which is a Swing `JComponent` that can be directly inserted into a panel (`JPanel`).

`IlpNetwork` provides the API for the most common uses of the network component, such as:

- setting or retrieving the associated data source: `getDataSource()`, `setDataSource(ilog.cpl.datasource.IlpDataSource)`

- accessing and modifying the selection: `getSelectionModel()`, `setSelectionModel(ilog.cpl.network.IlpNetworkSelectionModel)`, `addSelectionObject(ilog.cpl.model.IlpObject)`, `removeSelectionObject(ilog.cpl.model.IlpObject)`, `clearSelection()`, `isObjectSelected(ilog.cpl.model.IlpObject)`, `getSelectedObject()`, `getSelectedObjects()`

- setting or retrieving the view interactor: `setViewInteractor(ilog.cpl.interactor.IlpViewInteractor)`, `getViewInteractor()`

- changing the root nodes of the network through the data source adapter: `getAdapter()`

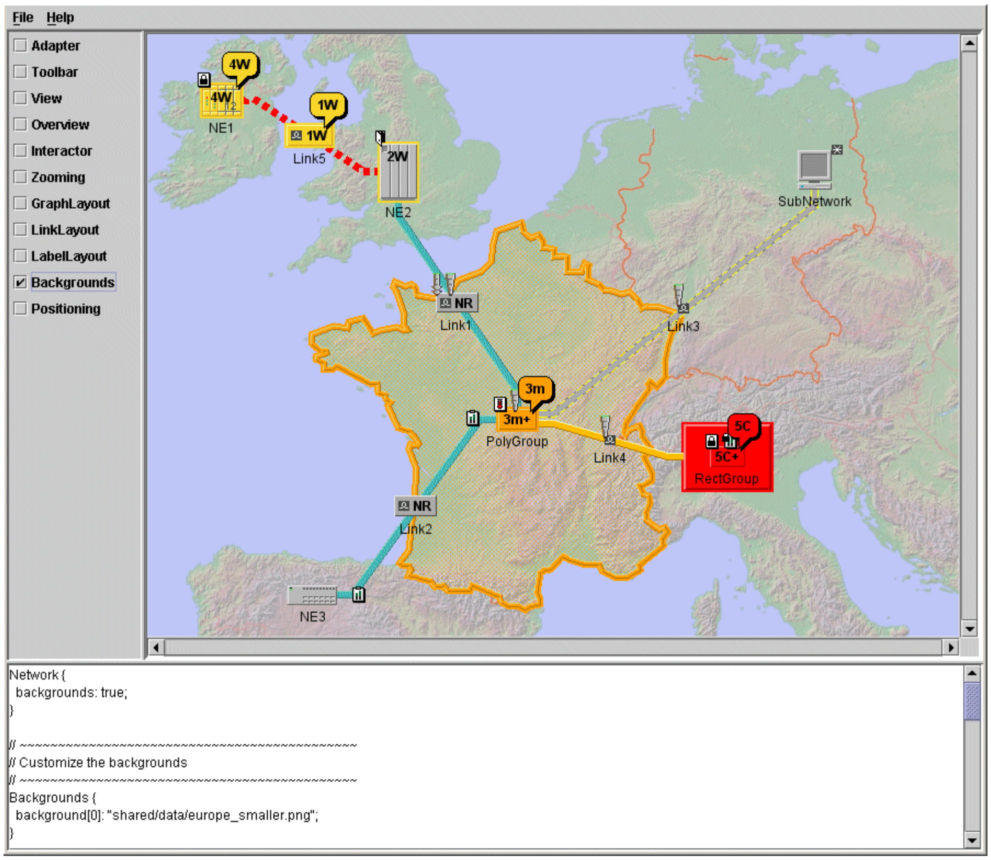
- filtering the network nodes: `setFilter(ilog.cpl.util.IlpFilter)`, `getFilter()`

`IlpNetwork` also acts as a façade for a number of lower-level components that it contains. These components provide more detailed APIs and advanced services. They are described in *Architecture of the network component*.

The information presented in this section is based on samples of typical network applications.

Creating a network component: a sample

The network component to be created is shown in the following figure.



```
Network {
  backgrounds: true;
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Customize the backgrounds
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Backgrounds {
  background[0]: "shared/data/europe_smaller.png";
}
```

A Styled Network

This example describes the steps for creating a frame as a network container, creating an instance of `IlpNetwork`, creating a data source and connecting it to the network, and finally reading in the network data.

How to create a basic network component

1. Create a frame to contain the network.

```
// Create a frame.
JFrame frame = new JFrame("ILOG JTGO network sample");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

2. Create the network component.

   ```
   IlpNetwork network = new IlpNetwork(networkDescriptionFileName,context);
   frame.add(network);
   ```

   You need to create a new instance of a network and make sure that the appropriate configuration is assigned. The network configuration is normally read in from a CSS file. You ensure that the network configuration file is taken into account and parsed by passing the URL and context or, as in the example, by passing the filename and context as arguments of `IlpNetwork`. You then add the network component to the frame.

   For more information on how to create the network configuration, see *Configuring a network component through a CSS file*

3. Create the data source and connect it to the network component.

   ```
   // Connect the data source to the network component.
   IltDefaultDataSource dataSource = new IltDefaultDataSource(context);
   network.setDataSource(dataSource);
   ```

   The data source holds the business objects that will be converted into representation objects by the network adapter through the data source API.

4. Read in an XML file, `network.xml`, that contains the network nodes and links.

   ```
   dataSource.parse("network.xml");
   ```

   The default data source creates the business objects by parsing an XML file or stream where the objects are described, as shown.

   You can create your own data sources from a file or database, or even create complex data sources based on proprietary object definitions.

How to add business objects to the data source for a network component through an XML file

The easiest way to populate a network is to read an XML data file into its data source.

The business object IDs must be unique within the given network.

The following XML code shows how to add a link.

```
<addObject id="1004035002697 60">
  <class>
    ilog.tgo.model.IltLink
  </class>
      ...
</addObject>
```

For information on how to define an XML file, refer to Defining the business model in XML
.

How to add business objects to the data source for a network component through the API

You can create business objects and insert them in the data source through the API, although this process is slower and less dynamic than reading an XML file. The following example shows how to insert an JViews TGO network element identified by its name, type and status into the data source as a node business object identified as `node1`.

```
// Put some objects into the datasource.
IlpObject node1 =
    new IltNetworkElement("washington",IltNetworkElement.Type.NE,
                          new IltObjectState());
dataSource.addObject(node1);
```

Configuring the network component

Identifies the rendering information necessary to display a network.

In this section

Introduction

To display a network, you need rendering information. This information defines how to display network data.

You can configure a network either through a CSS configuration file or through the API, the easiest and preferred way being the CSS configuration. You also have the possibility to load a project file which combines the CSS configuration and the network data.

A network configuration relates to one network only. It defines the behavior and properties of the network component as well as the behavior and some properties of any representation object created in the network model. (See *The model* for more information about features like representation objects and network model.)

You can customize the network adapter (filters, node and link factories, for example), the network view (toolbar, background, interactors, for example) and the network objects.

Configuring a network component through a CSS file

You can customize the following features in a CSS file:

- ♦ Network view
    - Toolbar visibility
    - Toolbar buttons
    - Overview window
    - View interactor
    - Zoom policy
    - Node layout
    - Link layout
    - Label layout
    - Background maps
    - Position converter
- ♦ Network Adapter
    - Expansion
    - Filtering
    - Origin
    - Node factory
    - Link factory
    - Accepted classes
    - Excluded classes

How to load a CSS file in a network component

The network configuration can be split across several CSS files. The method `setStyleSheets (int, java.lang.String)` accepts several CSS filenames.

There are three ways to apply a CSS configuration to a network component, depending on whether you have one or several configuration files:

- ♦ If you have a single configuration file and you do not want to inherit the settings from the default configuration file, pass the CSS configuration filename to the constructor of `IlpNetwork` as follows:

```
networkComponent = new IlpNetwork(myConfigurationFile);
```

If no CSS file is specified, the network component uses the default network configuration file, that is, `ilog.cpl.network.defaultConfiguration.css` from the `jviews-tgo-all.jar` file.

♦ If you have one or several configuration files to be applied, you can specify a project file that lists the style sheets and the data file to be loaded in the component (see *Loading a project file*). The project file will be as follows:

```
<?xml version="1.0"?>
<tgo xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/project.xsd"
style="configurationFile1.css,configurationFile2.css">
  <datasource javaClass="ilog.tgo.datasource.IltDefaultDataSource"
fileName="network.xml"/>
</tgo>
```

If the settings in two of the CSS files disagree, the effect depends on the order of the filenames in the list: the last file mentioned takes precedence over the first file.

♦ If you have several configuration files to be applied together, use the `setStyleSheets(int, java.lang.String)` method as follows:

```
networkComponent = new IlpNetwork();
networkComponent.setStyleSheets(
  new String[] { myConfigurationFile1, myConfigurationFile2 });
```

or, if you want to inherit and extend the settings from the default configuration file, use `setStyleSheets` as follows:

```
networkComponent = new IlpNetwork();
networkComponent.setStyleSheets(
  new String[] {
    IlpNetwork.DefaultConfigurationFileName,
    myExtraConfigurationFile
  });
```

If the settings in two of the CSS files disagree, the effect depends on the order of the filenames in the list: the last file mentioned takes precedence over the first file.

How to configure a network component in a CSS file

The following code represents an example of configuring a network in CSS. It is based on the CSS files located in ***<installdir>* /samples/network/styling** where `<installdir>` is the directory where you have installed JViews TGO.

The configuration in CSS is organized as a set of rules that define properties.

```
// **************************************************
// * COMPONENT CUSTOMIZATION                         *
// *                                                 *
// * This section is enclosed by                     *
// *    Network {                                    *
// *    }                                            *
// * and specifies which feature will be customized: *
// * - toolbar      (Network Component's toolbar)    *
// * - view         (Scrollbar configuration)        *
// * - overview     (Presence of an overview)        *
// * - interactor   (Default view interactor)        *
// * - zooming      (Zoom policy set in the view)    *
// * - graphlayout  (Layout for the nodes)           *
// * - linklayout   (Layout for the links)           *
// * - labellayout  (Layout for the labels)          *
// * - positioning  (Position policy for objects)    *
// * - backgrounds  (Backgrounds set in the view)    *
// * - adapter      (Network adapter customization)  *
// ***************************************************
Network {
    toolbar: true;
    interactor: true;
    zooming: true;
    adapter: true;
}
```

For detailed information about the CSS syntax, refer to Introducing cascading style sheets.

The Network rule

This rule specifies Boolean flags that indicate whether each customizable property is present. For example, customization of the property `GraphLayout` is not taken into account unless `graphlayout: true;` is declared in the Network rule.

This feature provides powerful cascading possibilities of CSS files. Thus, you can define `GraphLayout` customizations in a default CSS file and then turn them on or off in another CSS file.

The following properties are supported in the Network rule. You will find detailed documentation for each of these properties in the IBM® ILOG® JViews TGO *Java™ API Reference Documentation*, package `ilog.cpl.network.renderer`.

CSS properties of the network view

Property	Rule Type
`adapter`	Adapter
`toolbar`	ToolBar
`view`	View
`overview`	Overview
`interactor`	Interactor
`zooming`	Zooming
`graphLayout`	GraphLayout
`linkLayout`	LinkLayout
`labelLayout`	LabelLayout
`backgrounds`	Backgrounds
`positioning`	Positioning

The ToolBar rule

This rule controls the toolbar.

The property `enabled` is a Boolean property, with default value `true`. It controls whether the toolbar is visible or not.

The property `external` is a Boolean property, with default value `false`. It specifies whether the placement and visibility of the toolbar are managed by user code instead of internally by the network component.

Buttons can be added through the syntax `button[i]: @+ButtonId;` followed by the customization setting of the button with the given `ButtonId`.

A button has a mandatory property, `actionType`. This property specifies the action triggered by the button or a separator that is added to the toolbar. The value can be:

♦ a short name, such as `Select`, used for predefined actions, or

♦ the name of a subclass of `AbstractButton` with a constructor that takes an instance of `IlpViewsView` as argument, or

♦ the `Separator` short name to indicate that a separator should be placed in the specified position.

How to add a toolbar separator for the network component

You can add toolbar separators in specified positions of the network component toolbar. When configuring the network component toolbar, you can specify the position where a separator should be placed by using the predefined button action called `Separator`. This button action supports an optional property, `dimension`, which allows you to specify the dimensions of the separator in the toolbar.

The following example shows how to achieve this result:

```
ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
  button[1]: @+Separator;
  button[2]: @+PanButton;
}
Subobject#Separator {
  actionType: "Separator";
  dimension: "20,10";
}
```

The predefined values for the `actionType` property are the following:

Predefined values of the* `actionType` *property

`actionType` Values	Bean Class	Description
`ZoomIn`	`IlpNetworkZoomInButton`	Allows you to zoom in the view
`ZoomOut`	`IlpNetworkZoomOutButton`	Allows you to zoom out of the view
`ZoomBack`	`IlpNetworkZoomBackButton`	Allows you to go back to the previous zoom level
`ZoomReset`	`IlpNetworkZoomResetButton`	Allows you to reset the zoom level to the original level
`ZoomView`	`IlpNetworkZoomViewButton`	Allows you to specify a rectangular area on which to zoom
`FitToContents`	`IlpNetworkFitToContentsButton`	Allows you to fit the contents of the view to the size of the view
`ScrollToContents`	`IlpNetworkScrollToContentsButton`	Allows you to recenter the view
`Pan`	`IlpNetworkPanButton`	Allows you to pan the view
`Select`	`IlpNetworkSelectButton`	Allows you to select and move objects
`MakeLink`	`IlpNetworkMakeLinkButton`	Allows you to create links
`MakeLinearGroup`	`IlpNetworkMakeLinearGroupButton`	Allows you to create linear groups
`MakePolyGroup`	`IlpNetworkMakePolyGroupButton`	Allows you to create polygonal groups
`MakeRectGroup`	`IlpNetworkMakeRectGroupButton`	Allows you to create rectangular groups
`EditGroup`	`IlpNetworkEditGroupButton`	Allows you to edit the shape of groups
`EditLabel`	`IlpNetworkEditLabelButton`	Allows you to edit labels
`EditEquipmentObject`	`IlpNetworkEditEquipmentObjectButton`	Allows you to edit equipment objects
`LabelLayout`	`IlpNetworkLabelLayoutButton`	Allows you to trigger the label layout

A button can have a property `permanent`. This is a Boolean property, with default value `true`. For interactor buttons, this property denotes whether the interactor remains attached after it has performed its action.

A button can have a property `name`. This property specifies the name by which other elements in the file refer to the button. The default name is the short name used as `actionType`.

A button can have additional properties, corresponding to Bean properties of the Java™ class. For example, the `Select` button has the properties `multipleSelectionMode`, `moveAllowed`, `dragAllowed`, `editingAllowed`, `moveThreshold`, `opaqueMove`, `showingMovingObject`, `opaqueDragSelection`, `opaqueResize`, `opaquePolyPointsEdition`, `multipleSelectionModifier`, `selectionModifier`, which are documented in the class `ilog.cpl.network.action.toolbar.IlpNetworkSelectButton`.

An interactor button can have key or gesture actions attached to it. These actions are triggered by specific keystrokes or gestures while the interactor is active. They are added through the syntax `action[i]: @+ActionId;` followed by a customization setting for the action.

An action customization has the mandatory property `class`, which specifies the Java class of the `javax.swing.Action` object. Bean properties of this class are also customizable and the properties `key` and `gesture` can be set to specify when the action is to be executed. These two properties are not used in combination. For example, if you specify:

```
key: "control A";
gesture: "BUTTON1_CLICKED";
```

the action will be executed either when the key sequence ‘control-A’ is typed, or when the mouse `BUTTON1` is clicked. To define the property `gesture`, specify one of the predefined user gestures defined in class `IlpGesture`. To define the property `key`, specify a string that will be converted to a keystroke by the type converter (`IlpTypeConverter`).

The following predefined actions are available:

- `IlpSelectAllObjectsAction`
- `IlpRemoveSelectedObjectsAction`

An interactor can have a pop-up menu factory associated with it. This factory can be specified using the property `popupMenuFactory`. The value of this property should be a bean that implements the interface `IlpPopupMenuFactory`. For example,

```
Subobject#SelectButton {
    actionType: "Select";
    popupMenuFactory: @+popupMenuFactory;
  }

Subobject#popupMenuFactory {
    class: 'CustomPopupMenuFactory';
  }
```

In this example, the value of the property "`popupMenuFactory`" is a bean that is defined by the class `CustomPopupMenuFactory`. This class should implement the interface `IlpPopupMenuFactory`.

For more information on configuring the toolbar in a network view, refer to the class `IlpToolBarRenderer`.

How to add a predefined toolbar button to the network component

JViews TGO provides a list of predefined toolbar buttons usable in the network component (see *Predefined values of the `actionType` property*).

The following example shows how to add a predefined button that enables the `select` interactor. When this interactor is enabled, you can customize actions, pop-up menus and interactor properties as illustrated here:

```
ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
  action[0]: @+action0;
  popupMenuFactory: @+popupMenuFactory;
}

Subobject#popupMenuFactory {
  class: 'CustomPopupMenuFactory';
}

Subobject#action0 {
  key: "control A";
  class: 'ilog.cpl.graph.action.IlpSelectAllObjectsAction';
}
```

How to add a custom toolbar button to the network component

To add your own toolbar button, you need to create a new action class that inherits from `IlpNetworkInteractorAction` and contains a constructor that takes an `IlpViewsView` as parameter.

```
public class CustomButtonAction extends IlpNetworkInteractorAction {

  public CustomButtonAction(IlpViewsView view) {
     super(view);
     // Do any needed initialization
     // Define your own view interactor that will be active when the button
is
selected
     // in the toolbar
     IlpViewsViewInteractor interactor =  new IlpViewsViewInteractor();
     // Register the interactor in this action
     setIlpInteractor(interactor);
  }
}
```

Then, you need to register this new button in your component configuration, as follows:

```
ToolBar {
  enabled: true;
  button[0]: @+MyButton;
}

Subobject#MyButton {
  actionType: 'CustomButtonAction';
  toolTipText: "Custom";
  icon: @+customIcon;
}
Subobject#customIcon {
  class: 'javax.swing.ImageIcon';
  image: '@|image("custom.png")';
}
```

The custom action will be encapsulated in an `IlpNetworkInteractorButton`, and you will be able to customize the properties of this button as with the predefined buttons. For example, you can customize the following:

- `name`
- `usingObjectInteractor`
- `popupMenuFactory`
- actions associated with gestures and keystrokes

The View rule

This rule controls the view.

You can customize the following properties of the view:

View properties

Property Name	Type	Default Value	Description
`horizontalScrollBarPolicy`	`int`	`IlvJScrollManagerView.HORIZONTAL_SCROLLBAR_AS_NEEDED`	Defines the policy for the visibility of the horizontal scrollbar
`verticalScrollBarPolicy`	`int`	`IlvJScrollManagerView.VERTICAL_SCROLLBAR_AS_NEEDED`	Defines the policy for the visibility of the vertical scrollbar
`keepingAspectRatio`	`boolean`	`false`	Defines whether the view keeps the aspect ratio when zooming
`minZoomXFactor`	`double`	`0`	Specifies the minimum zoom factor

Property Name	Type	Default Value	Description
			allowed on the X (horizontal) axis of the view
`maxZoomXFactor`	`double`	`Double.MAX_VALUE`	Specifies the maximum zoom factor allowed on the X (horizontal) axis of the view
`minZoomYFactor`	`double`	`0`	Specifies the minimum zoom factor allowed on the Y (vertical) axis of the view
`maxZoomYFactor`	`double`	`Double.MAX_VALUE`	Specifies the maximum zoom factor allowed on the Y (vertical) axis of the view
wheelZoomingEnabled	`boolean`	`true`	Defines whether the view zooms in response to moving the mouse wheel while pressing the Control key
wheelScrollingEnabled	`boolean`	`true`	Defines whether the view scrolls in response to moving the mouse wheel

The following CSS sample shows how to customize the view:

```
View {
  horizontalScrollBarPolicy: AsNeeded;
  verticalScrollBarPolicy: Never;
  keepingAspectRatio: true;
}
```

For more information on configuring a network view, refer to the class `IlpViewRenderer`.

The Overview rule

This rule controls the overview window.

The property `enabled` controls the visibility of the overview window. The default value is `false`.

The following CSS sample shows how to customize the overview:

```
Overview {
  enabled: true;
}
```

For more information on configuring the overview window in a network view, refer to the class `IlpOverviewRenderer`.

The Interactor rule

This rule controls the interactor associated with the view.

You can customize the following properties of the interactor:

Interactor properties

Property Name	Type	Default Value	Description
`name`	`String`	none	Specifies the name of a toolbar button that activates an interactor. This button is activated at startup. Its interactor becomes the initial view interactor, as well as the default view interactor when another interactor stops its interaction. This property is only considered when the view has a toolbar configured and enabled.
`viewInteractor`	`IlpViewsViewInteractor`	none	Specifies the interactor instance that becomes the initial view interactor, and the default view interactor when another interactor stops its interaction.

How to configure a network interactor in a CSS file

Prior to configuring the network view interactor, you need to configure the network component so that the interactor configuration is enabled:

```
Network {
  interactor: true;
}
```

After that, you can customize the interactor property in the Interactor rule as illustrated by the next code extract. for details about the CSS syntax, refer to The CSS specification.

How to set the default view interactor from the toolbar of the network component

You can customize the default view interactor to be one of the interactors present in the toolbar configured for the network component. In this case, the toolbar button is identified when the network component is configured, and it is activated at startup. This interactor becomes the initial view interactor, and the default view interactor when another interactor stops its interaction. This configuration is achieved through the property `name`, whose value must be the name of one of the configured toolbar buttons.

The following CSS extract configures the network view to use the `Select` toolbar button as the default view interactor:

```
Interactor {
  name: "Select";
}
```

How to set the default view Interactor when the toolbar is disabled in the network view

When the toolbar is disabled, you can no longer specify the default view interactor by using a toolbar button name, as in the example above. However, you can specify the view interactor directly in CSS, as follows:

```
Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.graphic.views.IlpViewsViewInteractor';
}
```

The behavior of the view interactor is determined by the actions that are associated with user gestures and keystrokes. This behavior can also be customized through CSS. You can also configure a pop-up menu to be displayed in the network view. For more information about interactor customization, refer to *Interacting with the network view* and *Interacting with the network objects*.

When the interactor renderer is enabled, you can also customize objects interactors using property `interactor`. For further information, refer to `IlpInteractorRenderer` .

The Zooming rule

This rule controls the zoom policy.

The mandatory property `type` specifies the type of zoom. The possible values are `Logical`, `Physical`, or `Mixed`. Each zoom policy may have additional properties that you can also set using CSS:

♦ Logical zoom policy (see `IltLogicalZoomPolicy`)

Logical zoom property

Property Name	Type	Default	Description
`additionalZoom`	`double`	`1`	Specifies an additional zoom factor that is implicitly added to the zoom transformer of the view. This property is useful when printing with non standard transformers.

♦ Physical zoom policy (see `IltPhysicalZoomPolicy`)

Physical zoom properties

Property Name	Type	Default	Description
`decorationNames`	`String[]`	`null`	Specifies a list of decoration names that are customized at the zoom policy level. See `IltGraphicElementName` for a list of decoration names that can be used.
visibilityThresholds	double[]	null	Specifies a list of thresholds, one for each decoration name customized with property `decorationNames`. These thresholds indicate the zoom level below which the decorations become invisible in the view. It allows you to hide decorations as the user zooms out in the view.

♦ Mixed zoom policy (see `IltMixedZoomPolicy`)

Mixed zoom properties

Property Name	Type	Default	Description
zoomThreshold	double	1	Specifies the zoom threshold when the physical zoom or the logical zoom should be used
subnetworkZoomFactor	double	1	Specifies an additional zoom threshold that is applied to expanded subnetworks
`decorationNames`	`String[]`	`null`	Specifies a list of decoration names that are customized at the zoom policy level. See `IltGraphicElementName` for a list of decoration names that can be used.
visibilityThresholds	double[]	null	Specifies a list of thresholds, one for each decoration name customized with property `decorationNames`. These thresholds indicate the zoom level below which the decorations become

Property Name	Type	Default	Description
			invisible in the view. It allows you to hide decorations as the user zooms out in the view.

How to customize the zoom policy in a network component

The following CSS sample shows how to customize the zoom policy in a network view:

```
Zooming {
  type: "Mixed";
  zoomThreshold: 1.0;
  subNetworkZoomFactor: 1.0;
}
```

For more information on configuring the zoom behavior in a network view, refer to the class `IlpZoomingRenderer`.

How to configure the visibility of decorations for a specific network component

JViews TGO provides three predefined zoom policies that you can use directly in the network component. For more information, see *Zooming*. When using the physical or the mixed zoom policy, decorations may become invisible as the user zooms out in the view. By default, this configuration is global in the application. However, you may need to specify the visibility threshold for a specific view. This feature is supported by properties `decorationNames` and `visibilityThresholds`. Property `decorationNames` specifies each decoration that is configured for the view (see `IltGraphicElementName` for the list of decoration names that can be customized). Property `visibilityThresholds` specifies, for each decoration name, the visibility threshold below which the decoration becomes invisible. This configuration is illustrated by the following example:

```
Zooming {
  type: "Mixed";
  decorationNames[0]: Name;
  decorationNames[1]: AlarmBalloon;
  decorationNames[2]: AlarmCount;
  decorationNames[3]: Plinth;
  visibilityThresholds[0]: 0.5;
  visibilityThresholds[1]: 0.8;
  visibilityThresholds[2]: 0.5;
  visibilityThresholds[3]: 0.5;
}
```

The GraphLayout rule

This rule allows you to control the automatic node layout in the view and to configure nonautomatic node layouts. Nonautomatic node layouts can only be executed through the API. For more details, see *Layout*.

How to control the automatic node layout in the network view

Automatic node layout is configured through the property `class`. This property specifies the graph layout class, a subclass of `IlvGraphLayout`. Additional Bean properties can be specified, depending on the class.

The following CSS sample shows how to customize the graph layout:

```
GraphLayout {
    class:
     'ilog.views.graphlayout.uniformlengthedges.IlvUniformLengthEdgesLayout';

    respectNodeSizes: true;
    preferredLinksLength: 200;
    forceFitToLayoutRegion: true;
    layoutRegion: "50, 50, 700, 450";
}
```

The graph layout is always a subclass of `IlvGraphLayout` which supports the "`preserveFixedNodes`" property. This property allows you to switch the support of fixed nodes on or off. You can set a node as fixed in the business object customization.

Note: If a graph layout is set and supports fixed nodes, a link layout is required when links are connected to the fixed nodes.

The properties of each layout algorithm are fully explained in the *IBM® ILOG®* JViews Diagrammer *Using Graph Layout Algorithms* documentation.

The properties of the `IlvGraphLayout` subclasses conform to the following JavaBeans™ convention: if a class has a pair of methods called `setMyProp` (with a single parameter) and `getMyProp` (without parameters), then you can set the property `myProp` in the style sheet.

Note: If the value of the property is an enumeration of integer values defined by static member variables of the class, then you can use the name of the variable alone, or the variable name prefixed by the class name alone, or the variable name prefixed by the fully qualified class name. For example, the following declarations are all valid:

```
globalLinkStyle: "ORTHOGONAL_LINKS";

globalLinkStyle: "IlvHierarchicalLayout.ORTHOGONAL_LINKS";

globalLinkStyle: "ilog.views.graphlayout.hierarchical.
IlvHierarchicalLayout.ORTHOGONAL_LINKS";
```

How to configure multiple node layouts in a network view

The Graph Layout rule allows you to configure multiple node layouts, and to define the node layout to be used automatically in the view. Multiple node layout configuration is achieved through the properties `layouts` and `autoLayoutIndex`, as illustrated below:

```
GraphLayout {
  layouts[0]: @+treeLayout;
  layouts[1]: @+hierarchicalLayout;
  layouts[2]: @+springEmbedderLayout;
  autoLayoutIndex: 1;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
  flowDirection: Bottom;
}

Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Bottom;
}

Subobject#springEmbedderLayout {
  class: 'ilog.views.graphlayout.springembedder.IlvSpringEmbedderLayout';
  respectNodeSizes: true;
  preferredLinksLength: 200;
  forceFitToLayoutRegion: true;
  layoutRegion: "50, 50, 700, 450";
}
```

In this use case, three node layouts are configured for the view: `IlvTreeLayout`, `IlvHierarchicalLayout` and `IlvSpringEmbedderLayout`. The hierarchical layout is configured to be performed automatically when the contents of the view changes. This configuration is achieved by specifying the value of property `autoLayoutIndex` as the index of the hierarchical layout defined through the `layouts` property. The other node layouts can be performed on demand using the API (see `IlpGraphView.performAttachedLayout`).

How to configure nonautomatic node layouts in the network component

If you are not interested in automatic node layout, you can still configure multiple node layouts in CSS. To have only nonautomatic node layouts, set property `autoLayoutIndex` to `-1`, as illustrated below:

```
GraphLayout {
  layouts[0]: @+treeLayout;
  layouts[1]: @+hierarchicalLayout;
  layouts[2]: @+springEmbedderLayout;
  autoLayoutIndex: -1;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
  flowDirection: Bottom;
}

Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Bottom;
}

Subobject#springEmbedderLayout {
  class: 'ilog.views.graphlayout.springembedder.IlvSpringEmbedderLayout';
  respectNodeSizes: true;
  preferredLinksLength: 200;
  forceFitToLayoutRegion: true;
  layoutRegion: "50, 50, 700, 450";
}
```

How to specify a different node layout for each subnetwork

When the node layout is configured for a view, it is applied to the view and to all the subnetworks displayed in this network view.

However, you can specify different node layouts for subnetworks. The node layouts of subnetworks are configured in CSS using the same properties as for the view configuration (`layouts` and `autoLayoutIndex`).

When you configure the node layout for a specific subnetwork, you must declare the CSS selector with a specific pseudoclass that identifies the graph layout renderer. The pseudoclass must be `graphLayoutRenderer`, as illustrated below:

```
GraphLayout {
  layouts[0]: @+treeLayout;
  autoLayoutIndex: 0;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
  flowDirection: Bottom;
}

#SubNetwork:graphLayoutRenderer {
  layouts[0]: @+hierarchicalLayout;
  autoLayoutIndex: 0;
}

Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Top;
}
```

The property `autoLayoutIndex`, when used in an object selector with the `graphLayoutRenderer` pseudoclass, specifies the automatic node layout that is applied to the subnetwork.

You can also specify nonautomatic node layouts for subnetworks. This configuration is achieved in the same way as for the view configuration (`layouts` property with multiple node layouts and the `autoLayoutIndex` set to `-1`).

When you call the method `performAttachedLayout(int index)`, it is applied recursively in the model hierarchy. Therefore if a node layout is configured for the given index in the subnetwork, this specific layout is performed. Otherwise, the node layout configured for the parent network is executed.

How to set node or link parameters on graph layout objects in the network component

You can set parameters for a graph layout algorithm that applies to a particular node or link, in the style sheet. Such parameters are defined by a method of the form:

```
setMyParam(Object node,  value);
```

or

```
setMyParam(Object link,  value);
```

Node parameters are set in the style sheet as follows:

```
object."ilog.tgo.model.IltObject":graphLayoutRenderer {
  myParam: "value";
}
```

The name of the property is the name of the method, without the prefix `set`. The pseudoclass `graphLayoutRenderer` indicates that the declarations apply to the node layouts that are configured in the graph layout rule.

For example, the graph layout defines a `setFixed` method that lets you specify whether a node or link is fixed. Fixed nodes or links are not moved when the layout is applied. The signature of the method is:

```
setFixed(Object nodeOrLink, boolean fixed);
```

In the style sheet, you can set this parameter as follows:

```
object."ilog.tgo.model.IltObject":graphLayoutRenderer {
  fixed: true;
}
```

The value of the property can be any basic type (integer, String, float), or it can be the name of a public constant defined by the graph layout class, for example, `WEST` , which is defined in the class `IlvHierarchicalLayout`.

When the graph layout rule contains multiple node layouts, you can still specify node and link layout parameters by using pseudoclasses that identify the graph layout to which the declarations apply.

```
GraphLayout {
  layouts[0]: @+treeLayout;
  layouts[1]: @+hierarchical;
  autoLayoutIndex: 0;
}
```

```
Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
  flowDirection: Bottom;
}

Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Top;
}

#NE1:graphLayoutRenderer:tree {
  root: true;
}

#NE1:graphLayoutRenderer:hierarchical {
  specNodeLevelIndex: 0;
}

#NE2:graphLayoutRenderer:hierarchical {
  specNodeLevelIndex: 1;
}
```

In the example above, `NE1` is configured as the root object for the Tree Layout algorithm. This is achieved by declaring the property `root` in a selector that contains the pseudoclasses `graphLayoutRenderer` (indicates that this is a graph layout renderer per-object property) and `tree` (indicates that this is a property specific to the `IlvTreeLayout` algorithm).

At the same time, `NE1` is configured to be placed at level 0 in case of a hierarchical layout. This is achieved using pseudoclasses `graphLayoutRenderer` and `hierarchical`(indicates that this is a graph layout per-object property specific to the `IlvHierarchicalLayout` algorithm).

Each layout algorithm supports a set of per-object parameters. For more information on the parameters supported by each layout algorithm, refer to package `ilog.cpl.graph.css.renderer.graphlayout`.

In addition to the properties that are specific to the layout algorithms, the graph layout renderer also supports the following properties:

♦ `layoutIgnored`: If this property is set to `true`, the object is completely ignored by the graph model (using an `IlvLayoutGraphicFilter`).

♦ `markedForIncremental`: If the layout algorithm is an `IlvHierarchicalLayout`, you can use the property `markedForIncremental`. When this property is set to `true` for an object, the method `IlvHierarchicalLayout.markForIncremental(java.lang.Object)` is called for this object. This means that the position of the object is recomputed during the next incremental layout. This property has an effect only if the `incrementalMode` property of the layout itself is set to `true`. For example:

```
GraphLayout {
  layouts[0]: @+hierarchicalLayout;
}
```

```
Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  incrementalMode: true;
}

#NE1:graphLayoutRenderer:hierarchical {
  markedForIncremental : "true";
}
```

Important: Starting with JViews TGO 7.5, if you are not using any node or link parameters, you can disable this mechanism by specifying `IlpGraphLayoutRenderer.setUsePerObjectParameters(false)`. This will remove the overhead of testing the parameters and speed up the rendering process significantly.

How to disable the per-object layout parameters for node configuration in the network view

You can also disable the per-object layout parameters configuration through CSS as follows:

```
GraphLayout {
  layouts[0]: @+hierarchicalLayout;
  usePerObjectParameters: false;
}
```

For more information on configuring the node layout in a network view, refer to the class `IlpGraphLayoutRenderer`.

The LinkLayout rule

This rule controls the automatic link layout in the view.

The mandatory property `class` specifies the link layout class, a subclass of `IlvGraphLayout`. Additional Bean properties can be specified, depending on the class.

How to control the automatic link layout in the network view

The following CSS sample shows how to customize the link layout:

```
LinkLayout {
    class: "ilog.views.graphlayout.link.IlvLinkLayout";
    globalLinkStyle: MIXED_STYLE;
}
```

The link layout is always a subclass of `IlvGraphLayout` which supports the "`preserveFixedLinks`" property. This property allows you to switch the support of fixed links on or off.

How to specify a different link layout for each subnetwork

When the link layout is configured for a view, it is applied to the view and to all the subnetworks displayed in this network view.

However, you can also specify different link layouts for subnetworks. The link layout is configured in CSS using the property `linkLayout` and the pseudoclass `linkLayoutRenderer`, as illustrated below:

```
LinkLayout {
  class:'ilog.views.graphlayout.link.IlvLinkLayout';
}

#SubNetwork:linkLayoutRenderer {
  linkLayout: @+shortLinkLayout;
}

Subobject#shortLinkLayout{
  class: 'ilog.tgo.graphic.graphlayout.IltShortLinkLayout';
}
```

Like the graph layout renderer, the link layout renderer supports setting per-object link layout parameters through CSS. See *How to set node or link parameters on graph layout objects in the network component*.

How to specify per-object parameters for link layouts in the network view

Link parameters are set in the style sheet as follows:

```
object."ilog.tgo.model.IltAbstractLink":linkLayoutRenderer {
  linkStyle: ORTHOGONAL_STYLE;
}

#Link1:linkLayoutRenderer {
  linkStyle: DIRECT_STYLE;
}
```

Important: Starting with JViews TGO 7.5, if you are not using any node or link parameters, you can disable this mechanism by specifying `IlpLinkLayoutRenderer.setUsePerObjectParameters(false)`. This will remove the overhead of testing the parameters and speed up the rendering process significantly.

How to disable per-object layout parameters for link configuration in the network view

You can also disable the per-object layout parameters configuration through CSS as follows:

```
LinkLayout {
  class: 'ilog.views.graphlayout.link.IlvLinkLayout';
  usePerObjectParameters: false;
}
```

For more information on configuring the link layout in a network view, refer to the class `IlpLinkLayoutRenderer`.

The LabelLayout rule

This rule controls the automatic label layout in the view.

The mandatory property `class` specifies the label layout class, a subclass of `IlvLabelLayout`. Additional Bean properties can be specified, depending on the class. The usual setting is `IltAnnealingLabelLayout`.

The following CSS sample shows how to customize the label layout:

```
LabelLayout {
   class: 'ilog.tgo.graphic.graphlayout.labellayout.IltAnnealingLabelLayout';

    obstacleOffset: 10;
    labelOffset: 15;
}
```

Note that some properties can be set within the `IltAnnealingLabelLayout` and some within the `IlvAnnealingLabelLayout`. For more details, refer to the IBM® ILOG® JViews TGO *Java API Reference Documentation* .

For more information on configuring the label layout in a network view, refer to the class `IlpLabelLayoutRenderer`.

The Backgrounds rule

This rule allows you to configure two kinds of background that affect the network component representation:

♦ network background

♦ manager view background

Network backgrounds

This refers to background files such as maps or background images. You can specify a network background through:

1. A URL

The background file is specified directly by its URL:

```
Backgrounds {
  background[i]: "URL";
}
```

For example:

```
Backgrounds {
  background[0]: "backgrounds/sf-bayarea.png";
}
```

2. A CSS bean

   The background URL and its properties are specified through a CSS bean:

```
Backgrounds {
  background[i]: @+background0;
}

Subobject#background0 {
  class: "ilog.cpl.graph.background.css.IlpBackgroundCSSConfiguration";

  PROPERTY :  PROPERTY_VALUE;
  ...
}
```

The bean `IlpBackgroundCSSConfiguration` encapsulates all the properties supported by the predefined background types. You should use it if want to use one of the predefined types.

For example:

```
Backgrounds {
  background[0]: @+background0;
}

Subobject#background0 {
  class: "ilog.cpl.graph.background.css.IlpBackgroundCSSConfiguration";

  //////////////////////
  //Background properties
  //////////////////////
  url : "backgrounds/sf-bayarea.png";
  loadOnDemand : "true";
  threaded : "false";
}
```

If you intend to use a custom background type that has additional properties, you can either subclass the default `IlpBackgroundCSSConfiguration` and use it with the additional bean properties. Or you can provide a bean that contains all the required properties to configure the `IlpBackground` implementation. All bean properties are automatically communicated and stored in the `IlpBackground` implementation through its `IlpBackground.setProperty` interface method.

For details on the properties available and supported for each `IlpBackground` implementation, see *Background support*.

You can mix and match options 1 and 2 by specifying some backgrounds as beans and some as straight URL strings.

In the case of the Image Tile background type (`IlpImageTileBackground`), only the `url` property can be configured through CSS. For the other properties, use the XML configuration. See *Background support*.

Manager view background

This refers to the representation of the view as a background for your network backgrounds (the area that the network backgrounds do not cover).

You can configure the manager view background as follows:

```
Backgrounds {
  PROPERTY : PROPERTY_VALUE;
}
```

For example:

```
Backgrounds {
  backgroundColor : "white";
}
```

The following table lists the properties that allow you to customize the manager view background:

Properties of the manager view background

Property Name	Type	Default	Sample	Description
backgroundColor	Color	null	backgroundColor:"black";	Specifies the color to be used to fill the background of the view.
backgroundPattern	String	null	backgroundPattern:"pattern.png";	Specifies the location of the pattern image to be used to fill the background of the view.

For more information on configuring the background in a network, refer to the class `IlpBackgroundsRenderer`.

The Positioning rule

This rule controls the type and converter of the user-defined `IlpPosition`.

The property `positionClass` denotes the Java class name of the class or interface that implements `IlpPosition`.

The property converterClass denotes a Java class name or CSS bean that implements the IlpPositionConverter interface and determines the conversion between business data coordinates and (x,y) coordinates in the view.

The following CSS sample shows how to customize the positioning:

```
Positioning {
  positionClass: 'my.package.MyPosition';
  converterClass: 'my.package.MyPositionConverter';
}
```

For more information on configuring the positioning in a network view, refer to the class IlpPositioningRenderer.

The Adapter rule

This rule controls the configuration of the network adapter. The network adapter is responsible for converting the business objects in the data source to representation objects (network nodes) in the network component. It provides the following features:

♦ **Filtering**: applies a filter so that business objects currently in the data source are not mapped to representation objects in the network component.

♦ **Origins**: defines which objects become root nodes in the network.

♦ **Link factory**: defines how a link representation object will be created from its business object counterpart.

♦ **Node factory**: defines how a representation object that is not a link will be created from its business object counterpart.

♦ **Expansion strategy**: defines how the objects will be loaded in the network component, that is, either at initialization time or on demand, as the user interacts with the network nodes.

♦ **Accepted classes**: defines the list of business classes that are accepted by the network adapter. Only the business objects that match one of these business classes will be mapped to representation objects by the network adapter.

♦ **Excluded classes**: defines the list of business classes that are excluded by the network adapter. The business objects of these business classes will not be mapped to representation objects by the network adapter. By default, the IltAlarm business class is part of the list of excluded classes.

These network adapter features can be customized through CSS using the following properties:

CSS properties of the network adapter

Property Name	Property Type
`filter`	`IlpFilter`
`origins`	list of object identifiers
`networkNodeFactory`	`IlpNetworkNodeFactory`
`networkLinkFactory`	`IlpNetworkLinkFactory`
`expansionStrategyFactory`	`IlpExpansionStrategy`
acceptedClasses	list of `IlpClass`
excludedClasses	list of `IlpClass`

How to configure a network adapter in a CSS file

Prior to configuring the adapter, you need to configure the network component so that the adapter configuration is enabled:

```
Network {
  adapter: true;
}
```

After that, you can customize each adapter property in the Adapter rule as illustrated by the following code extract. Refer to The CSS specification in the *Styling* documentation for details about the CSS syntax.

```
Adapter {
  filter: @+Filter;
}

Subobject#Filter {
  class: 'CustomFilter';
  rejectObject[0]: "NE1";
  rejectObject[1]: "Link5";
}
```

How to programmatically configure adapter using CSS

You can programmatically modify the CSS configuration of the default network adapter (`IlpNetworkAdapter`) by using mutable style sheets through the `IlpMutableStyleSheet` API.

Important: The mutable style sheet is set to the adapter as a regular style sheet and is cascaded in the order in which it has been declared.

To use mutable style sheets:

1. Get the mutable style sheet.

You access the mutable style sheet through the `getMutableStyleSheet()` method in the network adapter API:

```
IlpMutableStyleSheet mutable = adapter.getMutableStyleSheet();
```

This method automatically registers the mutable style sheet into the adapter. You can manually instantiate an object of the class `IlpMutableStyleSheet` and register it yourself through the `setStyleSheet()` API:

```
IlpMutableStyleSheet mutable = new IlpMutableStyleSheet(adapter);
try {
  adapter.setStyleSheets(new String[] { mutable.toString() });
} catch (Exception x) {
  x.printStackTrace();
}
```

2. Set the CSS declarations.

   Once you have the mutable style sheet, you can set the declarations you want:

```
mutable.setDeclaration("#myObjectId", "expansion", "NO_EXPANSION");
```

   This creates the following CSS declaration into the mutable style sheet:

```
#myObjectId {
  expansion: NO_EXPANSION;
}
```

3. Register the mutable style sheet.

   The mutable style sheet should be set to the adapter as a regular style sheet using the `setStyleSheet()` method:

```
try {
  adapter.setStyleSheets(new String[] { mutable.toString() });
} catch (Exception x) {
  x.printStackTrace();
}
```

4. Set and update the CSS declarations.

   The mutable style sheet can be modified even after being registered to the adapter:

```
// Update the expansion type for 'myObjectId'
mutable.setDeclaration("#myObjectId", "expansion", "IN_PLACE");
// Add a new declaration
mutable.setDeclaration("#myOtherId", "expansion", "IN_PLACE");
```

Note: Like any style sheet, the mutable style sheet is lost when the `setStyleSheet()` API is invoked and a new set of style sheets is applied to the adapter.

How to customize the mutable style sheet

Reapplying a CSS configuration may be a heavy task, as the adapter may be forced to review filters, origins, recreate representation objects, and so on. It is important to use the mutable style sheet with care and to customize it properly to reapply the CSS wisely. To do so, there are two methods available in the `IlpMutableStyleSheet` API: `setUpdateMask()` and `setAdjusting()`.

1. `setUpdateMask()`

   This method controls what should be recustomized once a declaration of the mutable style sheet has been updated. The CSS configuration of the adapter is divided into two parts: *adapter customization* and *representation object* customization.

   The adapter customization handles the origins, filters, and so on:

```
Adapter {
  origins[0]: id0;
  origins[1]: id1;
  showOrigin: true;
  filter: @+myFilter;
}
```

   The representation object customization handles the expansion type of a representation object:

```
#myObjectId {
  expansion: IN_PLACE;
}
```

   The accepted values for `setUpdateMask()` are:

   ♦ `IlpStylable.UPDATE_COMPONENT_MASK`: Only the adapter part is recustomized.

   ♦ `IlpStylable.UPDATE_OBJECTS_MASK`: Only the representation object part is recustomized.

   ♦ `IlpStylable.UPDATE_ALL_MASK`: Bot the adapter and representation object parts are recustomized.

   ♦ `IlpStylable.UPDATE_NONE_MASK`: Nothing is recustomized.

   For example, if you update the expansion type of a representation object through the mutable style sheet, it is recommended that you set the update mask to `UPDATE_OBJECTS_MASK` as there is no need to reapply the CSS configuration for the adapter part:

```
mutable.setUpdateMask(IlpStylable.UPDATE_OBJECTS_MASK);
mutable.setDeclaration("object", "expansion", "IN_PLACE");
```

2. `setAdjusting()`

   This method is used when a series of declarations must be applied to the mutable style sheet. When the method is set to `true`, the mutable style sheet puts all the calls to `setDeclaration()` into a queue. When the method is set back to `false`, all the queued declarations are processed in a batch:

```
mutable.setAdjusting(true);
mutable.setDeclaration("#myObjectId", "expansion", "IN_PLACE");
mutable.setDeclaration("#myOtherId", "expansion", "IN_PLACE");
mutable.setAdjusting(false);
```

Configuring a network component through the API

For details of the classes involved in the architecture of the network component, see *Architecture of the network component*.

The following example shows how to configure the network view through the API. For details o programming the individual services, see *Network component services*.

How to configure the network view with the API

```
IlpNetworkView view = network.getView();

 //Toolbar
 view.getToolBar().add(new IlpNetworkSelectButton(view));
 view.getToolBar().add(new IlpNetworkZoomResetButton(view));
 // Overview
 view.setOverviewVisible(true);
 // View interactor
 IltSelectInteractor selInteractor = new IltSelectInteractor();
 selInteractor.setEditingAllowed(true);
 IlpViewsViewInteractor viewsInteractor =
    new IlpViewsViewInteractor(selInteractor);
 view.getController().setViewInteractor(viewsInteractor);
 // Zoom policy
 view.setZoomPolicy(
   new IltMixedZoomPolicy() {{
     setZoomThreshold(2.0);
   }});
 // Layout
 view.setNodeLayout(new IlvGridLayout());
 view.setLinkLayout(new IltShortLinkLayout());
 // Background
 view.addBackgroundURL(
   context.getURLAccessService().getFileLocation(
     "data/images/europe.jpg"));
 // LayerPolicy
 view.getCompositeGrapher().setLayerPolicy(myLayerPolicy)
 // Position
 view.setPositionConverter(
   new IlpGeographicPositionConverter(projection,true));
```

How to configure the network adapter with the API

The following example shows how to configure the network adapter through the API. For details on programming the individual services, see *Network component services*.

```
IlpNetworkAdapter adapter = network.getAdapter();

// Filter
```

```
IlpFilter myFilter = new MyFilter();
// (it is the same as network.setFilter(myFilter);)
adapter.setFilter(myFilter);

// Origin
List myOrigins = new ArrayList();
myOrigins.add(objectID_1);
myOrigins.add(objectID_2);
:
:
myOrigins.add(objectID_n);
// in this case we want to display the origins
boolean showOrigin = true;
adapter.setOrigins(myOrigins, showOrigin);

// Expansion Strategy Factory
// Usually the expansion strategy factory relies
// on the adapter to access the data source and
// to load/release objects
IlpExpansionStrategyFactory myExpFactory = new
  MyExpansionStrategyFactory(adapter);
adapter.setExpansionStrategyFactory(myExpFactory);

// Position Attribute
// Here, imagine that MyObject implements IlpObject
// interface and defines "Placement" as the
// IlpAttribute that defines the object position
IlpClass myObjectClass = MyObject.getIlpClass();
IlpAttribute myPosAttrib = MyObject.Placement;
adapter.setPositionAttribute(myObjectClass, myPosAttrib);

// Node and Link Factory
IlpNetworkNodeFactory myNodeFactory = new MyNodeFactory();
adapter.setNodeFactory(myNodeFactory);

IlpNetworkLinkFactory myLinkFactory = new MyLinkFactory();
adapter.setLinkFactory(myLinkFactory);
```

Loading a project file

A project is a combination of style sheets that supply rendering information and a data source that supplies the data to be represented in a network component. A project is saved as an XML file with extension `.itpr`.

Loading a project file is the recommended way to configure a graphic component in Java™ as it is the fastest.

How to load a project file into a network component

The following code sample shows how to load a project file into a network component, using the method `setProject`.

```
IlpNetwork network = new IlpNetwork();
network.setProject(new URL("file:project.itpr");
```

The project is represented by the `IlpTGOProject` class, included in the package `ilog.cpl.project`. When a new project is created, the style sheet and data source are both null.

How to create a new project for the network component

The following code sample shows how to create a new project file by setting the style sheets and data source, then saving the project.

```
IlpTGOProject project = new IlpTGOProject();
project.setStyleSheet(new URL("file:example.css");
IltDefaultDataSource dataSource = new IltDefaultDataSource();
dataSource.setFileName("data.xml");
project.setDataSource(dataSource);
project.write(new URL("file:example.itpr");
```

Customizing the rendering of network nodes and links

Network nodes and links can be customized through CSS according to their business class. For details, see Customizing network and equipment nodes and Customizing network and equipment links.

Network component services

Describes the services that are available for a network: view services, adapter services, and handler services.

In this section

Background support
Describes how to use the background API to integrate various types of background in the network and equipment components.

Filtering
Describes how to filter nodes displayed by the network component.

Accepted and excluded classes
Details how to specify the business classes to be accepted for or excluded from display in the network component.

Setting a list of origins
Describes how to set a list of orgins to explicitly select the root nodes to be displayed by the network component.

Node factory
Describes the node factory.

Link factory
Describes the link factory.

Expansion strategy
Describes the expansion strategy used by the network adaptor to determine whether objects should be loaded in the network model.

Introduction to network component services

The services that are available for a network are of three kinds:

- View services, related to the network view
  - *Interacting with the network view*
  - *Interacting with the network objects*
  - *Positioning*
  - *Layout*
  - *Label layout*
  - *Layers*
  - *Zooming*
  - *Background support*
- Adapter services, related to the network model
  - *Filtering*
  - *Accepted and excluded classes*
  - *Setting a list of origins*
  - *Node factory*
  - *Link factory*
  - *Expansion strategy*
- Handler services, related to the network controller

Interacting with the network view

The `IlpNetwork` allows you to associate behavior with the network view as a whole and with the business objects it contains. JViews TGO provides predefined view interactors to manage the behavior of the network view. See *View interactors*.

With the default view interactors, you can:

♦ associate actions with mouse events and focus events

♦ associate actions with keyboard events

♦ define a pop-up menu factory to build a pop-up menu that displays in the view

Each view interactor works with one network view only and is managed by the network controller. A network view can have several interactors, but only one interactor is active at a time.

View interactors have two modes of operation:

♦ Transient

♦ Permanent

In *transient* mode, the view interactor removes itself from the network view when it has performed its action.

In *permanent* mode, the view interactor remains in the view until the controller removes it.

By default, view interactors are permanent.

View interactors can display a pop-up menu.

A view interactor has a context implemented through `IlpViewInteractionContext`. When a user gesture is completed, the network view clones this context and makes it accessible through `IlpViewActionEvent`.

The predefined view interactors are in the `ilog.tgo.interactor` package and are subclasses of `IlvManagerViewInteractor`. When one of these interactors is installed, it must be wrapped in an `IlpViewsViewInteractor`.

How to wrap a predefined view interactor when installing it

```
IlvManagerViewInteractor iltInteractor = new Ilt...Interactor();
IlpViewInteractor ilpInteractor =
  new IlpViewsViewInteractor(iltInteractor);
controller.setViewInteractor(ilpInteractor);
```

Setting the view interactor

The method `setViewInteractor(ilog.cpl.interactor.IlpViewInteractor)` allows you to set the view interactor, that is, the object-independent interactor, which is active at a given moment in the view. This interactor is replaced whenever the end user activates a different interactor. Such activation occurs, for example, when the user clicks a toolbar button.

Some interactors are one-shot interactors, that is, they have the attribute `permanent:false` in CSS. When such an interactor finishes its interaction, it is replaced by the default view interactor.

To customize through CSS, refer to *The Interactor rule*.

View interactor and default view interactor

When you use the `setViewInteractor` method you are attaching the specified interactor directly to the view. On the other hand, the `setDefaultViewInteractor(ilog.cpl.interactor.IlpViewInteractor)` allows you to define the interactor that will be attached when the current interactor is detached from the view and no other interactor is attached. The default interactor is not attached automatically to the view, so it will not be available immediately.

The following example combines both methods:

```
// Configuring the default view interactor and making
// it active
IltSelectInteractor selInteractor = new IltSelectInteractor();
selInteractor.setEditingAllowed(true);
IlpViewsViewInteractor viewsInteractor =
      new IlpViewsViewInteractor(selInteractor);

network.setDefaultViewInteractor(viewsInteractor);
network.setViewInteractor(viewsInteractor);
```

In this example, you define the default view interactor through the call to `setDefaultViewInteractor`, and you activate it through the call to `setViewInteractor` so that it is immediately available.

How to associate an action with a mouse event in the network view

You can associate actions with mouse events by using either CSS or the API.

The following extract shows how to customize the default view interactor in CSS:

```
Network {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.graphic.views.IlpViewsViewInteractor';
  action[0]: @+viewAction0;
}
```

```
Subobject#viewAction0 {
  class: 'ilog.cpl.interactor.IlpGestureAction';
  gesture: BUTTON3_CLICKED;
  action: @+myAction;
}

Subobject#myAction {
  class: MyAction;
}
```

The same configuration can be achieved through the API, as follows:

```
IlpNetwork network = // ...

// Retrieve the view interactor
IlpViewInteractor viewInteractor = network.getDefaultViewInteractor();

// Create an actionAction
myAction = new MyAction();

// Clicking the 3rd mouse button will trigger myAction
viewInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED,myAction);
```

You can also customize the actions that are associated with the interactors defined in the network component toolbar. This configuration is done with the toolbar button definition. The following CSS extract illustrates how this can be achieved:

How to associate an action with a mouse event for a network ToolBar button interactor

You can associate actions with mouse events when one of the interactors defined in the network component toolbar is active. The following CSS extract illustrates this configuration:

```
Network {
  toolbar: true;
}

ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
  action[0]: @+action0;
}

Subobject#action0 {
  gesture: BUTTON1_DOUBLE_CLICKED;
  class: "ShowDetailsAction";
}
```

In this configuration, the action "ShowDetailsAction" is triggered when a double-click event occurs while the selection interactor is set in the network view. You can define any list of actions associated with gestures by using the indexed property `action`. To be accepted by the CSS customization, the action class must be a JavaBean™.

You can find out whether this event occurred on an `IlpObject` by means of the following code (which should be in the `MyAction` class):

How to check whether a given action occurred in the network view interactor

```
// Implementation of the ActionListener interface
public void actionPerformed(ActionEvent e) {
  // ILOG JTGO interactors use IlpViewActionEvent
  IlpViewActionEvent viewEvent = (IlpViewActionEvent)e;
  // Get the IlpObject (if any) where the interaction occurred
  IlpObject ilpObj = viewEvent.getIlpObject();
  // Perform operation on the given object
}
```

How to associate an action with a keyboard event in the network view

You can associate actions with keyboard events by using either CSS or the API.

The following extract shows how to proceed in CSS:

```
Network {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.graphic.views.IlpViewsViewInteractor';
  action[0]: @+viewAction0;
}

Subobject#viewAction0 {
  class: 'ilog.cpl.interactor.IlpKeyStrokeAction';
  keyStroke: 'typed D';
  action: @+myAction;
}

Subobject#myAction {
  class: MyAction;
}
```

The same configuration can be achieved through the API, as follows:

```
// Create an actionAction myAction = new MyAction();
// Typing CTRL+D will trigger myAction
viewInteractor.setKeyStrokeAction(KeyStroke.getKeyStroke('D',java.awt.Event.
CTR
L_MASK),myAction);
```

You can also customize the keystroke actions that are associated with the interactors defined in the network component toolbar. This configuration is performed with the toolbar button definition. The following CSS extract illustrates how this can be achieved:

How to associate an action with a keyboard event for a network toolbar button interactor

You can associate actions with keyboard events when one of the interactors defined in the network component toolbar is active. The following CSS extract illustrates this configuration:

```
Network {
  toolbar: true;
}

ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
  action[0]: @+action0;
}

Subobject#action0 {
  key: "control A";
  class: "ilog.cpl.graph.action.IlpSelectAllObjectsAction";
}
```

In this configuration, the action "`IlpSelectAllObjectsAction`" is triggered when a Control-A keyboard event occurs while the selection interactor is set in the network view. You can define any list of actions associated with keyboard events by using the indexed property `action`. To be accepted by the CSS customization, the action class must be a JavaBean.

How to define a pop-up menu factory for the network view

You can customize a pop-up menu factory for the network view either through CSS or through the API.

The following extract shows how to add a pop-up menu factory through CSS:

```
Network {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.graphic.views.IlpViewsViewInteractor';
  popupMenuFactory: @+viewPopupMenuFactory;
}

Subobject#viewPopupMenuFactory {
  class: MyPopupMenuFactory;
}
```

The same configuration can be achieved through the API, as follows:

```
// Subclass IlpAbstractPopupMenuFactory, which has useful shortcuts
IlpPopupMenuFactory popupMenuFactory = new IlpAbstractPopupMenuFactory() {
  // Add the identifier of each of the selected objects to the menu
 public JPopupMenu createPopupMenu (IlpObjectSelectionModel ilpSelectionModel)

{
    // Create an empty popup menu
    JPopupMenu menu = new JPopupMenu();
    // Access the selected objects from the selection model
    Collection selectedObjects = ilpSelectionModel.getSelectedObjects();
    // fill the menu according to the current selection
    return menu;

  }
};
```

The following code shows you how to associate the defined pop-up menu factory with the network component:

How to associate a pop-up menu factory with the network component

```
// Set the popup menu factory to the view interactor
viewInteractor.setPopupMenuFactory(popupMenuFactory);
```

You can also customize a pop-up menu factory that is associated with the interactors defined in the network component toolbar. This configuration is performed with the toolbar button definition. The following CSS extract illustrates how this can be achieved:

How to define a pop-up menu factory for a network toolbar button interactor

The following CSS extract illustrates this configuration:

```
Network {
  toolbar: true;
}

ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
  popupMenuFactory: @=viewPopupMenuFactory;
}

Subobject#viewPopupMenuFactory {
  class: 'AlarmPopupMenuFactory';
}
```

The pop-up menu factory is customized using property `popupMenuFactory` in the button configuration. To be accepted during the CSS customization, the pop-up menu factory class must be a JavaBean.

Selection interactor

The `IltSelectInteractor` class allows you to select, move, and interact directly with objects. You can:

♦ Click an object to select it and deselect all other objects.

♦ Use Shift-click to select an unselected object or to deselect a selected object while keeping other prior selected objects selected.

♦ Drag a selection rectangle to select all objects within this rectangle.

♦ Drag one selected object and thereby cause all other selected objects to move in relation to it.

If `setUsingObjectInteractor(true)` is called on the interactor, then you can also:

♦ Use other gestures that are understood by a specific object interactor.

You can configure the selection interactor to use Ctrl-click instead of Shift-click for multiple selection.

How to configure the selection interactor for multiple selection in the network view

In CSS, use the following rules:

```
Subobject#SelectButton {
  multipleSelectionModifier: "java.awt.event.InputEvent.CTRL_MASK";
  selectionModifier: "java.awt.event.InputEvent.SHIFT_MASK";
}
```

In the API, use the following code:

```
setMultipleSelectionModifier(InputEvent.CTRL_MASK);
setSelectionModifer(InputEvent.SHIFT_MASK);
```

Group reshape interactor

The `IltEditGroupInteractor` class allows you to change the shape of rectangular, polygonal, and linear groups (`IltGroup`, `IltLinearGroup`, `IltRectGroup` and `IltPolyGroup`). Clicking an object starts shape editing interaction. Clicking the background ends it.

For polygonal and linear groups:

- Dragging a vertex moves it.
- Ctrl-click on a vertex removes it.
- Ctrl-click on an edge adds a vertex at that point on the edge.

Make rectangular node interactor

The `IltMakeRectGroupInteractor` class creates a node with a rectangular shape (`IltRectGroup`). One corner of the rectangle is denoted by the point where the cursor is located when the mouse button is released. Make sure that you do really move the mouse between the time when you press and the time when you release the mouse button. Otherwise, the shape is created empty and the node might be invisible.

Make polygonal node interactor

The `IltMakePolyGroupInteractor` class creates a node with a polygonal shape (`IltPolyGroup`). A point is added each time the user clicks. Double-clicking marks the last point to be added.

Make polyline node interactor

The `IltMakeLinearGroupInteractor` class creates a node with a polyline shape (`IltLinearGroup`). A point is added each time the user clicks. Double-clicking marks the last point to be added.

Make link interactor

The `IltMakeLinkInteractor` class creates links (`IltLink`) between nodes. This interactor works in the following way: the user clicks one node and then goes on dragging the mouse over another node so that the two nodes are selected. When the user releases the mouse, a link is drawn between the two nodes.

For a detailed description of interactors and gestures, refer to *Interacting with the graphic components*.

Interacting with the network objects

Interacting with the network view describes how to set an interactor on the entire network view. You can also associate behavior with business objects (a whole class or individual objects), as well as with individual object instances.

To do so, you use object interactors, which offer you the same possibilities as the view interactor:

- ♦ Associating actions with mouse events
- ♦ Associating actions with keyboard events
- ♦ Defining a pop-up menu factory to build a pop-up menu that displays on representation objects

An object interactor handles any event occurring to the object with which it is associated, provided the view interactor has enabled the use of object interactors. You can check this with the `isUsingObjectInteractor` method or modify it with the `setUsingObjectInteractor` method.

Object interactors are enabled by default.

No default interactor is associated with any object. To associate actions with mouse or keyboard events, or to define a pop-up menu factory, you first have to create an instance of `IlpObjectInteractor`. You can use the `IlpDefaultObjectInteractor` class, extend it, or create your own implementation.

How to associate an object interactor with a network component object

You can associate an object interactor with a representation object by using either CSS or the API.

The following extract shows how to proceed in CSS:

```
Network {
  interactor: true;
}

object."ilog.tgo.model.IltNetworkElement" {
  interactor: @+objInteractor;
}

Subobject#objInteractor {
  class: 'ilog.cpl.interactor.IlpDefaultObjectInteractor';
}
```

The same configuration can be achieved through the API, as follows:

```
IlpNetwork network = // ...
IlpNetworkController networkController = network.getController();
```

```
// Create an object interactor
IlpObjectInteractor objectInteractor = new IlpDefaultObjectInteractor();
networkController.setObjectInteractor( bo, objectInteractor);
// Configuring the specific object interactor is similar to configuring
// a view interactor.
objectInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED, new MyAction())
;
```

Actions related to mouse and keyboard events can be customized in the same way as for the view interactor. You can also define a pop-up menu factory in the same way as for the view interactor. Refer to *Interacting with the network view*.

An object interactor can also be associated with a specific decoration that is part of the business object graphic representation in the network view. Each decoration represents a business attribute in the model. Therefore the customization of the interactor for a specific decoration takes into account the business object and a business attribute as illustrated below:

How to associate an object interactor with the label decoration in a network component object

You can associate an object interactor with one of the graphic decorations of the object by setting the interactor to the business attribute that is represented. You can do it using CSS or the API.

The following extract shows how to proceed in CSS:

```
Network {
  interactor: true;
}

object."ilog.tgo.model.IltNetworkElement/name" {
  interactor: @+objInteractor;
}

Subobject#objInteractor {
  class: 'ilog.cpl.interactor.IlpDefaultObjectInteractor';
}
```

The same configuration can be achieved through the API, as follows:

```
IlpNetwork network = // ...
IlpNetworkController networkController = network.getController();
// Create an object interactor
IlpObjectInteractor objectInteractor = new IlpDefaultObjectInteractor();
networkController.setObjectInteractor( bo, IltObject.NameAttribute,
objectInteractor);
// Configuring the specific object interactor is similar to configuring
// a view interactor.
objectInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED,   new
MyAction());
```

Actions related to mouse and keyboard events can be customized in the same way as for the view interactor. You can also define a pop-up menu factory in the same way as for the view interactor. Refer to *Interacting with the network view*.

For a detailed description of interactors and gestures, refer to *Interacting with the graphic components*.

Positioning

This facility is for defining where a given object is displayed on the screen.

Each representation object has the option of carrying a position. The position data can originate from the back end (mapped automatically from an attribute of the corresponding business object, or manually from a separate source), from computation by a default layout algorithm in the network view, or from explicit end-user gestures to move or reshape nodes in the view. It is possible to retrieve the origin of the position data through the `IlpPositionSource` enumeration. There are three possible origins for the position of representation objects:

- `BACKEND`: The position has been set by the adapter when creating the representation object.
- `LAYOUT`: The position has been set in the view by an automatic layout algorithm (either the node layout or the link layout).
- `USER`: The position has been set by the user through interactors.

To retrieve the position origin, you need to access the method `getPositionSource` in the network view.

The position implements the `IlpPosition` interface.The following supplied types implement `IlpPosition`:

- `IlpPoint` for fixed-size nodes
- `IlpRect` for variable-size nodes
- `IlpPolyline` for links
- `IlpPolygon` for polygonal regions
- `IlpRelativePoint` for elements of `IltCard`
- `IlpShelfItemPosition` for elements of `IltShelf` and `IltCardCarrier`

The position can be attached to a representation object through the method `setPosition` and retrieved through the method `getPosition` of the network view (`IlpNetworkView`).

A position converter is used to convert positions in the network model to positions in the network view or vice versa. The position converter implements the `IlpPositionConverter` interface.

JViews TGO provides a default position converter (`IlpDefaultPositionConverter`) for the predefined business objects position data. The following predefined position types are supported:

- `IlpPoint`
- `IlpRect`
- `IlpPolygon`
- `IlpPolyline`

♦ IlpShelfItemPosition

Since the network view already supports these position types, the IlpDefaultPositionConverter does nothing else than verify that the given position is one of the predefined types.

Positioning using geographic coordinates

A more complex converter is provided for geographic coordinates (IlpGeographicPositionConverter). It extends the default position converter to support the following types of positions:

♦ IlpGeographicPosition (in the model) <-> IlpPoint (in the view)

♦ IlpGeographicPolygon (in the model) <-> IlpPolygon (in the view)

♦ IlpGeographicPolyline (in the model) <-> IlpPolyline (in the view)

When you use a background map, you can give the positions of objects directly in geographic coordinates (latitude/longitude). The conversion to screen coordinates is performed by using an instance of IlpGeographicPositionConverter. You can parameterize this converter through classes that are part of the IBM® ILOG® JViews Maps product: an IlvProjection or an IlvMathTransform, optionally followed by an IlvTransformer. Refer to *The two coordinate systems*.

How to parameterize the geographic position converter

```
// Set the position converter. It converts the geographic coordinates
// of the objects in the XML file to planar (x,y) coordinates.
IlvProjection projection =
  new IlvEquidistantCylindricalProjection();
IlvTransformer t = new IlvTransformer(0.00001,0,0,0.00001,1,6.5);
IlpPositionConverter converter =
  new IlpGeographicPositionConverter(projection,true,t);
networkComponent.setPositionConverter(converter);
```

You can also parameterize the geographic position converter through CSS by using the following steps:

1. Define a converter class with an empty constructor.

```
public class MyConverter extends IlpGeographicPositionConverter {
       public MyConverter() {
         super(new IlvEquidistantCylindricalProjection(),
               true,
               new IlvTransformer(0.00001, 0, 0, 0.00001, 1, 6.5));
       }
     }
```

2. Reference the converter class in a CSS file as follows.

```
Positioning {
      positionClass: 'ilog.cpl.network.IlpGeographicPosition';
      converterClass: 'my.package.MyConverter';
    }
```

Important: The parameters used to configure an IlpGeographicPositionConverter should conform to the georeferencing configuration of the background map in use. For details, see *Background support*.

You can define your own application-specific implementation of the `IlpPosition` interface, for example, you could implement polar coordinates. When you define your own implementation of `IlpPosition`, you must also attach the corresponding implementation of the `IlpPositionConverter` to the view. In this particular case, you would attach a converter from polar positions to the predefined view position types.

When an object has no attached position, the view assigns a position to the corresponding graphic object. The position is assigned through the layout mechanism (node layout for positioning nodes, and link layout for shaping links).

If the position of an object changes due to user interaction, the controller requests the handler to confirm the change.

Layout

JViews TGO makes use of the IBM® ILOG® JViews graph layout algorithms. Each `IlpNetworkView` can be connected to several node algorithms and one link algorithm.

- The *node layout* algorithms are:
  - `IlvBusLayout`
  - `IlvCircularLayout`
  - `IlvHierarchicalLayout`
  - `IlvRandomLayout`
  - `IlvSpringEmbedderLayout`
  - `IlvTopologicalMeshLayout`
  - `IlvTreeLayout`
  - `IlvUniformLengthEdgesLayout`
- The *link layout* algorithms can be:
  - `IlvLinkLayout`
  - `IlvShortLinkLayout`
  - `IlvLongLinkLayout`
  - `IltLinkLayout`
  - `IltShortLinkLayout`
  - `IltLocalLinkLayout`
  - `IltStraightLinkLayout`

Note: The difference between Ilv... and Ilt... link layout algorithms is that Ilt... algorithms support connection ports whereas Ilv... algorithms don't.

The detailed description of all the graph layout algorithms can be found in the IBM® ILOG® JViews *Diagrammer Using Graph Layout Algorithms* documentation.

In case of multiple layouts, one layout can be set to be applied automatically whenever the contents of the view changes, while the others can be applied on demand. If the view contains subnetworks, you can specify different node layouts and a different link layout for the subnetworks.

To configure the layouts, it is recommended to use CSS (see *Configuring a network component through a CSS file*). Using CSS, you can also configure per-object layout parameters.

If a layout takes too much time to execute, or if you want to add toolbar buttons to execute a layout, you can configure the layout for the view and subnetworks, then execute it on demand by using the API method `performAttachedLayout(int)`. The advantage of this method over the method `performLayoutOnce(ilog.views.graphlayout.IlvGraphLayout)` is that the layout remains attached to the view, therefore storing any previously-defined configuration.

The class `IlpNetworkView` provides the following methods to handle the layout operation:

- void setNodeLayout (`IlvGraphLayout` layout, boolean perform). This method sets the given layout as the default for this `IlpNetworkView`. If the `perform` parameter is set to `true`, the layout is applied to the objects immediately. With this method the layout is executed every time the network content changes.
- void `setLinkLayout (IlvGraphLayout layout, boolean perform)`. This method sets the given layout as the default link layout for this instance of `IlpNetworkView`. If the `perform` parameter is set to `true`, the layout is applied to the links immediately. With this method the layout is executed every time the network content changes.
- `public void performLayoutOnce(IlvGraphLayout layout)`. This method executes the layout algorithm once on the manager content.
- `void startDelayingUpdates()`. This method suspends temporarily the layout operations. This mechanism avoids unnecessary computation when you intend to perform a sequence of operations that affect the network layout.
- `void endDelayingUpdates()`. This method resumes the layout operations suspended by a call to the method `startDelayingUpdates`. Any operation that requests a layout recalculation is suspended when it is executed between `startDelayingUpdates` and `endDelayingUpdates` calls.
- `void setGraphLayouts(IlvGraphLayout[] layouts)`. This method sets the given graph layouts for this `IlpNetworkView`. Several graph layouts can be set to position nodes in the view. One of them can be configured to be executed every time the network contents changes. This method does not apply the layout to the nodes immediately. All the graph layouts given as argument to the method are attached to the view.
- `void setGraphLayouts(int index, IlvGraphLayout layout)`. This method sets a new graph layout for the `IlpNetworkView` or replaces an existing graph layout. This method does not apply the layout to the nodes immediately.
- `IlvGraphLayout[] getGraphLayouts()`. This method returns the graph layouts that have been configured for the view.
- `IlvGraphLayout getGraphLayouts(int index)`. This method returns the graph layout that is configured for the view at the given index.
- `void setAutoLayoutIndex (int index)`. This method indicates, from the list of graph layouts that have been configured using method `setGraphLayouts`, which one is executed automatically when the contents of the view changes.
- `int getAutoLayoutIndex()`. This method returns the index of the graph layout that is executed automatically when the contents of the view changes.
- `void setGraphLayouts(IlpRepresentationObject ro, IlvGraphLayout[] layouts)`. This method allows you to set graph layouts that can be used to position nodes in the

subnetwork corresponding to the given representation object. One of the graph layouts can be configured to be executed automatically when the contents of the subnetwork changes. The other graph layouts are attached to the view and can be performed on demand.

- `void setGraphLayouts(IlpRepresentationObject ro, int index, IlvGraphLayout layout)`. This method allows you to set or replace a graph layout used to position nodes in a subnetwork.
- `IlvGraphLayout[] getGraphLayouts(IlpRepresentationObject ro)`. This method returns the graph layouts that have been configured for the given subnetwork.
- `IlvGraphLayout getGraphLayouts(IlpRepresentationObject ro, int index)`. This method returns the graph layout that has been configured for the given subnetwork at the given index.
- `void setAutoLayoutIndex(IlpRepresentationObject)`. This method defines which graph layout configured using `setGraphLayouts(IlpRepresentationObject)` will execute automatically when the contents of the subnetwork changes.
- `int getAutoLayoutIndex(IlpRepresentationObject)`. This method returns the graph layout that has been configured to execute automatically when the contents of the subnetwork changes.
- `void setLinkLayout(IlpRepresentationObject)`. This method defines a link layout for the given subnetwork.
- `void performAttachedLayout(int index)`. This method executes the layout that has been configured for the given index, recursively in the object tree. The layout has been already attached to the view and keeps the configuration whenever it is performed.
- `IlvGraphic getLayoutProxy(IlpRepresentationObject)`. This method returns the graphic object corresponding to the given representation object for layout purposes. This method should be used if you need to set per-object layout properties to configure the layout algorithms.

Note: Graphical parameters, such as the layout region, that are passed to the graph layout are expressed in view coordinates. Therefore, if you have expressed these parameters in stationary coordinates, you must transform them to view coordinates (by applying `network.getView().getCompositeGrapher().getZoomTransformer()` `network.getManagerView().getTransformer()`) before passing them to the graph layout.

How to use hierarchical node layout in the network component

In CSS, use the following rules:

```
Network {
  graphLayout: true;
}
```

```
GraphLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Bottom;
  levelJustification: Top;
  globalLinkStyle:
'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout.POLYLINE_STYLE';
  connectorStyle:
'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout.EVENLY_SPACED_PINS';
}
```

Note that the full class path is required for the properties `globalLinkStyle` and `connectorStyle` for the type converter to locate and convert the constants.

For an example of a CSS link layout, refer to *The LinkLayout rule*.

In the API, use the following code:

```
IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
layout.setFlowDirection (IlvDirection.Bottom);
layout.setLevelJustification (IlvDirection.Top);
layout.setGlobalLinkStyle (IlvHierarchicalLayout.POLYLINE_STYLE);
layout.setConnectorStyle (IlvHierarchicalLayout.EVENLY_SPACED_PINS);

network.setNodeLayout(layout);
```

Note: All layouts to be used with JViews TGO must be set in view coordinate mode. This mode is automatically set when you install layouts through `setNodeLayout(ilog.views.graphlayout.IlvGraphLayout)`, `setLinkLayout(ilog.views.graphlayout.IlvGraphLayout)` `setLinkLayout(ilog.views.graphlayout.IlvGraphLayout)`, or `performLayoutOnce(ilog.views.graphlayout.IlvGraphLayout)`. If you call the method `performLayout()` directly, you must first set the mode: `layout.setCoordinatesMode(IlvGraphLayout.VIEW_COORDINATES);`

Even when you decide to use a certain node or link layout, you may want some links or nodes to be *pinned*; that is, you may want to keep a certain element in a specified position that is not affected when the layout is executed on a network.

You can achieve this effect by using the following methods defined in `IlvGraphLayout`:

♦ `setPreserveFixedLinks (boolean preserve)`. This method determines whether or not the layout will preserve the position of the registered links.

♦ `setPreserveFixedNodes (boolean preserve)`. This method determines whether or not the layout will preserve the position of the registered nodes.

♦ `void setFixed (Object obj /* link or node */, boolean fix)`. This method determines whether or not the given object will be fixed in the network.

♦ `boolean isFixed (Object obj)`. The return value indicates whether or not the given object is marked to be fixed. The object to be passed is an `IlvGraphic` belonging to the `IlpGraphic` that represents the object and *not* the `IlpGraphic` or `IlpRepresentationObject` itself.

♦ `void unfixAllLinks()`

♦ `void unfixAllNodes()`

How to set a link to fixed shape and a node to fixed position in the network view

The following code illustrates how you can set a given link to have a fixed shape and a given node to have a fixed position in the network.

In CSS, use the following rules:

```
Network {
  graphLayout: true;
  linkLayout: true;
}

LinkLayout {
  class: 'ilog.views.graphlayout.link.IlvLinkLayout';
}

GraphLayout {
  layouts[0]: @+treeLayout;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
}

#Link:linkLayoutRenderer {
  fixed: true;
}

#NE1:graphLayoutRenderer:tree {
  fixed: true;
}
```

In the API, use the following code:

```
IlvGraphLayout layout = networkView.getLinkLayout();
layout.setPreserveFixedLinks (true);
IlpRepresentationObject linkRO =  networkAdapter.getRepresentationObject(link)
;
layout.setFixed(networkView.getLayoutProxy(linkRO));

layout = networkView.getNodeLayout();
layout.setPreserveFixedNodes (true);
```

```
IlpRepresentationObject neRO =  networkAdapter.getRepresentationObject(ne);
layout.setFixed(networkView.getLayoutProxy(neRO));
```

When the position or shape of an object is not handled by the layout, you must set it by calling the method `setPosition(ilog.cpl.model.IlpRepresentationObject, ilog.cpl.graphic.IlpPosition, ilog.cpl.graphic.IlpPositionSource)` (or `IlpNetworkView.setPosition`).

JViews TGO provides a default layout which uses `IlvShortLinkLayout` to shape and position links. This default layout sets all objects without an attached position to `(0,0)`.

How to use multiple node layouts in a network view

In this scenario, two node layouts are configured for the network view. The first one is configured to be executed automatically.

In CSS, use the following rules:

```
Network {
  graphLayout: true;
}

GraphLayout {
  layouts[0]: @+hierarchicalLayout;
  layouts[1]: @+treeLayout;
  autoLayoutIndex: 0;
}

Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Bottom;
  levelJustification: Top;
  globalLinkStyle: POLYLINE_STYLE;
  connectorStyle: EVENLY_SPACED_PINS;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
  flowDirection: Bottom;
}
```

The same layout that is applied to the network view will be applied to the whole subnetwork hierarchy.

In the API, use the following code:

```
IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
layout.setFlowDirection (IlvDirection.Bottom);
layout.setLevelJustification (IlvDirection.Top);
layout.setGlobalLinkStyle (IlvHierarchicalLayout.POLYLINE_STYLE);
layout.setConnectorStyle (IlvHierarchicalLayout.EVENLY_SPACED_PINS);

IlvTreeLayout treeLayout = new IlvTreeLayout();
```

```
layout.setFlowDirection(IlvDirection.Bottom);

network.setGraphLayouts(new IlvGraphLayout[] { layout, treeLayout });
network.getView().optimizeLayout();
```

To execute the tree layout in the view, use the method `performAttachedLayout`, where the index is the one defined in the CSS configuration, as illustrated below:

```
network.getView().performAttachedLayout(1);
```

How to use different layouts for the view and subnetworks

It is also possible to configure some subnetworks with layout algorithms that are different from the network view.

Note: If the subnetworks have intergraph links, the link layout renderer must be enabled, otherwise the intergraph links will not be routed.

In this scenario, the object '`SubNetwork1`' positions its nodes using a tree layout algorithm, while the nodes in the main view are positioned using a grid layout.

In CSS, use the following rules:

```
Network {
  graphLayout: true;
  linkLayout: true;
}

LinkLayout {
  class: 'ilog.views.graphlayout.link.IlvLinkLayout';
}

GraphLayout {
  layouts[0]: @+gridLayout;
}

Subobject#gridLayout {
  class: 'ilog.views.graphlayout.grid.IlvGridLayout';
}

#SubNetwork1:graphLayoutRenderer {
  layouts[0]: @+treeLayout;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
}
```

In the API, use the following code:

```
IlvGridLayout gridLayout = new IlvGridLayout();
network.setGraphLayouts(new IlvGraphLayout[] { gridLayout });

IlvTreeLayout treeLayout = new IlvTreeLayout();
layout.setFlowDirection(IlvDirection.Bottom);

IlpRepresentationObject ro =
network.getAdapter().getRepresentationObject("SubNetwork1");
network.getView().setGraphLayouts(ro, new IlvGraphLayout[] { treeLayout });
network.getView().optimizeLayout();
```

How to configure per-object layout properties in the network component

Some layout algorithms require a specific configuration in order to be properly executed. For example, the bus layout needs to have a bus object specified; or the tree layout, for which you may want to specify the root node prior to the layout execution. Starting from JViews TGO 7.5, you can configure these properties using CSS, as illustrated below:

```
Network {
  graphLayout: true;
}

GraphLayout {
  layouts[0]: @+busLayout
}

Subobject#busLayout {
  class: 'ilog.views.graphlayout.bus.IlvBusLayout';
  horizontalOffset: 50;
  verticalOffsetToLevel: 50;
  verticalOffsetToPreviousLevel: 40;
  margin: 30;
  marginOnBus: 50;
}

// Configure the bus object as the bus in the layout
// All layout configuration uses the 'graphLayoutRenderer'
// and the graph layout name pseudoclasses
#BUS:graphLayoutRenderer:bus {
  bus: true;
}

// Configure the bus to route the links that connect the
// bus to the nodes
#BUS {
  linksConnectToBase: true;
}
```

or, using the API:

```
IlvBusLayout busLayout = new IlvBusLayout();
network.setGraphLayouts(new IlvGraphLayout[] { busLayout });

IlpRepresentationObject ro =
network.getAdapter().getRepresentationObject("BUS");
IlvGraphic layoutProxy = network.getView().getLayoutProxy(ro);
busLayout.setBus((IlvPolyPointsInterface)layoutProxy);
```

How to disable the per-object layout properties configuration in the network view

By default, per-object layout parameters can be configured using CSS. However, if you are not interested in this feature, you can disable it by setting the property `usePerObjectParamenters` in the graph layout renderer and link layout renderer. Disabling the per-object layout properties configuration speeds up significantly the rendering process.

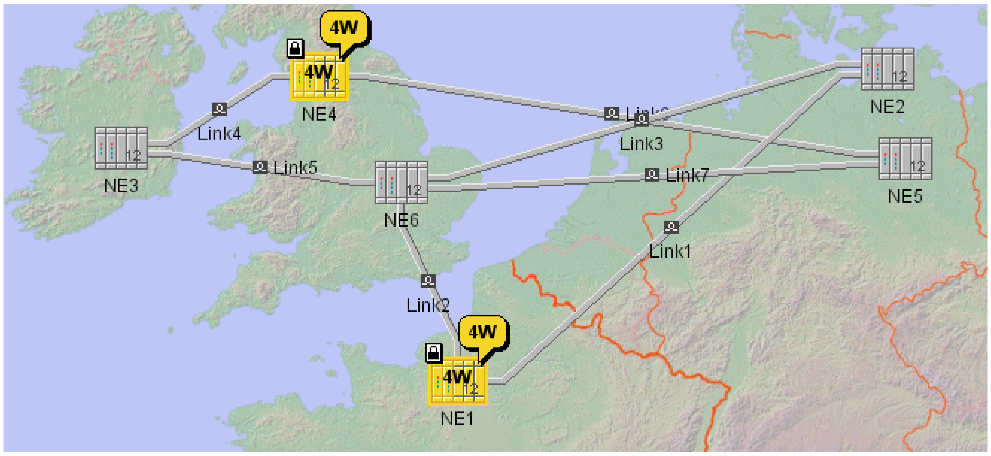
```
Network {
  graphLayout: true;
  linkLayout: true;
}

LinkLayout {
  class: 'ilog.views.graphlayout.link.IlvLinkLayout';
  usePerObjectParameters: false;
}

GraphLayout {
  layouts[0]: @+treeLayout;
  usePerObjectParameters: false;
}
```

Label layout

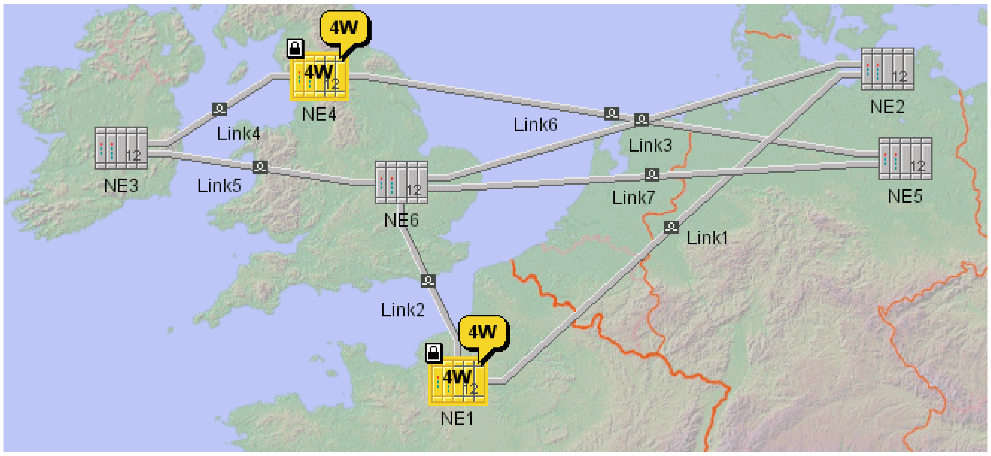
JViews TGO allows you to place labels in a network automatically to make them easier to read. This placement overrides the default placement of labels in JViews TGO, namely:

- ♦ Below network elements
- ♦ At the center of gravity of groups

Label layout allows you to reduce overlap between labels and other objects in the current view. Therefore, it is particularly useful for positioning labels on links. *Network Links without Label Layout* shows labels positioned by default. *Network Links with Label Layout* shows the same links with customized positioning of the labels through label layout.



Network Links without Label Layout



Network Links with Label Layout

Note: Label layout tries to find the best position for your labels, but sometimes even this mechanism does not achieve attractive results. To get the best results, leave large spaces between network objects.

Label layout in JViews TGO uses the label layout of IBM® ILOG® JViews. See the IBM® ILOG® JViews *Diagrammer Using Graph Layout Algorithms* documentation for more details on label layout.

Using label layout

JViews TGO provides the `IltAnnealingLabelLayout` class (a subclass of `IlvAnnealingLabelLayout`) that moves the labels of the nodes and the links so that they do not overlap. If it is impossible to prevent some overlap, this class will minimize the overlap between different labels.

The class `IlpNetworkView` provides the following method for handling label layout:

```
void setLabelLayout  (IltAnnealingLabelLayout layout);
```

This method sets the given layout for the specified view. This method is also available from the class `IlpNetwork`, for convenience.

Unlike node layout and link layout, label layout is not performed automatically when the content of the network changes. You need to call `labelLayout.performLayout()` explicitly. There is a toolbar button for triggering label layout computation. See the class `IlpNetworkLabelLayoutButton`.

Here is a typical example of how to use label layout.

How to use label layout

```
IlpNetwork network = new IlpNetwork();

// fill the network with your elements
IltAnnealingLabelLayout labelLayout =
  new IltAnnealingLabelLayout();
network.setLabelLayout(labelLayout);
labelLayout.performLayout();
network.setLabelLayout(null);
```

In this example you:

1. Create the label layout.

   ```
   IltAnnealingLabelLayout labelLayout =
     new IltAnnealingLabelLayout();
   ```

2. Attach this layout to the current network.

```
network.setLabelLayout(labelLayout);
```

3. Perform the layout, placing the labels esthetically and where they will be easy to read.

```
labelLayout.performLayout();
```

4. Optionally, detach the layout from the network and release the resources used by label layout. The positions of the labels are maintained.

```
network.setLabelLayout(null);
```

Defining the labels you want to position

Label layout positions only the labels of links by default. You can choose to apply the layout to other types of object through the following methods:

```
void setObjects (IltLabelLayoutConstants[] types)
void setObjects (int index, IltLabelLayoutConstants type)
IltLabelLayoutConstants[] getObjects()
IltLabelLayoutConstants getObjects (int index)
```

or through the following convenience methods:

```
setUsesOthers(boolean flag)
setUsesLinks(boolean flag)
setUsesNetworkElements(boolean flag)
setUsesBTS(boolean flag)
setUsesLinearGroups(boolean flag)
setUsesPolyGroups(boolean flag)
setUsesRectGroups(boolean flag)
```

How to apply label layout to network elements only

```
IltAnnealingLabelLayout labelLayout =
  new IltAnnealingLabelLayout();
labelLayout.setObjects(0, IltLabelLayoutConstants.NETWORK_ELEMENTS);
network.setLabelLayout(labelLayout);
network.performLabelLayout();
network.setLabelLayout(null);
```

The method `setObjects` indicates whether a certain type of object will have its label placed by the label layout. In this example, only Network Elements will have their labels placed by the label layout.

How to apply label layout to network elements only using CSS

```
Network {
  labelLayout: true;
}


LabelLayout {
  class: 'ilog.tgo.graphic.graphlayout.labellayout.IltAnnealingLabelLayout";
  objects[0]: NETWORK_ELEMENTS;
}
```

The property `objects` indicates whether a certain type of object will have its label placed by the label layout. In this example, only network elements will have their labels placed by the label layout.

Defining obstacles

The label layout positions the labels of the objects by taking into account the obstacles that are in the network view. By default, all objects are considered as obstacles, but you can configure this behavior in CSS or through the API:

How to specify obstables in the label layout using CSS

```
Network {
   labelLayout: true;
}

LabelLayout {
   class:
'ilog.tgo.graphic.graphlayout.labellayout.IltAnnealingLabelLayout';
   obstacles[0]: NETWORK_ELEMENTS;
   obstacles[1]: LINKS;
}
```

How to specify obstables in the label layout using the API

```
void setObstacles (IltLabelLayoutConstants[] types)
void setObstacles (int index, IltLabelLayoutConstants type)
IltLabelLayoutConstants[] getObstacles()
IltLabelLayoutConstants getObstacles (int index)
```

Constraints

The following constraints exist for label layout:

- Label layout does not work in subnetworks.
- If you move an object on which you have performed label layout, the label position will not be maintained. You must perform label layout again on the object after the move.

- Label layout is not appropriate for dynamically changing networks.

  Label layout is designed for stable networks. If you modify your network frequently, you need to call the method `performLayout()` whenever any object with a label changes. The IBM® ILOG® JViews algorithm for annealing label layout used in JViews TGO consumes a lot of resources. Therefore, it is not recommend to call the method `performLayout()` often. Note also that the layout does not always find the same position for the labels. See the IBM® ILOG® JViews *Diagrammer User's Documentation* for more information.

You can configure the label layout in a CSS file through the `labelLayout` property. For more information, see *The LabelLayout rule*.

Layers

The grapher in JViews TGO has a set of predefined rules for assigning layers to objects that are inserted in the grapher. The choice of layer for a given object can be customized using the abstract class `IltLayerPolicy`. The `IlpNetworkView` class provides the following default layering policy in `setLayerPolicy(ilog.tgo.composite.IltcLayerPolicy)`, for which no coding is required:

- ♦ Nodes are gathered in the same layer.
- ♦ Links are gathered in a layer below the node layer.
- ♦ Regions are gathered in a layer below the link layer.

JViews TGO adopts the following conventions for displaying network objects and their associated decorations:

- ♦ Within a layer, a graphic object is displayed with all its related decorations (site label, function icon, family label, status icon, and so on).
- ♦ The only exception concerns alarm balloons, which are systematically displayed above all the objects in the view. Information windows are displayed above all other objects.

The layer in which a representation object is displayed (along with all its associated decorations) is determined by the `IltLayerPolicy`.

Note: The network and each of its subnetworks have their own set of layers. The descriptions concerning layers apply separately to each grapher.

JViews TGO provides support for layers in the form of a layer policy and a layer visibility.

Layer policy

Each graphic representation of an object is an instance of the class `IlpGraphic`. It has a set of decorations that are instances of the class `IlvGraphic`. (Note that the `IlpGraphic` class itself is a subclass of `IlvGraphic`.) The graphic representations (`IlpGraphic`) are layered according to a given order, and so are the decorations. The mechanism that defines the stacking order for decorations and graphic representations is called *layer policy*, and is implemented by the abstract class `IltLayerPolicy`. This mechanism defines how two graphic representations overlap and ensures that their decorations do not intertwine. See the stacking order defined at the beginning of *Layers*.

Each `IlpNetworkView` has a layer policy that defines in which order the objects will be displayed in the grapher. This layer policy is created during initialization, but you can also modify it through the `IltCompositeGrapher` class:

- ♦ `getLayerPolicy()` returns the current layer policy.
- ♦ `setLayerPolicy(ilog.tgo.composite.IltcLayerPolicy)` changes the current layer policy. The `policy` object specified as a parameter should be an instance of `IltLayerPolicy`.

You get the `IltCompositeGrapher` that belongs to an `IlpNetworkView` by calling `network.getView().getCompositeGrapher()`.

The layer policy is responsible for allocating specific layers in the IBM® ILOG® JViews grapher. The grapher is used to define the stacking order of the decorations. JViews TGO provides a specific class for handling layers called `IltcLayer`. This interface is the extension of `IlvManagerLayer` for `IlpGraphic` objects. Even though JViews TGO uses `IltcLayer` instances, it is also possible to use `IlvManagerLayer` instances directly to place graphic objects like map backgrounds or annotations.

Important: The created layers must be referenced using `IlvManagerLayer` instead of the layer numbers, since JViews TGO may cause layers to be added or removed, which invalidates the previous layer numbers.

To create your own layer policy, you will have to implement the `IltLayerPolicy` interface that defines the following methods:

- `public IltcLayer`**getDefaultLayer** `(IlpGraphic graphic)`. This method returns the `IlpLayer` into which the main objects will be inserted. If you want to define specific layers according to the type of object, you should implement this method.
- `public IltcLayer`**getElementLayer** `(IlpGraphic graphic, IltGraphicElementName element)`. This method returns the `IltcLayer` into which a specific decoration will be inserted. This method determines whether alarm balloons will be displayed on top of the other decorations.
- `public boolean` **isRemovable** `( IltcLayer`. This method indicates which layers cannot be removed; for example, all the layers created by this layer policy.

The layers returned by the methods above can be created in layer policy initialization. The class `IltCompositeGrapher` provides methods through its superclass for dynamically allocating the layers in the grapher to execute this operation.

- `addLayerOnTop/addLayerBelow ( IltcLayer`. These methods create a new `IltcLayer` above or below the given `IltcLayer`.
- `addLayerOnTop/addLayerBelow ( IlvManagerLayer`. These methods create a new `IltcLayer` above or below the given `IlvManagerLayer`.
- `addIlvManagerLayerOnTop/addIlvManagerLayerBelow ( IltcLayer`. These methods create a new `IlvManagerLayer` above or below the given `IltcLayer`.
- `addLayerOnTop`. This method creates a new `IltcLayer` on top of all the other layers of this grapher.
- `addLayerAtBottom`. This method creates a new `IltcLayer` beneath all the other layers of this grapher.

The following code extract shows you how to create layers.

How to create layers

```
public MyLayerPolicy (IltCompositeGrapher grapher) {
  _groupLayer = grapher.addLayerOnTop ();
  _linkLayer = grapher.addLayerOnTop();
  _mainLayer = grapher.addLayerOnTop();
  _alarmBalloonLayer = grapher.addLayerOnTop();
  _infoWindowLayer = grapher.addLayerOnTop();
  _systemWindowLayer = grapher.addLayerOnTop();
}

public IltcLayer getElementLayer (IlpGraphic graphic,
                                  IltGraphicElementName element) {
  if (element == IltGraphicElementName.AlarmBalloon)
    return _alarmBalloonLayer;
  else if (element == IltGraphicElementName.InfoWindow)
    return _infoWindowLayer;
  else if (element == IltGraphicElementName.SystemWindow)
    return _systemWindowLayer;
  return null;
}
```

Note: When you use the layer policy and your own layers in an `IlpNetworkView`, JViews TGO expects all the layers for `IlpGraphic` instances and decorations to be adjacent. For example, if you intend to load an IVL file that has predefined layers for the vectorial elements, these layers must be previously allocated in the `IlvGrapher` instance before you create the `IlpNetworkView`. (See `IlvManager` in the IBM® ILOG® JViews *Java™ API Reference Documentation* for information on how to create layers.)

Layer visibility

The layering mechanism is frequently used to hide and show collections of objects. Since JViews TGO manages the allocation of layers and the distribution of `IlvGraphic` instances on them, the following methods are available for controlling the visibility of collections of objects.

There are three possible mechanisms for modifying the visibility of layers:

♦ To set the visibility of a specified JViews TGO layer to on or off, use the following:

  ● `setVisible(ilog.views.IlvManagerView, ilog.tgo.composite.IltcLayer, boolean)`IltCompositeGrapher.setVisible (IltcLayer layer, boolean visibility).

♦ To set the visibility of a specified IBM® ILOG® JViews Manager layer to on or off, use the following:

  ● `setVisible(ilog.cpl.graphic.views.IlpLayer, boolean)`IlpNetworkView.setVisible (IlvManagerLayer layer, boolean visibility) .

♦ To indicate that the layer must be hidden or displayed at a specific moment, use the following filter:

  - `addVisibilityFilter(ilog.views.IlvLayerVisibilityFilter)` IltcLayer.addVisibilityFilter (IlvLayerVisibilityFilter filter)

    See `IlvLayerVisibilityFilter` in the *IBM® ILOG® JViews Java™ API Reference Documentation*.

Note: The visibility of layers in subnetworks cannot be changed. Only layers in the top level of the grapher can be made invisible.

To control the visibility of an individual `IlpObject`, use the methods `addObject(ilog.cpl.model.IlpObject)` or `removeObject(java.lang.Object, boolean)` on `IlpMutableDataSource`. You can also use filters, or assign accepted or excluded classes to the component adapter (see *Filtering* for details).

To control the visibility of an individual `IlpRepresentationObject` use the methods `addRootObject(ilog.cpl.model.IlpRepresentationObject)` or `removeRootObject(ilog.cpl.model.IlpRepresentationObject)` on `IlpMutableNetworkModel`. For objects belonging to a container, use `addChild(ilog.cpl.model.container.IlpRepresentationNode)` or `removeChild(ilog.cpl.model.container.IlpRepresentationNode)`.

Zooming

JViews TGO supports three different zooming modes, as described below:

- **Physical zoom**. In this display mode, the sizes and coordinates of all objects change proportionally with the zoom factor. JViews TGO supports this zoom mode only for zoom factors less than one, that is, for zoom-out, not zoom-in. This mode is implemented by the `IltPhysicalZoomPolicy` class.

- **Logical zoom**. In this display mode, the following changes occur: the coordinates of network elements change proportionally but their sizes remain constant; the sizes of groups change proportionally; the link layout is recalculated according to the new coordinates and sizes. The application can define additional actions in the framework of logical zooming, for example groups being replaced with subnetworks when the user zooms in. This mode is implemented by the `IltLogicalZoomPolicy` class.

- **Mixed zoom**. In this display mode, physical zoom is used for zoom factors less than one, that is, for zoom-out, and logical zoom is used for zoom-in. The `IltMixedZoomPolicy` class implements this mode.

To summarize, physical zoom provides a fast miniaturized view of a network, whereas logical zoom keeps the graphical quality of the displayed objects.

Zoom support is implemented by the `IlpZoomPolicy` instances. By default, an `IlpNetworkView` instance has the physical zoom support set. However, the user can modify this configuration through the following method:

- `IlpNetworkView.setZoomPolicy(ilog.cpl.graphic.views.IlpZoomPolicy)` installs the specified zoom policy in the given view.

- `IlpNetworkView.setZoomPolicy(ilog.cpl.graphic.views.IlpZoomPolicy)` with parameter set to null, uninstalls the zoom policy from the view.

The two coordinate systems

This section assumes that you are familiar with the IBM® ILOG® JViews class `IlvTransformer` and with the difference between manager coordinates and view coordinates (see the method `getTransformer()` in the class `IlvManagerView`).

Because of the possibility of a logical zoom (where the coordinates of the objects are changed according to the zoom level), there are two coordinate systems in use in JViews TGO:

- **Stationary coordinates**. Coordinates passed in this coordinate system do not change over time. This is the coordinate system used to set the position of an object through the method `setPosition(ilog.cpl.model.IlpRepresentationObject, ilog.cpl.graphic.IlpPosition, ilog.cpl.graphic.IlpPositionSource).`

- **View coordinates**. Coordinates given in this coordinate system change when the transformer of the view changes; for example, in the case of a scrollable view, when the user moves one of the scrollbars. This is the coordinate system seen by the layout optimizers and by the connection to the IBM® ILOG® JViews graph layout.

Depending on the method you use, it may be necessary to convert from one coordinate system to another.

To convert from stationary coordinates to view coordinates, apply `network.getManagerView().getTransformer()`. This rule holds good only for objects in the top-level network. It cannot be applied to subnetworks.

To convert back, apply the inverse transformer. (See the methods `inverse(ilog.views.IlvRect)` and `computeInverse(ilog.views.IlvTransformer)` in the class `IlvTransformer`.)

Physical zoom

Graphic representations of telecom objects make intensive use of labels and icons. Therefore, some graphic objects cannot be resized without deteriorating their aspect when the physical zoom mechanism is used. This effect is even more visible when unzooming. While some JViews TGO objects, like groups, can be resized, other objects, such as network elements, have been intentionally designed to have an optimal size depending on the quantity of information they hold. Resizing these objects impairs the readability and compactness of their graphic representation.

For these reasons, the physical zoom mode was implemented to hide the decorations of telecom objects according to a certain configurable zoom factor. This factor is called the *visibility threshold* and represents the absolute value of the determinant of the view transformer. When the value of this determinant is lower than the decoration visibility threshold, decorations of the affected type are no longer displayed.

The visibility threshold for each decoration type can be configured locally to a network component through the methods:

- ♦ `IltPhysicalZoomPolicy.setDecorationNames(java.lang.String[])`
- ♦ `IltPhysicalZoomPolicy. setVisibilityThresholds(int, double)`

or through the CSS properties:

- ♦ `decorationNames`
- ♦ `visibilityThresholds`

How to set the visibility threshold for decorations in a specific network component

The following example describes a network component configuration that sets a physical zoom policy to the component, and defines visibility thresholds for the decorations `Name`, `AlarmBalloon`, `AlarmCount` and `Plinth`. Refer to *Configuring a network component through a CSS file* for more information.

```
Zooming {
  type: "Physical";
  decorationNames[0]: Name;
  decorationNames[1]: AlarmBalloon;
  decorationNames[2]: AlarmCount;
  decorationNames[3]: Plinth;
  visibilityThresholds[0]: 0.5;
  visibilityThresholds[1]: 0.8;
  visibilityThresholds[2]: 0.5;
  visibilityThresholds[3]: 0.5;
}
```

Visibility thresholds can also be configured globally for all network components through the methods:

- `IltrZoom. GetVisibilityThreshold(ilog.tgo.graphic.IltGraphicElementName)`
- `IltrZoom. SetVisibilityThreshold(ilog.tgo.graphic.IltGraphicElementName, double)`. This method sets the value of the visibility threshold for the specified element name. This method defines, for a specific type of decoration, the threshold above which the decoration will disappear when the view is zoomed.

How to set the visibility threshold of decorations for all network components

The following example shows how you can customize the visibility threshold of specific decorations globally, so that all network components created in the application have the same configuration:

```
IltrZoom.SetVisibilityThreshold (IltGraphicElementName.Name, 0.5);
IltrZoom.SetVisibilityThreshold (IltGraphicElementName.AlarmBalloon, 0.8);
IltrZoom.SetVisibilityThreshold (IltGraphicElementName.AlarmCount, 0.5);
IltrZoom.SetVisibilityThreshold (IltGraphicElementName.Plinth, 0.5);
```

Logical zoom

The logical zoom effectively transforms the proportional zoom mechanism of IBM® ILOG® JViews in such a way that the coordinates of JViews TGO objects in the manager are modified when the zoom factor changes. The effect of this operation is that network elements are not resized and the layout of the links is recalculated to correspond to the new coordinates of the nodes in the manager.

Only the main view of an `IlpNetwork` can hold the logical zoom support. The overview window always has a physical zoom.

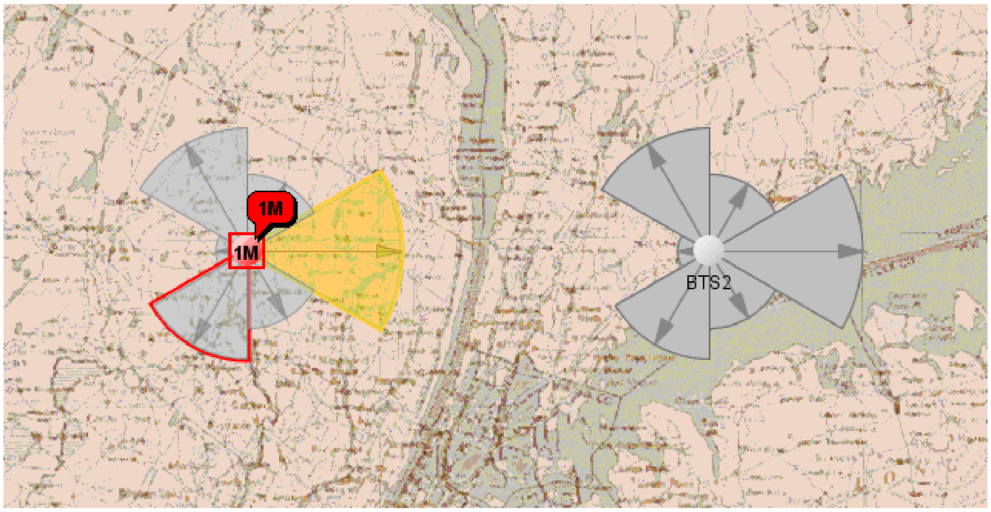
Combining physical and logical zoom (mixed zoom)

JViews TGO provides a mechanism that combines the physical and logical zoom policies. This mechanism is implemented by the class `IltMixedZoomPolicy` and uses physical zoom for zoom factors less than one and logical zoom for zoom factors greater than one, in the same view.

You can configure the zoom policy in the CSS file through the `zooming` property. For more information, see *The Zooming rule*.

Background support

IBM® ILOG® JViews TGO allows you to develop telecommunication user interfaces that display telecommunication elements on a geographic or non-geographic background.



Telecommunication objects on Top of a Geographic Background

The background API

The background API allows you to integrate various types of background in the network and equipment components. It is made up of the following classes (see *Background class relationships* and *Background support classes* for an illustration of the class relationships).

IlpBackground

Backgrounds are implementations of the `IlpBackground` interface. This interface is part of the `ilog.cpl.graph.background` package where all background classes reside.

An `IlpBackground` implementation is usually associated with a specific background format. JViews TGO provides implementations of this interface that cover the most used background formats. See the table below for a complete list:

Supported background formats

Background format	Background class	Background file extension
Scalable Vector Graphics	`IlpSVGBackground`	SVG
GZIP Scalable Vector Graphics	`IlpSVGZBackground`	SVGZ
Raster Images	`IlpImageBackground`	GIF, PNG, JPEG and JPG
Tiled Raster Images	`IlpImageTileBackground`	GIF, PNG, JPEG and JPG
ESRI Shape	`IlpShapeBackground`	SHP, DBF, SHX and IDX
MID/MIF	`IlpMIDMIFBackground`	MIF and MID
JViews Vector Graphics	`IlpIVLFrameworkBackground`	IVL and ILV
JViews Vector Graphics	`IlpIVLMapBackground`	IVL with Map Themes

If the background format you are interested in is not listed here, please see *Advanced support*.

The `IlpBackground` interface defines two key methods: `create` and `dispose`. The method `create` produces the representation of the background itself; the method `dispose` disposes of the constructs that compose the representation of the background.

The ultimate representation of an `IlpBackground` is made up of instances of `IlvGraphic` objects that are part of the IBM® ILOG® JViews Framework. These graphics typically reside on one or more instances of `IlvManagerLayer` which compose the actual background representation. An `IlpBackground` implementation is responsible for appropriately processing the data in a given background format, creating `IlvGraphic` instances that appropriately represent the background data and populating one or several `IlvManagerLayer` instances with these `IlvGraphic` instances. Each `IlvManagerLayer` instances is accessible through the `IlpBackground.getManagerLayer` method.

IlpAbstractBackground

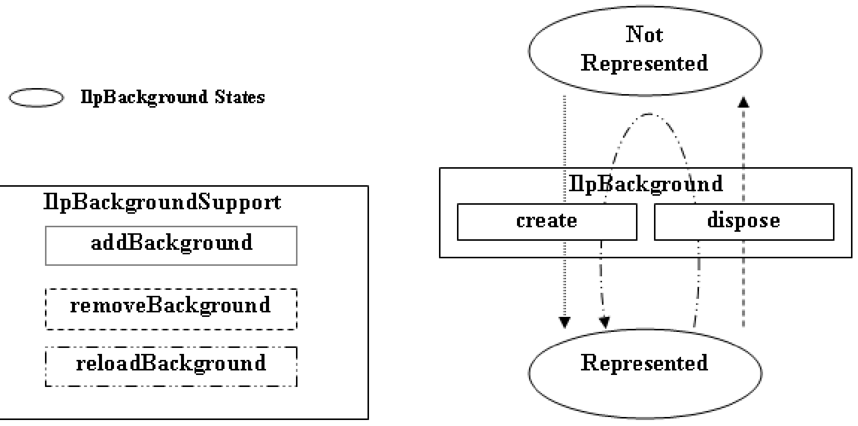
The `IlpAbstractBackground` class is the recommended base class that implements the common methods of the `IlpBackground` that should be used when introducing new `IlpBackground` types. It implements the `IlvBatchable` interface in order to minimize the performance side effects of property changes in a given background instance. For more information on how to use this interface, see the `IlvBatchable` and `IlpAbstractBackground` API.

IlpBackgroundSupport

Instances of `IlpBackground` are managed by an implementation of the `IlpBackgroundSupport` interface. JViews TGO provides a predefined implementation that is used by default, namely the `IlpDefaultBackgroundSupport`.

The `IlpBackgroundSupport` is in charge of providing all background-related functionality that graphic components like `IlpNetwork` and `IlpEquipment` may need. For example, it allows you to add, remove, move and reload backgrounds as well as access the added backgrounds and their constructs.

`IlpBackgroundSupport` is also the entity that handles the lifecycle of `IlpBackground` instances. It determines when the graphical representation of `IlpBackground` instances is created or disposed of. The following diagram illustrates the possible states and interactions involved when switching between them:



The following table summarizes the interactions where the `IlpBackground` API is triggered by the `IlpBackgroundSupport`:

Interactions between IlpBackground and IlpBackgroundSupport

IlpBackground method	Invoked
`create`	♦ when a background is added to the graphic component ♦ when a background is reloaded in the graphic component (preceded by a call to dispose)
`dispose`	♦ when a background is removed from the graphic component ♦ when a background is reloaded in the graphic component (followed by a call to create)

Although not optimal, it is nonetheless legal to use a given implementation of `IlpBackgroundSupport.moveBackground` to remove, then add again a given `IlpBackground` at the appropriate index in order to move a background. For information on the default implementation of this method, see `IlpDefaultBackgroundSupport.moveBackground`.

IlpMapDataSourceBackground

The `IlpMapDataSourceBackground` is an interface that allows you to integrate additional background formats provided in IBM® ILOG® JViews Maps via its Map DataSource API. It extends the `IlpBackground` interface by defining two additional methods that are necessary to establish the integration with JViews TGO: `IlpMapDataSourceBackground.createMapDataSource` to create the `IlvMapDataSource` that will handle the background file and IlpMapDataSourceBackground.getMapDataSource to provide access to the IlvMapDataSource of the background.

See *Limitations* for limitations related to the functionality provided by the IlpMapDataSourceBackground.

IlpAbstractMapDataSourceBackground

JViews TGO provides an abstract base class implementation of the IlpMapDataSourceBackground interface that allows the integration of new IlvMapDataSource implementations to take place with minor effort. This class is called IlpAbstractMapDataSourceBackground.

This class handles all the logistics involved in integrating IlvMapDataSource-based backgrounds within JViews TGO. It leaves as abstract the IlpMapDataSourceBackground.createMapDataSource which must be implemented by the concrete type. It introduces a new method, IlpAbstractMapDataSourceBackground.createRenderer, which returns an IlvFeatureRenderer that can be used to install a custom feature renderer to be used during the creation of the IlvGraphic instances for the provided IlvMapDataSource.

In addition, this type has a utility method, getMapStyle(), which provides access to the IlvMapStyle used by the underlying IlvMapDataSource.

This type is naturally the recommended base type for integrating new implementations of background formats that use the JViews Maps IlvMapDataSource API.

IlpAbstractIVLBackground

JViews TGO uses implementations of IlpAbstractIVLBackground to integrate backgrounds defined in IVL files. Besides the standard IlpBackground functionality, this type also allows users to add and remove IlvManagerLayer instances directly to and from the IlpAbstractIVLBackground instance through the IlpAbstractIVLBackground.addManagerLayer and IlpAbstractIVLBackground.removeManagerLayer, respectively.

The added IlvManagerLayer instances are treated just like another layer that was originated from the source IVL file, meaning that the background properties are propagated to these layers. Thus, the properties of the IlpBackground (like visibility) are applied to the added layers as the state of the IlpAbstractIVLBackground changes.

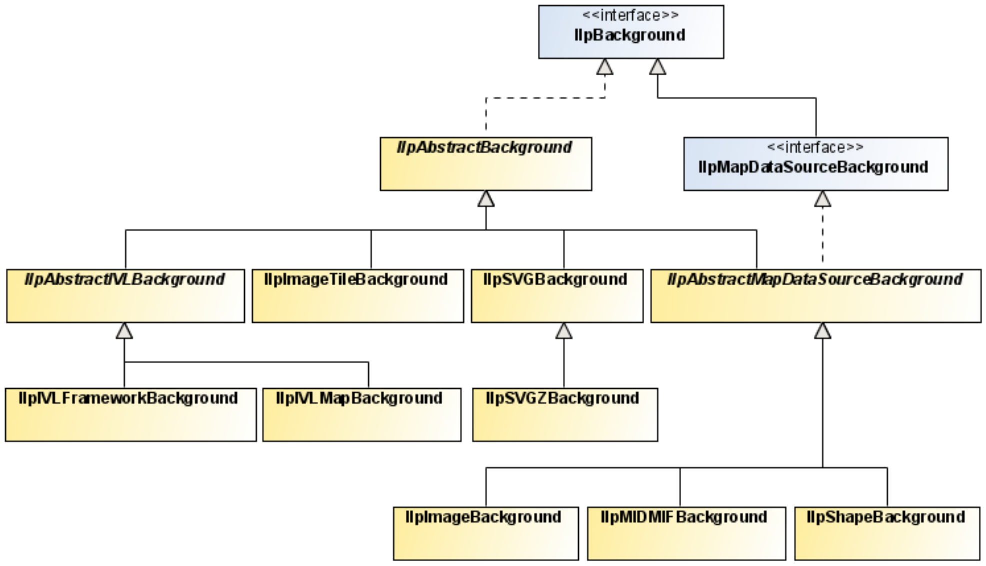
See *Limitations* for limitations related to the functionality provided by the IlpAbstractIVLBackground implementations.

There are two implementations of IlpAbstractIVLBackground: IlpIVLFrameworkBackground and IlpIVLMapBackground.

IlpIVLFrameworkBackground should be used to read standard IVL files that contain *only* JViews Framework content. For more information, see IlpIVLFrameworkBackground in the *Java™ API Reference Documentation*.

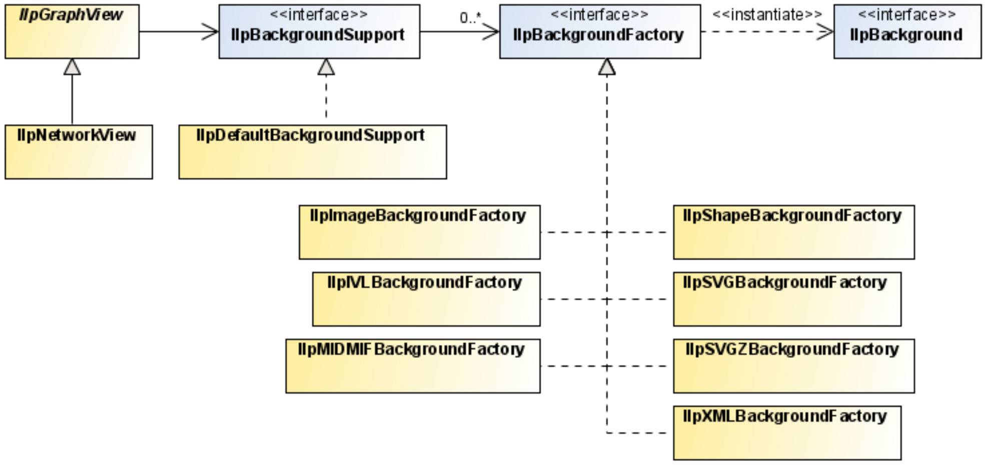
IlpIVLMapBackground should be used to read IVL files that contain JViews Maps content. For more information, see IlpIVLMapBackground in the *Java API Reference Documentation*.

Background class relationships illustrates the background classes.



Background class relationships

Background support classes illustrates the background support classes.



Background support classes

Configuring the background

Backgrounds can be configured at two different levels:

- the component level
- the individual background level

1. Component backgrounds

   As described earlier, you can use the `IlpBackgroundSupport` interface to manage backgrounds programmatically. You can add, remove, reload and access backgrounds. See *How to add a background to the network component* for a sample on how to add a background to the network component.

   You can also specify the precise background configuration through CSS. For more details, see the CSS configuration of backgrounds in *The Backgrounds rule*.

2. Individual background

   Each `IlpBackground` instance has a set of predefined properties that can be retrieved or set at runtime through the methods `IlpBackground.getProperty` or `IlpBackground.setProperty`. Each `IlpBackground` implementation defines the properties that are available to customize its behavior and representation. See the `IlpBackground` interface for general background properties.

   You can also specify the precise properties for a given background through CSS. For more details, see the CSS configuration of backgrounds in *The Backgrounds rule*.

   The following table lists the properties that are available and supported by each `IlpBackground` implementation:

IlpBackground properties

Name	Type	Default	Sample	Supported backgrounds	Description
`url`	String	`null`	`url:”sf-bayarea.png”;`	ALL	Defines the URL of the file that contains the background. This is a read-only property.
`visible`	boolean	`true`	`visible: ”true”`	ALL	Determines whether the background is visible or not.
`loadOnDemand`	boolean	`false`	`loadOnDemand: “false”`	-Shape (shp) -Image (gif, png and jpg)	Determines whether the background uses load-on-demand or not.
`threaded`	boolean	`false`	`threaded: ”true”`	-Image (gif, png and jpg) -Image Tile (gif, png and jpg)	Determines whether the internal processing of the background uses a multithreaded approach to improve performance.
`tileHeight`	integer	`300`	`tileHeight: “100”`	Image (gif, png and jpg)	Determines the height, in pixels, of the tile to be created. This property is taken into account only when the

Name	Type	Default	Sample	Supported backgrounds	Description
					`loadOnDemand` property is set to `true`.
`tileWidth`	integer	`300`	`tileWidth: “100”`	Image (gif, png and jpg)	Determines the width, in pixels, of the tile to be created. This property is taken into account only when the `loadOnDemand` property is set to `true`.
`mapThemed`	boolean	`true`	`mapThemed: ”true”`	JViews Vector Graphics (ivl)	Determines whether the provided IVL file contains Map Themes.

Map themes

The background support provided by JViews TGO has become more interactive. Users can now specify a Map Theme to be associated with backgrounds.

A Map Theme is composed of several background-related features such as, but not limited to:

♦ Map Styles - Allows to modify the background graphical representation according to map scale.

♦ Areas of Interest - Bookmarks areas in the view that are of interest.

♦ Coordinate System - The coordinate system that matches the background map.

♦ Display Preferences - Preferences that affect the display of cartographic data as backgrounds and background-related beans.

♦ Map Labeling - Allows the labeling of background data.

These features are provided by the underlying JViews Maps framework and exposed in JViews TGO. You can find more information on each of these features in the *JViews Maps Documentation, Using the Map Builder,* section *Map Themes and Zoom Levels*.

Integration

Map Themes integration into JViews TGO is available through the use of IVL background files generated from the JViews Maps Map Builder (more specifically through the use of `IlpIVLMapBackground`).

The typical steps for integrating Map Themes created in the Map Builder are:

1. Load the background formats of interest.
2. Edit the various Map features of interest (Map Theme).
3. Save the configured background and its Map Theme as an IVL file.
4. Use this IVL file as a standard background within JViews TGO:

```
Backgrounds {
 background[0]: @+background0;
}
Subobject#background0 {
 class: "ilog.cpl.graph.background.css.IlpBackgroundCSSConfiguration";
 url: "background/backgroundWithMapTheme.ivl";
 mapThemed : true;
}
```

Note: After editing the background and its Map Theme in the Map Builder, you can also save the Map Theme **only** in an IVL file. Then you can use the created IVL Map Theme file (which does not contain the background itself) as a standalone background in JViews TGO.

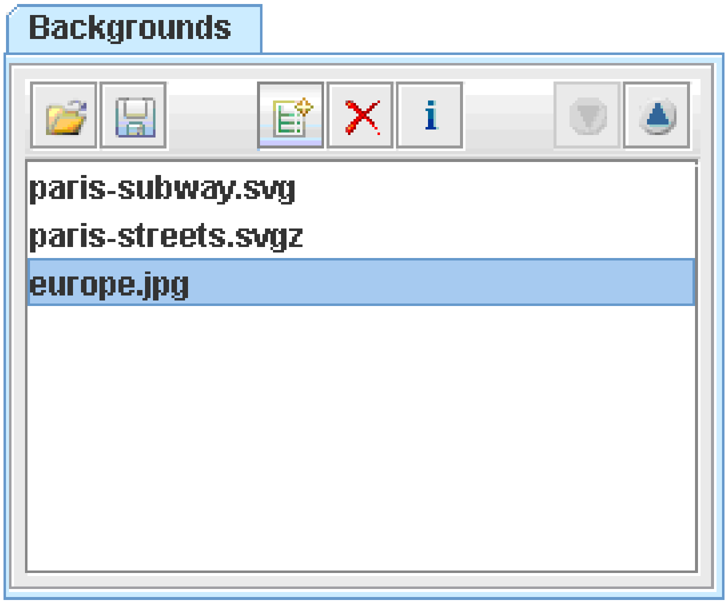
For more information on the JViews Maps Map Builder, see *Using the Map Builder* in the *JViews Maps documentation*.

See *Limitations* for limitations of the Map Theme functionality.

Background beans

IlpBackgroundPanel

This bean allows you to integrate into your user interface the ability to load, reorder, and save backgrounds for an `IlpNetwork` or `IlpEquipment`.



paris-subway.svg
paris-streets.svgz
europe.jpg

IlpBackgroundPanel Bean

A background located at the top has higher priority (drawing-wise) than the background below it. In the figure above, the `europe.jpg` background has the lowest priority of all and will be drawn below all other backgrounds. Whereas `paris-subway.svg` will be drawn on top of all backgrounds (highest priority).

The `IlpBackgroundPanel` bean provides the following features:

📂	Load a CSS file that contains a background configuration
💾	Save the current background configuration
📝	Add a background to the current configuration
✕	Remove the selected background from the current configuration
ℹ	Provide more information on the selected background
▲	Move the selected background up
▼	Move the selected background down

You can customize the background files that are filtered by this bean, by setting the `getBackgroundExtensions` method. You can also specify the default directory where it looks for backgrounds, by using the `setDefaultDirectory` method. Lastly, you can show or hide both the Add Background and the Remove Background buttons at runtime by using the `showAddBackgroundButton` and `showRemoveBackgroundButton` property accessors of `IlpBackgroundPanel`.

See the `IlpBackgroundPanel` Java API for additional information.

Some quick facts

- ♦ When the view is zoomed, the background map is also zoomed.
- ♦ The objects and the background map do not change positions during zoom operations.
- ♦ Backgrounds are stacked according to their indices, where the background at index 0 is the bottommost background, that is, it has the lowest priority drawing-wise. However, keep in mind that index i does not necessarily correspond to the `IlvManagerLayer` index i, because some backgrounds span over more than one layer.

Advanced support

JViews TGO provides advanced support of the following:

- ♦ memory management
- ♦ configuring backgrounds through XML

♦ integration of background formats not supported natively

Memory management

Some of the `IlpBackground` implementations allow you to configure the policy used to handle the management of the resources needed to represent its format.

This is the case in particular with the `IlpImageBackground` which leverages the advanced performance features provided by JViews Maps. More specifically, it takes advantage of the `IlvRasterMappedBuffer` which allows you to specify the memory management policy used to handle the rasters that ultimately represent the backgrounds.

By default, JViews TGO enforces the *in-memory* policy which stores the resources in memory. But you can also take advantage of the *disk-mapped* policy which stores pixel information on disk-mapped memory.

Important: The disk-mapped policy is not supported in applets.

For more details on this topic, see:

♦ The *IBM® ILOG® JViews Maps Documentation*, *Programming with JViews Maps*, section *Raster Image Management*.

♦ The `IlvRasterMappedBuffer` Java API.

XML background format

In addition to the natively supported background formats (see *Supported background formats*), JViews TGO also provides the ability to configure backgrounds via XML.

The XML format allows you to configure one or more background files and specify some of the properties that each background uses to configure itself. This format is typically used if you need an image tile background and need to specify its configuration statically. Image tile backgrounds consist of a rectangular array of JPG, GIF, or PNG files, each with a filename denoting the column number and the row number. For a sample of how to make use of this format and configure an image tile background, see *How to specify a tiled image background using the XML format*.

Note: The XML format does not support all the parameters, such as offsets and built-in projections, that a given background may internally use.

Unless you have specific needs, the recommended way to configure your backgrounds is through CSS which allows nearly any type of customization.

Integration of unsupported background formats

If you want to use of a background format that is not supported natively by JViews TGO (for example, TIGER Line maps), the recommended approach is to check if this format is supported by the underlying JViews Maps product that JViews TGO makes use of. If this map format is supported by JViews Maps, then you should follow these steps:

1. Use the JViews Maps Map Builder to load and customize the map as needed.
2. Export the map as an IVL file.
3. Use this IVL file as an IVL background file that contains JViews Maps content in JViews TGO, as described in *Integration*.

For more information on the JViews Maps Map Builder, see *Using the Map Builder* in the *JViews Maps documentation*.

If the map format is supported neither by JViews TGO nor by JViews Maps, then it is recommended to export the unsupported format into one of the formats supported by JViews Framework, JViews Maps or JViews TGO, so that the standard or above integration can take place.

Limitations

The following table lists the limitations of backgrounds in JViews TGO:

Background limitations

Area	Description
Map Themes - Multiple backgrounds	Loading a background that contains a map theme may overwrite some or all of the map theme settings of a previously loaded background. The recommended approach will be to: 1. Load all the backgrounds of interest in the JViews Maps Map Builder. 2. Customize the map theme of all these backgrounds. 3. Save the map theme as a whole (as opposed to individually). 4. Load the global map theme as a standard IVL background.
`IlpMapDataSourceBackground`	Access to the `IlvCoordinateSystem` of an `IlvMapDataSource` from an `IlpMapDataSourceBackground` is **read-only**. If you set the coordinate system of the `IlvMapDataSource` for a given `IlpMapDataSourceBackground` through the method `IlvMapDataSource.setCoordinateSystem()`, it will not take effect because the `IlvMapDataSource` is recreated during a background reload. If you need to customize the map data source with a custom `IlvCoordinateSystem`, the recommended approach is to overwrite the `IlpAbstractMapDataSource.createDataSource` method to return an `IlvMapDataSource` that already has the `IlvCoordinateSystem` of interest set on it.
`IlpAbstractIVLBackground`	When you add `IlvManagerLayer` instances manually to an `IlpAbstractIVLBackground`, these instances will **not** be restored during a background reload. If reload support is required, it is recommended to create an additional `IlpAbstractIVLBackground` that points to an IVL file containing the additional graphics, or to add again the additional

Area	Description
	`IlvManagerLayer` instances after the background has been reloaded.

How to add a background map in MIF format

```
URL url = context.getURLAccessService().getFileLocation("world.mif");
networkComponent.addBackgroundURL(url);
```

How to add a background to the network component

```
IlpNetwok network = …;

URL backgroundURL = context.getURLAccessService().getFileLocation("backgrounds/
world.png");
```

How to specify a tiled image background using the XML format

```
<?xml version="1.0" encoding="UTF-8"?>
<background
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="ilog/cpl/schema/background.xsd"
  type="ImageTile">
 <property name="pattern">jpp%c%r.jpg</property>
 <property name="tileWidth">400</property>
 <property name="tileHeight"400</property>
</background>
```

How to add an IVL background to the underlying IlvManager

The appropriate approach is to use an `IlpAbstractIVLBackground`, which gives you direct access to the underlying `IlvManager` that hosts the network (or equipment) component. If this approach does not satisfy your needs, you can (although this is not advised):

1. Obtain the `IlpNetwork` or (`IlpEquipment`) underlying `IlvManager` by calling `network.getView().getManagerView().getManager()`

2. Make sure that the needed number of `IlvManagerLayer` instances is inserted in this `IlvManager` (so that your background does not corrupt any of the existing `IlvManagerLayer` instances).

   You can add more `IlvManagerLayer` instances by calling `IlvManager.addLayer(0)`

3. Call `IlvManager.read` to read the IVL file.

How to find out the georeferencing configuration needed for my IlpGeographicPositionconverter

If your background is georeferenced, you will likely be using several advanced Map background settings that are required when configuring the `IlpGeographicPositionConverter`.

These settings can typically be found in the `IlvCoordinateSystem` associated with a given background.

♦ If you use an `IlpMapDataSourceBackground`, you can access its `IlvMapDataSource` and from there you can find the `IlvCoordinateSystem`.

♦ If you use a non `IlpMapDataSourceBackground`, the information is typically configured manually by setting it explicitly on the underlying `IlvManager` or indirectly through a Map Theme (in an IVL file). Either way, you can access the `IlvCoordinateSystem` through the call `IlvCoordinateSystemProperty.getCoordinateSystem()`.

Note that if you have different coordinate systems for which you want to find out an `IlvMathTransform`, you can use an `IlvCoordinateTransformation`.

Lastly, note that the different types of `IlvCoordinateSystem` provide different types of settings. For example, an `IlvProjectedCoordinateSystem` provides access to a possibly needed `IlvProjection`. So assuming that the background map has an `IlvProjectedCoordinateSystem`, to find out the `IlvProjection` used by it, you can do the following:

```
IlpMapDatasourceBackground background = …;
IlvCoordinateSystem coordinateSystem =
    background.getMapDataSource().getCoordinateSystem();
if(coordinateSystem instanceof IlvProjectedCoordinateSystem) {
    IlvProjectedCoordinateSystem projectedCoordinateSystem =
        (IlvProjectedCoordinateSystem)coordinateSystem;
IlvProjection projection = projectedCoordinateSystem.getProjection()
…
}
```

Filtering

The network component allows you to filter the nodes that are displayed. To do so, attach an instance of `IlpFilter` to the network component by using the method `setFilter`. The `accept()` method of the filter object will be invoked whenever the network is prompted to display an `IlpObject`. If the method returns `false`, the object will not be shown in the network. In the same way, an object will not be shown if its parent is not displayed.

For example, write the following code to show only objects of the class `IltNetworkElement`.

How to filter objects to be shown in the network component

```
IlpNetwork network = // ...
// Create a new IlpFilter instance
IlpFilter filter = new IlpFilter(){
  // This method is called for every object in the data source
  public boolean accept (Object object){
    IlpObject ilpObject = (IlpObject)object;
    IlpClass clz = ilpObject.getIlpClass();
    // Check if the class == IltNetworkElement
    return clz.equals(IltNetworkElement.GetIlpClass());
  }
// Set the filter to the network
network.setFilter(filter);
```

All the objects are refiltered whenever a new filter is set. If the filter is null (which is the default), all the objects under the root nodes will be displayed.

To retrieve the active filter, use the method `getFilter()`.

Note: The filtering takes actually place at the adapter level.

To see how to configure filtering through CSS, refer to *The Adapter rule*.

Accepted and excluded classes

You can specify the business objects that will be represented or not in the network component depending on their business classes. To do so, you need to specify the business classes to be accepted or excluded using methods `setAcceptedClasses` or `setExcludedClasses` in the network component adapter. To retrieve the adapter, use the `getAdapter` method. The adapter must be an instance of a subclass of `IlpAbstractNodeAdapter`. By default, business objects of the class `IltAlarm` are excluded from the network component, so that alarm objects in the data source are only used to compute the alarms present in a given managed entity instead of being graphically represented in the view.

How to specify excluded classes in the network component

You can specify that business objects from specific business classes are not represented in the network component. You can do that using the API, `setExcludedClasses(java.util.List)` method, or using CSS.

The following example shows you how to prevent objects from business classes `IltAlarm` and `IltLed` to be represented:

```
Adapter {
  excludedClasses[0]: "ilog.tgo.model.IltAlarm";
  excludedClasses[1]: "ilog.tgo.model.IltLed";
}
```

How to specify an accepted class in the network component

By default, all business classes, except `IltAlarm`, are accepted by the network component. If you want to specify exactly which business classes to represent, you should combine the list of excluded and accepted classes, so that you exclude all business classes except those that are marked in the accepted class list.

In the following example, the network component is configured in a way that it graphically represents only business objects from the class `IltNetworkElement`.

```
Adapter {
  excludedClasses[0]: "ilog.tgo.model.IltAlarm";
  excludedClasses[0]: "ilog.tgo.model.IltObject";
  acceptedClasses[0]: "ilog.tgo.model.IltNetworkElement";
}
```

Note: The filtering that is performed through the use of the accepted and excluded class lists takes actually place at the adapter level.

To see how to configure excluded and accepted classes through CSS, refer to *The Adapter rule*.

Setting a list of origins

By default, all the objects in the data source that do not have a parent are treated as root nodes by the network component. However, you can explicitly select the root nodes to be displayed through the adapter that forms a bridge between the data source and the network component. To retrieve the adapter, use the `getAdapter()` method. The adapter must be an instance of a subclass of `IlpAbstractNodeAdapter`.

The root nodes can be changed by modifying the list of *origins* for the adapter. These origins are set and retrieved as `IlpObject` identifiers. The network adapter has two options: either the origins are represented as root nodes, or they are hidden and their child objects are represented as root nodes.

The method `getOrigins` allows you to get the list of current origins. The method `isShowingOrigin` indicates whether the origins themselves or their child objects are represented as root nodes. By default, the list of origins is empty and the origins are not shown, which means that all objects without a parent are shown as root nodes. Thus, the entire contents of the data source are displayed in the network.

Note: The origins are specified using identifiers, not the `IlpObject` instances. You can retrieve the identifier of an `IlpObject` with the `getIdentifier` method of the object.

To change the list of origins, use the `setOrigins` method. This method takes a list of business object identifiers as its first parameter. Its second parameter is a Boolean flag that indicates whether or not the origins themselves should be shown as root nodes.

Calling this method with an empty list and the second parameter set to `true` empties the network:

```
setOrigins(Collections.EMPTY_LIST, true);
```

Calling the method with an empty list and the second parameter set to `false` restores the default; that is, all the objects in the data source are shown:

```
setOrigins(Collections.EMPTY_LIST, false);
```

How to show an object as the root node of a network

To show only a given `IlpObject` as the root node of a network, use the following code:

```
IlpNetwork network = ....;
IlpObject originObject = .....;
java.util.List originList = new ArrayList();
originList.add(originObject.getIdentifier());
network.getAdapter().setOrigins(originList, true);
```

See the `IlpAbstractHierarchyAdapter` class for additional methods to help you manage origins.

To see how to configure origins through CSS, refer to *The Adapter rule*.

Node factory

The network adapter converts business objects retrieved from the associated data source into instances of `IlpNetworkNode`. The new representation objects are created by a representation object factory. The network adapter uses by default the `IlpDefaultNetworkNodeFactory` that creates representation objects of type `IlpDefaultNetworkNode`.

To know how to configure a network node factory through CSS, refer to *The Adapter rule*.

Link factory

As the network node factory transforms business objects into representation objects, the link factory transforms business objects that are links into representation objects that are instances of `IlpNetworkLink`. The network adapter uses by default the `IlpDefaultNetworkLinkFactory` that creates representation objects of type `IlpDefaultNetworkLink`.

To see how to configure a network link factory through CSS, refer to *The Adapter rule*.

Expansion strategy

The network adapter uses an *expansion strategy* to identify whether objects should be loaded or not in the network model. The expansion strategy defines how an object is going to behave when it is expanded, for example, when the user opens a network node by double-clicking or by using the network expansion handles. The expansion strategy indicates whether load on demand is implemented and provides methods to load and release child nodes.

The network adapter uses an *expansion strategy factory* to decide the expansion strategy to apply to a network node when it is created by the adapter. The default expansion strategy factory implementation, `IlpDefaultNodeExpansionStrategyFactory`, checks the property "`expansion`" of each business object in the cascading style sheet loaded in the component to identify the expansion strategy to use.

The default network expansion strategy factory supports three types of expansion strategies:

♦ `IN_PLACE`: loads the child objects immediately in the network model. In this expansion strategy, nodes are considered as parent nodes only when they have containment relationships defined in the attached data source, through the `IlpContainer` interface. The child objects should already be loaded in the data source and should be visible according to the data source filter, if there is one defined.

♦ `IN_PLACE_MINIMAL_LOADING`: loads the child objects on demand in the network model, that is, as the user expands the parent nodes. All nodes with this expansion strategy are considered as possible parent nodes, and therefore are represented with an expansion icon. If the node does not contain child objects, the expansion icon will disappear when the expansion is executed for the first time.

♦ `NO_EXPANSION`: expansion is not supported by the node.

See Customizing the expansion of business objects in the *Styling* documentation for information on how to customize the business object expansion type, which is defined by the property `expansion`.

The expansion strategy factory can be customized for the adapter either through CSS or through the API.

Architecture of the network component

A graphic component encapsulates a model, a view, and a controller. The network component, like all the other graphic components, is based on the MVC architecture, which means that it has a model, a view and a controller associated with it. For a general introduction to the MVC architecture, see *Architecture of graphic components*. That section describes the classes and features of the network component that are specific to each module of the MVC architecture, and also explains the role of the adapter.

In this section

Class overview

The MVC architecture of the network component is implemented by the following classes:

- The class `IlpNetwork` contains a model, a view, and a controller.
- The model interfaces are `IlpNetworkModel` and `IlpMutableNetworkModel`.

  The interface `IlpMutableNetworkModel` provides API facilities for adding and removing objects. The two network model interfaces are implemented through the class `IlpDefaultNetworkModel`. Instances of `IlpNetwork` use the class `IlpDefaultNetworkModel`.

  Representation objects to be displayed in the component must be added to the model. The network model recognizes the following objects:

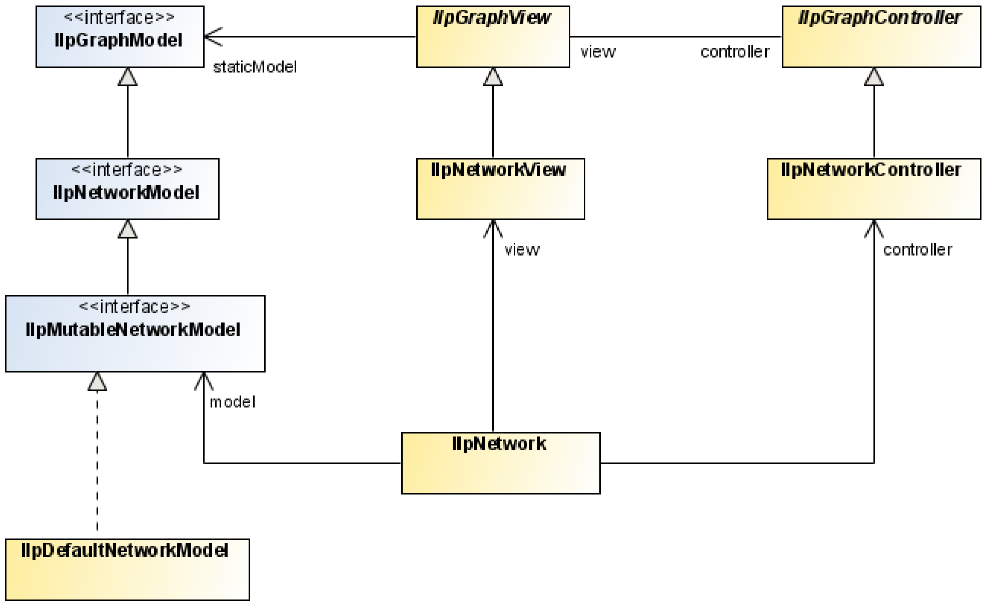
  - Nodes
  - Links, which connect nodes

  Representation objects to be added to the model must implement the `IlpNetworkNode` or `IlpNetworkLink` interface. Concrete implementations of these interfaces are provided as `IlpDefaultNetworkNode` and `IlpDefaultNetworkLink`. You can create subclasses of these representation object classes.
- The class `IlpNetworkView` defines the network view.

  It displays the objects in a rectangular area, with scrollbars and a toolbar to let users navigate. The view displays the objects contained in the model or made accessible through container expansion of objects in the model.
- The class `IlpNetworkController` defines the network controller.

  The controller configures the view interactor, reacts to actions triggered by the interactor, and forwards these actions to a handler.

Main classes used by the network component shows the main classes used by the network component.



Main classes used by the network component

The model

The model contains the representation objects used to produce the graphic objects in the view. The model describes the end points of links, and as such it models the topology of the network. The model allows the view to access contained objects through the method `getChildren(ilog.cpl.model.IlpRepresentationObject)`. The model provides support for load-on-demand; that is, for adding child objects asynchronously when expanding network objects through the class `IlpExpansionStrategy`.

Classes of the network model

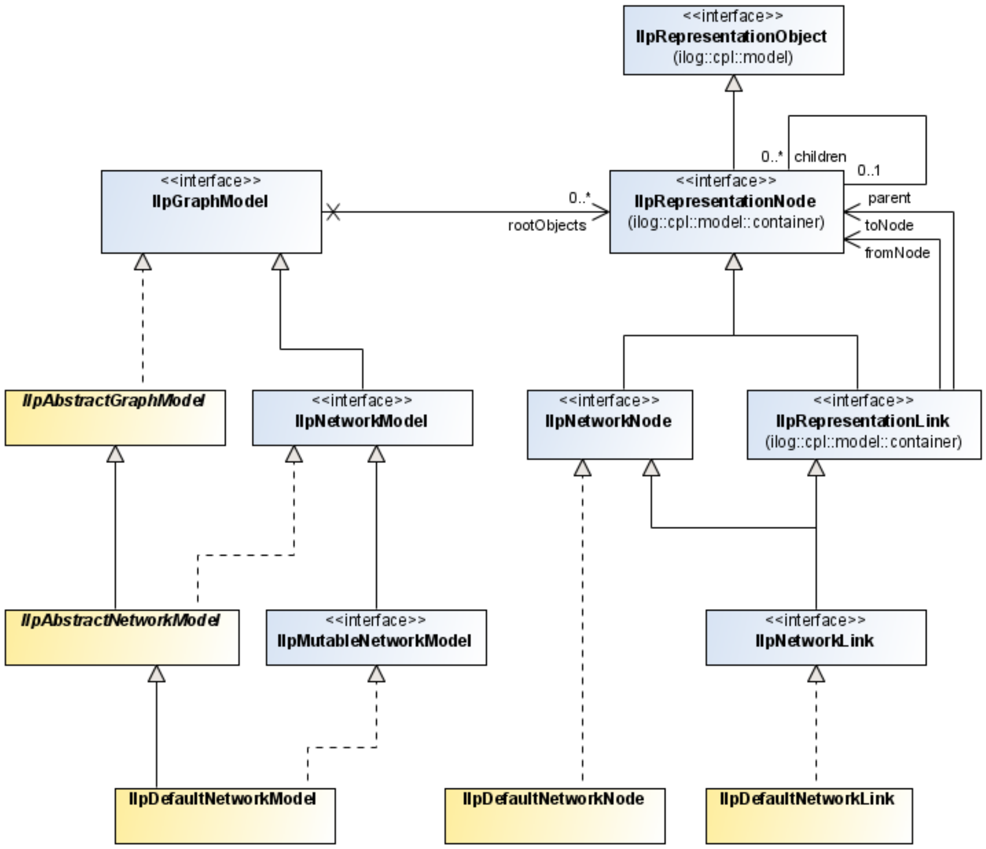
The interface `IlpNetworkModel` is implemented by the class `IlpDefaultNetworkModel`. This class is automatically created when you instantiate `IlpNetwork`. JViews TGO provides the following interfaces for creating nodes and links:

♦ `IlpNetworkNode` for nodes

♦ `IlpNetworkLink` for links

JViews TGO supplies default implementations of these interfaces.

Classes used by the network model shows the classes used by the network model.



Classes used by the network model

Creating nodes and links in the network model

When you want to create a new node or link, use the default implementations:

- `IlpDefaultNetworkNode`
- `IlpDefaultNetworkLink`

To create a linkset, create several links with the same end nodes.

To set the end nodes of a link, apply the methods `setFromNode` (for the start node) and `setToNode` (for the end node).

A link cannot be displayed unless it has end nodes, but you can create the link before you create the end nodes.

The following examples show how to use the classes for creating nodes and links.

How to create a node

```
IlpDefaultNetworkNode node = new IlpDefaultNetworkNode(MyAttributes,null);
network.setPosition(node,new IlpPoint(100,200), IlpPositionSource.BACKEND);
model.addRootObject(node);
```

How to create a link

```
IlpDefaultNetworkLink link = new IlpDefaultNetworkLink(MyAttributes,null);
link.setFromNode(node1);
link.setToNode(node2);
model.addRootObject(link);
```

For convenience, the main methods of `IlpNetworkModel` and `IlpMutableNetworkModel` are also accessible from `IlpNetwork`. Thus, the following statements are equivalent:

```
network.getModel().addRootObject(node);
```

and

```
network.addRootObject(node);
```

The view

The view displays a rectangular area of a theoretically infinite plane area. The view performs the following functions:

- ♦ Displays a subset of the objects of the model
- ♦ Allows navigation using scrollbars and provides a zoom facility
- ♦ Assigns default positions to nodes that have no value for `position` in the model
- ♦ Assigns shapes to links that have none
- ♦ Modifies link shapes when end nodes are moved
- ♦ Provides a toolbar for choosing a view interactor
- ♦ Optionally displays an overview window

Classes of the network view

The `IlpNetwork` class automatically creates the concrete class `IlpNetworkView`, which is provided for developing the network view. This class provides methods that allow you to configure the view in the following ways:

- ♦ Turn the scrollbars on or off: `setHorizontalScrollBarVisible(boolean)`, `setVerticalScrollBarVisible(boolean)`
- ♦ Set the zoom level: `getManagerView()`
- ♦ Set the toolbar on or off (here set to off): `setToolBarVisible(boolean)` (false)
- ♦ Set the overview panel to be displayed or not (here set to be displayed): `setOverviewVisible(boolean)`

For convenience, these methods are also accessible from the class `IlpNetwork`. They delegate to `IlpNetworkView`.

For information on how to configure the network view in CSS, see *Configuring a network component through a CSS file* .

For information on how to configure the network view through the API, see *How to configure the network view with the API* .

Graphic objects in the network component

Graphic objects display as many details as possible within the limits of the display area and without loss of readability.

The network component uses only composite graphic objects. The graphic objects are created by the view renderer, which calls the appropriate object renderers to obtain the composite graphic objects. The view determines which object renderer is required to draw the graphic object that translates a particular representation object or attribute from the style sheet properties.

The following basic variations of graphic object exist in the network component:

♦ Nodes—Nodes are the basic graphic objects and they are represented as network elements according to the conventions of the governing standards, such as ITU-T or ANSI, and the appropriate protocols.

♦ Links—Links are the connections between nodes.

End-user interaction with a graphic object is handled by an object interactor. Object interactors handle the gestures of an end user when performing a task. Gestures consist of one or more mouse events to perform one task.

For more information on object interactors, see *Object interactors*.

Graphic object classes

The graphic representation of each object displayed in the network component is implemented through the `IlpGraphic` interface. JViews TGO provides predefined network graphic objects that are produced by the default network component renderer. You can customize the rendering of the objects through CSS.

For custom business objects, JViews TGO provides a default representation with a set of properties that can be customized to better represent your objects. If you prefer, you can also specify a new graphic representation by defining an `IlvGraphic` class in the CSS. For more information, refer to Customizing user-defined business objects. You can also refer to ***<installDir>*** **/samples/network/compositeGraphic** to see how to create a new object representation by using the IBM® ILOG® JViews composite graphics feature.

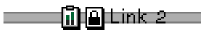
Predefined business objects already have a specific graphic representation that can only be changed through CSS customization by setting the object properties.

You can customize the graphic representation by adding new decorations. To see how to add new decorations to the objects using CSS, look at the decoration sample at ***<installDir>*** **/samples/network/decoration**.

Network graphic object renderers

Object renderers in the network component create one graphic object that translates a complete representation object. No graphic objects are created that correspond to attributes of a representation object. Instead, subcomponents of graphic objects are created from attributes for example, a label can be created from the `name` attribute.

Such subcomponents are combined into composite graphic objects, like the link with secondary states and a label shown in *Link with secondary states and a label* by using attachments.



Link with secondary states and a label

A composite graphic object constructed in this way looks like one object. The different instances of `IlpGraphic` used to build the graphic object cannot be distinguished as separate objects in the network. JViews TGO manages only the composite object. You cannot move the label separately from the link.

The controller

The controller is used for configuring view parameters, including layout and the background map. It manages the toolbar and the interactors: it determines which interactors are available from the toolbar.

The controller manages end-user interaction, such as object creation or move requests. It forwards these requests to the handler (see *The handler*). It forwards all end-user requests to:

- create objects
- delete objects

and actions triggered by the end user to:

- move objects
- reshape objects

Classes of the network controller

The `IlpNetwork` class automatically instantiates the class `IlpNetworkController`.

Using the network controller

The normal way of customizing the behavior of the controller is to attach a handler through the API.

The default handler is the `IlpNetworkHandlerWithoutDataSource`. When a mutable data source is connected to the network component, a new handler, `IlpNetworkHandlerWithDataSource`, is attached to the controller. You can customize either of these handlers by extending them. To set a new handler, use the method `setHandler (ilog.cpl.network.IlpNetworkHandler)`:

```
IlpNetwork network = ...
IlpNetworkController controller = network.getController();
IlpNetworkHandler myHandler = new MyCustomHandler();
controller.setHandler(myHandler);
```

Another controller configuration is to set the default view interactor through the method `setDefaultViewInteractor`. See also *Interacting with the network view*.

You can also customize the controller by using CSS rules in the network configuration. The following code gives an example. For more details, see *Configuring a network component through a CSS file*.

How to customize the network controller through CSS

```
Interactor {
  name: "Select";
}
```

This rule selects the default interactor.

The handler

In the same way as the adapter passes information about the objects in the data source to the network component, the handler passes information in the opposite direction, that is, from the network component to the data source.

The information notified in this way includes:

- Actions triggered by interactors for
  - Creating objects in the data source
  - Removing objects from the data source
  - Changing attributes of objects in the data source
  - Changing the parent object of an object in the data source
  - Expanding and collapsing container objects
- Propagating position changes of objects

  The position changes are usually due to layout, zoom change, or interactors.

The handler has been designed to simplify the customization of user interactions without rewriting the controller.

There are four types of handler:

- `IlpPositionHandler` to handle object position changes.
- `IlpNodeHandler` to handle object additions, removals and updates, as well as relationship changes.
- `IlpLoadHandler` to handle reloading of model objects from an XML file.
- `IlpExpansionHandler` to handle the expansion and collapsing of objects.

The `IlpNetworkHandler` interface indirectly extends all four types of handler.

The handler has a reference to the data source in the form of an `IlpMutableDataSource`, and to the network adapter.

You can customize the behavior of the handler by subclassing the class `IlpNetworkHandlerWithDataSource`. A particular method can be overridden for each of the possible actions.

You can customize the way position changes are propagated by overriding the method `propagatePositionToDataSource(ilog.cpl.model.IlpObject, ilog.cpl.graphic.IlpPositionSource)`. In a typical situation where the client is active, position changes are propagated to the data source. Therefore, this method returns `true` by default. In a situation where the client has read-only access, you may want to allow only user-requested position changes or no position changes at all to be forwarded to the data source. You can achieve this result by allowing the method `propagatePositionToDataSource(ilog.cpl.model.IlpObject, ilog.cpl.graphic.IlpPositionSource)` to return `false` in the appropriate cases.

The handler is most often subclassed to allow you to customize the creation of new objects in the data source. The object interactors may need to be customized in the same way. A customized object creation interactor typically calls the controller method `createObject(java.lang.Class, ilog.cpl.model.IlpAttributeGroup, java.util.Map, ilog.cpl.graphic.IlpPosition)` with specific properties. The controller then forwards these properties to the handler. Finally, the method `handleCreateObject(java.lang.Class, ilog.cpl.model.IlpAttributeGroup, java.util.Map, ilog.cpl.graphic.IlpPosition)` of the handler parses the additional properties and creates the new objects.

By default, the handler creates new objects in two steps:

1. The ID of the new object is created with the method `createObjectId(java.lang.Class, ilog.cpl.model.IlpAttributeGroup, java.util.Map)`.

2. The `IlpObject` corresponding to this ID is created with the method `createObject(java.lang.Class, ilog.cpl.model.IlpAttributeGroup, java.util.Map, java.lang.Object)`.

You can customize each step separately by overriding these methods in a subclass.

Any user interaction with the network is processed by the network controller which delegates action to the network handler. The handler has two default implementations:

♦ `IlpNetworkHandlerWithoutDataSource` is attached by default to the controller and performs user interactions directly in the network model.

♦ `IlpNetworkHandlerWithDataSource` is automatically attached to the controller when a data source is connected with the network component. User interactions are executed inside the data source that will notify the adapter, and the adapter in turn will notify the network model. In this particular use case, changes will be reflected on all network components, if any, connected to the data source.

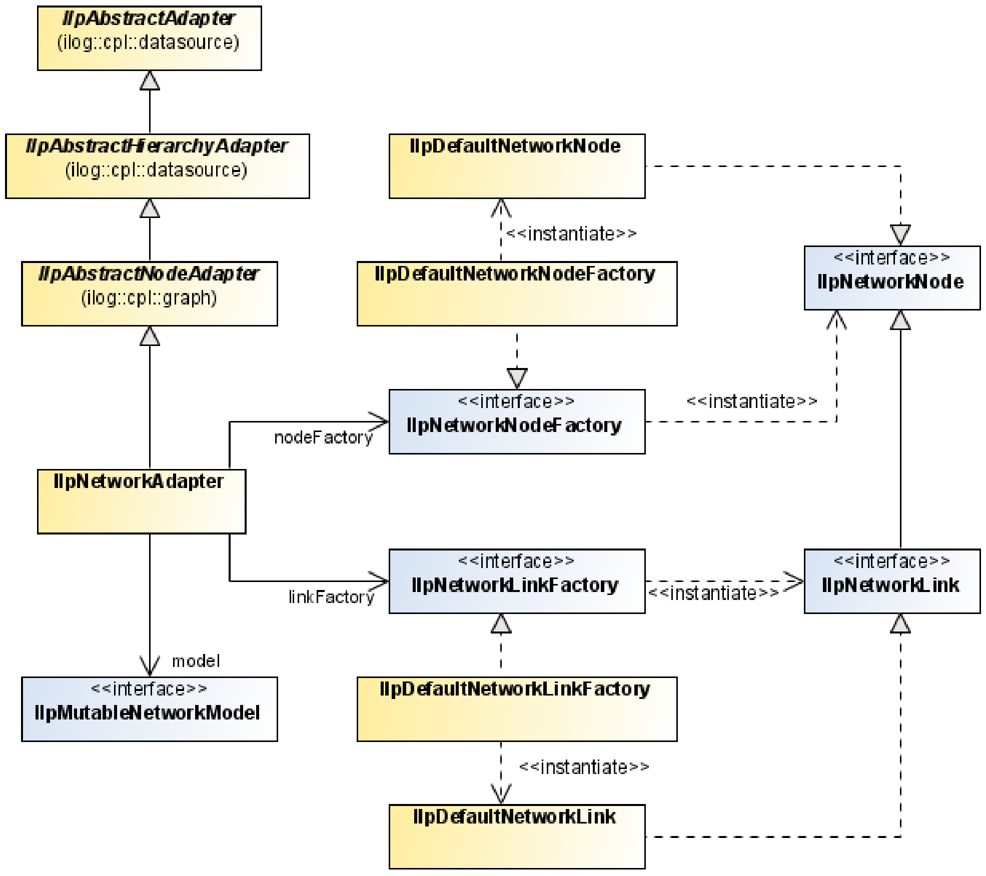
The adapter

The network adapter converts business objects into representation objects of type network node and network link. It is defined by the class `IlpNetworkAdapter`.

Network adapters retrieve structural information (that is, parent/child relationship) about business objects from the associated data source and determine whether an object should appear as a root representation object by examining a list of origins. See *Setting a list of origins* for details.

Like the tree adapter, the network adapter supports load on demand.

The following figure shows network adapter classes:


IlpAbstractAdapter (ilog::cpl::datasource)
IlpAbstractHierarchyAdapter (ilog::cpl::datasource)
IlpAbstractNodeAdapter (ilog::cpl::graph)
IlpNetworkAdapter
IlpDefaultNetworkNode
<<instantiate>>
IlpDefaultNetworkNodeFactory
<<interface>> IlpNetworkNode
<<interface>> IlpNetworkNodeFactory
nodeFactory
<<instantiate>>
<<interface>> IlpNetworkLinkFactory
linkFactory
<<instantiate>>
<<interface>> IlpNetworkLink
model
<<interface>> IlpMutableNetworkModel
IlpDefaultNetworkLinkFactory
<<instantiate>>
IlpDefaultNetworkLink

Network adapter classes

The network adapter creates either a node or a link representation object depending on the value returned by the method `getLinkInterface(java.lang.Object)`. If the return value is not null, then a link is created; otherwise, a node is created.

Nodes are created with an `IlpNetworkNodeFactory`. By default, this factory is an instance of the class `IlpDefaultNetworkNodeFactory`, which creates `IlpDefaultNetworkNode` instances.

Similarly, links are created with an `IlpNetworkLinkFactory`. By default, this factory is an instance of the class `IlpDefaultNetworkLinkFactory`, which creates `IlpDefaultNetworkLink` instances.

The network adapter handles the position or shape of objects. By default, it interprets every object attribute with the name `position` as denoting the position of the object. You can use the method `setPositionAttribute(ilog.cpl.model.IlpClass, ilog.cpl.model.IlpAttribute)` in `IlpAbstractNodeAdapter` to specify any other attribute to be used instead for all instances of a given `IlpClass`.

You can create a network adapter implicitly by instantiating the `IlpNetwork` component as shown in the following example.

How to create a network adapter by instantiating a network component

```
IlpNetwork ilpNetwork = new IlpNetwork();
IlpDataSource dataSource = new IlpDefaultDataSource();
ilpNetwork.setDataSource(dataSource);
```

If you want to configure the adapter, to set its origin for example, you must first retrieve it from the network component and then set it to the data source.

How to configure a network adapter

```
IlpNetwork ilpNetwork = new IlpNetwork();
IlpDataSource dataSource = new IlpDefaultDataSource();
// configure the adapter, for example set an origin
IlpNetworkAdapter adapter = ilpNetwork.getAdapter();
adapter.setOrigins(Collections.singletonList("origin"),false);
adapter.setDataSource(dataSource);
```

For information on how to configure the adapter in CSS, see *The Adapter rule*.

Controlling the display of objects as containers

Unlike the tree adapter, the network adapter considers by default that objects are not containers. There are two conditions for an object to be a container: its `getContainerInterface(java.lang.Object)` method should not return `null` and the property `expansion` applying to that object should be set to `ExpansionType.IN_PLACE`. By default, this property is set to `ExpansionType.NO_EXPANSION`. For information on how to set a property value, see Introducing cascading style sheets.

Note: There is no way to represent an object in a network component as a container, if its `getContainerInterface(java.lang.Object)` returns `null`.

Creating a temporary representation object

The network adapter supports temporary representation objects. These objects are placeholders that can be used in place of permanent representation objects for editing purposes and, more specifically, when new objects are created in the network view. When a business object corresponding to the temporary representation object is added to the data source, this temporary representation object is removed and replaced by the permanent representation of the business object. A filter, defined by `IlpFilter`, is used to determine when the representation object of a business object added to the data source is a candidate to replace the temporary representation object. Filtering criteria can be of any kind.

The following example shows how to add a temporary representation object to a network adapter.

How to add a temporary representation object to a network adapter

First you create the temporary representation object, like this:

```
IlpDefaultNetworkNode temp=
          new IlpDefaultNetworkNode(new IlpDefaultAttributeGroup());
```

Then you add it to the adapter along with the filtering criteria using the method `storeTemporaryRepresentationObject(ilog.cpl.model.container.IlpRepresentationNode, ilog.cpl.model.container.IlpMutableRepresentationNode, ilog.cpl.util.IlpFilter)`:

```
adapter.storeTemporaryRepresentationObject(temp, null, new IlpFilter() {
  public boolean accept(Object o) {
   IlpObject ilpO = (IlpObject)o;
   return ilpO.getIdentifier().equals("right one");
   }
};
```

The temporary representation object will be replaced by a permanent representation object as soon as a business object satisfying the filtering criteria is added to the data source.

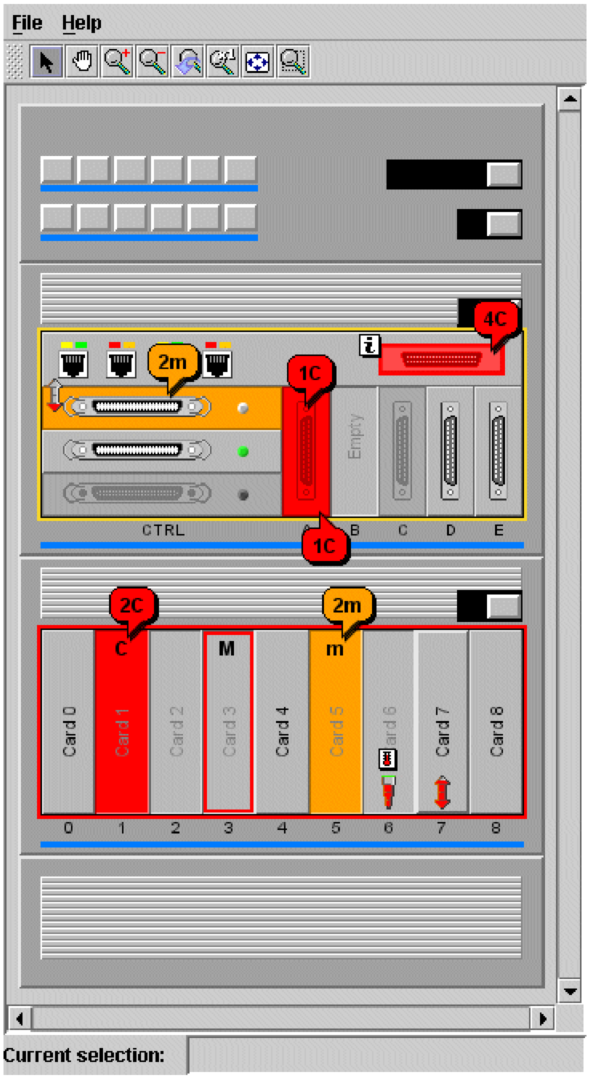
Equipment component

Describes the equipment component, which is one of the four graphic components provided in IBM® ILOG® JViews TGO. It is designed to display and interact over telecommunication equipment such as shelves, cards, ports, and LEDs.

In this section

Introducing the equipment component

The equipment component is based on the Swing equipment component. It allows you to display shelves, cards, ports and LEDs connected by links on top of a background map.



The equipment component is connected to a data source from which it obtains the business objects to be displayed. By default, the equipment component displays all the objects contained in the data source. However, it is also possible to restrict the contents displayed by:

♦ Selecting the root nodes to be shown

- Applying a filter
- Specifying the business classes to be accepted or excluded by the component
- Specifying whether nodes are expandable or not (load on demand)

The equipment is described in detail in Shelves and cards . You can customize: add, remove, configure shelves, cards, ports, and LEDs, either through the equipment component API or through a user-friendly Equipment Editor provided in ***<installdir>*** **/samples/equipment/equipmenteditor**.

Objects that do not have a parent are displayed as root nodes, while the others are displayed under their parent.

The equipment component offers the following notable features:

- Filtering capabilities

  The equipment component allows you to filter the nodes that are displayed. That is, the business objects present in the attached data source are only displayed if they are accepted by the current filter.

- Interaction support

  You can interact with the equipment view as a whole as well as with individual objects.

- Load on demand

  The equipment component supports load on demand for the business objects to be displayed. This means that the graphic representation of a given business object is only created when its parent object is expanded through code or through user interaction. By default, load on demand is customized through the CSS property `expansion` (see Customizing the expansion of business objects). More advanced customization can be performed at the adapter level (see *Expansion strategy*).

- Layout capabilities

  You can perform node and label layouts.

- Zooming capabilities

  There are three zoom policies: physical, logical, and mixed.

- Background support

  The equipment component allows you to display maps in the background.

The equipment component is implemented by the class `IlpEquipment`, which is a Swing `JComponent` that can be directly inserted into a panel (`JPanel`).

`IlpEquipment` provides the API for the most common uses of the equipment component, such as:

- setting or retrieving the associated data source: `getDataSource()`, `setDataSource(ilog.cpl.datasource.IlpDataSource)`
- accessing and modifying the selection: `getSelectionModel()`, `setSelectionModel(ilog.cpl.equipment.IlpEquipmentSelectionModel)`, `addSelectionObject(ilog.cpl.model.IlpObject)`, `removeSelectionObject(ilog.cpl.model.IlpObject)`, `clearSelection()`,

`isObjectSelected(ilog.cpl.model.IlpObject)`, `getSelectedObject()`, `getSelectedObjects()`

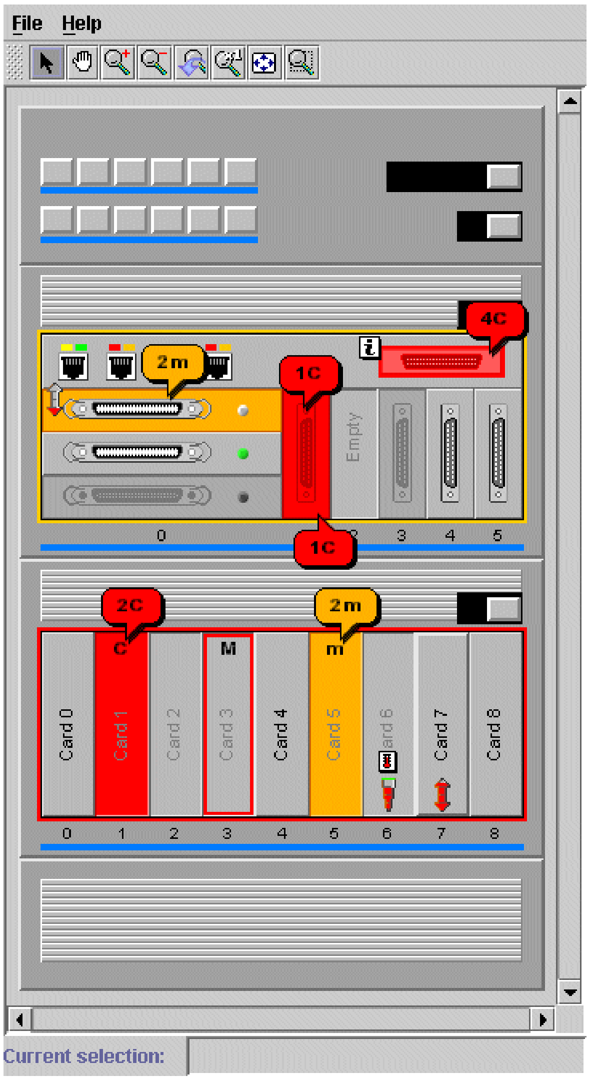
- setting or retrieving the view interactor: `setViewInteractor(ilog.cpl.interactor.IlpViewInteractor)`, `getViewInteractor()`
- changing the root nodes of the equipment through the data source adapter: `getAdapter()`
- filtering the equipment nodes: `setFilter(ilog.cpl.util.IlpFilter)`, `getFilter()`

`IlpEquipment` also acts as a façade for a number of lower-level components that it contains. These components provide more detailed APIs and advanced services. They are described in *Architecture of the equipment component*.

The information presented in this section is based on samples of typical equipment applications.

Creating an equipment component: a sample

This topic shows you how to create the following equipment view:



An equipment view

The following example describes the steps of creating a frame as an equipment container, creating an instance of `IlpEquipment`, creating a data source and connecting it to the equipment, and finally reading in the equipment data.

How to create a basic equipment component

1. Create a frame to contain the equipment.

```
// Create a frame.
JFrame frame = new JFrame("ILOG JTGO equipment sample");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

2. Create the equipment component.

```
IlpEquipment equipment = new IlpEquipment(sampleConfigurationFile, context)
;
frame.getContentPane().add(equipment);
```

   You need to create a new instance of an equipment and make sure that the appropriate configuration is assigned. The equipment configuration is normally read in from a CSS file. You ensure that the equipment configuration file is taken into account and parsed by passing the URL and context or, as in the example, by passing the filename and context as arguments of `IlpEquipment`. For more information on how to create the equipment configuration, see *Configuring an equipment component through CSS* . Then you add the equipment component to the frame.

3. Create the data source and connect it to the equipment component.

```
// Creates the data source
IltDefaultDataSource dataSource = new IltDefaultDataSource(context);
// Set data source for the equipment component
equipment.setDataSource(dataSource);
```

   Once the equipment component is created, you need to associate it with a data source. By default, no data source is set for the equipment component. The data source holds the business objects that will be converted into representation objects by the equipment adapter through the data source API.

4. Read in an XML file, `equipment.xml`, that contains the equipment nodes and links.

```
dataSource.parse("equipment.xml");
```

   The default data source creates the business objects by parsing an XML file or stream where the objects are described, as shown. You could instead create your own data source from a file or database, or even create a complex data source based on proprietary object definitions.

For information on how to define an XML file, refer to Defining the business model in XML
.

How to add business objects to the data source for an equipment component through an XML File

The easiest way to populate the data source for an equipment is to read an XML data file.

The business object IDs in the XML file must be unique within the given equipment.

The following example shows an object definition in the XML file.

```
<addObject id="1004035002697 60">
  <class>
    ilog.tgo.model.IltShelf
  </class>
      ...
</addObject>
```

How to add business objects to the data source for an equipment component through the API

An alternative way to populate the data source for an equipment is to create business objects and insert them in the data source through the API.

The following example shows how to insert a shelf.

```
// Creates an empty shelf using API
IlpObject obj = new IltShelf(3, 40, 30, 0);
// Add object to data source
dataSource.addObject(obj);
// Set its position to 200, 50 in the view
obj.setAttributeValue(IltShelf.PositionAttribute, new IlpPoint(200, 50));
```

Configuring the equipment component

Describes how to display an equipment using rendering information. This information defines how to display equipment data.

In this section

Configuring an equipment component through CSS

You can configure an equipment either through a CSS configuration file or through the API, the easiest and preferred way being the CSS configuration. You also have the possibility to load a project file which combines the CSS configuration and the equipment data. You can customize the behavior and properties of the equipment component, for example by redefining the toolbar, specifying a zoom policy, or choosing a specific background image. You can also customize the behavior and properties of the equipment objects.

You can customize the following features in a CSS file:

- Equipment view
    - Toolbar visibility
    - Toolbar buttons
    - Overview window
    - View interactor
    - Zoom policy
    - Link layout
    - Type of background map and its URL
    - Position converter
- Equipment adapter
    - Expansion
    - Filtering
    - Origin
    - Node factory
    - Link factory
    - Accepted classes
    - Excluded classes

How to load a CSS file in an equipment component

The equipment configuration can be split across several CSS files. The method `setStyleSheets(int, java.lang.String)` accepts several CSS filenames.

There are three ways to apply a CSS configuration to an equipment component, depending on whether you have one or several configuration files:

- If you have a single configuration file, and you do not want to inherit the settings from the default configuration file, pass the CSS configuration filename to the constructor of `IlpEquipment`, as follows:

```
equipmentComponent = new IlpEquipment(myConfigurationFile);
```

If no CSS file is specified, the equipment component uses the default configuration, that is, `ilog.cpl.equipment.defaultConfiguration.css` from the `jviews-tgo-all.jar` file.

♦ If you have one or several configuration files to be applied, you can specify a project file that lists the style sheets, the configuration files, and the XML data file to be loaded in the component (see *Loading a project file*). The following example shows a project file with two configuration files.

```
<?xml version="1.0"?>
<tgo xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/project.xsd"
style="configurationFile1.css,configurationFile2.css">
  <datasource javaClass="ilog.tgo.datasource.IltDefaultDataSource"
fileName="equipment.xml"/>
</tgo>
```

If the settings in two CSS files disagree, the effect depends on the order of the filenames in the list: the last file mentioned takes precedence over the first file.

♦ If you have several configuration files to be applied together, use the `setStyleSheets` method, as follows:

```
equipmentComponent = new IlpEquipment();
equipmentComponent.setStyleSheets(
  new String[] { myConfigurationFile1, myConfigurationFile2 });
```

or, if you want to inherit and extend the settings from the default configuration file, use `setStyleSheets` as follows:

```
equipmentComponent = new IlpEquipment();
equipmentComponent.setStyleSheets(
  new String[] {
    IlpEquipment.DefaultConfigurationFileName,
    myExtraConfigurationFile
  });
```

If the settings in two CSS files disagree, the effect depends on the order of the filenames in the list: the last file mentioned takes precedence over the first file.

How to configure an equipment component in a CSS file

The following code represents an example of configuring an equipment component in CSS. It is based on the CSS files located in ***<installdir>* /samples/equipment/styling** where `<installdir>` is the directory where you have installed JViews TGO.

The configuration in CSS is organized as a set of rules that define properties.

```
Equipment {
  toolbar: true;
  overview: true;
  interactor: true;
  zooming: true;
  graphLayout: true;
  backgrounds: true;
  positioning: true;
}

ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
  button[1]: @+ZoomResetButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
}

Subobject#ZoomResetButton {
  actionType: "ZoomReset";
}

Overview {
  enabled: true;
}

Interactor {
  name: "Select";
}

Zooming {
  type: "Mixed";
  zoomThreshold: 2.0;
}

GraphLayout {
  class: "ilog.views.graphlayout.grid.IlvGridLayout";
}

Backgrounds {
  background[0]: "data/images/europe.jpg";
}

Positioning {
  positionClass: "ilog.cpl.graphic.IlpPosition";
  converterClass: "ilog.cpl.network.IlpDefaultPositionConverter";
}
```

The Equipment rule

This rule specifies Boolean flags that indicate whether each customizable property is present. For example, customization of the property `GraphLayout` is not taken into account unless `graphlayout: true;` is declared in the Equipment rule.

This feature provides powerful cascading possibilities of CSS files. Thus, you can define `GraphLayout` customizations in a default CSS file and then turn them on or off in another CSS file.

The following properties are supported in the Equipment rule. You will find detailed documentation for each of these properties in the IBM® ILOG® JViews TGO *Java™ API Reference Documentation*, package `ilog.cpl.equipment.renderer`.

CSS properties of the equipment view

Property	Rule Type
`adapter`	Adapter
`toolbar`	ToolBar
`view`	View
`overview`	Overview
`interactor`	Interactor
`zooming`	Zooming
`graphLayout`	GraphLayout
`labelLayout`	LabelLayout
`backgrounds`	Backgrounds
`positioning`	Positioning

The ToolBar rule

This rule controls the toolbar.

The property `enabled` is a Boolean property, with default value `true`. It controls whether the toolbar is visible or not.

The property `external` is a Boolean property, with default value `false`. It specifies whether the placement and visibility of the toolbar are managed by user code instead of internally by the equipment component.

Buttons can be added through the syntax `button[i]: @+ButtonId;` followed by the customization setting of the button with the given `ButtonId`.

A button has a mandatory property, `actionType`. This property specifies the action triggered by the button or a separator that is added to the toolbar. The value can be:

- a short name, such as `Select`, used for predefined actions, or
- the name of a subclass of `AbstractButton` with a constructor that takes an instance of `IlpViewsView` as argument, or

- the `Separator` short name to indicate that a separator should be placed in the specified position.

How to add a toolbar separator for the equipment component

You can add toolbar separators in specified positions of the equipment component toolbar. When configuring the equipment component toolbar, you can specify the position where a separator should be placed by using the predefined button action called `Separator`. This button action supports an optional property, `dimension`, which allows you to specify the dimensions of the separator in the toolbar.

The following example shows how to achieve this result:

```
ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
  button[1]: @+Separator;
  button[2]: @+PanButton;
}
Subobject#Separator {
  actionType: "Separator";
  dimension: "20,10";
}
```

The predefined values for the `actionType` property are the following:

Predefined values of the `actionType` property

`actionType` Values	Bean Class	Description
`ZoomIn`	`IlpEquipmentZoomInButton`	Allows you to zoom in the view
`ZoomOut`	`IlpEquipmentZoomOutButton`	Allows you to zoom out of the view
`ZoomBack`	`IlpEquipmentZoomBackButton`	Allows you to go back to the previous zoom level
`ZoomReset`	`IlpEquipmentZoomResetButton`	Allows you to reset the zoom level to the original level
`ZoomView`	`IlpEquipmentZoomViewButton`	Allows you to specify a rectangular area on which to zoom
`FitToContents`	`IlpEquipmentFitToContentsButton`	Allows you to fit the contents of the view to the size of the view
`ScrollToContents`	`IlpEquipmentScrollToContentsButton`	Allows you to recenter the view
`Pan`	`IlpEquipmentPanButton`	Allows you to pan the view
`Select`	`IlpEquipmentSelectButton`	Allows you to select and move objects
`MakeLink`	`IlpEquipmentMakeLinkButton`	Allows you to create links
`MakeLinearGroup`	`IlpEquipmentMakeLinearGroupButton`	Allows you to create linear groups
`MakePolyGroup`	`IlpEquipmentMakePolyGroupButton`	Allows you to create polygonal groups
`MakeRectGroup`	`IlpEquipmentMakeRectGroupButton`	Allows you to create rectangular groups
`EditGroup`	`IlpEquipmentEditGroupButton`	Allows you to edit the shape of groups
`EditLabel`	`IlpEquipmentEditLabelButton`	Allows you to edit labels
`EditObject`	`IlpEquipmentEditObjectButton`	Allows you to edit equipment objects
`LabelLayout`	`IlpEquipmentLabelLayoutButton`	Allows you to trigger the label layout

A button can have a property `permanent`. This is a Boolean property, with default value `true`. For interactor buttons, this property denotes whether the interactor remains attached after it has performed its action.

A button can have a property `name`. This property specifies the name by which other elements in the file refer to the button. The default name is the short name used as `actionType`.

A button can have additional properties, corresponding to Bean properties of the Java™ class. For example, the `Select` button has the properties `multipleSelectionMode`, moveAllowed, dragAllowed, editingAllowed, moveThreshold, opaqueMove, showingMovingObject, opaqueDragSelection, opaqueResize, opaquePolyPointsEdition, multipleSelectionModifier, selectionModifier, which are documented in the class ilog.cpl.equipment.action.toolbar. IlpEquipmentSelectButton.

An interactor button can have key or gesture actions attached to it. These actions are triggered by specific keystrokes or gestures while the interactor is active. These actions are added through the syntax action[i]: @+ActionId; followed by a customization setting for the action.

An action customization has the mandatory property class which specifies the Java class name of the javax.swing.Action object. Bean properties of this class are also customizable. The properties key and gesture can be set to specify when the action is to be executed. These two properties are not used in combination. For example, if you specify:

```
key: "control A";
gesture: "BUTTON1_CLICKED";
```

the action will be executed either when the key sequence ‘control-A’ is typed, or when the mouse BUTTON1 is clicked. To define the property "gesture", specify one of the predefined user gestures defined in class IlpGesture. To define the property "key", specify a string that will be converted to a keystroke by the type converter (IlpTypeConverter).

The following predefined actions are available:

♦ ilog.cpl.graph.action. IlpSelectAllObjectsAction

♦ ilog.cpl.graph.action. IlpRemoveSelectedObjectsAction

An interactor can have a pop-up menu factory associated with it. You can specofy this factory using the Bean property popupMenuFactory. Its value should be an indirect reference (through @+ or @#) to a property class or to other Bean properties. For example,

```
Subobject#SelectButton {
    actionType: "Select";
    popupMenuFactory: @+popupMenuFactory;
  }

Subobject#popupMenuFactory {
    class: 'CustomPopupMenuFactory';
  }
```

In this example, the value of the property popupMenuFactory is a bean that is defined by the class CustomPopupMenuFactory. This class should implement the interface IlpPopupMenuFactory.

For more information on configuring the toolbar in an equipment view, refer to the class IlpToolBarRenderer.

How to add a predefined toolbar button to the equipment component

JViews TGO provides a list of predefined toolbar buttons usable in the equipment component (see *Predefined values of the actionType property*).

The following example shows how to add a predefined button that enables the `select` interactor. When this interactor is enabled, you can customize actions, pop-up menus and interactor properties as illustrated here:

```
ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
  action[0]: @+action0;
  popupMenuFactory: @+popupMenuFactory;
}

Subobject#popupMenuFactory {
  class: 'CustomPopupMenuFactory';
}

Subobject#action0 {
  key: "control A";
  class: 'ilog.cpl.graph.action.IlpSelectAllObjectsAction';
}
```

How to add a custom toolbar button to the equipment component

To add your own toolbar button, you need to create a new action class that inherits from `IlpEquipmentInteractorAction` and contains a constructor that takes an `IlpViewsView` as parameter.

```
public class CustomButtonAction extends IlpEquipmentInteractorAction {

  public CustomButtonAction(IlpViewsView view) {
      super(view);
      // Do any needed initialization
      // Define your own view interactor that will be active when the button 
is
selected
      // in the toolbar
      IlpViewsViewInteractor interactor =  new IlpViewsViewInteractor();
      // Register the interactor in this action
      setIlpInteractor(interactor);
  }
}
```

Then, you need to register this new button in your component configuration, as follows:

```
ToolBar {
  enabled: true;
  button[0]: @+MyButton;
}

Subobject#MyButton {
  actionType: 'CustomButtonAction';
  toolTipText: "Custom";
  icon: @+customIcon;
}
Subobject#customIcon {
  class: 'javax.swing.ImageIcon';
  image: '@|image("custom.png")';
}
```

The custom action will be encapsulated in an `IlpEquipmentInteractorButton`, and you will be able to customize the properties of this button as with the predefined buttons. For example, you can customize the following:

- `name`
- `usingObjectInteractor`
- `popupMenuFactory`
- actions associated with gestures and keystrokes

The View rule

This rule controls the view.

You can customize the following properties of the view:

View properties

Property Name	Type	Default value	Description
`horizontalScrollBarPolicy`	`int`	`IlvJScrollManagerView.HORIZONTAL_SCROLLBAR_AS_NEEDED`	Defines the policy for the visibility of the horizontal scrollbar
`verticalScrollBarPolicy`	`int`	`IlvJScrollManagerView.VERTICAL_SCROLLBAR_AS_NEEDED`	Defines the policy for the visibility of the vertical scrollbar
`keepingAspectRatio`	`boolean`	`false`	Defines whether the view keeps the aspect ratio when zooming
`minZoomXFactor`	`double`	`0`	Specifies the minimum zoom factor

Property Name	Type	Default value	Description
			allowed on the X (horizontal) axis of the view
`maxZoomXFactor`	`double`	`Double.MAX_VALUE`	Specifies the maximum zoom factor allowed on the X (horizontal) axis of the view
`minZoomYFactor`	`double`	`0`	Specifies the minimum zoom factor allowed on the Y (vertical) axis of the view
`maxZoomYFactor`	`double`	`Double.MAX_VALUE`	Specifies the maximum zoom factor allowed on the Y (vertical) axis of the view
wheelZoomingEnabled	`boolean`	`true`	Defines whether the view zooms in response to moving the mouse wheel while pressing the Control key
wheelScrollingEnabled	`boolean`	`true`	Defines whether the view scrolls in response to moving the mouse wheel

The following CSS sample shows how to customize the view:

```
View {
  horizontalScrollBarPolicy: AsNeeded;
  verticalScrollBarPolicy: Never;
  keepingAspectRatio: true;
}
```

For more information on configuring an equipment view, refer to the class `IlpViewRenderer`.

The Overview rule

The Overview rule controls the overview window.

The property `overviewVisible` controls the visibility of the overview window. The default value is `false`.

The following CSS sample shows how to customize the overview:

```
Overview {
  overviewVisible: true;
}
```

For more information on configuring the overview window in an equipment view, refer to the class `IlpOverviewRenderer`.

The Interactor rule

The Interactor rule controls the interactor associated with the view.

You can customize the following properties of the interactor:

Interactor properties

Property Name	Type	Default value	Description
`name`	`String`	none	Specifies the name of a toolbar button that activates an interactor. This button is activated at startup. Its interactor becomes the initial view interactor, as well as the default view interactor when another interactor stops its interaction. This property is only considered when the view has a toolbar configured and enabled.
`viewInteractor`	`IlpViewsViewInteractor`	none	Specifies the interactor instance that becomes the initial view interactor, and the default view interactor when another interactor stops its interaction.

How to configure an equipment interactor in a CSS file

Prior to configuring the equipment view interactor, you need to configure the equipment component so that the interactor configuration is enabled:

```
Equipment {
  interactor: true;
}
```

After that, you can customize the interactor property in the Interactor rule as illustrated by the next code extract. Refer to The CSS specification in the *Styling* documentation for details about the CSS syntax.

How to set the default view interactor from the toolbar of the equipment component

You can customize the default view interactor to be one of the interactors present in the toolbar configured for the equipment component. In this case, the toolbar button is identified when the equipment component is configured, and it is activated at startup. This interactor becomes the initial view interactor, and the default view interactor when another interactor stops its interaction. This configuration is achieved through the property `name`, whose value must be the name of one of the configured toolbar buttons.

The following CSS extract configures the equipment view to use the `Select` toolbar button as the default view interactor:

```
Interactor {
  name: "Select";
}
```

How to set the default view interactor when the toolbar is disabled in the equipment view

When the toolbar is disabled, you can no longer specify the default view interactor by using a toolbar button name, as in the example above. However, you can specify the view interactor directly in CSS, as follows:

```
Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.graphic.views.IlpViewsViewInteractor';
}
```

The behavior of the view interactor is determined by the actions that are associated with user gestures and keystrokes. This behavior can also be customized through CSS. You can also configure a pop-up menu to be displayed in the equipment view. For more information about interactor customization, refer to *Interacting with the equipment view* and *Interacting with the equipment objects* .

When the interactor renderer is enabled, you can also customize interactors for objects displayed in the equipment component. The property in the object selector will only be considered if the interactor renderer is enabled.

For more information on configuring the default view interactor in an equipment view, refer to class `IlpInteractorRenderer` .

The Zooming rule

Th Zooming rule controls the zoom policy.

The mandatory property `type` specifies the type of zoom. The possible values are `Logical`, `Physical`, or `Mixed`. Each zoom policy may have additional properties that you can also set using CSS:

- ♦ Logical zoom policy (see `IltLogicalZoomPolicy`)

Logical zoom property

Property name	Type	Default	Description
`additionalZoom`	`double`	`1`	Specifies an additional zoom factor that is implicitly multiplied with the view's transformer. This property is useful when printing with unusual transformers.

- ♦ Physical zoom policy (see `IltPhysicalZoomPolicy`)

Physical zoom properties

Property name	Type	Default	Description
`decorationNames`	`String[]`	`null`	Specifies a list of decoration names that are customized at the zoom policy level. See `IltGraphicElementName` for a list of decoration names that can be used.
visibilityThresholds	double[]	null	Specifies a list of thresholds, one for each decoration name customized with property `decorationNames`. These thresholds indicate the zoom level below which the decorations become invisible in the view. It allows you to hide decorations as the user zooms out in the view.

- ♦ Mixed zoom policy (see `IltMixedZoomPolicy`)

Mixed zoom properties

Property name	Type	Default	Description
zoomThreshold	double	1	Specifies the zoom threshold when the physical zoom or the logical zoom should be used
subnetworkZoomFactor	double	1	Specifies an additional zoom threshold that is applied to expanded subnetworks
`decorationNames`	`String[]`	`null`	Specifies a list of decoration names that are customized at the zoom policy level. See `IltGraphicElementName` for a list of decoration names that can be used.
visibilityThresholds	double[]	null	Specifies a list of thresholds, one for each decoration name customized with property `decorationNames`. These thresholds indicate the zoom level below which the decorations become

Property name	Type	Default	Description
			invisible in the view. It allows you to hide decorations as the user zooms out in the view.

How to customize the zoom policy in an equipment component

The following CSS sample shows how to customize the zoom policy in an equipment view:

```
Zooming {
  type: "Mixed";
  zoomThreshold: 1.0;
  subNetworkZoomFactor: 1.0;
}
```

For more information on configuring the zoom behavior in an equipment view, refer to class `IlpZoomingRenderer`.

How to configure the visibility of decorations for a specific equipment component

JViews TGO provides three predefined zoom policies that you can use directly in the equipment component (see *Zooming* for more information). When using the physical or the mixed zoom policy, decorations may become invisible as the user zooms out in the view. By default, this configuration is global in the application and can be customized using visibility thresholds. However, you may need to specify the visibility threshold for a specific view. This feature is supported by properties `decorationNames` and `visibilityThresholds`. Property `decorationNames` specifies each decoration that is configured for the view. For a list of the decoration names that can be customized, see `IltGraphicElementName` . Property `visibilityThresholds` specifies the visibility threshold below which the decoration becomes invisible for each decoration name. The following example shows a mixed configuration.

```
Zooming {
  type: "Mixed";
  decorationNames[0]: Name;
  decorationNames[1]: AlarmBalloon;
  decorationNames[2]: AlarmCount;
  decorationNames[3]: Plinth;
  visibilityThresholds[0]: 0.5;
  visibilityThresholds[1]: 0.8;
  visibilityThresholds[2]: 0.5;
  visibilityThresholds[3]: 0.5;
}
```

The GraphLayout rule

This rule allows you to control the automatic node layout in the view and to configure nonautomatic node layouts. Nonautomatic node layouts can only be executed through the API (see *Layout* for more details).

How to control the automatic node layout in the equipment view

Automatic node layout is configured through the property `class`. This property specifies the graph layout class, a subclass of `IlvGraphLayout`. Additional Bean properties can be specified, depending on the class.

The following CSS sample shows how to customize the graph layout:

```
GraphLayout {
    class:
     'ilog.views.graphlayout.uniformlengthedges.IlvUniformLengthEdgesLayout';

    respectNodeSizes: true;
    preferredLinksLength: 200;
    forceFitToLayoutRegion: true;
    layoutRegion: "50, 50, 700, 450";
}
```

The graph layout is always a subclass of `IlvGraphLayout` which supports the "`preserveFixedNodes`" property. This property allows you to switch the support of fixed nodes on or off. You can set a node as fixed in the business object customization.

Note: If a graph layout is set and supports fixed nodes, a link layout is required when links are connected to the fixed nodes.

The properties of each layout algorithm are fully explained in *Layout algorithms* in *Using graph layout algorithms* in the documentation for JViews Diagrammer.

The properties of the `IlvGraphLayout` subclasses conform to the following JavaBeans™ convention: if a class has a pair of methods called `setMyProp` (with a single parameter) and `getMyProp` (without parameters), then you can set the property `myProp` in the style sheet.

Note: If the value of the property is an enumeration of integer values defined by static member variables of the class, then you can use the name of the variable alone, or the variable name prefixed by the class name alone, or the variable name prefixed by the fully qualified class name. For example, the following declarations are all valid:

```
globalLinkStyle: "ORTHOGONAL_LINKS";

globalLinkStyle: "IlvHierarchicalLayout.ORTHOGONAL_LINKS";

globalLinkStyle: "ilog.views.graphlayout.hierarchical.
IlvHierarchicalLayout.ORTHOGONAL_LINKS";
```

How to configure multiple node layouts in an equipment view

The Graph Layout rule allows you to configure multiple node layouts, and to define the node layout to be used automatically in the view. Multiple node layout configuration is achieved through the properties `layouts` and `autoLayoutIndex`, as illustrated below:

```
GraphLayout {
  layouts[0]: @+treeLayout;
  layouts[1]: @+hierarchicalLayout;
  layouts[2]: @+springEmbedderLayout;
  autoLayoutIndex: 1;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
  flowDirection: Bottom;
}

Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Bottom;
}

Subobject#springEmbedderLayout {
  class: 'ilog.views.graphlayout.springembedder.IlvSpringEmbedderLayout';
  respectNodeSizes: true;
  preferredLinksLength: 200;
  forceFitToLayoutRegion: true;
  layoutRegion: "50, 50, 700, 450";
}
```

In this use case, three node layouts are configured for the view: `IlvTreeLayout`, `IlvHierarchicalLayout` and `IlvSpringEmbedderLayout`. The hierarchical layout is configured to be performed automatically when the contents of the view changes. This configuration is achieved by specifying the value of property `autoLayoutIndex` as the index of the hierarchical layout defined through the `layouts` property. The other node layouts can be performed on demand using the API (see `IlpGraphView.performAttachedLayout`).

How to configure nonautomatic node layouts in the equipment component

If you are not interested in automatic node layout, you can still configure multiple node layouts in CSS. To have only nonautomatic node layouts, set property `autoLayoutIndex` to `-1`, as illustrated below:

```
GraphLayout {
  layouts[0]: @+treeLayout;
  layouts[1]: @+hierarchicalLayout;
  layouts[2]: @+springEmbedderLayout;
  autoLayoutIndex: -1;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
  flowDirection: Bottom;
}

Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Bottom;
}

Subobject#springEmbedderLayout {
  class: 'ilog.views.graphlayout.springembedder.IlvSpringEmbedderLayout';
  respectNodeSizes: true;
  preferredLinksLength: 200;
  forceFitToLayoutRegion: true;
  layoutRegion: "50, 50, 700, 450";
}
```

How to set node or link parameters on graph layout objects in the equipment component

You can set parameters for a graph layout algorithm that applies to a particular node or link, in the style sheet. Such parameters are defined by a method of the form:

```
setMyParam(Object node,  value);
```

or

```
setMyParam(Object link,  value);
```

Node parameters are set in the style sheet as follows:

```
object."ilog.tgo.model.IltObject":graphLayoutRenderer {
  myParam: "value";
}
```

The name of the property is the name of the method, without the prefix `set`. The pseudoclass `graphLayoutRenderer` indicates that the declarations apply to the node layouts that are configured in the graph layout rule.

For example, the graph layout defines a `setFixed` method that lets you specify whether a node or link is fixed. Fixed nodes or links are not moved when the layout is applied. The signature of the method is:

```
setFixed(Object nodeOrLink, boolean fixed);
```

In the style sheet, you can set this parameter as follows:

```
object."ilog.tgo.model.IltObject":graphLayoutRenderer {
  fixed: true;
}
```

The value of the property can be any basic type (integer, String, float), or it can be the name of a public constant defined by the graph layout class, for example, `WEST` , which is defined in the class `IlvHierarchicalLayout`.

When the graph layout rule contains multiple node layouts, you can still specify node and link layout parameters by using pseudoclasses that identify the graph layout to which the declarations apply.

```
GraphLayout {
  layouts[0]: @+treeLayout;
  layouts[1]: @+hierarchical;
  autoLayoutIndex: 0;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
  flowDirection: Bottom;
}

Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Top;
}

#NE1:graphLayoutRenderer:tree {
  root: true;
}

#NE1:graphLayoutRenderer:hierarchical {
  specNodeLevelIndex: 0;
}

#NE2:graphLayoutRenderer:hierarchical {
  specNodeLevelIndex: 1;
}
```

In the example above, `NE1` is configured as the root object for the Tree Layout algorithm. This is achieved by declaring the property `root` in a selector that contains the pseudoclasses `graphLayoutRenderer` (indicates that this is a graph layout renderer per-object property) and `tree` (indicates that this is a property specific to the `IlvTreeLayout` algorithm).

At the same time, `NE1` is configured to be placed at level 0 in case of a hierarchical layout. This is achieved using pseudoclasses `graphLayoutRenderer` and `hierarchical` (indicates that this is a graph layout per-object property specific to the `IlvHierarchicalLayout` algorithm).

Each layout algorithm supports a set of per-object parameters. For more information on the parameters supported by each layout algorithm, refer to package `ilog.cpl.graph.css.renderer.graphlayout`.

In addition to the properties that are specific to the layout algorithms, the graph layout renderer also supports the following properties:

- `layoutIgnored`: If this property is set to `true`, the object is completely ignored by the graph model (using an `IlvLayoutGraphicFilter`).

- `markedForIncremental`: If the layout algorithm is an `IlvHierarchicalLayout`, you can use the property `markedForIncremental`. When this property is set to `true` for an object, the method `IlvHierarchicalLayout.markForIncremental(java.lang.Object)` is called for this object. This means that the position of the object is recomputed during the next incremental layout. This property has an effect only if the `incrementalMode` property of the layout itself is set to `true`. For example:

```
GraphLayout {
  layouts[0]: @+hierarchicalLayout;
}

Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  incrementalMode: true;
}

#NE1:graphLayoutRenderer:hierarchical {
  markedForIncremental : "true";
}
```

Important: Starting with JViews TGO 7.5, if you are not using any node or link parameters, you can disable this mechanism by specifying `IlpGraphLayoutRenderer.setUsePerObjectParameters(false)`. This will remove the overhead of testing the parameters and speed up the rendering process significantly.

How to disable the per-object layout parameters for node configuration in the equipment view

You can also disable the per-object layout parameters configuration through CSS as follows:

```
GraphLayout {
  layouts[0]: @+hierarchicalLayout;
  usePerObjectParameters: false;
}
```

For more information on configuring the node layout in an equipment view, refer to the class `IlpGraphLayoutRenderer`.

The LinkLayout rule

The LinkLayout rule controls the automatic link layout in the view.

The mandatory property `class` specifies the link layout class, a subclass of `IlvGraphLayout`. Additional Bean properties can be specified, depending on the class.

How to control the automatic link layout in the equipment view

The following CSS sample shows how to customize the link layout:

```
LinkLayout {
    class: "ilog.tgo.graphic.graphlayout.IltShortLinkLayout";
    globalLinkStyle: MIXED_STYLE;
}
```

The link layout is always a subclass of `IlvGraphLayout` which supports the "`preserveFixedLinks`" property. This property allows you to switch the support of fixed links on or off.

Like the graph layout renderer, the link layout renderer supports setting per-object link layout parameters through CSS. See *How to set node or link parameters on graph layout objects in the equipment component*.

How to specify per-object parameters for link layouts in the equipment view

Link parameters are set in the style sheet as follows:

```
object."ilog.tgo.model.IltAbstractLink":linkLayoutRenderer {
  linkStyle: ORTHOGONAL_STYLE;
}

#Link1:linkLayoutRenderer {
  linkStyle: DIRECT_STYLE;
}
```

Important: Starting with JViews TGO 7.5, if you are not using any node or link parameters, you can disable this mechanism by specifying `IlpLinkLayoutRenderer.setUsePerObjectParameters(false)`. This will remove the overhead of testing the parameters and speed up the rendering process significantly.

How to disable per-object layout parameters for link configuration in the equipment view

You can also disable the per-object layout parameters configuration through CSS as follows:

```
LinkLayout {
  class: 'ilog.views.graphlayout.link.IlvLinkLayout';
  usePerObjectParameters: false;
}
```

For more information on configuring the link layout in an equipment view, refer to class `IlpLinkLayoutRenderer`.

The LabelLayout rule

The LabelLayout rule controls the automatic label layout in the view.

The mandatory property `class` specifies the label layout class, a subclass of `IlvLabelLayout`. Additional Bean properties can be specified, depending on the class. The usual setting is `ilog.tgo.graphic.graphlayout.labellayout. IltAnnealingLabelLayout`.

The following CSS sample shows how to customize the label layout:

```
LabelLayout {
   class: 'ilog.tgo.graphic.graphlayout.labellayout.IltAnnealingLabelLayout';

    obstacleOffset: 10;
    labelOffset: 15;
}
```

Some properties can be set within `IltAnnealingLabelLayout` and some within `IlvAnnealingLabelLayout`. Refer to the IBM® ILOG® JViews TGO *Java™ API Reference Documentation* for more details.

For more information on configuring the label layout in an equipment view, refer to class `IlpLabelLayoutRenderer`.

The Backgrounds rule

The Backgrounds rule allows you to configure two kinds of background that affect the equipment component representation:

♦ Equipment backgrounds

♦ Manager view background

Equipment backgrounds

You can use background files such as maps or images. You can specify an equipment background through:

1. A URL

   The background file is specified directly by its URL:

   ```
   Backgrounds {
     background[i]: "URL";
   }
   ```

   For example:

   ```
   Backgrounds {
     background[0]: "backgrounds/sf-bayarea.png";
   }
   ```

2. A CSS bean

   The background URL and its properties are specified through a CSS bean:

```
Backgrounds {
  background[i]: @+background0;
}

Subobject#background0 {
  class: "ilog.cpl.graph.background.css.IlpBackgroundCSSConfiguration";

  PROPERTY :  PROPERTY_VALUE;
  ...
}
```

The bean `IlpBackgroundCSSConfiguration` encapsulates all the properties supported by the predefined background types. You should use it if want to use one of the predefined types.

For example:

```
Backgrounds {
  background[0]: @+background0;
}

Subobject#background0 {
  class: "ilog.cpl.graph.background.css.IlpBackgroundCSSConfiguration";

  //////////////////////
  //Background properties
  //////////////////////
  url : "backgrounds/sf-bayarea.png";
  loadOnDemand : "true";
  threaded : "false";
}
```

If you intend to use a custom background type that has additional properties, you can either subclass the default `IlpBackgroundCSSConfiguration` and use it with the additional bean properties or provide a bean that contains all the required properties to configure the `IlpBackground` implementation. All bean properties are automatically communicated and stored in the `IlpBackground` implementation through its `IlpBackground.setProperty` interface method.

For details on the properties available and supported for each `IlpBackground` implementation, see *Background support*.

You can mix and match options 1 and 2 by specifying some backgrounds as beans and some as straight URL strings.

In the case of the Image Tile background type (`IlpImageTileBackground`), only the `url` property can be configured through CSS. For the other properties, use the XML configuration. See *Background support*.

Manager view background

This refers to the representation of the view as a background for your equipment backgrounds (the area that the equipment backgrounds do not cover).

You can configure the manager view background as follows:

```
Backgrounds {
  PROPERTY : PROPERTY_VALUE;
}
```

For example:

```
Backgrounds {
  backgroundColor : "white";
}
```

The following table lists the properties that allow you to customize the manager view background:

Properties of the view background

Property name	Type	Default	Sample	Description
backgroundColor	Color	null	backgroundColor:"black";	Specifies the color to be used to fill the background of the view.
backgroundPattern	String	null	backgroundPattern:"pattern.png";	Specifies the location of the pattern image to be used to fill the background of the view.

For more information on configuring the background in an equipment view, refer to class `IlpBackgroundsRenderer`.

The Positioning rule

The Positioning rule controls the type and converter of the user-defined `IlpPosition`.

The property `positionClass` denotes the Java class name of the class or interface that implements `IlpPosition`.

The property `converterClass` denotes a Java class name or CSS bean that implements the `IlpPositionConverter` interface and determines the conversion between business data coordinates and (x,y) coordinates in the view.

The following CSS sample shows how to customize the positioning:

```
Positioning {
  positionClass: 'my.package.MyPosition';
  converterClass: 'my.package.MyPositionConverter';
}
```

For more information on configuring the positioning in an equipment view, refer to class `IlpPositioningRenderer`.

The Adapter rule

The Adapter rule controls the configuration of the equipment adapter. The equipment adapter is responsible for converting the business objects in the data source to representation objects (equipment nodes) in the equipment component. It provides the following features:

- **Filtering**: applies a filter so that business objects currently in the data source are not mapped to representation objects in the equipment component.
- **Origins**: defines which objects become root nodes in the equipment.
- **Link factory**: defines how a link representation object will be created from its business object counterpart.
- **Node factory**: defines how a representation object that is not a link will be created from its business object counterpart.
- **Expansion strategy**: defines how the objects will be loaded in the equipment component, that is, either at initialization time or as the user interacts with the equipment nodes.
- **Accepted classes**: defines the list of business classes that are accepted by the equipment adapter. Only the business objects that match one of these business classes will be mapped to representation objects by the equipment adapter.
- **Excluded classes**: defines the list of business classes that are excluded by the equipment adapter. The business objects of these business classes will not be mapped to representation objects by the equipment adapter. By default, the `IltAlarm` business class is part of the list of excluded classes.

These equipment adapter features can be customized through CSS using the following properties:

CSS Properties of the equipment adapter

Property name	Property type
`filter`	`ilog.cpl.util.IlpFilter`
`origins`	list of object identifiers
`nodeFactory`	`ilog.cpl.equipment.IlpEquipmentNodeFactory`
`linkFactory`	`ilog.cpl.equipment.IlpEquipmentLinkFactory`
`expansionStrategyFactory`	`ilog.cpl.util.IlpExpansionStrategyFactory`
acceptedClasses	list of `IlpClass`
excludedClasses	list of `IlpClass`

How to configure an equipment adapter in a CSS file

Prior to configuring the adapter, you need to configure the equipment component so that the adapter configuration is enabled:

```
Equipment {
  adapter: true;
}
```

After that, you can customize each adapter property in the Adapter rule as illustrated by the following code extract. Refer The CSS specification in the *Styling* documentation for details about the CSS syntax.

```
Adapter {
  filter: @+Filter;
}

Subobject#Filter {
  class: 'CustomFilter';
  rejectObject[0]: "NE1";
  rejectObject[1]: "Link5";
}
```

How to programmatically configure an equipment adapter using CSS

You can programmatically modify the CSS configuration of the default equipment adapter (`IlpEquipmentAdapter`) by using mutable style sheets through the `IlpMutableStyleSheet` API.

Important: The mutable style sheet is set to the adapter as a regular style sheet and is cascaded in the order in which it has been declared.

To use mutable style sheets:

1. Get the mutable style sheet.

   You access the mutable style sheet through the `getMutableStyleSheet()` method in the equipment adapter API:

   ```
   IlpMutableStyleSheet mutable = adapter.getMutableStyleSheet();
   ```

   This method automatically registers the mutable style sheet into the adapter. You can manually instantiate an object of the class `IlpMutableStyleSheet` and register it yourself through the `setStyleSheet()` API:

   ```
   IlpMutableStyleSheet mutable = new IlpMutableStyleSheet(adapter);
   try {
     adapter.setStyleSheets(new String[] { mutable.toString() });
   } catch (Exception x) {
     x.printStackTrace();
   }
   ```

2. Set the CSS declarations.

Once you have the mutable style sheet, you can set the declarations you want:

```
mutable.setDeclaration("#myObjectId", "expansion", "NO_EXPANSION");
```

This creates the following CSS declaration into the mutable style sheet:

```
#myObjectId {
  expansion: NO_EXPANSION;
}
```

3. Register the mutable style sheet.

   The mutable style sheet should be set to the adapter as a regular style sheet using the `setStyleSheet()` method:

```
try {
  adapter.setStyleSheets(new String[] { mutable.toString() });
} catch (Exception x) {
  x.printStackTrace();
}
```

4. Set and update the CSS declarations.

   The mutable style sheet can be modified even after being registered to the adapter:

```
// Update the expansion type for 'myObjectId'
mutable.setDeclaration("#myObjectId", "expansion", "IN_PLACE");
// Add a new declaration
mutable.setDeclaration("#myOtherId", "expansion", "IN_PLACE");
```

Note: Like any style sheet, the mutable style sheet is lost when the `setStyleSheet()` API is invoked and a new set of style sheets is applied to the adapter.

How to customize the mutable style sheet

Reapplying a CSS configuration may be a heavy task, as the adapter may be forced to review filters, origins, recreate representation objects, and so on. It is important to use the mutable style sheet with care and to customize it properly to reapply the CSS wisely. To do so, there are two methods available in the `IlpMutableStyleSheet` API: `setUpdateMask()` and `setAdjusting()`.

1. `setUpdateMask()`

   This method controls what should be recustomized once a declaration of the mutable style sheet has been updated. The CSS configuration of the adapter is divided into two parts: *adapter customization* and *representation object* customization.

   The adapter customization handles the origins, filters, and so on:

```
Adapter {
  origins[0]: id0;
  origins[1]: id1;
  showOrigin: true;
  filter: @+myFilter;
}
```

The representation object customization handles the expansion type of a representation object:

```
#myObjectId {
  expansion: IN_PLACE;
}
```

The accepted values for `setUpdateMask()` are:

- `IlpStylable.UPDATE_COMPONENT_MASK`: Only the adapter part is recustomized.
- `IlpStylable.UPDATE_OBJECTS_MASK`: Only the representation object part is recustomized.
- `IlpStylable.UPDATE_ALL_MASK`: Bot the adapter and representation object parts are recustomized.
- `IlpStylable.UPDATE_NONE_MASK`: Nothing is recustomized.

For example, if you update the expansion type of a representation object through the mutable style sheet, it is recommended that you set the update mask to `UPDATE_OBJECTS_MASK` as there is no need to reapply the CSS configuration for the adapter part:

```
mutable.setUpdateMask(IlpStylable.UPDATE_OBJECTS_MASK);
mutable.setDeclaration("object", "expansion", "IN_PLACE");
```

2. `setAdjusting()`

This method is used when a series of declarations must be applied to the mutable style sheet. When the method is set to `true`, the mutable style sheet puts all the calls to `setDeclaration()` into a queue. When the method is set back to `false`, all the queued declarations are processed in a batch:

```
mutable.setAdjusting(true);
mutable.setDeclaration("#myObjectId", "expansion", "IN_PLACE");
mutable.setDeclaration("#myOtherId", "expansion", "IN_PLACE");
mutable.setAdjusting(false);
```

Configuring an equipment component through the API

For details of the classes involved in the architecture of the equipment component, see *Architecture of the equipment component*.

The following example shows how to configure the equipment view through the API. For programming details on the individual services, see *Equipment component services*.

How to configure the equipment view with the API

```
// Instantiates equipment component
IlpEquipment equipment = new IlpEquipment();
// Sets Toolbar visible
equipment.setToolBarVisible(true);
// Set overview window visible
equipment.setOverviewVisible(true);
// Set default view interactor
IlpSelectInteractor select = new IlpSelectInteractor();
select.setEditingAllowed(true);
equipment.setViewInteractor(select);
// Set zoom policy to mixed
IltMixedZoomPolicy mixed = new IltMixedZoomPolicy();
mixed.setZoomThreshold(2.0);
equipment.setZoomPolicy(mixed);
// Sets Horizontal Scroll Bar invisible
equipment.setHorizontalScrollBarVisible(false);
```

How to configure the equipment adapter with the API

The following example shows how to configure the equipment adapter through the API. See *Equipment component services* for programming details on the individual services.

```
IlpEquipmentAdapter adapter = equipment.getAdapter();

// Filter
IlpFilter myFilter = new MyFilter();
// (it is the same as equipment.setFilter(myFilter);)
adapter.setFilter(myFilter);

// Origin
List myOrigins = new ArrayList();
myOrigins.add(objectID_1);
myOrigins.add(objectID_2);
:
:
myOrigins.add(objectID_n);
// in this case we want to display the origins
boolean showOrigin = true;
adapter.setOrigins(myOrigins, showOrigin);
```

```
// Expansion Strategy Factory
// Usually the expansion strategy factory relies
// on the adapter to access the data source and
// to load/release objects
IlpExpansionStrategyFactory myExpFactory = new
  MyExpansionStrategyFactory(adapter);
adapter.setExpansionStrategyFactory(myExpFactory);

// Position Attribute
// Here, imagine that MyObject implements IlpObject
// interface and defines "Placement" as the
// IlpAttribute that defines the object position
IlpClass myObjectClass = MyObject.getIlpClass();
IlpAttribute myPosAttrib = MyObject.Placement;
adapter.setPositionAttribute(myObjectClass, myPosAttrib);

// Node Factory
IlpEquipmentNodeFactory myNodeFactory = new MyNodeFactory();
adapter.setNodeFactory(myNodeFactory);
```

Customizing the rendering of equipment nodes and links

Equipment nodes and links can be customized through CSS according to their business class. For details, see Customizing network and equipment nodes and Customizing network and equipment links .

Loading a project file

A project is a combination of style sheets that supply rendering information and a data source that supplies the data to be represented in an equipment component. A project is saved as an XML file with extension `.itpr`.

Loading a project file is the recommended way to configure a graphic component in Java™ as it is the fastest.

How to load a project file into an equipment component

The following code sample shows how to load a project file into an equipment component, using the method `setProject`.

```
IlpEquipment equipment = new IlpEquipment();
equipment.setProject(new URL("file:project.itpr");
```

The project is represented by the `IlpTGOProject` class, included in the package `ilog.cpl.project`. When a new project is created, the style sheet and data source are both null.

How to create a new project for the equipment component

The following code sample shows how to create a new project file by setting the style sheets and data source, then saving the project.

```
IlpTGOProject project = new IlpTGOProject();
project.setStyleSheet(new URL("file:example.css");
IltDefaultDataSource dataSource = new IltDefaultDataSource();
dataSource.setFileName("data.xml");
project.setDataSource(dataSource);
project.write(new URL("file:example.itpr");
```

Equipment component services

Describes the services associated with the equipment component in JViews TGO, which are of three kinds: view services, related to the equipment view; adapter services, related to the equipment model; handler services, related to the equipment controller. As most of the equipment services are shared with the network component, you are strongly recommended to read the corresponding topics in *Network component*.

In this section

Background support
Identifies where to find more information on the use of background maps.

Filtering
Describes how to filter nodes displayed by the equipment component.

Accepted and excluded classes
Details how to specify the business classes to be accepted for or excluded from display in the equipment component.

Setting a list of origins
Describes how to set a list of orgins to explicitly select the root nodes to be displayed by the equipment component.

Node factory
Describes the node factory.

Link factory
Describes the link factory.

Expansion strategy
Describes the expansion strategy used by the equipment adaptor to determine whether objects should be loaded in the equipment model.

Interacting with the equipment view

The `IlpEquipment` allows you to associate behavior with the equipment view as a whole and with the business objects it contains. JViews TGO provides predefined view interactors to manage the behavior of the equipment view. See *View interactors*. These interactors allow you to change the zoom factor, to move and resize objects, and to change the relationship between objects. See *Configuring the equipment component* for details on how to enable the interactors.

The predefined interactors available in the equipment view are the following:

♦ Select interactor

  Allows you to select any object displayed in the equipment component and to move root objects. When a root object is moved, its `position` property value is updated to reflect the new user-defined position. Multiple selection is also possible.

♦ Edit interactor

  Like the select interactor, this interactor allows you to select any object displayed in the equipment component, and to move root objects.

  This interactor also allows you to reshape selected objects or to change their relationship. Root objects such as cards, shelves, and card carriers can be reshaped or moved around in the view. Cards and card carriers can be moved from the view to a shelf item container (shelf or card carrier) and vice versa; the same action can be performed on LEDs and ports, moving them from the view to a card item container (card). A child object can have its position updated or its relationships changed when it is moved from one container to another or to the view.

  By default, this interactor is not enabled. You can enable it either through a button in the CSS configuration file, as follows:

```
ToolBar {
  button[4]: @+button4;
}

Subobject#button4 {
  actionType: "EditObject";
}
```

  or through the API, as follows:

```
IlpEquipment equipment = new IlpEquipment();
equipment.setDefaultViewInteractor(new IlpEditEquipmentObjectInteractor())
;
```

♦ Pan interactor

Allows you to recenter the equipment view, without changing the position of the objects.

- Zoom In interactor

  Increases the zoom factor. Depending on the zoom configuration, the result may vary.

- Zoom Out interactor

  Decreases the zoom factor. Depending on the zoom configuration, the result may vary.

- Reset Zoom interactor

  Sets the current zoom factor back to its default value, regardless of the zoom configuration.

- Fit to Contents interactor

  Adjusts the zoom factor so that all the objects fit in the view. This setting may change the scale of the objects. However, the position and shape of the objects are not affected.

- Zoom on a Rectangle interactor

  Allows you to select a rectangular area and to zoom on the objects it contains, so that they fit in the view.

- Zoom Back interactor

  Allows you to zoom back to the last zoom level.

Each view interactor works with one equipment view only and is managed by the equipment controller. An equipment view can have several interactors, but only one interactor is active at a time.

With the default view interactors, you can:

- associate actions with mouse events and focus events
- associate actions with keyboard events
- define a pop-up menu factory to build a pop-up menu that displays in the view

View interactors have two modes of operation:

- ♦ Transient
- ♦ Permanent

In *transient* mode, the view interactor removes itself from the equipment view when it has performed its action.

In *permanent* mode, the view interactor remains in the view until the controller removes it.

By default, view interactors are permanent.

View interactors can display a pop-up menu.

A view interactor has a context implemented through `IlpViewInteractionContext`. When a user gesture is completed, the equipment view clones this context and makes it accessible through `IlpViewActionEvent`.

The predefined view interactors are in the `package-frame` package and are subclasses of `IlvManagerViewInteractor`. When one of these interactors is installed, it must be wrapped in an `IlpViewsViewInteractor`.

How to wrap a predefined view interactor when Installing It

```
IlvManagerViewInteractor iltInteractor = new Ilt...Interactor();
IlpViewInteractor ilpInteractor =
  new IlpViewsViewInteractor(iltInteractor);
controller.setViewInteractor(ilpInteractor);
```

Setting the view interactor

The method `setViewInteractor(ilog.cpl.interactor.IlpViewInteractor)` allows you to set the view interactor, that is, the object-independent interactor, which is active at a given moment in the view. This interactor is replaced whenever the end user activates a different interactor. Such activation occurs, for example, when the user clicks a toolbar button.

Some interactors are one-shot interactors, that is, they have the attribute `permanent:false` in CSS. When such an interactor finishes its interaction, it is replaced by the default view interactor.

To customize through CSS, refer to *The Interactor rule*.

View interactor and default view interactor

When you use the `setViewInteractor` method you are attaching the specified interactor directly to the view. On the other hand, `setDefaultViewInteractor(ilog.cpl.interactor.IlpViewInteractor)` allows you to define the interactor that will be attached when the current interactor is detached from the view and no other interactor is attached. The default interactor is not attached automatically to the view, so it will not be available immediately.

The following example combines both methods:

```
// Configuring the default view interactor and making
// it active
IltSelectInteractor selInteractor = new IltSelectInteractor();
selInteractor.setEditingAllowed(true);
IlpViewsViewInteractor viewsInteractor =
      new IlpViewsViewInteractor(selInteractor);

equipment.setDefaultViewInteractor(viewsInteractor);
equipment.setViewInteractor(viewsInteractor);
```

In this example, you define the default view interactor through the call to `setDefaultViewInteractor(ilog.cpl.interactor.IlpViewInteractor)`, and you activate it through the call to `setViewInteractor(ilog.cpl.interactor.IlpViewInteractor)` so that it is immediately available.

How to associate an action with a mouse event in the equipment view

You can associate actions with mouse events by using either CSS or the API.

The following extract shows how to customize the default view interactor in CSS:

```
Equipment {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.graphic.views.IlpViewsViewInteractor';
  action[0]: @+viewAction0;
}

Subobject#viewAction0 {
  class: 'ilog.cpl.interactor.IlpGestureAction';
  gesture: BUTTON3_CLICKED;
  action: @+myAction;
}

Subobject#myAction {
  class: MyAction;
}
```

The same configuration can be achieved through the API, as follows:

```
IlpEquipment equipment = // ...

// Retrieve the view interactor
IlpViewInteractor viewInteractor = equipment.getDefaultViewInteractor();
```

```
// Create an actionAction
myAction = new MyAction();

// Clicking the 3rd mouse button will trigger myAction
viewInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED,myAction);
```

You can also customize the actions that are associated with the interactors defined in the equipment component toolbar. This configuration is done with the toolbar button definition. The following CSS extract illustrates how this can be achieved:

How to associate an action with a mouse event for a equipment toolbar button interactor

You can associate actions with mouse events when one of the interactors defined in the equipment component toolbar is active. The following CSS extract illustrates this configuration:

```
Equipment {
  toolbar: true;
}

ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
  action[0]: @+action0;
}

Subobject#action0 {
  gesture: BUTTON1_DOUBLE_CLICKED;
  class: "ShowDetailsAction";
}
```

In this configuration, the action "ShowDetailsAction" is triggered when a double-click event occurs while the selection interactor is set in the equipment view. You can define any list of actions associated with gestures by using the indexed property `action`. To be accepted by the CSS customization, the action class must be a JavaBean™.

You can find out whether this event occurred on an `IlpObject` by means of the following code (which should be in the `MyAction` class):

How to check whether a given action occurred in the equipment view interactor

```
// Implementation of the ActionListener interface
public void actionPerformed(ActionEvent e) {
  // ILOG JTGO interactors use IlpViewActionEvent
  IlpViewActionEvent viewEvent = (IlpViewActionEvent)e;
  // Get the IlpObject (if any) where the interaction occurred
  IlpObject ilpObj = viewEvent.getIlpObject();
  // Perform operation on the given object
}
```

How to associate an action with a keyboard event in the equipment view

You can associate actions with keyboard events by using either CSS or the API.

The following extract shows how to proceed in CSS:

```
Equipment {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.graphic.views.IlpViewsViewInteractor';
  action[0]: @+viewAction0;
}

Subobject#viewAction0 {
  class: 'ilog.cpl.interactor.IlpKeyStrokeAction';
  keyStroke: 'typed D';
  action: @+myAction;
}

Subobject#myAction {
  class: MyAction;
}
```

The same configuration can be achieved through the API, as follows:

```
// Create an actionAction myAction = new MyAction();
// Typing CTRL+D will trigger myAction
viewInteractor.setKeyStrokeAction(KeyStroke.getKeyStroke('D',java.awt.Event.
CTR
L_MASK),myAction);
```

You can also customize the keystroke actions that are associated with the interactors defined in the equipment component toolbar. This configuration is performed with the toolbar button definition. The following CSS extract illustrates how this can be achieved:

How to associate an action with a keyboard event for a equipment toolbar button interactor

You can associate actions with keyboard events when one of the interactors defined in the equipment component toolbar is active. The following CSS extract illustrates this configuration:

```
Equipment {
  toolbar: true;
}

ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
  action[0]: @+action0;
}

Subobject#action0 {
  key: "control A";
  class: "ilog.cpl.graph.action.IlpSelectAllObjectsAction";
}
```

In this configuration, the action "`IlpSelectAllObjectsAction`" is triggered when a Control-A keyboard event occurs while the selection interactor is set in the equipment view. You can define any list of actions associated with keyboard events by using the indexed property `action`. To be accepted by the CSS customization, the action class must be a JavaBean.

How to define a pop-up menu factory for the equipment view

You can customize a pop-up menu factory for the equipment view either through CSS or through the API.

The following extract shows how to add a pop-up menu factory through CSS:

```
Equipment {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.graphic.views.IlpViewsViewInteractor';
  popupMenuFactory: @+viewPopupMenuFactory;
}
```

```
Subobject#viewPopupMenuFactory {
  class: MyPopupMenuFactory;
}
```

The same configuration can be achieved through the API, as follows:

```
// Subclass IlpAbstractPopupMenuFactory, which has useful shortcuts
IlpPopupMenuFactory popupMenuFactory = new IlpAbstractPopupMenuFactory() {
  // Add the identifier of each of the selected objects to the menu
  public JPopupMenu createPopupMenu (IlpObjectSelectionModel ilpSelectionModel)

{
    // Create an empty popup menu
    JPopupMenu menu = new JPopupMenu();
    // Access the selected objects from the selection model
    Collection selectedObjects = ilpSelectionModel.getSelectedObjects();
    // fill the menu according to the current selection
    return menu;

  }
};
```

The following code shows you how to associate the defined pop-up menu factory with the equipment component:

How to associate a pop-up menu factory with the equipment component

```
// Set the popup menu factory to the view interactor
viewInteractor.setPopupMenuFactory(popupMenuFactory);
```

You can also customize a pop-up menu factory that is associated with the interactors defined in the equipment component toolbar. This configuration is performed with the toolbar button definition. The following CSS extract illustrates how this can be achieved:

How to define a pop-up menu factory for a equipment toolbar button interactor

The following CSS extract illustrates this configuration:

```
Equipment {
  toolbar: true;
}

ToolBar {
  enabled: true;
  button[0]: @+SelectButton;
}

Subobject#SelectButton {
  actionType: "Select";
  usingObjectInteractor: true;
  opaqueMove: true;
  popupMenuFactory: @=viewPopupMenuFactory;
}

Subobject#viewPopupMenuFactory {
  class: 'AlarmPopupMenuFactory';
}
```

The pop-up menu factory is customized using property `popupMenuFactory` in the button configuration. To be accepted during the CSS customization, the pop-up menu factory class must be a JavaBean.

Selection interactor

The `IltSelectInteractor` class allows you to select, move, and interact directly with objects. You can:

- ♦ Click an object to select it and deselect all other objects.
- ♦ Use Shift-click to select an unselected object or to deselect a selected object while keeping other prior selected objects selected.
- ♦ Drag a selection rectangle to select all objects within this rectangle.
- ♦ Drag one selected object and thereby cause all other selected objects to move in relation to it.

If `setUsingObjectInteractor(true)` is called on the interactor, then you can also:

- ♦ Use other gestures that are understood by a specific object interactor.

You can configure the selection interactor to use Ctrl-click instead of Shift-click for multiple selection.

How to configure the selection interactor for multiple selection in the equipment view

In CSS, use the following rules:

```
Subobject#SelectButton {
  multipleSelectionModifier: "java.awt.event.InputEvent.CTRL_MASK";
  selectionModifier: "java.awt.event.InputEvent.SHIFT_MASK";
}
```

In the API, use the following code:

```
setMultipleSelectionModifier(InputEvent.CTRL_MASK);
setSelectionModifer(InputEvent.SHIFT_MASK);
```

Group reshape interactor

The `IltEditGroupInteractor` class allows you to change the shape of rectangular, polygonal, and linear groups (`IltGroup`, `IltLinearGroup`, `IltRectGroup` and `IltPolyGroup`). Clicking an object starts shape editing interaction. Clicking the background ends it.

For polygonal and linear groups:

- Dragging a vertex moves it.
- Ctrl-click on a vertex removes it.
- Ctrl-click on an edge adds a vertex at that point on the edge.

Make rectangular node interactor

The `IltMakeRectGroupInteractor` class creates a node with a rectangular shape (`IltRectGroup`). One corner of the rectangle is denoted by the point where the cursor is located when the mouse button is released. Make sure that you do really move the mouse between the time when you press and the time when you release the mouse button. Otherwise, the shape is created empty and the node might be invisible.

Make polygonal node interactor

The `IltMakePolyGroupInteractor` class creates a node with a polygonal shape (`IltPolyGroup`). A point is added each time the user clicks. Double-clicking marks the last point to be added.

Make polyline node interactor

The `IltMakeLinearGroupInteractor` class creates a node with a polyline shape (`IltLinearGroup`). A point is added each time the user clicks. Double-clicking marks the last point to be added.

Make link interactor

The `IltMakeLinkInteractor` class creates links (`IltLink`) between nodes. This interactor works in the following way: the user clicks one node and then goes on dragging the mouse over another node so that the two nodes are selected. When the user releases the mouse, a link is drawn between the two nodes.

Refer to *Interacting with the graphic components* for a detailed description of interactors and gestures.

Interacting with the equipment objects

Interacting with the equipment view describes how to set an interactor on the entire equipment view. You can also associate behavior with business objects (a whole class or individual objects), as well as with individual object instances.

To do so, you use object interactors, which offer you the same possibilities as the view interactor:

- ♦ associating actions with mouse events
- ♦ associating actions with keyboard events
- ♦ defining a pop-up menu factory to build a pop-up menu that displays on representation objects

An object interactor handles any event occurring to the object with which it is associated, provided the view interactor has enabled the use of object interactors. You can check this with the `isUsingObjectInteractor` method or modify it with the `setUsingObjectInteractor` method.

Object interactors are enabled by default.

No default interactor is associated with any object. To associate actions with mouse or keyboard events, or to define a pop-up menu factory, you first have to create an `IlpObjectInteractor`. You can use the `IlpDefaultObjectInteractor`, extend it, or create your own implementation.

How to associate an object interactor with an equipment component object

You can associate an object interactor with a representation object by using either CSS or the API.

The following extract shows how to proceed in CSS:

```
Equipment {
  interactor: true;
}

object."ilog.tgo.model.IltNetworkElement" {
  interactor: @+objInteractor;
}

Subobject#objInteractor {
  class: 'ilog.cpl.interactor.IlpDefaultObjectInteractor';
}
```

The same configuration can be achieved through the API, as follows:

```
IlpEquipment equipment = // ...
IlpEquipmentController equipmentController = equipment.getController();
```

```
// Create an object interactor
IlpObjectInteractor objectInteractor = new IlpDefaultObjectInteractor();
equipmentController.setObjectInteractor( bo, objectInteractor);
// Configuring the specific object interactor is similar to configuring
// a view interactor.
objectInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED, new MyAction())
;
```

Actions related to mouse and keyboard events can be customized in the same way as for the view interactor. You can also define a pop-up menu factory in the same way as for the view interactor. Refer to *Interacting with the equipment view*.

An object interactor can also be associated with a specific decoration that is part of the business object graphic representation in the equipment view. Each decoration represents a business attribute in the model. Therefore the customization of the interactor for a specific decoration takes into account the business object and a business attribute as illustrated below:

How to associate an object interactor with the label decoration in an equipment component object

You can associate an object interactor with one of the graphic decorations of the object by setting the interactor to the business attribute that is represented. You can do it using CSS or the API.

The following extract shows how to proceed in CSS:

```
Equipment {
  interactor: true;
}

object."ilog.tgo.model.IltNetworkElement/name" {
  interactor: @+objInteractor;
}

Subobject#objInteractor {
  class: 'ilog.cpl.interactor.IlpDefaultObjectInteractor';
}
```

The same configuration can be achieved through the API, as follows:

```
IlpEquipment equipment = // ...
IlpEquipmentController equipmentController = equipment.getController();
// Create an object interactor
IlpObjectInteractor objectInteractor = new IlpDefaultObjectInteractor();
equipmentController.setObjectInteractor( bo, IltObject.NameAttribute,
objectInteractor);
// Configuring the specific object interactor is similar to configuring
// a view interactor.
objectInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED,   new
MyAction());
```

Actions related to mouse and keyboard events can be customized in the same way as for the view interactor. You can also define a pop-up menu factory in the same way as for the view interactor. Refer to *Interacting with the equipment view*.

For a detailed description of interactors and gestures, refer to *Interacting with the graphic components* .

Positioning

Each object has a position value, which indicates where it will be displayed.

The position is defined by the `IlpPosition` interface which has the following predefined implementations:

♦ `IlpPoint` for shelves placed in the component

♦ `IlpShelfItemPosition` for cards and card carriers placed in the component

♦ `IlpRelativePoint` for ports and LEDs placed inside a card

♦ `IlpShelfItemPosition` for cards and card carriers placed inside a shelf or card carrier

Both `IlpRelativePoint` and `IlpShelfItemPosition` are relative positions. For more information, see *Relative positioning*.

The position value can originate from the back end as an attribute of the corresponding business object, or from separate data storage through the equipment component API, or from explicit end-user interactions to move objects in the component.

Relative positioning

An equipment object has a hierarchical structure: a shelf can contain cards, which can contain ports and LEDs, thus defining parent-child relationships. When positioning objects as children of other objects, relative positioning is applied instead of regular positioning.

Regular positioning applies to root objects (parents), and is based on the component; it may change from one application to another. In contrast, relative positioning applies to child objects and is based on the position of the parent object; this relative position remains the same, regardless of the position of the parent in the component.

The supported parent objects are the following:

♦ Shelves

  Business objects of the class `IltShelf`. The positioning of child objects is based on slots.

♦ Card carriers

  Business objects of the class `IltCardCarrier`. The positioning of child objects is based on slots.

  Card carriers are a special type of card that can contain other cards. Like a regular card, it can be placed inside a shelf or another card carrier.

♦ Cards

  Business objects of the class `IltCard`. The positioning of child objects is based on (x,y) coordinates.

Shelves and card carriers

Shelves and card carriers are containers for card objects that are positioned based on the available slots. A card can occupy one or more slots in the container.

To position a card inside a shelf or a card carrier, use the `IlpShelfItemPosition` class (instance of `IlpPosition`). This class allows you to define the slot index and spanning, so that the card can spread over more than one slot.

The class `IlpShelfItemPosition` has four attributes:

♦ `xIndex` defines the `x` coordinate of the slot.

♦ `yIndex` defines the `y` coordinate of the slot.

♦ `xSpan` defines the spanning over `x`.

♦ `ySpan` defines the spanning over `y`.

For the special case of card carriers, `yIndex` is always equal to `0` and `ySpan` is always equal to `1.0`.

Shelves support an array of slots, referenced by two indices (`xIndex` and `yIndex`), while card carriers support slots referenced by a single index (`xIndex`). The spanning attributes of `IlpShelfItemPosition` define by how much the card spans over the neighboring slots:

a value of `1.0` means no spanning at all; a value of `2.0` means that the next slot is fully occupied; a value of `1.5` means that the next slot is partially occupied (50%).

Note: One slot cannot hold more than one object, even if it is only partially occupied.

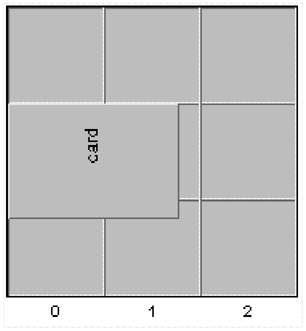
The following code gives an example of relative positioning of cards inside a shelf. (It assumes that a data source is connected to the equipment component.)

How to perform relative positioning of a card in a shelf

```
// Creates a shelf business object
IltShelf shelf = new IltShelf(3, 60, 3, 60, 0);
// Sets its view position to point (50, 50)
shelf.setPosition(new IlpPoint(50, 50));

// Creates a card business object
IltCard card = new IltCard((IltObjectState)null, "card");
// Sets its relative position to index x = 0, spanning over by 1.8
// and index y = 1, spanning over by 1.2.
card.setPosition(new IlpShelfItemPosition(0, 1, 1.8f, 1.2f));

// Sets the relationship between card and shelf
dataSource.setParent(card, shelf);
```

```
// Adds objects to the data source
dataSource.addObject(shelf);
dataSource.addOject(card);
```

The graphic result looks like this:



Card positioned inside a shelf

The XML file corresponding to *Card positioned inside a shelf* is the following:

```
<cplData xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/data.xsd">
 <addObject id="shelf" container="true">
  <class>ilog.tgo.model.IltShelf</class>
  <attribute name="slotSizes" javaClass="ilog.cpl.equipment.IlpSlotSizes">
    <width>
      <value>60</value>
      <value>60</value>
      <value>60</value>
    </width>
    <height>
      <value>60</value>
      <value>60</value>
      <value>60</value>
    </height>
  </attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpPoint">
  <x>50</x>
  <y>50</y>
  </attribute>
 </addObject>
 <addObject id="card" container="true">
  <class>ilog.tgo.model.IltCard</class>
  <parent>shelf</parent>
  <attribute name="name">card</attribute>
  <attribute name="position"
  javaClass="ilog.cpl.graphic.views.IlpShelfItemPosition">
    <xIndex>0</xIndex>
    <yIndex>1</yIndex>
    <xSpan>1.8</xSpan>
    <ySpan>1.2</ySpan>
  </attribute>
 </addObject>
</cplData>
```

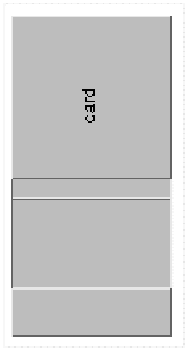
The following code gives an example of relative positioning of cards inside a card carrier. (It assumes that a data source is connected to the equipment component.)

How to perform relative positioning of a card in a card carrier

```
// Creates a card carrier business object
IltCardCarrier carrier = new IltCardCarrier((IltObjectState)null, 3);
// Sets its position and shape
carrier.setPosition(new IlpShelfItemPosition(50, 50, 100, 200));

// Creates a card business object
IltCard card = new IltCard((IltObjectState)null, "");
// Sets its relative position to index x = 0, spanning over by 1.8
card.setPosition(new IlpShelfItemPosition(0, 0, 1.8f, 0));

// Sets the relationship between card and card carrier
dataSource.setParent(card, carrier);
```

```
// Adds objects to the data source
dataSource.addObject(carrier);
dataSource.addOject(card);
```

The graphic result looks like this:



Card positioned inside a card carrier

The XML file corresponding to *Card positioned inside a card carrier* is the following:

```
<cplData xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/data.xsd">
 <addObject id="carrier" container="true">
  <class>ilog.tgo.model.IltCardCarrier</class>
  <attribute name="slotCount">3</attribute>
  <attribute name="position"
           javaClass="ilog.cpl.graphic.views.IlpShelfItemPosition">
    <x>50</x>
    <y>50</y>
    <width>100</width>
    <height>200</height>
  </attribute>
 </addObject>
 <addObject id="card" container="true">
  <class>ilog.tgo.model.IltCard</class>
  <parent>carrier</parent>
  <attribute name="name">card</attribute>
  <attribute name="position"
  javaClass="ilog.cpl.graphic.views.IlpShelfItemPosition">
    <xIndex>0</xIndex>
    <yIndex>0</yIndex>
    <xSpan>1.8</xSpan>
    <ySpan>0</ySpan>
  </attribute>
 </addObject>
</cplData>
```

Cards

Cards are containers for ports and LEDs. The top left corner of a card defines the origin for the relative positioning of ports and LEDs on the card. In other words, the child objects are placed at (x,y) pixels from the top left corner of the card.

This positioning system is dependent on the direction of the card. Usually, cards are oriented in a direction set to `top`. If this direction was set to `right`, the top right corner of the card would become the origin for positioning the child objects. If the card was oriented in the direction `bottom`, the (x,y) origin would be the lower right corner.

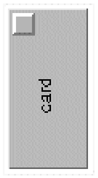
The `IlpRelativePoint` class, an instance of `IlpPosition`, is used to position a port or LED inside a card. `IlpRelativePoint` defines two attributes corresponding respectively to the horizontal (`x`) and vertical (`y`) distance from the container (x,y) origin (which has the position `(0,0)`).

The following code gives an example of the relative positioning of a port or LED inside a card. (It assumes that a data source is connected to the equipment component.)

How to perform relative positioning of a port or LED in a card

```
// Creates a card business object
IltCard card = new IltCard((IltObjectState)null, "card");
// Sets its position and shape
card.setPosition(new IlpShelfItemPosition(50, 50, 50, 100));

// Creates a LED business object
IltLed led = new IltLed("", IltLed.Type.Rectangular);
// Sets its relative position, which is (10, 10) from the xy origin
led.setPosition(new IlpRelativePoint(10, 10));

// Sets the relationship between LED and card
dataSource.setParent(led, card);

// Adds objects to the data source
dataSource.addObject(card);
dataSource.addObject(led);
```

The graphic result looks like the following figure.



LED positioned inside a card

The XML file corresponding to this figure is as follows.

```
<cplData xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/data.xsd">
 <addObject id="card" container="true">
  <class>ilog.tgo.model.IltCard</class>
  <attribute name="name">card</attribute>
  <attribute name="position"
          javaClass="ilog.cpl.graphic.views.IlpShelfItemPosition">
    <x>50</x>
    <y>50</y>
    <width>50</width>
    <height>100</height>
  </attribute>
 </addObject>
 <addObject id="led">
  <class>ilog.tgo.model.IltLed</class>
  <parent>card</parent>
  <attribute name="type">Rectangular</attribute>
  <attribute name="position" javaClass="ilog.cpl.graphic.IlpRelativePoint">
    <x>10</x>
    <y>10</y>
  </attribute>
 </addObject>
</cplData>
```

Layout

JViews TGO makes use of the IBM® ILOG® JViews graph layout algorithms. Each `IlpEquipmentView` can be connected to several node algorithms and one link algorithm.

- The *node layout* algorithms are:
  - `IlvBusLayout`
  - `IlvCircularLayout`
  - `IlvHierarchicalLayout`
  - `IlvRandomLayout`
  - `IlvSpringEmbedderLayout`
  - `IlvTopologicalMeshLayout`
  - `IlvTreeLayout`
  - `IlvUniformLengthEdgesLayout`
- The *link layout* algorithms can be:
  - `IlvLinkLayout`
  - `IlvShortLinkLayout`
  - `IlvLongLinkLayout`
  - `IltLinkLayout`
  - `IltShortLinkLayout`
  - `IltLocalLinkLayout`
  - `IltStraightLinkLayout`

Note: The difference between Ilv... and Ilt... link layout algorithms is that Ilt... algorithms support connection ports whereas Ilv... algorithms don’t.

The detailed description of all the graph layout algorithms can be found in the IBM® ILOG® JViews *Diagrammer Using Graph Layout Algorithms* documentation.

In case of multiple layouts, one layout can be set to be applied automatically whenever the contents of the view changes, while the others can be applied on demand.

To configure the layouts, it is recommended to use CSS (see *Configuring an equipment component through CSS*). Using CSS, you can also configure per-object layout parameters.

If a layout takes too much time to execute, or if you want to add toolbar buttons to execute a layout, you can configure the layout for the view, then execute it on demand by using the

API method `performAttachedLayout(int)`. The advantage of this method over the method `performLayoutOnce(ilog.views.graphlayout.IlvGraphLayout)` is that the layout remains attached to the view, therefore storing any previously-defined configuration.

The class `IlpEquipmentView` provides the following methods to handle the layout operation:

- void setNodeLayout (`IlvGraphLayout` layout, boolean perform). This method sets the given layout as the default for this `IlpEquipmentView`. If the `perform` parameter is set to `true`, the layout is applied to the objects immediately. With this method the layout is executed every time the equipment content changes.
- void `setLinkLayout (IlvGraphLayout layout, boolean perform)`. This method sets the given layout as the default link layout for this instance of `IlpEquipmentView`. If the `perform` parameter is set to `true`, the layout is applied to the links immediately. With this method the layout is executed every time the equipment content changes.
- `public void performLayoutOnce(IlvGraphLayout layout)`. This method executes the layout algorithm once on the manager content.
- `void startDelayingUpdates()`. This method suspends temporarily the layout operations. This mechanism avoids unnecessary computation when you intend to perform a sequence of operations that affect the equipment layout.
- `void endDelayingUpdates()`. This method resumes the layout operations suspended by a call to the method `startDelayingUpdates`. Any operation that requests a layout recalculation is suspended when it is executed between `startDelayingUpdates` and `endDelayingUpdates` calls.
- `void setGraphLayouts(IlvGraphLayout[] layouts)`. This method sets the given graph layouts for this `IlpEquipmentView`. Several graph layouts can be set to position nodes in the view. One of them can be configured to be executed every time the equipment contents changes. This method does not apply the layout to the nodes immediately. All the graph layouts given as argument to the method are attached to the view.
- `void setGraphLayouts(int index, IlvGraphLayout layout)`. This method sets a new graph layout for the `IlpEquipmentView` or replaces an existing graph layout. This method does not apply the layout to the nodes immediately.
- `IlvGraphLayout[] getGraphLayouts()`. This method returns the graph layouts that have been configured for the view.
- `IlvGraphLayout getGraphLayouts(int index)`. This method returns the graph layout that is configured for the view at the given index.
- `void setAutoLayoutIndex (int index)`. This method indicates, from the list of graph layouts that have been configured using method `setGraphLayouts`, which one is executed automatically when the contents of the view changes.
- `int getAutoLayoutIndex()`. This method returns the index of the graph layout that is executed automatically when the contents of the view changes.
- `void performAttachedLayout(int index)`. This method executes the layout that has been configured for the given index, recursively in the object tree. The layout has been already attached to the view and keeps the configuration whenever it is performed.

- `IlvGraphic getLayoutProxy(IlpRepresentationObject)`. This method returns the graphic object corresponding to the given representation object for layout purposes. This method should be used if you need to set per-object layout properties to configure the layout algorithms.

Note: Graphical parameters, such as the layout region, that are passed to the graph layout are expressed in view coordinates. Therefore, if you have expressed these parameters in stationary coordinates, you must transform them to view coordinates (by applying `equipment.getView().getCompositeGrapher().getZoomTransformer()` `equipment.getManagerView().getTransformer()`) before passing them to the graph layout.

How to use hierarchical node layout in the equipment component

In CSS, use the following rules:

```
Equipment {
  graphLayout: true;
}

GraphLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Bottom;
  levelJustification: Top;
  globalLinkStyle:
'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout.POLYLINE_STYLE';
  connectorStyle:
'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout.EVENLY_SPACED_PINS';
}
```

The full class path is required for the properties `globalLinkStyle` and `connectorStyle` for the type converter to locate and convert the constants.

For an example of a CSS link layout, refer to *The LinkLayout rule* .

In the API, use the following code:

```
IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
layout.setFlowDirection (IlvDirection.Bottom);
layout.setLevelJustification (IlvDirection.Top);
layout.setGlobalLinkStyle (IlvHierarchicalLayout.POLYLINE_STYLE);
layout.setConnectorStyle (IlvHierarchicalLayout.EVENLY_SPACED_PINS);

equipment.setNodeLayout(layout);
```

Note: All layouts to be used with JViews TGO must be set in view coordinate mode. This mode is automatically set when you install layouts through `setNodeLayout(ilog.views.graphlayout.IlvGraphLayout)`, `setLinkLayout(ilog.views.graphlayout.IlvGraphLayout)`, or `performLayoutOnce(ilog.views.graphlayout.IlvGraphLayout)`. If you call the method `performLayout()` directly, you must first set the mode: `layout.setCoordinatesMode(IlvGraphLayout.VIEW_COORDINATES);`

Even when you decide to use a certain node or link layout, you may want some links or nodes to be *pinned*; that is, you may want to keep a certain element in a specified position that is not affected when the layout is executed on an equipment.

You can achieve this effect by using the following methods defined in `IlvGraphLayout`:

- `setPreserveFixedLinks (boolean preserve)`. This method determines whether or not the layout will preserve the position of the registered links.
- `setPreserveFixedNodes (boolean preserve)`. This method determines whether or not the layout will preserve the position of the registered nodes.
- `void setFixed (Object obj /* link or node */, boolean fix)`. This method determines whether or not the given object will be fixed in the equipment view.
- `boolean isFixed (Object obj)`. The return value indicates whether or not the given object is marked to be fixed. The object to be passed is an `IlvGraphic` belonging to the `IlpGraphic` that represents the object and *not* the `IlpGraphic` or `IlpRepresentationObject` itself.
- `void unfixAllLinks()`
- `void unfixAllNodes()`

How to set a link to fixed shape and a node to fixed position in the equipment view

The following code illustrates how you can set a given link to have a fixed shape and a given node to have a fixed position in the equipment view.

In CSS, use the following rules:

```
Equipment {
  graphLayout: true;
  linkLayout: true;
}

LinkLayout {
  class: 'ilog.views.graphlayout.link.IlvLinkLayout';
}

GraphLayout {
  layouts[0]: @+treeLayout;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
}

#Link:linkLayoutRenderer {
  fixed: true;
}

#NE1:graphLayoutRenderer:tree {
  fixed: true;
}
```

In the API, use the following code:

```
IlvGraphLayout layout = equipmentView.getLinkLayout();
layout.setPreserveFixedLinks (true);
IlpRepresentationObject linkRO =
equipmentAdapter.getRepresentationObject(link);
layout.setFixed(equipmentView.getLayoutProxy(linkRO));

layout = equipmentView.getNodeLayout();
layout.setPreserveFixedNodes (true);
IlpRepresentationObject neRO = equipmentAdapter.getRepresentationObject(ne);
layout.setFixed(equipmentView.getLayoutProxy(neRO));
```

When the position or shape of an object is not handled by the layout, you must set it by calling the method `setPosition(ilog.cpl.model.IlpRepresentationObject, ilog.cpl.graphic.IlpPosition, ilog.cpl.graphic.IlpPositionSource)` (or `view.setPosition`).

JViews TGO provides a default layout which uses `IlvShortLinkLayout` to shape and position links. This default layout sets all objects without an attached position to `(0,0)`.

How to use multiple node layouts in an equipment view

In this scenario, two node layouts are configured for the equipment view. The first one is configured to be executed automatically.

In CSS, use the following rules:

```
Equipment {
  graphLayout: true;
}

GraphLayout {
  layouts[0]: @+hierarchicalLayout;
  layouts[1]: @+treeLayout;
  autoLayoutIndex: 0;
}
```

```
Subobject#hierarchicalLayout {
  class: 'ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout';
  flowDirection: Bottom;
  levelJustification: Top;
  globalLinkStyle: POLYLINE_STYLE;
  connectorStyle: EVENLY_SPACED_PINS;
}

Subobject#treeLayout {
  class: 'ilog.views.graphlayout.tree.IlvTreeLayout';
  flowDirection: Bottom;
}
```

To execute the tree layout in the view, use the method `performAttachedLayout`, where the index is the one defined in the CSS configuration, as follows:

```
equipment.getView().performAttachedLayout(1);
```

How to configure per-object layout properties in the equipment component

Some layout algorithms require a specific configuration in order to be properly executed. For example, the bus layout needs to have a bus object specified; or the tree layout, for which you may want to specify the root node prior to the layout execution. Starting from JViews TGO 7.5, you can configure these properties using CSS as follows:

```
Equipment {
  graphLayout: true;
}

GraphLayout {
  layouts[0]: @+busLayout
}

Subobject#busLayout {
  class: 'ilog.views.graphlayout.bus.IlvBusLayout';
  horizontalOffset: 50;
  verticalOffsetToLevel: 50;
  verticalOffsetToPreviousLevel: 40;
  margin: 30;
  marginOnBus: 50;
}

// Configure the bus object as the bus in the layout
// All layout configuration uses the 'graphLayoutRenderer'
// and the graph layout name pseudoclasses
#BUS:graphLayoutRenderer:bus {
  bus: true;
}

// Configure the bus to route the links that connect the
// bus to the nodes
#BUS {
  linksConnectToBase: true;
}
```

Alternatively, you can configure these properties using the API as follows:

```
IlvBusLayout busLayout = new IlvBusLayout();
equipment.setGraphLayouts(new IlvGraphLayout[] { busLayout });

IlpRepresentationObject ro =
equipment.getAdapter().getRepresentationObject("BUS");
IlvGraphic layoutProxy = equipment.getView().getLayoutProxy(ro);
busLayout.setBus((IlvPolyPointsInterface)layoutProxy);
```

How to disable the per-object layout properties configuration in the equipment view

By default, per-object layout parameters can be configured using CSS. However, if you are not interested in this feature, you can disable it by setting the property `useParObjectParamenters` in the graph layout renderer and link layout renderer. Disabling the per-object layout properties configuration speeds up significantly the rendering process.

```
Equipment {
  graphLayout: true;
  linkLayout: true;
}

LinkLayout {
  class: 'ilog.views.graphlayout.link.IlvLinkLayout';
  usePerObjectParameters: false;
}

GraphLayout {
  layouts[0]: @+treeLayout;
  usePerObjectParameters: false;
}
```

Zooming

There are three different zoom modes for the equipment component: physical, logical, and mixed. For more information on zoom modes, see *Zooming* in section *Network component*.

Background support

A background map can be used to create more realistic views of equipment and to have a more detailed look at equipment objects. For more information on background support, see *Background support* in *Network component*.

Filtering

The equipment component allows you to filter the nodes that are displayed. To do so, attach an instance of `IlpFilter` to the equipment component by using the method `setFilter`. The `accept(java.lang.Object)` method of the filter object will be invoked whenever the equipment is prompted to display an `IlpObject`. If the method returns `false`, the object will not be shown in the equipment view. In the same way, an object will not be shown if its parent is not displayed.

For example, write the following code to show only objects of the class `IltShelf`.

How to filter objects to be shown in the equipment component

```
IlpEquipment equipment = // ...
// Create a new IlpFilter instance
IlpFilter filter = new IlpFilter(){
  // This method is called for every object in the data source
  public boolean accept (Object object){
    IlpObject ilpObject = (IlpObject)object;
    IlpClass clz = ilpObject.getIlpClass();
    // Check if the class == IltShelf
    return clz.equals(IltShelf.GetIlpClass());
  }
// Set the filter to the equipment
equipment.setFilter(filter);
```

All the objects are refiltered whenever a new filter is set. If the filter is null (which is the default), all the objects under the root nodes will be displayed.

To retrieve the active filter, use the method `getFilter`.

Note: The filtering takes actually place at the adapter level.

To see how to configure filtering through CSS, refer to *The Adapter rule* .

Accepted and excluded classes

You can specify the business objects that will be represented or not in the equipment component depending on their business classes. To do so, you need to specify the business classes to be accepted or excluded using methods `setAcceptedClasses` or `setExcludedClasses` in the equipment component adapter. To retrieve the adapter, use the `getAdapter` method. The adapter must be an instance of a subclass of `IlpAbstractNodeAdapter`. By default, business objects of the class `IltAlarm` are excluded from the equipment component, so that alarm objects in the data source are only used to compute the alarms present in a given managed entity instead of being graphically represented in the view.

How to specify excluded classes in the equipment component

You can specify that business objects from specific business classes are not represented in the equipment component. You can do that using the API, `setExcludedClasses(java.util.List)` method, or using CSS.

The following example shows you how to prevent objects from business classes `IltAlarm` and `IltLed` to be represented:

```
Adapter {
  excludedClasses[0]: "ilog.tgo.model.IltAlarm";
  excludedClasses[1]: "ilog.tgo.model.IltLed";
}
```

How to specify an accepted class in the equipment component

By default, all business classes, except `IltAlarm`, are accepted by the equipment component. If you want to specify exactly which business classes to represent, you should combine the list of excluded and accepted classes, so that you exclude all business classes except those that are marked in the accepted class list.

In the following example, the equipment component is configured in a way that it graphically represents only business objects from the class `IltNetworkElement`.

```
Adapter {
  excludedClasses[0]: "ilog.tgo.model.IltAlarm";
  excludedClasses[0]: "ilog.tgo.model.IltObject";
  acceptedClasses[0]: "ilog.tgo.model.IltNetworkElement";
}
```

Note: The filtering that is performed through the use of the accepted and excluded class lists takes actually place at the adapter level.

To see how to configure excluded and accepted classes through CSS, refer to *The Adapter rule* .

Setting a list of origins

By default, all the objects in the data source that do not have a parent are treated as root nodes by the equipment component. However, you can explicitly select the root nodes to be displayed through the adapter that forms a bridge between the data source and the equipment component. To retrieve the adapter, use the `getAdapter()` method. The adapter must be an instance of a subclass of `IlpAbstractNodeAdapter`.

The root nodes can be changed by modifying the list of *origins* for the adapter. These origins are set and retrieved as `IlpObject` identifiers. The equipment adapter has two options: either the origins are represented as root nodes, or they are hidden and their child objects are represented as root nodes.

The method `getOrigins` allows you to get the list of current origins. The method `isShowingOrigin` indicates whether the origins themselves or their child objects are represented as root nodes. By default, the list of origins is empty and the origins are not shown, which means that all objects without a parent are shown as root nodes. Thus, the entire contents of the data source are displayed in the equipment.

Note: The origins are specified using identifiers, not the `IlpObject` instances. You can retrieve the identifier of an `IlpObject` with the `getIdentifier` method of the object.

To change the list of origins, use the `setOrigins` method. This method takes a list of business object identifiers as its first parameter. Its second parameter is a Boolean flag that indicates whether or not the origins themselves should be shown as root nodes.

Calling this method with an empty list and the second parameter set to `true` empties the equipment:

```
setOrigins(Collections.EMPTY_LIST, true);
```

Calling the method with an empty list and the second parameter set to `false` restores the default; that is, all the objects in the data source are shown:

```
setOrigins(Collections.EMPTY_LIST, false);
```

How to show an object as the root node of an equipment

To show only a given `IlpObject` as the root node of an equipment, use the following code:

```
IlpEquipment equipment = ....;
IlpObject originObject = .....;
java.util.List originList = new ArrayList();
originList.add(originObject.getIdentifier());
equipment.getAdapter().setOrigins(originList, true);
```

See the `IlpAbstractHierarchyAdapter` class for additional methods to help you manage origins.

To know how to configure origins through CSS, refer to *The Adapter rule* .

Node factory

The equipment adapter converts business objects retrieved from the associated data source into instances of `IlpEquipmentNode`. The new representation objects are created by a representation object factory. By default, the equipment adapter uses `IlpDefaultEquipmentNodeFactory`, which creates representation objects of type `IlpDefaultEquipmentNode`.

To see how to configure an equipment node factory through CSS, refer to *The Adapter rule* .

Link factory

As the equipment node factory transforms business objects into representation objects, the link factory transforms business objects that are links into representation objects that are instances of `IlpEquipmentLink`. The equipment adapter uses by default the `IlpDefaultEquipmentLinkFactory` that creates representation objects of type `IlpDefaultEquipmentLink`.

To see how to configure an equipment link factory through CSS, refer to *The Adapter rule* .

Expansion strategy

The equipment adapter uses an *expansion strategy* to identify whether objects should be loaded or not in the equipment model. The expansion strategy defines how an object is going to behave when it is expanded, for example, when the user opens an equipment node by double-clicking or by using the equipment expansion handles. By default, equipment objects are configured to expand their child objects in place; for example, shelves and cards are automatically expanded. The expansion strategy indicates whether load on demand is implemented and provides methods to load and release child nodes.

The equipment adapter uses an *expansion strategy factory* to decide the expansion strategy to apply to an equipment node when it is created by the adapter. The default expansion strategy factory implementation, `IlpDefaultNodeExpansionStrategyFactory`, checks the property "`expansion`" of each business object in the cascading style sheet loaded in the component to identify the expansion strategy to use.

The default equipment expansion strategy factory supports three types of expansion strategies:

♦ `IN_PLACE`: loads the child objects immediately in the equipment model. In this expansion strategy, nodes are considered as parent nodes only when they have containment relationships defined in the attached data source, through the `IlpContainer` interface. The child objects should already be loaded in the data source and should be visible according to the data source filter, if there is one defined.

♦ `IN_PLACE_MINIMAL_LOADING`: loads the child objects on demand in the equipment model, that is, as the user expands the parent nodes. All nodes with this expansion strategy are considered as possible parent nodes, and therefore are represented with an expansion icon. If the node does not contain child objects, the expansion icon disappears when the expansion is executed for the first time.

♦ `NO_EXPANSION`: expansion is not supported by the node.

See Customizing the expansion of business objects in the *Styling* documentation for information on how to customize the business object expansion type, which is defined by the property `expansion`.

The expansion strategy factory can be customized for the adapter either through CSS or through the API.

Architecture of the equipment component

Describes the classes and features of the equipment component that are specific to each module of the MVC architecture, and also explains the role of the adapter.

In this section

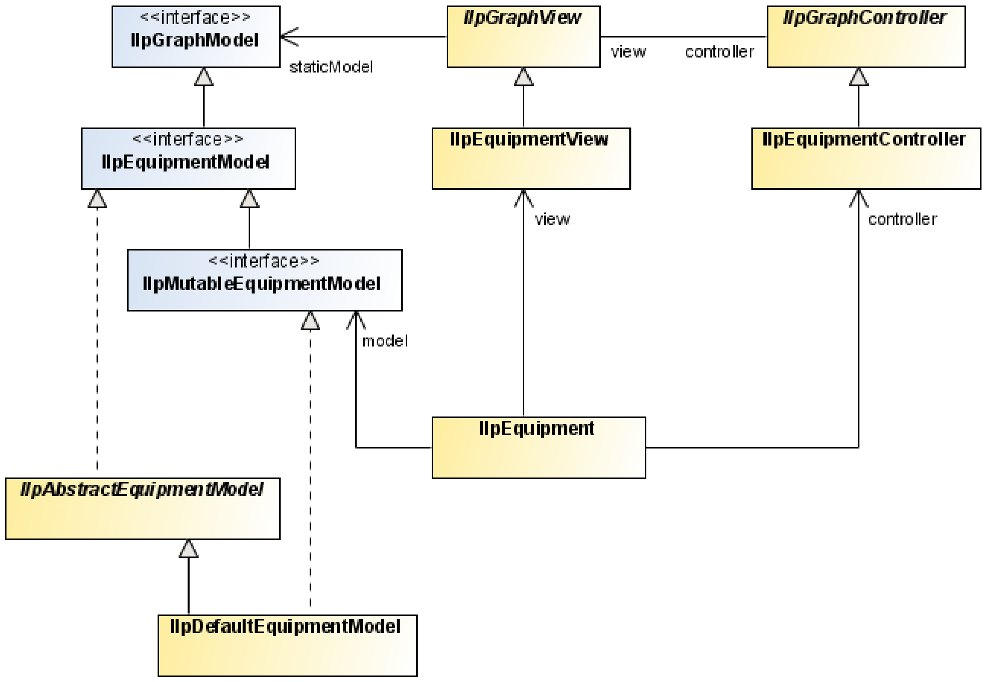
Class overview

A graphic component encapsulates a model, a view, and a controller. The equipment component, like all the other graphic components, is based on the MVC architecture, which means that it has a model, a view and a controller associated with it. For a general introduction to the MVC architecture, see *Architecture of graphic components*.

The MVC architecture for the equipment component is implemented by the following classes:

- The class `IlpEquipment` contains a model, a view, and a controller.
- The model interfaces are `IlpEquipmentModel` and `IlpMutableEquipmentModel`. `IlpMutableEquipmentModel` provides API facilities for adding and removing objects. The two equipment model interfaces are implemented through the class `IlpDefaultEquipmentModel`. Instances of `IlpEquipment` use the class `IlpDefaultEquipmentModel`.

  The representation objects to be displayed in the equipment component must be added to the model. The equipment model recognizes the following objects: nodes, and links that connect nodes. The representation objects to be added to the model must implement the `IlpEquipmentNode` or `IlpEquipmentLink` interface. Concrete implementations of these interfaces are provided as `IlpDefaultEquipmentNode` and `IlpDefaultEquipmentLink`. You can create subclasses of these representation object classes.
- The class `IlpEquipmentView` defines the equipment view that is automatically instantiated when an equipment object (an instance of `IlpEquipment`) is created.
- The class `IlpEquipmentController` creates a controller that manages interactions between the user and an `IlpEquipmentView`.



Main classes used by the equipment component

The model

The model in the equipment component contains and manages the representation objects, that is, the objects that are used to represent equipment business objects in JViews TGO (typically shelves and cards). The representation objects will in turn be converted to graphic objects displayed in the equipment view. Therefore, the equipment view needs to have access to the model to be able to render and graphically display the representation objects.

The model sends notifications for every single change in the representation node hierarchy, that is, for added or removed root and child nodes.

Classes of the equipment model

The model is made up of two interfaces:

- `IlpEquipmentModel`: Provides methods to access the model.
- `IlpMutableEquipmentModel`: Provides methods to set and update the model.

The default implementation of both interfaces is the class `IlpDefaultEquipmentModel`, which is automatically instantiated when the equipment component is created.

The model stores and maintains information about the representation objects, which may be added to the model either through the API or by an adapter. The adapter converts the business objects of the data source to representation objects. (An adapter is created automatically when a data source is connected to the equipment component.)

When a representation object is added, removed, or updated in the model, events of the `EquipmentModelEvent` class are sent to all registered listeners. The view itself is a listener to equipment model events, so that it is always synchronized with the contents of the model. To receive model events, you must implement the `EquipmentModelListener` interface and register your implementation with the model through the `addEquipmentModelListener` method.

When a data source is attached to the equipment component, the model listens to changes affecting the business objects; it updates the representation objects accordingly and fires model events to all is listeners.

Using the equipment model

The whole model API can be accessed through the `IlpEquipment. getModel()` method, but `IlpEquipment` also provides shortcut methods to access some model services.

How to retrieve the model root objects

To retrieve the root objects that are currently set in the equipment model, use the `IlpEquipmentModel` API, as follows:

```
IlpEquipment equipmentComponent = ...; // the equipment component
IlpMutableEquipmentModel model = equipmentComponent.getModel();
Collection roots = model.getRootObjects();
```

When a change occurs in the structure of the model (representation objects), a notification is fired to all listeners registered within the equipment model. In the case of multiple changes, it is more effective to store the changes in a buffer, thus postponing the internal processing of notifications. The method `startBatch()` allows you to execute a series of operations at once in the equipment component. Then, when `endBatch()` is called, all the notifications generated by these operations will be sent to the model and its listeners.

Note: It is your responsibility to manage both `startBatch()` and `endBatch()` methods. In other words, when the method `startBatch` is issued, it has to be followed by the method `endBatch`.

The following example shows how to batch a series of changes in the equipment model

How to manage notification in batch mode

```
// starts notification bufferization
IlpDefaultDataSource datasource = (IlpDefaultDataSource)
equipment.getDataSource();
datasource.startBatch();
datasource.addObject(newRoot1);
datasource.addObject(newRoot2);
datasource.addObject(newRoot3);
// ends notification bufferization
dataSource.endBatch();
```

In this example, implicit notifications arising from additions or removals of child objects will also be buffered.

To add a listener to the model, you must implement the interface `EquipmentModelListener` and register it with the model by using the method `addEquipmentModelListener(ilog.cpl.equipment.EquipmentModelListener)`. Conversely, to remove a listener from the model, use the method `removeEquipmentModelListener(ilog.cpl.equipment.EquipmentModelListener)`.

The listener is based on the `EquipmentModelEvent` class, which defines the equipment model events. It is divided into four different types of events: `ROOT_OBJECT_ADDED`, `ROOT_OBJECT_REMOVED`, `CHILDREN_ADDED` and `CHILDREN_REMOVED`. Each type of event corresponds to a specific update. There are two special types of events, `SERIES_BEGIN` and `SERIES_END`, used to delimit a series of notifications sent after the model has completed buffered changes. Right after the method `endBatch()` has been issued, an event of type `SERIES_BEGIN` is sent to the listeners indicating the beginning of buffered changes, followed by all the buffered events. Then, an event of type `SERIES_END` is sent, which indicates the end of buffered notifications.

The view

The view displays a rectangular surface of a theoretically infinite plane area. The view performs the following functions:

- ♦ Displays a subset of the objects of the model
- ♦ Allows navigation using scrollbars and provides a zoom facility
- ♦ Assigns a default position to nodes that have no value for position in the model
- ♦ Assigns a shape to links that have none
- ♦ Modifies link shapes when end nodes are moved
- ♦ Provides a toolbar for choosing a view interactor
- ♦ Optionally displays an overview window

Classes of the equipment view

The `IlpEquipment`class automatically creates the concrete class `IlpEquipmentView` that is provided for developing the equipment view. This class provides methods that allow you to configure the view in the following ways:

- ♦ Turn the scrollbars on or off: `setHorizontalScrollBarVisible(boolean)`, `setVerticalScrollBarVisible(boolean)`
- ♦ Set the zoom level: `getManagerView()`
- ♦ Set the toolbar on or off (here set to off): `setToolBarVisible(boolean)` (false)
- ♦ Set the overview window to be displayed or not (here set to be displayed): `setOverviewVisible(boolean)`

For convenience, these methods are also accessible from the class `IlpEquipment`. They delegate to `IlpEquipmentView`.

See *Configuring an equipment component through CSS* for how to configure the equipment view in CSS.

See *Configuring an equipment component through the API* for how to configure the equipment view through the API.

Graphic objects in the equipment component

Graphic objects display as many details as possible within the limits of the display area and without loss of readability.

The equipment component uses only composite graphic objects. The graphic objects are created by the view renderer, which calls the appropriate object renderers to obtain the composite graphic objects. The view determines which object renderer is required to draw the graphic object that translates a particular representation object or attribute from the style sheet properties.

The following basic variations of graphic object exist in the equipment component:

♦ Nodes—Nodes are the basic graphic objects and they are represented as equipment elements according to the conventions of the governing standards, such as ITU-T or ANSI, and the appropriate protocols.

♦ Links—Links are the connections between nodes.

End-user interaction with a graphic object is handled by an object interactor. Object interactors handle the gestures of an end user when performing a task. Gestures consist of one or more mouse events to perform one task.

For more information on object interactors, see *Object interactors*.

Graphic object classes

The graphic representation of each object displayed in the equipment component is implemented through the `IlpGraphic` interface. JViews TGO provides predefined equipment graphic objects that are produced by the default equipment component renderer. You can customize the rendering of the objects through CSS.

For custom business objects, JViews TGO provides a default representation with a set of properties that can be customized to represent your objects better. If you prefer, you can also specify a new graphic representation by defining an `IlvGraphic` class in the CSS. For more information, refer to Customizing user-defined business objects. You can also refer to the composite graphics sample at **<installdir>/samples/network/compositeGraphic** to see how to create a new object representation by using the IBM® ILOG® JViews composite graphics feature.

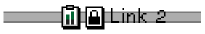
Predefined business objects already have a specific graphic representation that can only be changed through CSS customization by setting the object properties.

You can customize the graphic representation by adding new decorations. To see how to add new decorations to the objects using CSS, look at the decoration sample at ***<installdir>*** **/samples/network/decoration**).

Equipment graphic object renderers

Object renderers in the equipment component create one graphic object that translates a complete representation object. No graphic objects are created that correspond to attributes of a representation object. Instead, subcomponents of graphic objects are created from attributes for example, a label can be created from the `name` attribute.

Such subcomponents are combined into composite graphic objects, like the link with secondary states and a label shown in *Link with secondary states and a label* by using attachments.



Link with secondary states and a label

A composite graphic object constructed in this way looks like one object. The different instances of `IlpGraphic` used to build the graphic object cannot be distinguished as separate objects in the equipment. JViews TGO manages only the composite object. You cannot move the label separately from the link.

The controller

The controller manages the actions triggered by the end user, which have an immediate effect on the view, such as changing the zoom level. It is an instance of the class `IlpEquipmentController`.

The controller helps configure the view interactors and reacts to end-user interactions (such as requests to reshape or move objects) by forwarding actions to the handler, which updates the model directly or indirectly (through the data source).

The controller is used to support view parameters such as layers, zoom, and background. It also manages the toolbar and the interactors by defining which interactors are available in the toolbar.

Classes of the equipment controller

Each time an equipment component is created (instance of `IlpEquipment`), a default controller of class `IlpEquipmentController` is automatically created.

By default, a handler of class `IlpEquipmentHandlerWithoutDataSource` is instantiated. This handler is designed to forward end-user interactions directly to the equipment model. When a data source is attached to an equipment component, a different handler is instantiated, namely `IlpEquipmentHandlerWithDataSource`; this handler forwards end-user interactions to the data source instead of the model.

Using the equipment controller

Some of the most common methods of the class `IlpEquipmentController` can also be found in the class `IlpEquipment`.

Access to the controller API is through the method `getController()`.

By default, no data source is associated with the equipment component, which means that the default handler is an instance of the class `IlpEquipmentHandlerWithoutDataSource`. When a mutable data source is associated with the equipment component through the method `setDataSource(ilog.cpl.datasource.IlpDataSource)`, a new handler of class `IlpEquipmentHandlerWithDataSource` is instantiated, unless a custom handler has been set. The method `setHandler(ilog.cpl.equipment.IlpEquipmentHandler)` allows you to set new handlers to the controller.

The controller manages the view and object interactors. By default, an instance of the class `IlpEquipmentDefaultViewInteractor` is set as the view interactor for the controller, but it is possible to define specific interactors for the view through the `IlpEquipment` API.

The handler

In the same way as the adapter passes information about the objects in the data source to the equipment component, the handler passes information in the opposite direction, that is, from the equipment component to the data source.

The information notified in this way includes:

♦ Actions triggered by interactors for

- Creating objects in the data source
- Removing objects from the data source
- Changing attributes of objects in the data source
- Changing the parent object of an object in the data source
- Expanding and collapsing container objects

♦ Propagating position changes of objects

The position changes are usually due to layout, zoom change, or interactors.

The handler has been designed to simplify the customization of user interactions without rewriting the controller.

There are four types of handler:

♦ `IlpPositionHandler` to handle object position changes.

♦ `IlpNodeHandler` to handle object additions, removals and updates, as well as relationship changes.

♦ `IlpLoadHandler` to handle reloading of model objects from an XML file.

♦ `IlpExpansionHandler` to handle the expansion and collapsing of objects.

The `IlpEquipmentHandler` interface indirectly extends all four types of handler.

The handler has a reference to the data source in the form of an `IlpMutableDataSource`, and to the equipment adapter.

You can customize the behavior of the handler by subclassing the class `IlpEquipmentHandlerWithDataSource`. A particular method can be overridden for each of the possible actions.

You can customize the way position changes are propagated by overriding the method `propagatePositionToDataSource(ilog.cpl.model.IlpObject, ilog.cpl.graphic.IlpPositionSource)`. In a typical situation where the client is active, position changes are propagated to the data source. Therefore, this method returns `true` by default. In a situation where the client has read-only access, you may want to allow only user-requested position changes or no position changes at all to be forwarded to the data source. You can achieve this result by allowing the method `propagatePositionToDataSource(ilog.cpl.model.IlpObject, ilog.cpl.graphic.IlpPositionSource)` to return `false` in the appropriate cases.

The handler is most often subclassed to allow you to customize the creation of new objects in the data source. The object interactors may need to be customized in the same way. A customized object creation interactor typically calls the controller method `createObject(java.lang.Class, ilog.cpl.model.IlpAttributeGroup, java.util.Map, ilog.cpl.graphic.IlpPosition)` with specific properties. The controller then forwards these properties to the handler. Finally, the method `handleCreateObject(java.lang.Class, ilog.cpl.model.IlpAttributeGroup, java.util.Map, ilog.cpl.graphic.IlpPosition)` of the handler parses the additional properties and creates the new objects.

By default, the handler creates new objects in two steps:

1. The ID of the new object is created with the method `createObjectId(java.lang.Class, ilog.cpl.model.IlpAttributeGroup, java.util.Map)`.
2. The `IlpObject` corresponding to this ID is created with the method `createObject(java.lang.Class, ilog.cpl.model.IlpAttributeGroup, java.util.Map, java.lang.Object)`.

You can customize each step separately by overriding these methods in a subclass.

Any user interaction with the equipment is processed by the equipment controller which delegates action to the equipment handler. The handler has two default implementations:

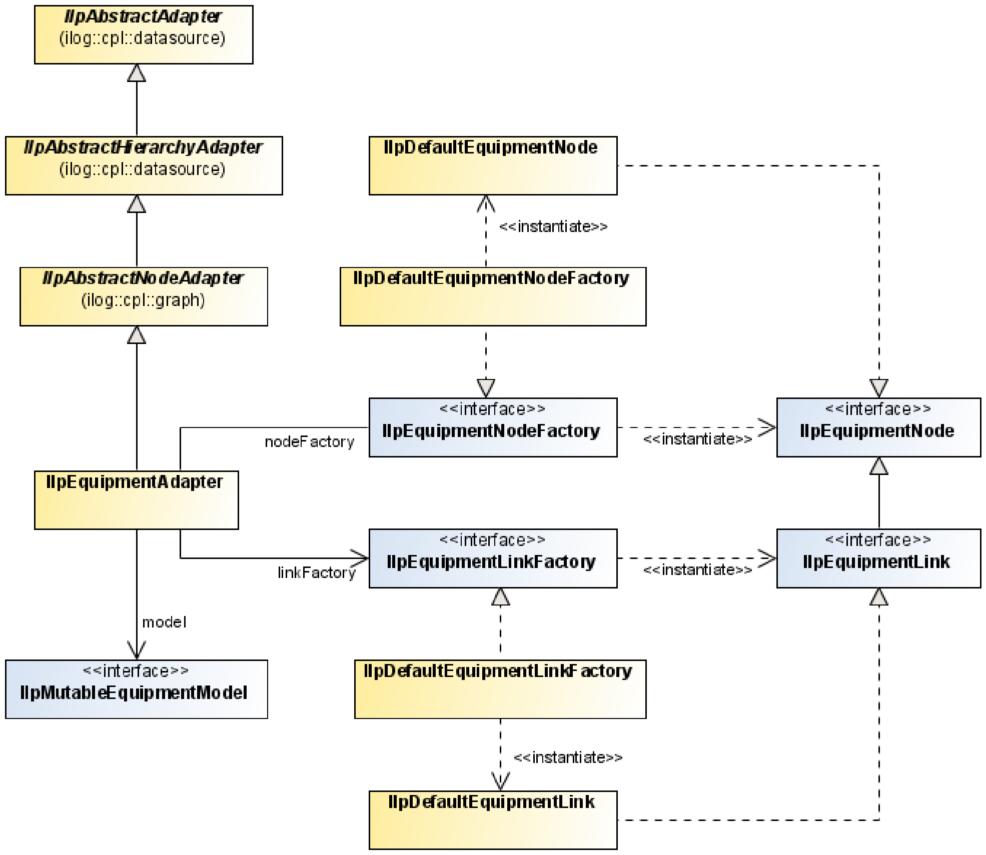
- ♦ `IlpEquipmentHandlerWithoutDataSource` is attached by default to the controller and performs user interactions directly in the equipment model.
- ♦ `IlpEquipmentHandlerWithDataSource` is automatically attached to the controller when a data source is connected with the equipment component. User interactions are executed inside the data source that will notify the adapter, and the adapter in turn will notify the equipment model. In this particular use case, changes will be reflected on all equipment components, if any, connected to the data source.

The adapter

The equipment adapter converts business objects into representation objects of type equipment node. It is defined by the class `IlpEquipmentAdapter`.

Equipment adapters retrieve structural information (that is, parent/child relationship) about business objects from the associated data source and determine whether an object should appear as a root representation object by examining a list of origins. See *Setting a list of origins* for details.

The following figure shows equipment adapter classes:



Equipment adapter classes

Nodes are created with an `IlpEquipmentNodeFactory`. By default, this factory is an instance of the class `IlpDefaultEquipmentNodeFactory` which creates `IlpDefaultEquipmentNode` instances.

Similarly, links are created with an `IlpEquipmentLinkFactory`. By default, this factory is an instance of the class `IlpDefaultEquipmentLinkFactory` which creates `IlpDefaultEquipmentLink` instances.

The equipment adapter supports the concept of position or shape of objects. By default, it interprets every object attribute with name `position` as the position of that object. You can specify any other attribute instead, for all instances of a given `IlpClass`, through the method `setPositionAttribute` of the class `IlpAbstractNodeAdapter`.

You can create an equipment adapter implicitly by instantiating the `IlpEquipment` component as shown in the following example.

How to create an equipment adapter by instantiating an equipment component

```
IlpEquipment ilpEquipment = new IlpEquipment();
IlpDataSource dataSource = new IlpDefaultDataSource();
ilpEquipment.setDataSource(dataSource);
```

If you want to configure the adapter, to set its origin for example, you must first retrieve it from the equipment component and then set it to the data source.

How to configure an equipment adapter

```
IlpEquipment ilpEquipment = new IlpEquipment();
IlpDataSource dataSource = new IlpDefaultDataSource();
// configure the adapter, for example set an origin
IlpEquipmentAdapter adapter = ilpEquipment.getAdapter();
adapter.setOrigins(Collections.singletonList("origin"),false);
adapter.setDataSource(dataSource);
```

The equipment adapter supports temporary representation objects. For more information, see *Creating a temporary representation object*.

Creating a temporary representation object

The equipment adapter supports temporary representation objects. These objects are placeholders that can be used in place of permanent representation objects for editing purposes and, more specifically, when new objects are created in the equipment view. When a business object corresponding to the temporary representation object is added to the data source, this temporary representation object is removed and replaced by the permanent representation of the business object. A filter, defined by `IlpFilter`, is used to determine when the representation object of a business object added to the data source is a candidate to replace the temporary representation object. Filtering criteria can be of any kind.

The following example shows how to add a temporary representation object to an equipment adapter.

How to add a temporary representation object to an equipment adapter

First you create the temporary representation object, like this:

```
IlpDefaultEquipmentNode temp=
          new IlpDefaultEquipmentNode(new IlpDefaultAttributeGroup());
```

Then you add it to the adapter along with the filtering criteria using the method `storeTemporaryRepresentationObject(ilog.cpl.model.container.IlpRepresentationNode, ilog.cpl.model.container.IlpMutableRepresentationNode, ilog.cpl.util.IlpFilter)`:

```
adapter.storeTemporaryRepresentationObject(temp, null,new IlpFilter() {
  public boolean accept(Object o) {
   IlpObject ilpO = (IlpObject)o;
   return ilpO.getIdentifier().equals("right one");
   }
};
```

The temporary representation object will be replaced by a permanent representation object as soon as a business object satisfying the filtering criteria is added to the data source.

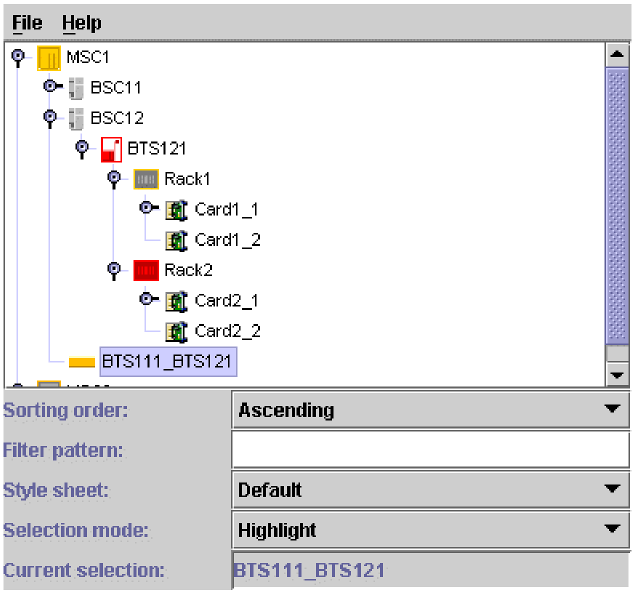
Tree component

The tree component is one of the four graphic components supplied with IBM® ILOG® JViews TGO. It provides a hierarchical view of the data contained in a data source. A tree has one or more root nodes, from which all the other nodes descend. Each tree node may be associated with a business object (for example, a managed object, an alarm, or a service). It has a number of graphic properties (for example, a label, an icon, a tooltip) that are set according to the values of one or more attributes of the object. The way in which business object attribute values map to the graphic representation of the business object is determined by the style sheets of the tree. For details on business objects, refer to Introducing business objects and data sources.

In this section

Introducing the tree component

The JViews TGO tree component is based on the Swing tree component. It allows you to display data in a hierarchical representation.



Sorting order: Ascending
Filter pattern:
Style sheet: Default
Selection mode: Highlight
Current selection: BTS111_BTS121

The tree component is connected to a data source, from which it retrieves the business objects to be displayed. By default, the tree displays all the objects contained in the data source. However, it is possible to restrict the contents displayed by:

- selecting the root nodes to be shown,
- specifying whether certain child objects should be visible or not,
- applying a filter.

Objects that do not have a parent are displayed as root nodes, while the others are displayed under their parent.

The tree component offers the following features:

An efficient tiny look and feel to represent business objects as tree nodes
This graphic representation style provides a comprehensive view of the object state in the tree, even if it contains less details than the normal representation style in the network and equipment components. The graphic representation of the business objects in the tree component can be customized through Cascading Style Sheets (CSS). For more information, refer to Using Cascading Style Sheets.

Smart selection modes
The tree component allows you to choose a look and feel for the selection: either the standard look and feel, where the selected cell appears highlighted, or the check box

look and feel, where a check box is displayed next to each tree node with a check mark indicating that the node is selected.

The tree component also provides a selection model that is responsible for setting, modifying and retrieving the objects selected in the component.

Sorting capabilities
The tree component allows you to sort the nodes that are displayed.

Filtering capabilities
The tree component allows you to filter the nodes that are displayed. That is, the business objects present in the attached data source are only displayed in the tree if they are accepted by the current filter.

Interaction support
The tree component allows you to associate behavior with the tree as a whole, and with the business objects it contains.

Load on demand
The tree component supports load on demand for the business objects to be displayed. This means that the graphic representation of a given business object is only created when its parent object is expanded through the API or through user interaction. By default, load on demand is customized through the CSS property `expansion` (see Customizing the expansion of business objects). More advanced customization can be performed at the adapter level (see *Expansion strategy*).

The tree component is implemented by the class `IlpTree`, which is a Swing `JComponent` that can be directly inserted into a panel (`JPanel`).

`IlpTree` provides the API for the most common uses of the tree component, such as:

♦ setting or retrieving the associated data source: `getDataSource()`, `setDataSource(ilog.cpl.datasource.IlpDataSource)`

♦ accessing and modifying the selection: `getSelectionModel()`, `setSelectionModel(javax.swing.tree.TreeSelectionModel)`, `addSelectionObject(ilog.cpl.model.IlpObject)`, `removeSelectionObject(ilog.cpl.model.IlpObject)`, `clearSelection()`, `isObjectSelected(ilog.cpl.model.IlpObject)`, `getSelectedObject()`, `getSelectedObjects()`

♦ setting or retrieving the view interactor: `setViewInteractor(ilog.cpl.interactor.IlpViewInteractor)`, `getViewInteractor()`

♦ changing the root nodes of the tree through the data source adapter: `getAdapter()`

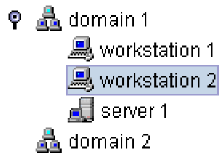
♦ filtering the tree nodes: `setFilter(ilog.cpl.util.IlpFilter)`, `getFilter()`

♦ sorting the tree nodes: `setSortComparator(java.util.Comparator)`, `getSortComparator()`

`IlpTree` also acts as a façade for a number of lower-level components that it contains. These components provide more detailed APIs and advanced services. They are described in *Architecture of the tree component* of this section.

Creating a tree component: a sample

This topic shows you how to create the following basic tree featuring a computer network. It contains extracts of sample code located in ***<installdir>* /samples/tree/basic** and ***<installdir>* /samples/tree/customClasses**.



A Basic Network Tree

The following sample source and resource files are located in ***<installdir>* /samples/tree/customClasses**:

- `Main.java`: source file for the sample
- `deploy.xml`: deployment descriptor
- `treenodes.xml`: data source input
- `tree.css`: style sheet defining how the tree and objects are graphically represented

How to create a basic network tree

The following list explains how to create a tree component, how to connect it to a data source, how to fill the data source from an XML file that describes the business objects, and how to configure the graphic representation of the tree and the objects.

1. Initialize the JViews TGO library.

   Prior to using any JViews TGO API, you must call `IltSystem.init`.

   ```
   IltSystem.init("deploy.xml");
   ```

   `deploy.xml` is a deployment descriptor file that defines the path to the application resources to be used:

   ```
   <deployment>
     <urlAccess>
       <!-- Add relative path to sample root directory -->
       <relativePath>../..</relativePath>
     </urlAccess>
   </deployment>
   ```

2. Create a tree component.

```
IlpTree treeComponent = new IlpTree();
```

3. Add the tree component to a container.

```
JFrame frame = //...
Container contentPane = frame.getContentPane();
contentPane.add(treeComponent);
```

   To display the tree component, you must add it to a container. In this code, the tree component is inserted in a `JFrame` container.

4. Create a data source and fill it from an XML file.

```
IltDefaultDataSource dataSource = new IltDefaultDataSource();
dataSource.parse("treenodes.xml");
```

   `treenodes.xml` contains the description of the business model classes and instructions to create business object instances in the data source.

   The tree component uses the `<parent>` XML tag to build its hierarchy. For example, the XML fragment:

```
<addObject id="Server 1">
    <class>Server</class>
    <parent>Domain 1</parent>
```

   indicates that the object “Server 1” will be created as a subnode of “Domain 1.”

   Refer to Adding business objects from JavaBeans in the *Business Objects and Data Sources* documentation for more information on the XML format recognized by the JViews TGO data source.

5. Connect the tree component to the data source.

```
treeComponent.setDataSource(dataSource);
```

6. Configure the graphic representation of the tree component .

   You can specify how the tree and the objects should be represented by using cascading style sheets (CSS), as follows:

```
String[] css = new String[] { "tree.css" };
try {
  treeComponent.setStyleSheets(css);
} catch (Exception e) {
}
```

Configuring the tree component

Identifies the rendering information necessary to customize a tree view.

In this section

Introduction

The tree component can be customized either through a CSS configuration file or through the API, the easiest and preferred way being the CSS configuration. You also have the possibility to load a project file which combines the CSS configuration and the tree data.

You can customize the tree view with properties such as background, cell renderer and selection look and feel. You can also customize the tree adapter and the way the business objects are represented.

Configuring the tree component through a CSS file

You can customize the following features in a CSS file:

Tree view

- ♦ Selection look and feel
- ♦ Background
- ♦ Tree cell renderer
- ♦ Row height
- ♦ Scrolling on expand

Tree adapter

- ♦ Filter
- ♦ Comparator
- ♦ Origins
- ♦ Accepted classes
- ♦ Excluded classes
- ♦ Expansion strategy factory
- ♦ Tree node factory

Tree nodes

- ♦ Properties listed under Customizing tree nodes in the *Styling* documentation.

Tree controller

- ♦ View interactor
- ♦ Object interactor

You can customize the following features in a CSS file:

How to load a CSS file in a tree component

The tree configuration can be split accross several CSS files that you can load by:

- ♦ Specifying a project file that lists the style sheets and the data file to be loaded in the component (see *Loading a project file*). The project file will be as follows:

```
<?xml version="1.0"?>
<tgo xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/project.xsd"
style="configurationFile1.css,configurationFile2.css">
  <datasource javaClass="ilog.tgo.datasource.IltDefaultDataSource"
```

```
fileName="tree.xml"/>
</tgo>
```

If the settings in two of the CSS files disagree, the effect depends on the order of the filenames in the list: the last file mentioned takes precedence over the first file.

♦ Using the method `IlpTree.setStyleSheets` as follows:

```
treeComponent = new IlpTree();
try {
  treeComponent.setStyleSheets(new String[] {
                      myConfigurationFile1,myConfigurationFile2 });
} catch (Exception e) {
}
```

If the settings in the CSS files disagree, the effect depends on the order of the filenames in the list: the last file listed takes precedence over the first one.

How to configure a tree component in a CSS file

The following code represents an example of configuring a tree using CSS. It is based on the CSS file located in **<*installdir*> /samples/tree/styling/srchtml/tree.css.html** where `<installdir>` is the directory where you have installed JViews TGO.

The configuration in CSS is organized as a set of rules that define properties.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Tree Component configuration
// Type: Tree
// The following list shows all possible properties for
// the tree component.
// - view : enables the tree view configuration
// - interactor: enables the interactor configuration
// - adapter: enables the adapter configuration
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Tree {
  view: true;
  adapter: true;

}
View {
  background: #FFFFDF;
  selectionLookAndFeel: Highlight;
}
```

The Tree rule

This rule specifies the elements of the tree component that will be customized. It contains Boolean flags that indicate whether some specific customizable properties are present. For example, the customization of the property `adapter` is not taken into account unless `adapter: true` is declared in the Tree rule.

This feature provides powerful cascading possibilities. You can define adapter customizations in a default CSS file and turn them on or off in another CSS file.

The following CSS properties affect the tree component:

CSS properties of the tree component

CSS Property	Type of Value	Default	Usage
`view`	`boolean`	`false`	Enables the customization of the tree view.
`interactor`	`boolean`	`false`	Enables the customization of the tree interactors.
`adapter`	`boolean`	`false`	Enables the customization of the tree adapter.

The View rule

This rule specifies the properties that are applied to the tree view.

The following CSS properties affect the appearance of the tree view:

CSS properties of the tree view

CSS Property	Type of Value	Default	Usage
`background`	`Color`	`null`	Color to be used in the background of the tree view. If the value is `null`, the color of the active look-and-feel is used.
`cellRenderer`	`TreeCellRenderer`	`IlpTreeCellRenderer`	Renders the tree nodes.
`scrollsOnExpand`	`boolean`	`true`	When a node is expanded, this property determines whether or not the node scrolls up, so that the maximum number of descendants are visible. The default is `true`.
`selectionLookAndFeel`	`int`	`HIGHLIGHT`	Sets the way the selection is rendered to/manipulated by the end-user. Two possible values: `HIGHLIGHT` or `CHECKBOX`
`toggleClickCount`	`int`	`2`	Number of mouse clicks before a node

CSS Property	Type of Value	Default	Usage
			expands or collapses. The default is 2.
`rootVisible`	`boolean`	`false`	Defines whether the root nodes in the tree are visible or not.
`showsRootHandles`	`boolean`	`true`	Defines whether the handles that enable nodes to be expanded by the user are visible or not.
`rowHeight`	`int`	`-1`	Defines the row height. If the value is -1, the row height depends on the tree node rendering. If the value is greater than 0, the tree node height corresponds to the value defined by the user.
`expandsSelectedPaths`	`boolean`	`true`	Defines whether the parent path of the selected node is expanded or not. If `true`, all the parents of the selected node are expanded, although they may not all be visible in the `JTree`. If `false`, the parent nodes are not expanded and thus not made visible in the `JTree`.

Refer to the class `IlpViewRenderer` in the IBM® ILOG® JViews TGO *Java™ API Reference Documentation* for more information on configuring the view in a tree component.

The Interactor rule

This rule controls the configuration of the tree view interactor. The tree view interactor is responsible for handling events that occur in the tree view. You can define an interactor for the tree view or specific interactors for the tree nodes.

CSS Properties of the Tree View Interactor

Property Name	Property Type
`viewInteractor`	`ilog.cpl.interactor.IlpViewInteractor`

How to configure a tree view interactor in a CSS file

Prior to configuring the view interactor, you need to configure the tree component so that the interactor configuration is enabled:

```
Tree {
  interactor: true;
}
```

After that, you can customize the view interactor in the Interactor rule as illustrated by the following code extract. Refer to The CSS specification in the *Styling* documentation for details about the CSS syntax.

```
Interactor {
  viewInteractor: @+viewInt;
}
Subobject#viewInt {
  class: 'ilog.cpl.interactor.IlpDefaultViewInteractor;'
  popupMenuFactory: @+viewPopupMenuFactory;
  action[0]: @+viewAction0;
}
Subobject#viewPopupMenuFactory {
  class: 'AlarmPopupMenuFactory';
}
Subobject#viewAction0 {
  class: 'ilog.cpl.interactor.IlpGestureAction';
  gesture: BUTTON2_CLICKED;
  action: @=showDetailsAction;
}
```

The behavior of the view interactor is determined by the actions that are associated with user gestures and keystrokes. This behavior can also be customized through CSS. You can also configure a pop-up menu to be displayed in the tree view. Please refer to *Interacting with the tree view* and *Interacting with the tree nodes* for more information about interactor customization.

When the interactor renderer is enabled, you can also customize interactors for specific tree nodes using CSS selectors to set the value of property `interactor`. For more information, refer to `IlpInteractorRenderer`.

The Adapter rule

This rule controls the configuration of the tree adapter. The tree adapter is responsible for converting the business objects in the data source to representation objects (tree nodes) in the tree component. It provides the following features:

- **Filtering**: applies a filter so that business objects currently in the data source are not mapped to representation objects in the tree component.
- **Comparator**: specifies that the position of the objects in the tree will follow a comparison rule. In this way, it is possible, for example, to display tree nodes in alphabetical order.
- **Expansion strategy**: defines how the objects will be loaded in the tree component, that is, either at initialization time or on demand, as the user interacts with the tree nodes.
- **Origins**: defines which objects become root nodes in the tree.
- **Accepted classes**: defines the list of business classes that are accepted by the tree adapter. Only the business objects that match one of these business classes will be mapped to representation objects by the tree adapter.
- **Excluded classes**: defines the list of business classes that are excluded by the tree adapter. The business objects of these business classes will not be mapped to representation objects by the tree adapter.
- **Object Attribute Changed Events Filtering**: applies a filter so that attribute value changes occuring in the business objects currently in the data source are not notified to the tree component. This allows you to fine tune the notifications that are sent to the tree regarding attribute value changes in the business objects present in the data source. If an attribute value change event is not important to your application needs or to the tree cell rendering, it can be discarded, improving the performance of your tree component.

These tree adapter features can be customized through CSS using the following properties:

CSS properties of the tree adapter

Property Name	Property Type
`filter`	`ilog.cpl.util.IlpFilter`
`origins`	list of object identifiers
`comparator`	`java.util.Comparator`
`nodeFactory`	`ilog.cpl.tree.IlpTreeNodeFactory`
`expansionStrategyFactory`	`ilog.cpl.util.IlpExpansionStrategy Factory`
acceptedClasses	list of `IlpClass`
excludedClasses	list of `IlpClass`
`objectAttributeChangedFilter`	`ilog.cpl.util.IlpFilter`

How to configure a tree adapter in a CSS file

Prior to configuring the adapter, you need to configure the tree component so that the adapter configuration is enabled:

```
Tree {
  adapter: true;
}
```

After that, you can customize each adapter property in the Adapter rule as illustrated by the following code extract. Refer to The CSS specification in the *Styling* documentation for details about the CSS syntax.

```
Adapter {
  filter: @+treeFilter;
  comparator: @+treeComparator;
  expansionStrategyFactory: @+treeExpStrategyFactory;
  nodeFactory: @+treeNodeFactory;
  objectAttributeChangedfilter: @+eventFilter;
  origins[0]: ROOT1;
  origins[1]: ROOT2;
}

Subobject#treeFilter {
  class: MyTreeFilter;
}
Subobject#treeComparator {
  class: MyTreeComparator;
}
Subobject#treeExpStrategyFactory {
  class: MyTreeExpansionStrategyFactory;
}
Subobject#treeNodeFactory {
  class: MyTreeNodeFactory;
  adapter: @adapter;
}
Subobject#eventFilter {
  class: MyTreeEventFilter;
}
```

How to programmatically configure a tree adapter using CSS

You can programmatically modify the CSS configuration of the default tree adapter (`IlpContainmentTreeAdapter`) by using mutable style sheets through the `IlpMutableStyleSheet` API.

Important: The mutable style sheet is set to the adapter as a regular style sheet and is cascaded in the order in which it has been declared.

To use mutable style sheets:

1. Get the mutable style sheet.

   You access the mutable style sheet through the `getMutableStyleSheet()` method in the tree adapter API:

```
IlpMutableStyleSheet mutable = adapter.getMutableStyleSheet();
```

This method automatically registers the mutable style sheet into the adapter. You can manually instantiate an object of the class `IlpMutableStyleSheet` and register it yourself through the `setStyleSheet()` API:

```
IlpMutableStyleSheet mutable = new IlpMutableStyleSheet(adapter);
try {
  adapter.setStyleSheets(new String[] { mutable.toString() });
} catch (Exception x) {
  x.printStackTrace();
}
```

2. Set the CSS declarations.

   Once you have the mutable style sheet, you can set the declarations you want:

```
mutable.setDeclaration("#myObjectId", "expansion", "NO_EXPANSION");
```

   This creates the following CSS declaration into the mutable style sheet:

```
#myObjectId {
  expansion: NO_EXPANSION;
}
```

3. Register the mutable style sheet.

   The mutable style sheet should be set to the adapter as a regular style sheet using the `setStyleSheet()` method:

```
try {
  adapter.setStyleSheets(new String[] { mutable.toString() });
} catch (Exception x) {
  x.printStackTrace();
}
```

4. Set and update the CSS declarations.

   The mutable style sheet can be modified even after being registered to the adapter:

```
// Update the expansion type for 'myObjectId'
mutable.setDeclaration("#myObjectId", "expansion", "IN_PLACE");
// Add a new declaration
mutable.setDeclaration("#myOtherId", "expansion", "IN_PLACE");
```

Note: Like any style sheet, the mutable style sheet is lost when the `setStyleSheet()` API is invoked and a new set of style sheets is applied to the adapter.

How to customize the mutable style sheet

Reapplying a CSS configuration may be a heavy task, as the adapter may be forced to review filters, origins, recreate representation objects, and so on. It is important to use the mutable style sheet with care and to customize it properly to reapply the CSS wisely. To do so, there are two methods available in the `IlpMutableStyleSheet` API: `setUpdateMask()` and `setAdjusting()`.

1. `setUpdateMask()`

   This method controls what should be recustomized once a declaration of the mutable style sheet has been updated. The CSS configuration of the adapter is divided into two parts: *adapter customization* and *representation object* customization.

   The adapter customization handles the origins, filters, and so on:

```
Adapter {
  origins[0]: id0;
  origins[1]: id1;
  showOrigin: true;
  filter: @+myFilter;
}
```

   The representation object customization handles the expansion type of a representation object:

```
#myObjectId {
  expansion: IN_PLACE;
}
```

   The accepted values for `setUpdateMask()` are:

   - `IlpStylable.UPDATE_COMPONENT_MASK`: Only the adapter part is recustomized.
   - `IlpStylable.UPDATE_OBJECTS_MASK`: Only the representation object part is recustomized.
   - `IlpStylable.UPDATE_ALL_MASK`: Bot the adapter and representation object parts are recustomized.
   - `IlpStylable.UPDATE_NONE_MASK`: Nothing is recustomized.

   For example, if you update the expansion type of a representation object through the mutable style sheet, it is recommended that you set the update mask to `UPDATE_OBJECTS_MASK` as there is no need to reapply the CSS configuration for the adapter part:

```
mutable.setUpdateMask(IlpStylable.UPDATE_OBJECTS_MASK);
mutable.setDeclaration("object", "expansion", "IN_PLACE");
```

2. `setAdjusting()`

This method is used when a series of declarations must be applied to the mutable style sheet. When the method is set to `true`, the mutable style sheet puts all the calls to `setDeclaration()` into a queue. When the method is set back to `false`, all the queued declarations are processed in a batch:

```
mutable.setAdjusting(true);
mutable.setDeclaration("#myObjectId", "expansion", "IN_PLACE");
mutable.setDeclaration("#myOtherId", "expansion", "IN_PLACE");
mutable.setAdjusting(false);
```

Configuring the tree component through the API

For details of the classes involved in the architecture of the tree component, see *Architecture of the tree component*.

For details about programming the individual services of a tree component, see *Tree component services* .

How to configure the tree view with the API

The following code sample shows how to configure the tree view (`IlpTreeView`) through the API.

```
IlpTreeView view = tree.getView();

// Setting the selection mode
view.setSelectionLookAndFeel(IlpTreeView.HIGHLIGHT_SELECTION_LOOK_AND_FEEL);

// Setting the tree cell renderer
view.setCellRenderer(new MyTreeCellRenderer());

// Setting the background color
view.setBackground(Color.white);
```

How to configure the tree view interactor with the API

The following code sample shows how to configure the tree view interactor through the API. For details of this interactor, see *Interacting with the tree view*.

```
IlpTree tree = // ...
// Retrieve the view interactor
IlpViewInteractor viewInteractor = tree.getViewInteractor();
// Create an action
Action myAction = new MyAction();
// Clicking the 3rd mouse button will trigger myAction
viewInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED,myAction);
```

How to configure the tree adapter with the API

The following code sample shows how to configure the tree adapter (`IlpAbstractTreeAdapter`) through the API.

```
IlpAbstractTreeAdapter adapter = tree.getAdapter();

// Setting the filter
IlpFilter myFilter = new MyFilter();
// (it is the same as tree.setFilter(myFilter)
adapter.setFilter(myFilter);

// Origin
List myOrigins = new ArrayList();
myOrigins.add(objectID_1);
myOrigins.add(objectID_2);
.
.
.
myOrigins.add(objectID_n);
// in this case we want to display the
// origins
boolean showOrigin = true;
adapter.setOrigins(myOrigins, showOrigin);

// Expansion Strategy Factory
// Usually the expansion strategy factory relies
// on the adapter to access the data source and to
// load/release objects
IlpExpansionStrategyFactory myExpFactory = new
   MyExpansionStrategyFactory(adapter);
adapter.setExpansionStrategyFactory(myExpFactory);

// Tree Node Factory
IlpTreeNodeFactory myNodeFactory = new MyTreeNodeFactory();
adapter.setNodeFactory(myNodeFactory);
```

```

// Setting the object attribute value changed event filter IlpFilter
myEventFilter = new MyEventFilter();
adapter.setObjectAttributeChangedFilter(myEventFilter);
```

Loading a project file

A project is a combination of style sheets that supply rendering information and a data source that supplies the data to be represented in a tree component. A project is saved as an XML file with extension `.itpr`.

Loading a project file is the recommended way to configure a graphic component in Java™ as it is the fastest.

How to load a project file into a tree component

The following code sample shows how to load a project file into a tree component, using the method `setProject`.

```
IlpTree tree = new IlpTree();
tree.setProject(new URL("file:project.itpr");
```

The project is represented by the `IlpTGOProject` class, included in the package `ilog.cpl.project`. When a new project is created, the style sheet and data source are both null.

How to create a new project for the tree component

The following code sample shows how to create a new project file by setting the style sheets and data source, then saving the project.

```
IlpTGOProject project = new IlpTGOProject();
project.setStyleSheet(new URL("file:example.css");
IltDefaultDataSource dataSource = new IltDefaultDataSource();
dataSource.setFileName("data.xml");
project.setDataSource(dataSource);
project.write(new URL("file:example.itpr");
```

Customizing the rendering of tree nodes

The JViews TGO tree component renders instances of both user-defined and predefined business classes using a default tree cell renderer (`IlpTreeCellRenderer`) that you can customize through style sheets. The default tree cell renderer uses CSS properties to define how each tree node will be represented.

For details on how to customize the rendering of tree nodes, see Customizing tree nodes.

Tree component services

Describes the services that are available for a tree: view services and adapter services.

In this section

Controlling the display of objects as tree leaves
Describes how to control whether an object is treated as a tree leaf.

Setting a list of origins
Describes how to set a list of orgins to explicitly select the root nodes to be displayed by the tree component.

Introduction

This section describes the services that are available for a tree. They are of two kinds:

- ♦ View services, related to the tree view
  - *Filling the tree with business objects*
  - *Interacting with the tree view*
  - *Interacting with the tree nodes*
  - *Handling the selection*
- ♦ Adapter services, related to the tree model
  - *Filtering the tree nodes*
  - *Accepted and excluded classes*
  - *Sorting the tree nodes*
  - *Controlling the display of objects as tree leaves*
  - *Setting a list of origins*

Filling the tree with business objects

The tree can be filled with business objects through an `IltDefaultDataSource`, as illustrated by the following code.

How to fill a tree with business objects

```
// Create a tree component and connect it to a data source
IlpTree treeComponent = new IlpTree();
IltDefaultDataSource dataSource = new IltDefaultDataSource();
// The second parameter specifies which kind of object contained
// in the data source the tree will show
treeComponent.setDataSource(dataSource, IltNetworkElement.GetIlpClass());

// Create a business object and insert it in the data source
IltObjectState os = new IltBellcoreObjectState();
os.set(IltBellcore.State.EnabledActive);
IltNetworkElement london =
  new IltNetworkElement("Fax", IltNetworkElement.Type.Fax, os);
dataSource.addObject(london);
```

Note that a data source can also load an XML file containing business objects, as shown in *Creating a tree component: a sample*.

You can also use project files to easily create and load data source business objects into your tree component, as illustrated below:

How to fill a tree with business objects using a project file

You can create the following project file to indicate the type of data source to be used and the XML file that contains the data source information:

```
<?xml version="1.0"?>
<tgo xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/project.xsd">
  <datasource javaClass="ilog.tgo.datasource.IltDefaultDataSource"
fileName="network.xml"/>
</tgo>
```

Once the project file is specified, you can load it in the component using method `IlpTree.setProject(URL projectURL)` or `IlpTree.setProject(IlpProject project)`. See *Loading a project file* for details.

Interacting with the tree view

The `IlpTree` allows you to associate behavior with the tree view as a whole, and with the business objects it contains.

In particular, using the default view interactor, you can:

- ♦ associate actions with mouse events and focus events,
- ♦ associate actions with keyboard events,
- ♦ define a pop-up menu factory to build a pop-up menu that displays in the view.

The `IlpTree` is associated with an `IlpDefaultViewInteractor`, which should satisfy most needs. You can retrieve this interactor by calling the method `getViewInteractor()`.

How to associate an action with a mouse event in the tree view

You can associate actions with mouse events by using either CSS or the API. The following CSS extract shows how to proceed:

```
Tree {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.interactor.IlpDefaultViewInteractor';
  action[0]: @+viewAction0;
}

Subobject#viewAction0 {
  class: 'ilog.cpl.interactor.IlpGestureAction';
  gesture: BUTTON3_CLICKED;
  action: @+myAction;
}
Subobject#myAction {
  class: MyAction;
}
```

The same configuration can be achieved through the API, as follows:

```
IlpTree tree = // ...
// Retrieve the view interactor
IlpViewInteractor viewInteractor = tree.getViewInteractor();
// Create an action
Action myAction = new MyAction();
```

```
// Clicking the 3rd mouse button will trigger myAction
viewInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED,myAction);
```

You can find out whether this event occurred on an `IlpObject` by means of the following code (which should be in the `MyAction` class).

How to check whether a given action occurred

```
// Implementation of the ActionListener interface
public void actionPerformed(ActionEvent e) {
  // ILOG JTGO interactors use IlpViewActionEvent
  IlpViewActionEvent viewEvent = (IlpViewActionEvent)e;
  // Get the IlpObject (if any) where the interaction occurred
  IlpObject ilpObj = viewEvent.getIlpObject();
  // Perform operation on the given object
}
```

How to associate an action with a keyboard event in the tree view

You can associate actions with keyboard events by using either CSS or the API. The following CSS extract shows how to proceed:

```
Tree {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.interactor.IlpDefaultViewInteractor';
  action[0]: @+viewAction0;
}

Subobject#viewAction0 {
  class: 'ilog.cpl.interactor.IlpKeyStrokeAction';
  keyStroke: 'typed D';
  action: @+myAction;
}

Subobject#myAction {
  class: MyAction;
}
```

The same configuration can be achieved through the API, as follows:

```
// Create an action
Action myAction = new MyAction();
```

```
// Typing CTRL+D will trigger myAction
viewInteractor.setKeyStrokeAction(
  KeyStroke.getKeyStroke('D',java.awt.Event.CTRL_MASK),myAction);
```

How to define a pop-up menu factory for the tree view

You can customize a pop-up menu factory for the tree view either through CSS or through the API. The following CSS extract shows how to configure a CSS file to add a pop-up menu factory:

```
Tree {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.interactor.IlpDefaultViewInteractor';
  popupMenuFactory: @+viewPopupMenuFactory;
}
Subobject#viewPopupMenuFactory {
  class: MyPopupMenuFactory;
}
```

The same configuration can be achieved through the API, as follows:

```
// Subclass IlpAbstractPopupMenuFactory, which has useful shortcuts
IlpPopupMenuFactory popupMenuFactory = new IlpAbstractPopupMenuFactory(){
  // Add the identifier of each of the selected objects to the menu
  public JPopupMenu createPopupMenu (IlpObjectSelectionModel ilpSelectionModel)
  {
    // Create an empty popup menu
    JPopupMenu menu = new JPopupMenu();
    // Access the selected objects from the selection model
    Collection selectedObjects = ilpSelectionModel.getSelectedObjects();
    // fill the menu according to the current selection
   return menu;
  }
};
```

The following code shows you how to associate the defined pop-up menu factory with the tree component.

How to associate a pop-up menu factory with the tree component

```
// Set the popup menu factory to the view interactor
viewInteractor.setPopupMenuFactory(popupMenuFactory);
```

Note: The selection is updated prior to invoking the pop-up menu factory. Specifically, right-clicking a selected object keeps the entire current selection, while clicking a nonselected object cancels the current selection and selects the object clicked. Clicking outside any object clears the selection.

Please refer to *Interacting with the graphic components* for a detailed description of interactors and gestures.

Interacting with the tree nodes

The section *Interacting with the tree view* describes how to set an interactor for the entire tree view. You can also associate behavior with business objects (for a class or for individual objects), as well as with individual `IlpTreeNode` instances. To do so, you use object interactors, which provide the same options as the view interactor, that is:

- ♦ associating actions with mouse events,
- ♦ associating actions with keyboard events,
- ♦ defining a pop-up menu factory to build a pop-up menu that displays on representation objects.

The object interactor handles any events occurring on the object with which the interactor is associated, provided the view interactor has enabled the use of object interactors. You can check this by means of the `isUsingObjectInteractor` method or modify it with the `setUsingObjectInteractor` method. Object interactors are enabled by default.

No default interactor is associated with any object. To associate actions with mouse or keyboard events or to define a pop-up menu factory, you first have to create an `IlpObjectInteractor`. You may use the `IlpDefaultObjectInteractor`, extend it, or create your own implementation.

How to associate an object interactor with a representation object in the tree

You can associate an object interactor with a representation object by using either CSS or the API. The following CSS extract shows how to proceed:

```
Tree {
  interactor: true;
}

object."ilog.tgo.model.IltNetworkElement" {
  interactor: @+objInteractor;
}
Subobject#objInteractor {
  class: 'ilog.cpl.interactor.IlpDefaultObjectInteractor';
}
```

The same configuration can be achieved through the API, as follows:

```
IlpTree tree = // ...
IlpTreeController treeController = tree.getController();
// Create an object interactor
IlpObjectInteractor objectInteractor = new IlpDefaultObjectInteractor();
// Associate the object interactor with a given representation object
IlpTreeNode treeNode = // ...
treeController.setObjectInteractor( treeNode, objectInteractor);
// Configuring the specific object interactor is similar to configuring
// a view interactor.
objectInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED,
  new MyAction());
```

Actions related to mouse and keyboard events can be customized in the same way as for the view interactor. A pop-up menu factory can also be defined in the same way as for the view interactor. Please refer to *Interacting with the tree view*.

Please refer to *Interacting with the graphic components* for a detailed description of interactors and gestures.

Handling the selection

The selection model requirements for an `IlpTree` are defined by the interface `IlpTreeSelectionModel`. This model extends the Swing `TreeSelectionModel` to provide convenience operations when working with business objects (`IlpObject`).

The selection model is responsible for setting, modifying, and retrieving the objects selected in the `IlpTree`.

The `IlpTreeSelectionModel` interface provides the following basic methods:

- ♦ modifying or retrieving the selected `IlpObject` instances,
- ♦ setting or retrieving the selection mode,
- ♦ registering or unregistering selection listeners.

This interface also has a number of advanced features described in *Architecture of the tree component* of this section.

`IlpDefaultTreeSelectionModel` is the default implementation of `IlpTreeSelectionModel`.

The following code shows you how to access the `IlpTreeSelectionModel` of an `IlpTree`.

How to access the selection model of a tree component

```
IlpTree tree = ...
//...
// Retrieve the selection model
IlpTreeSelectionModel selectionModel = tree.getSelectionModel();
```

How to retrieve selected objects from the selection model

The following code sample shows you how to retrieve the selected `IlpObject` instances from the selection model.

```
Collection selection = selectionModel.getSelectedObjects();
Iterator it = selection.iterator();
while (it.hasNext()) {
   IlpObject node = (IlpObject)it.next();
      // Do what you want with the selected node
      // ...
}
```

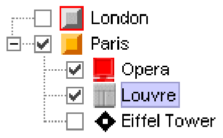
The sample ***<installdir>* /samples/tree/basic** demonstrates the use of the selection model.

Setting the selection look and feel

The `IlpTree` allows you to choose between the look-and-feel for the selection:

- HIGHLIGHT_SELECTION_LOOK_AND_FEEL: the standard mode, where a selected cell appears highlighted.
- CHECKBOX_SELECTION_LOOK_AND_FEEL: in this mode, a check box is displayed next to each tree node. A check mark indicates that the corresponding node is selected. Clicking a check box or a cell switches the selection state of the corresponding node. In this mode, the highlighting only indicates which cell has the focus.

Note: In this mode, collapsing a node does not deselect its descendants.



Sample tree using the check box selection look and feel

The following code shows you how to activate the CHECKBOX_SELECTION_LOOK_AND_FEEL.

How to activate a selection look and feel

```
IlpTree tree = ...
// Activate the CHECKBOX_SELECTION_LOOK_AND_FEEL
tree.setSelectionLookAndFeel
  (IlpTreeView.CHECKBOX_SELECTION_LOOK_AND_FEEL);
```

Filtering the tree nodes

The tree component allows you to filter the nodes that are displayed. To do so, attach an instance of `IlpFilter` to the tree component by using the method `setFilter`. The `accept()` method of the filter object will be invoked whenever the tree is prompted to display an `IlpObject`. If the method returns `false`, the object will not be shown in the tree. In the same way, an object will not be shown if its parent is not displayed.

For example, write the following code to show only objects of the class `IltNetworkElement`.

How to filter objects to be shown in the tree component

```
IlpTree tree = // ...
// Create a new IlpFilter instance
IlpFilter filter = new IlpFilter(){
  // This method is called for every object in the data source
  public boolean accept (Object object){
    IlpObject ilpObject = (IlpObject)object;
    IlpClass clz = ilpObject.getIlpClass();
    // Check if the class == IltNetworkElement
    return clz.equals(IltNetworkElement.GetIlpClass());
  }
// Set the filter to the tree
tree.setFilter(filter);
```

All the objects are refiltered whenever a new filter is set. If the filter is null (which is the default), all the objects under the root nodes will be displayed.

To retrieve the active filter, use the method `getFilter()`.

Note: The filtering takes actually place at the adapter level.

Accepted and excluded classes

You can specify the business objects that will be represented or not in the tree component depending on their business classes. To do so, you need to specify the business classes to be accepted or excluded using methods `setAcceptedClasses` or `setExcludedClasses` in the tree component adapter. To retrieve the adapter, use the `getAdapter` method. The adapter must be an instance of a subclass of `IlpAbstractTreeAdapter`.

How to specify excluded classes in the tree component

You can specify that business objects from specific business classes are not represented in the tree component. You can do that using the API, `setExcludedClasses(java.util.List)` method, or using CSS.

The following example shows you how to prevent objects from business classes `IltAlarm` and `IltLed` to be represented:

```
Adapter {
  excludedClasses[0]: "ilog.tgo.model.IltAlarm";
  excludedClasses[1]: "ilog.tgo.model.IltLed";
}
```

How to specify an accepted class in the tree component

By default, all business classes are accepted by the tree component. If you want to specify exactly which business classes to represent, you should combine the list of excluded and accepted classes, so that you exclude all business classes except those that are marked in the accepted class list.

In the following example, the tree component is configured in a way that it graphically represents only business objects from the class `IltNetworkElement`.

```
Adapter {
  excludedClasses[0]: "ilog.tgo.model.IltAlarm";
  excludedClasses[1]: "ilog.tgo.model.IltObject";
  acceptedClasses[0]: "ilog.tgo.model.IltNetworkElement";
}
```

Note: The filtering that is performed through the use of the accepted and excluded class lists takes actually place at the adapter level.

Please refer to *The Adapter rule* to know how to configure excluded and accepted classes through CSS.

Sorting the tree nodes

The tree component allows you to sort the displayed nodes. By default, the tree adapter sorts nodes using an arbitrary order (see `ARBITRARY_COMPARATOR`) in which the same set of objects is always displayed in the same order.

You can specify your own order by providing an object that implements the `Comparator` interface before the nodes are created. The `compare()` method of the comparator object will be called for pairs of `IlpObject` instances in the data source to establish a sort order. Objects are ordered locally, that is, within their parent node.

To retrieve the current sort order, use the method `getSortComparator`.

All the objects are sorted again when a new comparator is set.

For convenient sorting based on the attribute values of the business objects, JViews TGO provides the predefined `IlpAttributeComparator` class as an implementation of the `Comparator` interface. This class sorts objects based on the values of one or more of their attributes.

How to sort tree nodes

The following code sample shows you how to use `IlpAttributeComparator` to implement the sorting of tree nodes. This sample is available in ***<installdir>*** **/samples/framework/datasource-explorer** where `<installdir>` is the directory where you have installed JViews TGO.

```
// Create a comparator to sort all the nodes in the tree
// Use the predefined class IlpAttributeComparator to sort
// The first sort parameter is directory, so directories appear before files
// then the 'name' attribute
// then the 'path' attribute, as name is sometimes null (for roots)
IlpAttributeComparator sorter =
    new IlpAttributeComparator(dataSource.fileClass.getAttribute("directory")
,
          false);

sorter.setDirectionAndOrder(dataSource.fileClass.getAttribute("name"),2,true)
;

sorter.setDirectionAndOrder(dataSource.fileClass.getAttribute("path"),3,true)
;

// Set the sorter to the tree BEFORE setting the data source
treeComponent.getAdapter().setComparator(sorter);
```

Controlling the display of objects as tree leaves

By default, the tree adapter considers an object to be a tree leaf when the method `getContainerInterface(java.lang.Object)` returns `null` for that object.

You can have a finer control over whether an object should be considered as a tree leaf or not by means of the property `expansion`. By default, this property is set to the value `ExpansionType.IN_PLACE`, which produces the default behavior described earlier. To have an object considered as a leaf, even if the method `getContainerInterface(java.lang.Object)` does not return `null`, you must set this property to `ExpansionType.NO_EXPANSION`. For information on how to set a property value, see Introducing cascading style sheets.

Note: If an object is a leaf, that is, if its method `getContainerInterface(java.lang.Object)` returns `null`, there is no way to represent it as a tree branch.

Setting a list of origins

The tree component takes by default as root nodes all the objects in the data source that do not have a parent. However, you can explicitly select the root nodes to be displayed through the adapter that forms a bridge between the data source and the tree component. To retrieve the adapter, use the `getAdapter()` method. The adapter for the tree component must be an instance of a subclass of `IlpAbstractTreeAdapter`.

The root nodes can be changed by modifying the list of *origins* for the adapter. These origins are set and retrieved as `IlpObject` identifiers.

The method `getOrigins` allows you to get the list of current origins. The method `isShowingOrigin` indicates whether the origins themselves or their child objects are represented as root nodes. By default, the list of origins is empty and the origins are not shown, which means that all objects without a parent are shown as root nodes. Thus, the entire contents of the data source are displayed in the tree.

Note: The origins are specified using identifiers, not the `IlpObject` instances. You can retrieve the identifier of an `IlpObject` with the `getIdentifier()` method of the object.

To change the list of origins, use the `setOrigins` method. This method takes a list of business object identifiers as its first parameter. Its second parameter is a Boolean flag that indicates whether or not the origins themselves should be shown as root nodes.

Calling this method with an empty list and the second parameter set to `true` empties the tree:

```
setOrigins(Collections.EMPTY_LIST, true);
```

Calling the method with an empty list and the second parameter set to `false` restores the default; that is, all the objects in the data source are shown:

```
setOrigins(Collections.EMPTY_LIST, false);
```

How to show an object as the root node of a tree

To show only a given `IlpObject` as the root node of a tree, use the following code:

```
IlpTree tree = ....;
IlpObject originObject = .....;
java.util.List originList = new ArrayList();
originList.add(originObject.getIdentifier());
tree.getAdapter().setOrigins(originList, true);
```

For additional methods to help you manage origins, see `IlpAbstractHierarchyAdapter`.

Architecture of the tree component

Describes the classes and features of the tree component specific to each of the three modules of the MVC architecture, and also explains the role of the adapter.

In this section

Class overview

The tree component is internally based on the MVC architecture like the other JViews TGO components, which means that it has a model, a view, and a controller associated with it. For a general introduction to the MVC architecture, see *Architecture of graphic components*.

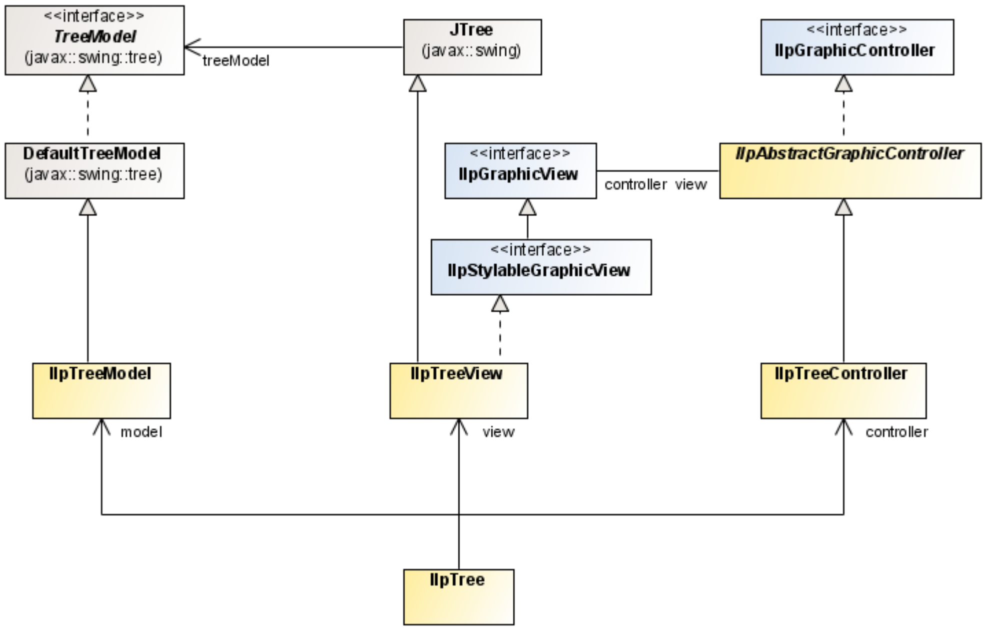
This topic describes the classes that you can use to create and manage trees. For a more detailed description, refer to the `overview-summary`. The classes are organized as follows:

- *MVC (model, view, controller) architecture*
- *Representation model and representation objects*
- *Graphic view and renderer*
- *Controller and interactors*

MVC (model, view, controller) architecture

The MVC architecture for the tree component is implemented by the following classes (see *Model, view, and controller for the tree component*):

- `IlpTreeView` is the view module of the tree in the MVC architecture. It defines a Swing-based tree used to display instances of `IlpTreeNode`. It uses an `IlpTreeModel` as entry model.
- `IlpTreeModel` is the model module of the tree in the MVC architecture. It is the representation model, and contains instances of `IlpTreeNode`.
- `IlpTreeController` is the controller module of the tree in the MVC architecture. It allows you to customize the behavior of the `IlpTreeView`.
- `IlpTree` component encapsulates an `IlpTreeModel`, an `IlpTreeView` and an `IlpTreeController`.



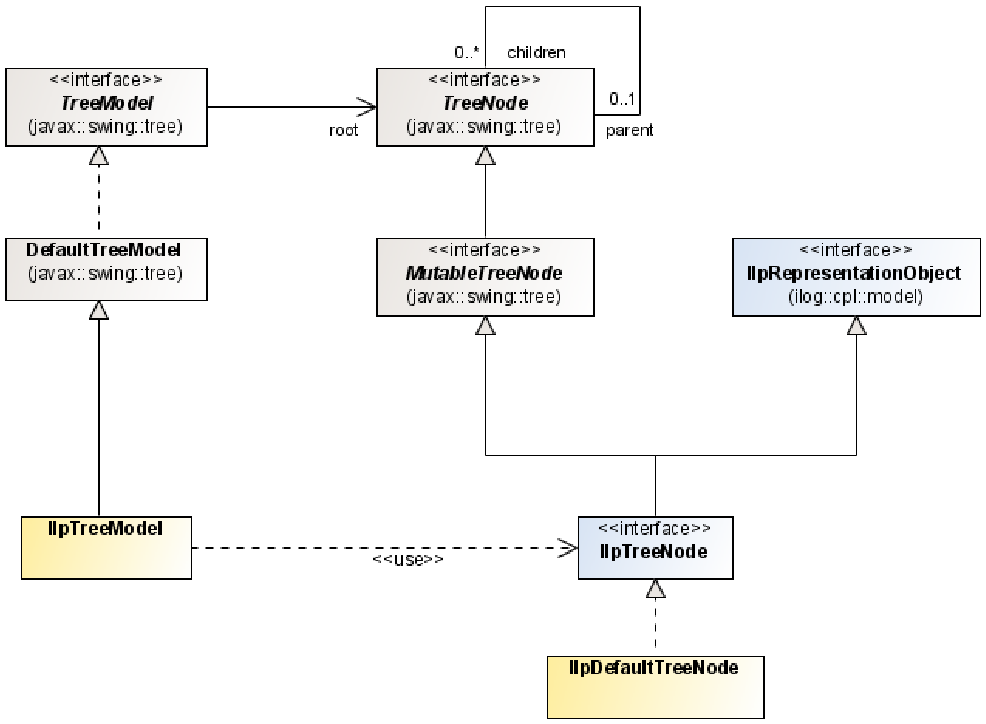
Model, view, and controller for the tree component

For general information about the model, the view, and the controller, see *Architecture of graphic components*.

Representation model and representation objects

The representation model for the tree component is implemented by the following classes (see *Representation model and representation objects for the tree component*):

- ♦ The `IlpTreeModel` class is the model module in the MVC architecture of the tree. It is the representation model and contains instances of `IlpTreeNode`.
- ♦ The `IlpTreeNode` interface defines the representation objects for tree nodes. The `IlpDefaultTreeNode` class is the default implementation of `IlpTreeNode`.



Representation model and representation objects for the tree component

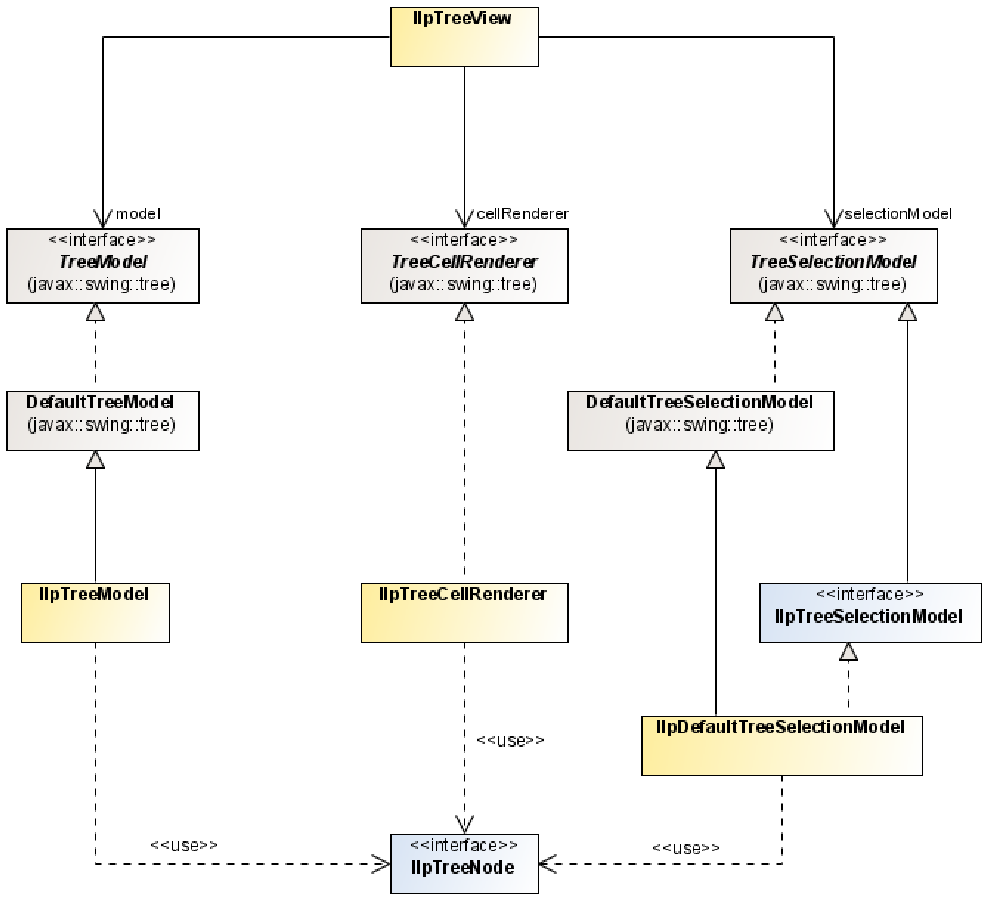
For general information about the representation model and representation objects, see *Architecture of graphic components*.

Graphic view and renderer

The graphic view and the renderer are implemented by the following classes (see *Graphic view and renderer for the tree component*)

- The `IlpTreeCellRenderer` is used to render tree nodes. It is based on the CSS configuration, which means that it uses the properties defined in the style sheet for the nodes. The default rendering is performed by a `JLabel`. If the "`class`" property is specified in the CSS file, it defines the `IlvGraphic` or `JComponent` that will be used to render the tree node. See *Using your own graphic representation* for details.

- The `IlpTreeSelectionModel` interface defines the requirements for a selection model containing `IlpTreeNode`. The `IlpDefaultTreeSelectionModel` class is the default implementation of `IlpTreeSelectionModel`.



Graphic view and renderer for the tree component

Controller and interactors

The controller is implemented by the `IlpTreeController` class, which allows you to register pop-up menus and interactors.

For general information about the controller, see *The controller*.

No special interactor is provided for the tree component.

The model

The tree model provides the representation objects to be displayed in the tree component. It can be connected to a back-end application by means of an adapter.

The representation model of the JViews TGO tree component includes the following classes:

♦ `IlpTreeModel` is a container of the tree representation objects that implements the characteristics specific to the JViews TGO tree model.

♦ `IlpTreeNode` defines the requirements associated with a node in an `IlpTreeModel`. `IlpDefaultTreeNode` is the default implementation of `IlpTreeNode`.

Note: `IlpTreeModel` has a predefined root node that you should not remove. The `IlpTreeView` automatically hides this node. Consequently, you cannot use the `setRoot()` method of `IlpTreeModel` to modify the model. A convenient `clear()` method removes all the nodes in a single operation, except the root node.

Integration with the back-end

If the tree is to be connected to a back-end application, a data source and an adapter must be instantiated. The adapter translates insertions, removals, and updates of `IlpObject` instances in the data source into insertions, removals, and updates of `IlpTreeNode` instances in the tree model.The adapter must be connected both to the data source and to the tree model.

The tree automatically creates an appropriate adapter (of class `IlpContainmentTreeAdapter`). If you want to use a different adapter, you will need to create one and connect it to both the data source and the tree component. The adapter must be an instance of a subclass of `IlpAbstractTreeAdapter`.

The following code connects an `IltDefaultDataSource` to an `IlpTree` by means of a custom adapter.

How to connect a data source to a tree Component with a custom adapter

```
IlpTree ilpTree = new IlpTree();
IltDefaultDataSource dataSource = new IltDefaultDataSource();
// Create an instance of a custom adapter
MyAdapter adapter = new MyAdapter(tree.getContext());
// Connect the adapter to the data source
adapter.setDataSource(dataSource);
// Connect the adapter to the tree component
ilpTree.setAdapter(adapter);
```

For more information about data sources, refer to Introducing business objects and data sources. For more information about adapters, refer to *Architecture of graphic components*.

The view

The tree view is responsible for displaying the graphic objects (tree nodes) and as such corresponds to the visible part of the architecture.

You can customize the representation of tree nodes by:

- ♦ *Using your own graphic representation*
- ♦ *Using an arbitrary TreeCellRenderer*

These ways are provided in addition to the CSS-based customization of the default tree renderer as explained in *Customizing the rendering of tree nodes*.

Using your own graphic representation

You can create an `IlvGraphic` or a `JComponent` directly by declaring the `class` property and setting it in a CSS file.

The `class` property name is a reserved keyword that indicates the class name of the generated graphic object. JViews TGO provides a predefined representation for the objects in all graphic components, which means that the `class` property is optional. It can be used when you want to replace the predefined representation.

```
object {
    class: ilog.views.sdm.graphic.IlvGeneralNode;
    foreground: red;
}
```

`IlpTreeCellRenderer` takes the information present in the CSS files to define how a tree node is rendered. If the property `class` is specified, it defines the `IlvGraphic` or `JComponent` that will be used to render the tree node.

To use a cascading style sheet `class` property in a tree view, do the following:

1. Implement your own graphic object or use an existing one, either as an `IlvGraphic` or as a `JComponent`.
2. Configure the properties of the objects or classes you want to represent with this property using CSS.
3. Load this CSS file in the tree component as illustrated in *How to load a CSS file in a tree component*.

For example, the sample ***<installdir>* /samples/tree/customClasses** shows how to use an `IlvGraphic` to render tree nodes. In this sample, the tree nodes are rendered as IBM® ILOG® JViews composite graphics where the label color changes from black to red when the object is selected.

```
object."Workstation" {
  class: 'ilog.views.graphic.composite.IlvCompositeGraphic';
  layout: @+attachmentLayout;
  children[0]: @+wsBase;
  children[1]: @+wsLabel;
  constraints[1]: @+wsLabelConstraint;
}

Subobject#attachmentLayout {
  class: 'ilog.views.graphic.composite.layout.IlvAttachmentLayout';
}
Subobject#wsBase {
  class: 'ilog.views.graphic.IlvIcon';
  image: '@|image("workstation.png")';
}
Subobject#wsLabel {
  class: 'ilog.views.graphic.IlvText';
  label: @name;
  foreground: black;
  font: 'arial-bold-12';
}
Subobject#wsLabel:selected {
  foreground: red;
}
Subobject#wsLabelConstraint {
  class: 'ilog.views.graphic.composite.layout.IlvAttachmentConstraint';
  hotSpot: Left;
  anchor: Right;
  offset: 3,0;
}
```

Using an arbitrary TreeCellRenderer

The renderer used by an `IlpTreeView` is an `IlpTreeCellRenderer`. You can replace it with you own renderer, using the `IlpTreeView.setCellRenderer()` method. This renderer must implement the `TreeCellRenderer` interface.

To write your own renderer, do the following:

1. Implement the `javax.swing.tree.TreeCellRenderer` interface.
2. Configure your tree component to use your renderer.

   The following code extract shows how to configure an `IlpTreeView` to use the `MyTreeCellRenderer` class to render all tree nodes:

   ```
   IlpTreeView view = //...
   view.setCellRenderer(new MyTreeCellRenderer());
   ```

The controller

The `IlpTreeController` class represents the controller module of the MVC architecture. It can be attached to a tree view by using the following code.

How to attach a controller to the tree view

```
IlpTreeView treeView = new IlpTreeView();
IlpTreeController treeController = new IlpTreeController();
treeView.setController(treeController);
```

Note that a controller is automatically attached to a tree view when a tree component is instantiated:

```
IlpTree treeComponent = new IlpTree();
IlpTreeController controller = treeComponent.getController();
// treeComponent.getView().getController() == controller
```

The tree controller is responsible for storing and setting the interactors to the view and to the objects.

When a controller is attached to a tree view, it listens to the keyboard, mouse, and focus events that occur in the tree and transfers them to the view interactor set to the controller.

For more details, see *Interacting with the tree view*.

There can be one and only one controller per view.

The adapter

The tree adapter converts business objects retrieved from the associated data source to tree nodes. It is defined by the class `IlpContainmentTreeAdapter`.

The tree adapter retrieves structural information (that is parent/child relationship) about business objects from the associated data source and determines whether an object should appear as a root representation object.

The default tree adapter implementation provides the following services:

♦ Filtering the tree nodes. For details, see *Filtering the tree nodes*

♦ Sorting the tree nodes. For details, see *Sorting the tree nodes*

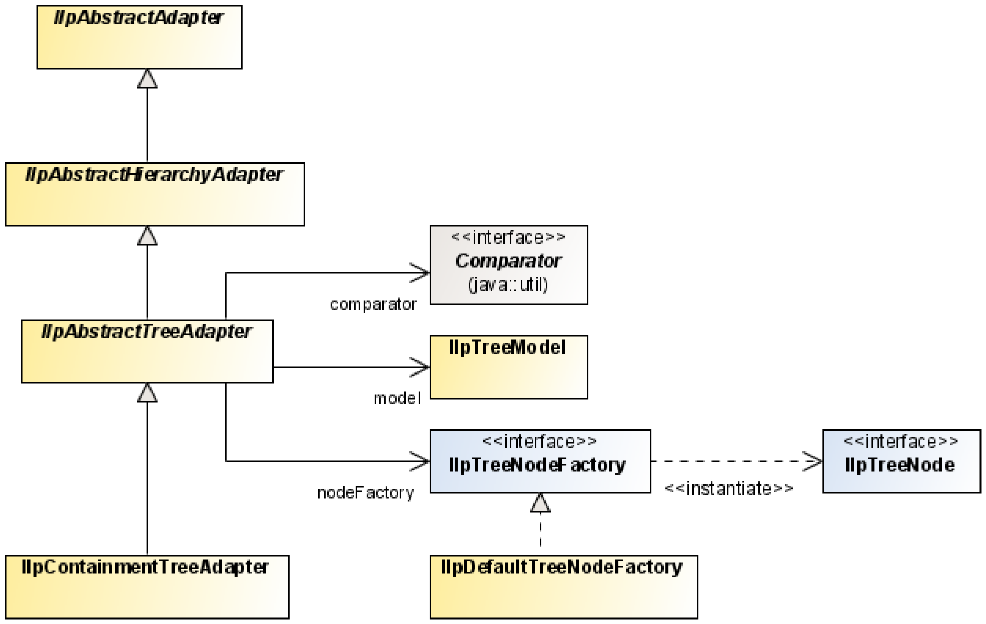
♦ Controlling the display of objects as tree leaves. For details, see *Controlling the display of objects as tree leaves*

♦ Setting a list of origins. For details, see *Setting a list of origins*.

♦ Creating the tree nodes. For details, see *Representation object factory*.

♦ Loading the tree nodes on demand. For details, see *Expansion strategy*.

The following figure shows the tree adapter classes.



Containment tree adapter classes

You can create a containment tree adapter implicitly by instantiating the `IlpTree` component as shown in the following example.

How to create a tree adapter by instantiating a tree component

```
IlpTree ilpTree = new IlpTree();
IlpDataSource dataSoure = new IlpDefaultDataSource();
ilpTree.setDataSource(dataSource);
```

How to retrieve a tree adapter

```
IlpAbstractTreeAdaper adapter = ilpTree.getAdapter();
```

Representation object factory

The tree adapter converts business objects retrieved from the associated data source to tree nodes. The new representation objects are created by a representation object factory. The factory interface varies according to the type of adapter. The containment tree adapter uses by default an `IlpDefaultTreeNodeFactory` that creates representation objects of type `IlpDefaultTreeNode`.

Expansion strategy

The tree adapter uses an expansion strategy to identify whether objects should be loaded or not in the tree model. An expansion strategy defines how an object is going to behave when an expansion is requested, for example, when the user opens a tree node by double-clicking or using the tree expansion handles. The expansion strategy indicates whether load on demand is implemented and provides methods to load and release child nodes.

The tree adapter uses an expansion strategy factory to define the expansion strategy for each tree node that it creates. The default expansion strategy factory implementation (`IlpDefaultTreeExpansionStrategyFactory`) verifies the property "`expansion`" defined for each business object in the cascading style sheet loaded in the component.

The default tree expansion strategy factory supports three types of expansion strategies:

♦ `IN_PLACE`: loads the child objects on demand, as the user expands the parent tree node. In this expansion strategy, tree nodes are considered as parent nodes, only when they have containment relationships defined in the attached data source, through the `IlpContainer` interface. The child objects should already be loaded in the data source, and should be visible according to the data source filter, if there is one defined.

♦ `IN_PLACE_MINIMAL_LOADING`: loads the child objects on demand, as the user expands the parent tree node. All tree nodes with this expansion strategy are considered as possible parent nodes, and therefore are represented with an expansion icon. If the tree node does not contain child objects, the expansion icon will disappear when the expansion is executed for the first time.

♦ `NO_EXPANSION`: expansion is not supported by the tree node.

For information on how to customize the business object expansion type, see Customizing the expansion of business objects .

The expansion strategy factory can be customized for the adapter either through CSS or through the API. See *Configuring the tree component*.

Editing

Adapter interfaces are read-only, meaning that they do not perform editing operations on the representation objects they create.

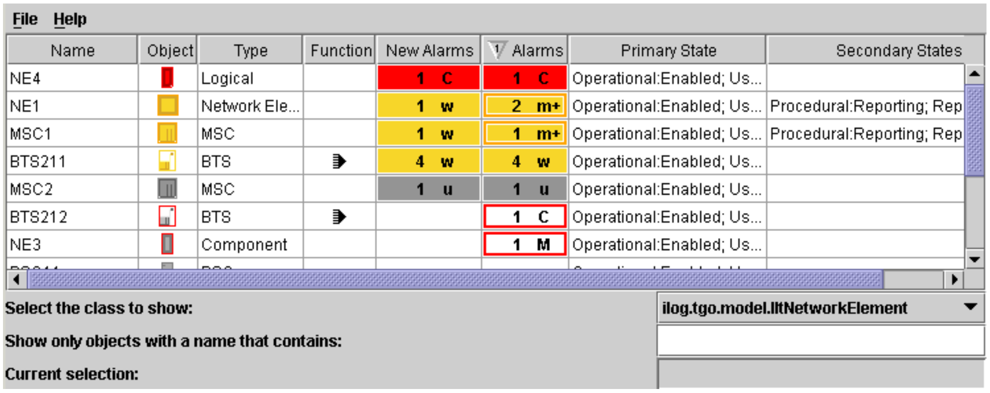
Table component

Describes the table component, which is one of the four graphic components provided in IBM® ILOG® JViews TGO. It displays data in a two-dimensional table format, composed of a set of rows and columns and of a header row. The rows in a table typically show business objects, while the columns show attributes of these objects, such as an FDN (Fully Distinguished Name), a severity, or a performance.

In this section

Introducing the table component

The JViews TGO table component is based on the Swing table component. It displays objects in rows and their attributes in columns.



Name	Object	Type	Function	New Alarms	Alarms	Primary State	Secondary States
NE4	■	Logical		1 C	1 C	Operational:Enabled; Us...	
NE1	■	Network Ele...		1 w	2 m+	Operational:Enabled; Us...	Procedural:Reporting; Rep
MSC1	■	MSC		1 w	1 m+	Operational:Enabled; Us...	Procedural:Reporting; Rep
BTS211	■	BTS	➧	4 w	4 w	Operational:Enabled; Us...	
MSC2	■	MSC		1 u	1 u	Operational:Enabled; Us...	
BTS212	■	BTS	➧		1 C	Operational:Enabled; Us...	
NE3	■	Component			1 M	Operational:Enabled; Us...	

Select the class to show: ilog.tgo.model.IltNetworkElement

Show only objects with a name that contains:

Current selection:

You can customize the rendering of cells, headers, and the component view itself.

The table component is connected to a data source, from which it retrieves the business objects to be displayed. By default, the table displays all the objects contained in the data source. However, it is possible to restrict the contents displayed by:

- Setting a filter
- Selecting the objects to be displayed based on their name
- Specifying an accepted class of objects

The most notable features of the table component include:

- Various selection modes
- Moving and resizing of columns
- Sorting columns
- Filtering at the adapter level and at the componont level

  Possibility to refine the filtering on a specific class of objects.
- Searching for a string in the table

The table component is implemented by the class `IlpTable` which is a Swing `JComponent` that can be directly inserted in a `JPanel`.

`IlpTable` provides the API for the most common uses of the table component, such as:

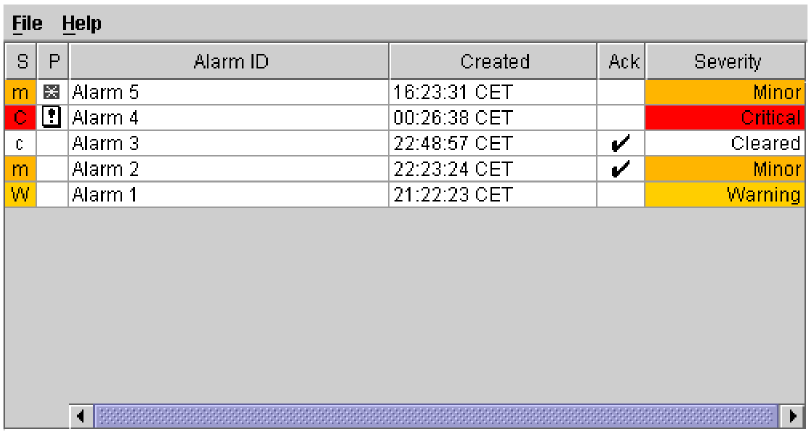
- setting or retrieving the associated data source: `getDataSource()`, `setDataSource(ilog.cpl.datasource.IlpDataSource)`

- accessing and modifying the selection: `getSelectionModel()`, `setSelectionModel()`, `addSelectionObject()`, `removeSelectionObject()`, `clearSelection()`, `isObjectSelected()`, `getSelectedObject()`, `getSelectedObjects()`
- setting or retrieving the view interactor: `setViewInteractor()`, `getViewInteractor()`
- filtering the table rows: `setFilter()`, `getFilter()`, `setAcceptedClass()`, `getAcceptedClass()`
- sorting the table columns: `addSortingCriteria()`, `getSortingOrder()`

`IlpTable` also acts as a façade for a number of lower-level components that it contains. These components provide more detailed APIs and advanced services. They are described in *Filtering rows*.

Creating a table component: a sample

This section shows you how to create the following table for displaying alarms.



S	P	Alarm ID	Created	Ack	Severity
m	▨	Alarm 5	16:23:31 CET		Minor
C	!	Alarm 4	00:26:38 CET		Critical
c		Alarm 3	22:48:57 CET	✓	Cleared
m		Alarm 2	22:23:24 CET	✓	Minor
W		Alarm 1	21:22:23 CET		Warning

An alarm table

The following example shows how to create a table component, how to connect it to a data source, how to fill the data source from an XML file that describes the business objects, and how to configure the graphic representation of the table and the objects.

How to create a table component

1. Initialize the JViews TGO library.

   Prior to using any JViews TGO API, `IltSystem.Init` must be called.

   ```
   IltSystem.Init("deploy.xml");
   ```

   `deploy.xml` is a deployment descriptor file that defines the path to the application resources to be used:

   ```
   <deployment>
     <urlAccess>
       <!-- Add relative path to sample root directory -->
       <relativePath>../..</relativePath>
     </urlAccess>
   </deployment>
   ```

2. Create a data source and fill it with business objects from an XML file.

   A data source contains business objects to be displayed in the graphic components.

   ```
   IltDefaultDataSource dataSource = new IltDefaultDataSource();
   ```

```
// Fill the data source with an XML file containing the business objects
dataSource.parse("alarms.xml");
```

3. Create a table component.

```
IlpTable tableComponent = new IlpTable();
```

4. Connect the data source to the table component.

   The table shows only instances of a single class (and its subclasses). Here, we retrieve the Alarm class that was created when the data file was parsed and ask the table to display all instances of this class that are contained in the data source.

```
// Get the default context of the application
IlpContext context = IltSystem.GetDefaultContext();
// Get the class manager, which contains all business classes
IlpClassManager classManager = context.getClassManager();
// Retrieve the Alarm class
IlpClass alarmClass = classManager.getClass("Alarm");
// Connect the data source to the table component indicating
// which class it accepts (Alarm class)
tableComponent.setDataSource(dataSource, alarmClass);
```

5. Configure the graphic representation of the table component.

   Once you have added objects to be displayed in the table component, you can specify how the objects and the table itself should be represented. To do so, you can use cascading style sheets, as follows:

```
String[] css = new String[] { "table.css" };
try {
  tableComponent.setStyleSheets(css);
} catch (Exception e) {
}
```

Configuring the table component

Identifies the rendering information necessary to display a table.

In this section

Introduction

The table component can be customized either through a CSS configuration file or through the API, the easiest and preferred way being the CSS configuration. You also have the possibility to load a project file which combines the CSS configuration and the table data.

You can customize the table view with properties such as background, grid color, cell and header renderer. You can also customize the table adapter and the way business objects are represented.

Configuring the table component through a CSS file

You can customize the following features in a CSS file:

- Table view
  - Background color
  - Grid color
  - Show/hide grid
  - Show/hide horizontal lines
  - Show/hide vertical lines
  - Row margin
  - Column margin
  - Fixed column count
  - Auto resize mode
  - Selection mode
  - Reordering
  - Header renderer
  - Default renderer
- Table adapter
  - Filter
  - Accepted class
  - Excluded classes
  - Table row factory
- Table column
  - Header renderer
  - Cell renderer
- Table row
  - Row height
- Table cell
- Table controller
  - View interactor

- Header interactor
- Cell interactor
- Object interactor

How to load a CSS file in a table component

The table configuration can be split accross several CSS files that you can load by:

♦ Specifying a project file that lists the style sheets and the data file to be loaded in the component (see *Loading a project file*). The project file will be as follows:

```
<?xml version="1.0"?>
<tgo xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/project.xsd"
style="configurationFile1.css,configurationFile2.css">
  <datasource javaClass="ilog.tgo.datasource.IltDefaultDataSource"
fileName="table.xml"/>
</tgo>
```

If the settings in two of the CSS files disagree, the effect depends on the order of the filenames in the list: the last file mentioned takes precedence over the first file.

♦ Using the method `IlpTable.setStyleSheets` as follows:

```
tableComponent = new IlpTable();
try {
  tableComponent.setStyleSheets(new String[] {
                      myConfigurationFile1,myConfigurationFile2 });
} catch (Exception e) {
}
```

If the settings in the CSS files disagree, the effect depends on the order of the filenames in the list: the last file listed takes precedence over the first one.

How to configure a table component in a CSS file

The following example shows you how you can customize the table component itself. It is based on the CSS file located in ***<installdir>*** **/samples/table/styling/srchtml/table.css.html**

where `<installdir>` is the directory where you have installed JViews TGO.

The configuration in CSS is organized as a set of rules that define properties.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Table Component configuration
// Type: Table
//
// This is the main selector when customizing
```

```
// a table component. It identifies the
// sub-components that will be addressed in the
// CSS customization. In the Table Component, it
// is possible to customize the view, controller
// and adapter using CSS.
//
// List of available properties:
// - view: boolean
// - interactor: boolean
// - adapter: boolean
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Table {
  view: true;
  interactor: true;
}
```

The Table rule

This rule specifies the elements of the table component that will be customized. It contains Boolean flags that indicate whether some specific customizable properties are present. For example, the customization of the property `adapter` is not taken into account unless `adapter: true` is declared in the Table rule.

This feature provides powerful cascading possibilities. You can define adapter customizations in a default CSS file and turn them on or off in another CSS file.

The following CSS properties affect the table component:

CSS properties of the table component

CSS property	Type of value	Default	Usage
`view`	`boolean`	`false`	Enables the customization of the table view.
`interactor`	`boolean`	`false`	Enables the customization of the table interactors.
`adapter`	`boolean`	`false`	Enables the customization of the table adapter.

The View rule

This rule specifies the properties that are applied to the table view.

The following CSS properties affect the appearance of the table view:

CSS Properties of the table view

CSS property	Type of value	Default	Usage

`autoResizeMode`	`IlpTableResizeMode`	`AUTO_RESIZE_OFF`	Resize mode of the
			values are:
			`AUTO_RESIZE_OF`
			`AUTO_RESIZE_AL`

CSS property	Type of value	Default	Usage
			AUTO_RESIZE_LA
background	Color	null	Color to be used in the table view. By d gray.
gridColor	Color	null	Color to be used fo
showGrid	Boolean	true	Determines whethe or not.
showHorizontalLines	Boolean	true	Determines whethe lines are displayed
showVerticalLines	Boolean	true	Determines whethe lines are displayed
rowMargin	Integer	1	Height in pixels of t each row.
columnMargin	Integer	1	Width in pixels of th each column.
reorderingAllowed	Boolean	true	Determines whethe to reorder the colur
fixedColumnCount	Integer	0	Defines the numbe
headerRenderer	IlpTableHeaderRenderer	IlpTableHeaderRenderer	The renderer used graphic representat the table columns.
defaultRenderer	IlpTableCellRenderer	IlpTableCellRenderer	The renderer used graphic representat
selectionMode	IlpTableSelectionMode	MULTIPLE_OBJECTS_SELECTION	Selection mode of t values are: EMPTY_SELECTIO SINGLE_OBJECT_ MULTIPLE_OBJEC SINGLE_ATTRIBU MULTIPLE_ATTRI SINGLE_CELL_SE MULTIPLE_CELLS_

How to configure a table view in a CSS file

Prior to configuring the table view, you need to configure the table component so that the view configuration is enabled:

```
Table {
  view: true;
}
```

Then, you can customize the view properties in the View rule as illustrated by the following code extract. Refer to The CSS specification in the *Styling* documentation for details about the CSS syntax.

```
View {
  reorderingAllowed : true;
  autoResizeMode : AUTO_RESIZE_OFF;
  selectionMode : MULTIPLE_OBJECTS_SELECTION;
  fixedColumnCount: 1;
}
```

Refer to `IlpViewRenderer` for more information.

The Interactor rule

This rule specifies the properties that are applied to the table controller. The following CSS properties affect how the table component handles mouse and keyboard events:

CSS Properties of the table interactor

Property name	Type	Default value	Usa
`viewInteractor`	`IlpViewInteractor`	`IlpDefaultTableViewInteractor`	Defi the inter that hand even the v
`headerInteractor`	`IlpDefaultTableHeaderInteractor`	`IlpDefaultTableHeaderInteractor`	Defi the inter that hand even the t head

Note: Events are only handled by the header interactor if the current view interactor is an `IlpDefaultViewInteractor`.

How to configure a table interactor in a CSS file

Prior to configuring the table interactor, you need to configure the table component so that the interactor configuration is enabled:

```
Table {
  interactor: true;
}
```

After that, you can customize the each interactor property in the Interactor rule as illustrated by the following code extract. Refer to The CSS specification in the *Styling* documentation for details about the CSS syntax.

```
Interactor {
  viewInteractor: @+viewInt;
  headerInteractor: @+headerInt;
}
Subobject#viewInt {
  class: 'ilog.cpl.table.interactor.IlpDefaultTableViewInteractor';
}
Subobject#headerInt {
  class: 'ilog.cpl.table.interactor.IlpDefaultTableHeaderInteractor';
}
```

The behavior of the view interactor is determined by the actions that are associated with user gestures and keystrokes. This behavior can also be customized through CSS. You can also configure a pop-up menu to be displayed in the table view or table header. For more information about interactor customization, refer to *Interacting with the table view* and *Interacting with the table cells*.

For more information, refer to `IlpInteractorRenderer`.

The Adapter rule

This rule controls the configuration of the table adapter. The table adapter is responsible for converting the business objects in the data source to representation objects (table rows) in the table component. It provides the following features:

- **Filtering**: applies a filter so that business objects currently in the data source are not mapped to representation objects in the table component.
- **Accepted class**: defines a specific class of business objects to be displayed in the table.
- **Excluded classes**: defines a list of business classes whose business objects will not be displayed in the table.

These table adapter features can be customized through CSS using the following properties:

CSS properties of the table adapter

Property name	Property type
`filter`	`ilog.cpl.util.IlpFilter`
`acceptedClass`	`String`
`excludedClasses`	list of `IlpClass`
`representationObjectFactory`	`ilog.cpl.table.IlpTableRowFactory`

How to configure a table adapter in a CSS file

Prior to configuring the adapter, you need to configure the table component so that the adapter configuration is enabled:

```
Table {
  adapter: true;
}
```

After that, you can customize each adapter property in the Adapter rule as illustrated by the following code extract. Refer to The CSS specification in the *Styling* documentation for details about the CSS syntax.

```
Adapter {
  filter: @+tableFilter;
  acceptedClass: 'ilog.tgo.model.IltNetworkElement;
  representationObjectFactory: @+repObjFactory;
}

Subobject#tableFilter {
  class: MyTableFilter;
}
Subobject#repObjFactory {
  class: MyRepresentationObjectFactory;
}
```

How to programmatically configure a table adapter using CSS

You can programmatically modify the CSS configuration of the default table adapter (`IlpTableListAdapter`) by using mutable style sheets through the `IlpMutableStyleSheet` API.

Important: The mutable style sheet is set to the adapter as a regular style sheet and is cascaded in the order in which it has been declared.

To use mutable style sheets:

1. Get the mutable style sheet.

You access the mutable style sheet through the `getMutableStyleSheet()` method in the table adapter API:

```
IlpMutableStyleSheet mutable = adapter.getMutableStyleSheet();
```

This method automatically registers the mutable style sheet in the adapter. You can manually instantiate an object of the class `IlpMutableStyleSheet` and register it yourself through the `setStyleSheet()` API:

```
IlpMutableStyleSheet mutable = new IlpMutableStyleSheet(adapter);
try {
  adapter.setStyleSheets(new String[] { mutable.toString() });
} catch (Exception x) {
  x.printStackTrace();
}
```

2. Set the CSS declarations.

Once you have the mutable style sheet, you can set the declarations you want:

```
mutable.setDeclaration("#myObjectId", "expansion", "NO_EXPANSION");
```

This creates the following CSS declaration into the mutable style sheet:

```
#myObjectId {
  expansion: NO_EXPANSION;
}
```

3. Register the mutable style sheet.

The mutable style sheet should be set as a regular style sheet for the adapter using the `setStyleSheet()` method:

```
try {
  adapter.setStyleSheets(new String[] { mutable.toString() });
} catch (Exception x) {
  x.printStackTrace();
}
```

4. Set and update the CSS declarations.

The mutable style sheet can be modified even after being registered to the adapter:

```
// Update the expansion type for 'myObjectId'
mutable.setDeclaration("#myObjectId", "expansion", "IN_PLACE");
// Add a new declaration
mutable.setDeclaration("#myOtherId", "expansion", "IN_PLACE");
```

Note: Like any style sheet, the mutable style sheet is lost when the `setStyleSheet()` API is invoked and a new set of style sheets is applied to the adapter.

How to customize the mutable style sheet

Reapplying a CSS configuration may be a heavy task, as the adapter may be forced to review filters, origins, recreate representation objects, and so on. It is important to use the mutable style sheet with care and to customize it properly to reapply the CSS wisely. To do so, there are two methods available in the `IlpMutableStyleSheet` API: `setUpdateMask()` and `setAdjusting()`.

1. `setUpdateMask()`

   This method controls what should be recustomized once a declaration of the mutable style sheet has been updated. The CSS configuration of the adapter is divided into two parts: *adapter customization* and *representation object* customization.

   The adapter customization handles the origins, filters, and so on:

```
Adapter {
  origins[0]: id0;
  origins[1]: id1;
  showOrigin: true;
  filter: @+myFilter;
}
```

   The representation object customization handles the expansion type of a representation object:

```
#myObjectId {
  expansion: IN_PLACE;
}
```

   The accepted values for `setUpdateMask()` are:

   ♦ `IlpStylable.UPDATE_COMPONENT_MASK`: Only the adapter part is recustomized.

   ♦ `IlpStylable.UPDATE_OBJECTS_MASK`: Only the representation object part is recustomized.

   ♦ `IlpStylable.UPDATE_ALL_MASK`: Bot the adapter and representation object parts are recustomized.

   ♦ `IlpStylable.UPDATE_NONE_MASK`: Nothing is recustomized.

   For example, if you update the expansion type of a representation object through the mutable style sheet, it is recommended that you set the update mask to `UPDATE_OBJECTS_MASK` as there is no need to reapply the CSS configuration for the adapter part:

```
mutable.setUpdateMask(IlpStylable.UPDATE_OBJECTS_MASK);
mutable.setDeclaration("object", "expansion", "IN_PLACE");
```

2. `setAdjusting()`

   This method is used when a series of declarations must be applied to the mutable style sheet. When the method is set to `true`, the mutable style sheet puts all the calls to `setDeclaration()` into a queue. When the method is set back to `false`, all the queued declarations are processed in a batch:

```
mutable.setAdjusting(true);
mutable.setDeclaration("#myObjectId", "expansion", "IN_PLACE");
mutable.setDeclaration("#myOtherId", "expansion", "IN_PLACE");
mutable.setAdjusting(false);
```

Configuring the table component through the API

For details of the classes involved in the architecture of the table component, see *Filtering rows*.

The following example shows how to configure the table view (`IlpTableView`) through the API. For details on programming the individual services, see *Table component services* .

How to configure the table view with the API

```
IlpTableView view = table.getView();

// Setting the selection mode
view.setSelectionMode(ListSelectionModel.SINGLE_OBJECT_SELECTION);

// Setting the table cell renderer
view.setDefaultRenderer(new MyTableCellRenderer());

// Setting the grid color
view.setGridColor(Color.blue);
```

The following example shows how to configure the table interactor through the API. For details, see *Interacting with the table view*.

How to configure the table interactor with the API

```
// Create the table, and retrieve the view interactor
IlpTable tableComponent = new IlpTable()
IlpViewInteractor viewInteractor = tableComponent.getViewInteractor();
// Create a Swing action
// We assume the MyAction class is defined elsewhere
Action myAction = new MyAction();
// Double-clicking the left mouse button will trigger myAction
viewInteractor.setGestureAction(IlpGesture.BUTTON1_DOUBLE_CLICKED,myAction);
```

The following example shows how to configure the table adapter through the API. See *Table component services* for details on programming the individual services.

How to configure the table adapter with the API

```
IlpListAdapter adapter = table.getAdapter();

// Accepted class
// (it is the same as adapter.setAcceptedClass)
table.setAcceptedClass(IltNetworkElement.GetIlpClass());

// Setting the filter
```

```
IlpFilter myFilter = new MyFilter();
// (it is the same as adapter.setFilter(myFilter)
table.setFilter(myFilter);

// Table Row Factory
IlpTableRowFactory myRowFactory = new MyTableRowFactory();
adapter.setRepresentationObjectFactory(myRowFactory);
```

Loading a project file

A project is a combination of style sheets that supply rendering information and a data source that supplies the data to be represented in a table component. A project is saved as an XML file with extension `.itpr`.

Loading a project file is the recommended way to configure a graphic component in Java™ as it is the fastest.

How to load a project file into a table component

The following code sample shows how to load a project file into a table component, using the method `setProject`.

```
IlpTable table = new IlpTable();
table.setProject(new URL("file:project.itpr");
```

The project is represented by the `IlpTGOProject` class, included in the package `ilog.cpl.project`. When a new project is created, the style sheet and data source are both null.

How to create a new project for the table component

The following code sample shows how to create a new project file by setting the style sheets and data source, then saving the project.

```
IlpTGOProject project = new IlpTGOProject();
project.setStyleSheet(new URL("file:example.css");
IltDefaultDataSource dataSource = new IltDefaultDataSource();
dataSource.setFileName("data.xml");
project.setDataSource(dataSource);
project.write(new URL("file:example.itpr");
```

Customizing column headers and rows

The table component uses a default renderer to render the column header (an instance of `IlpTableHeaderRenderer`).

This renderer is based on the CSS configuration, which means that it uses the properties defined in the cascading style sheets for the business objects, the attributes, and the table component itself.

How to modify the table column renderer

To modify the rendering of the table, you can do one of the following:

1. Replace the default renderer by another one using the method `setHeaderRenderer`.

Note: The Swing mechanism that allows you to associate a specific renderer with instances of a specific class in the table model is disabled.

2. Modify the JViews TGO default properties.

How to modify the default properties of a table

To modify the JViews TGO default properties in a table, create a custom CSS file and load it in your table component using the method `IlpTable.setStyleSheets`.

```
IlpTable tableComponent = new IlpTable();
String[] css = new String[] { "table.css" };
try {
  tableComponent.setStyleSheets(css);
} catch (Exception e) {
}
```

For a description of the properties that you can use in a CSS file to modify the rendering of the table header and of the table rows, refer to Customizing table column headers and rows.

Table component services

Describes the services that are available for a table: view services and adapater services.

In this section

Fixing columns in a table
Describes how to fix a set of columns on the left side of a table.

Moving columns
Describes how to rearrange columns using the API.

Searching for a string in a table
Explains how to search for a string in a table using the API.

Showing or hiding columns in a table
Explains how to show or hide columns in a table using the API.

Sorting columns
Explains how to sort columns using the API.

Adding new columns to the table
Describes how to add new columns using the API.

Filtering rows
Describes how to filter the rows in a table using the API.

Excluding table rows
Describes how to exclude business objects from the table using the API.

Introduction to table component services

Table component services are of two kinds:

- ♦ View services, related to the table view
  - *Selecting the accepted class of objects*
  - *Filling the table with business objects*
  - *Interacting with the table view*
  - *Interacting with the table cells*
  - *Handling the selection*
  - *Fitting to Contents*
  - *Resizing columns*
  - *Fixing columns in a table*
  - *Moving columns*
  - *Searching for a string in a table*
  - *Showing or hiding columns in a table*
  - *Sorting columns*
  - *Adding new columns to the table*
- ♦ Adapter services, related to the table model
  - *Filtering rows*
  - *Excluding table rows*

Selecting the accepted class of objects

The accepted class allows you to define the class of objects to be displayed in the table component. The table can display predefined business objects of the class `IltObject` or of a subclass of `IltObject`, or user-defined business classes.

You need to specify the accepted class when you create your table component. Once the accepted class has been set, the attributes present in this class determine the columns that will be displayed in the table.

How to set the accepted class

```
// Create a table component and connect it to a data source
IlpTable tableComponent = new IlpTable();
IltDefaultDataSource dataSource = new IltDefaultDataSource();
// The second parameter specifies which kind of object contained
// in the data source the table will show
tableComponent.setDataSource(dataSource, IltNetworkElement.GetIlpClass());
```

or, after connecting the table with a data source, by using the method `setAcceptedClass`:

```
tableComponent.setAcceptedClass(IltObject.GetIlpClass());
```

Filling the table with business objects

The table can be filled with business objects through an `IltDefaultDataSource`, as illustrated by the following code.

How to fill a table with business objects

```
// Create a table component and connect it to a data source
IlpTable tableComponent = new IlpTable();
IltDefaultDataSource dataSource = new IltDefaultDataSource();
// The second parameter specifies which kind of object contained
// in the data source the table will show
tableComponent.setDataSource(dataSource, IltNetworkElement.GetIlpClass());

// Create a business object and insert it in the data source
IltObjectState os = new IltBellcoreObjectState();
os.set(IltBellcore.State.EnabledActive);
IltNetworkElement london =
  new IltNetworkElement("Fax", IltNetworkElement.Type.Fax, os);
dataSource.addObject(london);
```

Note that a data source can also load an XML file containing business objects, as shown in *Creating a table component: a sample*.

You can also use project files to easily create and load data source business objects into your table component, as illustrated below:

How to fill a table with business objects using a project file

You can create the following project file to indicate the type of data source to be used and the XML file that contains the data source information:

```
<?xml version="1.0"?>
<tgo xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "ilog/cpl/schema/project.xsd">
  <datasource javaClass="ilog.tgo.datasource.IltDefaultDataSource"
fileName="network.xml"/>
</tgo>
```

Once the project file is specified, you can load it in the component using method `IlpTable.setProject(URL projectURL)` or `IlpTable.setProject(IlpProject project)`. See *Loading a project file* for details.

Interacting with the table view

The table component allows you to define behavior for the main view of the table, as well as specific behavior for the table header. In both cases, an interactor is used. The default interactors handle the basic interaction supported by the table (selection, moving columns, for example) and can be easily customized to:

♦ associate actions with mouse events and focus events,

♦ associate actions with keyboard events,

♦ build a pop-up menu to be displayed on the table.

Interacting with the main table view

A default view interactor (an instance of `IlpDefaultTableViewInteractor`) is associated with the main view of the table component. It is retrieved using the `getViewInteractor()` method of `IlpTable`.

How to associate an action with a gesture in a table component

You can associate actions with mouse events by using either CSS or the API. The following CSS extract shows how to proceed:

```
Table {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.table.interactor.IlpDefaultTableViewInteractor';
  action[0]: @+viewAction0;
}

Subobject#viewAction0 {
  class: 'ilog.cpl.interactor.IlpGestureAction';
  gesture: BUTTON1_DOUBLE_CLICKED;
  action: @+myAction;
}
Subobject#myAction {
  class: MyAction;
}
```

The same configuration can be achieved through the API, as follows:

```
// Create the table, and retrieve the view interactor
IlpTable tableComponent = new IlpTable()
IlpViewInteractor viewInteractor = tableComponent.getViewInteractor();
```

```
// Create a Swing action
// We assume the MyAction class is defined elsewhere
Action myAction = new MyAction();
// Double-clicking the left mouse button will trigger myAction
viewInteractor.setGestureAction(IlpGesture.BUTTON1_DOUBLE_CLICKED,
                         myAction);
```

.

Note: A gesture is a series of one or more atomic user input events that are meant to invoke a single action.

How to associate an action with a keyboard event in a table component

You can associate actions with keyboard events by using either CSS or the API. The following CSS extract shows how to proceed:

```
Table {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.table.interactor.IlpDefaultTableViewInteractor';
  action[0]: @+viewAction0;
}

Subobject#viewAction0 {
  class: 'ilog.cpl.interactor.IlpKeyStrokeAction';
  keyStroke: 'ctrl typed D';
  action: @+myAction;
}

Subobject#myAction {
  class: MyAction;
}
```

The same configuration can be achieved through the API, as follows:

```
// Create an action
Action myAction = new MyAction();
// Typing CTRL+D will trigger myAction
KeyStroke ctrlD = KeyStroke.getKeyStroke('D',java.awt.Event.CTRL_MASK);
viewInteractor.setKeyStrokeAction(ctrlD, myAction);
```

Both mouse and keyboard actions can be invoked anywhere in the table, except in the header.

Interacting with the table header

Usually, specific behavior is needed for the table header. For that reason, the table component also has a default interactor for the header. It is an instance of `IlpDefaultTableHeaderInteractor` and can be retrieved using the `getHeaderInteractor` method of `IlpTable`. This interactor can be customized in the same way as the interactor for the main view.

Note: If no header interactor is set to the table controller (that it, if it is null), the events are handled directly by the view interactor.

Using pop-up menus

The JViews TGO interactors allow you to create custom pop-up menus easily for the table. To do this, you should implement the `IlpPopupMenuFactory` interface. One abstract and two default menu factories are provided, which you can extend by subclassing:

♦ `IlpAbstractPopupMenuFactory` gives you easy access to the currently selected business objects.

♦ `IlpDefaultTableHeaderMenuFactory` allows you to show or hide a column and change the sorting criterion of a column.

♦ `IlpDefaultTableMenuFactory` allows you to fix columns and show the hidden columns.

The following code shows how to define a custom pop-up menu factory for the table view.

How to define a custom pop-up menu for the table view

You can customize a pop-up menu factory for the table view or the table header, either through CSS or through the API. The following CSS extract shows how to configure a CSS file to add a pop-up menu factory:

```
Table {
  interactor: true;
}

Interactor {
  viewInteractor: @+viewInt;
}

Subobject#viewInt {
  class: 'ilog.cpl.table.interactor.IlpDefaultTableViewInteractor';
  popupMenuFactory: @+viewPopupMenuFactory;
}
Subobject#viewPopupMenuFactory {
  class: MyTableMenuFactory;
}
```

The same configuration can be achieved through the API, as follows:

1. Create a class extending `IlpAbstractPopupMenuFactory` as follows:

```
public MyTableMenuFactory extends IlpAbstractPopupMenuFactory
{
  public JPopupMenu createPopupMenu
    (IlpObjectSelectionModel ilpSelectionModel)
  {
  // The following menu could be context-dependent
  JPopupMenu popupMenu = new JPopupMenu();
  // Create here the items and add them in the menu
  // ....
  return popupMenu;
  }
}
```

2. Use a pop-up menu in the table view or header:

```
IlpTable tableComponent = new IlpTable();
// Use a custom pop-up menu in the table view
IlpPopupMenuFactory tableMenuFactory = new MyTableMenuFactory();
tableComponent.getViewInteractor().setPopupMenuFactory(tableMenuFactory);
// Use the default header pop-up menu in the table header
IlpPopupMenuFactory headerMenu =
  new IlpDefaultTableHeaderMenuFactory();
tableComponent.getHeaderInteractor().setPopupMenuFactory(headerMenu);
```

For a detailed description of interactors and gestures, refer to *Interacting with the graphic components*.

Interacting with the table cells

Interacting with the table view describes how to set an interactor for the entire table view. You can also associate behavior with business objects (for a class or for individual objects), as well as with individual table cell instances. To do so, you use object interactors, which provide the same options as the view interactor, that is:

♦ Associating actions with mouse events

♦ Associating actions with keyboard events

♦ Defining a pop-up menu factory to build a pop-up menu that displays on representation objects

The object interactor handles any events occurring on the object with which the interactor is associated, provided the view interactor has enabled the use of object interactors. You can check this by means of the `isUsingObjectInteractor` method or modify it with the `setUsingObjectInteractor` method. Object interactors are enabled by default.

No default interactor is associated with any object. To associate actions with mouse or keyboard events or to define a pop-up menu factory, you first have to create an `IlpObjectInteractor`. You may use the `IlpDefaultObjectInteractor`, extend it, or create your own implementation.

How to associate an object interactor with a representation object in the table

You can associate an object interactor with a representation object by using either CSS or the API. The following CSS extract shows how to proceed:

```
Table {
  interactor: true;
}

object."ilog.tgo.model.IltNetworkElement" {
  interactor: @+objInteractor;
}
Subobject#objInteractor {
  class: 'ilog.cpl.interactor.IlpDefaultObjectInteractor';
}
```

The same configuration can be achieved through the API, as follows:

```
IlpTable table = // ...
IltNetworkElement ne = //...
IlpTableController tableController = table.getController();
// Create an object interactor
IlpObjectInteractor objectInteractor = new IlpDefaultObjectInteractor();
tableController.setObjectInteractor(ne, objectInteractor);
// Configuring the specific object interactor is similar to configuring
// a view interactor.
```

```
objectInteractor.setGestureAction(IlpGesture.BUTTON3_CLICKED, new MyAction())
;
```

How to associate an object interactor with a table cell

You can associate an object interactor with a table cell, which is identified by a representation object and an attribute, by using either CSS or the API. The following CSS extract shows how to customize a specific object interactor to the cell that represents the `name` attribute:

```
Table {
   interactor: true;
}
object."ilog.tgo.model.IltNetworkElement/name" {
   interactor: @+objInteractor;
}
Subobject#objInteractor {
   class: 'ilog.cpl.interactor.IlpDefaultObjectInteractor';
}
```

The same configuration can be achieved through the API, as follows:

```
IlpTable table = // ...
IltNetworkElement ne = //...
IlpAttribute attribute = IltNetworkElement.NameAttribute;
IlpTableController tableController = table.getController();
// Create an object interactor
IlpObjectInteractor objectInteractor = new IlpDefaultObjectInteractor();
tableController.setObjectInteractor( ne, attribute, objectInteractor);
```

Actions related to mouse and keyboard events can be customized in the same way as for the view interactor. A pop-up menu factory can also be defined in the same way as for the view interactor. Refer to *Interacting with the table view*.

For a detailed description of interactors and gestures, refer to *Interacting with the graphic components* .

Handling the selection

Selection in the table view is managed by an IlpTableSelectionModel. Different kinds of selection models can be set to the table by using the method setSelectionModel of the table view. However, only the IlpDefaultTableSelectionModel allows you to use the method setSelectionMode, as illustrated in the following code sample.

How to handle selection in the table

```
IlpTable tableComponent = new IlpTable();
// Set a selection mode to select multiple entire rows
tableComponent.setSelectionMode(IlpTableSelectionMode.
  MULTIPLE_OBJECTS_SELECTION);
```

The different selection modes, defined by the enumerated type IlpTableSelectionMode, are the following:

- EMPTY_SELECTION: nothing is selected when clicking the table.
- SINGLE_OBJECT_SELECTION: a single row can be selected in the table.
- MULTIPLE_OBJECTS_SELECTION: multiple rows can be selected (by using the traditional Shift and/or Control keys and dragging the mouse).
- SINGLE_ATTRIBUTE_SELECTION: a single column can be selected.
- MULTIPLE_ATTRIBUTES_SELECTION: multiple columns can be selected.
- SINGLE_CELL_SELECTION: a single cell can be selected.
- MULTIPLE_CELLS_SELECTION: multiple cells can be selected.

Note: The selection mode can also be customized through the selectionMode property. The default mode is MULTIPLE_OBJECTS_SELECTION.

The following methods can be used to select business objects:

- addSelectionObject(IlpObject object) adds a business object to the selection (the entire row is selected).
- removeSelectionObject(IlpObject object) removes a business object from the selection.
- getSelectedObjects() returns the collection of selected business objects.
- getSelectedObject() returns the first selected business object (the first in the collection of business objects.)
- isObjectSelected(IlpObject object) returns true if the given selected object is selected.

- `clearSelection()` deselects all the selected objets.
- `selectAll()` selects all the visible cells (this means that filtered objects and hidden columns are not selected.)

Similar methods exist in `IlpTableSelectionModel` to select individual columns and cells.

Fitting to Contents

The table can dynamically adjust the width of one or more of its columns in order to fit in the available space. The table supports the following resizing modes defined in the enumerated type `IlpTableResizeMode`:

- No adjustment: `AUTO_RESIZE_OFF`

  The width of the columns is never automatically adjusted. If there is more space than used by the columns, it remains empty. If there is less space than required, a horizontal scroll bar displays.

- Adjust last column: `AUTO_RESIZE_LAST_COLUMN`

  The last column of the table will be adjusted to allow the width of the table to fit the available space.

- Proportionally adjust all columns: `AUTO_RESIZE_ALL_COLUMNS`

  All columns will be proportionally shrunk or expanded to occupy the available space.

Note: Unlike what happens in the Swing `JTable` when the resizing mode is `AUTO_RESIZE_LAST_COLUMN` and when the entire table is resized, not all the columns are proportionally adjusted. Only the last column is adjusted.

The default resize mode is `AUTO_RESIZE_OFF`.

The following example applies the `AUTO_RESIZE_LAST_COLUMN` mode to `tableComponent`.

How to resize a table

```
IlpTable tableComponent = new IlpTable();
// Apply the policy
tableComponent.setAutoResizeMode
  (IlpTableResizeMode.AUTO_RESIZE_LAST_COLUMN) ;
// Now, when a column or the entire table is resized, only the last column
//is shrunk or expanded
```

Note: The resize mode can also be controlled through the `autoResizeMode` property.

Resizing columns

The user can dynamically resize the columns in a table by dragging the right border of their header.

To resize a column by programming, use the following code, setting a size of `10` to the `name` column.

How to resize columns in a table

```
IlpTable tableComponent = new IlpTable();
// This table will contain Network Elements
IlpClass acceptedClass = IltNetworkElement.GetIlpClass();
tableComponent.setAcceptedClass(acceptedClass);
...
// Retrieve the IlpAttribute corresponding to the name in
// IltNetworkElement class
IlpAttribute name = acceptedClass.getAttribute("name");
TableComponent.getTableColumn(name).setPreferredSize(10);
```

Note: The preferred width of a column can also be controlled through the `preferredWidth` property. For more information, see *Customizing column headers and rows*.

Fixing columns in a table

You can fix a set of columns on the left side of the table. To do so, call the method `setFixedColumnCount(int)` with the number of columns you want to fix as its parameter. The method has no effect if the value of its parameter is less than `0`. A value of `0` indicates that none of the columns in the table will be fixed. If the parameter is greater than the number of columns in the table, all the columns are fixed. Here is an example.

How to fix the position of columns in a table

```
IlpTable tableComponent = new IlpTable();
...
// Fix 3 columns
tableComponent.setFixedColumnCount(3) ;
// Fix a 4th column
tableComponent.setFixedColumnCount(4) ;
// Unfix all columns
tableComponent.setFixedColumnCount(0) ;
```

You can retrieve the number of fixed columns in the table by using the method `getFixedColumnCount()`, and know whether a given column is fixed by using the method `isColumnFixed(ilog.cpl.model.IlpAttribute)`.

The number of fixed columns can be configured using style sheets, as described in *Configuring the table component*.

Moving columns

The user can dynamically rearrange the columns in a table by dragging their header to a new location. This functionality can be disabled and enabled with the method `setReorderingAllowed(boolean)` of the class `IlpTable`.

To rearrange the columns by programming, use the following code.

How to rearrange columns in a table

```
IlpTable tableComponent = new IlpTable();
// This table will contain Network Elements
IlpClass acceptedClass = IltNetworkElement.GetIlpClass();
tableComponent.setAcceptedClass(acceptedClass);
...
// Retrieve the IlpAttribute corresponding to the name in
// IltNetworkElement class
IlpAttribute name = acceptedClass.getAttribute("name");
// Column "name" will take the second position
tableComponent.setColumnIndex(name,1);
```

The default order of columns for a business class can also be controlled through the `tableColumnOrder` property. For more information, see Customizing table column headers and rows.

Searching for a string in a table

You can search for a string in a table by using the searchValue method. Searching is performed on the values displayed in the cells (that is, with styles applied), not on the raw values stored by the representation objects. The searchValue method returns the coordinates of the first cell containing the string, starting from the specified cell.

If the startingcell parameter is null, the search starts from the first cell in the table.

The search is not case sensitive if the parameter caseMatching is false.

The search can be performed row by row (scanRowByRow parameter is true), or column by column (scanRowByRow parameter is false).

The method returns null if the search value cannot be found in the scope of the search.

How to search for a string in a table

```
IlpTable tableComponent = new IlpTable();
...
IlpCellCoordinates cellFound
   = tableComponent.searchValue("a string", // searched string
                                null,       // starting cell
                                false,      // scan row by row
                                false);     // case matching
```

Showing or hiding columns in a table

You can alternatively hide and show columns in a table by using the method `setColumnVisible`. This method takes as its parameters the `IlpAttribute` object of the affected column and a Boolean value setting the visibility to `true` or `false`. The attribute can be retrieved from the table model or from the table view using the method `getColumn (int)`.

How to show or hide columns in a table

The following example shows how to show and hide columns in a table.

```
IlpTable tableComponent = new IlpTable();
// This table will contain Network Elements
IlpClass acceptedClass = IltNetworkElement.GetIlpClass();
tableComponent.setAcceptedClass(acceptedClass);
...
// Retrieve the IlpAttribute corresponding to the name in
// IltNetworkElement class
IlpAttribute nameAttr = acceptedClass. getAttribute("name");
// Hide the column
tableComponent.setColumnVisible(nameAttr, false) ;
// Show the column again
tableComponent.setColumnVisible(nameAttr, true) ;
```

The method `isColumnVisible(ilog.cpl.model.IlpAttribute)` indicates whether the column specified by `attribute` is visible.

Note: The visibility of a column can also be controlled through the `visible` property. For more information, see *Customizing column headers and rows*.

Sorting columns

Columns can be sorted in ascending or descending order, interactively or by programming.

Interactively, the user clicks a column header once to sort it in ascending order, twice to sort it in descending order, and three times for no sorting at all. Using the Shift key while clicking the column header adds this column as a sorting criterion, whereas otherwise the column is set as the only sorting criterion.

By programming, you can use the following methods:

- ♦ `addSortingCriteria(ilog.cpl.model.IlpAttribute, int, boolean, boolean)`addSortingCriteria adds the given attribute as a sorting criterion. This method takes as its arguments:
  - the attribute corresponding to the column to be sorted
  - an `order` parameter that defines the position of the column among the sorting criteria
  - the `ascendingOrder` parameter, which, when set to `true`, sorts the column in ascending order
  - the parameter `useDisplayValue`, which indicates whether the sorting is applied to the display values (that is, with style sheets applied) or to the raw values

    Each sorted column serves as a sorting criterion. If `order` equals `1`, the column is selected as the first sorting criterion, if it equals `2`, it is selected as the second sorting criterion, and so on. If `order` is less than `1`, the column is considered as the first criterion. If `order` is greater than the current number of criteria, the column is considered as the next criterion.
- ♦ `getSortingOrder(ilog.cpl.model.IlpAttribute)` returns the position as a sorting criterion of the specified attribute.
- ♦ `isUsingAscendingOrder(ilog.cpl.model.IlpAttribute)` returns `true` if the specified attribute is sorted in ascending order.
- ♦ `isUsingDisplayValue(ilog.cpl.model.IlpAttribute)` returns `true` if the specified attribute is sorted by display values.
- ♦ `removeAllSortingCriteria()` removes all sorting criteria.
- ♦ `getSortedAttributesCount()` returns the number of sorted columns in the table.

Note: The sorting order can also be controlled through the "`sortingMode`" and "`sortingPriority`" properties. For more information, see *Customizing column headers and rows*.

Adding new columns to the table

It is possible to add custom attributes as new columns in a table component, as illustrated by the following code sample.

How to add columns to a table

```
// Create a datasource
IltDefaultDataSource dataSource = new IltDefaultDataSource();
// Read an XML file into the datasource
dataSource.parse("alarms.xml");
// Create a table component
IlpTable tableComponent = new IlpTable();
// Get the Alarm class
IlpClass alarmClass =
  IltSystem.GetDefaultContext().getClassManager().getClass("Alarm");
// Set the datasource to the component, and show instances
// of the Alarm class
tableComponent.setDataSource(dataSource, alarmClass);
// Add custom attributes
// Get the existing severity attribute
IlpAttribute severity = alarmClass.getAttribute("perceivedSeverity");
// Create a 'Short severity' attribute that represents the severity
// in a concise way
IlpAttribute shortSeverityAttribute =
  new IlpReferenceAttribute("shortSeverity", severity);
tableComponent.addAttribute(shortSeverityAttribute);
```

Filtering rows

You can filter the rows in a table by implementing the interface `IlpFilter` and setting it to the table.

To perform the filtering, you can use the method `setFilter` at two different levels:

At the adapter level

You have the choice to apply the filter to the table component or to the adapter, the result will be the same.

The following code shows how to create a filter displaying only `IltNetworkElement` instances with alarms.

How to set a filter to the table component

```
// Create a table component and set it to a data source
IlpTable tableComponent = new IlpTable();
IltDefaultDataSource dataSource = new IltDefaultDataSource();
// The second parameter specifies which kind of object contained
// in the data source the table will show
tableComponent.setDataSource(dataSource,
  IltNetworkElement.GetIlpClass());
IlpFilter alarmFilter = new IlpFilter() {
  public boolean accept(Object object) {
    IlpObject ilpObject = (IlpObject)object;
    Integer alarmCount =
      (Integer)ilpObject.getAttributeValue(IltObject.AlarmCountAttribute);
    return (alarmCount.intValue() > 0);
   }
};
tableComponent.setFilter(alarmFilter);
```

The following example shows how to set a filter to an adapter.

How to set a filter to the adapter

```
adapter.setFilter(new IlpFilter() {
  public boolean accept(Object object) {
   IlpObject bo = (IlpObject)object;
   return bo.getAttributeValue(boolAtt).equals(Boolean.TRUE);
  }
});
```

The argument `boolAtt` passed to the method `getAttributeValue(ilog.cpl.model.IlpAttribute)` is an instance of `IlpAttribute` that returns a Boolean value. Business objects will be transformed into representation objects only if this argument is `true`.

At the controller level.

Filtering takes places between the model and the view and is performed by the controller.

How to set a filter to the table controller

```
IlpTableController controller = table.getController();
controller.setFilter(new IlpFilter() {
  public boolean accept(Object object) {
    IlpObject bo = (IlpObject)object;
    return bo.getAttributeValue(boolAtt).equals(Boolean.TRUE);
  }
});
```

The `setFilter(ilog.cpl.util.IlpFilter)` method at the adapter level may seem redundant with the `setFilter` method at the controller level; however, the advantages and drawbacks are not the same. Modifying the filter associated with an adapter to which a number of representation objects have already been added causes a large number of objects to be created or destroyed. Whereas changing a filter set to the controller only alters intermediate data, without leading to object creation or destruction. Setting a filter to an adapter significantly improves the performance, since it avoids creating a great number of unused representation objects, and saves storage space in the memory of the model.

The most likely scenario for using the `setFilter(ilog.cpl.util.IlpFilter)` method at two different levels is the following: The `setFilter` method at the adapter level can be considered the first step in the filtering process in the sense that it reduces the number of objects of the accepted class that will be created and displayed. The next step consists in setting a filter at the controller level to further reduce the number of objects that will be actually displayed in the table component.

Excluding table rows

You can specify the business objects that will not be represented in the table component depending on their business classes. To do so, you need to specify the business classes to be excluded using method `setExcludedClasses` in the table component adapter. To retrieve the adapter, use the `getAdapter` method. The adapter must be an instance of a subclass of `IlpListAdapter`.

How to specify excluded classes in the table component

You can specify that business objects from specific business classes are not represented in the table component. You can do that using the API, `setExcludedClasses(java.util.List)` method, or using CSS.

The following example shows you how to prevent objects from business classes `IltAlarm` and `IltLed` to be represented:

```
Adapter {
  excludedClasses[0]: "ilog.tgo.model.IltAlarm";
  excludedClasses[1]: "ilog.tgo.model.IltLed";
}
```

Note: The filtering that is performed through the use of the excluded class list takes actually place at the adapter level.

To see how to configure excluded classes through CSS, refer to *The Adapter rule* .

Architecture of the table component

Like the other JViews TGO components, the table component is based on the MVC architecture, which means that it has a model, a view and a controller associated with it. For a general introduction to the MVC architecture, see *Architecture of graphic components* which describes the classes and features of the table component specific to each of the three modules of the MVC architecture, and also explains the role of the adapter.

In this section

Class overview

This topic describes the classes that you can use to create and manage tables. For a more detailed description, refer to the `overview-summary` IBM® ILOG® JViews TGO Java™ API Reference Documentation. The classes are organized as follows:

♦ *MVC (model, view, controller) architecture*

♦ *Representation model and representation objects*

♦ *Graphic view and renderers*

♦ *Controller and interactors*

MVC (model, view, controller) architecture

The MVC architecture for the table component is implemented by the following classes (see *Model, view, and controller for the table component*):

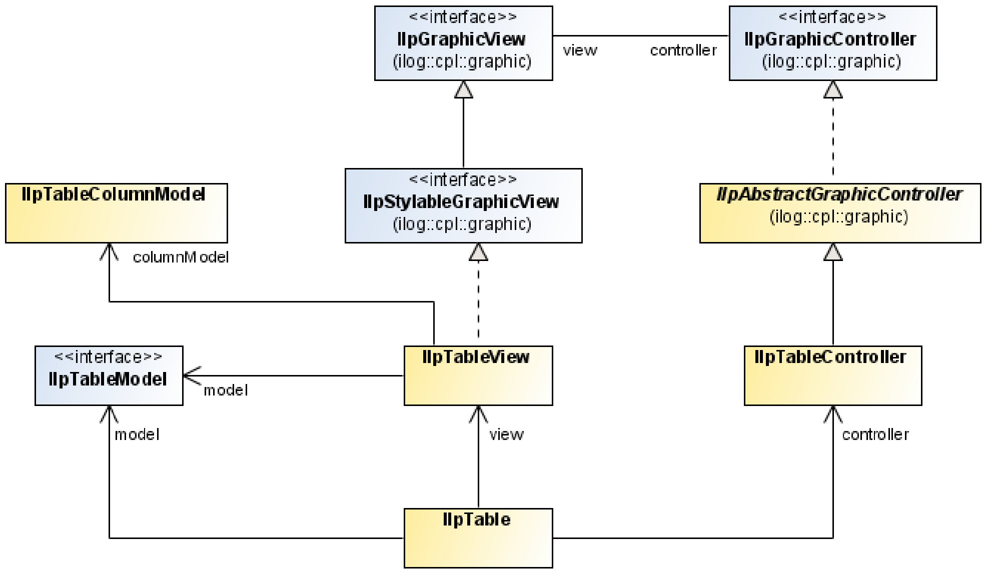
♦ The `IlpTableView` class is the view module of the MVC architecture. It defines a Swing-based table used to display representation objects. It uses an `IlpTableModel` as entry model.

♦ The `IlpTableModel` interface defines how the `IlpTableView` can access representation objects and attributes. Implementations of this interface relate to the model module of the MVC architecture.

♦ The `IlpTableColumnModel` class defines which columns of the table model are visible, which are fixed, as well as the order and the size of the columns.

♦ The `IlpTableController` class represents the controller module of the MVC architecture. This class is used to define the behavior of the `IlpTableView`. Its API contains methods to sort the table columns and to set interactors to the table.

♦ `IlpTable` is a convenient Bean that replicates the key API of the MVC components and can also be used inside an IDE. It creates an `IlpTableView`, an `IlpTableListModel`, and an `IlpTableController` and interconnects these objects.



Model, view, and controller for the table component

For general information about the model, the view, and the controller, see *Architecture of graphic components*.

Representation model and representation objects

The representation model for the table component is implemented by the following classes (see *Representation model and representation objects for the table component*):

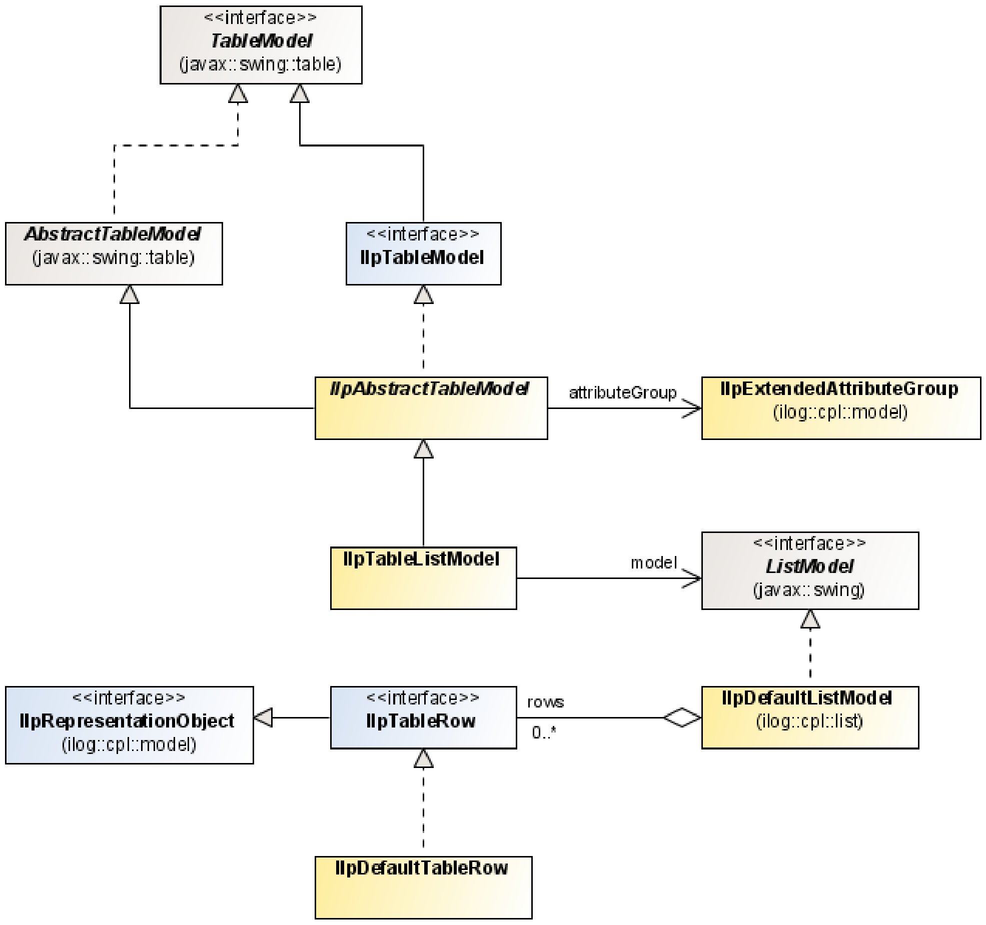
♦ The `IlpTableRow` interface defines the representation objects that can be displayed in the `IlpTableView`.

♦ The `IlpDefaultTableRow` class is the default implementation of `IlpTableRow`. It can use the attribute values of the representation object directly or take the attribute values of the corresponding business object.

♦ `IlpAbstractTableModel` is an abstract implementation of the `IlpTableModel` interface. It provides attributes to the `IlpTableView` based on an `IlpExtendedAttributeGroup`. It defines an arbitrary order of the attributes, which corresponds to the default order of the columns in the table.

♦ The `IlpDefaultListModel` class is provided to contain representation objects (`IlpTableRow` instances) stored as a list.

♦ The `IlpTableListModel` class extends `IlpAbstractTableModel`. It is designed to provide representation objects to the `IlpTableView`. The representation objects are stored as a list based on an `IlpDefaultListModel`. The `IlpDefaultListModel` can be connected to a back-end application by means of an adapter and a data source or be fed directly with representation objects.



Representation model and representation objects for the table component

For general information about the representation model and the representation objects, see *Architecture of graphic components*.

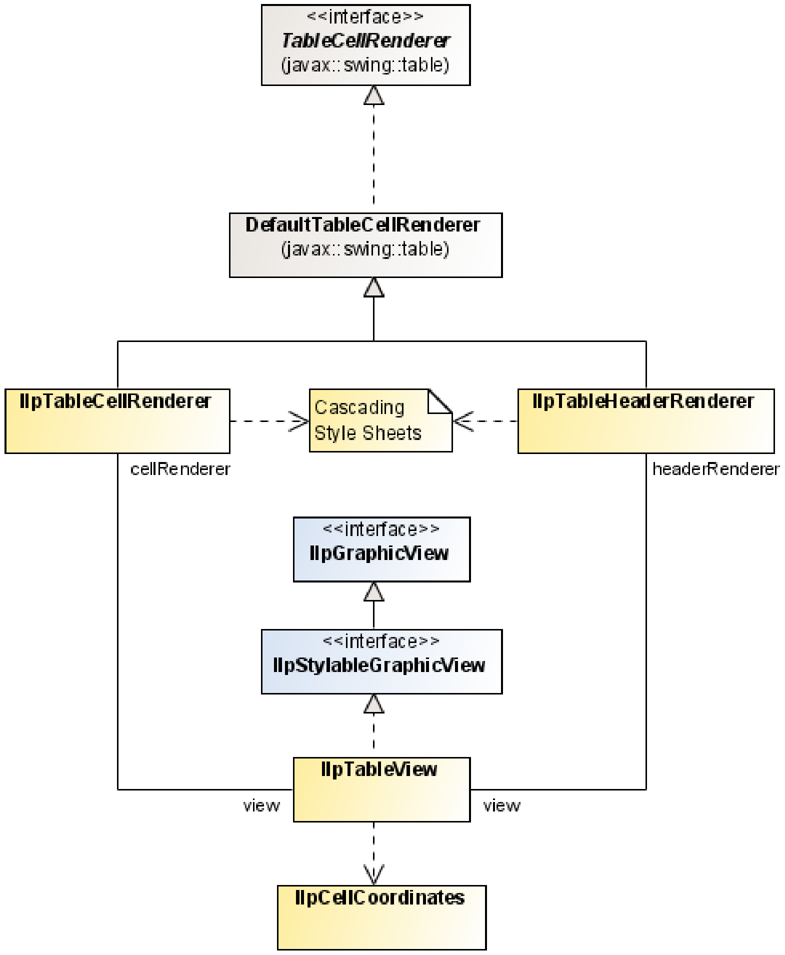
Graphic view and renderers

The graphic view and the renderers are implemented by the following classes (see *Graphic view and renderers for the table component*):

♦ The `IlpTableHeaderRenderer` is used to render the column headers of the table (view). In each column header, it displays an icon, indicating the sorting direction and sorting order of the columns, and uses the style sheet information set to the table to get the column label and description.

♦ The `IlpTableCellRenderer` is used to render the cells of the table. It uses the style sheet information set to the table to retrieve the graphic characteristics needed to display the cell contents, such as label, icon, font, or color.

- `IlpCellCoordinates` locates a cell in an `IlpTableView`. It is defined by a row index and a column index.



Graphic view and renderers for the table component

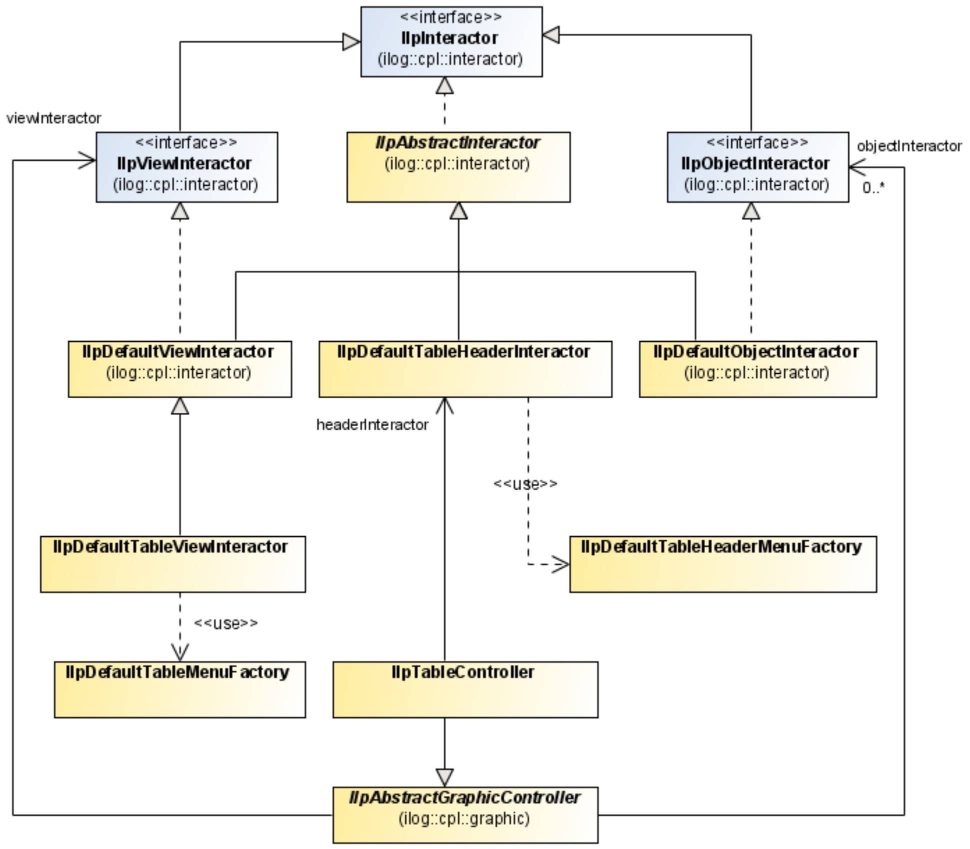
For general information about the graphic view and the rendering, see *Architecture of graphic components*.

Controller and interactors

The controller and interactors of the table component are implemented by the following classes:

- When set to the `IlpTableController`, an `IlpDefaultTableViewInteractor` object assigns default behavior to the table view, delegating events that occur in the table to an `IlpDefaultTableHeaderInteractor` or to an `IlpObjectInteractor`, according to the location of the events.

♦ `IlpDefaultTableMenuFactory` provides a default pop-up menu to be displayed on a table header.



Controller and interactors for the table component

For general information about the controller and interactors, see *Architecture of graphic components*.

The model

The table model provides the representation objects to be displayed in the associated table. It can be connected to a back-end application by means of an adapter. It refers to an attribute group `IlpExtendedAttributeGroup` to determine which attributes of the representation objects are to be displayed. Each time an attribute is added to or removed from this attribute group, a column is added to or removed from the table. (For more information on attributes, see Attribute API in the *Business Objects and Data Sources* documentation.)

Note: Each added column is arbitrarily placed at the end of the table, unless its position is defined as property `index` in the style sheets set to the table.

By default, the `IlpTableView` creates an instance of `IlpTableListModel`. This table model could be filled through an `IlpDefaultListModel` connected to a back end or could be used directly.

When the table is to be connected to a back-end application, a data source and a list adapter must be instantiated. Both the data source and the default list model must be connected to the adapter.

Managing attributes (columns)

Attributes contain the values of the representation objects, or of the business objects related to the representation objects, to be shown in the table. Attributes are displayed in the columns of the table.

Note: An attribute can also contain a computed value, that is, the result of a calculation, or refer to another attribute. See Introducing business objects and data sources.

The `IlpAbstractTableModel` class provides the following methods to manage these attributes:

♦ `setAttributeGroup(ilog.cpl.model.IlpExtendedAttributeGroup)` sets the attribute group used by the table model to determine which attributes are to be shown. A call to this method does not empty the table model (which keeps its representation objects.)

  Note: The representation objects and the table model do not necessarily use the same attribute group: the table model can provide computed attributes with values that are not in the representation objects.

  After a nonordered attribute group has been set to the table model, the attributes are alphabetically sorted in the table model. This constitutes the default order of the columns in the table. Each attribute added after the attribute group has been set to the table model is placed at the end of the table. This arbitrary order is modified when the property tableColumnOrder is defined for the accepted business class or when the property index is defined for the attribute represented in the column.

- getAttributeGroup() returns the attribute group used by the table model.
- getColumns() returns the attribute at the specified index.
- getColumnCount() returns the number of columns (attributes) contained in the table model.
- getColumnName(int)returns the name of the column, the index of which is given as parameter. The column name is the same as the attribute name.

Managing representation objects (rows)

The IlpTableListModel class, which extends IlpAbstractTableModel, provides the following methods to manage the representation objects in a table. The representation objects correspond to the rows in the table. These rows must be instances of an implementation of IlpTableRow.

- setModel(javax.swing.ListModel) sets the IlpDefaultListModel used by the table model to store the representation objects.
- getModel() returns the IlpDefaultListModel used by the table model.
- getRow(int) returns the representation object (IlpTableRow) at the specified index.
- getRowCount() returns the number of representation objects in the table model.

To add or remove representation objects when you do not use an adapter and a data source, you must use the following IlpDefaultListModel methods:

- add(java.lang.Object) adds a representation object, o, to the table model (o must be an IlpTableRow).
- addAll(java.util.Collection) adds a set of representation objects, c, to the table model (c must contain only IlpTableRow instances).
- remove(int) removes the representation object whose index is given as parameter.
- remove(java.lang.Object) removes the specified representation object from the table model.
- clear() removes all the representation objects from the table model.

The view

The table view is responsible for displaying the graphic objects (table cells) and as such corresponds to the visible part of the architecture.

You can customize the representation of table cells by:

♦ *Using your own graphic representation*

♦ *Using an arbitrary TableCellRenderer*

These ways are provided in addition to the CSS-based customization of the default table cell renderer as explained in Customizing table cells in the *Styling* documentation.

Using your own graphic representation

You can create an `IlvGraphic` or a `JComponent` directly by declaring the `class` property and setting it in a CSS file.

The `class` property name is a reserved keyword that indicates the class name of the generated graphic object. JViews TGO provides a predefined representation for the objects in all graphic components, which means that the `class` property is optional. It can be used when you want to replace the predefined representation.

```
object."ilog.tgo.model.IltNetworkElement/label" {
    class: ilog.views.sdm.graphic.IlvGeneralNode;
    foreground: red;
    label: @label;
}
```

`IlpTableCellRenderer` takes the information present in the CSS files to define how a table cell is rendered. If the property `class` is specified, it defines the `IlvGraphic` or `JComponent` that will be used to render the table cell.

To use a cascading style sheet `class` property in a table view, do the following:

1. Implement your own graphic object or use an existing one, either as an `IlvGraphic` or as a `JComponent`.
2. Configure the properties of the objects or classes you want to represent with this property using CSS.
3. Load this CSS file in the table component as illustrated in *How to load a CSS file in a table component*.

Using an arbitrary TableCellRenderer

The renderer used by an `IlpTableView` is an `IlpTableCellRenderer`. You can replace it with you own renderer, using the `setDefaultRenderer(ilog.cpl.table.IlpTableCellRenderer)` method. This renderer must implement the `TableCellRenderer` interface.

To write your own renderer, do the following:

1. Implement the `javax.swing.table.TableCellRenderer` interface.
2. Configure your table component to use your renderer.

   The following code extract shows how to configure an `IlpTableView` to use the `MyTableCellRenderer` class to render all table cells:

   ```
   IlpTableView view = //access the table view
   view.setDefaultRenderer(new MyTableCellRenderer());
   ```

The controller

The `IlpTableController` class represents the controller module of the MVC architecture. It can be attached to a table view by using the following code.

How to attach a controller to the table view

```
IlpTableView tableView = new IlpTableView();
IlpTableController tableController = new IlpTableController();
tableView.setController(tableController);
```

Note that a controller is automatically attached to a table view when a table component is instantiated:

```
IlpTable tableComponent = new IlpTable();
IlpTableController controller = tableComponent.getController();
// tableComponent.getView().getController() == controller
```

When a controller is attached to a table view, the controller listens to the keyboard, mouse, and focus events that occur in the table and transfers them to the view interactor set to the controller.

Only one controller can be attached to a view at a time.

By default, the `IlpTableController` delegates event management to an `IlpDefaultTableViewInteractor`. However, you can specify another interactor by using the following code:

```
tableController.setViewInteractor(new MyInteractor());
```

For more details, see *Interacting with the graphic components*.

When an interactor receives mouse events from the controller, it tries to recognize the associated user gestures. For example, when it receives an event `MOUSE_PRESSED` followed by an event `MOUSE_RELEASED`, the interactor recognizes the gesture `BUTTON1_CLICKED`. The set of basic gestures recognized by an interactor is defined in the class `IlpGesture`.

Interactors allow you to associate behavior with user gestures by means of Java™ actions. You can also associate actions with keyboard events.

The `IlpDefaultViewInteractor` manages the events received from the controller in the following way:

- ♦ If the event is a keyboard event, it checks whether an action has been associated with this key. If so, it triggers the action.
- ♦ If the event occurred in the header, it delegates the event management to the `IlpDefaultTableHeaderInteractor` set to the controller, if any.
- ♦ If the event occurred on an `IlpGraphic` object or on a representation object, it delegates the event management to the `IlpObjectInteractor` set to the controller, if any.

- It checks whether the event corresponds to the display of a pop-up menu. If so, and if a pop-up menu is set to this interactor, it will display the pop-up menu and stop.
- It tries to recognize gestures from the event. When a gesture is recognized, it triggers the action associated with this gesture, if any.

The API of the `IlpTableController` also contains methods to sort the columns and filter the rows of the table view. See *Table component services*.

The adapter

The table adapter converts business objects retrieved from the associated data source to table rows. The table adapter is defined by the interface `IlpListAdapter`. This adapter does not create representation objects in a table model but in a list model. This list model and the representation objects it contains are transformed into a table model at the level of the table component.

The default table adapter implementation provides the following services:

♦ Creating the table rows. For details, see *Representation object factory*.

♦ Filtering the table rows. For details, see *Filtering rows*.

♦ Excluding table rows. For details, see *Excluding table rows*

The following figure shows the table adapter classes:

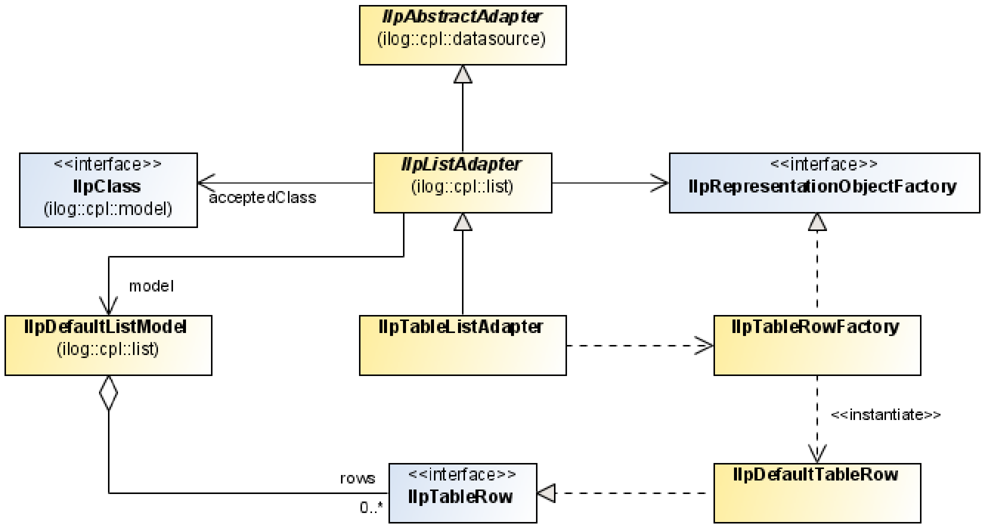


Table adapter classes

A given `IlpListAdapter` instance can only represent `IlpClass` objects of the same type. So that only objects of a specific class or of one of its subclasses be represented, you should set this class with the `setAcceptedClass` method.

The following code plugs an `IltDefaultDataSource` to an `IlpTableModel` through an `IlpTableListAdapter`.

How to create a table adapter by instantiating a table component

```
IlpTable ilpTable = new IlpTable();
```

```
IltDefaultDataSource dataSource = new IltDefaultDataSource();
ilpTable.setDataSource(dataSource, acceptedClass);
```

How to retrieve a table adapter

```
IlpTableListAdapter adapter = ilpTable.getAdapter();
```

How to set your own adapter to the table component

```
IlpTable ilpTable= new IlpTable();
IlpTableListAdapter adapter = new IlpTableListAdapter();
ilpTable.setAdapter(adapter);
ilpTable.setAcceptedClass(acceptedClass);
IltDefaultDataSource dataSource = new IltDefaultDataSource();
ilpTable.setDataSource(dataSource);
```

List adapters use by default an `IlpTableRowFactory` that creates representation objects of type `IlpDefaultTableRow`.

You can create a table adapter implicitly by instantiating the `IlpTable` component, as shown in the following example.

Representation object factory

The table adapter converts business objects retrieved from the associated data source to table rows. The new representation objects are created by a representation object factory. The factory interface varies according to the type of adapter. The table adapter uses by default an `IlpTableRowFactory` that creates representation objects of type `IlpDefaultTableRow`.

Editing

Adapter interfaces are read-only, meaning that they do not perform editing operations on the representation objects they create.

Architecture of graphic components

The IBM® ILOG® JViews TGO graphic components all implement the generic MVC (Model-View-Controller) architecture, an object-oriented pattern used in GUI design to clearly separate application objects and data from their graphic display and from the way the end user interacts with them. By separating application data from the logic for displaying and controlling this data, MVC provides a way to develop GUI systems that support multiple presentations of the same information. This section details the main components involved in the MVC architecture and explains the role of the adapter inside this architecture.

In this section

The MVC architecture: an overview

In the MVC paradigm, the end user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three kinds of object:

The model
: which manages the behavior and data of the application domain, responds to requests for information about its state (coming usually from the view), and responds to instructions to change its state (usually issued by the controller).

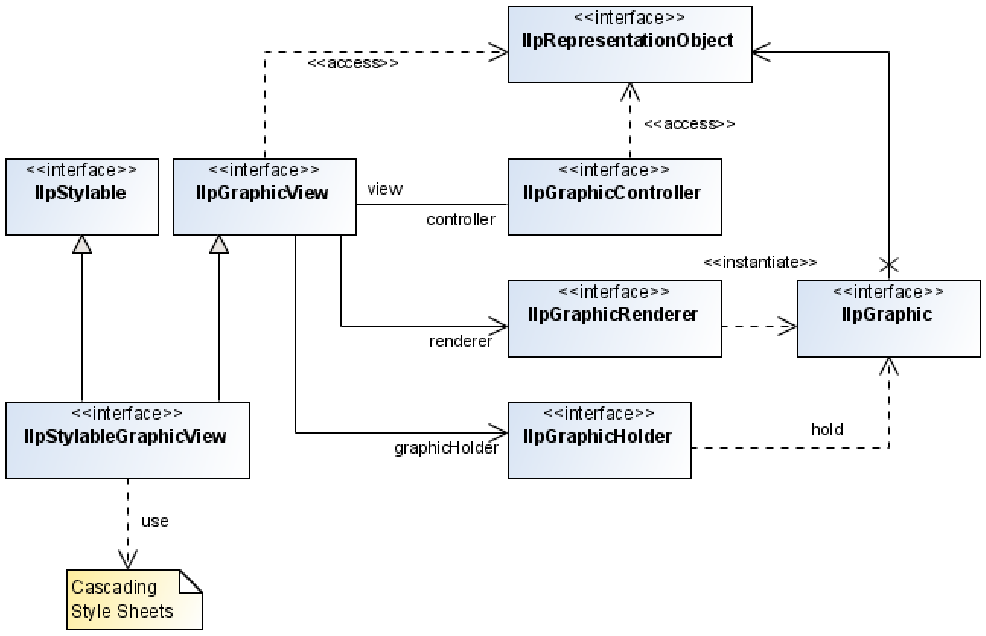
The view
: which manages the visual display of the data represented by the model.

The controller
: which translates interactions with the view, such as mouse clicks and key strokes, into updates to be performed on the model, instructing the model or the view or both to change as appropriate.

The MVC paradigm decouples the views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance reflects the state of the model, and a model must ensure that whenever its content undergoes modifications, all the connected views are updated accordingly. With this approach, multiple views can be connected to the same set of information while providing different graphic representations, which may be dependent on the view configuration.

Unlike the model, which may be loosely connected to multiple view-controller pairs, each view is associated with a unique controller and vice versa. Therefore, although the model is limited to sending notifications about changes in its structure or content using the subscribe/notify protocol, both the view and the controller can send messages directly to each other and to their model.

In JViews TGO, the MVC architecture is defined by the set of interfaces shown in the following figure.



Classes implementing the MVC architecture in IBM® ILOG® JTGO

The representation model

Describes the representation model of the MVC paradigm.

In this section

Overview

According to the MVC paradigm, the model represents the application data. In JViews TGO, the model module, referred to as the *representation model*, is a container for representation objects to which it provides access. See *Representation objects*.

Whenever possible, JViews TGO representation models are derived from existing Swing models and like them use listeners to notify attached views of any changes in the data. Like Swing models, JViews TGO representation models use the JavaBeans™ Event model to implement the notification process. In JViews TGO, notifications are stateful, which means that they not only inform that a modification has occurred, but also indicate what is the nature of that modification.

Also, like Java™ Swing models, representation models impose a specific structure on the objects they hold. Therefore, JViews TGO provides a different representation model for each type of data structure to be addressed and hence for each type of graphic component displaying this data.

JViews TGO provides the following representation model interfaces:

♦ A network model defined by the class `IlpNetworkModel`.

♦ An equipment model defined by the class `IlpEquipmentModel`.

♦ A tree model defined by the class `IlpTreeModel`.

♦ A table model defined by the class `IlpTableModel`.

For more information about these predefined representation models, see the information on each graphic component.

Representation objects

Representation objects constitute the basic elements of a specific graphic component model. Therefore, they cannot be shared across graphic components. More specifically, these objects map business objects to the specific object type by which they will be represented in the related component. For example, a business object to be displayed in a table is stored in the table representation model as a row representation object. A business object to be displayed in a network view is stored in the network representation model as a network node or link.

Representation objects contain sufficient data to be graphically represented inside a graphic component, but are independent of a particular graphic rendering. The graphic component, where the representation object is inserted, translates its attribute information into graphic objects using a renderer. You can associate graphic settings either with a representation object class or with a specific instance of that class in order to customize the rendering process. For further information, see Introducing cascading style sheets.

Representation objects are linked to business objects. For more information about business objects, see Introducing business objects and data sources. Representation objects are defined by the interface `IlpRepresentationObject`, which provides methods to:

♦ Link the representation object to a business object.

♦ Retrieve its attribute group.

♦ Get and set attribute values.

♦ Notify listeners about modifications to the representation object.

Like business objects, representation objects include an attribute group that can be either static or dynamic. For more information, see Attribute group .

If you instantiate a representation object directly, you have to create a specific attribute group to define the list of attributes attached to that object. When a representation object is linked to a business object, it contains all the attributes defined for the associated business object plus any other additional attributes that you would like to include, for example, a computed attribute calculated from other attributes present in the business object.

A dynamic attribute group does not necessarily have to be based on that of the corresponding business objects. To create such an attribute group, you have the following choices:

♦ Skip the business object.

  In basic applications, the user can directly instantiate representation objects. For example, when almost all data is displayed in only one graphic component or when the back-end application data is such that sharing the business model data across components is difficult.

♦ Use the business object directly.

  The attribute group of the representation object is taken from that of the corresponding business object.

♦ Extend the business object.

  The representation object may add extra attributes or hide some of the attributes present in the connected business object. Certain attribute values can be stored locally in the representation object.

- Define a custom attribute group.

  The attribute group for the representation object is completely different from that of the business object. Individual attributes may obtain their value directly or indirectly from the business object attributes, but most of the attributes are not the same.

Predefined representation object classes

JViews TGO includes predefined representation object classes for each one of the graphic components:

- `IlpNetworkNode` represents a node in a network.
- `IlpNetworkLink` represents a link connecting two nodes in a network.
- `IlpEquipmentNode` represents a piece of equipment in an equipment view.
- `IlpEquipmentLink` represents a link connecting two nodes in an equipment.
- `IlpTreeNode` represents a node in a tree.
- `IlpTableRow` represents a row in a table.

For more information about these classes, see the information on each graphic component.

The graphic view

Describes the graphic view, a container for graphic objects to which it provides access.

In this section

Introduction

The graphic view is a container for graphic objects to which it provides access. It is the physical space on the screen where these objects are drawn. In addition to storing the graphic objects created to represent the application data, the graphic view contains configuration and rendering information that is used to translate representation objects into graphic objects.

A graphic view is defined by the `IlpGraphicView` interface. This interface provides methods to:

- ♦ Set and retrieve the graphic holder. See *Graphic holders*.
- ♦ Set and retrieve the graphic view configuration. See *Graphic view configuration*.
- ♦ Set and retrieve the controller. See *The controller*.

Graphic objects

Business objects and their representation objects are graphically represented on the screen by means of graphic objects. Graphic objects make it possible for you to create sophisticated, high-quality graphic representations in a flexible way.

Graphic objects are always associated with a representation object and optionally with a representation object attribute. They are created for a specific graphic view and according to the configuration defined for that view.

Graphic object classes can be associated with default graphic settings that can be overwritten at the level of the graphic object instance. These graphic settings are stored as properties which are configured using cascading style sheets. For detailed information, see Introducing cascading style sheets .

The graphic view renderer is responsible for creating the final graphic representation for an object in a certain graphic view. See *Graphic view configuration*.

The preferred way to create and customize your graphic objects is using cascading style sheets (CSS). Using CSS, you can define `IlvGraphic` or `JComponent` objects to draw your representation objects according to the business model information. You can create simple representations such as labels, buttons and icons; or you can create sophisticated graphic objects that are composed of several objects using composite graphics. (See `IlvComponentGraphic` for more information.)

IBM® ILOG® JViews provides a set of convenience graphic objects that you can use directly when defining complex graphic representations:

♦ Simple graphic objects: label, icon

♦ Composite graphic objects: different balloons, stackers

Graphic object classes

JViews TGO provides all the necessary infrastructure for you to create applications that integrate both predefined business objects and custom business objects.

So that these business objects can be represented virtually in any type of graphic component (whether network, equipment, table, or tree), they are translated into representation objects specific to the graphic component in which they are to be displayed and then converted to graphic objects, ready to draw on the screen.

These graphic objects are defined by the interface `IlpGraphic` and can be either simple objects or composite ones. Simple graphics, also known as leaf graphics, are made up of a single graphic element, such as a label or an icon. Composite graphics consist of multiple graphic objects, which are grouped together by means of attachment rules.

JViews TGO supplies the following classes to help you create graphic objects and customize them:

♦ `IlpGraphic` is an interface that defines all the graphic objects.

♦ `IlpAbstractGraphic` is a basic implementation for graphic objects. This abstract class contains the basic information that allows you to create your own graphic object classes, if needed.

- `IlpLeafGraphic` is a concrete graphic object implementation that is used to create simple graphic objects, such as labels or icons.

Graphic holders

A graphic holder is defined by the interface `IlpGraphicHolder`. It is responsible for storing the graphic objects created for a given graphic view. Note that in simple cases where graphic objects can be recreated when needed, it may not be necessary to store them in a graphic holder.

When creating a graphic view, you can use one of the following convenience implementations of `IlpGraphicHolder`.

♦ `IlpDefaultGraphicHolder`. This default implementation stores graphic objects per representation object and attribute.

♦ `IlpEmptyGraphicHolder`. This simple implementation is used when graphic object instances do not need to be stored in the graphic view.

Graphic view configuration

JViews TGO provides support to customize the graphic representation of the objects in the different graphic components, and of the graphic components themselves by means of cascading style sheets (CSS). For details, refer to:

♦ *Configuring a network component through a CSS file*

♦ *Configuring an equipment component through CSS*

♦ *Configuring the tree component through a CSS file*

♦ *Configuring the table component through a CSS file*

You can load the CSS file in the graphic view using the method `setStyleSheets`. The CSS is dynamically interpreted, which means that there is no need to rerun the application after changes. Once your style sheets are set, all objects currently visible in the graphic view are redecorated accordingly. For details, see Introducing cascading style sheets.

The controller

Describes the controller in the MVC architecture, which is responsible for handling user input and modifying the model accordingly.

In this section

Introduction

In the MVC architecture, the controller is responsible for handling the user input and modifying the model accordingly. The controller makes sure that any changes in the model or in the graphic view configuration are reflected appropriately to the user. The controller is defined by the interface `IlpGraphicController`. This interface provides methods to:

- ♦ Retrieve the application context
- ♦ Retrieve the graphic view being controlled
- ♦ Retrieve the view interactor
- ♦ Retrieve interactors for specific representation objects and attributes

The controller has various roles:

- ♦ It manages *view interactors*, which operate on the view as a whole, and *object interactors*. For more information, see *Interacting with the graphic components*.
- ♦ It creates the graphic view configuration and sets it to the graphic view. The graphic view configuration is created based on the application context.
- ♦ It makes it possible for external components to register for events triggered when the user interacts with the graphic components or when the model is modified.
- ♦ The controller also provides filtering and sorting capabilities. It can insert a filter between the model and the view to display only representation objects that satisfy specified conditions and can set a sorter that defines the order in which representation objects are to appear within the view.

Note: Sorting is available only for the table and the tree component.

JViews TGO supplies default implementations of controllers for each one of the predefined graphic components. These are:

- ♦ `IlpTableController`
- ♦ `IlpTreeController`
- ♦ `IlpNetworkController`
- ♦ `IlpEquipmentController`

For more information, see the information on each graphic component.

Interacting with the graphic components

The controller manages interactors. Interactors translate user gestures—a series of one or more mouse events that are executed by the user to perform a single task—and keystrokes into actions to be performed, and more specifically into Swing action invocations. JViews TGO provides two types of interactor:

♦ View interactors that operate on the view as a whole and

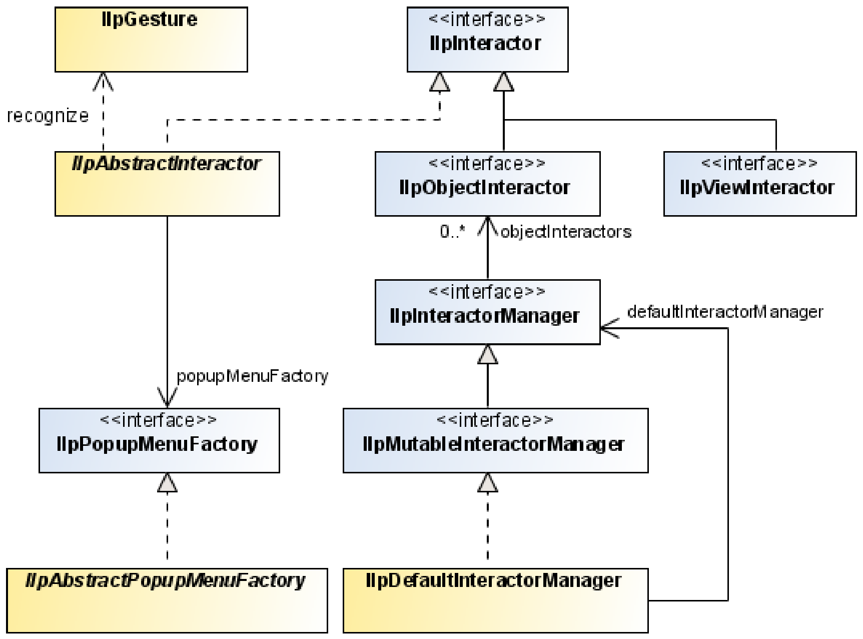
♦ Object interactors that apply to each object contained in the view individually.

Events associated with interactors can be customized, except for some gestures, such as selection management, tooltips, and display of pop-up menus, which have standard behavior in most user interfaces. All interactors recognize a number of basic user gestures, such as pressing a mouse button, while only certain component-specific interactors handle more complex gestures, such as moving a node or a table column or creating a link. In addition, all interactors include support for displaying a pop-up menu. For information about component-specific interactions, see the information on each graphic component.

You can define global object interactors (that apply to the entire application) using the `IlpInteractorManager` interface. A default implementation of this interface, `IlpDefaultInteractorManager`, is provided, which you can use globally and access through the default context. For information about the default context, see The application context.

By default, the controller is associated with a default interactor manager. If interactors specific to given representation objects have been defined, these will hide the global interactors. In this case, when an interactor is requested, it is first searched for among the local interactors and then, if none is found, in the default manager interactor.

The figure below shows the basic classes and interfaces that implement interactor support.


IlpGesture
<<interface>> IlpInteractor
recognize
IlpAbstractInteractor
<<interface>> IlpObjectInteractor
<<interface>> IlpViewInteractor
0..*
objectInteractors
<<interface>> IlpInteractorManager
defaultInteractorManager
popupMenuFactory
<<interface>> IlpPopupMenuFactory
<<interface>> IlpMutableInteractorManager
IlpAbstractPopupMenuFactory
IlpDefaultInteractorManager

Interactor classes and interfaces

- The class `IlpGesture` is an enumeration defining the basic user gestures that are recognized by all the interactors, whatever the component they are attached to.

- The interface `IlpInteractor` contains methods to attach keystrokes or user gestures to actions and to set a pop-up menu factory to the interactor.

- The class `IlpInteractionContext` contains information about the view to which the user events apply, the recognized gestures that are incomplete, the complete gestures, if any, and the position where this gesture takes place.

- The class `IlpAbstractPopupMenuFactory` provides pop-up menus for an interactor given the specified interaction context.

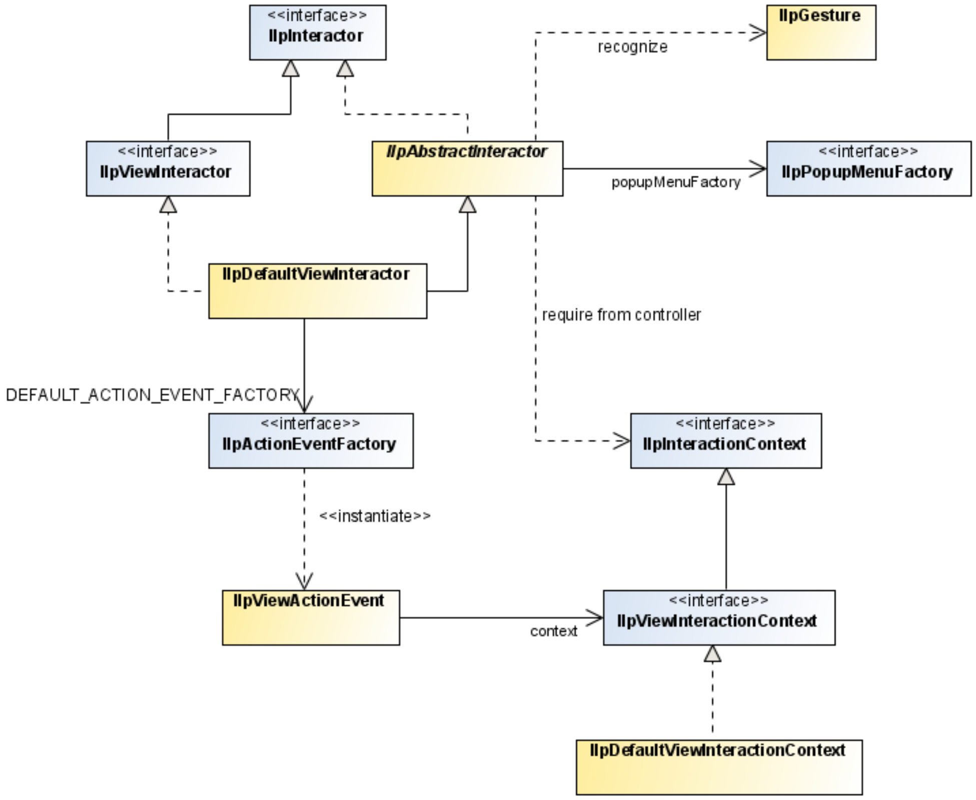
- The class `IlpAbstractInteractor` is an abstract implementation of `IlpInteractor`. This class provides an implementation for the method `processEvent(ilog.cpl.interactor.IlpInteractionContext, java.awt.AWTEvent)` that recognizes the basic gestures defined in `IlpGesture`. The only method that subclasses should implement is `createEventAction`. This method must provide an `ActionEvent` instance that is passed to the action associated with the recognized gesture.

View interactors

View interactors are defined by the interface `IlpViewInteractor`. They handle interactions with a graphic view and are attached to the controller associated with that view. When a view interactor is attached to the controller, the controller registers all the user events happening on that view. It retrieves the view interaction context or creates one, if none has been defined, and calls the `IlpViewInteractor.processEvent` method to process the events.

If an event applies to a graphic object, the view interactor will first check whether this object is associated with an object interactor and, if so, will delegate processing to that interactor. See *Object interactors*. The view interaction context, defined by the class `IlpViewInteractionContext`, extends `IlpInteractionContext` to add the appropriate graphic objects and an `IlpObjectInteractionContext` that is passed to object interactors.

The diagram below shows the interfaces and classes related to view interactors.



View interactors interfaces and classes

A default implementation is provided for the view interactor, which is defined by the class `IlpDefaultViewInteractor`. This class defines standard behavior for all its associated graphic views. The default view interactor provides an `IlpViewActionEvent` that gives access to information contained in the view interactor context to the action that is called when a user gesture is recognized.

Selection handling in pop-up menus

The selection behavior attached to pop-up menus is defined by the method `manageSelection (ilog.cpl.interactor.IlpInteractionContext, ilog.cpl.util.selection. IlpRepresentationObjectSelectionModel, ilog.cpl.model.IlpRepresentationObject)` and is as follows:

- When the mouse is over an object and this object is not selected, this object is selected and other objects are deselected. If the object is selected, the selection remains unchanged.
- When the mouse is not over an object, all the objects in the view are deselected.

To modify this behavior, you must redefine the method `manageSelection(ilog.cpl.interactor.IlpInteractionContext, ilog.cpl.util.selection.IlpRepresentationObjectSelectionModel, ilog.cpl.model.IlpRepresentationObject)`.

Tooltip support

In JViews TGO, tooltip support is based on the Swing tooltip support and is handled by the graphic view. It allows you to create tooltips for graphic objects that represent entire objects or attributes and can be customized by means of cascading style sheets. You can define tooltips as simple text or as complex graphic objects.

The following properties are available to configure tooltips:

- `tooltipText` defines a string value to be used as the object tooltip.
- `tooltipGraphic` defines a graphic object that is used to create a tooltip. This property has priority over `tooltipText`.

The following code shows how to add tooltip support to a given graphic view.

How to add tooltip support to the view

```
IlpNetwork network = new IlpNetwork();
IlpGraphicView view = network.getView();
IlpToolTipManager.AddToolTipSupport (view);
```

Once you have added tooltip support to a graphic view, you can configure its behavior with the `IlpToolTipManager` API.

How to configure tooltip support

```
IlpToolTipManager tmgr = IlpToolTipManager.GetToolTipManager(view);
```

How to remove tooltip support from the view

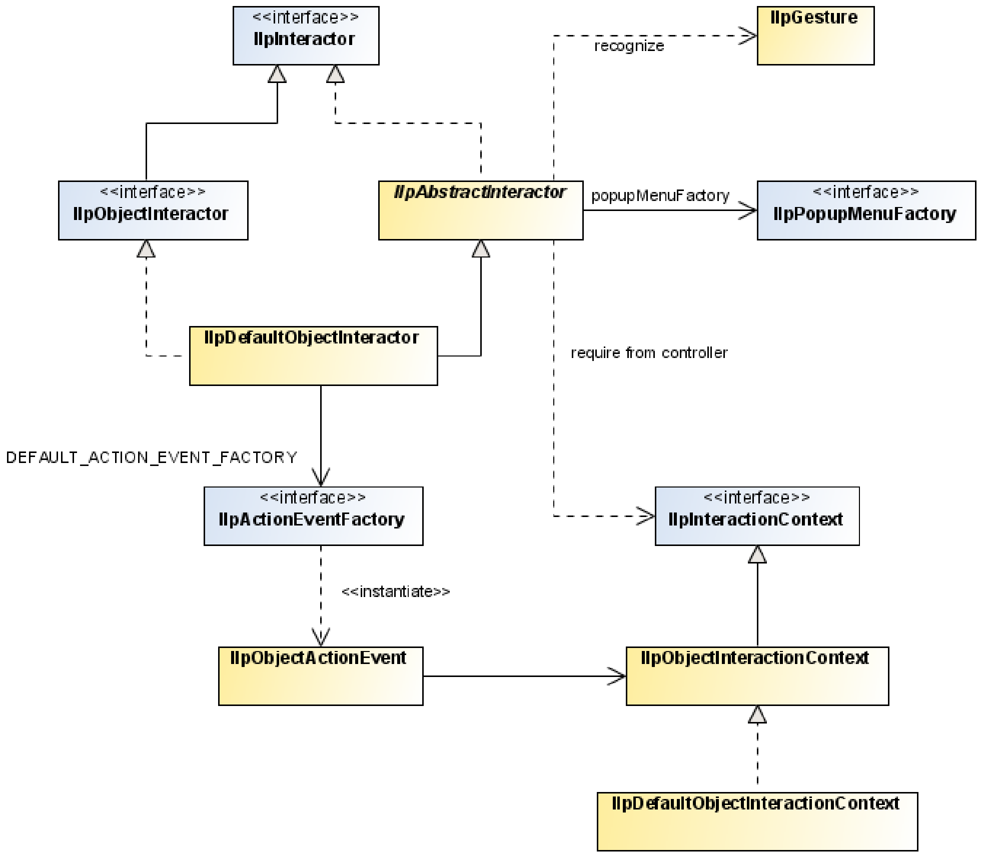
Tooltip support can be removed from a graphic view with the following method.

```
IlpToolTipManager.RemoveToolTipSupport (view);
```

Object interactors

Object interactors are defined by the interface `IlpObjectInteractor`. They handle interactions with simple or composite `IlpGraphic` objects or representation objects. If a user event applies to a simple graphic object that is part of a composite graphic object and that object has no associated interactor, then the event will be passed to and processed by the interactor of the parent object, if any.

The diagram below shows the interfaces and classes related to object interactors.



Object interactors interfaces and classes

`IlpObjectInteractionContext` extends `IlpInteractionContext` to add the concerned graphic object or representation object.

A default implementation is provided for the object interactor, which is defined by the class `IlpDefaultObjectInteractor`. The default object interactor provides an `IlpObjectActionEvent` that gives access to information contained in the object interactor context to the action that is called when a user gesture is recognized.

The adapter

The adapter is an object that acts like a bridge between a data source and a graphic component. It converts the business objects of the data source (`IlpObject` instances), which are component-independent, to representation objects which are suitable for a given representation model, that is, for a given graphic component. For information about the representation model and representation objects, see *The representation model*.

The use of adapters is optional as you can directly instantiate and insert representation objects into a representation model. However, they are highly recommended as they hide the complexity of creating representation objects relationships. Using adapters, you do not need to care about representation objects. All you have to do is create the business model in the data source and customize the view and objects using CSS to obtain an attractive graphic representation.

By default, adapters are meant to work in conjunction with a data source. By default, they are used in all JViews TGO graphic components.

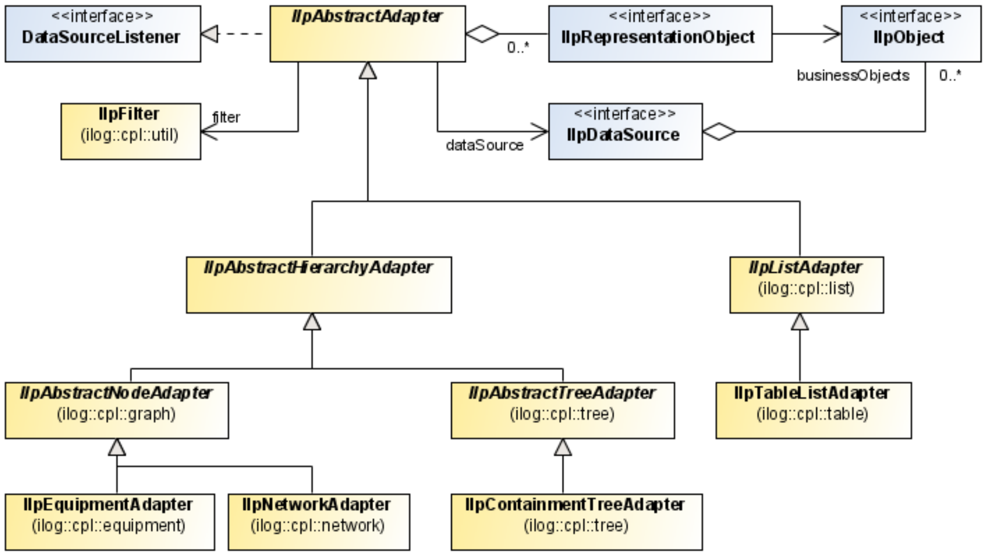
Adapters also handle synchronization.

JViews TGO supplies four kinds of adapter, each corresponding to a specific type of representation model:

- ♦ The table adapter creates representation objects of type table row. See section *Table component* for the specific features of the table adapter.
- ♦ The tree adapter creates representation objects of type tree node. See section *Tree component* for the specific features of the tree adapter.
- ♦ The network adapter creates representation objects of type network node and link. See section *Network component* for the specific features of the network adapter.
- ♦ The equipment adapter creates representation objects of type equipment node. See section *Equipment component* for the specific features of the equipment adapter.

These adapters are all based on an abstract adapter, defined by the interface `IlpAbstractAdapter`.

The following figure shows interfaces that are directly related to adapters.



Adapter interfaces

This diagram shows that an adapter knows a data source. The adapter listens to the data source, which notifies it whenever a new `IlpObject` instance is created. Once a new object is added to the data source, the adapter creates the corresponding `IlpRepresentationObject` instance if the adapter configuration allows it.

In some cases, adapters can also contain temporary representation objects that are created independently of the corresponding business object (before it is created and retrieved from the data source). These temporary objects are stored in the adapter along with a means to determine whether the corresponding business object has been created. For more information, see *Creating a temporary representation object*.

All adapters suppor filtering of business objects. When a filter is set to an adapter, only `IlpObject` instances in the data source that satisfy specified conditions are represented in the representation model. This filter is defined by the interface `IlpFilter`. For more information, see the information on each graphic component.

Using JViews products in Eclipse RCP applications

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse™ development environment and the Eclipse Rich Client Platform (RCP). This topic describes how to use JViews TGO inside Eclipse or RCP. It shows you how to display network, equipment, table, and tree components embedded in an SWT window.

Installing the JViews runtime plugin

JViews provides an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT widget. It allows you to use `IlpNetwork`, `IlpEquipment`, `IlpTree`, and `IlpTable` objects in an SWT window, together with other SWT or JFace controls. In this way, it provides a bridge between the AWT/Swing windowing system and the SWT windowing system.

In order to install the IBM® ILOG® JViews Eclipse plugins, you need to install from the local site as shown below.

For Eclipse 3.3:

1. Launch your Eclipse installation.
2. Go to **Help/Software Updates/Find And Install**.
3. In the `Install/Update` dialog box, click **Search for new features to install**.
4. Define a New Local Site with the directory `<installdir>/jviews-framework86/tools/ilog.views.eclipse.update.site`.
5. Select the features you want to install.

For Eclipse 3.4:

1. Launch your Eclipse installation.
2. Go to **Help/Software Updates** and select the **Available Software** tab.
3. Add a new local site: Click **Add Site**, then **Local** and specify the directory `<installdir>/jviews-framework86/tools/ilog.views.eclipse.update.site`
4. Select the features you want to install, and press the **Install** button.

Providing access to class loaders

Many services in JViews need to look up a resource. Since the classical way to provide access to resources is a classloader, JViews uses classloaders for this purpose. But in Eclipse/RCP applications, each plugin corresponds to a classloader, and the JViews classloader sees only its own resources, not the application resources. To fix this problem, you can register plugin classloaders with JViews through the `IlvClassLoaderUtil.registerClassLoader` function. Each resource lookup then considers the registered classloaders and, if the plugins are configured accordingly, also considers the dependencies of the registered classloaders.

The code for doing this is usually located in a plugin activator class. For example:

```
public class MyPluginActivator extends AbstractUIPlugin
{
    /**
     * This method is called upon plugin activation
     */
    public void start(BundleContext context) throws Exception {
      super.start(context);
      IlvClassLoaderUtil.registerClassLoader(getClass().getClassLoader());
    }

    /**
     * This method is called when the plugin is stopped
     */
    public void stop(BundleContext context) throws Exception {
      super.stop(context);
      IlvClassLoaderUtil.unregisterClassLoader(getClass().getClassLoader());
    }

  }
```

The overriding of `stop()` is necessary so that, when the plugin gets unloaded, JViews gets notified about the plugin that is going to stop and can drop references to its resources or instances of its classes. The activator plugin is usually also the place where `IlvProductUtil.registerApplication` is called. See section Before you start deploying an application for an example.

The bridge between AWT/Swing and SWT

The bridge between the AWT/Swing windowing system and the SWT windowing system consists of an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT widget. This class allows you to use `IlpNetwork`, `IlpEquipment`, `IlpTree`, and `IlpTable` objects in an SWT window, together with other SWT or JFace controls.

The following code shows how to create a bridge object:

```

Composite parent = ...;
IlpNetwork network = new IlpNetwork();
ControlSWTnetwork = new IlvSwingControl(parent, SWT.NONE, network);
```

At the JViews Framework level, the bridge between the AWT/Swing windowing system and the SWT windowing system consists of an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT widget. This class allows you to use `IlvManager` or `IlvJManagerViewPanel` objects in an SWT window, together with other SWT or JFace controls.

The following code shows how to create a bridge object at the JViews Framework level:

```
Composite parent = ...;
IlvManagerView mgrView = ...;
IlvJManagerViewPanel jmgrView = new IlvJManagerViewPanel(mgrView);
ControlSWTview = new IlvSwingControl(parent, SWT.NONE, jmgrView);
```

Using `IlvSwingControl` instead of the native SWT_AWT class has the following benefits:

♦ Simplicity: it is easier to use, since you do not have to worry about the details of the `Component` hierarchy (see *http://java.sun.com/javase/6/docs/api/java/awt/Component.html*).

- Portability: `IlvSwingControl` also works on platforms that do not have SWT_AWT, like X11/Motif® and MacOS® X 10.4.
- Less flickering: on Linux®/Gtk, flickering is reduced.
- Popup menus: popup menus can be positioned on each `Component` inside the AWT component hierarchy. For details of components, see *http://java.sun.com/javase/6/docs/api/java/awt/Component.html*.
- Better size management: the size management between SWT and AWT (`LayoutManager`) is integrated.
- Focus: it provides a workaround for a focus problem on Microsoft® Windows® platforms.

Note: The `IlvSwingControl` bridge is not supported on all platforms. It is only supported on Windows, UNIX® with X11 (Linux, Solaris™, AIX®, HP-UX®), and MacOS X 10.4 or later.

The `IlvSwingControl` bridge does not support arbitrary JComponents. Essentially, components that provide text editing are not supported. See `IlvSwingControl` for a precise description of the limitations.

Threading modes

You can handle the SWT-Swing user interface events in one or two threads.

Note: Single-thread mode is incompatible with AWT/Swing Dialogs. If you use single-thread mode, you cannot use AWT `Dialogs`, Swing `JDialogs`, or modal `JInternalFrames` in your application. There are also some other limitations. See the class `IlvEventThreadUtil` for a precise description of the limitations.

- Two-thread mode

  The SWT events are handled in the SWT event thread and AWT/Swing events are handled in the AWT/Swing event thread. This is the default mode.

  You can switch between the two threads by using the SWT method `Display.asyncExec()` and the AWT method `EventQueue.invokeLater()`.

  If your application uses this mode, you must be careful to:

  - Make API calls on SWT widgets only in the SWT event thread. Otherwise, you will get `SWTExceptions` of type `ERROR_THREAD_INVALID_ACCESS`.
  - Make API calls on JComponents, which include `IlpNetwork`, `IlpEquipment`, `IlpTree`, and `IlpTable`, only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

At the JViews Framework level, make API calls on JComponents, which include `IlvManager` and `IlvJManagerViewPanel`, only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

♦ Single-thread mode

In single-thread mode, SWT and AWT/Swing events are handled in the same thread.

Single-thread mode reduces the risk of producing deadlocks.

Enable this mode by calling `setAWTThreadRedirect` or `enableAWTThreadRedirect()` early during initialization.

The following example shows how to enable single-thread mode:

```
// Switch single-event-thread mode during a static initialization.
      static {
          IlvEventThreadUtil.enableAWTThreadRedirect();
      }
```

If you are using JComponents other than `IlpNetwork`, `IlpEquipment`, `IlpTree`, and `IlpTable` in your application, your JComponents must use the method `IlvSwingUtil.isDispatchThread()` rather than *`EventQueue.isDispatchThread()`* or *`SwingUtilities.isEventDispatchThread()`*.

For example:

```

   // Switch single-event-thread mode during a static initialization.
      static {
          IlvEventThreadUtil.enableAWTThreadRedirect();
      }
```

Note: This mode is incompatible with AWT/Swing Dialogs. If you use single-thread mode, you cannot use AWT `Dialogs`, Swing `JDialogs`, or modal `JInternalFrames` in your application. There are also some other limitations. See the class `IlvEventThreadUtil` for a precise description of the limitations.

At the JViews Framework level, if you are using JComponents other than `IlvManager` and `IlvJManagerViewPanel` in your application, your JComponents must use the method `isDispatchThread()` rather than `EventQueue.isDispatchThread()` (see *http://java.sun.com/javase/6/docs/api/java/awt/EventQueue.html#isDispatchThread()*) or `SwingUtilities.isEventDispatchThread()` (see *http://java.sun.com/javase/6/docs/api/javax/swing/SwingUtilities.html#isEventDispatchThread()*.)

Index