IBM ILOG JViews Graph Layout for Eclipse V8.6

Property sheets

© Copyright International Business Machines Corporation 1987, 2009
US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Copyright

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Trademarks

IBM, the IBM logo, ibm.com, WebSphere, ILOG, the ILOG design, and CPLEX are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at *http://www.ibm.com/legal/copytrade.shtml*

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Notices

For further copyright information see *<installdir>* `jviews-graphlayout-eclipse86/license/notices.txt`.

Table of contents

Working with property sheets..........5

Tabbed property sheet sections..........6

Adding the property sheets to the application..........7

Adding basic edit policies..........10

Managing persistence..........13

Introducing persistence..........14

Using EMF support for persistence..........16

Using support for persistence in your GMF application..........22

Customizing property sheets..........27

Providing predefined layouts..........28

Filtering customizable properties..........31

Customizing default values..........33

© Copyright IBM Corp. 1987, 2009

Working with property sheets

Describes the use of property sheets and how to add them to your GEF and GMF applications.

In this section

Tabbed property sheet sections
Describes the use of tabbed property sheet sections.

Adding the property sheets to the application
Describes how to add the property sheets to your application by making extension declarations in your plugin.xml file.

Adding basic edit policies
Describes how to add basic edit policies to make property sheets interact with your diagram selection.

© Copyright IBM Corp. 1987, 2009

Tabbed property sheet sections

You can include property sheets in your application to make the layout settings editable at run time. These take the form of *tabbed property sheet* sections that can be added to the existing property sheets of your GEF or GMF application. If you are not familiar with tabbed property sheets, there is a good introduction at this URL: *http://www.eclipse.org/articles/Article-Tabbed-Properties/tabbed_properties_view.html*.

If you are working with GEF, you need to make your plug-in project depend on the plug-in `ilog.views.eclipse.graphlayout.properties`. If you are working with GMF, the `ilog.views.eclipse.graphlayout.gmf.properties` plug-in is more suitable.

The following table lists the sections and filters available. The classes are in `ilog.views.eclipse.graphlayout.properties.sections` and the filter classes are in `ilog.views.eclipse.graphlayout.properties.filters`.

Class (Filter)	Usage
GraphLayoutPropertySection(LayoutPropertySectionFilter)	Choosing and configuring the graph layout (includes control.GraphLayoutControlPropertySection).
LinkLayoutPropertySection(LayoutPropertySectionFilter)	Choosing and configuring the link layout (includes control.LinkLayoutControlPropertySection).
LabelLayoutPropertySection(LayoutPropertySectionFilter)	Choosing and configuring the label layout (includes control.LabelLayoutControlPropertySection).
GraphLayoutNodePropertySection(GraphLayoutNodePropertySectionFilter)	Configuring per-node settings for graph and link layout.
GraphLayoutConnectionPropertySection(GraphLayoutConnectionPropertySectionFilter)	Configuring per-connection settings for graph and link layout.
LabelLayoutNodeLabelPropertySection(LabelLayoutNodeLabelPropertySectionFilter)	Configuring per-node label settings for label layout.
LabelLayoutConnectionLabelPropertySection(LabelLayoutConnectionLabelPropertySectionFilter)	Configuring per-connection label settings for label layout.
GraphLayoutControlPropertySection(LayoutPropertySectionFilter)	Choosing a predefined graph layout.
LinkLayoutControlPropertySection(LayoutPropertySectionFilter)	Choosing a predefined link layout.
LabelLayoutControlPropertySection(LayoutPropertySectionFilter)	Choosing a predefined label layout.

You can display a section to both choose and configure a layout or display a *control* section to choose a predefined layout. Filter classes are provided so that you can easily bind these sections to your current diagram selection (canvas, nested graph, node, link, or label).

There are specialized versions for GMF in `ilog.views.eclipse.graphlayout.gmf.properties.sections` and in `ilog.views.eclipse.graphlayout.gmf.properties.sections.control`.

You can subclass these classes so that you can configure them.

Adding the property sheets to the application

Declaring categories and tabs as described in this topic gives the following tabs in the following order:

1. Graph Layout
2. Link Layout
3. Label Layout
4. Layout Properties

Adding categories

The following XML example shows a `propertyContributor` extension declaration.

```
<extension point="org.eclipse.ui.views.properties.tabbed.propertyContributor">

  <propertyContributor contributorId="your_contributor_id">
    <propertyCategory category="graphLayout"/>
    <propertyCategory category="linkLayout"/>
    <propertyCategory category="labelLayout"/>
    <propertyCategory category="layoutProperties"/>
  </propertyContributor>
</extension>
```

The `layoutProperties` category groups per-object layout properties.

Adding tabs

The following XML example shows a `propertyTabs` extension declaration.

```
<extension point=”org.eclipse.ui.views.properties.tabbed.propertyTabs">
  <propertyTabs contributorId="your_contributor_id">
    <propertyTab
      category="graphLayout"
      id="property.tab.GraphLayoutPropertySection"
      label="Graph Layout"/>
    <propertyTab
      category="linkLayout"
      id="property.tab.LinkLayoutPropertySection"
      label="Link Layout"/>
    <propertyTab
      category="labelLayout"
      id="property.tab.LabelLayoutPropertySection"
      label="Label Layout"/>
    <propertyTab
      category="layoutProperties"
      id="property.tab.LayoutPropertiesPropertySection"
      label="Layout Properties"/>
```

```
  </propertyTabs>
</extension>
```

Adding sections

The following XML example shows how to add the GLE property sheet sections.

```
<extension point="org.eclipse.ui.views.properties.tabbed.propertySections">
  <propertySections contributorId="your_contributor_id">
    <propertySection id="property.section.GraphLayoutPropertySection"
      filter="ilog.views.eclipse.graphlayout.properties.filters.
        LayoutPropertySectionFilter"
      class="ilog.views.eclipse.graphlayout.properties.sections.
        GraphLayoutPropertySection"
      tab="property.tab.GraphLayoutPropertySection">
    </propertySection>
    <propertySection id="property.section.LinkLayoutPropertySection"
      filter="ilog.views.eclipse.graphlayout.properties.filters.
        LayoutPropertySectionFilter"
      class="ilog.views.eclipse.graphlayout.properties.sections.
        LinkLayoutPropertySection"
      tab="property.tab.LinkLayoutPropertySection">
    </propertySection>
    <propertySection id="property.section.LabelLayoutPropertySection"
      filter="ilog.views.eclipse.graphlayout.properties.filters.
        LayoutPropertySectionFilter"
      class="ilog.views.eclipse.graphlayout.properties.sections.
        LabelLayoutPropertySection "
      tab="property.tab.LabelLayoutPropertySection">
    </propertySection>
    <propertySection id="property.section.GraphLayoutNodePropertySection"
      filter="ilog.views.eclipse.graphlayout.properties.filters.
        GraphLayoutNodePropertySectionFilter"
      class="ilog.views.eclipse.graphlayout.properties.sections.
        GraphLayoutNodePropertySection"
      tab="property.tab.LayoutPropertiesPropertySection">
    </propertySection>
    <propertySection id=
      "property.section.GraphLayoutConnectionPropertySection"
      filter="ilog.views.eclipse.graphlayout.properties.filters.
        GraphLayoutConnectionPropertySectionFilter"
      class="ilog.views.eclipse.graphlayout.properties.sections.
        GraphLayoutConnectionPropertySection"
      tab="property.tab.LayoutPropertiesPropertySection">
    </propertySection>
    <propertySection id=
      "property.section.LabelLayoutConnectionPropertySection"
      filter="ilog.views.eclipse.graphlayout.properties.filters.
        LabelLayoutConnectionPropertySectionFilter"
      class="ilog.views.eclipse.graphlayout.properties.sections.
        LabelLayoutConnectionPropertySection"
      tab="property.tab.LayoutPropertiesPropertySection">
    </propertySection>
```

```
  </propertySections>
</extension>
```

To see how the property sections are bound to the editor, see the GMF Diagram Editor sample.

Adding basic edit policies

Edit parts need to return specific commands so that property sheets can edit current layouts. For this purpose, edit policies are provided. To get started, install basic edit policies that return commands for customizing the layout instances directly: `BasicLayoutEditPolicy` and `BasicLayoutPropertyEditPolicy`.

To use persistence capabilities, you must set up other edit policies. See *Using EMF support for persistence* and *Using support for persistence in your GMF application*.

The first basic policy, which is given by `BasicLayoutEditPolicy`, returns commands allowing control sections to change a layout algorithm by changing a layout instance set on a layout source. This edit policy needs to be set up on graphers that implement the `ILayoutSource` interface.

The second basic policy, which is given by `BasicLayoutPropertyEditPolicy`, returns commands allowing the property sheets to edit the properties of a layout set on a layout source. This edit policy must be set up on graphers for editing global properties and on nodes, connections, node labels, and connection labels as required for editing local properties.

The following code samples show how edit policies are instantiated on different kinds of edit parts.

The following code sample is for a grapher.

```
import ilog.views.eclipse.graphlayout.IGrapherEditPart;
import ilog.views.eclipse.graphlayout.edit.editpolicies.BasicLayoutEditPolicy;
import ilog.views.eclipse.graphlayout.edit.editpolicies.
BasicLayoutPropertyEditPolicy;
import ilog.views.eclipse.graphlayout.edit.editpolicies.LayoutEditPolicyRoles;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class MyDiagramEditPart extends AbstractGraphicalEditPart implements
  IGrapherEditPart {

  ...

  @Override
  protected void createDefaultEditPolicies() {
    super.createDefaultEditPolicies();
    ...
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_EDIT_ROLE,
        new BasicLayoutEditPolicy());
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_PROPERTY_EDIT_ROLE,
        new BasicLayoutPropertyEditPolicy());
  }

}
```

In the case of the grapher, both edit policies are instantiated.

The following code sample is for a node.

```
import ilog.views.eclipse.graphlayout.edit.editpolicies.
BasicLayoutPropertyEditPolicy;
import ilog.views.eclipse.graphlayout.edit.editpolicies.LayoutEditPolicyRoles;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class MyNodeEditPart extends AbstractGraphicalEditPart implements
  org.eclipse.gef.NodeEditPart {

  ...

  @Override
  protected void createDefaultEditPolicies() {
    super.createDefaultEditPolicies();
    ...
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_PROPERTY_EDIT_ROLE,
        new BasicLayoutPropertyEditPolicy());
  }

}
```

In the case of nodes (as for connections, node labels, and connection labels), only a `BasicLayoutPropertyEditPolicy` object is set up.

Managing persistence

Describes how to manage persistence in your GMF application or EMF model-based application.

In this section

Introducing persistence
Describes the support available for persistence.

Using EMF support for persistence
Describes how to manage persistent layout configurations in your EMF model.

Using support for persistence in your GMF application
Describes how to use support for persistence in your GMF application.

© Copyright IBM Corp. 1987, 2009

Introducing persistence

In most cases, having property sheets without persistent settings is not much use. Since adding persistence capabilities to layout configuration is not trivial, support is supplied with JViews Graph Layout for Eclipse.

Persistence information

The properties made persistent are described through an extension point: `layoutPersistenceInfo`. You can customize what properties must be made persistent by creating an extension using this extension point. Your extension can be global and can override the default persistence information by indicating a higher priority; it can also be specific to a given editor by indicating the ID of the editor.

The filter associated by default with property sheets will automatically filter the properties. See *Filtering customizable properties* .

Keep in mind that using this persistence environment implies that setting a layout or a layout property from the property sheet is done through the model. You can look at the `ilog.views.eclipse.graphlayout` plugin descriptor to see how persistent properties have been declared by default.

Warning: If you predefine a layout with certain properties that are not made persistent, they will be lost unless you define them as default values.

Example of partial persistence

The following XML example shows how to make persistent only the flow direction and global link style in the hierarchical layout.

```
<extension
  point="ilog.views.eclipse.graphlayout.layoutPersistenceInfo">
  <layout
    layoutClass="ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout">
    <persistentProperty name="flowDirection"/>
    <persistentProperty name="globalLinkStyle"/>
  </layout>
  <Priority name="High">
  </Priority>
</extension>
```

In the XML example shown, only the flow direction and the global link style are customizable; other properties (including, for example, the auto layout property) are not taken into account when setting a hierarchical link layout from the property sheets.

Persistence support

The persistence support consists of the following:

- Edit policies, commands, and listeners for updating layouts.
- Default string representations that enrich the information about the objects in your EMF or GMF notation model. These representations are in XML.

For EMF models, a mechanism is supplied in the *GLE with EMF support* feature to save layout configurations in your models.

For GMF, the procedure is similar to EMF but with dedicated classes. The main difference is that the configuration is stored in the GMF notation model and so you do not need to adapt your model. In the GMF case, the term “configuration” is replaced by the term “style”. This is because the extension of the GMF notation model is done through custom styles.

Using EMF support for persistence

To manage persistent layout configurations in a model-based EMF application:

1. You need to make your plug-in project depend on the `ilog.views.eclipse.graphlayout.emf.edit` plug-in, which is part of the "GLE with EMF Support" feature.

2. The grapher model object that the configuration concerns must contain string attributes for the layouts you want to make configurable. Therefore, if you want to customize the graph layout, the link layout, and the label layout, you need to create graph, link, and label layout string attributes.

3. The same applies for local configurations and contextual properties. Typically the model object of a node can contain two attributes for the local configurations of graph layout and link layout, and the model object of a node label can contain one attribute for the local configuration of label layout.

4. Once the model is designed to host these settings, you need to instantiate a specific `ILayoutSource` implementation in the graphers: a `PersistentEMFLayoutSource` object. This source listens for model changes in order to update the layout algorithms.

   See *Example of a layout source* for an example.

5. In addition to a specific `ILayoutSource` object, you need to instantiate local configuration controllers in nodes, connections, node labels and connection labels. These controllers listen to the model objects referenced by these edit parts and update the layout algorithms accordingly.

   The following configuration controllers are supplied:

   ♦ `EMFNodeOrConnectionConfigurationController` for node and connection edit parts

   ♦ `EMFLabelConfigurationController` for labels

   See *Example of configuration controllers* for an example.

6. You need to install edit policies that instantiate commands to modify the model instead of modifying the layout instances directly. Instead of installing `BasicLayoutEditPolicy` and `BasicLayoutPropertyEditPolicy`, you need to set up `EMFLayoutEditPolicy` and `EMFLayoutPropertyEditPolicy`.

   See *Example of edit policies* for examples.

Example of a layout source

The following code sample shows the use of a PersistentEMFLayoutSource instance.

```
import ilog.views.eclipse.graphlayout.IGrapherEditPart;
import ilog.views.eclipse.graphlayout.source.ILayoutSource;
import ilog.views.eclipse.graphlayout.emf.edit.source.LayoutSource;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class MyDiagramEditPart extends AbstractGraphicalEditPart implements
  IGrapherEditPart {
```

```
  // reference on the layout source implementation
  private PersistentEMFLayoutSource myLayoutSource;

  ...

  @Override
  public void activate() {
    // we choose to instantiate the layout source when the EditPart
    // is activated. It is important that it is
    // instantiated before its children are activated so that
    // children can behave consequently
    myLayoutSource = new PersistentEMFLayoutSource(this);
    myLayoutSource.setGraphLayoutConfigurationAttribute(
      MyModelPackage.eINSTANCE.getMyGrapher_GraphLayoutConfiguration());
    myLayoutSource.setLinkLayoutConfigurationAttribute(
      MyModelPackage.eINSTANCE.getMyGrapher_LinkLayoutConfiguration());
    myLayoutSource.setLabelLayoutConfigurationAttribute(
      MyModelPackage.eINSTANCE.getMyGrapher_LabelLayoutConfiguration());
    super.activate();
  }

  @Override
  public void deactivate() {
    super.deactivate();
    // cleanup
    myLayoutSource.dispose();
    myLayoutSource = null;
  }

  @Override
  public Object getAdapter(Class adapter) {
    if (adapter.equals(ILayoutSource.class)) {
      return myLayoutSource;
    }
    return super.getAdapter(adapter);
  }

  public boolean isTopLevel() {
    return true;
  }

}
```

Example of configuration controllers

The following code sample shows how to instantiate configuration controllers for nodes or connections.

```
Import ilog.views.eclipse.graphlayout.emf.edit.source.
EMFNodeOrConnectionConfigurationController;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class MyNodeEditPart extends AbstractGraphicalEditPart implements
  org.eclipse.gef.NodeEditPart {
```

```
  // reference on the configuration controller
  private EMFNodeOrConnectionConfigurationController configController;

  ...

  @Override
  public void activate() {
    configController = new EMFNodeOrConnectionConfigurationController(this);
    configController.setGraphLayoutLocalConfigurationAttribute(
      MyModelPackage.eINSTANCE.getMyObject_LocalGraphLayoutConfiguration());
    configController.setLinkLayoutLocalConfigurationAttribute(
      MyModelPackage.eINSTANCE.getMyObject_LocalLinkLayoutConfiguration());
    super.activate();
  }

  @Override
  public void deactivate() {
    configController.dispose();
    configController = null;
    super.deactivate();
  }

}
```

The same code is used for node and connection edit parts, with the possible exception of the getters for retrieving the meta-information of the attributes.

The following code sample shows how to instantiate configuration controllers for labels.

```
import ilog.views.eclipse.graphlayout.emf.edit.source.
EMFLabelConfigurationController;
import ilog.views.eclipse.graphlayout.runtime.labellayout.ILabelEditPart;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;


public class MyLabelEditPart extends AbstractGraphicalEditPart
  implements ILabelEditPart {

  // reference on the configuration controller
  private LabelConfigurationController configController;

  ...

  @Override
  public void activate() {
    configController = new EMFLabelConfigurationController(this);
    configController.setLabelLayoutLocalConfigurationAttribute(
      MyModelPackage.eINSTANCE.getMyLabel_LocalLabelLayoutConfiguration());
    super.activate();
  }

  @Override
  public void deactivate() {
```

```
    configController.dispose();
    configController = null;
    super.deactivate();
  }

}
```

Example of edit policies

These code samples show how to instantiate edit policies at diagram, node, and label levels.

The following code sample shows how to instantiate edit policies at diagram level.

```
import ilog.views.eclipse.graphlayout.IGrapherEditPart;
import ilog.views.eclipse.graphlayout.emf.edit.editpolicies.EMFLayoutEditPolicy;
import ilog.views.eclipse.graphlayout.emf.edit.editpolicies.
EMFLayoutPropertyEditPolicy;
import ilog.views.eclipse.graphlayout.edit.editpolicies.LayoutEditPolicyRoles;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class MyDiagramEditPart extends AbstractGraphicalEditPart implements
  IGrapherEditPart {

  ...

  @Override
  protected void createDefaultEditPolicies() {
    super.createDefaultEditPolicies();
    ...
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_EDIT_ROLE,
      new EMFLayoutEditPolicy(
        MyModelPackage.eINSTANCE.getMyGrapher_GraphLayoutConfiguration(),
        MyModelPackage.eINSTANCE.getMyGrapher_LinkLayoutConfiguration(),
        MyModelPackage.eINSTANCE.getMyGrapher_LabelLayoutConfiguration()
    ));
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_PROPERTY_EDIT_ROLE,
      new EMFLayoutPropertyEditPolicy(
        MyModelPackage.eINSTANCE.getMyGrapher_GraphLayoutConfiguration(),
        MyModelPackage.eINSTANCE.getMyGrapher_LinkLayoutConfiguration(),
        MyModelPackage.eINSTANCE.getMyGrapher_LabelLayoutConfiguration(),
       null, // in the case of nested graphs, it makes sense to also indicate

          the per node graph layout configuration attribute
       null, // in the case of nested graphs, it makes sense to also indicate

          the per node link layout configuration attribute
        null // do not make sense to have per label layout configuration here

    ));
  }

}
```

The following code sample shows how to instantiate edit policies at node level.

```
import ilog.views.eclipse.graphlayout.emf.edit.editpolicies.
EMFLayoutPropertyEditPolicy;
import ilog.views.eclipse.graphlayout.edit.editpolicies.LayoutEditPolicyRoles;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class MyNodeEditPart extends AbstractGraphicalEditPart implements
  org.eclipse.gef.NodeEditPart {

  ...

  @Override
  protected void createDefaultEditPolicies() {
    super.createDefaultEditPolicies();
    ...
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_PROPERTY_EDIT_ROLE,
      new EMFLayoutPropertyEditPolicy(
        null, null, null, // no global configurations
        MyModelPackage.eINSTANCE.getMyObject_LocalGraphLayoutConfiguration(),

        MyModelPackage.eINSTANCE.getMyObject_LocalLinkLayoutConfiguration(),
        null // does not make sense to have label layout configuration here
    ));
  }

}
```

The following code sample shows how to instantiate edit policies at label level.

```
import ilog.views.eclipse.graphlayout.emf.edit.editpolicies.
EMFLayoutPropertyEditPolicy;
import ilog.views.eclipse.graphlayout.edit.editpolicies. LayoutEditPolicyRoles;
import ilog.views.eclipse.graphlayout.runtime.labellayout.ILabelEditPart;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class MyLabelEditPart extends AbstractGraphicalEditPart
  implements ILabelEditPart {

  ...

  @Override
  protected void createDefaultEditPolicies() {
    super.createDefaultEditPolicies();
    ...
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_PROPERTY_EDIT_ROLE,
      new EMFLayoutPropertyEditPolicy(
        null, null, null, // no global configurations
        null, // does not make sense to have graph layout configuration here
        null, // does not make sense to have link layout configuration here
        MyModelPackage.eINSTANCE.getMyLabel_LocalLinkLayoutConfiguration()
    ));
  }
```

```
}
```

Using support for persistence in your GMF application

To manage persistent layout configurations in a GMF application:

1. You need to make your GMF editor project depend on the `ilog.views.eclipse.graphlayout.gmf.properties` plug-in.
2. Use a `ilog.views.eclipse.graphlayout.gmf.edit.source.PersistentGMFLayoutSource` and make it listen to the notation model.
3. Set up edit policies.
4. Set up style controllers. They have the same purpose as the configuration controllers for EMF: to listen to the model in order to update the layout algorithms.

After these steps, the property sheets will be completely integrated with your GMF editor.

Example of a layout source and edit policy setup

The following code sample shows a layout source that listens to the GMF notation model.

```
import ilog.views.eclipse.graphlayout.IGrapherEditPart;
import ilog.views.eclipse.graphlayout.source.ILayoutSource;
import ilog.views.eclipse.graphlayout.edit.editpolicies.LayoutEditPolicyRoles;
import ilog.views.eclipse.graphlayout.gmf.edit.notation.LayoutNotationPackage;
import ilog.views.eclipse.graphlayout.gmf.edit.source.PersistentGMFLayoutSource;
import ilog.views.eclipse.graphlayout.gmf.edit.editpolicies.
DefaultGMFLayoutEditPolicy;
import ilog.views.eclipse.graphlayout.gmf.edit.editpolicies.
DefaultGMFLayoutPropertyEditPolicy;


import org.eclipse.gef.editparts.AbstractGraphicalEditPart;
import org.eclipse.gmf.runtime.diagram.ui.editparts.DiagramEditPart;
import org.eclipse.gmf.runtime.notation.NotationPackage;

public class MyDiagramEditPart extends DiagramEditPart implements
    IGrapherEditPart {

  // reference on the layout source implementation
  private PersistentGMFLayoutSource myLayoutSource;

  ...

  @Override
  public void activate() {
    // we choose to instantiate the layout source when the EditPart
    // is activated. It is important that it is
    // instantiated before its children are activated so that
    // children can behave consequently
    myLayoutSource = new PersistentGMFLayoutSource(this);
    super.activate();
  }

  @Override
  public void deactivate() {
```

```
    super.deactivate();
    // cleanup
    myLayoutSource.dispose();
    myLayoutSource = null;
  }

  @Override
  protected void addNotationalListeners() {
    super.addNotationalListeners();
    // listen to notation model changes to update the layout
    addListenerFilter(PersistentGMFLayoutSource.STYLE_NOTATION_FILTER_ID,
        myLayoutSource, getDiagramView(), NotationPackage.eINSTANCE
            .getView_Styles());
    addListenerFilter(
        PersistentGMFLayoutSource.STYLE_GRAPH_LAYOUT_NOTATION_FILTER_ID,
        myLayoutSource, getDiagramView(), LayoutNotationPackage.eINSTANCE
            .getLayoutStyle_GraphLayoutConfiguration());
    addListenerFilter(
        PersistentGMFLayoutSource.STYLE_LINK_LAYOUT_NOTATION_FILTER_ID,
        myLayoutSource, getDiagramView(), LayoutNotationPackage.eINSTANCE
            .getLayoutStyle_LinkLayoutConfiguration());
    addListenerFilter(
        PersistentGMFLayoutSource.STYLE_LABEL_LAYOUT_NOTATION_FILTER_ID,
        myLayoutSource, getDiagramView(), LayoutNotationPackage.eINSTANCE
            .getLayoutStyle_LabelLayoutConfiguration());
  }

  @Override
  protected void removeNotationalListeners() {
    removeListenerFilter(PersistentGMFLayoutSource.STYLE_NOTATION_FILTER_ID);

    removeListenerFilter(
      PersistentGMFLayoutSource.STYLE_GRAPH_LAYOUT_NOTATION_FILTER_ID);
    removeListenerFilter(
      PersistentGMFLayoutSource.STYLE_LINK_LAYOUT_NOTATION_FILTER_ID);
    removeListenerFilter(
      PersistentGMFLayoutSource.STYLE_LABEL_LAYOUT_NOTATION_FILTER_ID);
    super.removeNotationalListeners();
  }

  @Override
  protected void createDefaultEditPolicies() {
    super.createDefaultEditPolicies();
    ...
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_EDIT_ROLE,
      new DefaultGMFLayoutEditPolicy());
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_PROPERTY_EDIT_ROLE,
      new DefaultGMFLayoutPropertyEditPolicy());
  }

  @Override
  public Object getAdapter(Class adapter) {
    if (adapter.equals(ILayoutSource.class)) {
      return myLayoutSource;
```

```
    }
    return super.getAdapter(adapter);
  }

  public boolean isTopLevel() {
    return true;
  }

}
```

In this code sample, the dedicated edit policies for GMF are set up as well as the layout source.

Example of style controllers

The following code sample shows how to use a `ilog.views.eclipse.graphlayout.gmf.edit.source.NodeOrConnectionStyleController` for a node or connection.

```
import ilog.views.eclipse.graphlayout.edit.editpolicies.LayoutEditPolicyRoles;
import ilog.views.eclipse.graphlayout.gmf.edit.notation.LayoutNotationPackage;
import ilog.views.eclipse.graphlayout.gmf.edit.editpolicies.
DefaultGMFLayoutPropertyEditPolicy;
import ilog.views.eclipse.graphlayout.gmf.edit.source.
NodeOrConnectionStyleController;

import org.eclipse.gmf.runtime.diagram.ui.editparts.ShapeNodeEditPart;
import org.eclipse.gmf.runtime.notation.NotationPackage;

public class MyNodeEditPart extends ShapeNodeEditPart {

  // reference on the style controller
  private NodeOrConnectionStyleController layoutStyleController;

  ...

  @Override
  protected void addNotationalListeners() {
    super.addNotationalListeners();
    layoutStyleController = new NodeOrConnectionStyleController(this);
    addListenerFilter(
      NodeOrConnectionStyleController.
        NODE_CONNECTION_STYLE_NOTATION_FILTER_ID,
      layoutStyleController, (View) getAdapter(View.class),
      NotationPackage.eINSTANCE.getView_Styles());
    addListenerFilter(
      NodeOrConnectionStyleController.
        NODE_CONNECTION_STYLE_GRAPH_LAYOUT_NOTATION_FILTER_ID,
      layoutStyleController, (View) getAdapter(View.class),
      LayoutNotationPackage.eINSTANCE.
        getNodeOrConnectionStyle_GraphLayoutConfiguration());
    addListenerFilter(
        NodeOrConnectionStyleController.
          NODE_CONNECTION_STYLE_LINK_LAYOUT_NOTATION_FILTER_ID,
      layoutStyleController, (View) getAdapter(View.class),
      LayoutNotationPackage.eINSTANCE.
        getNodeOrConnectionStyle_LinkLayoutConfiguration());
```

```
  }

  @Override
  protected void removeNotationalListeners() {
    removeListenerFilter(NodeOrConnectionStyleController.
      NODE_CONNECTION_STYLE_LINK_LAYOUT_NOTATION_FILTER_ID);
    removeListenerFilter(NodeOrConnectionStyleController.
      NODE_CONNECTION_STYLE_GRAPH_LAYOUT_NOTATION_FILTER_ID);
    removeListenerFilter(NodeOrConnectionStyleController.
      NODE_CONNECTION_STYLE_NOTATION_FILTER_ID);
    layoutStyleController.dispose();
    layoutStyleController = null;
    super.removeNotationalListeners();
  }

  @Override
  protected void createDefaultEditPolicies() {
    super.createDefaultEditPolicies();
    ...
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_PROPERTY_EDIT_ROLE,
        new DefaultGMFLayoutPropertyEditPolicy());
  }
}
```

The following code sample shows how to use a `ilog.views.eclipse.graphlayout.gmf.edit.source.LabelStyleController` for a label.

```
import ilog.views.eclipse.graphlayout.edit.editpolicies. LayoutEditPolicyRoles;
import ilog.views.eclipse.graphlayout.gmf.edit.notation.LayoutNotationPackage;
import ilog.views.eclipse.graphlayout.gmf.edit.editpolicies.
DefaultGMFLayoutPropertyEditPolicy;
import ilog.views.eclipse.graphlayout.gmf.edit.source.LabelStyleController;
import ilog.views.eclipse.graphlayout.runtime.labellayout.ILabelEditPart;

import org.eclipse.gmf.runtime.diagram.ui.editparts.LabelEditPart;
import org.eclipse.gmf.runtime.notation.NotationPackage;

public class MyLabelEditPart extends LabelEditPart implements ILabelEditPart
{

  // reference on the style controller
  private LabelStyleController labelStyleController;

  ...

  @Override
  protected void addNotationalListeners() {
    super.addNotationalListeners();
    labelStyleController = new LabelStyleController(this);
    addListenerFilter(LabelStyleController.LABEL_STYLE_NOTATION_FILTER_ID,
        labelStyleController, (View) getAdapter(View.class),
        NotationPackage.eINSTANCE.getView_Styles());
    addListenerFilter(
        LabelStyleController.LABEL_STYLE_LABEL_LAYOUT_NOTATION_FILTER_ID,
```

```
        labelStyleController, (View) getAdapter(View.class),
        LayoutNotationPackage.eINSTANCE.
          getLabelStyle_LabelLayoutConfiguration());
  }

  @Override
  protected void removeNotationalListeners() {
    removeListenerFilter(LabelStyleController.
      LABEL_STYLE_LABEL_LAYOUT_NOTATION_FILTER_ID);
    removeListenerFilter(LabelStyleController.LABEL_STYLE_NOTATION_FILTER_ID)
;
    labelStyleController.dispose();
    labelStyleController = null;
    super.removeNotationalListeners();
  }

  @Override
  protected void createDefaultEditPolicies() {
    super.createDefaultEditPolicies();
    ...
    installEditPolicy(LayoutEditPolicyRoles.LAYOUT_PROPERTY_EDIT_ROLE,
        new DefaultGMFLayoutPropertyEditPolicy());
  }
}
```

Customizing property sheets

Describes how to customize the property sheets in order to adapt them to your application.

In this section

Providing predefined layouts
Describes how to configure the list of predefined layouts.

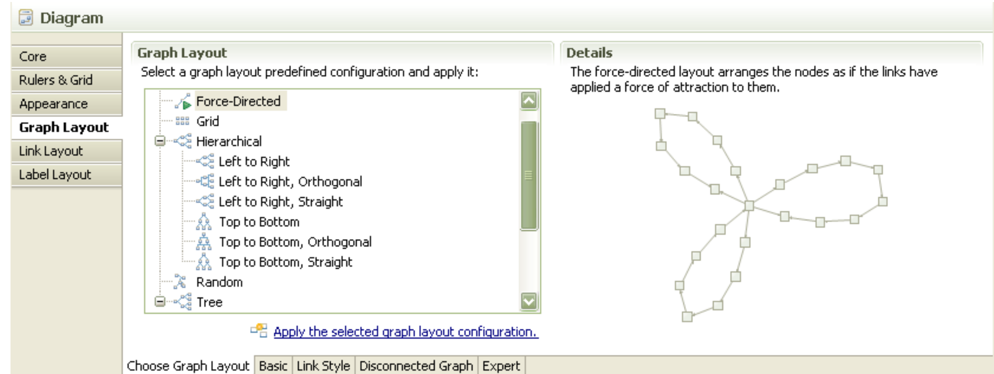
Filtering customizable properties
Describes the use of filters to expose subsets of properties.

Customizing default values
Describes how to customize the way default values are handled.

© Copyright IBM Corp. 1987, 2009

Providing predefined layouts

You can configure the list of predefined layouts available, which is in the graph and link layout control sections.



This list is populated by the `DefaultGraphLayoutConfigurationProvider` class (or the `DefaultLinkLayoutConfigurationProvider` class in the case of the link layout control section). You can derive from the relevant class to remove, modify, or add some predefined configurations. A configuration is encapsulated in the `GraphLayoutConfiguration` class. A configuration indicates the label, the description, the icon, and the preview graph to display; it also indicates the layout to set.

Removing a layout configuration

The following code sample shows how to remove the grid layout configuration from the graph layout control section.

```
import ilog.views.eclipse.graphlayout.properties.sections.control.
DefaultGraphLayoutConfigurationProvider;

public class MyGraphLayoutConfigurationProvider extends
  DefaultGraphLayoutConfigurationProvider {

  @Override
  protected void createConfigurations() {
    super.createConfigurations();
    removeGraphLayoutConfiguration(getGraphLayoutConfiguration
(GRAPH_LAYOUT_GRID));
  }

}
```

After removing a layout configuration, you must modify the configuration provider. For details, see *Modifying configuration providers*.

Modifying a layout configuration

The following code sample shows how to modify the hierarchical layout variants.

```
import ilog.views.eclipse.graphlayout.runtime.hierarchical.
IlvHierarchicalLayout;
import ilog.views.eclipse.graphlayout.properties.sections.control.
DefaultGraphLayoutConfigurationProvider;

public class MyGraphLayoutConfigurationProvider extends
  DefaultGraphLayoutConfigurationProvider {

  private class MyGraphLayoutConfigurationFactory
    extends GraphLayoutConfigurationFactory {

    @Override
    protected IlvHierarchicalLayout createHierarchicalLayout() {
      IlvHierarchicalLayout graphLayout = new IlvHierarchicalLayout();
      // sample modification
      graphLayout.setConnectorStyle(IlvHierarchicalLayout.CENTERED_PINS);
      return graphLayout;
    }

  }

  @Override
  protected GraphLayoutConfigurationFactory
    createGraphLayoutConfigurationFactory() {
    return new MyGraphLayoutConfigurationFactory();
  }

}
```

After modifying a layout configuration, you must modify the configuration provider. For details, see *Modifying configuration providers*.

Modifying configuration providers

Two `DefaultGraphLayoutConfigurationProvider` or `DefaultLinkLayoutConfigurationProvider` intern factories can be overridden: a `GraphFactory` to modify a preview graph and the `GraphLayoutConfigurationFactory` or `LinkLayoutConfigurationFactory` to customize the configuration.

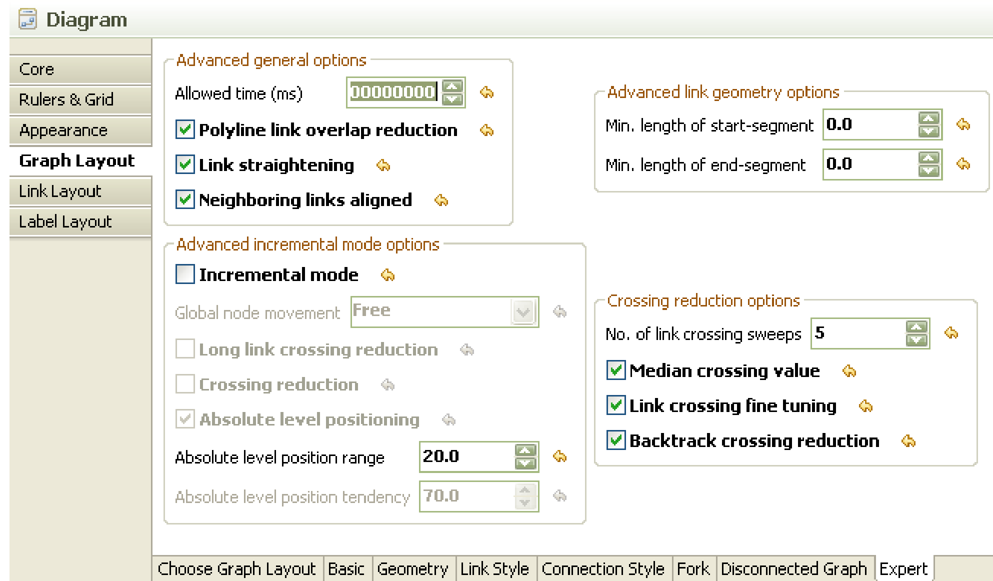
To create a new configuration and add it to the list, you need to create a new `GraphLayoutConfiguration` object and add it using the method `addConfiguration`. If you do not want to use the predefined configuration provider, you can directly create a class that derives from `AbstractGraphLayoutConfigurationProvider` and populate it as required.

You also need your control section to instantiate your configuration provider. The following code sample shows a specialized version of a `GraphLayoutPropertySection` object which instantiates a control section using this provider.

```
import ilog.views.eclipse.graphlayout.properties.sections.
GraphLayoutPropertySection;

import ilog.views.eclipse.graphlayout.properties.sections.control.
AbstractLayoutControlPropertySection;

import ilog.views.eclipse.graphlayout.properties.sections.control.
IGraphLayoutConfigurationProvider;

public class MyGraphLayoutSection extends GraphLayoutPropertySection {

  @Override
  protected AbstractLayoutControlPropertySection createLayoutControlSection()


  {
    return new GraphLayoutControlPropertySection() {

      @Override
      protected IGraphLayoutConfigurationProvider
        createGraphLayoutConfigurationProvider() {
          return new MyGraphLayoutConfigurationProvider();
      }
    };
  }
}
```

You also need to reference the section from your `plugin.xml` file as you referenced the default one.

Filtering customizable properties



Implementations of property filters are set by default as follows:

- The `DefaultPropertyFilter` filter class if you use standard property sheets
- The `GMFDefaultPropertyFilter` filter class if you use GMF property sheets

These filters look for the properties described as persistent and keep these. For details of persistent information, see *Introducing persistence*. You are recommended to override a default filter if you want to remove other properties.

The following code sample shows how to filter the `allowedTime` advanced parameter from different layout property sheets.

```
import ilog.views.eclipse.graphlayout.properties.sections.
AbstractLayoutPropertySection;
 import ilog.views.eclipse.graphlayout.properties.sections.
DefaultPropertyFilter;
 import java.beans.PropertyDescriptor;
 import org.eclipse.gef.GraphicalEditPart;

   public class MyPropertyFilter extends DefaultPropertyFilter {

     public MyPropertyFilter(AbstractLayoutPropertySection section) {
       super(section);
     }

     @Override
```

```
    public boolean hide(Set<? extends GraphicalEditPart> editParts,
           Object layout, PropertyDescriptor propertyDescriptor) {
      boolean hidden = super.hide(editparts, layout, propertyDescriptor);
      if (!hidden) {
        hidden = "allowedTime".equals(propertyDescriptor.getName());
      }
        return hidden;
    }
  }
```

You need to set the custom filter on the graph layout section, which is a similar process to customizing the predefined layout configurations.

The following code sample shows how to override the graph layout control section and set the custom filter.

```
import ilog.views.eclipse.graphlayout.properties.sections.
GraphLayoutPropertySection;

   public class MyGraphLayoutSection extends GraphLayoutPropertySection {

     public MyGraphLayoutPropertySection() {
       setPropertyFilter(new MyPropertyFilter(this));
     }
   }
```

You can also filter groups of properties using an `IGroupFilter` instance. The following code sample shows how to remove the expert options present on various layout property sheets.

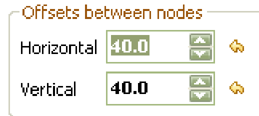
```
import ilog.views.eclipse.graphlayout.properties.sections.IGroupFilter;

import org.eclipse.gef.GraphicalEditPart;

public class MyGroupFilter implements IGroupFilter {

  public boolean hide(Set<? extends GraphicalEditPart> editParts,
              Object layout, String groupName) {
    return "ExpertGroup".equals(groupName);
  }
}
```

As with the property filter, you need to set the group filter on the property section.

```
import ilog.views.eclipse.graphlayout.properties.sections.
GraphLayoutPropertySection;

public class MyGraphLayoutSection extends GraphLayoutPropertySection {

  public MyGraphLayoutPropertySection() {
    setGroupFilter(new MyGroupFilter());
  }
```

Customizing default values

Default values are highlighted in bold in property sheets and can be set using the yellow arrow buttons.



When you use persistence capabilities, default values have more importance because layout properties are initialized with these values. In other words, in this context, a property whose value is not persistent takes this value.

Default values are managed by the class `LayoutDefaultValueProvider`. A default instance is used and this can be extended. By default, default values are described in layout classes and the provider returns them. Label descriptors are an exception: the default value provider looks at the label descriptor configured for a given type of edit part and recognizes its values as default ones.

To configure default values, you need to create a class that derives from `LayoutDefaultValueProvider`. In this context, you can access a registry (the `LayoutDefaultValueRegistry` class) to which you can add your rules.

The following code sample changes the default values of the horizontal and vertical offsets for the hierarchical layout (see the `IlvHierarchicalLayout` class).

```
import ilog.views.eclipse.graphlayout.util.LayoutDefaultValueProvider;
import ilog.views.eclipse.graphlayout.runtime.hierarchical.
IlvHierarchicalLayout;

public class MyDefaultValueProvider extends LayoutDefaultValueProvider {

  public MyDefaultValueProvider() {
    getRegistry().add(IlvHierarchicalLayout.class, “horizontalNodeOffset”,
      70);
    getRegistry().add(IlvHierarchicalLayout.class, “verticalNodeOffset”, 30);

  }

}
```

An index of the properties customizable through the property sheets is available in the Layout Properties Reference.

You also need to indicate that you want to use the new provider instead of the default one.

The following XML example shows how to use the extension point `layoutDefaultValueProvider` to set the new provider.

```
<extension point="ilog.views.eclipse.graphlayout.layoutDefaultValueProvider">

  <provider
    providerClass=
```

```
      "ilog.views.eclipse.graphlayout.util.LayoutDefaultValueProvider">
  </provider>
  <Priority name="High">
  </Priority>
</extension>
```

You can indicate a higher priority to replace the default one in a global way (as shown), or indicate the editor or view ID the provider concerns.