IBM ILOG JViews Graph Layout for Eclipse V8.6

Getting started

© 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

© Copyright IBM Corp. 1987, 2009

Getting started with JViews Graph Layout for Eclipse

Describes how to get started with graph layouts in Eclipse™ Graphical Editing Framework (GEF) and Eclipse Graphical Modeling Framework (GMF) and with label layout.

In this section

Prerequisites
Describes what you need to start work with JViews Graph Layout for Eclipse.

Graphers
Describes the main interface required to use the layout algorithms available in JViews Graph Layout for Eclipse.

Layout source
Describes the interface that allows you to use higher level services, such as GEF commands, to perform layouts or manipulate graphical property sheets.

Starting out with layout sources
Shows you how to start working with JViews Graph Layout for Eclipse by using one of the provided layout sources.

Calling the layout (GEF)
Describes how to perform layout by using a GEF command.

Calling the layout (GMF)
Describes how to use predefined layout providers for GMF applications.

© Copyright IBM Corp. 1987, 2009

Working with label layout
Describes label layout with the use of GMF-based code examples.

Working with subgraphs
Describes how to add layout capabilities to GEF subgraphs and GMF compartments.

Working with link crossings
Describes how to manage the detection and display of link crossings.

Prerequisites

Before you start to work with JViews Graph Layout for Eclipse you need a working GEF or GMF application. The layout algorithms work on a GEF diagram structure and take edit parts as input. If you are not familiar with GEF, see *http://eclipsewiki.editme.com/GefDescription* for an introduction to this framework.

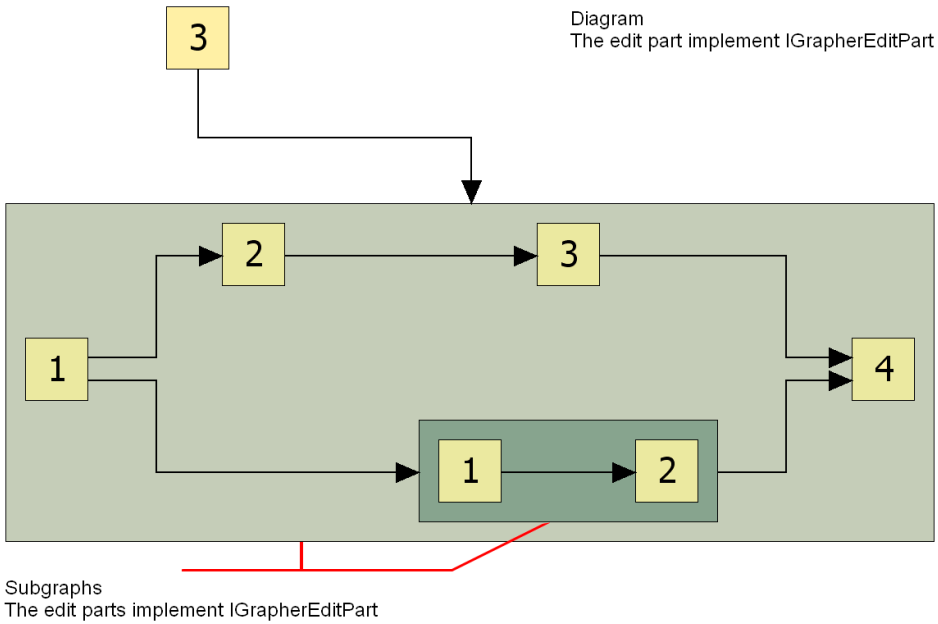
When your application is ready to host layout capabilities, you will need to make your plug-in project dependent on the JViews Graph Layout for Eclipse plug-ins. If you use GEF, you will need to make your plug-in project dependent on the `ilog.views.eclipse.graphlayout` plug-in. If you work with GMF, you will need to make your plug-in project dependent on the `ilog.views.eclipse.graphlayout.gmf` plug-in, which is part of the GMF feature.

Creating these dependencies means that you have in your classpath the API of the layout algorithms as well as the JViews Graph Layout for Eclipse integration facilities. You can now use this API to interface the layouts with your application.

Graphers

The grapher is the edit part of the container of the nodes.

The edit part of the container of your nodes must implement the `IGrapherEditPart` interface so that the layout algorithms can be run on it. This interface must be implemented by the edit part of your diagram or by the edit parts of subgraphs or both if you want diagrams or graphs to be laid out with a graph layout algorithm. To differentiate between them, the method `isTopLevel` must be implemented. It must return true if you implement `IGrapherEditPart` in the frame of the diagram.



Layout source

A higher level abstraction than the grapher; the layout source allows you to make diagrams or subgraphs that show the layouts that are currently set.

The interface `ILayoutSource` can be implemented by `IGrapherEditPart`.

This interface allows you to use higher level services, such as the predefined GEF commands provided in JViews Graph Layout for Eclipse, to perform layouts or manipulate graphical property sheets.

JViews Graph Layout for Eclipse provides default implementations of layout sources for your convenience. (Implementing layout sources can become quite complex.) Some of the default layout sources are designed for use with GEF; others are designed for use with GMF. Adopting a default implementation of a layout source means that most of the code you need to interface your GEF or GMF application with the layouts is already written for you.

The following default implementations are provided:

ilog.views.eclipse.graphlayout.source.LayoutSource
The default `ILayoutSource` for a GEF application, `LayoutSource`. It provides getter and setter methods for choosing the layout you want to perform on the grapher. It manages internally the container layout required to perform several kinds of layouts in parallel in a standard or recursive manner.

ilog.views.eclipse.graphlayout.source.gmf.GMFLayoutSource
The default `ILayoutSource` for a GMF application, `GMFLayoutSource`. It extends the basic LayoutSource to interface better with a GMF application.

ilog.views.eclipse.graphlayout.edit.source.PersistentEMFLayoutSource
Persistent layout sources such as `PersistentEMFLayoutSource` interpret the layout configuration stored in your model. It allows you to have a layout that can be configured at run time in a context that has persistent settings. This layout is designed so that you can easily get layout settings from your Eclipse™ Modeling Framework (EMF) model with some configuration. See Managing persistence .

ilog.views.eclipse.graphlayout.gmf.edit.source.PersistentGMFLayoutSource
Like `PersistentEMFLayoutSource`, `PersistentGMFLayoutSource` retrieves layout configurations from your model. In this case, it retrieves them from the GMF notation model.

Starting out with layout sources

You are recommended to use `LayoutSource` or, if you are working with GMF, `GMFLayoutSource` when you start working with JViews Graph Layout for Eclipse. This will simplify the work you need to do. You can add persistence capabilities later if necessary.

To create a grapher that makes use of these implementations, you must use the `Adapter` pattern, which is in widespread use in Eclipse™ .

The following code example shows you how to instantiate and use a layout source from your grapher in a GEF application.

```
import ilog.views.eclipse.graphlayout.IGrapherEditPart;
import ilog.views.eclipse.graphlayout.source.ILayoutSource;
import ilog.views.eclipse.graphlayout.source.LayoutSource;
import
 ilog.views.eclipse.graphlayout.runtime.hierarchical.IlvHierarchicalLayout;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class MyDiagramEditPart extends AbstractGraphicalEditPart implements
  IGrapherEditPart {

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

  @Override
  public void activate() {
    // Choice to instantiate the layout source when the edit part
    // is activated.
    // Caution: the layout source creation
    // must be done before the edit part is initialized.
    myLayoutSource = new LayoutSource(this);
    // Sets a hierarchical layout for example
    IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
    layout.setGlobalLinkStyle(IlvHierarchicalLayout.ORTHOGONAL_STYLE);
    layout.setFlowDirection(ilog.views.IlvDirection.Right);
    myLayoutSource.setGraphLayout(layout);
    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;
  }

}
```

The path is the same for a GMF layout source.

The code example shows how to set a graph layout, but you could also set link and label layouts.

To set a link layout use:

- An `IlvHierarchicalLayout`, which must have the same flow direction as your intended `IlvHierarchicalLayout` graph layout.
- An `IlvShortLinkLayout`.
- An `IlvLongLinkLayout`

Additional steps are required to use a label layout. See *Working with label layout*.

The diagram layout source must be initialized once the view is populated. To do this, in your editor or in your view, you need to retrieve the diagram layout source and call the `initialize()` method.

The following code example shows how to handle this in a GEF view.

```
import ilog.views.eclipse.graphlayout.source.ILayoutSource;

import org.eclipse.gef.EditPart;
import org.eclipse.gef.GraphicalViewer;
import ort.eclipse.ui.part.viewPart;

import org.eclipse.swt.widgets.Composite;

public class MyView extends ViewPart {

  ...

  public void createPartControl(Composite parent) {

    GraphicalViewer viewer = new ScrollingGraphicalViewer();

    ...
    viewer.setContent(myModel);
    EditPart diagramEditPart = (EditPart)viewer.getContents();
    ILayoutSource layoutSource = (ILayoutSource)diagramEditPart.getAdapter
     (ILayoutSource.class);
    layoutSource.initialize();
  }

}
```

The following code example shows how to handle this in a GMF editor.

```
import ilog.views.eclipse.graphlayout.source.ILayoutSource;

import org.eclipse.gmf.runtime.diagram.ui.resources.editor.parts.
DiagramDocumentEditor;

import org.eclipse.swt.widgets.Composite;

public class MyEditor extends DiagramDocumentEditor {

  ...

  @Override
  public void initializeGraphicalViewerContents(Composite parent) {
    super.initializeGraphicalViewerContents(parent);
    ILayoutSource layoutSource =
      (ILayoutSource)getDiagramGraphicalViewer().getContents().
      getAdapter(ILayoutSource.class);
    layoutSource.initialize();
  }

}
```

Calling the layout (GEF)

If you work with GMF, you should still find this section of interest. What is described is done by the GMF layout provider and can be implemented by a custom GMF layout provider.

A GEF command `PerformLayoutCommand` is provided to perform the layout(s) set on a layout source. A GMF-specific version, `GMFPerformLayoutCommand` is provided, which takes into account the animation preferences set in the preference store by GMF.

You can create a `JFace` action (or a GEF `org.eclipse.gef.ui.actions.WorkbenchPartAction` facility) to execute this command. The following code example shows you how.

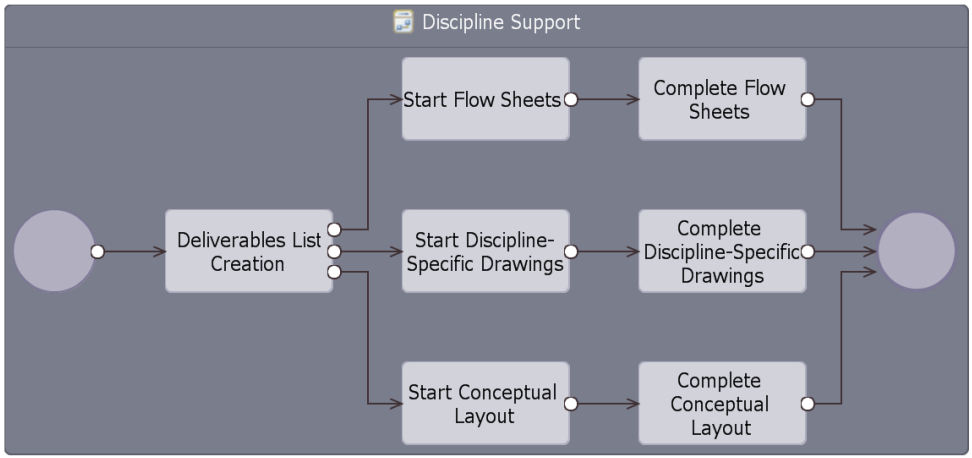
```
import ilog.views.eclipse.graphlayout.IGrapherEditPart;
import ilog.views.eclipse.graphlayout.commands.PerformLayoutCommand;
import org.eclipse.gef.GraphicalViewer;

...
public void run() {
  // get the GEF viewer
  GraphicalViewer viewer = getViewer();
  // get the diagram edit part
  // the edit part must expose a ILayoutSource interface
  IGrapherEditPart editPart = (IGrapherEditPart)viewer.getContent();
  // instantiate the command
  PerformLayoutCommand cmd = new PerformLayoutCommand(editPart);
  // execute it
  viewer.getEditDomain().getCommandStack().execute(cmd);
}
```

You can also instantiate the `PerformLayoutCommand` with a layout type argument.

This argument allows you to choose what type of layout you want to perform:

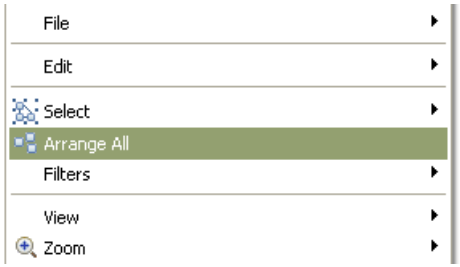
- A graph layout `GRAPH_LAYOUT`
- A link layout `LINK_LAYOUT`
- A label layout `LABEL_LAYOUT`
- All these layouts simultaneously `ALL_LAYOUTS`

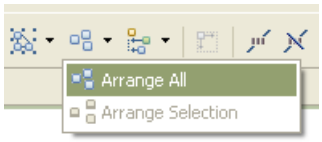
As a result, you should see that the layout runs and obtain something like the following representation.


Discipline Support
Start Flow Sheets
Complete Flow Sheets
Deliverables List Creation
Start Discipline-Specific Drawings
Complete Discipline-Specific Drawings
Start Conceptual Layout
Complete Conceptual Layout

Calling the layout (GMF)

If you are working with GMF you can use a specialized predefined layout provider. The IBM® ILOG® JViews layout will be performed automatically when you try to arrange your diagram in accordance with the GMF **Arrange All** command.





Note: There is very little support in the layout algorithms of JViews Graph Layout for Eclipse for partially laying out graphs. The integration of GMF **Arrange Selection** relies on some configuration tricks and is provided because the layout obtained can be good enough in some cases, but be aware that it can also give bad results.

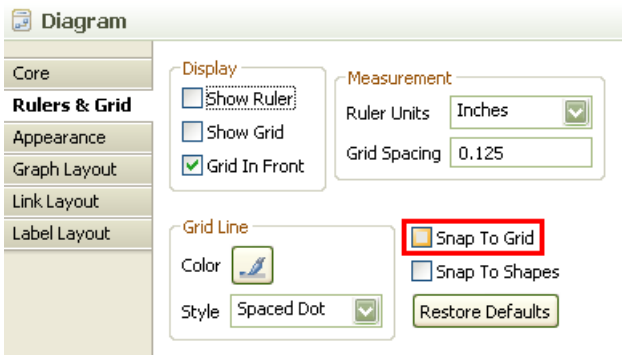
To customize the layout provider, you need to add a new extension by using the GMF `org.eclipse.gmf.runtime.diagram.ui.layoutProviders` extension point. Set `DefaultGMFLayoutProvider` as the layout provider class, with a `High` priority as shown in the following code example.

```
<extension point="org.eclipse.gmf.runtime.diagram.ui.layoutProviders">
  <?gmfgen generated="false"?>
  <layoutProvider
class="ilog.views.eclipse.graphlayout.gmf.providers.DefaultGMFLayoutProvider">

    <Priority name="High">
    </Priority>
  </layoutProvider>
</extension>
```

The default `Arrange` behavior is replaced by the execution of the JViews Graph Layout for Eclipse layout.

Exercise caution with the `Snap To Grid` function. Once layouts are executed, this GMF function can automatically snap the figures to the current grid. In turn, this can lead to small

position deltas that could result in bad visual effects, such as connections that are not orthogonal. You can disable an active `Snap To Grid` function through the GMF property sheet.



The `Snap To Grid` function can be disabled by default in the Preference pages of the editor, so that future diagrams do not use the feature.

Working with label layout

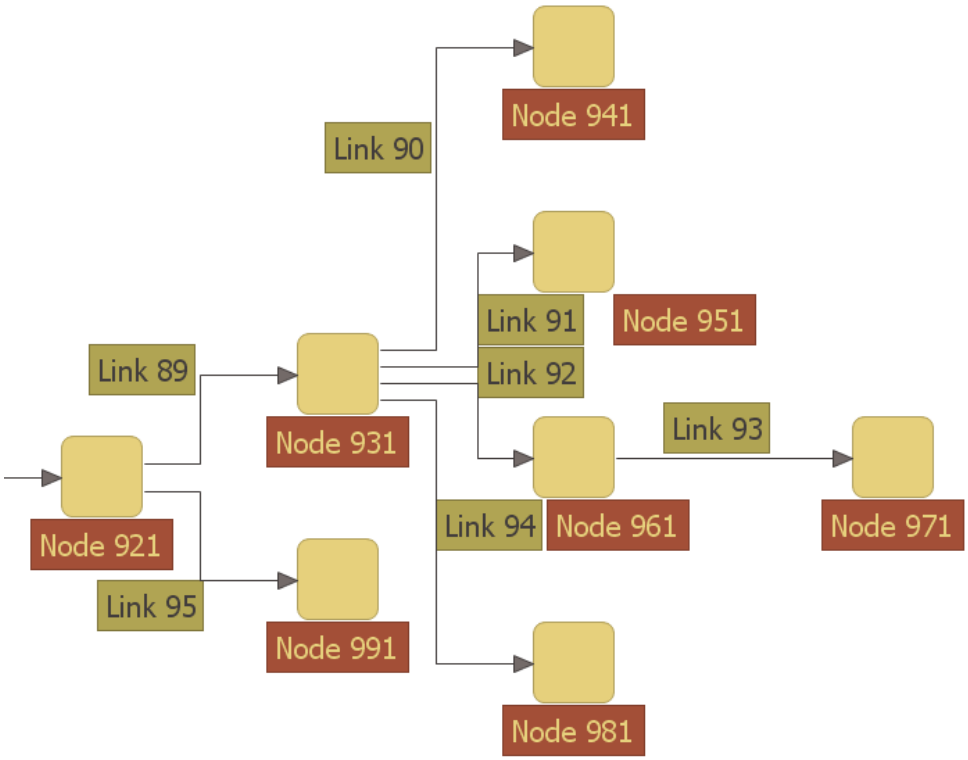
Additional code is required to work with label layout. Layout can be applied to the following kinds of labels:

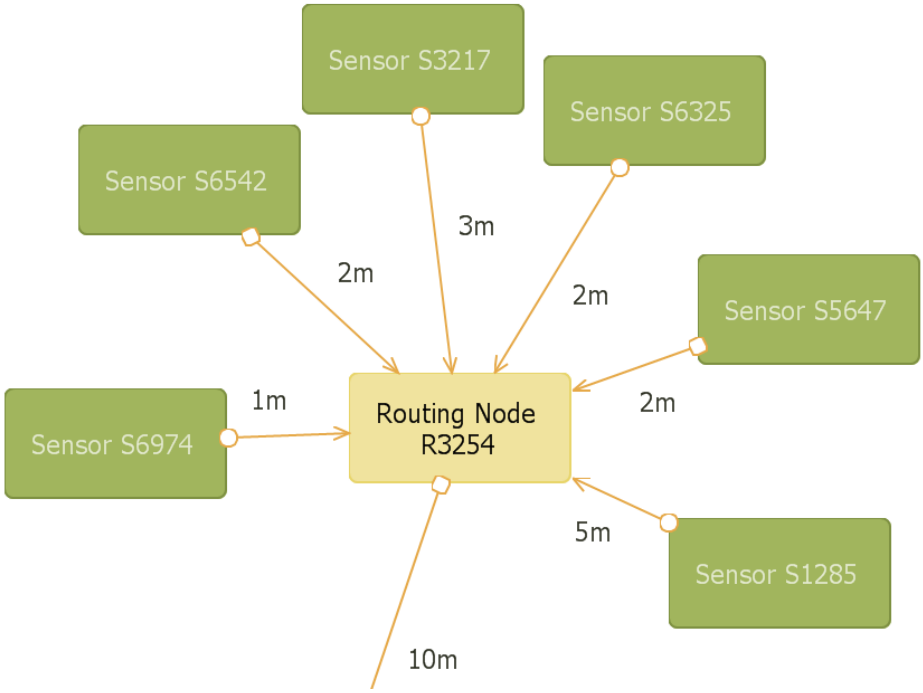
- Labels attached to nodes
- Labels attached to connections

Currently in GMF only labels attached to connections are supported. GMF constrains the way that node labels can be positioned around nodes.

Two label layouts are provided:

- The class `IlvAnnealingLabelLayout`
- The class `IlvRandomLabelLayout`





The way labels can be designed in GEF is somewhat arbitrary, so the code examples in this topic are based on the design used byGMF. You can easily adapt the code for a GEF application.

Obstacle edit parts

You must implement the `IObstacleEditPart` interface to indicate an obstacle in label layout. An obstacle typically consists of node or connection edit parts.

This interface requires you to indicate the labels that the obstacle contains.

The following code example shows the GMF connection implementation of `IObstacleEditPart`. In GMF the edit part of the label is a child of the connection edit part.

```
import ilog.views.eclipse.graphlayout.labellayout.ILabelEditPart;
import ilog.views.eclipse.graphlayout.labellayout.IObstacleEditPart;

import org.eclipse.gmf.runtime.diagram.ui.editparts.ConnectionNodeEditPart;

public class MyConnectionEditPart extends ConnectionNodeEditPart
  implements IObstacleEditPart {

  ...
```

```
  public Collection<ILabelEditPart> getLabels() {
    return Collections.singleton((ILabelEditPart)getChildren().get(0));
  }

}
```

If the obstacle has no label, this method can return null.

Label edit parts

The edit parts that control the labels of your diagram must implement the `ILabelEditPart` interface to be recognized as labels. The interface requires you to indicate the related obstacle, that is, the associated node or connection. It also requires you to implement a factory method to create an instance of a label descriptor.

Label descriptors

Annealing label layout `IlvAnnealingLabelLayout` requires label descriptors to be able to arrange labels around nodes and connections in a preferred position.

There are two kinds of label descriptors:

- `IlvAnnealingPointLabelDescriptor` for node labels
- `IlvAnnealingPolylineLabelDescriptor`

  for connection labels

The following code example shows the connection label implementation of `ILabelEditPart` in the context of GMF, where label edit parts are child objects of connection edit parts.

```
import ilog.views.eclipse.graphlayout.labellayout.ILabelEditPart;
import ilog.views.eclipse.graphlayout.labellayout.IObstacleEditPart;
import ilog.views.eclipse.graphlayout.runtime.labellayout.annealing.
IlvAnnealingLabelDescriptor;
import ilog.views.eclipse.graphlayout.runtime.labellayout.annealing.
IlvAnnealingPolylineLabelDescriptor;

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

public class MyConnectionLabelEditPart extends LabelEditPart implements
  ILabelEditPart {

  ...

  public IObstacleEditPart getRelatedObstacle() {
    return (IObstacleEditPart) getParent();
  }

  public IlvAnnealingLabelDescriptor createLabelDescriptor() {
    // we create a polyline label descriptor because the label concerns a
    // connection
    IlvAnnealingPolylineLabelDescriptor descriptor = new
      IlvAnnealingPolylineLabelDescriptor(
```

```
        this, getRelatedObstacle(),
        IlvAnnealingPolylineLabelDescriptor.CENTER,
        ilog.views.IlvDirection.Left, 0,
        IlvAnnealingPolylineLabelDescriptor.LOCAL);
    // we recommend that you set the auto correct flag, mainly if you use
    // property sheets. It automatically fixes min and max values
    // for a certain property regarding a preferred value
    descriptor.setAutoCorrect(true);
    descriptor.setPreferredDistFromPath(10.f);
    descriptor.setMaxDistFromPath(50.f);
    return descriptor;
  }

}
```

You can now set an annealing label layout `IlvAnnealingLabelLayout` on the layout source. The command to perform layout calls the label layout automatically.

Working with subgraphs

This topic shows you how to add layout capabilities to GEF subgraphs and GMF compartments.

GEF subgraphs

The layout source can be instantiated so that layouts are performed recursively. Thus, subgraphs are laid out before the parent graph, and so on. The default layout source constructor takes the argument `isRecursive`. If you set this field to `true`, the layout source will internally create an `IlvRecursiveMultipleLayout` and manage it. A layout source configured like this can only be instantiated at the diagram level. You cannot have child recursive layout sources.

Note: This behavior does not prevent subgraphs from having their own subgraphs.

The following code example shows how to instantiate a recursive layout source.

```
import ilog.views.eclipse.graphlayout.IGrapherEditPart;
import ilog.views.eclipse.graphlayout.source.ILayoutSource;
import ilog.views.eclipse.graphlayout.source.LayoutSource;
import ilog.views.eclipse.graphlayout.runtime.hierarchical.
IlvHierarchicalLayout;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

public class MyDiagramEditPart extends AbstractGraphicalEditPart
  implements IGrapherEditPart {

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

  ...

  public boolean isTopLevel() {
    return true;
  }

  @Override
  public void activate() {
    myLayoutSource = new LayoutSource(this, true, true);
    // we set a hierarchical layout for example
    IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
    layout.setGlobalLinkStyle(IlvHierarchicalLayout.ORTHOGONAL_STYLE);
    layout.setFlowDirection(ilog.views.IlvDirection.Bottom);
    myLayoutSource.setGraphLayout(layout);
    super.activate();
  }
```

```
}
```

The layout source constructor arguments are:

- Undo and redo support, which allows you to undo or redo the execution of the layout. This support is active by default.
- Recursive mode, which is inactive by default.

Nested graphs also need to implement layout sources. In this case, the layout sources cannot be configured recursively. The following code example shows a subgraph implementation of `ILayoutSource`.

```
import ilog.views.eclipse.graphlayout.IGrapherEditPart;
import ilog.views.eclipse.graphlayout.source.ILayoutSource;
import ilog.views.eclipse.graphlayout.source.LayoutSource;
import ilog.views.eclipse.graphlayout.runtime.hierarchical.
IlvHierarchicalLayout;

import org.eclipse.gef.editparts.AbstractGraphicalEditPart;
import org.eclipse.gef.NodeEditPart;

public class MyNestedGraphEditPart extends AbstractGraphicalEditPart
  implements IGrapherEditPart, NodeEditPart {

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

  ...

  @Override
  public void activate() {
    myLayoutSource = new LayoutSource(this);
    // we set a hierarchical layout for example
    IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
    layout.setGlobalLinkStyle(IlvHierarchicalLayout.ORTHOGONAL_STYLE);
    layout.setFlowDirection(ilog.views.IlvDirection.Right);
    myLayoutSource.setGraphLayout(layout);
    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 false;
  }

  @Override
  protected IFigure createFigure() {
    return new Figure() {
      @Override
      protected boolean useLocalCoordinates() {
        return true;
  }
    };
  }

}
```

The subgraph figure must use local coordinates. In this code example `isTopLevel` returns `false`.

GMF compartments

Compartments can be used for subgraphs in GMF. The GMF Diagram Editor sample is an illustration of how this can be done. Note that compartments are partially supported by JViews Graph Layout for Eclipse; recursive layouts and intergraph connections are not yet supported.

A compartment edit part can be laid out independently. As with a GEF subgraph, your compartment edit part must implement the `ILayoutSource` interface.

```
import ilog.views.eclipse.graphlayout.IGrapherEditPart;
import ilog.views.eclipse.graphlayout.source.ILayoutSource;
import ilog.views.eclipse.graphlayout.gmf.source.GMFLayoutSource;
import ilog.views.eclipse.graphlayout.runtime.IlvHierarchicalLayout;


import org.eclipse.gmf.runtime.diagram.ui.editparts.ShapeCompartmentEditPart;

public class MyShapeCompartmentEditPart extends ShapeCompartmentEditPart
  implements IGrapherEditPart {

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

  ...

  @Override
  public void activate() {
    myLayoutSource = new GMFLayoutSource(this);
    // we set a hierarchical layout for example
    IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
    layout.setGlobalLinkStyle(IlvHierarchicalLayout.ORTHOGONAL_STYLE);
```

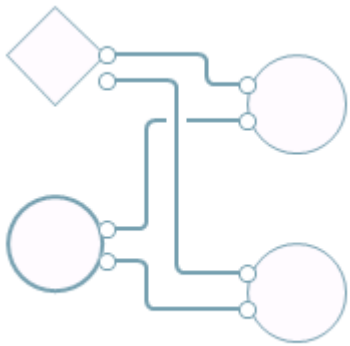
```
    layout.setFlowDirection(ilog.views.IlvDirection.Right);
    myLayoutSource.setGraphLayout(layout);
    super.activate();
  }

  @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 false;
  }

}
```

You need to instantiate a layout source at the diagram level. See *Starting out with layout sources*. The layout source must be a `GMFLayoutSource`.

Working with link crossings

JViews Graph Layout for Eclipse provides options to detect and display link crossings in a graph automatically. Link crossing detection is used when you need to show graphically the points where two links cross. This feature is often used, for example, in electrical schematic diagrams.

The following figure shows a graph in which link crossing detection is enabled.



Enabling and disabling link crossing detection

You will typically enable link crossing detection on the top-level `GraphicalEditPart` that contains your diagram. By default, this will also enable link crossing detection on all nested diagrams.

The edit part of the container of your nodes must implement the `ICrossableContainer` interface so that the crossing detection algorithms can be run on it.

This interface must be implemented by one of the following, depending on where you want to enable link crossing detection:

- The edit part of the whole diagram
- The edit parts of some or all nested diagrams

To differentiate between the whole diagram and nested diagrams, you must implement the method `getCrossableContainerParent()`. This method must return `null` if you are implementing the `CrossableContainer` object for the top-level diagram or the parent for a nested diagram.

The `CrossableContainer` object must create a `CrossingManager` object. The `CrossingManager` object computes the crossings for a set of polyline objects which must implement the `ICrossable` interface.

The `CrossableLink` object is a polyline `IFigure` implementation that implements the `ICrossable` interface.

The following code sample shows how to enable link crossing detection on a whole diagram.

```
import ilog.views.eclipse.crossing.CrossingManager;
import ilog.views.eclipse.crossing.ICrossableContainer;
import ilog.views.eclipse.crossing.LinkCrossings;
import ilog.views.eclipse.crossing.LinkCrossingsEnableMode;
import ilog.views.eclipse.crossing.LinkCrossingsStyle;

public class DiagramEditPart extends AbstractGraphicalEditPart implements
             PropertyChangeListener, IGrapherEditPart, ICrossableContainer {

  private LinkCrossings linkCrossings = new LinkCrossings();
  private CrossingManager crossingManager;

  public DiagramEditPart() {
    linkCrossings.setEnabled(LinkCrossingsEnableMode.Enabled);
    linkCrossings.setCrossingsStyle(LinkCrossingsStyle.Tunnel);
  }

  public void activate() {
    crossingManager = new CrossingManager(linkCrossings, this);
    super.activate();
  }

  public void deactivate() {
    crossingManager.dispose();
    super.deactivate();
  }

  public ICrossableContainer getCrossableContainerParent() {

  }

  public LinkCrossings getLinkCrossings() {
    return linkCrossings;
  }
```

The following code sample shows how to set up crossable links in order to have link crossing.

```
import ilog.views.eclipse.crossing.CrossableLink;

  public class LinkElementEditPart extends AbstractConnectionEditPart {

    protected IFigure createFigure() {
      return new CrossableLink(this);
    }
  }
```

You can choose to enable or disable link crossing detection on specific nested diagrams by setting the `Enabled` property of the `LinkCrossings` object to one of the values of the `LinkCrossingsEnableMode` enumeration.

The possible values of the `LinkCrossingsEnableMode` enumeration are:

LikeParent
This value means that link crossing detection is inherited from the parent container, that is, it is enabled if the `LinkCrossings` object of one of the ancestors of the container has its `Enabled` property set to `Enabled`.

Enabled
This value means that link crossing detection is enabled for this container, regardless of the settings on other containers.

Disabled
This value means that link crossing detection is disabled for this container, regardless of the settings on other containers.

The default value for all graphic containers is `LikeParent`, so if you enable link crossing detection for the top-level container, it is also by default enabled automatically for all nested containers.

Changing the appearance of link crossings

You can choose the way link crossings are displayed in a graphic container by setting the `Style` property of the `LinkCrossings` object of the container. The value of this property is an enumeration of type `LinkCrossingsStyle`.

The possible values of the `LinkCrossingsStyle` enumeration are:

Tunnel
This value means that crossings are drawn as small arcs of circle (bottom left in the figure)

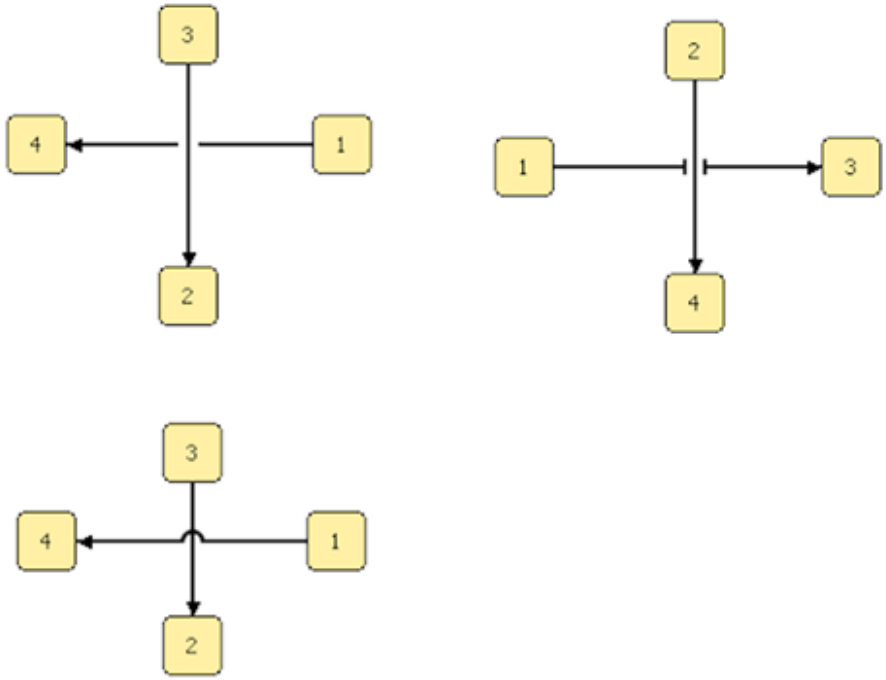
Bridge
This value means that crossings are drawn as two short segments orthogonal to the link path on either side of the crossing (top right in the figure).

Cut
This value means that the path of the link is cut at the crossing (top left in the figure).

None
This value means that no special crossings are drawn, that is, the links cross normally (not shown).

Default
This value means that the crossing style is inherited from the parent container, if any. If there is no parent container, the default crossing style is `Tunnel`.

The following figures shows the possible crossing types.



You can also change the size of the link crossings by setting the `Size` property of the `LinkCrossings` object. The default size is 10.

Instead of changing the appearance of crossings for all the links in a container, you can change the appearance for an individual `Link` object using the `CrossingStyle` and `CrossingSize` properties of the link. These override the values of the `Style` and `Size` properties of the `LinkCrossings` object of the graphic container. However, this is not generally recommended because diagrams look nicer if all link crossings have the same appearance and size.

Changing the orientation of link crossings

When there are many link crossings in a diagram, the display will generally look better if all the crossings have the same orientation. For example, if all the links are made up of horizontal or vertical segments, placing all the crossings on horizontal segments is preferable to having some crossings horizontal and some vertical.

The orientation of link crossings is controlled by the `Orientation` property of the `LinkCrossings` object of the graphic container. The value of this property is an enumeration of type `LinkCrossingsOrientation`.

The possible values of the `LinkCrossingsOrientation` enumeration are:

Horizontal
: This value means that all crossings are placed on horizontal segments. If the segments are not strictly horizontal or vertical, the closest orientation is used.

Vertical
This value means that all crossings are placed on vertical segments. If the segments are not strictly horizontal or vertical, the closest orientation is used.

Any
This value means that crossings are placed on the first segment on which they are detected, regardless of its orientation.