



IBM ILOG JViews Graph Layout for Eclipse V8.6

Using graph layout algorithms

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

| | |
|---|-----------|
| Conventions and Bibliography | 8 |
| Introducing graph layout | 11 |
| The concept of graph layout..... | 12 |
| The graph layout algorithms..... | 14 |
| Structure of the graph layout API..... | 18 |
| Using the graph layout API..... | 20 |
| Releasing resources used during the layout of a grapher..... | 24 |
| Layout algorithms | 25 |
| Overview of graph layout information | 27 |
| Determining the appropriate layout algorithm | 28 |
| Typical ways to choose a layout | 31 |
| Choosing a layout algorithm..... | 32 |
| Choosing the layout algorithm dynamically..... | 33 |
| Hard-coding a layout at run time..... | 34 |
| Generic parameters and features | 35 |
| Support by algorithms of generic features and parameters..... | 36 |
| Base class parameters and features..... | 39 |
| Layout characteristics | 52 |
| Topological Mesh Layout (TML) | 55 |

| | |
|--|------------|
| General information on the TML..... | 56 |
| Features and limitations of the TML..... | 58 |
| The TML algorithm..... | 59 |
| Generic features and parameters of the TML..... | 61 |
| Specific parameters of the TML..... | 64 |
| Refining a graph layout (TML)..... | 70 |
| Using a link clipping interface with the TML..... | 73 |
| Force-directed or Uniform Length Edges Layout (ULEL)..... | 75 |
| General information on the ULEL..... | 76 |
| Features and limitations of the ULEL..... | 81 |
| The ULEL algorithm..... | 82 |
| Generic features and parameters of the ULEL..... | 83 |
| Specific parameters of the ULEL..... | 85 |
| For experts: additional features of the ULEL..... | 88 |
| Using a link clipping interface with the ULEL..... | 91 |
| Tree Layout (TL)..... | 93 |
| General information on the TL..... | 94 |
| Features and limitations of the TL..... | 97 |
| The TL algorithm..... | 99 |
| Generic features and parameters of the TL algorithm..... | 101 |
| Specific parameters (for all tree layout modes)..... | 103 |
| Layout modes of the TL algorithm..... | 107 |
| Free layout mode..... | 109 |
| Level layout mode..... | 127 |
| Radial layout mode..... | 131 |
| Tip-over layout modes..... | 142 |
| Recursive mode..... | 145 |
| For experts: additional tips for the TL..... | 147 |
| Hierarchical Layout (HL)..... | 153 |
| General information on the HL..... | 155 |
| Features and limitations of the HL..... | 159 |
| The HL algorithm..... | 161 |
| Generic features and parameters of the HL..... | 164 |
| Specific parameters of the HL..... | 166 |
| Incremental mode with HL..... | 186 |
| Layout constraints for HL..... | 194 |
| Adding and removing constraints in Java for HL..... | 195 |
| Level range constraints (HL)..... | 197 |
| Level index parameter (HL)..... | 199 |
| Same level constraints (HL)..... | 200 |
| Group spread constraints (HL)..... | 201 |
| Relative level constraints (HL)..... | 202 |
| Position index parameter (HL)..... | 203 |

| | |
|---|------------|
| Relative position constraints (HL)..... | 204 |
| Side-by-side constraints (HL)..... | 205 |
| Extremity constraints (HL)..... | 207 |
| Swim lane constraints (HL)..... | 209 |
| Constraint priorities (HL)..... | 212 |
| For experts: constraint validation (HL)..... | 214 |
| For experts: more indices (HL)..... | 215 |
| Recursive layout..... | 217 |
| Link layout (LL)..... | 219 |
| General information on the LL..... | 221 |
| Features and limitations of the LL..... | 224 |
| The LL algorithms..... | 226 |
| Generic features and parameters of the LL..... | 228 |
| Specific parameters for both LL modes..... | 229 |
| Spacing parameters in short link mode..... | 237 |
| Spacing parameters in long link mode..... | 240 |
| For experts: additional features of LL..... | 243 |
| For experts: special options of the Short LL..... | 246 |
| For experts: special options of the Long LL..... | 255 |
| Random layout (RL)..... | 259 |
| RL sample..... | 260 |
| Features and limitations of the RL..... | 261 |
| The RL algorithm..... | 262 |
| Generic features and parameters of the RL..... | 263 |
| Specific parameters of the RL..... | 265 |
| Bus layout (BL)..... | 267 |
| BL - sample..... | 268 |
| Features of the BL..... | 269 |
| The BL algorithm..... | 270 |
| Generic features and parameters of the BL..... | 274 |
| Specific parameters of the BL..... | 276 |
| Circular layout (CL)..... | 291 |
| General information on the CL..... | 292 |
| Features and limitations of the CL..... | 294 |
| The CL algorithm..... | 295 |
| Generic features and parameters of the CL..... | 299 |
| Specific parameters of the CL..... | 301 |
| Grid layout (GL)..... | 311 |
| General information on the GL..... | 312 |
| Features of the GL..... | 314 |
| The GL algorithm..... | 315 |
| Generic features and parameters of the GL..... | 316 |

| | |
|---|------------|
| Specific parameters of the GL..... | 317 |
| Nested layouts..... | 325 |
| Concepts for nested layouts..... | 326 |
| Layout of nested graphs in code..... | 327 |
| The classes that support nested graphs..... | 328 |
| Order of layouts in recursive layouts..... | 329 |
| Simple recursion: applying the same layout to all subgraphers..... | 330 |
| Advanced recursion: mixing different layouts in a nested graph..... | 334 |
| Recursive layout..... | 337 |
| Overview of recursive layout..... | 338 |
| Features..... | 344 |
| Generic features and parameters..... | 345 |
| Recursive layout modes..... | 347 |
| Overview of recursive layout modes..... | 348 |
| Reference layout mode..... | 349 |
| Internal provider mode..... | 350 |
| Specified provider mode..... | 352 |
| Accessing all sublayouts..... | 353 |
| Specific parameters..... | 355 |
| Listener layout..... | 357 |
| For experts: more on layout providers..... | 358 |
| Multiple layout..... | 361 |
| General information..... | 362 |
| Features..... | 365 |
| Generic features and parameters..... | 366 |
| Specific parameters..... | 368 |
| Attaching graph and labeling models..... | 369 |
| Accessing sublayouts..... | 370 |
| Combining multiple and recursive layout..... | 371 |
| The reference labeling model..... | 372 |
| Automatic label placement..... | 373 |
| Using the label layout API..... | 375 |
| Overview..... | 376 |
| The label layout base class and its subclasses..... | 377 |
| Instantiating and attaching a subclass of <code>IlvLabelLayout</code> | 378 |
| Performing a layout..... | 379 |
| Performing a recursive layout on nested subgraphs..... | 380 |
| The label layout report..... | 382 |
| Layout events and listeners..... | 384 |
| Layout parameters and features in <code>IlvLabelLayout</code> | 386 |

| | |
|---|------------|
| Releasing resources used during the layout of labels..... | 387 |
| Annealing label layout..... | 389 |
| General information..... | 391 |
| Features..... | 392 |
| Limitations..... | 393 |
| The algorithm..... | 394 |
| Generic features and parameters..... | 395 |
| Label descriptors..... | 397 |
| Point label descriptor..... | 398 |
| Polyline label descriptor..... | 403 |
| Specific global parameters..... | 407 |
| For experts: implementing your own label descriptors..... | 411 |
| Using advanced features..... | 415 |
| Overview of advanced features..... | 416 |
| Using a graph layout report..... | 417 |
| Layout report classes..... | 418 |
| Creating a layout report..... | 419 |
| Accessing a layout report..... | 420 |
| Information stored in a layout report..... | 421 |
| Using event listeners..... | 423 |
| Laying out connected components of a disconnected graph..... | 425 |
| Defining your own type of layout..... | 427 |
| A sample custom layout algorithm..... | 428 |
| Implementing the layout method..... | 430 |
| FAQs about using the layout algorithms..... | 432 |
| Index..... | 435 |

Conventions and Bibliography

Conventions

Layout parameter names in the `GraphLayout`, `LinkLayout`, and `LabelLayout` sections always start with a lowercase letter. Layout parameter names in the node or link rules always start with an uppercase letter.

In Java means that you write Java™ code.

Accessors and Modifiers

Very often, you can set and retrieve a property of a class by using a pair of modifier/accessor methods, such as:

```
setFlowDirection(int direction);
int getFlowDirection();
setIncrementalMode(boolean mode);
boolean isIncrementalMode();
```

This document uses the standard Java naming scheme for the modifiers and accessors, that is, the *set* and *get/is* methods. However, when explaining the Java API, it often mentions only the *set* method. Please refer to the For a detailed list of all the *get/is* methods, see the *Java API Reference Documentation* at [index](#).

Books

Several books dedicated to graph layout have been published:

Di Battista, Giuseppe, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, 1999. See:

<http://www.cs.brown.edu/people/rt/gdbook.html>

or

<http://www.mypearsonstore.com/bookstore/product.asp?isbn=0133016153>.

Kaufmann, Wagner (Eds.): *Drawing Graphs*, Lecture Notes in Computer Science Vol. 2025, Springer 2001. See:

<http://link.springer.de/link/service/series/0558/tocs/t2025.htm>.

Graph layout is closely related to graph theory, for which extensive literature exists. See:

Clark, John and Derek Allan Holton. *A First Look at Graph Theory*. World Scientific Publishing Company, 1991.

For a mathematics-oriented introduction to graph theory, see:

Diestel, Reinhard, *Graph Theory*, 2nd ed., Springer-Verlag, 2000.

A more algorithmic approach may be found in:

Gibbons, Alan. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

Gondran, Michel and Michel Minoux. *Graphes et algorithmes*, 3rd ed., Eyrolles, Paris, 1995 (in French).

Bibliography

A comprehensive bibliographic database of papers in computational geometry (including graph layout) can be found at:

The Geometry Literature Database

<http://compgeom.cs.uiuc.edu/~jeffe/compgeom/biblios.html>.

The recommended bibliographic survey paper is the following:

Di Battista, Giuseppe, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. "Algorithms for Drawing Graphs: an Annotated Bibliography." *Computational Geometry: Theory and Applications* 4 (1994): 235-282 (also available at

<http://www.cs.brown.edu/people/rt/gd-biblio.html>.

Journals

The following are electronic journals:

Journal of Graph Algorithms and Applications

<http://jgaa.info/>

Algorithmica

<http://link.springer-ny.com/link/service/journals/00453/>

Computational Geometry: Theory and Applications

<http://www.elsevier.com/locate/comgeo>

Journal of Visual Languages and Computing

<http://www.elsevier.com/locate/jvlc>

The following journals occasionally publish papers on graph layout:

Information Processing Letters

<http://www.elsevier.com/locate/ipl>

Computer-aided Design

<http://www.elsevier.com/locate/cad>

IEEE Transactions on Software Engineering

<http://www.computer.org/tse/>

Many papers are presented at conferences in Combinatorics and Computer Science.

Conferences

An annual Symposium on Graph Drawing has been held since 1992. The proceedings are published by Springer-Verlag in the *Lecture Notes in Computer Science* series.

The 2008 Symposium on Graph Drawing was held in Heraklion, Crete, Greece:

<http://gd2008.org/>

The 2009 Symposium will be held in Chicago, USA.

Introducing graph layout

Describes the IBM® ILOG® JViews graph layout package and its features.

In this section

The concept of graph layout

Provides some background information about graph layout in general, not specifically related to IBM® ILOG® graph layout algorithms.

The graph layout algorithms

Lists the graph layout algorithms available with an example diagram of each.

Structure of the graph layout API

Describes the packages in the graph layout API.

Using the graph layout API

Describes how to apply a graph layout class to a grapher.

Releasing resources used during the layout of a grapher

Describes how to release resources that were created during the layout process.

The concept of graph layout

Simply speaking, a graph is a data structure that represents a set of entities, called nodes, connected by a set of links. A node can also be referred to as a vertex. A link can also be referred to as an edge or a connection. In practical applications, graphs are frequently used to model a very wide range of things: computer networks, software program structures, project management diagrams, and so on. Graphs are powerful models because they permit applications to benefit from the results of graph theory research. For instance, efficient methods are available for finding the shortest path between two nodes, the minimum cost path, and so on.

Layout of a graph

Graph layout is used in graphical user interfaces of applications that need to display graph models. To lay out a graph means to draw the graph so that an appropriate, readable representation is produced. Essentially, this involves determining the location of the nodes and the shape of the links. For some applications, the location of the nodes may already be known (for example, based on the geographical positions of the nodes). However, for other applications, the location is not known (a pure “logical” graph) or the known location, if used, would produce an unreadable drawing of the graph. In these cases, the location of the nodes must be computed.

What is meant by an “appropriate” drawing of a graph? In practical applications, it is often necessary for the graph drawing to observe certain quality criteria. These criteria may vary depending on the application field or on a given standard of representation. It is often difficult to tell what a good layout consists of. Each end user may have different, subjective criteria for qualifying a layout as “good”. However, one common goal exists behind all the criteria and standards: the drawing must be easy to understand and provide easy navigation through the complex structure of the graph.

What is a good layout?

To deal with the various needs of different applications, many classes of graph layout algorithms have been developed. A layout algorithm addresses one or more quality criteria, depending on the type of graph and the features of the algorithm, when laying out a graph.

The most common criteria are:

- ◆ Minimizing the number of link crossings
- ◆ Minimizing the total **area** of the drawing
- ◆ Minimizing the number of **bends** (in orthogonal drawings)
- ◆ Maximizing the smallest **angle** formed by consecutive incident links
- ◆ Maximizing the display of **symmetries**

How can a layout algorithm meet each of these quality criteria and standards of representation? If you look at each individual criteria, some can be met quite easily, at least for some classes of graphs. For other classes, it may be quite difficult to produce a drawing that meets the criteria. For example, minimizing the number of link crossings is relatively simple for trees (that is, graphs without cycles). However, for general graphs, minimizing the number of link crossings is a mathematical NP-complete problem (that is, with all known

algorithms, the time required to perform the layout grows very fast with the size of the graph).

Moreover, if you want to meet several criteria at the same time, an optimal solution may not exist with respect to each individual criteria because many of the criteria are mutually contradictory. Time-consuming trade-offs may be necessary. In addition, it is not a trivial task to assign weights to each criteria. Multicriteria optimization is, in most cases, too complex to implement and much too time-consuming. For these reasons, layout algorithms are often based on heuristics and may provide less than optimal solutions with respect to one or more of the criteria. Fortunately, in practical terms, the layout algorithms will still often provide reasonably readable drawings.

Methods for using layout algorithms

Layout algorithms can be employed in a variety of ways in the various applications in which they are used. The most common ways of using an algorithm are the following:

◆ *Automatic layout*

The layout algorithm does everything without any user intervention, except for perhaps the choice of the layout algorithm to be used. Sometimes, a set of rules can be coded to choose automatically (and dynamically) the most appropriate layout algorithm for the particular type of graph being laid out.

◆ *Semiautomatic layout*

The end user is free to improve the result of the automatic layout procedure by hand. In some cases, the end user can move and “pin” nodes at desired locations and perform the layout again. In other cases, a part of the graph is automatically set as “read-only” and the end user can modify the rest of the layout.

◆ **Static layout**

The layout algorithm is completely redone (“from scratch”) each time the graph is changed.

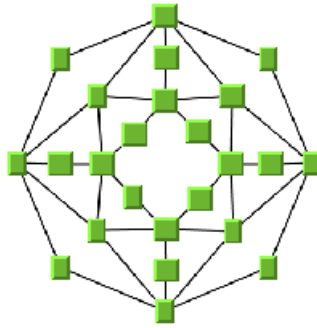
◆ **Incremental layout**

When the layout algorithm is performed a second time on a modified graph, it tries to preserve the stability of the layout as much as possible. The layout is not performed again from scratch. The layout algorithm also tries to save CPU time by using the previous layout as an initial solution. Some layout algorithms and layout styles are incremental by nature. For others, incremental layout may be impossible.

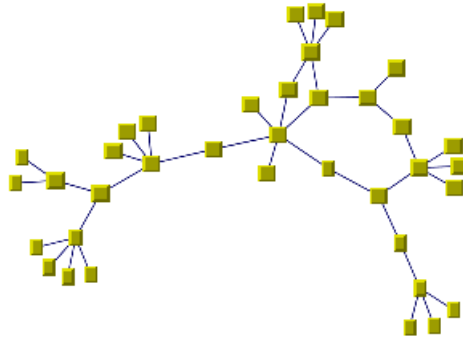
The graph layout algorithms

The graph layout package provides numerous ready-to-use layout algorithms. They are shown below with sample illustrations. In addition, you can develop new layout algorithms using the generic layout framework.

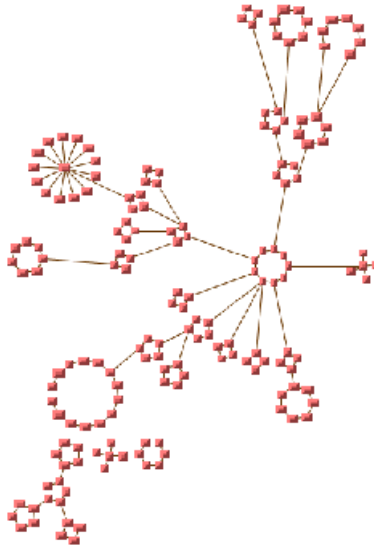
*Topological
Mesh Layout
(TML)*



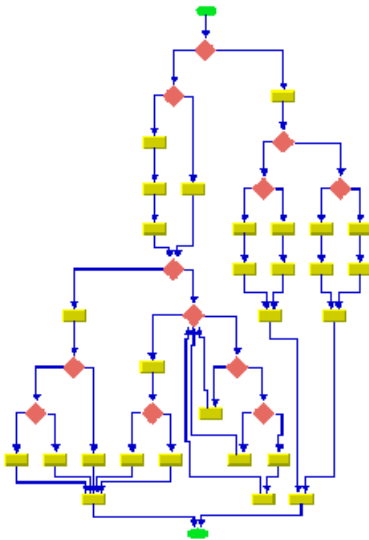
*Force-directed
or Uniform
Length Edges
Layout (ULEL)*



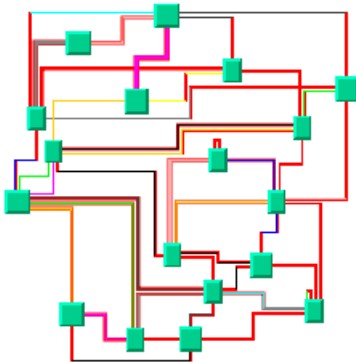
*Circular layout
(CL)
(Ring/Star)*



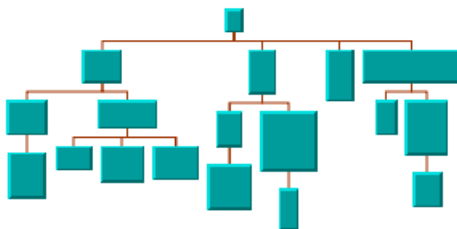
*Hierarchical
Layout (HL)*



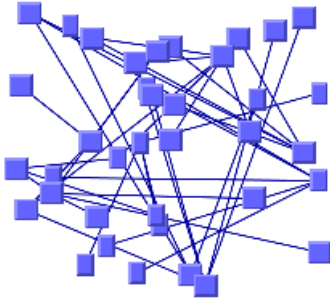
*Link layout
(LL)*



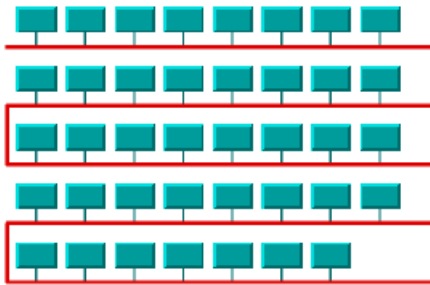
*Tree Layout
(TL)*



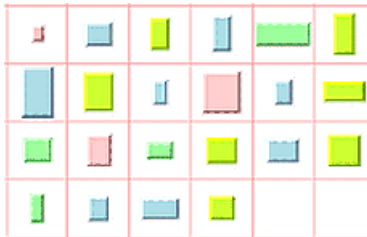
*Random layout
(RL)*



*Bus layout
(BL)*



*Grid layout
(GL)*



Structure of the graph layout API

The IBM® ILOG® JViews graph layout API is composed of:

- ◆ *The generic graph layout package*
- ◆ *The layout algorithm packages*
- ◆ *The label layout package*

The generic graph layout package

`ilog.views.graphlayout`: A high-level, generic framework for the graph layout services provided by IBM® ILOG® JViews.

The layout algorithm packages

- ◆ `ilog.views.graphlayout.bus`: A layout algorithm designed to display bus network topologies (that is, a set of nodes connected to a bus node).
- ◆ `ilog.views.graphlayout.circular`: A layout algorithm that displays graphs representing interconnected ring and/or star network topologies.
- ◆ `ilog.views.graphlayout.grid`: A layout algorithm that arranges the disconnected nodes of a graph in rows, in columns, or in the cells of a grid.
- ◆ `ilog.views.graphlayout.hierarchical`: A layout algorithm that arranges nodes in horizontal or vertical levels such that the links flow in a uniform direction.
- ◆ `ilog.views.graphlayout.link`: A layout algorithm that reshapes the links of a graph without moving the nodes.
 - `ilog.views.graphlayout.link.longlink`: For long orthogonal links.
 - `ilog.views.graphlayout.link.shortlink`: For short links.
- ◆ `ilog.views.graphlayout.multiple`: A facility that combines multiple layout algorithms and treat them as one algorithm object.
- ◆ `ilog.views.graphlayout.random`: A layout algorithm that moves the nodes of the graph at randomly computed positions inside an user-defined region.
- ◆ `ilog.views.graphlayout.recursive`: A layout algorithm that can be used to control the layout of nested graphs (containing subgraphs and intergraph links).
- ◆ `ilog.views.graphlayout.topologicalmesh`: A layout algorithm that can be used to lay out cyclic graphs.
- ◆ `ilog.views.graphlayout.tree`: A layout algorithm that arranges the nodes of a tree horizontally or vertically, starting from the root of the tree. A radial layout mode allows you to arrange the nodes of a tree on concentric circles around the root of the tree.

- ◆ `ilog.views.graphlayout.uniformlengthedges`: A layout algorithm that can be used to lay out any type of graph and allows you to specify the length of the links.

The label layout package

`ilog.views.graphlayout.labellayout`: A layout algorithm for automatic placement of labels.

- ◆ `ilog.views.graphlayout.labellayout.annealing`: For close label positioning.
- ◆ `ilog.views.graphlayout.labellayout.random`: For random placement.

Using the graph layout API

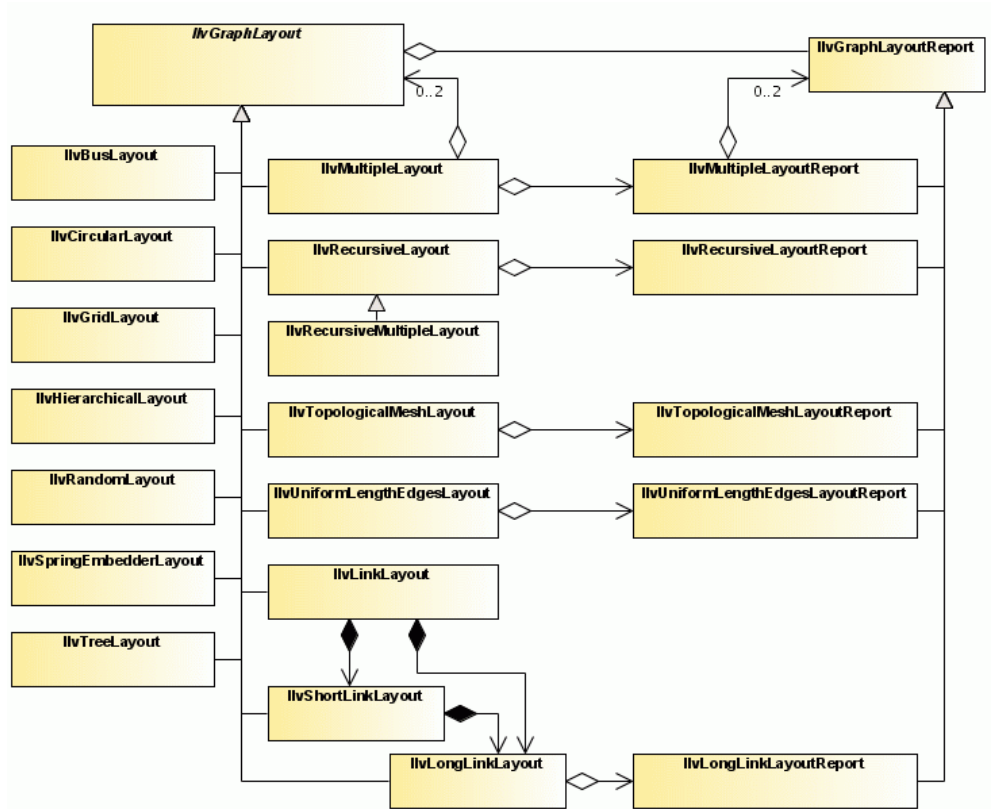
In an application that works directly on graphers (`IGrapherEditPart`) without using the `ILayoutSource` facility, operations such as attaching or detaching a graph layout instance must be performed explicitly.

The base class: `IlvGraphLayout`

The `IlvGraphLayout` class is the base class for all layout algorithms. This class is an abstract class and cannot be used directly. You must use one of its subclasses:

`IlvHierarchicalLayout`, `IlvTreeLayout`, `IlvUniformLengthEdgesLayout`,
`IlvTopologicalMeshLayout`, `IlvLinkLayout`, `IlvRandomLayout`, `IlvBusLayout`,
`IlvCircularLayout`, `IlvGridLayout`. You can also create your own subclasses to implement other layout algorithms. See *Defining your own type of layout*.

Despite the fact that only subclasses of `IlvGraphLayout` are directly used to obtain the layouts, it is still necessary to learn about this class because it contains methods that are inherited (or overridden) by the subclasses. And, of course, you will need to understand it if you subclass it yourself.



The Class *IlvGraphLayout* and its subclasses and relationships to layout reports

Instantiating a subclass of *IlvGraphLayout*

The class *IlvGraphLayout* is an abstract class. It has no constructors. You will instantiate a subclass as shown in the following example:

```
IlvLinkLayout layout = new IlvLinkLayout();
```

Attaching/detaching a grapher

You must attach the grapher before performing the layout. The attachment is done through a *GraphModel*, which is a graph abstraction manipulated by the layout algorithms to lay out graphs. The following method, defined on the class *IlvGraphLayout*, allows you to specify the grapher you want to lay out:

```
void attach(IlvGraphModel graphModel)
```

For example:

```
import ilog.views.eclipse.graphlayout.GraphModel;
...
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);
```

The `attach` method does nothing if the specified grapher is already attached. If a different grapher is attached, this method first detaches this old grapher, then attaches the new one. The attached graph model can be obtained by:

```
IlvGraphModel graphModel = layout.getGraphModel();
```

After layout, when you no longer need the layout instance, you should call the method

```
void detach()
```

If the `detach` method is not called, some objects may not be garbage-collected. It also removes layout parameters of nodes and links.

Note: A layout instance should stay attached as long as its layout parameters are relevant for the grapher. Only when the layout parameters, and therefore the entire layout instance, become irrelevant for this grapher should it be detached.

Performing a layout

The `performLayout` method starts the layout algorithm using the currently attached grapher and the current settings for the layout parameters. The method returns a report object that contains information about the behavior of the layout algorithm.

```
IlvGraphLayoutReport performLayout()
```

```
IlvGraphLayoutReport performLayout(boolean force, boolean redraw)
```

The first version of the method simply calls the second one with a `false` value for the first argument and a `true` value for the second argument. If the argument `force` is `false`, the layout algorithm first verifies whether it is necessary to perform the layout. It checks internal flags to see whether the grapher or any of the parameters have been changed since the last time the layout was successfully performed. A “change” can be any of the following:

- ◆ Nodes or links have been added or removed.
- ◆ Nodes or links have been moved or reshaped.

- ◆ The value of a layout parameter has been modified.

Users often do not want the layout to be computed again if no changes occurred. If there were no changes, the method `performLayout` returns without performing the layout. Note that if the argument `force` is passed as `true`, the verification is no longer performed.

The argument `redraw` is ignored by IBM® ILOG® JViews JViews Graph Layout for Eclipse.

The protected abstract method `layout(boolean redraw)` is then called. This means that the control is passed to the subclasses that are implementing this method. The implementation computes the layout and moves the nodes to new positions and/or reshapes the links.

The `performLayout` method returns an instance of `IlvGraphLayoutReport` (or of a subclass) that contains information about the behavior of the layout algorithm. It tells you whether the algorithm performed normally, or whether a particular, predefined case occurred. (For a more detailed description of the layout report, see *Using a graph layout report*.)

Note that the layout report that is returned can be an instance of a subclass of `IlvGraphLayoutReport` depending on the particular subclass of `IlvGraphLayout` you are using. For example, it will be an instance of `IlvTopologicalMeshLayoutReport` if you are using the class `IlvTopologicalMeshLayout`. Subclasses of `IlvGraphLayoutReport` are used to store layout algorithm-dependent information.

You must call the method `performLayout` inside a `try` block because it can throw an exception. The exception can be of the type `IlvGraphLayoutException` or `IlvInappropriateGraphException`. The first indicates internal problems in the layout algorithm or an unexpected situation. The second exception indicates that a particular grapher cannot be laid out with the layout algorithm. For example, the Topological Mesh Layout cannot be used on a tree). See *Layout exceptions for details and solutions*.

Further information

You can find more information about the class `IlvGraphLayout` in the following sections:

- ◆ *Base class parameters and features* contains the methods that are related to the customization of the layout algorithms.
- ◆ *Using event listeners* tells you about the layout event listener mechanism.
- ◆ *Defining your own type of layout* tells you how to implement new subclasses.

For details on `IlvGraphLayout` and other graph layout classes, see the Java™ *API Reference Documentation*.

Releasing resources used during the layout of a grapher

Various objects need to be created during the layout process. Most commonly, these are:

- ◆ Layout instances (subclasses of `IlvGraphLayout`)
- ◆ Other adapters (`GraphModel`)
- ◆ Layout providers.

For recursive layout, you may also instantiate layout providers (subclasses of `IlvDefaultLayoutProvider`). See also *Recursive layout*.

- ◆ Property objects

Some of the layout parameters are internally stored as property objects attached to the grapher object or to its nodes and links.

Rules for releasing resources

If you program graph layout directly in Java, you must respect some rules to ensure that all these allocated objects are correctly released:

1. When a layout instance instantiated by your code is no longer useful, call the method `detach()` on it to ensure that no grapher or graph model is still attached to it. Note that you can freely reuse a layout instance once the previously attached model has been detached.
2. Layout parameters that are specific to a node or a link are cleaned when calling `IlvGraphLayout.detach()`. This cleaning is done only for nodes and links that are still in the grapher when the `detach()` method is called. If per-node or per-link parameters have been specified and the node or the link needs to be removed before the `detach()` method can be called, you can call the methods `cleanNode` or `cleanLink` of the class `IlvGraphLayout` to perform the cleaning for the node or the link. However, you only need to do so if the removed node or link is reused by your code after removal. Otherwise, if your code does not keep any reference to it, the node or link will be garbage collected anyway, together with the property objects eventually stored by the layout.
3. When a graph model instantiated by your code is no longer useful, call the method `dispose()` on it to ensure that the resources it has used are released. Note that a graph model must not be used once it has been disposed.
4. When a layout provider (an instance of `IlvDefaultLayoutProvider`) instantiated by your code is no longer useful, call the method `detachLayouts(model, true)` on it, passing as arguments the graph models that have been used for performing a recursive layout with this provider.

Layout algorithms

Describes the IBM® ILOG® JViews Graph Layout algorithms.

In this section

Overview of graph layout information

Describes the information given for each graph layout algorithm.

Determining the appropriate layout algorithm

Explains how to determine which graph layout is appropriate.

Typical ways to choose a layout

Explains possible ways to choose a graph layout algorithm.

Generic parameters and features

Describes the support for generic features and parameters provided by each layout algorithm.

Layout characteristics

Describes the effect of settings on each layout algorithm.

Topological Mesh Layout (TML)

Gives information on the *Topological Mesh Layout (TML)* algorithm (class `IlvTopologicalMeshLayout` from the package `ilog.views.graphlayout.topologicalmesh`).

Force-directed or Uniform Length Edges Layout (ULEL)

Describes the *Force-directed layout* or *Uniform Length Edges Layout* algorithm (class `IlvUniformLengthEdgesLayout` from the package `ilog.views.graphlayout.uniformlengthedges`).

Tree Layout (TL)

Describes the *Tree Layout* algorithm (class `IlvTreeLayout` from the package `ilog.views.graphlayout.tree`).

Hierarchical Layout (HL)

Describes the *Hierarchical Layout* algorithm (class `IlvHierarchicalLayout` from the package `ilog.views.graphlayout.hierarchical`).

Link layout (LL)

Describes the *Link Layout* algorithm (class `IlvLinkLayout` from the package `ilog.views.graphlayout.link`).

Random layout (RL)

Describes the *Random Layout* algorithm (class `IlvRandomLayout` from the package `ilog.views.graphlayout.random`).

Bus layout (BL)

Describes the *Bus Layout* algorithm (class `IlvBusLayout` from the package `ilog.views.graphlayout.bus`).

Circular layout (CL)

Describes the *Circular Layout* algorithm (class `IlvCircularLayout` from the package `ilog.views.graphlayout.circular`).

Grid layout (GL)

Describes the *Grid Layout* algorithm (class `IlvGridLayout` from the package `ilog.views.graphlayout.grid`).

Overview of graph layout information

For each layout, the information given includes:

- ◆ Code samples
- ◆ Which types of graphs the layout may be used for
- ◆ The application domains, features, and limitations
- ◆ A brief description of the algorithm
- ◆ The specification
- ◆ The generic features and parameters, as well as the specific parameters of the algorithm

Determining the appropriate layout algorithm

When using the graph layout package, you need to determine which of the ready-to-use layout algorithms is appropriate for your particular needs. Some layout algorithms can handle a wide range of graphs. Others are designed for particular classes of graphs and will give poor results or will reject graphs that do not belong to these classes. For example, a Tree Layout algorithm is designed for tree graphs, but not cyclic graphs. Therefore, it is important to lay out a graph using the appropriate layout algorithm.

The following tables can help you determine which of the layout algorithms is best suited for a particular type of graph.

- ◆ Across the top of the table are various classifications of different types of graphs.
- ◆ The layout algorithms appear on the left side of the tables.
- ◆ Table cells containing illustrations indicate when a layout algorithm is applicable for a particular type of graph.

By identifying the general characteristics of the graph you want to lay out, you can see from the tables whether a layout algorithm is suited for that particular type of graph.

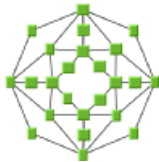





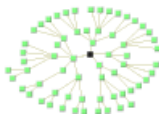



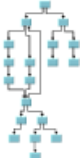
For example, if you know that the structure of the graph is a tree, you can look at the Domain-Independent Graphs/Trees column to see which layout algorithms are appropriate. The Uniform Length Edges Layout, Tree Layout, and Hierarchical Layout could all be used. Use the illustrations in the table cells to help you further narrow your choice.



You can use the *Recursive layout* to control the layout of nested graphs (containing subgraphs and intergraph links). This is in particular useful if different layout styles should be applied to different subgraphs. The Recursive Layout allows you to specify which layout is used for which subgraph, and it traverses the entire nested graph recursively when applying the layout. As a result, the entire nested graph is laid out.

You can use the *Multiple layout* to combine several different layouts into one instance. In this case, they become *sublayouts* of the Multiple Layout instance.

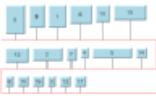

This is useful in particular for nested graphs when used in combination with the Recursive Layout. The Multiple Layout ensures that the normal layout, the routing of the intergraph links, and the layout of labels are applied in the correct order to a nested graph.

Layout algorithms and common types of graphs

| Layout | Domain-Independent Graphs | | |
|-----------------------------|---|--|---|
| | Trees | Cyclic Graphs | Any Graph |
| Topological Mesh Layout | |  |  Requires (semi)manual refinements |
| Uniform Length Edges Layout |  |  Preferable to avoid heavily interconnected graphs (large number of links) |  |
| Tree Layout |   | |  |
| Hierarchical Layout |  |  |  |

| Layout | Domain-Independent Graphs | | |
|------------------|---------------------------|---------------|---|
| | Trees | Cyclic Graphs | Any Graph |
| Link Layout | | |  |
| Grid Layout | | |  <p>Note that the algorithm does not take into account the links between the nodes.</p> |
| Recursive Layout | | | Nested graphs. |
| Multiple Layout | | | Combination of multiple different layout algorithms on the same graph (in particular for nested graphs). |

Telecom-Oriented Representations

| Layout | Telecom-Oriented Representations |
|-----------------|--|
| Bus Layout |  <p>For bus topologies</p> |
| Circular Layout |  <p>For interconnected ring/star topologies</p> |

Typical ways to choose a layout

Explains possible ways to choose a graph layout algorithm.

In this section

Choosing a layout algorithm

Explains the difference between automatic and semiautomatic layout selection.

Choosing the layout algorithm dynamically

Explains how to choose a layout algorithm automatically at run time.

Hard-coding a layout at run time

Explains how to choose a layout at run time.

Choosing a layout algorithm

The choice of the appropriate algorithm for a graph can be done either by the end user at run time or by the programmer when he develops the application. This process can be *semiautomatic*, when the user is involved, or *automatic*, when the application does everything with no user intervention.

As a programmer of applications, you can choose *Semiautomatic layout* to involve the end user in the choice of the layout, or *Automatic layout*, in which case the application does everything with no end user action.

Semiautomatic layout

For applications using a semiautomatic layout, the choice of the layout algorithm is done by the end user. The application can provide a menu or some other way to select the layout algorithm.

In some cases, this may be an iterative process. The user may try different layout algorithms with different values for the parameters and/or may apply manual refinements to find the best layout. The application may possibly provide some help using textual explanations or by automatically checking the graph to find out to which class it belongs. For example, to detect whether the graph that has been attached to a layout instance is a tree, the `IlvGraphLayoutUtil` class provides the method:

```
static boolean IsTree(IlvGraphLayout layout, Object startNode)
```

For details on this method, see `IsTree (ilog.views.graphlayout.IlvGraphLayout, java.lang.Object)`. See also *Attaching/detaching a grapher*.

Automatic layout

If an automatic layout is needed, the choice of the layout algorithm can be:

- ◆ Chosen dynamically at run time by means of heuristics or rules to determine the appropriate layout algorithm depending on the structure and/or size of the graph
- ◆ Hard-coded if the developer knows what types of graphs will be used and can determine the appropriate layout algorithm.

Choosing the layout algorithm dynamically

If nothing is known about the graphs that the application will need to lay out, the developer can write a routine that automatically chooses the layout algorithm at run time. The following simple rules could be applied:

1. If the nodes of the graph cannot be moved (they are geo-positioned), use the Link Layout.
2. If the graph is a tree, use the Tree Layout.
3. Otherwise, use one of the layout algorithms that are the less restricted to a given graph category, especially the Uniform Length Edges Layout. (The preferred length of the links could also be computed with respect to the size of the nodes.)
4. If the graph is too large, apply a “divide-and-conquer” strategy. Cut the graph into several subgraphs and apply the layout separately to each subgraph. If the graph is disconnected, you can use the built-in support provided by the layout library to perform this task automatically. (See *Layout of connected components*.)
5. If the graph is nested, use the Recursive Layout algorithm that controls which subgraph is laid out by which (flat) sublayout. Use steps 1 to 4 to determine the sublayouts for the subgraphs. The Hierarchical Layout and the Tree Layout also have special modes for nested graphs, see *Recursive mode* and *Recursive layout*.

Hard-coding a layout at run time

If the choice of the layout algorithm is hard-coded, but the layout must be performed at run time because the graphs are not known at programming time, one possible step-by-step procedure for the choice of the appropriate layout algorithm may be the following:

1. Look at sample graphs for your domain.
2. Try to determine some generalities about the properties of the structure and the size of the graph (Is the graph cyclic? Is the graph a tree? Is the graph a combination of the two? What is the number of nodes and links in the graph?)
3. Pick an appropriate layout algorithm.
4. Try out the algorithm on one or more samples.

See also

Determining the appropriate layout algorithm

Generic parameters and features

Describes the support for generic features and parameters provided by each layout algorithm.

In this section

Support by algorithms of generic features and parameters

Describes the support for generic features and parameters provided by each layout algorithm.

Base class parameters and features

Describes the generic features and parameters for customizing graph layout algorithms.

Support by algorithms of generic features and parameters

The following table indicates the generic features and parameters that are supported by each layout algorithm. These parameters are defined in the base class for all layout algorithms, `IlvGraphLayout`

Generic parameters supported by layout algorithms

| Layout Algorithm Parameters | TML | ULEL | TL | HL | LL | RL | BL | CL | GL | Recursive Layout | Multiple Layout |
|--------------------------------|-----|------|-----|-----|-----|-----|-----|-----|-----|------------------|-----------------|
| Allowed Time | Yes | Yes | Yes | Yes | Yes | Yes | Yes | | Yes | Yes | Yes |
| Fixed Links | | | Yes | Yes | Yes | | | | | | |
| Fixed Nodes | Yes | Yes | Yes | Yes | | Yes | Yes | Yes | Yes | | |
| Layout of Connected Components | Yes | Yes | Yes | Yes | | | Yes | Yes | | | Yes |
| Layout Region | Yes | Yes | | | | Yes | Yes | Yes | Yes | | |
| Link Clipping | Yes | Yes | Yes | Yes | | | Yes | Yes | | | |
| Link Connection Box | Yes | Yes | Yes | Yes | Yes | | Yes | Yes | | | |
| Memory Savings | | | | | | | | | | | |
| Percentage Complete | | Yes | | Yes | Yes | | Yes | | | Yes | Yes |
| Random Generator Seed Value | | | | | | Yes | | | | | |
| Stop Immediately | Yes | Yes | Yes | Yes | Yes | Yes | Yes | | Yes | Yes | Yes |

Key

TML Topological Mesh Layout

ULEL Uniform Length Edges Layout

TL Tree Layout

HL Hierarchical Layout

LL Link Layout

RL Random Layout

BL Bus Layout

CL Circular Layout

GL Grid Layout

Base class parameters and features

The `IlvGraphLayout` class defines a number of generic features and parameters. These features and parameters can be used to customize the layout algorithms.

Although the `IlvGraphLayout` class defines the generic parameters, it does not control how they are used by its subclasses. Each layout algorithm (that is, each subclass of `IlvGraphLayout`) supports a subset of the generic features and determines the way in which it uses the generic parameters. When you create your own layout algorithm by subclassing `IlvGraphLayout`, you decide whether you want to use the features and the way in which you are going to use them.

The `IlvGraphLayout` class defines the following generic features:

- ◆ *Allowed time*
- ◆ *Automatic layout*
- ◆ *Layout of connected components*
- ◆ *Layout region*
- ◆ *Link clipping*
- ◆ *Link connection box*
- ◆ *Memory savings*
- ◆ *Percentage of completion calculation*
- ◆ *Preserve fixed links*
- ◆ *Preserve fixed nodes*
- ◆ *Random generator seed value*
- ◆ *Stop immediately*
- ◆ *Use default parameters*

Support by algorithms of generic features and parameters provides a summary of the generic parameters supported by each layout algorithm. If you are using one of the subclasses provided with IBM® ILOG® JViews, check the documentation for that subclass to know whether it supports a given parameter and whether it interprets the parameter in a particular way.

Allowed time

Several layout algorithms can be designed to stop computation when a user-defined time specification is exceeded. This may be done for different reasons: as a security measure to avoid a long computation time on very large graphs or as an upper limit for algorithms that iteratively improve a current solution and have no other criteria to stop the computation.

Example of specifying allowed time

To specify that the layout is allowed to run for 60 seconds:

In Java™

Call:

```
layout.setAllowedTime(60000)
```

The time is in milliseconds. The default value is 32000 (32 seconds).

If you subclass `IlvGraphLayout`, use the following method to know whether the specified time was exceeded:

```
boolean isLayoutTimeElapsed()
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsAllowedTime()
```

The default implementation returns `false`. A subclass can override this method to return `true` to indicate that this mechanism is supported.

Automatic layout

For some layout algorithms, it may be suitable to have the layout automatically performed again after each change of the graph, that is, when a node or link moves, is added, or is removed. Automatic layout is most useful for link layouts, in a situation where the shape of the links must remain optimal after each editing action of the end-user. It also works well with other layout algorithms that offer an incremental behavior, that is, for which a small change of the graph usually produces only a small change of the layout. Automatic layout is generally not suitable for non-incremental layout algorithms.

Example of automatic layout

To enable automatic layout:

In Java

Call:

```
layout.setAutoLayout(true);
```

For more information about automatic layout, see the method `performAutoLayout()` in the *Java API Reference Documentation*.

Layout of connected components

The base class `IlvGraphLayout` provides generic support for the layout of a disconnected graph (composed of connected components). For details, see *Laying out connected components of a disconnected graph*.

Example of layout

To enable the placement of disconnected graphs:

In Java

Call:


```
setLayoutOfConnectedComponentsEnabled(true);
```

Note: Some of the layout classes (`IlvHierarchicalLayout`, `IlvCircularLayout`) have a built-in algorithm for placing connected components. This algorithm is enabled by default and fits the most common situations. For these layout classes, the generic mechanism provided by the base class `IlvGraphLayout` is disabled by default.

When enabled, a default instance of the class `IlvGridLayout` is used internally to place the disconnected graphs. If necessary, you can customize this layout.

Example of customizing layout

To customize this layout:

In Java

Call:

```
IlvGridLayout gridLayout = new IlvGridLayout();
gridLayout.setLayoutMode(IlvGridLayout.TILE_TO_ROWS);
gridLayout.setTopMargin(20);

layout.setLayoutOfConnectedComponents(gridLayout);
```

Example for experts

The various capabilities of the class `IlvGridLayout` cover most of the likely needs for the placement of disconnected graphs. However, if necessary, you can write your own subclass of `IlvGraphLayout` to place disconnected graphs and specify it instead of `IlvGridLayout`:

In Java

Call:

```
MyGridLayout myGridLayout = new MyGridLayout();

// settings for myGridLayout, if necessary

layout.setLayoutOfConnectedComponents(myGridLayout);
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsLayoutOfConnectedComponents()
```

The default implementation returns `false`. You can write a subclass to override this behavior.

Layout region

Some layout algorithms can control the size of the graph drawing and can take into account a user-defined layout region.

Example of specifying layout region

To specify a region of 100 by 100:

In Java

```
layout.setLayoutRegion(new IlvRect(0,0,100,100));
```

To access the layout region, use the method:

```
IlvRect getSpecLayoutRegion()
```

This method returns a copy of the rectangle that defines the specified layout region.

The layout algorithms call a different method:

```
IlvRect getCalcLayoutRegion()
```

This method first tries to use the layout region specification by calling the method `getSpecLayoutRegion()`. If this method returns a non-null rectangle, this rectangle is returned. Otherwise, the method tries to estimate an appropriate layout region according to the number and size of the nodes in the attached graph. If no graph is attached, or the attached graph is empty, it returns a default rectangle (0, 0, 1000, 1000).

To indicate whether a subclass of `IlvGraphLayout` supports the layout region mechanism, use the method:

```
boolean supportsLayoutRegion()
```

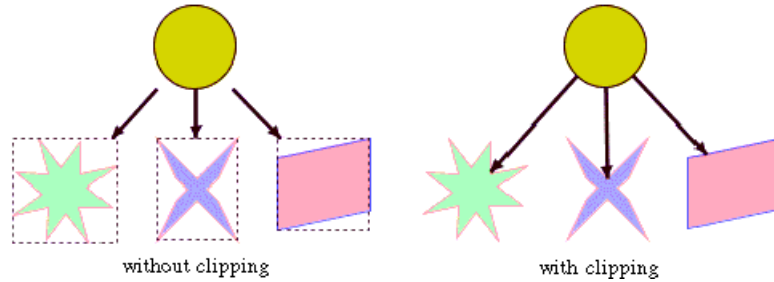
The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

Note: The implementation of the method `layout(boolean)` is solely responsible for whether the layout region is taken into account when calculating the layout, and in which manner. For details, refer to the documentation of the layout algorithms.

Link clipping

Some layout algorithms try to calculate the specific connection points of links at the border of nodes while other layout algorithms do not calculate any connection points.

If a layout algorithm calculates specific connection points, then it places the connection points of links by default at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want to place the connection points exactly on the border of the shape. This can be achieved by code by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. Some examples are shown in the following figure.



Effect of link clipping interface

Example of link clipping

To specify the link clip interface:

In Java

Use the method:

```
setLinkClipInterface(ilog.views.graphlayout.IlvLinkClipInterface)
```

You modify the position of the connection points of the links by implementing a class that implements the `IlvLinkClipInterface`. This interface defines the following method:

```
public IlvPoint getConnectionPoint
    (IlvGraphModel graphModel,
     Object node,
     IlvRect currentNodeBox,
     Object link,
     IlvPoint proposedConnectionPoint,
     IlvPoint auxControlPoint,
     boolean origin)
```

This method `getConnectionPoint(ilog.views.graphlayout.IlvGraphModel, java.lang.Object, ilog.views.IlvRect, java.lang.Object, ilog.views.IlvPoint, ilog.views.IlvPoint, boolean)` allows you to return the corrected connection point when the layout algorithm tries to connect to the proposed connection point. The `auxControlPoint` parameter is the auxiliary control point of the link segment that ends at the proposed connection point. The flag `origin` indicates whether the connection point is the start point or the end point of the link.

One strategy is to calculate the intersection between the ray starting at `auxControlPoint` and going through `proposedConnectionPoint` and the shape of the node. If there is any intersection, we return the one closer to `auxControlPoint`. If there is no intersection, clipping is not possible and we return the proposed connection point.

The following sample shows how to set a link clip interface that clips the connection points at the border of an ellipse or circle node:

```
layout.setLinkClipInterface(new IlvLinkClipInterface() {
    public IlvPoint getConnectionPoint
        (IlvGraphModel graphModel,
         Object node,
```

```

        IlvRect nodeBox,
        Object link,
            IlvPoint proposedConnectionPoint,
            IlvPoint auxControlPoint,
            boolean origin)
    {
        // get the intersections between the line through connect and control
        // point and the ellipse at currentNodeBox.
        IlvPoint[] intersectionPoints = new IlvPoint[2];
        int numIntersections = IlvClippingUtil.LineIntersectsEllipse(
            proposedConnectionPoint, auxControlPoint,
            nodeBox, intersectionPoints);
        // choose the result from the intersections
        return IlvClippingUtil.BestClipPointOnRay(proposedConnectionPoint,
            auxControlPoint,
            intersectionPoints,
            numIntersections);
    }
});

```

Link connection box

If a layout algorithm calculates specific connection points, it places the connection points of links by default at the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes (see *Effect of Link Connection Box Interface*). This can be achieved by specifying a link connection box interface. The link connection box interface allows you to specify, for each node, a node box different from the bounding box that is used to connect the links to the node.

Example of link connection box interface

In Java

To set a link connection box interface in Java, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface interface)
```

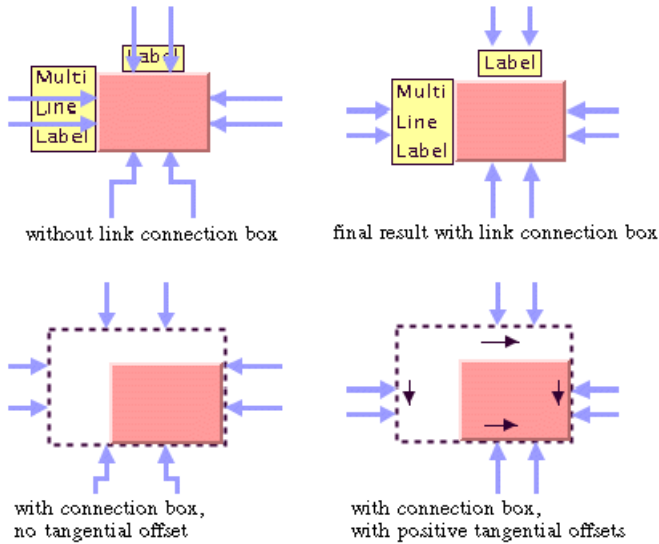
You implement the link connection box interface by defining a class that implements the `IlvLinkConnectionBoxInterface`. This interface defines the following method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

A second method defined on the interface allows the connection points to be “shifted” tangentially, in a different way for each side of each node:

```
public float getTangentialOffset(IlvGraphModel graphModel,
                                Object node, int nodeSide);
```



Effect of Link Connection Box Interface

For instance, to set a link connection box interface that returns a link connection rectangle that is smaller than the bounding box for all nodes of type `IlvShadowRectangle` and shifts up the connection points on the left and right side of all the nodes, call:

```
layout.setLinkConnectionBoxInterface(new IlvLinkConnectionBoxInterface() {
    public IlvRect getBox(IlvGraphModel graphModel, Object node) {
        IlvRect rect = graphModel.boundingBox(node);
        if (node instanceof IlvShadowRectangle) {
            // need a rect that is 4 pixels smaller
            rect.resize(rect.width-4.f, rect.height-4.f);
        }
        return rect;
    }
});

public float getTangentialOffset(IlvGraphModel graphModel,
                                Object node, int nodeSide) {
    switch (nodeSide) {
        case IlvDirection.Left:
        case IlvDirection.Right:
            return -10; // shift up with 10 for both left and right side
    }
    return 0; // no shift for top and bottom side
}
});
```

Some layout algorithms allow you to use the link connection box interface and the link clip interface in a combined way. It is specific to each layout algorithm how the interfaces will be used and which connection points are the final result.

To indicate whether a subclass of `IlvGraphLayout` supports the link connection box interface, use the method:

```
boolean supportsLinkConnectionBox()
```

The default implementation returns `false`. You can write a subclass to override this method in order to return `true` to indicate that this mechanism is supported.

Memory savings

The computation of a layout on a large graph may require a large amount of memory. Some layout algorithms optionally use two ways to store data: one which gives the priority to speed (this is the default case), the other which consumes less memory and is usually slower. The amount of memory savings depends, of course, on the implementation of the subclass of `IlvGraphLayout`. No matter which option you choose for memory savings, the resulting layout should be the same.

Example of memory savings

To enable memory savings:

In Java

Use the method:

```
void setMemorySavings(boolean option)
```

Memory savings is disabled by default.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsMemorySavings()
```

The default implementation returns `false`. You can write a subclass to override this method in order to return `true` to indicate that this mechanism is supported.

Percentage of completion calculation

Some layout algorithms can provide an estimation of how much of the layout has been completed. This estimation is made available as a percentage value that is stored in the graph layout report. When the algorithm starts, the percentage value is set to 0. The layout algorithm calls the following method from time to time to increase the percentage value by steps until it reaches 100:

```
void increasePercentageComplete(int newPercentage);
```

The percentage value can be accessed from the layout report using the following:

```
int percentage = layoutReport.getPercentageComplete();
```

To see an example of how to read the percentage value during the running of a layout, see *Graph layout event listeners*.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsPercentageComplete()
```

The default implementation returns `false`. A subclass can override this method to return `true` to indicate that this mechanism is supported.

Preserve fixed links

At times, you may want some links of the graph to be “pinned” (that is, to stay in their current shape when the layout is performed). You need a way to indicate the links that the layout algorithm cannot reshape. This makes sense especially when using a semi-automatic layout (the method where the end user fine-tunes the layout by hand after the layout is completed) or when using an incremental layout (the method where the graph and/or the shape of the links is modified after the layout has been performed, and then the layout is performed again).

Example of fixing links

To specify that a link is fixed:

In Java

Use the method:

```
void setFixed(Object link, boolean fixed)
```

If the `fixed` parameter is set to `true`, it means that the link is fixed. To obtain the current setting for a link:

```
boolean isFixed(Object link)
```

The default value is `false`.

To remove the fixed attribute from all links in the grapher, use the method:

```
void unfixAllLinks()
```

The fixed attributes on links will be taken into consideration only if you additionally call the following statement:

```
layout.setPreserveFixedLinks(true);
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsPreserveFixedLinks()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

Preserve fixed nodes

At times, you may want some nodes of the graph to be “pinned” (that is, to stay in their current position when the layout is performed). You need a way to indicate the nodes that the layout algorithm cannot move. This makes sense especially when using a semi-automatic layout (the method where the end user fine-tunes the layout by hand after the layout is completed) or when using an incremental layout (the method where the graph and/or the position of the nodes is modified after the layout has been performed, and then the layout is performed again).

Example of fixing nodes

To specify that a node is fixed:

In Java

Use the method:

```
void setFixed(Object node, boolean fixed)
```

If the `fixed` parameter is set to `true`, it means that the node is fixed. To obtain the current setting for a node:

```
boolean isFixed(Object node)
```

The default value is `false`.

To remove the fixed attribute from all nodes in the grapher, use the method:

```
void unfixAllNodes()
```

The fixed attributes on nodes will be taken into consideration only if you also call:

```
layout.setPreserveFixedNodes(true);
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsPreserveFixedNodes()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

Random generator seed value

Some layout algorithms use random numbers (or randomly chosen parameters) for which they accept a user-defined seed value. For example, the Random Layout uses the random

generator to compute the coordinates of the nodes. The Uniform Length Edges Layout uses the random generator to compute some internal variables.

Subclasses of `IlvGraphLayout` that are designed to support this mechanism allow the user to choose one of three ways of initializing the random generator:

- ◆ With a default value that is always the same.
- ◆ With a user-defined seed value that can be changed when re-performing the layout.
- ◆ With an arbitrary seed value, which is different each time. In this case, the random generator is initialized based on the system time.

The user chooses the initialization option depending on what happens when the layout algorithm is performed again on the same graph. If the same seed value is used, the same layout is produced, which may be the desired result. In other situations, the user may want to produce different layouts in order to select the best one. This can be achieved by performing the layout several times using different seed values.

Example of random generator seed value

In Java

This example shows how this parameter can be used in Java in combination with the `java.util.Random` class in your implementation of the method `IlvGraphLayout.layout()`:

```
Random random = (isUseSeedValueForRandomGenerator()) ?
    new Random(getSeedValueForRandomGenerator()) :
    new Random();
```

To specify the seed value in Java, use the method:

```
void setSeedValueForRandomGenerator(long seed)
```

The default seed value is 0.

The user-defined seed value is used only if you call additionally

```
layout.setUseSeedValueForRandomGenerator(true);
```

To indicate whether a subclass of `IlvGraphLayout` supports this parameter, use the method:

```
boolean supportsRandomGenerator()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this parameter is supported.

To specify the seed value, use the method:

```
void setSeedValueForRandomGenerator(long seed)
```

The default seed value is 0.

The user-defined seed value is used only if you call additionally

```
layout.setUseSeedValueForRandomGenerator(true);
```

To indicate whether a subclass of `IlvGraphLayout` supports this parameter, use the method:

```
boolean supportsRandomGenerator()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this parameter is supported.

Stop immediately

Several layout algorithms can stop computation when an external event occurs, for instance when the user hits a “Stop” button. In Java, to stop the layout, you can call:

```
boolean stopImmediately();
```

This method is typically called in a multithreaded application from a separate thread that is not the layout thread. The method returns `true` if the stop was initiated and `false` if the algorithm cannot stop. The method returns immediately, but the layout thread usually needs some additional time after initiating the stop to clean up data structures.

The consequences of stopping a layout process depend on the specific layout algorithm. Some layout algorithms have an iterative nature. Stopping the iteration process results in a slight loss of quality in the drawing, but the layout can still be considered valid. Other layout algorithms have a sequential nature. Interrupting the sequence of the layout steps may not result in a valid layout. Usually, these algorithms return to the situation before the start of the layout process.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsStopImmediately()
```

The default implementation returns `false`. You can write a subclass to override this method in order to return `true` to indicate that this mechanism is supported.

Use default parameters

All the generic parameters have a default value. After modifying parameters, you may want the layout algorithm to use the default values. Then, you may want to return to your customized values. IBM® ILOG® JViews keeps the previous settings when selecting the default values mode. In Java, you can switch between the default values mode and the mode for your own settings using the method:

```
void setUseDefaultParameters(boolean option)
```

To obtain the current value:

```
boolean isUseDefaultParameters()
```

The default value is `false`. This means that any setting you make will be taken into consideration and the parameters that have not been specified will have their default values.

Layout characteristics

It is often useful to know how certain settings will affect the resulting layout of the graph after the layout algorithm has been applied. The following table provides additional information about the behavior of the layout algorithms.

Layout characteristics of layout algorithms

| Layout algorithm | Do the initial positions of the nodes affect the layout? | How do I get a different layout of the same graph when I perform the layout a second time? |
|---|---|---|
| <i>Topological Mesh Layout (TML)</i> | No | You can completely change the layout by using the starting node, outer cycle, and fixed nodes parameters. To change only the dimensions of the graph, use the layout region parameter. See <i>Outer cycle (TML)</i> and <i>Using fixed nodes (TML)</i> . |
| <i>Force-directed or Uniform Length Edges Layout (ULEL)</i> | Yes | In incremental mode, you can completely change the layout by changing the initial positions of the nodes. To change only the dimensions of the graph, use the preferred length of the links or size of the layout region. See <i>Preferred length (ULEL)</i> . |
| <i>Tree Layout (TL)</i> | Yes (if incremental mode is switched on) | In incremental mode, you can change the layout by changing the initial positions of the nodes. Furthermore, you can change the layout by selecting a different <i>Root node (TL)</i> . To change only the dimensions of the graph, use the various offset parameters. |
| <i>Hierarchical Layout (HL)</i> | Yes (if incremental mode is switched on) | In incremental mode, you can change the layout by changing the initial positions of the nodes. Furthermore, you can use specified node level indices to change the level structure. See <i>Level index parameter (HL)</i> . You can use specified node position indices to change the node order within the levels. See <i>Position index parameter (HL)</i> . You can change the layout by changing the link priorities. See <i>Link priority parameter (HL)</i> . To change only the dimensions of the graph, use the various offset parameters. |
| <i>Link layout (LL)</i> | Yes | Link Layout routes the links depending on the node positions. It does not move the nodes. You can change the link style option and the dimensional parameters, such as |

| Layout algorithm | Do the initial positions of the nodes affect the layout? | How do I get a different layout of the same graph when I perform the layout a second time? |
|-----------------------------|---|--|
| | | the link offset and final segment length. You can also specify the rules for computing the connection points of the links. |
| <i>Random layout (RL)</i> | No | This is the default behavior when using the default parameter settings (the random generator is initialized differently each time). |
| <i>Bus layout (BL)</i> | No, except in incremental mode | You change the dimensions of the graph by using the various dimensional parameters. |
| <i>Circular layout (CL)</i> | No | You can completely change the layout by using clustering settings and the root clusters parameter. You can change the dimensions of the graph by using the dimensional parameters. |
| <i>Grid layout (GL)</i> | Yes (if incremental mode is switched on) | You can change various dimensional parameters, layout mode, and so on. |
| <i>Recursive layout</i> | Depends on the behavior of the sublayouts applied to the subgraphs. | Depends on the behavior of the sublayouts applied to the subgraphs. You can change the parameters of the sublayouts individually. |
| <i>Multiple layout</i> | Depends on the behavior of the sublayout that is applied first. | Depends on the behavior of the sublayouts of the Multiple Layout instance. You can change the parameters of the sublayouts individually. |

Topological Mesh Layout (TML)

Gives information on the *Topological Mesh Layout (TML)* algorithm (class `IlvTopologicalMeshLayout` from the package `ilog.views.graphlayout.topologicalmesh`).

In this section

General information on the TML

Provides samples of the layout and explains where it is likely to be used.

Features and limitations of the TML

Lists the features and limitations of the layout.

The TML algorithm

Gives an explanation of the concepts underlying TML, a brief description of the algorithm and a sample.

Generic features and parameters of the TML

Describes the generic parameters supported by TML and explains the particular way in which these parameters are used by this subclass.

Specific parameters of the TML

Describes the specific parameters supported by TML and gives samples of their use.

Refining a graph layout (TML)

Describes how to refine the layout by fixing some nodes and avoiding overlapping nodes.

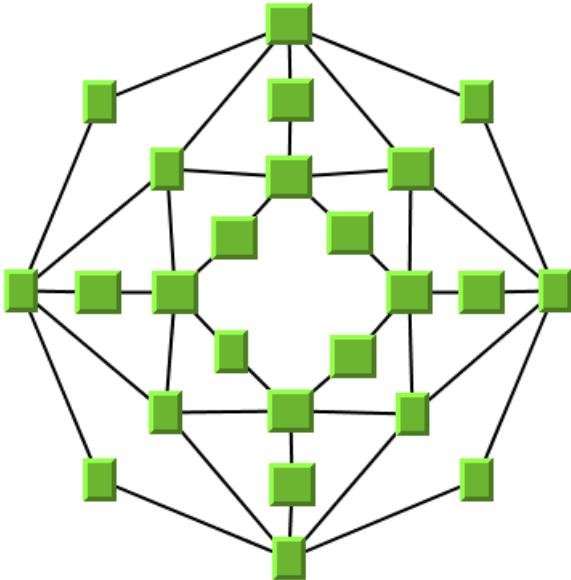
Using a link clipping interface with the TML

Describes the use of a link clipping interface.

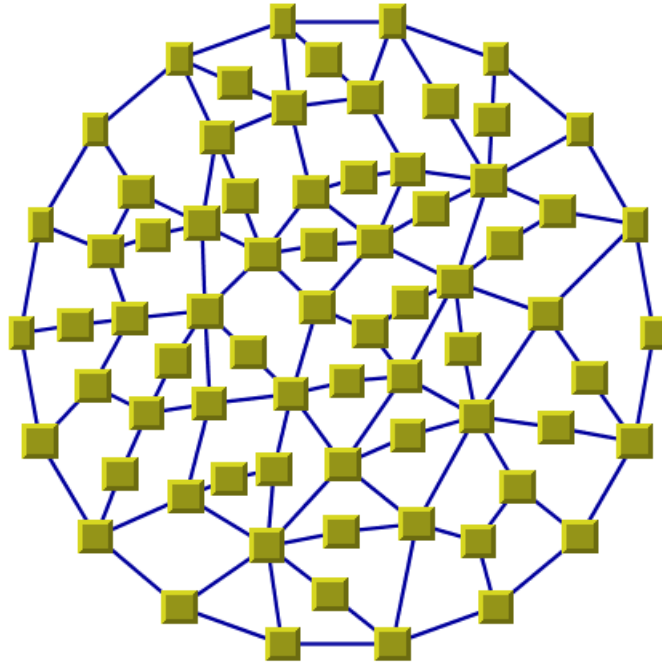
General information on the TML

TML samples

The following sample drawings were produced with TML.



Small cyclic graph drawing produced with TML



Large cyclic graph drawing produced with TML

What types of graphs suit the TML?

- ◆ Cyclic (2-connected graph) graphs. (Preferably without cut-nodes or cut-edges; otherwise, manual adjustments are necessary.)
- ◆ Cyclic (2-connected) graphs plus only a few branches. (You may need to make manual adjustments for the branches.)
- ◆ Both planar graphs and nonplanar graphs.

Application domains for the TML

Application domains of the Topological Mesh Layout include:

- ◆ Database and knowledge engineering (semantic networks, qualitative reasoning and other artificial intelligence diagrams)

Features and limitations of the TML

Features

- ◆ Most of the time, produces planar drawings of planar graphs, and drawings with a small number of link crossings for nonplanar graphs.
- ◆ Produces a nice layout for most small- and medium-size graphs relatively quickly. (The maximum cyclomatic number of the graph is about 30-50, but the number of nodes and links can be a lot higher.)
- ◆ Most of the time, produces symmetrical drawings of symmetrical graphs.
- ◆ The computation time for one iteration depends on the cyclomatic number of the graph, which is smaller than the number of nodes or links.
- ◆ The user can obtain several layouts of the same graph easily and quickly by simply changing a parameter (especially the starting node and the outer cycle) or by applying manual refinements to the layout. The best layout can then be selected from the resulting layouts.

Limitations

- ◆ The algorithm tries to minimize the number of link crossings (which is generally an NP-complete problem). It is mathematically impossible to quickly solve this problem for any graph size. Therefore, the algorithm uses heuristics that cannot always obtain a layout with the theoretical minimum number of link crossings.
- ◆ The computation time required to obtain an appropriate drawing grows relatively quickly with the cyclomatic number and the layout process may become very time-consuming for large graphs. Again, this is because the minimization of the number of link crossings is mathematically NP-complete in the general case.
- ◆ The algorithm cannot automatically produce appropriate drawings of some types of graphs:
 - For graphs containing branches and graphs containing cut-nodes or cut-edges, manual adjustments are necessary. (See *Refining a graph layout (TML)*.)
 - For disconnected graphs, the connected component layout feature should be used. (See *Layout of connected components*)
- ◆ The layout algorithm often produces a drawing with no overlapping nodes. Nevertheless, overlapping nodes cannot always be avoided. When overlapping occurs, you can try to increase the size of the layout region parameter or to change the outer cycle (see the method `setExteriorCycleId(int)`). You can also use manual adjustments to correct the problem.

The TML algorithm

TML is a heuristical approach for the layout of cyclic graph, either planar graphs or nonplanar graphs. TML is very simple to use. However, to use all the functionality of TML, you should understand its basic concepts.

When laying out a general graph, producing a drawing with a minimum number of link crossings is a mathematically NP-complete problem. The search space (and time) grows exponentially with the graph size. Traditionally, most of the existing layout algorithms use node coordinates from the beginning, searching for a coordinate set to minimize the cost function, which is mainly the number of link crossings. These coordinates can be constrained on a grid, but the number of combinations to explore is still enormous.

In contrast, TML uses a **two-step approach** that drastically reduces the number of combinations to explore. The first step of TML deals only with the pure topology (that is, the connectivity) of the graph without taking into consideration the node coordinates. This first step is called **topological optimization**. It chooses one of the cycles of the graph to be used in the second step.

In the second step, called **node placement**, the result of the first step is used to compute the coordinates of the nodes using a deterministic, high-speed barycenter algorithm. Of course, the problem still remains NP-complete and large graphs cannot be processed. In practice, however, you will often get better results for “mesh” graphs with TML than with many other algorithms.

Step 1: Topological optimization

Input

The topology of the graph (its connectivity or the neighborhood relationships between nodes).

Output

A set of possible outer cycles, ordered decreasingly by their lengths. The length of a cycle is the number of nodes in the cycle.

Explanation

This step determines what cycles of the graph, if used as an outer cycle for drawing the graph during nodes placement, will allow a drawing with a minimum number of link crossings. An optimization algorithm tries to minimize a heuristic cost function that estimates the number of link crossings for each solution, based on pure topology (graph connectivity)

Step 2: Node placement

Input

The output of topological optimization and the graph.

Output

A set of coordinates for the nodes. The coordinates are assigned to the nodes to obtain the graph drawing.

Explanation

This step is a variant of the “barycentric” layout algorithm. It takes a cycle from the output of topological optimization and draws it as a regular polygon. Then, it iteratively moves each node (except those on the regular polygon) at the “barycenter” of its neighbors (the nodes to which it is connected). This procedure always converges, and the final result is a graph drawing where the number of link crossings is dependent only on the choice of the outer cycle.

Example of TML

In Java

Below is a code sample using the `IlvTopologicalMeshLayout` class. This code sample shows how to perform a Topological Mesh Layout:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.runtime.topologicalmesh.*;
...

IlvTopologicalMeshLayout layout = new IlvTopologicalMeshLayout();
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);
try {
    IlvTopologicalMeshLayoutReport layoutReport =
        (IlvTopologicalMeshLayoutReport) layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
layout.detach();
graphModel.dispose();
```

It is possible to enable the link layout additionally, and in this case, the link layout determines the shape of the links.

Important: All explanations in the subsequent sections regarding the shape of the links in Topological Mesh Layout are valid only if the link layout is disabled.

Generic features and parameters of the TML

Overview (TML)

TML supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- ◆ *Allowed time (TML)*
- ◆ *Layout of connected components (TML)*
- ◆ *Layout region (TML)*
- ◆ *Link clipping (TML)*
- ◆ *Link connection box (TML)*
- ◆ *Memory savings (TML)*
- ◆ *Preserve fixed links (TML)*
- ◆ *Preserve fixed nodes (TML)*
- ◆ *Stop immediately (TML)*

Note that all the methods allowing the modification of these parameters are overridden in this subclass. This class keeps track of the changes for parameters that may affect the result of Topological Optimization separately from the parameters that may affect only the nodes placement step. In this way, the Topological Optimization step is not repeated. The previous results are used if no parameters were modified since the last time the layout was successfully performed on the same graph using the same layout instance.

Allowed time (TML)

The Topological Optimization step of TML stops if the allowed time setting has elapsed. In the same manner, the Nodes Placement step of TML stops if the allowed time is exceeded. (See *Allowed time*.)

You can specify separate time settings for each step. Each step is stopped if its specified time limit is exceeded. To learn how to do this, see *Optimization iterations and allowed time (TML)* and *Node placement iterations and allowed time (TML)*.

Layout of connected components (TML)

The layout algorithm can use the generic mechanism to lay out connected components. (For more information about this mechanism, see *Layout of connected components*.)

If the generic connected component layout mechanism is disabled, the algorithm lays out only the connected component that contains the starting node.

Layout region (TML)

The Nodes Placement step of TML first draws the outer cycle computed in the Topological Optimization step as a regular polygon. It uses the layout region setting (either your own or the default setting) to choose the size and the position of the polygon. The remaining nodes are moved inside this polygon. (See *Layout region*.)

If you are using the default settings, an estimation of the appropriate layout region according to the number and size of the nodes is used.

If TML produces a layout with overlapping nodes, one possible way to correct the problem is to increase the size of the layout region. (For details, see *Using the layout region parameter (TML)*.)

Link clipping (TML)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface with the TML* for details of the link clipping mechanism in TML.

Link connection box (TML)

The layout algorithm can use a link connection box interface (see *Link connection box*) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a link clipping interface with the TML*.

Memory savings (TML)

As with all classes supporting this parameter, a certain amount of memory savings can be obtained by selecting this option. Note that using this option does not change the resulting layout. It just slows down the computation. (See *Memory savings*.)

Preserve fixed links (TML)

TML does not reshape the links that are specified as fixed. (See *Preserve fixed links*. See also *Link style (TML)*.)

Preserve fixed nodes (TML)

TML does not move the nodes that are specified as fixed. Moreover, the algorithm takes into account the fixed nodes when computing the position of the nonfixed nodes. (See *Preserve fixed nodes*.)

If TML produces a layout with overlapping nodes, you can use the fixed nodes mechanism to correct the problem. (For details, see *Using fixed nodes (TML)*.)

Stop immediately (TML)

The layout algorithm stops after cleanup if the method `IlvTopologicalMeshLayout` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

See also

Support by algorithms of generic features and parameters

Specific parameters of the TML

The following parameters are specific to the `IlvTopologicalMeshLayout` class.

Link style (TML)

When the layout algorithm moves the nodes, straight-line links will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

To specify that the layout algorithm automatically remove all the intermediate points of the links (if any):

In Java™

Use the method:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

◆ `IlvTopologicalMeshLayout.NO_RESHAPE_STYLE`

None of the links is reshaped in any manner.

◆ `IlvTopologicalMeshLayout.STRAIGHT_LINE_STYLE`

All the intermediate points of the links (except for links specified as fixed) are removed. This is the default value.

Optimization iterations and allowed time (TML)

The iterative computation performed in the Topological Optimization step is stopped if the number of iterations exceeds the allowed number of iterations for optimization or the time exceeds the allowed time for optimization (or, of course, if the general layout time has elapsed; see *Allowed time (TML)*).

To specify the parameters:

In Java

Use the methods:

```
void setAllowedOptimizationTime(long time)
void setAllowedNumberOfOptimizationIterations(int iter)
```

The `time` is in milliseconds. The default value is 28000 (28 seconds).

Node placement iterations and allowed time (TML)

The iterative computation performed in the Nodes Placement step is stopped if the number of iterations exceeds the allowed number of iterations or the time exceeds the allowed time

for node placement (or, of course, if the general layout time has elapsed; see *Allowed time (TML)*).

To specify these parameters:

In Java

Use the methods:

```
void setAllowedNodesPlacementTime(long time)
```

```
void setAllowedNumberOfNodesPlacementIterations(int iter)
```

The `time` is in milliseconds. The default value is 28000 (28 seconds).

Node placement algorithm (TML)

Two barycentric algorithms are implemented for the Nodes Placement step of TML.

To specify the algorithm:

In Java

Use the method:

```
void setNodesPlacementAlgorithm(int option)
```

The valid values for `option` are:

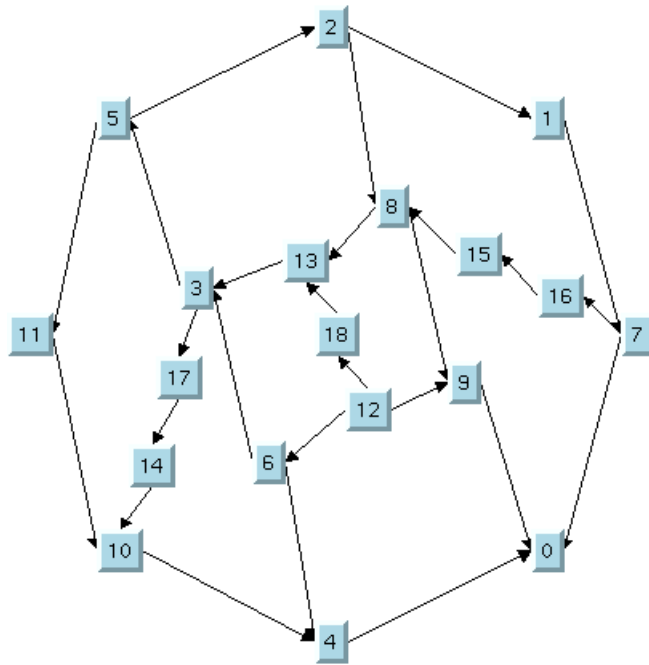
◆ `IlvTopologicalMeshLayout.SLOW_GOOD`

This option provides more uniformity of the nodes distribution inside the outer cycle, but is slightly slower.

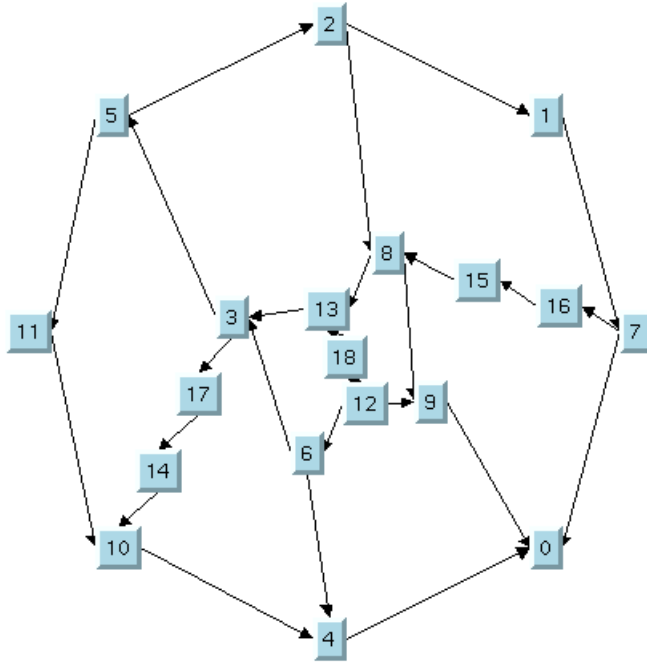
◆ `IlvTopologicalMeshLayout.QUICK_BAD`

This option provides less uniformity of the nodes distribution, but is slightly quicker.

In most cases, both algorithms are fairly quick. We recommend that you use the `SLOW_GOOD` version, which is the default value. Compare the layouts of the same graph in *Node placement algorithm: SLOW_GOOD* and *Node placement algorithm: QUICK_BAD* to get an idea of the difference between these algorithms.



Node placement algorithm: SLOW_GOOD



Node placement algorithm: *QUICK_BAD*

Outer cycle (TML)

The Topological Optimization step of TML computes a set of cycles that can be used as the outer cycle in the Nodes Placement step . By default, the longest cycle is actually used (that is, the cycle containing the largest number of nodes). However, you may find it useful to try a different outer cycle. To do so in Java, use the method:

```
void setExteriorCycleId(int cycleId)
```

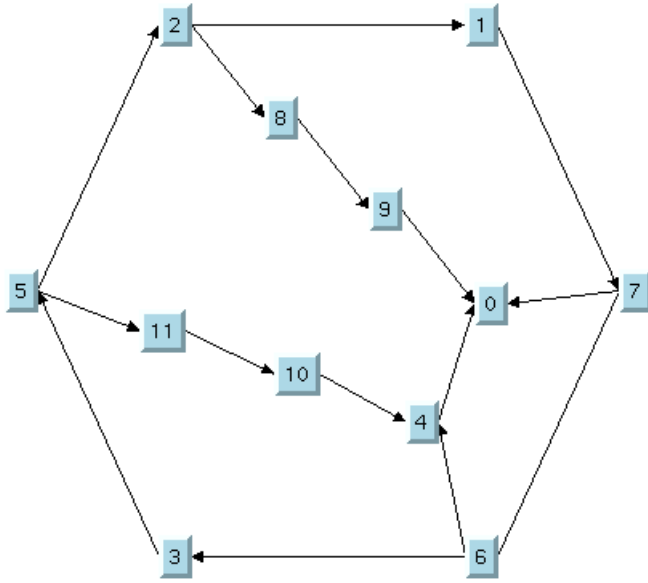
The valid values for `cycleId` range from zero to the number of cycles computed by the Topological Optimization step minus one. This number is returned by the method:

```
int getNumberOfPossibleExteriorCycles()
```

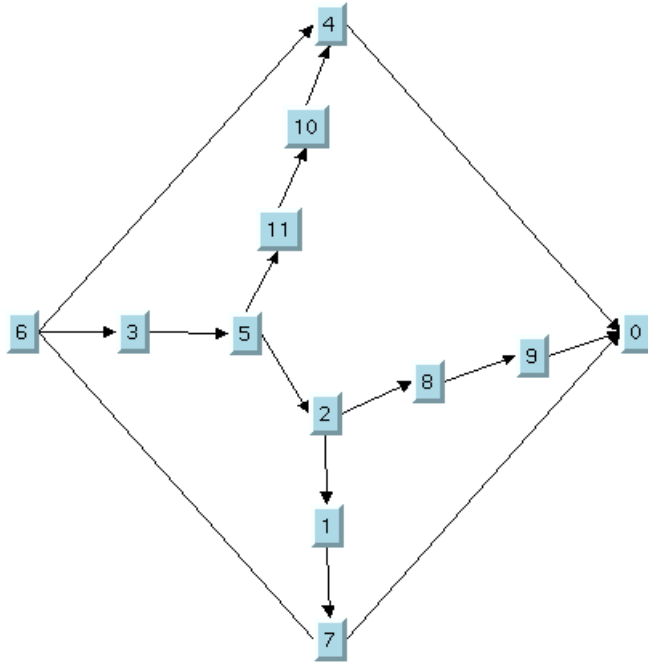
If the number is not in this range, the value zero is used.

You can use these methods only after having performed the layout successfully. Otherwise, no outer cycle is defined.

When the layout is performed again with a new outer cycle, only the Nodes Placement step of TML is performed, and not the time-consuming the Topological Optimization step. This is true if the topology of the graph has not been changed (that is, no nodes or links were added or removed), and no parameters that affect the Topological Optimization step have been changed.



Layout using 3rd outer cycle



Layout using 4th outer cycle

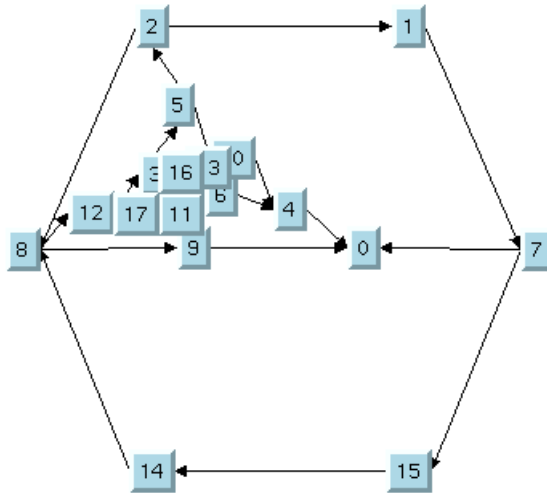
Refining a graph layout (TML)

After performing the layout on a graph, you may want to improve the quality of the layout by making some manual refinements. The subsequent sections describe several ways to refine your layouts. When the layout is performed again after the refinements have been applied, only the Nodes placement step of TML is redone. The results of the Topological Optimization are reused. This is an important benefit of TML because the algorithm can recompute a layout using new parameters very quickly, without performing the time-consuming Topological Optimization step again.

Using fixed nodes (TML)

One reason for applying manual refinements is to avoid overlapping nodes. To do this, you can use the fixed nodes mechanism. (See *Preserve fixed nodes*.)

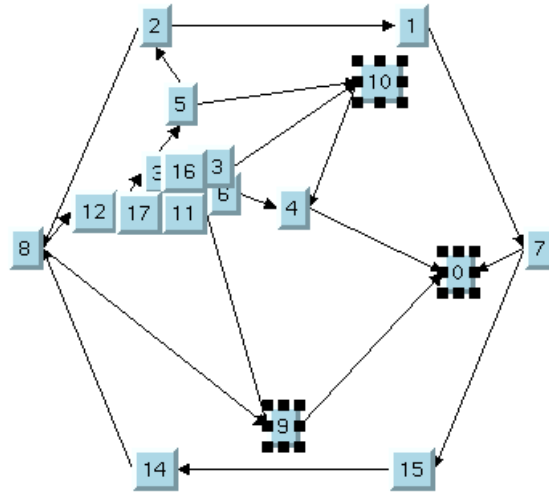
Take a look at the original layout shown in *The original TML layout*. Several overlapping nodes exist in the original layout because the nodes are concentrated in a small region and do not use the available space inside the outer cycle.



The original TML layout

To correct the problem, you can perform the following steps:

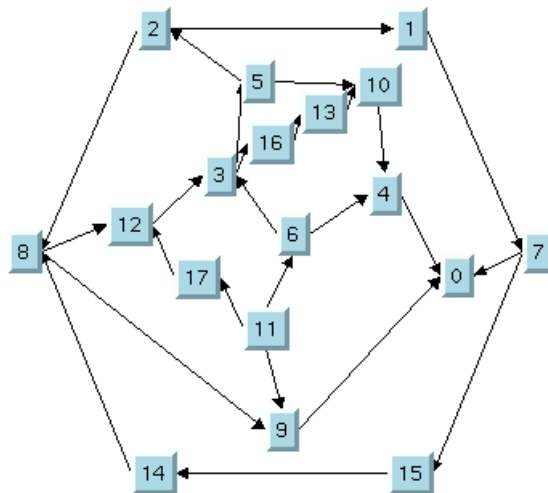
1. Move nodes 0, 9, and 10 to a place in the free space inside the outer cycle by hand as shown in *The TML layout with some nodes moved*.



The TML layout with some nodes moved

2. Specify nodes 0, 9, and 10 as fixed using the `setFixed(java.lang.Object, boolean)` method.
3. Use the `setPreserveFixedNodes(boolean)` method to specify that the fixed nodes will not be moved when the layout is performed.
4. Perform the layout again. Only Step 2 will be performed.

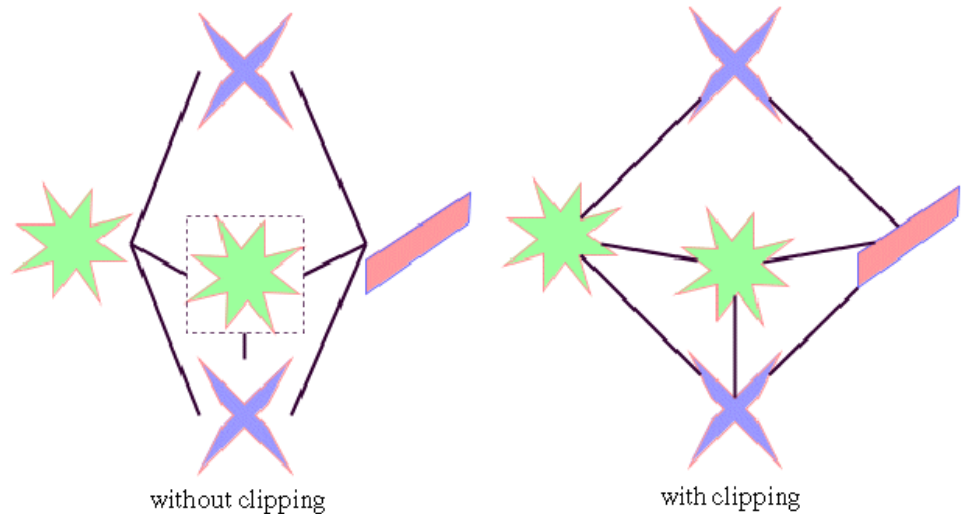
The fixed nodes “attract” the other nodes, which are distributed in the larger area inside the outer cycle as shown in *The final TML layout with some fixed nodes*.



The final TML layout with some fixed nodes

Using a link clipping interface with the TML

By default, TML does not place the connection points of links. The default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



Effect of Link Clipping Interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*. To set a link clip interface:

To set a link clip interface:

In Java™

Use the method:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*. To set a link connection box interface:

To set a link connection box interface:

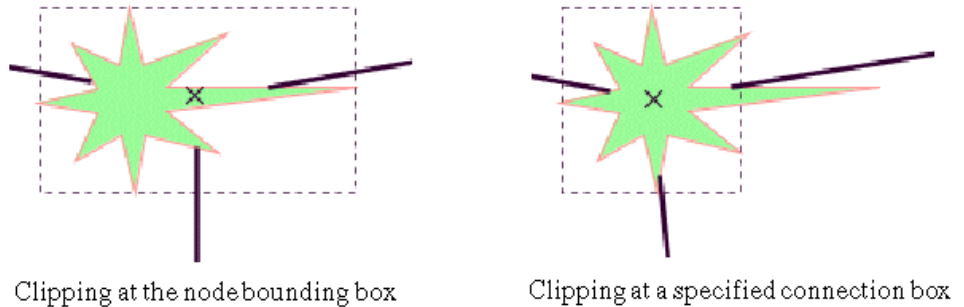
In Java

Use the method:

```
void setLinkConnectionBoxInterface(ILvLinkConnectionBoxInterface interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

The following figure shows an example of the combined effect.



Combined effect of link clipping interface and link connection box

If the links are clipped at the green irregular star node (see previous figure, left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (see previous figure, right). Alternatively, the problem could be corrected by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Force-directed or Uniform Length Edges Layout (ULEL)

Describes the *Force-directed layout* or *Uniform Length Edges Layout* algorithm (class `IlvUniformLengthEdgesLayout` from the package `ilog.views.graphlayout.uniformlengthedges`).

In this section

General information on the ULEL

Provides samples of the layout and explains where it is likely to be used.

Features and limitations of the ULEL

Lists the features and limitations of the layout.

The ULEL algorithm

Gives an explanation of the ULEL algorithm and a sample.

Generic features and parameters of the ULEL

Lists the generic features and parameters of the Uniform Length Edges layout (ULEL).

Specific parameters of the ULEL

Describes the specific parameters supported by ULEL and gives samples of their use.

For experts: additional features of the ULEL

Describes the parameters available to expert users.

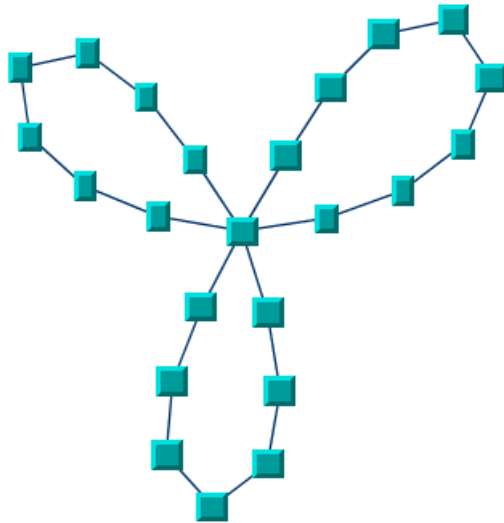
Using a link clipping interface with the ULEL

Describes the use of a link clipping interface with the Uniform Length Edges Layout (ULEL).

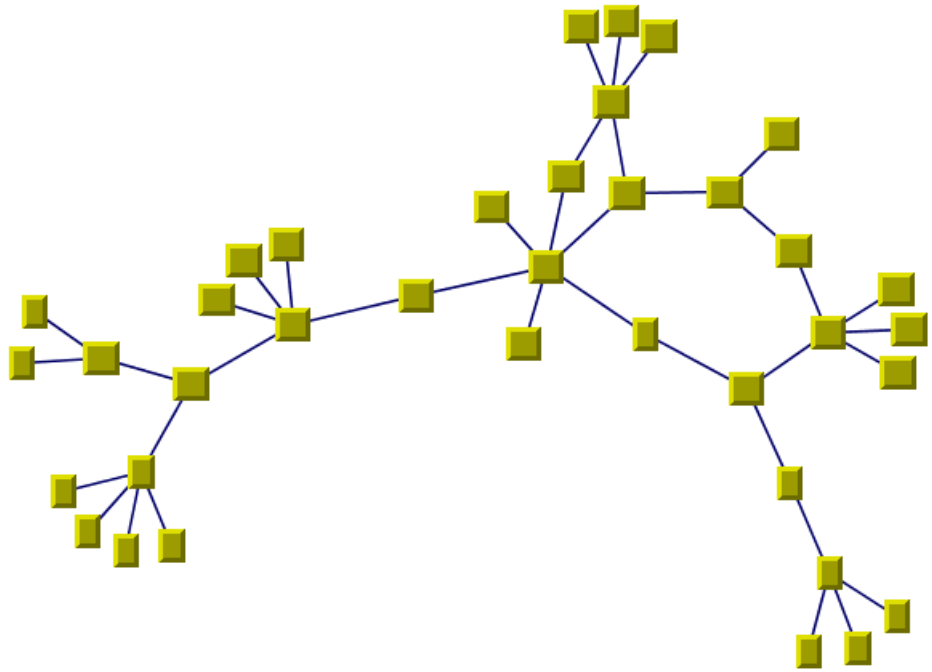
General information on the ULEL

ULEL samples

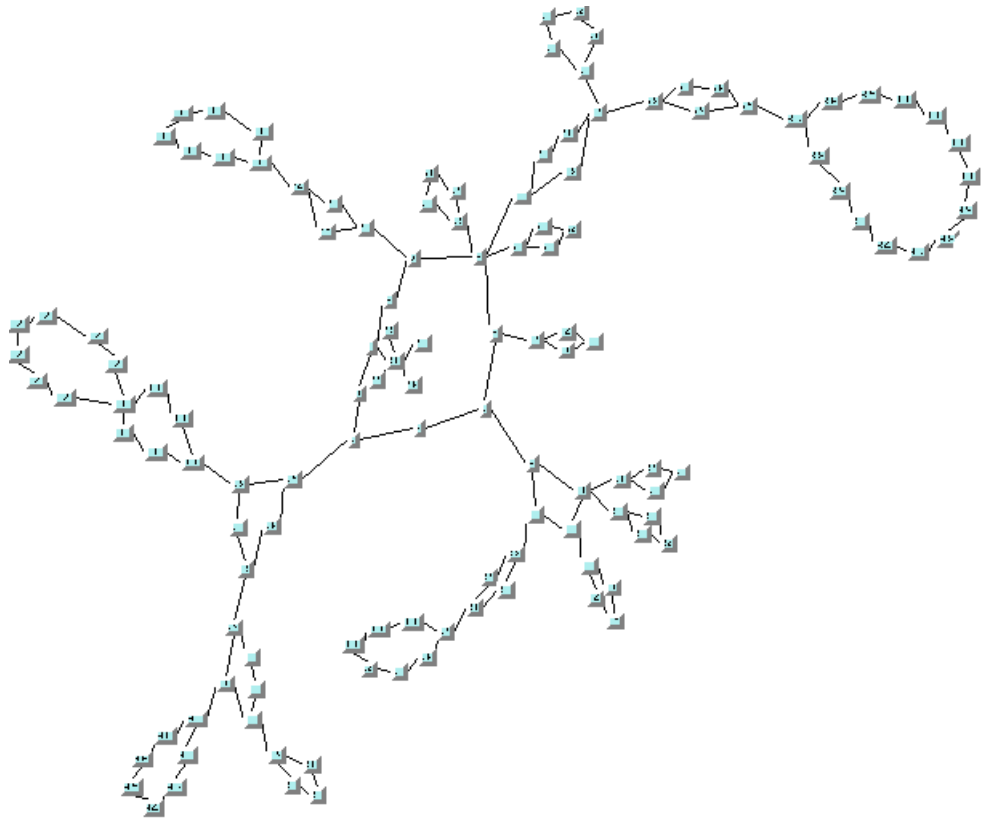
The following sample drawings are produced with the Uniform Length Edges Layout (ULEL).



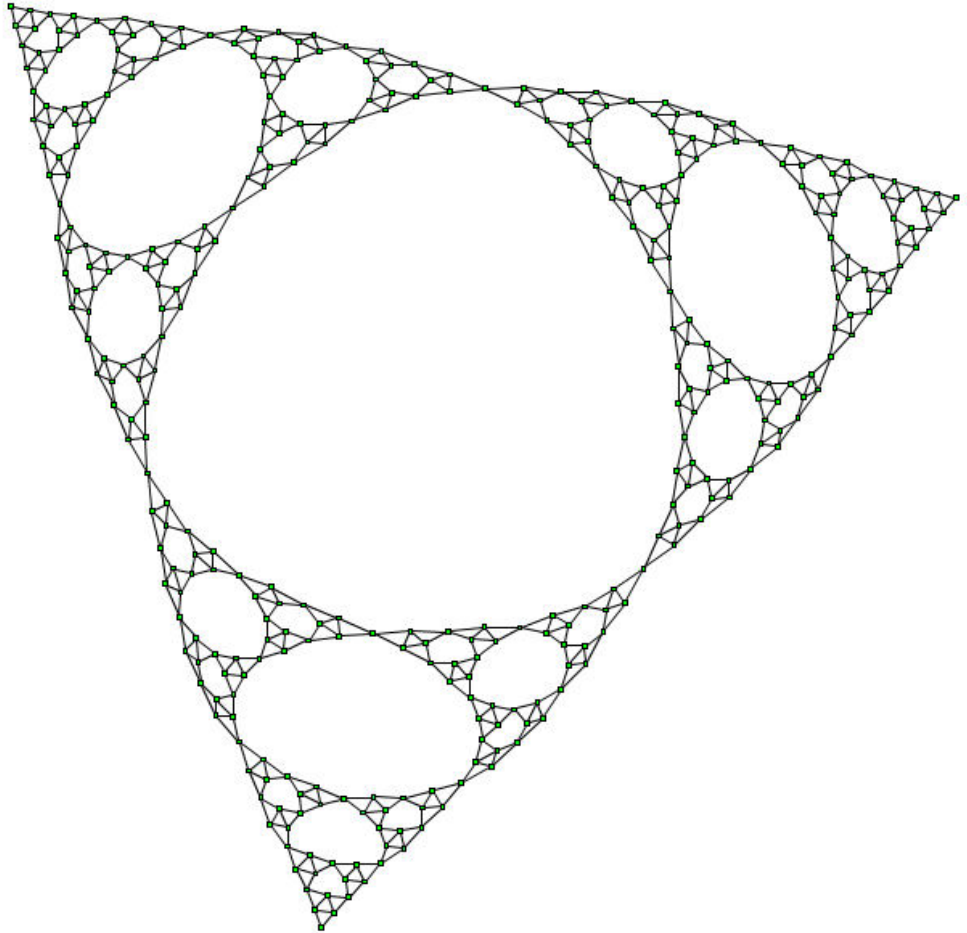
Small cyclic graph drawing produced with the Uniform Length Edges Layout



Medium graph drawing (combination of cycles and trees) produced with the Uniform Length Edges Layout



Large graph drawing (combination of cycles and trees) produced with the Uniform Length Edges Layout



Large graph drawing (Sierpinski Triangle) produced with the Uniform Length Edges Layout using the fast multilevel layout mode

What types of graphs suit the ULEL?

Any type of graph:

- ◆ connected graphs and disconnected graphs
- ◆ planar graphs and nonplanar graphs

Application domains for the ULEL

Application domains for the Uniform Length Edges Layout include:

- ◆ Telecoms and networking (WAN diagrams)

- ◆ Software management/software (re-)engineering (call graphs)
- ◆ CASE tools (dependency diagrams)
- ◆ Database and knowledge engineering (semantic networks, database query graphs, qualitative reasoning and other artificial intelligence diagrams, and so on)
- ◆ World Wide Web (Web hyperlink neighborhood)

Features and limitations of the ULEL

Features

Often provides a drawing without any or with only a few link crossings and with approximately equal length links for small- and medium-size graphs having a small number of cycles. The maximum number of nodes for which you can use the algorithm depends on the connectivity of the graph and is difficult to predict.

On demand, the algorithm can take into account the size (width and height) of the nodes. Otherwise, they are more efficiently considered as points.

It is possible to specify the length for each link individually.

The algorithm provides three optional layout modes: incremental, non-incremental and fast multilevel. The non-incremental and fast multilevel modes are in general faster and are recommended for large graphs. For details, see *Layout mode* .

Limitations

- ◆ The algorithm is not appropriate for all graphs. In particular, it will produce bad results on some highly connected cyclic graphs for which a planar drawing with equal-length links may simply not exist.
- ◆ The computation time required to obtain an appropriate drawing grows relatively quickly with the size of the graph (that is, the number of nodes and links) and the layout process may become time-consuming for large graphs.
- ◆ Overlapping nodes cannot always be avoided. Nevertheless, the layout algorithm often produces a drawing with no overlapping nodes.

The ULEL algorithm

This layout algorithm iteratively searches for a configuration of the graph where the length of the links is close to a user-defined or a default value.

Example of ULEL algorithm

In Java

The following code sample uses the `IlvUniformLengthEdgesLayout` class. This code sample shows how to perform a Uniform Length Edges Layout:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.runtime.uniformlengthedges.*;
...

IlvUniformLengthEdgesLayout layout = new IlvUniformLengthEdgesLayout();
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);
try {
    IlvUniformLengthEdgesLayoutReport layoutReport =
        layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
layout.detach();
graphModel.dispose();
```

Important: All explanations in the subsequent sections regarding the shape of the links in Uniform Length Edges Layout are valid only if the link layout is disabled.

Generic features and parameters of the ULEL

Overview (ULEL)

The `IlvUniformLengthEdgesLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- ◆ *Allowed time (ULEL)*
- ◆ *Layout of connected components (ULEL)*
- ◆ *Layout region (ULEL)*
- ◆ *Link clipping (ULEL)*
- ◆ *Link connection box (ULEL)*
- ◆ *Preserve fixed links (ULEL)*
- ◆ *Preserve fixed nodes (ULEL)*
- ◆ *Stop immediately (ULEL)*

The following subsections describe the particular way in which these parameters are used by this subclass.

Allowed time (ULEL)

The layout algorithm stops if the allowed time setting has elapsed. (See *Allowed time*.)

Layout of connected components (ULEL)

The layout algorithm can utilize the generic mechanism to lay out connected components. (For more information about this mechanism, see *Layout of connected components*.)

Layout region (ULEL)

The layout algorithm can use the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing.

Note that by default the Uniform Length Edges Layout algorithm does not use the layout region. (For details see also *Force fit to layout region (ULEL)*.)

If you are using the default settings, an estimation of the appropriate layout region according to the number and size of the nodes is used.

Link clipping (ULEL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface with the ULEL* for details of the link clipping mechanism.

Link connection box (ULEL)

The layout algorithm can use a link connection box interface (see *Link connection box*.) In combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a link clipping interface with the ULEL*.

Preserve fixed links (ULEL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links* and *Link style (ULEL)*.)

Preserve fixed nodes (ULEL)

The layout algorithm does not move the nodes that are specified as fixed. Moreover, the algorithm takes into account the fixed nodes when computing the position of the nonfixed nodes. (See *Preserve fixed nodes*.)

Stop immediately (ULEL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the ULEL

The following parameters are specific to the `IlvUniformLengthEdgesLayout` class.

Link style (ULEL)

When the layout algorithm moves the nodes, straight-line links will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links, the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

To specify that the ULEL algorithm automatically removes all the intermediate points of the links (if any):

In Java™

Use this method:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

◆ `IlvUniformLengthEdgesLayout.NO_RESHAPE_STYLE`

None of the links is reshaped in any manner.

◆ `IlvUniformLengthEdgesLayout.STRAIGHT_LINE_STYLE`

All the intermediate points of the links (if any) are removed. This is the default value.

Number of iterations (ULEL)

The iterative computation of the layout algorithm is stopped if the time exceeds the allowed time (see *Allowed time*) or if the number of iterations exceeds the allowed number of iterations.

To specify the number of iterations:

In Java

Use the method:

```
void setAllowedNumberOfIterations(int iterations)
```

Preferred length (ULEL)

The main objective of this layout algorithm is to obtain a layout where all the links have a given length. This is called the “preferred length.”

To specify the preferred length:

Globally

◆ In Java

Use the method:

```
void setPreferredLinksLength(float length)
```

The default value is 60.0.

Individually

It is also possible to specify a length for individual links. To do so:

◆ In Java

Use the method:

```
void setPreferredLength(Object link, float length)
```

To obtain the current value, use the method:

```
float getPreferredLength(Object link)
```

If a specific length is not specified for a link, the global settings are used.

Respect node sizes (ULEL)

By default, the layout algorithm ignores the size (width and height) of the nodes. For efficiency reasons, the nodes are approximated with points placed in the center of the bounding box of the nodes. When dealing with large nodes, the preferred length parameter can be increased in such a way that the nodes do not overlap.

However, to improve the support for graphs with heterogeneous node sizes, the algorithm provides a special mode in which the particular size of each node is taken into consideration.

To set this mode:

In Java

Use the method:

```
void setRespectNodeSizes(boolean respect)
```

The default value is `false`.

Force fit to layout region (ULEL)

For this layout algorithm, it is more difficult than for others to choose an appropriate size for the layout region. If the specified layout region is too small for a given graph, the resulting layout will not be the best. For this reason, by default, the Uniform Length Edges Layout algorithm does not use the layout region parameter. It can use as much space as it needs to lay out the grapher.

To specify whether the layout algorithm must use the layout region:

In Java

Use the method:

```
void setForceFitToLayoutRegion(boolean option)
```

The default value of the parameter is `false`.

Layout mode

To fit a variety of needs, the algorithm provides three optional modes:

◆ Incremental mode

The algorithm starts from the current position and iteratively tries to converge towards the optimal layout. Thus, in some cases, this mode avoids a major reorganization of the graph, which helps for preserving the "mental map" of the user as much as possible. However, this is not guaranteed, and depends on how far is the initial position of the nodes from the position that satisfies the criteria of the algorithm.

◆ Non-incremental mode

The algorithm is free to reorganize the graph without trying to stay close to the initial positions. Often, the non-incremental mode is faster than the incremental mode, sometimes at the price of a lower quality.

◆ Fast multilevel mode

The algorithm uses a multilevel graph decomposition strategy that leads to significant speed gain. This mode is usually the fastest for medium and large graphs.

To set this mode:

In Java

Use the method:

```
void setLayoutMode(int mode)
```

The default value is `IlvUniformLengthEdgesLayout.INCREMENTAL_MODE`.

For experts: additional features of the ULEL

Expert users can also try and use the following parameters.

Maximum allowed move per iteration (ULEL)

At each iteration, the layout algorithm moves the nodes a relatively small amount. This amount should not be too large; otherwise the algorithm may not converge. But it should not be too small either, otherwise the number of necessary iterations increases and the running time does also.

The maximum amount of movement at each iteration is controlled by a parameter.

To set this parameter:

In Java™

Use the method:

```
void setMaxAllowedMovePerIteration(float maxMove)
```

Typical values for this setting are 1 to 30, but it depends on the value of the `PreferredLinksLength` parameter. For example, if the setting for the `PreferredLinksLength` parameter is 1000, then a value of 100 for the `MaxAllowedMovePerIteration` parameter is still meaningful.

Link length weight (ULEL)

The layout algorithm is based on the computation of attraction and repulsion forces for each of the nodes and the iterative search of an equilibrium configuration. One of these forces is related to the objective of obtaining a link length close to the specified preferred length.

The weight of this force, representing the total amount of forces, is controlled by a parameter.

To set this parameter:

In Java

Use the method:

```
void setLinkLengthWeight(float weight)
```

The default value is 1. Increasing this parameter can help obtain link lengths closer to the specified length, but increasing too much can increase the number of link crossings.

Additional node repulsion weight (ULEL)

An additional repulsion force can be computed between nodes that are not connected by a link. The weight of this force, representing the total amount of forces, is controlled by a parameter.

To set this parameter:

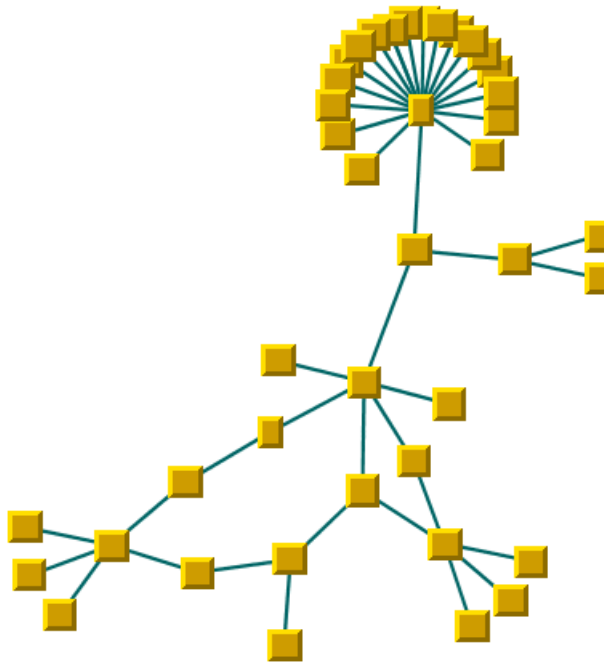
In Java

Use the methods:

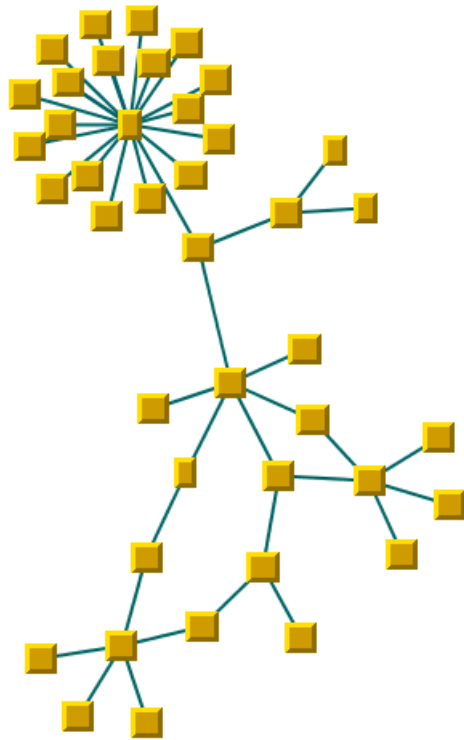

```
void setAdditionalNodeRepulsionWeight(float weight)
```

The default value of this parameter is $0.2f$. Increasing (or decreasing) the weight increases (or decreases) the priority that is given to maintain the nodes at a distance larger than the node distance threshold (see `setNodeDistanceThreshold(float)`). On the other side, increasing the weight decreases the ability for the algorithm to reach convergence quickly.

The following two figures enable you to compare the same graph laid out with additional repulsion disabled (*Additional repulsion disabled, produced with the Uniform Length Edges Layout*) and then enabled (*Additional repulsion enabled, produced with the Uniform Length Edges Layout*). You can see that the “star” configuration, where many nodes are connected to the same central node, is better displayed when additional repulsion is enabled.



Additional repulsion disabled, produced with the Uniform Length Edges Layout



Additional repulsion enabled, produced with the Uniform Length Edges Layout

Node distance threshold (ULEL)

The additional repulsion force between two nodes not connected by a link is computed only when their distance is smaller than a predefined distance.

To set this distance:

In Java

Use the method:

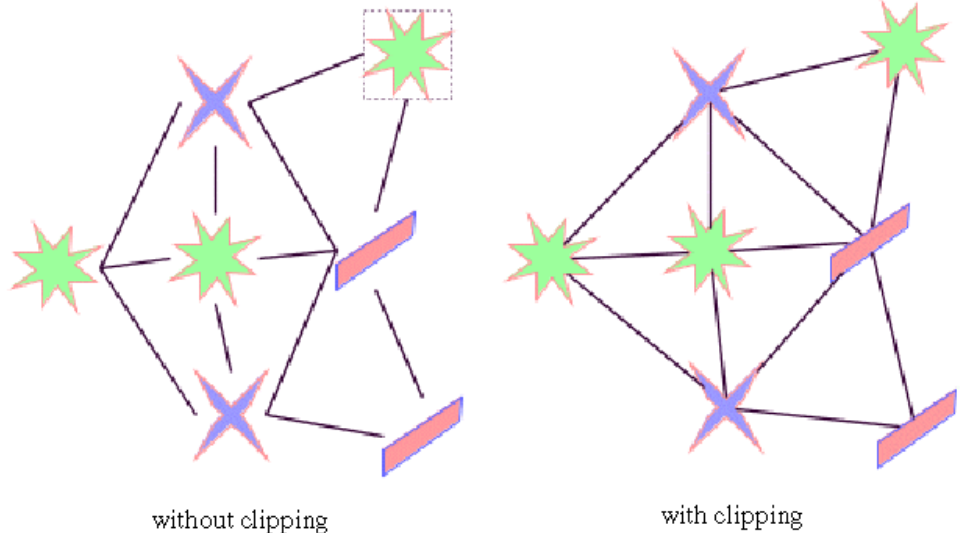
```
void setNodeDistanceThreshold(float threshold)
```

Note that this additional force is computed only if the “additional node repulsion weight” is set to a value larger than the default value 0.

It is recommended that this threshold be set to a value smaller than the preferred length of the links.

Using a link clipping interface with the ULEL

By default, the Uniform Length Edges Layout does not place the connection points of links. The default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



Effect of link clipping interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*.

To set a link clip interface:

In Java™

Use the method

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*.

To set a link connection box interface:

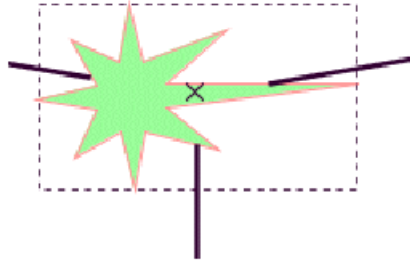
In Java

Use the method:

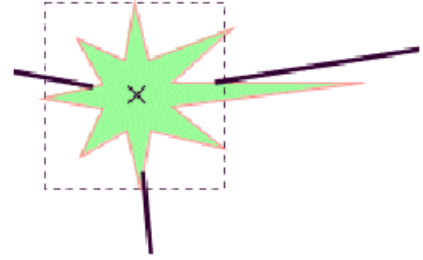
```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

The following figure shows an example of the combined effect.



Clipping at the node bounding box



Clipping at a specified connection box

Combined Effect of Link Clipping Interface and Link Connection Box

If the links are clipped at the green irregular star node (previous figure, left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (previous figure, right). Alternatively, the problem could be corrected by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Tree Layout (TL)

Describes the *Tree Layout* algorithm (class `IlvTreeLayout` from the package `ilog.views.graphlayout.tree`).

In this section

General information on the TL

Provides samples of the layout and explains where it is likely to be used.

Features and limitations of the TL

Lists the features and limitations of the layout.

The TL algorithm

Gives an explanation of the Tree Layout (TL) algorithm and a sample.

Generic features and parameters of the TL algorithm

Describes the generic parameters supported by the Tree Layout (TL) and explains the particular way in which these parameters are used by this subclass.

Specific parameters (for all tree layout modes)

Describes the specific parameters supported by the Tree Layout and gives samples of their use.

Layout modes of the TL algorithm

Describes the characteristics and the layout parameters of each layout mode in the TL algorithm.

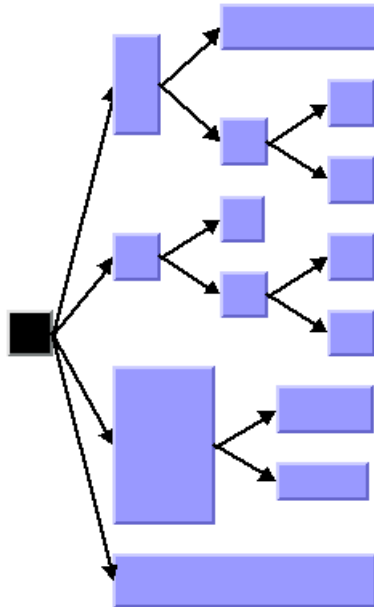
For experts: additional tips for the TL

Describes some tips and tricks for expert users of the Tree Layout (TL).

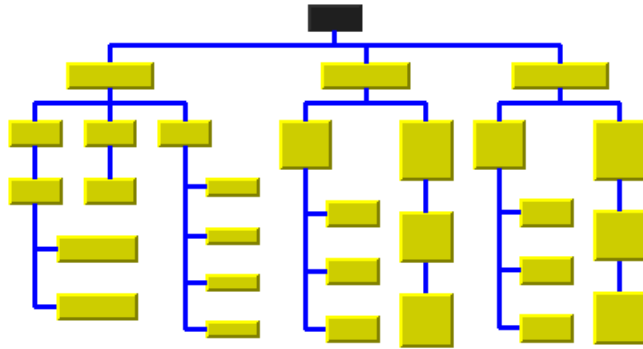
General information on the TL

TL samples

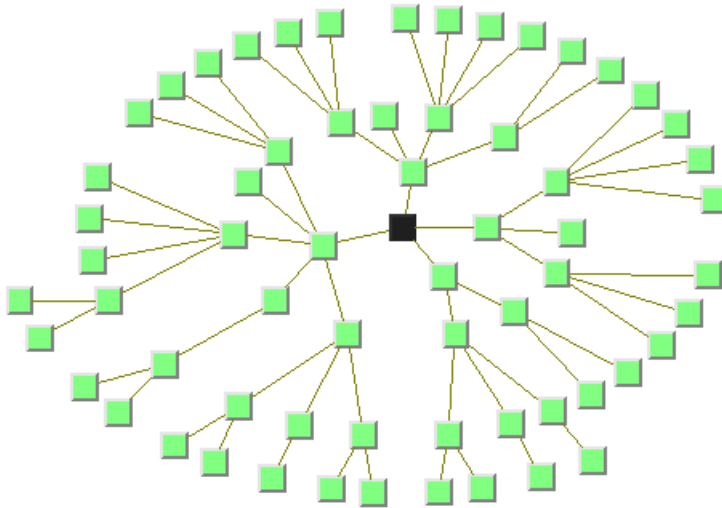
The following sample drawings are produced with the Tree Layout.



Tree layout in free layout mode with center alignment and flow direction to the right



Tree layout with flow direction to the bottom, orthogonal link style, and tip-over alignment at some leaf nodes



Tree layout in radial layout mode with aspect ratio 1.5

What types of graphs suit the TL?

- ◆ Primarily designed for pure trees. It can also be used for non-trees, that is, for cyclic graphs. In this case, the algorithm computes and uses a spanning tree of the graph, ignoring all links that do not belong to the spanning tree.
- ◆ Directed and undirected trees. If the links are directed, the algorithm automatically chooses the canonical root node. If the links are undirected, you can choose a root node.
- ◆ connected graphs and disconnected graphs. If the graph is not connected, the layout algorithm treats each connected component separately. Each component has exactly one root node. In this case, a forest of trees is laid out.

Application domains for the TL

Application domains for the Tree Layout include:

- ◆ Business processing (organizational charts)
- ◆ Software management/software (re-)engineering (UML diagrams, call graphs)
- ◆ Database and knowledge engineering (decision trees)
- ◆ The World Wide Web (Web site maps)

Features and limitations of the TL

Features

- ◆ Takes into account the size of the nodes so that no overlapping occurs.
- ◆ Optionally reshapes the links to give them an orthogonal form (alternating horizontal and vertical line segments).
- ◆ Various layout modes: *free*, *levels*, *radial*, or *automatic tip-over*.
 - In the free layout mode, arranges the children of each node, starting recursively from the root, so that the links flow uniformly in the same direction.
 - In the level layout mode, partitions the nodes into levels, and arranges the levels horizontally or vertically.
 - In radial layout mode, partitions the nodes into levels, and arranges the levels in circles or ellipses around the root.
 - In the tip-over mode, arranges the nodes in a similar way to the free layout mode, but tries to tip children over automatically to fit the layout better to the given aspect ratio.
- ◆ Provides several alignment and offset options.
- ◆ Allows you to specify nodes that must be direct neighbors.
- ◆ Provides incremental and nonincremental modes. Incremental mode takes the previous position of nodes into account and positions the nodes without changing the relative order of the nodes in the tree so that the layout is stable on incremental changes of the graph.
- ◆ Very efficient, scalable algorithm. Produces a nice layout quickly even if the number of nodes is huge.

Limitations

- ◆ If the orthogonal setting is not specified as the link style (see *Link style*), some links may in rare cases overlap nodes depending on the size of the nodes, the alignment parameters, and the offset parameters.
- ◆ The layout algorithm first determines a spanning tree of the graph. If the graph is not a pure tree, some links will not be included as part of the spanning tree. These links are ignored. For this reason, they may cross other links or overlap nodes in the final layout.
- ◆ For stability in incremental mode, the algorithm tries to preserve the relative order of the children of each node. It uses a heuristic to calculate the relative order from the previous positions of the nodes. The heuristic may fail if children overlap their old positions or are not aligned horizontally or vertically.
- ◆ Despite preserving the relative order of the children, in rare cases the layout is not perfectly stable in incremental radial layouts. Subsequent layouts may rotate the nodes around the root, although the relative circular order of the nodes within their circular levels is still preserved.

- ◆ The tip-over layout modes will perform several trial layouts with different tip-over alignment options according to various heuristics. From these trial layouts, the algorithm picks the layout that best fits the given aspect ratio. This may not be the optimal layout for the aspect ratio, but it is the best layout among the trials. Calculating the absolute best-fitting layout is not computationally feasible (it is generally an NP-complete problem).

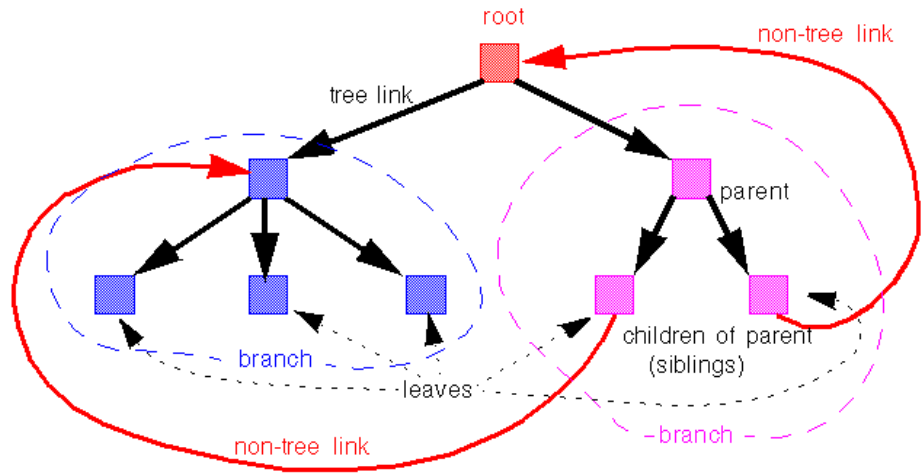
The TL algorithm

The core algorithm for the free, level, and radial layout modes works in just two steps and is very fast. The variations of the tip-over layout mode perform the second step several times and pick the layout result that best fits the given aspect ratio (the ratio between width and height of the drawing area). For this reason, the tip-over layout modes are slower.

Step 1: Calculating the spanning tree

If the graph is disconnected, the layout algorithm chooses a *root node* for each connected component. Starting from the root node, it traverses the graph to choose the links of the *spanning tree*. If the graph is a pure tree, all links are chosen. If the graph has cycles, some links will not be included as part of the spanning tree. These links are called *non-tree links*, while the links of the spanning tree are called *tree links*. The non-tree links are ignored in step 2 of the algorithm.

In *Tree layout in free layout mode with center alignment and flow direction to the right*, *Tree layout with flow direction to the bottom, orthogonal link style, and tip-over alignment at some leaf nodes*, and *Tree layout in radial layout mode with aspect ratio 1.5*, the root is the node that has no parent node. In the spanning tree, each node except the root has a parent node. All nodes that have the same parent are called *children* with respect to the parent and *siblings* with respect to themselves. Nodes without children are called *leaves*. Each child at a node starts a *subtree* (also called a branch of the tree). A *Spanning tree* show an example of a spanning tree.



A Spanning tree

Step 2: Calculating node positions and link shapes

The layout algorithm arranges the nodes according to the layout mode and the offset and alignment options. In the free mode and level mode, the nodes are arranged horizontally or vertically so that all tree links flow roughly in the same direction. In the radial layout modes, the nodes are arranged in circles or ellipses around the root so that all tree links flow radially away from the root. Finally, the link shapes are calculated according to the link style and alignment options.

Example of TL

The following code sample uses the `IlvTreeLayout` class. This code sample shows how to perform a Tree Layout:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.runtime.tree.*;
...
IlvTreeLayout layout = new IlvTreeLayout();
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);

/* Specify the root node, orientation and alignment */
layout.setRoot(rootNode);
layout.setFlowDirection(IlvDirection.Right);
layout.setGlobalAlignment(IlvTreeLayout.CENTER);

try {
    IlvGraphLayoutReport layoutReport = layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
layout.detach();
graphModel.dispose();
```

Important: All explanations in the subsequent sections regarding the shape of the links in Tree Layout are valid only if the link layout is disabled.

Generic features and parameters of the TL algorithm

Overview (TL)

The `IlvTreeLayout` class supports the following generic features defined in the `IlvGraphLayout` class (see also *Base class parameters and features*):

- ◆ *Allowed time (TL)*
- ◆ *Layout of connected components (TL)*
- ◆ *Link clipping (TL)*
- ◆ *Link connection box (TL)*
- ◆ *Percentage of completion calculation (TL)*
- ◆ *Preserve fixed links (TL)*
- ◆ *Preserve fixed nodes (TL)*
- ◆ *Stop immediately (TL)*

The following subsections describe the particular way in which these features are used by the subclass `IlvTreeLayout`.

Allowed time (TL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of connected components (TL)

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of connected components*). It has, however, a specialized internal mechanism to layout connected components and, therefore, the generic mechanism is switched off by default.

The generic connected component layout mechanism has the disadvantage that it moves connected components completely. Fixed nodes within a component do not preserve their old position, and the resulting layout may be unstable on incremental changes, depending on which layout instance is used for the component layout.

If the generic connected component layout mechanism is disabled, the algorithm uses its own specialized internal mechanism instead of the generic mechanism to lay out each component as a separate tree. This is usually faster and more stable on incremental changes than the generic mechanism. Furthermore, it enables the user to set the position of the layout.

Link clipping (TL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface* for details of the link clipping mechanism.

Link connection box (TL)

The layout algorithm can use a link connection box interface (see *Link connection box*) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a link connection box interface*.

Percentage of completion calculation (TL)

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of layout. (For a detailed description of this feature, see *Percentage of completion calculation* and *Graph layout event listeners*.)

Preserve fixed links (TL)

The layout algorithm does not reshape the links that are specified as fixed. (For more information on link parameters in the `IlvGraphLayout` class, see *Preserve fixed links* and *Link style (TML)*.)

Preserve fixed nodes (TL)

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see *Preserve fixed nodes*.) Moreover, the layout algorithm ignores fixed nodes completely and also does not route the links that are incident to the fixed nodes. This can result in unwanted overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Stop immediately (TL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call, and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters (for all tree layout modes)

The following parameters are specific to the `IlvTreeLayout` class. They apply to all layout modes.

Root node (TL)

The final layout is influenced mainly by the choice of the root node.

The root node is placed in a prominent position. For instance, in a top-down drawing with free layout mode, it is placed at the top of the tree. With the radial layout mode, it is placed at the center of the tree.

The spanning tree is calculated starting from the root node. If the graph is disconnected, the layout algorithm needs one root node for each connected component.

The layout algorithm automatically selects a root node when needed. It uses a heuristic that calculates preferences for all nodes to become a root. It chooses the node with the highest preference. The heuristic gives nodes without incoming links the highest preference and leaf nodes without outgoing links the lowest preference. Hence, in a directed tree, the canonical root is always chosen automatically.

It is possible to influence the choice of the root node.

To set a node explicitly as the root:

In Java

Use the method:

```
void setRoot(Object node);
```

This gives the node the maximal preference to become the root during layout. If only one node is specified this way, the algorithm selects this node. If several nodes of the same connected component are specified this way, the layout algorithm chooses one of them as the root.

For experts: additional options for root nodes (TL)

The layout algorithm manages a list of the root nodes that have been specified by the `setRoot` method. To obtain the nodes in this list in Java, use the method:

```
Enumeration getSpecRoots();
```

After layout, you can also retrieve the list of root nodes that were actually used by the algorithm. This list is not necessarily the same as the list of specified roots. For instance, it contains the chosen root nodes if none were specified or if too many were specified. To obtain the root nodes that were used by the algorithm in Java, use the method:

```
Enumeration getCalcRoots();
```

This example shows how to iterate over the calculated root nodes and print the root node preferences:

```
Enumeration e = layout.getCalcRoots();
while (e.hasMoreElements()) {
    node = e.nextElement();
    System.out.println("Preference:" + layout.getRootPreference(node));
}
```

To directly manipulate the root node preference value of an individual node:

In Java

Use the method:

```
setRootPreference(Object node, int preference);
```

In this case, the layout uses the specified value instead of the heuristically calculated preference for the node. The normal preference value should be between 0 and 10000. Specifying a root node explicitly corresponds to setting the preference value to 10000. If you want to prohibit a node from becoming the root, specify a preference value of zero (0).

A negative preference value indicates that the layout algorithm should recalculate the root node preference using the heuristic. If a root was specified by the `setRoot` method but this node should no longer be the root in subsequent layouts, use the following call to clear the root node setting:

```
layout.setRootPreference(node, -1);
```

This call also removes the node from the list of specified roots.

Position parameters (TL)

To set the position of the *top left corner* of the layout to (10, 10):

In Java

In Java, use the method:

```
layout.setPosition(new IlvPoint(10, 10), false);
```

If the graph consists of only a single tree, it is often more convenient to set the position of the root node instead. To do this:

In Java

Use the same method and pass `true` instead of `false`:

```
layout.setPosition(point, true);
```

If no position is specified, the layout keeps the root node at its previous position.

Using compass directions for positional layout parameters (TL)

The compass directions *north*, *south*, *east*, and *west* are used to simplify the explanations of the layout parameters. The center of the root node of a tree is considered the north pole.

In the nonradial layout modes, the link flow direction always corresponds to south. If the root node is placed at the top of the drawing, north is at the top, south at the bottom, east to the right, and west to the left. If the root node is placed at the left border of the drawing, north is to the left, south to the right, east at the top, and west at the bottom.

In the radial layout modes, the root node is placed in the center of the drawing. The meaning of north and south depends on the position relative to the root: the north side of the node is the side closer to the root and the south side is the side further away from the root. The east direction is counterclockwise around the root and the west direction is clockwise around the root. This is similar to a cartographic map of a real globe that shows the area of the north pole as if you were looking down at the top of the globe.

Compass directions are used to provide uniform naming conventions for certain layout options. They occur in the alignment options, the level alignment option, and the east-west neighboring feature, which are explained later. In *Flow directions* and *Radial layout mode*, the compass icons show the compass directions in these drawings.

Layout modes (TL)

The tree layout algorithm has several layout modes. The following example shows how to specify the layout mode.

In Java

Use the method:

```
void setLayoutMode(int mode);
```

The available layout modes are the following:

- ◆ `IlvTreeLayout.FREE` (the default)
- ◆ `IlvTreeLayout.LEVEL`
- ◆ `IlvTreeLayout.RADIAL`
- ◆ `IlvTreeLayout.ALTERNATING_RADIAL`
- ◆ `IlvTreeLayout.TIP_OVER`
- ◆ `IlvTreeLayout.TIP_ROOTS_OVER`
- ◆ `IlvTreeLayout.TIP_LEAVES_OVER`
- ◆ `IlvTreeLayout.TIP_ROOTS_AND_LEAVES_OVER`

Layout modes of the TL algorithm

Describes the characteristics and the layout parameters of each layout mode in the TL algorithm.

In this section

Free layout mode

Describes how the free layout mode organizes nodes and describes the parameters of this mode.

Level layout mode

Describes how the level layout mode organizes nodes and describes the parameters of this mode.

Radial layout mode

Describes how the radial layout mode organizes nodes and describes the parameters of this mode.

Tip-over layout modes

Describes the need for tip-over layout modes and how they operate.

Recursive mode

Describes how the recursive mode organizes nodes and describes the parameters of this mode.

Free layout mode

Describes how the free layout mode organizes nodes and describes the parameters of this mode.

In this section

Overview

Describes how the free layout mode organizes nodes.

Flow direction

Describes the flow direction parameter of the free layout mode.

Alignment parameter

Describes the alignment parameter of the free layout mode.

Link style

Describes the link style parameter of the free layout mode.

Connector style

Describes the connector style parameter of the free layout mode and how to use it in conjunction with two interfaces.

Using a link connection box interface

Describes how to use the link connection box interface in the free layout mode.

Using a link clipping interface

Describes how to use the link clipping interface in the free layout mode.

Spacing parameters

Describes how to use the spacing parameter of the free layout mode.

Overview

The free layout mode arranges the children of each node starting recursively from the root so that the links flow roughly in the same direction. For instance, if the link flow direction is top-down, the root node is placed at the top of the drawing. Siblings (nodes that have the same parent) are justified at their top borders, but nodes of different tree branches (nodes with different parents) are not justified.

To set the free layout mode:

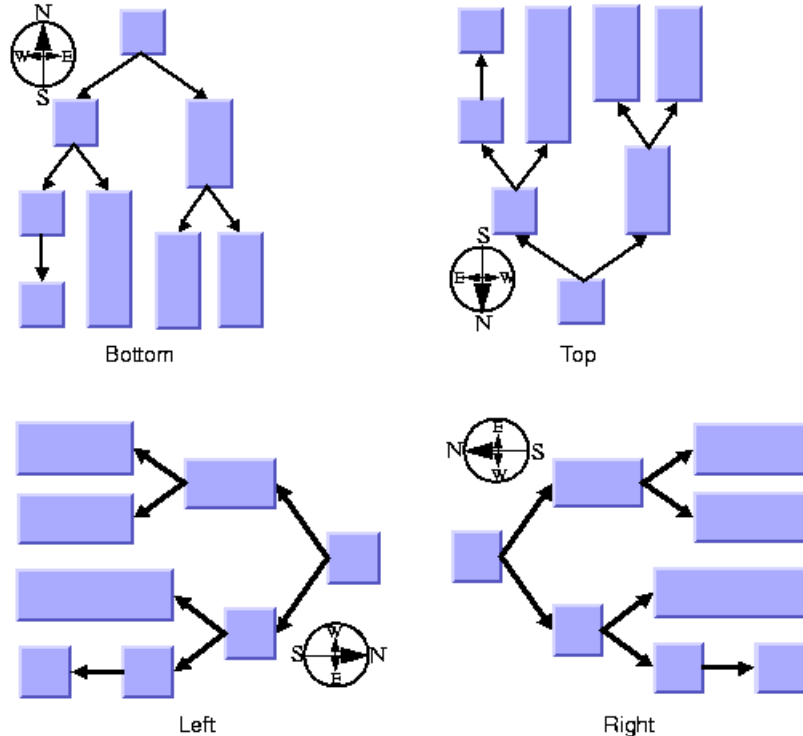
In Java™

Call:

```
layout.setLayoutMode(IlvTreeLayout.FREE);
```

Flow direction

The flow direction parameter specifies the direction of the tree links. The compass icons show the compass directions in these layouts.



Flow directions

If the flow direction is to the bottom, the root node is placed topmost. Each parent node is placed above its children, which are normally arranged horizontally. (This tip-over alignment is an exception.)

If the flow direction is to the right, the root node is placed leftmost. Each parent node is placed to the left of its children, which are normally arranged vertically.

To specify the flow direction:

In Java

In Java™, use the method:

```
void setFlowDirection(int direction);
```

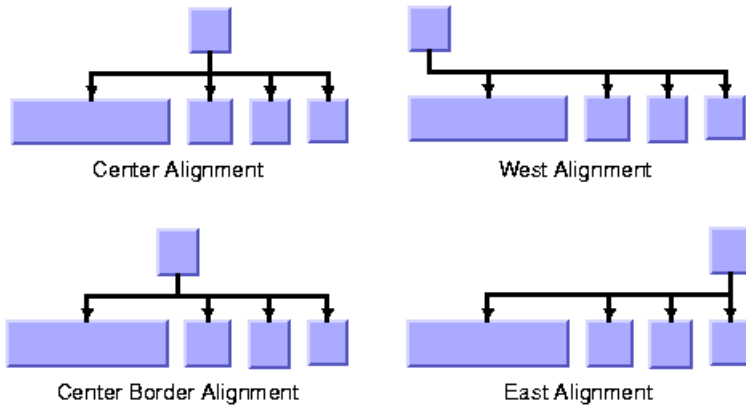
The valid values for the flow direction are:

- ◆ `IlvDirection.Right` (the default)

- ◆ `IlvDirection.Left`
- ◆ `IlvDirection.Bottom`
- ◆ `IlvDirection.Top`

Alignment parameter

The alignment option controls how a parent is placed relative to its children. The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.



Alignment Options

Global alignment

To set the global alignment:

In Java

In Java™, use the method:

```
void setGlobalAlignment(int alignment);
```

The valid values for the global alignment are:

◆ `IlvTreeLayout.CENTER` (the default)

The parent is centered over its children, taking the center of the children into account.

◆ `IlvTreeLayout.BORDER_CENTER`

The parent is centered over its children, taking the border of the children into account. If the size of the first and the last child varies, the border center alignment places the parent closer to the larger child than to the default center alignment.

◆ `IlvTreeLayout.EAST`

The parent is aligned with the border of its easternmost child. For instance, if the flow direction is to the bottom, east is the direction to the right. If the flow direction is to the top, east is the direction to the left. See *Using compass directions for positional layout parameters (TL)* for details.

◆ `IlvTreeLayout.WEST`

The parent is aligned with the border of its westernmost child. For instance, if the flow direction is to the bottom, west is the direction to the left. If the flow direction is to the right, west is the direction to the bottom. See *Using compass directions for positional layout parameters (TL)* for details.

◆ `IlvTreeLayout.TIP_OVER`

The children are arranged sequentially instead of in parallel, and the parent node is placed with an offset to the children. For details see *Tip-over alignment*.

◆ `IlvTreeLayout.TIP_OVER_BOTH_SIDES`

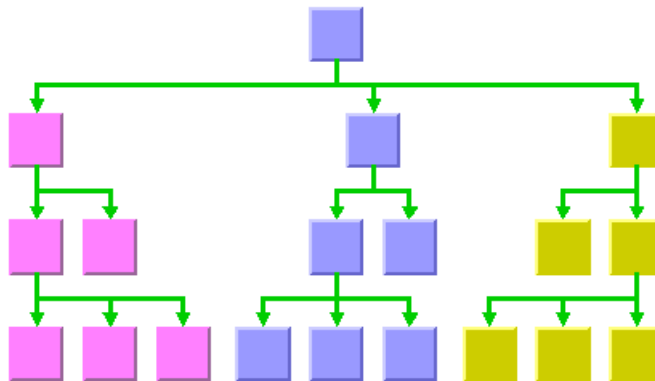
The children are arranged sequentially instead of in parallel. Whereas the alignment `TIP_OVER` arranges all children at the same side of the parent, this alignment arranges the children at both sides of the parent. For details see *Tip-over alignment*.

◆ `IlvTreeLayout.MIXED`

Each parent node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

Alignment of individual nodes

All nodes have the same alignment unless the global alignment is set to `MIXED`. Only when the global alignment is set to `MIXED` can each node have an individual alignment style.



Different Alignments Mixed in the Same Drawing

To specify the alignment of an individual node:

In Java

Use the methods:

```
void setAlignment(Object node, int alignment);
```

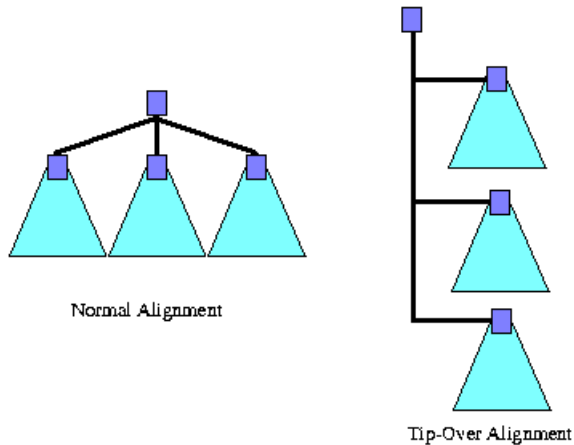
```
int getAlignment(Object node);
```

The valid values for alignment are:

- ◆ `IlvTreeLayout.CENTER` (the default)
- ◆ `IlvTreeLayout.BORDER_CENTER`
- ◆ `IlvTreeLayout.EAST`
- ◆ `IlvTreeLayout.WEST`
- ◆ `IlvTreeLayout.TIP_OVER`
- ◆ `IlvTreeLayout.TIP_OVER_BOTH_SIDES`

Tip-over alignment

Normally, the children of a node are placed in a *parallel* arrangement with siblings as direct neighbors of each other. Tip-over alignment means a *sequential* arrangement of the children instead.



Normal alignment and tip-over alignment

Tip-over alignment is useful when the tree has many leaves. With normal alignment, a tree with many leaves would result in the layout being very wide. If the global alignment style is set to tip-over, the drawing is very tall rather than wide. To balance the width and height of the drawing, you can set the global alignment to mixed, for example, in Java:

```
layout.setGlobalAlignment(IlvTreeLayout.MIXED);
```

Also, you can set the individual alignment to tip-over for some parents with a high number of children as follows:

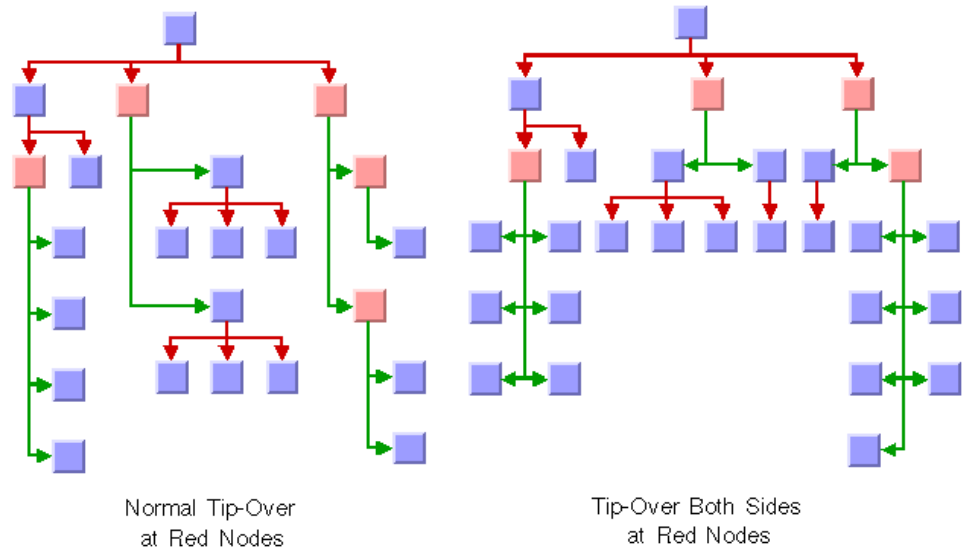
```
layout.setAlignment(parent, IlvTreeLayout.TIP_OVER);
```

Tip-over alignment can be specified explicitly for some or all of the nodes. Furthermore, the Tree Layout offers layout modes that automatically determine when to tip over, yielding a drawing that fits into a given aspect ratio. These layout modes are described in *Tip-over layout modes*.

Besides the normal tip-over alignment, there is also a variant that distributes the subtrees on both sides of the center line that starts at the parent. You can specify this variant at a parent node with a high number of children by the following code:

```
layout.setAlignment(parent, IlvTreeLayout.TIP_OVER_BOTH_SIDES);
```

The following figure illustrates the difference between normal tip-over alignment and tip-over at both sides. Tip-over alignment works very well with the orthogonal link style (see *Link style*).



Tip-over alignment

Link style

The links can be straight or have a specific shape with intermediate points. You can specify that the links be reshaped into an “orthogonal” form. You can set the link style globally, in which case all links have the same kind of shape, or locally on each link, in which case different link shapes occur in the same drawing.

Global link style

To specify the global link style:

In Java

Use the method:

```
void setGlobalLinkStyle(int style);
```

The valid values for `style` are:

◆ `IlvTreeLayout.NO_RESHAPE_STYLE`

None of the links is reshaped in any manner.

◆ `IlvTreeLayout.STRAIGHT_LINE_STYLE`

All the intermediate points of the links (if any) are removed. This is the default value. See *Tree layout in free layout mode with center alignment and flow direction to the right* and *Tree layout in radial layout mode with aspect ratio 1.5* as examples.

◆ `IlvTreeLayout.ORTHOGONAL_STYLE`

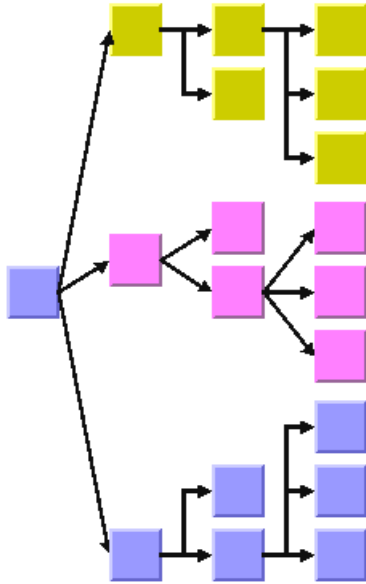
The links are reshaped in an orthogonal form (alternating horizontal and vertical segments). See *Tree layout with flow direction to the bottom, orthogonal link style, and tip-over alignment at some leaf nodes* and *Tip-over alignment* as examples.

◆ `IlvTreeLayout.MIXED_STYLE`

Each link can have a different link style. The style of each individual link can be set to have different link shapes occurring on the same graph.

Individual link style

All links have the same style of shape unless the global link style is `MIXED_STYLE`. Only when the global link style is set to `MIXED_STYLE` can each link have an individual link style.



Different Link Styles Mixed in the Same Drawing

To specify the style of an individual link:

In Java

Use the methods:

```
setLinkStyle(java.lang.Object, int)
```

```
getLinkStyle
```

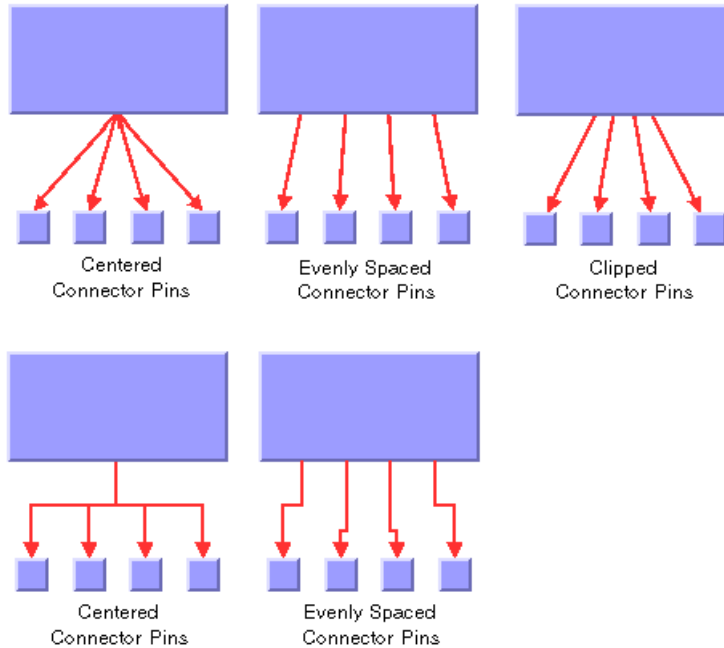
The valid values for style are:

- ◆ `IlvTreeLayout.STRAIGHT_LINE_STYLE` (the default)
- ◆ `IlvTreeLayout.NO_RESHAPE_STYLE`
- ◆ `IlvTreeLayout.ORTHOGONAL_STYLE`

Connector style

The layout algorithm automatically positions the end points of links (the connector pins) at the nodes. The connector style parameter specifies how these end points are calculated for the outgoing links at the parent node.

By default, the connector style determines how the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side.



Connector styles

To specify the connector style:

In Java™

Use the method:

```
void setConnectorStyle(int style);
```

The valid values for `style` are:

◆ `IlvTreeLayout.CENTERED_PINS`

The end points of the links are placed in the center of the border where the links are attached.

◆ `IlvTreeLayout.CLIPPED_PINS`

Each link pointing to the center of the node is clipped at the node border. The connector pins are placed at the points on the border where the links are clipped. This style affects straight links. It behaves like centered connector pins for orthogonal links.

◆ `IlvTreeLayout.EVENLY_SPACED_PINS`

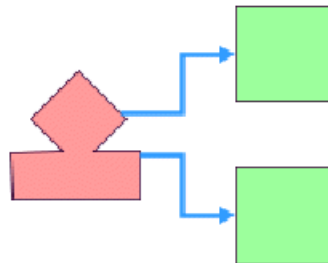
The connector pins are evenly distributed along the node border. This style works for straight and orthogonal links.

◆ `IlvTreeLayout.AUTOMATIC_PINS`

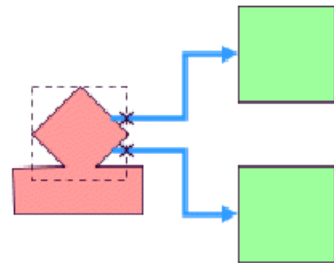
The connector style is selected automatically depending on the link style and the layout mode. In the nonradial modes, the algorithm always chooses centered pins. In the radial layout modes, it chooses clipped pins.

The connector style, the link connection box interface, and the link clip interface work together in the following way: by respecting the connector style, the proposed connection points are calculated on the rectangle obtained from the link connection box interface (or on the bounding box of the node, if no link connection box interface was specified). Then, the proposed connection point is passed to the link clip interface and the returned connection points are used to connect the link to the node.

The following figure shows an example of the combined effect.



Clipping at the node bounding box



Clipping at a specified connection box

Combined effect of link clipping interface and link connection box

If the links are clipped at the red node in the previous figure (left), they appear unsymmetrical with respect to the node shape, because the relevant part of the node (here: the upper rhombus) is not in the center of the bounding box of the node, but the proposed connection points are calculated with respect to the bounding box. This can be corrected by using a link connection box interface to explicitly specify a smaller connection box for the relevant part of the node (previous figure, right) such that the proposed connection points are placed symmetrically at the upper rhombus of the node.

Using a link connection box interface

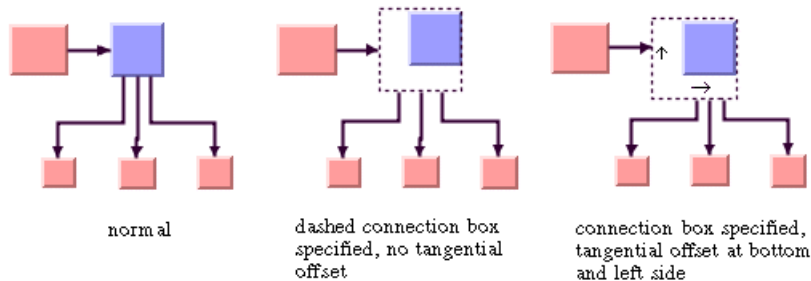
Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, possibly in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*. To set a link connection box interface in Java™, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface interface)
```

The link connection box interface provides each node with a link connection box and a tangential shift offset that defines how much the connection points are “shifted” tangentially depending on which side the links connect.

The following figure illustrates the effects of customizing the connection box when the connector style is evenly spaced.



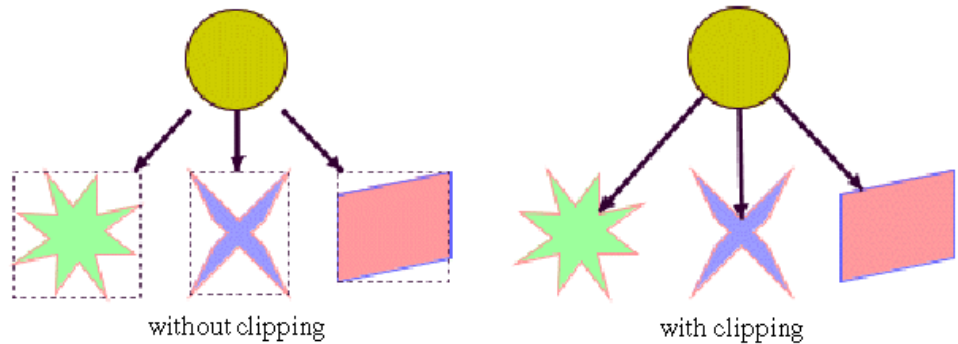
Effect of connection box interface

On the left is the result without any connection box interface. The middle picture shows the effect if the connection box interface returns the dashed rectangle for the blue node but the tangential offset at all sides of the node is 0. Notice that the outgoing links are spaced according to the dashed rectangle, which appears too wide for the blue node in this situation. The picture on the right shows the effect of the connection box interface if, in addition, a positive tangential offset was specified for the bottom side and a negative offset was specified for the left side of the blue node.

Using a link clipping interface

By default, the Tree Layout places the connection points of links at the border of the bounding box of the nodes.

If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



Effect of link clipping interface

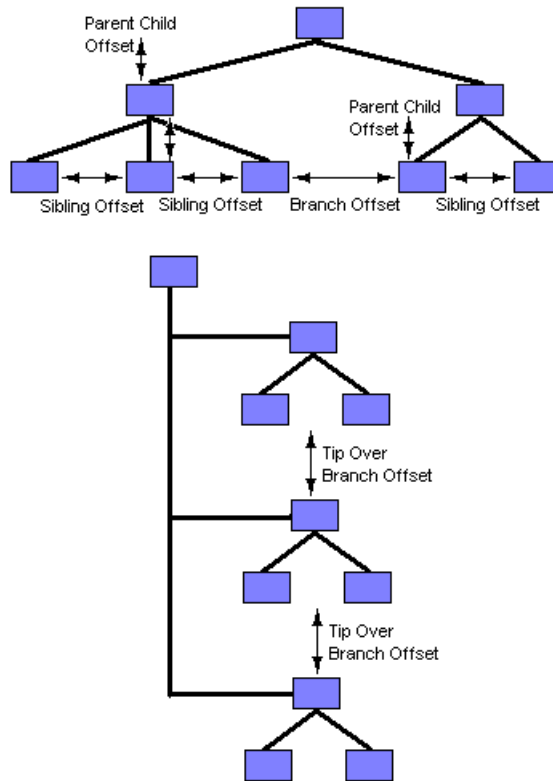
You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*. To set a link clip interface in Java™, call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

Spacing parameters

The spacing of the layout is controlled mainly by three spacing parameters: the distance between a parent and its children, the minimum distance between siblings, and the minimum distance between nodes of different branches. For instance, if the flow direction is to the top or bottom, the offset between parent and children is vertical, while the sibling offset and the branch offset are horizontal.

For tip-over alignment, an additional spacing parameter is provided: the minimum distance between branches starting at a node with tip-over alignment. This offset is always orthogonal to the normal branch offset. If the flow direction is to the top or bottom, the tip-over branch offset is vertical.



Spacing parameters

To specify the spacing parameters:

In Java

In Java™, use the methods:

```
void setParentChildOffset(float offset);
```

```
void setSiblingOffset(float offset);
```

```
void setBranchOffset(float offset);
```

```
void setTipOverBranchOffset(float offset);
```

For experts: additional spacing parameters

The spacing parameters normally specify the minimal offsets between the node borders. Hence, the layout algorithm places the nodes such that they do not overlap. You can also specify that the layout should ignore the node sizes.

In Java

In Java, call:

```
layout.setRespectNodeSizes (false);
```

In this case, the spacing parameters are interpreted as the minimum distances between the node centers, and the node sides are not taken into account during the layout. However, if the specified offset parameters are now smaller than the node size, the nodes and links will overlap. This often happens with orthogonal links in particular. It makes sense to use this option only if all nodes have approximately the same size, all links are straight, and the spacing parameters are larger than the largest node.

If the link style is orthogonal, the shape of the links from the parent to its children looks like a fork (see *Different Alignments Mixed in the Same Drawing*). The position of the bend points in this shape can be influenced by the *orthogonal fork percentage*, a value between 0 and 100. This is a percentage of the parent child offset. If the orthogonal fork percentage is 0, the link shape forks directly at the parent node. If the percentage is 100, the link shape forks at the child node. A good choice is between 25 and 75. This percentage can be set.

In Java

Use the method:

```
void setOrthForkPercentage(float percentage);
```

If the link style is not orthogonal, links may overlap neighboring nodes. This happens only in a very few cases, for instance, when a link starts at a very small node that is neighbored by a very large node. This deficiency can be fixed by increasing the branch offset. However, this influences the layout globally, affecting nodes without that deficiency. To avoid a global change, you can change the *overlap percentage* instead, which is a value between 0 and 100. This value is used by an internal heuristic of the layout algorithm that considers a node to be smaller by this percentage. The default percentage is 30. This usually results in better usage of the space. However, if very small nodes are neighbored to very large nodes, it is recommended to decrease the overlap percentage or to set it to 0 to switch this heuristic off to avoid links overlapping nodes.

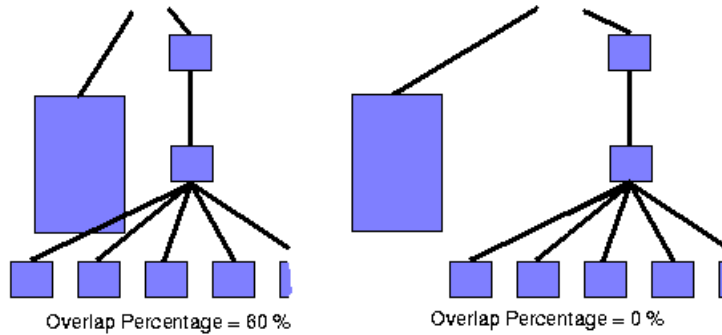
To set the overlap percentage: :

In Java

Use the method:

```
void setOverlapPercentage(float percentage);
```

Note: It is recommended that you always set the orthogonal fork percentage to a value larger than the value of the overlap percentage.



Effect of using the overlap percentage

Level layout mode

Describes how the level layout mode organizes nodes and describes the parameters of this mode.

In this section

Overview

Describes how the level layout mode organizes nodes.

General parameters

Describes the layout parameters of the level layout mode.

Level alignment

Describes the level alignment parameters of the level layout mode.

Overview

The level layout mode partitions the node into levels and arranges the levels horizontally or vertically. The root is placed at level 0, its children at level 1, the children of those children at level 2, and so on. In contrast to the free layout mode, in level layout mode the nodes of the same level are justified with each other even if they are not siblings (that is, they do not have the same parent).

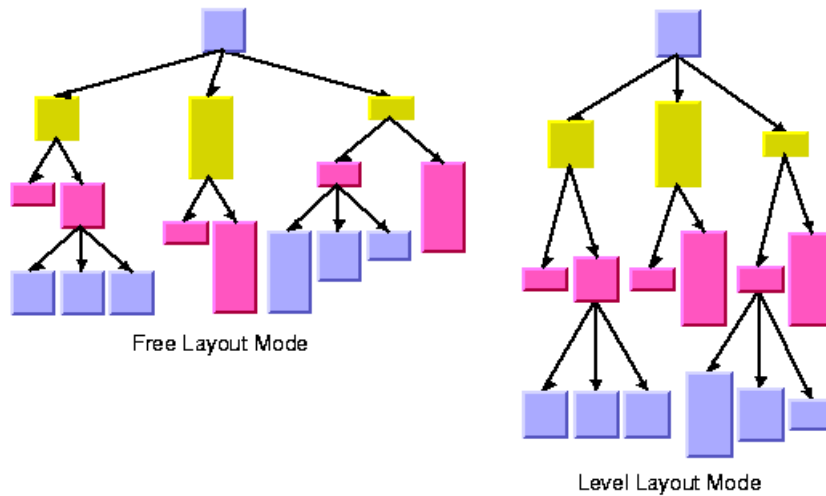
To set the level layout mode:

In Java

In Java™, call:

```
layout.setLayoutMode(IlvTreeLayout.LEVEL);
```

The following figure shows the same graph in free layout mode and in level layout mode.



Free layout mode and level layout mode

General parameters

Most layout parameters that work for the free layout mode work as well for the level layout mode. You can set the flow direction, the spacing offsets, the global or individual link style, and the global or individual alignment. See *Free layout mode* for details.

The differences from the free layout mode are:

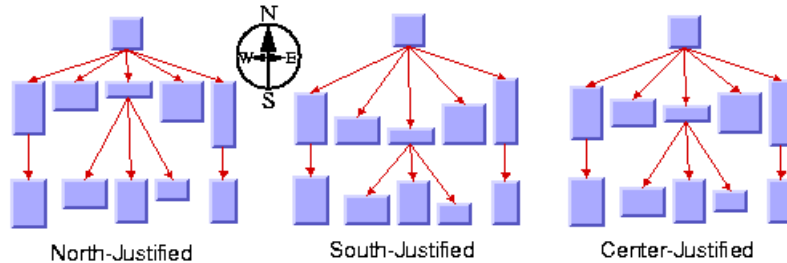
- ◆ The tip-over alignment does not work in level layout mode.
- ◆ The parent-child offset parameter controls the spacing between the levels. In level layout mode, it is the *minimum* distance between parent and its children, while in free layout mode, it is the *exact* distance between parent and its children.
- ◆ The overlap percentage has no effect in level layout mode.

Level alignment

In level layout mode with flow direction to the top or bottom, the nodes are organized in horizontal levels such that the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

In flow direction to the left or right, the nodes are organized in vertical levels approximately at the same x-coordinate. The nodes of the same level can be justified at the left border, at the right border, or at the center.

To distinguish the level alignment independently from the flow direction, the directions north and south are used (see *Using compass directions for positional layout parameters (TL)*). The north border of a node is the border that is closer to the level where its parent is placed, and the south border of a node is the border that is closer to the level where its children are placed. If the flow direction is to the bottom, the level alignment north means that the nodes are justified at the top border, and south means that the nodes are justified at the bottom border. If the flow direction is to the top, north and south are inverted: north means the bottom border and south means the top border. If the flow direction is to the right, then north means the left border and south means the right border.



Level Alignment

To specify the level alignment:

In Java

In Java™, use the method:

```
void setLevelAlignment(int alignment);
```

The valid values for `alignment` are:

- ◆ `IlvTreeLayout.CENTER` (the default)
- ◆ `IlvTreeLayout.NORTH`
- ◆ `IlvTreeLayout.SOUTH`

Radial layout mode

Describes how the radial layout mode organizes nodes and describes the parameters of this mode.

In this section

Overview

Describes how the radial layout mode organizes nodes.

General parameters

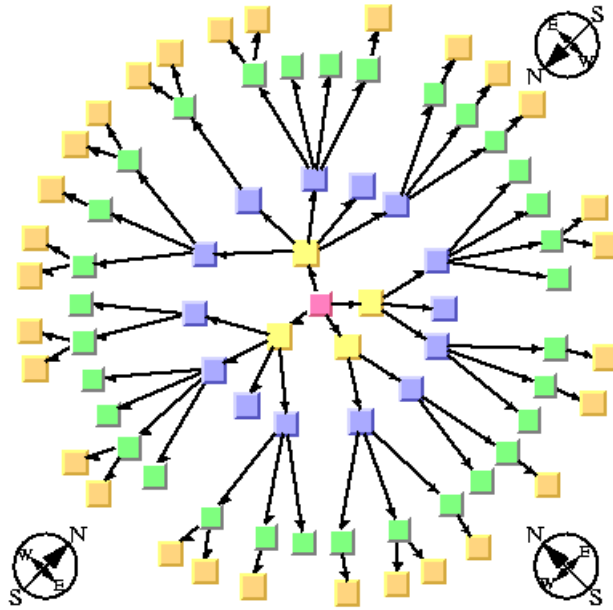
Describes the layout parameters of the radial layout mode.

Alternating radial mode

Describes how the alternating radial mode organizes nodes and describes the parameters of this mode.

Overview

The radial layout mode partitions the node into levels and arranges the levels in circles around the root node. *Radial layout mode* shows an example of the radial layout mode. The compass icons show the compass directions in this drawing.



Radial layout mode

To set the radial layout mode:

In Java

In Java™, call:

```
layout.setLayoutMode(IlvTreeLayout.RADIAL);
```

General parameters

Most layout parameters that work for the free and level layout mode work as well for the radial layout mode. You can set the spacing offsets, the level alignment, the global or individual link style, and the global or individual alignment. See *Free layout mode* and *Level layout mode* for details.

The radial layout mode differs from the other layout modes as follows:

- ◆ The tip-over alignment does not work in radial layout mode.
- ◆ The orthogonal link style does not work in radial layout mode.
- ◆ The clipped connector style is always used.
- ◆ The parent-child offset parameter controls the minimal distance between the circular levels. However, it is sometimes necessary to increase the offset between circular levels to obtain enough space on the circle to place all nodes of a level.
- ◆ The level alignment *north* indicates alignment at the inner border of the circular level (that is, towards the root), and the level alignment *south* indicates alignment at the outer border of the circular level (that is, away from the root).
- ◆ The level alignments *north* and *south* sometimes result in overlapping nodes.
- ◆ The overlap percentage has no effect in radial layout mode.

Alternating radial mode

Describes how the alternating radial mode organizes nodes and describes the parameters of this mode.

In this section

Overview

Describes how the alternating radial mode organizes nodes.

Aspect ratio

Describes the aspect ratio parameter of the alternating radial mode.

Spacing parameters

Describes the spacing parameters of the radial modes.

Tips and tricks

Describes some tips and tricks for expert users.

Overview

If levels of the graph contain many nodes, it is sometimes necessary to increase the radius of the circular level to provide enough space on the circumference of the circle for all the nodes. This may result in a considerable distance from the previous level. To avoid this, there is an *alternating* radial mode. The alternating radial mode places the nodes of a level alternating between two circles instead of one circle, resulting in better use of the space of the layout.

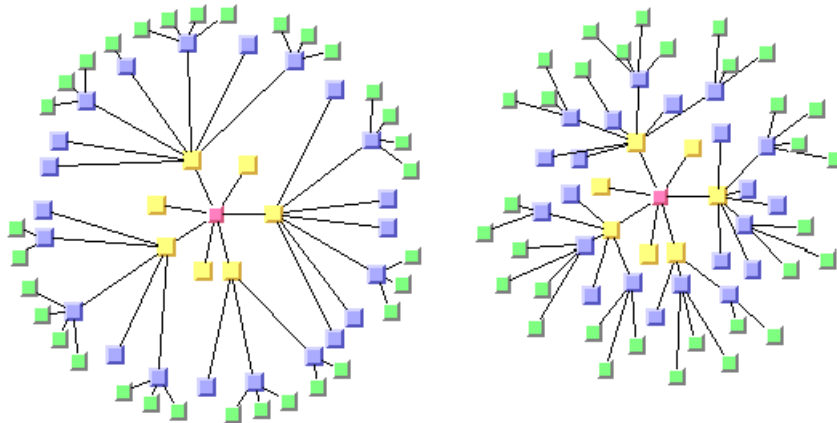
The alternating radial mode uses two circles only when necessary. For many small and light trees, there will be no difference from the normal radial mode. Only for large graphs with a large number of children will the alternating radial mode have an effect.

To set the alternating radial layout mode:

In Java

In Java™, call:

```
layout.setLayoutMode(IlvTreeLayout.ALTERNATING_RADIAL);
```



Radial layout mode (right) and alternating radial layout mode (left)

Aspect ratio

If the drawing area is not a square, arranging the levels as circles is not always the best choice. You can specify the aspect ratio of the drawing area to better fit the layout to the drawing area. In this case, the algorithm uses ellipses instead of circles. See *Tree layout in radial layout mode with aspect ratio 1.5* for an example.

To specify the aspect ratio:

In Java

In Java™, call:

```
void setAspectRatio(IlvRect rect);
```

If no rectangle is specified, you can calculate the aspect ratio from the width and height of the drawing area as `aspectRatio = width/height` and use the method:

```
void setAspectRatio(float aspectRatio);
```

Spacing parameters

The spacing parameters of the radial layout modes are controlled by the same methods as used for the free and level layout modes:

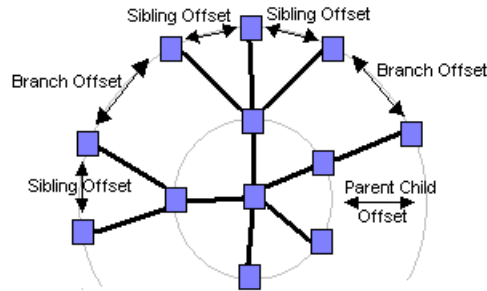
```
void setParentChildOffset(float offset);
```

```
void setSiblingOffset(float offset);
```

```
void setBranchOffset(float offset);
```

Note that the sibling and branch offsets are minimum distances tangential to the circles or ellipses, while the parent-child offset is a minimum distance radial to the circles or ellipses.

The following figure shows the spacing parameters in radial layout mode.



Spacing parameters in radial layout mode

Tips and tricks

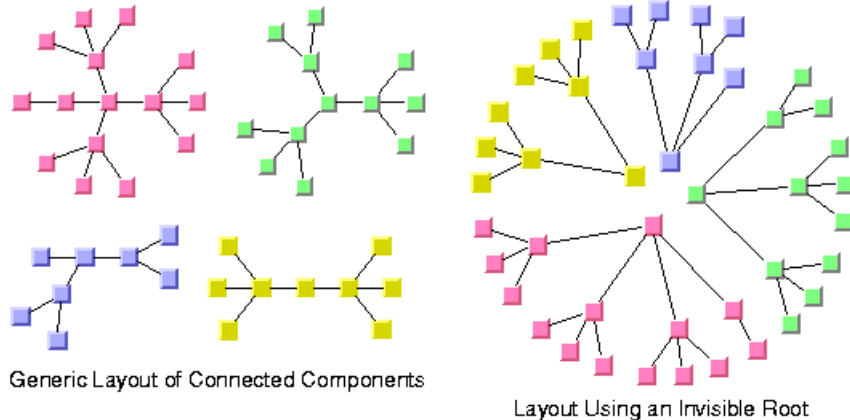
Adding an invisible root to the layout

If the graph contains several trees that are disconnected from each other, the layout places them individually next to each other. Each connected component has its own radial structure with circular layers. However, sometimes it is appropriate to fit all connected components into a single circular layer structure. Conceptually, this is done by adding an invisible root at the center and connecting all disconnected trees to this root. *Layout of connected components without and with an invisible root* shows the effect of using an invisible root. This works only if the generic mechanism to lay out connected components is switched off.

To add an invisible root to the layout:

In
Call:

```
layout.setLayoutOfConnectedComponentsEnabled(false);  
layout.setInvisibleRootUsed(true);
```



Layout of connected components without and with an invisible root

Even spacing for the first circle

The radial mode is designed to optimize the space such that the circles have a small radius and the overall space for the entire layout is small. To achieve this, the layout algorithm may create larger gaps on the inner circles for better space usage of the outer circles. This may produce unevenly spaced circles, most notably for the first circle where all nodes have the same parent node.

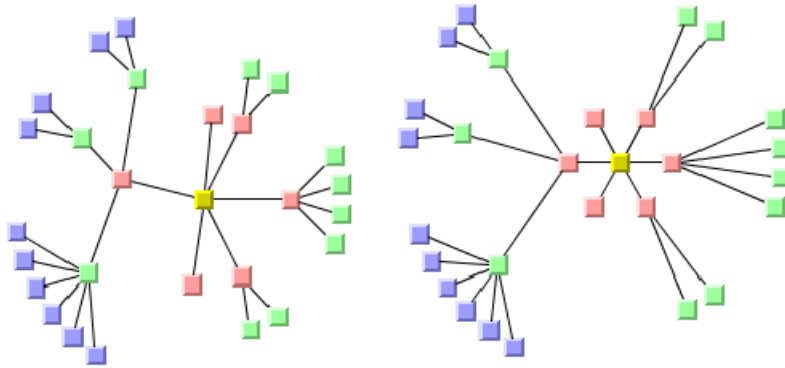
To avoid this effect, you can force the nodes to be evenly spaced on the entire first circle. Depending on the structure of the graph, this may cause the overall layout to waste more space on the other circles but may produce a more pleasing graph.

To enable even spacing:

In Java

In Java™, call:

```
layout.setFirstCircleEvenlySpacing (true);
```



Unevenly spaced first (red) circle

Evenly spaced first (red) circle

Evenly and Unevenly Spaced First Circle

For experts: forcing all levels to alternating

When the layout mode `ALTERNATING_RADIAL` is used, the layout checks whether the alternating node arrangement of a level saves space. If that does not save space, it uses the normal radial arrangement. Hence, for many sparse graphs, radial and alternating radial mode yield the same result because the alternating arrangement does not save space for any level. It is possible to disable the space check, that is, to perform an alternating arrangement for all levels even if this results in waste of space.

In Java

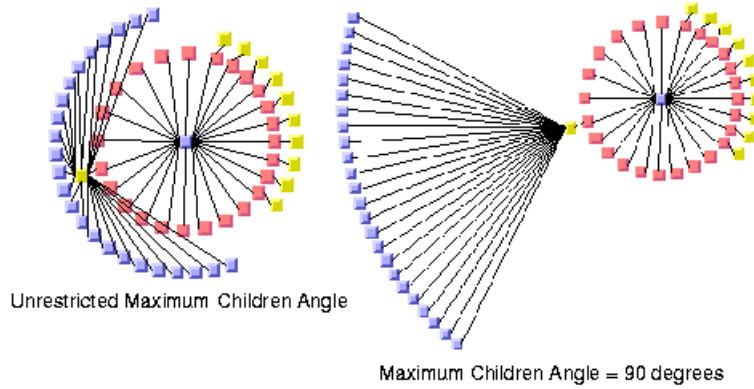
Call:

```
layout.setAllLevelsAlternating (true);
```

For experts: setting a maximum children angle

If a node has a lot of children, they may extend over a major portion of the circle and, therefore, are placed nearly 360 degrees around the node. This can result in links overlapping some nodes. The deficiency can be fixed by increasing the offset between parent and children. However, this affects the layout globally which means that nodes without the deficiency are also affected. To avoid a global change such as this, you can limit the maximum angle between the two rays from the parent (if it is not the root) to its two outermost children. This increases the offset between parent and children only where necessary.

In *Maximum Children Angle*, you can see in the layout on the left that many of the links overlap other nodes. In the layout on the right, you can see how this problem was solved by setting a maximum children angle between two rays from a parent to the two outermost children.



Maximum Children Angle

To set an angle in degrees:

In Java

Use the method:

```
void setMaxChildrenAngle(int angle);
```

Recommended values are between 30 and 180. Setting the value to 0 means the angle is unrestricted. The calculation of the angle is not very precise above 180 degrees or if the aspect ratio is not 1.0.

Tip-over layout modes

Drawing in radial layout mode and free layout mode can be adjusted according to the aspect ratio of the drawing area. To balance the height and depth of the drawing, free layout mode can also use tip-over alignment.

Tip-over alignment can be specified explicitly for individual nodes; the Tree Layout algorithm also has layout modes that automatically use tip-over alignment when needed. These are the tip-over layout modes.

The tip-over layout modes work as follows: Several trial layouts are performed in free layout mode. For each trial, tip-over alignment is set for certain individual nodes, while the specified alignment of all other nodes is preserved. The algorithm picks the trial layout that best fits the specified aspect ratio of the drawing area.

The aspect ratio can be set (see *Aspect ratio* in the Radial Layout Mode):

```
void setAspectRatio(IlvRect rect);
```

```
void setAspectRatio(float aspectRatio);
```

The tip-over modes are slightly more time-consuming than the other layout modes. For very large trees, it is recommended that you set the allowed layout time to a high value (for instance, 60 seconds) when using the tip-over modes.

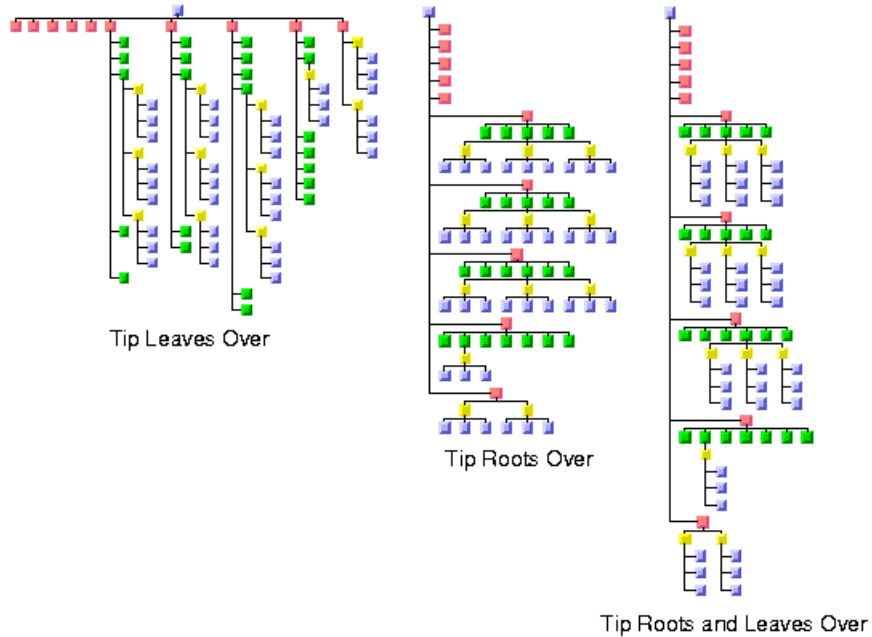
To set this mode:

In Java

Call:

```
layout.setAllowedTime(60000);
```

By using this call, you avoid running short of time for sufficient iterations of the layout algorithm. Because it would be too time-consuming to check all possibilities of tip-over alignment use, there are heuristics that check only certain trials according to the following different strategies, illustrated in the following figure.



Tip-over strategies

- ◆ *Tip leaves over*
- ◆ *Tip roots over*
- ◆ *Tip roots and leaves over*
- ◆ *Tip over fast*

Tip leaves over

To use this tip-over strategy, set the layout mode as follows:

In Java

```
layout.setLayoutMode(IlvTreeLayout.TIP_LEAVES_OVER);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the leaves, then the leaves and their parents, then additionally the parents of these parents, and so on. As a result, the nodes closest to the root use normal alignment and the nodes closest to the leaves use tip-over alignment.

Tip roots over

To use this tip-over strategy, set the layout mode as follows:

In Java

```
layout.setLayoutMode(IlvTreeLayout.TIP_ROOTS_OVER);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node, then the root and its children, then additionally the children of these children, and so on. As a result, the nodes closer to the leaves use normal alignment and the nodes closer to the root use tip-over alignment.

Tip roots and leaves over

To use this tip-over strategy, set the layout mode as follows:

In Java

```
layout.setLayoutMode(IlvTreeLayout.TIP_ROOTS_AND_LEAVES_OVER);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node and the leaves simultaneously; then the root and its children, and the leaves and its parent; then additionally the children of these children and the parents of these parents, and so on. As result, the nodes in the middle of the tree use normal alignment and the nodes closest to the root or leaves use the tip-over alignment.

This is the slowest strategy because it includes all trials of the strategy “*tip leaves over*” as well as all tries of the strategy “*tip roots over*.”

Tip over fast

The fast tip-over provides a compromise among all other strategies. The heuristic tries a small selection of the other strategies, not all possibilities. Therefore, it is the fastest strategy for large graphs.

To use this strategy, set the layout mode as follows:

In Java

```
layout.setLayoutMode(IlvTreeLayout.TIP_OVER);
```

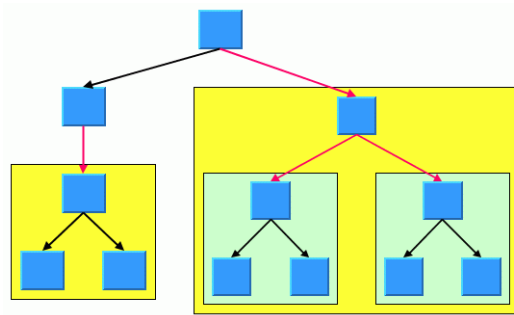
It is possible that all four strategies yield the same result because the strategies are not disjoint; that is, certain trials are performed in all four strategies. In addition, the tip-over modes do not necessarily produce the optimal layout that gives the best possible fit to the aspect ratio. The reason is that some unusual configurations of tip-over alignment are never tried because doing so would cause the running time to be too high.

Recursive mode

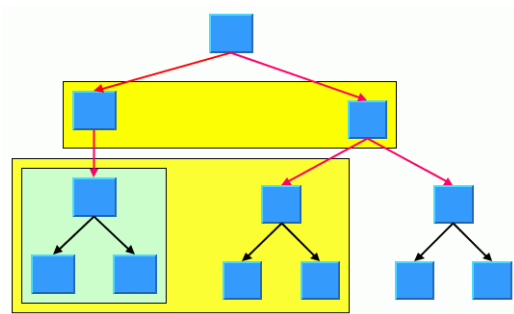
JViews Diagrammer supports nested graphs, that is, it can render graphs containing nodes that are graphs. A graph that is a node in another graph is called a subgraph. Links that connect nodes of different subgraphs are called intergraph links. In *Leaf recursive tree*, all red links are intergraph links and all black links are normal links. This is explained in detail in *Nested layouts*.

The tree layout can treat a nested graph in specific situation at once and route the intergraph links as well as the normal links that belong to the tree. It can handle a leaf-recursive tree at once. A leaf recursive tree has the following properties:

- ◆ it is a tree
- ◆ only leaf nodes of the tree can contain nested graphs
- ◆ the root node of the tree nested in a leaf node is connected by a link to the parent node of the leaf



Leaf recursive tree



Non leaf recursive tree

This graph is not a leaf recursive tree: the subgraphs are not nested in the leaves of the tree. The graph cannot be handled by the tree layout in recursive mode, but it can be handled by

the hierarchical layout in recursive mode. If the graph is a leaf recursive tree and the layout mode is not the radial layout mode, the tree layout can handle the nested graph at once.

To enable the recursive mode:

In Java

In Java™, use the method:

```
void setRecursiveLeafLayoutMode(boolean enable);
```

and call `performLayout` with the third parameter set to `true` as follows:

```
layout.performLayout(force, redraw, true);
```

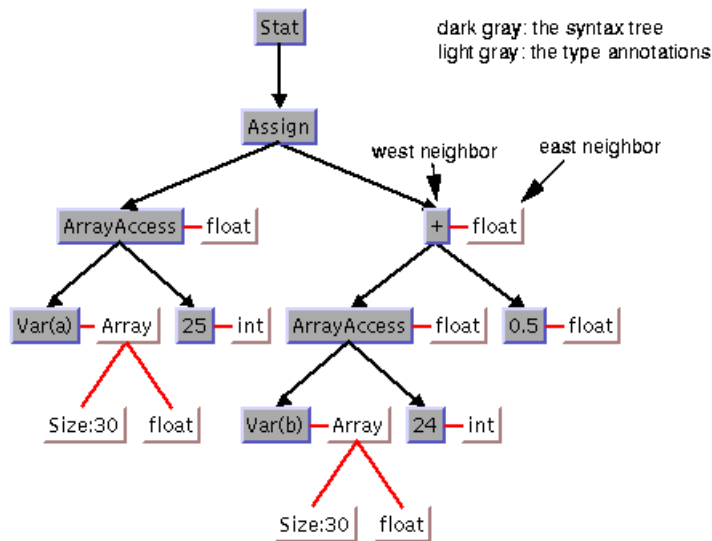
For experts: additional tips for the TL

Specifying east-west neighbors

You can specify that two unrelated nodes must be direct neighbors in a direction perpendicular to the flow direction. In the level and radial layout modes, the nodes are placed in the same level next to each other. In the free layout and tip-over modes, the nodes are placed aligned at the north border. Such nodes are called *east-west neighbors* because one node is placed as the direct neighbor on the east side of the other node. The other node becomes the direct neighbor on the west side of the first node. (See also *Using compass directions for positional layout parameters (TL)*).

Technically, the nodes are treated as parent and child, even if there may be no link between them. Therefore, one of the two nodes can have a real parent, but the other node should not because its virtual parent is its *east-west neighbor*.

The east-west neighbor feature can be used, for example, for annotating nodes in a typed syntax tree occurring in compiler construction. *Annotated Syntax Tree of Statement $a[25] = b[24] + 0.5$* ; shows an example of such a tree.



Annotated Syntax Tree of Statement $a[25] = b[24] + 0.5$;

To specify that two nodes are east-west neighbors, use the method:

```
void setEastWestNeighboring(Object eastNode, Object westNode);
```

You can also use the following method, which is identical except for the reversed parameter order:

```
void setWestEastNeighboring(Object westNode, Object eastNode);
```

If the flow direction is to the bottom, the latter method may be easier to remember because, in this case, west is to the left of east in the layout, which is similar to the text flow of the parameters.

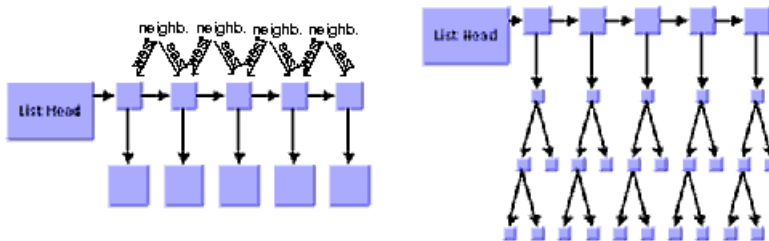
To obtain the node that is the east or west neighbor of a node, use the calls:

```
Object getEastNeighbor(Object node);
```

```
Object getWestNeighbor(Object node);
```

Note that each node can have at most one east neighbor and one west neighbor because they are *direct* neighbors. If more than one direct neighbor is specified, it is partially ignored. Cyclic specifications can cause conflict as well. For instance, if node B is the east neighbor of node A and node C is the east neighbor of B, then node A cannot be the east neighbor of C. (Strictly speaking, such cycles could be technically possible in some situations in the radial layout mode, but nonetheless they are not allowed in any layout mode.)

If B is the east neighbor of A, then A is automatically the west neighbor of B. On the other hand, the east neighbor of A can itself have another east neighbor. This allows the creation of chains of east-west neighbors, which is a common way to visualize lists of trees. Two examples are shown in *Chains of east-west neighbors to visualize lists of trees*.



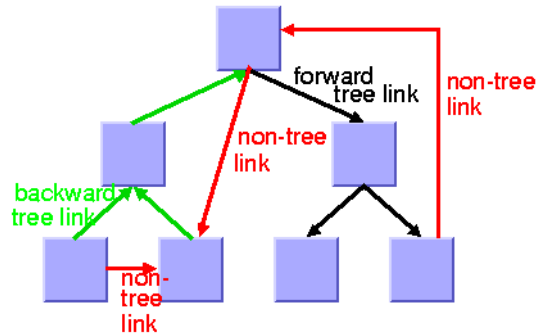
Chains of east-west neighbors to visualize lists of trees

Retrieving link categories

The Tree Layout algorithm works on a spanning tree, as mentioned in a *The TL algorithm*. If the graph to be laid out is not a pure tree, the algorithm ignores some links. To treat such links in a special way, you can obtain a list of nontree links.

Because there are parents and children in the spanning tree, the following link categories must be distinguished:

- ◆ A forward tree link is a link from a parent to its child.
- ◆ A backward tree link is a link from a child to its parent. If the link is drawn as a directed arrow, the arrow will point in the opposite direction to the flow direction.
- ◆ A nontree link is a link between two unrelated nodes; neither one is a child of the other.



Link categories

The layout algorithm uses these link categories internally but does not store them permanently to save time and ensure memory efficiency. If you want to treat some link categories in a special way (for example, to call the Link Layout on the nontree links), you must specify *before the layout* that you want to access the link categories *after the layout*. To do this, use the method `setCategorizingLinks(boolean)` in the following way:

```
layout.setCategorizingLinks(true);
// now perform a layout
layout.performLayout();
// now you can access the link categories
```

After the layout, the link categories can be obtained by the methods:

```
getCalcForwardTreeLinks()
getCalcBackwardTreeLinks()
getCalcNonTreeLinks()
```

The link category data gets filled each time the layout is called, unless you set the method `setCategorizingLinks(boolean)` back to false.

Sequences of layouts with incremental changes

You can work with trees that have become out-of-date, for example, those that need to be extended with more children. If you perform a layout after an extension, you probably want to identify the parts that had already been laid out in the original graph. The Tree Layout algorithm supports these incremental changes in incremental mode because it takes the previous positions of the nodes into account. It preserves the relative order of the children in the subsequent layout.

In nonincremental mode, the Tree Layout algorithm calculates the order of the children from the node order given by the attached graph model (or grapher). In this case, the layout is independent from the positions of the nodes before layout. It does not preserve the relative order of the children in subsequent layouts.

The incremental mode is enabled by default.

To disable the incremental mode:

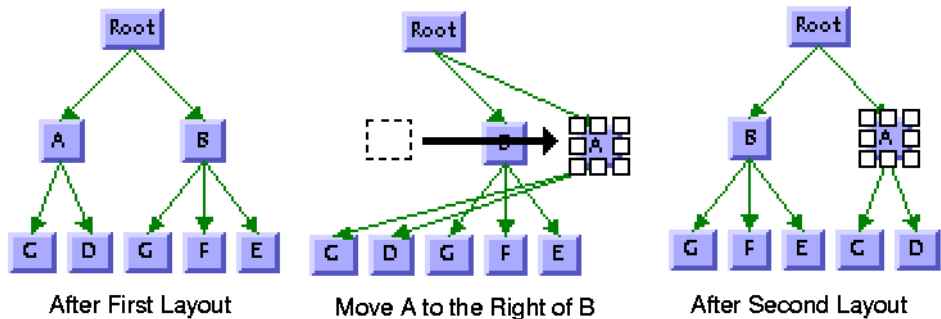
In Java

Call:

```
layout.setIncrementalMode (false);
```

Interactive editing

The fact that the relative order of the layout is preserved is particularly useful during interactive editing. It allows you to correct the layout easily. For instance, if the first layout places a node A left to its sibling node B but you need to reverse the order, you can simply move node A to the right of node B and start a new layout to clean up the drawing. In the second layout, A remains to the right of B, and the subtree of A will “follow” node A.



Interactive Editing to Achieve a Specific Order of Children

Specifying the order of children

Some applications require a specific relative order of the children in the tree. This means that, for instance, when the flow direction is to the bottom, which child must be placed to the left of another child. Even if the graph has never been laid out, you can use the coordinates to specify a certain order of the children at a node. You can use the following:

- ◆ First, make sure that the incremental mode is enabled.
- ◆ In free and level layout modes with flow direction to the bottom or top, determine the maximal width w of all nodes. Simply move the child that should be in the leftmost position to the coordinate $(0, 0)$, and the child that should get the i th relative position (in order from left to right) to the coordinate $((w+1) * i, 0)$.
- ◆ If the flow direction is to the left or to the right, determine the maximal height h of all nodes. Move the child that should be in the topmost position to the coordinate $(0, 0)$ and the child that should get the i th relative position (in the order from top to bottom) to coordinate $(0, (h+1) * i)$.
- ◆ In the radial layout modes, determine the maximal diagonal $D = w^2 + h$ of all nodes. If the position of the parent is (x, y) before the layout, move the child that should be the

first in the circular order to the coordinate $(x, y+D)$ and the child that should get the i th relative position in the circular order to coordinate $(x+D*i, y+D)$.

If you want to specify a relative order for all nodes in radial layout mode, you must do this for the parents before you do it for the children. In this case, moving the children can be performed easily during a depth-first traversal from the root to the leaves.

The layout that is performed after moving the children arranges the children with the relative order.

Hierarchical Layout (HL)

Describes the *Hierarchical Layout* algorithm (class `IlvHierarchicalLayout` from the package `ilog.views.graphlayout.hierarchical`).

In this section

General information on the HL

Provides samples of the layout and explains where it is likely to be used.

Features and limitations of the HL

Lists the features and limitations of the Hierarchical Layout (HL).

The HL algorithm

Gives an explanation of the Hierarchical Layout (HL) algorithm and a sample.

Generic features and parameters of the HL

Lists the generic features and parameters of the Hierarchical Layout (HL).

Specific parameters of the HL

Describes the specific parameters supported by HL (the `IlvHierarchicalLayout` class) and gives samples of their use.

Incremental mode with HL

Describes how to apply hierarchical layouts sequentially to the same graph.

Layout constraints for HL

Describes the constraints on the relative positions of nodes available with the Hierarchical Layout (HL).

Adding and removing constraints in Java for HL

Describes how to specify constraints in Java™ .

Level range constraints (HL)

Explains how modes are partitioned into levels and how to set constraints at a specific level.

Level index parameter (HL)

Describes how to force a node to a particular level with the level index parameter constraint.

Same level constraints (HL)

Describes how to force several nodes to be at the same level.

Group spread constraints (HL)

Describes how to force a group of nodes to the same level.

Relative level constraints (HL)

Describes how to force a node into a higher level than another node.

Position index parameter (HL)

Describes how to use the position index parameter.

Relative position constraints (HL)

Describes how to use relative position constraints.

Side-by-side constraints (HL)

Describes how to use side-by-side constraints.

Extremity constraints (HL)

Describes how to use extremity constraints.

Swim lane constraints (HL)

Describes how to use swim lane constraints.

Constraint priorities (HL)

Discusses constraint priorities.

For experts: constraint validation (HL)

Discusses how validation is done during layout and how to force it if necessary.

For experts: more indices (HL)

Describes how to specify level and position indices or retrieve calculated indices.

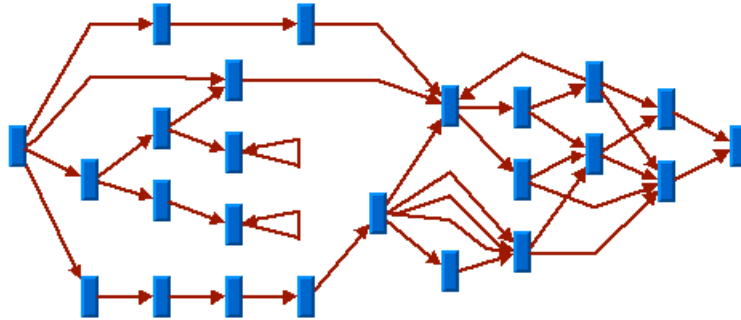
Recursive layout

Explains the recursive mode supported by the hierarchical layout.

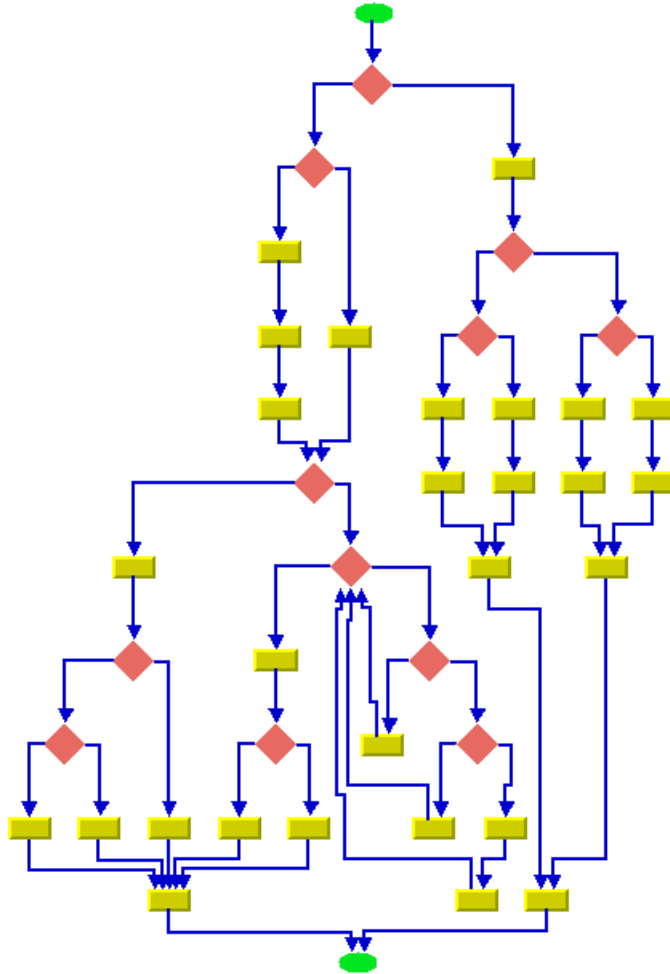
General information on the HL

HL samples

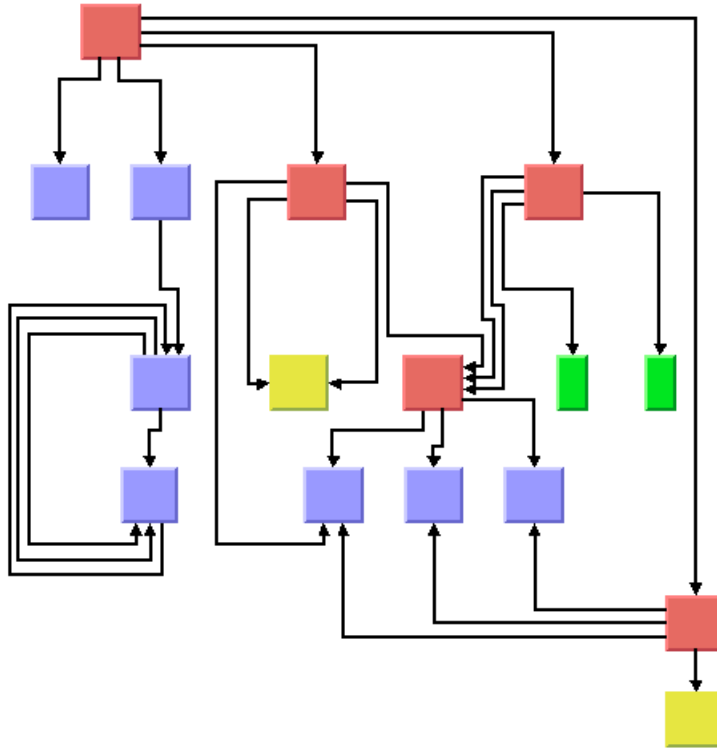
Here are some sample drawings produced with the Hierarchical Layout:



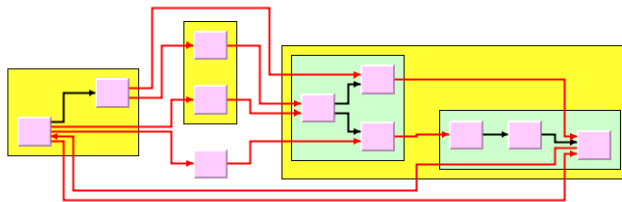
Sample layout with self-loops, multiple links, and cycles



Flowchart with orthogonal link style



Sample layout with ports and orthogonal link style



Sample layout of nested graph in recursive layout mode

What types of graphs suit the HL?

Any type of graph:

- ◆ Preferably graphs with directed links. A directed link has a direction from source node to target node and is usually drawn with an arrow. The algorithm takes the link directions into account..
- ◆ connected graphs and disconnected graphs

- ◆ planar graphs and nonplanar graphs
- ◆ nested graphs with intergraph links

Application domains for the HL

Application domains for the Hierarchical Layout include:

- ◆ Electrical engineering (logic diagrams, circuit block diagrams)
- ◆ Industrial engineering (industrial process diagrams, schematic design diagrams)
- ◆ Business processing (workflow diagrams, process flow diagrams, PERT charts)
- ◆ Software management/software (re-)engineering (UML diagrams, flowcharts, data inspector diagrams, call graphs)
- ◆ Database and knowledge engineering (database query graphs)
- ◆ CASE tools (designs diagrams)

Features and limitations of the HL

Features

- ◆ Organizes nodes without overlaps in horizontal or vertical levels.
- ◆ Arranges the graph such that the majority of links are short and flow uniformly in the same direction (from left to right, from top to bottom, and so on).
- ◆ Reduces the number of link crossings. Most of the time, produces drawings with no crossings or only a small number of crossings.
- ◆ Often produces balanced drawings that emphasize the symmetries in the graph.
- ◆ Supports self-links (that is, links with the same origin and destination node), multiple links between the same pair of nodes, and cycles.
- ◆ Efficient, scalable algorithm. Produces a nice layout for most sparse and medium-dense graphs relatively quickly, even if the number of nodes is very large.
- ◆ Provides several alignment and offset options.
- ◆ Supports port specifications where links attach the nodes. Allows you to specify which side of a node (top, bottom, left, right) a link can be connected to or to specify which relative port position should be used for the connection.
- ◆ Supports layout constraints. Allows you to specify relative positional constraints, for instance, that a node is above another node or left of another node.
- ◆ Incremental and nonincremental mode. In incremental mode, the previous position of nodes are taken into account. Positions the nodes without changing the relative order of the nodes so that the layout is stable on incremental changes of the graph.
- ◆ Can handle flat and nested graphs. In recursive layout mode, it routes the intergraph links of nested graphs and places the labels of nodes and links in subgraphs.
- ◆ The computation time depends on the number of nodes, the number of levels, and the number of links that cross several levels. Most of the time, the links are placed between adjacent levels, which keeps the computation time small.

Limitations

- ◆ The algorithm tries to minimize the number of link crossings (which is generally an NP-complete problem). It is mathematically impossible to solve this problem quickly for any graph size. Therefore, the algorithm uses a very fast heuristic that obtains a good layout, but not always with the theoretical minimum number of link crossings.
- ◆ The algorithm tries to place the nodes such that all links point uniformly in the same direction. It is impossible to place cycles of links in this way. For this reason, it sometimes produces a graph where a small number of links are reversed to point into the opposite direction. The algorithm tries to minimize the number of reversed links (which, again, is an NP-complete problem). Therefore, the algorithm uses a very fast heuristic resulting in a good layout, but not always with the theoretical minimum number of reversed links.

- ◆ The computation time required to obtain an appropriate drawing depends most significantly on the number of bends in the links. Since the algorithm places one bend whenever a link crosses a level, the number of bends can grow relatively quickly if the layout requires many long links that span several levels. Therefore, the layout process may become very time-consuming for dense graphs (the number of links is relatively high compared to the number of nodes) or for graphs that require a large number of node levels.

The HL algorithm

A brief description of the HL algorithm

This algorithm works in four steps:

Step 1: Leveling

The nodes are partitioned into groups. Each group of nodes forms a level. The objective is to group the nodes in such a way that the links always point from a level with smaller index to a level with larger index.

Step 2: Crossing reduction

The nodes are sorted within each level. The algorithm tries to keep the number of link crossings small when, for each level, the nodes are placed in this order on a line (see *Level and position indices*). This ordering results in the relative position index of each node within its level.

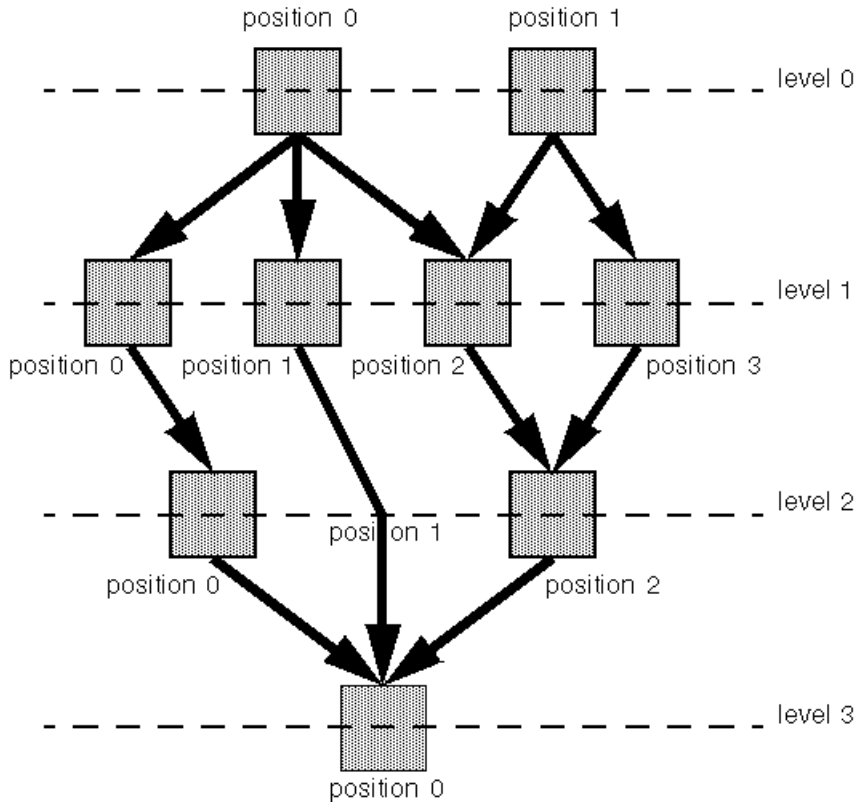
Step 3: Node positioning

From the level indices and position indices, balanced coordinates for the nodes are calculated. For instance, for a layout where the link flow is from top to bottom, the nodes are placed along horizontal lines such that all nodes belonging to the same level have (approximately) the same y-coordinate. The nodes of a level with a smaller index have a smaller y-coordinate than the nodes of a level with a higher index. Within a level, the nodes with a smaller position index have a smaller x-coordinate than the nodes with a higher position index.

Step 4: Link routing

The shapes of the links are calculated such that the links bypass the nodes at the level lines. In many cases, this requires that a bend point be created whenever a link needs to cross a level line. In a top-to-bottom layout, these bend points have the same y-coordinate as the level line they cross. (Note that these bend points also obtain a position index).

Level and position indices shows how the Hierarchical Layout algorithm uses the level and position indices to draw the graph.



Level and position indices

You can set parameters for the steps of the layout algorithm in several ways. For instance, you can specify the level index that the algorithm should choose for a node in Step 1 or the relative node position within the level in Step 2. You can also specify the justification of the nodes within a level and the style of the link shapes.

Example of HL

In Java

Below is a code sample that uses the `IlvHierarchicalLayout` class. This code sample shows how to perform a Hierarchical Layout:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.runtime.hierarchical.*;
...
```

```
IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);
try {
    IlvGraphLayoutReport layoutReport = layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
layout.detach();
graphModel.dispose();
```

Generic features and parameters of the HL

Overview of generic features

The `IlvHierarchicalLayout` class supports the following generic features defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- ◆ *Allowed time (HL)*
- ◆ *Layout of connected components (HL)*
- ◆ *Link clipping (HL)*
- ◆ *Link connection box (HL)*
- ◆ *Percentage of completion calculation (HL)*
- ◆ *Preserve fixed links (HL)*
- ◆ *Preserve fixed nodes (HL)*
- ◆ *Stop immediately (HL)*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed time (HL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of connected components (HL)

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of connected components*.) When using this mechanism, each component is laid out in its own individual level structure. Nodes of the first level of one component may be placed at a different position than nodes of the first level of another component.

The generic mechanism to layout connected components is, however, switched off by default. In this case, the layout algorithm can still handle disconnected graphs. It merges all components into a global level structure.

Link clipping (HL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface (HL)* for details of the link clipping mechanism.

Link connection box (HL)

The layout algorithm can use a link connection box interface (see *Link connection box*) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a link connection box interface (HL)*.

Percentage of completion calculation (HL)

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of the layout. (For a detailed description of this features, see *Percentage of completion calculation* and *Graph layout event listeners*.)

Preserve fixed links (HL)

The layout algorithm does not reshape the links that are specified as fixed. In fact, fixed links are completely ignored. (For more information on link parameters in the `IlvGraphLayout` class, see *Preserve fixed links* and *Link style*.)

Preserve fixed nodes (HL)

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see *Preserve fixed nodes*.) Moreover, the layout algorithm ignores fixed nodes completely and also does not route the links that are incident to the fixed nodes. This can result in unwanted overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Stop immediately (HL)

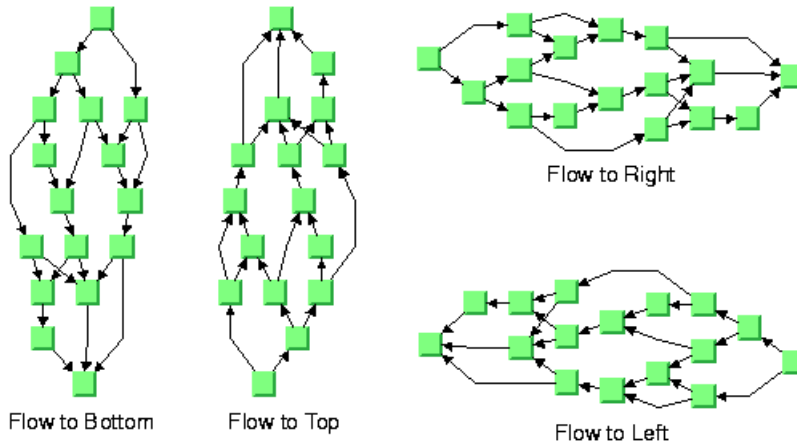
The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the HL

Flow direction (HL)

The flow direction parameter specifies the direction in which the majority of the links should point. If the flow direction is to the top or to the bottom, the node levels are oriented horizontally and the links mostly vertically. If the flow direction is to the left or to the right, the node levels are oriented vertically and the links mostly horizontally.

If the flow direction is to the bottom, the nodes of the level with index 0 are placed at the top border of the drawing. The nodes with level index 0 are usually the root nodes of the drawing (that is, the nodes without incoming links). If the flow direction is to the top, the nodes with level index 0 are placed at the bottom border of the drawing. If the flow direction is to the right, the nodes are placed at the left border of the drawing.



Flow directions

To specify the flow direction towards the bottom:

In Java

In Java, use the method:

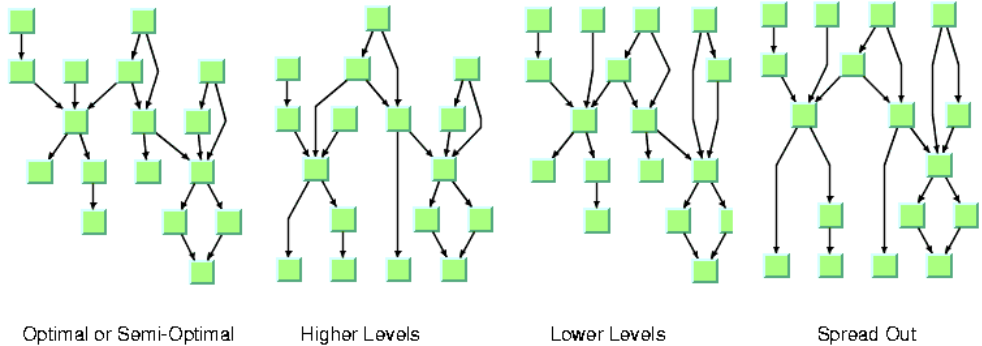
```
void setFlowDirection(int direction)
```

The valid values for the flow direction are:

- ◆ `IlvDirection.Right` (the default)
- ◆ `IlvDirection.Left`
- ◆ `IlvDirection.Bottom`
- ◆ `IlvDirection.Top`

Leveling strategy (HL)

The layout algorithm partitions the nodes into levels (see *A brief description of the HL algorithm*). The leveling strategy specifies how the levels are calculated. Besides the leveling strategy, layout constraints (see *Layout constraints for HL*), level indices (see *For experts: more indices (HL)*) as well as the incremental mode (see *Incremental mode with HL*) also affect the way the levels are calculated. If the incremental mode is disabled, the leveling strategy determines the levels of all nodes that are not subject to layout constraints and level index specifications.



Leveling strategies

To specify the leveling strategy:

In Java

In Java™, use the method:

```
void setLevelingStrategy(int strategy)
```

The valid values for the leveling strategy are:

◆ `IlvHierarchicalLayout.SEMI_OPTIMAL` (the default)

This produces often the same result as the optimal strategy, but it is quicker. The layout algorithm uses a heuristic to minimize the sum of level distances for all edges. It pulls root nodes to the highest-numbered possible level and leaf nodes to the lowest-numbered possible level.

◆ `IlvHierarchicalLayout.OPTIMAL`

This uses an algorithm that minimizes the sum of level distances for all edges. The optimal strategy is slower than the other strategies, but often produces the best result.

◆ `IlvHierarchicalLayout.HIGHER_LEVELS`

Nodes have a tendency to use the possible level with the highest level number. All leaf nodes will be at the highest-numbered level. All root nodes are pulled to high-numbered levels as much as possible.

◆ `IlvHierarchicalLayout.LOWER_LEVELS`

Nodes have a tendency to use the possible level with the lowest level number. All root nodes will be at level 0. All leaf nodes are pulled to low-numbered levels as much as possible.

◆ `IlvHierarchicalLayout.SPREAD_OUT`

This is a combination of the lower-level and higher-level strategies. All root nodes will be at level 0. All leaf nodes will be at the highest-numbered level. All inner nodes are at balanced positions.

Level justification (HL)

If the layout uses horizontal levels, the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, or the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

If the layout uses vertical levels, the nodes of the same level are placed approximately at the same x-coordinate. In this case, the nodes can be justified to be aligned at the left border, at the right border, or at the center of the nodes that belong to the same level.

To specify the level justification towards the top:

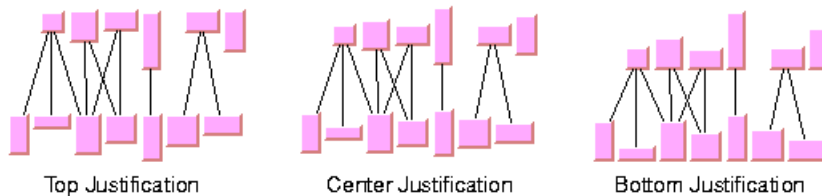
In Java

Use the method:

```
void setLevelJustification(int justification)
```

If the flow direction is to the top or to the bottom, the valid values for the level justification are:

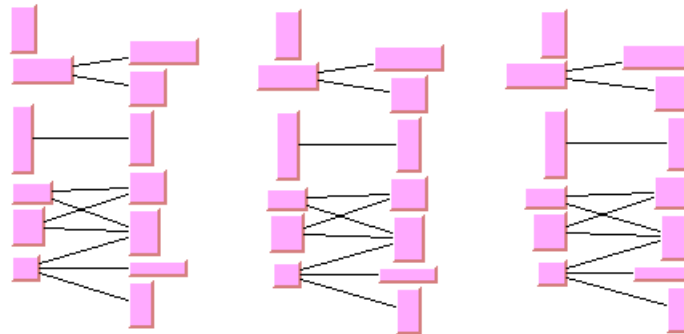
- ◆ `IlvDirection.Top`
- ◆ `IlvDirection.Bottom`
- ◆ `IlvDirection.Center` (the default)



Level justification for horizontal levels

If the flow direction is to the left or to the right, the valid values for the level justification are:

- ◆ `IlvDirection.Left`
- ◆ `IlvDirection.Right`
- ◆ `IlvDirection.Center` (the default)



Left Justification

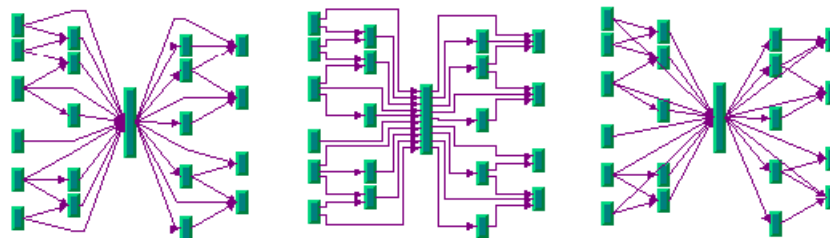
Center Justification

Right Justification

Level justification for vertical levels

Link style (HL)

The layout algorithm positions the nodes and routes the links. To avoid overlapping nodes and links, it creates bend points for the shapes of links. The link style parameter controls the position and number of bend points. The link style can be set globally, in which case all links have the same kind of shape, or locally on each link such that different link shapes occur in the same drawing.



Polyline Links

Orthogonal Links

Straight-line Links

Link styles

Link style and link shapes

Link styles work only when you use links that can be reshaped. Subclasses of `IlvPolylineLinkImage` or of `IlvSplineLinkImage`, (e.g., `IlvGeneralLink`) can be reshaped. Furthermore, link styles work only if free link connectors are installed. Free link connectors are subclasses of `IlvFreeLinkConnector`. If you use a diagram component, the free link connectors are automatically installed when needed unless specified differently. If you call layout on an `IlvGrapher` directly in Java, the layout algorithm may raise an `IlvInappropriateLinkException` if links are neither a subclass of `IlvPolylineLinkImage` nor of `IlvSplineLinkImage`, or if connectors are not a subclass of `IlvFreeLinkConnector`. In this case, you can use the methods `EnsureAppropriateLinkTypes`, `EnsureAppropriateLinkConnectors` or `EnsureAppropriateLinks` defined in the class `IlvGraphLayoutUtil` to replace inappropriate links or link connectors automatically, either before layout or when the `IlvInappropriateLinkException` is caught. For details on these

methods, see the *Java API Reference Manual*. For details on the graph model, see Using the Graph Model.

Global link style

To set the global link style:

In Java

Use the method:

```
void setGlobalLinkStyle(int style)
```

The valid values for the link style are:

◆ `IlvHierarchicalLayout.POLYLINE_STYLE`

All links get a polyline shape. A polyline shape consists of a sequence of line segments that are connected at bend points. The line segments can be turned into any direction. This is the default value.

◆ `IlvHierarchicalLayout.ORTHOGONAL_STYLE`

All links get an orthogonal shape. An orthogonal shape consists of orthogonal line segments that are connected at bend points. An orthogonal shape is a polyline shape where the segments can be turned only in directions of 0, 90, 180 or 270 degrees.

◆ `IlvHierarchicalLayout.STRAIGHT_LINE_STYLE`

All links get a straight-line shape. All intermediate bend points (if any) are removed. This often causes overlapping nodes and links.

◆ `IlvHierarchicalLayout.NO_RESHAPE_STYLE`

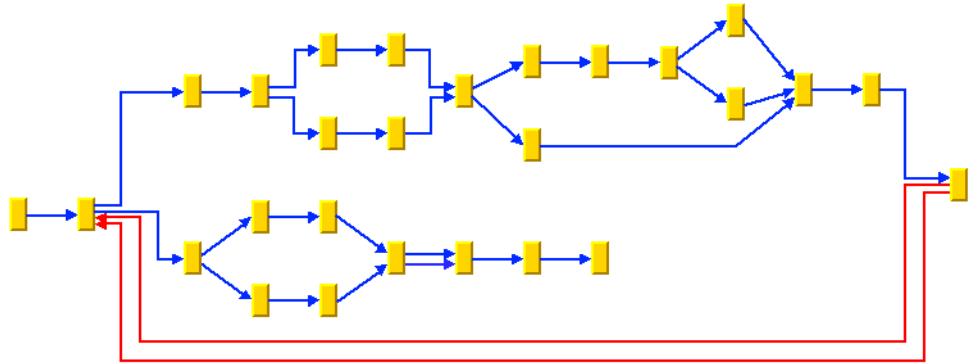
None of the links is reshaped in any manner. Note, however, that unlike fixed links, the links are not ignored completely. They are still used to calculate the leveling.

◆ `IlvHierarchicalLayout.MIXED_STYLE`

Each link can have a different link style. The style of each individual link can be set such that different link shapes can occur in the same graph.

Individual link style

All links have the same style of shape unless the global link style is `MIXED_STYLE`. Only when the global link style is `MIXED_STYLE` can each link have an individual link style.



Different Link Styles Mixed in the Same Drawing

To specify the style of an individual link:

In Java

Use the methods:

```
void setLinkStyle(Object link, int style)
```

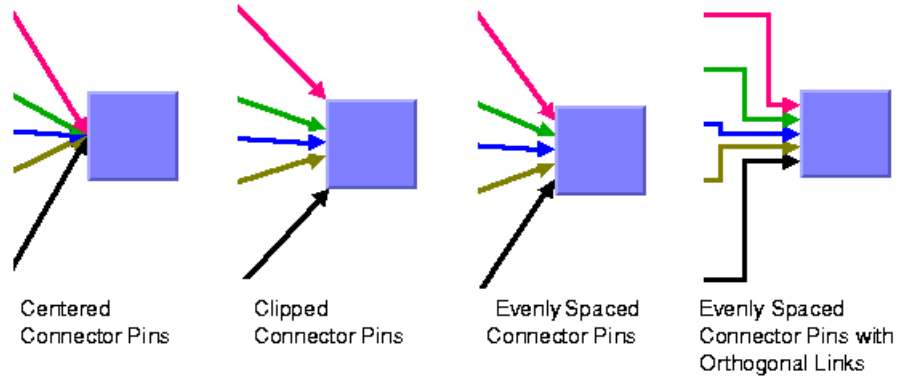
```
int getLinkStyle(Object link)
```

The valid values for the link style of local links are the same as for the global link style:

- ◆ `IlvHierarchicalLayout.POLYLINE_STYLE`
- ◆ `IlvHierarchicalLayout.ORTHOGONAL_STYLE`
- ◆ `IlvHierarchicalLayout.STRAIGHT_LINE_STYLE`
- ◆ `IlvHierarchicalLayout.NO_RESHAPE_STYLE`

Connector style (HL)

The layout algorithm positions the end points of links (the connector pins) at the nodes automatically. The connector style parameter specifies how these end points are calculated.



Connector styles

To specify the connector style:

In Java

Use the method:

```
void setConnectorStyle(int style)
```

The valid values for `style` are:

◆ `IlvHierarchicalLayout.CENTERED_PINS`

The end points of the links are placed in the center of the border where the links are attached. This option is well-suited for polyline links and straight-line links. It is less well-suited for orthogonal links, because orthogonal links can look ambiguous in this style.

◆ `IlvHierarchicalLayout.CLIPPED_PINS`

Each link pointing to the center of the node is clipped at the node border. The connector pins are placed at the points on the border where the links are clipped. This option is particularly well-suited for polyline links without port specifications. It should not be used if a port side for any link is specified.

◆ `IlvHierarchicalLayout.EVENLY_SPACED_PINS`

The connector pins are evenly distributed along the node border. This style guarantees that the end points of the links do not overlap. This is the best style for orthogonal links and works well for other link styles.

◆ `IlvHierarchicalLayout.AUTOMATIC_PINS`

The connector style is selected automatically depending on the link style. If any of the links has an orthogonal style or if any of the links has a port side specification, the algorithm chooses evenly spaced connectors. If all the links are straight, it chooses centered connectors. Otherwise, it chooses clipped connectors.

End point mode (HL)

Normally, the layout algorithm is free to choose the termination points of each link. However, the user can specify that the current fixed termination pin of a link should be used.

The layout algorithm provides two end point modes. You can set the end point mode globally, in which case all end points have the same mode, or locally on each link, in which case different end point modes occur in the same drawing.

Global end point mode

To set the global end point mode:

In Java

Use the methods:

```
void setGlobalOriginPointMode(int mode);
```

```
void setGlobalDestinationPointMode(int mode);
```

The valid values for `mode` are:

◆ `IlvLinkLayout.FREE_MODE` (the default)

The layout is free to choose the appropriate position of the connection point on the origin/destination node.

◆ `IlvLinkLayout.FIXED_MODE`

The layout must keep the current position of the connection point on the origin/destination node.

◆ `IlvLinkLayout.MIXED_MODE`

Each link can have a different end point mode.

Individual end point mode

All links have the same end point mode unless the global end point mode is `IlvLinkLayout.MIXED_MODE`. Only when the global end point mode is set to `MIXED_MODE` can each link have an individual end point mode.

To set the end point mode of an individual link:

In Java

Use the methods:

```
void setOriginPointMode(Object link, int mode);
```

```
int getOriginPointMode(Object link);
```

```
void setDestinationPointMode(Object link, int mode);
```

```
int getDestinationPointMode(Object link);
```

The valid values for mode are:

- ◆ `IlvLinkLayout.FREE_MODE` (the default)
- ◆ `IlvLinkLayout.FIXED_MODE`

Using a link connection box interface (HL)

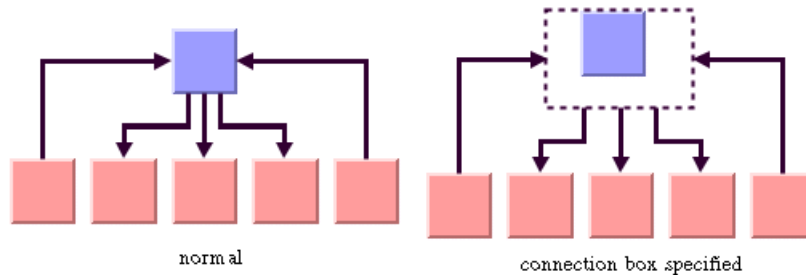
By default, the connector style determines how the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*. To set a link connection box interface in Java, use the method:

```
setLinkConnectionBoxInterface
```

The link connection box interface provides each node with a link connection box and tangential shift offsets. The Hierarchical Layout uses the link connection box but does not use the tangential offsets.

The following figure illustrates the effects of customizing the connection box. On the left is the result without any connection box interface. The picture on the right shows the effect if the connection box interface returns the dashed rectangle for the corresponding node.

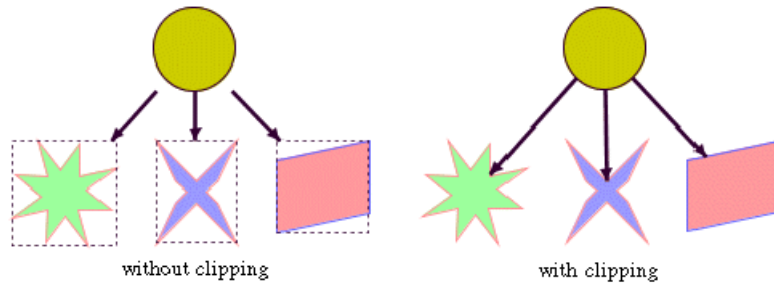


Effect of connection box interface

Using a link clipping interface (HL)

By default, the Hierarchical Layout places the connection points of links at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface

allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



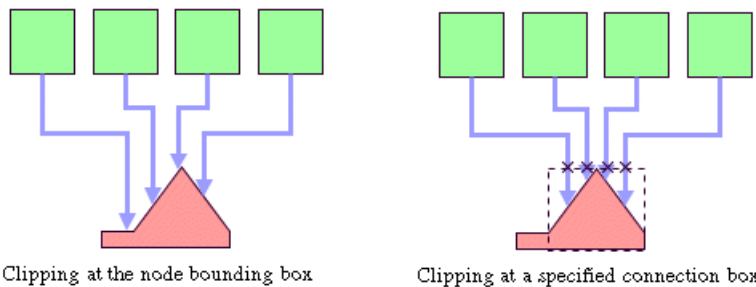
Effect of link clipping interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*. To set a link clip interface in Java, use the method:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

The connector style, the link connection box interface, and the link clip interface work together in the following way: by respecting the connector style, the proposed connection points are calculated on the rectangle obtained from the link connection box interface (or on the bounding box of the node, if no link connection box interface was specified). Then, the proposed connection point is passed to the link clip interface and the returned connection points are used to connect the link to the node.

The following figure shows an example of the combined effect.



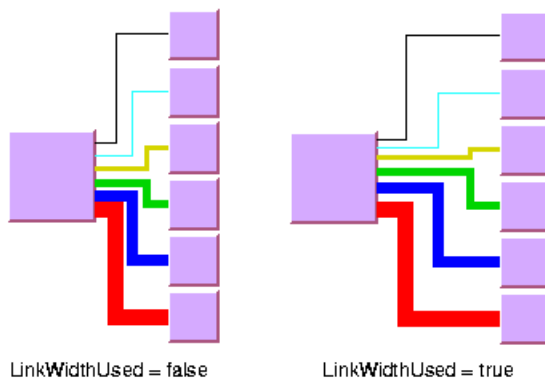
Combined effect of link clipping interface and link connection box

If the links are clipped at the red node in previous figure (left), they appear unsymmetrical with respect to the node shape, because the relevant part of the node (here: the triangle) is not in the center of the bounding box of the node, but the proposed connection points are calculated with respect to the bounding box. This can be corrected by using a link connection box interface to explicitly specify a smaller connection box for the relevant part of the node (previous figure, right) such that the proposed connection points are placed symmetrically at the triangle of the node.

For experts: thick links (HL)

If evenly spaced pins are used as connector style, the links can be evenly spaced with respect to the link center or with respect to the link border. The difference is only visible when links that connect to the same node have different widths. For instance, when the link width indicates the cost or capacity of a flow in the application, many different link width may occur.

Using the link width shows the effect of using different link widths. In the drawing on the left, the center of the links are evenly distributed at the left node. Each link has the same space available at the node side. Therefore, the thick links appear closer to each other than do the thinner links and the offsets between the link borders are different. In the drawing on the right, the thick links have more space available than do the thinner links. The offset between the link border (at the segments that connect to the left node) is constant because the link width is considered in the calculation of the connection points.



Using the link width

To enable the connector calculation to respect the link width:

In Java

Call:

```
layout.setLinkWidthUsed(true);
```

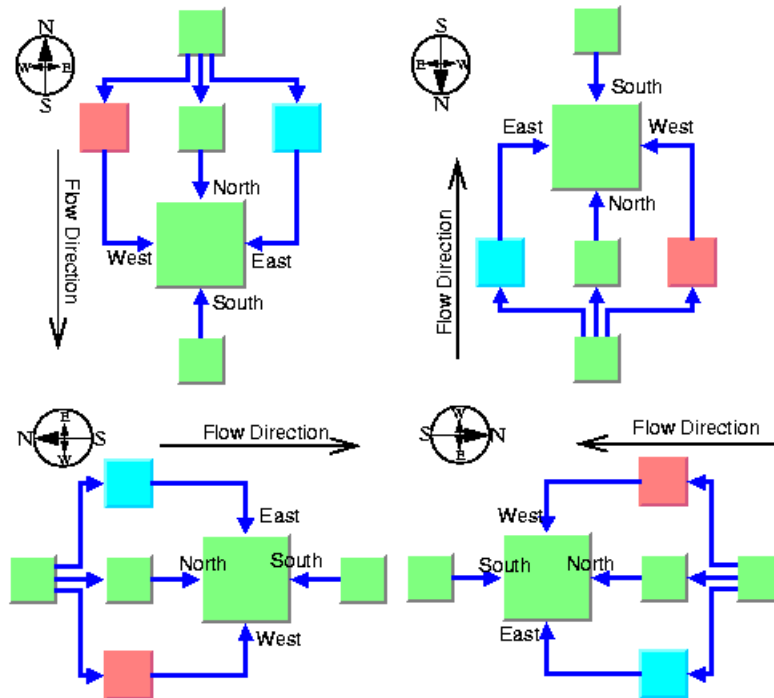
The link width setting is disabled by default. The link width has no effect if the connector styles `CENTERED_PINS` or `CLIPPED_PINS` are used.

Port sides parameter (HL)

The Hierarchical Layout algorithm produces a layout where the majority of the links flow are in the same direction. If the flow direction is towards the bottom, usually the incoming links are connected to the top side of the node and the outgoing links are connected to the bottom side of the node. It is also possible to specify on which side a link connects to the node.

To simplify the explanations of the port sides, we use the compass directions *north*, *south*, *east*, and *west*. The specified link flow direction is always towards south and the first level is towards north. If the flow direction is towards bottom, north is at the top, south at the bottom, east on the right, and west on the left side of the drawing. If the flow direction is towards right, north is on the left, south on the right, east at the top, and west at the bottom.

Link connections to port sides shows a drawing where the links connect to the larger middle node at the specified port sides. A compass icon shows the compass directions in these drawings.



Link connections to port sides

You can set at which side the link connects to its source node.

To set at which side the link connects to its source node:

In Java

Use the method:

```
void setFromPortSide(Object link, int side);
```

In a similar way, you can set at which side the link connects to its destination node.

To set at which side the link connects to its destination node:

In Java

Use the method:

```
void setToPortSide(Object link, int side);
```

The valid values for *side* are:

- ◆ `IlvHierarchicalLayout.UNSPECIFIED` (the default)
- ◆ `IlvHierarchicalLayout.NORTH`
- ◆ `IlvHierarchicalLayout.SOUTH`
- ◆ `IlvHierarchicalLayout.EAST`
- ◆ `IlvHierarchicalLayout.WEST`

To retrieve the current choice for a link, use the methods:

```
int getFromPortSide(Object link);
```

```
int getToPortSide(Object link);
```

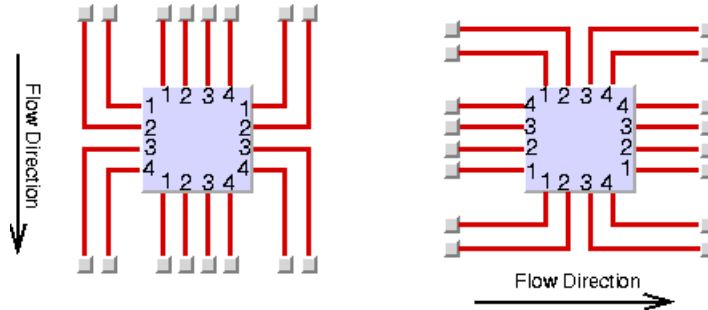
The port sides east and west work particularly well with the orthogonal link style. Polyline links with these port sides sometimes have unnecessary bends. Furthermore, if port sides are specified, the connector style `CLIPPED_PINS` should not be used.

Port index parameter (HL)

You can specify where the links connect to the node. You cannot specify the exact location, but you can specify the relative location compared to the connection points of the other links. This is done by using a port index. *Sample layout with ports and orthogonal link style* shows a sample layout with ports at many nodes.

Links that have the same port index connect at the same point of the node. The ports are evenly distributed at the node sides, in a similar way as with the connector style `EVENLY_SPACED_PINS`. The ports are ordered according to their indices. On the north and south side of a node, the port indices increase toward the east. On the east and west sides of a node, the port indices increase toward the south. By using port indices in this way, it is easier to rotate a graph by simply changing the flow direction without needing to update all the port specifications.

Port Index Numbering Conventions in Relation to Flow Direction show how the port indices depend on the flow direction.



Port Index Numbering Conventions in Relation to Flow Direction

Port numbers are normally used in combination with port sides. Therefore, you must specify how many ports are available on each side of a node.

To specify the number of ports:

In Java

Use the method:

```
void setNumberOfPorts(Object node, int side, int numberOfPorts);
```

For example, to use 4 ports on each side of a specific node, use the calls:

```
layout.setNumberOfPorts(node, IlvHierarchicalLayout.EAST, 4);
layout.setNumberOfPorts(node, IlvHierarchicalLayout.WEST, 4);
layout.setNumberOfPorts(node, IlvHierarchicalLayout.NORTH, 4);
layout.setNumberOfPorts(node, IlvHierarchicalLayout.SOUTH, 4);
```

The node side is specified again by EAST, WEST, NORTH, and SOUTH. To retrieve the retrieve the number of ports available at the node, use the method:

```
int getNumberOfPorts(Object node, int side);
```

After the number of ports per side is specified, you can choose which port each link connects to.

To choose the port side and the port index for a link:

In Java

To specify the connection at the source node, use the methods:

```
void setFromPortSide(Object link, int portSide);
```

```
void setFromPortIndex(Object link, int portIndex);
```

To specify the connection at the destination node, use the methods:

```
void setToPortSide(Object link, int portSide);
```

```
void setToPortIndex(Object link, int portIndex);
```

To obtain the current port index of a link, use the methods:

```
int getFromPortIndex(Object link);
```

```
int getToPortIndex(Object link);
```

Using the port side and port index specifications are additional constraints for the layout algorithm. The more constraints are specified, the more difficult it is to calculate a layout. Therefore, if too many links have a specified port index, this resulting layout may have more link crossings and be less balanced.

Fork link shapes (HL)

If several links start at the same position and are orthogonally routed, it is sometimes preferred that the links share the first two link segments. The shape of a link bundle of this kind looks like a fork. To enable the fork shape mode for outgoing links, call:

```
layout.setFromFork(true);
```

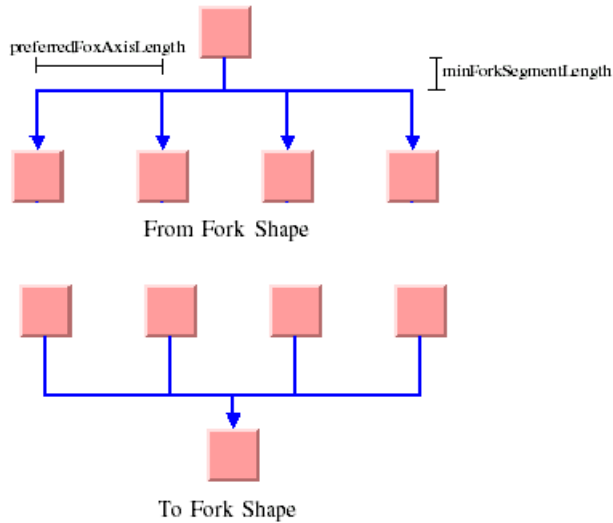
To enable the fork shape mode for incoming links:

In Java

Call:

```
layout.setToFork(true);
```

These statements have an effect only if the links are routed orthogonally. The fork appears only at those links that start or end exactly at the same point. Specifying `setFromFork(true)` by itself does not force the links to start at the same point. To force links to start or end at the same point, use the center connector style (see *Connector style (HL)*) or specify the same port for the links (see *Port index parameter (HL)*).



Fork Link Shapes

There are two spacing parameters for the fork shape:

In Java

```
void setMinForkSegmentLength(float length)
```

It sets the minimal length of the segment that is directly adjacent to the node.

```
void setPreferredForkAxisLength(float length)
```

This method sets the preferred length of the fork axis per branch (the second segment adjacent to the node). If the fork has five branches, the entire axis has the preferred length five times the specified parameter. The preferred fork axis length is only a hint for the layout algorithm. If enough space is available, the algorithm will enlarge the fork axis to avoid unnecessary link bends. If there is not enough space, the algorithm may as well calculate a fork axis that is smaller than the preferred one.

Fork link shapes may sometimes look ambiguous, in particular when a link starts at the same point where another link ends, because in this case it is impossible to recognize whether the arrowhead belongs to one or the other link.

Link priority parameter (HL)

The layout algorithm tries to place the nodes such that all links are short, point in the flow direction, and do not cross each other. However, this is not always possible. Often, links cannot have the same length. If the graph has cycles, some links must be reversed against the flow direction. If the graph is a nonplanar graph, some links have to cross each other.

The link priority parameter controls which links should be selected if long, reversed, or crossing links are necessary. Links with a low priority are more likely to be selected than links with a high priority. This does not mean that low-priority links are always longer,

reversed, or crossed, because the graph may have a structure such that no long, reversed or crossing links are necessary.

To set the link priority:

In Java

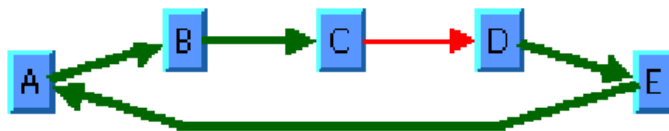
Use the methods.

```
void setLinkPriority(Object link, float priority)
```

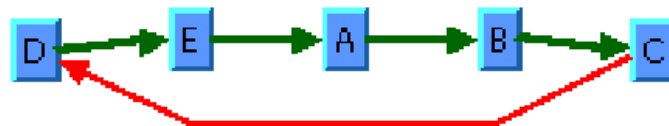
```
float getLinkPriority(Object link)
```

The default value of the link priority is 1.0. Negative link priorities are not allowed.

For an example of using the link priority, consider a cycle A->B->C->D->E->A. It is impossible to lay out this graph without reversing any link. Therefore, the layout algorithm selects one link to be reversed. To control which link is selected, you can give one link a lower priority than the others. This link will be reversed. In *Working with link priorities*, the bottom layout shows the use of the link priority. The link C->D was given the priority 0.5, while all the other links have the priority 1.0. Therefore C-D is reversed. The top layout in *Working with link priorities* shows what happens when all links have the same priority. Link E->A is reversed.



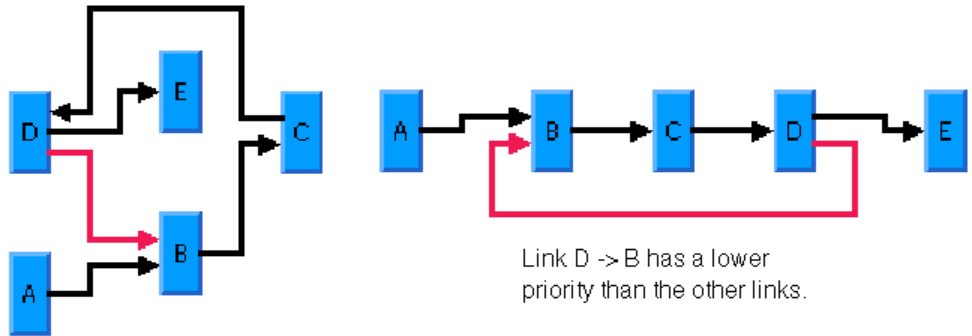
All links have the same priority.



Link C -> D has a lower priority than the other links.

Working with link priorities

The use of link priorities is important in combination with ports. Links with “from” ports on the south side and “to” ports on the north side are preferably laid out opposite to the flow direction. Such a feedback link may cause parts of the drawing to tip over. *Using Link Priorities and Ports* shows an example. The red link is a feedback link with port specifications. To obtain the correct result as shown in the right side of the following figure, you would set the priority of the feedback link to a very low value.

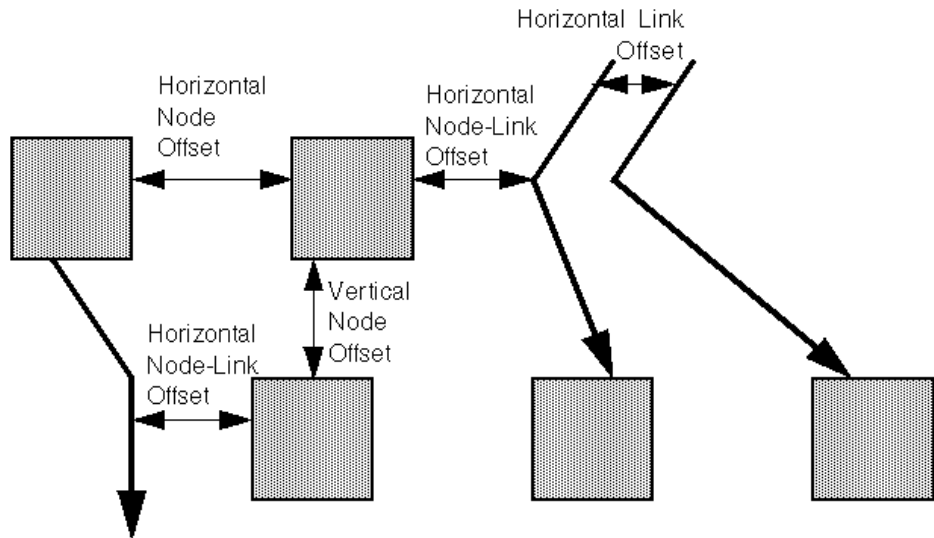


All links have the same priority.

Using Link Priorities and Ports

Spacing parameters (HL)

The spacing of the layout is controlled by three kinds of spacing parameters: the minimal offset between nodes, the minimal offset between parallel segments of links and the minimal offset between a node border and a bend point of a link or a link segment that is parallel to this border. The offset between parallel segments of links is at the same time the offset between bend points of links. All three kind of parameters occur in both directions: horizontally and vertically.



Spacing parameters

To set the spacing parameters:

In Java

- ◆ For the horizontal direction, use the methods:

```
void setHorizontalNodeOffset(float offset)
```

```
void setHorizontalLinkOffset(float offset)
```

```
void setHorizontalNodeLinkOffset(float offset)
```

- ◆ For the vertical direction, use the methods:

```
void setVerticalNodeOffset(float offset)
```

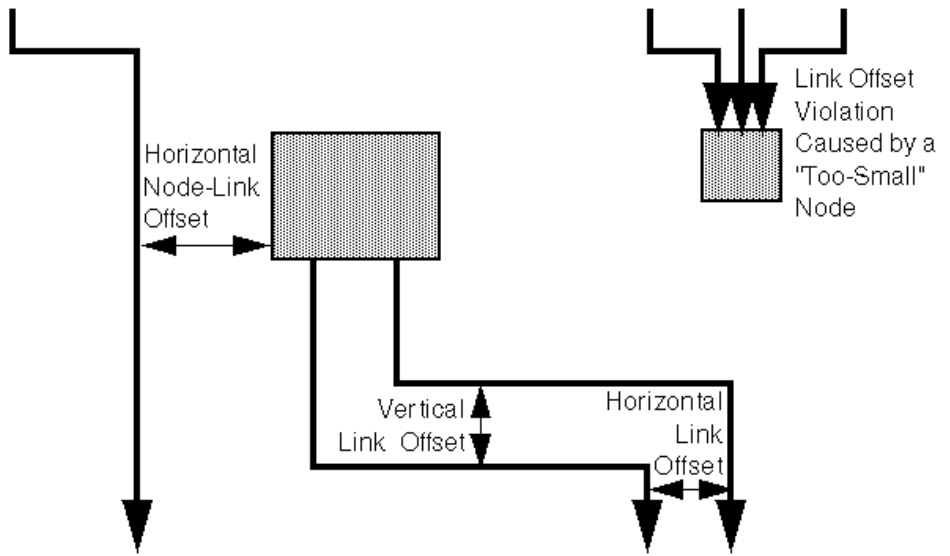
```
void setVerticalLinkOffset(float offset)
```

```
void setVerticalNodeLinkOffset(float offset)
```

For a layout with horizontal levels (the flow direction is to the top or to the bottom), the horizontal node offset is the minimal distance between nodes of the same level. The vertical node offset is the minimal distance between nodes of different levels, that is, the minimal distance between the levels. For non-orthogonal link styles, the horizontal link offset is basically the minimal distance between bend points of links. The horizontal node-link offset is the minimal distance between the node border and the bend point of a link. For horizontal levels, the vertical link offset and the vertical node-link offset play a role only if the link shapes are orthogonal.

Similarly, for a layout with vertical levels (the flow direction is to the left or to the right), the vertical node offset controls node distances within the levels. The horizontal node offset is the minimal distance between the levels. In this case, the vertical link offset and the vertical node-link offset always play a role, while the horizontal link offset and the horizontal node-link offset affect the layout only with orthogonal links.

For orthogonal links, the horizontal link offset is the minimal distance between parallel, vertical link segments. The vertical link offset is the minimal distance between parallel, horizontal link segments. However, the layout algorithm cannot always satisfy these offset requirements. If a node is very small but has many incident links, it may be impossible to place the links orthogonally with the specified minimal link distance on the node border. In this case, the algorithm places some link segments closer than the specified link offset.



Spacing parameters for orthogonal links

Incremental mode with HL

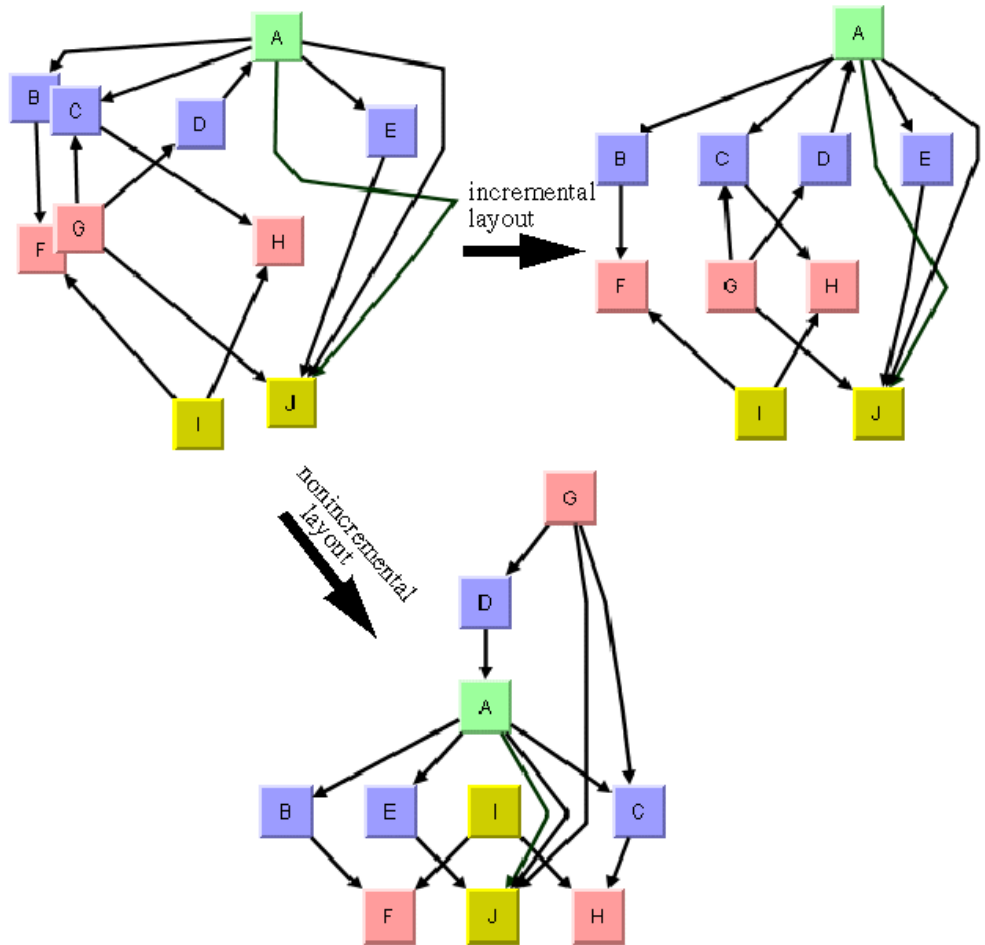
In some circumstances you may need to use a sequence of layouts on the same graph. For example:

- ◆ You work with graphs that have become out-of-date and you need to extend the graph. If you perform a layout on the extended graph, you probably want to identify the parts that were already laid out in the original graph. The layout should not change very much when compared with the layout of the original graph.
- ◆ The first layout results in a drawing with minor deficiencies. You want to solve these deficiencies manually and perform a second layout to clean up the drawing. The second layout probably should not greatly change the parts of the graph that were already acceptable after the first layout.

The Hierarchical Layout normally works nonincrementally. It performs a layout from scratch and moves all nodes to new positions and reroutes all links. The previous positions of nodes have no influence on the result of the layout. Hence, even a small change can cause a large effect on the next layout.

But the Hierarchical Layout also supports incremental sequences of layout that “do not change very much.” It can place the nodes close to their previous positions, so that you can more easily identify the parts that had already been laid out in the original graph. Incremental mode takes the previous positions of the nodes into account. In this mode the algorithm preserves the relative order of the levels and the nodes within the levels in the subsequent layout. It does not preserve the absolute positions of the nodes, but it tries to detect the structure of the previous layout by examining the node coordinates. For instance, if two nodes are in the same level, then they stay in the same level after an incremental layout. If a node is in a higher level than another node, it stays in the higher level.

The following figure illustrates the difference between an incremental and nonincremental layout.



Incremental and Nonincremental Layouts

Incremental mode is disabled by default.

To enable incremental mode:

In Java™

```
layout.setIncrementalMode(true);
```

Phases of the incremental mode

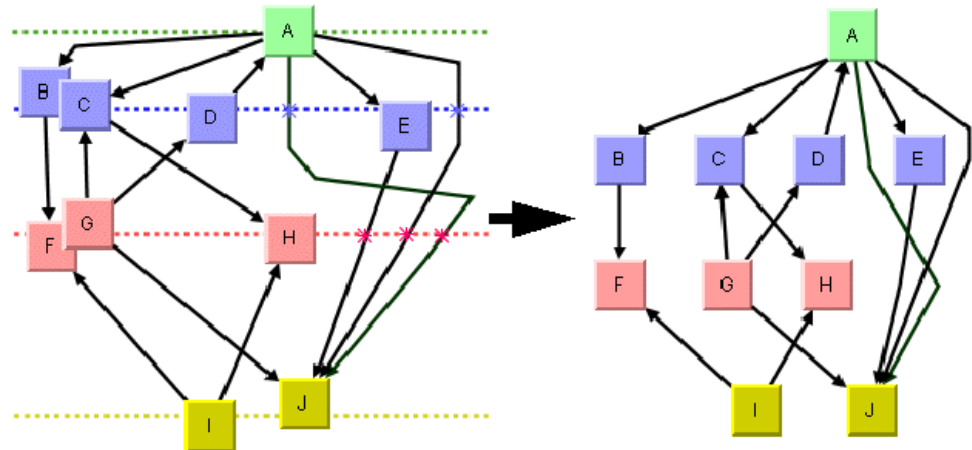
The layout algorithm analyzes the drawing in incremental mode in the following way:

1. First, it determines from the node coordinates which nodes must belong to the same level. For instance, if the flow direction is towards the bottom, it tries to detect horizontal reference lines at those vertical positions where many nodes are placed along a line.

The specified vertical node offset helps to detect these lines because the horizontal reference lines should be approximately the vertical node offset apart. See the following figure.

2. All nodes that touch the same reference line are assigned to the same level.
3. It determines the order of the nodes within each level by analyzing where the node touches the reference line. For instance, if the flow direction is towards the bottom, it determines from the x coordinate of the nodes how they are ordered within the levels.
4. If long links span several levels, the algorithm can preserve the shape of a long link. It determines the point where a link crosses the level reference line. It creates a bend point for the long link inside the level. It tries to preserve the order of the bend points in each level. For instance, if in a flow direction towards the bottom, a long link bypasses another node on the right side, then the incremental layout tries to find a similar shape of the link that bypasses the node on the right side, as illustrated in the following figure.
5. Finally, the layout tries to calculate the absolute positions of the nodes that respect the levels and the ordering within the levels. It tries to balance the node positions. However, it also tries to place each node close to its previous position. Both criteria often compete with each other, because to get a perfect balance, nodes must sometimes move far from their original position. The Hierarchical Layout contains a parametrized heuristic to satisfy both criteria.

The following figure shows the result of the incremental phases.



Incremental layout phases

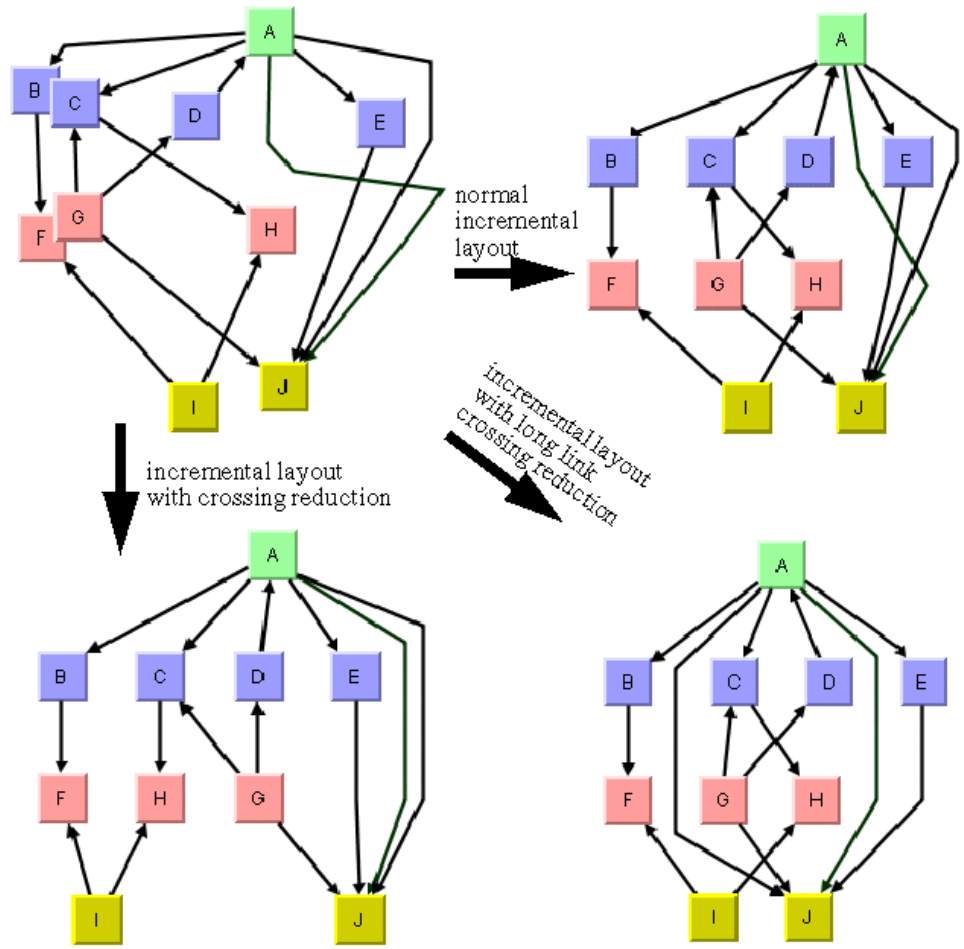
Expert parameters of the incremental mode

Each phase of the incremental mode can be parameterized. These layout parameters have an effect only if incremental mode is switched on.

Minimizing long link crossings

The incremental layout tries to preserve the shape of long links that cross several levels. This implies that link crossings between long links are not resolved. If crossings of long links

are not desired, it may be better to reroute long links from scratch. The following figure shows four hierarchy trees, with the original layout at the upper left. The bottom right shows the result if long links are rerouted, and the top right shows the result if the shape of long links is preserved.



Crossing Reduction During Incremental Layouts

To reroute long links from scratch, you must enable the crossing reduction mechanism for long links:

In Java

```
layout.setLongLinkCrossingReductionDuringIncremental(true);
```

The crossing reduction of long links determines only the shape of the links. It does not influence the order of the other nodes within the levels.

Minimizing all link crossings

Optionally, you can apply a crossing reduction to all nodes within each level. In this case, the incremental layout determines from the node coordinates which nodes belong to the same level, but it may reorder the nodes within the levels completely to avoid link crossings. It also reorders the long links in this case. The previous figure, bottom left shows the result. Notice that the order of the nodes “F,” “G,” and “H” have changed to resolve the link crossings.

To enable the crossing reduction for all nodes:

In Java

```
layout.setCrossingReductionDuringIncremental(true);
```

Setting absolute level positioning

The incremental layout tries to place the nodes in absolute positions that are close to the previous positions. It tries to avoid nodes moving a large distance, because even if the relative order of the nodes within the levels does not change, large movement distances can be confusing for users. It is much easier to keep a mental map of the diagram if the nodes remain close to the previous positions.

The following figure illustrates node repositioning with and without taking the previous positions into account. The incremental layout of the original graph at the top left results in the graph at the top right, which is easier to recognize as the same graph than the graph at the bottom.

The absolute level positioning feature is enabled by default, but it can be disabled.

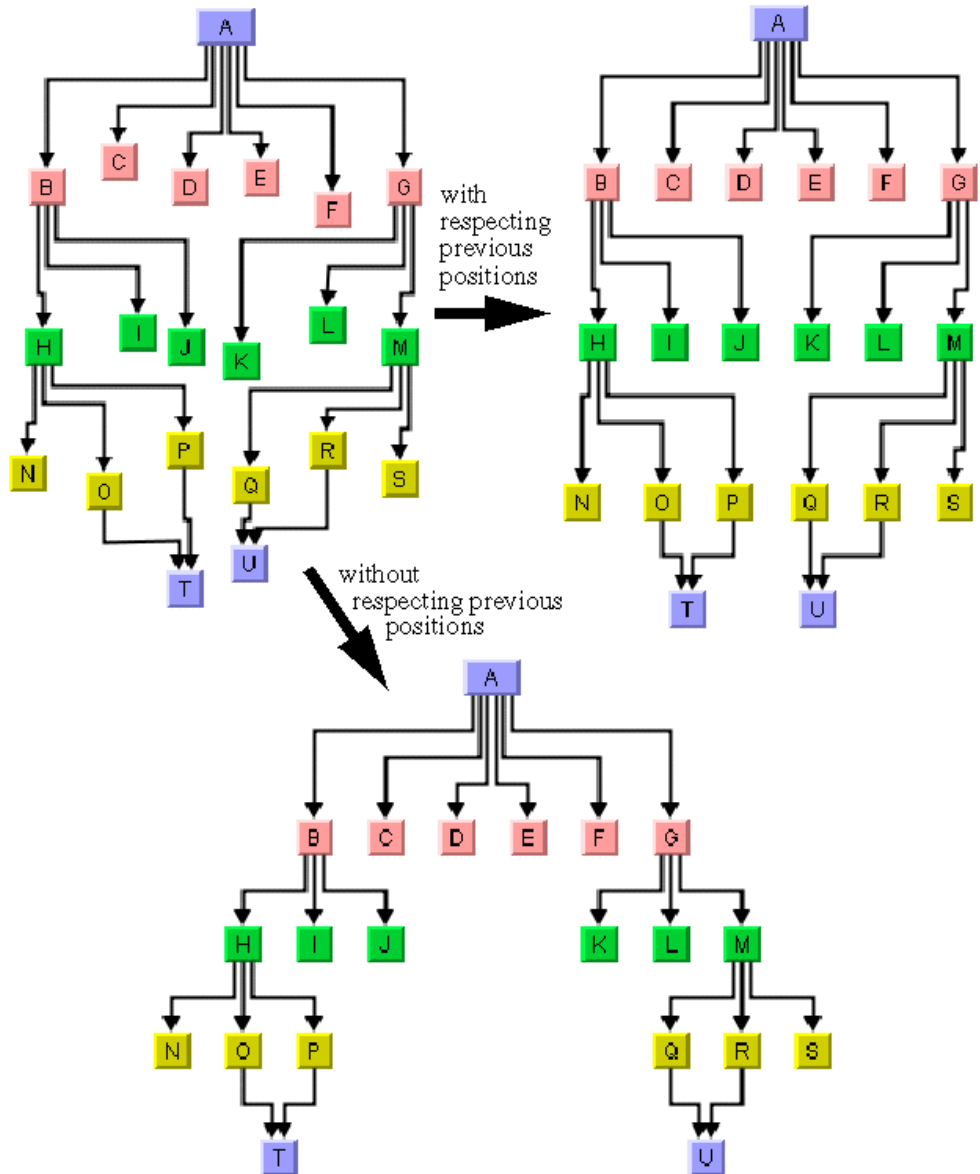
To disable the absolute level positioning feature:

In Java

Call

```
layout.setIncrementalAbsoluteLevelPositioning(false);
```

With this statement, the layout does not try to place the nodes close to the previous positions. It places the nodes such that the layout is balanced. However, to create a perfect balance, the layout may need to move a few nodes so far apart that you can no longer recognize the diagram after the layout from the node positions that were shown in the previous layout (see the following figure, bottom).



Absolute Positioning During Incremental Layouts

Setting absolute level position range and tendency

If absolute level positioning is enabled, it competes with the aesthetic criteria to create a balanced layout. Due to the fact that nodes must stay close to their previous positions, the diagram after incremental layout may be somewhat unbalanced and unsymmetrical. The

Hierarchical Layout algorithm uses a heuristic that you can influence by two parameters, the absolute level position range and tendency.

The absolute level positioning feature is enabled by default, but it can be disabled.

To disable the absolute level positioning feature:

In Java

Call:

```
layout.setIncrementalAbsoluteLevelPositionRange(100);
```

This statement specifies that within the range of 100 coordinate units from the old position of the node, the balance is the only criteria for the placement. This means that a node whose optimal position is less than 100 coordinate units away from its previous position is placed exactly at its optimal position. Nodes whose optimal position is farther away are placed at a position that is a compromise between previous position and optimal position. This is illustrated in figure below, right.

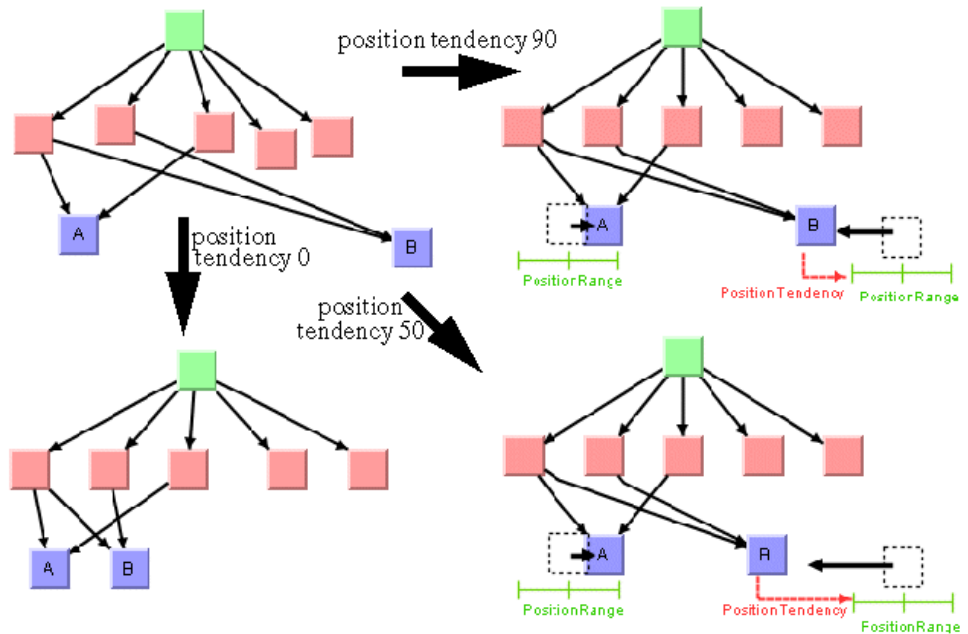
To set the absolute level position tendency:

In Java

Call:

```
layout.setIncrementalAbsoluteLevelPositionTendency(70);
```

This statement specifies that positions of nodes whose optimal positions are far away from their previous position are 70% influenced by their previous position and 30% influenced by their optimal positions. Imagine a rubber band that tries to pull a node to its previous position, and another rubber band that tries to pull the same node to its optimally balanced position. The level position tendency 70 means that one rubber band pulls with 70% of the force towards the previous position, and the other rubber band pulls with 30% towards the optimal position. Increasing the tendency means that the node stays closer to its old position, decreasing it means that the node moves closer to its optimal position. If you set the tendency to 0%, this has the same effect as disabling the incremental absolute level positioning (see the following figure).



Absolute Positioning During Incremental Layouts

Marking nodes for incremental layout

Incremental layout normally treats all nodes and links of the drawing in the same way. However, you may have added nodes and links to the drawing programmatically, and the new nodes and links do not have meaningful coordinates yet. Perhaps you have placed them all at the origin (0,0), or at random coordinates. In this case, you need an incremental layout that takes the coordinates of all nodes into account that were previously laid out, while it ignores the coordinates of all new nodes. The incremental mode of the Hierarchical Layout allows you to specify in Java which nodes cannot be laid out incrementally by calling the method:

```
layout.markForIncremental (nodeOrLink);
```

If you call this statement, the node or link is marked such that its coordinates are ignored during the next incremental layout. The positions of marked nodes and links are calculated from scratch. The mark is valid only until the next layout and is automatically cleared afterwards.

Layout constraints for HL

The Hierarchical Layout algorithm supports *relative position constraints* on nodes. Such a constraint is a rule on how a particular node (or a group of nodes) must be placed with respect to the other nodes. The constraints influence the relative positions. For example, you can force node A to be on the left side of node B, so the position of A is expressed relative to the position of B. It is theoretically possible to specify contradicting constraints: if you specify that node A must be on the left side of B and B must be on the left side of A, then these constraints are not solvable at the same time. If A is on the left side of B, then B must be on the right side of A. The Hierarchical Layout algorithm tries to detect and resolve constraint conflicts automatically. It ignores those constraints that are infeasible. Since the automatic constraint resolution is time consuming, it is recommended to specify nonconflicting constraints when possible.

Constraints should be used only if the incremental mode is switched off. In fact, the incremental mode is implemented by means of additional constraints that are added internally. Hence, if you use constraints during the incremental mode, it is very likely that the system detects so many constraint conflicts that you get unexpected results.

Constraints should be used carefully. The more constraints are specified, the more difficult it is to calculate a layout. Therefore, this resulting layout may have more link crossings and be less balanced than a graph with no constraints.

Each type of constraint is represented by a subclass of `IlvHierarchicalConstraint`. The following constraint types are available:

| | |
|--|--|
| <code>IlvLevelRangeConstraint</code> | Forces a node into a range of certain levels |
| <code>IlvSameLevelConstraint</code> | Forces two nodes to the same level. |
| <code>IlvRelativeLevelConstraint</code> | Forces a node to a lower/higher level than another node. |
| <code>IlvGroupSpreadConstraint</code> | Forces a group of nodes on levels that are no more than a specified spread value apart. |
| <code>IlvRelativePositionConstraint</code> | Forces a node to a lower/higher position than another node of the same level. |
| <code>IlvSideBySideConstraint</code> | Forces two nodes of the same level to be placed side by side. |
| <code>IlvExtremityConstraint</code> | Forces a node to the first or last level, or to the first or last position within a level. |
| <code>IlvSwimLaneConstraint</code> | Forces a group of nodes into the same rectangular swim lane area. |

Adding and removing constraints in Java for HL

You can add constraints to the Hierarchical Layout by allocating a new constraint object and calling this method on the `IlvHierarchicalLayout` instance:

```
void addConstraint(IlvHierarchicalConstraint constraint);
```

You can add as many constraints as you want. The constraints will be respected during the subsequent layout calls until you remove them. To remove the most recent constraint, call:

```
void removeConstraint();
```

To remove a specific constraint, call:

```
void removeConstraint(IlvHierarchicalConstraint constraint);
```

To remove all existing constraints, call:

```
void removeAllConstraints();
```

You can retrieve the constraints that were added to a Hierarchical Layout with the method:

```
Enumeration getConstraints()
```

Node groups

Some constraints affect single nodes. Other constraints affect groups of nodes. The class `IlvNodeGroup` is a convenient way to specify a group of nodes in Java. You can create a group of nodes in the following way:

```
group = new IlvNodeGroup();
while (...) {
    group.add(node);
}
```

A node group has a similar functionality to a vector. You can ask for the size and elements of the group, remove elements from the group, or check whether a node already belongs to the group. You can also convert a vector of nodes into a group:

| | |
|---|---|
| <code>group.add(node)</code> | Adds a node to the group. |
| <code>group.remove(node)</code> | Removes a node from the group. |
| <code>group.contains(node)</code> | Checks whether a node is in the group. |
| <code>group.size()</code> | Returns the number of nodes in the group. |
| <code>group.elements()</code> | Returns the nodes of the group as an Enumeration. |
| <code>group = new IlvNodeGroup(vector)</code> | Creates a new group that contains the nodes stored in the input vector. |

Level range constraints (HL)

In Step 1 of the layout algorithm (the leveling phase), the nodes are partitioned into levels. These levels are indexed starting from 0. For instance, when the flow direction is to the bottom, the nodes of the level index 0 are placed at the topmost horizontal level line and the nodes with larger level index are placed at a position lower than the nodes with smaller level index (see *Level and position indices*). The layout algorithm calculates these level indices automatically.

You can choose how the levels are partitioned by specifying the range of the level index for some nodes. The nodes are placed in the levels whose index is in the specified range. You have to specify the minimum and maximum index of the level.

To specify the minimum and maximum index of the level:

In Java

Call:

```
layout.addConstraint(new IlvLevelRangeConstraint(node, 5, 7));
```

If you want to place the node exactly at level 5, call:

```
layout.addConstraint(new IlvLevelRangeConstraint(node, 5, 5));
```

Alternatively, you can call:

```
layout.setSpecNodeLevelIndex(node, 5);
```

which has exactly the same meaning.

If you want to force the node to level 5 and above, set `UNSPECIFIED` as the maximal level.

In Java

Call:

```
layout.addConstraint(  
    new IlvLevelRangeConstraint(node, 5, IlvHierarchicalLayout.UNSPECIFIED));
```

If you want to force the node to level 5 and below (that is, level 0, ..., 5), set `UNSPECIFIED` as the minimal level; for example, in Java:

```
layout.addConstraint(  
    new IlvLevelRangeConstraint(node, IlvHierarchicalLayout.UNSPECIFIED, 5));
```

In this particular case, you could also use zero (0) as the minimal level because the level indices start at 0.

You can apply the constraint to a group of several nodes at once. This has the same effect as specifying the constraint for each single node of the group, but it is more memory efficient and convenient. For instance, if you want to force the group of three nodes to the levels between 5 and 7:

To specify these parameters:

In Java

Create a `IlvNodeGroup` object (see *Node groups*) of the three nodes and add it to the constraint in the following way:

```
layout.addConstraint(new IlvLevelRangeConstraint(nodeGroup, 5, 7));
```

Level index parameter (HL)

The level index is a special case of a level range constraint (see *Level range constraints (HL)*). It forces the node to one particular level. For your convenience, you can specify the level index of a node directly by the method:

```
void setSpecNodeLevelIndex(Object node, int index)
```

You pass a single node as the first argument (not a node group). The default index value is -1. If the default value is used, or if a node is set to a negative level index, the level index is considered to be unspecified. In this case the layout algorithm automatically calculates an appropriate level index during the leveling phase of the algorithm.

To obtain the specified level index for a node, use the method:

```
int getSpecNodeLevelIndex(Object node)
```

However, this method returns the value that was set by `setSpecNodeLevelIndex`. If the level index was specified by allocating a corresponding level range constraint that has the same meaning, `getSpecNodeLevelIndex` still returns -1.

Warning: Using arbitrarily large level indices is not recommended. For instance, if you set the level index of a node to 100000, the layout algorithm creates 100,000 levels even if the graph has far fewer nodes. This causes the layout algorithm to become unnecessarily slow.

Same level constraints (HL)

If you want to force several nodes to the same level with fixed index, you can set the level index parameter of these nodes accordingly (see *Level index parameter (HL)*) or use a level range constraint (see *Level range constraints (HL)*). However, if you want to force several nodes to the same level *without* forcing them to a specific level index, you cannot use these mechanisms. You must use a same level constraint.

To set the same level constraint:

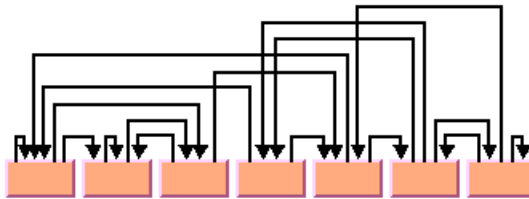
In Java™

Call:

```
layout.addConstraint(new IlvSameLevelConstraint(node1, node2));
```

This forces `node1` and `node2` to be placed into the same level, but it does not constrain them to any particular level.

The following figure illustrates the placement of nodes on the same level.



All Nodes Fixed at Same Level

Group spread constraints (HL)

An alternative way to force a group of nodes to the same level is by specifying a group spread constraint with a spread size of zero (0). In general, the group spread constraint forces a group of nodes to $k+1$ subsequent levels. The number k is the spread size. It does not select the lowest or highest level index of the group, but only requires that the nodes be placed no more than k levels apart. Hence, if $k=0$, all nodes of the group are placed at the same level.

To illustrate the general group spread constraint on nodes with ID “nodeA”, “nodeB” and “nodeC”:

In Java

To use the group spread constraint call:

```
IlvNodeGroup nodeGroup = new IlvNodeGroup();
nodeGroup.add(nodeA);
nodeGroup.add(nodeB);
nodeGroup.add(nodeC);
layout.addConstraint(new IlvGroupSpreadConstraint(nodeGroup, 2));
```

The constraint is satisfied if the highest level index for `nodeA`, `nodeB`, and `nodeC` is no more than two levels apart from the smallest level index of the nodes. For instance, the constraint is satisfied if the level indices for `nodeA`, `nodeB`, and `nodeC` are 1, 2, 3; or if they are 7, 8, 9; or if they are 16, 14, 15. The constraint is also satisfied if all three nodes are placed at level 5, or if two of the nodes are placed at level 15 and the third node at level 13. The constraint is not satisfied if the level indices for `nodeA`, `nodeB`, and `nodeC` are 3, 5, 6, because in this case the highest index (6) is more than two levels away from the lowest index (3).

Relative level constraints (HL)

If the flow direction is towards the bottom, level 0 is topmost in the drawing. In this layout you can specify by relative level constraints that a node be above or below another node. If the flow direction is towards the right, level 0 is leftmost in the drawing. Here you can specify by relative level constraints that a node be left or right of another node.

In Java

In Java™, call:

```
layout.addConstraint(  
    new IlvRelativeLevelConstraint(nodeA, nodeB, priority));
```

This forces `nodeA` to be placed at a level with a smaller index than `nodeB`. Since relative level constraints compete with each other, you must specify the priority of the constraint. In fact, links also impose constraints on the system, and the link priority has the same impact as the constraint priority. A link with priority 10 forces its (usually) source node (unless ports are specified) into a lower level than its target node. To force the source node into a higher level than the target node, you need to create a constraint with a higher priority than the link. For instance, to ensure that the constraints are satisfied even if there are many links, you can use link priorities between 0 and 10 and constraint priorities between 1000 and 10,000.

You can also create a relative level constraint between groups of nodes.

In Java

Call:

```
layout.addConstraint(  
    new IlvRelativeLevelConstraint(nodeGroup1, nodeGroup2, priority));
```

Position index parameter (HL)

In Step 2 of the layout algorithm (the crossing reduction phase), the nodes are ordered within the levels. All nodes that belong to the same level get a position index starting from 0. For instance, when the flow direction is to the bottom, the node with the position index 0 is placed in the leftmost position within its level. The nodes with a larger position index are placed farther to the right than the nodes with a smaller position index in the same level. The nodes of different levels are independent. The node of the first level with the position index 0 is to the left of the node of the first level with the position index 1, but not necessarily to the left of a node of another level with position index 0. Note that long links crossing a level also obtain a position index (see *Level and position indices*). The layout algorithm calculates these position indices automatically.

You can affect how the nodes are positioned within each level by specifying the position index of some nodes. The nodes are placed at the specified position within their level.

To specify the position index of a node in Java™, use the method:

```
void setSpecNodePositionIndex(Object node, int index)
```

The default value is -1. If the default value is used, if a node is set to a negative position index, or if a node is set to a position index that is larger than the number of nodes of its level, the layout automatically calculates an appropriate position index during the crossing reduction step.

To obtain the current position index of a node, use the method:

```
int getSpecNodePositionIndex(Object node)
```

Relative position constraints (HL)

Working with absolute node position indices is inconvenient in certain situations. For example, if two nodes belong to the same level, you may want to force one node to a position with a lower index than the other node without fixing the absolute positions of the nodes. You can achieve this by using a relative position constraint.

The relative position constraint forces a specific order upon the nodes of a level, but it does not specify which nodes are directly neighbored. For instance, a relative position constraint may force `nodeA` to be placed somewhere at a lower position than `nodeB`, but there may be many nodes between `nodeA` and `nodeB`.

In Java™

Call:

```
layout.addConstraint(  
    new IlvRelativePositionConstraint(nodeA, nodeB, priority));
```

This forces `nodeA` to a lower position than `nodeB`. If the flow direction is towards the bottom, the nodes are in horizontal levels; hence the constraint means that `nodeA` is placed at the left side of `nodeB`. If the flow direction is towards the right, the nodes are in vertical levels; hence the constraint means that `nodeA` is placed below `nodeB`.

The relative position constraint has an effect only if both nodes actually belong to the same level. To achieve this, you can, for instance, use a same level constraint in addition. There is no way to influence the relative position of nodes that belong to different levels.

Similar to the relative level constraint, the relative position constraint can be applied to node groups. These constraints also have priorities that indicate which constraints dominate if a constraint conflict occurs. The higher the priority, the more likely the constraint is satisfied when resolving constraint conflicts.

Side-by-side constraints (HL)

To force nodes to be directly neighbored, use the side-by-side constraint.

In Java™

You can create a side-by-side constraint on a group of type `IlvNodeGroup` (see *Node groups*):

```
layout.addConstraint(  
    new IlvSideBySideConstraint(nodeGroup, priority));
```

If the node group consists of just two nodes, it forces the two nodes to be placed side by side. However, it does not specify which node is at the lower node position and which node is at the higher node position. If the group consists of more than two nodes, it forces the nodes to be placed at consecutive positions such that all nodes are clustered together. A node that does not belong to the group cannot be placed between the nodes of the group.

For example, assume that the group contains the three nodes A, B, C. The constraint is satisfied if the position indices of A, B, and C are 3, 4, 5 or 9, 7, 8. However, if node D is placed between A and B (say, D has position 4, A has position 3, and C has position 5), then the constraint is not satisfied because D does not belong to the same group.

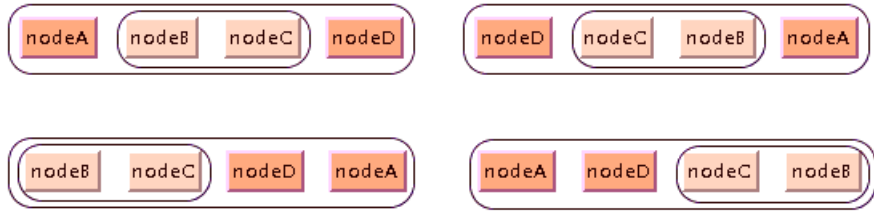
The side-by-side constraint has an effect only if the nodes actually belong to the same level. To achieve this, you can, for instance, use a same level constraint in addition.

Side-by-side constraints have priorities that decide how to resolve constraint conflicts. The higher the priority, the more likely the constraint is satisfied.

You can use side-by-side constraints to create nested clusters. For example, in Java:

```
IlvNodeGroup group1 = new IlvNodeGroup();  
group1.add(nodeA);  
group1.add(nodeB);  
group1.add(nodeC);  
group1.add(nodeD);  
layout.addConstraint(  
    new IlvSideBySideConstraint(group1, 10.0f));  
IlvNodeGroup group2 = new IlvNodeGroup();  
group2.add(nodeB);  
group2.add(nodeC);  
layout.addConstraint(  
    new IlvSideBySideConstraint(group2, 10.0f));
```

The first constraint specifies that `nodeA`, `nodeB`, `nodeC`, and `nodeD` must be clustered. The second constraint specifies that `nodeB` and `nodeC` are clustered inside the larger cluster. This means that no other node can be placed between the four nodes and, furthermore, neither `nodeA` nor `nodeD` can be placed between `nodeB` and `nodeC`. The following figure shows four solutions that satisfy both constraints.



Sketch of Solutions for Side-By-Side Constraints

Extremity constraints (HL)

To force a node to the first level, you can specify:

```
layout.setSpecNodeLevelIndex(node, 0);
```

However, you cannot specify a level index for the last level because it is unknown at the beginning of layout how many levels will be created. It is unwise to specify:

```
layout.setSpecNodeLevelIndex(node, java.lang.Integer.MAX_VALUE);
```

because this will create many empty levels between the levels actually used and the last one. Even though these empty levels are removed in postprocessing steps, this influences the speed and quality of the layout. (In fact, the algorithm will run out of memory if you set the specified level index unreasonably high.)

By using constraints you can achieve the same effect more efficiently.

To force a node to the first level:

In Java

In Java™, call:

```
layout.addConstraint(  
    new IlvExtremityConstraint(node, IlvHierarchicalLayout.NORTH));
```

To force a node to the last level:

In Java

Call:

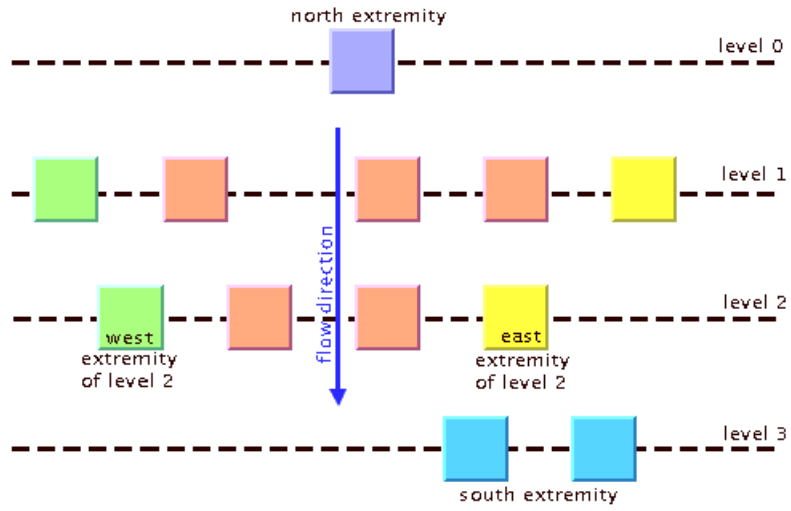
```
layout.addConstraint(  
    new IlvExtremityConstraint(node, IlvHierarchicalLayout.SOUTH));
```

With compass directions as a convenient reference (see *Port sides parameter (HL)*), the first level indicates the north pole and the last level indicates the south pole. You can also specify extremity constraints for the east and west sides:

```
layout.addConstraint(  
    new IlvExtremityConstraint(node1, IlvHierarchicalLayout.EAST));  
layout.addConstraint(  
    new IlvExtremityConstraint(node2, IlvHierarchicalLayout.WEST));
```

The west extremity constraint forces the node to the lowest position index within its level, and the east extremity constraint forces the node to the highest position index within its level. The position indices specify the relative position within the level. For instance, a node with west extremity constraint will be the leftmost node within its level, if the flow direction is towards the bottom. However, this does not affect other levels; there may be a node in another level that is still placed farther to the left.

The following figure illustrates some extremity constraints.



Sketch of Extremity Constraints

Swim lane constraints (HL)

Swim lanes are rectangular areas orthogonal to the levels.

- ◆ If the link flow direction is towards the bottom or top, the levels are horizontal rows and the swim lanes are vertical columns.
- ◆ If the flow direction is towards the left or right, the levels are vertical columns and the swim lanes are horizontal rows.

Swim lanes can be used if the nodes are partitioned into groups, to indicate which nodes belong to a certain group. The nodes of the same swim lane are placed so that it is possible to draw a surrounding rectangle around them. Swim lanes allow you to organize the graph in a table-like manner. For instance, you may have a workflow diagram where nodes represent actions; then the swim lanes could represent the departments that perform these actions. Each node can belong to only one swim lane.

To associate a group of nodes with the same swim lane:

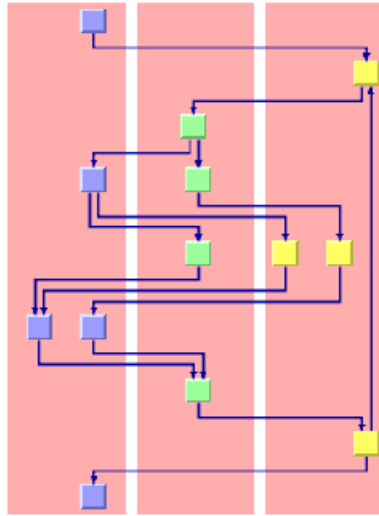
In Java

In Java™, call:

```
layout.addConstraint(new IlvSwimLaneConstraint(new IlvNodeGroup(nodeVector)))
;
```

All nodes of the node vector will be placed in the same swim lane rectangle. If a graph has many swim lane rectangles, the relative order of these swim lanes is determined automatically. The size of the swim lane rectangle depends on the nodes that belong to the swim lane. However, you can specify the relative order, relative size, and the margins of the swim lane as well by using the constructor:

```
public IlvSwimLaneConstraint(IlvNodeGroup group,
                             float relativeSize,
                             int positionIndex,
                             float minMargin)
```



The red background rectangle indicates where the swim lane is.

Swim Lanes

The relative size indicates how large this swim lane is compared to the other swim lanes. Assume that the flow direction is towards the bottom. In this case, the relative size indicates the width of the swim lane. All swim lanes with the same relative size will have the same width. A swim lane with a relative size that is twice the value of another swim lane will have twice the width of the other swim lane. The actual number of this parameter does not matter, only how large the value is compared to the other swim lanes. If you do not want to restrict the size of the swim lane, set the value to 0. In this case, the width of the swim lane will be independent of the other swim lanes.

The minimal margin is the margin of the swim lane in absolute coordinates. If the flow direction is towards the bottom, then it is the minimal horizontal distance between the leftmost or rightmost node of the swim lane and the swim lane border.

The position index indicates the order of the swim lanes. Just as nodes have position indices, the swim lanes are placed sequentially at relative positions numbered from 0 to n. In a top-down layout, the swim lane with position 0 is the leftmost swim lane, and the swim lanes with higher position indices are placed farther to the right. If the swim lanes have the position index -1, the layout algorithm determines the appropriate position automatically.

A swim lane constraint is always evaluated, even if the incremental mode is enabled. The constraint has a higher priority than the relative position constraint and the side-by-side constraint. You can specify side-by-side constraints for a group of nodes that belong to the same swim lane, but side-by-side constraints of nodes of different swim lanes are ignored. You can specify relative position constraints between nodes of the same swim lane. You can also specify relative position constraints between one entire swim lane group and another swim lane group, which effectively orders the swim lanes. But relative position constraints are ignored if they would require breaking the swim lanes apart. The swim lane constraint dominates the specified position indices and the extremity constraints, that is, if a swim lane constraint is used, you cannot specify position indices or east/west extremity constraints for any node.

Tip: The automatic conflict resolution can handle conflicting constraints. However, to speed up the layout, it is recommended that you specify constraints in such a way that there are no conflicts.

Constraint priorities (HL)

A set of constraints may cause conflicts. This means that not all of the constraints can be satisfied at the same time. For instance, it is impossible to force two nodes into the same level by an `IlvSameLevelConstraint` while at the same time forcing one of the nodes to a higher level than the other node by an `IlvRelativeLevelConstraint`. In this case, one of the two constraints must be ignored during layout.

In general, constraint conflicts are resolved by ignoring the constraints with the lowest priority while the constraints with the highest priority get satisfied. The following rules explain the constraint priorities in detail.

- ◆ The constraints that influence into which level a node is placed are applied before the constraints that influence the position of the node within a level.
- ◆ The `IlvExtremityConstraint` is translated into a sequence of constraints with high priority. For instance, the extremity constraint with the south side is translated into several same level constraints and several relative level constraints.
- ◆ The `IlvSameLevelConstraint` and the `IlvGroupSpreadConstraint` have the highest priority. They are never in conflict with each other. They dominate all other constraints, even the specified level index.
- ◆ The `IlvLevelRangeConstraint` (and the direct level index specification) has the second highest priority. If two nodes are forced to the same level but have disjoint specified level ranges, then the level range is ignored. In the following example:

```
layout.addConstraint(new IlvSameLevelConstraint(node1, node2));  
layout.setSpecNodeLevelIndex(node1, 5);  
layout.setSpecNodeLevelIndex(node2, 10);
```

both `node1` and `node2` will be placed at level 5. The conflicting specification: `layout.setSpecNodeLevelIndex(node2, 10)` is ignored.

- ◆ The `IlvRelativeLevelConstraint` is dominated by the same level constraint, by the level range constraint, and by the direct specification of level indices. If several relative level constraints conflict each other, the one with the highest specified priority dominates. However, note that all links are implicitly considered relative level constraints as well. If links with high priority force a node to a certain level, then a relative level constraint with lower priority will be ignored.
- ◆ The `IlvSwimLaneConstraint` is always evaluated, even if the incremental mode is enabled. The constraint has a higher priority than the relative position constraint and the side-by-side constraint. You can specify side-by-side constraints for a group of nodes that belong to the same swim lane, but side-by-side constraints of nodes of different swim lanes are ignored. You can specify relative position constraints between nodes of the same swim lane. You can also specify relative position constraints between one entire swim lane group and another swim lane group, which effectively orders the swim lanes. But relative position constraints are ignored if they would require breaking the swim lanes apart. The swim lane constraint dominates the specified position indices and the extremity constraints, that is, if a swim lane constraint is used, you cannot specify position indices or east/west extremity constraints for any node.

- ◆ The `IlvSideBySideConstraint` is evaluated only if the corresponding nodes belong to the same level. Typically you will use a same level constraint to force the nodes to the same level, and then a side-by-side constraint to force the nodes to a certain ordering. The side-by-side constraints dominate the relative position constraints. If several side-by-side constraints are conflicting, the one with the highest specified priority dominates the other constraints.
- ◆ The `IlvRelativePositionConstraint` is also evaluated only if the corresponding nodes belong to the same level. It is dominated by the side-by-side constraint; however, conflicts with side-by-side constraints are rare. If several relative position constraints are conflicting, the one with the highest specified priority dominates the other constraints.

For experts: constraint validation (HL)

Constraints that you specify in Java™ may become invalid. For instance, if you add a constraint that node A must be to the left side of node B, but you remove A from the graph, then this constraint becomes invalid. It simply does not make sense any more, even though it does not conflict with any other constraint. The layout instance automatically removes invalid constraints from time to time because they are a waste of memory. The validation check is done during layout. Forcing a validation check is normally not necessary but if you want to do this, call:

```
layout.validateConstraints();
```

This removes all invalid constraints from the Hierarchical Layout and cleans up the memory. The constraint validation does not check which constraints have conflicts. The main effect of the validation is that the constraint system uses less memory afterwards.

Note: A constraint is valid if it is meaningful. Two valid constraints are conflicting if the system cannot satisfy them both at the same time. Invalid constraints cannot be conflicting because they are meaningless.

Hence, constraint validation and constraint resolution are different phases. Constraint validation performs a quick local test. It removes invalid constraints from the layout instance completely. It does not affect conflicting constraints.

Constraint resolution checks whether a set of valid constraints are in conflict with each other. Thus, constraint resolution is a complex process on a network of multiple related constraints. Constraint resolution decides which constraints can be solved and which cannot. But the constraint resolution does not remove conflicting constraints from the layout instance, it just delivers a solution that may ignore some constraints.

For experts: more indices (HL)

The Hierarchical Layout allows you to specify the level index and the position index of a node.

In Java™

You specify the level and position index of a graphic node in the following way:

```
layout.setSpecNodeLevelIndex(node, 5);  
layout.setSpecNodePositionIndex(node, 33);
```

How these indices are used depends on the graph topology and the additional constraints. For example, the specified level index can be in conflict with some `IlvLevelRangeConstraint` or `IlvSameLevelConstraint`. In this case, the constraint priorities determine how the conflict is resolved (see *Constraint priorities (HL)*). If the incremental mode is switched on, the specified node level and position index are ignored, since the incremental mode tries to preserve old node positions. It is also possible to obtain the indices of nodes that were calculated during layout.

Calculated level index

The layout algorithm allows you to access the level index that was calculated for a node by a previous layout. To do this, use the method:

```
int getCalcNodeLevelIndex(Object node)
```

If the node was never laid out, this method returns -1. Otherwise, it returns the previous level index of the node.

In an application that specifies layout parameters entirely programmatically, the method can be used to specify the level index for the next layout in the following way:

```
int index = layout.getCalcNodeLevelIndex(node);  
layout.setSpecNodeLevelIndex(node, index);
```

When this is done, it ensures that the node is placed at the same level as in the previous layout.

If the graph is detached from the layout algorithm, the calculated level index of a node is set back to -1.

Note: You should be aware of the difference between the methods

`getCalcNodeLevelIndex(java.lang.Object)` and `getSpecNodeLevelIndex(java.lang.Object)`. The first one returns the level index calculated by the previous layout. The second one returns the specified level index, even if there was no previous layout.

For instance, consider two nodes A and B. Node A has no specified level index and node B has a specified level index 5. Before the first layout, the method `getCalcNodeLevelIndex` returns -1 for both nodes because the levels have not been calculated yet. However, `getSpecNodeLevelIndex` returns -1 for A and 5 for B. After the first layout, node A may be placed at level 4. Now, `getCalcNodeLevelIndex` returns 4 for node A and 5 for node B and `getSpecNodeLevelIndex` still returns -1 for A and 5 for B.

Calculated position index

The layout algorithm allows you to access the position index within a level that was calculated for a node by a previous layout. To do this, use the method:

```
getCalcNodePositionIndex(java.lang.Object)
```

If the node was never laid out, this method returns -1. Otherwise, it returns the previous position index of the node within its level.

To ensure that the node is placed at the same level at the same relative position as in the previous layout, use in an application that specifies layout parameters entirely programmatically the following:

```
layout.setSpecNodeLevelIndex(node,  
    layout.getCalcNodeLevelIndex(node));
```

This example code works only if the generic connected component layout is disabled and the port sides `EAST` or `WEST` are not used in the layout.

If the graph is detached from the layout algorithm, the calculated position index of a node is set back to -1.

Note: You should be aware of the difference between the methods `getCalcNodePositionIndex` and `setSpecNodePositionIndex`. The first one returns the position index calculated by the previous layout and -1 if there was no previous layout. The second one returns the specified position index even if there was no previous layout. This behavior is similar to the behavior of the specified and calculated level index (see *Calculated level index*).

Recursive layout

JViews Diagrammer supports nested graphs, that is, it can render graphs containing nodes that are graphs. A graph that is a node in another graph is called a subgraph. Links that connect nodes of different subgraphs are called intergraph links. In Recursive hierarchical layout on nested graph with polyline link style, all red links are intergraph links and all black links are normal links. This is explained in detail in *Nested layouts*.

The hierarchical layout can treat a nested graph at once, placing all nested nodes and routing all links including the intergraph links. It can even place the labels in the nested graph.

To enable the recursive mode:

In Java™

Use this method:

```
void setRecursiveLayoutMode(boolean enable);
```

and call `performLayout` with the third parameter set to `true` in the following way:

```
layout.performLayout(force, redraw, true);
```

Label layout

If the recursive layout mode is enabled, the hierarchical layout can also place the node and link labels. This is useful, because placing labels after a recursive layout may change the bounds of subgraphs again, and hence would require the hierarchical layout to rerun. Therefore, an annealing label layout is integrated into the hierarchical layout which is executed during the recursive layout mode. In order to set label descriptors, you can access this label layout by using the following code:

```
public IlvAnnealingLabelLayout getLabelLayout()
```

If the labels are contained in a subgraph, use the following code:

```
// node and label are directly contained in subgraph
IlvHierarchicalLayout sublayout =
    (IlvHierarchicalLayout) topLevelLayout.getRecursiveLayout().getLayout
(subgraph);
IlvAnnealingLabelLayout labellayout = sublayout.getLabelLayout();
lvAnnealingPointLabelDescriptor d =
    new IlvAnnealingPointLabelDescriptor(label, node,
IlvAnnealingPointLabelDescriptor.RECTANGULAR,
    IlvDirection.Bottom);
labellayout.setLabelDescriptor(label, d);
```

When the recursive layout mode is used, the label layout is automatically used. It is recommended to keep it enabled if nodes or links in subgraphs have labels. It can be disabled if there are no labels.

To disable the label layout:

In Java

```
layout.setLabelLayoutEnabledDuringRecursiveLayoutMode(false);
```

For more details on how to use the `IlvAnnealingLabelLayout` see *Annealing label layout*.
For more details on how to use the `IlvRecursiveLayout` see *Recursive layout*.

Link layout (LL)

Describes the *Link Layout* algorithm (class `IlvLinkLayout` from the package `ilog.views.graphlayout.link`).

In this section

General information on the LL

Provides samples of Link Layout and explains where it is likely to be used.

Features and limitations of the LL

Lists the features and limitations of the layout.

The LL algorithms

Describes how the algorithm for each mode operates.

Generic features and parameters of the LL

Describes the generic features and parameters of the layout.

Specific parameters for both LL modes

Describes the parameters that are specific to the `IlvLinkLayout` class.

Spacing parameters in short link mode

Describes how to use the spacing parameters in short link mode.

Spacing parameters in long link mode

Describes how to use the spacing parameters in long link mode.

For experts: additional features of LL

Describes the features available in both Link Layout modes.

For experts: special options of the Short LL

Describes the options of the Short Link Layout for expert use.

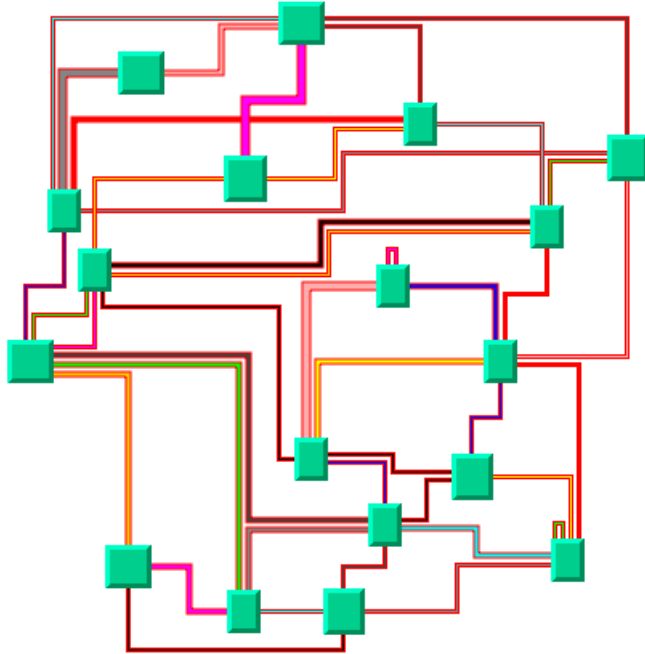
For experts: special options of the Long LL

Describes the options of the Long Link Layout for expert use.

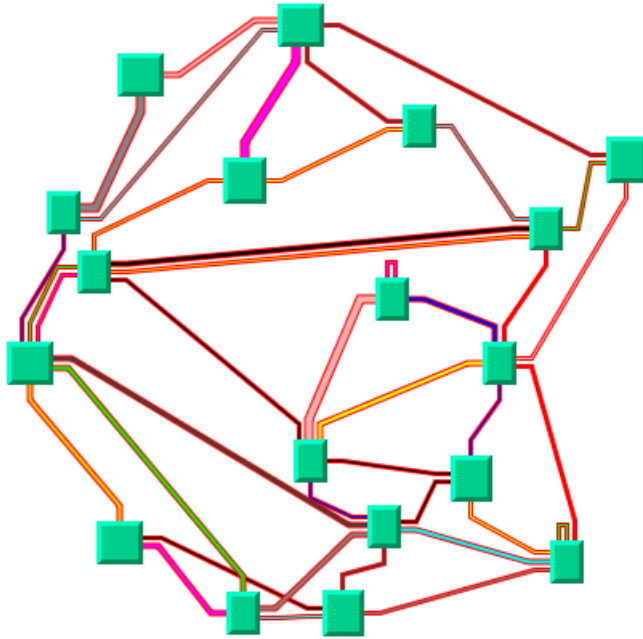
General information on the LL

LL samples

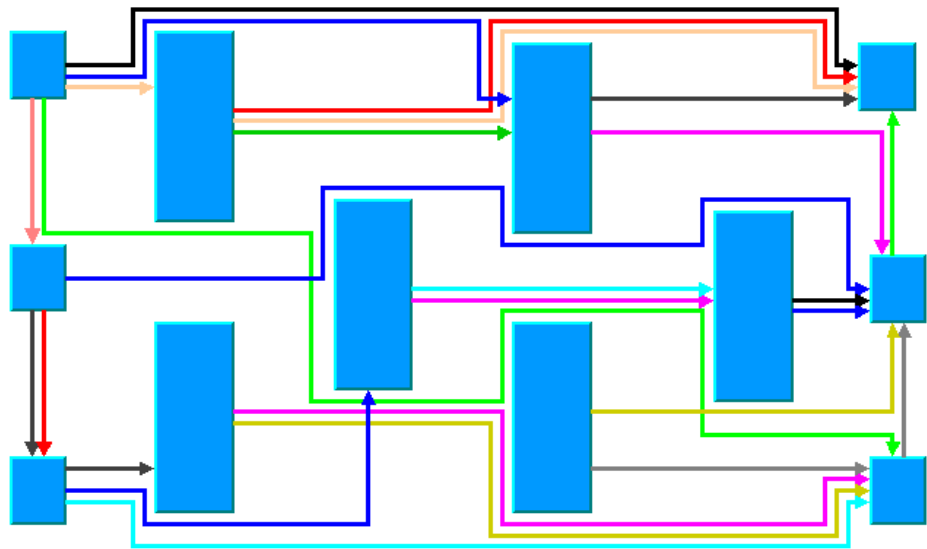
These sample drawings were produced with the Link Layout algorithm:



Link layout in short link mode with orthogonal links



The same graph in short link mode with direct links



Link layout in long link mode with orthogonal links

What types of graphs suit the LL?

Any type of graph where nodes are fixed and links need to be routed:

- ◆ connected graphs and disconnected graphs
- ◆ planar graphs and nonplanar graphs.
- ◆ nested graphs with intergraph links

Application domains for the LL

Application domains of the Link Layout include:

- ◆ Electrical engineering (circuit block diagrams)
- ◆ Industrial engineering (schematic design diagrams, equipment/resource control charts)
- ◆ Business processing (entity relation diagrams)
- ◆ Software management/software (re-)engineering (data inspector diagrams)
- ◆ Database and knowledge engineering (sociology, genealogy)
- ◆ CASE tools (design diagrams)

Features and limitations of the LL

Features of both modes (LL)

- ◆ Reshapes the links of a graph in either an orthogonal or a direct style, without moving the nodes. Orthogonal and direct style links can be combined in the same layout.
- ◆ Allows you to specify which side of the node (top, bottom, left, or right) a link can be connected to, or to preserve the existing connection points of the links.
- ◆ Supports self-links (that is, links with the same origin and destination node).
- ◆ Supports multiple links (that is, more than one link between the same origin and destination nodes).
- ◆ Allows you to specify pinned (fixed) links that the layout algorithm cannot reshape.
- ◆ Supports intergraph links of nested graphs. An intergraph link is a link whose end nodes belong to different subgraphs of a nested graph.
- ◆ Supports an incremental mode: If new links are added to a drawing, the next layout takes the shapes of the old links into account.
- ◆ Two layout modes: *short links* with a limited number of bends or *long links* with unlimited number of bends.

Features of short link mode (LL)

- ◆ Links are placed freely in the space.
- ◆ Link-to-link and link-to-node crossings are reduced, if this is possible with link shapes that have a maximum of 4 bends.
- ◆ Links of different width are supported.
- ◆ Link bundles between the same pair of nodes are supported. Optionally, the algorithm can ensure that multiple links are bundled together by giving them parallel shapes.
- ◆ Automatically arranges the final segments of the links (the segments near the origin or destination node) to obtain a bundle of parallel links.
- ◆ Provides two optional shapes for the self-links.
- ◆ Very fast algorithm with low memory footprint.

Features of long link mode (LL)

- ◆ Links are placed on a grid.
- ◆ Link-to-node crossings of orthogonal links are avoided, even if this introduces many bends.
- ◆ Orthogonal link segments do not overlap.

- ◆ Does not bundle the final segments. Instead, it distributes the links on the border of each end node according to which border has more free space.
- ◆ Fast algorithm: speed and memory footprint depend on the grid spacing.

Limitations

- ◆ When routing intergraph links, the incremental mode cannot be used. Due to the complexity of intergraph link routing, more crossings and overlappings may occur than when routing normal links.
- ◆ In short link mode, crossings and overlapping of links with other links and nodes cannot always be avoided because the algorithm uses link shapes with a limited number of bends. This happens in particular when there are many obstacles between the end points of a link.
- ◆ In long link mode, link crossings cannot always be avoided. Segment overlappings of orthogonal links are always avoided unless there is no free space remaining on the border of the end nodes. Any overlapping of nodes and links is always avoided unless one end node is inside an enclave. An enclave is an area that is surrounded by other nodes such that the area cannot be reached from the other end node. (See *A Node inside an enclave.*)
- ◆ In long link mode, segment overlapping or overlapping between nodes and links cannot always be avoided if the direct link style is used.
- ◆ The long link mode is slower and uses more memory if the grid spacing is very small.

The LL algorithms

The Link Layout algorithm utilizes two sublayout classes:

- ◆ `IlvShortLinkLayout` in short link mode.
- ◆ `IlvLongLinkLayout` in long link mode.

They implement different strategies to find the link shapes.

Short Link Layout algorithm

The Short Link Layout algorithm is based on a combinatorial optimization that chooses the “optimal” shape of the links to minimize a cost function. This cost function is proportional to the number of link-to-link and link-to-node crossings.

For efficiency reasons, the basic shape of each link is chosen from a set of predefined shapes. These shapes are different for each link style option. For the orthogonal link style, the links are reshaped to a polygonal line of up to five alternating horizontal and vertical segments (see *Link layout in short link mode with orthogonal links*). For the direct link style, the links are reshaped to a polygonal line composed of three segments: a straight-line segment that starts and ends with small horizontal or vertical segments (see *The same graph in short link mode with direct links*).

The shape of a link also depends on the relative position of the origin and destination nodes. For instance, when two nodes are very close or they overlap, the shape of the link is chosen to provide the best visibility of the link.

The exact shape of a link is computed taking into account additional constraints. The layout algorithm tries to do the following:

- ◆ Minimize the number of crossings between the links incident to a given side of a node.
- ◆ Space the final segments of the links incident to a given side of a node equally on the node border.

Long Link Layout algorithm

The Long Link Layout algorithm first treats each link individually. For each link, it first calculates the connection points at the end nodes that are on the grid and orders them according to a penalty value. Connection points on used grid points have a very high penalty and, therefore, are very unlikely to be used.

For the orthogonal links (see *Link layout in long link mode with orthogonal links*), the Long Link Layout algorithm then uses a grid traversal to search a route over free grid points from the start connection point to the end connection point. Therefore, in contrast to the short link mode, orthogonal links can have any shape with a large number of bends if this is necessary to bypass obstacle nodes to avoid overlappings. For the direct links (see *The same graph in short link mode with direct links*), it shortens the search by using a direct segment between the connection points.

After all links are placed, a crossing reduction phase examines pairs of links and eliminates link crossings by exchanging parts of the routes between both links.

The Long Link Layout algorithm relies on the fact that links fit to the grid spacing and parts of the routes between different links can be exchanged. Therefore, the Long Link Layout algorithm does not take the link width into account because it would be too difficult to find the parts of two links that can be exchanged. It is recommended to set the grid spacing larger than the largest link width.

Example of Link Layout

In Java

Below is a code sample using the `IlvLinkLayout` class. This code sample shows how to perform a Link Layout:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.link.*;
...

IlvLinkLayout layout = new IlvLinkLayout();
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);

/* Specify the layout mode */
layout.setLayoutMode(IlvLinkLayout.SHORT_LINKS);

try {
    IlvGraphLayoutReport layoutReport = layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
layout.detach();
graphModel.dispose();
```

Generic features and parameters of the LL

The `IlvLinkLayout` class supports the following generic parameters as defined in the class `IlvGraphLayout` (see *Base class parameters and features*):

- ◆ *Allowed time (LL)*
- ◆ *Automatic layout (LL)*
- ◆ *Preserve fixed links (LL)*
- ◆ *Stop immediately (LL)*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed time (LL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) If the layout stops early because the allowed time has elapsed, some links may not be routed in the best possible way. The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Automatic layout (LL)

The Link Layout routes the links so that they bypass the nodes and cross each other as few times as possible. It does not position any nodes. However, if the user moves, adds, or resizes nodes, or adds or removes links, the Link Layout drawing usually becomes invalid; that is, the links no longer look orthogonal, overlap the moved nodes, or cross other links.

Using the automatic layout feature of the `IlvGraphLayout` class, the layout is reperformed whenever a change of the graph occurs. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Automatic layout*.)

Preserve fixed links (LL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links*.) The fixed links are taken into account when computing the optimal layout of the nonfixed links.

Stop immediately (LL)

The layout algorithm stops if the method `IlvLinkLayout` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early, some links may not be routed in the best possible way. The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters for both LL modes

Layout mode (LL)

The Link Layout algorithm has two layout modes.

To select a layout mode:

In Java™

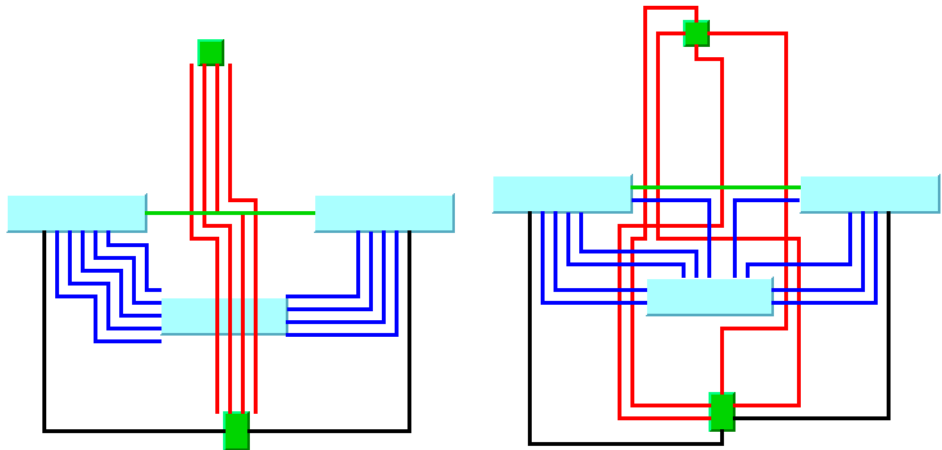
Use the method:

```
void setLayoutMode(int mode);
```

The valid values for `mode` are:

- ◆ `IlvLinkLayout.SHORT_LINKS` (the default)
- ◆ `IlvLinkLayout.LONG_LINKS`

Short and Long Link Modes with Orthogonal Links shows a small sample graph in short and long link mode. The short link mode bundles the links very well. However, due to the bundling, some red links appear to be unconnected to the green nodes. Furthermore, the algorithm cannot find a route for the long red links without overlapping some nodes or without overlapping the green link. The long link mode works on a grid. It is specialized for long links and avoids overlapping any nodes or link segments. It can connect to the green nodes by choosing connection points on different sides of the end nodes. This advantage, however, is paid for by a less regular structure that does not bundle the links and a larger number of link crossings.



Short Link Layout Mode

Long Link Layout Mode

Short and Long Link Modes with Orthogonal Links

Choosing the appropriate layout mode (LL)

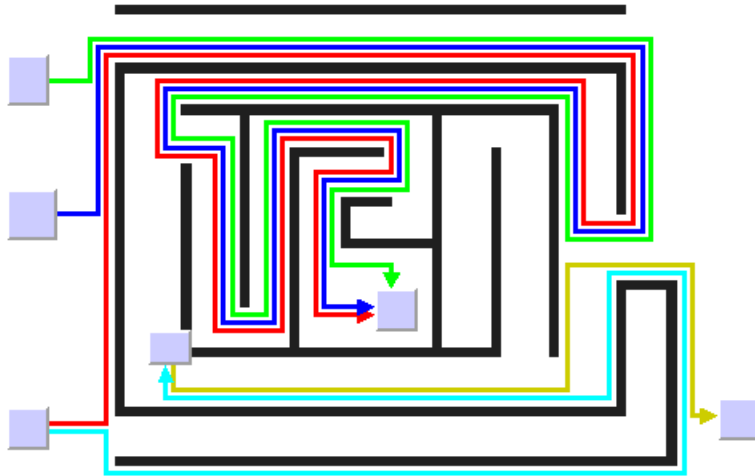
The short link mode should be used if any of the following conditions apply:

- ◆ The majority of links is short and it is not fatal if long links overlap obstacles.
- ◆ The link routes must be placed freely and cannot be restricted to a grid.
- ◆ It is important to limit the number of bends.

The long link mode should be used if any of the following conditions apply:

- ◆ Many links are long and it is important that long links do not overlap obstacles.
- ◆ There is a preferred routing because the nodes are already placed on the grid.
- ◆ It is important to have a guaranteed minimal distance between link segments.
- ◆ An increasing number of bends is acceptable if it avoids any overlappings.

Labyrinth routing with the long link mode shows how the long link mode can be used to find an orthogonal route without overlappings in a labyrinth of node obstacles.



Labyrinth routing with the long link mode

Link style (LL)

The layout algorithm provides two link styles. You can set the link style globally, in which case all links have the same kind of shape, or locally on each link, in which case different link shapes occur in the same drawing.

Global link style

Example of setting global link style (Link Layout algorithm)

To set the global link style:

In Java

Use the method:

```
void setGlobalLinkStyle(int style);
```

The valid values for `style` are:

◆ `IlvLinkLayout.ORTHOGONAL_STYLE` (the default)

The links are reshaped in an orthogonal form (alternating horizontal and vertical segments). See *Link layout in short link mode with orthogonal links* and *Link layout in long link mode with orthogonal links* as examples.

◆ `IlvLinkLayout.DIRECT_STYLE`

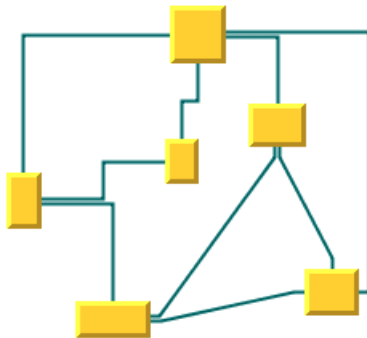
The links are reshaped to a polygonal line composed of three segments: a straight-line segment that starts and ends with a small horizontal or vertical segment. See *The same graph in short link mode with direct links* as an example.

◆ `IlvLinkLayout.MIXED_STYLE`

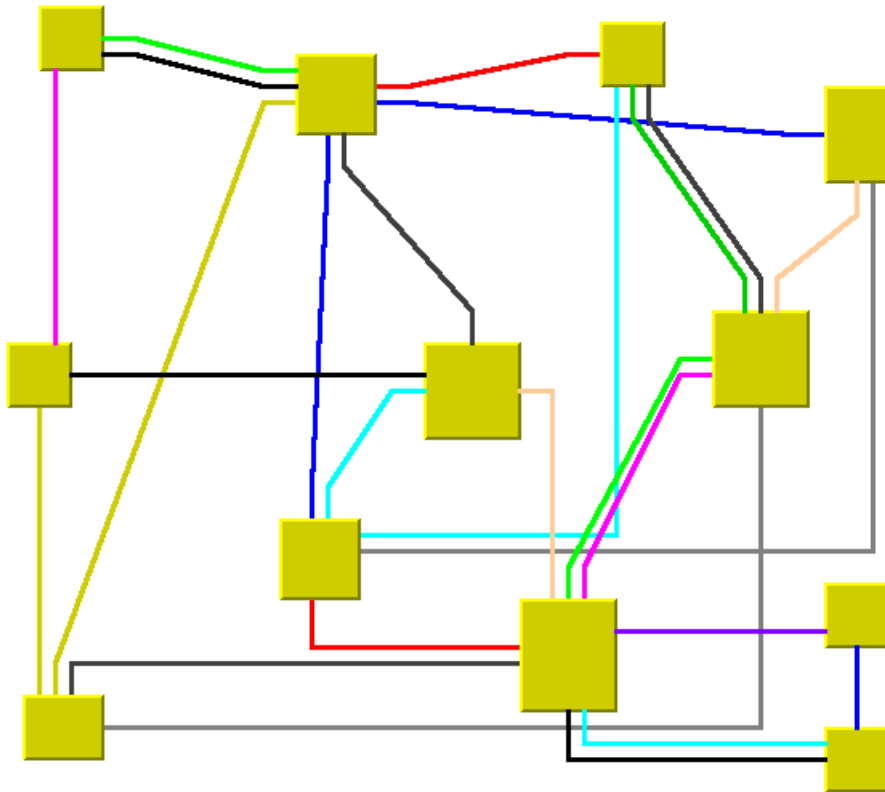
Each link can have a different link style. The style of each individual link can be set to have different link shapes occurring on the same graph.

Individual link style

All links have the same style of shape unless the global link style is `IlvLinkLayout.MIXED_STYLE`. Only when the global link style is set to `MIXED_STYLE` can each link have an individual link style.



Different link styles mixed in the same drawing (short link mode)



Different link styles mixed in the same drawing (long link mode)

Example of specifying individual link style (Link Layout algorithm)

To set and retrieve the style of an individual link:

In Java

Use the methods:

```
void setLinkStyle(Object link, int style);
```

```
int getLinkStyle(Object link);
```

The valid values for style are:

- ◆ `IlvLinkLayout.ORTHOGONAL_STYLE` (the default)
- ◆ `IlvLinkLayout.DIRECT_STYLE`
- ◆ `IlvLinkLayout.NO_RESHAPE_STYLE` (that is, the link is not reshape in any manner)

End points mode (LL)

Normally, the layout algorithm is free to choose the termination points of each link. However, the user can specify that the current fixed termination pin of a link should be used.

The layout algorithm provides two end point modes. You can set the end point mode globally, in which case all end points have the same mode, or locally on each link, in which case different end point modes occur in the same drawing.

Global end point mode

Example of specifying global end point mode (Link Layout algorithm)

To set the global end point mode:

In Java

```
void setGlobalOriginPointMode(int mode);
```

```
void setGlobalDestinationPointMode(int mode);
```

The valid values for `mode` are:

◆ `IlvLinkLayout.FREE_MODE` (the default)

The layout is free to choose the appropriate position of the connection point on the origin/destination node.

◆ `IlvLinkLayout.FIXED_MODE`

The layout must keep the current position of the connection point on the origin/destination node.

◆ `IlvLinkLayout.MIXED_MODE`

Each link can have a different end point mode.

Individual end point mode

All links have the same end point mode unless the global end point mode is `IlvLinkLayout.MIXED_MODE`. Only when the global end point mode is set to `MIXED_MODE` can each link have an individual end point mode.

Example of specifying individual end point mode (Link Layout algorithm)

To set the mode of an individual link:

In Java

Use the methods:

```
void setOriginPointMode(Object link, int mode);
```

```
int getOriginPointMode(Object link);
```

```
void setDestinationPointMode(Object link, int mode);
```

```
int getDestinationPointMode(Object link);
```

The valid values for mode are:

- ◆ `IlvLinkLayout.FREE_MODE` (the default)
- ◆ `IlvLinkLayout.FIXED_MODE`

Incremental mode (LL)

The Link Layout algorithm normally routes all links from scratch. If the graph changes incrementally because you add or remove links or nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode.

Example of enabling incremental mode (Link Layout algorithm)

To enable the incremental mode:

In Java
Call:

```
layout.setIncrementalMode(true);
```

In incremental mode, the layout tries to minimize the changes to the layout. A link is only rerouted if it is new, if a link bend moved, if its layout parameters have changed, or if a node was moved such that it overlaps the link.

In short link mode, if the next layout is incremental, the links preserve the connection side and the general shape calculated by a previous layout, except if one of their end nodes has been moved or resized.

In the long link mode, a new route is searched for the links that are no longer on the grid or that overlap with nodes. The shape and the connection side of the rerouted links can change completely. However, links that are already on the grid and do not overlap nodes or other links are not rerouted in incremental mode. It is also possible to specify which link must be rerouted by the next incremental layout even though the layout has not changed.

Example of specifying which link must be rerouted by the next incremental layout (Link Layout algorithm)

To select an individual link to be used for incremental rerouting:

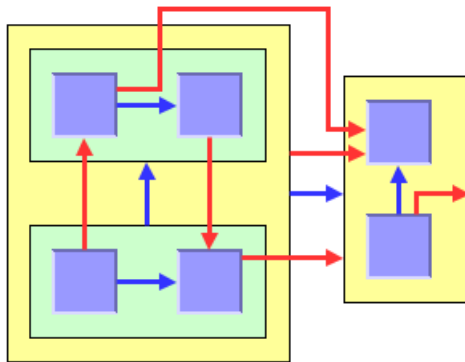
In Java

Use the method:

```
void markForIncremental(Object link);
```

Intergraph link routing (LL)

A nested graph is a graph with nodes that are subgraphs. In a nested graph, normal links and intergraph links can occur. Normally, both end nodes of a link belong to the same subgraph. Intergraph links are those links whose end nodes belong to different subgraphs. Intergraph links belong to the lowest common grapher in the nesting structure that contains both end nodes. The following figure shows a nested graph with blue normal links and red intergraph links.



Nested Graph With Normal Links (blue) and Intergraph Links (red)

By default, the Link Layout routes both the normal links and the intergraph links.

Example of routing only normal links (Link Layout algorithm)

In order to route only normal links, disable intergraph link routing:

In Java™

Call:

```
layout.setInterGraphLinksMode(false);
```

Example of routing intergraph and/or normal links (Link Layout algorithm)

If the intergraph links mode is enabled, you can select whether only the intergraph links are routed, or whether the intergraph links and the normal links are routed at the same time.

In Java

If you call:

```
layout.setCombinedInterGraphLinksMode(false);
```

the next layout routes the intergraph links but does not reshape any normal links. If you call:

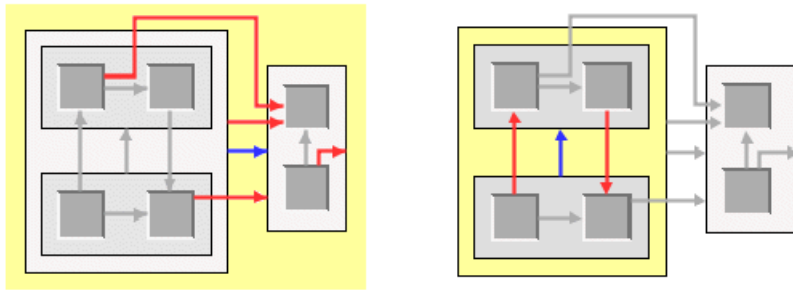
```
layout.setCombinedInterGraphLinksMode(true);
```

the next layout routes both the normal links and the intergraph links.

When the intergraph links mode is enabled, the layout cannot route the links incrementally (see *Incremental mode (LL)*).

Notice that the layout routes only those links that belong to the attached graph. In a nested graph, each subgraph is attached to a different layout instance. Therefore, when starting a normal (nonrecursive) layout for the top-level graph (see *Nested Graph With Normal Links (blue) and Intergraph Links (red)*) not all links are routed that are shown in this figure, but only those links that belong to the top-level graph.

The following figure shows two situations: the yellow subgraph indicates the subgraph where the nonrecursive layout is currently applied, and color of the links indicate which links are currently routed. Depending on the intergraph links mode, the red and/or blue links are routed, but the grey links are not reshaped.



Routed Link in a Nested Graph when Layout is Performed for the Yellow Subgraph

To route *all links* of a nested graph, you need to apply the Link Layout recursively. Details of the recursive layout mechanism are explained in *Recursive layout*. For instance:

```
layout.setInterGraphLinksMode(true);
layout.performLayout(force, redraw, true);
```

routes the intergraph links recursively in all subgraphs. If you use a layout provider (a class that implements the interface `IlvLayoutProvider`), you need to set the intergraph links mode for all subgraphs explicitly:

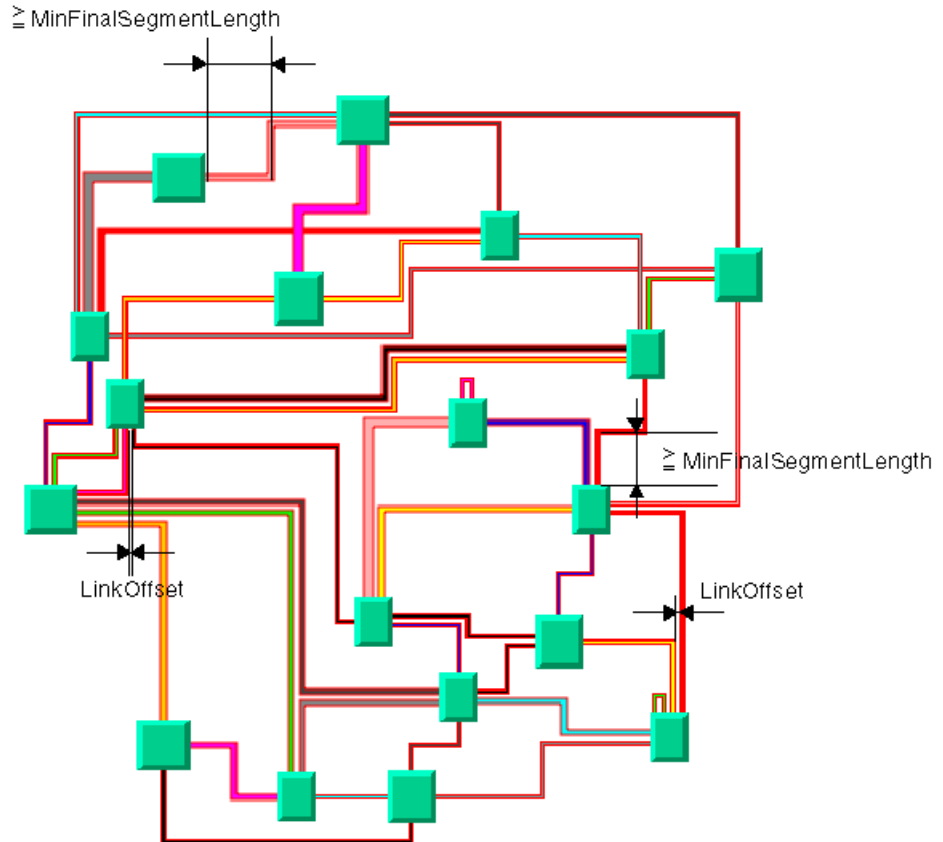
```
IlvLayoutProvider layoutProvider = ...
// first, set the intergraph mode for all layouts
Enumeration e = graphModel.getLayouts(layoutProvider, true);
while (e.hasMoreElements()) {
    IlvGraphLayout layout = (IlvGraphLayout) e.nextElement();
    if (layout instanceof IlvLinkLayout)
        ((IlvLinkLayout) layout).setInterGraphLinksMode(true);
}
// then perform layout recursively using the provider
graphModel.performLayout(layoutProvider, force, redraw, true);
```

If you want to recursively perform the intergraph link routing in combination with a layout that places the nodes or that arranges labels, we recommend that you use an instance of the class `IlvMultipleLayout` to encapsulate the Link Layout and the other layouts, and then perform the Multiple Layout recursively all at once. For details, see *Recursive layout*.

Spacing parameters in short link mode

Since the short link mode places the links freely in the space, only two parameters are necessary to control the spacing: the minimal distance between links and the minimal length of the final segment.

Spacing Parameters for the Short Link Mode shows the spacing parameters used in the short link mode.



Spacing Parameters for the Short Link Mode

Link offset

The layout algorithm computes the final connecting segments of the links (that is, the segments near the origin and destination nodes) to obtain parallel lines spaced at a user-defined distance. In short link mode, the algorithm takes into account the width of the links when computing the offset.

Example of specifying link offset (Link Layout algorithm)

To specify the offset:

In Java™

Use the method:

```
void setLinkOffset(float offset)
```

The offset is measured from the border of one link to the nearest border of the other link. Therefore, if the specified offset is zero, the border of a link touches the border of its neighbor link.

Minimum final segment length

You can specify a minimum value for the length of the final connecting segments of the links (that is, the segments near the origin and destination nodes).

Example of specifying minimum final segment length (Link Layout algorithm)

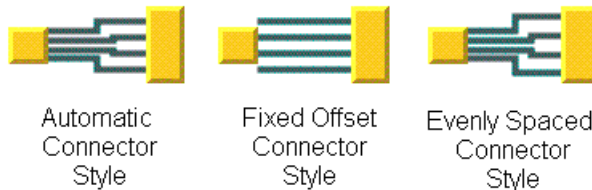
In Java

Use the method:

```
void setMinFinalSegmentLength(float length)
```

Connector style

The layout algorithm positions the end points of links (the connector pins) at the nodes automatically. The connector style parameter specifies how these end points are calculated.



Connector styles

The layout algorithm provides two connector styles. You can set the connector style globally, in which case all the nodes (hence, all the links) have the same kind of connector style, or locally on each node (that is, for all the links connected to the node), in which case different connector styles occur in the same drawing.

Global connector style

Example of specifying the global connector style (Link Layout algorithm)

To specify the global connector style:

In Java

Use the following method:

```
void setGlobalConnectorStyle(int style);
```

The valid values for `style` are:

◆ `IlvShortLinkLayout.FIXED_OFFSET_PINS`

The connection pins are spaced along the node border at a distance equal to the link offset parameter. See *Spacing Parameters for the Short Link Mode* as an example.

◆ `IlvShortLinkLayout.EVENLY_SPACED_PINS`

The connector pins are evenly spaced along the node border, preserving a margin which is determined by the `evenlySpacedPinsMarginRatio` parameter (see the accessor `getEvenlySpacedPinsMarginRatio()`). See *Spacing Parameters for the Short Link Mode* as an example.

◆ `IlvShortLinkLayout.AUTOMATIC_PINS` (the default)

Uses the connector style `FIXED_OFFSET_PINS` except if this pushes a connection point outside the border the link is attached to, in which case it uses the connector style `EVENLY_SPACED_PINS`. See *Spacing Parameters for the Short Link Mode* as an example.

◆ `IlvShortLinkLayout.MIXED_STYLE`

Each node can have a different connector style. The style of each individual node can be set to have different connector styles occurring on the same graph.

Individual connector style

All nodes have the same connector style unless the global connector style is `IlvShortLinkLayout.MIXED_STYLE`. Only when the global connector style is set to `MIXED_STYLE` can each node have an individual connector style.

Example of specifying individual node connector style (Link Layout algorithm)

To specify the connector style of an individual node:

In Java

Use the following methods:

```
void setConnectorStyle(Object node, int style);
```

```
int getConnectorStyle(Object node);
```

The valid values for `style` are:

◆ `IlvShortLinkLayout.FIXED_OFFSET_PINS`

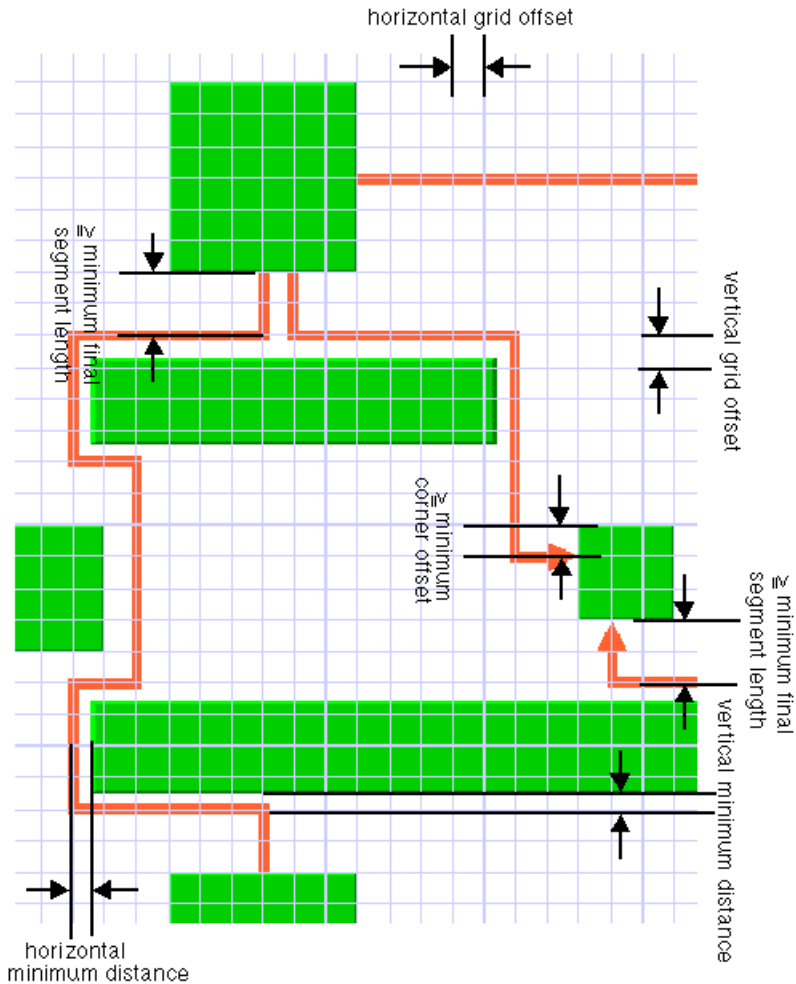
◆ `IlvShortLinkLayout.EVENLY_SPACED_PINS`

◆ `IlvShortLinkLayout.AUTOMATIC_PINS` (the default).

The default value is 10.

Spacing parameters in long link mode

The long link mode places the links on a grid. Four parameters control the grid offsets and five parameters control the spacing of links in relation to other objects. *Spacing parameters for the long link mode* shows the spacing parameters used in the long link mode.



Spacing parameters for the long link mode

Grid offset parameters

The grid offset parameters control the spacing between grid lines. Links are routed such that the center of the orthogonal link segments are on the grid lines. The grid offsets should be set to a value larger than the largest link width value to avoid links that visually overlap.

Example of specifying grid offset parameters (Link Layout algorithm)

To set the horizontal and vertical grid offset:

In Java

In Java™, use the methods:

```
void setHorizontalGridOffset(float offset);
```

```
void setVerticalGridOffset(float offset);
```

The grid offset is the critical parameter for the long link mode. If the grid offset is too large, there may be no grid lines between nodes even though some free space exists between the nodes. In this case, the link routings cannot use the free space. However, if the grid offset is too small, the algorithm needs a long time to traverse the grid.

Grid base parameters

Sometimes it is necessary to shift the whole grid by a small amount because the nodes are not aligned on the grid. For instance, to have grid lines at positions 3, 13, 23, 33, and so on, you can set the grid offset to 10 and the grid base to 3.

Example of specifying grid base parameters (Link Layout algorithm)

To adjust the grid base:

In Java

Use the methods:

```
void setHorizontalGridBase(float coordinate);
```

```
void setVerticalGridBase(float coordinate);
```

Minimum distance parameters

The minimum distance controls how closely a link can be placed to the border of a node that needs to be bypassed. If the node border is not aligned to the grid, the minimum distance specifies the next grid line close to the border that can be used. For instance, if a node covers the x-coordinates 25 to 65 on a grid with offset 10 and base 0, the next grid lines used to bypass the node would normally be at 20 and 70. If you specify a minimum distance of 8, these grid lines are too close to the node and then the grid lines at 10 and 80 would be used.

Example of specifying minimum distance parameters (Link Layout algorithm)

To set the minimum distance:

In Java

Use the methods:

```
void setHorizontalMinOffset(float offset);
```

```
void setVerticalMinOffset(float offset);
```

Minimum node corner offset parameter

The minimum corner offset is the minimum distance between a node corner and a link that connects to the node. This parameter is used to avoid having a link that connects exactly to the corner or outside the border of the node (see *Minimal corner offset*).

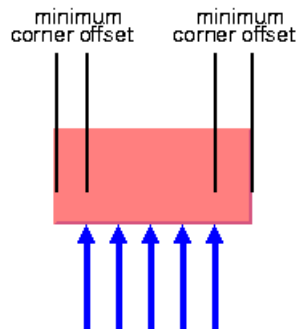
Example of specifying minimum node corner offset parameter (Link Layout algorithm)

To set the minimum corner offset:

In Java

Use the method:

```
void setMinNodeCornerOffset(float offset);
```



Minimal corner offset

Minimum final segment length

As with the short link mode, the long link mode respects the minimum value for the length of the final connecting segments of the links.

Example of specifying minimum final segment length (Link Layout algorithm)

To set the minimal length of the final segment:

In Java

Use the method:

```
void setMinFinalSegmentLength(float length)
```

For experts: additional features of LL

Using a node-side filter

Some applications require that links are not connected to specific sides of certain nodes. The Link Layout algorithm allows you to restrict to which node side a link can connect by using a node-side filter. A node-side filter is any class that implements the interface `IlvNodeSideFilter`. This interface defines the following method:

```
public boolean accept(IlvGraphModel graphModel,
                    Object link,
                    boolean origin,
                    Object node,
                    int side);
```

This method allows you to let the input link to connect its `origin` or `destination` to the input side of the input node.

As an example, assume that the application requires that for end nodes of type `MyNodeEditPart1`, links can connect their origin only at the top and bottom side.

For end nodes of type `MyNodeEditPart2`, links can connect their destination only at the left and right side. You can obtain this result with the following node-side filter:

```
class MyFilter implements IlvNodeSideFilter
{
    public boolean accept(IlvGraphModel graphModel,
                        Object link,
                        boolean origin,
                        Object node,
                        int side)
    {
        if (node instanceof MyNodeEditPart1 && origin)
            return(side == IlvDirection.Top || side == IlvDirection.Bottom);
        if (node instanceof MyNodeEditPart2 && !origin)
            return(side == IlvDirection.Left || side == IlvDirection.Right);
        return true;
    }
}
```

Example of setting node-side filter (Link Layout algorithm)

To set this node-side filter:

In Java

In Java™, call:

```
layout.setNodeSideFilter(new MyFilter());
```

To remove the node-side filter, call:

```
layout.setNodeSideFilter(null);
```

Using a node box interface

Some applications require that effective area of a node is not exactly its bounding box. For instance, if the node has a shadow, the shadow is included in the bounding box. However, the shadow may not be considered as an obstacle for the links. In this case, the effective bounding box of a node is smaller.

Example of using a node box interface (Link Layout algorithm)

In Java

You can modify the effective bounding box of a node by implementing a class that implements the `IlvNodeBoxInterface`.

This interface defines the following method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective bounding box. For instance, to set a node box interface that returns a smaller bounding box for all nodes of type `MyNodeEditPart`, call:

```
layout.setNodeBoxInterface(new IlvNodeBoxInterface() {
    public IlvRect getBox(IlvGraphModel graphModel, Object node) {
        IlvRect rect = graphModel.boundingBox(node);
        if (node instanceof MyNodeEditPart) {
            // for example, the size of the bounding box is reduced by 4
            units
            rect.resize(rect.width-4.f, rect.height-4.f);
        }
        return rect;
    }
});
```

Using a link connection box interface

By default, the connection points of the links are distributed on the border of the bounding box of the nodes. Sometimes, it may be necessary to place the connection points on a rectangle that is smaller or larger than the bounding box. For instance, this can happen when labels are displayed below or above nodes.

Example of using a link connection box interface to modify position of connection points (Link Layout algorithm)

In Java

You can modify the position of the connection points of the links by implementing a class that implements the `IlvLinkConnectionBoxInterface`. This is a subinterface of `IlvNodeBoxInterface` (see *Using a node box interface*). It defines again the method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

Additionally, the interface `IlvLinkConnectionBoxInterface` defines a second method:

```
public float getTangentialOffset(IlvGraphModel graphModel, Object node, int nodeSide);
```

This method is used only in the short link mode. For details, see *Using a link connection box interface*. When using the Link Layout in long link mode, just implement the method by returning the value 0.

For experts: special options of the Short LL

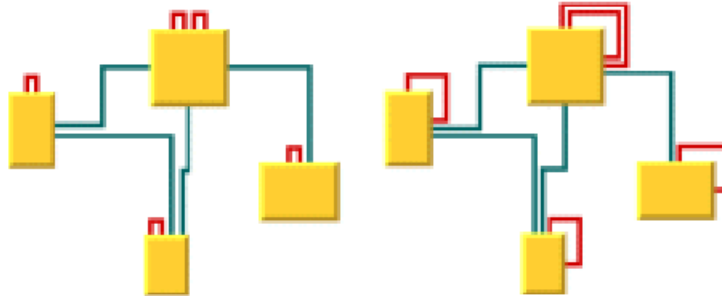
The Link Layout algorithm utilizes the class `IlvShortLinkLayout` as a subalgorithm. `IlvShortLinkLayout` is a subclass of `IlvGraphLayout` and can be used a stand-alone as well. To access the instance of `IlvShortLinkLayout` that is used by the Link Layout algorithm, call:

```
IlvShortLinkLayout getShortLinkLayout();
```

Using this accessor, you can control many special features of the Short Link Layout that are not made available by the `IlvLinkLayout` class because these features are for experts only.

Self-link style

Self-links are links whose origin and destination is the same node. The Short Link Layout provides two optional shapes for self-links.



Two-bend Self-link Style

Three-bend Self-link Style

Self-link Style Options

Example of setting the style of the self-links (Link Layout algorithm)

To set the style of the self-links:

In Java™

Call `layout.getShortLinkLayout().setGlobalSelfLinkStyle(int)`

The valid values for style are:

- ◆ `IlvShortLinkLayout.TWO_BENDS_ORTHOGONAL_STYLE`
- ◆ `IlvShortLinkLayout.THREE_BENDS_ORTHOGONAL_STYLE`

Number of optimization iterations

The link shape optimization is stopped if the time exceeds the allowed time (see *Allowed time (LL)*) or if the number of iterations exceeds the allowed number of iterations.

Example of specifying the number of optimization iterations (Link Layout algorithm)

To set the allowed number of iterations to 3:

In Java

Call:

```
layout.getShortLinkLayout().setAllowedNumberOfIterations(3);
```

Note: You may want to disable the link shape optimization by setting the number of iterations to zero to increase the speed of the layout process.

Evenly spaced pins margin ratio

The margin ratio allows you to customize the way connection points are computed when the connector style (see *Connector style*) is `EVENLY_SPACED_PINS`, and when the `AUTOMATIC_STYLE` places the connection points using the `EVENLY_SPACED_PINS` style. This option has no effect if the connector style `FIXED_OFFSET_PINS` is used.

In the “evenly spaced pins” connector style, the connection points of the links are evenly spaced along the node border, preserving a margin to each extremity of the node border. The size of this margin is controlled by the margin ratio and is computed by multiplying the offset between the links by the ratio.

Example of specifying margin ratio (Link Layout algorithm)

To specify this option

In Java

Call `layout.getShortLinkLayout().setEvenlySpacedPinsMarginRatio(float)`

The input value must be a positive or zero value. The default value is 0.5. *Evenly Spaced Pins Margin Ratio* shows examples of values with their meaning.

Evenly Spaced Pins Margin Ratio

| Ratio value | Meaning |
|---------------------|--|
| 0 | No margin |
| 0.5 (default value) | The margin is equal to half the offset between the links. |
| 1 | The margin is equal to the offset between the links. |
| 2 | The margin is equal to twice the offset between the links. |

Link overlap nodes forbidden

This option allows you to ask the layout algorithm to avoid strictly to reshape links such that they overlap some nodes. If overlaps are not forbidden, the algorithm tries to avoid overlaps anyway, but may create overlaps, for instance for the link to cross other links.

Note: Forbidding overlaps may slow down the layout and may increase the number of bends for those links that would overlap nodes if overlaps were not strictly forbidden.

Example of specifying link overlap nodes forbidden (Link Layout algorithm)

To specify this option:

In Java

Call

```
layout.getShortLinkLayout().setLinkOverlapNodesForbidden(boolean)
```

The default value of this option is `false`.

When overlaps are forbidden, the Short Link Layout algorithm uses the Long Link Layout as an auxiliary algorithm for laying out only the links that would otherwise overlap nodes.

Example of specifying Long Link Layout when overlaps forbidden (Link Layout algorithm)

To retrieve the auxiliary instance of Long Link Layout:

In Java

Call this method on the `IlvShortLinkLayout` instance:

```
IlvLongLinkLayout getAuxiliaryLongLinkLayout()
```

This method allows you to get this auxiliary layout instance and to customize its parameters if needed. Notice that you should neither modify the origin and destination point mode, nor disable the preservation of fixed links. Notice also that an `IlvGraphModel` instance is attached to the `IlvLongLinkLayout` instance only if needed, therefore the method `getAuxiliaryLongLinkLayout().getGraphModel()` may return `null`.

Incremental link reshape mode

In incremental mode, it is possible to customize the rules used by the Short Link Layout to determine which links should keep their current shape as much as possible, as computed by the previous layout execution. The incremental link reshape mode allows you to customize these rules separately for two categories of links. See the methods:

```
IlvShortLinkLayout.getLinkConnectionBoxInterface()
```

and

```
IlvShortLinkLayout.getNodeBoxInterface()
```

- ◆ The “modified links”: the links that have either a different “link connection box” or are connected to nodes which have a different bounding box as during the previous layout execution.
- ◆ The “unmodified links”: the links that have the same “link connection box” and are connected to nodes which have the same bounding box as during the previous layout execution.

The mode can be customized either for both or for only one of these categories of links.

The incremental link reshape mode has no effect if the incremental mode is disabled.

The layout algorithm provides two incremental link reshape modes. You can set the mode globally, in which case all the links have the same mode, or locally on each link, in which case different modes occur in the same drawing.

Global incremental link reshape mode

Example of specifying global incremental link reshape mode (Link Layout algorithm)

To specify the global incremental link reshape mode:

In Java

Use the following methods:

```
layout.getShortLinkLayout().setGlobalIncrementalModifiedLinkReshapeMode
```

```
layout.getShortLinkLayout().setGlobalIncrementalUnmodifiedLinkReshapeMode
```

The valid values for `mode` are:

◆ `IlvShortLinkLayout.FIXED_SHAPE_TYPE_MODE` (the default)

The incremental layout preserves the shape type of the link. This means that both the number of bends and the node sides to which the link is connected are preserved.

◆ `IlvShortLinkLayout.FIXED_NODE_SIDES_MODE`

The incremental layout preserves the node sides to which the links are connected.

◆ `IlvShortLinkLayout.FIXED_CONNECTION_POINTS_MODE`

The incremental layout preserves the connection points of the links.

◆ `IlvShortLinkLayout.FIXED_MODE`

The links are not reshaped at all during incremental layout. Only newly added links are rerouted.

◆ `IlvShortLinkLayout.FREE_MODE`

The incremental layout is allowed to freely reshape the links. This is equivalent to a non-incremental behavior for all the links, hence it is recommended to disable the incremental mode instead of using `FREE_MODE` as global incremental reshape mode.

Of course, the settings that may have been done by “fixing” links (see *Preserve fixed links (LL)*) or by customizing the origin or destination point mode (see *End points mode (LL)*) are still respected.

◆ `IlvShortLinkLayout.MIXED_MODE`

Each link can have a different mode.

Individual incremental link reshape mode

All links have the same incremental link reshape mode unless the global incremental link reshape mode is `IlvShortLinkLayout.MIXED_MODE`. Only when the global mode is set to `MIXED_MODE` can each link have an individual mode.

Example of specifying individual incremental link reshape mode (Link Layout algorithm)

To specify the mode of an individual link:

In Java

Use the following methods on the `IlvShortLinkLayout` instance:

```
void setIncrementalModifiedLinkReshapeMode(Object link, int mode);
```

```
void setIncrementalUnmodifiedLinkReshapeMode(Object link, int mode);
```

```
int getIncrementalModifiedLinkReshapeMode(Object link);
```

```
int getIncrementalUnmodifiedLinkReshapeMode(Object link);
```

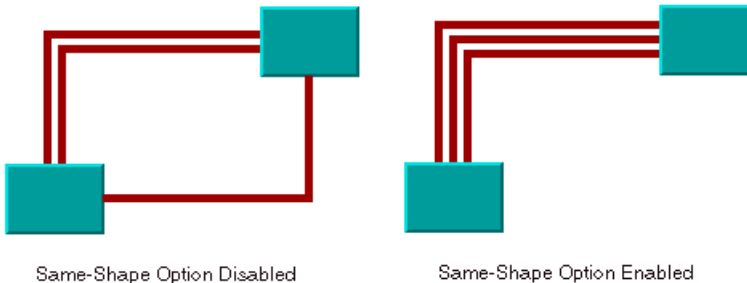
The valid values for `mode` are:

- ◆ `IlvShortLinkLayout.FIXED_SHAPE_TYPE_MODE` (the default)
- ◆ `IlvShortLinkLayout.FIXED_NODE_SIDES_MODE`
- ◆ `IlvShortLinkLayout.FIXED_CONNECTION_POINTS_MODE`
- ◆ `IlvShortLinkLayout.FREE_MODE`
- ◆ `IlvShortLinkLayout.FIXED_MODE`

Same shape for multiple links

You can force the layout algorithm to compute the same shape for all the links having common origin and destination nodes. The links will have parallel shapes.

When this option is disabled, the layout is free to compute different shapes for links connecting the same pair of nodes. Generally, different shapes are chosen to avoid some overlaps.



Self-link style options

Example of specifying same shape for multiple links (Link Layout algorithm)

To enable same shape for multiple links:

In Java

Use the method:

```
layout.getShortLinkLayout().setSameShapeForMultipleLinks(true);
```

The default value is `false`.

Link crossing penalty

The computation of the shape of the links is driven by the objective to minimize a cost function, which is proportional to the number of link-to-link crossings and link-to-node crossings. By default, these two types of crossings have equal weights of 1. You can increase the weight of the link-to-node crossings.

Example of specifying link-to-node crossing penalty (Link Layout algorithm)

To increase the weight of the link-to-node crossings:

In Java

Use the method:

```
layout.getShortLinkLayout().setLinkToNodeCrossingPenalty(5.f);
```

This increases the possibility of obtaining a layout with no link-to-node crossings (or with only a few crossings), with the expense that there may be more link-to-link crossings.

Alternatively, you can increase the weight of the link-to-link crossings.

Example of specifying link-to-link crossing penalty (Link Layout algorithm)

To increase the weight of the link-to-link crossings, for instance, to a value of 3s:

In Java

Use the method:

```
layout.getShortLinkLayout().setLinkToLinkCrossingPenalty(3.f);
```

This increases the possibility of obtaining a layout with no link-to-link crossings (or with only a few crossings), with the expense that there may be more link-to-node crossings.

Bypass distance

If the origin and destination nodes are too close, there may not be enough space for routing the link directly between the end nodes. Therefore, by default, if the end nodes are closer than a threshold distance, the layout chooses link shapes that bypass the interval between close nodes. (See *End nodes and bypass distance*.)



End-nodes distance larger than the
bypass distance

End-nodes distance smaller than the
bypass distance

End nodes and bypass distance

The bypass distance is the minimum distance between the origin and destination nodes for which a link shape going directly from one node to another is allowed. The algorithm tries to avoid link shapes that connect directly the sides of the end nodes that are closer than the bypass value.

Example of specifying the bypass distance (Link Layout algorithm)

To set the bypass distance:

In Java
Call

```
void setBypassDistance(float dist)
```

The default value is a strictly negative value. If the bypass distance is strictly negative, the value of the minimum final segment length (see *Minimum final segment length*) parameter is used as the bypass distance. This allows the automatic adjustment of the bypass distance according to the current value of the minimum final segment length. This behavior is suitable in most cases. However, you can specify a non-negative value to override the default behavior.

Using a link connection box interface

By default, the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes, it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes.

Example of using a link connection box interface to modify the position of the connection points (Link Layout algorithm)

You can modify the position of the connection points of the links by implementing a class that implements the `IlvLinkConnectionBoxInterface`.

In Java

This interface defines the following method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

A second method defined on the interface allows the connection points to be “shifted” tangentially, in a different way for each side of each node:

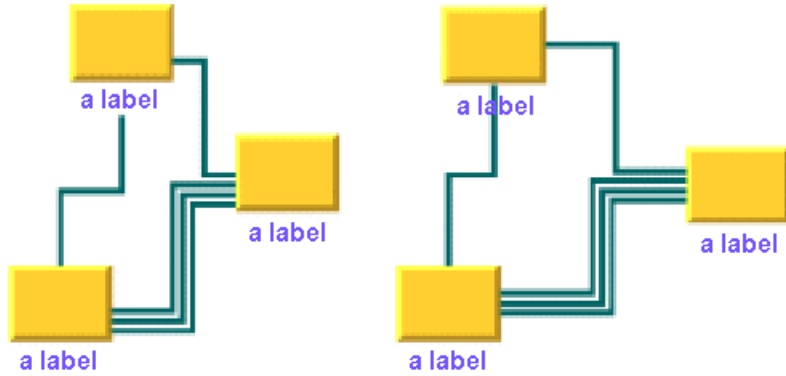
```
public float getTangentialOffset(IlvGraphModel graphModel, Object node, int nodeSide);
```

For instance, to set a link connection box interface that returns a link connection rectangle that is smaller than the bounding box for all nodes of type `MyNodeEditPart` and shifts up the connection points on the left and right side of all the nodes, call:

```
layout.setLinkConnectionBoxInterface(new IlvLinkConnectionBoxInterface() {
    public IlvRect getBox(IlvGraphModel graphModel, Object node) {
        IlvRect rect = graphModel.boundingBox(node);
        if (node instanceof MyNodeEditPart) {
            // for example, the size of the bounding box is reduced by 4 units

            rect.resize(rect.width-4.f, rect.height-4.f);
        }
        return rect;
    }
    public float getTangentialOffset(IlvGraphModel graphModel,
        Object node, int nodeSide) {
        switch (nodeSide) {
            case IlvDirection.Left:
            case IlvDirection.Right:
                return -10; // shift up with 10 for both left and right side
            case IlvDirection.Top:
            case IlvDirection.Bottom:
            default:
                return 0; // no shift for top and bottom side
        }
    }
});
```

Self-link Style Options shows the effects of customizing the connection box. On the left is the result using the default settings: the connection points are distributed on the bounding box of the node (which includes the label) and are symmetric with the middle of each node side (including the label). On the right, is the result after specifying a link connection box interface. On the bottom side of the nodes, the links are now connected to the node (passing over the label), while on the left and right side the nodes are now symmetric to the middle of the node (without the label).



Default Settings

Customized Settings

Customization of the link connection box

For experts: special options of the Long LL

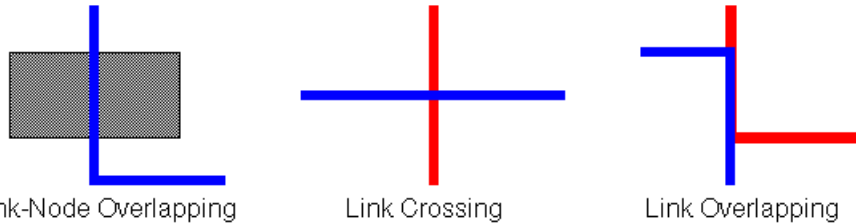
The Link Layout algorithm utilizes the class `IlvLongLinkLayout` as subalgorithm. `IlvLongLinkLayout` is a subclass of `IlvGraphLayout` and can be used a stand-alone as well. To access the instance of `IlvLongLinkLayout` that is used by the Link Layout algorithm, use the method:

```
IlvLongLinkLayout getLongLinkLayout();
```

Using this accessor, you can control many special features of the Long Link Layout that are not made available by the `IlvLinkLayout` class because these features are for experts only.

Specifying additional obstacles

The Long Link Layout algorithm considers nodes to be obstacles that cannot be overlapped and links to be obstacles that can be crossed at an angle of 90 degree (approximately, if the link style is direct), but that cannot be overlapped.



Crossings and Overlappings

Example of specifying additional obstacles (Link Layout algorithm)

If an application requires additional obstacles that are not links or nodes, these can be specified as follows:

In Java™

Call:

```
layout.getLongLinkLayout().addRectObstacle(iolog.views.IlvRect)
layout.getLongLinkLayout().addLineObstacle(iolog.views.IlvRect)
layout.getLongLinkLayout().addLineObstacle
```

Rectangular obstacles behave like nodes: links cannot overlap the rectangles. Line obstacles behave like link segments: other links can cross the line segments, but cannot overlap the segments. These obstacle settings can be removed by the following:

```
layout.getLongLinkLayout().removeAllRectObstacles()
layout.getLongLinkLayout().
removeAllLineObstacles()
```

Penalties for variable end points

If the termination points of the links are not fixed, the algorithm uses a heuristic to calculate the termination points of each link. It examines all free grid points that are close to the border of the start and end node and assigns a penalty to each grid point. If a node-side filter is installed, the penalty depends on whether the node side is allowed or rejected.

A more precise way to affect how the termination points are chosen is the termination point filter. This enables the user to specify the penalty for each grid point.

Example of specifying the termination point filter (Link Layout algorithm)

In Java

A termination point filter is a class that implements the interface `IlvTerminationPointFilter` that defines the following method:

```
public int getPenalty(IlvGraphModel graphModel, Object link,
boolean origin, Object node, IlvPoint point,
int side, int proposedPenalty);
```

To select the `origin` or destination point of the input link, the input point (a grid point on the input side of the node) is examined. The `proposedPenalty` is calculated by the default heuristic of the algorithm. You can return a changed penalty or you can return `java.lang.Integer.MAX_VALUE` to reject the grid point. If the grid point is rejected, it is not chosen as termination point of the link.

The termination point filter can be set as follows:

Call on the `IlvLongLinkLayout` instance: `setTerminationPointFilter`

Manipulating the routing phases

As mentioned in *Long Link Layout algorithm*, the algorithm first treats each link individually and then applies a crossing reduction phase to all links. To find a route for an individual link, the algorithm first checks whether a routing (such as a straight line or with only one bend) is possible. If this kind of routing is not possible, it uses a sophisticated, but more time consuming, grid search algorithm with backtracking to find a route with many bends.

Example of manipulating the routing phases (Link Layout algorithm)

To switch off the phase that finds a straight-line or one-bend routing:

In Java

Call:

```
layout.getLongLinkLayout().setStraightRouteEnabled(boolean)
```

The backtrack search for a route with many bends can be affected in the several ways.

A more convenient way is to specify the maximum time available to search for the route for each link.

Example of specifying backtrack steps (Link Layout algorithm)

You can specify the maximum number of backtrack steps by using the following:

In Java

In Java, call:


```
layout.getLongLinkLayout().setMaxBacktrack(int)
```

The default maximum backtrack number is 30000.

Example of specifying maximal time for route search (Link Layout algorithm)

To specify the maximum time available to search for the route for each link.

In Java

Call:

```
setAllowedTimePerLink(long)
```

The default allowed time per link is 2000 milliseconds (2 seconds).

Finally, you can specify how many steps should be done during the crossing reduction phase.

Example of specifying number of steps in crossing reduction phase (Link Layout algorithm)

To specify how many steps should be done during the crossing reduction phase:

In Java

Call

```
setNumberCrossingReductionIterations(int)
```

Example of disabling crossing reduction (Link Layout algorithm)

You can disable the crossing reduction completely by using the following:

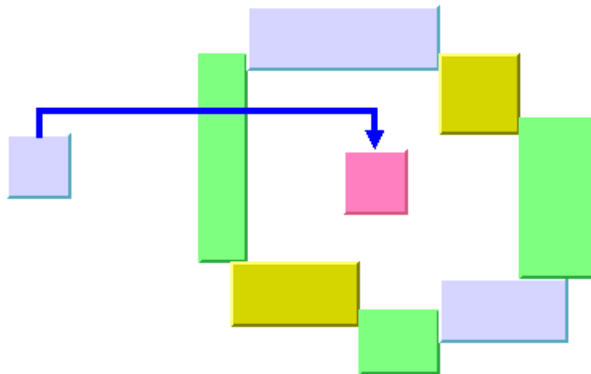
In Java

Call

```
setCrossingReductionEnabled(boolean)
```

Fallback mechanism

The Long Link Layout algorithm may not be able to find a routing for a link, if one of the end nodes is inside an enclave. In *A Node inside an enclave*, the red node is inside an enclave. In this case, the backtrack search algorithm fails to find a routing without overlapping nodes. The backtrack search algorithm may also fail if the situation is so complex that the search exceeds the allowed time per link.



A Node inside an enclave

When the backtrack search algorithm fails to find a routing, a simple fallback mechanism is applied that creates a routing with node overlappings.

Example of disabling fallback mechanism (Link Layout algorithm)

To disable the fallback mechanism:

In Java

```
layout.getLongLinkLayout().setFallbackRouteEnabled(false);
```

If the fallback mechanism is disabled, these links are not routed at all and remain in the same shape as before the layout. In Java code, you can retrieve the links that could not be routed in the usual way without the fallback mechanism.

Example of retrieving links without the fallback mechanism (Link Layout algorithm)

To retrieve the links that, without the fallback mechanism, could not be routed in the usual way :

In Java

```
Enumeration e = layout.getLongLinkLayout().getCalcFallbackLinks();
```

For instance, you can iterate over these links and apply your own specific fallback mechanism instead of the default fallback mechanism of the Long Link Layout algorithm.

Random layout (RL)

Describes the *Random Layout* algorithm (class `IlvRandomLayout` from the package `ilog.views.graphlayout.random`).

In this section

RL sample

Gives some samples of the random layout and explains where it is used.

Features and limitations of the RL

Gives a list of features and limitations.

The RL algorithm

Describes the placement of the nodes and gives samples of the specifications.

Generic features and parameters of the RL

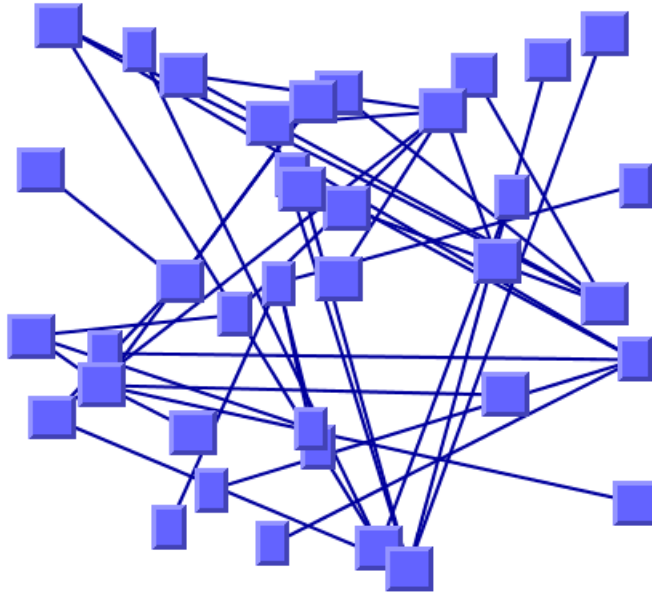
Describes the generic features and parameters of the layout.

Specific parameters of the RL

Describes the parameters that are specific to the `IlvRandomLayout` class:

RL sample

The following figure shows a sample drawing produced with the Random Layout (RL).



Graph drawing produced with the Random Layout

What types of graphs suit the RL?

Any type of graph:

- ◆ connected graphs and disconnected graphs
- ◆ planar graphs and nonplanar graphs

Features and limitations of the RL

Features

Random placement of the nodes of a grapher inside a given region.

Limitations

- ◆ The algorithm computes random coordinates for the upper-left corner of the graphic objects representing the nodes. In some cases, this may not be appropriate.
- ◆ To ensure that the nodes do not overlap the margins of the layout region, the algorithm computes the coordinates randomly inside a region whose width and height are smaller than the width and height of the layout region. The difference is the maximum width and the maximum height of the nodes, respectively. In some cases, this may not be appropriate.

The RL algorithm

The Random Layout (RL) algorithm is not really a layout algorithm. It simply places the nodes at randomly computed positions inside a user-defined region. Nevertheless, the Random Layout algorithm may be useful when a random, initial placement is needed by another layout algorithm or in cases where an aesthetic, readable drawing is not important.

Example of RL

In Java

Below is a code sample using the `IlvRandomLayout` class. This code sample shows how to perform a Random Layout:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.runtime.random.*;
...

IlvRandomLayout layout = new IlvRandomLayout();
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);
try {
    IlvGraphLayoutReport layoutReport = layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
layout.detach();
graphModel.dispose();
```

Generic features and parameters of the RL

The `IlvRandomLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- ◆ *Layout region (RL)*
- ◆ *Percentage of completion calculation (RL)*
- ◆ *Preserve fixed links (RL)*
- ◆ *Preserve fixed nodes (RL)*
- ◆ *Random generator seed value (RL)*
- ◆ *Stop immediately (RL)*

The following sections describe the particular way in which these parameters are used by this subclass.

Layout region (RL)

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. (See *Layout region*.)

Percentage of completion calculation (RL)

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of the layout. (For a detailed description of this features, see *Percentage of completion calculation* and *Graph layout event listeners*.)

Preserve fixed links (RL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links*.)

Preserve fixed nodes (RL)

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve fixed nodes*.)

Random generator seed value (RL)

The Random Layout uses a random number generator to compute the coordinates. You can specify a particular value to be used as a seed value. (See *Random generator seed value*) For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph.

Stop immediately (RL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the RL

Link style (RL)

When the layout algorithm moves the nodes, straight-line links will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links, the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

Example of removing intermediate link points (RL algorithm)

To specify that the layout algorithm to automatically removes all the intermediate points of the links (if any):

In Java™

Use the method:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- ◆ `IlvRandomLayout.NO_RESHAPE_STYLE`
None of the links is reshaped in any manner.
- ◆ `IlvRandomLayout.STRAIGHT_LINE_STYLE`
All the intermediate points of the links (if any) are removed. This is the default value.

Bus layout (BL)

Describes the *Bus Layout* algorithm (class `IlvBusLayout` from the package `ilog.views.graphlayout.bus`).

In this section

BL - sample

Gives a sample of the Bus Layout (BL) and explains where it is used.

Features of the BL

Lists the features of the layout.

The BL algorithm

Describes the Bus Layout algorithm and gives samples of the specification.

Generic features and parameters of the BL

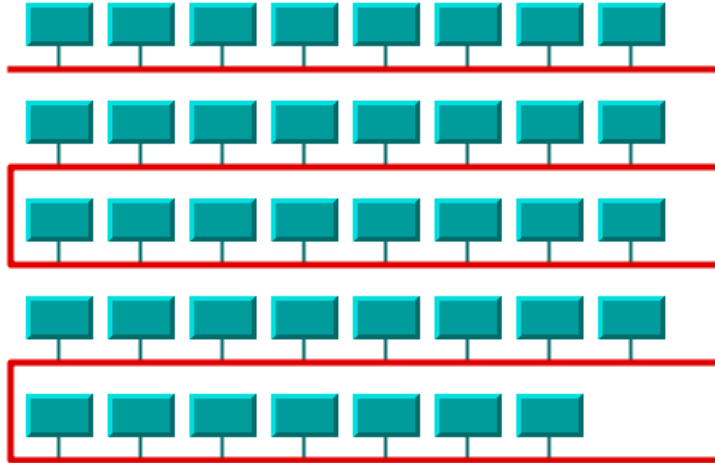
Lists the generic features and parameters of the Bus Layout (BL).

Specific parameters of the BL

Lists the specific parameters of the Bus Layout (BL).

BL - sample

The following figure shows a sample drawing produced with the Bus Layout (BL).



Bus topology produced with the Bus Layout

What types of graphs suit the BL?

- ◆ Bus network topologies (a set of nodes connected to a bus object)

Application domains for the BL

Application domains of the Bus Layout include:

- ◆ Telecom and networking (LAN diagrams)
- ◆ Electrical engineering (circuit block diagrams)
- ◆ Industrial engineering (equipment/resource control charts)

Features of the BL

- ◆ Displays bus topologies.
- ◆ Takes into account the size of the nodes so that no overlapping occurs.
- ◆ Provides several ordering, alignment, and flow direction options.
- ◆ Allows easy customization of the dimensional parameters.

The BL algorithm

Bus topology is well known in network management and telecommunications fields. The Bus Layout class can display these topologies nicely. It represents the “bus” as a “serpent” polyline. The width of the “serpent” is user-defined (via the width of the layout region parameter) and the height is computed so that enough space is available for all the nodes.

BL Sample

Bus Edit Part

An edit part must be dedicated to the bus. It must implement the interface `IlvPolyPointsInterface` and behave as a node. The following code sample shows a possible implementation of the bus:

```
public class BusEditPart extends AbstractGraphicalEditPart implements
    PropertyChangeListener, NodeEditPart, IlvPolyPointsInterface {
    ...

    private BusElement getBus() {
        return (BusElement) getModel();
    }

    protected IFigure createFigure() {
        Polyline bus = new Polyline();
        return bus;
    }

    public boolean allowsPointInsertion() {
        return true;
    }

    public boolean allowsPointMove(int index) {
        return true;
    }

    public boolean allowsPointRemoval() {
        return true;
    }

    public IlvRect boundingBox(IlvTransformer t) {
        Rectangle bounds = getBus().getPoints().getBounds();
        return new IlvRect(bounds.x, bounds.y, bounds.width, bounds.height);
    }

    public IlvPoint getPointAt(int index, IlvTransformer t) {
        Point p = getBus().getPoint(index);
        return new IlvPoint(p.x, p.y);
    }

    public int getPointsCardinal() {
        return getBus().getPoints().size();
    }
}
```

```

}

public void insertPoint(int index, float x, float y, IlvTransformer t) {
    getBus().insertPoint(index, x, y);
}

public void movePoint(int index, float x, float y, IlvTransformer t) {
    getBus().movePoint(index, x, y);
}

public boolean pointsInBBox() {
    return true;
}

public void removePoint(int index, IlvTransformer t) {
    getBus().removePoint(index);
}

public ConnectionAnchor getSourceConnectionAnchor(
    ConnectionEditPart connection) {
    BusAnchor anchor = new BusAnchor(getFigure());
    return anchor;
}

public ConnectionAnchor getTargetConnectionAnchor(
    ConnectionEditPart connection) {
    BusAnchor anchor = new BusAnchor(getFigure());
    return anchor;
}

public ConnectionAnchor getSourceConnectionAnchor(Request request) {
    if (request.getType() == RequestConstants.REQ_RECONNECT_SOURCE) {
        BusAnchor anchor = new BusAnchor(getFigure());
        return anchor;
    }
    return null;
}

public ConnectionAnchor getTargetConnectionAnchor(Request request) {
    if (request.getType() == RequestConstants.REQ_RECONNECT_TARGET) {
        BusAnchor anchor = new BusAnchor(getFigure());
        return anchor;
    }
    return null;
}

protected void refreshVisuals() {
    PointList points = getBus().getPoints();
    if (points != null)
        getFigure().setPoints(points);
}

public Polyline getFigure() {
    return (Polyline) super.getFigure();
}
}

```

```

protected void createEditPolicies() {
    installEditPolicy(EditPolicy.NODE_ROLE, new GraphicalNodeEditPolicy());
}
}

```

The edit part must return dedicated anchors of type `BusAnchor`. The following code sample shows an implementation of the associated model object “BusElement”:

```

public class BusElement {
    ...

    PointList points = new PointList();

    public void setPoints(PointList points) {
        this.points = points;
        // fire event to refresh the edit part
    }

    public PointList getPoints() {
        return points;
    }

    public void insertPoint(int index, float x, float y) {
        PointList points = getPoints();
        PointList newPoints = points.getCopy();
        newPoints.insertPoint(new PrecisionPoint(x, y), index);
        setPoints(newPoints);
    }

    public void movePoint(int index, float x, float y) {
        PointList points = getPoints();
        PointList newPoints = points.getCopy();
        newPoints.setPoint(new PrecisionPoint(x, y), index);
        setPoints(newPoints);
    }

    public void removePoint(int index) {
        PointList points = getPoints();
        PointList newPoints = points.getCopy();
        newPoints.removePoint(index);
        setPoints(newPoints);
    }

    public Point getPoint(int index) {
        return getPoints().getPoint(index);
    }

    public Rectangle getConstraint() {
        return getPoints().getBounds();
    }
}

```



```
}
```

Calling the layout

The following code sample uses the `IlvBusLayout` class. This code sample shows how to perform a Bus Layout:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.runtime.bus.*;
...

IlvBusLayout layout = new IlvBusLayout();
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);
/* Specify the bus node */
layout.setBus(myBusEditPart);
try {
    IlvGraphLayoutReport layoutReport = layout.performLayout();
    int code = layoutReport.getCode();
    System.out.println("Layout completed (" + layoutReport.codeToString(code) +
        ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
layout.detach();
graphModel.dispose();
```

Generic features and parameters of the BL

The `IlvBusLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- ◆ *Allowed time (BL)*
- ◆ *Layout of connected components (BL)*
- ◆ *Layout region (BL)*
- ◆ *Link clipping (BL)*
- ◆ *Preserve fixed links (BL)*
- ◆ *Preserve fixed nodes (BL)*
- ◆ *Stop immediately (BL)*

The following sections describe the particular way in which these parameters are used by this subclass.

Allowed time (BL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of connected components (BL)

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of connected components*.)

Layout region (BL)

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing (See *Layout region*.)

The size of the layout is chosen with respect to the layout region (see *Dimensional Parameters for the Bus Layout Algorithm*). The height of the layout region is not taken into account. The height of the layout will be smaller or larger, depending on the number of nodes, the size of the nodes, and the other specified parameters.

Link clipping (BL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface with the Bus Layout* for details of the link clipping mechanism.

Preserve fixed links (BL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links*.)

Preserve fixed nodes (BL)

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve fixed nodes*.)

Stop immediately (BL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the BL

The following parameters are specific to the `IlvBusLayout` class.

Order parameter (BL)

The order parameter specifies how to arrange the nodes.

Example of specifying node ordering option (BL algorithm)

To specify the ordering option for the nodes:

In Java™

Use the method:

```
void setNodeComparator(Comparator comparator)
```

The valid values for `comparator` are:

- ◆ `IlvBusLayout.DESCEENDING_HEIGHT`
The nodes are ordered in the descending order of their height.
- ◆ `IlvBusLayout.ASCENDING_HEIGHT`
The nodes are ordered in the ascending order of their height.
- ◆ `IlvBusLayout.DESCEENDING_WIDTH`
The nodes are ordered in the descending order of their width.
- ◆ `IlvBusLayout.ASCENDING_WIDTH`
The nodes are ordered in the ascending order of their width.
- ◆ `IlvBusLayout.DESCEENDING_AREA`
The nodes are ordered in the descending order of their area.
- ◆ `IlvBusLayout.ASCENDING_AREA`
The nodes are ordered in the ascending order of their area.
- ◆ `IlvBusLayout.ASCENDING_INDEX`
The nodes are ordered in the ascending order of their index (see `setIndex(java.lang.Object, int)`).
- ◆ `IlvBusLayout.DESCEENDING_INDEX`
The nodes are ordered in the descending order of their index (see `setIndex(java.lang.Object, int)`).
- ◆ `null`
The nodes are ordered in an arbitrary way.
- ◆ Any other implementation of the `Comparator` interface.

The nodes are ordered according to this custom comparator.

The default is `null`.

The ordering of the nodes starts at the upper-left corner of the bus.

Note that in incremental mode (see `setIncrementalMode(boolean)`) or when nodes are fixed (see `setFixed(java.lang.Object, boolean)`), the order is not guaranteed to obey the comparator, because it competes with the other constraints.

More about the `ASCENDING_INDEX` and `DESCENDING_INDEX` options (BL)

These options allow you to specify the order of the nodes according to a user-defined index value specified for each node. If this option is chosen, the algorithm sorts the nodes in ascending order according to their index values.

Example of specifying index options (BL algorithm)

The index is an integer value associated with a node. To specify the index:

In Java

Use the method:

```
void setIndex(Object node, int index)
```

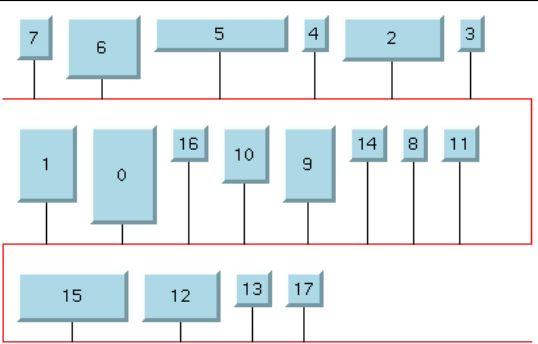
The values of the indices cannot be negative. To obtain the current index of a node, use the method:

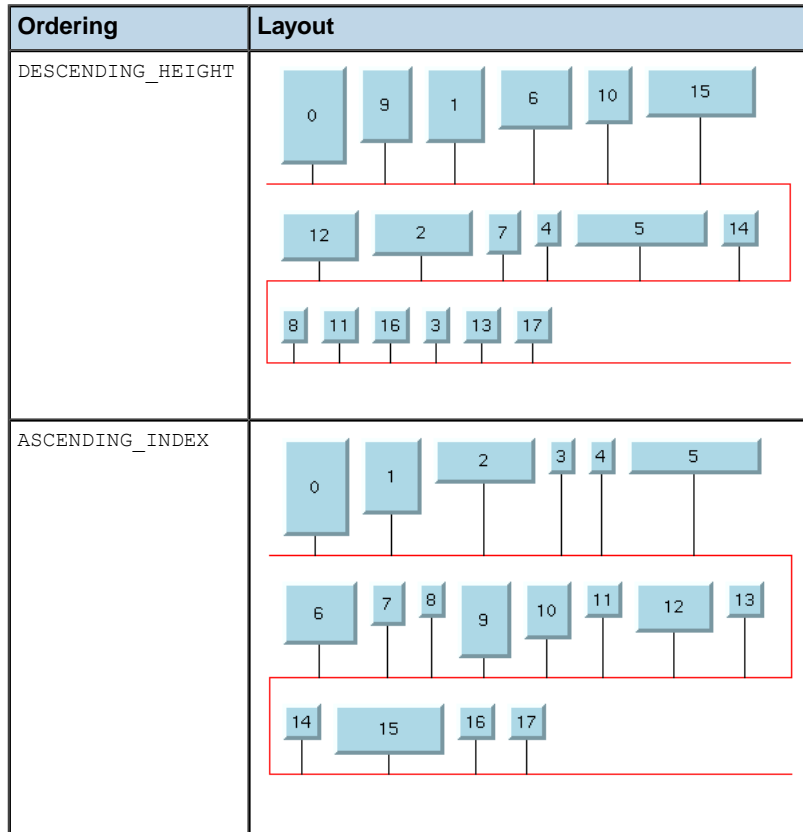
```
int getIndex(Object node)
```

If no index is specified for the node, the value `IlvBusLayout.NO_INDEX` is returned.

The following table shows the ordering options for the Bus Layout algorithm.

Examples of ordering options for the nodes for the Bus Layout algorithm

| Ordering | Layout |
|----------|--|
| No order |  |



Bus node (BL)

To represent bus topologies, the algorithm reshapes a special node, called the “bus node”, and gives it a “serpent” form. This bus node must be an instance of the `IlvPolyPointsInterface` class. Before performing the layout, the grapher must contain this node.

(The number of points in the object you create is not important.) Then, you must specify the node as “bus node” using the method:

```
void setBus(IlvPolyPointsInterface bus)
```

If none is specified, the Bus layout automatically tries to find an appropriate node that can be used as bus object.

The bus object must implement the interface `IlvPolyPointsInterface` and it must allow the insertion and removal of points (see the methods `allowsPointInsertion()` and `allowsPointRemoval()` defined by the interface). The initial number of points is not significant.

Link style (BL)

When the layout algorithm moves the nodes, straight-line links will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links, the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can specify that the layout algorithm automatically removes all the intermediate points of the links (if any).

Example of specifying BL to automatically remove all intermediate points of the link (BL algorithm)

To specify that the layout algorithm automatically removes all the intermediate points of the links (if any):

In Java

Use the method:

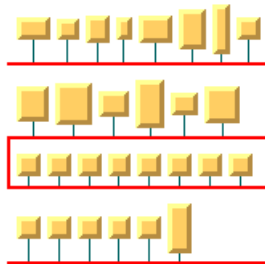
```
void setLinkStyle(int style)
```

The valid values for `style` are:

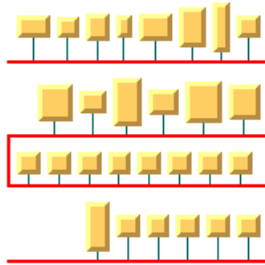
- ◆ `IlvBusLayout.NO_RESHAPE_STYLE`
None of the links are reshaped in any manner.
- ◆ `IlvBusLayout.STRAIGHT_LINE_STYLE`
All the intermediate points of the links (if any) are removed. This is the default value.

Flow direction (BL)

The flow direction options control the horizontal alignment of each row (bus level) with respect to the left and right sides of the layout region. The rows can be either all left-aligned on the left border of the layout region or can alternate between the left and right alignment.



Bus layout with left-to-right flow direction



Bus layout with alternate flow direction

Example of setting the flow direction (BL algorithm)

To set the flow direction:

In Java

Use the method:

```
void setFlowDirection(int direction);
```

The valid values for `direction` are:

- ◆ `IlvBusLayout.LEFT_TO_RIGHT` (the default)
All the rows (bus levels) are left-aligned.
- ◆ `IlvBusLayout.ALTERNATE`
The even rows (bus levels) are left-aligned and the odd rows are right-aligned.

Maximum number of nodes per level (BL)

By default, the layout places as many nodes on each level as possible given the size of the nodes and the dimensional parameters (layout region and margins). If needed, the layout can additionally respect a specified maximum number of nodes per level (see *Bus width adjusting disabled and bounded number of nodes per level* and *Bus width adjusting enabled and bounded number of nodes per level*).

Example of setting the maximum number of nodes per level (BL algorithm)

To set the maximum number of nodes per level:

In Java

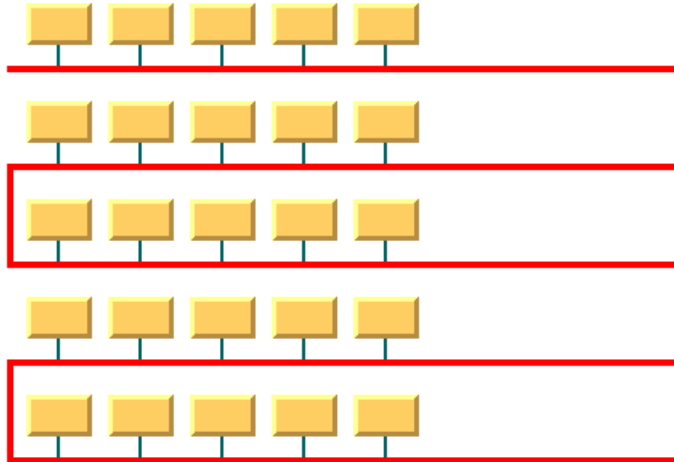
Use the method:

```
void setMaxNumberOfNodesPerLevel(int nNodes);
```

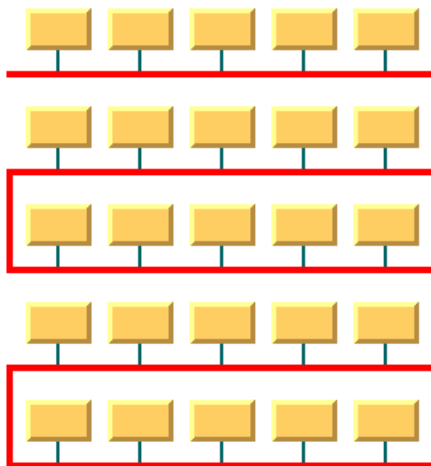
The default value is `Integer.MAX_VALUE`. This means that the number of nodes placed in each level is only bounded by the size of the nodes and the dimensional parameters. The specified value must be at least 1.

Bus width adjusting (BL)

By default, the width of the bus object, that is the difference between the maximum and minimum x-coordinates, depends on the width of the layout region and the other dimensional parameters (see *Dimensional Parameters for the Bus Layout Algorithm*). Optionally, the width of the bus object can be automatically adjusted to the total width of the nodes, plus the offsets and the margins. This option can be particularly useful in conjunction with the customization of the maximum number of nodes per level (see *Maximum number of nodes per level (BL)*).



Bus width adjusting disabled and bounded number of nodes per level



Bus width adjusting enabled and bounded number of nodes per level

Example of enabling/disabling the bus width adjustment (BL algorithm)

To enable or disable bus width adjusting:

In Java

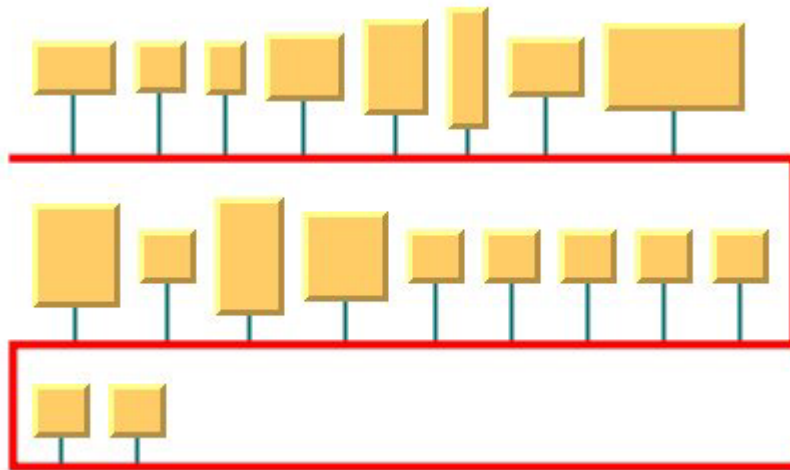
Use the method:

```
void setBusWidthAdjustingEnabled(boolean enable);
```

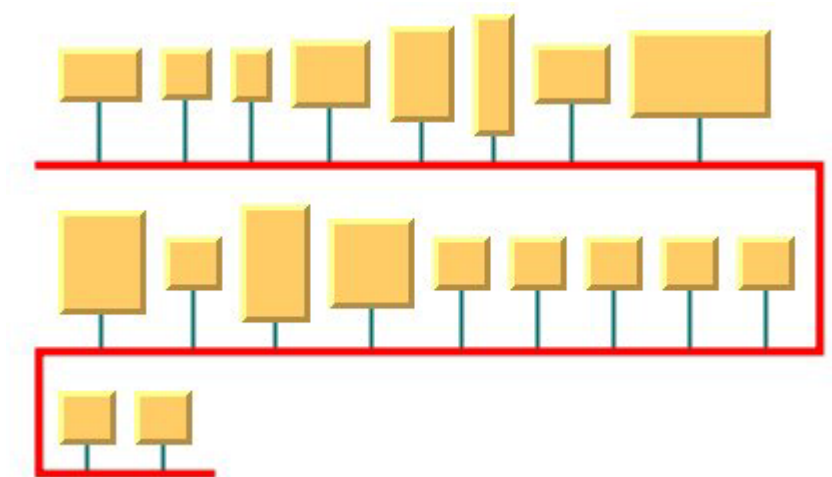
The bus width adjusting is disabled by default.

Bus line extremity adjusting (BL)

If necessary, the bus line can be adjusted to stop where the nodes stop (plus the margins). This can make a difference when there is only one horizontal bus line, or when the flow direction is `ALTERNATE`.



Bus Layout with bus line extremity disabled



Bus Layout with bus line extremity enabled

Example of enabling/disabling the bus line extremity adjustment (BL algorithm)

To enable or disable the adjustment of the bus line extremity:

In Java

Use the method:

```
void setBusLineExtremityAdjustingEnabled (boolean enable);
```

The adjustment of the bus line extremity is disabled by default.

Alignment parameters (BL)

The alignment options control how a node is placed above its row (bus level). The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.

Global alignment parameters

Example of setting global alignment (BL algorithm)

To set the global alignment:

In Java

Use the method:

```
void setGlobalVerticalAlignment (int alignment);
```

The valid values for `alignment` are:

- ◆ `IlvBusLayout.CENTER` (the default)

The node is vertically centered over its row (bus level).

◆ `IlvBusLayout.TOP`

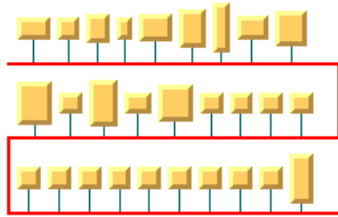
The node is vertically aligned on the top of its row (bus level).

◆ `IlvBusLayout.BOTTOM`

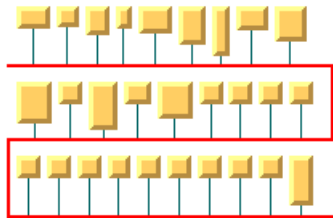
The node is vertically aligned on the bottom of its row (bus level).

◆ `IlvBusLayout.MIXED`

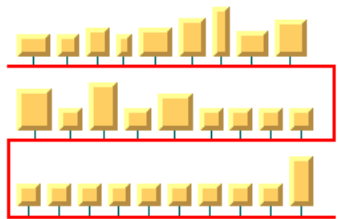
Each node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.



Bus Layout with center vertical alignment



Bus Layout with top vertical alignment



Bus Layout with bottom vertical alignment

Alignment of individual nodes

All nodes have the same alignment unless the global alignment is set to `IlvBusLayout.MIXED`. Only when the global alignment is set to `MIXED` can each node have an individual alignment style.

Example of setting the alignment of an individual node (BL algorithm)

To set the alignment of an individual node:

In Java

Use the methods:

```
void setVerticalAlignment(Object node, int alignment);
```

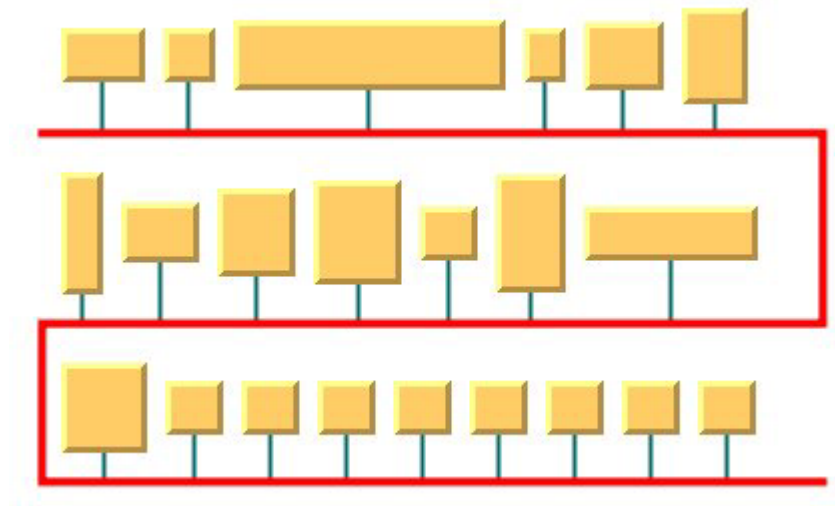
```
int getVerticalAlignment(Object node);
```

The valid values for node alignment are:

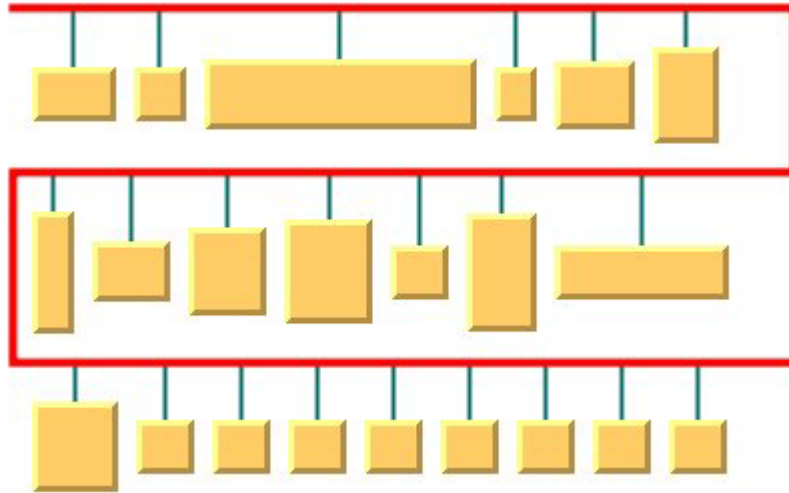
- ◆ `IlvBusLayout.CENTER` (the default)
- ◆ `IlvBusLayout.TOP`
- ◆ `IlvBusLayout.BOTTOM`

Node position (BL)

The nodes can be placed either above or below the corresponding bus line.



Bus Layout with nodes above the bus



Bus Layout with Nodes Below the Bus

Example of setting node position (BL algorithm)

To set the node position:

In Java

Use the method:

```
void setNodePosition(int position);
```

The valid values for node positions are:

- ◆ `IlvBusLayout.NODES_ABOVE_BUS` (the default)

The nodes are placed above the corresponding bus line.

- ◆ `IlvBusLayout.NODES_BELOW_BUS`

The nodes are placed below the corresponding bus line.

Incremental mode (BL)

The Bus Layout algorithm normally places all the nodes from scratch. If the graph incrementally changes because you add, remove, or resize nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode. In incremental mode, the layout tries to place the nodes at the same location or in the same order as in the previous layout whenever it is possible

Example of enabling incremental mode (BL algorithm)

To enable the incremental mode:

In Java

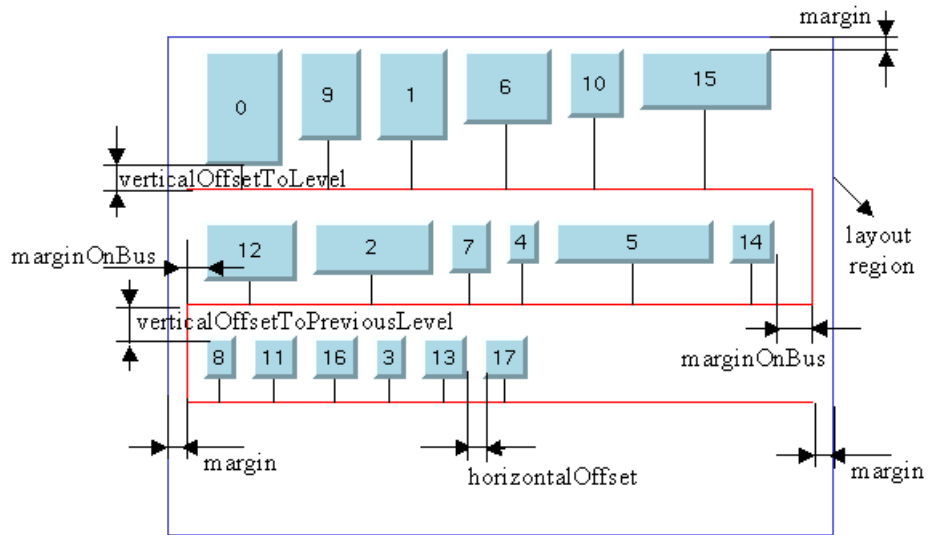
Call:

```
layout.setIncrementalMode(true);
```

Note: To preserve stability, the incremental mode can keep some regions free. Therefore, the total area of the layout can be larger than in nonincremental mode, and, in general, the layout may not look as nice as in nonincremental mode.

Dimensional parameters (BL)

Dimensional Parameters for the Bus Layout Algorithm illustrates the dimensional parameters used in the Bus Layout algorithm. These parameters are explained in the subsequent sections.



Dimensional Parameters for the Bus Layout Algorithm

Horizontal offset (BL)

This parameter represents the horizontal distance between two nodes.

Example of specifying the horizontal offset (BL algorithm)

To specify the horizontal offset:

In Java

Use the method:

```
void setHorizontalOffset(float offset)
```

Vertical offset to level (BL)

This parameter represents the vertical distance between a row of nodes and the next horizontal segment of the bus node.

Example of specifying vertical offset (BL algorithm)

To specify this parameter:

In Java

Use the method:

```
void setVerticalOffsetToLevel(float offset)
```

Vertical offset to previous level (BL)

Example of setting vertical offset to the previous level (BL algorithm)

To set the vertical offset to the previous level:

In Java

This parameter represents the vertical distance between a row of nodes and the previous horizontal segment of the bus node. To specify this parameter, use the method:

```
void setVerticalOffsetToPreviousLevel(float offset)
```

Margin (BL)

This parameter represents the offset distance between the layout region and the bounding rectangle of the layout.

Example of specifying the margin (BL algorithm)

To specify the margin:

In Java

Use the method:

```
void setMargin(float margin)
```

Margin on bus (BL)

On the odd horizontal levels (first, third, fifth, and so on) of the bus, starting from the top, this parameter represents the offset distance between the left side of the first node on the left and the left side of the bus object.

On the even horizontal levels (second, fourth, sixth, and so on) of the bus, starting from the top, this parameter represents the offset distance between the right side of the last node on the right and the right side of the bus object. (See *Dimensional Parameters for the Bus Layout Algorithm* for an illustration of the margin-on-bus parameter.)

Example of specifying the margin on bus (BL algorithm)

To specify this parameter:

In Java

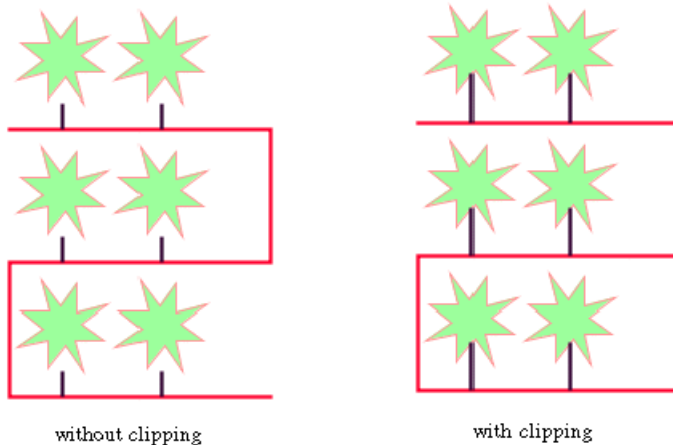
Use the method:

```
void setMarginOnBus(float margin)
```

Using a link clipping interface with the Bus Layout

By default, the Bus Layout does not place the connection points of links at the nodes. The default behavior is to connect to a point at the border of the bounding box of the nodes.

If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



Effect of Link Clipping Interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*.

Example of setting a link clipping interface with the Bus Layout

To set a link clip interface:

In Java

Call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```


Circular layout (CL)

Describes the *Circular Layout* algorithm (class `IlvCircularLayout` from the package `ilog.views.graphlayout.circular`).

In this section

General information on the CL

Gives samples of the Circular Layout (CL) and explains where it is used.

Features and limitations of the CL

Lists the features and limitations of the Circular Layout (CL).

The CL algorithm

Describes the Circular Layout (CL) algorithm and gives samples.

Generic features and parameters of the CL

Describes the generic features and parameters of the layout.

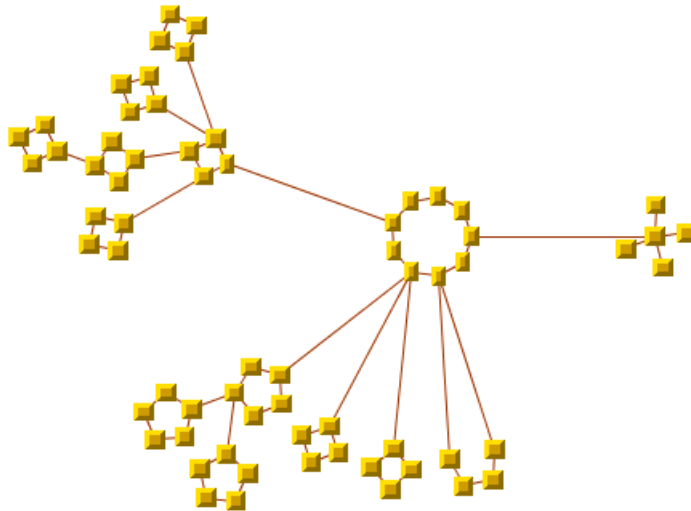
Specific parameters of the CL

Describes the parameters specific to the `IlvCircularLayout` class:

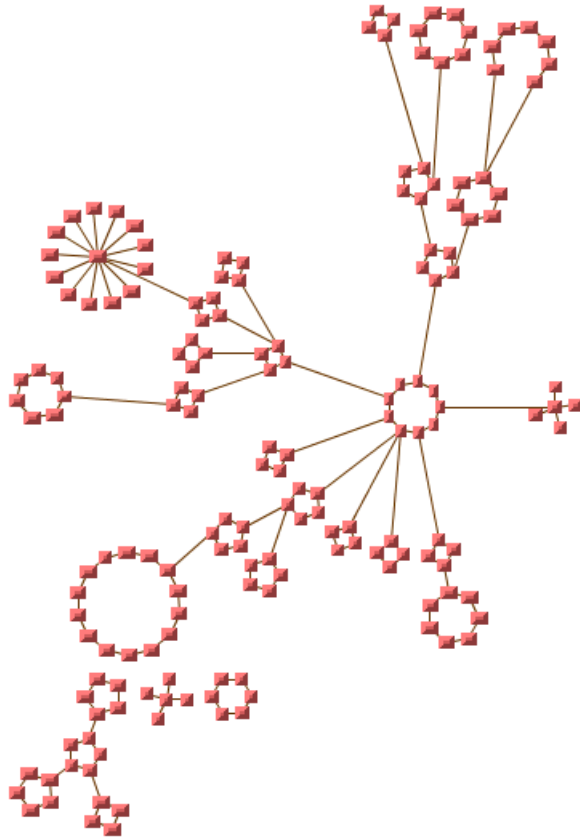
General information on the CL

CL samples

The following figures show sample drawings produced with the Circular Layout.



Ring-and-star topology drawing produced with the Circular Layout



Large ring-and-star topology drawing produced with the Circular Layout

What types of graphs suit the CL?

- ◆ Graphs representing interconnected ring and/or star network topologies

Application domains for the CL

Application domains for the Circular Layout include:

- ◆ Telecom and networking (LAN diagrams)
- ◆ Business processing (organization charts)
- ◆ Database and knowledge engineering (sociology, genealogy)
- ◆ The World Wide Web (Web hyperlink neighborhood)

Features and limitations of the CL

Features

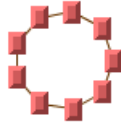
- ◆ Displays network topologies composed of interconnected rings and/or stars.
- ◆ Provides two clustering modes (see *Clustering mode (CL)*). The first mode lays out clusters as circles and places the clusters. This mode is designed for rings/stars that are interconnected in a tree structure, but it can produce acceptable results even if the graph contains cycles. The second mode lays out the clusters as circles of nodes, minimizing the link crossings while keeping the clusters at their initial position.
- ◆ Takes into account the size of the nodes so that no overlapping occurs. (See also *The CL algorithm*).

Limitations

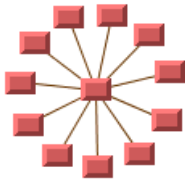
Link crossings cannot always be avoided.

The CL algorithm

Ring and star topologies are similar in several ways. Take a look at *Ring topology* and *Star topology* to get an idea of their similarities.



Ring topology

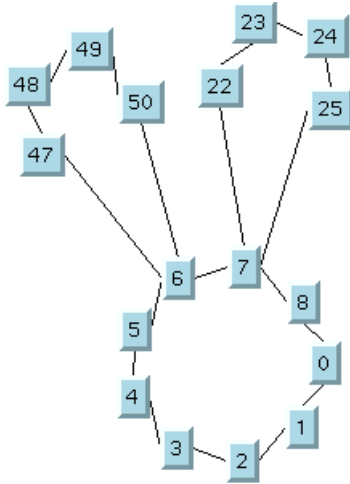


Star topology

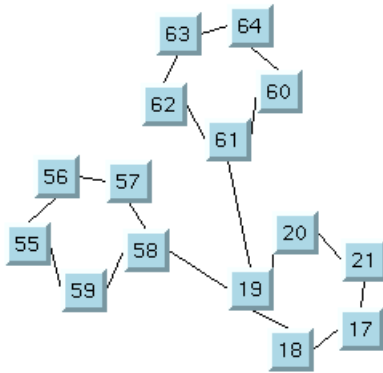
Both topologies are composed of nodes drawn on a circle. For the Circular Layout algorithm, the only difference between the ring and star topologies is that the star has a special node, called the star center, that is drawn at the center of the circle. The user must specify the node that is the star center. (See *Star center (CL)* for information on how to specify the node.)

For each ring or star (generically called a cluster), the Circular Layout algorithm, in one of its modes (see *Clustering mode (CL)*), allows you to specify the order of the nodes on the circle (this is discussed in *Cluster membership and order of the nodes on a cluster (CL)*). Otherwise, an arbitrary order is automatically chosen. In another mode, the order is computed automatically such that the number of link crossings is small.

The network topology can be composed of more than one ring or star. These rings and stars can be partially interconnected; that is, two or more clusters can have a common node as shown in *Rings interconnected by common nodes*. They can also be interconnected by links between nodes of two different clusters as shown in *Rings interconnected by links*.



Rings interconnected by common nodes



Rings interconnected by links

The Circular Layout algorithm lays out the ring/star topologies in a way that preserves the visual identity of each cluster and avoids overlapping nodes and clusters. (See the sample drawings in *CL samples*.)

To understand how the layout is performed in the clustering mode `BY_CLUSTER_IDS`, consider a graph in which each node represents a ring or star cluster of a network topology. Add a link between two nodes each time there is an interconnection between the corresponding clusters. The Circular Layout algorithm is designed for the case where the graph obtained in this manner is a tree (that is, a graph with no cycles). If cycles exist, the layout is performed using a spanning tree of the graph.

Starting from a root cluster (either a ring or a star), the clusters that are connected to the root cluster are drawn on a circle that is concentric to the root cluster. The radius of the circle is computed to avoid overlapping clusters. Next, the algorithm lays out the clusters connected to these last clusters on a larger circle, and so on. Each circle is called a level.

For networks that are not connected (that is, disconnected groups of clusters exist in the graph), more than one spanning tree exists. Each spanning tree is laid out separately and placed near the others. You can see this in the sample drawings in *CL samples*.

In the clustering mode `BY_SUBGRAPHS`, each subgraph (cluster) keeps its initial position. The subgraphs can be placed either by a different layout algorithm or interactively.

CL Example

In Java

Below is a code sample using the `IlvCircularLayout` class. This code sample shows how to perform a Circular Layout:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.circular.*;
...
IlvCircularLayout layout = new IlvCircularLayout();
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);

...

// create identifier for cluster 0
IlvClusterNumber clusterId = new IlvClusterNumber(0);

// specify the cluster identifier for cluster 0
// Assume there are three nodes: node1, node2, node3
// the ordering of the nodes: node1 -> node2 -> node3
layout.setClusterId(node1, clusterId, 0); // index 0
layout.setClusterId(node2, clusterId, 1); // index 1
layout.setClusterId(node3, clusterId, 2); // index 2

// create identifier for cluster 1
clusterId = new IlvClusterNumber(1);

// specify the cluster identifier for cluster 1
// Assume there are three nodes: node4, node5, node6
// the ordering of the nodes: node4 -> node5 -> node6
layout.setClusterId(node4, clusterId, 1); // index 1
layout.setClusterId(node5, clusterId, 2); // index 2
layout.setClusterId(node6, clusterId, 0); // index 0

try {
    IlvGraphLayoutReport layoutReport = layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
```

```
}  
layout.detach();  
graphModel.dispose();
```

Generic features and parameters of the CL

The `IlvCircularLayout` class supports the following parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- ◆ *Layout of connected components (CL)*
- ◆ *Layout region (CL)*
- ◆ *Link clipping (CL)*
- ◆ *Link connection box (CL)*
- ◆ *Preserve fixed links (CL)*
- ◆ *Preserve fixed nodes (CL)*
- ◆ *Stop immediately (CL)*

The following comments describe the particular way in which these parameters are used by this subclass.

Layout of connected components (CL)

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of connected components*).

Layout region (CL)

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

It is not possible to allow the user to control the size of the layout by specifying a bounding box for the drawing. The layout algorithm chooses the size to have enough space to avoid overlapping nodes and clusters.

The layout region setting (either your own or the default setting) is used only to choose the position of the center of the drawing. This means that only the center of the layout region is taken into consideration. (See *Layout region*.)

Link clipping (CL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface with the Circular Layout* for details of the link clipping mechanism.

Link connection box (CL)

The layout algorithm can use a link connection box interface (see *Link connection box*) in combination with the link clip interface. If no link clip interface is used, the link connection

box interface has no effect. For details see *Using a link clipping interface with the Circular Layout*.

Preserve fixed links (CL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links*.)

Preserve fixed nodes (CL)

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve fixed nodes*.)

Stop immediately (CL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the CL

Clustering mode (CL)

The Circular Layout algorithm has two clustering modes.

Example of selecting a clustering mode (CL algorithm)

To select a clustering mode:

In Java™

Use the method:

```
void setClusteringMode(int mode);
```

The valid values for `mode` are:

- ◆ `IlvCircularLayout.BY_CLUSTER_IDS` (the default): Cluster identifiers need to be explicitly provided for each node (see *Cluster membership and order of the nodes on a cluster (CL)*). A tree-like algorithm places the clusters.
- ◆ `IlvCircularLayout.BY_SUBGRAPHS`: The algorithm handles a nested graph, including intergraph links. It arranges the nodes of each subgraph on a circle, so that the number of link crossings is small. It respects the intergraph links and rotates the cluster so that the number of link crossings is small. It assumes that all nodes are nearly square and that all nodes are in subgraphs, but the subgraph nesting is only 1. Nodes that are inside subgraphs of subgraphs are not handled. Note that in this mode each subgraph keeps its initial position. The subgraphs can be placed either by a different layout algorithm or interactively.

Cluster membership and order of the nodes on a cluster (CL)

This section applies only if the clustering mode is set to `BY_CLUSTER_IDS`.

Before performing the layout, you must specify to which cluster each node of the graph belongs.

Example of specifying node cluster (CL algorithm)

To specify to which cluster each node of the graph belongs:

In Java

To specify the cluster membership, use a cluster identifier; that is, an instance of a subclass of the class `IlvClusterId` (which is abstract). Two subclasses are provided:

- ◆ `IlvClusterNumber`, which uses integer numbers as cluster identifiers.
- ◆ `IlvClusterName`, which uses string names as cluster identifiers.

You can combine these two types of identifiers as any other subclass of `IlvClusterId`. For example, you can write:

```
// create identifier for first cluster (integer)
IlvClusterNumber clusterId1 = new IlvClusterNumber(1);
```

```
// create identifier for second cluster (string)
IlvClusterNumber clusterId2 = new IlvClusterName("R&D network");
```

Then, if node1 to node3 belong to the first cluster, you can write:

```
layout.setClusterId(node1, clusterId1);
layout.setClusterId(node2, clusterId1);
layout.setClusterId(node3, clusterId1);
```

Assume layout is an instance of `IlvCircularLayout`.

If you want the nodes to be drawn in a special order (for example, node1 -> node2 -> node3), you should also specify an index (an integer value) for each node:

```
layout.setClusterId(node1, clusterId1, 0);
layout.setClusterId(node2, clusterId1, 1);
layout.setClusterId(node3, clusterId1, 2);
```

Two methods allow you to specify the cluster to which a node belongs:

```
void setClusterId(Object node, IlvClusterId clusterId)
```

```
void addClusterId(Object node, IlvClusterId clusterId)
```

If you call the first method, the node belongs only to the cluster whose identifier is `clusterId`. The second method allows you to specify that a node belongs to more than one cluster.

These methods have another version with an additional argument, an integer value representing the index:

```
void setClusterId(Object node, IlvClusterId clusterId, int index)
```

```
void addClusterId(Object node, IlvClusterId clusterId, int index)
```

This value is used to order the nodes on the cluster. If you specify these indices, the algorithm sorts the nodes in ascending order according to the index values.

Note that the values of the index cannot be negative. They do not need to be continuous; only the order of the values is important.

To obtain the current index of a node on a given cluster, use the method:

```
int getIndex(Object node, IlvClusterId clusterId)
```

If no index is specified for the node, the method returns the value `IlvCircularLayout.NO_INDEX`. It is a negative value.

To obtain an enumeration of the cluster identifiers for the clusters to which the node belongs, use the method:

```
Enumeration getClusterIds(Object node)
```

The elements of the enumeration are instances of a subclass of `IlvClusterId`.

To efficiently obtain the number of clusters to which a node belongs, use the method:

```
int getClusterIdsCount(Object node)
```

To remove a node from a cluster with a given identifier, use the method:

```
void removeClusterId(Object node, IlvClusterId clusterId)
```

To remove a node from all the clusters to which it belongs, use the method:

```
void removeAllClusterIds(Object node)
```

Star center (CL)

Example of specifying star center (CL algorithm)

To specify whether a node is the center of a star:

In Java

Use the method:

```
void setStarCenter(Object node, boolean starCenter)
```

To know whether a node is the center of a star, use the method:

```
boolean isStarCenter(Object node)
```

By default, a node is not the center of a star.

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Root clusters (CL)

The algorithm arranges the clusters of each connected component of the graph of clusters around a “root cluster”. By default, the algorithm can choose this cluster. Optionally, you can specify one or more root clusters (one for each connected component).

Example of specifying root clusters (CL algorithm)

To specify one or more root clusters (one for each connected component):

In Java

Use the methods:

```
void setRootClusterId(IlvClusterId clusterId)
```

To obtain an enumeration of the identifiers of the clusters that have been specified as root clusters, use the method:

```
Enumeration getRootClusterIds()
```

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Area minimization (CL)

For very large graphs, the radius of the concentric circles on which the clusters are placed can become very large. Therefore, the Circular Layout provides an optional mode that reduces the total area of the layout. To reduce the total area, the clusters are distributed more equally on the circle.

Example of specifying area minimization mode (CL algorithm)

To enable or disable the area minimization mode:

In Java

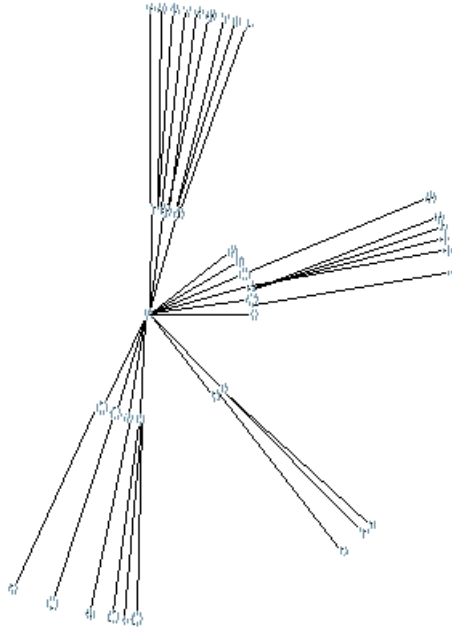
Use the method:

```
void setAreaMinimizationEnabled(boolean option)
```

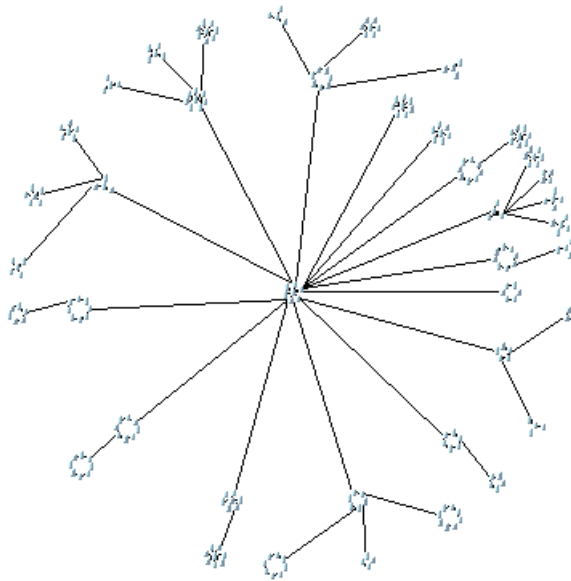
The default value is `false` (area minimization is disabled).

Deciding whether to enable the area minimization mode essentially depends on the size of the network. We recommend the area minimization mode for very large networks.

To get an idea of the difference between these modes, compare the following layouts of the same network:



Area minimization disabled (default)

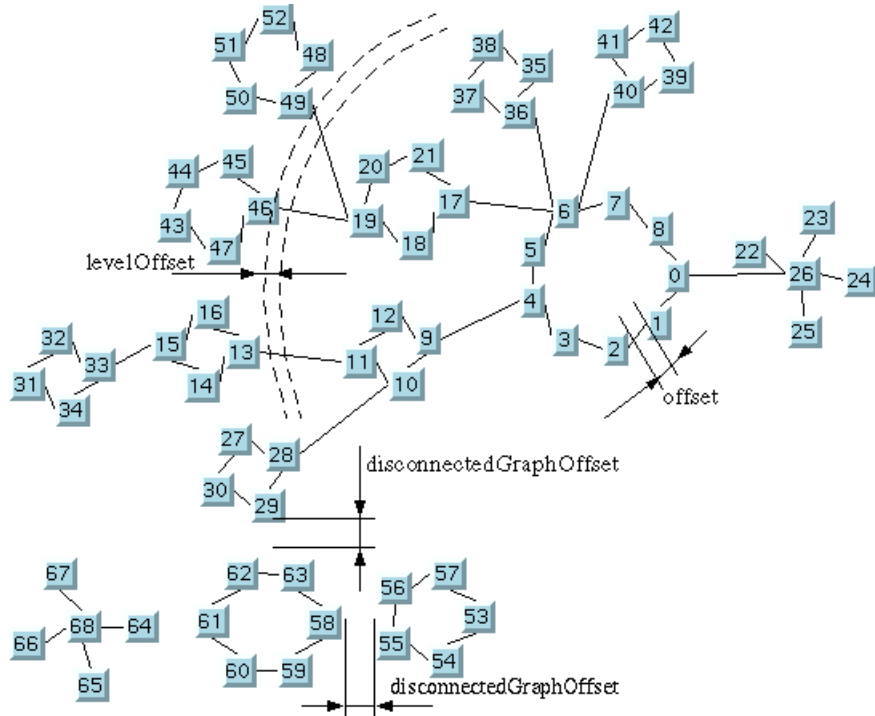


Area minimization enabled

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Dimensional parameters (CL)

Dimensional Parameters for the Circular Layout Algorithm illustrates the dimensional parameters used in the Circular Layout algorithm. These parameters are explained in the sections that follow.



Dimensional Parameters for the Circular Layout Algorithm

Offset (CL)

The layout algorithm tries to preserve a minimum distance between nodes (see *Dimensional Parameters for the Circular Layout Algorithm*).

Example of specifying the offset (CL algorithm)

To specify the offset:

In Java

Use the method:

```
void setOffset(float offset)
```

Level offset (CL)

If the clustering mode is `BY_SUBGRAPHS`, the level offset parameter controls the minimal offset between nodes that belong to the same cluster.

The following applies if the clustering mode is `BY_CLUSTER_IDS`.

As explained in *The CL algorithm*, interconnected rings and/or clusters are drawn on concentric circles around a root cluster. The radius of each concentric circle is computed to avoid overlapping clusters. In some cases, you may want to increase this radius to obtain a clearer drawing of the network. To meet this purpose, the radius is systematically increased with a “level offset” value (see *Dimensional Parameters for the Circular Layout Algorithm*).

Example of specifying the level offset (CL algorithm)

To specify the level offset:

In Java

Use the method:

```
void setLevelOffset(float offset)
```

The default value is zero.

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Disconnected graph offset (CL)

As explained in *The CL algorithm*, each connected component of the network is laid out separately and the drawing of each component is placed near the others (see *Dimensional Parameters for the Circular Layout Algorithm*).

Example of specifying the offset between each connected component (CL algorithm)

To specify the offset between each connected component:

In Java

Use the method:

```
void setDisconnectedGraphOffset(float offset)
```

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Get the contents, the position, and the size of the clusters (CL)

At times, you might need to know the position and the size of the circle on which the nodes for each cluster are drawn. This may be the case if you want to perform some reshaping operations on the links. To do this, you can obtain a vector containing all the cluster identifiers after the layout is performed.

Example of obtain a vector containing all the cluster identifiers (CL algorithm)

To obtain a vector containing all the cluster identifiers after the layout is performed:

In Java

Use the method:

```
Vector getClusterIds()
```

The vector contains instances of a subclass of `IlvClusterId`. By browsing the elements of this `Vector`, you can get the necessary information for each cluster:

```
float getClusterRadius(int clusterIndex)
```

```
IlvPoint getClusterCenter(int clusterIndex)
```

```
Vector getClusterNodes(int clusterIndex)
```

The `getClusterNodes` method returns the nodes that make up the cluster. The argument `clusterIndex` represents the position of the cluster in the `Vector` returned by the method `getClusterIds()`.

Do not use these methods if the clustering mode is `BY_SUBGRAPHS`.

Link style (CL)

When the layout algorithm moves the nodes, straight-line links will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links, the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

Example of specifying automatic removal of all intermediate points of the links (CL algorithm)

To specify that the layout algorithm automatically removes all the intermediate points of the links (if any):

In Java

Use the method:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

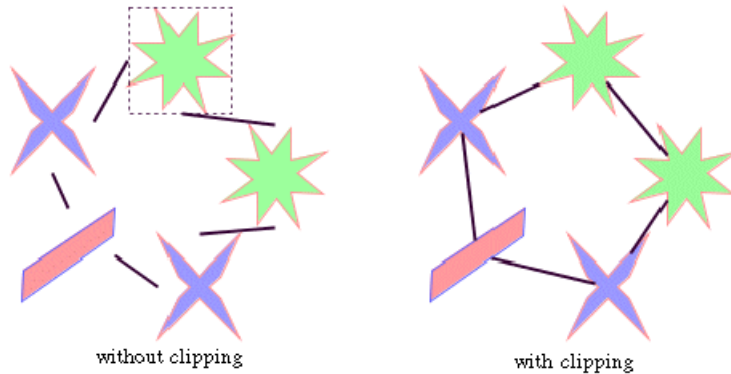
- ◆ `IlvCircularLayout.NO_RESHAPE_STYLE`
None of the links is reshaped in any manner.
- ◆ `IlvCircularLayout.STRAIGHT_LINE_STYLE`
All the intermediate points of the links (if any) are removed. This is the default value.

Using a link clipping interface with the Circular Layout

By default, the Circular Layout does not place the connection points of links. The default behavior is to connect to a point at the border of the bounding box of the nodes.

If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape.

The following figure shows an example of link clipping.



Effect of link clipping interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*.

Example of setting a link clip interface (CL algorithm)

To set a link clip interface:

In Java

Use the method:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

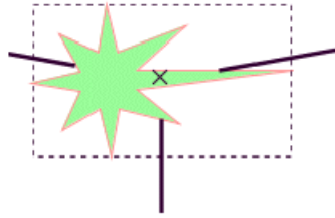
Link connection box (CL)

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*. To set a link connection box interface in Java, call:

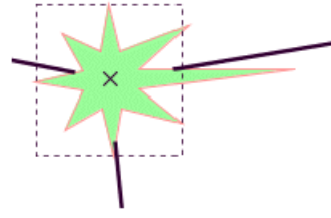
```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

The following figure shows an example of the combined effect.



Clipping at the node bounding box



Clipping at a specified connection box

Combined effect of link clipping interface and link connection box

If the links are clipped at the green irregular star node (previous figure, left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (previous figure, right). Alternatively, the problem could be corrected by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Grid layout (GL)

Describes the *Grid Layout* algorithm (class `IlvGridLayout` from the package `ilog.views.graphlayout.grid`).

In this section

General information on the GL

Gives samples of the Grid Layout (GL) and explains where it is used.

Features of the GL

Lists the features of the Grid Layout (GL).

The GL algorithm

Describes the algorithm for the Grid Layout (GL) and gives samples of the specification.

Generic features and parameters of the GL

Describes the generic features and parameters of the Grid Layout (GL).

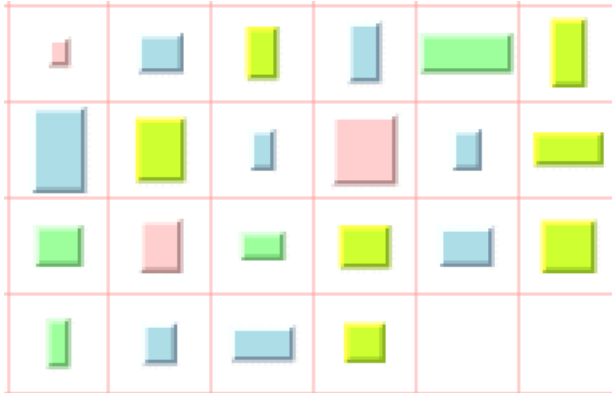
Specific parameters of the GL

Describes the parameters specific to the `IlvGridLayout` class.

General information on the GL

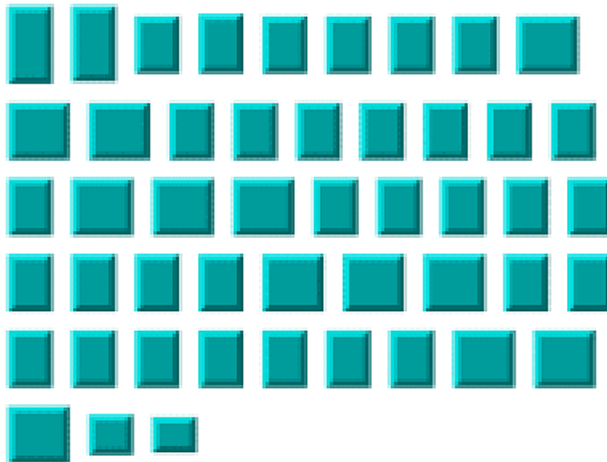
GL sample

The following sample drawings are produced with the Grid Layout (GL).



TILE_TO_GRID_FIXED_WIDTH mode with CENTER horizontal and vertical alignment

In *TILE_TO_GRID_FIXED_WIDTH mode with CENTER horizontal and vertical alignment*, the red lines are drawn to help identify the grid cells; they are not drawn by the layout algorithm.



TILE_TO_ROWS mode with CENTER vertical alignment.

What types of graphs suit the GL?

Any graph. However, the links are never taken into consideration. This algorithm is designed for placing nodes independently of their links, if they have any.

Application domains for the GL

Any domain where a collection of isolated nodes needs to be laid out.

Features of the GL

- ◆ Arranges a collection of isolated nodes or connected components.
- ◆ Takes into account the size of the nodes so that no overlapping occurs.
- ◆ Provides several alignment options and dimensional parameters.
- ◆ Provides full support for fixed nodes (overlapping of nonfixed nodes with fixed nodes is avoided).
- ◆ Provides an incremental mode which helps the retention of a mental map on incremental changes made to a collection of nodes.

The GL algorithm

The Grid Layout (GL) has two main modes: *grid* and *row/column*.

- ◆ In grid mode, the layout arranges the nodes of a graph in the cells of a grid (matrix). If a node is too large to fit in one grid cell (with margins), it occupies multiple cells. The size of the grid cells and the margins are parameters of the algorithm.
- ◆ In row/column mode, the layout arranges the nodes of a graph either by rows or by columns (according to the specified option). The width of the rows is controlled by the width of the layout region parameter. The height of the columns is controlled by the height of the layout region parameter. The horizontal and vertical margins between the nodes are parameters of the algorithm.

GL Example

In Java

Below is a code sample using the `IlvGridLayout` class. This code sample shows how to perform a Grid Layout:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.grid.*;
...

IlvGridLayout layout = new IlvGridLayout();
GraphModel graphModel = new GraphModel(myGrapherEditPart);
layout.attach(graphModel);

try {
    IlvGraphLayoutReport layoutReport = layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
layout.detach();
graphModel.dispose();
```

Generic features and parameters of the GL

The `IlvGridLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- ◆ *Allowed time (GL)*
- ◆ *Layout region (BL)*
- ◆ *Preserve fixed nodes (BL)*
- ◆ *Stop immediately (BL)*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed time (GL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout region (GL)

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. (See *Layout region*.)

The layout region is considered differently depending on the layout mode. For details, see *Layout modes (GL)*.

Preserve fixed nodes (GL)

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve fixed nodes*.) Moreover, nonfixed nodes are placed in such a manner that overlaps with fixed nodes are avoided.

Stop immediately (GL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the GL

Order parameter (GL)

The order parameter specifies how to arrange the nodes.

Example of specifying node placement iterations and allowed time (GL algorithm)

To specify the ordering option for the nodes:

In Java™

Use the method:

```
void setNodeComparator(Comparator comparator)
```

The valid values for `comparator` are:

◆ `AUTOMATIC_ORDERING`

The algorithm is free to choose the order in such a way that it tries to reduce the total area occupied by the layout.

◆ `NO_ORDERING`

No ordering is performed.

◆ `DESCENDING_HEIGHT`

The nodes are ordered in the descending order of their height.

◆ `ASCENDING_HEIGHT`

The nodes are ordered in the ascending order of their height.

◆ `DESCENDING_WIDTH`

The nodes are ordered in the descending order of their width.

◆ `ASCENDING_WIDTH`

The nodes are ordered in the ascending order of their width.

◆ `DESCENDING_AREA`

The nodes are ordered in the descending order of their area.

◆ `ASCENDING_AREA`

The nodes are ordered in the ascending order of their area.

◆ `ASCENDING_INDEX`

The nodes are ordered in the ascending order of their index (see `setIndex(java.lang.Object, int)`).

◆ `DESCENDING_INDEX`

The nodes are ordered in the descending order of their index (see `setIndex(java.lang.Object, int)`).

◆ `null`

The nodes are ordered in an arbitrary way.

◆ Any other implementation of the `java.util.Comparator` interface.

The nodes are ordered according to this custom comparator.

The default is `AUTOMATIC_ORDERING`.

Note that in incremental mode (see `setIncrementalMode(boolean)`) and with fixed nodes (see `setFixed(java.lang.Object, boolean)`), the order of the nodes is not completely preserved.

Note also that, if the layout mode is `TILE_TO_GRID_FIXED_WIDTH` or `TILE_TO_GRID_FIXED_HEIGHT`, the order options are applied only for nodes whose size (including margins) is smaller than the grid cell size (see `setHorizontalGridOffset(float)` and `setVerticalGridOffset(float)`).

Layout modes (GL)

The Grid Layout algorithm has four layout modes.

Example of selecting a layout mode (GL algorithm)

To select a layout mode:

In Java

Use the method:

```
void setLayoutMode(int mode);
```

The valid values for `mode` are:

◆ `IlvGridLayout.TILE_TO_GRID_FIXED_WIDTH` (the default).

The nodes are placed in the cells of a grid (matrix) that has a fixed maximum number of columns. This number is equal to the width of the layout region parameter divided by the horizontal grid offset.

◆ `IlvGridLayout.TILE_TO_GRID_FIXED_HEIGHT`

The nodes are placed in the cells of a grid (matrix) that has a fixed maximum number of rows. This number is equal to the height of the layout region parameter divided by the vertical grid offset.

◆ `IlvGridLayout.TILE_TO_ROWS`

The nodes are placed in rows. The maximum width of the rows is equal to the width of the layout region parameter. The height of the row is the maximum height of the nodes contained in the row (plus margins).

◆ `IlvGridLayout.TILE_TO_COLUMNS`

The nodes are placed in columns. The maximum height of the columns is equal to the height of the layout region parameter. The width of the column is the maximum width of the nodes contained in the column (plus margins).

Alignment parameters (GL)

Global alignment parameters

The alignment options control how a node is placed over its grid cell or over its row or column (depending on the layout mode). The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.

Example of setting global alignment (GL algorithm)

To set the global alignment:

In Java

Use the following methods:

```
void setGlobalHorizontalAlignment(int alignment);
```

```
void setGlobalVerticalAlignment(int alignment);
```

The valid values for the alignment parameter are:

◆ `IlvGridLayout.CENTER` (the default)

The node is horizontally and/or vertically centered over its grid cell or row or column.

◆ `IlvGridLayout.TOP`

The node is vertically aligned on the top of its cell(s) or row. Not used if the layout mode is `TILE_TO_COLUMNS`.

◆ `IlvGridLayout.BOTTOM`

The node is vertically aligned on the bottom of its grid cell(s) or row. Not used if the layout mode is `TILE_TO_COLUMNS`.

◆ `IlvGridLayout.LEFT`

The node is horizontally aligned on the left of its grid cell(s) or column. Not used if the layout mode is `TILE_TO_ROWS`.

◆ `IlvGridLayout.RIGHT`

The node is horizontally aligned on the right of its grid cell(s) or column. Not used if the layout mode is `TILE_TO_ROWS`.

◆ `IlvGridLayout.MIXED`

Each node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

Alignment of individual nodes

All nodes have the same alignment unless the global alignment is set to `IlvGridLayout.MIXED`. Only when the global alignment is set to mixed can each node have an individual alignment style.

Example of setting alignment of individual nodes (GL algorithm)

To set and retrieve the alignment of an individual node:

In Java

In Java

Use the following methods:

```
void setHorizontalAlignment(Object node, int alignment);
```

```
void setVerticalAlignment(Object node, int alignment);
```

```
int getHorizontalAlignment(Object node);
```

```
int getVerticalAlignment(Object node);
```

The valid values for the alignment parameter are:

- ◆ `IlvGridLayout.CENTER` (the default)
- ◆ `IlvGridLayout.TOP`
- ◆ `IlvGridLayout.BOTTOM`
- ◆ `IlvGridLayout.LEFT`
- ◆ `IlvGridLayout.RIGHT`

Maximum number of nodes per row or column (GL)

By default, in `IlvGridLayout.TILE_TO_ROWS` or `IlvGridLayout.TILE_TO_COLUMNS` mode, the layout places as many nodes on each row or column as possible given the size of the nodes and the dimensional parameters (layout region and margins). If needed, the layout can additionally respect a specified maximum number of nodes per row or column.

Example of specifying the maximum number of nodes per row or column (GL algorithm)

To set the maximum number of nodes per row or column:

In Java

Use the method:

```
void setMaxNumberOfNodesPerRowOrColumn(int nNodes);
```

The default value is `Integer.MAX_VALUE`, that is, the number of nodes placed in each row or column is bounded only by the size of the nodes and the dimensional parameters. The

specified value must be at least 1. The parameter has no effect if the layout mode is `IlvGridLayout.TILE_TO_GRID_FIXED_WIDTH` or `IlvGridLayout.TILE_TO_GRID_FIXED_HEIGHT`.

Incremental mode (GL)

The Grid Layout algorithm normally places all the nodes from scratch. If the graph incrementally changes because you add, remove, or resize nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode. In incremental mode, the layout tries to place the nodes at the same location or in the same order as in the previous layout whenever it is possible.

Example of enabling the incremental mode (GL algorithm)

To enable the incremental mode:

In Java

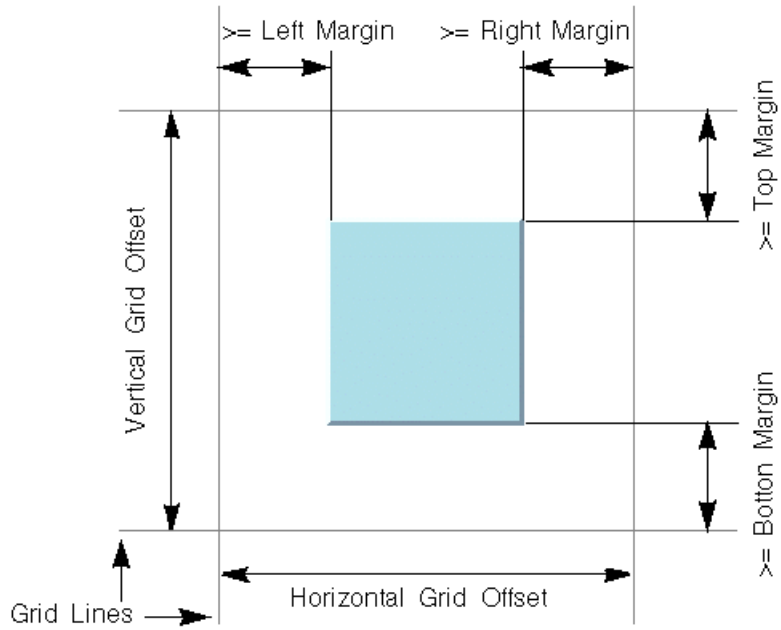
Call the method `setIncrementalMode(boolean)` as follows:

```
layout.setIncrementalMode(true);
```

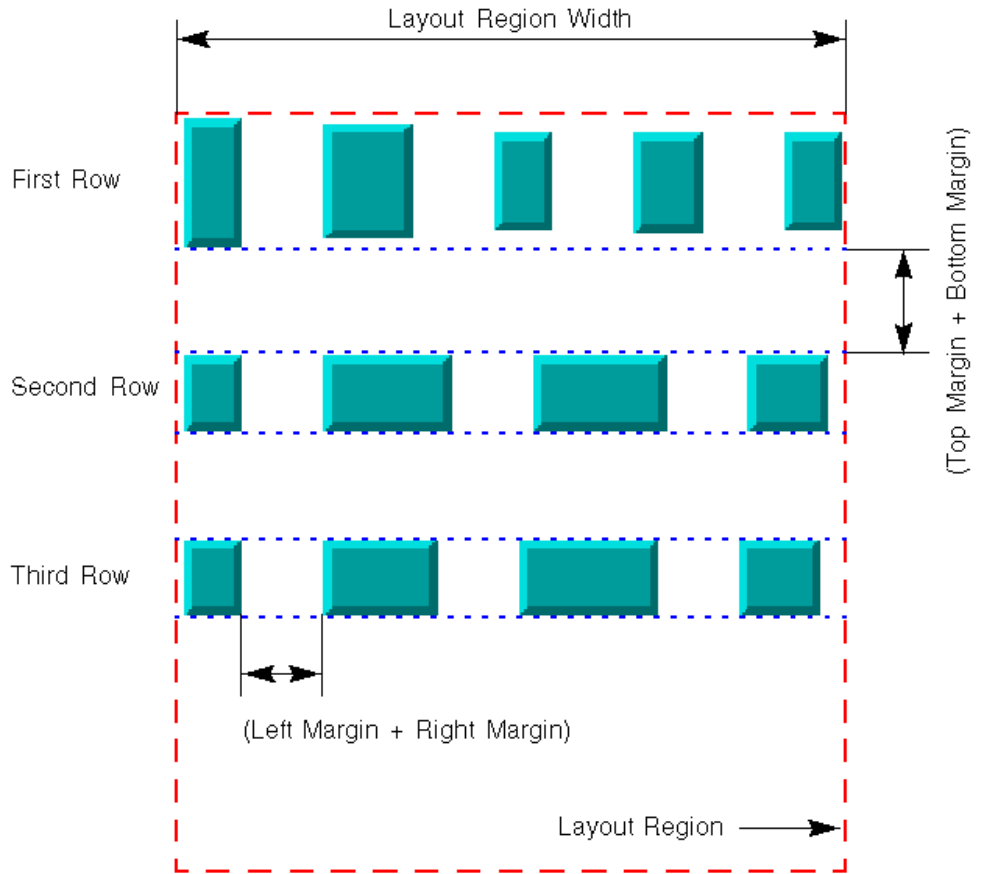
Note: To preserve the stability, the incremental mode may keep some regions free. Therefore, the total area of the layout may be larger than in nonincremental mode, and, in general, the layout may not look as nice as in nonincremental mode.

Dimensional parameters (GL)

Dimensional parameters for the grid mode of the Grid Layout algorithm and *Dimensional parameters for the row/column mode of the Grid Layout algorithm* illustrate the dimensional parameters used in the Grid Layout algorithm. These parameters are explained in the sections that follow.



Dimensional parameters for the grid mode of the Grid Layout algorithm



Dimensional parameters for the row/column mode of the Grid Layout algorithm

Grid offset (GL)

The grid offset parameters control the spacing between grid lines. It is taken into account only by the grid mode (layout modes `TILE_TO_GRID_FIXED_WIDTH` and `TILE_TO_GRID_FIXED_HEIGHT`).

Example of setting grid offset (GL algorithm)

To set the horizontal and vertical grid offset:

In Java

Use the methods:

```
void setHorizontalGridOffset(float offset);
```

```
void setVerticalGridOffset(float offset);
```

The grid offset is the critical parameter for the grid mode. If the grid offset is larger than the size of the nodes (plus margins), an empty space is left around the node. If the grid offset is smaller than the size of the nodes (plus margins), the node will need to be placed on more than one grid cell. The best choice for the grid offsets depends on the application. It can be computed according to either the maximum size of the nodes (plus margins) or the medium size, and so on. Of course, if all the nodes have a similar size, the choice is straight-forward.

Margins (GL)

The margins control the space around each node that the layout algorithm keeps empty.

Example of specifying margins (GL algorithm)

To set the margins:

In Java

Use the methods:

```
void setTopMargin(float margin);
```

```
void setBottomMargin(float margin);
```

```
void setLeftMargin(float margin);
```

```
void setRightMargin(float margin);
```

The meaning of the margin parameters is not the same for the grid modes as for the row/column modes.:

- ◆ In grid modes, they represent the minimum distance between the node border and the grid line (see *Dimensional parameters for the grid mode of the Grid Layout algorithm.*)
- ◆ In row/column modes, they are used to control the vertical distance between the rows or the horizontal distance between the columns and the horizontal or vertical minimal distance between the nodes in the same row or column (see *Dimensional parameters for the row/column mode of the Grid Layout algorithm.*)

The default value for all the margin parameters is 5.

Nested layouts

Describes how to perform a layout on a nested graph and explains the utilities that are available for nested graphs.

In this section

Concepts for nested layouts

Explains nested graphs and related concepts.

Layout of nested graphs in code

Describes how to perform a layout on nested graphs.

Recursive layout

Describes the *Recursive Layout* (class `IlvRecursiveLayout` from the package `ilog.views.graphlayout.recursive`).

Recursive layout modes

Describes the modes available in this layout.

Multiple layout

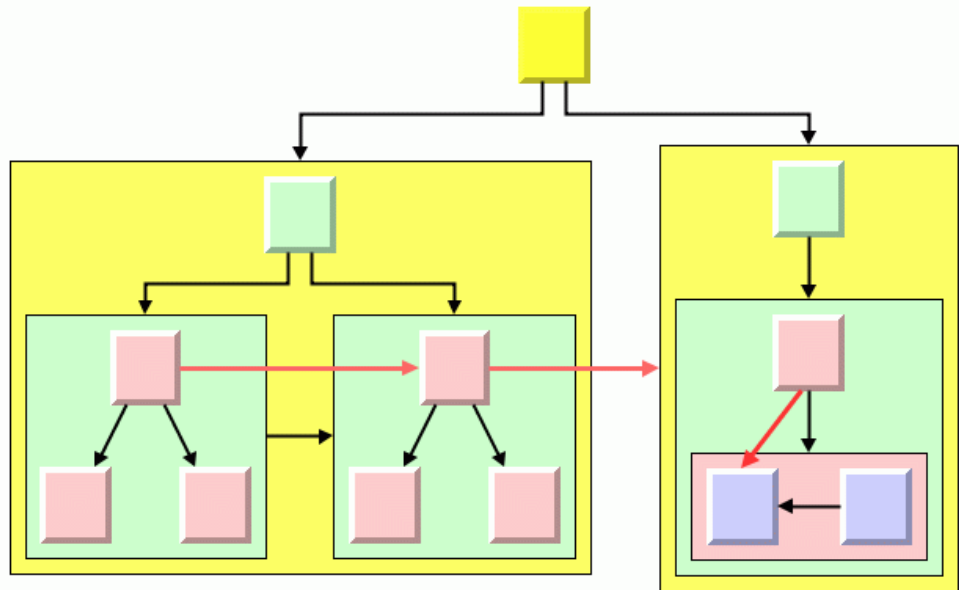
Describes the *Multiple Layout* class (class `IlvMultipleLayout` from the package `ilog.views.graphlayout.multiple`).

Concepts for nested layouts

IBM® ILOG® JViews Graph Layout for Eclipse supports GEF nested graphs, that is, it can render graphs containing nodes that are graphs.

Warning: The GMF compartments are partially supported by JViews Graph Layout for Eclipse; recursive layouts and intergraph links are not yet supported. The Nested layouts section is only interesting for GEF users.

The following figure shows an example of a nested graph.



Example of a Nested Graph

A graph that is a node in another graph is called a subgraph. Links that connect nodes of different subgraphs are called subgraph links. The red links in the figure are intergraph links.

Layout of nested graphs in code

Describes how to perform a layout on nested graphs.

In this section

The classes that support nested graphs

Explains how layouts are performed on nested graphs.

Order of layouts in recursive layouts

Explains the order in which recursive layouts are applied on nested graphs.

Simple recursion: applying the same layout to all subgraphs

Describes how to obtain a nested graph with the same layout throughout.

Advanced recursion: mixing different layouts in a nested graph

Describes the case where you want to mix different layouts in one nested graph.

The classes that support nested graphs

The `IGrapherEditPart` interface provided by IBM® ILOG® JViews Graph Layout for Eclipse allows nested graphs to be laid out.

In an application that works directly on an instance of `IGrapherEditPart`, a recursive layout must be performed explicitly.

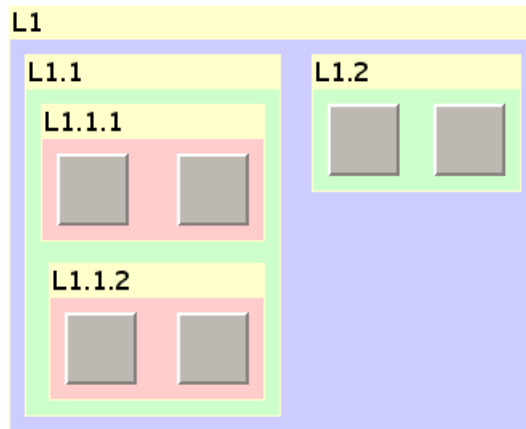
For more information, see `Graphers`.

The mechanism uses the auxiliary classes `IlvRecursiveLayout` and `IlvMultipleLayout` internally. They are explained in detail in *Recursive layout* and *Multiple layout*.

Order of layouts in recursive layouts

Assume grapher 1 contains two subgraphers L1.1 and L1.2, and subgrapher 1.1 contains two subgraphers L1.1.1 and L1.1.2, as shown in the following figure. The recursive layout needs to be applied in reverse order, as follows:

1. Layout on L1.1.1
2. Layout on L1.1.2
3. Layout on L1.1
4. Layout on L1.2
5. Layout on L1



Nested graph with recursive layouts

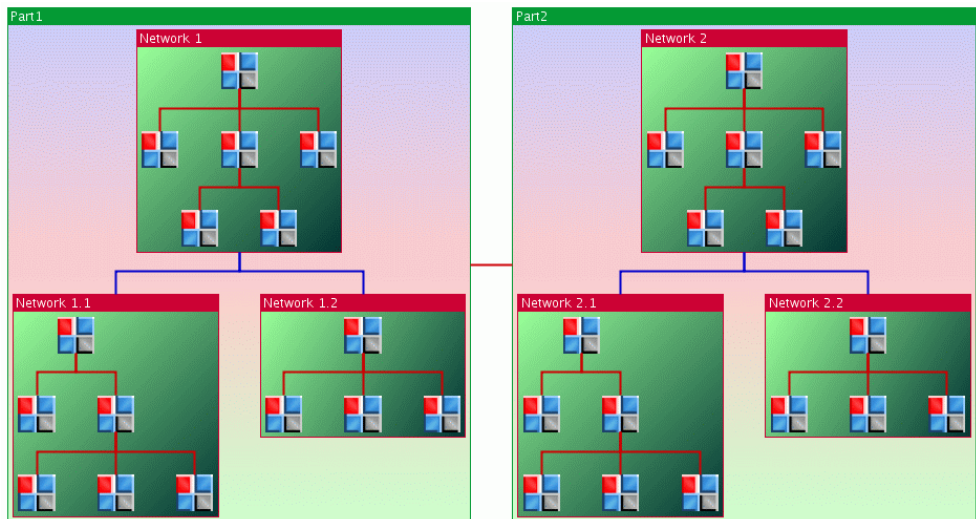
This means that the layout is applied to the graph once all the layouts of its subgraphs have been applied first. In our example, all layouts of subgrapher L1.1 are finished before the layout of grapher L1 starts. This is the correct order for a recursive layout. This order ensures that the layout of a subgraph does not invalidate the layout of its parent graphs.

Simple recursion: applying the same layout to all subgraphs

You can apply the same layout where both the following conditions hold:

- ◆ The same layout algorithm needs to be applied to the topmost graph and all its subgraphs.
- ◆ The settings of the layout algorithm (that is, the layout parameters) need to be the same for the topmost graph as for all the subgraphs.

The following figure shows an example where a Tree Layout is applied to the topmost graph as well as to all its subgraphs. Moreover, the settings of the Tree Layout algorithm are the same for all the graphs: the application does not need, for instance, one flow direction in the topmost graph and a different one in the subgraphs.



Example of a recursive layout of a nested graph

Obtaining such recursive layouts is very easy. The class `IlvGraphLayout` provides a special version of the `performLayout` method:

```
performLayout(boolean force, boolean redraw, boolean traverse)
```

When the last `boolean` argument is set to `true`, the layout is applied not only to the graph attached to the layout instance, but also, in a recursive way, to its subgraphs.

Internal mechanism

The internal mechanism is based on the principle that a given layout instance is used for only one graph and is not reused for its subgraphs. Therefore, the Tree Layout instance is automatically “cloned” using the `copy()` method of the class `IlvGraphLayout`.

Furthermore, the graph layout is applied to a graph model, and the same principle holds for the graph models: a given graph model instance is used for only one graph and is not reused for subgraphs.

The graph models for the subgraphs are created by calls to the `getGraphModel(java.lang.Object)` method of the class `IlvGraphLayout`, which in turn creates the graph model using the method `createGraphModel(java.lang.Object)` of the class `IlvGraphModel`.

All these operations are done automatically, in a completely transparent way. All you have to do is to call the method `performLayout` with the `traverse` argument set to `true`.

If needed, you can get the layout instances applied on the subgraphs by calling the following method on `IlvGraphLayout`:

```
Enumeration getLayouts(boolean preOrder)
```

This method returns an enumeration of instances of `IlvGraphLayout`. If the `preOrder` flag is `true`, the layout of the parent graph occurs before the layout of its children in the enumeration. If the `preOrder` flag is `false`, the layout of the parent graph occurs after the layout of its children. For example, in the graph of *Nesting structure in a graph*, the call `getLayouts(true)` returns the layouts for the subgraphs in this order: L1, L1.1, L1.1.1, L1.1.2, L1.2. The call `getLayouts(false)` returns the layouts for the subgraphs in this order: L1.1.1, L1.1.2, L1.1, L1.2, L1.

Java code sample

The following code sample illustrates how to apply a single layout algorithm to a nested graph:

```
// Create the layout instance
IlvTreeLayout layout = new IlvTreeLayout();

// Attach the topmost grapher to the layout
// The grapher has child edit parts instance of IGrapherEditpart
GraphModel graphModel = new GraphModel(myTopLevelGrapherEditPart);
layout.attach(graphModel);

// Perform the recursive layout
try {
    int code = layout.performLayout(true, true, true);

    System.out.println("Layout completed (code " +
        code + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
...
// Detach the grapher when layout no more needed
layout.detach();
graphModel.dispose();
...
```

Layout parameters

Section *Internal mechanism* explains that, when applying the same layout algorithm in a recursive way, the layout instances for the subgraphs are obtained by “cloning” the layout instance attached to the topmost graph.

The layout parameters of the “clone” are the same as the parameters of the topmost layout, except for the parameters that are specific to a node or a link. Such parameters are not copied when the layouts are cloned and need to be set separately for each layout instance.

For example, if you need to declare a node `node1` contained in the subgraph `mySubgrapherEditPart` of the topmost graph `myTopLevelGrapherEditPart` as fixed (see *Preserve fixed nodes*), you can use the following code:

```
...

// Create the layout instance
IlvTreeLayout layout = new IlvTreeLayout();

// Attach the topmost grapher to the layout
// Contains the child edit part mySubgrapherEditPart instance of
IGrapherEditPart
layout.attach(new GraphModel(myTopLevelGrapherEditPart));

// Ask the layout algorithm to not move the nodes
// specified as fixed. This settings is automatically
// copied on the sublayouts. Do not specify this global
// settings directly on the sublayout, because it gets automatically
// the same settings as the topmost layout
layout.setPreserveFixedNodes(true);

// Search the layout instance used for mySubgrapherEditPart
IlvGraphLayout subLayout = null;
Enumeration layouts = layout.getLayouts(true);
while (layouts.hasMoreElements()) {
    subLayout = (IlvGraphLayout)layouts.nextElement();
    if (((GraphModel)subLayout.getGraphModel()).getContents() ==
mySubgrapherEditPart)
        break;
}

// Specify node1 (contained in mySubgrapherEditPart) as fixed
subLayout.setFixed(node1, true);

// Now perform the recursive layout. The node node1 will be considered as fixed
// by the layout applied to mySubgrapherEditPart
...

```

Note: You should not try to change any global settings of the layouts applied to the subgraphs (that is, settings that are not specific to a node or a link). These settings are copied

anyway from the layout instance of the topmost grapher, so your changes would be erased just before the recursive layout runs.

Advanced recursion: mixing different layouts in a nested graph

The need for mixing layouts arises when at least one of the following conditions is met:

- ◆ The layout algorithm to be applied on subgraphs is not the same as the algorithm needed for the topmost graph.
- ◆ Different layouts need to be applied to different subgraphs.
- ◆ The same layout algorithm needs to be applied to different graphs but with different settings.

In these cases of *advanced recursion*, where you want to apply different layouts to different subgraphs, you need to specify which layout should be used for which subgraph. Furthermore you need to start the layouts in the correct order. This is called *recursive layout*.

The class `IlvRecursiveLayout` is a subclass of `IlvGraphLayout`, but it is not a real layout algorithm. It is rather a facility to apply other layout algorithms recursively on a nested graph.

The class `IlvRecursiveLayout` can also be used to apply the same layout to all subgraphs. In fact, when using the API explained in subsection *Simple recursion: applying the same layout to all subgraphs*, an instance of `IlvRecursiveLayout` is used internally.

The class `IlvRecursiveLayout` can furthermore be used to apply multiple layouts to the same nested graph. This is for instance necessary if for each subgraph, a node layout and a separate link layout must be applied.

Further details and code samples of the class `IlvRecursiveLayout` are explained in the following section *Recursive layout*.

To apply layout algorithms recursively:

1. Allocate and attach an instance of `IlvRecursiveLayout`. Since it is a subclass of `IlvGraphLayout`, you use the same mechanism as for all other graph layout classes:

```
IlvRecursiveLayout recLayout = new IlvRecursiveLayout();
recLayout.attach(new GraphModel(myTopLevelGrapherEditPart));
```

2. Specify which layout style should be used for each subgraph. You must allocate an individual instance of `IlvGraphLayout` for each subgraph.

```
recLayout.setLayout(subgraph1, new IlvTreeLayout());
recLayout.setLayout(subgraph2, new IlvBusLayout());
recLayout.setLayout(subgraph3, new IlvGridLayout());
```

3. Set the layout parameters of these individual layouts of the subgraphs as needed.
4. Apply the recursive layout to the top-level grapher. This automatically applies the sublayouts to the subgraphs as well. Since `IlvRecursiveLayout` is a subclass of `IlvGraphLayout`, you use the same method as for all other graph layout classes

```
try {
```

```
        recLayout.performLayout();
    }
    catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
    }
}
```

- 5.** Detach the recursive layout from the top-level grapher when it is no longer needed. This automatically detaches all sublayouts from all subgraphers.

```
recLayout.detach();
```


Recursive layout

Describes the *Recursive Layout* (class `IlvRecursiveLayout` from the package `ilog.views.graphlayout.recursive`).

In this section

Overview of recursive layout

Describes classes associated with recursive layout with a diagram.

Features

Describes the features of the layout.

Generic features and parameters

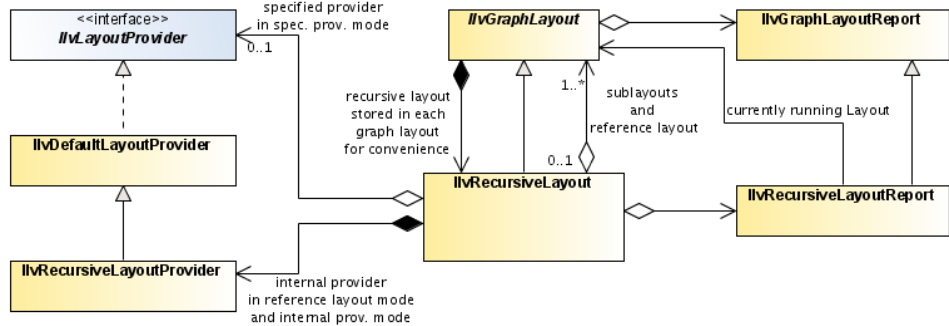
Describes the generic features and parameters of the layout.

Overview of recursive layout

The Recursive Layout class is not a layout algorithm but rather a facility to apply another layout algorithm recursively on a nested graph. It traverses the nesting structure starting from the graph that is attached to the Recursive Layout itself and recursively applies a layout on all subgraphs. You can tailor which sublayout must be applied to which subgraph.

There are basically two scenarios:

- ◆ The same layout style must be applied to all subgraphs.
- ◆ An individual layout style must be applied to each subgraph.



The class `IlvRecursiveLayout` which manages sublayouts for nested graphs

Java code sample: same layout style everywhere

This sample assumes that you want to apply a Tree Layout to a nested graph and that each subgraph should be laid out with the same global layout parameters.

The Tree Layout algorithm handles only flat graphs, that is, if applied to an attached graph, it lays out only the nodes and links of the attached graph, but not the nodes and links of the subgraphs that are nested inside the attached graph. Hence the Tree Layout must be encapsulated into a Recursive Layout.

The Recursive Layout traverses the entire nesting hierarchy of the attached graph, while the encapsulated Tree Layout lays out each (flat) subgraph of the nesting hierarchy during the traversal.

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.runtime.recursive.*;
import ilog.views.eclipse.graphlayout.runtime.tree.*;
...
IlvRecursiveLayout layout = new IlvRecursiveLayout (IlvTreeLayout ());
```

```

GraphModel graphModel = new GraphModel(myTopLevelGrapherEditPart);
layout.attach(graphModel);
try {
    IlvRecursiveLayoutReport layoutReport =
        (IlvRecursiveLayoutReport)layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
...
// detach the Recursive Layout when it is no longer needed
layout.detach();
graphModel.dispose();
...

```

This mode of the Recursive Layout is called *reference layout mode*. In this case, a Tree Layout is performed recursively on the top-level graph and on each subgraph. All layouts are performed with the same global layout parameters.

Note: The term “global layout parameter” applies to the parameters that do not depend on a specific node or link. For example, Tree Layout has a global layout parameter set by `setGlobalLinkStyle`, as well as a layout parameter set by `setLinkStyle(link, style)` which is local to a link.

You can change the global layout parameters by accessing the reference layout of the Recursive Layout:

```

IlvTreeLayout treeLayout = (IlvTreeLayout)layout.getReferenceLayout();
treeLayout.setFlowDirection(IlvDirection.Left);

```

Technically, the reference layout instance is not applied to each subgraph because each subgraph needs an individual layout instance. The reference layout instance is only applied to the top-level graph. Furthermore, a clone of the reference instance is created for each subgraph. This clone remains attached to the subgraph as long as the Recursive Layout is attached to the top-level graph. Before layout is performed, the global layout parameters are copied from the reference layout instance to each cloned layout instance.

Sometimes, you want to specify local layout parameters for individual nodes and links. In this case, you need to access the cloned layout instance that is attached to the subgraph that owns the node or link. For instance, to the link style of an individual link, use:

```

IlvTreeLayout treeLayout =
(IlvTreeLayout) layout.getLayout(GraphModel.getLowestCommonAncestor(link.

```

```
getSource(), link.getTarget()));  
treeLayout.setLinkStyle(link, IlvTreeLayout.ORTHOGONAL_STYLE);
```

You cannot use the reference layout mode in the following cases:

- ◆ The layout algorithm to be applied on subgraphs is not the same as the algorithm needed for the topmost graph (the reference layout).
- ◆ The same layout algorithm, but using different global parameter settings, needs to be applied on different subgraphs.

In these cases, you can use one of the other modes.

Java code sample: mixing different layout styles

The following example shows the second scenario: Each subgraph should be laid out by a different layout style or with individual global layout parameters. In this case, you use the *internal provider mode* of the Recursive Layout.

We assume that you have a graph with three subgraphs. The top-level graph and the first subgraph should be processed with Tree Layout, the second subgraph with Bus Layout, and the third subgraph with Grid Layout. You have to specify which layout should be used for which subgraph, and then you can perform the layout.

```
...  
import ilog.views.*;  
import ilog.views.eclipse.graphlayout.runtime.*;  
import ilog.views.eclipse.graphlayout.GraphModel;  
import ilog.views.eclipse.graphlayout.runtime.recursive.*;  
import ilog.views.eclipse.graphlayout.runtime.tree.*;  
import ilog.views.eclipse.graphlayout.runtime.bus.*;  
import ilog.views.eclipse.graphlayout.runtime.grid.*;  
...  
IlvRecursiveLayout layout = new IlvRecursiveLayout();  
  
GraphModel graphModel = new GraphModel(myTopLevelGrapherEditPart);  
layout.attach(graphModel);  
  
// specify the layout of the top level graph  
layout.setLayout(null, new IlvTreeLayout());  
// specify the layout of subgraphs  
layout.setLayout(subgraph1, new IlvTreeLayout());  
layout.setLayout(subgraph2, new IlvBusLayout());  
layout.setLayout(subgraph3, new IlvGridLayout());  
  
// perform layout  
try {  
    IlvRecursiveLayoutReport layoutReport =  
        (IlvRecursiveLayoutReport) layout.performLayout();  
  
    int code = layoutReport.getCode();  
  
    System.out.println("Layout completed (" +  
        layoutReport.codeToString(code) + ")");  
}
```

```

}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
...
// detach the Recursive Layout when it is no longer needed
layout.detach();
graphModel.dispose();
...

```

In this scenario, there is no reference layout. All layout parameters of different subgraphs are independent. You need to specify new, independent layout instances for each subgraph; otherwise no layout will be performed for the corresponding subgraph. The layout instances are attached to the subgraph as long as the Recursive Layout is attached to the top-level graph. You can specify in this example different global layout parameters for the Tree Layout of the top-level graph and the Tree Layout of `subgraph1`. You access the layout instance of each individual subgraph to change global layout parameters for this subgraph as well as parameters of nodes and links of the subgraph. For instance if `node1` belongs to `subgraph1` and `node2` belongs to `subgraph2`, you can set individual global and local layout parameters in this way:

```

// access the layout of the top level graph
IlvTreeLayout treeLayout1 = (IlvTreeLayout) layout.getLayout(null);
treeLayout1.setFlowDirection(IlvDirection.Bottom);
// access the layouts of the subgraphs
IlvTreeLayout treeLayout2 = (IlvTreeLayout) layout.getLayout(subgraph1);
treeLayout2.setFlowDirection(IlvDirection.Left);
treeLayout2.setAlignment(node1, IlvTreeLayout.TIP_OVER);
IlvBusLayout busLayout = (IlvBusLayout) layout.getLayout(subgraph2);
busLayout.setOrdering(IlvBusLayout.ORDER_BY_HEIGHT);
busLayout.setBus(node2);
IlvGridLayout gridLayout = (IlvGridLayout) layout.getLayout(subgraph3);
gridLayout.setLayoutMode(IlvGridLayout.TILE_TO_COLUMNS);

```

Java code sample: using a specified layout provider

The IBM® ILOG® JViews Graph Layout for Eclipse library provides a flexible mechanism for the choice of the layout instance to be applied to each subgraph in a nested graph: the layout provider. In the previous example, a layout provider was used internally. For simplicity, the details of the mechanism are hidden, and you select the choice of layout by using the method `setLayout` on the Recursive Layout instance. Therefore, this layout mode is called *internal provider mode*.

However, you can also design your own layout provider and use it inside the Recursive Layout. This is the *specified provider mode* of the Recursive Layout.

A layout provider is a class that implements the interface `IlvLayoutProvider`. The interface has a unique method:

```

getGraphLayout(IlvGraphModel graphModel)

```

This method must return the layout instance to be used for the graph model passed as the argument, or `null` if no layout is required for this graph. When performing the Recursive Layout, these methods get the layout instance to be used for each graph from the specified layout provider.

To implement the interface `IlvLayoutProvider`, you must decide how the choice of the layout instance is done. A possible implementation of the `getGraphLayout` method is the following:

```
public IlvGraphLayout getGraphLayout(IlvGraphModel graphModel)
{
    EditPart editPart = ((GraphModel)graphModel).getContents();

    IlvGraphLayout layout = null;
    // if the edit part is instance of MySubgrapherEditPart1, returns a Tree
    // Layout, otherwise returns a Hierarchical Layout

    if (editPart instanceof MySubgrapherEditPart1)
        layout = new IlvTreeLayout();
    else
        layout = new IlvHierarchicalLayout();

    layout.attach(graphModel);

    return layout;
}
```

Of course, this is only an example among many possible implementations. The implementation may decide to store the newly allocated layout instance to avoid allocating a new one when the method is again called for the same graph.

If you have implemented a layout provider, you can use it in the Recursive Layout in the following way:

```
..
import ilog.views.*;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.recursive.*;

IlvLayoutProvider layoutProvider = ...
IlvRecursiveLayout layout = new IlvRecursiveLayout(layoutProvider);
GraphModel graphModel = new GraphModel(myTopLevelGrapherEditPart);
layout.attach(graphModel);

// Perform the layout
try {
    IlvRecursiveLayoutReport layoutReport =
        (IlvRecursiveLayoutReport) layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
```

```
        layoutReport.codeToString(code) + " ");
    }
    catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
    }
    ...
    // detach the Recursive Layout when it is no longer needed
    layout.detach();
    graphModel.dispose();
    ...
```

Features

- ◆ This subclass of `IlvGraphLayout` is not a usual layout algorithm but rather a facility to manage the layout of a nested grapher.
- ◆ Three layout modes: reference layout mode, internal provider mode, and specified provider mode.
- ◆ Allows you to perform a layout algorithm recursively in a nested grapher.
- ◆ Allows you to perform a recursive layout on a nested grapher while each subgrapher uses an individual layout style.
- ◆ Layout features, speed, and quality depend on the features, speed, and quality of the sublayouts.

Generic features and parameters

Depending on the support of its sublayouts, Recursive Layout may support the following generic parameters defined in the `IlvGraphLayout` class (see *Generic parameters and features*):

- ◆ *Allowed time*
- ◆ *Percentage completion calculation*
- ◆ *Stop immediately*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed time

The Recursive Layout can stop the entire layout of a nested graph after a certain amount of time. If the allowed time setting has elapsed, the Recursive Layout stops; that means it stops the currently running layout of a subgraph and skips the subsequent layouts of subgraphs that have not yet been started. If at the stop time point a sublayout is running on a subgraph that does not support the “allowed time” feature, then this sublayout first runs to completion before the Recursive Layout is stopped. If the Recursive Layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Percentage completion calculation

The Recursive Layout calculates the percentage of completion. This value can be obtained from the layout report during the run of the layout. The value is, however, a very rough estimation. If the layouts on the subgraphs do not support the calculation of the percentage completion, the Recursive Layout can report the percentage based only on the information how many layouts of subgraphs are already finished. For instance, if the entire nesting structure contains five nested graphs, the mechanism reports 20% after the layout of the first subgraph has finished, 40% after the layout of the second subgraph has finished, and so on. If the layouts of the subgraphs support the calculation of the percentage completion, the Recursive Layout calculates a more detailed percentage. In most cases, the calculated percentage is only a very rough estimation that does not always grow linearly over time. (For a detailed description of this feature, see *Percentage of completion calculation* and *Listener layout*)

Stop immediately

The Recursive Layout can be stopped at any time. It stops the currently running layout of a subgraph after cleanup if the method `stopImmediately()` is called and skips the subsequent layouts of subgraphs that have not yet been started. If at the stop time point a sublayout is running on a subgraph that does not support the “stop immediately” feature, then this sublayout first runs to completion before the Recursive Layout is stopped. For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*. If the layout stops before completion, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Recursive layout modes

Describes the modes available in this layout.

In this section

Overview of recursive layout modes

Describes the modes available in this layout.

Reference layout mode

Describes the reference layout mode for a nested graph.

Internal provider mode

Describes the internal provider mode for a nested graph.

Specified provider mode

Describes the specified provider mode for a nested graph.

Accessing all sublayouts

Describes how to access all sublayouts through recursive layout.

Specific parameters

Describes how to access all sublayouts through recursive layout.

Listener layout

Describes some specific details of the Recursive Layout related to layout listeners.

For experts: more on layout providers

Describes the way to use the default layout provider.

Overview of recursive layout modes

The Recursive Layout has three different layout modes:

- ◆ *Reference layout mode*
- ◆ *Internal provider mode*
- ◆ *Specified provider mode*

The layout mode is determined by the constructor that you use. The way how to set global layout parameters of the sublayouts that are applied to the subgraphs is slightly different for each layout mode. You can query the current layout mode by using

```
int getLayoutMode()
```

The possible return values are:

- ◆ `IlvRecursiveLayout.REFERENCE_LAYOUT_MODE`: The same layout style with the same global layout parameters is applied to all subgraphs of the nested graph.
- ◆ `IlvRecursiveLayout.INTERNAL_PROVIDER_MODE`: The layout is applied using an internal recursive layout provider. The layout styles of individual subgraphs can be specified by using the method `setLayout`.
- ◆ `IlvRecursiveLayout.SPECIFIED_PROVIDER_MODE`: The layout is applied using an explicitly specified layout provider.

This section is divided as follows:

- ◆ *Accessing all sublayouts*
- ◆ *Convenience method for setting reference layout mode*

Reference layout mode

Use this mode if you want to apply the same layout style with the same global layout parameters to the entire nested graph. You first need to allocate the reference layout, that is a new instance of any graph layout algorithm (except `IlvRecursiveLayout`) that should be applied to all subgraphs of the nested graph. Then you allocate the Recursive Layout using the constructor with the reference layout as argument

```
IlvRecursiveLayout (IlvGraphLayout referenceLayout)
```

The reference layout is internally only used for the top-level graph of the nested graph. Clones of the reference layout are used for the subgraphs. Hence, all subgraphs are laid out with the same global layout parameters. To change the global layout parameters, you can access the reference layout by

```
IlvGraphLayout getReferenceLayout ()
```

Global layout parameters are those parameters that are independent from specific nodes or links. Other layout parameters are local to specific nodes or links. For instance, in `IlvHierarchicalLayout`, the method `setGlobalLinkStyle (style)` is a global layout parameter, while the method `setLinkStyle (link, style)` is a local layout parameter.

If you need to set layout parameters that are local to an individual node or link, you need to access the particular clone of the reference layout that is responsible for the subgraph that owns the node or link. After attaching the Recursive Layout to the top-level grapher or graph model, you can retrieve the layout instance of a specific subgraph by

```
IlvGraphLayout getLayout (Object subgraph)
```

However, in reference layout mode, it makes no sense to modify any global layout parameter on the returned instance. The global layout parameters are always taken from the reference layout only. If you pass `null` as subgraph, you get the layout instance of the top-level graph. This is actually the same layout instance as the reference layout.

The reference layout and its clones used during recursive layout remain attached to the subgraphs (or the graph models of the subgraphs, respectively) as long as the Reference Layout itself is attached. When detaching the Reference Layout, all layouts of subgraphs are automatically detached as well.

Internal provider mode

Use this mode if you want to perform graph layout on a nested graph, but either you need individual global layout parameters for specific subgraphs, or you want to lay out different subgraphs with different styles. In this case, there is no reference layout. You allocate the Recursive Layout using the constructor with no arguments

```
IlvRecursiveLayout ()
```

Before you can perform a layout, you need to specify which layout is used for which subgraph. First, you should attach the Recursive Layout to a graph. Then, to specify the layout of the top-level graph, call:

```
recursiveLayout.setLayout (null, sublayout);
```

To specify the layout of a specific subgraph, call

```
recursiveLayout.setLayout (subgraph, sublayout);
```

It is important that you assign a different layout instance for each subgraph. You cannot share the same layout instance among different subgraphs. We recommend, that you allocate a new, fresh layout instance for each subgraph. If you pass `null` as `sublayout`, then no layout is performed for this particular subgraph.

To set the layout for a subgraph and recursively for all its subgraphs, you can use

```
setLayout (Object subgraph, IlvGraphLayout layout, boolean traverse)
```

and pass the `true` argument for the `traverse` flag. This sets the layouts to a clone of the input layout for each subgraph starting at the input subgraph.

Internally, the Recursive Layout uses a layout provider of type `IlvRecursiveLayoutProvider`. You can access the current layout provider by

```
IlvLayoutProvider getLayoutProvider ()
```

However, in internal provider mode, it is mostly not necessary to manipulate the layout provider directly.

Since there is no reference layout, global layout parameters are independent for each subgraph. Global and local layout parameters can be set by accessing the particular layout instance that is assigned to a specific subgraph. After attaching the Recursive Layout to the top-level grapher or graph model, you can retrieve the layout instance of a specific subgraph by

```
IlvGraphLayout getLayout (Object subgraph)
```

If you pass `null` as `subgraph`, you get the layout instance of the top-level graph.

The layout instances of the subgraphs used during recursive layout remain attached to the subgraphs (or the graph models of the subgraphs, respectively) as long as the Reference Layout itself is attached. When you detach the Reference Layout, all layouts of subgraphs are automatically detached as well.

Specified provider mode

The specified provider mode can be used if you want to perform graph layout on a nested graph, but either you need individual global layout parameters for specific subgraphs, or you want to lay out different subgraphs with different styles. It is your own responsibility to manage the specified layout provider (unlike the case with the internal provider mode), but this is probably only necessary in very advanced applications.

In specified provider mode, there is no reference layout. You allocate the Recursive Layout using the constructor with your layout provider as argument

```
IlvRecursiveLayout (IlvLayoutProvider specifiedProvider)
```

You can access the current layout provider by

```
IlvLayoutProvider getLayoutProvider()
```

You should implement your layout provider in a way so that it delivers a different layout instance for each subgraph. The delivered layout instance must be attached to the graph model of the corresponding subgraph.

Since there is no reference layout, global layout parameters are independent for each subgraph. It is recommended that the implementation of the layout provider takes care of the setting of global and local layout parameters. Theoretically, you can use the method

```
IlvGraphLayout getLayout (Object subgraph)
```

which will return the layout instance that the specified layout provider delivers for the graph model of the input subgraph. If you pass `null` as `subgraph`, you get the layout instance of the top-level graph. However, the effect of this method depends on the implementation of the layout provider that was passed to the constructor of Recursive Layout.

The layout instances of the subgraphs used during recursive layout should be attached by the layout provider. They are usually not automatically detached when the Recursive Layout is detached. Unless you use one of the predefined providers of class

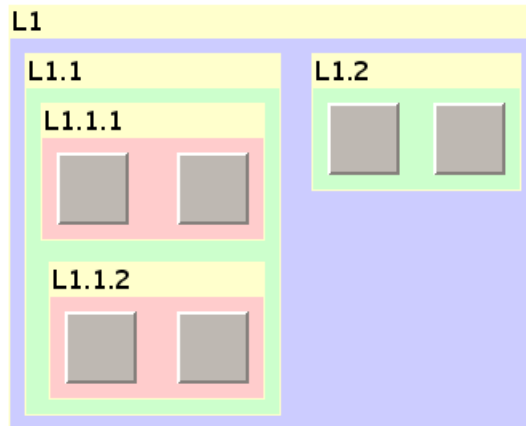
`IlvDefaultLayoutProvider` or `IlvRecursiveLayoutProvider`, you should traverse all layouts and detach them explicitly.

Accessing all sublayouts

When the Recursive Layout is attached, you can conveniently access all layouts that will be used during layout. This works in all layout modes:

```
Enumeration getLayouts(boolean preOrder)
```

As explained in *Internal mechanism*, the `getLayouts` method returns an enumeration of instances of `IlvGraphLayout`. If the `preOrder` flag is `true`, the layout of the parent graph occurs before the layout of its children in the enumeration. If the `preorder` flag is `false`, the layout of the parent graph occurs after the layout of its children. For example, in the graph in the following figure, the call `getLayouts(true)` returns the layouts for the subgraphs in this order: L1, L1.1, L1.1.1, L1.1.2, L1.2; the call `getLayouts(false)` returns the layouts for the subgraphs in this order: L1.1.1, L1.1.2, L1.1, L1.2, L1.



Nesting structure in a graph

Note: In specified provider mode, the enumeration returned by `getLayouts` contains the instances that are delivered by the specified provider. If the specified provider returns a different instance in each call of `getGraphLayout(IlvGraphModel)`, then the enumeration does not contain the instances that are later used during layout. Hence it is recommended to use layout providers that store the layout instances internally and return the same instance for the same graph model. The predefined `IlvDefaultLayoutProvider` and `IlvRecursiveLayoutProvider` store the layout instances internally.

Convenience method for setting reference layout mode

The class `IlvGraphLayout` contains a convenience method. To perform a recursive layout recursively, you can use:

```
int performLayout(boolean force, boolean redraw, boolean traverse)
```

If the `traverse` flag is `true`, it traverses the nested graph and performs the layout on each subgraph. In fact, this is just a shortcut for the reference mode of Recursive Layout. The statement

```
flatLayout.performLayout(force, redraw, true);
```

is equivalent to creating a Recursive Layout in reference mode that uses the `flatLayout` as reference layout:

```
IlvRecursiveLayout recursiveLayout = new IlvRecursiveLayout(flatLayout);  
recursiveLayout.performLayout(force, redraw);
```

Specific parameters

Besides some expert parameters, Recursive Layout does not provide any specific layout parameters. You can set specific layout parameters of the sublayouts individually by accessing them via `getLayout(Object)`:

```
IlvGraphLayout sublayout = recursiveLayout.getLayout(subgraph);
sublayout.setParameter(parameter);
```

However, Recursive Layout has some convenient methods that automatically traverse the nested graph recursively and set the corresponding parameter at each sublayout of a subgraph that supports this parameter. This works well particularly in reference layout mode. In internal or specified provider mode, it takes only the current nesting structure into account. If you change the specific layout of a subgraph or the nesting structure (for example, by adding a new subgraph) after using such a convenience method, the new layout of the new subgraph usually has a different setting, so you may need to apply the convenience method again.

The following methods traverse the nested graph recursively and set the corresponding parameter on all sublayouts (see *Generic parameters and features* and *Using advanced features* for details):

- ◆ `void setUseDefaultParameters(boolean option)`
- ◆ `void setMinBusyTime(long time)`
- ◆ `void setInputCheckEnabled(boolean enable)`
- ◆ `void propagateLayoutOfConnectedComponentsEnabled(boolean enable)`
- ◆ `void propagateLayoutOfConnectedComponents(IlvGraphLayout layout)`
- ◆ `void propagateLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface linkConnectionBoxInterface)`
- ◆ `void propagateLinkClipInterface(IlvLinkClipInterface linkClipInterface)`
- ◆ `void setConnectionPointCheckEnabled(boolean enable)`

There is a generic propagation mechanism for setting any parameter which is implemented by reflection. For example, the following call traverses the nested graph recursively, checks for each sublayout using introspection whether a method called `setFlowDirection` exists, and passes the input value `direction` to this method. As a result, all sublayouts that have a flow direction parameter will use the same flow direction, while the layout parameters of those layouts that do not have a flow direction remain unchanged:

```
int code = recursiveLayout.propagateLayoutParameter("flowDirection",
null, direction);
```

The second argument of `propagateLayoutParameter` can be used to select only specific layout classes. The call

```
int code = recursiveLayout.propagateLayoutParameter("flowDirection",
                                                    IlvHierarchicalLayout.class, direction);
```

propagates the flow direction only to all those sublayouts that are instances of `IlvHierarchicalLayout`. For example if a subgraph uses a `Tree Layout`, its flow direction remains unchanged in this case, even though `IlvTreeLayout` has a method `setFlowDirection`.

The return code of the propagation method indicates whether setting the parameter has been successful. It is a bitwise-Or combination of the following bit masks:

- ◆ `IlvRecursiveLayout.PROPAGATE_PARAMETER_SET` - the parameter was successfully applied at some layout instance of a subgraph.
- ◆ `IlvRecursiveLayout.PROPAGATE_PARAMETER_AMBIGUOUS` - the method to set the parameter could not uniquely be determined at some layout instance, because there were many methods with the same name, which creates an unresolvable ambiguity. In this case, an arbitrary method is chosen among the ambiguous methods.
- ◆ `IlvRecursiveLayout.PROPAGATE_CLASS_MISMATCH` - the parameter was not applied at some layout instance of a subgraph because the layout instance did not match the specified layout class. This can happen only when a non-null layout class is specified as the second parameter of the method `propagateLayoutParameter`.
- ◆ `IlvRecursiveLayout.PROPAGATE_PARAMETER_MISMATCH` - the parameter was not applied at some layout instance of a subgraph because no matching method with appropriate argument types was found via reflection, or because the security manager of the Java Virtual Machine did not allow reflection.
- ◆ `IlvRecursiveLayout.PROPAGATE_EXCEPTION` - the method to set the parameter was applied but threw an exception at some layout instance of a subgraph.

For further details about the propagation mechanism, see the class `IlvRecursiveLayout` in the *Java API Reference Manual*.

Listener layout

Event listener layout is an advanced feature documented in *Using event listeners*. You need to understand that general description and the concept of layout listeners before you read this section.

The application can listen for layout events sent by the Recursive Layout or by each sublayout individually. For example, a progress bar that displays the progress of the entire nested layout should listen for the layout events fired by the Recursive Layout itself, while an application that wants to detect when a specific sublayout of a subgraph is started or stopped should listen for the layout events sent by that particular sublayout.

To install a layout event listener at the Recursive Layout, call usually:

```
recursiveLayout.addGraphLayoutEventListener(listener);
```

To install a layout listener that receives the layout events of all sublayouts of the Recursive Layout, you can call:

```
recursiveLayout.addSubLayoutEventListener(listener);
```

Note that in this case, the listener is installed at the Recursive Layout instance (not at the sublayout instances) but receives the events from the sublayouts (not from the Recursive Layout). An internal mechanism makes sure that the events are forwarded to the listener.

Alternatively, you could traverse the nesting structure and install the same listener at all subgraph layouts. However, this would have two disadvantages: it requires more memory and you need to reinstall or update the listener whenever you change the layout of a subgraph or the nesting structure by adding or removing subgraphs. When you use `addSubLayoutEventListener`, updating the listener is not necessary in this case.

For experts: more on layout providers

For information on use the Recursive Layout with a specified layout provider, see *Specified provider mode*.

The library provides a default implementation of the interface `IlvLayoutProvider`, named `IlvDefaultLayoutProvider`. In many cases, it is simpler either to use this class as is, or to subclass it, rather than directly implementing the interface.

The class `IlvDefaultLayoutProvider` allows you to set the layout instance to be used for each graph (called the preferred layout) with the method:

```
setPreferredLayout(IlvGraphModel graphModel, IlvGraphLayout layout, boolean detachPrevious)
```

The layout instance specified as the preferred layout is stored in a property of the graph model. The current preferred layout is returned by the method:

```
getPreferredLayout(IlvGraphModel graphModel)
```

The method returns `null` if no layout has been specified for this graph.

When the method `getGraphLayout` is called on the default provider, the previously specified preferred layout is returned, if any. Otherwise, a new layout instance is allocated by a call to the method

```
createGraphLayout(IlvGraphModel graphModel)
```

This newly created layout is recorded as the preferred layout of this graph, which is attached to the layout instance.

When a preferred layout has been specified for a given graph, the default implementation of the method `createGraphLayout` copies the layout instance that is the preferred layout of the nearest parent graph. Therefore, if a preferred layout `L` is specified for a graph `G` and no preferred layout is set for its subgraphs, then the graph `G` and all its subgraphs are laid out using the same layout algorithm `L` (copies of it are used for the subgraphs).

Note: You must call the method `detachLayouts` when you no longer need the layout provider instance; otherwise, the garbage collector may fail to remove some objects.

Java Code Sample:

The following Java™ code sample illustrates the use of the class `IlvDefaultLayoutProvider`.

```
...  
GraphModel graphModel = new GraphModel(myTopLevelGrapherEditPart);
```

```

// Create the layout provider
IlvDefaultLayoutProvider provider = new IlvDefaultLayoutProvider();

// Specify the preferred layouts for each grapher
// (this automatically attaches the layouts)
provider.setPreferredLayout(graphModel, new IlvTreeLayout());
provider.setPreferredLayout(graphModel.getGraphModel(mySubgrapherEditPart),
new IlvGridLayout());

// Create a recursive layout in specified provider mode
IlvRecursiveLayout layout = new IlvRecursiveLayout(provider);

// Perform the layout
try {
    IlvRecursiveLayoutReport layoutReport =
        (IlvRecursiveLayoutReport)layout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
...
// detach the layouts when the provider is no longer needed
provider.detachLayouts(graphModel, true);
// dispose the topmost adapter when no longer needed
graphModel.dispose();
...

```


Multiple layout

Describes the *Multiple Layout* class (class `IlvMultipleLayout` from the package `ilog.views.graphlayout.multiple`).

In this section

General information

Describes the multiple layout facility.

Features

Lists the features of multiple layout and shows the class diagram.

Generic features and parameters

Describes the generic features and parameters of multiple layout.

Specific parameters

Describes the specific parameters of multiple layout.

Attaching graph and labeling models

Describes how to attach graph and labeling models.

Accessing sublayouts

Describes how to access sublayouts of a multiple layout.

Combining multiple and recursive layout

Describes how to combine a multiple layout with a recursive layout.

The reference labeling model

Describes the reference labeling model for a recursive multiple layout.

General information

What is multiple layout?

The Multiple Layout class is not a layout algorithm but rather a facility to compose multiple layout algorithms and treat them as one algorithm object. This is necessary in particular when dealing with the recursive layout of nested submanagers (see *Recursive layout* and *Layout of nested graphs in code*) because performing the layouts recursively one after the other has a different effect than combining the layouts into one algorithm object and applying this object all at once. Multiple Layout should also be used to combine a normal layout with a Link Layout that routes intergraph links. This is illustrated in the following sample.

Java code sample

You can, for instance, combine a Tree Layout, a Link Layout, and an Annealing Label Layout into one object of type `IlvGraphLayout` in the following way:

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.multiple.*;
import ilog.views.eclipse.graphlayout.runtime.tree.*;
import ilog.views.eclipse.graphlayout.runtime.link.*;
import ilog.views.eclipse.graphlayout.runtime.labellayout.annealing.*;

IlvTreeLayout treeLayout = new IlvTreeLayout();
IlvLinkLayout linkLayout = new IlvLinkLayout();
IlvAnnealingLabelLayout labelLayout = new IlvAnnealingLabelLayout();
IlvMultipleLayout multipleLayout =
    new IlvMultipleLayout(treeLayout, linkLayout, labelLayout);

layout.attach(new GraphModel(myGrapherEditPart));

... /* Fill in code to set the layout parameters of treeLayout,
     * linkLayout and labelLayout.
     */
linkLayout.setInterGraphLinksMode(true);
...
```

By constructing a Multiple Layout instance in this way, the Tree Layout, Link Layout, and Label Layout become *sublayouts* of the Multiple Layout instance. Attaching the Multiple Layout will automatically attach its sublayouts.

The Multiple Layout has two slots for graph layouts and one slot for the label layout. Not all slots need to be used. You can pass `null` as the sublayout for unused slots. If you need more slots, you can compose a Multiple Layout that contains another Multiple Layout as a sublayout.

To perform the composed layout you use one of the following:

◆ *Simple layout*

◆ *Recursive layout*

Simple layout

You can perform the composed layout on a flat grapher (one which contains no subgraphers) in the following way:

```
try {
    IlvMultipleLayoutReport layoutReport =
        (IlvMultipleLayoutReport)multipleLayout.performLayout();

    int code = layoutReport.getCode();

    System.out.println("Layout completed (" +
        layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
```

The statement with `multipleLayout.performLayout()` in this case has the same effect as the sequence of the three statements:

```
treeLayout.performLayout();
linkLayout.performLayout();
labelLayout.performLayout();
```

Recursive layout

If you perform the Multiple Layout on a grapher that contains subgraphers, there is a difference in the order of the layout (see *Order of layouts in recursive layouts*). You apply a recursive layout on the grapher and its subgraphers in the following way:

```
IlvRecursiveLayout recursiveLayout = new IlvRecursiveLayout(multipleLayout);
try {
    IlvGraphLayoutReport layoutReport = recursiveLayout.performLayout();
    ...
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
```

or alternatively, in the following way (both ways are equivalent):

```
try {
    int layoutCode =
        (IlvMultipleLayoutReport)multipleLayout.performLayout(
```

```

        ... true, true, true);
    }
    catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
    }
}

```

Assume the attached grapher A contains a subgrapher B. The combined Multiple Layout applies its sublayouts in reverse order, as follows:

1. Tree Layout on B
2. Link Layout on B
3. Label Layout on B
4. Tree Layout on A
5. Link Layout on A
6. Label Layout on A

This means that all layouts of subgrapher B have finished before the layout of grapher A starts. This is the correct order for a recursive layout.

If you do not combine the three component layouts into a Multiple Layout, you can only apply them sequentially:

```

treeLayout.performLayout(true, true, true);
linkLayout.performLayout(true, true, true);
labelLayout.performLayout(true, true, true);

```

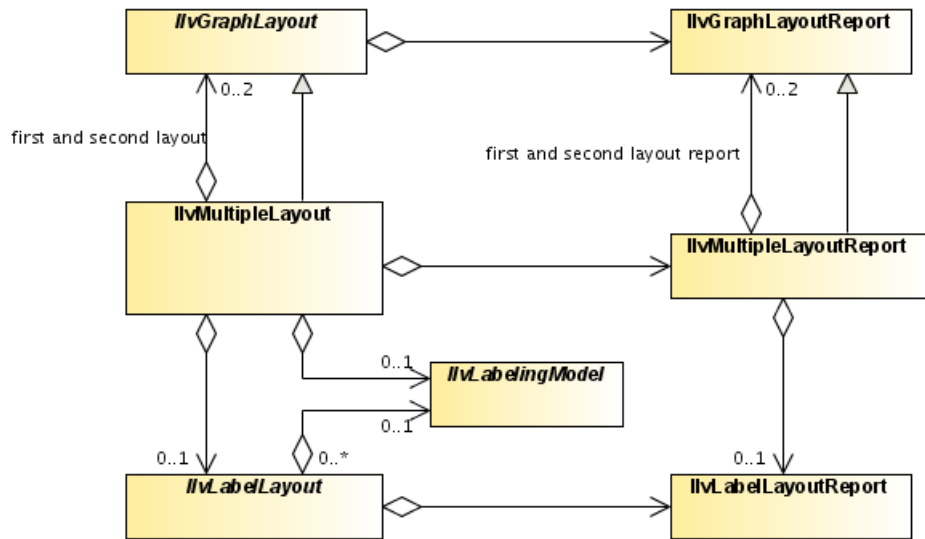
The effect of these three statements is slightly different than the effect of the Multiple Layout. The layouts are now applied in the following order:

1. Tree Layout on B
2. Tree Layout on A
3. Link Layout on B
4. Link Layout on A
5. Label Layout on B
6. Label Layout on A

This order is not usually suitable for the layout of nested graphers because the Tree Layout of grapher A is started too early. The Label Layout on grapher B in Step 5 may change the position of grapher B within grapher A, invalidating the result of the Tree Layout in Step 2. Hence, it is recommended that you combine multiple layout algorithms into one Multiple Layout object and apply this object as a whole to a nested grapher.

Features

- ◆ Allows the composing of two graph layout algorithms and one label layout algorithm into one layout object.
- ◆ Should be used to achieve the correct layout order when dealing with nested graphers.
- ◆ Layout features, speed, and quality depend on the features, speed, and quality of the sublayouts.



*The class *IlvMultipleLayout* can contain two sublayouts and one label layout*

Generic features and parameters

Depending on the support of its sublayouts, Multiple Layout may support the following generic parameters and features defined in the `IlvGraphLayout` class (see *Generic parameters and features*):

- ◆ *Allowed time*
- ◆ *Layout of connected components*
- ◆ *Percentage completion calculation*
- ◆ *Stop immediately*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed time

A Multiple Layout instance supports this feature if all of its sublayouts support the feature. If the allowed time setting has elapsed, the Multiple Layout stops; that means it stops the currently running sublayout and skips the subsequent sublayouts that have not yet been started. If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of connected components

The Multiple Layout instance can use the generic mechanism to lay out connected components if the sublayouts of type `IlvGraphLayout` support this feature. The sublayout of type `IlvLabelLayout` does not need special handling of connected components. For more information about this mechanism, see *Layout of connected components*.

Percentage completion calculation

The Multiple Layout calculates the percentage of completion. This value can be obtained from the layout report during the run of the layout. The value is, however, a very rough estimation. If the sublayouts do not support the calculation of the percentage completion, the Multiple Layout can report the percentage based only on the information that the sublayout has already finished. For instance, if there are three sublayouts, the mechanism reports 33% after the first sublayout has finished, 66% after the second sublayout has finished, and 100% after all three sublayouts have finished. If the sublayouts support the calculation of the percentage completion, the Multiple Layout calculates a more detailed percentage. For a detailed description of this feature, see *Percentage of completion calculation* and *Graph layout event listeners*.

Stop immediately

The Multiple Layout instance supports this feature if all its sublayouts support this feature. It stops the currently running sublayout after cleanup if the method `stopImmediately()` is called and skips the subsequent sublayouts that have not yet been started. For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*. If the layout stops before

completion, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters

Multiple Layout does not provide any specific layout parameters. However, you can set the generic and specific layout parameters of the sublayouts individually. For instance, you can construct a Multiple Layout instance from two graph layouts. Even though the Multiple Layout does not support setting fixed nodes on itself, you can fix nodes for the sublayouts individually by applying `setFixed` to the sublayout instances if the sublayouts support this feature:

```
IlvMultipleLayout multipleLayout =
    new IlvMultipleLayout(layout1, layout2, null);
GraphModel graphModel = new GraphModel(myGrapherEditPart);
multipleLayout.attach(graphModel);
if (layout1.supportsPreserveFixedNodes()) {
    layout1.setFixed(node1, true);
    ...
}
if (layout2.supportsPreserveFixedNodes()) {
    layout2.setFixed(node2, true);
    ...
}
try {
    // perform a multiple layout: node1 is fixed while layout1 runs
    // and node2 is fixed while layout2 runs
    IlvMultipleLayoutReport layoutReport =
        (IlvMultipleLayoutReport)multipleLayout.performLayout(
            true, true, true);
    ...
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
multipleLayout.detach();
graphModel.dispose();
```

Attaching graph and labeling models

If a graph model is attached to the Multiple Layout instance, the same graph model is automatically attached to the sublayouts of type `IlvGraphModel`.

Before you attach the graph model, if a sublayout of type `IlvLabelLayout` is defined, you need to call the method `setLabelingModel` in the following way:

```
multipleLayout.setLabelingModel(new LabelingModel(myGrapherEditPart));
multipleLayout.attach(new GraphModel(myGrapherEditPart));
```

If the Multiple Layout instance is detached from the graph model, all sublayouts are automatically detached as well.

Accessing sublayouts

You can obtain the sublayouts of a Multiple Layout instance by the following methods:

- ◆ `getFirstGraphLayout()`
which returns the graph layout that is applied first.
- ◆ `getSecondGraphLayout()`
which returns the graph layout that is applied second.
- ◆ `getLabelLayout()`
which returns the label layout that is applied last.

You can also change the sublayouts. Of course, you should not change the sublayouts while the Multiple Layout instance is attached to a graph. You should detach the graph first.

To set the sublayouts, the following methods are available:

```
void setFirstGraphLayout (IlvGraphLayout layout)
```

```
void setSecondGraphLayout (IlvGraphLayout layout)
```

```
void setLabelLayout (IlvLabelLayout layout)
```

Combining multiple and recursive layout

Often, the Multiple Layout is used inside a Recursive Layout. For convenience, IBM® ILOG® JViews Graph Layout for Eclipse provides a layout algorithm that combines both mechanisms: the Recursive Multiple Layout. This is a Recursive Layout (see *Recursive layout*) that uses an instance of Multiple Layout for each subgraph.

To apply a Tree Layout, a Link Layout, and an Annealing Label Layout recursively on a nested graph, you can use:

```
IlvRecursiveMultipleLayout layout = new IlvRecursiveMultipleLayout(  
    new IlvTreeLayout(),  
    new IlvLinkLayout(),  
    new IlvAnnealingLabelLayout());
```

This is in principle the same as a Recursive Layout that has a Multiple Layout as a reference layout:

```
IlvRecursiveLayout layout = new IlvRecursiveLayout(  
    new IlvMultipleLayout(  
        new IlvTreeLayout(),  
        new IlvLinkLayout(),  
        new IlvAnnealingLabelLayout());
```

The Recursive Multiple Layout has a first and second graph layout instance per subgraph, and a label layout instance per subgraph. You access these instances by the following methods:

- ◆ `IlvGraphLayout getFirstGraphLayout(Object subgraph)`
which returns the graph layout that is applied first to the subgraph.
- ◆ `IlvGraphLayout getSecondGraphLayout(Object subgraph)`
which returns the graph layout that is applied secondly to the subgraph.
- ◆ `IlvLabelLayout getLabelLayout(Object subgraph)`
which returns the label layout that is applied last to the subgraph.

If the `subgraph` parameter is `null` in these methods, the layout instances of the top-level graph are returned.

The reference labeling model

The Multiple Layout allows you to specify a particular labeling model by using the method `setLabelingModel`, but when you encapsulate the Multiple Layout into a Recursive Layout, this specification would need to be repeated for each layout instance of each subgraph. This would be inconvenient. However, the Recursive Multiple Layout takes care of this mechanism automatically.

The Recursive Multiple Layout generates all labeling models for all subgraph from a reference labeling model. Before attaching the Recursive Multiple Layout instance, you need to set the reference labeling model in the following way:

```
recursiveMultipleLayout.setReferenceLabelingModel(new LabelingModel
(myGrapherEditPart));
recursiveMultipleLayout.attach(new GraphModel(myGrapherEditPart));
```

The reference labeling model is used for the label layout of the top-level grapher. Clones of the reference labeling model are used for the label layouts of the subgraphers.

A simple way to perform a label layout recursively is the following:

```
...
IlvRecursiveMultipleLayout layout = new IlvRecursiveMultipleLayout(labelLayout)
;
GraphModel graphModel = new GraphModel(myGrapherEditPart);
LabelingModel labelingModel = new LabelingModel(myGrapherEditPart);
layout.setReferenceLabelingModel(labelingModel);
layout.attach(graphModel);
try {
    IlvGraphLayoutReport layoutReport = layout.performLayout();
    ...
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
layout.detach();
labelingModel.dispose();
graphModel.dispose();
```

If the Recursive Multiple Layout instance is detached from the top level graph model, all sublayouts are automatically detached as well and all labeling models of subgraphs (including the reference labeling model) are disposed of.

Automatic label placement

Describes the label placement algorithms.

In this section

Using the label layout API

Describes how to perform a label layout.

Releasing resources used during the layout of labels

Describes how to release resources that were created during the layout process.

Annealing label layout

Describes the *Annealing Label Layout* algorithm (class `IlvAnnealingLabelLayout` from the package `ilog.views.graphlayout.labellayout.annealing`).

Using the label layout API

Describes how to perform a label layout.

In this section

Overview

Provides useful links for label layout.

The label layout base class and its subclasses

Describes the classes associated with Label Layout.

Instantiating and attaching a subclass of `IlvLabelLayout`

Describes how to subclass the Label Layout class.

Performing a layout

Describes how to start a layout algorithm.

Performing a recursive layout on nested subgraphs

Describes how to start layout algorithms recursively on a nested grapher hierarchy.

The label layout report

Describes the report on label layout which is generated when you apply the layout.

Layout events and listeners

Describes the events provided by the label layout framework and how to listen for them.

Layout parameters and features in `IlvLabelLayout`

Explains which generic parameters and features are defined by the label layout class.

Overview

Before reading this information, you should be familiar with Obstacle and Label edit parts. For more information, see Working with label layout.

Note: Before reading this information, you should be familiar with the `IlvGraphLayout` class (see *Using the graph layout API*). Many of the concepts for the labeling layout mechanism are similar and not all details are repeated in this topic.

The label layout base class and its subclasses

The `IlvLabelLayout` class is the base class for all label layout algorithms. This class is an abstract class and cannot be used directly.

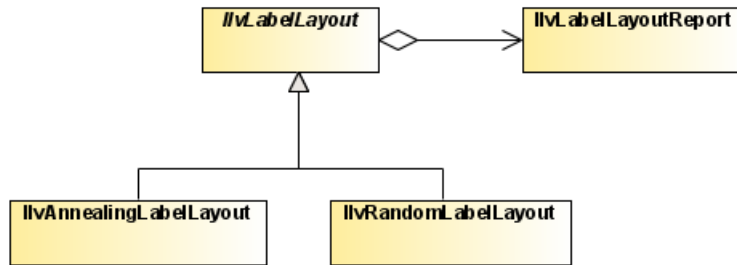
Subclasses of `IlvLabelLayout`

There are currently two subclasses:

- ◆ `IlvRandomLabelLayout` which randomizes the label positions for demonstration purpose.
- ◆ `IlvAnnealingLabelLayout` which carries out real label arrangement.

You can also create your own subclasses to implement other label layout algorithms.

Despite the fact that only subclasses of `IlvLabelLayout` are directly used to obtain the layouts, it is still necessary to learn about this class because it contains methods that are inherited (or overridden) by its subclasses. And, of course, you will need to understand it if you subclass it yourself.



The class `IlvLabelLayout` and its subclasses and relationship to layout reports

Instantiating and attaching a subclass of `IlvLabelLayout`

The class `IlvLabelLayout` is an abstract class. It has no constructors. You will instantiate a subclass as shown in the following example:

```
IlvAnnealingLabelLayout layout = new IlvAnnealingLabelLayout();
```

In order to place labels, a grapher needs to be attached to the layout instance. The attachment is done through a `LabelingModel`, an abstraction manipulated by the label layout algorithms to lay out labels. The following method, defined on the class `IlvLabelLayout`, allows you to specify the grapher you want to lay out:

```
void attach(IlvLabelingModel labelingModel)
```

For example:

```
...
LabelingModel labelingModel = new LabelingModel(myGrapherEditPart)
layout.attach(labelingModel);
```

The `attach` method does nothing if the specified labeling model is already attached. If a different labeling model is attached, this method first detaches this old labeling model, then attaches the new one. The labeling model can be obtained by:

```
IlvLabelingModel labelingModel = layout.getLabelingModel();
```

After layout, when you no longer need the layout instance, you should call the method

```
void detach()
```

If the `detach` method is not called, some objects may not be garbage-collected.

Performing a layout

The `performLayout` methods start the layout algorithm using the currently attached grapher and the current settings for the layout parameters. The method returns a report object that contains information about the behavior of the label layout algorithm.

```
IlvLabelLayoutReport performLayout ()  
IlvLabelLayoutReport performLayout (boolean force, boolean redraw)
```

The first method simply calls the second one with the `force` argument set to `false` and the `redraw` argument set to `true`.

- ◆ Because the `force` argument is set to `false` (by default), the layout algorithm first verifies whether it is necessary to perform the layout. It checks internal flags to see whether the manager or any of the parameters have changed since the last time the layout was successfully performed. A “change” can be any of the following:
 - Obstacles or labels were added or removed.
 - Obstacles or labels were moved or reshaped.
 - The value of a layout parameter was modified.
 - Users often do not want the layout to be computed again if no changes occurred. If there were no changes, the method `performLayout` returns without performing the layout. If the argument `force` is passed as `true`, the verification is skipped, and layout is performed even if no changes occurred.
- ◆ The `redraw` argument is ignored by IBM® ILOG® JViews Graph Layout for Eclipse.

The protected abstract method `layout (boolean redraw)` is then called. This means that control is passed to the subclasses that are implementing this method. The implementation computes the layout and moves the labels to new positions.

Performing a recursive layout on nested subgraphs

The examples and explanations above assume that you work with a flat grapher but you can create a hierarchy of nested graphers (see the following figure).

You can apply a recursive graph layout to the nested grapher hierarchy by calling:

```
graphLayout.performLayout(true, true, true);
```

However, it usually makes no sense to apply a label layout alone to nested graphers. When labels are placed in a subgrapher, this will likely change the bounds of the subgrapher; hence the node positions in its parent grapher will no longer be up-to-date and a new graph layout will be necessary.

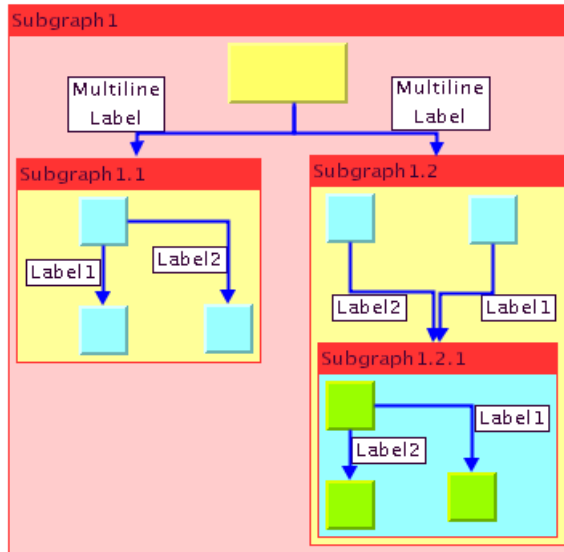
It makes sense to apply a label layout in combination with another graph layout to nested graphers.

1. First, the graph layout is applied to arrange the nodes and links nicely.
2. Then the label layout is applied to position the labels according to the node and link positions.
3. When this is finished for all subgraphers, then it can be done for the parent grapher.

To perform a graph layout and a label layout together, you can use the Multiple Layout class. This is a subclass of `IlvGraphLayout` that allows combining graph layouts with a label layout. The following sample shows how to apply a Tree Layout and an Annealing Label Layout in combination on a subgrapher.

```
IlvTreeLayout treeLayout = new IlvTreeLayout();
IlvAnnealingLabelLayout labelLayout = new IlvAnnealingLabelLayout();
IlvGraphLayout multipleLayout = new IlvMultipleLayout(treeLayout,
                                                    null,
                                                    labelLayout);
// Now set the parameters for tree layout and label layout ...
// Finally, perform a recursive layout that handles tree layout and label
// layout together
GraphModel graphModel = new GraphModel(myGrapherEditPart);
LabelingModel labelingModel = new LabelingModel(myGrapherEditPart);
multipleLayout.setLabelingModel(labelingModel);
multipleLayout.attach(graphModel);
try {
    multipleLayout.performLayout(true, true, true);
} catch (IlvGraphLayoutException e) {
    ...
}
multipleLayout.detach();
labelingModel.dispose();
graphModel.dispose();
```

Thus, the label layout does not provide a separate mechanism for a recursive layout on submanagers. By incorporating the label layout into a multiple graph layout, you can use all the graph layout facilities that are available for nested graphs (see also *Nested layouts*).



Nested subgraphs with labels

The label layout report

The label layout report contains information about the particular behavior of a label layout algorithm. After the layout is completed, this information is available for reading from the label layout report. The information can also be obtained during layout by using a layout listener, as described in *Layout events and listeners*.

The layout report is created automatically at the start of layout via the method `createLayoutReport` and is available as long as the grapher is attached to the layout instance.

To read a layout report, all you need to do is store the layout report instance returned by the `performLayout` method and read the information, as shown in the following example:

```
...
IlvLabelLayoutReport layoutReport = labelLayout.performLayout();
if (layoutReport.getCode() == IlvLabelLayoutReport.LAYOUT_DONE)
    System.out.println("Label layout done.");
else
    System.out.println("Label layout not done, code = " +
        layoutReport.getCode());
```

The class `IlvLabelLayoutReport` stores the following information, which is very similar to the information stored in an `IlvGraphLayoutReport` (see *Information stored in a layout report* for details):

Code

This field contains information about special, predefined cases that may have occurred during the layout. The possible values are the following:

- ◆ `IlvLabelLayoutReport.LAYOUT_DONE`
- ◆ `IlvLabelLayoutReport.STOPPED_AND_VALID`
- ◆ `IlvLabelLayoutReport.STOPPED_AND_INVALID`
- ◆ `IlvLabelLayoutReport.NOT_NEEDED`
- ◆ `IlvLabelLayoutReport.NO_LABELS`
- ◆ `IlvLabelLayoutReport.EXCEPTION_DURING_LAYOUT`

To read the code, use the method:

```
int getCode()
```

Layout time

This field contains the total duration of the layout algorithm at the end of the layout. To read the time (in milliseconds), use the method:

```
long getLayoutTime()
```

Percentage of completion

This field contains an estimate of the percentage of the layout that has been completed. To access the percentage, use the method:

```
int getPercentageComplete()
```

Layout events and listeners

The label layout framework provides the same event mechanism as the graph layout framework. Various events may occur.

Label layout events

The class `GraphLayoutEvent` corresponds to the class `GraphLayoutEvent` (see *Graph layout event listeners*). You can install a listener for these events at the layout instance by using the method:

```
labelLayout.addLabelLayoutEventListener(listener);
```

The listener must implement the `LabelLayoutEventListener` interface and receives events while the layout is running. A typical example is to check how much of the layout has already completed:

```
class MyLabelLayoutListener
    implements LabelLayoutEventListener
{
    public void layoutStepPerformed(LabelLayoutEvent event)
    {
        IlvLabelLayoutReport layoutReport = event.getLayoutReport();
        System.out.println("percentage of completion: " +
            layoutReport.getPercentageComplete());
    }
}
```

Label layout parameter events

The class `LabelLayoutParameterEvent` corresponds to the class `GraphLayoutParameterEvent` (see *Parameter event listeners*). You can install a listener to these events at the layout instance by

```
labelLayout.addLabelLayoutParameterEventListener(listener);
```

The listener must implement the `LabelLayoutParameterEventListener` interface and receives events when layout parameters change. It also receives a special event at the end of a successful layout. For example:

```
class MyLabelLayoutParameterListener
    implements LabelLayoutParameterEventListener
{
    public void parametersUpToDate(LabelLayoutParameterEvent event)
    {
        if (!event.isParametersUpToDate())
            System.out.println("Any label layout parameter has changed.");
    }
}
```



```
}  
}
```

Layout parameters and features in `IlvLabelLayout`

The class `IlvLabelLayout` defines a number of generic features and parameters. These are a subset of the mechanism, methods, and parameters that are available for the `IlvGraphModel` class. Therefore, they are only listed here; for detailed explanations, refer to the appropriate subsection in *Generic parameters and features* which describes the corresponding features for the `IlvGraphLayout` class.

Although the `IlvLabelLayout` class defines the generic parameters, it does not control how they are used by its subclasses. Each label layout algorithm (that is, each subclass of `IlvLabelLayout`) supports a subset of the generic features and determines the way in which it uses the generic parameters. When you create your own label layout algorithm by subclassing `IlvLabelLayout`, you decide whether to use the features and the way in which you are going to use them.

The `IlvLabelLayout` class defines the following generic features:

- ◆ *Allowed time*
- ◆ *Percentage of completion calculation*
- ◆ *Random generator seed value*
- ◆ *Stop immediately*
- ◆ *Use default parameters*

To specify that the label layout is allowed to run for 60 seconds:

In Java™

Call:

```
labelLayout.setAllowedTime(60000);
```

For more details of all generic features, see *Generic parameters and features*.

Releasing resources used during the layout of labels

Various objects need to be created during the layout process. Most commonly, these are:

- ◆ Layout instances (subclasses of `IlvLabelLayout`)
- ◆ Labeling models (subclasses of `LabelingModel`).
- ◆ Property objects

Some of the layout parameters are internally stored as property objects attached to the labeling model, to the manager, or to its labels and obstacles.

Methods for releasing resources

The class `IlvLabelLayout` provides four methods that help you release all these objects once they are no longer needed. Obsolete objects can be garbage -collected and memory leaks (in the Java™ sense) avoided:

```
void detach()
```

The method `cleanLabelingModel` releases property objects that the layout instances stored globally for the labeling model or for the grapher. The methods `cleanLabel` and `cleanObstacle` release property objects stored for the label or obstacle passed as an argument. These three methods are automatically called when the method `detach` is called.

Rules for releasing resources

The following rules must be respected:

1. When a layout object instantiated by your code is no longer useful, call the method `IlvLabelLayout.detach` on it to ensure that no grapher or labeling model is still attached to it. Note that you can freely reuse a layout instance once the previously attached model has been detached.
2. If labels or obstacles need to be removed while a layout instance is attached, the cleaning is done automatically.
3. When a labeling model instantiated by your code is no longer useful, call the method `dispose()` on it to ensure that the resources it has used are released. Note that a labeling model must not be used once it has been disposed of.

Summary

The following is a quick summary of what you need to do:

- ◆ When you attach a labeling model, you need to call `detach` on the layout, `dispose` on the labeling model.

Annealing label layout

Describes the *Annealing Label Layout* algorithm (class `IlvAnnealingLabelLayout` from the package `ilog.views.graphlayout.labellayout.annealing`).

In this section

General information

Gives samples of the Annealing Label Layout.

Features

Lists the features of the Annealing Label Layout.

Limitations

Lists the limitations of the Annealing Label Layout.

The algorithm

Describes the simulated annealing algorithm used by the Annealing Label Layout.

Generic features and parameters

Lists the generic features and parameters supported by the Annealing Label Layout.

Label descriptors

Describes the use of label descriptors to specify placement.

Point label descriptor

Describes the point label descriptor used by the Annealing Label Layout to place labels.

Polyline label descriptor

Describes the polyline label descriptor used by the Annealing Label Layout to place labels.

Specific global parameters

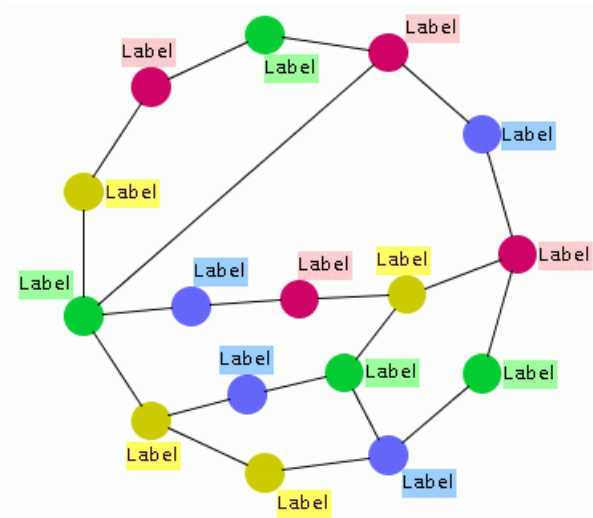
Describes the global parameters used by the Annealing Label Layout.

For experts: implementing your own label descriptors

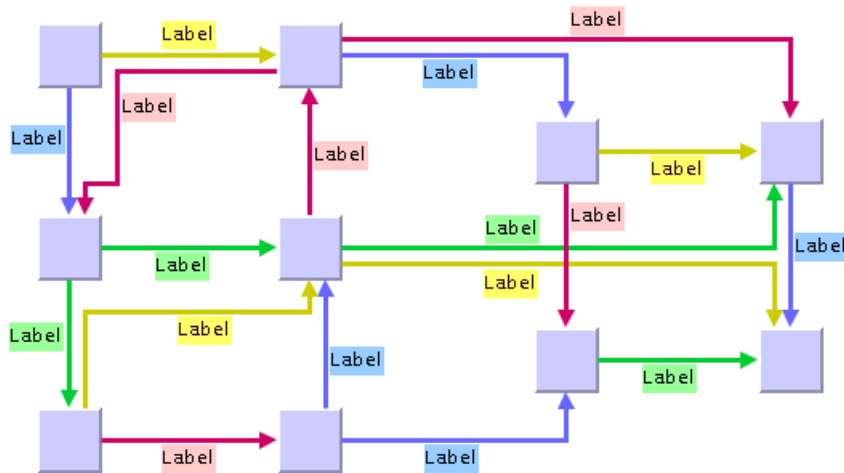
Describes how to create a label descriptor for the Annealing Label Layout.

General information

The following sample drawings are produced with the Annealing Label Layout.



Label placement at nodes with the Annealing Label Layout



Label placement at links with the Annealing Label Layout

Features

- ◆ Places only labels. Does not move any obstacles around.
- ◆ Quality-controlled, randomized iterative heuristic.
- ◆ Can place labels at points, rectangles, ellipses, and polylines.
- ◆ Can be used to place labels at any nodes and links.
- ◆ Can place multiple labels at the same object.
- ◆ Tries to avoid overlaps among labels, and between labels and obstacles, by using the available free space.
- ◆ Provides several anchor and preference options.
- ◆ Easily extensible via subclassing of label descriptors.
- ◆ Efficient, scalable algorithm. Produces nice label placements even with a large number of labels.

Limitations

- ◆ The Annealing Label Layout algorithm, as a randomized iterative heuristic, does not guarantee that labels are placed without overlaps whenever possible. However, it produces a high quality layout with a high probability of minimum overlap. The more iterations, the higher the probability of high quality.
- ◆ The algorithm is not able to create free space for labels by moving obstacles around. It is recommended that you perform a graph layout algorithm with large spacing parameters to create the necessary free space before placing the labels.
- ◆ While the algorithm is able to place labels at straight and polyline graphics, it is not able to place labels precisely at smooth curves such as spline graphics or spline links.

The algorithm

The algorithm uses *simulated annealing*. This is a general, randomized optimization technique that simulates a thermodynamic particle system. Each label is moved to a new random position within the limits given by its label descriptor. The quality of the new position is calculated and compared to the quality of the old position. If the quality has not improved, the label is moved back to the old position. The amount of movement is controlled by a conceptual temperature: when the system is hot, the labels can move long distances, producing potentially large global quality improvements. When the system cools down, the move distances become smaller and hence focus on local fine-tuning of the position.

Each label has its own label descriptor. The label descriptor describes the path on which the label can move. If a label must be placed at a specific point, the `IlvAnnealingPointLabelDescriptor` can be used and describes an approximately elliptical path around the point. If a label must be placed at a polyline, the `IlvAnnealingPolylineLabelDescriptor` can be used and describes a boundary path at both sides of the polyline.

Generic features and parameters

The `IlvAnnealingLabelLayout` class supports generic parameters defined in the `IlvLabelLayout` class. The following sections describe the particular way in which these parameters are used by the subclass `IlvAnnealingLabelLayout`.

- ◆ *Allowed time*
- ◆ *Percentage of completion calculation*
- ◆ *Random generator seed value*
- ◆ *Stop immediately*
- ◆ *Use default parameters*

Allowed time

The label layout algorithm stops if the allowed time setting has elapsed. This feature works similarly to the same feature in `IlvGraphLayout`; see *Allowed time*. If the layout stops early because the allowed time has elapsed, the result code in the layout report is:

- ◆ `IlvLabelLayoutReport.STOPPED_AND_VALID` if the labels were moved to some better (but not yet optimal) positions.
- ◆ `IlvLabelLayoutReport.STOPPED_AND_INVALID` if the time elapsed even before that.

Percentage of completion calculation

The label layout algorithm calculates the estimated percentage of completion. This value can be obtained from the label layout report during the run of the layout. (For a detailed description of this feature, see *Percentage of completion calculation* and *Layout events and listeners*.)

Random generator seed value

The Annealing Label Layout is a randomized heuristic. It uses a random number generator to control the movements. For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph. You can specify the particular value to be used as a seed value.

Example of specifying seed value

To specify the seed value 10:

In Java™

Call:

```
layout.setUseSeedValueForRandomGenerator(true);  
layout.setSeedValueForRandomGenerator(10);
```

Stop immediately

The label layout algorithm stops after cleanup if the method `stopImmediately` is called. This method works for the `IlvLabelLayout` class similarly to the corresponding method in the `IlvGraphLayout` class. For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*. If the layout stops early in this way, the result code in the layout report is:

- ◆ `IlvLabelLayoutReport.STOPPED_AND_VALID` if the labels were moved to some better (but not yet optimal) positions.
- ◆ `IlvLabelLayoutReport.STOPPED_AND_INVALID` if the layout stopped even before that.

Use default parameters

After modifying any label layout parameter, you may want the layout algorithm to use the default values. You select the default values for all global parameters by:

```
layout.setUseDefaultParameters(true);
```

IBM® ILOG® JViews Graph Layout for Eclipse keeps the previous settings when selecting the default values mode. You can switch back to your own settings by:

```
layout.setUseDefaultParameters(false);
```

This setting affects only the global layout parameters. The label descriptors have no default values, so parameters of the label descriptors do not change depending on this flag.

Label descriptors

To define where a label must be placed, you must specify a label descriptor for each label. The algorithm places only those labels that have a label descriptor.

A label descriptor describes the locations that are allowed for the label.

Subclasses of label descriptors

There are two predefined subclasses of label descriptors:

- ◆ *Point label descriptor*
- ◆ *Polyline label descriptor*

Depending on the parameters passed during the construction, these subclasses allow you to place a label:

- ◆ Close to a given point.
- ◆ Close to a specific rectangular or elliptic obstacle (such as a node).
- ◆ Along an imaginary polyline.
- ◆ Close to a polyline obstacle.
- ◆ Close to a link.

You can also implement your own label descriptors by subclassing `IlvAnnealingLabelDescriptor`. This is explained in the section *For experts: implementing your own label descriptors*.

Point label descriptor

The `IlvAnnealingPointLabelDescriptor` can be used to place a label at a specific obstacle or point. This is known as the *point labeling problem*.

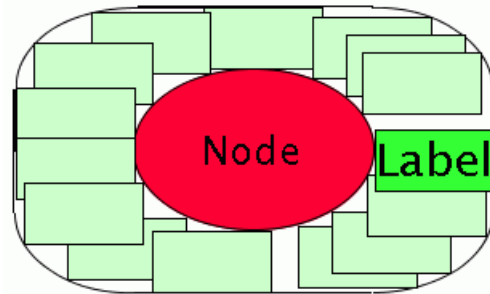
Warning: This kind of label descriptors does not work with GMF yet.

Positioning at an obstacle

The following example shows how to position a label at a specific obstacle in your `ILabelEditPart` implementation:

```
@Override
public IlvAnnealingLabelDescriptor createLabelDescriptor() {
    return new IlvAnnealingPointLabelDescriptor(this, relatedObstacleEditPart,
        IlvAnnealingPointLabelDescriptor.ELLIPTIC, IlvDirection.Right);
}
```

This specification can be used if the label must be placed at a node that has an elliptical or circular shape. The label is placed in the free area around the node so that the border of the label just touches the border of the node (see the following figure). The preferred position is the right side of the node, but this preferred position is used only if it does not create overlaps. If the node is moved or reshaped, the next call of label layout will update the position of the label automatically so that it follows the node.



Potential label positions around a node

The example uses the following constructor:

```
IlvAnnealingPointLabelDescriptor(Object label,
                                Object relatedObstacle,
                                int shape,
                                int preferredDirection)
```

This constructor takes the following parameters:

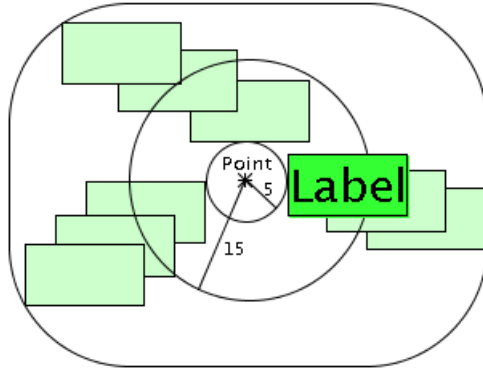
- ◆ The `relatedObstacle` parameter is the obstacle that gets the label. The label is placed outside but close to this obstacle. The related obstacle is typically a node of a graph. The shape of the related obstacle should be either an ellipse, a circle, or a rectangle.
- ◆ The `shape` argument can take the following values:
 - `IlvAnnealingPointLabelDescriptor.ELLIPTIC` for ellipses or circles,
 - `IlvAnnealingPointLabelDescriptor.RECTANGULAR` for rectangles.If the real shape of the related obstacle is neither of these, pass the shape that is the best approximation. For instance, if the obstacle is an `IlvRoundRectangle`, it can be considered as a rectangular shape and the `RECTANGULAR` option is then the best approximation.
- ◆ The `preferredDirection` parameter is a suggestion of where the label layout algorithm should preferably place the label. If the area at the preferred position is occupied, the label will be placed elsewhere. Options for the preferred position are:
 - `IlvDirection.Left`
 - `IlvDirection.Right`
 - `IlvDirection.Top`
 - `IlvDirection.Bottom`

Positioning at a point

The following example shows how to position a label at a specific point in your `ILabelEditPart` implementation:

```
@Override
public IlvAnnealingLabelDescriptor createLabelDescriptor() {
    new IlvAnnealingPointLabelDescriptor(this, null, new IlvPoint(25, 75), 5f,
    15f,
    IlvDirection.Right);
}
```

Use this specification if the label must be placed close to specific coordinates, like (in this example 25, 75) regardless of any obstacle. The label must be at least 5 coordinate units and at most 15 coordinate units away from the point (see the following figure). The preferred position is at the right side of the point.



Potential label positions between 5 and 15 units away from a point

The example uses the following constructor:

```
IlvAnnealingPointLabelDescriptor(Object label,
                                Object relatedObstacle,
                                IlvPoint referencePoint,
                                float minDist,
                                float maxDist,
                                int preferredDirection)
```

This `IlvAnnealingPointLabelDescriptor(java.lang.Object, java.lang.Object, ilog.views.IlvPoint, float, float, int)` constructor takes the following parameters:

- ◆ `relatedObstacle` and `referencePoint`: The label is placed close to the reference point. It does not take the actual position of the related obstacle into account. If the related obstacle is moved, the label does not follow the obstacle on the next call of layout, but stays at the reference point.

If a related obstacle is given, the label is not pushed away from the related obstacle. Rather, it is pushed away from all other obstacles to avoid overlaps. You can set the `relatedObstacle` parameter to `null` if the label is independent of all obstacles.

- ◆ The parameters `minDist` and `maxDist` are the minimal and maximal distances from the reference point, measured from the border of the label. If you set the minimal and maximal distance to 0, the label will just touch the reference point. To keep the circular area around the reference point free, set the minimal distance accordingly. Most of the time you probably want to keep the label close to the reference point; hence, set the minimal and maximal distances to the same value.
- ◆ The `preferredDirection` parameter indicates whether the label should be placed to the left, right, top, or bottom of the reference point. This is a suggestion for the labeling algorithm, as described for *Positioning at an obstacle*.

Positioning on multiple criteria

The most powerful constructor combines all the possibilities described in *Positioning at an obstacle* and *Positioning at a point*:


```

IlvAnnealingPointLabelDescriptor(Object label,
                                Object relatedObstacle,
                                IlvPoint referencePoint,
                                int shape,
                                float halfWidth,
                                float halfHeight,
                                float maxDistFromPath,
                                float preferredDistFromPath,
                                int preferredDirection)

```

This `IlvAnnealingPointLabelDescriptor` (`java.lang.Object`, `java.lang.Object`, `ilog.views.IlvPoint`, `int`, `float`, `float`, `float`, `float`, `float`, `int`) constructor takes the following parameters:

- ◆ `relatedObstacle` and `referencePoint`: If a related obstacle is given and the reference point is `null`, the label is placed close to the related obstacle. If a reference point is not `null`, the label is placed close to the reference point independently of the related obstacle position.
- ◆ `shape` : the shape of the free area around the point can be rectangular or elliptic.
- ◆ `halfWidth` and `halfHeight`: The parameter `halfWidth` is the minimal distance of the label to the reference point in the horizontal direction. The parameter `halfHeight` is the minimal distance of the label to the reference point in the vertical direction. If the reference point is `null`, the parameters `halfWidth` and `halfHeight` are calculated from the bounding box of the related obstacle.
- ◆ The parameter `maxDistFromPath` specifies the maximal additional distance allowed for the label (shown in *Potential Label Positions With Rectangular Shape at a Point*).
- ◆ The parameter `preferredDistFromPath` specifies the preferred additional distance for the label. Its value should be between 0 and `maxDistFromPath`.
- ◆ The `preferredDirection` parameter indicates whether the label should be placed to the left, right, top, or bottom of the reference point or related obstacle. This is a suggestion for the labeling algorithm, as described in *Positioning at an obstacle*.

Starting from an empty descriptor (point)

An alternative way to create a point label descriptor is to start from the empty descriptor:

```

descriptor = new IlvAnnealingPointLabelDescriptor();

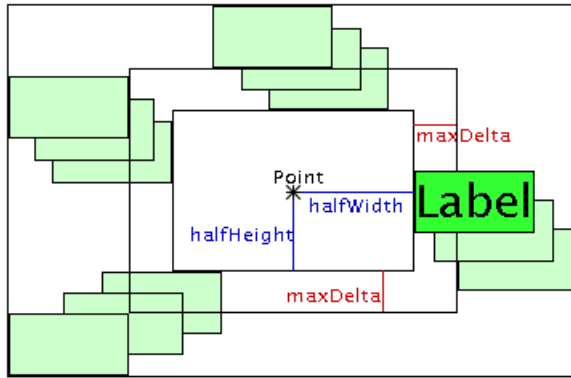
```

Before using the empty descriptor, you must fill it with information on how the label should be placed. As a minimum, you need to specify a related obstacle or a reference point. For example:

```

descriptor.setRelatedObstacle(obstacle);
descriptor.setShape(IlvAnnealingPointLabelDescriptor.ELLIPTIC);
descriptor.setPreferredDirection(IlvDirection.Left);

```



Potential Label Positions With Rectangular Shape at a Point

Polyline label descriptor

If you want to place labels at straight lines, polylines, or links, you should use the class `IlvAnnealingPolylineLabelDescriptor`. The allowed area for labels at a polyline is different from the rectangular or elliptical area considered for placing labels at a reference point (see *Positioning at a point*). A polyline has two sides where the label can be placed along a path. This is known as the *polyline labeling problem*.

Note: The polyline label descriptor is not suitable for placing labels at splines or spline links. Because splines have a complex geometric shape, the automatic placement of labels at splines is currently not supported.

Simple positioning at a polyline obstacle

Use this specification if the label must be placed at a polyline obstacle. The polyline obstacle is typically a link.

Here is an example of `ILabelEditPart createLabelDescriptor()` implementation:

```
@Override
public IlvAnnealingLabelDescriptor createLabelDescriptor() {
    return new IlvAnnealingPolylineLabelDescriptor(this,
        relatedObstacleEditPart,
        IlvAnnealingPolylineLabelDescriptor.FREE, IlvDirection.Left, IlvDirection.
        TopLeft,
        IlvAnnealingPolylineLabelDescriptor.GLOBAL);
}
```

The label is placed anywhere at the left or top side of the polyline obstacle, with preference given to the left side.

The example uses the following constructor:

```
IlvAnnealingPolylineLabelDescriptor
    (Object label,
     Object relatedPolylineObstacle,
     int anchor,
     int preferredSide,
     int allowedSide,
     int sideAssociation)
```

The options for the anchor parameter are:

- ◆ `IlvAnnealingPolylineLabelDescriptor.CENTER`: places the label near the center of the link (that is, in the middle third of the link path).
- ◆ `IlvAnnealingPolylineLabelDescriptor.START`: places the label near the source node of the link (that is, in the first third of the link path).

- ◆ `IlvAnnealingPolylineLabelDescriptor.END`: places the label near the target node of the link (that is, in the last third of the link path).
- ◆ `IlvAnnealingPolylineLabelDescriptor.FREE`: places the label anywhere on the link.

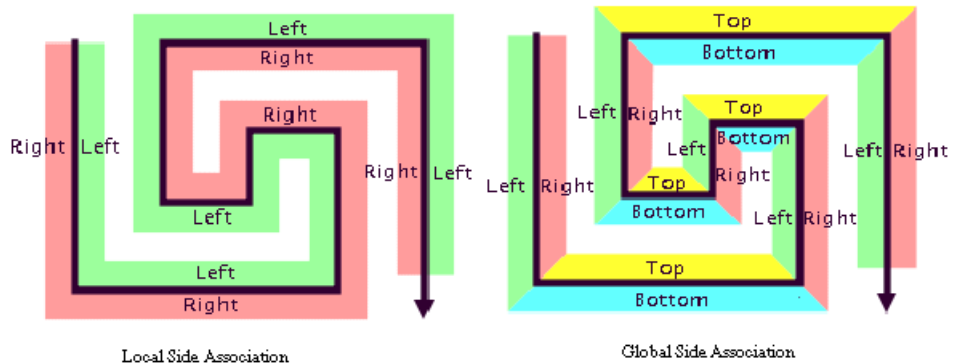
The value of the `preferredSide` parameter is a suggestion of where the label layout algorithm should preferably place the label. If the area at the preferred side is occupied, the label is placed elsewhere.

In contrast, the `allowedSide` parameter is a strict constraint: it is always obeyed, even if the entire area at the allowed side is occupied and the label must overlap the obstacles in that area.

Side association

The orientation of the preferred and allowed sides depend on the `sideAssociation` parameter. This parameter can take the following values (see the following figure):

- ◆ `IlvAnnealingPolylineLabelDescriptor.LOCAL`
- ◆ `IlvAnnealingPolylineLabelDescriptor.GLOBAL`



Side Associations

Local side association

If the side association is local, each polyline has two sides: left and right. The sides can be determined from the flow direction of the polyline from start point to end point. Consider yourself standing on the polyline looking in the direction where the polyline continues towards the end point, and then determine which is the left and which is the right side. Hence, the meaning of left and right in local side association is relative to the polyline. The options for the `preferredSide` and `allowedSide` parameters are in this case:

- ◆ `IlvDirection.Left`
- ◆ `IlvDirection.Right`

You can also specify the value 0 for the allowed side, which indicates that you do not want to restrict the side: both sides are allowed.

Global side association

If the side association is global, the side specification is independent of the flow direction of the polyline and more like a compass direction: north is top, south is bottom, west is left, and east is right. Here more options are possible: in addition to the basic top, bottom, left, right, all meaningful combinations of these are allowed. You specify the sides in the following way:

- ◆ `IlvDirection.Left`
- ◆ `IlvDirection.Right`
- ◆ `IlvDirection.Top`
- ◆ `IlvDirection.Bottom`

You can also use combinations of these, such as:

- ◆ `IlvDirection.Left | IlvDirection.Right` (left or right but not top or bottom).
- ◆ `IlvDirection.Left | IlvDirection.Top` (which is the same as `IlvDirection.TopLeft`, meaning the left or the top side).

You can specify the value 0 for the allowed side if all sides should be allowed.

Full positioning at a polyline obstacle

The most powerful `IlvAnnealingPolylineLabelDescriptor` constructor is the following:

```
IlvAnnealingPolylineLabelDescriptor
    (Object label,
     Object relatedObstacle,
     IlvPoint[] referencePoints,
     float lineWidth,
     float minPercentageFromStart,
     float maxPercentageFromStart,
     float prefPercentageFromStart,
     float maxDistFromPath,
     float preferredDistFromPath,
     int preferredSide,
     int allowedSide,
     int sideAssociation,
     float topOverlap,
     float bottomOverlap,
     float leftOverlap,
     float rightOverlap)
```

It combines all previously mentioned possibilities. If the label should be placed at a polyline obstacle, then pass this object as the related obstacle. If the label should be placed at an imaginary polyline, then pass the polyline with the `points` parameter and the width of the polyline with the `lineWidth` parameter. Instead of an anchor, you can pass the area where the label is placed by the minimal, maximal, and preferred percentage values relative to the polyline length. The minimal and maximal percentages are strictly obeyed, while the preferred percentage is only a recommendation for the layout.

- ◆ For instance, if you want to specify that the label can be placed anywhere but you prefer the center of the polyline, specify 0 and 100 for the minimal and maximal percentages and 50 for the preferred percentage. If there is not enough free space at the center, the label will be placed elsewhere.
- ◆ But if you want to specify that the label must be placed close to the center even if there is not enough space, then specify, for instance, 40 and 60 for the minimal and maximal percentages instead.

Starting from an empty descriptor (polyline)

An alternative way to create a polyline label descriptor is to start with the empty descriptor:

```
descriptor = new IlvAnnealingPolylineLabelDescriptor();
```

Before using the empty descriptor, you must fill it with information on how the label should be placed. As a minimum, you need to specify a related obstacle or reference points and line width. For instance:

```
descriptor.setRelatedObstacle(polyline);  
descriptor.setMinPercentageFromStart(10f);  
descriptor.setMaxPercentageFromStart(90f);  
descriptor.setPreferredPercentageFromStart(50f);
```

Specific global parameters

The following global parameters are specific to the `IlvAnnealingLabelLayout` class:

- ◆ *Label offset*
- ◆ *Obstacle offset*
- ◆ *Label movement policy*
- ◆ *Automatic update*
- ◆ *Expert parameters*

Label offset

The label offset controls the desired minimal distance between two neighbored labels (see *Label and Obstacle Offsets*, left). To avoid labels being placed too close to each other, you can increase the label offset. This conceptually pushes the labels farther apart. However, depending on the available space, the minimal distance between labels cannot always be maintained.

Example of specifying label offset

To set the label offset:

In Java™

Call:

```
layout.setLabelOffset(25f);
```

Obstacle offset

The obstacle offset controls the desired minimal distance between a label and an unrelated obstacle. The obstacle offset is usually more important than the label offset because, if the obstacle offset is too small, the label may be placed so close to an unrelated obstacle that it incorrectly appears to be assigned to that obstacle (see *Label and Obstacle Offsets*, right: does, for example, the green label belong to the upper yellow node or to the green node?). Increasing the obstacle offset conceptually pushes the label away from the obstacle. However, depending on the available space, the minimal distance between label and obstacle cannot always be maintained.

The obstacle offset should not be set to an unreasonably large value (such as `Float.MAX_VALUE`) because this can cause computational problems.

Example of specifying node placement iterations and allowed time (GL algorithm)

To set the obstacle offset:

In Java

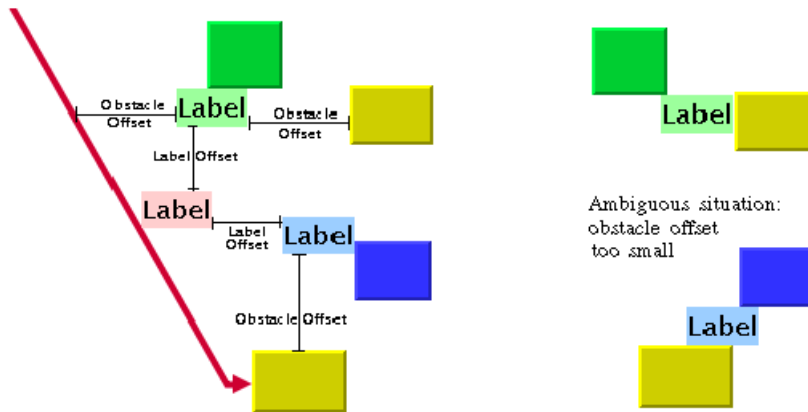
Call:

```
layout.setObstacleOffset(25f);
```

The specified obstacle offset works globally for all labels.

It is also possible to specify a smaller obstacle offset for specific label/obstacle pairs. You need to install an obstacle offset interface that returns the obstacle offset for a given pair.

The effective offset is the lower of the values returned by the obstacle offset interface and the globally specified offset respectively.



Label and Obstacle Offsets

Label movement policy

The label movement policy is an easy way to define which labels should be moved by the label layout algorithm.

Example of specifying label movement policy

In Java

The following code installs a label movement policy such that the layout moves only labels with a height greater than 100:

```
layout.setLabelMovementPolicy(new IlvLabelMovementPolicy() {  
    public boolean allowMove(IlvLabelingModel labelingModel, Object label)  
    {  
        return (labelingModel.boundingBox(label).height > 100);  
    }  
});
```

Labels with smaller heights are not moved. However, they are also not completely ignored, because the layout tries to position the movable labels so that they do not overlap the immovable labels, and the label offset is respected between movable and immovable labels.

A more general useful example is a movement policy that prohibits moving labels that initially do not overlap anything. This predefined movement policy is available through the class `IlvOverlappingLabelMovementPolicy`. You can use this class in applications that have their own label positioning mechanism and use the `Annealing Label` layout only as a postprocessing step to improve the positions of overlapping labels. To install this policy, call:


```
layout.setLabelMovementPolicy(new IlvOverlappingLabelMovementPolicy());
```

Automatic update

After layout, the labels are placed close to the related obstacle according to the label descriptor. For instance, a link label is placed close to its link. However, if you move the link interactively, the label normally stays at the old position, which may be far away from the link after the movement. The label loses the connection to the link, and a new layout is necessary.

Because it is too time-consuming to redo the layout after each single interactive move, the Annealing Layout algorithm has a feature that automatically updates the labels on geometric changes, that is, the label follows the link when the link moves.

Example of specifying automatic update

To enable this feature:

In Java

Call:

```
layout.setAutoUpdate(true);
```

If automatic update is enabled, the algorithm does not perform a full layout of all labels during each interactive change. It repositions only the label whose related obstacle has moved in one step. Thus it may produce more overlaps than a full layout. The automatic update mechanism is much faster, however, and hence better suitable for interactions.

Expert parameters

A few parameters are available for an advanced use of the Annealing Label Layout.

Simulated annealing

Simulated annealing is an iterative mechanism. In each iteration step, all labels are tested for better positions. Usually, the algorithm is capable of detecting automatically when to stop. The algorithm stops if:

- ◆ The maximal number of iterations is reached.
- ◆ After several iterations, no better position was found for any label.
- ◆ After several iterations, the quality did not improve by a given percentage.

In a few cases, it may be necessary to limit the number of iterations, which can be done by calling:

```
layout.setAllowedNumberOfIterations(100);
```

As a general hint, to obtain a reasonable layout, the allowed number of iterations should not be considerably lower than the number of labels.

Simulated Annealing stops if, after several iterations, no better position was found for any label. Because the search is randomized, this does not necessarily mean that the best position

was already found; however, it indicates that finding the best position would require too much layout time. The number of ineffective iterations before stopping can be changed by calling:

```
layout.setMaxNumberOfFailIterations(maxNumber);
```

The default value is 20. If you set it to a higher value, the layout slows down but may find better positions for the labels. If you set it to a lower value, the layout stops sooner, but the label positions may be far from optimal.

In some cases, the algorithm improves the quality in each step, but the amount of improvement gets smaller in each step. In this situation, the previous fail-iteration criteria does not work well because there is an improvement in each step, but the amount of the improvement is so negligibly small that we want to stop. Therefore, it is also possible to require that the quality must improve in each step by a minimum percentage.

For example, to specify that the algorithm must improve over ten rounds by at least 2%, call:

```
layout.setNumberIterationsForMinImprovement(10);  
layout.setMinImprovementPercentageToContinue(2);
```

By default, the layout stops if the quality did not improve by 0.5% over five iterations. If you set the required improvement percentage higher or the number of iterations lower, the layout stops sooner, but the label positions may be far from optimal. If you set the required percentage to 0%, this stop criterion is disabled and will no longer have any effect.

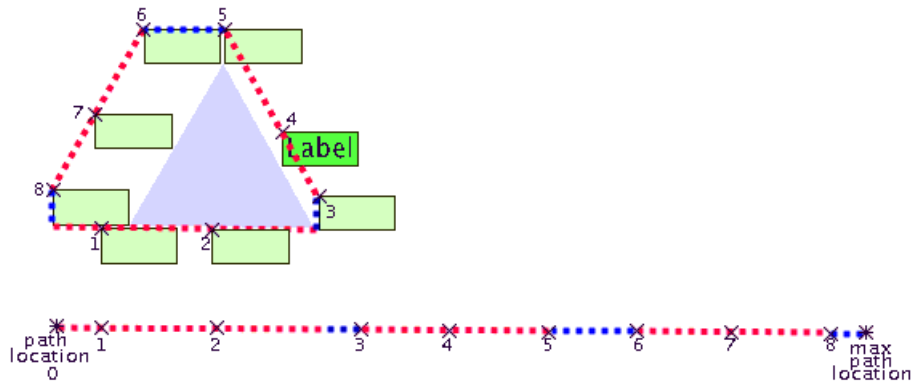
For experts: implementing your own label descriptors

The Annealing Label Layout is extensible. The point label descriptor and the polyline label descriptor are designed to cover the majority of the cases. In rare situations, you may want to implement your own label descriptor by subclassing the base class `IlvAnnealingLabelDescriptor`. This section describes the necessary steps.

A label descriptor basically specifies the path where the top-left corner of a label can be placed. For simplification, it considers the path rolled out such that the path has only one dimension. If the path is known, the precise label position can be specified by just one value: the path location. The Annealing Label Layout proposes different path locations during the layout; however, it does not know what the path looks like. The task of the label descriptor is to translate the path location into concrete (x, y) coordinates of the label.

As an example, we want to create a label descriptor that can place labels precisely at a triangular obstacle. We could use the point label descriptor as an approximation, but it does not place the labels precisely at a triangular shape.

In the following figure, the upper diagram shows the path around the triangle (the dashed red and blue line). Below, you can see the same path rolled out in one dimension. The Annealing Label Layout may ask the label descriptor to place the label at position 1 to 8. For the Annealing Label Layout, these positions are just numbers between 0 and `maxPathLocation`. The task of the label descriptor is to translate these numbers into the correct positions as shown in the upper part of the figure.



Path locations at a triangle label descriptor

The base class `IlvAnnealingLabelDescriptor` has two protected data members:

- ◆ `actPathLocation` is the current path location of the label.
- ◆ `maxPathLocation` is the maximal value of the path location.

To create a new label descriptor, you need to implement a method that initializes the path constructor at the beginning of layout. You should calculate the maximal path location `maxPathLocation` and initialize the `actPathLocation` here. The method is called only once during layout:

```
void initialize(IlvLabelingModel labelingModel)
```

In the previous figure, the maximal path location for an equilateral triangle is:

```
3 * sidelength + 2 * labelwidth + 2 * labelheight
```

At each iteration step, the layout calls the method `setPosition` and provides an actual value for the path location. The method `setPosition` should store the value into `actPathLocation` and translate the path location into appropriate (x, y) coordinates. Then it should call the predefined method `updatePosition(x, y)` with these coordinates:

```
public void setPosition(double pathLocation, float distFromPath)
{
    float x, y;
    // make sure the position is between 0 and max
    while (pathLocation > maxPathLocation)
        pathLocation -= maxPathLocation;
    while (pathLocation < 0)
        pathLocation += maxPathLocation;
    // store the actual position
    actPathLocation = pathLocation;
    // translate the path location into (x, y)
    if (pathLocation < labelwidth + sidelength) {
        x = (float)pathLocation;
        y = triangleBottom;
    } else if (pathLocation < labelwidth + labelheight + sidelength) {
        x = labelwidth + sidelength;
        y = triangleBottom - (float)pathLocation + labelwidth + sidelength;
    } else if ... (other cases) ...
        ...
    // finally, update the internal data structures
    updatePosition(x, y);
}
```

The label may have a preferred position at the triangle. The Annealing Layout checks a location close to the preferred position from time to time. You should implement the following method to return the preferred path location:

```
double getPreferredPathLocation()
```

Furthermore, you should implement a strategy on how to come close to the preferred location. Towards the end of layout, the algorithm calls the method:

```
setTowardsPreferredPosition(pathLocation, dist, i, maxI)
```

to perform a sequence of steps that shift the label from the current position closer to the preferred position.

with i from 1 to maxI . Implement the method so that at each step you calculate a path location closer to the preferred location. When i is maxI , it should be exactly at the preferred

location. You can call `setPosition` to move the label to the preferred (x, y) position. For instance:

```
public void setTowardsPreferredPosition(
    double pathLocation, float dist, int i, int maxI)
{
    double offset = pathLocation - getPreferredPathLocation();
    double newLocation = pathLocation - i * offset / maxI;
    setPosition(newLocation, dist);
}
```

These methods take the `distance` parameter in addition to the path location. This is the distance from the path. If the label must always be on the path, you can assume this distance is 0. Set it to a different value only if your label descriptor allows the label to have a variable offset from the path.

Using advanced features

Describes advanced features including how to define new types of layouts.

In this section

Overview of advanced features

Explains the purpose of the advanced features.

Using a graph layout report

Describes what graph layout reports are and how to use them.

Using event listeners

Describes the listeners for different kinds of events.

Laying out connected components of a disconnected graph

Explains how to use graph layout when you have a disconnected graph.

Defining your own type of layout

Describes how to develop a custom graph layout algorithm if you need one.

FAQs about using the layout algorithms

Lists some FAQs about the use of the layout algorithms.

Overview of advanced features

These advanced features give you a powerful way to adapt or extend graph layouts.

Using a graph layout report

Describes what graph layout reports are and how to use them.

In this section

Layout report classes

Lists the layout classes and corresponding layout report classes.

Creating a layout report

Explains how to create a layout report.

Accessing a layout report

Explains how to access a layout report.

Information stored in a layout report

Lists the fields in a layout report.

Layout report classes

Graph layout reports are objects used to store information about the particular behavior of a layout algorithm. After the layout is completed, this information is available to be read from the layout report.

Each layout class instantiates a particular class of `ilog.views.graphlayout.IlvGraphLayoutReport` each time the layout is performed. The following table shows the layout classes and their corresponding layout reports.

Layout report classes

| Layout Class | Layout Report Class |
|--|--|
| <code>IlvTopologicalMeshLayout</code> | <code>IlvTopologicalMeshLayoutReport</code> |
| <code>IlvUniformLengthEdgesLayout</code> | <code>IlvUniformLengthEdgesLayoutReport</code> |
| <code>IlvTreeLayout</code> | <code>IlvGraphLayoutReport</code> |
| <code>IlvHierarchicalLayout</code> | <code>IlvGraphLayoutReport</code> |
| <code>IlvLinkLayout</code> | <code>IlvGraphLayoutReport</code> |
| <code>IlvRandomLayout</code> | <code>IlvGraphLayoutReport</code> |
| <code>IlvBusLayout</code> | <code>IlvGraphLayoutReport</code> |
| <code>IlvCircularLayout</code> | <code>IlvGraphLayoutReport</code> |
| <code>IlvGridLayout</code> | <code>IlvGraphLayoutReport</code> |
| <code>IlvMultipleLayout</code> | <code>IlvMultipleLayoutReport</code> |
| <code>IlvRecursiveLayout</code> | <code>IlvRecursiveLayoutReport</code> |

Creating a layout report

All layout classes inherit the `performLayout` method from the `IlvGraphLayout` class. This method calls `createLayoutReport` to obtain a new instance of the layout report. This instance is returned when `performLayout` returns. The default implementation in the base layout class creates an instance of `IlvGraphLayoutReport`. Some subclasses override this method to return an appropriate subclass. Other classes, such as `IlvRandomLayout`, do not need specific information to be stored in the layout report and do not override `createLayoutReport`. In this case, the base class `IlvGraphLayoutReport` is used.

When using the layout classes with IBM® ILOG® JViews Graph Layout for Eclipse, you do not need to instantiate the layout report yourself. This is done automatically.

Accessing a layout report

The method `performLayout` returns the layout report. The following example shows how to read the information from the layout report in this case:

```
...
try {
    IlvGraphLayoutReport layoutReport = layout.performLayout();
    if (layoutReport.getCode() ==
        IlvGraphLayoutReport.LAYOUT_DONE)
        System.out.println("Layout done.");
    else
        System.out.println("Layout not done, code = " +
            layoutReport.getCode());
}
catch (IlvGraphLayoutException e) {
    System.err.println(e.getMessage());
}
```

Information stored in a layout report

The base class `IlvGraphLayoutReport` stores the following information:

- ◆ *Code*
- ◆ *Layout time*
- ◆ *Percentage of completion*
- ◆ *Additional information*

Code

This field contains information about special, predefined cases that may have occurred during the layout. The possible values are the following:

- ◆ `LAYOUT_DONE` appears if the layout was performed successfully.
- ◆ `STOPPED_AND_VALID` appears if the layout was performed but was stopped before completion, either because the layout time elapsed or because the method `stopImmediately` was called. The positions of nodes and links are valid at the stopping point because the layout algorithm uses an iterative mechanism.
- ◆ `STOPPED_AND_INVALID` appears if a (noniterative) layout was performed but was stopped before completion, either because the layout time elapsed or because the method `stopImmediately` was called. The positions of nodes and links are not valid at the stopping point. Often, they have not yet been changed at all.
- ◆ `NOT_NEEDED` appears if the layout was not performed because no changes occurred in the grapher and parameters since the last time the layout was performed successfully.
- ◆ `EMPTY_GRAPHER` appears if the grapher is empty.

To read the code, use the method:

```
int getCode()
```

Layout time

This field contains the total duration of the layout algorithm at the end of the layout. To read the time (in milliseconds), use the method:

```
long getLayoutTime()
```

Percentage of completion

This field contains an estimation of the percentage of the layout that has been completed. This can be used if the layout algorithm supports the generic percentage completion calculation feature. (See *Percentage of completion calculation*.) It is typically used inside

layout event listeners that are described in the following section. To access the percentage, use the method:

```
int getPercentageComplete()
```

Additional information

Additional information for particular layout algorithms is stored by the subclasses of `IlvGraphLayoutReport`. For details, see the reference documentation of these classes:

- ◆ `IlvTopologicalMeshLayoutReport`
- ◆ `IlvUniformLengthEdgesLayoutReport`
- ◆ `IlvMultipleLayoutReport`
- ◆ `IlvRecursiveLayoutReport`

Using event listeners

All layout classes support two kinds of events: layout events and parameter events. The listening mechanism therefore provides:

- ◆ *Graph layout event listeners*
- ◆ *Parameter event listeners*

Graph layout event listeners

The layout event listening mechanism provides a way to inform the end user of what is happening during the layout. At times, a layout algorithm may take a long time to execute, especially when dealing with large graphs. In addition, an algorithm may not converge in some cases. No matter what the situation, the end user should be informed of the events that occur during the layout. This can be done by implementing a simple progress bar or by displaying appropriate information, such as the percentage of completion after each iteration or step.

The layout event listener is defined by the `GraphLayoutEventListener` interface. To receive the layout events delivered during the layout, a class must implement the `GraphLayoutEventListener` interface and must register itself using the `addGraphLayoutEventListener` method of the `IlvGraphLayout` class.

When you implement the `GraphLayoutEventListener` interface, you must implement the `layoutStepPerformed` method. The layout algorithm will call this method on all the registered layout event listeners, passing the layout report as an argument (see *Using a graph layout report*). In this way, you can read information about the current state of the layout report. (For example, you can read this information after each iteration or step of the layout algorithm).

The following example shows how to implement a layout event listener:

```
class LayoutIterationListener
    implements GraphLayoutEventListener
{
    public void layoutStepPerformed(GraphLayoutEvent event)
    {
        IlvGraphLayoutReport layoutReport = event.getLayoutReport();
        System.out.println("percentage of completion: " +
            layoutReport.getPercentageComplete());
    }
}
```

Then, register the listener on the layout instance as follows:

```
layout.addGraphLayoutEventListener(new LayoutIterationListener());
```

Parameter event listeners

The layout parameter event listeners mechanism provides a way to inform the end user that any layout parameter has changed. This is useful when the layout parameter values are displayed in a dialog box that needs to be updated to indicate parameter changes.

The parameter event listener is defined by the `GraphLayoutParameterEventListener` interface. To receive the layout parameter events, a class must implement the `GraphLayoutParameterEventListener` interface and must register itself using the `addGraphLayoutParameterEventListener` method of the `IlvGraphLayout` class.

When you implement the `GraphLayoutParameterEventListener` interface, you must implement the `parametersUpToDate` method. The layout class will call this method on all the registered layout parameter event listeners. The layout parameter event contains a flag accessible by the `isParametersUpToDate` method:

- ◆ It returns `true` if the event occurs at the end of a run of the layout when the layout is considered up-to-date with respect to the layout parameters.
- ◆ It returns `false` if the event occurs when any layout parameter has changed.

The following example shows how to implement a layout parameter event listener.

```
class LayoutParameterListener
    implements GraphLayoutParameterEventListener
{
    public void parametersUpToDate(GraphLayoutParameterEvent event)
    {
        if (!event.isParametersUpToDate())
            System.out.println("Any layout parameter has changed.");
    }
}
```

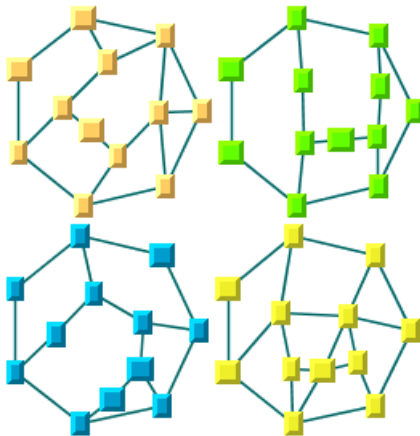
Then, register the listener with the layout instance as follows:

```
layout.addGraphLayoutParameterEventListener(new LayoutParameterListener());
```

Laying out connected components of a disconnected graph

IBM® ILOG® JViews Graph Layout for Eclipse provides special support for the layout of a disconnected graph.

If a graph is composed of several connected components or contains isolated nodes (nodes without any links), it can be desirable to apply the layout algorithm separately on each connected component and then to position the connected components using a specialized layout algorithm (usually, `IlvGridLayout`). The following figure shows an example of a graph containing four connected components. Simply by enabling the layout of the connected components on the regular layout instance (here, `IlvTopologicalMeshLayout`), the connected components are automatically identified and laid out individually. Finally, the four connected components are positioned using a highly customizable placement algorithm (`IlvGridLayout`).



Automatic layout of connected components in a disconnected graph

To indicate whether a subclass of `IlvGraphLayout` supports this feature, use the method in the class `IlvGraphLayout`:

```
boolean supportsLayoutOfConnectedComponents()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this feature is supported.

IBM® ILOG® JViews Graph Layout for Eclipse allows you to enable the layout of the connected components using the method:

```
void setLayoutOfConnectedComponentsEnabled(boolean enable)
```

To obtain the current setting:

```
boolean isLayoutOfConnectedComponentsEnabled()
```

The default value is the value returned by the following method:

```
boolean isLayoutOfConnectedComponentsEnabledByDefault()
```

The default implementation of this method in `IlvGraphLayout` returns `false`. For some of the layout classes, it is appropriate that this feature is enabled by default. Therefore `IlvUniformLengthEdgesLayout` overrides this method to return `true`.

If enabled on a layout class that supports this feature, the method `performLayout` of the class `IlvGraphLayout` cuts the attached graph model into connected components and lays out each connected component separately.

How does the layout of connected components feature work when this mechanism is enabled in the layout classes that support this feature? Instead of directly calling the method `layout(boolean)` to perform the layout on the entire graph, the method `performLayout(boolean, boolean)` first cuts the graph into connected components. Then, each connected component is laid out separately by a call of the method `layout`. To do this, the attached graph is temporarily changed into internally generated graphs corresponding to each of the connected components of the original graph. Finally, the layout instance returned by the method:

```
IlvGraphLayout getLayoutOfConnectedComponents()
```

is used to position the connected components. To specify the layout instance that places the connected components, use the following method:

```
void setLayoutOfConnectedComponents(IlvGraphLayout layout)
```

If no layout instance is specified using this method, the method `getLayoutOfConnectedComponents` returns an instance of `IlvGridLayout`. Its layout region parameter is set by default to the rectangle (0, 0, 800, 800). Its “layout mode” parameter is set to `TILE_TO_ROWS`.

Note: The Tree, Hierarchical, and Circular layouts contain built-in support for disconnected graphs. For the Tree and Hierarchical layouts, the result can be different from the result of the generic mechanism (the layout of connected components feature) provided by the base class `IlvGraphLayout`. Depending your particular needs, you can use either the generic mechanism or the built-in support.

Defining your own type of layout

Describes how to develop a custom graph layout algorithm if you need one.

In this section

A sample custom layout algorithm

Describes the features of a custom layout algorithm and shows an example.

Implementing the layout method

Explains how to implement a layout method.

A sample custom layout algorithm

If the layout algorithms provided with IBM® ILOG® JViews Graph Layout for Eclipse do not meet your needs, you can develop your own layout algorithms by subclassing `IlvGraphLayout`.

When a subclass of `IlvGraphLayout` is created, it automatically fits into the generic IBM® ILOG® JViews Graph Layout for Eclipse layout framework and benefits from its infrastructure:

- ◆ generic parameters: see *Base class parameters and features*
- ◆ notification of progress: see *Using event listeners*
- ◆ capability to apply the layout separately for the connected components of a disconnected graph: see *Laying out connected components of a disconnected graph*
- ◆ capability to lay out nested graphs (see *Nested layouts*), and so on.

Example

To illustrate the basic ideas for defining a new layout, the following simple example shows a possible implementation of the simplest layout algorithm, the Random Layout. The new layout class is called `MyRandomLayout`.

The following shows the skeleton of the class:

```
public class MyRandomLayout
    extends IlvGraphLayout
{
    public MyRandomLayout()
    {
    }

    public MyRandomLayout(MyRandomLayout source)
    {
        super.source(source);
    }

    public IlvGraphLayout copy()
    {
        return new MyRandomLayout(this);
    }

    protected void layout(boolean redraw)
    {
        ...
    }
}
```

The constructor with no arguments is empty. The `copy` constructor and the `copy` method are implemented; they are used when laying out a nested graph (see *Nested layouts*).

Then, the abstract method `layout(boolean)` of the base class is implemented as follows:

```
protected void layout(boolean redraw)
{
    // obtain the graph model
    GraphModel graphModel = (GraphModel)getGraphModel();

    // obtain the layout report
    IlvGraphLayoutReport layoutReport = getLayoutReport();

    // obtain the layout region
    IlvRect rect = getCalcLayoutRegion();
    float xMin = rect.x;
    float yMin = rect.y;
    float xMax = rect.x + rect.width;
    float yMax = rect.y + rect.height;

    // initialize the random generator
    Random random = (isUseSeedValueForRandomGenerator()) ?
        new Random(getSeedValueForRandomGenerator()) :
        new Random();

    // browse the objects in the grapher
    Enumeration nodes = graphModel.getNodes();
    while (nodes.hasMoreElements()) {
        Object node = nodes.nextElement();

        // skip fixed nodes
        if (isPreserveFixedNodes() && isFixed(node))
            continue;

        // compute coordinates
        float x = xMin + (xMax - xMin) * random.nextFloat();
        float y = yMin + (yMax - yMin) * random.nextFloat();

        // move the node to the computed position
        graphModel.moveNode(node, x, y, redraw);

        // notify listeners on layout events
        callLayoutStepPerformedIfNeeded();
    }

    // set the layout report code
    layoutReport.setCode(IlvGraphLayoutReport.LAYOUT_DONE);
}
...

```

Note that the `layout` method is `protected`, which is the access type of the method in the base class. This will not prevent a user outside the package containing the class from performing the layout because it is started using the public method `performLayout`.

Implementing the layout method

Depending on the characteristics of the layout algorithm, some of the steps required may be different or unnecessary, or other steps may be needed.

Depending on the particular implementation of your layout algorithm, other methods of the `IlvGraphLayout` class may need to be overridden. For instance, if your subclass supports some of the generic parameters of the base class, you must override the `supports [ParameterName]` method (see *Base class parameters and features*). For further information about the class `IlvGraphLayout`, refer to the API reference documentation.

To implement the `layout` method in the sample custom layout algorithm:

1. Obtain the graph model (`getGraphModel()` on the layout instance).

```
GraphModel graphModel = (GraphModel) getGraphModel();
```

2. Obtain the instance of the layout report that is automatically created when the `performLayout` method from the superclass is called (`getLayoutReport()` on the layout instance). See *Using a graph layout report*.

```
IlvGraphLayoutReport layoutReport = getLayoutReport();
```

3. Obtain the layout region parameter to compute the area where the nodes will be placed.

```
IlvRect rect = getCalcLayoutRegion();
```

4. Initialize the random generator.

```
Random random = (isUseSeedValueForRandomGenerator()) ?  
    new Random(getSeedValueForRandomGenerator()) :  
    new Random();
```

(For information on the seed value parameter, see *Random generator seed value*.)

5. Get an enumeration of the nodes (`getNodes()` on the graph model instance).

```
Enumeration nodes = graphModel.getNodes();
```

6. Browse the nodes, skipping fixed nodes (`isFixed(node)` on the layout instance) if asked by the user (`isPreserveFixedNodes()` on the layout instance).

```
while (nodes.hasMoreElements()) {  
    Object node = nodes.nextElement();  
    ...  
}
```

(For details on fixed nodes, see *Preserve fixed nodes*).

7. Move each node to the newly computed coordinates inside the layout region (`graphModel.moveNode()`).

```
graphModel.moveNode (node, x, y, redraw);
```

8. Notify the listeners on layout events that a new node was positioned (`callLayoutStepPerformedIfNeeded()` on the layout instance). This allows the user to implement, for example, a progress bar if a layout event listener was registered on the layout instance.

```
callLayoutStepPerformedIfNeeded();
```

(For details on event listeners, see *Using event listeners*.)

9. Finally, set the code in the layout report.

```
layoutReport.setCode (IlvGraphLayoutReport.LAYOUT_DONE);
```

Once you have implemented your own layout algorithm `MyRandomLayout`, you can use it directly in Java™ .

FAQs about using the layout algorithms

The following list of FAQs provides some helpful suggestions for using the layout algorithms. You may find some answers to questions that come up when using the graph layout package.

FAQs about the layout algorithms

| Question | Answer |
|---|---|
| I perform the layout and nothing happens (no node is moved). Why? | <p>One possible reason may be: the layout algorithms provided in IBM® ILOG® JViews Graph Layout for Eclipse are all designed to do nothing, by default, if no change occurred in the graph since the last time the layout was performed successfully on the same graph. A change means that a node was moved, or a node or link was added, removed, or reshaped.</p> <p>Note that you can force the layout to be performed again, even if no change occurred, by calling the <code>performLayout(boolean, boolean)</code> method with a <code>true</code> value for the force argument.</p> <p>Another possible reason may be: an error or a special case occurred during the layout. First, you should check whether the <code>performLayout()</code> method has thrown an exception. If no exception was thrown, call the <code>getCode()</code> method on the instance of the layout report returned by the <code>performLayout</code> method. Check this value with respect to the documentation of the appropriate layout report class. (For details, see <i>Using a graph layout report.</i>)</p> |
| With the Uniform Length Edges algorithm, after having performed the layout once, I don't see any movement even if I use the force layout option. Why? | <p>The reason is probably that the first time you performed the layout, the algorithm reached the convergence. When the layout is performed again, it detects that the convergence has been already reached and stops. If you really want to continue working, for instance in order to "declutter" a particular part of the graph, you may need to move one or several nodes in order to change the initial configuration. (The algorithm is dependent on the initial configuration.)</p> |
| After performing the layout, the graph is laid out far from its initial position. Why? | <p>Most of the layout algorithms use a layout region parameter to control the size and position of the layout. (For details, see <i>Layout region.</i>) Depending on the value of this parameter, the nodes may be moved far from their initial positions.</p> <p>To know whether a layout algorithm is designed to use a layout region parameter, check the documentation to see if the layout class overrides the <code>supportsLayoutRegion()</code> method of the base class in order to return <code>true</code>.</p> <p>Other algorithms have a different mechanism that allows you to specify the desired location of the layout. It may happen that the default value of the location parameter is such that the graph is laid out far from its initial position.</p> |
| When I use certain layout algorithms on certain graphs, there are overlapping nodes. Why and what can I do? | <p>One possible reason may be related to the different ways layout algorithms deal with the size of the nodes:</p> <ul style="list-style-type: none">-The Topological Mesh algorithm is not able to explicitly take into account the size of the nodes. |

| Question | Answer |
|--|--|
| | <p>- The Tree, Hierarchical, Bus, and Grid algorithms always avoid overlapping nodes. (The Link algorithm does not move the nodes. It only reshapes the links such that the crossings and overlaps are reduced. The size of the nodes is taken into account.)</p> <p>- The Uniform Length Edges algorithm (with the option "Respect Node Sizes" enabled) and the Circular algorithm, in many cases, succeed in avoiding overlapping nodes.</p> <p>In any case, if the layout algorithm supports the layout region mechanism (see <i>Layout region</i>), you should try to increase the size of the layout region. For example, if your graph contains hundreds of nodes, it is not reasonable to use a small layout region, such as 600x600. There will be not enough space for all the nodes. You should try a larger layout region, for example 5000x5000.</p> <p>The optimal size of the layout region depends, of course, not only on the number of nodes, but also on their size. If the nodes are relatively large with respect to the size of the layout region, it may be necessary to adjust some of the parameters (for instance, the preferred link length for the Uniform Length Edges Layout).</p> |
| <p>In some networks, there are two (or more) subnetworks that are not connected. How will this affect the layout algorithms?</p> | <p>This depends on the layout class you use:</p> <ul style="list-style-type: none"> - <code>IlvTopologicalMeshLayout</code>: It will work on the connected component of the graph that contains the starting node. (You can specify this node as a parameter.) If the "starting node" is not specified, it is automatically chosen in an arbitrary way. The nodes of the other "connected components" will not be moved. You may want to perform the layout separately on each connected component using different layout regions and starting node settings. This is what you get automatically when you enable the "layout of connected components" parameter. (See <i>Layout of connected components</i>.) - <code>IlvUniformLengthEdges</code>: This algorithm supports disconnected graphs, but usually it is better to rely on the automatic "layout of connected components" parameter. (See <i>Layout of connected components</i>.) - <code>IlvBusLayout</code>: It will work on the "connected component" of the graph that contains the "bus object." (You must specify the bus object as a parameter.) The other nodes that are not connected to the bus will not be moved. You may need to perform the layout separately on each connected component. This is what you get automatically when you enable the "layout of connected components" parameter. (See <i>Layout of connected components</i>.) - <code>IlvCircularLayout</code>, <code>IlvHierarchicalLayout</code>, <code>IlvTreeLayout</code>: They have built-in support for disconnected graphs. Alternatively, you can use the automatic support from the base class. (See <i>Layout of connected components</i>.) |

| Question | Answer |
|--|--|
| | <p>- <code>IlvLinkLayout</code>, <code>IlvGridLayout</code>, <code>IlvRandomLayout</code>: These algorithms support both connected and disconnected graphs. Their behavior is the same for both categories of graphs.</p> |
| <p>There are some attributes of the network that we know about (for instance, we know what the core switch is and what the center should be). Are such attributes taken into account by the layout algorithm?</p> | <p>It depends on the layout algorithm.</p> <ul style="list-style-type: none"> - The Circular Layout is designed to allow you to specify information about the physical topology of the network. You can specify which nodes belong to the same cluster (ring or star), the order of the nodes on the cluster, and which node is the center of a star cluster. - In the Tree Layout, you can specify the root node. - In the Bus Layout algorithm, you can specify the bus object. - In the Hierarchical Layout algorithm, you can specify node position indices and level indices, as well as relative positioning constraints. |
| <p>If I use IBM® ILOG® JViews Graph Layout for Eclipse on different computers or with different Java Virtual Machines (JVM) or both, I sometimes get different layouts for the same graph and with the same parameters. Why?</p> | <p>There are two possible reasons:</p> <ol style="list-style-type: none"> 1. Different computers and JVMs may be slower or faster. If the layout algorithm you use stops the computation when the specified allowed time has elapsed, a slower computer or JVM will cause the computation to stop earlier. That may be the cause of different results. This may happen even with the same computer and JVM if the charge of the computer is increased. You may need to increase the allowed time specification when running on a slower computer or JVM. 2. If you use a layout algorithm that uses the random generator and if you use the default option for the seed value (that is, the system clock is used), you get different results for each successive run of the layout on the same graph. This allows you to obtain alternative results and to choose the one you prefer. If you want to prevent different results for successive runs, you can specify a constant seed value. |
| <p>I use the Link Layout algorithm to lay out the links of a network of graphical objects. When several links connect to the same side of a node, they overlap, while I expect them to respect the “link offset” (or the “grid size”) parameter of the Link Layout. Why?</p> | <p>Some dimensional parameters of the layout algorithms need to be chosen with respect to the size of the nodes. This is the case of the “link offset” and the “bypass distance” parameters for the Short Link Layout and the grid size for the Long Link Layout. Indeed, their default values are not appropriate when the nodes are very large. Compared to this size, the default values of the parameters are so small that they appear to be zero.</p> <p>The solution is to increase the values of the dimensional parameters, taking into account the size of the nodes. If different nodes have different sizes, either the medium or the largest size of the nodes can be used to compute the parameters as a fraction of this size.</p> |

Index

A

- absolute level position range/tendency **191**
- accessing
 - all sublayouts **353**
- addGraphLayoutEventListener method
 - IlvGraphLayout class **423**
- advanced recursion **334**
- alignment options
 - Bus Layout **283**
 - Grid Layout **319**
 - Tree Layout (free mode) **113**
- allowed time parameter
 - Bus Layout **274**
 - Grid Layout **316**
 - Hierarchical Layout **164**
 - in IlvGraphLayout **39**
 - Link Layout **228**
 - Multiple Layout **366**
 - Recursive Layout **345**
 - Topological Mesh Layout **61**
 - Tree Layout **101**
 - Uniform Length Edges Layout **83**
- angle layout criteria **12**
- Annealing Label Layout **389**
 - allowed time parameter **395**
 - automatic update **409**
 - description **394**
 - expert parameters **409**
 - features **392**
 - generic parameters **395**
 - global parameters **407**
 - label descriptors **397**
 - implementing your own **411**
 - subclasses **397**
 - label movement policy **408**
 - label offset parameter **407**
 - limitations **393**

- obstacle offset parameter **407**
- percentage of completion calculation parameter **395**
- point label descriptor **398**
- polyline label descriptor **403**
- random generator seed value parameter **395**
- stop immediately parameter **396**
- use default parameters **396**

- area layout criteria **12**
- area minimization parameter, Circular Layout **304**
- aspect ratio parameter
 - Tree Layout (radial mode) **137**
 - Tree Layout (tip-over mode) **142**
- attach method
 - IlvGraphLayout class **21**
 - IlvLabelLayout class **378**
- attaching/detaching a grapher **21**
- automatic label placement **373**
- automatic layout
 - description **13, 32**
 - Link Layout **228**
- automatic update
 - labels **409**

B

- bends layout criteria **12**
- Bus Layout
 - alignment options **283**
 - applicable graph types **268**
 - application domains **268**
 - bus line extremity adjusting **282**
 - bus node parameter **278**
 - description **270**
 - dimensional parameters **287**
 - features **269**
 - flow direction parameter **279**
 - generic parameters **274**
 - global alignment parameters **283**

- horizontal offset parameter **287**
- incremental mode parameter **286**
- individual node alignment parameter **284**
- link clipping parameter **274**
- link style parameter **279**
- margin on bus parameter **288**
- margin parameter **288**
- maximum nodes per level parameter **280**
- node position **285**
- order parameter **276, 317**
- sample drawing **268**
- specific parameters **276**
- vertical offset parameter **288**
- vertical offset to previous level parameter **288**
- width adjusting **281**
- bus line extremity adjusting
 - Bus Layout **282**
- bus node parameter, Bus Layout **278**
- bypass distance parameter, Link Layout (short link mode) **251**

C

- calculated level index parameter, Hierarchical Layout **215**
- calculated position index parameter, Hierarchical Layout **216**
- Circular Layout
 - applicable graph types **293**
 - application domains **293**
 - area minimization parameter **304**
 - cluster contents parameter **307**
 - cluster membership parameters **301**
 - cluster position parameter **307**
 - cluster size parameter **307**
 - clustering mode parameter **301**
 - description **295**
 - dimensional parameters **306**
 - disconnected graph offset parameter **307**
 - features **294**
 - generic parameters **299**
 - level offset parameter **307**
 - limitations **294**
 - link clipping parameter **299, 308**
 - link connection box parameter **299, 309**
 - link style parameter **308**
 - offset parameter **306**
 - order of nodes parameter **301**
 - ring topology **295**
 - root clusters parameter **303**
 - sample drawings **292**
 - specific parameters **301**
 - star center parameter **303**
 - star topology **295**
- cluster contents parameter, Circular Layout **307**

- cluster membership parameters, Circular Layout **301**
- cluster position parameter, Circular Layout **307**
- cluster size parameter, Circular Layout **307**
- clustering modes, Circular Layout **301**
- compass directions, Tree Layout **105**
- connected components parameter
 - Bus Layout **274**
 - Circular Layout **299**
 - Hierarchical Layout **164**
 - Multiple Layout **366**
 - Topological Mesh Layout **61**
 - Tree Layout **101**
 - Uniform Length Edges Layout **83**
- connector style parameter
 - Hierarchical Layout **171**
 - Tree Layout **119**
- createGraphLayout method
 - IlvDefaultLayoutProvider class **358**
- createLayoutReport method
 - IlvGraphLayout class **419**
- CSS (Cascading Style Sheet)
 - none
 - for most advanced features **416**
- CSS samples
 - Random Layout **262**

D

- detach method
 - IlvLabelLayout class **21, 378**
- detachLayouts method
 - IlvDefaultLayoutProvider class **353**
- dimensional parameters
 - Bus Layout **287**
 - Circular Layout **306**
 - Grid Layout **321**
- disconnected graph
 - laying out connected components **425**
 - offset parameter, Circular Layout **307**

E

- east-west neighbors, Tree Layout **147**
- end points mode parameter
 - Hierarchical Layout **173**
 - Link Layout **233**
- evenly spaced pins margin ratio, Link Layout **247**
- events, label layout **384**
- extremity constraints, Hierarchical Layout **207**

F

- fallback mechanism, Link Layout (long link mode) **257**
- FAQ **432**
- fixed links parameter
 - Bus Layout **275**
 - Circular Layout **300**
 - Hierarchical Layout **165**

- Link Layout **228**
- Random Layout **263**
- Topological Mesh Layout **62**
- Tree Layout **102**
- Uniform Length Edges Layout **84**
- fixed nodes parameter
 - Bus Layout **275**
 - Circular Layout **300**
 - Grid Layout **316**
 - Hierarchical Layout **165**
 - Random Layout **263**
 - Topological Mesh Layout **62**
 - Tree Layout **102**
 - Uniform Length Edges Layout **84**
 - using to refine a TML graph layout **70**
- flow direction parameter
 - Bus Layout **279**
 - Hierarchical Layout **166**
 - Tree Layout (free mode) **111**
- force fit to layout region, Uniform Length Edges Layout **86**
- fork link shapes, Hierarchical Layout **180**
- free layout mode (Tree Layout)
 - alignment parameter **113**
 - description **109**
 - flow direction **111**
 - global alignment **113**
 - global link style parameter **117**
 - individual link style **117**
 - individual node alignment **114**
 - link style **117**
 - orthogonal fork percentage **124**
 - respect node sizes **124**
 - spacing parameters **123**
 - spacing parameters for experts **124**
 - tip-over alignment **115**

G

- getAlignment method
 - IlvTreeLayout class **114**
- getBox method
 - IlvLinkConnectionBoxInterface class **252**
- getCalcBackwardTreeLinks method
 - IlvTreeLayout class **148**
- getCalcForwardTreeLinks method
 - IlvTreeLayout class **148**
- getCalcNodeLevelIndex method
 - IlvHierarchicalLayout class **215**
- getCalcNodePositionIndex method
 - IlvHierarchicalLayout class **216**
- getCalcNonTreeLinks method
 - IlvTreeLayout class **148**
- getCalcRoots method
 - IlvTreeLayout class **103**
- getCode method

- IlvGraphLayoutReport class **421**
- IlvLabelLayoutReport class **382**
- getDestinationPointMode method
 - IlvHierarchicalLayout class **173**
 - IlvLinkLayout class **233**
- getEastNeighbor method
 - IlvTreeLayout class **147**
- getFirstGraphLayout method
 - IlvMultipleLayout class **370**
- getGraphLayout method
 - IlvGraphLayout class **341**
- getHorizontalAlignment method
 - IlvGridLayout class **320**
- getLabelLayout method
 - IlvMultipleLayout class **370**
- getLayoutOfConnectedComponents method
 - IlvGraphLayout class **425**
- getLayouts method
 - IlvGraphLayout class **330**
- getLayoutTime method
 - IlvGraphLayoutReport class **421**
 - IlvLabelLayoutReport class **382**
- getLinkConnectionBoxInterface method
 - IlvShortLinkLayout class **248**
- getLinkStyle method
 - IlvHierarchicalLayout class **170**
 - IlvLinkLayout class **239, 249**
 - IlvTreeLayout class **117**
- getLongLinkLayout method
 - IlvLinkLayout class **255**
- getNodeBoxInterface method
 - IlvShortLinkLayout class **248**
- getNumberOfPossibleExteriorCycles method
 - IlvTopologicalMeshLayout class **67**
- getOriginPointMode method
 - IlvHierarchicalLayout class **173**
 - IlvLinkLayout class **233**
- getPenalty method
 - IlvTerminationPointFilter class **256**
- getPercentageComplete method
 - IlvLabelLayoutReport class **383**
- getPreferredLayout method
 - IlvDefaultLayoutProvider class **358**
- getPreferredPathLocation method
 - IlvAnnealingLabelDescriptor class **411**
- getSecondGraphLayout method
 - IlvMultipleLayout class **370**
- getShortLinkLayout method
 - IlvLinkLayout class **246**
- getSpecNodeLevelIndex method
 - IlvHierarchicalLayout class **199**
- getSpecNodePositionIndex method
 - IlvHierarchicalLayout class **203**

- getSpecRoots method
 - IlvTreeLayout class **103**
- getTangentialOffset method
 - IlvLinkConnectionBoxInterface class **252**
- getVerticalAlignment method
 - IlvGridLayout class **320**
- getWestNeighbor method
 - IlvTreeLayout class **147**
- global alignment parameters
 - Bus Layout **283**
 - Grid Layout **319**
 - Tree Layout (free mode) **113**
- global connector style parameter
 - Link Layout **238**
- global end point mode parameter
 - Hierarchical Layout **173**
 - Link Layout **233**
- global incremental link reshape mode **249**
- global link style parameter
 - Hierarchical Layout **170**
 - Link Layout **231**
 - Tree Layout (free mode) **117**
- global side association for polyline label descriptors **405**
- graph layout
 - class packages **18**
 - features **14**
 - questions and answers **432**
 - report **417**
- graph layout parameters
 - allowed time **39**
 - description **39**
 - preserve fixed links **46**
 - use default parameters **50**
- grapher
 - attaching/detaching **21**
 - laying out connected components of a disconnected graph **425**
- GraphLayoutEventListener interface **423**
- grid base parameter, Link Layout (long link mode) **241**
- Grid Layout
 - alignment options **319**
 - applicable graph types **313**
 - application domains **313**
 - description **315**
 - dimensional parameters **321**
 - features **314**
 - generic parameters **316**
 - global alignment parameters **319**
 - grid offset parameter **323**
 - incremental mode parameter **321**
 - individual node alignment parameter **320**
 - layout modes **318**
 - margin parameter **324**

- maximum nodes per row or column parameter **320**
 - sample drawing **312**
 - specific parameters **317**
- grid offset parameter
 - Grid Layout **323**
 - Link Layout (long link mode) **241**

H

- Hierarchical Layout
 - applicable graph types **157**
 - application domains **158**
 - calculated level index parameter **215**
 - calculated position index parameter **216**
 - connector style parameter **171**
 - description **161**
 - end points mode parameter **173**
 - extremity constraints **207**
 - features **159**
 - flow direction parameter **166**
 - fork link shapes **180**
 - generic parameters **164**
 - global end point mode parameter **173**
 - global link style parameter **170**
 - individual end point mode parameter **173**
 - individual link style parameter **170**
 - layout constraints **194**
 - level index parameter **199**
 - level justification parameter **168**
 - leveling strategy parameter **167**
 - limitations **159**
 - link clipping parameter **164, 174**
 - link connection box parameter **165, 174**
 - link priority parameter **181**
 - link style parameter **169**
 - link width parameter **176**
 - port index parameter **178**
 - port sides parameter **176**
 - position index parameter **203**
 - relative position constraints **194, 204**
 - sample drawings **155**
 - side-by-side constraints **205**
 - spacing parameters **183**
 - specific parameters **166**
 - swim lane constraint **209**
- horizontal offset parameter, Bus Layout **287**

I

- IlvAnnealingLabelDescriptor class
 - getPreferredPathLocation method **411**
 - initialize method **411**
 - setTowardsPreferredPosition method **411**
- IlvAnnealingLabelLayout class **377, 395, 407**
- IlvAnnealingPointLabelDescriptor class
 - constructor **398, 399, 400**
- IlvAnnealingPolylineLabelDescriptor class

constructor **403, 405**
 IlvBusLayout class
 setBus **method 278**
 setVerticalOffsetToPreviousLevel **method 288**
 IlvDefaultLayoutProvider class **358**
 createGraphLayout **method 358**
 detachLayouts **method 353**
 getPreferredLayout **method 358**
 setPreferredLayout **method 358**
 IlvGraphLayout class
 addGraphLayoutEventListener **method 423**
 attach **method 21**
 attaching/detaching a grapher **21**
 createLayoutReport **method 419**
 getGraphLayout **method 341**
 getLayoutOfConnectedComponents **method 425**
 getLayouts **method 330**
 instantiating a subclass **20**
 isLayoutOfConnectedComponentsEnabled **method 425**
 isLayoutOfConnectedComponentsEnabledByDefault **method 425**
 isUseDefaultParameters **method 50**
 layout **method 21, 378, 430**
 layout parameters and features **39**
 layoutStepPerformed **method 423**
 performLayout **method 22, 330, 419, 432**
 setLayoutOfConnectedComponents **method 425**
 setLayoutOfConnectedComponentsEnabled **method 425**
 setLinkClipInterface **method 122, 174, 308**
 setLinkConnectionBoxInterface **method 121, 174, 309**
 setUseDefaultParameters **method 50**
 subclassing **427**
 supportsAllowedTime **method 46, 50**
 supportsLayoutOfConnectedComponents **method 425**
 supportsLayoutRegion **method 432**
 IlvGraphLayoutReport class
 description **418**
 getCode **method 421**
 getTime **method 421**
 stored information **421**
 IlvGraphLayoutUtil class
 IsTree **static method 32**
 IlvGridLayout class
 getHorizontalAlignment **method 320**
 getVerticalAlignment **method 320**
 setHorizontalAlignment **method 320**
 setVerticalAlignment **method 320**
 IlvHierarchicalLayout class
 getCalcNodeLevelIndex **method 215**
 getCalcNodePositionIndex **method 216**
 getDestinationPointMode **method 173**
 getLinkStyle **method 170**
 getOriginPointMode **method 173**
 getSpecNodeLevelIndex **method 199**
 getSpecNodePositionIndex **method 203**
 setDestinationPointMode **method 173**
 setGlobalDestinationPointMode **method 173**
 setGlobalLinkStyle **method 170**
 setGlobalOriginPointMode **method 173**
 setLinkStyle **method 170**
 setOriginPointMode **method 173**
 setSpecNodeLevelIndex **method 199**
 setSpecNodePositionIndex **method 203**
 IlvLabelLayout class **377**
 attach **method 378**
 detach **method 21, 378**
 performLayout **method 379**
 IlvLabelLayoutReport class
 getCode **method 382**
 getPercentageComplete **method 383**
 IlvLinkConnectionBoxInterface class
 getBox **method 252**
 getTangentialOffset **method 252**
 IlvLinkLayout class
 getDestinationPointMode **method 233**
 getLinkStyle **method 239, 249**
 getLongLinkLayout **method 255**
 getOriginPointMode **method 233**
 getShortLinkLayout **method 246**
 setDestinationPointMode **method 233**
 setGlobalDestinationPointMode **method 233**
 setGlobalLinkStyle **method 231, 238, 249**
 setGlobalOriginPointMode **method 233**
 setLinkStyle **method 239, 249**
 setOriginPointMode **method 233**
 IlvLongLinkLayout class
 setTerminationPointFilter **method 256**
 IlvMultipleLayout class
 getFirstGraphLayout **method 370**
 getLabelLayout **method 370**
 getSecondGraphLayout **method 370**
 setFirstGraphLayout **method 370**
 setLabelLayout **method 370**
 setSecondGraphLayout **method 370**
 IlvMultipleLayoutReport class **422**
 IlvRandomLabelLayout class **377**
 IlvRecursiveLayoutReport class **422**
 IlvShortLinkLayout class

- getLinkConnectionBoxInterface method **248**
- getNodeBoxInterface method **248**
- IlvTerminationPointFilter class
 - getPenalty method **256**
- IlvTopologicalMeshLayout class
 - getNumberOfPossibleExteriorCycles method **67**
 - setExteriorCycleId method **67**
- IlvTopologicalMeshLayoutReport class **422**
- IlvTreeLayout class
 - getAlignment method **114**
 - getCalcBackwardTreeLinks method **148**
 - getCalcForwardTreeLinks method **148**
 - getCalcNonTreeLinks method **148**
 - getCalcRoots method **103**
 - getEastNeighbor method **147**
 - getLinkStyle method **117**
 - getSpecRoots method **103**
 - getWestNeighbor method **147**
 - setAlignment method **114, 115**
 - setAspectRatio method **137**
 - setBranchOffset method **123, 138**
 - setConnectorStyle method **119**
 - setEastWestNeighboring method **147**
 - setFlowDirection method **111**
 - setGlobalAlignment method **113, 115**
 - setGlobalLinkStyle method **117**
 - setInvisibleRootUsed method **139**
 - setLayoutMode method **105, 110, 128, 132, 136**
 - setLayoutOfConnectedComponentsEnabled method **139**
 - setLevelAlignment method **130**
 - setLinkStyle method **117**
 - setOrthForkPercentage method **124**
 - setOverlapPercentage method **124**
 - setParentChildOffset method **123, 138**
 - setPosition method **104**
 - setRoot method **103**
 - setRootPreference method **103**
 - setSiblingOffset method **123, 138**
 - setTipOverBranchOffset method **123**
 - setWestEastNeighboring method **147**
- IlvUniformLengthEdgesLayoutReport class **422**
- incremental layout **13**
- incremental link reshape mode **248**
 - global **249**
 - individual **249**
- incremental mode parameter
 - Bus Layout **286**
 - Grid Layout **321**
 - Link Layout **234**
- individual connector style parameter
 - Link Layout **239, 249**

- individual end point mode parameter
 - Hierarchical Layout **173**
 - Link Layout **233**
- individual incremental link reshape mode **249**
- individual link style parameter
 - Hierarchical Layout **170**
 - Link Layout **231**
 - Tree Layout (free mode) **117**
- individual node alignment parameter
 - Bus Layout **284**
 - Grid Layout **320**
 - Tree Layout (free mode) **114**
- initialize method
 - IlvAnnealingLabelDescriptor class **411**
- intergraph link routing **235**
- internal provider mode
 - Recursive Layout **340, 341, 350**
- isLayoutOfConnectedComponentsEnabled method
 - IlvGraphLayout class **425**
- isLayoutOfConnectedComponentsEnabledByDefault method
 - IlvGraphLayout class **425**
- IsTree static method
 - IlvGraphLayoutUtil class **32**
- isUseDefaultParameters method
 - IlvGraphLayout class **50**

J

- Java code samples
 - applying a single layout to a nested graph **331**
 - defining a new type of layout **428**
 - labels, positioning
 - at a point **399**
 - at an obstacle **398**
 - on multiple criteria **400**
 - Multiple Layout **362**
 - Recursive Layout
 - layout providers **358**

L

- label descriptors **397**
 - implementing your own **411**
- Label Layout
 - annealing **389**
 - base class **377**
 - events and listeners **384**
 - IlvLabelLayout parameters **386**
 - instantiating and attaching a subclass **378**
 - performing a layout **379**
 - recursively on nested subgraphs **380**
 - report **382**
 - using in Java **375**
- labeling model
 - reference, in Recursive Multiple Layout **372**
- labels

- automatic placement **373**
- descriptors, subclasses **397**
- movement policy **408**
- point label descriptor **398**
- polyline label descriptor **403**
- positioning
 - at a point, Java code sample **399**
 - at an obstacle, Java code sample **398**
 - on multiple criteria, Java code sample **400**
- layout algorithms
 - choosing **28**
 - questions and answers **432**
 - setting the selection method **31**
 - table of additional information **52**
 - table of applicable graphs **28**
 - table of generic parameters supported **36**
- layout constraints **194**
 - in Java **195**
- layout criteria
 - angle **12**
 - area **12**
 - bends **12**
 - link crossings **12**
 - symmetries **12**
- layout method
 - `IlvGraphLayout` class **21, 378, 430**
 - steps for implementing **430**
- layout methods, types of
 - automatic **13, 32**
 - incremental **13**
 - semi-automatic **13, 32**
 - static **13**
- layout modes
 - Grid Layout **318**
 - Link Layout **229**
 - Recursive Layout **347**
 - Tree Layout **105**
- layout providers
 - Recursive Layout **358**
- layout region parameter
 - Bus Layout **274**
 - Circular Layout **299**
 - Grid Layout **316**
 - Random Layout **263**
 - Topological Mesh Layout **62**
 - Uniform Length Edges Layout **83**
 - using to refine a TML graph layout **72**
- layouts
 - applying the same recursively **330**
 - Bus Layout **267**
 - Circular Layout **291**
 - defining your own type **427**
 - code sample **428**
 - Force-directed layout **75**
 - Grid Layout **311**
 - Hierarchical Layout **153**
 - implementing the layout method **430**
 - Link Layout **219**
 - mixing different in nested graph **334**
 - Multiple Layout **361**
 - order of recursive layouts **329**
 - performing **21**
 - Random Layout **259**
 - Recursive Layout **337**
 - Topological Mesh Layout **55**
 - Tree Layout **93**
 - Uniform Length Edges Layout **75**
- `layoutStepPerformed` method
 - `IlvGraphLayout` class **423**
- level index parameter, Hierarchical Layout **199**
- level justification parameter, Hierarchical Layout **168**
- level layout mode (Tree Layout)
 - description **127**
 - general parameters **129**
 - level alignment parameter **130**
- level offset parameter, Circular Layout **307**
- leveling strategy parameter
 - Hierarchical Layout **167**
- limitations
 - Hierarchical Layout **159**
 - Link Layout **225**
 - Random Layout **261, 294**
 - Topological Mesh Layout **58**
 - Tree Layout **97**
 - Uniform Length Edges Layout **81**
- link box connection interface
 - Link Layout (short link mode) **252**
- link categories, retrieving (Tree Layout) **148**
- link clipping parameter
 - Bus Layout **274**
 - Circular Layout **299, 308**
 - Hierarchical Layout **164, 174**
 - Topological Mesh Layout **62, 73**
 - Tree Layout **102, 122**
 - Uniform Length Edges Layout **83, 91**
- link connection box interface
 - Link Layout **244**
- link connection box parameter
 - Circular Layout **299, 309**
 - Hierarchical Layout **165, 174**
 - Topological Mesh Layout **62**
 - Tree Layout **102, 121**
 - Uniform Length Edges Layout **84**
- link crossing penalty parameter, Link Layout (short link mode) **251**
- link crossings layout criteria **12**
- Link Layout
 - applicable graph types **223**

application domains **223**
 bypass distance parameter (short link mode) **251**
 choosing the appropriate layout mode **230**
 connector style parameter **238**
 description **226**
 end points mode parameter **233**
 evenly spaced pins margin ratio (short link mode) **247**
 fallback mechanism (long link mode) **257**
 features **224**
 generic parameters **228**
 global connector style parameter **238**
 global end point mode parameter **233**
 global link style parameter **231**
 grid base parameter (long link mode) **241**
 grid offset parameter (long link mode) **241**
 incremental link reshape mode **248**
 incremental mode parameter **234**
 individual connector style parameter **239, 249**
 individual end point mode parameter **233**
 individual link style parameter **231**
 intergraph link routing **235**
 layout mode parameter **229**
 limitations **225**
 link box connection interface (short link mode) **252**
 link connection box interface **244**
 link crossing penalty parameter (short link mode) **251**
 link offset parameter (short link mode) **237**
 link routing parameters (long link mode) **256**
 link style parameter **230**
 long link layout algorithm **226**
 minimal distance parameter (long link mode) **241**
 minimal node corner offset parameter (long link mode) **242**
 minimum final segment length parameter (long link mode) **242**
 minimum final segment parameter **238**
 node-side filter feature **243**
 number of optimization iterations (short link mode) **246**
 obstacle parameters (long link mode) **255**
 same shape for multiple links parameter (short link mode) **250**
 sample drawing **221**
 self-link style parameter (short link mode) **246**
 short link layout algorithm **226**
 spacing parameters (long link mode) **240**
 spacing parameters (short link mode) **237**
 specific parameters **229**
 variable end point parameters (long link mode) **256**
 Link Layout (short link mode) **238**
 link offset parameter, Link Layout (short link mode) **237**
 link overlap nodes forbidden parameter
 Link Layout (short link mode) **247**
 link priority parameter, Hierarchical Layout **181**
 link routing parameters, Link Layout (long link mode) **256**
 link style parameter
 Bus Layout **279**
 Circular Layout **308**
 Hierarchical Layout **169**
 Link Layout **230**
 Random Layout **265**
 Topological Mesh Layout **64**
 Tree Layout (free mode) **117**
 Uniform Length Edges Layout **85**
 link width parameter, Hierarchical Layout **176**
 listener, layout event
 code example **423**
 description **423**
 GraphLayoutEventListener interface **423**
 listeners, label layout **384**
 local side association for polyline label descriptors **404**
 long link mode (Link Layout)
 algorithm description **226**
 fallback mechanism **257**
 features **224**
 grid base parameter **241**
 grid offset parameter **241**
 link routing parameters **256**
 minimal distance parameter **241**
 minimal node corner offset parameter **242**
 minimum final segment length parameter **242**
 obstacle parameters **255**
 spacing parameters **240**
 variable end point parameters **256**

M

margin on bus parameter, Bus Layout **288**
 margin parameter
 Bus Layout **288**
 Grid Layout **324**
 maximum nodes per level parameter
 Bus Layout **280**
 Grid Layout **320**
 memory savings parameter
 Topological Mesh Layout **62**
 minimal distance parameter, Link Layout (long link mode) **241**
 minimal node corner offset parameter, Link Layout (long link mode) **242**

minimum final segment length parameter, Link Layout (long link mode) **242**
minimum final segment parameter, Link Layout (short link mode) **238**

Multiple Layout

- accessing sublayouts **370**
- allowed time parameter **366**
- application domain **362**
- attaching graph and labeling models **369**
- combining multiple and recursive layout **371**
- connected components parameter **366**
- features **365**
- for experts **369, 371**
- generic parameters **366**
- Java code sample **362**
- percentage completion parameter **366**
- recursive layout **363**
- reference labeling model **372**
- simple layout **363**
- specific parameters **368**
- stop immediately parameter **366**

N

nested subgraphs **380**

node position

- Bus Layout **285**

node-side filter feature, Link Layout **243**

nodes placement algorithm, Topological Mesh Layout **65**

nodes placement allowed time, Topological Mesh Layout **64**

nodes placement iterations parameter, Topological Mesh Layout **64**

number of iterations parameter, Uniform Length Edges Layout **85**

number of optimization iterations, Link Layout (short link mode) **246**

O

obstacle offset parameter **407**

obstacle parameters, Link Layout (long link mode) **255**

obstacles (Label Layout)

- positioning a label at **398, 399**
- positioning at polyline **403, 405**
- related obstacles **398, 399, 400**
 - examples **398**

offset parameter, Circular Layout **306**

optimization allowed time parameter, Topological Mesh Layout **64**

optimization iterations parameter, Topological Mesh Layout **64**

order of nodes parameter, Circular Layout **301**

order parameter, Bus Layout **276, 317**

orthogonal fork percentage parameter, Tree Layout **124**

outer cycle parameter, Topological Mesh Layout description **67**

- using to refine a TML graph layout **72**

overlap

- Tree Layout **124**

P

parameters

- generic

- Annealing Label Layout **395**

- Bus Layout **274**

- Circular Layout **299**

- Grid Layout **316**

- Hierarchical Layout **164**

- Link Layout **228**

- Multiple Layout **366**

- Random Layout **263**

- Recursive Layout **345**

- Topological Mesh Layout **61**

- Tree Layout **101**

- Uniform Length Edges Layout **83**

- specific

- Bus Layout **276**

- Circular Layout **301**

- Grid Layout **317**

- Hierarchical Layout **166**

- Link Layout **229**

- Multiple Layout **368**

- Random Layout **265**

- Recursive Layout **355**

- Topological Mesh Layout **64**

- Tree Layout **103**

- Uniform Length Edges Layout **85**

- supported by layout algorithms (table) **36**

percentage of completion parameter

- Hierarchical Layout **165**

- Multiple Layout **366**

- Random Layout **263**

- Recursive Layout **345**

- Tree Layout **102**

performLayout method

- IlvGraphLayout class **22, 330, 419, 432**

- IlvLabelLayout class **379**

point label descriptor **398**

- positioning

- at a point **399**

- at an obstacle **398**

- on multiple criteria **400**

- starting from an empty descriptor **401**

point labeling problem **398**

polyline label descriptor

- full positioning

- at a polyline obstacle **405**

- simple positioning

- at a polyline obstacle **403**

- starting from an empty descriptor **406**

- port index parameter, Hierarchical Layout **178**
- port sides parameter, Hierarchical Layout **176**
- position index parameter, Hierarchical Layout **203**
- position parameter, Tree Layout **104**
- preferred length parameter, Uniform Length Edges Layout **85**
- preserve fixed links parameter
 - Bus Layout **275**
 - Circular Layout **300**
 - Hierarchical Layout **165**
 - in `IlvGraphLayout` **46**
 - Link Layout **228**
 - Random Layout **263**
 - Topological Mesh Layout **62**
 - Tree Layout **102**
 - Uniform Length Edges Layout **84**
- preserve fixed nodes parameter
 - Bus Layout **275**
 - Circular Layout **300**
 - Grid Layout **316**
 - Hierarchical Layout **165**
 - Random Layout **263**
 - Topological Mesh Layout **62**
 - Tree Layout **102**
 - Uniform Length Edges Layout **84**

Q

- questions and answers **432**

R

- radial layout mode (Tree Layout)
 - adding an invisible root node **139**
 - alternating radial mode **135**
 - aspect ratio parameter **137**
 - description **131**
 - evenly spaced first circle **139**
 - setting a maximal children angle **140**
 - spacing parameters **138**
- random generator seed value parameter
 - Random Layout **263**
- Random Layout
 - applicable graph types **260**
 - CSS sample **262**
 - description **262**
 - features **261**
 - generic parameters **263**
 - limitations **261**
 - link style parameter **265**
 - sample drawing **260**
 - specific parameters **265**
- Recursive Layout
 - accessing all sublayouts **353**
 - advanced recursion **334**
 - allowed time parameter **345**
 - applying the same layout **330**
 - combining multiple and recursive layout **371**

- convenience mechanism of reference layout mode **353**
- definition **338**
- features **344**
- generic parameters **345**
- internal provider mode **340, 341, 350**
- Java code samples
 - different layout styles **340**
 - layout providers for experts **358**
 - same layout style **338**
 - specified layout provider **341**
- layout modes **347**
- layout parameters **332**
- layout providers for experts **358**
- mixing different in nested graph **334**
- order of layouts **329**
- percentage of completion parameter **345**
- reference layout mode **338, 349**
- simple **330**
- specific parameters **355**
- specified provider mode **341, 352**
- stop immediately parameter **345**
- Recursive Multiple Layout **371**
- reference layout mode
 - convenience mechanism (Recursive Layout) **353**
 - Recursive Layout **338, 349**
- refining a graph layout **70**
 - using fixed nodes parameter **70**
 - using layout region parameter **72**
 - using outer cycle parameter **72**
- related obstacles (Label Layout) **398**
- relative position constraints, Hierarchical Layout **194, 204**
- reports
 - graph layout **417**
 - Label Layout **382**
- respect node sizes parameter
 - Tree Layout **124**
 - Uniform Length Edges Layout **86**
- ring topology, Circular Layout **295**
- root clusters parameter, Circular Layout **303**
- root node parameter, Tree Layout **103**
 - additional options **103**

S

- same shape for multiple links parameter, Link Layout (short link mode) **250**
- self-link style parameter, Link Layout (short link mode) **246**
- semi-automatic layout **13, 32**
- `setAlignment` method
 - `IlvTreeLayout` class **114, 115**
- `setAspectRatio` method
 - `IlvTreeLayout` class **137**
- `setBranchOffset` method

IlvTreeLayout class **123, 138**
 setBus method
 IlvBusLayout class **278**
 setConnectorStyle method
 IlvTreeLayout class **119**
 setDestinationPointMode method
 IlvHierarchicalLayout class **173**
 IlvLinkLayout class **233**
 setEastWestNeighboring method
 IlvTreeLayout class **147**
 setExteriorCycleId method
 IlvTopologicalMeshLayout class **67**
 setFirstGraphLayout method
 IlvMultipleLayout class **370**
 setFlowDirection method
 IlvTreeLayout class **111**
 setGlobalAlignment method
 IlvTreeLayout class **113, 115**
 setGlobalDestinationPointMode method
 IlvHierarchicalLayout class **173**
 IlvLinkLayout class **233**
 setGlobalLinkStyle method
 IlvHierarchicalLayout class **170**
 IlvLinkLayout class **231, 238, 249**
 IlvTreeLayout class **117**
 setGlobalOriginPointMode method
 IlvHierarchicalLayout class **173**
 IlvLinkLayout class **233**
 setHorizontalAlignment method
 IlvGridLayout class **320**
 setInvisibleRootUsed method
 IlvTreeLayout class **139**
 setLabelLayout method
 IlvMultipleLayout class **370**
 setLayoutMode method
 IlvTreeLayout class **105, 110, 128, 132, 136**
 setLayoutOfConnectedComponents method
 IlvGraphLayout class **425**
 setLayoutOfConnectedComponentsEnabled method
 IlvGraphLayout class **425**
 IlvTreeLayout class **139**
 setLevelAlignment method
 IlvTreeLayout class **130**
 setLinkClipInterface method
 IlvGraphLayout class **122, 174, 308**
 setLinkConnectionBoxInterface method
 IlvGraphLayout class **121, 174, 309**
 setLinkStyle method
 IlvHierarchicalLayout class **170**
 IlvLinkLayout class **239, 249**
 IlvTreeLayout class **117**
 setOriginPointMode method
 IlvHierarchicalLayout class **173**
 IlvLinkLayout class **233**
 setOrthForkPercentage method
 IlvTreeLayout class **124**
 setOverlapPercentage method
 IlvTreeLayout class **124**
 setParentChildOffset method
 IlvTreeLayout class **123, 138**
 setPosition method
 IlvTreeLayout class **104**
 setPreferredLayout method
 IlvDefaultLayoutProvider class **358**
 setRoot method
 IlvTreeLayout class **103**
 setRootPreference method
 IlvTreeLayout class **103**
 setSecondGraphLayout method
 IlvMultipleLayout class **370**
 setSiblingOffset method
 IlvTreeLayout class **123, 138**
 setSpecNodeLevelIndex method
 IlvHierarchicalLayout class **199**
 setSpecNodePositionIndex method
 IlvHierarchicalLayout class **203**
 setTerminationPointFilter method
 IlvLongLinkLayout class **256**
 setting a maximal children angle, Tree Layout **140**
 setting even spacing for the first circle, Tree Layout **139**
 setting invisible root node parameter, Tree Layout **139**
 setTipOverBranchOffset method
 IlvTreeLayout class **123**
 setTowardsPreferredPosition method
 IlvAnnealingLabelDescriptor class **411**
 setUseDefaultParameters method
 IlvGraphLayout class **50**
 setVerticalAlignment method
 IlvGridLayout class **320**
 setVerticalOffsetToPreviousLevel method
 IlvBusLayout class **288**
 setWestEastNeighboring method
 IlvTreeLayout class **147**
 short link mode (Link Layout)
 algorithm description **226**
 bypass distance parameter **251**
 connector style parameter **238**
 features **224**
 link box connection interface **252**
 link crossing penalty parameter **251**
 link offset parameter **237**
 minimum final segment parameter **238**
 number of optimization iterations **246**
 same shape for multiple links parameter **250**
 self-link style parameter **246**

- spacing parameters **237**
- side association for polyline label descriptors **404**
- side-by-side constraints, Hierarchical Layout **205**
- simulated annealing **394**
- spacing parameters
 - Hierarchical Layout **183**
 - Link Layout (long link mode) **240**
 - orthogonal fork percentage (Tree Layout) **124**
 - overlap percentage (Tree Layout) **124**
 - Tree Layout (free mode) **123, 124**
 - Tree Layout (radial mode) **138**
- specified provider mode
 - Recursive Layout **341, 352**
- star center parameter, Circular Layout **303**
- star topology, Circular Layout **295**
- static layout **13**
- stop immediately parameter
 - Bus Layout **275**
 - Circular Layout **300**
 - Grid Layout **316**
 - Hierarchical Layout **165**
 - Link Layout **228**
 - Multiple Layout **366**
 - Random Layout **264**
 - Recursive Layout **345**
 - Topological Mesh Layout **63**
 - Tree Layout **102**
 - Uniform Length Edges Layout **84**
- sublayouts
 - Recursive Layout **353**
- supportsAllowedTime method
 - IlvGraphLayout class **46, 50**
- supportsLayoutOfConnectedComponents method
 - IlvGraphLayout class **425**
- supportsLayoutRegion method
 - IlvGraphLayout class **432**
- swim lane constraint, Hierarchical Layout **209**
- symmetries layout criteria **12**

T

- time, stop computation algorithms **39**
- tip-over alignment, Tree Layout (free mode) **115**
- tip-over layout modes (Tree Layout)
 - aspect ratio parameter **142**
 - description **142**
 - tip leaves over **143**
 - tip over fast **144**
 - tip roots and leaves over **144**
 - tip roots over **143**
- Topological Mesh Layout
 - applicable graph types **57**
 - application domains **57**
 - description of algorithm **59**
 - features **58**
 - generic parameters **61**
 - limitations **58**

- link clipping parameter **62, 73**
- link connection box parameter **62**
- link style parameter **64**
- nodes placement algorithm **65**
- nodes placement allowed time parameter **64**
- nodes placement iterations parameter **64**
- optimization allowed time parameter **64**
- optimization iterations parameter **64**
- outer cycle parameter **67**
- refining a layout **70**
- sample drawings **56**
- specific parameters **64**
 - using fixed nodes parameter to refine **70**
 - using layout region parameter to refine **72**
 - using outer cycle parameter to refine **72**
- Tree Layout
 - adding an invisible root node (radial mode) **139**
 - algorithm description **99**
 - alternating radial mode **135**
 - applicable graph types **95**
 - application domains **96**
 - aspect ratio parameter (tip-over mode) **142**
 - aspect ration parameter **137**
 - compass directions **105**
 - connector style parameter **119**
 - evenly spaced first circle (radial mode) **139**
 - features **97**
 - flow direction parameter **111**
 - free layout mode **109**
 - generic parameters **101**
 - global link style parameter **117**
 - individual link style parameter **117**
 - interactive editing **150**
 - layout modes **105**
 - level alignment parameter **130**
 - level layout mode **127**
 - limitations **97**
 - link clipping parameter **102, 122**
 - link connection box parameter **102, 121**
 - link style parameter **117**
 - making incremental changes **149**
 - orthogonal fork percentage **124**
 - overlap percentage parameter **124**
 - position parameter **104**
 - radial layout mode **131**
 - respect node sizes **124**
 - retrieving link categories **148**
 - retrieving list of root nodes used by algorithm **103**
 - retrieving list of specified root nodes **103**
 - root node parameter **103**
 - additional options **103**
 - sample drawings **94**

- setting a maximal children angle (radial mode) **140**
- setting a root node **103**
- spacing parameters (free mode) **123**
- spacing parameters (radial mode) **138**
- specific parameters **103**
- specifying east-west neighbors **147**
- specifying root node preference **103**
- specifying the order of children **150**
- tip-over alignment (free mode) **115**
- tip-over layout modes **142**

U

- Uniform Length Edges Layout
 - additional node repulsion weight **88**
 - applicable graph types **79**
 - application domains **79**
 - description **82**
 - features **81**
 - force fit to layout region parameter **86**
 - generic parameters **83**
 - limitations **81**
 - link clipping parameter **83, 91**
 - link connection box parameter **84**
 - link length weight **88**
 - link style parameter **85**
 - maximum allowed move per iteration **88**
 - node distance threshold **90**
 - number of iterations parameter **85**
 - preferred length parameter **85**
 - respect node sizes parameter **86**
 - sample drawings **76**
 - specific parameters **85**
- use default parameters
 - in `IlvGraphLayout` **50**

V

- variable end point parameters, Link Layout (long link mode) **256**
- vertical offset parameter, Bus Layout **288**
- vertical offset to previous level parameter, Bus Layout **288**

W

- width adjusting
 - Bus Layout **281**