JGraph User Manual

For JGraph Version 5.13.0.0 – 25th September 2009

We are always interested in feedback on JGraph products, if you have any questions please feel free to contact us using any of the following methods:

Post :	JGraph Ltd. 35 Parracombe Way, Northampton NN3 3ND U.K.
Telephone:	+44 (0)20 8144 9116
Fax:	+44 (0)870 762 4282
Internet:	http://www.jgraph.com/contact.html for private contact or http://www.jgraph.com/forum for community discussion
Email:	info_nospam@jgraph.com , remove the _nospan suffix.

Table of Contents

1 Introduction

JGraph is a mature, feature-rich open source graph visualization library written in Java. JGraph is written to be a fully Swing compatible component, both visually and in its design architecture. JGraph can be run on any system supporting Java version 1.4 or later.

1.1 What does JGraph do?

JGraph provides a range of graph drawing functionality for client-side or server-side applications. JGraph has a simple, yet powerful API enabling you to visualize, interact with, automatically layout and perform analysis of graphs. The following sections define these terms in more detail.

Example applications for a graph visualization library include; process diagrams, workflow and BPM visualization, flowcharts, traffic or water flow, database and WWW visualization, networks and telecoms displays, mapping applications and GIS, UML diagrams, electronic circuits, VLSI, CAD, financial and social networks, data mining, biochemistry, ecological cycles, entity and cause-effect relationships and organisational charts.

JGraph, through it's programming API, provides the means to configure how the graph or network is displayed and the means to associate a context or metadata with those displayed elements.

1.2 What is a Graph?

JGraph visualization is based on the mathematical theory of networks, graph theory. If you're seeking Java bar *charts*, pie *charts*, Gantt *charts*, have a look at the JFreeChart project instead.

A graph consists of vertices, also called nodes, and of edges (the connecting lines between the nodes). Exactly how a graph appears visually is not defined in graph theory. The term *cell* will be used throughout this manual to describe an element of a graph, either edges or vertices.

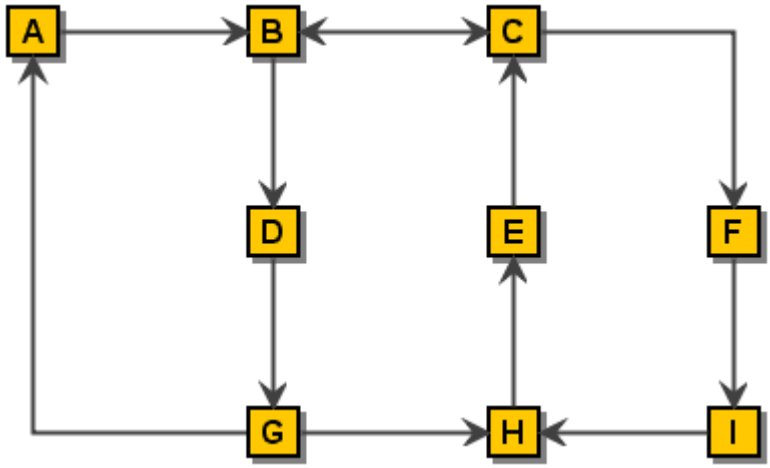

Illustration 1 : A simple Graph

There are additional definitions in graph theory that provide useful background when dealing with graphs, they are listed in Appendix A if of interest to you.

1.2.1 GRAPH VISUALIZATION

Illustration 2 : Graph Visualization of a transport system. (c) Tourizm Maps 2003, http://www.world-maps.co.uk

Visualization is the process of creating a useful visual representation of a graph. The scope of visualization functionality is one of JGraphs' main strength. JGraph supports a wide range of features to enable the display of cells to only be limited by the skill of the developer. Vertices may be shapes, images, other Swing components (including other JGraphs), animations, virtually any graphical operations available in Swing.

1.2.2 GRAPH INTERACTION

Interaction is the way in which an application using JGraph can alter the graph model through the application GUI. JGraph supports dragging and cloning cells, re-sizing and re-shaping, connecting and disconnecting, drag and dropping from external sources, editing cell labels in-place and more. One of the key benefits of JGraph is the flexibility of how interaction can be programmed.

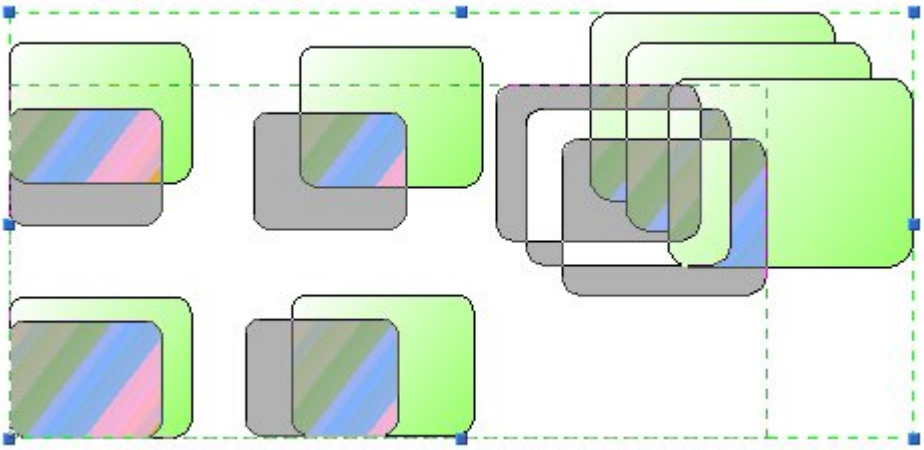

Illustration 4 : Live-Preview of a graph resize drag

1.2.3 GRAPH LAYOUTS

Graph cells can be drawn anywhere in a simple application, including on top of one another. Certain applications need to present their information in a generally ordered, or specifically ordered structure. This might involve ensuring cells do not overlap and stay at least a certain distance from one another, or that cells appear in specific positions relative to other cells, usually the cells they are connected to by edges. This activity, called the layout application, can be used in a number of ways to assist users set out their graph. For non-editable graphs, layout application is the process of applying a layout algorithm to the cells. For interactive graphs, these that can be edited, layout application might involve only allowing users to make changes to certain cells in certain positions, to re-apply the layout algorithm after each change to the graph, or to apply the layout when editing is complete.

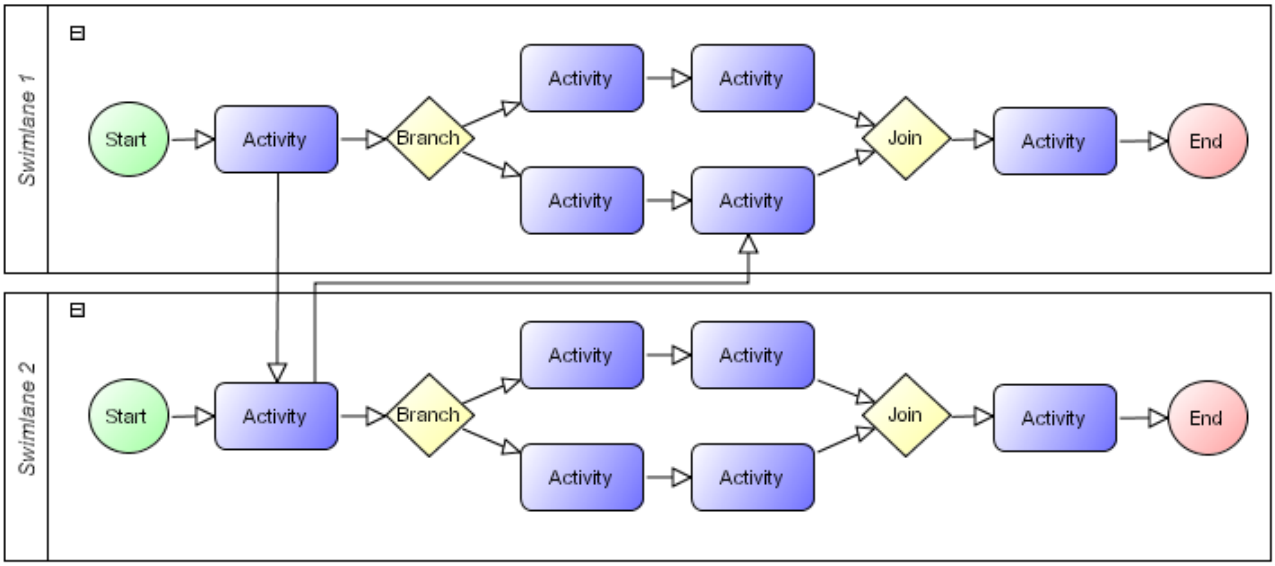

Illustration 5 : Layout of a workflow using the hierarchical layout in JGraph.Layout

Com.jgraph.layout is the supported layout package within the JGraph core, designed for speed, API stability, functional flexibility and consistency. They support a range of tree, force-directed and hierarchical layouts which will fit most layout needs, see the later Chapters for details.

1.2.4 GRAPH ANALYSIS

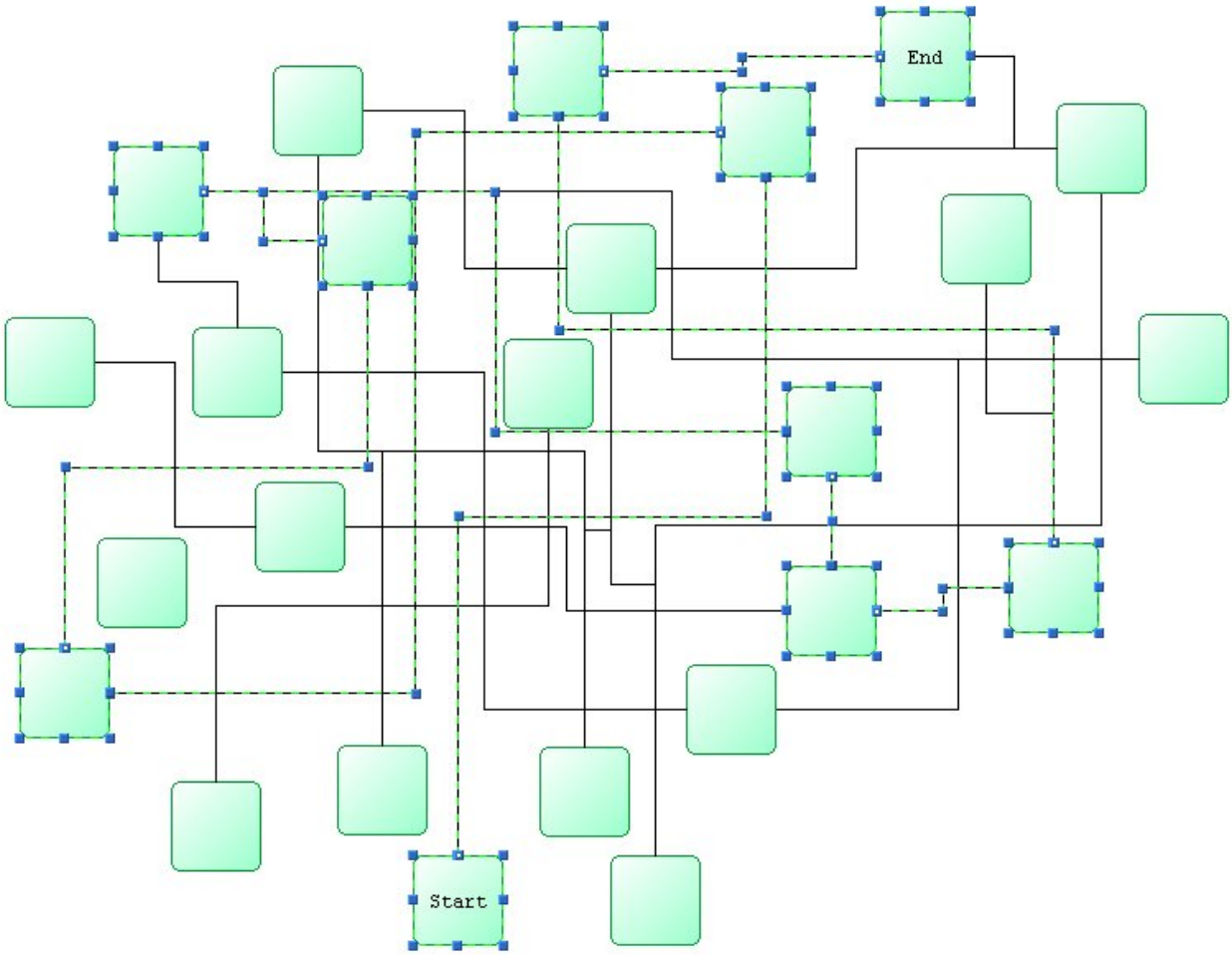

Illustration 6 : Shortest Path Analysis

Analysis of graphs involves the application of algorithms determining certain details about the graph structure, for example, determining all routes or the shortest path between two cells. There are more complex graph analysis algorithms, these being often applied in domain specific tasks. Techniques such as clustering, decomposition, and optimization tend to be targeted at certain fields of science and currently have not been implemented in the core JGraph packages. However, a number of generic performance optimized analysis algorithms can be found in the JGraph.Layout package.

1.3 About this Manual

1.3.1 PRE-REQUISITES FOR THIS MANUAL

To benefit fully from this manual you will need to have a reasonable understanding of Java and at least a high-level overview of Swing. Not all aspects of Swing are required, but knowledge of the Swing MVC pattern is important, in particular how the renderer components are used. It would also be useful to study one of the major Swing components in more detail, in particular the JTree class, since JGraph is similar to JTree in a number of ways at a design level.

If you lack experience with programming the Java language, there are many good books on the subject available. A useful free introduction is the Sun Java Tutorial.

1.3.2 GETTING ADDITIONAL HELP

There are many mechanisms for receiving help for working with the JGraph software. The community help forum provides free assistance to JGraph users. The forums combine the advantages of many users helping to answer questions along with the guidance of active JGraph developers ensuring the quality and correctness of responses and that as many questions as possible are answered. However, there is no assurance of getting free assistance, either the answer being correct, or getting an answer at all.

When posting at the forums please read these posting guidelines, following these will help you get a better answer and encourage more people to help you. Please remember people helping you on the forums are giving up their free time to do so, but note that the JGraph team cannot guarantee that answers provided on the forums are correct as they cannot always monitor all discussion threads. If you require guaranteed response time support please contact sales_nospam@jgraph.com for support contract information. Purchased JGraph products all come with 30 days technical support, you also have the option of a 12 months support package.

Please do not privately contact JGraph developers asking for free support. It is unfair to expect special treatment and puts them in the awkward position of asking you to re-post your question on the forums. Answering the question on the forums means other people can read the thread and solve their problem without having to take more developers' time.

1.4 About JGraph

1.4.1 JGRAPH SWING COMPATIBILITY

JGraph complies with all of Swings standards, such as pluggable look and feel, data-transfer, accessibility, internationalization and serialization. For more advanced features such as undo/redo, printing and XML support, the standard Swing designs were also used. The design of JGraph has much in common with that of JTree and the view concepts comes from Swings text components. JGraph itself is an extension of JComponent, which is Swings base class for all components. JGraph also complies with the Java conventions for method and variable naming, source code layout and javadocs comments.

1.4.2 THE JGRAPH PACKAGES

There are three separate packages available from JGraph.com.
The main package is **JGraph** itself which comprises the basic JGraph swing component:

Java Package Name	Functionality
org.jgraph	Basic JGraph class
org.jgraph.event	Graph Event Models
org.jgraph.graph	Graph Structure and nodes
org.jgraph.plaf	Graph UI delegate component
org.jgraph.util	General purpose utilities
com.jgraph.algebra	Graph Analysis Routines
com.jgraph.layout	JGraph Facade and utilities
com.jgraph.layout.organic	Force directed layouts
com.jgraph.layout.tree	Tree layouts
com.jgraph.layout.routing	Edge routing algorithms
com.jgraph.layout.hierarchical	Hierarchical layouts

Table 1 : JGraph Packages

1.4.3 MXGRAPH

mxGraph is a browser based graph library for all major platforms. mxGraph uses the native vector graphics drawing language available to provide rich diagramming functionality in a thin client architecture. mxGraph also includes back-end functionality for .NET, PHP and Java that provide access to the graph model and persistence across the majority of server technologies. The software is only available under the terms of the mxGraph License, a standard commercial license. Evaluations are available on the mxGraph web site.

1.4.4 JGRAPH LICENSING

The core JGraph library is open source software. This means the source code is freely available. The licensing of the various components at the time of writing is:

- JGraph – Simplified (modern) 3 clause BSD license and JGraph License version 1.1.

For detailed licensing question you are always advised to consult a legal professional.

1.5 Getting Started

1.5.1 THE JGRAPH WEB SITE

To start with navigate to the JGraph web site. The most useful areas to you when starting JGraph are listed below. Use the navigation bar on the left hand side to locate the appropriate section:

- Documentation - All freely available documents relating to JGraph. If you are reading this as part of the JGraph user manual, this is the most up-to-date documentation at the time of writing. Additional examples to JGraph are available at a small cost that demonstrate specific features within JGraph.
- Forum - Here you can ask the JGraph community your questions. A timely and correct answer cannot be guaranteed, however the JGraph developers tend to keep a close eye on questions posted. Try to break your problem down into single smaller questions. If you post asking to have someone write your project for you, you are unlikely to receive a reply. If you require commercial-level support please contact support@jgraph.com. Before posting to the forum please search the documentation, the FAQs and search the forum using the search facility provided. The JGraph team have spent a great deal of effort putting those resources in place, please try to save them having to point you at them because you have not searched yourself.
- FAQ– The FAQ contains a number of the question received most often in a summary format.
- Tracker - The tracker contains current bugs within JGraph. If you think you have a bug, check it has not already been reported and also check in the forum if you are unsure if it is a real bug. If you are sure, please do report the bug.

1.5.2 DOWNLOADING JGRAPH

On the downloads page on JGraph web site you will find the latest free packages available. Older versions of JGraph are available from the archive site.

1.5.3 INSTALLING JGRAPH

Having downloaded the three JGraph packages select a folder that will be the root development folder somewhere on your hard disk. JGraph is delivered in a self-extracting .jar file. Double-clicking the file in Windows will usually start the installation process. To start the installation from the command line type:

```
>java -jar jgraph-5_12_3_0-src.jar
```

replacing the .jar filename as appropriate. A dialog will first appear asking you to agree to the license under which you will use JGraph, you are advised to read the license. Next, the installation process will prompted you to select the directory to install JGraph into.

1.5.4 PROJECT STRUCTURE AND BUILD OPTIONS

Once Java and Ant are installed launch the command prompt on windows, or shell terminal on *nix or Mac, navigate to the root folder where you installed JGraph. Typing `ant command`, where `command` is one of the targets in the ant build file, will perform the function of that command, as described below.

doc/	Documentation root
src/	Source root
examples/	Examples root
build/	Build environment

Table 1. Project Directory Structure

all	Clean up and produce all distributions (*the default target)
apidoc	Generate the API specification (javadoc)
build	Run all tasks to completely populate the build directory
clean	Delete all generated files and directories
compile	Compile the build tree
compile-example	Compile the main example
dist	Produce fresh distributions
distclean	Clean up the distribution files only
doc	Generate all documentation
example	Run the main example
init	Initialize the build
jar	Build all Java archives (JARs)
generate	Generate the build tree

Table 2. Ant command options

For example, to compile and run the example UI type the following:

```
ant example
```

2 JGraph and the Graph Model

2.1 Important Application Note

From Java 6 update 10, XOR is broken on the Sun Windows implementation, because D3D (Direct3D) is on by default from this build onward. This means that applications that use JGraph (which by default uses XOR painting) need to turn the D3D flag off at the start of the application. There is nothing that can be done in the library to correct this. You can find more details on the JGraph XOR painting problem forum thread. Java 6 update 14 fixed this problem.

Unless the extra size caused by bundling a specific JRE with your application is an issue, bundling should be used to provide users with a consistent experience.

2.2 Understanding the HelloWorld application

In this chapter we will walk through each line of a simple Hello World application and explain the main classes being used and the primary API used to create and manipulate a simple graph. The package statement and imports are omitted, it is assumed you are familiar with the basics of Java:

```
public class HelloWorld {
     public static void main(String[] args) {
          GraphModel model = new DefaultGraphModel();
          GraphLayoutCache view = new GraphLayoutCache(model,
                                          new
DefaultCellViewFactory());
          JGraph graph = new JGraph(model, view);

          DefaultGraphCell[] cells = new DefaultGraphCell[3];

          cells[0] = new DefaultGraphCell(new String("Hello"));

          GraphConstants.setBounds(cells[0].getAttributes(), new
               Rectangle2D.Double(20,20,40,20));

          GraphConstants.setGradientColor(
cells[0].getAttributes(),
                                               Color.orange);
          GraphConstants.setOpaque(cells[0].getAttributes(), true);

          DefaultPort port0 = new DefaultPort();
          cells[0].add(port0);

          cells[1] = new DefaultGraphCell(new String("World"));

          GraphConstants.setBounds(cells[1].getAttributes(), new
               Rectangle2D.Double(140,140,40,20));
```

```
            GraphConstants.setGradientColor(
cells[1].getAttributes(),
                        Color.red);
            GraphConstants.setOpaque(cells[1].getAttributes(), true);

            DefaultPort port1 = new DefaultPort();
            cells[1].add(port1);

            DefaultEdge edge = new DefaultEdge();
            edge.setSource(cells[0].getChildAt(0));
            edge.setTarget(cells[1].getChildAt(0));
            cells[2] = edge;

            int arrow = GraphConstants.ARROW_CLASSIC;
            GraphConstants.setLineEnd(edge.getAttributes(), arrow);
            GraphConstants.setEndFill(edge.getAttributes(), true);

            graph.getGraphLayoutCache().insert(cells);

            JFrame frame = new JFrame();
            frame.getContentPane().add(new JScrollPane(graph));
            frame.pack();
            frame.setVisible(true);
      }
}
```

Swing Refresher – Swing uses a containment hierarchy to provide a simple way to put any number of its standard components into an application. The main window of many applications is referred to as a frame (the Swing `JFrame`). It is termed a top-level container and this set includes Dialogs (`JDialog`) and applets (`JApplet`).

Panels (`JPanel`), sometimes called panes, are a typical intermediate container. These assist in positioning a number of basic Swing components as well as offering common features such as scrolling (`JScrollPane`) and tabbed elements (`JTabbedPane`). Top-level containers contain an intermediate container called a content pane, to which visible content is added. In the `HelloWorld` example above, a scrollable pane is added to this content pane.

At the lowest level lies the atomic components. Labels (`JLabel`), tables (`JTable`) and trees (`JTree`) are a few of the many components available in this category. These components provide specific display and/or input functionality and are not necessarily designed to embed further components. `JGraph` is an atomic component, designed to be used in a manner and provide an API that is consistent, as far as possible, with other Swing atomic components. Adding components to containers is either done during the construction of the container, or by calling the `add()` method on the container.

In the `HelloWorld` example, specifically the last 4 lines, a `JGraph` instance is passed to the `JScrollPane` at construction and the whole thing made visible in the content pane of the main frame.

2.2.1 CREATING THE JGRAPH

At the very core of the `JGraph` library is the `org.jgraph.JGraph` class. The `JGraph` class extends `JComponent` and you create one `JGraph` instance per graph component in your application, the same way as you would one `JLabel` for one label. Instances of this class bind the graph model, any graph view(s) and the user interface control handling all together in one place. Creating a `JGraph` instance without any parameters creates an example graph showing a very basic UML diagram of the `JGraph` architecture:

```
public class Example {
     public static void main(String[] args) {
          JGraph graph = new JGraph();
          JFrame frame = new JFrame();
          frame.getContentPane().add(new JScrollPane(graph));
          frame.pack();
          frame.setVisible(true);
     }
}
```

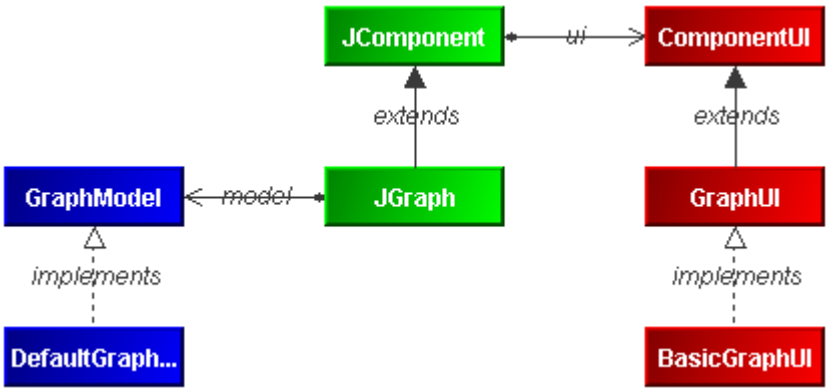

Illustration 7 : Sample data presented on the creation of an empty JGraph

However, this is not very informative for our purposes. Instead, in our example we create a model of the graph, using the default implementation provided, `DefaultGraphModel`. We then pass the `JGraph` constructor this model which represented the data model we wish to use to describe the graph. We also create a default implementation of a view of the graph, the `GraphLayoutCache` and inform the `JGraph` this is the view that will be used (don't worry about the cell view factory for now):

```
          GraphModel model = new DefaultGraphModel();
          GraphLayoutCache view = new GraphLayoutCache(model,
                                       new DefaultCellViewFactory());
          JGraph graph = new JGraph(model, view);
```

Information - The `GraphLayoutCache` is often thought of as the graph view, and in previous versions of `JGraph` was named `GraphView`. The reason for the term layout cache is that `JTree` has a class named `AbstractLayoutCache` that holds information about the geometry of the tree nodes. The GraphLayoutCache is different to a standard view in

Swing, since it contains information that is solely stored in the view, i.e. it is stateful. It is the term `GraphLayoutCache` we will use from now on when referring to what might be thought of as the graph view.

If the terms model and view are not familiar to you, it is worth getting a basic overview from a text such as [REF]. In simple terms, the model holds the data about the graph and provides various methods to access that data. The view(s) are one or more layers logically above the model that perform the task of visually presenting the graph and these are updated automatically when the model data changes. By default, views will show the same graph, but a variety of functionality is available to display the graph differently in each view, if required. It is possible for the model to contain all the information needed to represent the graphs logical structure, its geometric layout and its visual representation. Some of these aspects would be expected to only be considered in the graph views, but a graph Swing component is somewhat more complex than any of the standard Swing components due to the virtually unlimited flexibility of cell positioning available. JGraph 1.0 did place more weight on storing visual attributes in the views over the model, JGraph 2.0 reversed this, shifting common visual attributes into the model and was found to be the better solution architecturally.

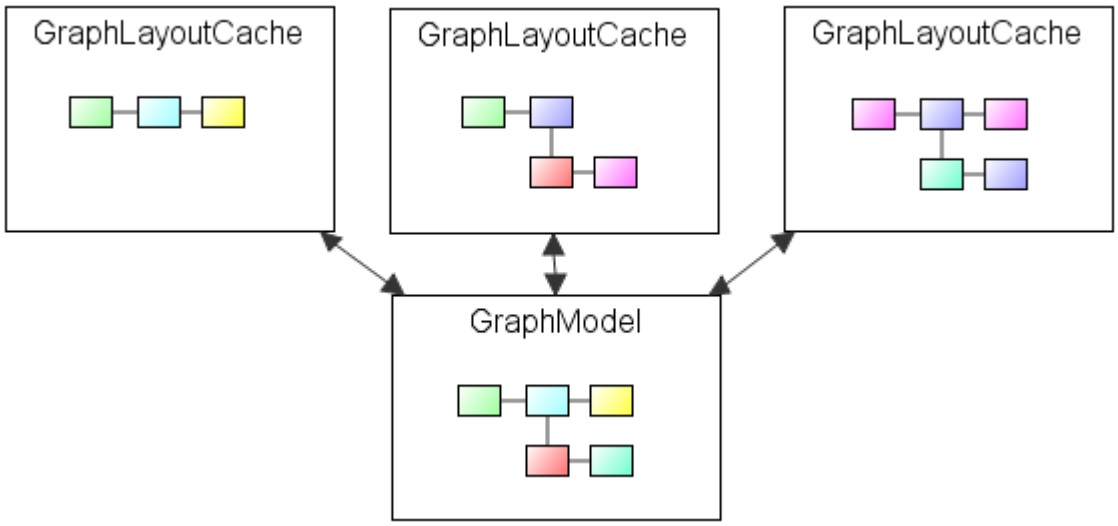

Illustration 8 : Multiple views can share the same model

Information - For simple applications it is tempting to avoid the `GraphLayoutCache` completely and work directly on the `GraphModel`, as the `GraphModel` provides all the necessary methods to manipulate the graph. You are recommended, unless you have a solid technical understanding and a good reason otherwise, to start by always working on the `GraphLayoutCache`. People often find, as their application grows, that view-specific features are required and all the calls to the `GraphModel` have to be changed to calls the `GraphLayoutCache`. One important exception to this principle is that if you `edit()` an invisible cell in the `GraphLayoutCache`, it becomes visible. In this case editing the model directly is preferable. The `GraphLayoutCache` is discussed further in Chapter 4.

In between the first 3 lines of main() that set up our `JGraph` and the last 4 lines that display the `JGraph` in the application, lies the codes that creates the graph cells, configures them and inserts them into the graph. We'll look at them in order.

2.2.2 INSERTING CELLS

The three graph cells we are going to create in the `HelloWorld` application are two vertices and one edge connecting the vertices:

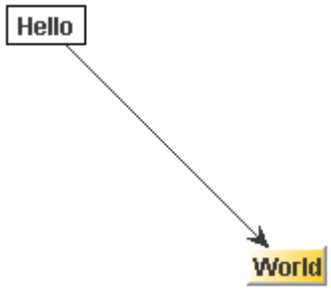

Illustration 9 : The basic Helloworld example shows two vertices connected by one edge

```
cells[0] = new DefaultGraphCell(new String("Hello"));
...
DefaultPort port0 = new DefaultPort();
cells[0].add(port0);
port0.setParent(cells[0]);
...
DefaultEdge edge = new DefaultEdge();
```

We can create new simple vertices by constructing `DefaultGraphCells` and edges with `DefaultEdge`. These classes can be instantiated with no parameters, or with an object. By default, whatever that object returns in its `toString()` method will appear as the text for that vertex or edge. Obviously, `String` objects return themselves in `toString()` and this is sometimes the only object used in this parameter. In the `HelloWorld` example we use this mechanism to assign one vertex the label “Hello” and other vertex the label “World”.

The other object, a `DefaultPort`, might be confusing if you are familiar with graph theory. Ports are an artificial addition in `JGraph` used to indicate places on a vertex where an edge may connected to that vertex. The ends of edges connect to vertices by these ports and ports are represented, at least in the default model provided with JGraph, as being children of one vertex. The `add()` and `setParent()` calls are the mechanism used in `JGraph` in indicate the parent/child relationship between the vertex and its port(s).

Setting up the vertices and edge to display how we would like them is done by modifying their attributes. All cells, including ports, have what is called an attribute map. This is a `java.util.Map`, the JGraph default implementation of an attribute map, `AttributeMap`,

is a subclass of `Map`. Ensure you understand how Java maps operate and their basic API before using attribute maps. Attributes are stored in key/value pairs where the keys are attributes like color, position and text font. It is worth, at this point, you having a look at the `org.jgraph.GraphConstants` class.

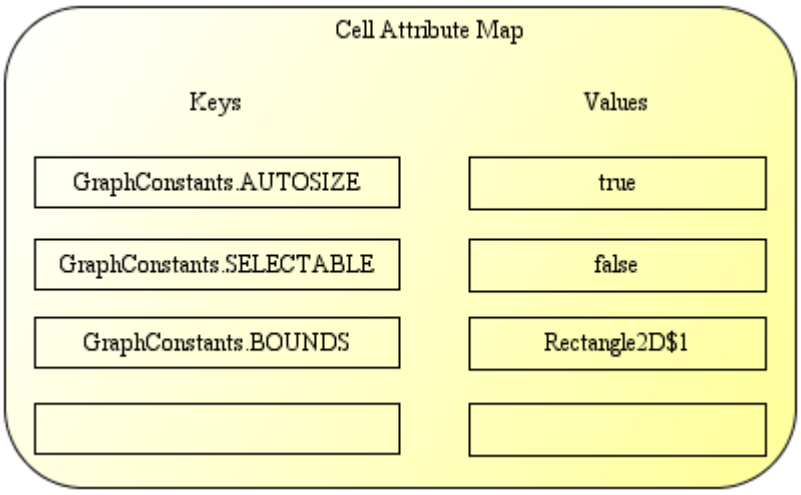

Illustration 10 : Key/Value pairs of a cell attribute map describing the cells visual attributes

`GraphConstants` is a utility class designed to allow you to access attribute maps in a type-safe way, i.e. ensure you are using the correct types of objects for the available attributes. It also provides a useful guide to what attributes can be set for the various cell types. In `GraphConstants`, after some initial enumeration variables, you will find a list of `Strings` that represent the possible keys in attribute maps. The bottom half of the source file, roughly, contains all the accessor methods (`getXXX()` and `setXXX()` methods) that you should use in your application to read and change the attributes. The Javadocs of these methods and key strings are the most up to date and complete description available, repeating them in documents such as this one is avoided as such references quickly become outdated.

2.2.2.1 Configuring Cells' Attributes before Insertion

All graph cells have a storage map that you can obtain using `getAttributes()`. When inserting cells you can obtain the attribute map that belongs to that cell and manipulate it before inserting the cell into the graph. This practice is only generally advised for inserting cells, when editing cell the process of using transport maps, not the actual cell's map (the storage map) should be used (see Editing the Graph). Below is the call, as an example, of setting the gradient color on the first cell to orange. The attribute map from the cell is obtained with `getAttributes()`, the construction of `DefaultGraphCell` ensures you receive a non-null map. Then the appropriate setter method in `GraphConstants` is called passing in the map and the new value to set:

```
GraphConstants.setGradientColor(cells[0].getAttributes(),
                                Color.orange);
```

Another example is:

```
GraphConstants.setBounds(cells[0].getAttributes(), new
     Rectangle2D.Double(20,20,40,20));
```

Cell bounds is something you will come across many times using `JGraph`, in particular the `setBounds()` method when moving any cells in the graph. The bounds of a cell is the minimum rectangle that encloses the cell completely. In the above example a new `double` precision rectangle is created and applied to the cell using the `setBounds()` method. The x,y co-ordinates are set to (20,20), the width of the cell to 40 and the height to 20.

The process of applying attributes to edges is the same, as shown in this example:

```
int arrow = GraphConstants.ARROW_CLASSIC;
GraphConstants.setLineEnd(edge.getAttributes(), arrow);
GraphConstants.setEndFill(edge.getAttributes(), true);
```

Here the line end is set to be a standard arrow and the creation of the end shape for the edge enabled. Note that edges all have a direction internally within `JGraph`, it is up to you whether you want to reveal this directed behaviour on the visible graph. It is also worth noting that the accessor methods frequently only apply to one or a limited number of types of cells. Setting the line end of a vertex is meaningless and nothing will happen because of it, no error is caused by doing such a thing for performance reasons, since no harm will come of it. The Javadocs of the methods state when they only apply to particular cell type(s).

In terms of indicating how the edge is connected, in our example, these are the lines that perform this function;

```
edge.setSource(cells[0].getChildAt(0));
edge.setTarget(cells[1].getChildAt(0));
```

As mentioned, edges have a direction, internally, and connect to vertices by the ports assigned to those vertices. Edges can be viewed as going from a **source** to a **target**. The methods `setSource()` and `setTarget()` on the `Edge` interface specify which ports each end of the edge connects to. In the example, the ports have been obtained from the vertices by asking for their first child. `getChildAt(int)` returns the child at the index specified in the single parameter. We know there is one child attached to each vertex since we created the ports and assigned them as children previously. Note that this method of determining ports is enough for our example, but sometimes isn't good programming practice when we get to non-trivial applications involving multiple ports.

Having created our cells, configured them and connected the edge to the vertices, we can now insert this all into the graph:

```
graph.getGraphLayoutCache().insert(cells);
```

We will always work on the `GraphLayoutCache` in our examples of inserting, editing and removing cells. You will find there a number of variants of the insert method and the one shown is the simplest. It takes an array of vertices and edges and inserts them into the graph. Try running the `HelloWorld` example provided with the `JGraph` package. Details

of how to do this are in the Introduction chapter. The code is slightly different, but the functionality is the same.

Try playing with `HelloWorld` for a few moments to see what simple functionality the `JGraph` library provides to you. Select a vertex and what are called **handles** appear around the vertex. You can drag the handles to resize the vertex, or click and drag the main part of a vertex to move it. Double-click a vertex to bring up a simple editor that allows you to alter the labels, you can do the same for the edge too. Click and hold the mouse down near the top-left of the graph area and drag the mouse towards the bottom-right of the graph and release. The rectangle that is formed during the drag is termed a **marquee**, releasing the mouse causes all three cells to be selected if the marquee completely overlaps the cells. Dragging any part of the selection causes the whole selection to move at once. Functionality related to this marquee is handled by the `BasicMarqueeHandler`.

2.2.3 EDITING GRAPH CELLS

When changing a graph, collect your changes together in one nested map and pass it to `GraphLayoutCache.edit()`. That will sort out the change on your view, pass it to the model, create an undoable edit on your undo command history and refresh everything that needs refreshing. For example:

```
Map nested = new Hashtable();
Map attributeMap1 = new Hashtable();
```

The *nested* map is the map passed into `edit()` as the first parameter. `attributeMap1` contains details of the attributes on a particular cell that we want to edit. Let's say we want to change the `lineColor` of a cell:

```
GraphConstants.setLineColor(attributeMap1 , Color.orange);
```

Again, `GraphConstants` is used to indicate the attribute setting. But there's a difference to the new `HelloWorld` example here. Instead of fetching the attribute map belonging to any one map, a new `Hashtable` has been constructed. Why this is different to manipulating an attribute map during an insert will be explained shortly.

You can create attribute maps describing the attribute changes for any number of cells. Each attribute map describes all the changes for one cell. It you split the changes for one cell across many maps, this would still work but be inefficient. The next step for our attribute map, assuming we only want to set its color, is to `put()` the attribute map into the nested map. When doing this you provide the cell you want to alter as the key to the attribute map, i.e. *this* cell is getting *these* attribute changes.

```
nested.put(cell1, attributeMap1 );
```

You don't have to call `edit()` with the nested map just yet, in fact it might be a bad idea to do so. Calling `edit()` adds that edit to the undo history, so if you want a number of things to be grouped into one undo, make sure they are performed as part of one `edit()`.

So, maybe you want to make the label on another edge to lie flat along the edge and this to be part of the same atomic change;

```
Map attributeMap2 = new Hashtable();
GraphConstants.setLabelAlongEdge(attributeMap2 , true);
nested.put(cell2, attributeMap2 );
```

And so on. Finally we pass the nested map into `edit()` and you should find the resulting graph is changed accordingly.

```
graph.getgraphLayoutCache().edit(nested, null, null, null);
```

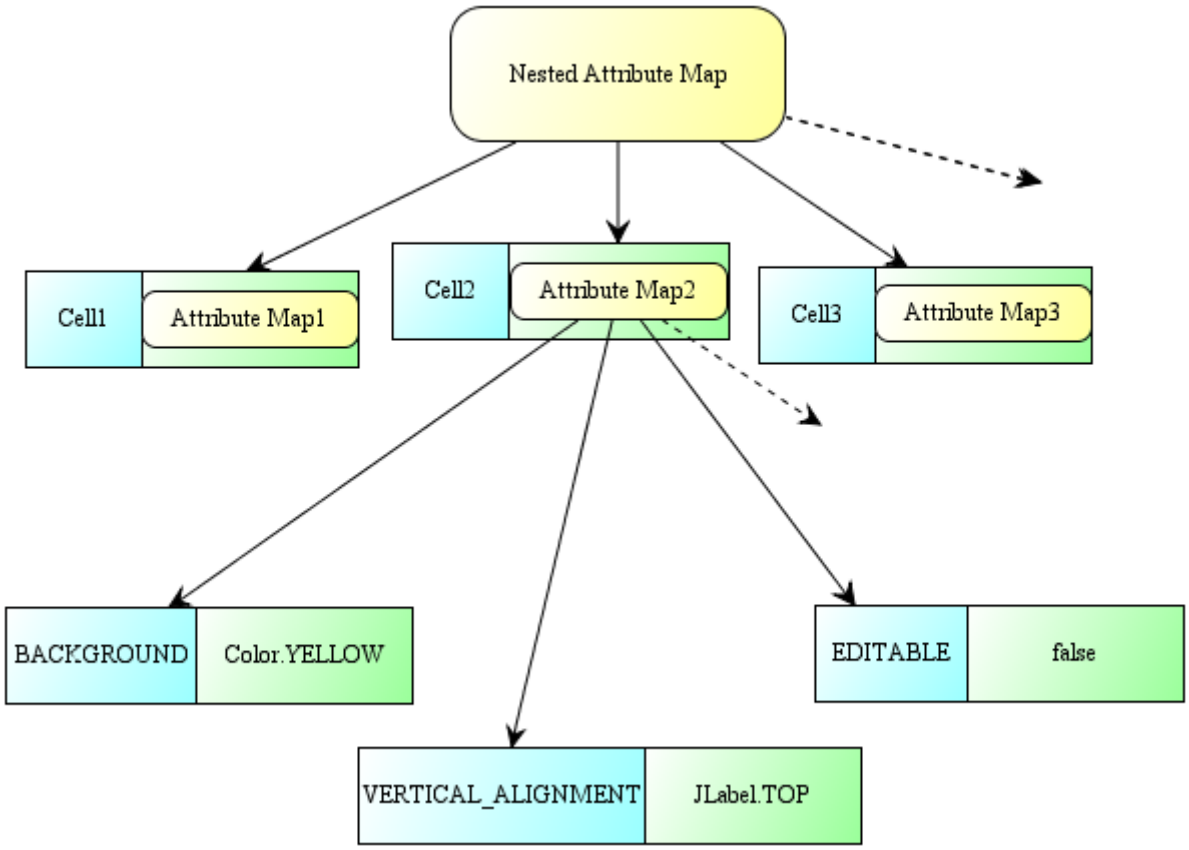

Illustration 11 : Representation of a nested attribute map passed into an edit call. The entries into the nested map are key/value pairs representing the cell to be changed and a map of attributes to change in that cell. Within that second attribute map are a set of key/value pairs representing keys from the GraphConstants class and the new values that those visual attributes are to be assigned by this edit call.

When editing you should not edit the attributes of a cell directly, you should store the changes in a new map and ask `JGraph` to apply them for you. This mechanism was not necessary when inserting because the cell(s) have no existing attribute map to be altered. When an insert, edit or remove call is made, the graph model creates an object that describes the changes that are to be made, this object is called an **edit**. This edit is executed on the current state of the graph to determine the resulting graph. The reasons for abstracting the change into an actual object is two-fold: 1) to provide listeners of the event that executing

the edit fires a means to obtain information as to what happened in the edit, 2) to provide undo support within JGraph by storing the edit on the undo history.

`edit()` checks the requested graph state changes requested against the current graph state. If there is found to be no change requested then no action is taken. If you edit the attributes in-place on the cells attribute map before an edit, the attribute maps passed in will be checked against those currently held by the cells and found to be the same. This is because they will be the same object and so `edit()` does not change the graph since it sees nothing different in the change request. The reason this process of creating new map to pass into `edit()` calls isn't necessary for `insert()` calls is that for inserts the cell doesn't exist in the graph and so there is no attribute map comparison to be done. If you dislike having two different methods (the simple insert and nested map) of configuring attributes, the use of nested hashtables is possible with both methods. However, editing in-place on inserts provides better performance. The corresponding call to insert would be:

```
graph.getGraphLayoutCache().insert(nested, null, null, null);
```

There are a couple of items of terminology used for attribute maps. The permanent map associated with a cell is called a **storage map** and requires the use of a specialized attribute map class. A temporary map used only to indicate an edit change and then discarded is called a **transport map,** most generic `Map` implementations can be used for this.

2.2.3.1 Removing Cell Attributes

A common mistake in JGraph is to resort to using a cells attribute map directly because the mechanism to completely remove an attribute from an attribute map is not so obvious. As a result, users get the map directly, remove the appropriate key and call `edit()`. The correct way to do this is to call `setRemoveAttribute()`:

```
Object[] keys = new Object[] { GraphConstants.ICON };
GraphConstants.setRemoveAttributes(map, keys);
```

This example removes the icon key from a cells attribute map. The possible set of keys you can pass in with the array are at the top of the `GraphConstants` class. Remember to set all the removed attributes at once, as any new array will overwrite previous entries. Alternatively, fetch the array using `getRemoveAttributes()`, copy the previous values into a new array whilst also adding the new values and pass the new array to the `setRemoveAttributes()` method.

2.2.4 REMOVING CELLS

The remaining basic graph editing operation is that of removing cells. The simplest `remove()` method takes an array of cells to be removed. Like `insert()` and `remove()`, this method is available at both the model and layout cache levels. Special consideration needs to be given when removing grouped cells, see Chapter 4 for more details.

2.2.5 ATTRIBUTE MAPS

The map of attributes that each cell holds is termed an attribute map. The default class within `JGraph` for defining attribute maps is named `AttributeMap`, but always try to access attribute maps using the `Map` interface for the usual reasons of encapsulation and de-coupling of the interface from the implementation. Attributes are held within the values of the key/value pairs in the map and the keys are well-known constants that the drawing functionality understands and interprets the values to produce graphics configured as the attributes dictate.

Previously mentioned was the use of the `GraphConstants` class to provide definitions of the map keys that the default `JGraph` implementation understands and to provide a way to access the values in a typesafe manner. For example, the implementations of `setFont()` and `getFont()` in `GraphConstants` look like:

```
public static void setFont(Map map, Font font) {
     map.put(FONT, font);
}

public static Font getFont(Map map) {
     Font font = (Font) map.get(FONT);
     if (font == null)
          font = DEFAULTFONT;
     return font;
}
```

Note that the methods are static, you specify the `Map` they are to act upon in the parameter list. These methods ensure that the type of the value object stored under the key `GraphConstants.FONT` is a `Font`. In the case of `getFont()` the method also ensures that a default font is used if any particular cell does not have a font set. In another part of the JGraph library, the part that deals with drawing labels on vertices and edges, the value of `getFont()` will be obtained by passing in the attribute map of corresponding cell and used to render the label correctly.

It should be noted that keys of attribute maps defined in `GraphConstants` relate almost entirely to visual properties of cells. In general in `JGraph`, if the user would like to add new attributes then only visual attributes (color for example) and visual control attributes (selectable for example) should be added to a custom class that provides the appropriate key constant, as well as the static `setXXX()` and `getXXX()` methods. One thing to remember is that attributes are undoable, this might affect whether you might it an attribute or associate it with your cell in another way. There is no requirement for this custom class to be a sub-class of `GraphConstants`, since virtually everything in that class is statically defined. The subject of associating custom non-visual data with a cell is covered in chapter 3 in the discussion on cell user objects.

2.2.5.1 Attribute Map changes after edit calls

The standard way to alter the contents of cells' attribute maps is to pass a new map of attributes with the cell in a `Map` entry as part of a nested map to the `edit()` methods. Since

the attribute map of a cell already exists there are four state changes that might happen to individual attributes within the attribute map:

1. The attribute remains unchanged,
2. The attribute is changed,
3. The attribute is removed from the map
4. A new attribute is added to the map.

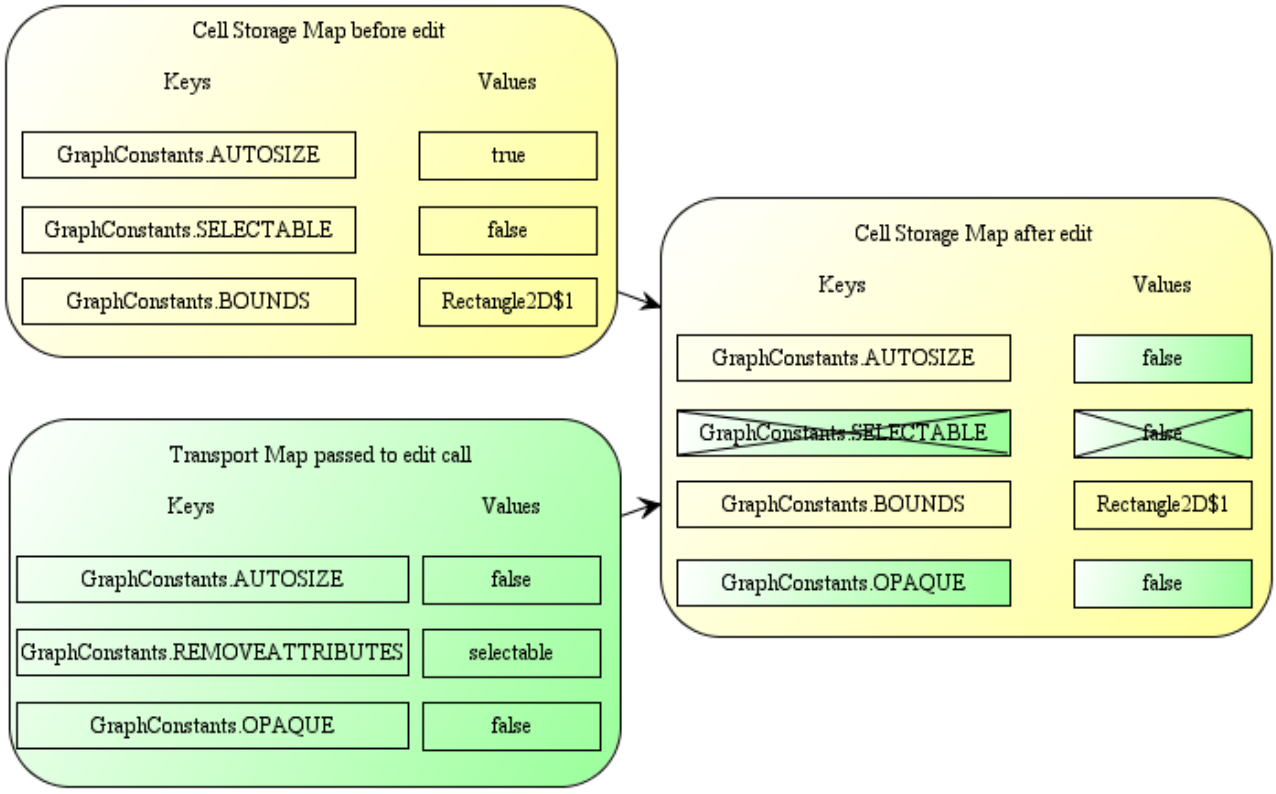

Illustration 12 : How transport maps passed through edit calls affect cell storage maps.

Illustration 14 pictorially shows the four possible attribute entry state changes during an `edit()` call. The yellow box represents the state of the cell attribute map before the edit call and the green box the attribute map passed in the `edit()` call within a nested map.

1. The BOUNDS attribute is not in the transport map and so remains unchanged in the post-edit storage map.
2. The AUTOSIZE attribute is in both the pre-edit storage map and the transport map. In this case the post-edit storage map holds the value passed in through the transport map.
3. The transport map holds an Object array value with the REMOVEATTRIBUTES key. This array has one element, GraphConstants.SELECTABLE which is actually a String of value “selectable”. The edit call checks to see the referenced key is present in the pre-edit storage map. It is in our example and so the map entry is deleted from the post-edit storage map.
4. The OPAQUE attribute is present in the transport map, but not the pre-edit storage map. The key and value pair are copied into the post-edit storage map.

2.2.6 COMPLEX TRANSACTIONS

There are times when you need to group several inserts, edits and/or removes into a single change. The graph model has the concept of transactions as from version 5.12.4.0. Calling `beginUpdate()` on the `GraphModel` increments an internal transaction counter. Calling `endUpdate()` decrements that counter, and if that decrement causes it to reach 0 all the insert, edit and remove calls that were made on the model between the initial begin and last end update are actioned. This produces a single undo and only causes the change events to be fired and the graph to repaint after the final `endUpdate()`.

```
graphModel.beginUpdate(); // count = 1
graphModel.insert(...);
graphModel.beginUpdate(); // count = 2
graphModel.edit(....);
graphModel.endUpdate();   // count = 1
graphModel.remove(...);
graphModel.edit(...);
graphModel.endUpdate();   // count = 0
```

In the above example, not until the last `endUpdate()` call will the 4 model changes be actioned. This allows abstraction of model changes into, for example, methods wrapped by the begin/end update calls, that can then be used as part of a larger transaction.

Note that the transaction applies in the sequence it is called, if there is a conflict. If a later edit changes the same attribute of a cell that an earlier edit alters, the later change is applied. Also, note that the transaction system only applies to the model at the current time, you cannot make `GraphLayoutCache` calls as part of a transaction, nor does the `GraphLayoutCache` have begin/end update methods.

2.2.7 SUMMARY

In this section we looked at inserting cells into the graph model and manipulating them. Each cell has an attribute map used to describe its appearance and behaviours. Using the `insert()`, `edit()` and `remove()` methods on the `GraphLayoutCache` we can change cells in a way that the graph model is updated, the screen is repainted properly, an undo of the change is added to the undo history and all listeners to the model are informed of it changing. These methods are commonly referred to as the 3 editing methods and it is worth remembering that they form one of the key parts of the JGraph API. There currently exists no method in the `GraphModel` interface that perform a compound of 3 editing methods to enable insertion, attribute editing and removal in one atomic, undoable operation.

2.3 Creating and Configuring the JGraph class

The `JGraph` class itself ties together the main components of the graph , provides top-level configuration of the graph and a number of general utility methods. The model-view-controller pattern for JGraph is shown below:

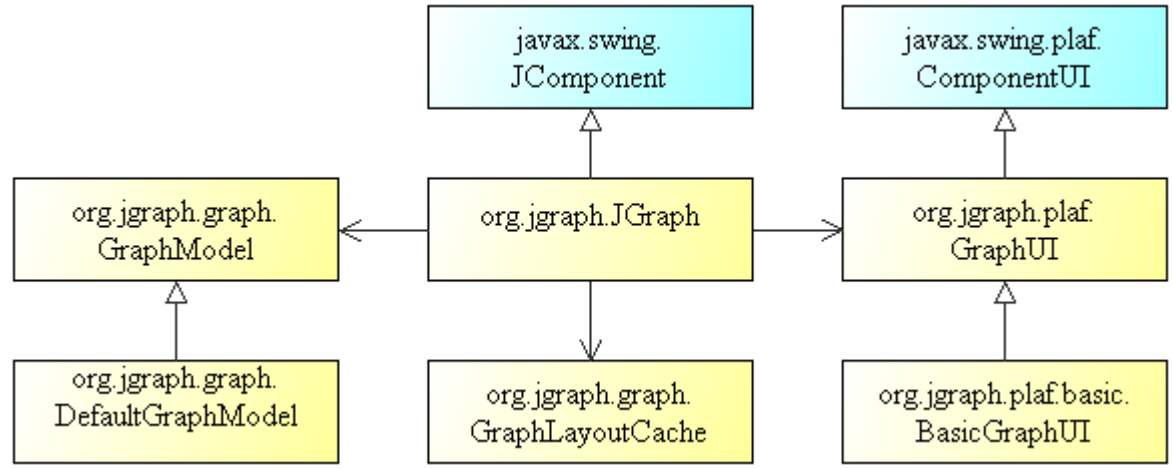

Illustration 13 : JGraph MVC

`JGraph` is a `JComponent` and holds references to its model, view and UI. The basic structure of the component, namely the Swing MVC architecture, is inherited from `JTree`. However, `JGraph` has an additional reference to a graph layout cache, which is not typically used in Swing MVC. The graph layout cache is analogous to the root view in Swing's text components, but it is not referenced by the UI-delegate. Instead, it is referenced by the `JGraph` object such that it preserves the state when the look-and-feel is changed.

When creating your `JGraph` instance and associated objects, it is important to get the order of object creation correct and to ensure that the objects correctly reference each other where appropriate. The `JGraph` holds references to the current `GraphModel` and `GraphLayoutCache` and the `GraphLayoutCache` needs to have a reference to the `GraphModel`. The simplest method of instantiating a `JGraph` is:

```
JGraph graph = new JGraph();
```

This will create a `DefaultGraphModel` and `GraphLayoutCache` for you and set up the reference in the `GraphLayoutCache` to point at the new model. Say, for example, you have your own graph model, use:

```
GraphModel model = new MyGraphModel();
JGraph graph = new JGraph(model);
```

The `GraphLayoutCache` will be set up correctly for you in the same way as before. Next, your own `GraphLayoutCache`:

```
GraphModel model = new DefaultGraphModel();
GraphLayoutCache view = new MyGraphLayoutCache(model,
```

```
                                                    new
DefaultCellViewFactory());
          JGraph graph = new JGraph(model, view);
```

You could pass null as the first parameter to the `MyGraphLayoutCache` (note that we're assuming your custom cache object constructors have the same signatures as `GraphLayoutCache`) and a `DefaultGraphModel` would be created and all the references set up for you. However, explicitly creating the model and passing it in makes the code much clearer. Of course, the last permutation is a custom model and layout cache. Simply use your own model in place of the `DefaultGraphModel` in the last example above to achieve this.

Another area where references need to be kept correct is when either the model or the layout cache are changed after the `JGraph` has been constructed. To do this use the `setModel()` and `setGraphLayoutCache()` methods on the `JGraph` class passing the new model and layout cache instances respectively. Upon setting the model any layout cache currently associated with the `JGraph` instance will be updated to use the new model instead. When setting a new layout cache, the model associated with that layout cache will be passed to `jgraph.setModel()` automatically. If you wanted to keep the current model associated with the `JGraph` instance you should create the new layout cache and pass the current model to its constructor before passing the layout cache to `jgraph.setGraphLayoutCache()`.

2.3.1 CONFIGURING JGRAPH

Many of the main features in `JGraph` can be enabled or disabled through the `JGraph` class. Below is a list of configuring methods worth learning, note that some are inherited from superclasses. Not all the configuration methods in `JGraph` are listed below. Some others will be introduced in later sections. Keep in mind these are accessor methods, for each `set` method there is a corresponding `is`, or `get` method. If you would like to try out the effects of any of the `set` methods mentioned, try applying them to the `JGraph` instance in `HelloWorld`, just after you create it.

- `setEnabled(boolean)` is the highest level configuration in `JGraph` (the method is actually in `JComponent`). This determines whether or not mouse events are handled. When set to false this disables selection, moving cells, editing labels, resizing, anything that requires mouse interaction. The underlying variable is `true` by default.

- `SetEditable(boolean)` determines whether or not vertices and edges may be edited. Editing should not be confusing with enabling, editing refers solely to the process of clicking on a graph a set number of times (see `setEditClickCount()`) to bring up an editor in-place (over or around the vertex) that allows the string content of the cells label to be altered. The underlying variable is `true` by default.

- `setEditClickCount(int)` determines the number of time you have to click on a editable cell (by default those allowed string labels) before the editor for that label is invoked. The underlying variable defaults to 2, i.e. double-click to edit.

- `SetMovable(boolean)` determines whether or not vertices and edges may be moved. Note that ports cannot be moved in the default implementation at any time. The underlying variable is `true` by default.

- `SetConnectable(boolean)` determines whether or not new connections are allowed to be established. Note that this only applies the connecting operations performed in the GUI, attempts to programmatically connect an edge will still work even if this method is set disabled. If you try this in the `HelloWorld` example, the graph appears with the edge connected. You can still disconnect the edge by selecting the edge, then clicking and dragging one end of the edge away from the attached vertex. However, if you try to drag the edge back onto the vertex there is no way to reconnect it if you have called `setConnectable(false)`. The underlying variable is `true` by default.

- `SetDisconnectable(boolean)` determines whether or not connected edges may be disconnected from their attached vertices. Specifically, can you grab the end of the edge attached to the vertex and move it from its attachment point. `JGraph` based applications like workflow editors often do not allow disconnected edges and so use this method to enforce that behaviour. The underlying variable is `true` by default, i.e. edges may be disconnected.

- `SetDisconnectOnMove(boolean)` determines whether or not connected edges should be disconnected when moved. This is different to `setDisconnectable` in that it relates to moving the edge as a whole, rather than one end of the edge. The underlying variable is `false` by default.

- `SetGridEnabled(boolean)` determines whether or not cells are 'snapped' into particular positions in the graph to form a more regular structure. The concept of a grid is that a number of points are laid out throughout the graph co-ordinate space as a grid and cells are positioned on their closest grid point, a process naming snapping. The grid can be configured by the distance between each point. The underlying variable is `false` by default, i.e. cells are inserted or moved to double precision co-ordinates and not moved onto the grid positions.

- `setGridVisible(boolean)` determines whether or not the grid is visible. If `setGridEnabled` is set to true you get 'snapping' to grid points, otherwise no 'snapping' will occur.

- `SetMoveBelowZero(boolean)` determines whether or not cells are allowed to have the position of their top-left corner anywhere in negative co-ordinate space. It is generally recommended not to allow this unless there is a good reason. The underlying variable is `false` by default, i.e. all top-left corners of cells are always in positive co-ordinate space.

- `SetAntiAliased(boolean)` determines whether or not to enable anti-aliasing for the JGraph component. Anti-aliasing is a technique for blurring sharp, jagged lines using color gradients. The underlying variable is `false` by default.

- `SetSelectionEnabled(boolean)` determines whether or not any cells may be selected. The underlying variable is `true` by default.

2.4 The Graph Model

2.4.1.1 Introduction

The graph model stores the logical structure of the graph and this fits in with the MVC idea of the data of an object being stored within the model. `GraphModel` defines the interface for objects that may serve as a data source for the graph. This interface dictates, to an extent, how the underlying data that describes the graph model must be stored within classes implementing this interface. The default implementation of `GraphModel`, `DefaultGraphModel`, not only is useful as an instructive tool for explaining graph models, but also is suitable for the majority of simple applications that use `JGraph`. If you want custom graph model behaviour, your first approach should be to extend `DefaultGraphModel`, even very simple models are reasonably complex to implement from scratch.

Design Background - Some people are slightly confused by the presence of visual information being indirectly stored by the model, specifically, the graph cell's attributes. These attributes comprise information such as positioning, renderering details and control behaviours such as whether the cell can be dragged. Many graph libraries architect their graph model in such a way that it only graph cells (vertices and edges) and the relationships between the cells are stored. JGraph is targeted at graph visualization over general graph analysis and the decision was taken during its design to make the model represent that of a visualized graph, not just a graph structure. This makes the API better suited to graph visualization at the slight expense of performance when performing graph analysis.

2.4.1.2 The 3 editing methods

The `insert()`, `edit()` and `remove()` methods on `GraphModel` perform the corresponding function that their `GraphLayoutCache` methods perform, though their parameters look rather more complex. The use of these methods, and their corresponding signatures methods in `GraphLayoutCache` will be covered in the section on Advanced Editing. As previously mentioned, inserting, editing and removing directly into the model means that all views based on that model will receive the same changes. Using only this approach means that multiple independent views are not possible, a decision that needs to be considered at the specification stage of an application.

2.4.1.3 Accessing the Graph Model Data

The next methods to be considered in `GraphModel` are `getRootCount()`, `getRootAt()`, `getIndexOfRoot()` and `contains()`. Why the data structure of JGraph models is how it is and why these methods of access to the data structure are used requires knowledge of how the model of JTree has been extended to JGraph, as well as the

terminology used to describe the relationship of nodes within a JTree. Background on this topic in available in Appendix A.

By default each vertex or edge inserted into a JGraph forms the root node of a tree in the graph data model. Ports, since they logically belong to vertices and edges, are children of the cells they are attached to. Therefore, adding the two vertices and single edge as in the Helloworld example would result in the roots structure looking like this:

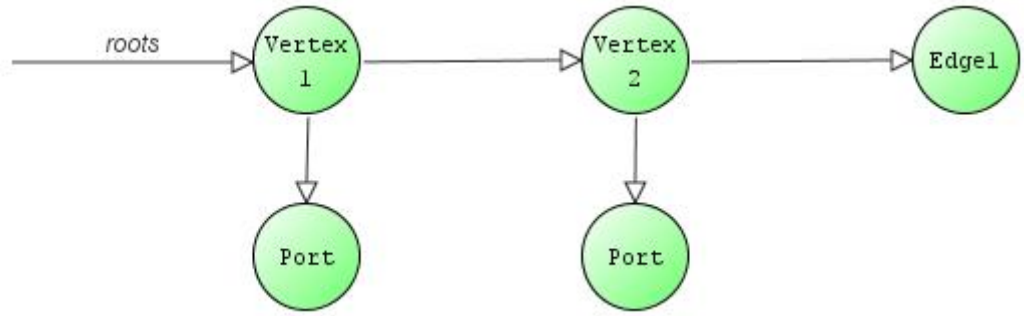

Illustration 14 : Representation of the roots structure after the Helloworld application has run. The vertices and edges inserted into roots and the ports are children of the cells that they are logically part of.

The convention is to call the structure that stores the top-level vertices and edges of the graph `roots`. This name will be used throughout this manual to refer to that structure, being the graph data model structure. The `roots` structure is technically a forest of connected trees. The trees can become more complex when dealing with grouped cells, but this will be covered in a later chapter.

It should be clear now, from the above diagram, what function the four methods mentioned perform:

- `getRootCount()` returns the number of elements in the `roots` structure, this would return 3 in the above example.
- `getRootAt(int)` takes a integer parameter and returns the element referring to that index in the `roots` structure. Note that this implies that `roots` is an ordered collection and this is vital for a number of pieces of functionality that `JGraph` provides. Of the methods that require navigation of the roots structure, `getRootAt()` is by far the most used and so also the most performance sensitive method usually. It is for this reason that roots in `DefaultGraphModel` is an `ArrayList`, by default, enabling this method to complete in constant time. Calling getRootAt(1) would return the vertex represented by Vertex2 in the diagram above. The convention is for the first entry in roots to have an index of zero.
- `getIndexOfRoot(Object)` return the index of the cell in the roots structure. Passing in the object corresponding to Edge1 in the above diagram would return 2. If `roots` does not contain the object the method returns -1.
- `contains(Object)` return a boolean indicating whether or not the specified object can be found within `roots`.

Information - Changing `roots` to be something other than an `ArrayList` could be done with a custom graph model, but there are a number of important reasons for this choice. As mentioned, `getRootAt()` is usually the bottleneck method of the four and choosing another `List` type or even a `Map` or `Set` would result in the method performance degrading from constant time, O(1), to being proportional the number of entries in roots, O(|V|+|E|). Also, `getIndexOfRoot()` naturally lends itself to using the `indexOf()` method of `List`, the semantics of the return values match up. If a `Set` or a `Map` were used, keep in mind that roots must be ordered, so a `LinkedHashMap` and `LinkedHashSet` would be appropriate. They were only introduced in JDK 1.4, anyone using earlier version of Java has little option but to use an `ArrayList`.

These four methods form the basic means to navigate and interrogate the roots structure. There are additional methods that deal with the parent/child relationship that will be covered in the section on Groups. It should be remembered that the `GraphModel` interface should always be used to access the graph data model structure. The interface provides the means to obtain the necessary information about `roots` and the type checking is purposefully weak, cells are always passed as `Objects`, to allow complete flexibility in the way cells are designed. Also, accessing roots through the `GraphModel` interface provides independence from the actual model implementation. If the model needs to be exchanged for one that provides improved performance or database synchronization, for example, this can be done without changes to the calling code.

2.4.1.4 Cloning the Graph Model

2.4.1.5 Navigating Connections Using the GraphModel interface

`Object getSource(Object edge)` and `Object getTarget(Object edge)` methods in `GraphModel` provide the means to obtain the cells, if any, that any particular edge connects to. Note that edges implicitly have a direction in JGraph. This does not preclude the visualization of undirected graphs, however. Avoiding the use of arrowheads on edges is all that is required to visually make any graph look undirected.

To obtain an `Iterator` of edges connected to a particular cell, `edges(Object port)` is available. Although, the parameter is named 'port' the `GraphModel` interface does not enforce that only `Ports` may be connected to edges. However, `DefaultGraphModel`, for example, does enforce this rule. The arrangement of vertices have children ports that form connection with edges is the best design for the majority of graph models. There are occasions when this isn't so efficient, for example, graphs with very large numbers of vertices each that only have one port can be speed up and have a reduced memory by combining the vertex and port into one object. This model arrangement is explained in the later chapter on performance issues. Also, if the same cells are to be used in multiple models, with different connection relationships in each model, the `DefaultGraphModel` is not suitable. This is because connection relationships are stored in the cells, making it impossible to define connections separately in different models. For

this reason it is advised not to share cells between graph models, this is a trait shared with `JTreeModels`.

`boolean isEdge(Object edge)` and `boolean isPort(Object port)` are implementation dependent methods that must adhere to the idea that edges can only connect to ports and that ports are allowed to have edges connected to them.

The final methods that allow navigation between elements in the graph model data structure are those that navigate parent/child relationships. These methods will also be discussed in the context of grouping in a later chapter.

- `Object getParent(Object child)` returns the parent, if any, of the specified cell in the graph model data structure. As in trees, all children may only have one parent.
- `int getIndexOfChild(Object parent, Object child)` returns the index of the specified child in the collection the parent holds of its children. Note that this collection must be ordered to be deterministic.
- `Object getChild(Object parent, int index)` returns the child at the specified index in the collection that each parent holds of its children. Again, for this method to be deterministic the collection must be ordered.
- `int getChildCount(Object parent)` returns the number of elements in the specified cells collection of children.

For those familiar with `JTreeModel` you will recognize that navigating up and down any tree starting at an element of `roots` is almost the same as the mechanism used in `JTrees`. From these methods presented we are able to navigate between all elements of a graph model data structure, parents and children and between connections. We will now walk through example code showing how to navigate between the various elements using only the `GraphModel` interface. For this we have to assume some implementation of the graph model since the relationship between vertices and ports is not defined explicitly or implicitly in the `GraphModel` interface. For this we will use the `DefaultGraphModel`, where ports are separate objects to vertices and ports are always direct children of the vertices they are part of. Note that utility methods to carry out the functions described below are already available in JGraph, the examples to follow are for those wishing to understand the architecture of JGraph more thoroughly. A representation of the relationship between two vertices connected by one edge is shown in the diagram below.

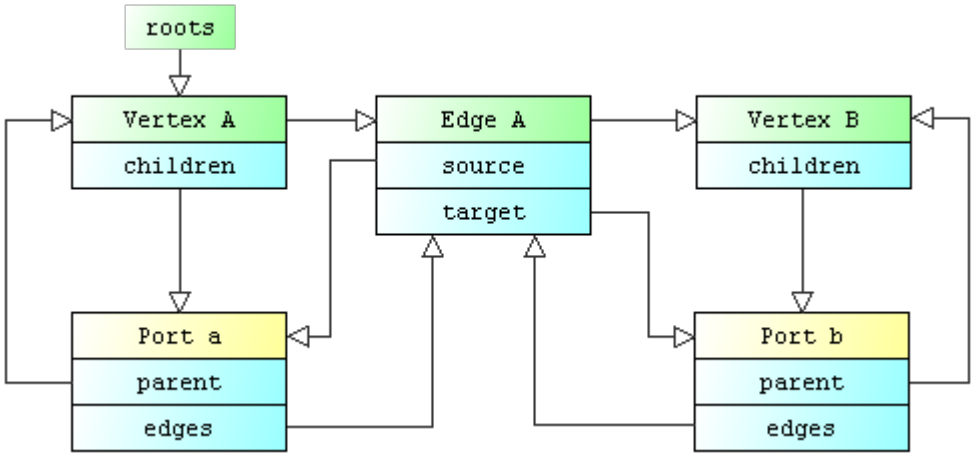

Illustration 15 : Representation of the associations between graph model data elements with 2 vertices connected by 1 edges inserted into a DefaultGraphModel

2.4.1.5.1 Obtaining a collection of edges connected to a vertex

To obtain a collection of edges using only the GraphModel interface given a vertex you must cycle through each port belonging to that vertex and then within each port iterate through each edge connected to that port:

```
List listEdges = new ArrayList();
int numChildren = model.getChildCount(cell);
for (int i = 0; i < numChildren; i++) {
     Object port = model.getChild(cell, i);
     if (model.isPort(port)) {
          Iterator iter = model.edges(port);
          while (iter.hasNext()) {
               listEdges.add(iter.next());
          }
     }
}
```

Note the requirement to check if a child of a vertex is a port, vertices can also be children of vertices, the basis of JGraph grouping functionality.

2.4.1.5.2 Obtaining the Source and Target Vertices of an Edges

To obtain the source and target vertices that an edge connects to through the ports on the vertex, only using the GraphModel, you obtain the ports at either end of the edges using getSource() and getTarget() and then obtain the parents of those ports:

```
Object sourceVertex = model.getParent(model.getSource(edge));
Object targetVertex = model.getParent(model.getTarget(edge));
```

2.5 Design Contracts in JGraph

A reasonably frequent question is why are so many parameters and return values Object types rather than Vertices or Edges or Ports. This relates to design decision to make the contract that JGraph requires adherance to being as simple and small as possible. The GraphModel interface is the other interface of the major elements of the library that must be adhered to. Extending from that:

1. Any object can be used as a cell in a GraphModel. It is not required that cells implement an interface.

2. The Edge and Port interfaces are only used in the DefaultGraphModel. They are a contract between the default model and its cells. (They are not used anywhere else in JGraph.)

3. The Graph structure should only be accessed through the GraphModel interface, not through the Edge or Port interfaces. It is even not required that a GraphModel uses ports (it is however required that every edge is represented by an object in the model).

4. Neither the JGraph component nor one of the algorithms for graph traversal uses the Edge or Port interface, they all use the GraphModel interface which in turn uses the Edge and Port interface to retrieve the Graph structure from the cells. This way, the storage structure can be hidden from the GraphModel client.

3 Cells

3.1 Types of Cells

As previously mentioned, there are three types of graph cells in `JGraph`, vertices, edges and ports. Vertices form the main objects that the user can see about the graph, the squares, the circles, the icons and even more complex objects such as other `JComponents`. Edges are usually lines that represent graph structure connections between vertices. You can have multiple edges between the same pair of vertices, termed **parallel edges**, or even edges that start (source) and finish (target) at the same vertex, termed **self-loops**.

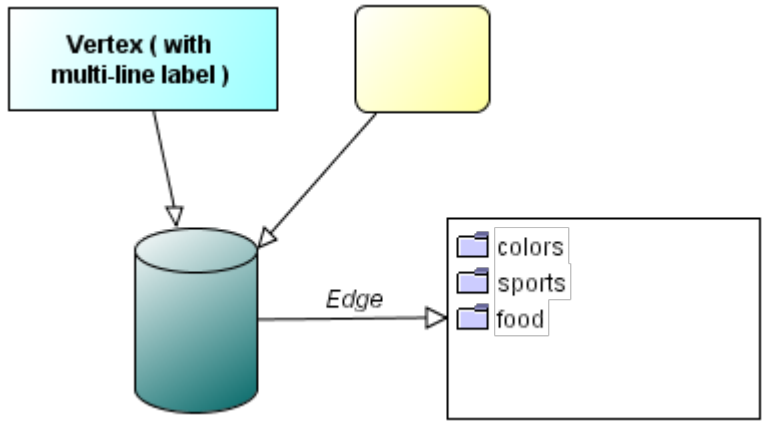

Illustration 16 : A variety of vertices, some connecting edges and available ports visible as small squares

The above diagram shows some vertices and their connecting edges. Also visible are small squares, these are ports attached to the vertices and edges. Ports visually represent points at which the ends of edges may be connected to vertices or other edges. The reason for having a logically-separate entity for ports is that multiple ports can be fixed (offset) to specified positions on vertices, so the graph model data structure needs to distinguish between connections to different points within its boundary.

3.2 Cell Interfaces and Default Implementations

3.2.1 GRAPHCELL INTERFACE

`GraphCell` is the interface to which graph cells should adhere. Note the use of the word *should*. If desired another interface could be used and the correct use of the `GraphModel` interface would mean this change is transparent to the user of the model. However, many of the application and extensions to `JGraph` as well as default interface implementations, `DefaultGraphModel` for example, assume the use of the `GraphCell` interface. Unless you have a very good reason otherwise, have your cells inherit from `GraphCell` hierarchy.

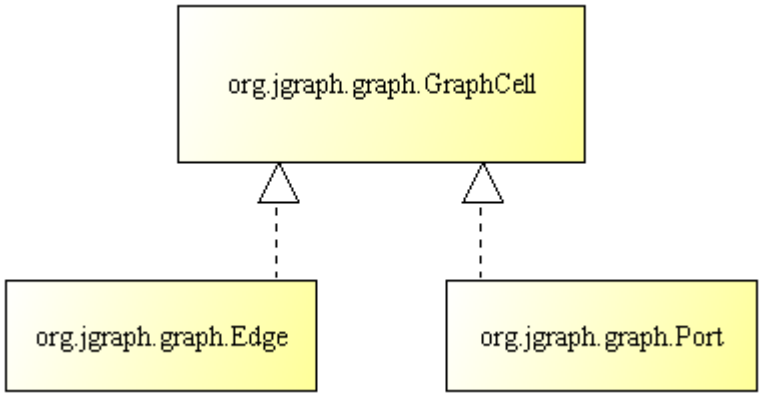

Illustration 17 : The GraphCell interface hierarchy

The vertex is considered the default case and so uses the `GraphCell` interface itself. `Edges` and `Ports` are considered to be specializations of vertices and so have their own interfaces. `GraphCell` itself only offers two methods, `getAttributes()` and `setAttributes()`. Attributes were mentioned in Chapter 2 and, as the `GraphCell` interface suggests, are key to defining how cell appear visually. It is unlikely that you should ever need to call `setAttributes()` on a graph cell, the 3 editing methods are the default route for changing attributes and setting an attribute map directly would retain no undo history and not refresh the cell and display accordingly.

`getAttributes()` is more commonly used, you saw it being used instead of creating a nested map for cell insertion in the `HelloWorld` example in Chapter 2. Again, this method of accessing *and altering in-place* the cells storage map directly should only generally be used for cell insertion. However, there is another exception to this rule, when you wish to change the attributes of a cell or cells without adding the change to the undo history and you require high performance for the operation. A common example of this is a mouse rollover. Calling `edit()` is excessive, for example, to highlight a cell when the mouse is over it. In this case you should obtain the attributes of the cell, make the changes to the storage map in-place, refresh the cell and repaint the appropriate area. This operation is usually best performed on the view of the cell, see the section later in this chapter on cell views. Note, since JGraph 5.11, the double buffering interface needs to be explicitly told about in-place changes. The bounds of the change should be sent to the `JGraph.addOffscreenDirty()` method like so:

```
graph.getGraphLayoutCache().refresh(viewsArray, false);
Rectangle2D cellBounds = GraphLayoutCache.getBounds(viewsArray);
graph.addOffscreenDirty(cellBounds);
graph.repaint();
```

3.2.2 THE EDGE AND PORT INTERFACES

The `Edge` interface defines the methods required to set and determine the connections for a particular edge. These are `getSource()`, `getTarget()`, `setSource()` and `setTarget()`. Their functions will be reasonably obvious from the names, remember all

edges have a direction in the model traveling from the source end to the target end. Again, the types involved in these methods are all `Objects` to provide complete flexibility. In combination with the `GraphModel` interface it is possible to obtain the cell(s) connecting to edges, determine their type and navigate consistently, without referring to implementation specifics.

The `Port` interface defines the necessary methods to add, remove and obtain the edges connected to it. Remember that ports are conceptually a entity associated with a vertex to which any number of edges may connect. `Edges` may connect either their source or target end to a port, making that port the source or target port, respectively. An edge may also connect both its source and target ends to the same port, making the edge a self-loop. Note that self-loops are created when an edge has the same vertex as its source and target, not just if the source and targets port is the same. When testing for self-loops you should ensure that you obtain the source and target vertices, generally the parents of the source and target port, and see if they are the same vertex.

The `Edge` interface also defines the static `Routing` interface. This interface defines the route method which deal with drawing the edge given a number of points through which the edge passes. This will be expanded upon later in this Chapter on the section on using edges.

`Port` defines the methods `edges()`, `addEdge()` and `removeEdge()`. The add and remove methods take an `Object` as per-standard in the JGraph design. `Edges()` returns an `Iterator` to the `Collection` of `Edges` connected to this `Port`, as set up by the `add` and `remove` methods.

Note that `Port` does not store any information about whether or not it is the source or target port of edges that connect to it. This information is only in `Edges` to avoid redundancy and the danger of the information getting out of synchronization.

The two lesser known methods in `Port` are `getAnchor()` and `setAnchor()`. The idea of anchoring also requires an explanation of how ports are positioned, be they relative or absolute and where the origin of their offset is or if they have no offset at all. This is described in the later section in this chapter on Using Ports.

3.2.3 THE DEFAULTGRAPHCELL

`DefaultGraphCell` is the standard implementation of a graph cell provided in `JGraph` and as with most of the default implementations is suitable either as-is, or as the superclass of your cells for the majority of applications. Like the corresponding interface, vertices use the `DefaultGraphCell` class and edges and ports use default classes subclassed from `DefaultGraphCell`:

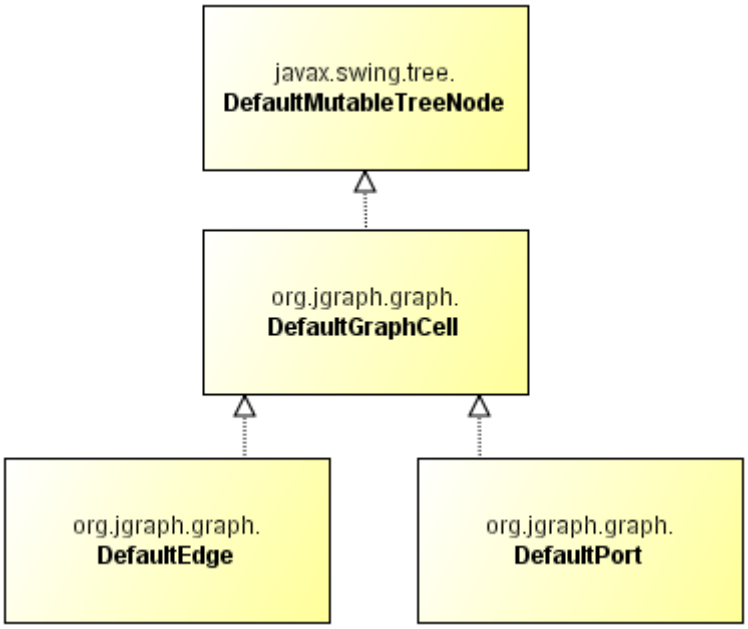

Illustration 18 : The class hierarchy for the default graph cells

The design extension of `JGraph` from `JTree` is again apparent here from `DefaultMutableTreeNode` being the super class of the default graph cells implementations. Two important principles are inherited from the tree nodes, that of the parent/child relationship that cells may have with one another and the user object. The `TreeNodes` interface provides basic methods to determine a cell's parents and children. Where possible, you should use the graph model methods for traversing the parent/child relationships in preference to that supplied by `DefaultGraphCell`, since `GraphModel` is the interface and the design contract.

3.2.3.1 The Default Graph Cells Constructors and Methods

This leads us onto the four constructors of `DefaultGraphCell`, each taking an additional parameter. If you look at the source code to the class you will see each constructor passes a null value for missing parameters until they all end up calling this constructor:

```
    public DefaultGraphCell(Object userObject, AttributeMap storageMap,
MutableTreeNode[] children)
```

The first two parameters you should be reasonably familiar with. The `userObject` parameter becomes the user object of the cell, obviously. The `storageMap` parameter, if non-null, is intended to be the `AttributeMap` used by that cell, generally for its lifetime. This parameter is most used when your application requires a custom attribute map for storage. Note that cell cloning does not make use of this mechanism for transferring attribute maps. The last parameter, `children`, is an array of the cells you wish to make children of the current cell in the tree node relationship. In the `HelloWorld` example we could have

created the `DefaultPort` earlier and inserted it using this parameter instead of adding it explicitly later using the `add()` method of `DefaultMutableTreeNode`.

The other methods in `DefaultGraphCell` are just simple implementations of those in `GraphModel`, other than the additional `clone()` method. `setAttributes()` ensures that an attribute map is created if null is passed in and only the 3-parameter version of the constructor calls `setAttributes()` within `JGraph`. If you have your own attribute map sub-class, if possible, you should create sub-classes of each of the default cell types with a new instance of your custom attribute map within `setAttributes()`, in case you forget to create an instance every time in the `DefaultGraphModel` constructor.

The `clone()` method returns a deep copy by using the super-class clone methods and the `clone` method of the attribute map.

The `DefaultPort` implementation is trivial, comprised mainly of getters and setters and the additional `clone` method. Of note, the collection of edges is implemented as a `HashSet`, this means that the order in which the `Iterator` returned from `edges()` presents the edges is not assured. When Java 1.3 becomes end of life in December 2006 it is intended to change this to a `LinkedHashSet` to retain ordering. You may wish to make the change yourself until then if you are using Java 1.4 or higher and require this feature.

The `DefaultEdge` implementation, again, is generally obvious. In the `clone()` method, however, it might not be so clear why the source and target objects are not copied. This is because an edge may be cloned into a different model where the original ports do not exist. `DefaultEdge` also contains the default routing algorithm, `DefaultRouting`, which alters the list of points passed into the `route()` method to route the edge in a more aesthetically pleasing manner.

3.2.4 CLONING CELLS

The `clone()` method of `DefaultGraphCell` calls the superclass clone and adds a cloned version of the attribute map of that cell to the new cell. It should be noted that this cloning mechanism does not add clones of children of the original cell, or even references to the original children to the clone. Nor does this mechanism clone the user object (see section shortly on User Objects), it only adds a reference to the original user object. To obtain a "deeply" cloned version of a cell, one with cloned children and cloned user object, there is a static utility method on `DefaultGraphModel` to perform this action:

```
Object clone = DefaultGraphModel.cloneCell(graph.getModel(), vertex);
```

3.3 User Objects

The `userObject` of `DefaultMutableTreeNodes`, and so also of `DefaultGraphCell`, is an `Object` that can play an important part in the way you construct more complex JGraph-based applications. User objects store any data that is associated with the graph cell that does not belong as part of the graph cell or it attributes. An example of this is a workflow editor designed to export to a particular workflow format. The editor would have cells representing a start, a branch, a join an activity, and so on. The

`userObject` would be used to store information specific to that type of cell, so this information could be fed into the export stage. For an activity cell this might include a `String` of the name of the person assigned the activity and a `URL` containing information about it. The application would provide some means to modify the `userObject` and so the `userObject` needs to be accessed by a specialization of a graph cell, usually of the `DefaultGraphCell`, so that it is aware of the real object type of the `userObject`.

The only method that must be implemented in a `userObject` to be usable in `JGraph` is the `toString()` method. By default the `String` returned is what is displayed as the label for that cell. In simple applications with no data storage requirements for the `userObject`, use a `String` itself as the `userObject`, as shown in the `HelloWorld` example:

```
cells[0] = new DefaultGraphCell(new String("Hello"));
```

The parameter to the `DefaultGraphCell` is actually the cells `userObject`. Since the `toString` method as a `String` returns `this`, `Strings` fits the minimum requirements for user objects.

Note that the value to be displayed in the cell's label has an indirection through `JGraph.convertValueToString(Object)`. This method allows the cell label to display alternative text for the same cells in different instances of `JGraphs`.

3.3.1 OBTAINING AND CHANGING THE USER OBJECT

The user object of a cell is only stored as an object associated with a cell, it is not stored in the a cell's storage attribute map. However, to provide consistency with changes to user objects through editing calls you can obtain the user object using:

```
GraphModel.getValue(Object)
```

and set the user object using

```
GraphConstants.setValue(Object)
```

The attribute map will ensure the user object does not end up in the eventual storage map, but setting the object in this way and calling edit() will ensure that the change to the user object is correctly added to the undo history.

3.4 Cell Views

The MVC pattern applies to graph cells within `JGraph`, as well as the overall design itself. All graph cells have at least one associated cell view that deals with various visual functionality and the process of updating the visualization of that cell. Cell views associate a renderer, and editor and a cell handle.

Design Background - Readers familiar with `JTable` and `JTrees` might be wondering why there is a cell view at all, you might expect just a cell object and a renderer for that object. A graph component has considerable more visual complexity and geometric pattern

flexibility than any of the current standard Swing components. At the design level the `GraphModel` and `GraphCells` basically implement a graph structure, without any implicit assumption of any visualization capability. Moving all of the functionality of the cell view into the graph cells would unnecessarily bloat the pure graph model aspects of `JGraph` for those only wishing to perform graph analysis. In the JTree design, the AbstractLayoutCache holds boolean values indicating whether or not each node is expanded or not. Graph cells have considerably more visual state than a boolean, there has to be a class to abstract this state rather than the GraphLayoutCache hold collections of each cell state for, say, each of 10 elements of functionality, this class is the cell view. Also in a JTree there is not the option of associating different cell renderers with cells. The JTree instance directly references the cell renderer, because of JGraph's renderering flexibility this would require the GraphCell to reference the renderer without a cell view. This breaks the design rule of the GraphCell only dealing with the graph structure. Finally, without cell views, it is not possible to have view-local attributes, that is, cells in different views being displayed differently.

Renderers are part of the Swing design, they abstract the drawing functionality of a component into a single static class instance, a pattern also known as the *flyweight* design. The idea is for all component views that may draw the same thing, just with different visual attributes, to share this common instance. This avoids excessive memory requirements for large numbers of the same component. When a graph cell is rendered the attributes of the cell view are fetched and inserted into the renderer instance, a process known as configuring the renderer. The cell is then painted by the renderer and this process continues for each cell. This method can save a great deal of memory against the worst-case one instance per cell mechanism. However, the process of installing attributes causes a small performance hit, but this is usually negligible compared to the computational requirements of painting components. Renderers are described in more detail later in this chapter.

The editor associated with a cell view is the same principle as cell editors for `JTables` or `JTree` elements. If you double-click on a vertex or edge in the `HelloWorld` example it brings up what is called an **in-place** editor, that is a component where you can edit text associated with a cell at the location where the cell being edited is positioned. The default editor provided is a simple, single-line editor called `DefaultGraphCellEditor`, that extends `CellEditor`. It is possible to implement multi-line, rich text, or even a word processor style editor if required.

3.4.1 CELL HANDLES

Cell handles do not have a parallel concept in Swing, in other Swing components cell editing is only performed by means of in-place editing. JGraph introduces the concepts of changing the boundary size of cells, as well as moving cells to arbitrary locations. Cell handles perform the task of displaying a visual representation indicating that the cell affords resizing and moving, as well as the task of processing interactive manipulation on a cell or group of cells. The name *handle* implies that they possess the properties that allow you to handle the cell, using the mouse or other input device. Handles appear around cells that are currently selected, indicating the cell may have moving and resizing operations applied.

Handles are a common paradigm in many graphical applications, for example in a word processor if you select an image handles will appear on the perimeter of the image to indicate that it affords moving and resizing.

Handles are based on the *Composite* pattern in JGraph. A root object provides access to children based on a common interface, the `CellHandle` interface. The UI-delegate creates a handle, usually called `RootHandle`, and the root handle, in turn, uses the `CellView's getHandle` method to create its child handles.

The `CellHandle` interface defines the basic functionality a handle must provide, note that the `CellHandle` interface is very similar to that of `MouseMotionListener` and `MouseListener`. For visualization there is `paint(Graphics g)` and `overlay(Graphics g)`. The `paint` method draws handle for each selected cell for when the cells are static and the `overlay` method deals with drawing during live-preview, usually implemented as fast XOR'ed-painting for speed whilst cells are being dynamically moved. We will come back to handles in the chapter on Events.

The default implementations of handles in `JGraph` are the `SizeHandle`, the `EdgeHandle` and the `RootHandle`. The root handle is responsible for moving cells, the size handle is used to resize cells and the edge handle allows the connection and disconnection of edges, as well as the interactive addition, modification and removal of individual points to/from edges.

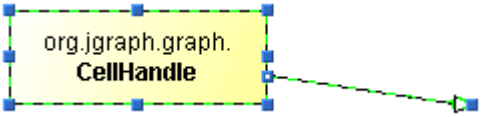

Illustration 19 : The static handle around a selected cell and edge drawn by the paint method of the cell handle

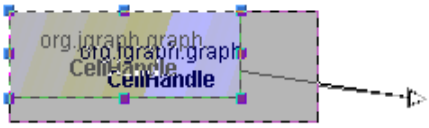

Illustration 20 : A dynamic handle drawn by overlay when a cell is resized

3.4.2 THE CELL VIEW HIERARCHY

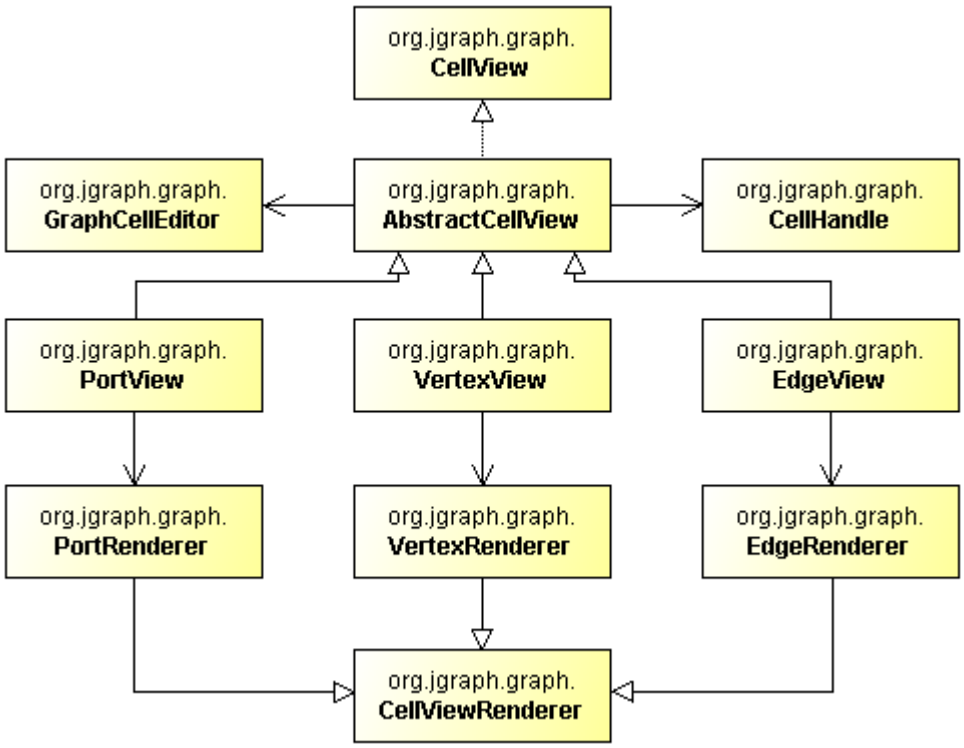

Illustration 21 : The CellView interface, default implementations and static relations

The `CellView` interface defines a number of methods associated with on-screen updating, accessing and modifying visual attributes and accessing associated visual components. The `getRendererComponent()`, `getHandle()`, `getEditor ()` methods return the renderer, the handle and the editor associated with the cell view, as described earlier in this section.

The `refresh()` method is called whenever the model cell that the cell view is associated with changes. This performs the necessary updating to the cell view attributes, but does not cause the cell views to repaint. Note that the 3 editing methods automatically call refresh and repaint for all views affected by the change. It was mentioned earlier that if you wish to affect a high-performance change to a cell, without the need for an undo history of the change, you should change the cells attribute, call `refresh` and then `repaint`. The `refresh` is only called when the corresponding cell has changed, not when a dependent cell of the graph cell changes

The `update()` method is the method that `refresh` uses to synchronize its own attributes with that of the associated graph cell. This method is called when the associated model cell changes, but also when a dependent cell changes, or when just a view update is required, which occurs during live previews.

The `update` method is good place to implement automatic attribute modification, such as edge routing or other functionality that is based on other attributes of the cell, or the graph geometry.

The cell view hierarchy stores parent/child relationships separately to the graph model

structure. Although this may seem confusing, it provides for a great deal of flexibility and much of the use of this functionality is hidden from the developer. Without this information you would need to go from the cell view to its model cell, obtain its parent and navigate back to the according parent cell view. A dependency from model to view is highly undesirable. An example of the use of this structure is the way edges connected to cell within a collapsed group visually attach to the perimeter of the first visible parent of the cell. This is performed entirely in the view, without the need to reference the model. The methods for accessing the cell view relationships are `getParentView()`, `getChildViews()`, `removeFromParent()` and `isLeaf()`, all perform the function obvious from their naming.

3.4.2.1 getPerimeterPoint

`getPerimeterPoint()` is the first method in `CellView` you are in any danger of actually having to implement for a simple application. `getPerimeterPoint` returns the point on the perimeter of the view where the edge specified in the parameter list intersects. This is important to get right, since the basic type of port, the floating port, uses this method to determine where an edge should terminate on the boundary of a vertex. The use of ports and floating ports are described in more detail towards the end of this chapter.

AbstractCellView, the abstract superclass of all default cell views will return the center point of the cell if you do not provide an implementation of `getPerimeterPoint` further down the class hierarchy.

3.4.2.2 getRenderer

The other method you are likely to have to concern yourself with if you create a cell type is `getRenderer()`. `getRenderer` is not actually in the `CellView` interface, only `getRendererComponent` is. The implementation of `getRendererComponent` in `AbstractCellView`, the class that you will subclass from directly or indirectly for 99.9% of custom cell views, looks like this:

```
public Component getRendererComponent(JGraph graph, boolean selected,
                boolean focus, boolean preview) {
     CellViewRenderer cvr = getRenderer();
     if (cvr != null)
          return cvr.getRendererComponent(graph, this, selected, focus,
                          preview);
     return null;
}
```

As previously mentioned, each cell type consists of the cell, the cell view and the cell renderer. If a new cell type you create is visually distinct from the ones you already have, for example, you want to add a circle cell, you need to create a renderer class that paints a circle and ensure the view of that cell returns that renderer.

3.4.2.2.1 How to Create your Own Cell View and Renderer

Below is a template of what you might start with when creating your own view:

```
public class MyView extends AbstractCellView {

     protected static MyRenderer renderer = new MyRenderer();

     public MyView() {
          super();
     }


     public MyView(Object arg0) {
          super(arg0);
     }

     public CellViewRenderer getRenderer() {
          return renderer;
     }

     public Point2D getPerimeterPoint(EdgeView edge, Point2D source,
Point2D p) {
          if (getRenderer() instanceof MyRenderer)
               return ((MyRenderer)
getRenderer()).getPerimeterPoint(this,
                         source, p);
          return super.getPerimeterPoint(edge, source, p);
     }

     public static class MyRenderer extends JLabel implements
                              CellViewRenderer, Serializable {

          public void paint(Graphics g) {
          }

          public Component getRendererComponent(JGraph graph, CellView
               view, boolean sel, boolean focus, boolean preview) {
          }

          public Point2D getPerimeterPoint(VertexView view, Point2D
               source, Point2D p) {
          }
     }
}
```

Keep in mind it is advised to stick to the *flyweight* pattern and hold a single static renderer instance for each type of cell view to reduce the memory footprint.

3.4.3 CREATING CELL VIEWS AND ASSOCIATING THEM WITH CELLS

The process of creating a cell view for each graph cell created would be somewhat tedious to perform manually and so it is done behind the scenes using a cell view factory. The interface `CellViewFactory` defines one method, `createView()`. This takes an

instance of a graph model and the graph cell for which the view is to be created, creates the appropriate cell view and associates the cell and the view accordingly. The cell view factory is associated with the `GraphLayoutCache` and some constructors of `GraphLayoutCache` take the `CellViewFactory` as a parameter. You can change and access the cell view factory during the life of the cache using `setFactory` and `getFactory`.

The default implementation of `CellViewFactory` is `DefaultCellViewFactory`, if you do not specify a `CellViewFactory` when creating a `GraphLayoutCache`, you will get the default factory instantiated for you. `DefaultCellViewFactory`, with the deprecated methods removed, looks like this:

```
public CellView createView(GraphModel model, Object cell) {
     CellView view = null;
     if (model.isPort(cell))
          view = createPortView(cell);
     else if (model.isEdge(cell))
          view = createEdgeView(cell);
     else
          view = createVertexView(cell);
     return view;
}

protected VertexView createVertexView(Object cell) {
     return new VertexView(cell);
}

protected EdgeView createEdgeView(Object cell) {
     return new EdgeView(cell);
}

protected PortView createPortView(Object cell) {
     return new PortView(cell);
}
```

To associate your new cells and cell views extend the `DefaultCellViewFactory` class, add checks for your cell types and return a new instance of the associated cell view appropriately. For example, if you add `MyVertex` and `MyVertexView`:

```
protected VertexView createVertexView(Object cell) {
     if (cell instanceof MyVertex) {
          return new MyVertexView(cell);
     }
     return new VertexView(cell);
}
```

Or if you just want to make the default vertex use the circle view you have created, without creating your own cell type:

```
protected VertexView createVertexView(Object cell) {
     return new MyCircleView(cell);
```

```
}
```

Remember, like all factories, the cell view returned must be a new instance. Your application will not function correctly if they are not.

3.4.4 DEFAULT CELL VIEW AND RENDERER IMPLEMENTATIONS

3.4.4.1 The Cell Views

The default cell view implementations for the 3 basic cell types are `VertexView`, `EdgeView` and `PortView`. `VertexView` is probably the simplest implementation of the three, other than the `SizeHandle` (see Chapter 5 on Events). Its `update` method ensures that the vertex view has a bounds and the `getRenderer` and `getPerimeterPoint` just defer to the vertex renderer.

`PortView` has a size hard-coded into the final variable `SIZE` and returned in the `getBounds()` method. Ports tend to be visually rather simple and the default implementation has no handles, meaning no resizing. If you would like variable sized ports you might subclass `PortView` and implement `getBounds` to return the bounds attribute of the port's attribute map instead.

`PortView` also has some additional functionality relating to the port location. `getLocation()` and `shouldInvokePortMagic()` provide functionality that make it possible to have interactively movable ports as well as the local optimization of adjusting a ports position on a vertex in order to straighten an edge or edges connecting to it.

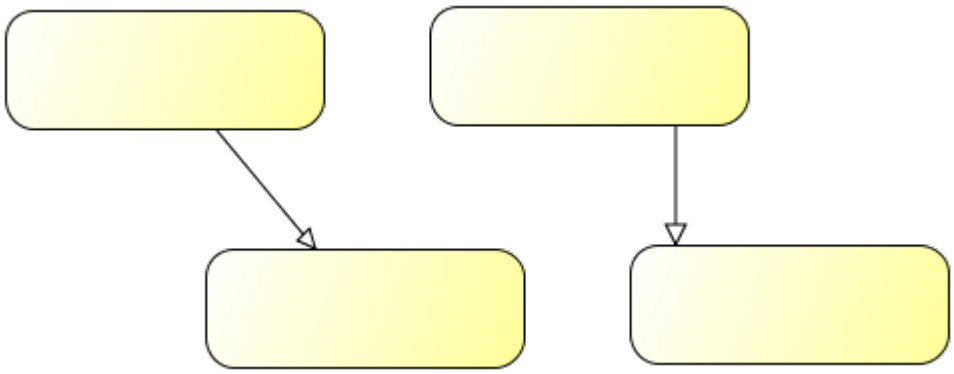

Illustration 22 : A standard floating port edge (left) and an edge connected to port using 'port magic' (right)

`EdgeView` is by far the most complex of the default views, since it needs to implement most of its functionality from scratch, as opposed to the vertex which gets a lot of inherited functionality in its renderer from `JLabel`. Without going in undue detail just yet, there are some general design principles in the edge view worth mentioning.

Edges have a label like vertices, but also have the concept of extra labels. The main label behaves like a vertex label with the usual in-place editing and the extra labels do not have in-place editing. The primary reason for adding the extra labels was to support multiplicity in UML diagrams. They are separated from the main label to simplify usage for those only

requiring one label and because they do not behave in the same manner for in-place editing.

The actual path the edge takes is held in a `GeneralPath` object, a `Graphics2D` utility object that consists of a sequence of `java.awt.Shapes` and inherits from `Shape` itself. The start and end drawings on edges, which often consist of some type of arrowhead, are also `Shape` objects. The positions through which the edge passes are called points and a default edge has two, the start point and the end point. A point may, in fact, be a real point or a port object. Any additional points to these two are called control points and the line shape of the edge is drawn as a sequence of individual line shapes between each sequential pair of points in the `points` list. Note that for this reason, the collection that stores the list of points must be ordered.

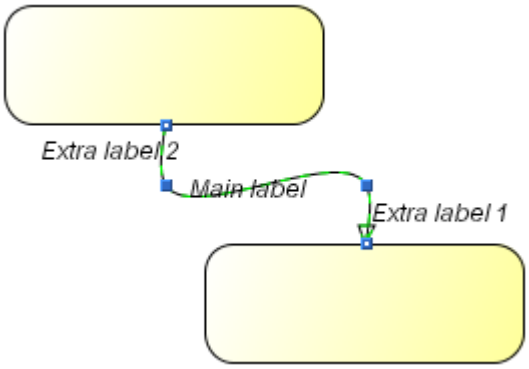

Illustration 23 : An edge with its main label and two extra labels. The edge has two control points and the line style is set to GraphConstants. STYLE_SPLINE. Since the edge is selected all points on the edge are indicated by the edge handle

As the comments at the top of `EdgeView` suggest, there are some class type assumptions made about the renderer in about 5 of the methods. If you subclass `EdgeView` and provide your own renderer, you must re-implement these methods referencing your own renderer type instead.

3.4.4.2 The Cell Renderers

3.4.4.2.1 PortRenderer

The `PortRenderer` is a simple `JComponent`. Have a look at the `getRendererComponent` method, here is where you need to install the attributes of the current cell view being painted. Remember that there is only one `PortRenderer` instance for all `PortViews` and so when we paint each one we have to setup the renderer for the current port view. This principle extends to all graph cell renderers using the flyweight pattern, the port renderer is just a simple example of this. This is why the cell view and the three cell states, `selected`, `preview` and `focus` are passed into `getRendererComponent()` and stored in the renderer's own variables. In the `paint` method the renderer uses these stored variables to draw the cell in the appropriate manner. `selected` indicates whether or the cell is selected, `preview` is whether or not the cell is

being drawn in live-preview (the XOR'ed preview you get of the graph whilst dragging before you release the mouse) and `focus` is whether or not the cell is currently the focus (this can be a different state to `selected`).

3.4.4.2.2 VertexRenderer

The `VertexRenderer` inherits from `JLabel`. This provides a lot of functionality for free, although seemingly simple tasks can be constrained by the use of a `JLabel`. Again, `getRendererComponent()` sets the renderer up for a specific view and the cell view states. In addition to storing these states local, `VertexRenderer` adds an internal method called `installAttributes()`. This performs the task of obtaining the attribute map of the vertex and storing all the visual attributes that are taken into account during in the paint method locally in the class. Note that most of the attributes belong to `JLabel` rather than `VertexRenderer` and this gives some idea of the usefulness of using `JLabel` as the parent class.

Most of the painting functionality lies in the parent class, apart from the painting of the selection border and of gradient color fills. As you would think, the `selected` flag passed into `getRendererComponent` is used is trigger the painting of the selection border. `getPerimeterPoint()` is where the actual calculation of where an incoming edge meets the boundary of the vertex is performed. Note that for a rectangular vertex the calculation isn't trivial, the simplest `getPerimeterPoint` implementation is actually for a circular vertex.

3.4.4.2.3 EdgeRenderer

`EdgeRenderer` follows the same pattern of installing the view, its state and its attributes, though it requires somewhat more code since it inherits from `JComponent` and must perform its own label painting. The extra functionality also presents another issue with synchronizing the renderer and the cell view when one of the public methods is called without the use of `getRendererComponent`. For example, if you call `getLabelPosition(EdgeView view)` to determine the position of the main label, the attributes necessary to determine the label position must be installed. There is a method `setView()` to perform this function, and you will see it used near the top of many of the public get methods to set the cell view and install the attributes. This can mean that for repeated work on the same edge there can be redundant attribute installation which can cause a performance hit, usually small in total percentage terms, however.

If you extend `EdgeRenderer` or attempt to implement your own version, bear in mind the requirements to ensure attribute installation always occurs. If you find that edges are being drawn in the wrong place or with the label of another edge, you have probably missed out a call to `setView` or the equivalent method in your own class.

Unique to the `EdgeRenderer` class are the `createShape` and `createLineEnd` methods. There are three `Shape` objects in the `EdgeView` that are created as necessary in `createShape`, these are `beginShape`, `endShape` and `lineShape`. `lineShape` is the sequence of `Shapes` between sequential pairs of points in the edge, each one drawn

depending on the line style, the dash pattern applied and so on. `beginShape` and `endShape` are the decoration, usually arrowheads, that may be placed are either end of `lineShape` and their creation is dealt with in `createLineEnd`. If you are looking to create new line styles or end decorations, these are the methods you need to adapt.

In the mechanism of installing cell view attributes in a renderer prior to painting, there is an issue when this functionality can be called from within more than one thread. If you do need to perform graph structure operations in one thread and painting in another thread you should split the event firing/catching mechanism that links these areas so that all attribute installation and painting occurs within one thread only.

3.5 Using Cells

3.5.1 USING VERTICES

In this section we will look at the various built-in features available for displaying vertices. As mentioned, the default vertex renderer inherits from `JLabel`. `JLabel` can display text and/or an icon.

3.5.1.1 Bounds

One of the basic concepts of all cells is its bounds. The bounds of a cell is the minimum rectangle that completely encloses that cell. JGraph uses double co-ordinates throughout and so the type of any cell's bounds is `Rectangle2D.Double`. The bounds of all cells are available through the `GraphConstants.BOUNDS` key in their storage attribute map. Since the position and the dimension of vertices are particularly useful data, `VertexView` stores a cached value of the bounds in the member variable named `bounds`. This may be accessed through the `getBounds()` method on that class.

It was mentioned previously that the `update()` method in cell views is a good place to put code that performs updating functions that need processing, not only during graph cell changes, but also during live-preview changes. It is `VertexView.update()` that updates the cached bounds value in `VertexView`, it also ensures that the value for bounds is non-null:

```
bounds = GraphConstants.getBounds(allAttributes);
if (bounds == null) {
     bounds = allAttributes.createRect(defaultBounds);
     GraphConstants.setBounds(allAttributes, bounds);
}
```

You generally won't need to perform any `setBounds` calls for interactive manipulation of the graph, JGraph takes care of this for you. If you wish to programmatically position or resize nodes, create a nested map of cell/transport map pairs, as described in Chapter 2, and

pass the new bounds values into the `edit()` call.

Whenever obtaining the bounds of a cell, you should do so from the cell view. If you have the cell view object, `getBounds()` provides a convenient method to do so. If you do not, there is a utility method in the `JGraph` class called `getCellBounds(Object cell)` which will return you the bounds value of the cell view for the cell passed in as the parameter. Another useful utility in the `JGraph` class is `getCellBounds(Object[] cells)`. This takes an array of cells and returns the total bounds of the according cell views, i.e. the minimum bounding rectangle of all of the cells.

3.5.1.2 Constraining Vertex Bounds

There are occasions when you want to force the dimensions of a vertex to be equal horizontally and vertically. Obvious examples are when the shape of a vertex must be a square, not just a rectangle, or a circle instead of a general ellipse.

```
GraphConstants.setConstrained(map, true);
```

will cause JGraph to enforce this condition where the map is a transport map applied to the cell during an edit call or the storage map of the cell before the cell is inserted.

3.5.1.3 Resizing and Autosizing

When you insert a cell you may want to ensure that the label in the cell (whatever the `userObject` of the cell returns in its `toString` call) is entirely visible. Getting the font metrics, calculating the width for the given font and `String` value would be tedious and so calling:

```
GraphConstants.setResize(map, true);
```

will cause the cell to be resized upon insertion so that the label is fully visible. This is a one-off effect, however. JGraph will remove the `GraphConstants.RESIZE` key from the storage map of the cell once the action is performed. If you wish one more resize to occur, set the attribute to true again and call `edit()`. Note that altering the cell's storage attribute map in-place and calling refresh and repaint will not work, resizing depends on a graph model or graph layout cache change event being fired, which requires an `edit` call.

Of course, setting the cell to resize on every edit is not practical if you want the cell to always be set to its preferred size. Instead, you should use the `AUTOSIZE` key:

```
GraphConstants.setAutoSize(map, true);
```

This will set the cell to its preferred size after model and layout cache changes. One difference you will notice with cells that have autosize enabled is that they do not have a handle when selected. After all, it is fairly pointless to allow the user to resize a cell if the application will simply revert the change immediately.

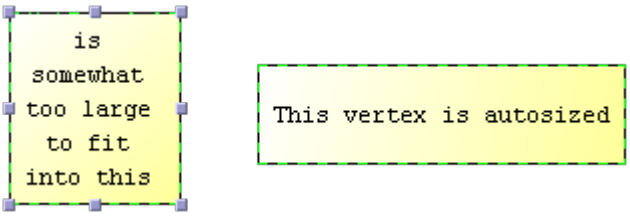

Illustration 24 : The right-hand vertex is autosized, note both vertices are selected but the autosized one has no handles

The underlying mechanism to determine the size of a cell upon a resize or autosize event is actually `getPreferredSize()` in the cell renderer. By default a vertex will return a rectangle slightly larger than the minimum bounding rectangle of the icon and text of the `JLabel`, if either exist.

3.5.1.4 Icon

`JLabels` are capable of displaying text and an `Icon`, the icon is set using:

```
GraphConstants.setIcon(Map, Icon)
```

The specified icon will be displayed within the cell, if `AUTOSIZE` is enabled this will ensure that the bounds of the cell are at least large enough to accommodate the icon. Without autosize enabled the bounds of the cell might clip the icon or be somewhat larger than the icon requires. You might wish to adapt the paint method of the renderer if you want to scale the icon, this could be performed within a subclass of `VertexRenderer` as follows:

```
public static class ScaledVertexRenderer extends VertexRenderer {
     public void paint(Graphics g) {
          Icon icon = getIcon();
          setIcon(null);
          Dimension d = getSize();
          Image img = null;
          if (icon instanceof ImageIcon)
               img = ((ImageIcon) icon).getImage();
          if (img != null)
               g.drawImage(img, 0, 0, d.width - 1, d.height - 1,
graph);
          super.paint(g);
     }
}
```

The methods available in `JLabel` are repeated in `GraphConstants` enabling you to align the contents of the label along the Y-axis:

```
GraphConstants.setVerticalAlignment(Map, int);
```

where the int parameter may be one of the following constants defined in `SwingConstants`: `TOP`, `CENTER` (the default), or `BOTTOM`. Also for alignment of the label's contents with the X-axis

```
GraphConstants.setHorizontalAlignment(Map, int);
```

where the int parameter may be one of the following constants defined in `SwingConstants`: `LEFT`, `CENTER` (the default for image-only labels), `RIGHT`, `LEADING` (the default for text-only labels) or `TRAILING`.

Illustration 25 : An Icon aligned using various horizontal and/or vertical alignment settings

3.5.1.5 Label Text

The text component of the `JLabel` may also be aligned relative to the icon. Vertical relative alignment is performed using:

```
GraphConstants.setVerticalTextPosition (Map, int);
```

where the int parameter may be one of the following constants defined in `SwingConstants`: `TOP`, `CENTER` (the default), or `BOTTOM`. Horizontal relative alignment is achieved using:

```
GraphConstants.setHorizontalTextPosition(Map, int);
```

where the int parameter maybe one of the following constants defined in `SwingConstants`: `LEFT`, `CENTER`, `RIGHT`, `LEADING`, or `TRAILING` (the default).

`GraphConstants` also provides the `setFont()` method to enable you to configure the font of the text displayed and `GraphConstants.setForeground()` to set the color of the text. Full details of how to use fonts are beyond the scope of this manual, see any good reference on Java graphics for more information.

3.5.1.6 Borders

Borders are a Swing function that enables you to paint aesthetically pleasing borders around the edges of your Swing components. Since the standard vertex is rendered as a `JLabel`, you can set a border to your vertices using standard borders with:

```
GraphConstants.setBorder(Map, Border);
```

More information on the types of `Borders` available can be found at the `Border` API package summary:
http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/border/package-summary.html.
Also at http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/BorderFactory.html you will

find useful factory methods simplifying the process of creating those borders. For example:

```
GraphConstants.setBorder(map, BorderFactory.createRaisedBevelBorder());
```

creates a raised border of the type of effect you would see typically on a button.

```
GraphConstants.setBorder(map,
BorderFactory.createLineBorder(graph.getBackground(), 6) );
```

will create a blank border around the vertex using the background color of the graph to paint out. This is useful is you wish to have edges terminate a short distances from vertices rather than directly on the perimeter. The color may be also changed using `GraphConstants.setBorderColor()`.

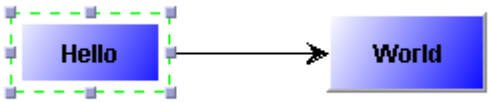

Illustration 26 : On the left a line Border of the color of the graph background. On the right a raised bevel Border

3.5.1.7 Colors

```
GraphConstants.setBackground(map, Color)
```

set the fill color of vertices to a constant color, whereas:

```
GraphConstants.setGradientColor(map, Color)
```

sets a gradient fill across vertex, starting white and progressively darkening across the vertex to the specified color.

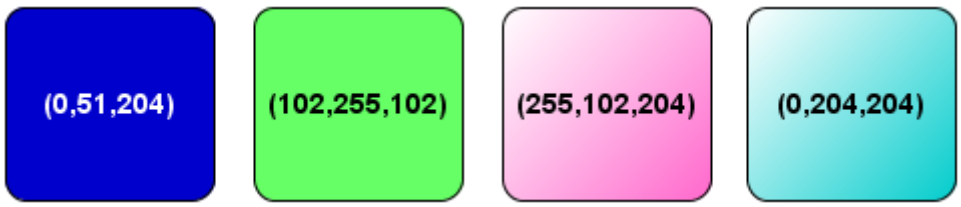

Illustration 27 : On the left two vertices filled using setColor, and on the right filled using setGradient. The (Red, Green, Blue) values of the colors used are indicated

3.5.1.8 Inset

`GraphConstants.setInset()` provides a means to place a buffered area around a label, so `getPreferredSize()` returns a `Dimension` large enough for the label plus the inset. This `Dimension` is used by the one-shot resize and autosizing functionality.

3.5.2 USING EDGES

3.5.2.1 Bounds

Bounds work slightly differently in edges. The values of `BOUNDS` for an edge is still the minimum rectangle that enclosed the edge, but its use is rather more limited than for a vertex. The bounds of an edge gives no indication where it starts or finishes, or what path it takes between those two points. The use of edge bounds should be limited to determining the clip bounds if you need to manually force a repaint.

3.5.2.2 Control Points and Routing

As mentioned previously, the `DefaultEdge` holds a collection of ordered points which describe the path the edge follows. At its simplest the edge will be drawn as a sequence of straight lines between these points. You can set these points using an ordered `List`:

```
GraphConstants.setPoints(Map, List)
```

However, JGraph also supports orthogonal, bezier and spline style drawings and the line style property of an edge is set using:

```
GraphConstants.setLineStyle(map, int)
```

on the attribute map of the edge passing in `GraphConstants.STYLE_ORTHOGONAL`, `GraphConstants.STYLE_BEZIER` or `GraphConstants.STYLE_SPLINE` as appropriate. Remember that these styles require control points, otherwise they will simply appear as straight lines.

Control points may either be added manually or using a routing method. We briefly mentioned earlier in this chapter the `Routing` interface defined inside the `Edge` class. A routing algorithm is generally a static class instance shared by all the edges being routed in that manner. You set routing for an edge using:

```
GraphConstants.setRouting(map, GraphConstants.ROUTING_SIMPLE);
```

to use `DefaultEdge.DefaultRouting`, the basic routing algorithm supplied with JGraph. `DefaultRouting` sets appropriate control points for the various line styles, saving you having to define the points yourself. `DefaultRouting` also sets the routing for self-loops, so that they visually leave the vertex and return to it.

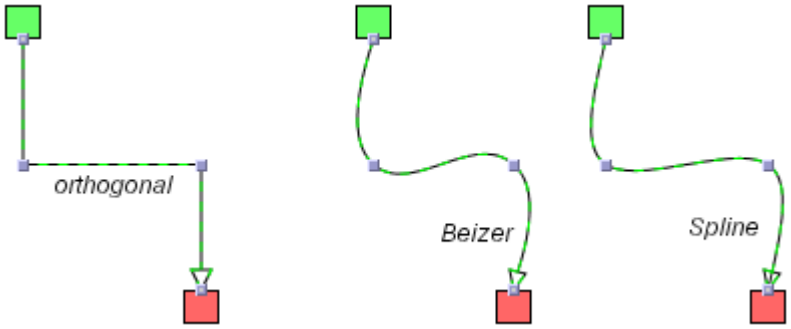

Illustration 28 : Three edges with the indicated line styles routed using the DefaultRouting algorithm

The `Routing` interface itself defines one method, `route()`, which takes the `Edge` to be routed and the list of points to be altered as parameters. Two implementations of parallel edge routers are available in the JGraphpad Community Edition, if you require a different custom implementation it is worth checking with the JGraph team to see if someone has already done it.

If you wish to restrict the interactive (using a mouse) addition or removal of control points to and from an edge, `setBendable(Map, false)` will forbid these actions for the specified edge even if an application supports such functionality.

3.5.2.3 Positioning edge labels

The configuring of label positioning is somewhat more flexible than that for vertices and any number of labels are supported. The main label displays, like for vertices, whatever the `toString()` method of the user object attached to the edge returns. Setting of the label position is performed using:

```
GraphConstants.setLabelPosition(Map, Point2D);
```

The point parameter defines the relative distance across the edge that the label lies in the X coordinate and the distance orthogonal to the edge that the label lies in the Y coordinate. The relative distance across the edge is measured from 0, at the start of the edge, to `GraphConstants.PERMILLE`, at the end of the edge. So, passing in:

```
Point2D point = new Point2D.Double(GraphConstants.PERMILLE/2, 0);
```

will result in the label being centered mid-way between the start and end points.

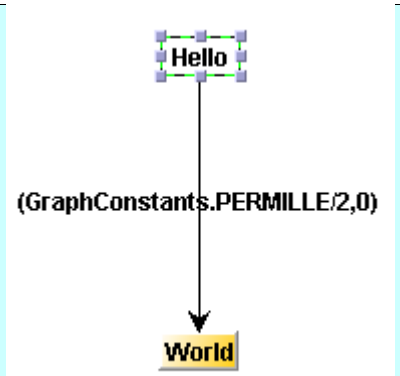

Illustration 29 : (GraphConstants.PERMILLE/2, 0)

```
Point2D point = new Point2D.Double(GraphConstants.PERMILLE/2, 100);
```

will result in the label being positioned mid-way between the x-axis positions on the start and end point and 100 pixels below the mid-point of the start and end points' y-axis positions. Note, when we write "below" we are talking about beneath the plane the edge makes when the start point is the left of the end point.

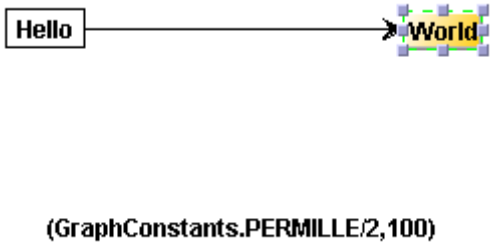

Illustration 30 : (GraphConstants.PERMILLE/2, 100)

You can also go beyond the bounds of the 0 to `GraphConstants.PERMILLE` range.

*Illustration 31 : (GraphConstants.PERMILLE*2, -50)*

The extra labels on edges are stored as `Objects`, whatever they return in their `toString()` method is what is displayed. It is unlikely you will use anything other than `Strings` as these objects. The positioning system is the same as for the main label. The method you use to set and position these extra labels are:

```
GraphConstants.setExtraLabels(Map, Object[])
GraphConstants.setExtraLabelPositions(Map, Point2D)
```

So using this code:

```
Object[] labels = {new String("0...*"),new String("1")};
Point2D[] labelPositions = {new Point2D.Double
      (GraphConstants.PERMILLE*7/8, -20), new Point2D.Double
      (GraphConstants.PERMILLE/8, -20)};
GraphConstants.setExtraLabelPositions(edge.getAttributes(),
     labelPositions);
GraphConstants.setExtraLabels(edge.getAttributes(), labels);
```

we can place a label at either end of the edge and slightly offset from the plane of the edge so that the labels are not overlapped by the edge. Again, note the y-offset is relative to the plane of the edge, so rotating the edge you still have the labels appearing in the correct relative positions.

Illustration 32 : Extra labels on an edge keeping their relative positioning after rotation, no comments about the odd state of my galaxy please

Another effect that is useful with regards to label drawing is the ability to force the label to be parallel with the edge:

```
setLabelAlongEdge(map, true);
```

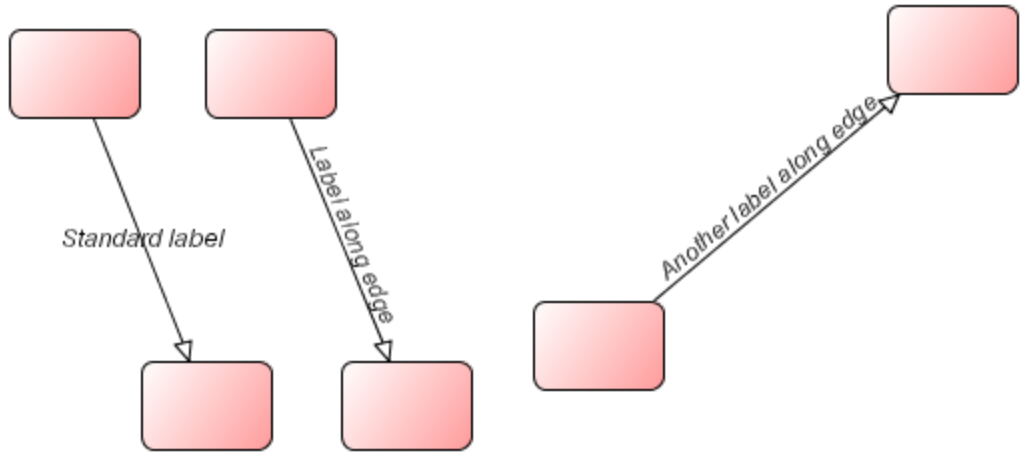

Illustration 33 : Labels being drawn parallel to their edges

The other consideration for edge labels is the color of the text is set using:

```
GraphConstants.setForeground(Map, Color)
```

3.5.2.4 Edge Styles

There exist a number of configuration options to change the appearance of the main line part of an edge (we will discuss end decorations next). The two simple options are:

```
GraphConstants.setLineWidth(map, 5); //sets the edge line width to 5
GraphConstants.setLineColor(map, Color.blue); // sets the edge line
color to blue
```

The other two edge style configuration methods (other than the line shape previously mentioned) are:

```
setDashPattern(Map, float[])
setDashOffset(Map, float)
```

These two values correspond to the last two parameters passed into the BasicStroke constructor:

```
BasicStroke(float, int, int, float, float[], float)
```

but the Javadocs for these parameters is less than helpful.

The dash pattern is a sequence of solid and clear lengths of edge that repeats throughout the edge. So, creating an array equal to {10, 10} would mean to edge is drawn as solid for 10 units, then clear for 10 units and this repeats along the edge. {10, 2, 2, 2} would mean solid for 10 units, clear for 2, solid for 2 units and clear for another 2, and repeat. Obviously, it only really makes sense to have an even number of entries to this array. This pattern is applied consistently, regardless of the shape or number of points in the edge. A static variable `GraphConstants.dash` representing a dash pattern of {5,5} is defined in case

you wish to save memory by using this single instance, instead of creating an instance for every cell attribute map.

Illustration 34 : Dash patterned edges with their pattern displayed

The dash offset, also known as the dash phase, determines how far into the dash pattern the drawing should be started. By default the dash offset is 0, setting it to 5 for the {10,10} dash pattern would result in the start of the edge drawing 5 units of solid line, then 10 units of clear line, then back to the 10 solid / 10 clear repeating pattern. Setting the value to 12 would result in the line starting with 8 units of clear line and then back to the 10 solid / 10 clear repeating pattern.

Illustration 35 : A {10,10} dash pattern using the indicated offset values

The most common application for the dash offset is for animating edges. JGraph purposefully never assumes any multi-threading, so it can't directly offer such animation. However, a timer thread triggering a method in the event dispatch thread to alter the dash offset and update the edge is simple to implement. For each update you would need to get a collection of the edges' cell views to be updated, change the dash offset value in-place, then call refresh on the edge and finally repaint the entire affected area. This would require a static variable holding the current value of the dash offset, which would be decremented after each timer tick. The dash offset call itself would need to use the modulus of the current dash offset value so it repeats between the limits of the dash pattern range. For example, say the dash pattern is {10,10}, the complete range of the modulus result used for the dash offset needs to be 0 to 19:

```
GraphConstants.setDashOffset(map, 20 - Math.abs(dashOffset % 20));
```

should be used in this case. Note that the dash offset needs to be decremented in order to get animation from the start of the edge to the end, and vice versa. To speed up the animation, decrease the timer interval or decrement the dash offset value by more than one on each tick.

3.5.2.5 Edge end decorations

At either end of the edge you can configure end decorations to be drawn, this usually consists of arrowheads. JGraph provides a number of commonly used end decorations that may be enabled using:

```
GraphConstants.setLineBegin(Map, int)
GraphConstants.setLineEnd(Map, int)
```

where the int parameter is one of the options available in `GraphConstants`. To remove a decoration use the method above for the appropriate end of the edge and pass in `GraphConstants.ARROW_NONE`. You may also remove the attribute entirely from the attribute map using `GraphConstants.setRemoveAttribute()`, as discussed in Chapter 2.

Each of these end style may be filled or not using:

```
GraphConstants.setBeginFill(Map, boolean)
GraphConstants.setEndFill(Map, boolean)
```

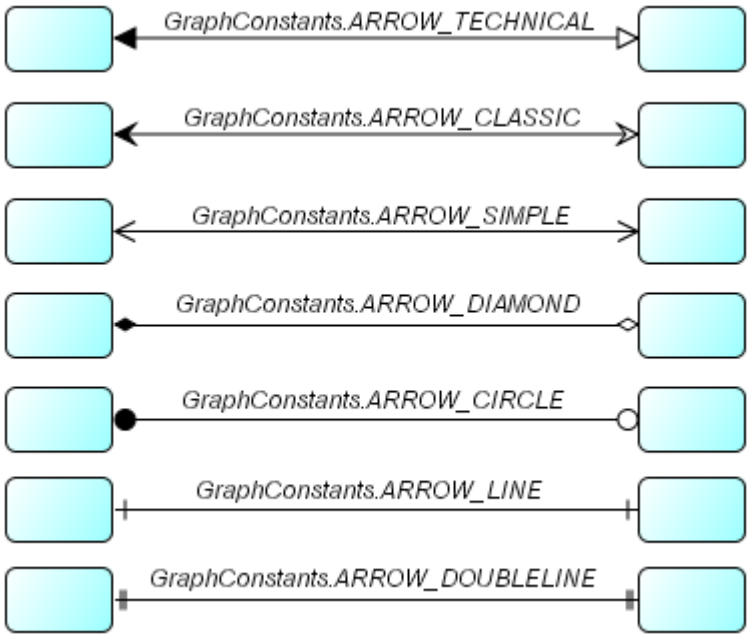

Illustration 36 : The available line end decorations, with fill set to true on the left-hand side

If you wish to change the size of the end decorations, this is done using:

```
GraphConstants.setBeginSize(Map, int)
GraphConstants.setEndSize(Map, int)
```

3.5.2.6 Connections restraining

It is possible, on a per-edge basis, to set whether or not the edges may be connected or disconnected interactively using the mouse. The methods are self-explanatory:

```
setConnectable(Map, boolean)
setDisconnectable(Map, boolean)
```

3.5.3 ATTRIBUTES FOR BOTH VERTICES AND EDGES

3.5.3.1 Constraining Basic Editing Functions

The next three attributes require little explanation. `GraphConstants.setSizeable()` controls whether or not cells may be resized using the mouse. If set to false, the user is not presented with any handles to resize with. `GraphConstants.setMoveable` determines whether or not the cell may be moved (repositioned, not resized) interactively. `GraphConstants.setEditable(false)` disables in-place editing for labels. You could set this value to `false` if you want double-clicking on a cell to perform a different function to label editing. However, you might also like to change the number of mouse clicks required to start editing using `JGraph.setEditClickCount()`.

Whether or not cells can be moved on a per-axis basis can be configured using

```
GraphConstants.setMoveableAxis(Map, int)
```

where the int parameter is either `GraphConstants.X_AXIS` or `GraphConstants.Y_AXIS`. Obviously, to forbid moving entirely use `setMoveable`. Setting the moveable axis to being `X_AXIS` causes the vertex to only be movable horizontally and `Y_AXIS` causes the vertex to only be movable vertically.

3.5.3.2 Opaqueness

```
GraphConstants.setOpaque(Map, boolean)
```

will pass the `boolean` value up to the `JComponent.setOpaque()` method of the renderer of the cell. As the Javadocs for that method say "*If true the component paints every pixel within its bounds. Otherwise, the component may not paint some or all of its pixels, allowing the underlying pixels to show through.*". In the case of vertices constant background fill and gradient fill colors are not painted if `opaque` is set to `false`. Text, icon images and borders are still painted regardless of the setting of this attribute. In the case of edges, the default implementation of an edge is not affected by this attribute, but you should take it into account it you were to produce your own, more complex, implementation of an edge.

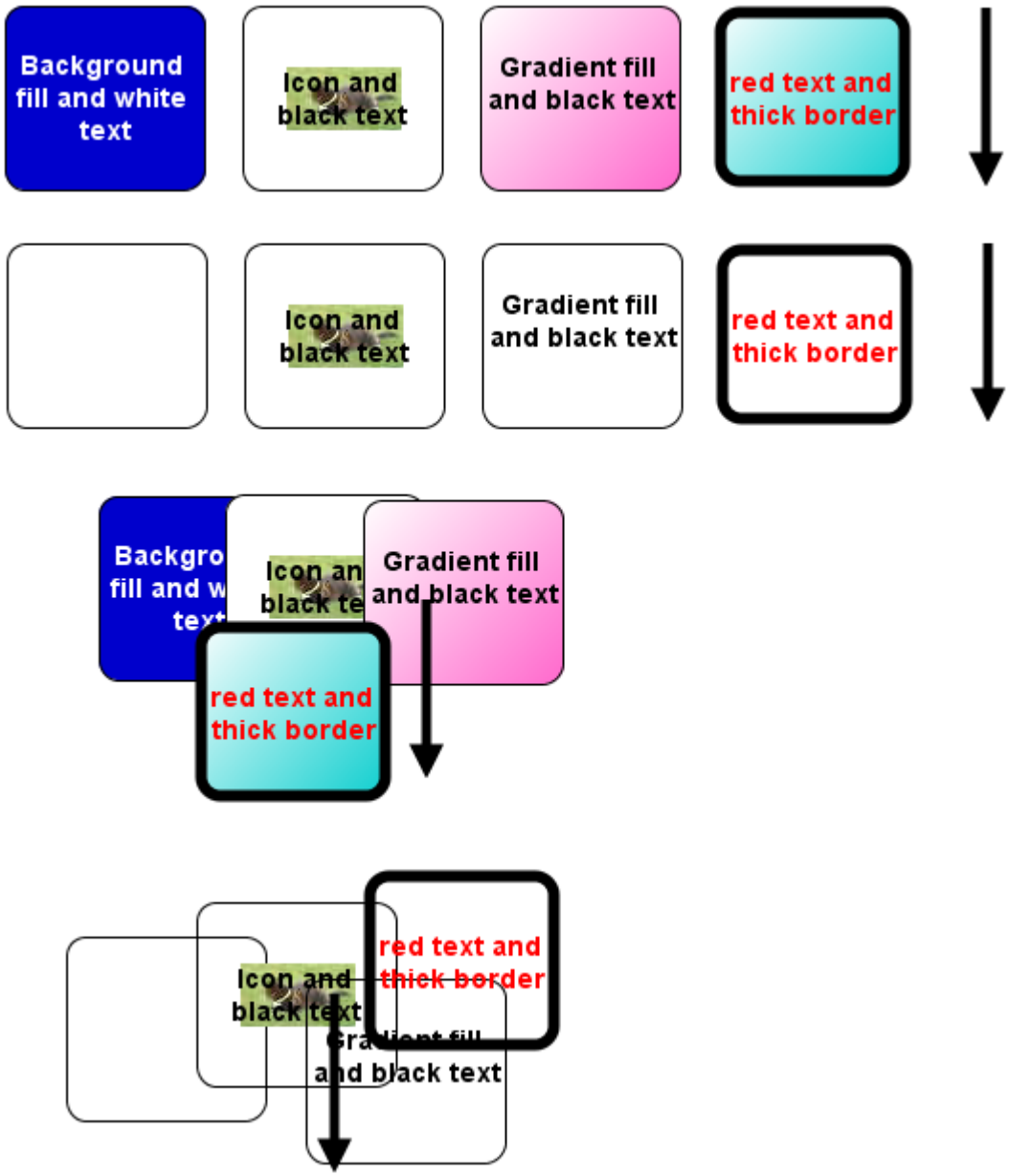

Illustration 37 : The cells on the first and third sets of cell (from the top) are opaque, the second and bottom sets of cells have opaque set to false

The figure above shows a set of cells, first with opaque set to `true` and then to `false`. It also shows both sets of cells overlapping each other to various degrees. The opaque versions completely obscure those they are in front on, whereas, in the case of the non-opaque cells overlapping, just the borders, icon and labels obscure cells painted beneath them.

3.5.3.3 Selection

```
GraphConstants.setSelectable(map, boolean)
```

determines whether or not the cell may be selected. Setting this value to `false` basically causes the cell to stop reacting to any interactive function. The cell may not be resized,

moved (unless it is a connected edge that moves along with a connected vertex that is being moved) or have in-place editing performed.

3.5.4 USING PORTS

3.5.4.1 Port Positioning

When a port is attached as a child of a vertex, by default it is what is know as a floating port. This means it has no fixed position, any edge connecting the vertex will be seen to terminate at the boundary of the vertex. Note that the edge isn't just hidden by the vertex, floating ports terminate edges exactly on the boundary, otherwise known as the perimeter point, of cells and so arrowheads are visible and correctly placed. This default implementation works for the majority of applications since it resolves the issues associated with edges traveling across vertices to a fixed point on the vertex boundary. Note, this relies on the `getPerimeterPoint()` method on the renderer of the vertex being implemented correctly.

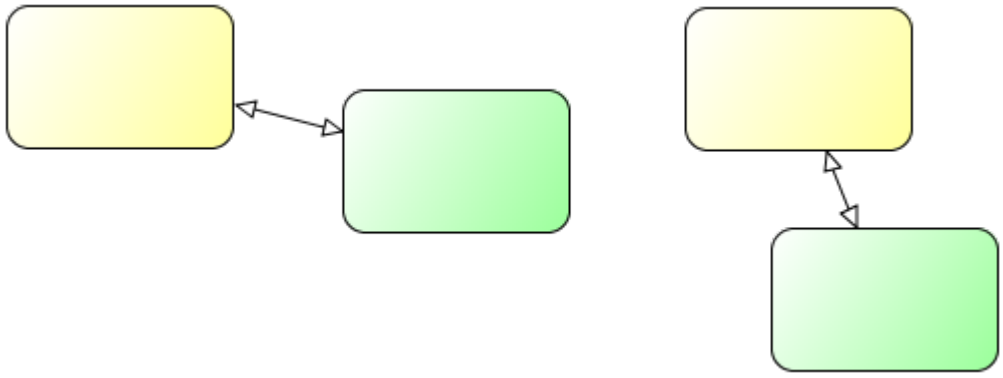

Illustration 38 : Two vertices connected by an edge using their floating ports. Note the edge terminates correctly on either vertex regardless of the edge direction

A second type of positioning for ports involves offsets. Invoking:

```
GraphConstants.setOffset(Map, Point2D)
```

on a port cell fixes the port position relative to the cell. A value of (0,0) corresponds to the top left corner of the cell and (`GraphConstants.PERMILLE`, `GraphConstants.PERMILLE`) corresponds to the bottom right-hand corner of the cell. Since the value are a proportion of the cells dimensions, the ports are always placed in the same relative positions regardless of the size of the vertex.

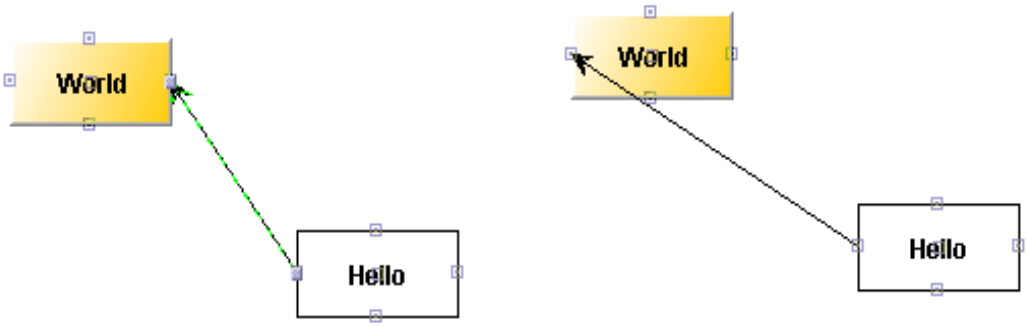

Illustration 39 : The HelloWorld example with offset ports added at (0, GraphConstants.PERMILLE/2) , (GraphConstants.PERMILLE/2, 0) , (GraphConstants.PERMILLE/2, GraphConstants.PERMILLE) , (GraphConstants.PERMILLE, GraphConstants.PERMILLE/2). Connecting edges between offset ports means it is possible that the edge or vertex might overlap each other. This doesn't happen with floating ports.

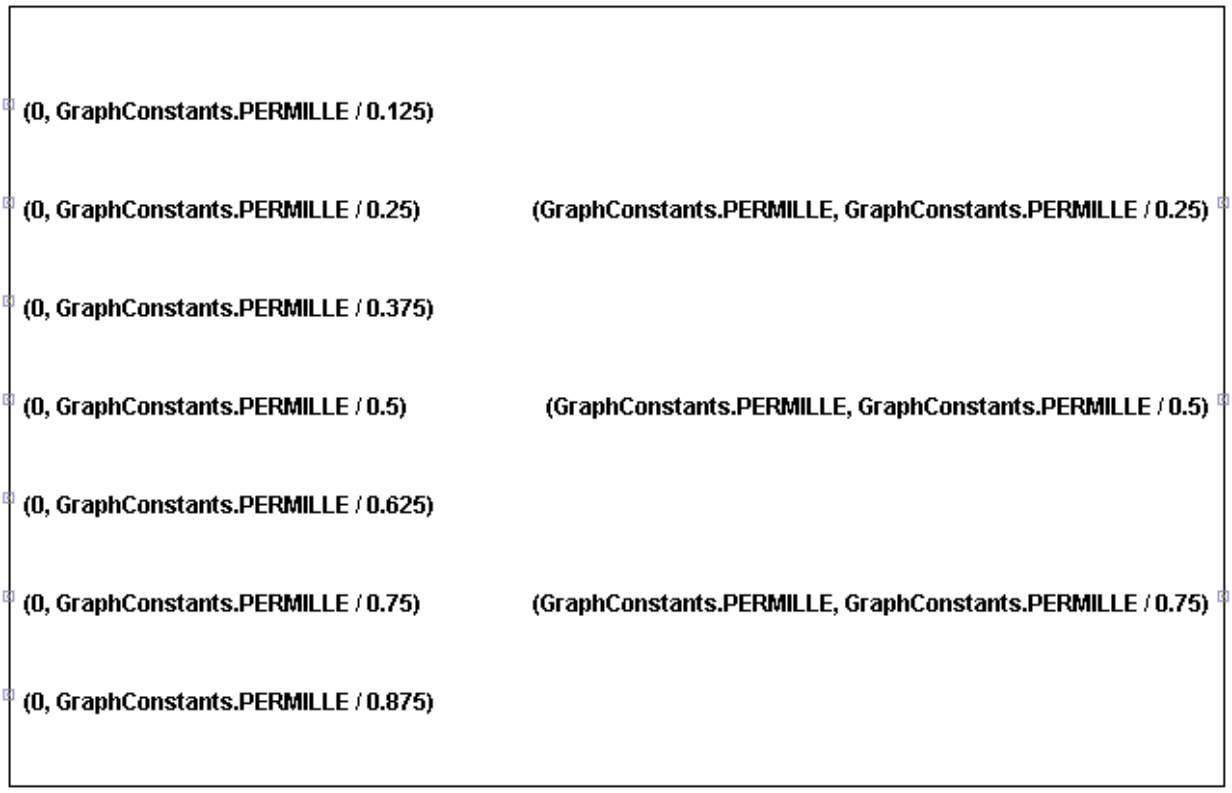

Illustration 40 : A vertex containing a number of visible ports with their offset values shown (the entire rectangle is the vertex, the labels belong to the ports in this example)

A third method of setting the port position is to do so in absolute coordinates relative to the origin of the vertex. With absolute ports their positions relative to the dimensions of the vertex will not remain the same through resizing, but their position relative to the vertex origin will. Which axis are absolute is configurable independently:

```
GraphConstants.setAbsoluteX(Map, boolean)
GraphConstants.setAbsoluteY(Map, boolean)
```

or both together:

```
GraphConstants.setAbsolute(Map, boolean)
```

After setting this flag, you position the ports using the `GraphConstants.setOffset()` method again, this time the `Point2D` parameter is the absolute offset from the vertex origin.

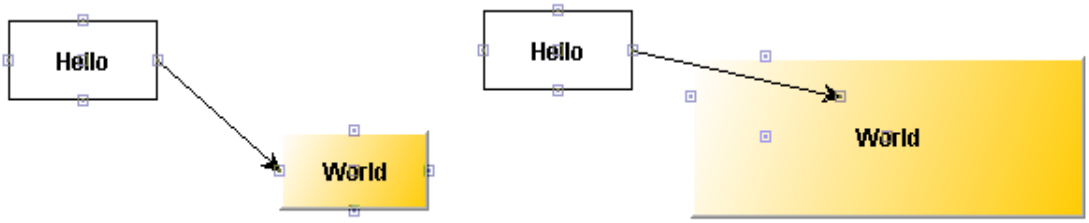

Illustration 41 : Absolute offset ports often do not appear correctly when the parent vertex is scaled

The fourth method is using port anchors, which involves defining another port that this port will be offset relative to. This anchor is the anchor referred to in the `Port` interface in `getAnchor` and `setAnchor`. Setting another port as anchor makes that port the origin for this port, instead of the vertex origin. You can still define the offset as a proportion of the vertex dimensions using just `setOffset`, or you can define the offset as an absolute value using `setAbsolute(map, true)` and `setOffset()`. The anchoring mechanism is useful if you wish to define a chain of ports that have fixed positions relative to each other. *Note: Port anchors are disabled in JGraph 5.6.2.1.x pending a bug resolution.*

3.6 Summary

- A range of configuration options for visual attributes of the default cells is available

through the accessor methods of `GraphConstants`.

- To add a new cell type, define the new cell class, its view class and its renderer class. Automate the creation of the view using the cell view factory and ensure the view returns the renderer in the appropriate method(s).
- If you wish to add new functionality to a cell you might do so by 1) subclassing attribute map and adding new attribute type to support the new functionality, 2) by providing the functionality through methods and variables on the cell class, or 3) by storing the data in the user object of the cell.
- One important note about cells is that you can only pass cells into `edit`, `insert` and `remove` calls, never cell views.

4 Advanced Editing

4.1 Grouping

Grouping, within JGraph, is the concept of logically associating cells with one another. This is commonly referred to as the concept of sub-graphs in many graph toolkits. Grouping involves one or more vertices or edges (ports are generally not discussed with grouping functions, even though they are children of other cells) becoming children of a parent vertex or edge (usually a vertex) in the graph model data structure. This causes the parent cell, also known as the group cell, to take the bounds of the minimum bounding rectangle that encloses all of the children cells. Once grouped, the group cell may be moved and resized like a stand-alone cell, but the operation affects all of the children cells as well.

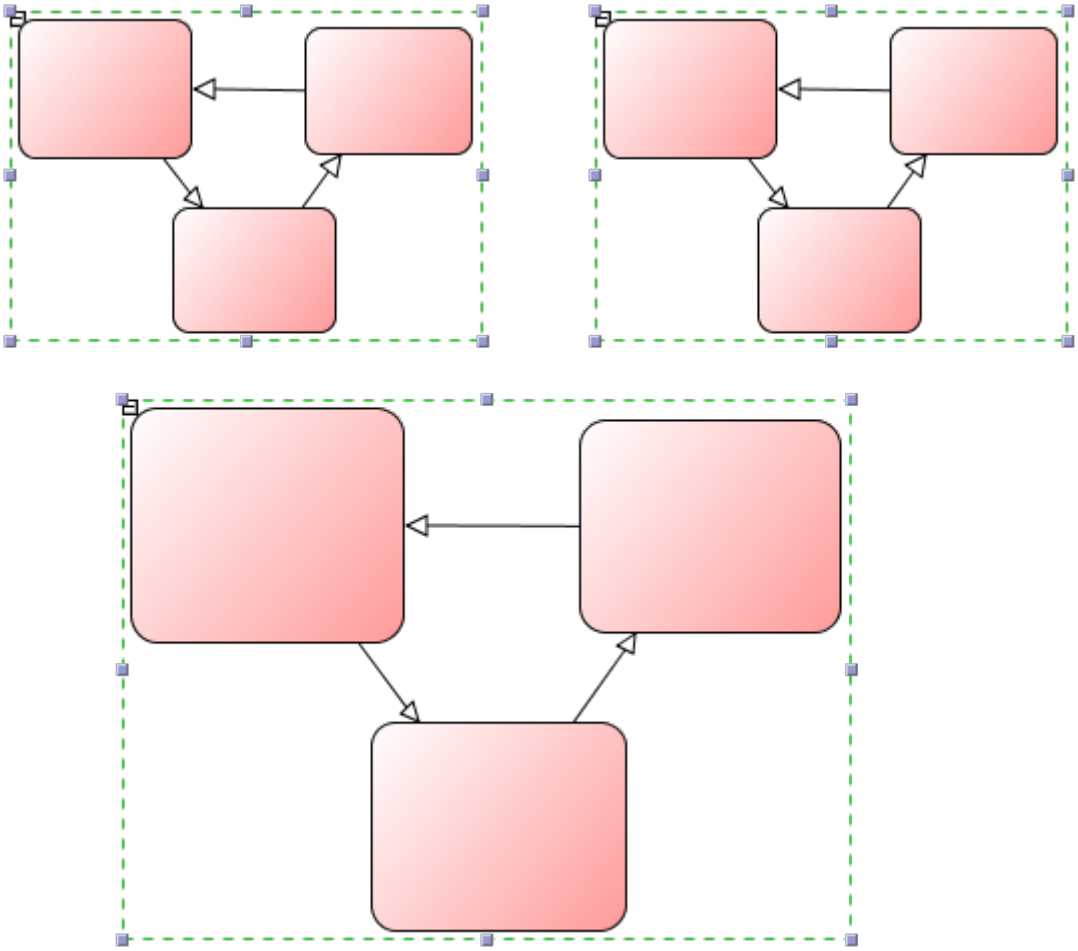

Illustration 42 : Moving a group and resizing it

Moving a group cell causes an equal translation on the children cell(s), scaling a group cell causes the children cells to be scaled by the same proportions.

4.1.1 GRAPH MODEL REPRESENTATION OF GROUPING

As mentioned, cells that lie within a group are child cells of the group cell. This relationship can be nested any number of times, so a group can contain another group, and so on.

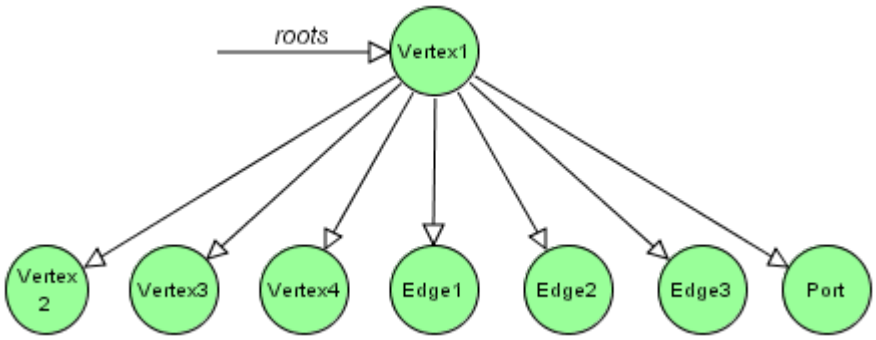

Illustration 43 : How the Graph Model will look after 3 vertices and 3 edges are grouped (additional ports not shown for clarity)

The simplest method to group cells programmatically is to set up the parent/child relationship prior to all the group cell(s) being inserted. Note only the topmost group cell needs to be specified in the `insert` call if the child relationships are correctly formed prior to the `insert`. This could be done using the `add()` method available in `DefaultGraphCell`:

```
vertex1.add(vertex2);
```

in the same way we added ports to vertices in the `HelloWorld` example. You may also use the constructor of `DefaultGraphCell` that accepts an array of children:

```
Object[] children = {vertex2, vertex3, vertex4, edge1, edge2, edge3};

DefaultGraphCell vertex1 = new DefaultGraphCell(new String"Vertex1",
                                                null, children);
```

`JGraph.getDescendants(Object[])` provides a method to obtain all of the descendant cells (children) of those specified in the single parameter. Along with the `getRoots` method, these two methods combine to make the primary command you should use to obtain all cells in the graph:

```
graph.getDescendants(graph.getRoots());
```

Just obtaining the roots will only work as long as there are no group structures.

Note that you must explicitly create the group cell in the normal way you might create any cell. Grouping together any number of cells will not automatically create a parent cell. There is a helper method in the `GraphLayoutCache`:

```
insertGroup(Object group, Object[] children)
```

that groups the cells in the array parameter under the group cell and performs the `insert` command.

These methods mentioned, however, do not allow for the changing of the parent/child relationship during `edit` and calls, nor are they capable of adding the grouping operation to the undo history as part of an `insert()` call. For this, you must use a `ParentMap`.

4.1.2 PARENTMAP

The `ParentMap` class defines the parent/child relationships of cells. It can be used in the appropriate `edit()` and `insert()` calls in `GraphModel` and `GraphLayoutCache` that have a ParentMap as one of their attributes. `ParentMaps` are stored as part of the graph model edit, or graph layout cache edit, so any changes to the parent/child relationship(s) are undoable. The idea with `ParentMaps` is to describe the parent/child relationship you would like to alter the graph model to represent and pass the parent map to the `edit` or `insert` method.

`ParentMaps` may be created in one of three ways. The first is to pass the children and parent to the `ParentMap` constructor:

```
ParentMap parentMap = new ParentMap( children, parent );
```

this causes the array of children to have the specified parent in the parent map. To invoke this change call:

```
graph.getGraphLayoutCache.edit(null, null, parentMap, null);
```

note that you can also make changes to cell attributes using the first parameter at the same time as changing the group structure using the `ParentMap`. Within the `edit` call the change made to the group structure will be stored as well as the grouping structure prior to `the` edit call. This enables undo/redo to be able to restore the current and previous states.

The second method of creating a parent map is to construct the class either using the default constructor, or the constructor just mentioned, and then to add further entries using the `addEntry()` or `addEntries()` methods. `addEntries` allows you to assign multiple children to a single parent and addEntry add a single child and associated single parent to the parent map. These methods add one or more `Entry` objects to the `ParentMap`, each `Entry` object representing one parent/child relationship.

When we describe the `ParentMap` and how it is composed of some number of `Entry` pairs, remember that the parent of any `Entry` pair may be `null`. This is how you represent a parentless cell, i.e. a cell you want to add to the model roots. Generalizing the whole concept of a parent map, there are three operations you can use it to describe. Below we show those three operations:

1. You currently have a cell with no parent, you want to assign it a parent. Add an entry to the `ParentMap` with the cell as the child and the new parent.

```
ParentMap pm = new ParentMap();
```

```
pm.addEntry(childCell, groupCell);
```

2. You currently have a cell with a parent, you want it to have no parent. Add an entry to the `ParentMap` with the cell as the child and set the parent to `null`.

```
Object[] children = {childCell};
ParentMap pm = new ParentMap(childCell, null);
```

3. You currently have a cell with a parent, you want to assign it a different parent. Add an entry to `ParentMap` with the cell as the child and the new parent.

```
ParentMap pm = new ParentMap();
pm.addEntry(childCell, newGroupCell);
```

Other examples you might find useful are the operation to group selected cells:

```
DefaultGraphCell group = new DefaultGraphCell();
graph.getGraphLayoutCache().edit(null, null, new ParentMap
                            (graph.getSelectionCells(), group), null);
```

and the operation to ungroup selected cells:

```
graph.getGraphLayoutCache().edit(null, null, new ParentMap
                            (graph.getSelectionCells(), null), null);
```

4.1.3 GROUP INSETS

`GraphConstants.setInset()` can also be used on group cells to provide a boundary between the minimum bounding rectangle of the child cells and the group cell itself.

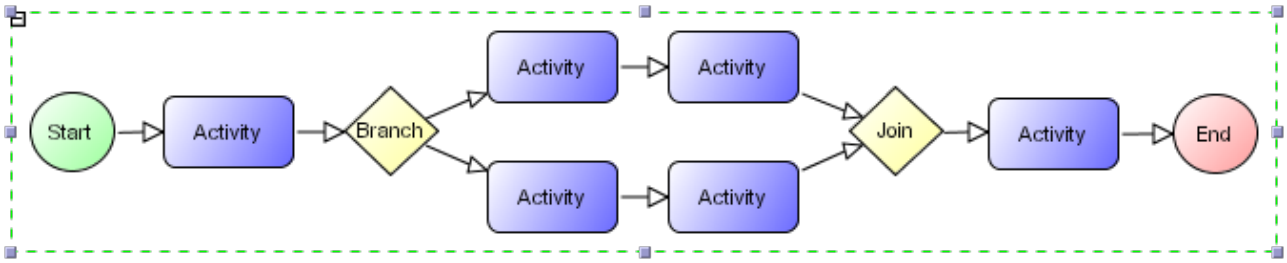

Illustration 44 : A group cell with an inset of 10

4.1.4 MOVE INTO/OUT OF GROUPS

In the `JGraph` class there exist two methods, `setMoveIntoGroups(boolean)` and

`setMoveOutOfGroups(boolean)`. These determine whether or not to make a cell part of a group cell when you drag the cell into or completely out of a group cell. So, with `setMoveIntoGroups` set to `true`, moving cells so that the mouse position is inside the bounds of an existing visible group cell will cause the cells to become direct child of that group. With `setMoveOutOfGroups` set to `true`, dragging a child within a group cell completely out of the group cell will cause the cell to become a root cell, i.e. have no parent.

4.1.5 REMOVING CHILD CELLS

Using the `remove()` call on cells that are part of a group structure is slightly different to the pattern for other editing calls. If you call remove on the vertices numbered 3 and 4 in the figure below:

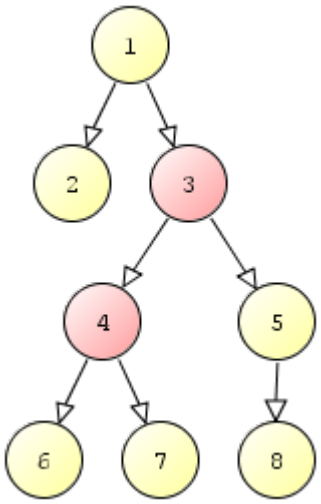

Illustration 45 : A group structure before cells 3 and 4 are removed

those cells will be removed from the group structure and leaving:

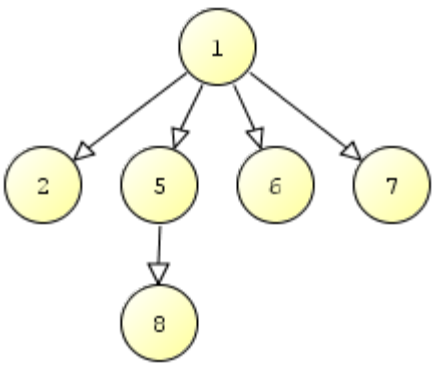

Illustration 46 : The group structure after the remove() call

A call to `remove()` only passing in vertex1 would be, essentially, an ungroup command and the same applies to any cell which acts as a group. If you wish the remove the entire group structure you need to call `JGraph.getDescendants(Object)` (or use the method of the same name in the DefaultGraphModel) on the topmost parent cell to obtain a collection containing the cell and all its children and then pass all these cells to the `remove()` method.

4.2 ConnectionSet

`ConnectionSet` is the final of the three main parameters to `insert` and `edit` calls, the other two being the nested map of attributes and the parent map. A `ConnectionSet` describes the connection state of any number of edges and so is also stored as part of any edit change object to enable correct undo/redos.

The design of `ConnectionSet` is similar to that of the `ParentMap`, there is the overall class that holds one or more entries, or connections in this case, and they may be set up through the constructor, individually or as a collection.

```
ConnectionSet(Object edge, Object port, boolean source)
```

creates a simple `ConnectionSet` associating the specified port and edge and also indicating whether or not the port is at the source or target end of the edge. This creates a `Connection` object, which is an inner class of `ConnectionSet`, and adds it to the set of connections held. You can also create the set of connections yourself and pass it in using:

```
ConnectionSet(Set)
```

Individual connections can be created using:

```
connect(Object edge, Object source, Object target)
```

which sets the edge within the `ConnectionSet` to have the specified source and targets,

```
connect(Object edge, Object port, boolean source)
```

which sets the edge to be connected to the port within the `ConnectionSet` and whether or not it is the source or target port is indicated by the `boolean` parameter. Also:

```
disconnect(Object edge)
```

sets the edge as being disconnected at both ends within the `ConnectionSet` and

```
disconnect(Object edge, boolean source)
```

disconnects just the source or target end, as specified, within the `ConnectionSet`.

Also available is the static utility method, `ConnectionSet.create()`:

```
ConnectionSet create(GraphModel m, Object[] cells, boolean disconnect)
```

This returns a new `ConnectionSet` instance based on the array of cells passed in

which contains edges and/ or ports. If `disconnect` is `true` the `ConnectionSet` returned describes those specified cells in a disconnected state. If `true`, it describes the edges connected accordingly to `model.getSource(cell)` and `model.getTarget(cell)` and ports according to the return value of `port.edges()`.

4.3 The GraphLayoutCache

The `GraphLayoutCache` holds the cell views, one for each cell in the model. It holds a list of cell view roots and another cached list of port views for performance reasons. `GraphLayoutCache` also holds a mapping from the cells to cell views, the only place in JGraph where you can translate in the model-to-view direction. `GraphLayoutCache` actually implements the `CellMapper` interface which defines methods to add and get mappings between cells and cell views. The `CellMapper` interface is not such an obvious design contract as the `GraphModel` is, but when obtaining the cell view for a cell, you must always use `getMapping`:

```
cellView = graph.getGraphLayoutCache().getMapping(cell);
```

The reverse mapping from graph view to graph model is not required since CellViews have references to their corresponding graph cells. Seeing the role the GraphLayoutCache plays in the mapping between the model and view domain, it may make more sense now why the GraphLayoutCache holds the reference to the CellViewFactory, the factory class that creates cell views depending on the cell type.

4.3.1 VIEW-LOCAL INDEPENDENCE

The `GraphLayoutCache` object provides the means to override information held in the graph model so that you may have multiple independent views of the same model. This enables features such as cell visibility, view-local attributes and expanding and collapsing. To set up a `GraphLayoutCache` in this way you need to set its `partial` attribute to `true`, this **must** be done in the `GraphLayoutCache` constructor:

```
GraphLayoutCache(GraphModel model, CellViewFactory factory, boolean
partial)
```

To change the partial status of a `GraphLayoutCache` during its lifetime would cause serious synchronization issues and so a `setPartial()` method is not made available.

Once a `GraphLayoutCache` has been made `partial` there is a difference in functionality between performing the 3 editing methods on the `GraphLayoutCache` and on the graph model. Performing them on the `GraphLayoutCache` will always update the view you are working in. Performing them on the graph model will make the changes to the model, but not reflect those changes in any `partial GraphLayoutCache`. So if you insert cells directly into the model, they will not appear in views where the `GraphLayoutCache` is `partial`. This is the recommend technique for inserting invisible cells.

The reason for the naming of the `partial` attribute is to indicate that the

GraphLayoutCache is a partial representation of what lies in the model, although the boundary case is that the contents are the same and it is the whole representation.

4.3.2 VISIBILITY

With a partial GraphLayoutCache, you are able to set any individual cell to being invisible using:

```
graph.getGraphLayoutCache().setVisible(cell, false);
```

which will perform the edit and appropriate updates for you. You can also define a set of cells to be made visible and another set of cells to be made invisible in one call using:

```
setVisible(Object[] visible, Object[] invisible)
```

A cell being set to be invisible simply means it is not drawn in that view, the model remains unchanged, only the GraphLayoutCache holds additional visibility information when partial.

4.3.2.1 Configuring Visibility after Editing Operations

There are a number of configuration options for editing operations that automatically deal with visibility issues for cells that have some relationship in the graph model. For example, if a vertex is made invisible it usually does not make sense to leave edges connected to that vertex visible. The hidesExistingConnections variable set to true ensures this happens and true is its default value.

For the reverse operation, showsExistingConnections determines whether or not edges that have both vertices connected to it made visible are made visible themselves. The default is, again, true.

showsChangedConnections determines whether or not edges should be made visible when they are reconnected to different vertices which are both visible, the default is true.

showsInsertedConnections determines whether or not inserted edges should me made visible if either their source or target are already visible, the default value is true.

Finally, hidesDanglingConnections determines whether or not edges should be made invisible when either connected vertex is removed from the model. The default for this value is false

4.3.3 VIEW-LOCAL ATTRIBUTES

Visibility is one of the important view-independent features in JGraph. Another is view-local attributes. View-local attributes enable you to have any of the attribute types available (in GraphConstants, or any extra attributes you might define) store a local value in the cell view storage attribute map and have that value override the value stored in the storage attribute map of the corresponding graph model cell. There are two variables in the GraphLayoutCache that support this functionality, allAttributesLocal and

`localAttributes`.

`allAttributesLocal` is a `boolean` flag that determines whether or not to make **all** attributes view-local, so the all attributes set in the `GraphLayoutCache` are stored locally in the cell views and those are the attributes used for the visualization. `localAttributes` is a `Set` of attribute keys (e.g. `GraphConstants.BOUNDS`, `GraphConstants.FONT`, etc.) that use the value in the cell view attribute map over that in the graph model cell. You can set all attributes to view-local using:

```
setAllAttributesLocal(true);
```

and set the value of localAttributes using:

```
setLocalAttributes(attributeSet);
```

Note the setting of the local attribute set overwrite the current set, it does not add to it. Therefore, if you wish to add to it, call `getLocalAttributes()` and add to the set obtained in-place.

Note, if you wish to remove a view-local attribute this requires more than simply removing the key from the local attributes set. The attribute value should also be removed from all cell view that have that attribute set. Depending on application requirements, you will either leave the attributes deleted or re-add them to the equivalent graph model cells' attribute maps. From JGraph 5.6.3 onwards the method `removeLocalAttribute(Object attribute, boolean addToModel)` is available in the `GraphLayoutCache` to assist this process. The `attribute` is the key to be removed and the flag indicates whether or not to re-add the deleted attribute to the model cells.

As previously mentioned, if you perform an insert call to the model with a partial layout cache, the cell is invisible to start with in the layout cache. If you perform an edit on the model and you change an attribute which is view-local in a graph layout cache, the value does not get passed to the cell views' attribute maps. Similarly, if you perform an edit directly on a graph layout cache any view-local attributes are not passed onto the model cells. This means you can have colors, cell positions and size, text font, any of the attributes in `GraphConstants` display differently in one view to another by using partial layout caches, setting the appropriate attributes to be view-local and editing those attributes using the edit call on the partial layout cache.

In the examples directory of the package you received with this user manual you will find the file `org.jgraph.example.GraphEdMV.java`. This is an example implementation of a simple multi-view application. The following code in the constructor of `GraphEdMV` sets the view-local attributes:

```
Set localAttributes = new HashSet();
localAttributes.add(GraphConstants.BOUNDS);
localAttributes.add(GraphConstants.POINTS);
localAttributes.add(GraphConstants.LABELPOSITION);
localAttributes.add(GraphConstants.ROUTING);
graph.getGraphLayoutCache().setLocalAttributes(localAttributes);
```

setting the cell positions and sizes, the edge points and routing and the label positions to be view independent.

4.3.4 EXPANDING AND COLLAPSING GROUPS

JGraph supports the expansion and collapsing of grouped cells. Obviously, in your own application you don't want to ask users to perform the grouping operation, so you will generally have some means of determining which cells the user is referring to in a collapse operation and perform the grouping and collapsing in one operation.

The GraphEdX example demonstrates the manual grouping and expanding and collapsing of cells. The figure below shows a selection of cells being grouped, collapsed and expanded again. The GraphEdX source code can be found in the examples directory of your User Manual or JGraph installation. The demo renderers a small “-” or “+” in the top left corner of the group cell to indicate that the group affords being collapsed and expanded. Mouse press events on that corner need to be captured (see chapter 5, Events) to trigger the calls to expand and collapse.

The actual call that causes cells to collapse is:

```
graph.getGraphLayoutCache().collapse(graph.getSelectionCells());
```

and to expand:

```
graph.getGraphLayoutCache().expand(graph.getSelectionCells());
```

The cell(s) passed in as the single parameter is the group cell. You might notice that the edge from cell 11 into the group terminates on the perimeter on the group when it is collapsed. This behavior is standard for visual collapsing and expanding. When cells are invisible it is checked to see if they have a visible parent, direct or indirect. If so, any edges connected to the invisible cell are promoted, *in the view only*, to terminate at the perimeter point of the first visible parent cell. There are no model changes involved in this process.

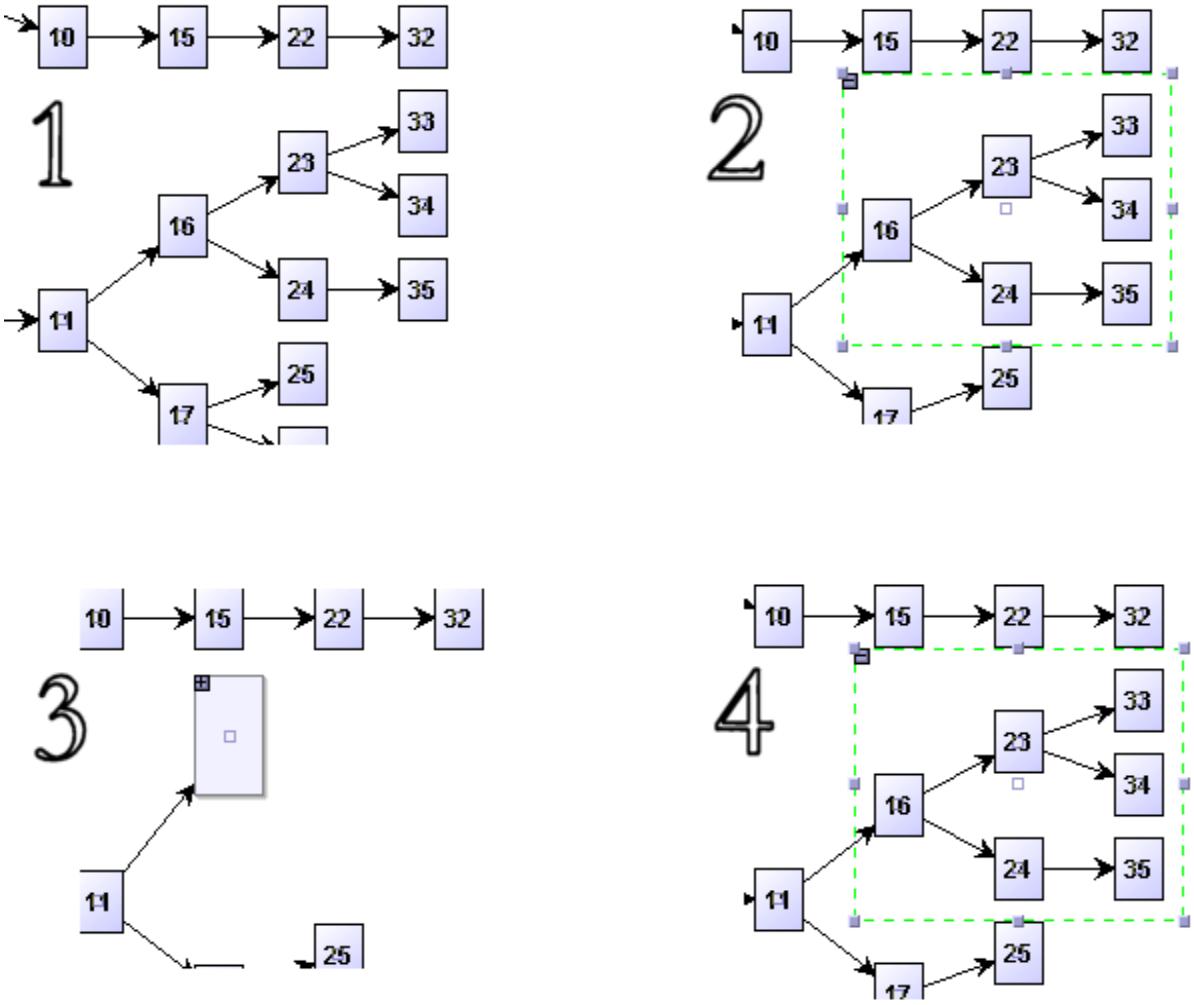

Illustration 47 : A selection of cells being grouped (2), collapsed (3) and expanded again (4)

For example, in the case of a user application involving a tree structure that can expand and collapse, you might prefer to render the "-" on the base of all cells. The user clicking on that symbol would cause the application to find all cells below that tree node, group them and the node itself, then collapse the group, all in one operation.

4.3.5 OTHER GRAPHLAYOUTCACHE OPTIONS

The `GraphLayoutCache` has a few more visual configuration options:

- `autoSizeOnValueChange` – when set to `true` all vertices are resized to their preferred size plus any inset value in the cell view attribute map every time their label text changes. This function might be seen as a global override of the per-cell autosize function. The important difference between this function and the per-cell autosize attribute is `autoSizeOnValueChange` still allows you to resize cells manually.
- `selectsAllInsertedCells` – determines whether or not inserted cells are selected. The default value is `true`.
- `selectsLocalInsertedCells` – determines whether or not local inserted cells, that is, cells inserted to a partial graph layout cache, are selected. The default value is `true`.

4.4 Advanced Model Functions

4.4.1 MODEL ORDERING

The graph model has an order to its cells defined by the order of the roots collection. Child cells are also deterministically ordered when accessing them from the parent and so the entire model has an order. This is important when performing analysis on the graph model, or layouts, since this ordering means the results can be relied upon to be deterministic. The ordering in the model also is used for **layering** the cells.

Layering relates to the way in which any cell can overlap any other and there needs to be some method to determine which cells lie in front of which. The rule is that the cell at the start of `roots` lies upon the back-most layer and each sequential root cell lies upon the next layer up until you reach the last entry in `roots` which lies on the topmost layer. If you perform an insert operation adding two cells the order the cells are inserted in is the same as the ordering in the cell array passed into `insert()`. The first cell will be the first entry into roots and lie behind the second in the layering structure.

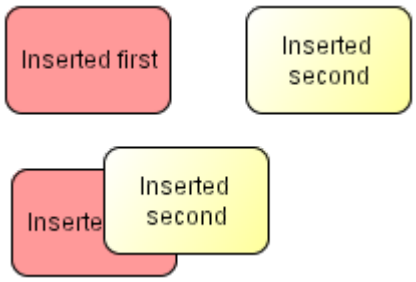

Illustration 48 : The layering resulting from the insertion of two cells

Regardless of how you drag the cells, the cell inserted second will remain over the first when they overlap. Since child cells of groups lie entirely within the bounds of the group cell, the whole group has the layer position of the root cell. Within the group each level of child cells are ordered and, again, the first entry of any level lies on the back-most layer within that group. This pattern continues to an arbitrary level of nesting.

Rather than provide ultra-fine grained positioning of cell layers it is more effective to simply be able to move a specified set of cells to the back-most layer:

```
toBack(Object[] cells)
```

or to the foremost layer:

```
toFront(Object[] cells)
```

these methods exist in both the `GraphLayoutCache` class and the `GraphModel` interface. Note that a number of cells may be affected and cells cannot share the same layer. Therefore, the operations move the specified cells to start or end of the level of the graph

structure they exist upon but retain the same relative order between those cells.

4.4.2 EDITS

When you perform an insert, edit or remove call, an object called an edit is created. In the case of calls to the `GraphModel` a `GraphModelEdit` object is created and for calls to the `GraphLayoutCache` a `GraphLayoutCacheEdit` is created in addition to the `GraphModelEdit`. These edit objects encapsulate the change made, holding information about the attribute map changes made using the nested attribute parameter, changes to the group structure made using the parent map parameter and changes to the connection states using the connection set parameter.

Some of the simplified edit calls in `GraphLayoutCache` do not offer all of these parameters, but values for them are created internally and held in the edit object as necessary. The edit object completely describes the change from the current state of the graph to the next state and in reverse and so is used to perform undo and redo functions. In fact, editing methods are performed by creating the edit object and executing it, exactly the same as a redo command functions.

4.4.2.1 Undo/Redo

Undo-support, that is, the storage of the changes that were executed so far, is typically implemented on the application level. This means, JGraph itself does not provide a running history, it only provides the classes and methods to support it on the application level. This is because the history requires memory space, depending on how many steps it stores (which is an application parameter). Also, history is not always implemented as some applications do not require it.

The `GraphChange` object is sent to the `UndoableEditListeners` that have been registered with the model. The object therefore implements the `GraphChange` interface and the `UndoableEdit` interface. The latter is used to implement undo-support, as it provides an undo and a redo method. (The code to execute, undo and redo the change is stored within the object, and travels along to the listeners.)

4.4.2.1.1 Undo-support Relay

Aside from the model, the graph view also uses the code that the model provides to notify its undo listeners of an undoable change. This can be done because each view typically has a reference to the model, whereas the model does not have references to its views. (The `GraphModel` interface allows relaying `UndoableEdits` by use of the fourth argument to the edit method.)

The `GraphLayoutCache` class uses the model's undo-support to pass the `UndoableEdits` that it creates to the `UndoableEditListeners` that have been registered with the model. Again, the objects that travel to the listeners contain the code to execute the change on the view, and also the code to undo and redo the given change.

This has the advantage that the `GraphUndoManager` must only be attached to the model, instead of the model and each view.

4.4.2.1.2 GraphUndoManager

Separate geometries, which are stored independently of the model, lead to changes that are possibly only visible in one view (view-only), not affecting the other views, or the model. The other views are unaware of the change, and if one of them calls undo, this has to be taken into account.

An extension of Swing's `UndoManager` in the form of `GraphUndoManager` is available to undo or redo such changes in the context of multiple views. `GraphUndoManager` adds the `undo` and `redo` methods with an additional argument, which allows specifying the calling view as a context for the undo/redo operation. The basic code to create and setup a graph undo manager is:

```
undoManager = new GraphUndoManager();
// Register UndoManager with the Model
graph.getModel().addUndoableEditListener(undoManager);
```

The parameter that is passed to the `GraphUndoManager's undo` and `redo` method is used to determine the last or next relevant transaction with respect to the calling view. Relevant in this context means visible, that is, all transactions that are not visible in the calling view are undone or redone implicitly, until the next or last visible transaction is found for the specified parameter.

As an example consider the following situation: Two views share the same model and both have at least one view-local attribute. This means, each view can change independently, and if the model changes, both views are updated. The model notifies its views if cells are added or removed, or if the group structure or connectivity of the graph is modified, meaning that either the source or target port of one or more edges have changed.

If the view-local attributes are only the points or bounds and if cells are moved, resized, or if points are added, modified or removed for an edge, then these changes are view-only transactions. All views but the source view are unaware of such view-only transactions, because such transactions are only visible in the source view.

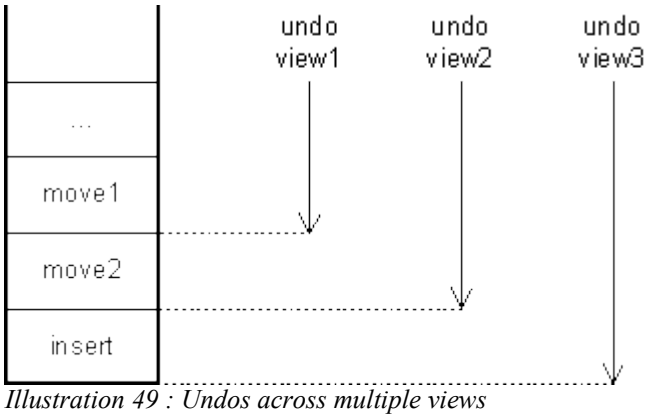

Illustration 49 : Undos across multiple views

In the above figure, the state of the command history is shown after a cell insertion into the model, move by the second view, and subsequent move by the first view. After insertion of the cell into the model, the cell's position is the same in all views, namely the position that was passed to the `insert` call. The arrows illustrate the edits to be undone by the `GraphUndoManager` for the respective views. In the case of view 3, which only sees the insert, all edits are undone.

As mentioned above, even if there are possibly many sources which notify the `GraphUndoManager`, the implementation of this undo-support exists only once, namely in the graph's model. Thus, the `GraphUndoManager` must only be added to one global entry point, which is the `GraphModel` object.

4.5 Drag and Drop

Drag and drop refers to the action of a user in a GUI of selection a visual object, usually by clicking on the object and moving the mouse while holding the mouse button down. The "dropping" part is where the mouse button is released. In AWT and Swing this means selecting a visual element in one component and dropping it in another. It you have not used DnD before, it is worth reading about the standard Swing mechanism at http://java.sun.com/docs/books/tutorial/uiswing/misc/dnd.html as JGraph is mostly compliant with the standard mechanisms.

JGraph supports drag and drop in the same way most Swing and AWT components do, dragging and dropping between JGraph instances in supported in the core library. The methods `setDropEnabled()` and `setDragEnabled()` on the `JGraph` object control whether these functionalities are available. As with most components, by default, drop is enabled and drag disabled after creating a `JGraph`.

Information - From Java 1.4 onwards a high-level event listener called `TransferHandler` was introduced to simply drag and drop. This is the only Java 1.4 specific feature in JGraph and the feature the build system swaps out when building for Java 1.3. The Java 1.3 drag and drop framework was somewhat more complex to use and will not be described in this manual.

There are two important interfaces defined in Swing relating to drag and drop, `Transferable` and `TransferHandler`. `Transferable` implementations describe the actual object(s) being transferred. Within a `Transferable` implementation (it is an interface) are referenced a number of `DataFlavor` instances which describe the format that a `Transferable's` data might take. Example flavors are `stringFlavor`, `imageFlavor` and `javaFileListFlavor`.

In the case of JGraph `org.jgraph.graph.GraphTransferable` defines a description of a graph transfer. It holds the cells being transferred, a `ConnectionSet` of connections between the cells and a `ParentMap` describing the group structure. In addition there is a nested `AttributeMap` with the cells' attributes and the bounds of the collective cells as a rectangle. This information is enough to recreate the graph when dropped (usually

in a JGraph component).

The other important element in drag and drop is the TransferHandler class. This class handles the creation of Transferables, via the createTransferable() method, and deals with their interpretation when dropped. When a TransferHandler receives a Transferable object it uses its canImport() method to determine whether or not it is capable of accepting the DataFlavor being offered. The importData() method deals with the actual process of accepting the drop and, in case of a JGraph component, editing the graph appropriately. If you wish the change the default drop behaviour it is an overridden importData() method where you would do this.

JGraph has to subclass TransferHandler because of a non-Swing standard feature it possesses. org.jgraph.graph.GraphTransferHandler is the default handler for exporting and importing a graph. The reason this is necessary is that standard TransferHandler transfers bean properties and the graph's selection cannot be implemented as a bean property. GraphTransferHandler understands drop from other JGraphs, but not from other Swing components by default. A common question is how to accept a drop in a JGraph from a component other than a JGraph. There are two possible ways of doing this.

The first is to make your graph understand other imported data flavors. To do this you need to create a sub class of GraphTransferHandler and override canImport() and importData() to confirm that the object can be imported and then to properly handle the import. importData() in the core GraphTransferHandler will give you a reasonable idea how to perform JGraph operations given a particular import.

The second method to adapt the TransferHandler of the exporting component, specifically the createTransferable() method, so that it creates a GraphTransferable that JGraph understands by default. This mechanism is only useful if the only thing its data will be exported to is a JGraph, since no other component understand how to import a graph.

To set a new TransferHandler on a graph call:

```
graph.setTransferHandler (new MyGraphTransferHandler());
```

The Clipboard in Java needs a small mention, when you perform cut, copy or paste operations drag and drop is performed via a clipboard. In Java there is the system (operating system) clipboard and any other instances of clipboard you create within your application. These store Transferable objects during a data transfer. If you use the shared system clipboard your data will be transferred to the native operating system clipboard, so you could transfer data to it and still have it available in another JVM session. If this is not the behaviour you require then create a Clipboard instance only for your application.

One issue at occasionally causes confusion is when developers try to write functionality that accepts drops onto heavyweight cells on the graph. The problem is to do with the use of the flyweight pattern for renderering (i.e. there is only one renderer component shared between all similar cell types). When you drop onto a heavyweight component, it doesn't really have a component instance (except when editing because an editor component instance is active at the time), i.e. you're dropping onto something painted on the JGraph component. So your transfer handler on the JGraph needs to handle this drop, there is no

cell component there to handle it. You can use the `getNextViewAt()` method on `JGraph` to determine where the drop has occurred. A sensible next step would be to pass the drop event to the component by calling `getRendererComponent()` on that view to install the actual component for that heavyweight and hand it the drop event to handle.

4.6 Zooming

Within the `JGraph` class the current scale of the graph is stored and may be altered through the `setScale()` method. The scale is stored as a double type and a value of 1.0 is the default, unscaled condition. Values above 1.0 refer to a scaling up of the graph (zooming in) and values below 1.0 indicate the graph is zoomed out. Setting the scale to 2.0 corresponds to x2 magnification, 4.0 to x4 magnification, 0.5 to x0.5 magnification (x2 reduction), and so on.

The `setScale(double)` method zooms leaving the center point on the screen unchanged. If the point around which the scaling is to take place is not the center point, use the `setScale(double, Point2D)` method where the point is the new center of the graph. This is useful, for example, when zooming to a particular mouse click point or specified marquee area.

4.7 Summary

- Grouping is part of the graph model structure and is represented through the parent/child relationships between cells.
- The editing methods can use parent maps and connection sets to describe a new state of grouping structure and connection states.
- The GraphLayoutCache can be made partial, meaning that some or any of the cell views in it can be made invisible. This technique is used for expanding and collapsing cells.
- Cell views may have view-local attributes, which override those in the corresponding graph model cell. Which attributes are view-local is defined in the GraphLayoutCache.
- Undo and redo is built into the editing methods and follows the Swing standard. Some extra functionality is required when dealing with undos/redos in multiple, independent views.
- When a model or layout cache change occurs, it is possible to have a listener detect this change and obtain a change object to examine the details of that change.
- Mouse events passed into MouseHandler by default and from there are passed onto a specific handler or handlers for context-specific processing.

5 Events

5.1 Graph Change Events and Listeners

The `GraphModel` defines methods for the handling and registration of two types of listeners, `UndoableEditListeners` and `GraphModelListeners`. Every notification of the undo listener is accompanied by a notification of the model listener, since the view needs to be updated and the display repainted. However, model events do not trigger undo events for obvious reasons.

If you wish to have certain functionality triggered upon the firing on a model event, you must implement the `GraphModelListener` interface which specifies one method, `graphChanged(GraphModelEvent e)`. The listener needs to be registered with the model in order to receive those events:

```
graph.getModel().addGraphModelListener(graphModelListener)
```

Once a change occurs you will be able to determine the details of the change by interrogating the event.

Both the graph model and the graph layout cache support this event model. The graph layout cache event only contains information specific to that view, i.e. changes to view-local attribute and local visibility changes. If the graph layout cache is not partial or no view-local attributes or states change during an edit, the graph layout cache event will be empty and only the graph model event contain any information. In this way graph layout cache events can be used to determine what happened only in a view. The complete picture of what changed during an edit can, therefore, be determined by examining both the graph model event and the graph layout cache event. If your graph layout cache is not partial and has no view-local attributes then only examining the graph model event will suffice.

`GraphModelEvent` and `GraphLayoutCacheEvent` are both found in the `org.jgraph.event` package. Within these classes are defined the `GraphModelChange` and `GraphLayoutCacheChange` respectively and these inner change object can be obtained using the `getChange()` method on the event interfaces. These changes are constructed for insertion, removal or modifications of cells in the model. Note that these objects contain both the description and execution of the change in one place.

The necessary getter methods to extract information out of the `GraphModelChange` are `getConnectionSet`, `getPreviousConnectionSet`, `getParentMap` and `getPreviousParentMap`. The `GraphLayoutCacheChange` interface defines `getChanged`, `getInserted`, `getRemoved`, `getAttributes` and `getPreviousAttributes`. `GraphModelChange` extends `GraphLayoutCacheChange` and so also defines the methods in the second list.

From the naming of the methods you will fairly easily be able to deduce how to access the pre-edit and post-edit versions of the object that store edit state, the parent map, the connection set and the nested attribute map. `getInserted` returns those cells that were inserted in the edit, `getRemoved` those that were removed in the edit and `getChanged` returns those cells that existed both before and after the edit, but whose attributes changed in

that edit.

Keep in mind that after you a perform an undo, the previous and current attribute in the edit are swopped around. This is so that the redo function works correctly.

5.2 The GraphUI and handling mouse input

The `org.jgraph.plaf.GraphUI` interface provides the UI-delegate interface for JGraph and inherits from `ComponentUI`. The default implementation, `BasicGraphUI`, provides all the usual methods you expect to paint, update and return component sizes.

However, the most common area of difficultly users get into with JGraph is working out where mouse event enter JGraph and how they are passed between the various mouse handling functions.

`BasicGraphUI` defines the method `createMouseListener()` which installs a mouse handler into the graph UI. If you subclass `BasicGraphUI` and create your own mouse handler, remember to override `createMouseListener()` to create your mouse handler. The same idea applies to any other custom functionality you add to your subclass, to call the `createXXX()` methods available in `BasicGraphUI`.

Have a look at the `MouseHandler` inner class in `BasicGraphUI`. This is where all mouse input events come into JGraph by default. It provides `mousePressed`, `mouseDragged`, `mouseMoved`, `mouseReleased`, as you might expect.

In mousePressed the first thing that happens is handler is set to null. This is the handler that is going to deal with the mouse event. MouseHandler goes through a series of checks to work out what was under the mouse when it was pressed.

Slightly lower down you will find this line:

```
int s = graph.getTolerance();
```

5.2.1 MOUSE TOLERANCE

When a user tries to select a cell, JGraph provides some assistance using the `tolerance` variable in `JGraph`. When a mouse press occurs the default mouse handler creates a rectangle around the point where the mouse event happened. The distance from the center of this rectangle to any side is the value returned from `getTolerance()`. It is within this rectangle that JGraph will process available cells. If you find that you have overlapping cells and the wrong cell is being processed due to the tolerance value, simply set it to 0.

The line after the `getTolerance` call reads:

```
Rectangle2D r = graph.fromScreen(new Rectangle2D.Double(e.getX()- s,
                                    e.getY() - s, 2 * s, 2 * s));
```

5.2.2 ZOOMING

JGraph uses the `Graphics2D` class to implement its zoom. The framework is feature-aware, which means that it relies on the methods to scale a point or rectangle to screen or to model coordinates, which in turn are provided by the `JGraph` object. This way, the client code is independent of the actual zoom factor.

Because JGraph's zoom is implemented on top of the `Graphics2D` class, the painting on the graphics object uses non-scaled coordinates (the actual scaling is done by the graphics object itself). For this reason, JGraph always returns and expects non-scaled coordinates.

For example, when implementing a `MouseListener` to respond to mouse clicks, the event's point will have to be downscaled to model coordinates using the `fromScreen` method in order to find the correct cell through the `getFirstCellForLocation` method.

On the other hand, the original point is typically used in the component's context, for example to pop-up a menu under the mouse pointer. Make sure to clone the point that will be changed, because `fromScreen` modifies the argument in-place, i.e. without creating a clone of the object. To scale from the model to screen, for example to find the position of a vertex on the component, the `toScreen` method is used.

Continuing further in the source code to `BasicGraphUI.mousePressed` there is a call to `isForceMarqueeEvent`.

5.2.3 MARQUEEHANDLER

The marquee in JGraph is the rectangular selection (sometime called “rubber-band” selection) you get when you click an empty area of the JGraph and drag. The `BasicMarqueeHandler` class is used to implement this type of selection. From an architectural point of view, the marquee handler is a "high-level" listener that is called by low-level listeners, such as the mouse listener, which is installed in the UI-delegate.

With regard to its methods, the marquee handler is more similar to the cell handle, because the marquee handler deals with mouse events, and allows additional painting and overlaying of the marquee . (The marquee is a rectangle that constitutes the to-be selected region.)

`isForceMarqueeEvent` checks to see if whatever mechanism there is in the current marquee handler is enabled to force handling of the mouse event to be passed onto the marquee handler. In the case of `BasicMarqueeHandler` this is caused by pressing and holding the 'alt' key during the mouse operation.

5.2.4 HANDLES

We mentioned handles in Chapter 3, it is within the `BasicGraphUI` we actually direct mouse events to the handles. The `BasicGraphUI` stores the current `CellHandle` in

the `handle` variable. This is updated in the `updateHandle()` method which creates cell handles depending on the current selection state of the graph.
For moving operations the mouse event will be passed to `RootHandle`, which is another inner class of `BasicGraphUI`. For resizing operations on vertices the mouse event will be passed to `SizeHandle`, which is an inner class of `VertexView`. And for edge moving and resizing functions the mouse event will be passed to `EdgeHandle`, which is an inner class of `EdgeView`.

6 I/O and JGraph Applications

6.1 XML Persistence

Java 1.4 and later provides the `XMLEncoder` and `XMLDecoder` mechanisms to serialize the objects of your application in a standard manner. An example of what your encoding phase might look like is shown below:

```
XMLEncoder enc = new XMLEncoder(out);
enc.setExceptionListener(new ExceptionListener() {
     public void exceptionThrown(Exception e) {
          // Dealt with exception
     }
});
// Configure persistence delegates here
enc.writeObject(object);
enc.close();
```

Java uses the mechanism of persistence delegates to identify what data from certain classes needs to be serialized. Note that it is not necessary to persist the `JGraph` object using the `writeObject` method, most application need only persist their `GraphLayoutCache`. This contains all the graph model and view geometry information:

```
enc.writeObject(graphLayoutCache);
```

If you are not familiar with the use of XML encoding and how to use persistence delegates, it is worth reading the Sun article on Using XMLEncoder. Obviously, to write the correct persistence delegates for a custom application you need to understand the mechanism. The basic idea is that you create persistence delegates corresponding to class constructors that you wish to be called when the XML is decoded later on. The classes described by the delegates must not be private, nor must the constructors be. The class itself must not be an inner class, it needs to be static or exist in its own file. Also, the class member variables must follow the Bean properties design where `setXXX()` and `getXXX()` methods exist for each variable XXX that is to be persisted.

As a general guide below are shown the typical delegates that will enable you to persist a simple JGraph base application:

```
XMLEncoder encoder;
try {
     encoder = new XMLEncoder(outputStream);

     // Better debugging output, in case you need it
     encoder.setExceptionListener(new ExceptionListener() {
          public void exceptionThrown(Exception e) {
               e.printStackTrace();
          }
```

```
     });

     encoder.setPersistenceDelegate(DefaultGraphModel.class,
               new DefaultPersistenceDelegate(new String[] { "roots",
                         "attributes" }));
     encoder.setPersistenceDelegate(GraphLayoutCache.class,
               new DefaultPersistenceDelegate(new String[] { "model",
                         "factory", "cellViews", "hiddenCellViews",
                         "partial" }));
     encoder.setPersistenceDelegate(DefaultGraphCell.class,
               new DefaultPersistenceDelegate(
                         new String[] { "userObject" }));
     encoder.setPersistenceDelegate(DefaultEdge.class,
               new DefaultPersistenceDelegate(
                         new String[] { "userObject" }));
     encoder.setPersistenceDelegate(DefaultPort.class,
               new DefaultPersistenceDelegate(
                         new String[] { "userObject" }));
     encoder.setPersistenceDelegate(AbstractCellView.class,
               new DefaultPersistenceDelegate(new String[] { "cell",
                         "attributes" }));
     encoder.setPersistenceDelegate(
               DefaultEdge.DefaultRouting.class,
               new PersistenceDelegate() {
                    protected Expression instantiate(
                              Object oldInstance, Encoder out) {
                         return new Expression(oldInstance,
                                   GraphConstants.class,
                                   "getROUTING_SIMPLE", null);
                    }
               });
     encoder.setPersistenceDelegate(DefaultEdge.LoopRouting.class,
               new PersistenceDelegate() {
                    protected Expression instantiate(
                              Object oldInstance, Encoder out) {
                         return new Expression(oldInstance,
                                   GraphConstants.class,
                                   "getROUTING_DEFAULT", null);
                    }
               });
     encoder.setPersistenceDelegate(ArrayList.class, encoder
               .getPersistenceDelegate(List.class));
     encoder.writeObject(graph.getGraphLayoutCache());
     encoder.close();
} catch (Exception e) {
     JOptionPane.showMessageDialog(graph, e.getMessage(), "Error",
               JOptionPane.ERROR_MESSAGE);
}
```

It should be noted that an output created in this way can be somewhat verbose for even a small graph. An technique to reduce the file size is the use of the `getConnectionSet` method of the `DefaultGraphModel`. By using this method, the redundancy between the port's edge set and the edge's source and target field can be removed from files. To do this, the model's persistence delegate must be changed to fetch the connection set from the

respective method and pass it to the constructor at construction time:

```
model.addPersistenceDelegate(JGraphpadGraphModel.class,
  new DefaultPersistenceDelegate(new String[] { "roots",
    "attributes", "connectionSet" }));
```

To avoid storing the respective properties of the cells, they must be made transient (which is done in the static initializer in the preceding step):

```
JGraphEditorModel.makeTransient(DefaultPort.class, "edges");
JGraphEditorModel.makeTransient(DefaultEdge.class, "source");
JGraphEditorModel.makeTransient(DefaultEdge.class, "target");
```

The `makeTransient` method looks like this:

```
public static void makeTransient(Class clazz, String field) {
     try {
          BeanInfo info = Introspector.getBeanInfo(clazz);
          PropertyDescriptor[] propertyDescriptors = info
                                        .getPropertyDescriptors();
          for (int i = 0; i < propertyDescriptors.length; ++i) {
               PropertyDescriptor pd = propertyDescriptors[i];
               if (pd.getName().equals(field)) {
               pd.setValue("transient", Boolean.TRUE);
               }
          }
     } catch (IntrospectionException e) {
          // Dealt with exception
     }
}
```

To read the XML back into your application you will need code similar to that below. Remember that you object will be of the type that you wrote out in the encoding phase.

```
XMLDecoder dec = new XMLDecoder(in);
if (dec != null) {
     Object obj = dec.readObject();
     dec.close();
     return obj;
}
return null;
```

Note that the GraphEdX example that comes with all User Manual distributions demonstrates the functionality to load and save a graph using XML encoding.

6.2 Image Exporting

Using the various image processing functionality available in Java, it is relatively simple to produce an image of your graph in JPEG, bitmap (.bmp) or Portable Network Graphics

(.png) format. A utility method, `getImage()` is provided in the `JGraph` class to make exporting a simple task. `getImage()` takes two parameters, the first is the background color of the output image and the second is any inset to be use around every side of image produced.

The background color, you may wish to simply be the background color of the graph, but for the PNG output format there is the option of a transparent background. In the example below you need to use your own graph, your own output stream and select an appropriate background, but otherwise this code should work for all cases:

```
JGraph graph = getGraph(); // Replace with your own graph instance
OutputStream out = getOutputStream(); // Replace with your output stream
Color bg = null; // Use this to make the background transparent
bg = graph.getBackground(); // Use this to use the graph background
color
BufferedImage img = graph.getImage(bg, inset);
ImageIO.write(img, ext, out);
out.flush();
out.close();
```

6.3 SVG Export

There are two methods that may be used to export a `JGraph` to SVG format. The first is to use the Apache Batik library to perform the export, the second is to natively produce the SVG mark-up within your application. The second method is employed in SVG example you can find in the examples package of the JGraph.Layout product. Natively writing the SVG output provides large performance improvements over the Batik library. The Batik library produces output that only uses very primitive graphics elements and so post-processing of the SVG output is not possible since the graph context is not discernible from the output. The Batik library, at the time of writing, also is missing certain useful features, such as the association of a Hyperlink with a cell or text element.

The first method is the one currently most often used and the one that will be described here. The Batik library may be downloaded from its home page, which also provides a number of useful tutorials regarding the use of the library. The basic principle is to create a `SVGGraphics2D` object and paint the graph to that, the best explanation of how to do this is the code itself, shown below:

```
public static void writeSVG(JGraph graph, OutputStream out, int inset)
          throws UnsupportedEncodingException, SVGGraphics2DIOException
{
     Object[] cells = graph.getRoots();
     Rectangle2D bounds = graph.toScreen(graph.getCellBounds(cells));
     if (bounds != null) {
          // Constructs the svg generator used for painting the graph to
          DOMImplementation domImpl = GenericDOMImplementation
                    .getDOMImplementation();
          Document document = domImpl.createDocument(null, "svg", null);
```

```
            SVGGraphics2D svgGenerator = new SVGGraphics2D(document);
            svgGenerator.translate(-bounds.getX() + inset, -bounds.getY()
                        + inset);

            // Paints the graph to the svg generator with no double
            // buffering enabled to make sure we get a vector image.
            RepaintManager currentManager = RepaintManager
                        .currentManager(graph);
            currentManager.setDoubleBufferingEnabled(false);
            graph.paint(svgGenerator);

            // Writes the graph to the specified file as an SVG stream
            Writer writer = new OutputStreamWriter(out, "UTF-8");
            svgGenerator.stream(writer, false);

            currentManager.setDoubleBufferingEnabled(true);
      }
}
```

Viewing the output may be performed using the Squiggle browser produced by Apache, Internet Explorer with the Adobe SVG plug-in or Firefox 1.5 or greater. In the author's experience Internet Explorer with the Adobe plug-in produces the best quality output at the time of writing.

6.4 Exporting in a Headless Environment

On *nix systems the architecture of the X Windows system means that Swing requires some kind of graphics buffer to write to. When using a 1.3 version of the Java Virtual Machine (JVM) in order to produce exported images on such systems a framebuffer is required, the absence of a buffer to write to would cause a headless exception to be fired. Note that Windows systems do not have have this issue since they do not have the same client/server separation. With the popularity of *nix on the server-side, the common requirement of producing graph images on a server and then streaming those images to a client side browser could be non-trivial.

Previously, on *nix systems you would generally either set-up a VNC server or run a virtual framebuffer if there was no X Windows server available. Having to change the server environment was often not acceptable and so from JVM 1.4 the concept of a headless mode was introduced to work around this issue. By setting the `-Djava.awt.headless=true` option in the JVM arguments it is possible to create instances of lightweight components. Sun provide a useful tutorial explaining the use of headless mode in Java. Both the core JGraph library and JGraph.Layout are designed to work correctly in headless mode.

To display lightweight components it is necessary to add them to a heavyweight component such as a `Window` or a `Frame`, which cannot be used in a headless environment. Instead of creating a `Frame` and calling `pack()` there is a workaround where you may create a `JPanel` and call `addNotify()` to achieve the same effect. Although, `addNotify()` is not strictly meant to be called by developers, this is a widely accepted workaround:

```
        JGraph graph = getGraph();  // Replace with your graph instance
        JPanel panel = new JPanel();
        panel.setDoubleBuffered(false);  // Always turn double buffering
off when exporting
        panel.add( graph );
        panel.setVisible( true );
        panel.setEnabled( true );
        panel.addNotify();   // workaround to pack() on a JFrame
        panel.validate();
        Color bg = null; // Use this to make the background transparent
        bg = graph.getBackground(); // Use this to use the graph
background color
        BufferedImage img = graph.getImage ( bg, 0 );
```

Using the above workaround means that you can use JGraph the same way you would in a desktop Swing application. There is another method to use JGraph in that does not require the creation of the JGraph Swing component, this is described in the section below entitled "Working without the Swing component".

6.5 Working without the Swing component

The creation of the Swing component is not always required, for example when an application only creates a graph, applies a layout and finally extracts the position results through the API. In JGraph the `GraphLayoutCache` object must be created in order to obtain most of the available functionality, as well as an implementation of the `GraphModel` interface. The `JGraph` instance performs the task of adding the `GraphLayoutCache` as a listener to the `GraphModel`. Without a `JGraph` instance being created, it is recommended this is done by creating a subclass of `GraphLayoutCache` and making that subclass implement `GraphModelListener`. This subclass should add itself as a listener to the model and deal with graph changes appropriately.

In the examples folder of the JGraph.Layout produce there is an example named `com.jgraph.layout.JGraphHeadlessLayoutExample` which demonstrates a graph simple graph being laid out without a `JGraph` instance being created. The `GraphLayoutCache` subclass described above is implemented in the `com.jgraph.layout.DataGraphLayoutCache` class.

6.6 JGraph in an Applet

Java Applet applications can be created easily by extending the `JApplet` class and referencing the applet in the html of a page. Java recommends against static references to UI components for this reason, but the flyweight pattern is a central part of the design of JGraph and so JGraph requires certain static class references to be recreated if the applet is reloaded. The `org.jgraph.examples.GraphEd` example is itself an applet, refer to

the `destroy()` method of that class. This method resets the static references to UI components in JGraph. Note, if your application has additional such references, they should be recreated in the same way to avoid issues when applets reload.

6.7 Printing

Printing in built into Swing and with JDK 1.4 the `javax.print` package provides detailed control over the printing process. This package contains the `PrinterJob` class which is the main printing control class. The basic mechanism to print is implemented using the following code:

```
PrinterJob printJob = PrinterJob.getPrinterJob();
printJob.setPrintable(graphPane); // where graphPane is a JScrollPane
with a graph in it, for example
if (printJob.printDialog()) {
    printJob.print();
}
```

You require a Swing container that implements the `Printable` interface. This container needs to implement a `print()` method that is called when a print job is invoked. This is the only method the interface defines and it takes three parameters: `graphics` – the graphics context to paint the page on, `pageFormat` – a description of the size and orientation of the page and `pageIndex` – the index of the page to be drawn (starts from zero).

You use the standard Swing printing functionality to set which `Printable` element is to be printed and to start the print:

```
public int print(Graphics g, PageFormat printFormat, int page) {
    Dimension pSize = graph.getPreferredSize(); // graph is a JGraph
    int w = (int) (printFormat.getWidth() * pageScale);
    int h = (int) (printFormat.getHeight() * pageScale);
    int cols = (int) Math.max(Math.ceil((double) (pSize.width - 5)
            / (double) w), 1);
    int rows = (int) Math.max(Math.ceil((double) (pSize.height - 5)
            / (double) h), 1);
    if (page < cols * rows) {

        // Configures graph for printing
        RepaintManager currentManager =
RepaintManager.currentManager(this);
        currentManager.setDoubleBufferingEnabled(false);
        double oldScale = getGraph().getScale();
        getGraph().setScale(1 / pageScale);
        int dx = (int) ((page % cols) * printFormat.getWidth());
        int dy = (int) ((page % rows) * printFormat.getHeight());
        g.translate(-dx, -dy);
        g.setClip(dx, dy, (int) (dx + printFormat.getWidth()),
                (int) (dy + printFormat.getHeight()));

        // Prints the graph on the graphics.
```

```
            getGraph().paint(g);

            // Restores graph
            g.translate(dx, dy);
            graph.setScale(oldScale);

            currentManager.setDoubleBufferingEnabled(true);
            return PAGE_EXISTS;
        } else {
            return NO_SUCH_PAGE;
        }
}
```

7 Layouts

7.1 Introduction

7.1.1 WHAT DOES JGRAPH.LAYOUT DO?

JGraph.Layout takes graph structures defined using the JGraph library and performs either or both of two specific functions on that graph structure:

1. Position the vertices of that graph using an algorithm(s) that attempts to fulfil certain aesthetic requirements,
2. Add and remove control points of edges in the graph using an algorithm(s) that attempts to fulfil certain aesthetic requirements.

Exact what these aesthetic criteria are depend upon individual application or layouts requirements. Generally, these might involve spreading out vertices evenly without them overlapping each other, avoiding edges overlapping vertices and crossing other edges, clustering connected vertex neighbours and ordering vertices to reflect overall graph direction.

The standard facade in JGraph.Layout requires a JGraph instance in order to operate. The facade in JGraph.Layout extracts information from the GraphLayoutCache and graph model attached to this graph instances and stores it for processing by the layouts. The facade can then be passed to one or more layouts and store the compound result within forcing the result to be applied back to the graph.

JGraphModelFacade does not have any dependency on a JGraph object, instead the constructors take a GraphModel as a parameter. This means you are able to create a graph and apply a layout to it without having to instantiate a JGraph, ideal for server-side layouting.

Some confusion can arise as to whether a layout acts upon the GraphLayoutCache object (i.e. the view of the graph as the application displays it) or upon the filtered view produced by the JGraphFacade. The layout acts the graph as the facade describes it and this may be different to the view provided by the cache.

For example, the GraphLayoutCache may be set to not display edges when their connected vertices are not visible. However, the facade, through the edgePromotion flag, may promote those edges to the first visible parent. This means the layouts will act as though the edge is there, even though it is not drawn.

7.2 Running a layout

There are two important classes required for configuring and running a layout, `JGraphLayout` and `JGraphFacade`. Classes inheriting from `JGraphLayout` perform the mathematical operations of producing the layout, whereas, `JGraphFacade` performs filtering on the graph and provides various utility methods for the layout to extract

information about the graph. The advantage of this mechanism is that the exact data transferred to the layout is de-coupled from the layout algorithm itself, providing a more stable API during the lifetime of the package as new layouts are introduced. It also means that layout algorithm is able to use the output of any other layout as its input, i.e. the facade is manipulated by one by layout and then passed to another.

The first thing to be done when running a layout is to create the facade object that stores information about the graph to be acted upon and its configuration. The constructors require an instance of `JGraph` so the facade knows which graph is being referenced in the layout. If a tree layout is being used, the constructor must also be passed the root node(s) of the trees. `JGraphFacade` has a number of switches also that enable the layout to act upon the correct cells in the graph. By setting these switches, the facade configures what it returns from certain utility methods, again encapsulating the configuration of the layout in the facade. For example, by default the `getNeighbours()` method on the facade returns neighbour cells regardless of their visibility, whereas with the `ignoresHiddenCells` flag set to `true`, only cells visible in the current graph view will be returned. The layouts are designed to access information through such methods in the facade, performing stateful filtering. The switches on the facade are:

- `ignoresHiddenCells` - Stores whether or not the layout is to act on only visible cells i.e. `true` means only act on visible cells, `false` act on cells regardless of their visibility. The default value is `true`.
- `ignoresUnconnectedCells` - Stores whether or not the layout is to act on only cells that have at least one connected edge. `true` means only act on connected cells, `false` act on cells regardless of their connections. The default value is `true`.
- `ignoresCellsInGroups` - Stores whether or not the layout is to only act on root cells in the model. `true` means only act on root cells, `false` means act upon roots and their children. The default value is `false`.
- `directed` - Stores whether or not the graph is to be treated as a directed graph. `true` means follow edges in target to source direction, `false` means treat edges as directionless. The default value is `true`.

The facade object not only stores the input to the layout, but also the output. The result of a layout is not automatically applied to a graph in case the developer wishes to check the result or perform another algorithm. To enable this the result of the layout is stored as a nested map of the attributes where the graph cell is the key to each pair, and an attribute map, detailing the changes made to that cell by the layout, is the value. This map may be obtained by a call to `getNestedMap()` on the facade and is suitable for sending directly to the `edit()` method on the `GraphLayoutCache` or `GraphModel`. Below is a simple example showing the steps of setting the objects up, executing the layout and applying the layout back to the graph:

```
      JGraphFacade facade = new JGraphFacade(graph); // Pass the facade
the JGraph instance
      JGraphLayout layout = new JGraphFastOrganicLayout(); // Create an
instance of the appropriate layout
```

```
      layout.run(facade); // Run the layout on the facade. Note that
layouts do not implement the Runnable interface, to avoid confusion
      Map nested = facade.createNestedMap(true, true); // Obtain a map
of the resulting attribute changes from the facade
      graph.getGraphLayoutCache().edit(nested); // Apply the results to
the actual graph
```

The method to obtain a nested map of the results of the layout, `createNestedMap`, takes two parameters:

- `ignoreGrid` - whether or not the map returned is snapped to the current grid
- `flushOrigin` - whether or not the bounds of the resulting graph should be moved to (0,0)

7.2.1 WRITING YOUR OWN LAYOUT

Any new layout created should conform to the JGraphLayout interface. A new layout is complex to write, but mostly due to the algorithm of the layout, the process of interfacing with JGraph is simple.

The run() method of any layout must determine the required information from the facade as it currently exists, perform the layout and finally apply the results of the layout back to facade. Remember, the facade is a stateful filter of the graph. The reason for always using the facade and not the graph model or graph layout cache, is that many layouts might be applied in sequence and the output of the last layout should be the input of the next. Also, the facade flags are taken into account in the graph model or view.

One of the first things all layouts do is obtain the position and size of the vertices to be laid out. This is done using the getBounds() method on the facade. Layouts normally store a copy of the bounds values locally within the layout class. An array of vertices is passed into the getBounds() method, this is obtained using facade.getVertices().toArray().

As well as the positioning of vertices, the connections between those vertices will usually be required. getNeighbours() is often used to determine this, also getEdgesBetween(), getOutgoingEdges() and getIncomingEdges() are useful in this regard.

Finally, having applied the layout alogrithm, the position of the vertices after the layout must be available. These are then set back on the facade using setLocation(). If the layout does this correctly, calling in the manner described above will result in the layout being applied to the graph.

7.2.2 EDGE CONTROL POINTS

Some of the layout algorithms are designed specifically to manipulate and insert/remove edge control points in order to provide better edge routing in the end result. Because routing algorithms may be defined on a per edge basis, the layout algorithms only alters the control

points of edges is required by that algorithm. Therefore, if one algorithm changes an edge's control points and another layout is immediately applied then the control points will probably look incorrect in the new layout. Rather than try to second-guess whether or not inserted control point were added purposefully or accidentally it is left to the developer to deal with the state of control points prior to a layout being applied. The utility method `resetControlPoints()` on `JGraphFacade` is available to clear all control points should you require this to be done before any layout is run.

7.2.3 EXAMPLES

In the examples package of the JGraph.Layout product you will find a series of examples that demonstrates the layout features, as well as some additional features such as using an overview panel, exporting to SVG and implementing rich text label editiors. Note that the JGraphLayoutExample requires the use of the external L2FProd common library to run. This library is available under the Apache Software License. The JGraph team have used it for several years and found both the software to be of high quality and the lead developer to be very responsive to bug reports.

7.3 Using the layouts

7.3.1 THE TREE LAYOUTS

The tree layout classes currently available in the JGraph.Layout package are:

- `com.jgraph.layout.tree.JGraphTreeLayout`
- `com.jgraph.layout.tree.JGraphCompact TreeLayout`
- `com.jgraph.tree.JGraphRadialTreeLayout`.

Note that at least one root must be specified for all tree layouts using the `roots` parameter of the facade constructors. Note that these are the roots of the tree, not the roots of the graph model. Tree layouts will follow edges from the root node(s) to determine the structure of the tree(s), taking into account the settings of the facade.

Layout Pro also supports the concept of laying out sub-trees as show in the example application. Selection of any node and the execution of a tree layout will result in only the child tree nodes being laid out as a tree with the selected node as root. Note that the facade needs to be set to directed (the default value), otherwise the algorithm determining the tree structure will process the parents of the sub-node. However, this technique can be used to change the root node of an entire tree.

Here is how you might set up the facade to process a tree layout:

```
      Object roots = getRoots(); // replace getRoots with your own
Object array of the cell tree roots. NOTE: these are the root cell(s) of
the tree(s), not the roots of the graph model.
      JGraphFacade facade = new JGraphFacade(graph, roots); // Pass the
facade the JGraph instance
      JGraphLayout layout = new JGraphTreeLayout(); // Create an
instance of the appropriate layout
      layout.run(facade); // Run the layout on the facade.
      Map nested = facade.createNestedMap(true, true); // Obtain a map
of the resulting attribute changes from the facade
      graph.getGraphLayoutCache().edit(nested); // Apply the results to
the actual graph
```

7.3.1.1 Tree Layout

The tree layout arranges the nodes, starting from a specified node(s), into a tree-like structure. The tree may by oriented in the four cardinal compass points options on the layout include alignment of same-level nodes selection, setting the minimum distance between nodes on adjacent levels of the tree and setting the minimum distance between nodes on the same levels. The performance of the tree layout is O(|V|), i.e. proportional to number of nodes in the layout.

7.3.1.1.1 Alignment

Alignment refers to which part of vertices will be aligned for all vertices on a given level. Using the `setAlignment()` method you can set the alignment of the graph to `SwingConstants.TOP`, `SwingConstants.CENTER` or `SwingConstants.BOTTOM`. The literal values of these constants are 1, 0 and 3 respectively at the time of writing, but the variable names should always be used.

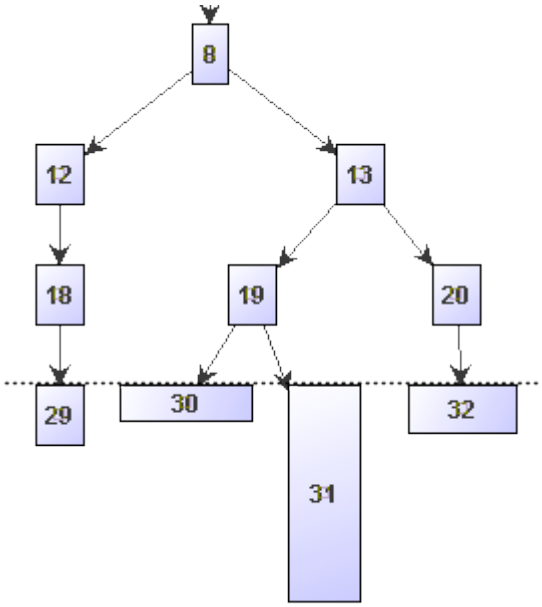

Illustration 50 : SwingConstants.TOP

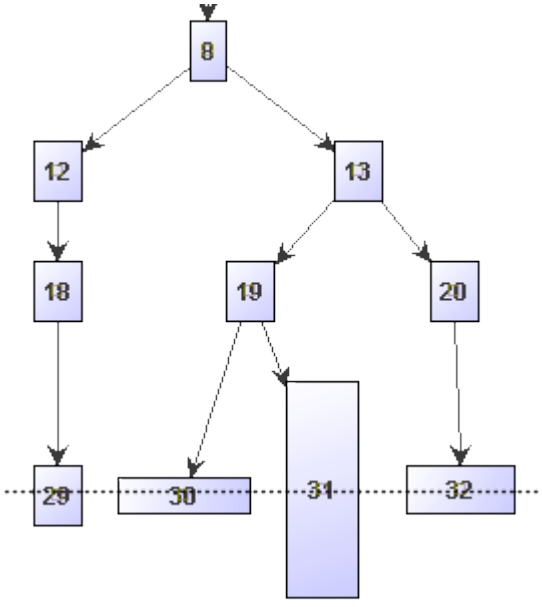

Illustration 51 : SwingConstants.CENTER

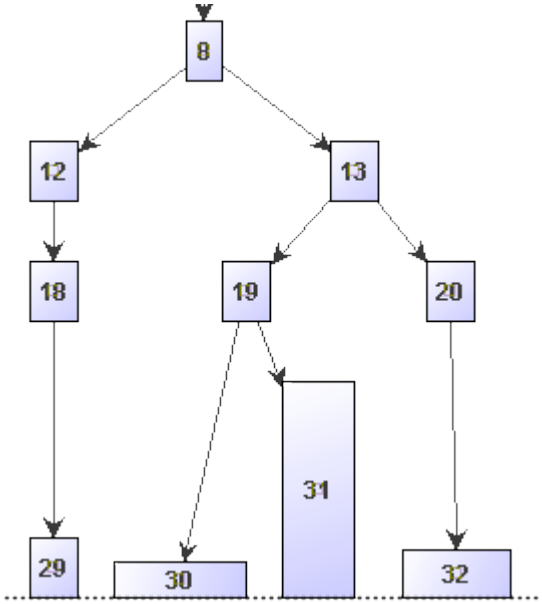

Illustration 52 : SwingConstants.BOTTOM

7.3.1.1.2 Orientation

Orientation refers to the compass direction in which the root node(s) of the tree will be located relative to the rest of the tree. Using the `setOrientation()` method you can set the orientation to `SwingConstants.NORTH`, `SwingConstants.EAST`, `SwingConstants.SOUTH` or `SwingConstants.WEST`. The literal values of these constants are 1, 3, 5 and 7 at the time of writing, but the variable names should always be used.

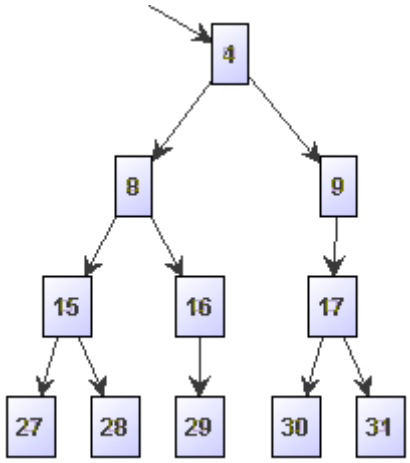

Illustration 53 : SwingConstants.NORTH

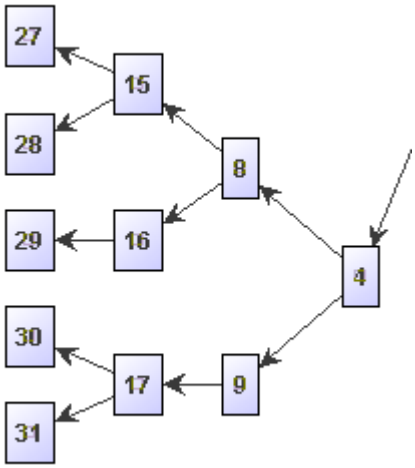

Illustration 54 : SwingConstants.EAST

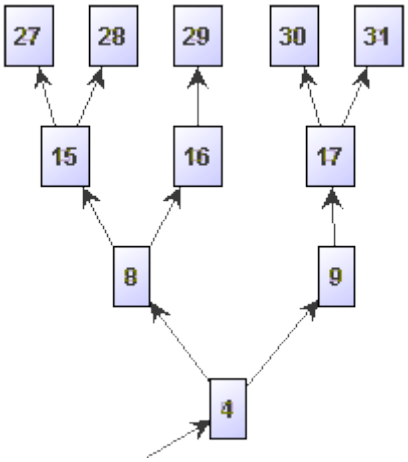

Illustration 55 : SwingConstants.SOUTH

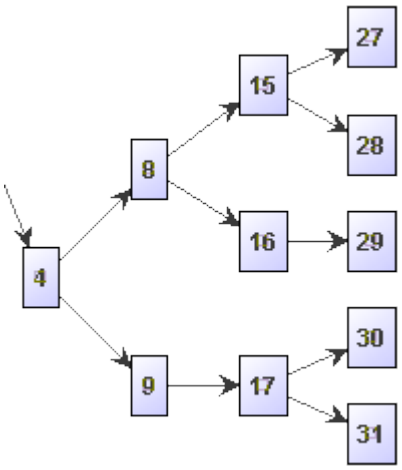

Illustration 56 : SwingConstants.WEST

7.3.1.1.3 levelDistance and nodeDistance

`levelDistance` is the distance between the lowest point of any vertex on one level to the highest point of any vertex on the next level down. `nodeDistance` is the minimum distance between any two vertices on the same level. Note that levels closer to the root tend to be spaced a further apart than this, assuming the density of nodes is lower towards the start of the tree.

.

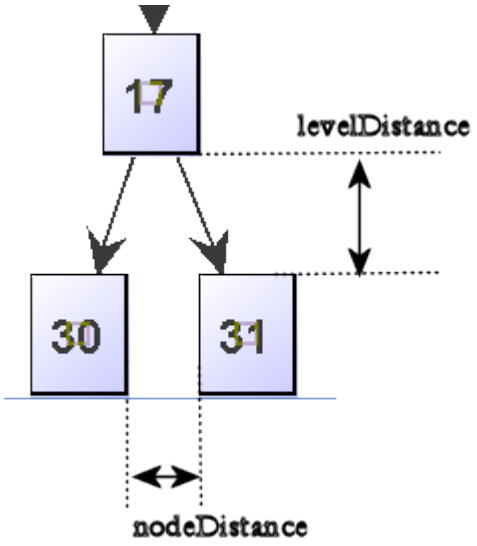

Illustration 57 : levelDistance and nodeDistance definitions

7.3.1.1.4 combineLevelNodes

The `combineLevelNodes` flag specifies whether or not to ensure that nodes on the same tree level are aligned across the entire tree. When nodes vary in size it is possible to save space on sub-trees with smaller nodes by setting this flag to `false`. However, this can make it difficult to determine visually which nodes occupy the same level on the tree. If this flag is set to `true`, the `alignment` variable determines exactly which part of nodes of the same level are aligned.

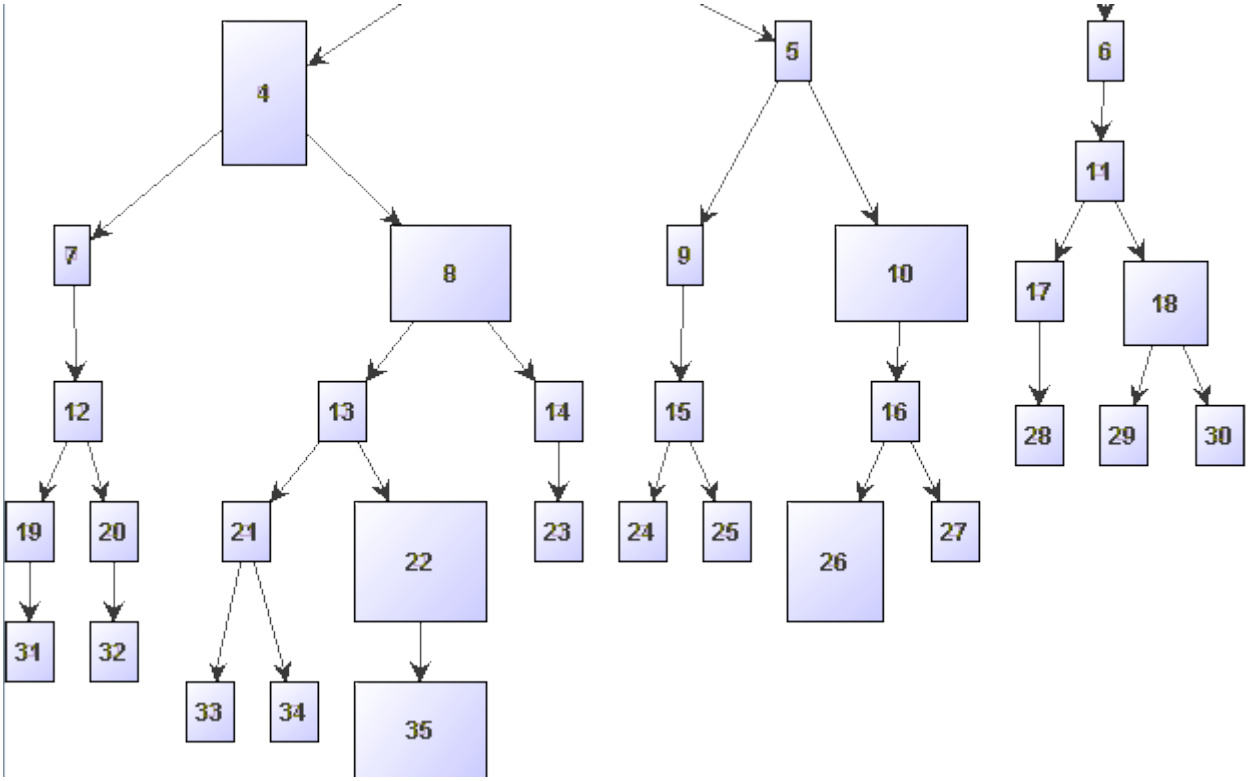

Illustration 58 : combineLevelNodes = false

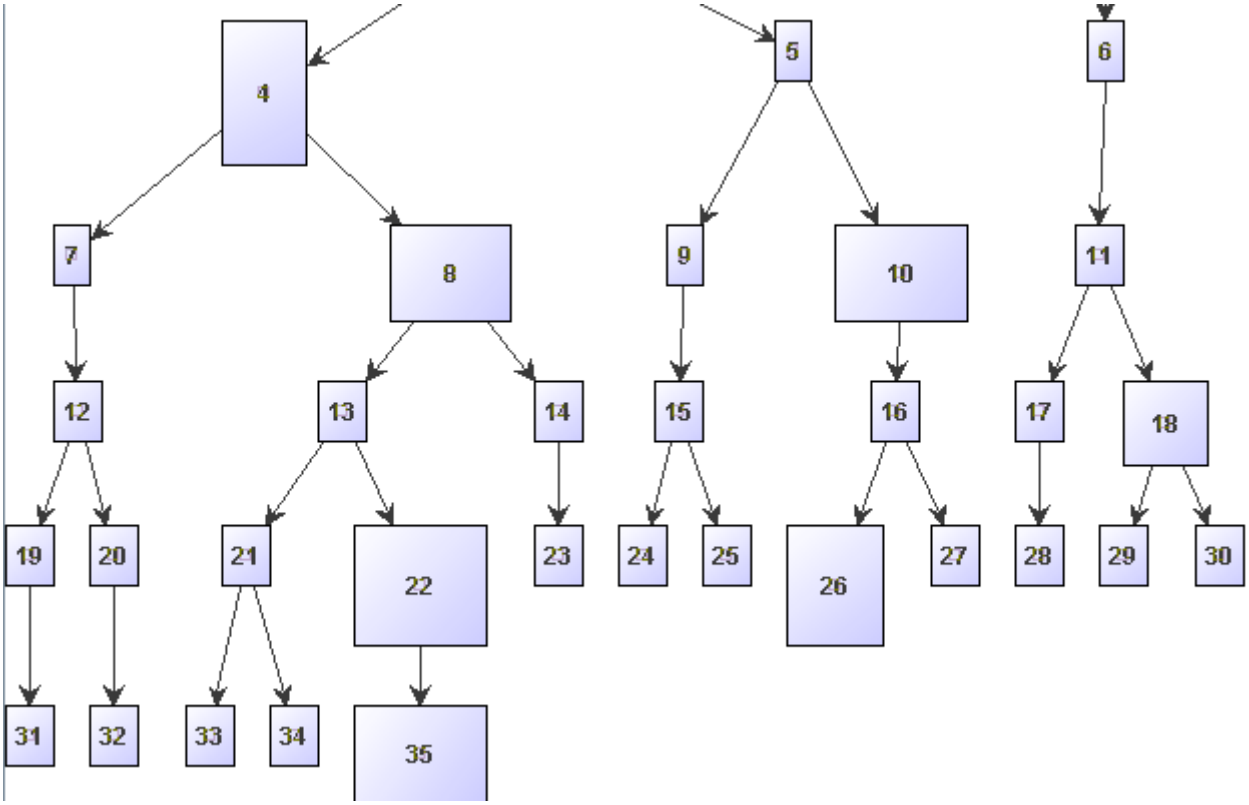

Illustration 59 : combineLevelNodes = true

7.3.1.1.5 positionMultipleTrees and treeDistance

`positionMultipleTrees` determines whether or not to separate distinct trees so there is no overlap between the trees. Each of the distinct trees to be separated would have to be specified in the `roots` parameter of `JGraphFacade`. The distance between each of the trees is defined by the `treeDistance` variable.

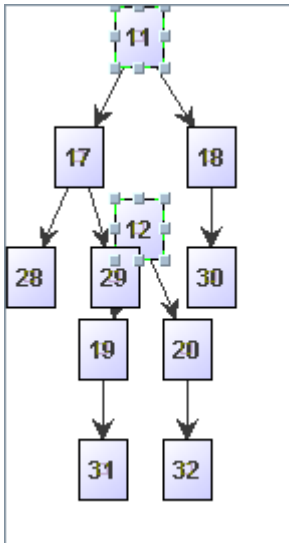

Illustration 60 : positionMultipleTrees = false

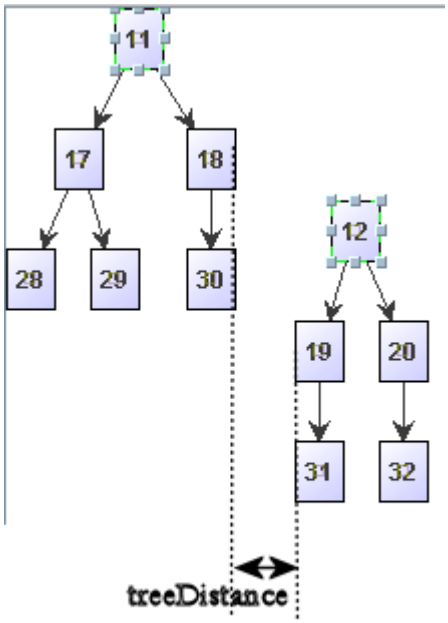

Illustration 61 : positionMultipleTrees = true , treeDistance = 30

7.3.1.2 Compact Tree Layout

The Compact Tree Layout (formerly called the Moen) is another layout in the tree-family, it makes some improvements over standard tree layouts. The Compact Tree takes cell shapes into account and concentrates on producing as compact a result as possible. The Compact Tree also describes mechanisms to compute deltas of the layout, so the entire computation does not have to be performed on every layout. The exact mechanism for how to do this depends upon the application. If you require this performance advantage, contact JGraph support for information on how to apply it in your application. The Compact Tree manages to compact more tightly than the standard tree by storing sub-trees as polygons. In terms of performance the time to lay out using the layout is O(|V|), i.e. proportion to the number of vertices.

7.3.1.3 Radial Tree Layout

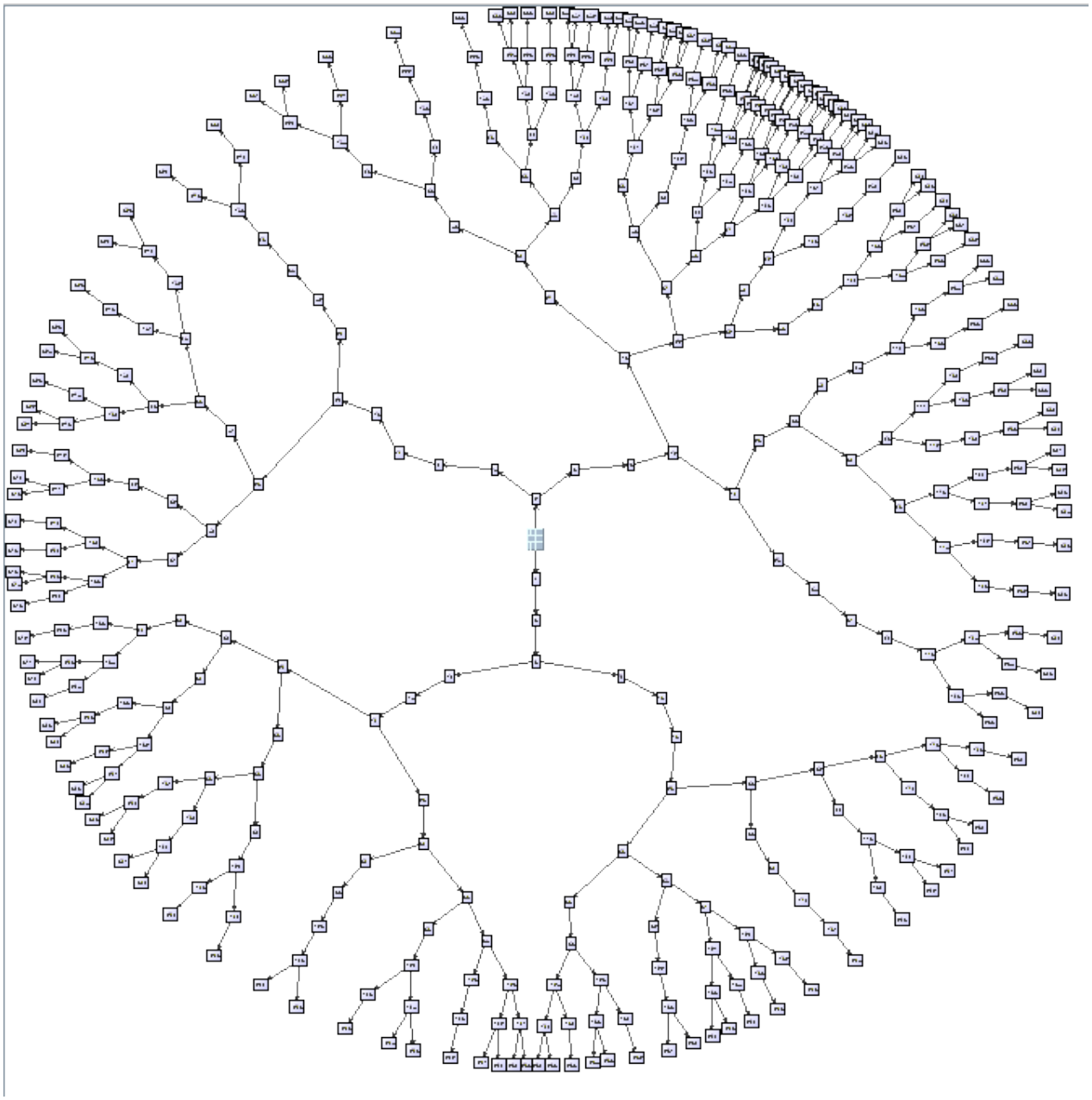

Illustration 62 : A Radial Tree Layout

The Radial Tree Layout draws the root node of the tree in the centre of the layout and lays out the other nodes in concentric rings around the focus node. Each node lies on the ring corresponding to its shortest network distance from the root node. Immediate neighbours of the root node lie on the smallest inner ring, their neighbours lie on the second smallest ring until the most distance nodes form the outermost rings. The angular position of a node on its ring is determined by the sector of the ring allocated to it. Each node is allocated a sector within the sector assigned to its parent, with size proportional to the angular width of that node's subtree. The performance of the radial tree is O(|V|), i.e. proportion to the number of vertices.

7.3.2 ORGANIC LAYOUTS

7.3.2.1 Spring Embedded

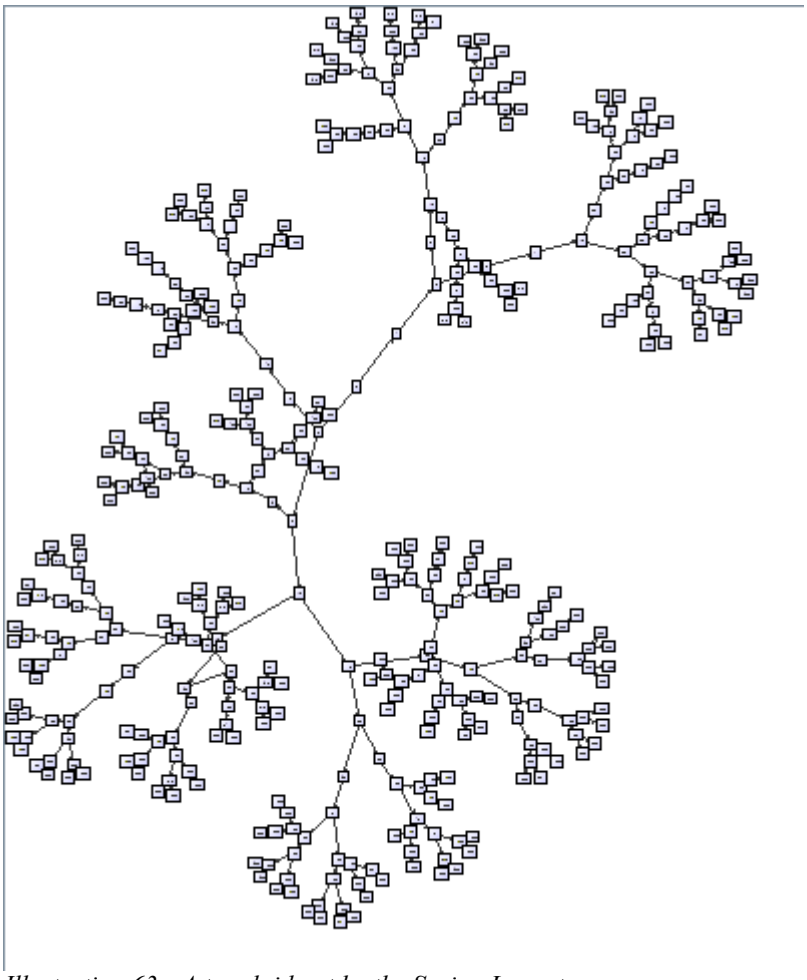

Illustration 63 : A tree laid out by the Spring Layout

The Spring Layout is a force-directed layout algorithm designed to simulate a system of particles each with some mass. The vertices simulate mass points repelling each other and the edges simulate springs with attracting forces. The algorithm moves through a number of iteration trying to minimize the energy of this physical system. This means a certain number of iterations are required to bring the system close to equilibrium, however, further iterations will perform very small changes and simply waste CPU time.

The performance of the Spring layout is $O(|V|^2)$, i.e. proportional to the number of vertices squared. This time also needs to be multiplied by the number of iterations in the layout to get the full time worst-case. Generally, the spring is best applied to smaller graphs with a more regular structure.

The springs have a natural length, if compressed to less than this length they repulse the attached nodes, if extended to more than this length they attract the attached nodes. The force with which they act upon the attached nodes is proportional to the difference between the current spring length and its natural spring length. The force with which each pair of nodes repulse each other is proportional to the inverse of the distance between the nodes

squared.

The key values in the spring layout are the spring length, the spring force and the repulsive force. The default values of the layout are set to behave well for a general graph. Increasing or decreasing the repulsive force only tends to affect local clusters shapes. Higher values for the spring force tends to lead to instability and oscillation of clusters and even the whole graph. Spring length tends to only affect the density of the graph, not the actual layout formed.

The Spring layout acts fairly slowly and so many iterations are required before an equilibrium between the nodes is found, the number of iterations tends to increase with the number of nodes in the layout. The spring layout constructor takes the number of iterations to be performed as a parameter.

The speed with which the spring layout produces a pleasing result can depend upon the input graph. Sometimes it is worth placing the nodes in random positions before applying the spring layout, or possibly applying the circle layout first. The `tilt()` method on `JGraphFacade` provide random placements of specified nodes. The example that ships with Layout Pro applies snap to grid to the cells. If the spring layout has short spring lengths and high spring forces, this can result in cells being overlaid. The spring layout might be used without snap to grid in this case.

7.3.2.2 Fast Organic Layout

The two aesthetic aims of the Fast Organic (FO) layout are that vertices connected by edges should be drawn close to one another and that vertices should not be drawn to close to one another. The attractive and repulsive forces are simply variations on those used in the spring embedded layout. Their formulae are intended to be easier to compute and better at overcoming local minima positions. The FO layout adds the concept of temperature, whereby the maximum distance that nodes can move decreases over between each iteration. This is intended to reduce instability in the layout and force the layout to settle in its later stages.

The performance of the FO layout is $O(|V|^2 + |E|)$ per iteration, i.e. proportional to the number of vertices squared. This time also needs to be multiplied by the number of iterations in the layout to get the full time worst-case. Generally, the FO is best applied to smaller graphs with a more regular structure.

The FO layout is much like the Spring Embedded in that it is a force directed layout with the same top level algorithm. Each iteration consists of taking each vertex in turn and calculating a force upon it based on connected edge and their distance to all other vertices. The FO layout also introduced the idea of temperature, whereby the maximum move of any vertex decreasing with each iteration, assisting the layout to 'settle'.

The force repulsing vertices in the FO is proportional to inverse of the distance between the nodes and the attractive forces between connected nodes are proportionals to the square of the distance between them. The constant, k, also used in both equations is the distance at which connected vertices are at equilibrium. The lack of a logarithmic calculation, as required in the Spring Embedded algorithm, make the FO one of the faster force directed layouts. The number of iterations required to produce a pleasing result cannot be determined

in advance, but the number of nodes in the graph will affect this number.

7.3.2.3 Inverted Self Organising Map

Although not strictly a force-directed layout, the ISOM layout uses the idea of filling the space evenly with vertices and of causing connected vertices to attract each other. Rather actually calculating forces to be applied to vertices, the ISOM layout uses an heuristic to achieve its aim. The algorithm involves selecting a random point in the graph area and picking the vertex closest to that point. This vertex is moved towards that points as well as all vertices connected to that initial vertex by up to a set number of edge steps. The amount by which the vertices are moved decreases the greater the number of edges in the shortest path between the current and initial vertex. The initial number of edge steps is decreased during the layout so that the later steps form local clusters of connected vertices.

The computational effort per iteration is linear, O(|N|). This comes from the effort of finding the closest node to the random point. When JGraph implements a spatial index structure this will improve to O(log|N|). Only a selection of nodes are moved per iteration and so a greater number of iterations are required for larger graphs. Generally, the number of iterations required is proportional to the number of vertices and so the computational effort, including the number of iterations, will always be O(|V|). The paper describes 500 iterations as being enough for 25 nodes, thus maxIterationsMultiple, which defines the vertices to number of iterations factor, defaults to 20. The ISOM is the fastest of the force-directed family of layouts in this package.

The two important data to setup in an ISOM layout are the radius and the bounds of graph. The bounds determines within which area the random positions will be located and so the area within which the nodes will be distributed. If you prefer to just specify an average density of nodes, use `densityFactor` to do this. The `moveRadius` field determines the number of neighbour nodes, in addition to the closest node to the random position, that are moved towards that point. It defines the actual number of edges limit that will be traversed to find node to move. Changing this value affects the clustering behaviour of the layout.

7.3.2.4 Organic Layout

This layout is an implementation of a simulated annealing layout, which describes the following criteria as being favourable in a graph layout: (1) distributing nodes evenly, (2) making edge-lengths uniform, (3) minimizing cross-crossings, and (4) keeping nodes from coming too close to edges. These criteria are translated into energy cost functions in the layout. Nodes or edges breaking these criteria create a larger cost function, the total cost they contribute is related to the extent that they break it. The idea of the algorithm is to minimise the total system energy. Factors are assigned to each of the criteria describing how important that criteria is. Higher factors mean that those criteria are deemed to be relatively preferable in the final layout. Most of the criteria conflict with each other to some extent, the default values selected are a broad balance between the criteria, though note that the factors are not normalized and so their values vary somewhat.

In addition to the four aesthetic criteria the concept of a border line which induces an energy cost to nodes in proximity to the graph bounds is introduced to attempt to restrain the graph. All of the 5 factors can be switched on or off within the layout.

Simulated Annealing is the most expensive layout in this package computationally (when all criteria switched on), but it can produce good results over a range of graphs. Layouts like the spring layout only factor in edge length and inter-node distance being the factors that provide the most aesthetic gain relative to their computational intensity. The additional factors are relatively more expensive but can have very attractive results. The performance of the Simulated Annealing layout is $O(|V|^2|E|)$ per iteration in the worst case.

In the configuration details that follow, there are examples of the different results produced by the annealing layout using the different settings. Note that the same input graph was used for each example and that the `isDeterministic` flag was set to `true`, i.e. there were no random elements in the layout process.

Since the annealing layout is the most costly computationally, a good approach, where improved performance is required, is to perform an ISOM layout followed by the annealing just in the fine tuning stage.

7.3.2.4.1 isOptimizeNodeDistribution and nodeDistributionCostFactor

`isOptimizeNodeDistribution` determines whether or not to attempt to distribute nodes evenly around the available space. If `isOptimizeNodeDistribution` is set to `true` then `nodeDistributionCostFactor` is the factor by which the cost of a particular node distribution is multiplied by to make an energy cost contribution to the total energy of a particular graph layout. Increasing this value tends to result in a better distribution of nodes across the available space, at the partial cost of other graph aesthetics, in particular edge lengths.

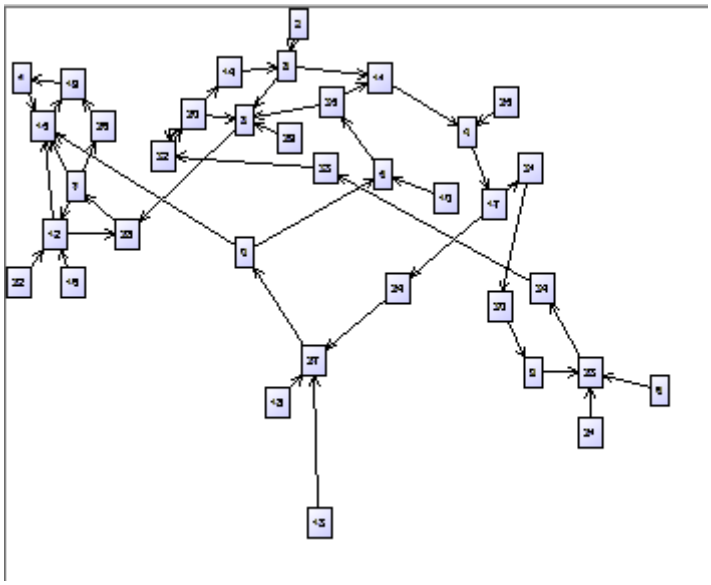

Illustration 64 : nodeDistributionCost = 10,000

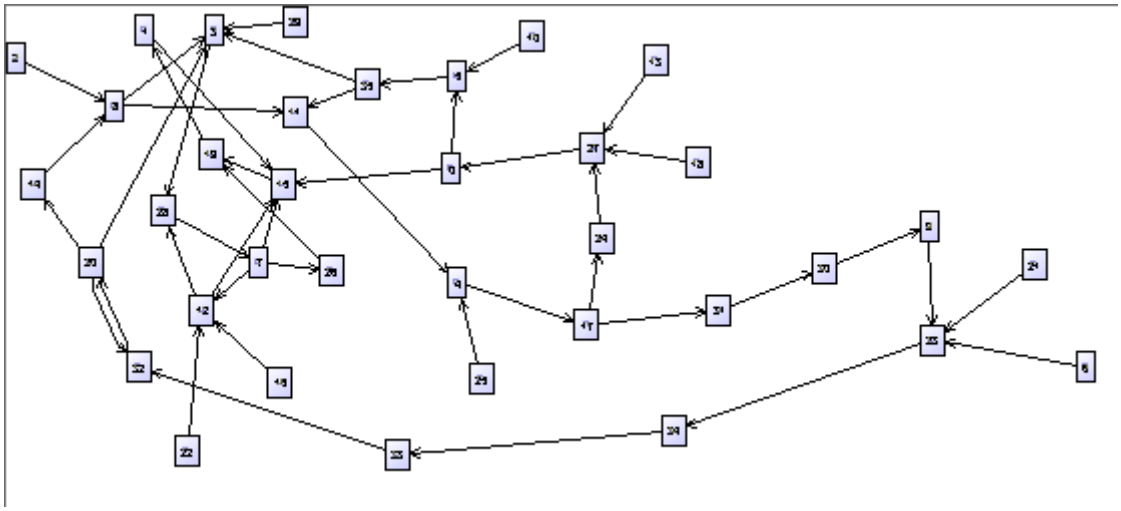

Illustration 65 : The same graph with nodeDistributionCost = 500,000

7.3.2.4.2 isOptimizeEdgeLength and edgeLengthCostFactor

`isOptimizeEdgeLength` determines whether or not to attempt to minimise edge lengths. If `isOptimizeEdgeLength` is set to `true` then `edgeLengthCostFactor` is the factor by which the cost of a particular set of edge lengths is multiplied by to make an energy cost contribution to the total energy of a particular graph layout. Increasing this value tends to result in shorter overall edge lengths, at the partial cost of other graph aesthetics, in particular node distribution.

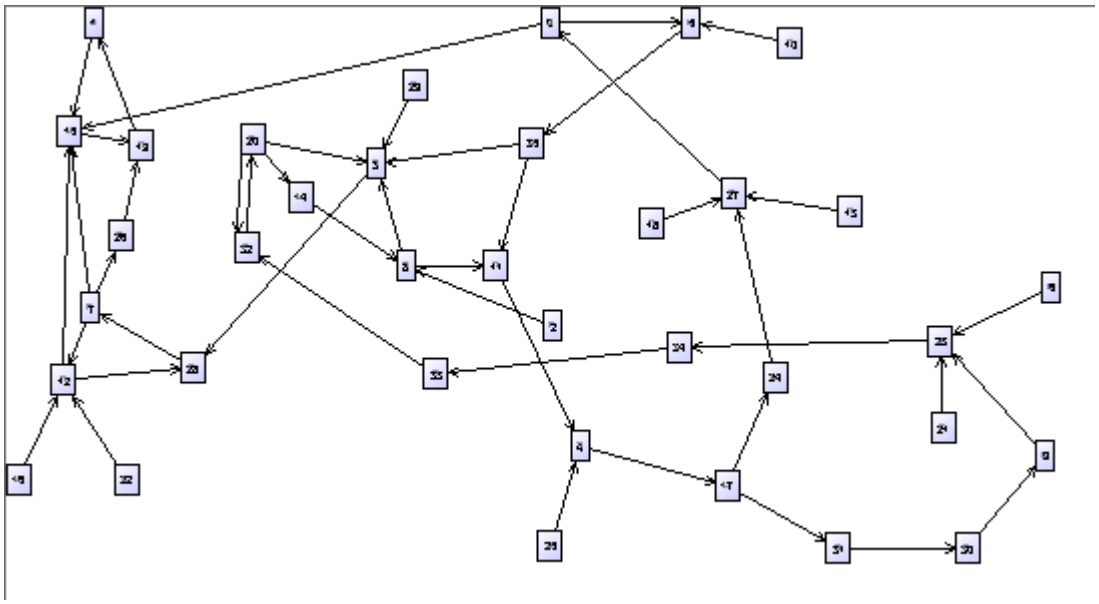

Illustration 66 : edgeLengthCostFactor = 0.01

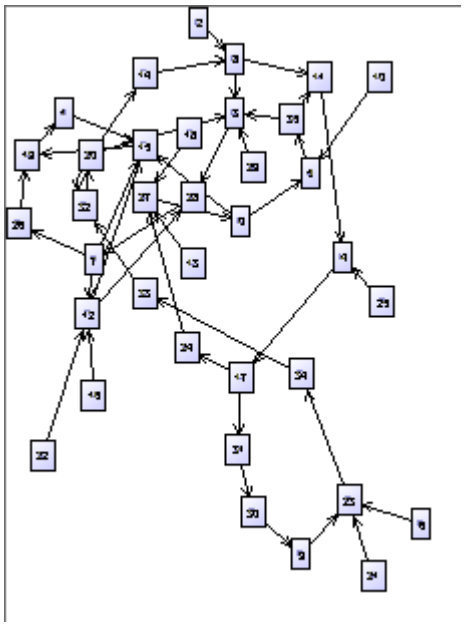

Illustration 67 : The same graph with EdgeLengthCostFactor = 0.1

7.3.2.4.3 isOptimizeEdgeCrossing and edgeCrossingCostFactor

`isOptimizeEdgeCrossing` determines whether or not to attempt to minimise the number of edges crossing that appear in the laid out graph. If `isOptimizeEdgeCrossing` is set to `true` then `edgeCrossingCostFactor` is the factor by which the cost of instance of an edge crossing is multiplied by to make an energy cost contribution to the total energy of a particular graph layout. Increasing this value tends to result in few edge crossing, at the partial cost of other graph aesthetics, usually edge length. A number of types of graph do not work well with aggressively high values for `edgeCrossingCostFactor`. This is because trying to avoid edge crossing results in nodes being spread out to avoid edge overlap and this results in longer edges. If the graph cannot be laid out in a way that avoid a number of overlaps, the longer edges can result in an increase in the number of edge crossing, as shown in the example below.

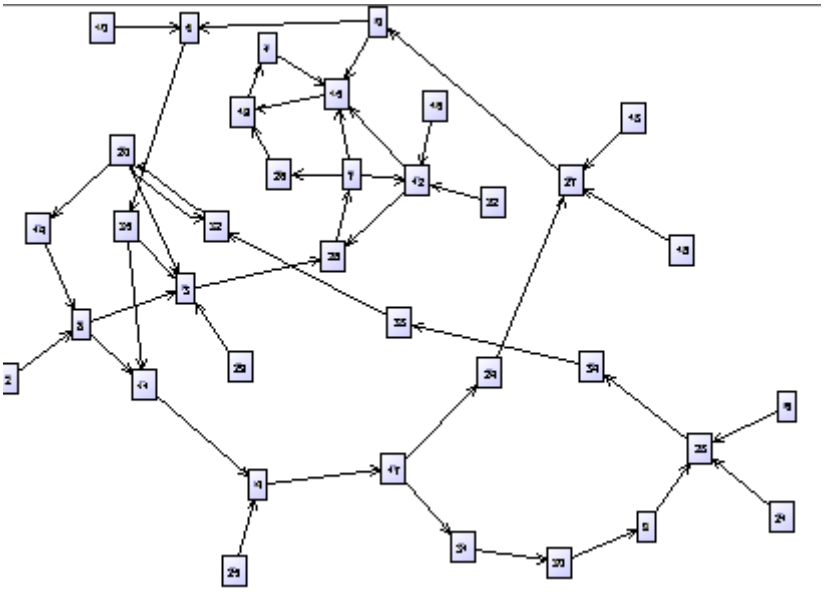

Illustration 68 : edgeCrossingCostFactor = 500

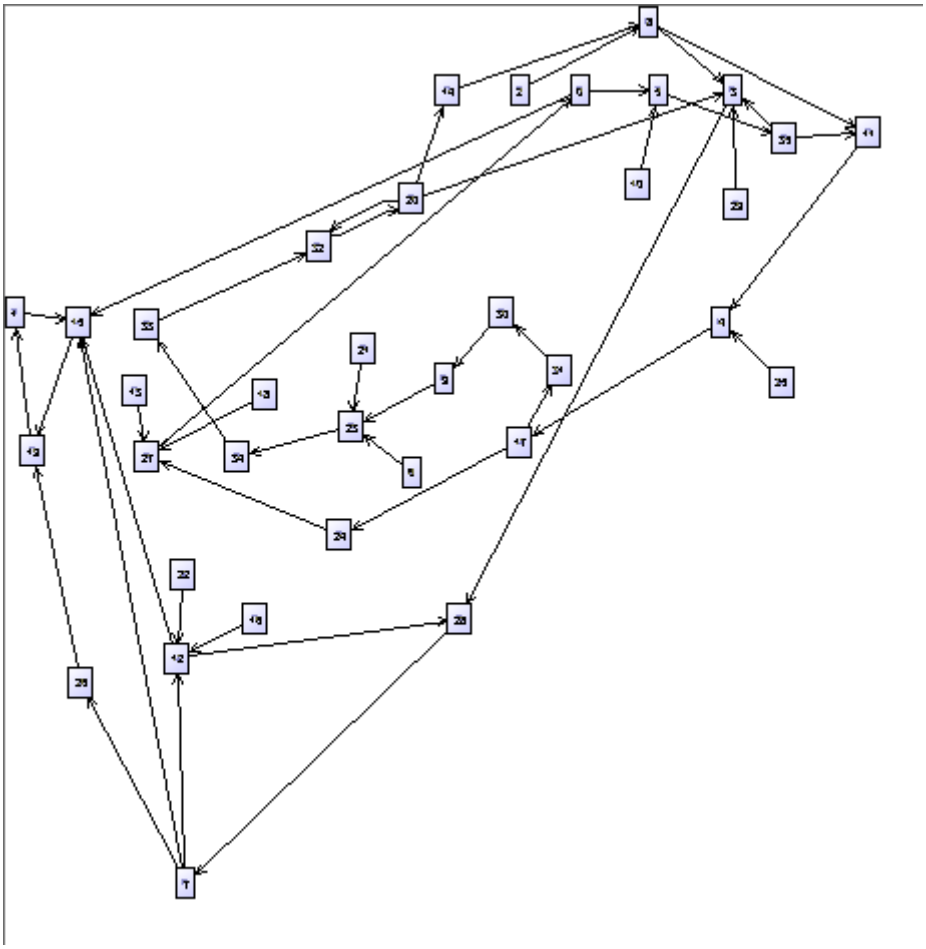

Illustration 69 : The same graph with edgeCrossingCostFactor = 500,000

7.3.2.4.4 isOptimizeEdgeDistance, edgeDistanceCostFactor, isFineTuning and fineTuningRadius

`isOptimizeEdgeDistance` determines whether or not to attempt to move nodes away from edges that pass close by to them. If `isOptimizeEdgeDistance` is set to `true` then `edgeDistanceCostFactor` is the factor by which the cost of a particular set of edge to nodes distances is multiplied by to make an energy cost contribution to the total energy of a particular graph layout. Increasing this value tends to result in nodes being moved away from edges, at the partial cost of other graph aesthetics, usually node distribution and edge length.

Optimizing edge to node distance to computational expensive and pointless until the end of an annealing layout. For this reason, it is deemed a fine tuning mechanism to be performed in the final stages of the layout. `isFineTuning` determines whether or not any fine tuning will take place. If it is set to `false` then the `isOptimizeEdgeDistance` value is ignored. If it is set to `true`, then fine tuning will start when the current `moveRadius` (see the section on `moveRadius`) reaches the value held by `fineTuningRadius`.

In summary, edge to node distance will only be taken into account if `isFineTuning` and `isOptimizeEdgeDistance` are both set to `true`, which are their default values. The radius within which new test positions for cells that are candidates for moving decreases through each layout iteration. When it reaches `fineTuningRadius`, the edge to node distance cost factor will start to be used and continue until the layout terminates.

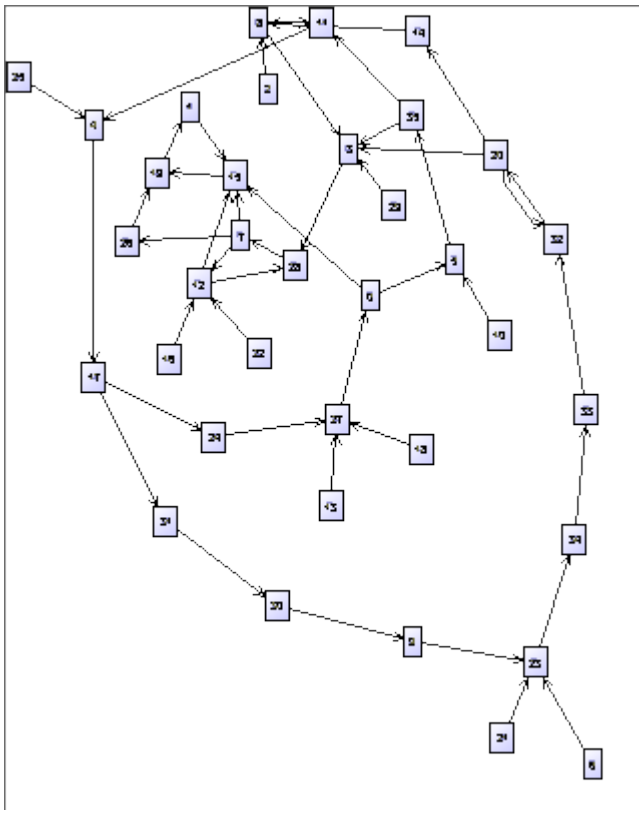

Illustration 70 : No fine tuning - no edge to node distance cost factor used

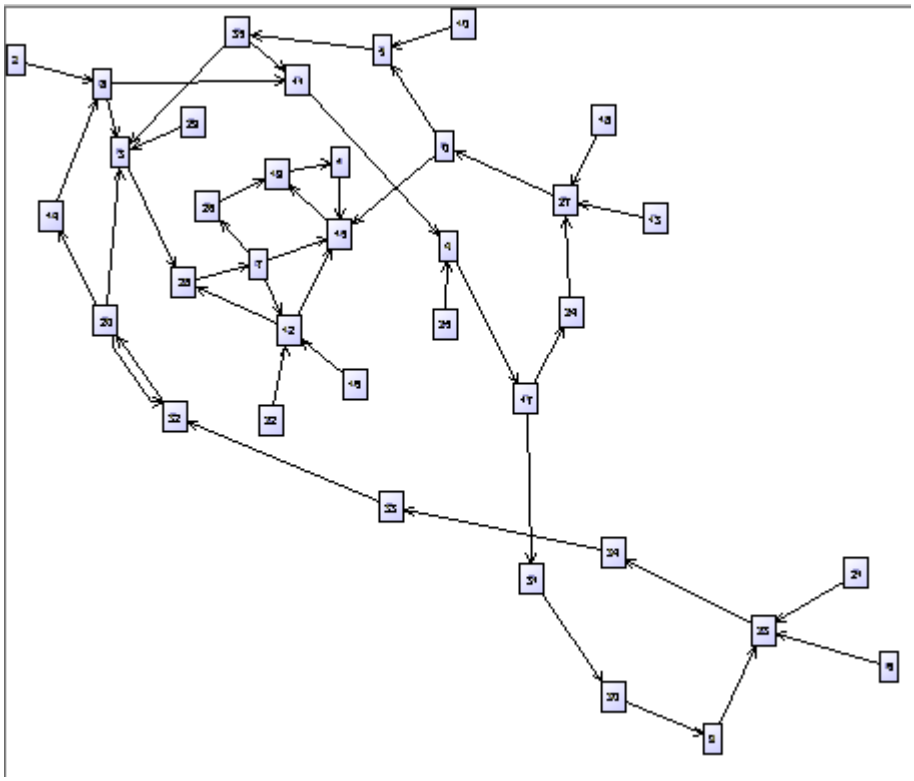

Illustration 71 : The same graph with edgeDistanceCostFactor = 4000

7.3.2.4.5 isOptimizeBorderLine, borderLineCostFactor and averageNodeArea

`isOptimizeBorderLine` determines whether or not to attempt to restrain the nodes within a set boundary. If `isOptimizeBorderLine` is set to `true` then `borderLineCostFactor` is the factor by which the cost of a particular set of node to boundary distances is multiplied by to make an energy cost contribution to the total energy of a particular graph layout. Increasing this value tends to result in nodes staying within the boundary, at the partial cost of other graph aesthetics, usually node distribution if the graph is densely packed. It is not impossible that a node might escape this boundary, though this becomes less likely the higher the value given to this factor.

There are three ways of setting the boundary within which nodes are attempted to be constrained. The first method is to set `averageNodeArea` before calling the `run()` method. This variable defines the average area that each node will be given and using this and the total number of nodes the total area of the boundary is calculated. Note that the boundary will be square shaped. This is a good way to keep the node density reasonably constant without having to worry about the size of the graph. Setting this variable to a non-zero positive value overrides any other method of setting the boundary for this layout.

The second mechanism is to use the constructor of the annealing layout that accepts a rectangle. The sets up the boundary for the lifetime of the layout object instance, unless overridden by setting `averageNodeArea`.

The third method is used automatically if neither of the first two are. This just sets the boundary to the bounds of the graph before the layout is applied.

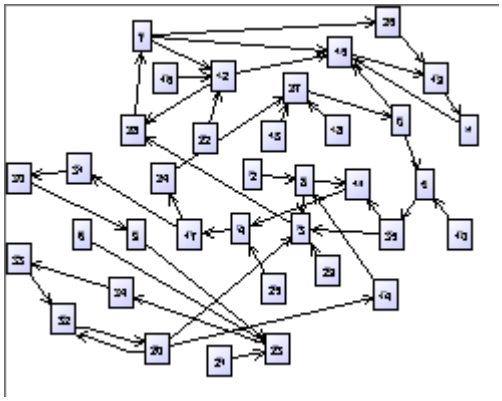

Illustration 72 : Bounds set using constructor and borderLineCostFactor = 500

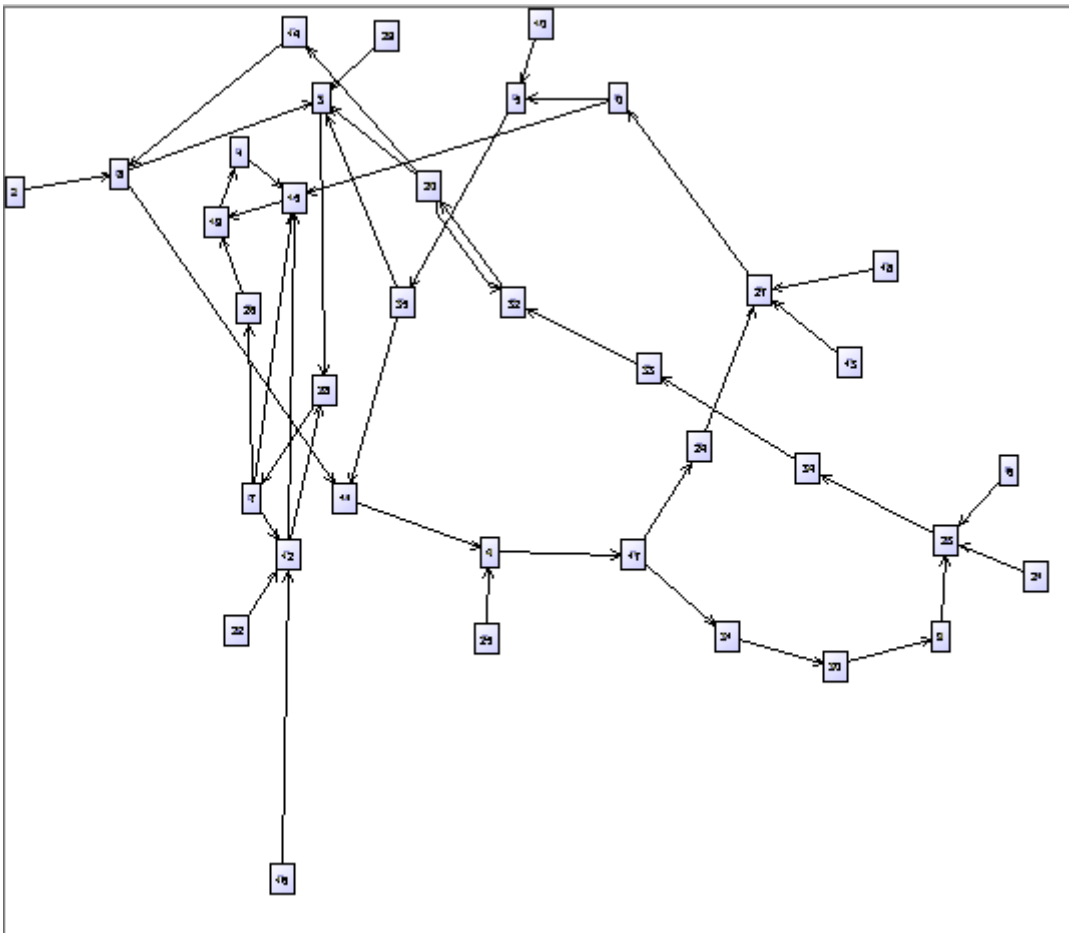

Illustration 73 : The same graph with isOptimizeBorderLine = false

7.3.2.4.6 minMoveRadius, initialMoveRadius and radiusScaleFactor

At each iteration each cell has a number of positions around it selected as candidate positions to move to, in an attempt to decrease the total system energy. Those candidates positions are at fixed angles around the perimeter of a circle that has the node as its centre. The radius of that circle starts at `initialMoveRadius` and decreases with each iteration by being multiplied by `radiusScaleFactor`. The value of `initialMoveRadius` is determined by the layout, there is no need to override it unless for a specific reason. `radiusScaleFactor` is a `double` between 0.0 and 1.0, lower values improve performance but raising it towards 1.0 can improve the resulting graph aesthetics. When the radius hits the minimum move radius defined, `minMoveRadius`, the layout terminates, unless the maximum number of iterations is reached first. The minimum move radius should be set a value where the move distance is too minor to be of interest.

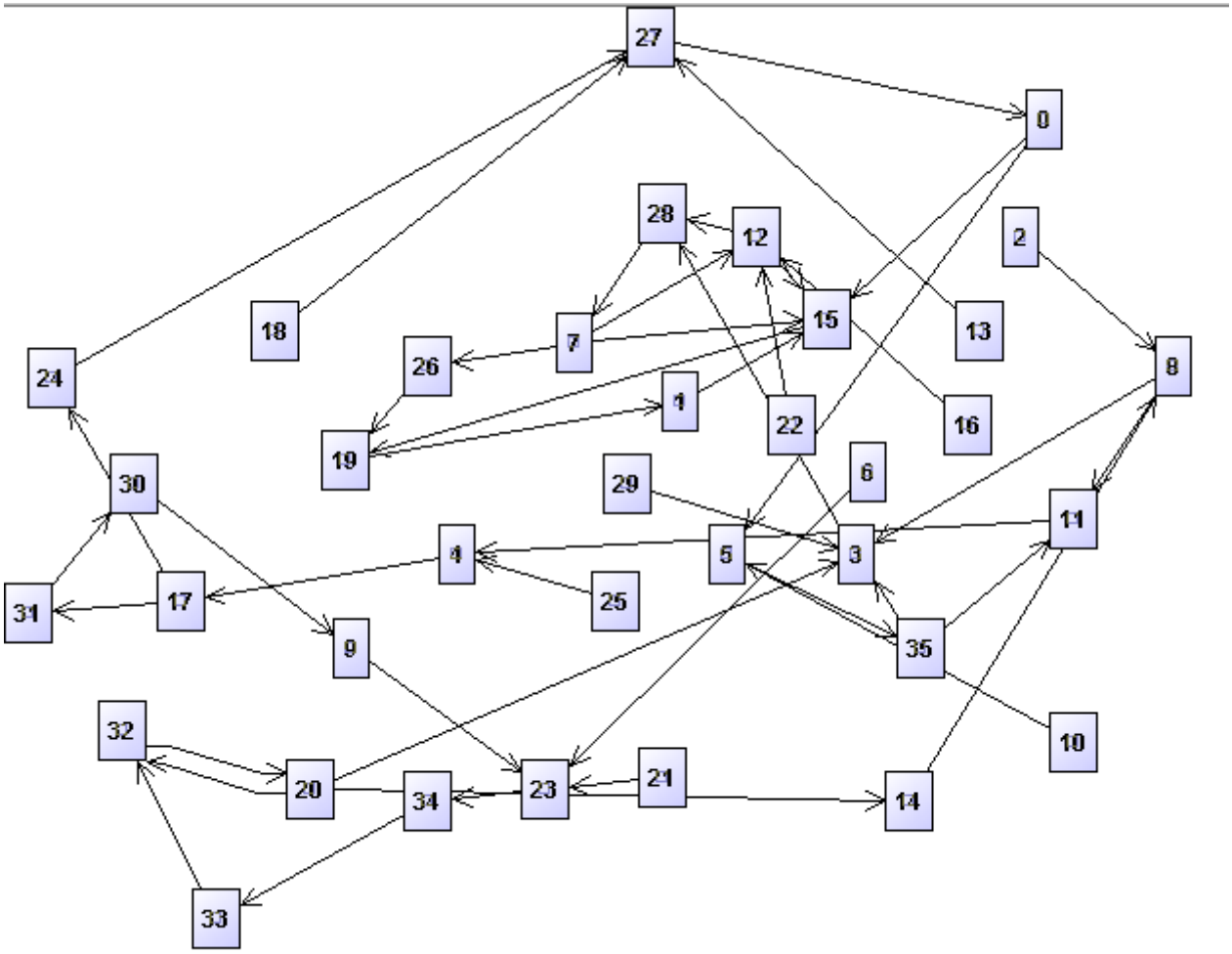

Illustration 74 : radiusScaleFactor = 0.5

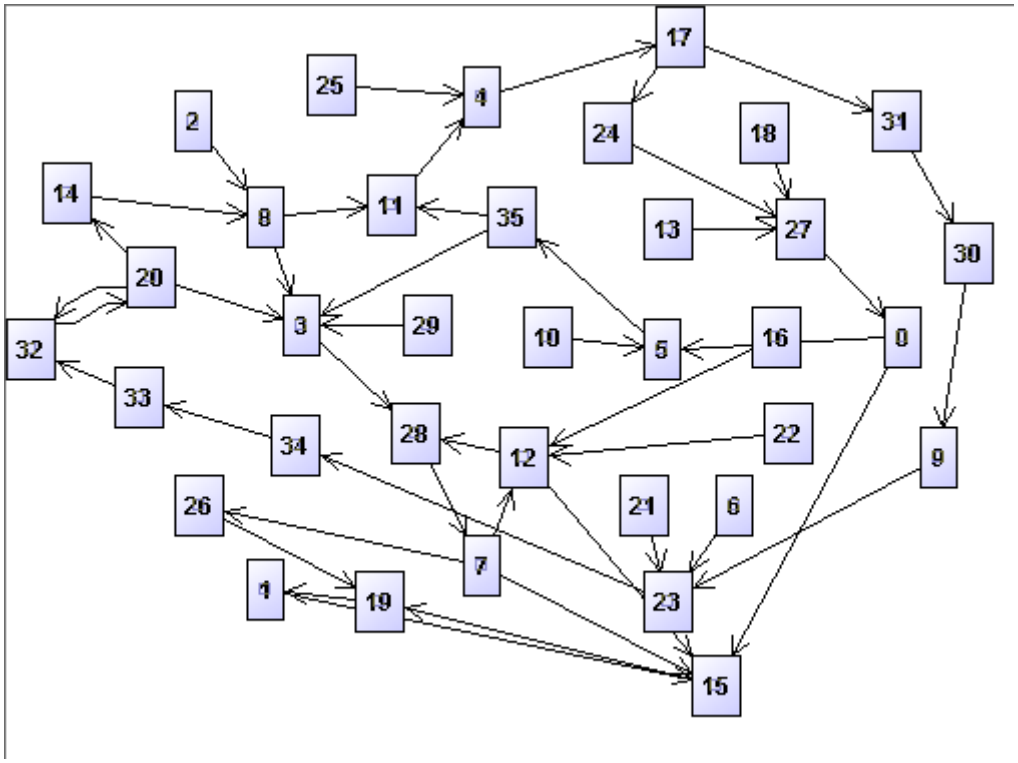

Illustration 75 : The same graph with radiusScaleFactor = 0.9

7.3.2.4.7 maxIterations

maxIterations is the maximum number of layout iterations that can take place. Layouts can terminate before this value is reached because the minimum radius value has been reached, or the layout has been unchanged for a certain number of rounds.

7.3.2.4.8 unchangedEnergyRoundTermination

If, at the end of an iteration it is determined whether any changes were made. If not, the count of number of rounds where no change has taken place is incremented. If this count reaches `unchangedEnergyRoundTermination` the layout terminates. If nothing is being moved after a number of rounds it is assumed a good layout has been found. In addition to this if no nodes are moved during an iteration the move radius is halved, presuming that a finer granularity is required.

7.3.2.4.9 isDeterministic

The `isDeterministic` flag defines whether or not the annealing layout should produce the same result for a given input graph and settings. The annealing layout uses random values in a few places to attempt to improve the output. Setting `isDeterministic` to `true` degrades the output only marginally, if at all, and is useful if you would like to experiment with the layout settings knowing that constant settings values produce a constant output.

7.3.2.5 Hierarchical Layout

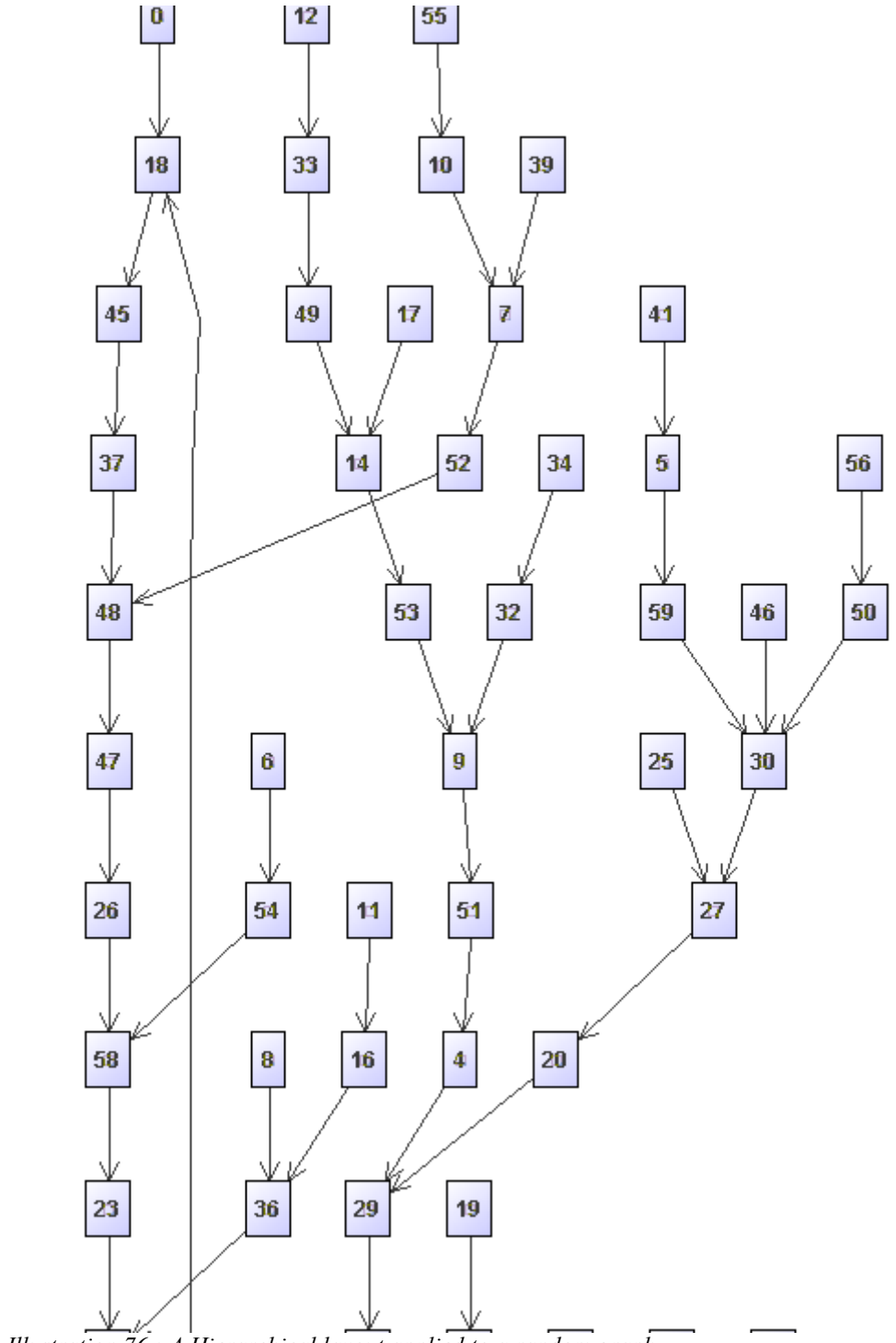

Illustration 76 : A Hierarchical layout applied to a random graph

The hierarchical layout is designed to work on directed graphs that have an overall flow, that is, some start point(s), some end point(s) and some overall flow between those points. Often graphs that have become too complex for a tree layout require the use of a hierarchical layout. These layouts are commonly applied to workflows, process modelling diagrams, software engineering diagrams and processes, databased visualization and other directed models.

The graph should have some distinct start and end node(s), that is at least one node with no incoming edges and at least one node with no outgoing nodes, respectively. The roots of the layout may be set explicitly, alternatively, by passing them in through the constructor:

```
      Object roots = getRoots(); // replace getRoots with your own
Object array of the hierarchical roots. NOTE: these are the root cell(s)
of the tree(s), not the roots of the graph model.
      JGraphFacade facade = new JGraphFacade(graph, roots); // Pass the
facade the JGraph instance
      JGraphLayout layout = new JGraphHierarchicalLayout(); // Create an
instance of the hierarchical layout
      layout.run(facade); // Run the layout on the facade.
      Map nested = facade.createNestedMap(true, true); // Obtain a map
of the resulting attribute changes from the facade
      graph.getGraphLayoutCache().edit(nested); // Apply the results to
the actual graph
```

It should be noted that the hierarchical layout might insert control points in certain edges to route them correctly. This should be taken into account when performing additional editing without applying the layout again. The `JGraphFacade` provides a method `resetControlPoints` to assist with removing control points. Calling this method will remove all additional control points from the edges passed into the next layout applied.

7.3.2.5.1 Orientation

Orientation refers to the compass direction in which the root node(s) of the layout will be located relative to the rest of the tree. Using the `setOrientation()` method you can set the orientation to `SwingConstants.NORTH`, `SwingConstants.EAST`, `SwingConstants.SOUTH` or `SwingConstants.WEST`. The literal values of these constants are 1, 3, 5 and 7 at the time of writing, but the variable names should always be used.

7.3.2.5.2 Intra Node Distance and Inter Rank Cell Spacing

`interRankCellSpacing` is the distance between the lowest point of any vertex on one layer of the layout to the highest point of any vertex on the next level down. `intraCellSpacing` is the minimum distance between any two vertices on the same level.

7.3.2.5.3 isDeterministic

The `isDeterministic` flag defines whether or not the hierarchical layout should produce the same result for a given input graph and settings. The hierarchical layout does not assure that layer will be ordered in the order as provided by the graph model unless this flag is set. Setting `isDeterministic` to `true` may degrade the output somewhat for larger graphs, since it introduces a component with performance somewhere between linear and square.

7.3.3 EDGE ROUTING

7.3.3.1 Orthogonal Edge Routing

OrthogonalLinkRouter is a global edge router designed to avoid the overlap of edges with vertices by constructing the edges in vertical and horizontal segments. The router is run like any standard layout, its performance means it is not a good candidate for displaying during a preview.

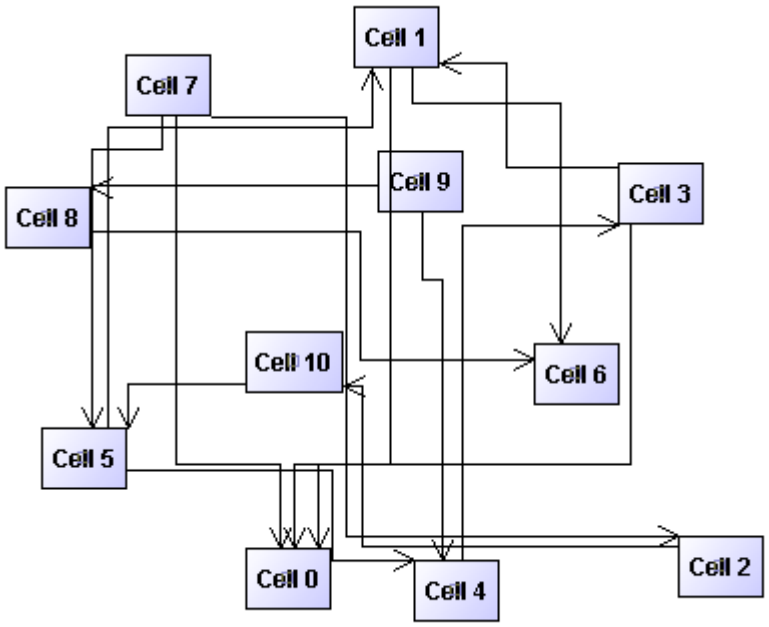

Illustration 77: The Orthogonal Edge Router

7.3.4 SIMPLE LAYOUTS

7.3.4.1 Circle Layout

The circle layout arranges all the node into a circle, with constant spacing between each neighbour node. The performance of this layout is proportional to the number of vertices in the circle. Although, circle layouts are not commonly used by themselves, it has been noted that some non-deterministic layouts (force-directed mainly) produce a better result if separated out by a circle layout first. If a better result is not produce, often the same quality of result can be obtained quicker (through less iterations of a force-directed layout) then without the initial circle applied. There isn't a separate class for this layout as it is a trivial implementation. Instead, the method, `circle(List vertices)`, is part of the facade. Below is an example of using the circle layout:

```
      JGraphFacade facade = new JGraphFacade(graph); // Pass the facade
the JGraph instance
      JGraphLayout layout = new
JGraphSimpleLayout(JGraphSimpleLayout.TYPE_CIRCLE); // Create an
instance of the circle layout
      layout.run(facade); // Run the layout on the facade.
      Map nested = facade.createNestedMap(true, true); // Obtain a map
of the resulting attribute changes from the facade
      graph.getGraphLayoutCache().edit(nested); // Apply the results to
the actual graph
```

7.4 Using the Example Source Code

7.4.1 THE PROGRESS METER

Some of the layouts are more CPU intensive than others and so require some graphical indication than the application is still performing processing and has not crashed. The standard way to do this is using a progress meter. A custom progress meter class is provided, `JGraphLayoutProgress`, that may be used on layouts that implement the `Stoppable` interface defined in `JGraphLayout` that enables the user to stop the layout running and return to the previous graph if the layout takes too long. Layouts supporting the progress meter fire a property change event to set the maximum value of the progress meter has well each time a significant change to the value of the progress occurs.

The maximum value of the progress meter is set either as a constructor parameter, or passed into the `reset()` method. Layouts call the `setProgress()` method during the running of the layout to update the progress.

To implement a progress meter in an application, base it on the example in `JGraphExampleLayoutCache.layout()`. Here, a `PropertyChangeListener` is created that processes the possible event types. These event types are, specifically, `JGraphLayoutProgress.PROGRESS_PROPERTY` for a new value of the progress meter and `JGraphLayoutProgress.MAXIMUM_PROPERTY` to set the maximum progress value. A standard `ProgressMonitor` can be used and implement cancellation functionality as shown in the example code.

Appendix A – Definitions

- **self-loop** - an edge with both endpoints attached to the same vertex, also known as a reflexive edge.

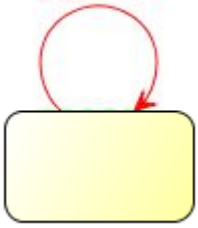

Illustration 78: A self-loop edge

- **parallel edges** – more than one edge connecting a pair of vertices.

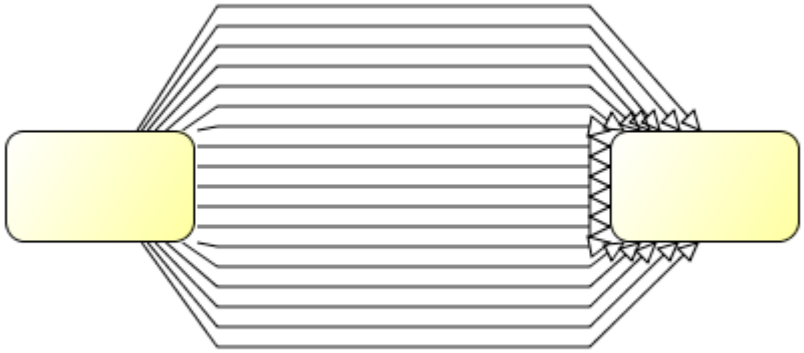

Illustration 79: A number of parallel edges

- **directed edge** - is an edge with a specific direction, like a vector. Directed edges have source cells and target cells at their endpoints to indicate the direction. Note that all edges in JGraph have a direction internally. It is up to an application whether to take edge direction into account or to draw edge arrows.
- **hyperedge** - an edge that has more than two endpoints and so cannot be represented by just a line.
- **incident** – If an edge connects to a vertex it is described as incident of that vertex.
- **degree** – The degree of a vertex is the number of edges incident upon it.
- **simple graph** – A graph that has no loops and no parallel edges
- **directed graph** – all edges of the graph are directed. Exchanging all the directed edges for undirected edges provides the underlying graph.
- **oriented graph** – a directed graph whose underlying graph is simple.
- **hypergraph** - a graph with hyperedges