

# **JGraph in Action**

**Using JGraph in real-world applications**

**Gaudenz Alder**

---

# **JGraph in Action: Using JGraph in real-world applications**

Gaudenz Alder

Published 2005

Copyright © 2005 Gaudenz Alder

## **Abstract**

This document describes the implementation of the JGraphpad Pro editor application, the underlying JGraph Editor Framework, the customizations of the JGraph component used in the application and the plugins implemented on top of it. The JGraphpad Pro application serves as an example for using the framework, and provides a set of add-on components for JGraph. The document is aimed at the practitioner with a focus on problem solving and provides solutions for file I/O, printing, web-integration, datatransfer, custom models, tools, cell views, renderers, groups, folding, layout and display- and interaction fine-tuning for JGraph and an extensible application development framework.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the author.

The programs in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations nor does it accept any liabilities with respect to the programs.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from JGraph, Ltd.

Neither JGraph, Ltd. nor its employees are responsible for any errors that may appear in this publication. The information in this publication is subject to change without notice.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

PostScript is a registered trademark of Adobe Systems Incorporated.

GIF is a trademark of Unisys corporation.

---

---

---

---

# Table of Contents

1. Introduction .....	1
1. Overview .....	1
2. History .....	1
2.1. Comparison .....	1
3. Design .....	1
3.1. Goals .....	1
3.2. Decisions .....	2
3.3. Components .....	3
4. Directory Structure .....	4
4.1. Jar Files .....	5
4.2. Dependencies .....	5
5. Build Process .....	5
2. JGraph Editor Framework .....	7
1. Introduction .....	7
2. Editor .....	8
2.1. Exit Hook .....	9
3. Document Model .....	9
3.1. Input and Output .....	10
3.2. Files and Diagrams .....	11
4. Editor Kit .....	12
4.1. Actions .....	13
4.2. Tools .....	14
5. UI Factory .....	15
5.1. Factory Methods .....	16
6. Settings .....	18
6.1. Merging UI Configuration Files .....	19
6.2. Application-wide Map .....	19
6.3. Shutdown Hooks .....	20
7. Resources .....	20
8. Diagram Pane .....	21
8.1. Undo Support .....	23
8.2. Backgrounds .....	23
8.3. Autoscaling .....	23
8.4. Printing .....	24
8.5. Rulers .....	24
9. UI Components .....	24
9.1. Toolbox .....	24
9.2. Combo Box .....	25
9.3. Navigator .....	26
10. Summary .....	27
3. JGraphpad Pro .....	28
1. Introduction .....	28
2. JGraphpad and -Applet .....	28
3. Startup .....	29
3.1. Static Initialization .....	29
3.2. Editor Construction .....	30
3.3. Plugins Initialization .....	33
3.4. UI Construction .....	33
4. Shutdown .....	33
4.1. Quitting the Applet .....	34
5. JGraphpad Document Model .....	34
5.1. Adding objects .....	34
5.2. Removing objects .....	35

6. Custom Factory Methods .....	35
6.1. Main Window .....	35
6.2. Library Pane .....	41
6.3. Window Menu .....	43
6.4. Open Recent Menu .....	44
6.5. Status Bar .....	45
6.6. Console .....	45
6.7. Combo Boxes .....	46
7. Custom Tools .....	47
7.1. Adding Tools .....	47
7.2. Vertex Tool .....	49
7.3. Edge Tool .....	50
8. Dialogs .....	51
8.1. Built-In Dialogs .....	51
8.2. Custom Dialogs .....	53
9. Utilities .....	55
9.1. File Filter .....	55
9.2. Focus Manager .....	55
9.3. Morphing Manager .....	56
9.4. Other Utilities .....	56
10. Summary .....	57
4. Customizing JGraph .....	58
1. Introduction .....	58
2. User Objects .....	58
2.1. Properties .....	59
2.2. Tooltips .....	59
2.3. Cloning .....	60
2.4. Editing .....	60
2.5. Transactions .....	61
3. Rendering .....	62
3.1. Universal Renderer .....	63
3.2. Shapes .....	64
3.3. Heavyweights .....	64
3.4. Rich Text .....	64
4. In-Place Editing .....	65
4.1. Heavyweight Editing .....	66
4.2. Rich Text Editing .....	66
5. Moveable Ports .....	67
5.1. Port Location .....	67
5.2. Edge Handle .....	67
6. Drag and Drop .....	68
6.1. Chaining Handlers .....	68
6.2. Importing Files .....	69
6.3. Importing Text .....	69
7. Intercepting Mouse Events .....	69
7.1. Control Flow .....	69
7.2. Popup Menus .....	70
7.3. Folding .....	71
8. Special Features .....	72
8.1. Edge Groups .....	73
8.2. Connecting Edges .....	73
8.3. Edge Promotion .....	73
8.4. Grouping by Mouse .....	74
9. Summary .....	75
5. Plugins .....	76
1. Introduction .....	76
2. Internationalization .....	76
3. Browser Integration .....	76

---

4. Bean Shell .....	76
5. Codecs .....	77
6. EPS (Postscript) .....	77
7. JGX (JGraphpad 5.x) .....	77
8. L2FProd (UI Components) .....	77
9. Automatic Layout .....	78
10. PDF Export .....	78
11. SVG Export .....	78
11.1. Embedded Webserver .....	79
12. Summary .....	80
A. Actions .....	81
B. Tools .....	88
C. Software License .....	89
References .....	92

---

## List of Figures

1.1. Components .....	4
2.1. UI Factory .....	7
2.2. Editor Participants .....	8
2.3. Document Model .....	11
2.4. JGraphEditorDiagramPane .....	21
2.5. JGraphEditorToolbox .....	24
2.6. JGraphEditorComboBox .....	25
2.7. JGraphEditorNavigator .....	26
3.1. JGraphpadApplet .....	28
3.2. JGraphpad Startup .....	30
3.3. createKit .....	32
3.4. createFactory .....	32
3.5. JGraphpad Document Model .....	34
3.6. JGraphpad Document Tree .....	35
3.7. Main Window Structure .....	35
3.8. JGraphpadPane .....	38
3.9. JGraphpadLibraryPane .....	41
3.10. JGraphpadWindowMenu .....	43
3.11. Combo Boxes .....	46
3.12. Steps of the Vertex Tool .....	49
3.13. File Dialog .....	51
3.14. L2FProd Plugin Font Dialog .....	53
3.15. JGraphpadAboutDialog .....	53
3.16. Parallel Routers .....	56
4.1. User Object .....	58
4.2. User Object Properties .....	59
4.3. Tooltips .....	59
4.4. Cloning .....	60
4.5. In-Place Editing .....	61
4.6. Available Shapes .....	62
4.7. Rich Text Cell .....	65
4.8. Rich Text In-Place Editing .....	65
4.9. Heavyweight In-Place Editing .....	66
4.10. Moveable Ports .....	67
4.11. Popup Menus .....	70
4.12. Folding Icons .....	71
4.13. Group Resizing and Repositioning .....	72
4.14. Edge Groups .....	73
4.15. Connecting Edges .....	73
4.16. Edge Promotion .....	74
4.17. Grouping by Mouse .....	74
5.1. Bsh Plugin .....	76
5.2. Layout Plugin .....	78
5.3. SVG Plugin .....	78

---

## List of Tables

1.1. Build targets supported by the build file .....	5
2.1. Action Resources supported by the JGraphEditorFactory class .....	14
2.2. Tool Resources supported by the JGraphEditorFactory class .....	15
2.3. Autoscale policies supported by the JGraphEditorDiagramPane class .....	24
3.1. Property events fired by the JGraphpadFocusManager class .....	56
A.1. Actions implemented in the JGraphpadFileAction class .....	81
A.2. Actions implemented in the JGraphpadEditAction class .....	81
A.3. Actions implemented in the JGraphpadViewAction class .....	82
A.4. Actions implemented in the JGraphpadFormatAction class .....	84
A.5. Actions implemented in the JGraphpadCellAction class .....	86
A.6. Actions implemented in the JGraphHelpAction class .....	87
B.1. Implemented Tools .....	88



---

# Chapter 1. Introduction

## 1. Overview

The JGraph project contains three subprojects:

- **JGraph** — Open source graph component for JFC/Swing
- **JGraph Layout Pro** — Layout and algebra package for JGraph
- **JGraphpad Pro** — Diagram editor framework for JGraph

The three projects build on each other: JGraph Layout Pro uses JGraph, and JGraphpad Pro uses JGraph Layout Pro, even though it is also possible to build and run JGraphpad Pro without the layouts. The JGraph Swing component is described in [bib-Benson] and is a recommended reading for understanding this document. This document focuses on the framework and application and use of JGraph therein.

## 2. History

Historically, JGraphpad has been available since the start of the JGraph project. Initially it was provided as a proof of concept for JGraph, later extended to provide some hooks for subclassers. However, it turned out that most people use JGraphpad as a basis for their applications, which is why this book provides a complete new implementation of JGraphpad called JGraphpad Pro aimed at the professional development community.

Because the old architecture of JGraphpad was based on the Notepad example, which is a trivial application in terms of functionality and did not scale well, this reimplementaion is not considered to be a *re-design*, it is rather looked at as a new application that only has the name in common with the old version.

Note that the old version of JGraphpad was licensed under the GNU Public License, whereas JGraphpad Pro is licensed under a more commercially-oriented license. The term JGraphpad in this document always refers to the new implementation, namely JGraphpad Pro, unless explicitly stated.

### 2.1. Comparison

From a technical point of view, the old and the new version of JGraphpad are completely different. The new version provides a fully documented new codebase with a layered architecture, and acts as a platform to implement new applications, whereas the old version was merely a collection of useful code snippets.

However, from a end user's point of view, the applications appear to be visually similar, with JGraphpad Pro providing some useful additions, such as multidiagram documents, rich text cells, additional shapes, moveable ports and plugins, which enable further extension. However, the old version of JGraphpad is not actively maintained and developed any longer.

## 3. Design

### 3.1. Goals

The following design goals guided the implementation of JGraphpad Pro:

- MVC** The Model-View-Control design pattern is common in Java and Swing, and does not require a detailed explanation. The basic idea of this design pattern is "to separate the model from the view and establish a relation between them, such that more than one view can be attached to the same model, and all views are automatically updated when the model changes" ([bib-GOF]). This design pattern is used throughout all layers, for example to implement the document model in the framework.
- IOC** Inversion of Control (aka. dependency injection) suggests that instances of dependent classes should be created outside of a class and passed to the constructor or a setter method ("injected"). This allows better control of class configuration and makes the framework ready to be used with other frameworks that use or simplify dependency injection, such as the Spring framework. The design pattern also leads to simpler and more readable code and reduces complexity by pushing dependencies down along the class hierarchy.
- KISS** Keep it small and simple is not itself a design pattern, it *is* however an important design goal. As the only non-formal goal it is not possible to establish a set of coding rules from it, nevertheless, KISS has a strong impact on all layers because it establishes an implementation philosophy, namely one that tries to avoid all kinds of "unnecessary" complexity. The key issue here is to make the framework flexible enough but not bloated, or as Einstein said: "make it as simple as possible, but not simpler".

## 3.2. Decisions

Apart from the design goals, which do affect the design and implementation of the application, we also have to establish certain design decisions, which will affect the functionality of the application. It is important to make such decisions early on, and assess them with respect to the application functionality and limitations (eg. later use of the framework). Since design decisions can help to simplify code but also block later extensions, it is recommended to take the time and analyze all various aspects of such decisions, as it may be too expensive or even impossible to change them at a later time.

For the JGraphpad application, we tried to keep such decisions minimal, limit them to the use of JGraph and its graph cells and push them out of the framework into the application. The following important design decisions must be kept in mind when working with all parts of the code:

### 3.2.1. Graph Cells

Custom cells contain the actual data of an application within the user objects. In a custom application it is possible that cells may be created throughout the code. In order to centralize custom cell creation we need some control over where and how the cells are created. To implement custom cells, we use the fact that the graph model knows how to clone them (using the `cloneCells` method). Therefore, if we have an existing cell (with a user object stored in it) we can simply pass the instance to the model to get a clone, which may then be inserted into this same model.

This leads to the issue of creating the first cell for each cell type, the notion of *prototypes*. These are instances of graph cells which are publically available to all the various parts of the application that are in charge of creating cells. When a cell is created, the prototype instance is passed to the current graph model, cloned and the clone is inserted into the model. In the example below, an additional call is made to the model to set the user object of the clone:

```
Object clone = DefaultGraphModel.cloneCell(model, prototypeCell);
model.valueForCellChanged(clone, label);
```

This design decision has one major impact on all application sections that create cells: You need to

provide them with one or more prototypes. Furthermore, when using multiple models with different user objects within the same application, you must make sure that the `cloneCells` method of all graph models are able to clone all cells that they support, or return a null pointer to signal to a caller that this prototype is not compatible with the current model. Alternatively return a different user object and/or cell which is compatible with the current model.

In JGraphpad, the *toolbox*, *import*, *datatransfer* (drag and drop, copy/paste), *grouping* and *connecting* create new cells and do therefore have a reference to a prototype. Note that these prototypes are created prior to creating any graph model. The current graph model will get a chance to provide the caller with it's own cells and user objects during application runtime. This is achieved by overriding the `cloneCells` method. Consequently, this means that prototypes are part of the application configuration and are never changed during the lifecycle of the application. (Note that while the *LibraryPane* also creates new cells to implement the *autoboxing* feature, these cells are never actually used in the diagrams and therefore do not need to be cloned from the prototypes.)

### 3.2.2. Graphs

In the Editor Framework, a factory is in charge of creating the actual JGraph instance for a given Graph-LayoutCache. In order to create new graph instances, this factory must be overridden. (Note that it is therefore not recommended for a Plugin to introduce a custom graph, since a Plugin should not override the core classes.)

It is important to understand that the use of the graph layout cache, which is part of the editor's document model and created by a custom action in your application, is not affected by this design decision. This means the cell views (CellViewFactory), renderers, handlers and editors are customizable, only the actual JGraph instance (which defines basic aspects of a diagram, such as tooltips, drag and drop and double buffering) is created using a method of the factory.

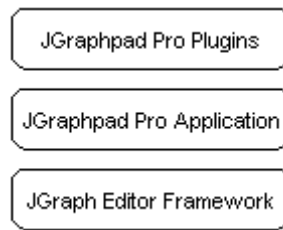
In order to support custom graphs, all components that create their own graphs (eg. *navigator*) must be either tied to one specific implementation of JGraph during their lifecycle, or they must provide a way to exchange their *backing graph* based on the currently focused graph. Note that it is only required to replace the backing graph if it affects the appearance or behaviour, eg. in the case of a custom tooltip you do not need to replace the backing graph of the default navigator, because it does not display tooltips. In general, you should always check the backing graph type against the focused graph, even if it is not required to replace the backing graph in certain situations, for example if it is read-only and your custom graph only affects behavioural functionality, such as an overridden `getDefaultPortForCell` method.

The default components implement both: All components are tied to exactly one graph during their lifecycle (because JGraphpad only has one custom graph), and they provide a way to replace the backing graph so they can be tied to multiple graphs if required. The initial graph is created using the factory's `createGraph` method with a dummy graph layout cache as an argument. For example, the following code is used to create the initial backing graph for the navigator:

```
JGraph backingGraph = editor.getFactory().createGraph(  
    new GraphLayoutCache());  
new JGraphEditorNavigator(backingGraph);
```

In the presence of multiple JGraph implementations, the process of updating the graph when the focus traverses entails checking the graph type (and replacing the graph when of the wrong type), updating the layout cache reference, and applying the settings of the focused graph to the backing graph. Note that the `GraphLayoutCache` instances created in the *New Repository* and *New Diagram* actions are not affected by this decision.

## 3.3. Components

**Figure 1.1. Components**

While earlier implementations of JGraphpad came without components (or layers), this new implementation has been split into three components, namely the *Editor Framework*, the *JGraphpad Pro Application* and the *JGraphpad Pro Plugins*. This way a clear separation between the various parts of the application is accomplished:

- The **JGraph Editor Framework** (framework) contains the common parts of all editors that use JGraph and defines the basic application structure and lifecycle. Initially planned as a separate product, it is now bundled with and named after the main application, which also forms the main example of the framework.
- The **JGraphpad Pro Application** (application) uses the framework and provides common components to implement a diagram editor. The developer is free to modify the JGraphpad Pro application and devise a new application from it.
- The **JGraphpad Plugins** contain blocks of functionality, typically with dependencies to external libraries (eg. SVG export, which uses the Batik library). However, it is not required for a Plugin to define a large block, it can also be a small block, such as a menu item. In general, plugins are a flexible way to add functionality to existing applications without having to recompile the application itself.

## 4. Directory Structure

The directory structure reflects the three components. Each component lives in a directory within the source tree named after the component, namely `src/framework`, `src/application` and `src/plugins/...`. The plugins directory contains "subcomponents" (plugins) with separate source trees. Furthermore, there is a `doc` directory which contains this book in PDF and HTML, and the API specification, a `jars` directory, which contains the required libraries for running and compiling JGraphpad Pro (dependencies) and the `lib` directory, which contains the framework and application jar files.

The Ant build file, called `build.xml`, the `LICENSE` and `ChangeLog` are the only files residing in the top-level directory:

```
src/framework
src/application
src/plugins/...
doc/api
doc/html
doc/pdf
jars
lib
build.xml
```

LICENSE  
ChangeLog

## 4.1. Jar Files

The jar files in the `lib` directory represent the three components into which the architecture is split. The `jgrapheditor.jar` contains the Editor Framework and JGraph component. The `jgraphmicro-pad.jar` is a stripped-down version of the editor, containing the Editor Framework, JGraph component and the minimal parts of the application. The `jgraphpad.jar` contains the full-featured application, including plugins and dependent libraries.

## 4.2. Dependencies

From the files in the `jars` directory, only `jgraph.jar` is required to build the framework and application. All other files, including the `jgraphlayout.jar` are used by the optional plugins. The JGraph version to be used with JGraph Layout Pro should match the version in the `jgraph.jar` file. Using a different version may lead to runtime exceptions when applying a layout and should therefore be avoided. The other dependencies are used by the plugins as mentioned in the respective chapter. The `looks-1.3.1.jar` is used to implement a nice look and feel. It may be downloaded from <http://www.jgoodies.com/>.

## 5. Build Process

The build process is entirely Ant-based and requires Java and Ant to be properly installed on your system. To keep it simple, all components are build using the top-level `build.xml` file, which provides the following build targets:

**Table 1.1. Build targets supported by the build file**

Target	Description
all (default)	Clean up and produce all distributions
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
dist	Produce fresh distributions
distclean	Clean up the distribution files only
dpc	Generate all documentation
generate	Generate the build tree
init	Initialize the build
jar	Build all Java archives (JARs)

All is the default target. If no argument is given to the Ant program, it uses the target to build the complete distribution tree and all class files. It then creates the jar files from the `classes` directory by using specific filters. To produce a stripped-down version of the application, individual plugins can be removed from the compile step. Note that you should also remove the plugin's dependencies from the `jars` directory or alternatively add an exclusion rule to the respective `unjar` command in the build file. The current command includes all jar files, adding approximately 3M to the resulting jar file (the initial jar

size is ~400K):

```
<unjar dest="${build.home}/classes">
  <fileset dir="${basedir}/lib/" includes="*.jar" />
</unjar>
```

Additionally, the built-in library `default.xml` and the default settings `default.ini` can be removed to reduce the size of the jar files.

To build all jar files, enter the following command in the top-level directory:

```
$ ant
...
BUILD SUCCESSFUL
Total time: 18 seconds
```

This will execute the default target (all) and display a series of output messages on the screen. The last two lines should look similar to the above. To verify the jar files enter the following commands:

```
$ java -jar lib/jgrapheditor.jar
JGraphEditor (v6.0.0.0)
$ java -jar lib/jgraphmicropad.jar -V
JGraphpad (v6.0.0.0)
JGraph (v5.6.1.0)
$ java -jar lib/jgraphpad.jar -V
JGraphpad (v6.0.0.0)
JGraph (v5.6.1.0)
```

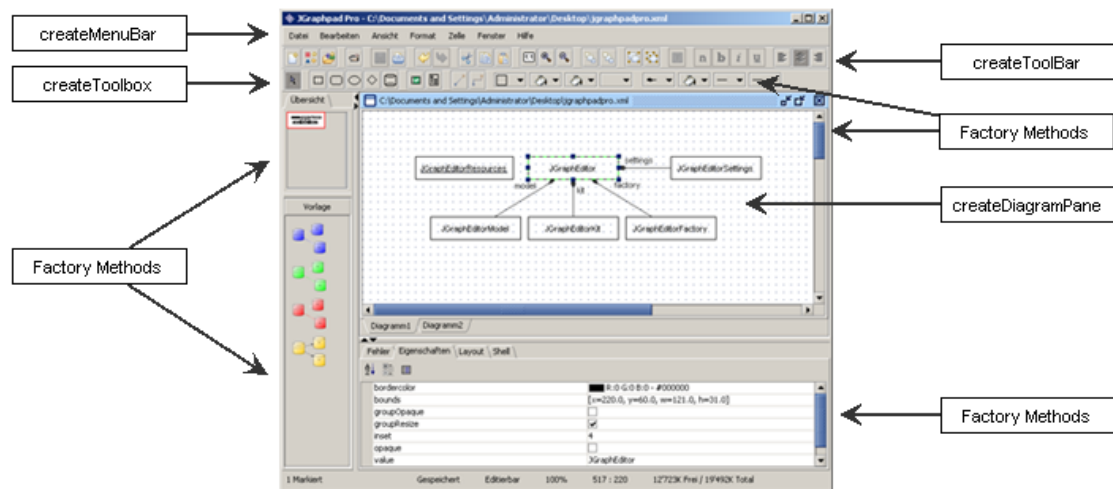
This will print the version numbers of the framework, application and JGraph component. Note the `-V` flag to print the version number of the application. Use the `-?` or `-help` flag to display usage information, or no flag to launch the application.

---

# Chapter 2. JGraph Editor Framework

## 1. Introduction

Figure 2.1. UI Factory



The Editor Framework defines the structure and lifecycle of applications. It is not a UI- or rich-client framework. It simply defines an architecture for dealing with actions and tools and create a user interface with them. The design is inspired by [bib-Davidson], and may be summarized as follows:

- **Externalize the user interface** — The UI is created using a factory based on external files. This idea is extended to use *factory methods* in the Editor Framework.
- **Separate structure and content** — The separation between these two yields separate configuration files, namely XML files for describing the tree-like UI-structure, and resources bundles (aka. properties files) for defining language-dependent resources.

Furthermore, the Editor Framework uses an MVC design-pattern, which leads to the following classes, all residing in the `com.jgraph.editor` package, except for the main `JGraphEditor` class, which resides in the top-level package:

- `com.jgraph.JGraphEditor`: The main class which is a composition of the functionality from the `com.jgraph.editor` package. This class can be considered "glue-code" to hold the other classes together.
- `com.jgraph.editor.JGraphEditorModel`: Following the MVC design pattern, the document model contains the actual application data, namely files, and is in charge of input and output.
- `com.jgraph.editor.JGraphEditorKit`: Holds actions and tools and updates their state depending on the state of the application. The actions and tools in an editor kit are used to define the functionality and behaviour of applications.

- `com.jgraph.editor.JGraphEditorFactory`: Creates and wires up the user interface according to the MVC design pattern. The user interface is created based on the configuration data and uses either built-in or *factory methods*.
- `com.jgraph.editor.JGraphEditorSettings`: Holds the structural definition of the user interface and general dynamic configuration data. This may be modified by various parties during application startup.
- `com.jgraph.editor.JGraphEditorResources`: Static class to hold internationalized resources and general application resources, such as images. All resource bundles must be added before the UI is created.

Apart from these classes the `com.jgraph.editor.JGraphEditorPlugin` is used to define external functionality, but is not actively used in the framework and its use is optional.

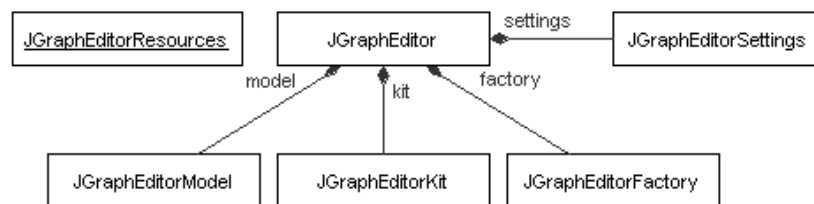
The lifecycle of the editor normally has the following phases:

1. Create settings and add resource bundles, actions, tools and factory methods to the resources, kit and factory respectively.
2. Initialize external plugins to add or modify the settings, and add or replace existing resources, actions, tool and factory methods.
3. Create the user interface using the configured settings, the functionality defined by the editor kit and factory methods.
4. Run the application.
5. Exit the application by calling the exit method in the editor instance.

Note that the exit method is the only hook offered by the `JGraphEditor` class, and it is implemented to call the shutdown method in the `JGraphEditorSettings` class to give applications a chance to save their settings.

## 2. Editor

**Figure 2.2. Editor Participants**



The `JGraphEditor` class holds the main participants. That is, it has a reference to an editor kit, a document model, factory and settings. The resources are static so they may be used from everywhere. Apart from holding these elements together, the `JGraphEditor` class defines one important hook, namely the exit method, which is offered to subclassers to implement and call from within their actions.



## 2.1. Exit Hook

Having said that, it must be pointed out that this approach is optional and you can use other means to exit an application. However, it is often the case that one only has an editor instance at hand, and even if everything else is application dependent, all applications will want to exit in one or another way.

It is important to keep in mind that the base implementation of this method is not empty. Therefore, subclasses must call the superclass implementation, or if you use your own way of terminating your program you should make sure that the settings are given a chance to execute the shutdown hooks before the VM exits, using the following statement:

```
settings.shutdown();
```

A typical subclasser terminates the VM in the appropriate way for how the program was started. For example, in an Applet, you can create an inner class that calls the exit method based on the `isActive` state of the applet:

```
new JGraphEditor() {
    public void exit(int code) {
        super.exit(code);
        if (!isActive())
            System.exit(code);
    }
}
```

The above code avoids calling `System.exit` if the applet is active. In the case of the applet, it is assumed that the exit hook is called from within a listener that is processing an exit event from the enclosing browser.

## 3. Document Model

The `JGraphEditorModel` class is a tree model that provides a great deal of flexibility for modelling the documents in the editor. The editor provides two interfaces for such elements, both contained in the `com.jgraph.editor` package:

- `com.jgraph.editor.JGraphEditorFile`: Defines the basic requirements for a file. A file consists of its children in the document model tree, and is defined by the filename and the `isModified` and `isNew` flags.
- `com.jgraph.editor.JGraphEditorDiagram`: Defines the basic requirements for a diagram in a document model. A diagram is an object that has a name and contains a graph layout cache that may be used to create a `JGraph` instance.

These two interfaces define the known elements of a document model. It is possible to implement both interfaces in the same class or in separate classes. When using separate classes, the editor assumes that a diagram always has a parent file or is itself a file. This setup allows for a wide range of applications and introduction of new element types to the document model.

As with all other models in Swing, the developer must make sure to use the model's methods to change the state of the elements. This is because in Swing, the model is in charge of notifying the listeners of changes, the individual elements do not provide such notification (because it would result in a large number of listeners and overhead for book-keeping). Therefore, to update the modified state of a file the

following helper method should be used:

```
model.setModified(file, false);
```

## 3.1. Input and Output

Aside from defining the document model for the application, the editor's document model is also responsible for input and output. Input/output is implemented using the XML encoding and decoding that is built into Java. The document model provides the *persistence delegates* required to encode its elements. Persistence delegates are a well-defined concept in Java and do not require further explanation. More information about XML encoding may be found in [bib-Winchester]. (Note that the encoding used in previous versions of JGraphpad is no longer used. The GXL plugin is used to convert old files to the new file format.) The model offers the respective helper methods to simplify the use of input and output.

For example, to add a local file or URL as a root element into the document tree, taking into account the file ending to perform decompression and notifying the listeners of the change, you use the following single line of code:

```
model.addFile(filename);
```

The above will use a standard XMLDecoder to read the file contents. In order to save a file, the model configures the respective XMLEncoder using the registered persistence delegates. Therefore, for saving files one must always use the model's method as in the following example:

```
OutputStream out = model.getOutputStream(filename);
model.writeObject(file, out);
out.flush();
out.close();
```

In the above example the support for URLs is limited to creating a ByteArrayOutputStream that holds the data, the developer is in charge of uploading this data to the respective URL as this is highly application dependent. (The methods for posting data to URLs are located in the JGraphpadFileAction class.)

### 3.1.1. Persistence Delegates

In the static initializer of the default model all cell view properties except *cell* and *attributes* of the known cell views, namely VertexView, PortView and EdgeView are made transient. Additionally, a set of persistence delegates is added for the DefaultGraphModel, GraphLayoutCache and DefaultEdge.DefaultRouting classes in the JGraphEditorModel constructor. The first two use a DefaultPersistenceDelegate, which defines the properties to be written to the file and passed to the constructor when read back into memory:

```
addPersistenceDelegate(DefaultGraphModel.class,
    new DefaultPersistenceDelegate(new String[] { "roots",
        "attributes" }));
addPersistenceDelegate(GraphLayoutCache.class,
    new DefaultPersistenceDelegate(new String[] { "model",
        "factory", "cellViews", "hiddenCellViews", "partial" }));
```

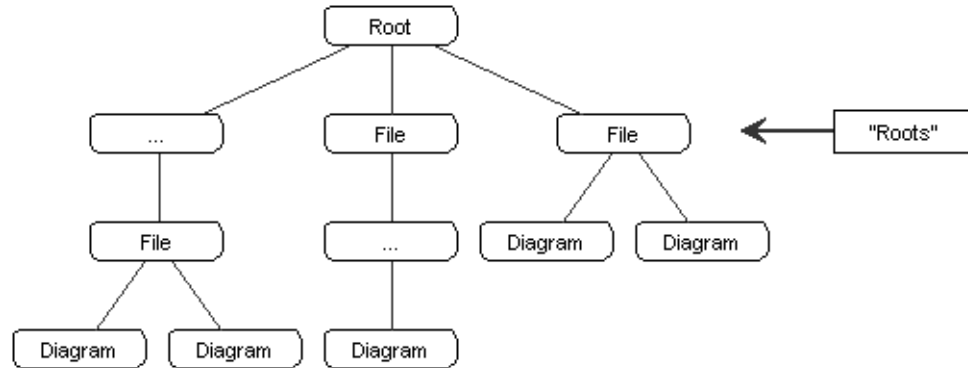
Thus, the graph model is defined by the roots and the model's attributes, and the layout cache is defined by the model, factory, partial state and the cell views (and hidden cell views) it contains.

The DefaultEdge.DefaultRouting persistence delegate makes sure the decoder invokes the GraphConstants.getDEFAULT\_ROUTING method to get an instance of the class:

```
addPersistenceDelegate(DefaultEdge.DefaultRouting.class,
    new PersistenceDelegate() {
        protected Expression instantiate(Object oldInstance,
            Encoder out) {
            return new Expression(oldInstance,
                GraphConstants.class, "getROUTING_SIMPLE", null);
        }
    }
);
```

## 3.2. Files and Diagrams

**Figure 2.3. Document Model**



The default model makes no restrictions as to what must be put into the tree and on which level. Also, it does not impose any limitations on the nesting of the tree elements. However, some default behaviour has been built into the document model. For example, when using the addFile method the JGraphEditor-File is added using the addRoot method (the children of the root node are called "roots" in the document model), thus, it assumes that files are to be inserted as roots. (This is only assumed in this one helper method for simplicity. Programmatically, files can be added at any level of the document tree using the addChild method.)

Another assumption is that for each element in the document model, there exists at least one file along the path to the root. This is implemented by the static getParentFile method, which returns the first parent. The method is used, for example, to update the modified state of the parent file when a child diagram changes. In the application, the document model is assumed to contain a set of JGraphEditorFiles, each of which corresponds to exactly one physical file. The JGraphEditorFile can contain any number of children, for example JGraphEditorDiagram objects, or it can itself be a JGraphEditorDiagram (in which case it is considered its own parent file).

For example, the JGraphpad application offers two types of files: *documents* and *libraries*. Since documents can contain multiple diagrams, a nested structure is used to represent the documents in the tree model. JGraphpadFile implements the JGraphEditorFile interface and holds the JGraphpadDiagrams, which implement the JGraphEditorDiagram interface. The JGraphpadLibrary instead requires only one diagram, thus, it implements both, the JGraphEditorFile and JGraphEditorDiagram interface in one class.

## 4. Editor Kit

The `JGraphEditorKit` class defines the actual functionality of an application by defining its actions and tools. The actions and tools must be added to the kit at startup time, and are defined by the following base classes, both contained in the `com.jgraph.editor` package:

- `com.jgraph.editor.JGraphEditorAction`: May be inserted into buttons and menu items to provide the functionality that is accessible via the user interface (eg. menubar, toolbar, popup menus etc).
- `com.jgraph.editor.JGraphEditorTool`: May be used to implement special *marquee handlers* to insert specific graph cells or implement other functionality, such as *panning* or *marquee selection* (default) in a graph.

The registration of actions and tools ensures that various parties can add or replace functionality during the startup phase. In the following example, the `JGraphEditorAction.Bundle` interface was used to implement `AllActions` of the file menu, which is a way of grouping a set of actions and adding them using `addBundle`:

```
kit.addBundle(new JGraphpadFileAction.AllActions());
```

The actions and tools defined in the editor kit are referenced by name, and are used by the factory in order to be placed in the respective UI elements using the `getAction(String)` and `getTool(String)` methods of the kit.

This mechanism may be used by the main application and all plugins likewise: A plugin may add actions and tools to the kit and merge its UI configuration into the existing configuration, so that the new functionality is subsequently placed in the user interface by the factory. Note that it is possible for a plugin to replace registered actions or tools with new ones. To avoid this, you must either be the last to add the actions and tools or subclass the `JGraphEditorKit` and make sure the respective keys are only added once, as in the following example for actions:

```
new JGraphEditorKit() {  
    public Object addAction(JGraphEditorAction action) {  
        if (getAction(action.getName()) == null)  
            return super.addAction(action);  
        return null;  
    }  
}
```

In order to update the states of the actions contained in an editor kit, the update method must be called. That is, the editor kit does not itself register any listeners, it only provides the update method to be called from within listeners. The update method in turn calls update on all registered bundles. Thus, in order to update the states of all actions in all registered bundles in a kit when the permanent focus owner changes, the following code may be used:

```
KeyboardFocusManager focusManager = KeyboardFocusManager  
    .getCurrentKeyboardFocusManager();  
focusManager.addPropertyChangeListener(new PropertyChangeListener() {  
    public void propertyChange(PropertyChangeEvent e) {  
        if (e.getPropertyName().equals("permanentFocusOwner"))  
            kit.update();  
    }  
})
```

```
    }  
  });  
}
```

## 4.1. Actions

The `JGraphEditorAction` class defines the abstract class for all actions in the framework. It adds three properties to the `AbstractAction` superclass, namely `isToggleAction`, `isSelected` and `isVisible`. The `isToggleAction` property is used to determine whether a toggleable user interface component should be created in the factory. The other two properties are *bound* properties, which means a `PropertyEvent` is fired when they are changed. The factory creates a listener for these properties (and also for the `isEnabled` property) to update the state of the user interface components it creates.

The Editor Framework does itself not implement any actions. However, the `JGraphpad` application implements quite a few actions and uses a special structure to implement them:

All action names are defined as static variables to be passed to the constructor:

```
public class JGraphpadHelpAction extends JGraphEditorAction {  
  
    public static final String NAME_ABOUT = "about";  
  
    public JGraphpadHelpAction(String name) {  
        super(name);  
    }  
}
```

While it is not required by the `Bundle` interface for all its actions to be implemented in one class, it has been proved good practice to do so, as it will lead to a smaller number of action files with less redundancy. On the other hand, it requires the `actionPerformed` method to do an additional check on the action name to find the correct implementation:

```
public void actionPerformed(ActionEvent event) {  
    if (getName().equals(NAME_ABOUT))  
        doAbout(getPermanentFocusOwner());  
}
```

The above calls a protected method of the class which displays the *About* dialog. For the sake of this example we assume that the displaying of the dialog requires a `JGraph` component to be focused, so the `doAbout` method would have to check for that. What's more important, is that the state of the action needs to be updated when the permanent focus owner changes.

After having added the `PropertyChangeListener` to the keyboard focus manager in Section 4, “Editor Kit”, it can be safely assumed that the update method of all action bundles will be called when the permanent focus owner changes. So the action must be added to a bundle, which is implemented as an inner class:

```
public static class AllActions implements Bundle {  
  
    public JGraphEditorAction actionAbout =  
        new JGraphpadHelpAction(NAME_ABOUT);  
}
```

The `Bundle` interface offers two methods: `getActions` to return all actions and `update` to change the state of the actions based on, in this case, the permanent focus owner:

```
public JGraphEditorAction[] getActions() {
    return new JGraphEditorAction[] { actionAbout };
}

public void update() {
    actionAbout.setEnabled(
        getPermanentFocusOwner() instanceof JGraph);
}
}
```

### 4.1.1. Action Resources

The factory uses the following suffixes to get resources for action user interface components:

**Table 2.1. Action Resources supported by the JGraphEditorFactory class**

Suffix	Description
.label	The label of the element to be created eg. About
.icon	The icon resource to be used, eg. /com/jgraph/pad/images/about.gif
.shortcut	The shortcut to be assigned to the action, eg. control shift A
.tooltip	The tooltip to be associated with the element, eg. About JGraphpad Pro
.mnemonic	The mnemonic to be used for the menu item this action is placed in

The suffixes are to be appended to the action name in the resource files. Thus, for the about action from above, the resources would look like this in a properties file:

```
about.label=About
about.icon=/com/jgraph/pad/image/about.gif
about.shortcut=control shift A
about.tooltip=About JGraphpad Pro
about.mnemonic=a
```

## 4.2. Tools

The JGraphEditorTool class implements a basic tool in the framework. A tool is a *basic marquee handler* with a name and a `isAlwaysActive` property. To understand the basics of editor tools it is important to understand what the BasicMarqueeHandler is used for in JGraph. Technically speaking, the BasicMarqueeHandler is a paintable mouse listener base class with an added `isForceMarqueeEvent` method. The real magic behind the marquee handler comes from the way it is used in the BasicGraphUI. The class in charge is the MouseHandler inner method in the BasicGraphUI class. Its `mousePressed` method has the following basic logic:

```
if (!event.isConsumed() && graph.isEnabled()) {
    [...]
    if (marquee.isForceMarqueeEvent(event)) {
        marquee.mousePressed(event);
        [...]
    } else {
        [...]
    }
}
```

```
}
```

It follows that the marquee handler's `isForceMarqueeEvent` method is used to fetch control for this and the subsequent events for the *gesture*. Since the internal mouse handler of the `BasicGraphUI` is usually added as the first mouse listener to the enclosing `JGraph` component, this is the first method to be called in the event dispatching loop. Therefore, the marquee handler, and hence the editor tools, can be seen as *event interceptors* to fetch control of mouse gestures on a graph.

The default implementation provides no special behaviour. It inherits all functionality from the super-class and may therefore be used as a basic select tool, which may be added to an editor kit as follows:

```
kit.addTool(new JGraphEditorTool("selectTool"));
```

## 4.2.1. Tool Resources

The factory uses the following suffixes to get resources for tool user interface components:

**Table 2.2. Tool Resources supported by the `JGraphEditorFactory` class**

Suffix	Description
<code>.icon</code>	The icon resource to be used, eg. <code>/com/jgraph/pad/images/select.gif</code>
<code>.tooltip</code>	The tooltip to be associated with the element, eg. <code>Marquee Tool</code>

The suffixes are to be appended to the tool name in the resource files. Thus, for the select tool from above, the resources would look like this in a properties file:

```
selectTool.icon=/com/jgraph/pad/image/select.gif  
selectTool.tooltip=Marquee Selection
```

## 5. UI Factory

The `JGraphEditorFactory` class uses the classes from the `com.jgraph.editor.factory` package and the user interface description in the `JGraphEditorSettings` class to create the user interface. It does this by creating known elements, such as menu items, toolbar buttons and toolboxes and placing actions and tools from the editor kit inside them. To create new elements without subclassing, the factory allows to register *factory methods*. Factory methods are objects that implement the `JGraphEditorFactoryMethod` interface, which offers one single method to create new `Component` instances:

```
public abstract Component createInstance(Node configuration);
```

The method takes a document node from a configuration file as an optional parameter. The configuration is passed in from the factory when the method is executed, which is implemented by the following code in `JGraphEditorFactory`:

```
public Component executeMethod(String factoryMethod,  
    Node configuration)  
{  
    JGraphEditorFactoryMethod method = getMethod(factoryMethod);
```

```
if (method != null)
    return method.createInstance(configuration);
return null;
}
```

getMethod looks up the factory method for the given name in a hashtable, and returns it. Note that the factory always tries to first execute a factory method for creating a named item in a menubar, toolbar or toolbox, thus ensuring that factory methods have precedence over actions and tools in the kit. (Again, plugins may replace existing factory methods which may need to be blocked as shown in Section 4, "Editor Kit".)

By subclassing the factory or registering a factory method, support for other containers than toolbars, menubars and toolboxes can be added. To create the default containers, the factory offers a series of create and configure methods. These methods may be overridden to alter the returned objects, or may be "overloaded" with a factory method.

## 5.1. Factory Methods

Factory methods implement the JGraphEditorFactoryMethod interface. They are used to create elements of the user interface, either by invoking them explicitly or implicitly by using the name of the factory method as a key in a user interface description, as in the following example:

```
<menubar>
  <item key="createOpenRecentMenu" />
</menubar>
```

As an example of an explicit call, consider the following to execute the factory method with the name createFrame via the editor factory:

```
Component c = factory.executeFactoryMethod("createFrame");
```

The framework provides implementations of factory methods in the *navigator* and *combo box* components, however, these factory methods are not added to the factory by default. This can be done using the following code:

```
factory.addMethod(new JGraphEditorNavigator
    .FactoryMethod(editor));
factory.addMethod(new JGraphEditorComboBox
    .BorderComboFactoryMethod());
factory.addMethod(new JGraphEditorComboBox
    .LineDecorationComboFactoryMethod());
factory.addMethod(new JGraphEditorComboBox
    .LineWidthComboFactoryMethod());
factory.addMethod(new JGraphEditorComboBox
    .DashPatternComboFactoryMethod());
```

Note that the navigator's factory method takes an editor as an argument to its constructor. This is normally the case if the factory method requires access to parts of the editor, such as the model or the factory, as in the above case. There are various possible reasons why a factory method could use an editor. In the case of the navigator, the createInstance method looks as follows:

```
public Component createInstance(Node configuration) {
```



```
JGraph backingGraph = editor.getFactory().createGraph(
    new GraphLayoutCache());
backingGraph.setEnabled(false);
backingGraph.setFocusable(false);
return new JGraphEditorNavigator(backingGraph);
}
```

Thus, the method requires an editor variable to be set in the constructor to create the backing graph instance. (Note that in the above example the graph layout cache will be replaced with the focused graph layout cache. The configuration parameter is ignored.)

### 5.1.1. Connecting Factory Methods and Actions

Under certain circumstances it is required to connect actions to objects created during the execution of `createInstance` in a factory method. In such cases it is important to keep in mind that *all* actions and factory methods are to be registered prior to their execution. Thus, at the time the factory method is executed the action is already registered in the editor kit, and may be retrieved from there.

An example of such a case is the layout action in the layout plugin, which calls the `actionPerformed` method of the execute button in the layout panel. The action provides a setter method to set the layout panel, which is to be called from the corresponding factory method's `createInstance` method (see `JGraphpadLayoutPanel.FactoryMethod`):

```
JGraphpadLayoutPanel panel = new JGraphpadLayoutPanel(editor);
JGraphEditorAction action = editor.getKit().getAction("layout");
if (action instanceof JGraphpadLayoutAction)
    ((JGraphpadLayoutAction) action).setLayoutPanel(panel);
```

In the above code, the action is fetched from the kit by name, and the newly created layout panel is assigned to it.

### 5.1.2. Connecting Factory Methods

When connecting factory methods the basic problem is that the actual instances returned by the `createInstance` method are not stored for later use. Furthermore, in the general case, the execution order of the factory methods is undefined. Thus, an application-wide map should be used to store the relevant objects in each `createInstance` method for later wiring by the application code, usually after the user interface has been created.

An example of this case is the wiring between the desktop pane and the window menu. Both are created using a factory method. In the case of the window menu, the factory method is rather simple, as it only creates, registers and returns the instance (see `JGraphpadWindowMenu.FactoryMethod`):

```
public Component createInstance(Node configuration) {
    JMenu menu = new JGraphpadWindowMenu();
    editor.getSettings().putObject("windowMenu", menu);
    return menu;
}
```

As you can see in the above code, the `JGraphEditorSettings` object referenced from the editor instance is used as the application-wide map to store the menu instance.

The second component to be registered is created in the `JGraphpadPane` constructor, which is invoked from the `createInstance` method of the pane's factory method. In this case, the registration of the component takes place in the constructor of the `JGraphpadPane`, which is being passed in an editor:

```
public JGraphpadPane(JGraphEditor editor) {  
    [...]   
    editor.getSettings().putObject("desktopPane", desktopPane);  
}
```

In the code that creates the user interface, after having created the complete user interface, the components are wired up:

```
JDesktopPane desktopPane = (JDesktopPane) editor.getSettings()  
    .getObject("desktopPane");  
JGraphpadWindowMenu windowMenu = (JGraphpadWindowMenu) editor  
    .getSettings().getObject("windowMenu");  
if (desktopPane != null && windowMenu != null)  
    windowMenu.setDesktopPane(desktopPane);
```

Note that the actual installation of required listeners is done in the `setDesktopPane` of the *master* component (in this case the menu), the above code is only used to make sure both instances exist and call the respective methods.

## 6. Settings

Apart from acting as an application-wide map, the `JGraphEditorSettings` class is used to hold a set of configuration files, namely XML documents and ini files, and provides a set of methods to access and modify the data stored in these files. The application-wide map is used to implement a variety of functionality. The settings class lifecycle has three phases which are defined by the framework:

- **Setup** — In the setup phase, objects and files are added or merged into the settings.
- **Runtime** — Use the configuration data, and the map object to hold current settings.
- **Shutdown** — Triggered by the editor's exit hook, in this phase the settings are saved.

The settings class is best explained by looking at its actual use in an application. In `JGraphpad`, the settings class is used to describe the user interface and also to hold the user settings and store objects created by factory methods, among others. The description of the user interface uses XML files from the classpath, whereas the user settings are stored in an ini file in the user's home directory. For adding the XML file, the file must first be parsed and the resulting Document then added under a specified key:

```
settings.add("ui", JGraphEditorSettings.parse(  
    JGraphEditorResources.getInputStream("/com/jgraph/pad/resources/ui.xml")));
```

The above code parses the XML file found at `/com/jgraph/pad/resources/ui.xml` in the classpath and adds it to the settings under the `ui` key. To later refer to this document, the following code is used:

```
settings.getDocument("ui");
```

Thus, the `createFrame` example from above can be rewritten to pass the UI configuration along and cast the return value as follows:

```
Window wnd = (Window) factory.executeMethod("createFrame",
    settings.getDocument("ui").getDocumentElement());
```

To add ini files, no parsing is required:

```
settings.add("userSettings", JGraphEditorResources
    .getInputStream("/home/johndoe/.editor.ini"));
```

Note that the `JGraphEditorResources.getInputStream` method is used for retrieving files from both, the classpath and from the filesystem. This is a fallback mechanism, which first tries to open the stream from the classpath and, if that fails, tries to open the stream as an external file or URL.

## 6.1. Merging UI Configuration Files

XML configuration files are used in the framework to describe the user interface description. The fact that the XML file is translated into a Document Object Model (DOM) in memory is used to implement a simple "merging" mechanism for configuration files, which may be used to extend the user interface definition.

For example, to create a simple menu, the following configuration file can be added to the settings:

```
<menubar>
  <menu key="fileMenu">
    <item key="exit"/>
  </menu>
</menubar>
```

If no document has been added for the specified name, the document is added to the settings as is. If another document has been added for the same name, then the document is merged into the existing one. For example, consider the following file is being added under the same name as the above:

```
<menubar>
  <menu key="fileMenu">
    <item key="newDiagram" before="exit"/>
  </menu>
</menubar>
```

Then the resulting in-memory DOM looks like this:

```
<menubar>
  <menu key="fileMenu">
    <item key="newDiagram"/>
    <item key="exit"/>
  </menu>
</menubar>
```

## 6.2. Application-wide Map

The application-wide map serves various purposes. One example we have already come across is the storing of objects created during the execution of the `createInstance` method in a factory method for later

wiring by the application. Another example is the storing of objects for later use in a shutdown hook, for example by the caller of a factory method.

An example use for this case is in the `createApplication` method of the `JGraphpad` class:

```
Window mainWindow = createMainWindow(editor, "createFrame");
editor.getSettings().putObject("mainWindow", mainWindow);
editor.getSettings().restoreWindow("userSettings", "mainWindow");
```

The last line in the above example shows how to use the built-in `restoreWindow` method to restore the bounds of the window from a properties file.

## 6.3. Shutdown Hooks

The settings class offers a mechanism to add code that should be executed before application shutdown. For this purpose, it provides the `ShutdownHook` inner interface, which simply provides a shutdown method with no parameters. Shutdown hooks may be added using the `addShutdownHook` method:

```
editor.getSettings().addShutdownHook(
    new JGraphEditorSettings.ShutdownHook() {
        public void shutdown() {
            editor.getSettings().storeWindow("userSettings", "mainWindow");
        }
    }
);
```

The above example will execute the following steps before application shutdown:

1. Get the Window stored under the `mainWindow` key from the key
2. Get the properties stored under the `usetSettings` name
3. Store the bounds in the properties under the `mainWindow` key

Note that the `Properties` object in Java is used to represent ini files, while the `ResourceBundle` object is used to represent properties files.

## 7. Resources

Internationalized resources and resource files (such as images) are accessed through the `JGraphEditorResources` class. The class models a set of resource bundles, which in turn contain a set of files, namely one for each supported language. Resource files are usually shipped with the application and are to be found on the classpath. (Resource bundles are part of Java and are not further explained here.)

Using the `JGraphEditorResources` typically comprises of two steps: Adding resource bundles at startup and using resources (retrieving strings, images and streams) at runtime. To add a resource bundle, the following code is used:

```
JGraphEditorResources.addBundle("com.jgraph.pad.resources.actions");
```

This will add all files belonging to the *actions* resource bundle in the `src/com/jgraph/pad/resources` directory (the actual string passed to the `addBundle` method is referred to as the *basename* of the resource bundle). For example, if an English and German resource file exists, this will add one of the following files, based on the current system locale: `actions.properties` or `actions_de.properties`. (Alternatively one can make the German file the default, in which case the English filename would be `actions_en.properties` and the german file would not have a language extension, ie. only contain the *basename* and the *properties* extension.)

To get resources and resource files from the resource bundles, the class offers a number of static methods:

- `getString`: Returns the resource for the specified key and replaces the placeholders with the corresponding values
- `getImage`: Tries to load the specified image using the classloader (no resource bundles used)
- `getInputStream`: Tries to open the specified filename using the classloader (no resource bundles used)

The `getString`-family of methods is best described with a small example: Assume the resource bundle contains the following key-assignment:

```
save.dialog=Save changes to {0}?
```

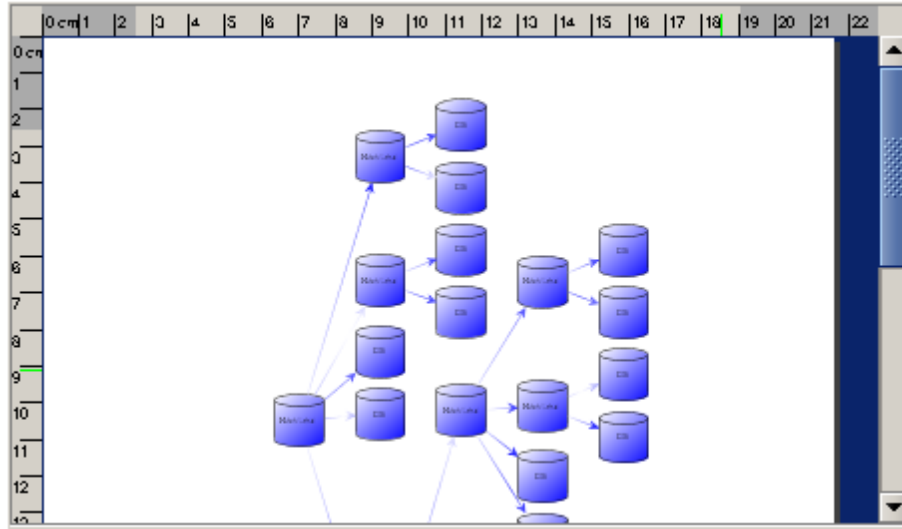
This is a resource string with a placeholder. When using the `getString(String, String[])` method, the placeholder `{0}` will be replaced with the respective value in the `String`-array (at the corresponding index 0). In order to use this in a program to construct a string, the following code would be used:

```
JGraphEditorResources.getString("save.dialog", filename);
```

Note that the `getString(String, String)` method is a shortcut for only one placeholder, and the `getString(String)` method will not replace any placeholders. Also note that the order in which resource bundles are added is significant as existing key-assignments will be overridden with new assignments for the same key.

## 8. Diagram Pane

**Figure 2.4. JGraphEditorDiagramPane**



The `JGraphEditorDiagramPane` is an extension of `JScrollPane` that acts as a wrapper for a `JGraph` component. The diagram pane provides the following add-on functionality to a `JGraph`:

- **Undo Support** — Creates an undo manager for the contained graph
- **Backgrounds** — Paints background colors, images and the page background (aka. layout view)
- **Autoscaling** — Automatic zooming of the contained graph according to a scaling policy
- **Printing** — Implements the `Printable` interface for printing the graph
- **Rulers** — Acts as a scrollpane with optional rulers

The diagram pane is the only component that provides a reference to a `JGraphEditorDiagram`, and is therefore the preferred way of finding a diagram in a component hierarchy, as implemented by the `getPermanentFocusOwnerDiagram` method in the `JGraphEditorAction` class:

```
public static JGraphEditorDiagram getPermanentFocusOwnerDiagram() {
    JGraphEditorDiagramPane diagramPane =
        getPermanentFocusOwnerDiagramPane();
    if (diagramPane != null)
        return diagramPane.getDiagram();
    return null;
}
```

Note that the following always holds for diagram panes:

```
getDiagram().getGraphLayoutCache() == getGraph().getGraphLayoutCache()
```

To use the diagram pane, the respective method in the factory must be invoked with a diagram object as a parameter. An example of such use is inside a tree model listener, which is installed in the document model to wait for diagrams to be inserted:

```
public void treeNodesInserted(TreeModelEvent event) {
    TreeModel source = (TreeModel) arg0.getSource();
    Object[] children = arg0.getChildren();
    for (int i = 0; i < children.length; i++) {
        if (children[i] instanceof JGraphEditorDiagram)
            add(editor.getFactory().createDiagramPane(
                (JGraphEditorDiagram) diagram));
    }
}
```

Note that the above code does not take into account that the event describes *subtrees*, and the inserted nodes must be traversed recursively to find the inserted diagrams. The important part is that the diagram pane is created with a single method call. The functionality is meant to be working "out of the box".

## 8.1. Undo Support

The diagram pane constructs a new `GraphUndoManager` and adds it to the graph at construction time. (The default history size is 100 and is defined in `javax.swing.undo.UndoManager`.) To implement the respective undo and redo actions, the undo manager in the diagram pane can be used as follows:

```
GraphUndoManager manager = diagramPane.getGraphUndoManager();
GraphLayoutCache cache = diagramPane.getGraph().getGraphLayoutCache();
if (manager.canUndo(cache))
    manager.undo(cache);
```

For redo, the last two lines must be replaced with the following:

```
if (manager.canRedo(cache))
    manager.redo(cache);
```

## 8.2. Backgrounds

The painting of the background is implemented in a custom viewport within the diagram pane. To control this painting, the diagram pane offers the `setBackgroundImage`, `setBackground` and `setPageVisible` methods. The `setBackground` and `setPageVisible` take a `Color` and `boolean` argument respectively. The painting of the background page may be further controlled using the `setPageFormat` and `setPageScale` methods, which specify the page format and scale to be used to render the background page to the screen.

The `setBackgroundImage` method takes an `ImageIcon`, which is not encoded by the default XML encoder. To come across this problem, JGraphpad provides the `JGraphpadImageIcon` class for storing images as external references (not binary data) in XML files. Thus, to set the diagram background image in JGraphpad, the following code is used:

```
diagramPane.setBackgroundImage(
    new JGraphpadImageIcon(filename));
diagramPane.repaint();
```

## 8.3. Autoscaling

Autoscaling is implemented using various predefined policies, namely:

**Table 2.3. Autoscale policies supported by the JGraphEditorDiagramPane class**

Policy	Description
AUTOSCALE_POLICY_NONE	No autoscaling
AUTOSCALE_POLICY_WINDOW	All cells fit the window
AUTOSCALE_POLICY_PAGE	The page fits the window
AUTOSCALE_POLICY_PAGEWIDTH	The page width fits the window

The autoscale policy may be changed using the `setAutoScalePolicy` method. Note that the diagram pane installs a listener in the graph to automatically reset the policy if the scale is changed by other means than autoscaling.

## 8.4. Printing

To implement printing in Java, the `Printable` interface must be implemented. The interface offers the `print` method which may be used to print a diagram as follows:

```
PrinterJob printJob = PrinterJob.getPrinterJob();
printJob.setPrintable(diagramPane);
if (printJob.printDialog())
    printJob.print();
```

The default `print` method in the diagram pane supports printing diagrams across multiple pages.

## 8.5. Rulers

To implement the `setRulersVisible` method, a `JGraphEditorRuler` object is added to or removed from the column and row header of the scrollpane on the fly. The `setMetric` method further controls the units displayed by these rulers.

# 9. UI Components

All remaining classes of the framework reside in the `com.jgraph.editor.factory` package. For the `JGraphDiagramPane` (which uses the `JGraphEditorRuler`) and the `JGraphEditorToolbox` class the factory offers "built-in" methods to create them:

```
public JGraphEditorDiagramPane createDiagramPane(
    JGraphEditorDiagram diagram);
public JGraphEditorToolbox createToolbox(Node configuration);
```

The other two classes, the `JGraphEditorComboBox` and `JGraphEditorNavigator` provide factory methods which may be added to a factory instance using the code from the example above (see Section 5.1, "Factory Methods").

## 9.1. Toolbox

**Figure 2.5. JGraphEditorToolbox**





The `JGraphEditorToolbox` component is a subclass of the `JToolBar` class. In contrast to a toolbar, the elements used in the toolbox are not actions, they are tools. For each tool in the toolbox, the factory creates a toggle button and updates the selection tool of the toolbox when the toggle button is pressed. Since the toolbox installs the selection tool as a marquee handler in the current `JGraph`, there must also be a way to set that current graph on a toolbox.

The toolbox itself does not install any listeners to update the current graph. It only offers the `setGraph` method for this purpose. An example use of the `setGraph` method, where the current graph in the toolbox is updated using a helper class, which triggers a `PropertyChangeEvent` if the focused graph changes is found in `JGraphpadPane`:

```
JGraphpadFocusManager.getCurrentGraphFocusManager()
    .addPropertyChangeListener(new PropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent e) {
            if (e.getPropertyName()
                .equals(JGraphpadFocusManager.FOCUSED_GRAPH_PROPERTY)
                && e.getNewValue() instanceof JGraph)
            {
                toolbox.setGraph((JGraph) e.getNewValue());
            }
        }
    });
```

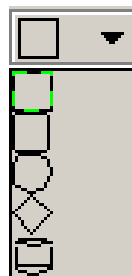
The toolbox uses an inner class to redirect events to either the current selection tool or the previous marquee handler of the graph based on the return values of the `isForceMarqueeEvent` methods of the two. If the previous marquee handler's `isForceMarquee` event returns true, then the events are redirected to the previous handler instead of the selection tool.

### 9.1.1. isAlwaysActive

The `isAlwaysActive` flag may be used to set the return value for the tool's `isForceMarqueeEvent` method, however, the previous marquee handler will always have precedence over the selection tool. The `isAlwaysActive` flag is used for example in the edge tool (tool to insert new edges in `JGraphpad`) in order to avoid selection cells to be moved when the mouse drags them, as in this case a new connection should be inserted starting at this vertex.

## 9.2. Combo Box

**Figure 2.6. JGraphEditorComboBox**



The `JGraphEditorComboBox` class implements a special combo box which uses renderers from cell

views to display its elements. A combo box is constructed with an array of maps which contain the attributes and values for the individual entries and a boolean indicating whether to use an edge view or vertex view to render the entries. In Figure 2.6, “JGraphEditorComboBox”, the array was constructed using the VERTEXSHAPE custom attribute with the following code:

```
Map[] attrs = new Hashtable[defaultShapes.length];
for (int i = 0; i < defaultShapes.length; i++) {
    attrs[i] = new Hashtable(2);
    JGraphpadGraphConstants.setVertexShape(attrs[i],
        defaultShapes[i]);
}
```

The defaultShapes are stored in an int[]. If the default cell views provided by the combo box are not suitable, as in the screenshot where a custom renderer was used, then an optional constructor may be used to pass a custom cell view along:

```
AbstractCellView view = new JGraphpadVertexView("");
GraphConstants.setBorderColor(view.getAttributes(), Color.BLACK);
Rectangle2D bounds = new Rectangle2D.Double(0, 0, 14, 14);
GraphConstants.setBounds(view.getAttributes(), bounds);
JGraphEditorComboBox comboBox = new JGraphEditorComboBox(attrs,
    view, false);
```

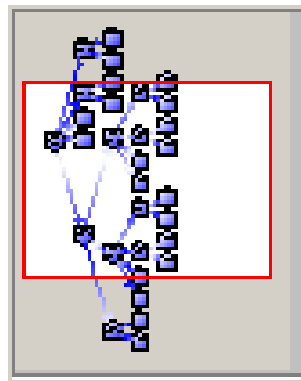
In the above code, a dummy String object is used as the cell for the vertex view.

The combo box requires a backing graph for rendering the entries. However, since the graph is not displayed directly, as in the case of the navigator, the backing graph for the combo box is created using a default JGraph instance, without calling the respective factory method. If this is not suitable then the backing graph of each combo box may be updated using the setBackingGraph method.

Noteworthy are the createVertexView, getVertexViewSize and createEdgeView hooks, which are used to create the default cell views. If a custom cell view must be provided to the combo box, it is possible to override one of the create methods instead of using the optional constructor. The methods can also be extended to set the default attributes used for rendering the combo box entries. (The getVertexViewSize is a special hook for this purpose which allows to return the size of the vertex view to be used.)

## 9.3. Navigator

**Figure 2.7. JGraphEditorNavigator**



The `JGraphEditorNavigator` class implements a birds-eye view using a backing graph. In this case, the backing graph must be passed to the constructor as it is being used directly in the user interface. As shown in Section 5.1, “Factory Methods”, the factory method of the navigator uses the `createGraph` method of the factory to create the backing graph. (In most cases a default `JGraph` instance would be sufficient, because the navigator implements a read-only view with no tooltips, which means most features of `JGraph` are disabled.)

In order to set the graph being displayed, the `setCurrentGraph` method is used. The same technique as for updating the toolbox may be used to update the current graph in the navigator:

```
JGraphpadFocusManager.getCurrentGraphFocusManager();
.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
        String prop = e.getPropertyName();
        if (JGraphpadFocusManager.FOCUSED_GRAPH_PROPERTY
            .equals(prop)
            && e.getNewValue() instanceof JGraph) {
            navigator.setCurrentGraph((JGraph) e
                .getNewValue());
        }
    }
});
```

The navigator provides built-in support for rendering the background image of the diagram pane that the current graph is contained in. This feature may be switched off using the `setBackgroundVisible` switch.

When added to a factory, the navigator's factory method requires an editor instance to be passed in, which is used to create the backing instance passed to the navigator constructor as explained above:

```
addFactoryMethod(new JGraphEditorNavigator.FactoryMethod(editor));
```

## 10. Summary

The JGraph Editor Framework provides a simple document model and defines a way of dealing with actions, tools and add-on methods to construct a user interface. A factory constructs the UI based on external configuration files and resource bundles by using these objects.

In the startup phase, objects are added to the kit and factory, and files are added to the settings and resources. Then, optional plugins are initialized to add additional objects and files. The user interface is then created, typically by calling a sequence of factory methods.

In the shutdown phase, after having displayed all dialogs, the `exit` method of the editor object should be invoked, which in turn calls the `shutdown` method and `-hooks` in the editor settings. Subclassers should make sure to call the superclass when overriding this method.

The framework also provides a toolbox, diagram pane and navigator component. The toolbox displays a set of marquee handlers to be used in a graph. The diagram pane is a scrollpane with add-on functionality, and the navigator provides a birds-eye view for graphs.

---

# Chapter 3. JGraphpad Pro

## 1. Introduction

The JGraphpad Pro application is implemented on top of the framework, the framework with JGraph being its only dependency. The application does not introduce new concepts, it rather applies those from the framework, namely actions and tools to implement the functionality, factory methods to create the user interface and serializable objects to implement the document model. The application implements roughly the same functionality as the old JGraphpad, with the addition of plugins and using a clean codebase with a more elaborate architecture.

The user interface is defined by a set of configuration files in the `com.jgraph.pad.resources` package, and all images have been placed in `com.jgraph.pag.images`. Apart from these packages, the application is made up by the following packages, the JGraphpad and -Applet class residing in the top-level `com.jgraph` package:

- `com.jgraph.pad`: Defines the elements of the document model
- `com.jgraph.pad.action`: Provides actions to be used in the kit
- `com.jgraph.pad.dialog`: Contains a set of customizable dialogs
- `com.jgraph.pad.factory`: Implements factory methods for constructing the UI
- `com.jgraph.pad.graph`: Provides a custom JGraph implementation
- `com.jgraph.pad.tool`: Implements the tools to be added to the kit
- `com.jgraph.pad.util`: Offers various utility classes

As you can see from this list, the `pad`, `action`, `factory` and `tool` packages implement the framework concepts. The `dialog`, `graph` and `util` packages are used to display common dialogs, provide a custom graph and general utility classes, respectively.

The following sections will look at the main classes, startup and shutdown, document model, factory methods and utility classes. The next chapter focuses on the `graph` package. The complete list of actions and tools may be found in the appendix.

## 2. JGraphpad and -Applet

**Figure 3.1. JGraphpadApplet**



The JGraphpad class is the main class of the application, ie. it provides the main method. This makes it suitable for use as an application and with Java Webstart. The JGraphpadApplet class is an applet that only displays a start button, as shown in Figure 3.1, “JGraphpadApplet”. The implementation of the start button's `actionPerformed` method uses the same code as the main method of the JGraphpad class, namely the following line to create a new application:

```
new JGraphpad().createApplication(arguments);
```

The optional *arguments* parameter is used to pass in a hashtable of key/value pairs, which is obtained either from the applet parameters in the `init` method or the command line arguments in the `main` method, depending on how the program was started. (The code is security-aware, which means it handles security exceptions in restricted environments, such as the applet sandbox.)

Despite the relatively large number of lines, the `JGraphpad` class only creates an editor object, starts the application and defines a shutdown mechanism. The many lines of code are a consequence of IOC, which yields that no specialization should be used to deal with external dependencies. Instead, these objects are to be "injected" from within the main class. (This is also the reason why `JGraphpad` does not require to extend the settings, model, kit or factory objects as these are created and configured in the main class using dependency injection.)

## 3. Startup

The startup phase of the application entails the following steps:

1. **Static Initialization** — Adds resource bundles and modifies bean information for encoding
2. **Editor Construction** — Creates and configures objects that make up the editor
3. **Plugins Initialization** — Initializes plugins which may alter the existing configuration
4. **UI Construction** — Creates the user interface based on the final configuration

### 3.1. Static Initialization

The static initialization is executed when the class is loaded by the classloader. It adds a set of resource bundles using the following code:

```
JGraphEditorResources.addBundles(new String[] {  
    "com.jgraph.pad.resources.actions",  
    "com.jgraph.pad.resources.menus",  
    "com.jgraph.pad.resources.strings",  
    "com.jgraph.pad.resources.tools" });
```

The resource bundles used in the application are split into resources for actions, menus and tools. General resources are placed in the strings resource bundle. Note: Internationalized resources are located in the `src/plugins/i18nplugin` directory to simplify *stripping-down* the editor, for example for embedded use.

To avoid encoding redundant properties with the `XMLEncoder`, the static initializer also issues a series of calls to make fields transient. For example, the code makes all irrelevant fields of the `JGraphpadVertexView` transient:

```
JGraphEditorModel  
    .makeCellViewFieldsTransient(JGraphpadVertexView.class);
```

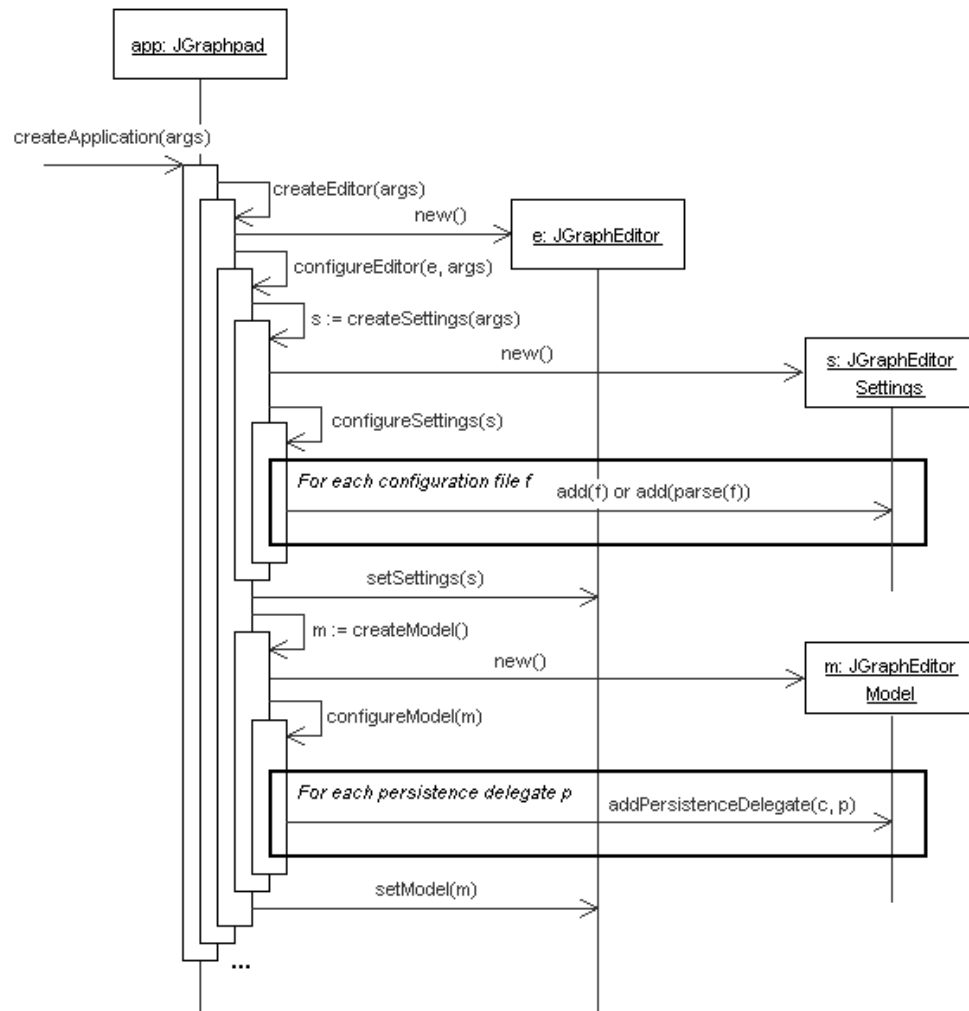
In general, it is recommended to call this method for all custom cell views of an application, as it helps to reduce the size of the files produced by the `XMLEncoder`.

Other examples of transient fields include the filename and isModified flag of the JGraphpadFile class, which is being made transient using the following code in the static class initializer:

```
JGraphEditorModel.makeTransient(JGraphpadFile.class, "modified");
JGraphEditorModel.makeTransient(JGraphpadFile.class, "filename");
```

## 3.2. Editor Construction

**Figure 3.2. JGraphpad Startup**



The application startup is triggered by the execution of the main method or by invoking the init method of the applet. Both call the createApplication method as shown above. The createApplication method, after applying general arguments, such as the look and feel to be used, will call the createEditor method while displaying a splash window (not shown in the sequence diagram).

The createEditor method creates a new empty JGraphEditor object, installs an exit hook and calls configureEditor. Following the pattern from the factory, the configureEditor method creates the enclosed objects, namely the settings, model, kit and factory (in this order) by calling the respective create meth-

ods. Note that the args parameter is looped-through to the createSettings call, which is assumed to fetch the relevant arguments and store them in the settings. (Thus, the args parameter is only passed to createSettings, but none of the later calls.)

### 3.2.1. Settings

The createSettings method creates a new JGraphEditorSettings object and passes it to the configureSettings method. The latter adds the ui.xml configuration file and the user settings file, if it exists. If it does not exist, it adds the default.ini settings file from the resources package. Additionally, it installs a shutdown hook to store the user settings file back to the filesystem when the app terminates. Note that the shutdown hooks are processed in reverse order, thus making sure that this will be the last shutdown hook to be invoked.

Finally, it installs the prototypes cells which are created using the createVertex, createGroup and createEdge methods. The prototypes are placed in the map using this code:

```
settings.putObject(KEY_GROUPPROTOTYPE, createGroup());
settings.putObject(KEY_VERTEXPROTOTYPE, createVertex());
settings.putObject(KEY_EDGEPROTOTYPE, createEdge());
```

The prototypes are used in factory methods and action bundles which require prototypes, such as the group action:

```
Object groupPrototype = editor.getSettings().getObject(
    JGraphpad.KEY_GROUPPROTOTYPE);
actionGroup.putValue(KEY_PROTOTYPE, groupPrototype);
```

Note that the prototype is stored in the action under a different key to decouple the action implementation from the settings object.

### 3.2.2. Document Model

The createModel method passes a new instance of JGraphEditorModel to the configureModel method, which adds persistence delegates for all objects that the document model may contain, such as documents, libraries, diagrams, layout caches, graph models, cells, user objects and attributes.

An interesting trick 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 edgeset 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:

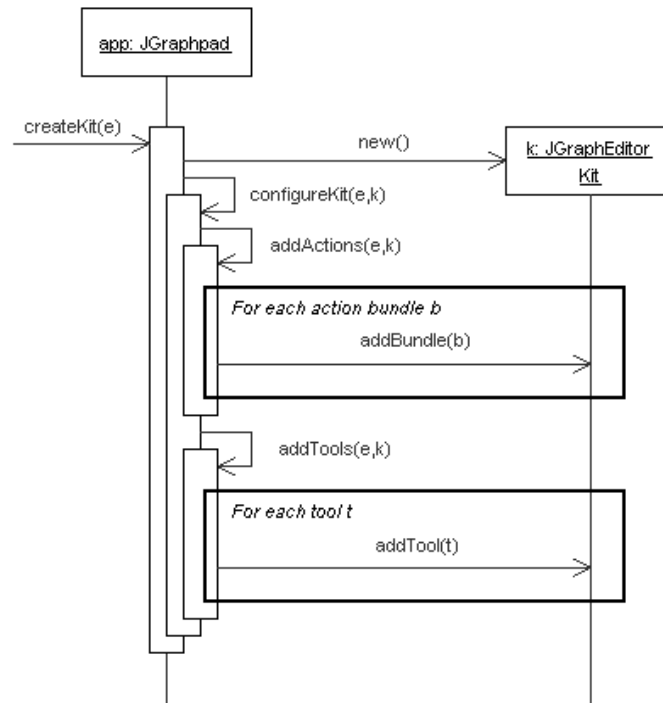
```
model.addPersistenceDelegate(JGraphpadGraphModel.class,
    new DefaultPersistenceDelegate(new String[] { "roots",
        "attributes", "connectionSet" }));
```

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");
```

### 3.2.3. Kit

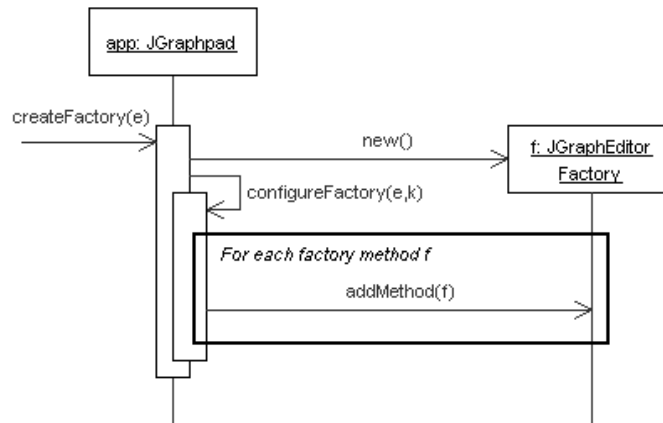
**Figure 3.3. createKit**



Again, the create method passes a new instance to the configure method. The kit is configured using the `addActions` and `addTools` helper methods, which add the actions and tools respectively. The rest of the method creates various listeners to invoke the update method of the kit, namely for all graph changes, focus traversal and selection changes in the library.

### 3.2.4. Factory

**Figure 3.4. createFactory**





The construction of the factory completes the creation of the editor object. In `createFactory`, the `createGraph` method of the factory is redirected to the local `createGraph` method for ease of use. (It is assumed that the `createGraph` method will be the most frequent overridden method of the factory.) In the `configureFactory` method, the various factory methods are added to create the user interface.

### 3.3. Plugins Initialization

As mentioned earlier, plugins are an optional concept in the framework. The application uses them in various situations, namely if they require external libraries or if they contain code which is likely to be removed from production code. An advantage of this implementation is that JGraphpad Pro compiles without the plugins, since plugins are dynamically loaded at runtime. Thus, the application ships with a static array of plugin names but will only initialize those that are available in the classpath. The plugin loading and initialization code looks like this:

```
JGraphEditorPlugin plugin = (JGraphEditorPlugin) Class
    .forName(defaultPlugins[i]).newInstance();
plugin.initialize(editor, null);
```

The above code is part of the `createPlugins` method in JGraphpad. Note that this implementation does not make use of the configuration parameter for plugin initialization.

During the initialization of the plugins the configuration, kit and factory are likely to be changed by added or replaced objects in the kit and factory and inserted or merged files in the settings.

### 3.4. UI Construction

The final step of the startup phase is the construction of the user interface, which is implemented using the following method call:

```
Window mainWindow = createMainWindow(editor,
    JGraphpadPane.FactoryMethod.NAME);
```

The `createMainWindow` method calls the respective factory method and configures the main window. (For example by installing the exit action in the window's close icon.) The rest of the `createApplication` method deals with post-initializing the user interface, eg. restoring window bounds and divider locations, wire up factory methods or open default files and libraries. Note that in Swing, the divider locations need to be restored after the window has been made visible.

## 4. Shutdown

The application shutdown is implemented with the global exit hook which is to be called after all dialogs have been displayed. The exit hook is meant to invoke the shutdown method of the settings object and terminate the VM in the proper way. In JGraphpad, the exit hook is overridden in the `createEditor` method:

```
JGraphEditor editor = new JGraphEditor() {
    public void exit(int code) {
        super.exit(code);
        Window wnd = (Window) getSettings().getObject(KEY_MAINWINDOW);
        if (wnd != null) {
            wnd.setVisible(false);
            wnd.dispose();
        }
    }
}
```

```
        JGraphpad.this.exit(code);  
    }  
};
```

The overridden method makes sure the superclass method is called and disposes the application main window. It then redirects the exit call to a local exit method, which is implemented in the following way:

```
protected void exit(int code) {  
    System.exit(code);  
}
```

## 4.1. Quitting the Applet

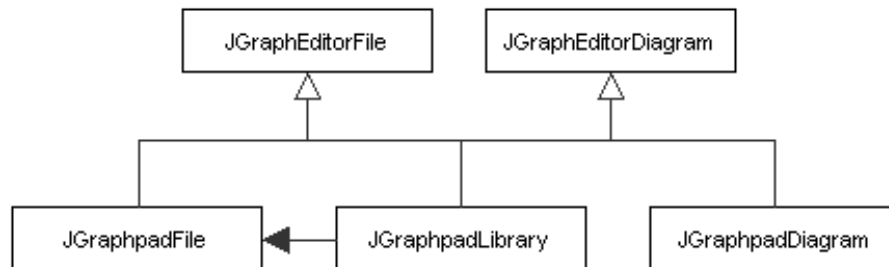
Subclasses can override this method to provide another exit method. For example, the JGraphpadApplet class overrides the method in the following way:

```
protected void exit(int code) {  
    // empty  
}
```

In the case of an applet it is not allowed to call System.exit. The method above overrides the parent method with an empty implementation to avoid the call.

# 5. JGraphpad Document Model

**Figure 3.5. JGraphpad Document Model**



The document model in JGraphpad consists of three classes: The JGraphpadFile and JGraphpadDiagram classes represent the documents, and the JGraphpadLibrary class represents the libraries. A library contains a set of predefined cells to be used in the graphs, aka. stencils, templates, repository etc. From a UI's point of view they are completely different and reside in separate component hierarchies. However, from a model's point of view the library represents the combination of a file and a diagram in a single object, and thus the library extends the JGraphpadFile class and adds most of the code of the JGraphpadDiagram class.

## 5.1. Adding objects

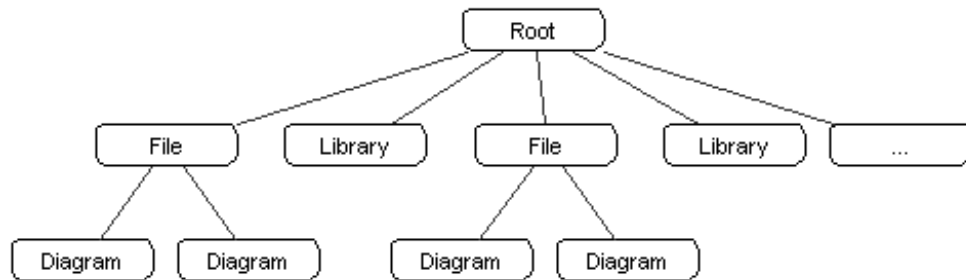
The JGraphpadFile and JGraphpadLibrary objects are added using the addFile method, whereas the diagrams are added to existing files using addChild in the doNewDiagram method of the JGraphpadFileAction class using this code:

```
int number = model.getChildCount(file) + 1;
JGraphpadDiagram newDiagram = new JGraphpadDiagram(
    getString("Diagram") + number);
model.addChild(newDiagram, (JGraphpadFile) file);
```

The cast in the `addChild` call is required to satisfy the `MutableTreeNode` requirement for the *parent* parameter.

The document tree is of the following form:

**Figure 3.6. JGraphpad Document Tree**



All `JGraphpadFile` and `JGraphpadLibrary` objects are roots, and all `JGraphpadDiagram` objects are direct children of `JGraphpadFile` objects. The root contains one or more files (documents or libraries), and the documents contain one or more diagrams.

## 5.2. Removing objects

Objects are removed from the document model using the following code:

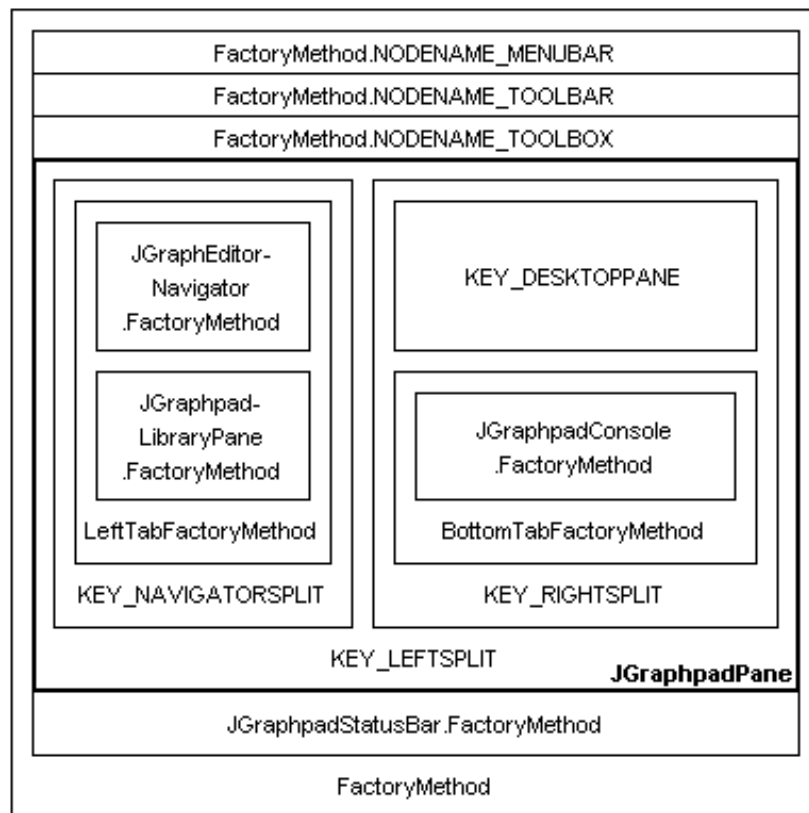
```
model.removeNodeFromParent(mutableTreeNode);
```

Note: Because the application uses the focused component to find the current file in the document model, the `doRemoveDiagram` method in the `JGraphpadFileAction` class does not allow to remove the last diagram in a file. This is because the diagram pane, as mentioned earlier, is the only component that allows to map from the focused component to an object in the document model.

# 6. Custom Factory Methods

## 6.1. Main Window

**Figure 3.7. Main Window Structure**



The main window of the application is created using the `JGraphpadPane.FactoryMethod` class. It is in charge of creating the complete `JFrame` instance, including the menubar, toolbar, toolbox, statusbar and main area. The main area is implemented by the enclosing class, which is a `JPanel` extension, and is split into a left, bottom and content part. The left and bottom part are created by separate factory methods, which return instances of `JTabbedPane`. (The reason for splitting out the left and bottom tabs to separate factory methods is to make it possible for plugins to add new tabs.) The left tab contains a navigator and a library pane, the bottom tab contains an error console which are all created using their respective factory methods as shown in Figure 3.7, “Main Window Structure”.

The figure also shows the nodenames used to retrieve the configurations for creating the menubar, toolbar and toolbox and the keys for the split panes. The desktop pane is stored in the settings for later wiring up with the window menu as shown in Section 5.1.2, “Connecting Factory Methods”.

### 6.1.1. Factory Method

The main factory method of the `JGraphpadPane` class is used to create a frame containing a set of components configured with either built-in or add-on methods of the factory. For example, to create the toolbox, the `createToolbox` method of the factory is used:

```
Node toolboxConfig = JGraphEditorSettings.getNodeByName(
    configuration.getChildNodes(), NODENAME_TOOLBOX);
JGraphEditorToolbox toolbox = editor.getFactory()
    .createToolbox(toolboxConfig);
```

The configuration object represents the top-level node called `ui` in an XML document, and the `getNodeByName` returns the child with the name `toolbox`. The following shows the relevant lines of the configuration file for configuring the toolbox:

```
<ui>
  <toolbox>
    <item key="selectTool"/>
    <separator/>
    <item key="createDashPatternCombo"/>
  </toolbox>
</ui>
```

The above shows that the toolbox does not only contain tools, it also contains components created with factory methods. (The toolbox contains the combo boxes which are used to change the attributes of selected cells.)

The menubar and toolbar are created likewise, by using the respective nodes in the configuration data and the createMenuBar and createToolBar methods, respectively:

```
Node menuBarConfig = JGraphEditorSettings.getNodeByName(
    configuration.getChildNodes(), NODENAME_MENUBAR);
frame.setJMenuBar((JMenuBar) editor.getFactory().createMenuBar(
    menuBarConfig));
[...]
Node toolbarConfiguration = JGraphEditorSettings.getNodeByName(
    configuration.getChildNodes(), NODENAME_TOOLBAR);
JToolBar toolbar = editor.getFactory().createToolBar(
    toolbarConfiguration);
```

In both cases it is possible to use factory methods for creating the named items in the containers, namely JMenuBar and JToolBar.

The code then adds the main area, which is implemented using a JGraphpadPane instance:

```
final JGraphpadPane editorPane = new JGraphpadPane(editor);
frame.getContentPane().add(editorPane, BorderLayout.CENTER);
```

The instance needs to be declared final as it is being used in anonymous inner classes later in the code. To update the JGraphpad pane (editor pane), that is, to add internal frames for new files and tabs for diagrams, and remove them automatically, a listener is added to the document model:

```
editor.getModel().addTreeModelListener(
    new DocumentTracker(editorPane));
```

An inner class called DocumentTracker is used, which acts as a *mediator* between the document model and the editor pane. For example, if tree nodes are changed in the document model, it updates the title of the respective internal frame or tab by calling the updateDiagramTitle and updateFrameTitle methods as shown here:

```
public void treeNodesChanged(TreeModelEvent event) {
    Object[] children = event.getChildren();
    for (int i = 0; i < children.length; i++) {
        if (children[i] instanceof JGraphEditorDiagram)
            pane.updateDiagramTitle((JGraphEditorDiagram) children[i]);
        else if (children[i] instanceof JGraphpadFile)
            pane.updateFileTitle((JGraphpadFile) children[i]);
    }
}
```

```
}

```

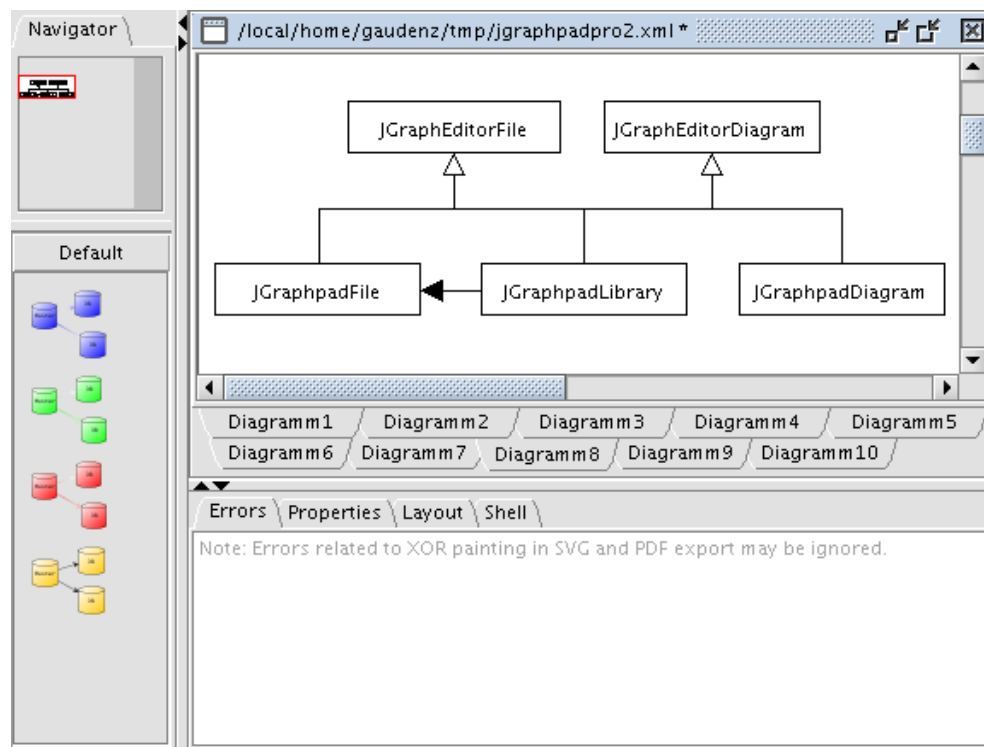
Finally, the code installs listeners to update the frame title and the current graph in the toolbox, and adds a status bar:

```
Component statusBar = editor.getFactory().executeMethod(
    JGraphpadStatusBar.FactoryMethod.NAME);
frame.getContentPane().add(statusBar, BorderLayout.SOUTH);
```

Note that this time, no configuration node is passed to the factory method and therefore, a shortcut method without the configuration parameter is used.

## 6.1.2. Component

**Figure 3.8. JGraphpadPane**



The main area of the window is made up by an instance of JGraphpadPane, which contains three areas separated by split panes. The split panes are created in the constructor, however, from the three areas, only the content area, which is implemented using a JDesktopPane, is created in the constructor. The left and bottom area are created by factory methods using the following code:

```
Component leftTab = editor.getFactory().executeMethod(
    LeftTabFactoryMethod.NAME);
Component bottomTab = editor.getFactory().executeMethod(
    BottomTabFactoryMethod.NAME);
```

The components are then added to the respective split panes. The constructor registers the desktop pane and the split panes in the settings object for restoring their divider location when the window is made visible, and a shutdown hook is added to store the divider locations.

The constructor installs an instance of the `JGraphpadMouseAdapter` class in the desktop pane as a mouse listener using the following code:

```
desktopPane.addMouseListener(new JGraphpadMouseAdapter(editor,
    NODENAME_DESKTOPPOPUPMENU));
```

The code is in charge of displaying a popup menu using the specified configuration from the ui configuration file if the user right-clicks on the desktop pane background.

The rest of the class offers a series of methods to be called by the document tracker when files and diagrams are added, changed or removed. An interesting method is the `configureDiagramPane` hook, which is called when a new diagram pane is created for a diagram. The default implementation of this method restores settings in the diagram pane and the graph, and installs the property change listeners to update the diagram if the properties are changed in the user interface.

This is required to make such properties persistent, as the diagram pane and graph are not stored in the files. The following code is used to update the properties of the graph:

```
diagramPane.getGraph().addPropertyChangeListener(
    new PropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent event) {
            String name = event.getPropertyName();
            if (name.equals(JGraph.ANTIALIASED_PROPERTY)
                [...]
                || name.equals(JGraph.SCALE_PROPERTY))
                diagram.getProperties().put("graph." + name,
                    event.getNewValue());
        }
    });
```

The code names a series of properties which are to be stored in the diagram if they are changed. A similar listener is used to store the properties of the diagram pane. Therefore, the keys under which these properties are stored are to be considered *reserved* keys for other diagram properties. The implementation adds a prefix to the key to find the respective object for restoring the property later.

The method to restore the value is optimized for this implementation, as it takes one key/value pair and a map of prefix/object pairs. If the map does not contain an object for the prefix, then the value is ignored. The code to assign the value uses reflection to set the property of the object:

```
Method setter = object.getClass().getMethod("set" + name,
    new Class[] { clazz });
setter.invoke(obj, new Object[] { value });
```

The name variable in the above code contains a capitalized variant of the key, the `clazz` is used for downcasting object values to their atomic values, and the object is the actual value of the property to be changed. The code is part of the static `setProperty` method in the `JGraphpadPane` class.

### 6.1.3. Left Tab Method

The left tab factory method is used to create a tabbed pane which represents the left area of the main

window. The `createInstance` method creates a navigator and library pane, and puts the navigator into a listener to update the current graph based on the permanent focus owner. The navigator and library pane are created using the following method calls:

```
factory.executeMethod(JGraphEditorNavigator.FactoryMethod.NAME);
factory.executeMethod(JGraphpadLibraryPane.FactoryMethod.NAME);
```

The components are then added to a split pane, using the respective method of the factory to create split panes, and the split pane is registered in the settings:

```
JSplitPane navigatorSplit = editor.getFactory().createSplitPane(
    navigator, libraryPane, JSplitPane.VERTICAL_SPLIT);
editor.getSettings().putObject(KEY_NAVIGATORSPLIT, navigatorSplit);
```

The registered instance is used in a shutdown hook which is installed using the following code:

```
editor.getSettings().addShutdownHook(
    new JGraphEditorSettings.ShutdownHook() {
        public void shutdown() {
            editor.getSettings().storeSplitPane(
                JGraphpad.NAME_USERSETTINGS,
                KEY_NAVIGATORSPLIT);
        }
    });
```

The hook stores the location of the split pane stored under the key in the user settings in memory. The divider of the split pane location is restored in the `createApplication` of the `JGraphpad` class after making the main window visible.

## 6.1.4. Bottom Tab Method

The bottom tab consists of one tab with a console, which shows the output of the `System.out` stream if the security permissions allow it. Otherwise, the console shows a message indicating that it was not possible to redirect the stream. The tabbed pane is created using the respective method in the factory:

```
JTabbedPane tabPane = editor.getFactory().createTabbedPane(
    JTabbedPane.TOP);
```

Then the factory method is executed and the component is added to the tabbed pane. However, this time the component is wrapped up in a scrollpane, which is created using a method of the factory, too:

```
Component console = editor.getFactory().executeMethod(
    JGraphpadConsole.FactoryMethod.NAME);
if (console != null)
    tabPane.addTab(JGraphEditorResources.getString("Errors"),
        editor.getFactory().createScrollPane(console));
```

The code above makes sure that the returned value from the `executeMethod` call is not null, otherwise it does not add the tab. It uses the resource string under *Errors* for the tab title. This resource is to be found in the `strings` resource bundle.



*Journal of Management Education* 36(8) 907-924



```
JGraphpadPane.NODENAME_DESKTOPPOPUPMENU));  
LibraryTracker tracker = new LibraryTracker(tabPane, editor);  
editor.getModel().addTreeModelListener(tracker);  
return tabPane;  
}
```

The factory method is being passed in an editor instance at construction time. The editor instance is used to construct the tabbed pane. The tabbed pane is added a utility mouse listener to display popup menus, and a library tracker is installed to add, update and remove tabs as libraries are being added, changed and removed in the document model. In contrast to the document tracker in the JGraphpadPane class, the library tracker does not rely on special methods on the object to be updated. It uses internal methods to create the instances of the enclosing class and add them to the tabbed pane, as shown in the following example, which is used inside treeNodesRemoved to remove the corresponding tabs:

```
Object[] children = arg0.getChildren();  
for (int i = 0; i < children.length; i++) {  
    if (children[i] instanceof JGraphpadLibrary) {  
        Object tab = tabs.remove(children[i]);  
        if (tab instanceof Component)  
            tabPane.remove((Component) tab);  
    }  
}
```

The library tracker uses an internal hashtable to hold the library/tabs pairs. This is required because the library pane is potentially wrapped up in a number of decorators, making it difficult to iterate over the tabs to find the correct one for a specific library.

## 6.2.2. Component

The JGraphpadLibraryPane extends JComponent and is the user interface widget for libraries. It uses a backing graph to render the entries of the library, a custom TransferHandler to redirect the drop events to the backing graph and a mouse listener to start the dragging and display popup menus. The backing graph has another custom transfer handler to implement the autoboxing feature, which is used to group all inserted cells into one group on a drag. (If the feature is turned off as many entries will be created in the library as cells are dragged on to it.)

The custom transfer handler of the graph overrides the createTransferable and handleExternalDrop methods to implement autoboxing in the following way. When an external drop is received, the complete transfer data is cloned in the first step of the overridden handleExternalDrop method:

```
Map clones = graph.getModel().cloneCells(cells);  
GraphConstants.replaceKeys(clones, nested);  
cs = cs.clone(clones);  
pm = pm.clone(clones);  
for (int i = 0; i < cells.length; i++)  
    cells[i] = clones.get(cells[i]);
```

In the code above, the cloneCells method of the backing graph model is used to create a map with cell/clone pairs. The map is then used to replace all cells by their clones in the nested map, connection set and parent map using the replaceKeys and clone methods on the respective objects.

In the second step, a new parent cell is created. The parent cell is created using a subclass of the DefaultGraphCell called AutoBoxCell, which serves only to identify a cell as an autoboxing cell:

```
DefaultGraphCell parent = new AutoBoxCell();
```

The cell is then added to the cells array and inserted as the topmost cell into the group hierarchy:

```
List tmp = new ArrayList(cells.length + 1);
tmp.add(parent);
for (int i = 0; i < cells.length; i++) {
    tmp.add(cells[i]);
    if (graph.getModel().getParent(cells[i]) == null)
        pm.addEntry(cells[i], parent);
}
```

To find the topmost cells, the code picks the cells with no parents. The transfer data is then passed on to the superclass, which implements the actual insertion of the transfer data into the graph:

```
super.handleExternalDrop(graph, tmp.toArray(),
    nested, cs, pm, dx, dy);
```

To export data from the library into a graph, the `createTransferable` method removes the parent `AutoBoxCell` from the current selection and returns a transferable for the children. To create a list with the children of a node, the following code is used:

```
int childCount = model.getChildCount(cell);
List children = new ArrayList(childCount);
for (int i = 0; i < childCount; i++)
    children.add(model.getChild(cell, i));
```

To create the transferable, the superclass offers the `createTransferable(Object[])` method, which is being passed in the children as an array:

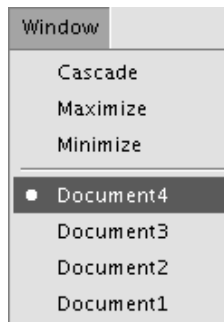
```
return createTransferable(graph,
    graph.getDescendants(graph.order(children
        .toArray()))); // exit
```

The `//exit` comment is sometimes used to mark *abrupt completion* as defined in [bib-JLS], which is normally avoided.

The `JGraphpadLibraryPane` passes the `removeEntry`, `moveEntryToFront` and `sendEntryToBack` methods to the backing graph, and offers a series of methods that specify the geometry of the entries, such as the width, height and horizontal and vertical gap between the entries.

## 6.3. Window Menu

**Figure 3.10. JGraphpadWindowMenu**



The *Window* menu is a partly dynamic menu, ie. items are added and removed at runtime. Therefore it is implemented using a factory method, namely the `JGraphpadWindowMenu.FactoryMethod` class. The enclosing `JGraphpadWindowMenu` class provides a setter method to be assigned a `JDesktopPane` to be used as a source for the dynamic items.

### 6.3.1. Factory Method

The factory method simply constructs a new instance of the enclosing class and registers it with the editor settings:

```
JMenu menu = new JGraphpadWindowMenu();
editor.getSettings().putObject(KEY_WINDOWMENU, menu);
```

The object is used later to set the desktop pane using the `setDesktopPane` method.

### 6.3.2. JGraphpadWindowMenu

The constructor of the `JGraphpadWindowMenu` creates a set of static and dynamic entries, the latter are being kept in a list. Each time an internal frame is added to or removed from the desktop pane all menu items are replaced with the items for the current frames. The items are implemented as a group of radio menu items, one of which is always selected. On selection of the item, the respective frame is selected and brought to front:

```
try {
    frame.setIcon(false);
    frame.setSelected(true);
} catch (PropertyVetoException e1) {
    // ignore
}
frame.toFront();
```

The code above is located in the `createItem` method, which may be overridden to create other menu items for the internal frames.

## 6.4. Open Recent Menu

The *Open Recent* menu is implemented by the `JGraphpadOpenRecentMenu` class and is another dynamic menu, which is obtained using the names of the recent files stored in the settings. When files are opened or saved, the filename is added to the list, making sure that the list has no more than a certain number of entries. When the factory method is executed, the entries in the list are turned into a menu, however, the menu is not changed at runtime. A shutdown hook is installed to write the list to the user settings in memory.

### 6.4.1. Factory Method

In the factory method, the menu is being created by looking through the list of recent files, which is kept in a properties object in the settings:

```
tmp = props.getProperty(KEY_RECENTFILES + i++);
```

The menu items are configured to call `addFile` on the model. As many items are added non-null values are returned for the `getProperty` call. To update the recent files list, the `pushListEntryProperty` method is used in the following way:

```
editor.getSettings().pushListEntryProperty(  
    NAME_USERSETTINGS, KEY_RECENTFILES, filename,  
    MAX_RECENTFILES);
```

The above code adds the filename to the list of entries in the user settings under the specified key, making sure the list has no more than `MAX_RECENTFILES` entries. To implement the list entries, the key is appended an index number, starting at 0. The list is being read from the disk at startup time, and is kept and modified in memory. At shutdown time, the list is stored back to disk using the global shutdown hook. (Note: No custom shutdown hook is required to write the values from the list to the properties object, like in the case of the window bounds and divider locations, as the list entries are stored directly in the properties object.)

The `JGraphpadOpenRecentMenu` is a pure factory method, ie. it inherits from the `Object` class. The `JMenu` instance is being created in the `createInstance` method of the class.

## 6.5. Status Bar

The `JGraphpadStatusBar` is a `JPanel` extension with inner factory method class. The factory method simply creates a new instance of the class and returns it. In the constructor, the panel is created and all required listeners are installed. The status bar listens to focus traversal and document model events. It also listens to mouse motion events on the focused graph. In order to not having to deal with the adding and removing of the respective listener as the focus traverses, the following code is used:

```
JGraphpadFocusManager.getCurrentGraphFocusManager()  
    .addMouseMotionListener(new MouseMotionListener() {  
        public void mouseDragged(MouseEvent e) {  
            mouseLabel.setText(e.getX() + " : " + e.getY());  
        }  
        public void mouseMoved(MouseEvent e) {  
            mouseLabel.setText(e.getX() + " : " + e.getY());  
        }  
    });
```

As you can see, the code updates the text of the mouse label with the coordinates in the mouse event. The `JGraphpadFocusManager` is used to decouple the listener from focus traversal, and keep it installed in the graph that currently has the focus. Thus, the above code is always triggered whenever the mouse is moved over the focused graph, regardless of which `JGraph` instance currently has the focus.

## 6.6. Console

The console object is a text area which is being attached to one of the system streams, namely the `out`

or `err` stream. The class extends `JTextArea` and contains a factory method, implemented as an inner class, which is used to construct the console and assign it the `err` stream if the security settings allow to do so.

### 6.6.1. Factory Method

In the factory method, a console is created for the `err` stream and returned:

```
public Component createInstance(Node configuration) {
    JGraphpadConsole console = new JGraphpadConsole(System.err);
    console.setEditable(false);
    return console;
}
```

The `setEditable` blocks user input in the text area but still allows to copy text to the clipboard.

### 6.6.2. JGraphpadConsole

In the constructor, the system stream is redirected to a wrapper print stream around the text area if the security restrictions allow it. Otherwise a message is printed:

```
PrintStream textStream = new JTextAreaOutputStream(this, stream, true);
try {
    if (stream == System.out)
        System.setOut(textStream);
    else if (stream == System.err)
        System.setErr(textStream);
} catch (SecurityException e) {
    textStream.println(JGraphEditorResources
        .getString("SecurityRestrictions"));
}
```

The code above creates a print stream around the text pane. The print stream offers the various print methods of the system streams and redirects them to the text area. The print stream is then used to redirect the passed in stream.

## 6.7. Combo Boxes

**Figure 3.11. Combo Boxes**



Apart from the combo boxes provided by the framework, the application provides another set of combo boxes for colors and shapes. The code uses a special feature of the combo box to display an icon in place of the cell view renderer if the attributes contain only an icon, as created by the following code:

```
Map attrs = new Hashtable();
GraphConstants.setIcon(attrs, JGraphEditorResources
    .getImage("/com/jgraph/pad/images/color.gif"));
```

The map is being passed in to the combo box constructor as part of an array with other maps, however, this special entry displays the icon instead of the cell. The entry is used to display a color dialog when it

is selected, for which purpose the code overrides the `getSelection` method in the inner `ComboBoxListener` static class. In the method, the index is used to identify the icon entry. If the above entry is added at index 0 in the combo box, the first part of the listener looks as follows:

```
comboBox.addActionListener(new ComboBoxListener() {
    protected Object getSelection(JComboBox box) {
        if (box.getSelectedIndex() == 0) {
            [...]
        }
    }
});
```

In the second part, an attribute map is created using a color from a dialog:

```
Color color = JGraphpadDialogs.getSharedInstance()
    .colorDialog(
        box,
        JGraphEditorResources
            .getString("SelectColor"), null);
Map map = new Hashtable(2);
setColor(map, color);
```

All color combo boxes are implemented using the same factory method, which is being passed in a switch at construction time. Thus, to add all combo boxes to the factory, the following code is used:

```
factory.addMethod(new JGraphpadComboBox.ColorComboFactoryMethod());
factory.addMethod(new JGraphpadComboBox.ColorComboFactoryMethod(
    METHOD_CREATELINECOLORCOMBO, JGraphpadComboBox.TYPE_LINECOLOR));
factory.addMethod(new JGraphpadComboBox.ColorComboFactoryMethod(
    METHOD_CREATEGRADIENTCOMBO, JGraphpadComboBox.TYPE_GRADIENT));
```

This will add three combo boxes to the factory for choosing the background, gradient and linecolor respectively.

The second type of combo box, with only one implementation, is implemented by the `VertexShapeComboFactory` method. It allows to change the shape of the selection vertex. This factory method passes a custom view to the combo box to render the shapes.

## 7. Custom Tools

The basic `JGraphEditorTool` that comes with the framework only implements a basic marquee handler with a name and a `isAlwaysActive` flag. The application provides two standard subclasses for the default tool, the `JGraphpadVertexTool` and the `JGraphpadEdgeTool`, the latter extending the first. The vertex tool is used to insert cells with bounds, the edge tool is used to insert cells with points. The two use a prototype cell which is being assigned at construction time and cloned using the target graph model at insertion time. The tools are added to the kit in the `addTools` method of the `JGraphpad` class. To construct the tools and prototypes, the `JGraphpad` class contains a series of methods, most importantly the `createVertexTool` and `createEdgeTool`, which in turn use the `createVertex` and `createEdge` method to create the prototypes based on the passed in data.

### 7.1. Adding Tools

To add a new tool based on the `JGraphpadVertexTool` or `JGraphpadEdgeTool`, the `createVertexTool` and `createEdgeTool` methods are to be used, respectively. The following example uses an instance of `JTree` as the user object for a new tool called *treeTool*:

```
JTree tree = new JTree();
tree.setRootVisible(false);
kit.addTool(createVertexTool("treeTool", tree,
    JGraphpadVertexRenderer.SHAPE_RECTANGLE, null));
```

The `createVertexTool` method takes the name of the tool, the user object, vertex shape and optional image for the prototype to be created. The bounds are later added by the vertex tool at insertion time. Note that the use of a Component as the user object is a special case in terms of rendering and editing which is discussed later. What's important for now is that the application handles instances of `JGraphpadRichTextValue` and Component different from normal objects. For example, to add the default rectangle tool, which supports rich text values, the following code is used:

```
kit.addTool(createVertexTool(NAME_VERTEXTOOL,
    new JGraphpadRichTextValue(""),
    JGraphpadVertexRenderer.SHAPE_RECTANGLE, null));
```

This code assigns a rich text user object and rectangle shape to the prototype in the new tool. Likewise, to add a new edge tool, the following code is used:

```
kit.addTool(createEdgeTool("orthogonalEdgeTool",
    "", GraphConstants.ROUTING_SIMPLE));
```

The `createEdgeTool` takes the name, user object and optional routing of the edge prototype to be created. Internally, the `createEdgeTool` uses the following code to create the prototype and link it with the new edge tool:

```
DefaultEdge edge = createEdge(createEdgeUserObject(defaultValue));
if (routing != null)
    GraphConstants.setRouting(edge.getAttributes(), routing);
return new JGraphpadEdgeTool(name, edge);
```

The code above creates a new edge tool using the `createEdgeUserObject`, which creates a wrapper around *defaultValue*. The edge is then created using the `createEdge` method, assigned the optional routing and passed to the tool's constructor.

The final step of adding a new tool is to add the resources and configurations accordingly. The resources are added to the `tools` resource bundle as follows:

```
# treeTool
treeTool.tooltip=Tree
treeTool.icon=/com/jgraph/pad/images/tree.gif
```

The line starting with `#` is a comment and is ignored by the parser. The use of comments to separate the entries is recommended to keep the file readable. Note that the tooltip is not assigned a label, as the corresponding button in the toolbox only displays an icon and tooltip.

The final step is to add the tool to the `ui.xml` configuration file, so it is added to the toolbar at UI construction time:

```
<ui>
```



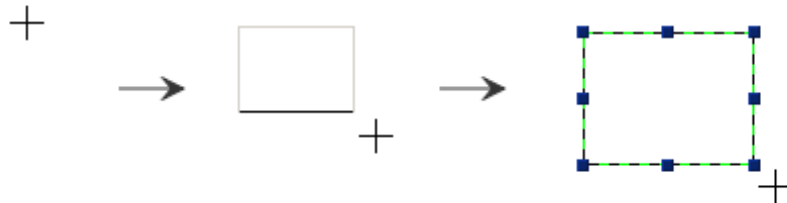
```

<toolbox>
  [...]
  <item key="treeTool"/>
</toolbox>
</ui>

```

## 7.2. Vertex Tool

**Figure 3.12. Steps of the Vertex Tool**



The process of creating new cells with a tool typically starts with a call to the `mousePressed` method. In the `mousePressed` method, a new cell is created based on the prototype using the following method:

```

protected Object createCell(GraphModel model) {
    return DefaultGraphModel.cloneCell(model, prototype);
}

```

The method creates a *deep clone*, ie. including all referenced objects, such as children, user object and attribute map. The clone is then passed to the cell view factory to create a temporary cell view:

```

Rectangle2D rect = graph.fromScreen(graph
    .snap((Rectangle2D) marqueeBounds.clone()));
previewView.getAttributes().applyValue(
    GraphConstants.BOUNDS, rect);
previewView.refresh(graph.getModel(), graph
    .getGraphLayoutCache(), false);

```

The first line aligns the event point to the grid and scales if required. The new bounds is then assigned to the *view-local attributes* of the cell view. The refresh call makes sure the new bounds are merged with the attributes of the clone, so that the cell view may be used to implement a for the clone.

The next phase of the insertion process is a sequence of `processMouseDragged` calls, during which the preview cell is updated:

```

Rectangle2D rect = graph.fromScreen(graph
    .snap((Rectangle2D) marqueeBounds.clone()));
previewView.getAttributes().applyValue(GraphConstants.BOUNDS, rect);
previewView.update();

```

The code again snaps and scales the bounds of the marquee down to model coordinates to update the cell

view's bounds in-place and calls update on the view to reflect the change. Note that refresh is not required this time, as no model data (ie. children, connections or cell attributes) has changed.

The basic marquee handler calls the processMouseDragged hook method from the mouseDragged method in the following way:

```
overlay(graph, g, true);
processMouseDraggedEvent(e);
g.setColor(bg);
g.setXORMode(fg);
overlay(graph, g, false);
```

Thus, the processMouseDragged method provides a hook to update the state of the tool between clearing and repainting of the canvas, which is typically used to update the temporary cell view.

The final step of the process is to insert the clone with the new bounds. To assign the bounds to the clone, the following code is used:

```
Object cell = previewView.getCell();
graph.getModel().getAttributes(cell).applyMap(
    previewView.getAllAttributes());
execute(graph.getGraphLayoutCache(), cell);
```

As can be seen from above, the code invokes the execute method to actually insert the cell into the graph. The execute method is implemented as follows:

```
protected void execute(GraphLayoutCache cache, Object cell) {
    cache.insert(cell);
}
```

Thus, it inserts the clone into the specified graph layout cache. Note that this method is provided as a hook for subclassers and is overridden in the edge tool.

The overlay method is used to paint the preview cell view.

## 7.3. Edge Tool

The edge tool is implemented by the JGraphpadEdgeTool class, which extends JGraphpadVertexTool. The edge tool requires a different prototype to be used and sets the isAlwaysActive flag to true. This is because in contrast to the vertex tool, which lets events "through" to selection cells, the edge tool should handle such events.

For example, if a port is under the mouse pointer when the mouse is pressed, and the port is an accepted source for the edge, then the port is used as a start point:

```
PortView tmp = graph.getPortViewAt(event.getX(), event.getY());
if (graph.getModel().acceptsSource(previewView.getCell(),
    (tmp != null) ? tmp.getCell() : null))
    start = tmp;
```

Note that the code above uses the fact the superclass implementation has created a preview cell view using a clone of the prototype.

In the processMouseDragged method, the port of the preview edge view are updated and the view is updated:

```
edge.setSource(start);  
edge.setTarget(current);  
edge.update();
```

The mouseReleased method is overridden to reset the tool state, and the execute method performs the actual insert as follows:

```
protected void execute(GraphLayoutCache cache, Object edge) {  
    Object source = (start != null) ? start.getCell() : null;  
    Object target = (current != null) ? current.getCell() : null;  
    cache.insertEdge(edge, source, target);  
}
```

Note the use of the insertEdge helper method which creates the appropriate connection set.

## 8. Dialogs

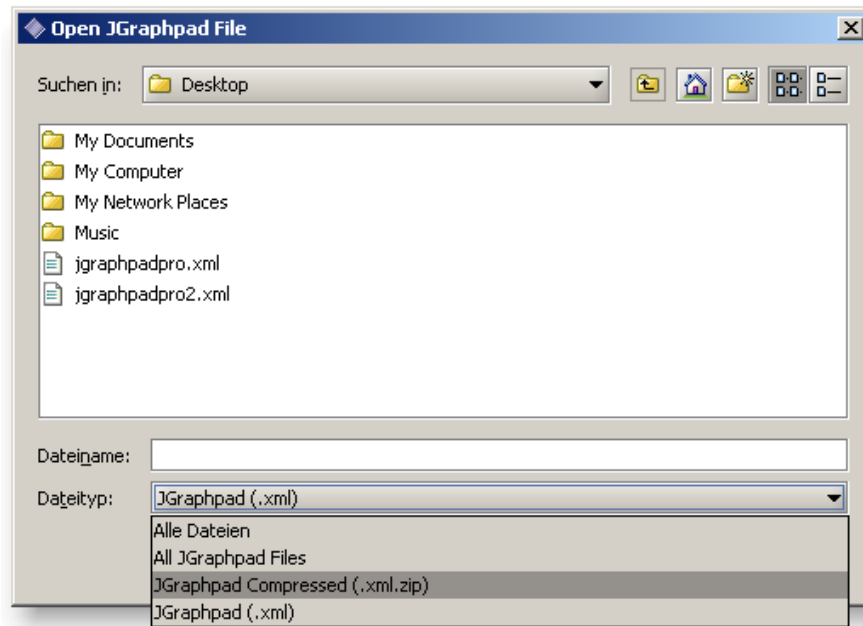
The application offers a series of classes to create and display dialogs located in the dialog package. The built-in dialogs, such as file, color, value etc. are provided using methods in a *singleton* class, whereas the other classes in this package may be used to create custom dialogs.

### 8.1. Built-In Dialogs

The JGraphpadDialogs class is implemented as a singleton class and provides an indirection to display standard Swing dialogs in a pre configured way. The colorDialog, input and message dialogs are implemented using the JColorChooser and JOptionPane classes, respectively. They do not require further explanation.

#### 8.1.1. File Dialogs

**Figure 3.13. File Dialog**



Apart from those simple dialogs, the class offers reconfigured file dialogs for application files and images, which are to be used as in the following example to ask for a filename:

```
JGraphpadDialogs.getSharedInstance().editorFileDialog(
    getActiveFrame(), getString("OpenJGraphpadFile"),
    null, true, lastDirectory)
```

The null argument is used to indicate that no default filename should be displayed in the dialog, true is used to display an *open* dialog. The last directory will serve as the current directory in the dialog. Note that since many actions may share the same file dialog instance, the last directory is remembered in the actions, not the dialog, using the following code:

```
if (!JGraphEditor.isURL(filename))
    lastDirectory = new File(filename).getParentFile();
```

Because the lastDirectory is a file and is used to define a default directory on the local filesystem, the code above makes sure the file is not a remote file, and uses its parent directory as the default directory for the next use of the dialog in the action. (For the sake of simplicity the chosen file filter will remain unchanged throughout the actions, and will not be reset between multiple uses of the dialog.)

The imageFileDialog method works in the same way for image files. The open dialog offers an additional file filter for all supported image files, like the file dialog for editor files. If the dialogs are used for saving files, then the filterset is reduced and the code assures that the returned filename matches the extension of the chosen filter.

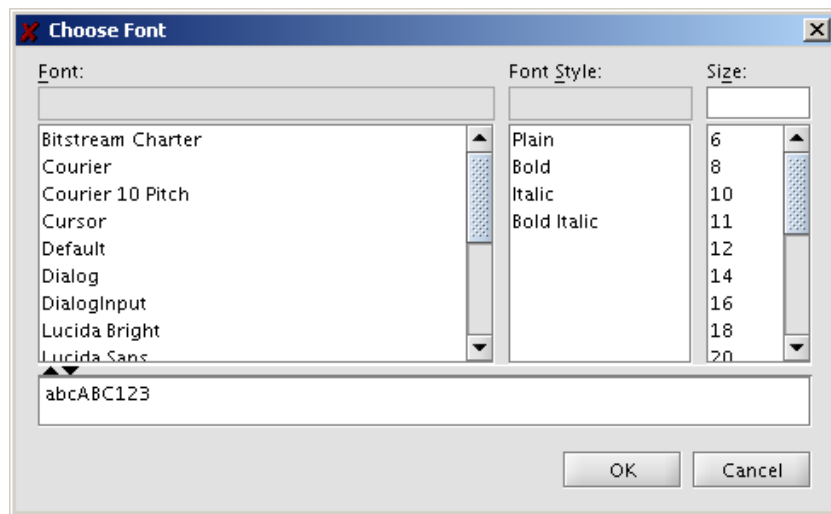
The fileDialog method is a general implementation for custom files. It is used for example to display the file dialog for importing comma-separated value (CSV) files. CSV files contains triples of the form: source label, target label, edge label. For each entry, an edge is inserted between the specified source and target, which in turn are inserted if they do not already exist. To display the CSV file dialog, the following code is used:

```
JGraphpadDialogs.getSharedInstance().fileDialog(
    getPermanentFocusOwner(), getString("OpenCSVFile"),
    true, ".csv",
    getString("CommaSeparatedFileDescription"),
    lastDirectory));
```

Again, the last directory is kept in the action object, not the dialog. Note the leading dot in the extension passed to the method.

## 8.1.2. Font Dialog

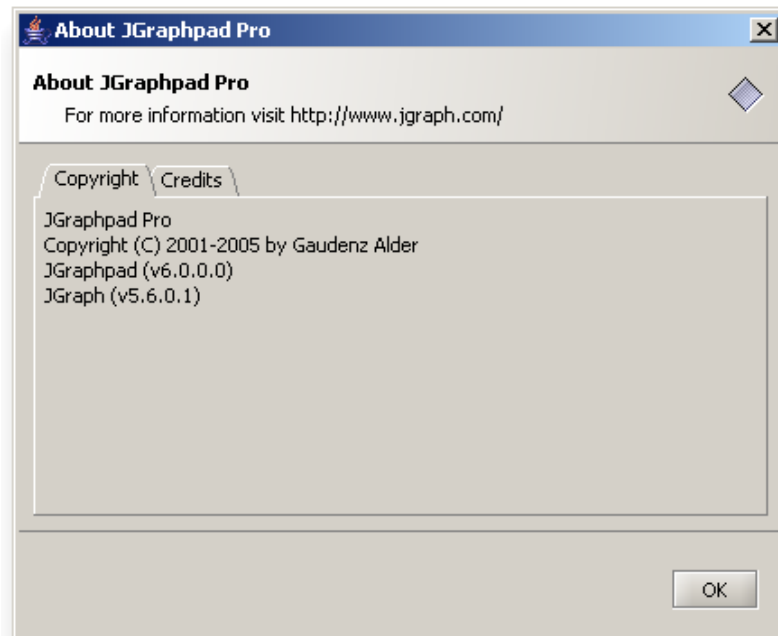
**Figure 3.14. L2FProd Plugin Font Dialog**



The `fontDialog` method is used to display a font dialog, which is not part of Swing, however, it is considered a standard dialog for applications. The method provides a simple default implementation which should be replaced in production code to display a more elaborate dialog, such as the one provided with the L2FProd component suite as shown in Figure 3.14, “L2FProd Plugin Font Dialog”. Note that since the dialog is provided by a plugin, which should not replace the singleton instance, the plugin replaces the respective action instance in the kit instead. (It is not allowed for plugins to replace the singleton instance because each plugin would replace the previous instance.)

## 8.2. Custom Dialogs

**Figure 3.15. JGraphpadAboutDialog**



To establish a consistent user interface, the custom dialogs inherit from a common base class, which defines the basic elements of the dialog. The `JGraphpadDialog` (note the singular) is used for this purpose. The class provides two constructors, one offering a title only, the other offering an additional subtitle and icon. If the second constructor is used, the dialog will be added a gradient header displaying the title as shown in Figure 3.15, “`JGraphpadAboutDialog`”. The second standard element of dialogs is a set of buttons at the bottom of the window. For example, to add the OK button in the about dialog, the following code is used:

```
JButton okButton = new JButton(JGraphEditorResources.getString("OK"));
addButtons(new JButton[] { okButton });
```

The `addButtons` method will add the buttons in the respective panel on the dialog. The listener is to be installed by the calling code, for example to close the dialog:

```
okButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        setVisible(false);
    }
});
```

Note that the code above does not dispose the dialog, so the instance can be reused throughout the application lifecycle.

To create the actual content of the dialog, the `createContent` method may be overridden or the `setContent` method may be used to set the component that represents the content area.

In the case of the about dialog, a special root pane is created to close the dialog when the Escape key is pressed:

```
protected JRootPane createRootPane() {
    KeyStroke stroke = KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0);
    JRootPane rootPane = new JRootPane();
    rootPane.registerKeyboardAction(new ActionListener() {
        public void actionPerformed(ActionEvent actionEvent) {
            setVisible(false);
        }
    }, stroke, JComponent.WHEN_IN_FOCUSED_WINDOW);
    return rootPane;
}
```

## 8.2.1. Authentication Dialogs

A special case of a dialog is the `JGraphpadAuthenticator`. It is a subclass of the `Authenticator` class in the `java.net` package. This dialog is normally not used directly. Rather, it is passed to the `Authenticator` class as the default dialog in the static initializer of the `JGraphpad` class:

```
Authenticator.setDefault(new JGraphpadAuthenticator());
```

The dialog is used to obtain authentication for network connections.

# 9. Utilities

From the classes in the `util` package we have already come across some usage scenarios for `JGraphpadFocusManager`, `JGraphpadImageIcon` and `JGraphpadMouseAdapter`. The file dialog filter, GIF image encoder, morphing animation manager, shadow border, tree model adapter and the two routers are new.

## 9.1. File Filter

The `JGraphpadFileFilter` is only used in conjunction with file dialogs to filter files for certain extensions. The class contains the `ImageFileFilter` and `EditorFileFilter` inner classes for filtering a set of valid files in the case of open dialogs, where a filter may be used to filter more than one file. The class is used for example to configure the open editor file dialog in the `JGraphpadDialogs` class as follows:

```
FileFilter allEditorFilter = new EditorFileFilter(
    JGraphEditorResources.getString("AllJGraphpadFilesDescription"));
FileFilter compressedEditorFilter = new JGraphpadFileFilter(".xml.zip",
    JGraphEditorResources
        .getString("JGraphpadCompressedFileDescription"));
```

The code creates an `EditorFileFilter` and a specialized filter for compressed editor files, using resources for the filter descriptions.

## 9.2. Focus Manager

The `JGraphpadFocusManager` class is a helpful utility class for installing listeners in graphs in a focus-independent way. The focus manager decouples the listeners from the focus owner graph. Internally, the object contains a set of listeners which travel along with the focused graph. The events received by the listeners are redirected to those installed in the focus manager.

The focus manager provides this indirection for graph layout cache listeners, graph model listeners, graph selection listeners, mouse- and mouse motion listeners and property change listeners. The focus manager fires the following additional property change events:

**Table 3.1. Property events fired by the JGraphpadFocusManager class**

Property	Description
FOCUSED_GRAPH_PROPERTY	Focused graph changed
GRAPHLAYOUT_CHANGE_NOTIFICATION	Graph Layout cache changed
MODEL_CHANGE_NOTIFICATION	Graph model changed
SELECTION_CHANGE_NOTIFICATION	Selection model changed
UNDOABLE_CHANGE_NOTIFICATION	Undoable edit happened

A listener interested in all graph model- and layout cache change events of the focused graph can therefore install a property change listener for the respective properties in the shared focus manager.

### 9.3. Morphing Manager

The JGraphpadMorphingManager is used to animate a set of location changes and is constructed using a graph instance:

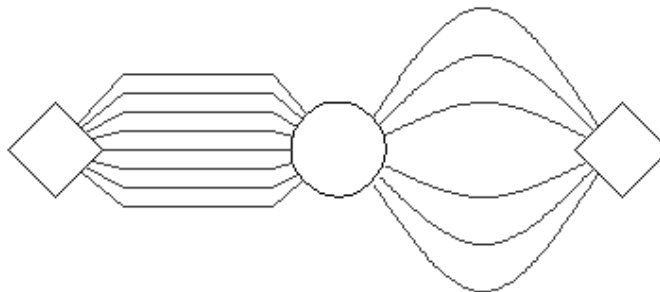
```
morpher = new JGraphpadMorphingManager(graph);
```

The morphing is started using the morph method, passing in the nested map that describes the change:

```
morpher.morph(map);
```

Note that the method returns immediately to not block the *UI dispatcher thread*. For the time of the animation the graph instance is disabled to not interfere with mouse events. (The animation is implemented using a Timer that changes the bounds of the cell views in-place and calls repaint on the graph. Before calling the edit method, the cell view bounds must be restored to handle undo correctly.)

### 9.4. Other Utilities

**Figure 3.16. Parallel Routers**

The other utilities don't have much complexity. The JGraphpadImageEncoder is a copy of the ACME GIF encoder, and the JGraphpadImageIcon is a bean version of the existing ImageIcon class. The JGraphpadShadowBorder is used as a Border with a shadow, and the tree model adapter is an empty implementation of a tree model listener. The JGraphpadMouseAdapter finally is a mouse listener that is



used to display popup menus based on a configuration node. The two routers have been ported from the previous versions of JGraphpad.

## 10. Summary

The JGraphpad Pro application uses the framework to implement a fully functional diagram editor that may be used as an application, applet or with Java Webstart. During application startup, the editor is created by adding the actions, tools, factory methods, resource bundles and configuration files to the respective editor objects.

The user interface is implemented using factory methods, which create the main window with the desktop pane to contain the documents and diagrams, and the library pane to contain the libraries. The document model may contain any number of documents and libraries representing physical files, and documents contain one or more diagrams.

Two base classes are provided to add vertex and edge tools. In addition, a set of reusable classes helps to quickly create new applications. JGraphpad Pro serves as an example for the framework and can be used to create new applications by changing the functionality and configuration.

---

# Chapter 4. Customizing JGraph

## 1. Introduction

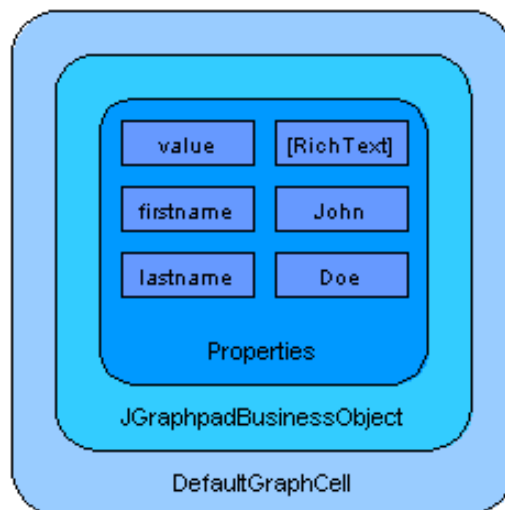
This chapter looks at the various customizations that have been applied to JGraph for this application in the following areas:

- User objects and graph model to implement properties
- Cell view renderer for rich text, heaveweights and shapes
- In-place editing of rich text and heavyweights
- Edge views to support moveable ports
- Transfer handler to implement drag and drop
- Marquee handler for event intercepting

Some of the features that have been part of the custom graph in the previous JGraphpad version, such as printing and background images, have been moved to the JGraphpadPane object. In the following sections, each of the above features will be discussed.

## 2. User Objects

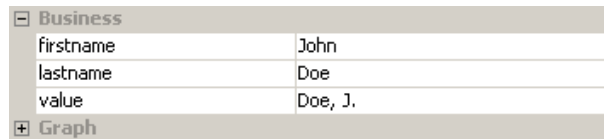
**Figure 4.1. User Object**



To implement custom user objects, one implements the user object and a graph model to take care of the cloning and getting and setting of values. In this case the user object may contain not only string values, but also rich text or heavyweights (Swing components) which are to be rendered and handled by the in-place editor. This section looks at the user object and graph model implementation.

## 2.1. Properties

**Figure 4.2. User Object Properties**



Business	
firstname	John
lastname	Doe
value	Doe, J.
Graph	

The user object is implemented by the `JGraphpadBusinessObject` class. It is implemented as a bean, ie. it offers *accessors* for all properties and provides an empty constructor. It implements the `Serializable` interface for datatransfer and the `Cloneable` interface.

The business object contains the *value* property which is used as the value for rendering the object and to create a string representation in the `toString` method. The business object makes no restrictions as to what can be used as a value, however, it provides the `isComponent` and `isRichText` helper methods for checking if the value is a rich text or heavyweight value. The methods are used for rendering the cells and choose the correct in-place editor.

The user object's clone method is implemented to support being cloned by the graph model. The clone method is expected to return a *deep clone*, which is obtained by creating a new hashtable around the attributes:

```
clone.setProperties(new Hashtable(getProperties()));
```

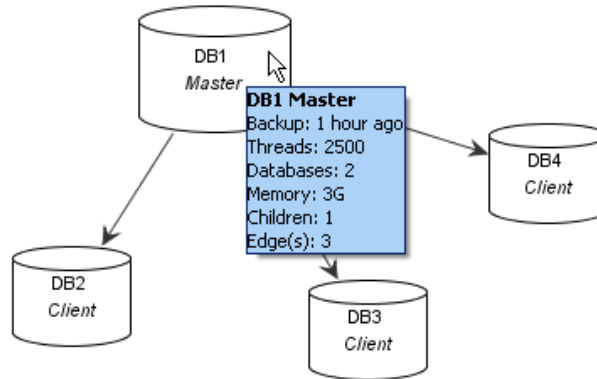
The code then checks some known properties and clones their values. In the current implementation the code checks if the value is a `JTree`, and provides the clone with a new instance:

```
JTree source = (JTree) value;
JTree target = new JTree();
target.setRootVisible(source.isRootVisible());
clone.setValue(target);
```

Note that this is required for all properties that have complex values which are changed in-place. In the case of the tree value, the actual instance is used as the in-place editor and is therefore changed in-place. For rich text values, no special handling is required as the object is replaced upon an in-place edit.

## 2.2. Tooltips

**Figure 4.3. Tooltips**



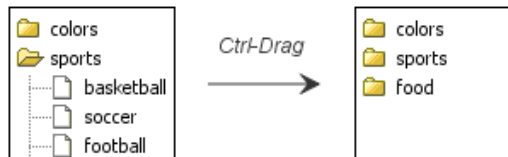
The last important method in the business object is the `getTooltip` method which is used to construct the tooltips in the graph. For this purpose, a special JGraph class is provided which only serves to create the tooltip based on this method. The class overrides the `getToolTipText` method, which finds the cell under the mouse pointer and creates a tooltip for that cell using the `getToolTipForCell(Object)` method, which in turn calls the business objects `getToolTip` method as part of the tooltip construction.

Remember to register the graph with the tooltip manager to support tooltips. This is implemented in the `configureGraph` method of the `JGraphpad` class using the following code:

```
ToolTipManager.sharedInstance().registerComponent(graph);
```

## 2.3. Cloning

**Figure 4.4. Cloning**

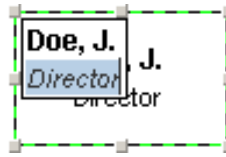


When cells are cloned using the `cloneCells` method of the graph model, the implementation defers the cloning of the user object to the `cloneUserObject` method, which is overridden as follows:

```
protected Object cloneUserObject(Object userObject) {  
    if (userObject instanceof JGraphpadBusinessObject)  
        return ((JGraphpadBusinessObject) userObject).clone();  
    return super.cloneUserObject(userObject);  
}
```

The code creates a clone of the user object if the user object is of the correct type or else calls the super-class method.

## 2.4. Editing

**Figure 4.5. In-Place Editing**

For editing, the `valueForCellChanged` method must be overridden. If the cell's relation to the user object is not implemented using the `DefaultMutableTreeNode`'s user object, then the `getValue` method must be overridden as well. It is called from the graph's `convertValueToString` method to get the user object of a cell and return it as a string for the initial editor value. Since the business object redirects `toString` to the value property, the default implementation will do fine in this case.

The `valueForCellChanged` method is called from within the graph model's `handleAttributes` method to update the user object in the process of handling an edit. Note that the default in-place editor return strings for the new values, however, the mechanism can be extended to use any object, such as a rich text value, which is taken into account in the code:

```
if (newValue instanceof String
    || newValue instanceof JGraphpadRichTextValue
    || newValue instanceof Component) {
    Object oldValue = businessObject.getValue();
    businessObject.setValue(newValue);
    return oldValue; // exit
}
```

In the case of a heavyweight the actual instance is never cloned and therefore it could be ignored. However, since this means that the command history will not work with respect to in-place changes of the tree values, the code that handles the in-place edit is implemented to support heavyweight cloning in the condition. The method also supports replacing the properties of the user object:

```
else if (newValue instanceof Map) {
    Map oldProperties = businessObject.getProperties();
    businessObject.setProperties((Map) newValue);
    return oldProperties; // exit
}
```

In the code above, the properties of the user object are replaced with the new value and the old properties are returned for undo. In the case of an undo, the method will be called with the old properties as the argument.

## 2.5. Transactions

So simplify the process of setting multiple properties of the user object, and to make it part of the command history, the `JGraphpadGraphModel` class extends the `handleAttributes` method to support business objects. In other words, the method allows to use business objects and properties in the same way as cells and attributes are used. For example, to create a nested map that changes the *name* property of a business object, the following code is used:

```
Map trx = GraphConstants.createAttributes(
    businessObject, "name", "John");
```

The *trx* object will contain a hashtable under the key *userObject* which contains the value *John* under the key *name*. The nested map is subsequently passed to the graph layout cache, which constitutes the *transaction demarcation* line.

There is one exception to the rule: To remove properties the code uses the special `VALUE_EMPTY` object, defined in the `JGraphpadGraphModel` class. The value is used to setup a nested map as in the following example:

```
Map change = new Hashtable();
change.put("name", JGraphpadGraphModel.VALUE_EMPTY);
change.put("firstname", "John");
Map trx = new Hashtable();
trx.put(businessObject, change);
```

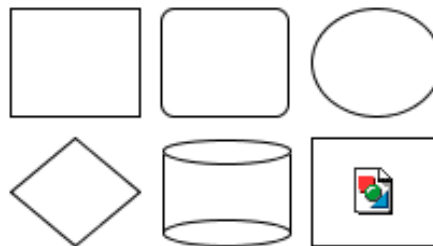
The nested map is passed to the graph model as follows. It will remove the *name* property and add a new *firstname* property to the user object and update the view:

```
graph.getModel().edit(trx, null, null, null);
```

Note that the model, not the layout cache is used to invoke the edit method.

## 3. Rendering

**Figure 4.6. Available Shapes**



The vertex renderer implemented by the `JGraphpadVertexRenderer` class provides a *universal* renderer, that is, one that can be used to render all possible configurations of the `JGraphpadVertexView` class. This includes rendering various shapes, gradient background, image scaling, rich text and redirecting to heavyweight values.

The `JGraphpadVertexRenderer` class is a subclass of `VertexRenderer`, and thus is in fact a `JLabel`. The `JLabel` class provides some default painting which is used to display the vertex, and overridden where necessary. The `JLabel` class is able to paint a rectangular border with an image and a label in the content area. There are various alignment and positioning options. However, the `JLabel`'s painting code is only used if the value is a string, but not for rich text and heavyweight values.

Likewise, the border painting of the superclass is only used if the shape is a rectangle, otherwise the painting of the border is done by `JGraphpadVertexRenderer`. The superclass' icon painting is omitted when images are to be scaled to fill up the vertex area. In general, the implementation uses a combination of `JLabel`, `VertexRenderer` and local code to paint the cell.

The introduction of a custom renderer in JGraph requires to implement a new cell view, which uses the

renderer. Also, to use the cell view in the layout cache, the cell view factory must be replaced. The cell view factory to return the custom cell views is implemented by the `JGraphpadCellViewFactory` class, the cell views are called `JGraphpadVertexView`, `JGraphpadEdgeView` and `JGraphpadPortView`.

## 3.1. Universal Renderer

The advantage of the universal renderer is the reduced redundancy and complexity of the required cell type hierarchy and factory implementation. On the downside it requires special handling of fixed ports when the shape is changed and makes the default renderer code considerably more complex. (Note that the term *fixed port* in this context refers to absolute *and* relative ports.)

To come around the first problem, the default cells contain only one port which is *floating* to make sure the renderer code is used to compute its location. To still make the port stay at a fixed location the *moveable ports* feature is introduced in the next chapter.

Note that the cells are not limited to have only one port, for the default diagram editor it seemed unreasonable to implement a generic mechanism to update the fixed port locations based on the rendered shape, as the renderer is usually fixed to a specific shape in the presence of multiple ports.

However, the `addPorts` method, which is in charge of adding the ports to the cells during the setup of the prototypes is implemented using the following loop to support multiple ports:

```
for (int i = 0; i < offsets.length; i++) {
    GraphCell port = createPort(parent, createPortUserObject(null));
    configurePort(port, offsets[i]);
}
```

The passed in `offsets` reference the global `defaultPortLocations` variable, which is defined as follows:

```
public Point2D[] defaultPortLocations = new Point2D[] { null };
```

The array contains the offsets of the ports to be created. Therefore, in the default case the `addPorts` method creates one port with no offset, also known as a *floating* port. To use relative ports at a specified location, the parent shape must be known. Here are some examples for adding 9 ports to the rectangle, rhombus and circle shapes.

```
int unit = GraphConstants.PERMILLEGraphConstants.PERMILLE;
int u2 = GraphConstants.PERMILLE / 2;
int u0 = (int) (GraphConstants.PERMILLE * cornerPortOffset);
int u1 = (int) (GraphConstants.PERMILLE * (1 - cornerPortOffset));

public static Point[] ninePortLocations = new Point[] {
    new Point(u2, u2), new Point(u0, u0), new Point(u2, 0),
    new Point(u1, u0), new Point(0, u2), new Point(unit, u2),
    new Point(u0, u1), new Point(u2, unit), new Point(u1, u1) };
```

The shape is abstracted using the `cornerPortOffset`, which is to be assigned the following values:

- Rectangle — `cornerPortOffset = 0`
- Circle — `cornerPortOffset = 0.15`

- Rhombus — `cornerPortOffset = 0.25`

The rhombus is referred to as *diamond* throughout the application.

## 3.2. Shapes

If the shape is not a rectangle, then the local code is in charge of painting the background and border. When the paint method is called, the `paintBackground` method is first invoked, which is in charge of painting the background for non-rectangular shapes. The method also prepares some general geometry objects which are used later in the code.

Next, the clipping area is set to the area of the shape, so that all subsequent painting is limited to the shape area. The clipping region is kept until after the super call and painting of the rich text content.

The background image is then painted if image scaling is used and the object is prepared for the super-call of the paint method by setting the border, label and image property accordingly. For example, the following code is used to block the painting of the border, background and selection border if the shape is non-rectangular:

```
if (shape != SHAPE_RECTANGLE) {
    setBorder(null);
    setOpaque(false);
    selected = false;
}
```

The border of the shape is then painted using the `paintShapeBorder` method. In the case of a selection cell, the method is called twice to render the shape and selection border, respectively. The paint method then paints the folding icon which is used to implement one-click-folding in the graph.

## 3.3. Heavyweights

In order to render heavyweights and rich text, two different techniques are used. For heavyweights, the fact that the heavyweight itself is a `Component` instance is used to override the `getRendererComponent` method as follows:

```
return new JComponent() {
    public void paint(Graphics g) {
        valueComponent.setSize(getSize());
        if (graph.getEditingCell() != view.getCell())
            valueComponent.paint(g);
    }
};
```

The `valueComponent` is the actual heavyweight. The code is only invoked if the `valueComponent` is not null and creates a wrapper component around the heavyweight to avoid changes of its properties. The `valueComponent` variable is set in the `installAttributes` method or cleared in the `resetAttributes` method if `installAttributes` is not called, as controlled by the *groupOpaque* attribute defined by the core library.

Note that the approach shown here may not work for all heavyweights. A list of built-in Swing components is available in [bib-Swing].

## 3.4. Rich Text



**Figure 4.7. Rich Text Cell**

The process of rendering rich text uses a different technique that allows to combine the rendering of a text component with that of the universal renderer, which would otherwise not support rich text. To do this, the renderer has a static instance of JTextPane:

```
protected static JTextPane textPane = new JTextPane();
```

In the `installAttributes` the user object's value is checked, and if it is rich text, then the following code is used to update the text pane:

```
StyledDocument document = (StyledDocument) textPane.getDocument();  
((JGraphpadRichTextValue) ((JGraphpadBusinessObject) value)  
    .getValue()).insertInto(document);
```

The code inserts the data represented by the rich text value into the document of the text pane, replacing all previous content. The text pane is then used to paint the rich text value to the graphics:

```
textPane.paint(g);
```

## 4. In-Place Editing

**Figure 4.8. Rich Text In-Place Editing**

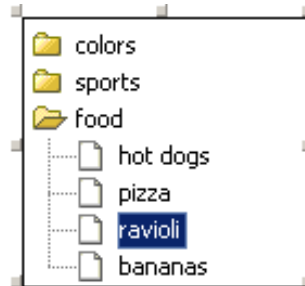
The in-place editing, like the rendering differentiates between string, rich text and heavyweight values. For strings, the default editor is used. for rich text and heavyweights, two different editors are returned by the `getEditor` method based on what the user object contains:

```
if (obj.isRichText())  
    return editor;  
else if (obj.isComponent())  
    return redirector;
```

The above returns two different local editor variables depending on the value in the user object. The default case is to defer the call to the superclass.

## 4.1. Heavyweight Editing

**Figure 4.9. Heavyweight In-Place Editing**



In the case of the heavyweight, a similar technique is used like for rendering. The fact that the heavyweight is a component is used to return the heavyweight as an editor. This is done in the `getCellEditorComponent` method of the `JGraphpadVertexView.RedirectorEditor` class. When editing is finished, the `getCellEditorValue` method is in charge of returning the value to be used in the nested map as follows:

```
public Object getCellEditorValue() {  
    return componentValue;  
}
```

Thus, in the case of heavyweights the components are changed in-place and returned without creating a new instance.

## 4.2. Rich Text Editing

For rich text editing, the default editor is replaced with a text pane. The text pane is configured in the `getCellEditorComponent` method as in the above case:

```
StyledDocument document = (StyledDocument) editorComponent  
    .getDocument();  
((JGraphpadRichTextValue) ((JGraphpadBusinessObject) value)  
    .getValue()).insertInto(document);
```

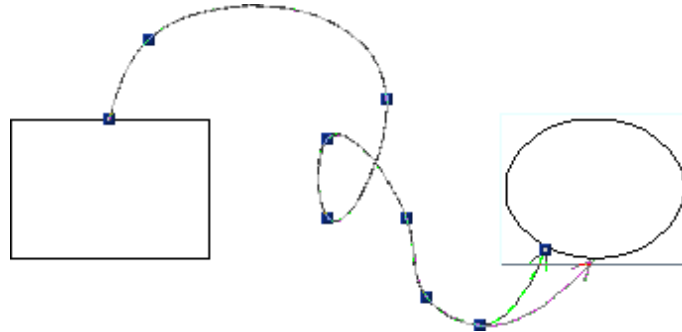
The `editorComponent` is the text pane that is used for rendering. When editing is finished, the value is fetched from the editor using the `getCellEditorValue` method by creating a new rich text object from the changed document:

```
public Object getCellEditorValue() {  
    return new JGraphpadRichTextValue(editorComponent.getDocument());  
}
```

Note that the rich text editor is setup to accept Shift-Enter and Ctrl-Enter and transform them into newlines in the document. The Enter keystroke will stop the editing and apply the value.

## 5. Moveable Ports

**Figure 4.10. Moveable Ports**



The moveable ports feature is implemented by the `JGraphpadEdgeView` class, with two attributes in the `JGraphpadGraphConstants`. Simply put, the moveable port feature stores the relative offset of the port in the edge to achieve a per-edge location of the port. This means the same port can be moved along the vertex boundaries by use of the mouse and locked separately for each edge.

### 5.1. Port Location

It may come as a surprise that this feature does not require a custom port view. By overriding the `getNearestPoint` method in the edge view the port location can be modified in a suitable way. This is implemented by returning the required location from `getNearestPoint`, which is subsequently passed to `getPerimeterPoint` of the port's parent. Since the passed in point is on the perimeter, the `getPerimeterPoint` will return the passed in location as required.

### 5.2. Edge Handle

The setting of the moveable port position is implemented in the `JGraphpadEdgeHandle` inner class, which also allows to reset the port to its floating state. The latter is implemented by using the `isRemovePoint` trigger or by moving the port to its non-floating location (usually the center of the cell). Note that port moving is only enabled when an edge is selected.

In the `snap` method of the edge handle, the port offset for the source or target is updated, depending on which end of the edge was initially clicked:

```
if (isSource)
    JGraphpadGraphConstants.setSourcePortOffset(edge
        .getAllAttributes(), offset);
else
    JGraphpadGraphConstants.setTargetPortOffset(edge
        .getAllAttributes(), offset);
```

The offset is computed using the `computeOffset` method, which takes the perimeter for the event point and turns it into a relative offset to be used by the `getNearestPoint` method. This way, the fixed port location is made size independent, however, if the shape of the vertex is changed, then the port must be repositioned manually.

An interesting hook method that is called from `mousePressed` in the basic edge handle is `processNestedMap`. It is called to setup the nested map which is being used to perform the edit call on the layout

cache. In the case of the edge handle, it is used to remove the offset attributes from the original edge if the offset location was reset in the preview as follows:

```
Map attrs = (Map) nested.get(edge.getCell());
if (attrs != null) {
    String[] removeAttributes = new String[] { (source) ?
        JGraphpadGraphConstants.SOURCEPORTOFFSET
        : JGraphpadGraphConstants.TARGETPORTOFFSET };
    GraphConstants.setRemoveAttributes(attrs, removeAttributes);
}
```

Since we are dealing with a *nested* map the code must extract the actual change for the edge using the first line. It then sets the remove attributes to remove the source or target port offset.

## 6. Drag and Drop

To implement drag and drop, the `JGraphpadTransferHandler` class is used, which is a subclasser of Swing's `TransferHandler`. The class is used to create transfer data on a drag and accept and insert it on a drop on a `JGraph` instance. The `JGraphpadTransferHandler` extends the `GraphTransferHandler` class, which takes care of drag and drop for cells. The `JGraphpadTransferHandler` class extends this to accept more types of transfer data (namely files, text and images). The exported data is not affected by the class.

To be able to create new cells for the inserted objects, the `JGraphpadTransferHandler` requires a prototype. The prototype is cloned and assigned a new user object using the following code:

```
Object cell = DefaultGraphModel.cloneCell(model, prototype);
model.valueForCellChanged(cell, userObject);
```

Thus, the code to create the cell with the user object is completely generic. The graph model's `cloneCells` and `valueForCellChanged` method will make sure the cell is setup correctly for the model.

### 6.1. Chaining Handlers

The actual import is carried out by the `importDataImpl` method. This method is called in the `GraphTransferHandler`'s `importData` method. The reason for introducing this method is that the return value of the `importData` method has a reserved semantic, that is, if it returns true the drag source should remove the transfer data from the local model (move), otherwise not (copy). There is no way of indicating whether the drag has actually been handled.

For this purpose, the `importDataImpl` is in charge of handling the import data. Its return value indicates if the data has been handled. This way, a chain of extensions may be defined which call their respective superclass implementation and handle all transfer data that has not been handled so far. The `JGraphpadTransferHandler` does just that, using the following code at the beginning of the `importDataImpl` method:

```
if (super.importDataImpl(comp, t))
    return true; // exit
```

The code avoids local processing of the transfer data if it has been handled by the superclass, in which case the method returns immediately without further processing.

## 6.2. Importing Files

If the superclass does not handle the import, then it is analyzed for supported data flavors. The current implementation first checks if the transfer data is a file list, using this condition:

```
if (t.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
```

If the above condition holds, then the transferable is turned into an array of filenames and passed to the `insertValues` method, thus, the files are handled as a list of strings. Dropping a file or its name onto a graph results in the same cell being created.

## 6.3. Importing Text

Otherwise, the code tries to find a text flavor for parsing the data using the following code:

```
DataFlavor bestFlavor = DataFlavor.selectBestTextFlavor(t
    .getTransferDataFlavors());
```

If a text flavor may be found, then the text data is passed to `insertValues` using this code:

```
Reader reader = bestFlavor.getReaderForText(t);
insertValues(graph, p,
    new Object[] { getString(reader) });
```

The `getString` method is used to turn the reader into a string. The string is then used in `insertValues` to act as the user object of the cell to be created.

To insert the data, the code checks if the string is the name of a local or remote image file and uses the image in the cell if possible. Additionally, the code assigns initial bounds to the cell and sets the `resize` attribute, so the size is updated on the first insert to reflect the label and optional image. Note that the code should not set any other attributes as the prototype is expected to be properly configured.

# 7. Intercepting Mouse Events

The tricky part of interaction fine-tuning in JGraph is the control flow, which is ultimately defined by the mouse handler in the basic marquee handler. In order to be the first to be notified of an event, and block all further event processing, the marquee handler is used in JGraph.

If more control over the event processing is needed, the `createMouseListener` method in the `BasicGraphUI` class may be overridden to provide a custom mouse handle. However, thanks to the marquee handler, a custom mouse handle is not required to implement folding and popup menus.

## 7.1. Control Flow

The marquee handler is a new class in JGraph that helps to remedy the somewhat limited action listener chaining in Swing. The problem with the action listener chaining is that as soon as the component UI installs a listener into the component, it is no longer possible to block this event processing by adding another listener, since the listeners will be processed in order of insertion. Thus, it is only possible to block event processing in listeners that have been added later.

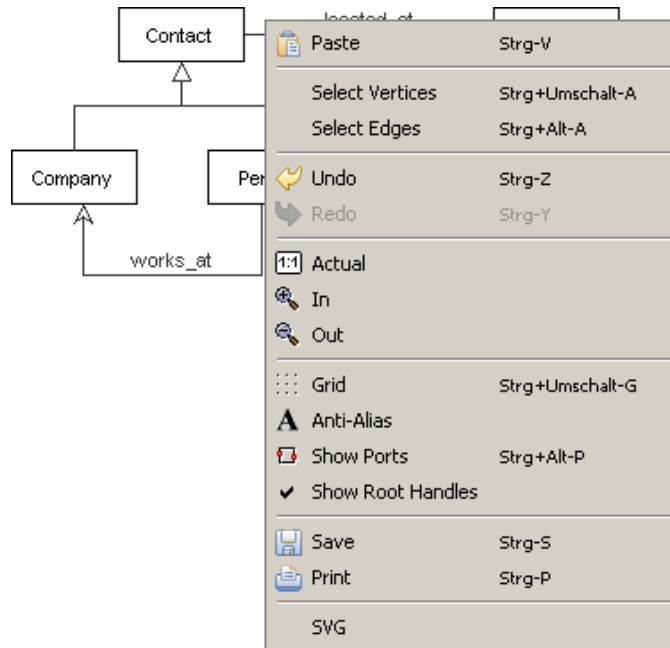
This is because in the event dispatcher loop of the `JComponent` class, the event's `isConsumed` flag is checked and the loop is exited when set. However, at the time the new listener gets a chance to consume

the event, the previous listeners have already been processed.

Thus, to control the event processing in JGraph, the marquee handler should be used to act as an event interceptor. To take *full* control of the control flow, you should override the `createMouseListener` method and provide your own mouse handler.

## 7.2. Popup Menus

**Figure 4.11. Popup Menu**



Popup menus take precedence over all other event processing and are tied to the right mouse button (without holding shift down). If this mouse button is pressed then a popup menu will be displayed and the control will be taken over by the marquee handler for all subsequent events of the gesture. The complete sequence of the gesture is defined by a `isForceMarqueeEvent`, `mousePressed`, a series of `mouseDragged` and a `mouseReleased` call.

In the `isForceMarqueeEvent` method, the popup menu condition is checked as follows:

```
if (SwingUtilities.isRightMouseButton(event) && !event.isShiftDown())
    return true;
```

This will signal the caller to pass the control flow to the marquee. The next step is implemented in the `mousePressed` method, where the code checks if there is a cell under the mouse pointer. If there is no selection cell the code will either select the cell under the mouse, or clear the selection if there is no cell at that location. The popup menu is not yet displayed, only the selection is changed at this point.

The `mouseDragged` method does not do anything in this case and is therefore not overridden. In the `mouseReleased`, the popup menu is displayed using a configuration node:

```
Node configuration = getPopupMenuConfiguration(e);
```

```

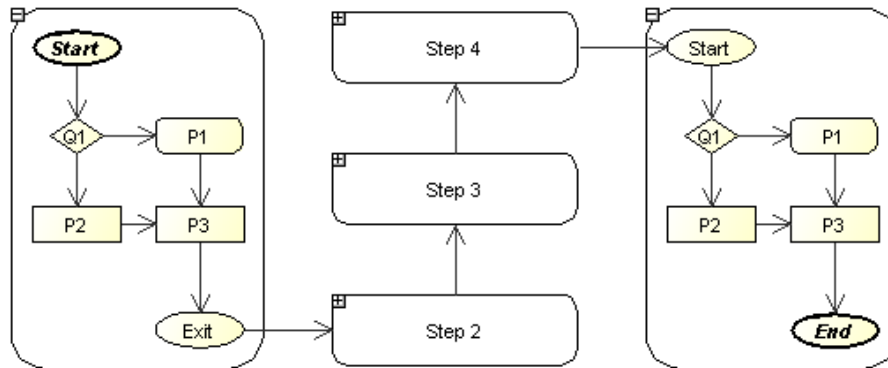
if (configuration != null) {
    getGraphForEvent(e).getUI().setInsertionLocation(e.getPoint());
    JPopupMenu popupMenu = editor.getFactory().createPopupMenu(
        configuration);
    popupMenu.setFocusable(false);
    popupMenu.show(e.getComponent(), e.getX(), e.getY());
    e.consume();
}

```

The configuration node is created using the `getPopupMenuConfiguration` method, which creates the configuration based on the number of selected cells in the graph. The insert location in the graph's UI is then updated in order to paste cells at the location of the mouse. The event consumption make sure that all further event processing is blocked. If the event is not consumed, the mouse handle will perform a delayed selection.

## 7.3. Folding

**Figure 4.12. Folding Icons**



The term *folding* refers to the process of expanding and collapsing group cells, ie. showing or hiding their children. The collapse/expand feature is implemented in the core JGraph API, this section looks at the implementation of one-click-folding by displaying a folding icon in the group renderer and detecting hits on this icon in the marquee handler. The code to paint the icon is implemented in the renderer, which provides a method to check if the icon has been hit. The method is used in the marquee handler's `isForceMarqueeEvent` method to find the respective group cell:

```

groupView = getGroupByFoldingHandle(graph, graph
    .fromScreen((Point2D) event.getPoint().clone()));
if (groupView != null)
    return true;

```

The `getGroupByFolding` handle method helps to find the group at the specified location. The `groupView` is stored for implementing the toggling of the collapsed state in the `mouseReleased` method as follows:

```

if (groupView != null) {
    JGraph graph = getGraphForEvent(e);
    JGraphpadCellAction.toggleCollapsedState(graph,
        new Object[] { groupView.getCell() }, false, false);
    graph.setSelectionCell(groupView.getCell());
}

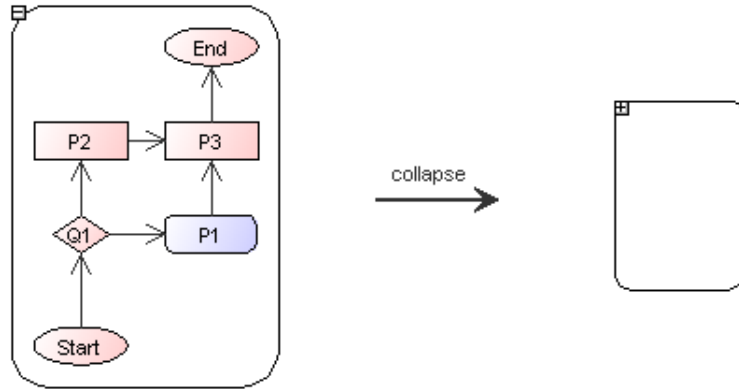
```

```

    groupView = null;
    e.consume();
}

```

**Figure 4.13. Group Resizing and Repositioning**



The `toggleCollapsedState` method collapses or expands the cell based on its current collapsed state. It does some additional resizing and moving of the group cell if the respective attributes are enabled. That is, the `groupResize` attribute will resize the group to half of the width and height of the child area, and the `groupReposition` attribute moves the group to the center of the child area on a collapse. Note that the `groupResize` attribute is only applied once in the lifecycle of a group.

To control the display of the folding icons, a special *client property* is used in the graph. To disable folding icons, the following code is used:

```

graph.putClientProperty(
    JGraphpadVertexRenderer.CLIENTPROPERTY_SHOWFOLDINGICONS,
    new Boolean(false));

```

The client property is checked in the renderer's painting code and also in `inHitRegion`. If the client property is false, then the `inHitRegion` method always returns false:

```

Boolean showFoldingHandle = (Boolean) graph
    .getClientProperty(CLIENTPROPERTY_SHOWFOLDINGICONS);
if (showFoldingHandle == null || showFoldingHandle.booleanValue())
    return handle.contains(Math.max(0, pt.getX() - 1), Math.max(0, pt
        .getY() - 1));
return false;

```

As you can see from the condition, the default value of the property is true.

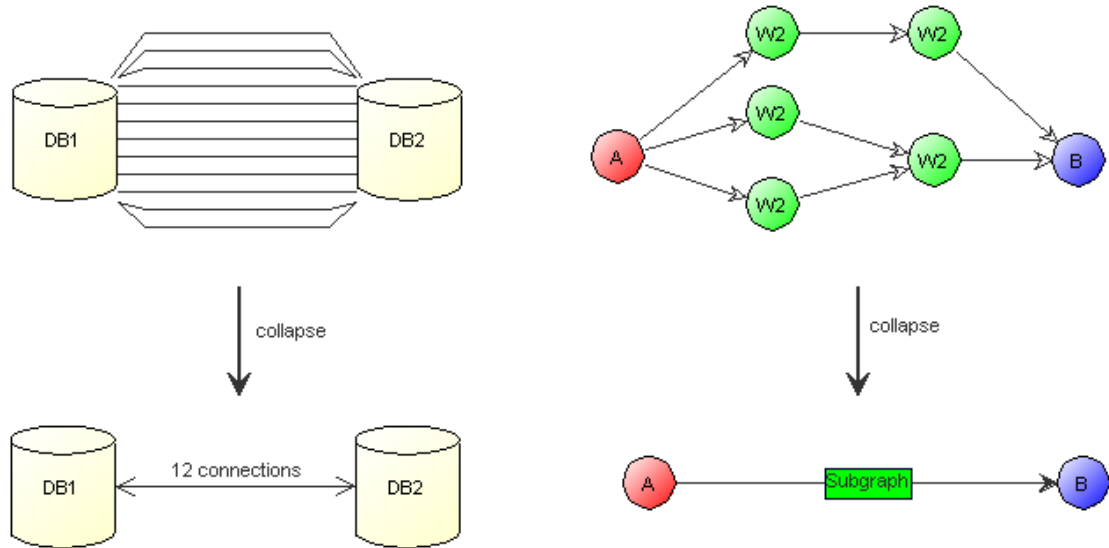
## 8. Special Features

Most features described in this section are fully or partly implemented in the core component. These features are described in full detail in [bib-Benson]



## 8.1. Edge Groups

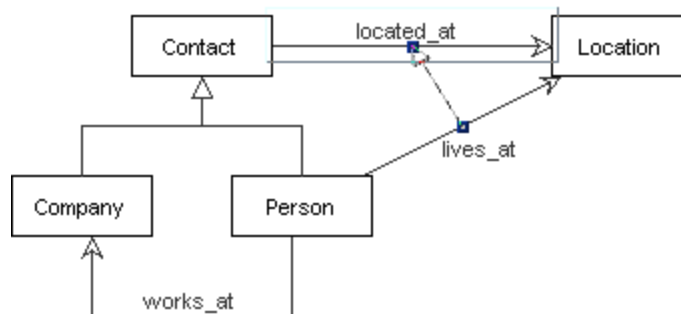
**Figure 4.14. Edge Groups**



Edge groups are a simple extension of the grouping action, which adds a new cell as a parent to the selection cells. This implementation replaces the parent group with an edge and sets its start and end point to the upper left and lower right corner of the child area. The edge group may then be collapsed, in which case it is rendered as a single edge, and expanded, such that the children are displayed. This is useful to group, for example, a set of parallel edges or a path or subgraph as shown in Figure 4.14, “Edge Groups”.

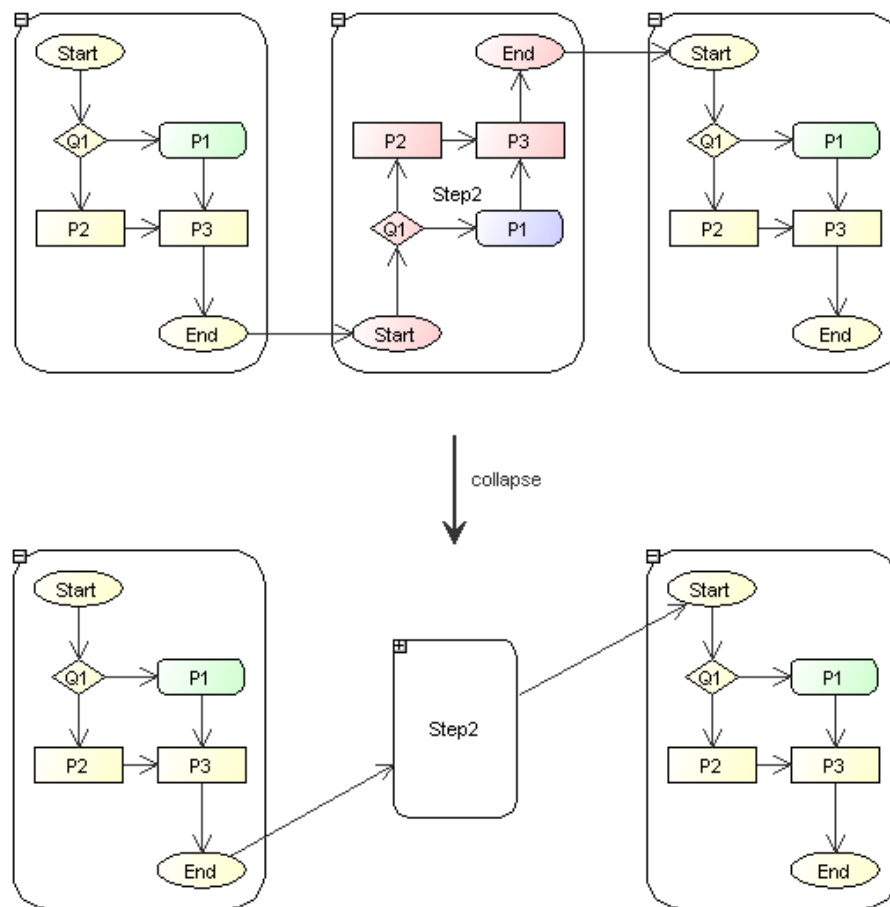
## 8.2. Connecting Edges

**Figure 4.15. Connecting Edges**



In order to allow connecting edges to other edges, a simple default implementation is provided by the JGraph component that can be extended for special purposes. To enable the feature, the application adds a floating port to the edge prototype in the createEdge method. The rest is handled by the core component.

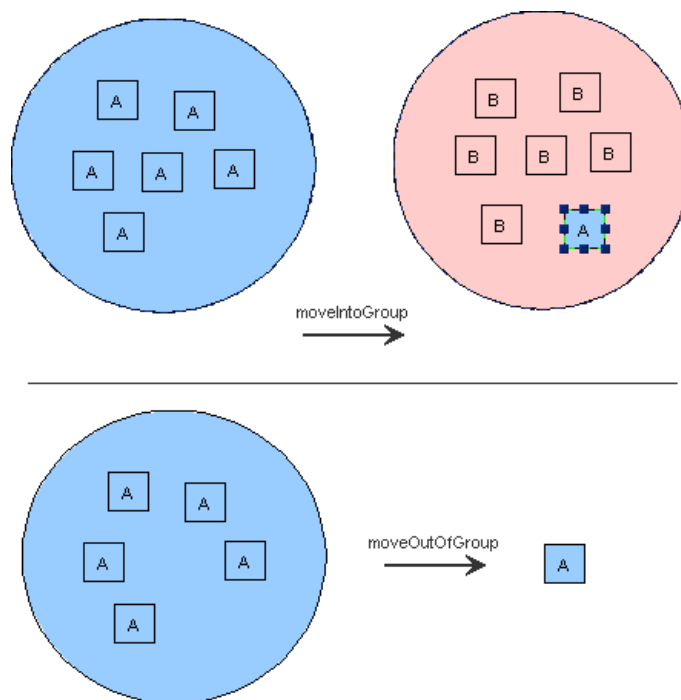
## 8.3. Edge Promotion

**Figure 4.16. Edge Promotion**

The term *edge promotion* refers to the process of painting edges along the parent group boundaries when children are hidden. The process is purely visual, that is, the connection is not actually changed on the model. In the edge view, the port location is simply moved to the boundaries of the parent view if the source or target port is not visible. Note that the layout cache offers various switches for automatic hiding and showing of such edges.

## 8.4. Grouping by Mouse

**Figure 4.17. Grouping by Mouse**



The grouping by mouse feature is implemented completely in the core component as of version 5.6. The feature is controlled with the `isMoveIntoGroups` and `isMoveOutOfGroups` switches. If `isMoveIntoGroups` is enabled, then cells which are moved over groups are inserted into those groups. Note that the actual location of the mouse pointer is used to find the respective group. Likewise, if `isMoveOutOfGroups` is enabled, then cells which are moved outside of their parent group are removed from their parents. Note that the child must no longer intersect with the parent to be removed from the group. This was implemented to protect from accidental removal, but can of course be changed.

## 9. Summary

The custom JGraph component provides a user object with properties and a graph model that is in charge of cloning and updating the user object. The universal renderer allows the cell configuration to be changed at runtime, and implements various default shapes, rich text and heavyweight rendering.

In-place editing was customized to support the editing of such values, and to come across the problems of changing cell shapes at runtime, a moveable port has been implemented. The editor and renderer are installed in the graph layout cache by using a custom cell view factory.

To accept files and text from the filesystem, a custom transfer handler is added to the graph component at configuration time. Likewise, the marquee handler is replaced with one that supports one-click-folding for groups and displays popup menus.

---

# Chapter 5. Plugins

## 1. Introduction

To deal with external dependencies and add-on functionality, the framework provides the JGraphEditor-Plugin interface. The interface is used to define an entry point for initializing a plugin with a specific configuration for an enclosing editor.

JGraphpad defines a set of default plugins to implement the interface in various ways, in most cases by using external libraries. Plugins that do not use external libraries are the i18n-, browser-, codec- and jgx plugins. These plugins consist of class files and resources only. The reason for implementing them as plugins is to make it easy to remove them from the application, and to showcase alternative uses of the plugin interface.

Note that it is only required to restart the application after the removal of a plugin, a recompile is not required. To add plugins to the application, a field of the JGraphpad class must be changed and thus a recompile is required for this implementation. However, factoring out the plugin names to an external configuration file is a trivial task.

## 2. Internationalization

The term *internationalization* (i18n) refers to adding language files to resource bundles. The i18n plugin contains a German translation of all resource files, including the plugin resources. This was done to gather translated resources in one place, rather than distributing them among the plugins. In general, the plugin is meant to contain all non-English resources as they are added over time. This plugin only contains properties files, it does not require compilation and does not define classes.

## 3. Browser Integration

Apart from accepting transfer data from the operating system, desktop integration should also include using the web browser. The browser plugin provides the JGraphpadBrowserLauncher class to allow opening URLs in the default browser of the operating system and adds the homepage action to visit the project website using the following code:

```
JGraphpadBrowserLauncher.openURL("http://www.jgraph.com/");
```

The reason for moving this into a plugin was that the code is tied to some default browser names, and uses special calls to invoke the browsers which are not suitable in all environments. The browser launcher plugin is based on the BrowserLauncher project at <http://browserlauncher.sourceforge.net/>.

## 4. Bean Shell

**Figure 5.1. Bsh Plugin**

```

bsh % graph = mgr.getLastFocusedGraph();
bsh % cell = graph.getSelectionCell();
bsh % print(cell);
The quick brown
fox jumps over
the lazy dog.
bsh % graph.clearSelection();
bsh % graph.getGraphLayoutCache().setVisible(new Object[]{cell}, false);
bsh % graph.getGraphLayoutCache().setVisible(new Object[]{cell}, true);
bsh %
  
```

The bean shell plugin allows scripting (macros) to be added to the application. The plugin is based on the BeanShell project at <http://www.beanshell.org>. The plugin adds a bottom tab to the main window which hosts the bean shell console and requires the `bsh-2.0b2.jar` file. Note that the demo version does not ship with the bean shell plugin, as it is not useful in an *obfuscated* distribution.

## 5. Codecs

The codecs (encoders/decoders) plugin is a standalone plugin that includes code from the previous version of JGraphpad for importing and exporting GXL and GraphViz (Dot) files. It adds the respective actions to the File->Export and File->Import menus. Note that the actual codecs are licensed differently. The GraphViz codec is licensed under the GNU Lesser Public License, whereas the GXL codec is under a BSD-style license. The code was moved to the plugin so it can be easily removed in case you cannot use LGPL licensed in your distribution.

## 6. EPS (Postscript)

The EPS plugin is used for generating Postscript output and is based on the Java EPS Graphics2D package at <http://www.jibble.org/epsgraphics/>. The plugin adds an action to the File->Export menu to export the selection diagram as an EPS file using the `epsgraphics.jar` file. To produce the EPS file, the following code is used to paint a graph object onto a `EPSGraphics2D` object:

```

EpsGraphics2D graphics = new EpsGraphics2D();
[...]
graph.paint(graphics);
out.write(graphics.toString().getBytes());
  
```

## 7. JGX (JGraphpad 5.x)

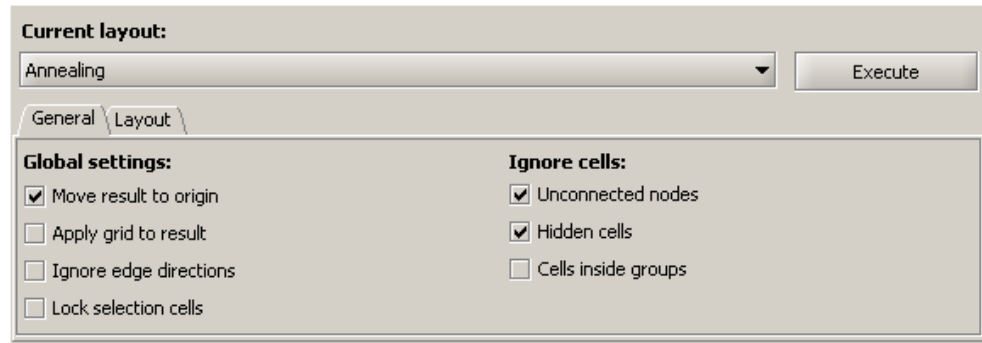
The JGX plugin provides a way of reading files created with the community version of JGraphpad into JGraphpad Pro. The code is based on a modified version of the community edition file parser. This plugin is a standalone plugin with no external dependencies. The action is added to the File->Import menu, however, the functionality is expected to be rarely used and removed over time.

## 8. L2FProd (UI Components)

The `l2fprod` plugin is used to implement the property sheet, replace the default font dialog and the tabbed pane that contains the libraries by an Outlook-style "stacked" pane. The plugin requires the `l2fprod-common-all.jar` file and replaces the font action to use the new dialog. The library pane factory method is replaced to use the new tabbed pane, and the property sheet is added to the bottom tab using the extension mechanism as explained above. Note that the `l2fprod-common-all.jar` or at least the `l2fprod-common-sheet.jar` (not shipped) is also required for the layout plugin. The jar files and more details about the project may be obtained from <http://www.l2fprod.com/>.

## 9. Automatic Layout

Figure 5.2. Layout Plugin



The layout plugin provides automatic layout for the diagrams by using the JGraph Layout Pro package, which is located in the `jgraphlayout.jar` file. The plugin adds a bottom tab and a layout action to the View menu.

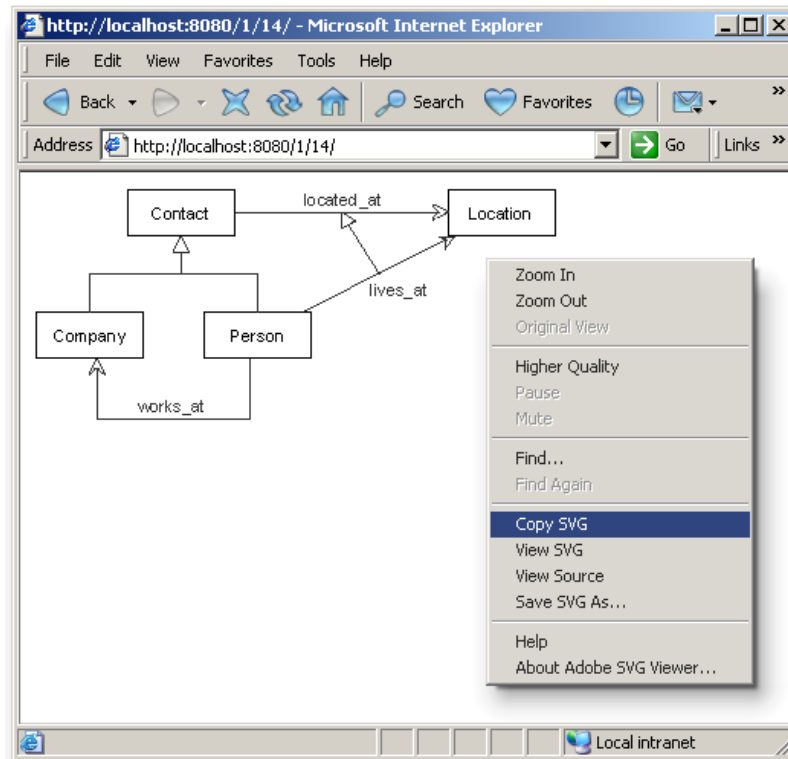
## 10. PDF Export

The PDF plugin adds an export action to the File->Export menu and uses the `itext-1.02b.jar` file from the iText project at <http://www.lowagie.com/iText/> to implement to creation of the PDF files using this code:

```
Document document = new Document();
PdfWriter writer = PdfWriter.getInstance(document, out);
[...]
graph.paint(g2);
[...]
document.close();
```

## 11. SVG Export

Figure 5.3. SVG Plugin



The SVG plugin adds an export action to the File->Export menu and uses the `batik-*.jar` files from the Batik project at <http://xml.apache.org/batik/> to implement the creation of SVG files using the following code:

```
DOMImplementation domImpl = GenericDOMImplementation
    .getDOMImplementation();
Document document = domImpl.createDocument(null, "svg", null);
SVGGraphics2D svgGenerator = new SVGGraphics2D(document);
graph.paint(svgGenerator);
Writer writer = new OutputStreamWriter(out, "UTF-8");
svgGenerator.stream(writer, false);
```

## 11.1. Embedded Webserver

In addition to the export action, the plugin adds an embedded webserver which may be started using the `svgServer` action in the File menu. The webserver is used to showcase streaming image- and SVG data to browsers and is based on the NanoHttpd project at <http://nanohttpd.sourceforge.net/>. The NanoHttpd code is located in a single source file and requires no additional jars.

To use the embedded server, go to File->SVG Server and specify the port to run the webserver on, eg. 8080. Then open a browser and point it to `http://localhost:8080/` which presents you with a list of all diagrams. To get the image or SVG version of a diagram, click on the respective links.

Note that you need the SVG plugin installed for viewing SVG files. The SVG plugin is available free of charge from <http://www.adobe.com/svg/viewer/install/main.html>. However, the embedded server also supports JPG, PNG, GIF and BMP images.

## 12. Summary

Plugins are an easy way of dealing with external dependencies and add-on functionality. The default plugins in JGraphpad are used to import and export files, and to produce various image and vector graphics formats.

Additionally, the plugins are used to define additional language resources, replace UI components with improved versions, or provide access to more complex functionality, such as in the case of the layout plugin.

To remove a plugin, the respective code and/or jar file can be removed from the classpath without having to recompile the application. However, to add a plugin a recompile is required. As an alternative, the list of plugin names may be factored out to an external configuration file.



# Appendix A. Actions

**Table A.1. Actions implemented in the JGraphpadFileAction class**

Menu/Key	Description	Shortcut
<b>File</b>		
open	Inserts a file into the document model	Ctrl-O
download	Inserts a remote file into the document model	Ctrl-W
close	Closes the focused file	
closeAll	Closes all files	
save	Saves the focused file	Ctrl-S
saveAs	Displays a file dialog to save the focused file	Ctrl-Shift-S
uploadAs	Displays a URL dialog to upload the focused file	Ctrl-Shift-U
saveAll	Saves all modified files	Ctrl-Alt-Shift-S
removeDiagram	Removes the focused diagram	
renameDiagram	Renames the focused diagram	
print	Prints the focused diagram	Ctrl-P
pageSetup	Displays the system page setup dialog for the focused diagram	Ctrl-Shift-P
exit	Exits the application	Ctrl-W
<b>File-&gt;New</b>		
newDocument	Inserts a new document into the document model	Ctrl-N
newDiagram	Inserts a new diagram into the focused file	Ctrl-M
newLibrary	Inserts a new library into the document model	Ctrl-L
<b>File-&gt;Export</b>		
saveImage	Saves the focused diagram as an image	
<b>File-&gt;Import</b>		
importCSV	Imports a comma-separated file into the focused diagram	

**Table A.2. Actions implemented in the JGraphpadEditAction class**

Menu/Key	Description	Shortcut
<b>Edit</b>		
edit	Edits the selection cell in the focused diagram	F2
find	Finds the first cell matching an expression in the focused diagram	Ctrl-F
findAgain	Find the next cell matching the expression in the focused diagram	F3
undo	Undos the last change in the focused diagram	
redo	Redos the last change in the focused diagram	
cut	Moves the selection cells from the focused diagram or library to the clipboard	Ctrl-X
copy	Copies the selection cells from the focused diagram or library to the clipboard	Ctrl-C

Menu/Key	Description	Shortcut
paste	Inserts the contents of the clipboard into the focused diagram or library	Ctrl-V
delete	Deletes the selection cells from the focused diagram or library	Delete
<b>Edit-&gt;Select</b>		
selectAll	Selects all cells in the focused diagram	Ctrl-A
clearSelection	Clears the selection in the focused diagram	Ctrl-D
selectVertices	Selects all vertices in the focused diagram	Ctrl-Shift-V
selectEdges	Selects all edges in the focused diagram	Ctrl-Shift-E
deselectVertices	Deselects all vertices in the focused diagram	
deselectEdges	Deselects all edges in the focused diagram	
invertSelection	Inverts the selection in the focused diagram	Ctrl-Shift-I

**Table A.3. Actions implemented in the JGraphpadViewAction class**

Menu/Key	Description	Shortcut
View		
toggleGrid	Shows/hides the grid in the focused diagram	Ctrl-Shift-G
gridSize	Sets the grid size in the focused diagram	
toggleRulers	Shows/hides the rulers in the focused diagram	
toggleRootHandles	Shows/hides the root handles in the focused diagram	
togglePorts	Shows/hides the ports in the focused diagram	
portSize	Sets the portsize in the focused diagram	
toggleDoubleBuffered	Toggle double buffering in the focused diagram	
toggleAntiAlias	Toggle antialiasing in the focused diagram	
toggleDragEnabled	Enables/disables dragging cells in the focused diagram	
View->Library		
libraryLarger	Increases the size of the entries in the focused library	
librarySmaller	Decreases the size of the entrie in the focused library	
View->Background		
togglePage	Shows/hides the background page in the focused diagram	
backgroundImage	Sets the background image of the focused diagram using a local file	
backgroundImageURL	Sets the background image of the focused diagram using a remote file	
background	Sets the background color of the focused diagram	
clearBackground	Clears background of the focused diagram	
View->Zoom		
zoomActual	Shows/hides the background page in the focused diagram	
zoomCustom	Sets the background image of the focused diagram using a local file	
zoomIn	Sets the background image of the focused diagram using a remote file	
zoomOuz	Sets the background color of the focused diagram	

Menu/Key	Description	Shortcut
fitNone	Resets the automatic zoom	
fitPage	Sets the zoom so the background page fits the window	
fitWindow	Sets the zoom so the diagram fits the window	
fitWidth	Sets the zoom so the background page width fits the window	
<b>View-&gt;Options</b>		
tolerance	Sets the selection tolerance in the focused diagram	
editClickCount	Sets the edit click count in the focused diagram	
marqueeColor	Sets the marquee color in the focused diagram	
gridColor	Sets the grid color in the focused diagram	
handleColor	Sets the handle color in the focused diagram	
lockedHandleColor	Sets the locked handle color in the focused diagram	
handleSize	Sets the handle size in the focused diagram	
toggleMetric	Toggles the unit system in the rulers of the focused diagram	
toggleEditable	Enables/disables editing functionality in the focused diagram	
toggleDisconnectOn-Move	Toggles the disconnectOnMove property of the focused diagram	
toggleDisconnectOn-Move	Toggles the disconnectOnMove property of the focused diagram	
toggleMove-BelowZero	Toggles the moveBelowZero property of the focused diagram	
toggleGraphClone-able	Enables/disables cloning in the focused diagram	
toggleGraphMove-able	Enables/disables moving in the focused diagram	
toggleGraphSizeable	Enables/disables sizing in the focused diagram	
toggleGraphBendable	Enables/disables edge bending in the focused diagram	
toggleGraphConnect-able	Enables/disables connecting in the focused diagram	
toggleGraphDiscon-nectable	Enables/disables disconnecting in the focused diagram	
toggleSelectsLocalIn-sertedCells	Toggles the selectsLocalInsertedCells property of the focused diagram	
toggleSelectsAllIn-sertedCells	Toggles the toggleSelectsAllInsertedCells property of the focused diagram	
toggleShowsExisting-Connections	Toggles the showsExistingConnections property of the focused diagram	
toggleShowsInserted-Connections	Toggles the showsInsertedConnections property of the focused diagram	
toggleShowsChanged-Connections	Toggles the showsChangedConnections property of the focused diagram	
toggleHidesExisting-Connections	Toggles the hidesExistingConnections property of the focused diagram	
toggleHidesDangling-Connections	Toggles the hidesDanglingConnections property of the focused diagram	
toggleRemembers-CellViews	Toggles the remembersCellViews property of the focused diagram	

Menu/Key	Description	Shortcut
toggleMovesIn-toGroups	Toggles the movesIntoGroups of the focused diagram	
toggleMovesOutOf-Groups	Toggles the movesOutOfGroups property of the focused diagram	
togglePortsScaled	Toggles the portsScaled of the focused diagram	
toggleJumpsToDe-faultPort	Toggles the jumpsToDefaultPort property of the focused diagram	

**Table A.4. Actions implemented in the JGraphpadFormatAction class**

Menu/Key	Description	Shortcut
<b>Format</b>		
toggleConnectable	Toggles the connectable property of the selection cells	
toggleDisconnectable	Toggles the disconnectable property of the selection cells	
<b>Format-&gt;Position</b>		
toggleCellMoveable	Toggles the moveable property of the selection cells	
toggleGroupReposi-tion	Enables/disables the group repositioning feature for the selection groups	
switchLockX	Locks the X location of the selection cells	
switchLockY	Locks the Y location of the selection cells	
<b>Format-&gt;Size</b>		
resize	Resizes the selection cells	
toggleAutoSize	Toggles the autosize property of the selection cells	
toggleCellSizeable	Toggles the sizeable property of the selection cells	
toggleConstrained	Toggles the constrained property of the selection cells	
switchLockWidth	Locks the width of the selection cells	
switchLockHeight	Locks the height of the selection cells	
<b>Format-&gt;Shape</b>		
switchShapeRect-angle	Switches the shapes of the selection cells to rectangles	
switchShapeRounded	Switches the shapes of the selection cells to rounded rectangles	
switchShapeCircle	Switches the shapes of the selection cells to circles	
switchShapeDiamond	Switches the shapes of the selection cells to diamonds	
switchShapeCylinder	Locks the width of the selection cells	
<b>Format-&gt;Background</b>		
cellBackgroundColor	Sets the background color of the selection cells	
cellGradientColor	Sets the gradient color of the selection cells	
toggleCellOpaque	Toggles the opaque property of the selection cells	
toggleGroupOpaque	Toggles the group opaque property of the selection cells	
<b>Format-&gt;Image</b>		
cellImage	Sets the image of the selection cells	
cellImageURL	Sets the image of the selection cells using a remote image file	

Menu/Key	Description	Shortcut
toggleStretchCellImage	Toggles image stretching on the selection cells	
clearCellImage	Clears the image on the selection cells	
<b>Format-&gt;Border</b>		
cellBorderColor	Sets the border color of the selection cells	
cellBorderWidth	Sets the border width of the selection cells	
cellInset	Sets the inset size of the selection cells	
clearCellBorder	Clears the border of the selection cells	
<b>Format-&gt;Font</b>		
font	Sets the font of the selection cells or text	
fontColor	Sets the font color of the selection cells or text	
fontPlain	Resets the font style of the selection cells or text	
fontBold	Makes the selection cells or text bold	
fontItalic	Makes the selection cells or text italic	
fontUnderline	Underlines the selection text	
<b>Format-&gt;Align</b>		
switchAlignTop	Switches the vertical alignment of the selection cells to top	
switchAlignMiddle	Switches the vertical alignment of the selection cells to middle	
switchAlignBottom	Switches the vertical alignment of the selection cells to bottom	
switchAlignLeft	Switches the horizontal alignment of the selection cells to left	
switchAlignCenter	Switches the horizontal alignment of the selection cells to center	
switchAlignRight	Switches the horizontal alignment of the selection cells to right	
<b>Format-&gt;Label</b>		
switchLabelTop	Switches the vertical label position of the selection cells to top	
switchLabelMiddle	Switches the vertical label position of the selection cells to middle	
switchLabelBottom	Switches the vertical label position of the selection cells to bottom	
switchLabelLeft	Switches the horizontal label position of the selection cells to left	
switchLabelCenter	Switches the horizontal label position of the selection cells to center	
switchLabelRight	Switches the horizontal label position of the selection cells to right	
toggleCellEditable	Switches the horizontal label position of the selection cells to center	
toggleLabelAlongEdge	Toggles the labelAlongEdge property of the selection cells	
switchAlignRight	Switches the horizontal alignment of the selection cells to right	
<b>Format-&gt;Edge</b>		
lineWidth	Sets the linewidth of the selection edges	
lineColor	Sets the linecolor of the selection edges	
dashPattern	Sets the dash pattern of the selection edges	
dashOffset	Sets the dash pattern offset of the selection edges	
toggleEdgeBendable	Toggles the bendable property of the selection edges	
<b>Format-&gt;Style</b>		
switchStyleOrthogonal	Switches the style of the selection edges to orthogonal	

Menu/Key	Description	Shortcut
switchStyleSpline	Switches the style of the selection edges to spline	
switchStyleBezier	Switches the style of the selection edges to bezier	
beginSize	Sets the size of the begin decoration of the selection edges	
clearBegin	Clears the begin decoration of the selection edges	
endSize	Sets the size of the end decoration of the selection edges	
clearEnd	Clears the end decoration of the selection edges	
Format->Routing		
switchRoutingNone	Switches the routing of the selection edges to none	
switchRoutingSimple	Switches the routing of the selection edges to simple	
switchRoutingParallel	Switches the routing of the selection edges to use JGraphpadParallelEdgeRouter	
switchRoutingParallelSpline	Switches the routing of the selection edges to use JGraphpadParallelSplineRouter	

**Table A.5. Actions implemented in the JGraphpadCellAction class**

Menu/Key	Description	Shortcut
Cell		
collapse	Hides the children of the selection groups	Ctrl-E
toggleCollapsed	Toggles the collapsed state of the selection groups	
expand	Shows the children of the selection groups	
expandAll	Shows the children of the selection groups recursively	Ctrl-B
toBack	Sends the selection cells to back	
toFront	Brings the selection cells to front	
group	Inserts the selection cells into a new group	Ctrl-G
groupAsEdge	Inserts the selection cells into a new edge group	
ungroup	Removes the selection groups	
removeFromGroup	Removes the selection cells from their parent groups	Ctrl-U
connect	Connects the selection cells	
disconnect	Disconnects the selection cells	
addProperty	Adds a property to the selection user objects	Ctrl-Shift-C
removeProperty	Removes a property from the selection user objects	
invert	Inverts the selection cells	
Format->Clone		
cloneValue	Clones the value of the first selection cell to the selection cells	Ctrl-Alt-V
cloneSize	Clones the size of the first selection cell to the selection cells	Ctrl-Alt-S
cloneAttributes	Clones the attributes of the first selection cell to the selection cells	Ctrl-Alt-A
Format->Align		
cellsAlignTop	Aligns the selection cells to top left of the selection area	Ctrl-Alt-V
cellsAlignMiddle	Aligns the selection cells to the middle of the selection area	Ctrl-Alt-V
cellsAlignBottom	Aligns the selection cells to the bottom of the selection area	Ctrl-Alt-V
cellsAlignLeft	Aligns the selection cells to the left of the selection area	Ctrl-Alt-V

Menu/Key	Description	Shortcut
cellsAlignCenter	Aligns the selection cells to the center of the selection area	Ctrl-Alt-V
cellsAlignRight	Aligns the selection cells to the right of the selection area	Ctrl-Alt-V
<b>Format-&gt;Selectable</b>		
toggleSelectable	Toggles the selectable property of the selection cells	
toggleChildrenSelectable	Toggles the childrenSelectable property of the selection cells	
allSelectable	Sets the selectable property of all cells to true	

**Table A.6. Actions implemented in the JGraphHelpAction class**

Menu/Key	Description	Shortcut
<b>Help</b>		
about	Displays the about dialog	

---

# Appendix B. Tools

**Table B.1. Implemented Tools**

Key	Description
selectTool	Marquee selection
vertexTool	Rectangular rich text vertex
roundedTool	Rounded rectangle rich text vertex
circleTool	Ellipse rich text vertex
diamondTool	Rhombus rich text vertex
cylinderTool	DB-shape rich text vertex
imageTool	Image plain text vertex
heavyTool	Swing component vertex
edgeTool	Edge with technical end arrow
orthogonalEdgeTool	Edge with technical end arrow and simple routing



---

# Appendix C. Software License

## JGRAPH GENERAL LICENSE STATEMENT AND LIMITED WARRANTY IMPORTANT - READ CAREFULLY

This license statement and limited warranty constitutes a legal agreement ("License Agreement") between you (either as an individual or a single entity) and JGraph Ltd. for the software product ("Software") identified above, including any software, media, and accompanying on-line or printed documentation.

BY INSTALLING, COPYING, OR OTHERWISE USING THE SOFTWARE, YOU AGREE TO BE BOUND BY ALL OF THE TERMS AND CONDITIONS OF THE LICENSE AGREEMENT.

Upon your acceptance of the terms and conditions of the License Agreement, JGraph Ltd. grants you the right to use the Software in the manner provided below.

This Software is owned by JGraph Ltd. and is protected by copyright law and international copyright treaty. Therefore, you must treat this Software like any other copyrighted material (e.g., a book), except that you may either make one copy of the Software solely for backup or archival purposes or transfer the Software to a single hard disk provided you keep the original solely for backup or archival purposes.

You may transfer the Software and documentation on a permanent basis provided you retain no copies and the recipient agrees to the terms of the License Agreement. Except as provided in the License Agreement, you may not transfer, rent, lease, lend, copy, modify, translate, sublicense, time-share or electronically transmit or receive the Software, media or documentation.

If you are not in receipt of the source code of the Software, you acknowledge that the Software is a confidential trade secret of JGraph Ltd. and therefore you agree not to reverse engineer, decompile, or disassemble the Software.

### ADDITIONAL LICENSE TERMS FOR SOFTWARE

JGraph Ltd. grants to you as an individual, a personal, nonexclusive license to install and use the Software for the sole purposes of designing, developing, testing, and deploying application programs which you create. You may install copies of the Software on computers in a manner consistent with the type of license purchased. A Single Developer License may be installed on a computer and be freely moved from one computer to another, providing that you have purchased a number of Single Developer Licenses equivalent to the maximum possible number of developers using that Software concurrently. A Site Developer License may be installed on any number of computers and be used by any number of developers at any time at one geographical location. A geographical location is defined as a building or site occupied by the employees of one company or organization.

If you are an entity, JGraph Ltd. grants you the right to designate one individual within your organization ("Named User") to have the right to use the Software in the manner provided above, in the case of the Single Developer License.

### GENERAL TERMS THAT APPLY TO COMPILED PROGRAMS AND REDISTRIBUTION

You may write and compile (including byte-code compile) your own application programs using the Software, including any libraries and source code included for such purpose with the Software. You may reproduce and distribute, in executable form only, programs which you create using the Software and accompanying Software libraries without additional license or fees, subject to all of the conditions in this License Agreement.

#### ADDITIONAL REDISTRIBUTION TERMS FOR SOFTWARE

You may not distribute any program or file which includes, is created from, or otherwise incorporates portions of the Software if such program or file is a general purpose development tool, library, and/or component, or is otherwise generally competitive with or a substitute for any JGraph Ltd. product.

#### SOURCE CODE

In addition to the license and rights granted, JGraph Ltd. grants you the right to use and modify the SOFTWARE source provided you purchased source code.

You may not distribute the SOFTWARE source code, or any modified version or derivative work of the SOFTWARE source code, in source code form.

The source code contained herein and in related files is provided to the registered developer for the purposes of education and troubleshooting. Under no circumstances may any portion of the source code be distributed, disclosed or otherwise made available to any third party without the express written consent of JGraph Ltd.

Under no circumstances may the source code be used in whole or in part, as the basis for creating a product that provides the same, or substantially the same, functionality as any JGraph Ltd. product.

The registered developer acknowledges that this source code contains valuable and proprietary trade secrets of JGraph Ltd. The registered developer agrees to expend every effort to insure its confidentiality.

SOURCE CODE IS SOLD AS IS. JGRAPH LTD. DOES NOT PROVIDE ANY TECHNICAL SUPPORT FOR SOURCE CODE.

#### MARKETING

JGraph Ltd is permitted to reference you as a user of the Software in customer lists on the JGraph web-site, in presentations to clients and at trade events.

#### LIMITED WARRANTY

JGraph Ltd. warrants that the Software, as updated and when properly used, will perform substantially in accordance with the accompanying documentation, and the Software media will be free from defects in materials and workmanship, for a period of ninety (90) days from the date of receipt. Any implied warranties on the Software are limited to ninety (90) days. Some states/jurisdictions do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you.

This Limited Warranty is void if failure of the Software has resulted from accident, abuse, or misapplication. Any replacement Software will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, JGRAPH LTD. AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NON-INFRINGEMENT, WITH REGARD TO THE SOFTWARE, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM STATE/JURISDICTION TO STATE/JURISDICTION.

LIMITATION OF LIABILITY TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL JGRAPH LTD. OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT

LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF JGRAPH LTD. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

#### HIGH RISK ACTIVITIES

The Software is not fault-tolerant and is not designed, manufactured or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons systems, in which the failure of the Software could lead directly to death, personal injury, or severe physical or environmental damage ("High Risk Activities"). JGraph Ltd. and its suppliers specifically disclaim any express or implied warranty of fitness for High Risk Activities.

#### GENERAL PROVISIONS

This License Agreement may only be modified in writing signed by you and JGraph Ltd. If any provision of this License Agreement is found void or unenforceable, the remainder will remain valid and enforceable according to its terms. If any remedy provided is determined to have failed for its essential purpose, all limitations of liability and exclusions of damages set forth in the Limited Warranty shall remain in effect.

#### GOVERNING LAW AND JURISDICTION

This Agreement shall be subject to and governed by the Law of England and Wales. Any dispute arising out of or in connection with this Agreement shall be exclusively dealt with by the courts of England and Wales. This License Agreement gives you specific legal rights; you may have others which vary from state to state and from country to country. JGraph Ltd. reserves all rights not specifically granted in this License Agreement.

---

# References

- [bib-Benson] <http://www.jgraph.com/>. Benson. *JGraph User Manual*. 2005.
- [bib-Winchester] <http://jdj.sys-con.com/read/37550.htm>. Winchester. *XML Serialization of Java Objects*. 2003.
- [bib-Davidson] <http://www.javadesktop.org/articles/actions/index.html>. Davidson. *An Easy Architecture for Managing Actions*. 2003.
- [bib-JLS] [http://java.sun.com/docs/books/jls/third\\_edition/html/j3TOC.html](http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html). Sun Microsystems. Palo AltoCA. *The Java Language Specification, Third Edition*.
- [bib-Geary] Geary. *Graphic Java 2, Volume II: Swing (3rd Edition)*. Sun Microsystems. Palo AltoCA. 1999.
- [bib-Geary2] Geary. *Graphic Java 2, Volume III: Advanced Swing (3rd Edition)*. Sun Microsystems. Palo AltoCA. 1999.
- [bib-Swing] *The Swing Tutorial*. <http://java.sun.com/docs/books/tutorial/index.html>. Sun Microsystems. Palo AltoCA.
- [bib-JTree] *JTree API Specification*. <http://java.sun.com/j2se/1.4/docs/api/javaw/swing/JTree.html>. Sun Microsystems. Palo AltoCA.
- [bib-GOF] Gamma, Helm, Johnson, and Vlissides. *Design Patterns*. Addison-Wesley. ReadingMA. 1995.
- [bib-Tutorial] Alder. <http://www.jgraph.com/tutorial.html>. *The JGraph Tutorial*. 2002.
- [bib-Paper] Alder. <http://www.jgraph.com/paper.html>. *Design and Implementation of the JGraph Swing Component*. 2002.
- [bib-Java1.4] *The Java 1.4 API Specification*. Sun Microsystems. Palo AltoCA.
- [bib-API] <http://www.jgraph.com/doc/jgraph>. Alder. *The JGraph API Specification*.
- [bib-Alder] Alder. *JGraph. Semester Work*. Department of Computer Science, ETH. Zürich. 2001.
- [bib-Alder2] Alder. *JGraph. Diploma Thesis*. Department of Electrical Engineering, ETH. Zürich. 2002.