The JGraph Swing Component

Diploma thesis by Gaudenz Alder

November 2001 – March 2002

Department of Computer Science
Federal Institute of Technology ETH, Zurich, Switzerland
Supervisors: Prof. Bernhard Plattner (TIK, ETHZ)
Prof. Gerhard Tröster (IfE, ETHZ)

To Ana, who makes it all worthwhile.

Preface

This dipoma thesis is based on my semester work at the Global Information Systems group. In this semester work, a first implementation of the JGraph component and the JGraphpad example application was developed, and published on the Internet as an open-source project.

The diploma thesis is based on this first release, and the numerous feature requests, bug reports, comments and suggestions that were submitted by the Java community in reply. It turned out that one of the biggest problems was the lack of a *detailed* documentation, and that some parts were not fully Swing compatible.

The submissions were used to identify other weak parts of the package, and define the final architecture and set of features. Men Muheim made important contributions to this final design with his experiments on using JGraph as a UI to his Audio Comodity System (CAOS). This application was used to define the requirements of the final architecture, and additional features:

- Separate and shared attributes for each view
- Multiview command history support
- Stepping-into groups for selection
- Unified GraphModel and ports

Thus, this diploma thesis consists of the redesign of the JGraph component and implementation of new features, and the documentation, which contains a paper and a tutorial. The API Specification is also part of the documentation, but it is not included in this printed version of the thesis.

The rest of this document contains two parts:

- Part I: Design and Implementation of the JGraph Swing Component
- Part II: The JGraph Tutorial

The CD accompanying this document includes the JGraph Source code and Binaries, the JGraphpad example application and a Mirror of the JGraph Home Page, which includes additional examples, the API specification and the paper and tutorial in electronic form.

Zürich, March 2002

Gaudenz Alder

Acknowledgments

The following people and groups have made the JGraph Project possible:

Thanks to Prof. Moira Norrie, Prof. Bernhard Plattner and Prof. Gerhard Tröster for their support.

Beat Signer from the Global Information Systems Group was instrumental in helping to get this project off the ground. He arranged that JGraph could be handed-in as a semester work.

The new design was strongly influenced by Men Muheim's experiments with JGraph in another project. Thanks to Men for the redesign, and willingness to accept this work as a diploma thesis.

Christophe Avare translated JGraphpad to French. Thomas Suter, Lars Gersmann, Markus Schmidt, Antonio Caliano, Martina Huber and Andri Krämer suggested new features and read the drafts.

Special thanks to my parents, Sue and Tis, with love and admiration.

Part I

Design and Implementation of the JGraph Swing Component

Abstract

Today's user interface (UI) libraries offer components for lists, tables, and trees, but graph components are rarely implemented. JGraph provides a fully standards-compliant graph component for the Java Swing UI library that supports extended display and editing options.

This paper provides provides a description of the JGraph component. The document is structured into modules, and illustrated using UML and other diagrams. The study will outline JGraph's design and implementation with focus on Swing compatibility, and explain where it was necessary to extend Swing design.

The target readers are practitioners who need a definition of JGraph's architecture. The document provides an in-depth discussion of the component's design and implementation, and thoroughly explains the ideas and concepts behind the architecture.

Table of Contents

1 Introduction

After a brief overview of the fundamentals of graph theory and Swing, this introduction outlines the structure of the document, and the literature that was used. For additional information, updates, binaries and source code see the Home Page of JGraph at http://www.jgraph.com.

1.1 Overview

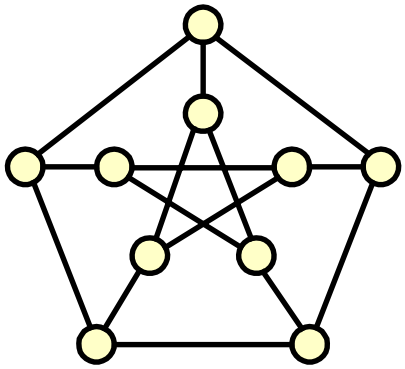

Fig. 1: The Peterson graph

The intention of this project is to provide a freely available and fully Swing compliant implementation of a graph component. As a Swing component for graphs, JGraph is based on the mathematical theory of networks, called *graph theory*, and the *Swing* user interface library, which defines the architecture. By combining these two, a Swing user interface component to visualize graphs will be obtained. (The term *graph* is used in the sense of graph theory throughout this document, not to be confused with function plots.)

The following principles guided the implementation of JGraph:

- Full Swing Compatibility
- Clear & Efficient Design
- Short Download Time
- 100 % Pure Java

The idea behind these principles is based on the experience with other graph libraries, which typically come as large and complex frameworks that are not standards compliant.

In JGraph instead, the basic architecture is the same as for standard Swing components, and the method and variable naming complies with Java code conventions. This has the advantage of reduced learning costs, and existing source code can be reused, resulting in shorter development time.

1.1.1 Graph Theory

The concept of a *graph* is based on a well-founded mathematical theory, called graph theory, which is rigorously defined in [1], [2] and [3]. A graph consists of *vertices* and of *edges* connecting certain pairs of vertices. The exact geometric pattern is not specified.

As a mathematical generalization of lists and trees, graphs come into play where lists and trees are not sufficient to model the relations. For example, in a tree each *node* has at most one *parent*, and zero or more *children*, whereas in a graph, each *vertex* simply has zero or more *neighbors*. (In the case of the list, each node has at most two neighbors.)

The single term *neighbor* to emphasize the more general setup replaces the terms *parent* and *child* used in the context of trees. The term *cell* is used in place of *node* throughout this document to distinguish between the elements of graphs, and those of lists and trees.

Formally, a graph G consists of a non-empty set of elements $V(G)$ and a subset $E(G)$ of the set of unordered pairs of distinct elements of $V(G)$. The elements of $V(G)$, called vertices of G, may be represented by points. If (x, y) $\in E(G)$, then the edge (x, y) may be represented by an arc joining x and y. Then x and y are said to be *adjacent*, and the edge (x, y) is *incident* with x and y. If (x, y) is not an edge, then the vertices x and y are said to be *nonadjacent*.

Graph theory also offers standard algorithms to solve common graph problems. For example, the *Dijkstra* algorithm can be used to find the shortest path between two vertices of a graph. The Dijkstra algorithm, and many other standard algorithms are explained in [4], [5] and [6]. (An implementation of the Dijkstra algorithm ships with the JGraphpad example application.)

To summarize, graphs are used as a paradigm in JGraph to display any network of related objects. A road or computer network, a molecule, software architecture, or database schemas are examples of graphs. Since graphs are a mathematical generalization of lists and trees, JGraph may also be used to display simpler structures, as for example a file system tree.

1.1.2 Swing

Swing is the user interface library that ships with Java, and provides the elements that can be placed on an application’s windows. Such elements are called user interface components, or simply *components*. Common components are buttons, lists, trees and tables.

Swing is based on AWT, which stands for Abstract Windowing Toolkit. AWT is another user interface library that ships with Java and may be used independently of Swing. In contrast to AWT, Swing is more sophisticated, and provides many built-in features.

A good introduction to Java and AWT is given in [7] and [8]. [9] and [10] are for the more experienced programmers. In contrast to AWT, Swing is only explained in [11] and [12], and there is a good web-based Swing Tutorial [13].

The documents mentioned before discuss existing components in great detail, but do not explain how to create new components. (Not to think of an in-depth study of *Swing MVC*, which is the basic architecture that all Swing components have in common [17].) Swing's source code must be used to elaborate the required information.

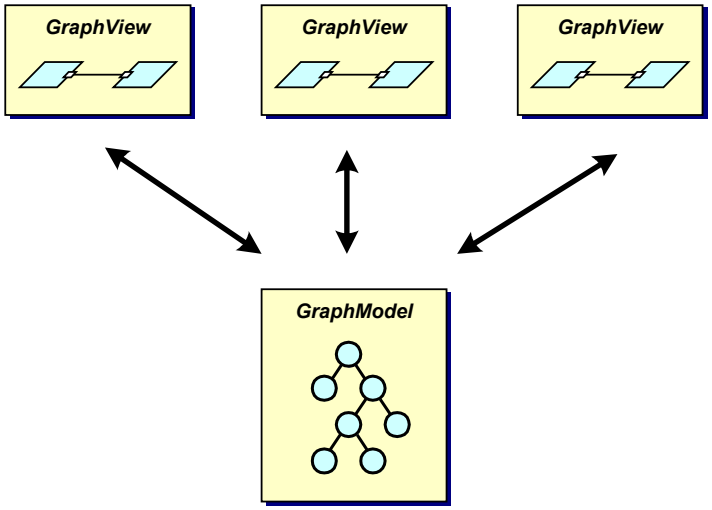

Fig. 2: Model-View-Control (MVC)

In a nutshell, the *model* provides the data, which is a graph – in the sense used in graph theory – and the JGraph object provides a *view* of the data, in platform specific manner. The Swing MVC design pattern is used 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.

JGraph is largely based on Swing's component for trees, called JTree, which is explained in [14]. Some ideas were adopted from Swing's text components, namely the Element interface, which is discussed in [15], and the views [16]. The JGraph class itself is an extension of `JComponent`, which is Swing's base class for all components.

JGraph complies with all of Swing's standards, such as pluggable look & feel, datatransfer, accessibility, internationalization, and serialization. Also, all concepts used in JGraph are common, well-known concepts from Swing. For the more sophisticated features such as undo/redo, printing, and XML support, additional Swing standards were used. Aside from Swing's standards, JGraph also complies with the Java conventions for method and variable naming, source code layout, and javadoc comments.

1.2 Decomposing JGraph

The decomposition of JGraph into modules splits the overall definition into a series of tractable subdefinitions. By following the MVC pattern, the framework is split into a *model*, *view* and *control* part. These chapters are preceded by an overview of the component.

The *component overview* compares JGraph to JTree, and outlines the main differences and consequences on the API. It further discusses the API with focus on the MVC pattern, and on multiple views.

The *model* part describes the underlying graph model interface, and selection model, and the elements that they contain; as well as the classes used to change the graph model.

The *view* part studies the display's internal representation of the graph, and the mapping and update between the model and the view. Special focus is given to the *geometric pattern* and *context* of graphs.

The *control* part explains the rendering process, the steps involved in in-place editing and cell handling, and the objects involved in datatransfer and marquee selection.

The *event model* is explained in an additional chapter, outlining the update and undo mechanisms, and the dynamic aspects of the system.

The final chapter summarizes the main results, draws conclusions about the current implementation, and discusses possible future enhancements.

1.3 Sources and Literature

The JGraph component is largely based on [14]. Some ideas are used from [15]. The description of design patterns uses the naming from [17].

The standard book about Swing is certainly [11] and is a good basis to understand this document. Some implementations of graph specific algorithms are based on [4] and [5]. Also, the reader should be familiar with the most important concepts of graph theory, which can be obtained from [1].

In addition to this paper, a tutorial [18], and the API specification [20] can be found on the JGraph Home Page at http://www.jgraph.com.

2 The JGraph Component

The implementation of JGraph is entirely based on the source code of the JTree [14] class, which is Swing's component for displaying trees. Rather than explaining JGraph from scratch, this description focuses on the differences between the two classes. Swing features such as serialization, datatransfer, and the Swing MVC pattern are explained in the appendix.

2.1 Foundation

JGraph is *not* an extension of JTree; it is a modification of JTree's source code. In the following, the changes are briefly outlined, pointing to the classes and methods that were modified or introduced. The modifications are grouped into:

1. Differences between trees and graphs
2. JGraph requirements (features)

The following important differences between trees and graphs lay the foundation of the JGraph component:

2.1.1 Overlapping Cells

In a tree, the position and visibility of a cell depends on the expansion state. In a graph, instead, the position and size of a cell is user-defined, possibly *overlapping* other cells. Consequently, a way to enumerate the cells that intersect the same position, and to change the order in which they are returned is provided.

2.1.2 Multiple Cell Types

A tree consists of nodes, whereas a graph possibly consists of *multiple cell types*. Consequently, JGraph provides a view for each cell, which specifies the renderer, editor and cell handle. Additionally, a graph view is provided, which has its own internal representation of the graph. The idea of views has been adopted from Swing's text components [16].

2.1.3 Separate Geometry

The mathematical definition of a graph does *not* include the geometric pattern or layout. Consequently this pattern is not stored in the model, it is stored in the view. The view provides a notification mechanism, and undo-support, which allows changing the geometric pattern independently of the model, and of other views. Since Swing's undo manager is not suitable for this setup, JGraph provides an extension of this Swing's default undo mechanism in the form of the `GraphUndoManager` class.

2.2 Features

The changes to meet the requirements, or features, may be grouped into:

1. *Inheritance*: JTree's implementation of pluggable look and feel support and serialization is used without changes.
2. *Modification*: The existing implementation of in-place editing and rendering was modified to work with views, and history.
3. *Extension*: JGraph's marquee selection and stepping-into groups extend JTree's selection model.
4. *Enhancement:* JGraph is enhanced with datatransfer, attributes and history, which are Swing standards not used in JTree. (A special history must be used in the context of separate geometric patterns.)
5. *Implementation*: The layering, grouping, handles, cloning, zoom, ports and grid are new features, which are standards-compliant with respect to architecture, and coding conventions.

2.2.1 Inheritance

The pluggable look and feel and serialization are used without modifications. As in the case of JTree, the UI-delegate implements the current look and feel, and serialization is based on the `Serializable` interface, and `XMLEncoder` and `XMLDecoder` classes for long-term serialization.

2.2.2 Modification

The *multiple cell types* property affects the rendering and the in-place editing of JGraph. In contrast to JTree, where the renderer and editor for the single node type is referenced from the tree object, the editors and renderers in JGraph are referenced from the cell view, to avoid a look-up of the renderer via the cell's type. The renderer is statically referenced so it can be shared among all instances of a class. Additionally, the cell view specifies a *handle* that allows extended editing. The idea of handles closely follows the design used for *in-place editing*.

2.2.3 Extension

Like the JTree selection model, JGraph's selection model allows single and multiple cell selection. JGraph's selection model additionally allows to *step-into* groups. Marquee selection is also provided using a m*arquee handler*, which is based on the design of Swing's t*ransfer handler*.

2.2.4 Enhancement

JGraph allows transferring the cells of a model, together with a description of their group and graph structure, and their geometric pattern either by using drag-and-drop, or via the clipboard.

Based on an idea of Swing's text components, JGraph provides *maps* to describe and change the attributes of a cell. These maps encapsulate the state of the cell, and may be accessed in a type-safe way using the `GraphConstants` class.

JGraph provides *command history*, or *history*, which is the ability to undo or redo changes. The design follows the design of Swing's text components; however, a special undo manager must be used in the context of separate geometric patterns.

2.2.5 Implementation

There are two groups of features in the implementation chapter:

1. Features that only affect the JGraph class
2. Features that require new classes

The first group consists of the *cloning*, *zoom*, and *grid*, which are implemented by methods in the JGraph class. The rest of the framework does not offer classes or methods to implement these features, however, it is *feature-aware*, which means it relies on the respective methods of the JGraph class.

The second group consists of the *layering*, *handles*, *grouping*, and *ports*, which are not used elsewhere in Swing, and require new classes and methods. Special care has been taken to base these features on existing Swing functionalities, and make them analogous with regard to design and implementation. For example, the handles feature closely follows the design and implementation of Swing's in-place editing, so that it is easy for the programmer to adopt this new feature based on his or her understanding of in-place editing.

Because these features are new, some of them are briefly defined below:

- *Layering:* Since cells may overlap, the order in which they are returned is significant. This order is referred to as Layering, and may be changed using the `toBack` and `toFront` method of the `GraphView` object. The layering is part of the view, and is explained in the view part of this document.
- *Handles:* Handles are, like editors, objects that are used to change the appearance of a cell. In contrast to in-place editing, which uses a text component to change the value of a cell, handles use other means to provide the user with a visual feedback of how the graph will look after the successful execution of the change (live-preview). Handles and in-

place editing are explained in the control part, because the UI-delegate provides this functionality.

- *Grouping and Ports:* Ports and grouping are related because ports are implemented on top of the group structure in the graph model. The grouping is therefore explained in the model part of this document.

2.3 JGraph MVC

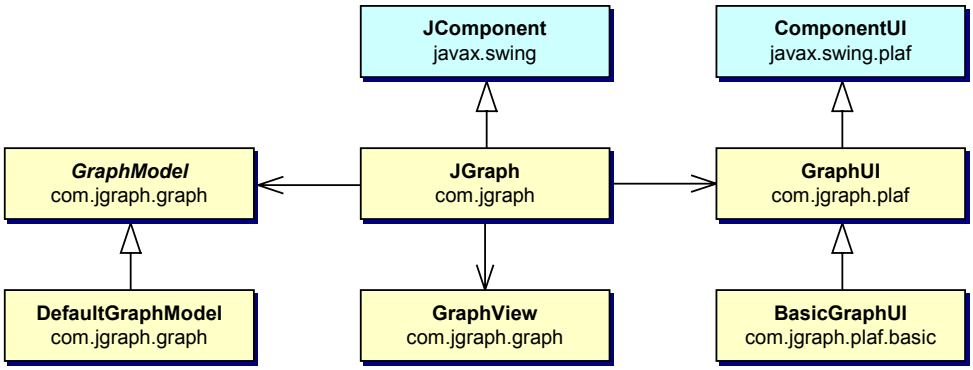

Fig. 3: JGraph MVC

JGraph extends `JComponent`, and has a reference to its `GraphUI`. JGraph has a reference to a `GraphModel` and a `GraphView`, and instantiates `BasicGraphUI`, which extends `GraphUI`, which in turn extends `ComponentUI`.

The basic structure of the component, namely the Swing MVC architecture, is inherited from JTree. However, JGraph has an additional reference to a graph view, which is not typically used in Swing MVC. The graph view is analogous to the root view in Swing's text components, but it is not referenced by the UI-delegate. Instead, it is referenced by the JGraph object such that it preserves the state when the look-and-feel is changed. (The appendix provides an in-depth discussion of MVC, Swing MVC, and how it is applied to JTree and JGraph.)

2.4 Startup

When working with attributes (see 2.5), the startup-sequence is significant. The fact that the default model does not store attributes must be taken into account when inserting cells, because the attributes of such cells are passed to the attached views. If no views are attached, the attributes are ignored!

In the case where a view is added later, the view uses default values for the cell's positions and sizes, resulting in the fact that each cell is located at the same point, and has the same size.

Therefore, when creating a graph with a custom model, first the JGraph instance should be created, using the model as an argument, and then, cells should be inserted into the model (not vice versa). By constructing the JGraph instance, a view is automatically registered with the model.

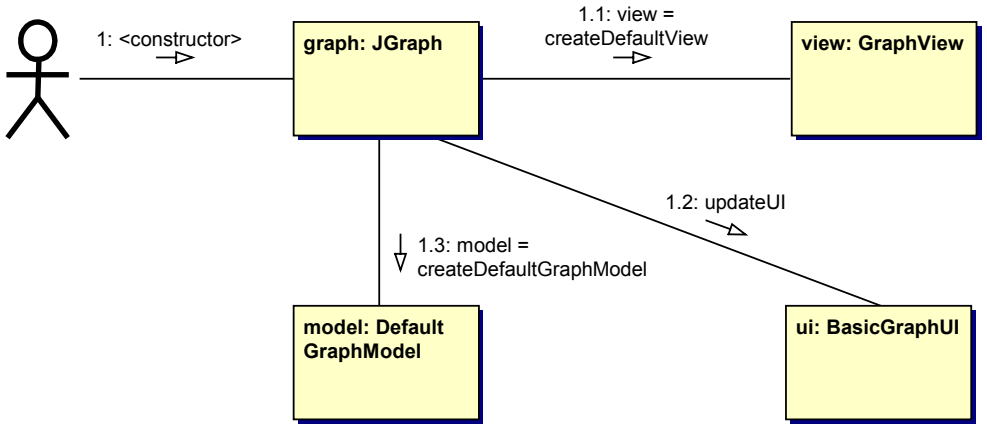

Fig. 4: JGraph's default constructor

The same holds for setting the model on a JGraph object, in which case the view is notified, and holds a reference to the new model. Anyway, because the model does not store the attributes, the view will use default values as in the case where it is registered with the model *after* an `insert` call.

The above does not hold if the model's `isAttributeStore` returns `true`, in which case all attributes are stored in the model instead of the view, making the timing issues irrelevant.

2.5 Attributes

JGraph's attributes are only *conceptually* based on those of Swing, some dynamic aspects, and the class to access these attributes are different from Swing, and must therefore be explained.

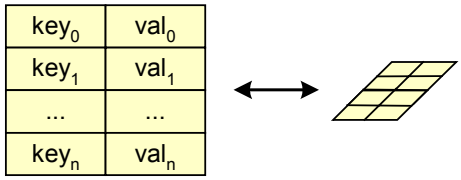

Fig. 5: Symbol used for attributes

Attributes are implemented using maps, from keys to values, and may be accessed by use of the `GraphConstants` class.

2.5.1 GraphConstants

The `GraphConstants` class is used to create maps of attributes, and to access the values in these maps in a type-safe way. Aside from the creation of, and the access to maps, the class also provides a method to clone maps, and a method to apply a change of more than one value on a target map.

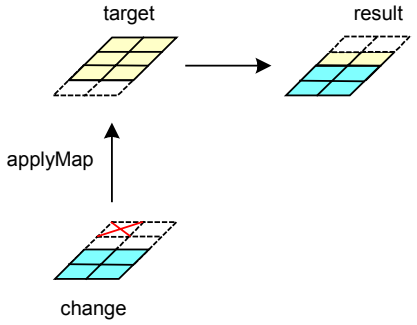

Fig. 6: Changing attributes with the GraphConstants class

The `applyMap` method combines common entries from the target map and the change map by overriding the target values. Entries that are only present in the change map are inserted into the target map, and entries that are only present in the target map are left unchanged.

To remove entries from the target map, the `setRemoveAttributes` method is used, providing the keys that should be removed as an argument. The keys are stored as an entry in the change map, and handled by the `applyMap` method. If the change map replaces the target map completely, then the `setRemoveAll` method must be used on the change map to indicate that all keys of the target map should be removed.

Attributes may be used in the model *and* in the view. In both cases, the attribute maps are created and accessed by use of the `GraphConstants` class. The relation between a cell's attributes and its corresponding view's attributes is as follows:

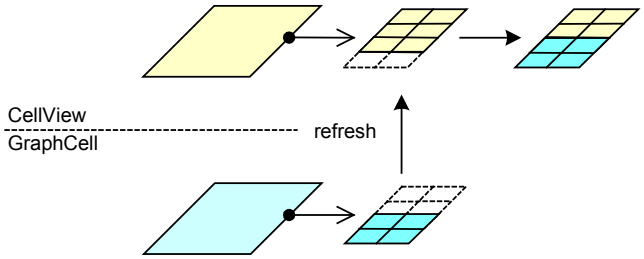

Fig. 7: Relation between a cell's and its view's attributes

A cell's attributes have precedence over its view's attributes, such that the cell can override the view's values for a specific key with its own value. This means, if a view specifies a value for a key that is also specified by the cell, then the cell's value is used instead of the view's value. (Two attributes are equal if the `equals` method on their respective keys returns `true`.)

In other words, the blue attributes have precedence over the yellow ones, and if a blue attribute is not present in the yellow map, then it will be inserted. Yellow entries that do not exist as blue entries are left unchanged. (Since this mechanism is based on the `applyMap` method, the behavior is exactly the same.)

The `GraphConstants` class does not distinguish between the attributes for *cells* and *views*, because they are based on the same underlying structures. However, in contrast to cells, which accept all attributes, the view performs an additional test on each key. The view's renderer is used to determine if the key is supported, and if not, the entry is removed from the corresponding view's attribute map in order to reduce redundancy. (Both `setAttributes` methods, for cells and views, are based on the `applyMap` method.)

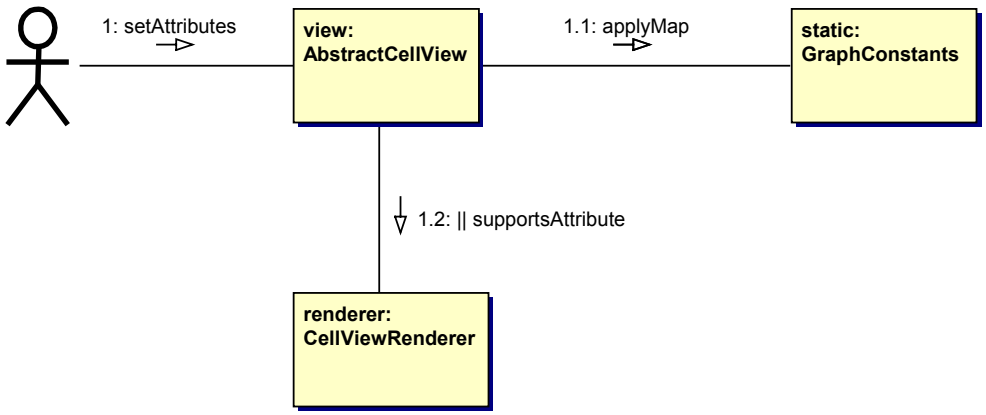

Fig. 8: Implicit use of the GraphConstants class

In JGraph's default implementation, the UI changes the *view's* attributes upon interactive changes. By overriding the model's `isAttributeStore` method, the model can gain control. If the method returns `true`, then the *cell's* attributes will be changed instead of the view's, resulting in an immediate update of *all* attached views. In the default case, only the local view is updated (unless the change constitutes a change to the model).

This is because *all* views are updated upon a change of the model by the model's notification mechanism, and the cell's attributes are used in all views. An exception is the value attribute, which is in sync with a cell's user object. The value attribute is stored in the cell regardless of the model's `isAttributeStore` method.

2.5.2 The Value Attribute

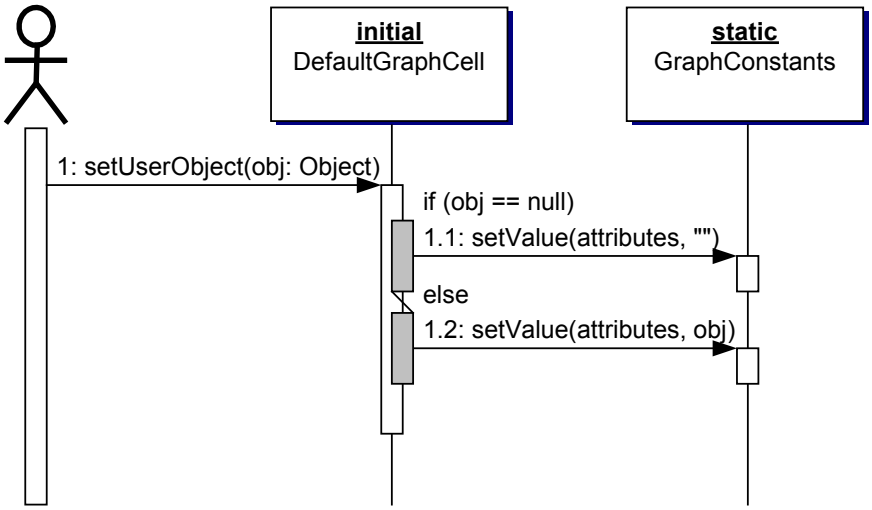

Fig. 9: Synchronization upon change of a cell's user object

The `setUserObject` method of the `DefaultMutableTreeNode` class is overridden by the `DefaultGraphCell` class such that the passed-in user object is stored under the `value`-key in the cell's attribute map. Vice versa, the `setAttributes` method overwrites the previous user object if a new object for the `value`-key is specified. Thus, the value attribute points to the user object and vice versa.

The value attribute is used in the context of history and in-place editing. By introducing the value attribute, the complete state of the cell may be represented by its attributes; the user object does not require special handling. A change to the state (and equally to the user object) may be undone using the `applyMap` method, providing the previous and current states as arguments.

By default, in-place editing replaces the user object with the new String, which is not always desirable. Therefore, the user object may implement the `ValueChangeHandler` interface, which changes this default behavior.

2.5.3 ValueChangeHandler Interface

The `ValueChangeHandler` interface, which is an inner interface of the `DefaultGraphCell` class, may be used to prevent the user object from being overwritten upon in-place editing. If a user object implements this interface, then the `DefaultGraphCell` informs the user object of a change, the latter of which is responsible for storing the new, and returning the old value. The user object reflects the change through its `toString` method.

(JGraph's `convertValuetoString` method is used to convert a *cell* to a `String`.)

2.5.4 Attributes and Instance Fields

Should new properties be implemented as attributes or as instance fields? In the first case, the cloning of the cell and the undo mechanism are already in place, but the attribute must be accessed through a hash table. In the second case, the cloning of the cell requires special handling, but the variable may be accessed as an instance field, which might be required for inheritance. The combination of the two leads to an increase of redundancy, and complexity. (An example is the value attribute from above.)

Basically, attributes should only be used for rendering, even if there are no technical restrictions for storing custom attributes in a cell. Since only supported attributes are propagated to the view, such custom attributes add no redundancy with respect to the view's attributes. (The value attribute is an exception that is an instance field, *and* is supported by all renderers.)

Instead of extending the `DefaultGraphCell` class or using the attributes to store additional information, the class, which inherits from JTree's `DefaultMutableTreeNode` also provides another mechanism to store user-defined data, namely through the user object. The user object is of type `Object`, and therefore provides a way to associate any object with a `GraphCell`.

The user object may be in an arbitrary class hierarchy, for example extending the `Hashtable` class. Since the user object's `toString` method is used to provide the label, this method should probably be overridden.

2.6 Cloning

A new feature in JGraph is the possibility to clone cells automatically. This feature is built into the default implementation's clipboard and cell handles, and is based on the `clone` method of the `Object` class, and on JGraph's `cloneCells` method. The feature may be disabled using the `setCloneable` method on the JGraph object. (By disabling this feature, a Control-Drag will be interpreted as a normal move.)

The process of cloning is split into a *local* and a *global* phase: In the local phase, each cell is cloned using its `clone` method, returning an object that does not reference other cells. The cell and its clone are stored in a hash table, using the cell as a key and the clone as a value.

In the global phase, all cell references from a clone's original cell are replaced by references to the corresponding clones (using the before mentioned hash table). Therefore, in the process of cloning cells, first all cells are cloned using the `clone` method, and then all cell references are consistently replaced by references to the respective clone.

2.7 Zoom and Grid

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

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

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

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

To support the grid, each point that is used in the graph must be applied to the grid using the `snap` method. As in the case of zooming, the `snap` method changes the argument in-place instead of cloning the point before changing it. This is because instantiation in Java is expensive, and it is not always required that the point is being cloned before it is changed. Thus, the cloning of the argument is left to the client code.

JGraph provides two additional bound properties that belong to the grid: one to make the grid visible, and the other to enable the grid. Thus, the grid can be made visible, but still be disabled, or it can be enabled and not visible.

3 Model

The model provides the data for the graph, consisting of the cells, which may be vertices, edges or ports, and the connectivity information, which is defined using *ports* that make up an edge's source or target. This connectivity information is referred to as the *graph structure*. (The geometric pattern is not considered part of this graph structure.)

3.1 Grouping

The model provides access to another structure, called the *group structure*. The group structure allows the cells of a graph to be nested into part-whole hierarchies, that is, it allows composing new cells out of existing ones. The following graph contains such a group, which in turn contains the vertices A and B, and the edge 1. The vertex A in turn contains the port *a* is a child, and the vertex B contains the ports b_0 and b_1 as its children.

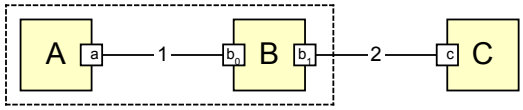

Fig. 10: A graph with a group

Since the group structure adds a dimension to the graph model, the above figure is not suitable to visualize both structures, even if the graph alone is fully defined. (The group in this figure is drawn using a dashed rectangle, but in reality, it is typically not visible.)

To illustrate the underlying group structure, a three-dimensional figure is used that depicts the graph structure in the X-Y-plane, and the group layers shifted along the z-axis.

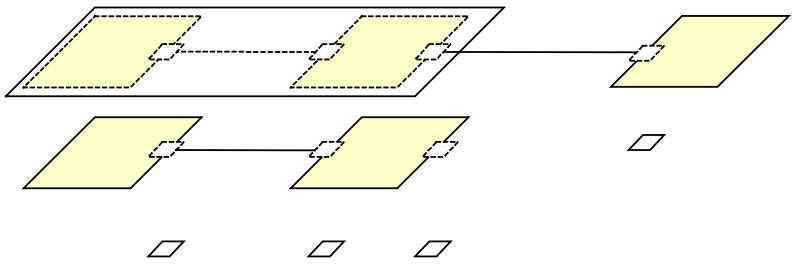

Fig. 11: Group layers of the graph

The *group layer diagram* reveals the grouping in the graph. Dashed rectangles are used to illustrate visibility of cells on upper layers; the actual

cell instances “live” on lower layers. Note that ports are part of this group structure, and appear as children of their vertices.

To illustrate the group structure alone, another diagram is used. In this diagram, the groups appear as trees, with their roots stored in a linked list. The elements are drawn as circles to underline the different purpose and content of the diagram. The empty cell at the root of A, 1 and B is group that has no label. It is not possible to deduce the original graph from the *group structure diagram*.

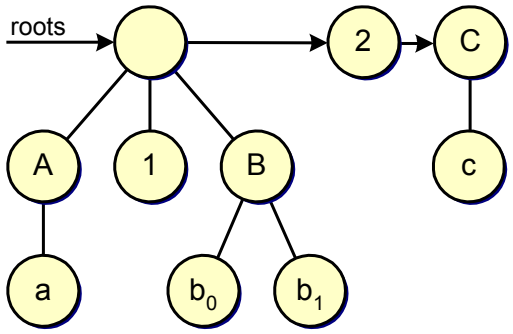

Fig. 12: Group structure of the graph

The group structure can be thought of as a forest of trees, where the roots of these trees may be retrieved using the `GraphModel` interface. Once these roots have been retrieved from the model, their descendants may be retrieved using the respective methods of the `GraphModel` interface.

Logically, the group structure can be seen as an extension of the `TreeModel` interface. However, such an extension is technically impossible because the `TreeModel` interface relies on `TreeModelListeners`, which are not used in the context of graphs.

An important fact that should be kept in mind is that the source and target of an edge implement the `Port` interface, which is used as an indirection to provide multiple connection points for the same vertex. The relation between the port and the vertex is implemented on top of the group structure, the vertices being the parents of the ports. (Ports are somewhat artificial, and from a strictly mathematical point of view, do not belong to the definition of a graph.)

To summarize, the graph model not only provides access to the graph structure, it also provides access to an independent structure, called group structure, which allows cells to be nested into part-whole hierarchies. The group structure can be seen as a set of trees, where the roots of the trees represent the parents of the groups. (If a virtual node is added as a parent to these roots, then the model itself is a tree.) The graph structure instead provides the methods to retrieve the source and target port of an edge, and to return the edges that are connected to a port, or to an array of cells.

3.2 Graph Cells

Graph cells are the key ingredients of JGraph, which provides default implementations for the most common cells: vertices, edges and ports. The vertex is considered as the default case, which does not need an interface, and uses the default implementation of the common superclass. Ports do not have to be treated as special children, such that their relation to the vertex can be implemented on top of the group structure, provided by the `GraphModel` interface.

The above only holds for the model part of the framework, the view has a different internal representation of the graph. In the view's data structure, children and ports are treated as different entities, and are kept in different lists. One of the reasons for this is that ports (if visible) do always appear in the front of the graph, regardless of their order, and the other reason is better performance.

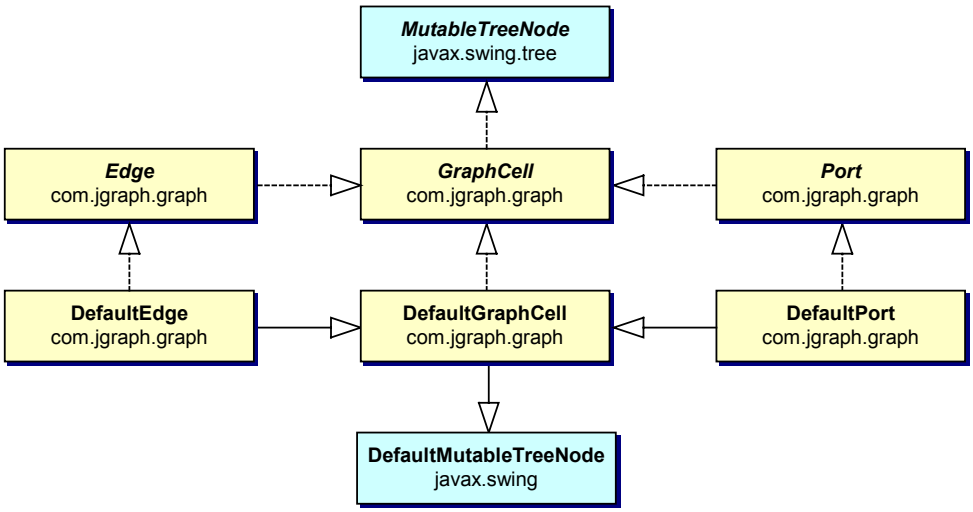

Fig. 13: GraphCell interface hierarchy and default implementations

`DefaultPort` and `DefaultEdge` extend `DefaultGraphCell`, `Port`, and `Edge`, respectively, which in turn extend `GraphCell`. `DefaultMutableTreeNode` has a reference to a `TreeNode` that represents the parent, and an array of `TreeNodes` that represents the children. `DefaultEdge` has a reference to the source and target `Object`, which typically extend `Port`. `DefaultPort` has a reference to a `Port` that represents the anchor, and instantiates a set that holds the connected edges.

3.2.1 GraphCell Interface Hierarchy

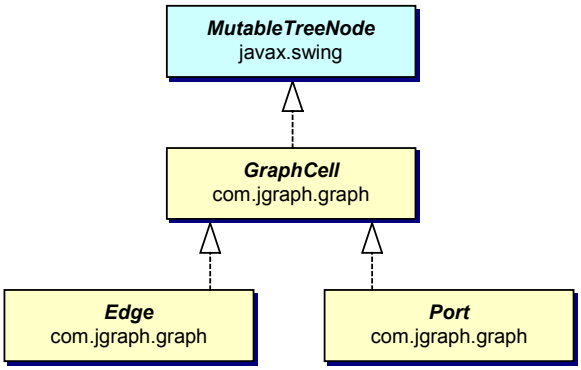

Fig. 14: GraphCell interface hierarchy

The `GraphCell` interface is an extension of Swing's `MutableTreeNode` interface, which provides the methods for the `DefaultGraphModel` to store its internal group and graph structure. The `GraphCell` interface itself provides methods to access the attributes of a cell, whereas the methods to access the children and parent are inherited by the `MutableTreeNode` interface.

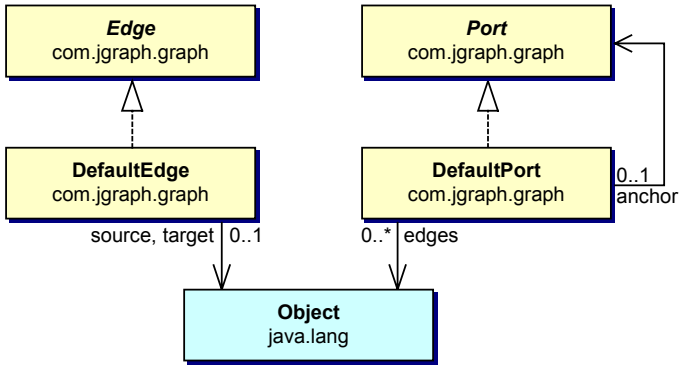

Fig. 15: The graph structure is stored in the edges and ports

The `Edge` interface provides the methods to access the source and target port, and the `Port` interface provides access to all edges attached to a specific port. Thus, the graph structure only relies on the `Edge` and `Port` interfaces.

With regard to text component analogies, the `GraphCell` interface is analogous to the `Element` interface [15], whereas the `CellView` interface is JGraph's analogy to the text component's `View` interface [16].

3.2.2 GraphCell Default Implementations

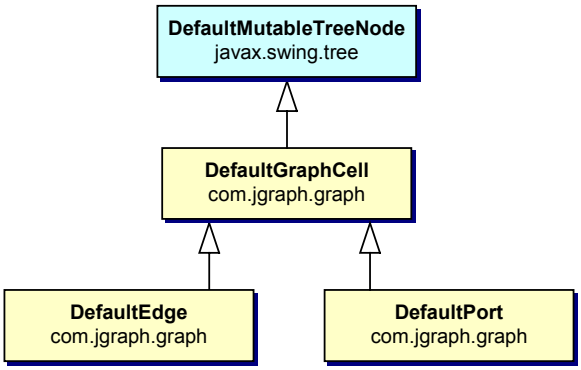

Fig. 16: GraphCell default implementations

The abstract base class, called `DefaultGraphCell`, provides the functionality that is common to all graph cells, namely for

- Cloning the cell and its children
- Handling attributes
- Synchronizing the value attribute and the user object

The `DefaultGraphCell` class, which is the common superclass of all graph cells, is an extension of `DefaultMutableTreeNode`. As a consequence, all cells in JGraph may be nested into groups.

The default implementation makes no restriction on cells being used as groups or not, and what parents and children are accepted. The programmer must ensure that only groups can be composed that make sense with respect to the application. Typically, vertices are used as groups. A vertex that is a group is invisible and only appears as a highlighted frame around its children when the group is selected.

What if a `DefaultEdge` is used as a group? The renderer will paint an edge between the specified source and target port, if available. A possible application is the visualization of long paths between two vertices, where the hops on the path are not important.

However, care has to be taken about the compatibility with the renderer in this case, since the renderer might rely on certain attributes to function properly. Therefore, if a cell other than a vertex is used to provide a group, it must also contain the attributes that are required by its respective renderer.

Ports may also be used as groups, which is even more “pathologic”. For an explanation of the synchronization between the value attribute and the user object, please read the chapter on *Attributes*.

3.3 Graph Model

3.3.1 GraphModel Interface

```
public interface GraphModel {
    // Attributes
    Map getAttributes(Object node);
    boolean isAttributeStore();
    // Roots
    int getRootCount();
    Object getRootAt(int index);
    int getIndexOfRoot(Object root);
    boolean contains(Object node);
    // Layering
    void toBack(Object[] cells);
    void toFront(Object[] cells);
    boolean isOrdered();
    // Graph structure
    Object getSource(Object edge);
    Object getTarget(Object edge);
    Iterator edges(Object port);
    boolean acceptsSource(Object edge, Object port);
    boolean acceptsTarget(Object edge, Object port);
    // Group structure
    Object getParent(Object child);
    int getIndexOfChild(Object parent, Object child);
    Object getChild(Object parent, int index);
    int getChildCount(Object parent);
    boolean isLeaf(Object node);
    // Change support
    void insert(Object[] roots, ConnectionSet cs, ParentMap pm, Map attributeMap);
    void remove(Object[] roots);
    void edit(ConnectionSet cs, Map nestedMap, ParentMap pm, UndoableEdit[] e);
    // Listeners
    void addGraphModelListener(GraphModelListener l);
    void removeGraphModelListener(GraphModelListener l);
    void addUndoableEditListener(UndoableEditListener listener);
    void removeUndoableEditListener(UndoableEditListener listener);
}
```

Listing 1: The GraphModel interface

The `GraphModel` interface defines the requirements for objects that may serve as a data source for the graph. The methods defined in this interface provide access to two independent structures:

- Graph structure
- Group structure

The graph structure follows the mathematical definition of a graph, where edges have a source and target, and vertices have a set of connected edges. The connection between edges and vertices is represented by a special entity, called *port*. In a sense, ports are children of cells and belong to the group structure. However, they are also used to describe the relations between edges and vertices, what makes them part of the graph structure.

Additionally, the `GraphModel` interface defines methods for the handling and registration of two different listener types: `UndoableEditListeners` and `GraphModelListeners`. The two are somewhat related in a way that

every notification of the undo listener is accompanied by a notification of the model listener.

The inverse is not the case: When a change is undone or redone, the model listeners are notified in order to update the view and repaint the display, but the command history must not be updated because it already contains this specific change.

Another important feature of the `GraphModel` interface is the ability to decide which connections are allowed. For this purpose, the `acceptsSource` and `acceptsTarget` methods are called from the graph's UI before an edge is connected to, or disconnected from a port, with the edge and port as arguments.

By overriding these methods, a custom model can return `true` or `false` for a specific edge, port pair to indicate whether the specified connection is valid, that is, if it may be established be the graph's UI. (These methods are also called when edges are disconnected, in which case the port is `null`.)

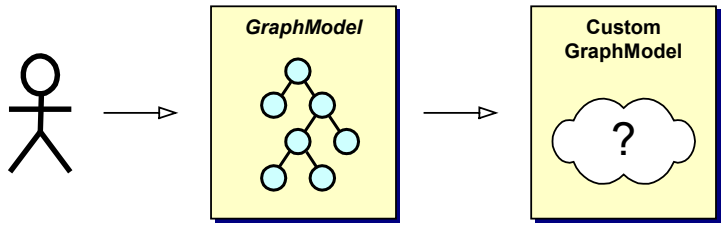

Fig. 17: Use the GraphModel interface to access the graph

It is the general practice to use the `GraphModel` interface to access the graph- and group structure, not the `GraphCell` or `TreeNode` interfaces that provide the same functionality. The reason for this is that the `DefaultGraphModel` already relies on these interfaces to implement its methods, whereas in a future implementation, a model could contain objects that do not implement the `GraphCell` interface, thus making it necessary for the model to store the data by other means. If the graph is accessed through the `GraphModel` interface, the code is independent of the model's internal structure, such that the model's implementation can be exchanged without having to change the client code.

3.3.2 GraphModel Default Implementation

JGraph provides a default implementation of the `GraphModel` interface in the form of the `DefaultGraphModel` class, which uses the `GraphCell` interface and its superclass, the `MutableTreeNode` interface to access the graph and group structure. This means, the model stores the data in the cells, that is, the cells make up the models group and graph data structure.

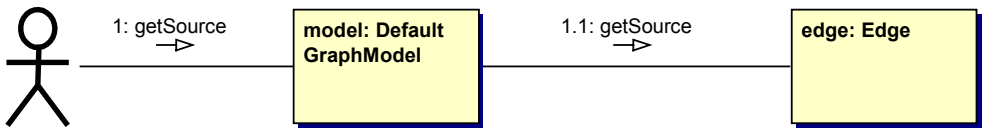

Fig. 18: How the source of an edge is accessed in the DefaultGraphModel

In analogy to Swing's `DefaultTreeModel`, which stores the data in the nodes, JGraph's `DefaultGraphModel` stores the data in the cells. This is efficient, but if cells are to be used in multiple models, with different relations in each model, the `DefaultGraphModel` is not suitable. This is because all relations are stored in the cells, making it impossible to determine the set of relations that belong to a specific model. (The same holds for the `DefaultTreeModel`.)

Consider for example a graph that contains two vertices A and B, and one edge. Each of the two vertices has one port as the only child, namely the port *a* for vertex A, and the port *b* for vertex B, as depicted in the figure below.

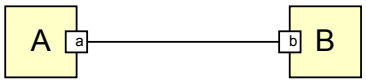

Fig. 19: A graph with two vertices and ports, and one edge in between

Fig. 20 shows the `DefaultGraphModel`'s representation of the graph. The root cells, that is, the cells with a `null`-parent, are stored in a list and may be retrieved using the `getRootCount` and `getRootAt` methods. The children are not contained in the list, they must be accessed through the `getChildCount` and `getChildAt` methods of the model, providing the parent and, if applicable, the index of the child as an argument.

The order imposed by these methods is used as the default layering in the graph view. The layering may be changed individually for each view, thus making it possible for each view to provide its own layering.

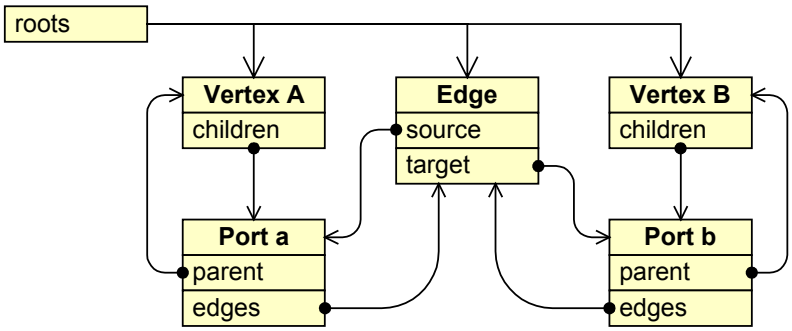

Fig. 20: Representation of the graph in the DefaultGraphModel

The above figure unveils that the `DefaultGraphModel` contains redundancy with respect to the storage of the graph and group structure. The set of edges connected to a port is deducible from the source and target field of the edge, but requires a traversal of the graph, examining the source and target of all contained edges.

Vice versa, computing the source and target for an edge out of the ports is even more expensive, because it requires a full graph traversal *and* a traversal of the edge sets for each port. Because such traversal is expensive, this redundancy is used as a cache in order to improve performance when accessing the graph model.

The `DefaultGraphModel` implements the `acceptsSource` and `acceptsTarget` methods to return `true` for any arguments, and the `isAttributeStore` method to return `false`, which means that the `DefaultGraphModel` is not an attribute store. By extending the `DefaultGraphModel`, these methods may be overridden to return something more meaningful with respect to the application.

3.4 Changing the Model

Changes to the graph model are atomic, and may be stored in a command history for later undo and redo. Possible manipulations of the graph model are:

- Insert
- Remove
- Change

Composite changes, that is, changes that combine two or all of the above are not provided by the `GraphModel` interface, however, such changes are sometimes used internally, by the `DefaultGraphModel`.

Changes to the layering of the cells could be implemented in the model, but in JGraph's default implementation, the `GraphView` class provides this functionality.

To insert and change cells, a number of classes are used as arguments to the respective methods:

- Nested maps, from cells to attribute maps allow defining new or changed attributes.
- The `ConnectionSet` class provides the graph structure by providing the connections to be established or removed.
- The `ParentMap` class describes the group structure, which consists of parent, child pairs.

3.4.1 Nested Maps

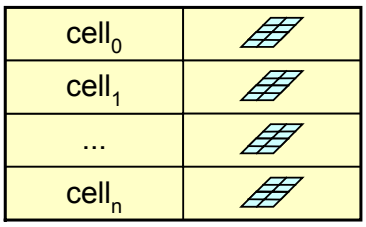

Fig. 21: Pictorial representation of a map from cells to attributes

The `insert` and `edit` methods allow to specify a mapping from cells to attributes. Thus, it is not only a map; it is a map of maps. This may cause confusion, especially when changing only one cell, in which case the outer map contains only one entry. In order to keep the `GraphModel` interface simple, however, this is the only way that individual cells may be changed, because it is the most general approach.

3.4.2 ConnectionSet

The `ConnectionSet` class is used to construct a change to the graph structure, that is, it changes the connectivity of the graph. By adding edge, port pairs, the to-be established or removed connections are defined. The name contains the term *set* because this object imposes a set semantic on the data that it contains.

Internally, the connections are described by instances of the `Connection` class, which is an inner class of the `ConnectionSet` class. Two connections are equal if they describe the source or target of the same edge, and the connection that is added later overrides the existing connection. (To be precise: The `Connection` class holds a reference to the edge, port and a `boolean` that indicates whether the port is a source or target. The equality is defined over the edge and the `boolean`.)

A connection set that describes the existing connections for an array of cells in a given model may be created using a factory method, which is a `static` class method that returns a `ConnectionSet` instance. The resulting connection set may either describe the connection, or the disconnection of the specified cells in a given model, based on a `boolean` argument.

3.4.3 ParentMap

The `ParentMap` class is used to change the group structure. Using the parent map, child, parent pairs are stored, which are then used to establish a parent-child relation between the specified cells in the model. The term *map* is used because this object is in fact a map, from children to parents. As with all

maps, the keys are stored in a set, thus, for each child there exists exactly one parent in a parent map. Like with `ConnectionSets`, old entries are overridden by new entries if the same child is used twice.

Internally, the parent map counts the number of children a parent will have after its execution. A method is provided to access this count, so that future empty parents may be marked to be removed during the construction of the transaction. In this sense, there *are* transactions that combine the change and removal of cells, but this feature is not available through the `GraphModel` interface. (It is used internally, to remove groups that have no children.)

The `ParentMap` class provides a constructor that may be used to create the group structure for a cell array in a given model. The object may either be constructed to *establish* these relations, or to *remove* them.

3.4.4 Inserting Cells

When inserting cells into the model, only the topmost cell must be inserted. Since the model relies on the cell to retrieve the children, these are inserted implicitly by inserting their parent. Thus, when inserting a cell hierarchy, only the topmost cell of the hierarchy must be inserted. (While it is possible to insert the children of a cell into the model's root list, it usually should be avoided.)

Upon insertion, the model checks if it is an attribute store, and if so, it absorbs the attributes by storing them locally in the cells. (If the model is not an attribute store, then the attributes are passed to the views, and not used in the model.) An object that describes the insertion is created, executed and sent to the undo and model listeners using the model's `postEdit` method.

3.4.5 Removing cells

When removing cells from the model, the children must be treated in a special way, too. In contrast to the above, if children are omitted on a remove, the resulting operation is an *ungroup*. That is, the specified cells are removed from the root list of the model, and the first generation of children is removed from the cell, and added to the parent's parent.

Another special case is the removal of all children from a group, without removing the group (cell) itself. In this case, the default model removes the group automatically. (This also occurs when all children of a group are moved to another parent.)

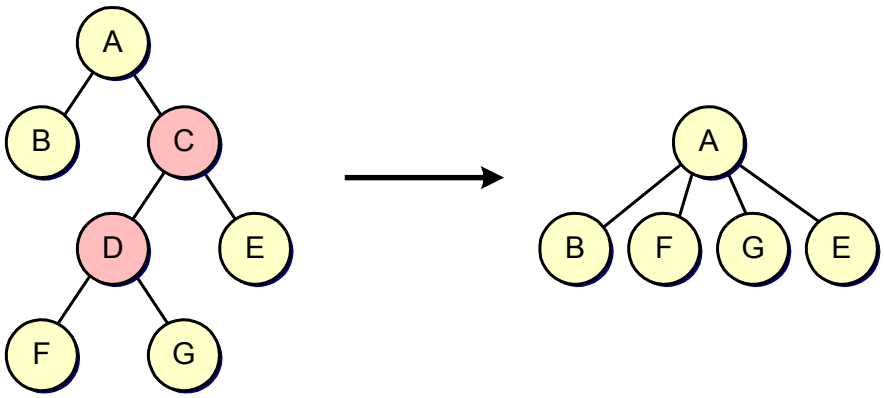

Fig. 22: Group structure before and after the removal of C and D without children

On the left, the group A contains a group C, which in turn contains a group D and a cell E. The group D contains the cells F and G. In the figure on the right, the resulting group structure consists of a group A, which contains the cells B, E, F and G. The removed cells' children have been added to their *parents* parent, hence, they only become roots of the model if the parent was itself a root cell, that is, if its respective parent points to `null`.

3.4.6 Changing Cells

When cells are modified, there are four arguments to the `edit` method: The `ConnectionSet` to change the graph structure, the `ParentMap` to change the group structure, a map of maps to change the attributes, and an array of undoable changes. The third argument is a map, from cells to attributes, which is in turn a map from keys to values. (The last argument is used internally for undo-support, by the graph view.)

If the last argument to the edit call is the only non-`null` argument, then the model relays the undoable changes to the undo listeners registered with it. In any other case, the model creates an object that describes the modification, executes it and notifies the undo- and model-listeners using the `postEdit` method.

3.5 Selection Model

Logically, JGraph's selection model is an extension of the JTree selection model, offering all its functionality, such as single and multiple cell selection. Additionally, JGraph's selection model allows stepping into groups, as outlined below. The selection model therefore is in charge of computing the currently selectable cells based on the current selection, and it also must ensure certain properties on the selection. For example, it must ensure that a cell and its descendants are never selected at the same time.

In order to retrieve the selection candidates, that is, the cells that are currently selectable, the interface provides the `getSelectables` method. The stepping into feature can also be switched off, in which case the selection model offers the normal single- and multiple cell selection modes, as in the case of JTree's selection model.

3.5.1 Stepping-Into Groups

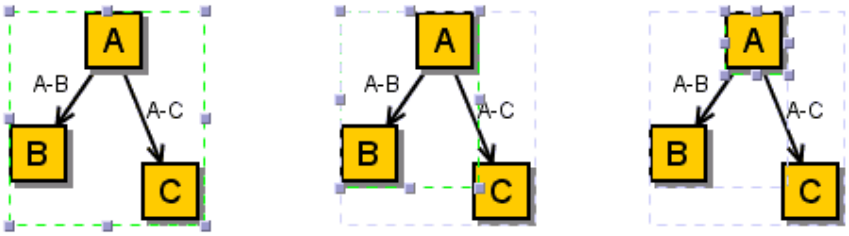

Fig. 23: Stepping-into groups in a UI, from the outermost group to the innermost cell

When explaining the stepping into feature, it is important to outline the order in which cells are selected. The basis of this order is the view-dependent layering of the cells, or the model's order if the model is an attribute store.

The order in which cells are selected is defined over the sequence of selectable cells, which is inductively defined by the following, and is initially equal to the model's list of root cells. (Clearly, a cell may only be selected if it is selectable, that is, if it is contained in the sequence of selectable cells.)

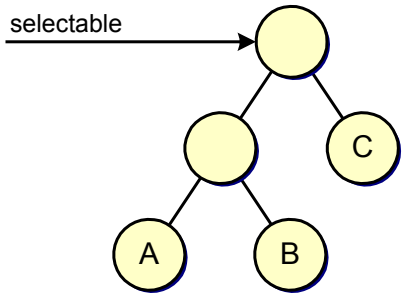

Fig. 24: Initially, the root cells are selectable

If the user clicks on the graph, the selectable sequence will be traversed, checking for each cell if it intersects the point of the mouse click. If an unselected intersecting cell is found, it will be selected, unless the `isToggleSelectionEvent` method returns true, in which case the topmost selected intersecting cell is removed from the selection (typically Shift-Click).

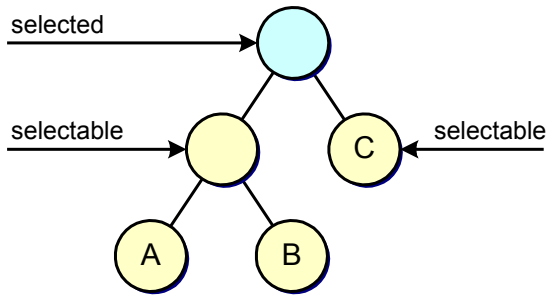

Fig. 25: Children of selected roots are selectable

When a cell is added to the selection, its children are added between the selected cell and the next cell in the sequence (the selected cell is not removed from the sequence). Thus, the system first selects the children of the topmost cell, and then continues selecting the underlying cells, and their children respectively, upon selection of their parent.

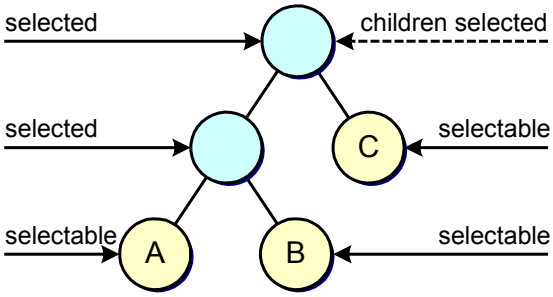

Fig. 26: Children of selected children are selectable

When a cell is removed from the selection, all its children are removed from the sequence. The cell itself is deselected, but still contained in the selectable list. Thus, a cell only ever leaves the selectable list if its parent is deselected, which is only the case for non-root cells, because root cells have no parents.

Since each view may provide a different layering for the cells, and the selection model has no notion of view-layering, the selectable list that the selection model returns is more precisely defined as a *set* of candidates to be selected. The set is then turned in to a sequence by ordering the cells based on the view-dependent layering according to the rules stated above.

4 View

The view in JGraph is somewhat different from other classes in Swing that carry the term *view* in their names. The difference is that JGraph's view is stateful, which means it contains information that is solely stored in the view. The `GraphView` object and the `CellView` instances make up the view of a graph, which has an internal representation of the graph's model.

4.1 Graph View

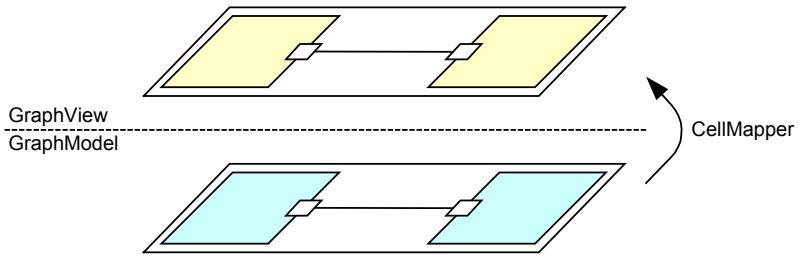

Fig. 27: GraphView and GraphModel

The `GraphView` object holds the cell views, namely one for each cell in the model. These views are kept in two structures, an array that holds the port views, and a list that holds the views for the root cells. Thus, ports and other cells are kept in different data structures in the graph view, whereas in the model, they are kept in the same structure. The graph view also has a reference to a hash table, which is used to provide a mapping from cells to cell views. The inverse mapping is not needed, because the cell views point to their corresponding cells.

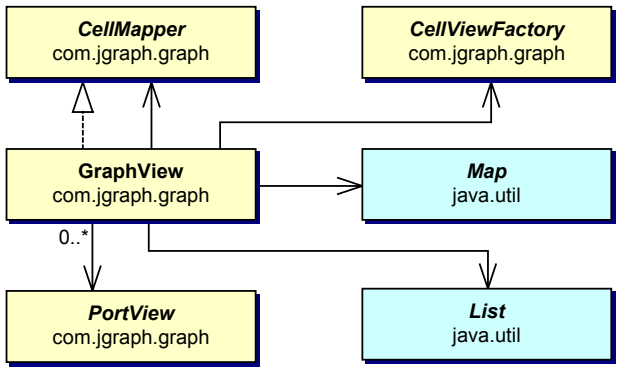

Fig. 28: GraphView class and static relations

`GraphView` extends `CellMapper`, has a reference to a `CellMapper` (usually `this`), and a `CellViewFactory`, and instantiates an array of

`PortViews`, a `List` that holds the root views, and a map that is used to implement the `CellMapper` interface.

The children may be accessed through the `CellView` interface, and are not accessible through the `GraphView` object. The creation of new views is deferred to the `CellViewFactory`. By default, JGraph is used to provide this factory, which means that the JGraph object is ultimately responsible to create the views.

The graph view's representation of the graph is different from the model's internal structure, in that it was designed with the view's operations in mind. For example, the view, in contrast to the model, makes a difference between children that implement the `Port` interface, and other children. The children that implement the `Port` interface are treated separately, both, as children, and with respect to storage, and are kept in a separate array.

This is because the set of available ports is often accessed independently of the rest, for example when the graph is drawn with port visibility set to `true`, or when the source or target of an edge is changed interactively, in which case the new ports must be found and highlighted. To find the port at a specific location, the graph view loops through the array of ports in the view, and checks if a port intersects the specified point, and if so, returns this port.

Even if the ports are always visible and painted on top of all cells, the order in which ports are stored reflects the layering of the corresponding cells in the view. This means, if two ports are on top of each other, the port that belongs to the topmost cell will be selected first.

4.2 Cell Mapper

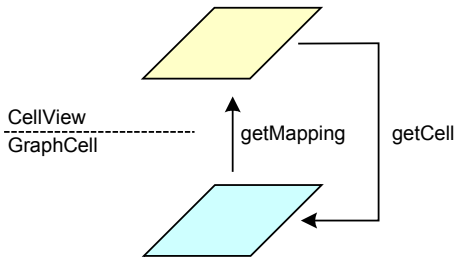

Fig. 29: The CellMapper maps from GraphCells to CellViews

The `GraphView` class implements the `CellMapper` interface, which defines the mapping from cells to views. The interface provides two methods, one to retrieve the cell view for a given cell – and optionally create the view if it does not exist, using the `CellViewFactory` – and one to associate a given cell with a newly created cell view.

The graph view provides additional useful methods that are not required by this interface, namely a method to map an array of cells to an array of views,

one to get all descendants for a specific view (*not* including ports), and one to return the visual bounds for an array of views.

The `CellMapper` interface is used in JGraph to encapsulate the most important part of a `GraphView` object, namely the mapping from `GraphCells` to `CellViews`. (It also allows the graph view's mapping to be exchanged at run-time, which is needed for *live-preview* that is explained later.) The reverse mapping is not needed, because each `CellView` instance has a reference to its corresponding `GraphCell` instance.

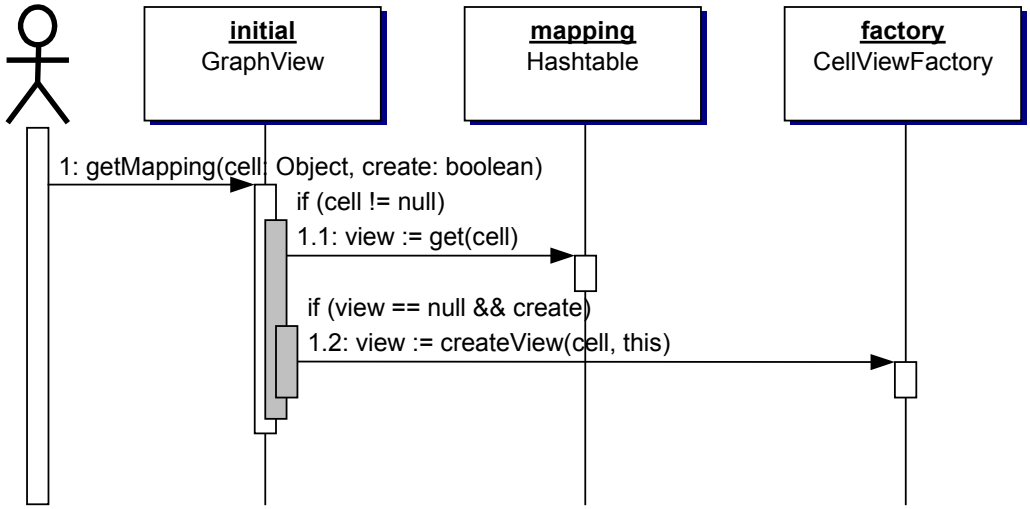

Fig. 30: CellViews may be created automatically if they do not exist

When a cell is looked-up in the graph view in order to find its corresponding cell view, a hash table access is performed using the cell's hash code, which is provided for every Java object by the `Object` class, as a key. If the cell is not associated, that is, if the hash table returns `null`, the view is created based on the `create` argument, using the `CellViewFactory` interface.

4.3 CellView Factory

The association between a cell and its view is established during the creation of the view, from within the `CellViewFactory`. This is because the creating of a cell view entails the creation of its dependent views, such as children, or the source and target of an edge. Therefore, the association between the initial cell and its view must be available by the time the views of the children are created (because child views have a reference to their parent views, and edge views have a reference to their source and target port views, which are looked-up using the `CellMapper` object that is passed to the `createView` method upon creation of the view.)

4.4 Cell Views

Cell views are used to store the geometric pattern of a graph, and also to associate a renderer, editor and cell handle with a certain cell view. The renderer is responsible to paint the cell, the editor is used for in-place editing, and the cell handle is used for more sophisticated editing. For efficiency, the *Flyweight* pattern is used to share the renderer among all instances of a given specific `CellView` class. This is achieved by making the reference to the renderer `static`. The editor and cell handle, instead, are created on the fly.

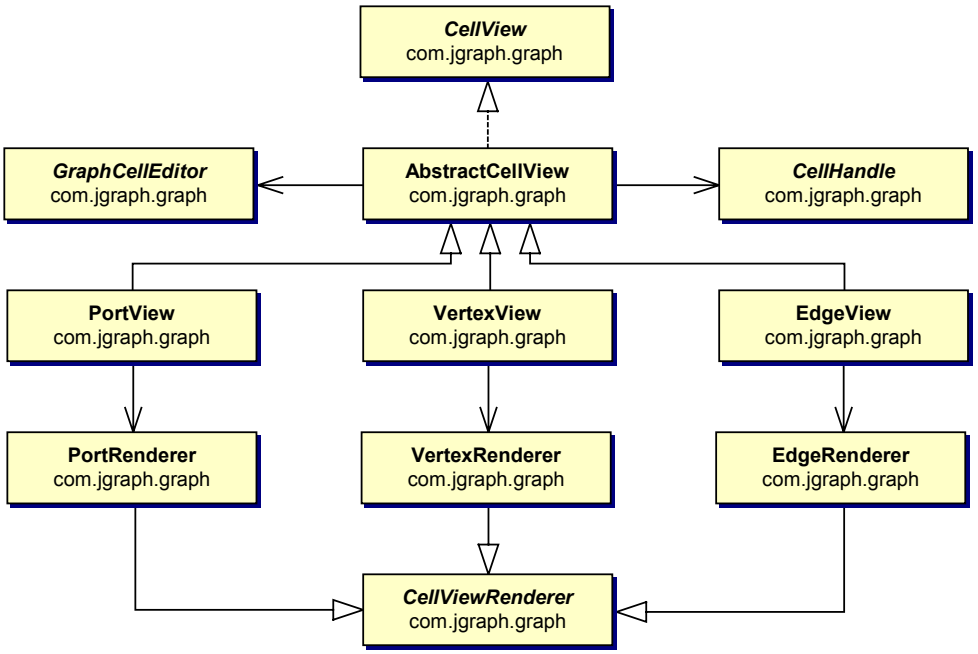

Fig. 31: CellView interface, default implementations and static relations

`VertexView`, `EdgeView` and `PortView` extend `AbstractCellView`, which in turn extends `CellView`. `AbstractCellView` has a static reference to the `GraphCellEditor` and to the `CellHandle`. Each concrete view has a `static` reference its corresponding subclass of `CellViewRenderer`, which is used to paint the cell.

4.4.1 CellView Interface

The `CellView` interface is used to access the view's graph and group structure, and to hold attributes. The `CellView` interface provides access to the parent and child views, and to a `CellViewRenderer`, a `GraphCellEditor` and a `CellHandle`. The `getBounds` method returns the bounds of the view, and the `intersects` method is used to perform hit-detection.

The `refresh` method is messaged when the corresponding cell has changed in the model, and so is the `update` method, but the latter is also invoked when a dependent cell has changed, or when the view needs to be updated (for example during a live-preview, explained in chapter 5.4.1).

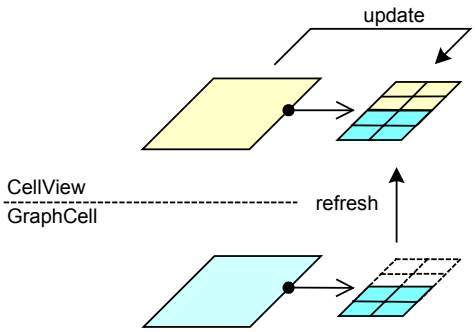

Fig. 32: CellView update and refresh

The `update` method is a good place to implement automatic attributes, such as edge routing, that is, attributes that are computed based on other attributes or the graph's geometry. The attributes of a cell view are accessed using the `getAttributes` method, with the exception of the bounds, which are cached, and accessed using the `getBounds` method.

4.4.2 CellView Default Implementations

By default, the JGraph object, which acts as a `CellViewFactory` is able to return cell views for each of the default implementations of the `GraphCell` interface. The cell view is created based on the type of the passed-in cell. For objects that implement the `Edge` interface, `EdgeViews` are created, and for objects that implement the `Port` interface, `PortViews` are created. For other objects, `VertexViews` are created.

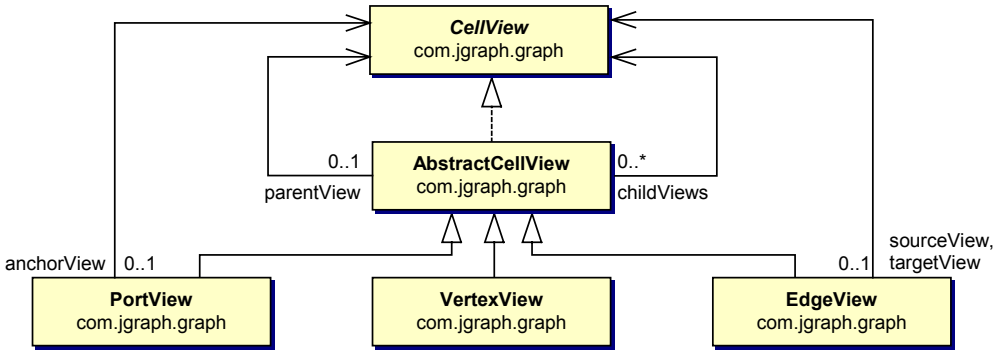

Fig. 33: CellView default implementations and static relations

`AbstractCellView` has a reference to the parent `CellView`, and an array of `CellViews` that represent the views of the children. `EdgeView` has a reference to the `CellView` of the source and target port, and `PortView` has a reference to the `PortView` of the anchor. In contrast to the `DefaultPort`'s edge set, the `PortView` does not provide access to the set of attached edges.

4.4.2.1 AbstractCellView

The `AbstractCellView` class provides the basic functionality for its subclasses. For example, it implements the `getRendererComponent` method to create a renderer, configure and return it. The concrete renderer is retrieved using an inner call to the `getRenderer` abstract method, which must be implemented by subclassers to return a specific renderer.

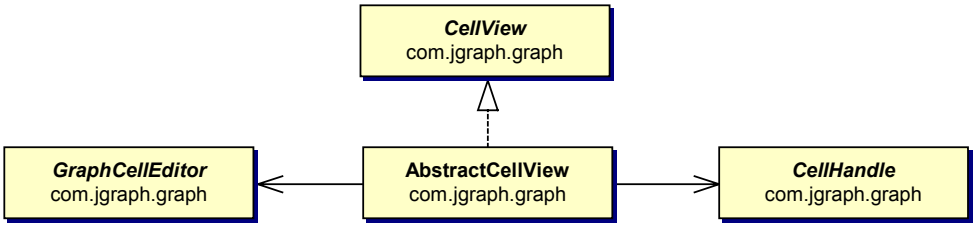

Fig. 34: AbstractCellView

The `intersects` method, which is used to implement *hit-detection*, uses the `getBounds` method to check if a given point intersects a cell view. The `EdgeView` class overrides this default implementation, because in the context of an edge, a hit should only be detected if the mouse is over the shape of the edge (or its label), but not if it is inside the bounds and outside the shape.

4.5 Changing the View

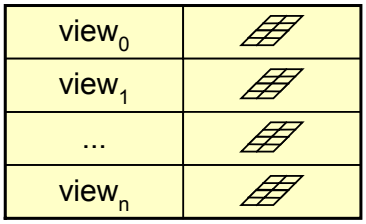

Fig. 35: Nested map for the graph view's edit method

Using the view's `edit` method, the attributes of the cell views may be changed. The argument is a map, from views to attributes. The `toBack` and `toFront` methods are used to change the layering of the cell views.

4.6 Graph Context

The `GraphContext` class is named after the fact that each selection in a graph may possibly have a number of dependent cells, which are not selected. These cells must be treated in a special way in order to display a live-preview, that is, an exact view of how the graph will look after the current transaction. This is because for example an edge may change position when its source and target port are moved, even if the edge itself is not selected.

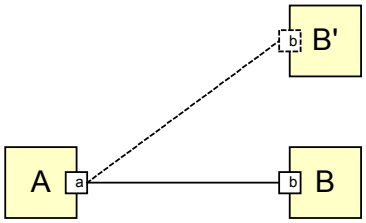

Fig. 36: GraphContext for the moving of vertex B

For example, when moving the vertex B in the above graph, its port, and the attached edge must be duplicated and repainted to reflect the new position of the vertex, even if the edge and port themselves are not selected. Thus, the selection consists of the vertex B only, and the port and edge are duplicated because they are part of the context of this selection. (In the case where the user holds down the control-key while moving cells, which is interpreted as cloning, the unselected cells are not visible, because they will not be cloned when the mouse button is released.)

The term *context* is used to describe the set of cells that are *indirectly* modified by a change, but are not themselves part of the selection. The `GraphContext` object is used to compute and manage this set. Additionally, the object may be used to disconnect a set of edges from the original graph, that is, from all the ports that are not part of the `GraphContext`. (Being part of the graph context either means that a cell or one of its ancestors is selected, or the cell is part of the context of this selection.)

4.6.1 Construction

To create a `GraphContext` object, the current selection is passed to the constructor. In the constructor, the set of descendants of the selected cells is computed using the model's `getDescendants` method, and stored within the `GraphContext` object. Then, the set of edges that is attached to these descendants is computed using the models `getEdges` method, and the set of descendants is subtracted from it, resulting in the context, that is, the set of cells that are attached, but neither being selected, nor having selected ancestors.

Having these two sets at hand, the graph context may create the temporary views for the selected cells and their context. The temporary views are typically used to provide a live-preview for the change that is currently in progress, such as a move or a resize.

For such a change, the temporary views are created using the above mechanism, and then changed in-place, until the user releases the mouse, in which case the changes are applied to the original attributes using the `applyMap` method. If the user presses *escape* during the change, then the change is aborted, and therefore the initial attributes are not changed in this case.

4.6.2 Temporary Views

The temporary views are created in two steps: First, the temporary views of the selected cells are created, and associated with their corresponding cells using a local hash table. Then, the temporary views for the *context* are created, using the `GraphContext` instance as the `CellMapper`, and also stored in this local hash table.

The mapping in the graph context uses the local hash table to look-up the references for the temporary views. That is, the views created by the graph context reference the temporary views as their parents, children or source and target ports, if such temporary views exist. (The temporary views are created lazily, when needed, based on the cell being contained in either the set of descendants, or the context, and stored in the local hash table upon creation.)

Thus, in contrast to the graph view, this cell mapper does only create a new (temporary) cell view if the cell is part of the context, and therefore, the resulting views reference the original graph's cell views if the corresponding cells are not part of the context.

These temporary views are used for live-preview, which is explained in the next chapter, because it is related with the `CellHandles`, which are used in the UI-delegate (the "control").

5 Control

The *control* defines the mapping from user events to methods of the graph- and selection model and the view. It is implemented in a platform-dependent way in the *UI-delegate*, and basically deals with in-place editing, cell handling, and updating the display. As the only object that is exchanged when the look and feel changes, the UI-delegate also specifies the look and feel specific part of the rendering.

5.1 UI-Delegate

The `GraphUI` abstract class and its default implementation, the `BasicGraphUI` constitute JGraph's UI-delegate. Aside from the event listeners to track user input, the UI-delegate is also used to paint the graph, and update the display when the graph or the selection changes. When a graph is painted, the UI-delegate loops through the views and paints them by painting their respective renderers.

5.1.1 GraphUI Interface

The `GraphUI` abstract class defines the methods for an object that may be used as a UI-delegate for JGraph, whereas the `BasicGraphUI` provides default implementations for these methods. The `BasicGraphUI` in turn may be subclassed to implement the UI for a specific look and feel, but the default implementation already contains some look and feel specific coloring.

5.1.2 GraphUI Default Implementation

The `BasicGraphUI` class maintains a reference to a `RootHandle`, which is used to react to mouse events. The mouse listener, which calls this handler, is an inner-class of the UI-delegate. It defines the basic behavior of JGraph with regard to mouse events, and implements the two main functionalities: selection and editing. Selection includes single-cell and marquee selection. Editing can either be based on the `CellEditor` interface, for in-place editing, or on the `CellHandle` interface, for cell handling.

The `BasicGraphUI` creates the root handle, whereas the cell handles are created by the cell views. The root handle is responsible to move the selection, whereas its children are used to change the shape of the cells. A set of default keyboard bindings is supported by the UI-delegate, and listed in Table 1. (The number of clicks that trigger in-place editing can be changed using the `setEditClickCount` method of JGraph.)

Double-click / F2	Starts editing current cell
Shift-Click	Forces marquee selection
Shift-Select	Extends selection
Control-Select	Toggles selection
Control-Drag	Clones selection
Shift-Drag	Constrained drag

Table 1: JGraph's default keyboard bindings

5.2 Renderers

Renderers do not paint cells; they paint objects that implement the `CellView` interface. For each cell in the model, there exists exactly one cell view in each graph view, which has its own internal representation of the graph model. The renderers are instantiated and referenced by the cell views.

Renderers are based on the idea of the `TreeCellRenderer` class [16] from Swing, and on the *Flyweight* design pattern (see [17], page 195 ff). The basic idea behind this pattern is to "use sharing to support large numbers of fine-grained objects efficiently." [17]

Because having a separate component for each `CellView`-instance would be prohibitively expensive, the component is shared among all cell views of a certain class. A cell view therefore only stores the *state* of the component (such as the color, size etc.), whereas the renderer holds the component's overhead, and the painting code (for example a `JLabel` instance).

The CellViews are painted by *configuring* the renderer, and painting the latter to a `CellRendererPane` [19], which may be used to paint components without the need to add them, as in the case of a container. *Configuring a renderer* means to fetch the state from the cell view, and apply it to the renderer. For example in the case of a vertex, which uses a `JLabel` instance as its renderer, the cell's label is retrieved using `graph.convertValueToString`, and set using `setText` on the renderer, together with other supported attributes that make up the state of a vertex.

The renderers in JGraph are used in analogy to the renderer in JTree, just that JGraph provides more than one renderer, namely one for each type of cell. Thus, JGraph provides a renderer for vertices, one for edges, and one for ports. For each subtype of the `CellView` interface, overriding the `getRenderer` method may associate a new renderer. The renderer should be `static` to allow it to be shared among multiple instances of a class.

The renderer itself is typically an instance of the `JComponent` class, with an overridden `paint` method that paints the cell, based on the attributes of the passed-in cell view. The renderer also implements the `CellViewRenderer`

interface, which provides the `getRendererComponent` method to configure the renderer.

5.2.1 CellViewRenderer Interface

This interface provides two methods: one to configure and return the renderer, the other to check if an attribute is supported. The first method's arguments are used to specify the cell view and its state, that is, if it is selected, if it has focus, and if it is to be painted as a preview or in highlighted state.

The preview argument is `true` if the renderer is used in the context of live-preview, which requires faster painting. In this case, the renderer may decide not to paint the cell in full quality, based on the cost of this painting. The arguments passed to this method can *not* be deduced from the cell view; they are determined by the UI-delegate, the selection-model or from the context in which the renderer is used.

5.2.2 CellViewRenderer Default Implementations

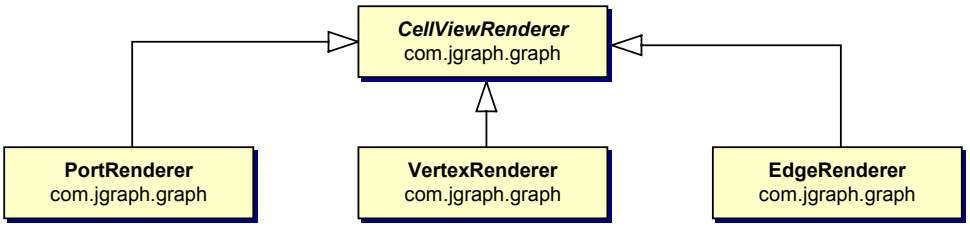

Fig. 37: CellViewRenderer default implementations

The three default implementations of the `CellViewRenderer` interface are the `VertexRenderer`, `EdgeRenderer` and `PortRenderer` classes. The `VertexRenderer` is an extension of `JLabel`, which offers the code to paint the label, the icon, the border and background of a vertex. The `paint` method is only overridden to paint the selection border.

The other renderers are subclasses of the `JComponent` class, because they perform special rendering, which is not offered by any of the `JComponent` extensions. The `EdgeRenderer` uses the `Graphics2D` class and the `Shape` interface to paint the label, edge, and decorations. The `PortRenderer` provides the code to paint a small cross, or a highlighted port in the case where the highlight argument is `true`.

The `preview` flag is ignored in the context of `PortRenderers` because in live-previews, ports are never painted, and therefore, the flag is never `true` when used in a `PortRenderer`.

5.3 Editors

In-place editing is accomplished through a set of methods that the UI-delegate provides. Internally, the object that is used as an editor is retrieved from the cell view, just like the renderer is. In contrast to the renderer, however, there is only one `GraphCellEditor` for all cells, even if the design allows different editors to be specified for each view type, even for each view *instance*.

As in the case of the `CellViewRenderer` interface, the `GraphCellEditor` interface provides a method that is used to return a configured editor for the specified cell. The argument to this method is not a `CellView`, as in the case of `CellViewRenderer`; it is a cell instead, which is of type `Object`. This underlines the fact that the label is typically stored in the model, not in the view.

The editor is placed inside the graph by the UI-delegate, which is also responsible to remove it from there, and dispatch the change to the model. Thus, for customized editing, it might be necessary to provide a custom UI-delegate, overriding the protected `startEditing` and `completeEditing` methods of the `BasicGraphUI` class.

5.4 Cell Handles

Cell handles are a new concept that is not used elsewhere in Swing. This is because in Swing, the only method to edit cells is by means of in-place editing. It is not possible to change the bounds of a cell, or to move the cell to an arbitrary location. Rather, the UI-delegate and the state of the component determine the location of a cell. In JGraph instead, the position and size are stored in the attributes, and do not depend on the state of the graph. Therefore, a way must be provided to change the attributes in a transaction-oriented way.

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

While the root handle is active, the UI-delegate's internal mouse handler messages its methods, and the root handle in turn delegates control to the correct subhandle, or absorbs the events in order to move the selection.

5.4.1 Live-Preview

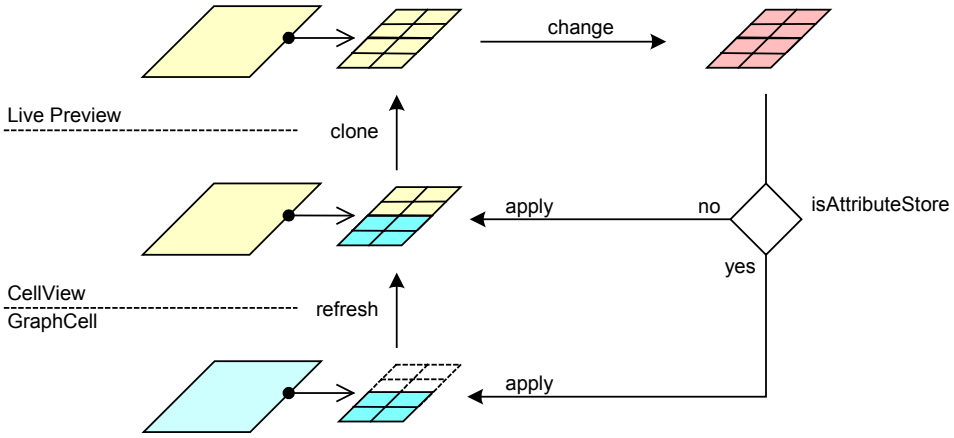

Fig. 38: Live-preview

During a change to the graph, the new attributes are displayed by use of temporary cell views, which have previously been created using the `GraphContext` object. This is referred to as the *live-preview*, meaning that the graph is overlaid with the change, so that the user can see what the graph will look like when the mouse is released. (The change may be cancelled by pressing the Escape key.)

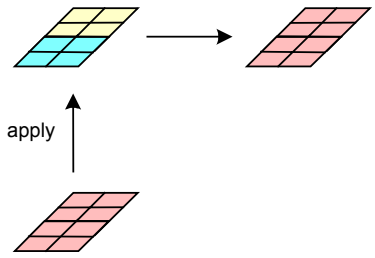

Fig. 39: Live-preview uses the applyMap method

If the model is an attribute store, then the change is executed on the model. Otherwise, the change is executed on the view. In both cases, the `applyMap` method is used, which is provided by the `GraphConstants` class.

5.4.2 CellHandle Interface

The `CellHandle` interface is very similar to a `MouseAdapter`, providing all the methods that are used to handle mouse events. In addition, the interface defines two methods to paint the handle, namely the `paint` method,

which is used to draw the static part (the actual handles), and the `overlay` method, which is used to draw the dynamic part (the live-preview), using fast XOR'ed-painting.

5.4.3 CellHandle Default Implementations

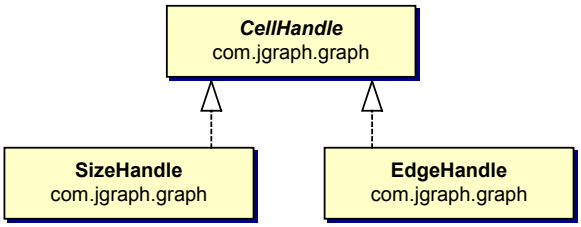

Fig. 40: CellHandle default implementations

The default implementations of the `CellHandle` interface are the `SizeHandle`, `EdgeHandle`, and `RootHandle` classes. The root handle is responsible to move cells, the size handle is used to resize cells, and the edge handle allows to connect and disconnect edges, and to add, modify or remove individual points.

5.5 GraphTransferable

The UI-delegate provides an implementation of the `TransferHandler` class, which in turn is responsible to create the `Transferable` based on the current selection.

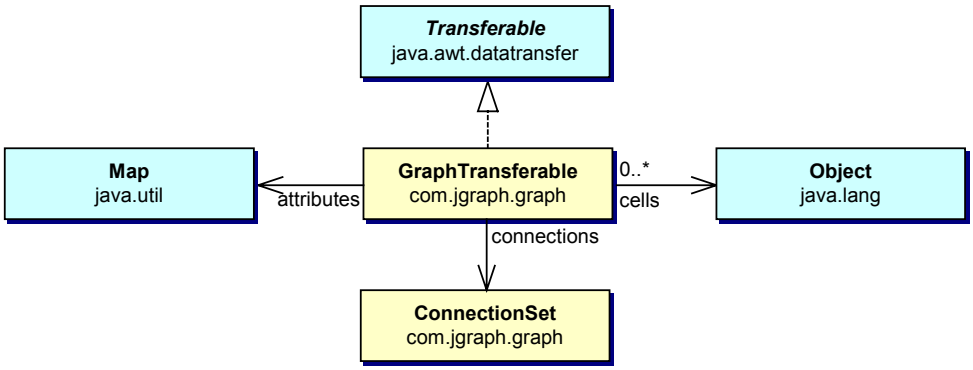

Fig. 41: The GraphTransferable class

The transferable is represented by the `GraphTransferable` class, which in turn has a reference to the `ConnectionSet`, the cells and their corresponding view's attributes. The `ConnectionSet` is created using the

`static` factory method that was outlined before. The data may be returned as serialized objects, or as plain text or HTML. In the latter two cases the label of the selected cell is returned, or an empty string, if the transferable contains more than one cell.

5.6 Marquee Selection

Marquee selection is the ability to select a rectangular region by use of the mouse, which is not provided by Swing. The `BasicMarqueeHandler` class is used to implement this type of selection. From an architectural point of view, the marquee handler is analogous to the transfer handler, because it is a "high-level" listener that is called by low-level listeners, such as the mouse listener, which is installed in the UI-delegate.

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

5.7 Event Model

In JGraph, the graph model, selection model and graph view may dispatch events. The events dispatched by the model may be categorized into

- Change notification
- History support

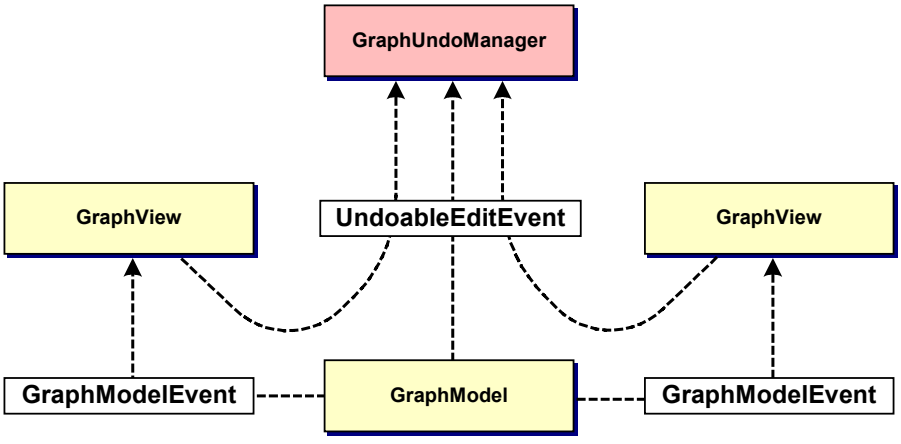

Fig. 42: JGraph's event model

The `GraphModelListener` interface is used to update the view and repaint the graph, and the `UndoableEditListener` interface to update the history. These listeners may be registered or removed using the respective methods on the model.

The selection model dispatches `GraphSelectionEvents` to the `GraphSelectionListeners` that have been registered with it, for example to update the display with the current selection. The view's event model is based on the `Observer` and `Observable` class to repaint the graph, and also provides undo-support, which uses the graph model's implementation.

An additional notification mechanism between the model and the view enables a single model to have multiple views. Note that the view does not itself implement the `GraphModelListener` interface, it is messaged by the model listener that is part of the UI-delegate, passing along the change information as an argument.

Each change is either executed on the model, or on a specific view, or it is a combination of both. No changes exist that affect multiple views at a time, except implicitly, through a change of the model. This leads to a separate notification between the local view and its UI to be repainted. The `GraphView` class implements the `Observable` interface, and the `BasicGraphUI` provides an `Observer` instance, which may be registered with the graph view, and is in charge of repainting the graph upon a view-only change.

5.7.1 Change Notification

Upon insertion, removal or modification of cells in the model, the model constructs an object that executes and describes the change. Because the *execution* and the *description* of such a change are closely related, these are stored in the same object. (Upon a change, the factory methods of the `ParentMap` and `ConnectionSet` classes are used to construct the respective objects that describe the change.)

The object is then sent to the model listeners such that they may repaint the affected region of the graph, and remove, add or update the cell views for the changed cells. The listeners might also want to ensure other properties, like for auto sizing, in which case the preferred size is recomputed when a cell has changed.

In contrast to the `TreeModelListener` interface, which provides three methods to be messaged, namely one for insertion, one for removal, and one if cells are changed, the `GraphModelListener` interface only provides one method to be informed on any change. (The latest release of the `TreeModelListener` even has an additional method, called `treeStructureChanged`.)

This has a subtle consequence when it comes to *composite changes*, which can insert, remove and change cells at the same time. In JTree listeners, these changes are handled by different parts of the handler code, resulting in up to four separate updates for one composite change, whereas in JGraph, the whole change is handled in one method, resulting in one single update, which is more efficient, even if such changes are rare.

Model listeners are notified using objects that implement the `GraphModelEvent` interface, which in turn returns an object that implements the `GraphChange` interface. The `GraphChange` interface is used to describe the change, that is, it returns the cells that were added, removed, or changed. An inner class of the model, which also contains the code to execute and undo the change, implements the `GraphChange` interface.

5.7.2 Undo-support

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

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

5.7.3 Undo-support Relay

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

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

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

5.7.4 GraphUndoManager

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

An extension of Swing's `UndoManager` in the form of `GraphUndoManager` is available to undo or redo such changes in the context of multiple views. `GraphUndoManager` overrides methods of Swing's `UndoManager` with an additional argument, which allows specifying the calling view as a *context* for the undo/redo operation.

This *context* is used to determine the last or next relevant transaction with respect to the calling view. *Relevant* in this context means *visible*, that is, all transactions that are not visible in the calling view are undone or redone implicitly, until the next or last visible transaction is found for the context.

As an example consider the following situation: Two views share the same model, which is not an attribute store. This means, each view can change independently, and if the model changes, both views are updated accordingly. The model notifies its views if cells are added or removed, or if the connectivity of the graph is modified, meaning that either the source or target port of one or more edges have changed. All other cases are view-only transactions, as for example if cells are moved, resized, or if points are added, modified or removed for an edge. All views but the source view are unaware of such view-only transactions, because such transactions are only visible in the source view.

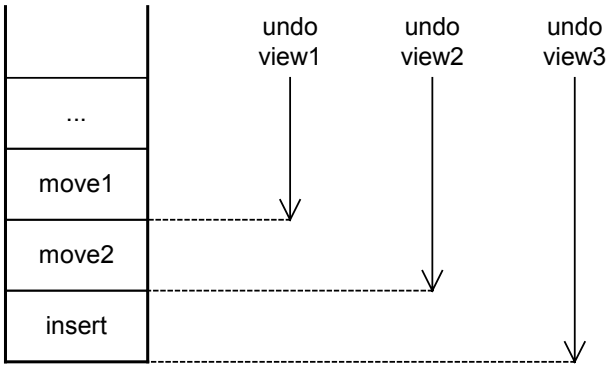

Fig. 43: Command history and multiple views

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

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

6 Conclusions

The JGraph component supports extended display and editing options, but the semantic of the graph, that is, the meaning of the vertices and edges, is defined by the application. Therefore, JGraph is highly customizable, and the classes provide “hooks” for subclassers. In cases where these customizations are not sufficient, the component’s source must be available to change the default implementation.

To summarize, the components for trees and lists are mostly used to *display* data structures, whereas this graph component is typically also used to *modify* a graph, and handle these modifications in an application-dependent way.

Because of JGraph’s attributes the following should be kept in mind: Register a view with the model before using the `insert` method, because otherwise the attributes that are passed to the method are ignored, resulting in cells that have the same size and position.

6.1 Cells and Components

Due to the special setup inherited from JTree, JGraph’s cells and cell views may not be used as components, and vice versa, components may not be added to JGraph. To use a cell view as a component, an instance of its renderer must be used instead, providing a dummy cell view that provides the data to be displayed. Vice versa, to place a component in a graph, the component should be used as a renderer for a specific cell view.

6.2 Composite Changes

Using the `GraphModel` interface, it is only possible to insert, remove or change the model, but it is not possible to create and execute composite changes, which do more than one of the above. To implement composite changes, a custom model must provide additional methods which either allow to create transactions out of other transactions, or access the model in a transaction-oriented way using `begin`, `commit` and `rollback` methods.

6.3 Scalability

In contrast to JTree’s default model, which offers a flag that affects the way data is stored and cached, in JGraph, data is always cached by the cell views. Thus, the default graph model scales well, and must only be extended if the algorithm for spatial search or the storage data structure needs to be replaced with a more efficient algorithm.

7 Appendix

7.1 Model-View-Control (MVC)

The term MVC is often used in conjunction with the architecture of user interface components, and stands for Model-View-Control, a design pattern that is explained in much more detail in [17]. (A design pattern is simply "the core of the solution to a problem which occurs over and over again".)

In this document, the terms *model* and *view* are used in the sense of the MVC pattern. The term *model* is used when speaking of the graph's model, that is, when speaking of objects that implement the `GraphModel` interface. The term *view* is used in the context of the `GraphView` object, and for objects that implement the `CellView` interface. In a sense, a cell may be seen as a model for the CellView, whereas the `GraphModel` is the model for the graph, and the `GraphView` is the view for this model.

In general, a view never contains the data, and a model never contains painting code. This is also true in JGraph, with the exception that aside from the data that the model provides, the view may contain additional data, which is independent from the model.

The MVC pattern is a collaboration of design patterns, most importantly the *Publish-Subscribe*, or *Observer* pattern, which defines the relationship between the model and the view. In general, the Observer pattern is used to "define a one-to-many dependency between objects so that when one object changes state, all its dependent objects are notified and updated automatically" [17].

One of the parameters of the Observer pattern is how to implement the notification scheme, that is, how much information should travel between the model and the view in order to describe a change. Basically, one solution computes the change information only *once* in the model, and dispatches it to all listeners, whereas the other solution simply notifies the listeners of a change, and the listeners in turn use the model to compute the change information if they are interested in handling the event.

Obviously, the first solution has the advantage that the change information is only computed once, whereas in the second solution, it is computed for every attached view. The disadvantage of the first approach is that a lot of information travels from the model to the view (possibly over a network), even if the view is not interested in handling the event. Of course, these solutions can be mixed as to compute part of the change information in the model, and the other part in the listeners. (JGraph uses the first approach, such that each event carries the full change information, which is also used to implement the command history.)

7.2 Swing

7.2.1 Swing MVC

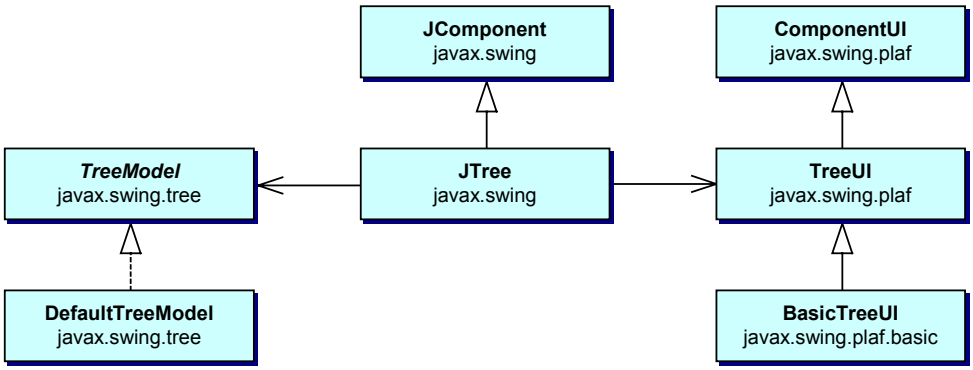

Fig. 44: Swing MVC in JTree

JTree extends `JComponent`, and has a reference to its `TreeUI`. JTree has a reference to a `TreeModel` and instantiates `BasicTreeUI`, which extends `TreeUI`, which in turn extends `ComponentUI`. The `ComponentUI`, sometimes called UI-delegate, is in a separate class hierarchy, so that it can be exchanged independently when the look and feel changes.

This basic structure of non-trivial Swing components is called the *Swing MVC pattern*. The control in Swing MVC is encapsulated in the *UI-delegate*, which is in charge of rendering the component in platform-specific manner, and mapping the events from the user interface to transactions that are executed on the model.

Since Swing offers the possibility to change the look and feel at run time, the component and its UI-delegate are implemented within different class hierarchies, such that the implementation of the UI-delegate can be exchanged at run-time, independently of the component. The JGraph component inherits this basic setup from the `JComponent` class and its UI-delegate, which implements the `ComponentUI` abstract class. (JGraph is in charge of instantiating the correct UI-delegate when the look and feel changes, and providing a reference to the current instance.)

7.2.2 MVC and Text Components

In analogy to characters in Swing's text components, which do not change when the look and feel changes, JGraph's cells do not depend on the look and feel. However, the color scheme, keyboard bindings and other aspects of the system *do* depend on the look and feel and must reflect it. As in the case of text components, the split between platform-dependent and non-platform-

dependent attributes is implemented using the concept of *views*, which are independent from the elements that appear in the model.

In Swing's text components, the elements of the model implement the `Element` interface, and for each of these elements, there exists exactly one view. These views, which implement the `View` interface, are either accessed through a mapping between the elements and the views, or through an entry point called *root view*, which is referenced from the text component's UI-delegate.

7.2.3 JGraph MVC

JGraph has an analogous setup, with the only difference that a graph view is referenced by the JGraph instance. The cells of the graph model implement the `GraphCell` interface, which is JGraph's analogy to the `Element` interface, and the cell views implement the `CellView` interface, in analogy to Swing's `View` interface. The cell views are accessed through the `CellMapper` mechanism, or through the graph view, which is an instance of the `GraphView` class.

The `GraphView` class can also be seen as a variation of JTree's default layout cache, because it separates the model from the view and provides view-dependent state-information for each cell, like the expanded property of a tree node, or one of its ancestors in JTree. However, since the `GraphView` class works together with other classes, the analogy with Swing's text components is more helpful to understand the separation between the cell and the view; even if JGraph was devised from JTree's source code.

In contrast to text components, where the geometric attributes are stored in the model only, JGraph allows to store such attributes separately in each view, thus allowing a graph model to have multiple geometric configurations, namely one for each attached view. The `GraphView` class can therefore be seen as a view-dependent geometric configuration of the model, which preserves state when the look and feel changes.

7.2.4 Serialization

Serialization is the ability of an object to travel across a network, or to be written to persistent storage. Two different kinds of serialization are available: short-term serialization and long-term serialization. Short-term serialization is based on a binary format, whereas long-term serialization is based on XML, making the result human-readable.

The JGraph component supports short-term serialization, but long-term serialization, which is offered as of the 1.4 release of Java, is not yet implemented. Therefore, serialization should not be used to implement a storage format for an application. This is because the format may not be compatible with future releases of JGraph.

It is in general not sufficient to store the model alone, because part of the information, namely the geometric pattern of the graph, is stored in the view (unless the model is an attribute store). Thus, the views should be stored along with the model in order to store the graph with its geometry.

7.2.5 Datatransfer

The ability to transfer data to other applications or the operating system by means of Drag-and-Drop or the clipboard is summarized under the term *datatransfer*. Drag-and-Drop allows transferring data using the mouse, whereas the clipboard provides an intermediate buffer to store and retrieve data using cut, copy and paste.

Between Java 1.3 and 1.4, the datatransfer functionality was extended with a high-level event listener, called `TransferHandler`. The `TransferHandler` is called from low-level listeners that are already installed in the `JComponent` class, and provides a set of methods that unify Drag-and-Drop and the clipboard functionality.

By inheriting from this class, the programmer can specify what data is transferred to the clipboard, or through Drag-and-Drop, and what happens after a successful drop on another target, or on the JGraph component, and how to handle non-standard contents of the clipboard.

Having said that, the concept of *data flavors* and *transferables* must briefly be outlined. Each object that is transferred to the clipboard implements the `Transferable` interface, which provides methods to retrieve the different forms in which the transferable is available to an application (for example as HTML or plain text). The `Transferable` itself can be seen as a wrapper that contains the different representations (MIME-types) of the transferable data, together with the methods to access these representations in a type-safe way.

JGraph provides an implementation of the `Transferable` interface in the form of the `GraphTransferable` class, which allows to transfer the current selection either as cells (serialized objects), as plain text, or as HTML. (Swing's default `TransferHandler` can't be used, because it transfers bean properties, and the graph's selection can't be implemented as a bean property.)

The `GraphTransferable` transfers the serialized cells along with their attributes, and a connection set and parent map that define the relations between the transferred cells in the originating model. These objects may be used as arguments in a target model's `insert` method, or individually, to provide a custom data flavor.

7.3 Packages

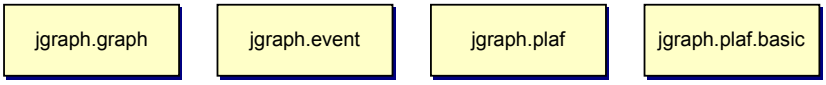

Fig. 45: JGraph's packages

Graphs are made up of a number of classes and interfaces defined in their own package - the `jgraph.graph` package. The `jgraph.graph` package provides support classes that include the graph model, graph cells, graph cell editors, and renderers. The `JGraph` class represents the graph component, which is the only class residing in the topmost `jgraph` package.

The `jgraph.event` package contains event classes and listener interfaces that are used to react to events fired by `JGraph`. The `GraphUI` class in the `jgraph.plaf` package extends Swing's `ComponentUI` class and defines abstract methods specific to graphs. The `BasicGraphUI` class in the `jgraph.plaf.basic` package defines default implementations for these methods.

In general, the graphical representation is look and feel dependent, which means that the graph reloads its user interface (UI) when the look and feel changes. The UI allows single-cell and marquee selection, as well as in-place editing, modification and moving of cells. Zoom, command history, Drag-and-Drop, and clipboard-support are also provided.

Cells have a user object. User objects are of type `Object` and therefore provide a way to associate any object with a cell. Graphs have a fairly simple model, and each `JGraph` instance maintains a reference to a graph model. The key classes from the `jgraph.graph` package are listed in Table 2.

`DefaultGraphModel`	A graph model that defines methods for adding, removing and changing cells.
`GraphView`	An object that constitutes the geometric pattern of the graph.
`DefaultGraphCell`	The base class for all elements in the model, also represents a vertex.
`DefaultEdge`	An edge that may connect two vertices via a source and target port.
`DefaultPort`	Acts as a child of a vertex, and the connection point for edges.

Table 2: Key classes from the jgraph.graph package

7.4 Class Index

GraphUndoManager (see 5.7.4)
GraphView (see 4.1)
ParentMap (see 3.4.3)
Port (see 3.2.1)
PortRenderer (see 5.2.2)
PortView (see 4.4.2)
VertexRenderer (see 5.2.2)
VertexView (see 4.4.2)

Plaf

GraphUI (see 5.1.1)

Basic

BasicGraphDropTargetListener (see [19])
BasicGraphUI (see 5.1.2)
BasicTransferable (see [19])

7.5 UML Reference

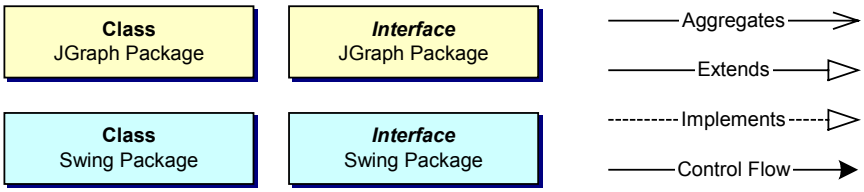

Fig. 46: UML for static structure diagrams

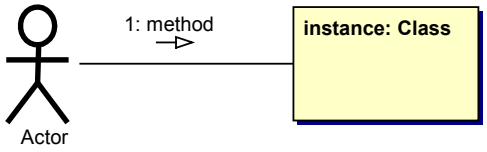

Fig. 47: UML for collaboration diagrams

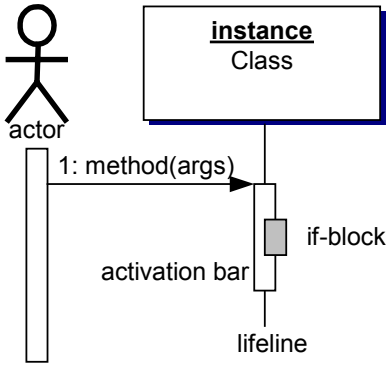

Fig. 48: UML for sequence diagrams

8 References

[1] Biggs. Discrete Mathematics (Revised Edition). Oxford University Press, New York NY, 1989.

[2] Aigner. Diskrete Mathematik. Vieweg, Braunschweig/Wiesbaden, 1993.

[3] Ottmann, Widmayer. Algorithmen und Datenstrukturen (2. Auflage). BI-Wiss.-Verl., Mannheim, Leipzig, Wien, Zürich. 1993

[4] Sedgewick. Algorithmen. Addison-Wesley, Reading MA, 1991.

[5] Nievergelt, Hinrichs. Algorithms and data structures. Prentice-Hall, Englewood Cliffs NJ, 1993.

[6] Harel. Algorithmics. Addison-Wesley, Reading MA, 1992.

[7] Bishop. Java Gently. Addison-Wesley, Reading MA, 1998.

[8] Jackson, McClellan. Java 1.2 by example. Sun Microsystems, Palo Alto CA, 1999.

[9] Flanagan. Java in a nutshell (2nd Edition). O'Really & Associates, Sebastopol CA, 1997.

[10] Stärk, Schmid, Börger. Java and the Java Virtual Machine. Springer, Heidelberg, 2001.

[11] Geary. Graphic Java 2, Volume II: Swing (3rd Edition). Sun Microsystems, Palo Alto CA, 1999.

[12] Geary. Graphic Java 2, Volume III: Advanced Swing (3rd Edition). Sun Microsystems, Palo Alto CA, 1999 (?).

[13] The Swing Tutorial. http://java.sun.com/docs/books/tutorial/index.html

[14] JTree API Specification. http://java.sun.com/j2se/1.4/docs/api/javax/swing/JTree.html

[15] Violett. The Element Interface. http://java.sun.com/products/jfc/tsc/articles/text/element_interface/

[16] View Interface API Specification. http://java.sun.com/j2se/1.4/docs/api/javax/swing/text/View.html

[17] Gamma, Helm, Johnson, Vlissides. Design Patterns. Addison-Wesley, Reading MA, 1995.

[18] Alder. The JGraph Tutorial. http://jgraph.sourceforge.net/tutorial.html

[19] The Java 1.4 API Specification. http://java.sun.com/j2se/1.4/docs/api/

[20] Alder. The JGraph 1.0 API Specification. http://api.jgraph.com

[21] Alder. JGraph. Semester Work, Department of Computer Science, ETH Zürich, Switzerland, 2001.

Part II

The JGraph Tutorial

Abstract

This document provides an experimental analysis of the JGraph component, based on working examples. Source code from the JGraphpad application will be used, together with two additional examples. One example is a diagram editor, the other is a GXL to SVG file converter for batch processing an automatic layout.

This document is *not* a specification of the API, and it does *not* provide an in-depth study of the component's architecture. The target readers are developers who need a well-documented collection of working examples that demonstrate how to use the JGraph component in a custom application.

How to use Graphs

With the JGraph class, you can display objects and their relations. A JGraph object doesn't actually contain your data; it simply provides a view of the data. Like any non-trivial Swing component, the graph gets data by querying its data model. Here's a picture of a graph:

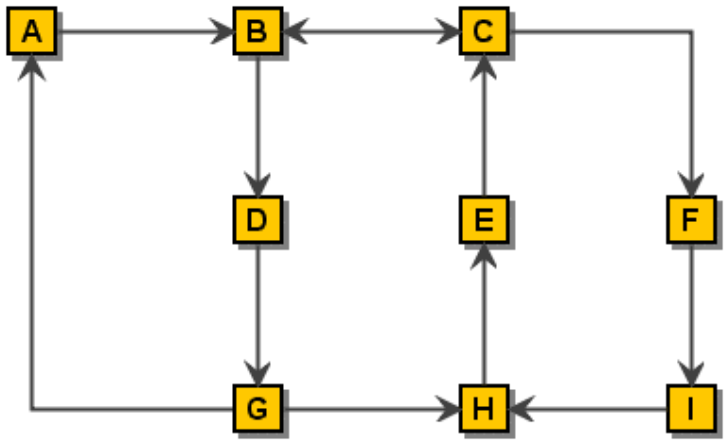

Fig. 1: A directed graph (*Digraph*)

As the preceding figure shows, JGraph displays its data by drawing individual elements. Each element displayed by the graph contains exactly one item of data, which is called a *cell*. A cell may either be a vertex, an edge or a port. Vertices have zero or more neighbours, and edges have one or no source and target vertex. Each cell has zero or more children, and one or no parent. (Instances of ports are always children of vertices.)

The rest of this tutorial discusses the following topics:

- Creating a Graph
- Customizing a Graph
- Responding to Interaction
- Customizing a Graph's Display
- Dynamically changing a Graph

Two working examples are provided:

- Client-side Example

  Implements a simple diagram editor with a custom graph model, marquee handler, tooltips, command history and a popup menu.

- Server-side Example

  Uses the JGraph component for converting a GXL file into an SVG file, and applying a simple circle-layout.

Creating a Graph

Here is a picture of an application that uses a graph in a scroll pane:

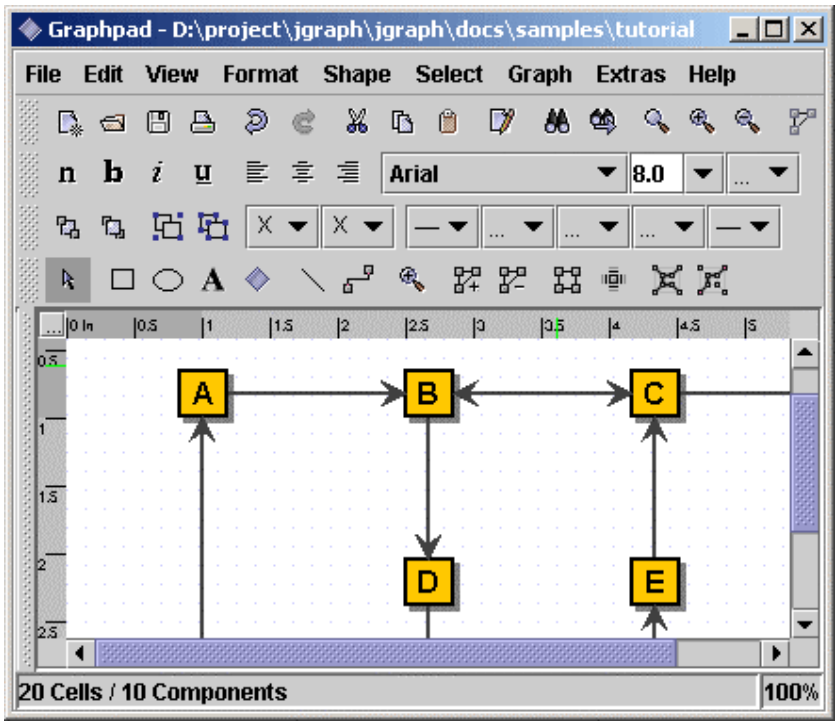

Fig. 2: An application that uses a graph in a scrollpane

The following code creates a JGraph object:

```
JGraph graph = new JGraph();
...
JScrollPane scrollPane = new JScrollPane(graph)
```

The code creates an instance of JGraph and puts it in a scroll pane. Jgraph's constructor is called with no arguments in this example. Therefore, JGraph will use an instance of `DefaultGraphModel`, create an instance of `GraphView` for it, and add a sample graph to the model.

JGraph's default implementations define the following set of keyboard bindings:

- Alt-Click forces marquee selection if over a cell
- Shift- or Ctrl-Select extends or toggles the selection
- Shift-Drag constrains the offset to one direction
- Ctrl-Drag clones the selection
- Doubleclick or F2 starts editing

To summarize, you can create a graph by invoking the JGraph constructor. You should probably put the graph inside a scroll pane, so that the graph won't take up too much space. You don't have to do anything to support cell editing, selection, marquee selection, vertex and edge resizing and moving.

Customizing a Graph

JGraph offers the following forms of interactions:

- In-place editing
- Moving
- Cloning
- Sizing
- Bending (Adding/Removing/Moving edge points)
- Establishing Connections
- Removing Connections

All interactions can be disabled using `setEnabled(false)`. In-place editing is available for both, vertices and edges, and may be disabled using `setEditable(false)`. The number of clicks that triggers editing may be changed using `setEditClickCount`.

Moving, cloning, sizing, and bending, establishing connections and disconnecting edges may be disabled using the respective methods, namely `setMoveable`, `setCloneable`, `setSizeable`, `setBendable`, `setConnectable` and `setDisconnectable` on the graph instance.

The model offers finer control of connection establishment and disconnection based on the `acceptsSource` and `acceptsTarget` methods. By overriding these methods, you can decide for each edge, port pair if it is valid with respect to the edge's source or target. (Before an edge is disconnected from a port, the respective method is called with the port set to `null` to check if disconnection is allowed.)

`CellViews` offer yet another level of control in that they allow/disallow being edited, moved, cloned, resized, and shaped, or connected/disconnected to or from other cells. (Note: In the context of multiple views, a cell may be connectable in one view, and not connectable in another.)

There are a number of additional methods to customize JGraph, for example, `setMinimumMove` to set the minimum amount of pixels before a move operation is initiated, and `setSnapSize` to define the maximum distance from a cell to be selected.

With `setDisconnectOnMove` you can indicate if the selection should be disconnected from the unselected graph when a move operation is initiated, `setDragEnabled` enables/disables the use of Drag-and-Drop, and `setDropEnabled` sets if the graph accepts Drops from external sources. (The latter also affects the clipboard, in that it allows/disallows to paste data from exernal sources.)

Responding to Interaction

You can either respond to mouse events, or to events generated by JGraph. JGraph offers the following notifications:

- Changes to the model
- Changes to the view
- Changes to the selection
- Undoable edit happened

What's "Undoable edit happened"?

JGraph is compatible to Swing's Undo-Support. Each time the user performs an action, the model dispatches an edit that describes the change. The edit provides an `undo` method to undo the change.

If `undo` is called on an edit, the model fires a `GraphModelEvent`, but it does not fire an `UndoableEditEvent`. The latter only fires if JGraph wants to indicate that an edit was added to the command history, which is not the case for an `undo`. (In the context of multiple views, you must use an instance of `GraphUndoManager` to ensure correctness.)

Responding to Mouse Events

For detection of double-clicks or when a user clicks on a cell, regardless of whether or not it was selected, a `MouseListener` should be used in conjunction with `getFirstCellForLocation`. The following code prints the label of the topmost cell under the mouse pointer on a doubleclick. (The `getFirstCellForLocation` method scales its arguments.)

```
// MouseListener that Prints the Cell on Doubleclick
graph.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
    if (e.getClickCount() == 2) {
      // Get Cell under Mousepointer
      int x = e.getX(), y = e.getY();
      Object cell = graph.getFirstCellForLocation(x, y);
      // Print Cell Label
      if (cell != null) {
        String lab = graph.convertValueToString(cell);
        System.out.println(lab);
      }
    }
  }
});
```

Responding to Model Events

If you are interested in handling model notifications, implement the `GraphModelListener` interface and add the instance using the method

`addGraphModelListener`. The listeners are notified when cells are inserted, removed, or when the label, source, target, parent or children of an object have changed. (Make sure to also add an `Observer` to `GraphView` in order to be notified of all possible changes to a graph!)

The following code defines a listener that prints out information about the changes to the model, and adds the listener to the graph's model:

```
// Define a Model Listener
public class ModelListener implements GraphModelListener {
  public void graphCellsChanged(GraphModelEvent e) {
    System.out.println("Change: "+e.getChange())
  }
}
// Add an Instance to the Model
graph.getModel().addGraphModelListener(new ModelListener());
```

Responding to View Events

Visual modifications are typically handled by the `GraphView`, which extends the `Observable` class. To respond to view-changes, implement the `Observer` interface and add it to the view using the method `addObserver`. Observers are notified when the size, position, color etc. of a cell view has changed. (Note: If the model's `isAttributeStore` returns `true`, then the view is bypassed, and all attributes are stored in the model.)

```
// Define an Observer
public class ViewObserver implements Observer {
  public void update(Observable o, Object arg) {
    System.out.println("View changed: "+o);
  }
}
// Add an Instance to the View
graph.getView().addObserver(new ViewObserver());
```

Responding to Selection Changes

The following code prints out selection events:

```
// Define a Selection Listener
public class MyListener implements GraphSelectionListener {
  public void valueChanged(GraphSelectionEvent e) {
    System.out.println("Selection changed: "+e);
  }
}
// Add an Instance to the Graph
graph.addGraphSelectionListener(new MyListener());
```

The preceding code creates an instance of `MyListener`, which implements the `GraphSelectionListener` interface, and registers it on the graph.

Responding to Undoable Edits

To enable Undo-Support, a `GraphUndoManager` must be added using `addUndoableEditListener`. The `GraphUndoManager` is an extension of Swing's `UndoManager` that maintains a command history in the context of multiple views. You can safely use an instance of `UndoManager` if your graph has only one view. Otherwise, a `GraphUndoManager` must be used for correct behaviour.

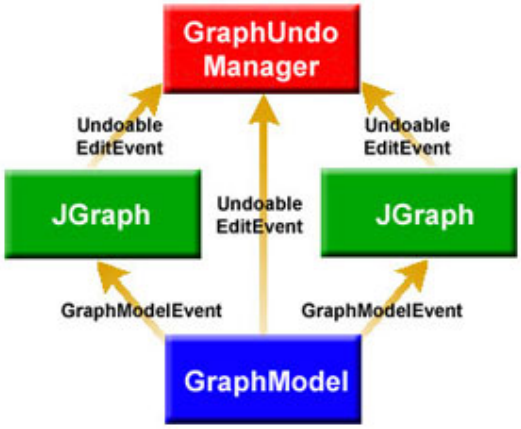

Fig. 3: Multiple Views

The figure above shows a graph model in the context of multiple views. From a logical point of view, both, `JGraph` and the `GraphModel` fire `UndoablEditEvents`. However, in reality only the model supports these events and the view uses this support to dispatch its own `UndoableEdits`. The view is updated based on `GraphModelEvents`.

Extending the Default Handlers

JGraph provides two high-level listeners that are used to control mouse and datatransfer functionality. By overriding these, you can gain full control of marquee selection and datatransfer, that is, how cells are imported and exported via Drag-and-Drop and the clipboard. The client-side example at the end of this Tutorial provides such a custom handler. The custom marquee handler in this example is used to establish connections between cells by dragging ports.

Customizing a Graphs Display

JGraph performs some look-and-feel specific painting. You can customize this painting in a limited way. For example, you can modify the grid using `setGridColor` and `setGridSize`, and you can change the handle colors

using `setHandleColor` and `setLockedHandleColor`. The background color may be changed using `setBackground`.

Subclassing Renderers

If you want finer control over the rendering, you can subclass one of the default renderers, and override its `paint`-method. A renderer is a `Component`-extension that paints a cell based on its attributes. Thus, neither JGraph nor its look-and-feel-specific implementation actually contain the code that paints a cell. Instead, JGraph uses the cell renderer's painting code. A new renderer may be associated with a cell by overriding the `getRendererComponent` method of the corresponding `CellView`, or the `getRenderer` method for extensions of the `AbstractCellView` class.

Adding new Cell Types to a Graph

The following code was taken from JGraphpad (http://start.jgraph.com) to illustrate how to add new cell types and renderers. The code adds an oval vertex to the graph. The easiest way to do this is by extending JGraph. Since JGraph implements the `CellViewFactory` interface, it is in charge of creating views.

When creating a view, JGraph assumes a cell is a vertex if it is not an instance of `Edge` or `Port`, and calls the `createVertexView` method. Thus, we only need to override this method to identify an oval vertex (based on a typetest) and return the corresponding view.

```
// Overrides JGraph.createVertexView
protected VertexView createVertexView(Object v,
                                      GraphModel model,
                                      CellMapper cm) {
  // Return an EllipseView for EllipseCells
  if (v instanceof EllipseCell)
    return new EllipseView(v, model, cm);
  // Else Call Superclass
  return super.createVertexView(v, model, cm);
}
```

The oval vertex is represented by the `EllipseCell` class, which is an extension of the `DefaultGraphCell` class, and offers no additional methods. It is only used to distinguish oval vertices from normal vertices.

```
// Define EllipseCell
public class EllipseCell extends DefaultGraphCell {
  // Empty Constructor
  public EllipseCell() {
    this(null);
  }
  // Construct Cell for Userobject
  public EllipseCell(Object userObject) {
    super(userObject);
```

```
    }
  }
```

The `EllipseView` is needed to define the special visual aspects of an ellipse. It contains an inner class which serves as a renderer that provides the painting code. The view and renderer are extensions of the `VertexView` and `VertexRenderer` classes, respectively. The methods that need to be overridden are `getPerimeterPoint` to return the perimeter point for ellipses, `getRenderer` to return the correct renderer, and the renderer's `paint` method.

```
// Define the View for an EllipseCell
public class EllipseView extends VertexView {
  static EllipseRenderer renderer = new EllipseRenderer();
  // Constructor for Superclass
  public EllipseView(Object cell, GraphModel model,
                     CellMapper cm) { super(cell, model, cm); }
  // Returns Perimeter Point for Ellipses
  public Point getPerimeterPoint(Point source, Point p) { … }
  // Returns the Renderer for this View
  protected CellViewRenderer getRenderer() {
    return renderer;
  }
  // Define the Renderer for an EllipseView
  static class EllipseRenderer extends VertexRenderer {
    public void paint(Graphics g) { … }
  }
}
```

The reason for overriding `getRenderer` instead of `getRendererComponent` is that the `AbstractCellView` class, from which we inherit, already provides a default implementation of this method that returns a configured `CellViewRenderer`, which in turn is retrieved through the method that was overridden.

Adding Tooltips to a Graph

Tooltips can be implemented by overriding JGraph's `getToolTipText` method, which is inherited from the `JComponent` class. The following displays the label of the cell under the mouse pointer as a tooltip.

```
// Return Cell Label as a Tooltip
public String getToolTipText(MouseEvent e) {
  if(e != null) {
    // Fetch Cell under Mousepointer
    Object c = getFirstCellForLocation(e.getX(), e.getY());
    if (c != null)
      // Convert Cell to String and Return
      return convertValueToString(c);
  }
  return null;
}
```

The graph must be registered with Swing's TooltipManager to correctly display tooltips. This is done with the following code on startup:

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

Customizing In-Place Editing

In graphs that display complex structures, it is quite common to offer a property dialog instead of the simple in-place editing. To do this, the `BasicGraphUI`'s `startEditing` and `completeEditing` methods must be overridden. Then, in order to use this UI in a graph, the graph's `updateUI` method must be overridden, too:

```
// Define a Graph with a Custom UI
public class DialogGraph extends JGraph {
  // Sets the Custom UI for this graph object
  public void updateUI(){
    // Install a new UI
    setUI(new DialogUI());
    invalidate();
  }
}
```

The `DialogUI` class takes the view's editor, and puts it in a dialog, which blocks the frame until the dialog is closed. The code for the DialogUI class is not printed here. It is included in the Tutorial.java file, which is available for download (see references at the end of this Tutorial).

Dynamically changing a Graph

In JGraph, either the model or the view is modified, or they are modified in parallel with a single transaction. When working on the model, objects that implement the `GraphCell` interface are used, whereas objects that implement the `CellView` interface are used in the context of a `GraphView`. `GraphViews` allow to edit `CellViews`, whereas `GraphModels` allow to insert, remove, and edit `GraphCells`.

In this chapter, a `DefaultGraphModel` is used along with a graph that provides a view to the model. This way, we can clarify which methods belong to the model, and which belong to the view.

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

Attributes

JGraph separates the model and the view. The model is defined by the `GraphModel` interface, and contains objects that implement the `GraphCell` interface, whereas the view is represented by the `GraphView` class, and

contains objects that implement the `CellView` interface. The mapping between cells and views is defined by the `CellMapper` interface:

Fig. 4: Mapping between `GraphCells` and `CellViews`

A model has zero or more views, and for each cell in the model, there exists exactly one `CellView` in each `GraphView`. The state of these objects is repesented by a map of key, value pairs. Each `CellView` combines the attirbutes from the corresponding `GraphCell` with its own attributes.

When combining the attributes from a `GraphCell` with the attributes from the `CellView`, the graph cell's attributes have precedence over the view's attributes. The special `value` attribute is in sync with the cell's user object.

Dynamically Changing Attributes

The state of a cell, and likewise of a view is represented by its attributes. In either case, the `GraphConstants` class is used to change the state in two steps:

1. Construct the object that constitutes the change
2. Execute the change on the model, or the graph view

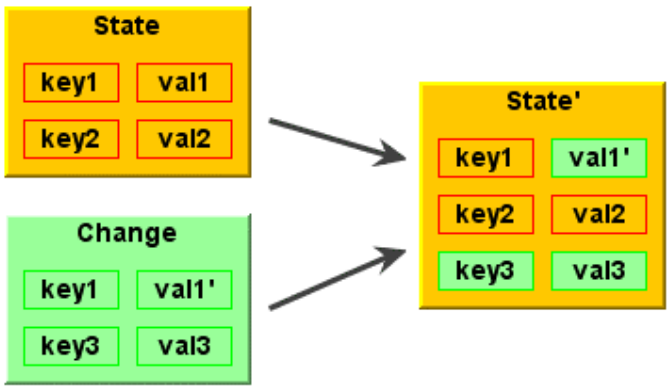

Fig. 5: Using maps to change the state

To construct the object that constitutes the change, a new map is created using the `createMap` method of the `GraphConstants` class. When a map is applied to a view's or cell's state, it does not replace the existing map. The

entries of the new map are added or changed in-place. As a consequence, the `GraphConstants` class offers the `setRemoveAttributes` and `setRemoveAll` methods, which are used to remove individual or all keys from the existing state. (Note: The notion of states using maps closely resembles the structure of XML-documents.)

Automatically Changing Attributes

The `update` method of the `CellView` interface is used to message the `CellView` when one of its attributes was changed programmatically, or one of its neighbours has changed attributes. Thus, the `update` method is a good place to automatically set attributes, like for example the points of an edge. To reflect changes to the view's corresponding cell, for example to point to the current source and target port, the `refresh` method is used.

Working with the GraphModel

You can think of the model as an access point to two independent structures: the graph structure and the group structure. The graph structure is based on the mathematical definition of a graph, ie. vertices and edges. The group structure is used to enable composition of cells, ie. parents and childs.

The graph structure is defined by the `getSource` and `getTarget` methods, which return the source and target port of an object. The port in turn is a child of a vertex, which is used as an indirection to allow multiple connection points.

The group structure is defined by the `getChild`, `getChildCount`, `getIndexOfChild`, and `getParent` methods. The objects that have no parents are called *roots*, and may be retrieved using the `getRootAt` and `getRootCount` methods.

Inserting a Vertex into the Model

Here is a method that creates a new `DefaultGraphCell` and adds it to the model. The method adds two ports to the cell, and creates a map which maps the cells to their attributes. The attributes are in turn maps of key, value pairs, which may be accessed in a type-safe way by use of the `GraphConstants` class:

```
void insertVertex(Object obj, Rectangle bounds) {
  // Map that Contains Attribute Maps
  Map attributeMap = new Hashtable();
  // Create Vertex
  DefaultGraphCell cell = new DefaultGraphCell(userObject);
  // Create Attribute Map for Cell
  Map map = GraphConstants.createMap();
  GraphConstants.setBounds(map, bounds);
  // Associate Attribute Map with Cell
```

```
    attributeMap.put(cell, map);
    // Create Default Floating Port
    DefaultPort port = new DefaultPort("Floating");
    cell.add(port);
    // Additional Port Bottom Right
    int u = GraphConstants.PERCENT;
    port = new DefaultPort("Bottomright");
    // Create Attribute Map for Port
    map = GraphConstants.createMap();
    GraphConstants.setOffset(map, new Point(u, u));
    // Associate Attribute Map with Port
    attributeMap.put(port, map);
    cell.add(port);
    // Add Cell (and Children) to the Model
    Object[] insert = new Object[]{cell};
    model.insert(insert, null, null, attributeMap);
  }
```

The first argument to the `insertVertex` method is the user object – an object that contains or points to the data associated with the cell. The user object can be a string, or it can be a custom object. If you implement a custom object, you should implement its `toString` method, so that it returns the string to be displayed for that cell. The second argument represents the bounds of the vertex, which are stored as an attribute.

Note: The vertex is passed to the `insert` method without its children. The fact that parent-child relations are stored in the cells is used here to insert the children implicitly, without providing a `ParentMap`. (Future implementations should provide an additional argument to allow separate storage.)

The `attributeMap` argument is not used by the model. It is passed to the views to provide the attributes for the cell views to be created. The third parameter of the `insert` call can be used to provide properties, that is, attributes that are stored in the model.

Finding the Port of a Vertex

Since ports are treated as normal children in the model (using the model's group structure), the `GraphModel` interface may be used to find the "default port" of a vertex:

```
Port getDefaultPort(Object vertex, GraphModel model) {
  // Iterate over all Children
  for (int i = 0; i < model.getChildCount(vertex); i++) {
    // Fetch the Child of Vertex at Index i
    Object child = model.getChild(vertex, i);
    // Check if Child is a Port
    if (child instanceof Port)
      // Return the Child as a Port
      return (Port) child;
  }
  // No Ports Found
  return null;
}
```

The code is not provided by the core API because it introduces the notion of a “default port”, which is typically application dependent. (The code above uses the first port as the default port.)

Inserting an Edge into the Model

The following method creates a new `DefaultEdge`, and adds it to the model, along with the connections to the specified source and target port.

```
void insertEdge(Object obj, Port source, Port target) {
  // Create Edge
  DefaultEdge edge = new DefaultEdge(userObject);
  // Create ConnectionSet for Insertion
  ConnectionSet cs = new ConnectionSet(edge, source, target);
  // Add Edge and Connections to the Model
  Object[] insert = new Object[]{edge};
  model.insert(insert, cs, null, null);
}
```

Again, the first argument represents the user object of the cell, and the second and third argument specify the source and target port of the new edge. To insert connections into a graph model, an instance of `ConnectionSet` is required. The instance is used to collect the new ports and targets of edges, and execute the change as a single transaction.

Removing Cells from the Model

If a cell is removed from the model, the model checks if the cell has children, and if so, updates the group structure accordingly, that is, for all parents and children that are not to be removed. As a consequence, if a cell is removed *with* children, it can be reinserted using `insert`, that is, without providing the children or a `ParentMap`. If a cell is removed *without* children, the resulting operation is an “ungroup”.

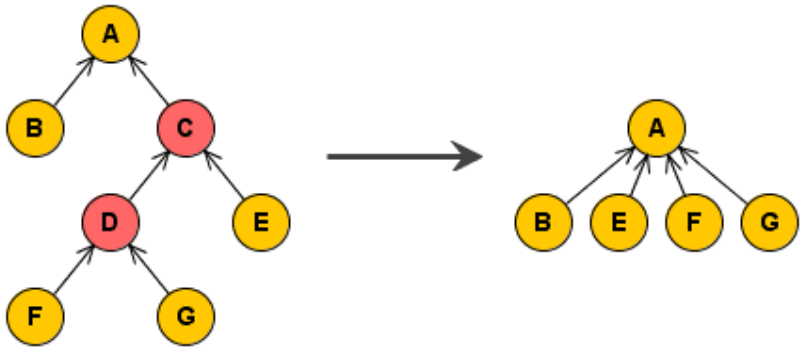

Fig. 7: Remove (ungroup) cells C and D.

The figure above shows a group A, which contains the cell B, and a group C, which in turn contains E, a group D, which in turn contains F, and G. The second figure shows the group structure after the removal of cells C and D.

The following removes all selected cells, *including* all descendants:

```
// Get Selected Cells
Object[] cells = graph.getSelectionCells();
if (cells != null) {
  // Remove Cells (incl. Descendants) from the Model
  graph.getModel().remove(graph.getDescendants(cells));
}
```

Changing the Model

The `ConnectionSet` and `ParentMap` classes are used to change the model. The attributes may be changed using a map that contains cell, attributes pairs. The attributes are in turn represented by maps of key, value pairs, which may be accessed in a type-safe way by using the `GraphConstants` class.

Let `edge`, `port` and `vertex` be instances of the `DefaultEdge`, `DefaultPort` and `DefaultGraphCell` classes, respectively; all contained in the model. This code changes the source of `edge` to `port`, the parent of `port` to `vertex`, and the user object of `vertex` to the String *Hello World*.

```
// Create Connection Set
ConnectionSet connectionSet = new ConnectionSet();
connectionSet.connect(edge, port, true);
// Create Parent Map
ParentMap parentMap = new ParentMap();
parentMap.addEntry(port, vertex);
// Create Properties for Vertex
Map properties = GraphConstants.createMap();
GraphConstants.setValue(properties, "Hello World");
// Create Property Map
Map propertyMap = new Hashtable();
propertyMap.put(vertex, properties);
// Change the Model
model.edit(connectionSet, propertyMap, parentMap, null);
```

The last argument of the `edit` call may be used to specify the initial edits that triggered the call. The edits specified are considered to be part of the transaction. (This is used to implement composite transactions, ie. transactions that change the model and the view in parallel.)

Changing the View

Each `GraphCell` has one or more associated `CellViews`, which in turn contain the attributes. The attributes of a `CellView` may be changed using `editCells` on the parent `GraphView`. In contrast to the `propertyMap`

argument used before, and the `attributeMap` argument used to insert cells into the model, the `attributeMap` argument used here contains instances of the `CellView` class as keys. The attributes, again, are maps of key, value pairs.

The following changes the border color of `vertex` to black:

```
// Work on the Graph's View
GraphView v = graph.getView();
// Create Attributes for Vertex
Map attributes = GraphConstants.createMap();
GraphConstants.setBorderColor(attributes, Color.black);
// Get the CellView to use as Key
CellView cellView = v.getMapping(vertex, false);
// Create Attribute Map
Map attributeMap = new Hashtable();
attributeMap.put(cellView, properties);
// Change the View
v.editCells(attributeMap);
```

Creating a Data Model

Extending the Default Model

The `DefaultGraphModel` provides three methods for subclassers, namely the `acceptsSource` and `acceptsTarget` method to allow or disallow certain connections to be established in a model, and the `isAttributeStore` method to gain control of the attributes that are changed by the UI. The first two methods are used in the GraphEd example in this tutorial, the latter is explained in detail in the paper. The `isOrdered` method is used to indicate if the model's order should be used instead of the view's, and the `toBack` and `toFront` methods are used to change this order. Note that the graph views methods with the same name are redirected to the model if the `isOrdered` method returns `true`.

Providing a Custom Model

If `DefaultGraphModel` doesn't suit your needs, then you will have to implement a custom data model. Your data model must implement the `GraphModel` interface. `GraphModel` accepts any kinds of objects as cells. It doesn't require that cells implement the `GraphCell` interface. You can devise your own cell representation.

Examples

GXL to SVG Converter (Gxl2svg)

In this example, JGraph is used in a batch-processing environment, instead of a user interface. The input is a GXL file, which defines a graph with vertices and edges, but not its geometric pattern. The file is parsed into a graph model, and the vertices are arranged in a circle. The display is then stored as an SVG file, using the Batik library (http://xml.apache.org/batik/).

Here is the main method:

```
// Usage: java SVGGraph gxl-file svg-file
public static void main(String[] args) {
  (…)
  // Construct the graph to hold the attributes
  JGraph graph = new JGraph(new DefaultGraphModel());
  // Read the GXL file into the model
  read(new File(args[0]), graph.getModel());
  // Apply Layout
  layout(graph);
  // Resize (Otherwise not Visible)
  graph.setSize(graph.getPreferredSize());
  (…)
  // Write the SVG file
  write(graph, out);
  (…)
}
```

Read

The `read` method creates the cells defined in the GXL file, and constructs the arguments for the insert method, namely a list of cells, a connection set, and a map, from cells to attributes. To establish the connections, the vertices in the file are given unique IDs, which are referenced from the edge's source and target. (The `ids` map is used to look-up the vertices.)

```
public static void read(File f, GraphModel model) (…) {
  (…)
  // List for the new Cells
  List newCells = new ArrayList();
  // ConnectionSet to Specify Connections
  ConnectionSet cs = new ConnectionSet();
  // Map from Cells to Attributes
  Hashtable attributes = new Hashtable();
  // Map from ID to Vertex
  Map ids = new Hashtable();
```

In the innermost loop of the read method, cells are created based on the current XML element. (Let `label` point to the element's label, and `type` point to the element's type, which is „node“ for vertices and „edge“ for edges.)

The following creates a vertex with a port:

```
// Create Vertex
if (type.equals("node")) {
  (…)
  // Fetch ID Value
  id = tmp.getNodeValue();
  // Need unique valid ID
  if (id != null && !ids.keySet().contains(id)) {
    // Create Vertex with label
    DefaultGraphCell v = new DefaultGraphCell(label);
    // Add One Floating Port
    v.add(new DefaultPort());
    // Add ID, Vertex pair to Hashtable
    ids.put(id, v);
    // Add Default Attributes
    attributes.put(v, createDefaultAttributes());
    // Update New Cell List
    newCells.add(v);
  }
}
```

Otherwise, an edge is created and its source and target IDs are looked up in the `ids` map. The first child, which is the port of the returned vertex is used to establish the connection in the connection set. (The source and target ID are analogous, so only the source ID is shown below.)

```
// Create Edge
else if (type.equals("edge")) {
  // Create Edge with label
  DefaultEdge edge = new DefaultEdge(label);
  // Fetch Source ID Value
  id = tmp.getNodeValue();
  // Find Source Port
  if (id != null) {
    // Fetch Vertex for Source ID
    DefaultGraphCell v = (DefaultGraphCell) ids.get(id);
    if (v != null)
      // Connect to Source Port
      cs.connect(edge, v.getChildAt(0), true);
  }
  (…)
  // Update New Cell List
  newCells.add(edge);
}
```

When the innermost loop exits, the new cells are inserted into the model, together with the connection set and attributes. Note that the `toArray` method is used to create the first argument, which is an array of `Objects`.

```
  // Insert the cells (View stores attributes)
  model.insert(newCells.toArray(), cs, null, attributes);
}
```

A helper method creates the attributes for the vertex, which are stored in a map, from keys to values. This map is then stored in another map, with the vertex as a key. The latter is used as a parameter for the `insert` method.

```
// Create Attributes for a Black Line Border
public static Map createDefaultAttributes() {
  // Create an AttributeMap
  Map map = GraphConstants.createMap();
  // Black Line Border (Border-Attribute must be null)
  GraphConstants.setBorderColor(map, Color.black);
  // Return the Map
  return map;
}
```

Layout

Here is the layout method. Using this method, the vertices of the graph are arranged in a circle. The method has two loops: First, all cells are filtered for vertices, and the maximum width or height is stored. (The order of the cells in the circle is equal to the order of insertion.)

```
public static void layout(JGraph graph) {
  // Fetch All Cell Views
  CellView[] views = graph.getView().getRoots();
  // List to Store the Vertices
  List vertices = new ArrayList();
  // Maximum width or height
  int max = 0;
  // Loop through all views
  for (int i = 0; i < views.length; i++) {
    // Add Vertex to List
    if (views[i] instanceof VertexView) {
      vertices.add(views[i]);
      // Fetch Bounds
      Rectangle b = views[i].getBounds();
      // Update Maximum
      if (bounds != null)
        max = Math.max(Math.max(b.width, b.height), max);
    }
  }
```

The number of vertices and the maximum width or height is then used to compute the minimum radius of the circle, and the angle step size. The step size is equal to the circle, divided by the number of vertices.

```
  // Compute Radius
  int r = (int) Math.max(vertices.size()*max/Math.PI, 100);
  // Compute Radial Step
  double phi = 2*Math.PI/vertices.size();
```

With the radius and step size at hand, the second loop is entered. In this loop, the position of the vertices is changed (without using the history).

```
  // Arrange vertices in a circle
  for (int i = 0; i < vertices.size(); i++) {
```

```
      // Cast the Object to a CellView
      CellView view = (CellView) vertices.get(i);
      // Fetch the Bounds
      Rectangle bounds = view.getBounds();
      // Update the Location
      if (bounds != null)
        bounds.setLocation(r+(int) (r*Math.sin(i*phi)),
                           r+(int) (r*Math.cos(i*phi)));
    }
  }
```

Write

Then the graph is written to an SVG file using the `write` method. To implement this method, the Batik library is used. The graph is painted onto a custom `Graphics2D`, which is able to stream the graphics out as SVG.

```
// Stream out to the Writer as SVG
public static void write(JGraph graph, Writer out) (…) {
  // Get a DOMImplementation
  DOMImplementation dom =
            GenericDOMImplementation.getDOMImplementation();
  // Create an instance of org.w3c.dom.Document
  Document doc = dom.createDocument(null, "svg", null);
  // Create an instance of the SVG Generator
  SVGGraphics2D svgGenerator = new SVGGraphics2D(doc);
  // Render into the SVG Graphics2D Implementation
  graph.paint(svgGenerator);
  // Use CSS style attribute
  boolean useCSS = true;
  // Finally, stream out SVG to the Writer
  svgGenerator.stream(out, useCSS);
}
```

Simple Diagram Editor (GraphEd)

This example provides a simple diagram editor, which has a popup menu, a command history, zoom, grouping, layering, and clipboard support. The connections between cells may be established in a special connect mode, which allows dragging the ports of a cell.

The connect mode requires a special marquee handler, which is also used to implement the popup menu. The history support is implemented by using the `GraphUndoManager` class, which extends Swing's `UndoManager`. A custom graph overrides the edge view's default bindings for adding and removing points (the right mouse button that is used for the popup menu, and shift-click is used instead). Additionally, a custom model that does *not* allow self-references is implemented.

The editor object extends `JPanel`, which can be inserted into a frame. It provides the inner classes `MyGraph`, `MyModel` and `MyMarqueeHandler`, which extend `JGraph`, `DefaultGraphModel` and `BasicMarqueeHandler` respectively. A `GraphSelectionListener` is used to update the toolbar buttons, and a `KeyListener` is responsible to handle the *delete* keystroke, however, the implementation of these interfaces is not shown below.

Here are the class definition and variable declarations:

```
public class Editor extends JPanel (…) {
  // JGraph object
  protected JGraph graph;
  // Undo Manager
  protected GraphUndoManager undoManager;
```

The `main` method constructs a frame with an editor and displays it:

```
  // Usage: java -jar graphed.jar
  public static void main(String[] args) {
    // Construct Frame
    JFrame frame = new JFrame("GraphEd");
    (…)
    // Add an Editor Panel
    frame.getContentPane().add(new Editor());
    (…)
    // Set Default Size
    frame.setSize(520, 390);
    // Show Frame
    frame.show();
  }
```

Constructor

The constructor of the editor panel creates the `MyGraph` object and initializes it with an instance of `MyModel`. Then, the `GraphUndoManager` is constructed, and the `undoableEditHappened` is overridden to enable or

disable the undo and redo action using the `updateHistoryButtons` method (which is not shown).

```
// Construct an Editor Panel
public Editor() {
  // Use Border Layout
  setLayout(new BorderLayout());
  // Construct the Graph object
  graph = new MyGraph(new MyModel());
  // Custom Graph Undo Manager
  undoManager = new GraphUndoManager() {
    // Extend Superclass
    public void undoableEditHappened(UndoableEditEvent e) {
      // First Invoke Superclass
      super.undoableEditHappened(e);
      // Update Undo/Redo Actions
      updateHistoryButtons();
    }
  };
```

The undo manager is then registered with the graph, together with the selection and key listener.

```
  // Register UndoManager with the Model
  graph.getModel().addUndoableEditListener(undoManager);
  // Register the Selection Listener for Toolbar Update
  graph.getSelectionModel().addGraphSelectionListener(this);
  // Listen for Delete Keystroke when the Graph has Focus
  graph.addKeyListener(this);
```

Finally, the panel is constructed out of the toolbar and the graph object:

```
  // Add a ToolBar
  add(createToolBar(), BorderLayout.NORTH);
  // Add the Graph as Center Component
  add(new JScrollPane(graph), BorderLayout.CENTER);
}
```

The `Editor` object provides a set of methods to change the graph, namely, an `insert`, `connect`, `group`, `ungroup`, `toFront`, `toBack`, `undo` and `redo` method, which are explained below. The cut, copy and paste actions are implemented in the JGraph core API, and are explained at the end.

Insert

The `insert` method is used to add a vertex at a specific point. The method creates the vertex, and adds a floating port.

```
// Insert a new Vertex at point
public void insert(Point pt) {
  // Construct Vertex with no Label
  DefaultGraphCell vertex = new DefaultGraphCell();
  // Add one Floating Port
  vertex.add(new DefaultPort());
```

Then the specified point is applied to the grid, and used to construct the bounds of the vertex, which are subsequently stored in the vertices' attributes, which are created and accessed using the `GraphConstants` class:

```
    // Snap the Point to the Grid
    pt = graph.snap(new Point(pt));
    // Default Size for the new Vertex
    Dimension size = new Dimension(25,25);
    // Create a Map that holds the attributes for the Vertex
    Map attr = GraphConstants.createMap();
    // Add a Bounds Attribute to the Map
    GraphConstants.setBounds(attr, new Rectangle(pt, size));
```

To make the layering visible, the vertices' attributes are initialized with a black border, and a white background:

```
    // Add a Border Color Attribute to the Map
    GraphConstants.setBorderColor(attr, Color.black);
    // Add a White Background
    GraphConstants.setBackground(attr, Color.white);
    // Make Vertex Opaque
    GraphConstants.setOpaque(attr, true);
```

Then, the new vertex and its attributes are inserted into the model. To associate the vertex with its attributes, an additional map, from the cell to its attributes is used:

```
    // Construct the argument for the Insert (Nested Map)
    Hashtable nest = new Hashtable();
    // Associate the Vertex with its Attributes
    nest.put(vertex, attr);
    // Insert wants an Object-Array
    Object[] arg = new Object[]{vertex};
    // Insert the Vertex and its Attributes
    graph.getModel().insert(arg, null, null, nest);
}
```

Connect

The `connect` method may be used to insert an edge between the specified source and target port:

```
// Insert a new Edge between source and target
public void connect(Port source, Port target) {
    // Construct Edge with no label
    DefaultEdge edge = new DefaultEdge(edge);
    // Use Edge to Connect Source and Target
    ConnectionSet cs = new ConnectionSet(edge, source, target);
```

The new edge should have an arrow at the end. Again, the attributes are created using the `GraphConstants` class. The attributes are then associated with the edge using a nested map, which is passed to the model's `insert` method:

```
    // Create a Map that holds the attributes for the edge
    Map attr = GraphConstants.createMap();
    // Add a Line End Attribute
    GraphConstants.setLineEnd(attr, GraphConstants.SIMPLE);
    // Construct a Map from cells to Maps (for insert)
    Hashtable nest = new Hashtable();
    // Associate the Edge with its Attributes
    nest.put(edge, attr);
    // Insert wants an Object-Array
    Object[] arg = new Object[]{edge};
    // Insert the Edge and its Attributes
    graph.getModel().insert(arg, cs, null, nest);
  }
```

Group

The `group` method is used to compose a new group out of an array of cells (typically the selected cells). Because the cells to be grouped are already contained in the model, a parent map must be used to change the cells' existing parents, and the cells' layering-order must be retrieved using the `order` method of the `GraphView` object.

```
  // Create a Group that Contains the Cells
  public void group(Object[] cells) {
    // Order Cells by View Layering
    cells = graph.getView().order(cells);
    // If Any Cells in View
    if (cells != null && cells.length > 0) {
      // Create Group Cell
      DefaultGraphCell group = new DefaultGraphCell();
```

In the following, the entries of the parent map are created by associating the existing cells (children) with the new cell (parent). The parent map and the new cell are then passed to the model's `insert` method to create the group:

```
      // Create Change Information
      ParentMap map = new ParentMap();
      // Insert Child Parent Entries
      for (int i = 0; i < cells.length; i++)
        map.addEntry(cells[i], group);
      // Insert wants an Object-Array
      Object[] arg = new Object[]{group};
      // Insert into model
      graph.getModel().insert(arg, null, map, null);
    }
  }
```

Ungroup

The inverse of the above is the `ungroup` method, which is used to replace groups by their children. Since the model makes no distinction between ports and children, we use the cell's *view* to identify a group. While the `getChildCount` for vertices with ports returns the number of ports, the corresponding `VertexView`'s `isLeaf` method only returns `true` if at least one child is not a port, which is the definition of a group:

```
  // Determines if a Cell is a Group
  public boolean isGroup(Object cell) {
    // Map the Cell to its View
    CellView view = graph.getView().getMapping(cell, false);
    if (view != null)
      return !view.isLeaf();
    return false;
  }
```

Using the above method, the `ungroup` method is able to identify the groups in the array, and store these groups in a list for later removal. Additionally, the group's children are added to a list that makes up the future selection.

```
// Ungroup the Groups in Cells and Select the Children
public void ungroup(Object[] cells) {
  // Shortcut to the model
  GraphModel m = graph.getModel();
  // If any Cells
  if (cells != null && cells.length > 0) {
    // List that Holds the Groups
    ArrayList groups = new ArrayList();
    // List that Holds the Children
    ArrayList children = new ArrayList();
    // Loop Cells
    for (int i = 0; i < cells.length; i++) {
      // If Cell is a Group
      if (isGroup(cells[i])) {
        // Add to List of Groups
        groups.add(cells[i]);
        // Loop Children of Cell
        for (int j = 0; j < m.getChildCount(cells[i]); j++)
          // Get Child from Model
          children.add(m.getChild(cells[i], j));
      }
    }
    // Remove Groups from Model (Without Children)
    m.remove(groups.toArray());
    // Select Children
    graph.setSelectionCells(children.toArray());
  }
}
```

To Front / To Back

The following methods use the graph view's functionality to change the layering of cells, or get redirected to the model based on `isOrdered`.

```
// Brings the Specified Cells to Front
public void toFront(Object[] c) {
  if (c != null && c.length > 0)
   graph.getView().toFront(graph.getView().getMapping(c));
}

// Sends the Specified Cells to Back
public void toBack(Object[] c) {
  if (c != null && c.length > 0)
   graph.getView().toBack(graph.getView().getMapping(c));
}
```

Undo / Redo

The following methods use the undo manager's functionality to undo or redo a change to the model or the graph view:

```
// Undo the last Change to the Model or the View
public void undo() {
  try {
    undoManager.undo(graph.getView());
  } catch (Exception ex) {
    System.err.println(ex);
  } finally {
    updateHistoryButtons();
  }
}

// Redo the last Change to the Model or the View
public void redo() {
  try {
    undoManager.redo(graph.getView());
  } catch (Exception ex) {
    System.err.println(ex);
  } finally {
    updateHistoryButtons();
  }
}
```

Graph Component

Let's look at the implementation of the `MyGraph` class, which is GraphEd's main UI component. A custom graph is necessary to override the default implementation's use of the right mouse button to change an edge's points, and also to change flags, and to use the custom model and marquee handler, which are printed below. The `MyGraph` class is implemented as an inner class:

```
// Custom Graph
public class MyGraph extends JGraph {

  // Construct the Graph using the Model as its Data Source
  public MyGraph(GraphModel model) {
    super(model);
    // Use a Custom Marquee Handler
    setMarqueeHandler(new MyMarqueeHandler());
    // Tell the Graph to Select new Cells upon Insertion
    setSelectNewCells(true);
    // Make Ports Visible by Default
    setPortsVisible(true);
    // Use the Grid (but don't make it Visible)
    setGridEnabled(true);
    // Set the Grid Size to 6 Pixel
    setGridSize(6);
    // Set the Snap Size to 1 Pixel
    setSnapSize(1);
  }
```

To override the right mouse button trigger with a shift trigger, an indirection is used. By overriding the `createEdgeView` method, we can define our own

`EdgeView` class, which in turn overrides the `isAddPointEvent` method, and `isRemovePointEvent` method to check the desired trigger:

```
    // Override Superclass Method to Return Custom EdgeView
    protected EdgeView createEdgeView(Edge e, CellMapper c) {
      // Return Custom EdgeView
      return new EdgeView(e, this, c) {
        // Points are Added using Shift-Click
        public boolean isAddPointEvent(MouseEvent event) {
          return event.isShiftDown();
        }
        // Points are Removed using Shift-Click
        public boolean isRemovePointEvent(MouseEvent event) {
          return event.isShiftDown();
        }
      };
    }
  }
```

Graph Model

The custom model extends `DefaultGraphModel`, and overrides its `acceptsSource` and `acceptsTarget` methods. The methods prevent self-references, that is, if the specified port is equal to the existing source or target port, then they return `false`:

```
  // A Custom Model that does not allow Self-References
  public class MyModel extends DefaultGraphModel {
    // Source only Valid if not Equal to Target
    public boolean acceptsSource(Object edge, Object port) {
       return (((Edge) edge).getTarget() != port);
    }
    // Target only Valid if not Equal to Source
    public boolean acceptsTarget(Object edge, Object port) {
      return (((Edge) edge).getSource() != port);
    }
  }
```

Marquee Handler

The idea of the marquee handler is to act as a „high-level“ mouse handler, with additional painting capabilites. Here is the inner class definition:

```
  // Connect Vertices and Display Popup Menus
  public class MyMarqueeHandler extends BasicMarqueeHandler {

    // Holds the Start and the Current Point
    protected Point start, current;

    // Holds the First and the Current Port
    protected PortView port, firstPort;
```

The `isForceMarqueeEvent` method is used to fetch the subsequent `mousePressed`, `mouseDragged` and `mouseReleased` events. Thus, the marquee handler may be used to gain control over the mouse. The argument

to the method is the event that triggered the call, namely the `mousePressed` event. (The graph's `portsVisible` flag is used to toggle the connect mode.)

```
// Gain Control (for PopupMenu and ConnectMode)
public boolean isForceMarqueeEvent(MouseEvent e) {
  // Wants to Display the PopupMenu
  if (SwingUtilities.isRightMouseButton(e))
    return true;
  // Find and Remember Port
  port = getSourcePortAt(e.getPoint());
  // If Port Found and in ConnectMode (=Ports Visible)
  if (port != null && graph.isPortsVisible())
    return true;
  // Else Call Superclass
  return super.isForceMarqueeEvent(e);
}
```

The `mousePressed` method is used to display the popup menu, or to initiate the connection establishment, if the global port variable has been set.

```
// Display PopupMenu or Remember Location and Port
public void mousePressed(final MouseEvent e) {
  // If Right Mouse Button
  if (SwingUtilities.isRightMouseButton(e)) {
    // Scale From Screen to Model
    Point l = graph.fromScreen(e.getPoint());
    // Find Cell in Model Coordinates
    Object c = graph.getFirstCellForLocation(l.x,l.y);
    // Create PopupMenu for the Cell
    JPopupMenu menu = createPopupMenu(e.getPoint(), c);
    // Display PopupMenu
    menu.show(graph, e.getX(), e.getY());
  // Else if in ConnectMode and Remembered Port is Valid
  } else if (port != null && !e.isConsumed() &&
              graph.isPortsVisible())
  { // Remember Start Location
    start = graph.toScreen(port.getLocation(null));
    // Remember First Port
    firstPort = port;
    // Consume Event
    e.consume();
  } else
    // Call Superclass
    super.mousePressed(e);
}
```

The `mouseDragged` method is messaged repeatedly, before the `mouseReleased` method is invoked. The method is used to provide the live-preview, that is, to draw a line between the source and target port for visual feedback:

```
// Find Port under Mouse and Repaint Connector
public void mouseDragged(MouseEvent e) {
  // If remembered Start Point is Valid
  if (start != null && !e.isConsumed()) {
```

```
        // Fetch Graphics from Graph
        Graphics g = graph.getGraphics();
        // Xor-Paint the old Connector (Hide old Connector)
        paintConnector(Color.black, graph.getBackground(), g);
        // Reset Remembered Port
        port = getTargetPortAt(e.getPoint());
        // If Port was found then Point to Port Location
        if (port != null)
          current = graph.toScreen(port.getLocation(null));
        // Else If no Port found Point to Mouse Location
        else
          current = graph.snap(e.getPoint());
        // Xor-Paint the new Connector
        paintConnector(graph.getBackground(), Color.black, g);
        // Consume Event
        e.consume();
      }
      // Call Superclass
      super.mouseDragged(e);
    }
```

The following method is called when the mouse button is released. If a valid source and target port exist, the connection is established using the editor's `connect` method:

```
    // Establish the Connection
    public void mouseReleased(MouseEvent e) {
      // If Valid Event, Current and First Port
      if (e != null && !e.isConsumed() && port != null &&
          firstPort != null && firstPort != port)
       {
          // Fetch the Underlying Source Port
          Port source = (Port) firstPort.getCell();
          // Fetch the Underlying Target Port
          Port target = (Port) port.getCell();
          // Then Establish Connection
          connect(source, target);
          // Consume Event
          e.consume();
      } else {
        // Else Repaint the Graph
        graph.repaint();
      }
      // Reset Global Vars
      firstPort = port = null;
      start = current = null;
      // Call Superclass
      super.mouseReleased(e);
    }
```

The marquee handler also implements the `mouseMoved` method, which is messaged independently of the others, to change the mouse pointer when over a port:

```
  // Show Special Cursor if Over Port
  public void mouseMoved(MouseEvent e) {
```

```
    // Check Mode and Find Port
    if (e != null && getSourcePortAt(e.getPoint()) != null &&
        !e.isConsumed() && graph.isPortsVisible())
    {
      // Set Cusor on Graph (Automatically Reset)
      graph.setCursor(new Cursor(Cursor.HAND_CURSOR));
      // Consume Event
      e.consume();
    }
    // Call Superclass
    super.mouseReleased(e);
  }
```

Here are the helper methods used by the custom marquee handler. The first is simply used to retrieve the port at a specified position. (The method is named `getSourcePortAt` because another method must be used to retrieve the target port.)

```
    // Returns the Port at the specified Position
    public PortView getSourcePortAt(Point point) {
      // Scale from Screen to Model
      Point tmp = graph.fromScreen(new Point(point));
      // Find a Port View in Model Coordinates and Remember
      return graph.getPortViewAt(tmp.x, tmp.y);
    }
```

The `getTargetPortAt` checks if there is a cell under the mouse pointer, and if one is found, it returns its „default“ port (first port).

```
    // Find a Cell and Return its Default Port
    protected PortView getTargetPortAt(Point p) {
      // Find Cell at point (No scaling needed here)
      Object cell = graph.getFirstCellForLocation(p.x, p.y);
      // Shortcut Variable
      GraphModel model = graph.getModel();
      // Loop Children to find first PortView
      for (int i = 0; i < model.getChildCount(cell); i++) {
        // Get Child from Model
        Object tmp = graph.getModel().getChild(cell, i);
        // Map Cell to View
        tmp = graph.getView().getMapping(tmp, false);
        // If is Port View and not equal to First Port
        if (tmp instanceof PortView && tmp != firstPort)
          // Return as PortView
          return (PortView) tmp;
      }
      // No Port View found
      return getSourcePortAt(point);
    }
```

The `paintConnector` method displays a preview of the edge to be inserted. (The `paintPort` method is not shown.)

```
    // Use Xor-Mode on Graphics to Paint Connector
    void paintConnector(Color fg, Color bg, Graphics g) {
```

```
      // Set Foreground
      g.setColor(fg);
      // Set Xor-Mode Color
      g.setXORMode(bg);
      // Highlight the Current Port
      paintPort(graph.getGraphics());
      // If Valid First Port, Start and Current Point
      if (firstPort != null && start != null &&
          current != null)
      {
        // Then Draw A Line From Start to Current Point
        g.drawLine(start.x, start.y, current.x, current.y);
      }

    } // End of Editor.MyMarqueeHandler
```

The rest of the Editor class implements the methods to create the popup menu (not shown), and the toolbar. These methods mostly deal with the creation of action objects, but the copy, paste, and cut actions are exceptions:

```
// Creates a Toolbar
public JToolBar createToolBar() {
  JToolBar toolbar = new JToolBar();
  toolbar.setFloatable(false);
  (…)
  // Copy
  action = graph.getTransferHandler().getCopyAction();
  url = getClass().getClassLoader().getResource("copy.gif");
  action.putValue(Action.SMALL_ICON, new ImageIcon(url));
  toolbar.add(copy = new EventRedirector(action));
  (…)
}
```

Because the source of an event that is executed from the toolbar is the `JToolBar` instance, and the copy, cut and paste actions assume a graph as the source, an indirection must be used to change the source to point to the graph:

```
// This will change the source of the actionevent to graph.
protected class EventRedirector extends AbstractAction {

  protected Action action;

  // Construct the "Wrapper" Action
  public EventRedirector(Action a) {
    super("", (ImageIcon) a.getValue(Action.SMALL_ICON));
    this.action = a;
  }

  // Redirect the Actionevent to the Wrapped Action
  public void actionPerformed(ActionEvent e) {
    e = new ActionEvent(graph, e.getID(),
        e.getActionCommand(), e.getModifiers());
    action.actionPerformed(e);
  }
}
```

References

The source code used in this document may be obtained from:

- http://jgraph.sourceforge.net/downloads/Tutorial.java
- http://jgraph.sourceforge.net/downloads/gxl2svg.zip
- http://jgraph.sourceforge.net/downloads/graphed.zip

All figures (except Fig. 3) have been created using JGraphpad. For more information on JGraphpad and the JGraph component visit the JGraph Home Page at http://www.jgraph.com/.

JGraphpad is available for Web Start at http://start.jgraph.com/.

The JGraph API specification is at http://api.jgraph.com/.

Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

TIK in Zusammenarbeit mit IfE Wintersemester 2001/02

DIPLOMARBEIT

für
Herrn Gaudenz Alder

CAOS Configuration Tool

Betreuer: Men Muheim, ETZ H60.1
Stellvertreter: Mathias Stäger, ETZ H61.1

Ausgabe: 30. Oktober 2001
Abgabe: 28. Februar 2002

Einleitung

In einer Semesterarbeit wurde eine Java Swing Komponente (JGraph [1]) für die Visualisierung von Graphen entwickelt. Mögliche Anwendungen von JGraph reichen von Graphik Design Tools über Netzwerkvisualisierung bis hin zu Software Programmierumgebungen.

In unserem Fall ist die Anwendung ein graphisches Konfigurationstool, mit dem Softwarekomponenten auf den Rechnerknoten instanziert und miteinander virtuell verdrahtet werden können.

Ziel dieser Arbeit ist es, JGraph so zu erweitern, dass es zur Implementierung dieses Tools verwendet werden kann. Die Arbeit bietet ebenfalls Anlass, weitere Funktionen und Änderungen an der Komponente anzubringen. Die neue Architektur soll ausführlich getestet und dokumentiert sein. Die Dokumentation beschreibt diese Architektur und deren Anwendung anhand von Beispielen.

Dabei gilt es, die existierende Software Komponente so zu perfektionieren, dass sie als open-source Bibliothek verwendet werden kann. Die Anwendung von JGraph im oben erwähnten graphischen Konfigurationstool soll als Beispiel in die ausführliche Dokumentation einfliessen.

Aufgabenstellung

1. Machen Sie sich mit dem existierenden graphischen CAOS Konfigurationstool vertraut. Prüfen Sie in Zusammenarbeit mit Ihrem Betreuer die Konzepte von JGraph anhand von neuen Features (z.B. Grouping) für das Konfigurationstool.

2. Erarbeiten Sie die angestrebte JGraph Architektur und Implementieren sie diese.

3. Welches sind die inhaltlichen und didaktischen Ziele der Dokumentation?

4. Schreiben Sie die Dokumentation und Publizieren Sie die Arbeit als open-source Paket auf dem Internet.

Durchführung der Diplomarbeit

Allgemeines

- Der Verlauf des Projektes "Diplomarbeit" soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.

- Sie verfügen über einen Arbeitsplatz an der ETH.

- Stellen Sie Ihr Projekt zu Beginn der Diplomarbeit in einem Kurzvortrag (am *1. November 2001*, 5 Minuten) vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums vom *25. Februar 2002* (20 Minuten).

- Besprechen Sie Ihr Vorgehen regelmässig mit Ihrem Betreuer.

Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichtes spätestens am *28. Februar 2002* dem betreuenden Assistenten oder seinem Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden (vgl. [5], Kap. 1.7 Bericht).

- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen usw. bestehen bleiben. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.

Zürich, den 30. Oktober 2001

Prof. Dr. B. Plattner

Literaturverzeichnis

[1] Gaudenz Alder. JGraph Website. http://www.jgraph.com/.

[2] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Publishing, 1994.

[3] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing, 1994.

[4] Manfred Schneider. Cetus links. http://www.cetus-links.org/.

[5] M. Thaler. *Semester- und Diplomarbeiten am Institut für Elektronik*. 10/95 edition, Oktober 1995.