1. High-Level Overview

1.1 Introduction
QuarkGUI is a Java application that allows users to view and manipulate YQGM graphs generated by the Quark XML database system, as well as execute queries on a remote Quark server. This document focuses on the manipulation of YQGM graphs, which corresponds to the package yqgm_gui_java. Other components, such as edu.cornell.cs.quark.gui, are relatively simple in functionality and implementation. The code are easy to follow and their explanations are skipped in this documentation.

Each subsequent section corresponds to a sub-directory in yqgm_gui_java, which is a logical component of yqgm_gui_java.

1.2 edu.cornell.cs.quark.yqgm_gui.graph
This package provides a mechanism for rendering arbitrary directed graphs as Swing components. It is designed to be extensible and easily applicable to other purposes than the display of YQGM graphs.

This component is organized as follows: a graph contains objects (GraphObject), which may be either nodes (GraphNode) or edges (GraphEdge). These objects are the “logical” components of the graph; they have properties associated with them (such as a label, and the PropertySheet [a tabbed component for conveying information about the object] displayed when the object is selected) but know nothing about how they’re actually rendered as Swing components. The actual components that render these objects are called widgets; there is an interface GraphObjectWidget which is implemented by GraphNodeWidget and GraphEdgeWidget. Each of these is rendered using the properties of the associated GraphObject.

Finally, all of this comes together on the canvas (GraphCanvas) onto which the graph is drawn. The GraphCanvas itself is a container which contains GraphObjectWidgets; the user can select an object by clicking on its widget, or move the widget by dragging it (only if it’s a GraphNodeWidget). When an object is selected, its PropertySheet is drawn in the infoBox, an arbitrary JComponent that is given to the canvas’s constructor (in our particular implementation, the infoBox is a JPanel appearing to the right of the GraphCanvas).
The standard use case for the yqgm_gui.graph component is as follows:
0. Write classes implementing the GraphNode and GraphEdge interfaces
1. Create a new instance of GraphCanvas
2. For each node/edge in your graph, call addNode() and addEdge() on the canvas
3. Call layoutGraph() on the canvas to lay out the graph in a tree structure
4. Add the canvas to a JScro llPane in your application

1.3 edu.cornell.cs.quark.yqgm_gui.yqgm
This package contains the data structures used to represent YQGM graph elements in Java, and also contains the code used to parse an XML document containing a graph into these data structures.

The YQGMEElement interface is implemented by all YQGM elements that have IDs (in other words, all elements other than the top-level query). Each such element actually has two IDs: a GUID (globally-unique ID) which is the ID number assigned to it by the Quark server, and a local ID assigned to it by its parent (where “parent” is defined by the spanning tree; for example, a Query is the parent of an Operator, which is the parent of OutputColumns, etc.). Each element, including Query, has a constructor which takes in a DOM node and parses the XML fragment rooted at that node. The constructors of any child elements are also called; therefore, a call to Query’s constructor results in parsing the entire graph.

The Operator class implements GraphNode; the Quantifier and Correlation classes implement GraphEdge.

The standard use case for the yqgm_gui.yqgm component is as follows:
1. Create a new instance of Query, passing it a Document containing the XML representation of an YQGM graph.
2. Call the query’s draw() method, passing it the GraphCanvas instance onto which to render the graph. This method automatically calls the layoutGraph() method on the canvas.

1.4 edu.cornell.cs.quark.yqgm_gui.util
This package contains a number of classes which serve various functions in yqgm_gui_java and don’t really belong anywhere else:
- **ActionProxy** is a class implementing the Action interface, allowing one Action to take on different meanings depending on context. See 2.1.
- **DOMUtil** contains several static methods implementing commonly-needed functionality when dealing with DOM trees.
- **GUICallBack** is used to maintain a level of abstraction between the top layer of an application (user interface) and a lower layer (data processing). See 2.2.
- **PropertyList** is an ordered list of (String key, String value) pairs.
- **ScrollablePane** is a specialized component which automatically resizes and scrolls itself when placed inside of a JViewport. See 2.3.
- **Util** provides a collection of general-purpose static methods which don’t belong anywhere else.
2. Detailed Description of GUI Functionality

This section decomposes the graphical user interface aspect of the application. Each subsection describes, in grotesque and probably painful detail, the purpose and implementation details of a particular feature of the application.

2.1 Actions

In general, an Action¹ is associated with (a) one or more GUI elements, such as a menu item or a toolbar button, and (b) a piece of code (actionPerformed()) that is executed when the action is fired. In addition, it can be enabled/disabled (in which case all GUI elements associated with it are either enabled or grayed out) and assigned other properties as (String key, Object value) pairs. This serves as a useful tool to allow the programmer to communicate simultaneously with all GUI elements that fire a specific action, resulting in cleaner code.

One thing that Actions don’t handle well is the concept of context. For example, consider the Action associated with the “Back” button. This button selects the previously-selected object in the current graph, and is enabled only if it is possible to move back in the history. Therefore, the enabled state of the button, as well as the actionPerformed() code for it, depend on the currently-selected window, which suggests that the application should change the button’s Action when the focus changes to a different window (i.e. each GraphCanvas should have its own back Action). However, this eliminates the main benefit of the Action interface; we now have to call setAction() on each component associated with the back action. So if we add a menu item called “Back”, we need to update its action when the focus changes. This adds complexity to the code, with potential for new bugs (if we forget to add the new code to the focus-change handler, the “Back” menu item will appear to work at first but will break as soon as the focus changes). Another option is to have a single back Action at the application level, which calls the back() method of the currently-selected GraphCanvas. The problem is that it is not clear how the GraphCanvas can change the enabled state of the Action; it would break a level of abstraction if the GraphCanvas required knowledge of the concept of a “currently-selected window”, since it only requires knowledge of itself and its infoBox.

The solution is ActionProxy, a class implementing the Action interface. As its name suggests, it is a proxy for another action (the “proxied” action): its enabled/disabled state, properties, and actionPerformed() method behave as though they belonged to the proxied Action. The ActionProxy has a setAction() method, which changes the proxied Action, and notifies all PropertyChangeListeners if any of the properties have changed (for example, if the previous proxied Action was disabled, and the new one was enabled, then the Swing components which had previously registered interest in PropertyChangeEvent would be notified of the change). If setAction() is called with a null Action, the ActionProxy assumes a disabled state, no additional properties, and actionPerformed() is a no-op. This is the state it begins in.

¹ See http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/Action.html
ActionProxy is used in yqgm_gui_java as follows: GraphCanvas exposes a number of Actions, such as the back and forward Actions. The main application also has a back and forward Action, each of which is an instance of ActionProxy. When the focus changes, the event handler in the main application calls setAction() on each of these proxies, setting the proxied action to the corresponding Action in the newly-selected GraphCanvas, or null if the selected window is not a GraphCanvas window.

A final note on Actions: there is one unconventional use of Actions in this application, which is to enable/disable global options shared by all GraphObjects. Currently this is used for the stickyTabs option, but it can be easily extended to include other options. There is a single Action, optionsAction, in the main application. This Action finds the action command of the button (in this case, a JCheckboxMenuItem) which triggered it, and the selected state of that button (i.e. whether or not the menu item is checked), and sets the value (action command, new Boolean(selected)) in its properties. Thus, to add a new global option, the user just needs to create a new JCheckboxMenuItem using optionsAction, and call the menu item’s setActionCommand() method with the unique name of the option. The optionsAction is passed to the GraphCanvas constructor when a new GraphCanvas is created. The GraphCanvas then uses the value of optionsAction.getValue(“stickyTabs”) to determine whether or not stickyTabs mode should be enabled. It is conceivable that enabling or disabling such a global action might actually require an immediate response from all GraphCanvases (for example, changing the global font size); this extension could be achieved quite easily with this infrastructure, as the GraphCanvas constructor would just have to call addPropertyChangeListener() on the optionsAction to register interest in changes to the options. However, in this particular case, changing the stickyTabs option only affects future behavior so GraphCanvas doesn’t register a new PropertyChangeListener.

2.2 GUICallBack

The GUICallBack class allows user interaction without breaking the abstraction between the GUI layer of the application and the “guts” of the lower layers, which have no knowledge of the way the user interface works (as well they shouldn’t!). The upper layers can pass an instance of GUICallBack to the lower layers, which then use methods inside GUICallBack to communicate errors and other messages to the user.

Currently, the following methods are supported:
- notice(): Shows the user a message
- warn(): Shows the user a message, possibly with an associated exception, and gives him the option of aborting or continuing
- error(): Indicate to the user that a fatal exception has occurred (possibly with an exception attached)

The warn() method is different from the other two in that it can throw an exception, UserAbortedException, if the user chooses the “abort” option. The other two options only present the user with one choice (in the case of notice(), it is assumed that there is no problem; in the case of error() it is assumed that the problem is fatal and won’t allow the user to continue anyway). The UserAbortedException will include the chained
exception, if any. This exception can then be either caught by the lower level and handled there; more likely, it will be propagated back to the interface level.

A future improvement would be to make GUICallBack an interface, and create an implementation equivalent to the current GUICallBack class, which uses Swing dialog boxes to communicate with the user. However, we omit this improvement, as there is no foreseeable need for any other methods of user interaction in Yqgm_gui_java.

2.3 Swing trickery: Fighting with scrollbars
One aspect of Swing that is rather deficient is scrollbar support. We combat this particular deficiency on two fronts. The first is in JDesktopPane.

The JDesktopPane\(^2\) class lays down the framework for MDI (multiple-document interface) in Java. It creates a virtual desktop onto which MDI child windows (JInternalFrame) can be added. Unfortunately, the implementation is rather primitive, and doesn’t do many things developers might take for granted: for example, a “Window” menu containing the list of all MDI children, and scrollbars that automatically appear when an MDI child is dragged past the bounds of the application window. A while ago, Tom Tessier wrote an article\(^3\) in JavaWorld providing an extension to the Java MDI framework addressing these limitations. This open-source package has since been updated and now lives at http://jscroll.sourceforge.net. Yqgm_gui_java uses a slightly modified version of jscroll for its MDI interface. The slight modifications are:

- The button bar is at the bottom, not the top, of the screen.
- The underlying JDesktopManager is exposed, allowing internal frames to be programmatically maximized.

Truth be told, jscroll is rather poorly designed (the fact that the above two changes needed to be made to its source code shows the authors’ lack of concern for extensibility) but unfortunately it is the best there is. A few notes about bugs/inherent limitations of jscroll, which may be of use to any developer who needs to deal with it:

- Only outline dragging (as opposed to “live” dragging, where the window contents are shown as it is moved) is allowed for internal frames. Briefly, the reason for this is that when a frame is dragged past the upper/left corner of the desktop, it compensates by enlarging the desktop and shifting all frames down/right. However, if this shift occurs while the frame is being dragged (which is the case when live drag mode is enabled, but not with outline dragging) then the frame’s mouse motion handler has inconsistent information about the position the window was in when dragging started. In other words, enabling live drag mode for the jscroll framework would involve many more changes than is immediately apparent.

\(^2\) http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JDesktopPane.html
\(^3\) http://www.javaworld.com/javaworld/jw-11-2001/jw-1130-jscroll_p.html
- When you call add(JInternalFrame) on the JScrollDesktopPane, **do not expect InternalFrameListeners registered with that frame to be fired!** The reason is that a “wrapper” is created around the frame, but the wrapper does not propagate InternalFrameEvents to the original frame. Once again, this shows the lack of concern by the authors for universal applicability. This is the sort of thing that makes me want to rewrite a piece of code myself, because it would probably take less time to write it correctly than to deal with the idiosyncrasies of this package.

A second area in which scrollbars are an issue is in the GraphCanvas, where (quite similarly to the issue that jscroll addresses, actually) we need to handle the fact that users can drag items around and may need to resize the window in response. This is tackled using a class called ScrollablePane, a subclass of JPanel, which exhibits the following behavior:

- If a child component has moved (presumably because it was dragged by the user), ensure that all components are within the bounds of the pane. If they are not, enlarge the pane (and, if the component moved outside the upper/left side, shift everything over to accommodate the change).
- If the user manipulates the containing viewport (that is, scrolls using the scrollbars), then attempt to shrink down any part of the pane that is outside of the viewable area.

This results in an intuitive dynamics in which dragging a component outside of the pane causes the pane to scroll over to accommodate the action (so if you drag a graph node up past the top of the pane, it will scroll up), but it will not shrink any visible part of the pane (so if you drag that node back down, you can leave “empty space” above it). However, if you then use the scrollbars, any such “empty space” which is not in the visible area will be removed.

GraphCanvas is a subclass of ScrollablePane.

Note that a ScrollablePane does not actually contain a JScrollPane. Instead, a ScrollablePane can be added to a JScrollPane (or any other container that uses JViewports) and has a HierarchyListener which monitors for a change in its parent.