**Galant** Developer’s Guide

This document describes the Galant infrastructure for continued development and support.

**Contents**

1. Project Environment 2
2. Macros 2
3. Graph Components/API 5
4. Graph Dispatch 7
5. Visual Graph Editor 7
6. Textual Graph Editor 9
7. GraphML Parser 10
8. Preferences 10
9. Automatic Graph Repositioning 11
10. Running an Algorithm 12

**1. Project Environment**

Galant is a Java 6 project with no external dependencies. The project can be imported as an Eclipse project or developed without an IDE.

Compilation:

If no IDE is preferred, an ant script is provided to build and bundle the code into a .jar file. To execute, point a terminal at the project root and run the ‘ant’ command with no arguments.

If Eclipse is used, the distributable product can be built by File > Export... > Runnable JAR File

**2. Macros**

The Macro class forms the base of the macro structure. Each macro is represented by an instance of Macro, which contains two main parts: a Pattern used to find instances of text to be replaced, and the modify(String code, MatchResult match) method, which returns a string to replace it with.

This string is a replacement string as used with Matcher.appendReplacement(StringBuffer sb, String replacement). This means that references to capturing groups from the match can be included with $*num*, where *num* is the number of the capturing group. Dollar signs and backslashes can be escaped with backslashes; to ensure a literal string, Matcher.quoteReplacement(String s) can be used.

modify can also return null, which indicates that the given match is not actually a valid match. This can be used to check conditions that aren’t included in the macro’s Pattern.

Note that at the moment, macros are unable to ignore text in string literals or comments.

ParameterizedMacro and MacroUtil extend this basic structure. ParameterizedMacro provides an easy way to create macros that take arguments and optionally include a block of code. MacroUtil contains utility methods for creating macros.

Of particular note are those for dealing with stuff like nested brackets. If a macro involves an area of arbitrary code (for example, as an argument), it most likely wants to only deal with the top level of that code, ignoring stuff like the arguments passed to a method call within that code. This is quite simply beyond the capabilities of regular expressions. There are two related methods in MacroUtil that help with handling this situation: nestedRegex(String prefix, String suffix) and evaluateNestedRegexMatch(List<Pair<Character, Character>> blockDelimiters, String openDelim, String closeDelim, String match, String... toFind).

nestedRegex creates a regex to match the given prefix and suffix with arbitrary code in between. This will match something of the form prefix-*code*-suffix. evaluateNestedRegexMatch takes *code* between the prefix and suffix from nestedRegex and produces a NestedRegexResult. This splits *code* into two parts: *match* and *extra*, resulting in prefix-*match*-*extra*-suffix. *extra*, if it exists, always starts with closeDelim; this is, however, moved to the end of the string. Let’s use *stuff* to represent *extra* without that closeDelim (again, assuming *extra* exists in the first place); then what’s actually there is prefix-*match*-closeDelim-*stuff*-suffix. But *extra* is turned into *stuff*-closeDelim, so, assuming closeDelim and suffix are the same, this is equivalent to prefix-*match*-suffix-*extra*.

The NestedRegexResult also includes the indices of the strings in toFind. These indices only include instances in the top level of the code: so, for instance, if you want a comma-separated list, including “,” in toFind gives you the indices of all the top-level commas, but not those in, say, new int[]{1, 2, 3, 4, 5}.

The application of macros to the user’s code is handled by the Macro.applyTo(String code) and CodeIntegrator.toJavaClass(String algorithmName, String userCode) methods. toJavaClass iterates through two lists of macros, calling applyTo on each one and passing it the user’s code. applyTo goes through the code and replaces each match with the result of modify. Each replacement is applied immediately, so the next match will be searched for in the modified code rather than the original code.

The two lists of macros used by toJavaClass are Macro.MACROS and Macro.GENERATED\_MACROS. MACROS includes the hardcoded macros in the Macros class; GENERATED\_MACROS includes macros generated by other code. The difference between these lists is that the macros in GENERATED\_MACROS are removed from the list as they’re applied, to avoid duplicates.

MalformedMacroException indicates that a macro was used incorrectly. This should be handled much the same as a compilation error. At the moment, there is very little support for either, and none within the program proper.

**Error Handling**

This is a section that needs a significant amount of work. If a user’s code contains errors, either compile-time or runtime, this is not made clear within Galant.

* If a MalformedMacroException is thrown, its message is printed in the log, as well as the fact that it’s a MalformedMacroException.
* If the Java compiler encounters errors, the log shows the Java class generated and the compiler’s messages.
* If an exception is encountered while the animation is running, this is treated the same as an exception within Galant’s code; that is, there is no distinction between an error on the part of the algorithm programmer and one on the part of Galant’s developers.

**Functions**

Essentially, a function declaration is translated to an instance of the class Function<P, R>, with the invoke method containing the function’s code. The type parameters of the Function class are the type of the parameter and the type of the return value, respectively. Multiple parameters are simulated with the Pair<E1, E2> class. A Pair is simply an object with references to two other objects. They can be chained together indefinitely to create a sort of linked list, but completely typesafe (and thus with a much more complicated type). Thus, for example, a function that takes a String, an int, and another String would have a parameter type of Pair<String, Pair<Integer, String>>.

To allow recursive functions, a function must be accessible from within its own code block. Since functions are accessed by calling invoke on a variable of type Function, this means that each function’s variable must be visible from within the body of its invoke method. However, since the user’s code is inserted into a method, all the variables are local variables. For them to be accessible within an anonymous class, they must be declared final. Simply saying final Function<…> f = new Function<…>(){…f.invoke(…)…}; is not enough, though. Calling the final variable f within its own initialization makes the compiler complain that it may not have been initialized. And because it’s final, it can’t just be initialized with some default value to start with. Therefore, it’s made a final Pair with the first element being the function. Since it’s final, it’s accessible within the inner class; since it’s an object, the first element can be initalized as null and later changed to point to the function.

**3. Graph Components/API**

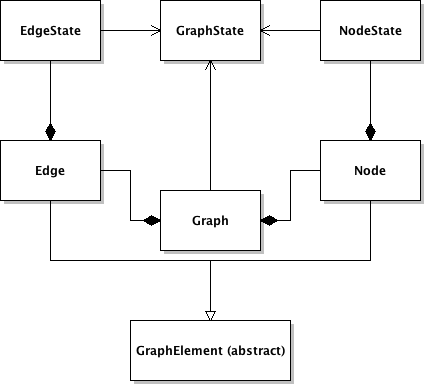


figure 2.1: Graph components

The components shown in figure 2.1 are located in the package edu.ncsu.csc.Galant.algorithm.graph.component

The relationship between Graph components can be seen in figure 2.1. A Graph consists of a GraphState, Nodes (stored in a List), and Edges (stores in a List). An Edge is composed of EdgeStates, stored in a List. A Node is composed of NodeStates, stored in a List.

States are the main idea around which these classes operate. Rather than store a new copy of the entire graph for each change made by an algorithm, a single Graph object is maintained whose components can create and store internally new “states” of themselves. Each of these states, stored in NodeState or EdgeState, maintains a set of properties associated with an integer state. For each change made in a graph component, a new state is made and the global graph state is incremented. Mechanisms also exist in GraphState to lock the state when it is not desired that the graph change state (e.g. editing the graph in Edit mode or doing several operations at once).

Node and Edge both extend the abstract class GraphElement, which is currently only used in the ComponentEditPanel to allow the user to change Node and Edge properties in the visual editor. If any additional properties are added that relevant to both Nodes and Edges, adding the methods to access them to the GraphElement class will allow the use of a single generic ComponentEditPanel. Adding any properties that are unique to one or the other will necessitate separate panels.

When new APIs are added to either Node or Edge, they should follow the established form. In the following, ‘PROPERTY’ is a placeholder for the new property. Most APIs support two getters and a setter.

The no-arg getPROPERTY() should return the value of the property in the latest state of the graph.

The 1-arg getPROPERTY(int state) should return the value of the property in the specified state. Shown below is the format this follows.

NodeState/EdgeState s = getLatestValidState(state);

return s==null ? null : s.getPROPERTY();

This gets the latest valid state of the object at the specified time. If the object was last changed at state 5, and state 7 is requested, state 5 will be the latest valid state.

The setPROPERTY method should follow this form

NodeState/EdgeState s = newState();

s.setPROPERTY(VALUE);

addNodeState/addEdgeState(s);

Step 1 is creating a new state. The newState method clones the latest state and sets its state to the value in GraphState (this will always be the latest state of the graph). Then the desired property needs to be set. the addNodeState and addEdgeState methods handle adding the state and resolve cases where multiple States are added with the same state value.

Currently, all properties of Edges and Nodes are stored in their respective State classes, with the exception of a Node’s position and a Node’s list of Edges. All edges ever associated with a node are stored in a list in the Node object and rely on the properties of the Edge itself to see if it is valid in a particular state.

**4. Graph Dispatch**

edu.ncsu.csc.Galant.GraphDispatch

Galant consists of two interconnected windows: a Visual Graph editor and a Textual editor. The GraphDispatch class is the component that connects these two components. This class is a singleton that is instantiated when the application is opened.

The GraphDispatch class has several functions important to communication between the components.

* Push updates in the visual editor to the textual editor
* Push updates in the textual editor to the visual editor
* Maintain a single active graph and link it to its source textual editor
* Maintain properties that need to be shared between the windows (Graph canvas dimensions, animation mode flag)

In order for updates to be pushed to their respective components, each component needs to register a PropertyChangeListener in the GraphDispatch singleton. Calling the pushToGraphEditor notifies the GraphWindow that it needs to update. Calling pushToTextEditor notifies the text editor associated with the active graph to update itself. If more fine grained pushes are required in the future, new a new push and a unique string identifying the type will need to be added to GraphDispatch. Currently, notifyAddEdge, notifyAddNode, and notifyDeleteNode are commented because there is currently no distinction between these actions.

**5. Visual Graph Editor**

The Visual Graph Editor is defined in

edu.ncsu.csc.Galant.gui.window.GraphWindow

This is also composed of components

edu.ncsu.csc.Galant.gui.window.panels.ColorPanel

edu.ncsu.csc.Galant.gui.window.panels.ComponentEditPanel

edu.ncsu.csc.Galant.gui.window.panels.GraphPanel

The GraphWindow class handles user input to visually edit Graphs, change various modes of Graphs, edit properties of Nodes and Edges, and step through animations. A MouseListener handles what happens when you do various manipulations to the graph, e.g. clicking when the Add Node mode is selected.

GraphWindow contains a menu, an editing toolbar, a GraphPanel used to draw the graph, a ComponentEditPanel used to edit properties of components, and a set of animation controls.

enums at the beginning of the GraphWindow class define different states of toolbar groups and store the location of their icons (used in the various createButton methods to create a JToggleButton with an icon). The several createButton methods are used to register any listeners to handle changes to button toggles, e.g. setting the window mode to Add Node.

If the GraphDispatch triggers the ANIMATION\_MODE property change, this component disables its editing features and enables the animation controls. These work by requesting the GraphPanel

New animation controls can be added in the initAnimationPanel method if future development requires additional control over animations. The existing controls work by incrementing and decrementing the display state stored in the GraphPanel.

GraphPanel overrides the paintComponent method and draws the Graph on the panel’s canvas. The GraphPanel stores a display state, which holds an integer that represents the Graph state to be displayed. Recall that Graphs store themselves in states. GraphPanel iterates through every Node and Edge in the Graph, and if it is in scope at the display state, then it draws the node and its corresponding properties in that particular Graph state. Edges are drawn first to keep them behind nodes.

Drawing text, surrounding textboxes, and directed edges utilize an AffineTransform to get a transformed canvas and draw them logically as if they were horizontal. The getEdgeTransform method accepts two points as parameters and will return a canvas transformed such that a line drawn between the two points is the new x axis.

ComponentEditPanel allows the user to change Node and Edge properties in the visual editor. If any additional properties are added that relevant to both Nodes and Edges, adding the methods to access them to the GraphElement class will allow the use of the single generic ComponentEditPanel. Adding any properties that are unique to one or the other will necessitate separate panels.

**6. Textual Graph and Algorithm Editor**

The edu.ncsu.csc.Galant.gui.editor package contains all the relevant classes for the text editor. It is best to understand the editor from the structural standpoint of how the GUI components are arranged to create the interface.

**GEditorFrame and GEditorMenuBar**

The editor window is provided by the JFrame extension GEditorFrame and strictly occurs in a singleton instance so as to be accessible by all components. Multiple editor windows is not possible and not recommended for future developers. The GEditorFrame is responsible for handling Open, Save, and Saves As actions by the user. These menu items are held in the GEditorMenuBar class.

**GTabbedPane**

The GEditorFrame contains exactly one instance of the JTabbedPane extension GTabbedPane. The GTabbedPane is responsible for storing most information about the logical state of the editor. It tracks which files are currently open for editing, and restores the last edit session upon reopening Galant. The GTabbedPane also mediates the relationship between the user and GEditorPanels; for example, if the user attempts to close Galant while a file is dirty, the GTabbedPane is responsible for detecting the conflict and warning the user. Lastly the GTabbedPane is responsible for managing tab selections and creating a new editor tab when the user clicks on the “Create New” buttons. This is handled via the magic of ChangeListener.

**GEditorPanel, GAlgorithmEditorPanel, and GGraphEditorPanel**

Each JPanel extension GEditorPanel and its extensions GAlgorithmEditorPanel and GGraphEditorPanel represents a separate file edit session. This class provides the *content* of a tab in the GTabbedPane, so if there are three graphs and two algorithms open for editing, then there will three GGraphEditorPanel instances and two GAlgorithmEditorPanel instances. These panels are, of course, responsible for storing the text content of the editor, as well as whether the file is dirty or clean. They are also responsible for the respective nuances of the editor behavior such as syntax highlighting on GraphML keywords or tooltip text on API calls.

**GAlgorithmSyntaxHighlighting and GGraphSyntaxHighlighting**

These classes are *not* GUI components. They are essentially utility classes for

delegating the annoying functions of syntax highlighting. From a logical standpoint they could be contained in GAlgorithmEditorPanel and GGraphEditorPanel, respectively, but that is not done in order to prevent exorbitantly long class files.

**7. GraphML Parser**

Currently, if the parser encounters a malformed file it returns without trying to partially parse the file. Future plans may want the ability to load partially correct GraphML.

The GraphML Parser, edu.ncsu.csc.Galant.graph.parser.GraphMLParser, uses a DocumentBuilder to grab elements from a GraphML file by tag name. It creates a list of all Node elements and all Edge elements, then iterates through each and gets individual properties from each by looking for the nodeValue of a namedItem with the property name.

Any data not specified is input as some default value.

The reverse, going from a Graph object to GraphML, is done by the Graph and its components’ toString methods. This generates a clean GraphML representation of everything within the graph and returns it as a String.

**8. Preferences**

Implementation of preferences is split across two packages and a class: edu.ncsu.csc.Galant.prefs, edu.ncsu.csc.Galant.gui.prefs, and edu.ncsu.csc.Galant.GalantPreferences. prefs contains the non-GUI aspects of the implementation; gui.prefs contains the GUI aspects; and GalantPreferences contains the actual preferences used in the program.

The base of preferences is the Preference<V> class, in prefs. This shouldn’t be confused with java.util.prefs.Preferences, though that class is used as well. A Preference<V> is essentially just a container for a value of type V, which uses Preferences to store that value between sessions. A BackingStoreAccessor is used to

access Preferences in a typesafe manner.

A PreferenceComponent<V, C extends Component> uses a GUI component of type C to let the user set the value for a preference of type V. Each PreferenceComponent is associated with a Preference of the correct type. When the user sets the value on a PreferenceComponent, that value does not immediately go through to the associated Preference; the user must confirm by pressing “Apply” or “Save”.

Each Preference belongs to a PreferenceGroup; PreferenceGroups are arranged in a tree-like hierarchy, and new PreferenceGroups are created by calling addNewChild(String label) on their parents. A PreferenceGroupPanel is associated with a PreferenceGroup, and displays all of the PreferenceComponents for that group.

PreferencesPanel contains the overall Preferences dialog into which the PreferenceGroupPanels are inserted. This includes a tree used to select a PreferenceGroupPanel, as well as buttons to apply changes and close the dialog.

GalantPreferences defines the preferences used in Galant, as well as the preference groups. Each preference and group is a public static final variable, so they can be easily accessed from wherever in the program they’re needed.

The actual definitions are in the static initializer. Adding a preference involves creating the preference, adding it to its preference group (and creating that group if necessary), making stuff call the preference when needed, and creating the associated PreferenceComponent, overriding apply() (but including a call to super()) to apply a changed value to the program if just changing the value by itself isn’t enough.

**9. Automatic Graph Repositioning**

The automatic repositioning code is found in the Graph object, implemented in smartReposition, forceDirected, centerInWindow, nudgeToCenter, scaleToWindow, nudgeToEdge, forceAttractive, forceRepulsive, unitVector, magnitude, totalChange, updateStepLength, and pathExists.

The algorithm is based on the paper published at <http://www.mathematica-journal.com/issue/v10i1/contents/graph_draw/graph_draw_3.html> and its implementation is outlined in javadoc. This algorithm treats all nodes as similarly charged particles that repel each other and all edges as springs (attractive forces) between nodes. The algorithm requires that disconnected subgraphs are considered individually to avoid repelling each other infinitely.

If additional repositioning schemes are desired, for example a tree positioning algorithm, a flag will need to be added to Graph and smartReposition will need to choose based on the flag.

**10. Running an Algorithm**

Currently, the algorithm is run in its entirety before the animation appears. This creates a series of states on the graph; graph states essentially act like frames in the animation. Clicking forward or back through the algorithm is just changing the state of the graph. (For more about graph states, see section 3, Graph Components/API.)

The code of the algorithm is inserted into the run() method of a class that extends Algorithm. This means that "global variables" in Galant are really local variables in the run() method, so once the run() method is finished executing, they're no longer available.

This means that if you want to reference them from within a function’s body, they must be declared final. When you declare a function, you're actually creating an object with a method containing the function's code. This object could potentially last beyond the execution of the run() method and have its method called after the run() method's local variables are no longer available. Therefore, it can only access those local variables if they're declared final, which means that their values won't change, so they can be stored when the object is created and won't get out of sync.