Object Oriented Programming
Final Project Report
Graph Editor

First Author Second Author
Andrés Salinas Lima (S3837416) Russell Sammut-Bonnici (S3839397)

June 19, 2019

1 Problem Description

The task at hand is to create an undirected graph editor with a graphical user interface in Java. It should be designed using Swing and Java IO, with the MVC (Model-View-Controller) architectural pattern kept in mind. The MVC approach is important in order to keep the reasonably complex code maintainable.

Using the keyboard and/or mouse the user should be able to edit graphs as well as save and load them. The input behaviour should be keyboard/mouse signals received from the user, which should in turn activate actions when detected. The output behaviour should emerge from the activated actions and result in observable changes in the graph.

2 Problem Analysis

The graph editor can be thought of in two parts. The first part is the graph itself; what it consists of, how it can be interpreted during saving and loading, and so on. The second part is the editor; how the structural information of the graph is utilized, how the graph is displayed and what actions the user can impose on the graph.

2.1 The Graph

A graph model is described as a composition of vertices and edges. A vertex is represented by a rectangle, with its own name, size and location. An edge is represented by a line that connects one vertex to another. The pair is unordered in this case as the graph is undirected.

2.2 The Editor

The Editor is the graphical interface that will allow the user to interact (visualize and modify) with the graph. Though simplistic of a concept, this is worth mentioning as much of the program's code has to follow this graph-editor separation for clarity.

The editor is represented by a window, composed of a frame and a panel. The structural information of the graph model is called at this level for presentation in the panel. The user can edit, open, save or create the graph model through actions accessed through buttons, menu items and keyboard shortcuts.

3 Program Design

This program was designed with the MVC architectural pattern kept in mind. Tackling from these three aspects makes the problem much more approachable as well as the program in itself more maintainable and secure.

More maintainable because the approach is modular. When something goes wrong in one section, it is easily identifiable from which section the issue is stemming from. More secure because having to call classes from outside their packages motivates encapsulation, where secure getters and setters are the only way to interpret and redefine instantiated objects.

Source code files were structured through packages according to model-view-controller as seen in Figure 1 below. A brief description of each part will be given in this section.

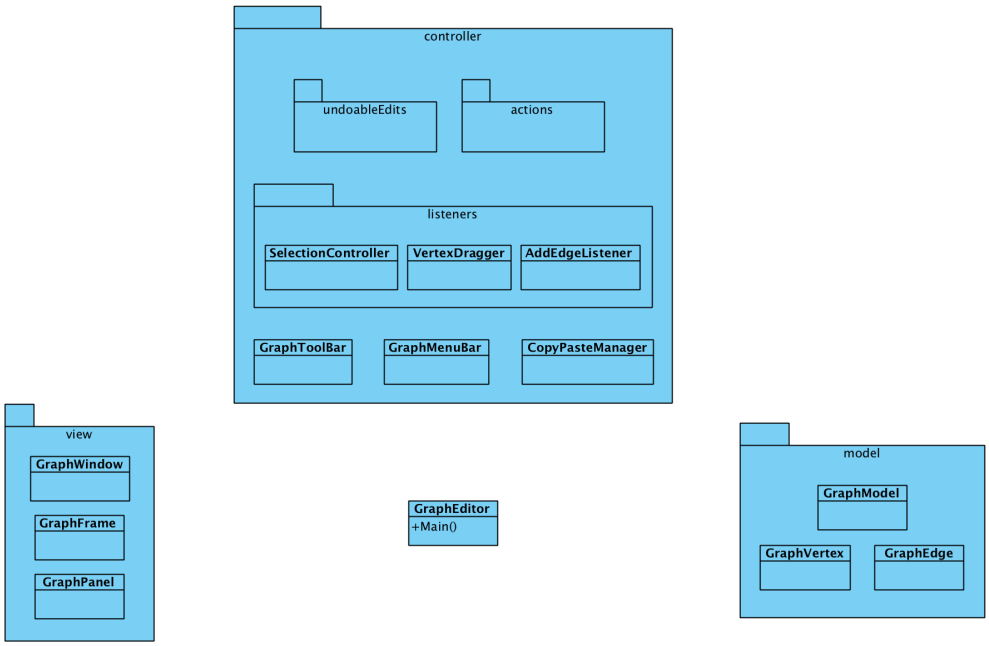

Figure 1: Package diagram

3.1 Model

The Model is the graph structure described in section 2.1, which is implemented by the *GraphModel*, *GraphVertex* and *GraphEdge* classes. The graph components are essentially implemented by an *ArrayList<GraphVertex>* for vertices and an *ArrayList<GraphEdge>* for edges, with selected vertices and edges kept track of by two additional object ArrayLists.

GraphModel could be considered the most important class of this program, as it's on charge of modifying the graph, keeping track of the changes made to its vertices and notifying the view of all these changes. This was easy to make thanks to the observer design pattern and the already implemented *Observable* class and *Observer* interface of Java. In addition to that, the *GraphModel* is also the place where the *UndoManager* and the *CopyPasteManager* reside.

A class diagram of the Model is shown in Figure 2.

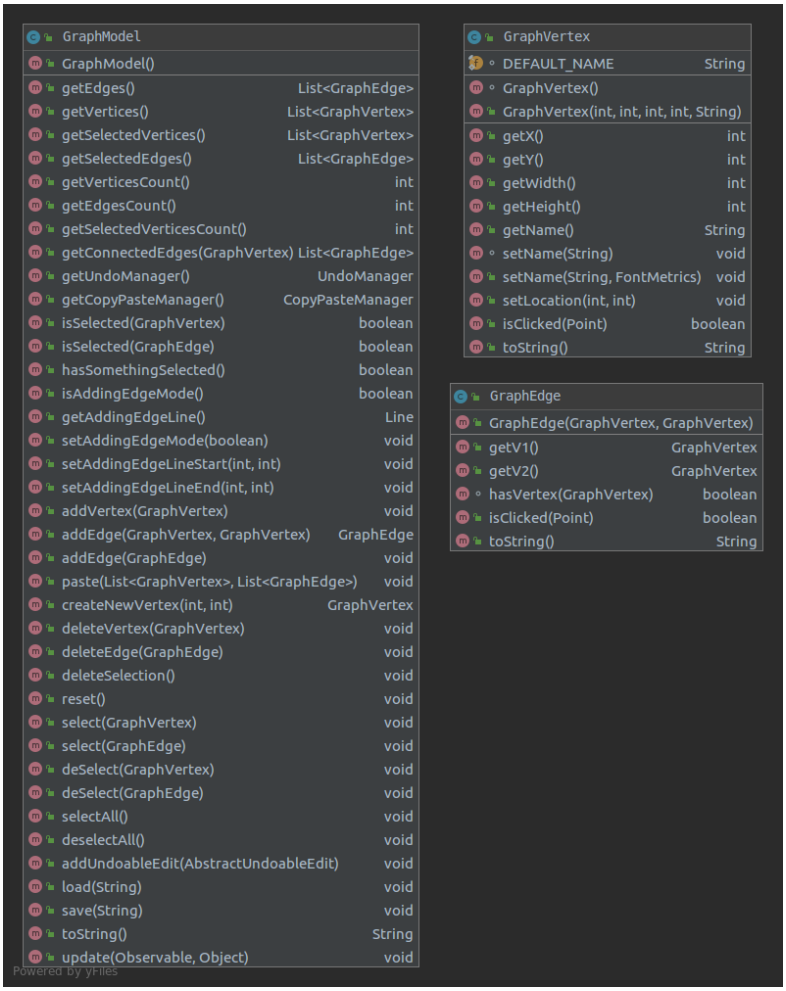

Figure 2: Graph Model

3.2 View

The View is the window composed of the frame and panel as well as the presentation of the model within the panel: *GraphWindow* contains the *GraphFrame* which contains the *GraphPanel* inside of a *ScrollPane* which allows the user to scroll the panel. All the painting work is done in the *GraphPanel* class for graph model display (Figure 3). When the graph model is modified, via the observer design pattern, the *GraphPanel* gets notified and repaints the panel to display the updated model.

3.3 Controller

The Controller generally consists of the Menu Bar (*GraphMenuBar* class), the Tool Bar (*GraphToolBar* class), the actions for both the Menu and Tool Bars (*actions* package), the mouse and keyboard listeners ((*listeners* package with the *AddEdgeListener*, *SelectionController* and *VertexDragger* classes) and all the undoable edits for the model (*undoableEdits* package) that will be managed by the UndoManager.

Since the functionality of the class *CopyPasteManager* is an extension of the program it is later

explained in section 4. On the other hand, the *GraphToolBar* isn't an extension but it isn't necessarily essential either. Its purpose is user-friendliness. It basically provides the user with easy access to the frequently used actions; new graph, open, save, undo, redo, add vertex, add edge, rename vertex, select all and delete selection. The same actions, alongside more complex actions, can be easily accessed via menu items in the *GraphMenuBar*. It's organised as follows:

3.3.1 The Graph Menu Bar

Note that each of the menu items (apart from Show Toolbar and Default Size) have keyboard shortcuts assigned to them, which are further discussed in section 4.1.

1. **File**
   - 1.1. **New Graph**: Simply resets the graph by deleting all of its vertices and edges
   - 1.2. **Open**: Loads a graph from a .txt file.
   - 1.3. **Save**: Saves the current graph as a .txt file.
   - 1.4. **Quit**: Exits the program.
2. **Edit**
   - 2.1. **Undo**: Undoes the user's most recent edit.
   - 2.2. **Redo**: Reverts the previous undo action.
   - 2.3. **Copy**: Copies the selected vertices (and edges between them).
   - 2.4. **Cut**: Cuts the selected vertices (and edges between them).
   - 2.5. **Paste**: Pastes the copied/cut vertices and edges.
   - 2.6. **Add Vertex**: Adds a new vertex to the graph with the default name in the default position but when added, its name and position will be adjusted if they conflict with already existing vertices. New vertices will always appear in the top left corner.
   - 2.7. **Add Edge**: Creates a new edge from the selected vertex. A new line is drawn from the selected vertex to the user's cursor (Figure 5). When a user selects another vertex an edge is created between those two vertices, otherwise (if something else is selected in the panel) the line is removed and no edge is added. An edge cannot be created between two vertices if there already exists (Figure 7). An edge can't be made from a vertex to itself (Figure 8).
   - 2.8. **Rename Vertex**: Renames the selected vertex (Figure 10). Multiple vertices cannot be renamed (Figure 9).
   - 2.9. **Select All**: Selects all the vertices and edges.
   - 2.10. **Delete Section**: Removes any selected vertices and edges from the graph.
3. **Window**
   - 3.1. **Show Toolbar** Initially set to visible, it toggles the visibility of the GraphToolBar (Figure 6).
   - 3.2. **New Window**: Creates a new window with a new graph.
   - 3.3. **Duplicate Window**: Creates a new window but with the current graph. All the windows containing the same graph will update in real time.
   - 3.4. **Default Size**: It simply resets the window back to its initial default size.

4 Extensions of the Program

4.1 Keyboard Shortcuts

The keyboard shortcuts were mapped to their respective menuItem in the GraphMenuBar. This was done by calling *setAccelerator()* from the extended class JMenuBar. Inside the parameters we fed the keystroke of the keyboard shortcut to be assigned. The keystroke was obtained using *KeyStroke.getKeyStroke()*.

Menu name	Action name	Keyboard shortcut
File	New Graph	*Ctrl-N*
	Open	*Ctrl-O*
	Save	*Ctrl-S*
	Quit	*Ctrl-Q*
Edit	Undo	*Ctrl-Z*
	Redo	*Ctrl-Shift-Z*
	Copy	*Ctrl-C*
	Cut	*Ctrl-X*
	Redo	*Ctrl-V*
	Add Vertex	*Alt-V*
	Add Edge	*Alt-E*
	Rename Vertex	*Ctrl-R*
	Select All	*Ctrl-A*
	Delete Selection	*Ctrl-Backspace*
Window	New Window	*Ctrl-Shift-N*
	Duplicate Window	*Ctrl-Shift-D*

Table 1: Keyboard shortcuts to menuItems

4.2 Edge Selection

In order to select an edge the user simply has to click on it. Of course, clicking on a thin line could be tricky, so there is a margin of 5 pixels around it.

This is done in *GraphEdge.isClicked(Point)* by creating an imaginary square around the click coordinates. Then we just intersect this rectangle with the edge lines.

4.3 Multiple Vertex and Edge selection

As seen in 4, it was made so that on holding down Ctrl, the user could select multiple vertices and edges. This was done using a boolean variable *ctrlIsDown*. A KeyListener was implemented into SelectionController so that when the Ctrl key is pressed *ctrlIsDown* is set to true. On press of a vertex or an edge, the program then performs a different procedure, based on a condition that the boolean variable is set to true. Instead of deselecting all previous vertices and selecting the current one, it simply adds the current one to the previously selected and when a selected vertex or edge is pressed it is deselected. This allows a toggle mechanic for selection, and in result allows for selection of multiple vertices and edges.

It is worth mentioning that we initially placed the *ctrlIsDown* variable in SelectionController. We changed its position because we experienced a bug that when a new window was created via a Ctrl key-

board shortcut, the program wouldn't update *ctrlIsDown* and leave it outdated on true. The new window would shift the focus before the Ctrl button could be released. This made it so that the program could select multiple even though Ctrl wasn't held down. Our solution was to place *ctrlIsDown* in GraphFrame instead, and update it via getters and setters. This allowed us to update it appropriately in the scenarios that new windows were created such as in SaveAction, OpenAction, NewWindowAction and DuplicateWindowAction. It is also worth noting that if the button ESC is pressed while selecting components, the selection will be cancelled (as well as the "adding edge" action). Of course, once this multiple selection is done, the user is able to drag, copy, cut or delete all the elements at once.

4.4 Automatic Vertex Width Adjustment

When renaming vertices, if the new name is too long, the width of the vertex will be increased so that the name fits inside. There is however a limit of 50 characters for the name to avoid excessively large vertices.

4.5 Vertex Restriction Border

If the user tries to drag a vertex (or group of vertices) out of the panel (which will cause them to disappear), the drag will be cancelled and the vertices go back to their original location.

4.6 Copy, Cut and Paste

It is possible to copy, cut and paste vertices and edges. It's worth mentioning that in order to avoid problems, the only way to copy/cut edges is to copy/cut the vertices that these edges connect. The pasted vertices and edges are, in reality, completely new vertices and edges (with same name and relative positions though), which allows the user to paste multiple times without problems. Also, the position of the pasted vertices is shifted every time so they don't overlap. And, of course, it's possible to undo/redo the cut or paste of vertices.

4.7 Scrollable Frame

The *GraphPanel* is inside of a *ScrollPane*, which allows the user to scroll the panel in order to have a much bigger working area.

4.8 New Window

If the user clicks on the "New Window" menu item (or uses the *Shif+Ctrl+N* shortcut), a new window is open with an empty graph. This allows the user to work on different graphs at the same time and even open an existing graph from a file; there are no limits.

5 Future Extensions

The following is a list of possible extensions to further expand the functionality of our graph editor:

- Ability to drag vertices with arrow keys.
- Add a *presentationName* to the undoableEdits and show it in the menu so the user knows which action they are going to undo.
- Help menu with user instructions.

- Pop up window that warns the user when they quit the program or open a new graph without saving the changes made to the current graph.
- Another file format, like *GraphViz*.
- Select multiple Vertices and Edges with a Rectangle Select Tool.
- Directed graphs.

6 Conclusion

As required, the editor program is able to create, modify save and load undirected graphs with the MVC approach kept in mind. The program was coded bottom-up. We first started from the model layer then proceeded to working on the view and controller layer.

By the end, one of the major difficulties we experienced was catering for conflicting controls. At a point, the keyboard shortcuts conflicted with the multiple vertex selection feature. The select multiple vertices mode activated even when it shouldn't have. At another point, the listeners for dragging and selecting disagreed. One listener wanted to drag a vertex but couldn't as the other listener would only select it on click rather than on press. To avoid features conflicting with one another, effort went into simplifying and generalizing the features as much as possibly deemed.

In conclusion, object oriented concepts such as encapsulation, inheritance, polymorphism and the model-view-controller approach were put into practice. To us, the project not only provided further knowledge on how to implement such concepts but it also illustrated their overall importance in the object oriented programming paradigm.

7 Screenshots

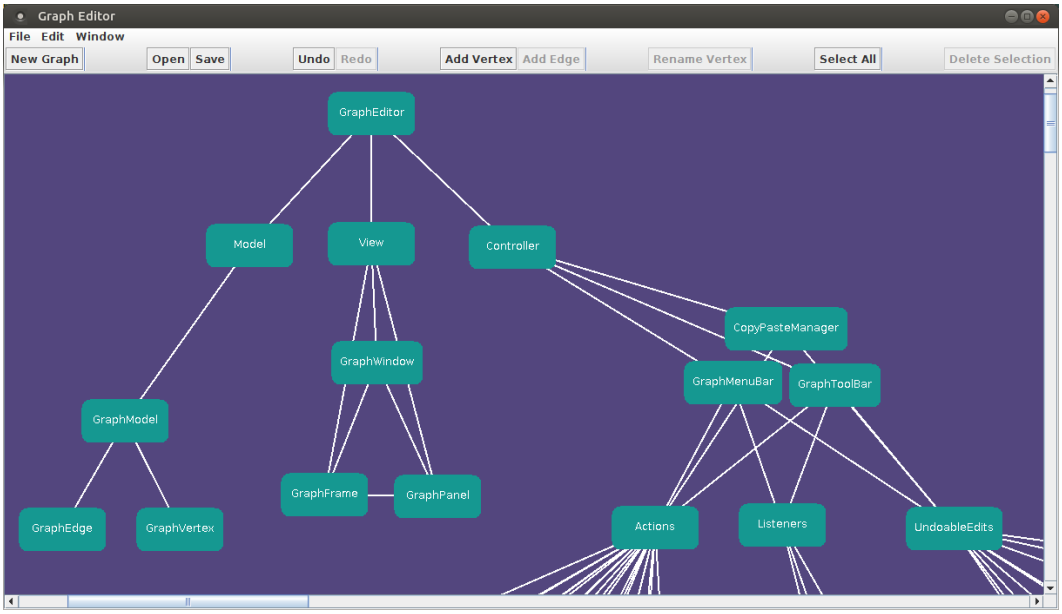

Figure 3: Graph model display in panel

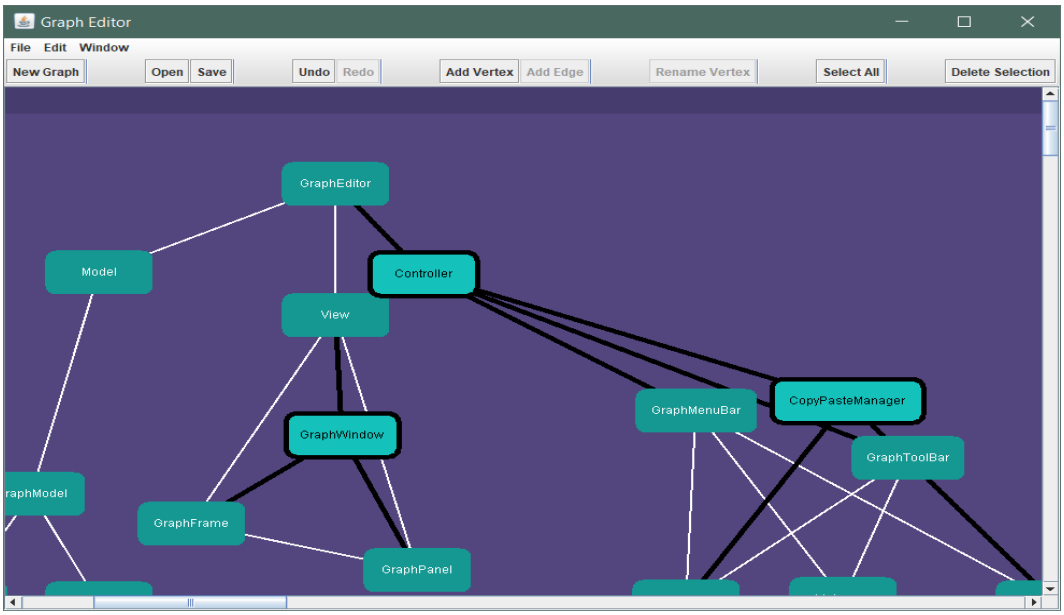

Figure 4: Selecting multiple vertices and edges

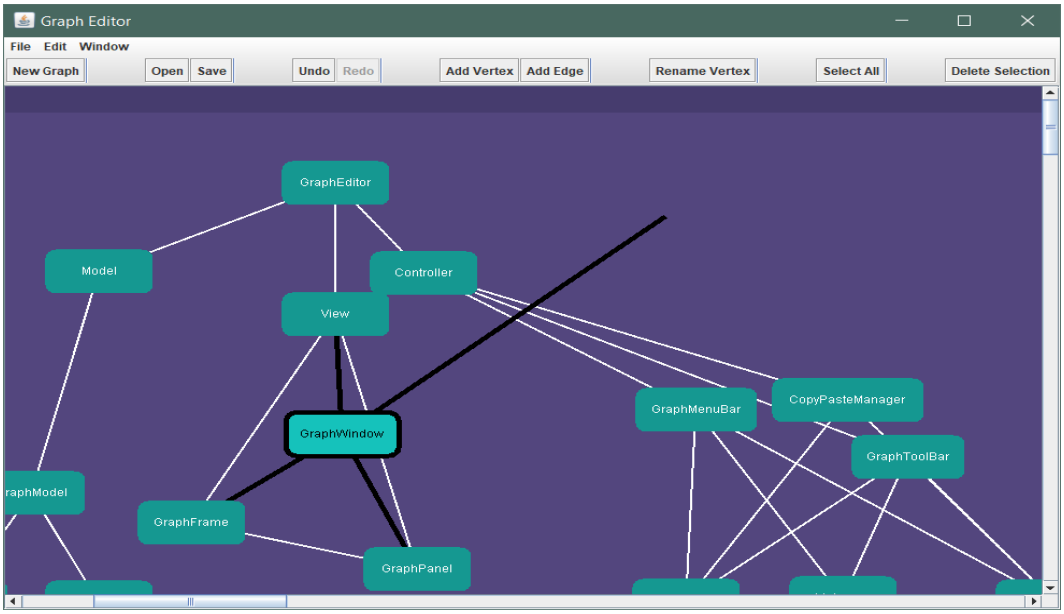

Figure 5: Adding an edge via the mouse

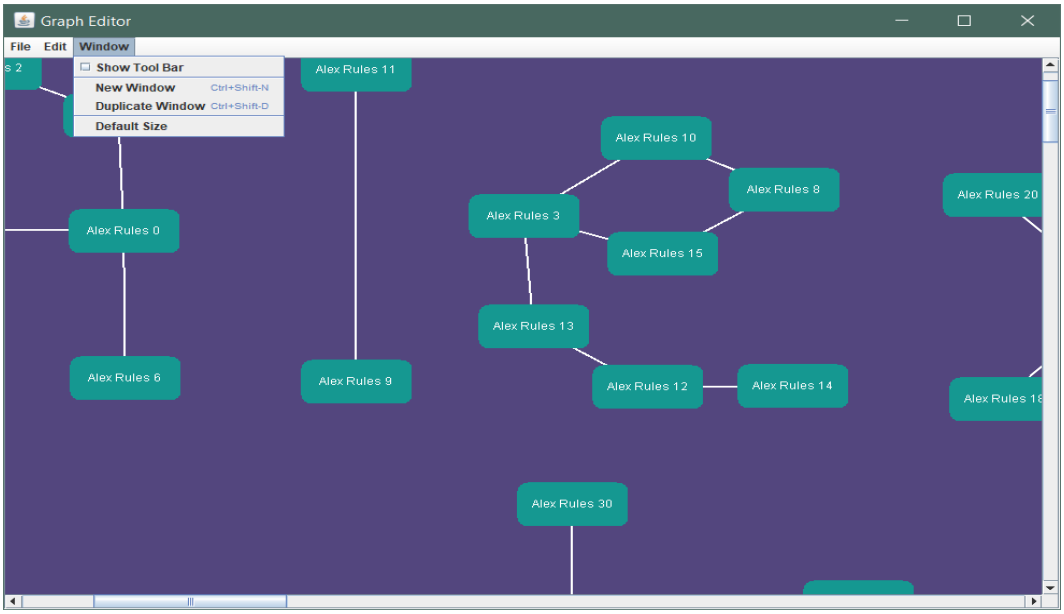

Figure 6: Toggled off tool bar via the menu bar

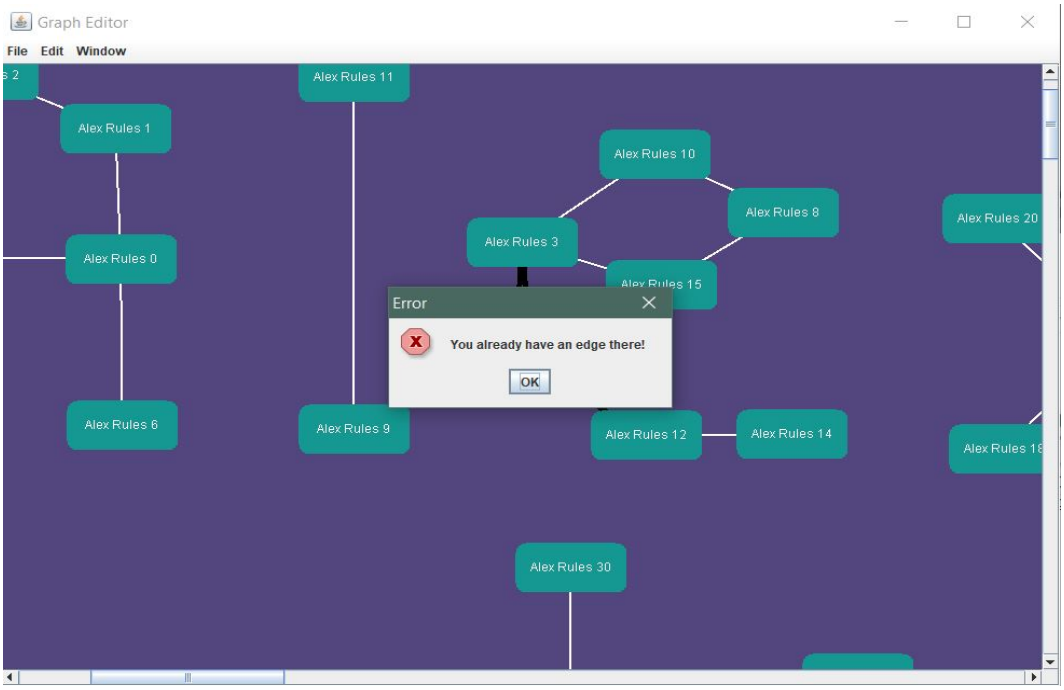

Figure 7: Pop up error for adding a clone edge

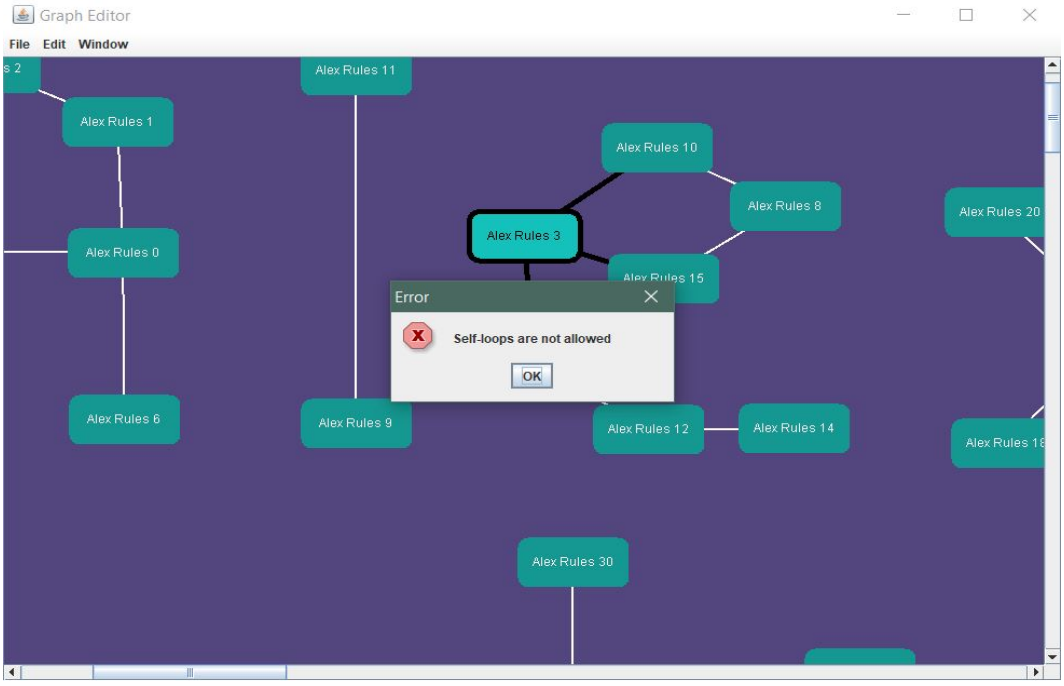

Figure 8: Pop up error for connecting a vertex to itself

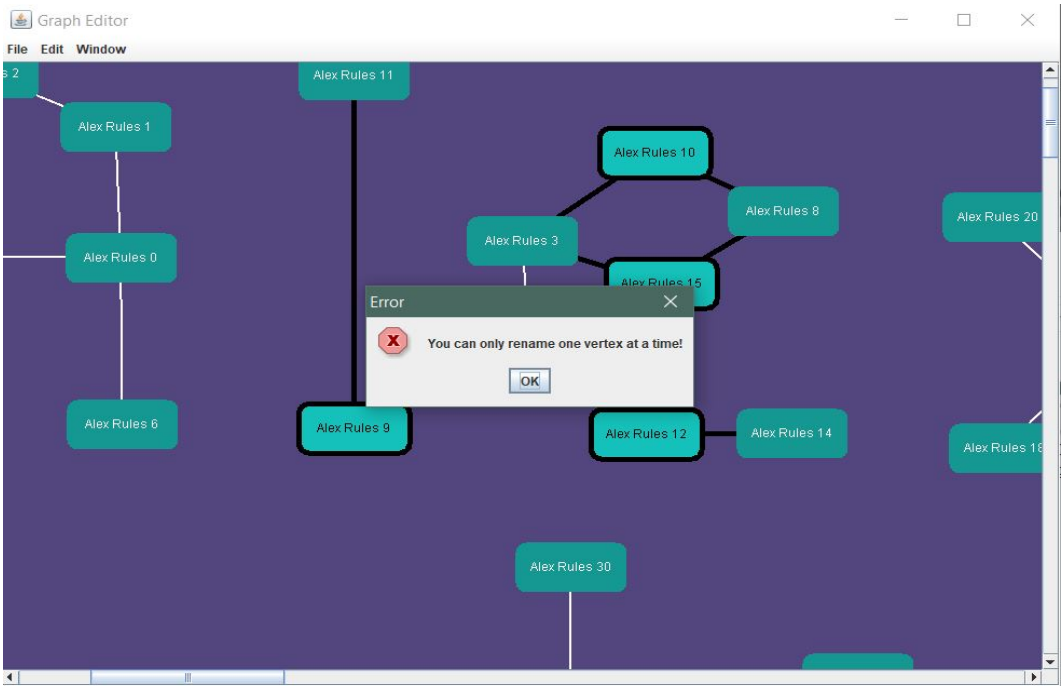

Figure 9: Pop up error for renaming multiple vertices

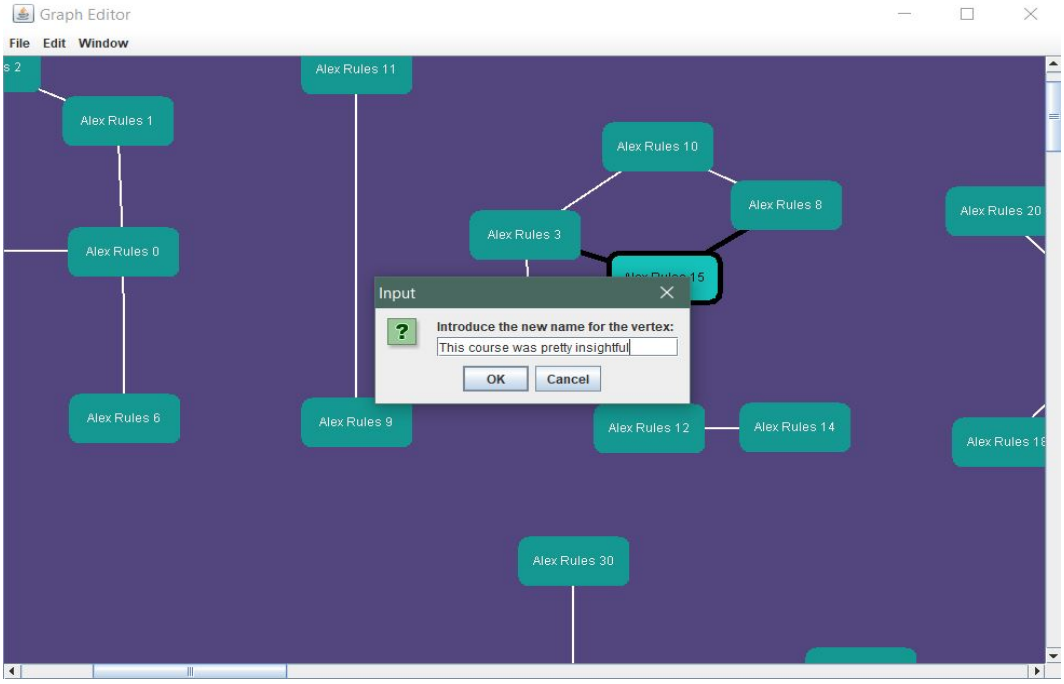

Figure 10: Renaming a vertex