Object Oriented Programming
Programming report
Graph Editor

Robert Patrona Yona Moreda
s3322211 s3324338

June 17, 2018

1 Problem description

In this assignment, we need to create a Graph Editor that has a simple and easy to use User Interface. The Editor contains a simple, undirected and unweighted graph, where vertices are represented by rectangles, and edges are represented by lines that connect two vertices. The graph is simple, so there are no parallel edges, no self-loops for vertices, and all the edges do not have a direction and do not have a weight. The Graph Editor needs to be able to execute certain commands. These commands are creating a new vertex, deleting a vertex, creating a new edge, deleting an edge, saving and loading the file, renaming, resizing, copying and pasting a vertex, and the ability to undo and redo any commands.

2 Problem analysis

In trying to solve the problem, we used abstraction. This means that we first organized the code according to the MVC pattern. The Model contains the data of the program, such as how many edges and vertices we currently have on the graph, and what their properties are, for example name and size for the vertex and the two connecting vertices for the edge. Moreover, it contains getters, in order to retrieve the information from the model, and setters, in order to modify it.

The View contains all the visual elements of the program, which include a panel, where the graph is displayed, a Status Bar, which shows the currently selected vertex and the copied vertex, a few menus, with multiple menu items and several buttons. The Controller is the bridge between the Model and the View. It manipulates the Model accordingly, based on input received from the View. Since it cannot directly modify the data of the Model, it uses functions to respond to the change in input and modify the model.

3 Program design

In order to achieve MVC, the class GraphFrame was the main View class, GraphController is the main Controller class and GraphModel represents the model of the program. The GraphFrame class implements several listeners. This means that the view listens for input from the user. The Action Listener activates when one of the menu subitems is clicked, such as ”Save”. The Menu Listener expands a certain menu, such as ”File”, in order to display its corresponding menu items. The mouse listeners are used to detect mouse activity, which is used when we add a new edge, and when we select vertices. The Listeners enable the View to send information to the Controller, which eventually modifies the Model. For example, when clicking the menu item ”Resize” in the UI, the View listens and detects this action and then calls the function ResizeVertex in the Controller.

Further in the View, the GraphPanel class uses the data from the Model to repaint the frame after every single move done by the user. StatusBar manages the bar at the bottom of the frame which displays the selected and copied vertex. When constructing the view, we made use of Inheritance. GraphFrame extends JFrame, GraphPanel extends JPanel, and StatusBar extends JLabel. Using Inheritance, we were able to make use of the super constructor and the features and properties of the parent classes.

Moreover, to facilitate communication in the JVM, we made use of the Observer Pattern. GraphModel implements Observable, while GraphFrame and StatusBar implement Observer. This means that whenever the Model changes, it notifies the observers, which is the View, which then change according to the information observed from the Model.

In the Model, in addition to the data about the current graph, there are also Edit functions that correspond to the actions that can be Undoable and Redoable. The Model contains an UndoManager, which contains an ordered list of edits which keeps track of past actions. In the Edit classes, we make all actions Undoable. Thus, whenever we call an action, it gets saved in the UndoManager. We then override the ”undo” and ”redo” methods in these classes in order to define their behaviour when Undo or Redo is called.

Furthermore, in order to prevent adding unnecessary actions to the UndoManager, we used Overloading. For example, in GraphModel, there are two ”removeVertex” functions. The first has one paramenter, while the second has two. The second parameter is the boolean ”addEdit”.

If the first ”removeVertex” function is called, the boolean defaults to true, while the second function enable you to specify that the boolean is either true or false. If the boolean ”addEdit” is false, then that action is not added to the UndoManager. For instance, in the RemoveVEdit class in the Model, in the ”redo” method, the function ”removeVertex” is called with the false boolean, which means that when we redo ”Remove Vertex”, we do not want that redo action to be added to the UndoManager, since that would mean that we would do more Undo actions than we have to.

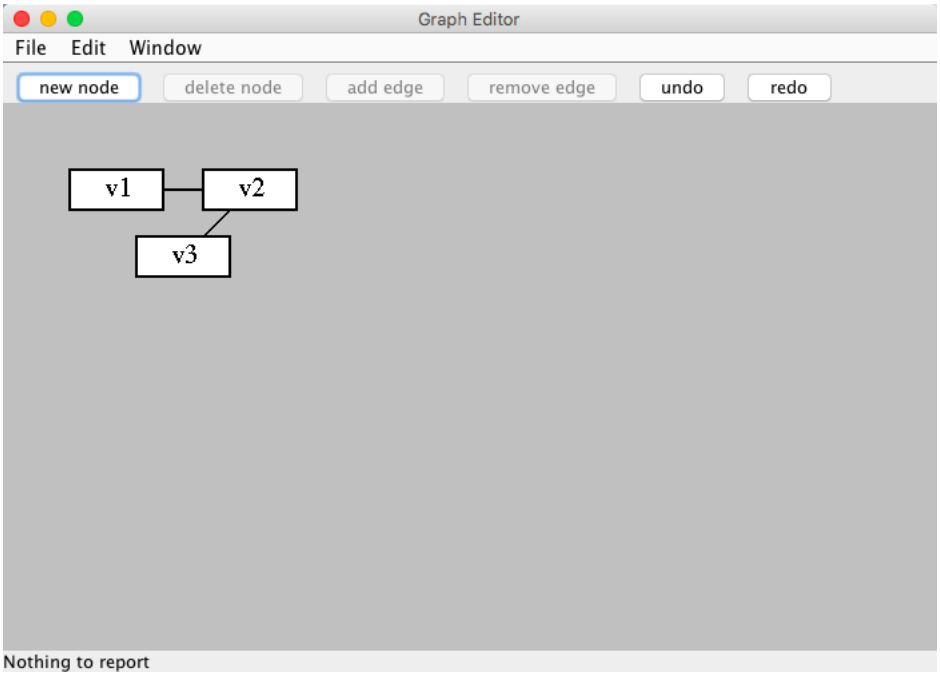
When implementing the Buttons, we thought about putting all buttons in one class and using inner classes. Howeer, we decided to make separate classes. The reason for that is because different buttons get enabled and disabled based on different actions, so it would have been very complicated and messy. Moreover, having individual classes for the buttons means that the code is more organized and easier to understand.

4 Evaluation of the program

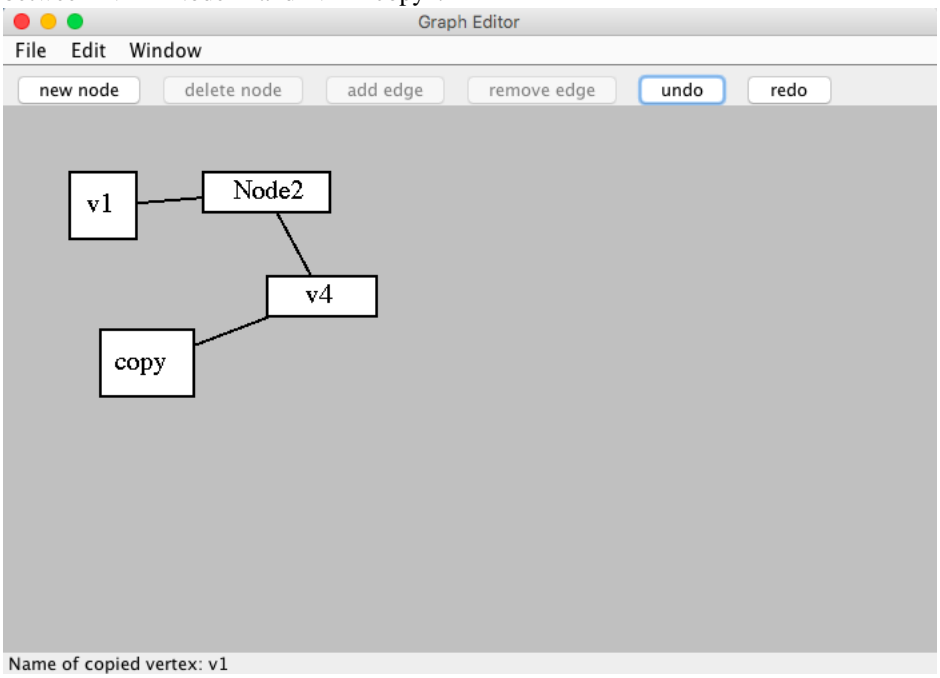
We noticed that the program takes a bit of time to start, but after it has started, it is very fast and responsive. The end result is mostly as we expected in the beginning. The main two aspects that are different from our expectations is the way we add and remove an edge. First, we initially designed that when we remove en edge, we would be able to select and delete it. However, we realized that since the edge is represented by a line, selecting it would be difficult, so we came up with a more convenient approach. When removing an edge, we select a vertex, and then we enter the name of a connecting vertex in a popup window, in order to delete the edge connecting them.

Second, we thought about adding an edge by inputting 2 unconnected vertices, then drawing a line between them. We then changed our implementation since in the reader, it is required for us to start in a vertex, and to have an edge that traces the mouse until we click on a second vertex. Thus we created a temporary line that follows the cursor, and we add an edge only when a new vertex is selected. Clicking outside all vertices cancels the action and deletes the temporary line.

In order to show the behaviour of the program, we will show the effect of appliying a few actions. The following shows the deafult graph:



We applied the following actions: Rename ”v2” to ”Node2”, Resize v1 to 50x50, Copying and Pasting ’v1’, Renaming the second ”v1” to ”copy”, Adding a new Vertex ”v4”, Removing ”v3”, and Adding Edges between ”v4”-”Node2” and ”v4”-”copy”.



5 Conclusions

In conclusion, the program solves the problem. It contains all the features needed for a simple visual Graph Editor. In this assignment, the easiest part was creating the model, and the view elements, such as the buttons, the menus and the menu items.

One of the problems that we had was related to our definition of the edge. In the GraphEdge class, we use integers to represent what vertices are connected by the edge. These integers represent indices in the Array List of vertices in the model. For instance, if GraphEdge connects vertices 1 and 2, that means that it connects the second and third vertices in the Array List in the Model.

When deleting a vertex, and undoing to add it back in the graph, it is removed from its old position in the Array List and added at the end. Since its index changes, it will have other edges connected to it, rather than the original one. To solve that, we expressed the edges in terms of the GraphVertex object, and not in terms of integers, in the entire program, except in the definition of the edge. Since we use object reference instead of integers, the edges kept the same vertices after deleting vertices and undoing.

Another difficulty that we had was that after we clicked the button ”Add Edge”, and moved the cursor outside the panel, the temporary line tracing the cursor would disappear and we would get an error message. We then included a try-catch clause to look for null pointer exceptions.

An additional issue was overlapping vertices. Whenever we had 2 vertices overlapping, and we clicked the one at the top, the bottom one would get selected. To fix this, we had to make sure that even though we paint the vertices from the beginning to the end of the Array List of vertices, when we select a vertex, we go from the end of the vertices list to the beginning.

The most difficult part of our assignment was that we had was that we had some Controller functions in the View. The GraphFrame class contained some Controller functions, such as ”resizeVertex”, which meant that the View was modifying the Model directly. We then moved the functions to the GraphController class, such that the View can only access the graph using getters, while modifying the graph requires to call the necessary function from the Controller. During this process, we created methods in the controller that call their corresponding methods in the Model which call one of the Edit classes, which in turn does the required action.

We better learned about the importance of JVM, we better understood how the observer pattern works and especially how to implement Swing.

6 Appendix: program text

The program code is uploaded on GitHub. It is too long to be included here.