Object-Oriented Programming
Programming Report
Graph Editor

Diego Renzo Sariol S3593649
Felix Zailskas S3918270

June 22, 2020

1 Introduction

For the third and final assignment of this course, we were tasked with creating a fully functioning graph editor that adheres to the strict MVC pattern. The graph editor had a set of requirements that we had to design and program ourselves, such as: the ability to create custom nodes, edit nodes, connect nodes via edges, and allowing the graph components to be fully draggable and editable. From a simple RPG text based game to a fully functioning Magic the Gathering and Heartstone inspired card game, we have learned how to incorporate object oriented programming functionality with friendly user interfaces through the help of Java Swing. This allows us to provide the user three things: a controller, a view, and a model. By following the MVC pattern, the user can navigate our graph editor through the controller, which updates the model accordingly, thus notifying its observers and presenting the user with an updated view.

Our implementation of a graph editor allows the user to construct simple node graphs with custom names, size, and coordinates. Rather than include a series of buttons, we simply have four JMenus that provide the user with drop down options for file, node, edge, and edit options. Each JMenuItem displays their corresponding keyboard combination for faster and easy graph constructing/editing. Creating nodes is as easy as selecting the drop down menu or simply **Ctrl + D**. The nodes are fully draggable and creating adjacent edges is as simple as selecting a node and moving the cursor to a node of your choice. As the cursor glides over your screen, an edge will be formed from the selected node to your current cursor's location on the screen. Want to recreate a certain node with all of its node properties, well that is as simple as using the copy paste keyboard shortcuts everyone is accustomed to, or simply using the drop down menu options.

2 Program Design

Model:
The Model contains the actual graph model. It stores all information on the current state of the model. This is done via three main classes *Node*, *Edge* and *GraphModel*. Both the node and the edge class extend the abstract class *GraphCoponent* so that both nodes and edges can be stored as selected in one field of the graph model.

- Node - A node is an entity in the graph, it can be connected to other nodes using edges. Each node in the graph has a name, a size and two dimensional coordinates. They also keep track of the edges connecting them to other nodes in the graph. A node can be copied, it can be checked if two nodes are equal, it can be checked if two nodes share an edge and they can be moved.
- Edge - An edge is used to connect two nodes to each other. Each Edge keeps track of the two nodes it is connecting. It can be checked if a node is connected to another node via this edge.
- Graph Model - The graph model is the actual graph we are editing with this program. It uses array lists to keep track of all nodes and all edges that currently populate the graph. Edges and nodes can be added or deleted from this graph model. This is done by running over the respective array list and then removing the first item that is equal to the node/edge that is to be deleted. Since our program implements undo-able edits, the graph model also contains an undo manager. Additionally, The model keeps track of a node that can be copied and pasted, as well as a graph component which is selected at a given time.

Controller:
The controller is used to change values in the graph model. Every edit that is done to the graph model is controlled via the controller package. It is made up by a set of action, menu and clicker classes which handle the user input.

- Actions - All actions that edit the graph model extend the class *AbstractUndoableAction*, so that they can be added to the graph model's undo manager and can be undone and redone. These actions handle the events of adding nodes/edges, removing nodes/edges, editing nodes, copying/pasting nodes and resizing nodes. There is another set of actions which are not undo-able but are also handled by the controller package. These actions include saving/loading a GraphModel, creating a new graph model and undoing/redoing an edit to the graph model.

- Menus - The menus in this program are shown at the top of the screen. They provide the user with buttons to execute all of the above mentioned actions to edit the graph model. Every action shown in the menus can also be used with a shown keyboard shortcut. The menu package also includes the pop-up menus that appear when creating a node, editing a node or when loading a graph model which does not fit the current size of the window.

- Clicker - The clicker classes are responsible for handling any mouse input from the user. This includes selection of a node or an edge, resizing of a node, editing a node and resizing the screen. When a node or an edge is clicked it will be selected and appropriate actions become available to the user. This means when a node is selected it can be deleted, edited, resized or copied. In the case that an edge is selected it can be deleted. Double clicking a node automatically enters the editing menu for that node. When resizing a node the user can easily drag the node to the preferred size. Moving a node can be done by dragging it to the desired location, note however that nodes can never leave the window they are displayed in. For easier edge creation it is possible to select a node and then select another node to create an edge between them. Adjusting the window size will result in nodes that would exit the window to be pushed inwards by the new dimensions of the window.

View:
The view handles the visual representation of the graph model. It is notified when a change in the model occurs, so it can always display the most current state of the model. It consists of a frame and a panel.

- Frame - The frame is the main container for the panel drawing the graph model. It contains the menus of the controller and has a minimum size of 900x600 pixels.

- Panel - The panel keeps track of the graph model it is displaying. It continuously draws all nodes and edges between the nodes on the screen. It uses the coordinates and sizes of the nodes to do this. Nodes are shown as black rectangles with their name written in them. Edges are represented by black lines between the nodes. If a certain graph component is selected by the user, it will appear as green on the screen. Nodes which are currently being resized have a red border drawn around them. When a node is selected a theoretical edge is drawn between that node and the cursor to see where the edge would be if it were being created.

3 Evaluation of the program

The final program is more advanced than what we were initially asked for, as we included a range of additional features to enhance the user's experience. Our GUI is not as advanced as our previous project; however, we wanted to include all of the requirements and make the user's experience as enjoyable as possible. A simple GUI is far better than a complex GUI, this is why we kept our node and edge interaction as simple as possible, yet we still offer the user a multitude of ways to interact with the graph and allow the keyboard and mouse to be used for each action rather than manually selecting drop down menus and repeating this tiresome process over and over. The use of the keyboard and mouse controllers allows the user to easily create and manipulate their graph models.

We tested our graph editor by purposefully performing illegal actions or actions that we knew would immediately throw errors. This type of bug testing immediately presented several undo and redo errors, especially when it came to resizing nodes. Although these were very easy fixes, they took a long time to fully locate the code that would cause these errors. Eventually we created new types of edit node action constructors for two types of cases when editing a node. By finding these bugs, we actually got new ideas for editing nodes, such as using a keyboard combination and even double clicking the mouse to enter an edit node configuration menu. Despite the undo redo bugs, we had no other major bugs in our code, as this project was much simpler compared to our previous project, where we created our own implementation of Magic the Gathering and HeartStone. That project quickly presented itself as very buggy as we intertwined the components of MVC. This is why we knew going into this project, how to set up MVC properly and make the graph editor as smooth as our GUI editing skills could go.

4 Extension of the Program

We decided to include the following extensions in our implementation of a graph editor as we found these extremely useful when constructing graph models and the resemblance to a real graph editor allows users to quickly grow accustomed to our version. The following four extensions are complete game changers as they allow the user to quickly edit and change their entire graph model with the aid of the keyboard rather than manually selecting a new drop down option to make an edit.

- Copy and Pasting - The ability to Copy and Paste changed the computer forever as duplicating files or copying text became as incredibly easy as two simple shortcuts. Our graph editor allows the user to copy and paste nodes while maintaining their original node properties, such as name and size. The copied node is instantly placed at the location of the user's cursor which allows for easier placement and less confusion had a node been pasted in its original spot defined by its x and y coordinates.
- Resizing Nodes - Each node possesses the ability to allow the user to edit their name, size, and coordinates. If the user simply wants to edit only the size of the node, they can select the drop down option for editing a node or simply press **Ctrl + E** to quickly edit the node. They are presented with the default node editor fields, but they have the choice of filling in any field and the node will retain the original properties for the fields left blank. This easy access editing menu allows for further changes to be made to each node via the keyboard and mouse. Nodes can also be resized via resize node option in drop down menu or **Ctrl + R** which allows the user to manually resize the node via the mouse.
- Easy Edge Deletion - When our graph editor implementation was in its early stages, the process to delete an edge was as complicated as selecting the two adjoining nodes and then selecting the drop down menu for deleting an edge. By calculating where the user's mouse click has occurred, the mouse event can determine if its cursor's coordinates coincide with an edge. The selected edge will become highlighted in green, indicating to the user that the edge has been selected. To delete the edge, the user can use the drown down menu or simply press **Backspace** on their keyboard (same goes for deleting a node).
- Keyboard Combinations - In this modern day and age, all computer programs have at least a few keyboard shortcuts as this allows a user to easily navigate and manipulate said program. Our graph editor can be seen as a simple program; however, navigating through each drop down menu to create new nodes and edit the current model can easily be sped up with the use of keyboard shortcuts. Each drop down menu option displays their own combination of keyboard keys in order to execute said function. This allows for speedier graph model creation and editing.

5 Process Evaluation

When we started this project we first focused on the model. After creating nodes edges and a graph model containing them, we made a simple console view and console controller to test if the model is working correctly. These classes can still be seen in the final project since they were quite helpful in detecting bugs and implementing the view and controller used in the final result. After confirming that the model was working properly we started making a view using a *JFrame* and *JPanel*. Since this only needed to loop over each edge and node in the graph model and draw them at the specified location we had no problems implementing it. This was also the point at which we introduced selection of nodes and edges and draw the selected component in a specific color.

This lead us to the controller package which was the most extensive part of this project. We started by assigning *ActionListeners* to the *JMenuItems* so that we can replace the manual input over the console. This did not pose a challenge since we only had to move functionality from the console controller to the correct *JMenuItem*. After all *JMenuItems* were working properly we separated their functionality to different classes which extend the *AbstractUndoableEdit* class. This allowed us to introduce the *UndoManger* to the graph model and undo and redo all edits of the model. We encountered a small bug at this point where the undo item was not correctly activated but fixed this fairly quickly. After assigning a keyboard shortcut to all actions we were only left with mouse input.

We started with the *SelectionController*. First we had to make sure a node or edge is selected if the mouse is clicked while hovering over the component. To do this we looped over every node and edge to check if the mouse coordinates intersected with the hit-boxes, represented by a graph component's coordinates and size dimensions, if so it was selected. When the mouse is dragged with a selected node we move that nodes coordinates according to the mouse position. We noticed here that when holding down the mouse button and dragging over a node the node would falsely be selected so we introduced the mouse pressed method additionally to the mouse clicked method. This fixed the problem. Then for resizing nodes we adjust the size and coordinates of the selected node based on where the mouse is dragged. We change size if the mouse is to the bottom/right of the node and coordinates for the top/left. This let us avoid the problem of the nodes having a negative size and not being displayed anymore. In the end we used the window size to restrict all moving and resizing of nodes so that they could not be off screen. However, when loading a graph which was too big for the current window size they would still be out of screen, so we implemented a check for that and prompt the user with a warning to adjust the screen size. To ensure the mouse input works properly we made a class to print information on mouse events to the console.

Generally the process of creating this program was not a great challenge to us. With the experience using the MVC schema, GUIs in java swing and mouse input we had from the previous assignment, the few problems we encountered could be fixed easily.

6 Conclusions

In conclusion, our graph editor meets all the requirements and includes several other additional features that make the graph editor easy to use and understand for anyone who downloads our Java Executable. We followed the strict MVC pattern in order to ensure that the controller modifies the model which in turn updates the view accordingly. It is very important to maintain this pattern in order to keep the three packages separated and allow for future modifications such as implementation of a new controller or advanced view for the user. We have included all of the required functionalities for the graph editor such as: creating/editing/deleting nodes, creating/deleting edges, node draggability, and the ability to undo and redo each and every action made. Java's UndoManager allows us to store every undaoble action in order to allow the user to undo and redo up to a certain extent. Thorough bug testing has revealed minor bugs that become present when the user performs unexpected actions; however, full bug testing is not yet possible as we cannot fully predict what the user will do in the graph editor.

Our code includes all of the mandatory Java Docs so that any new avid programmer can easily and quickly understand the function of each and every class. We followed the MVC pattern strictly in order to allow future modifications to be made to the code without the hassle of each aspect of MVC being intertwined. If a future programmer wants to swap out our controller or view for a more advanced one, the process of connecting the model to these new advancements is a simple task. The code is fully maintainable as we have three main packages: controller, model, and view. This strict separation allows for future modifications to be made to any of the three aspects of MVC and will not result in broken code as our packages do not heavily rely on each other. A few future extensions that would be quite enjoyable to have are: custom image nodes, image backgrounds, changing a node's color and shape, multiple node selection alongside copy and paste, and the ability to view the current status of the UndoManager in order to select a specific previous action rather than cycle through all the undos/redos.