Object-Oriented Programming
Programming Report
Graph Editor

Annika Möller *Andrei Mihalachi*
S3585832 *S3491862*

June 23, 2020

1 Introduction

For this assignment we had to make a graph editor. The program allows the user to add nodes, move them around, connect them with edges, and rename the nodes. An edge always connects two nodes. Likewise, the nodes and edges can be deleted, and the nodes can be renamed. All of the previously mentioned actions can be undone and redone. A graph can be saved to a file, opened from a file, loaded from a provided command-line argument, or a new empty graph can be created.

2 Program design

Our program is structured based on the MVC (model/view/controller) pattern. The model handles the logic of the program, the view structures what the user sees and the controller is how the user interacts with the model. A clear structure of how this was implemented can be seen bellow:

- **Model**
  The model package is where the classes GraphModel, Node and Edge are set. These classes handle the functionality of the nodes, edges and the graph itself, i.e. the logic of the program. The GraphModel class holds the structure of the graph, composed by several nodes and edges, while also being responsible for editing said graph (removing nodes and its edges, adding nodes and edges, etc). The Node and Edge classes are responsible for the nodes and edges and their specific sub-fields, such as node names or information about which edge connects which node.

- **Controller**
  The controller package holds all of the code for controlling the graph editor. The file 'SelectionController' extends MouseAdapter and handles everything to do with mouse input. It implements MousePressed to select a node or an edge, or a second node when connecting two nodes with an edge. MouseDragged is used for dragging nodes. MouseMoved follows the cursor's location so that when the user selects 'add an edge', a line is drawn from the first node to the cursor, wherever the cursor is moved to. MouseReleased is used so that after a node has been dragged, it creates a MoveNode UndoableEdit and adds it to the undoManager.

  - **Undo/Redo**
    Our program makes use of an UndoManager() inside of the GraphModel class, so that whenever an action is performed, it is added to the stack of UndoableEdits. Therefore, we have a class for each AbstractUndoableEdit related to each action, stored inside this package. Each one has its own implementation of redo() and undo(), which are called whenever the action is undone or redone. Redo() is also called whenever an action is performed (not necessarily being redone).

  - **Actions**
    The actions are connected to buttons at the top of the screen. These are for performing the key actions that define the functionality of the editor such as adding/removing nodes/edges, editing a node's name, and undoing/redoing actions. When a button is clicked, it calls the code inside the relevant action class. The action class itself does not run the code, but instead calls the redo() function inside the action's relevant UndoableEdit, so that it is added to the graph's UndoManager().

- **View** The view package holds everything to do with what is displayed on the user's screen. The GraphFrame class constructs the actual frame where the panel and menu bars are displayed. GraphPanel holds all the methods for painting and highlighting nodes and edges. MenuBar is a JMenuBar that holds all of the buttons for adding/removing nodes/edges, renaming, undoing, and redoing, while TopMenuBar holds the drop-down 'File' menu that allows the user to create a new empty graph, open a graph from a file, or save their current graph. MenuPanel is a second panel in the frame that contains the menu bars - MenuBar and TopMenuBar. The SaveAndLoad class holds the methods for displaying a JFileChooser whenever the user chooses to save or load a file.
  - **Buttons** This package holds the code for creating the buttons on MenuBar. Each button is connected to its corresponding action in the 'actions' folder in the 'controller' package.
- **Input/Output** The io package handles saving and loading graphs. In our program, saving is done by writing to a file the nodes, their names and their respective edges, this is all done using a FileWriter. This file is saved to a chosen directory with the extension .graph.
  Loading is done by scanning a chosen file with the extension .graph. We scan the file character by character until we reach the end of it and store all the information (nodes, names and edges that connect the nodes) accordingly. The scanning is done using a scanner instead of a BufferedInputStream. The reason we choose to do this is because we are scanning integers and not just characters. Using a scanner seemed more straight forward in our case than a BufferedInputStream although BufferedInputStream might have been a faster way to scan the file.

3 Evaluation of the program

The program is almost completely as we intended it to be in the beginning. One thing that we tried to do but did not succeed at was making sure that the user cannot drag a node outside of the bounds of the JFrame. This was not a requirement but it seemed like sensible element of functionality to add. Another thing that doesn't function completely satisfactorily in our program is the dragging of the node with the mouse. Once the node is clicked on, the cursor does not stay on that position, but can instead be moved around within the bounds of the node while it is being dragged. We tried to fix this but could not find any solution. Finally, since the cursor can move within node bounds while a node is being dragged, when one node is dragged right next to another, the cursor sometimes switches to the adjacent node, making it possible to swap between nodes while dragging, which is not ideal. Again, this is an issue with the cursor not staying in place on the rectangle of the node once clicked, which we didn't manage to fix.

4 Extension of the program

We decided to implement the following extensions to the program:

Allow the user to select edge for easy deletion:
This extra was added accidentally when implementing the basic structure of the program. For us it seemed to be the most straight forward way of deleting edges and decided to base the way we delete edges on it.

Resizing the node name to fit the node:
We're not sure if this really counts as an extra, but having a node name that was too long would result in the text going outside of the bounds of the node. Hence, we created a method to decrease the size of the font as the length of the node name increased.

5 Process evaluation

We followed the recommended process structure given in the assignment. First we created classes for the nodes, edges, and the methods inside of the GraphModel, which was fairly straightforward. At first we were simply storing the indices of the nodes connected to the edges inside of the Edge class, but later on we decided to switch to storing a list of the edges connected to a node inside of the Node class, since it was easier to remove and add nodes this way. Then we started on the Save and Load classes, using a Scanner to write the contents of the GraphModel to a file. Then we created GraphEditor to test what we had done so far by creating a GraphModel and saving its state (to a .txt file at first) - then opening the file and seeing if it was correct. Later we changed the file extension

to .graph. Part 1 went smoothly, but we had some more problems in part 2. We noticed that when painting the edges, they were being painted over the nodes, but later managed to fix this by changing the order with which we called all of the paintX() methods inside of paintComponent(). Painting an element last means that it will be on top - so we made use of this to fix that problem. Another issue that we ran into here was displaying two menus, one directly after the other. We spent quite a lot of time researching how to do this using java swing's box layout functionality. Eventually we managed to create two separate panels, one for the graph and one to store the two menu bars, and layout the two menu bars correctly on the top panel. It was also a challenge to be able to add edges between two nodes, since the way that we do this is by first selecting one node, then clicking the 'Add Edge' button (which causes the edge to be drawn from the node to the cursor), then clicking on the second node. We managed to do this by using the boolean variable isCurrentlyAddingEdge() in GraphModel, so that the SelectionController would know that it was clicking on the second node. Using the UndoManager was quite challenging and took a lot of time, since we had to rewrite every action for each button to be inside the redo() method of the undoableEdit. Additionally, we had to make different versions of the redo() method - one for actually executing the action and one for redoing the action, since the corresponding variables used in the method were different depending on this condition.

6 Conclusions

The program satisfies all the requirements given in the assignment and works properly. We paid close attention to the model-view-controller pattern, making sure that each part of the code is organized into the correct package (or sub-package), and that the relationship between the components of the code is in line with this. The controller updates the model, and the model updates the view. As such, this helps the maintainability of the code. We also tried our best to adhere to OOP standards, separating long methods into smaller ones with clearly defined purposes, using proper naming conventions and encapsulation, and commenting the code. We didn't have time to implement many extensions to the code, but some that would be nice and probably not too difficult to add in the future would be resizing nodes, selecting a shape for the node, using custom colors, and copy-pasting nodes.