Object-Oriented Programming
Programming Report
Graph Editor

Radu Rusu *Nadia Bosgoed*
S4036999 *second author's s-number*

June 25, 2020

1 Introduction

In this assignment we had to create a graph editor with a graphical user interface, capable of doing a set of operations in order to edit undirected, simple, unweighted graphs without self loops. The requirements were divided in 2 parts: functionality and graphical interface. These fragments are meant to create and model the undirected graphs, as well as letting the user to interactively check the changes done.

In terms of UI, we implemented the basic functions like saving, loading a graph or start a new blank page. More specific functions for editing are adding a node, edge or removing one of them. In addition, the handy methods like Undo and Redo were also important for the completeness of the graph editor.

Speaking about GUI, our program is capable to display all the components of a graph, modify its shape or rename its objects, some operations are accessible by different menus and buttons, which are also one of the initial requirements.

It is also worth to mention that we had to implements certain patterns and special packages that are part of Java, that would pack everything in a nice program, so that an user can intuitively and without much effort use the graph editor. We are going to explain the main part of them in the next paragraphs.

List of requirements:

1. Draw a graph
2. Create a new empty graph
3. Save and load graphs
4. Load a graph from the path provided in a command-line argument
5. Add nodes, edges
6. Remove nodes, edges
7. Rename nodes
8. Select and move nodes
9. Undo the following actions: adding nodes/edges,removing nodes/edges,renaming and moving
10. Redo the following actions: adding nodes/edges,removing nodes/edges,renaming and moving

2 Program design

The structure of the program is mainly defined by the **MVC** pattern, we have divided the logic of the program into three parts: view, model and controller, communicating with each other. In this way we enhanced the encapsulation constraints of the program, since we separated all the operations done to store or compute information from the way it is presented to the user.

First of all, we developed the model package, where the data, logic and rules of the application are managed. The head class that carries all the data is **GraphModel**, it receives any inputs from the controller and decides what

to tell the view to display. As the main components of a graph are the nodes and edges, we defined a separate class for each of them and created 2 ArrayLists in GraphModel in order to easily add or delete components and to keep track of them. We implemented the relation between a pair of nodes and an edge by including two integer variables in the Edge class, which store the index of the nodes that are connected. There were more possibilities for saving which nodes are linked by an edge, for example keeping the exact instances of nodes in the Edge class. However, we used the indexes as described above, because of the saving format we had in the requirements of the assignment. As a result, we discovered some possible bugs, which we will describe in the following parts of the report. When deciding how to design the Node class, we had a choice to make it be an extension of the Rectangle class in Java. However, we realized that there would be harder to make each node observable, so we chose to define the variables that would store the X, Y position, the width and height of our node. As it was mentioned before we used the **Observer** pattern to notify the view that the state of the model has changed. This suggests that, we defined in GraphModel all the methods that modifies the graph, like adding and removing, nodes or edges and reactions of the model in case of events from the controller part. Moreover, another important part of the model is the IO, we created a separate package for this purpose with a **SaveLoad** class. There are two functions SavaGraph and LoadGraph, we used the **Decorator** pattern to create a new PrintWriter, so that we can write to a txt file all the important information that has to be saved. Another district design decision was implementing a JFileChooser, so that the IO interface would be more user-friendly.

As a next step, we implemented the view package, which is responsible for presenting the data to the user, acting as an observer over the changes that happen in the model. At this stage, one of our main targets was making the interface intuitive and simple for the user, as well as, creating room for other future extensions. Consequently, we decided to use the predefined **javax.swing** package, rather than creating our custom one or any other more complicated ways, so that we could implement the view components and also some aspects regarding the controller part. The start was creating the common **DrawPanel** and **DrawFrame** classes. The Frame has a "is-a" relationship with the JFrame class, as a result it was easy to define the initial settings regarding appearance using the methods of the super. We set up the program to stop when closing the frame and its default size, made a new instance of JMenuBar, DrawPanel and created a controller for the mouse input. Since the foundation of the view was done, we implemented the Panel that would display the components of the graph and defined three methods: paintAddingEdge, paintEdges and paintNodes. Each of these functions, paint a specific object, the nodes, the edges and the new edge that is added to the graph using the cursor. In addition, we created two getters: getRatioX and getRatioY, which maintain the the dimension of the nodes when modifying the size of the screen.

The last part of the program is the controller package. Its main purpose is to act as a transmitter between the view and the model, listening to events that are triggered by the view and responding with a reaction to them. In our case, each button press or mouse actions has a specific method in the model, this circle process ends when the view is refreshed after all the changes were made. We divided this package in 2 other ones: actions and buttons. Actions consist of four distinct classes that extend the **AbstractAction** class and contain the name and the call for the corresponding function in the model for each button. Moreover, we created four button classes that extend **JButton** for the most important methods of the model, manipulation with the components of the graph. Nevertheless, we have two more classes: ButtonBar and SelectionController. The first one is an extension of the **JMenuBar** and combines the buttons set earlier and two JMenus that have some basic buttons like, opening a new page, saving, loading and UndoRedo functionality. On the other hand, SelectionController contains the overridden methods of the **MouseInputAdapter** super, mouseClicked, mouseReleased, mouseDragged and mouseMoved. These functions are needed to recognize if the user clicks on any of the nodes or tries to modify their position. The information provided by the mouse adapter is vital for many functions from the model, which select a specific node or edge and change its settings. Every modification is transmitted to the view, so the user can see the result of his actions.

As can be seen, the MVC pattern enables logical grouping of related actions, so the code is tidier and more easy to debug or implement new functionality, since each part does not rely so much on the rest ones.

3 Evaluation of the program

The program is according to our initial plan, we implemented step-by-step each point from the requirements of the assignment. After some adjustments, the nodes are moving smoothly and the edges follow the exact trajectory of the cursor. However, after some more testing, we noticed that when loading from a saved file, one of the nodes is always selected and the same happens if we delete one of the nodes. In our opinion, it is not a big issue, sometimes it can be handy, as the user do not have to do so many clicks. This can be solved by analyzing how the variables that store the selected node are reset. Another problem is the UndoRedo functions, they are not always really accurate. We think that by the end of the assignment we will find a solution for every problem.

4 Extension of the program

[*This section is optional. If you added any extra or extensions to your program, list and describe them in this section.*]

5 Process evaluation

In the process of developing the program, we encounter a set of bugs when removing some components of the graph, which in the end were successfully solved. First of all, as we mainly work with the indexes of the nodes and edges, there was a problem with the nodes stored in the edge instances. Since we implemented a method that removes Nodes, using the $remove()$ function from ArrayList package, we have not taken into consideration the fact that after deletion all the nodes that had a greater index than the removed object, are shifted with one position down. This have led to errors because of invalid relations between nodes. Our solution was quite simple, as we added some more lines of code to the remove method. We iterate over all our edges and for each of them in case the node index stored is greater then a input index we subtract one.

Another mistake that we did when coding the program was that there was impossible to delete an edge in case it was the only one. The method for removing edges has as parameters the two indexes that are stored in each instance of edge, so it has to iterate over the array to find the right one and delete it. However, we could not use the **for-each** loop, because it does not support deletion when iterating and triggers a concurrent modification exception. As a next attempt, we used a standard **for** loop, but unfortunately, as we go up to the size of the array minus one, in some cases, the loop was not even starting and there was no effect on the edges. In the end, the fully working alternative was using lambda, by defining a predicate that removes an instance if the condition holds.

All in all, we can say that the assignment was not the easiest one, since we had to make use of all the concepts that we learned in this course. Neither of us had much experience with development of GUI, so coding using swing and MVC pattern dividing was the hardest part of the assignment. However, we learned a lot about the implementation of user interfaces and some minor but important things typical for java.

6 Conclusions

In conclusion, we believe that our program works properly, since it has not failed the functionality tests and satisfy the initial requirements. There might be some minor details that can be done in a better way, but because we were limited in time, our main concern was the stability and clarity. We think that the way the program was implemented grants a quite wide room for scaling it up with some extra functionality. Developing the support for different types of graphs or adding more customization settings are some of the extensions that would be nice to have in the future. We are convinced that we will have the chance to use all our creativity later in another projects.