Object-Oriented Programming
Programming Report
Graph Editor

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

July 9, 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 two 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 starting 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, all 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. Moreover, this way of organizing the code is very handy when more people work on the same project and is widely used nowadays in designing web applications in Java and other programming languages.

2.1 Model

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 components and functionality of the model is **GraphModel**, it receives any inputs from the controller and it is part of the Observer pattern, acting as an observable object for the view.

Another two important components of a graph are the **nodes** and **edges**, so according to the object oriented fundamentals, we defined a separate class for each of them. The relation between a pair of nodes and an edge is implemented by saving the two Node instances in the Edge class. There were more possibilities for saving which nodes are linked by an edge, for example keeping their index or any other unique identifier of the nodes in the Edge class. However, we used the most optimal solution, because in this way we have saved a bit of memory, since there are no extra variables with identifiers, reduced the chance of some unexpected bugs and we have also respected the OOP principles.

When deciding how to design the **Node** class, we had a choice to make it be an extension of the Rectangle class of Java. Initially, we have not done that, but after going more deep in the essence of the program requirements, we realized that that would be the best and most efficient way to cope with some future problems regarding more advanced functionality. One great feature of the super class that is the fact that we could get the area of a node and check if the controller components like the cursor interacts with it. We will provide more details in the next paragraphs about the controller part.

As the basic components of a graph were implemented, we stored them in two **ArrayLists** which were set up in the GraphModel class. 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, create new object instances and other functions in case of events from the controller or view part. In addition, we have thought a little bit ahead and make one more array **selectedNodes**, which has all the nodes instances that are selected by the controller. In this way, we organized everything, so that we could easily modify any component of the model and make them communicate the necessary information to the rest of the program.

Another important part of this package is the IO functionality, we created a separate package for this purpose with a **SaveLoad** class. There are three functions **SavaGraph, LoadPath and LoadGraph** that create save files and fetch all the important information that has to be saved in a predefined format. The last one is responsible for choosing the right loading path, as the program can be started from the command-line with the path of a save file. We used the **Decorator** pattern to create a new **PrintWriter**, so that we can write to a file and also save it with our custom extension **.graph**. The most important component of IO is **FileChooser**, as it has a lot of default features, so that we have not have to created a new frame in order to make the IO interface user-friendly.

The last requirement for the model of this program was to make some actions undoable. We decided to implement this using **AbstractUndoableEdit** class and the built-in manager **UndoManager**. We created an instance of the manager in the GraphModel and also a getter method, so we can use it when any action from the controller is triggered. We defined an edit for adding, removing, renaming and moving a component of the graph and placed them in the **UndoRedo** package of the model. Each edit has two methods: **Undo** and **Redo**, which implement the way an action can be undone or redone, for example an **addNode** edit, will call the **removeNode** method in the undo function and actually the **addNode** in the redo one. The idea behind this is pretty simple, before the program does modify its state, it makes an edit with the present properties and adds it to the manager, then it calls the redo method to perform the action that the user desires.

2.2 View

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 a custom **WindowController**, so that the program stops and warns the user about saving the current work when closing the frame. In addition we defined the default and minimum size, made a new instance of ButtoMenuBar, ButtonBar, DrawPanel and created a controller for the mouse input. The first bar is an extension of the **JMenuBar** and combines two JMenus: **File** and **Edit** that have a set of **JMenuItems** that trigger actions like opening a new page, saving, loading, undoing and redoing.

In the first version of the program, all the buttons were part of the **ButtonMenuBar**, but later we created another class **ButtonBar** that extends JPanel and implements Observer interface. We have done this way because we wanted to make the UI look nicer and also split the most used functions in a more accessible place. We created four button classes that extend **JButton** and added them to the bar. Moreover, both bars have to be able to set its buttons enabled or disabled according to the status of the model, that's why they both implement the Observer interface, were added as observers to the GraphModel in the DrawFrame class and have a method **changeButtonState()**. As an example, the **removeNode** button is enabled if and only if the **selectedNodes** array from the GraphModel is not empty.

Since the foundation of the view was done, we implemented the **DrawPanel** 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 by iterating over the arrays with this components from the model and the new edge that is added to the graph using the cursor. In addition, we have made the controller communicate directly with the view by setting the position of the cursor in the DrawPanel, since we need it when we draw an edge creation.

One of the last improvements that we have done was the **Warning** class. It extends the **JOptionPane** and shows different types of pop-ups. There are a lot of situations when this class might be handy, so there is some room for other new functionality that will require multiple windows for receiving an input or a yes/no selection. However, we only used this for saving warnings when user tries to close or load another graph.

2.3 Controller

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 action calls a specific method in the model or view, this circle process ends when the view is refreshed after all the changes were made. The controller divides in two parts the **Actions** package and two controllers: **SelectionController** and **WindowController**.

Actions consist of several distinct classes that extend the **AbstractAction** class and contain the name and the **actionPerformed** method which is inherited from the super and overridden with the corresponding function in the model for each button. As an example, the **ExitAction** has the GraphModel as a parameter in the constructor and the function actionPerformed, which is called when an event of the corresponding button is received, creates a **Warning** for saving and then exits the program.

The **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, selecting them, tries to modify their position or whants to create a new edge, so the coordonates of the cursor are transmitted to the DrawPanel. This class modifies the state of the model and since it is observable, every modification causes the view to repaint, so the user can see the result of his actions.

The last class from this package is the **WindowController**, which extends **WindowAdapter** and modifies the behavior of the **DrawFrame** when it is closed by the user, triggering a saving warning and only after that it stops the program and closes the frame.

All in all, 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 and if needed the controller or the view can be swapped with another ones without a lot of changes in the model.

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 and also introduced some extra ones. After some adjustments, the nodes are moving smoothly as we took into consideration the offset of the cursor, the edges follow the exact trajectory of the cursor and all the buttons and actions are functioning as intended. We noticed that when saving if the user types the name and also an extension, the name of the file will have two extensions, which is not very good. However, we think that is not really an issue, since the user has to write only the name in that space as our custom extension and the format of the saving name is explicitly shown in the **File of type** box.

An warning message could be implemented in the future, but we can say that we implemented all the requirements as they were described in the assignment.

4 Extension of the program

4.1 Keyboard Shortcuts

We added keyboard shortcuts for the most important functions to the graph editor. We chose intuitive shortcuts, like **Ctrl + A** for selecting all the elements of the graph or **Ctrl + Z** to undo an action, so that the user can do some basic actions faster. All the shortcuts can be seen when opening the menus.

4.2 Select all elements

We implemented a special button and action for selecting all the elements, so that multiple nodes can be deleted at once and the edges that are connected to them.

4.3 Warning Pop-ups

We created a class that creates different warning messages and implemented a saving warning, when the user closes the window, makes a new graph or loads an existing one.

4.4 Dark Mode

Another extension of the program was a dark mode, that changes the color of the background and the components, so that the user can work in a more comfortably environment, as his eyes do not get tired so fast.

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. The most challenging part was to debug and make the **SelectionController** and **UndoableEdits** work stable.

We had some problems when trying to move a node and overlapping two of them, since the controller was switching to the other node. We managed to solve this issue by defining more accurate and strict conditions in the mouse controller functions and introducing the **checkForNode()** method, that is always called before some interaction with a node is done.

Speaking about the undoable actions, the main problem was that we had to rewrite some of our code in order to implement this functionality and create the edits before actually doing the action, since some of our code was badly organised and the MVC pattern was a bit violated. Fortunately, we were able cope with this challenge by rethinking some of our algorithms and transferring some of the methods that were part of the model package to the controller and view.

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.