Object-Oriented Programming
Programming Report
Graph Editor

George Bondar *George Buzas*
s4060989 *s3964531*

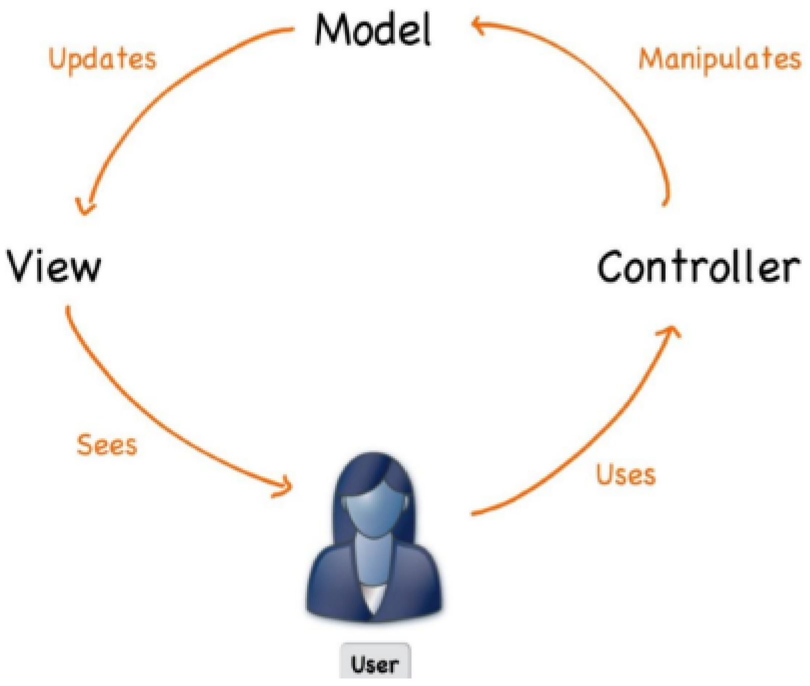
June 23, 2020

1 Introduction

Writing a graph editor program in java was a challenging experience. The initial requirements of the assignment, which consisted of creating the base model of the graph-editor along with saving and loading, were approached by us following the techniques taught in this course. The model of the graph-editor is represented by nodes and edges, with both containing some specific characteristics. The nodes of the graph, represented by a name along with a size and a location, whereas for the edges of the graph, they are represented through a name and the nodes, they are connected to. Furthermore, the model that keeps track of all the edges and nodes, allows the user to edit the graph by adding or removing any of them. When it comes to, saving and loading, this is done in a custom format, where the user can choose the naming convention of the file along with the location where it can be saved or loaded from. To give a general overview, the graph-editor program is largely concentrated around the idea of editing an undirected, unweighted graph without self-loops by moving around, inserting or deleting different nodes, respectively, edges.

2 Program design

The program design has been mainly focused on the use of the Model View Controller principle, Java I/O, and Swing. Having this said, we created three main packages, **controller, model** and **view** which are based on the following diagram:



Following the MVC principle in the **controller** package we have created buttons and corresponding actions for them that would make the editing of the graph possible. Not only that, but we decided that it would be a good design choice to implement a *ButtonBar* which extends **JMenuBar** that would get a hold of all the buttons. Moreover, each action has a corresponding method inside our graph modeling class, methods that make the graph-editor functional. To sustain the functionality, enabling conditions have been implemented for the buttons. In order to allow the user to make use of the mouse cursor's location, we created a class called **SelectionController** in which we allow the user to select a node, draw an edge or move a node inside the panel. This specific class provides us the means of some *Override* methods for the mouse action events. Following our program design choices, this class contains a **mousePressed** method that makes the selection of a node and the drawing of an edge possible through the event of pressing the mouse key. We also decided that it would be a good idea to indicate that a node is selected via a **setter**. The hit-box for a node was created using Java **Rectangle** which has as bounds the x,y coordinates, and the height, width of a node. When it comes to dragging a node this requirement was implemented using the *Override* method **mouseDragged**. This method besides creating a hit-box for a node in the same way it was created for selecting it, contains some conditions that prevent the user from moving a node outside the panel. Furthermore, a **mouseMoved** method was created in order to let the user move an edge along with the mouse cursor to the node that he/ she wants to connect it to. In the end, to make everything possible inside our **SelectionController** constructor we initialised our graph with properties that handle the events of a mouse when it is or it is not in motion.

Following the **model** package and the methods that correspond to the actions of the buttons, created inside our **GraphModel**, we are going to discuss the ones that feel more important and would have different ways of being implemented. When it comes to removing a node, for example, this is done by applying two conditions, one says that there has to be a selected node that is going to be removed and the other one checks if there are any edges connected to that node, which are going to be removed as well. The last condition is checked by making use of the *getters* that correspond to the nodes, the edge has a connection with. Furthermore, a boolean method that determines the connection of an edge between two nodes and a setter that indicates that connection was created. Last but not least, we created **move** method that allows the user to drag a node. This method takes as *parameters* the x and y coordinates and sets them via **setters**, that were created inside our node representative class since those coordinates are related to a selected node from the graph. To conclude, in order to display all the changes made in the graph we created an **update** method that notifies all the observers of those changes, along with making the model of the graph-editor *Observable*.

Inside the **model** package, besides the **GraphModel** class, there can be found two more classes: **GraphNode** and **GraphEdge**. Inside the **GraphNode** class you can find the definition of a node. A node is defined by its size, location, and name. The defined characteristics of a node were important since it empowered us to implement the action of drawing an edge in a correct manner. On the other hand, they were also useful for preventing a bug when the user could drag a node outside the graph panel (these were useful to set the bounds correctly).

The **GraphEdge** class is where the definition of an edge can be found. Except for the methods, that were extremely useful throughout the project, an important particularity that this class possesses, is the following: an edge is defined between **two nodes**. This way of implementing the **GraphEdge** class gave us the opportunity to manipulate with ease the methods where we draw/ remove an edge.

Ultimately, the last out of the main modules that will be discussed is the **View** package. The classes that belong to this package: **GraphFrame**, **GraphPanel** set up the opportunity for a user to visualize the changes that have been made to the **Graph** due to user's actions. The **GraphFrame** class is where we set our graph-editor frame with the corresponding buttons and a panel where the view of the graph manipulation is displayed.

Inside our **view** package the **GraphPanel** class plays an important role in this project because inside this class, we are doing the drawing of the graph components. The nodes represented through rectangles and the edges through lines. Following up, we decided that it would be a good idea to work with *JOptionPane* and *JComboBox* to allow the user to select an edge to be removed. By making use of an index, the corresponding method inside our **GraphModel** which is passed to the action of the button, allows the user to remove the edge he/ she wishes to. Another method that gives the user a chance to rename a node is done using a *String* variable that contains the new name of that node typed in a standard dialog box, so-called *JOptionPane*. In addition, the new name of the node is created inside our **GraphModel** through another method that references the one inside our panel and is further passed to the action of the button.

An extra package was created to represent Java I/O, more precisely it contains a *Serializer* class with all the methods needed for saving and loading a graph. The naming convention for this class was chosen due to the fact that it contains two extra methods that allow the user to save and load a graph using **Serializable** technique. Furthermore, the other two methods were created based upon the assignment requirements, a saving method is done using **FileWriter** and **BufferedWriter** for the sake of the custom format that is suggested and a loading method that uses a *Scanner*, **FileReader** and a **BufferedReader** to load the custom format of the saved graph.

Thanks to **JFileChooser**, saving and loading done in a custom format, enable the user to choose the location of the file, that he/ she wants to save it to or load it from.

3 Evaluation of the program

After finishing the implementation of each method, my colleague and I were testing the edge cases where the program might crash. We solved any bug that we observed our program posses, and we did this by surfing the internet and also by asking for help from our teaching assistant. Contrary to our initial beliefs, our final program is not as good as we intended to be since it does not meet all the requirements(we did not implement the undo or redo functionalities). Moreover, before writing our first line of code for this project, we expected that we would implement more extras than we did.

4 Extension of the program

When it comes to extras, we have added two more methods inside the **Serializer** class that belongs to the **io** package. These two functions are regarding the save and load functionalities of this program. The **Serializable** interface provided us with another approach regarding saving and loading the state of a graph. By implementing this interface, the Java objects have been converted into a stream of bytes that can be saved in a file. The extra **Save** method enables the user to save a graph in a specific directory and with a specific file name. The **Load** function gives the user the opportunity to load a saved graph given a specific directory and file name. The difference between these functionalities and the custom format for saving and loading a graph is that in the custom format the graphs are saved as a ".txt" file, whereas, using the Serializer type, the program encodes the information provided by the objects that are going to be saved and uses the load method to decode those objects' information in order to be further displayed for the user. For successfully implementing the serializer saving and loading features, we used FileOuputStream, ObjectOutputStream and FileInputStream, ObjectInputStream. This way of implementing the above-mentioned functionalities implies that any class that we want to save should implement "Serializable". However, each class that implements that interface should also have a field called **serialVersionUID**. It goes without saying that if the number of classes that needed to be saved is larger than **one**, the value for the field **serialVersionUID** should be **distinct** in each class.

Another extra that has been implemented is regarding the design of the panel. We have added two **JMenu**s such that both options of saving and loading could be displayed for the user and would give him/ her the chance to choose.

5 Process evaluation

Although this project was a laborious one, it was beneficial to ourselves because it enabled us to have a better vision regarding the Object-Oriented Programming concepts. This process of creating the graph editor program, given our lack of experience, lasted almost a month. We worked to get it to its final state on a daily basis during this period. After having finished the code for the project we started writing this report, which enables us not only to outline the most important aspects of the program but also to describe what factors influenced us during this creation. When it comes to the difficulty level of the assignment, we found more accessible parts of the project where we implemented the design features. Though, even the methods that enable the user to add or remove a node or an edge, create a new empty graph, and rename a node were not too difficult. The arduous parts of the project were those that refer to the functionalities that enable the user to drag a node, draw an edge between 2 nodes, and remove an edge. The part where we check if a node has exceeded the bounds of the panel is neither easy nor difficult. During the process of creating the graph editor, we made some interesting mistakes that need to be described in more detail. For example, we misunderstood the way in which an edge should be drawn. Initially, we thought that only when the user has selected exactly two nodes and chooses to connect them via an edge, the program should draw the edge. But thanks to the help of our teaching assistant and our process of thinking we succeeded in drawing an edge in the required way. Besides this mistake, until we realised how we should set the bounds of the panel correctly, we also found a bit cumbersome implementing a method that does not allow the user to drag a node outside the panel since our first implementation of this functionality was slightly bugged(when a node would reach a border or a corner of the panel the node would bounce a few pixels). This assignment helped me and my colleague to grasp the main principles of Object-Oriented Programming. Consequently, we found this assignment useful as it increased our teamwork skills which is a mandatory asset to possess in this field of expertise. Furthermore, it enabled us to apprehend the importance of encapsulation whereas the use of getters and setters to manipulate or extract the value of a field.

6 Conclusions

Unfortunately, I and my colleague did not accomplish all the requirements such as the undo and redo functionalities. From our perspective, the code that we wrote is maintainable for the following reasons. Since we treated our code with respect to the MVC pattern, it would be pretty straightforward for the developer to make a change. Likewise, we used Java documentation to confer more insight into what each class or method does which is an important factor in programming because in case another person, who did not participate in developing the code, would want to understand what the code does, it would be too tough. In addition to this, we followed the Java naming convention which supposes that we used whole words and avoided abbreviations. In the same way, we used the camel case convention for naming our variables. However, each word in a class name is a capital letter. Given these facts, we assume that the maintainability of our code is on spot. When it comes to future extensions, our program could implement the undo and redo functionalities, whereas it could implement methods for creating a directed, weighted graph. Additionally, our program could implement a method such that when the user types the number of nodes and edges, it will draw an entire graph from the outset that contains as many nodes and edges as the user has chosen and created random names and locations for the nodes. Another useful extra, that we believe this program could have is that we create a button named 'Duplicate Graph' which basically when pressed, will look the same as the initial graph. To sum up, we are glad that we successfully implemented almost all the requirements and that we evolved as programmers.