Object Oriented Programming
Graph Editor

Rocholl, Niels
s3501108

Zhelev, Ivaylo
s3782255

June 22, 2020

1 Introduction

For the final project of the Object-Oriented programming course we were assigned the task to make a basic graph editor with a graphical user interface. For this assignment the GUI must be implemented using Swing and the MVC pattern. A graph can be seen as a collection of nodes and edges. All nodes are represented by rectangles which have a specific location and all edges are connected to two different nodes. In the following section we discuss this in greater detail. The graph editor has a few minimum 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.
6. Remove nodes.
7. Add edges.
8. Remove edges.
9. Rename nodes.
10. Select and move nodes.
11. Undo the following actions:
    (a) adding nodes/edges.
    (b) removing nodes/edges.
    (c) renaming and moving.
12. Redo the following actions:
    (a) adding nodes/edges.
    (b) removing nodes/edges.
    (c) renaming and moving.

We implemented all of the listed requirements and we also included a number of extra functionalities within our graph editor. The extras we implemented are:

1. Resizing nodes by dragging either corner or side of its corresponding rectangle.
2. Different themes/modes.
3. Keyboard shortcuts.
4. Optional save and Save as...
5. Copy/pasting nodes.
6. Selecting edges.
7. Directed graphs.

In the following section we will take a look at the program design.

2 Program design

In this section we will discuss the design of our graph editor program. This section will be split up in to 5 sections, in which we will discuss the workings of the classes within a particular package. We included 5 key components in our design - the traditional Model, View and Controller, as well as Metadata and IO. Metadata was needed because there is a lot of information that needed to be observed by the view and manipulated by the controller but semantically did not belong in the model. IO was used for loading and saving files, including the icons and sound files we used for the Super Mode extra.

2.1 Model

The model package holds three classes, namely `GraphModel`, `Node` and `Edge`. We will shortly go over these classes and explain their working since their logic can be seen as the backbone of the whole program.

GraphModel
The class Graphmodel keeps track of the nodes and edges within the program, which it does through the use of array lists. Next to that it contains the methods that ensure safe manipulation of the model, like for instance "createNode" or "removeNode". So once a button is clicked, the underlying action will trigger a method in GraphModel.

Node
The class Node is used to create nodes. A node is characterized by a location (x,y), a height, width and a name. We use the build in java class Rectangle to store the x, y, width and height values. Next to that we use a string "name" to store the name of the node. The rectangle together with the name is a node. Next to this the Node class also houses methods for resizing the node.

Edge
The class Edge is used to create edges. An edge is characterized by the two nodes which it connects, node N_{start} and node N_{end}. These two nodes can be used to acquire the (x,y) coordinates it needs to connect the two nodes.

As we can see, the model of the Graph editor is fairly simple, since it is only defined by N amount of nodes and E amount of edges. In the following section we will move on to the view. There we will discuss how the edges, nodes, buttons and menu items will be displayed to the user.

2.2 View

The view package holds all classes that create the GUI. The package holds five classes.

GraphFrame
The class GraphFrame extends the class JFrame and therefore allows us to create the JFrame in which we can create JPanels and JMenuBars.

GraphPanel
The class GraphPanel extends the class JPanel, and is used to create a panel in which we draw our nodes and edges. It draws the nodes by looping through an array list of nodes which it retrieves from the GraphModel. For each node the rectangle object is used to draw a node/box according to the rectangle's parameters (x, y, height, width), and the name will be drawn inside the node.

ButtonPanel
The class ButtonPanel extends the class JPanel, it is used to house the buttons. There are five buttons:

- Add Node.
- Remove Node.
- Add Edge.
- Remove Edge.
- Rename Node.

MenuBar
The class MenuBar extends the class JMenuBar, it is used to house three JMenu's:

- File.
- Edit.
- Mode.

Arrow

In order to implement a "Directed graph" view for our design, we needed functionality that allows us to draw arrows from one node to another. As Swing does not provide such functionality, this is done by a custom class *Arrow*. The latter contains the static void method draw with the two rectangles that need to be connected, passed as arguments.

As this is a drawing function that is only used by the view, we really did not need instances of every arrow that is drawn. This is because they are not going to be accessed anywhere after they are drawn, which is why we decided to make the method *draw* static.

The class contains a non-static subclass *Vector2d* which we used for all the computer graphic functionality we needed for drawing the nodes. We encapsulated this class into the *Arrow* class as it is not needed anywhere else.

These are all the classes which are used to create the graph editors GUI. From these 5, GraphPanel is probably the most complex since it has to draw and redraw the nodes every time the state of the model changes.

2.3 Controller

The controller has 5 sub packages. We will discuss them one by one.

2.3.1 Actions

The actions package houses all the actions which are used in the buttons and menu items. In total there are 15 actions, so for the sake of conciseness, we will only name the actions and not give an explanation. This should not be a problem since most actions share similar logic and the class names are a good indicator of their functionality. They use the method `actionPerformed` in order to execute an action. The action is in most cases triggered by calling a method within the model. It might be worth noting that we made most actions observable by the ViewModel, so that we could keep track of when they should be enabled. They all implement the Observer pattern and contain `update` method that checks whether their pre-conditions are satisfied (and updates their enability correspondingly). The actions this package contains are:

a) ActionAddEdge.
b) ActionAddNode.
c) ActionCopyNode.
d) ActionCreateNew
e) ActionLoad.
f) ActionModeOff.
g) ActionModeOn.
h) ActionPasteNode.
i) ActionRedo.
j) ActionRemoveNode.
k) ActionRemoveEdge.
l) ActionRenameNode.
m) ActionSave.
n) ActionSaveAs.
o) ActionUndo.

These actions are used by the buttons in order to execute the requested action.

2.3.2 UndoableEdits

Every action that needs to be undoable (can be undone) was included here. When the different actions in the previous subsection are performed or when resizing/moving nodes, instead of performing the actions directly, the program creates undoable edits that are redone and saved into an *UndoManager* in the model. Each undoable edit contains all needed information in order to undo/redo an action. Typically, this includes the GraphModel, the property that was changed, its old value and sometimes the ViewModel. Depending on the specifics, they may contain several additional properties.

Every action that should be undoable contains a corresponding undoable edit, including:

a) UndoableEditAddEdge.
b) UndoableEditAddNode.
c) UndoableEditRemoveEdge
d) UndoableEditMoveNode
e) UndoableEditResizeNode
f) UndoableEditPasteNode.
g) UndoableEditRemoveNode.
h) UndoableEditRenameNode.

There is also an *UndoableEditDoubleRemoveEdge* that is used to remove 2 edges that connect the same nodes but in opposite directions.

2.3.3 Buttons

This module only contains a declaration of the *AbstractButton* with a constructor containing the *AbstractAction* that the button needs to perform and a method that sets the properties of the button.

2.3.4 Menus

This is a fairly simple component that just declares all the menus. An *AbstactMenuItem* is used to declare an overarching class that is extended by all the concrete menus, namely *Edit*, *File*, *Mode* and *Preferences*

2.3.5 SelectionController

This component was responsible for handling all the mouse events. The logic of it was quite complex but most simply put, different events were responsible for manipulating different components of the model and for implementing different actions. The *MouseClicked* event was used only for selecting a node or an edge. For adding/removing edges, we use the *MousePressed* method. When the button for adding/removing edge is pressed, it still has to wait for the user to select the connecting node which was handled in *MousePressed*. *MouseDragged* was used for moving/resizing nodes. *MouseReleased* was used to create timestamps of the moving/resizing node's location and size. This was very handy for the undoable edits of moving and resizing. Finally, *MouseMoved* was only used to track the cursor of the mouse when an user is adding an edge (this was needed so that the line coming from the node tracks the cursor while the user is selecting a connecting node).

2.4 IO

The IO package contains five classes which handle input or output of the program. The assignment gave us room to implement saving and loading as we wish, as long as we follow the requirements. Therefore, we made a custom loading and saving class which saves or loads .json files. We started off using .txt files, however we noticed that this format was error prone. The advantage of JSON is that it is human-readable and easily serializable, which means we can manually check the data which is loaded to a file. This is, in our opinion, quite useful since it removes a level of unnecessary abstraction. Next to this, the data is easily loaded to our program since we only need to look for the right keys in order to extract the right value from the JSON file. We will now shortly go over all classes.

json
This class is used to parse the data. This class is very small and only contains one method which is used in the class SaveAndLoad.

LoadIcons
The class LoadIcons is used to load 3 images and 1 audio file. We made a separate class for this so that the icons and sound files don't unnecessarily reload during runtime, and also, in order to handle all file-loading exceptions there. This class has 4 methods which return Image, BufferImage and audio files.

Mp3Player
The class Mp3Pllayer is used to play audio files. It can only play .wav files and it has one method to start playing an audio file and one method to stop playing an audio file.

SaveAndLoad
This is the most important class in the IO package, since it handles the saving and loading of the graph model. We already explained a bit about this class in the introduction of section 2.4, however since it is a custom class we will explain it in a bit more detail.

The save and load class saves the data of the model to a JSON file. This is done in the method *saveModel()*. Here we save each value of a node (x,y,height,width) in combination with a key. Each value has a unique key. The key for the x value of the first node will be x1 and for the second node x2 and so on. This way we can easily look for these values when loading a model. Loading can be done in two ways; upon startup of the program, we can specify a path for a saved model which will then be immediately loaded; or, we can manually load the file using the FileSelector. If no path is specified we create an empty model.

Selector

This class uses extends JFileChooser and it is used to create a save/load dialog interface for the user when loading or saving models. If we load a model the user will only see files with a .json extension since we implemented a filter. This filter prevents users from loading wrong files. If the user saves a model, he/she only needs to enter the desired file and the extension will be added automatically. Next to this, we also set up a default directory, which leads to the "saved" folder in the project.

2.5 Metadata

In order to make the software as user-friendly as possible, as well as to fulfill all the requirements as stated in the assignment, there were a lot of variables which had to be stored and accessed by all other components of the project. For example, a placeholder for the currently selected node was needed that had to be manipulated by the controller and observed by the view, therefore the only way we could implement this under MVC constrains is to include such data in the model.

However this data is not inherent to the model and stores meta-variables about the interaction of the model and the user. Therefore, we borrowed a few micro-elements of the MVVM model in order to keep this meta-data separate from the model which allowed us to get a cleaner and safer model which did not violate any constrains of the MVC framework.

ViewModel
This is an integral element of our program. It contains all placeholders for information about the current user-controlled processes that could not be included anywhere else without violating the MVC predicates. It is not an implementation of an MVVM approach - it is just a mini-viewmodel that contains important variable which do not belong to the general model.
It consists of properties that represent place holders for things like:

- The currently selected edge/node
- Boolean flags indicating whether the user is in the process of moving/resizing a node
- The last fixed location/size of the node that is currently being moved/resized
- The cursor location
- The currently selected modes and preferences.
- The size of the GraphFrame

All the methods of this class are related to manipulating and retrieving its properties, i.e. all methods are some variations of getters and setters.

ClipBoard
For copying and pasting nodes, we also realized a *ClipBoard* class which we also considered to be a type of meta-data so we placed it here. It consists of a *Node* property and a list of edges which are connected to it. The latter is needed so that when one copies a node, they also copy the edges associated with it.

It also has two static methods - copy and paste. The methods are static as there is no need for passing instances of the clipboard around when the clipboard is shared between all the elements of the program.

ResizeOption
This contains a simple enumerator that indicates if and how a node is resized. It could be resized from either side or corner of the node but the logic behind the different options is slightly different, hence we implemented an enumerator to indicate that.

3 Evaluation of the program

Initially the program seemed fairly simple, since a model is just N amount of nodes and E amount of edges. However, as we got further and further, the program became a lot more complex, especially with all the extras we implemented. We kept the OOP style and the MVC predicates in mind at all times, and it seemed to work well for us. It allowed us to divide the tasks efficiently and it made working with each other's code a lot easier.

As required by the assignment, we implemented the MVC pattern in this program. This pattern however turned out to not be the most efficient or effective style of designing such a Desktop app. Some parts of the program turned out to be almost impossible to do without going against the MVC pattern. With the help of our TA we decided to design a separate class (ViewModel) in order to circumvent this problem. If we would do a project in our free time, we would probably use a different design pattern like MVVM (Model-View-ViewModel). However it was still a very useful experience.

In the end it turned out that the actual coding was not too hard, since the creation of particular sections like drawing nodes or creating buttons was done without many problems. But making them in a way that was efficient and object oriented was hard. There were multiple times where we needed to redo some part, because it turned out to be error prone or not efficient. However in the end we managed to get a reasonably complex working program, which to our knowledge is made in a way that respects the MVC pattern and OOP.

As to all the requirements, the final project worked out great, covering our expectations. We did not exactly anticipate how much time we would have to implement extra functionality, but the execution exceeded our expectations

We tested the program meticulously each time we added new functionality (including undoability, exceptions, hindering the other actions). As far as we are aware, there are no bugs in the program, except for one: the icons and sound files, used in the super mode, do not load on one of our computers because the model is looking for them in the wrong directory. As we implemented the supermode in a safe manner, this does not hinder any other functionality.
As far as we know, the icon/sound files loading is done correctly, searching for them in the root directory, however this root directory seems to be different between our computers. We found no workaround for that, however this should not be an issue for users, since they are going to use an installable or portable version of the program which will set up the root directory environmental variable correctly and consistently.

4 Extension of the program

We added the following extensions to our program:

1. Resizing nodes.

   This enables the user to resize a node. This can be done by pressing and dragging one of the four corners or one of the four sides of a node. The name within the node will also

automatically be resized.
We implemented boundary conditions (e.g. if the user starts resizing from the top left corner, they could not go beyond the top right corner or the bottom left corner).

2. Different themes/modes.

   The second extra we added is different themes/modes. We added a dark mode (which is the default mode) and since almost no one uses a "light mode" we decided to add a nyan cat mode. In this mode all nodes turn into nyan cats, the background and menu bar change and a short nyan cat theme song is played. If the image/audio files are incorrectly loaded, the mode is disabled.

3. Keyboard shortcuts.

   The third extra we added is keyboard shortcuts. Every action is coupled with a keyboard shortcut and these shortcuts are displayed in the buttons and menu items.

4. Copy/paste.

   The fourth extra we added is a copy-paste functionality, which allows the user to copy and paste nodes. We realized this, so that when an user copies a node, it copies all the edges associated with it. Not including the associated edges would defeat the purpose of copy-pasting, since the user could just add another node.

   Pasting is redoable but copying is not. This was so if the user wants to borrow elements from a previous version of the graph, they could iteratively undo until they get to that point, copy an element, redo back to the current state and paste.

5. Directed graphs.

   All graphs created with our editor are directed. The user has the option to see the directions of the edges or not.

   When each edge is added, the program keeps track of its direction. The edge always starts at the pre-selected node and ends in the post-selected one (see Fig.2).

   The user can change whether or not, they want to see the edge directions through the menu "Preferences"

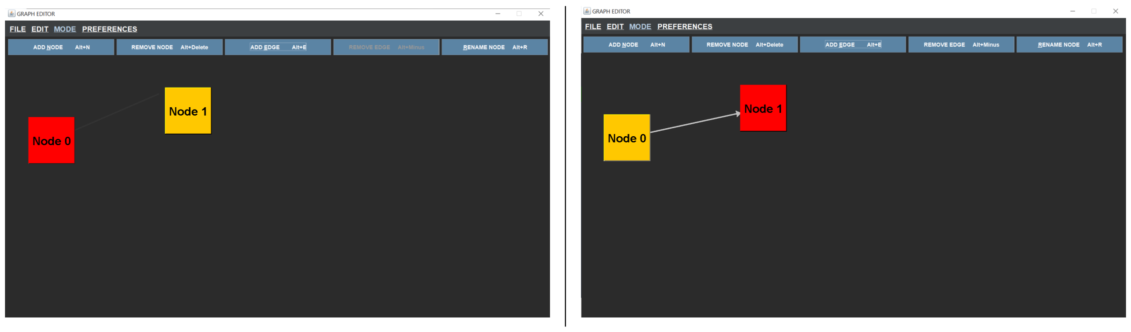


Figure 1: Tracking the direction of each added edge

6. Select edges.

   In order to easily remove edges, we implemented functionality to select them by clicking on them. This turns them green.
   It is important to note that this is not the only way of removing edges. The user could click a node, then click the "Remove edge" button and then click the other node that is connected

to the edge. If there is no edge between the two nodes, the program outputs a message dialog to let the user know of that.

This is important since the latter way of removing edges keeps track of their direction, meaning it only removes nodes that start with the pre-selected node and end in the post-selected one. Our method of selecting edges allows the user to simultaneously delete two edges between the same node if they are in opposite directions (see Fig. 3).

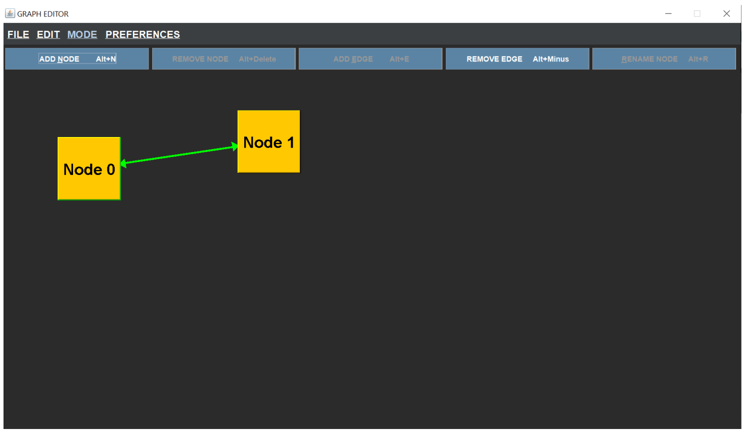
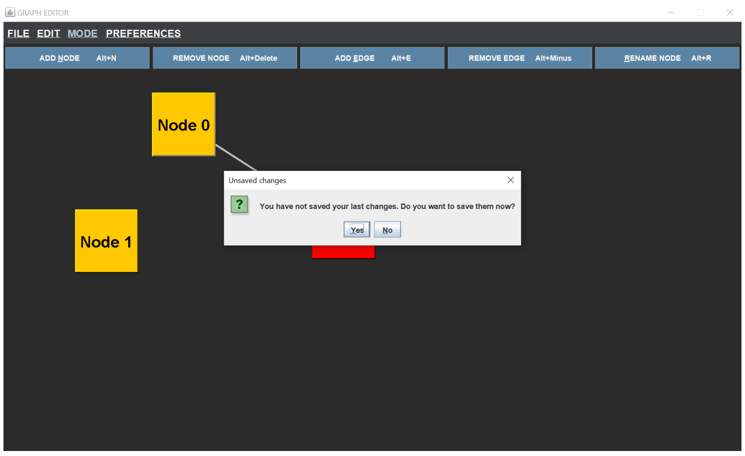


Figure 2: The functionality allows the user to simultaneously select two opposite edges

7. Optional save and save as...

   Whenever an user has already saved their current model into a file, the model remembers where the model is stored. Therefore, the next time the user wants to save a file, they don't need to specify the file location again.

   When the user attempts to create a new graph or load an old one without saving the current one, the program checks if the user has saved their last changes. If not, it asks the reader whether or not to save the last changes (see Fig. 4).



Unsaved changes

You have not saved your last changes. Do you want to save them now?

Figure 3: When a user attempts to close the model without saving it, it is prompted by the program about whether or not to save the last changes

Additionally, if the user wants to save their current model somewhere different than before, they could use the traditional "Save as" function in the File menu.

5 Process evaluation

The whole process was fairly long, since we had about 3 weeks for this assignment. This allowed us to think long about the design of our program and not rush any decisions. The hard part was respecting MVC, since it gave us some problems during the coding process. We learned a lot of things along the way, especially about the design of a GUI. Next to this we also learned how to work together through GitHub, since we both had no partner until this final assignment. We learned that it is very easy to work on one program if we use the object oriented programming style.

We also learned the great importance of unit testing, striving for simplicity and obeying the OOP standards. We thought about our design choices in great detail, as to create a scalable and maintainable code. This made it very easy to us to add functionality and extras without breaking what was already achieved. Now, we understand how important it is, to think of the maintainability of the code in the beginning.

6 Conclusions

We are both pleased with the program we created for this assignment. Especially with all extras we implemented, it now feels like a complete program (given the time frame, and requirements). The code is maintainable as we did not reach any troubles extending the assignment and adding extras. It works properly and due to the object-oriented style it could be extended in many ways.

A fun idea might be to convert this program into a ER/EER diagram creator. Also, we wanted to add a controller for the *MouseMoved* method, so that the cursor changes, when the user is resizing a node. We also could extend this model to actually execute mathematical operations with the graph (search algorithms, decision trees, etc.).
We learned a lot during the labs and without the help of our TA the program would have been a lot less efficient/clean. Proper guidance is, therefore, a huge help in these kinds of projects.