# Object-Oriented Programming
# Programming Report
# Graph Editor

*Rosario Scavo* *Brody Hartman*
*S4243439* *S4079337*

June 23, 2020

## 1 Introduction

We were given the task of building a graph editing program from scratch using Java Swing to make a GUI, and following the model, view, controller (MVC) design pattern. The editor had to be able to create a simple unweighted graph by adding and removing nodes and edges from the graph. These nodes and edges would also have to be drawn on screen for the user to interact with, and also be updated in real time as edits are made. Our code would also have to be able to save the graph to a file and then be able to load a graph again from those files. It would also need to be capable of having undo and redo functionality for any action like deleting the wrong node on accident to not cause a huge issue for the user.

## 2 Program design

As stated before, we needed to have the user input update the GUI in real time, therefore the MVC design pattern was the easiest way to implement real time updating of the screen. The code is split into 3 main packages which we will split up and talk about here.

- **Model**
  The model of this graph editor is the graph itself and where we store the nodes and edges and their locations. First we created a basic node data structure which stores its name, shape, color, and location, and then an edge class that stores 2 nodes as endpoints. These are the building blocks of our main model class called `GraphModel`. The graph model stores all the nodes and edges, and has methods that support creating new nodes and edges between them, as well as safely removing both nodes and edges. It also contains functions to save the current graph to a file and to load a graph from a file, which use two other classes in the Model package that we will talk about next. The last element of the `GraphModel` to mention is the Undo Manager which saves the actions performed by the user in order as undo-able edits. These edits are created as instances of each of our undoable actions classes, from the controller package, which preserve the state of the current graph's data and have their own redo and undo methods.
  We have a separate input output package in our model package that contains the classes `GraphSaver` and `GraphLoader` which handle the saving and loading of graphs and which both use methods from the last class in the input output package called `GraphFileFormat`.The `GraphSaver` uses a `FileWriter` to write all the data from a `GraphFileFormat` to a file specified by the user. The `GraphLoader` uses a `Scanner` to read from files stored using our custom `GraphFileFormat` as .graph files. The `GraphFileFormat` class is instantiated to store all the data about a graph before saving it and can also be created from a file to retrieve that information when loading a graph.

- **View**
  The view of the graph editor is the GUI itself and is made of a custom JFrame that holds our menu bar and tool bar as well as the main graph editing panel. The `MenuBar` contains the file operations to load, save, and reset the graph, as well as the undo and redo options, access to the node color changer, and our select all and deselect all actions. While the `ToolMenu` class contains the buttons to add nodes and edges, remove nodes and edges, and more accessible undo and redo buttons. The `GraphPanel` class is where the actual graph is displayed for the user to see. It also handles drawing the edges between the nodes and

it uses another class from the view called `NodeLabel` which stores the node and visual settings to display each node according to its location, dimension, and background color. The `GraphPanel` and `NodeLabel` classes both implement the property change listener interface which is our method of updating the view. The `NodeLabel` only listens for a change in the name of its node, whereas the `GraphPanel` updates and repaints the graph each time any action is performed that can cause a visual update.

- **Controller**
  The controller package contains all of the actions that can be performed by the user that are accessible from the view. The custom tool bar that we place in the frame contains the various buttons that the user can use to add and remove both edges and nodes. We also made a custom menu bar that contains the file operations for saving and loading graphs, as well as creating a new empty graph and deleting your current one. It also has menu items for editing the graph that allow for undoing and redoing your actions. This editing menu also contains handy shortcuts for selecting all the nodes and deselecting all nodes. This is also where we put the color changing feature which we will explain in the Extensions section.
  The controller contains another package called items which contains all the buttons and menu items that are displayed in the tool bar and menu bar. In the actions package, there is a class for each action that isn't undo-able. These actions are separated from the other actions because they involve changing the file being edited, either by saving the current graph or loading one from a file. Our undo/redo package contains all of our undo-able edits that can be performed and they each correspond to one specific button or menu item from the items package. The undoable actions include adding nodes, adding edges, removing nodes, removing edges, moving nodes, selecting all nodes, deselecting all nodes, renaming nodes, and changing the color of selected nodes.
  The controller also contains our `SelectionController` class. This class is what allows the user to select nodes by clicking on them or move them by dragging them with the mouse. They can also move multiple nodes at once if there are multiple nodes selected. Lastly we have the undo/redo package that contains all of our undo-able edits. Each class in this package has an undo method and redo method which are called by the Undo Manager stored in the graph model. When the user makes a change the action is logged and able to be undone.

- **MVC Design**
  Now that we have described the structure of each package individually, we can explain how they work together and how our code uses the MVC pattern effectively. Starting from the view, the user has many ways of interacting with the program. For example, if they use a button to add a node, the button triggers an action event using its corresponding `AddNodeUndoable` class from the controller package. The action will make an instance of itself and add it to our undo manager with all the data needed to undo the edit or redo it. Then it calls a method from the `GraphModel` class, in the model package, to add a node the the graph. The model will add the node and then it will fire a property change event to the `GraphPanel` which is a property change listener. The view will then be updated by the `propertyChange` method in the `GraphPanel` class. This follows the MVC pattern exactly where the user sees the graph in the view, uses the controller to change something about the model which then updates the view immediately. They are also able to do this loop using the `SelectionController` class, which is just another way to read their inputs with the nodes themselves. Each instance of the `NodeLabel` class creates its own `SelectionController` object to track when it has been selected and moved, and both of those controller actions also follow the same MVC loop by editing the model which then updates the view.

- **Design Considerations**
  - The first decision to mention is that we are using the PropertyChangeListener interface instead of the Observer interface. This is for simple reasons, as both classes accomplish a very similar effect. The first is that we can pass variables in the new value and old value fields of a property change event when we make a new one. The second is because we can also give a name for the event which can then be read after it has been fired. We use both of these features when updating our `GraphPanel` class and observable does not have those same functionalities.
  - The second decision is why we chose to make the `SelectionController` class linked to each `NodeLabel` object instead of having the controller connected to the `GraphPanel`. There was a consideration to make the selection controller a part of the panel so that it would track your mouse and be able to check all the nodes for their coordinates. It could then find which node your mouse would be intersecting with when you try to select a node or drag one. This however is not as efficient as having the controller on the `NodeLabel`. First of all, the function must loop through all of the nodes and their coordinates and then do a comparison with the mouse coordinates, and each time the screen is updated

it must run this loop again. This can also lead to bugs where you might suddenly switch which node you are moving if your mouse moves over another node that is earlier in the loop than your current moving node, causing your next `MouseMotionEvent` to move a different node. This problem is completely avoided when putting the controller on each `NodeLabel`, as each controller object only has to select or move one specific node and requires no checks or logic.

# 3 Evaluation of the program

After completing the final features we ended up with a very functional program. It is very simple to use and has many functionalities that are helpful for the user to make visually pleasing graphs. We had progressively been bug testing the software as we wrote it to make sure it remained functional throughout the many changes we implemented, and after its completion we were able to test everything again but with full functionality.

# 4 Extension of the program

In our graph editor there are some small visual enhancements that we implemented. First of all, we made the graph a directed graph by adding arrows which show the relation between the two nodes. It is drawn in the middle of the edge and points from the first end point to the second endpoint of the edge. Another visual enhancement we added was the ability to color nodes. You can select any number of nodes and use the menu options to change their colors all at once. This action is also fully undo-able and it uses the built in java swing color chooser dialog. We also made many of the menu items and button actions more accessible with keyboard shortcuts. The last extension is a very simple group move option, where the user can select a group of nodes and then move all of them at once. This is a very minor extension but a handy one for actually making larger graphs.

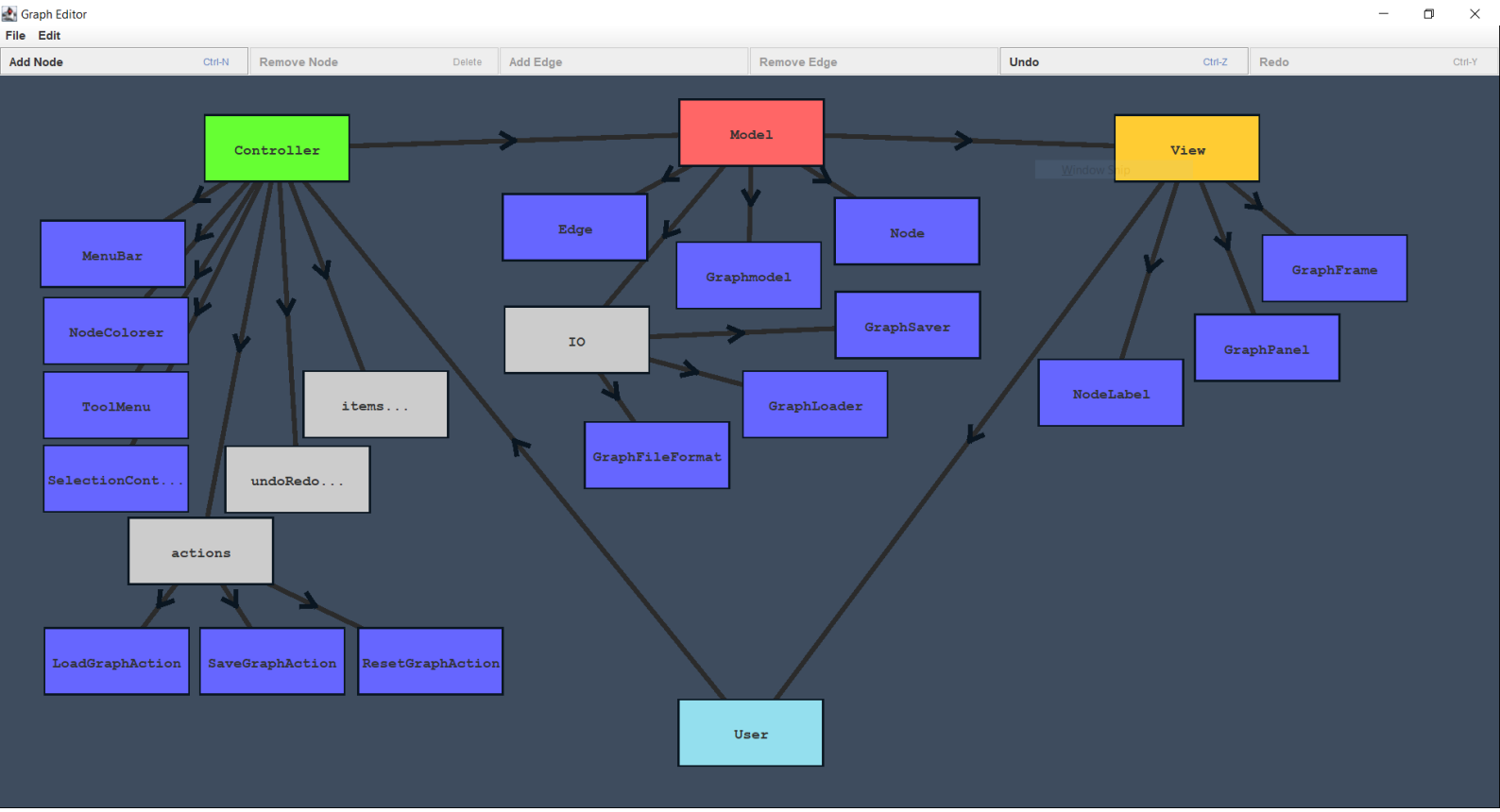
# 5 Process evaluation

When writing the code for our graph editor we had the work mostly split into the model and file actions for one of us, and the view plus the buttons and menus and their actions for the other. This split was working quite well for most of the code, however one of us had issues when trying to implement adding an edge. The requirement asked that we make the edge follow our cursor until we click a new node to end the edge, but he couldn't come up with an effective way to do that. The other partner came up with the idea to modify the selection controller class to make the `mouseClicked` method work to select nodes, rename nodes, and end the edge when you're making a new one. All three actions work differently and only trigger off of different events to remain distinct. This assignment was a very large project involving many different classes with relations between them. It required a lot of effort to make sure you have accounted for each new feature and that they don't interfere with each other; much more than the other projects from this class. But it really helped show why the MVC design pattern is so commonly used, as it definitely made coding it much simpler to conceptualise.

# 6 Conclusions

Our final code for the graph editor fulfills all the requirements and adds more to the editor to make it work the way we wanted it to. The code itself is quite maintainable as we used many design patterns that are abstract enough to allow for new implementations. It is very functional and has no known bugs, and can be used quite easily to make a graph quickly. There are some ideas for possible upgrades such as resizing nodes, or allowing the edges to be shaped so the user can curve them around other nodes and edges. These are purely visual enhancements though and are not needed to make a graph properly. A useful upgrade would be to have an option to add weights to the graph and possibly add more functionality for the weights, such as a tool that calculates the shortest path between 2 nodes. And finally to prove that it is a useful tool we have used our graph editor to map the class hierarchy and the MVC design pattern that we utilized to make this program into a relational graph.(*Fig. 1*)

*Fig. 1*


Graph Editor
File
Edit
Add Node
Ctrl-N
Remove Node
Delete
Add Edge
Remove Edge
Undo
Ctrl-Z
Redo
Ctrl-Y
Controller
Model
View
MenuBar
Edge
Node
Graphmodel
GraphFrame
NodeColorer
IO
GraphSaver
GraphPanel
ToolMenu
items...
GraphLoader
NodeLabel
SelectionCont...
undoRedo...
GraphFileFormat
actions
LoadGraphAction
SaveGraphAction
ResetGraphAction
User