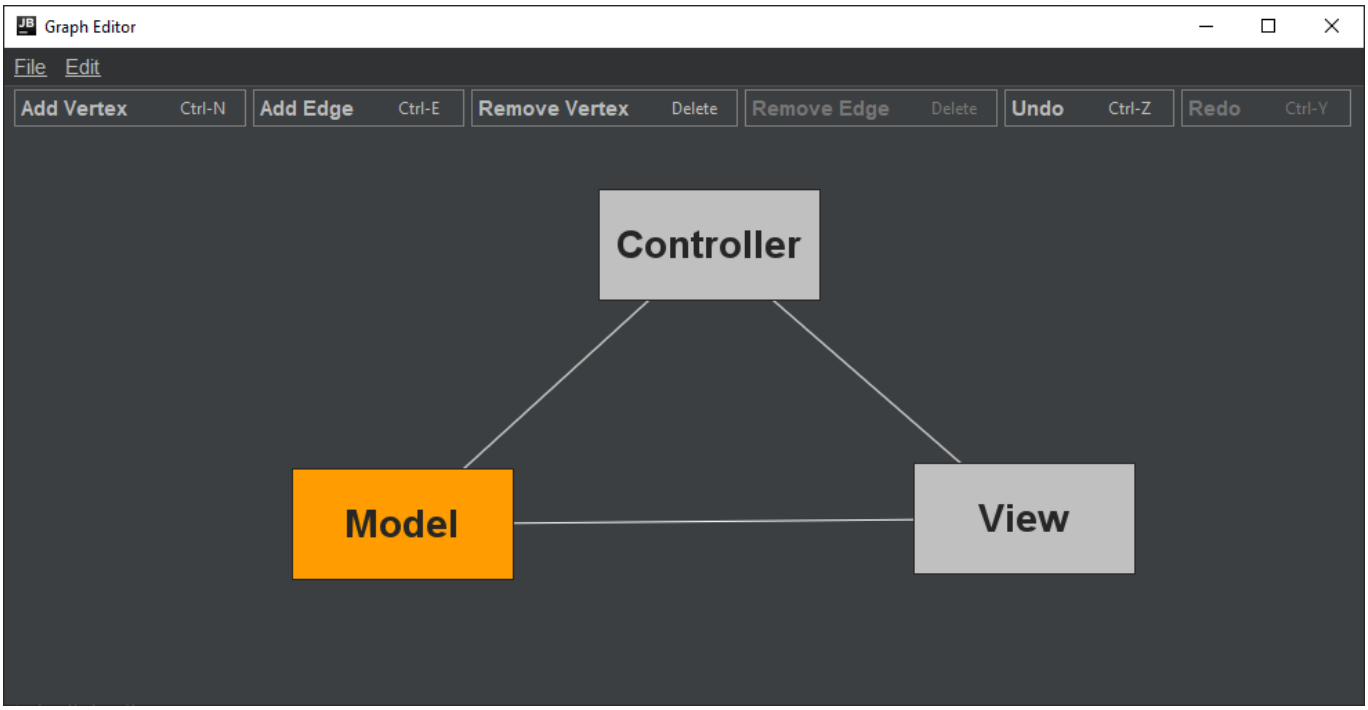
Graph Editor



In the epic conclusion of the OOP cinematic universe, you will be creating a graph editor with a graphical user interface. This final assignment will test all the skills you have gathered in the past few weeks, such as Swing, MVC, Java IO, and the ability to complete a reasonably complicated project. Just as the previous assignment, the GUI must be implemented using Swing and the MVC pattern. Note that the image above is just an example; feel free to design the UI in any way you want (while adhering to the requirements).

The assignment is split into two parts. In the first parts you will be creating the model of the graph along with saving and loading. The second part will be about the graphical user interface and user interaction. For this assignment you will also have to write a report.

There will be two deadlines:

- **Intermediate Deadline**

  Part 1 should be completed. It is **mandatory** to create a pull request for this, but the corresponding demo is optional. This is not graded, but if you do not create a pull request for the intermediate deadline, you will receive a penalty for the assignment.

- **Final Deadline**

  Part 1 and 2 should be completed in addition to the report.

Part 1

1.1 The Graph

The model part of this exercise is represented by an undirected, simple, unweighted graph without self loops. We will briefly walk through the steps to implement this below.

1. Start with creating an empty class `GraphModel` which will be the class to represent a graph.
2. Create a class that represents a node.
    - A node will be represented by a rectangle that has a size and a location. To do this, you can use java.awt.Rectangle.
    - A node should have a name.
    - Every node should have a default constructor that creates a node with a default name, size and location.
3. Create a class that represents an edge.
    - An edge always connects two nodes
4. The `GraphModel` should keep track of all the nodes and edges.
    - Add methods for adding and removing nodes and edges. Remember that for removing a node, every connected edge should also be removed.
5. Instead of using a class `Main.java` to start your application, use a class called `GraphEditor.java` for the main method.

1.2 Saving and Loading

Now that we have a basic graph, let's take a look at saving and loading. Saving and loading will be done in a custom format, an example of which is provided below:

- The first line contains two numbers *N* and *E*, where *N* is the number of nodes and *E* is the number of edges.
- *N* lines describing the nodes. A node is described by:
    - Two numbers for its location: *x* and *y*
    - Two numbers for its size: height and width
    - The name of the node
- *E* lines describing the edges. An edge is described by the indices of the two nodes that the edge is connected to.

An example of this could be:

```
3 2
0 0 30 40 one
100 100 30 30 two
50 200 80 20 three
0 1
1 2
```

6. Create a method that saves the graph to a file in the format described above.
    - Think of a nice file extension name for your save files.

- Allow the user to choose where the file will be saved. You can use JFileChooser for this.

7. Create a method that loads a graph from a file (said file should adhere to the format above).
   - You can use `JFileChooser` again to let the user pick a file.
   - Only allow the user to see the files with the same file extension as the save file.
8. Add a constructor to `GraphModel` that uses the load method to immediately load a graph from file.
9. When you run the program from the command-line using `java GraphEditor filepath`, the graph stored in `filepath` should be loaded into the program.

Part 2

Now that we have the basic model, we can start working on the GUI. ٩(◕‿◕)۶

2.1 A Basic Frame

10. Start by creating a class that inherits from `JFrame`. This will be the main frame of the graph editor.
11. Give the frame an appropriate title and default size.
12. Create a panel class that inherits from `JPanel`. This is where all the drawing of the nodes and edges will take place.
13. Make sure that this panel is visible in the main frame.
14. Give the frame a basic menu bar with buttons for adding and removing nodes and edges. Do not add actions to these buttons just yet (just pass `null` as the required action).

2.2 Painting

16. Make sure that the panel has access to the information it needs for drawing the graph.
17. Create a method that draws all the nodes. A node is drawn by first drawing a rectangle and then the name.
18. Create a method that draws all the edges. Keep in mind that everything that is drawn first ends up behind something that is drawn later.

2.3 Selecting Nodes

19. Allow the user to select nodes by implementing a class `SelectionController` that extends MouseAdapter. A node should be selected when the user clicks on it.
20. Selected nodes should be highlighted in some way (e.g. different color), so that the user knows whether a node is selected or not.
21. Allow the user to drag nodes using the same `SelectionController` class.
22. When something in the graph changes, the panel should be updated. Implement the `Observer` pattern to ensure that whenever something changes in the graph, the panel is redrawn.

2.4 UI Functionality

23. Now that we are able to select nodes, we can add the functionality for the buttons:
    - Allow the user to add a node.
    - Allow the user to remove a node when a node is selected.

    - Allow the user to add an edge when a node is selected. When the user wants to add an edge, a line should appear from the selected node to the cursor. The edge will only be made when another node is selected.
    - Allow the user to remove an edge.

24. Ensure that the buttons are disabled when the pre-conditions are not met (for example, the user did not select a node, so the button for removing a node is not enabled).

25. Allow the user to change the name of a node.

26. Allow the user to create a new empty graph.

27. Add buttons to the menu that allow the user to save/load a graph.

2.5 Undo and Redo

Basically everything has undo and redo functionality these days, so let's add this to our program. Since we need to remember the actions, we need to represent all the operations we execute as classes. For this we can use the AbstractUndoableEdit class. Any class that inherits this is forced to implement the `undo()` and `redo()` methods. `redo()` should simply contain the code for the operation, while `undo()` contains the code that undoes the operation. Java has a built-in manager for these `UndoableEdits` called UndoManager.

28. For every operation, create a class that extends `AbstractUndoableEdit`.
29. Add an `UndoManager` as a field to the `GraphModel`.
30. Whenever the user clicks on a button, the corresponding `AbstractAction` should not perform the action itself, but rather create a new instance of the corresponding `UndoableEdit`. You can then execute the operation by calling `redo()`.
31. After executing the operation, add it to the `UndoManager` using the `addEdit()` method.
32. Make sure that Undo and Redo can be used from the menu. These should call the `undo()` and `redo()` methods from the `UndoManager`. This will make sure that the appropriate action is being executed and you do not have to keep track of all these operations yourself.

Requirements Summary

Below is a list of all the things your program should be able to do. Note that you should still pay attention to the steps described above! This only serves as a very compressed summary.

Your program should be able to:

- draw a graph
- create a new empty graph
- save and load graphs
- load a graph from the path provided in a command-line argument
- add nodes
- remove nodes
- add edges
- remove edges
- rename nodes
- select and move nodes
- undo the following actions: adding nodes/edges, removing nodes/edges, renaming and moving

- redo the following actions: adding nodes/edges, removing nodes/edges, renaming and moving

---

Report

For this assignment you will have to write a report in which you will describe your program and the corresponding design decisions. We do not expect a 10 page, full-on architecture document; just be sure to discuss everything mentioned in the template, which will probably result in ~3-4 pages. A LaTeX template containing all the sections and explanations for each section can be found on Nestor. Once finished, add the pdf of the report to the `assignment_3` directory.

Extra

Now that the basic graph editor is done, you can finally add some extras! As with the previous assignments, these are not mandatory, but can give you some bonus points. Examples could be:

- Keyboard shortcuts such as `Ctrl + n`, `Ctrl + z`, `Ctrl + y` etc.
- Copy and pasting nodes
- Make the interface as nice as possible. Add icons to buttons , add extra menus, dark mode etc.
- Directed graphs
- Allow the user to select a shape for a Node
- Custom colors for nodes
- Resizing nodes
- Allow the user to select edges for easy deletion

If you decide to add extra actions, do not forget to add undo/redo for those to. Any action for which it makes sense to undo/redo should have it.

Handing in

Handing in will go as usual: create a pull request from `development` into `master`. Again, you can ignore the CircleCI output for now; just make sure the code compiles and runs. Do not forget to include the report!