SE450 FINAL PROJECT REPORT

LIST OF FEATURES

My application is fully functional per the amended requirements in the instructions. I largely maintained the original structure of the application state and the controller lambdas as there seemed to be a good flow built in from event action to pattern response and change of state. Though, throughout the process I was tempted many times to completely diverge.

- CHOOSE SHAPE, CHOOSE PRIMARY COLOR, CHOOSE SECONDARY COLOR, CHOOSE SHADING TYPE, CHOOSE START/ENDPOINT MODE all function as expected, build the requisite shape, and store in ShapeList.
- UNDO/REDO implementations are built out for DRAW, MOVE, COPY, PASTE, DELETE
- I also included a CLEAR button and functionality, though I didn't implement UNDO/REDO for this. It basically sets fresh from the beginning which helped me test my other features.
- While I periodically debugged and tested my code, my console print-outs also provide a very clear set of steps along the way, and if I had more time I could see how you would easily be able to swap out the GUI for CUI.

NOTES ON DESIGN

1. My Gui**Observer class** was perhaps the most crucial design element that I chose to introduce. As a way to ensure that the MVC archetype was followed, I instantiated a single Observer in my PaintCanvas and passed it to the MouseListener class nested within my PaintCanvas. This is why I called it GuiObserver as opposed to CanvasObserver.

   While in many typical examples of observer patterns there may be multiple instantiations and many update/create/deletes to concrete observers, I found that in this use case we could get away with just a single concretion that lived in my Controller architecture as a constant middle man. In reality, what I call Observer here may have ended up becoming more of a concrete Proxy object than anything (particularly as I didn't implement an Observer interface or hold a buffer for a single update() function) but I stuck with the Observer description in the end.

   This observer/proxy is crucial in playing the pivot between real-time events in my view, as updated through the event listeners, and implementations of the data stored in my Model.

   This GuiObserver is responsible for basically all of the intermediary storage of concretions such as ShapeList, ShapeFactory, allSelectedShapes, and Clipboard. As a result, View knows nothing about Model and vice versa. Instead, View events generate calls to GuiObserver which holds and makes changes to the appropriate data structures while pulling up the blueprints/data for objects (individual shapes, colors, shading etc.) and logs a history of changes (Command History) in the Model.

   This methodology also allowed me to maintain the lambda function controller architecture that already existed in setup().

2. My Model holds all of the individual implementations for Ellipse, Rectangle and Triangle. Instantiation is determined in the Shape**Factory** using a simple switch statement. Each Shape implements the IShape interface and has its own methodology for creation, update and delete.

3. Creation of new objects, as orchestrated by my GuiObserver, is handled by a simple **Builder** pattern which used the ShapeFactory to instantiate the new IShape and setter methods to assign dimensions, color and other options.

4. As recommended, I integrated a simple **Adapter** pattern into my ShapeColor class to convert ShapeColor to Color from awt.

5. ShapeList & Clipboard implement **Iterable** which makes it easy for my GuiObserver to take changes from the UI and amend the standing ShapeList or individual Shape characteristics by iterating through and comparing. I would then clear the existing view and rebuild the new shapelist.

6. The last pattern I implemented was a version of the **Command** pattern which I linked to the provided CommandHistory logic. Using an ICommand interface, I created run(), undo() and redo() implementations for Draw, Move, Delete, Copy & Paste actions. For Draw & Move, I instantiate new Command objects for the CommandHistory stack in my MouseListener class. For Delete, Copy & Paste, I instantiate my Command object in my JPaintController lambda functions.

   For my Copy & Paste commands specifically, I created a Clipboard class, instantiated as needed and passed to each individual Command to hold a snapshot of the ShapeList at the time the Command was created.

   I use this clipboard to adjust the current ShapeList when dealing with one of these commands.

   I could see the benefit of decoupling the MouseListener from observer entirely, particularly for integration of CUI. In the end, I separated these as much as I could think to in the time given.

SUCCESSES & FAILURES

- It took a LONG time to fully digest the available tools and framework we were given initially. The controller lambda functions specifically were a total mystery to me until I really took the time to step through the entire process in debug. A lot of this had to do with a lack of familiarity with the documentation of Swing and AWT. I also created a complete UML diagram of the original project specs which helped familiarize me with the initial structure. (included in .zip)

- Once I had my mouse controls installed on my PaintCanvas and could create shapes using the dialog mechanics, the next huge hurtle was clearing the current view. I finally read through enough StackOverflow to figure out how to clearRect(), which in and of itself is a bit of step-around. You'll notice that I was unable to get the background color to stay WHITE through each step. I opted instead to operate on the translucent background of the graphics object.
- This means that I clear the board before every operation and rebuild the shapelist as stored in my observer. While this doesn't seem totally efficient, after a ton of research I noticed that this was the recommended strategy by basically everyone.
- Triangle creation took a lot of tinkering. My first rendition created a flexible 3-point calculated triangle which had a LOT of extra code to calculate midpoint(). While this was a more customizable approach, there were many triangles which would look funky or were barely there because of the nature of the calculation. Instead, I went with your recommendation to create bounded right-triangles. Even as it stands, though my triangles are built perfectly, they still flip on the Y axis when moved.
- BoundedBox is my solution to selection mechanics and actually works pretty well. The one area that I could optimize is when you undo/redo MOVE shapes. I calculated all moves by start and endpoint of the shape and so when you move/undo/redo move it offsets the original position by the part of the boundBox which you selected in the current scenario. Instead I could implement a centerOfShape() logic and make my movements correspond to this single precise point.
- A big success of this project is that I recognize that this is just one of many ways to implement the features. My biggest change to the original design was the Observer/proxy object I placed at the head of my controller while I know from research that Swing has a "double buffer" functionality which basically serves the purpose of holding the intermediate steps. I think my class does a great job of standing in for this while granting me some additional flexibility.