Nick Naber

SE 450

Final Project Report

**List of Features:**

The list of required features has largely been implemented with discrepancies noted below.

* **Functional:**

Pick a shape, Pick a primary/secondary color, click and drag to draw a shape, select shading type, select shapes, click and drag select shapes (bounding box), deselect on blank canvas click, copy, paste, delete shape, undo/redo (Draw,Paste,Delete), move shape by click and drag.

* **Extra Credit:**

Being able to change color/shading of selected shapes.

* **Limited Functionality Notes:**

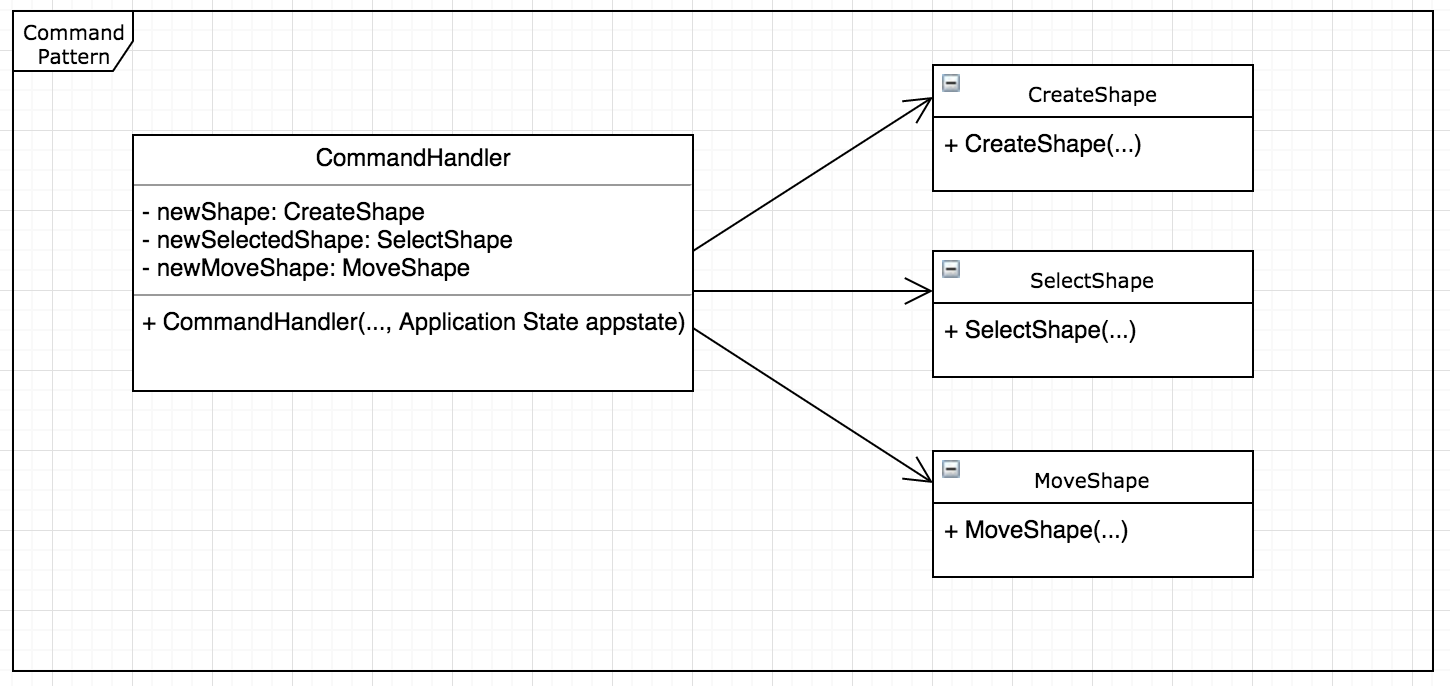
1. Move: when multiple shapes are selected they will all move to the same point when the new location is selected on mouse release.
2. Undo/Redo: does not undo the move command.
3. Paste: shape objects are pasted directly to the right of the selected object.

**Notes on Design:**

Four different major design patterns were used in the project, with one used multiple times.

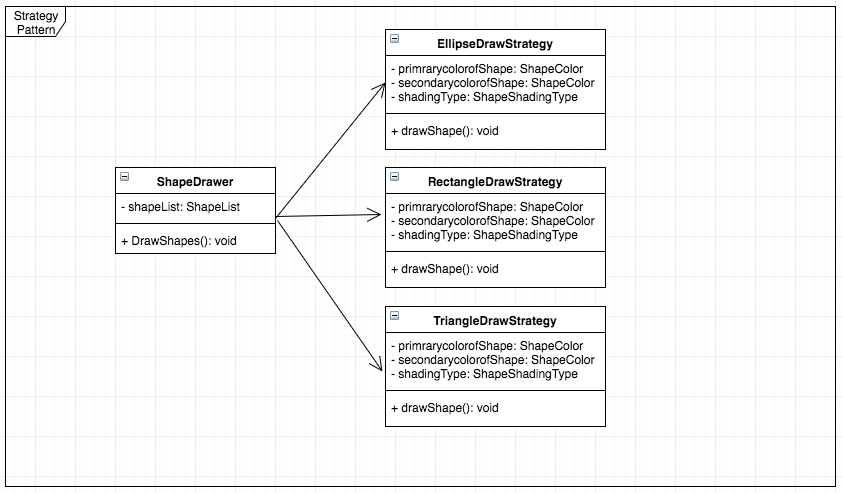
* **Command Pattern:**

The command pattern was used to structure the relationship between my CommandHandler and the Draw, Select, and Move commands. The CommandHandler gathers the mouse data and application state from the ClickDragHandler and passes it to the individual commands which execute Draw, Select, or Move receiver classes depending on the application state. The CommandHandler acts as the invoker in this case and has no knowledge of how the actions are carried out. If the application state indicates Draw in CommandHandler, the CreateShape command will create a new Shape object, add it to the singleton ShapeList, and start ShapeDrawer.

****

* **Strategy Pattern:**

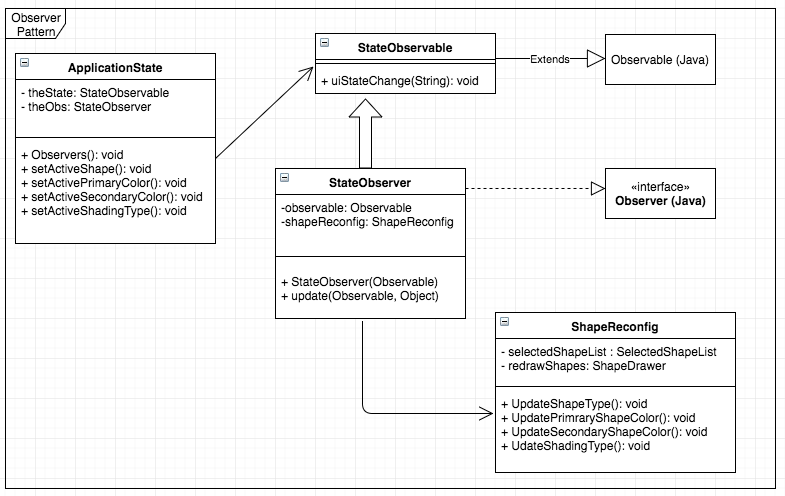
The strategy pattern was used to structure the specific shape drawing classes called in ShapeDrawer. The master ShapeList of Shape objects runs through a for loop and a Shape drawing strategy is selected based on the what the Shape object is reporting as its ShapeType. There is a Shape draw strategy for each of the three ShapeTypes and a background one to clear the canvas. This pattern allowed me to have the different algorithms needed to draw each ShapeType and keep them isolated from the main ShapeDrawer class. I could change the RectangleDrawStrategy without affecting the other shape drawing algorithms or ShapeDrawer.

****

* **Observer Pattern:**

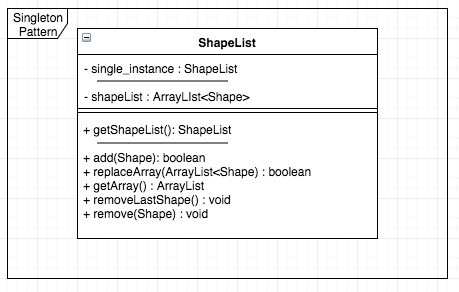
The observer pattern enabled the implementation of changing ShapeType, color, or shading of selected Shape objects. A method was added to the ApplicationState class that started a new observable class (StateObserverable) and added an observer (State Observer) to it. A line was also added to each set method (ex. setActiveShape) to pass the type of state change to the observable class (theState variable). The method uiStateChange in the observable class StateObserverable notifies the StateObservers that were added, in this case ‘theObs’ initialized in ApplicationState.

StateObserver takes the object passed to all subscribers, a String with the type of change, and executes a specific method in the ShapeReconfig class. For example, if the argument passed is “ShapeChange” the StateObserver will run the UpdateShapeType method in ShapeReconfig. This method will update the ShapeType in the selected Shape object and redraw the ShapeList. (diagram on next page)

****

* **Singleton Pattern:**

The singleton pattern was used for the lists that required persistence throughout the program. This solved the problem of having multiple instances of a master shape list or selected shape list. An example of one of these classes, ShapeLIst, is below. Methods were added to add, get the class, remove the last shape, remove a particular shape, and replace the entire ArrayList inside the class.



**Successes and Failures:**

The success of the project is the vast majority of the project requirements are implemented using some of the design patterns discussed in class. There are some elements, like interfaces, that I had a more complete understanding of closer to the end of class and wish I had used more throughout the project. It wasn’t until implementing Undo/Redo at the end that I really had a handle on them and would go back and use (iApplicationState) or create (iListManager,iCommandHandler) several more if given more time.

I was not sure where to put some of the classes in the folder structure and their locations might not make sense. If I did the project again I would try to create the overall structure of classes before starting that matched the required functionality. A minor UI design issue arose when I added Delete and the button did not show up on the menu above the canvas. I had to manually increase the horizontal resolution of the window in order to see the button on the right.

Expounding on the limited functionality notes at the beginning, there are a couple of items I wanted to get in a better state and some ideas on how they could be implemented. When using the Move function on multiple shapes I wanted each shape to retain its location in relation to the other shape objects, but they all snap to the same location on mouse release. I imagine you have to treat all the selected shapes as if they were in one bounding box and have the Move function move that entire box. On mouse release Shape objects on the SelectedShapes list would have their relative location data updated followed by a redraw.

Another Move issue I would have liked to resolve was getting Undo/Redo working for that function. Undoing Draw, Paste, and Delete were not difficult conceptually, and I believe I could made Move work with a stack in each Shape object storing previous location data.

Finally, I wanted to be able to select a location for a pasted shape with the mouse. It was not a requirement, but seemed like more relevant functionality. I tried accessing the mouse handler again, but it just wasn’t working in the manner intended and I abandoned it in favor of the pasted shape just drawing to the right of each selection.