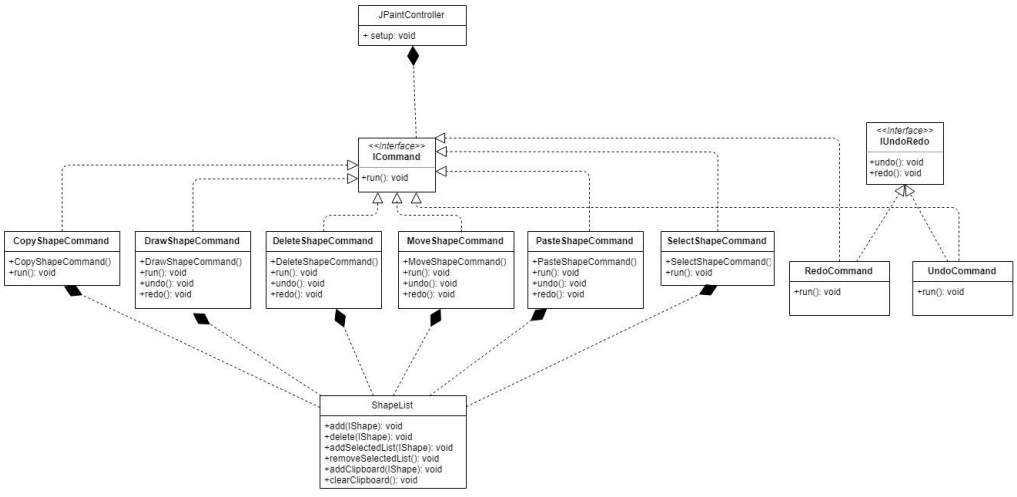
Final Project Reports

Edward Lin #1942366

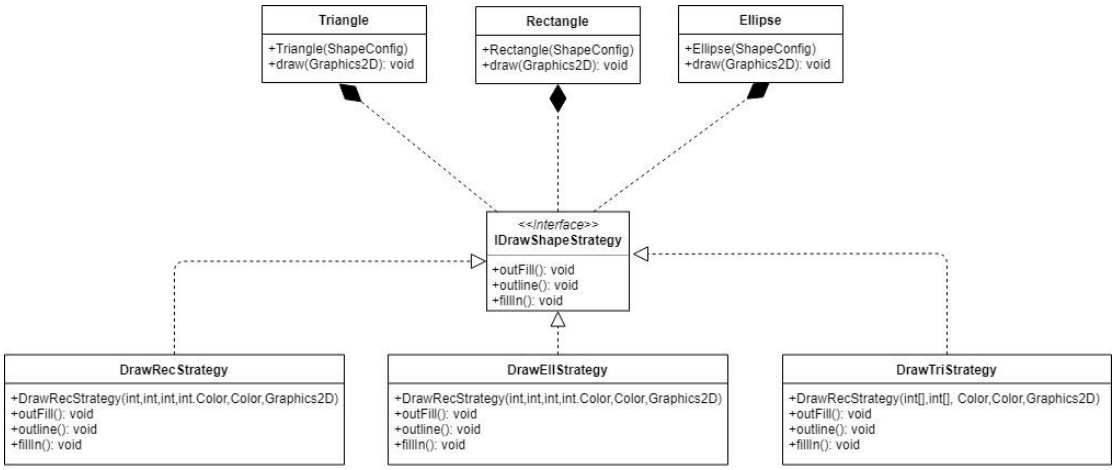
Design Pattern:

1. **Command Pattern:**
   - **Interface involved:** ICommand
   - **Classes involved:**
     DrawShapeCommand, SelectShapeCommand, MoveShapeCommand, CopyShapeCommand, PasteShapeCommand, DeleteShapeCommand
   - **Why I chose to implement this pattern? What problem it solved?**
     The main reason I implement command pattern is that Command pattern is good for recording steps what users have done and then allow users undo or redo action. Furthermore, it makes our code extensible as we can add new commands without changing existing code. The command pattern encapsulate method call about the command and exposing only the call to execute the command.
   - **Realization:**
     All command classes include a run method with the functionality of the command. When Mouse is released, the mouseObservers(Move, Select, Draw) will create a specific command dynamically and call the run() method.
   - **UML diagram (including association):**



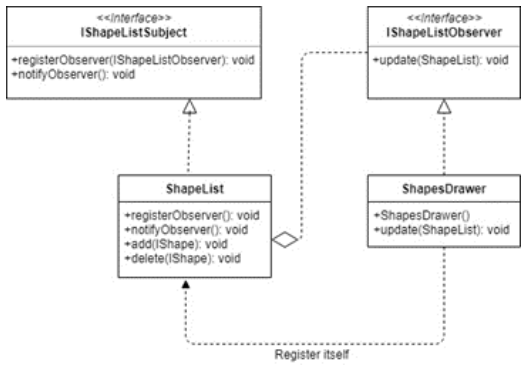
2. **Strategy Pattern:**

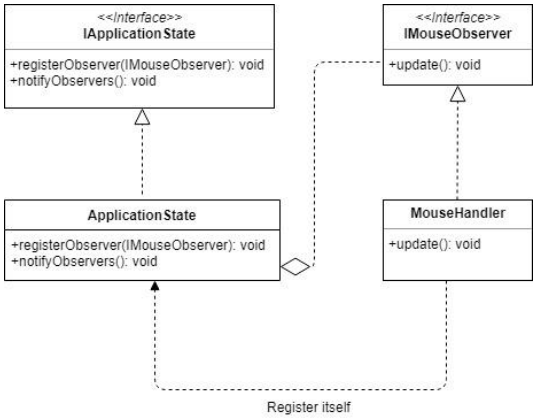
- **Interface involved:**
  IDrawShapeStrategy
- **Classes involved:**
  DrawRecStrategy, DrawTriStrategy, DrawEllStrategy, Rectangle, Triangle, Ellipse
- **Why I chose to implement this pattern? What problem it solved?**
  The main reason I implement strategy pattern is that by encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced. According to the reason, I can have more shapes class without modify the existing code, it follows **Open Closed principle** and allows for my code to be more maintainable, and extensible.
- **Realization:**
  The common behavior is draw in each shape class. When ShapesDrawer call the draw method of one of shapes, it initializes the relative shape strategy and makes a decision about which method of strategy to use.
- **UML diagram (including association):**



3,4. **Observer Pattern(used twice):**

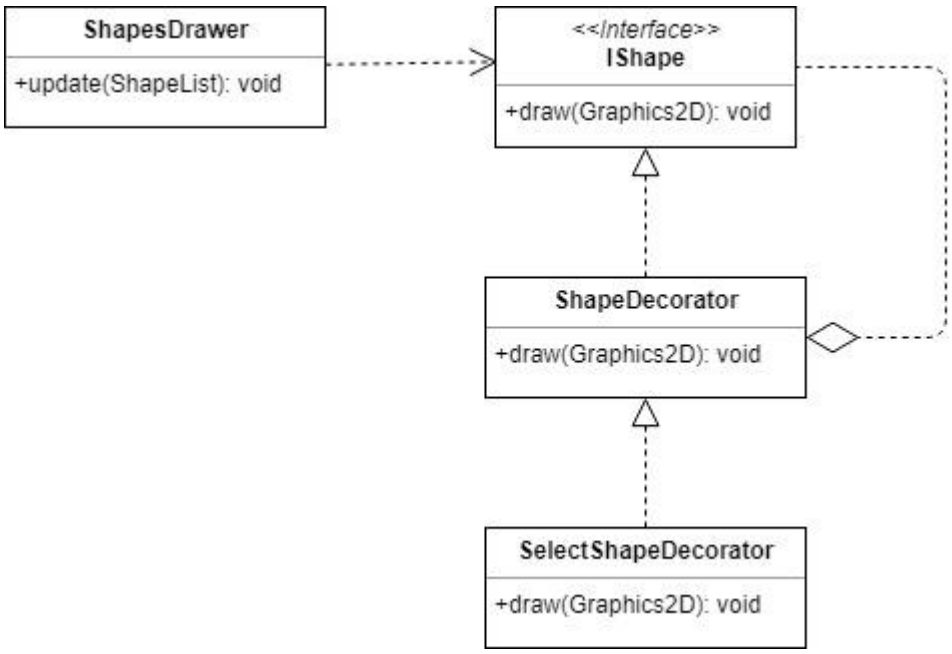
- **Interface involved:**
  IShapeListSubject, IShapeListObserver, IMouseObserver
- **Classes involved:**
  First: {ApplicationState(Subject), MouseHandler, MouseDrawObserver, MouseSelectObserver, MouseMoveObserver}
  Second: {ShapeList(Subject), ShapesDrawer(Observer)}
- **Why I chose to implement this pattern? What problem it solved?**
  The main reason I implement observer pattern is that observer pattern is very useful when we need to notify multiple objects when the select lists are chosen. The Observer pattern allows dependent objects to be notified automatically when changes are made.
- **Realization:**
  When users select one of the modes, ApplicationState would notify mouseHandler and mouseHandler will decide which Observer to execute; In ShapeList, when a shape is added into or delete from list, they both notify the registered observers to do functionality.
- **UML diagram (including association):**





5. **Decorator Pattern:**

- **Interface involved:** IShape
- **Classes involved:**
  ShapeDecorator, SelectShapeDecorator
- **Why I chose to implement this pattern? What problem it solved?**
  The main reason I implement Decorator pattern is that decorator pattern allows me to add new functionality to an existing object without altering its structure. This design pattern acts as a wrapper to existing class.
- **Realization:**
  I created an abstract decorator class ShapeDecorator implementing IShape interface and having IShape object and other necessary parameters as its instance variable.
- **UML diagram (including association):**



Successes and Failures:

First of all, I'd like to say this project was the most challenging assignment I have had since I study at DePaul and I struggled a lot when working on this project, but just like the old saying, "what doesn't kill you makes you stronger", I learned a lot after all this hard working and the pain was worth it. My project may not be perfect, but I think there are still something went right with it. This course improved my skill of programing in Java and design pattern very well. for example, I successfully applied different design patterns on my project: command pattern, strategy pattern etc. The patterns mentioned above are my favorite patterns, they help me to write code in elegant and smarter way and I think that's the most important thing for a programmer. When my existing design of codes can't meet the requirement of new Sprints and I need to make a decision to refactor the codes, that's the moment I love and hate. In first time, I want to apply observer pattern on ShapeList to ShapeDrawer, which means every time I add a shape into shape list, Shape List would trigger the shape Drawer (strategy pattern) so that Drawer can draw shapes every time I add a shape into list. But there is a question, in this scenario, how do I move shape (move shape would erase old shape and add new shape to shapelist)? Because I can't make a big white rectangle when I want to move one of shapes or many of shapes, it would make all shapes I drown disappeared, therefore I need a drawer, a drawer which would clean the screen and draw every shape in my list so that the shapes I drawn won't disappear and the shapes I want to move can be moved to the new position. I'm glad I made my codes work by making these decisions. Despite all efforts I've done, there is something I can't figure out, which is group and ungroup. Due to the period of course is about to end, I can't figure it out before the deadline, but I would really want to keep searching and trying to apply design pattern on this feature after the end of course.

List of missing features:

Group and ungroup

Bugs:

There are two minor bugs, which I would like to fix in an upcoming weeks. The selection outline of Triangle doesn't move with the Triangle while the selection outline of Oval and Rectangle work perfectly. When Pasting copied shapes multiple times, the pasted shapes will all pasted in the same position