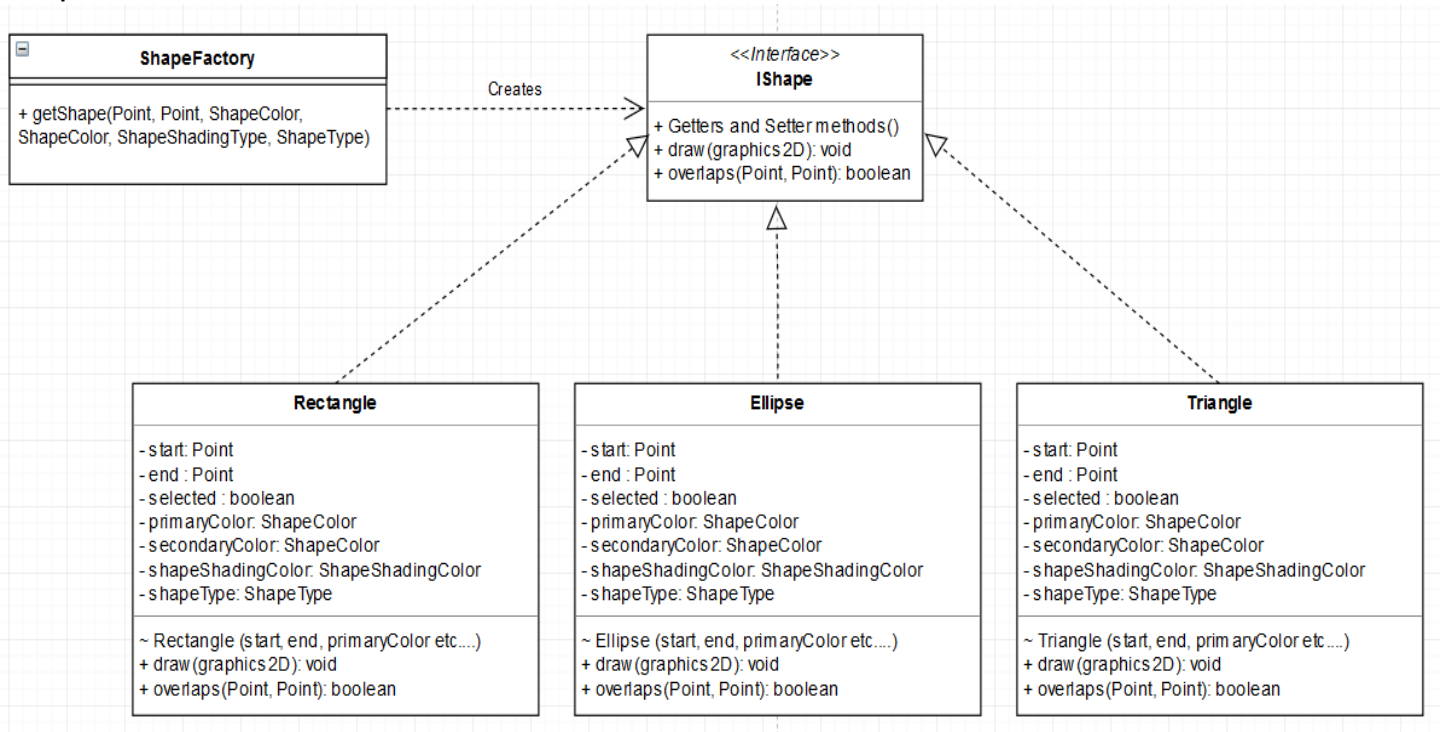
Howard Zhang
CSC 435
March 10, 2018

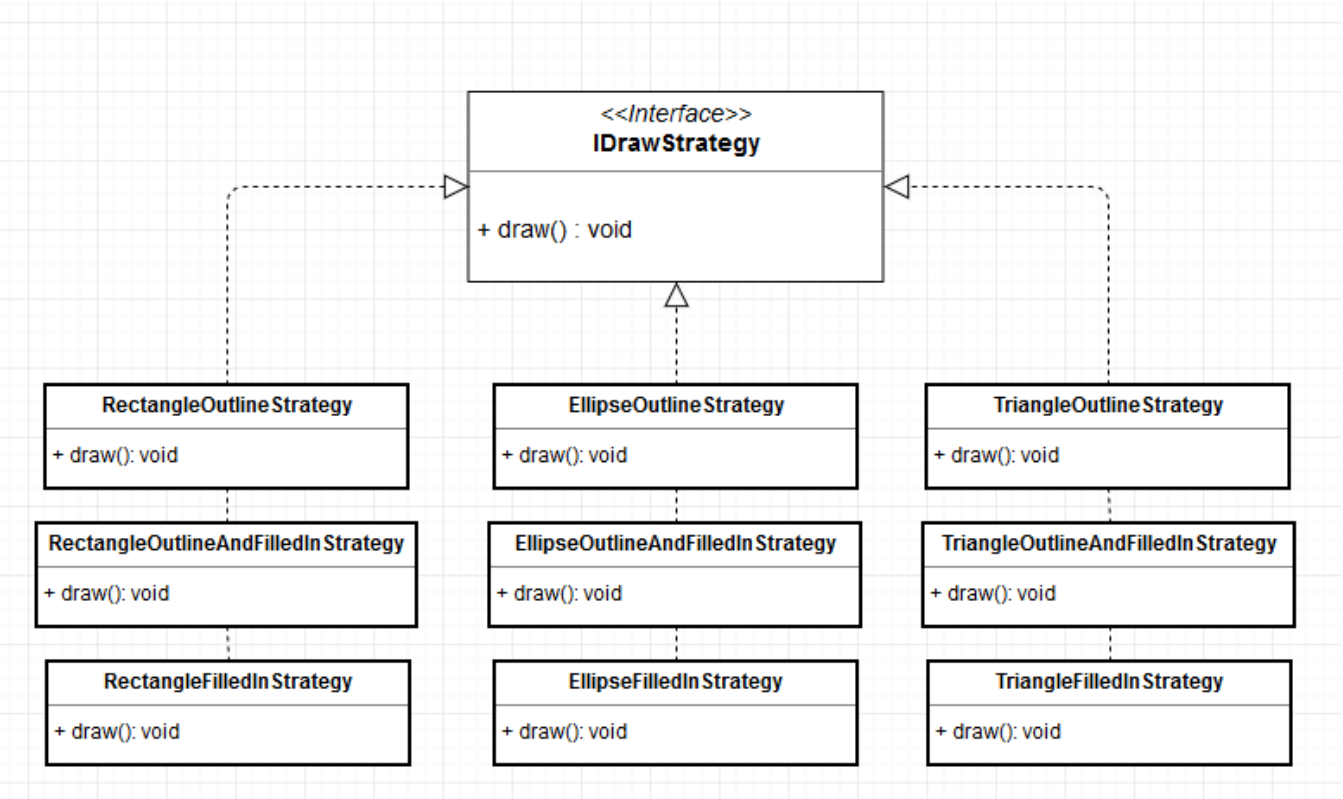
Shape Project Write-Up

UML Diagrams:

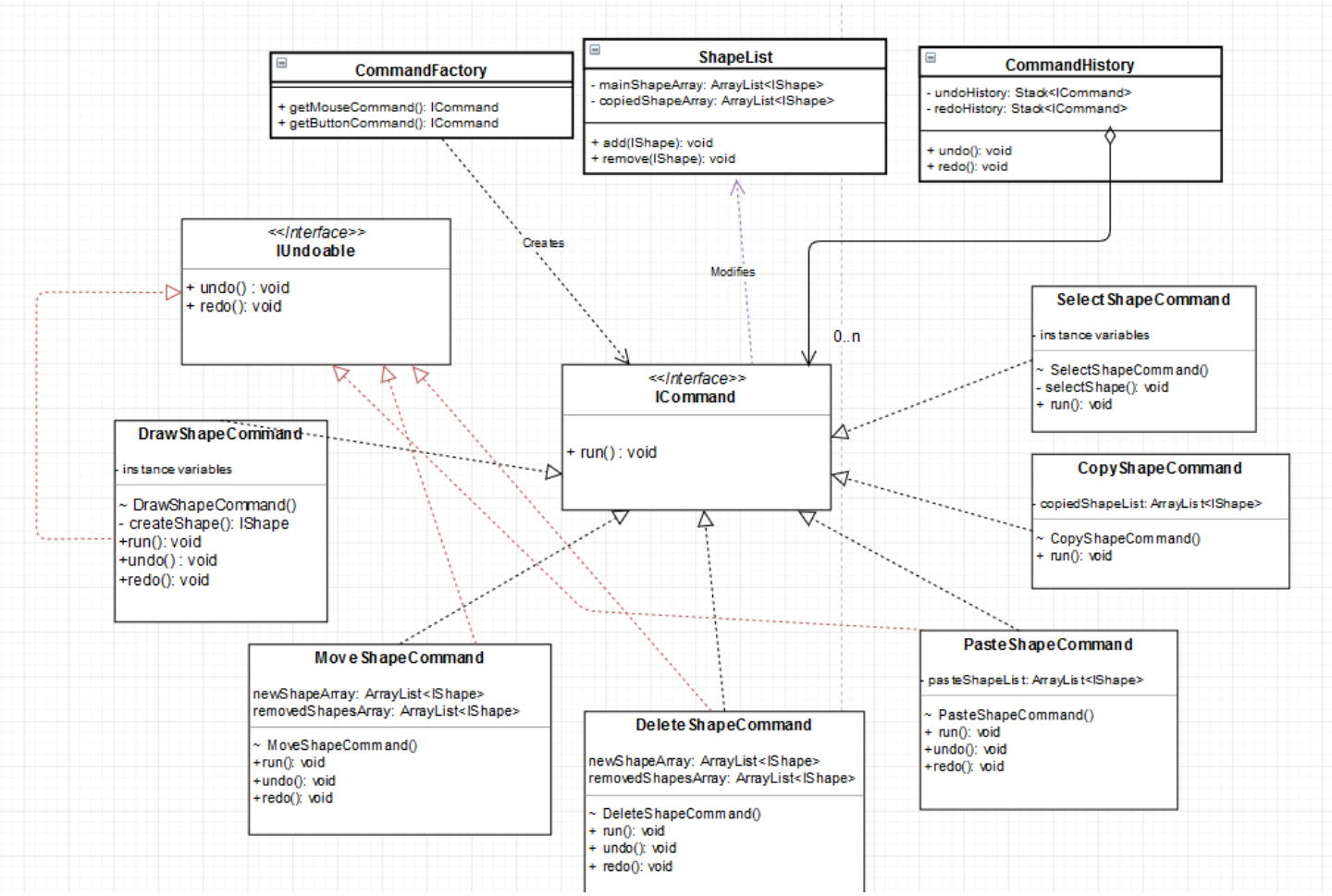
Shape Classes:



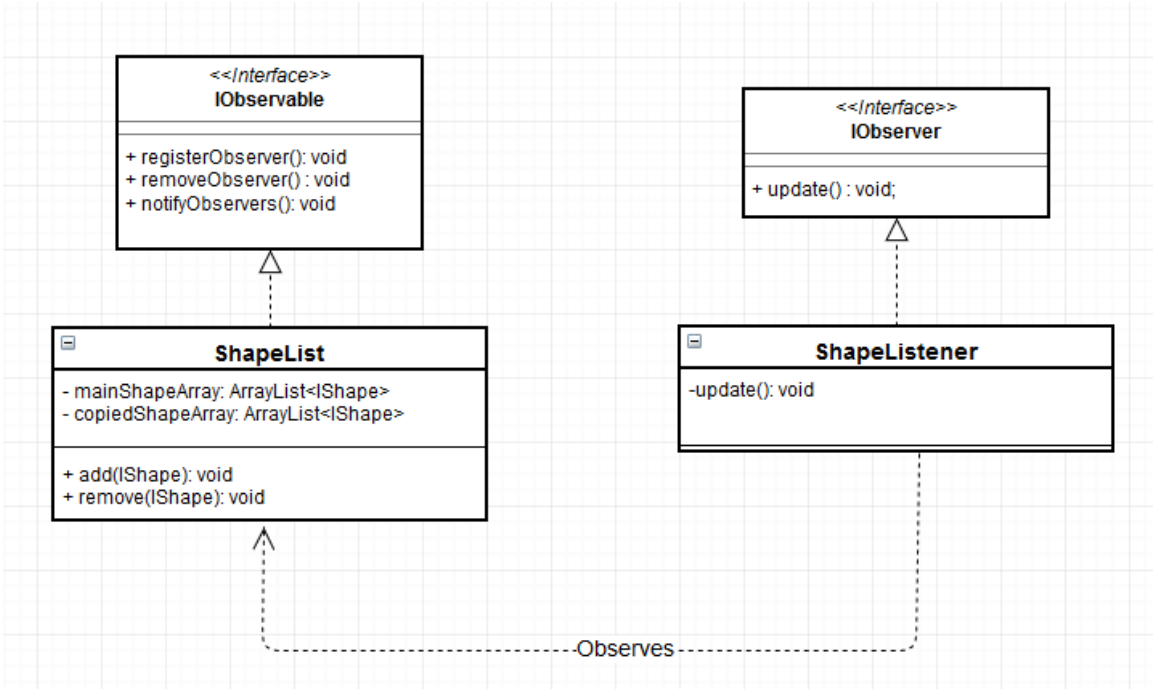
Strategy Classes:



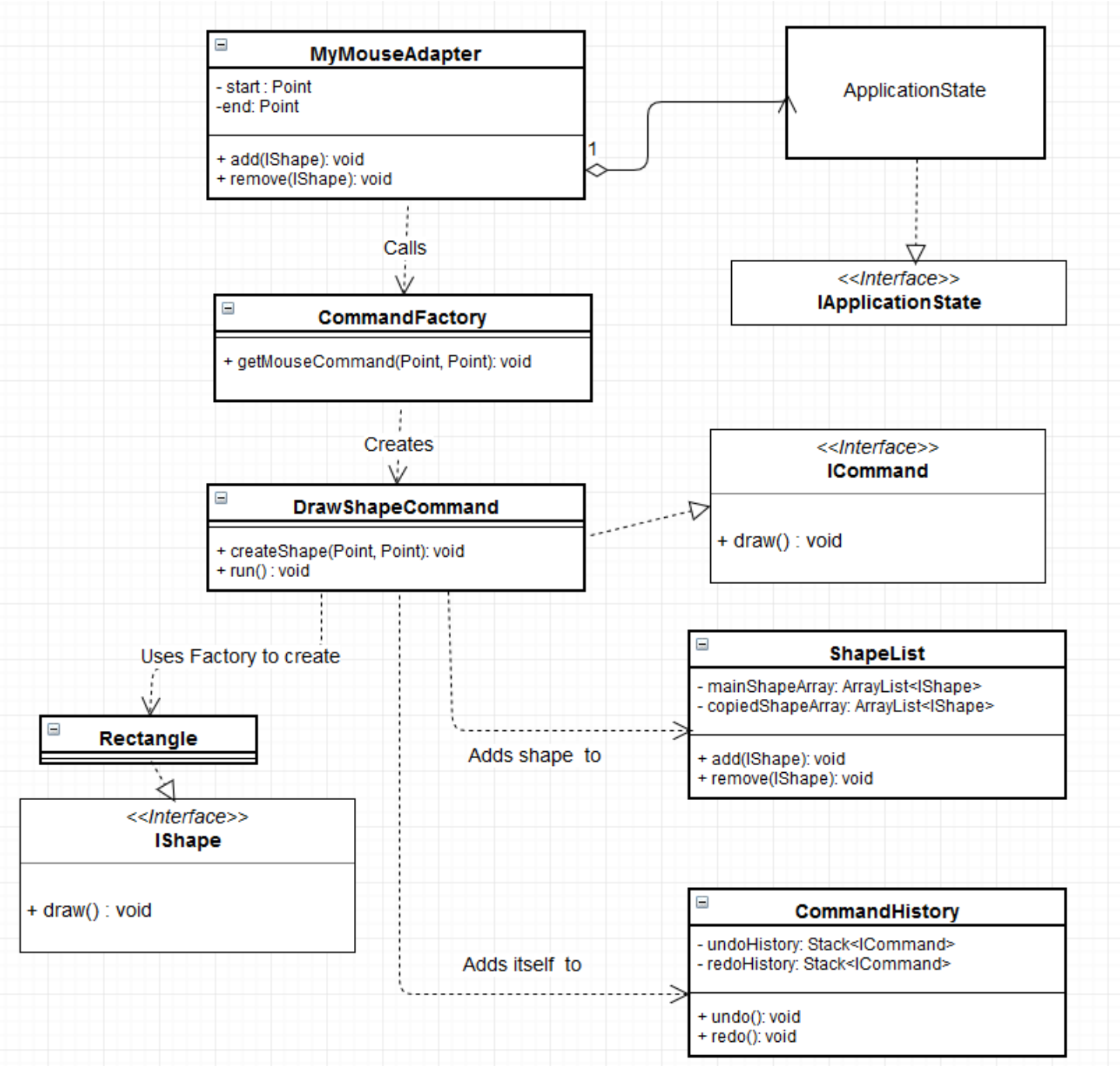
Command Classes:



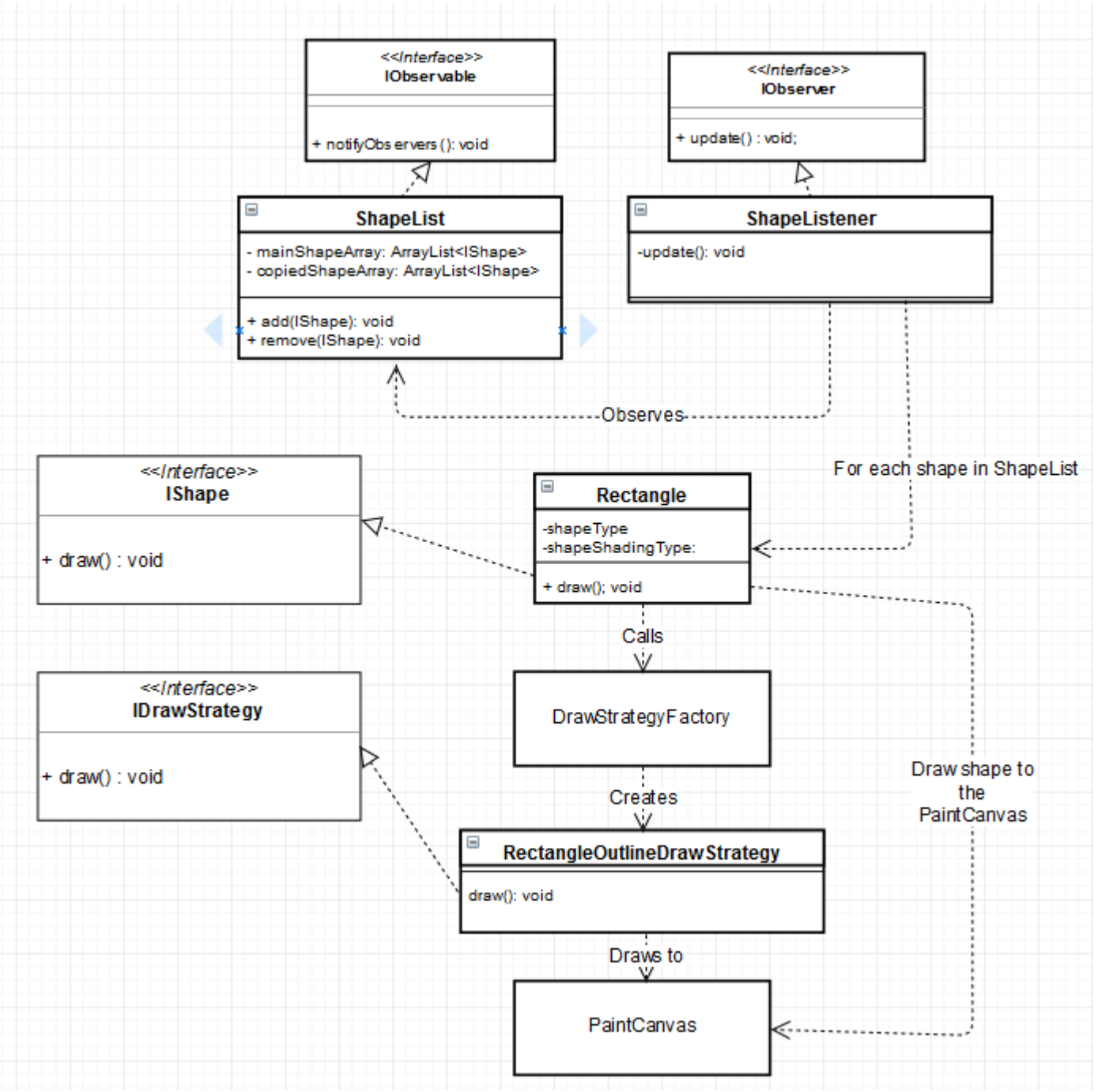
Observer Pattern:



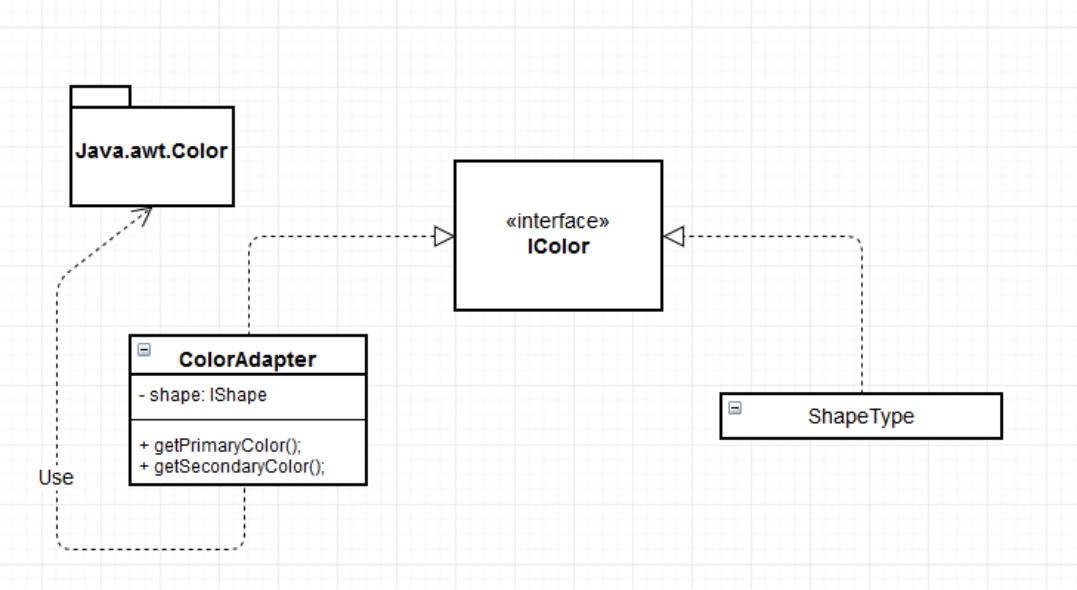
Command Pattern Example: Listening for Mouse Events



Strategy Pattern: How shapes are drawn after ShapeListener is updated.



Adapter Pattern:



Time Journal:

Note: Some dates are accurate (based on GitKraken commits). Some dates are approximate.

Jan 23, 2018

1. Examined starter code provided by Professor Sharpe
2. Experimented with code by changing dimensions, colors, and enums to see what would happen.
3. Began visualizing how the code and components fit together. Added some comments.

Feb 14, 2018

1. Sat down and re-watched the lecture from Feb. 6th detailing Professor Sharpe's UML Sequence Diagram on how to implement the JPaint project
2. Implemented IShape interface. IShape only contained 1 method: draw().
3. Implemented Rectangle, Ellipse and Triangle Classes. Each shape only had 3 instance variables: Start Point, End Point and Application State.
4. Implemented a Static ShapeFactory for creating my IShape objects.

Feb 17, 2018

1. Added a mouse adapter class with Mouse Pressed and Mouse Released functionality. I could print out the coordinates of e.getPoint() in the console but was unable to draw anything on the PaintCanvas.
2. Refactored my ShapeFactory to take in the Application State instead of a String

Feb 20, 2018

1. My friend Karthik helped me get my Mouse Adapter to draw. I did not know I had to add the MouseAdapter to my PaintCanvas and get a Graphics2D object from the PaintCanvas.

2. Created my first DrawShape command. It only drew a rectangle outline, but I could do so by dragging my mouse across the PaintCanvas.
3. Created three different draw methods that would call drawRect(), drawOval(), and drawPolygon().
4. Figured out how to draw filled-in shapes using fillRect(), fillOval(), and fillPolygon().
5. Figured out how to draw outline and filled-in shapes by first calling draw filled in, then draw outline. Note: if I do it in reverse, some of the outline gets cropped out.

Feb 21, 2018

1. Created a DrawShape method that took in the ApplicationState and drew a shape with the corresponding ShapeType and ShapeShadingType.
2. An Issue: I was unsuccessful in taking in PrimaryColor and SecondaryColor from the ApplicationState. Turns out Graphics2D.draw<Shape> is incompatible with ShapeType and requires that you use java.awt.Color. I needed a way to convert a ShapeColor to java.awt.Color.
3. Solved the issue by creating a ColorFactory. ColorFactory takes in a ShapeColor and returns the corresponding java.awt.Color via a list of if/else statements.

Feb 24, 2018

1. Implemented my IObserver and IObservable interface.
    I. IObserver methods: update()
    II. IObservable methods: registerObserver(), removeObserver(), notifyObservers()
2. Created a new ShapeList class for storing shapes. ShapeList contained only an ArrayList of shapes.
3. PaintCanvas now implements IObserver. Added update() method which iterates over and draws all the shapes in ShapeList.
4. ShapeList now implements IObservable. Updated IObservable interface methods to add PaintCanvas as an observer and then notify it each time a shape is added or removed.
5. Refactored DrawShape method to no longer draw shapes. Instead it adds the new shape to ShapeList.
6. Implemented SelectShape method based on algorithm from: https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection
7. Added two methods to IShape interface: overlaps(), setSelected()
8. Updated Rectangle, Ellipse, and Triangle to implement the new IShape interface and contain an overlaps() method which checks if two points overlap with it
9. Created a new ICommand interface that takes one method: run().
10. Refactored DrawShape method to implement ICommand. Moved method into run().
11. Refactored SelectShape method into an ICommand object.
12. Implement Command Pattern in MouseAdapter class. Created a CommandFactory that returns the appropriate command based on the Application State.

Feb 28, 2018

1. Began working on MoveShape command. I tried to move shapes by changing their start and end points but was unsuccessful. Turns out that I did not have any way of changing the X and Y coordinates of my Start and End points.
2. In preparation of for Undo/Redo functionality, I decided it would be easier to remove the current shape from ShapeList and add a new one at the desired location.
3. Implemented MoveShape command by removing and adding new I Shapes instead of changing coordinates.
4. An issue: Moving the object would add the new shape but would not successfully remove the old shape on the PaintCanvas. After doing some testing, I verified that the shape was removed from ShapeList. The problem was that I was not able to repaint the canvas before drawing the new ShapeList.

5. Issues with repaint(): Repaint will successfully clear the paintCanvas but will not draw the new ShapeList. After reading Java's documentation, repaint() automatically calls the update method. The first call in my update method is to repaint(). This leads to a function that recursively calls itself and does nothing.
6. Another issue: Moving multiple shapes will cause a ConcurrentModificationException. I'm speaking from the future, but it turns out that I can't remove an IShape from my ShapeList while I'm iterating over it.

March 5, 2018

1. Added Strategy Pattern to IShape's draw() method
2. Added IDrawShapeStrategy Interface with one method: draw()
3. Refactored Draw() methods in Rectangle, Ellipse and Triangle classes to call an IDrawStrategy.draw() based on application state. I did not implement a static factory for this case.
4. Refactored MouseAdapter to be able to draw from any direction. I did so by adjusting the start and end points from MousePressed() and MouseReleased() based on their position to one another. This is encapsulated in my adjustStartAndEndPoints() method in the MouseAdapter class.

March 7, 2018

1. Found a hack for repaint(). Instead of repaint(), I modified my update() method in PaintCanvas to draw a big white rectangle the size of the canvas. Not elegant but it works.
2. Fixed the MoveShape command. Apparently, you cannot remove Shapes from the ShapeList while I was iterating over it. Instead I created a separate ArrayList in my MoveShape command and iterated over that to add and remove shapes from ShapeList. Seems memory inefficient but it works.
3. Added ShapeListener class so that the ShapeList does not directly communicate with the PaintCanvas. ShapeListener now implements IObserver and Paintcanvas does not. Moved the update() method to ShapeListener.
4. Created IUndoable Interface with two methods: Undo() and Redo()
5. Refactored MoveShape command to implement IUndoable
6. Refactored DrawShape command to implement IUndoable
7. Created CommandHistory class to store command history. CommandHistory is heavily based on the CommandHistory class that Professor Sharpe created during lecture 3.
8. Created CopyShape and PasteShape commands.
9. CopyShape and PasteShape command implement IUndoable.
10. Added a CopiedShapeArray in ShapeList so PasteShape command would know what to print to the PaintCanvas.
11. Got JButtons to work by using UIModule.addEvent () -> EventName.NAME, () -> command. One important thing that wasted 30 minutes for me: since I implement ICommand, I forgot I should have called ICommand.run().
12. Tested my IUndoable commands and fixed any issues with their undo() and redo() methods. This is a lot easier when the JButtons work!
13. New issue: I can no longer only create shapes based on the ApplicationState. In the event of a Copy, Undo, or Redo event, I need to use a previous shape to create a new shape. However, I have no way of getting and IShape object's instance variables.
14. Refactored IShape Interface to include getters and setters for its instance variables. These included: getStartPoint(), getEndPoint(), getPrimaryColor(), getSecondaryColor() and so on….
15. Testing and refactoring for IUndoable commands until they worked.

March 8th, 2018

1. Putting on some finishing touches.

2. Added Error messages in case you try to Copy or Delete with nothing selected.
3. Have a working version that satisfies the requirements.

Time Summary

Week	1	7	8	9	10
Design	3	2	2	3	1
Code	0	10	15	24	12
BigBug	0	2	4	5	2

Notes on Patterns

Static Factory Pattern:

I listed the Static Factory Pattern first because it is the design pattern I used most. I often combined the Static Factory with another pattern such as the Strategy pattern or Command Pattern.

The benefit of a Static Factory is that I can delegate what object to create until runtime. It also consolidates all the if/else statements into one place, which minimizes the extent I must break the Open-Closed Principle. For example, if I wanted to add a Rhombus shape class, I would only have to refactor the ShapeFactory instead of all my shape command objects. You can find my CommandFactory in the controller.commands package, my ShapeFactory in model.shapes package, my DrawStrategyFactory in the and my ColorFactory in the model package.

In all my factory patterns, only the factory is public. The constructors for the objects they return are package private. Users call the static get<Object>() command and pass in the relevant variables. The get<Object>() methods then determines which object to create and return.

Strategy Pattern:

I use the strategy pattern combined with the static factory pattern in the draw() method of my Rectangle, Ellipse, and Triangle classes. Depending on the ApplicationState's ShapeShadingType, the code required to draw Outline, Filled-In, and OutlineAndFilledIn shapes was different. I encapsulated what varies by using the strategy pattern. You can find all the strategies in the model.strategies package.

To implement my strategy pattern, I have an interface called IDrawStrategy which encapsulates the code needed to draw any specific type of shape. I then use a DrawStrategyFactory to create the an IDrawStrategy based on the ShapeType and ShapeShadingType passed in to the factory. This way my IShape classes do not need to how to draw themselves. A call to IDrawStrategy.draw() takes care of the drawing.

Command Pattern:

I use the command pattern in the JPaintController's JButtonPane and in the MyMouseAdapter class and in the view.gui package. The command pattern is combined with my CommandFactory. The benefit of the command pattern + static factory is that I can encapsulate all the possible actions needed to be performed after a button press or mouse click and store them for later use as needed during run time.

In my implementation, either the MyMouseAdapter or the JPaintController is the invoker. Once the MouseAdapter receives a MouseEvent it calls the CommandFactory to run return the appropriate command

which the MyMouseAdapter runs. For the JPaintController, once a JButton is pressed, it registers an Event that is paired with an ICommand. The JPaintController then executes the command. This decouples the GUI from knowing the implementation of which command to call and is only responsible for listening to user events. This fits well the Dependency Inversion principle since both MyMouseAdapter and the Commands interact through the ICommand interface.

By storing all the actions performed in a command object this enables my Undo and Redo functionality. Commands that can be undone implement my IUndoable interface which has the commands undo() and redo(). Each time an IUndoable is executed, that are added to my CommandHistory class's Undo stack. Each time an Undo() command is executed, that command's undo() code is executed and the ICommand is transferred to the redo stack. Without the command pattern, this functionality would not be possible in this project.

Observer Pattern:

I use the observer pattern for repainting my ShapeList to my PaintCanvas every time there is a change to the ShapeList. Each time we add or remove a shape to the ShapeList, it notifies a ShapeListener class, which then clears the PaintCanvas and proceeds to redraw every shape in the ShapeList. ShapeList implements my IObservable interface, and ShapeListener implements my IObserver interface. The benefit of the observer pattern is that I don't need to call update() whenever I draw, paste, or delete any shapes. The ShapeList will take care of repainting the canvas automatically.

Adapter Pattern:

I use the adapter pattern to make my ShapeColor class compatible with Java.awt.Color. This was necessary because Graphics2D only accepts Java.awt.Color arguments. I created an IColor interface without any methods and made ShapeColor and ColorAdapter both implement IColor. ColorAdapter imports the Java.awt.Color package and contains two public methods getPrimaryColor() and getSecondaryColor() which convert a ShapeType class into a java.awt.Color class. This way, when drawing shapes, we can call colorAdapter.getPrimaryColor() and receive an IColor that is compatible with Graphics2D.

Successes and Failures

Successes:

Many things went right with my project. I began the JPaint project after the project check-in due to workload constraints from my other class. I dedicated many full nights on this project over the three weeks to make up for starting late. Good news though. Not only was I able to get my project to compile and satisfy all the requirements, I also used 5 design patters instead of 4.

Some extra features worth noting:

1. You can draw and select shapes by dragging the mouse in any direction.
2. Error checking (e.g. No shapes selected while using Copy / Delete command).
3. After a Draw, Paste or Move command, the newly added shape(s) are automatically selected. This is a feature, not a bug. I enabled this for testing, and I left it as is because it was so convenient not having to reselect the shape each time I drew it or moved it.

I will skip over my design pattern successes because they are mentioned in the "Notes on Patterns" section. I want to mention some additional design decisions not mentioned previously.

One success I had was refactoring my ShapeList to adhere to the Open-Closed principle. I originally had methods for add, copy, paste, delete and undo inside my ShapeList class. My ICommands (e.g. MoveShapeCommand) would call ShapeLists's default implementation for the function. I quickly realized that every time I added a new command, I would have to change the ShapeList, thus violating the Open-Closed principle. I also decided that my ShapeList should only be responsible for storing shapes, and decoupled from knowing how to copy, paste and delete shapes.

My happiest moment was when I figured out how to get my move command to work and how to get my update() call in ShapeListener to redraw the ShapeList. At first my Move Command would not fail and throw a ConcurrentModificationException exception. I posted on the forums and got a reply from Professor Sharpe. The issue was that I was removing an item from the ArrayList I was iterating over. I also heard a tip from the forums to draw a big rectangle covering the PaintCanvas instead of repaint(). Now that my PaintCanvas behaved as expected, I had created made my first real application.

Even though this project was a lot of work, I did learn a lot about programming, and dare I say…it was fun!

Failures:
I originally wanted to separate IShape into two separate Interfaces: IShape and IDrawable. This way IShape would be responsible for representing a Shape, and IDrawable would implement the methods needed to draw the shape. This was more in line with the Single Responsibility Principle. However, during implementation, I realized that IDrawable needed access to most if not all of IShape's methods in addition to its own. Instead of making IDrawable extend IShape, I just left the IShape Interface alone. I'm still figuring out how to best separate the functionality.

I tried unsuccessfully to structure my project using a true Model View Controller architecture. To do so properly would require that none of the classes in my View and Controller communicate directly with each other but only communicate through the Controller. The only time my program satisfies the proper MVC architecture is when the user clicks on the JButtons in the UIModule. EventNames are processed by the JPaintController and passed on to the appropriate ICommand to modify the ShapeList. Otherwise, my View directly changes the Model (aka. MyMouseAdapter in View directly calling CommandFactory in Model), and likewise, the model directly alters the View (aka. when the ShapeListener directly draws onto the PaintCanvas after ShapeList notifies it).

Something that still bugs me is that I still haven't resolved is how to get repaint() to work. My repaint() method would only clear the PaintCanvas but would not redraw. I used a hack by drawing a big white rectangle to cover the PaintCanvas instead.

Lastly, if I have one regret, it is that I didn't have time to implement some unit tests for my project. I was always taught in my introductory classes to insert System.out.println() statements in the code to see what was occurring. You will likely see many statements printed out in the console as a result. However, I would love to move on to more advanced testing techniques and become not just a better programmer but also a better software engineer.