1. List of missing features, bugs, extra credit, and miscellaneous notes. List features that don’t work and make me aware of any bugs in the code. List any extra credit or any other miscellaneous notes as well. Essentially, list anything I should know when grading.

**List of features I have implemented:** \*(Ex) = Extra Credit

* Pick a shape
  + Ellipse
  + (Ex) Perfect Circle
  + Rectangle
  + (Ex) Perfect Square
  + Right Triangle
  + (Ex) Isosceles Triangle
* Pick a primary color
* Pick a secondary color
* Click + drag to draw a shape
* Select shading type (outline only, filled-in, outline and filled-in)
* Select shapes
* Click and drag to select shapes
* (Ex) Right click to draw with secondary color (flip primary and secondary color)
* Copy
* Paste
* (Ex) Delete shape
* Undo
* Redo
* Move shape
* Support keyboard shortcuts
  + (Ex) Ctrl + C (Copy)
  + (Ex) Ctrl + V (Paste)
  + (Ex) Delete or backspace (Delete)
  + (Ex) Ctrl + Z (Undo)
  + (Ex) Ctrl + Y (Redo)
* (Ex) Selecting a shape displays dotted outline of the shape(s) you have selected – visually helpful
* (Ex) Canvas updates as the mouse is dragged for shape position/size preview
  + (Ex) Draw
  + (Ex) Select
  + (Ex) Move
* No tests written
* No known bugs
* The one thing I wish I had time to complete from the original design, is my Draw Strategies are a little long due to a switch statement that encompasses all 6 shapes to properly draw them on canvas.
* Developed on a mac (just in case key commands or other things don’t work)
* Performance of mouse drag isn’t amazing but certainly works well.

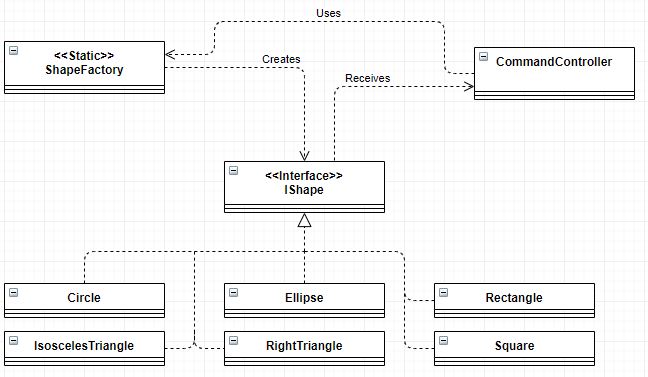
2. Notes on design. Indicate five four design patterns used in your project. There should be some variety in patterns you discuss – you can’t discuss the same pattern more than twice.

For each pattern, in a few sentences, describe the classes involved, why you chose to implement that pattern, and what problem it solved. Include a design class diagram as well. Be sure to include all significant class relationships: realization, specialization, and association. Show associations as dependencies, aggregations or compositions when appropriate. Show attributes and methods only if they are crucial to understanding the class relations (e.g. for Dependency relationships). You do not need to show all fields and methods!

These notes/diagrams should take approximately 2-3 pages.

**Design Implementations:**

* Static Factory
  + Shape Factory

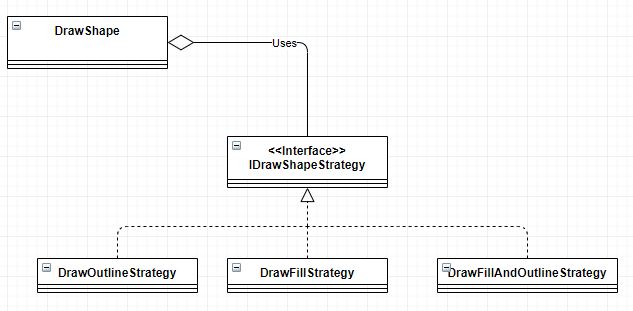


Classes Involved: ShapeFactory, Circle, Rectangle, RightTriangle, IsoscelesTriangle, Ellipse, Square, IShape, CommandController

IShape is the interface that all shapes are based from, it describes the methods that each shape will include and what their return types and arguments are. ShapeFactory is the class that does the creation of shapes in the type IShape. Each method in the ShapeFactory takes in many arguments, that probably creates and returns a class of the shape created. The CommandController calls the ShapeFactory and then receives an IShape.

Having the static factory made my code cleaner so that when any other coder looks at my program, they can find a centralized creation of shapes. Furthermore, that each shape will have the same basic methods coming from IShape. This also makes it possible for easier future additions of other shapes.

* Strategy
  + Draw Shading Style (Example below)
  + Mouse Dragged
  + Mouse Released

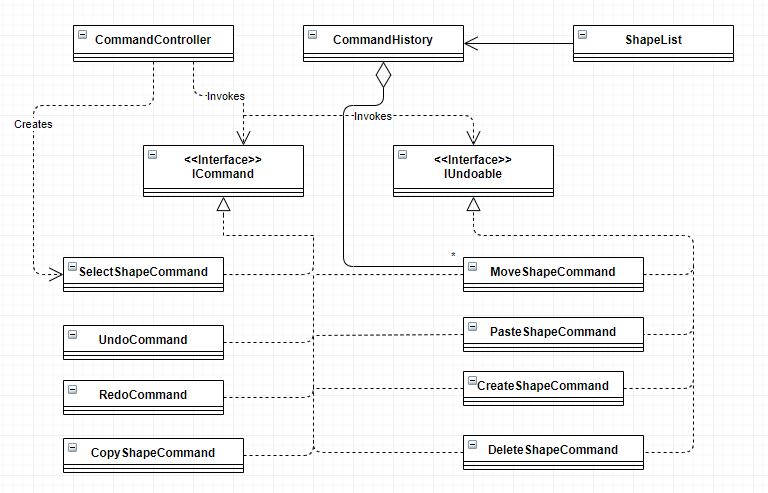


Classes Involved: IDrawStrategy, DrawOutlineStrategy, DrawFillStrategy, DrawOutlineAndFillStrategy, DrawShape.

IDrawStrategy is the interface that all Draw Strategies are based from, it describes the methods that each strategy will include and what their return types and arguments are. DrawShape is the class that determines which strategy to use while drawing based on which shape shading type is selected on the application. A new Strategy class is then created passing in IShape and PaintCanvas.getInstance().getGraphics2D to properly draw the selected shape with shading style.

Having the strategy design made my code cleaner. It also made my code a lot more readable by getting rid of long nasty conditional statements and replacing them with much easier to understand conditionals, as well as making it easier to implement other strategies in the future.

* Command
  + Mouse Pressed

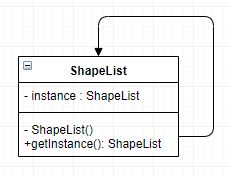


Classes Involved: CommandHistory, CommandController, ICommand, IUndoable, ShapeList, CreateShapeCommand, SelectShapeCommand, MoveShapeCommand, CopyShapeCommand, PasteShapeCommand, DeleteShapeCommand, UndoCommand, RedoCommand.

ICommand is the interface that all Commands are based from, it describes the run() method that each strategy will include and what it’s return type and any arguments. Some Commands also base their methods from IUndoable interface that describes undo() and redo() methods. Each command that implements IUndoable feeds itself into the CommandHistory so that a state can be accomplished. The state in this case is ShapeList. Each command that implements IUndoable has methods that modify the ShapeList to correctly undo and redo commands. CommandController is the class that calls these commands. CreateShapeCommand, MoveShapeCommand, PasteShapeCommand, and DeleteShapeCommand are the four classes that implement IUndoable and modify the ShapeList state and can be undone and redone. SelectShapeCommand and CopyShapeCommand are the commands that only implement ICommand and have only a run method.

Having the command design made my code cleaner, of course. It also made my code a lot more readable by getting rid of long nasty conditional statements, and replacing them with much easier to understand conditionals, as well as making it easier to implement other commands in the future. The command pattern also allowed me to implement an undo and redo function that can be controlled with a keylistener as well as the GUI buttons. I’m unsure how I would even implement these functions without a ShapeList state.

* Singleton
  + Shape List (Example below)
  + Paint Canvas



Classes Involved: ShapeList, Mouse Strategies, Commands, and CommandController.

The ShapeList singleton provides a single instance of the ShapeList class not allowing other shapelists to be created thus controlling the state tightly. The other classes involved call upon this instance to use in their methods which may alter the state of the ShapeList. Since the ShapeList is a singleton no other ShapeLists can be created accidentally.

Having the singleton design made it possible to remove many class and method dependencies throughout my code. When the shapelist is required I called the instance instead of passing it along a daisy chain of arguments.

**Things that went right with my project:**

I was very happy that I was able to figure out how to get a shape to update as the mouse was dragged. It was a feature that kept nagging at me throughout the beginning implementation since I’m used to how MSPaint and other paint programs function. I was also really excited to get the select feature working that visually shows which shapes are selected by surrounding them with a dotted rectangle.

The ShapeFactory was one of the first design patterns I implemented because I was able to immediately visualize the necessity and design. Draw strategies were similar for me, I understood the problem I was running into with a long conditional statement and that a strategy design would prove useful.

Reversing the colors on right click I think took all of 10 seconds at the project due to ease of implementation.

Implementing isosceles triangles went well due to my background in math.

I was very happy to be able to use HashMaps to store original x and y points for each shape, this allowed me to use mouse drag when moving shapes around and have them update instantly.

**Things that went wrong:**

Initially trying to solve the mouse dragged issue proved extremely difficult. I was just trying to implement this feature too early in development.

I had a really difficult time trying to work through the ability to clear the canvas and redraw shapes. I was trying to get the “repaint” function to work and after maybe 5 hours of struggling with it I decided to ditch it and work with the ability to just fill the canvas with a large white rectangle to cover up old shape drawings.

One of my other difficulties was getting the Mouse Handler to work. I ended up having to ask fellow students how the handler class added listeners to the canvas. My understanding of how the classes interacted was lacking.

Lastly, I had some difficulty understanding how the command history would work, specifically with what arguments were needed. The homework and demos provided excellent examples to follow allowing me to understand how to develop a command history.

**Things that failed:**

I’m still trying to figure out how to refactor my drawing strategies for the long conditional that draws each shape. I think that I can just apply a strategy for each shape but 18 classes sounds like too much and there might be an easier solution.