Bryan Morandi

SE450 – Prof. Dan Walker

JPaint Project Report

September 22, 2021

Missing Features and bugs:

Sprint 3

- This may not be the biggest bug, but the dotted selection outline that appears around shapes will overlap with the triangle shapes that are very large. This only occurs in the most extreme cases when selecting triangles.

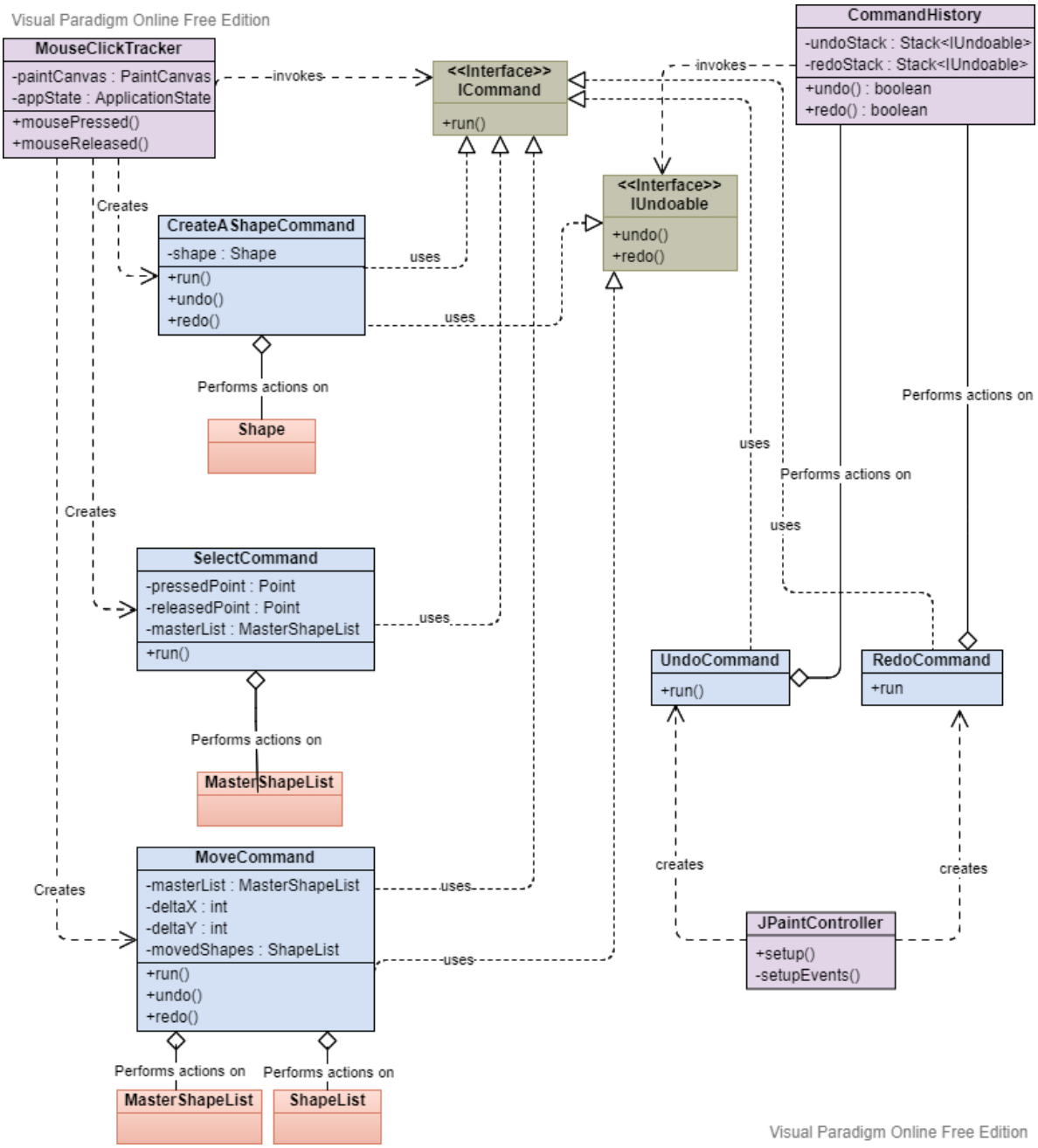
Sprint 4

- Group Paste-redo and ungrouping a pasted group of groups
    - After pasting a copied group then undoing, then redoing, I cannot undo the redo of the group. I believe this is an issue with my pasteShape() or undoPaste() methods in my ShapeGroup Class.
    - After pasting a group that is a group of groups, then undoing that group, results in a lot of children being rendered onto the canvas that should not yet be ungrouped. I also believe this is an issue with my pasteShape() method in my ShapeGroup Class.

Link to GitHub Repository: https://github.com/Byte-of-Bryan/jpaint

Design Patterns:

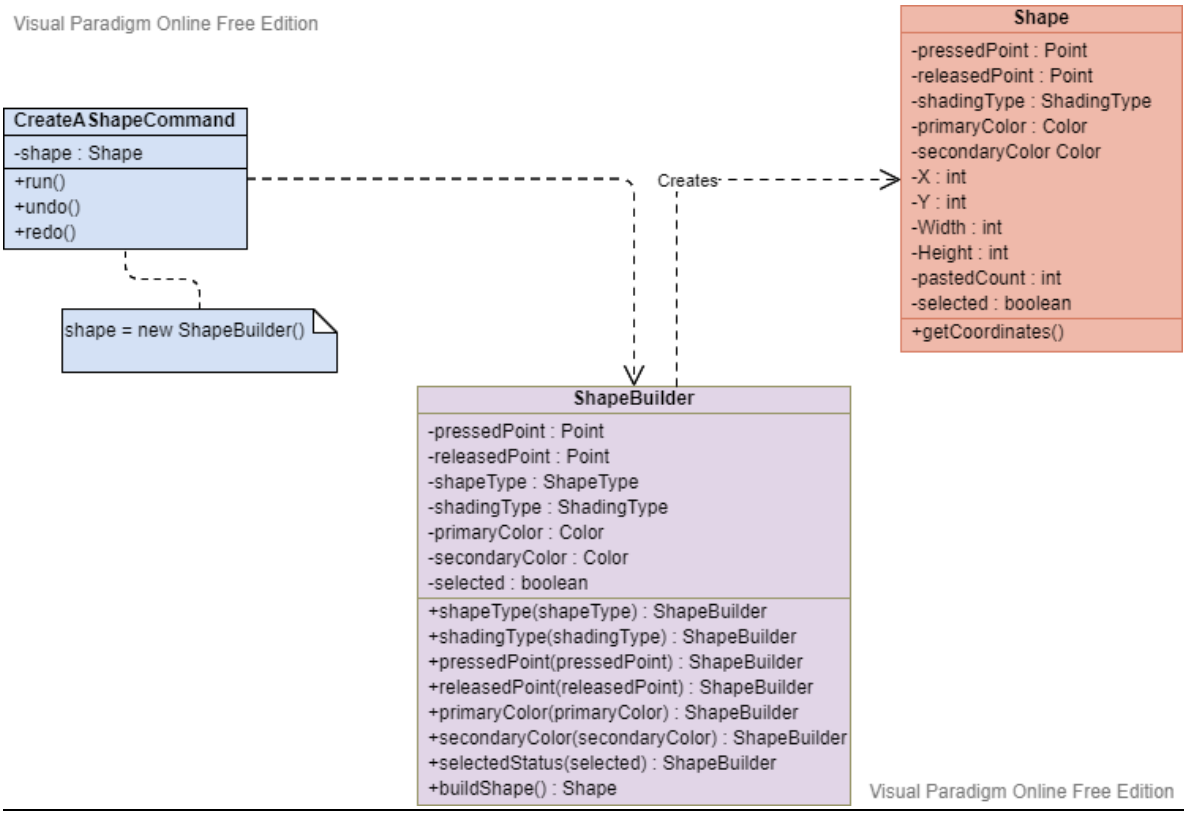
Command Pattern



The command pattern is my most used pattern, so the classes that make up this pattern only cover a small section of the classes that utilize the command pattern. The classes are: MouseClickTracker, CreateAShapeCommand, SelectCommand, MoveCommand, Shape, MasterShapeList, JPaintController, UndoCommand, RedoCommand, CommandHistory, ICommand and IUndoable. I chose to implement this pattern, because there are many commands that need to be executed when a certain action happens or when a button is pressed. The command pattern makes implementing these commands simple because each command class can implement the ICommand interface and utilize the method

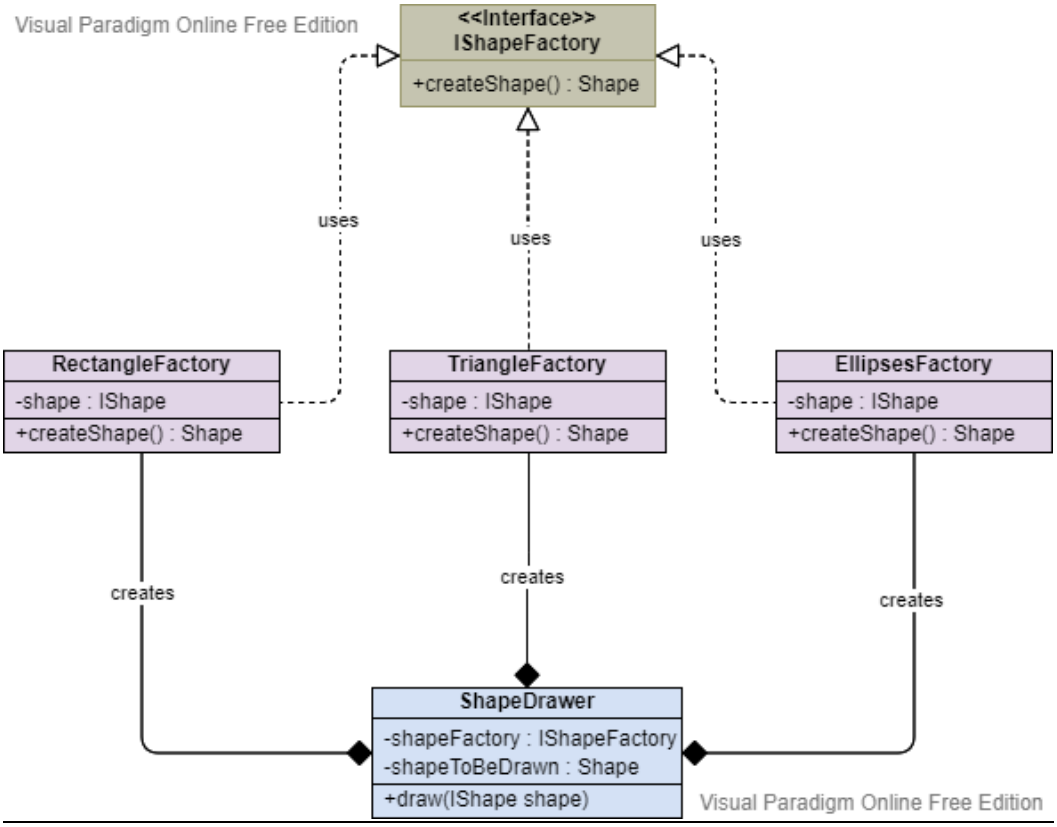
differently depending on the input or button that is pressed. The command pattern solved the problem of what the mouse should do depending on the active mouse mode. If the mode is Draw, the draw command is run, if the mode is Select, the select command is run, and so on. Additionally, the Command Pattern also solved the problem of undoing and redoing commands that were previously run. The UndoCommand and RedoCommand use the ICommand interface to perform actions on the undoStack and redoStack that are attributes on the CommandHistory class. Other classed implement the IUndoable interface, log their commands into the respective stacks, and whenever the user presses the undo or redo button, their previous commands are respectively undone or redone.

Builder Pattern



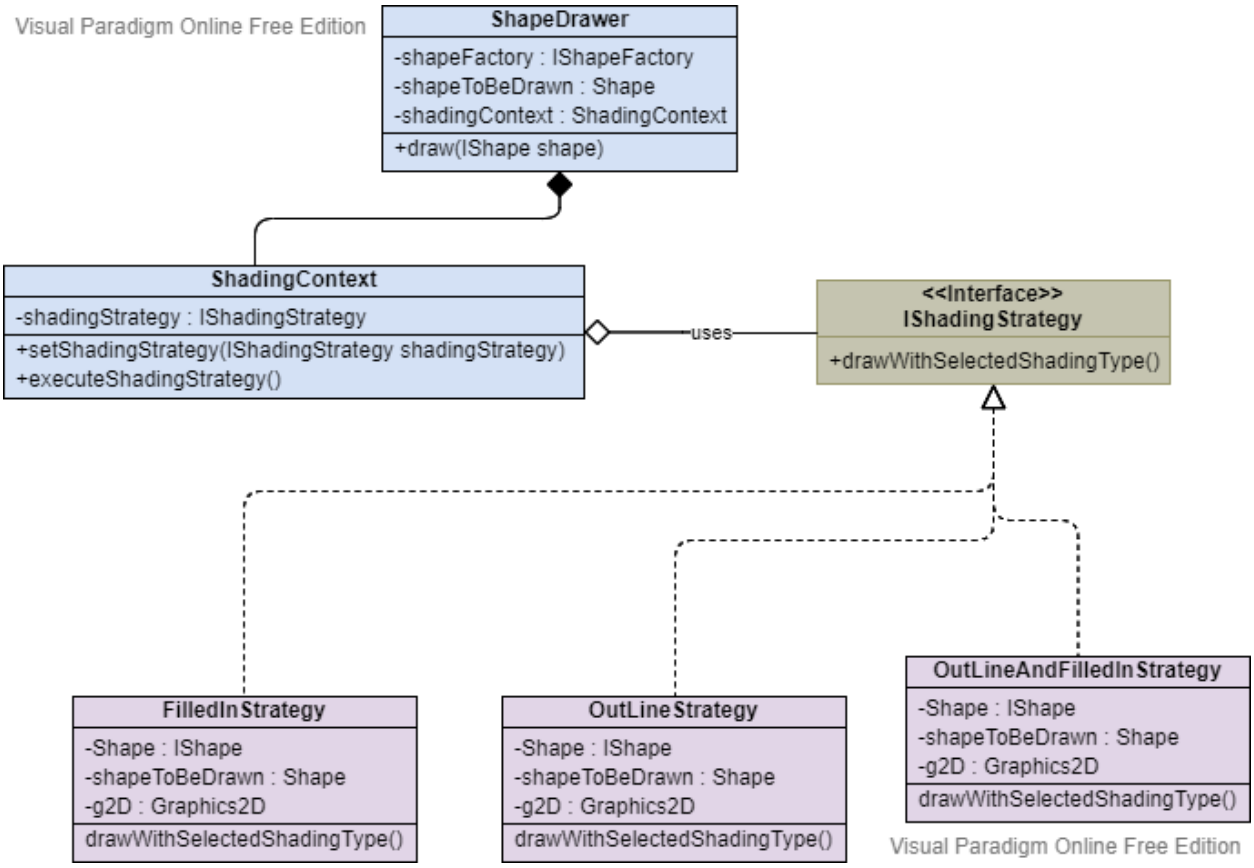
The classes that are used in the builder pattern are: CreateAShapeCommand, ShapeBuilder, and Shape. I chose to implement the builder pattern because I wanted to create an easy way for the attributes of a shape to be initialized to make new instances of the Shape class. The builder pattern solves the issue of having to manually enter in the fields of the Shape class each time I want to build a new Shape instance. The CreateAShapeCommand class sets a new instance of a shape to a ShapeBuilder class, this class will automatically enter in all the fields to construct a new Shape.

Factory Method Pattern



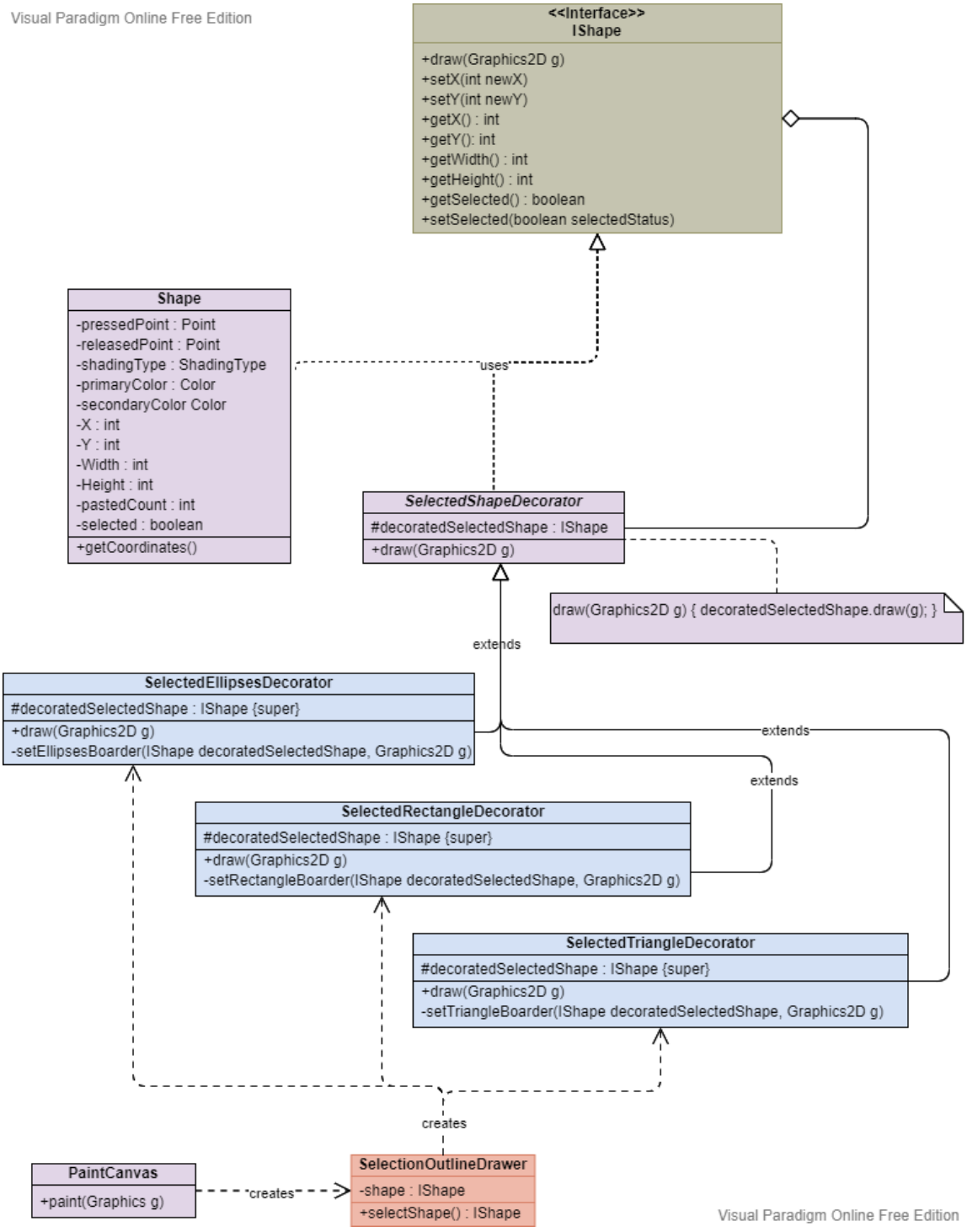
The classes that I used in the factory method pattern are: IShapeFactory, RectangleFactory, EllipsesFactory, TriangleFactory and ShapeDrawer. I chose to implement the factory method pattern because I wanted to create a solution to draw different kinds of shapes based on the users Shape Type selection. A benefit to using the factory method pattern is that I can create many kinds of shapes, that relate to the base IShape class. My implementation solves the issue of only being able to draw rectangles from the base code. After implementing the factory method pattern, user can additionally draw ellipses and triangles pointing towards the 4 cardinal directions based on mouse click and release positioning.

Strategy Pattern



The classes that I used in the strategy pattern are: ShapeDrawer, ShadingContext, IShadingStrategy, FilledInStrategy, OutLineStrategy and OutLineAndFilledInStrategy. I chose to implement the strategy pattern because I wanted to have my ShadingContext class store the currently selected shading type (filled-in, outline, outline-and-filled-in). This allows the ShapeDrawer class to draw the shape without having to care about what shading strategy it must print. The strategy pattern is useful because it "encapsulates what varies". In this use case I am encapsulating the different shading strategies, so the ShadingContext does the work of executing the correct strategy, so the ShapeDrawer doesn't have to.

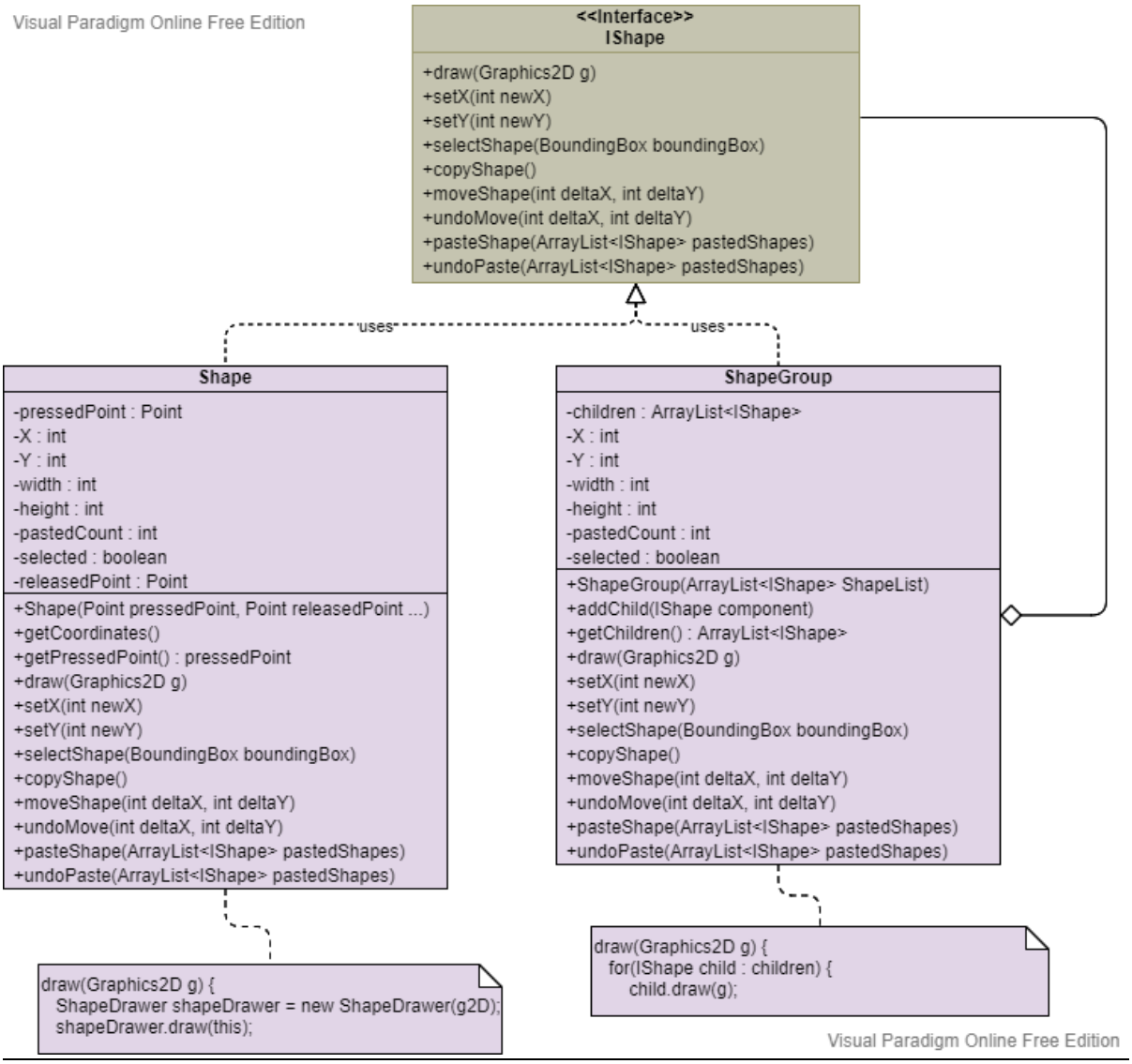
Decorator Pattern



The classes that I used in the decorator pattern are: IShape, Shape, SelectedShapeDecorator, SelectedEllipsesDecorator, SelectedRectangleDecorator, SelectedTriangleDecorator, SelectionOutlinDrawer and PaintCanvas. I chose to implement the decorator pattern, because this pattern allows me to wrap a target class in another class to add additional functionality, this was an ideal option to add the dotted outline to shapes when selected. The decorator pattern solved the issue

of not being able to see what shapes are selected and deselected upon mouse click inside each shapes bounding box. I did have to go through the PaintCanvas with this pattern because that is where my Graphics2D object lives when drawing shapes. Also, to have the outlines appear on top of each shape, I first looped through all shapes and printed, then looped through the selected shapes with the added decorator functionality to print the outlines.

Composite Pattern



The classes that I used in the composite patterns are: IShape, Shape, and ShapeGroup. I chose to use the composite pattern because it seemed like the simplest option to create a group of shapes out of the patterns that we were taught. The composite pattern solved the issue of not being able to group shapes. The pattern makes it so once a group is formed, every child of the group is equally affected by move, paste, copy, etc. The client doesn't know whether it is moving a single shape (leaf) or a group of shapes (composite).

Successes and Failures

Successes

- This is the largest single project that I have worked on up to this point, so being able to complete the project and implement many different patterns is a big success.
- I think the way that I created the paste shape functionality to be offset whenever the paste button is pressed was clever. I added a paste count attribute to my IShape class and a couple methods to increment and decrement the counter depending on if the shape is undone or redone. I multiply an offset by the current paste count and then add that value to a new X and Y coordinate. This ensured the shape is pasted in a new location each time.
- On the same note, I was successful in offsetting a pasted shape to be offset from the last pasted shape location on the clipboard, even if the copied shape is moved. I created another algorithm that alters the X and Y coordinates of a pasted shape after a move based on the current X/Y values, pasted shape count and a delta X and Y offset. This algorithm is implemented in the Move class.

Failures

- The hardest part of this whole project for me was Sprint 4, I think a lot of my issues came from implementing code that changed the X and Y coordinates of a single shape. So, when I tried to implement the composite pattern, I had to rework the way my command pattern classes worked and break out the functionality so that individual shapes could be altered in a composite. Therefore, I created separate Move and Paste classes. I had some success with this by getting a group of shapes to move properly, but when pasting a group, I still have some issues that I couldn't solve.
- I originally thought I was implementing the strategy pattern to create different shapes (triangle, rectangle, ellipses) but after some reworking of the class, I realized that I was basically using the factory method pattern. So that was an easy change to move to a different pattern that worked better for the purpose of creating different shapes. I was still able to implement the strategy pattern to draw the different shading options.
- My graphics2D object lives in my PaintCanvas class, so this caused a few issues when I implemented my decorator pattern to draw the dotted line around selected shapes. My outlines were only showing up under my shapes, so I included my SelectionOutlineDrawer class inside my PaintCanvas. When shapes are repainted, another loop is run after the base shapes are drawn to draw the outlines around the currently selected shapes. I think this is a little different to how the decorator pattern is supposed to work, but this change resulted in outlines that are always on top of selected shapes.