JPaint

FINAL PROJECT

Ishita Mehta

SE 450 | AUGUST 16, 2018

Q: Is it a Feature or a Bug? A: Yes

- Pick a shape

- Pick a primary color

- Pick a secondary color

- Click + drag to draw a shape

    - All the shapes draw correctly. If click + drag is in reverse direction, then all the shapes except triangle draws with a corrected orientation.

- Select shading type (outline only, filled-in, outline and filled-in)

- Select shapes

    - The selection uses both clicking on a single shape. It only deselects in the select mode.

- Click and drag select shapes

    - It does select by clicking and dragging, but I have noticed it does not paste everything.

- Copy

- Paste

- Delete shape

- Undo

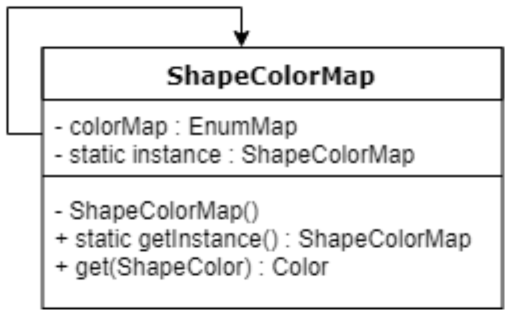
- Redo

- Move shape

    - It moves by clicking at the destination, not by clicking + dragging. So, clicking in move mode again will make a copy of the selected shape (as it “moves” it again). Also, moving multiple selected things will move it all to the one spot, instead of relative to the destination click.

Notes on Design

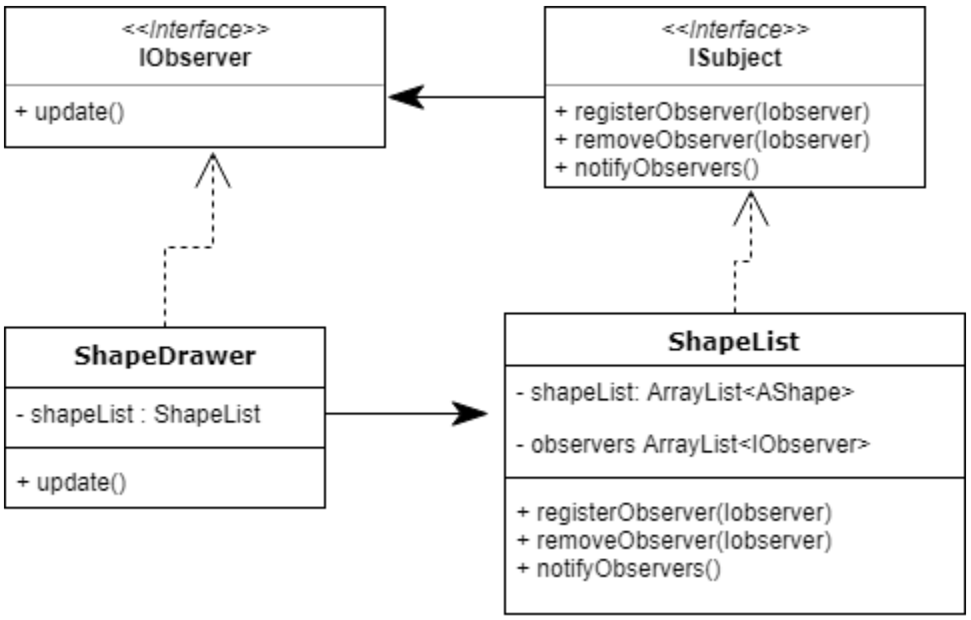
Singleton


ShapeColorMap
- colorMap : EnumMap
- static instance : ShapeColorMap
- ShapeColorMap()
+ static getInstance() : ShapeColorMap
+ get(ShapeColor) : Color

The ShapeColorMap class uses the Singleton pattern. It has an EnumMap as a private field, and an instance of itself as well. The constructor is made private, so this object can only be instantiated within this class. The public methods getInstance() and get(ShapeColor) allow access to the instance and the EnumMap.

Using the singleton pattern allows for a guarantee of Single Responsibility Principle. The ShapeColorMap's responsibility is to the hold the reference to itself and provide an easy relation from our ShapeColor to Java's Color Enum. Also, Singleton pattern solves the problem of encapsulation, where this class is basically serves the function of holding this relational data. It is worth noting that in this case, ShapeColorMap does not adhere to Open Closed principle. If more colors are added to the UI, this class will need to be modified. Nevertheless, for the scope of this application, the singleton pattern saves a lot of memory and confusion by not having various instances of the EnumMap and can be encapsulated in one place.

Observer

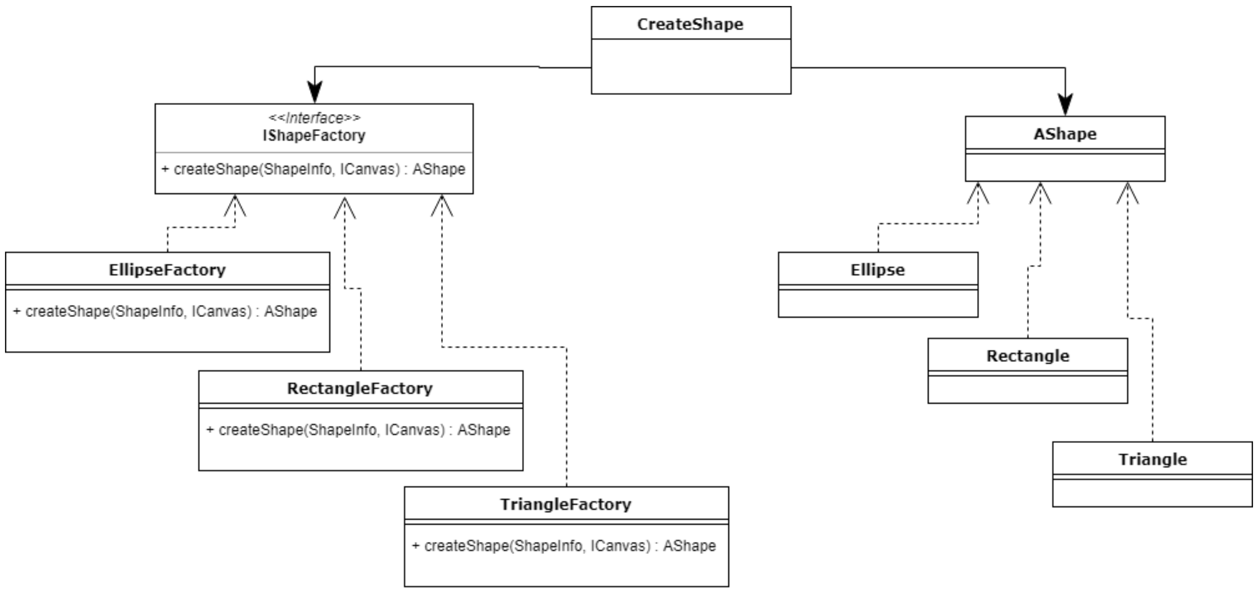


<<Interface>>
IObserver
+ update()
<<Interface>>
ISubject
+ registerObserver(Iobserver)
+ removeObserver(Iobserver)
+ notifyObservers()
ShapeDrawer
- shapeList : ShapeList
+ update()
ShapeList
- shapeList: ArrayList<AShape>
- observers ArrayList<IObserver>
+ registerObserver(Iobserver)
+ removeObserver(Iobserver)
+ notifyObservers()

ShapeDrawer class implements the Observer pattern. Whenever the ShapeList class changes, it updates its observers automatically (the ShapeDrawer), which carry out the appropriate actions. ISubject notifies all IObservers it has registered, and in this case ShapeDrawer observes ShapeList.

The biggest problem Observer pattern solves is that it allows for automatic update of the canvas, without having to make manual calls outside of the classes. Every time a shape is added or removed from the ShapeList, the canvas updates automatically. Dependency Injection Principle is adhered to in this pattern, as there is a layer of interfaces between the concrete classes. Moreover, if there was a need to add another observer to ShapeList, it would not require modifying any code that already exists (Open-closed principle).

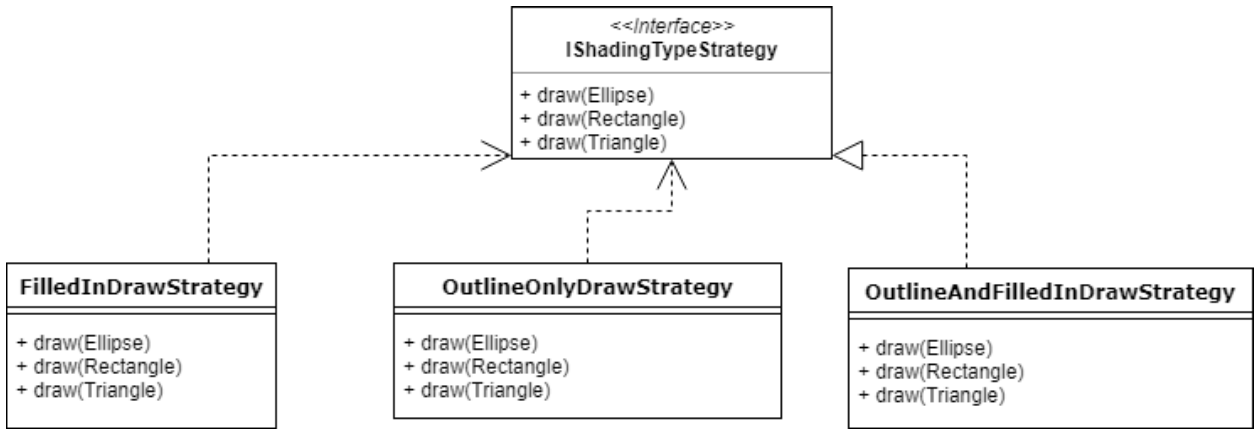
Abstract Factory


CreateShape
<<Interface>>
IShapeFactory
+ createShape(ShapeInfo, ICanvas) : AShape
AShape
EllipseFactory
+ createShape(ShapeInfo, ICanvas) : AShape
RectangleFactory
+ createShape(ShapeInfo, ICanvas) : AShape
TriangleFactory
+ createShape(ShapeInfo, ICanvas) : AShape
Ellipse
Rectangle
Triangle

IShapeFactory implements the Abstract Factory Pattern. It provides a public method for creating shapes that the interface implementations will define as appropriate. AShape is the abstract class that these abstract factory classes will produce. The pattern implements individual factory classes that are responsible for creating the individual concrete subclasses of AShape.

Abstract Factory pattern solves the problem of abstraction as it provides an interface to group together creation of similar objects and client can use it without having to specify the concrete type. This pattern also encapsulates the logic for creating each shape from the contextual class CreateShape. The Dependency Injection Principle holds strongly here because more shapes or shape factories can be added without having to change the structure of these classes.

Strategy



IShadingTypeStrategy class implements the strategy pattern by delegating the drawing algorithm of each shape to concrete classes that relate to each ShadingType Enum. The client assigns every shape with a strategy when the shape is constructed. Whenever the shape needs to be drawn, simply the draw method of the bound strategy is called.

This pattern looks very similar to Template in this case, mainly because each shape is an abstract class. Although, it made sense to delegate the algorithm outside the abstract classes because of the multitude of combinations possible between the concrete strategy and concrete shape. Using the Strategy pattern solves the problem of abstraction since there is a layer of interfaces between the concrete strategies and shapes. The pattern also solves the problem of encapsulation since the implementation of the draw methods are well hidden and can easily be reused.

Successes and Failures

What went right?

- The Command, Abstract Factory, Singleton, and Observer patterns were the easiest to construct.
- The Dependency Injection Principle was the easiest to keep in mind and follow through on.
- Creating the “pipeline” for carrying the appropriate information starting from main all the way to the individual shapes using Command, Observer, various Factories, Strategy patterns made the application easy to test and add features to.
- Used Dependency Injection Principle and created ICanvas interface to pass around the reference to PaintCanvas.
- Added a component listener to GuiWindow to listen for window resizing event and draw all the shapes as they were.
- Fixed the bugs that would flip the orientation of the shapes if the click + drag direction was reversed (except for Triangle)

What went wrong?

- Despite being confident in my “pipeline”, the AShape abstract class is where the application runs into some design problems. First, it doesn’t follow Single Responsibility Principle. It often became the catchall since I followed the “story” of what happens when user does something and AShape was towards the end of the story.
- MoveShapes Command was the toughest to implement, and it still has bugs of creating extra shapes if clicked twice.

Design issues

- The strategies for ShadingType have code duplicates, but I wasn’t sure how to minimize that. I was trying to delegate the drawing to ShapeDrawer using Strategy pattern and follow open closed principle. Future refactoring would probably have to combine Strategy, Proxy and Builder patterns to avoid a lot of code duplication between the three shading strategies.