James Valles

SE 450 – Final Report

Program: JPaint version 1

Overview

JPaint is a Microsoft Paint like application that allows users to draw three basics shapes, ellipse, rectangle, and triangle in a variation of different colors and shading types. These shapes can be selected, moved, copied, pasted, and deleted from the application's canvas. The program is written in Java and utilizes five design patterns: Command, Factory, Observer, Strategy, and Singleton. The following is an overview of the application's development process, including successes and failures, notes on design, and a list of features.

List of Features

Draw: JPaint allows the user to pick and draw the following shapes to canvas: ellipse, rectangle, and triangle. Ellipse is the default shape and draw is the default Start and End Point Mode, which will run the ShapeDrawCommand. The application allows you to pick and draw shapes in different colors. The primary color is the shape's outline. The secondary color is the shape's fill. Shapes can be drawn to the canvas using three Shading Types: Filled_In, (this is the application's default setting), which draws a filled-in shape using the application's secondary color selection. Outline draws a shape's outline with no fill using the program's primary color selection. Both Filled_In and Outline draws a shape that is filled-in and outlined using both the selected primary and secondary color. To draw, while in draw mode, simply press on the canvas and drag. Aside from the basic draw capabilities, I have also included, as extra credit, the ability to right click mouse to activate drawing a shape in flipped primary and secondary colors. To undo this feature and revert back to the original color primary and secondary color selections, simply click the mouse's left button. This feature was tested on a Mac PC. Since I only own a Mac, I was unable to test on Windows PC.

Select: The select feature is used to select specific shapes to either move, copy, or delete. All shapes must be selected prior to running move, copy or delete. To use the select feature the Start and End Point Mode must be set to select. This will run the ShapeSelectCommand. To select a shape simply click on the shape, this will run the contain method in the shapes' draw class (DrawEllipseShape, DrawRectangleShape, DrawTriangleShape) to determine whether the mouse point is contained within the selected shape. Once a shape is selected, if point is contained within shape, shape will be added to the selectedShapesList array list in the ShapeList class. To select additional shapes, while in select mode simply click on the additional shapes. To reset selected shapes, click anywhere on the canvas that does not contain a shape. Only selected shapes can be moved, copied, or deleted; therefore, user has to toggle back and forth between modes.

Move: The move feature allows a user to move selected shapes on canvas. To move, Start and End Point mode must be set to move. Keep in mind, move will only move those shapes that have been selected while in select mode. To select other shapes, user must enter select mode, clear selected shapes by clicking anywhere on canvas other than a shape, reselect shape(s) and then set mode to move. The move feature works for all shapes, Ellipse, Rectangle, and Triangle.

Copy: While in select mode, select the shapes you wish to copy. Press the Copy Jbutton on canvas to activate the CopyCommand, which will copy selected shapes to clipboard array list in the ShapeList class. Only selected shapes will be copied.

Paste: While in select mode, select the shapes you wish to paste. Press the Paste Jbutton on canvas to activate PasteCommand, which will essentially create a duplicate copy of all selected shapes(s). To rerun paste, user must enter select mode and reselect shape(s) each time.

Delete: While in select mode, simply select those shapes you wish to delete. To delete selected shape(s) from canvas, click delete, which will run the Delete Command.

Undo: Undo allows a user to undo the previous operation using the CommandHistory class. This feature will undo a shape being drawn, deleted, pasted, or moved. You do not have to use select for this feature to work, simply click undo.

Redo: Redo, the opposite of undo, allows a user to redo the previous operation using the CommandHistory class. This feature works for shapes that are drawn, deleted, pasted, or moved. You do not have to use select for this feature to work, simply click redo.

Extra Credit / Additional Features

In addition to the basic features mentioned above: I also added the following application enhancements:

Clear All: User can click on the Clear All Jbutton, in any mode, to clear all shapes displaying on the canvas. This will reset the selectedShapesList, the clipboardList, and the main internalShapeList, which PaintCanvas uses to draw all shapes to canvas.

Draw w/ flipped colors: Click mouse's right button to draw shape(s) using flipped primary and secondary colors. This works in draw mode only. To activate, simply right click, and then draw. To revert back to original colors, simply left click on mouse and draw. This feature worked on Mac PC and Mac mouse. Console output will display "right" and "left" clicks to help you keep track of your clicks.

Unit Tests: Unit Tests under main package tests correct Shape Configuration output, ensures that getters and setters are working correctly. Additionally, it ensures ColorAdapter is correctly mapping ShapeColor to Color. Moreover, the last two tests ensure that the ShapeFactoryCommand along with factory's shape configuration, and shapeList is functioning properly.

Additional design pattern: The project required four design patterns. I have implemented Command, Observer, Factory, Strategy. And, as extra credit the Singleton design pattern in the class ColorSingleton.

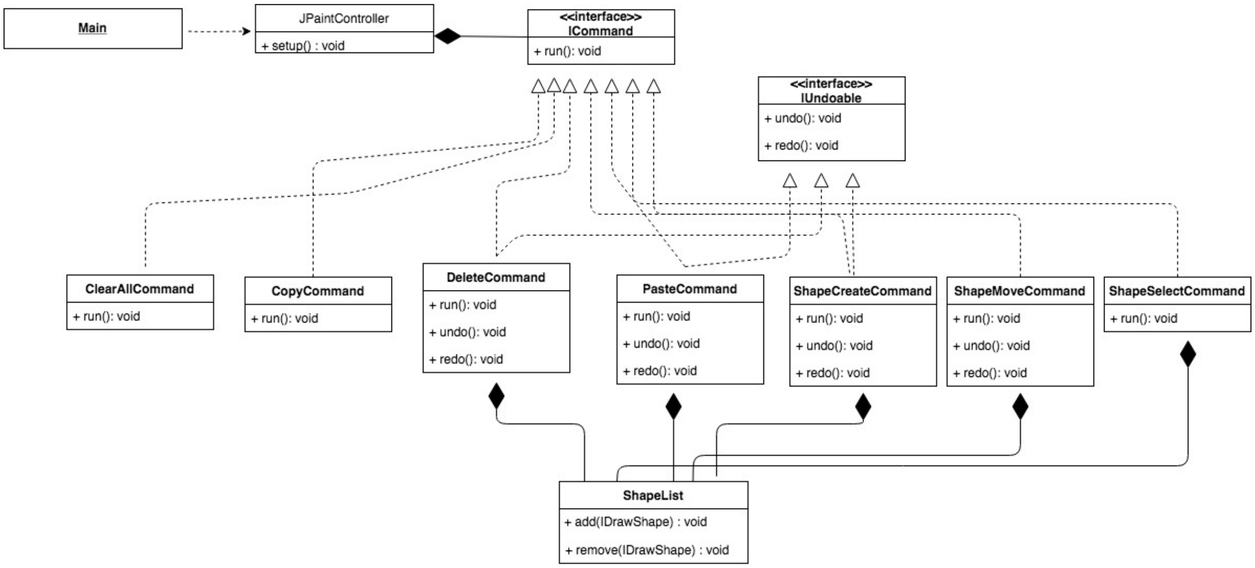
Change color once shape is drawn: Please **test this feature last** as undo and redo functionality hasn't been added. And, it doesn't work properly with paste as the old color is pasted. This feature will change the selected shape's primary, secondary, shading type, once drawn. To use, change mode to select, and select the shape, once you change the shape you are able to change its shading type, primary color, or secondary color. You must reselect the shape each time, you want to change its shading type, primary color, or seconday color. Worst paint program ever. Files involved: ColorObserver, IColorObserver, ApplicationState, ShapeList

Bugs:

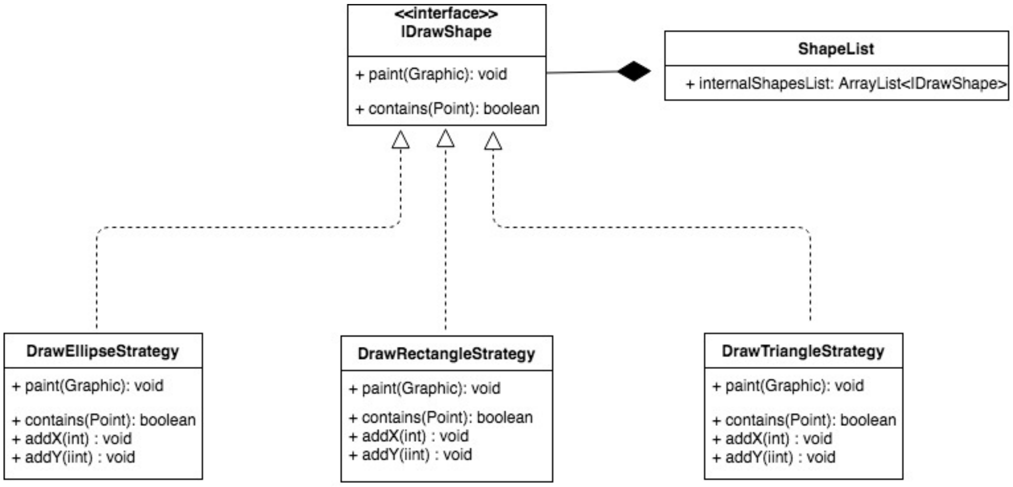
There are two known minor bugs, which I would like to fix in an upcoming release. While paste works. It does require user to re-select the original selected shape each time to continue pasting. You don't have to toggle the StartPoint/EndPoint mode, just select the shape and hit paste. This is annoying. Moreover, while move works correctly. There appears to be a bug with undo, while in move mode, the original shape is removed completely and does not appear. Redo appears to function correctly.

Notes on Design

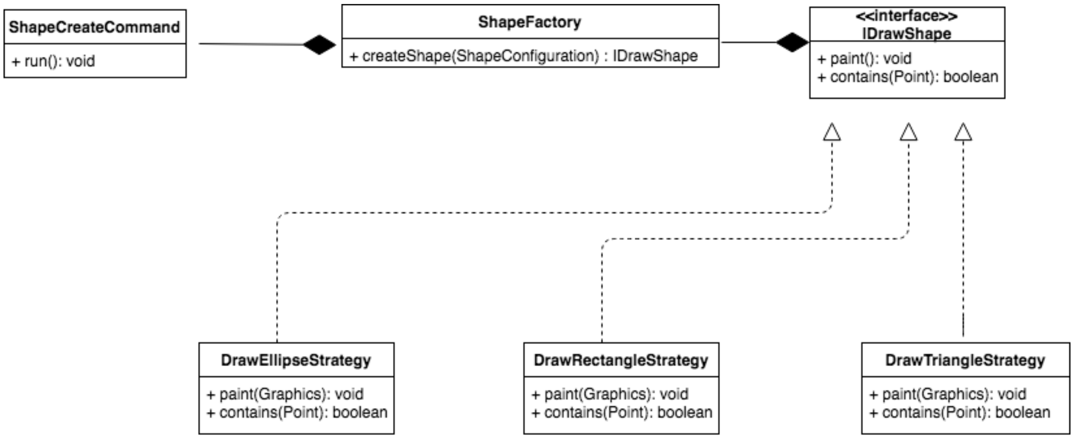
Command Pattern: The application uses five design patterns. It relies heavily on the Command Pattern, a behavioral design pattern, that provides loose coupling and encapsulates a request to either draw, select, move, copy, paste, undo, redo only exposing the run() method and encapsulating everything else. Using the command pattern in this project, made it easier for me to add new commands throughout the development life cycle without having to change existing code making it easier to build out new future features. The command pattern hides the inner details (code) about the command exposing only the call to execute the command. This allowed me to change the functionality of the actual command without having to modify the invocation code. Classes involved include: ICommand interface, ShapeCreateCommand, ShapeMoveCommand, ShapeSelectCommand, PasteCommand, CopyCommand, DeleteCommand, Redo/Undo Commands, ClearAllCommand and the mouse adapters which create and run a command dynamically. All command classes include a run method with details the functionality of the command. Based on the Start and End Point Mode as set in the ApplicationState when Mouse is released, the mouseAdapters will create a specific command dynamically and call the run() method. This alleviates the problem of code re-use, and having multiple if-else conditional statements in a class, which would make it extremely harder to scale or modify in the future.



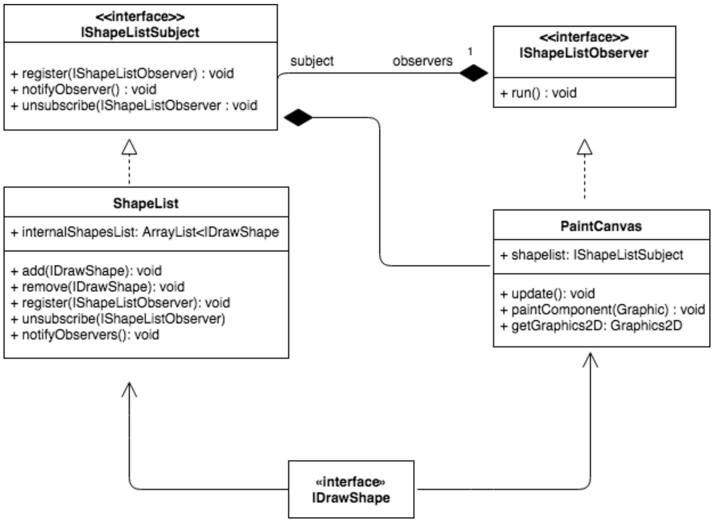
Strategy Pattern: The strategy pattern, a behavioral pattern, essentially encapsulates algorithms, and allows them to be interchangeable at run time. The abstraction is contained in an interface, and implementation details are found in the classes. In this project, I used the pattern to draw shapes. The interface is IDrawShape, the classes are DrawEllipseStrategy, DrawRectangleStrategy, DrawTriangleStrategy. The common behavior here is draw, except that different shapes are being drawn. The strategy pattern is the perfect pattern to address this problem as it separates what is different from what attribute stay the same. For one, it allowed me to get rid of using complicated and long switch, if-else statements. It also allows for my code to be more maintainable, and extensible, which saved me time and headaches. The pattern is a 1 to many relationship and allows me to add more shapes when needed. Moreover, the pattern allowed me to move behaviors into specific classes, which follows the single responsibility principle. It also allows us to follow the Liskov substitution principle as we are not putting all properties into one class. The code allows for design using Interface segregation.



Factory Pattern: The Factory pattern is a creational pattern that is used to create a specific shape whether it's an ellipse, rectangle, or triangle dynamically. It provides flexibility, loose coupling, allows for mock unit testing and the open/closed principle. Furthermore, it allowed me to make changes to the application without changing the dependent code. The pattern allowed me to encapsulate object creation logic which provides for an easier means to change creation logic in the future. Depending on the shape type selected by the application state, the factory will create a new shape object, which is invoked by the ShapeCreateCommand dynamically at run-time (polymorphic creation). This addresses the issue of needing to construct several different objects, which relies on several other objects. Since we are creating several different shapes the client doesn't need to know the creation details. Object creation is handled by the classes (DrawEllipseShape, DrawRectangleShape, DrawTriangleShape, and the interface IDrawShape (which are a part of the strategy pattern). The factory pattern allows us to choose which strategy to run at run-time. The factory method ensures the code is open to change where I can easily add new shapes. The Factory pattern is often defined as "A method whose sole responsibility is to abstract the creation process of an object and return that object."



Observer Pattern: The Observer pattern, which is a behavioral design pattern, was extremely useful in notifying the paint canvas that a new object had been added to the internalShapeList so that new shapes could be either removed or drawn to the canvas. Additionally, it was used to notify the mouse adapters when the StartAndSelectMode changes, which would lead to a new command to run. The Observer pattern allows dependent objects to be notified automatically when changes are made. The pattern provided many benefits in the development of this application. For one, it allowed for loose coupling between objects. Data transfer is done effectively without having to make modification to the Subject and Observer classes. And, Observers can be added/removed as needed. The observer pattern works much like a newsletter subscription platform, when a new e-newsletter gets published everyone on the list gets notified. To implement this pattern, I had to create two interfaces, IShapeListObserver and IShapeListSubject, and add the following methods to the subject – registerObserver, notifyObserver. This set up was used in the ShapeList so that each time a new shape gets added to the internalshapelist, it would the notify the Paint Canvas class (an observer) that a new shape has been added which then runs update() which calls JComponents repaint() method. I also used the Observer pattern in the MouseAdapter classes. Each time the application state's setActiveStartAndEndPointMode is run, the MouseObserver class, which implements the IMouseAdapter Observer is notified, which then runs() code to add a mouse listener to the paint canvas with the correct select, move, or draw mouse adapter (MouseDrawAdapter, MouseMoveAdapter, MouseSelectAdapter).


<<interface>>
IShapeListSubject
+ register(IShapeListObserver) : void
+ notifyObserver() : void
+ unsubscribe(IShapeListObserver : void
subject
observers
1
<<interface>>
IShapeListObserver
+ run() : void
ShapeList
+ internalShapesList: ArrayList<IDrawShape
+ add(IDrawShape): void
+ remove(IDrawShape): void
+ register(IShapeListObserver): void
+ unsubscribe(IShapeListObserver)
+ notifyObservers(): void
PaintCanvas
+ shapelist: IShapeListSubject
+ update(): void
+ paintComponent(Graphic) : void
+ getGraphics2D: Graphics2D
«interface»
IDrawShape

Successes & Failures

Overall this project was the most challenging assignment I have worked on since starting my studies at DePaul and the pain was well worth it in the end. While the application can be improved, there are several things that went right. My understanding of programming in Java has greatly improved. I managed to implement all project requirements: draw, select, move, copy, paste, undo and redo commands. I managed to successfully integrate five design patterns and learn the importance of decoupling, design patterns, how and when to use them, unit tests, the SOLID principles, refactoring, the importance of using interfaces, and avoiding duplicate code, including repetitive if-else statements, etc. The patterns were extremely useful during the evolution of the application. Had it not been for the implementation of design patterns, I would not have been able to complete the project by deadline as the constant additions of new commands, shapes, adapters, subjects would have led to an unnecessary amount of code involving revisions and conditional statements. In order words, things would have become messy quick and hard to modify. I really found creating a Shape Configuration object in the application's Application State to be helpful. Creating this class allowed me to pass the shape's shape type, shading type, primary color, secondary color, start/end points without having to use long and drawn out constructors. This worked well.

I would say the hardest part of the project was figuring out how to get things to work. It took me a while to understand what controlled what and decide on which design patterns to use. Understanding Swing and Awt definitely involved a learning curve. And, to successfully build the application, it was essential to understand how everything fit together. This became overwhelming as there were too many things to learn at once. I really struggled to understand how to get draw to work, in the beginning, how to use the observer pattern to notify the paint canvas to draw when a new shape was created and added to the internal shape list. I spent hours trying to understand the role of the application state, and using dependency injection to pass around objects. But, in the end, I figured it all out. This I consider a success.

What did go wrong was trying to resolve the application's identified bugs. First, undoing a move. I spent days trying to get this bug fixed, which included researching object copying (shallow, deep, etc), trying new things, but sadly to no avail. Redo works. But, undo results in the shape disappearing. Another failure was Paste. While it works perfectly when the first paste command is executed, it does not work when subsequent paste commands are called on the selected shape, unless you specifically click on the first selected shape. Then it works great. This is frustrating. I also spent days trying to resolve this without a resolution.

Despite the minor bugs, I feel great about what I have accomplished and learned. There is still much more refactoring and encapsulating that can be done. Overall, not only did this project require me to take my Java skills to the next level, it forced me to think and make specific design decisions.