SE 450: Object-Oriented Programming – Final Project

List of Missing Features and Bugs:

Overall, this JPaint program runs perfectly well in most circumstances. Basic functionalities when working on a shape-by-shape basis: drawing, selecting, moving, copying, pasting, deleting, grouping/ungrouping, and Undo/Redo work as intended. There is a small bug where it is necessary to double click a certain command, in order for it to register. However, the larger bug of the program has to deal with groups. There is a graphical bug when it comes to grouping of objects; a selection outline is visible for unselected groups. Lastly, when working with groups of groups, this property of groups doesn't behave as described in the rubric.

Link to my Github:

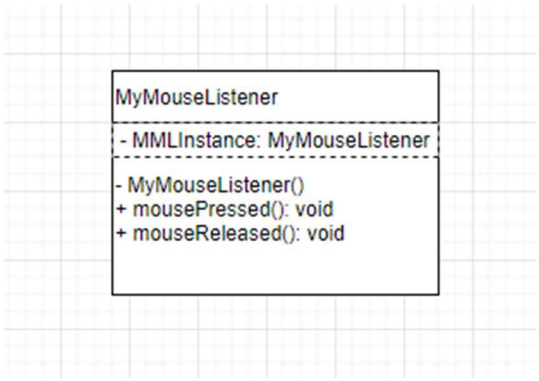
Account: https://github.com/bcole1409

Repository: https://github.com/bcole1409/BC-JPaint

Notes on Design Patterns:

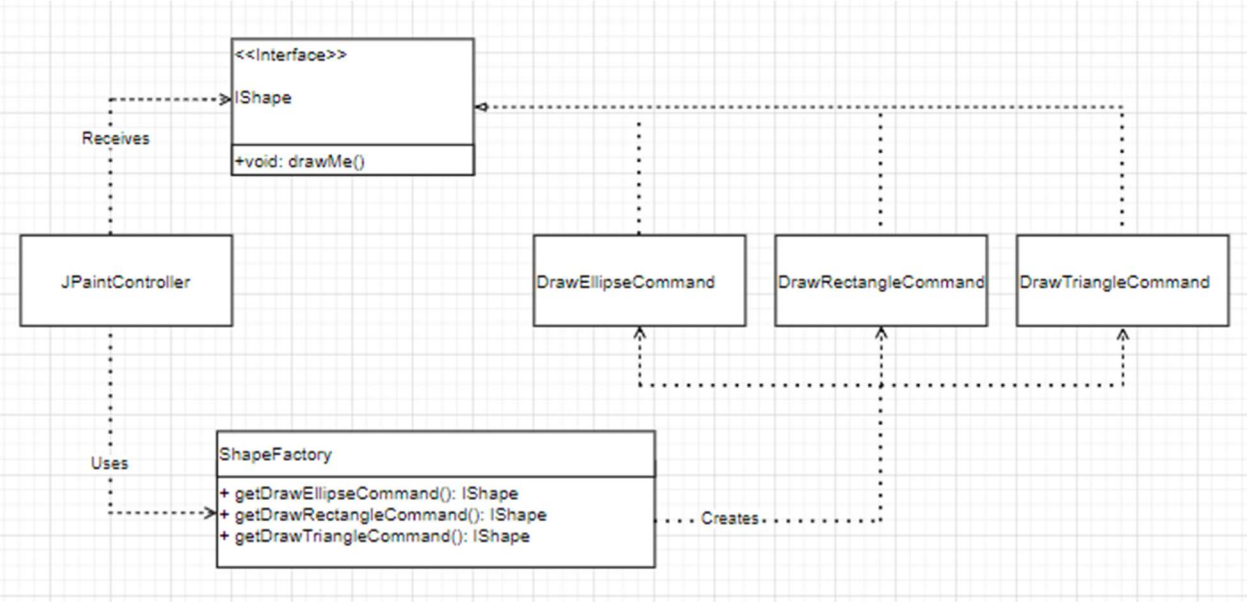
1) Singleton Pattern – MyMouseListener:

The first pattern I used was the Singleton Pattern implemented in "MyMouseListener.java." "Main.java" creates the instance of "MyMouseListener" and passes a reference to JPaintController. Mouse Events are then detected by "MyMouseListener" and then it passes raw mouse data to "JPaintController," via the helper function MouseReleasedController( ). I choose to implement the Singleton Pattern for this class after considering several options. My candidates were all classes where it would make sense for there to be only one instance of the class. I considered JPaintController and CommandHistory as candidates, but decided that reimplementing these to fit Singleton Pattern, would require a non-trivial amount of refactoring. And so, I selected the "mouseListener" because it required the least refactoring to implement the singleton Pattern. The problem that this Singleton implementation solved was that mouse events should only be detected by one class instance in our architecture. If multiple instances of that class are created, then multiple mouse events will be sent to the rest of our system for handling when the user creates just one single event. This would create all sorts of bugs and break the invariant that a single user input will be treated as a single user input.



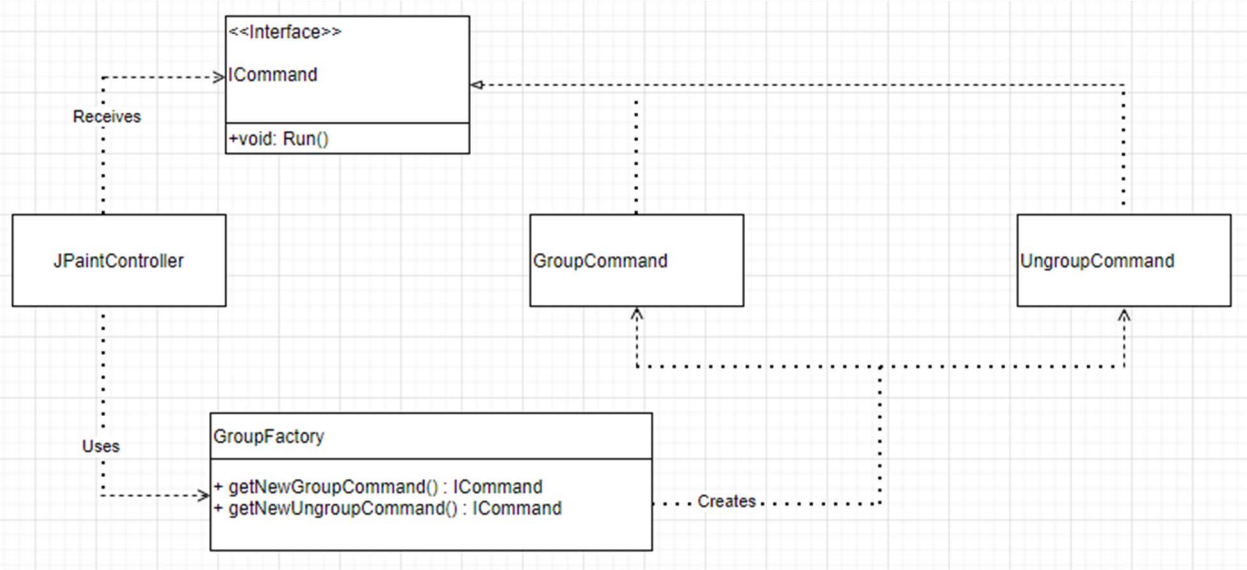
2) Static Factory Pattern – DrawCommands

The second pattern I used was the Static Factory Pattern implemented in "ShapeFactory.java." "JPaintController" calls the static methods of "ShapeFactory.java" to create new ShapeCommands. (For example: drawRectangleCommand.java, drawTriangleCommand.java, drawEllipseCommand.java – theses children classes all inherit from their parent class ShapeCommand.java). The factory pattern is very effective when implemented in a part of our system that creates many instances of several different classes. The draw code in JPaintController that created new shapes seemed like an excellent fit for the factory pattern. The problem that the factory pattern solves is that it hides the class names (drawRectangleCommand.java, drawTriangleCommand.java, drawEllipseCommand.java) from the client (JPaintController). This is important because the class names may need to change in the future, and also because it adheres to the "principle of least knowledge."



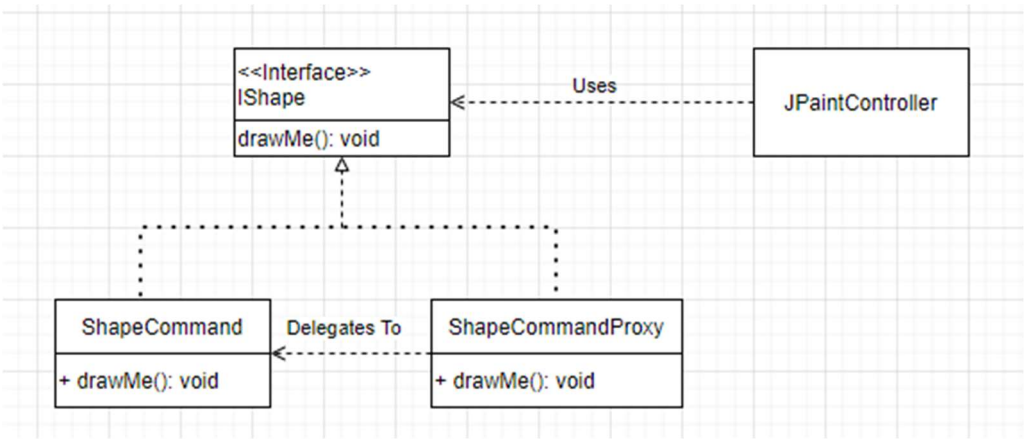
3) Static Factory Pattern – GroupCommands

The third pattern I used was the Static Factory Pattern again, but this time for Groups, implemented in "GroupFactory.java." Similar to pattern 2 above, "JPaintContoller" calls the static methods of "GroupFactory.java" to create new Group Commands. (For Example: GroupCommand.java and UngroupCommand.java) The factory pattern is more straight forward and easier to implement than some of the other more complex patterns covered in lecture. I wanted to be sure that implementation was clean and bug free, and so, for my choice of implementing one pattern twice, I chose the factory pattern. Reading through the code base I recognized that instances of Group and Ungroup Commands were created in a similar fashion to all of the instances of ShapeCommands. And so, for the same reasons stated above in pattern 2, I chose to implement the Static Factory Pattern for Group Commands. Again, the problem the Static Factory Solved is that it hid the class names of the Group and Ungroup Commands from the client (JPaintController). This is important because the class names may need to change in the future, and also because it adheres to the "principle of least knowledge."



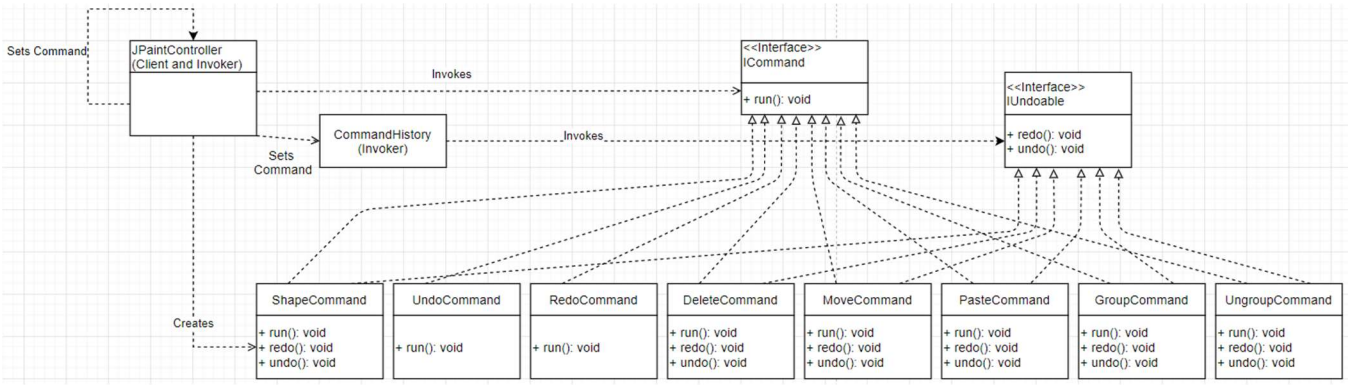
4) Proxy Pattern

The fourth pattern I used was the Proxy Pattern implemented in "ShapeCommandProxy.java." "ShapeCommandProxy.java" is a wrapper for "ShapeCommand.java," so it contains an inner instance of ShapeCommand. The core functionality of "ShapeCommandProxy" is to draw a dashed outline around a shape to indicate that that shape is currently selected. This is achieved when JPaintController handles selection (and thereby creates instances of ShapeCommandProxy). Proxy patterns are best used when trying to add functionality to another class. The dashed outline indicating selection was an excellent candidate for the proxy pattern because I had already implemented Shape Command Classes and needed to add this additional functionality onto these classes. The proxy pattern is useful here because it implements the same interface as the subject class (ShapeCommand.java) so it can used in place of the subject without the client knowing.



5) Command Pattern

For the fifth pattern I chose the Command Pattern. This is implemented by all of the command classes: ShapeCommand, DeleteCommand, MoveCommand, PasteCommand, GroupCommand, UngroupCommand, RedoCommand, and UndoCommand. JPaintController creates instances of Commands, which are then added to "CommandHistory.java" (which manages the undo and redo command stacks). This pattern was chosen because the example code provided in lecture showed a clear and straight forward way to implement a CommandHistory and the best way to achieve undo/redo functionality in this project was to have a CommandHistory, and therefore the Command Pattern. This pattern is highly useful, because it wraps up functionality into an object that can be run at a later time (for example: when the user clicks Undo/Redo).



<u>Successes and Failures:</u>

Overall, this project was the hardest assignment I've had as a Computer Science student; however, I firmly believe this project has given me a strong foundation for Software Engineering and where ever my future takes me in programming.

With success, comes pitfalls, and there were quite a few when starting this project. My familiarity with creating a program that interacts with a GUI was foreign to me, so getting acclimated right away was tough. Getting started with the "MouseListener" took me a few days to understand the process of recording raw data and somehow turning that into Points, and then somehow creating

shapes based on those points. Putting this all together was extremely hard for me at the beginning, and ultimately put me behind in the first couple weeks of this project.

The next pitfall came from understanding design patterns and knowing how to implement them. Knowing what classes and interfaces would work for certain patterns was hard for me to grasp. Considering there were many patterns to choose from, at times, it was confusing to understand which pattern would work best in certain scenarios. At times, I would implement a pattern, then halfway during the implementation process, I'd realize that maybe this wasn't the correct way to go about implementing a pattern. As a result, I'd have to go back to a clean slate, and try a different pattern.

Lastly, the greatest pitfall for me was trying to correct bugs that came from refactoring code. I'm used to writing code that deals with fewer classes/interfaces, not so much with a large program that has 20+ classes/interfaces. So, a project of this magnitude was quite new to me, and for that reason, distinguishing where bugs were coming from was very hard. For example, at one point in my project, single shapes, when drawn, were creating duplicates in the "drawList". I scanned the code for many hours trying to find where this bug was occurring. In some obscure class, there was a function call that shouldn't be happening, which was causing this simple, yet, problematic bug.

I firmly believe the successes of this project outweigh the downfalls. One major success, is that I felt I structured my code in such a way that it was easy to go back and understand where to make changes. The way I structured my code, made it easy to refactor and less likely to change existing code, which was a huge time saver. If I didn't follow the SOLID Principles, every time I had to implement new functionality to the existing code, it would've taken much longer.

Another accomplishment, was that I completed all the criteria for this project on time. Considering this was an extremely in-depth assignment, I was able to always stay on top of things and turn in quality code each Sprint. Time management was extremely important for this project, and always staying up on feet and not procrastinating helped in the long run.

Lastly, the greatest takeaway for me was that my programming is now much stronger, and more simplistic. The way I structure classes, functions, methods, and whole programs are better off having taken this class.