Before we begin going over the programming abomination that is my copy of jPaint, I think its important to take the time to recognize to recognize the developer of the application. Who am I you ask? LinkedIn says I’m a 31-year-old software engineer whose been at S&C Electric for 3 years (crossed that milestone on the 14th this month!). I’ve been working professionally as a software engineer since the summer of 2013. I did some professional level Object Oriented Programming early on in my career from 2013-2016 with C# and Java. Prior to that, I had only worked with these languages and concepts at an academic level in undergrad from 2010 – 2013. Over the course of this report, I will cover what went right and wrong from a development standpoint, the design patterns I used to handle the various aspects of the project, some important functional information about the project itself, ultimately what I’ve learned from this project, and what I took away from it.

Over the course of this project, I encountered various development trials and tribulations that helped or hindered the pace of development. In this segment, I will be covering what went right, what went wrong, and what can be done better for next time.

Within Sprint #1, taking the time to setup a proper development environment for this project proved to have major advantages. Using IntelliJ IDEA got me super comfortable with Java’s syntax, and naming conventions, which I always struggled with early on with my exposure to Object Orientation. Over the course of all sprints, I ran into various runtime issues that hindered development pace, but they were all easily resolved by google searching, and using the debugging tools provided from IntelliJ like breakpoints, stepping through lines of code, debug print statements, and using the watcher tool.

Just by looking at the simplicity of what went right, its safe to assume that more went wrong, and by that assessment, you’d be right! I would identify 3 key things as issues that are identified as key failures, and they would be the scrapping of code early on in Sprint #1, and getting color to work in Sprint #2/3, and the various bugs that were never resolved over the course of development.

The first thing that went wrong on this project was having to scrap my point implementation early on in favor for the Point object provided by AWT. I lost probably an evening’s worth of progress during that incident, but it reminded me to be more thorough about reading documentation when it comes to using libraries or tools that I am unfamiliar with. That scrappage of work could have been avoided if I had been proactive about reading things ahead of time. The second thing that went wrong on this project was getting coloring shapes to work properly. It literally took 12 commits of work over 6 days for it to work. The time it took to get it to work during Sprint #2 delayed the “Move Shape” functionality, which then never really worked properly, and continues to propagate issues upon the completion of Sprint #4.

Other issues that popped up were due to me not reading/rereading/remembering project requirements, when it came to selecting shape and that’s why that function also still doesn’t work. The delete function currently deletes the last shape drawn, and not the selected shape. If I had reached out to other students earlier for help, I wouldn’t have fallen behind in functionality.

At no point in my career before this class have I ever heard of a design pattern, so I’m going to take a moment to address what a design pattern is and why its important, before I go into the design patterns I used in my project.

What is a design pattern you ask? I found myself asking this very question a few weeks ago and I think I can totally answer that question. It’s a reusable solution to a commonly occurring problem within a given context in software design. Why is it important? Design patterns can speed up the development process by providing tested, proven development paradigms. The project required we implement 5 design patterns into our project. Over the course of the next couple paragraphs I will go over the design patterns, what is it, how I used it, and what classes it impacted, in the order in which I learned about them.

An Adapter Pattern is a design pattern which makes incompatible objects adaptable to each other. I used it to convert mouse coordinates into coordinate points. MouseAdapter.java used the Adapter pattern to convert mouse position on a coordinate plane into points in which operations can happen based on those inputs.



Figure - Adapter Pattern UML Diagram

A Factory pattern is a design pattern to create different shapes. Depending on the application settings, it will create a shape based on that information. A bunch of shape strategy classes fed into the ShapeFactory.java. A ShapeFactory Object gets instantiated inside Main.java, and then gets passed into MouseAdapter.



Figure - Factory Pattern UML Diagram

A strategy pattern allows you to make changes as run time. I used it for all the different strategies required for drawing the linework for each shape. I had 3 strategy per shape files that fed into IShapeStrategy to draw the different styles of shapes. Based on what is set in Application Settings, the shape will be rendered appropriately.



Figure - Strategy Pattern UML Diagram

A singleton pattern is a way to provide one and only one object of a type at a time. I used it to return color based on shapeColor. I originally had implemented Singleton as a class file but I ended up implementing it inside shapeColor.



Figure - Singleton Pattern UML Diagram

A command object pattern allows you to setup multiple things as one command. I made a command for every function and then fed it into the Interface Shape Command. Of all the various commands that manipulate different aspects, it makes sense to use this design pattern to clean up the selection code for what is being executed.



Figure - Command Pattern UML Diagram

First and foremost, I was not able to complete the project and here are all of my programming shortcomings for jPaint. Starting with missing functionality, I ran out of time and was not able to add the following functionality of Group and Ungroup. I also have some major bugs over the course that haven’t been resolved. I can only move or select the most recently drawn shape. If I had completedthis previously mentioned functionality, I would have liked to setup an automated build pipeline with GitHub Actions, develop a set of UI/Unit tests to be implemented into the pipeline, and add keyboard shortcuts. This workflow of automated building and testing would have enabled me to catch bugs in my code by making them more apparent with automated build reports.

Over the course of this project I learned to really appreciate design patterns as an approach to programming. I don’t do Object Orientation at work, but I want to find a way to implement the concepts I’ve learned here into build and test automation at work. Other things I’ve learned is to not rush head first into things without reading documentation, and take the time to plan out what I’m doing instead of just spinning my wheels trying to cover ground.

**Appendix and Closing remarks**

Project URL on github: <https://github.com/BakedBeanz1111/CrappyPaint>

I’ve attached full sized PDFs of all the UML Diagrams inside a directory called “UML Diagrams” in the root of the project directory.