1. Description

"MultiPixel is an online relaxation video game designed to offer a more enjoyable experience for stressed individuals during their leisure time. Our program is unique as it offers free, easy access to pixel painting software, which can be interacted with anytime, simply through a web browser rather than through a download. We plan to implement several pages for all kinds of uses including but not limited to a free draw page, a page to draw on different art templates, and a workshop page.

2. Architecture

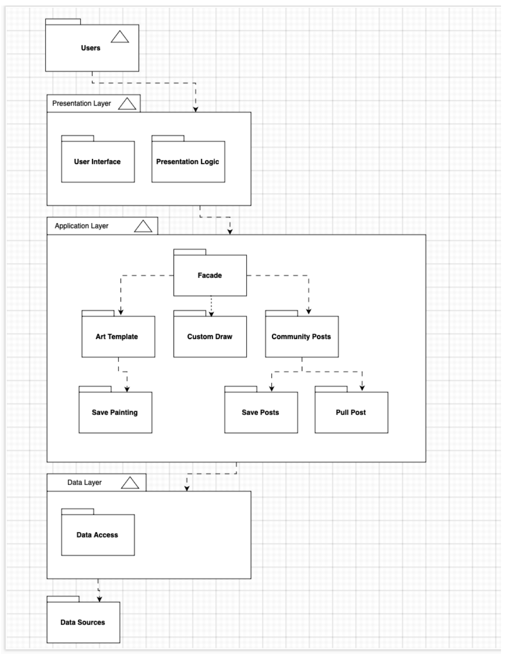


Figure 1: Architecture Diagram

The design of the UML Package diagram and the architecture of our systems has three layers. The first layer is the Presentation Layer accessed by the users. In this layer is all of the User Interface for the website and all the features of the site along with all the logic for the presentation.The logic includes mostly all of the coloring aspects in the main feature of the website, the online pixel painting. Beneath this layer is the Application Layer contains all the other logic and features of the site, the main things this layer contains are the Art Template, Custom Draw, and Community Posts. All three of these are storable information with sub-features below them that involve saving and pulling from each of the three. The final layer is the Data Layer which is connected to the database and allows for storing and pulling data into the database.

3. Class diagram

Present a refined class diagram of your system, including implementation details such as visibilities, attributes to represent associations, attribute types, return types, parameters, etc. The class diagram should match the code you have produced so far, but not be limited to it (e.g., it can contain classes not implemented yet).

The difference between this class diagram and the one that you presented in D.3 is that the latter focuses on the conceptual model of the domain while the former reflects the implementation. Therefore, the implementation details are relevant in this case.

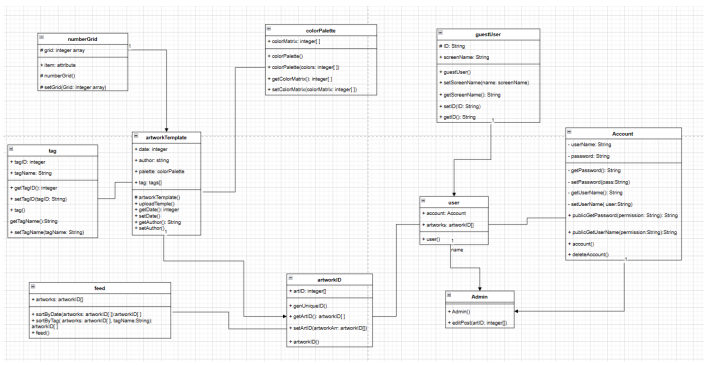


Figure 2: Class Diagram

4. Sequence diagram

Present a sequence diagram that represents how the objects in your system interact for a specific use case. Also, include the use case's description in this section. The sequence diagram should be consistent with the class diagram and architecture.

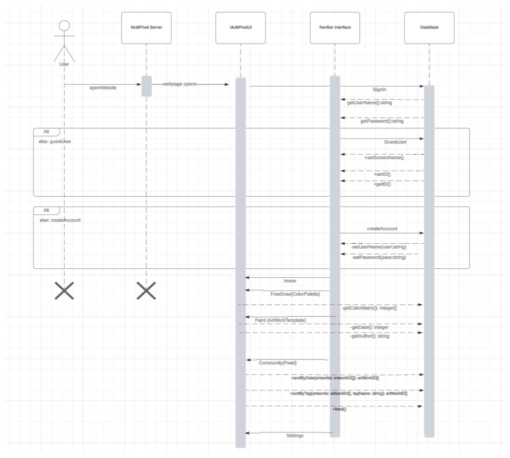


Figure 3: Sequence Diagram

5. Design Patterns

The first design pattern we used was the Structural Design pattern Facade. Since we contain all of our Classes with a wrapper in App.js, we found that we were using this design pattern all along. The links to each class are below the diagram.

MultiPixel App: https://github.com/thomasrotchford/CS386-2024-multiPainter/blob/main/multipainter/src/App.js

Home: https://github.com/thomasrotchford/CS386-2024-multiPainter/blob/main/multipainter/src/Home/Home.js

Community: https://github.com/thomasrotchford/CS386-2024-multiPainter/blob/main/multipainter/src/Community/Commu

NavBar: https://github.com/thomasrotchford/CS386-2024-multiPainter/blob/main/multipainter/src/NavBar/NavBar.js

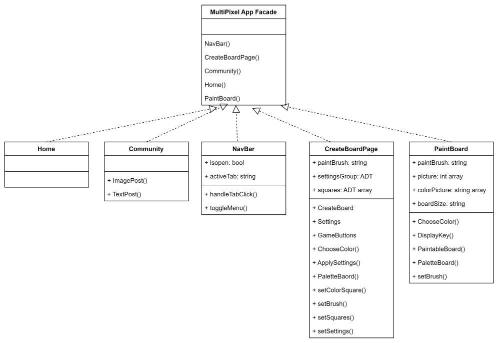


Figure 4: Facade Design Pattern

CreateBoardPage: https://github.com/thomasrotchford/CS386-2024-multiPainter/blob/main/multipainter/src/Create/Create

PaintBoard: https://github.com/thomasrotchford/CS386-2024-multiPainter/blob/main/multipainter/src/Paint/PaintBoardUt

The second design pattern that we captured was a Behavioral Pattern called the Command Design Pattern. We used this because in the scope of our board rendering, we need to pass down functions to objects much lower down the chain. Below, BoardSquare, PaletteBoard, and Settings all mutate the properties of CreateBoardPage such as the squares colors, the settings, and the paintbrush. The individual components need to mutate these so that they can render a complete board and different elements can respond accordingly which is why the properties exist at the top level.

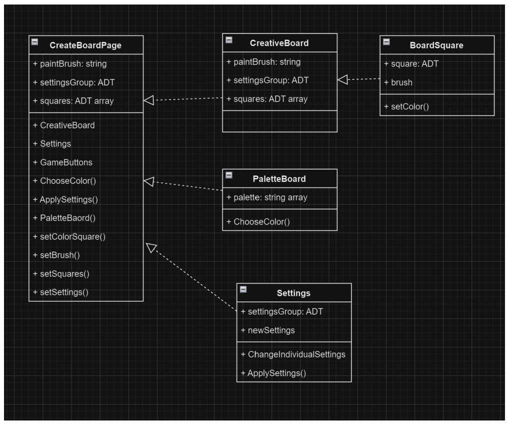


Figure 5: Facade Design Pattern

CreateBoardPage, CreativeBoard, BoardSquare, and Settings are all contained here: https://github.com/thomasrotchford/CS38 2024-multiPainter/blob/main/multipainter/src/Create/CreateBoard.js

PaletteBoard: https://github.com/thomasrotchford/CS386-2024-multiPainter/blob/main/multipainter/src/utilities/Palette.js

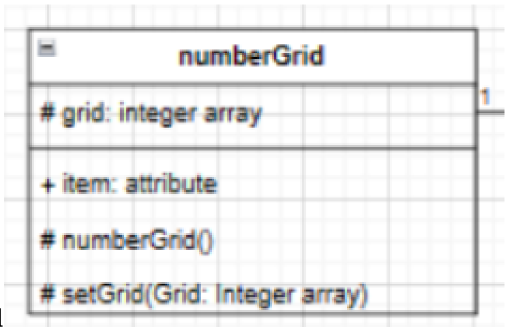
6. Design Principles

When making our classes for our project we made sure to follow the SOLID Principles, Which I will first describe right now, so it can be known that our understanding is concrete

SOLID is an acronym for 5 Design Principles for Developers which are also acronyms themself, which is an interesting example of acronym inception! Robert C. Martin introduced them and have now become fundamental in lots of Object-Oriented Design (OOD). The principles are as follows

The **Single Responsibility Principle (SRP)** states, “A class should have one and only one reason to change, meaning that a class should have only one responsibility". This principle works with the principle of cohesion that we have covered before as it promotes classes working on their easy-to-understand task, so they can work better together as a whole.

A good example of a class we have doing this is **The Number Grid**



The reason our class follows the single responsibility principle is that it fulfills the conditions of only one reason to change and one responsibility. Its one reason to change is the **setGrid()** method, which is called whenever a pixel on the pixel grid is changed. This is the only reason we need to update or change the grid. Continuing, our grid has only one purpose, and that is to display the pixel art that the user is currently drawing

Open/Closed Principle (OCP): This principle states that " Objects or Software entities should be open for extension, but closed for modification". This principle recommends that you should be able to extend the behavior of a class without needing to modify its code. This is best done through the use of user interfaces

A good example of a class we have doing this is **The Relationship Between Artwork Template and Number Grid**

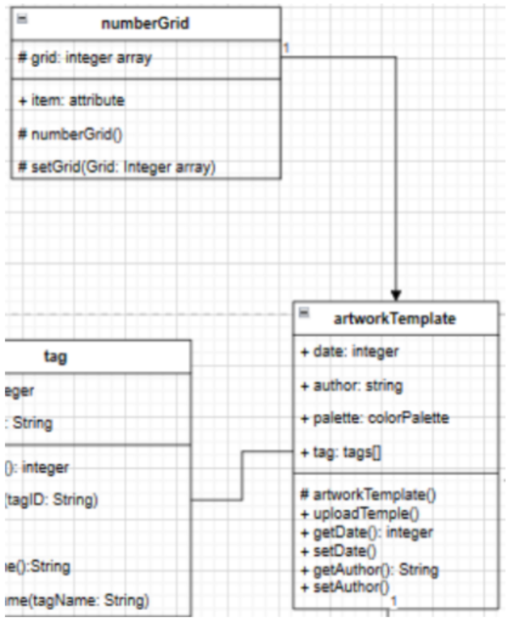


Figure 6: OCP

Our class for the Artwork Template extends from Number Grid. this is to say it inherits some attributes of the grid, and while the Artwork Template does have some of its variables and methods, it does NOT modify the original class. Because of this, we can extend the class behavior of our normal grids into a template grid without changing any code, so we follow the OCP Principle

Liskov Substitution Principle (LSP): This principle states that "Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program." This principle means that derived classes must be substitutable for their base classes without altering the desired behavior of the program. For example, if "Audi" is derived from "Car" you should be able to "Audi" anywhere "Car" is needed without error

A good example of a class we have doing this is **The connection between Guest User Class and User Class**

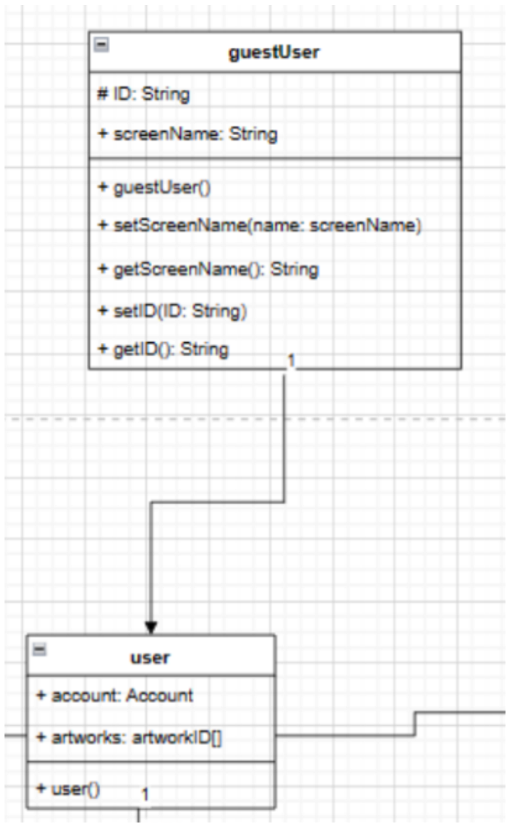


Figure 7: LSP

This satisfies the Liskov Substitution Principle as you can replace the guestUser class with any instance of the user class. We have the classes set up this way because guest users will be what most people visiting the site will use, so that's why we wanted to specify it as the superclass because it will be used so much. Once the user makes an account they will now have a user account and be able to do anything the guestUser can do via the LSP.

Interface Segregation Principle (ISP): This principle states that "Clients should not be forced to depend on interfaces they don't use." This principle suggests that it's better to have many specific interfaces than one general-purpose interface, to avoid having too much of a dependency on one place"

A good example of a class we have doing this is **The Feed Class**

This follows the Interface Segregation Principle as it is an interface that can be used, but it does not need to be used. Because of this, one does not need to rely on the feed if they don't wish to use it. It will be an optional tab, and it does not have any important methods to it so it the user can work completely with or without it if they want. Because of this, our class here satisfies the ISP.

Dependency Inversion Principle (DIP): This principle states that "High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details;

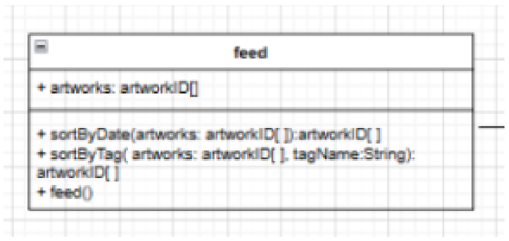

feed
+ artworks: artworkID[]
+ sortByDate(artworks: artworkID[ ]):artworkID[ ]
+ sortByTag(artworks: artworkID[ ], tagName:String):
artworkID[ ]
+ feed()

Figure 8: ISP

details should depend on abstractions". This principle encourages decoupling between modules, allowing for easier substitution of implementations and promoting flexibility and testability.

A good example of a class we have doing this is **The connection between Guest User Class and User Class**

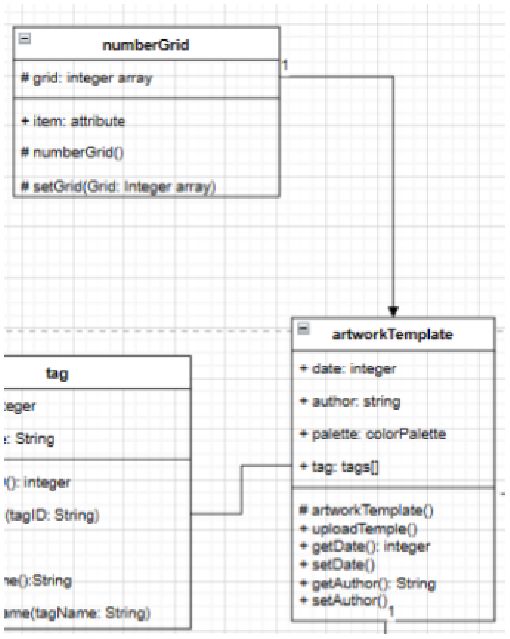


Figure 9: DIP

This example satisfies the Dependency Inversion principle for the following requirements reasons. It is an example of a high-level module (artworkTemplate) that depends on the abstraction of another high-level module (numberGrid). These are both high level as the user will interact with them a lot and through the class methods, they will be able to see their output. The details of these classes depend on the abstractions of some smaller portions such as the **grid** or **color palette.** But neither class relies on anything strictly or concretely, as the smaller details of the abstractions can always be altered

With all these examples together, we can safely say that our design observes SOLID principles and we have a very maintainable, understandable, and robust program.