DrawBerry

CS3217

Final Report

Calvin Chen, Ho Hol Yin, Jon Chua, See Zi Yang

Requirements

Overview

DrawBerry is a competitive multiplayer drawing game. It is inspired by other drawing games currently available, such as Drawful, Draw Something and so on. DrawBerry is developed natively for iOS and allows for single-device multiplayer and multi-device multiplayer. Furthermore, there are multiple game modes for DrawBerry players to play. These game modes are described below.

Classic mode

In each round of this mode, players draw based on a topic set by a particular player, known as the round master. Players will then try to guess the drawing of the round master. The position of the round master then rotates among the players.

Furthermore, instead of having the player draw a certain word/object, by allowing users to enter any topic, this can elucidate different kinds of drawing and also act as a trivia game/friendship test between friends. This also adds some competitiveness element to the game as players can try to deceive each other.

Players can choose between rapid games or non-rapid games. In rapid games, players are given a limited time to draw. They then have to vote for the drawing they think is drawn by the round master. A correct pick will earn them points, and if others incorrectly pick another player's drawing, that player will also earn some points.

Whereas for non-rapid games, players do not have to draw and guess immediately, they can resume the game as and when they like and the game lasts for as many rounds as they want.

Competitive mode

This mode is similar to classic mode, where players can draw anything under the sun. However, this mode is played on a single iPad device and allows up to 4 players. Furthermore, players have to draw their drawings within a single stroke (draw without removing their finger from the iPad).

Powerups will spawn randomly at a random location on their canvas while they are drawing. To activate the powerup, players have to draw onto the powerup. This activates the powerup but risks destroying their own drawing because they have to draw onto the powerup to activate it. Some examples of powerups include blocking other player's view of the canvas partially with an ink splotch, getting an extra stroke for your drawing and so on.

After the time limit, players will vote for the best-looking and second-best drawing. Voting for the best drawing is worth 2 votes while voting for the second-best drawing is worth 1 vote.

After all players have voted, these votes are collated and results are shown on each player's screen. Points are then awarded to each player according to their standings and a new round begins. After 5 rounds, the game ends and the player with the highest number of points is declared the winner.

Cooperative mode

Players will each take turns to draw a portion of the entire drawing. When a player is drawing, he can only see a small portion of the drawing done by the player before him, so as to connect the current drawing with the previous drawing. After every player has completed drawing, the drawings will be pieced together.

Team Battle mode

Players can team with a partner and compete with one another in this mode. Players will be divided into teams of 2, with one player drawing based on a given topic, and the other player guessing the word based on the drawing. Each team will be given 3 topics, so the game will last for 3 rounds. The drawing will only be viewable by the guesser after the drawer has submitted it.

When every team has finished their drawings and guessing, the team score for each team will be calculated and the team's ranking among all teams will be displayed.

Features and Specifications

Features (features bolded and in blue were done in Sprint 3)

1. Drawing canvas
    a. Different colours
    b. Different widths
    c. Different types of brushes/opacity
    d. Undo last stroke
    e. Clear drawing
2. User account
    a. Signup and create account
    b. Login and logout
3. Single-device multiplayer support - Competitive mode
    a. Support for four players drawing simultaneously on one iPad
    b. Powerups:
        i. Hide Drawing
        ii. Extra Stroke
        iii. Ink Splotch
        iv. Invulnerability
        v. Earthquake
    c. Voting and Ranking screens
4. Multi-device multiplayer support - Classic mode
    a. Host room to play with friends
    b. Enter room code to enter a room
    c. Start classic mode game (both rapid/real time rounds and non-rapid rounds)
    d. Enter topic before each round
    e. Timer for drawing in rapid mode
    f. See drawings done by other players
    g. Guess and vote for a drawing
    h. View who voted for a player's drawing
    i. View each players' cumulative points
    j. Results screen at the end of rapid game mode
    k. Pause or join back a round for non-rapid games
    l. See list of active non-rapid games the user is in
    m. View non-rapid game's results at any time
5. Multi-device multiplayer live - Cooperative mode
    a. Host room to play with friends
    b. Enter room code to enter a room
    c. Start cooperative mode game
    d. View players drawing live as soon as they complete it
6. **Team Battle Mode:**
    a. Host room to play with friends
    b. Enter room code to enter a room
    c. Start team battle game

    d. **Word bank to generate random word list**
    e. **Sharing word list across network**
    f. **Drawer draws based on word list**
    g. **Guesser guesses word based on drawing**
    h. **Calculate team result**
    i. **Compare results of all teams and display final game result**
7. Profile feature
    a. View other player's profile in game rooms
    b. Auto save drawings to profile
    c. View other profile's drawings

User Manual

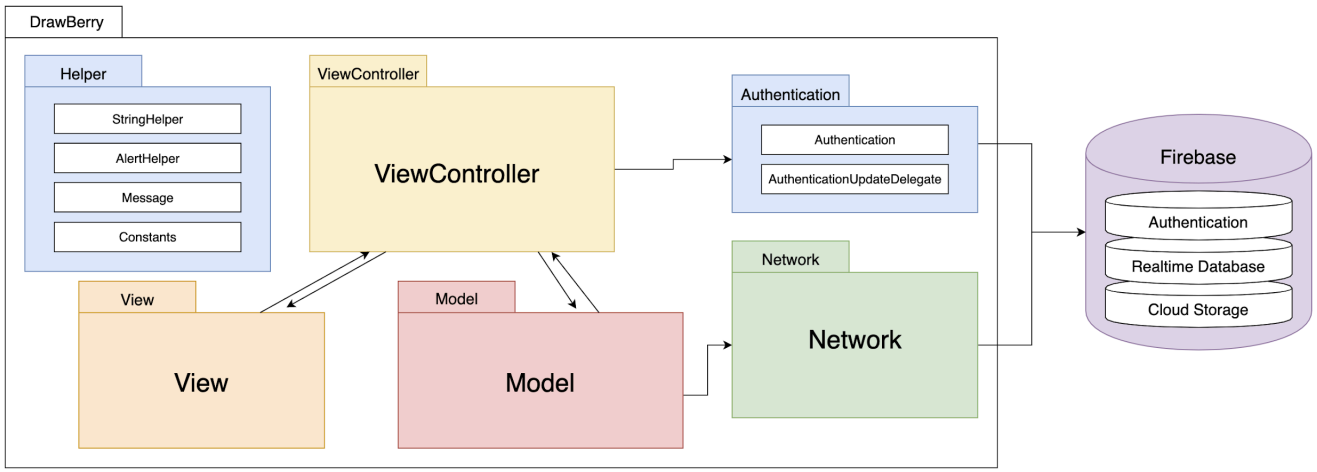
Please refer to the appendix document titled "DrawBerry User Manual".

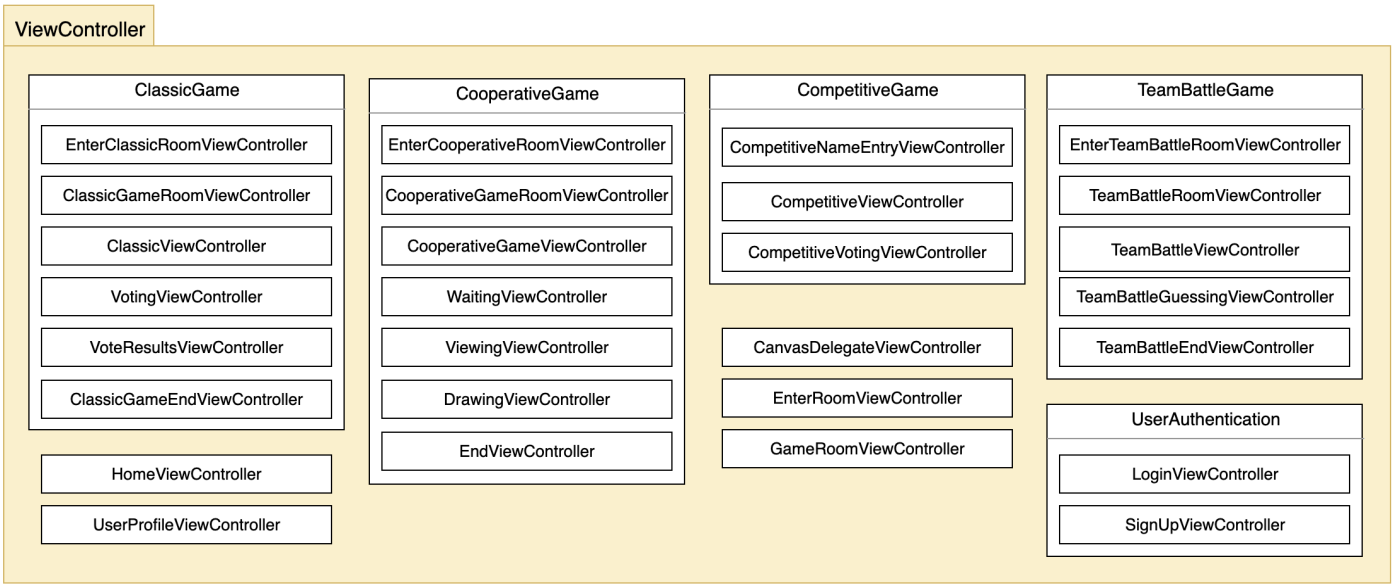
Design

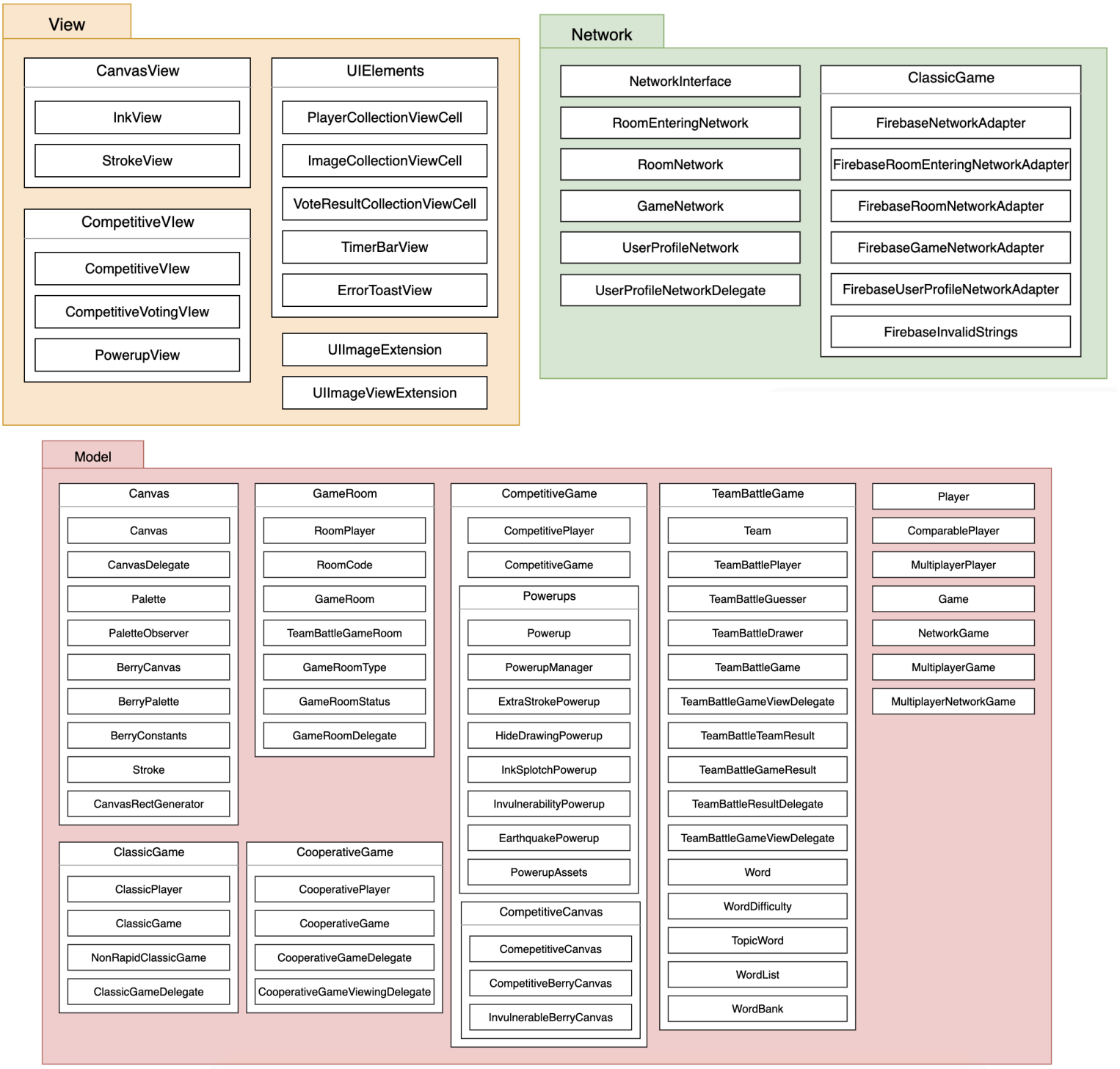
Overview

DrawBerry adopts the Model-View-Controller (MVC) architectural pattern.

Architecture Diagram







DrawBerry also makes use of Google's Firebase for its backend services, including user authentication and the database. As seen from the architectural diagram, aside from the 3 main components, Model, View and View Controller, we have:

- Network component to deal with the real-time syncing of data and game state between online players
- Authentication component to deal with user signups and logins
- Helper component for constants, error messages and convenience functions.

In this sprint, we kept to the same architecture and similar design choices when implementing the new features.

Runtime Structure

Representation of BerryCanvas using PencilKit (Apple's custom drawing library)

We refer to our custom canvas class as **BerryCanvas**. To support our features, we have defined **BerryPalette** (a UIView), which contains the colours, thickness and eraser for the player to choose from. The inclusion of **BerryPalette** will, therefore, affect the choice of our implementation.

Option 1: Apply **inheritance**, making **BerryCanvas** extend from **PKCanvasView**, a UIView from *PencilKit* that detects finger/pencil touches.

In Option 1, we will consequently add **BerryPalette** as a subview of **PKCanvasView** and subsequently break the Liskov Substitution Principle. This is because **BerryPalette** will be a subview of **PKCanvasView** and therefore become drawable, and this unintended behaviour will force us to manually disable the ability to draw in the region of **BerryPalette,** within **BerryCanvas**. Since **BerryPalette** should not even be drawable in the first place, we should not let it be a subview of **PKCanvasView**.

Option 2: Apply **composition**, allowing **BerryCanvas** to contain a **PKCanvasView**.

Composition will allow us to compose the **PKCanvasView** with **BerryPalette**, and this was rather useful for us since we can separate **PKCanvasView** and **BerryPalette** into different components that behave differently. Consequently, we can provide a **Canvas** protocol that is conformed by **BerryCanvas**, which ties these components up together such that they behave as a single entity to the outside world.

We have chosen Option 2.

Representation of CompetitiveView in Competitive game mode

The competitive game mode splits the iPad screen into four equal parts, one for each player to draw. In this game mode, Powerups are added to the game to make it more interesting, as well as a time limit of 45 seconds for players to complete their drawings. In order to draw these powerups and timers on the screen, we needed to make a design decision regarding how the CompetitiveView is structured.

Option 1: The competitive game has one big CompetitiveView which contain multiple subviews of PowerupView

At first, we started off with this option where the competitive game is made up of one big competitive view that composes the entire screen. The competitive view has a set of PowerupViews, where each powerup has its own x and y values relative to the iPad's origin.

However, after adding more features to the application, we realized that this design could definitely be improved. This chosen design requires a lot of conversions to convert between a player's current drawing coordinates (which was given relative to the canvas origin) to the coordinates relative to the iPad's origin. This was a big hint that our design could be split into each player having their own view and thus we switched our design to Option 2, as described below.

Option 2: Each player has its own CompetitiveView which encloses their Canvas and contains multiple subviews of PowerupView

In this design, the View Controller maintains a mapping between each player and their CompetitiveView, which is sized to fit over their canvas. Each CompetitiveView contains a set of PowerupViews, which represent the Powerups available on each player's CompetitiveView.

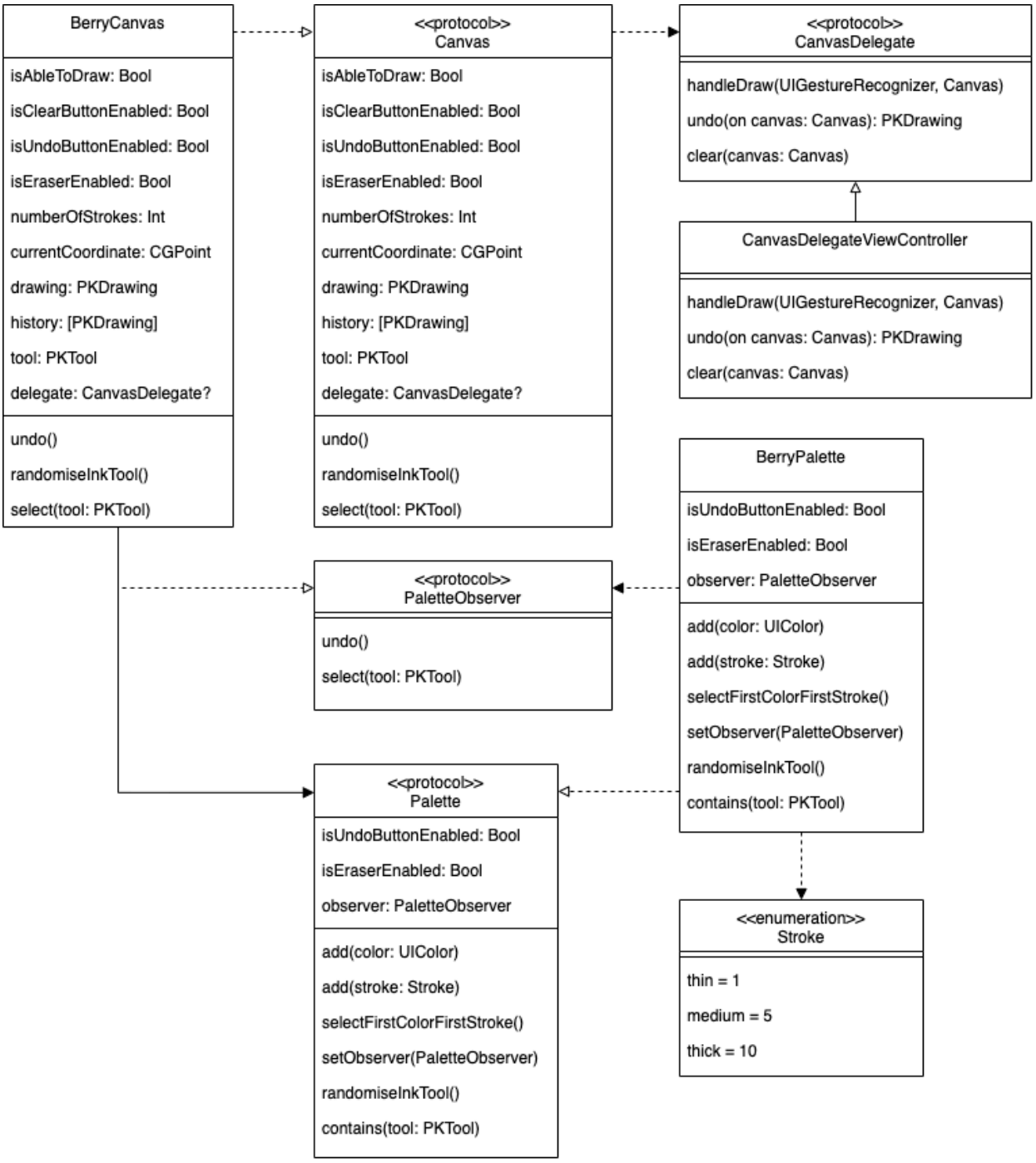
This chosen design is preferable as the powerups that belong to each player are added to their view directly instead of one master view. Additionally, since the views are placed and sized to fit the canvas, we do not need to do any conversions to convert between the player's current drawing coordinates to the coordinates relative to the canvas origin when checking for powerup collisions.

In sprint 2, we continued implementing this option to our voting view controllers. The CompetitiveVotingViewController maintains a mapping between each player and their CompetitiveVotingView, which encapsulates all artists' drawings as DrawingViews.

Module Structure

Canvas

Class Diagram for Canvas



Shown above is the class diagram for the classes related to the Canvas component.

Canvas - Observer Pattern

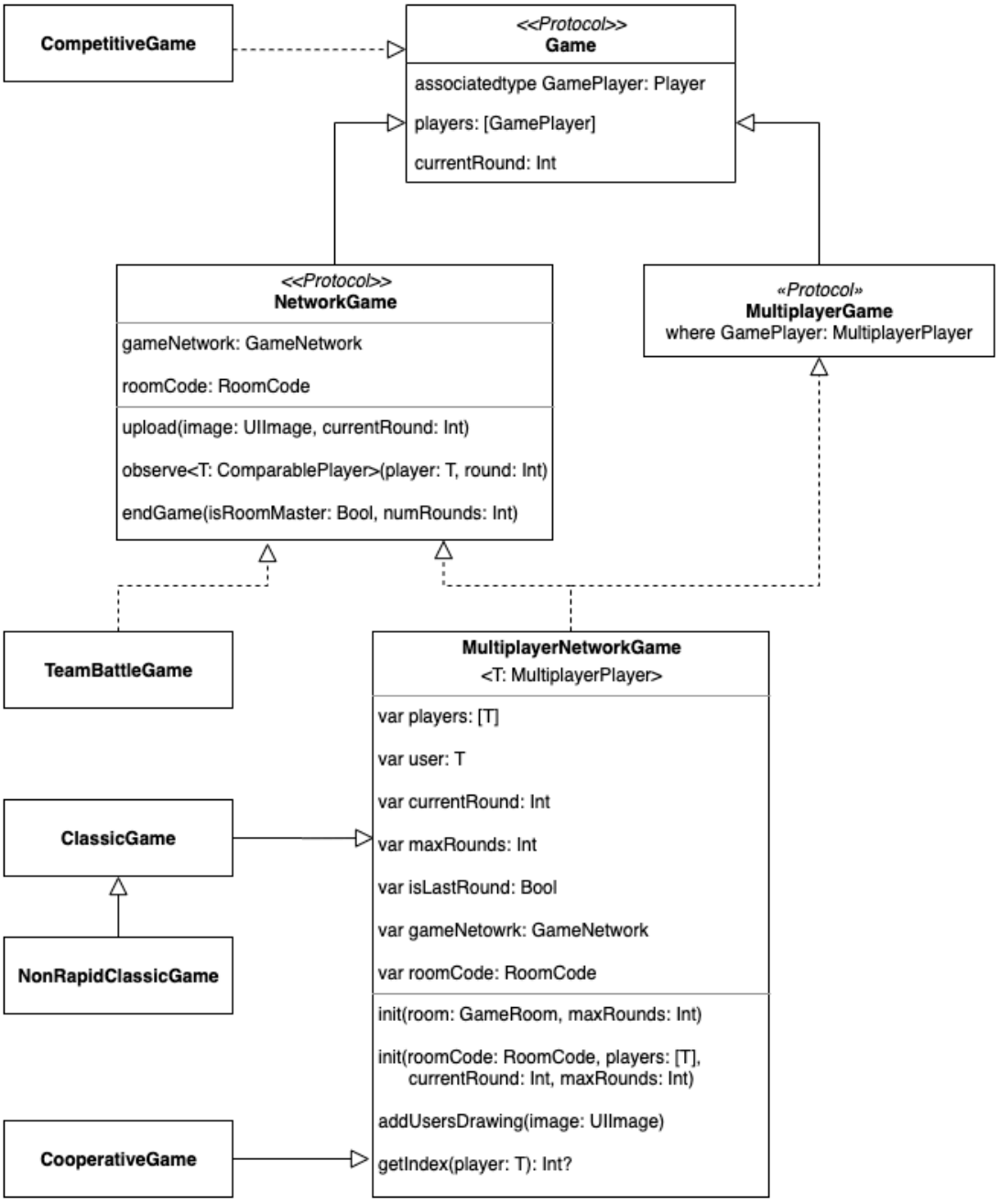
We have employed the **Observer** design pattern in BerryCanvas class (our custom canvas class that conforms to our defined Canvas protocol). The BerryCanvas class observes the BerryPalette and is alerted when a colour or thickness is selected. The BerryPalette simply informs the observer of these changes, and therefore, we have made the BerryCanvas conform to the PaletteObserver class as well, allowing it to be added as an observer of the BerryPalette. This way, we reduce the coupling between the BerryPalette and BerryCanvas, and BerryPalette does not have to know (and should not know) the existence of BerryCanvas that is containing it. This also complies with the **Dependency Inversion Principle**, where both BerryCanvas and BerryPalette depend on protocols Canvas and Palette, and these protocols serve as an abstraction that does not depend on details of implementation.

Canvas - Delegate Pattern

We have employed the **Delegate** design pattern in BerryCanvas class. Since BerryCanvas is a UI element, we do not want it to be involved with any logic handling. Therefore, when detecting a draw, it simply invokes the delegate's handleDraw method without having to handle the state of the drawing. Since we also support undo functions, we also require the delegate to revert to the previous state for the BerryCanvas. By doing this, we decouple the domain logic with the presentation logic, adhering to the **Single Responsibility Principle**.

Game

Class Diagram for Game model



For the module structure of our different game modes, we chose to use some base protocols and protocol extensions to define the kinds of game we can have. The base protocol, **Game**, is defined as a game that contains players and goes through rounds. Since the competitive game mode has all players playing on the same device, CompetitiveGame directly implements this protocol.

We also have the **NetworkGame** protocol for games that require the internet network, and inside this protocol, we also used protocol extensions to provide default implementations of

core methods such as uploading a player's drawing, observing another player's drawing and ending the game.

The **MultiplayerGame** protocol is for games where each of the players are playing on different devices. The **MultiplayerNetworkGame** class conforms to both of these protocols and provides default implementation of functions that multiplayer network games have.

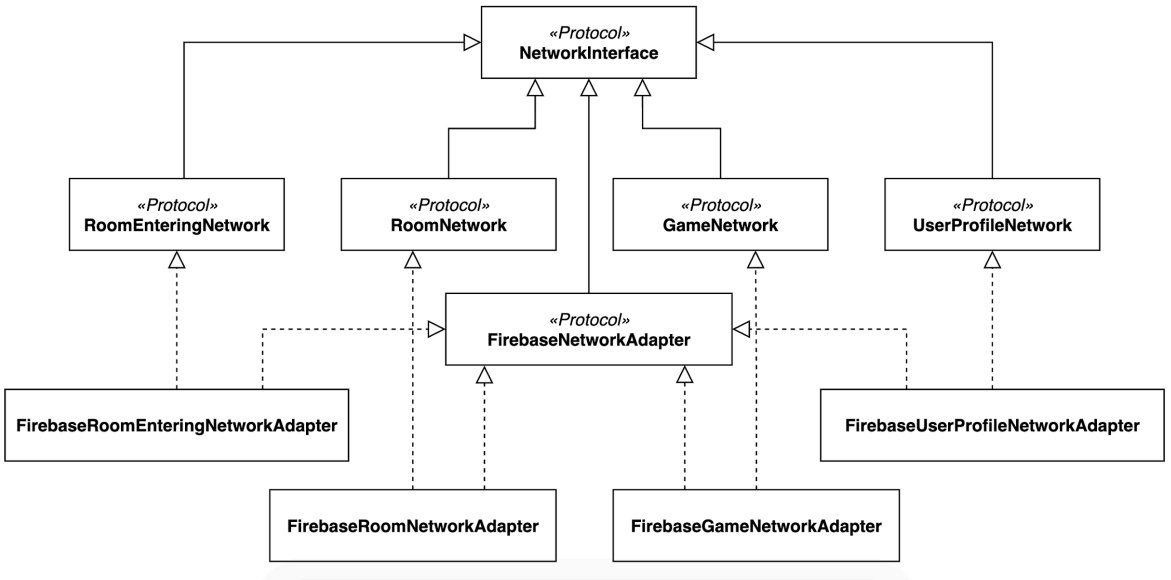
ClassicGame and CooperativeGame inherits from MultiplayerNetworkGame as these 2 game modes consist of players playing against each other over the network. Whereas, for TeamBattleGame, since we are dealing with teams instead of individual players, it directly conforms to NetworkGame.

The rationale for such a modular structure and inheriting the protocols this way is to better allow code reusability and ease of extension, and to also better adhere to SOLID principles, especially, **Single Responsibility Principle**, **Open-Closed Principle** and **Interface Segregation Principle**. For example, the NetworkGame is responsible for dealing with the network and implementing game modes do not have to provide additional code for syncing their game state over the network. Moreover, in the future, we can easily extend our Game protocol to other protocols, such as LocalMultiplayerGame for local bluetooth game modes, etc. Such a structure also allows each of us to easily work on different game modes without worrying about merge conflicts and integration issues.

Each of the GameMode can also easily be subclassed to provide slightly different functionalities. For example, the newly introduced Non-rapid Classic Game, whereby players do not have to play in real time like normal classic games, extends from ClassicGame and overrides some of its functions in order to support this feature. Such **polymorphism** allows the view controllers for the classic game screens to easily update the model based on the player's input.

Network Component

Class diagram for Network



The network component follows the **Dependency Inversion Principle** and heavily utilizes the **Adapter Pattern**, where the classes in our model, such as the GameRoom and the different game modes, rely on a network interface to **reduce coupling**; while the concrete implementations of these protocols are Firebase adapters. The network adapter classes allow the client to work with the Firebase services without knowing the underlying implementation. Additionally, in the future, if we change our backend service provider, we can easily add new classes that conform to the existing protocols without having to heavily modify the code in the classes of our model component.

Furthermore, as we are using multiple services provided by Firebase, including the Realtime Database and Cloud Storage, the protocols act as interfaces with simpler and more readable methods for syncing data between the players on different devices. This abstracts away the database's data model and Firebase's convoluted API from the application's model component.

This leads to better **separation of concerns** as the model classes (GameRoom and Game) do not include code that directly interacts with the backend, while the adapter contains all the code that interacts with the various services of Firebase. This also fulfils the **Single Responsibility Principle** as it separates the data conversion logic from the business logic of the application.

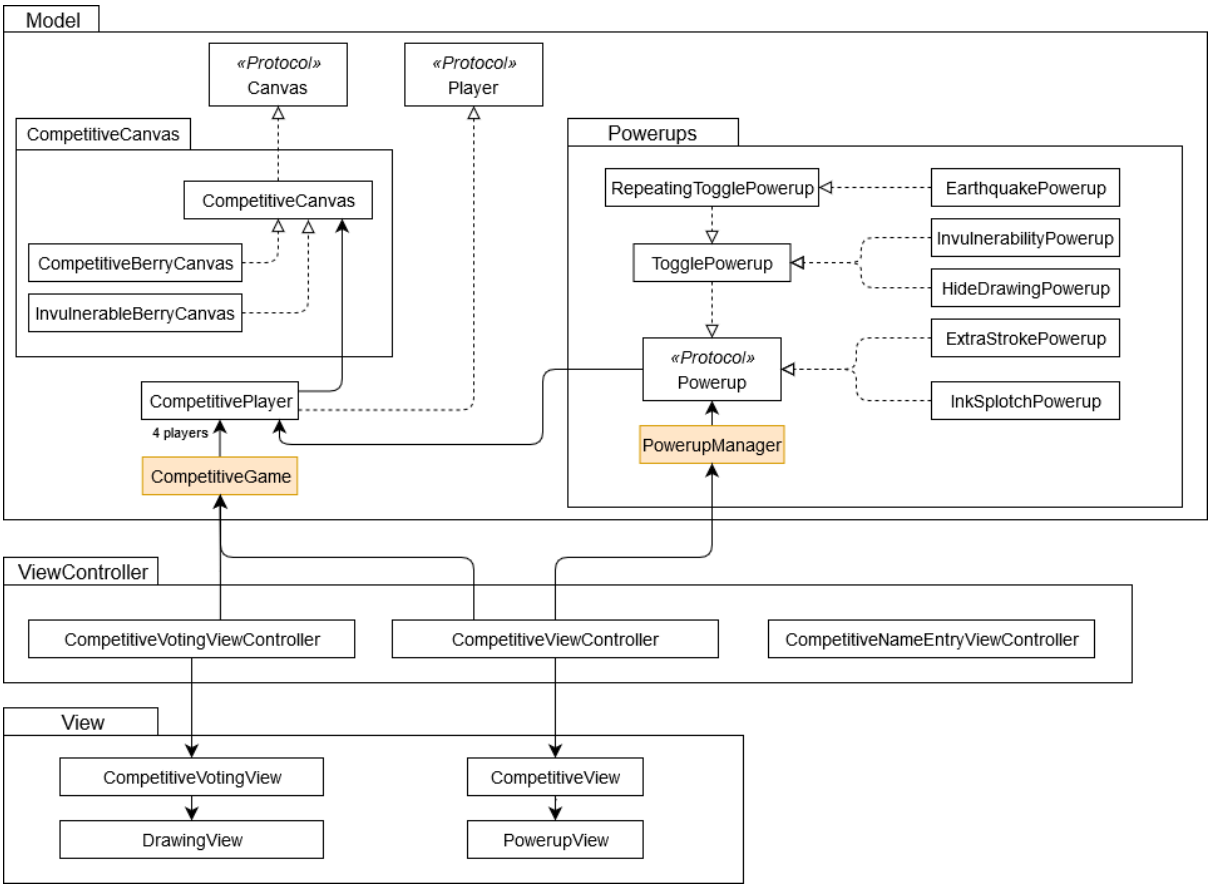
Moreover, the use of the protocols also allows better testability as they can easily be extended and stubbed in test cases. Thus, this decouples the reliance on the third-party library's code and services to work correctly in the test cases.

We chose to use callback closures instead of delegates to update the model once the asynchronous network calls are done as some of the completion callbacks are different even for the same function. Moreover, the call to network functions in the model classes were called from both within the model to sync state from other players and also from the view controller where the user updates state through UI input. Thus, the delegate pattern would not be as appropriate.

Nevertheless, we felt that we could further improve our code with greater protocol segregation for our GameNetwork protocol, as advised by the **Interface Segregation Principle**. For example, some functions in GameNetwork protocol are only used by ClassicGame and another two functions only used by TeamBattleGame. We wanted to segregate out these game-specific functions into separate protocols and subsequently utilize associated types in the NetworkGame protocol and generics in the MultiplayerNetworkGame class to allow the different types of Games to contain different subtypes of GameNetwork protocol. However, due to a limitation with Swift, since we are programming to a protocol, our classes' generics could not contain the protocol types. Hence, we had to go with a slightly more loaded interface for our GameNetwork protocol as we could not find an elegant way to achieve what we want while maintaining our usage of protocols for the network component.

Competitive Mode

Class Diagram for Competitive Mode



Shown above is a class diagram for the competitive game mode.

The *CompetitiveCanvas* protocol extends from the original *Canvas* protocol, describing additional functionality that the competitive game should support, such as the adding of ink splotches and hiding the player's drawings, which is used by the Powerups. Both *CompetitiveBerryCanvas* and *InvulnerableBerryCanvas* implement this protocol and are used in the **decorator** pattern as described below.

The *Powerup* protocol describes the functionality of powerups, such as the powerup's owner, targets and location. Additionally, it defines an *activate()* and *deactivate()* function which executes and unexecutes each powerup's functionality. This is used in the **command** pattern as illustrated below. Furthermore, it defines a required initializer which is used by PowerupManager in the **factory** pattern as depicted below. *TogglePowerup* extends upon the functionality described by the *Powerup* protocol and defines a duration in which powerups deactivate themselves when a certain amount of time has elapsed after they have been activated. Next, *RepeatingTogglePowerup* extends upon the functionality described by *TogglePowerup* and defines the number of times each powerup repeats itself toggling between the powerup's activation and deactivation.

Powerups - Factory Pattern

We have employed the **Factory** design pattern in the Powerup protocol. The Powerup protocol defines a designated initializer which all Powerups must conform to. In this way, the manager class PowerupManager can easily create new Powerup objects using the defined initializer in the Powerup protocol.

Therefore, we do not have to resort to clunky switch statements to initialize each Powerup separately depending on its type. We are able to reduce the coupling between Powerup and PowerupManager because PowerupManager just needs to call .init() for each Powerup to get an instance of that Powerup. This is important as PowerupManager generates random Powerups for each player and does not care what is the underlying type of the returned Powerup.

Furthermore, supporting new Powerups to the game is easy as we just need to conform to the Powerup protocol (and ensure that PowerupManager knows of this new Powerup by adding its type into the defined static variable *ALL_POWERUPS)*.

Powerups - Command Pattern

We have also employed the **Command** design pattern in the Powerup protocol. The Powerup protocol defines a method *activate()* and *deactivate()* which executes and undoes the Powerup effects respectively. The invoker class PowerupManager calls this *activate()* method on each Powerup to execute it, without knowledge of its actual underlying Powerup type.

This allows for **separation of concerns** where the actual code that the Powerup executes when it is activated is decoupled from PowerupManager.

Powerups - Decorator Pattern

With the introduction of the invulnerability powerup for Competitive mode in Sprint 2, changes have to be made to how powerups interact with the canvas. For example, the Powerup that is currently being activated should not affect the target player's canvas if the current player is currently invulnerable.

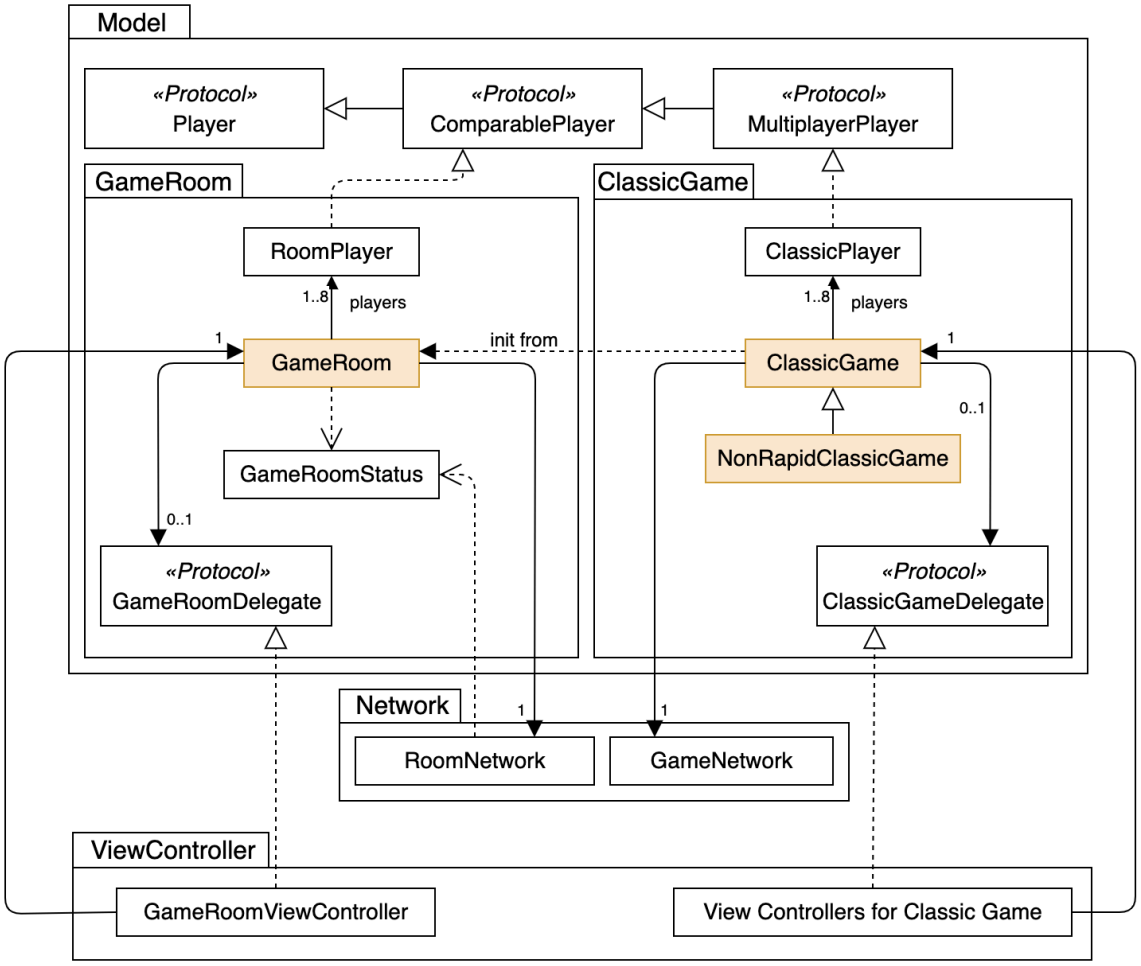
In Sprint 2, we used a wrapper over the canvas to decide if the player was invulnerable which Powerups interact with. If the player was invulnerable, this wrapper returns nil, else it returns the player's canvas. This can be combined with Swift's higher-order functions to get the list of player's canvases to effect. However, in Sprint 3, we have decided to refactor this to the **Decorator** pattern instead. The design outlined above, while meeting the requirements, was not general enough to allow future modifications to the application. Furthermore, we had to resort to using a boolean flag *isInvulnerable*, which may not be the best design if we wish to extend the Powerup system to support more varied powerups.

This new design now defines a protocol *CompetitiveCanvas* which extends from the original canvas. It defines further functionality that the canvas used for competitive mode must support, such as adding ink splotches, hiding and showing the drawings and so on. *CompetitiveBerryCanvas* and *InvulnerableBerryCanvas* then implement this protocol. When the player draws over an invulnerability powerup, an instance of *InvulnerableBerryCanvas* is created and 'decorates' the player's original *CompetitiveBerryCanvas*. This new *InvulnerableBerryCanvas* now interacts with other powerups as it implements the same protocol. However, it makes the user 'invulnerable' by containing an empty implementation for *addInkSplotch* for example, hence the ink splotch call does not go through to the original *CompetitiveBerryCanvas* that the invulnerable canvas wraps over. This continues until the timer expires for the player's invulnerability duration, after which the *InvulnerableBerryCanvas* removes itself from the player's canvas.

This new design is extensible as it is general enough to support future extensions of powerups, fulfilling the **Open Close Principle**. For example, a powerup that reduces harmful effects by half (such as reducing the size of ink splotches by half) would be easily implementable in this design, we can just create another canvas type that implements *CompetitiveCanvas* with the required functionality.

Network Game Interaction

Class diagram for GameRoom and ClassicGame



The class diagram and interaction is similar for other network game modes like the cooperative game, where we employ the **delegate pattern** to update the UI whenever our model updates.

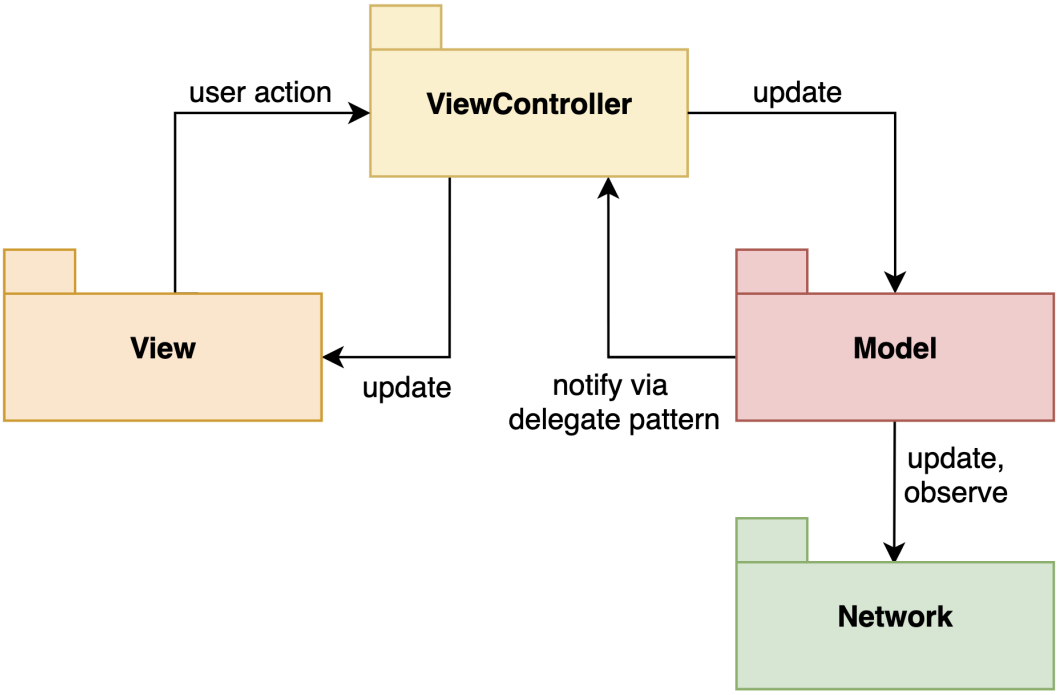
The GameRoom encompasses the room lobby which players enter before the game start. The GameRoomViewController consists of a GameRoom, which is the model, which then consists of a RoomNetwork.

When another player enters the room, the RoomNetwork updates the list of RoomPlayers in GameRoom. The update in GameRoom causes the function playersDidUpdate to be called in the GameRoomDelegate protocol, hence updating the list of players in the UI through the GameRoomViewController.

Similarly each of the different screen's ViewController during the actual classic mode gameplay consists of a ClassicGame object. When the user finishes drawing, or votes for another player etc, the ClassicGame's NetworkGame protocol methods are called which updates the database through the GameNetwork object.

Likewise, at the appropriate screen, the ClassicGame also listens for other player's actions by observing the database through NetworkGame functions that call the methods in GameNetwork. When there is an update, the ClassicGame model will be updated, which then calls the method in ClassicGameDelegate protocol such as drawingsDidUpdate and votesDidUpdate to update the view controllers which then updates the views.

Hence, our usage of the **MVC Architectural Pattern** leads to **high cohesion** as the game classes in model focus on game logic and modelling state of the game, the view controller focus on updating the views and handling user interaction, while the network component deals with interacting with the backend and data conversion logic. Therefore, we also achieve **loose coupling** between the components which increases testability and ease of extension.



Authentication

Authentication - Facade Pattern

We have also employed the **Facade** design pattern in the Authentication class. The Authentication class acts as a **Facade** that provides a simple interface to complex authentication services in Firebase.

Although the facade provides limited functionality as compared to working with Firebase directly, it provides only the relevant functions that we are interested in, which is sufficient. Furthermore, the facade simplifies the use of a complex library which allows for easier understanding and reduces the coupling between our code and third-party libraries.

The rationale for us using facade pattern instead of through an interface and adapter pattern is because the View Controller for the login and signup page directly interacts with the Authentication module's class level methods which is more appropriate than having authentication classes in the model component that do not contain any properties.

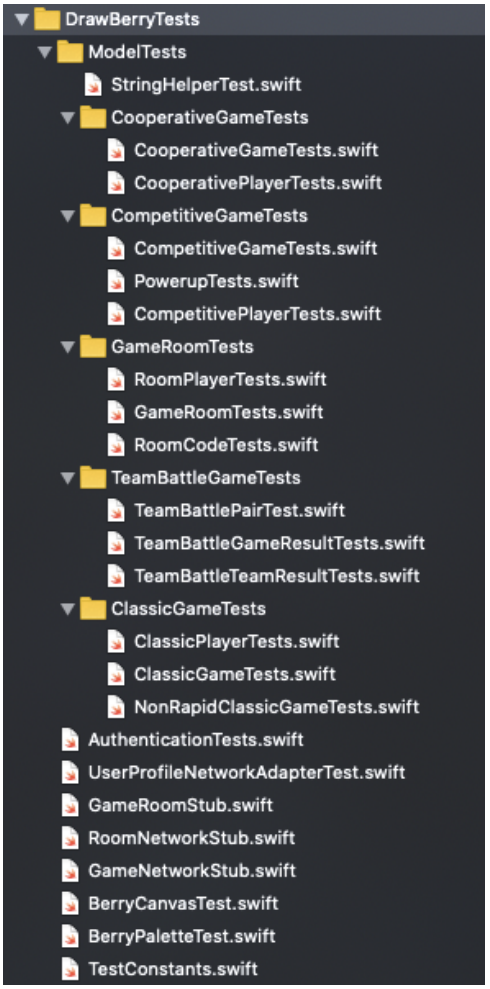
Testing

Test Strategy

The overall test strategy for our application is a bottom-up approach. We first start with unit tests and integration tests, before proceeding to UI tests. We will outline our strategy for stress tests and performance tests. Our testing strategy employs a mix of scripted and exploratory testing which is the most suitable as we developed our app in rapid iterations.

Unit Tests and Integration Tests

For unit tests and integration tests, we are employing Xcode's XCTest framework. We have unit tests for each of our classes in the model component and helper component. The unit tests and integration tests can be found under the DrawBerryTests folder of our Xcode project as seen below.



For model components that rely on third-party libraries and the correct provision of external services, we utilise dependency injection and stubbed these classes. For example, the

GameRoom class in model depends on the RoomNetwork interface in Network component to provide the syncing of players in the room. In our unit tests, we stubbed a RoomNetworkStub class that does not rely on our Firebase implementation.

Our tests are written using a gray-box approach, where we design our test cases based on some important information about the class and methods. We also adhere to tried and tested test case design heuristics such as using equivalence partitions and boundary value analysis. For example, this can be seen extensively in our test case that handles user inputs such as emails and room code.

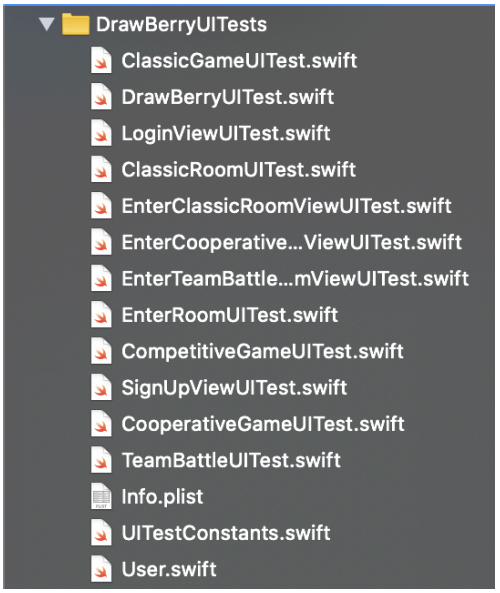
We believe using a gray-box approach is more appropriate as compared to using the glass-box approach, it will be less prone to developer oversight or bias. Whereas, it is also more comprehensive and time-saving compared to the glass-box approach as we have some idea how a function handles different partitions of inputs. Thus, we can have more meaningful test cases that are more effective and efficient. We also track the statement coverage of our test cases through Xcode.

UI Tests

We have employed the use of FBSnapshotTestCase to compare the actual screenshot of the app and the expected screenshot of the app. This method compares the screenshots pixel by pixel and will therefore be effective in ensuring that the layout of the app in the various views is how we expect it to be. We have made use of XCUIApplication to programmatically (and automatically) navigate through the app to obtain the required screenshots for comparisons.

In each view controller, we have a UILayoutTest that asserts that all UI elements are in place. For view controllers that have unique behaviours (for instance, the Classic Mode view controller has a drawable canvas with selectable colors and thickness of brushes), we programmatically interact with the view controller and assert the resulting view from the interactions. In the example of the canvas, a sample test would be to draw a stroke, press the undo button, and then compare the current screen with an empty canvas screen.

The UI tests can be found under the DrawBerryUITests folder of our Xcode project as seen below.



Stress Test and Performance Test

Classic/Cooperative mode

Since the classic and cooperative mode is online, when the number of concurrent users exceeds a large number (~1000), it is possible that there might be some decrease in performance for our app. Some metrics for stress testing include:

1. Network Response
    - Average time taken to join/host a room
    - Average time taken to retrieve drawing images after game ends

2. Failures
    - Number of failed connections by the client

This can be done by simulating numerous users and logging the requests on our backend side.

Competitive Mode

By tweaking the probability in which powerups spawn to 1 (instead of the current value of 0.005), we can perform an ad-hoc performance and stress test on the system. On an iPad 10.5”, the current design is able to handle around 3000 powerups before showing some sluggish performance when handling drawing strokes.

Other Tests

Aside from the aforementioned tests, other test strategies that may be harder for us to implement includes end-to-end testing to test the flow of the whole application. Other than scripted testing, we may want to employ some more manual testing such as user acceptance testing.

Reflection

In this final sprint, we focused on refactoring our code and also managed to complete all the core features of our application.

Evaluation

In Sprint 1 and Sprint 2, we focused on churning up the features in the short deadline to have a MVP. In this sprint, we shifted our focus to refactoring and future-proofing our code. This also included the cleaning up of our codebase to remove duplicated code we added when added new game modes in previous rounds. We also improved the implementation of a few of our components, such as refactoring the Network component to better utilise protocols and allow for ease of extension/usage of other network services, refactoring the Game model, and refactoring the View Controllers for similar screens in different game modes. In Competitive mode, we refactored the Powerup system to make it more general in order to allow further extensions to the Powerup system, as well as refactoring some voting functionality to the Player instead of the View Controllers. We also extended the Network Component to support a new Cooperative Game Mode that reuses the network code that was initially only supporting the Classic Game Mode.

In Sprint 3, the Network component was further refactored to become more general so that it becomes easily extendable to support new game modes in the future. One example is the new Team Battle Mode that we had introduced in Sprint 3, again making use of the extensibility of the Network Component to support it. In addition, we now define the Network Component as a protocol instead of a concrete class in accordance to the **Dependency Inversion Principle**, and created an instance of FirebaseNetwork classes that conform to the Network Protocol. This is done so that in the future if we decide to use other backend services like AWS, we will just need to conform to the same Network Protocol for the new service to be compatible with our application.

We were able to extend the application with more features mainly due to the adequate design choices we had made in the first sprint. By adequately specifying the base classes of the types of game, we could easily use **inheritance** to implement new game modes. For example, by having a protocol NetworkGame for games that require the networking and with the use of the **adapter** pattern for the implementing classes of our network interface, the new game modes do not have to worry about dealing with connecting to our database and syncing game state between players. This is in line with the **Open Close Principle** where our design of the game model is open for extensions (the new team battle game mode) and closed for modifications (that we do not need to rewrite code within the adapter since the existing functions are generic enough to allow reusability for future game modes).

For competitive mode, in Sprint 3, we received feedback from Prof. Wai Kay after Sprint 2 and decided to change the Powerup design from a simple wrapper to the **decorator** pattern.

This allows us to make the design more general so that we can accommodate future modifications and extensions to the Powerup system. This is explained further in the Competitive Mode section under Module Structure.

Lessons

Throughout the three weeks of sprint 3, we learnt a lot, including:

- How to take requirements that were given/assigned to us and implement them
- How to apply design patterns to allow easier understanding and maintenance of our code
- How to model each game round properly in the database
- How to fix random Firebase bug that gives null keys
- How to design protocols and objects such that they are easily extensible and modifiable
- How to refactor protocols and objects to make them more reusable and also to reduce code duplication
- Things seem easier to implement until you start implementing and getting random bugs

We will definitely be applying this newfound knowledge to future Software Engineering projects and modules as we share with each other what we learnt.

Known Bugs and Limitations

Known limitations that we plan to address in the upcoming sprints include:

- Unfortunately, close to the end of sprint 3, we realised certain ISPs, such as the ISP NUS network is on, have blocked FirebaseStorage's links and API calls, thus, if you are on these ISPs, you will not be able to play the multiplayer network game modes. As it was too close to the deadline of our project, we did not have enough time to explore other backend service providers.
- Multiple logins on the same account should not be allowed
- Players do not leave active game rooms if they quit the app suddenly
- In competitive mode, it may be hard for players to judge the amount of invulnerability time they have left.

Appendix

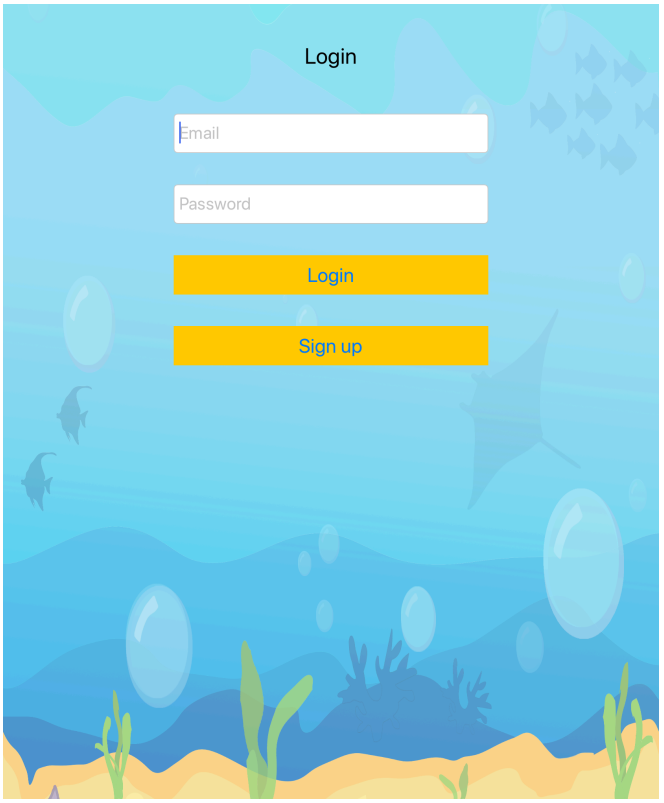
Test Cases

Unit Tests and Integration Tests - Under “DrawBerryTests” of our project on Github

UI Tests - Under “DrawBerryUITests” of our project on Github

The following is the test plan for UI Testing. Do note that we have written test cases (programmatically) to test the layout of the UI elements in the view controllers.

Login Page UI Testing



These are the following UI elements that we check when we first enter the page:

1) The background is loaded properly as seen in the screenshot above
2) Email text field
3) Password text field
4) “Login” button
5) “Sign up” button
6) Error message (Not shown initially)

Test Buttons:

1) Login
   Error message shown when:
   - Empty/ whitespaces filled text fields

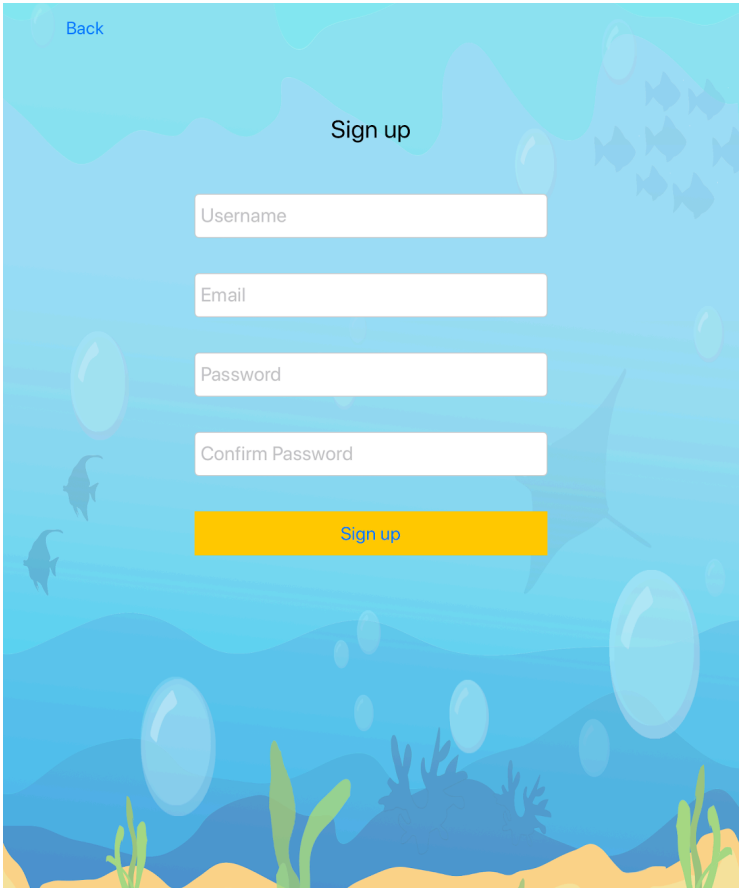
- Email address not containing “@” or “.” separated by at least 1 alphabet each.
- Password not alphanumeric (contains special characters)
- Password less than 8 characters

If no error:

- Goes to the main menu page

2) Sign up
    - Goes to the sign up page

Sign Up Page UI Testing



These are the following UI elements that we check when we first enter the page:

1) The background is loaded properly as seen in the screenshot above
2) Username text field
3) Email text field
4) Password text field
5) Confirm password text field
6) “Sign up” button
7) “Back” button at top left
8) Error message (Not shown initially)

Test Buttons:

1) Sign up

Error message shown when:

- Empty/ whitespaces filled text fields
- Email address not containing “@” or “.” separated by at least 1 alphabet each.
- Password not alphanumeric (contains special characters)
- Password less than 8 characters
- Password confirmation does not match password

If no error:

- Goes to the main menu page

2) Back
    - Goes to the login page

Classic Mode UI Testing

In the classic mode, these are the following UI elements that we check when we first enter the gameplay:

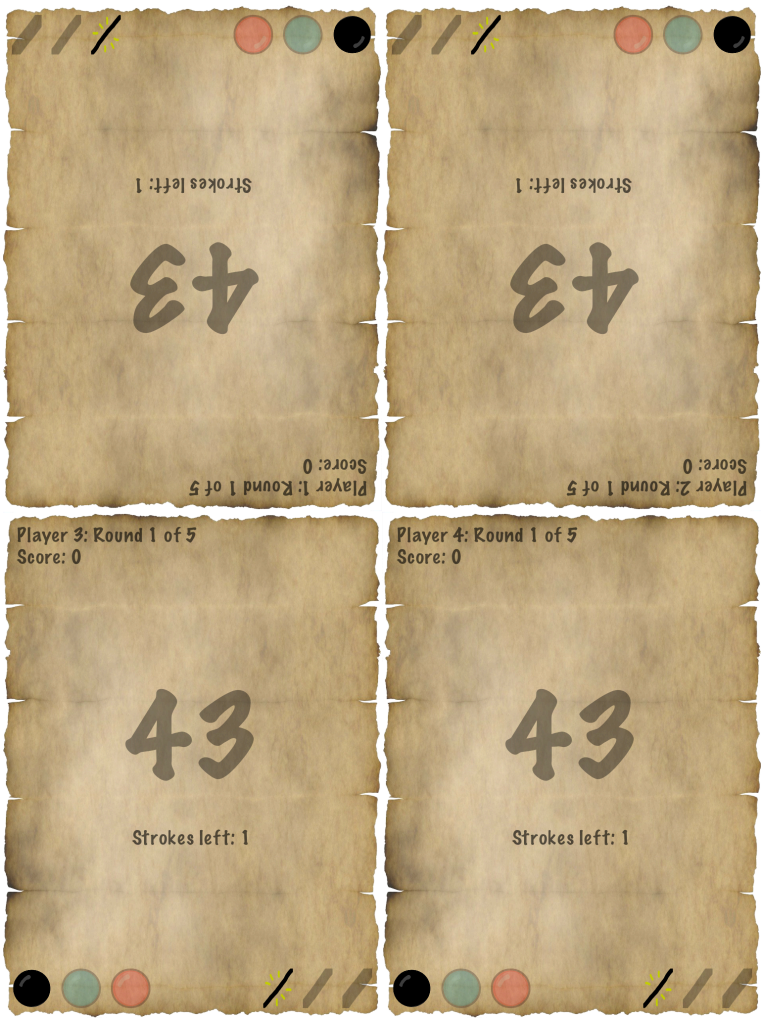
3) The background is loaded properly as seen in the screenshot above
4) Three colours (black, green and red) are visible at the bottom left of the screen.
5) Three strokes of varying thickness visible at the bottom right of the screen.
6) An eraser icon to the right of the three strokes.

7) An undo button to the right of the eraser icon
8) A clear button at the top right of the screen
9) A done button in the bottom third of the screen, centralised as seen in the screenshot

Additionally, the state of the screen is as seen in the screenshot above:

1) The black ink is selected and has full opacity
2) The blue and red ink is not selected and both of them have half opacity
3) The thinnest (leftmost) stroke is selected and has full opacity with yellow stripes
4) The medium and thick strokes to the right are not selected and both of them have half opacity, with no yellow stripes
5) The eraser button, undo button and clear button all have full opacity

Competitive Mode UI Testing



In the competitive mode, these are the following UI elements that we check when we first enter the gameplay:

1) The background is loaded properly in 4 canvases as seen in the screenshot above.
2) The two canvases at the top are rotated 180 degrees, with their UI elements in place with respect to the degree of rotation.
3) Three colours (black, green and red) are visible at the bottom left of each canvas.
4) Three strokes of varying thickness visible at the bottom right of each canvas.

5) There is no eraser icon to the right of the three strokes in each canvas.
6) There is no undo button to the right of the eraser icon in each canvas.
7) There is no clear button at the top right corner of each canvas.
8) A countdown timer is seen in the middle of each canvas, centralised and faded.
9) Players can see the current round, score and their name on the top left-hand corner of their screen.
10) Players can also see the number of strokes they have left below the countdown timer.

Additionally, the state of the screen is as seen in the screenshot above:

1) The black ink is selected and has full opacity for each canvas.
2) The blue and red ink is not selected and both of them have half opacity for each canvas.
3) The thinnest (leftmost) stroke is selected and has full opacity with yellow stripes for each canvas.
4) The medium and thick strokes to the right are not selected and both of them have half opacity, with no yellow stripes for each canvas.
5) The eraser button, undo button and clear button all have full opacity for each canvas.

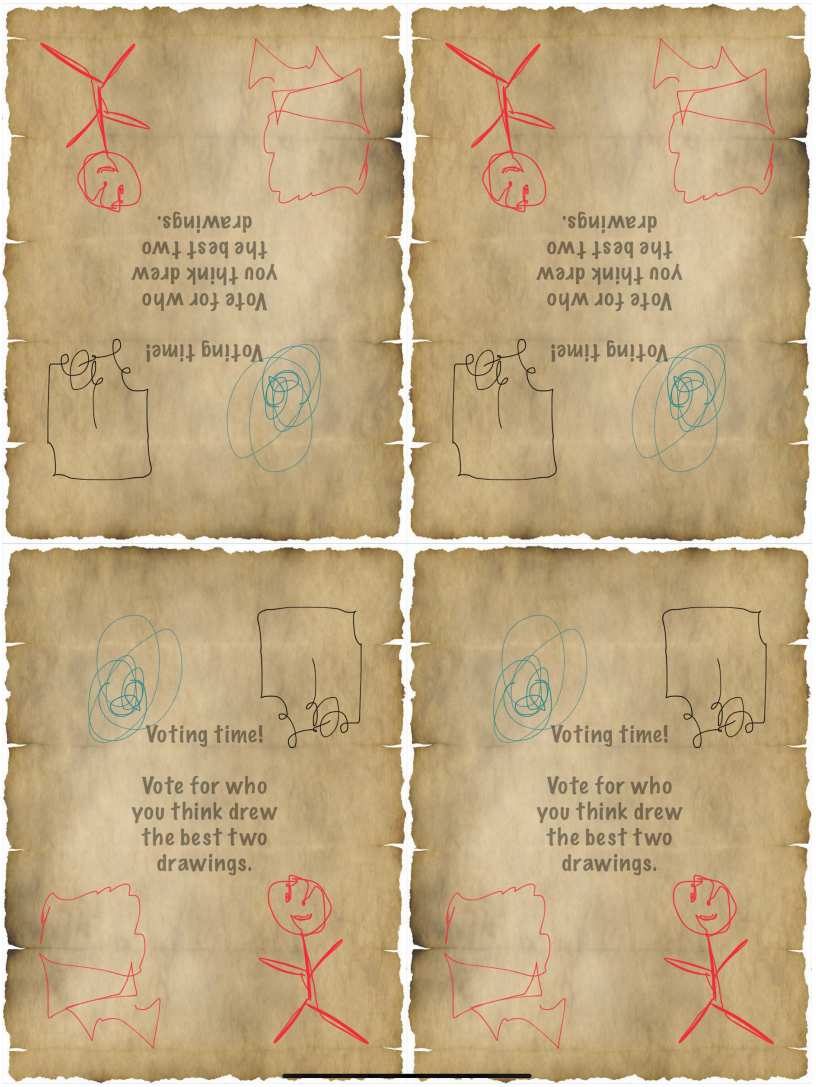
We also test for the following gameplay mechanics:

6) The timer decrements by 1 every second.
7) Once a player has used all of their strokes, they are not allowed to draw any more on the canvas. This includes cases such as users drawing off their canvas bounds which ends their stroke prematurely.
8) Powerups will spawn on each player's canvas at random locations randomly while they are drawing.
9) When a player draws on a powerup, the powerup activates and disappears from that view. Furthermore, a message explaining the activated powerup will appear.
    a) When a player draws over an Extra Stroke powerup (green powerup), they get an extra stroke. The Extra Stroke message appears on the owner's view. We can test that the extra stroke is indeed given to the player by releasing our finger and drawing a new stroke.
    b) When a player draws over a Hide Drawing powerup (purple powerup), all other players' canvases disappear for 1 second, then reverts back to normal. The Hidden Drawing message appears on all targets' views.
    c) When a player draws over an Ink Splotch powerup (yellow powerup), all other player's screens will render an ink splotch of random size and random location. This ink splotch appears over the player's drawings and the timer. The Ink Splotch message appears on all targets' views.
    d) When a player draws over an Earthquake powerup (brown powerup), all other players' screen shake. The Earthquake message appears on all targets' views.
    e) When a player draws over an Invulnerability powerup (pink powerup), the player is given invulnerability to all opposing powerups for 5 seconds. The Invulnerability message appears on the owner's view.

i) While the player is invulnerable, we test the Ink Splotch, Hide Drawing and Earthquake powerups on that player. These powerups should not activate on the invulnerable player. The status message still appears on the invulnerable player's view, but there is a message appended informing that the player that he/she is invulnerable and hence there is no effect.

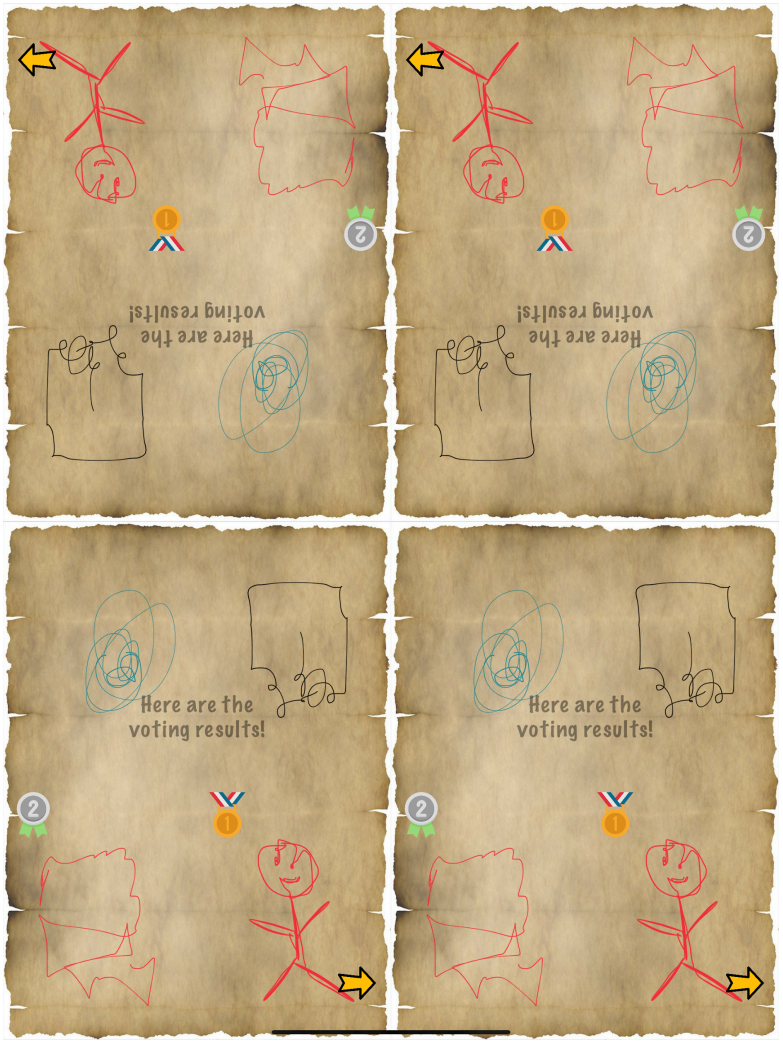
ii) While the player's canvas is under effect by a Hide Drawing / Earthquake powerup, we test activating the Invulnerability powerup. The powerup that was activated before Invulnerability should still run, and should be able to deactivate itself normally. However, all powerups activated within 5 seconds after the Invulnerability powerup was activated should have no effect on the invulnerable player's canvas.

10) When all players have finished drawing (i.e. all players have used all strokes), the timer decreases to 3 or the current countdown time, whichever is lower.
11) When the timer hits 0, all players are unable to continue drawing, even if they have strokes remaining or are in the middle of their stroke. Next buttons appear on each player's view, and when they are tapped, the next button disappears (which means that they are 'ready').
12) When all players touch the next button, the voting screen is shown.

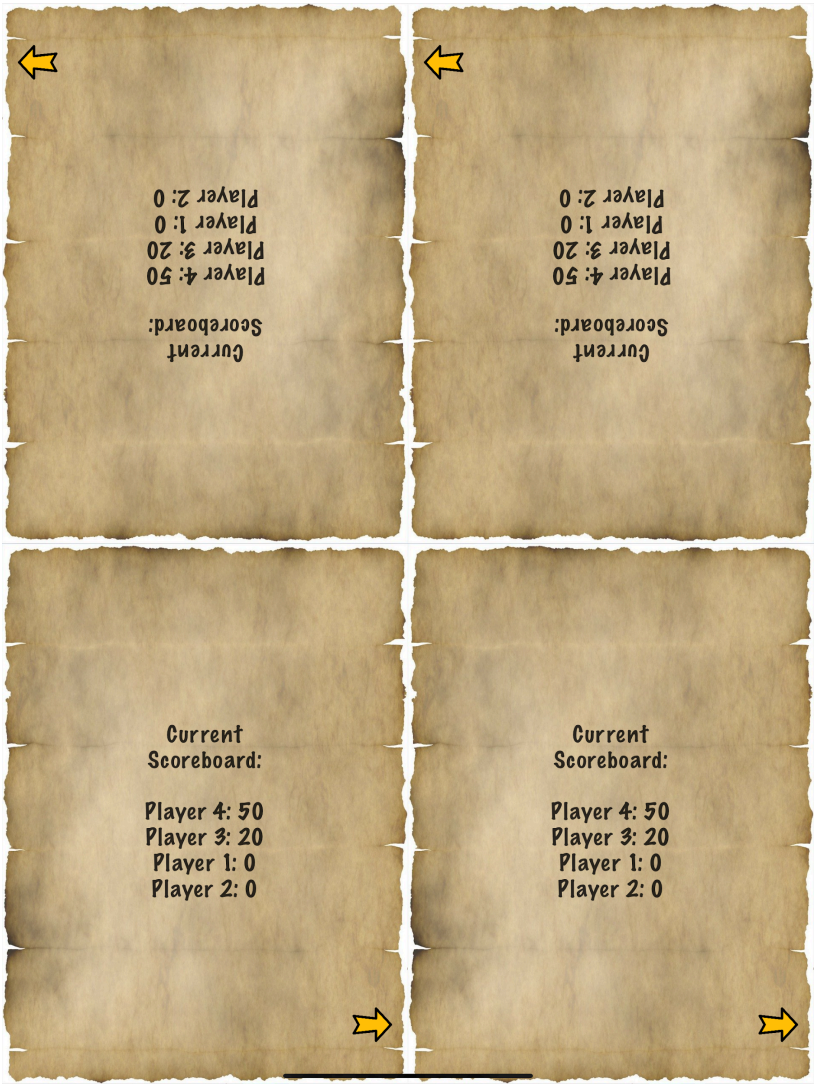


We test the voting screen as follows:

13) Players are able to vote for two players' drawings. The first vote represents their vote for the best drawing (worth 2 votes) while the second vote represents their vote for the second-best drawing (worth 1 vote). When the user votes, the text should update to show that they have voted for that player. We test corner cases such as:
    a) The player votes for themselves (the text should inform the player that they cannot vote for themselves)
    b) The player has already cast their vote for that artist and wants to vote for the same artist's drawing again (the text should inform the player that they cannot cast multiple votes for the same player)
    c) The player has already used all their votes (the text should inform the player that they have already used all their votes)
14) Once all players have voted, the results will be tallied. 50 points are given to the player(s) with the most votes and 20 points are given to the player(s) with the second-most votes. Again, next buttons will be shown on each player's view and tapping them removes the next buttons (they are 'ready').



15) Once all players have tapped the next button on the voting screen, we transition to the results screen.



We test the results screen as follows:

16) We test that the score is updated correctly (50 points for the player(s) with the highest number of votes and 20 points for the player(s) with the second-highest number of votes). Next buttons appear here as well for players to indicate that they are ‘ready’ to move on to the next round/end the game.
    a) If this was their final round, the “Current Scoreboard:” message would be changed to “Final Results:” instead, indicating that this is the final scoreboard.
17) If this round was not the last round (there are 5 rounds in total in 1 competitive game), the game moves to the next round after all players are ready. Else, the game moves back to the main menu screen.

Cooperative Mode UI Testing

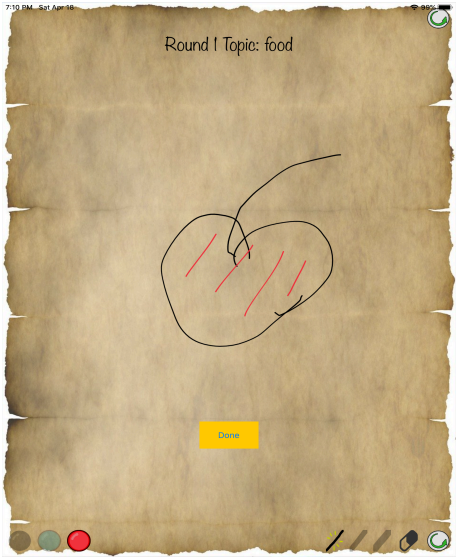
In the cooperative mode, these are the following UI elements that we check when we first enter the gameplay:

1) The background is loaded properly as seen in the screenshot above
2) Three colours (black, green and red) are visible at the bottom left of the screen.
3) Three strokes of varying thickness visible at the bottom right of the screen.
4) An eraser icon to the right of the three strokes.
5) An undo button to the right of the eraser icon
6) A clear button at the top right of the screen
7) A done button in the bottom third of the screen, centralised as seen in the screenshot
8) A shaded portion to denote the out-of-bounds area in the canvas.

Additionally, the state of the screen is as seen in the screenshot above:

6) The black ink is selected and has full opacity
7) The blue and red ink is not selected and both of them have half opacity
8) The thinnest (leftmost) stroke is selected and has full opacity with yellow stripes
9) The medium and thick strokes to the right are not selected and both of them have half opacity, with no yellow stripes
10) The eraser button, undo button and clear button all have full opacity

Team Battle Mode UI Testing



As a drawer in the team battle mode, these are the following UI elements that we check when we first enter the gameplay:

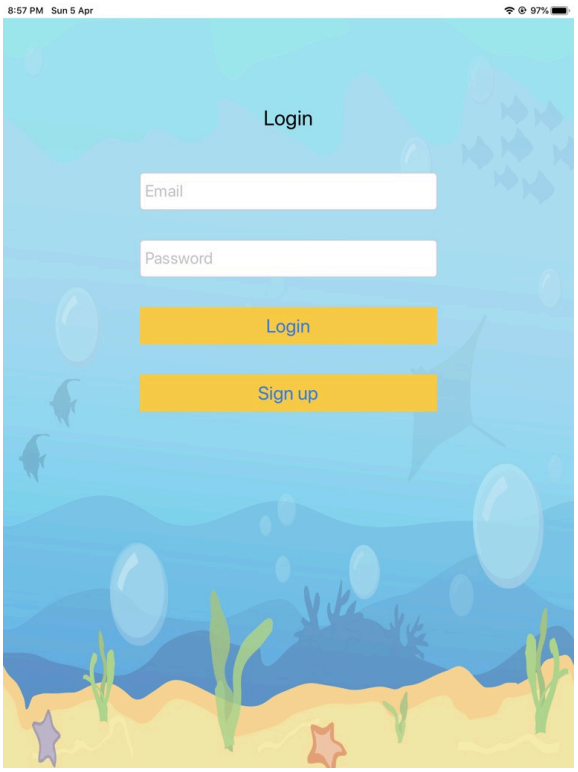
1) The background is loaded properly as seen in the screenshot above
2) The topic word is at the top of the screen, centralised as seen in the screenshot
3) Three colours (black, green and red) are visible at the bottom left of the screen.
4) Three strokes of varying thickness visible at the bottom right of the screen.
5) An eraser icon to the right of the three strokes.
6) An undo button to the right of the eraser icon
7) A clear button at the top right of the screen
8) A done button in the bottom third of the screen, centralised as seen in the screenshot

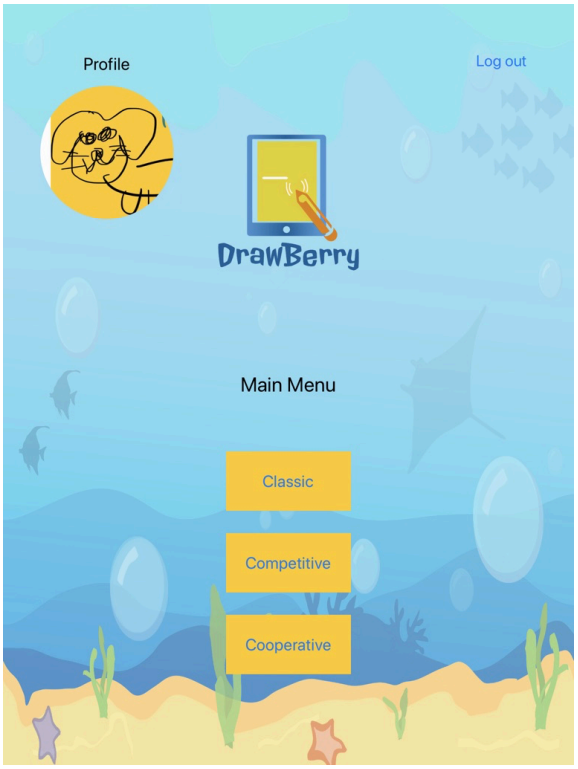
As a drawer in the team battle mode, these are the following UI elements that we check when we first enter the gameplay:

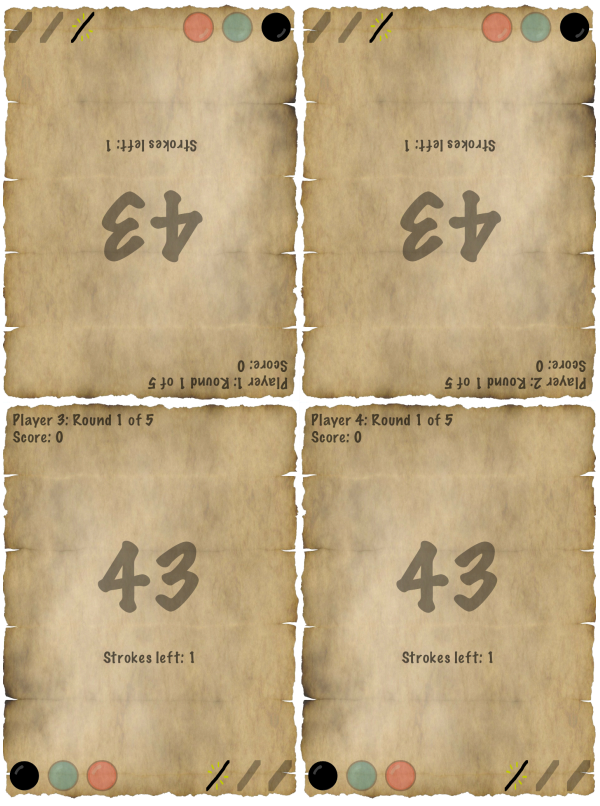
1) The background is loaded properly as seen in the screenshot above
2) An input text field at the bottom of the screen
3) A guess button at the bottom left of the screen
4) A next button at the bottom right of the screen
5) A done button in the bottom third of the screen, centralised as seen in the screenshot
6) If the player inputs a wrong guess, an error message is shown on the bottom right in red as seen in the screenshot.

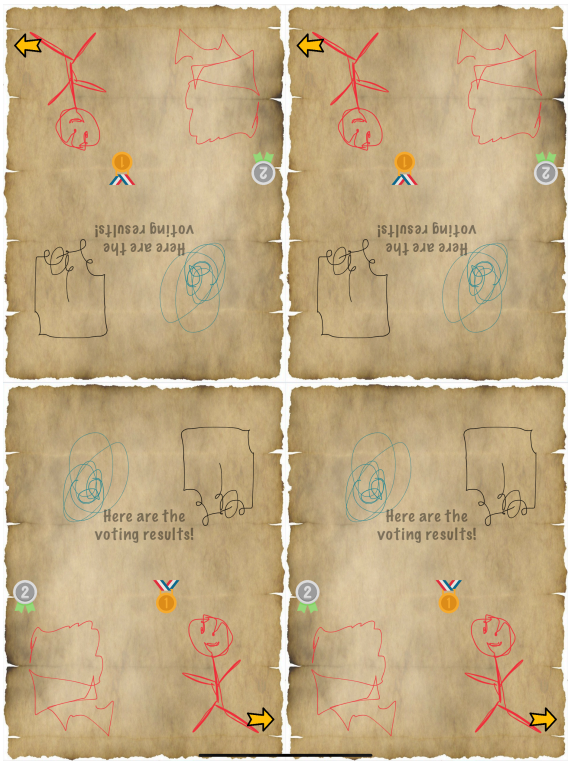
GUI Screenshots

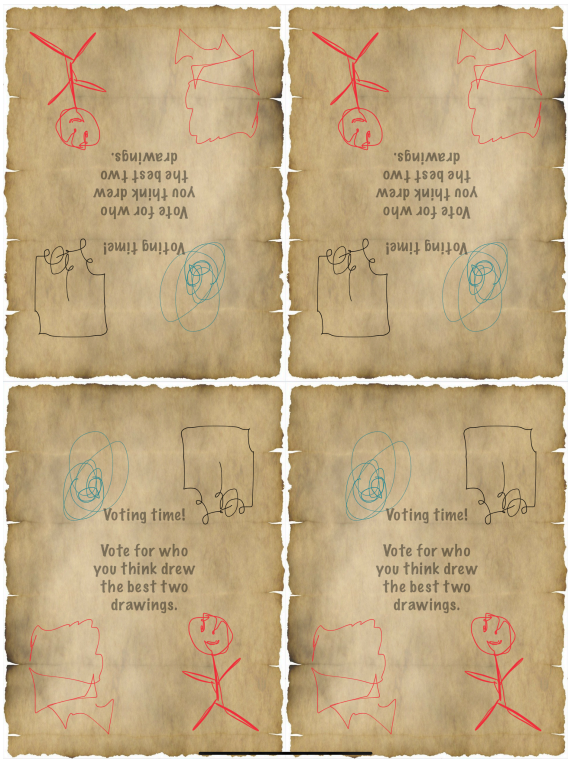
You can refer to the **User Manual** affixed below for more screenshots.

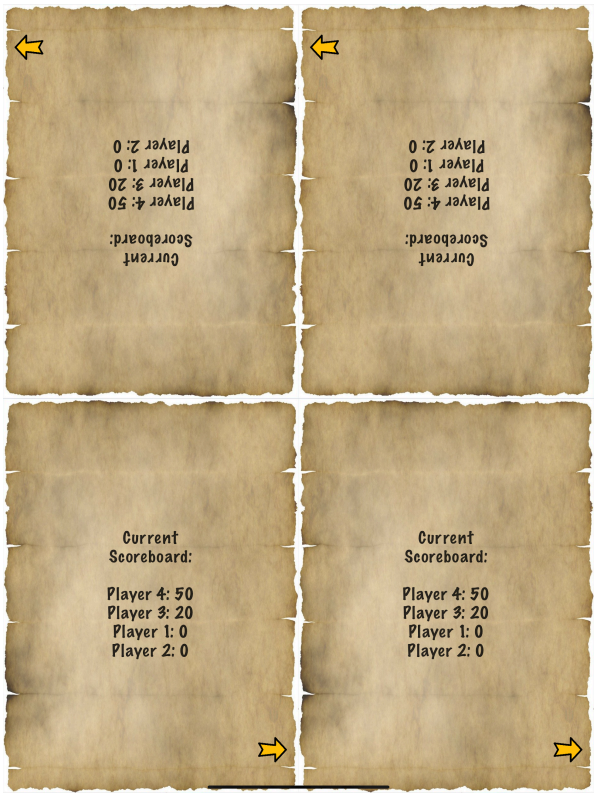







Here are the voting results!
Here are the voting results!


Voting time!
Vote for who you think drew the best two drawings.
Voting time!
Vote for who you think drew the best two drawings.


Current Scoreboard:
Player 4: 50
Player 3: 20
Player 1: 0
Player 2: 0
Current Scoreboard:
Player 4: 50
Player 3: 20
Player 1: 0
Player 2: 0