CS3217: Ninja Warriors

Final Project Report

Outline

Final Project Report

1: Requirements

1.1: Overview

Ninja Warriors is an online multiplayer top-down game belonging to the multiplayer online battle arena (MOBA) genre. It features a ninja theme, with all characters portrayed as ninjas in various outfits. The game offers a single-player mode for players to practise their movements and aiming skills. In multiplayer mode, players can choose to play as a guest or log in to their account. Each multiplayer game session can accommodate two to four players. The game is dynamic, with each player spawning in a specific square arena at the beginning of the game. As it is a top-down game, there is no gravity affecting the players. Three game modes are available: obstacle mode, gem collection mode, and closing zone mode.

1.2: Revised Specification

1. Single Player

Lobbies

- A holding area where only one player will enter the game.
- Once a player presses ready, the player will immediately see a start button.
- Pressing the start button will bring the player into the game.

Single Player Combat

- Players can damage 3 bots that are stationary

2. Multiplayer

Lobbies

- A holding area where at least 2 players and at most 4 players can gather to play a multiplayer mode.
- Players will be able to see how many people in the ready queue
- Once a player queues up, the player will not be able to leave the queue
- Once the desired number of players are in queue, players will see a start button. Pressing the start button will bring the player into the game.

Multiplayer Combat

- Players can view the same game state as other players in the lobby, such as their positions, health, and skills.

3. Gameplay

Shape

- The player is resizable at any point during the game
- The player can change shape at any point during the game
- The player can rotate and change orientation at any point during the game

Health

- The player starts with 100 health units, with a UI display over their Ninja.

Player Status

- The player that is controlled by the current user is labelled as 'You' while opponents are labelled as 'Enemy'.

Movement

- The player can be moved across the map using a draggable joystick.
- Movement options include horizontal, vertical, and diagonal directions.
- The player's speed is determined by the distance between the centre of the inner stick and the centre of the outer stick.
- Releasing the joystick causes the player to cease movement in the current direction.

Skill Activation

- Players can activate skills by tapping the corresponding buttons in the gameplay area.
- The name of the skill will be at the bottom of the corresponding buttons.

Dash Skill:

- Activates only when moving: Dashes 100 units forward with a 1-second animation.
- Cooldown: Varies between characters (3 to 100 seconds)
- Collision: Stops both dashing players, triggering a stop animation.

Slash Skill:

- Display state: Initiates a slashing animation.
- Effect: Damages players within 10 units by 10 health points.
- Health effect: Shows damage on opponent's health bar.
- Cooldown: Varies between characters (3 to 100 seconds)
- Interaction: Both slashers receive damage if they attack simultaneously.

Hadouken Skill:

- Display state: Initiates a blast animation.
- Effect: Damages players within a maximum distance of 100 CGDouble by 20 health points.
- Health effect: Shows damage on opponent's health bar.
- Cooldown: Varies between characters (5 to 8 seconds)

- Interaction: Both players who activated Hadouken receive damage if they attack simultaneously.

Dodge Skill:

- Display state: Displays a bubble shield effect around the player.
- Effect: Grants 2 seconds of invulnerability, allowing movement.
- Cooldown: Varies between characters (3 to 100 seconds)

Refresh Skill:

- When the game starts: Begins with a 30-second cooldown.
- Effect: Resets cooldowns for all skills except itself.
- Cooldown: Varies between characters (8 to 30 seconds)

4. Gamemode

Closing Zone:

- The safe zone is a circle that gets progressively smaller as time passes during the game.
- Eventually, the zone will stop closing when the safe diameter is 50 CGDouble
- Players outside the zone will have their health reduced by 5 every 0.5 seconds.
- Players inside the zone will not take damage from the zone.

Gems:

- Only players are able to collect the gems.
- Players collecting gems will cause them to disappear.
- Collecting gems will not slow down the player.
- Skills will pass through gems.

Obstacles:

- Player colliding with obstacles will stop the player.
- Skills colliding with obstacles will cause the skill to disappear.
- Players hiding behind obstacles with a gap of 10 CGDouble distance will not get damaged by the obstacle

5. Achievements

Achievements are collectable trophies that players receive as a reward for successfully reaching a particular goal in NinjaWarriors. The game currently has achievements that are played in a single game round.
Achievements can also be marked as repeatable, or non-repeatable - meaning that they can be repeatedly earned for every new game round, or can only be earned once.

The game’s achievements are:

- Killed ten people

- Played ten games
- First damage in game
- Three dashes in game
- High damage but no kill

More details on how achievement was built can be seen in the achievements observer pattern section.

1.3: User Manual

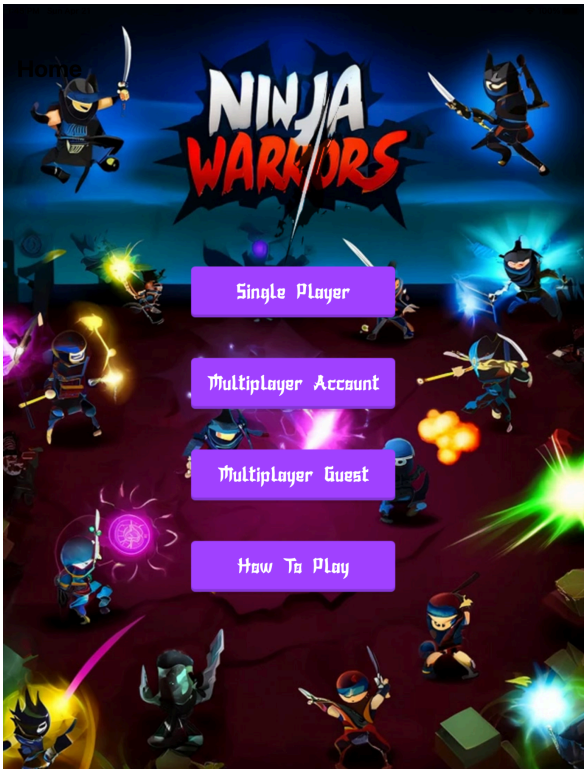
User Manual

Welcome to NinjaWarriors, a competitive iOS game supporting 1-4 players, featuring various game modes, maps, skills, and characters!

NinjaWarriors offers both single-player and multiplayer gameplay, requiring an active internet connection for both modes, and is only available on iPad with iOS 16 and above.

Authentication Page

When you start the game, you'll arrive at the authentication page, which provides four options: a single-player mode, a multiplayer account mode requiring login or signup, a multiplayer guest mode allowing play without logging in, and a how-to-play page with basic instructions.

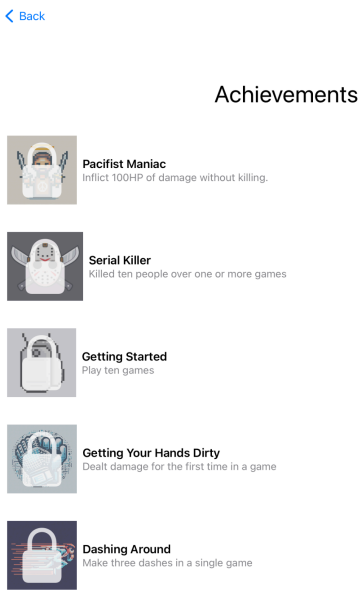


How to Play Page

Let's begin by grasping the basics of gameplay through the how-to-play page. The joystick, as seen in the top left but will be situated at the bottom left of the gameplay screen, governs player movement. It's crucial to shift the joystick from its initial position; otherwise, the player won't budge. One of the game modes, the closing zone, is depicted at the top right through animation. To prevent unnecessary health loss, players must remain within the zone. In another mode, gem collection, players strive to amass as many gems as possible. Some of these gems are visible at the bottom left of the screen. Lastly, to inflict damage on other players or activate skills, buttons are located at the bottom right. The page should resemble the figure on the next page.

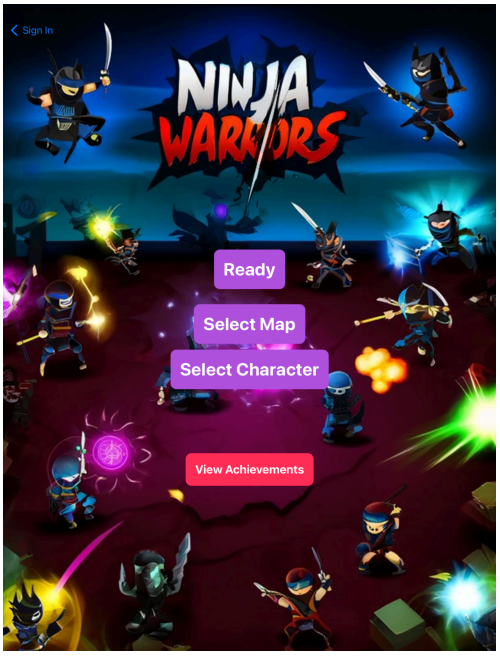
Achievement Page

Before diving into a game, click on the 'Achievements' button to view the list of possible achievements. Afterwards, simply click 'Back' to return to the lobby page.



Single Player

Now, if you're eager to dive into NinjaWarriors but prefer not to compete against others initially, you can opt for the Single Player mode. Simply click on the Single Player option in the lobby page. Following that, you'll be directed to the page depicted below.

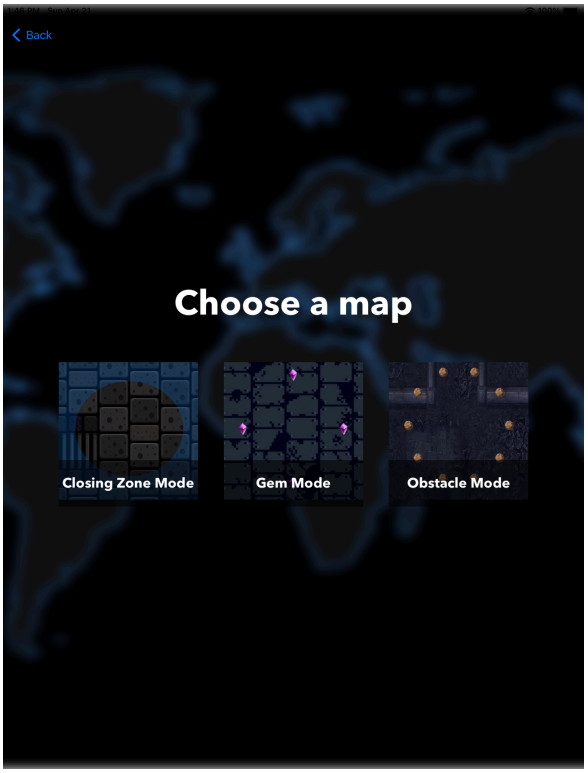


Character Selection

Before playing in single-player mode, consider changing your character by clicking on 'Select Character'. On the character selection screen, you'll find images, names, and corresponding sets of four skills for each character. Once you've chosen your preferred player from the character boxes, simply press 'Back' to return to the lobby page.

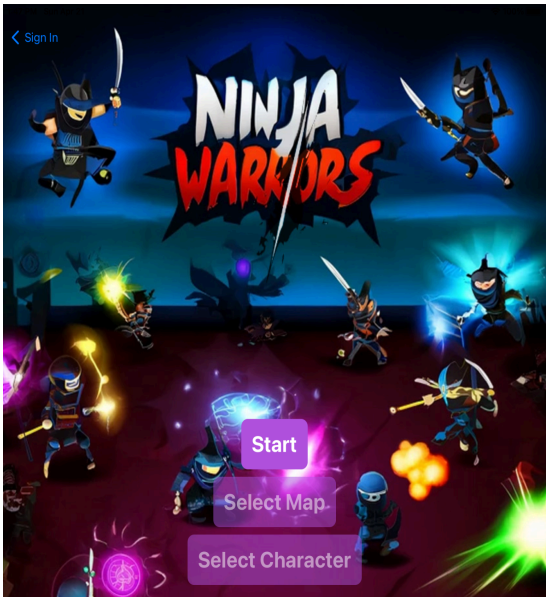
Map Selection

Even after selecting characters, you may wish to change the map, which also alters the game mode. Clicking on 'Select Map' will lead you to the page depicted below. Simply choose your desired map, then click 'Back' to return to the lobby page once again.



Start Page

After selecting your desired character and map, clicking on 'Ready' will promptly display the start button, as depicted below (for single-player mode). Clicking on 'Start' will then transport you to the actual gameplay page.

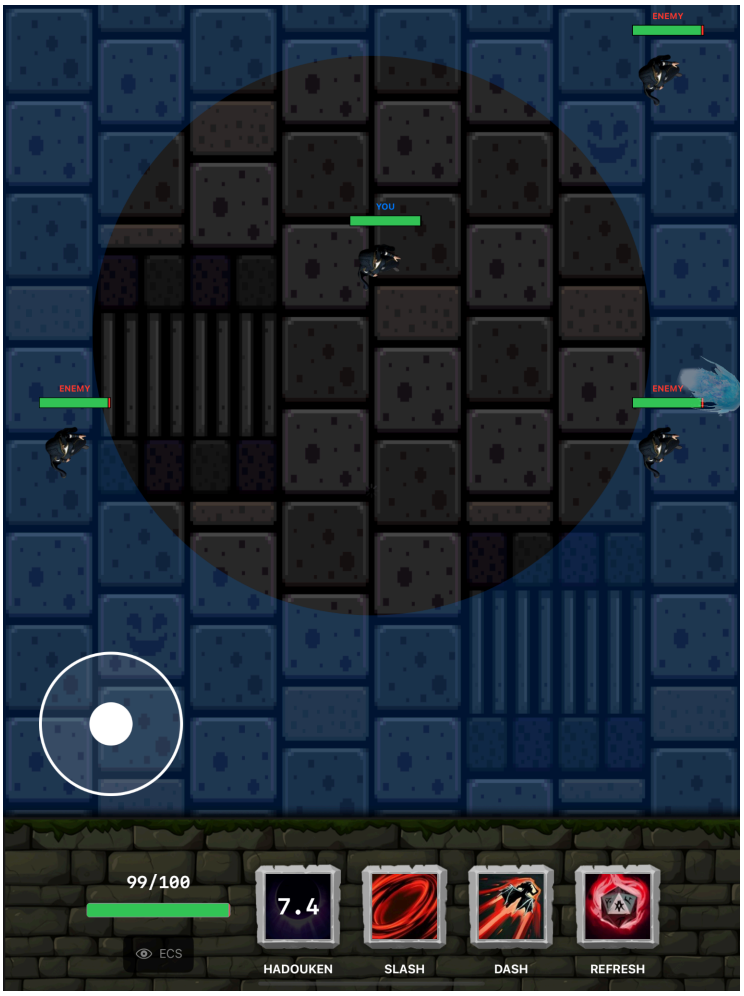


Gameplay Page & Closing zone game mode

Suppose you selected closing zone mode:

- The fog will start to enter from all 4 corners of the map to force the player to fight in a smaller map
- If the player is outside the safe zone which is a circle, i.e. they enter the fog, the player will receive 5 damage every 2 seconds.

This is the current view of the closing zone in single-player mode:



Walls

- Game area bounds, walls, are positioned at the sides of the map.
- Characters cannot pass through walls.
- If a character attempts to move towards a wall, they will remain stationary upon contact.
- If a wall is within 100 units and the character dashes towards it, they become stationary upon collision.
- The bounding area is defined by a line (or rectangle if considering all sides of the wall on the map).

Controlling the joystick

As mentioned in the how to play page, to move the ninja, drag the joystick for horizontal movement. The speed of the joystick is proportional to the distance between the centre of the inner joystick to the centre of the outer joystick. The skills button fetches the list of skills available for the player.

Casting Skills

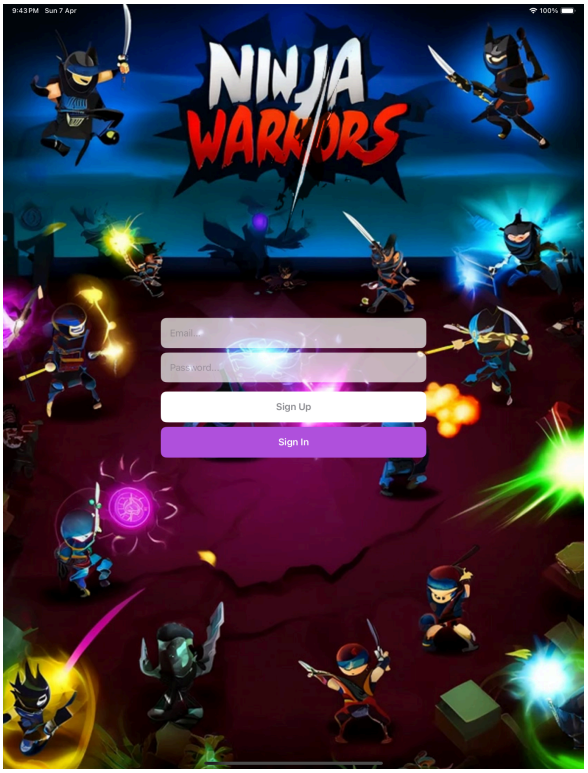
Also as mentioned in the how to play page, in the dashboard, there are 4 skills buttons that when pressed, will activate their skill logic and go into cooldown. After the cooldown duration has passed, the user can use the skills again.

Last Man Standing Mode

In single-player mode, the default game mode is the Last Man Standing mode. This means that when you are the last one alive, you win the game. Upon winning, you'll receive a list of achievements that you accomplished during the game.

Multiplayer Mode

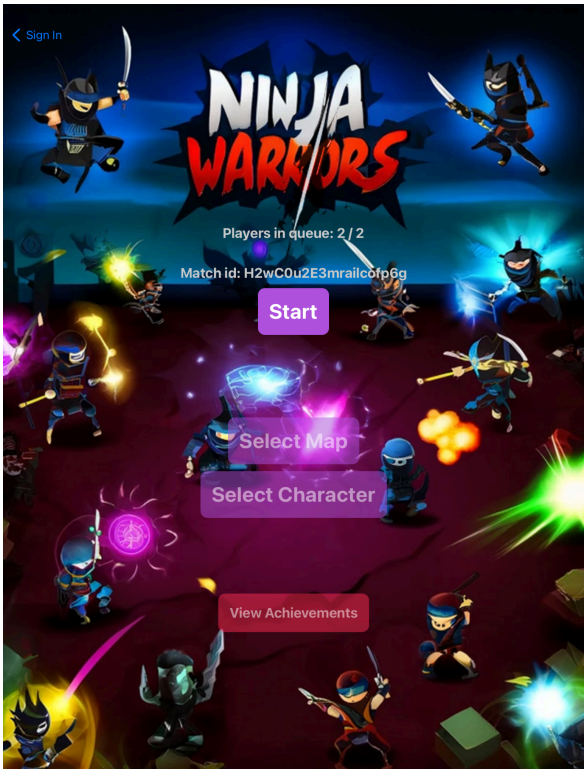
Now that you've familiarised yourself with the game, testing your skills against other players is a great way to gauge your abilities! You have two options: either play multiplayer mode by logging into an account or play as a guest. Should you choose to select Multiplayer Account, you will be shown with the page as seen below.



If you do not have an account yet, you can press sign up after keying in a valid email and a valid password (see sign up test plan on details of a valid email and valid password)

Once you have an account, you can use the same email and password to sign in in subsequent launches of the application.

If you opt to enter guest mode, please be aware that any achievements attained will not be associated with your original account. After selecting one of the two modes, the character selection page and the map selection page are identical to those in single-player mode. The only difference is that in multiplayer mode, you'll need to wait in a queue after clicking 'Ready'. In the figure below, the game can accommodate exactly 2 players, so the start button will appear only after both players have entered the queue, as depicted in the diagram below. Click start to play the game with others.



Congratulations! You've reached the end of the user guide. We hope you'll find as much joy in playing the game as we do!

1.4: Performance

The performance of a video game like Ninja Warriors is crucial to provide a smooth, engaging player experience and ensure the game's mechanics function as intended under various conditions. This section outlines the expected performance metrics for Ninja Warriors, including memory and compute usage, and network demands for both single-player and multiplayer modes.

Single Player Performance

Memory Usage

Initial Load

Upon game startup, the game loads the necessary assets into memory, including textures, models, sound effects, and initial game state configurations. Expected memory usage during initial load is approximately 10 MB in the main menu.

Runtime

During single player and multiplayer game modes, the average memory usage is expected to stabilise around 30 MB, with potential spikes up to 45 MB during intense scenes involving multiple entities and effects.

Compute Usage

CPU Load

The game's Entity Component System (ECS) efficiently handles various game mechanics such as movement, combat, which are computed each frame. On average, the CPU usage is expected to be moderate, around 10% on a standard mid-range gaming setup on initial load to main menu, with spikes up to 30-40% during gameplay and heavy combat phases.

Frame Processing Time

Each frame should ideally be processed within 16.67 milliseconds (ms) to achieve a smooth frame rate of 60 frames per second (FPS). In less complex scenes, frame times are expected to be around 10-12 ms, while in more demanding scenarios, such as multiple enemies and extensive use of skills, frame times could approach or slightly exceed 16.67 ms.

Expected Game Duration

Level Duration

Each level in single-player mode is designed to last between 5 to 10 minutes, depending on the player's skill levels and game modes.

Multiplayer Performance

Memory Usage

Per Player Increase

Each additional player in a multiplayer session slightly increases the total memory usage due to the need to manage additional player data and synchronise states. An estimated increase of 15 MB per player is expected.

Total Usage

For a four-player session, total memory usage is expected to peak around 60-80 MB, subject to the specifics of the game scene and the number and complexity of entities involved.

Compute Usage

CPU Load

Multiplayer mode generally requires more CPU resources due to the increased number of entities and the need for frequent state synchronisation. Average CPU usage might increase to 30-40% with peaks up to 60% during intense multiplayer interactions.

Frame Processing Time

The target frame time of 16.67 ms is still maintained; however, network delays and synchronisation overhead can cause variability. Frame times in multiplayer sessions are expected to range from 15 to 25 ms in standard conditions.

Network Usage

Data Transfers

Each player's actions, movements, and game events are transmitted to all other players. Average network usage is expected to be around 100 kilobytes per second (KB/s) per player.

Latency Impact

To maintain a smooth gameplay experience, a network latency of less than 100 milliseconds is ideal, with latencies up to 150 milliseconds being acceptable before players might notice significant lag.

Guest vs. Signed-in Sessions

Signed-in players will have their game progress and player states saved to the server, which requires additional data transfers at the beginning and end of sessions. Guest players' data, while temporarily stored during the session, does not incur these additional data costs.

Signed-in sessions include an initial authentication step, which involves minor additional network usage and compute resources to verify player credentials.

Conclusion

In summary, Ninja Warriors is designed to perform efficiently in both single-player and multiplayer modes. The game's ECS architecture helps optimise compute resources, while careful management of assets ensures memory usage remains within acceptable limits. Multiplayer mode introduces additional complexities, particularly with network synchronisation, but the game is engineered to handle these effectively to maintain a responsive and enjoyable player experience. Ensuring the game operates smoothly across various devices and network conditions is a priority, necessitating ongoing performance monitoring and optimization as the game scales and evolves.

2: Design

2.1: Overview

NinjaWarriors adopts two architectures: **Model-View-ViewModel (MVVM)** for the game mode selection, character selection, how to play display, authentication, log in, sign up and lobby page, and **Entity-Component-System (ECS)** for the ninja characters, ninja skills, lifecycle, and rendering during the actual single player and multiplayer game.

In ECS, an Entity is a class that contains a unique EntityID and represents anything appearing in the game world. Currently, the entities are: Player, ClosingZone, Obstacle, SlashAOE, Hadouken, and Gem. A Component is a class that contains raw data and strictly represents a single behaviour, such as Rigidbody, Collider, etc. It is important to note that a component has an **unowned and not weak reference** to its entity. Lastly, the System keeps track of lists of components for modification.

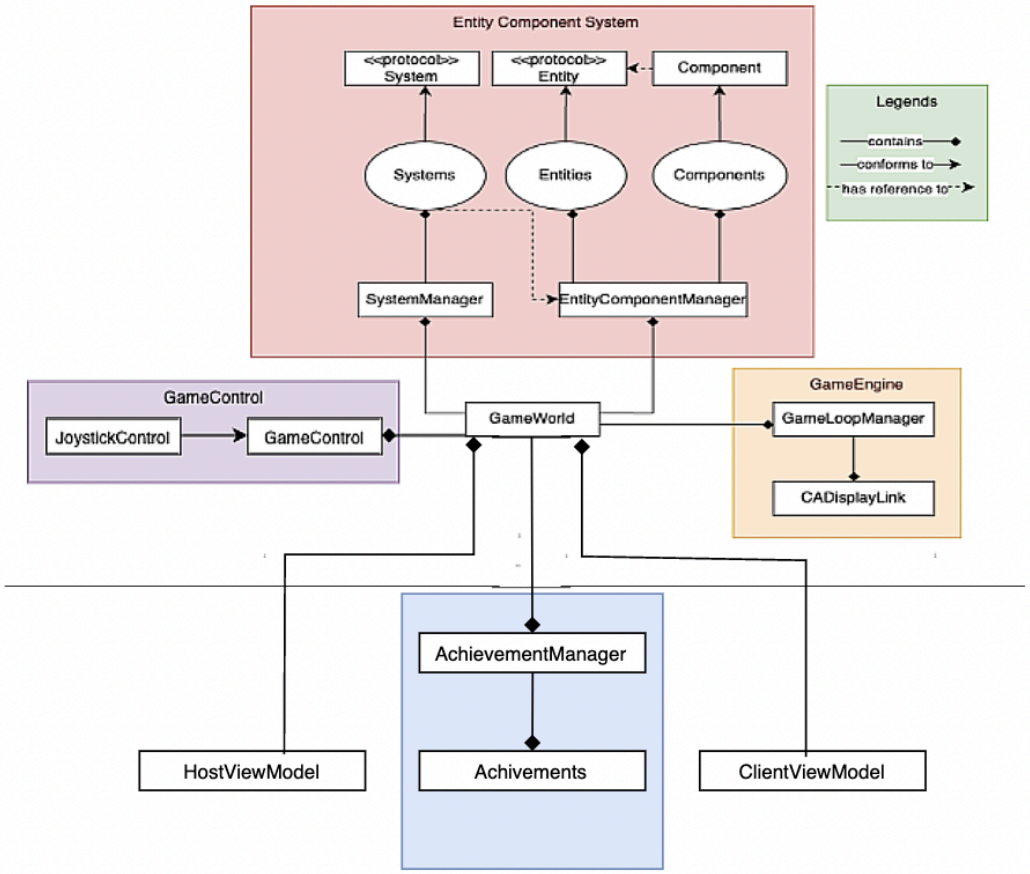
We chose Entity-Component-System due to our use of traditional OOP. Using OOP, if we want our ninja character to be movable, we would have to inherit from a movable class. However, this also means that we can no longer inherit from any other classes, as Swift restricts inheritance to only one class at most. Besides this restriction, modelling behaviours through inheritance is not suitable for games, as things can quickly become messy with more complex relationships between subclasses and parents.

We opted to draw inspiration from Unity and lean towards composition over inheritance. For instance, if we want to make the entity movable, we simply compose the Rigidbody component to the entity.

Referring to the high-level architecture provided below in Figure 1, HostViewModel contains GameWorld, which in turn includes SystemManager, EntityComponentManager, and GameLoopManager. This signifies that GameWorld acts as the bridge between the ECS, the view-model, and the game engine. Systems have a reference to the **same instance of a single** EntityComponentManager. The trade-offs in implementation are discussed in dependency injection vs singleton. Each concrete component in Components inherits from the Component superclass. Similarly, each

concrete entity in Entities implements the Entity protocol. Lastly, each system in Systems implements the System protocol.

Figure 1: High-Level Overview of Architecture



Design Issues

As seen from the figure above, we made a design choice to have game worlds and game loops on both the host and the client. Having game loops on both the client and the host facilitated easier implementation, but it also increased the occurrence of race conditions. When only one device had the game loop, race conditions were not apparent as the clients simply observed any changes that occurred. However, with each device having its own game loop and game world, numerous scenarios arose where interleaving of instructions was possible. Consequently, the `EventQueue` was employed to ensure atomic read and writes and prevent synchronisation issues.

2.2: Runtime Structure

Custom Point, Vector, Line, and Shape data types

Custom Point, Vector data types

One of the pivotal custom data types we employed was `Line`. These lines were utilised in constructing shapes, another essential custom data type in our system. `Lines` are constructed using custom `Point` and `Vector` primitives, with `Vector` being built upon the custom `Point` primitive. Let's first dive into the purpose of `Point` and `Vector`.

The rationale behind employing custom `Vector` and `Point` structs, instead of utilising `CGPoint` and `CGVector`, stemmed from our desire to invoke functions within the struct rather than redundantly repeating ourselves, thereby adhering to the DRY (Don't Repeat Yourself) software engineering principle.

In the context of the application, performing operations such as vector scaling due to resizing, dot product calculations, cross product computations between two vectors, obtaining the complement of a vector, normalising a vector, determining angles between two vectors, and altering the direction of the vector within a single function call each were deemed crucial. This approach maintains the cleanliness and organisation of the application base, particularly when users execute actions such as movement, skill activation, or collision with a boundary.

Representation Invariants of Point data type

Firstly, it must have non-negative coordinates.

Also, we chose to adopt the standard SwiftUI graphics Cartesian coordinate system, where (0,0) represents the top left corner. However, we use the center of each object as a reference point against the top left, rather than strictly adhering to the top left for all calculations, for the sake of convenience.

An alternative approach would be to consistently use the top left corner as the reference point for all calculations. This would eliminate the need for half-length variables (such as halfWidth, radius, halfHeight), as the entire length of the object could be stored instead. Although this alternative might simplify calculations, we opted for the center approach as it intuitively aligns better with our understanding of object placement and manipulation.

Custom Line data type

Now that the purposes of the custom `Vector` and `Point` structs have been elucidated, let's delve into the crucial purpose of the `Line` struct. Each line comprises a start point, an end point, and the vector connecting these two points. This structure allows for the representation of shapes through arrays of lines, where the end point of one line coincides with the start point of another, or vice versa. Consequently, this interconnected network of lines enables the modelling of shapes without being confined solely to horizontal or vertical

boundaries. By extending this flexibility to any boundaries, our application becomes highly adaptable and extensible.

Representation Invariants of Line data type

- Each vertex must be found in one of the line's start or end point
- There cannot be duplicate vertices
- There cannot be duplicate edges
- A line segment defined by a start point A and an end point B is equivalent to a line segment defined by start point B and end point A. In other words, a line segment is directionless, and its orientation is not inherently defined by the order of its points
- A line segment cannot have the same start point and end point. While this configuration may be mathematically valid, it lacks practical usefulness in the application. If a line segment has identical start and end points, it would be more appropriately represented as a point

Custom Line data type alternative

An alternative to implementing a `Line` struct could involve assuming boundaries as strictly horizontal and vertical. However, this approach is flawed because entities in the game, as will be explained later, can have shapes of any kind. Additionally, while the boundaries of an iPad may be straight, lines might not seem necessary to model them. Yet, there could be instances where the actual boundary of the game, such as a pit or a rock, is not straight. Therefore, simply modelling the boundaries of every game object as a square or rectangle would not be accurate, despite its ease of implementation. `Lines` will be utilised in `Shapes`, as explained above, to accurately model how the object would appear.

Custom Shape data types

Moving on to the `Shape` class, it incorporates several essential attributes: a center, halfLength, orientation, an array of points (to denote vertices), and an array of lines (to depict edges).

Subclasses of `Shape`, such as `RectangleShape` and `TriangleShape`, naturally inherit from `Shape` and adhere to specific representation invariants regarding the number of vertices and edges for each shape. For instance, a triangle must always have three vertices and three edges. While it's conceivable to create different types of triangles, such as equilateral triangles where all sides are equal, we opted not to delve into this level of granularity to strike a balance between abstraction and development time. Implementing numerous triangle types could lead to a **class explosion** which we aimed to avoid. Instead, we focused on using `Shape` as the base class and introducing `TriangleShape` and `RectangleShape` subclasses. Any other shape beyond these would simply be represented as a generic `Shape`.

There might be some confusion regarding why our **colliders** have a `colliderShape` attribute rather than the entity itself. This design choice mirrors Unity's approach. However, we're not blindly emulating Unity; we've embraced their rationale that a collider should maintain its own shape. This distinction is crucial because the entity's shape could be highly

complex, leading to significant runtime complexity when checking for intersections. If every single line (as described earlier) had to be checked for collision, it would impose a considerable performance overhead, especially since every line for every entity must be checked against all other entities in the `GameWorld` (will be elaborated on later). Therefore, by assigning a collider its own shape, we can perform **hitbox approximation**, drastically reducing computational costs and time complexity.

Game Update Loop

The game update loop is the main driver of NinjaWarriors. It calls the system's update function 60 times per second (60 FPS), which triggers updates and changes throughout the various systems in the ECS architecture.

Internally, the `GameLoopHandler` utilises Swift's `CADisplayLink`. We've configured the loop so that the refresh rate is kept to a fixed 60 FPS. Upon each invocation of the `loop()` function in `GameLoopHandler`, a closure named `onUpdate` is triggered. This setup enables the `GameWorld`, which contains `GameLoopHandler` to performs two tasks on every frame:

1. Invoke systemManager.`update(after: deltaTime)`. This method proceeds to iterate through all the systems and performs the heavy processing work to compute the next outcome based on the current states.
2. Invoke hostViewModel's `updateViewModel()`, which publishes the latest state to the real-time database, and also displays the relevant new states to the user.

Hence, by utilising closures, it enables two-way communication. One way is through the normal instantiation path, where the GameWorld creates a GameLoopManager. The other way is by propagating information whenever a new loop or frame runs from the instantiated GameLoopManager back to the owner, which is the GameWorld.

View - View Model

Instead of relying on SwiftUI's `@Published` property wrapper, which automatically updates the variable upon any change, we opted to use `objectWillChange.send()`. This approach grants us finer control over when the view is updated, resulting in lower latency.

While we are on topic of views, it is apt to mention that `EntityView` is customised for a specific purpose. Below is an image from `EntityView`.

Figure 2: EntityView Frontend Design (MVVM)

An entity (player) with Health, Dodge, Sprite components

Within HostView, each entity retrieved from the game's viewmodel is rendered as an **EntityView,** taking in an **EntityViewModel(components: [Component])** as its parameter. The role of EntityViewModel is to monitor any changes in the components, such that the Entity's view is updated whenever there are changes.

Within EntityView, it is composed of multiple layers of Views that are rendered based on the different components that make up the Entity. For example, an entity would usually be rendered as its Sprite component image. If it has a Health component, EntityView will render its health bar, and if it has an activated Dodge component, a shield will be visible wrapping the entity.

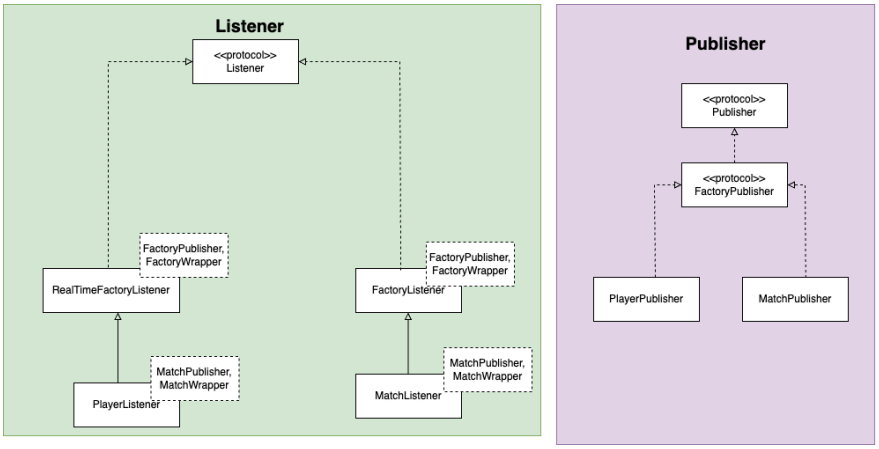
This method of rendering the Entity makes use of the composition-nature of ECS, and allows us to build the visual representation of the Entity through its components.

Realtime Database to EntityComponentManager communication

To ensure that the `EntityComponentManager` receives notifications when the RealTimeManager undergoes modifications, a listener must be invoked. Instead of monitoring the entire database, a streamlined approach was adopted, focusing solely on the specific matchId.

Furthermore, FactoryListeners, FactoryPublishers, as well as FactoryWrappers were implemented in order to have the listeners as extensible as possible.

Figure 3: Listener, Publisher, and Wrapper



Codable Wrappers

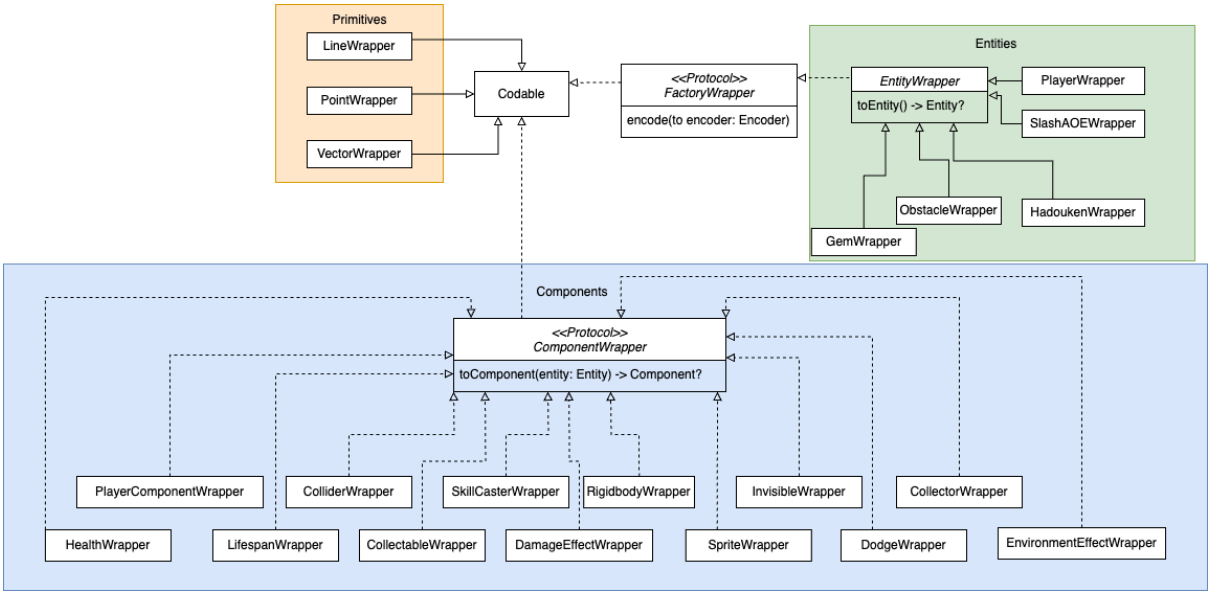
To avoid coupling with `Codable`, **none of the classes and structs in the game** inherit from `Codable`. However, this approach prevents the direct encoding and decoding of these classes and structs into and from `Firebase`.

Therefore, wrappers are employed to facilitate conformance to `Codable`. Currently, each component, entity, and primitive has its own generic wrapper. These generic wrappers enable the encoding and decoding of any struct or class that conforms to them.

While it may seem more convenient to forgo wrappers and simply make every struct and class conform to `Codable`, this approach carries potential risks. If the `Codable` protocol undergoes changes to its methods, the entire application could crash. Conversely, using wrappers allows for straightforward hot-swapping of a new wrapper that conforms to the updated API calls, ensuring the continued operation of the application.

Each wrapper includes a method to convert it into a non-codable version for internal application use. Likewise, each non-codable version includes a `wrapper()` function to wrap it, enabling conformance to `Codable` before encoding and decoding into `Firebase`. Further details can be found in Figure 4 below. It is important to note that with `Wrapper` present, it essentially means that every class that needs to be encoded and decoded must have its corresponding `Wrapper`.

Figure 4: Codable Wrappers

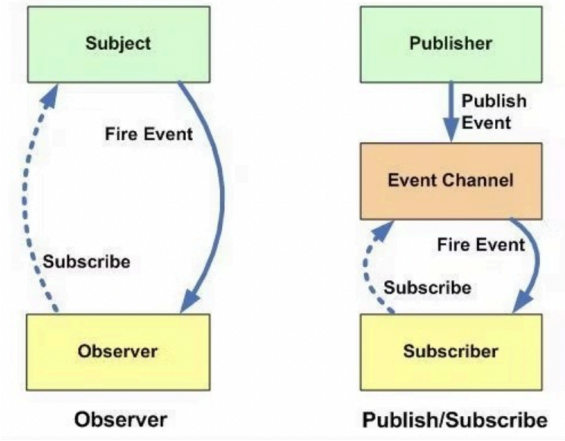


These generic wrappers prove useful when implementing publisher-subscriber pattern, which is explained in the Multiplayer section below.

Multiplayer (pub-sub vs observer)

There are two ways to get live updates from other players. One is the observer pattern, and the other is the publisher-subscriber pattern. The difference can be seen in Figure 5 below.

Figure 5: Observer vs Pub-Sub



The current implementation involves listeners (also known as observers) being aware of the observable. The observable is the real-time database reference, with the corresponding matchId serving as the key. Therefore, when a player publishes a new state, the observers who have invoked: `self.databaseReference?.observe(.value)` will be able to see the changes.

To ensure the application remains open to extension and closed to modification, the listeners are parameterized with generic types: `FactoryPublisher` and `FactoryWrapper`. Moreover, these listeners inherit from the Listener protocol. Consequently, we can easily incorporate an array of listeners and initiate observation of Firebase by simply invoking `startListening()` in a loop. This signifies that the addition of any new entity will not necessitate any changes to `HostViewModel`; we only need to include the corresponding concrete `Listener` and `Wrapper` for the new entity. This approach preserves the application's openness to extension while restricting modifications.

Reason why observer pattern was chosen

When deciding between the Observer pattern and the Publisher-Subscriber (Pub-Sub) model for integrating Firebase Realtime Database functionality, we carefully considered the real-time nature of Firebase. Given that Firebase operates on a continuous stream of data changes with immediate updates to all subscribed clients, the Observer pattern appeared more suitable. This pattern establishes a direct relationship between application components, such as listeners, and the Firebase objects they observe.

We opted not to use third-party libraries to avoid coupling with external dependencies. Implementing our own pub-sub system would entail addressing challenges such as event piggybacking, latency handling, queueing theory, and other complex topics that are beyond the scope of this application's goals. Since our primary aim for this application is to enhance software engineering skills rather than focusing extensively on networking intricacies, the observer pattern was chosen in the end.

EntityComponentManager dictionary mapping data types

We structured our components and entire Entity-Component-System (ECS) framework closely following **Unity's 2D ECS framework**. Currently, our ECM employs three mappings within its runtime structure:

```
var entityComponentMap: [EntityID: Set<Component>]
var entityMap: [EntityID: Entity]
var componentMap: [ComponentType: Set<Component>]
```

Firstly, there existed a mapping from the `EntityID`, represented as a String, to a set of `Components`. While utilising a set of component IDs offered simplicity, it effectively resulted in **type erasure**, as the actual component type was not utilised during runtime. Consequently, adhering to better software engineering principles, we opted to store all the components instead.

Additionally, there existed a mapping from the `EntityID` to the corresponding `Entity` object. This mapping facilitated the retrieval of relevant entities when the `EntityID` was fetched from the real-time database. Consequently, we consistently utilised the same entity memory reference, thereby avoiding unowned reference errors that may occur when attempting to access a deallocated entity's unowned reference.

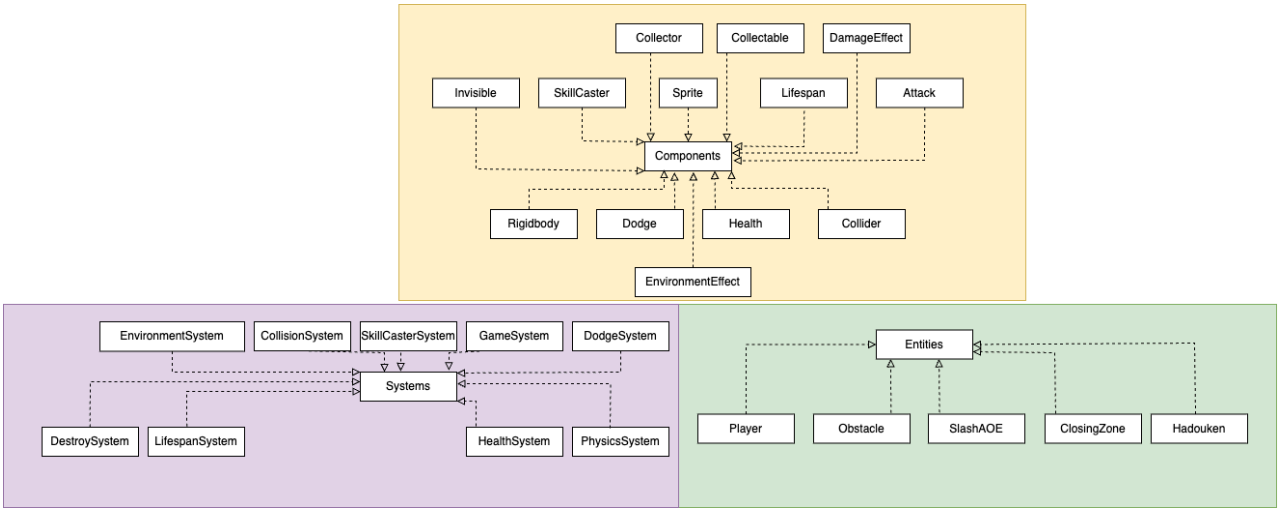
An alternative approach would be to store entities in an array. However, this method is clearly inferior because fetching an entity would require traversing the entire array, resulting in a time complexity of O(n), where n is the number of entities. In contrast, utilising the entityMap provides an amortised constant time complexity of O(1) for entity retrieval, offering significantly improved performance.

The third mapping involves associating each component type with the set of components conforming to that type. This setup enables the system to efficiently retrieve the relevant component types as needed.

2.3: Module Structure

Figure 6 below illustrates the high level structure of all the entities, components, and systems that have been employed in NinjaWarriors.

Figure 6: High level module structure of Entity Component System (ECS)



Once again, it's important to note that each entity and components has a wrapper that conforms to Codable since conforming to Codable is required to be stored in the database. If Extensions, Factory, Adapter classes, Generic classes, and wrappers are not explicitly listed, it should be assumed that most of the classes conform to one of them to adhere to the Open-Closed Software Engineering (SWE) principle.

Entity Component System Module Structure

The **ECS (Entity Component System)** is where the main game logic lies. `Players` and `Obstacles`, `SlashAOE`, `Hadouken`, and `ClosingZone` are represented as entities in the ECS.

Each entity can be mapped to multiple components in the `EntityComponentManager`, but at most one component of each type. The currently implemented component types are shown in the table below:

Components

Component	Rationale
SkillCaster	Contains all the relevant skill primitives associated with the player. The detailed breakdown of skills can be seen in Figure 7 just below the table
Sprite	Abstracts the image of each component to be rendered in the view.

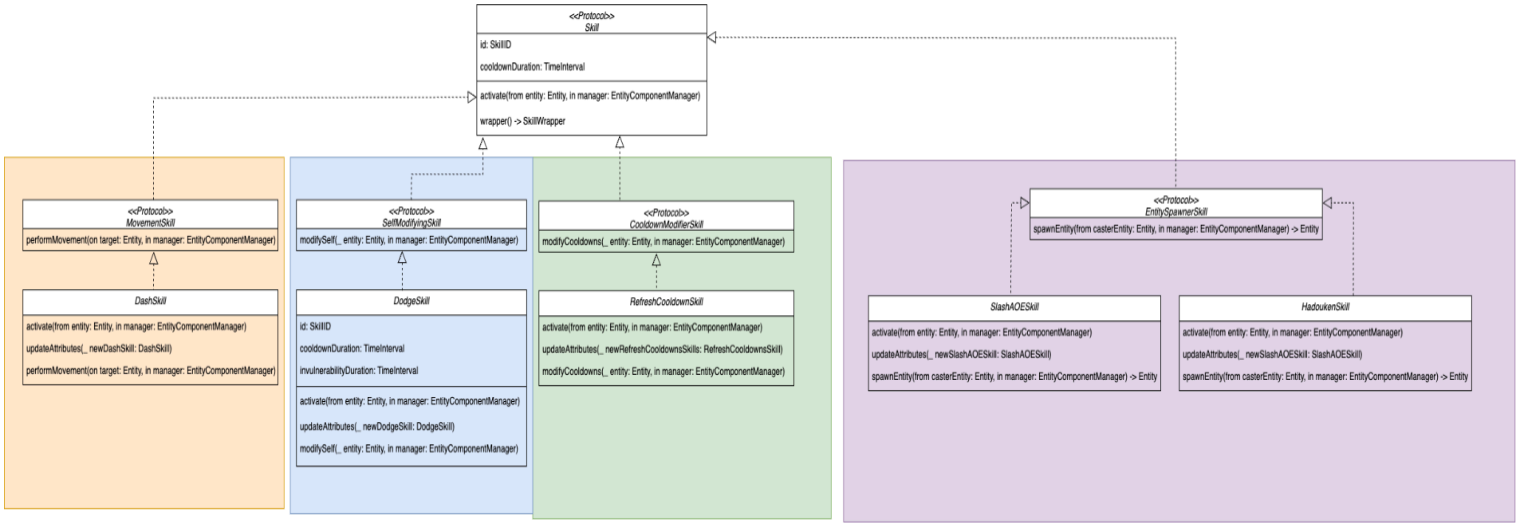
Lifespan	Adds a lifespan duration to the player. After a specified time period indicated in the component, the player will be removed from the game by the corresponding system.
Dodge	Enables the player to avoid inflicting damage for 2 seconds, even when within the radius of the attack.
Attack	Targets entities within the player's radius who initiated the attack.
Health	Has a maximum health of 100, and a minimum health of 0. Once the player's health reaches 0, the player will be removed from the game.
Collider	Enables objects to respond to collisions & keeps track of a set of collided entities at every game tick. Contains `colliderShape` to respond to collisions based on line intersections
Rigidbody	Enables objects to respond to physics and game controls. Contains attributes such as mass, velocity and angularVelocity used to calculate movement and rotation
EnvironmentEffect	Let us create zones/environments in the map where being inside or outside the zone can have effects on other entities. For example, the Closing Zone has the effect of reducing health for entities outside of the closing zone.
Invisible	Allows the entity to pass through other entities that contain an Invisible component as well
Collector	Allows the entity to collect entities that has a collectable component
Collectable	Allows the entity to be collected by an entity that has a collector component
PlayerComponent	Marks an entity as a player to provide player specific logic to the entity
DamageEffect	Damages the entity

As indicated in the table, the details of the skills contained within the skillCaster will now be explained. Since skills are such a core feature of our game, they are implemented as primitives, and define the behaviour of the skills. We opted to design Skills as a **primitive**, like the Shape type, meaning that they are not an Entity, Component, nor System. As they deal with various actions such as modifying certain properties, which do not deal with creating any game entities, they are not treated as Entities but as classes that contain the behaviour of the skill when executed.

Each skill can have one of four different functionalities: spawning an entity, modifying oneself, affecting one's cooldown, or serving as a movement skill. Many specific concrete

skills extend from one of these four overarching types. For instance, both `SlashAOE` and `HadoukenSkill` spawn entities, so they both extend from the `EntitySpawnerSkill` class. More details regarding the functions of each concrete skill can be found in the skill class diagram below.

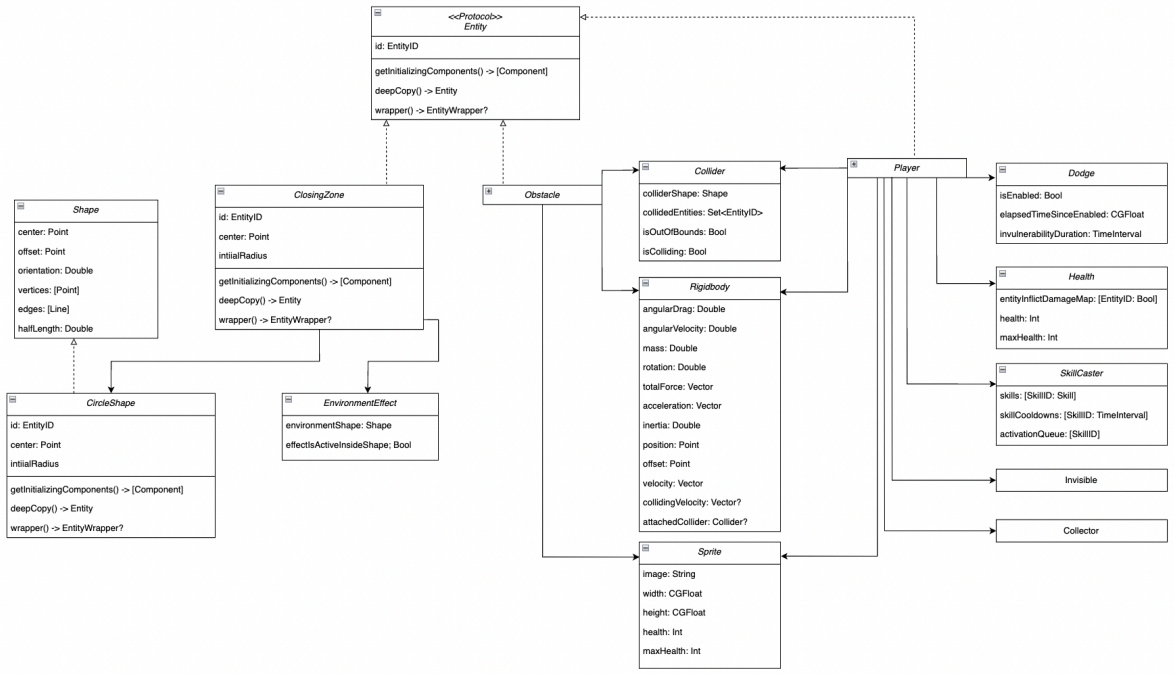
Figure 7: Skill Class Diagram



Entities

Now that we have covered important architectural diagrams, the rationale for using ECS, and the rationale for each component, here is the detailed class diagram for each entity and its associated components (note that entities do not keep references to components; associations are kept in `EntityComponentManager`)

Figure 8: Entity-Components Class Diagram

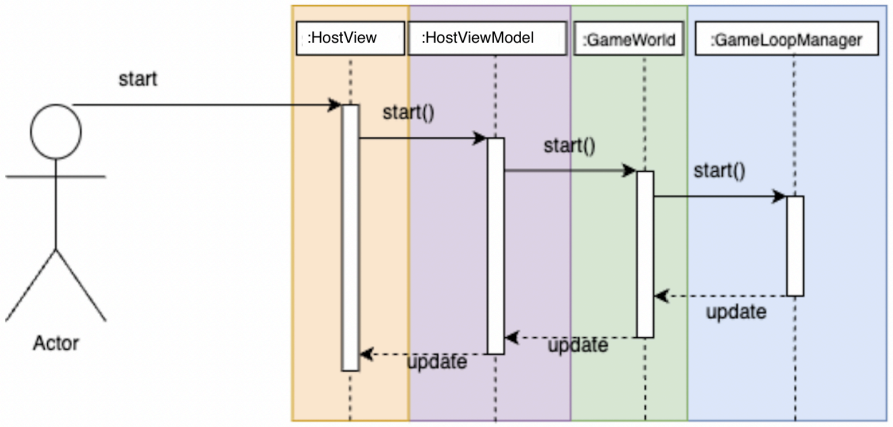


With entities and components being discussed, the next step is to discuss the systems. Each system does not hold dependency to a single component alone. While there are cases where this may be true, in general, in our version of ECS, the system can refer to more than one component in order to achieve the desired behaviour.

NinjaWarriors utilises a **client-host design,** establishing the host as the **single source of truth**. Despite this, all devices maintain a continuous game loop at 60 FPS. This approach was selected to address the limitation of relying solely on local data without fetching from the real-time database, which lacks a unified **source of truth**. Consequently, one of the four devices is designated as the host, while the remaining three serve as clients, ensuring the establishment of a **single source of truth**.

The sequence diagram in Figure 9 illustrates what happens when a device starts the game.

Figure 9: Start Sequence Diagram



With this in mind, let's proceed to RigidbodyHandler to clarify what occurs when a client moves. However, for other subsequent systems, only the simplified sequence diagram will be shown for brevity.

Systems

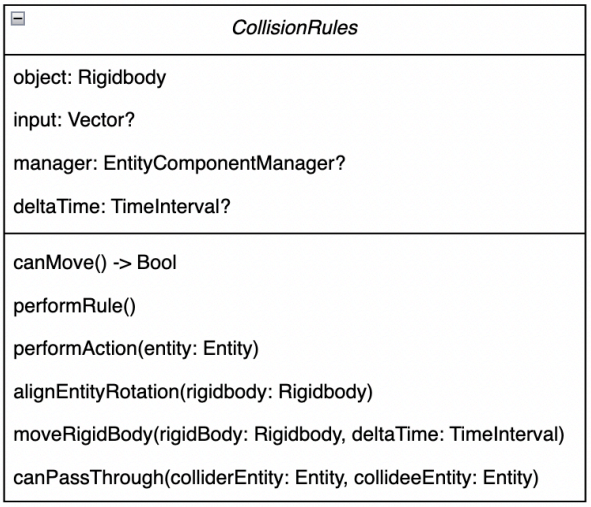
RigidbodyHandler System

The primary function of `RigidBodyHandler` is to adjust the position, speed, and rotation of the entity with a `Rigidbody` component. However, straightforward modification of these attributes on every tick is insufficient due to the presence of boundaries and obstacles. Therefore, before any movement occurs, `RigidBodyHandler` must verify that updating the rigidbody's attributes will not result in a collision. Only then should the modifications be applied.

To adhere to good software engineering principles, we avoided hardcoding collision checks in the `RigidbodyHandler`. Instead, we abstracted this functionality into `CollisionRules`. `CollisionRules` checks various components of an entity. If no rules are specified, the `Rigidbody` component of the entity is allowed to be modified. Otherwise,

modification of the `Rigidbody` component is disallowed. To provide a clearer understanding of how `CollisionRules` operates, a class diagram has been included below.

Figure 10: Collision Rules Class Diagram



When the client moves, it publishes to its own `EntityComponentManager`. Since the client has a game loop that publishes and fetches to and from the real-time database at 60 FPS, it uploads the next position to the real-time database. The host listens for a change and updates its own `EntityComponentManager` with the new changes from the client. Then, it passes this new state through the `RigidbodyHandler` system. If no collision is detected, the host propagates the new position to the real-time database, and the client fetches it and displays the new position. Additionally, the host displays the new client's position. Below is the sequence diagram of how `CollisionRules` interacts with `RigidbodyHandler` from the client's perspective. The sequence diagrams can be seen below.

Figure 11: RigidbodyHandler Sequence Diagram ref

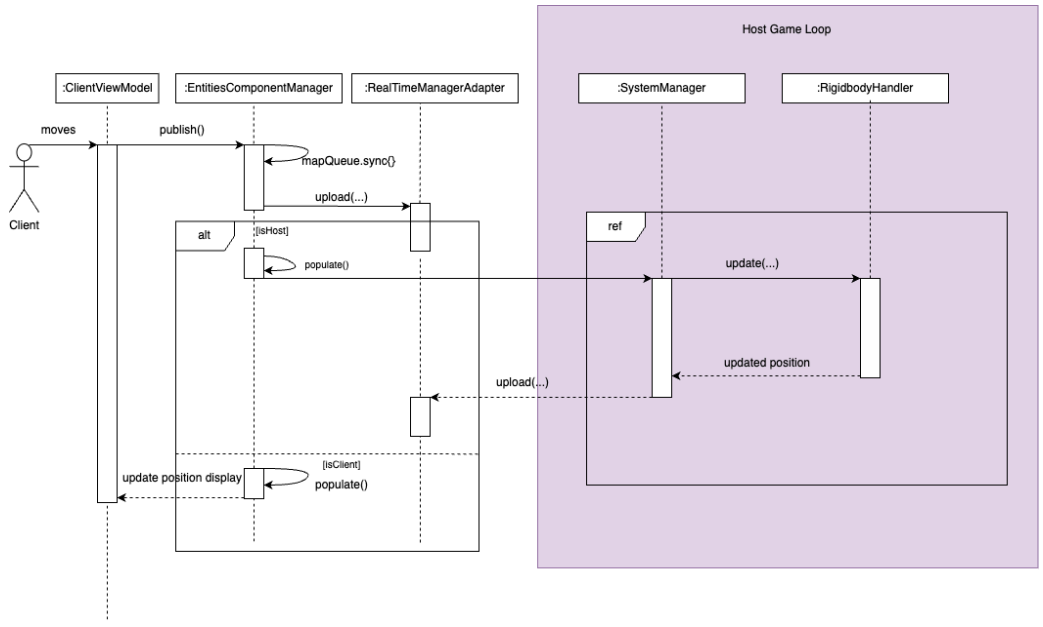
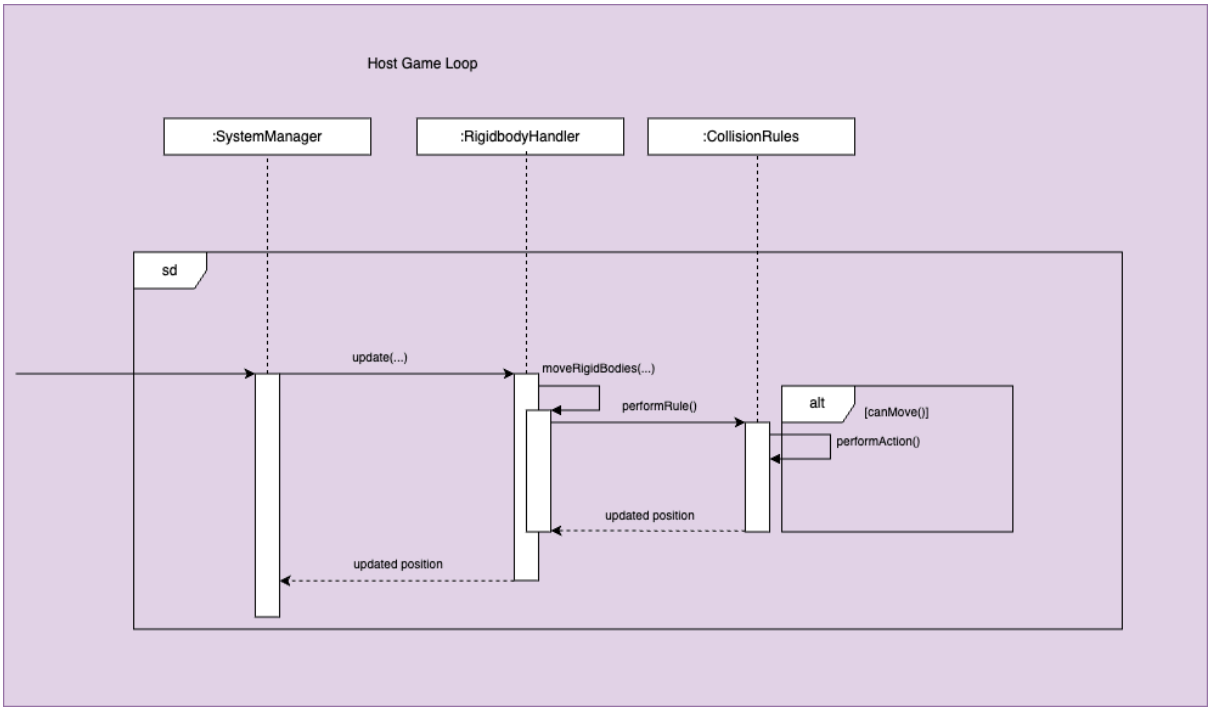


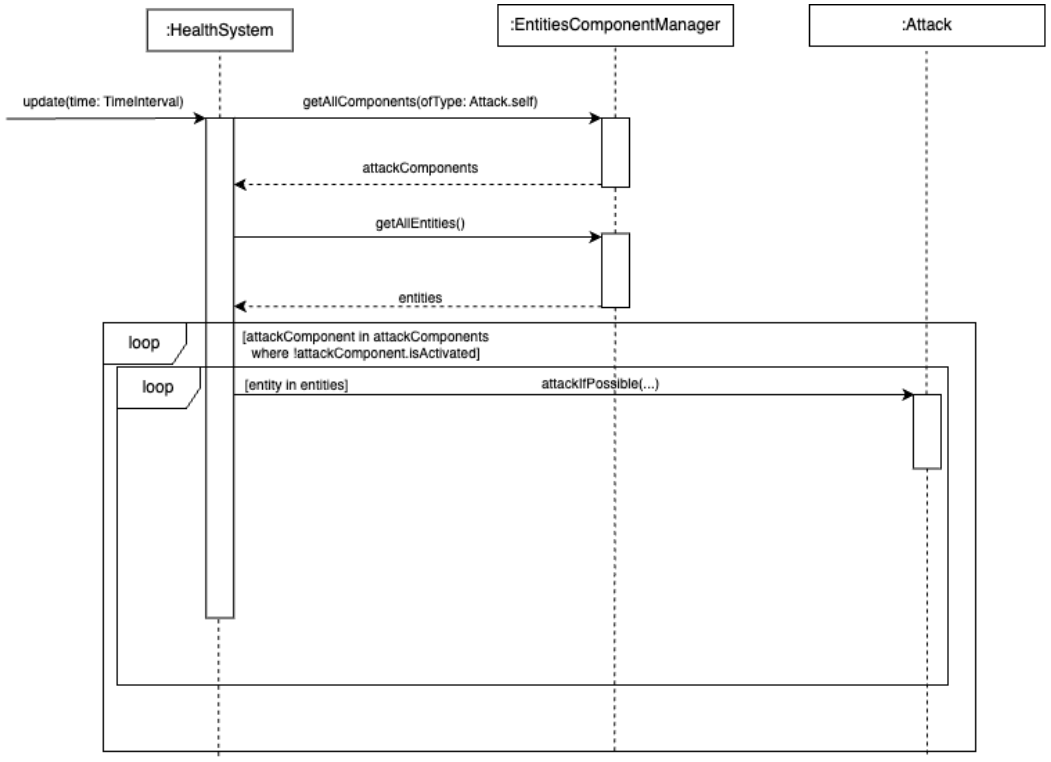
Figure 12: Rigidbody Handler Sequence Diagram sd



Health System

For the health systems, it fetches all `Attack` components, as well as all entities. Then, it checks if the attack is possible, and if so, attacks the target by attaching a `DamageEffect` component to the target. Below is the sequence diagram for the health systems.

Figure 13: HealthSystem Sequence Diagram



Combat System

The Combat System is a single system that aims to handle all damage and healing events between players in the game. For example, when the SlashAOE entity applies damage to a Player, it does not directly modify its health component. Instead, it adds a DamageEffect ‘debuff’ to the Player that can have initial damage, and a damage per tick for a specified duration.

Any priorities for damage, damage avoiding (i.e. Dodge), is handled within this Combat system. In an extended version of Ninja Warriors, if players had different attributes for damage resistance, or damage over time resistance, the calculations would be calculated here, in the Combat system.

The alternative to this was to implement an event bus, whereby each player has their own subscriber listener that reacts to damage events posted to their specific event channel. When a player is attacked, instead of adding a DamageEffect component, the attacking entity would publish an attack event containing the damage details (initial damage, damage per tick, duration) to the targeted player's event channel. The Combat System would then process these events from each player's event queue, applying damage accordingly over time as specified by the event's parameters.

In this event-driven model, the sequence of actions following an attack would be as follows:

- Attack Execution: An attacker executes an attack, which triggers an event creation with all the relevant damage information (initial damage, damage per tick, and duration).
- Event Posting: This event is posted to the targeted player's dedicated event channel, which is part of a larger event bus system that handles all such interactions in the game environment.
- Event Handling: The player's event listener detects the new event, retrieves it from the queue, and processes the damage according to the event's specifications. This processing might involve immediately applying the initial damage and setting up timed effects for the damage per tick.
- Damage Application: The Health System periodically checks for any active DamageEffect events on each player and applies tick damage as necessary based on the event's frequency and duration stipulations.
- Event Cleanup: Once the duration of the damage event expires, the event is removed from the player's queue, and no further tick damage is applied from that specific event.

Advantages of Using an Event Bus:

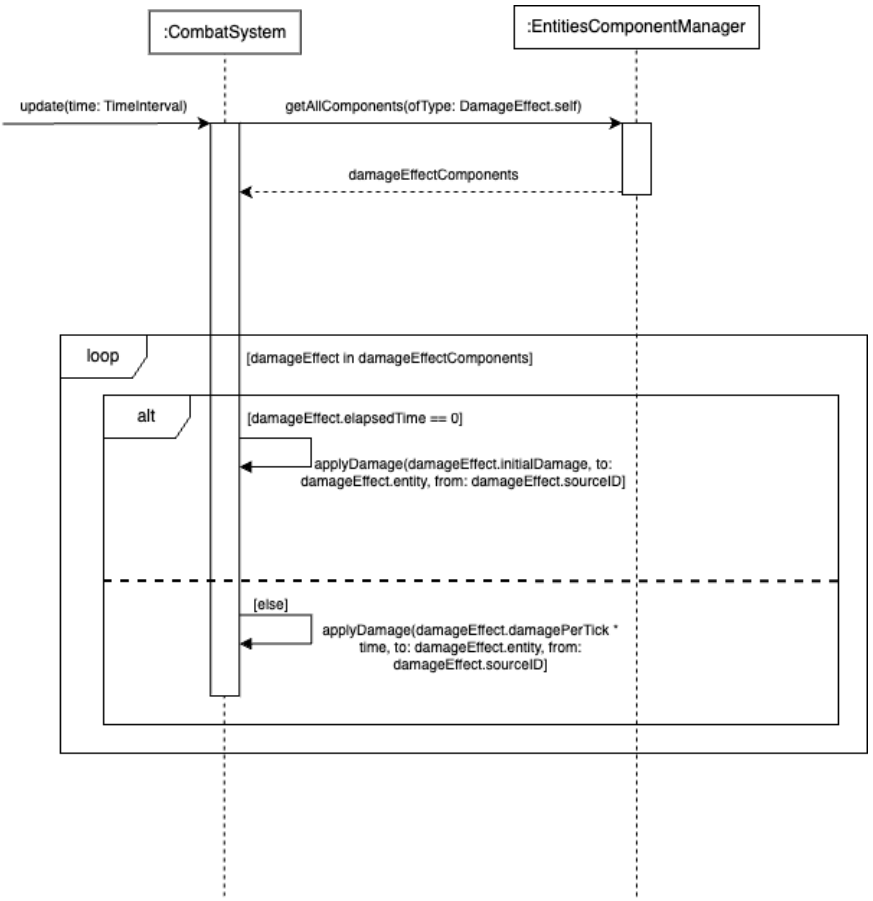
- Decoupling of Game Logic: Combat interactions are handled through events, meaning the Combat System does not need direct access to the components of other entities it interacts with, such as the Health components of players. This decouples the systems for greater modularity and easier maintenance.
- Flexibility in Handling Effects: The system can easily introduce new types of effects or modify existing ones without major changes to the Combat System's implementation, as these can be encapsulated entirely within the event data.

- Asynchronous Processing Capability: Damage processing can be handled asynchronously, allowing for smoother gameplay as the computational load of applying complex or numerous damage effects can be spread out over multiple frames or server ticks.

However, given that we wanted to implement customizable DamageEffects since our game revolves around the implementation of custom skills and buffs/debuffs, we ultimately decided to stick to applying only a DamageEffect component. Furthermore, as our game is not designed around a universal event bus system to handle all game events, and because most of our synchronisation techniques are synchronising the Entity and Component data, we decided to replace this 'damage' event with the added DamageEffect Component, which achieves the similar purpose.

After the health system has attached a `DamageEffect` component to the target when attack is possible, the Combat system will first fetch all `DamageEffect`, and then decrement the health of each entity that has a `DamageEffect` component. There are two ways of decreasing the health, the first is by the `initialValue`, and the subsequent tick is by the `damagePerTick`. Below is the sequence diagram for the combat system. The sequence diagram can be seen below.

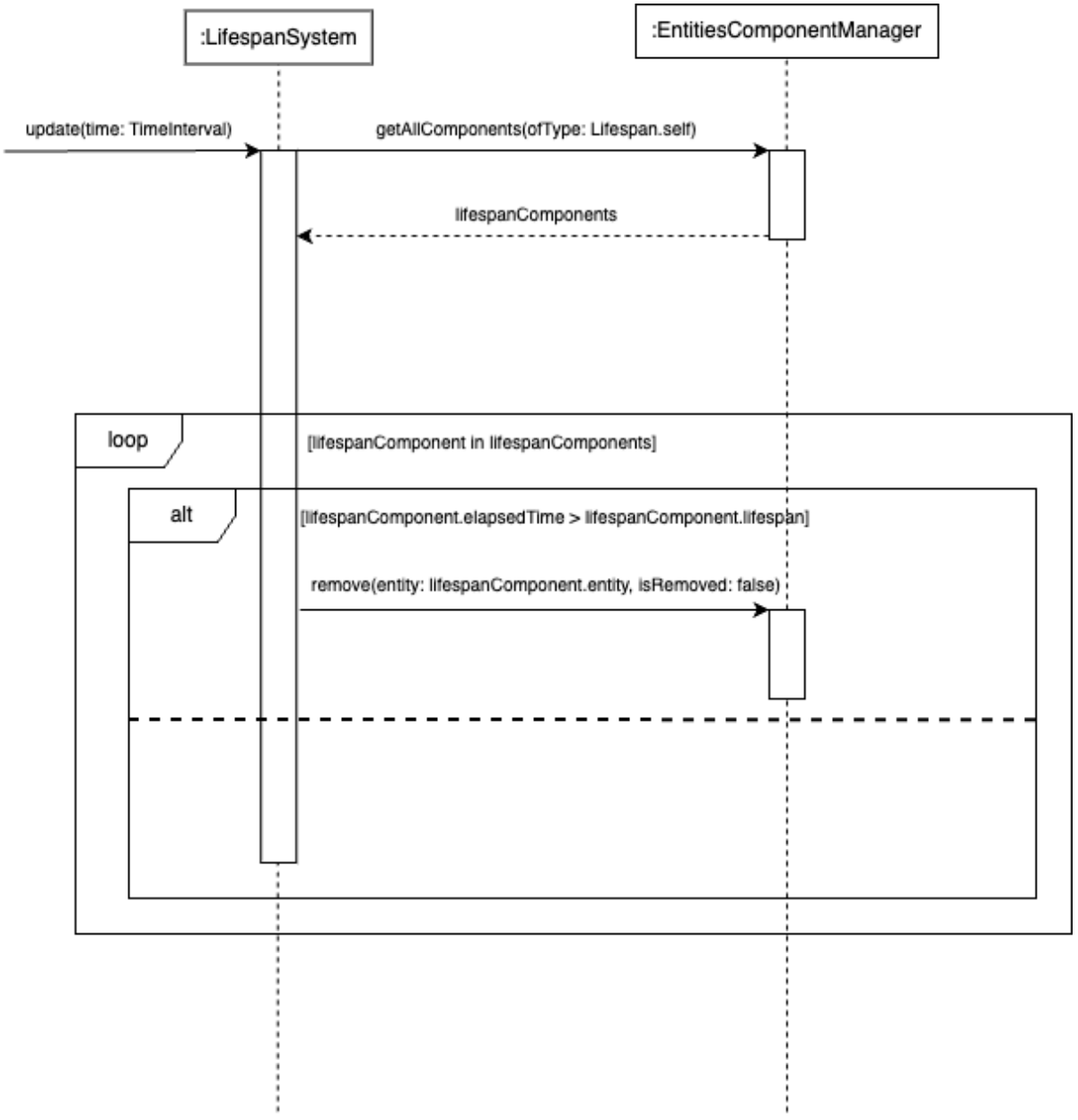
Figure 14: CombatSystem Sequence Diagram



Lifespan System

Certain entities, particularly skill entities, should have a limited lifespan. Otherwise, any player passing by their location will sustain unintended damage, particularly with melee skills. Therefore, the system checks if the elapsed time for the lifespan component has exceeded its lifespan. If so, the entity is removed; otherwise, it remains, with the elapsed time incremented by the time tick. The sequence diagram can be seen below.

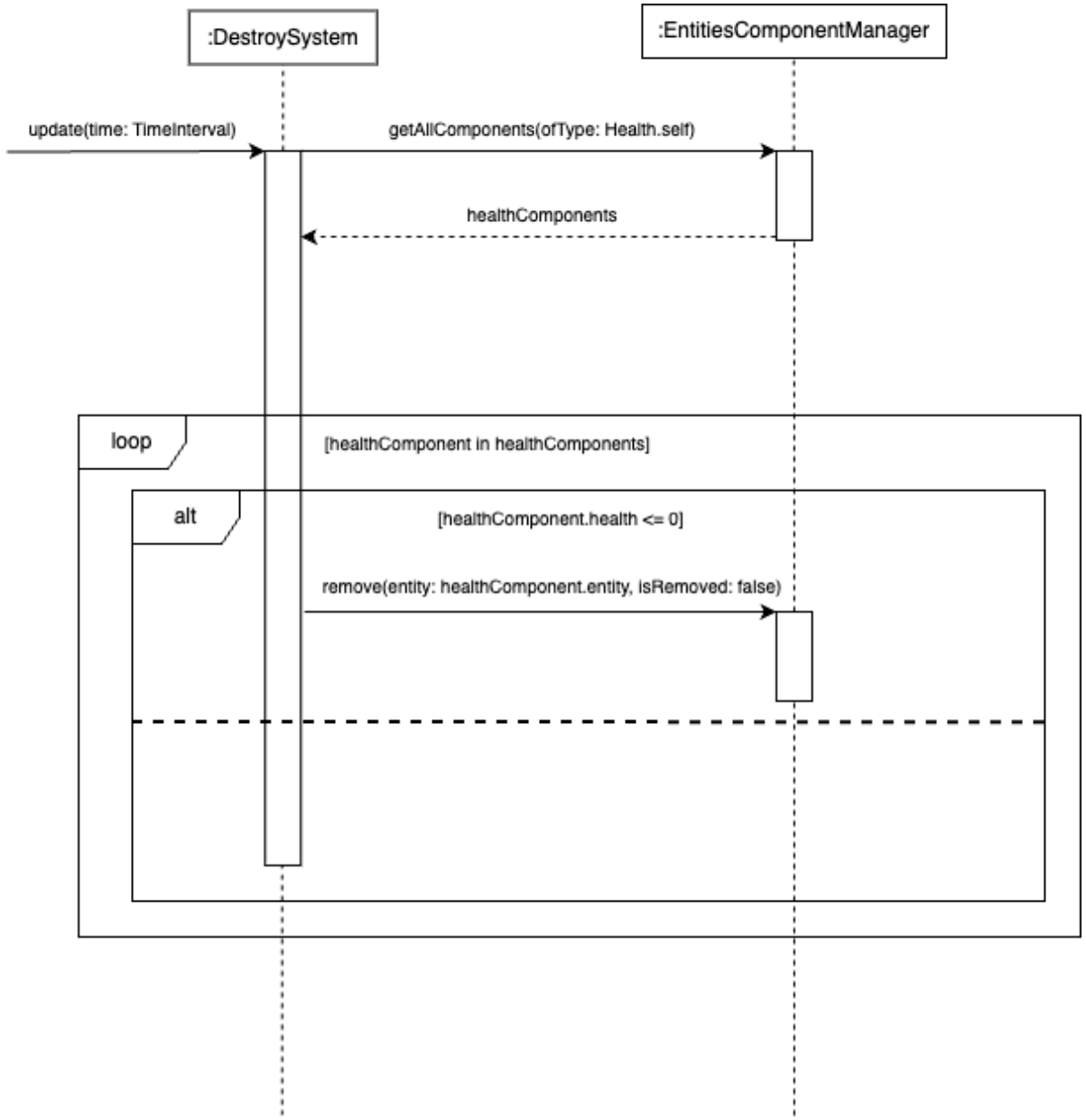
Figure 15: LifespanSystem Sequence Diagram



Destroy System

The destroy system is placed last in every loop in `GameWorld` because we want to remove entities only after all systems that reference the entity to be removed have been processed. Placing the destroy system randomly could prevent certain systems from referencing the destroyed entity, potentially causing undesired behaviour. The destroy system simply checks if any health component has health less than 0 and removes the entity from real-time as well as `EntityComponentManager`. The sequence diagram can be seen below.

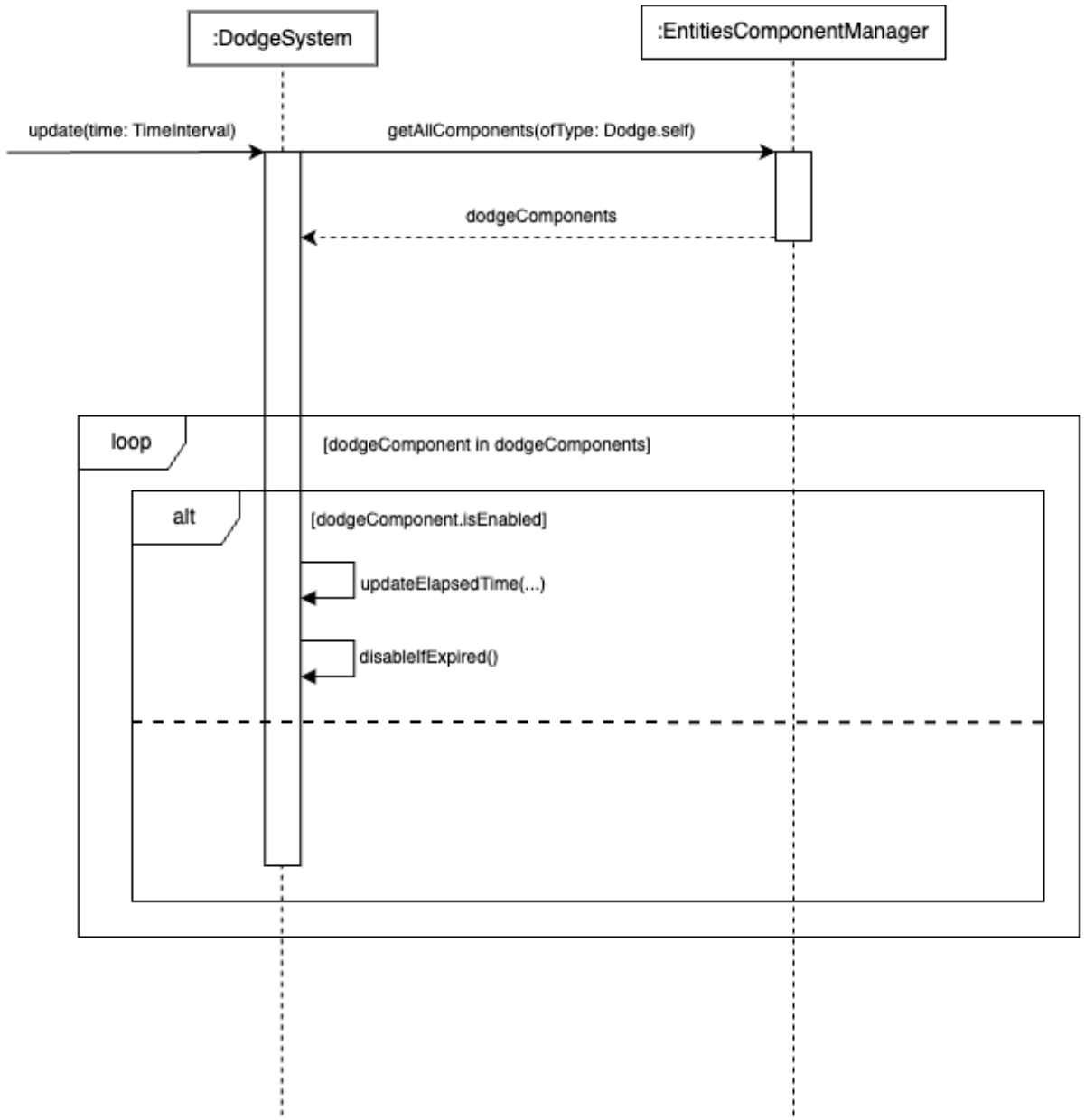
Figure 16: DestroySystem Sequence Diagram



Dodge System

As mentioned in the components section,, the dodge component, when activated, provides the entity with immunity against skills for 2 seconds. Synchronisation is crucial because if an entity activates dodge on one device but not on another, the health of the entity with dodge activated will still decrease on the other device, defeating the purpose of the dodge component. Hence, the dodge system is essential. In the dodge system, it immediately sets the dodge component to not activate only if more than 2 seconds have passed. The sequence diagram can be seen below.

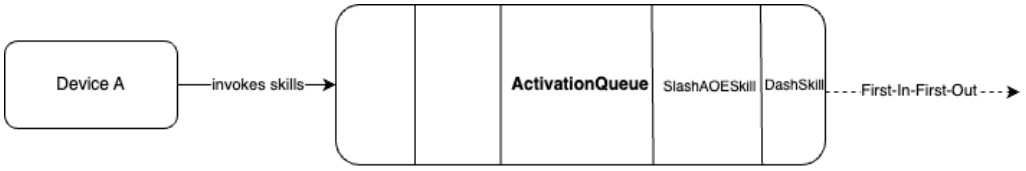
Figure 17: DodgeSystem Sequence Diagram



SkillCaster System

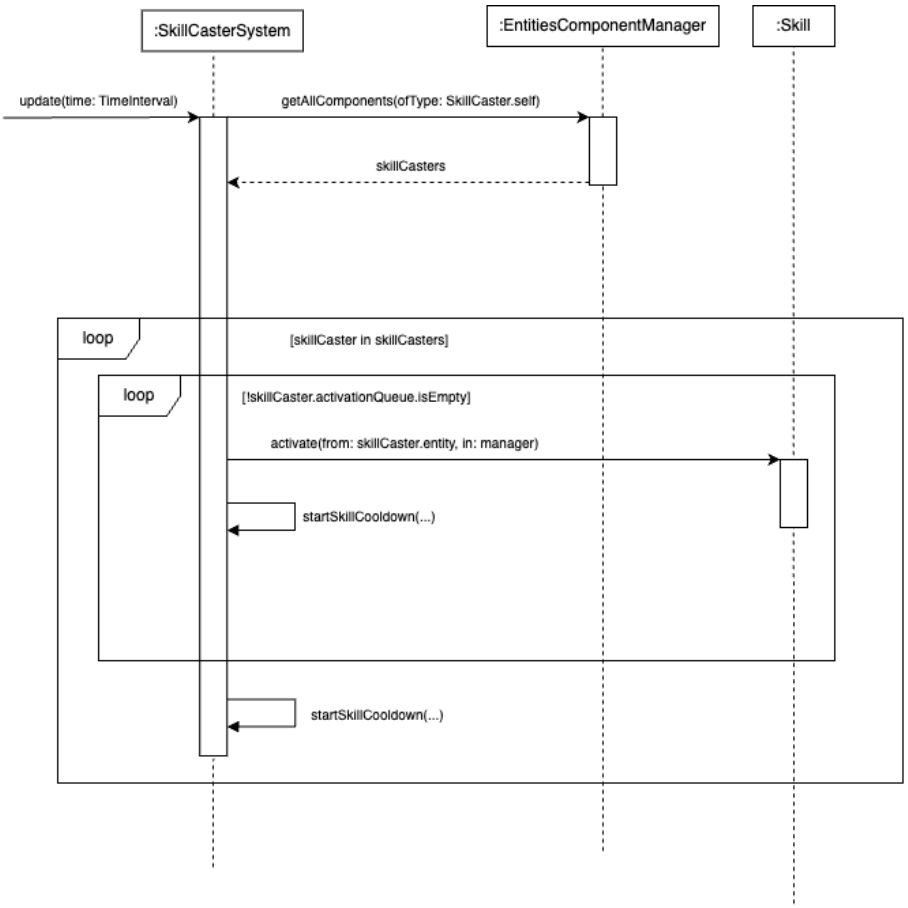
The `SkillCaster` system is arguably one of the more complex systems due to the involvement of a queue. At every tick, it fetches the skillCasters, and for each `SkillCaster`, it checks the activation queue. As long as there are skills in the activation queue, the skills are removed on a First-Come-First-Serve (FIFO) basis. This activation queue serves as a single source of truth for the execution order of the skills, ensuring they are executed in a deterministic manner rather than randomly, and can be seen below.

Figure 18: Skills Activation Queue



Additionally, to prevent skills from becoming overpowered, the `SkillCaster` system immediately initiates a cooldown for the skill that has just been activated after its activation. The overall sequence diagram is shown below.

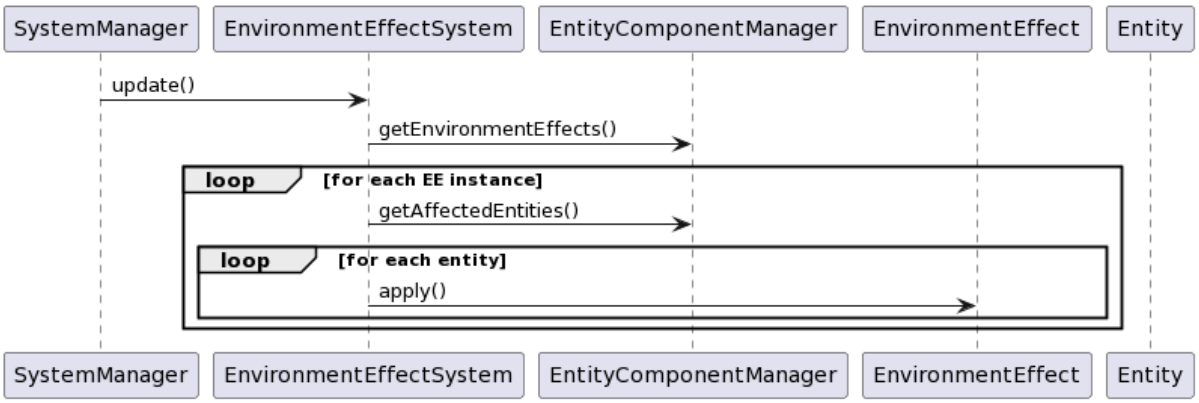
Figure 19: SkillCasterSystem Sequence Diagram



EnvironmentEffect System

The `EnvironmentEffect` system allows us to create zones that can affect certain entities. Specifically, we utilised this feature to implement the `ClosingZone`, which contains an `EnvironmentEffect` component that reduces the health (i.e., decrements the Health) of all entities outside the zone. We designed the system to accommodate any zone shape and dynamically adjust its size, showcasing NinjaWarriors' openness to extension and adherence to good Software Engineering Principles by being closed to modification. The sequence diagram can be seen below.

Figure 20: Environment Effects for Closing Zones

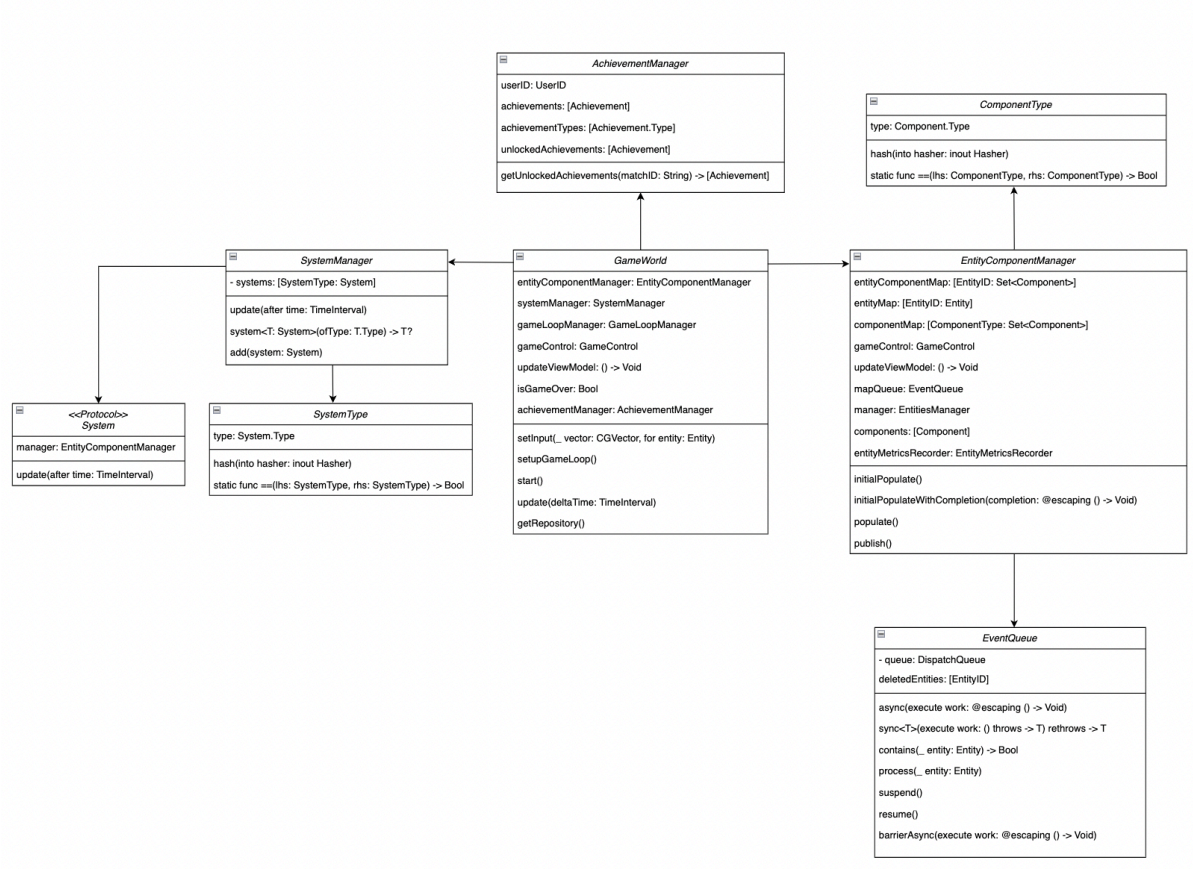


Now that we've discussed all the details of entities, components, and systems, let's explore how the entity-component system is utilised in NinjaWarriors. As mentioned in the Game Update Loop section above, there is a game loop that ensures the application runs at a fixed 60 FPS. The entities and components are stored in the EntityComponentManager, using the data types described in the EntityComponent Manager's dictionary mapping section above.

Once all the relevant entities and components are initialised inside the entity component manager, the entity component manager is then injected into all the relevant systems. These systems process and update all the components. The rationale behind why dependency injection is chosen is discussed in the Singleton vs Dependency Injection section below.

The entire interaction between the game loop, game world, systems, and entity component manager can be observed in the detailed class diagram in Figure 21.

Figure 21: Game Logic Class Diagram



GameWorld (Singleton vs Dependency Injection)

Given that `SystemManager` must contain systems, each of which requires the same reference to an `EntityComponentManager` to update the corresponding components as mentioned above, a question arises regarding whether a **singleton** or **dependency injection** should be used to have the same reference to `EntityComponentManager`.

Figure 22 illustrates the UML class diagram when **singleton pattern** is used, while Figure 23 shows the UML class diagram when **dependency injection** is used.

Figure 22: Singleton

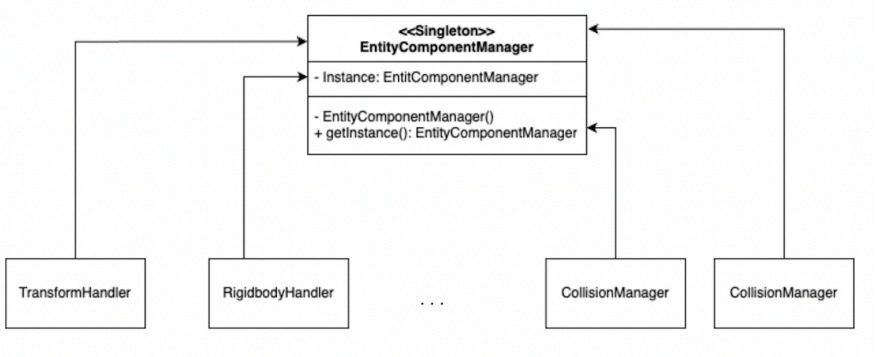
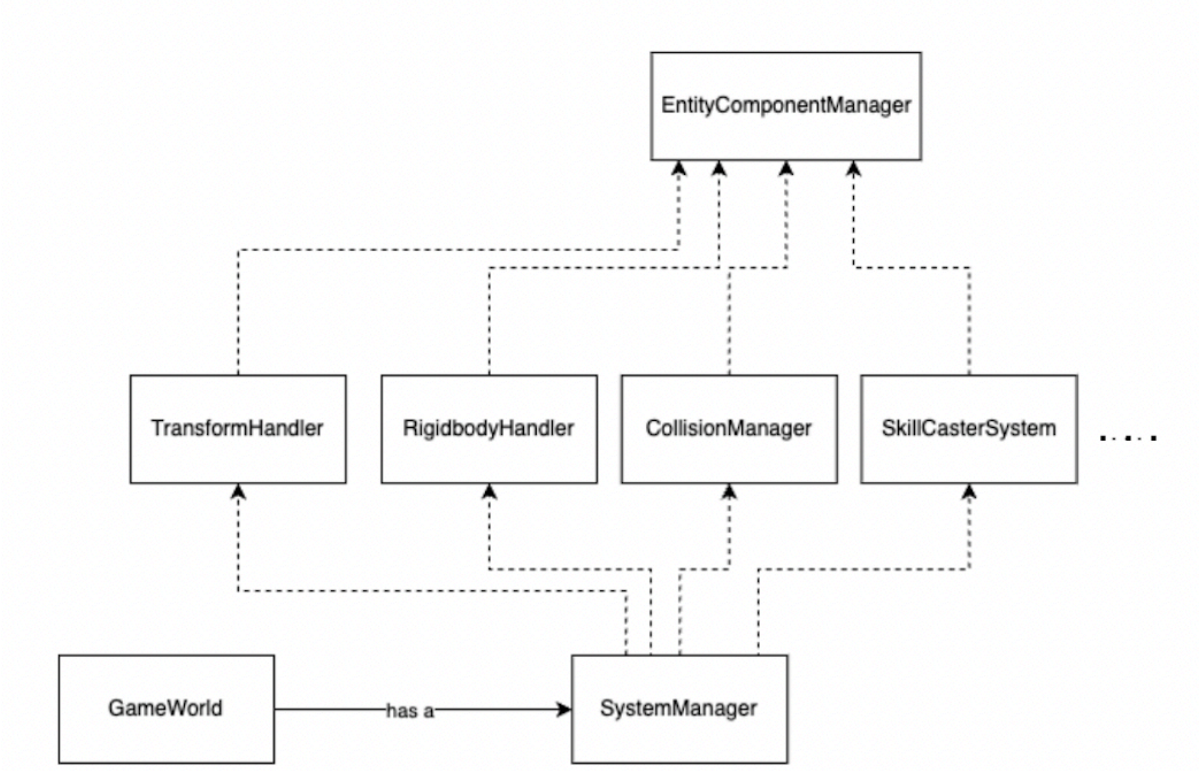


Figure 23: Dependency Injection



Following **Martin Fowler's implementation** where a dashed arrow from A to B means B is injected into A. Hence in this case, EntityComponentManager is injected into the relevant systems.

Reason for choosing dependency Injection

In this case, we decided to opt for dependency injection because a singleton pattern would tightly couple the application. Additionally, designating one class as a singleton could potentially lead to a slippery slope of making other classes singletons to avoid the need for maintaining and injecting relevant classes or structs. Therefore, from a good software engineering principle standpoint, dependency injection was chosen to ensure loose coupling within the application.

Here is a snippet showing how dependency injection is implemented (details omitted):

```
class GameWorld {
    let entityComponentManager = EntityComponentManager()
    init() {
        let collisionManager = CollisionManager(for: entityComponentManager)
        let rigidbodyHandler = RigidbodyHandler(for: entityComponentManager)
        let skillsManager =  SkillCasterSystem(for: entityComponentManager)
        …
        setupGameLoop()
    }
}
```

With the entire Entity, Component, and System explained above, let's move on to other design patterns. We utilised various design patterns, namely the Observer pattern, Adapter pattern, Strategy Pattern, Singleton pattern, and inheritance in the case of Achievements. The rationale for using these patterns is to make NinjaWarriors as extensible as possible.

Adapter Pattern

Since NinjaWarriors is an online game that requires players to relay information to one another, a multiplayer infrastructure is necessary. In our case, we chose to use Firebase for Authentication, Matchmaking, and the gameplay itself. However, directly importing the module into the classes involved in Authentication, Matchmaking, and gameplay would lead to strong coupling with a third party, Firebase in our case. Hence, the adapter pattern was utilised for all three aspects.

For multiplayer, matchmaking utilises the **adapter pattern**. The target (`MatchManager`), adapter (`MatchManagerAdapter`), and adaptee (`FirestoreCollectionReference`) are categorised under the same Matches folder. However, the client, which is the `LobbyViewModel`, is not categorised in the same folder; instead, it is classified under the ViewModel folder. This separation allows for easier reference in the future, as it is more closely coupled with the LobbyView.

Similarly, for sign in, the **adapter pattern** is also used. The target (`Authentication`), adapter (`AuthenticationAdapter`), and adaptee (`FirebaseAuth`) is also classified under the same Authentication folder. Likewise, the client, which is the `SignInViewModel`, is not categorised in the same folder; instead, it is classified under the ViewModel folder. This separation allows for easier reference in the future, as it is more closely coupled with the SignInView.

Adapter Pattern Rationale

Suppose there arises a need to hot swap the authentication framework, matchmaking framework, or in-game multiplayer framework in the future, transitioning, for instance, from `Firebase` to `CoreData` or `Mongo`. By simply modifying the adapter's call to the adaptee (the framework), no changes are required on the client or the target. This approach upholds decoupling and allows the application to be extensible, adhering to the principles of being open to extension and closed to modification, in accordance with software engineering (SWE) principles.

The adapter pattern used for matchmaking can be seen in Figure 24 below, and the adapter pattern used for sign in can be seen in Figure 25. Document storage uses an adapter pattern as seen in Figure 26. Realtime also employs the adapter pattern, similar to matchmaking, as seen in Figure 27.

Figure 24: Adapter pattern for matchmaking

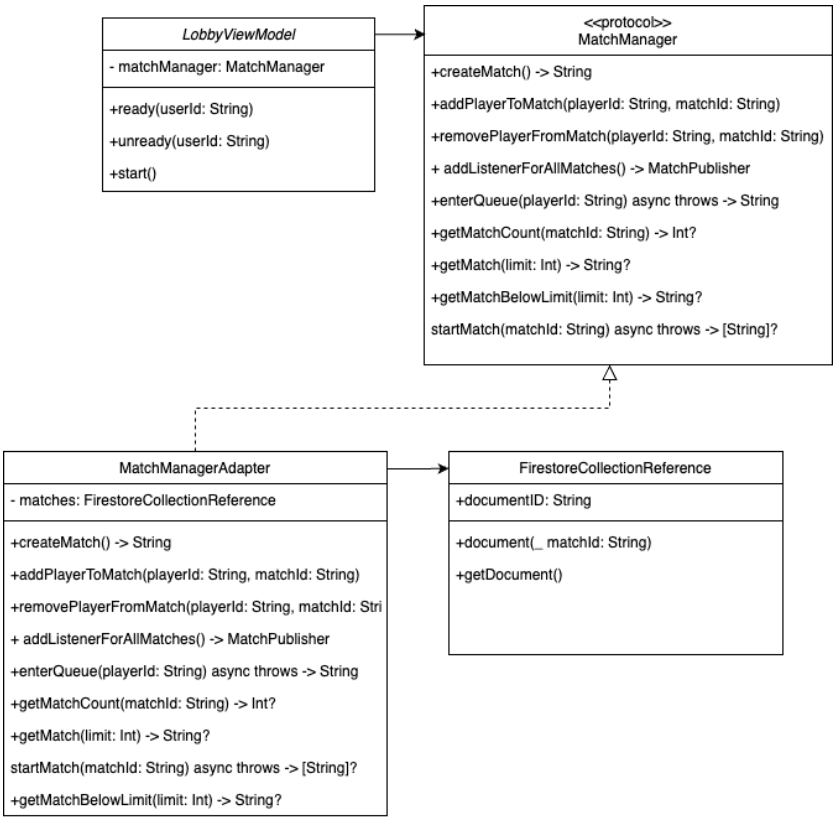


Figure 25: Adapter pattern for sign in

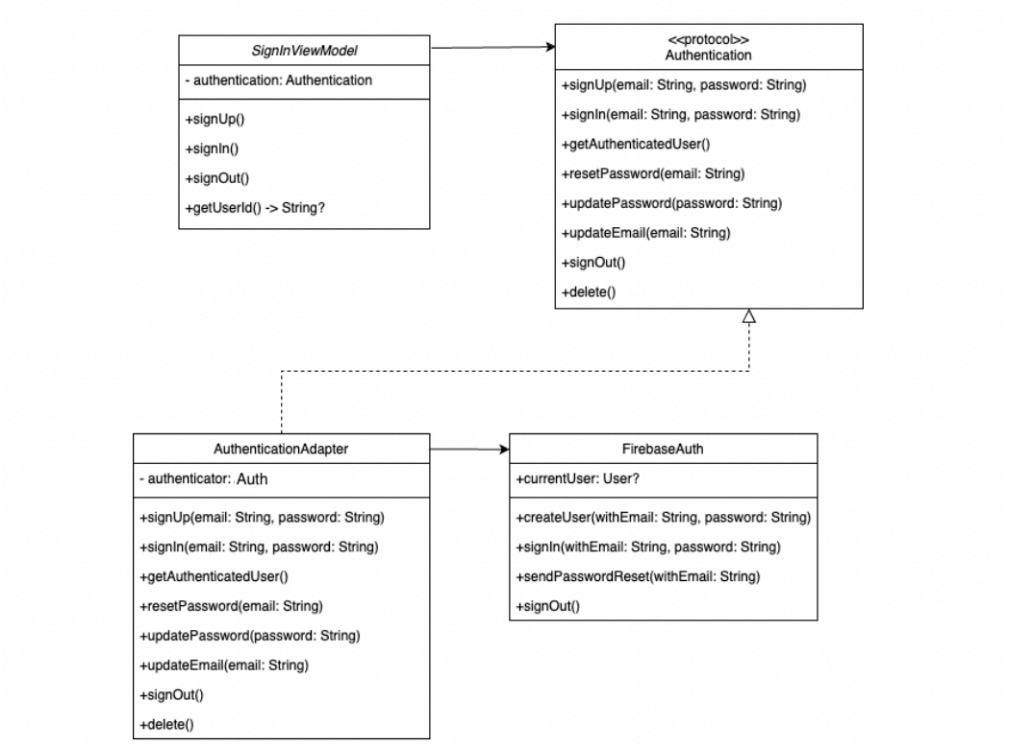


Figure 26: Adapter pattern for document storage

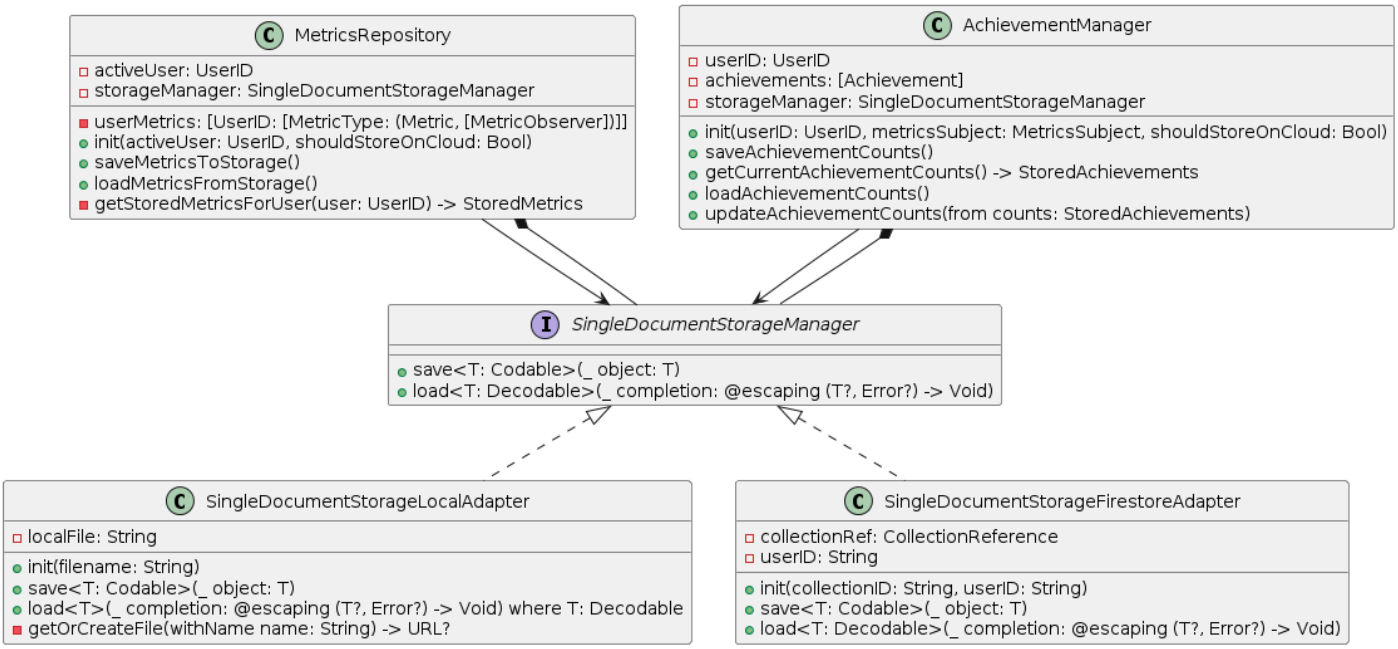
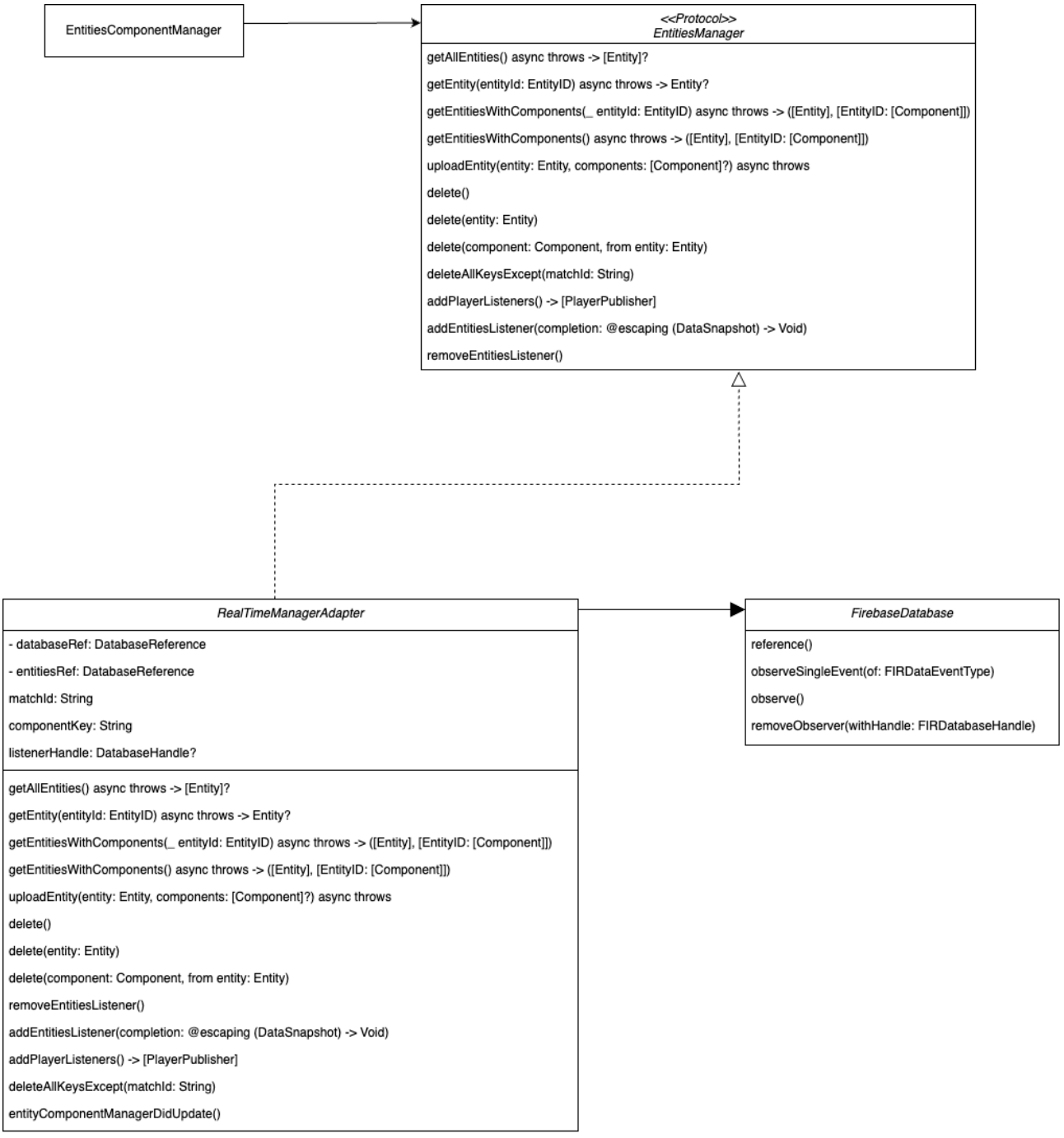


Figure 27: Realtime Adapter Pattern

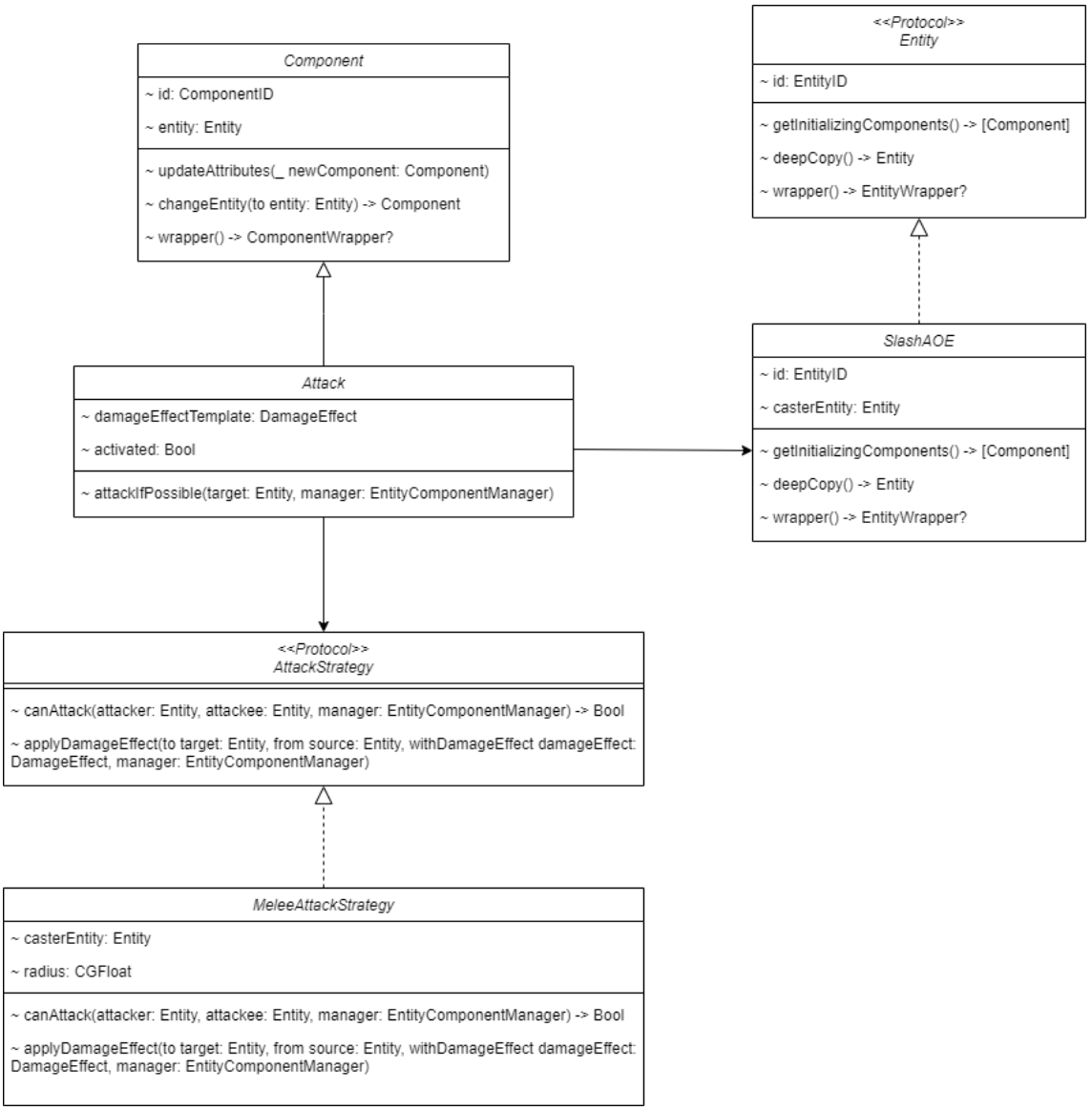


Strategy Pattern

Attack Strategy Pattern

The `Attack` component is responsible for dealing damage from skills. When a skill that can damage is activated, an `Attack` component is created, and attached either onto an entity created by the skill (i.e. `SlashAOE` entity representing a slash effect) or onto the entity who casted the skill. The `Attack` component contains an `AttackStrategy`, which has two methods, `canAttack()` which determines which entities can be affected by the attack, and `applyDamageEffect()` which applies the damage effect to the target. Different attack strategies can be created, each with their own logic of which entities can be affected (i.e. every player on the map, players within a certain radius of the caster, players within a certain radius of the entity created by the skill, such as a bullet), as well as how damage is dealt to these entities (i.e. damage over time, or a fixed amount of damage, etc.). By employing a strategy pattern, we make attacking skills extensible to all sorts of future implementations.

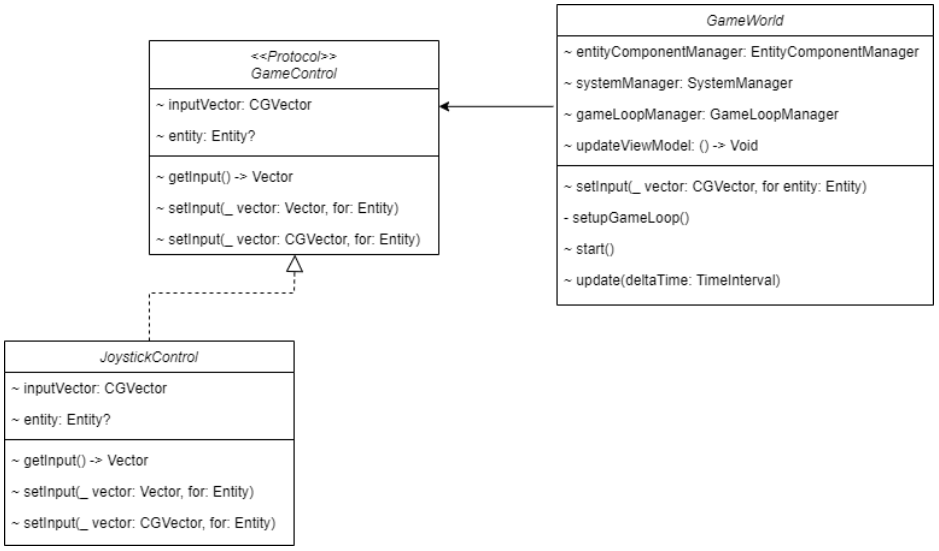
Figure 28: Attack Strategy Pattern



Game Control Strategy Pattern

Currently, the player's in-game movement is controlled by a joystick. The joystick has its own `JoystickView` and `JoystickControl`, which conforms to `GameControl` protocol. We employed the **strategy pattern** for game controls. This means that should there be alternative methods of controlling the game, these methods should extend the `GameControl` and be passed into the `GameControl` in `GameWorld`. This allows for future development of various controls for this game, including keyboard, controller, etc. The strategy pattern can be seen in Figure 29 below.

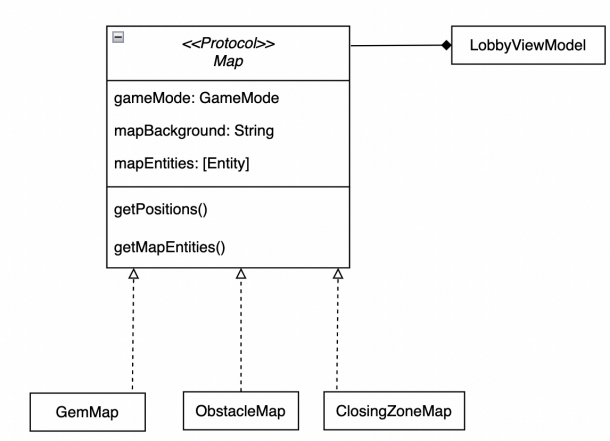
Figure 29: Game Control Strategy Pattern



Map Strategy Pattern

Currently, `SingleLobbyViewModel` and `LobbyViewModel` have a variable of type `Map`. In order to allow maps to be hot swapped and also abide by open-closed principle, which posits that a software should be open to extension and closed to modification, concrete map implementations extend from `Map`. This **strategy pattern** allows alternative maps to be added in the future as well. The class diagram can be seen below.

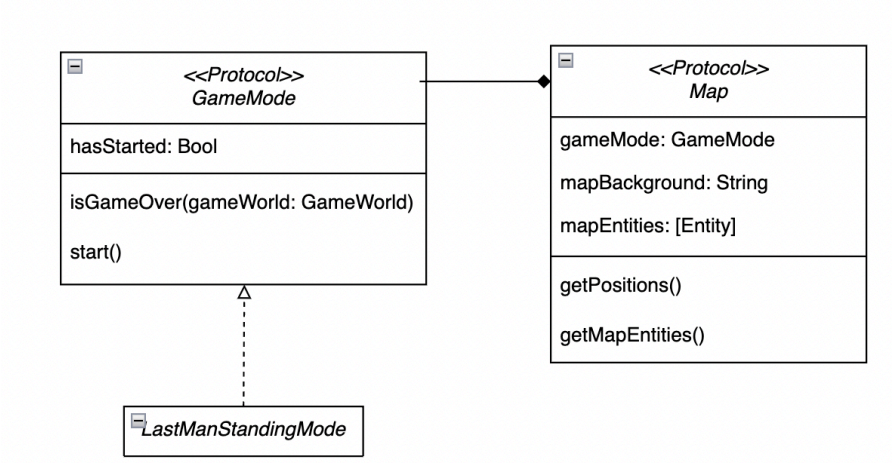
Figure 30: Map Strategy Pattern



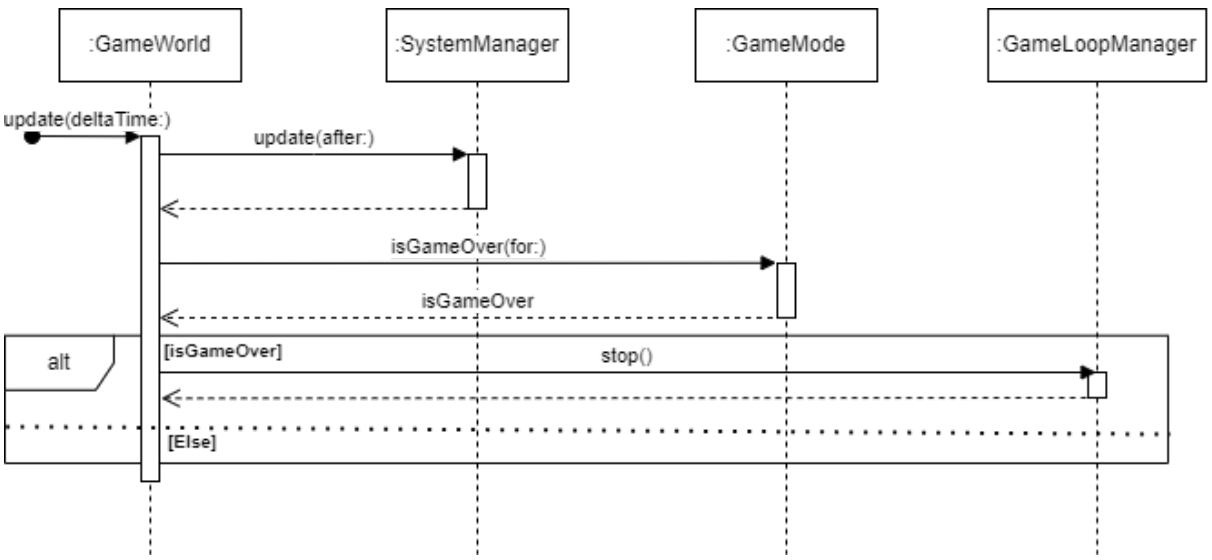
GameMode Strategy Pattern

Currently, `Map` depends on `GameMode`. We also adopted the **strategy pattern** for this to decouple map and game modes. By decoupling map and game modes, we allow for partition-ability as different team members can work on concrete implementations of map and game modes without needing to communicate constantly. In addition, we allow for extensibility, because should other game modes need to be implemented, it can be passed into the concrete implementation of `Map` without modifying `Map`. This allows for possible reuse of maps for different game modes in the future.

Figure 31: Gamemode Strategy Pattern



By implementing the strategy pattern for game mode, we also allow for various game over logic to be implemented based on the game mode. For instance, LastManStandingMode can implement isGameOver(for:) by checking for 1 player remaining, while other game modes like GemCollectionMode can implement isGameOver(for:) by checking for the number of gems collected by each player. We allow for extensibility in game over logic to stop the game loop without needing to change the structure of our app for each game mode.



Skill Strategy Pattern

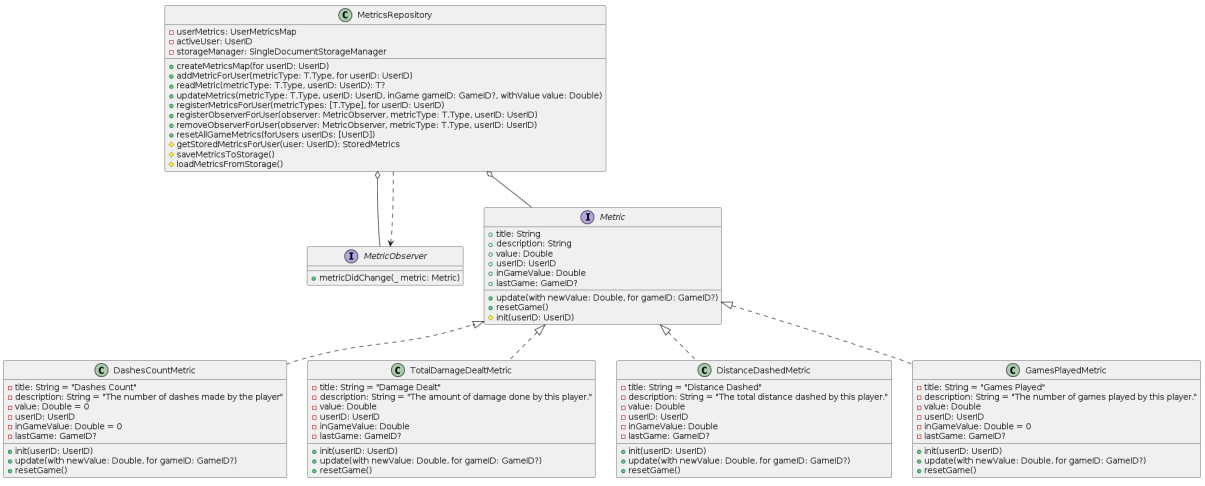
```
func update(after time: TimeInterval) {
    guard let manager = manager else { return }

    let skillCasters = manager.getAllComponents(ofType: SkillCaster.self)
    for skillCaster in skillCasters {
        while !skillCaster.activationQueue.isEmpty {
            let skillId = skillCaster.activationQueue.removeFirst()
            guard let skill = skillCaster.skills[skillId], !skill.isOnCooldown() else { continue }

            skill.activate(from: skillCaster.entity, in: manager)

        }

        skillCaster.decrementAllCooldowns(deltaTime: time)
    }
}
```

Within SkillCasterSystem, which is the System that handles the update loops, it handles the cooldowns of all skills and decrements them in each update frame, and executes the skills. It uses the **Strategy Pattern**, shown in the way different skills implement the Skill protocol to define their unique activate behaviour. Each Skill subclass (DashSkill, SlashAOESkill, etc.) provides a different algorithm or strategy for its activate() method, which is called in SkillCasterSystem::update(). This allows the algorithm's behaviour to vary independently from the clients that use it, in this case, the SkillCasterSystem that activates the skills. The sole responsibility of the SkillCasterSystem is to ensure that skills are activated in order, and activated only if they are allowed to do so. The behaviours of the skills are defined within the skill itself.

Observer Pattern

The observer pattern was implemented for Achievements. `MetricRepository` contains the mapping of `UserID` to `MetricsAndObserverMap`. `MetricsAndObserverMap` contains a mapping of `MetricType` to the corresponding `Metric` and the array of `MetricObserver`.

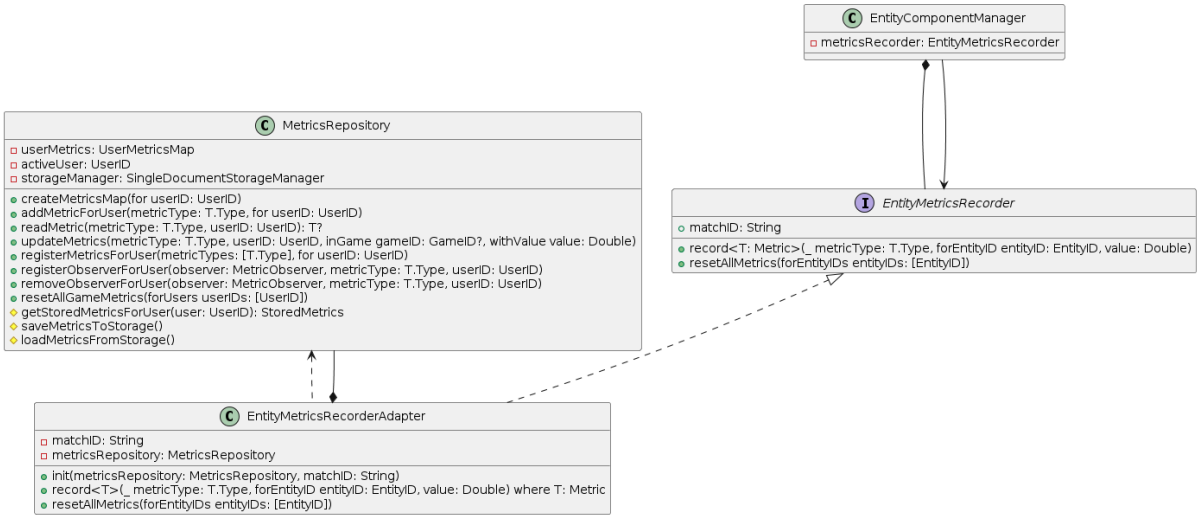
Figure 32: MetricsRepository and some metrics that it contains



`MetricRepository` is injected into `GameWorld`, which then injects it into `EntityComponentManager` as an implementation of the `EntityMetricsRecorder`

protocol via an adapter. Then, the metrics are updated in the update loop by various relevant systems.

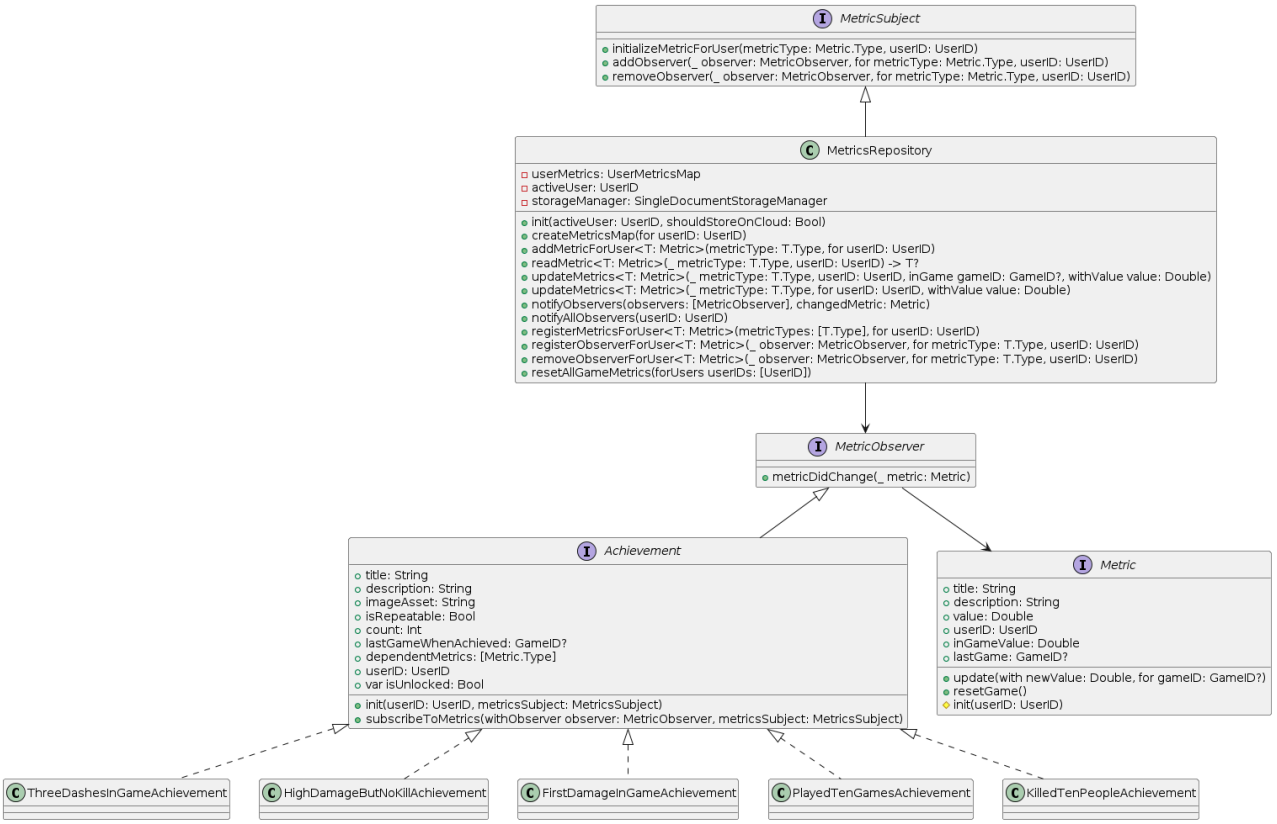
Figure 33: How the ECS systems update metrics



Now, the concrete classes that implement each `Achievement` protocol also implement `MetricObserver`. This means that when there is an update in the metrics, the achievements will be notified. Each metric has an inGameValue as well as a cumulative value, as well as an optional field for last match ID. Encapsulating all these different metrics lets achievement logic potentially compute many different combinations of metrics to check against achievement conditions. Potentially, things other than achievements may also be implemented to observe metrics, promoting extensibility.

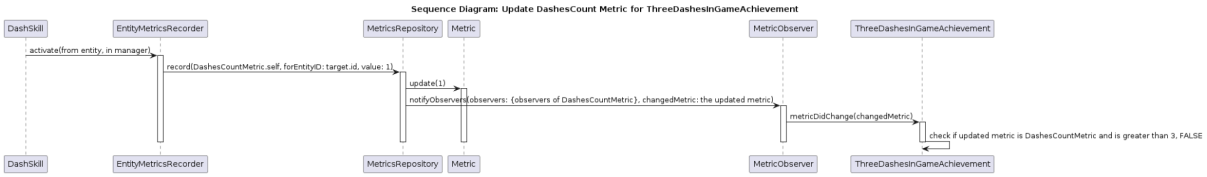
By having achievements only observe metrics, we have successfully developed NinjaWarriors achievements to be as extensible as possible.

Figure 34: Adding observers onto metrics for achievements



If we want to add new achievements, we don't need to change or modify the metrics; instead, we can simply add more concrete achievements and observe the necessary metrics. Similarly, developers can add new metrics to be measured and easily Each concrete `Achievement` can define its own logic for checking whether an achievement has been earned. This means that fairly complicated conditions for achievements can be implemented, as seen in examples like `HighDamageNoKillAchievement.` Achievement objects can maintain their own state to assist in computing when an achievement has happened. Let's see how this works out with a sequence diagram for the dash achievement.

Figure 35: How a metrics update bubbles from ECS to achievements



Lastly, achievements and metrics both need to be persisted. For single-player mode and multiplayer guest mode, we save the metrics and achievements locally but separately. For multiplayer with sign-in, we save both on the cloud. We considered two options:

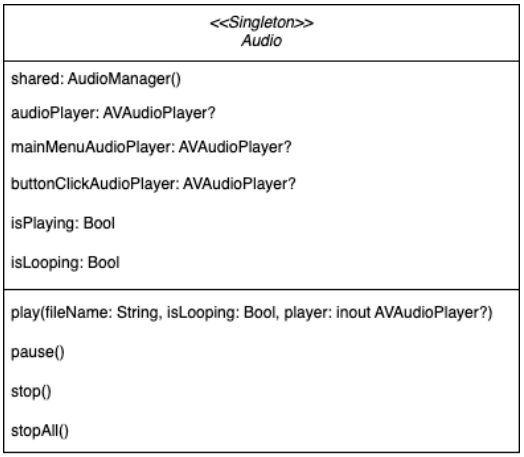
1. Sync all metrics and achievements continuously on the cloud
2. Save metrics and achievements at defined times

We opted for 2 since we couldn't see any value in option 1, especially given the increased network bandwidth that would cause. Another insight was that the information that we want to persist for metrics and for achievements is very minimal: for a user, we only care about the value of each metric (since that is cumulative and needs to be preserved across games) and the count of each achievement. Other details like title, description etc can be reconstructed in-memory. Given this need for simple information storage, we use JSON/NoSQL approaches to persist metrics and achievements. The diagram can be seen in Figure 15 in our Adapter Pattern section.

Audio (Singleton)

Lastly, to improve the user's game experience, we included audio, which is managed by `AudioManager` singleton. We would want a single instance so that multiple audios will not be playing over each other. The class diagram for the audio manager can be seen below.

Figure 36: Audio Singleton Pattern



3: Testing

3.1: Strategy

The development of Ninja Warriors, a real-time multiplayer game, demanded a rigorous and methodical testing strategy to ensure the software met its functional requirements and performance benchmarks effectively. Given the complexity and interactive nature of the game, combining manual testing with targeted automated tests provided a comprehensive evaluation of the system's reliability and usability. Our approach focused primarily on manual testing complemented by selective automated testing for critical backend components.

Selection of Testing Techniques

Manual Testing

Due to the game's graphical and interactive elements, manual testing was extensive, and was conducted by all group members frequently. It involved:

Gameplay Mechanics

Testing player movements, skill activations, and interactions manually to ensure they behaved as expected according to game rules.

Multiplayer Interactions

Playing the game in a multiplayer setting to verify real-time interactions and data synchronisation across different clients.

User Interface

Ensuring that the game's UI is intuitive and responsive, providing a seamless experience across various devices and screen sizes.

Automated Testing

Automated tests were written for the foundational backend logic, particularly where precise and repeatable testing was feasible:

RealTimeManagerAdapter

To ensure robustness in handling real-time multiplayer states.

Authentication Processes

Verifying the reliability and security of login and registration functionalities.

Entity Component System (ECS)

Focused tests to ensure that game logic components interacted correctly.

Achievement Integration

Testing the correct awarding of achievements based on player actions and game outcomes.

Primitive Data Handling

Including automated tests for custom data types like points, vectors, and shapes, particularly to validate collision detection logic.

Local Document Storage Integration

Automated integration tests to check whether storing documents (which is used for metrics and achievements) works successfully with metrics, achievements, as well as arbitrary codable objects

Integration Testing

While comprehensive integration testing was limited, crucial interactions between the frontend UI and the backend logic were manually tested to confirm data consistency and functional behaviour under typical usage scenarios.

Expected Classes of Errors

The primary classes of errors anticipated included:

- Functional Errors: Potential mistakes in the logic for skill effects, player movement, and health updates.
- Synchronisation Errors: Given the game's real-time nature, issues could arise in keeping player states consistent across devices at an industry-standard level.
- UI Inconsistencies: Errors in rendering the UI correctly on outdated iOS devices, affecting gameplay fluency and user interaction.

Errors less likely to be detected through our testing approach were:

- Edge-Cases in Multiplayer Concurrency: Rare timing issues caused by unusual sequences of actions taken by multiple players simultaneously.
- Complete Network Failure Scenarios: While basic connectivity issues were tested, more extreme network conditions (like intermittent connectivity) were not rigorously evaluated.

Aspects Influencing Testability

Facilitators

Modular Architecture: The use of an ECS architecture facilitated isolated testing of game logic components, making unit tests more straightforward and effective.
Decoupled Components: The separation between the game's logic (handled by ECS) and the presentation layer allowed independent testing of game mechanics without UI dependencies.

Barriers

Complex Interaction Patterns: The dynamic and interactive nature of the game made it challenging to automate testing fully, particularly for features involving multiple players.
Limited Automation: The reliance on manual testing for the UI and integration points meant that some less apparent issues could go undetected during the initial testing phases.

3.2: Test Results

Module Testing Overview

Authentication and Real-Time Management

Automated tests for these modules showed that the login processes and real-time state management were reliable, with a focus on handling typical and atypical inputs.

ECS and Achievements

The ECS provided a robust framework for managing game objects and interactions, which performed well in tests involving component logic and integration. Achievement system tests confirmed that player accomplishments were tracked and rewarded correctly.

Primitive Data Operations

Tests on custom data types ensured that foundational mathematical operations, essential for game mechanics like movement and collision detection, were accurate and efficient.

Confidence and Remaining Faults

Confidence Level

The manual testing approach, combined with targeted automated tests, has given us strong confidence in the system's functionality under normal operating conditions. The game performs well in standard multiplayer scenarios and exhibits the correct behaviour in response to user inputs and interactions.

Potential Remaining Faults

Intermittent Bugs and Edge Cases

Due to the nature of manual testing, some rare conditions or unusual sequences of actions may not have been thoroughly tested.

Performance Under Stress

High player loads and extreme network conditions were not fully tested, which may impact gameplay fluidity and synchronisation in untested scenarios.

In conclusion, while the current testing regime has provided good coverage of typical game functionalities and critical system components, ongoing testing, especially in live environments and under stress conditions, would help uncover less common issues and ensure the game's reliability and performance as it scales to accommodate more players. The flexibility of the testing strategy, particularly the integration of more automated tests, would enhance coverage and confidence as Ninja Warriors evolves.

4: Reflection

4.1: Evaluation

Having not previously implemented an Entity-Component-System (ECS), learning and implementing it simultaneously posed challenges, especially considering the existence of multiple ECS versions. While we didn't fully adopt Unity's approach, we drew inspiration from it. We chose not to utilise the memory management techniques employed by Unity.

Encoding and decoding multiplayer data into and from Firebase presented its own set of challenges. We opted to use wrappers instead of making every struct and entity conform to Codable.

In Sprint 1, we successfully implemented nearly all of our core features, including authentication, multiplayer, ECS, and important components such as collider system, and foundation skill system. However, we did not manage to implement the logic for damage, health, and collision.

In sprint 2, we managed to successfully add damage, skill details, and nuances, as we've designed the application to be highly extensible.

Although it was tempting to expedite implementation using third-party libraries like pub-sub, we aimed to minimise dependency on such libraries beyond Firebase, SwiftUI, and Foundation.

Here are the overall features we aimed to achieve at the beginning of sprint 1, along with the status of whether we accomplished them by the end of sprint 3.

Features	Status
Single player mode	✅
Multiplayer guest mode (multiplayer without logging in)	✅
Account creation and login	✅
Game lobby and matchmaking	✅
Player movement with a joystick game controller	✅
Player skills	✅
Closing zone mode	✅

Obstacle mode	✅
Gem collection mode	✅
Collision handling	✅
Attacks	✅
Skills cooldown	✅
Four customised skills	✅
Achievements	✅
Game Over	✅

What we initially set out to achieve was all accomplished by the end of sprint 3. Initially, we built it such that it was only for the player's position. However, as the sprints progressed, we built it such that it is more extensible to any entities and any components such that adding any entities or adding any components on the fly will still work.

Overall, this experience has been educational and we learnt a lot about team collaboration.

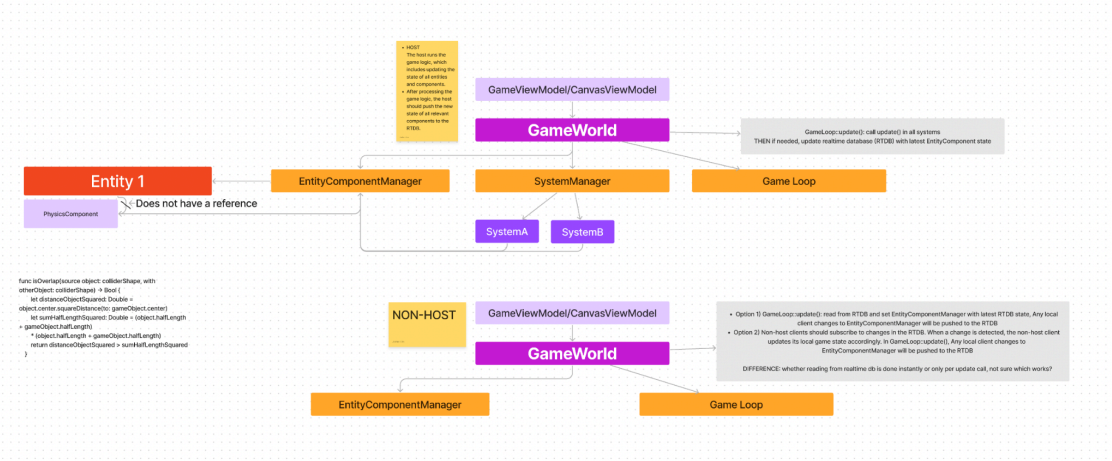
4.2: Lessons

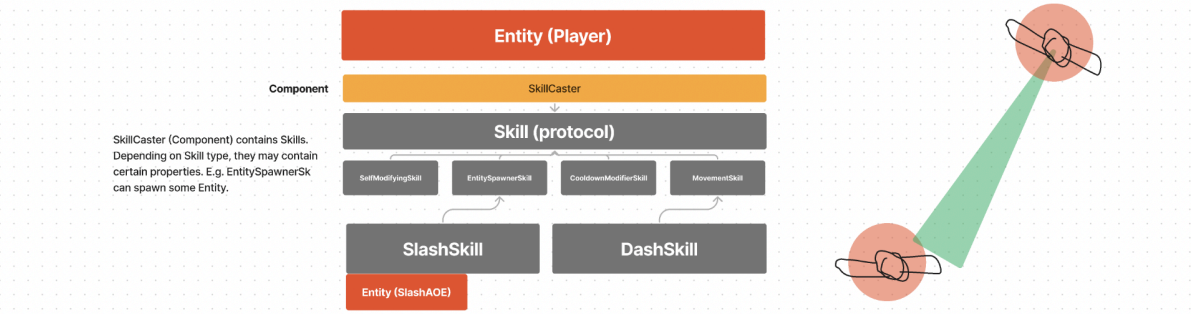
Lessons

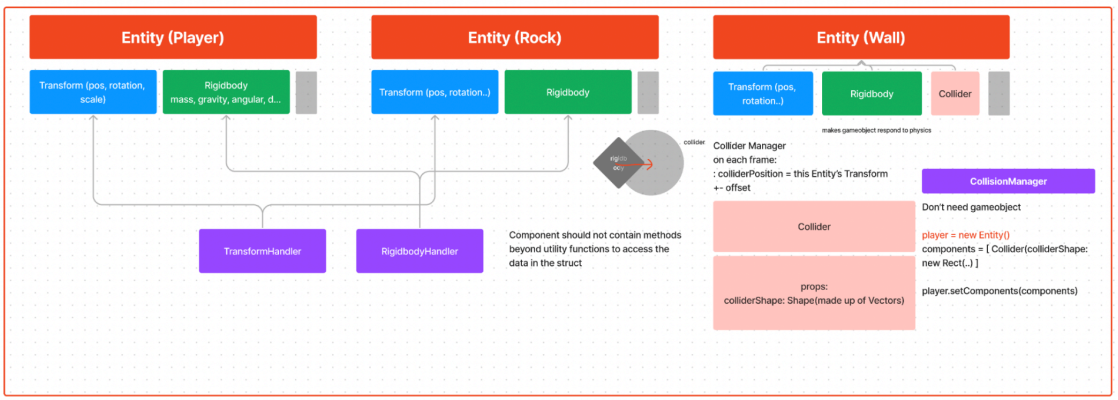
We have learnt that working as a team to build a full stack application within a short period of time and abiding by software engineering principles is not easy. In the past, it was common to just implement hackish solutions to problems but this time, the software itself is heavily graded so we must properly implement them.

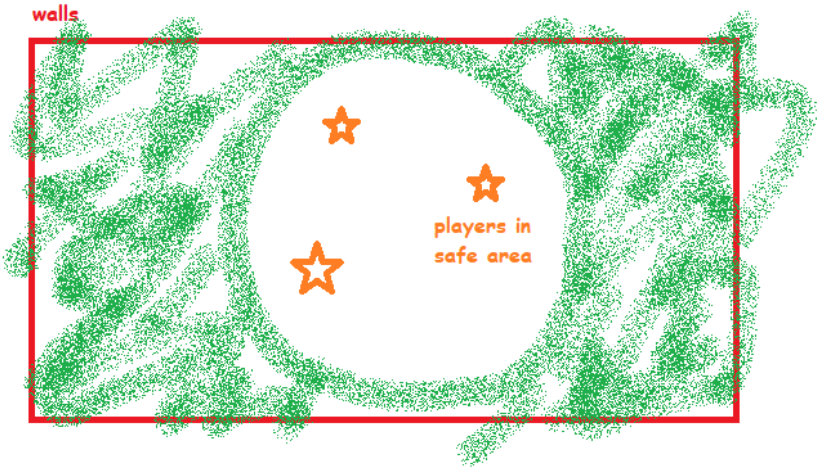
We have also learnt to utilise the strengths of teammates. Some of us are better at abstraction, while others are better at solving problems quickly. Hence, by combining these two, we are able to rectify problems in the correct manner.

Lastly, we have learnt that having regular calls and constant delegation of tasks within the teammates is important to abide by the Agile principle. Attached below are a few diagrams from our calls that we made in Figma.


• HOST
The host runs the game logic, which includes updating the state of all entities and components.
• After processing the game logic, the host should push the new state of all relevant components to the RTDB.
GameViewModel/CanvasViewModel
GameWorld
GameLoop.update(): call update() in all systems
THEN if needed, update realtime database (RTDB) with latest EntityComponent state
Entity 1
EntityComponentManager
SystemManager
Game Loop
PhysicsComponent
Does not have a reference
SystemA
SystemB
func isOverlap(source object: colliderShape, with otherObject: colliderShape) -> Bool {
let distanceObjectSquared: Double = object.center.squareDistance(to: gameObject.center)
let sumHalfLengthSquared: Double = (object.halfLength + gameObject.halfLength)
* (object.halfLength + gameObject.halfLength)
return distanceObjectSquared > sumHalfLengthSquared
}
NON-HOST
GameViewModel/CanvasViewModel
GameWorld
• Option 1) GameLoop.update(): read from RTDB and set EntityComponentManager with latest RTDB state, Any local client changes to EntityComponentManager will be pushed to the RTDB
• Option 2) Non-host clients should subscribe to changes in the RTDB. When a change is detected, the non-host client updates its local game state accordingly. In GameLoop.update(), Any local client changes to EntityComponentManager will be pushed to the RTDB
DIFFERENCE: whether reading from realtime db is done instantly or only per update call, not sure which works?
EntityComponentManager
Game Loop


Entity (Player)
Component
SkillCaster
Skill (protocol)
SkillCaster (Component) contains Skills. Depending on Skill type, they may contain certain properties. E.g. EntitySpawnerSk can spawn some Entity.
SelfModifyingSkill
EntitySpawnerSkill
CooldownModifierSkill
MovementSkill
SlashSkill
DashSkill
Entity (SlashAOE)


Entity (Player)
Transform (pos, rotation, scale)
Rigidbody mass, gravity, angular, d...
Entity (Rock)
Transform (pos, rotation..)
Rigidbody
Entity (Wall)
Transform (pos, rotation..)
Rigidbody
Collider
makes gameobject respond to physics
collider
Collider Manager
on each frame:
: colliderPosition = this Entity's Transform
+- offset
CollisionManager
Don't need gameobject
player = new Entity()
components = [Collider(colliderShape: new Rect(..)]
player.setComponents(components)
Collider
props:
colliderShape: Shape(made up of Vectors)
TransformHandler
RigidbodyHandler
Component should not contain methods beyond utility functions to access the data in the struct



Known bugs and limitations

Since we are using SwiftUI, the navigation stack is prone to be deprecated, which means our SwiftUI would have to use something else to navigate.

Additionally, we are using the free firebase version, which means we cannot have unlimited storage, reads, and writes when the application scales. However, this is not an issue since we are focused on building good software rather than making the application scale globally.

We are also in the midst of ensuring there are no race conditions, as well as unowned references.

Finally, we need to ensure the client does not lag as much, and the goal is for the client's gameplay to be as smooth as the host's gameplay.

5: Appendix

5.1: Test Cases

Integration Tests Plan

How to Play:

- Entering the how to play page should display an animated joystick moving left and right at the top left of the page
- Entering the how to play page should display a closing zone animation that is on loop at the top right of the page
- Entering the how to play page should display blinking gems at the bottom left of the page
- Entering the how to play page should display skill buttons at the bottom right of the page
- Pressing the back button should bring the user back to the home page

Single Player

- Entering the single player mode should not display a sign in page
- Entering the single player mode should bring the user to the lobby page immediately
- Clicking achievements in single player mode should display a list of locked achievements in a sheet
- Clicking select map in single player mode should display three map modes to select
- Clicking on any of the map mode should have a white border around the selected map mode
- Clicking on select character in single player should display eight characters to select
- Clicking on any of the character in character selection should display the character image on screen
- Clicking on any of the character in character selection should display the character name on screen
- Clicking on any of the character in character selection should display the character skills names
- Clicking back in character selection should bring the user back to the lobby page
- Clicking ready in the single player mode should display the start button without having a player in queue count display
- Clicking ready in the single player mode should disable all buttons except start button

Sign in

- Entering a previously signed up email with the correct password, and pressing sign in should log a player in.
- Entering a previously signed up email with the correct password, and pressing sign in should navigate the player to the lobby view.
- Entering a previously signed up email with an empty password, and pressing sign in should not allow the player to log in
- Entering a previously signed up email with an incorrect password, and pressing sign in should not allow the player to log in
- Entering a wrong email with the correct password, and pressing sign in should not allow the player to log in
- Entering a wrong email with the the wrong password, and pressing sign in should not allow the player to log in
- Entering the correct email with a space at the end and the correct password field should not allow the player to log in
- Entering in an incorrect email and password, and pressing sign in should not allow a player to log in. The player should not be redirected and should be prompted by an incorrect credentials message displayed to them
- Not entering anything and pressing sign in should not allow a player to log in

Sign up

- Entering a valid email that has not been signed up by any users (with a minimum length of 6 characters, containing '@' and a '.' after at least one character following the '@') and a strong password (minimum length of 6 characters) should allow the user to sign up.

- Entering an email which has not signed up by any users of the application, and a password
- Entering a previously signed up email with the correct password, and pressing sign up should not allow the player to log in.
- Entering a previously signed up email with an empty password, and pressing sign up should not allow the player to log in.
- Entering an empty email with password, and pressing sign up should not allow the player to log in.

Multiplayer Account Lobby

- Entering the lobby should display the user id of the player
- Entering the lobby should display the email of the user
- Clicking achievements should display a list of locked achievements in a sheet
- Clicking select map should display three map modes to select
- Clicking on any of the map mode should have a white border around the selected map mode
- Clicking on select character should display eight characters to select
- Clicking on any of the character in character selection should display the character image on screen
- Clicking on any of the character in character selection should display the character name on screen
- Clicking on any of the character in character selection should display the character skills names
- Clicking back in character selection should bring the user back to the lobby page
- Entering the ready button should allow the user the enter the queue with the most number of in queue players
- Entering the ready button should display the number of players in queue
- Entering the ready queue when there are three existing players in the queue should display the start button
- Entering the start button should navigate the player to the start game view
- Clicking ready should disable all buttons except start button

Multiplayer Guest Lobby

- Entering the lobby should display the user id of the player
- Entering the ready button should allow the user the enter the queue with the most number of in queue players
- Entering the ready button should display the number of players in queue
- Clicking achievements should display a list of locked achievements in a sheet
- Clicking select map should display three map modes to select
- Clicking on any of the map mode should have a white border around the selected map mode
- Clicking on select character should display eight characters to select
- Clicking on any of the character in character selection should display the character image on screen
- Clicking on any of the character in character selection should display the character name on screen

- Clicking on any of the character in character selection should display the character skills names
- Clicking back in character selection should bring the user back to the lobby page
- Entering the ready queue when there are three existing players in the queue should display the start button
- Entering the start button should navigate the player to the start game view
- Clicking ready should disable all buttons except start button

Gameplay

- **Game Control**
    - When the joystick is dragged from the centre, the player should move according to how far the joystick is dragged from the centre
    - When the user lets go of the joystick, the joystick should snap back to the centre and the player should not be moving at all
    - When one player moves, other players should be able to see the same movement with at most 0.2 seconds lag
    - When one player stops after moving, other players should be able to see the stopped player with at most 0.2 seconds lag
- **Collision**
    - When the player attempts to move beyond the right boundary of the screen, the player should stop at the right boundary
    - When the player attempts to move beyond the top boundary of the screen, the player should stop at the top boundary
    - When the player attempts to move beyond the left boundary of the screen, the player should stop at the left boundary
    - When the player attempts to move beyond the bottom boundary of the screen, the player should stop at the bottom boundary
    - When the player collides with an obstacle, the player should stop moving
    - When the player collides with another player, both players should stop moving
    - When a Hadouken collides with an obstacle, the Hadouken should disappear
    - When a Hadouken collides with the right boundary, the Hadouken should disappear
    - When a Hadouken collides with the left boundary, the Hadouken should disappear
    - When a Hadouken collides with the top boundary, the Hadouken should disappear
    - When a Hadouken collides with the bottom boundary, the Hadouken should disappear
    - When a Hadouken collides with an obstacle, the Hadouken should not damage any player that is hiding behind the obstacle
    - When a SlashAOE collides with an obstacle, the SlashAOE should disappear
    - When a SlashAOE collides with the right boundary, the SlashAOE should disappear
    - When a SlashAOE collides with the left boundary, the SlashAOE should disappear

    - When a SlashAOE collides with the top boundary, the SlashAOE should disappear
    - When a SlashAOE collides with the bottom boundary, the SlashAOE should disappear
    - When a SlashAOE collides with an obstacle, the SlashAOE should not damage any player that is hiding behind the obstacle
- **Gem**
    - When the player moves over a gem, the gem should disappear
    - When the player moves over a gem, the player should not slow down
    - When the player moves over a gem, the gem should disappear on all devices
    - When the player moves over a gem, the gem should not reappear after the player leaves the gem position.
    - When a Hadouken moves over a gem, the gem should not disappear
    - When a Hadouken moves over a gem, the Hadouken should not slow down
    - When a Hadouken moves over a gem, the Hadouken should not disappear
    - When a SlashAOE moves over a gem, the gem should not disappear
    - When a SlashAOE moves over a gem, the SlashAOE should not slow down
    - When a SlashAOE moves over a gem, the SlashAOE should not disappear
- **Closing Zone**
    - When the game starts, the zone should continuously get smaller.
    - When the game starts, being outside of the zone should damage the player at a rate of 5 health units per 0.5 seconds.
    - The zone should stop getting smaller when the diameter of the safe zone is already 50 CGDouble.
    - The zone should never get bigger.

CombatSystem:

- When a DamageEffect is initially applied to an entity, then the entity should immediately incur damage specified as initialDamage.
- When there is a tick update in CombatSystem, then DamageEffect.elapsedTime should increase by the time interval provided.
- When a DamageEffect's elapsedTime equals or exceeds its duration, then it should be removed from the entity.
- When a DamageEffect is active and the game updates, then the entity should receive damage calculated as damagePerTick * time at each update.
- When an entity with an enabled Dodge component is attacked, then no damage from DamageEffect should be applied.
- When damage from a DamageEffect is applied to an entity, then the Health component of the entity should decrease by the amount of damage.
- When damage is applied to an entity, then DamageDealtMetric and DamageTakenMetric should record the damage appropriately for both the source and the target.
- When CombatSystem processes multiple DamageEffects on a single entity in one update, then each effect should independently apply its specified damage.
- When a DamageEffect expires, then subsequent updates should not process this effect, and it should not apply any more damage.

- When a DamageEffect is removed from an entity before expiring, then it should immediately cease to apply damage, and no further damage should be recorded.
- When damage is applied through DamageEffect, then the health reduction of the entity should exactly match the cumulative damage specified by the DamageEffect over its active period.
- When multiple updates to CombatSystem are invoked simultaneously from different threads, then the system should synchronize access to entity components to prevent data corruption.
- When a DamageEffect's duration has completed, then the effect should be removed, and the entity should not receive further damage from this effect in future updates.
- When an entity's health reaches zero due to DamageEffect, then no negative health value should be recorded, demonstrating proper handling of minimum health constraints.
- When a non-player entity like a Hadouken or SlashAOE moves over a gem, then the gem should not disappear and should not affect the entity's speed or visibility.

Skill System

- **SlashAOESkill**
    - Activation Cooldown:
        - When activated, cooldownRemaining should be set to cooldownDuration of the skill.
        - After activation, cooldownRemaining should decrease over time and reach 0 after cooldownDuration seconds.
    - Activation Effect:
        - Activating SlashAOESkill will spawn a SlashAOE entity at the caster's position.
        - All devices should display the activated SlashAOESkill
        - When another player's collider overlaps with the SlashAOE entity's collider, and the target player's entityId is not the same as the caster's entityId, the target player should receive the specified DamageEffect of SlashAOE
        - No health deduction should occur if the target player is outside the collider.
    - Simultaneous Activation:
        - When two players activate SlashAOE simultaneously (i.e. in the same update() call) within range, both players should receive the DamageEffect debuff.
        - Health decrease in one device should be the same as the health decrease in all other devices
    - Cooldown After Activation:
        - The skill should not be re-activatable while its cooldown is greater than 0.
- **HadoukenSkill**
    - Activation Cooldown:

        - When activated, cooldownRemaining should be set to cooldownDuration of the skill.
        - After activation, cooldownRemaining should decrease over time and reach 0 after cooldownDuration seconds.
    - Activation Effect:
        - Activating HadoukenSkill will spawn a Hadouken entity at the caster's position.
        - Based on the caster's rotation, Hadouken will move in the direction of where the caster is facing
        - All devices should display the activated Hadouken
        - When another player's collider overlaps with the Hadouken entity's collider, and the target player's entityId is not the same as the caster's entityId, the target player should receive the specified DamageEffect of Hadouken
        - No health deduction should occur if the target player is outside the collider.
    - Simultaneous Activation:
        - When two players hit the same Hadouken simultaneously (i.e. in the same update() call) within range, both players should receive the DamageEffect debuff.
        - Health decrease in one device should be the same as the health decrease in all other devices
    - Cooldown After Activation:
        - The skill should not be re-activatable while its cooldown is greater than 0.
- **DashSkill**
    - Movement Requirement:
        - When activated, the player's position should be moved in the direction it's facing by 100 units.
    - Distance Verification:
        - Upon activation, the player's position should move forward by 100 units.
    - Collision Handling:
        - If a collision occurs with another dashing player, both players should stop and trigger a stop animation.
        - No further movement should occur upon collision.
    - Cooldown Management:
        - cooldownRemaining should be set to cooldownDuration upon activation.
        - The skill should be inactive while on cooldown.
- **DodgeSkill**
    - Invulnerability Effect:
        - Upon activation, the player should receive a DodgeComponent.
        - All devices should display the shield view around the player that activated it
        - No damage should be received while DodgeComponent is active.

        - After the expiry duration, DodgeComponent is removed from the player.
        - All devices should no longer display the shield view around the player
    - Movement During Dodge:
        - The player should be able to move freely while dodging.
        - The glowing effect should be visible during the dodge.
        - The player should still be able to have the same collision behaviours as when the dodge skill has not been activated
    - Cooldown Activation:
        - cooldownRemaining should be set to cooldownDuration upon activation.
        - The skill should not activate again during cooldown.
- **RefreshCooldownsSkill**
    - Cooldown Reset:
        - Upon activation, all other skill cooldowns should reset to 0, allowing immediate reactivation.
        - RefreshCooldownsSkill's on cooldown should not reset, but the skill goes on cooldown instead.
    - Reactivation Post-Cooldown:
        - After the cooldown elapses, RefreshCooldownsSkill's should be activatable again.
        - Upon activation, the cooldown should be reset to its original duration.

5.2: Individual Contributions

Reyaaz: Authentication, Matchmaking, Multiplayer (Adapters, Factory / Generic Classes), Wrappers, Primitive Data Types, Shapes, Collider Component, Rigidbody Component, TransformSystem, CollisionSystem, PhysicsSystem, CanvasView / CanvasViewModel, LobbyView / LobbyViewModel, SignInView / SignInViewModel, EntityComponentManager, Host, Client ViewModels, Health, Score, Dodge, Gem, Characters, Map, Collector, Collectable, Invisible, Skills multiplayer, Audios

Bertrand: JoystickView, GameControl, Attack Component, AttackStrategy, Health System, Dodge Component, Dodge System, Lifespan Component, Lifespan System, GameMode, GameOverView, PlayerComponent, Collector, Collectable, Collection System, EntityView / EntityViewModel (gemCount), View Navigation, Fonts

Jivesh: Entity, Component, System, SystemManager, EntityComponentManager, EntityMetricsRecorder(+adapter), MetricsRepository, Metrics(++), MetricsObserver(++), AchievementManager, Achievement(++), SingleDocumentStorageManager(++), Environment Effects System, Closing Zone Entity,

Joshen: Skills, SkillCaster, SkillCasterSystem, Skill Protocols, GameLoop, EntityOverlay, ECS, CanvasView, SkillWrapper, EntityView, EntityViewModel, System Design, Assets, DamageEffect, CombatSystem, Hadouken, Frontend Views, STEPS Poster, STEPS Video