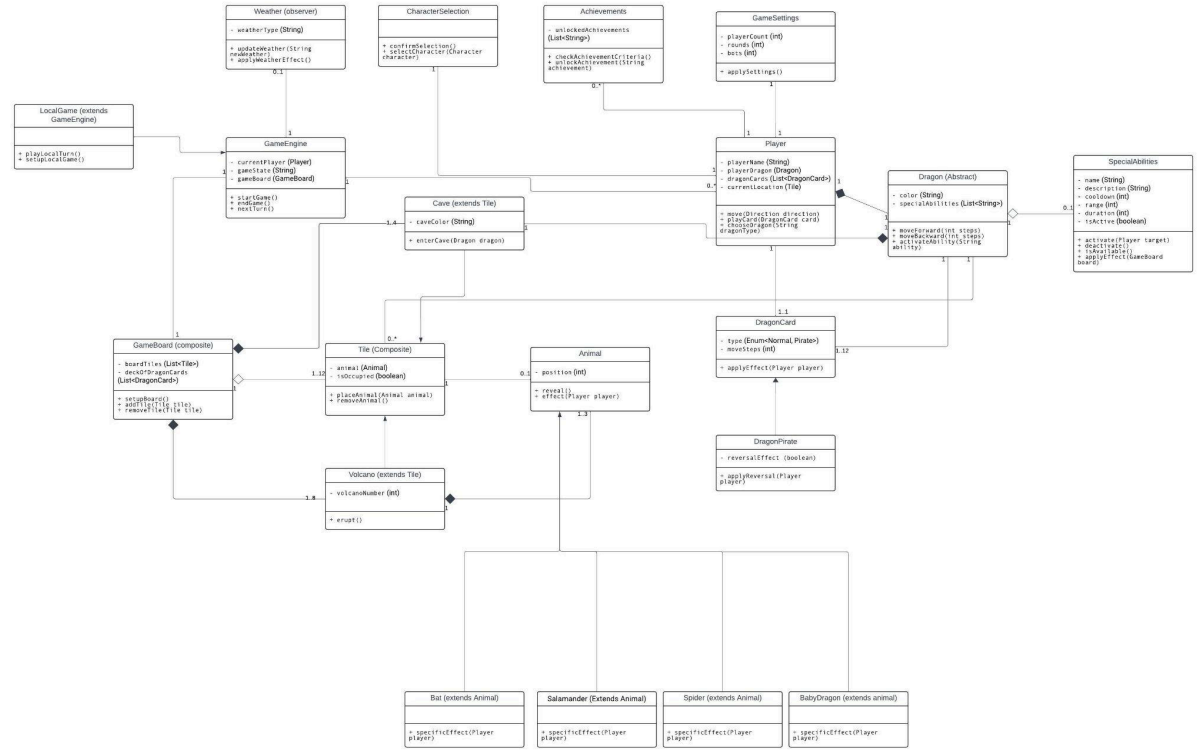
Sprint 2

Mun Hong Ooi 31199399

Object-Oriented Design and Design Rationales



[Class Diagram]

Summary of Key Game Functionalities

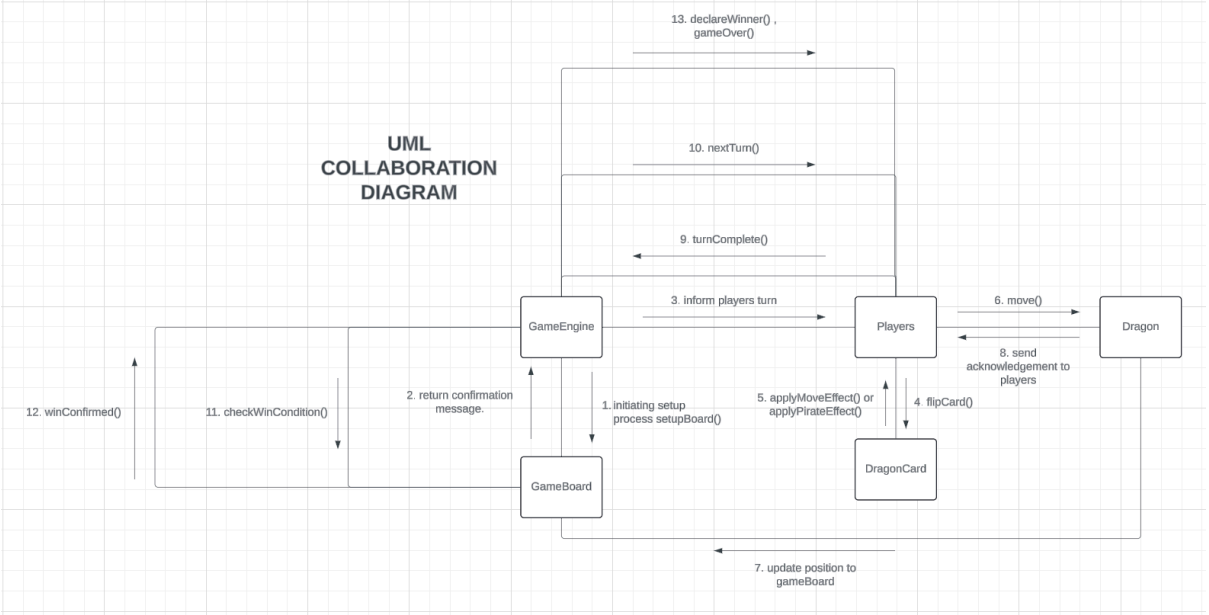
(i) Setting Up the Initial Game Board:
The GameBoard class is responsible for initialising the game state. It randomly positions the dragon cards on the board. The setup process includes creating volcano segments and placing animals on them. The GameEngine class invokes the setup by calling GameBoard.setupBoard().

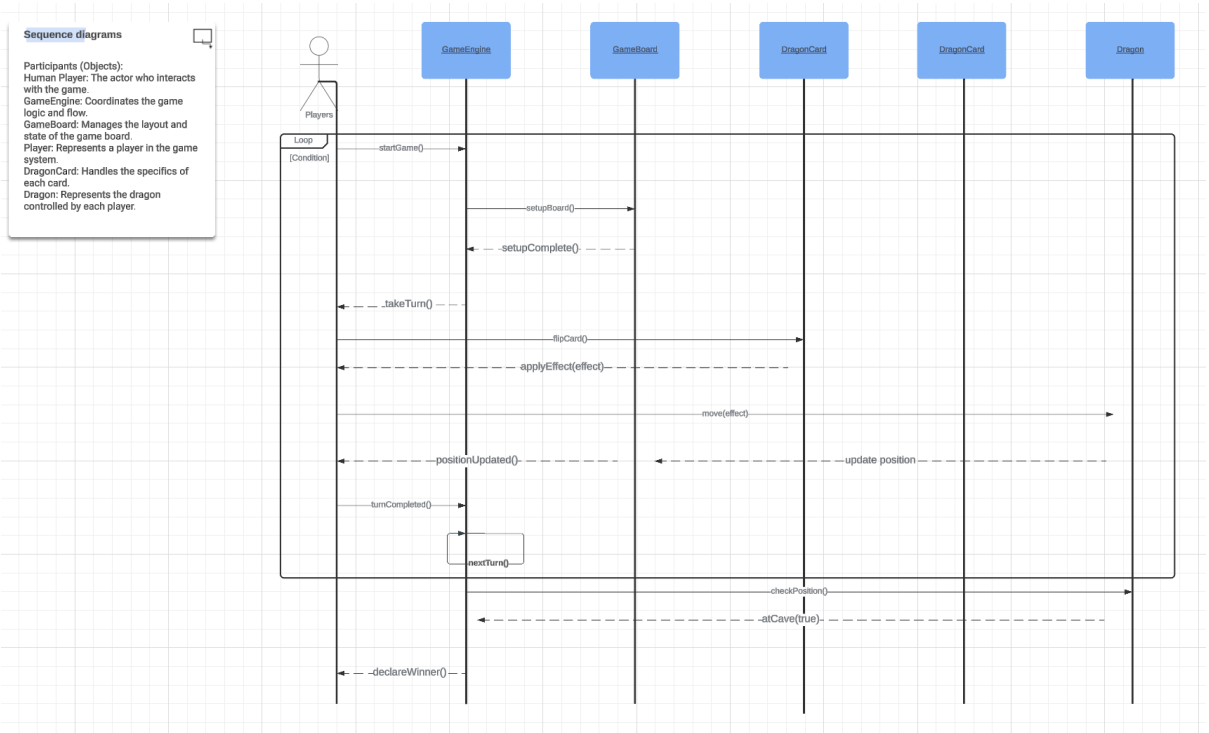
(ii) Flipping of Dragon ("chit") Cards:
Each Player has a method flipCard() which interacts with the DragonCard class. When a card is flipped, its effect is determined by the card type, which could be either a normal move or a pirate effect executed by the applyEffect() method of the DragonCard.

(iii) Movement of Dragon Tokens:
The Dragon class has a move() method that takes the flipped card's effect and applies it to the dragon's position on the board. The move depends on both the current position of the dragon and the effect of the last flipped dragon card.

(iv) Change of Turn to the Next Player:
The GameEngine manages the turns and has a method nextTurn() which updates the currentPlayer attribute. It ensures that the game flow proceeds in a round-robin fashion amongst the players.

(v) Winning the Game:
The condition for winning is checked after every move. The GameEngine checks if any Dragon has returned to its starting Cave with an exact roll. If so, the winGame() method declares the player as the winner and ends the game.





UML Sequence Diagram and UML collaboration diagram

Design Rationales

Key Classes

GameBoard: The decision to design GameBoard as a class rather than a method is based on several factors. Firstly, as the central component responsible for managing the state of the game board throughout the entire game session, it requires a cohesive structure to encapsulate its functionality. By implementing it as a class, we create a self-contained entity with its own attributes and methods, facilitating better organisation and modularity in the codebase. Additionally, designing GameBoard as a class allows for easier instantiation and manipulation, enabling seamless integration with other components of the game system.

Player: The Player class serves as a pivotal element in the game, representing individual participants and encapsulating all related functionalities. By defining Player as a class, we establish a clear abstraction for managing player-specific data and behaviour. This design choice promotes encapsulation, allowing us to group together relevant attributes such as the player's dragons, cards, and score within a single entity. Moreover, utilising a class-based approach enables the implementation of player-specific methods and interactions, facilitating intuitive player management and enhancing the overall maintainability of the codebase.

Key Relationships

Aggregation vs. Composition (GameBoard and Tile): In the relationship between GameBoard and Tile, the decision to utilise aggregation reflects the nature of their association. While GameBoard oversees the management and arrangement of Tiles on the game board, each Tile maintains its own independent state and functionality. This distinction underscores the concept of aggregation, wherein the GameBoard aggregates multiple Tile instances without directly controlling their lifecycle. By modelling this relationship using aggregation, we establish a clear separation of concerns, promoting modularity and flexibility in the design.

Composition (Player and Dragon): The relationship between Player and Dragon is characterised by composition, emphasising the strong ownership and dependency between the two entities. In the context of the game, a Player inherently possesses Dragons, with the lifecycle of the Dragon tightly bound to that of the Player. By encapsulating the Dragon object within the Player class, we establish a cohesive unit wherein the Player serves as the container for its associated Dragons. This composition-based design promotes encapsulation and code organisation, simplifying the management of player-related functionalities and promoting code reuse.

Decisions Around Inheritance

Inheritance for DragonCard and DragonPirate: The use of inheritance for the DragonCard and its subclass DragonPirate is driven by the need for code reuse and extensibility. Both DragonCard and DragonPirate share common attributes and methods, such as flipping the card and applying card effects. By defining DragonPirate as a subclass of DragonCard, we can leverage inheritance to inherit these shared functionalities while allowing for specialised behaviours unique to the Pirate card type. This approach facilitates code reuse, promotes consistency, and enables polymorphic behaviour, enhancing the flexibility and scalability of the card system.

Cardinality Decisions

Player to DragonCard (0..1): The cardinality between Player and DragonCard is specified as 0..1 to accurately model the relationship between a player and their hand of cards. In the game context, a player may hold zero or one DragonCard at any given time, representing the absence or presence of a card in their hand, respectively. This cardinality constraint ensures that the player's hand remains limited to a single card, preventing scenarios where multiple cards are held simultaneously and maintaining the integrity of the game mechanics.

GameBoard to Tile (1..12): The cardinality relationship between GameBoard and Tile is defined as 1..12 to reflect the expected configuration of the game board. In most game setups, a GameBoard typically consists of twelve Tiles arranged in a predetermined layout. By enforcing this cardinality constraint, we ensure that the GameBoard always contains the necessary number of Tiles required for gameplay, facilitating consistent and predictable behaviour throughout the game session.

Design Pattern Application

Factory Pattern: The Factory Pattern is commonly utilized to handle the dynamic creation of various types of Animal objects to cater to specific game setup requirements. However, in the current implementation of the game, which primarily focuses on the functionality of flipping cards, the need for dynamically creating different types of objects at runtime is not apparent. Given the simplicity of the existing functionality and the absence of complex game mechanics like player movement or winning conditions, introducing the Factory Pattern might add unnecessary complexity to the codebase. As the game development process follows an iterative approach, it's more practical to implement design patterns like the

Factory Pattern when they are specifically needed for new features rather than upfront. Therefore, the decision not to incorporate the Factory Pattern at this stage is based on considerations of simplicity, future expansion potential, and the iterative nature of development.

Singleton Pattern: the Singleton Pattern wasn't utilized in the current implementation, given that the code focuses on a single functionality rather than a complete game. Firstly, Singletons are typically employed when there's a clear need for a single instance of a class throughout the application's lifecycle. In this early stage of development, with only one aspect of the game being implemented, the necessity for a global instance might not be evident. Secondly, avoiding the Singleton Pattern allows for greater flexibility in the future development phases. As the game expands and additional functionalities are introduced, the design might evolve to accommodate multiple instances of various classes, making a rigid Singleton implementation less suitable. Thirdly, the absence of a Singleton promotes simplicity in the codebase. Since the focus is currently on a specific feature, keeping the design straightforward without the added complexity of managing a Singleton instance simplifies both development and maintenance tasks. Overall, deferring the use of the Singleton Pattern until the broader architectural requirements of the game become clearer enables a more adaptable and scalable design approach.

Observer Pattern: While not explicitly implemented, the Observer Pattern could be leveraged to notify players of changes in the game state. The Observer Pattern enables a decoupled communication mechanism whereby observers (players) register interest in specific events (game state changes) and receive notifications when those events occur. Although the current game design may not necessitate real-time event notification, the potential application of the Observer Pattern highlights the design's flexibility and scalability to accommodate future enhancements or complexities in game functionality.

Decorator Pattern: Although the Decorator Pattern could be useful for dynamically adding or modifying behaviors of game elements, such as cards or tiles, it might be overkill for the current requirements. With the game primarily revolving around card flipping and basic player interactions, the need for dynamically altering or enhancing game elements' behavior is minimal. Implementing the Decorator Pattern could introduce additional layers of abstraction and complexity, making the code harder to understand and maintain, especially considering the relatively simple functionality of the game at present. Therefore, deferring the use of the Decorator Pattern until the introduction of more complex game mechanics or customization options could lead to a more focused and manageable codebase.

Software Prototype

The game has set up the initial game board, which includes a home page, and when the “PLAY NOW” button is clicked, it will be directed to the game board. The chosen functionality of this game currently is flipping of chits cards. The chits cards can be flipped

when the start button is pressed, and cannot be flipped once the stop button is pressed. There is also a shuffle card button, when it is clicked, it will shuffle the chits. The volcano cards are placed around the chits randomly, and there are four cave cards, each with a specific coloured token placed on top of it to represent the players. All of the artwork for the cards images are generated by ChatGPT, and the music used in the game is created by GameChops. (2020). Poke & Chill [Album]. Retrieved from https://www.youtube.com/watch?v=2DVpys50LVE, with no copyrights. You will notice that there are some classes mentioned in the design rationale and summary of key game functionalities, are not yet implemented. Some classes like PlayerCard.py and Player.py are also not fully implemented, as these classes are not required to be done to implement the "flipping of chits cards" functionality in this individual sprint.

Executable Deliverable

The executable file in the Unix executable file format has been compiled and downloaded into dist. The dist folder contains both the game file in unix format and in .app format along with the assets. The folder containing all of the files required to launch the game are also compiled into a zip folder named updateGameFile.zip

Download the zip file named updateGameFile.zip or distVer2 folder into your local machine. Start the game by clicking on the Game file with the unix executable format. When clicked on the game file, a terminal will pop up and the game will start in a few seconds after the Game file is opened. If the game does not start, click on the Game.app icon while within the Game file to start the game. Another option is to type in the following command into the terminal when launching the Game file: ./dist/Game.app/Contents/MacOS/Game. The target platform for this game is MacOS, and the game will work with macOS Ventura version 13.2.1 or later, note that the game will not work on older macOS versions.