FIT3077 - Sprint 2

33156018

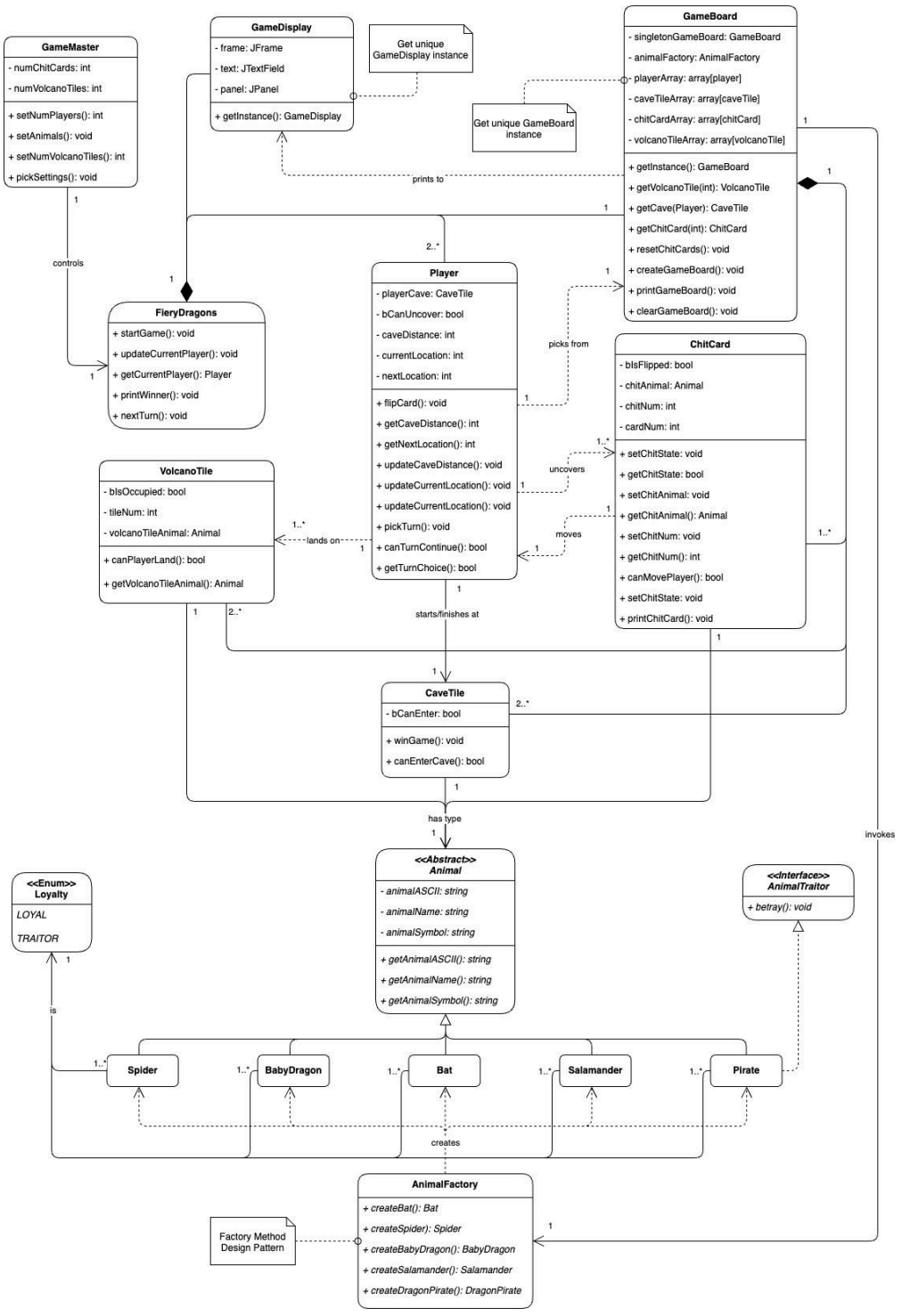
Georgia Kanellis

Video Demonstration Link **2**
UML Diagram **2**
Key Game Functionalities **3**
Initial Game Board Set-up 3
Flipping Dragon (“Chit”) Cards 4
Movement of Dragon Tokens 5
Change of Player Turn 6
Winning the Game 7
Design Rationale **8**
Key Classes 8
Animal 8
ChitCard 8
Key Relationships 9
Inheritance 9
Cardinalities 9
Design Patterns 10
Singleton Pattern 10
Factory Method Pattern 10

Video Demonstration Link

https://www.youtube.com/watch?v=eVytryTgb3M

UML Diagram

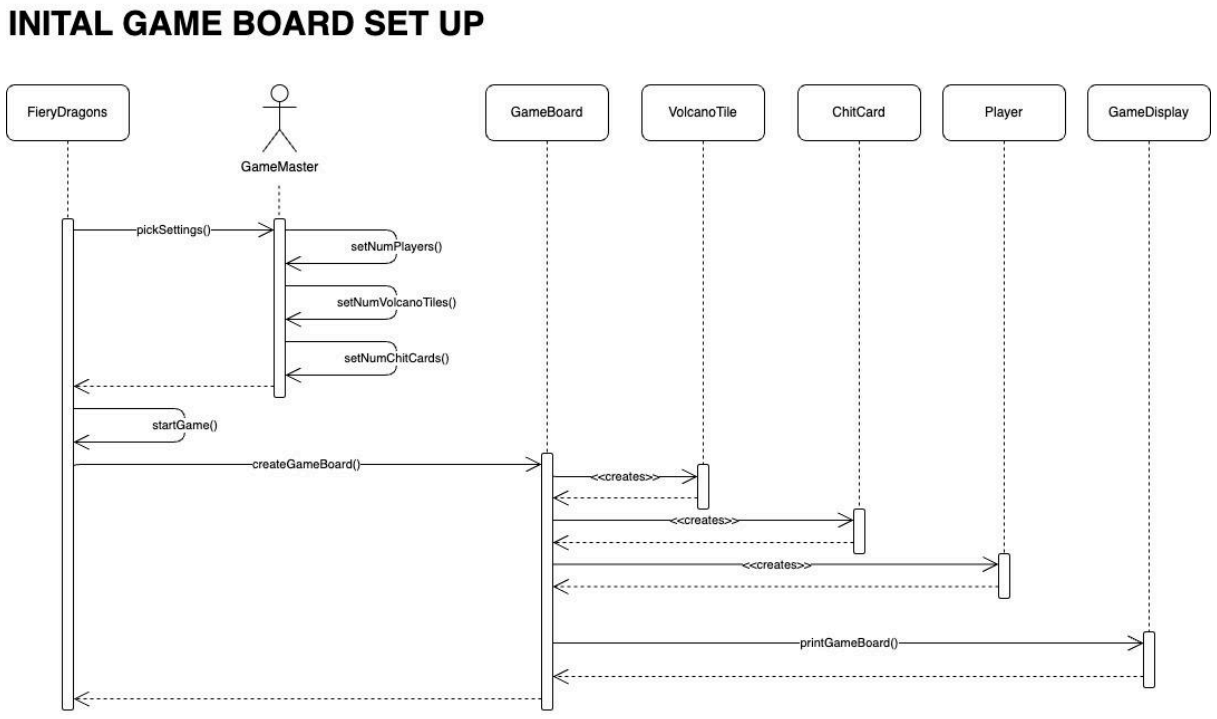


Key Game Functionalities

Initial Game Board Set-up

The Fiery Dragons game board is initially set up when the FieryDragons class is run, as it is my main file. The main method immediately creates an instance of the gameBoard, and calls the GameMaster method pickSettings(). PickSettings() chooses whether or not to play the game with default rules (e.g. 4 players, 24 volcano tiles, 26 chit cards etc.) or to play with augmented rules (choose number of players etc). Augmented settings is out of the scope for this sprint. After this occurs, the game board is initialised using the createGameBoard() method. This method creates the game board, initialising the volcano tiles, chit cards, players and caves. Each of these game board components are represented through arrays filled with instances of its respective classes. For example, instances of chitCard are held within chitCardArray(), whereas the volcanoTile instances are held within the volcanoTileArray. The chit cards are allocated a random animal type, as well as a random number to represent their effect on the player. The randomisation of animal types is controlled so that there is always a relatively even spread of animal types across the chit cards to minimise unfairness. Each animal type has its own ASCII representation to allow players to differentiate between them. After the game board has been initialised, the printGameBoard() method is called. Which prints the GameBoard to the JFrame displaying the game board's UI to the users. This UI is made up of a combination of JSwing entities and ASCII art. After the game board is printed, the actual game of Fiery Dragons is ready to begin.

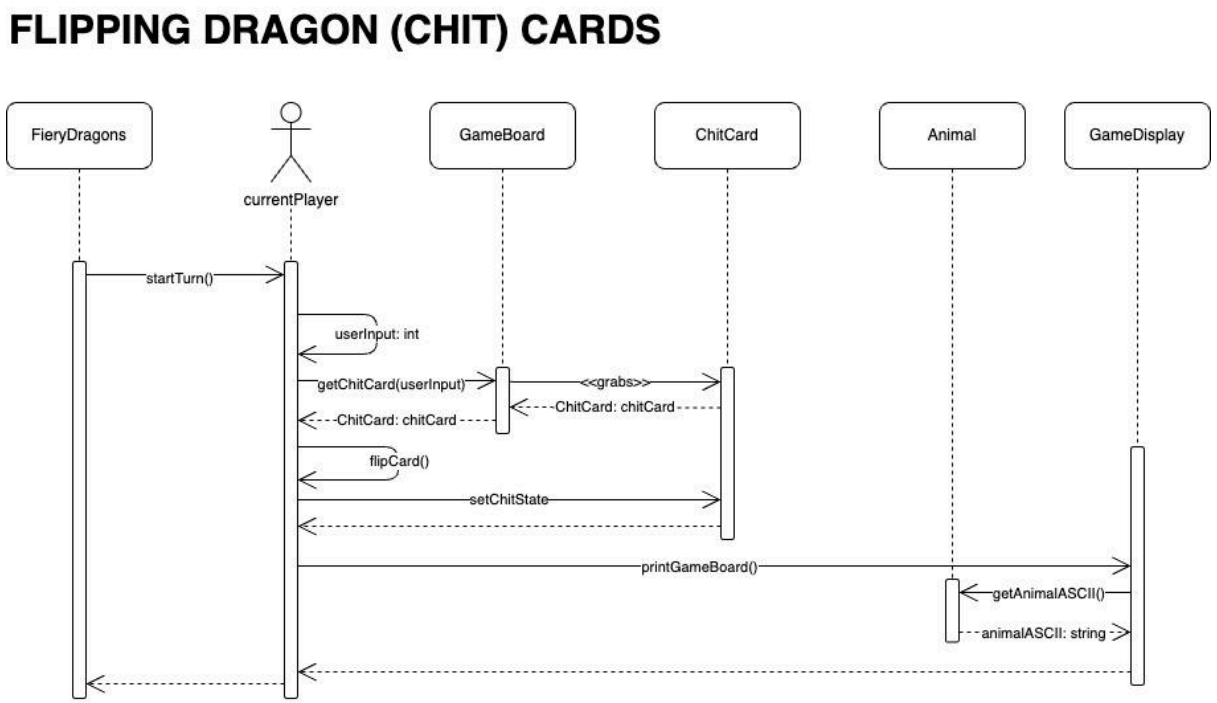
Sequence Diagram



Flipping Dragon (“Chit”) Cards

Flipping the Chit cards is done by the currentPlayer, and is the act of picking a generic, numbered card out of an array, and turning it to show its uncovered ASCII design. The uncovered design displays a number of animals (one or more) of a certain type (Bat, BabyDragon, DragonPirate, etc.). The currentPlayer is prompted to select a chit card at the beginning of the startTurn() method. This selects a chit card from the chitCardArray, by using the player input to index the chitCardArray and access a specific instance. After the current player’s input has been verified (not already flipped, is an integer and is in range) the flipCard() method is called. The selected chitCard instance’s state is switched to uncovered. The GameBoard is reprinted after every flipCard() call, so that the new uncovered ASCII art is revealed to the players.

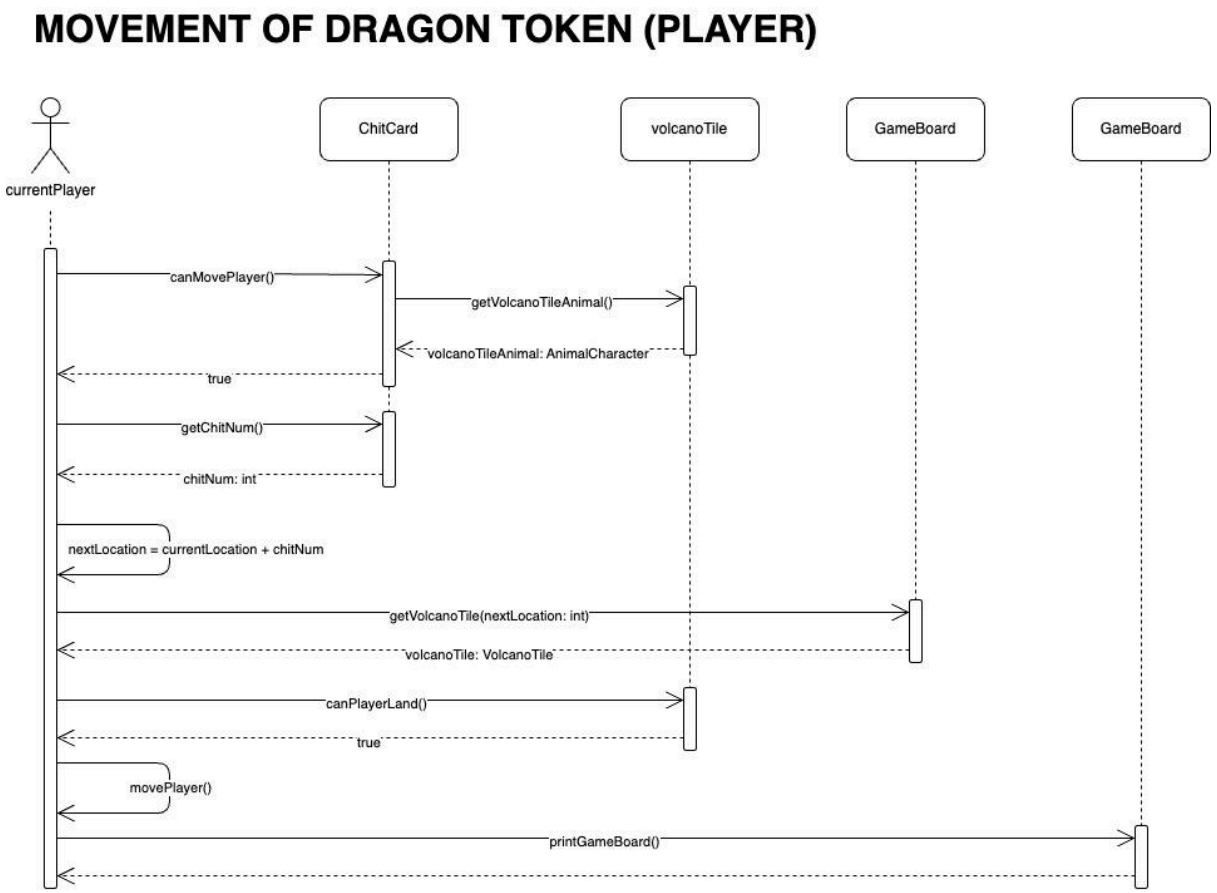
Sequence Diagram



Movement of Dragon Tokens

The movement of Dragon Tokens is triggered after the flipCard() method is called to uncover a chitCard. To check if movement is possible, the flipped chitCard's canMovePlayer() method is called. This method compares the Animal type found at the currentPlayer's currentLocation, and its own Animal type. If these two Animal types are different, the canMovePlayer() returns false. In the case that the two Animal types are the same, an additional check is performed. The currentPlayer's next potential location is updated to be the currentLocation plus the chitNum. This nextLocation value is then used within the getVolcanoTile(int) method to find the volcanoTile instance at that index. The canPlayerLand() method is then called, checking if the volcanoTile is already occupied by another player. If the canPlayerLand() method returns true, then movePlayer() is called and updates the currentPlayer's currentLocation and reprints the gameBoard to show the updated Dragon Token location.

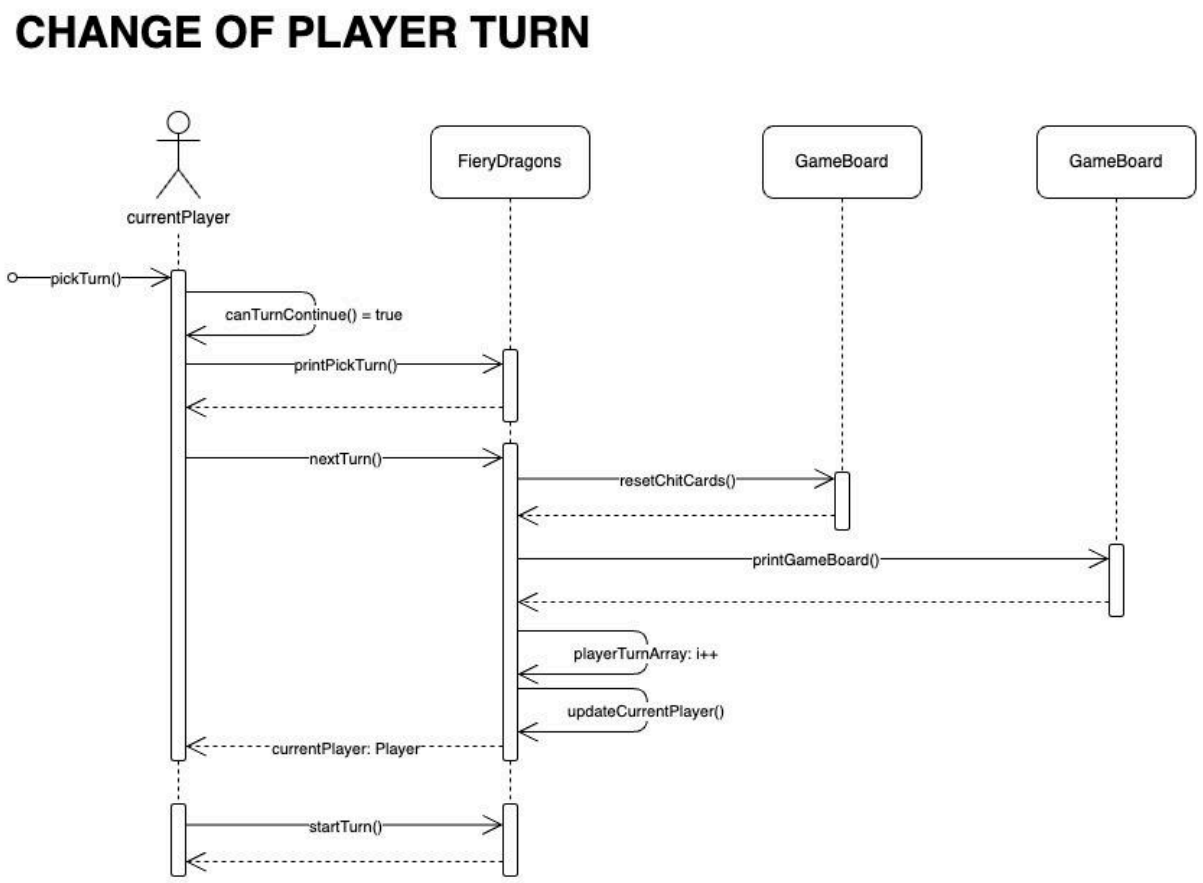
Sequence Diagram



Change of Player Turn

The change of turn to the next Player occurs after the currentPlayer has either no more available turns to pick, or has chosen to pass after flipping a chitCard. This is determined through currentPlayer's canTurnContinue() method. If the currentPlayer chooses to flip a chitCard, but the Animal types are different, bCanUncover is switched to false. After currentPlayer has flipped a chitCard, pickTurn() is called. If canTurnContinue() is false, then nextTurn() is called. Additionally, if canTurnContinue() is true, but the currentPlayer chooses to pass, then nextTurn() is similarly called. At the beginning of the nextTurn() method, all the chitCards' states are reset, displaying the generic, numbered ASCII design, after which the gameBoard is reprinted. The playerTurnArray is iterated over, and the player at the next index is updated to be the currentPlayer, after which the startTurnPrompt is printed, which asks the user to pick a chitCard. This begins the next player's turn.

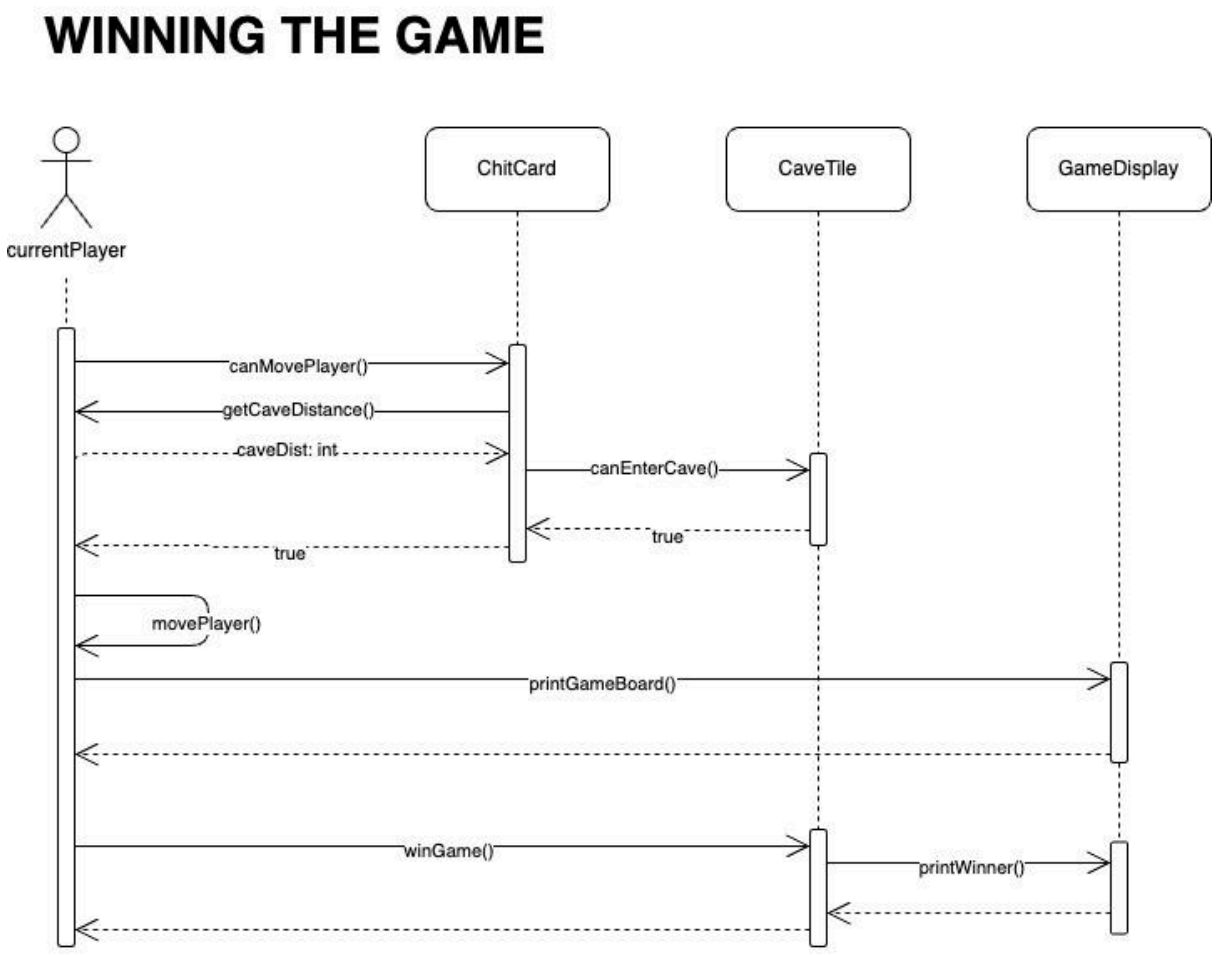
Sequence Diagram



Winning the Game

The game is won when a Player reaches their cave after traversing the entire gameBoard. This occurs when the currentPlayer's caveDist value is equal to zero. During the canMovePlayer() method, the currentPlayer's caveDist is compared with the chitNum. If the chitNum is higher than caveDist, then canMovePlayer() returns false, otherwise canEnterCave() is called. If chitNum and caveDist are equal then canEnterCave() is set to true. Now movePlayer() is called and updates the currentPlayer's currentLocation and reprints the gameBoard to show the updated Dragon Token location back inside its cave. After this occurs, winGame() is called, ending the game and calling printWinner() which displays a win screen to the console.

Sequence Diagram



Design Rationale

Key Classes

Animal

The Abstract Animal class serves as a component in representing the various animals that the player will come into contact with over the course of “Fiery Dragons”. Each Animal instance embodies specific attributes and methods, such as an ASCII representation, a character symbol, and a name. Instances of Animal also each have a loyalty, differentiating themselves between LOYAL and TRAITOR. The Animal class is abstract due to the common attributes between the FieryDragon animals of Bat, Baby Dragon, Spider, Salamander, and Dragon Pirate. Thus, by encapsulating these commonalities within the Abstract Animal class, we enhance the reusability and extensibility of the game. Adding additional Animals to the game is much easier, as I have a framework and methods that all inheriting Animals must implement. Additionally, by not including the betray() method within the Abstract Animal class, even though the DragonPirate utilises it, follows the Interface-Segregation principle. As none of the other subclasses implement that method, it was inappropriate to include within the abstracted methods.

It is not appropriate for the Animal Class to be a method, mainly because it encloses all the common attributes and methods of its subclasses. This also allows for future extensions to be made easily, as all the necessary behaviours of an Animal are held within the Animal class. Therefore, if we wish to add additional Animal types, they can easily be added without affecting the current Animal Types, adhering to the Open-Close Principle. If the Animal class was a method, it would be incapable of such encapsulation, and would fail to maintain organisation across closely related classes like the different Animal Types.

ChitCard

The ChitCard class encapsulates the information and behaviour of a single chit card. It holds data relating to the instance’s animal type, the number of animals on that card, its generic number, its covered state, as well as the generic ASCII art that is common across all chit cards. The ChitCard class is necessary as it facilitates the creation and manipulation of multiple chit cards throughout the game. By encapsulating chit card specific attributes and behaviours, it allows for easy instantiation of new chit cards, as well as manipulation of existing chit cards, without affecting other parts of the system.

The ChitCard class could not be a method, as it separates responsibilities of the system. This allows us to increase our maintainability as all relevant information and responsibilities are held together. This follows the Single Responsibility principle, ensuring that all information pertaining ChitCard’s are held in the same place.

Key Relationships

Within my system, I have used a compositional relationship between the classes ChitCard and GameBoard. This is because the GameBoard cannot exist without the ChitCardArray, an array filled with instances of the ChitCard class. This relationship is appropriate as the GameBoard would be incomplete and unplayable without the presence of Chit Cards. The GameBoard class is responsible for initialising and maintaining the chitCards, and as such cannot exist without it (and thus, an aggregation relationship would be inappropriate). By using composition, the GameBoard class tightly controls the initialization and maintenance of chitCard objects. This ensures that ChitCards are properly managed within an instance of the GameBoard class.

Another key relationship that I included within my system is an association relationship between the classes Player and CaveTile. This association enables the Player Class to interact within the VolcanoTile objects, allowing them to move their Dragon Tokens to different locations over the course of the game. Unlike a composition relationship, as seen above, where one class is made up of instances of another, an association is not dependent on the other class for its own existence. The Player class and the volcanoTile class exist independently of each other, as the Player can land on a different volcanoTile instance, leaving the previous instance empty. Thus, association is an appropriate relationship to demonstrate the flexibility and independence between the two classes.

Inheritance

In my design of the Fiery Dragons game, the animal subclasses (e.g. BabyDragon, Salamander, Spider, Bat and DragonPirate) all inherit from the Abstract Animal superclass. This implementation aligns with multiple key principles of Object-Oriented Design and enhances the overall maintainability and extensibility of the system. By establishing an abstract Animal class, the common attributes and methods shared by all animal types are encapsulated within the one class. Ths promotes code reusability, and reduces redundancy across the system. All the subclasses inherit these shared characteristics, helping to streamline and quicken future implementation, as each class has familiar code structure. Extensibility is drastically enhanced through the use of inheritance, as it ensures that any future additions to the animal subclasses will have uniform code structure, and methods. This ensures that its methods and functionalities will be the same while still being open for the instruction of unique behaviour. This follows the Open/Close principle, as addition of new animal types is extremely easy, but does not require modifying already existing code.

Cardinalities

One set of cardinalities that is interesting within the game Fiery Dragons, is the relationship of one Player that can land on one or more volcanoTiles. This cardinality indicates that one Player can be associated with multiple volcanoTiles. This is because, as the Player moves around the gameboard, they will land on different VolcanoTiles.

Another set of cardinalities that I chose specifically within the system, is the composite relationship between 2 or more Players and one FieryDragon class. This cardinality indicates that in order to implement the Fiery Dragons game, more than one person must be playing.

Without more than one person to play the game, it is impossible to play. I also did not give the Player cardinality an upper limit. This is because in the augmented rules of the game, more than the default number of players is allowed. The default number of players within the game is four players.

Design Patterns

Singleton Pattern

One Design pattern that I have utilised within the implementation of this system is the Singleton. The Singleton is a Pattern that ensures that only one instance of a class can exist at any point in time. I have utilised this pattern within the GameBoard class, as well as in the GameDisplay class. Within these two classes it is especially important to only have one instance, as without it, many errors and bugs could arise. For instance, the GameBoard class is composed of a chitCardArray which holds instances of the chitCard object at each index. Without utilising the Singleton pattern, selecting a chitCard from the chitCardArray would not work. This is because tthe flipCard() method would be accessing an instance of GameBoard that had not initialised the chitCardArray, even though the GameBoard has already been instantiated elsewhere. By utilising the Singleton design pattern I could guarantee that every time I accessed the GameBoard it would be the desired instance.

The other location where I utilise the Singleton Design Pattern is within the GameDisplay class where a similar problem to the previous example could occur. When attempting to clear the JFrame, the instance that was being accessed would not be the same instance as desired. The Singleton pattern would once again guarantee that I would always be working within the instance that had already been created, and would not allow the creation of a separate instance.

Factory Method Pattern

I also utilised the Factory Method Design Pattern within the Fiery Dragons game implementation. I used this within the AnimalFactory class as an alternative way to create Animal subclasses such as the Bat, Salamander etc. In my implementation, the AnimalFactory is my concrete creator, with the concrete product being the Animal subclasses, which inherited from the Abstract Animal class, my product. The creator in my implementation is the GameBoard. I utilised this pattern to increase the maintainability and extensibility of the Animal subclasses. To create an additional Animal subclass, all that is necessary is to create an additional FactoryMethod() and another Animal Type. This follows the Open/Closed Principle, as no modification is needed, just extension.

Another reason I chose this pattern is to allow for different configurations of animals in the augmented rules sets of the Fiery Dragons game. Although this implementation is not part of this sprint., by utilising the Factory Method, I have decoupled the object creation from the class itself. Now its creation is reliant on the client, the GameBoard. As such the GameBoard decides which animals are part of its ruleset. And thus, different animal type configurations will be much easier to implement in the future, especially dynamic configurations. This follows the Open/Closed Principle, as it allows for the implementation of additional functionality without needing to modify the existing code.