FIT3077 Sprint 2

Name: Ong Jing Wei
Student ID: 32909764

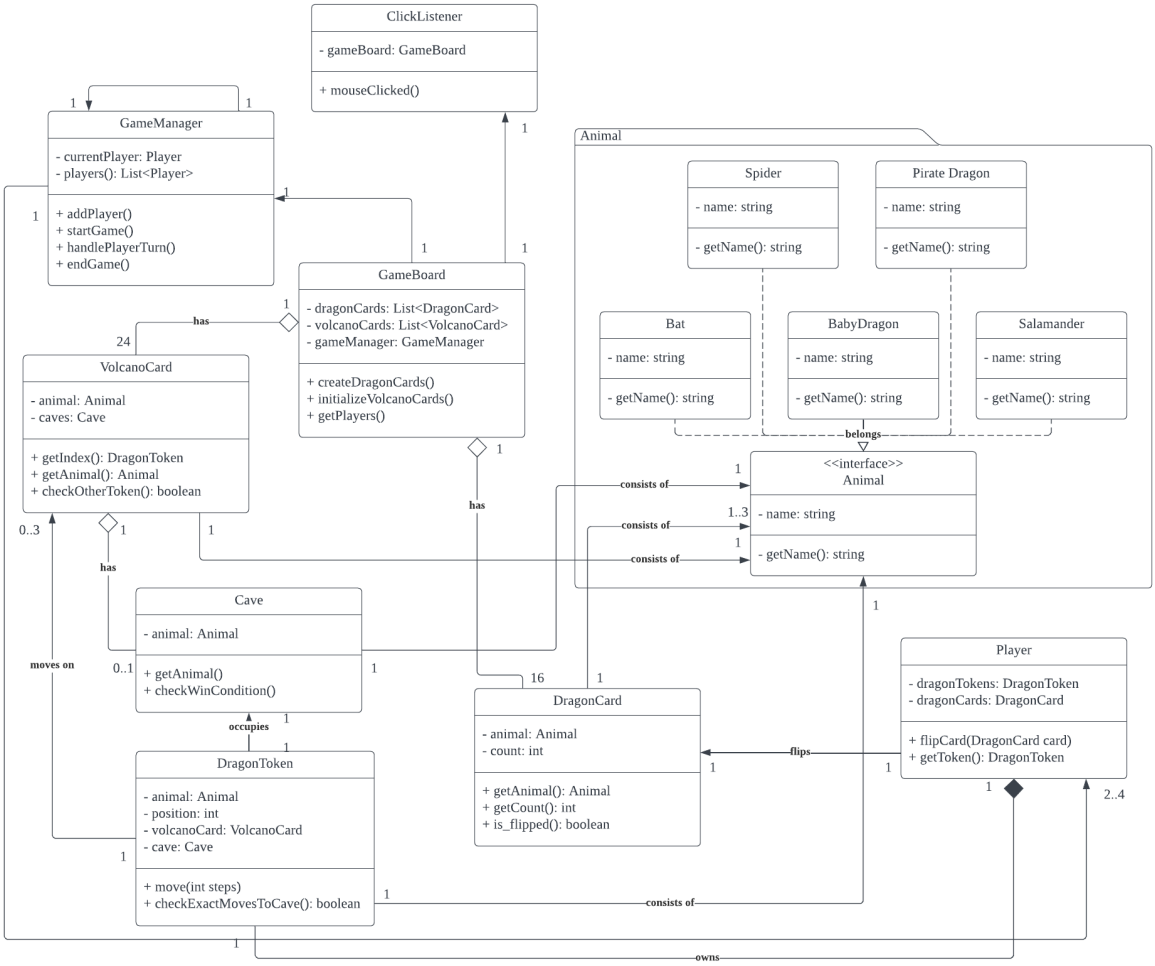
I acknowledge the use of ChatGPT to generate advantages and disadvantages for the design rationale, suitable design pattern and refine my academic language in this report.

* Executable file is located under Project directory. The command line to run this file is
java -jar 32909764_FieryDragonGame.jar

This class file version is 63.0, where you need a more recent version of Java Runtime.

The jar file is created using command
javac *.java Animal/*.java
jar cfe 32909764_FieryDragonGame.jar GameBoard *.class Animal/*.class

Class diagram



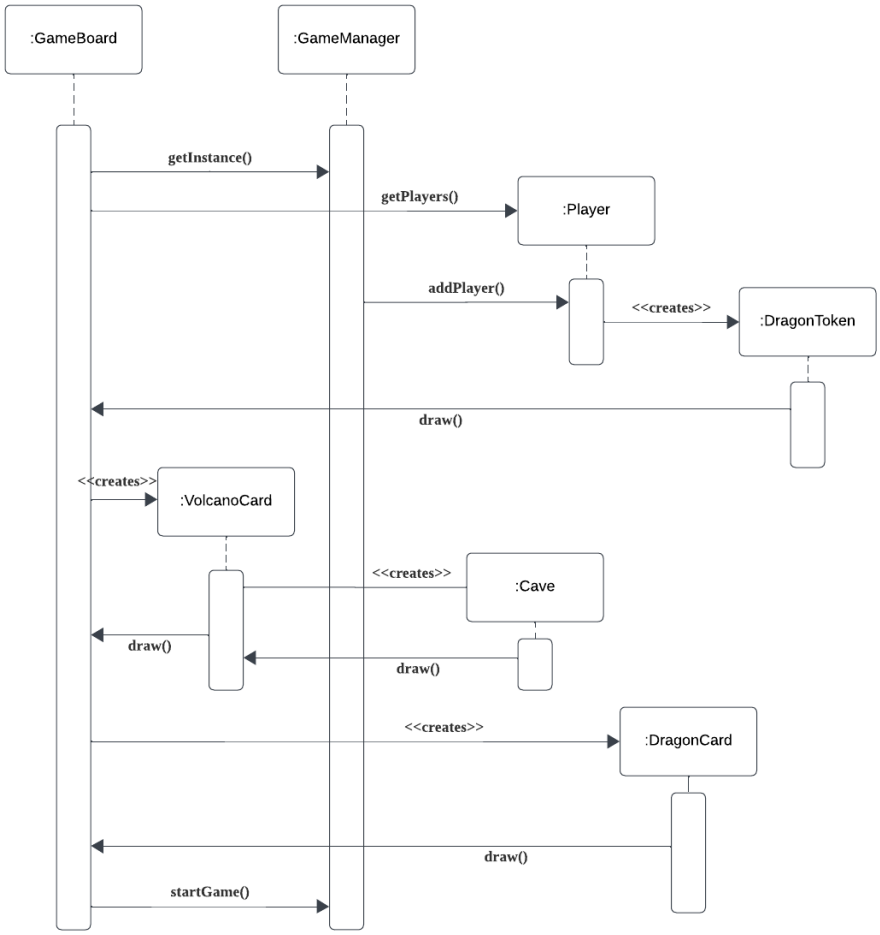
In the class diagram, it is important to note that some getter and setter methods are not included. These methods are commonly used for accessing and modifying the attributes of classes, but for simplicity and clarity, I removed it from the diagram.

Additionally, after implementing the code, there were a few slight changes made to the UML class diagram. These changes reflect adjustments and refinements to the structure and relationships of the classes based on the actual implementation.

It is also worth mentioning that certain classes, such as JPanel, JFrame, and draw functions, were not included in this UML class diagram. As the focus of the diagram is primarily on representing the game flow and core functionality of the game. Including these classes and functions might clutter the diagram and distract from its main purpose, which is to provide a clear overview of the game's structure and interactions.

Summary of key game functionalities

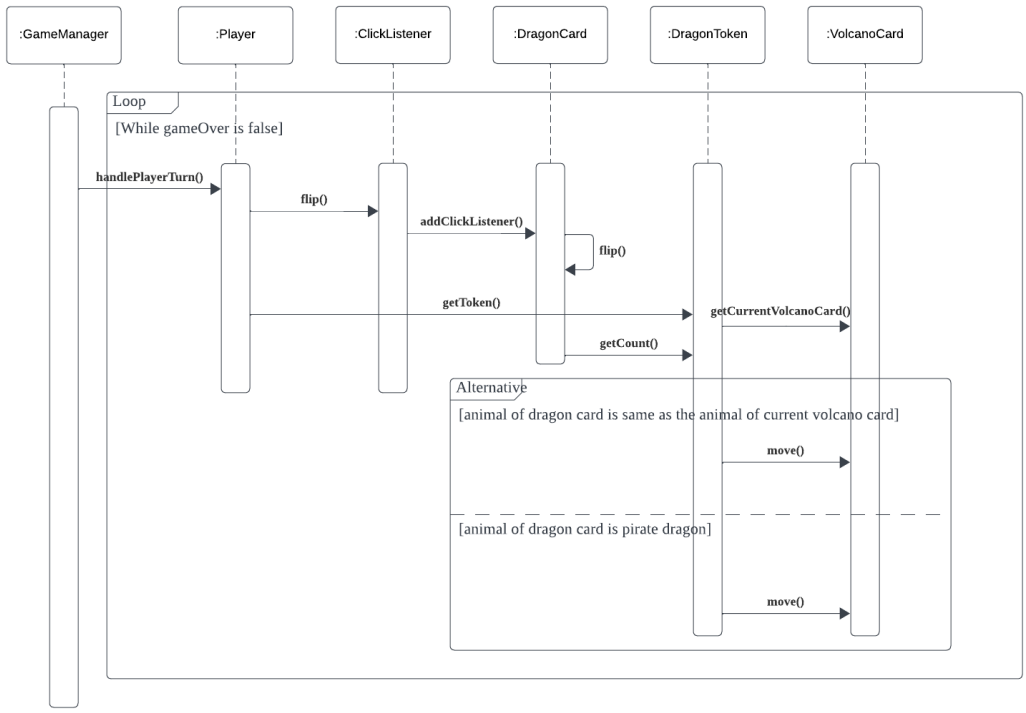
(i) set up the initial game board (including randomised positioning for dragon cards)



In the set-up of the initial game board, the GameBoard first ensures there's only one GameManager to manage the game. Then, it sets up the main game components which are VolcanoCards and DragonCards. Additionally, it gets the players and each of the player initializes their own token. Some VolcanoCards also create attached Caves. Once all these

elements are set, the GameBoard initiates the game by calling the GameManager to start the gameplay.

(iii) movement of dragon tokens based on their current position as well as the last flipped dragon card



Once the GameManager's startGame function is invoke, it continuously triggers the handleTurnPlayer function until one player wins the game. During each player's turn, the ClickListener becomes active, allowing players to identify the dragon card they flip. Subsequently, the player's token is checked to determine its current volcano card. The dragon card's animal count is then retrieved. If the animal on the flipped card matches that of the current volcano card, the token moves accordingly. However, if a pirate dragon is flipped, the token moves by a count of -1. In my codes, I combined the move forward and backward action together by setting negative values for moving backward, hence it was able to calculate the correct next volcano card after flipping the dragon card.

Design Rationale

Animal interface class

Issue:
We need to define a structure for representing different types of animals in our game. These animal types share no common behaviour or properties other than their names. Therefore, we need to decide between using an interface or an abstract class to define this structure.
Alternative A: Animal Interface Class
Advantages:
- Provides a contract for implementing classes to follow to without any shared implementation
- Enables flexibility in implementing various animal types with different names

Disadvantages:
- Cannot contain any implementation details, including default behaviour or shared code
- Cannot define any state or common properties for the animal types

Alternative B: Animal abstract class
Advantages:
- Allows for providing default behaviour or shared implementation for animal types
- Enables the definition of common properties or methods shared among different animal types

Disadvantages:
- May introduce unnecessary complexity if no shared behaviour or properties are needed
- Could lead to tight coupling between subclasses and the abstract class if not designed carefully

Decision:
Considering that the animal types in our game do not have any shared behaviour or properties beyond their names, and our primary objective is to define a contract for these types, I have decided to implement the Animal class as an interface. This choice aligns with our requirement of providing a contract for implementing classes without introducing any default behaviour or shared implementation. By using an interface, I ensure that each animal type can be represented independently, with no unnecessary coupling or complexity introduced by shared implementation details. This decision promotes flexibility, maintainability, and adherence to the principle of defining the simplest possible solution. It also adheres to the Open Closed Principle by allowing the extension of new types of animals without modifying existing code.

<u>Player class</u>
Issue:
In our game design, we need to determine how to represent player actions such as flipping dragon cards and owning tokens. This decision will impact the organization and structure of my codebase.
Alternative A: Introducing a player class
Advantages:
- Centralizes player-related actions and responsibilities within a dedicated Player class, promoting clarity in the codebase

Disadvantages:
- May introduce additional complexity if player-related actions are closely tied to other game components, requiring careful coordination and communication between classes

Alternative B: Distributing methods across other classes
Advantages:
- Reducing the size and complexity of individual classes by distributing responsibilities across multiple classes

Disadvantages:

- Could lead to scattered and fragmented code, making it difficult to understand and maintain player-related functionality

Decision:

I have chosen introducing a Player class to represent player actions and ownership of tokens. By centralizing player-related functionality within a dedicated class, we can enhance clarity and encapsulation. The Player class serves as a clear abstraction for player behaviour, encapsulating actions such as flipping dragon cards and owning tokens. This approach facilitates easier maintenance and future enhancements, as player-related functionality is neatly organized and separated from other game components.

GameManager class

Issue:

We need to design a component responsible for managing the game state, including the start of the game, player's turn, and end of the game. This component which is GameManager, must ensure that the game flows smoothly and that players' turns are managed appropriately.

Alternative A: GameManager as a Singleton class

Advantages:

- Ensures that only one instance of the GameManager exists throughout the game session
- Provides a global access point to the GameManager instance, facilitating consistent game state management
- Prevents conflicts that may arise from multiple GameManager instances attempting to manage the game state concurrently

Disadvantages:

- May introduce tight coupling between GameManager and other components if not designed carefully
- Could lead to issues with testability if dependencies on the GameManager are not adequately managed

Alternative B: Regular GameManager class

Advantages:

- Promotes flexibility and testability by allowing multiple instances of the GameManager to be created as needed
- Reduces coupling between GameManager and other components, making the codebase more modular and easier to maintain

Disadvantages:

- Requires careful management of GameManager instances to ensure consistent game state management
- May lead to potential issues with maintaining global game state if not managed properly
- Could introduce complexity in managing the lifecycle of GameManager instances

Decision:

After evaluating the alternatives, we have decided to implement the GameManager class as a Singleton. This decision aligns with my requirement for ensuring consistent game state management for managing the game flow. While the Singleton pattern has its drawbacks, such as potential issues with testability and tight coupling, I believe that these concerns can be mitigated through careful design and implementation. By using a Singleton GameManager, we ensure that the game state is managed effectively and that conflicts arising from multiple GameManager instances will be avoided. Additionally, the Singleton pattern simplifies access to the GameManager instance throughout the game making it easier to use.

GameBoard relationship with DragonCard and VolcanoCard

Issue:

In our class diagram, I need to define the relationship between the GameBoard class with the VolcanoCard and DragonCard classes. This relationship will determine how these components are associated with the game board and how their lifecycles are managed.

Alternative A: Aggregation relationship
Advantages:

- Indicates that VolcanoCard and DragonCard objects are part of the GameBoard's structure but can exist independently
- Provides flexibility in managing the lifecycle of VolcanoCard and DragonCard objects, as they can be created and destroyed independently of the GameBoard

Disadvantages:

- May introduce complexity in managing the association between the GameBoard with VolcanoCard and DragonCard classes

Alternative B: Composition relationship
Advantages:

- Ensures that VolcanoCard and DragonCard objects are created and destroyed with the GameBoard which simplifies lifecycle management
- Existence of the cards is dependent on the existence of the game board

Disadvantages:

- Limiting flexibility in reusing VolcanoCard and DragonCard objects in other contexts
- May introduce unnecessary coupling between the GameBoard with VolcanoCard and DragonCard classes

Decision:
After evaluating the alternatives, I have an aggregation relationship between the GameBoard with VolcanoCard and DragonCard classes. This decision aligns with my design requirements, as it accurately represents the relationship between the game board and the cards while allowing for flexibility in managing their lifecycles. By using aggregation, we indicate that VolcanoCard and DragonCard objects are part of the GameBoard's structure but can exist independently. This approach promotes modularity and reusability by decoupling the lifecycle of the cards from that of the game board. Additionally, it ensures that VolcanoCard and DragonCard objects can be managed independently of the GameBoard when needed.

<u>Player relationship with DragonToken</u>
Issue:
In our class diagram, we need to define the relationship between the Player class and the DragonToken class. This relationship will determine how players are associated with their respective dragon tokens.

Alternative A: Aggregation relationship
Advantages:

- Allows for flexibility in managing the association between players and dragon tokens, as tokens can be shared or exchanged among players

Disadvantages:

- May introduce confusion regarding ownership of dragon tokens, as aggregation does not imply exclusive ownership

Alternative B: Composition relationship
Advantages:

- Ensures clear ownership semantics, with each player responsible for the creation and destruction of their token

Disadvantages:

- Limiting flexibility in scenarios where players may need to share or exchange tokens

Decision:
I have chosen a composition relationship between the Player and DragonToken class. This decision aligns with my design requirements, as it accurately represents the ownership of dragon tokens by individual players. By using composition, we indicate that each player exclusively owns their own token, simplifying ownership semantics and ensuring clear responsibility for managing the lifecycle of tokens. This approach promotes clarity and reduces confusion regarding token ownership, enhancing the maintainability of the design. Additionally, it aligns with the game's concept of players possessing their own tokens throughout the game session, supporting a more coherent gameplay experience.

Cardinality between VolcanoCard and Cave

Issue:
In our class diagram, we need to specify the cardinality between the VolcanoCard and Cave classes, indicating how many caves are associated with each volcano card. This cardinality will define the relationship between the two classes.

Alternative A: VolcanoCard 1 to Cave 0..1
Advantages:
- Allows for flexibility in associating volcano cards with caves, as not all volcano cards may have a corresponding cave

Disadvantages:
- Requires careful management to ensure consistency and coherence in associating volcano cards with caves

Alternative B: VolcanoCard 1 to Cave 1
Advantages:
- Provides a clear and straightforward association between each volcano card and exactly one cave, simplifying game mechanics and rules.

Disadvantages:
- May require additional rules or exceptions to handle scenarios where a volcano card does not have a corresponding cave

Decision:
I have chosen a cardinality of 1 to 0..1 between the VolcanoCard and Cave classes. This decision aligns with my design requirements, as it allows for flexibility in associating volcano cards with caves. By using a cardinality of 1 to 0..1, we indicate that each volcano card may or may not have a corresponding cave, accommodating scenarios where only few caves for each player. This approach promotes flexibility in game design and avoids unnecessary restrictions on the association between volcano cards and caves.

Cardinality between GameManager and Player

Issue:
In our class diagram, we need to define the cardinality between the GameManager class and the Player class, indicating how many players can participate in a game. This cardinality will determine the number of players allowed in a game session.

Alternative A: GameManager 1 to Player 2..4
Advantages:
- Allows flexibility in the number of players that can participate in a game session

Disadvantages:

- Requires careful coordination with other game components, may introduce complexity in managing game state

Alternative B: GameManager 1 to Player 4

Advantages:

- Simplifies game mechanics by enforcing a fixed number of players for each game session, reducing complexity in managing game state and interaction

Disadvantages:

- Limits flexibility in accommodating different group sizes

Decision:

I have chosen a cardinality of 1 to 2..4 between the GameManager and Player classes. This decision aligns with my design requirements, as it allows users to choose the number of players for the game, providing flexibility and customization options. By using a cardinality of 1 to 2..4, we indicate that each game manager can host a game session with 2 to 4 players, accommodating different group sizes and preferences. This approach promotes a more dynamic and engaging gameplay experience. Additionally, it supports our goal of providing players with options for both small and large game sessions.

Design Patterns

Factory Method Pattern

The factory method pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. It defines an abstract method for creating objects, which subclasses can override to produce objects of different types without changing the overall structure of the code. Factory Method Pattern encapsulates object creation logic, making it easier to maintain and extend. It also promotes code reusability by allowing subclasses to define their own object creation logic.

In the Fiery Dragon game, the Factory Method Pattern is utilized to create different types of animal objects, such as Baby Dragon, Bat, Salamander, and Spider. Each of these animal types implements the Animal interface class, acting as a factory for creating instances of that specific animal. By defining a factory method within the Animal interface and implementing it in each subclass, the game allows for the instantiation of objects of different animal types without modifying existing code hence this supports the open-closed principle.

Iterator Pattern

The Iterator pattern is a behavioural design pattern that provides a way to access elements of a collection sequentially without exposing the underlying representation of the collection. It defines an interface for accessing elements one by one and maintains the current position within the collection. The iterator encapsulates the traversal logic and data access methods, hiding the details of how the collection is structured and accessed.

Regarding the planned use of the Iterator Pattern in the Fiery Dragon game, it was initially intended to iterate over the VolcanoCards when moving the DragonToken on the VolcanoCards after flipping the DragonCard. However, it was determined that the Iterator Pattern is not suitable for this scenario because the movement of the DragonToken depends on the number of counts in the DragonCard, rather than a sequential traversal of the VolcanoCards. Therefore, an alternative approach, such as direct calculation based on the count in the

DragonCard, would be more appropriate for determining the next position of the DragonToken on the game board.

Singleton Pattern

The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It involves creating a class with a method that returns the same instance of the class every time it is called, ensuring that there is only one instance of the class throughout the application's lifecycle.

In the Fiery Dragon game, the Singleton Pattern is used in the GameManager class to ensure that only one GameManager is instantiated. This class is responsible for managing the flow of the game, coordinating interactions between different game components, handling game events, and maintaining the overall state of the game. By restricting the GameManager to a single instance, we can avoid inconsistencies and conflicts that may arise from multiple instances attempting to manage the game simultaneously. Additionally, using a Singleton for the GameManager simplifies access to game-related functionalities and promotes a more organized and centralized approach to game management.