MONASH University

FIT3077 Software engineering: Architecture and design

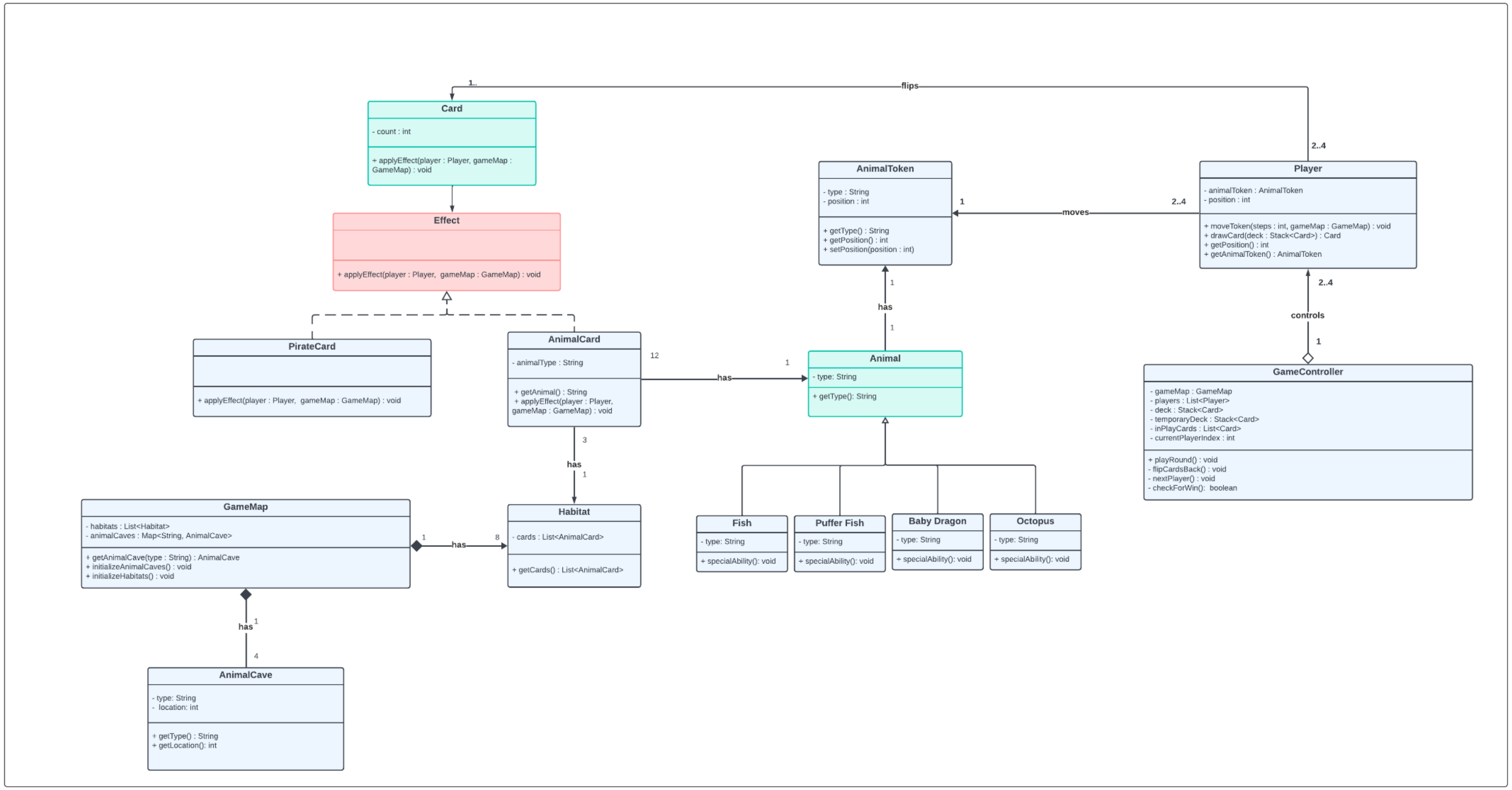
Sprint 2

Student Name: Racheal Lim

Student ID: 33354308

Table of Contents

1.0 UML Class Diagram



1.1 Identify 2 Key Classes

1.1.1 GameMap

The GameMap serves as the primary controller for every element that forms the game's physical design, including the animal caves and habitats which complies with the Single Responsibility Principle. It facilitates a streamlined and centralised management system by supervising the arrangement and interaction among different components to form the game board. The GameMap takes care of configuring the game board with functions like initializeAnimalCaves() and initializeHabitats(). To ensure proper game setup, this might include using complicated logic to distribute animal cards throughout habitats. The GameMap class may conveniently encapsulate the extra techniques required to maintain track of the interdependencies and execution order of these independent functions.

The GameMap offers a scalable method to add new features in the event that the game's complexity increases (if there were extensions to be implemented later on). The GameMap class, for instance, might be expanded without changing the fundamental principles of the game if new card kinds or regulations impacting the game board were added. The GameMap class properly encapsulates habitats and animal caves, rather than having them as global variables. Encapsulating the game state to prevent unauthorised access and alteration is a fundamental tenet of object-oriented design.

Interaction with the game board is facilitated via methods in the GameMap. For example, getAnimalCave(type) offers a simple method to retrieve an animal cave according to its type. Managing such interactions with dispersed functions or data structures would be more difficult and less obvious.

If it were implemented as a standalone method, the same code would have to be duplicated because there wouldn't be a central class to handle the game map. This goes against the DRY (Don't Repeat Yourself) philosophy and increases the likelihood of errors because every modification must be repeated several times.

1.1.2 Player

Every Player instance, including an animal token and position, constitutes a unique entity with its own state. A class may encapsulate and keep state throughout the life cycle of an instance, but methods cannot maintain state between calls without the assistance of external state management. Since Player is a class, it may be instantiated several times to produce players, each with unique behaviours and states. To differentiate between various player states, methods alone would need further mechanisms, which would result in more complicated and less reusable code.

The game map and other players are two examples of other elements that players frequently need to engage with. The GameController and other classes can call the Player class's methods to handle these interactions. There would not be the same degree of clarity or user-friendliness with a standalone method to player functions. In order to provide regulated access to its fields, a class may have methods for getting and setting its internal state. As the game proceeds, the Player class, for instance, has methods to update the player's location and the animal token they control with the getAnimalToken() and getPosition() functions.

1.2 Identify 2 Key Relationships

1.2.1 Aggregation (GameController and Player)

In my designed class diagram, the list of players in the diagram indicates that the GameController class maintains track of the players in a game. Its aggregate, however, suggests that the GameController does not have sole control over how Player instances are arranged and used during the game, which is consistent with the flexible nature of many gaming systems in which players enter and quit games. The main interaction between the player and the game controller is seen from this aggregation cardinality. The game controller manages the game's progression and gives the players instructions on when to move tokens and draw cards, among other activities. It does not, however, control the participants' own existence, which is characteristic of an aggregate connection.

The GameController may not be the exclusive owner of the Player instances. It is possible for players to participate in more than one game, or for their data to be kept up to date outside of a particular game session, indicating that their lifespan is not tightly tied to the GameController.

Aggregation shows that the existence of Player instances is not exclusively under the control of the GameController. In composition, if the container (GameController) is destroyed, the enclosed object (Player) will also be destroyed. Nevertheless, Player instances can survive the GameController through aggregation. Gamers can be formed before or after a game session begins, and they can exist without a gaming controller. In contrast to composition, which would imply that Player objects are generated and destroyed in tandem with the GameController, this independence of lifetime suggests a weaker link. It is possible for various games or GameController instances to use the same Player objects. Since players often join and exit games on gaming platforms, composition relationships are inappropriate to apply in this situation.

1.2.2 Composition (GameMap and Habitat)

GameMap has a strong ownership over Habitat. In the game, a habitat does not make sense without the game map it belongs to. The habitat's lifecycle is closely tied to the lifecycle of the game map as habitat acts as the game board and exists to form the game map. The '8' next to the composition diamond indicates the number of habitats that make up a game map, and the '1' next to the habitat class indicates that each habitat belongs to a single game map. This makes sense given that there are three animal cards in each set of habitats and that a game map is made up of eight sets of habitats. The sense of composition is further reinforced by this multiplicity, which suggests that the habitats are parts that come together to make a whole GameMap.

A game map's habitats are an essential component whose existence is constrained by the game map itself. The habitats are constructed and initialised at the same time as the game map. The ecosystems vanish along with the game map. This illustrates the 'death' connection in composition, in which the game map is not more durable than its constituent elements (habitats). Every Habitat is exclusive to a single GameMap; it is not shared with any other instances or system components. This exclusivity is a feature of composition as opposed to aggregation, in which several owners may share the pieces.Habitats are fundamental components of a game map in terms of the game logic; that is, they are necessary for the definition and functionality of the map.

This is why composition is used rather than aggregation; the habitats are not standalone elements that can be detached and reattached to different game maps. They are created for and exist solely within the scope of their game map.

1.3 Identify Inheritance

1.3.1 Inheritance (Card & Animal)

In my UML diagram, inheritance is represented by the generalization relationship—a line with a hollow arrowhead pointing from the subclasses to the superclass (Animal & Class). This notation visually conveys the idea that subclasses are all types of the abstract class.

Inheritance is the link between the Card class and its subclasses, PirateCard and AnimalCard. The triangle arrowhead pointing at the Effect interface class that directs to the Card class illustrates this. ApplyEffect() is a method in the abstract Card class while it implementing the Effect interface. The subclasses will give their own implementations of this function, which is meant to be overridden by them. Given that both AnimalCards and PirateCards are specialised card types with shared characteristics and behaviour described in the card, inheritance makes sense in this situation. Every subclass, however, also possesses extra characteristics or traits specific to its kind; for example, AnimalCard has an animalType that is associated to the player's movement with the ApplyEffect() method, while PirateCard have special impacts on the status of the game (backward movement).

Using inheritance allows for polymorphic behavior, where a method can invoke applyEffect() on a Card object without needing to know if it's an AnimalCard or PirateCard. This polymorphism simplifies code that works with cards and enables the addition of new card types in the future without modifying code that operates on the base Card type.

The use of Animal as an abstract class with specialized subclasses like Fish, Puffer Fish, Baby Dragon, and Octopus is an example of inheritance too in my designed class diagram. All of the subclasses do not need to repeat this code because the abstract Animal class has common methods and properties (such move(), specialAbility(), and type). This makes maintenance easier and cuts down on redundancy. The game logic may handle many animal kinds in a general way by treating the subclasses as their superclass type. Polymorphic behaviour is made possible, for instance, by the fact that a list of Animal can contain instances of Fish, Puffer Fish, and other subclasses.

It's considerably simpler to introduce new animal species and modify the behaviour of current ones as necessary. Subclasses can be added or changed without impacting other system components. This is essential to the game's scalability going forward.

1.4 Cardinality

The cardinality between Player and AnimalToken indicates that a single game may be linked to a minimum of two and a maximum of four Player instances at any given moment. It corresponds with the game's regulations, which permit two to four players according to the text. As a result, the game is meant to be played by two players minimum and four players maximum. The ‘1’ next to AnimalToken also shows that there is precisely one AnimalToken associated with each Player instance. The cardinality is precisely "1" since there isn't a situation in the game when a player possesses neither no animal token nor more than one. In this scenario, using "1..2" would go against the game's regulations, which stipulate that there must be a minimum of two players. Instead, it would indicate that there may be one or two Player instances linked with something. The game could only be played with one player or up to two players if you had a cardinality of "1..2" for Player, which is against the established regulations.

The cardinality between AnimalCard and GameMap indicates that there are precisely 3 AnimalCard instances connected to each Habitat instance. This cardinality accurately depicts that each habitat will always have three animal cards and no more or less, given the rules of our game, where a habitat consists of precisely three animal cards. The ‘1’ relationship next to habitat shows that each AnimalCard is associated with one and only one Habitat. In other words, an animal card cannot belong to more than one habitat at a time.

The cardinality is set by the game's regulations. It should be noted that every habitat consists of precisely three animal cards; a game map consists of eight sets of these habitats. This one-to-many link between a single habitat and the three animal cards it holds is reflected in the "3 to 1" cardinality.

It's crucial to understand that cardinalities are per-instance connections rather than totals across the course of the programme. Therefore, even though there are a total of 24 animal cards on the game map, "3 to 1" refers to the cardinality between a single habitat and its animal cards.

1.5 Design Patterns

1.5.1 Strategy Pattern

The Card, PirateCard, AnimalCard, and Effect interfaces in the given diagram are all designed using the Strategy pattern. This pattern becomes clear when I have a family of algorithms that are each implemented by an interface (Effect) and contained in a separate set of classes (PirateCard and AnimalCard). In this design, the strategy interface is the Effect interface. It introduces the applyEffect method, which is the one that changes based on the specific strategy that is used.

The Effect strategy interface is concretely implemented in PirateCard and AnimalCard. For the applyEffect method, PirateCard's applyEffect uses reasoning to shift a player's token backwards. Whereas, AnimalCard's applyEffect advance a player's token or in the later implementations phase could carry out additional actions according to the kind of animal.

With this approach, I may employ multiple card effects interchangeably during the game without the GameController's code changing by implementing the Strategy pattern. The game controller can use applyEffect on the card object when a card is drawn and its effect has to be applied. This works regardless of whether the card is an AnimalCard or a PirateCard; the appropriate action is carried out depending on the class of the real object.

By separating the particular actions from the context in which they are used, this allows for the addition of new card types with various effects without changing the game logic or classes that already exist.

1.5.2 Adapter Pattern (Not used)

In software design, the Adapter pattern is commonly employed to facilitate communication between two incompatible interfaces. It functions as a bridge, enabling the collaboration of two disparate systems that would not otherwise be possible because of incompatible interfaces or techniques. But this pattern isn't applied in our situation because it might not be required.

We are using Java as our coding tools and Java has features like interfaces and inheritance that frequently eliminate the requirement for an adapter. As with my design, two classes may frequently be used interchangeably by simply extending the same abstract class or implementing the same interface. When an adapter is not required, it might cause overcomplication in the design by introducing extra layers of abstraction. It could be ineffective to introduce an adapter if the present system architecture is simple and satisfies all needs without the need for interface compatibility layers.

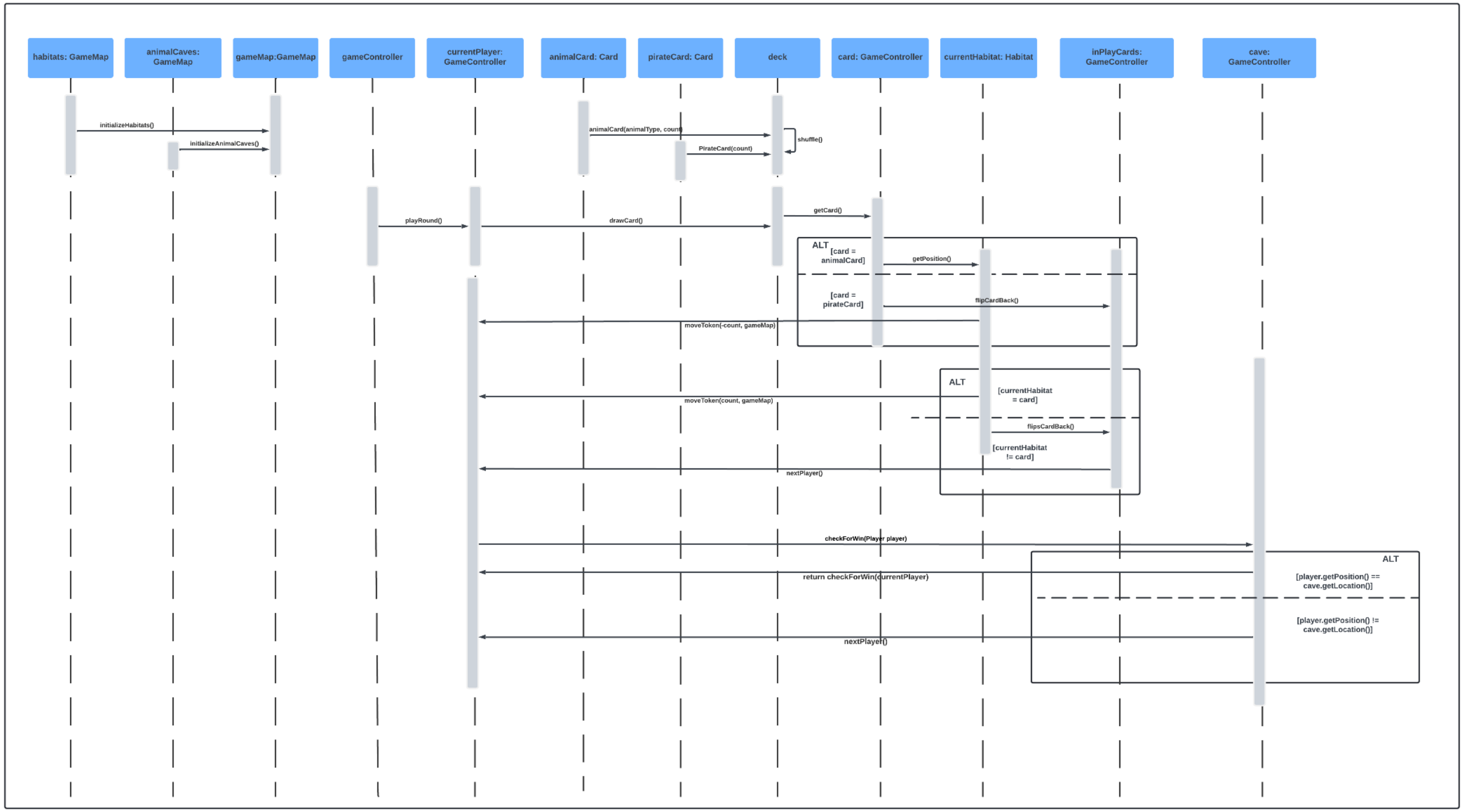
1.5.3 Factory Method (Not used)

The Factory Method pattern lets subclasses modify the kind of objects that are generated while still providing an interface for doing so in a superclass. It's frequently used when a class assigns responsibility to one or more helper subclasses, when a class cannot predict the class of objects it needs to generate, and when subclasses wish to define objects they create.

However, our design creates instances of PirateCard and AnimalCard directly, without the need for encapsulation or complicated logic or different creation methods. The simplicity of the direct instantiation approach eliminates the necessity for a factory method in the game production process. Strong Java features like reflection enable for more flexible object creation than would be possible with a Factory Method approach. The patterns made for language constraints, such Factory Method, would not be the ideal option because our implementation will make use of these capabilities in the next sprints. Additionally, Card objects just need to send arguments to a constructor. Hence setting up a factory might complicate our design needlessly.

2.0 UML Sequence Diagram

The UML Sequence Diagram below shows how each key game functionality is covered



2.1 Setting up initial game board

To illustrate the setup of the GameMap based on the sequence diagram provided in numerical orders:

1. The GameMap class, which is in charge of initialising the game environment, starts the process. To hold the game's caves and habitats, it has two primary objects: animalCaves and habitats.
2. InitializeHabitats() is the method called by the GameMap class to begin the setup process. Using this technique, a variety of habitat items (8 sets in this case) are added to the habitat object. The task of storing a collection of AnimalCard objects falls to each habitat.
3. The GameMap invokes the initializeAnimalCaves() function when the habitats have been initialised. By linking various AnimalCave objects to the appropriate animal categories, this technique populates the animalCaves map.
4. Next, the GameMap is connected to the GameController object, which controls player interactions and the game's flow. To manage the game logic, the GameController will make use of the GameMap.
5. To maintain track of who is taking turns, the GameController uses the currentPlayer as an object to contain a reference to the active player.
6. The number of animals or pirates that each AnimalCard and PirateCard object represents is indicated by the count that is produced and passed. The shared characteristics and actions of many card kinds are encapsulated in the Card base class, of which these cards are instances.
7. All created cards, including AnimalCard and PirateCard objects, are added to the deck. This deck is a object of cards that the players will draw from during the game.
8. Finally, to change the order of the cards, the deck's shuffle() function is used. This guarantees that the initial state of the game is unexpected and equitable for every player.

2.2 Flipping of pirates/animals card (open)

To illustrate the initial flipping card process based on the sequence diagram provided in numerical orders:

1. The playRound() method is called by the GameController to start the player's turn. This technique denotes the beginning of a new game round in which different activities will occur.
2. The GameController accesses the currentPlayer, a reference to the player whose turn is currently in progress, inside playRound().
3. A card is flipped when the drawCard() function on the deck is called by the currentPlayer.
4. The getCard() method is called after flipping a card, indicating that more information about the drawn card, including its kind (PirateCard or AnimalCard), needs to be retrieved to store in the card object

2.3 Flipping of pirates/animals card (closed - Maintaining game state)

Following the draw, there's an alternative (ALT) flow where the type of the card drawn dictates the next steps. Two conditions are checked: if the card object is an animalCard or a pirateCard.

1. If an animalCard is drawn:

   a) The GameController retrieves the player's position with getPosition().
   b) If the currentHabitat matches the card, the game continues without flipping the card back, implying that the player’s turn might continue.
   c) If it doesn't match, flipCardBack() is called, indicating that the card should be turned face down again, and the player’s turn ends.

2. If a pirateCard is drawn:

   a) Then, flipCardBack() is called, indicating that the cards are flipped back to their original position and that any progress achieved during this turn is undone.

2.4 Movement of animal tokens based on their current position on the animal habitat as well as the last flipped animal card.

The getPosition() method is used to find the current position of the player on the game board within the currentHabitat. Following the flipped card, an alternative (ALT) block, which stands for a decision-making stage in the process, is then used to analyse the card type:

1. If an animalCard is drawn, a check is made to see if the currentHabitat matches the card. If it matches (currentHabitat == card), the player moves their token by calling moveToken(count, gameMap).
2. If the card is a pirateCard, the player's token moves backwards on the game board by calling moveToken(-count, gameMap).
3. At the same time, if the ALT block circumstances indicate that the cards should be placed back in their face-down position, then the flipCardBack() function is invoked. This occurs when an animal card that does not correspond with the current environment is drawn, or when a pirate card is drawn.

2.5 Change of turn to next player

The sequence diagram ends with a call to nextPlayer(), indicating that the turn has ended and the next player's turn will begin once conditions are met.

As illustrated in the sequence diagram, nextPlayer() is called when flipCardBack () is being called or when player.getPosition() != cave.getLocation(), indicating player hasnt reach its respective animal token's cave hence round still continues.

2.6 Winning the game

After the move, the game controller checks if the current player's animal token has reached the location (player’s animal’s token cave) that constitutes a win by invoking the checkForWin(player) method.

Following by an ALT block, if [player.getPosition() == cave.getLocation()] evaluates to true, then it will result in the game ending and the player being declared the winner by returning checkForWin(currentPlayer). If the condition is false, nextplayer() is called to continue the game with the next player's turn.