FIT3077 Software Engineering Architecture and Design

Sprint 2 Deliverables

Hang Jui Kai | 32203047

Design Rationale

Key Classes Explanation

In accordance with the class diagram, I've designed few classes for the software to adhere to the guidelines of object-oriented programming. Considering each object has unique state (attributes/ data) and behaviour (methods/ functions). For instance, Square is a class that represent every tile in the volcano card, designating the various animal species, including bats, dragons, salamanders, and spiders. Inspiration for this class's creation was that it may serve as a template for creating objects for various volcano card tiles that have comparable characteristics and behaviours. In contrast to creating it as a method, I believe that creating Square as instances of classes will provide the capacity of reusability, meaning that the object may be used repeatedly for other objects that are comparable. In this case, I can embed three distinct squares, each with its own features, in each volcano card. This implies that as players play the game, the game's algorithm will be able to determine whether the square is stuffed by another token or verifying the species of animals to reveal the next move when player flip the dragon card by invoking methods or accessing necessary attributes. Since most of the instructions needed are inappropriate for the technique, Square is therefore inappropriate as a method. Conversely, Game is defined as a logic class that will manage the majority of routine duties, like initialising players, the game board, and operations to determine the winner of each round. Furthermore, given each utility method in the class has a defined purpose that makes it easier to maintain and understand, it complies with the Single Responsibilities Principle (SRP). It is evident that it is not appropriate to create it as a method given that it requires handling the significant portion of the game's logic with several related methods to accomplish each operation.

Key Relationship Explanation

Several approaches including composition, aggregation, association, and dependency has been utilized to determine the relationships between the classes according to the class diagram. It illustrates a logical relationship or connection between classes, which may be useful in illustrating how different classes relate to and engage with one another. As an instance, the relationship between Game and Game Board is composition. Consider that, as the lifecycles of both classes are closely tied to one another, the relationship between the classes should be strong whole-part relationships. That's due to Game without the Game Board, players are unable to see which square is being occupied or where the token is situated, and the algorithm could not be able to determine the victor if a token has not been admitted into the cave. On the flip side, there is an association between Square and Cave given that Square has the field or attribute Cave from another class, whereas Cave has the identical field or attribute from the Square class. Additionally, as a way to carry out various activities, both classes need to call the methods of other classes. Thus, this indicates that classes interact directly and engage in common behaviours. In summary, the relationship shouldn't be considered dependent due to the fact it shows reliance and indirect influence on a different class without direct instantiation.

Decision around Inheritance Explanation

The vast majority of classes related to actions, such as Move Forward, Move Backward, and End Turn, will inherit from an abstract class called Action. It guarantees that the code can be reused by creating common functionality in a base class and using it in numerous derived classes given that these classes share comparable characteristics and behaviours. Further, it will assist in preventing code duplication within the action-related class. The inclusion of it in the design is justified by the fact that it makes maintenance easier and guarantees uniform behaviour throughout the relevant class. This implies that all derived classes will immediately reflect the changes I make to the base class's functionality. In contrast, there are certain similarities among the characteristics of Player and Token. I have chosen not to use inheritance for these classes, nevertheless. Considering that the characteristics and behaviors of both classes would vary. When eliminating or drastically changing the base class, refactoring it will become more complicated that all derived classes will need to be adjusted. In addition to that, both classes ought to be independent since they can require different procedures at various stages in the game. Consequently, it makes sure the code is well-organized and simpler to maintain should an upgrade be needed in the future.

Cardinalities Explanation

There will only be one Game Board in the Game, based on the cardinalities of the relationship between the Game and the Game Board, states as 1 to 1. Each game cannot have two game boards, as the logic can only execute in one game board, and executing the game in two separate game boards could cause errors in the state or value. In summary, this is preferable that 1 to 1 relationship than 1 to 2 connection that make irrelevant. Beyond from that, the relationships between the token and the player are 1 to 1, not 1 to 2 or 1 to 3, considering each player can only carry one token at a time throughout the game. Being that each token does not move in accordance with the chit card that the player flips, the game can't determine which player will win, thus the player is unable to regulate the two or more tokens.

Design Pattern in the Design

Factory Method Design Pattern

Creational design pattern that emphasizes object creation is called Factory Method. Square and Chit Cards have been designed using this design pattern. It should be used since it retains the logic for creating objects separate from the code which utilizes them. By giving subclasses control over instantiation, it separates the code from the specifics of implementation. Considering the changes to the creation process don't require changes to the code, this will ensure that the code is more adaptable and maintainable. In the event that the game updates and new squares with distinct numbers are needed on each chit card, the new feature can be created as a new concreate implementation without changing the existing code. Thus, it strengthens, separates, and facilitates code extension while adhering to the Open/ Closed Principle. Additionally, both the chit card initialization and square construction methods will be invoked within a designated class, which facilitates code organization and maintenance. Indicate it adhere to Single Responsibility Principle. On top of that, since the specified class will invoke the method without being aware of the specifics of the way objects are formed, it may conceal some implementation details. It will encapsulate the creation logic, making it easier to manage and maintain. In summary, employing this approach will increase flexibility, encourage the reuse of code, and guarantee that object creation will remain separate from the code.

Observer Design Pattern

The behavioral design pattern termed "Observer" aims to make easier for classes to interact and communicate with one another. Inside the system, there is a Subscriber interface class and a Game class that serves as a publisher. Taking into account that the game's user interface requires alter each time a player flips a card and move a token depending on the number of animals are indicated on the chit card. Thus, it ought to be appropriate to apply in this particular situation. Furthermore, it will facilitate flexible connection between subjects and observers, permitting the dynamic addition or removal of observers without altering the subjects. It indicates that a particular subscriber may be removed when it is not required and that the list of subscribers will be reinserted following particular processes, which maintain their adaptability and flexibility to modify activities in conformity with current requirements. In contrast, given a new subscriber can be added without altering the publisher's code, it complies with the Open/Closed Principle. Besides, it will guarantee that modifications made to one object automatically spread to other objects that share interest. Consequently, it fosters modularity and flexibility by separating the sender and receiver. In result, it offers an organized and effective means of managing changes and communication amongst objects, increasing the system's extensibility and maintainability.

Executable File

Executable File Details

Name: FieryDragon.jar
Java JDK: Version 21
JavaFX SDK: Version 22

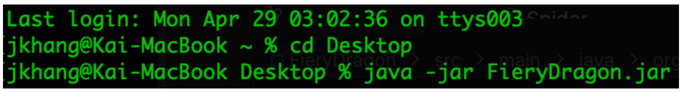
Platform: MacOS
* Ensure that the Java JDK has been upgraded or downgraded to Version 21 of Java JDK and install Version 22 of JavaFX SDK before launching the executable file.

Executable File Location

MA_Tuesday12pm_Team999/Project/Sprint 2/FieryDragon/out/artifacts/FieryDragon_jar/FieryDragon.jar

Run FieryDragon.jar

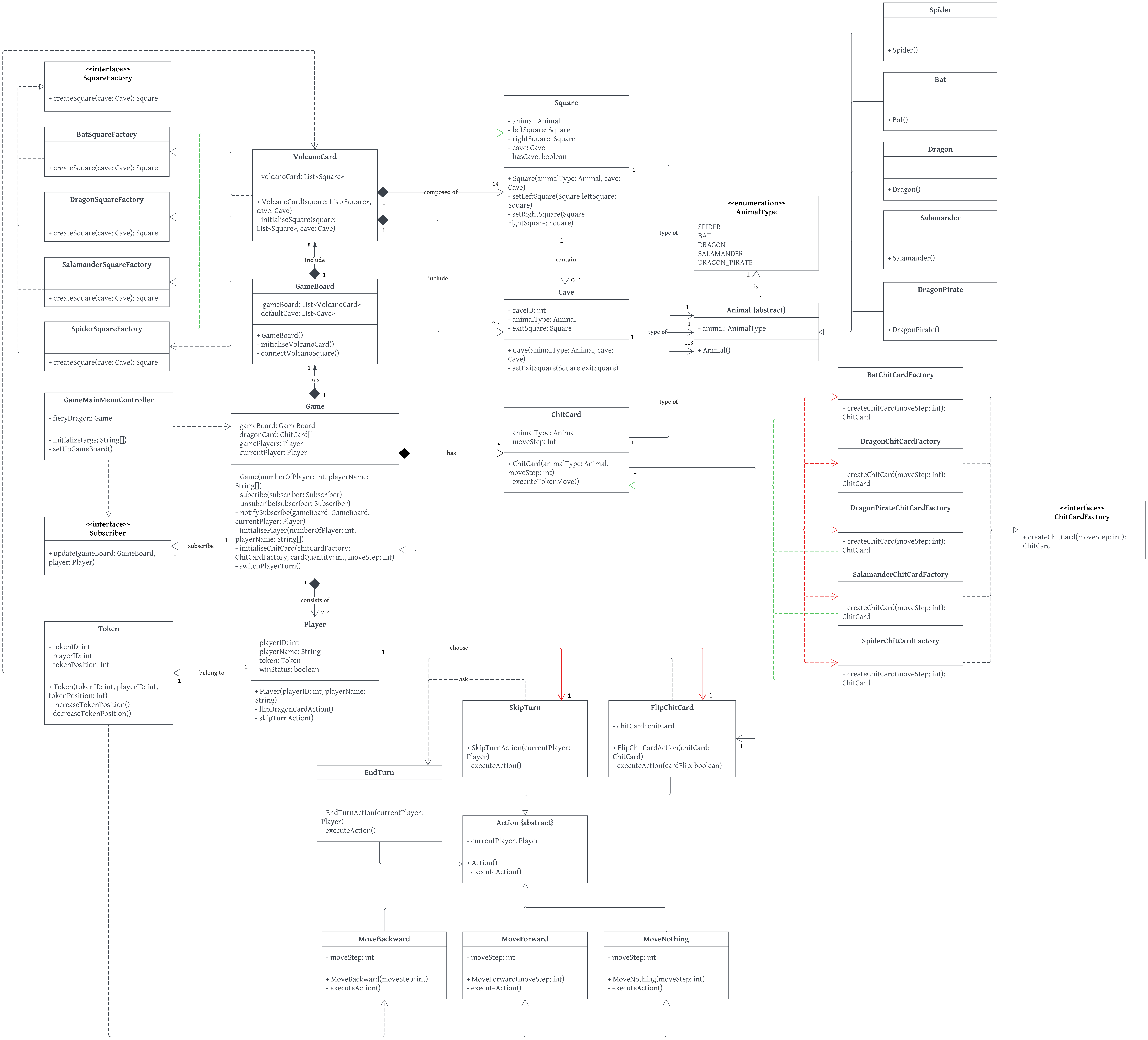
1. Copy the executable file to your designated folder.
2. Open terminal, change the directory to the folder you store in.
3. Run the command: java -jar FieryDragon.jar



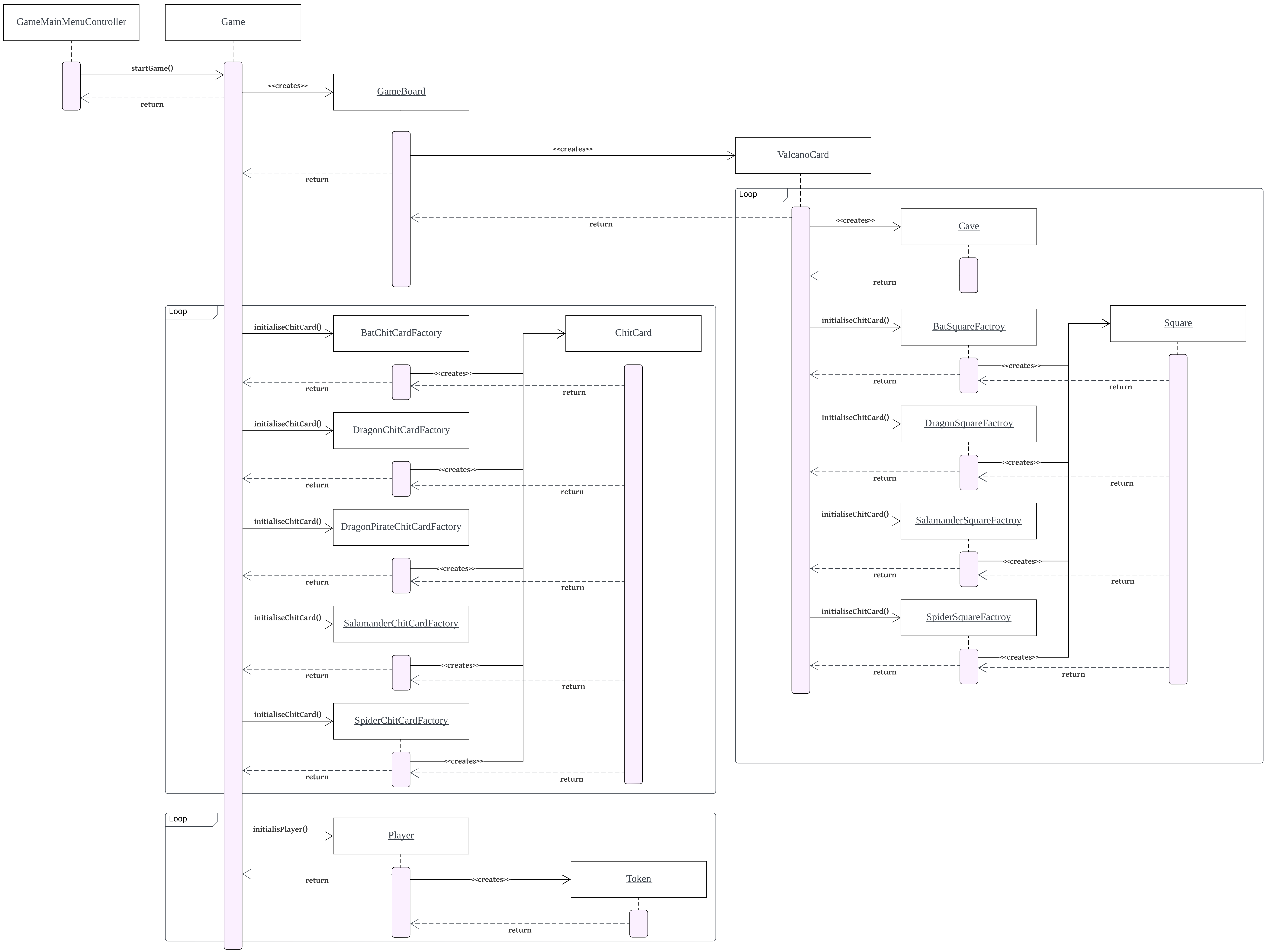
```
Last login: Mon Apr 29 03:02:36 on ttys003
jkhang@Kai-MacBook ~ % cd Desktop
jkhang@Kai-MacBook Desktop % java -jar FieryDragon.jar
```

4. There are two button in the executable file, press Scenario for invoke MockSimulation, then press Run Simulation to demonstrate the feature.

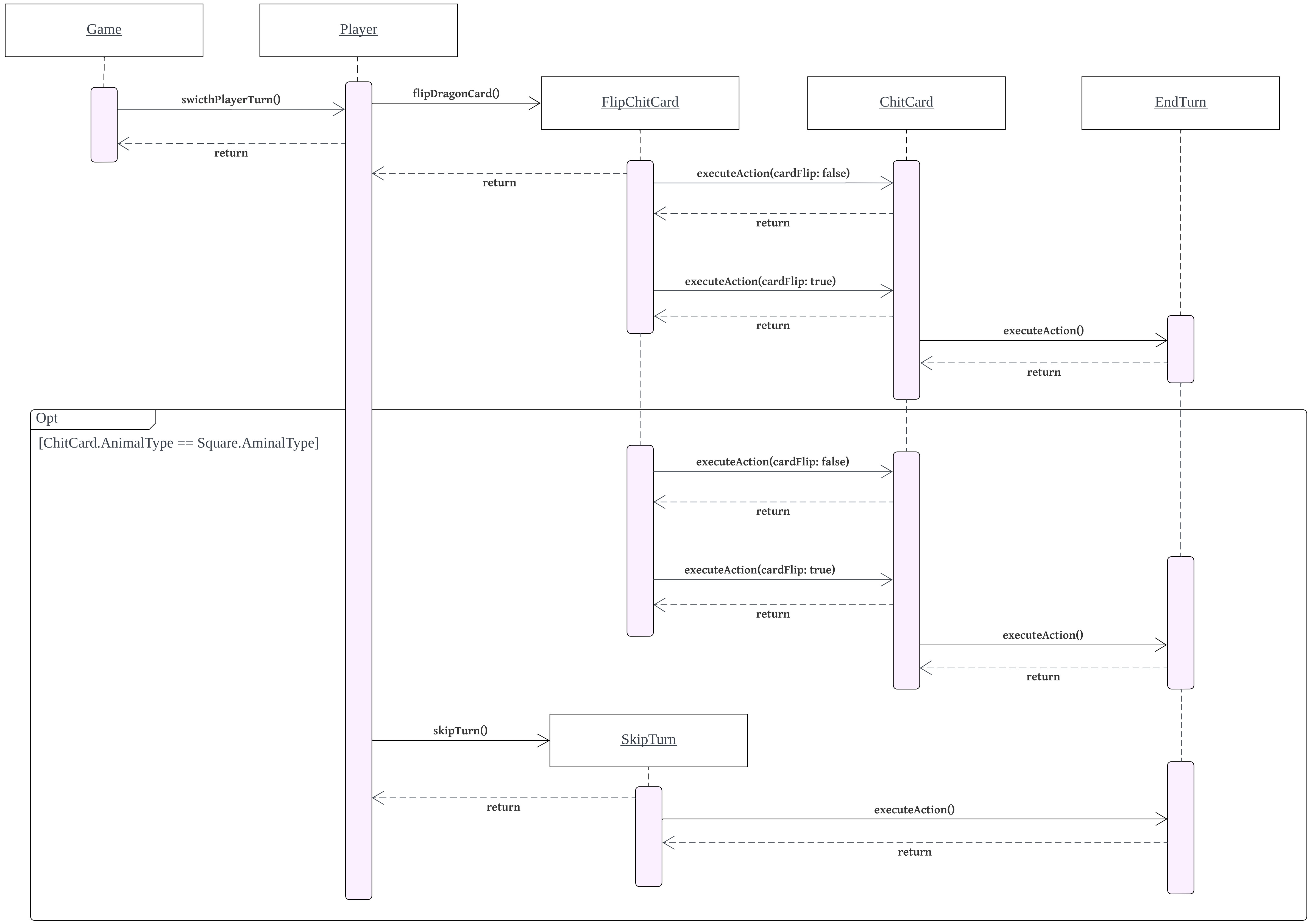
Class Diagram


<<interface>>
SquareFactory
+ createSquare(cave: Cave): Square
BatSquareFactory
+ createSquare(cave: Cave): Square
DragonSquareFactory
+ createSquare(cave: Cave): Square
SalamanderSquareFactory
+ createSquare(cave: Cave): Square
SpiderSquareFactory
+ createSquare(cave: Cave): Square
VolcanoCard
- volcanoCard: List<Square>
+ VolcanoCard(square: List<Square>, cave: Cave)
- initialiseSquare(square: List<Square>, cave: Cave)
composed of
1
24
1
include
2..4
8
include
1
GameBoard
- gameBoard: List<VolcanoCard>
- defaultCave: List<Cave>
+ GameBoard()
- initialiseVolcanoCard()
- connectVolcanoSquare()
1
has
1
Square
- animal: Animal
- leftSquare: Square
- rightSquare: Square
- cave: Cave
- hasCave: boolean
+ Square(animalType: Animal, cave: Cave)
- setLeftSquare(Square leftSquare: Square)
- setRightSquare(Square rightSquare: Square)
1
contain
0..1
Cave
- caveID: int
- animalType: Animal
- exitSquare: Square
+ Cave(animalType: Animal, cave: Cave)
- setExitSquare(Square exitSquare)
1
type of
1
type of
1
1
1..3
type of
<<enumeration>>
AnimalType
SPIDER
BAT
DRAGON
SALAMANDER
DRAGON_PIRATE
1
is
1
Animal {abstract}
- animal: AnimalType
+ Animal()
Spider
+ Spider()
Bat
+ Bat()
Dragon
+ Dragon()
Salamander
+ Salamander()
DragonPirate
+ DragonPirate()
GameMainMenuController
- fieryDragon: Game
- initialize(args: String[])
- setUpGameBoard()
Game
- gameBoard: GameBoard
- dragonCard: ChitCard[]
- gamePlayers: Player[]
- currentPlayer: Player
+ Game(numberOfPlayer: int, playerName: String[])
+ subcribe(subscriber: Subscriber)
+ unsubcribe(subscriber: Subscriber)
+ notifySubscribe(gameBoard: GameBoard, currentPlayer: Player)
- initialisePlayer(numberOfPlayer: int, playerName: String[])
- initialiseChitCard(chitCardFactory: ChitCardFactory, cardQuantity: int, moveStep: int)
- switchPlayerTurn()
1
has
16
ChitCard
- animalType: Animal
- moveStep: int
+ ChitCard(animalType: Animal, moveStep: int)
- executeTokenMove()
1
1
<<interface>>
Subscriber
+ update(gameBoard: GameBoard, player: Player)
1
subscribe
1
1
consists of
2..4
Token
- tokenID: int
- playerID: int
- tokenPosition: int
+ Token(tokenID: int, playerID: int, tokenPosition: int)
- increaseTokenPosition()
- decreaseTokenPosition()
1
belong to
1
Player
- playerID: int
- playerName: String
- token: Token
- winStatus: boolean
+ Player(playerID: int, playerName: String)
- flipDragonCardAction()
- skipTurnAction()
1
choose
ask
1
1
SkipTurn
+ SkipTurnAction(currentPlayer: Player)
- executeAction()
FlipChitCard
- chitCard: chitCard
+ FlipChitCardAction(chitCard: ChitCard)
- executeAction(cardFlip: boolean)
1
EndTurn
+ EndTurnAction(currentPlayer: Player)
- executeAction()
Action {abstract}
- currentPlayer: Player
+ Action()
- executeAction()
MoveBackward
- moveStep: int
+ MoveBackward(moveStep: int)
- executeAction()
MoveForward
- moveStep: int
+ MoveForward(moveStep: int)
- executeAction()
MoveNothing
- moveStep: int
+ MoveNothing(moveStep: int)
- executeAction()
BatChitCardFactory
+ createChitCard(moveStep: int): ChitCard
DragonChitCardFactory
+ createChitCard(moveStep: int): ChitCard
DragonPirateChitCardFactory
+ createChitCard(moveStep: int): ChitCard
SalamanderChitCardFactory
+ createChitCard(moveStep: int): ChitCard
SpiderChitCardFactory
+ createChitCard(moveStep: int): ChitCard
<<interface>>
ChitCardFactory
+ createChitCard(moveStep: int): ChitCard

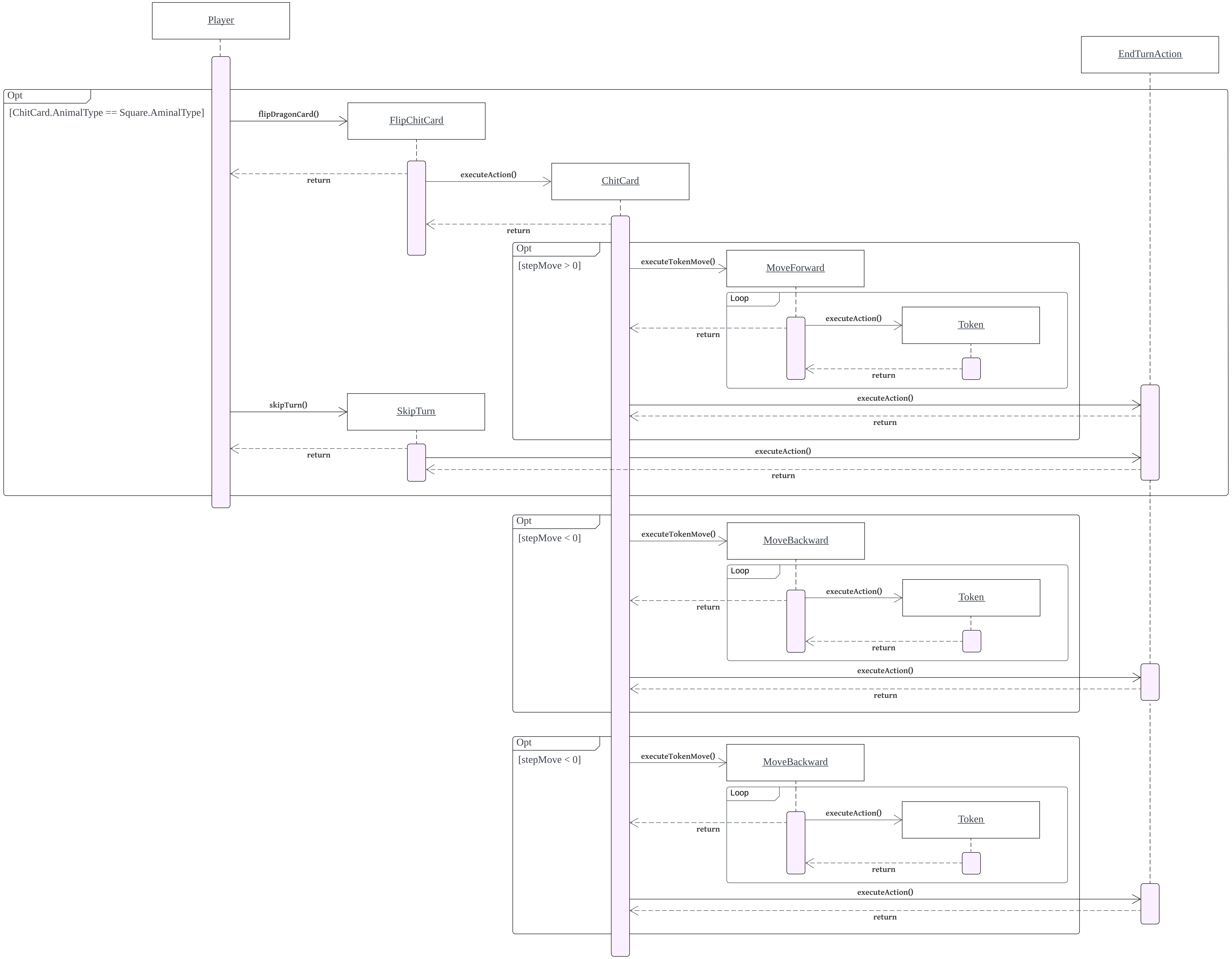
Sequence Diagram I



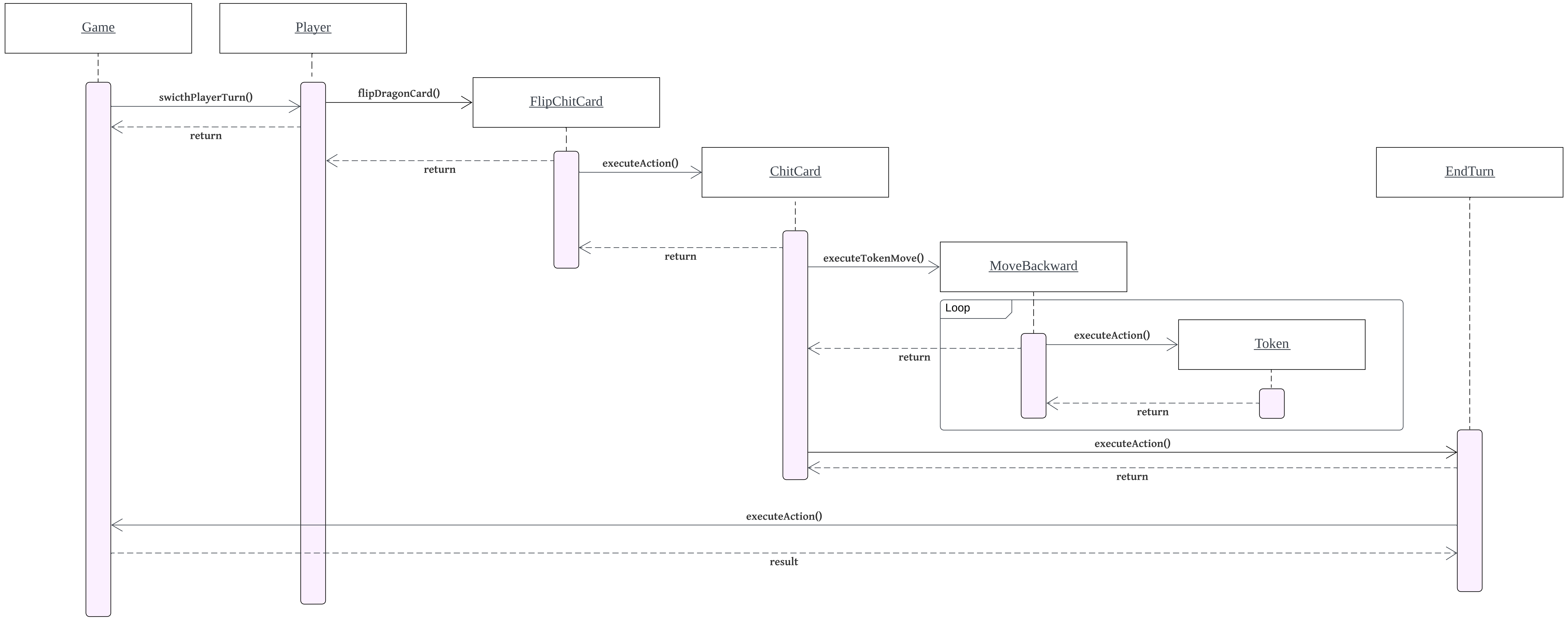
Sequence Diagram II



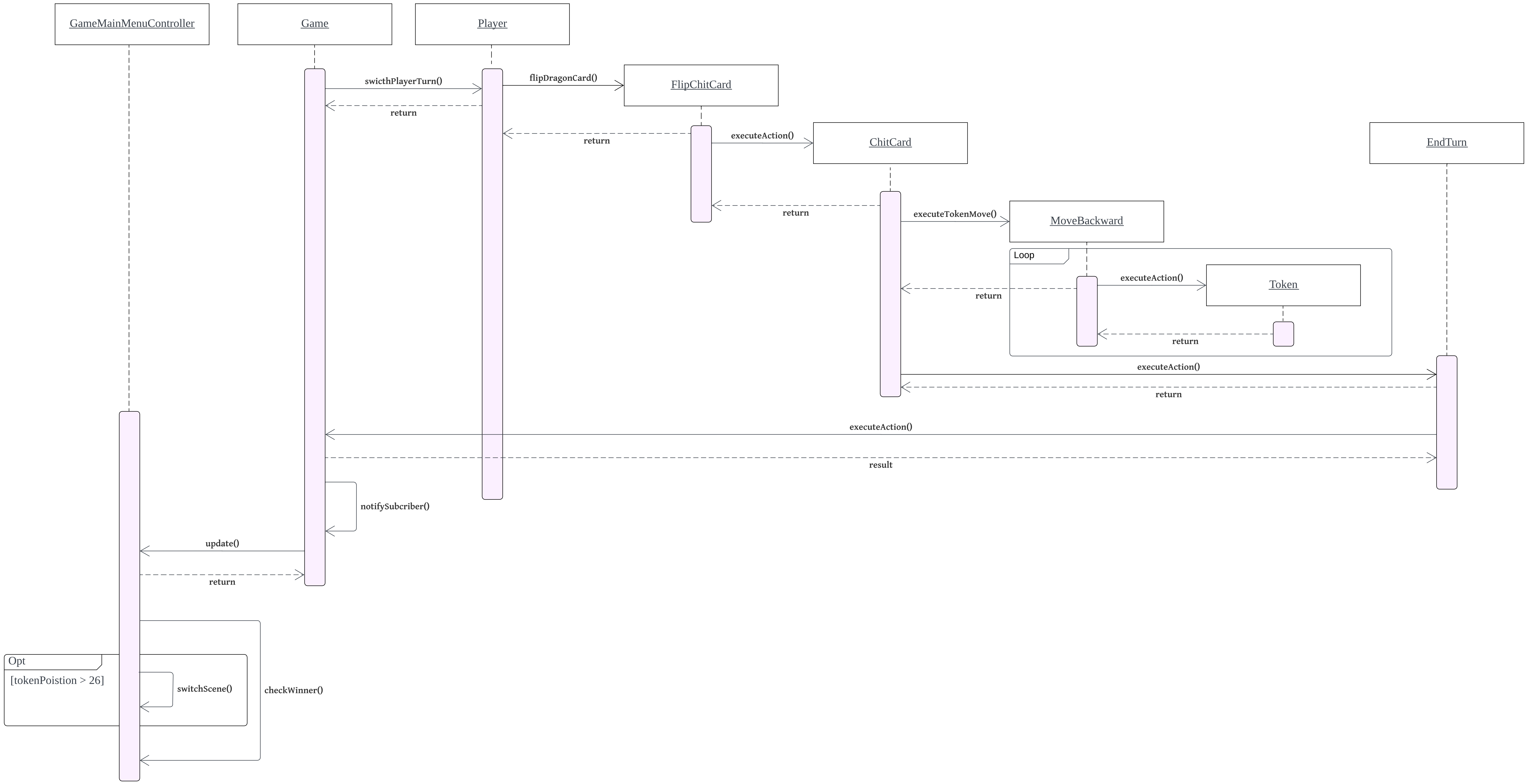
Sequence Diagram III



Sequence Diagram IV



Sequence Diagram V


GameMainMenuController
Game
Player
FlipChitCard
ChitCard
EndTurn
MoveBackward
Token
swicthPlayerTurn()
return
flipDragonCard()
return
executeAction()
return
executeTokenMove()
Loop
executeAction()
return
return
executeAction()
return
executeAction()
result
notifySubcriber()
update()
return
Opt
[tokenPoistion > 26]
switchScene()
checkWinner()