1. Object Oriented Design and Design Rationales

Note for the drawn deliverables:

1) Getters and setters are generally not shown on the class diagram.
2) The main FieryDragonsApplication class extends FXGL's GameApplication class.
3) Not all attributes and methods in the source code are included in the class diagram, as the diagram is only meant to show the design of the system.
4) Javafx scene components (Circle, Rectangle, Text, Group, Image, etc are **not** shown). Only the classes directly involved in the design are included for clarity.
5) The sequence diagrams only show the essential parts for understanding the design, so not all method calls are shown.
6) The sequence diagram for displaying the text message for a player turn is trivial and not shown.
7) Adding components to the UI is done by using FXGL and it is done in two steps:

```
FXGL.spawn("dragonToken", this.spawnData);
```

followed by defining this method

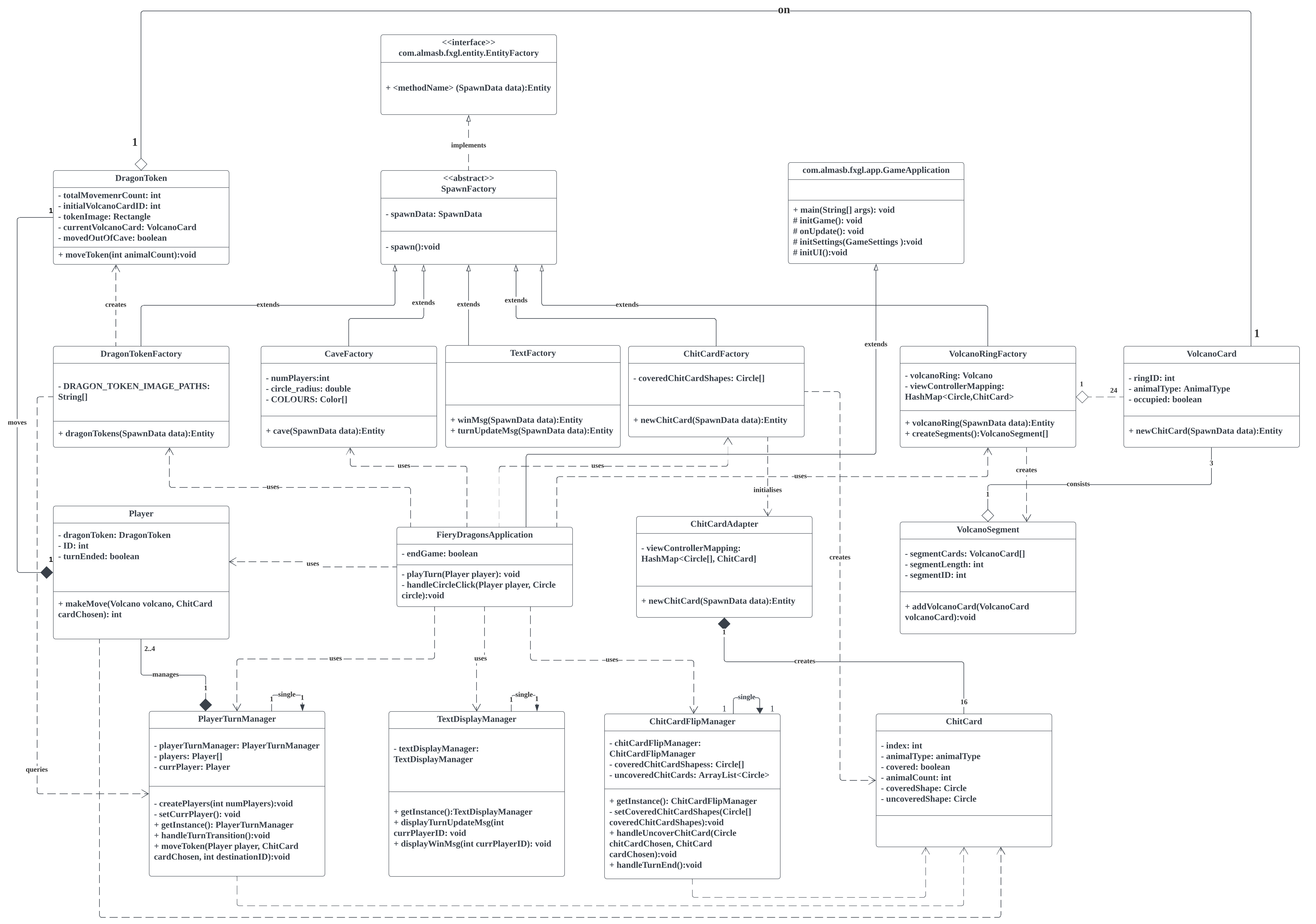
```
@Spawns("dragonToken")
public Entity dragonTokens(SpawnData data)
```

that ends with

```
return FXGL.entityBuilder(data).view(tokenGroup).build();
```

It is within this method the appropriate shapes can be created manually or retrieved from the 'data' payload.

1.1 Class Diagram


on
1
<<interface>>
com.almasb.fxgl.entity.EntityFactory
+ <methodName> (SpawnData data):Entity
implements
<<abstract>>
SpawnFactory
- spawnData: SpawnData
- spawn():void
com.almasb.fxgl.app.GameApplication
+ main(String[] args): void
# initGame(): void
# onUpdate(): void
# initSettings(GameSettings):void
# initUI():void
DragonToken
- totalMovemenrCount: int
- initialVolcanoCardID: int
- tokenImage: Rectangle
- currentVolcanoCard: VolcanoCard
- movedOutOfCave: boolean
+ moveToken(int animalCount):void
1
creates
extends
extends
extends
extends
extends
extends
DragonTokenFactory
- DRAGON_TOKEN_IMAGE_PATHS: String[]
+ dragonTokens(SpawnData data):Entity
CaveFactory
- numPlayers:int
- circle_radius: double
- COLOURS: Color[]
+ cave(SpawnData data):Entity
TextFactory
+ winMsg(SpawnData data):Entity
+ turnUpdateMsg(SpawnData data):Entity
ChitCardFactory
- coveredChitCardShapes: Circle[]
+ newChitCard(SpawnData data):Entity
VolcanoRingFactory
- volcanoRing: Volcano
- viewControllerMapping: HashMap<Circle,ChitCard>
+ volcanoRing(SpawnData data):Entity
+ createSegments():VolcanoSegment[]
1
24
1
VolcanoCard
- ringID: int
- animalType: AnimalType
- occupied: boolean
+ newChitCard(SpawnData data):Entity
moves
uses
uses
uses
uses
initialises
creates
creates
3
consists
1
Player
- dragonToken: DragonToken
- ID: int
- turnEnded: boolean
+ makeMove(Volcano volcano, ChitCard cardChosen): int
1
uses
FieryDragonsApplication
- endGame: boolean
- playTurn(Player player): void
- handleCircleClick(Player player, Circle circle):void
ChitCardAdapter
- viewControllerMapping: HashMap<Circle[], ChitCard]
+ newChitCard(SpawnData data):Entity
1
VolcanoSegment
- segmentCards: VolcanoCard[]
- segmentLength: int
- segmentID: int
+ addVolcanoCard(VolcanoCard volcanoCard):void
2..4
manages
1
uses
uses
uses
creates
16
1
single
1
1
single
1
1
single
1
PlayerTurnManager
- playerTurnManager: PlayerTurnManager
- players: Player[]
- currPlayer: Player
- createPlayers(int numPlayers):void
- setCurrPlayer(): void
+ getInstance(): PlayerTurnManager
+ handleTurnTransition():void
+ moveToken(Player player, ChitCard cardChosen, int destinationID):void
queries
TextDisplayManager
- textDisplayManager: TextDisplayManager
+ getInstance():TextDisplayManager
+ displayTurnUpdateMsg(int currPlayerID): void
+ displayWinMsg(int currPlayerID): void
ChitCardFlipManager
- chitCardFlipManager: ChitCardFlipManager
- coveredChitCardShapess: Circle[]
- uncoveredChitCards: ArrayList<Circle>
+ getInstance(): ChitCardFlipManager
- setCoveredChitCardShapes(Circle[] coveredChitCardShapes):void
+ handleUncoverChitCard(Circle chitCardChosen, ChitCard cardChosen):void
+ handleTurnEnd():void
ChitCard
- index: int
- animalType: animalType
- covered: boolean
- animalCount: int
- coveredShape: Circle
- uncoveredShape: Circle

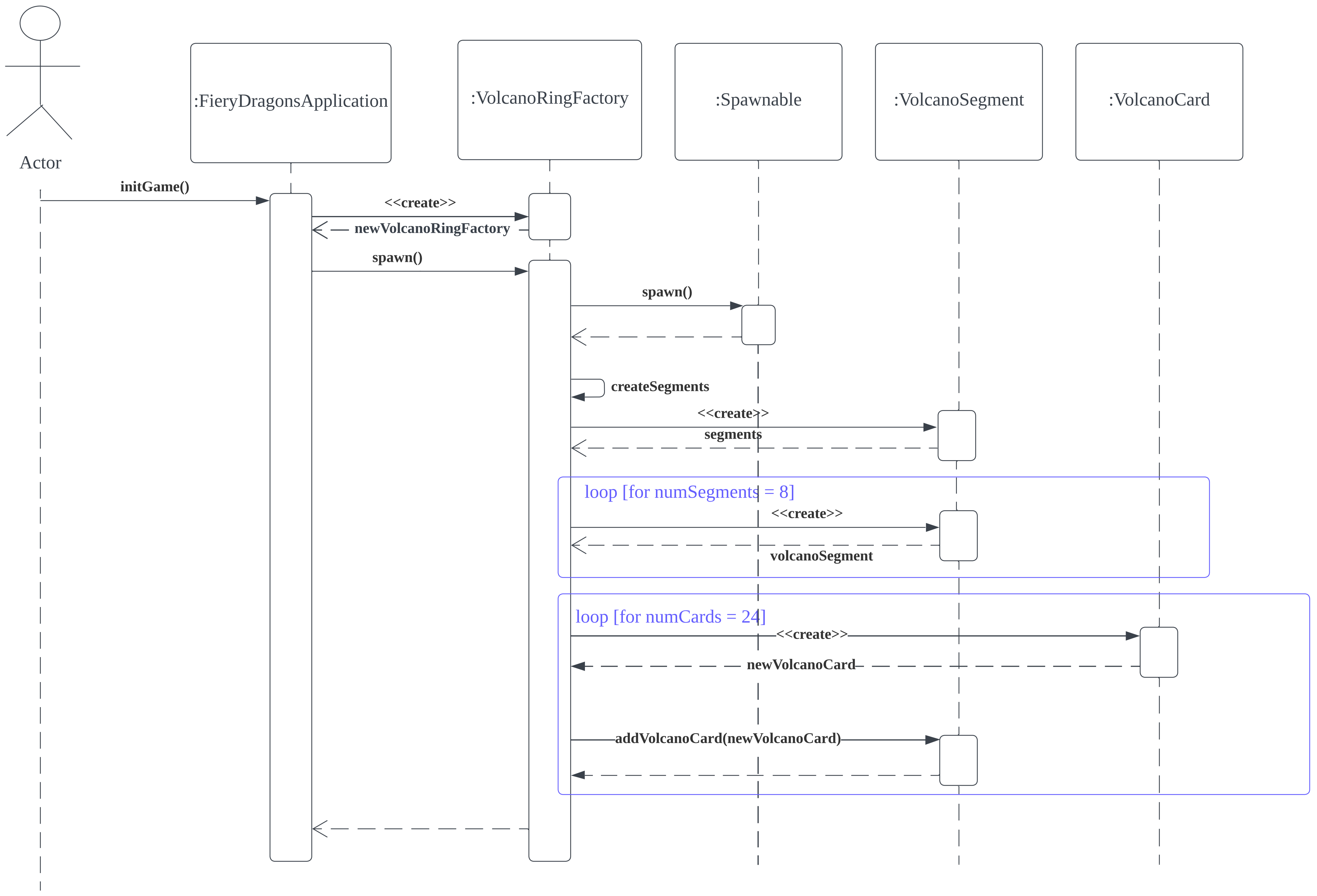
1.2 Sequence Diagrams

1.2.1 Setup of game board

1.2.1.1 Volcano Ring

The FieryDragonApplication class creates a VolcanoRingFactory and calls its spawn() method. This causes it to invoke its super class' (Spawnable) spawn() method, which in turn, calls upon the FXGL library's methods to register VolcanoRingFactory as an entity factory. This is required for the FXGL's game world to draw the shapes required in the UI. The VolcanoRingFactory class creates 8 VolcanoSegments of 3 VolcanoCards. **The following is NOT shown in the Sequence Diagram:** The actual method required by FXGL (implemented as "volcanoRing" in the source code) is called to add the VolcanoCards as a Group into the game world. This volcanoRing method contains additional logic to produce random configurations of the 4 uncut segments (i.e. the other 4 segments without a cave).

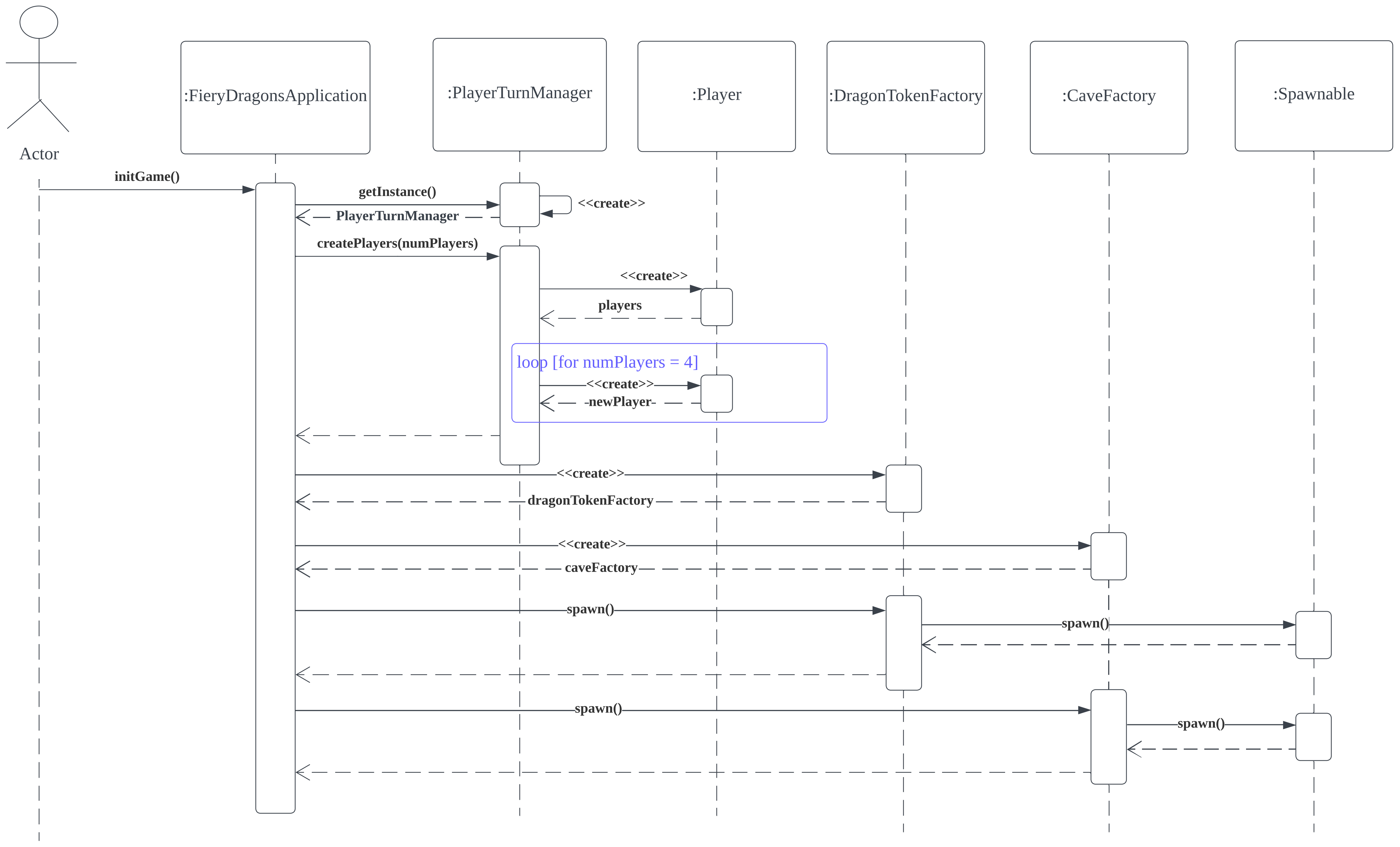
Sequence Diagram for setup of VolcanoRing



1.2.1.2 Setup of Player, DragonToken, and Cave

The FieryDragonsApplication class gets a Singleton instance of the PlayerTurnManager class. It then invokes its createPlayers() method to create the players. (the specific number is defined in the Constants class). This involves creating an array of the required size, and looping through to fill the players. Then, a DragonTokenFactory instance and a CaveFactory instance are created. They are instructed to invoke their spawn() methods similar to the case for VolcanoRingFactory above. **The following is NOT shown in the Sequence Diagram**: Their respective methods for FXGL are ‘dragonTokens’ and ‘cave’.

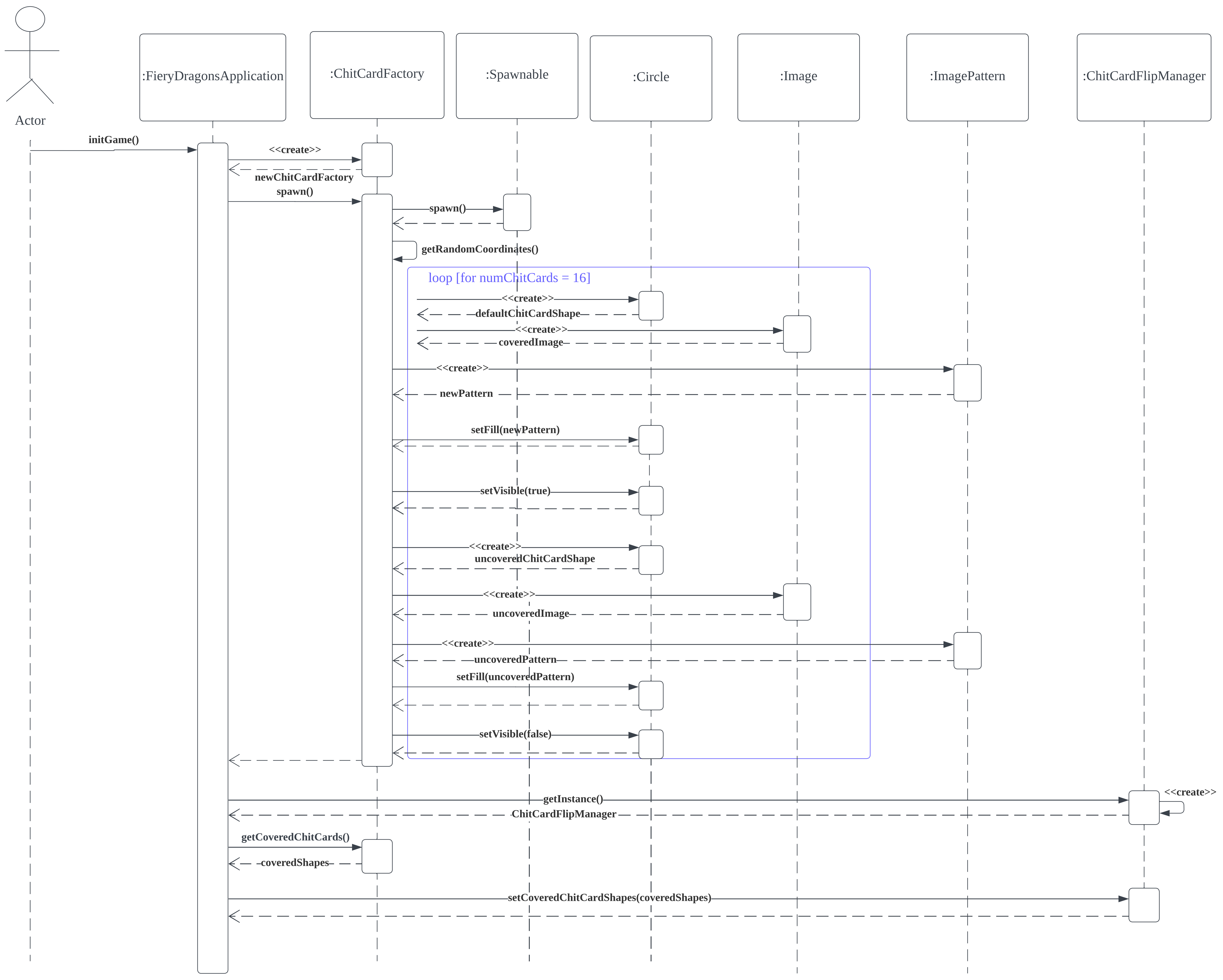
Sequence Diagram for setup of DragonToken and Cave, which are directly related to Player



1.2.1.3 ChitCard setup

The FieryDragonsApplication class creates a ChitCardFactory class and invokes its spawn() method. Again, it first calls its super() spawn() method discussed in 1.2.1.1. Then, it calls its getRandomCoordinates() to obtain 16 random (x,y) coordinates which correspond to the 4-by-4 grid where the ChitCards are placed within the Volcano Ring. Then, we loop through the number of chit cards (16), creating both the covered and uncovered shapes using the classes provided by Javafx like Circle, Image and ImagePattern. We set the visibility of the covered side to true and vice versa, since initially all chit cards are covered while the uncovered chit cards should be hidden. Whenever the chit card is flipped, the card's front/back side can be shown or hidden by toggling the setVisible() method provided by JavaFX. Finally, the information collected for the coveredShapes so far is passed to a newly created instance of the ChitCardFlipManager Singleton class (essentially filling the database for the first time)

Sequence Diagram for ChitCard-related setup

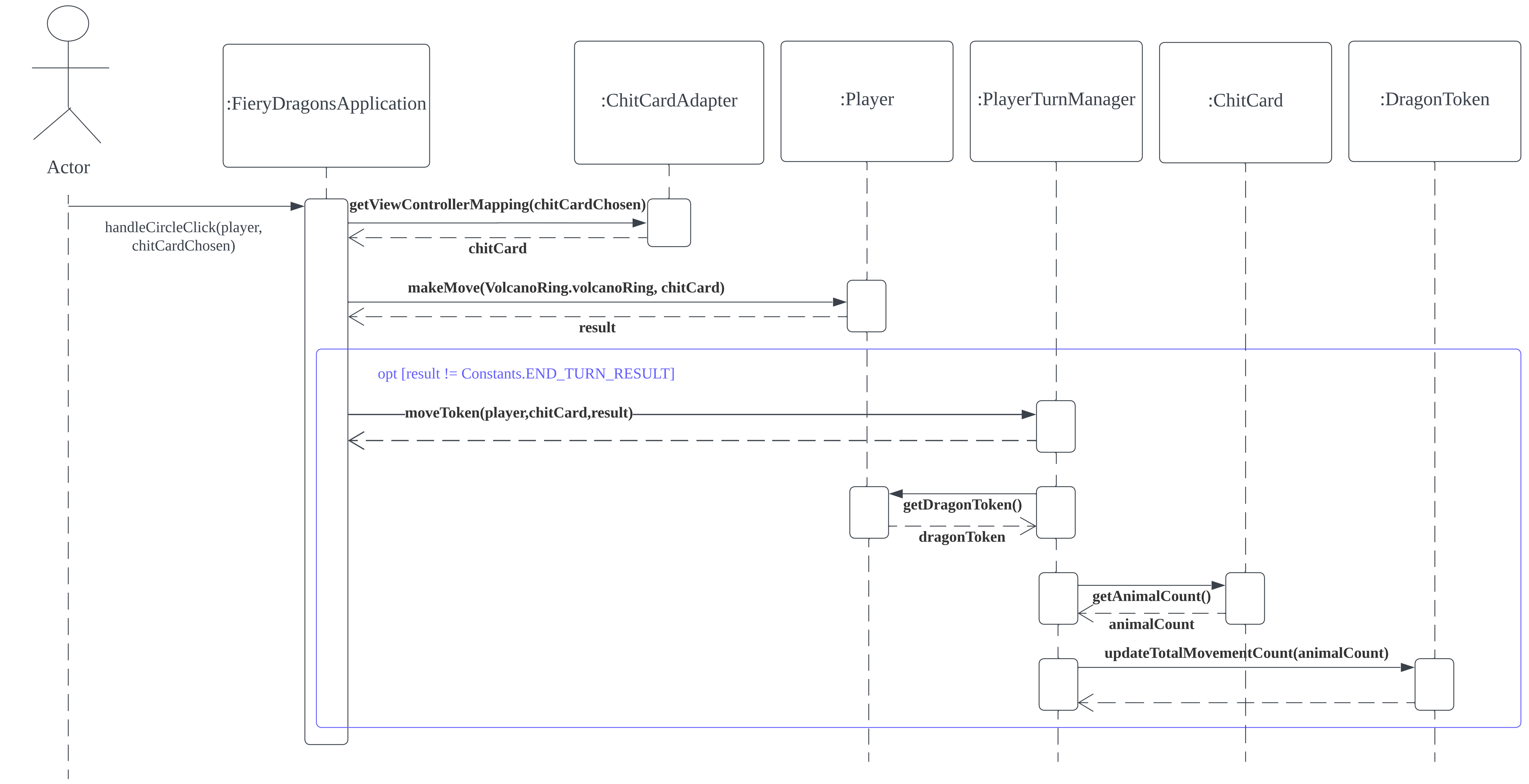


1.2.2 Winning the Game

1.2.2.1 Updating the key variable for detecting win condition

The FieryDragonsApplication class invokes the handleCircleClick() method when a Circle representing the covered chit card is clicked. It then invokes getViewControllerMapping from the ChitCardAdapter class to transform the available information (Circle) into the form required (ChitCard) to interact with the Player's makeMove() method. (See Adapter design pattern below) The Player instance then determines the result of making such a move, and the integer result is returned. If the result is not 0 (Constants.END_TURN_RESULT), then it means the integer is the ID (array index) of the VolcanoCard in the VolcanoCard array. This information is then passed to the PlayerTurnManager to manage the state update. The PlayerTurnManager gets a reference to the player's dragon token, and then invokes the getAnimalCount() on the ChitCard obtained earlier to determine the number of moves made. Finally, updateTotalMovementCount() is called on the player's dragon token to update the total movement count with the number of moves. Note that if the animal count is negative (for Dragon Pirate chit cards), then the total movement count decreases instead of increases.

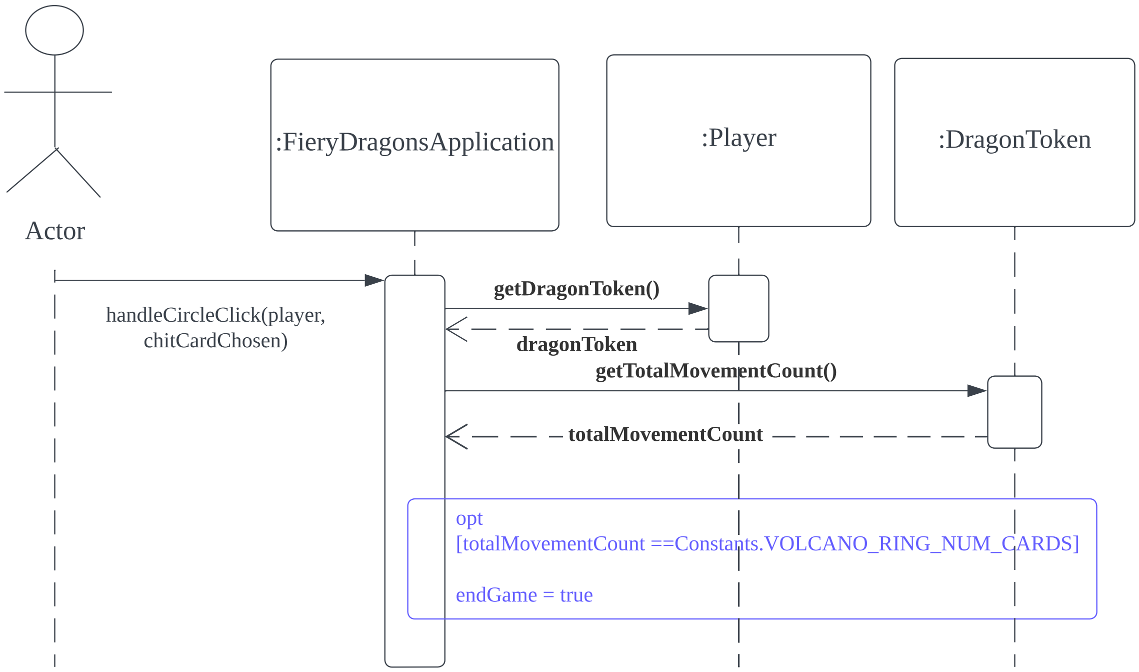
Sequence Diagram for updating key variable(DragonToken.
totalMovementCount) used for detecting win condition


Actor
:FieryDragonsApplication
:ChitCardAdapter
:Player
:PlayerTurnManager
:ChitCard
:DragonToken
handleCircleClick(player, chitCardChosen)
getViewControllerMapping(chitCardChosen)
chitCard
makeMove(VolcanoRing.volcanoRing, chitCard)
result
opt [result != Constants.END_TURN_RESULT]
moveToken(player,chitCard,result)
getDragonToken()
dragonToken
getAnimalCount()
animalCount
updateTotalMovementCount(animalCount)

1.2.2.2 Detecting win condition

Continuing from the above's handleCircleClick() method call, once the necessary updates to the dragon token's total movement count have been made, we simply check if it equals the number of cards in the volcano ring. If it is True, then endGame is set to True which is an attribute of the FieryDragonsApplication to handle this case.

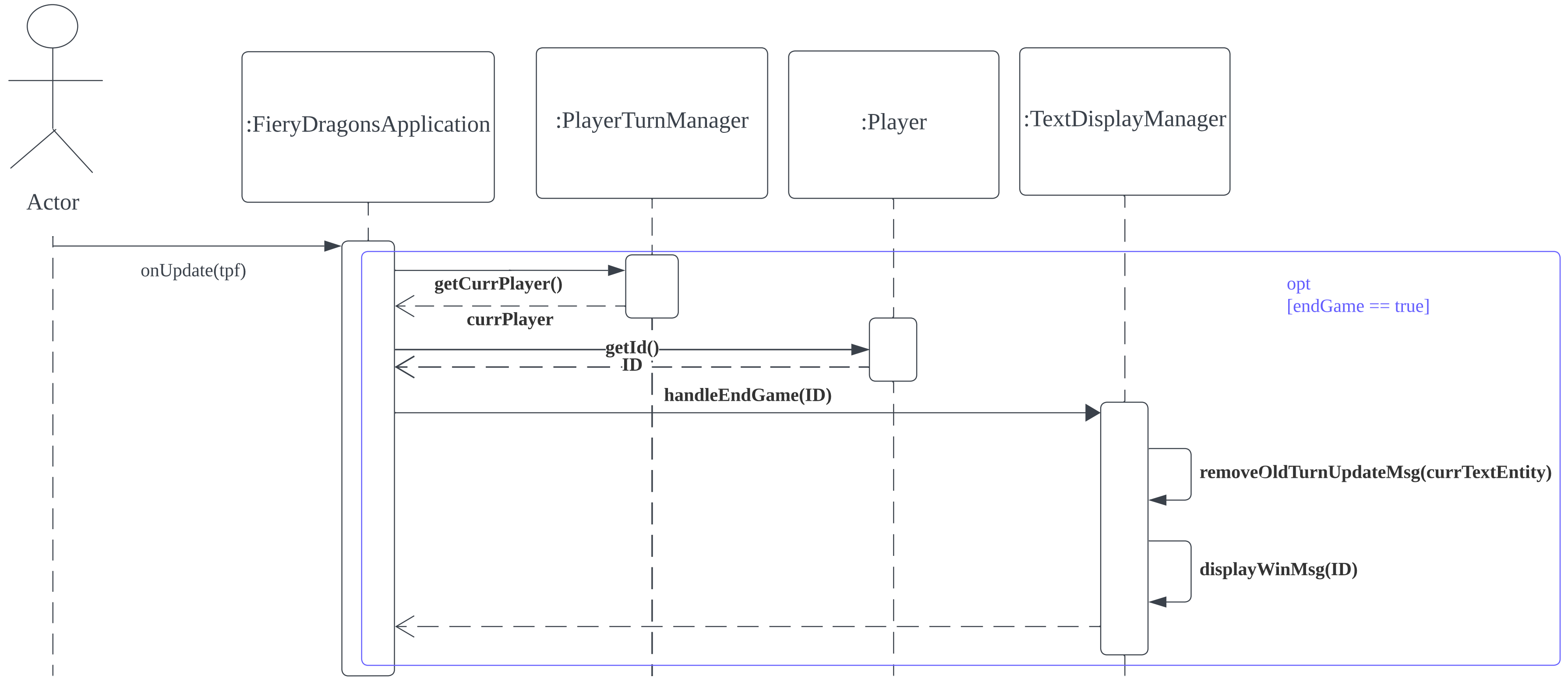
Sequence Diagram for Detecting Win Condition



1.2.2.3 Displaying win message

Continuing from the above, if endGame is true, the onUpdate() method which is called every frame asks the PlayerTurnManager to get the current player. Then it asks the TextDisplayManager to remove the old player turn message, and instead add a message indicating the player who won.

Sequence Diagram for Displaying Win Message



2. Design rationales

2.1. Classes

2.1.1 FieryDragonsApplication class

The main class FieryDragonApplication sets up the game board and the players (represented by their dragon token) via the initGame() method, and this justifies creating classes for these components.

There are several responsibilities involved for the components above: initialization, information storage, and state management, and I decided to dedicate classes for each responsibility. This supports the Single Responsibility principle and enforces the separation of concerns. As a side note, I decided to model the game board as its separate components(dragon token, cave, chit card, volcano ring) instead of putting everything into a single ‘GameBoard’ class to avoid making it a GOD class.

2.1.2 Factory classes

For initializing each component’s state and UI, I created a Factory class (DragonTokenFactory, CaveFactory, ChitCardFactory, VolcanoRingFactory). I did not create a PlayerFactory class, as the setup for the player is mainly handled via its dragon token. I created a TextFactory class which helps to display text indicating the current player’s turn or a win message.

2.1.3 State classes

To store the state of each component, I created DragonToken, ChitCard, VolcanoSegment and VolcanoCard classes. I decided to model the volcano ring as an array of VolcanoCards. The VolcanoSegment stores an array of 3 Volcano Card instances, and exists purely as a helper class to help the VolcanoRingFactory set up random configurations of the Volcano Card segments. The Player class stores a reference to the dragon token it controls, and is responsible for determining the outcome of flipping a chit card (see ‘makeMove’ method)

2.1.4 Manager classes

State management is handled by the appropriate Manager class (PlayerTurnManager, TextDisplayManager, ChitCardFlipManager). These classes manage the state of objects in response to player input, as well as update the UI with their methods. The PlayerTurnManager class manages player turns as well as dragon token state, as they are closely tied together. While creating a separate DragonTokenManager class would shift some responsibility from the PlayerTurnManager class, it would make synchronizing the logic more difficult.

I did not create a Cave class nor a CaveManager class because it only has a UI component that remains static throughout the game, so there is no information storage/state management required. This means the CaveFactory class only needs to perform initial spawning of caves.

2.2. Relationships

2.2.1 Dependencies between FieryDragonsApplication with Factory

The FieryDragonsApplication creates the factory classes and calls their spawn() methods to perform the initial board setup. This only occurs once in the initGame(), so no need to store references to these classes.

2.2.2 Dependencies between FieryDragonsApplication with the Manager classes

An association relationship is not required because the Manager classes follow the Singleton pattern, so the main class can call the getInstance() method of the Manager classes whenever needed.

2.2.3 Composition relationship between (i) Player and DragonToken, (ii) PlayerTurnManager and Player, (iii) ChitCardAdapter and ChitCard.

Player instances are created before the DragonTokenFactory creates the DragonTokens, so DragonTokens only exist after the Player is created. However, the main point is that dragon tokens are only accessed or modified via the Player or PlayerTurnManager class (which also calls upon the player to get the dragon token). This means it is intrinsically tied to the Player instance, and cannot exist without the Player instance. The same is true for the composition relationships (ii) and (iii).

2.2.4 Aggregation relationship between (i) DragonToken and VolcanoCard, (ii) VolcanoSegment and VolcanoCard, (iii) VolcanoRingFactory and VolcanoCard

Notice that the VolcanoCard is present as the second class in the association relationships above, which indirectly proves the aggregation relationship. A VolcanoCard instance is first created by the VolcanoRingFactory. It is subsequently added to the VolcanoSegment's array (for random setup of volcano ring). Finally, the DragonToken class will update its reference to the VolcanoCard it is currently on each player's move. Clearly, the VolcanoCard exists independently of its owner's existence, as it interacts with other classes.

2.3. Inheritance

2.3.1 Factory classes extend SpawnFactory

The SpawnableFactory class which implements the EntityFactory interface (from the FXGL package) is created as an abstract class for all the Factory classes. This is because of the common functionality of all factory classes: (i) they must have a SpawnData attribute for calling the EntityFactory interface's method (ii) they must register themselves as an EntityFactory for FXGL to spawn them as UI elements via a special method call (see spawn() in SpawnableFactory class). By defining SpawnFactory as the parent class, and providing a default implementation of spawn() which performs the two tasks above, we ensure that all the Factory classes correctly register themselves in the FXGL gameworld by calling 'super.spawn()'. In addition, the child classes can override this default implementation to add any required data to the payload (SpawnData) before the actual spawning is done.

2.3.2 No inheritance for UI shapes from JavaFX

The ChitCard and Cave components use a Circle as the shape, while the VolcanoCard and DragonToken components use a Rectangle as a base. The Text component is also used for displaying text. An alternative design is to define a base class (call it BaseShape) for all these UI components which would contain common information: x postion, y position, imageFill, etc, and add methods for setting specific properties. Then, each class that has a UI component would have an attribute referencing their specific JavaFX:scene component. The significant benefit of this design is that it relieves some responsibility from the Factory classes by reducing the number of instantiations and method calls of these JavaFX Shapes. However, the two drawbacks are

(i) Redundancy. The JavaFX shapes already come with parameters and methods that can be used as-is; creating an extra class that sets these parameters is redundant.

(ii) (ii) Added complexity in assigning images to Shapes. Currently, assigning image resources for shape fill is handled using indexes, which is easier when the Factory classes manage it. If we used a BaseShape class, this index information would need to be initialized and managed as its attribute, which achieves the same outcome but with extra complexity.

2.4. Cardinalities

2.4.1 Basic board setup

All the cardinalities shown (except for the 1..1 ones) model the actual game board. For example, the game can be played by 2-4 players, so PlayerTurnManager creates the array of players according to the actual number of players. (In the source code, this is set in the Constants file). The volcano ring consists of 24 cards, and one segment consists of 3 cards. There are 16 chit cards.

2.4.2 Other cardinalities

PlayerTurnManager, TextDisplayManager and ChitCardFlipManager have 1..1 self-association loops because they create a single static instance when their getInstance() methods are called (See Singleton pattern below). There is a 1..1 relation between Player and DragonToken because each player controls their unique DragonToken. Similarly, the 1..1 relation between DragonToken and VolcanoCard represents the fact that the DragonToken is always on one volcano card, and it should be updated whenever the token moves to a new card.

3. Design Patterns

3.1 Singleton Pattern

The PlayerTurnManager, TextDisplayManager, and ChitCardFlipManager classes are implemented as Singleton classes, as shown by the self-association loop. (See also source code for the private constructor, as well as getInstance() method which only ever creates one instance). These classes receive information about the instantiated objects by the Factory classes, and take over the handling of these objects for the remainder of the game. This is important because the global data managed by these Singleton classes needs to be kept consistent throughout the game to prevent logic errors. Therefore, a global access point is provided for other classes for read operations. For example, the DragonTokenFactory requires access to the players involved in the game, so that the Players can set their DragonToken attribute, and this service is provided by the PlayerTurnManager instance. Similarly, the TextDisplayManager class manages a global entity for the text to be displayed according to changes in the Player's turn and handling the win condition. Finally, the ChitCardFlipManager must manage the updates to the UI components (uncovering chit card when a player clicks chit card, and covering up chit cards when the player's turn ends), as well as the state of the ChitCard (setting the covered attribute accordingly). In summary, the Singleton Pattern is useful here for ensuring that state updates are consistent, and avoids unnecessary object creation.

3.2 Adapter pattern

The ChitCardFactory creates both the UI components (Circle) and the state for chit cards (ChitCard) in the game. When the Circle component detects a player's click, the other components that need to manage their state in response to this event need to communicate with the ChitCard instance, because it is the class used for storing the state of each ChitCard. This justifies a mapping of the UI component to the state component, i.e. Circle -> ChitCard. This is where the adapter class ChitCardAdapter helps to store this mapping, and ‘adapts’ the Circle to its ChitCard instance for further state processing required by other classes. In the code, the ChitCardFactory helps to initialize this mapping, while the FieryDragonsApplication, ChitCardFlipManager, and Player classes are the Adaptees that need to access the ChitCard instance, and this service is provided by the ChitCardAdapter class.

3.3 Why I did not implement the Memento Pattern

When a player’s token lands on a volcano card that is already occupied by another player’s dragon token, the dragon token should move back to its original position. In my specific implementation, I decided to decouple the actions from the moves and introduce move validation. The player’s makeMove() method works together with the VolcanoRingFactory to check if the destination volcano card is occupied. If it is occupied, the dragon token does not move, and it returns Constant.END_TURN_RESULT = 0, which indicates that the player’s turn has ended.

4. Video Demonstration

Link to YouTube video (3 minutes 15 seconds): https://youtu.be/2bmGBBa5XHw

Link to YouTube video showcasing board setup for 2/3 players (38 seconds): https://youtu.be/HlQhavesGrk