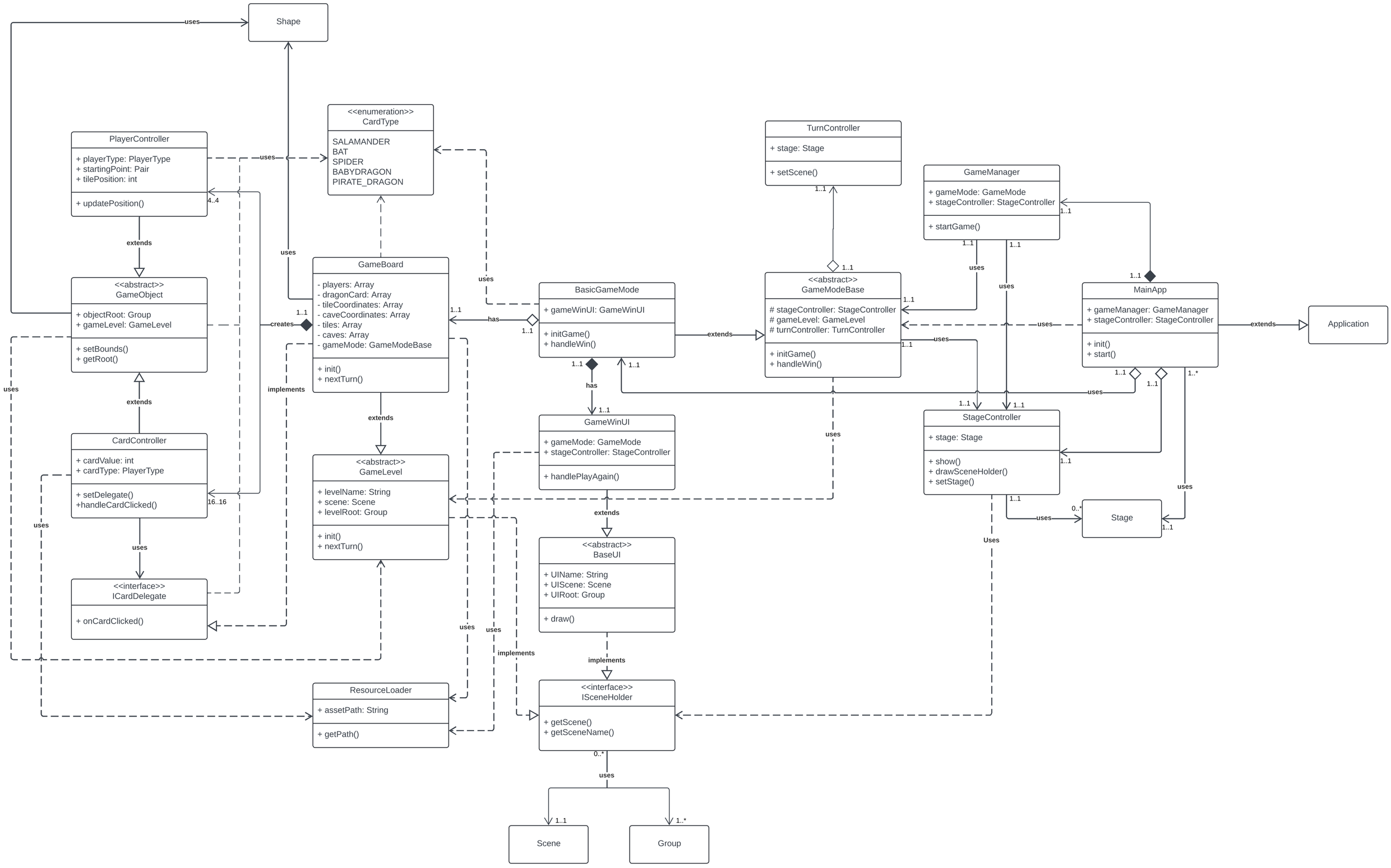
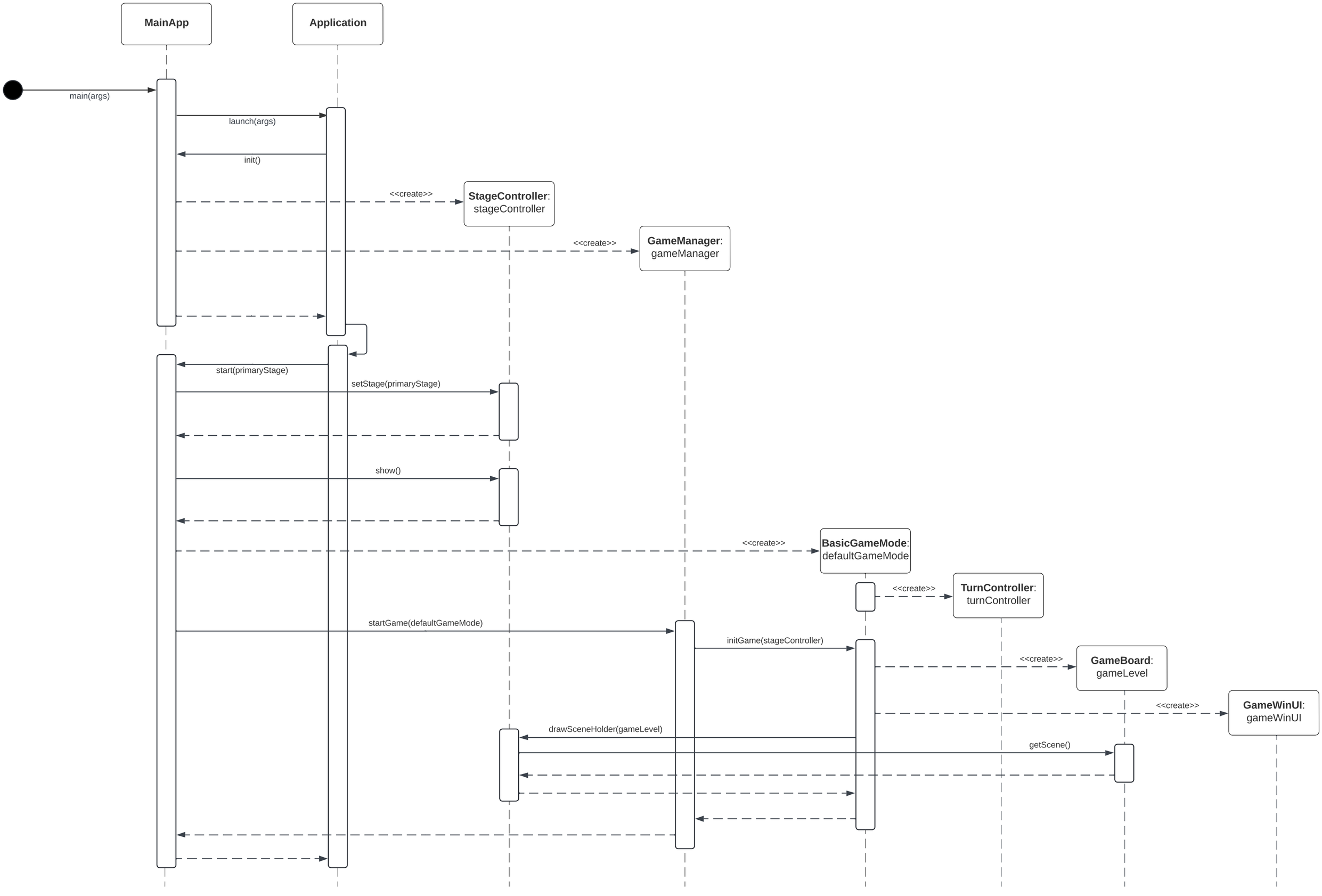
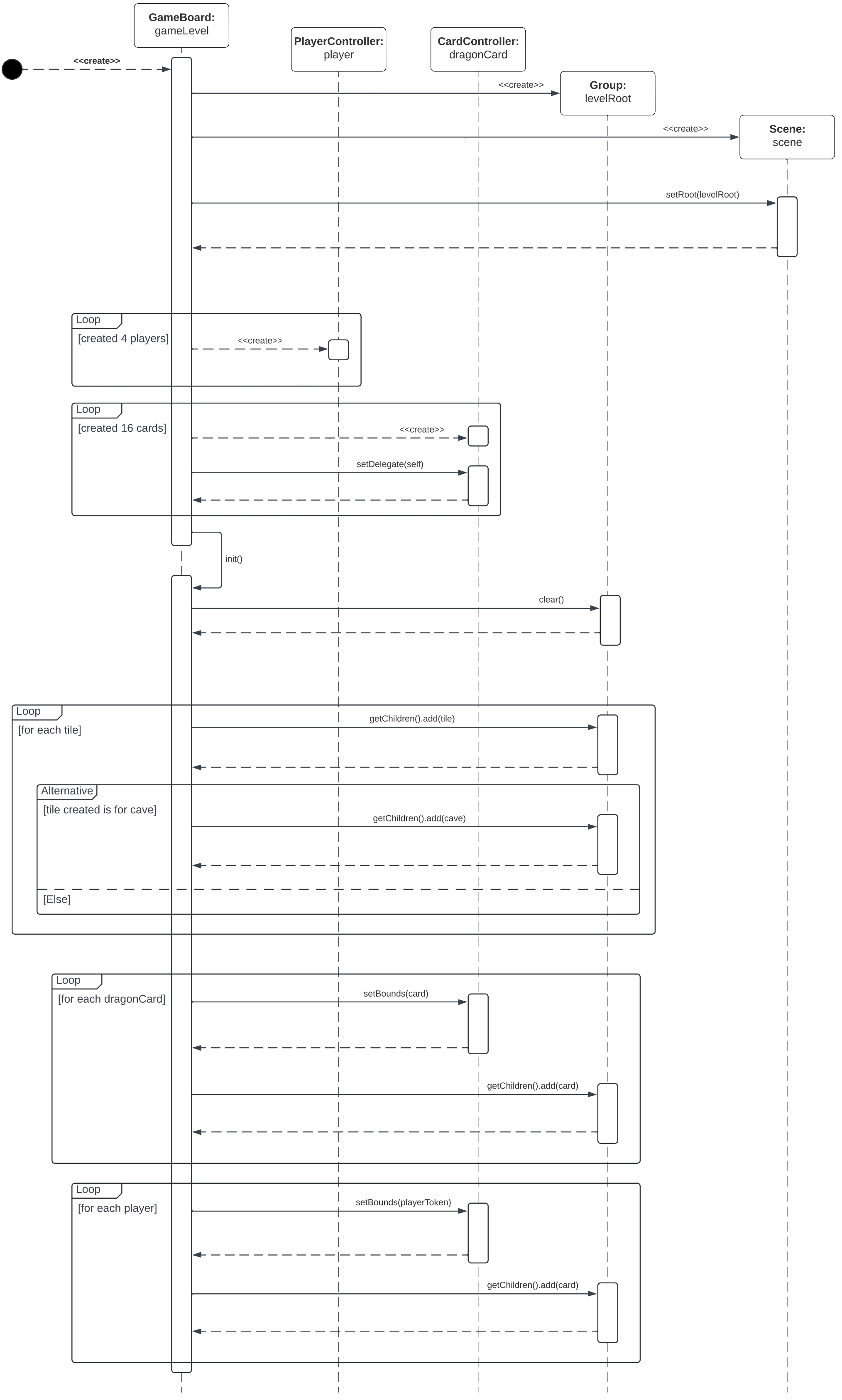
Class Diagram



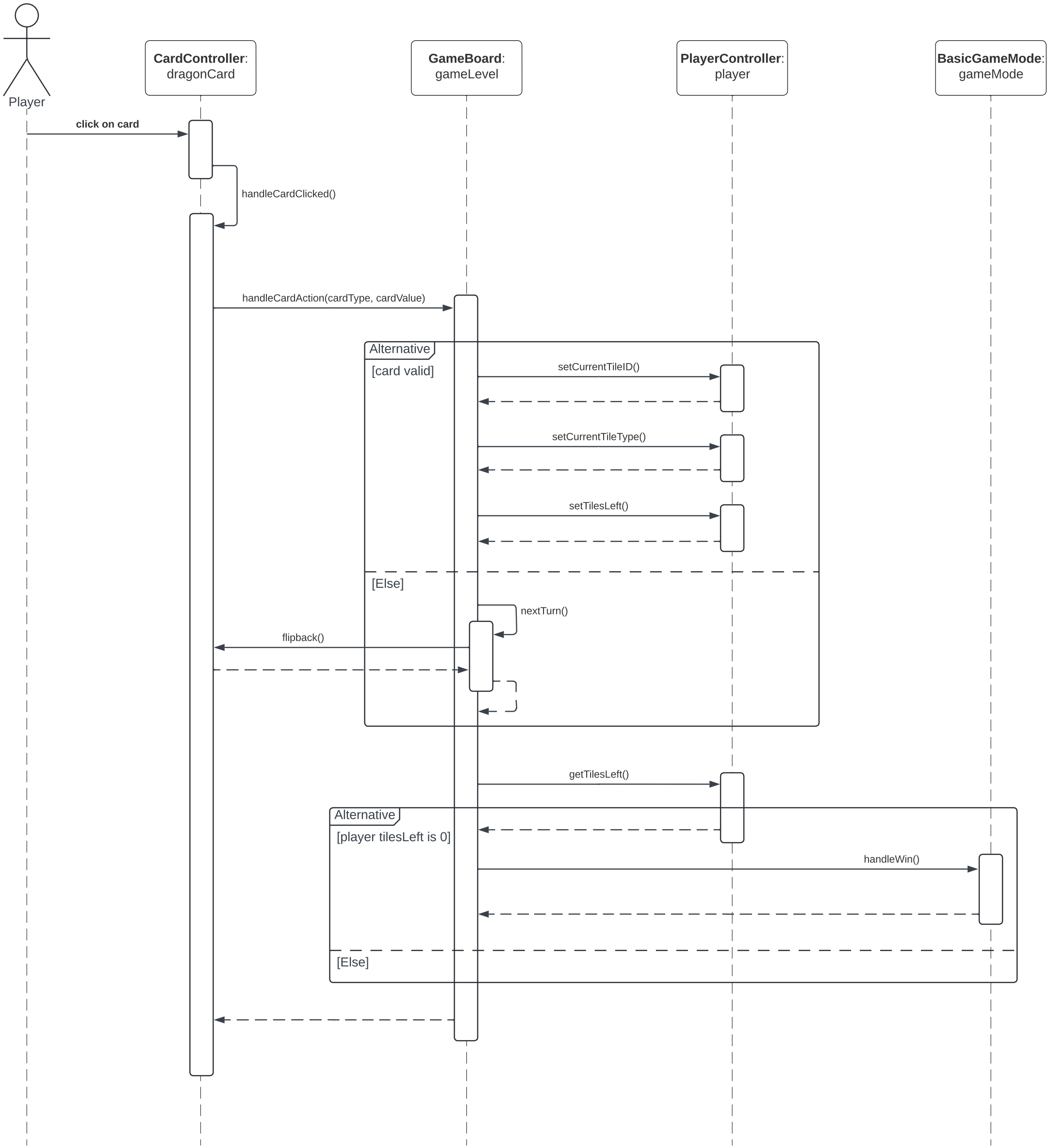
StartUp Sequence



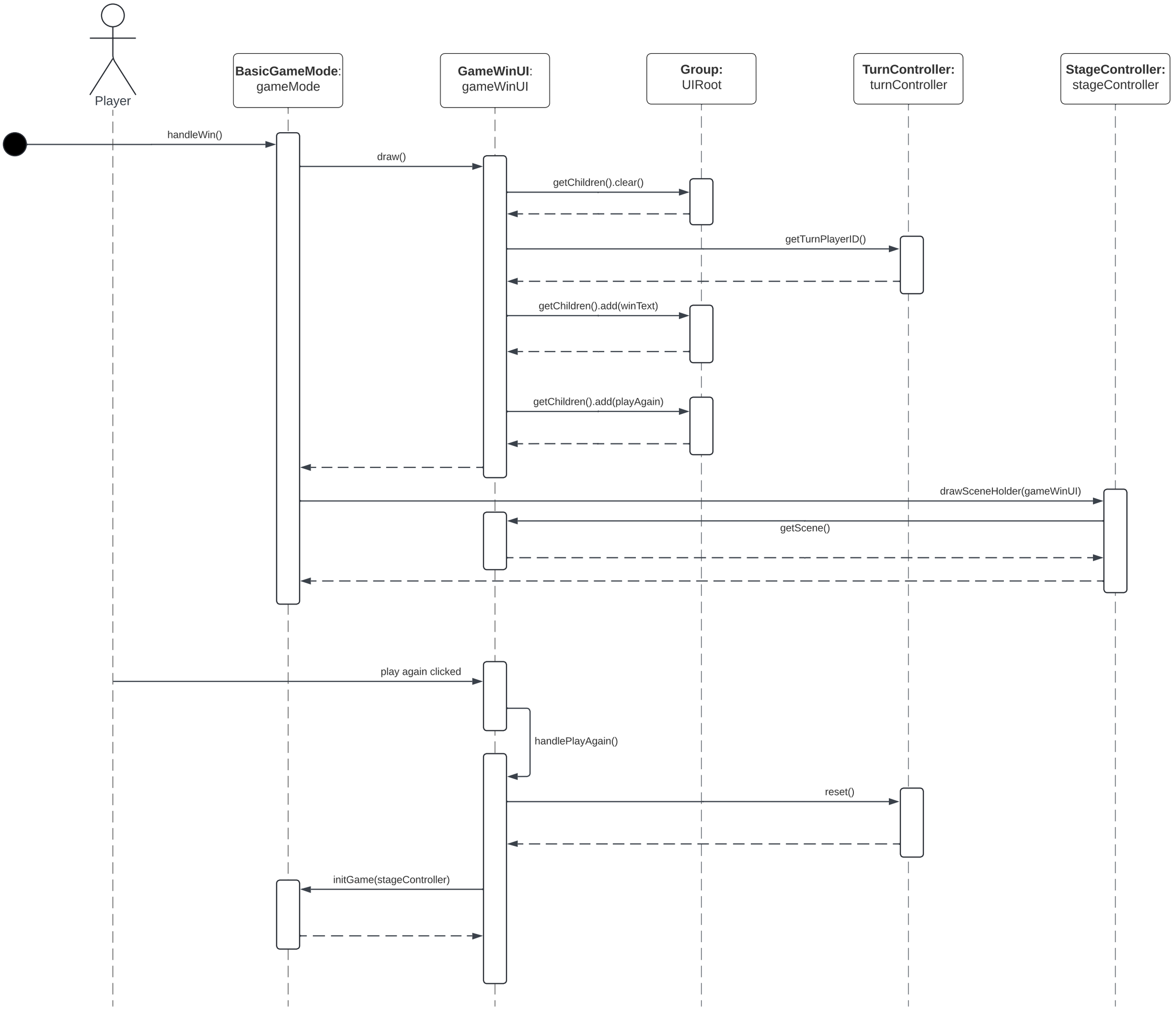
GameLevel Draw Sequence



CardFlip Sequence



HandleWin Sequence



Fiery Dragon Design Rationale

In my design, there are the GameModeBase and GameLevel abstract classes that I then proceed to extend into BasicGameMode and GameBoard each. The reasoning behind this is on the concept, separation of concern. GameMode (hereafter abbreviated as GameMode) in my design is used for starting a game with a provided setting, therefore the concern on how the game should be is the responsibility of it, while GameBoard is for the drawing and anything related to the GameBoard, e.g. drawing of game objects, determining coordinates for game objects. Another use case that GameBoard has is it acts as the bridge between the CardController and PlayerController where when a card is clicked and thus "flipped", the GameBoard will translate any related value into coordinates for the PlayerControllers to store/act upon.

Due to this relationship of CardController and PlayerController relying on GameBoard and their lifetime is dependant on GameBoard as logically speaking, game objects should not exist outside the context of a GameLevel, therefore a composition relationship from GameBoard to them is warranted. Another relationship that is of similar nature is the relationship between BasicGameMode and GameWinUI. In this case, my idea was that GameMode will define the method of handleWin for GameLevels to invoke when needed, since using examples of games throughout times, the winnning is determined by the game level, for example, chess and checkers although share the same mode of play (two players, and you move pieces on a checkered board), but each game have a different winning factor depending on the game even though the board is the same, therefore deducing GameLevel determining the winner to end the game, then call the handleWin method of GameMode, which then comes to the relationship we are looking at, the relationship between GameWinUI and GameMode. Here the UI is specifically drawn/coded for said GameMode so we would not want it to appear anywhere other that GameMode, so a composition relationship is hence drawn to ensure when implementing, the lifetime of GameWinUI is tied to the lifetime of GameMode.

In terms of inheritances in my design, the two key classes mentioned above, BasicGameMode and GameBoard both inherits GameModeBase and GameLevel. Other inheritances includes, GameWinUI inheriting BaseUI, PlayerController & CardController inherits GameObject. The thing these inheritances have in common is that they are abstractions of the inheriting children. The way I design these abstract classes are around the core functions or use of said classes. Looking at GameLevel and GameObject, GameLevel has an init method that is forced onto inheriting classes as without any graphical implementation, a GameLevel is not a valid level, and centralizing leveling initialization in one method makes sure no boiler plate code is in place to confuse developers. The same goes for GameObject where in the core system I designed, a game object is defined to have a bound, where can be provided by a GameLevel, here it does not have an init is solely based on the reason that a game object have no right to exist without a level, so the graphics should be provided by the level through the setbounds method. BaseUI also have similar

reasoning, where UI needs to be drawn and therefore a draw method, which shares a similar concept with GameLevel just with different naming and implementation/sequence. The finally abstract class and inheritance in my design is the abstact class GameModeBase, the core system is designed based on the game always being turn based, so it will automatically provide a TurnController for all inheriting classes. In the top part of this paragraph it was mentioned that all GameMode will have a method for GameLevel to invoke at someone winning, therefore having a BaseGameMode class for all these base method that needs to be forced upon all future inheriting classes is warranted by nature.

In terms of cardinality, all core systems from StageController, GameManager, GameModeBase to GameLevel, should only exist one at a time so at most one, as having more of them is not logical as only one class instance is at work at anytime and having more of them is just memory overhead. In terms of PlayerController and CardController, it just depends on the gamerule, since the current gamerule we are following states that the game is to be played with 4 players and 16 cards, the cardinality reflects said rules.

On the topic of design patterns, one design pattern I used to design the interaction, where after the player clicked a card, the corresponding token then moves accordingly, is the delegate pattern, which is somewhat similar to the strategy pattern, where the execution logic is passed on to another class that has the ability to execute said logic. In this case, the ICardDelegate is implemented by GameBoard as it has the ability to determine the movement logic as it is the only class that knows the actual location of the card and player tokens. The GameLevel itself more or less is a flyweight pattern in concept, as I separated the bulk of the computation to draw the level into the init method and anything a level redraw is needed, only the init method needs to be invoked, instead of the reinstantiating the level over and over again. One last pattern that is used is composite pattern, as the way my entire application scene is structured is through a tree of nodes. GameLevel when drawing GameObjects adds the tree structure of the GameObject into the GameLevel own tree structure, which then the JavaFX framework can interpret and render. So instead of dealing with each graphical object individually, a composite tree structure is used to keep track of all the objects.

Fiery Dragon Executable

(The executable can be run on any windows machine)
Executable Name: FieryDragon.exe