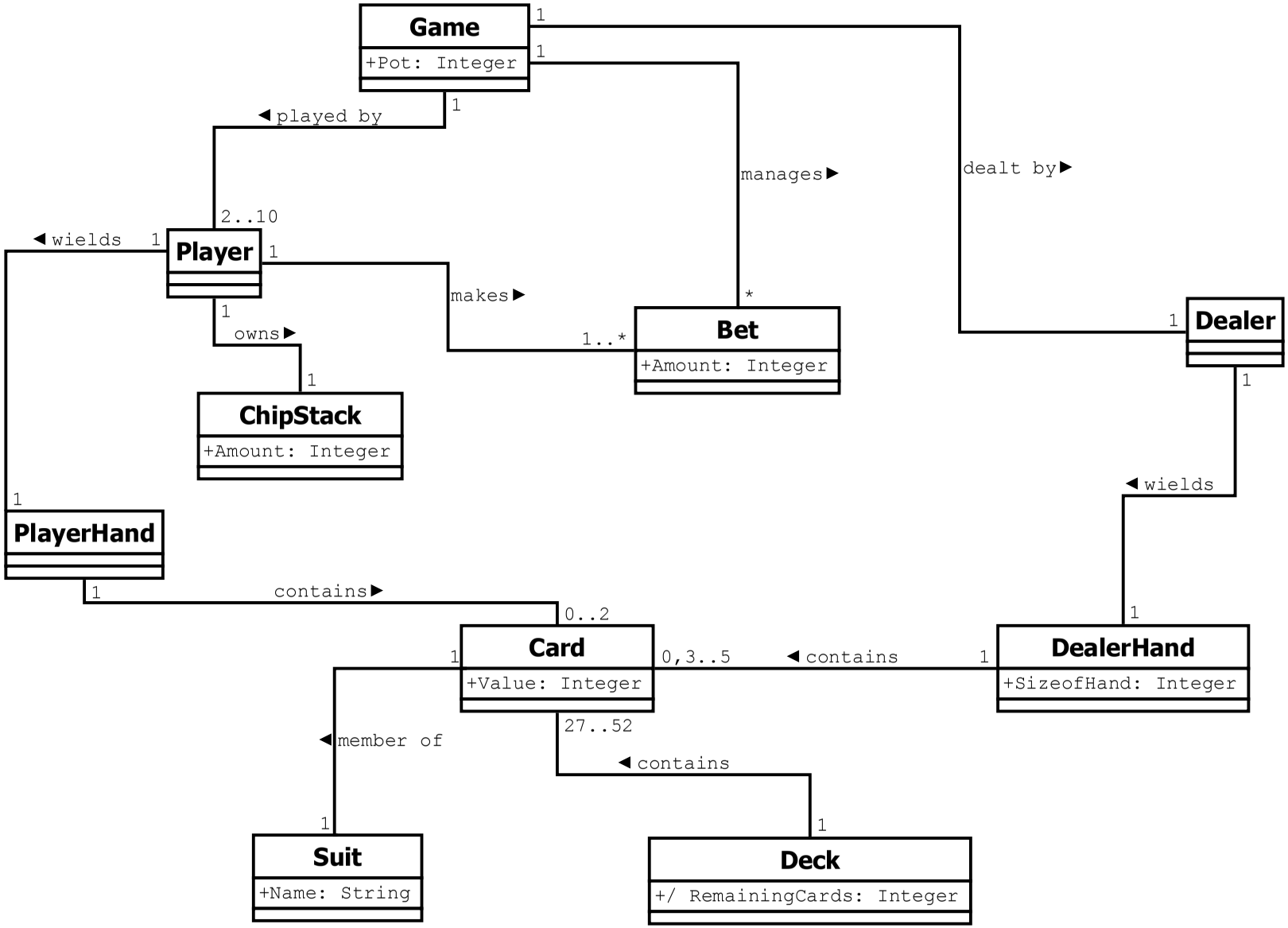
Milestone 4

Concurrent Poker Player Team

Mark Jenne, Bennie Waters, Ian Roberts, Sarah Jabon

2/1/2010

Contents


Game
+Pot: Integer
played by
manages
dealt by
Player
wields
makes
owns
Bet
+Amount: Integer
Dealer
ChipStack
+Amount: Integer
PlayerHand
wields
contains
Card
+Value: Integer
contains
DealerHand
+SizeofHand: Integer
member of
contains
Suit
+Name: String
Deck
+/ RemainingCards: Integer

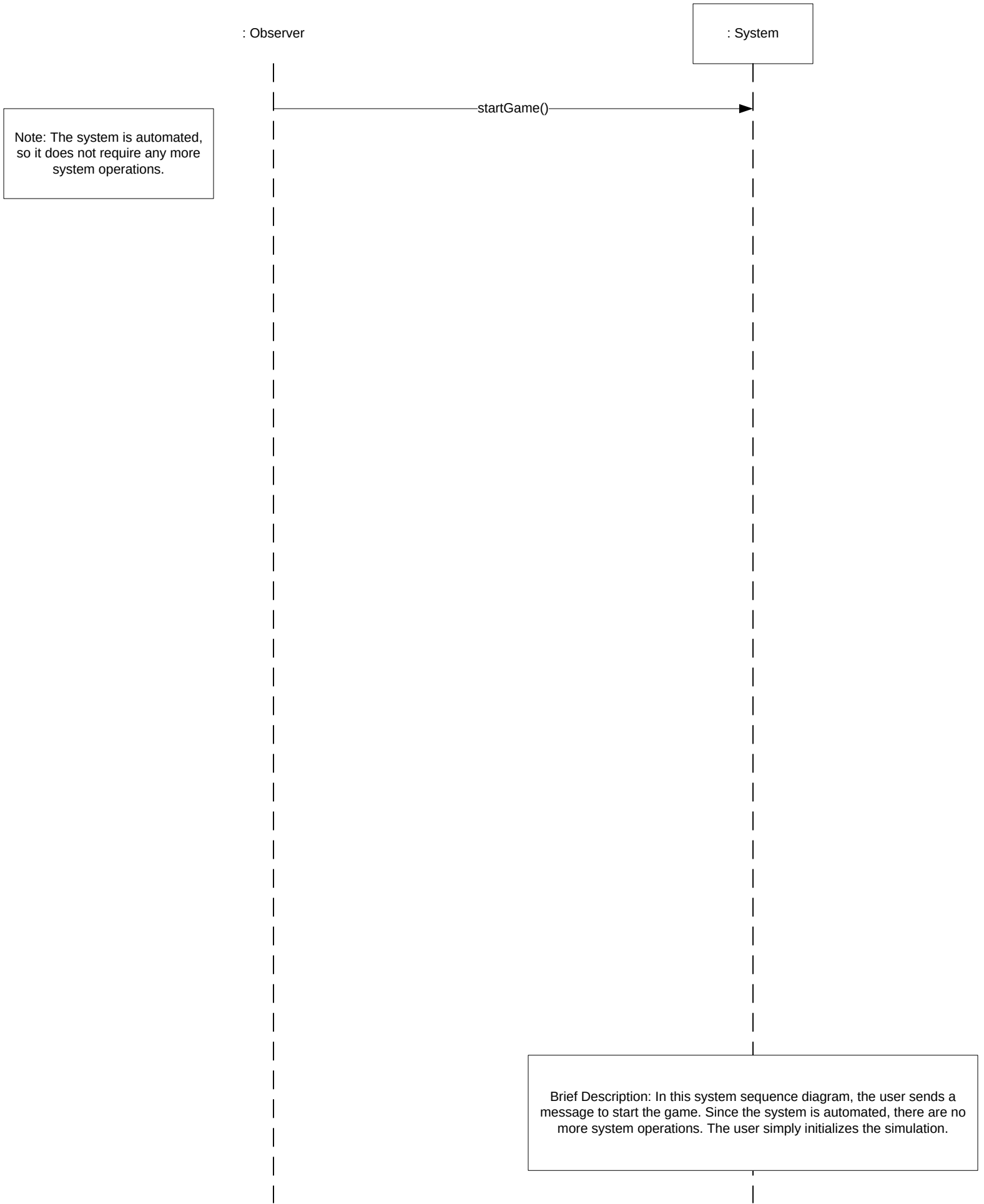
The Game class represents the entire poker game. The Game class has one attribute, Pot, which describes the amount of money currently in play. The Game manages Bets, because there are rules in the game that specify the amounts that a bet can be based on game states.

A Dealer wields one DealerHand. In Texas Hold ‘Em Poker, the DealerHand consists of zero to five Cards depending on the round in the game. Therefore, SizeofHand is an attribute of DealerHand. SizeofHand is always zero, three, four, or five based on Texas Hold ‘Em play.

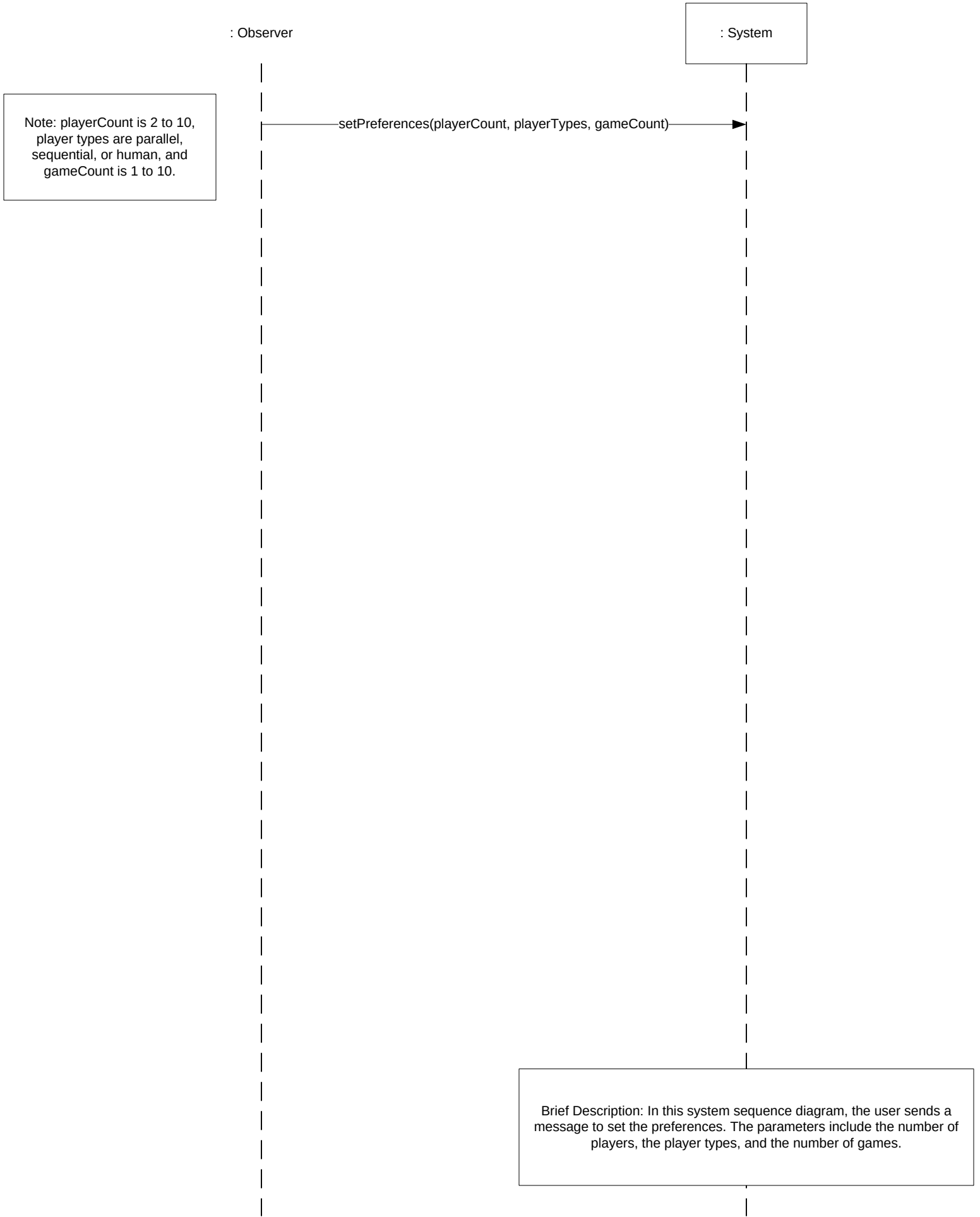
A Deck contains all the Cards that are not currently in play. The number of cards in the Deck is an attribute of Deck called RemainingCards. The Deck will always contain between 27 and 52 Cards, because the largest number of Cards that can be in play is 25 (two for each Player and five for the Dealer).

A Player wields one PlayerHand, which contains zero to two Cards, depending on the round of the poker game. At the beginning of the game, the Player has zero cards. Throughout the rest of the game, the Player will have two cards. A Player also owns one Stack, which has an attribute of Amount. Amount is an integer representing the amount of money with which the Player can use to bet. The Player makes Bets. Since the Player must make a Bet at the beginning of the poker game, the Player must make at least one Bet. The Bets have an attribute, Amount, which represents the amount of money the Player bet.

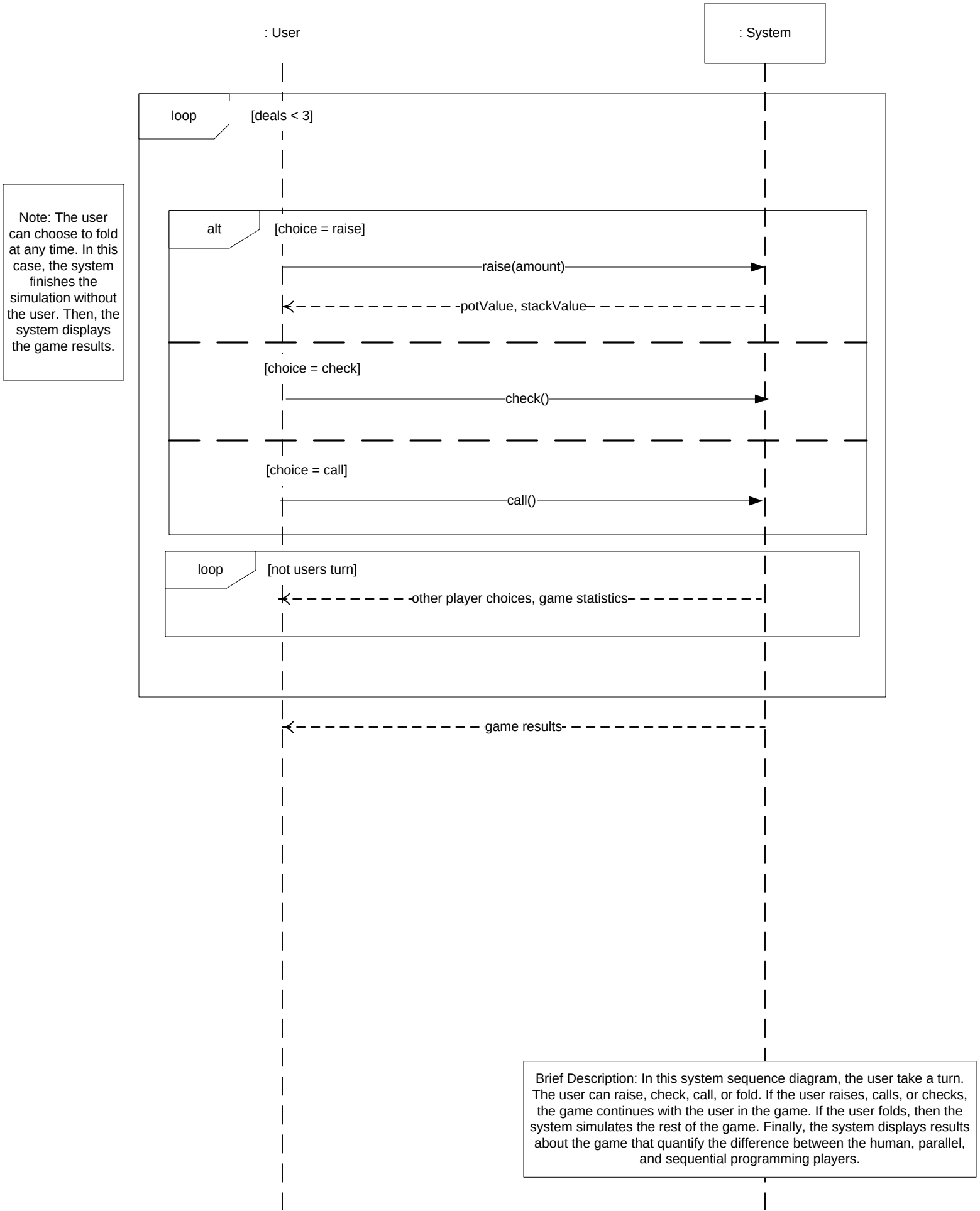
SSD – Use Case 1
(Start Game)



SSD – Use Case 2
(Set Preferences)



SSD – Use Case 3
(Take Turn)



Contract CO1: startGame

Operation:	startGame()
Cross References:	Use Cases: Start Game
Preconditions:	The system is set to create a game with two players – one sequential and one parallel.
Postconditions:	An instance of Game, *game*, was created An instance of GameStatistics, *gameStats*, was created Attributes of *gameStats* were set based on simulation

Contract CO2: setPreferences

Operation:	setPreferences(playerCount, playerTypes{List}, gameCount)
Cross References:	Use Cases: Set Preferences
Preconditions:	The application has initialized properly Parameter menus are functioning properly An instance of Game, *game*, exists
Postconditions:	*game.playerCount* was set to playerCount *game.gameCount* was set to gameCount *player*, an instance of HumanPlayer, was created Instances of other Players were created corresponding to each playerType

Contract CO3: raise

Operation:	raise(amount)
Cross References:	Use Cases: Take Turn
Preconditions:	The game is in play It is the user's turn in the game An instance of Game, *game*, exists *player[i]* is currentPlayer *player[i]* is an instance of HumanPlayer *player[i+1]* is nextPlayer
Postconditions:	*game.pot* was increased by amount *player[i].stack* was decreased by amount *player[i]*'s turn ended *player[i+1]*'s turn began

Contract CO4: check

Operation:	check()
Cross References:	Use Cases: Take Turn
Preconditions:	The game is in play *players{list}* are playing game *player[i]* is currentPlayer *player[i]* is an instance of HumanPlayer *player[i+1]* is nextPlayer
Postconditions:	*game.pot* was increased by the amount of the last raise *player[i].stack* was decreased by amount of the last raise *player[i]*'s turn ended *player[i+1]*'s turn began

Contract CO5: check

Operation:	call()
Cross References:	Use Cases: Take Turn
Preconditions:	The game is in play *players{list}* are playing game *player[i]* is currentPlayer *player[i]* is an instance of HumanPlayer *player[i+1]* is nextPlayer
Postconditions:	*player[i]*'s turn ended *player[i+1]*'s turn began

Contract CO6: fold

Operation:	fold()
Cross References:	Use Cases: Take Turn
Preconditions:	The game is in play *players{list}* are playing game *player[i]* is currentPlayer *player[i]* is an instance of HumanPlayer *player[i+1]* is nextPlayer
Postconditions:	*player[i]*'s turn ended *player[i+1]*'s turn began *player[i]* was removed from *players{list}*

GUI

Display Area

Animation

Button

Menu

Image

Package Diagram With Classes

Game

State «interface»

DisplayCard «interface»

Gamestate

Hand

Card

Game

AIandPlayers

Player «interface»

Parallel Player

Sequential Player

MoveGenerator

System

Persistor «interface»

FileWriter

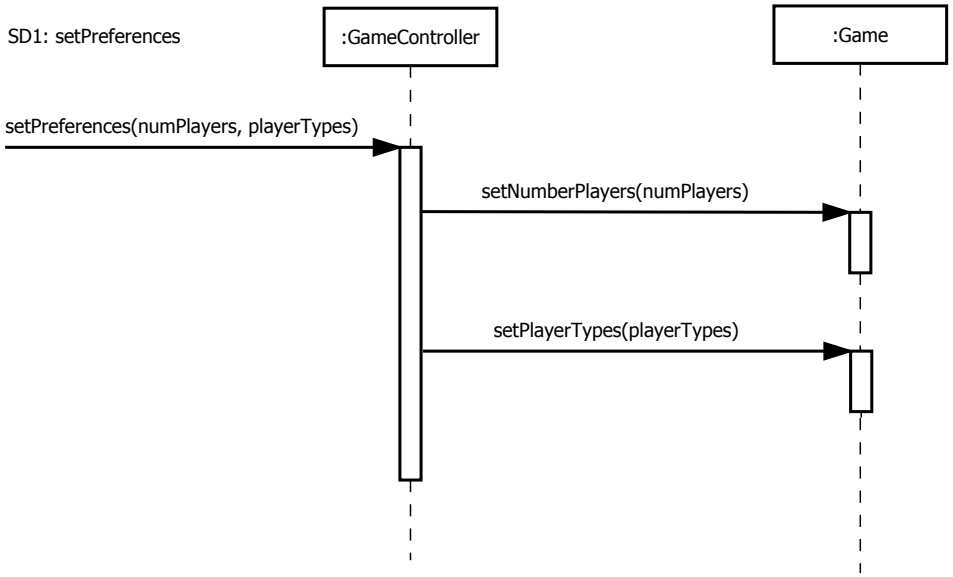
Network

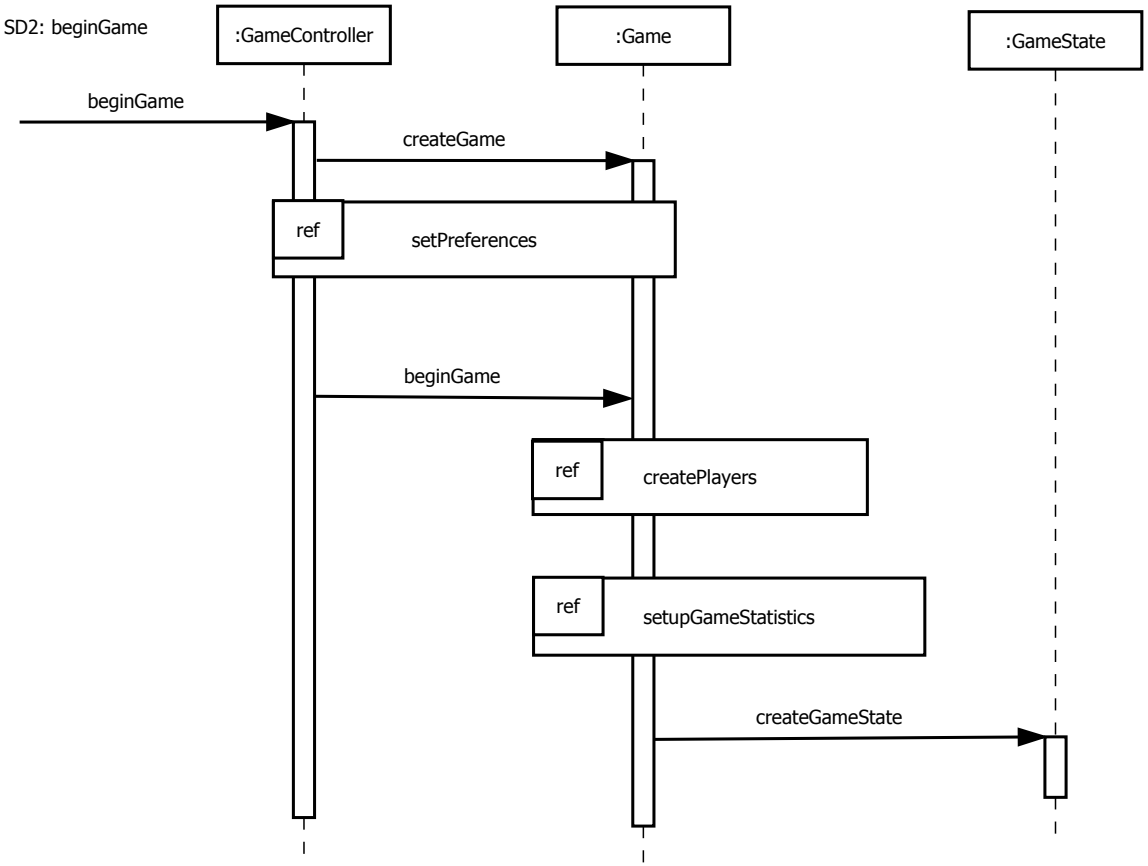
SequentialLearning

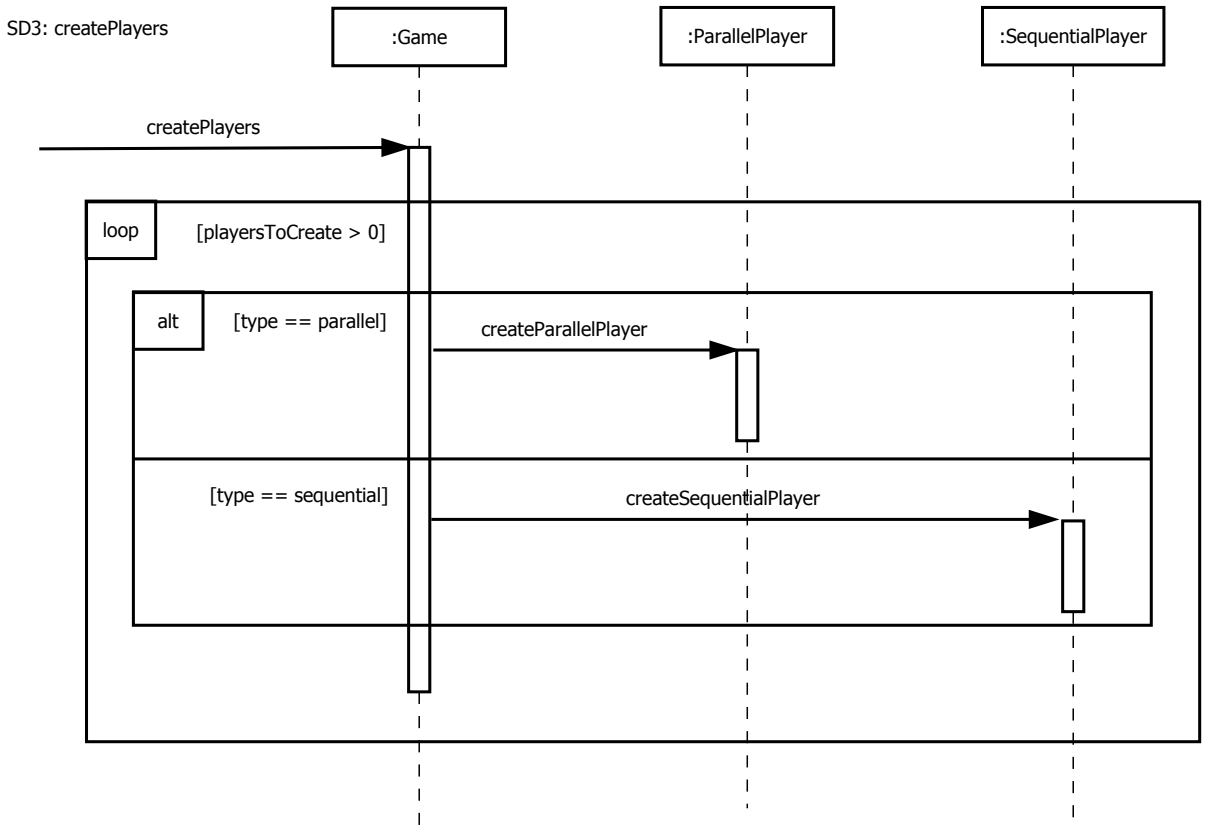
TDLambda«interface»

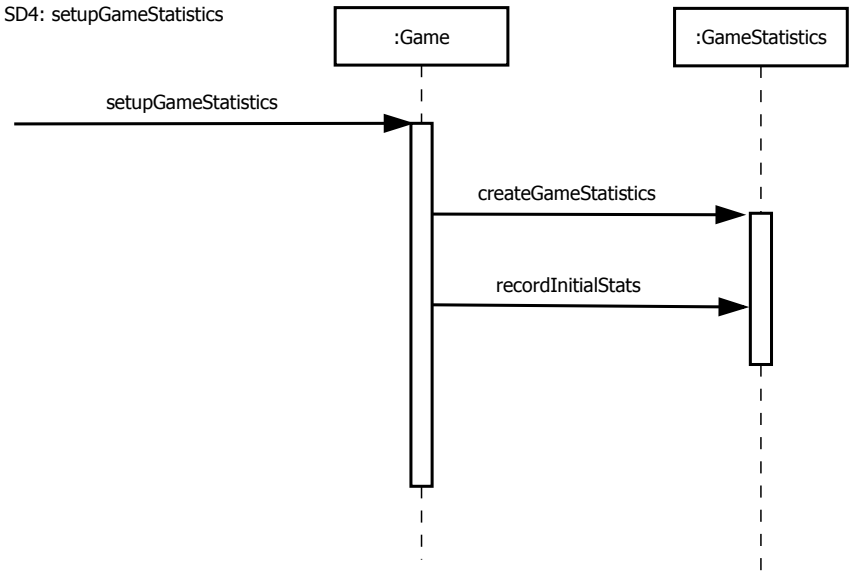
Node

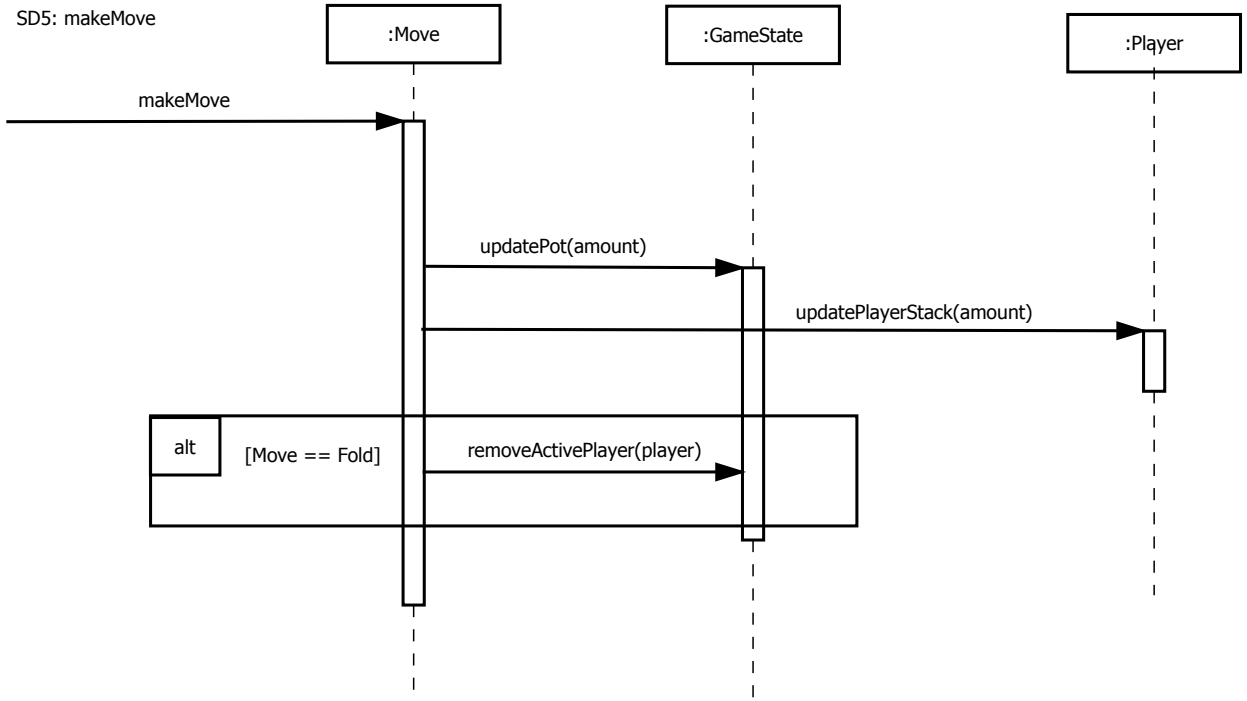
ParallelLearning

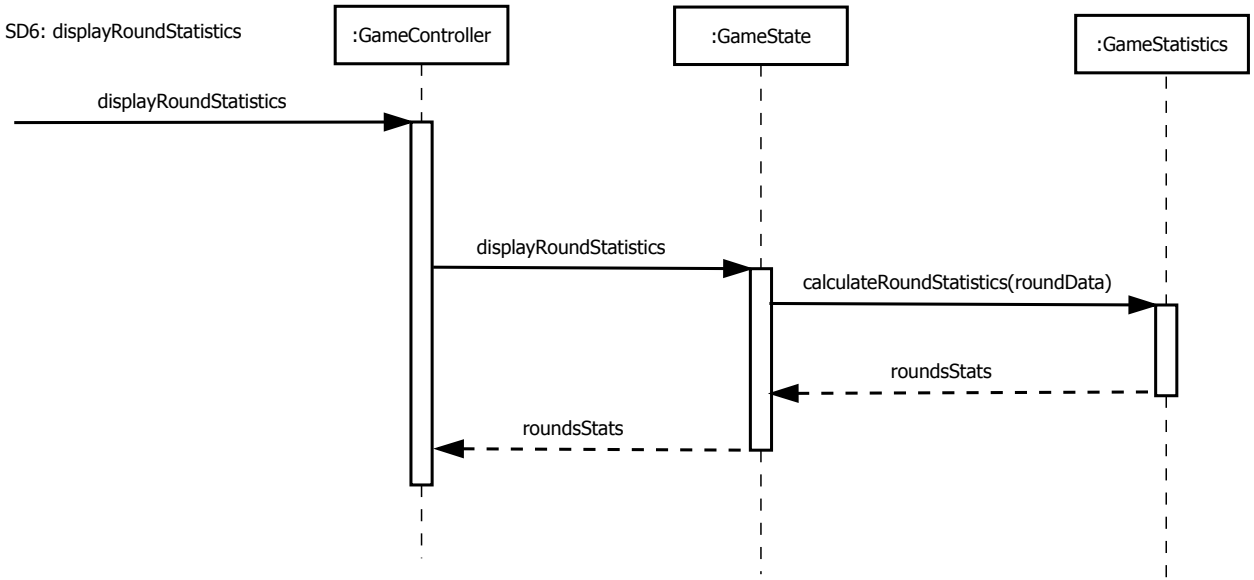

SD1: setPreferences
:GameController
:Game
setPreferences(numPlayers, playerTypes)
setNumberPlayers(numPlayers)
setPlayerTypes(playerTypes)

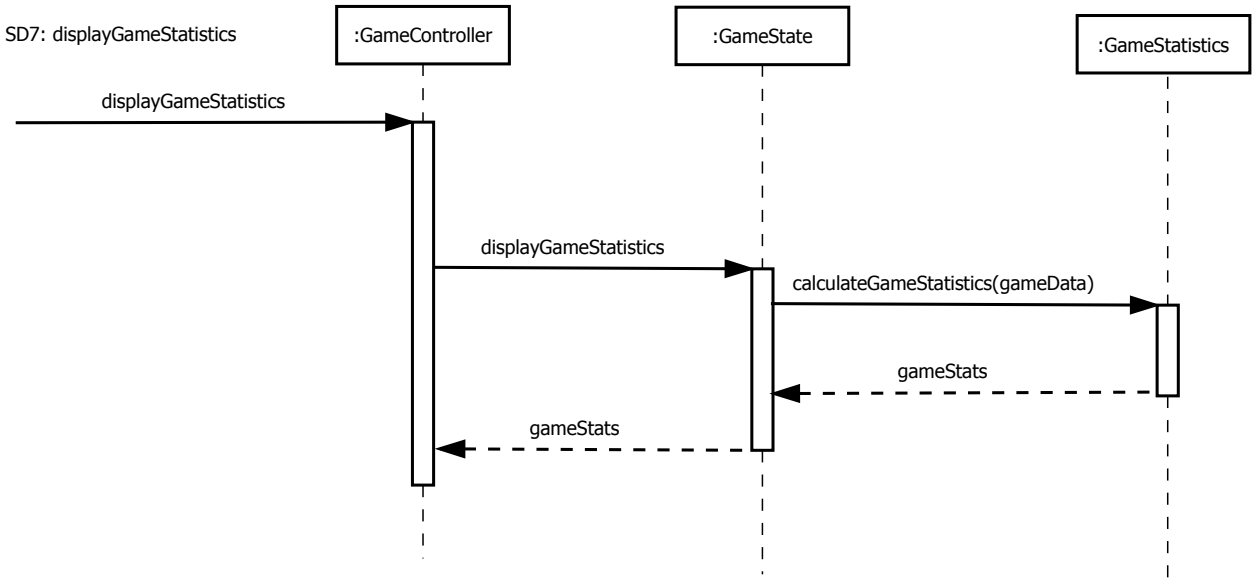

SD2: beginGame
:GameController
:Game
:GameState
beginGame
createGame
ref
setPreferences
beginGame
ref
createPlayers
ref
setupGameStatistics
createGameState

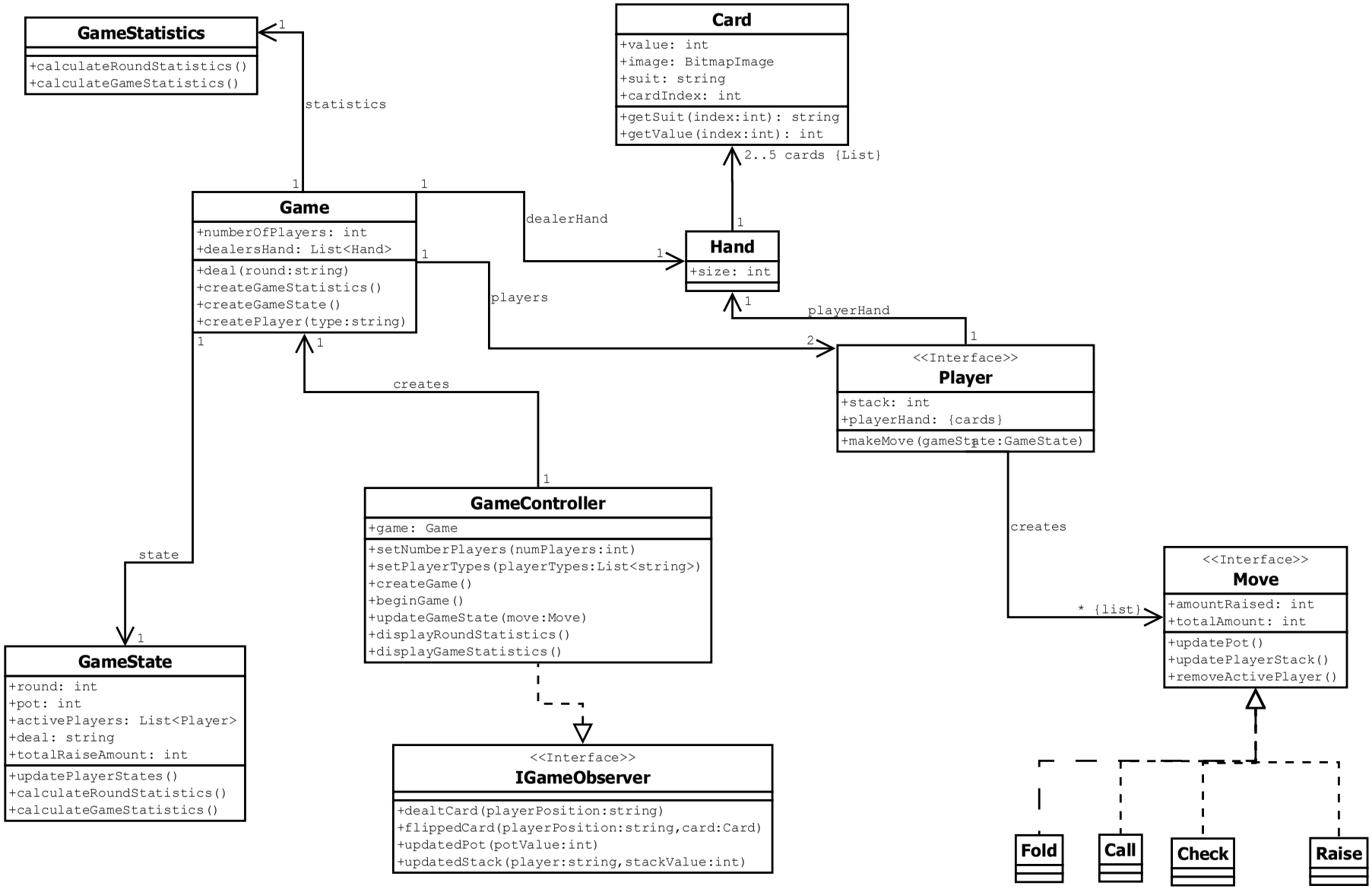

SD3: createPlayers
:Game
:ParallelPlayer
:SequentialPlayer
createPlayers
loop
[playersToCreate > 0]
alt
[type == parallel]
createParallelPlayer
[type == sequential]
createSequentialPlayer

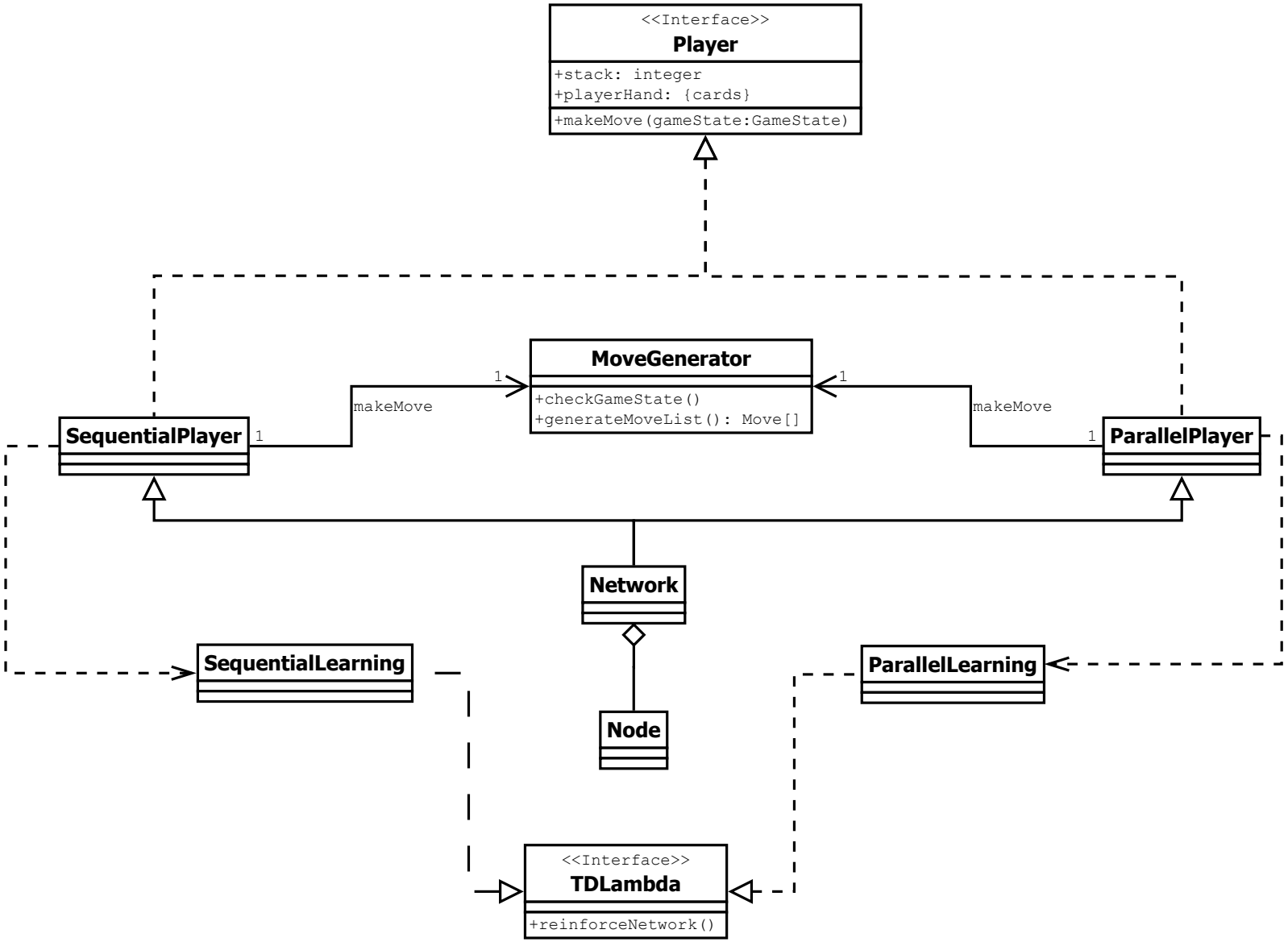

SD4: setupGameStatistics
:Game
:GameStatistics
setupGameStatistics
createGameStatistics
recordInitialStats


SD5: makeMove
:Move
:GameState
:Player
makeMove
updatePot(amount)
updatePlayerStack(amount)
alt
[Move == Fold]
removeActivePlayer(player)


SD6: displayRoundStatistics
:GameController
:GameState
:GameStatistics
displayRoundStatistics
displayRoundStatistics
calculateRoundStatistics(roundData)
roundsStats
roundsStats


SD7: displayGameStatistics
:GameController
:GameState
:GameStatistics
displayGameStatistics
displayGameStatistics
calculateGameStatistics(gameData)
gameStats
gameStats


GameStatistics
+calculateRoundStatistics()
+calculateGameStatistics()
statistics
Card
+value: int
+image: BitmapImage
+suit: string
+cardIndex: int
+getSuit(index:int): string
+getValue(index:int): int
2..5 cards {List}
Game
+numberOfPlayers: int
+dealersHand: List<Hand>
+deal(round:string)
+createGameStatistics()
+createGameState()
+createPlayer(type:string)
dealerHand
Hand
+size: int
players
playerHand
<<Interface>>
Player
+stack: int
+playerHand: {cards}
+makeMove(gameState:GameState)
creates
state
GameController
+game: Game
+setNumberPlayers(numPlayers:int)
+setPlayerTypes(playerTypes:List<string>)
+createGame()
+beginGame()
+updateGameState(move:Move)
+displayRoundStatistics()
+displayGameStatistics()
GameState
+round: int
+pot: int
+activePlayers: List<Player>
+deal: string
+totalRaiseAmount: int
+updatePlayerStates()
+calculateRoundStatistics()
+calculateGameStatistics()
<<Interface>>
IGameObserver
+dealtCard(playerPosition:string)
+flippedCard(playerPosition:string,card:Card)
+updatedPot(potValue:int)
+updatedStack(player:string,stackValue:int)
creates
* {list}
<<Interface>>
Move
+amountRaised: int
+totalAmount: int
+updatePot()
+updatePlayerStack()
+removeActivePlayer()
Fold
Call
Check
Raise


<<Interface>>
Player
+stack: integer
+playerHand: {cards}
+makeMove(gameState:GameState)
MoveGenerator
+checkGameState()
+generateMoveList(): Move[]
1
makeMove
1
1
makeMove
1
SequentialPlayer
ParallelPlayer
Network
Node
SequentialLearning
ParallelLearning
<<Interface>>
TDLambda
+reinforceNetwork()

GRASP Patterns

Low Coupling

In order to achieve low coupling, we made sure that there were no unnecessary relations between classes. For instance, we considered having Card related to Game and Hand. Since we considered the Low Coupling pattern, we realized that we only needed Card to be coupled with Hand, since Card was already coupled with Hand and Hand was already coupled with Game.

High Cohesion

In order to have high cohesion, we made sure that our classes all have a number of highly related methods that do not too many operations. For instance, the only method that Player has is makeMove(). Before, we had considered putting methods about calculations of statistics in Player. However, based on the High Cohesion principle, we decided to have only highly related methods in Player.

Information Expert

We used the Information Expert pattern to identify what class should have any operations dealing with changing the GameStatistics class. Since Game has all the information about the game, such as number of players and player types, then Game was the appropriate choice based on the Information Expert pattern. Therefore, Game has the operations related to modifying and creating GameStatistics.

Creator

We used the Creator principle to decide what class should create the Players. Since the Game contains the Players, closely uses the Players, and has the data to initialize the Players, Game was a good choice for the Creator of the players (SD3).

Controller

We used the Controller pattern to decide what class should coordinate system operations. We decided to use a Façade Controller, because the operations are coming in over a single pipeline. Namely, all the operations are coming directly from the GUI. In addition, there are not many system operations. There really is only two main ones: setPreferences (SD1) and beginGame (SD2). Therefore, a Façade Controller was appropriate. Our Façade Controller is our GameController class. It delegates tasks to other objects and controls the activity in the system.

Polymorphism

We used Polymorphism when deciding how to deal with the different types of players. Since ParallelPlayer and SequentialPlayer were very similar but have a few different behaviors, we decided to have an abstract class, Player, and then two subclasses, ParallelPlayer and SequentialPlayer. This will also be useful for any future versions of the system. If a developer wants to add a different type of player, he can easily add another subclass to Player.

Indirection

Since we did not want to have Player directly coupled with the GameState, we used the Indirection pattern. We used an intermediate object, Game, to mediate between GameState and Player. That way, we did not have to have extra coupling between Player and GameState.

Pure Fabrication

Information Expert would lead us into high coupling with regard to presenting the details of the poker game to the user. For instance, each Player would have to output its data about the move decision to the GUI and the game would also have to be related to the GUI. Instead of following the Information Expert pattern for game statistics, we used the Pure Fabrication pattern. We made a GameStatistics class. GameStatistics is used to summarize the statistics about the game without having too high coupling. We were careful to avoid a common contraindication of Pure Fabtrication, which is too much data being passed to other objects for calculations. We made sure that GameStatistics does not request many calculations to be done by other classes.

Protected Variations

The most unstable part of the system is the neural network. Therefore, we used the Protected Variations pattern to create a stable interface around the neural network. We created a MoveGenerator class that is between the system components and the neural network. That way, if the neural network needs to be changed in the future, it can easily be altered without affecting the whole system.