Milestone 5

Concurrent Poker Player Team (23)

Mark Jenne, Ian Roberts, Bennie Waters, Sarah Jabon

2/19/2010

Table of Contents

Contents

Introduction

The goal of this project was to design a Texas Hold ‘Em poker application with which a user can interact. The user would be able to either watch a simulation of a poker game or actually participate in the game itself. The application will be used within Microsoft as a sample application to demonstrate certain aspects of the current .NET framework. The application was to be written in either C++ or C#, and Visual Studio 2010 Beta 2 was to be the integrated development environment.

Analysis Models

Domain Model

The domain model is a visual representation of conceptual classes and their relationships. Our domain model explains a Texas Hold'em Poker game.

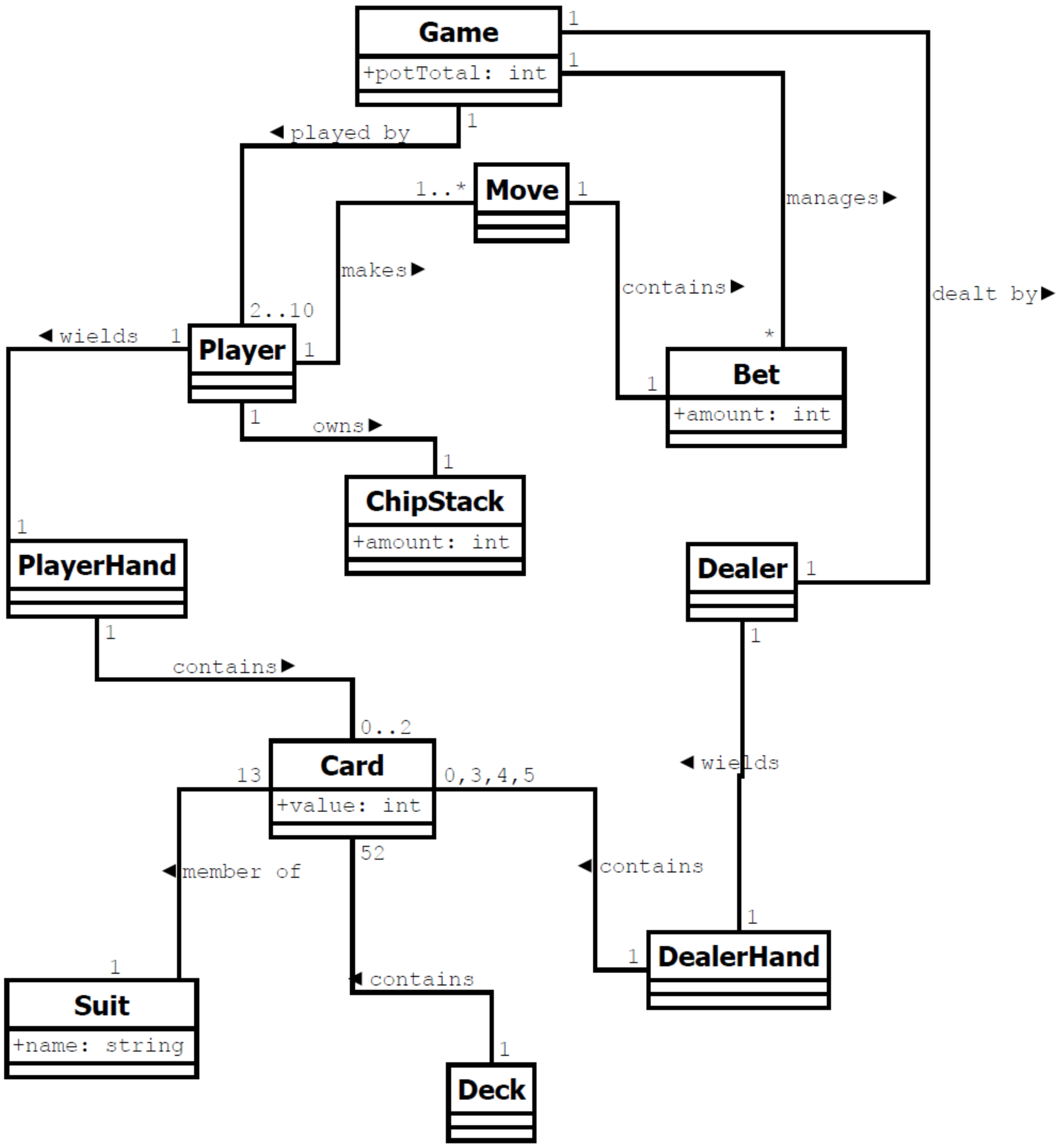


Figure 1: Domain Model

The Game class represents the entire poker game. The Game class has one attribute, potTotal, 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 typical Texas Hold'em Poker game can have anywhere from two to 10 participants, so the Game is played by two to 10 Players.

A Deck contains all the Cards that are in the poker game. There are 52 cards in a typical deck of playing cards, so there are 52 Cards in the Deck.

These Cards will make up hands. There are two different kinds of hands – PlayerHands and DealerHands. A PlayerHand has zero or two Cards. A DealerHand consists of zero, three, four, or five Cards. The amount of Cards in both the PlayerHand and the DealerHand depends on the current deal in the game.

A Dealer wields one DealerHand.

A Player wields one PlayerHand. A Player also owns one ChipStack, which has an attribute, amount. Amount represents the amount of money that the Player can use to bet in the poker game. The Player makes Moves throughout the game. The Player must make at least one Move when playing the game. These Moves have Bets. The Bets have an attribute, amount, which represents the amount of money the Player bet.

System Sequence Diagrams

System sequence diagrams represent the interaction between the user and the system as a whole. We have three system sequence diagrams that each correspond to a Use Case. They represent the user's interaction involved in starting the game, setting the preferences for the game, and playing the game.

System Sequence Diagram #1 – Begin Game

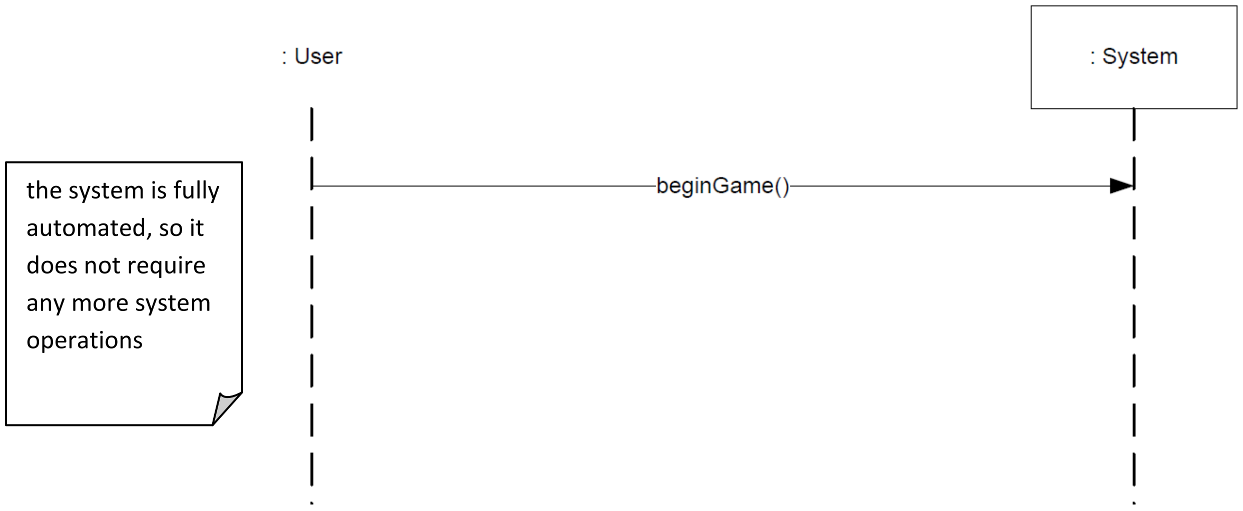
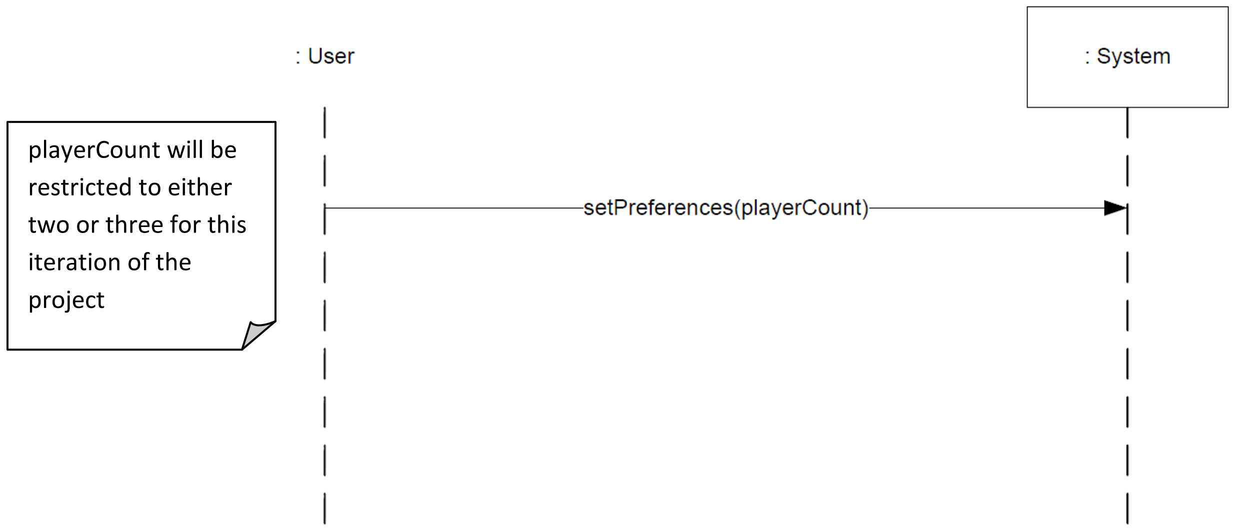

: User
: System
beginGame()
the system is fully automated, so it does not require any more system operations

Figure 2: SSD #1 - Start Game

This system sequence diagram corresponds to Use Case #1, which describes the user simply starting the game. In order to begin the game, the user simply sends in a message to the system to initiate play. Since the system is fully automated, no more system operations are necessary.

System Sequence Diagram #2 – Set Preferences



: User
: System
playerCount will be restricted to either two or three for this iteration of the project
setPreferences(playerCount)

Figure 3: SSD #2 - Set Preferences

This system sequence diagram corresponds to Use Case #2, which describes the user setting the preferences for the game. In this iteration of the project, the user can choose to have two or three players in the game. If the user chooses three players, the user will be involved in the game, and Use Case #3 will be relevant.

System Sequence Diagram #3 – Play Game

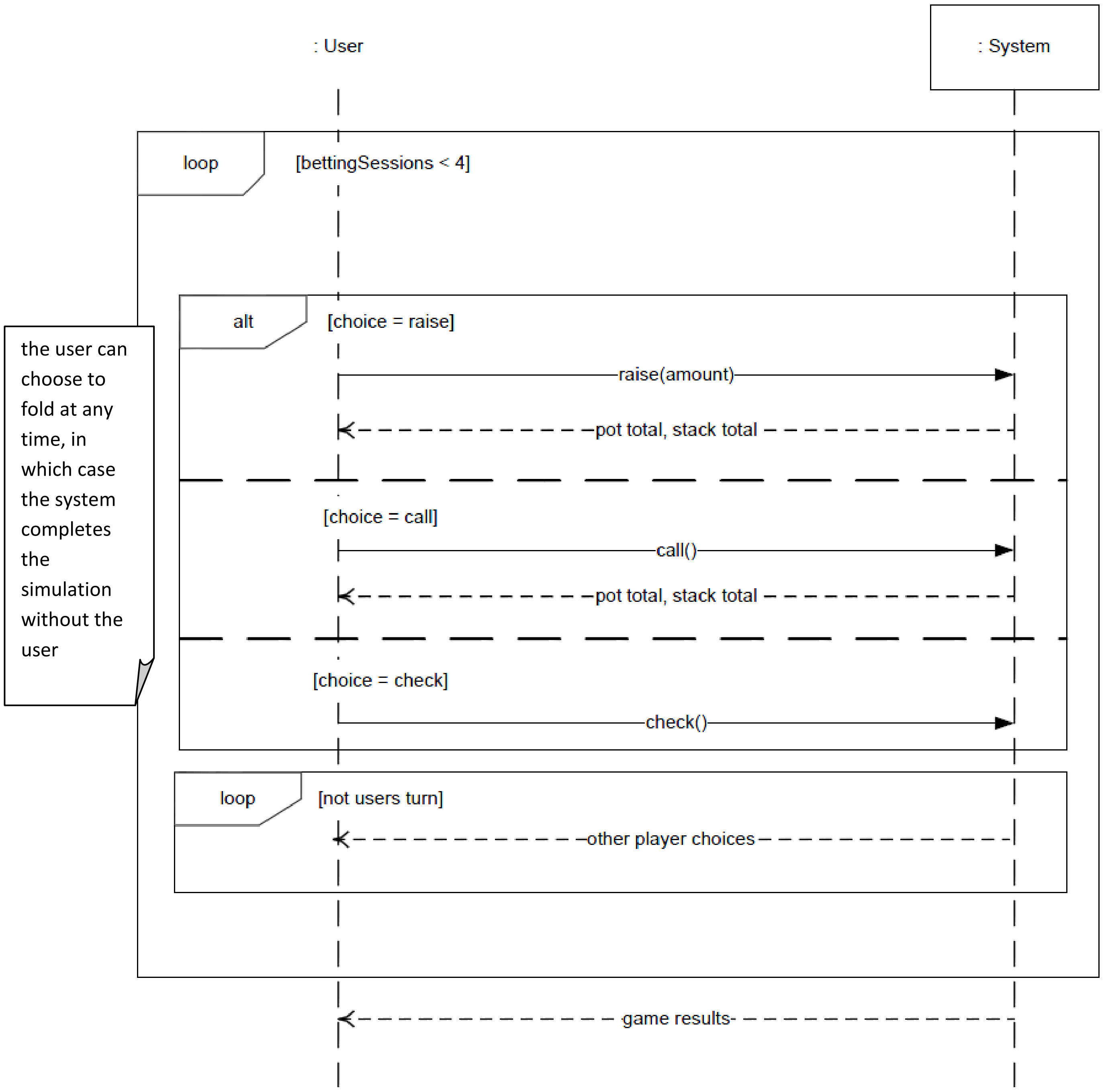


Figure 4: SSD #3 - Play Game

This system sequence diagram corresponds to Use Case #3, which describes the user playing the game. The user can raise, check, call, or fold. If the user raises, checks, or calls, the game continues with the user in the game. If the user folds, then the system simulates the rest of the game with the other players. Finally, the system displays the results of the game which describe the end state of the game.

Operation Contracts

Operation contracts provide more detail for the system operations in the system sequence diagrams. We elaborate on the following system operations with operation contracts: startGame(), setPreferences(playerCount), raise(amount), call(),check(), and fold().

Operation Contract #1: beginGame

Operation:	beginGame()
Cross References:	System Sequence Diagram #1: Start Game
	Use Case #1: Start Game
Preconditions:	The application has initialized properly
Postconditions:	An instance of Game, *game*, was created

Operation Contract #2: setPreferences

Operation:	setPreferences(playerCount)
Cross References:	System Sequence Diagram #2: Set Preferences
	Use Case #2: 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

Operation Contract #3: 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

Operation Contract #4: call

Operation:	call()
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 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

Operation Contract #5: check

Operation:	check()
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:	*player[i]*'s turn ended *player[i+1]*'s turn began

Operation Contract #6: fold

Operation:	fold()
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:	*player[i]*'s turn ended *player[i]* was removed from *players{list}* *player[i+1]*'s turn began

Logical Architecture

Package Diagram

Package diagrams describe the grouping of elements. Our package diagram represents how the elements in the system are grouped.

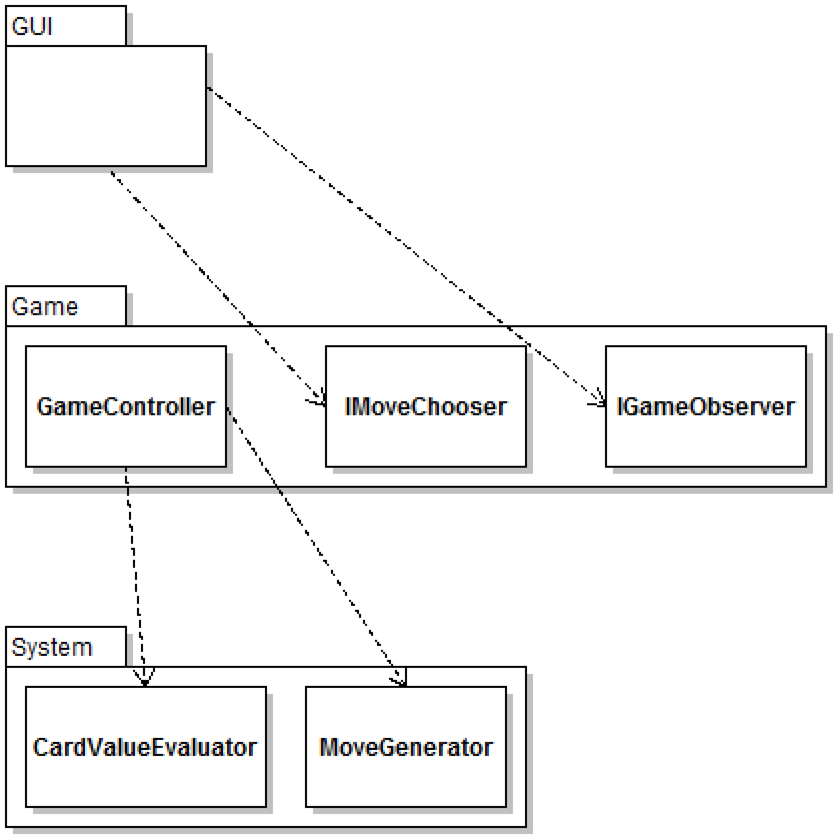


Figure 5: Package Diagram

There are three main layers in the package diagram: GUI, Game, and System. The GUI layer is what the user interacts with. We have two interfaces that the GUI implements: IMoveChoiceStrategy and IGameObserver. IGameObserver enables the GUI to be updated based on the current game state. IMoveChoiceStrategy helps the system get a move choice from the user. The GameController is in the domain, where the game actually happens. The System is the lower level calculators and evaluators. We have a MoveGenerator that generates legal moves based on the game state, and a CardValueEvaluator that evaluates the value of a set of cards.

Sequence Diagrams

Sequence diagrams are interaction diagrams, which illustrate how objects interact via messages within the system.

Begin Game

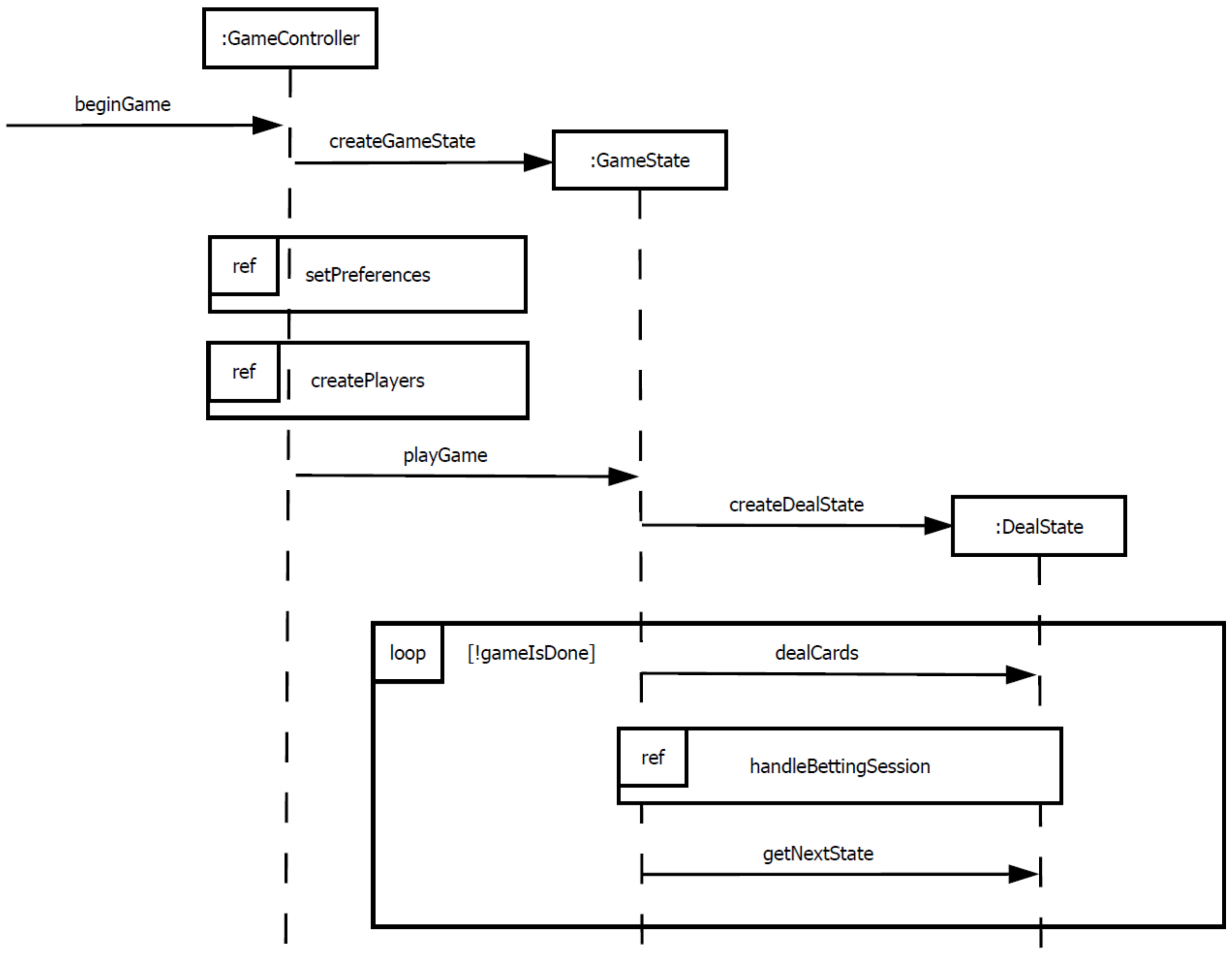


Figure 6: SD #1 - Begin Game

This sequence diagram corresponds to the beginGame operation found in operation contract 1, which describes the creation of a game. Here the GameController sets the preferences input by the user and creates the players based on those preferences. A GameState is initialized, and then the GameController signals the GameState to play the game. The heart of the game play is a loop in which the current DealState deals cards to the dealer hand, goes through a betting session, and moves to the next Deal State. The Controller pattern is visible here as the GameController object coordinates system operations beyond the user interface (MainWindow) layer. Establishing GameController as a Façade Controller was appropriate since there are a select few system operations generated from the user interface and since they are all coming in over a single pipeline. The GameController class then delegates tasks to over objects in order to control the activity of the system. This interaction diagram also demonstrates how the Creator pattern has influenced our design. Since the GameController has a GameState object, it is the most viable choice for handling the creation of the GameState. This also applies for GameState and DealState. The GameState object contains a DealState object, so it will create a DealState object when needed. For further analysis related to the GRASP and GoF patterns, see the section entitled "Design Principles and Patterns."

Set Preferences

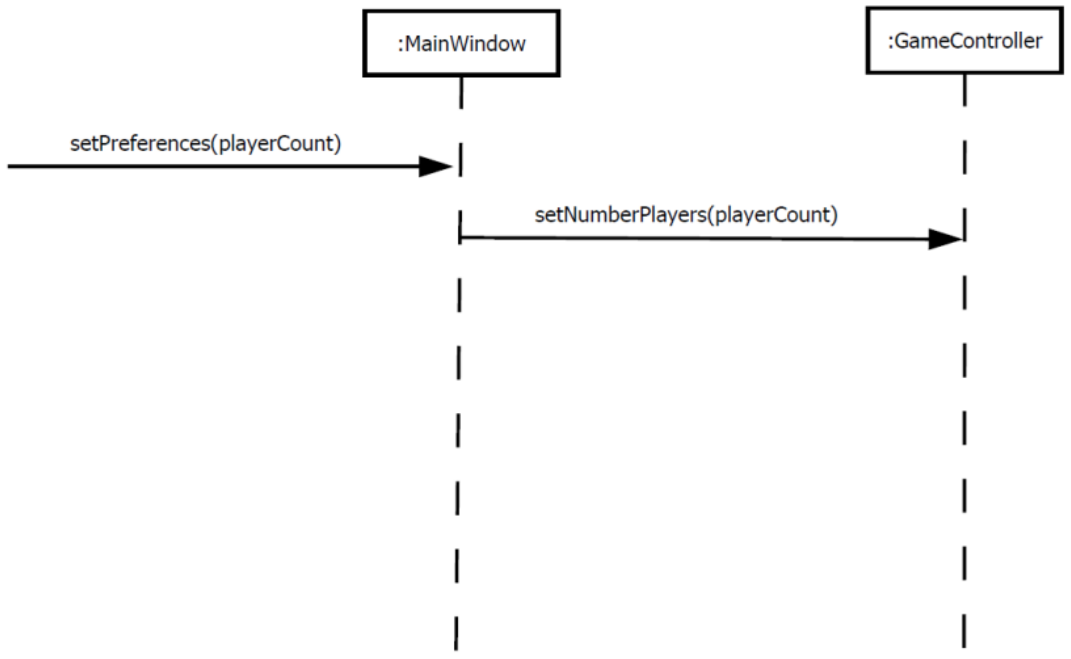


Figure 7: SD #2 – Set Preferences

This interaction diagram shows one of the system operations that originated from the user interface and is passed into the domain layer to the GameController. Here the MainWindow is receiving a message regarding the number of players involved in the game (currently regulated to be only two or three), which is then piping that information to the GameController, which will incorporate this preference into the domain logic of the game. This further exhibits the Controller pattern in action by showing the single pipeline through the GameController that connects the user interface layer with the domain layer and how messages are passed between layers. For further analysis related to the GRASP and GoF patterns, see the section entitled "Design Principles and Patterns."

Create Players

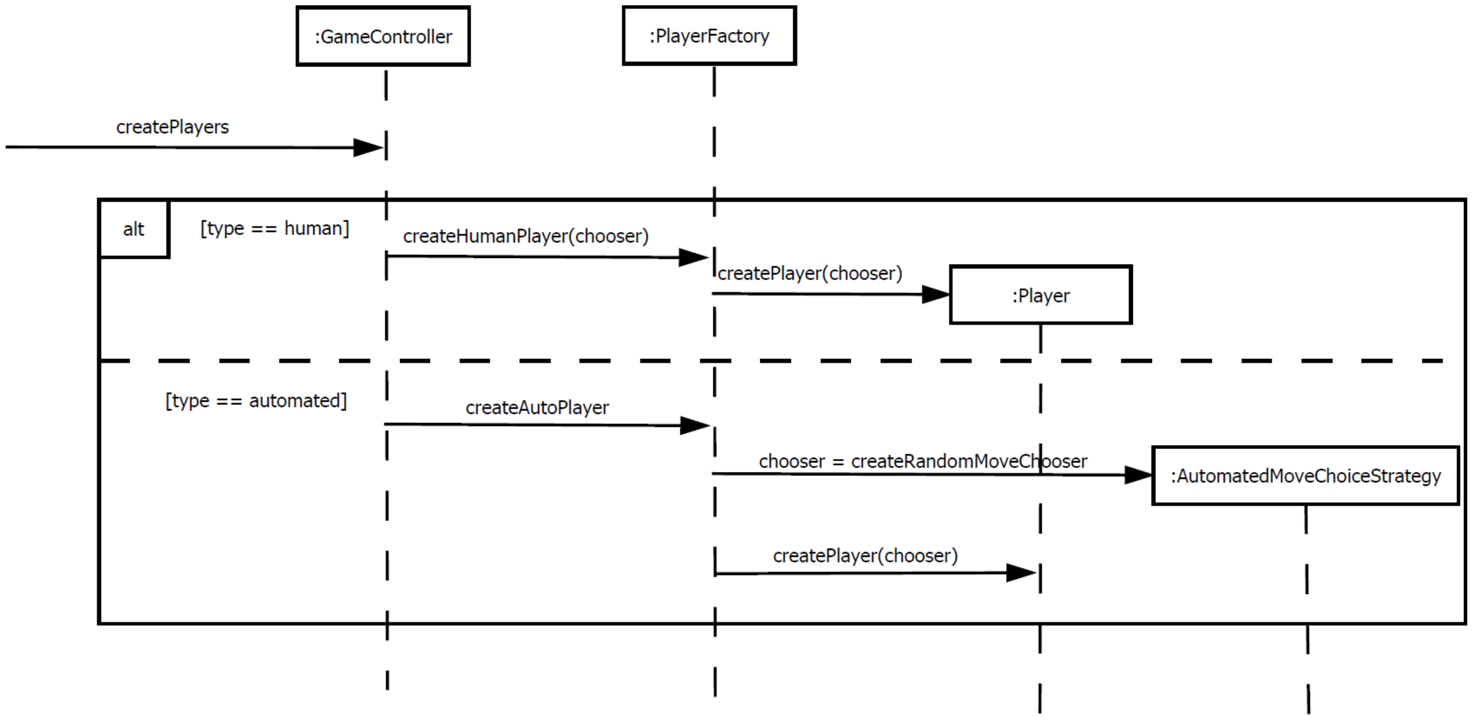


Figure 8: SD #3 – Create Players

This interaction diagram reveals the internal communication for creating players. Currently, the player types are dependent on the number of players in the game. If two players are involved, both will be automated. Should three players be in the game, then two will be automated and the remaining player will be controlled by the user. The GameController handles the process of creating the players, with the different types being differentiated by the instance of the IMoveChoiceStrategy interface each implements. The Factory pattern is appropriate here because the logic involved in the creation of the players differs based on player type. Thus the responsibility of actually creating the players with consideration for varying creation logic is given to a PlayerFactory class. The Strategy pattern can also be seen in this interaction diagram in the AutomatedMoveChoiceStrategy object that is created for an automated player. Here AutomatedMoveChoiceStrategy is an implementation of the IMoveChoiceStrategy interface for automated players. For further analysis related to the GRASP and GoF patterns, see the section entitled “Design Principles and Patterns.”

Handle Betting Session

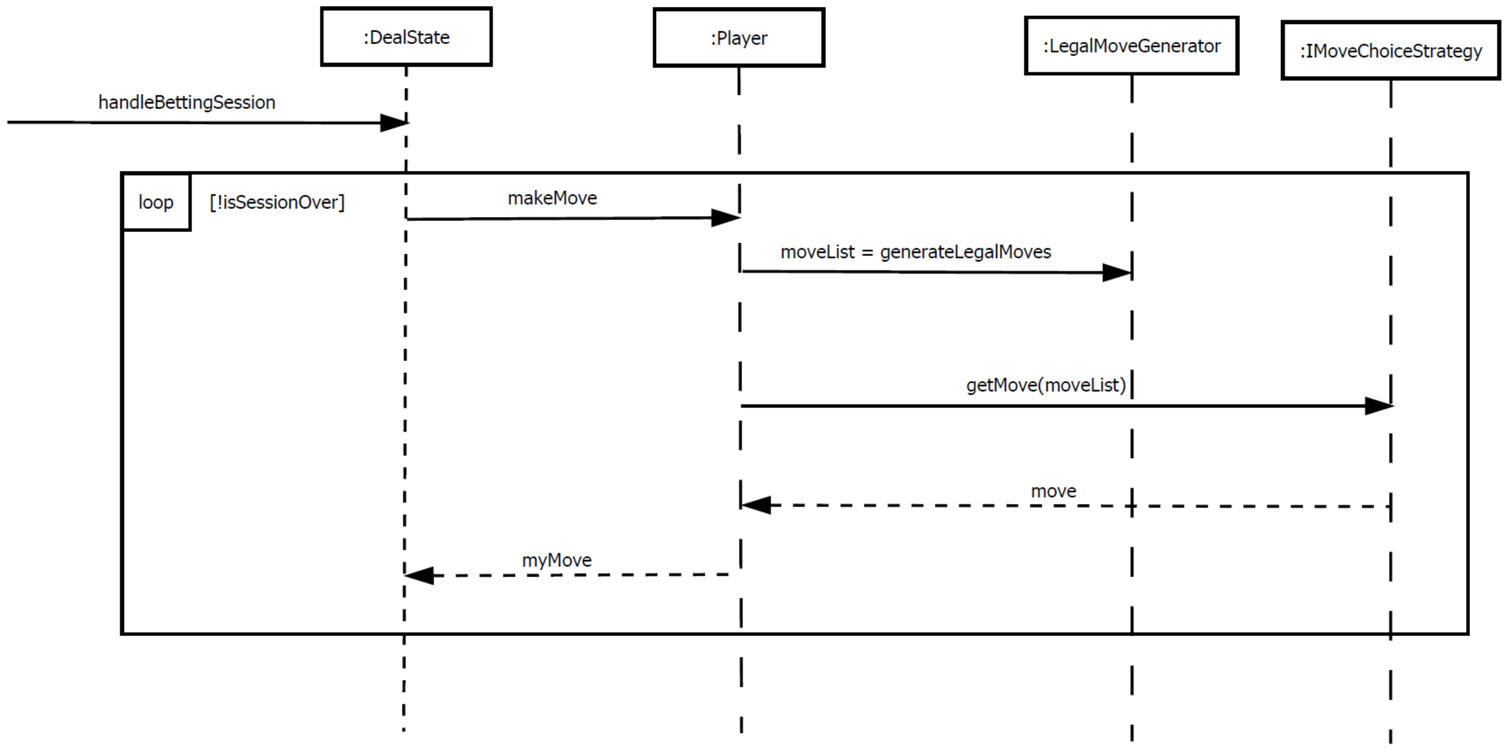


Figure 9: SD #4 – Handle Betting Session

This interaction diagram shows the sequence of system operations that occur when the handleBettingSession operation is invoked. Within the logic of the game, a betting session can go on for an undetermined, though limited, number of moves. The DealState controls the betting session and goes through all of the players requesting their move. Several design patterns are apparent from this interaction diagram. Information Expert was used in deciding how changes in the game state (i.e. moves) should be made. Here the player is told to make a move, but then consults a legal move generator for a list of possible moves that it can make, which was added to the system in considering the Pure Fabrication pattern. The implementation of IMoveChoiceStrategy for that player is then used to pick which move to make from the list generated. The use of Information Expert has lead to high cohesion and low coupling among the operations involved here. Also apparent is the Strategy pattern. IMoveChoiceStrategy is a Strategy interface that is implemented by different classes that determine which move from the list of legal moves should be made. Strategy over Polymorphism is appropriate here because the different algorithms for choosing a move are related, but contain significant variation, which could not be accounted for solely through Polymorphism. For further analysis related to the GRASP and GoF patterns, see the section entitled “Design Principles and Patterns.”

Logical Design

Design Class Diagrams

Design class diagrams are used to visualize the domain. They display attributes and associations of all the classes in the domain. We have three design class diagrams. The first has no methods and very few attributes. It is solely to display the relationships between classes. The second design class diagram is a subset of the domain. It exhibits the relationships between the MainWindow, GameController, GameState, and DealStates. The third design class diagram shows how the Player, Move, IMoveChoiceStrategy, and MainWindow interact.

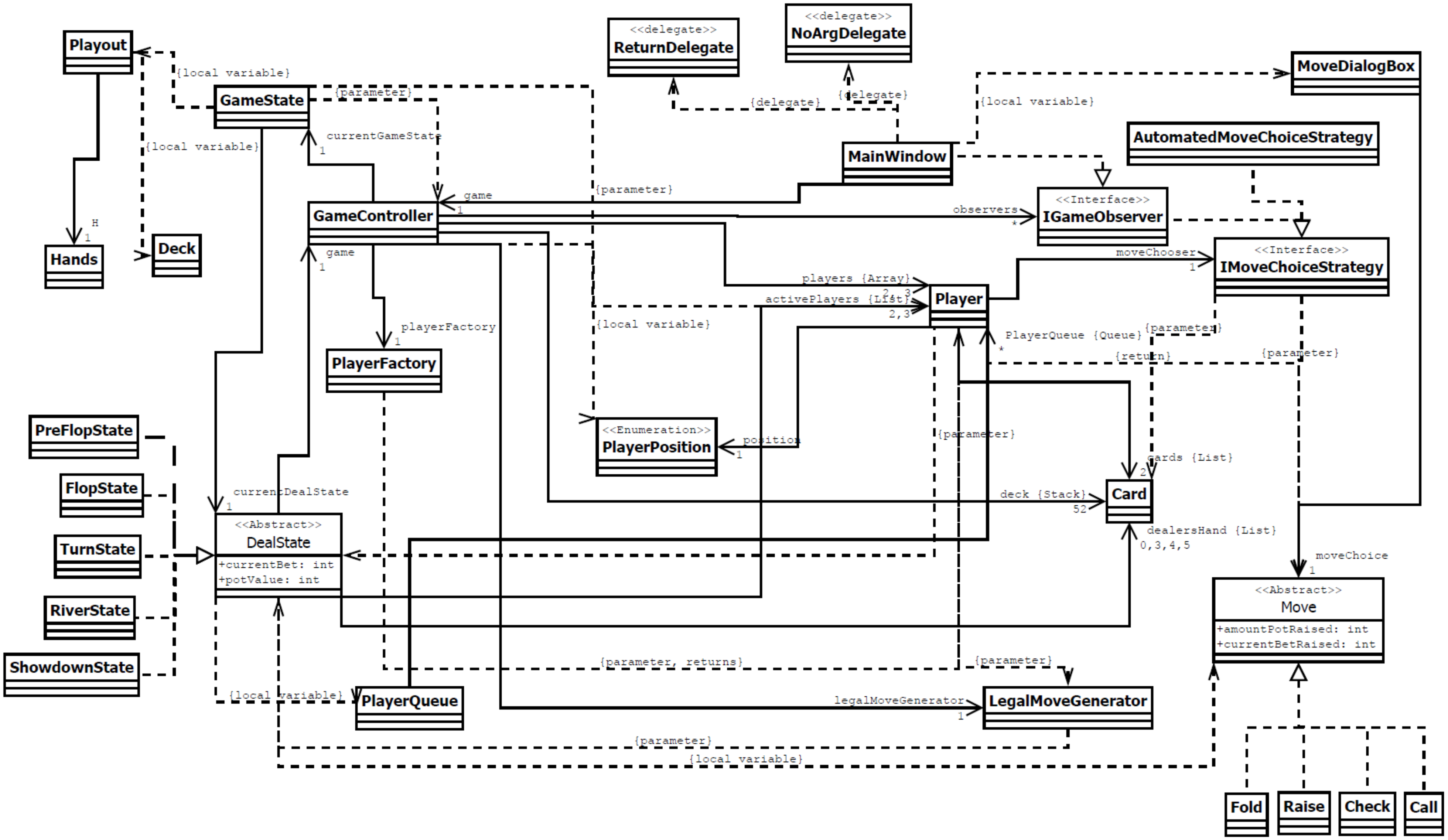
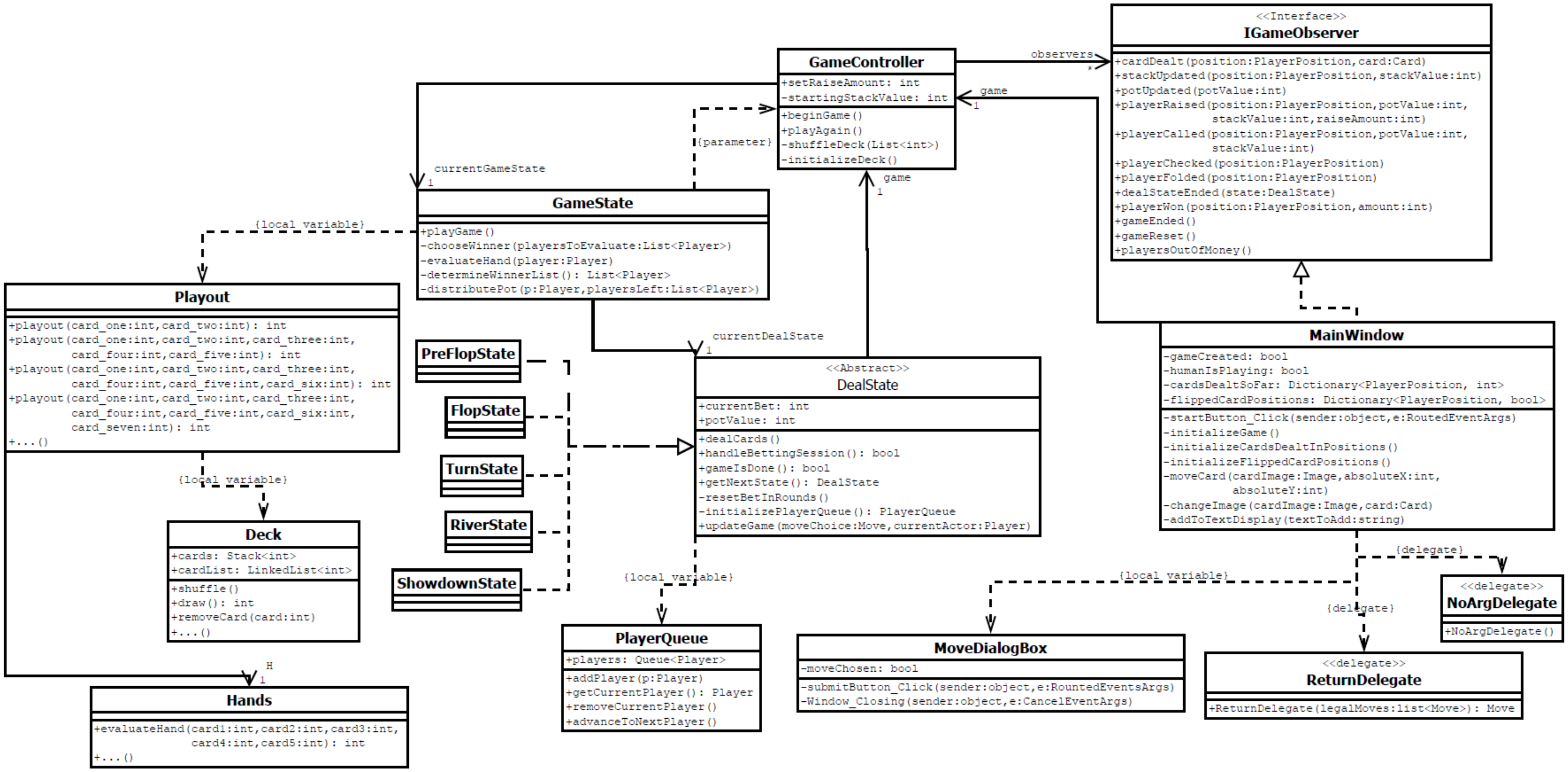


Figure 6: Relational Design Class Diagram



GameController
+setRaiseAmount: int
-startingStackValue: int
+beginGame()
+playAgain()
-shuffleDeck(List<int>)
-initializeDeck()
observers
*
game
1
{parameter}
<<Interface>>
IGameObserver
+cardDealt(position:PlayerPosition,card:Card)
+stackUpdated(position:PlayerPosition,stackValue:int)
+potUpdated(potValue:int)
+playerRaised(position:PlayerPosition,potValue:int, stackValue:int,raiseAmount:int)
+playerCalled(position:PlayerPosition,potValue:int, stackValue:int)
+playerChecked(position:PlayerPosition)
+playerFolded(position:PlayerPosition)
+dealStateEnded(state:DealState)
+playerWon(position:PlayerPosition,amount:int)
+gameEnded()
+gameReset()
+playersOutOfMoney()
currentGameState
1
GameState
+playGame()
-chooseWinner(playersToEvaluate:List<Player>)
-evaluateHand(player:Player)
-determineWinnerList(): List<Player>
-distributePot(p:Player,playersLeft:List<Player>)
{local variable}
Playout
+playout(card_one:int,card_two:int): int
+playout(card_one:int,card_two:int,card_three:int, card_four:int,card_five:int): int
+playout(card_one:int,card_two:int,card_three:int, card_four:int,card_five:int,card_six:int): int
+playout(card_one:int,card_two:int,card_three:int, card_four:int,card_five:int,card_six:int, card_seven:int): int
+...()
{local variable}
Deck
+cards: Stack<int>
+cardList: LinkedList<int>
+shuffle()
+draw(): int
+removeCard(card:int)
+...()
H
1
Hands
+evaluateHand(card1:int,card2:int,card3:int, card4:int,card5:int): int
+...()
PreFlopState
FlopState
TurnState
RiverState
ShowdownState
currentDealState
1
<<Abstract>>
DealState
+currentBet: int
+potValue: int
+dealCards()
+handleBettingSession(): bool
+gameIsDone(): bool
+getNextState(): DealState
-resetBetInRounds()
-initializePlayerQueue(): PlayerQueue
+updateGame(moveChoice:Move,currentActor:Player)
{local variable}
PlayerQueue
+players: Queue<Player>
+addPlayer(p:Player)
+getCurrentPlayer(): Player
+removeCurrentPlayer()
+advanceToNextPlayer()
MainWindow
-gameCreated: bool
-humanIsPlaying: bool
-cardsDealtSoFar: Dictionary<PlayerPosition, int>
-flippedCardPositions: Dictionary<PlayerPosition, bool>
-startButton_Click(sender:object,e:RoutedEventArgs)
-initializeGame()
-initializeCardsDealtInPositions()
-initializeFlippedCardPositions()
-moveCard(cardImage:Image,absoluteX:int, absoluteY:int)
-changeImage(cardImage:Image,card:Card)
-addToTextDisplay(textToAdd:string)
{local variable}
MoveDialogBox
-moveChosen: bool
-submitButton_Click(sender:object,e:RountedEventsArgs)
-Window_Closing(sender:object,e:CancelEventArgs)
{delegate}
{delegate}
<<delegate>>
NoArgDelegate
+NoArgDelegate()
<<delegate>>
ReturnDelegate
+ReturnDelegate(legalMoves:list<Move>): Move

Figure 7: Game and GUI Design Class Diagram

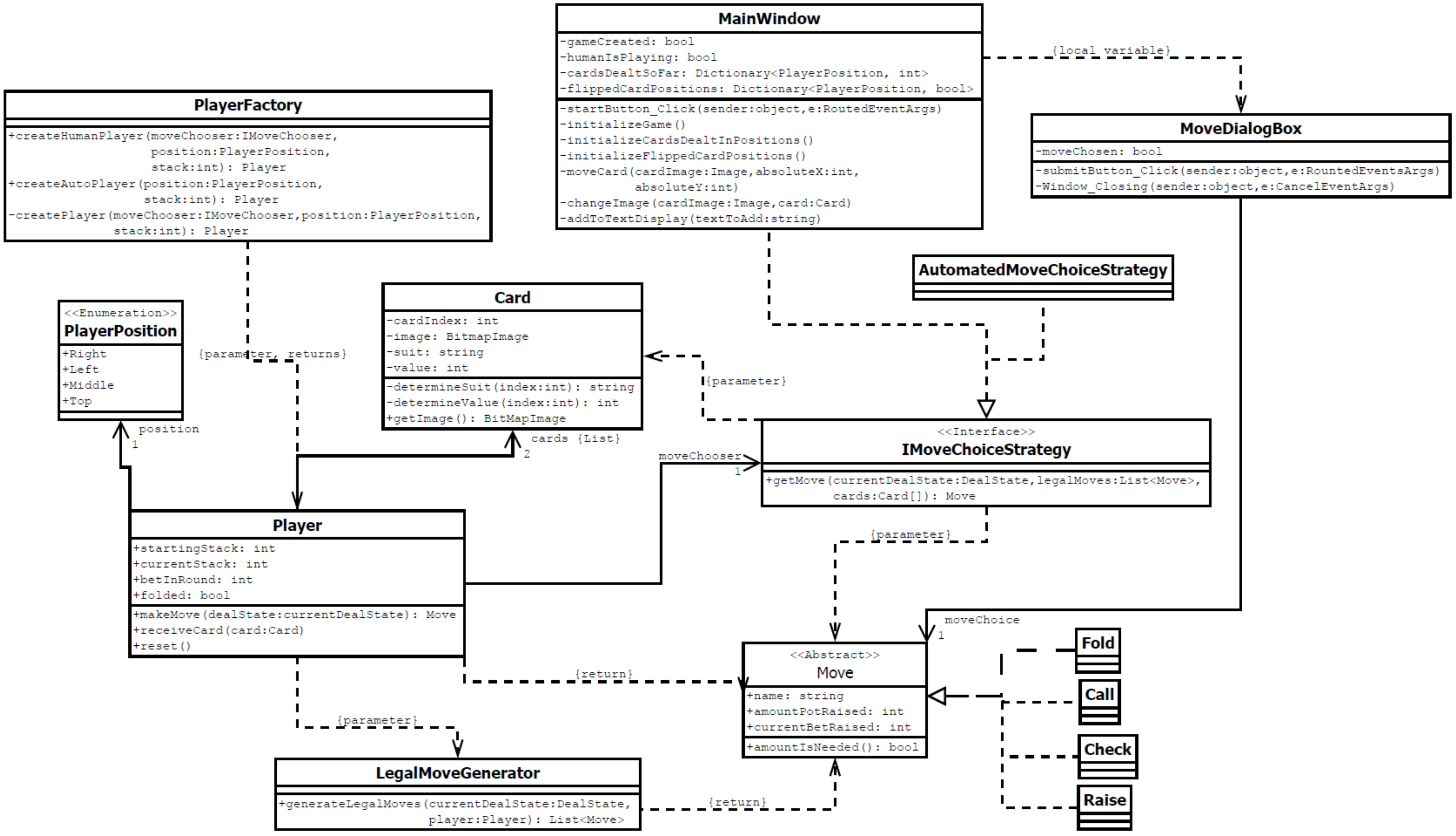

MainWindow
-gameCreated: bool
-humanIsPlaying: bool
-cardsDealtSoFar: Dictionary<PlayerPosition, int>
-flippedCardPositions: Dictionary<PlayerPosition, bool>
-startButton_Click(sender:object,e:RoutedEventArgs)
-initializeGame()
-initializeCardsDealtInPositions()
-initializeFlippedCardPositions()
-moveCard(cardImage:Image,absoluteX:int, absoluteY:int)
-changeImage(cardImage:Image,card:Card)
-addToTextDisplay(textToAdd:string)
{local variable}
MoveDialogBox
-moveChosen: bool
-submitButton_Click(sender:object,e:RountedEventsArgs)
-Window_Closing(sender:object,e:CancelEventArgs)
PlayerFactory
+createHumanPlayer(moveChooser:IMoveChooser, position:PlayerPosition, stack:int): Player
+createAutoPlayer(position:PlayerPosition, stack:int): Player
-createPlayer(moveChooser:IMoveChooser,position:PlayerPosition, stack:int): Player
AutomatedMoveChoiceStrategy
<<Enumeration>>
PlayerPosition
+Right
+Left
+Middle
+Top
{parameter, returns}
Card
-cardIndex: int
-image: BitmapImage
-suit: string
-value: int
-determineSuit(index:int): string
-determineValue(index:int): int
+getImage(): BitMapImage
{parameter}
<<Interface>>
IMoveChoiceStrategy
+getMove(currentDealState:DealState,legalMoves:List<Move>, cards:Card[]): Move
position
1
cards {List}
2
moveChooser
1
Player
+startingStack: int
+currentStack: int
+betInRound: int
+folded: bool
+makeMove(dealState:currentDealState): Move
+receiveCard(card:Card)
+reset()
{parameter}
moveChoice
1
<<Abstract>>
Move
+name: string
+amountPotRaised: int
+currentBetRaised: int
+amountIsNeeded(): bool
{return}
Fold
Call
Check
Raise
{parameter}
LegalMoveGenerator
+generateLegalMoves(currentDealState:DealState, player:Player): List<Move>
{return}

Figure 8: Player and Move Design Class Diagram

GRASP Principles

Low Coupling

In order to achieve low coupling, we made sure that there were no unnecessary relations between classes. For instance, we considered having the Player objects related to GameState. Since we considered the Low Coupling design principle, we realized that we only needed Player to be coupled with DealState, since GameState knows about the current DealState, only having DealState know about the Player objects prevented unnecessary coupling from arising in our system.

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 perform too many operations. For instance, the only method that Player has is makeMove(). Before, we had considered putting methods about deciding which moves were legal and then for deciding which move to make in Player, but decided that this would make the player class less cohesive. As such, we pushed these specialized operations to other classes which merely communicate with the Player class.

Information Expert

Information Expert is another fairly general principle that we have taken into consideration throughout most of the design of our system. In considering Information Expert we have attempted to assign all responsibilities within the system to the classes that have the necessary information to perform those responsibilities. The use of the Information Expert design principle can be seen through the interaction of GameState and DealState. Since a DealState represents an individual deal state during a game, this design principle helped us in deciding that is DealState that should govern the moves made in a deal state rather than the GameState class. Another example of Information Expert having influence on our design can be seen in how each Player object knows if it has folded or not in the current game.

Creator

We have take the Creator pattern into consideration when determining which classes should be responsible for creating instances of other classes. An example of where the Creator pattern was considered in the design of our system is in the GameController being the Creator of a GameState. When the user selects to play a game, that message is passed into the domain layer through the GameContoller. Since the GameController closely uses the GameState in order to keep the user interface updated through the use of an observer class (IGameObserver) and GameController has the data to initialize a GameState, we decided that GameController should be the class to create a GameState object.

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 are 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

In designing our system we came up with a few places where alternative classes would be needed based on type. One of those points within the system ended up being better suited by use of the Strategy pattern. However, the DealState and Move classes still needed polymorphism to function properly. Within our system, there are various alternative moves that can be made, but which all follow the same general operations. The same can be said about the alternative deal states that exist in a game of poker. In considering polymorphism in the design of our system, both Move and DealState proved to be points in our system very suited for the use of polymorphism.

Pure Fabrication

In order to maintain high cohesion and low coupling among the various classes involved in deciding what move a Player object should make, we introduced an artificial class called LegalMoveGenerator. This class provides a cohesive set of responsibilities for determining the set of all of moves that a player can legally make during a turn. Pure Fabrication was considered here because without generating an artificial class to handle this responsibility, either Player or the IMoveChoiceStrategy would have to perform this task, which would severely lessen the cohesion among that class.

Indirection

Since we did not want to have Player directly coupled with the GameState, we used the Indirection pattern. We used an intermediate object, DealState, to mediate between GameState and Player. That way, we did not have to have extra coupling between Player and GameState. The use of Indirection can also be seen in the implementation of IMoveChososer. This Strategy interface prevents Player from talking directly with MainWindow and IGameObserver, helping to lower the coupling between classes in our system.

Protected Variations

From the onset of this project, there has been significant risk and instability in the aspect of our system for choosing what move a player should make. To prevent his from having undesirable effects on other elements in our system, we have identified this point of instability and have created a stable façade interface (IMoveChoiceStrategy) to surround the various implementations of the interface. This has allowed us to “wall off” this risky and instable portion of our system from the rest of the domain architecture to ensure that the rest of our system is working before we move into this point of instabilitiy.

Gang of Four Design Patterns

Strategy

After deciding that only the difference between various Player objects in our system was the algorithm each uses for determining what move to make, we decided to employ the Strategy pattern. IMoveChoiceStrategy functions as the common interface between the various Strategy classes. This allows for related, but varying move choosing algorithms to be incorporated into our system while still maintaining the idea of pluggable software components. This also allows this point in our system to be highly flexible to extension and evolution as new strategies can be implemented with relatively little change to the other classes in the architecture.

Observer
We used the Observer pattern to update the MainWindow about what is going on in the domain. We have an IGameObserver class that has methods for each important game event, such as ‘playerRaised’ and ‘potUpdated.’ GameController has a list of objects that implement IGameObserver. The GameController loops through each object and updates each one as the game is in play. The list currently only contains one IGameObserver, which is MainWindow. However, having an IGameObserver can allow for other objects to be updated about the game state in future iterations.

State
We used the State pattern to represent the different rounds in the poker game. There are five different rounds in a poker game: preflop, flop, turn, river, and showdown. Since the behavior of each state differs, we created a state for each. They all implement a DealState class, which has the basic functionality that is consistent throughout most of the states. Then, the rounds’ states override any methods that differ.

Façade
We used the Façade pattern to wall off the move making decisions of each player. Since the decisions about moves was a very high-risk part of the design, we used the IMoveChoiceStrategy to keep the decision-making separate from the main components of the game. The IMoveChoiceStrategy is a higher-level interface that hides a subsystem – that of making the move choices. In the future, different ways of determining moves can be created but still be accessed through the IMoveChoiceStrategy.

Factory
The Factory Pattern has influenced the design of our system by helping us decide who should create the various Player objects. Between the automated and human players there is some complex creation logic involved as far as giving each an appropriate implementation of IMoveChoiceStrategy. An object representing a human player has that object passed in as the MainWindow, but with automated players, the creating object must instantiate the specific IMoveChoiceStrategy object to be given to that player. Here the use of a Pure Fabrication Factory class has proven the best design decision. This has allowed us to separate the creation of various types of players to improve cohesion while still allowing for each player to have the desired algorithm for choosing which move to make.

Acceptance Test Plan

This section includes both use cases and test cases. Use cases describe the interaction between the user and the system, and test cases provide a method of testing if the system satisfies requirements defined by the use cases.

Use Cases

Use Case #1: Starting the Game

Name	Starting the Game
Description	Describes how the user starts the Texas Hold 'Em poker game
Actors	A potential Microsoft customer or a Microsoft associate
Pre-Conditions	1. User has opened the application
Basic Flow	1. User clicks "Start" button (alternate flow 1 possible) 2. System prompts user as to whether or not they would like to participate in the poker game
Alternate Flows	1. User chooses to exit the program a. User clicks "Exit" button b. System exits
Post-Conditions	1. The user has initialized the poker simulation
Other Stakeholders	Other people watching the user's interaction with the system
Systems/Subsystems	The computer's hardware
Special Requirements	1. The system will not allow the user to begin a poker game when a poker game is already in progress

Use Case #2: Choosing Whether or Not to Participate

Name	Choosing Whether or Not to Participate
Description	Describes how the user chooses to participate or refrain from participating
Actors	A potential Microsoft customer or a Microsoft associate
Pre-Conditions	1. User has opened the application 2. User has started the game
Basic Flow	1. System prompts user as to whether or not they would like to participate in the poker game 2. User clicks either "Yes" or "No" (alternate flow 1 possible)
Alternate Flows	1. User chooses to exit the program a. User clicks "Exit" button b. System exits
Post-Conditions	1. The system has received the users' request to participate or to refrain from participating
Other Stakeholders	Other people watching the user's interaction with the system
Systems/Subsystems	The computer's hardware
Special Requirements	1. The system will only allow the user to choose "Yes" or "No"

Use Case #3: Playing the Game

Name	Playing the Game
Description	Describes how the user actively participates in the Texas Hold 'Em poker game
Actors	A potential Microsoft customer or a Microsoft associate
Pre-Conditions	1. User has opened the application 2. User has started the game 3. User has selected to participate in the game
Basic Flow	1. System displays the user's cards and any automated players' turns that precede the user's 2. System prompts user for move choice 3. At each turn, user submits a move choice (alternate flow 3 possible) 4. System continues to display automated players' turns until it is the user's turn again (alternate flows 1 and 2 possible) 5. System completes game and displays a notification about which player has won the game
Alternate Flows	1. All automated players fold a. System displays notification that user has won the game 2. All rounds are complete a. System displays notification about which player has won the game 3. User's choice is to fold a. System continues to display automated players' turns until the game completes b. System displays notification about which player has won the game
Post-Conditions	1. The system has completed the entire poker game 2. The system has displayed the results of the game
Other Stakeholders	Other people watching the user's interaction with the system
Systems/Subsystems	The computer's hardware
Special Requirements	1. A new game can be started after the game finishes if there are still two players with enough money to play

Test Cases

Test cases are used to evaluate the performance of the system. They focus on the interaction between the system and the user.

Test Case Set #1: Starting the Game

Scenario Matrix

Scenario Number	Originating Flow	Alternate Flow
1	basic flow	
2	basic flow until 1	alternate flow 1

Test Cases

Test Case ID	Scenario	Description	Condition: "Start" is selected	Condition: "Exit" is selected	Expected Result
1	1	User starts the game	User clicks "Start" button	Invalid	The system displays participation choice dialog
2	2	User exits	Invalid	User clicks "Exit" button	System exits

Test Case Set #2: Choosing Whether or Not to Participate

Scenario Matrix

Scenario Number	Originating Flow	Alternate Flow
1	basic flow	
2	basic flow until 2	alternate flow 1

Test Cases

Test Case ID	Scenario	Description	Condition: Participation choice is selected	Condition: "Exit" is selected	Expected Result
1	1	User selects his participation option	User chooses participation option	Invalid	The system begins a game according to user's choice
2	2	User exits	Invalid	User clicks "Exit"	System exits

Test Case Set #3: Playing the Game

Scenario Matrix

Scenario Number	Originating Flow	Alternate Flow
1	basic flow	
2	basic flow until 3	alternate flow 3
3	basic flow until 4	alternate flow 1
4	basic flow until 4	alternate flow 2

Test Cases

Test ID	Scenario	Description	Condition	Condition: "Exit" is selected	Expected Result
1	1	User plays game	User has not folded	Invalid	The system completes the game with user participation and displays game results
2	2	User chooses to fold in preflop	User folds	Invalid	System completes rest of game without user participation and displays results
3	2	User chooses to raise in preflop	User raises	Invalid	System decreases stack of player and increases pot; system continues round until human player's move again
4	2	User chooses to check in preflop	User checks	Invalid	System continues round until human player's move again
5	2	User chooses to calls in preflop	User calls	Invalid	System decreases stack of player and increases pot; system continues round until human player's move again
6	2	User chooses to fold in flop	User folds	Invalid	System completes rest of game without user participation and displays results
7	2	User chooses to raise in flop	User raises	Invalid	System decreases stack of player and increases pot; system continues round until human player's move again
8	2	User chooses to check in flop	User checks	Invalid	System continues round until human player's move again
9	2	User chooses to calls in flop	User calls	Invalid	System decreases stack of player and increases pot; system continues round until human player's move again
10	2	User chooses to fold in turn	User folds	Invalid	System completes rest of game without user participation and displays results
11	2	User chooses to raise in turn	User raises	Invalid	System decreases stack of player and increases pot; system continues round until human player's move again
12	2	User chooses to check in turn	User checks	Invalid	System continues round until human player's move again
13	2	User chooses to calls in turn	User calls	Invalid	System decreases stack of player and increases pot; system continues round until human player's move again
14	2	User chooses to fold in river	User folds	Invalid	System completes rest of game without user participation and displays results
15	2	User chooses to raise in river	User raises	Invalid	System decreases stack of player and increases pot; system continues round until human player's move again
16	2	User chooses to check in river	User checks	Invalid	System continues round until human player's move again
17	2	User chooses to calls in river	User calls	Invalid	System decreases stack of player and increases pot; system continues round

					until human player’s move again
18	3	All automated players fold	All automated players fold	Invalid	System displays results of game
19	4	All rounds are done	N/A	Invalid	System displays results of game