SE 3XA3: Module Interface Specification
Poker Project

Team 12
Safwan Hossain and hossam18
Eamon Earl and earle2
Tyler Magarelli and magarelt

April 12, 2022

Card (ADT)

Card

Uses

None

Syntax

Imported Constants

None

Imported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
greater_than	Card	$\mathbb{B}$	

Semantics

State Variables

$suit : I$
$rank$: $\mathbb{I}$

State Invariant

$1 <= suit <= 4$
$2 <= rank <= 14$

Assumptions

None

Considerations

None

Access Routine Semantics

greater_than(C):

- $rank >= C.rank \implies true \mid false$

Player (ADT)

Player

Uses

Card

Syntax

Imported Constants

None

Imported Types

Card

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new Player	String, $\mathbb{I}$		
clear_hand			
hasTurn		$\mathbb{B}$	
giveTurn			
takeTurn			
set_chips	$\mathbb{I}$		
get_chips		$\mathbb{I}$	
take_chips	$\mathbb{I}$		IllegalArgumentException
insert	Card		
get_hand		Card [ ]	

Semantics

State Variables

$name : String$
$chips : \mathbb{I}$

$hand$: `Card [ ]`
has_turn : $\mathbb{B}$

State Invariant

$0 <= chips$

Assumptions

None

Considerations

None

Access Routine Semantics

new Player(s, c):

- transition : $hand, name, chips, has\_turn := \epsilon, s, c, False$

clear_hand():

- transition : $hand := \epsilon$

hasTurn():

- return : has_turn

giveTurn():

- transition : $has\_turn := True$

takeTurn():

- transition : $has\_turn := False$

set_chips(c):

- transition : $chips := c$

get_chips(c):

- return : $chips$

take_chips(c):

- transition : $chips := chips - c$
- error : $chips - c < 0 \implies IllegalArgumentException$

insert(C):

- transition : $C \in hand$
- post-condition: $\forall i \in [0..len(hand) - 2] : hand[i].rank <= hand[i + 1].rank$
- description : inserts the card C into hand such that the hand is ordered in ascending order by rank

get_hand():

- return : $hand$

Deck (ADT)

Deck

Uses

Card Player

Syntax

Imported Constants

None

Imported Types

Card Player

Exported Access Programs

Routine name	In	Out	Exceptions
fillDeck			
shuffle			
reset			
draw	$\mathbb{I}$	Card [ ]	StackOverflowException

Semantics

State Variables

$deck$: `Card [52]`
$flop$: `Card [ ]`
$stack\_p$: $\mathbb{I}$

State Invariant

$0 <= stack\_p <= 51$

Assumptions

None

Considerations

Deck is suggested to be implemented as a stack, but the choice is ultimately up to the development team. If it is not implemented as such, the stack_p state variable will not be needed and its associated invariant can be disregarded.

Access Routine Semantics

fill_deck():

- transition : fills the deck stack with all 52 unique playing cards (of type Card)

shuffle():

- transition : randomly shuffles the current cards in the deck to a degree wherein the sequence can be expected to be drastically different from the precondition of the deck stack

reset():

- transition : returns all 52 unique cards to the deck stack and shuffles the deck, s:= 0

draw(n):

- transition: removes the top n cards from the deck, and places them into a list
- exception: $n >$ remaining cards in the deck $\Longrightarrow StackOverflowException$
- returns: Card [n]

Game

Game

Uses

Card Player Deck

Syntax

Imported Constants

None

Imported Types

Card Player Deck

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new Game	Player [ ], $\mathbb{I}$		
startGame	$\mathbb{I}$		
removePlayer	Player		
foldPlayer	Player		
is_round_over		$\mathbb{B}$	
dealCards	$\mathbb{I}$		
getNextPlayer		Player	RuntimeException
getCurrentPlayer		Player	RuntimeException
giveNextTurn			

Semantics

State Variables

- deck : Deck
- players: Player [ ]
- unfoldedPlayers: Player [ ]
- currentPlayerIndex: $\mathbb{I}$
- nextPlayerIndex: $\mathbb{I}$
- minimumCallAmount: $\mathbb{I}$
- round_over: $\mathbb{B}$

State Invariant

- there must always be a number of unfolded players less than or equal to the number of players
- currentPlayerIndex must always be greater than or equal to 0 and less than the length of unfoldedPlayers

- same as above, but for nextPlayerIndex

Assumptions

None

Considerations

Implementing players and unfoldedPlayers as dynamic arrays seems ideal.

Access Routine Semantics

new Game(p_list, x):

- transition:
  - deck := new Deck()
  - players := p_list
  - minimumCallAmount := x
  - currentPlayerIndex := 0
  - nextPlayerIndex := 0
  - round_over := False

startGame(c):

- action: dealcards(c), giveNextTurn()

removePlayer(p):

- transition: players := {players - p}

foldPlayer(p):

- transition: unfoldedPlayers := {unfoldedPlayers - p}

- transition: if no more unfolded players, round_over := True

is_round_over():

- return: round_over

dealCards(c):

- transition: insert c cards into each players hand using Player.insert()

getNextPlayer():

- return: the next player to go catching out of bounds errors

getCurrentPlayer():

- return: the current player to go catching out of bounds errors

giveNextTurn():

- action: triggers the next players turn

Hand Evaluator

HandEval

Uses

Card

Syntax

Imported Constants

None

Imported Types

Card

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
evaluate	Card		

Semantics

State Variables

None

State Invariant

None

Assumptions

This module is made for standard 5 card poker hands, and will be used solely to evaluate such hands

Considerations

It might make sense to use auxiliary functions to evaluate hand states, as different ranks of hands can have similar properties.

Access Routine Semantics

evaluate(c_list):

- returns : a tuple of integers, the first representing the relative rank of the hand (regarding standard 5 card poker rules), and the second representing the rank of the highest card in the hand for tie breakers

GameView

GameView

Uses

Card

Syntax

Imported Constants

None

Imported Types

Card

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
display	Card		

Semantics

State Variables

None

State Invariant

None

Assumptions

None

Considerations

None

Access Routine Semantics

display(C):

- behaviour: displays the suit and rank of card c

MainController Module

Uses

Client, Game, Gameview, MainMenuView, Server

Syntax

Imported Constants

None

Imported Types

Client, Game, Server

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
getValidUsername	Scanner	String	
getValidOption	Scanner	$\mathbb{Z}$	
getValidSocketForServer	Scanner	Socket	
hostServer			IOException
joinServer	Scanner		IOException
exitProgram			
performMainMenuOperation	Scanner		
enterProgram			

Semantics

Environment Variables

Keyboard: Scanner(System.in)

State Variables

username : *String*
socket : *Socket*
client : *Client*

State Invariant

None

Assumptions

None

Considerations

None

Access Routine Semantics

getValidUsername(scanner):

- return : A String that is non-empty if all white spaces are deleted.

getValidOption(scanner):

- return : $option : \mathbb{Z} | 0 < option <= MAX\_NUM\_OPTIONS$.
- description : Returns an integer between 0 and the maximum number of available main menu options.

getValidSocketForServer(scanner):

- return : A socket that has successfully established a connection with a server.

hostServer():

- transition : Creates a new server using the user's current IP address.
- exception : IO Exception. Can be caused by thousands of issues (IP address, ports, connectivity issues).

joinServer(scanner):

- transition : Joins an existing server given a server IP address.
- exception : IO Exception. Can be caused by thousands of issues (IP address, ports, connectivity issues).

exitProgram():

- transition : Shuts down the program.

performMainMenuOperation(scanner):

- transition : Perform a main menu task, given a number that represents the task to perform. For example, user inputs the number 2 and according to the main menu, number 2 represents the task join a server, so user will join a server.

enterProgram:

- Transition : Displays welcome screen once. Then displays the main menu and asks the user to select an option in a never-ending loop.

ClientController Module

Uses

Client, Game, Gameview, PlayerAction, GameInfo

Syntax

Imported Constants

None

Imported Types

Client, Game, PlayerAction, GameInfo

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
listenForIncomingMessages			
performGameAction	Scanner		IOException
getValidPlayerAction	Scanner	PlayerAction	
getValidBet	Scanner	$\mathbb{Z}$	
CreateGameInfo	PlayerAction, $\mathbb{Z}$	GameInfo	

Semantics

State Variables

$player : Player$
$client : Client$
$game : Game$

State Invariant

None

Assumptions

None

Considerations

None

Access Routine Semantics

listenForIncomingMessages():

- transition : On a separate thread, continuously listens for messages received by *client* from the server.

performGameAction(scanner):

- transition : Uses getValidPlayerAction() and getValidBet() to ask the user to make their next move, then stores that information in a new GameInfo object and sends the object to the server from *client*.
- exception : Throw IO Exception if there are connectivity issues. Can be caused by thousands of issues (IP address, ports, connectivity issues).

getValidPlayerAction(scanner):

- return : $playerAction : PlayerAction$
- description: Asks the user for a valid player action. If the user input matches a PlayerAction enumerator then return the PlayerAction enmerator. Otherwise ask again.

getValidBet(scanner):

- return : $amount : \mathbb{Z} | amount >= 0$

- description: Asks the user for a valid betting amount. If the user input an integer that is greater or equal to zero then return the integer. Otherwise ask again.

CreateGameInfo(playerAction, amount):

- return : GameInfo(*client.clientID*, *player.name*, playerAction, amount)
- description : Creates a GameInfo object with the current player's information (clientID and name) and the move the player wants to make (playerAction and amount).

Client ADT Module

Uses

Syntax

Imported Constants

None

Imported Types

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new Client	Socket, String		IOException
getClientID		String	
setClientID	String		
IsConnectedToServer		$\mathbb{B}$	
listenForMessage		Object	IOException, ClassNotFoundException
sendMessage	Object		IOException
closeEverything			

Semantics

State Variables

$clientID : String$
$playerName : String$
$socket : Socket$
$inputStream : ObjectInputStream$
$outputStream : ObjectOutputStream$

State Invariant

None

Assumptions

None

Considerations

None

Access Routine Semantics

new Client(socket, name):

- transition : $self.socket, playerName, inputStream, outputstream :=$ $socket, name, new\ ObjectInputStream, new\ ObjectOutputStream$
- exception : Throw IO Exception if there are connectivity issues. Can be caused by thousands of issues (IP address, ports, connectivity issues).

getClientID():

- return : $clientID$

setClientID(clientID):

- transition : $self.clientID := clientID$

IsConnectedToServer(scanner):

- return : $socket.IsConnected()$

listenForMessage():

- return : An Object from $outputStream$ (once recieved).
- exception : Throw IO Exception if there are connectivity issues. Can be caused by thousands of issues (IP address, ports, connectivity issues).
- exception : Throw ClassNotFoundException if an Object cannot be recieved.

sendMessage(object):

- transition : Sends in an Object into $inputStream$.
- exception : Throw IO Exception if there are connectivity issues. Can be caused by thousands of issues (IP address, ports, connectivity issues).

closeEverything(playerAction, amount):

- transition : Close $socket, inputStream$ and $outputStream$
- description : Closes all sockets, streams and any connections to the servers.

ClientHandler ADT Module

Template Module implements Runnable Interface

Client Handler

Uses

Runnable, GameInfo, Game

Syntax

Imported Constants

None

Imported Types

GameInfo

Exported Access Programs

Routine name	In	Out	Exceptions
new ClientHandler	Socket	ClientHandler	
run			
updateClients	GameInfo		
closeEverything			

Semantics

State Variables

$clientUsername : String$
$clientHandlers$: static sequence of $ClientHandler$
$game$: static $Game$
$socket : Socket$
$inputStream : ObjectInputStream$
$outputStream : ObjectOutputStream$

State Invariant

None

Assumptions

None

Considerations

None

Access Routine Semantics

new ClientHandler(socket):

- return : $self$
- transition : $self.socket, inputStream, outputStream, clientHandlers :=$ $socket, newObjectInputStream(), newObjectOutputStream, clientHandlers || < self >$
- description : initializes $socket$, creates new input and output streams and adds $self$ to $clientHandlers$ (which is a static sequence).

run():

- transition : Get any commands coming from $outputStream$, input that command into $game$ then send new game information to all clients.
- description : Each ClientHandler is responsible for taking in input from a single client connected to a server. Everytime a client sends a command (their game move) to the server, their designated ClientHandler will receive that command and input that command into the game on the server on behalf of the client's name (as if the client had inputted the command directly to the game). Then the ClientHandler will forward the resulting state of the game after the input, synchronizing the game for all clients.

updateClients(gameInfo):

- transition : For every clientHandler's output stream, write in $gameInfo$ as the output and send.

closeEverything(scanner):

- transition : Close $socket, inputStream$ and $outputStream$
- description : Closes all sockets, streams and any connections to the servers.

Server ADT Module

Uses

ClientHandler

Syntax

Imported Constants

None

Imported Types

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new Server	ServerSocket		
startServer			
closeServer			

Semantics

State Variables

$serverSocket : ServerSocket$

State Invariant

None

Assumptions

None

Considerations

None

Access Routine Semantics

new Server(serverSocket):

- return : $self$
- transition : $self.serverSocket := serverSocket$

startServer():

- transition : Listen for any attempts to connect to *serverSocket* by a Client. If an attempt is made, try to get the client's socket and create a new ClientHandler (using the client's socket) on a new thread and start the thread.
- description : The ClientHandler is responsible for taking in input from a single Client connected to a server. Everytime a Client connects to the server, a new ClientHandler will be created on a new Thread to listen for input from that specific Client.

closeServer():

- transition : Close *serverSocket*
- description : Closes all sockets, streams and any connections to the clients.

GameInfo ADT Module

Template Module implements Serializable

GameInfo

Uses

PlayerAction

Syntax

Imported Constants

None

Imported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
new GameInfo	String, String, PlayerAction, $\mathbb{Z}$	GameInfo	
getClientID		String	
getPlayerName		String	
getPlayerAction		PlayerID	
getAmount		$\mathbb{Z}$	

Semantics

State Variables

$clientID : String$
$playerName : String$
$playerAction : PlayerAction$
$amount : \mathbb{Z}$

State Invariant

None

Assumptions

None

Considerations

None

Access Routine Semantics

new GameInfo(serverSocket):

- return : $self$
- transition : $self.clientID, self.playerName, self.playerAction, self.amount := clientID, playerNa$
- description : GameInfo is essentially a data structure that Client and Server will use to communicate.

getClientID():

- return : $clientID$

getPlayerAction():

- return : $playerAction$

getPlayerName():

- return : $playerName$

getAmount():

- return : $amount$

toString():

- return : $playerName$ ‖ " performs the action " ‖ $playerAction$ ‖ " for an amount of " ‖ amount;

PlayerAction Module

Uses

PlayerAction

Syntax

Exported Constants

None

Exported Types

PlayerAction = {
FOLD, #Player wants to fold
CHECK, #Player wants to check
CALL, #Player wants to call
RAISE, #Player wants to raise
BET #Player wants to bet
}

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
isABet	PlayerAction	$\mathbb{B}$	
actionIsValid	String	$\mathbb{B}$	
getActionByString	String	PlayerAction	IllegalArgumentException

Semantics

State Variables

None

State Invariant

None

Assumptions

None

Considerations

PlayerAction is an enum class that represent the possible actions a player can make

Access Routine Semantics

isABet():

- return : $self == BET \lor self == RAISE$

actionIsValid(action):

- return : True if the String *action* is a PlayerAction. False if not.

getActionByString(action):

- return : Corresponding PlayerAction that matches *action*
- exception : Throw IllegalArgumentException if there is no string value for PlayerAction that matches *action*

MainMenuView Module

Uses

Syntax

Imported Constants

None

Imported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
displayWelcomeScreen			
displayMainMenu			
askForMenuOption			
displayInvalidMenuOption			
askForUsername			
displayInvalidUsername			
displayServerIPAddress	String		
displayServerJoinMenu			
displayFailedToConnectToServer	String		
displaySuccessfullyStartedServer			
displaySuccessfulConnection			
displayWaitingForHost			
displayExitingProgram			

Semantics

State Variables

None

State Invariant

None

Assumptions

None

Considerations

None

Access Routine Semantics

displayWelcomeScreen():

- output : out := Display a welcome screen message.

displayMainMenu():

- output : out := Display a main menu options.

askForMenuOption():

- output : out := Display a message asking the user to enter in their desired main menu option.

displayInvalidMenuOption():

- output : out := Display a message saying that the main menu option they entered is invalid.

askForUsername():

- output : out := Display a message asking the user to enter in their desired username.

displayInvalidUsername():

- output : out := Display a message saying that the username they entered is invalid.

displayServerIPAddress(serverIP):

- output : out := Display a message saying that the IP address of the server is $serverIP$.

displayServerJoinMenu():

- output : out := Display a join to server menu.

displayFailedToConnectToServer():

- output : out := Display an error message saying the user failed to connect to the server.

displaySuccessfullyStartedServer():

- output : out := Display a success message saying the user successfully started the server.

displaySuccessfulConnection():

- output : out := Display a success message saying the user successfully connected to the server.

displayWaitingForHost():

- output : out := Display a message saying that the server is waiting for the host to start the game.

displayExitingProgram():

- output : out := Display an exit game message.