CIS-17C Project 2 Documentation

“Blackjack! With Friends v2”

Saim Ahmed

June 6th, 2025

Advisor: Mark Lehr

Introduction:

My reasoning for this project carried through from my first version of Blackjack with Friends. I chose this because I wanted to add more meat to a relatively simple, yet fun and engaging card game. I like to play blackjack (in a non-gamblers way) with friends at parties or events, so I know my way around it. With this expansion of the game's nature, I believe there’s a lot of room to implement features that make the process of playing blackjack more interesting and friendly, especially as you increase the number of people involved.

I spent around a week designing, implementing, and debugging the program, with about an average of 3-4 hours per day on it. The project is approximately 1158 lines (pure code), and there are 7 classes and 1 standalone function being the main method.

This project is stored and controlled through a github repository.

Approach to Development:

For the second iteration of this project I had to look back at what I did for the first version, and try to add the concepts we’ve learned these past few months. I was able to implement recursion onto the deck class, Trees onto the dealer class, Hashing and Graphs onto the GameGraph class as a way to improve my code performance and intuitiveness. I learned so much during this enhancement project, especially regarding the syntax of C++, and how data structures can operate in tandem with one another. All help was found through StackOverflow or

GeeksForGeeks in order to get a better visualization of the concepts I had to add.

Version control was preserved through github, in order to ensure all data and updates were saved. Some iterations of the version are entirely for functionality, while others simply add more detail and refining to existing features.

Game Rules:

1. Players and Bets:
    - Players are each prompted to make a wager.
    - Dealer grants two cards, one faced down, another up.
2. Player decisions
    - Players can decide to hit (take another card), or stand (keep current hand).
    - Players who exceed 21 points automatically lose their wagers.
    - After all players have completed their decisions, the dealer turns.
3. Dealer:
    - Dealer follows through with revealing hand and must hit until their hand value is 17 or higher
4. Player Goal:
    - The ideal scenario for each player is to get a blackjack (automatic win), score higher than the dealer, or have the dealer bust.
    - If the dealer busts, all remaining players win.
5. Dealer Goal:
    - Dealers' hands must exceed players to win, or get a blackjack to win automatically.

   - If the players win, payout their wager reward, if they lose, remove their wager, and if they tie, return their wager entirely.
6. Winning: For players, the goal is to beat the dealer's hand by all means necessary. Doing so means the wager is won and doubled by some casino value.

Program Classes:

Card Class:

- Represents a single card within a deck with an added suit and rank.
- Functions to create cards, get suits, get ranks, flip, get values, and a toString menu for game flow.
- To be used in the Deck Class.

Deck Class:

- Creates a deck of 52 cards (standard deck) to be used by the game players and dealer logic.
- Randomly shuffles cards as needed.
- Deck functions to populate, deal, add and retrieve from discard pile, and iterators to traverse through deck.

Hand Class:

- Represents both player and dealer hand (for compartmentalization purposes).
- Keeps track of hand values, clears hand, adds card, checks for bust or blackjack.

Player Class:

- Represents non-Robot players in the game.
- Tracks player name, money, current bet, and current hand.
- Used to grant wins, losses, or tie pushes, as well as track busts, hits, blackjacks, and hands.

DealerTree Class:

- Represents all non-Player actions and overrides within game
- Functions to match human player traits (hits, busts, show hand).
- Admin-like functions to shuffle deck, deal a card, and signal an empty deck.
- Uses a tree structure decision tree to determine hit or stand status for dealers.

GameGraph Class:

- Represents leaderboard and tracker of player stats within the game.
- Used to record wins, losses, high scores, and get them accordingly.
- Uses a graph structure to manage transitions from state to state in a closed graph.

Game Class:

- Primary logic flow class.
- Adds and removes players
- Controls gameplay, dealing cards, placing bets, player turn, dealer turns, payouts, and cleanup
- Records and updates highscores, wins, losses, and action logs.
- Graphic interface stored here too.

New Concepts Applied:

Recursion:

a. Location: Hand Class
b. Purpose: Uses a recursive call to handle card logic for ace cards based on the player's total hand, as well as return the hand total based on face, rank, and/or ace value.

Hashing:

a. Location: GameStats Class
b. Purpose: An unordered_map is used in the GameGraph class to store key value pairs, using hash tables to organize the elements.

Trees:

a. Location: DealerTree Class
b. Purpose: The addition of a treelike structure manages the decision to stand or hit within the dealer in a tree form. Returning if the desired hand value is true to hit or not.

   The tree also allows for a much more detailed determination, leaning closer to an artificial dealer logic.

Graphs:

a. Location: GameGraph Class
b. Purpose: Implements a finite state machine graph system in order to manage and validate state transitions. Keeps track of current state and allows the system to move through the graph, validating each transition as it moves through.

Sample Input/Output:

```
=====WELCOME TO BLACKJACK! WITH FRIENDS=====
=====Your goal as the player is to try and beat the dealer by
getting as close to 21 without going over!
=====Each player starts with two cards. The dealer's first card
is hidden.
=====Each player also must decide to hit (take another card), or
stand (stop taking cards).
=====The dealer will play against you! With some mental math and
a little bit of luck, you could get rich!
=====To get started, each of you must create a player profile
with your name!
=====Good luck, and have fun!

=====MENU=====
1. Play Game
2. Create Player Profile
3. Load Player Profile
```

```
4. View Stats
5. View High Scores
6. Quit Game

Enter choice: 2

Enter your name:  Saim
Creating a new profile for  Saim with $ 1000 to start with.
Player profile created successfully!

=====MENU=====
1. Play Game
2. Create Player Profile
3. Load Player Profile
4. View Stats
5. View High Scores
6. Quit Game

Enter choice: 2

Enter your name: Momo
Creating a new profile for Momo with $ 1000 to start with.
Player profile created successfully!

=====MENU=====
1. Play Game
2. Create Player Profile
3. Load Player Profile
4. View Stats
5. View High Scores
6. Quit Game

Enter choice: 1

===== PLACING BETS =====
 Saim , you've got $1000. Place your bet: $100
 Saim bet $100
Momo , you've got $1000. Place your bet: $100
Momo bet $100

===== DEALING CARDS =====
===== TABLE STATUS =====
The Dealer's hand: [FACED DOWN], 8 of Hearts
 Saim ($1000, bet: $100): 7 of Hearts, Ace of Spades, Total: 18
Momo ($1000, bet: $100): 5 of Diamonds, King of Clubs, Total: 15

===== PLAYER TURNS =====

 Saim's turn:
 Saim, do you want to hit? (y/n) y
 Saim receives: 3 of Diamonds
 Saim has 21.
 Saim stands with 21.

Momo's turn:
Momo, do you want to hit? (y/n) n
Momo stands with 15.

===== DEALER'S TURN =====
The Dealer's hand: 9 of Diamonds, 8 of Hearts, Total: 17
Dealer stands with 17.

===== RESULTS =====
 Saim: Blackjack! Won $150.
Momo: Lost $100 with 15 under dealer's 17.

EVENT:  Saim won $150 with blackjack.
EVENT: Momo lost $100 with 15 under dealer's 17.

===== PLAYER STATS =====
Player         Wins Losses   Win Rate
===========================================
Saim           1    0         100.0%
Momo           0    1         0.0%

===== HIGH SCORES =====
Rank   Player       Money
===========================================
1.    Saim          $1150
2.    Momo          $900

=====PLAYER MONEY STATS=====
Current Top Earner:  Saim with $1150
Current Low Earner: Momo with $900

Continue playing? (y/n): n

=====MENU=====
1. Play Game
2. Create Player Profile
3. Load Player Profile
4. View Stats
5. View High Scores
6. Quit Game

Enter choice: 6
Goodbye!

(end of program)
```

Checkoff Sheet:

Sequences:

- List:
    a. Location: Deck Class

    b. Purpose: Used to store Cards in the form of a Deck.

Associative Containers:

- Set:
    a. Location: GameGraph class
    b. Purpose: used to store player highscores in the form of a set of pairs containing the money won, and the player name.
- Map (x2):
    a. Location: (Both) Game Class
    b. Purpose: Used to store card values with key value pair for rank and suit (string, int). The second was used to store special characters for string suits (string, string).

Container Adaptors:

- Stack:
    a. Location: Deck Class
    b. Purpose: To simulate the behavior of a discard pile in a game of blackjack. Stacks follow the LIFO principle, accurate to a face down discard pile of cards.
- Queue:
    a. Location: Game Class
    b. Purpose: Used to store events of the game into an action log.
- Priority_Queue:
    a. Location: Game Class
    b. Purpose: Used to store, process, and exhaust all recorded game events. Also to create a game queue in an ordered manner by priority for game logic flow.

Iterators:

- Random Access Iterator:
    a. Location: Hand Class
    b. Purpose: Used to iterate through, compare, and increment/decrement Card object structures. Used all throughout classes that use Card objects, especially in instances where for-each loops aren't viable.

Algorithms:

- for_each:
    a. Location: (Primary) Game Class
    b. Purpose: To iterate through structures searching for specific function values for comparison and game flow.
- find:
    a. Location: GameGraph Class
    b. Purpose: Used to find, collect and compare player wins, losses, update high scores, and calculate win rate.

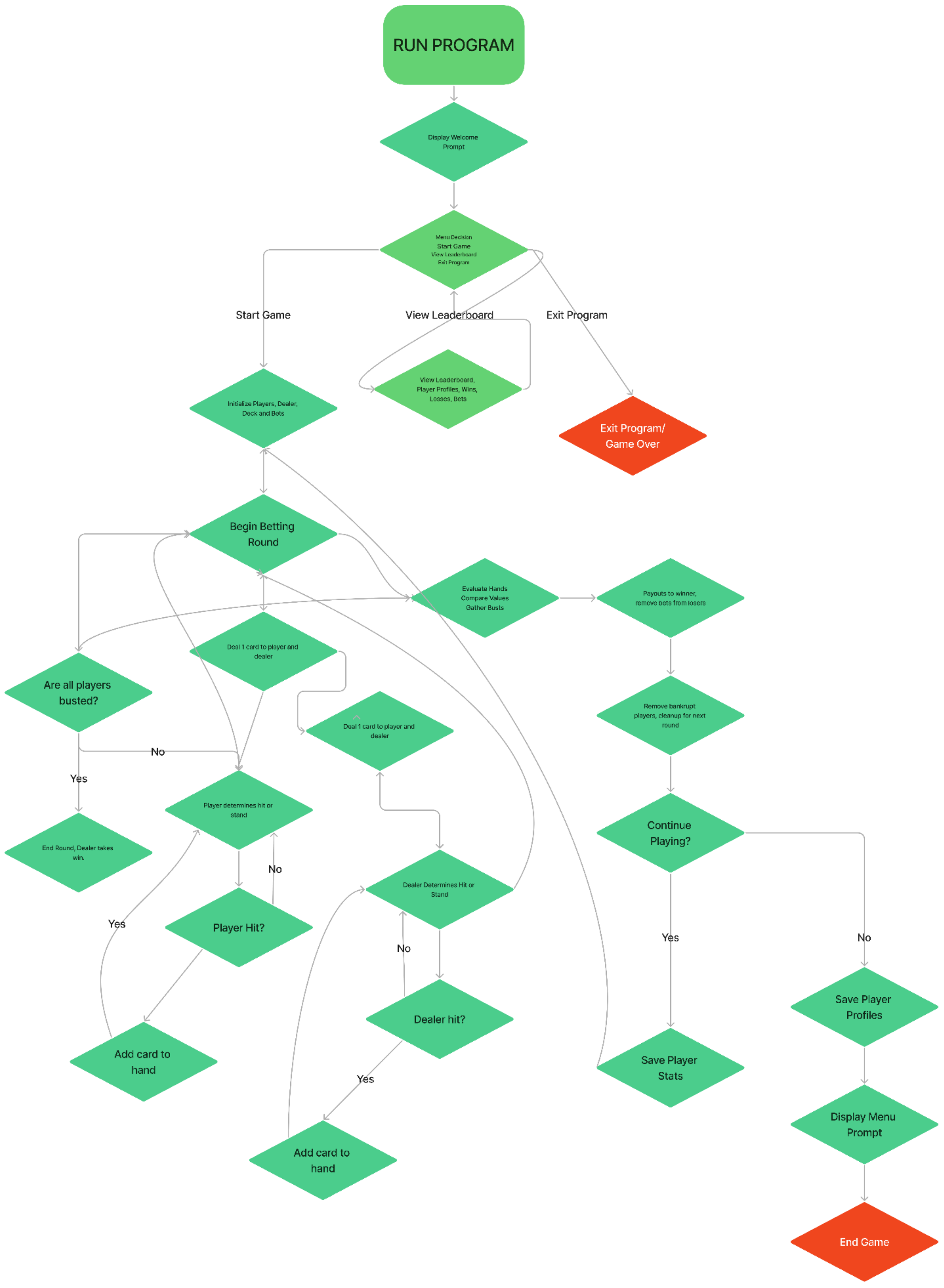
Mutating Algorithms:

- Remove:
    a. Location: Game Class
    b. Purpose: Used to erase players from a list of players and cleanup blackjack tables.
- Shuffle:
    a. Location: Deck Class
    b. Purpose: Fisher-Yates implementation of shuffle to accommodate for list iterations, used to randomize deck status for game logic

Organization:

- Minimum and Maximum:
    a. Location: Game Class
    b. Purpose: Used to find and output the highest and lowest earner in the current game of blackjack, until game ends and scores are recorded in a leaderboard.

Flowchart:

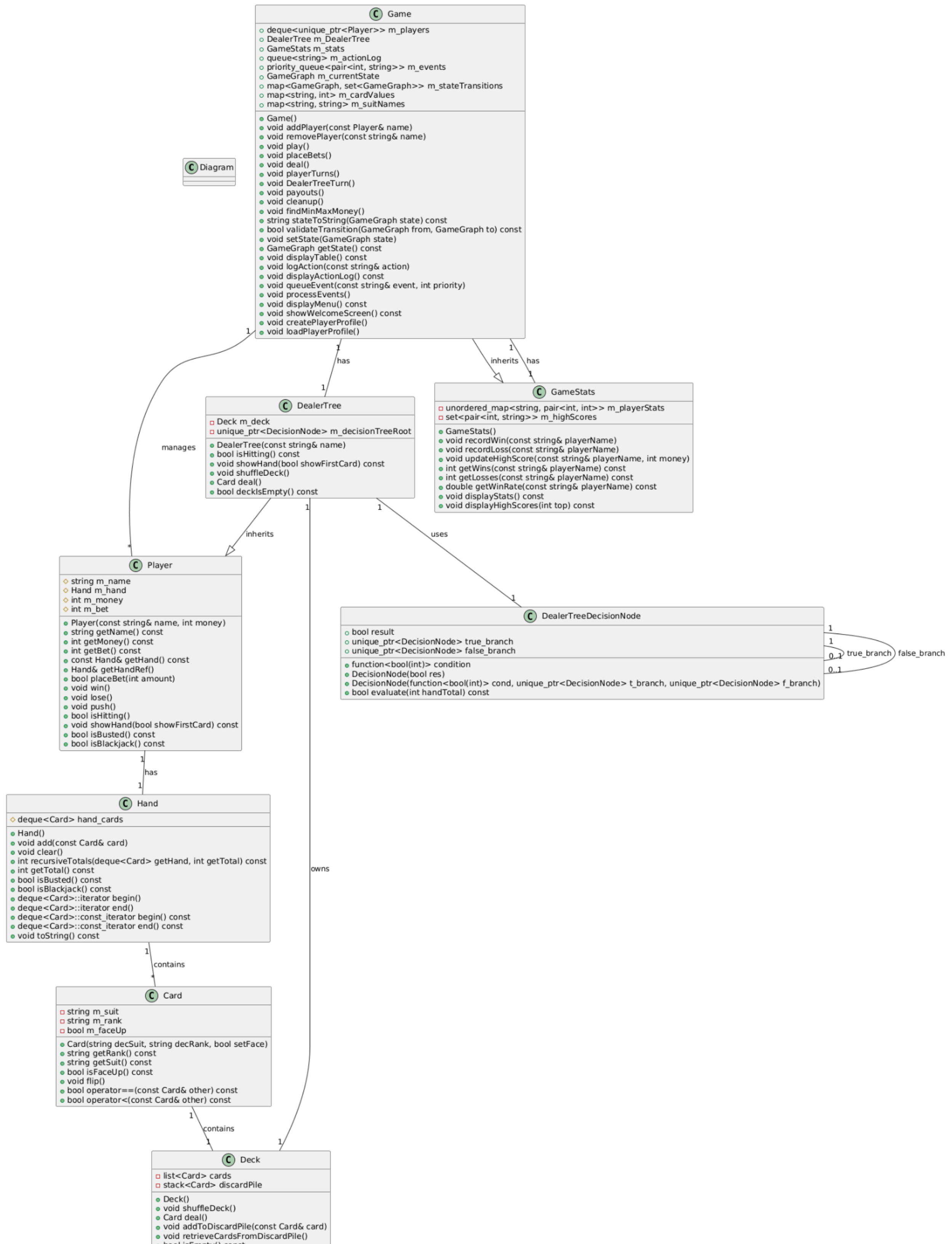

RUN PROGRAM
Display Welcome Prompt
Menu Decision
Start Game
View Leaderboard
Exit Program
Start Game
View Leaderboard
Exit Program
Initialize Players, Dealer, Deck and Bets
View Leaderboard, Player Profiles, Wins, Losses, Bets
Exit Program/ Game Over
Begin Betting Round
Evaluate Hands Compare Values Gather Busts
Payouts to winner, remove bets from losers
Deal 1 card to player and dealer
Are all players busted?
Deal 1 card to player and dealer
Remove bankrupt players, cleanup for next round
No
Yes
Player determines hit or stand
End Round, Dealer takes win.
Continue Playing?
No
Yes
Player Hit?
Dealer Determines Hit or Stand
No
Yes
No
Dealer hit?
Save Player Profiles
Add card to hand
Save Player Stats
Yes
Display Menu Prompt
Add card to hand
End Game

(New) Pseudocode:

```
// Global Constants
DEFINE MAX_CARDS = 52
DEFINE MAX_PLAYERS = 4
// Class Card
CLASS Card
    ATTRIBUTES:
        suit: string
        rank: string
        faceUp: boolean
    CONSTRUCTOR(suit, rank, faceUp = true):
        SET Card.suit = suit
        SET Card.rank = rank
        SET Card.faceUp = faceUp
    METHOD getRank(): RETURN rank
    METHOD getSuit(): RETURN suit
    METHOD isFaceUp(): RETURN faceUp
    METHOD flip(): Toggle faceUp status
    METHOD OPERATOR== (other_card): Check if rank and suit are
identical
    METHOD OPERATOR< (other_card): Compare ranks, then suits if
ranks are equal
END CLASS
// Class Deck
CLASS Deck
    ATTRIBUTES:
        cards: list of Card
        discardPile: stack of Card
    CONSTRUCTOR():
        Populate 'cards' list with 52 standard playing cards (4
suits x 13 ranks)

    METHOD shuffleDeck():
        Use a custom algorithm to randomly reorder 'cards' list
(iterating and swapping elements)
    METHOD deal():
        REMOVE top card from 'cards' list
        RETURN removed card
    METHOD addToDiscardPile(card):
        ADD card to 'discardPile' stack
    METHOD retrieveCardsFromDiscardPile():
        WHILE 'discardPile' is NOT empty:
            MOVE top card from 'discardPile' to 'cards' list
    METHOD isEmpty():
        RETURN TRUE if 'cards' list is empty, ELSE FALSE
    METHOD cardsRemaining():
        RETURN count of cards in 'cards' list
    METHOD toString():
        PRINT current size of 'cards' list and 'discardPile'
stack
END CLASS
// Class Hand
CLASS Hand
    ATTRIBUTES:
        hand_cards: deque of Card
    CONSTRUCTOR():
        Initialize empty hand_cards
    METHOD add(card):
        ADD card to 'hand_cards'
    METHOD clear():
        REMOVE all cards from 'hand_cards'
    METHOD recursiveTotals(current_hand: deque of Card,
current_total: integer):
        IF current_hand IS empty:
            RETURN current_total
        SET current_card = first card in current_hand
        REMOVE first card from current_hand
        SET rank = current_card.rank

        IF rank IS "Ace":
            SET total_if_ace_is_11 =
recursiveTotals(current_hand (copy), current_total + 11)
            SET total_if_ace_is_1 = recursiveTotals(current_hand
(copy), current_total + 1)
            IF total_if_ace_is_11 <= 21:
                RETURN total_if_ace_is_11
            ELSE:
                RETURN total_if_ace_is_1
        ELSE IF rank IS "King" OR "Queen" OR "Jack":
            SET card_value = 10
        ELSE:
            SET card_value = integer value of rank

        RETURN recursiveTotals(current_hand, current_total +
card_value)

    METHOD getTotal():
        RETURN recursiveTotals(a copy of hand_cards, 0)
    METHOD isBusted():
        RETURN TRUE if getTotal() > 21, ELSE FALSE
    METHOD isBlackjack():
        RETURN TRUE if size of hand_cards is 2 AND getTotal() is
21, ELSE FALSE

    // Iterators for traversing hand_cards (begin, end)

    METHOD toString():
        IF hand_cards IS empty:
            PRINT "Hand's empty..."
        FOR each card in hand_cards:
            IF card is face up:
                PRINT rank and suit
            ELSE:
                PRINT "[FACED DOWN]"
        PRINT "Total: " + getTotal()
END CLASS

// Class Player
CLASS Player
    ATTRIBUTES:
        name: string
        hand: Hand object
        money: integer
        bet: integer
    CONSTRUCTOR(name = "Player", money = 1000):
        SET Player.name = name
        SET Player.money = money
        SET Player.bet = 0
    METHOD getName(): RETURN name
    METHOD getMoney(): RETURN money
    METHOD getBet(): RETURN bet
    METHOD getHand(): RETURN hand (const reference)
    METHOD getHandRef(): RETURN hand (mutable reference)
    METHOD placeBet(amount):
        IF amount is invalid (<= 0 or > money):
            RETURN FALSE
        SET bet = amount
        DEDUCT amount from money
        RETURN TRUE
    METHOD win(): ADD (bet * 2) to money, SET bet = 0
    METHOD lose(): SET bet = 0
    METHOD push(): ADD bet to money, SET bet = 0
    METHOD isHitting():
        PROMPT player to hit (y/n)
        READ player choice
        RETURN TRUE if choice is 'y' or 'Y', ELSE FALSE
    METHOD showHand(show_first_card = true):
        PRINT player's name and hand using Hand.toString()
    METHOD isBusted(): RETURN hand.isBusted()
    METHOD isBlackjack(): RETURN hand.isBlackjack()
END CLASS
// Function for DealerTree's Decision Tree Condition
FUNCTION isHandTotalLessThanSeventeen(handTotal: integer):
    RETURN handTotal < 17
// Class DealerTree (inherits from Player)
CLASS DealerTree (EXTENDS Player)
    ATTRIBUTES:
        deck: Deck object
        decisionTreeRoot: unique_ptr to DecisionNode (root of
dealer's decision tree)
    NESTED STRUCT DecisionNode:
        ATTRIBUTES:
            condition: function<bool(int)> // A function wrapper
for a decision rule
            result: boolean // The outcome if this is a leaf
node
            true_branch: unique_ptr to DecisionNode // Next node
if condition is true
            false_branch: unique_ptr to DecisionNode // Next
node if condition is false
        CONSTRUCTOR(res: boolean): Initializes a leaf node with
a result
        CONSTRUCTOR(cond: function, true_branch_node,
false_branch_node): Initializes an internal node with a
condition and branches

        METHOD evaluate(handTotal: integer):
            IF condition IS NOT set (i.e., this is a leaf node):
                RETURN result
            IF condition(handTotal) IS TRUE:
                RETURN true_branch.evaluate(handTotal)
            ELSE:
                RETURN false_branch.evaluate(handTotal)

    CONSTRUCTOR(name = "The DealerTree"):
        CALL Player constructor
        CREATE standNode (result = false)
        CREATE hitNode (result = true)
        CREATE decisionTreeRoot (condition = [](int handTotal){
return handTotal < 17; }, true_branch = hitNode, false_branch =
standNode)
        SET m_decisionTreeRoot = decisionTreeRoot
    METHOD isHitting() const:
        // Use the decision tree to determine if the DealerTree
should hit
        RETURN m_decisionTreeRoot.evaluate(m_hand.getTotal())
    METHOD showHand(show_first_card = true):
        IF show_first_card IS TRUE:
            PRINT DealerTree's name and full hand
        ELSE:
            PRINT DealerTree's name and "[FACED DOWN]", then
display remaining cards
    METHOD shuffleDeck(): CALL deck.shuffleDeck()
    METHOD deal(): RETURN deck.deal()
    METHOD deckIsEmpty(): RETURN deck.isEmpty()
END CLASS
// Class GameStats
CLASS GameStats
    ATTRIBUTES:
        playerStats: unordered_map<string, pair<int, int>> //
Stores <player_name, <wins, losses>> (uses hashing internally)
        highScores: set<pair<int, string>> // Stores <money,
player_name> (sorted by money)
    CONSTRUCTOR(): Initialize empty maps/sets
    METHOD recordWin(player_name): INCREMENT wins for
player_name in playerStats
    METHOD recordLoss(player_name): INCREMENT losses for
player_name in playerStats
    METHOD updateHighScore(player_name, money):
        REMOVE old score for player_name from highScores
        INSERT new pair (money, player_name) into highScores
    METHOD getWins(player_name): RETURN wins for player_name
    METHOD getLosses(player_name): RETURN losses for player_name
    METHOD getWinRate(player_name): RETURN (wins / (wins +
losses)) * 100%
    METHOD displayStats():
        PRINT "===== PLAYER STATS ====="
        IF playerStats IS empty: PRINT "No stats yet."
        ELSE: PRINT formatted table of player stats (Player,
Wins, Losses, Win Rate)
    METHOD displayHighScores(top = 5):
        PRINT "===== HIGH SCORES ====="
        IF highScores IS empty: PRINT "No high scores yet."
        ELSE: PRINT formatted ranked list of top scores (Rank,
Player, Money)
END CLASS
// Class Game (inherits from GameStats)
CLASS Game (EXTENDS GameStats)
    ATTRIBUTES:
        players: deque of unique_ptr to Player
        DealerTree: DealerTree object
        stats: GameStats object
        actionLog: queue of string
        events: priority_queue of <int, string> (priority,
event_description)
        currentState: GameState enum
        // Game State Graph
        stateTransitions: map<GameState, set<GameState>>
        cardValues: map<string, int>
        suitNames: map<string, string>
    ENUM GameState: BETTING, DEALING, PLAYER_TURN, DEALER_TURN,
PAYOUT, CLEANUP, INVALID_STATE
    CONSTRUCTOR():
        SET currentState = BETTING
        Initialize cardValues and suitNames maps
        // Define Game State Graph transitions:
        SET stateTransitions for each state:
            BETTING -> {DEALING, CLEANUP}
            DEALING -> {PLAYER_TURN, CLEANUP}
            PLAYER_TURN -> {DEALER_TURN, CLEANUP}
            DEALER_TURN -> {PAYOUT, CLEANUP}
            PAYOUT -> {CLEANUP}
            CLEANUP -> {BETTING}
    METHOD addPlayer(player_object): ADD unique_ptr to
player_object to 'players' list, LOG action
    METHOD removePlayer(player_name): REMOVE player by name from
'players' list, LOG action
    METHOD play():
        IF no players: PRINT message, RETURN
        DealerTree.shuffleDeck(), LOG action
        LOOP while continue_playing is TRUE:
            setState(BETTING), placeBets()
            setState(DEALING), deal()
            setState(PLAYER_TURN), playerTurns()
            setState(DEALER_TURN), DealerTreeTurn()
            setState(PAYOUT), payouts()
            setState(CLEANUP), cleanup()
            findMinMaxMoney()
            PROMPT user to continue playing (y/n), READ choice,
SET continue_playing
    METHOD placeBets():
        PRINT "===== PLACING BETS ====="
        FOR each player in players:
            PROMPT player for bet amount
            WHILE bet is invalid: RE-PROMPT
            LOG action: player_name bet amount
    METHOD deal():
        PRINT "===== DEALING CARDS ====="
        CLEAR hands for all players and DealerTree
        DEAL two cards to each player (one at a time, iteratively)
        DEAL two cards to DealerTree (one at a time)
        displayTable()

    METHOD playerTurns():
        PRINT "===== DEALING CARDS =====" (Note: This is a copy-paste error from original code, should be "PLAYER TURNS")
        FOR each player in players:
            PRINT player's turn info, show player's hand
            IF player has Blackjack: PRINT "Blackjack!", LOG action, CONTINUE to next player
            WHILE player NOT busted AND player chooses to hit:
                DEAL new card to player, PRINT card received
                ADD card to player's hand
                IF player busted: PRINT "Bust!", LOG action
                ELSE: PRINT player's current total
            IF player NOT busted: PRINT "Stands!", LOG action
    METHOD DealerTreeTurn():
        PRINT "===== DEALERS TURN ====="
        CHECK if all players have busted
        DealerTree.showHand(true) // Reveal DealerTree's hidden card
        IF all players busted: PRINT "All players busted, DealerTree wins!", LOG action, RETURN
        WHILE DealerTree.isHitting():
            DEAL new card to DealerTree, PRINT card received
            ADD card to DealerTree's hand
            PRINT DealerTree's current total
        IF DealerTree.isBusted(): PRINT "DealerTree busts!", LOG action
        ELSE: PRINT "DealerTree stands!", LOG action
    METHOD payouts():
        PRINT "===== RESULTS ====="
        GET DealerTree's total, bust status, blackjack status
        FOR each player in players:
            GET player's name, total, bust status, blackjack status
            PRINT player's name:
            IF player busted: player.lose(), recordLoss(), PRINT outcome, LOG action
            ELSE IF DealerTree busted: player.win(), recordWin(), PRINT outcome, LOG action
            ELSE IF player has Blackjack AND DealerTree does NOT: player.win() (1.5x bet), recordWin(), PRINT outcome, LOG action
            ELSE IF player has NO Blackjack AND DealerTree HAS: player.lose(), recordLoss(), PRINT outcome, LOG action
            ELSE IF player total > DealerTree total: player.win(), recordWin(), PRINT outcome, LOG action
            ELSE IF player total < DealerTree total: player.lose(), recordLoss(), PRINT outcome, LOG action
            ELSE (tie): player.push(), PRINT outcome, LOG action
            updateHighScore(player_name, player_money)
    METHOD cleanup():
        REMOVE players with <= 0 money from the game, LOG action
        processEvents()
        displayStats()
        displayHighScores()
    METHOD findMinMaxMoney():
        IF no players: PRINT message, RETURN
        FIND player with max money and player with min money
        PRINT "===== PLAYER MONEY STATS ====="
        PRINT richest and poorest player with their money
    METHOD gameStateToString(state: GameState):
        CONVERT GameState enum to string representation (e.g.,
BETTING -> "BETTING")
        RETURN string

    METHOD validateTransition(from_state: GameState, to_state:
GameState):
        FIND 'from_state' in 'stateTransitions' map
        IF found:
            RETURN TRUE if 'to_state' is in the set of valid
transitions for 'from_state', ELSE FALSE
        ELSE:
            RETURN FALSE // No transitions defined for
'from_state'

    METHOD setState(new_state: GameState):
        TRY:
            IF current_state IS new_state OR
validateTransition(current_state, new_state):
                LOG action: "Transitioning from " +
current_state_string + " to " + new_state_string
                SET current_state = new_state
            ELSE:
                THROW new_state // Indicate invalid transition
        CATCH (any exception):
            PRINT "Invalid Transition!" // Handle error
    METHOD getState(): RETURN currentState
    METHOD displayTable():
        PRINT "===== TABLE STATUS ====="
        DealerTree.showHand(false) // DealerTree's initial hand
(one card hidden)
        FOR each player in players:
            PRINT player's name, money, bet, and hand details
    METHOD logAction(action_string): ADD action_string to
'actionLog' queue
```

```
    METHOD displayActionLog(): PRINT contents of 'actionLog'
queue
    METHOD queueEvent(event_string, priority_int): ADD
<priority, event_string> to 'events' priority_queue
    METHOD processEvents(): WHILE 'events' is NOT empty: PRINT
top event, REMOVE from queue
    METHOD displayMenu(): PRINT game menu options
    METHOD showWelcomeScreen(): PRINT welcome message and game
rules
    METHOD createPlayerProfile():
        PROMPT for new player name
        CHECK if player already exists
        IF not exists: CREATE new Player with 1000 money, ADD to
game, PRINT success
        ELSE: PRINT "Player already exists."
    METHOD loadPlayerProfile():
        PROMPT for player name
        FIND player profile
        IF found: PRINT welcome, current balance, wins, losses,
win rate
        ELSE: PROMPT to create new profile, CALL
createPlayerProfile if 'y'
// Main Program
FUNCTION main():
    CREATE game_instance of Game
    SET exit_game = FALSE
    LOOP WHILE exit_game IS FALSE:
        game_instance.displayMenu()
        READ user_decision
        SWITCH user_decision:
            CASE 1: game_instance.play()
            CASE 2: game_instance.createPlayerProfile()
            CASE 3: game_instance.loadPlayerProfile()
            CASE 4: game_instance.displayStats()
            CASE 5: game_instance.displayHighScores()
            CASE 6: SET exit_game = TRUE
            DEFAULT: PRINT "Invalid Input."
    PRINT "Goodbye!"
END FUNCTION
```

UML Class Diagram:



```
Diagram

Game
○ deque<unique_ptr<Player>> m_players
○ DealerTree m_DealerTree
○ GameStats m_stats
○ queue<string> m_actionLog
○ priority_queue<pair<int, string>> m_events
○ GameGraph m_currentState
○ map<GameGraph, set<GameGraph>> m_stateTransitions
○ map<string, int> m_cardValues
○ map<string, string> m_suitNames
● Game()
● void addPlayer(const Player& name)
● void removePlayer(const string& name)
● void play()
● void placeBets()
● void deal()
● void playerTurns()
● void DealerTreeTurn()
● void payouts()
● void cleanup()
● void findMinMaxMoney()
● string stateToString(GameGraph state) const
● bool validateTransition(GameGraph from, GameGraph to) const
● void setState(GameGraph state)
● GameGraph getState() const
● void displayTable() const
● void logAction(const string& action)
● void displayActionLog() const
● void queueEvent(const string& event, int priority)
● void processEvents()
● void displayMenu() const
● void showWelcomeScreen() const
● void createPlayerProfile()
● void loadPlayerProfile()

DealerTree
□ Deck m_deck
□ unique_ptr<DecisionNode> m_decisionTreeRoot
● DealerTree(const string& name)
● bool isHitting() const
● void showHand(bool showFirstCard) const
● void shuffleDeck()
● Card deal()
● bool deckIsEmpty() const

GameStats
□ unordered_map<string, pair<int, int>> m_playerStats
□ set<pair<int, string>> m_highScores
● GameStats()
● void recordWin(const string& playerName)
● void recordLoss(const string& playerName)
● void updateHighScore(const string& playerName, int money)
● int getWins(const string& playerName) const
● int getLosses(const string& playerName) const
● double getWinRate(const string& playerName) const
● void displayStats() const
● void displayHighScores(int top) const

Player
◇ string m_name
◇ Hand m_hand
◇ int m_money
◇ int m_bet
● Player(const string& name, int money)
● string getName() const
● int getMoney() const
● int getBet() const
● const Hand& getHand() const
● Hand& getHandRef()
● bool placeBet(int amount)
● void win()
● void lose()
● void push()
● bool isHitting()
● void showHand(bool showFirstCard) const
● bool isBusted() const
● bool isBlackjack() const

DealerTreeDecisionNode
○ bool result
○ unique_ptr<DecisionNode> true_branch
○ unique_ptr<DecisionNode> false_branch
● function<bool(int)> condition
● DecisionNode(bool res)
● DecisionNode(function<bool(int)> cond, unique_ptr<DecisionNode> t_branch, unique_ptr<DecisionNode> f_branch)
● bool evaluate(int handTotal) const

Hand
◇ deque<Card> hand_cards
● Hand()
● void add(const Card& card)
● void clear()
● int recursiveTotals(deque<Card> getHand, int getTotal) const
● int getTotal() const
● bool isBusted() const
● bool isBlackjack() const
● deque<Card>::iterator begin()
● deque<Card>::iterator end()
● deque<Card>::const_iterator begin() const
● deque<Card>::const_iterator end() const
● void toString() const

Card
□ string m_suit
□ string m_rank
□ bool m_faceUp
● Card(string decSuit, string decRank, bool setFace)
● string getRank() const
● string getSuit() const
● bool isFaceUp() const
● void flip()
● bool operator==(const Card& other) const
● bool operator<(const Card& other) const

Deck
□ list<Card> cards
□ stack<Card> discardPile
● Deck()
● void shuffleDeck()
● Card deal()
● void addToDiscardPile(const Card& card)
● void retrieveCardsFromDiscardPile()
```