Black Jack Project Specification

Blackjack Description:

This program is a simple version of the single player casino game BlackJack. The Card class keeps track of all the cards in a deck. The Hand class adds or clears the cards to the hand and also calculates the total. Classes Player and House inherit the class GenericPlayer and displays the status (Hitting, win, lose etc) of each player and the house respectively. The Deck class inherits the Hand Class and takes care of shuffling, populating and dealing the cards. The Game Class keeps track of the player involved and runs the BlackJack game. The student needs to follow the rules of the casino game BlackJack and mimic them in this program.

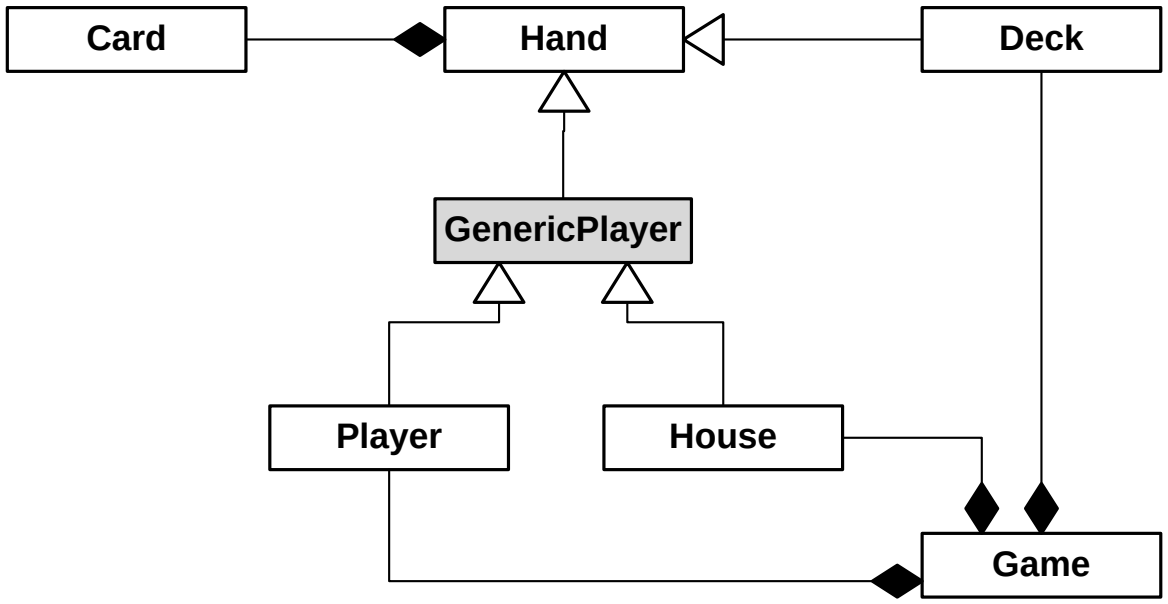
Overview

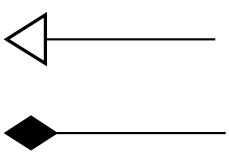
In blackjack, the cards are valued as follows:

- An Ace can count as either 1 or 11.
- The cards from 2 through 9 are valued at their face value.
- The 10, Jack, Queen, and King are all valued at 10.

The suits of the cards do not have any meaning in the game. The value of a hand is simply the sum of the point counts of each card in the hand. For example, a hand containing (5,7,9) has the value of 21. The Ace can be counted as either 1 or 11. The value of the Ace is not specified. It's assumed to always have the value that makes the best hand. Once all the players are ready, the dealer will deal the cards to the players. He'll make two passes so that the players and the dealer have two cards each. The dealer will flip one of his cards over, exposing its value. Once the cards are dealt play proceeds around the table. Each player in turn indicates to the dealer how he wishes to play the hand. The most common decision a player must make during the game is whether to draw another card to the hand ("hit"), or stop at the current total ("stand"). A blackjack, or natural, is a total of 21 in your first two cards. The basic premise of the game is that you want to have a hand value that is closer to 21 than that of the dealer, without going over 21.

Class Diagram Overview:



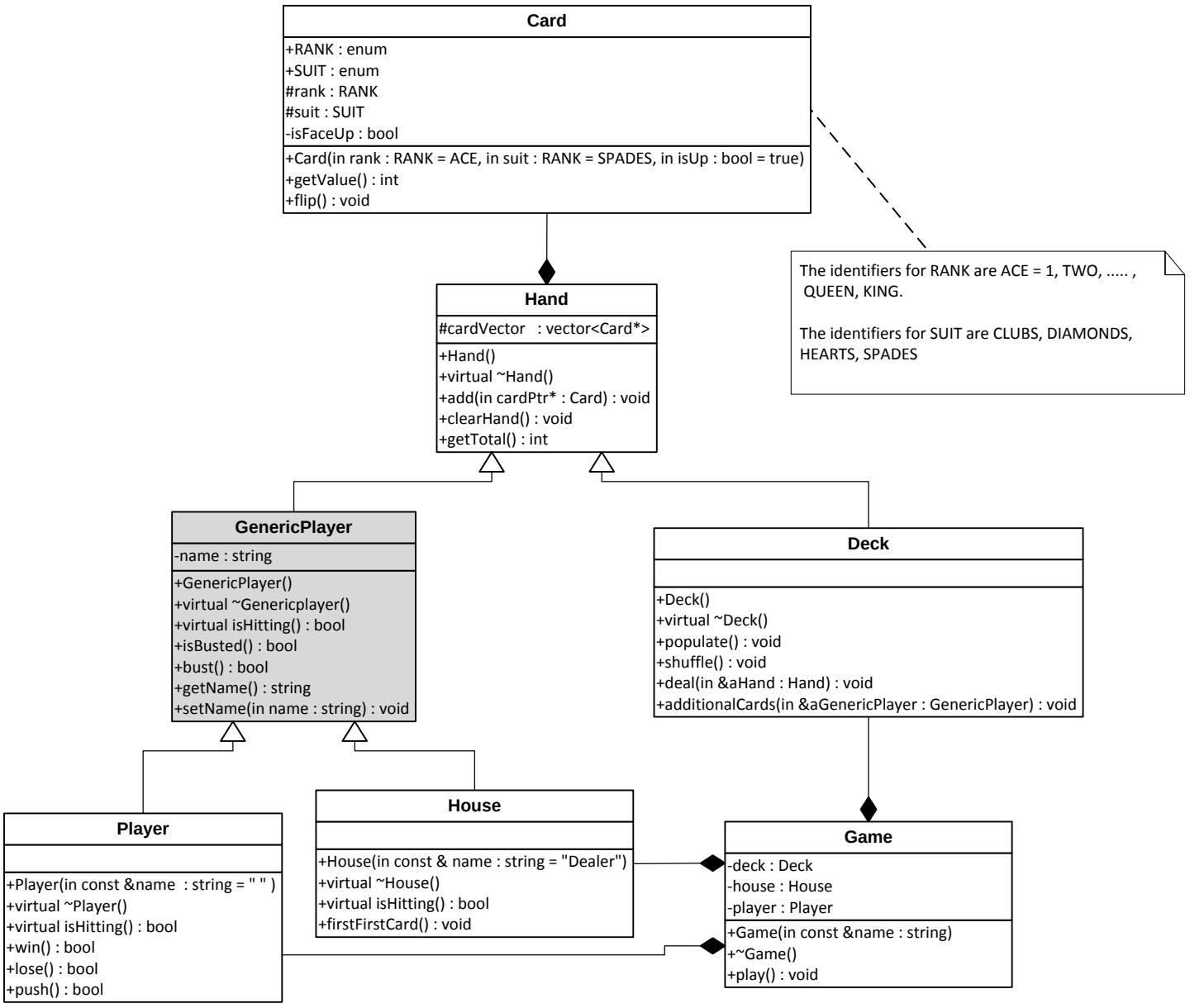


Inheritance (Is - A Relationship)

Composition (Has – A Relationship)

Class	Base Class	Description
`Card`	None	A Blackjack playing card
`Hand`	None	A blackjack hand. A collection of `Card` objects
`Deck`	`Hand`	A Blackjack deck. Has extra functionality that `Hand` doesn't, such as shuffling and dealing.
`GenericPlayer`	`Hand`	A generic Blackjack player. Not a full player, but the common elements of a human player and the computer player. This is an abstract class.
`Player`	`GenericPlayer`	A human Blackjack player.
`House`	`GenericPlayer`	The computer player, the house.
`Game`	None	A Blackjack game

Detailed Class Diagram:



All accessor functions, predicate functions, bust, win, lose, push, and flipFirstCard functions are constants

Access Specifier Notations
- Private
+ Public
# Protected

Additional Notes:
All inheritances are public.
If pseudocode is not provided for destructors then the body of the function remains empty.
Generic Player is an abstract class.
isHitting in GenericPlayer class is a pure virtual function

Card Class and GenericPlayer Class both contain overloaded << operator.
Write an appropriate prototype for these functions. The pseudocode for these functions will be provided later.

Card Class

```
Constructor
	Initialize the member variables using the member initializer list in the constructor. The body
of the constructor remains empty.
END FUNCTION

Function getValue
	Return the value of the card if the card is facing up. Otherwise return 0.
	NOTE: The king, queen, jack should also return a value 10
END FUNCTION

Function flip()
	Flips a card. Face up becomes face down, and face down becomes face up
END FUNCTION
```

Hand Class

```
Constructor
	Call the reserve() on CardVector and pass a value of  7
END FUNCTION

Destructor
	Call the function clearHand()
END FUNCTION

Function add
	Adds a card to the hand.
	Call push_back() on the cardVector and pass the cardPtr as its argument
END FUNCTION

Function clearHand
	Clears all cards from the hand
	Declare an iterator of the type vector<Card*>
	using the iterator delete each *iter [name of the iterator]
	Set the *iter to NULL -- Good practice
	Call function clear() on cardVector
END FUNCTION

Function getTotal
	Returns the total value of the hand.

	IF cardVector is empty [Hint: empty is a function defined in vector class] THEN
		return 0
	ENDIF

	IF the value of the first card is 0 THEN
		return 0
		Hint: use cardVector[0]-> getValue()
	ENDIF

	To add up the card values create a const_iterator
	LOOP through the vector (using the iterator)
		calculate the total value for all the cards [Hint use (*iter)->getValue() to
		get the value of each card.]
	END LOOP
```

```
       LOOP through the hand and check if it contains an Ace
              IF the total IS LESS THAN AND EQUAL TO 11, THEN
                     add 10 to the total [since we have already set the ace to 1].
                     Return the total.
              ENDIF
       END LOOP
END FUNCTION
```

Generic Player Class

```
Constructor
       Initialize the member variables using the member initializer in the constructor.
       The body of the constructor remains empty.
END FUNCTION

Function isBusted
       Indicates whether the generic player is busted.

       IF getTotal() IS GREATER THAN 21 THEN
              Return appropriate value
       ELSE
              Return appropriate value
       END IF
END FUNCTION

Function bust()
       Print the name and display that the player has busted
END FUNCTION

NOTE: WRITE APPROPRIATE CODE FOR SETTER AND GETTER FUNCTIONS FOR THIS CLASS
```

Player Class

```
Constructor
       Call the base class constructor to initialize the member variable.
END FUNCTION

Function isHitting
       Concrete function. Indicates whether the player wants to hit
       Using name of the player prompt to check if he wants to hit and return appropriate values
END FUNCTION

Function win
        Print the name and display that the player has won
END FUNCTION

Function lose
        Print the name and display that the player has lost
END FUNCTION

Function push
        Print the name and display that the player has pushed
END FUNCTION
```

House Class

```
Constructor
       Call the base class constructor to initialize the member variable.
END FUNCTION
```

```
Function isHitting
        Concrete function.  Indicates whether the dealer wants to hit

        IF the value returned by getTotal() IS LESS THAN EQUAL TO 16 THEN
                The House hits (return appropriate values, use if..else, if needed)
        ENDIF
END FUNCTION

Function flipFirstCard
        Flips over the first card.
        IF cardVector is not empty THEN
                call the flip function on the first element of the vector.
        ELSE
                Display an appropriate message indicating that there are no cards to flip
        END IF
END FUNCTION
```

Deck Class

```
Constructor
        Call the function cardVector.reserve(52)
        Call function populate()
END FUNCTION

Function populate
        Creates a standard deck of 52 cards.
        Call function clearHand()
        To create standard deck iterate through all ranks and suits [use nested loop on each enumerator]
        call function add(new Card(static_cast<Card::RANK>(r), static_cast<Card::SUIT>(s)))
END FUNCTION

Function shuffle
        Shuffles cards.
        Call function random_shuffle(cardVector.begin(), cardVector.end())
        [You will need to include the standard library named algorithms for this to work]
END FUNCTION

Function deal
        Deals one card to a hand. A hand is any player.
        IF cardVector is not empty [use empty() in vector library] THEN
                call aHand.add(cardVector.back())
                call cardVector.pop_back()
        ELSE
                display an appropriate message indicating that you are out of cards and are unable to
                deal.
        END IF
END FUNCTION

Function additionalCards
        Gives additional cards to any player for as long as, that player can and wants to hit.

        WHILE generic player object is not busted and keeps hitting
                call the function deal and pass the generic player object to it.
                Display generic player object using cout [this will invoke the overloaded insertion
                operator]

                IF generic player object is busted THEN
                        call the function bust() using the generic player object
                END IF
        END WHILE
END FUNCTION
```

Game Class

```
Constructor
       Call the setName function for player object and pass it the name.
       Call function populate() using the deck object.
       Call function shuffle() using the deck object
END FUNCTION


Function play
       Deal initial 2 cards to each player. [You will need a loop that runs twice and call deal
       function for each player]

       Using the house object call function flipFirstCard().

       Display player object using cout [to invoke the overloaded insertion operator]
       Display house object using cout [to invoke the overloaded insertion operator]

       Call the additionalCards() using the deck object and pass the player object to it.

       Using the house object call function flipFirstCard().

       Display house object using cout [to invoke the overloaded insertion operator]

       Call the additionalCards() using the deck object and passthe house object to it.

       IF the house has busted [call isBusted()] THEN
              IF player has not busted THEN
                     invoke the win function using the player object.
              END IF
       ELSE

              IF player has not busted THEN
                     compare the total score of the player with the house and display the appropriate
                     winning, losing, push messages. [requires nested if..else statements]
              END IF
       END IF

       call the clearHand function for player object

       Call the clearHand() funtion for house object

END FUNCTION
```

Main

```
Print appropriate welcome message to begin the game

Create a character variable called again and initialize it to 'y'

Prompt for the player name and store it in variable called name.

Create a game object and pass the name to it

Call the function play() using the game object. [Use a sentinel controlled loop around this to prompt
the player and check if he wants to play again to stay in the game. You exit the game when the user
chooses 'no']

END FUNCTION
```

overloaded << operator for Card Class

Create 2 constant string arrays named RANKS and SUITS (RANKS will contain elements like “0”, “A” etc and SUITS will contain “S”, “C” etc) and use these string arrays to print appropriate rank and suit when displaying the card. The enumerators created in the card class will work like the index for the above arrays.

```
IF the card is facing up THEN
       store the rank and suit in the output object.
ELSE
       store "XX"
END IF
```

Return output object [See overloaded << class example]

HINT: This function should print JH for jack of hearts or 5S for 5 spades etc for each card.

END FUNCTION

overloaded << operator for GenericPlayer Class

Store the name followed by a tab space in the output object.

Create a vector iterator of type Card*

```
IF the cardVector for the generic player object is not empty THEN
       LOOP through all cards for that player
              store the value of the card followed by the tab space [ use *(*iterator_name)]
       END LOOP

       IF the total score for the player is NOT EQUAL TO 0 THEN
              append the total to the output object
       END IF
ELSE
       append "<empty>"
END IF
```

Return output object

Example Output: Say the name of the player is Player1 and the player has 5 spades and a jack of hearts and the dealer’s first card is still facing down then this function should print an output as follows:

```
Player1       5s     JH     (15)
House         XX     9C
```

END FUNCTION

	Measurement of performance of aspect.
Aspect	Objectives Substantially Met
Correctness Max Points : 10	1. Classes implement appropriate inheritance where necessary 2. All accessor, predicate and other specified functions are constant functions 3. The function prototypes for overloaded insertion operators are written correctly and are placed in appropriate classes/files. 4. All vectors (wherever specified in the specification) utilize correct iterators for traversal. 5. Student correctly initialize the member variables and as specified in the specification. **(5 Points)** 6. Program runs and completes all required tasks. **(3 Point)** 7. Program handles special cases. **(1Point)** 8. Executes without errors (Logical). **(1 Point)**
User Interface Max Points : 2	1. Specification is followed correctly and includes all appropriate, descriptive and user friendly input prompts **(1 Point)** 2. Specification is followed correctly and output is user-friendly, and clearly describes what is expected from the user. **(1 Point)**
Documentation Max Points : 6	1. Program contains required project commenting header (for each file) with correct format and clearly describes the purpose of the project. **(1.5 Point)** 2. Program contains function commenting headers clearly describing the purpose of the function. **(3 Point)** 3. Program contains comments for the statements where ever necessary. **(1 Points)** 4. Self-commenting variable, method and class names are used. **(0.5 Points)** **NOTE: Documentation must correctly outline inheritance, composition, member initializer list and calls for the base class constructor where ever required.**

Construction Max Points : 2	1. OOPs concepts must be used ie., in this case the C++ classes must be used to complete the program. Procedural or any other programming is not allowed. 2. Tasks are modularized and well defined. 3. Each class should have a .h file for class declaration, and a .cpp file for the implementation. 4. All control structures must use {} even if the control structure contains only one statement. 5. Source code is logically laid out, using current best practices. **(1 Point)** 6. Code uses proper control structures 7. The students should use camel-casing notation only throughout the program. Students must use appropriate variables names [identifiers] and functions throughout the program. 8. Students must use data types efficiently. 9. Program follows the required specification and uses appropriate set and get functions wherever necessary **(1 Point)**

Correctness Score (10 points) ____ User Interface Score (2 points) ____

Documentation Score (6 points) ____ Construction Score (2 points) ____

Total (20 points) ____ Actual = Total /2 Max Score 10 ____