Vishal Yathish
506-166-275
COM SCI 32
12 October 2023

Programming Project #1: Blackjack Report

This report will be divided into three parts: Project Implementation, Project Testing, and Project Debugging. The first will be a present-tense walkthrough of my implementation of Blackjack, as defined in the assignment instructions. These will include my notes as I work, my questions, and the logic I use as I solve each part. The second is a description of the testing I subject my code to prior to the final submission, as I require an excellent grade on this project. This will include the test cases provided in the skeleton as well as other test cases created by me. The third part will be a real-time walkthrough of any errors that I encounter when I have completed my first pass at the project, and my attempts to work through those errors as they arise. This may be longer or shorter depending on how 'buggy' my code is.

PART 1: PROJECT IMPLEMENTATION

I'm starting this project by first getting an understanding of how Blackjack works – having never played this game before personally. I used the simulation of the game provided to us in the assignment instructions and came away with the following points of note.

1. The goal of the game is to get the total value of your hand to 21, or at the very least closer to 21 than your opponent's hand. This is where strategy cards are useful, as they show you in which combinations what move to take.
2. The King, Queen, and Jack all count for the same value, 10. This is in spite of the fact that there is already a 10 card in the deck.

I began this project from the *Card.cpp* file, as it seemed the simplest and most straightforward starting place.

Additional Note: As I'm working on this project, there seemed to be some chatter in the discussion forums about "double-splitting," and some error that Professor Howard made. I really don't understand what they are talking about – something about splitting or split hitting – so I am just going to ignore it for now. I will come back to it when I need to implement the strategy card.

Card.cpp

The first task is to modify the Card class to include the setter methods, methods that allow the user to assign a new *Suit* object to the mSuit variable, and a new *Face* object to the mFace variable.

These seem relatively straightforward, with one line of code for each – simply setting the attribute variable to be the new object. However, I am worried that there may be more to it. Do we need to use enumerations for something, like perhaps if statements to say which face it is exactly? Or is this it?

The second task is to add the getValue method to the class. This would presumably find the value of the card for the purposes of adding to the score. When implementing this method, I am using a switch statement, which each of the face enumerations being a case. They go in ascending order, with an Ace having a value of 1, and the Ten, Jack, King, & Queen Cards all sharing a value of 10 – according to the rules of Blackjack I read online.

Hand.cpp

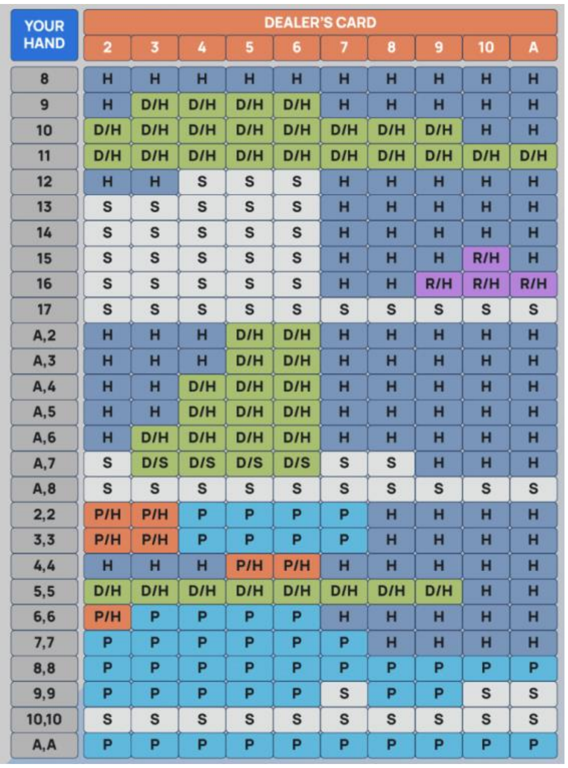
This class contains the bulk of this project. We need to implement three methods: evaluateHand(), isPair(), and isSoft(). I will start with the latter two as they are shorter and easier to implement.

The isPair method returns True when both of the Hand's cards contain the same value and returns False otherwise. My first thought is to call the getValue method we implemented in the class Card for both of the contained cards, mCard1 and mCard2. However, this would not work as Tens, Jacks, Queens, and Kings all have the same value, so the isPair method would be unable to differentiate between two Jacks and one Jack and one King. I looked at the getValue method again, to see if this was an error in my code that I needed to correct – perhaps by providing character values for the Jacks ('J'), Queens ('Q'), and Kings ('K'). However, this would not be right, as the variable 'result', as provided by the skeleton, is an integer.

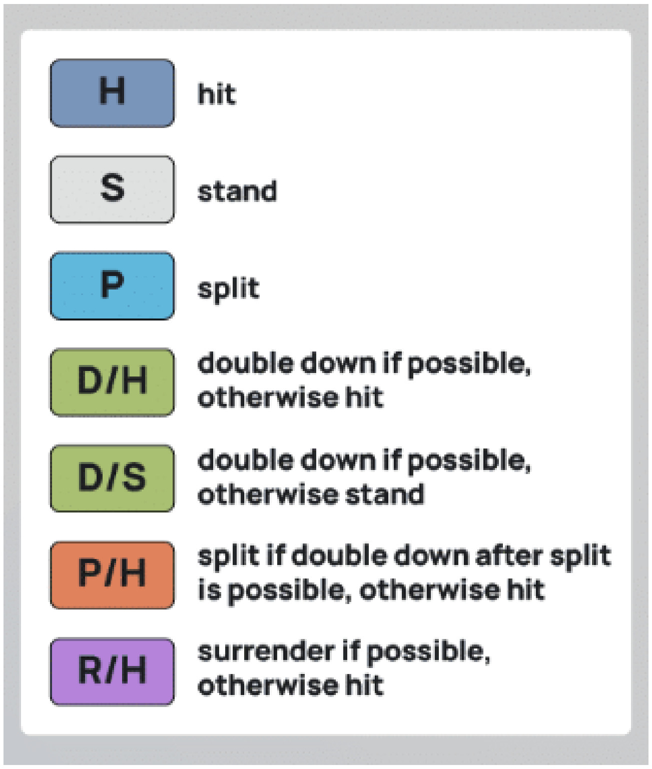
So, we need to utilize a different approach for the isPair method. I'm wondering whether we can use the getFace method from the *Card* class and compare them. Can you use the == operator to compare two Face objects? Otherwise, you would have to use nested if statements, for each face, asking if mFace1.getFace() == Face::ACE, Face::DEUCE, etc. Either case, we would have to use the == operator to compare them, so using it directly in an if statement should work, and it takes a fraction of the time.

We apply this same logic to isSoft(). Since we only require one of the cards to be an Ace, if either of the cards fits this criterion, then the function can return True. Two simple if statements are all that are required, if the above logic is accurate and we can compare Face objects using the == operator, the first to check if mCard1.getFace() == Face::ACE and the second to check if mCard2.getFace() == Face::ACE. In either case, the function returns True. If not, the function returns False.

Finally, we get to the evaluateHand method, which implements the strategy table provided in the project description.



YOUR HAND	DEALER'S CARD									
	2	3	4	5	6	7	8	9	10	A
8	H	H	H	H	H	H	H	H	H	H
9	H	D/H	D/H	D/H	D/H	H	H	H	H	H
10	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H	H	H
11	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H
12	H	H	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	H
15	S	S	S	S	S	H	H	H	R/H	H
16	S	S	S	S	S	H	H	R/H	R/H	R/H
17	S	S	S	S	S	S	S	S	S	S
A,2	H	H	H	D/H	D/H	H	H	H	H	H
A,3	H	H	H	D/H	D/H	H	H	H	H	H
A,4	H	H	D/H	D/H	D/H	H	H	H	H	H
A,5	H	H	D/H	D/H	D/H	H	H	H	H	H
A,6	H	D/H	D/H	D/H	D/H	H	H	H	H	H
A,7	S	D/S	D/S	D/S	D/S	S	S	H	H	H
A,8	S	S	S	S	S	S	S	S	S	S
2,2	P/H	P/H	P	P	P	P	H	H	H	H
3,3	P/H	P/H	P	P	P	P	H	H	H	H
4,4	H	H	H	P/H	P/H	H	H	H	H	H
5,5	D/H	D/H	D/H	D/H	D/H	D/H	D/H	D/H	H	H
6,6	P/H	P	P	P	P	H	H	H	H	H
7,7	P	P	P	P	P	P	H	H	H	H
8,8	P	P	P	P	P	P	P	P	P	P
9,9	P	P	P	P	P	S	P	P	S	S
10,10	S	S	S	S	S	S	S	S	S	S
A,A	P	P	P	P	P	P	P	P	P	P



H	hit
S	stand
P	split
D/H	double down if possible, otherwise hit
D/S	double down if possible, otherwise stand
P/H	split if double down after split is possible, otherwise hit
R/H	surrender if possible, otherwise hit

I am now going back and parsing through what is being discussed in the discussion forum. The Enums.h file in the skeleton apparently contains one minor error – there is one extra choice DOUBLESPLIT – that is not represented in the strategy card above. I made the modification in my version, so I don't have to redownload the skeleton.

According to Professor Howard – per the discussion forum – the choice enumerations equate to the symbols on the card above as follows:

H = Choice::HIT
S = Choice::STAND
P = Choice::SPLIT
D/H = Choice::DOUBLEHIT
D/S = Choice::DOUBLESTAND
P/H = Choice::SPLITHIT
R/H = Choice::SURRENDERHIT

The strategy card (when considered top to bottom), is divided into three rough sections. The first is when the hand is neither a pair nor soft. In these cases, only the total value of the hand is considered, and compared to the dealer's card. The second section is when the hand is soft, meaning that there is one ace – the case where there are two aces is dealt with later. The third section is when the hand is a pair, meaning that both cards in the hand have the same value, including if both are Aces.

This division can be replicated using an if statement. I checked if isSoft() is true in the first if statement, and isPair() is true in the following else-if statement. Thus, the program will check if the hand is soft first, meaning that we need to determine if the hand is a pair of Aces here.

The pair of Aces is an easy case, as regardless of dealer hand, the card recommends splitting. The next sub-case is if the hand is soft, but not a pair. I am going to create a helper function to assist with this one, as we don't know which of the cards (mCard1 or mCard2) is an Ace and we don't want to make assumptions – as that could be reversed in a test case. Also, we need to know the value of any non-Ace card to fill out the rest of the strategy card. My helper function would provide us with that information. It returns the value of the non-Ace card in the Hand; this operates under the assumption that it is only used if the hand is soft, otherwise it will only return 0.

My method is implemented, and I have added new sub-sub cases for each non-Ace card value represented on the strategy card. Now, within each, I need to add sub-sub-sub cases for each dealer card value for each non-Ace card value represented. I am using switch statements, but if these turn out to be an issue, I will use if statements; the logic will be the same.

I added one more variable to the entire evaluateHand method, which returns the value of the dealer's card. This is what I am going to be using in all of my sub-sub-sub switch cases, iterating over values 1 (Ace) to 10.

And just like that, I'm done with my first iteration of the project. It did not take as much time as I thought. Since I had all of the if statements and sub-if statements already structured, all I needed to do was copy and paste a switch statement to iterate over all possible dealer card values and modify the output choice based on the strategy card. The program is extremely long, around 853 lines – it is probably not the most efficient or aesthetic method of solving the problem. However, it does work.

PART 2: PROJECT TESTING

All of the test cases provided in the skeleton work well. The code runs without any issues. I added a string at the end "All tests passed" to make sure that my code didn't quit without notifying me, and it was successfully outputted. Using the Card objects established in the skeleton, I used and ran the following test cases of my own:

```
Hand h(two, six);
Hand h(six, two);
```

These two tests follow the case where the hand is neither soft nor a pair. It also tests reversing the order of the cards to see whether that makes an impact on the program. Both tests ran successfully.

```
Hand h(ace, six);
Hand h(six, ace);
```

These two cases test when the hand is soft but is not a pair. Again, I wanted to make sure that the program would run regardless of whether the Ace was the first or the second card. Both tests ran successfully.

```
Hand h(king, jack);
Hand h(jack, king);
Hand h(king, ace);
Hand h(ace, king);
```

All four of the cases listed here are those that were not represented by the strategy card. In all of these cases (regardless of dealer card), the program should revert to the default choice, which is STAND. All tests ran successfully and had the same output, regardless of order.