Object Oriented Programming

Math in Casinos: Video Poker

Project Report made by:

João Beirão (75988)

João Raposo (75323)

Luís Alves (75189)

Table of Contents:

14th May 2017

Introduction

This project is within the scope of the Object Oriented Programming course from Instituto Superior Técnico in Alameda. The goal is the implementation of the Video Poker Game, with the Double Bonus 10/7 variant, using Java. The main component involves three playing modes:

- Interactive, where the game is played through commands in the command line;
- Debug, where the game is loaded from two files (commands and cards);
- Simulation, where the game is played automatically using a perfect strategy in order to perform a statistical analysis of its functioning.

The initial approach consisted of designing an UML (Unified Modelling Language) which represented the complete structure of the game classes. Then, the implementation of the solution was accomplished while performing tests to confirm the consistency of the process.

A functional and extensible solution for the Video Poker Game is proposed according to the stipulated details. A graphical user interface (GUI) using Swing is also provided.

This report is organized in three main sections. The first part explains a few relevant decisions and approaches used. The second chapter describes the extensibility of the developed solution and the last section includes an analysis and discussion of the results obtained.

Evaluation of hands

In order to successfully implement the Video Poker game the program needs to detect if the hand of the player matches any of the hand combinations of the paytable. Similarly, in order to provide the best strategy to win, the program should test the possibilities using the 5 cards the player has in the hand. As expected the number of verifications, both to evaluate a hand and to give an advice according to the hand, is quite high. In order to minimize the computation needed to do both of the actions some of the possible strategies to run a large set of hands were analyzed. For example, arrays of cards for counting, where for each suit and for each card there would be a counter with the number of times each case occurred in the hand. Considering this approach, a solution based on bit masks and a hash array was implemented.

Approach to the problem

This approach is based on the usage of the `long` data type to represent the 52 cards from the deck. In Java SE 8 and later, the `long` data type can represent an unsigned 64-bit long, with a minimum value of 0 and a maximum of 2^{64}-1. Therefore, if each card has a bit associated it is possible to represent the whole deck and still have bits unused. The following images explain which bits have cards associated.

Figure 1 - 16 bits representation of a complete set of cards from Clubs suit.

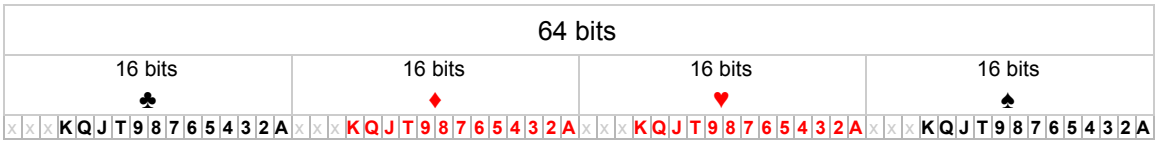

Figure 2 - 64 bits representation of a complete deck.

For example, given a hand with the cards K♣ Q♦ J♥ 9♥ 7♠ , one `long` with value 1152930300783755328 will represent the hand. The decimal value in binary representation is displayed in Figure 3.

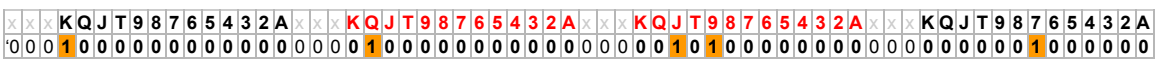

Figure 3 - Binary representation of a hand.

Since all the evaluations will be rank wise or suit wise (or both), the first step to use this representation is to separate a hand in those two parts. This step will also allow to use integers from now on. An array of integers with length equal to 4 will represent each suit. For each integer, the respective 16 bits of that suit will be taken from the hand by a bitwise AND

with a mask of 16 bits set (1111111111111111) that will be shifted of 16 times at a time. For the Figure 3 example of a hand, the third position of the array (corresponding to Hearts suit) is represented in Figure 4.

suit[2]													
K	Q	J	T	9	8	7	6	5	4	3	2	A	
0	0	1	0	1	0	0	0	0	0	0	0	0	

Figure 4 - Binary representation of the Hearts suit in the example of a hand in Figure 3.

For the integer representation of the ranks, each position of the array of suits will be bitwise OR'ed (suit[0] OR suit[1] OR suit[2] OR suit[3]) resulting in a 16 bits representing if a specific rank exists in the hand. For the Figure 3 hand example, the resulting integer representing all the ranks that the hand has is represented in Figure 5.

ranks													
K	Q	J	T	9	8	7	6	5	4	3	2	A	
1	1	1	0	1	0	1	0	0	0	0	0	0	

Figure 5 - Binary representation of the ranks for the example hand.

Lastly, in order to have all the variables to proceed to the actual evaluation of the hand we use as base to run this representation a *hash array*. The array size is given by the combination of 5 cards in a 13 cards set (7936 to be precise) and each position of the array will have the number of bits set (equal to 1) in the hand represented by that position. For example, if the array is named *numberOfRanks* and 1280 is the decimal representation of suit[2] then in the position 1280 of the array *numberOfRanks* we will have the decimal value 2 since two of the bits of suit[2] are one. The array is built using a static initializer in order to have it ready as soon the program starts and to ensure it is built only once and it remains the same until the end of the program.

After the initial setup, all the variables are ready to test each type of hand combinations. To test each hand, bit masks were used to select the bits of interest for a specific case.

Let's take the example of a Royal Flush evaluation (a Ten, a Jack, a Queen, a King and an Ace of the same suit). The relevant bits are the ones correspondent to a Ten, a Jack, a Queen, a King and an Ace. In order to avoid unnecessary verifications, when possible, an initial verification was previously made, checking bit by bit if there was a relevant bit set. For the Royal Flush case, an initial condition would be verifying if we actually have 5 cards of the same suit in the hand. If yes, then we will evaluate if they are the relevant ones. The Figure 6 shows the code and the binary representation of the mask used.

```
royalflushMask = 0x1E01;
if (numberOfRanks(suit[0]) == 5 ||
   numberOfRanks(suit[1]) == 5 ||
   numberOfRanks(suit[2]) == 5 ||
   numberOfRanks(suit[3]) == 5){
        mask = ranks ^ royalflushMask;
        if (mask == 0){
                ...
        }
}
```

royalflushMask												
K	Q	J	T	9	8	7	6	5	4	3	2	A
1	1	1	1	0	0	0	0	0	0	0	0	1

Figure 6 - Snippet of code used for the Royal Flush evaluation and binary representation of Royal Flush binary mask used.

Regarding the advice for which cards to hold according to the perfect strategy, let's use the example of the 3 to a flush with two high cards. The initial verification, in this case, is to check if there are at least two high cards in the hand. Then, we verify if for each suit, there are 3 cards, and only then we verify if there are two high cards in those 3 cards of the same suit. When all those conditions are verified, we take the rank and the suit of the cards found, present in the suit integer. The following figure represents the code used and the relevant mask for this case.

```
highcardMask = 0x1C01;
if (numberOfRanks((ranks & highcardMask)) >= 2){
        for (suitNb = 0; suitNb < 4; suitNb++){
                if (numberOfRanks(suit[suitNb]) == 3){
                        if (numberOfRanks(suit[suitNb]
& highcardMask) >= 2){
                                ...
                                // Evaluates which cards
                                   are the 3 to a flush
                        }
                }
        }
}
```

highcardMask												
K	Q	J	T	9	8	7	6	5	4	3	2	A
1	1	1	0	0	0	0	0	0	0	0	0	1

Figure 7 - Snippet of code used for the 3 to a flush with 2 high cards case evaluation and binary representation of the High cards binary mask used.

Extensibility of the solution

One of the requirements included in the assignment of this project was the implementation of an extensible solution concerning Video Poker variants. A few examples of existing variations of the game were presented, however, this project focus was on Double Bonus 10/7 variant.

In order to achieve the desired extensibility, an interface called `Variants` was created with the common and relevant methods for Video Poker variations. The differences between variants are mainly related to the alternatives for winning hands and the payback credit according to both the betting value and the final hand.

When developing a class that implements this interface, the programmer must provide two attributes:

- `private final static String[] winningHands` → an array of type `String` which contains the list of winning hands of the specific variation.
- `private final static int[][] paytable` → a matrix of type `Integer` where each row corresponds to a winning hand (according to the order specified in the `String` array) and each column corresponds to the betting value (from 1 to 5, in order from left to right).

They are both final because they correspond to the core of the game, so immutable, and also static since they are attributes of the class and not of the object.

Regarding the methods, five operations were considered common for every variation:

- `int evaluation(Card[] hand)` → Method that receives a set of 5 cards (hand) and which returns -1 in case of lost, and the respective index of the `winningHands` array, otherwise.
  Example: For Double Bonus, the alternatives for winning hands are {Royal Flush, Straight Flush, Four Aces, Four 2-4, Four 5-K, Full House, Flush, Straight, Three of a Kind, Two Pair, Jacks or Better}. In case of the input 5 cards correspond to Four Aces, this method must return 2 (considering the indexes starting in 0). In case of the input 5 cards correspond to Straight, this method must return 7.
- `int[] getAdvice(Card[] hand)` → Method that receives a set of 5 cards (hand) and returns an array of type `Integer` with the indexes of the cards to hold according to the perfect strategy (cards' indexes are from 1 to 5). Note that the order of the indexes is not important since the array will be sorted afterwards; and in case the suggestion is to hold zero cards, the return must be an array of `Integer` with length 0.
  Example: For Double Bonus, in case the input 5 cards are KC QC JC TC 9C (Straight Flush), this method must return [1, 2, 3, 4, 5]. In case of the input 5 cards are AH AD AS 2C 2S (Three Aces), this method must return [1, 2, 3].

- `String[] getWinningHands()` → Method that returns the array of `String winningHands`. This getter is important for defining the object `data` from class `Statistics` in class `Game` since the statistical analysis of the game is dependent on the alternatives for winning hands. Note that the class `Statistics` is also prepared for dealing with other variants since the table with the average scores of the game is also dependent on the variation of Video Poker being used.
- `String getWinHand(int i)` → Method that returns the `String` from the `winningHands` array associated with the input index. This getter is important in method `doEvaluation()` from class `Game` for printing in the terminal a message with the winning hand combination. Note that this message is also extensible according to the variant on use.
- `int getPaytableValue(int row, int column)` → Method that returns the credit earned by a specific winning hand combination (input `row`) according to the value of the bet (input `column`). This getter is important in method `doEvaluation()` from class `Game` for incrementing the credit won by the player. Note that the money earned when the player wins is also dependent on the variation of the game and, for that reason, must to be extensible.

Discussion of Results

In order to check the probabilities of each hand, the solution was tested 10 times, each time with different values of initial credit and number of deals (ranging from 100k to 1Million).
By dividing each average number of hands by the total number of deals and comparing them to the probabilities expected (that are shown at the "10/7" Double Bonus table in https://wizardofodds.com/games/video-poker/strategy/double-bonus/10-7/), the following table is obtained:

Hand	Experimental Probability	Expected Probability	Absolute Error	Relative Error (%)
Royal Flush	0,000020	0,000021	-0,000001	-4,761904762
Straight Flush	0,000108	0,000113	-0,000005	-4,424778761
Four Aces	0,000200	0,000199	0,000001	0,502512563
Four 2-4	0,000513	0,000524	-0,000011	-2,099236641
Four 5-K	0,001595	0,001608	-0,000013	-0,808457711
Full House	0,011220	0,011190	0,000030	0,268096515
Flush	0,014847	0,014953	-0,000106	-0,708887849
Straight	0,014390	0,015019	-0,000629	-4,188028497
Three of a Kind	0,072534	0,072199	0,000335	0,463995346
Two Pair	0,125223	0,124658	0,000565	0,453240065
Jacks or Better	0,191079	0,192379	-0,001300	-0,675749432
Other	0,567678	0,567136	0,000542	0,095567906

Table 1 - Testing the probability of each hand.

As shown in Table 1, it is possible to verify that the results obtained and the expected results are very much alike. To test the Gain/Return Rate of the proposed solution, the program was tested with different initial credit and number of deals, as shown in Table 2.

		Number of deals		
		100k	500k	1M
Initial credit	100k	-1,233	-16,909	-15,680
	500k	0,342	1,693	0,073
	1M	-0,218	-1,499	-3,037

Table 2 - Testing the variation of the percentage of gain with different initial arguments. The values correspond to a percentage that describes the relation between the initial and final balances.

The values obtained in Table 2 represent the Gain, which is similar to the Return Rate with a difference of 100%. In the case of the “10/7” Double Bonus, it is expected to be 0,2% (which means that the Return Rate is expected to be 100,2%). In the best case scenario, it was obtained a percentage of gain of 0,07% with an initial credit of 500k and with 1 Million deals.

The difference between the experimental and the expected results may be due to the use of a different approach than the one used to achieve the perfect strategy (the *getAdvice* method). The one used in the game of the Wizard of Odds website is more elaborated and, for that reason, eventually more accurate.

Taking that into consideration, the results obtained (both on Tables 1 and 2) are very close to the ones that were expected.