Design & Development

The purpose of this document is to have a written account of the design and development decisions made during the development of this application. This will detail the consideration of various data structures and any algorithms developed with clear justification of my decisions and rejection of alternatives.

1.1 - Use, Justification and Rejection of Data-Structures

To understand which data-structure is most appropriate to use, a detailed overview of what purpose(s) the aspect of the system in question will serve is required. Following this, I will eliminate and consider appropriate data structures and its suitability in terms of its strengths and weaknesses and finally provide a conclusion on which data-structure is most suitable and any possible alternatives.

1.1.1 - The Deck

Firstly, the deck will simulate a standard deck of playing cards, these cards will then need to be shuffled before being able to be dealt to the player's hand. Therefore, the deck will be of a fixed size consisting of 52 cards and will be initially populated with each rank and suit combination. After the data structure in question has been populated, it will then need to be shuffled without possibility of a rigged deck. Finally, when called, a card must be able to be dealt from the deck.

The first data-structures I will consider is the suitability of using a Stack, Queue, or other related data-structures to carry out these actions which are all considered Linked Lists. At first, due to the nature of a deck of cards I would assume that, at face value, these data structures would be more than suitable for the task at hand; the initial reasoning behind this is that both data-structures can only access the front or top entry. Similarly, to how a conventional deck works, I would only need access to the front or top entry to deal a card to the hand. However, for the same reason this data-structure succeeds, it also fails; this is because the cards will need to be shuffled randomly. The only way of carrying out this process with these data-structures is to create two temporary stacks or queues and alternating where the entries will *"pop"* or be *"dequeued"* too before adding them back to the main data-structure, this process could also be repeated a random number of times. I have elected not to pursue this data-structure mainly for the fact that this would create a rigged deck as it will never be truly randomized and would be incredibly inefficient to carry out.

Following this, I will consider the prospects of using the Bag as the data-structure for this aspect of the system. The Bag ADT meets the criteria in the following aspect, it contains a finite number of objects, which, in this case would be 52 cards. However, the Bag can only be used to removed specified object; At first, I believed this could be maneuvered around by getting the value for a random card and removing it from the deck, however, if an identical card is called again, it won't be able to return any value. Furthermore, the Bag has behaviors that are rendered redundant for what I need it to carry out, these behaviors being, removing an occurrence of an object in the bag, getting the frequency of an object in the bag and test whether the bag contains a particular object. Some of these issues can be resolved in a *"Linked Implementation of the Bag"*, where I can remove an unspecified entry from the bag, however, for the same reasons as the Stack and Queue, it would produce a rigged deck if it tried to be shuffled in this implementation. For the reasons described above, I have decided not to pursue this data-structure, mainly due to the issues that come with trying to remove an object.

Finally, I will discuss the feasibility of utilizing the List data-structure. In terms of the purpose this needs to serve in relation to the aspect of the system, this data-structure is far superior to the ones forementioned. Most notably, is that the list can be used to add, replace, and remove entries at precise positions, which means that the deck can be thoroughly shuffled without compromise. However, the List isn't of a fixed size, this isn't an issue regarding its functionality as it is still fully capable of carrying out what is required of it, but instead, this is an issue when it comes to efficiency.

In conclusion, I have decided not to use any of the previously mentioned data-structures, instead I see the benefits of simply using an array. The reason behind this decision is that it is of a fixed-size and once populated and shuffled, entries don't need to be removed. Instead of removing an element in the array to deal a card, the

value for the card object at the end of the array can be returned and increment the number of cards down by one for each time. Since the functions of the array and list will be identical in this case, my decision to use an array is based on efficiency as when it comes to the list its size can fluctuate when it isn't required.

1.1.2 - The Hand

The hand will store a collection of cards dealt from a shuffled deck, the number of cards dealt is within the range of 5-10 depending on user input and will then need to be sorted in ascending order before it can be displayed. Furthermore, the user can also choose to nominate a card to discard and replace it with another card dealt from the deck. Therefore, the hand will be of fixed size determined by the user input consisting of cards dealt from a shuffled deck. It will need to be sorted in ascending order and the user must be able to choose a card they want to nominate, so the data structure in question must be able to replace elements at a specified position.

From this I can immediately eliminate the possibility of either using a Stack, Queue or any other similar data-structures such as PQueue or Deque; the reasoning behind this decision is that they restrict access to its entries. The only way to look at an entry that isn't either at the front of the queue or top of the stack is to repeatedly remove items until the desired item reaches the front/ top. As the hand both needs to be sorted and able to replace entries at specified positions, it is apparent that these data-structures are unsuitable.

I can also eliminate the Bag data-structure for the same reasons mentioned throughout this section regarding its unsuitability. The bag is of a fixed size which at least meets one of the requirements, however, the bag can only remove or find entries if the object is already known meaning that it can't be sorted in ascending order as the position of the objects in the data-structure is required and you also can't select which card you want to exchange without typing out the full name of the card exactly to remove it from the bag. For these reasons I will not be using the Bag ADT.

From what I have gathered so far, these leaves us with either using an array or the List ADT. For now, I will discuss the usability of the List data-structure. The List allows objects to be added, removed, and replaced at precise positions making it eligible to be sorted in ascending order. It also has a behavior that allows an object to be replaced at a position with the value of another object which makes it ideal for what the hand is needed to do. However, once the hand has been created with its size determined by the user, it will no longer need to be resized from this point on. The List would be a suitable alternative and would function as required, however, if implemented its features wouldn't be fully utilized and seems unnecessary.

In conclusion, similarly to my choice regarding the deck, I have decided to use an array for the hand. My reasoning for this action is that once created the hand does not need to be resized. Since the array uses an index, it can be sorted and objects at specified positions can be replaced by a card dealt from the deck. A List could theoretically be used in place of an array as they both are almost identical with the list having some bonuses but since the hand does not need resized once created it makes the List seems like the most expensive performance wise when it comes to this selection. For these reasons above I will be using an array to carry out the hand's functions.

1.1.3 - Replay

The replay will display each card in the hand for every hand played in a single player game, it will also display the card in which the user decided to exchange if applicable. From this, I can tell that the data-structure in question will need to be of a dynamic size as the number of cards in the data-structure will depend on the hand size for the game and the number of hands played. Since the data-structure will be purely used to display each card of the hand and the card the user exchanged I will separate these in to two data-structures of the same type as they are dependent of each other (e.g., the final hand displayed will not have a card the user decided to exchange printed alongside). Finally, the data-structure does not need to be able to have access to all its entries as it will be purely used to display its contents so it will linearly progress through each object.

Firstly, the Bag ADT would be unsuitable for this process. Similarly, to reasons I discussed previously throughout this section, the Bag is of a fixed-size and can only remove or search for elements if it is already known. If I tried to use this data-structure in this context I would need to use another data-structure to be able to tell the Bag what object in the hand is meant to come next so that it can be displayed, and I would need to constantly create

temporary bags with a larger size to add the original bags contents to and to add the new hands object too. For these reasons I have decided not to use the Bag ADT.

Secondly, I will discuss the List ADT. Now the List is functionally capable of carrying out the replay and meets its requirements being of a dynamic size so I can add as many cards to it as required. Furthermore, in theory you could get the length of the list and display the removed element at the end and repeat this process until it is empty. However, even though the List is functionally capable there are far more efficient data-structures to choose from, mainly being because the list uses an index which we simply do not need for this process as we only need the front or top elements. Using the list would mean constantly checking if it is empty, getting the length and removing the last element to display which is highly inefficient when compared to a Linked List. The List would be a feasible alternative, but other data-structures are available to choose from which are more suitable for this aspect and for this reason I will not be using the List.

Before discussing Linked Lists, I will discuss why an array isn't suitable for the task at hand. This is for two main reasons; one is that it is of a fixed size which means I would need to use a temporary array larger than the original to store new elements to copy it back. The second reason is that, like a List, it utilizes an index which is simply not required and if used would need to follow a similar procedure to the list to print out the last object.

The requirements indicate that a Linked List data-structure would be most suitable since they do not have a restricted size and only access the front or top entry which in this scenario is all it needs to do, so I will need to decide from here which Linked List would be most suitable. Firstly, I don't believe the Stack would be the best way of carrying out this process, this is due to the reason that the behavior of the Stack is known as *"LIFO"* (Last In, First Out). Which essential means that the last card in the last hand I added to the stack will be the first to be displayed which means it will be in reverse order, for this reason the Stack is not a suitable data-structure.

Finally, this leaves us with the Queue and relating data-structures. I firmly believe that the Queue is more than suitable to carry out the replay, the reason for this is that the behavior of the queue is known as *"FIFO"* (First In, First Out). Which means that the first card from the first hand will be displayed first which is exactly what I require it to do. There are other alternatives to the Queue that could be used but are unnecessary such as the Priority Queue which allows elements in the queue to be priorities but for this instance isn't required and the Deque which allows elements from the front and back to be accessed which also isn't required. For the reasons above, I have chosen to use the Queue for the data-structure to carry out the replay. It not only has a dynamic size, but it allows me to add every card object from the hand and display it in the order that it got added from which makes it the most efficient and suitable out of all the linked lists.

1.1.4 - Scoreboard

The scoreboard will be used to store the names and scores for each round after a game has been completed, it will be used to display the best 5 scores, sorted in descending order. First and foremost, the data-structure will need to be able to replace elements/ add elements at a specified positions to accurately sort the entries. In addition to this, the data-structure in question will also need to be of a dynamic size so that it can store all the round scores played.

Like the hand, I can immediately eliminate the prospect of using the Stack, Queue or any other data-structures that preform similarly. Even though these data-structures aren't restricted by a fixed-size which meets one of the criteria previously mentioned, it fails in other aspects. As stated previously, the reasoning behind this is that they restrict access to its entries with the only way of looking at an entry that isn't at the front/ top is to repeatedly remove elements until the desire item is reached. As the scoreboard will need to be sorted in ascending order, the data-structure in question will need unrestricted access to its entries to be able to accurately find where to sort the next entry.

Furthermore, I can disregard the Bag. Mainly because it fails to meet the two of the requirements; Firstly, a bag is of a fixed size, due to the nature of the Scoreboard the size will constantly be changing depending on how many rounds has been played out so far which will decide how many scores are currently held in the scoreboard. Secondly, the Scoreboard will need to be sorted, the Bag is a data-structure that isn't capable of being sorted as you can only test whether the bag contains a particular object or remove a particular object; to be able to sort

the scoreboard in descending order of score I would need to be able to specify the position of the object I need to add, remove, or replace. For these reasons listed above, I will not be utilizing the Bag ADT in this context.

Before I talk about the List ADT, I will briefly mention why an array is also unsuitable. Even though with an array you can specify the objects position making it desirable for sorting, it is of a fixed size. However, this is avoidable by creating a temporary array being bigger than the original to copy each of the original array's objects and add new objects, this is just inefficient and unnecessary as the List ADT can perform all these functions faster and for this reason, I will also not be using an array.

Finally, there is the List ADT, which in-conclusion I find to be the most desirable data-structure to use for this aspect of the system. This is because the objects in the list have a position, objects can be added, remove, or replaced in any position and it does not have a fixed length. This allows the size of the scoreboard to constantly fluctuate and due to objects having a position, also allows these objects to be sorted in descending order. With this data-structure we can also get its current length which allows us to remove any elements after position 5 having already been sorted to ensure that the scoreboard doesn't hold any more scores than it needs too. Furthermore, I will be using a Linked Chain Implementation of the List as normally, the list has an array implementation which can become full or can contain lots of used space and a resizable array requires data to be moved each time the array expands. The main benefit of a List Chain is that it is dynamic in size and does not need to be resized. For these reasons I find the List with a Linked Chain implementation to be the most suitable compared to the previously mentioned data-structures.

1.2 - Algorithms Developed

1.2.1 - Sort Algorithm

	STABLE?	IN-PLACE?	BEST CASE	AVERAGE CASE	WORST CASE
BUBBLE SORT	✓	✓	N-1	¼ N^2	½ N^2
INSERTION SORT	✓	✓	N-1	¼ N^2	½ N^2
SHELL SORT		✓	N / N log N	?	≈ $N^{3/2}$
MERGE SORT	✓		(3N - 1) log N	(3N - 1) log N	(3N - 1) log N
QUICKSORT		✓	N log N	2N log N	½ N^2

From this table, we can decide which sorting algorithm would be best used for this application. As both the "Bubble Sort" and "Insertion Sort" appear to be the most ideal out of this list being both stable and in-place with similar performances we will only be discussing which out of the two is best, they also have identical Big-O complexity cases. The other sorting algorithms may be favourable in certain aspects being less complex in some cases but as they are only stable or in-place and not both, I will be choosing accuracy over performance as it is imperative to the systems functionality that everything is in place accurately.

First, I will discuss what each sorting algorithm does to compare them. The "Bubble Sort" will order a list of values by repetitively comparing neighbouring elements and swapping their positions if necessary. It will scan the list, exchanging adjacent elements if they are not in relative order, scan the list again, bubbling up the second highest value and repeat this process until all elements are in their proper place in the list. Each swap operation moves an item closer to its final correct position, an improved "Bubble Sort" will also keep track of the last swap made and exit when none is required, this is so redundant passes through the main loop are avoided.

Next is the "Insertion Sort". This sorting algorithm will order a list of values by repetitively inserting a particular value into a sorted subset of the list. "Insertion Sort" of a data-structure partitions it into two parts, the sorted part initially containing the first entry and the unsorted part containing the remaining entries. The algorithm removes the first entry from the unsorted part and inserts it into its proper position in the sorted part, it then chooses the proper position by comparing the unsorted entry with the sorted entries beginning at the end of the sorted part and continuing towards its beginning.

Now that we know what both sorting algorithms do, I can now discuss which I find most suitable, that being the "Insertion Sort". This is because this sorting algorithm includes a smaller number of swaps because it inserts the

entry into the sorted partition of the data-structure where it is required instead of constantly swapping the elements until it is in order. Whilst the "Insertion Sort" is more complex than a "Bubble Sort", from external research, in some cases it is almost twice as fast as the "Bubble Sort" making it more efficient than the latter where the operation can be costly.

1.2.2 - Shuffling the Deck

After the deck as been populated by each card and suit combination to have a total of 52 cards, the deck must be randomly shuffled, to do this I have decided to implement a method called shuffle in the Deck class.

This algorithm will use a for loop to move backwards through each card object in the deck, at each position it will pick an object at a random position before the current object and swap it with the current object. This will ensure that the deck is thoroughly shuffled and does not produce a rigged deck to maintain the games integrity (ensures that users won't be able use a pattern to predict what the next card in the deck will be based on their hand).

1.2.3 - Finding the highest Streak in a hand

Once the hand has been populated with cards or a hand has had a card replaced, the highest streak in the hand must be found. To carry this out I have developed three separate methods:

getStreak

This will find the length of the streak starting on the position of the card given. This algorithm is a necessity as it will be needed for the getScore algorithm to find each streak in the hand to determine which of the streaks is the largest.

This algorithm works by first determining if the first object and the next object in the data-structure are consecutive (if this card is equal to the next card minus one), if they are it will continue with a for loop that will compare the current element to the next element for each object in the data-structure until it either reaches the end or the for loop is broken by the objects not being consecutive. It will then return the value of that streak.

getBonus

This will find the total amount of bonuses to award for a given streak. This algorithm is a necessity as it will be needed for the getScore algorithm to find the bonus of a given streak, this will only be preformed if the streak found is greater than the highest streak found or if they are equal to determine which streak has a larger bonus.

This algorithm will use a for loop to progress through each object for the length of the streak in the data-structure, starting from the same position the streak started on. It will check if the current object is the same color or the same suit as the next object, if they are not a Boolean will be set to false for their respective bonuses and the for loop will break if neither are true. After the for loop has completed an if statement will be used to determine if either Boolean is true to indicate what bonuses to apply and return that value.

getScore

This will be used to find the highest streak and its bonus, if the highest streak is equal in length to another streak it will apply the largest bonus of the two streaks. This algorithm in combination with the algorithms mentioned prior will be used to get the score of the current player's hand.

The algorithm works by using a for loop to progress through each object in the data-structure and getting the streak for the current object and then following this, the bonus for that streak. Inside the same for loop there are two if statement one will make the current streak the highest streak if it is greater than the last highest streak found (this includes setting current bonus to highest bonus as the bonus is dependent on the streak) and another that if the current streak and highest streak are equal it will compare the highest bonus and current bonus to determine which is larger and set that value to the highest bonus. Finally, the position of the for loop will be set to that of the end of the current streak to increase its efficiency. This process will repeat until the for loop has completed and return the value of the highest streak and the highest bonus combined.

1.3 - Abstract Data Types Developed

1.3.1 - Card Object

The Card Object is an ADT that is constantly used throughout this application in most aspects such as in the replay, the deck, the hand, etc. It is used to carry out a range of operations but mainly to store the values of a given card. This object needs a *"toString"* operation to be able to print out the contents of the Card Object in a way that is readable, i.e., "2 of Clubs" or "King of Spades"

It will perform nine operations:

- The first two being called *"getRank"* and *"getSuit"* will be used to return the objects rank or suit held in a String array. These are both private operations used solely inside the Card Class.
- The next two being like the first two, these are called *"getRankValue"* and *"getSuitValue"* which will be used to return the objects rank or suit position in the String array applicable. These are both private operations having its use restricted to the Card class.
- The next operator is *"compareTo"* which will compare a Card Object to another and determine if it is less than or equal to, returning a value indicative of this. This is a public operator that will be used in the sorting algorithm to determine where the Card Object needs to be inserted in the data-structure.
- Following this is the *"isStreak"* which will determine if this Card Object rank value is equal to a given Card Objects rank value minus 1 and will return a Boolean value depending on this. This is a private operator used in the *"getStreak"* algorithm detailed above.
- After this is the *"getColour"* operation which, like the name, will return the color of the Card Object using *"getSuit"* to first determine the objects suit and then determine the color of that suit and return it as a String. This is a private operation used later in the *"isColourBonus"* operation.
- Finally, the last two operations are *"isColourBonus"* and *"isSuitBonus"* which will compare this card object to a given card object to determine if it either is of the same color or of the same suit and return true or false depending on this. These are public operators solely used in the *"getBonus"* algorithm mentioned previously.

The Card Object was developed for a multitude of reasons. Mainly to allow the various algorithms to sort the card and to get the scores with those scores depending on both its rank and suit combination compared to another card object given. A final reason for its development was to ensure that, when printed, it would be readable.

1.3.2 - Deck Object

The Deck Object is a simple ADT. This object will utilize a data-structure with a length of 52, once a new Deck Object has been created it will populate the data-structure with each rank and suit combination following this it will then be shuffled. The only operator publicly accessible is used to deal a card from the deck.

It will preform two operations: the first is a private operation to shuffle cards after the data-structure has been populated using the algorithm mentioned above called *"shuffle"*. The second is a public operation which after a Deck Object has been created will allow a card from the top of the deck to be dealt (it will return the card object at the specified element of the data-structure, that being the number of cards), once being dealt the number of cards will be decremented so this process can repeat as expected, this operation is called *"deal"*.

The Deck was developed into an object so that a new deck can be constructed for each round played out, the reasoning for this is that in the requirements it specifically states that a fresh deck will be used for each round played.

1.3.3 - Scoreboard Object

The Scoreboard Object is also a simple ADT. The sole purpose of creating a scoreboard object is to store the names and scores for each player at the end of the round. It contains a *"toString"* operation to properly format a Scoreboard Object when printed so that it looks uniform in the Hi-Score Table.

It can be used to preform two operations: the two being utilized solely for the sorting algorithm, that being the *"getScore"* is a private operation used for the next operation which will return the score for the object and *"compareTo"* which is a public operation that will compare a Scoreboard Object to another and determine if it is less than or equal to, returning a value indicative of this. This operation is used by the *"Insertion Sort"* algorithm described above.

The Scoreboard Object was developed solely for the purpose of formatting the values to be displayed in the Hi-Score Table and to allow the sorting algorithm to sort the object by score as the requirements indicate.