COM498 Algorithms and Data Structures – Design & Development Document
Niall Morrissey (B00787301)

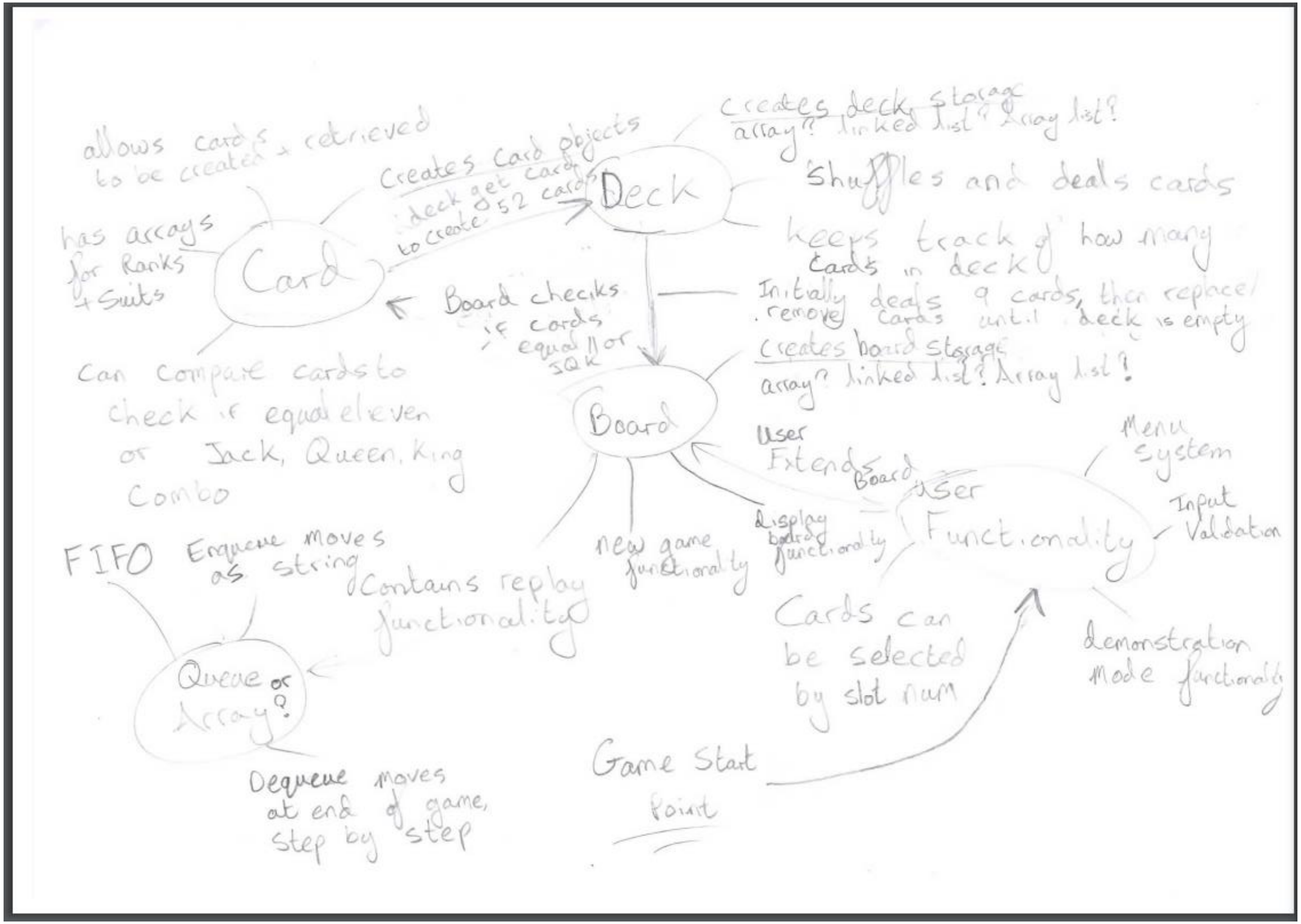
Design and Development

Introduction

I started the design process by preparing a flow chart of the various elements that I would need in the game of Elevens. I split these elements into classes which would represent the main elements of the game, the Card, the Deck, and the Board. I set out the main functionality required by a game of Elevens and how this might be achieved. This approach allowed me to consider each element and assess the most appropriate method for achieving that element. It also gave me a holistic view of the game and how I might structure the development of the code. Through the development phase of the project, I adapted and changed some of the design elements to improve the code. In the sections below, I have explained in greater detail my design choices and outlined why I consider them to be the best approach to implementing the functionality required to play the game.

In the table below, I have set out the main classes that were used in the development of my game and have provided an explanation as for the reason on my choice and how it would achieve the functionality.

Initial Design Flow Chart and Ideas


allows cards to be created + retrieved
Creates card objects
deck get card to create 52 cards
Deck
creates deck storage array? linked list? Array list?
Shuffles and deals cards
keeps track of how many cards in deck
has arrays for Ranks + Suits
Card
Board checks if cards equal 11 or JQK
Initially deals 9 cards, then replace/remove cards until deck is empty
creates board storage array? linked list? Array list?
Can compare cards to check if equal eleven or Jack, Queen, King Combo
Board
User Extends Board
Menu system
User Functionality
Input Validation
Display board functionality
new game functionality
FIFO
Enqueue moves as string
Contains replay functionality
Queue or Array?
Cards can be selected by slot num
demonstration mode functionality
Dequeue moves at end of game, step by step
Game Start Point

Detailed Development & Design of Individual Classes

	Functionality Required	Design Decision taken and reason
AListArray Class	• Ability to add and store 52 Cards. • Ability to quicky get entry from range of cards. • Ability to remove a random specific card entry from range of cards. • Ability to quickly select a specific entry from range of cards and replace it. • Ability to get length of cards stored. • Ability to return all cards stored as string. • Ability to clear all cards stored.	• I decided to go with a ListArray Abstract Data Type to store my card objects as I believe it is the most appropriate storage type available. It provides the ability for all the required functionalities set out to the left with its included getEntry, remove, replace, toString, getLength and clear methods. It also can dynamically grow and shrink to accommodate the number of elements within. • A Linked List was the best alternative as it contains many similar methods and can also dynamically resize but as ListArray's can quickly index each card entry stored it makes it a much more efficient solution as most of the program revolves around quick access to specific elements within the Deck and Board arrayLists to be manipulated throughout the game. • ListArrays have notation a O(1) for random access of stored elements, whereas a Linked List is O(n) so is much slower in this regard. Also, as ListArrays store array elements in contiguous memory it can be much faster to search for an element, whereas linked list nodes could potentially be scattered over many different pages of memory resulting in slower retrievals of entries. • An array bag could be used but is not dynamic like a ListArray as has a fixed capacity and doesn't have the functionality to get a specific card entry from an index position or to replace a card entry at a specific position. • An object array could be used but lacks the functionality methods that come with a ListArray and can only be a fixed size whereas ListArrays are dynamic and can scale with size of number of elements. • Queue and Stack Abstract Data Types are also alternatives; however, both rely on a linked node approach so the same issues with a Linked List would apply here. Also, both are designed for First in First Out and Last in First Out approaches and do not have the required functionality set out to the left.
Queue Class	• Record moves and store cards as they are played • Allow replay of the game for the player to see moves • Player to review moves starting with the most recent move and moving backwards through the game	• I decided to go with a queue as this allows a First in First Out approach which is exactly what is required for replaying the moves of a previous game. It also contains all the necessary functionality required as set out to the left with its included enqueue, dequeue, getFront, isEmpty and clear methods. • The main alternative was to use an array to store each of the moves and iterate over them at the end, however a queue is more suited to enforcing the sequential rules of First in First Out. Also, the size of the queue can dynamically grow and shrink to the number of moves made for each game, whereas an array is fixed size which wouldn't work unless sized to the maximum amount of moves which wastes memory allocation. • Another alternative was a stack, however as it is Last in first Out it conceptually doesn't fit the replay requirements set out to the left and wouldn't be appropriate.
Card Class	• Create an individual card object with a Rank and Suit for all 52 variations of card.	• Store both the possible card RANKS and SUITS as private final String arrays as they are fixed length and will never need to be changed.

	• Ability to print the properties of the card object as a string. • Check whether two card ranks when added together equal eleven. • Check whether three card ranks equal a Jack, Queen, King combination.	• Create A Card class constructor that takes two passed in integer variables to be used to index the RANKS and SUITS array and stores them as instance variables for each Card object created. This allows each individual card object to have unique ranks and suits. • Referenced card objects can be retrieved by public method toString which uses two private methods getRank and getSuit to access the String arrays and return the rank and suit. • Create equalEleven and equalJQK methods public so can be referenced by other classes and cards can be checked.
Deck Class	• Create a deck to store the 52 cards within a standard deck of cards. • Ability to deal cards out in a shuffled order to the board class as required. • Remaining cards to be counted and reported to player during game.	• Create a deck using the Abstract Data Type ListArray to store the 52 card objects. I chose a ListArray to store the cards as it grows and shrinks dynamically with the size of the deck. This is very useful as when the deck of cards shrinks throughout the playing of the game, the size of the list array shrinks to match the length lowering the overall memory usage of the application. A ListArray also contains key methods such as getLength to retrieve the overall length of the deck for remaining cards functionality and getEntry to retrieve an indexed element within the deck which can be used by the deal cards functionality. • Create Deck class constructor that utilises the Card class constructor. Using a for loop with the range of 52 to create 52 unique card objects and add each to the created DECK ListArray. This is a quick and effective way of creating 52 card objects for a deck of cards that is run as the deck class is being initialised. • Create a public deal method that utilises the java.util.Random library to generate a random integer between the range of 1 and the length of deck + 1. Then return a card found in the ListArray Deck at that random position. This ensures that each card dealt to the board is completely random and is shuffled for the game of elevens. • Create two public methods getDeckLength and getDeckEntry that allow other classes to retrieve the overall length of the deck or specific entry within deck. This ensures the Deck ListArray can be kept private from other classes where access is not necessary to heighten security.
Board Class	• Create the main board to store the 9 shuffled cards for playing the game. • Display the current state of the board to the user highlighting remaining cards in deck. • Ability to replace cards on the board if any are remaining in the deck or just remove them if there are not.	• Create a board using the Abstract Data Type ListArray to store the 9 card objects. I chose a ListArray to store the cards as it grows and shrinks dynamically with the size of the board. This is useful as towards the end of the game when no cards are available in the deck and cards need to be removed with no replacement, the size of the Board ListArray can shrink to accommodate the new length. This is both more efficient as it saves memory and allows for extra functionality as the length will reflect the number of cards currently in the board which is very useful for user input validation and restricting possible input. Another key reason for using a ListArray is it allows quick access to specific elements on the Board through indexing. It also allows a card entry to be replaced in a specific position within the middle of the ListArray which is important for replace card functionality. For creation of a new game, it provides functionality to clear the board which is useful.

	• Check for any possible remaining moves and return findings. • Replay the game step by step displaying each step to the user. • Create a new game	• Create a Board class constructor that creates a new instance of Deck and deals the first 9 shuffled cards that are adding each to the board. This is a quick and effective way of dealing the initial 9 cards for the game of Elevens. • Create a displayBoard method that loops through for the length of the board, displaying each of the cards in the board and its corresponding slot number. This method can be repeatedly called throughout the game to check the current state of the board reducing unnecessary repeated code. • Create two methods for replacing & removing cards both called replaceCards utilising method overloading as one method takes two parameters (cards that equal eleven) and the other three (cards that equal JQK combination). The replace card method should take the card slot numbers selected to be replaced as parameters. Then will create a temp int array to sort the passed in slot numbers from highest to lowest to prevent the code crashing as if lowest slot entry removed first then index may be out of range for the higher slots when attempted to be removed after. Then use a loop with the range of the number of parameters (2 or 3), replacing the card for that specific entry in the board as passed in with a new card dealt from the deck or else if no cards remaining just removing the card from the slot. The replace functionality of the ListArray ADT is crucial for this step and cannot be found from most other ADT other than a linked list which is not as efficient as can't index and need to be searched through. • Both replaceCards methods will initially enqueue the current state of the board toString to the replay queue. Then will enqueue each card entry removed to the replay queue so can be dequeued at end of game. This is crucial for the replace card functionality of the program where an integer entry of an existing card will be passed in and replaced using the deal method or else just removed if none remaining. • CheckPossibleMoves method that is used to search through and return the hint integer ListArray of the slot entries for possible moves. ListArray used as a normal integer array would have to be a set length whereas ListArray can be dynamic in case of 2 or 3 cards to be returned. Two loops used, one to check for possible moves that equal eleven and the other to check for possible moves that equal a JQK combination. For checking cards that equal eleven use two nested for loops checking all possible combinations. If combination found, add moves to hint and return. For checking cards that equal a JQK combination, use single loop checking the rank value of each card. If a correct rank value appears each for a Jack, a Queen and King, add moves and return as hint. Else return blank integer array. • For replay functionality, create replaySteps method which allows user to step through replay queue while not empty. Queue ADT is perfect for the replay method as it is designed to be first in first out, which matches the need of the replay function exactly, unlike a stack which is the complete opposite and is last in first out. To step through each move made, require user to press 'enter'. Dequeue the board and any following removals and then require user to press enter again. Repeat until all steps have been dequeued.

		If no more moves remain, print out 'no more moves left to replay'. Dequeuing any following removals will make it much nicer for the user to operate, as then will only have to press 'enter' for each step of the game instead of each individual move. • Make the board ListArray and all core methods within Board class protected so can only be accessed through inheritance by the ElevensGame class. This provides encapsulation and protects the board and methods from being called by unnecessary outside entities and only by classes that require the functionality.
ElevensGame Class	• Rules of the game to be included in this class including a menu. • Main game functionality allowing users to select combinations of cards to be checked and replaced. • Ability to allow user to select card slots within range of slots currently on board. • Check for a hint if prompted by the user. • Ability to run game through demonstration mode.	• Create the main public game method that greets the user to Elevens, explains the rules and asks whether they want to play a game or run demonstration mode. If play a game selected, run through a loop while the board length is greater than 0. Within the loop, first display the board of card objects and check whether checkPossibleMoves method is empty which results in a stalemate and a menu being called. Otherwise, continue allowing the user to select first card by calling selectCard method. If first card in selected slot has rank less than or equal 10, require one more card that should equal eleven be selected. Once both card slots selected, use the equalEleven method in Card class to check whether the two cards equal eleven. If they do equal eleven, call replaceCard method in Board class to replace the selected slots in the board with two new cards. Else if two cards don't equal eleven, prompt this to user. If the first card in selected slot has rank value greater than or equal to 11, require two other cards be selected to make a Jack, Queen, King combination. If cards equal JQK, pass through slot numbers to replaceCard method. Else if they don't equal JQK, prompt this to user. Repeat through the while loop eliminating cards until it either ends in stalemate or the while loop condition is met, and the game is won. If game is won, prompt user with menu. This approach is efficient as most of the game is contained into a single while loop that calls other methods where necessary. As the selectCard method has been included within the if statement checking whether to ask for 2 or three more cards, it should make usability of the game much better for the user as they won't have to specify how many cards to remove prior to removing. • The menu functionality should be prompted at the end of a game either by winning or stalemate. It should prompt the user if they wish to replay their moves by calling the replaySteps method within the board class, play again by calling the game method or to exit the program. • Create selectCard method functionality that enables the user to return a selected slot but included integer only and range validation to ensure only entries in current board can be returned. • Demonstration functionality should be like main game method but passing in the parameters retrieved from the checkPossibleMoves method to replaceCard method instead of user input. It should then cycle through all possible moves until the games has either ended in a win or a stalemate.