
# Design and Development - Elevens

**Author**

Mr. Adam Torok - B00798824

Jordanstown, November 26, 2021

## Required Functionality

Please see the `Diagram.drawio.png` in the project root folder to have an overview of the project. Simply it is too big to include it here.

### Level 1

> "The facilities to create a new game are present and the game can be set up in its initial state, with 9 cards dealt face-up from a shuffled deck."

This functionality is completed successfully. The `main()` function calls the `selectMode()` function which controls the menu selection. The main part of the game takes place in the `play()` function. This function takes a boolean parameter to initiate user mode (false) and demonstration mode (true). Every data are handled in ADTs. The implementation based on the linked list data type. The base element is a `Node`. The `Card` class extends the `Node` class. This provides the `Card` objects to store a value and reference to another cards.

The next two classes are the `Deck` and `Board`. Both extends the linked list data type. The related methods are stored separately in this class, which makes the class more portable. Linked list was chosen because it seemed more difficult to implement it. However if we can implement the connection between the objects, the only main difference is the way we access them.

The `Deck` object creates the set of cards in the constructor and then shuffles them. After shuffling, a lock variable will prevent to add more cards to an already shuffled deck (or even shuffle them it again). We can access the size of the deck, we can print it's content, get the first and the last card. With the overloaded constructor we can create empty deck for testing.

The `Board` class is more complicated. In this class we keep track the KJQ cards on the board. This is a cheap way of checking the board because we will know the cards anyway (cards from A to T are not tracked this way). This is important because as soon as we see these card, we have to pick them (to maximize the win probability). There is no card-re creation, duplication, replication. All card objects moves from the deck to the board straight, keeping their object reference. Important security layer, since we don't have deal with duplicates (apart from multiple calls). The class is capable to return with a given card value (board index: 0-9). The `validMove` Stack stores all the possible valid combinations which can be accessed from the outside. The board is sorted using a bubble sort after every board operation (card remove, card add).

### Level 2

> "The basic game mechanism is in place. Users can select legal combinations of cards to be removed and they are replaced by new cards dealt from the deck (for as long as the deck has cards available)."

The users can enter the board index (0-9) to select their cards in one input (no spaces). Alternatively a hint system can be used (h key), which shows a possible combination (KJQ prioritized) and automatically remove them. The application handles the order in ascending and descending order e.g. 789 is exactly same as 987. Behind the scenes, the choice is ordered into a descending order to avoid any decreasing index issues. For example when we input 789, the 7th elements gets removed, and then the index of the following elements will drop by 1 (that is the main reason to sorting the board). If we start with the bigger index, we never run into this issue. After every successful input, we pull more cards from the deck (of there is any) and automatically build a `validMove` Stack. If the stack is empty, no more moves left, game over. In addition the program handles the zero padded inputs (350), the duplicated input (55). If there is no valid moves left, the application automatically announce the end of the game. A Deck counter and the removed card are displayed. On valid move, we let the garbage collector do it's thing, but we store a string representation of the two selected card (and the board state as well in the replay stack).

### Level 3

> " playable game is available. The application is able to prompt the player when the game is won (all cards removed) or lost (stalemate is reached as no cards can be removed)."

If no valid moves left and the board is till not empty (or everything empty) the program ends the game.

### Level 4

> "The game is able to provide a hint to the player (on request) suggesting a valid move or confirming that no move is possible."

This feature is based on the `validMove` stack, which gets updated after every board change. The user is able to use these hints (key h). A valid combination is popped from the stack and automatically applied. When there is no more moves available, the application ends the game (user friendly).

## Level 5

> "On completion of a game (whether the player has won or lost) the application is able to replay the game move by move, with the user prompting each replayed move by a keypress.

User can select to replay their moves. After every move, the selected cards and the boards' string representation are stored in the stack. Upon requests, these are popped off, reversed and printed. A queue representation would have been the right data type for this.

## Level 6

> "The application is able to play a complete game in "demonstration mode", where the only input from the player is to prompt the application to play its next move. The computer should provide a commentary on each move made such as "3 of Spades and 8 of Diamonds removed", "King of Clubs, Queen of Diamonds and Jack of Hearts removed" and so on."

`validMove` Stack. Every point of the game we know the legal combination without any request. So the only thing we need to do here is pop one off and apply it. if the `validMove` Stack is empty, the game is over.

The main difference in the user and the demonstration mode is that the demonstration mode always use the `validMove stack` (As an input). The users can mimic this feature (as long they spam the h key), but their input will be validated it he `Board` class, and compared with the `ValidMoves` Stack for double safety.