**REPORT: ASSIGNMENT 2 NOT FREECELL**

**TABLE OF CONTENTS**

|  |  |  |  |
| --- | --- | --- | --- |
| Serial | Name | | Page |
| 1. | Introduction | | 2 |
| 2. | Background | | 3 |
| 3. | Not FreeCell | Flow Diagram | 4 |
| Use-case Diagram | 5 |
| Technical Specification | 6 |
| 4. | Conclusion – merits, limitations, future direction | | 10 |
| 5. | References | | 11 |

1. **INTRODUCTION**

The software is an attempt at designing Not FreeCell. Not FreeCell is a game inspired by the original game FreeCell which was created by Microsoft for release with the Windows Operating System. This report briefly explains the rules of the original game FreeCell and contrast those with the rules implemented in this version. The report then illustrates the flow diagram and the use-case diagram for current Not FreeCell version. Technical Specifications are provided which outline the design choices implemented in the current version – justifying the use of variables and methods used in the program and providing details to the complicated sections of the code. The report is concluded with a summarization of the merits and future directions for developing the current version of the software.

Please refer to the video provided with the source files to view the demonstration of the 2-suit game in play.

The game follows the following rules:

. Only one card may be moved at a time.

. The four foundations must be built starting from the Ace of the appropriate suit, followed by the 2, then

the 3 etc. until the King is placed.

. If a card is placed on a cascade, it must be placed of a card of the opposite colour, and of a suit that is

one higher than itself. E.g. a red 2 can be placed on a black 3, but not a black 4.

. Any card may be placed in an empty cascade.

. Cards from the foundations may be placed back onto a cascade, or an empty cascade slot.

. Only one card at a time can occupy a cell slot.

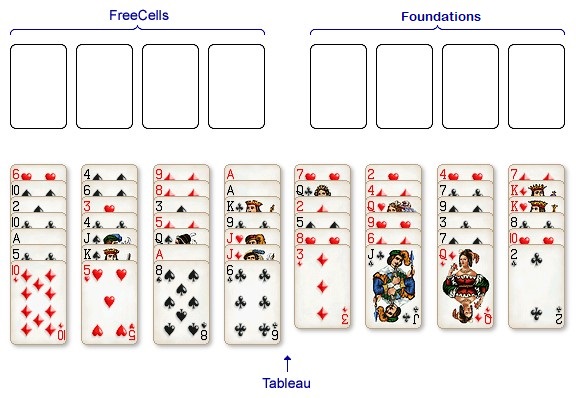
Victory is achieved when all four foundations are \_lled with their respective suits from Ace to King.

1. **Background and Rules for FreeCell**

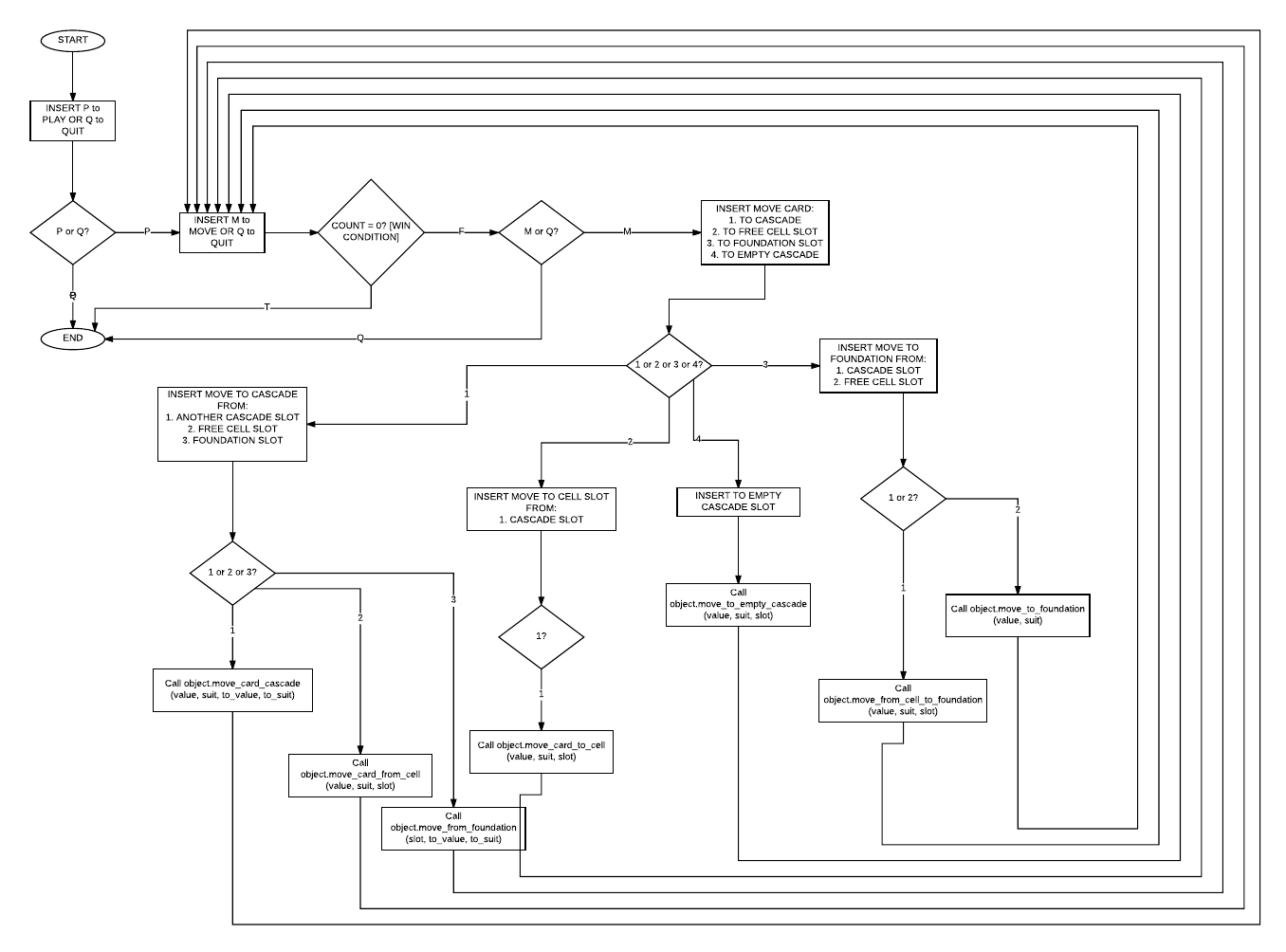
One of the oldest ancestors of Window’s FreeCell is the mathematical game called ‘Eight Off’ created by C. L. Baker as described in Scientific American June 1968 by Martin Gardner. The first computer version of the game is believed to have been created by Paul Alfille in 1978 for the PLATO system. Microsoft developer Jim Horne implemented a version with colour graphics for Windows.

Not FreeCell is a solitaire-based card game which is played with a 52-card standard deck. Not FreeCell is an ‘open’ solitaire game meaning that all the cards are dealt face up at the start of the game and this lets you analyse the outcome of moves before you make them. The construction and layout of the game can be summarized as follows:

1. Cascade Tableau: In this version of Not FreeCell, 8 cascade columns are present. Other alternative versions may exist. Initially all the 52 cards are placed in the cascade with 4 columns containing 7 cards and 4 columns containing 6 cards.
2. Free Cells: Free Cells are empty slots which can contain one single card at a time. Card from the free cell can be moved to the foundation cell or back to the cascade.
3. Foundation Cells: Foundation cells are stacks of each suit which can hold cards only of the suit it is associated with. Each foundation cell represents a stack for a suit. Cards may be moved to the foundation cells from the cascade tableau or the free cells. The game results in a victory when all the cards from cascade tableau and the free cells are stacks in the foundation cells.



* 1. **Flow Diagram for Not FreeCell**



**3.2 Not FreeCell Use Case Diagram**



**3.3 Technical Specifications**

A. Class Card:

-Variables Used:

* self.card\_face – represents the value of the card.
* self.card\_suit – represents the suit of the card.
* self.card\_colour – represents the colour of the card as a digit (0 or 1).
* self.card\_suit\_nbr – represents the number of the suit used for data manipulation in Freecell Class.

-Methods Implemented:

* \_\_init\_\_(self,value,suit) – to initialize card object with value, suit, suit colour and suit number.
* \_\_str\_\_(self) – string representation of the card object.
* \_\_eq\_\_(self, other) – equality condition for comparing two card objects.
* get\_face(self) – Accessor method. Returns the face value of the card object.
* get\_suit(self) – Accessor method. Returns the suit value of the card object.
* get\_colour(self) – Accessor method. Returns the suit colour value of the card object.
* get\_card\_suit\_nbr(self) – Accessor method. Returns the suit number value of the card object.
* set\_face(self,value) – Mutator method. Sets the face value of the card object with the provided value.
* set\_suit(self, suit) – Mutator method. Sets the suit value of the card object with the provided suit value.
* set\_colour(self, colour) – Mutator method. Sets the colour value of the card object with the provided colour value.
* set\_card\_suit\_nbr(self, number) – Mutator method. Sets the suit number value of the card object with the provided number value.

B. Class Deck: a collection of card objects.

-Variables Used:

* self.value\_start – represents the start value of the deck object.
* self.value\_end – represents the end value of the deck object.
* self.nbr\_suits – represents the suits value of the deck object.
* self.list\_of\_cards – represents the list of card objects associated with the deck object.
* self.stack\_of\_rev\_value – represents a list of stacks corresponding to each suit, containing the values of card objects in reverse order.
* self.list\_of\_value – represents the list of values of card object used to create the deck object.
* self.list\_of\_suit – represents the list of suit values of card object used to create the deck object.

-Methods Implemented

* \_\_init\_\_(self, value\_start, value\_end, number\_of\_suits) – to initialize the deck object variables with the user input provided.
* create\_deck(self) – creates a deck object of card objects based on the object variables. It is made possible to create deck with more than 4 suits with suits repeating in an incrementing order. separate for loops are implemented for number of suits equal to or less than 4 and for number of suits greater than 4.
* shuffle(self) – to shuffle the deck of card objects using 3 seed values randomly generated using randint method of random library. 3 seeds are used because with one or two seeds the result was not satisfactory.
* add(self, value, suit) – to add a new card object to the deck object with value as face value and suit as suit value.
* draw(self) – to draw a card object from the deck object and reduce the number of card objects in the deck object by 1. Implemented using list.pop() method.
* get\_value\_start(self) – Accessor method. Returns the start value for the deck object.
* get\_value\_end(self) – Accessor method. Returns the end value for the deck object.
* get\_nbr\_suits(self) – Accessor method. Returns the numeric value of suits present in the deck object.
* get\_list\_of\_value(self) – Accessor method. Returns the list of values available for card objects present in the deck object.
* get\_list\_of\_suit(self) – Accessor method. Returns the list of suit values available for card objects present in the deck object.
* get\_list\_of\_cards(self) – Accessor method. Returns the list of card objects present in the deck object.
* get\_stack\_of\_rev\_value(self) – Accessor method. Returns a list of top value present in each stack of reverse values.
* set\_value\_start(self, value) – Mutator method. Sets the start value of the deck object with the value provided.
* set\_value\_end(self, value) – Mutator method. Sets the end value of the deck object with the end value provided.
* set\_nbr\_suits(self, suits) – Mutator method. Sets the number of suits value of the deck object with the suits value provided.
* set\_list\_of\_value(self, value, position) – Mutator method. Sets the value in list\_of\_value variable according to the corresponding position provided.
* set\_list\_of\_suit(self, value, position) – Mutator method. Sets the suit value in list\_of\_suit variable according to the corresponding position provided.
* set\_list\_of\_cards(self, value, suit, position) – Mutator method. Sets the card value and suit variable of the card object present in the list\_of\_cards list variable based on corresponding position provided.
* \_\_eq\_\_(self, other) – equality condition to compare two deck. Assumption: the decks are not shuffled in any way.
* \_\_str\_\_(self) – returns the string representation of the deck object.

C. Class Freecell: Makes use of Card objects and Deck object to implement rules of FreeCell and create a game which is playable.

-Variables used:

* self.list\_of\_cell\_slots – represents a list of free cell slots. Four in number by default.
* self.list\_of\_suit\_stacks – represents a list of foundation cells implemented as a suit stack for each suit in play. Four in number by default.
* self.deck\_in\_play – represents a Deck object used to play freecell.
* self.value\_stack – represents a list of stacks representing value in ascending order used in conjunction with list of reverse value implemented in descending order. Top values in each of the corresponding suit’s value stack and reverse stack are compared for data manipulation purposes. (implementing foundation cells)
* self.op1, self.op2, self.op3, self.op4 – These variables are used to find the self.size of the cascade tableau considering worst case scenario in place.
* self.empty – represents an empty cascade slot.
* self.cascade – represents the cascade tableau used to play freecell.

Methods implemented:

* \_\_init\_\_(self) – Initializes the freecell game by creating a deck object containing card objects and creating an empty cascade tableau to be used in the game and initializes the free cell slots list & the foundation slot stack list.
* new\_game(self) – creates a new game by placing the cards from deck object onto the cascade tableau.
* linear\_search(self, the\_list, target\_item) – performs a linear search in a list for a given target item to be searched.
* move\_card\_cascade(self, face, suit, to\_face, to\_suit) – move given card from one cascade slot to another cascade slot of opposite colour and value one higher than the card to be moved. Limit – if a cascade column is empty, this method is not used and move\_to\_empty\_cascade() method is implemented. Four for loops are used – two for card\_to\_move and two from card\_dest. Upon satisfying the condition, the card can be moved.
* move\_card\_to\_cell(self, face, suit, cell\_slot) – move given card constructed and compared from cascade to the free cell slot based on the number of the slot provided by the user. The position of the given card in cascade is replaced by ‘ ‘ to represent empty slot.
* move\_card\_from\_cell(self, cell\_slot, to\_face, to\_suit) – move given card from free cell slot to the cascade slot below a card of opposite colour and value one higher than the card to be moved. Limit – if a cascade column is empty, this method is not used and move\_to\_empty\_cascade() method is implemented.
* move\_to\_foundation(self, face, suit) – move given card from the cascade tableau to the corresponding foundation slot.
* move\_from\_foundation(self, suit, to\_face, to\_suit) – move the card from foundation slot to in front of the given card.
* move\_from\_cell\_to\_foundation(self, cell\_slot) – move card from the given cell slot to the corresponding foundation slot.
* Move\_to\_empty\_cascade(self, value, suit, cascade\_slot = 0) – move the give card from cascade/free cell slot/foundation to the empty cascade slot provided.
* \_\_str\_\_(self) – returns the string representation of the freecell object consisting of free cell slots, foundation slots and the cascade tableau.

D. def main():

-Variables used: various iteration variables used.

-Methods used:

* os.system(‘cls’ if os.name ==’nt’ else ‘clear’) – to clear the screen of the python script after every move is made and the entire string representation is printed post running this command.
* Isinstance() – used to determine the instance type of the object in consideration for error avoidance.

1. **Conclusion**

The software is a playable version of FreeCell game. The user-interface accepts the input from the user and place it in appropriate functions to perform desired actions. The Card class can be used to represent any card object – not limiting to the traditional deck of card with values [A to K] and suits [C,D,H,S]. The Deck class can be used to create a deck of any number of suits. However, the card colour and card suit will need to be individually varied as per requirement. The freecell class performs all the functions desired in the freecell game. The only movement limits would be to move a card from one free cell slot to another free cell slot and to move a card from foundation to free cell slot. These two movement limitations do not affect the flow of the game and were deemed redundant and thus skipped from the code. They can be implemented easily with a few lines of code if needed in the future.

The software performs error detection and ensures that the game does not crash in case of an invalid input provided by the user and returns to the main menu without obstructing the user.

1. **References**

* <http://www.solitairecity.com/FreeCell.shtml>
* <https://en.wikipedia.org/wiki/FreeCell>