**Solitaire Game**



**Project Supervisor:**

Mr. Nazeef Ul Haq

**Submitted By:**

Waleed Sarfaraz 2023-CS-102

**Course:**

CSC-200 Data Structures and Algorithms

Department of Computer Science

University of Engineering and Technology, Lahore Pakistan

**Contents**

[Introduction: 4](#_Toc182219512)

[Objective: 4](#_Toc182219513)

[Target Audience: 4](#_Toc182219514)

[Scope: 4](#_Toc182219515)

[Project Overview: 4](#_Toc182219516)

[1. Game Rules: 4](#_Toc182219517)

[2. User Interface: 4](#_Toc182219518)

[Git lab Link: 4](#_Toc182219519)

[Design and Implementation: 4](#_Toc182219520)

[Class and Module Structure: 5](#_Toc182219521)

[Card Class: 5](#_Toc182219522)

[Deck Class: 5](#_Toc182219523)

[Frontend Class: 5](#_Toc182219524)

[Foundation Class: 5](#_Toc182219525)

[Stockpile Class: 5](#_Toc182219526)

[Tableau Class: 5](#_Toc182219527)

[Data Structure: 5](#_Toc182219528)

[Queue (FIFO): 5](#_Toc182219529)

[Stack (LIFO): 5](#_Toc182219530)

[Linked List: 5](#_Toc182219531)

[Array/List: 5](#_Toc182219532)

[Key Algorithms: 5](#_Toc182219533)

[Move Validation: 5](#_Toc182219534)

[Shuffling Algorithm: 6](#_Toc182219535)

[User Interface (UI): 6](#_Toc182219536)

[Pygame Interface: 6](#_Toc182219537)

[Game Screen: 6](#_Toc182219538)

[Win Screen: 6](#_Toc182219539)

[Interaction Design: 6](#_Toc182219540)

[Move base Card Movement: 6](#_Toc182219541)

[Stockpile: 6](#_Toc182219542)

[Wireframes: 7](#_Toc182219543)

[Game Screen: 7](#_Toc182219544)

[Win Screen: 7](#_Toc182219545)

[Testing And Validation: 8](#_Toc182219546)

[Test Cases: 8](#_Toc182219547)

[Legal Move: 8](#_Toc182219548)

[Card Flipping: 8](#_Toc182219549)

[Victory Condition: 8](#_Toc182219550)

[User Testing: 8](#_Toc182219551)

[Challenges and Solutions: 8](#_Toc182219552)

[Technical Challenges: 8](#_Toc182219553)

[a. Handling Multiple Data Structures: 8](#_Toc182219554)

[Solution: 8](#_Toc182219555)

[a. Modular Design: 8](#_Toc182219556)

[Future Improvements: 8](#_Toc182219557)

[Feature Extension: 8](#_Toc182219558)

[Code Optimization: 8](#_Toc182219559)

[Conclusion: 9](#_Toc182219560)

[References: 9](#_Toc182219561)

[Appendices: 9](#_Toc182219562)

[Important Code Snippets: 9](#_Toc182219563)

[Move Cards Form Tableau: 9](#_Toc182219564)

[Place Card to Stock Pile: 10](#_Toc182219565)

[Add Card to Tableau: 11](#_Toc182219566)

[Move Cards to Foundation: 11](#_Toc182219567)

# Introduction:

## Objective:

The goal of this project is to develop a digital version of the classic Solitaire game, adhering to Klondike rules. It will utilize various data structures like stacks, queues, and linked lists. Key features will include card shuffling, validation of legal moves, undo and redo functionality, and determining the conditions for winning.

## Target Audience:

This game is designed for a broad audience, including students learning data structures who seek hands-on experience, game developers interested in creating or analyzing classic games, and Solitaire enthusiasts looking for a dynamic digital experience. By merging fundamental programming concepts with the enduring charm of Klondike Solitaire, it provides both educational benefits and entertainment.

## Scope:

The project includes:

* Setting up the game based on Klondike Solitaire rules
* Implementing card movement and validation
* Creating a graphical user interface using Pygame
* Undo/Redo function.
* Handling win conditions

# Project Overview:

## Game Rules:

In Klondike Solitaire, the objective is to move all cards to the foundation piles, sorted by suit in ascending order from Ace to King. In the tableau, cards must be stacked in descending order with alternating colors. Players draw from the stockpile to make moves, and the game is won when all foundation piles are fully completed.

## User Interface:

The game uses a graphical interface built with Pygame, which allows players to interact with the game via drag-and-drop actions.

# Git lab Link:

<https://gitlab.com/dsa7104145/csc200m24pid102>

# Design and Implementation:

## Class and Module Structure:

### Card Class:

Each card is represented with attributes such as suit, rank, and face orientation (either face-up or face-down).

### Deck Class:

Manages the deck of 52 cards, including functions of initializing the deck, shuffling the cards, and pop cards.

### Frontend Class:

The main game logic controller, managing the overall state of the game. It enforces game rules, handles move between piles, tracks the pause and resume history and monitors win conditions.

### Foundation Class:

**Foundation Piles:** Represented as stacks, where cards are added in ascending order by suit.

### Stockpile Class:

**Stockpile and Waste Pile:** The stockpile is managed as a queue (FIFO), where cards are drawn one at a time.

### Tableau Class:

**Tableau Piles:** represented as stacks or linked lists, allowing for the LIFO nature of card movement.

## Data Structure:

### Queue (FIFO):

The queue data structure is used for the stockpile (draw pile) to ensure cards are drawn in sequence, following a first-in, first-out (FIFO) order. Cards drawn from the stockpile are then moved to the waste pile for later use in the game.

### Stack (LIFO):

The stack data structure is used to manage the tableau and foundation piles, ensuring that only the top card can be accessed and moved. Its last-in, first-out (LIFO) behavior is perfect for handling the organization of these piles.

### Linked List:

The linked list is employed in the tableau piles to enable efficient card insertion and removal, streamlining the process of manipulating cards within the tableau.

### Array/List:

The deck of 52 cards is stored in an array before being shuffled and dealt to the tableau piles. Arrays offer simple access and indexing for efficient card management.

## Key Algorithms:

### Move Validation:

Enforces the rules of Klondike Solitaire, such as alternating colors and sequential ranks.

### Shuffling Algorithm:

Uses Python’s random.shuffle() to randomize the deck of 52 cards.

# User Interface (UI):

## Pygame Interface:

The graphical interface, created with Pygame, offers an interactive and visually appealing experience for playing Solitaire. It features two primary screens:

### Game Screen:

The main screen where the Solitaire game is played, showing the tableau, foundation piles, stockpile, and waste pile and the time showing from the start of the game. The layout mirrors a typical Solitaire arrangement, with distinct areas for each type of pile.

### Win Screen:

This screen is displayed when the player successfully moves all the cards to the foundation piles, signaling the completion of the game. It includes a congratulatory message.

## Interaction Design:

### Move base Card Movement:

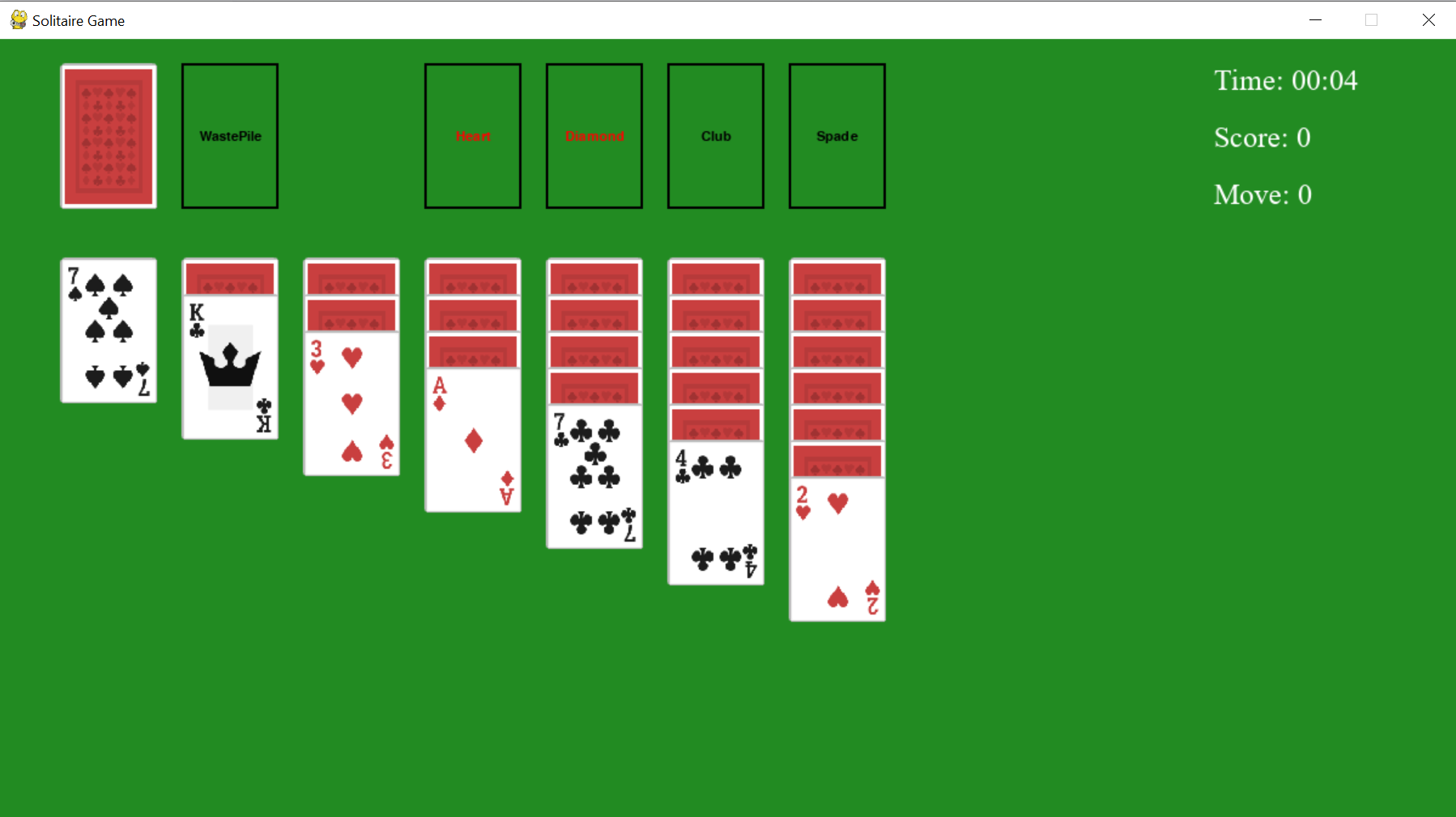
Cards are moved by drag-and-drop. Players click a card and drag it to the destination pile, simplifying user interaction.

### Stockpile:

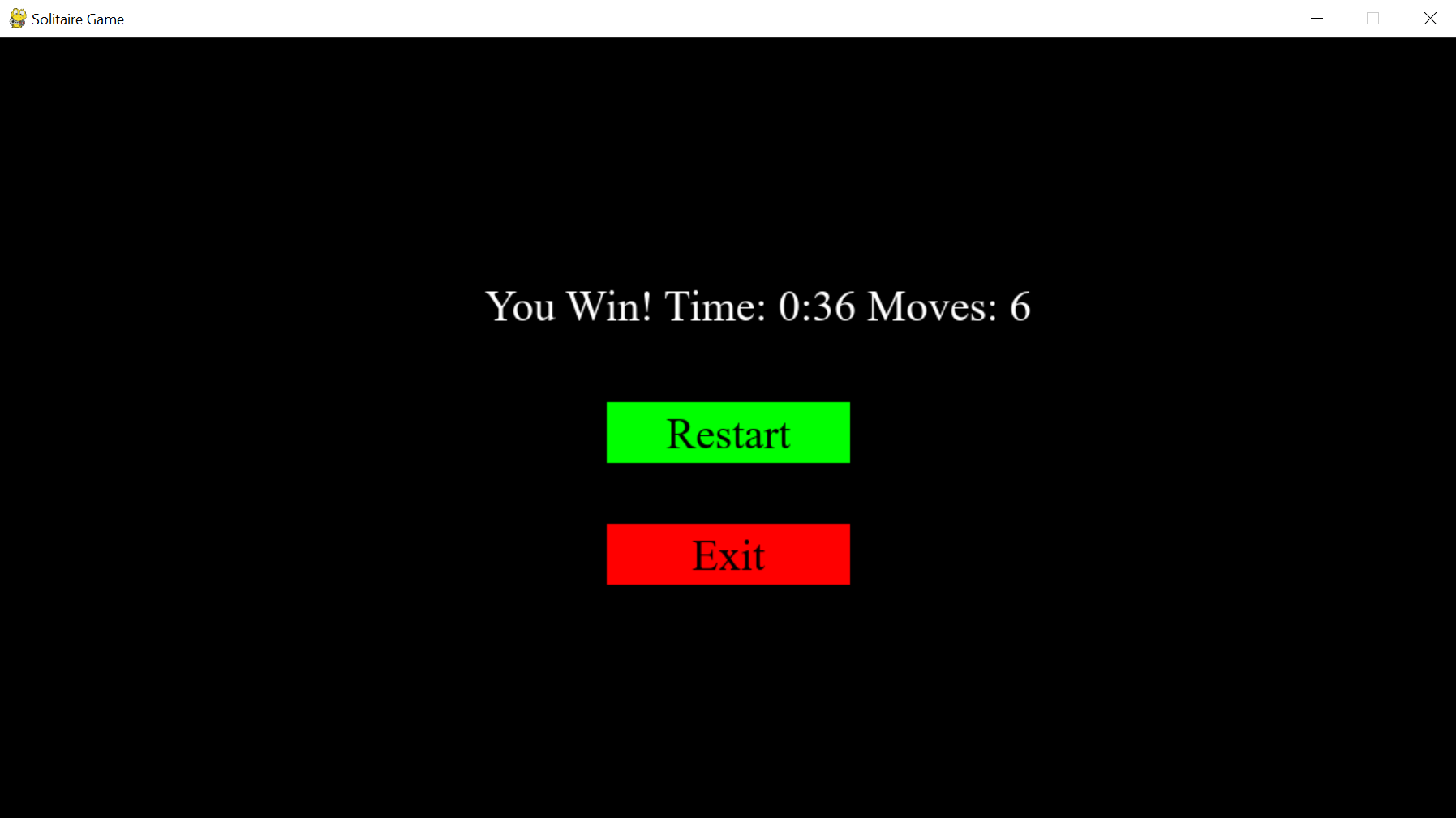
Clicking on the stockpile reveals the next set of cards, which are then placed in the waste pile for possible moves.

## Wireframes:

### Game Screen:



### Win Screen:



# Testing And Validation:

## Test Cases:

### Legal Move:

Ensured that only moves to adhere to the rules of alternating colors and sequential ranks are permitted.

### Card Flipping:

Verified that cards flip correctly when they become the top card after a move in the tableau.

### Victory Condition:

Confirmed that the game ends properly when all foundation piles are completed in the correct order.

## User Testing:

I tested the game to confirm that it is user-friendly and adheres to standard Solitaire mechanics.

# Challenges and Solutions:

## Technical Challenges:

### a. Handling Multiple Data Structures:

Managing stacks, queues, and linked lists in tandem to represent different game components was challenging.

## Solution:

### a. Modular Design:

The project was divided into modular classes, each handling specific data structures, making it easier to manage the complexity.

# Future Improvements:

## Feature Extension:

* Hint System.
* Animation improvement.
* Undo/Redo.

## Code Optimization:

Certain sections such as move validation, card flipping, and piles code could be optimized further to improve performance.

# Conclusion:

This project successfully leveraged data structures to build a fully functional Solitaire game, offering hands-on experience with stacks, queues, and linked lists, and demonstrating their relevance in practical applications. The Pygame-based graphical user interface provided an interactive and immersive gameplay experience. Overall, this project highlights the importance of selecting appropriate data structures and algorithms for efficiently managing complex systems.

# References:

* Solitaire rules from Solitaired.com
* Python and Pygame documentation
* Data Structures and Algorithms by Mr. Nazeef Ul Haq and Mr. Muhammad Waseem

# Appendices:

## Important Code Snippets:

### Move Cards Form Tableau:

def move\_card(self, from\_col\_index, to\_col\_index, card):

        if from\_col\_index is None or to\_col\_index is None:

            return False

        from\_pile = self.Piles[from\_col\_index]

        to\_pile = self.Piles[to\_col\_index]

        if card and card.FaceUp and self.CanAddCard(to\_pile, card):

            cards\_to\_move = from\_pile.RemoveFrom(card)

            if not from\_pile.IsEmpty():

                from\_pile.top().Flip()

            if self.CanAddCard(to\_pile, card) and to\_pile.Head is None:

                to\_pile.PushStack(cards\_to\_move)

                return True

            if cards\_to\_move.Head:

                to\_pile.PushStack(cards\_to\_move)

                new\_top\_card = from\_pile.top()

                if from\_pile is None:

                    from\_pile.Head=None

                if new\_top\_card and not new\_top\_card.FaceUp:

                    new\_top\_card.Flip()

                if to\_pile.top() and not to\_pile.top().FaceUp:

                    to\_pile.top().Flip()

                return True

        return False

### Place Card to Stock Pile:

def place\_card(self, event, dragged\_card, tableau, foundation):

        if event.type == pygame.MOUSEBUTTONUP and event.button == 1:

            mouse\_x, mouse\_y = event.pos

            card\_width = dragged\_card.Image.get\_width()

            card\_height = dragged\_card.Image.get\_height()

            for i, pile in enumerate(tableau.Piles):

                x, y = tableau.column\_position[i]

                pile\_rect = pygame.Rect(x, y, card\_width, card\_height)

                pile\_rect.height = max(card\_height, 500)

                if pile\_rect.collidepoint(mouse\_x, mouse\_y) and tableau.CanAddCard(pile, dragged\_card):

                    pile.Push(dragged\_card)

                    self.DrawnCards.pop()

                    return True

            for i, foundation\_pile in enumerate(foundation.piles):

                pile\_rect = pygame.Rect(foundation\_pile.x, foundation\_pile.y, foundation\_pile.width, foundation\_pile.height)

                if pile\_rect.collidepoint(mouse\_x, mouse\_y) and foundation.CanAddCard(i, dragged\_card):

                    foundation.piles[i].push(dragged\_card)

                    self.DrawnCards.pop()

                    return True

        return False

### Add Card to Tableau:

def CanAddCard(self, to\_pile, CardToAdd):

        if to\_pile.Head is None and CardToAdd.Ranks=='K':

            return True

        if not CardToAdd:

            return False

        if not to\_pile:

            return CardToAdd.Ranks == 'K'

        top\_card = to\_pile.top()

        if not top\_card:

            return False

        top\_card = to\_pile.top()

        opposite\_color = CardToAdd.color != top\_card.color

        RanksOrder = ["ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "jack", "queen", "king"]

        card\_to\_move\_index = RanksOrder.index(CardToAdd.Ranks)

        top\_card\_index = RanksOrder.index(top\_card.Ranks)

        correct\_rank = (card\_to\_move\_index + 1 == top\_card\_index)

        return opposite\_color and correct\_rank

### Move Cards to Foundation:

def move\_card(self, from\_col\_index, card, Piles):

        if from\_col\_index is None:

            return False

        from\_pile = Piles[from\_col\_index]

        if from\_pile.top()!=card:

            return False, Piles

        if card and card.FaceUp and self.can\_add\_card(card):

            cards\_to\_move = from\_pile.RemoveFrom(card)

            if cards\_to\_move.Head:

                new\_top\_card = from\_pile.top()

                if from\_pile is None:

                    from\_pile.Head=None

                if new\_top\_card and not new\_top\_card.FaceUp:

                    new\_top\_card.Flip()

                self.add\_card(card)

                return True, Piles

        return False, Piles