Simulacion Grafica de Solitario Klondike

Implementacion con Python y Tkinter
Estadística Descriptiva y Probabilidad 2025-II

Deyvi Samuel Barrera Rodriguez

14 de diciembre de 2025

Índice

1. Introduccion

El presente documento describe la implementacion de una simulacion grafica del juego de cartas **Solitario Klondike**, conocido popularmente como el solitario clasico de Windows. El proyecto fue desarrollado utilizando Python como lenguaje de programacion y Tkinter como framework para la interfaz grafica.

1.1. Objetivos del Proyecto

- Implementar un juego de mesa completo con interfaz grafica interactiva
- Aplicar principios de Programacion Orientada a Objetos
- Demostrar el manejo de estructuras de datos complejas
- Crear una experiencia de usuario fluida con eventos y animaciones

1.2. Seleccion del Juego

Se eligio el Solitario Klondike por las siguientes razones:

1. Es un juego ampliamente conocido y reconocible
2. Presenta desafios interesantes en terminos de logica de programacion
3. Permite demostrar manejo de multiples estructuras de datos
4. Ofrece oportunidades para implementar drag and drop y animaciones

2. Marco Teorico

2.1. Programacion Orientada a Objetos

La Programacion Orientada a Objetos (POO) es un paradigma de programacion que organiza el codigo en objetos que contienen datos (atributos) y comportamientos (metodos). Los pilares fundamentales de la POO son:

Encapsulamiento: Agrupa datos y metodos relacionados, ocultando los detalles de implementacion.

Herencia: Permite crear nuevas clases basadas en clases existentes.

Polimorfismo: Capacidad de objetos de diferentes clases de responder al mismo mensaje.

Abstraccion: Simplifica sistemas complejos modelando clases apropiadas al problema.

2.2. Tkinter

Tkinter es la biblioteca estandar de Python para crear interfaces graficas de usuario (GUI). Caracteristicas principales:

- Incluida por defecto en la instalacion de Python
- Multiplataforma (Windows, macOS, Linux)
- Basada en el toolkit Tk
- Proporciona widgets como botones, etiquetas, canvas, etc.

2.3. Reglas del Solitario Klondike

El Solitario Klondike se juega con una baraja estandar de 52 cartas:

- **Objetivo:** Construir cuatro fundaciones (pilas base) ordenadas del As al Rey, una por cada palo.
- **Tableau:** Siete columnas donde se distribuyen las cartas al inicio.
- **Mazo:** Cartas restantes que pueden ser robadas.
- **Waste:** Pila de descarte para cartas robadas del mazo.

Movimientos permitidos:

1. Cartas en el tableau deben alternarse en color (rojo-negro) y decrecer en valor
2. Solo los Reyes pueden colocarse en espacios vacios del tableau
3. Las fundaciones deben comenzar con As y construirse en orden ascendente del mismo palo

3. Diseno e Implementacion

3.1. Arquitectura del Sistema

El sistema esta compuesto por tres clases principales que interactuan entre si:

- **Card**: Representa una carta individual
- **CardWidget**: Visualizacion de carta en canvas
- **SolitaireGame**: Logica principal del juego

3.2. Clase Card

La clase `Card` representa una carta individual del juego:

```
@dataclass
class Card:
suit: str       # Palo: picas, corazones, diamantes,
    treboles
rank: str       # Rango: A, 2-10, J, Q, K
face_up: bool = False  # Estado de la carta

def get_value(self) -> int:
return RANKS.index(self.rank) + 1

def get_color(self) -> str:
return COLORS[self.suit]
```

Listing 1: Definicion de la clase Card

Justificacion del diseno:

- Uso de dataclass para reducir codigo
- Metodos auxiliares para comparaciones logicas
- Separacion clara entre datos y comportamiento

3.3. Clase CardWidget

Responsable de la representacion visual de cada carta:

```
def draw(self):
if self.card.face_up:
fill = 'white'
color = self.card.get_color()
# Dibujar simbolos del palo
else:
fill = '#2E5090'
# Dibujar patron de carta boca abajo
```

Listing 2: Metodo draw de CardWidget

Caracteristicas importantes:

- Separacion entre modelo (Card) y vista (CardWidget)
- Metodo move_to() con soporte para animaciones
- Gestion eficiente de elementos graficos en canvas

3.4. Clase SolitaireGame

Clase principal que coordina toda la logica del juego:

3.4.1. Estructuras de Datos

```
1         self.deck: List[Card] = []
2         self.waste: List[Card] = []
3         self.foundations: List[List[Card]] = [[] for _ in range
              (4)]
4         self.tableau: List[List[Card]] = [[] for _ in range(7)]
```

Listing 3: Estructuras de datos principales

Cada estructura tiene un proposito especifico:

Estructura	Proposito
deck	Almacena cartas no robadas
waste	Cartas robadas del mazo
foundations	4 pilas objetivo (una por palo)
tableau	7 columnas de juego

Cuadro 1: Estructuras de datos del juego

3.4.2. Gestion de Eventos

El sistema implementa un sofisticado manejo de eventos mouse:

```
1         self.canvas.bind('<Button-1>', self.on_click)
2         self.canvas.bind('<B1-Motion>', self.on_drag)
3         self.canvas.bind('<ButtonRelease-1>', self.on_release)
```

Listing 4: Vinculacion de eventos

El flujo de eventos sigue este patron:

1. **Click**: Detecta que elemento fue clickeado
2. **Drag**: Actualiza la posicion durante el arrastre
3. **Release**: Valida y ejecuta el movimiento

3.5. Validacion de Movimientos

3.5.1. Movimientos al Tableau

```
1         def can_move_to_tableau(self, cards, pile_idx):
2         pile = self.tableau[pile_idx]
3         first_card = cards[0]
4
5         if not pile:
6         return first_card.rank == 'K'
7
```

```
        top_card = pile[-1]
        return (first_card.get_color() != top_card.get_color()
            and
        first_card.get_value() == top_card.get_value() - 1)
```

Listing 5: Validacion de movimiento al tableau

Condiciones verificadas:

- Pilas vacias solo aceptan Reyes
- Los colores deben alternar (rojo-negro)
- El valor debe ser exactamente uno menor

3.5.2. Movimientos a Fundaciones

```
        def can_move_to_foundation(self, card, foundation_idx):
        foundation = self.foundations[foundation_idx]

        if not foundation:
        return card.rank == 'A'

        top_card = foundation[-1]
        return (card.suit == top_card.suit and
        card.get_value() == top_card.get_value() + 1)
```

Listing 6: Validacion de movimiento a fundacion

Reglas implementadas:

- Fundaciones vacias requieren As
- Mismo palo obligatorio
- Secuencia ascendente estricta

4. Caracteristicas Avanzadas

4.1. Sistema Drag and Drop

El sistema de arrastre utiliza una estructura de datos temporal:

```
        self.drag_data = {
            'cards': [],
            'source': None,
            'start_x': 0,
            'start_y': 0
        }
```

Listing 7: Estructura para drag and drop

4.2. Deteccion de Colisiones

Para determinar donde soltar las cartas:

```
        def on_release(self, event):
        x, y = event.x, event.y

        foundation_idx = self.is_click_on_foundation(x, y)
        if foundation_idx is not None:
        if self.can_move_to_foundation(...):
        self.move_to_foundation(foundation_idx)

        for i in range(7):
        if self.is_position_in_pile(x, y, i):
        if self.can_move_to_tableau(...):
        self.move_to_tableau(i)
```

Listing 8: Deteccion de zona de soltado

4.3. Animaciones

El metodo move_to() incluye soporte para animacion:

```
        def move_to(self, x, y, animate=False):
        if animate:
        steps = 10
        dx = (x - self.x) / steps
        dy = (y - self.y) / steps

        for _ in range(steps):
        self.x += dx
        self.y += dy
        self.draw()
        self.canvas.update()
        self.canvas.after(20)
```

Listing 9: Animacion de movimiento

5. Analisis de Complejidad

5.1. Complejidad Temporal

Operaciones principales y su complejidad:

5.2. Complejidad Espacial

- **Cartas**: O(52) = O(1) - tamanio fijo

Operacion	Complejidad	Justificacion
Inicializar juego	O(n)	Crear y mezclar 52 cartas
Dibujar carta	O(1)	Operaciones graficas constantes
Validar movimiento	O(1)	Comparaciones simples
Redibujar todo	O(n)	n cartas visibles
Detectar colision	O(1)	Calculos aritmeticos simples

Cuadro 2: Analisis de complejidad temporal

- **Widgets graficos**: O(n) donde n menor o igual a 52
- **Pilas de juego**: O(1) - estructuras fijas
- **Total**: O(1) - espacio constante

6. Pruebas y Validacion

6.1. Casos de Prueba

6.1.1. Prueba 1: Inicializacion del Juego

Objetivo: Verificar que el juego se inicializa correctamente.
Procedimiento:

1. Ejecutar el programa
2. Observar la distribucion inicial de cartas
3. Verificar que hay 7 columnas en el tableau
4. Confirmar que la ultima carta de cada columna esta boca arriba

Resultado esperado: 28 cartas distribuidas, 24 en mazo.

6.1.2. Prueba 2: Movimientos Validos

Objetivo: Validar la logica de movimientos permitidos.
Casos:

- Mover carta negra sobre carta roja de valor superior
- Intentar mover carta del mismo color (debe rechazarse)
- Colocar Rey en espacio vacio
- Mover As a fundacion vacia

6.1.3. Prueba 3: Condicion de Victoria

Objetivo: Verificar deteccion de victoria.
Procedimiento:

1. Manipular el juego hasta completar las 4 fundaciones
2. Verificar aparicion del mensaje de victoria
3. Confirmar que se muestra el numero de movimientos

6.2. Resultados de Pruebas

Caso de Prueba	Estado	Observaciones
Inicializacion	Paso	Distribucion correcta
Movimientos validos	Paso	Validacion funcional
Movimientos invalidos	Paso	Rechazos apropiados
Drag and Drop	Paso	Fluido y preciso
Condicion de victoria	Paso	Deteccion correcta
Reinicio de juego	Paso	Estado limpio

Cuadro 3: Resultados de pruebas funcionales

7. Aplicacion de Conceptos de la Unidad 4

7.1. Estructuras de Datos

El proyecto hace uso extensivo de estructuras de datos de Python:

Listas Para representar pilas de cartas

Diccionarios Para configuracion y estado temporal

Tuplas Para retornos multiples inmutables

7.2. Algoritmos Implementados

7.2.1. Algoritmo de Mezcla

Utiliza el algoritmo Fisher-Yates implementado en random.shuffle():

```
1         self.deck = [Card(suit, rank) for suit in SUITS
2         for rank in RANKS]
3         random.shuffle(self.deck)
```

Complejidad: O(n) donde n = 52

7.2.2. Algoritmo de Busqueda

Para detectar clics, se utiliza busqueda lineal:

```
        def is_click_on_tableau(self, x, y):
        for i, pile in enumerate(self.tableau):
        for j, card in enumerate(pile):
        if is_inside(x, y, card_bounds):
        return i, j
```

Complejidad: O(7 por m) donde m es el promedio de cartas por columna.

7.3. Programacion Orientada a Objetos

7.3.1. Encapsulamiento

Cada clase encapsula su propia logica:

- Card: Datos y operaciones de una carta
- CardWidget: Renderizado visual
- SolitaireGame: Logica del juego

7.3.2. Abstraccion

Los metodos publicos ocultan detalles de implementacion:

```
        # Interfaz simple
        game.draw_from_deck()

        # Implementacion interna compleja
        def draw_from_deck(self):
        if self.deck:
        card = self.deck.pop()
        card.face_up = True
        self.waste.append(card)
        self.increment_moves()
```

7.3.3. Composicion

SolitaireGame compone multiples CardWidget:

```
        self.card_widgets = []

        for card in visible_cards:
        widget = CardWidget(self.canvas, card, x, y)
        self.card_widgets.append(widget)
```

8. Mejoras Futuras

8.1. Funcionalidades Adicionales

1. Sistema de Pistas: Resaltar movimientos posibles
2. Deshacer/Rehacer: Stack de estados anteriores
3. Estadisticas: Guardar records y porcentaje de victorias
4. Diferentes Variantes: Spider, FreeCell, etc.
5. Modo Tutorial: Guia para nuevos jugadores

8.2. Optimizaciones Tecnicas

1. Renderizado Selectivo: Solo redibujar elementos modificados
2. Doble Clic Automatico: Mover automaticamente a fundaciones
3. Animaciones Suavizadas: Usar interpolacion cuadratica
4. Resolucion Adaptativa: Ajustar tamanio segun ventana

8.3. Arquitectura

1. Patron MVC: Separar modelo, vista y controlador
2. Sistema de Eventos: Implementar observer pattern
3. Persistencia: Guardar y cargar estado del juego
4. Testing: Suite de pruebas unitarias con pytest

9. Conclusiones

9.1. Logros del Proyecto

El proyecto cumplio exitosamente con todos los objetivos planteados:

- Implementacion completa de un juego de mesa funcional
- Interfaz grafica atractiva y responsiva
- Aplicacion correcta de principios de POO
- Manejo eficiente de estructuras de datos
- Sistema de eventos robusto

9.2. Aprendizajes Clave

Durante el desarrollo se obtuvieron aprendizajes valiosos:

1. Diseno de Software: La importancia de planificar la arquitectura
2. Gestion de Estado: Mantener sincronizacion entre modelo y vista
3. Debugging Visual: Tecnicas para depurar interfaces graficas
4. Manejo de Eventos: Complejidad de sistemas interactivos

9.3. Aplicacion Practica

Este proyecto demuestra competencias en:

- Programacion Orientada a Objetos
- Estructuras de datos complejas
- Desarrollo de interfaces graficas
- Algoritmos de validacion
- Gestion de eventos de usuario

9.4. Reflexion Final

La implementacion del Solitario Klondike representa un desafio completo que integra multiples areas de la programacion. Este proyecto demuestra la capacidad de:

- Analizar un problema complejo
- Disenar una solucion escalable
- Implementar codigo mantenible
- Crear experiencias de usuario agradables

El codigo resultante es extensible, permitiendo agregar facilmente nuevas caracteristicas o variantes del juego.

10. Referencias

1. Python Software Foundation. (2024). Python Documentation. https://docs.python.org/3/
2. Tkinter Documentation. (2024). Tk Commands. https://docs.python.org/3/library/tkinter.html
3. Gamma, E. y otros (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
4. Lutz, M. (2013). Learning Python, 5th Edition. O'Reilly Media.
5. Wikipedia Contributors. (2024). Klondike solitaire. Wikipedia.
6. Python.org. (2024). Python Type Hints. https://docs.python.org/3/library/typing.html
7. Real Python. (2024). Python GUI Programming With Tkinter.

A. Codigo Completo

El codigo completo del proyecto esta disponible en el archivo solitaire_klondike.py.

A.1. Estructura de Archivos

El proyecto consta de los siguientes archivos:

- solitaire_klondike.py - Codigo principal
- informe.tex - Este documento
- informe.pdf - Documento compilado
- README.md - Instrucciones de ejecucion

A.2. Requisitos del Sistema

- Python 3.8 o superior
- Tkinter incluido en instalacion estandar de Python
- Sistema operativo: Windows, macOS, o Linux
- Resolucion minima: 800x650 pixeles

A.3. Instrucciones de Ejecucion

Para ejecutar el juego:

```
cd proyecto_solitario
python solitaire_klondike.py
```

Para compilar el informe LaTeX:

```
pdflatex informe.tex
pdflatex informe.tex
```

B. Glosario de Terminos

Canvas Area de dibujo en Tkinter donde se renderizan elementos graficos

Dataclass Decorador de Python para crear clases con menos codigo

Drag and Drop Tecnica de interaccion donde se arrastra y suelta un elemento

Event Handler Funcion que responde a eventos del usuario

Foundation Pila objetivo donde se construyen secuencias As-Rey

Framework Conjunto de herramientas y bibliotecas para facilitar desarrollo

GUI Interfaz Grafica de Usuario

Tableau Area principal de juego con 7 columnas de cartas

Widget Componente visual reutilizable de una interfaz grafica

Waste Pila de descarte para cartas robadas del mazo