**Universidad Tecnológica Centroamericana**

**UNITEC**

# Proyecto I

# TDA Fundamentales


**Estudiante:**

Edilson Fernando González 11211070


**Estructura de Datos, Sección: 3524**

**Catedrático: Ing. Carlos Arias**

**Tegucigalpa, M.D.C., 17 de mayo de 2013**

# Contenido

# Introducción

El presente documento contiene un resumen, manual técnico y de usuario para el juego Video Poker, presentado como proyecto de la clase de Estructura de Datos. Este tiene como objetivo el uso de los Tipos de Datos Abstractos Fundamentales: Lista, Pila y Cola.

Incluye un resumen de cada una de las clases y métodos utilizados para implementar el juego completo. Además de una explicación resumida de cómo se juega al póker.

El software fue desarrollado en Java y puede obtener el código fuente en la dirección https://github.com/efgm1024/Poker_EstructuraDatos.git

# Marco Teórico

## Video póker

Video póker es una variante del juego póker original en el cual, un solo jugador posee una mano de cartas, puede retener a voluntad de 1 hasta 5 cartas para luego evaluar la jugada. Los tipos pueden ser:

| Nombre de la jugada | Descripción |
|---|---|
| **Royal Straight Flush** | Cinco cartas seguidas del mismo palo del 10 al As. |
| **Straight Flush** | Cinco cartas seguidas del mismo palo, pero que no tiene As como carta alta. |
| **Four of a kind** | Cuatro cartas iguales en su valor. |
| **Full House** | Tres cartas iguales, más otras dos iguales. |
| **Flush** | Cinco cartas del mismo palo sin ser seguidas. |
| **Straight** | Cinco cartas seguidas de palos diferentes. |
| **Three of a kind** | Tres cartas iguales en su valor. |
| **Two Pairs** | Dos pares de cartas. |
| **Jacks or Better** | Una pareja de J, Q, K o As |

Esta variante permite que se comience una partida con cierta cantidad de dinero apostado y vaya aumentando a partir de la jugada que se ha obtenido. La tabla de apuestas es la siguiente:

| Nombre de la jugada | x 1 | x 2 | x 3 | x 4 | x 5 |
|---|---|---|---|---|---|
| **Royal Straight Flush** | 250 | 500 | 750 | 1000 | 4000 |
| **Straight Flush** | 50 | 100 | 150 | 200 | 250 |
| **Four of a kind** | 25 | 50 | 75 | 100 | 125 |
| **Full House** | 9 | 18 | 27 | 36 | 45 |

| | | | | | |
|---|---|---|---|---|---|
| **Flush** | 6 | 12 | 18 | 24 | 30 |
| **Straight** | 4 | 8 | 12 | 16 | 20 |
| **Three of a kind** | 3 | 6 | 9 | 12 | 15 |
| **Two Pairs** | 2 | 4 | 6 | 8 | 10 |
| **Jacks or Better** | 1 | 2 | 3 | 4 | 5 |

El multiplicador es determinado al inicio de la partida.

Además, si el usuario obtiene una jugada de las descritas anteriormente tiene la opción de recolectar lo que ha ganado hasta el momento o apostarlo para ganar el doble. El modo doble consiste en que el Dealer saca una carta y el jugador tendrá entonces que sacar otra carta, si ésta es mayor que la del Dealer gana en el doble y vuelve a tener las mismas opciones iniciales, caso contrario pierde lo ganado hasta ese momento.

Si el jugador pierde en cualquiera de los casos descritos anteriormente se le resta el multiplicador aportado y se reinicia a la situación inicial del juego.

## Tipos de datos abstractos lista, pila y cola

En general, un tipo de dato abstracto es aquel que posee datos y operaciones solamente definidas, no implementadas.

El TDA Lista se define como una secuencia de elementos donde cada uno tiene una posición. Las operaciones para este TDA son:

| Operación | Descripción |
|---|---|
| **Init() : void** | Inicializa la lista en memoria |
| **Insert(E, p) : boolean** | Inserta un elemento en la lista, si la posición suministrada es válida. Retorna true si la operación fue exitosa, false en caso contrario. |
| **First() : E** | Retorna el primer elemento de la lista, si existe, caso contrario retorna un null. |

| | |
|---|---|
| **Last() : E** | Retorna el último elemento de la lista, si existe, caso contrario retorna un null. |
| **Size() : int** | Retorna un entero con el tamaño de la lista. |
| **Capacity() : int** | Retorna un entero que representa la capacidad de esa lista. |
| **Remove(p) : E** | Elimina un objeto de la lista, si la posición es válida, caso contrario retorna un null. |
| **isEmpty() : boolean** | Retorna un valor booleano que expresa si la lista está vacía, true, o contiene elementos, false. |
| **isFull() : boolean** | Retorna un valor booleano que expresa si la lista está llena, true, o no, false. |
| **Clear() : void** | Elimina todos los elementos de una lista |
| **indexOf(E) : int** | Método de búsqueda que localiza un objeto de la lista y retorna el índice donde si encuentra. Si éste no pertenece a la lista, retorna -1. |
| **Get(p) : E** | Retorna el elemento en la posición p suministrada, si es válida, caso contrario retorna null. |

La pila es una estructura de datos en donde los elementos se insertar por un lado y se extraen desde ese mismo lado comenzando por el último insertado. Las operaciones del TDA son:

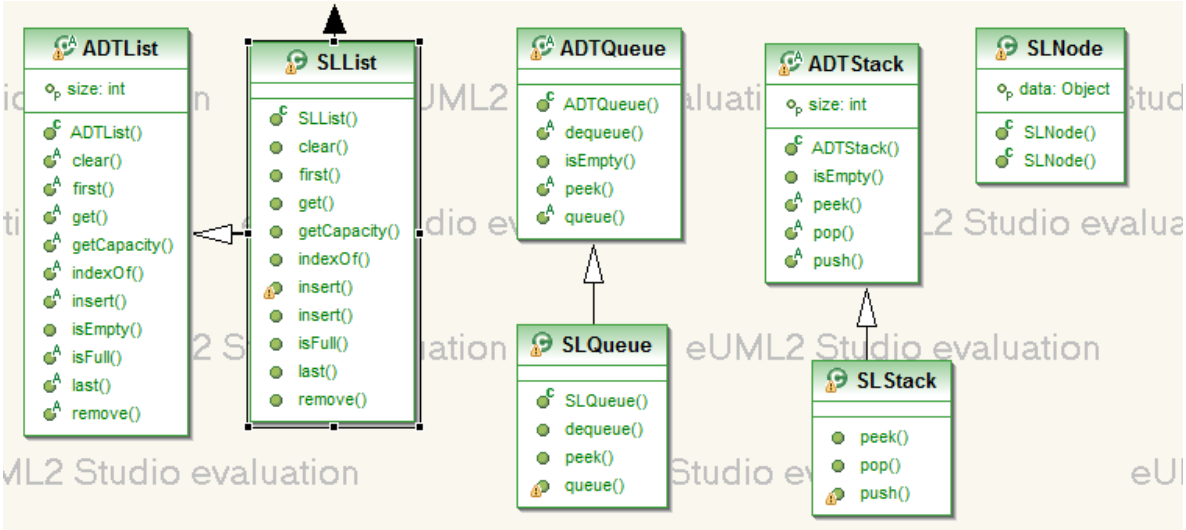
| Operación | Descripción |
|---|---|
| **Init() : void** | Inicializa la pila en memoria |

| | |
|---|---|
| **Push(E) : boolean** | Inserta un elemento en la parte superior de la pila y retorna true. Retorna false si la operación no fue exitosa |
| **Pop() : E** | Retorna el elemento que está en la parte superior de la pila y lo extrae de ésta. Si no hay un elemento retorna null. |
| **Peek() : E** | Retorna el elemento que está en la parte superior de la pila pero sin extraerlo de ésta. Retorna null si la pila está vacía |

El tipo de datos abstracto cola se define como una estructura de datos lineal, en donde los elementos se insertar por un lado y se extraen por otro. Las operaciones del TDA son:

| Operación | Descripción |
|---|---|
| **Init() : void** | Inicializa la cola en memoria |
| **queue(E) : boolean** | Inserta un elemento en la parte inferior de la cola y retorna true, si la operación fue exitosa, false en caso contrario. |
| **dequeue() : E** | Retorna el elemento que está en la parte superior de la cola y lo extrae de ésta. Si no hay un elemento retorna null. |
| **Peek() : E** | Retorna el elemento que está en la parte superior de la cola pero sin extraerlo de ésta. Retorna null si la cola está vacía |

# Implementación




## Paquete edu.unitec.adt

### ADTList.java

Clase Padre Abstracta ADTList. Posee las definiciones de los métodos que una lista conlleva. Implementa Serializable para ser guardada en archivos binarios.

| *Operación* | *Descripción* |
|---|---|
| *ADTList()* | Constructor de la lista. |
| *clear()* | Elimina todo el contenido de la lista. |
| *first()* | Retorna el objeto que se encuentra en la primera posición de la lista. |
| *get(int p)* | Retorna el objeto que está en la posición especificada como parámetro del método, si ésta es válida, caso contrario retorna null. |
| *getCapacity()* | Retorna la capacidad actual de la lista. |
| *getSize()* | Método accesor para el tamaño de una lista. |
| *indexOf(Object E)* | Busca la posición de un objeto específico de la lista. |
| *insert(Object E, int p)* | Inserta un elemento en la lista. |
| *isEmpty()* | Retorna un booleano que identifica si la lista está vacía o no. |

| | |
|---|---|
| *isFull()* | Método que retorna un indicador booleano de is la lista está llena o no. |
| *last()* | Retorna el último objeto de la lista. |
| *remove(int p)* | Elimina un elemento de la lista. |

### ADTStack.java

Clase Padre Abstracta ADTStrack que modela el comportamiento de una pila. Implementa Serializable para almacenarse en archivos binarios.

| *Operación* | *Descripción* |
|---|---|
| *ADTStack()* | Constructor por defecto de la clase. |
| *getSize()* | Método accesor para el tamaño de la pila. |
| *isEmpty()* | Método que devuelve una representación booleana del estado de la pila. |
| *peek()* | Retorna el último elemento ingresado a la pila sin extraerlo de ésta. |
| *pop()* | Saca el último elemento ingresado a la pila. |
| *push(Object E)* | Ingresa un elemento a la pila. |

### ADTQueue.java

Clase Padre Abstracta ADTQueue, define todos los métodos necesarios para una cola. Implementa Serializable para ser guardada en archivos binarios.

| *Operación* | *Descripción* |
|---|---|
| *ADTQueue()* | Constructor predeterminado que inicializa la cola. |
| *dequeue()* | Elimina un elemento del inicio de la cola. |
| *isEmpty()* | Metodo que retorna un booleano que identifica si la cola está vacía o no. |

| | |
|---|---|
| *peek()* | Método que retorna el elemento inicial de la cola sin extraerlo de ella. |
| *queue(Object E)* | Inserta un elemento al final de la cola. |

### SLNode.java

Estructura nodo que encapsula los datos y mantiene un apuntador al siguiente elemento nodo. Implementa Serializable para ser almacenado en archivos binarios.

| *Operación* | *Descripción* |
|---|---|
| *SLNode()* | Constructor por defecto. |
| *SLNode(Object data)* | Constructor que recibe una referencia a un objeto para ser almacenado en la estructura, inicializa la referencia al siguiente nodo en null. |
| *getData()* | Método accesor para los datos del nodo. |
| *getNext()* | Método accesor para la referencia al siguiente nodo. |
| *setData(Object data)* | Mutador para los datos del nodo. |
| *setNext(SLNode n)* | Mutador para la referencia al siguiente nodo. |

### SLList.java

Clase SLList que implementa los métodos de ADTList. Esta implementación está basada en listas con enlaces simples, por lo tanto revise que el rendimiento de su programa no se vea afectado por el orden de las operaciones de esta clase. Implementa Serizalizable para guardarse en archivos binarios.

| *Operación* | *Descripción* |
|---|---|
| *SLList()* | Constructor de la clase, inicializa la cabeza de la lista en null. |
| *clear()* | Implementación del método clear de ADTList. |
| *first()* | Implementación del método first de ADTList. Retorna el objeto que se encuentra en la primera posición de la lista. |

| | |
|---|---|
| *get(int p)* | Implementación del método get de ADTList. Retorna el objeto que está en la posición especificada como parámetro del método, si ésta es válida, caso contrario retorna null. |
| *getCapacity()* | Implementación del método getCapacity de ADTList. Retorna siempre -1 debido a que la capacidad de la lista depende de cuanta memoria haya disponible para reservar espacio para los nodos. |
| *indexOf(Object E)* | Implementación del método indexOf de ADTList. Busca la posición de un objeto específico de la lista. Si lo encuentra retorna su posición, caso contrario retorna -1. Le recomendamos sobrescriba el método equals del objeto para la comparación de igualdad entre dos objetos. |
| *insert(Object E, int p)* | Implementación del método insert de ADTList. Inserta un elemento en la lista. |
| *insert(Object E)* | Agrega un elemento al final de la lista. |
| *isFull()* | Implementación del método isFull de ADTList. Retorna siempre false debido a que depende de la memoria que se pueda reservar para crear nuevos nodos. |
| *last()* | Implementación del método last de ADTList. Retorna el último objeto de la lista. |
| *remove(int p)* | Implementación del método remove de ADTList. Elimina un elemento de la lista. |

### SLStack.java

Implementación mediante nodos con enlaces simples de una pila.

| *Operación* | *Descripción* |
|---|---|
| *SLStack()* | Constructor por defecto de la clase. |

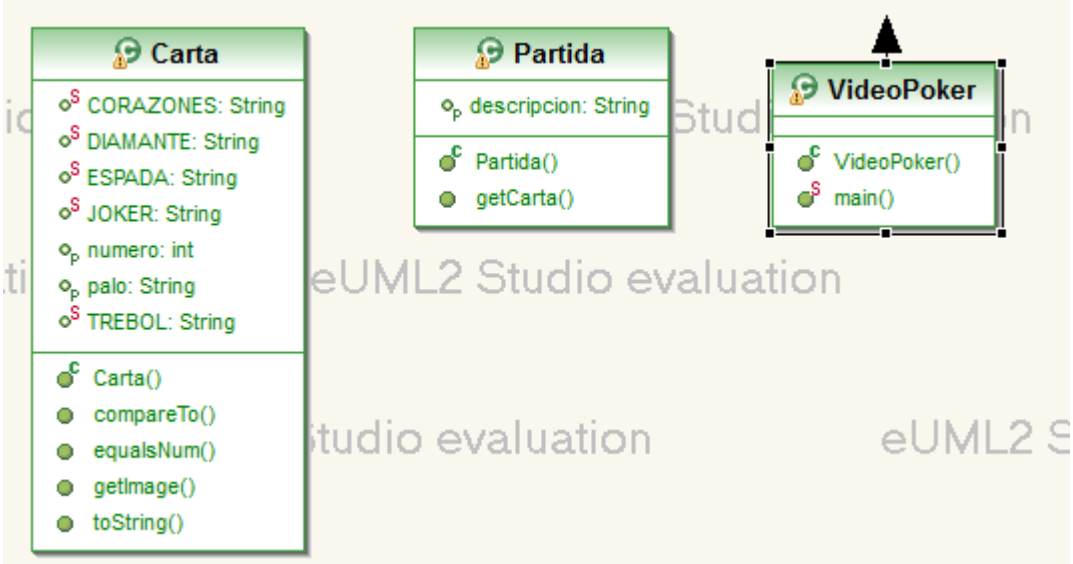
| | |
|---|---|
| *peek()* | Implementación del método peek de ADTStack. Retorna el último elemento ingresado a la pila sin extraerlo de ésta. |
| *pop()* | Implementación del método pop de ADTStack. Saca el último elemento ingresado a la pila. Retorna null si la pila está vacía. |
| *push(Object E)* | Implementación del método push de ADTStack. Ingresa un elemento a la pila. |

### SLQueue.java

Implementación con nodos enlazados de manera simple de una cola. Utiliza mecanismos similares a los de SLList.

| *Operación* | *Descripción* |
|---|---|
| *SLQueue()* | Constructor predeterminado de una SLQueue. |
| *dequeue()* | Implementación del método dequeue de ADTQueue. Elimina un elemento del inicio de la cola. |
| *peek()* | Implementación del método peek de ADTQueue. Método que retorna el elemento inicial de la cola sin extraerlo de ella. |
| *queue(Object E)* | Implementación del método queue de ADTQueue. Inserta un elemento al final de la cola. |

## Paquete edu.unitec.videopoker




### Carta.java

Almacena toda la información necesaria para representar una carta. Además de guardar el número y el palo, almacena también la imagen de la carta. Implementa la interfaz Serializable del paquete java.io para ser guardada posteriormente en archivos binarios.

| *Operación* | *Descripción* |
|---|---|
| *Carta(int numero, String palo, ImageIcon img)* | Inicializa la carta con su número, palo e imagen correspondiente. |
| *compareTo(Object other)* | Método que compara dos cartas. El As siempre será mayor que cualquier otra carta dentro de la baraja. Retorna un entero que identifica si un el this y otro objeto es mayor, menor o igual. |
| *equalsNum(Carta other)* | Método que compara dos cartas para saber si son iguales en número. |
| *toString()* | Representación String de un objeto Carta. |
| *getImage()* | Método accesor que retorna la imagen de la carta. |
| *getNumero()* | Método accesor que retorna el número de la carta. |

| | |
|---|---|
| *getPalo()* | Método accesor que retorna a que palo pertenece la carta. Revisar las constantes que provee esta clase. |

### Partida.java

Esta clase sirve para almacenar las cartas de una partida (5) y una descripción de esta (si ha perdido o ha ganado con alguna combinación). Implementa Serializable para guardarse en archivos binarios.

| *Operación* | *Descripción* |
|---|---|
| *Partida(Carta[] cards, String des)* | Constructor que inicializa una partida a partir de un arreglo de cartas y una descripción de lo que sucedió en la partida. |
| *getCarta(int p)* | Obtiene alguna de las cartas dentro de la partida almacenada (0-4). |
| *getDescripcion()* | Método accesor para la descripción almacenada de la partida. |

### VideoPoker.java

Ventana principal del juego Video Poker.

| *Operación* | *Descripción* |
|---|---|
| *ajustarApuesta(int RANGO)* | Ajusta la tabla de apuestas dependiendo del rango designado. |
| *animacionCambio()* | Animación que se encarga de asignar a los botones que simulan las cartas la parte trasera de éstas para ocultar las verdaderas imágenes. |
| *borrarSeleccion()* | Si los botones están seleccionados, borra la selección para que vuelvan a su estado original. |
| *btn_backActionPerformed(ActionEvent evt)* | Se mueve hacia atrás en la lista de partidas. |

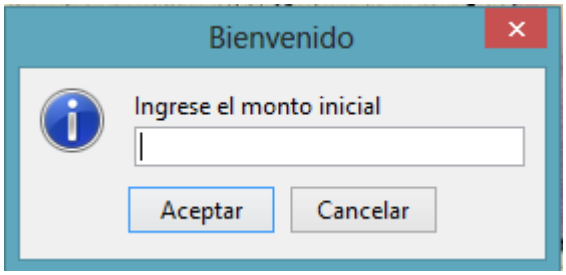
| | |
|---|---|
| *btn_betActionPerformed(ActionEvent evt)* | Ajusta la apuesta en cada accionar del botón, mantiene un contador de rango que activa los diferentes rangos (1-5) para cambiar la apuesta de la tabla. |
| *btn_collectActionPerformed(ActionEvent evt)* | Suma lo ganado hasta el momento dentro de los créditos y el pago se ve reflejado en los labels. |
| *btn_dealActionPerformed(ActionEvent evt)* | Llama al método deal para lanzar una mano de cartas por primera vez. |
| *btn_doubleUpActionPerformed(ActionEvent evt)* | Configura toda la situación inicial del modo double up. |
| *btn_drawActionPerformed(ActionEvent evt)* | Llama al método draw para sustituir las castas no retenidas. |
| *btn_maxbetActionPerformed(ActionEvent evt)* | Apuesta máxima. |
| *btn_nextActionPerformed(ActionEvent evt)* | Se mueve hacia adelante en la lista de partidas. |
| *carta1ActionPerformed(ActionEvent evt)* | Metodo que en modo double up activa el comprobador de cartas para él y en modo normal marca la carta en estado "Held" para su retención luego de presionar el botón draw |
| *carta2ActionPerformed(ActionEvent evt)* | Metodo que en modo double up activa el comprobador de cartas para él y en modo normal marca la carta en estado "Held" para su retención luego de presionar el botón draw |
| *carta3ActionPerformed(ActionEvent evt)* | Metodo que en modo double up activa el comprobador de cartas para |

| | |
|---|---|
| | él y en modo normal marca la carta en estado "Held" para su retención luego de presionar el botón draw |
| *carta4ActionPerformed(ActionEvent evt)* | Metodo que en modo double up activa el comprobador de cartas para él y en modo normal marca la carta en estado "Held" para su retención luego de presionar el botón draw |
| *carta5ActionPerformed(ActionEvent evt)* | Metodo que en modo double up activa el comprobador de cartas para él y en modo normal marca la carta en estado "Held" para su retención luego de presionar el botón draw |
| *comprobadorDoubleUp()* | Método comprobador para el modo Double Up. |
| *deal()* | Activa la primera evaluación de las cartas. |
| *draw()* | Este método sustituye las cartas que no fueron marcadas con "Held" y llama al método evaluador para verificar que combinacion ganó el usuario. |
| *escribirArchivoPartidas(Partida part)* | Método que se encarga de escribir una partida dentro del historial de partidas, que se conforma de un archivo binario. |
| *inicializarCartas()* | Inicializa toda una baraja de cartas de manera ordenada cargando las imágenes con éstas. |

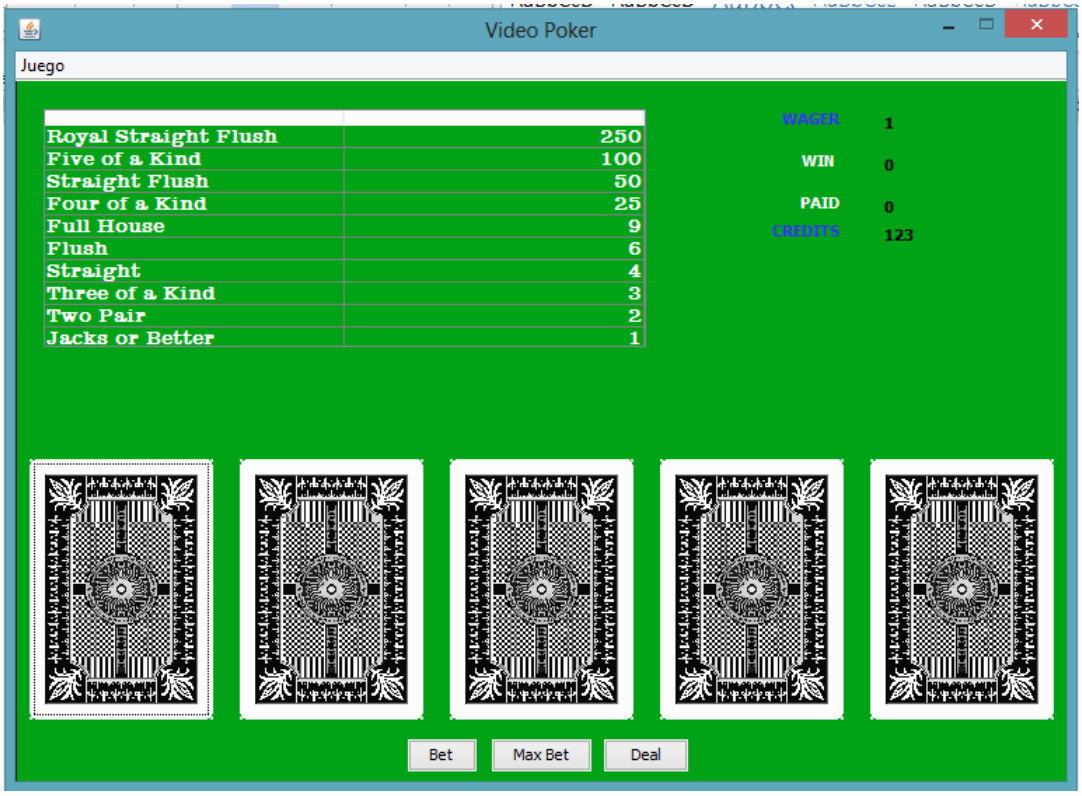
| | |
|---|---|
| *initComponents()* | This method is called from within the constructor to initialize the form. |
| *leerArchivoPartidas()* | Método que se encarga de la lectura del archivo partidas. |
| *mnu_juego_partidasActionPerformed(ActionEvent evt)* | Lanza la ventana de partidas para que sean visualizadas por el usuario. |
| *mnu_juego_salirActionPerformed(ActionEvent evt)* | Para salir del juego. |
| *nuevaBarajar()* | Crea una nueva baraja para usar en el programa. |
| *ocultarHelds()* | Oculta las etiquetas "Held" posicionadas en la parte superior de las cartas. |
| *pedirMonto()* | Método que despliega InputDialog de JOptionPane para pedir el monto inicial del juego, si el usuario no ingresa ningún monto y cierra la ventana termina la ejecución del programa. |
| *situacionInicial()* | Prepara todos los componentes para la situación inicial del juego. |
| *validadorCombinaciones()* | Se encarga de validar la combinación de cartas que obtuvo el usuario. |
| *ventanaPartidasWindowActivated(WindowEvent evt)* | Cada vez que se activa la ventana de partidas, se lee el archivo de partidas para actualizar la lista de partidas. |

## Manual de Usuario

1. Para comenzar el juego debe ingresar el monto inicial a apostar. Esta será su cuenta ¡Hágala crecer a una fortuna! **Ingrese el monto y presione Enter.**

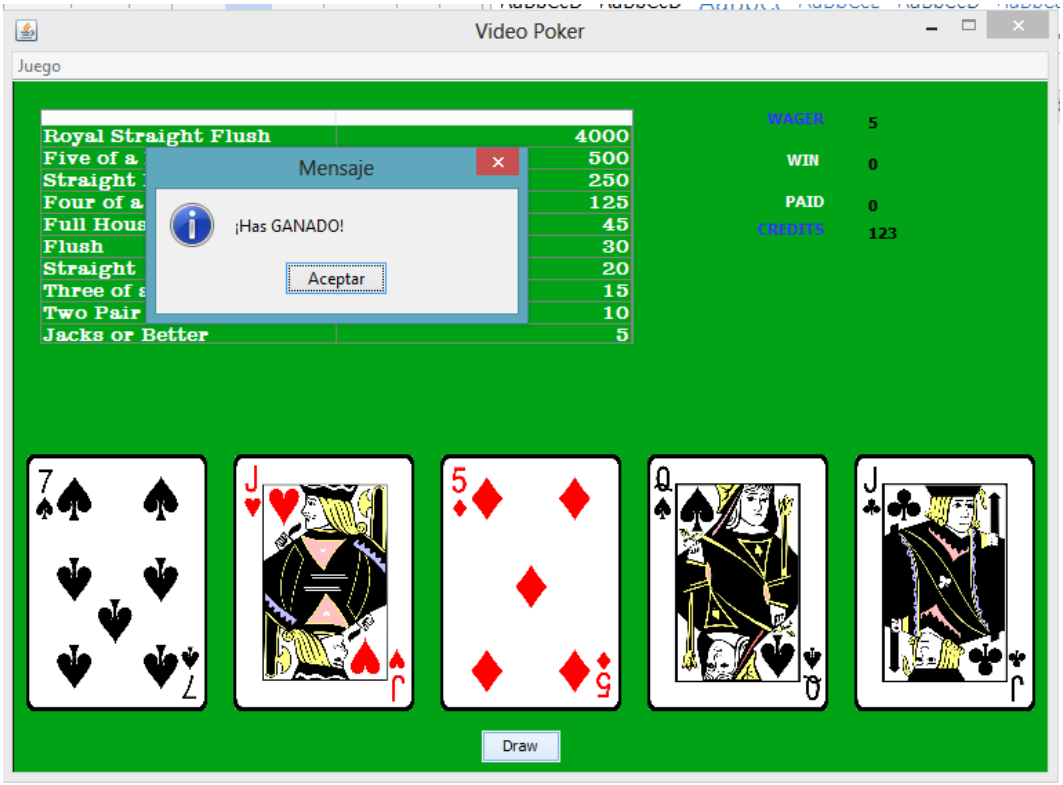



2. Esta es la ventana principal del juego. En ella puede comenzar a apostar y hacer su fortuna. Para comenzar puede ajustar la apuesta con el botón Bet, verá que la tabla de apuestas se ajusta a su gusto. Si te consideras con suerte hoy puedes apostar la máxima cantidad con el botón Max Bet. Si solo ajustaste la apuesta como cualquiera entonces presiona el botón deal.



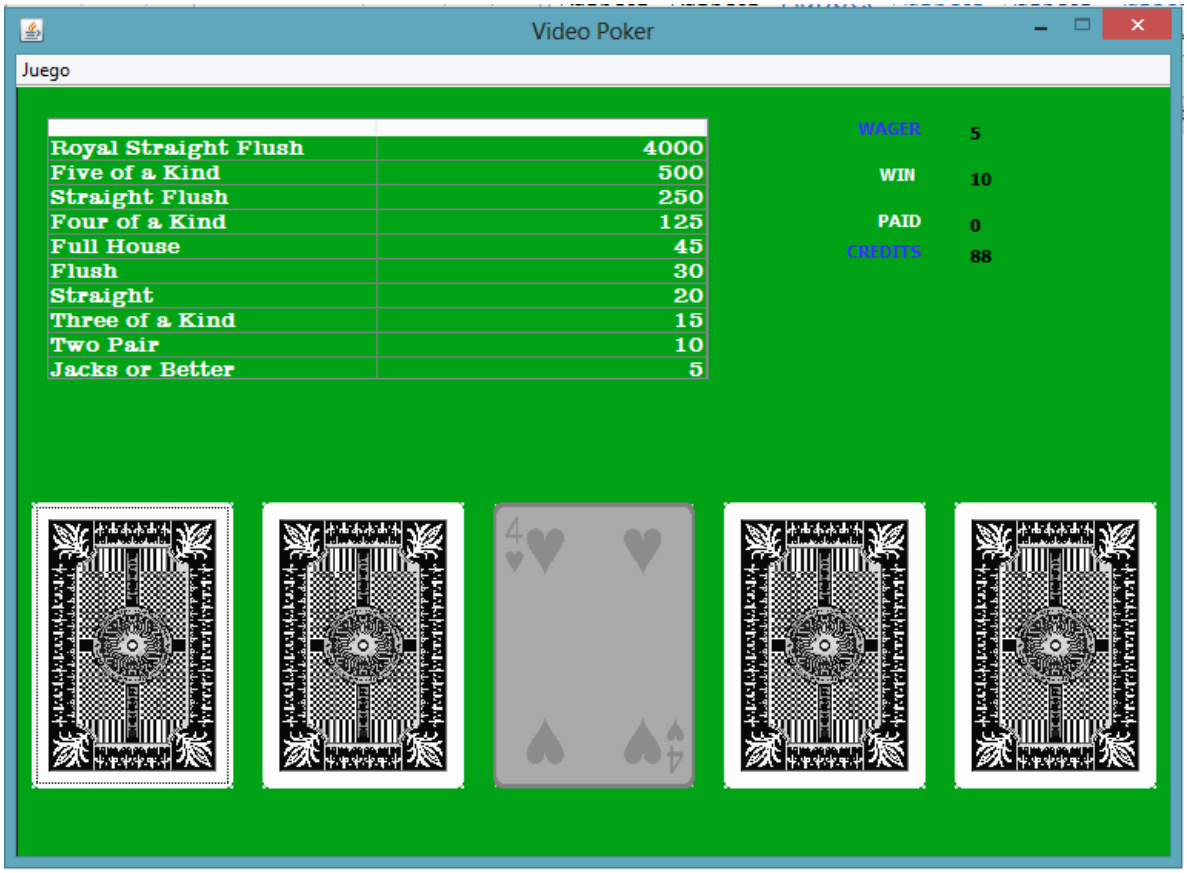
3. Te hemos dado una mano de cartas. Ahora escoge bien tus cartas, recuerda observan la tabla superior para saber las combinaciones posibles que puedes alcanzar. Cuando decidas bien tu jugada presionando las cartas que quieres retener cliquea Deal.


4. ¡Enhorabuena! Has logrado un Jacks or Better. Presiona aceptar para continuar.

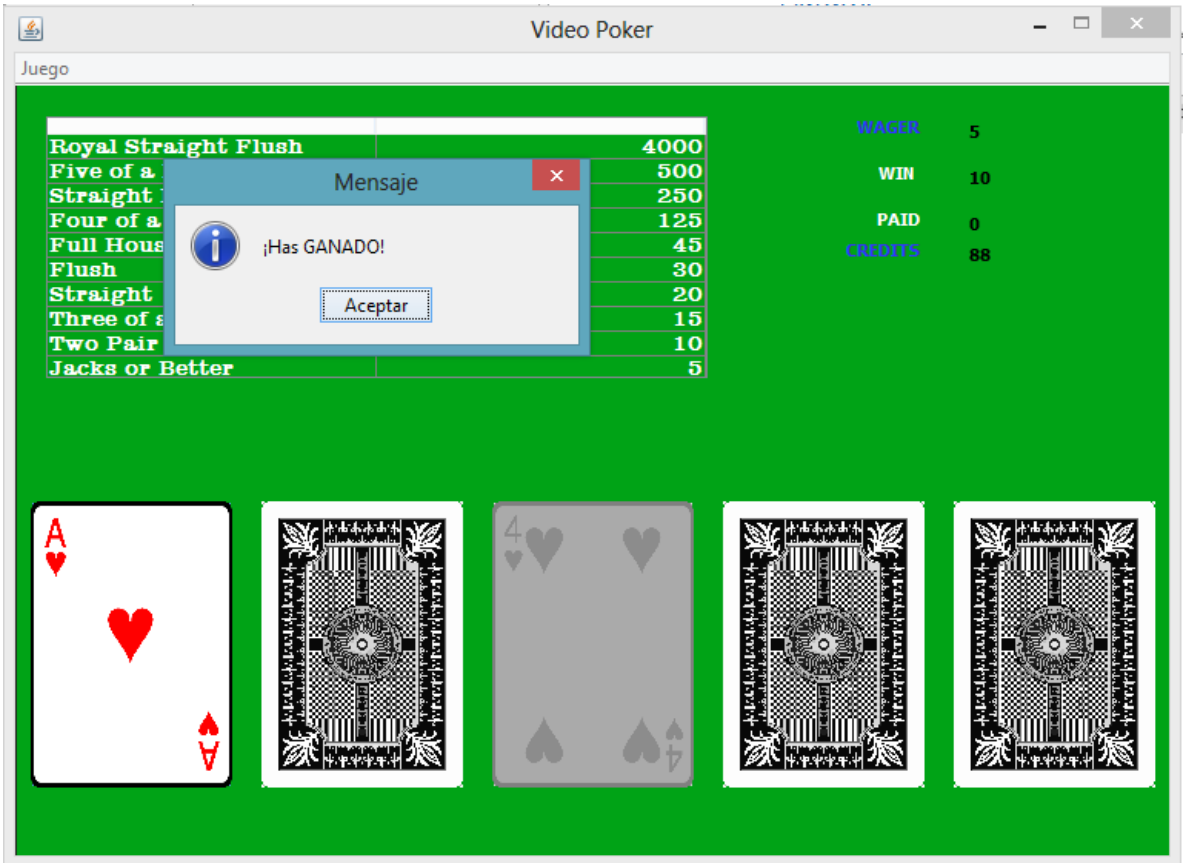



5. Puedes reclamar tu paga o doblarla, ¡Vamos que estamos de suerte hoy! Presiona Double Up para doblar tus ganancias.

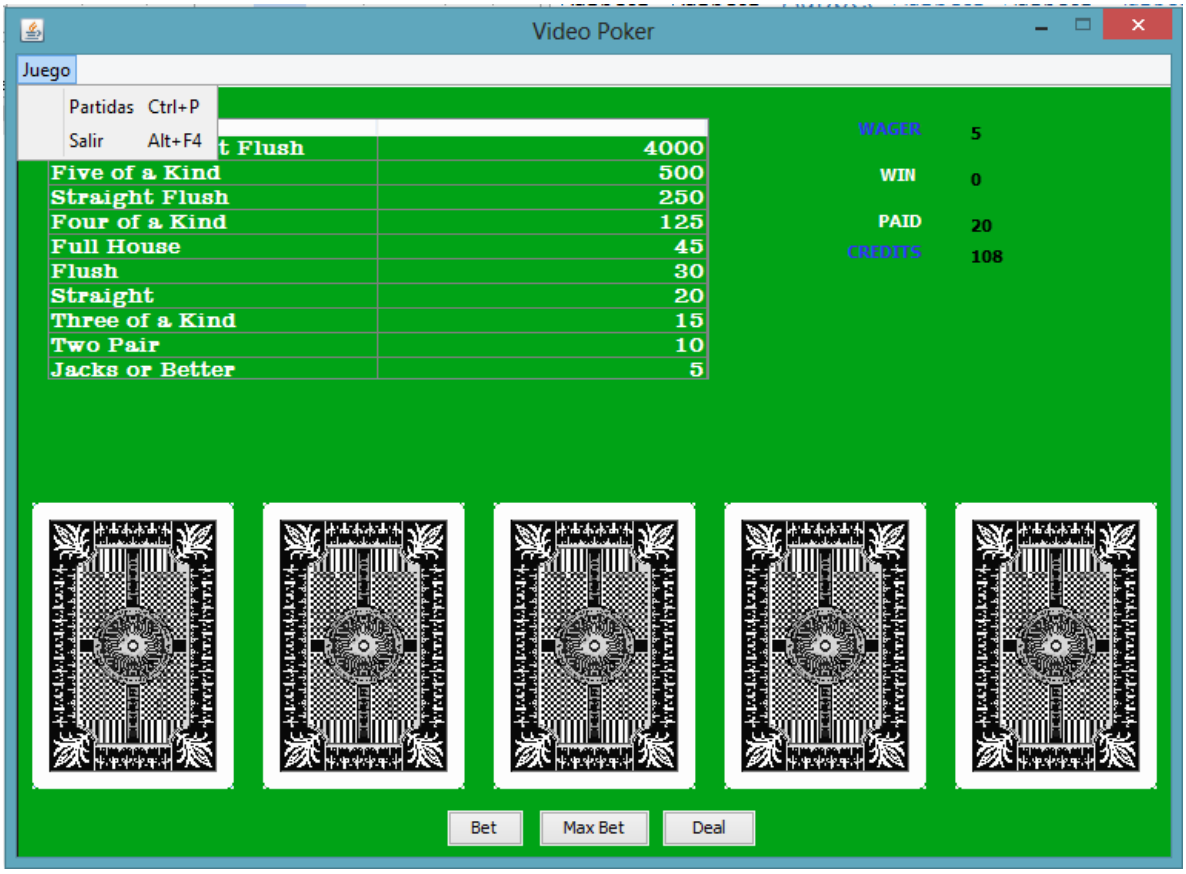

6. Veamos, el Dealer ha escogido una carta… y ha sido un 4 así que para ganar necesitas una carta con un número más alto, presiona alguna carta para probar tu suerte.



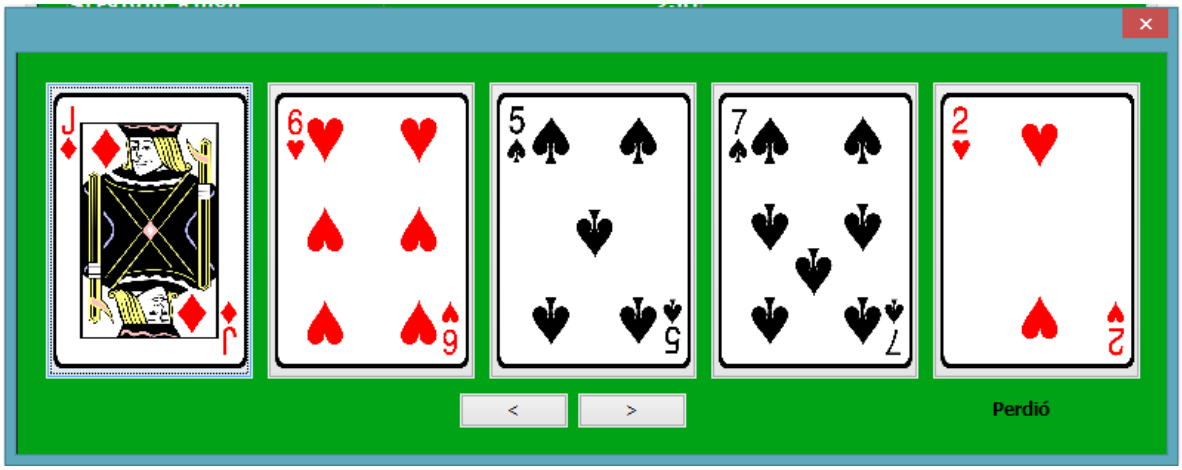

7. ¡Enhorabuena, has sacado un As! Esta vez presiona Aceptar y luego Collect para reclamar tu paga.




8. Si quieres ver tus partidas puedes ir al menú juego y luego partidas, o presonar Ctrl + P.




9. Se despliega una nueva ventana con todas tus partidas. Obsérvalas bien y aprende de ellas.



## Conclusiones

- La dinámica del juego es más fácil de implementarla mediante métodos y eventos que en consola con alguna librería gráfica respectiva, como ncurses.

- Usar TDA's fundamentales como la Lista, Pila y Cola, hacen que el juego tenga una simulación real en las partidas virtuales.

- El rendimiento de ciertos algoritmos primitivos utilizados en este programa hicieron el desarrollo del juego un tanto lento en comparación con los resultados esperados.