PRÁCTICA FINAL: JUEGO DEL 7

PROGRAMACIÓN II

06-2022

Creado por: Luis Miguel Vargas Durán
Mario Ventura Burgos

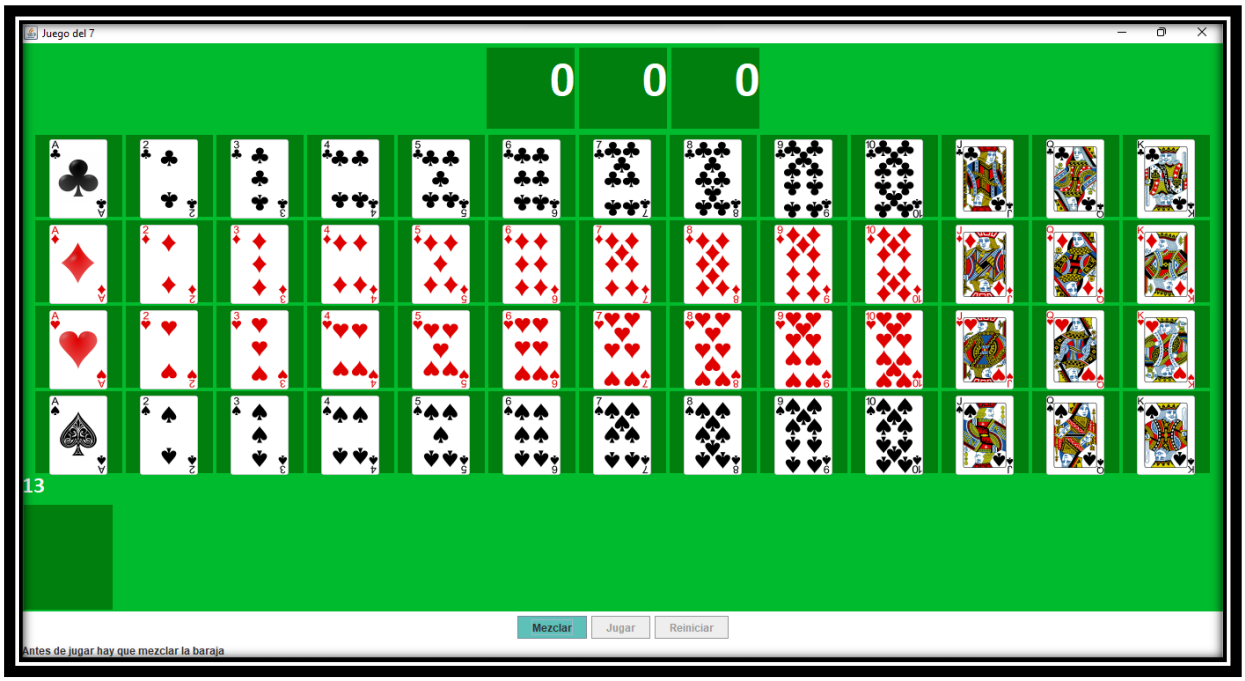
Vídeo: https://youtu.be/tnXSn2-Ue3E

Índice

Introducción

El objetivo de la práctica es programar un juego cartas basado en el conocido juego del siete. La mecánica es muy simple, se juega con una baraja francesa con todas las cartas sin comodines, al principio del juego se reparten todas las cartas entre los jugadores y el juego consiste en desprenderse de todas las cartas en orden, empezando con un 7 y siguiendo con sus vecinos (los del 7, por ejemplo, serian el 6 y el 8). Ganará el jugador que primero termine las cartas.

Inicialmente, el juego muestra lo siguiente:

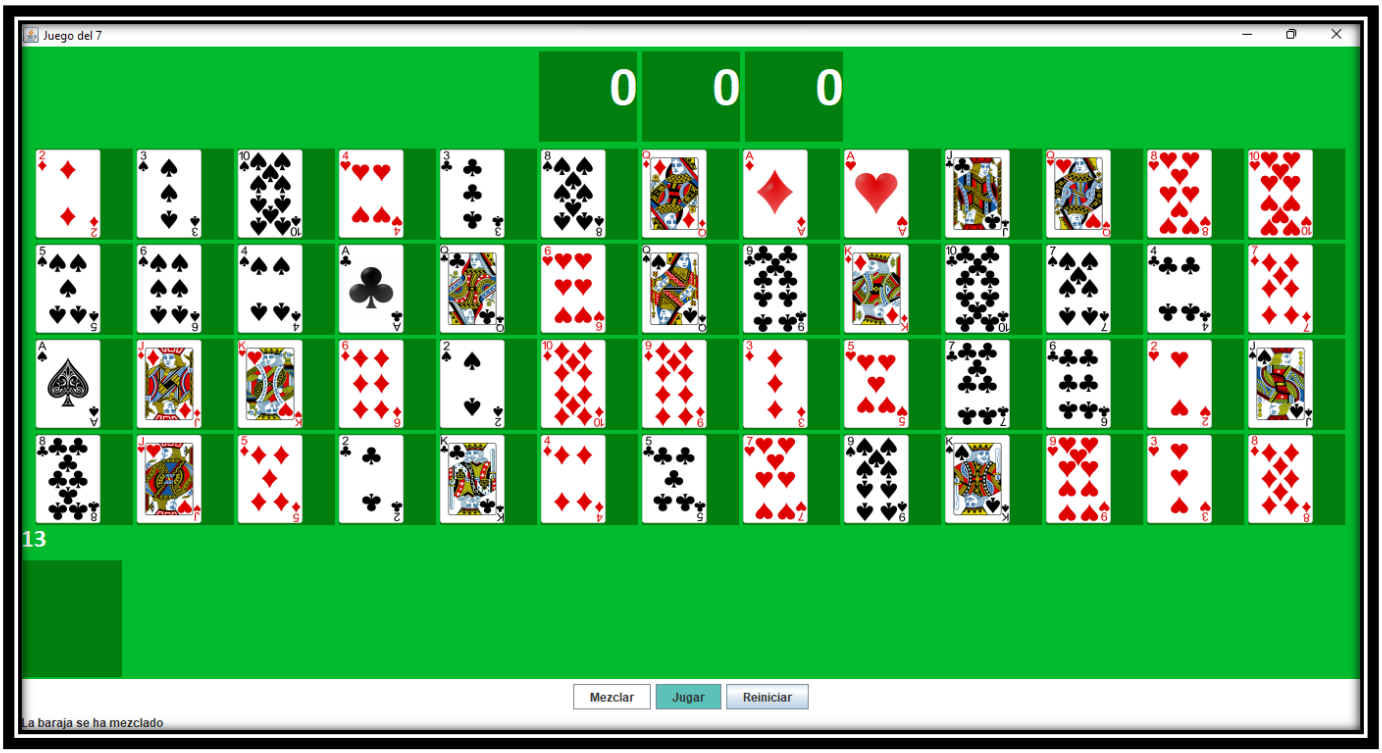


La parte superiro representa los 3 jugadores de la cpu (todos ellos, sin cartas todavía). En el centro, tenemos la mesa de juego, donde inicialmente se coloca la baraja con la que se jugará, que se muestra a todos los jugadores antes de ser mezclada. Es obligatorio mezclar la baraja antes de comenzar a jugar. Esta información (y otras tantas que se enseñan al usuario) aparecen en el panel inferior, que actúa de modo informativo.

También en la zona inferior del juego, bajo la mesa en la que ahora mismo están las cartas, tenemos:

1. Las cartas del usuario (de momento no aparecen ya que no se han repartido todavía.
2. Los botones con los que jugaremos.

Tal y como muestra el panel informativo, se debe mezclar la baraja antes de comenzar a jugar. Una vez lo hagamos, las cartas seguiran apareciendo sobre la mesa, pero mezcladas. Podremos mezclar tantas veces como queramos antes de comenzar la partida.



Una vez pulsemos el botón de jugar, las cartas se repartirán y no se podrán volver a mezclar.

Antes de comenzar el juego, sin embargo, se nos preguntará el nombre. Esto se ha hecho como algo **opcional** que aporta un valor añadido al juego y le añade realismo. El hecho de que todos los jugadores tengan un nombre puede hacer mas “humano” o “realista” este juego.

A los jugadores de la CPU también se les asignará un nombre aleatorio.

Hecho todo esto, la situación es la siguiente:

Las cartas de la zona inferior son las cartas con las que jugará el usuario. En la zona superior, podemos ver como ahora los jugadores de la CPU tienen un nombre, y podemos ver también cuantas cartas tiene cada uno.

A partir de este momento, la CPU estará esperando a que hagamos nuestra tirada (el usuario siempre tira primero) o a que pasemos turno en caso de no poder colocar ninguna carta.

Cada vez que hagamos un turno, se nos bloqueará y se activará la opción de hacer que la CPU haga su turno. De esta forma, el juego transcurrirá de forma en que cada uno haga un turno (o niunguno si se pasa el turno) hasta que alguno de los 4 jugadores se quede sin cartas.

Este será el jugador ganador.

Funcionamiento Interno

La abstracción y la orientación al objeto (POO) han sido vitales para el diseño e implementación de este proyecto.

Respecto a la abstracción, el hecho de descomponer un problema tan complejo a priori, en diferentes partes más pequeñas, fáciles de manejar y resolver; para posteriormente juntarlas y hacer que funcionen en conjunto como piezas de una máquina mayor, ha facilitado el trabajo.

Por otro lado, es evidente la necesidad de crear Objetos.

Echemos un ojo a los diferentes objetos que componen el programa en su totalidad y como se relacionan entre ellos:

1. Objeto Carta
2. Objeto Jugador
3. Objeto Baraja
4. Objeto Mesa
5. Palo (tipo Enumerado)

CARTA:

El objeto carta viene definido por un número y un palo. El palo será un tipo enumerado de Java y hay 4: CLUBS, SPADES, HEARTS y DIAMONDS. Un ejemplo de objeto sería: [7 of Hearts]

BARAJA:

Un conjunto de objetos carta. Tiene cuatro atributos:

1. La cantidad de cartas por palo. En nuestro caso, **MAXCARTASPORPALO = 13**
2. La cantidad de Palos. En nuestro caso, **NUMPALOS = 4**
3. La cantidad de cartas totales (**NUMPALOS x MAXCARTASPORPALO**, es decir, 13x4 = 52 Objetos Carta).
4. Un array de objetos carta, ya que una baraja es un conjunto de ellas de un tamaño fijo (52 en el caso de nuestro programa).

5. La clase baraja, además, tendrá la posibilidad de contemplar una excepción personalizada que saltará cuando se detecte que la baraja no tiene cartas

```
//Clase para crear excepciones personalizadas
public static class barajaVacia extends Exception {
    //Constructor vacío
    public barajaVacia() {

    }
    //Constructor con string
    public barajaVacia(String e) {
        //Llamar a la clase madre de la que se hereda (clase Exception)
        //Pasandole el string recibido como parámetro
        super(e);
    }
}
```

JUGADOR:

Un jugador vendrá definido por:

1. Un nombre
2. Un entero que representa las cartas que tiene el jugador. Si tras un turno este atributo vale 0, significará que el jugador ha tirado sus cartas, y por tanto, ha ganado
3. Un booleano que represente si el jugador es el ganador o no. Inicialmente estará establecido a false.
4. Un array de cartas que representará la lista de las cartas en posesión del Jugador. Cuando una carta se coloque, esta pasará a tener el valor **null** en el array

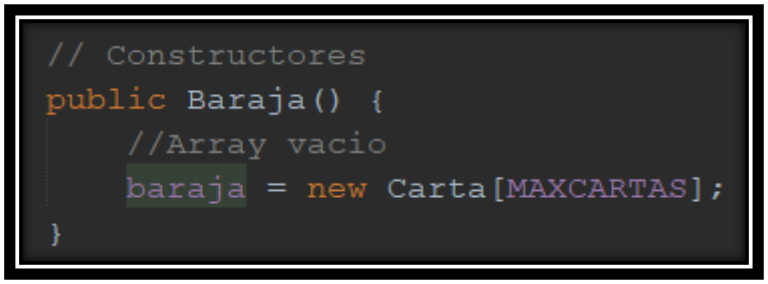
MESA:

Un objeto mesa es una matriz de objetos carta que contendrá el estado de la partida. Vendrá definido por la cantidad de filas y columnas que tendrá la matriz (en nuestro caso, 4 filas y 13 columnas), y un booleano que inicialmente se establecerá a false y se activará poniéndolo a true una vez la partida que representa esta mesa haya acabado.

Ahora que tenemos claro que objetos forman el juego, veamos como se han usado para llegar a la solución planteada.

Los jugadores de la cpu y el usuario son 4 objetos Jugador en total. El nombre del jugador usuario lo introduce este, pero los de la CPU se escogen aleatoriamente entre los dispoibles en un array con 8 nombres posibles. Todos los jugadores serán creados con 0 cartas inicialmente.

Por otro lado, el objeto baraja se crea mediante un constructor que instancia el array de cartas



Posteriormente, se usará un método inicializar() para dar valor a cada carta, ya que estos no deben ser arbitrarios. Debe haber un total de 52 cartas, 13 de cada palo.

El método que se encarga de esto lo hace iterando tantas veces como cartas tiene la Baraja. El código es el siguiente:

```
//Inicializar la baraja creando cartas
  public void inicializar() {
    //Variables auxiliares
    int numero = 1;
    int palo = 1;
    //Inicializar array de cartas
    baraja = new Carta[MAXCARTAS];
    Palo p = null;

    //Dar valor a las cartas
    for (int i = 0; i < MAXCARTAS; i++) {
```

```
if (palo == 1) {
   baraja[i] = new Carta(p.clubs, numero);
   numero++;
   //System.out.println(baraja[i].toString() + " ");
   //Pasar al siguente palo
   if (numero == 14) {
      palo++;       //Siguiente palo
      numero = 1;    //Reiniciar numero
   }
} else if (palo == 2) {
   baraja[i] = new Carta(p.diamonds, numero);
   numero++;
   //Pasar al siguente palo
   if (numero == 14) {
      palo++;       //Siguiente palo
      numero = 1;    //Reiniciar numero
   }
} else if (palo == 3) {
   baraja[i] = new Carta(p.hearts, numero);
   numero++;
   //Pasar al siguente palo
   if (numero == 14) {
      palo++;       //Siguiente palo
      numero = 1;    //Reiniciar numero
   }
} else if (palo == 4) {
   baraja[i] = new Carta(p.spades, numero);
   numero++;
```

```
        if (numero == 14) {
          i = MAXCARTAS;
        }
      }
    }
    //Nmero de cartas de la baraja
    n = MAXCARTAS;
  }
```

De esta forma, como en cada iteración se crea un objeto Carta, se generan 13 cartas de cada palo y, por tanto, una baraja completa que además estará ordenada.

Como se ha mencionado anteriormente, el juego inicialmente muestra la baraja sin haberse mezclado. Esto se hace colocando las cartas de la baraja sobre la mesa para que se vean. Esta expresión en lenguaje natural la aplicaremos "a lo literal" creando e instanciando un objeto mesa donde se colocarán todos los objetos carta que forman el objeto baraja.

En la clase Mesa:

```
public Mesa(int f, int c) {
    filas = f;
    columnas = c;
    cartas = new Carta[f][c];
    acabada = false;
}
```

```
//Inicializar mesa
public void inicializar() {
    for(int i=0; i<filas; i++) {
        for(int j=0; j<columnas; j++) {
            cartas[i][j] = new Carta();
        }
    }
}
```

```
//Añadir una baraja a la mesa
public void añadirBaraja(Baraja b) {
    int idx = 0;
    for(int i=0; i<b.NUMPALOS; i++) {
        for(int j=0; j<b.MAXCARTASPORPALO; j++) {
            cartas[i][j] = b.getCartaAt(idx);
            idx++;
        }
    }
}
```

En el programa principal:

```
//---> Inicializar mesa
mesa = new Mesa(4,13);          //Crear mesa
mesa.inicializar();             //Inicializar mesa
mesa.añadirBaraja(baraja);      //Añade la baraja en la mesa
mesa.imprimirMesa();            //Imprimir (por consola)
```

Hecho esto, la baraja se deberá mezclar mediante el algoritmo de *Fisher-Yates*. Este algoritmo, lógicamente, se encuentra en la clase Baraja y es el siguiente:

```
//Barajar las cartas
public void mezclar() {
    //Barajar las cartas con la implementación
    //Durstenfeld del algoritmo de Fisher-Yates
    Random rnd = new Random();
    //Iteración 'downto' de las cartas para barajarlas todas
    for (int i = MAXCARTAS - 1; i > 0; i--) {
        int pos = rnd.nextInt(i + 1);
        Carta c = baraja[pos];
        baraja[pos] = baraja[i];
        baraja[i] = c;
    }
}
```

Esto se puede hacer cuantas veces se quiera y una vez acabado, las cartas se reparten y la mesa se vacía; por tanto será así como se muestre. Para repartir las cartas, recorremos la baraja mezclada y vamos entregando las cartas a los jugadores dando una a cada uno.

```java
//Repartir las cartas de la baraja hasta que quede vacía
private void repartirCartas() throws barajaVacia {

    //Repartir las cartas a los jugadores de la partida
    for(int i=0; i<baraja.getNumCartes(); i++) {
        jugadoresCPU[0].darCarta(baraja.getCartaAt(i));
        baraja.setCartaAt(i,new Carta());
        i++;
        jugadoresCPU[1].darCarta(baraja.getCartaAt(i));
        baraja.setCartaAt(i,new Carta());
        i++;
        jugadoresCPU[2].darCarta(baraja.getCartaAt(i));
        baraja.setCartaAt(i,new Carta());
        i++;
        player.darCarta(baraja.getCartaAt(i));
        baraja.setCartaAt(i,new Carta());
    }
    System.out.println("\n\n--SE REPARTEN LAS CARTAS Y SE OBTIENE:\n");
    cartasJugadores();
}
```

```java
//Método para dar una carta a un jugador
public void darCarta(Carta c) throws Baraja.barajaVacia {
    //MODIFICAR ATRIBUTO cartas[]
    for(int i=0; i<cartas.length; i++) {
        if(cartas[i] == null) {
            cartas[i] = c;
            i = cartas.length;
        }
    }

    //MODIFICAR ATRIBUTO numCartas
    numCartas++;
}
```

Tras esta operación, cada jugador tendrá 13 cartas totalmente aleatorias y la partida ya puede empezar.

La mesa, actualmente vacía, se llenará de cartas a medida que pasen los turnos del juego.

Cuando un jugador seleccione en su turno una carta que puede ser colocada sobre la mesa, esta carta desaparecerá de la lista de las cartas del usuario y aparecerá sobre la mesa, que al ser una matriz, conservará la información de las cartas que contiene y donde las contiene.

```java
//Colocar cartas
public void colocarCarta(Carta c, int fila, int columna) {
    //Se coloca la carta dada en la posicion indicada
    cartas[fila][columna] = c;
}
```

```java
//Método que imprime las cartas del Jugador
public void printCartas() {
    System.out.println("LAS CARTAS DE "+nombre+" SON:");
    for (int i=0; i<cartasPerPlayer; i++) {
        if(cartas[i].getPal()!=null) {
            System.out.print(cartas[i].toString() + " ");
        }
    }
    System.out.println("\nTOTAL ---> "+numCartas+" Cartas");
}
```

Tras cada turno, la variable entera de la clase jugador que representa la cantidad de cartas de este jugador, disminuirá

Jugador.setNumCartas(jugador.getNumCartas( ) - 1);

A medida que avance el juego, los jugadores cada vez tendrán menos cartas y la mesa cada vez más. Cuando algún jugador llegue a tener 0 cartas pasará lo siguiente:

1. Su atributo booleano **ganador** se establecerá a true para representar que este jugador ha ganado.
2. El atributo booleano **acabada** de la clase mesa se establecerá a true para indicar que la partida jugada sobre esa mesa está acabada.
3. El programa principal al detectar que algún jugador tiene el atributo **ganador** con valor true, lanza un aviso al usuario diciendo que este jugador ha ganado la partida, y que esta ha acabado. Este aviso se lanza mediante una ventana emergente, pero la cuestión gráfica la trataremos posteriormente.
4. Tras avisar al usuario de que la partida ha acabado, se le pregunta si quiere reiniciar y jugar otra partida o salir del juego
5. Por último se hace lo que el usuario haya escogido en la decisión anterior.

Como último tema a tratar antes de pasar a la parte gráfica del programa, es útil entender cómo se sabe si una carta puede ser colocada sobre la mesa.

Para poder colocar una carta sobre la mesa, se debe cumplir alguna de las siguientes posibilidades:

1. Si es el primer turno, solo se pueden colocar cartas cuyo número sea el 7, independientemente del palo
2. Tras el primer turno, si sobre la mesa hay una carta que tiene palo P y número N, se podrán colocar cartas con palo P y número N-1 o N+1 (es decir, las cartas vecinas)

La clase Mesa tiene un método llamado **sePuedeColocar()** que recibe un objeto Carta. Este método comprueba si se puede colocar la carta recibida dado el estado actual de la mesa. En caso de que se pueda, colocará la carta y devolverá true al método que lo llamó, para que este sepa que se ha colocado la carta sobre la mesa, y que, por tanto, se debe hacer lo siguiente:

1. Disminuir en 1 la cantidad de cartas del usuario
2. Establecer a null la carta del jugador, para que no siga estando en posesión de una carta que en realidad está en la mesa

Cada vez que se hace un turno, cada jugador hace uso de este método para saber si se puede colocar la carta seleccionada. Los métodos que simulan los turnos reciben como parametros un booleano que representa si se trata del primer turno o no, el objeto mesa sobre el que se está jugando, y el objeto carta que se quiere colocar sobre esta mesa.

Además, son métodos que devuelven un booleano que representa:

- **True:** El jugador ha colocado una carta en la mesa.
- **False:** El jugador no ha podido colocar la carta.

Tras cada turno, se comprueba el valor de retorno del método que simula este turno. Si es true, querrá decir que el jugador ha colocado una carta, cosa que implica que su cantidad de cartas habrá disminuido. Tendremos que comprobar, por tanto, si la cantidad de cartas de este usuario es 0 tras el turno, ya que de ser así, este último turno habría sido la tirada ganadora del jugador.

El código es el siguiente:

```
//Hacer tirada
if (player.turnoManual(primerTurno, mesa, c)) {
    s = (player.getNombre()+" ha colocado "+c.toString());
    player.setCartaAt(i,new Carta());
    playerMovement = true;
} else {
    s = "Esa carta no puede colocarse, escoge una válida";
}
```

---- Código del método turnoManual() ----

```
//Método que simula un turno manual del usuario
  public boolean turnoManual(boolean primerTurno, Mesa m, Carta c) {
    //SIMULA EL TURNO DE UN JUGADOR (típicamente, el usuario)
    //RECIBE: Booleano que indica si es el primer turno, Mesa sobre la que se juega,
Carta que se quiere poner
    //DEVUELVE: True si tras el turno el jugador ha hecho un movimiento

    ganador = false;
    boolean accion = false;

    //---< Primer Turno >---
    if (primerTurno == true) {
      //Comprobar si la carta es un 7 de cualquier palo
      if (c.getNum() == 7) {
        //Colocar la carta en la mesa
        posicionCarta = c.getPosicionMesa();
        m.colocarCarta(c, posicionCarta, 6);
        System.out.println(nombre + " HA COLOCADO LA CARTA " + c.toString() + "
EN LA MESA POSICION 6," + posicionCarta);
        numCartas--;
        cartaColocada = c;
        accion = true;
      }

    //---< Otros Turnos >---
    } else {
      //Recorrer las cartas y comprobar si se pueden colocar
        if (m.sePuedeColocar(c)) {
          //Se ha colocado la carta
          System.out.println(nombre+" HA COLOCADO LA CARTA "+c.toString());
          numCartas--;
```

```
            cartaColocada = c;
            c = new Carta();
            accion = true;
          }
      }

      //Comprobar si el jugador es el ganador, es decir, si ya no tiene cartas
      if(numCartas == 0) {
        m.setEstado(true);  //El estado de la mesa se declara como acabada
        ganador = true;     //EL jugador pasa a ser el ganador
      }

      //Imprimir mesa tras el turno
      if (accion == false) {
        //El jugador no podia colocar carta y ha pasado turno
        System.out.println(nombre+" HA PASADO SU TURNO PORQUE NO PUEDE
COLOCAR NINGUNA CARTA\n");
      } else {
        //El jugador colocó carta y el estado de la mesa ha cambiado
        m.imprimirMesa();
      }

      //Retorno
      return accion;
    }
```

Cabe mencionar también, el uso de los bloques try-catch para gestionar posibles excepciones a lo largo de la ejecución del programa.

En nuestro caso, hemos optado por hacer que las excepciones se gestionen todas en el mismo método: el método principal. De esta manera, todos los métodos que son llamados desde este, lo que harán será lanzar estas excepciones al nivel superior para que sean gestionadas como se requiera.

Esto evitará usar bloques try catch constantemente. Los usaremos estratégicamente una vez, en el método donde todas excepciones serán lanzadas (incluso las excepciones personalizadas como *barajaVacia*).

Aspectos Gráficos

Respecto al uso de los gráficos, lo que se ha hecho ha sido representar mediante paneles, componentes, etiquetas, etc… el transcurso de la partida.

La aplicación se abre en una ventana *JFrame* sobre la que se insertan 2 paneles:

1. Panel *JPanel* que simula la mesa sobre la que se juega: panelMesa
2. Panel *JPanel* inferior que muestra la información: panelInformacion

La ventana usa un administrador de Layout de tipo *BorderLayout*, que se usa para insertar el panel panelMesa en *BorderLayout.CENTER* y panelInformacion en *BorderLayout.SOUTH.*

panelMesa:

Además, el panel panelMesa, que también usa un BorderLayout, contiene:

1. **NORTE:** Panel que muestra los 3 jugadores de la CPU. Usa un *FlowLayout*
2. **CENTRO:** Panel que simula gráficamente un objeto Mesa. Este panel usa un *GridLayout* con 4 filas y 13 columnas para representar en forma matricual las cartas que se colocarán a lo largo de la partida. Cada uno de los elementos de este panel es otro panel, que a su vez tiene una etiqueta cuyo icono ImageIcon es la imagen de la carta.
3. **SUR:** Panel que muestra el nombre del usuario y la cantidad de cartas que tiene, y que también muestra estas cartas. Para mostrar las cartas se hace lo mismo que en el panel central, usando un GridLayout pero de 1 fila y 13 columnas. De esta forma, se visualizan las 13 cartas del usuario.

   Además, encima de cada panel que muestre una carta del usuario, se añadirá un botón *JButton* invisible, para poder detectar que carta ha clicado el usuario y poderla pasar así como parámetro al método que simula una tirada.

   Cuando una carta se coloca sobre la mesa, se elimina de la lista de cartas del usuario haciendo que la imagen de esta desaparezca. De esta manera, de forma gráfica, el usuario ve como una de sus cartas desaparece de la lista de cartas disponibles, y aparece situada en la posición correspondiente sobre la mesa (panel descrito anteriormente)

Si la carta clicada no puede ser colocada, se informará al usuario imprimiendo un aviso en la etiqueta informativa del panel panelInformacion que explicaremos a continuación:

panelInformacion:

En este panel también se usa un BorderLayout para poder añadir otros 2 paneles de la siguiente forma:

1. **CENTRO:** Panel que contendrá los botones para mezclar la baraja, simular un turno, reiniciar la partida, etc…

   El panel de botones será un panel que hará uso de un *FlowLayout*, para que los botones que se le añadan aparezcan siempre centrados en el panel.
2. **SUR:** Panel que contendrá una etiqueta *JLabel* donde se imprimirán mensajes para dar información e indicaciones al usuario

Todos estos paneles se añadirán a la ventana mediante las sentencias siguientes:

```
ventana.getContentPane().add(panelMesa,BorderLayout.CENTER);
ventana.getContentPane().add(panelInformacion,BorderLayout.SOUTH);
ventana.setSize(1000,650);
ventana.setVisible(true);
ventana.setBackground(colorPoker);
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Por último, cuando el usuario escoja reiniciar la partida (pulsando el botón reiniciar o escogiendo la opción al acabar una partida), todas las variables se reestablecerán y los componentes gráficos se reiniciarán. Cualquier carta situada en la mesa, cualquier mensaje en la etiqueta informativa, etc… volverá a tener el valor inicial y se llamará nuevamente al método que crea el entorno gráfico. De esta forma el programa se deshace de cualquier información anterior y crea una partida nueva. A lo largo del programa se ha hecho uso de librerías de java como:

1. import java.awt.Font para modificar la Fuente
2. import java.awt.Color para crear colores como la constante colorPoker
3. import java.util.Random para crear números aleatorios

entre otras tantas.

```
//---< MÉTODO PARA CREAR UNA INTERFAZ GRÁFICA >---
   //Método que crea una interfaz gráfica
   private void graficos() {

      //-----> Instanciar variables gráficas
      if(reiniciar == false) {
         //Es la 1ª vez
         ventana = new JFrame("\tJuego del 7");
      }

      ventana.setLayout(new BorderLayout());
      //---Zona superior de la mesa---
      //Paneles
      panelMarcador = new JPanel();
      panelMarcador.setLayout(new FlowLayout());
      panelMarcador.setBackground(colorPoker);
      panelJugador1 = new JPanel();
      panelJugador1.setBackground(colorVerdeOscuro);
      panelJugador1.setLayout(new BorderLayout());
      panelJugador2 = new JPanel();
      panelJugador2.setBackground(colorVerdeOscuro);
      panelJugador2.setLayout(new BorderLayout());
      panelJugador3 = new JPanel();
      panelJugador3.setBackground(colorVerdeOscuro);
      panelJugador3.setLayout(new BorderLayout());
      //Jugador 1
      etiquetaCartasJug1 = new JLabel();
      etiquetaCartasJug1.setBounds(40,40,50,50);
      etiquetaCartasJug1.setIcon(new
ImageIcon(imagen.getImage().getScaledInstance(68,92,Image.SCALE_SMOOTH)));
      etiquetaCartasJug1.setHorizontalAlignment(SwingConstants.CENTER);
      etiquetaCartasJug1.setVerticalAlignment(SwingConstants.CENTER);
      etiquetaCartasJug1.setFont(new Font("Calibri",Font.BOLD,55));
      etiquetaCartasJug1.setForeground(Color.WHITE);
      etiquetaCartasJug1.setText("0");
      //Jugador 2
      etiquetaCartasJug2 = new JLabel();
      etiquetaCartasJug2.setBounds(40,80,50,50);
```

.
.
.

Conclusiones

Para finalizar podemos destacar 4 puntos los cuales consideramos que son los que hacen que esta práctica sea particular:

- La importancia en utilizar las organizaciones matriciales para, por ejemplo, crear el objeto mesa, que es una matriz de cobjetos carta
- Contemplar excepciones personalizadas para poder contemplar cosas tan concretas como que una baraja no tenga cartas
- Práctica completamente orientada al objeto, hemos utilizado 4 clases, 1 enum y el programa o clase principal (main)
- Uso de la abstracción: creación de muchos métodos diferentes que funcionan de forma conjunta para un propósito general

Por último, cabe recalcar, las utilidades que nos ofrecen los gráficos y lo potentes que pueden llegar a ser en la resolución de este tipo de prácticas o problemas. También como estos pueden ser usados para mostrar gráficamente estructuras de datos como la matriz de cartas, donde en realiad esta simula la mesa.