ICC311
Estructuras de Datos

Semestre I, 2020

Semana 03 - Parte 00

Ayudantía

1. Problema mazo de cartas

Contexto problema

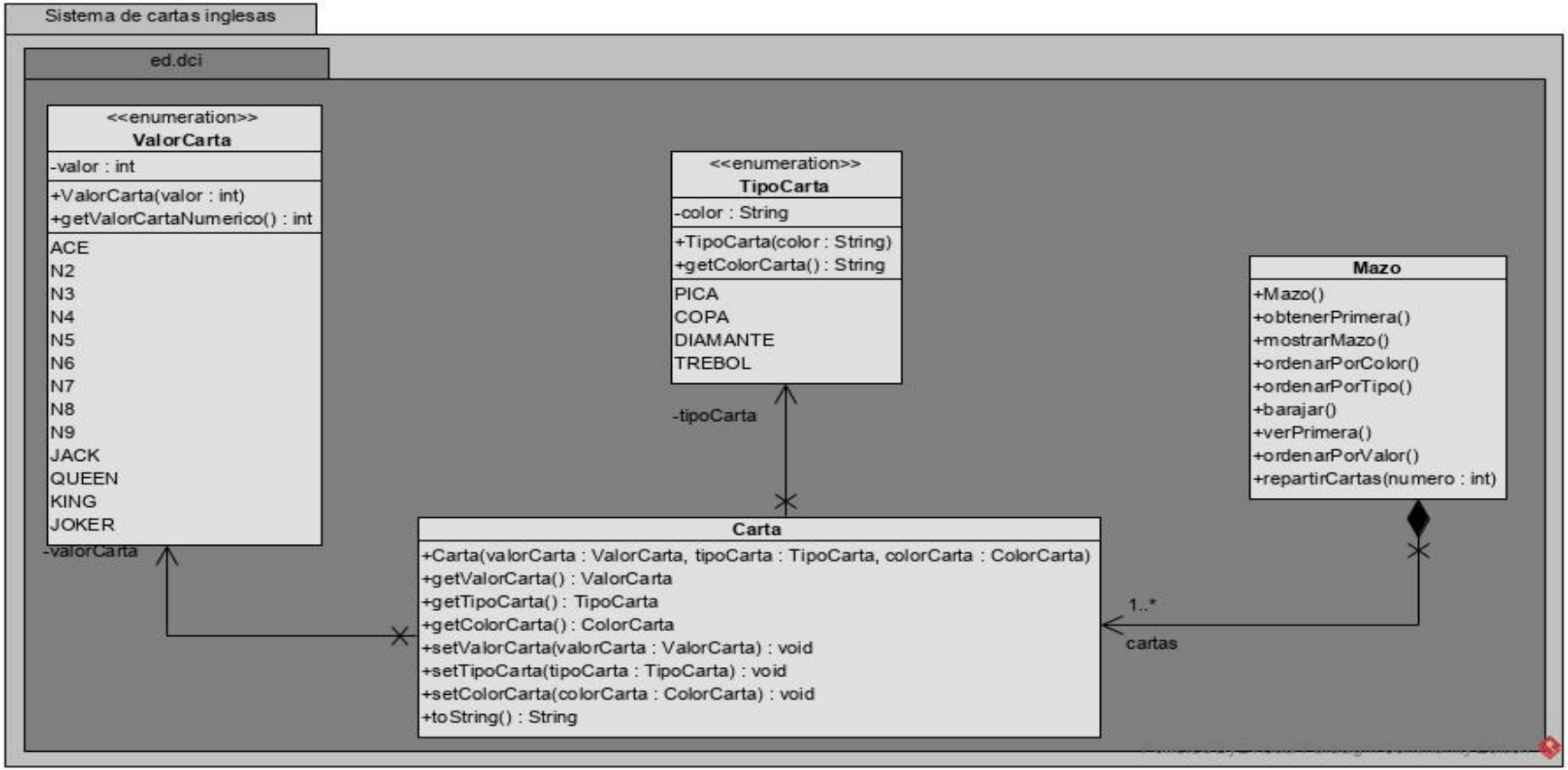
Problema Mazo de Cartas

¿Cómo representar efectivamente un mazo de cartas del naipe inglés en código?. Para implementar la representación de cartas del naipe inglés en código, utilizaremos como contexto el juego Carioca. Entonces, un mazo de cartas debería tener 52 cartas distintas donde cada una de ellas puede ser completamente representada por su tipo (Corazones ❤️, Pica ♠️, Diamante ♦️, Trébol ♣️) y su rango (Ace, 2, 3, ..., 10, Jack, Queen, King). Una estructura de datos que permita representar un mazo de cartas debería ser capaz de representar cualquier secuencia y cualquier número de cartas distintas entre 0 y 52. La siguiente imagen, expone la representación de una partida de juegos con un mazo de cartas central y cuatro jugadores que cuentan con 12 cartas cada uno.

1. ¿Cómo representar una carta?
2. ¿Cómo representar una mazo de cartas?
3. ¿Cómo representar las cartas de un jugador?
4. Si el mazo de cartas incluye cierta cantidad de cartas que están boca abajo sobre la mesa e incluye una carta que está expuesta en sentido contrario (como el ace). ¿Qué debería modificar/agregar a su representación actual de mazo de cartas?

Diagrama de clases

Diagrama de clases



Implementación

Implementación

```
public enum TipoCarta {
    PICA( color: "NEGRA"),
    CORAZON( color: "ROJA"),
    DIAMANTE( color: "ROJA"),
    TREBOL( color: "NEGRA");

    private String color;

    TipoCarta(String color) { this.color = color; }
    public String getColor() { return this.color; }

    public String getTipoCarta() { return this.name(); }
```

Implementación

```java
public enum ValorCarta {
    JOKER(0),
    ACE(1),
    N2(2),
    N3(3),
    N4(4),
    N5(5),
    N6(6),
    N7(7),
    N8(8),
    N9(9),
    JACK(10),
    QUEEN(11),
    KING(12);


    private int valor;
    ValorCarta(int valor) { this.valor = valor; }
```

Se agrega el valor numérico para poder ordenarlos fácilmente.
Se agrega N por qué un enam no acepta números.

Implementación

La carta está definida por ambos enum (tipo, color y valor de carta).

```
public Carta(TipoCarta tipoCarta, ValorCarta valorCarta) {
    this.tipoCarta = tipoCarta;
    this.valorCarta = valorCarta;
}
```

Implementación

```
private Stack<Carta> cartas;

public Mazo(){
    this.cartas = new Stack();
    TipoCarta tipos[] = TipoCarta.values();
    ValorCarta valoresCartas[] = ValorCarta.values();
    for(TipoCarta tipo:tipos){
        for(ValorCarta valorCarta:valoresCartas){
            this.cartas.push(new Carta(tipo,valorCarta));
        }
    }
}
```

Se utiliza una pila (Stack) debido a las operaciones que se utilizan (sacar de la cabeza).

En el constructor se cargan todas las cartas del mazo.

Implementación

Un objeto Iterator permite remover objetos de Collections, puedes moverte adelante o atrás con previous() y next(), funciona en cualquier tipo de colección y dependiendo de la arquitectura puede funcionar más rápido que un loop normal.

```
public void mostrarMazo(){
    Iterator<Carta> it = cartas.iterator();
    while(it.hasNext()){
        System.out.println(it.next().toString());
    }
}
```

Implementación

pop() elimina la primera carta, peek() muestra la primera carta y shuffle mezcla los elementos de la pila (o cualquier coleccion que se le entregue).

```
public Carta obtenerPrimera(){ return this.cartas.pop(); }

public Carta verPrimera() { return this.cartas.peek(); }

public void barajar() { Collections.shuffle(this.cartas); }
```

Implementación

Al momento de ordenar existen muchas opciones, la primera es utilizar sort de Collections y declarar un objeto Comparator y sobreescribir el método compare que debe entregar: 1 si el primer elemento es mayor, -1 si no es mayor y 0 si son iguales (así funciona compareTo). Ventaja alta adaptabilidad, desventaja desordenada.

```
public void ordenarPorColorOP1(){
    //Collections.sort(this.cartas);  //descomentar para ejemplificar como funciona implem
    Collections.sort(this.cartas,new Comparator<Carta>() {
        @Override
        public int compare(Carta c1, Carta c2) {
            return c1.getTipoCarta().getColor().compareTo(c2.getTipoCarta().getColor());
        }
    });
}
```

Implementación

Segunda opción, ordenar utilizando el sort de la lista y una función lambda, que debe seguir las mismas reglas de retorno que la opción anterior. Altamente optimizable, pero con limitaciones, más ordenada que la anterior.

```
public void ordenarPorColorOP2(){//lambda
    this.cartas.sort((c1,c2)->
            c1.getTipoCarta().getColor().compareTo(c2.getTipoCarta().getColor()));
}
```

Implementación

Implementar la interfaz Comparable, mucho más ordenado pero solo permite un tipo de comparación que utiliza como default.

```
public class Carta implements Comparable<Carta>{
```

```
@Override
public int compareTo(Carta o) { //decir que el problema de utilizar esta aprox
    return this.tipoCarta.getColor().compareTo(o.getTipoCarta().getColor());
}
```

```
Collections.sort(this.cartas);
```

Implementación

También se pueden declarar comparadores estáticos en la clase para poder comparar distintos tipos de datos (de ser necesario).

```
public static Comparator<Carta> ValorComparator
        = new Comparator<Carta>() {
    @Override
    public int compare(Carta o1, Carta o2) {
            int uno = o1.getValorCarta().getValorNumerico();
            int dos = o2.getValorCarta().getValorNumerico();
            if(uno>dos){
                return 1;
            }else{
                if(uno==dos){
                    return 0;
                }else {
                    return -1;
                }
            }
```

```
public void ordenarPorValor() {// menor a mayor, dar vuel
    Collections.sort(this.cartas,Carta.ValorComparator);/
}
```

Implementación

Idea de cómo repartir cartas. ¿cómo hacer que dependa de la cantidad de jugadores o la cantidad de cartas que quedan? ¿cómo devolver las cartas que quedan al mazo? ¿como enlazar las cartas a los jugadores?

```
public Stack<Carta> repartirCartas(int numeroCartas){
    Stack<Carta> mano = new Stack<>();
    for(int i=0;i<numeroCartas;i++){
        mano.push(this.cartas.pop());
    }
    return mano;
}
```

ICC311
Estructuras de Datos