FACULTAD DE INGENIERIA
Universidad de Buenos Aires

# Trabajo Práctico 2 — Balatro

[7507/9502] Algoritmos y Programación III
Grupo UBAlatro
Segundo cuatrimestre de 2024


| Alumno: | Número de Padrón | Email |
|---|---|---|
| RUIZ DIAZ Carolina | 111442 | cruiz@fi.uba.ar |
| SPARTA Mateo | 109418 | msparta@fi.uba.ar |
| TREBEJO CRUZ Luis | 109988 | ltrebejo@fi.uba.ar |
| PIZZI Mariana | 110280 | mpizzi@fi.uba.ar |

# Índice

# 1. Introducción

El presente informe reune la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación de un sistema de Juego llamado Balatro utilizando un lenguaje de tipado estático (Java) con un diseño del modelo orientado a objetos y trabajando con las técnicas de TDD e Integración Continua.

# 2. Supuestos

- Puntaje: Las cartas de poker, cartas de tipo tarot y comodines tienen un puntaje propio, es decir, una instancia de la clase `Puntaje`, tiendo esta los atributos `puntos` y `multiplicador`. Cada entidad aporta al cálculo del puntaje final de la jugada según su comportamiento. La manera en la que estos puntajes se acumulan al calcular cuánto vale la jugada es: los puntos se suman entre sí y los multiplicadores se multiplican. Entonces, si por ejemplo se juega Par (que tiene un puntaje de `10*2`) con dos Ases (que tienen un puntaje de `11*1`), el cálculo que se realiza es `(10+11+11)*(2*1*1)`.
- Tarot: Para las cartas de tipo tarot decidimos que la forma de alterar el puntaje de la carta sea incrementándolo en lugar de establecer un puntaje fijo, haciendo que sea acumulable. Por ejemplo, si se tiene un As de picas, cuyo puntaje por defecto es `11*1` y usamos una carta de tipo tarot con puntaje `4*2`, entonces luego de aplicarse el efecto la carta de poker pasa a tener un puntaje de `15*2`.
- Descartes: La manera de modelar los descartes fue haciendo que estos sean un tipo de jugada, el cual por defecto tiene un puntaje `0*1`, pero puede ser modificado por un comodin. De esa manera si el jugador tiene un comodin cuya activación sean los descartes, al efectuarlo la ronda recibe el puntaje correspondiente al bonus del comodin. Un ejemplo de esto ocurre con el comodin "Cumbre Mística", el cual hace que los descartes pasen a tener un puntaje de `1*15`.

# 3. Modelo de dominio

Este proyecto tiene como objetivo replicar el juego de cartas "Balatro", implementando un sistema que simula su funcionamiento básico y presenta una experiencia estructurada en 8 rondas. Durante estas, el jugador debe formar jugadas de póker utilizando cartas seleccionadas de su mano, con el propósito de alcanzar un puntaje objetivo que le permita avanzar en el juego.

En cada ronda, el jugador dispone de ocho cartas en su mano. Puede seleccionar hasta cinco de ellas para jugar o descartar, y tras realizar esta acción, repone las cartas faltantes desde su mazo. Cada jugada realizada tiene un puntaje base, que puede incrementarse utilizando comodines y cartas de tarot estratégicamente. El objetivo principal es alcanzar el puntaje requerido antes de agotar el número de movimientos disponibles en la ronda. Si el jugador logra superar este desafío, avanza a la siguiente ronda. El juego finaliza con éxito si se alcanza el puntaje objetivo en la última ronda; de lo contrario, se considera una derrota.

La solución se estructuró en torno a clases que representan los principales elementos del juego. Entre ellas se destacan: las **Cartas** (`CartaPoker`), que encapsulan el valor y el palo de cada carta; las **Manos** (`Mano`), que gestionan las cartas disponibles para el jugador; el **Mazo** (`Mazo`), responsable de contener y distribuir las cartas; y las **Jugadas** (`Jugada`), que calculan los puntajes. Además, se diseñó la clase **Ronda** (`Ronda`), encargada de manejar la lógica de cada etapa del juego.

El desarrollo se realizó utilizando el enfoque de **Desarrollo Guiado por Pruebas (TDD)** con **JUnit**, lo que permitió construir el sistema de forma iterativa. Asimismo, la interfaz gráfica fue implementada utilizando **JavaFX**, proporcionando una experiencia de usuario visualmente atractiva y funcional.

# 4. Diagramas de clase

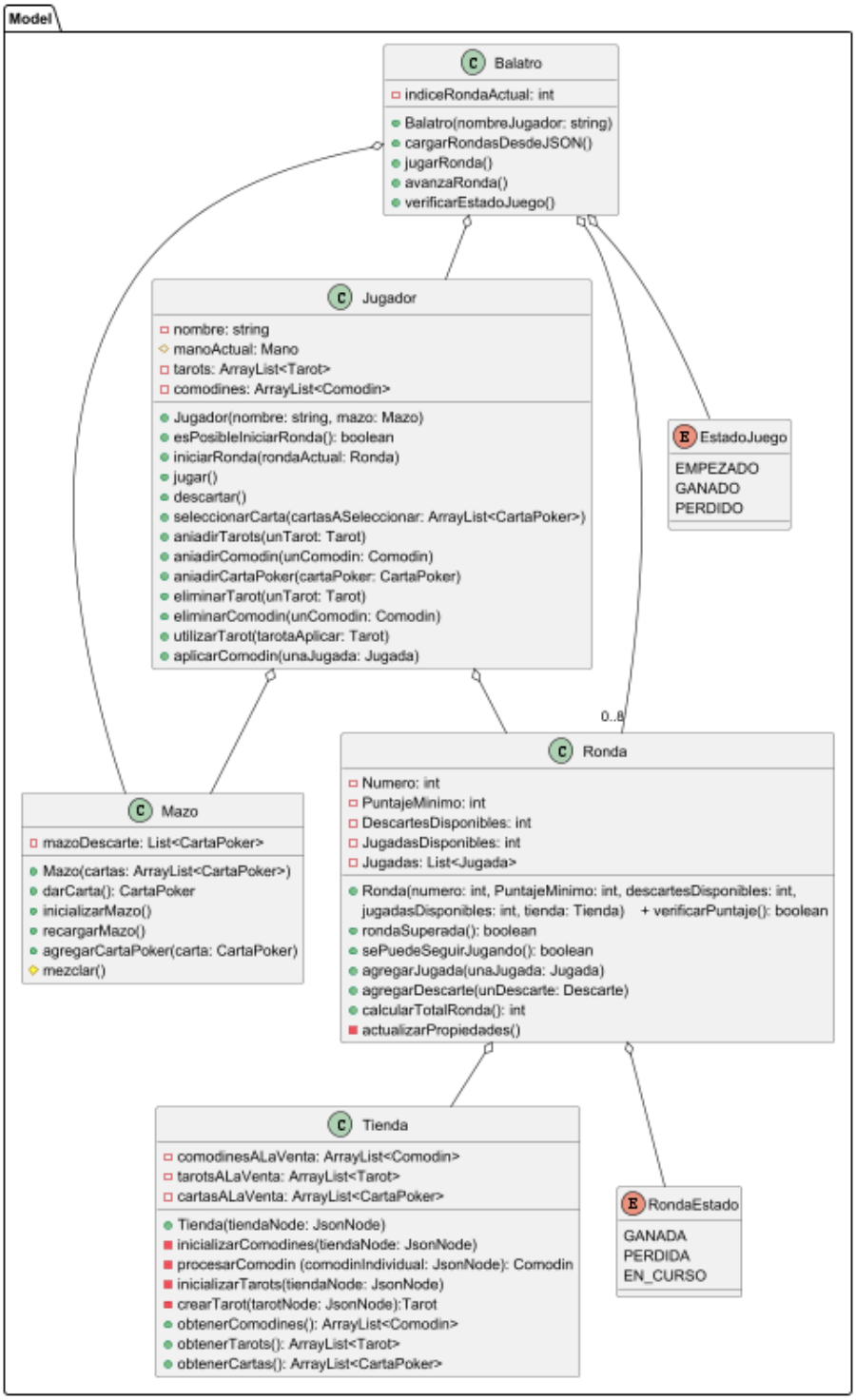


Figura 1: Estrcutura general del juego

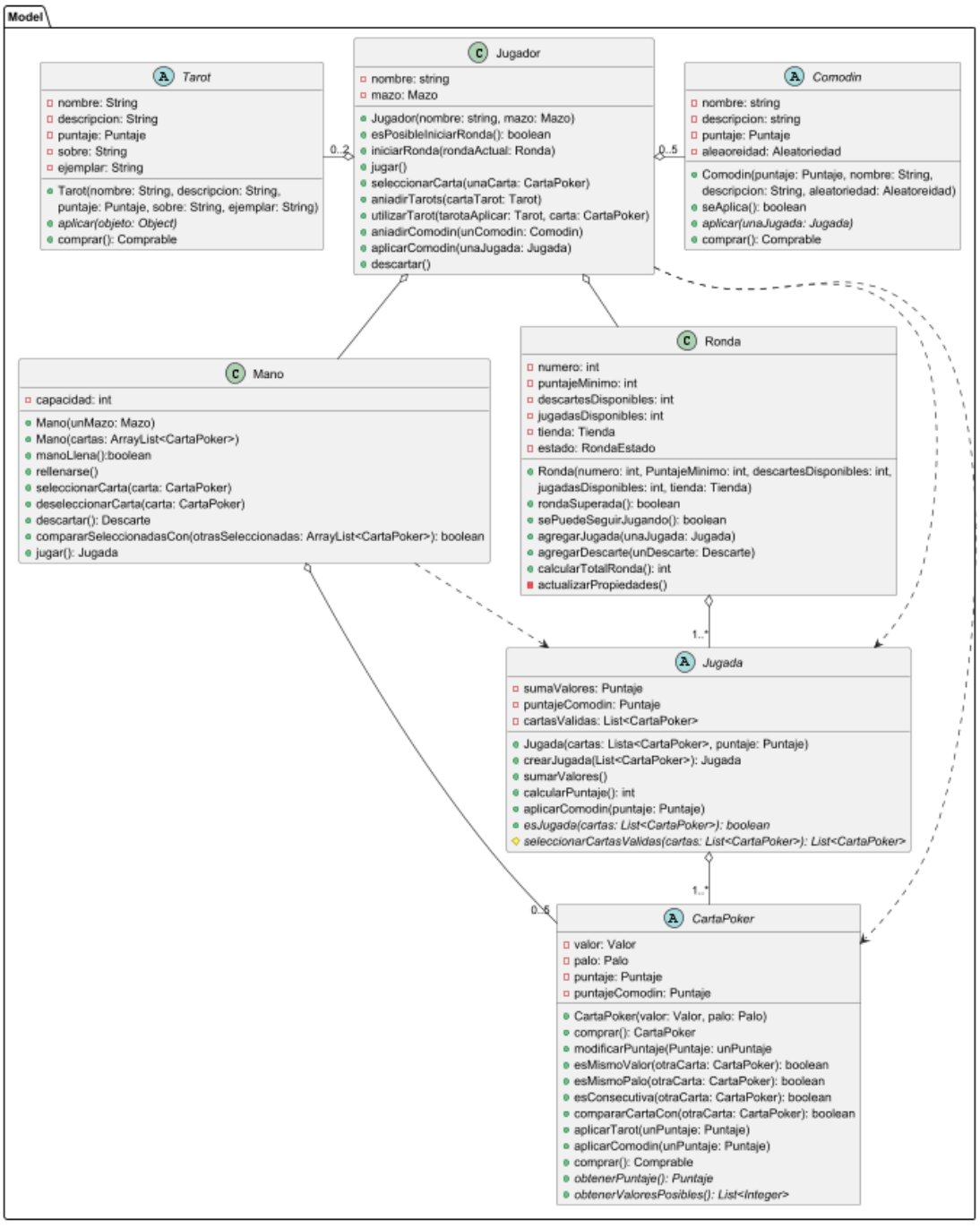


Figura 2: Relación entre el jugador y las demás entidades

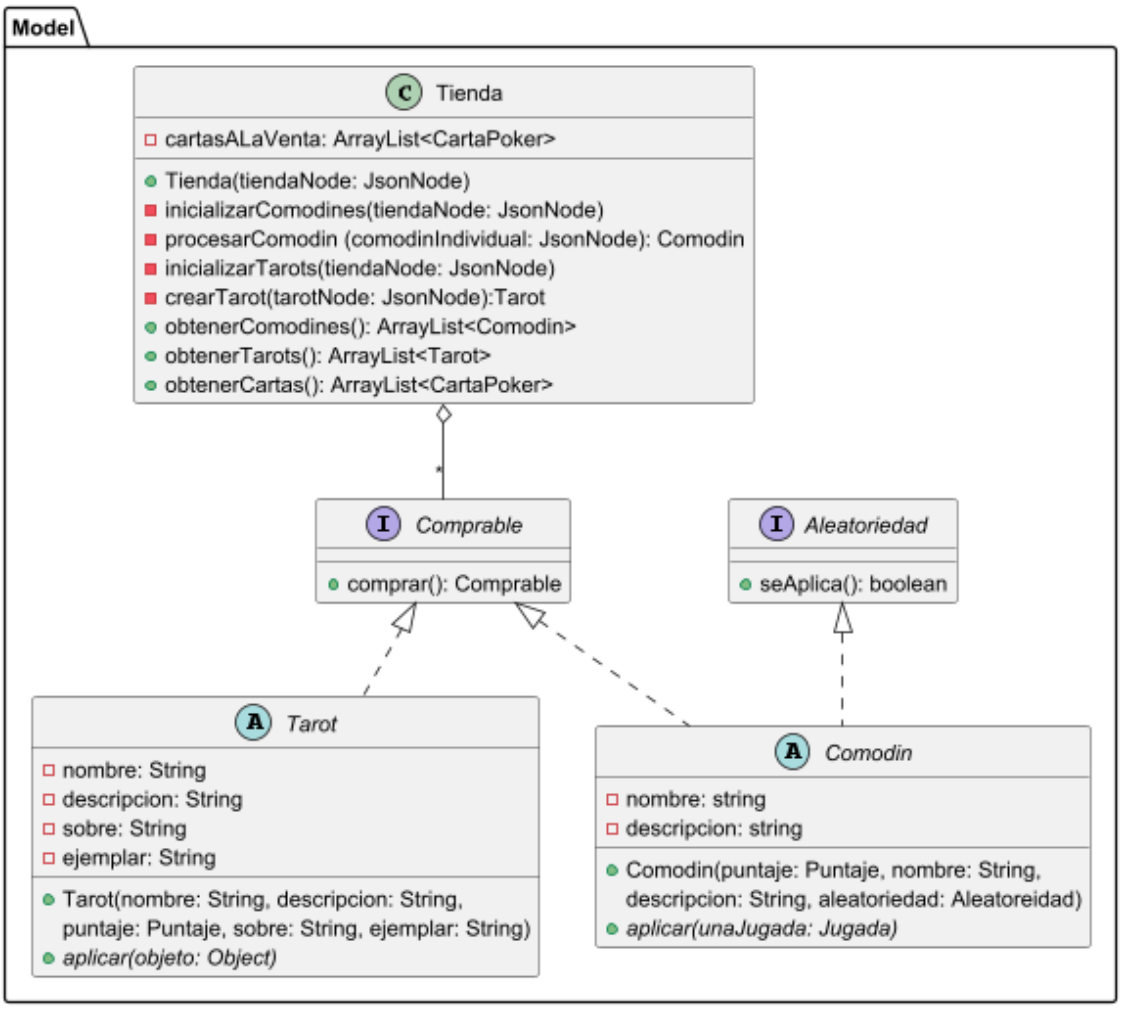


Figura 3: Modelado de la tienda

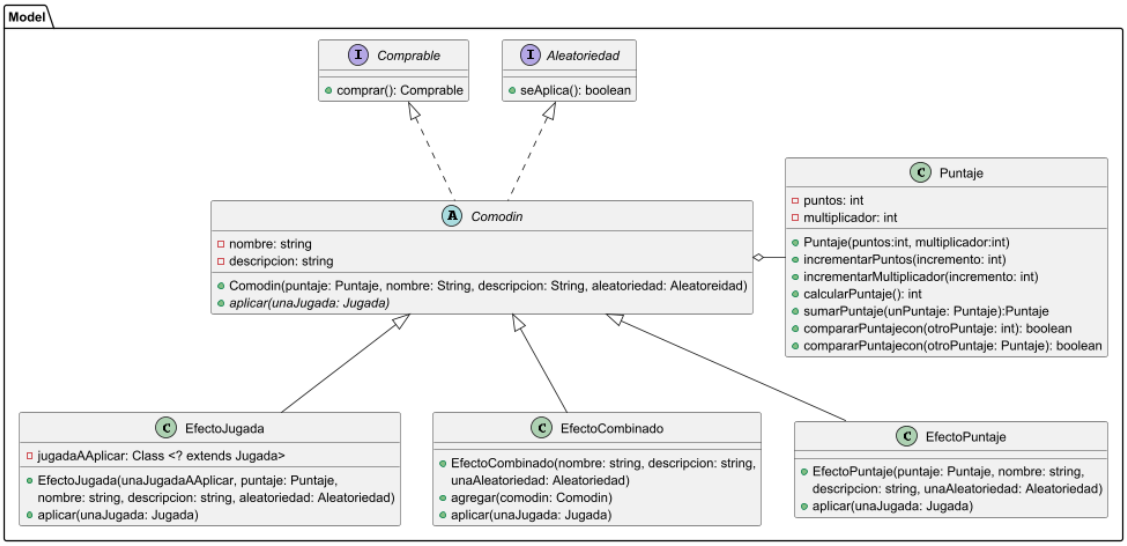


Figura 4: Modelado de los comodines

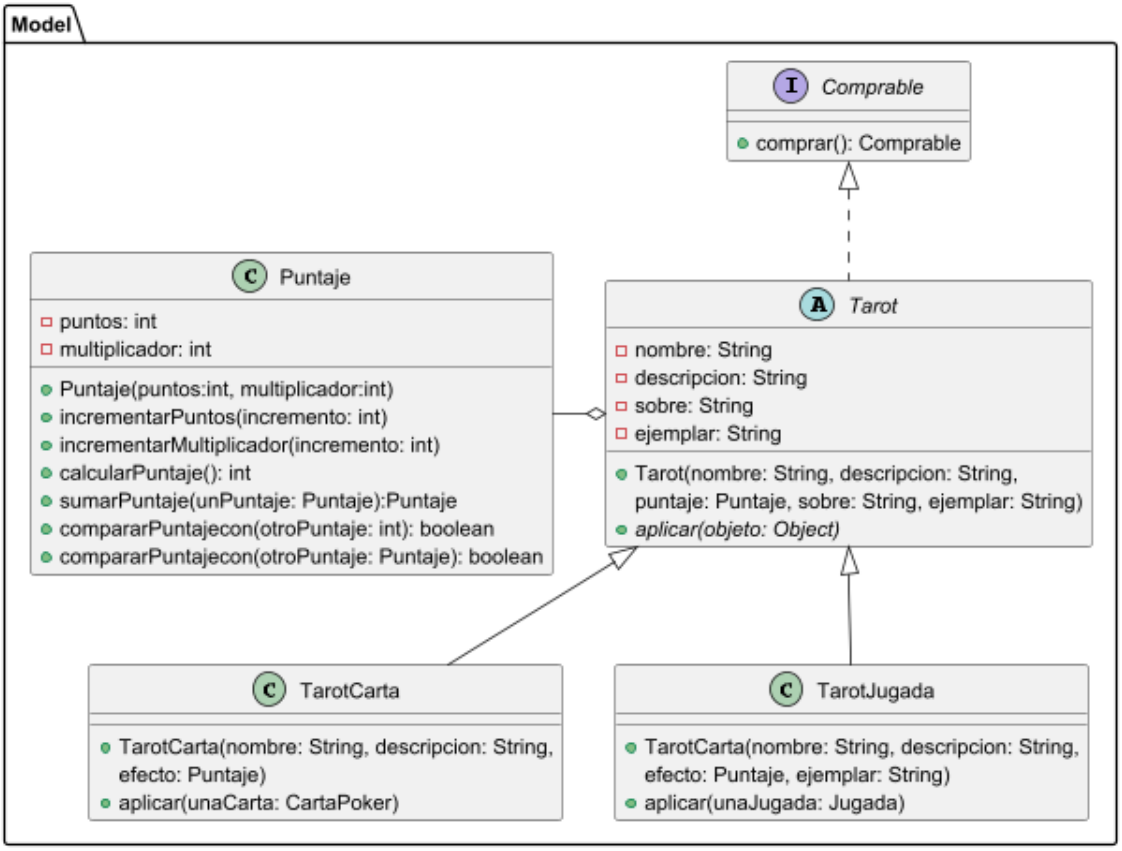


Figura 5: Modelado de las cartas de tipo tarot


Figura 6: Modelado de las cartas de poker

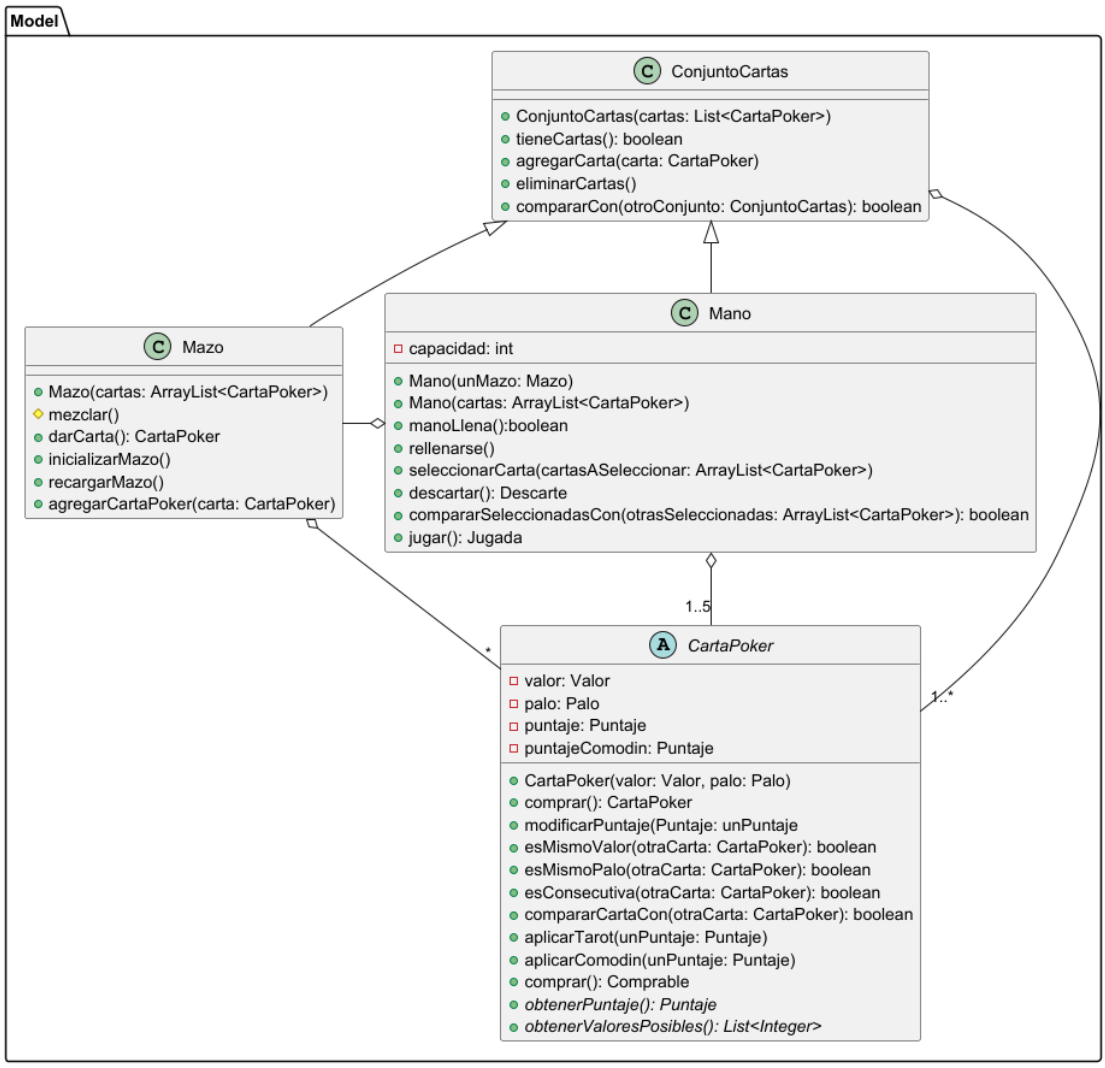


Figura 7: Relación entre los diferentes conjuntos de cartas

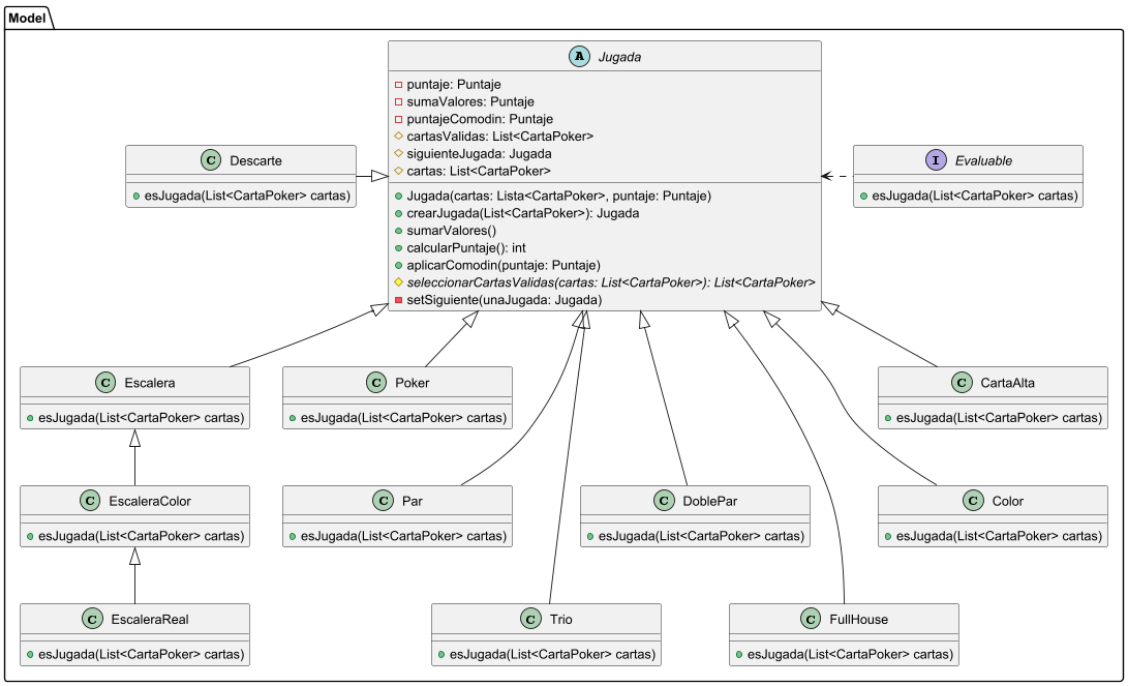


Figura 8: Modelado de las jugadas usando el patrón Chain of Responsability

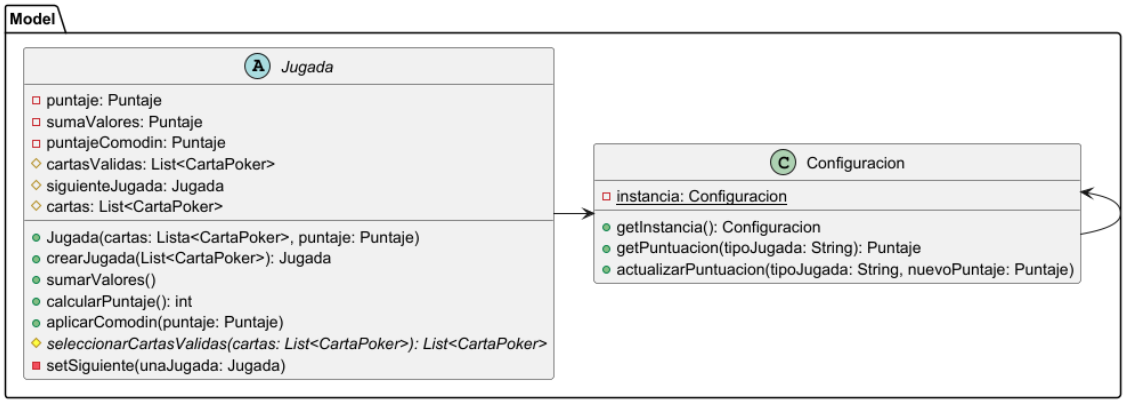


Figura 9: Utilización del patron Singleton para configurar el puntaje de las jugadas.

# 5. Detalles de implementación

## 5.1. Herencia

- Jugadas: Las clases `CartaAlta`, `Color`, `DoblePar`, `Escalera`, `FullHouse`, `Par`, `Poker` y `Trio` heredan de la clase `Jugada`. Las clases `EscaleraColor` y `EscaleraReal`, como son casos más particulares de la clase `Escalera`, forman una especie de cadena en la que una herda de la otra (esto se puede ver bien en la Figura 8). Cada una sabe evaluarse a si misma, es decir, determinar si el conjunto de cartas pertence a su clase o no. Se aplica herencia porque todas comparten un comportamiento en comun, representan jugadas a las cuales se les puede

calcualar el puntaje y aplicar comodines y tartos.

- Comodines: En los comodines tambien se aplica herencia, existiendo tres subclases, `EfectoJugada`, `EfectoPuntaje` y `EfectoCombiando`, hijas de La clase abstracta `Comodin`.
- Cartas de tipo Tarot: La clase `Tarot` es una clase abstracta que define los atributos comunes (como nombre, descripcion, puntaje, etc.) y el método abstracto `aplicar()`. Esta clase sirve como una superclase para las clases `TarotCarta` y `TarotJugada`. La herencia permite que estas subclases compartan los atributos y métodos comunes de `Tarot` y, a la vez, puedan implementar sus propias versiones del método `aplicar()`.

## 5.2. Delegación

- Inicio de una ronda: La clase `Baltro` se encarga de gestionar el flujo del juego. Al comenzar cada una de las ocho rondas, esta clase delega la verificación, correcto inicio y la acción de pedirle cartas al mazo a la clase `Jugador` a través de su propio método:

```
public void jugarRonda() {
    Ronda rondaActual = getRondaActual();
    this.jugador.iniciarRonda(rondaActual);
}
```

La clase `Jugador` luego recibe el mensaje `iniciarRonda()` y se encarga de dar comienzo a la ronda actual:

```
public void iniciarRonda(Ronda rondaActual){
    if(!esPosibleIniciarRonda()) throw new MazoVacioError();
    this.rondaActual = rondaActual;
    this.manoActual.rellenarse();
}
```

A su vez el método `esPosibleIniciarRonda()` delega la responsabilidad de verificar si la instancia de `Mazo` tiene cartas, respetando su encapsulamiento:

```
 public boolean esPosibleIniciarRonda(){
    return this.mazo.tieneCartas();
}
```

Finalmente la clase `Mano` a su vez delega la acción de rellenarse a la clase `Mazo` a través de la iteración del método `darCarta()`, ya que el mazo es el que gestiona sus propias cartas:

```
public CartaPoker darCarta() {
    if (!tieneCartas()) {
        throw new MazoVacioError();
    }
    this.mazoDescarte.add(this.cartas.get(0));
    return cartas.remove(0);
}
```

- Verificación dinámica: A lo largo de la ronda jugada el método `verificarEstadoJuego()` de `Balatro` delega la verificación del puntaje a la instancia de `Ronda`:

```
public void verificarEstadoJuego() {
    if (indiceRondaActual == this.rondas.size() - 1) {
        if (this.getRondaActual().rondaSuperada()) {
            this.estadoJuego = EstadoJuego.GANADO;
        } else {
            this.estadoJuego = EstadoJuego.PERDIDO;
        }
    } else if (!getRondaActual().rondaSuperada()) {
        this.estadoJuego = EstadoJuego.PERDIDO;
    }
}
```

  La ronda recibe el mensaje `rondaSuperada` el cual verifica si el jugador alcanzó el `puntajeMinimo` para superar la ronda, según el `totalRonda` acumulado hasta ahora:

```
public boolean rondaSuperada() {
    int totalRonda = calcularTotalRonda();
    return (this.puntajeMinimo <= totalRonda);
}
```

- Calculo del puntaje de la jugada: Cuando la clase `Jugada` determina su puntaje final en el método `calcularPuntaje()`:

```
public int calcularPuntaje() {
    sumarValores();
    this.puntaje = this.puntaje.sumarPuntaje(this.puntajeComodin);
    return this.puntaje.calcularPuntaje();
}
```

  Esta delega el cálculo a la clase `Puntaje`, ya que está dispone de los atributos `puntos` y `multiplicador`, con los cuales opera para determinar el puntaje final

```
public int calcularPuntaje() { return this.multiplicador * this.puntos; }
```

- Comodines: La delegación es visible en `Comodin` en el uso de atributo `aleatoreidad` de tipo `Aleatoreidad`. La clase delega la responsabilidad de determinar si se aplica o no a la interfaz `Aleatoreidad` a través del método `seAplica()`.

## 5.3. Polimorfismo

- Comparación entre cartas: En la clase `CartaPoker` el polimorfismo se utiliza a través de la implementación del método `compareTo` de la interfaz `Comparable`, permitiendo que las instancias de las clases `Numero`, `Figura` y `As` sean comparados de manera polimórfica.
- Variedad de jugadas posibles: Los métodos abstractos `esJugada()` y `seleccionarCartasValidas()` son sobrescritos en cada una de las clases que heredan de `Jugada`. Esto permite que cada tipo de jugada tenga su propio comportamiento específico. Por ejemplo, una jugada de `EscaleraColor`, el método `esJugada()` tendrá una lógica diferente a la de un `Poker`, pero el mismo nombre del método. Además el método `jugar()` en la clase `Jugador` utiliza polimorfismo al retornar un tipo de jugada que puede ser una instancia de varias clases diferentes, permitiendo que se añadan nuevas jugadas sin modificar el código de `Jugador`.
- Distintas respuestas de los comodines: En la clase `Comodin` el método `aplicar()` está definido como abstracto. Esto permite que las subclases de `Comodin` definan cómo aplicar el comodín a una jugada, lo que genera un comportamiento polimórfico. Dependiendo del tipo específico de `Comodin` que se esté utilizando, el comportamiento de aplicar cambiará, pero siempre

podrá ser tratado de la misma manera debido a que se invoca sobre una referencia de tipo `Comodin`.

- Distinatas respuestas de las cartas de tipo tarot: En `Tarot`, el método `aplicar()` es abstracto y se espera que las subclases lo implementen. Esto es un ejemplo de polimorfismo, ya que se comportará de manera diferente dependiendo de si es llamado en una instancia de `TarotCarta` o `TarotJugada`.

## 5.4. Principio de Responsabilidad Única

- Clase `Balatro`: tiene una única responsabilidad, gestionar el flujo del juego, mientras que la creación de rondas, su inicio y la verificación del resultado se delegan a **Ronda**.
- Clase `CartaPoker`: tiene una única responsabilidad centrada en la representación de una carta.
- Clase `ConjuntoCartas`: su responsabilidad principal es gestionar un conjunto de cartas. Sus métodos se centran en agregar, eliminar y comparar cartas.
- Clase `Jugada`: su responsabilidad principal es representar una jugada de póker, contener las cartas involucradas en la jugada y calcular su puntaje. Cada tipo específico de jugada (como Escalera, Poker, etc.) es manejado por clases especializadas que heredan de `Jugada`. Esto permite que cada tipo de jugada tenga una responsabilidad clara y aislada.
- Clase `Mano`: tiene como única responsabilidad relacionada a la gestión de las cartas, y no se ocupa de otras responsabilidades como la lógixa del puntaje o el comportamiento de las cartas en sí.
- Clase `Mazo`: tiene una responsabilidad bien definida y no hace nada que no esté relacionado directamente con el mazo de cartas.
- Clase `Ronda`: su única responsabilidad es gestionar el ciclo de vida de una ronda.
- Clase `Comodin`: tiene como responsabilidad representar un comodín y cómo se aplica a una jugada.
- Clase `Tarot`: gestiona el puntaje de un tarot como única responsabilidad.

## 5.5. Principio de Sustitución de Liskov

- La clase `Jugada` es una clase abstracta y las subclases heredan de ella, por lo que se puede sistituir cualquier instancia de `Jugada` con una instancia de sus subclases sin que el comportamiento global se vea afectado.
- `Comodin` tiene subclases, estas subclases pueden ser sustituidas por instancias de `Comodin` sin alterar el comportamiento esperado. Los objetos de tipo `Comodin` (y sus subclases) pueden ser usados sin problemas en cualquier parte del código que espere un **Comodin**.
- Las subclases `TarotCarta` y `TarotJugada` pueden ser sustituidas por instancias de `Tarot` sin afectar el comportamiento del sistema, siempre que el contrato del método aplicar se mantenga.

## 5.6. Principio Abierto/Cerrado

- La clase `Balatro` está abierta a la extensión a través del método crearRonda(), lo que permite que el comportamiento de la ronda pueda ser modificada sin cambiar la clase `Balatro` directamente (por ejemplo, sobreescribiendo crearRonda() en una subclase si fuera necesario).

- La clase `CartaPoker` está cerrada para modificaciones, pero abierta para extensiones en cuanto a su funcionalidad. Si se quiere extender el comportamiento de la carta, como crear un tipo de carta diferente.
- La clase `ConjuntoCartas` cumple con este principio ya que si se necesita en un futuro cambiar cómo s comparan las cartas, no tendríamos que modificar la clase, sino que podríamos extenerla o crear una nueva clase que implemente de manera diferente la comparación.
- La clase **Jugada** está cerrada para modificaciones en el sentido de que su estructura no necesita cambiar si se agregan nuevas jugadas, pero se abre a la extensión porque se pueden crear nuevas clases que hereden de **Jugada** sin modificar la clase.
- La clase **Mazo** cumple con este principio ya que se pueden añadir nuevas funcionalidades a través de la herencia.
- La clase **Ronda** esta diseñada para ser extendida (por ejemplo añadiendo nuevas jugadas) sin tener que modificarse.
- La clase **Comodin** tiene el método aplicar el cual es abstracto, por lo que cada clase concreta que herede de **Comodin** puede definir su propia implementación sin afectar al código existente.
- La clase **Tarot** está abierta a la extensión mediante las subclases TarotCarta y TarotJugada. Si necesitamos agregar nuevos tipos de tarot, podemos hacerlo creando nuevas subclases de Tarot sin modificar el código existente.

## 5.7. Principio Inversión de Dependencias

- Clase **CartaPoker**: depende de la clase **Puntaje**.
- Clase **Jugada**: depende de abstracciones (como **Puntaje** y **CartaPoker**) en lugar de detalletes concretos, lo que facilita la extensión y mantenimiento.
- Clase **Comodin**: depende de la interfaz Aleatoriedad y no de implementaciones concretas.

## 5.8. Principio de Segregación de Interfaces

- Clase **Jugada**: implementa la interfaz **Evaluable**, lo cual permite que la clase **Jugada** se enfoque en los métodos necesarios para evaluar las jugadas, sin obligarla a implementar métodos inncesarios.
- Clase **Comodin**: no tiene que implementar toda la funcionalidad de aleatoriedad; simplemente depende de una interfaz que le da la posibilidad de delegar la implementación. Esto permite que diferentes clases de comodines tengan diferentes implementaciones de Aleatoriedad, respetando así la separación de responsabilidades.

## 5.9. Patrones

- En la clase `Comodin`, el patrón **Strategy** se implementa a través de la relación con la interfaz `Aleatoriedad`. Los distintos comportamientos de aleatoriedad (Aleatorio y NoAleatorio) están encapsulados en clases concretas que implementan esta interfaz. La clase `Comodin` delega la decisión de si un efecto se aplica al objeto, que puede ser dinámicamente intercambiado mediante el método `setAleatoreidad`.
- El patrón **Composite** se utiliza en la implementación del comodín `EfectoCombinado`, que extiende la clase `Comodin`. Esta clase actúa como un çontenedor"que puede almacenar múltiples instancias de `Comodin`, permitiendo que estas se comporten de manera uniforme y en conjunto.

```
public class EfectoCombinado extends Comodin {
    private final List<Comodin> combinaciones;

    public EfectoCombinado(String nombre,
                           String descripcion,
                           Aleatoriedad aleatoriedad) {
        super(new Puntaje(0, 1), nombre, descripcion, aleatoriedad);
        this.combinaciones = new ArrayList<>();
    }
}
```

- El patrón **Chain of Responsability** se implementa en los métodos `configurarCadena()`, `evaluar()`, y `crearJugada()` de la clase `Jugada`:

  El método `configurarCadena` establece la cadena de objetos `Jugada`. Cada tipo de jugada se enlaza con otro, formando una estructura jerárquica en la que cada jugada delega la responsabilidad al siguiente objeto en la cadena si no puede manejar la solicitud.

```
private static Jugada configurarCadena(List<CartaPoker> cartas) {
    Jugada escaleraReal = new EscaleraReal(cartas);
    Jugada escaleraColor = new EscaleraColor(cartas);
    Jugada poker = new Poker(cartas);
    Jugada fullHouse = new FullHouse(cartas);
    Jugada color = new Color(cartas);
    Jugada escalera = new Escalera(cartas);
    Jugada trio = new Trio(cartas);
    Jugada doblePar = new DoblePar(cartas);
    Jugada par = new Par(cartas);
    Jugada cartaAlta = new CartaAlta(cartas);

    escaleraReal.setSiguiente(escaleraColor);
    escaleraColor.setSiguiente(poker);
    poker.setSiguiente(fullHouse);
    fullHouse.setSiguiente(color);
    color.setSiguiente(escalera);
    escalera.setSiguiente(trio);
    trio.setSiguiente(doblePar);
    doblePar.setSiguiente(par);
    par.setSiguiente(cartaAlta);

    return escaleraReal;
}
```

  El método `evaluar()` recorre la cadena de jugadas para determinar cuál es capaz de procesar la solicitud (en este caso, identificar qué tipo de jugada corresponde a las cartas proporcionadas).

```
public Jugada evaluar(List<CartaPoker> cartas) {
    if (esJugada(cartas)) {
        return this;
    } else if (this.siguiente != null) {
        return this.siguiente.evaluar(cartas);
    }
    return null;
}
```

  El método `crearJugada()` inicia la cadena de responsabilidad llamando a `evaluar()` sobre

el primer objeto de la cadena (`EscaleraReal`).

```
public static Jugada crearJugada(List<CartaPoker> cartas) {
    Jugada inicioCadena = configurarCadena(cartas);
    return inicioCadena.evaluar(cartas);
}
```

- El patrón **Factory Method** se utiliza en la clase `CartaFactory` mediante el método `crearCarta()`, ya que decide, en tiempo de ejecución, qué clase concreta de **CartaPoker** (As, Figura o Numero), dependiendo del valor proporcionado.

```
public static CartaPoker crearCarta(Valor valor, Palo palo) {
    if (valor == Valor.AS) {
        return new As(valor, palo);
    } else if (valor.ordinal() >= 10) {
        return new Figura(valor, palo);
    } else {
        return new Numero(valor, palo);
    }
}
```

- El patrón **Singleton** se utiliza en la clase `ConfiguracionJugadas` para garantizar que solo exista una única instancia de la tabla de puntajes de jugadas en todo el sistema. Es útil ya que centraliza el acceso a los valores de las jugadas y además permite que, si el puntaje de alguna jugada es modificado (con una carta de tipo tarot por ejemplo), este permanezca modificado para el resto de la partida.

# 6. Excepciones

**ArchivoNoEncontradoError** se lanza en el caso de que el archivo JSON no sea encontrado en el recurso especificado.

**CargarArchivoError** se lanza cuando ocurre un problema general durante la lectura o procesamiento del archivo JSON.

**CapacidadLlenaError** se lanza cuando se quiere agregar un objeto pero este ya llegó a su capacidad máxima.

**CartaNulaError** se lanza cuando se quiere trabajar sobre un objeto de tipo carta que no existe.

**MazoVacioError** se lanza cuando se intenta extraer una carta de un mazo vacío. En la clase `Mazo`, este error se utiliza para controlar situaciones donde el método `darCarta()` intenta entregar una carta, pero el mazo ya no contiene más cartas disponibles.

**NoHayDescarteDisponiblesError** se utiliza cuando se superó la cantidad máxima de descartes que tiene el jugador por ronda.

**NoHayJugadasDisponiblesError** se utiliza cuando se superó la cantidad máxima de jugadas que puede realizar el jugador por ronda.

**PuntajeNuloError** se lanza cuando se trata de trabajar con un objeto Puntaje y este no existe.

**TarotDistintaJugadaError** se utiliza cuando una carta de tipo Tarot se intenta utilizar sobre una jugada que no corresponde.

**TarotsNoDisponiblesError** se utiliza cuando no hay más cartas de tipo "Tarot"disponibles para una acción solicitada. Esta excepción se utiliza, por ejemplo, cuando el juego requiere asignar o utilizar una carta especial de tipo "Tarotz el mazo o la fuente de cartas ya ha agotado todas las disponibles.

**NoEsCartaError** se utiliza cuando en la clase TarotCarta se envia un objeto y este no es un objeto CartaPoker.

# 7. Diagramas de secuencia

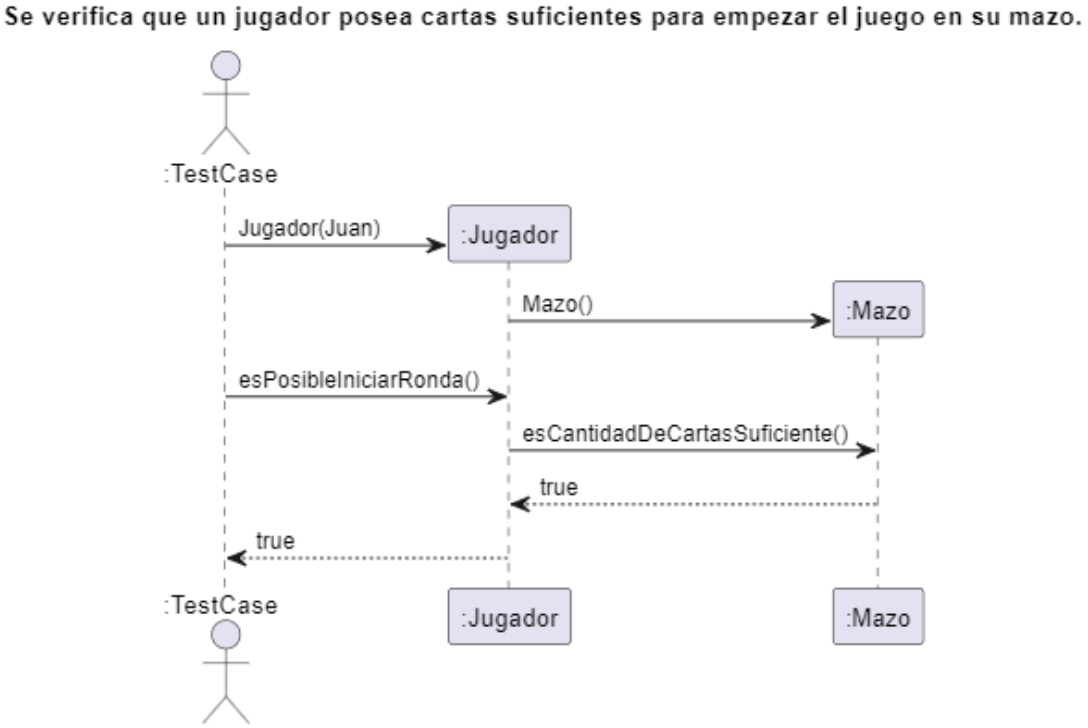


Figura 10: Test01.

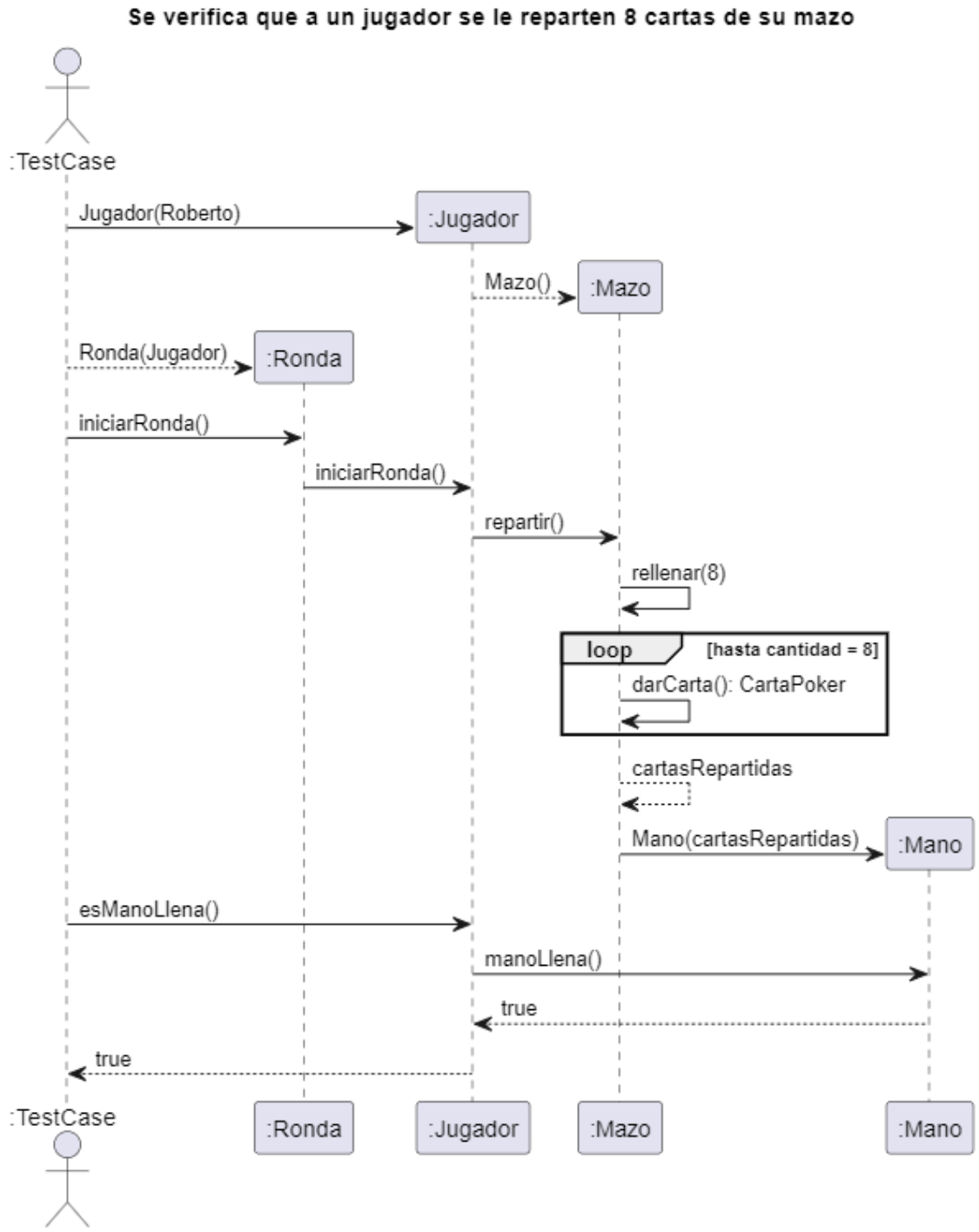


Figura 11: Test02

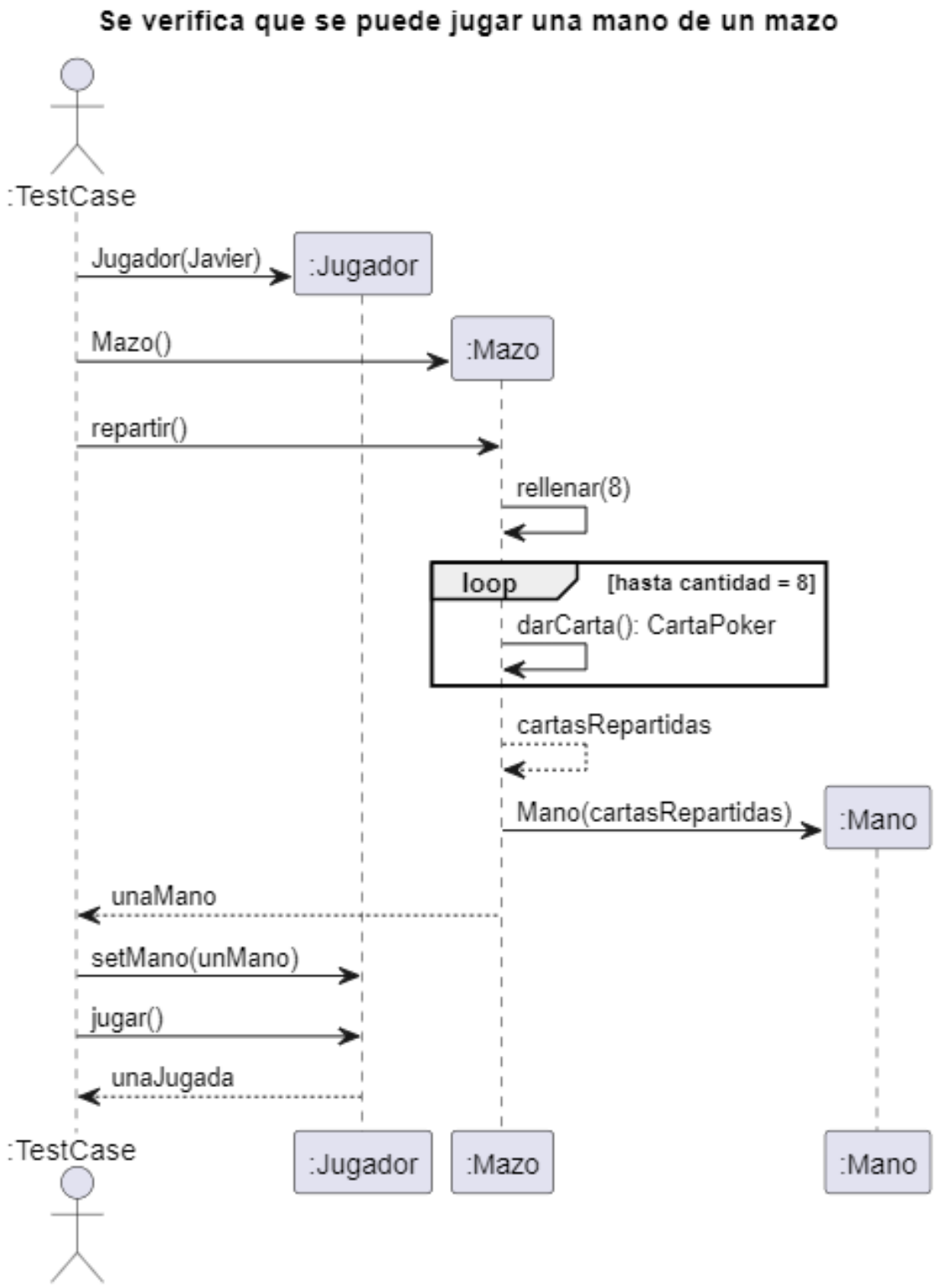


Figura 12: Test03

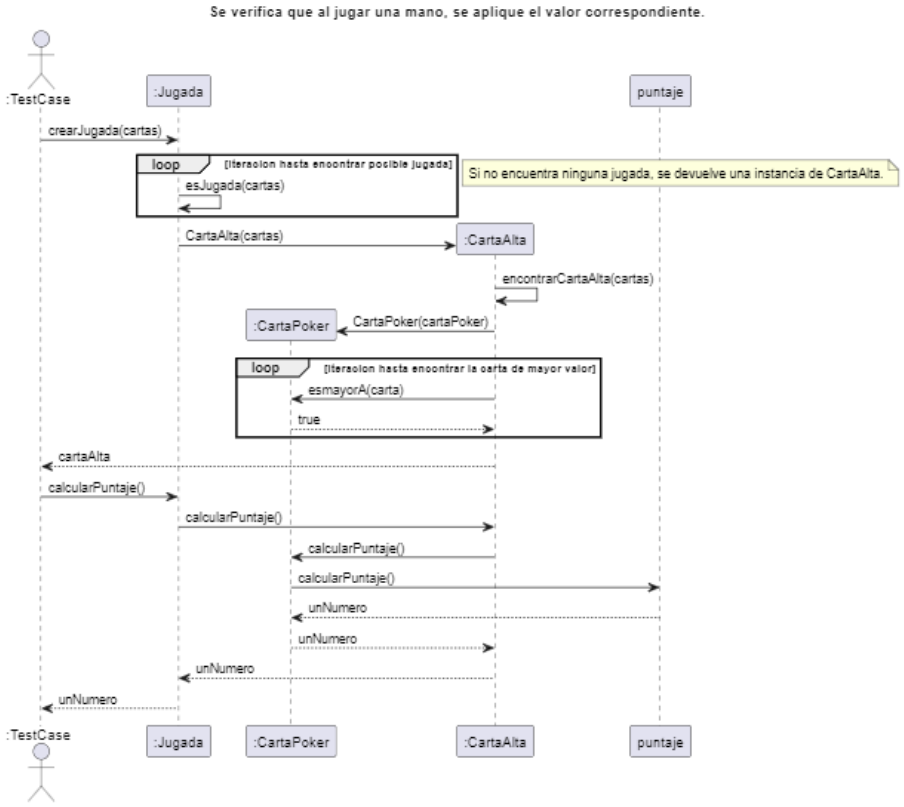


Figura 13: Test04.

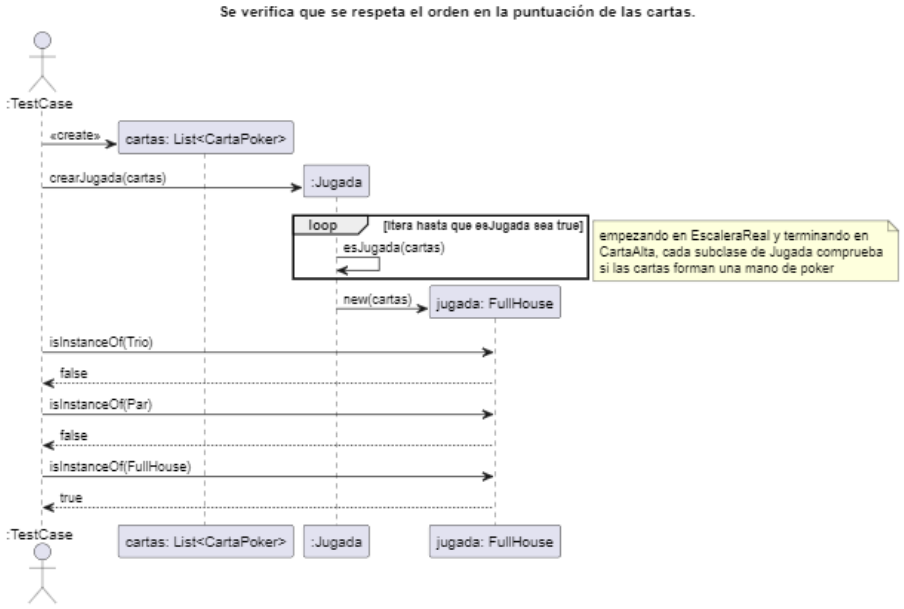


Figura 14: Test05.

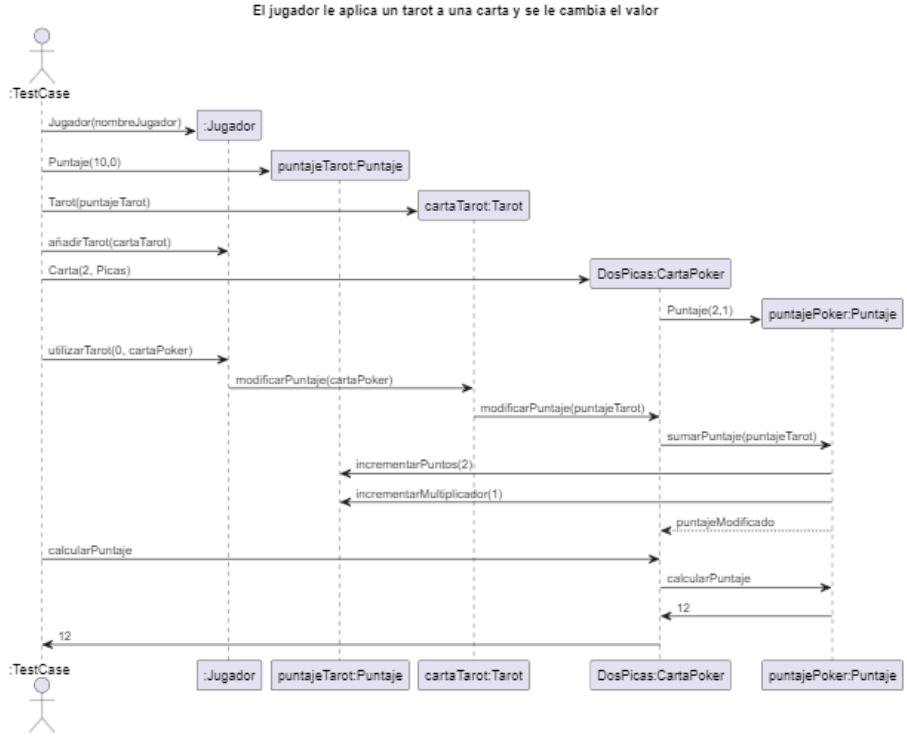


Figura 15: Test06.

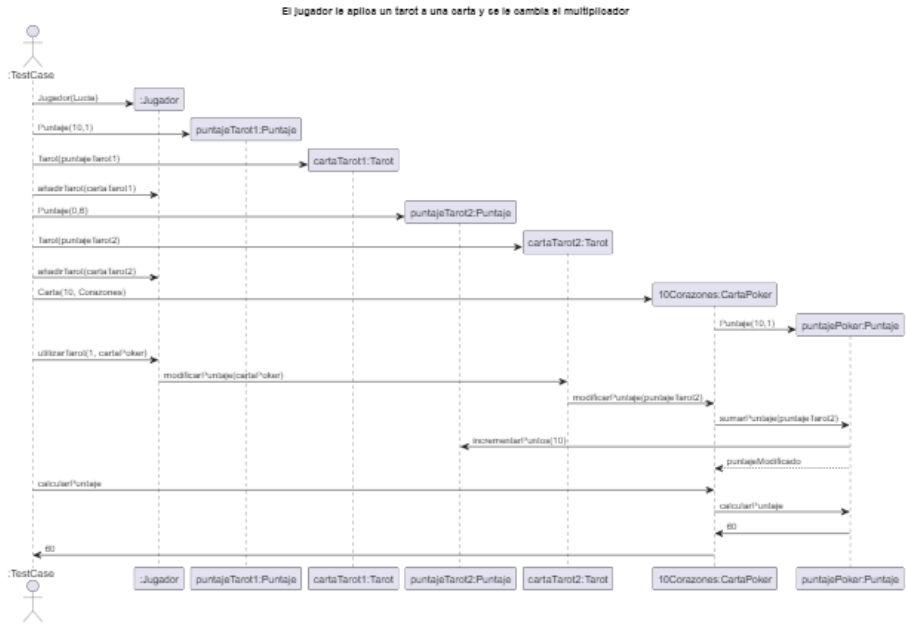


Figura 16: Test07.