FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Balatro

[7507/9502] Algoritmos y Programación III
Curso 01-Suarez
Segundo cuatrimestre de 2024

Alumno:	Número de padrón:	Email:
Limardo, Martin Facundo	110780	mlimardo@fi.uba.ar
Camilo Ignacio Campos Durán	109368	ccamposd@fi.uba.ar
Franco Gabriel Boggia	109175	fboggia@fi.uba.ar
Basterra Sebastián Nahuel	110428	sbasterra@fi.uba.ar

Índice

1. Introducción

El trabajo práctico consta de la realización de un programa realizado en Java, utilizando el paradigma de *Programacion Orientada a Objetos*, y siguiendo *Test Driven Development* como enfoque de desarrollo. El programa debe replicar el videojuego "Balatro", pero con un alcance simplificado. Para la realización del modelo del trabajo se utilizó el IDE IntelliJ con Java, sumado a JavaFX versión 19 y el Scene Builder de Gluon para la creación de la interfaz gráfica; además, el trabajo fue realizado por 4 alumnos, entre ellos 3 utilizaron el sistema operativo Windows 10 y 1 utilizó una distribución de Linux, la última entrega fue revisada y compilada en la última versión de Windows 10 disponible al día de la fecha.

2. Supuestos

- Entre los Jokers, estos solo se pueden activar en 3 instancias de forma automatica: cuando el jugador realiza una jugada, cuando el jugador descarta una carta, o de forma aleatoria con una probabilidad fija cuando se realiza una jugada.
- Entre los Tarots, estos solo poseen 2 categorias, los que se activan y modifican los valores propios de cada jugada, o los que modifican los valores propios de una carta.
- El calculo de los puntos de una mano jugada por el usuario es realizado todo a la vez, al llamado de una funcion que lleve a cabo dicha jugada y el calculo.
- De entre los 4 archivos .JSON proveidos por la catedra, solo leeremos dentro del proyecto el archivo "Balatro.JSON", el resto son utilizados como guias para determinar el alcance de los Jokers, Tarots y las cartas de poker.
- El jugador tiene 8 cartas en su posesion, y estas pasan a formar parte de su "mano."al elegir 5 que desee usar en una jugada. El jugador es ademas responsable de sus cartas, jokers y tarots.
- Continuando del supuesto anterior, no existe un tablero, ya que la forma en que se dispone el juego puede ser entendido como una suerte de "Solitario", en el que la mesa podria ser cualquier cosa siempre que se le puedan apoyar las cartas; la mesa no posee funcion alguna en como se juega, y por tanto no es tomada en cuenta en la abstraccion.

3. Diagramas de clase

Figura 1: Diagrama de clases del Juego y sus clases directas asociadas

Figura 2: Diagrama de clases para Jugador y sus clases asociadas

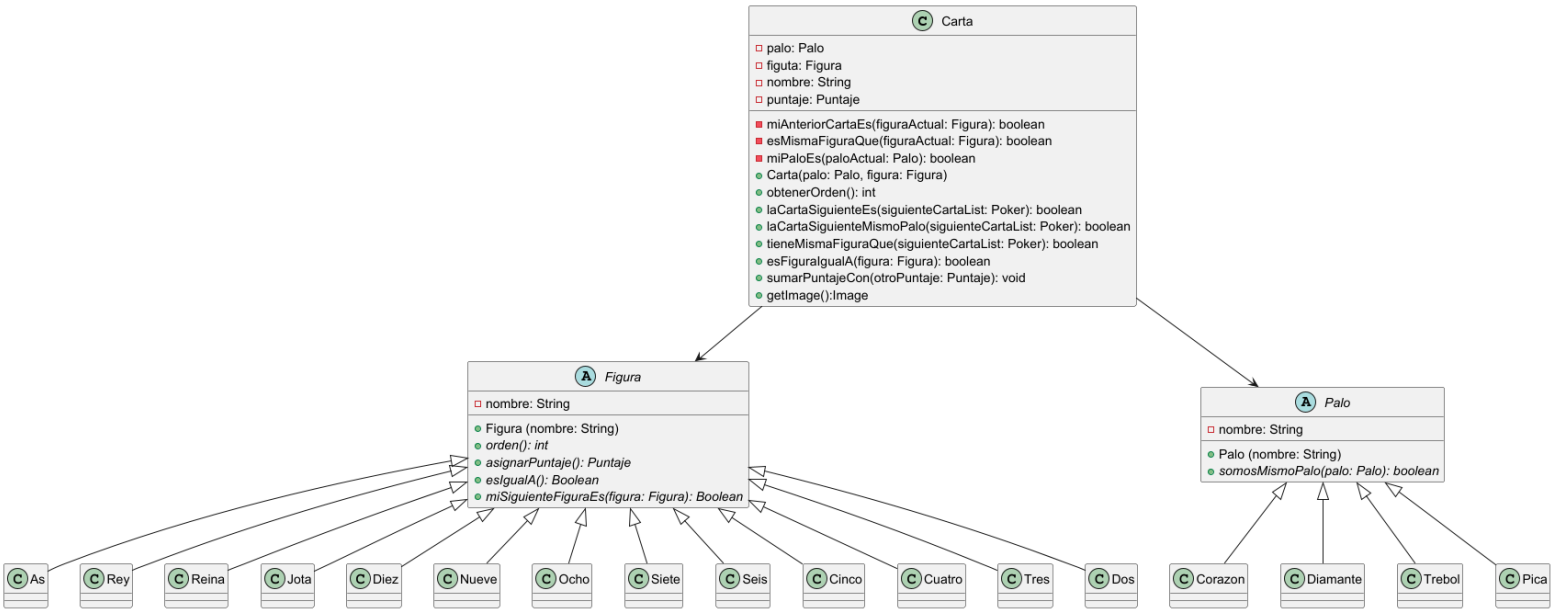

Figura 3: Diagrama de clases de Carta, Figura y Palo

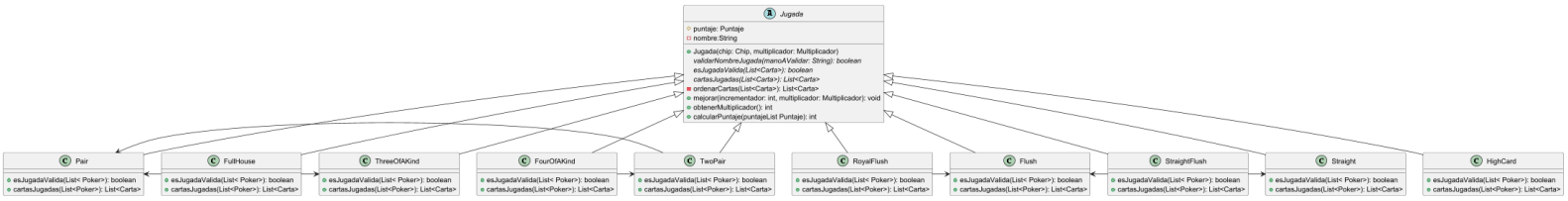

Figura 4: Diagramas de todas las Jugadas de Poker

Notese que algunas jugadas poseen dentro de si instancias de otras jugadas, esto es porque en la practica, por ejemplo: un full es un par y un trio, asi que vale revisar esas dos jugadas previamente, y si existen en simultaneo en una misma mano, tenemos un full.

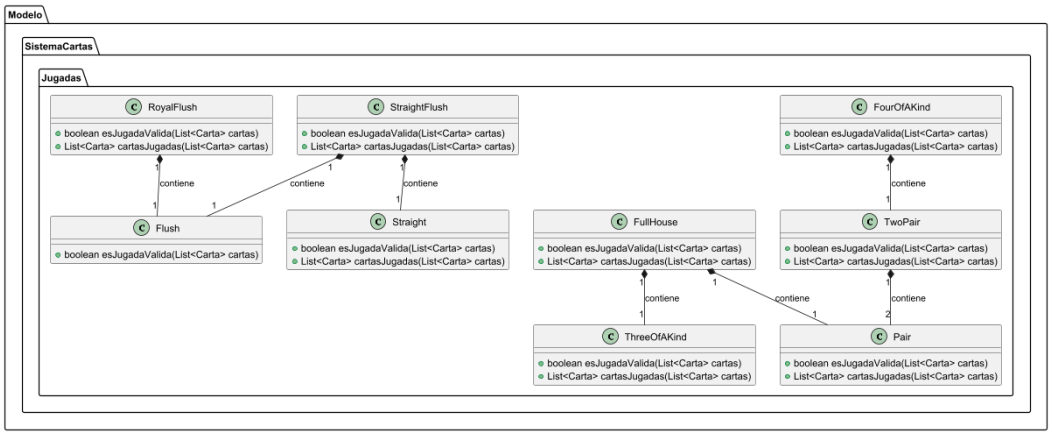

Figura 5: Relacion entre las clases de jugadas de poker

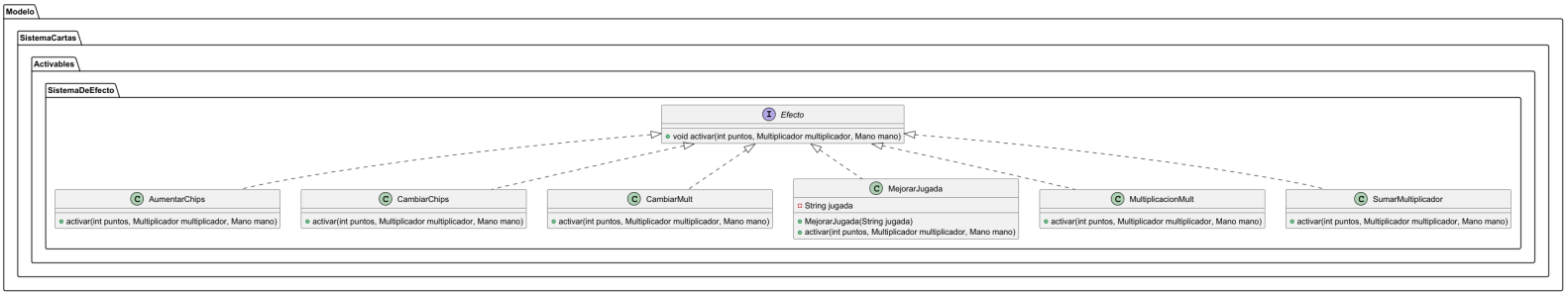

Figura 6: Diagrama de la interfaz Efecto

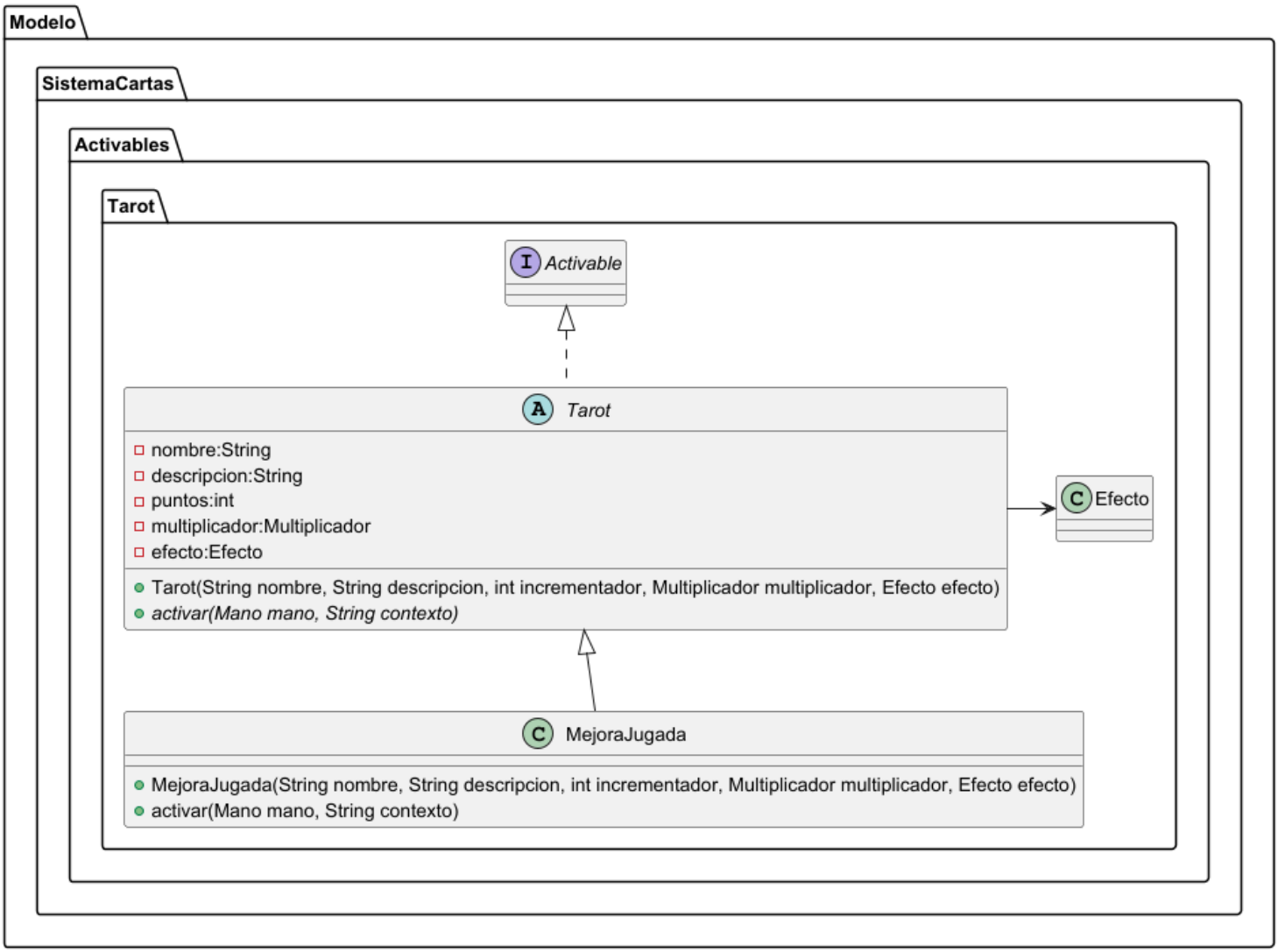

Figura 7: Diagrama de clases de la clase Tarot

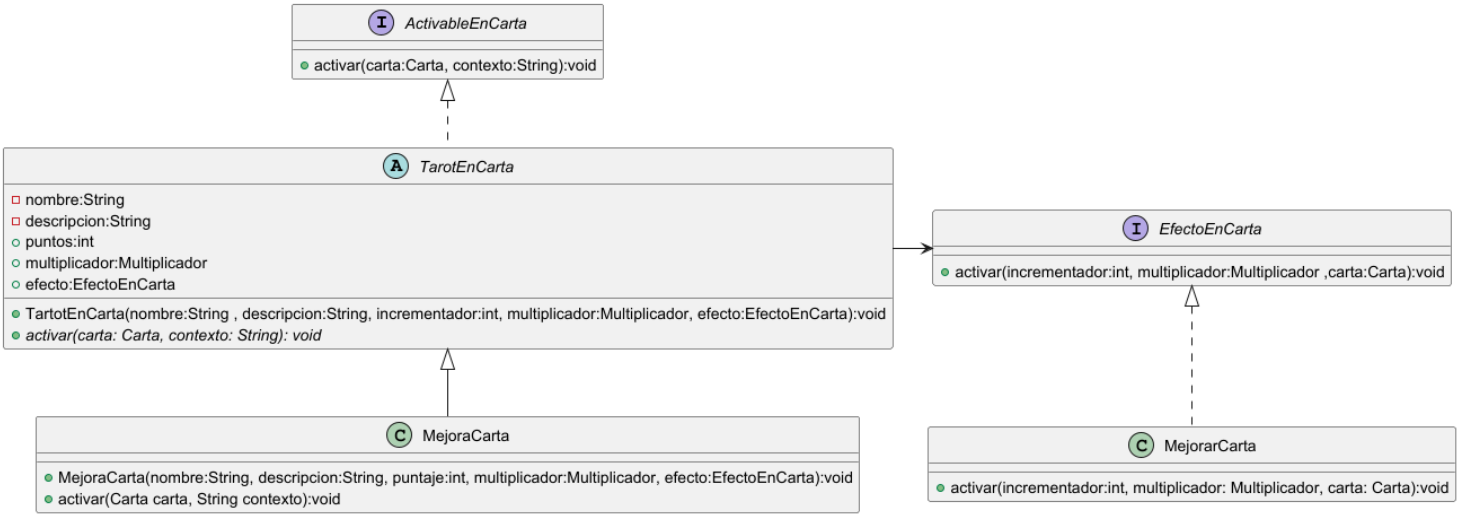

Figura 8: Diagrama de Clases para cartas que aplican efectos en otras cartas

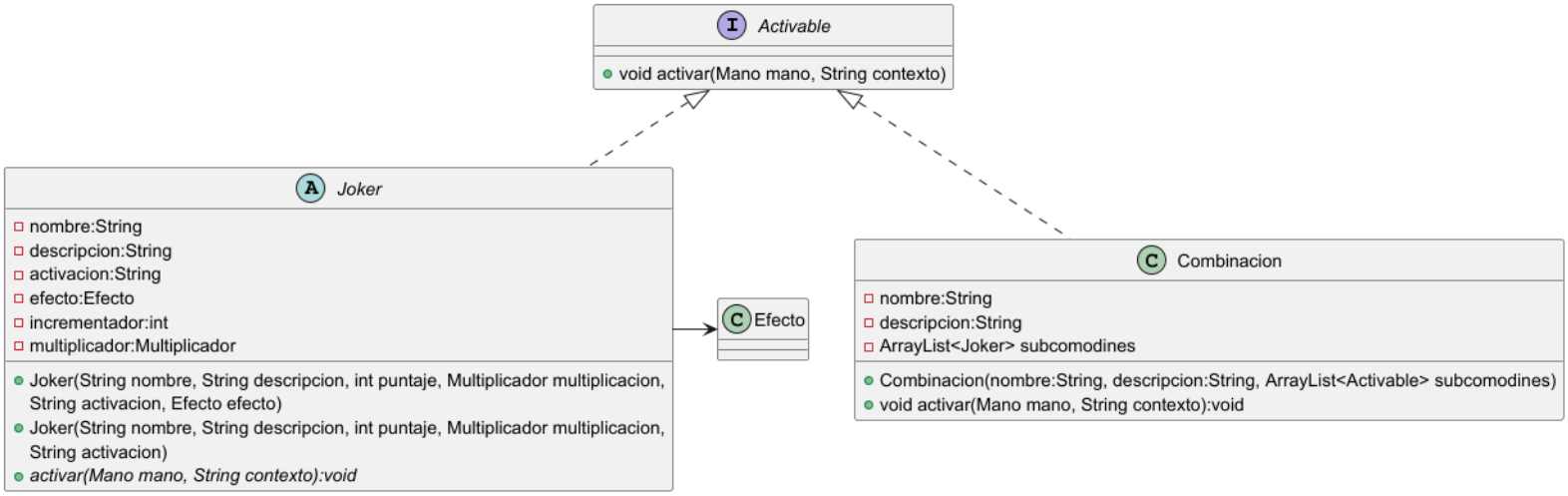

Figura 9: Diagrama de clases de Joker y la interfaz Activable

A tener en cuenta, a pesar de que no se muestra en este diagrama por problemas de espacio en la hoja, Joker cuenta con 4 clases hijas, a saber:

- **PorJugada:** Joker que se activa cuando la jugada correspondiente ocurre.
- **PorDescarte:** Joker que se activa cuando se descarta una carta.
- **UnoEn:** Joker que tiene una chance aleatoria de activarse.
- **AlPuntaje:** Joker que se activa de forma automatica incondicionalmente.

Cada Joker y cada Tarot que implementa la interfaz *Activable* hace uso de un efecto en su estado. Cada efecto realiza una accion distinta sobre la mano del jugador, y todos se utilizan de la misma manera.

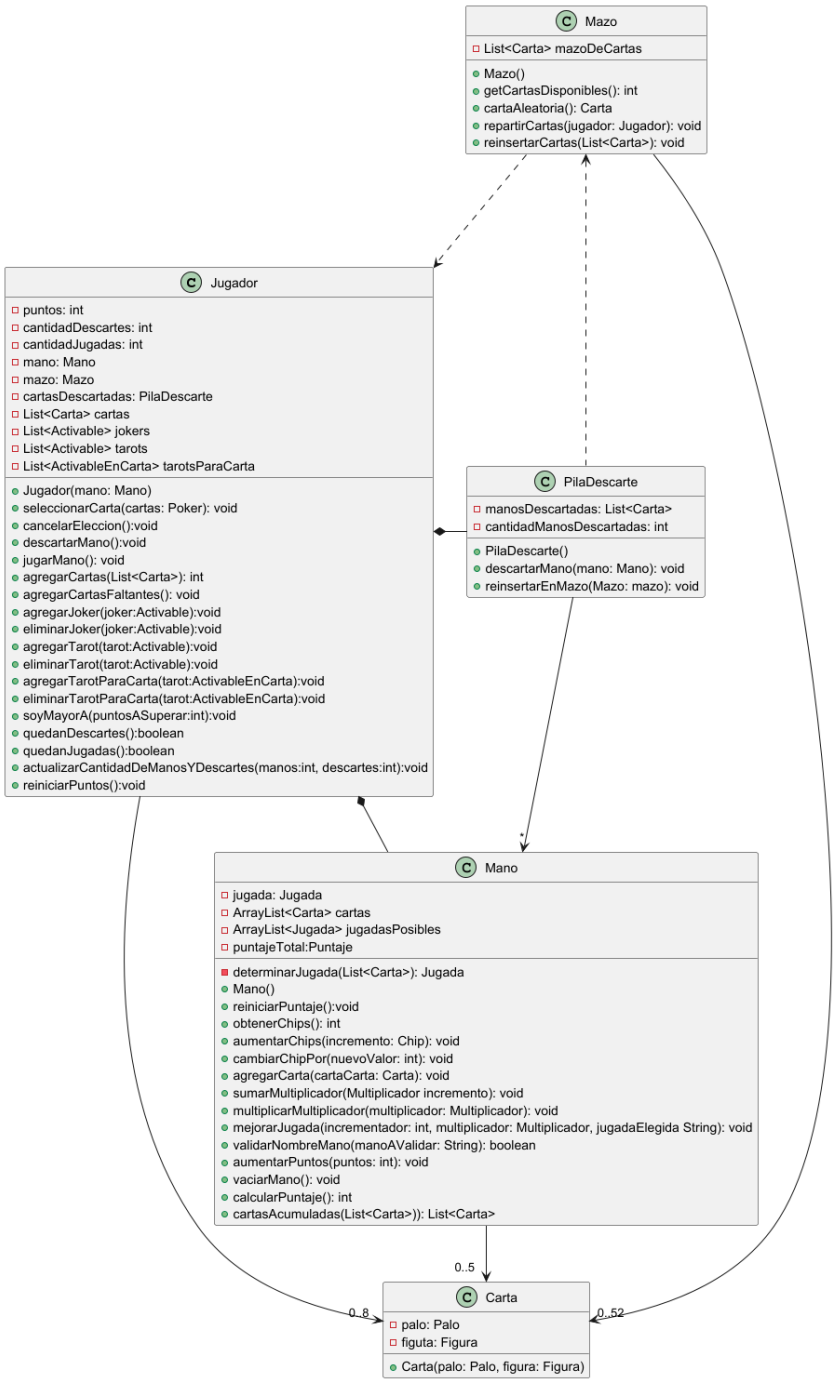

Figura 10: Diagrama de la pila de descarte del jugador

4. Diagramas de secuencia

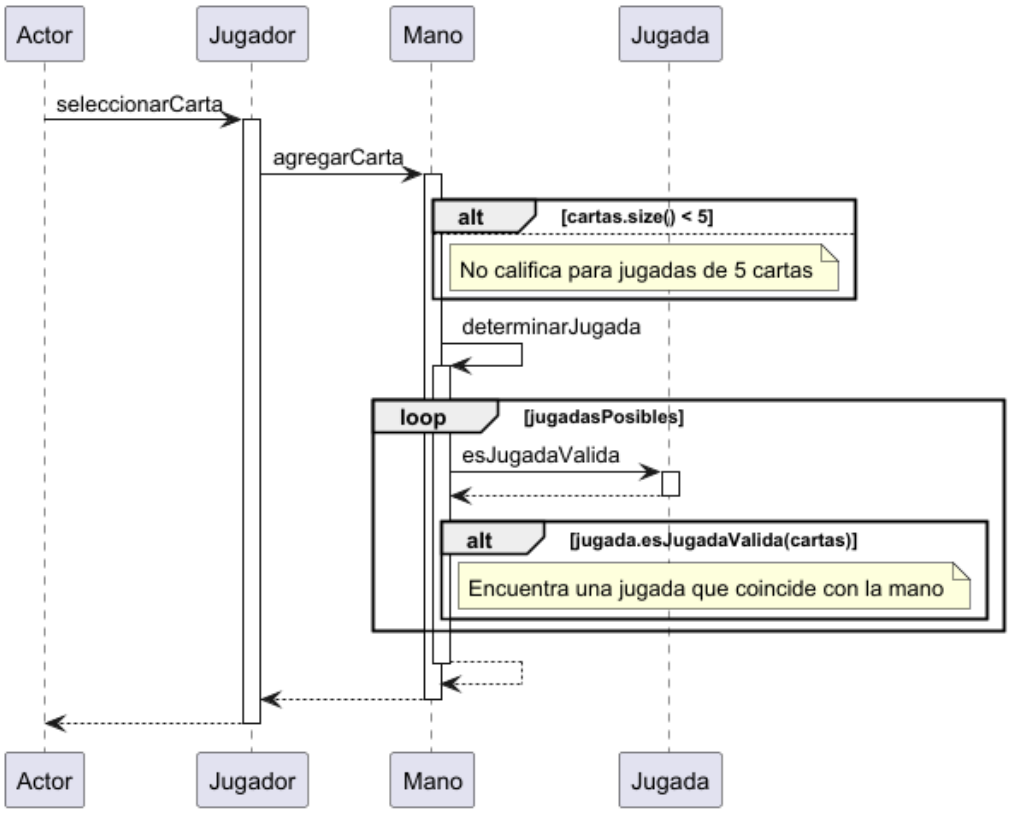

Figura 11: Diagrama de secuencia del Jugador seleccionando una carta para agregar a su mano

Los siguientes diagramas representan las secuencias mas involucradas del modelo, realizar una jugada y calcular el puntaje de una jugada.

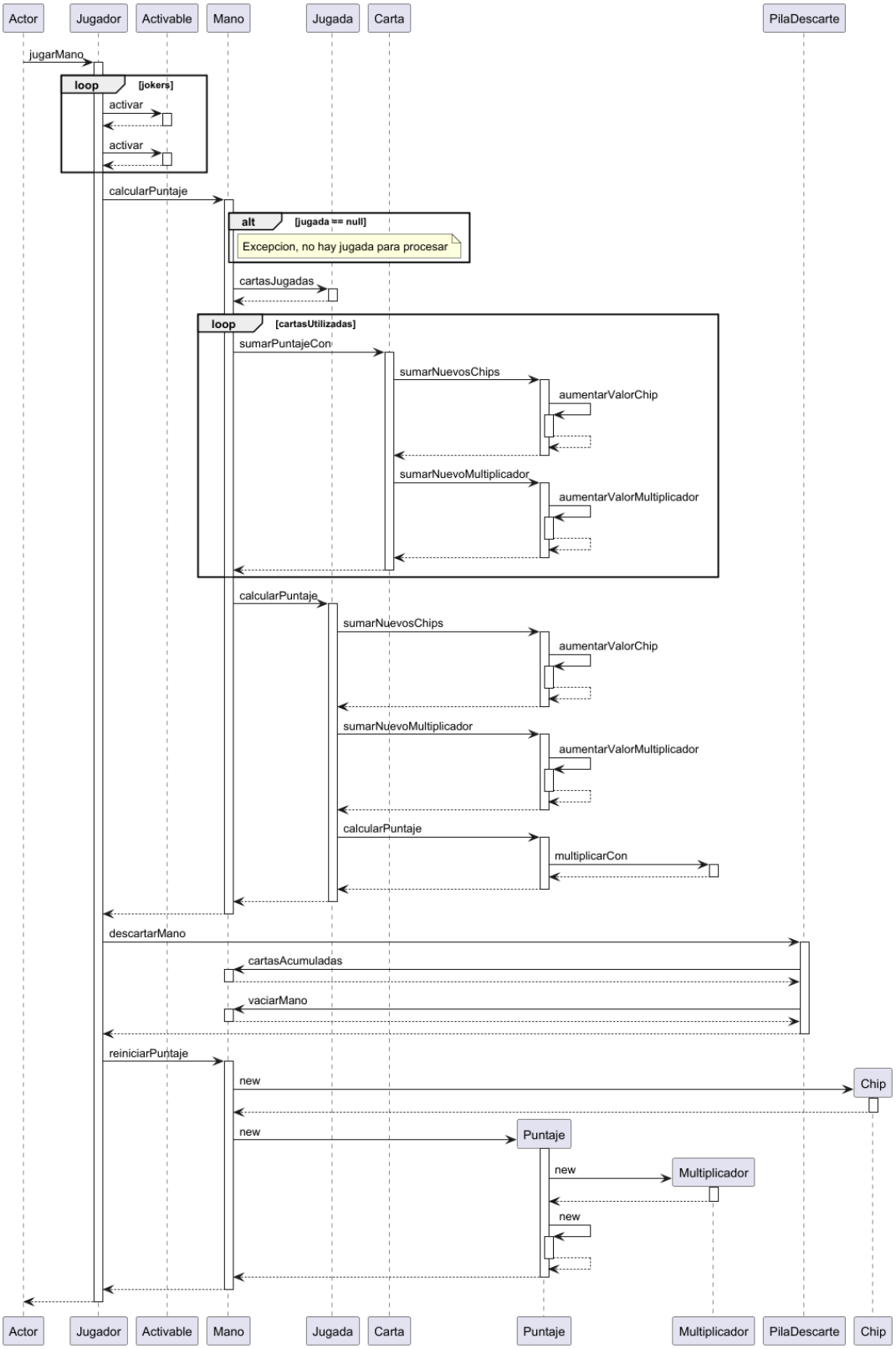

Figura 12: Diagrama de secuencia del jugador realizando una jugada utilizando las cartas en su mano

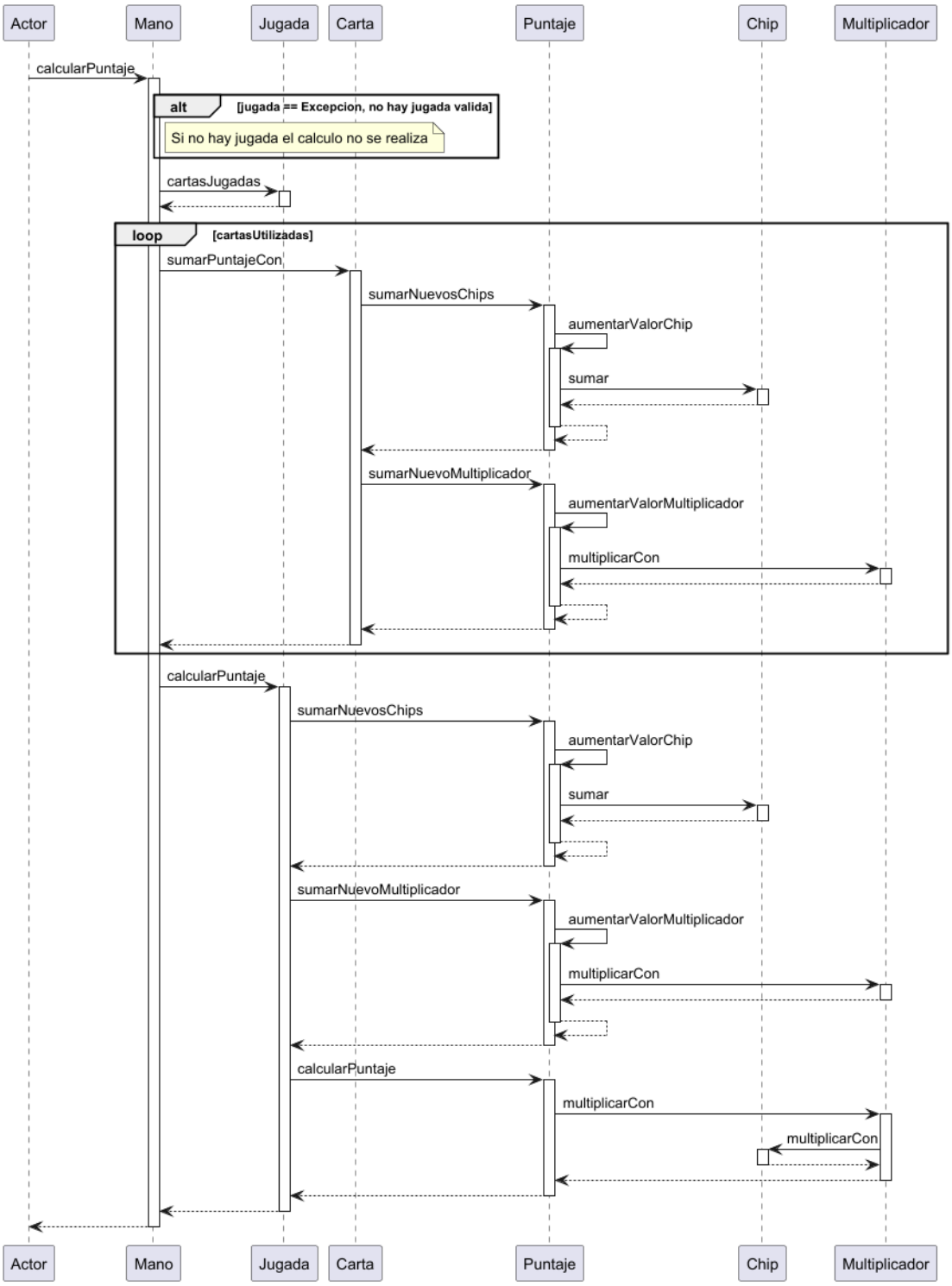

Figura 13: Diagrama de secuencia del calculo de puntaje realizado por la mano en una jugada

5. Diagramas de paquete

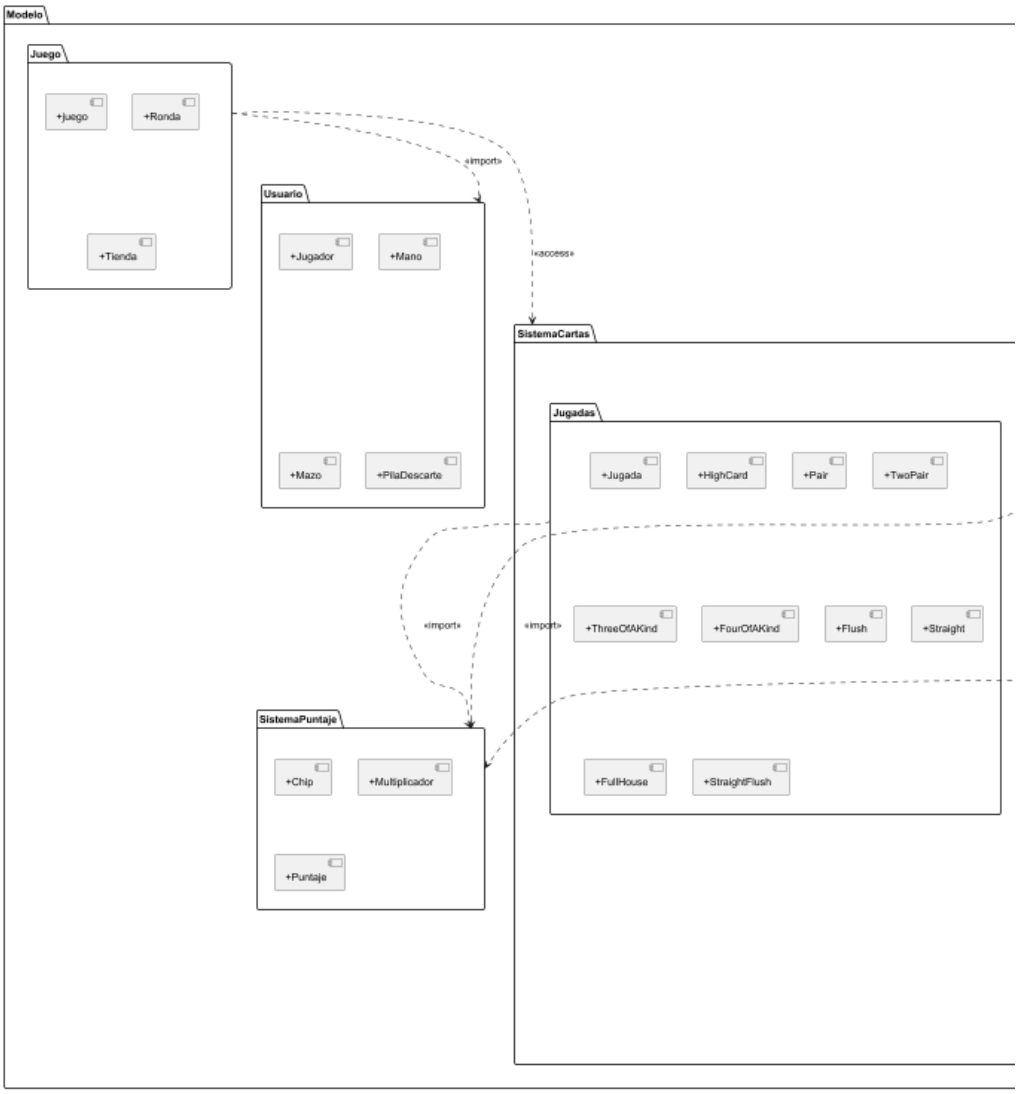

Figura 14: Diagrama de Juego, Usuario, SistemaPuntaje y primera parte de Sistema Carta

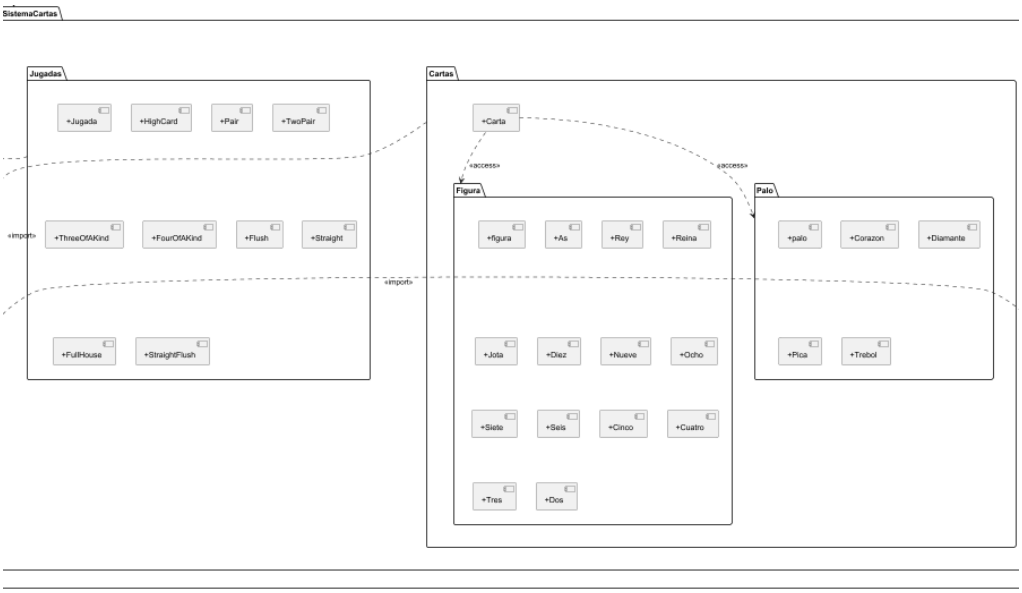

Figura 15: Diagrama de la primer mitad de SistemaCarta

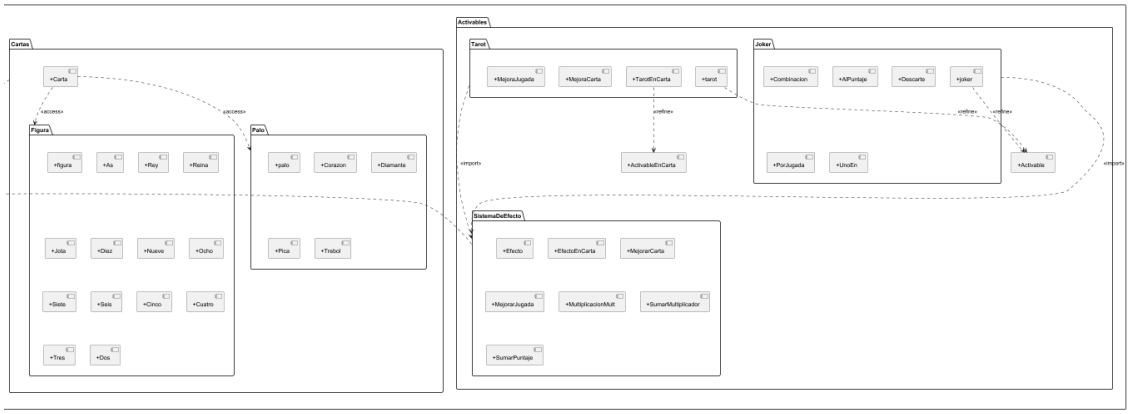

Figura 16: Diagrama de la segunda mitad de sistema carta

6. Detalles de implementación

6.1. Detalles Generales

El proyecto consta de 65 archivos .java, 2 archivos .fxml, 1 archivo .css, y 4 archivos .JSON otorgados por la catedra, de entre los cuales empleamos uno: "balatro.JSON", para la ejecucion del programa.
Entre los archivos .java, no contamos tests, pero si contamos 4 interfaces, 7 clases abstractas, 48 clases concretas, 2 clases dedicadas al patron factories, 2 controladores dedicados a Javafx, y un archivo Main.
El programa dispone entre sus archivos .java dos divisiones a nivel general, la primera esta detallada en el diagrama de paquetes: el modelo que describe el videojuego "Balatro"; la segunda parte corresponde a Javafx, y corresponde a los dos controladores **ModeloController** y **TiendaController**. Ambas partes estan relacionadas gracias al archivo Main.java, que instancia la aplicacion, llama al stage de Javafx y este se comunica directamente con el modelo.
En tanto al modelo, en terminos generales este dispone de la clase **Juego.java** que almacena todo lo necesario para poder, valga la redundancia, jugar al videojuego. Juego hace uso del patron de diseño "Singleton" para evitar que mas de un juego pueda existir a la vez y facilitar su uso desde los controladores, su implementacion se ve de la forma:

```
private static Juego instancia;
public static Juego getInstance() throws FileNotFoundException {
        if (instancia == null) {
            instancia = new Juego(); // Inicializa la instancia si no existe
        }
        return instancia; // Devuelve la instancia existente
}
```

De esta forma, cada vez que el controlador de la tienda o del juego tenga que interactuar con el juego, nos aseguramos que estamos interactuando siempre con el mismo juego, independientemente de los cambios que sufra en ejecucion.

6.2. Interfaz Activable

Para el uso de cartas, que no son cartas de poker, a saber, *Jokers* y *Tarots*, hicimos uso de las interfaces **Activable** y **ActivableEnCarta** respectivamente. El "contrato" impuesto por estas interfaces le permiten a los objetos que se adhieran activar efectos sobre la mano del jugador o sobre una carta en la mano del jugador.

6.2.1. Activable

Mayormente utilizada en **Jokers** y tambien utilizada por un tipo de **Tarot**, `Activable` otorga la firma de los metodos:

```
public  void activar(Mano mano, String contexto);

public String getDescripcion();

public Image getImage();
```

- `activar`: es el metodo principal, recibe una mano del jugador y un contexto dado por un string. El contexto es utlizado por **Joker**, ya que estos pueden activarse en una variedad de escenarios, el contexto es uno de sus atributos que le permite identificar si el uso del metodo resultara en el uso del **Efecto** que tambien tiene como atributo. **Tarot** no hace uso de contexto, directamente hace uso de su efecto cuando se llama a este metodo.

- `getImage`: es un metodo compartido por ambas interfaces y por la clase **Carta**, y nos permite obtener una imagen asociada al nombre de un activable, utilizado en la interfaz grafica del proyecto.

6.2.2. ActivableEnCarta

Esta interfaz es muy similar a **Activable**, y es utilizada en concreto en **Tarot** para nuestro uso, sus metodos tienen la forma:

```
public  void activar(Carta carta, String contexto);

String getDescripcion();

public Image getImage();
```

Donde la principal diferencia es que el metodo `activar` ahora recibe una carta en lugar de una mano. Esto fue creado a favor de no tener que implementar que requiera que siempre que queramos interactuar con una unica carta tengamos que interzactuar con toda una mano y tomar una carta en particular de su lista de cartas. Ademas, esto abre las puertas a posibles comodines, u otros tipos de cartas, en el futuro que interactuen con una unica carta.

La interfaz `Efecto` y su compañera `EfectoEnCarta` siguen los mismos principios, y son utilizados en los mismos escenarios. La diferencia es que se utilizan para los efectos en lugar de las cartas, permitiendole a cualquier tarot o joker simplemente "activar" su efecto sin preocuparse por que es lo que tiene que hacer ese efecto o que combinacion se presente en el archivo de configuracion.

6.3. Factories

6.3.1. CartaFactory

La clase **CartaFactory** implementa el Patrón Factory, cuyo propósito es encapsular la lógica de creación de objetos de tipo `Carta`. Este patrón permite desacoplar la lógica de construcción de instancias específicas, facilitando la extensión del código en el futuro. En este caso, el método estático `crearCarta` recibe como parámetros cadenas de texto que representan el "paloz la "figura"de la carta.

El método utiliza estructuras condicionales para mapear las cadenas de texto como "Trebol" o "Rey" a las clases concretas `Trebol`, `Rey`, y similares. Si las entradas no coinciden con los valores esperados, lanza una excepción `IllegalArgumentException`, garantizando la integridad de los objetos creados. Al final, el método combina las instancias de `Palo` y `Figura` para devolver un objeto `Carta` totalmente inicializado.

El uso del Patrón Factory aquí aporta varias ventajas. Reduce el acoplamiento entre las clases cliente y las implementaciones específicas de `Carta`, y mejora la escalabilidad del sistema al facilitar la adición de nuevos tipos de `Palo` o `Figura`.

6.3.2. JokerFactory

La clase**JokerFactory** también implementa el Patrón Factory, aunque en este caso la complejidad de la lógica de creación es mayor debido a las diversas combinaciones posibles de los objetos `Joker`. El método principal, `crearJoker`, recibe varios parámetros como el nombre, descripción, grupo de activación, incrementador y efecto asociado. Dependiendo del grupo de activación, el método decide qué subclase de `Joker` instanciar, como `PorJugada`, `UnoEn` o `Descarte`.

Adicionalmente, la clase utiliza clases relacionadas con efectos, como `SumarPuntaje` o `MultiplicacionMult`, que se inyectan en las instancias de `Joker`, promoviendo la composición sobre la herencia. Este diseño es flexible y sigue el principio de **Open/Closed**, permitiendo extender las funcionalidades de los `Joker` sin modificar el código existente.

6.4. JSONReader.java

La clase **JSONReader** actúa como un parser que procesa datos en formato JSON y los convierte en objetos del dominio, como `Joker`, `Carta` y otros elementos relacionados con la lógica del juego. Utiliza la biblioteca Gson para analizar las estructuras JSON, y ofrece varios métodos estáticos, entre ellos `obtenerJokers`, `obtenerTarotsEnJugada`, `obtenerTarotsEnCarta` y `obtenerRondas`.

El procesamiento de los datos JSON se divide en dos etapas principales: validación y construcción de objetos. El método `validarJSON` se asegura de que los datos contengan las claves necesarias antes de procesarlos. Posteriormente, métodos como `procesarJokers` transforman las estructuras JSON en listas de objetos `Activable`, manejando también subestructuras complejas como combinaciones de comodines. Además, `JSONReader` interactúa con las clases `CartaFactory` y `JokerFactory` para delegar la creación de objetos específicos, manteniendo su enfoque en el parsing.

6.5. Jugada

La clase `Jugada` representa una acción o combinación de cartas jugada por un jugador durante una ronda. Esta clase es fundamental para evaluar las jugadas y determinar su validez, así como para calcular los puntos obtenidos en base a las reglas del juego.

6.5.1. Métodos principales

- `+ boolean esJugadaValida(List<Carta>cartas):`
  Evalúa las condiciones específicas de la jugada correspondiente.
- `+ List<Carta>cartasJugadas(List<Carta>cartas):`
  Devuelve una lista de cartas ordenadas que pertenecen a una Jugada en concreto.
- `+ float calcularPuntaje(Puntaje puntajeList):`
  Este método utiliza el puntaje actual de la jugada, de sus cartas involucradas, y lo combina con el puntaje proporcionado como parámetro para producir un puntaje final acumulado.

6.5.2. Relaciones con otras clases

- **Relación con la clase** `Mano`:
  La clase `Mano` representa un conjunto de cartas en posesión de un jugador, y su relación con `Jugada` es clave, ya que:
  - **Determinación de jugadas:** `Mano` utiliza las subclases de `Jugada` (como `RoyalFlush`, `Straight`, etc.) para identificar cuál de estas jugadas es válida según las cartas actuales en la mano. Esto se realiza a través del método `deteminarJugada`, que evalúa las jugadas posibles llamando a `esJugadaValida`.
  - **Evaluación de cartas jugadas:** Cuando una jugada es válida, `Mano` invoca el método `cartasJugadas` para obtener las cartas específicas que forman dicha jugada.
- **Relación con la clase** `Puntaje`:
  La clase `Puntaje` se utiliza para calcular y manipular los puntos asociados a una jugada. Su relación con `Jugada` y `Mano` es la siguiente:
  - `Jugada` encapsula un objeto de tipo `Puntaje`, lo cual permite modificar y mejorar los puntos asignados a cada jugada mediante métodos como `mejorar`.

6.5.3. Clases Hijas

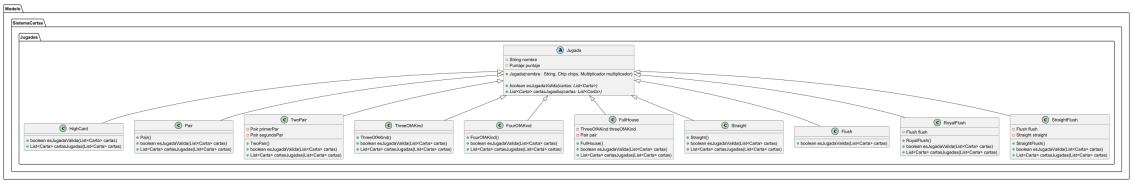

Figura 17: Diagrama de Clases: Sistema Puntaje

6.6. Sistema Puntaje

Como esta planteado nuestro modelo, creamos las clases **Puntaje**, **Chip** y **Multiplicador**, esto ocurre principalmente por la complejidad que se maneja a la hora de relacionar distintas clases, por ejemplo: **Carta** con **Joker** ó **Jugada** con **Tarot**

Se presenta el diagrama de clases con el fin de entender las relaciones de las clases, que se explicarán más adelante:

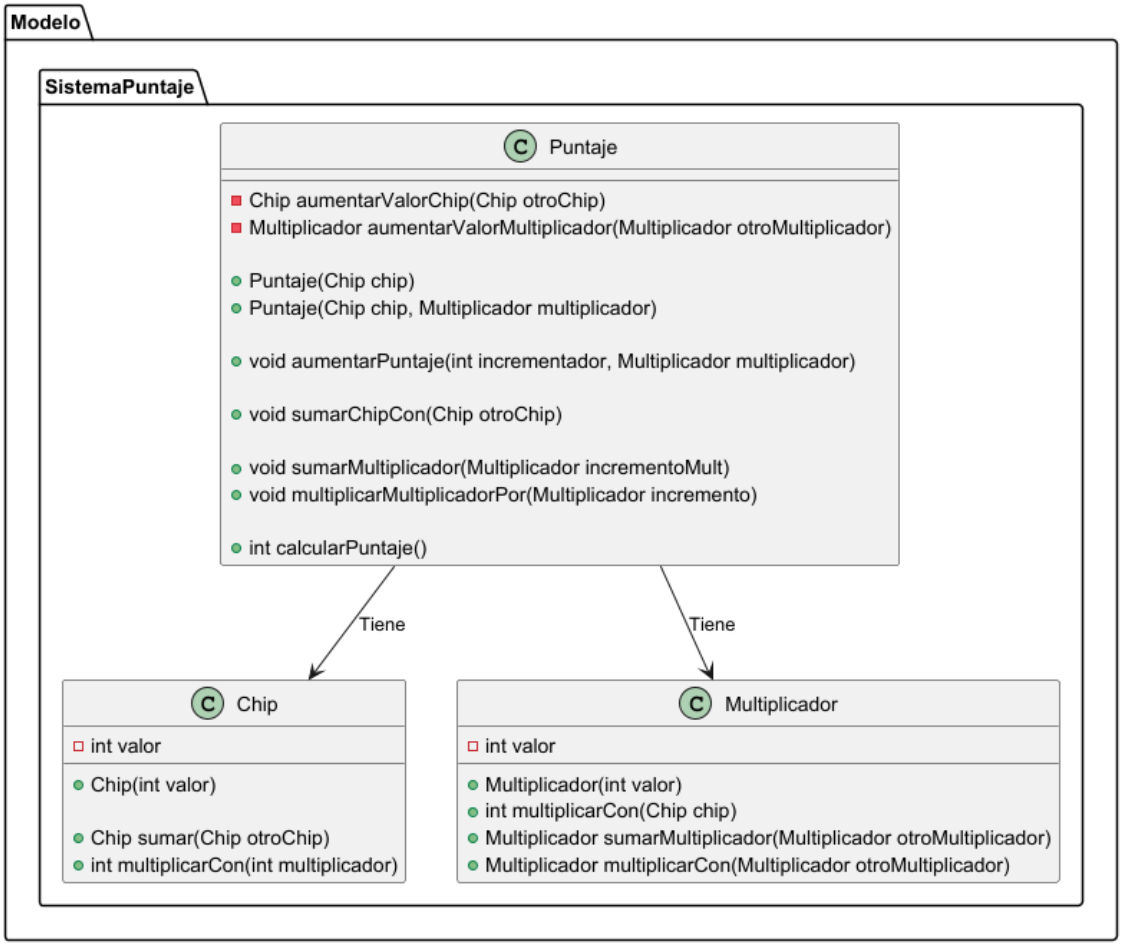

Figura 18: Diagrama de Clases: Sistema Puntaje

6.6.1. Puntaje

La clase **Puntaje** es el núcleo del sistema de puntuación, encargada de gestionar y combinar valores provenientes de las clases **Chip** y **Multiplicador**. Esta separación modular permite operar de manera eficiente sobre las diferentes propiedades del puntaje.

Métodos principales

- **Gestión del Chip**:
  - `sumarChipCon(Chip otroChip)`: Combina el valor del `Chip` actual con el de otro `Chip`.
  - `aumentarValorChip(Chip otroChip)`: Aumenta el valor del `Chip` interno con el valor de otro `Chip`.

- **Gestión del Multiplicador**:
  - `sumarMultiplicador(Multiplicador incrementoMult)`: Incrementa el valor del multiplicador.
  - `multiplicarMultiplicadorPor(Multiplicador incremento)`: Realiza la multiplicación entre multiplicadores.
  - `aumentarValorMultiplicador(Multiplicador otroMultiplicador)`: Aumenta el valor interno del multiplicador con el de otro `Multiplicador`.
- **Operaciones combinadas**:
  - `aumentarPuntaje(int incrementador, Multiplicador multiplicador)`: Suma un incremento al puntaje actual, considerando el multiplicador dado.
  - `calcularPuntaje()`: Calcula el puntaje final multiplicando el valor del `Chip` por el `Multiplicador`.

La clase **Puntaje** proporciona una clara y eficiente forma para manipular la puntuación, asegurando que los valores sean consistentes y que las interacciones entre sus componentes se manejen correctamente.

6.6.2. Chip

La clase **Chip** representa el componente base del sistema de puntuación, encargándose de almacenar y manipular un valor numérico que sirve como puntaje inicial antes de ser afectado por un `Multiplicador`.

Atributos

- `valor`: Un entero que representa el valor numérico asociado al `Chip`.

Métodos principales

- `sumar(Chip otroChip)`: Devuelve un nuevo `Chip` cuya suma es el resultado de combinar el valor del `Chip` actual con el de otro `Chip`.
- `multiplicarCon(int multiplicador)`: Multiplica el valor del `Chip` por el valor proporcionado, devolviendo el resultado como un entero.

La clase **Chip** actúa como el punto de partida en el cálculo de puntuaciones, proporcionando una abstracción simple y eficiente para manipular valores individuales. Su integración con la clase `Puntaje` asegura que los cálculos más complejos sean correctamentes delegados, manteniendo una separación de responsabilidades adecuada dentro del modelo.

6.6.3. Multiplicador

La clase **Multiplicador** representa el factor que se utiliza para modificar el valor del `Chip` en el sistema de puntuación. Su principal objetivo es proporcionar operaciones matemáticas que permitan la combinación y escalamiento de puntuaciones.

Atributos

- `valor`: Un entero que representa el factor multiplicador actual.

Métodos principales

- `multiplicarCon(Chip chip)`: Devuelve el producto del valor del `Multiplicador` y el valor del `Chip` proporcionado.
- `sumarMultiplicador(Multiplicador otroMultiplicador)`: Devuelve un nuevo `Multiplicador` que resulta de sumar el valor actual con el de otro `Multiplicador`.
- `multiplicarCon(Multiplicador otroMultiplicador)`: Devuelve un nuevo `Multiplicador` que resulta de multiplicar el valor actual con el de otro `Multiplicador`.

La clase **Multiplicador** es clave para la lógica del sistema de puntuación, ya que permite escalar y combinar valores de manera flexible. Su interacción con la clase `Chip` asegura que los cálculos complejos de puntuación sean realizados con precisión y claridad, ofreciendo una delegación sencilla para gestionar operaciones.

7. Excepciones

Para este trabajo, no fueron creadas excepciones nuevas, pero si se hizo uso de las siguiente excepciones propias de Java: **FileNotFoundException** en Juego, **IllegalArgumentException** en JSONReader y en factories, **RuntimeException** en factories e **IllegalStateException** en mano.

FileNotFoundException fue utilizada en Juego ya que esta clase busca el archivo JSON adecuado al llamar a su constructor, y si dicho archivo no se encuentra presente o no es accesible, se produce la excepción.

IllegalArgumentException es utilizada en el JSONReader y las factories ya que estas clases esperan argumentos específicos para instanciar las clases deseadas, y si los argumentos no son válidos y no se hallan en el alcance del dominio entonces podemos manejarlos con esta excepción.

RuntimeException es genérica, pero fue utilizada en JokerFactory para manejar el caso en que el efecto descrito en el JSON fuese inválido y se hallase fuera del alcance del dominio del proyecto.

IllegalStateException Es utilizada en Mano en caso de que se intenten agregar cartas a esta cuando ya hay 5 cartas en su lista. Está siendo utilizada para impedir una acción no deseada basada en el estado de la clase.