FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 1
Segundo cuatrimestre de 2024

Grupo 11

Nombre y Apellido	Email	Padrón
Juan Balella	jbalella@gmail.com	110271
Héctor David Condori	hcondori@fi.uba.ar	104518
Matías Bagatto	luis.rodriguez@mail.com	108812
Valentín Gabriel Somoza	vsomoza@fi.uba.ar	109188
Giuseppe Avelino Manuel Prieto Bonilla	pedro.lopez@mail.com	56789012

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III, que consiste en desarrollar un juego similar a **Balatro** en **Java** utilizando **JUnit 5**, **JavaFX**, **Maven**, **GitHub Actions** y los **conceptos** del paradigma de la orientación a objetos vistos hasta ahora en el curso.

2. Supuestos

- Supuesto 1: Siempre el jugador sumara puntos en una jugada ya que existe "Hig card çomo mano.
- Supuesto 2: El jugador pierde cuando se queda sin Manos y no llego al puntaje objetivo de la Ronda.
- Supuesto 3: La Tienda ni el Jugador poseen Dinero por lo tanto los Jokers y Tarots en la Tienda son gratis.
- Supuesto 4: Cuando el jugador pierde se cierra el juego.
- Supuesto 5: Cuando el jugador gana puede volver a jugar.
- Supuesto 6: Se puede modificar los valores de los archivos .json.

3. Diagramas de Clases

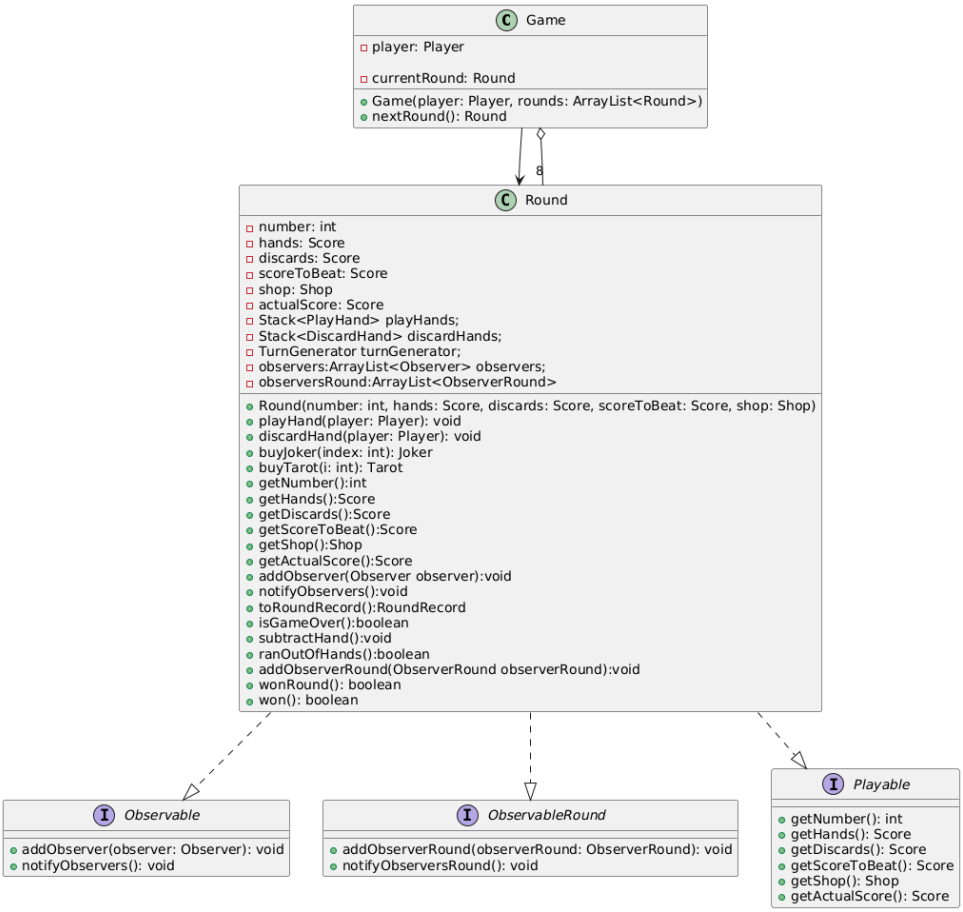

Figura 1: Diagrama de Game - Round

En la Figura 1 se puede observar la clase principal Game que contiene toda la lógica del juego. Primeramente tiene como atributo a un ArrayList<Round>que lo inicializa con 8 Rounds dentro. Estas Round se van iterando a medida que el jugador va pasando de Ronda. Luego tiene otra Round pero es para indicar la Ronda actual y es solamente para facilitarnos las funcionalidades.

Round es uno de los Objetos mas extensos ya que posee toda la información de la ronda. Implementa 3 interfaces que le permiten ser observada por otros objetos, los observadores son principalmente los objetos que pertenecen a la interfaz gráfica para que puedan actualizar sus datos. Luego implementa Playable que simplemente es para establecer un contrato y el código sea mas legible.

Game también posee un Player que es el jugador en cuestión. Este objeto objeto se explica mas adelante.

Figura 2: Diagrama de Player

En la figura 2 tenemos a Player que es uno de los objetos de mas alto nivel. Esta clase se encarga de administrar las acciones del usuario y delegando las decisiones a otras clases para que las ejecuten. El Player posee como atributo a EnglishDeck que es el mazo de cartas inglesas y también implementa interfaces para aplicar correctamente el patrón Observer y brindarle información actualizada a la interfaz gráfica. Todos los métodos que tiene Player son para administrar las cartas que depende de la decisión del usuario.

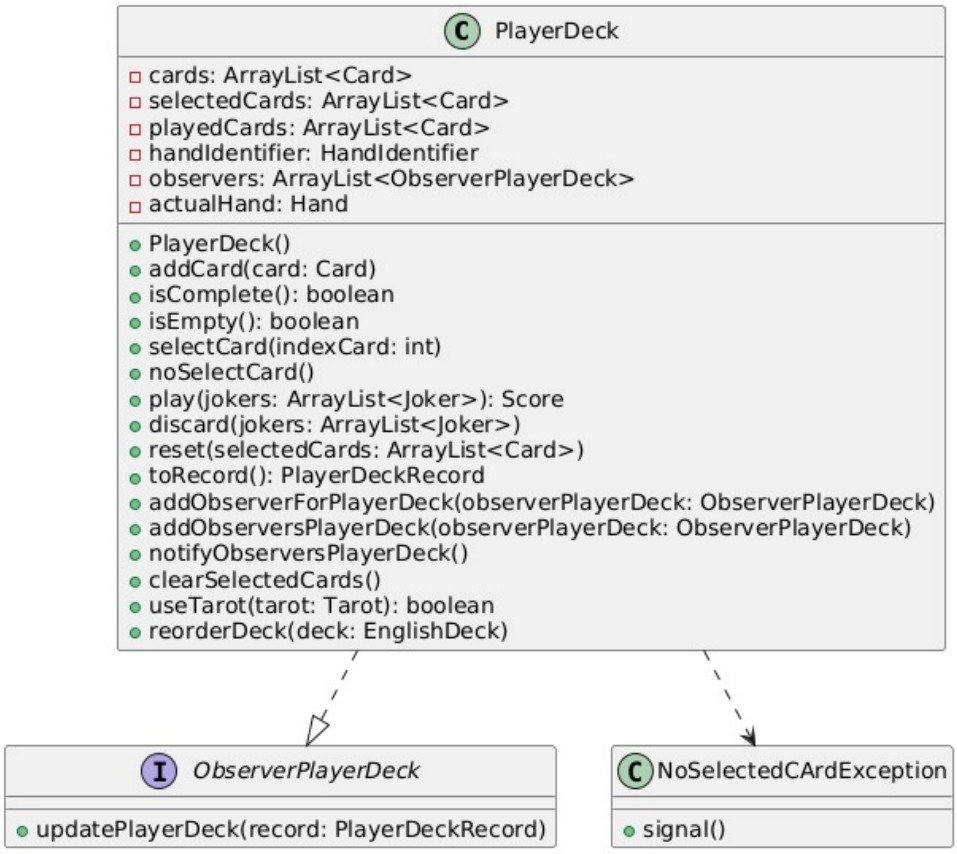

Figura 3: Diagrama de PlayerDeck

En la figura 3 podemos observar a PlayerDeck, este objeto representa las 8 cartas que el Jugador puede observar en la partida. Son las cartas que se le reparten para que el decida que hacer. Posee como atributos a HandIdentifier y a Hand. La responsabilidad de este objeto es administrar correctamente las cartas que el jugador 'clickea' para seleccionarlas para luego decidir si descartar o jugar esa mano. Una vez que el jugador selecciona las cartas con las que va a hacer algo PlayerDeck delega la jugada a HandIdentifier.

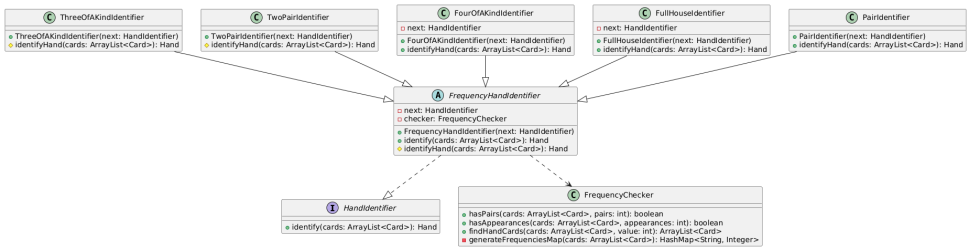

Figura 4: Diagrama de HandIdentifier - FrequencyHandIdentifier - FrequencyChecker

En la figura 4 podemos ver a FrequencyHandIdentifier que implementa HandIdentifier, esto es para manejar el flujo de manos posibles.

La clase HandIdentifier y sus implementaciones tienen como objetivo identificar el tipo de mano de cartas (como Full House, Poker, Escalera, etc.) en un conjunto de cartas. Para hacer esto de manera flexible y extensible, se utiliza el patrón Chain of Responsibility. La clase HandIdentifier tiene una propiedad next, que es una referencia al siguiente identificador de manos en la cadena. Esto permite que el flujo de identificación de la mano

se pase de un identificador a otro si el identificador actual no es capaz de identificar la mano. La clase FrequencyChecker trabaja con un conjunto de cartas (ArrayList<Card>cards) para realizar verificaciones sobre cuántas veces aparece un determinado valor o tipo de carta. Esta verificación se hace utilizando un Map de frecuencias donde la clave es el valor de la carta y el valor es el número de veces que aparece esa carta en el conjunto.

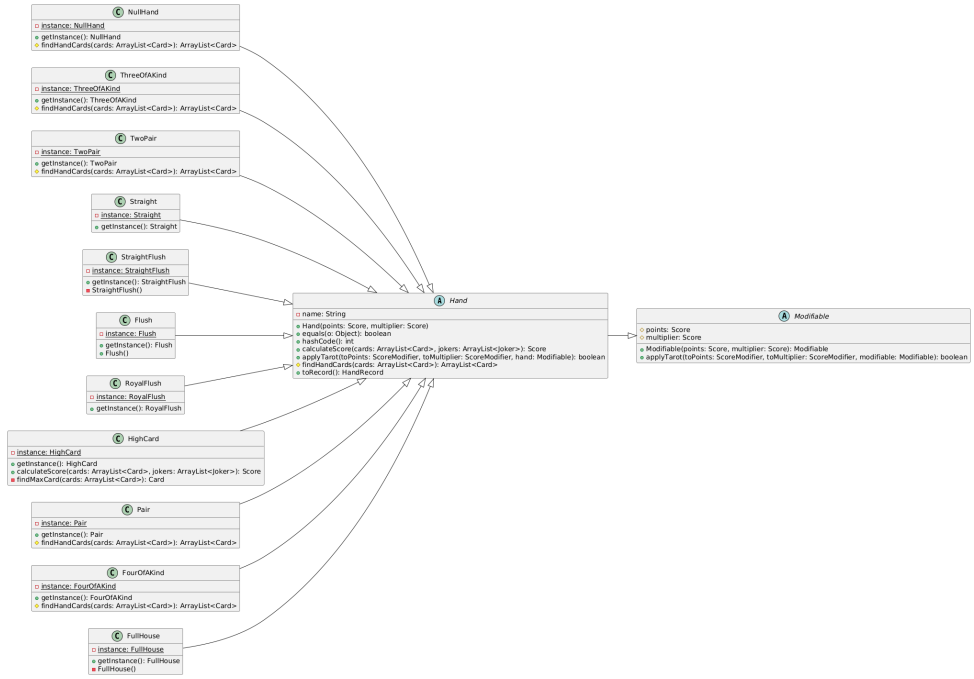

Figura 5: Diagrama de Hand

En la figura 5 tenemos la clase Hand que representa una mano de cartas que un jugador tiene, y se utiliza para evaluar o identificar las combinaciones de cartas que forman parte de la mano. Una Hand puede referirse a una combinación particular de cartas que puede tener un valor asociado (como en el póquer, donde una mano de cartas puede tener un puntaje determinado según las reglas).Hand tiene la responsabilidad de identificar el tipo de mano de un jugador y calcular su puntaje, lo que influye en la dinámica del juego. La estructura de clases, como HandIdentifier, permite que el proceso de identificación de manos sea flexible, modular y escalable, lo cual es una gran ventaja para implementar la lógica de diferentes combinaciones de cartas.

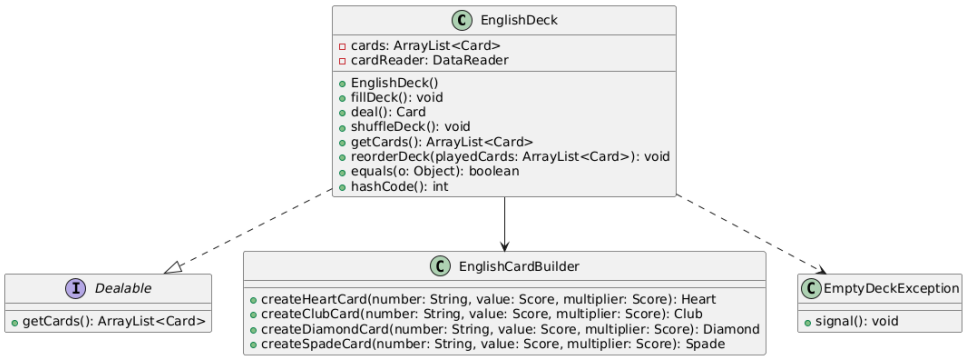

Figura 6: Diagrama de EnglishDeck

En la figura 6 se puede observar a EnglishDeck que implementa Dealable para implementar getCards(). También posee un ArrayList<Card>en donde se almacenan todas las cartas del juego, 52 en especifico. Posee un EnglishCardBuilder que es el encargado de crear las Cards aplicando el patron Factory Method. Este objeto hace referencia al mazo del juego y maneja todo lo que tiene que ver con ello. Luego posee como atributo a un objeto de tipo DataReader que se explica mas adelante.

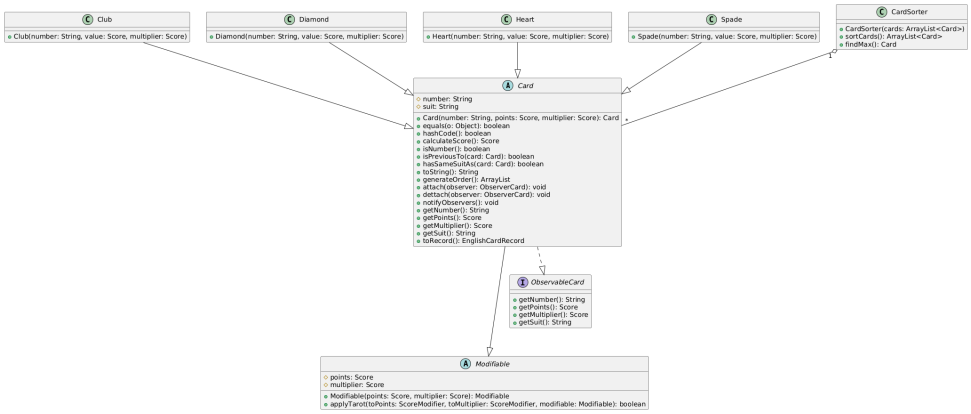

Figura 7: Diagrama de Card

En la figura 7 tenemos a Card que es una clase Abstracta que nos sirve para que implementen las subclases que representan a los Palos de cada una de ellas. Estas subClases asignan su atributo suit a su palo. Esto lo implementamos de esta manera para no pasar valores primitivos por parámetros a la hora de querer adicionar una funcionalidad.

Card hereda de Modifiable que es otra clase Abstracta que permite a Card tener un Score y ser Modificable ya que en tiempo de ejecución se modifica el valor del puntaje de las cartas. Card maneja todo lo que conlleva el relacionarse con otra Card o el manejo interno de sus atributos. Esta clase también implementa ObservableCard que es parte del patrón Observer para poder actualizar la información de la interfaz gráfica.

CardSorter es una clase concreta que forma parte de los atributos de Card y se utiliza para ordenar las cartas para implementación interna.

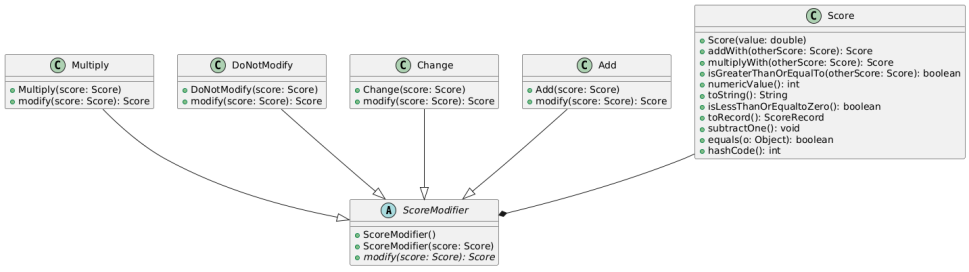

Figura 8: Diagrama de Score

En la figura 8 tenemos a Score que es utilizado por muchas clases ya que es el que se encarga de manejar el puntaje. Posee como atributo solamente a un value de tipo double el cual representa el numero del puntaje. Esta clase posee todos los métodos para relacionarse con otro Score y para ser mostrado por la interfaz gráfica. Score es utilizado también por ScoreModifier que es una clase abstracta que posee subclases encargadas de hacer

operaciones unicamente con Score se implemento de esta manera para facilitar la lectura del código y mantener un orden.

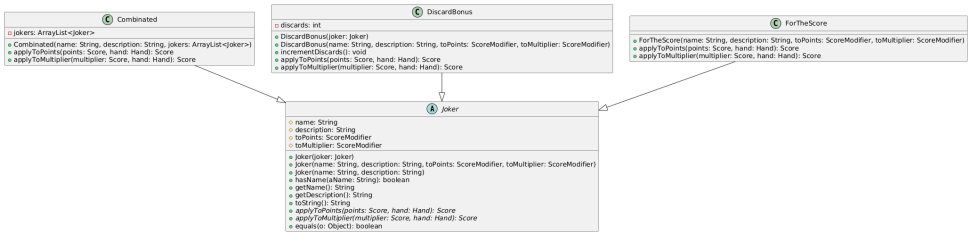

Figura 9: Diagrama de Joker

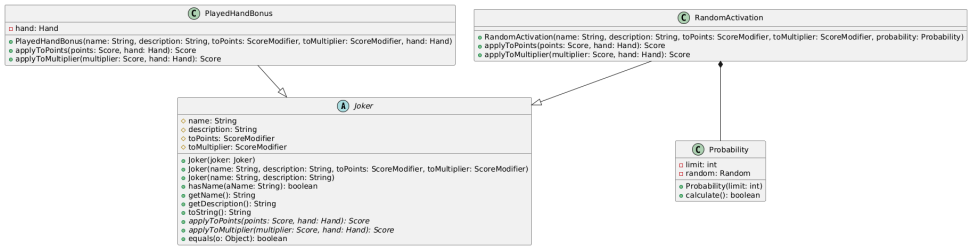

Figura 10: Diagrama de Joker

En la figura 9 y 10 podemos ver como esta compuesto Joker. Es una clase abstracta la cual hereda a sus subclases los comportamientos básicos de Joker. Las clases hijas que se observan son los tipos de Jokers que posee el juego. Probability es una clase encargada de generar un numero random para los Jokers que funcionan con probabilidades.

Figura 11: Diagrama de JokerReader

En la figura 11 podemos ver el diagrama de JokerReader que posee varias cosas. Primeramente JokerDeck que cumple la función de mazo de comodines, esta clase almacena todos los Jokers que lee JokerReader del archivo comodines.json. JokerReader se encarga de leer el .json justamente y hace uso de JokerCreator que utiliza el patrón Factory Method para crear todos los Jokers que se le piden. JokerData es el objeto que utilizamos para transferir todos los datos de model a la interfaz. Effect seria un objeto Data también y se encarga de identificar los efectos que aparecen en los archivos .json para modelarlos en el código.

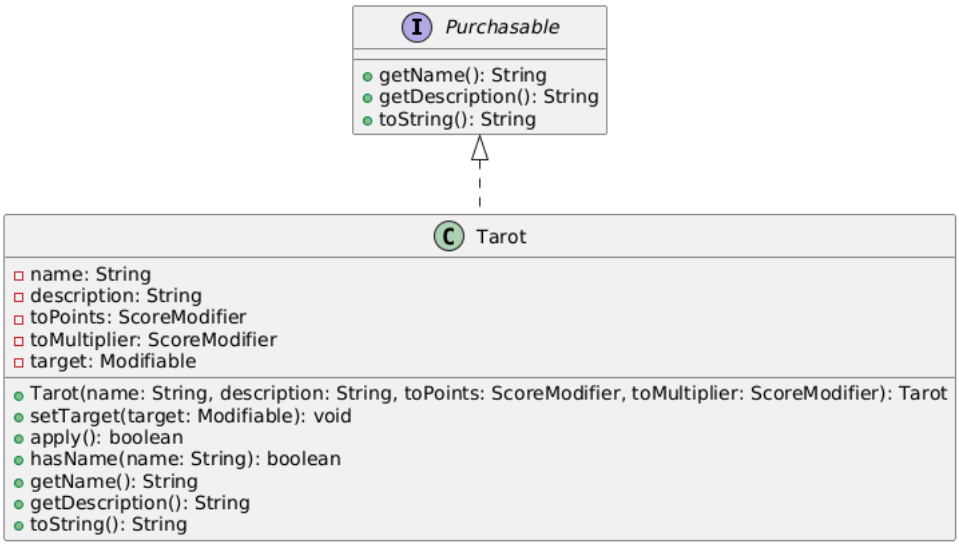

Figura 12: Diagrama de Tarot

En la figura 12 vemos Tarot que implementa Purchasable al igual que Joker. Esta clase modela la carta Tarot que fue previamente obtenida a partir del tarots.json. Todos los métodos que posee son para el manejo interno de la clase y para modificar una Card o una Hand.

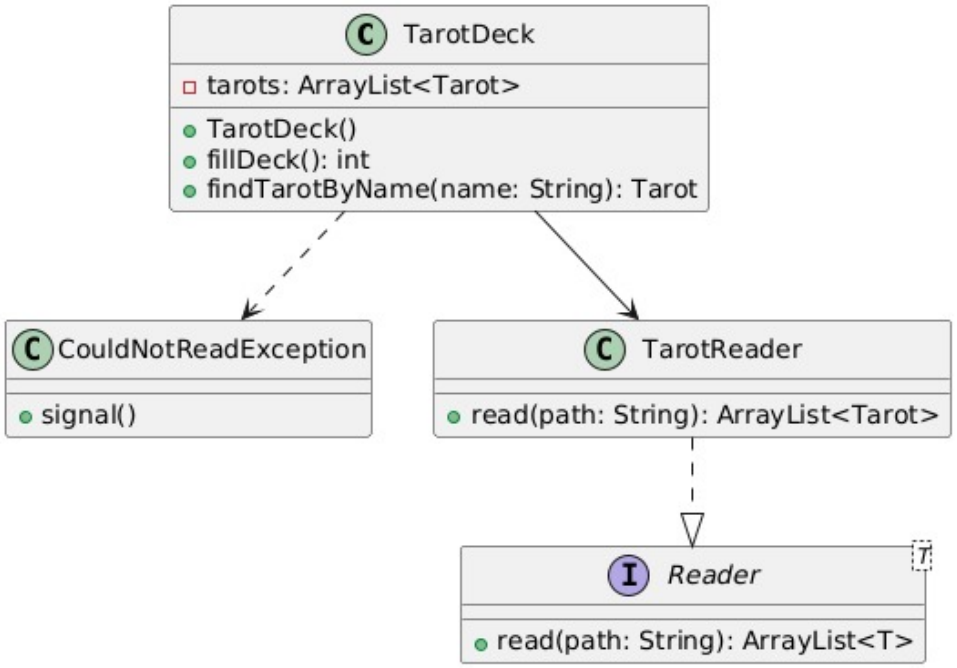

Figura 13: Diagrama de TarotDeck - TarotReader

En la figura 13 tenemos a TarotDeck que funciona como un mazo de todos los Tarots. TarotReader se encarga de leer el archivo tarots.json para crear todas las cartas.

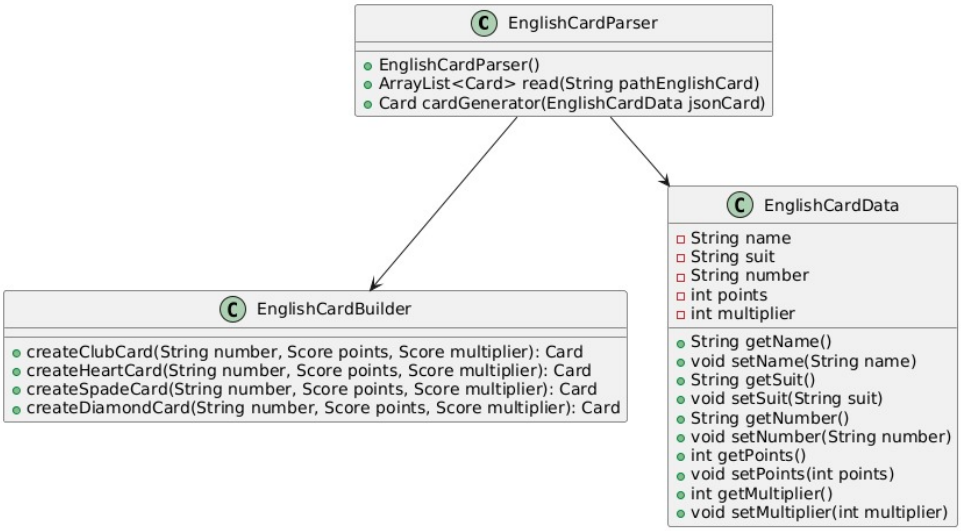

Figura 14: Diagrama de EnglishCardParser

En la figura 14 tenemos a EnglishCardParser el cual se encarga de leer el archivo balatro.json y crea las cartas con el EnglishCardBuilder que utiliza Factory Method y ya lo nombramos anteriormente. EnglishCardData es una clase destinada a comunicar los datos de model a la interfaz gráfica.

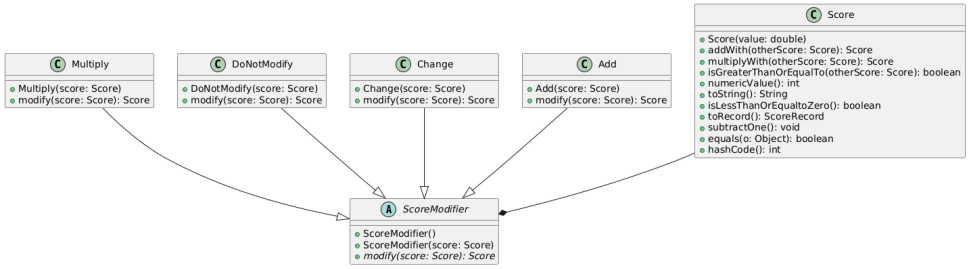

Figura 15: Diagrama de Shop

En la figura 15 tenemos a Shop clase concreta que se encarga de modelar la Tienda que utiliza el usuario antes de cada comienzo de ronda. Posee la carta, tarots y jokers que vende. Como el Juego no posee Dinero estos objetos simplemente pueden ser adquiridos de forma gratuita.

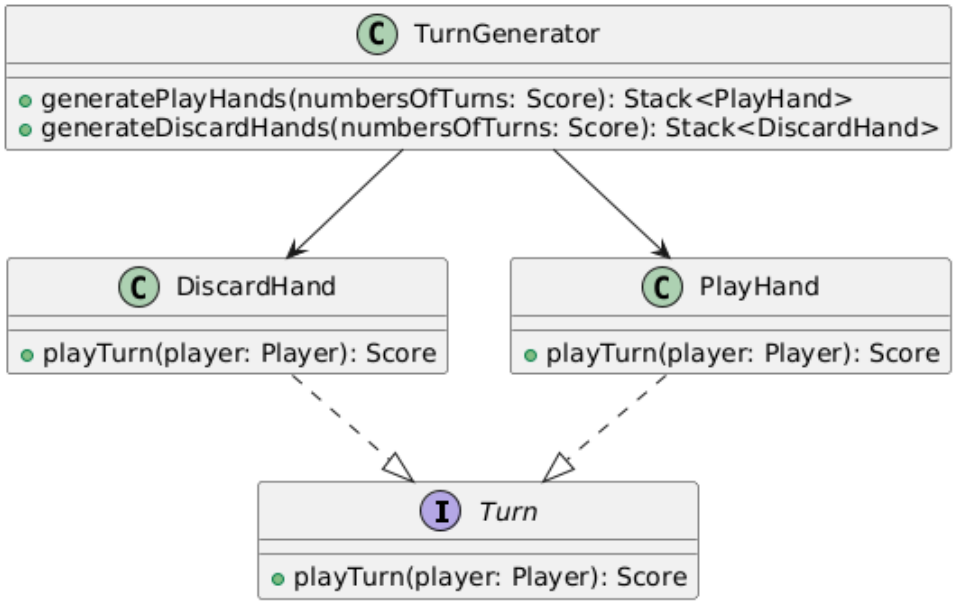

Figura 16: Diagrama de TurnGenerator

En la figura 16 tenemos a TurnGenerator que modela el comportamiento de turnos. Esta clase es capaz de generar Jugadas o Descartes que a su vez DiscardHand y PlayHand implementan Turn. La interfaz simplemente obliga a que implementen playTurn() para realizar su ejecución.

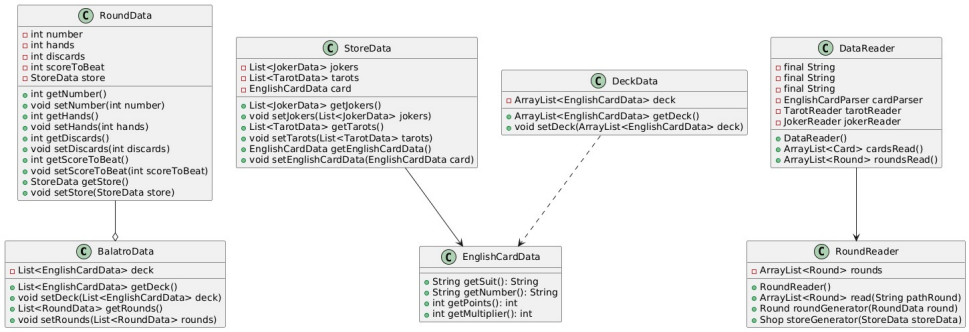

Figura 17: Diagrama de todos los Objetos Data

En la figura 17 tenemos a todos los objetos Data que sirven para la librería Jackson, estos objetos mapean el json para luego crear objetos que si nos sirven para modelar en el código.

Todos los objetos que son Readers lo implementamos con la librería Jackson. Para poder leer el json, creamos una clase que mapea la información que vamos a leer de json y con la información de esa clase vamos creando nuestra clases como carta, joker, tarot, etc.

4. Diagramas de Secuencia

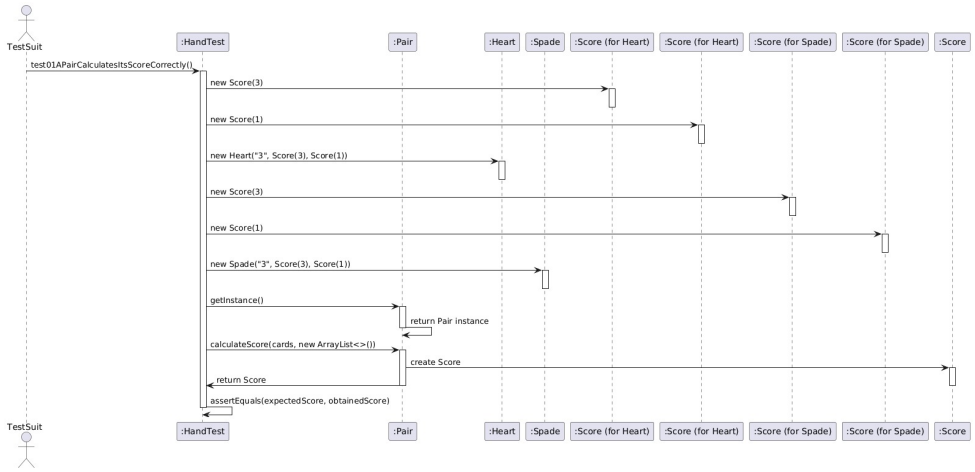

Figura 18: Diagrama de Secuencia Test 1 de Hand

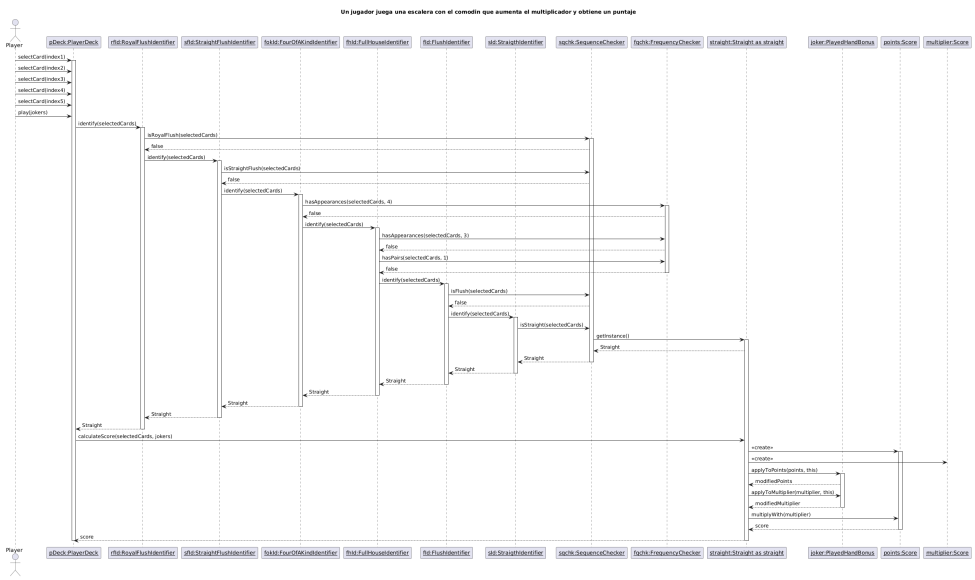

Figura 19: Diagrama de Secuencia caso de uso 2 de la Entrega 2

5. Diagrama de Paquetes

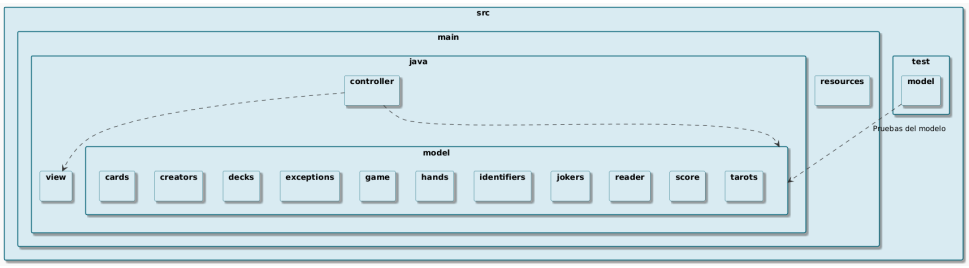

Figura 20: Diagrama Paquetes Principales

En la figura 19 podemos observar el nivel de acoplamiento del proyecto. Este diagrama nos da una vista rápida de donde se encuentra cada cosa. Principalmente existe una relación entre los paquetes model, view, controller y resources que se encargan de toda la parte gráfica. Luego dentro de model es donde esta la lógica completa del proyecto. Se ve bastante resumido por que si mostramos las clases seria poco legible.

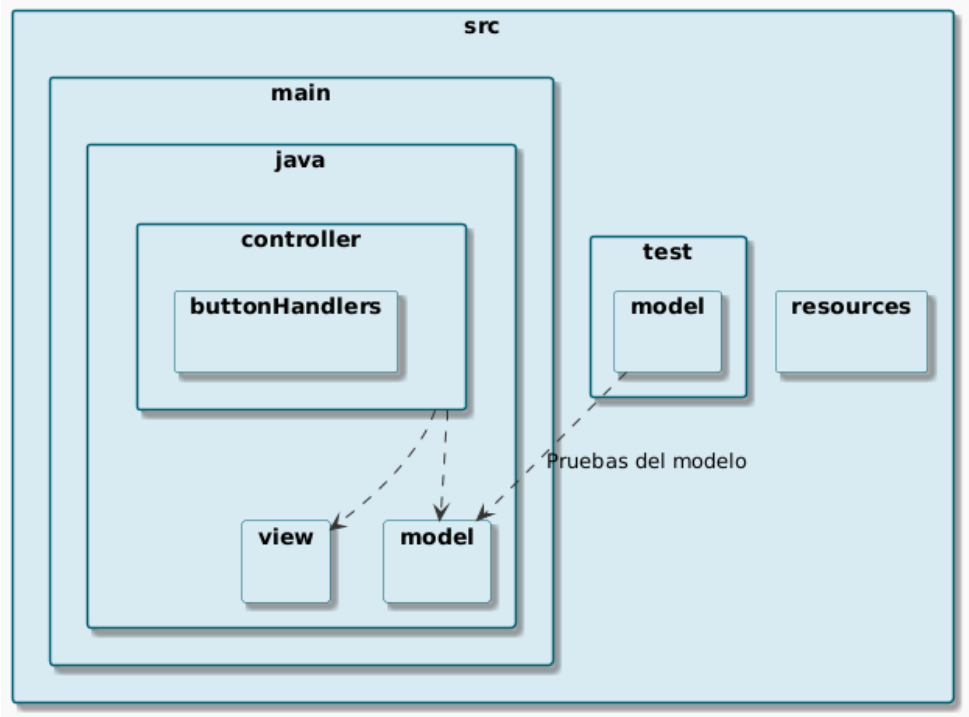

Figura 21: Diagrama de Paquetes

En la figura 20 podemos ver un subpaquete que falta en el paquete Controller.

6. Detalles de Implementación

Para implementar la interfaz partimos desde el patrón MVC que nos permite ordenar el código con funcionalidad. La interfaz hecha con javafx nos permitió subir imágenes y facilitarnos mucho el trabajo. Ademas otro concepto clave fue la utilización del patrón Observer que lo utilizamos para actualizar la información de View.

Las Manos Posibles dadas por consigna tuvimos que re diseñarlas varias veces y nos quedamos con un modelo que utiliza Chain of Responsibility creando cada mano posible como una clase que hereda de la clase Hand. Esta Herencia se aplico para hacer programación por diferencia y facilitar el manejo de las mismas mediante el principio de sustitucion de Liskov. Cada clase es instanciada una única vez en PlayerDeck a traves de un atributo de tipo HandIdentifier. Entonces cada vez que PlayerDeck realiza una jugada con las cartas seleccionadas este le delega la tarea a su atributo HandIdentifier que mediante Chain of Responsibility itera en cada mano posible delegando el conjunto de cartas a la mano posible. Esto permite que cada mano posible verifique si el conjunto de cartas es una jugada de su tipo. En el caso que no lo sea retorna el valor (null) pasado por parámetro.

6.1. Justificación del uso de Herencia vs Delegación

Entendemos que debemos favorecer la Delegación antes que la Herencia ya que la Herencia es una relación muy fuerte que en un caso futuro podría traer problemas ya sea al refactorizar o intentar simplemente añadir una nueva utilidad.

A continuación se explican los casos particulares en los que se aplico Herencia por encima de Delegación:

Hand En este caso, dado que todas las manos van a compartir los mismos atributos (puntos y multiplicador), y la forma en que se calculan sus puntajes es la misma, optamos por aplicar herencia. Ademas, la forma de compararse es igual para todas, al igual que aplicar el efecto de un `Tarot`.

Card Al igual que `Hand`, comparten tanto puntos como multiplicador. La forma de calcular su puntaje es igual independientemente de si se trata de una Card de palo `Club`, `Diamond`, `Heart` o `Spade`.

Joker Ademas de compartir las mismas características que los casos anteriores, todos los `Joker` comparten la característica de que son objetos que se pueden adquirir desde el `Shop`, por lo que vale la pena agruparlos en una misma clase madre.

FrequencyHandIdentifier En este caso, se utiliza una clase madre base en el patrón de diseño *Chain of Responsibility*. Los `HandIdentifier` que hereden de esta clase, tendrán los mismos atributos y en particular cada uno de ellos deberá implementar el método `identifyHand()`.

SequenceHandIdentifier Pasa lo mismo en este caso, esta clase es una clase madre base en el patrón de diseño *Chain of Responsibility* y comparte las mismas características que en el caso anterior.

ScoreModifier La principal motivación para heredar en este caso es que facilita la forma en que los `Joker` y `Tarot` se inicializan, aportando mayor legibilidad en esas clases.

Modifiable Este caso esta muy relacionado al anterior, ya que al heredar nos permite modificar el puntaje de una `Card` o una `Hand` mediante un `Tarot`. Como ambas tienen puntos y multiplicadores en sus atributos, optamos por agruparlos y hacer todo lo que tenga que ver con la modificación del puntaje mediante esta clase.

Principios de Diseño Aplicados

A continuación se explican los principios de diseño aplicados en el proyecto:

Principio de responsabilidad única Este principio lo aplicamos en muchos objetos para que el código sea más fácil de entender, más sencillo de mantener y aumentar la modularidad. Se aplicó en clases como `Score`, `Player`, entre otras. En cada clase, podemos encontrar varios métodos que realizan distintas tareas, pero lo importante es que cada método tiene una única responsabilidad.

Principio Abierto-Cerrado Lo implementamos mediante el uso de varias interfaces y el aprovechamiento del polimorfismo. Las clases que implementan este principio buscan tener la mayor cohesión posible y un bajo acoplamiento. De esta manera, las funcionalidades quedan abiertas a la extensión y cerradas para la modificación.

Principio de sustitución de Liskov Este principio fue aplicado principalmente a través de una correcta implementación de la herencia y aprovechando al máximo el polimorfismo. Fue fundamental para la implementación del patrón *Observer* y el funcionamiento general del código.

Principio de segregación de interfaz Lo implementamos en todas las interfaces, ya que cada una establece un contrato de pocos métodos. Esto permite que las clases que implementan esas interfaces no tengan que cumplir con un contrato extenso que no utilizarían.

Principio de inversión de dependencias Este principio no lo pudimos implementar tanto debido a que no realizamos muchas refactorizaciones, principalmente por la limitación de tiempo.

6.2. Patrones de Diseño Utilizados

MVC (Modelo Vista Controlador) Este patrón de diseño nos permite separar las distintas responsabilidades dentro del juego. En este caso la utilizamos sobre una interfaz gráfica.

Observer Nos permite actualizar la información del Model para que la reciba el Controller y poder actualizar View. Hacemos que las Clases pertenecientes a Model sean 'observables' y en donde se necesita se implementa 'observador'.

Strategy Este patrón de diseño nos fue de mucha utilidad a la hora de definir el comportamiento de cada `Joker` y de cada `Tarot` en particular. A partir de este contexto, se define una clase abstracta `ScoreModifier`, y sus clases derivadas (`Add`, `Change`, `DoNotModify` y `Multiply`) son quienes modifican un `Score` aplicando sus respectivas estrategias. Esto nos permite que, si en un futuro cambia el comportamiento de algún `Joker` o `Tarot`, solo basta con designarle alguna estrategia o crear nuevas clases que representen la estrategia requerida.

Chain of Responsibility Al momento de identificar que mano forman las cartas elegidas por el `Player`, una forma flexible de hacerlo es mediante este patrón de diseño. Se establece como primer eslabón de la cadena al `RoyalFlushIdentifier`, quien tendrá una referencia a su siguiente eslabón, el `StraightFlushIdentifier`, y así sucesivamente hasta llegar al `HighCardIdentifier` que, no por casualidad, es la mano que menor puntaje posee. Definir este orden nos permite darle prioridad a las mejores manos, y que dadas las cartas a identificar, siempre se obtenga la mejor mano posible.

Factory Method Lo utilizamos para crear los objetos en Model a partir del Parser que lee los archivos .json. De esta manera se instancia al creador para crear el objeto en cuestión.

7. Excepciones

Las Excepciones que implementamos solamente se muestran por consola. Ninguna se visualiza en la interfaz gráfica ni se las maneja en ella.

CouldNotReadException Excepción que se lanza cuando no se puede leer algún archivo .json, ya sea porque no lo encontró o el archivo es invalido.

EmptyDeckException Esta excepción se lanza cuando se intenta pedir una carta al mazo de cartas inglesas. Si este esta vació lanzara esta excepción

EmptyPlayerDeckException Esta excepción se lanza cuando se intenta jugar una mano o descartar una mano y el PlayerDeck esta vació.

InvalidJokerException Esta excepción se lanza cuando se intenta añadir un Joker a la posesión del Jugador pero este Joker es nulo o el Jugador ya posee 5 Jokers.

InvalidTarotException Se lanza cuando se intenta añadir un Tarot al Jugador pero este ya posee 2 o el Tarot es nulo.

NotSelcetedCardsException Se lanza cuando se intenta hacer una jugada o un descarte pero anteriormente no se seleccionaron cartas para dichas acciones.