[75.07 / 95.02 / TB025]

Algoritmos y programación III
Paradigmas y programación

Trabajo práctico 2: Balatro

Estudiantes:

Nombre	Padrón	Mail
Valentino Carmona	110113	vcarmona@fi.uba.ar
Sebastian Loe	110106	sloe@fi.uba.ar
Nicolas Darnay	107469	ndarnay@fi.uba.ar
Fabrizio Lamanna	111049	fmlamanna@fi.uba.ar
Ivan Da Ruos	111567	idaruos@fi.uba.ar

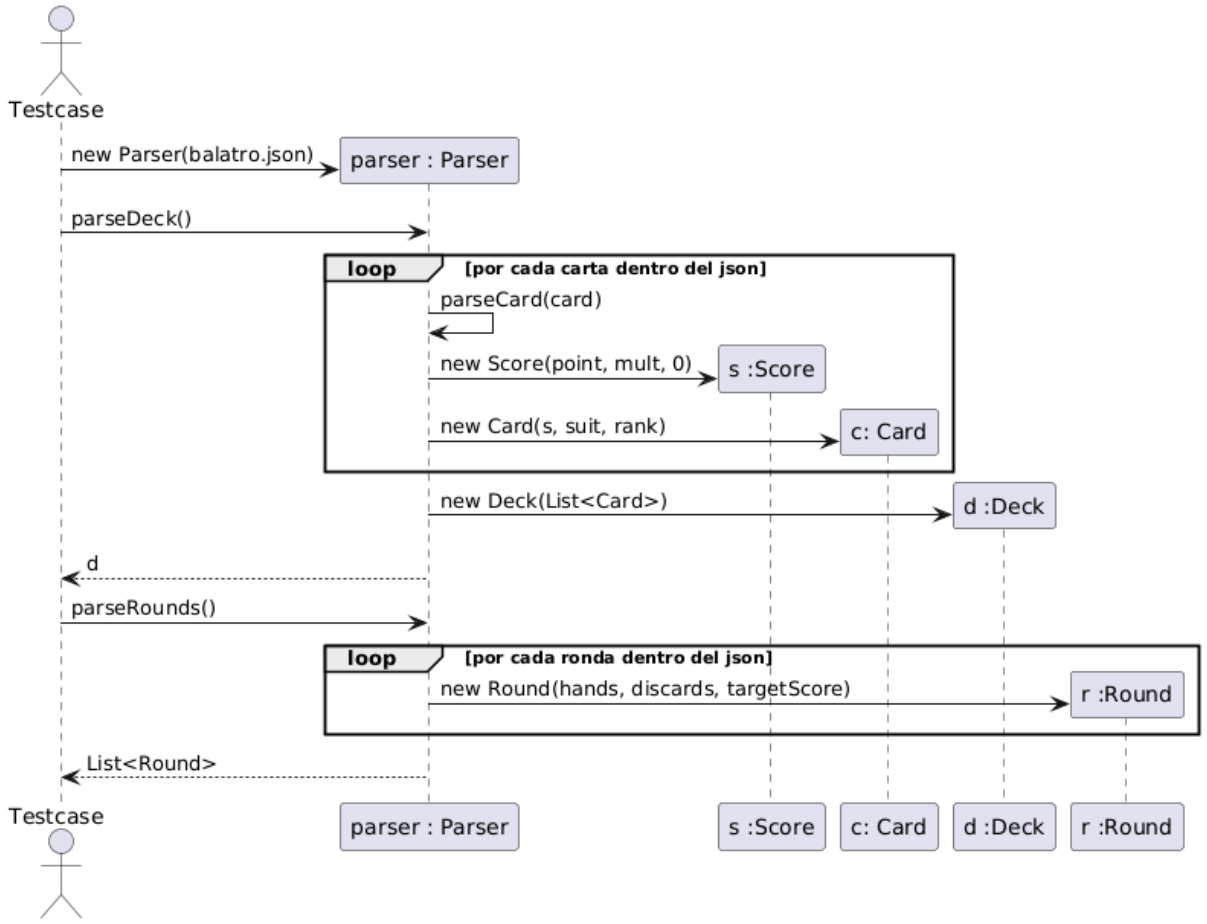
Tutor: Pablo Suárez, Joaquin Rivero

Nota Final:

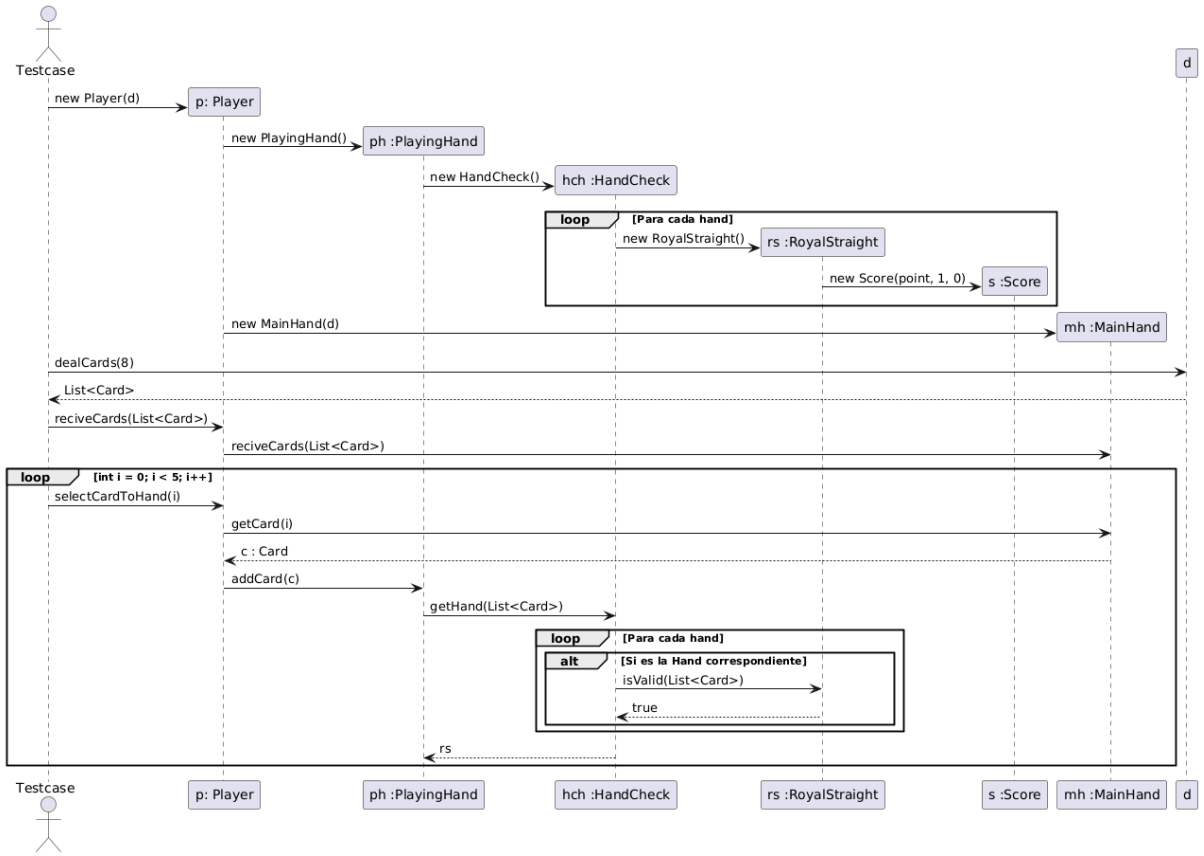
Supuestos:

- Suponemos que el puntaje internamente se calcula sumando al mult y luego multiplicando al mult
- Los Jokers RandomDestroy se aplican antes de destruirse
- En El parser, suponemos que los Jokers que se aplican de forma aleatoria o que se destruyen de forma aleatoria sólo pueden tener el ScoreJokerStrategy, esta decisión la tomamos en base al archivo balatro.json que es usado de seed.

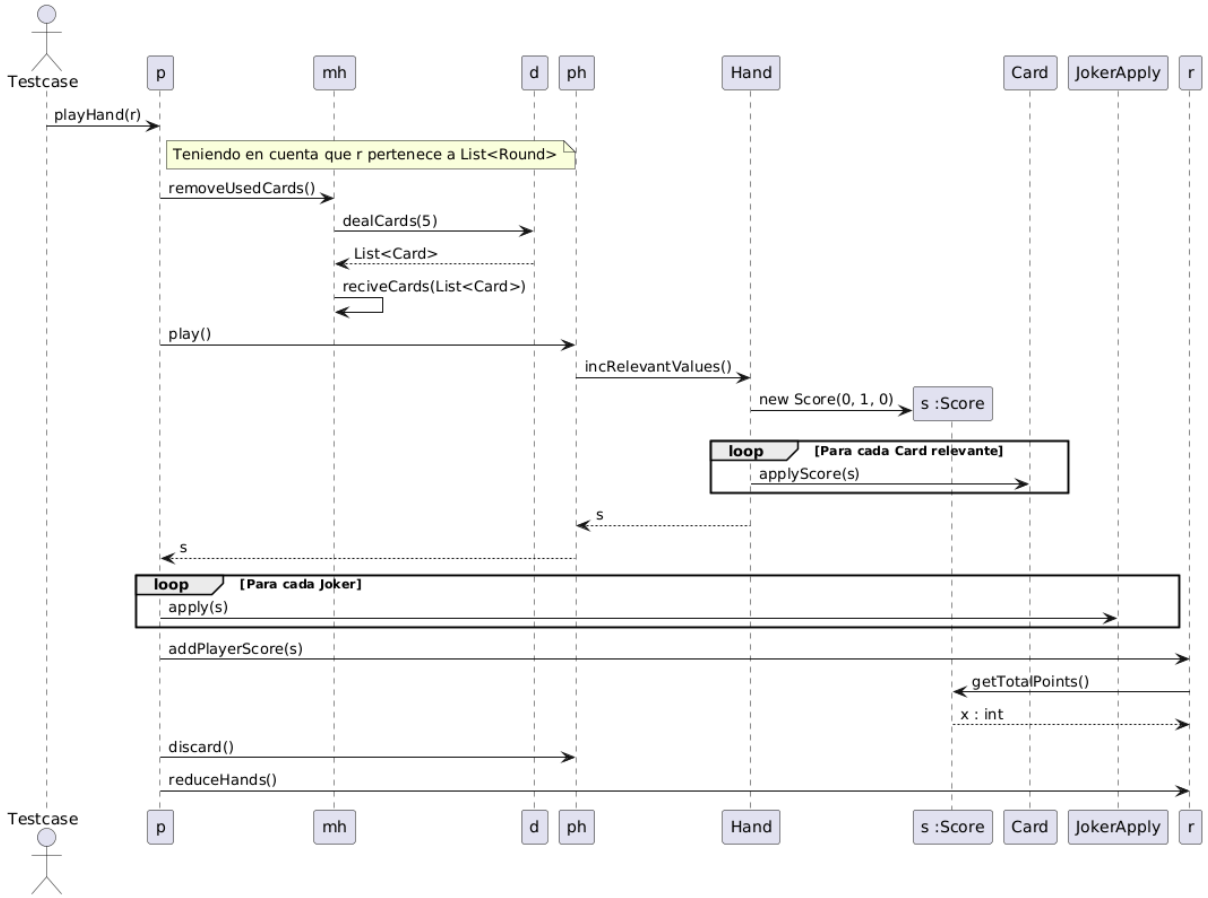
Diagramas de secuencia:



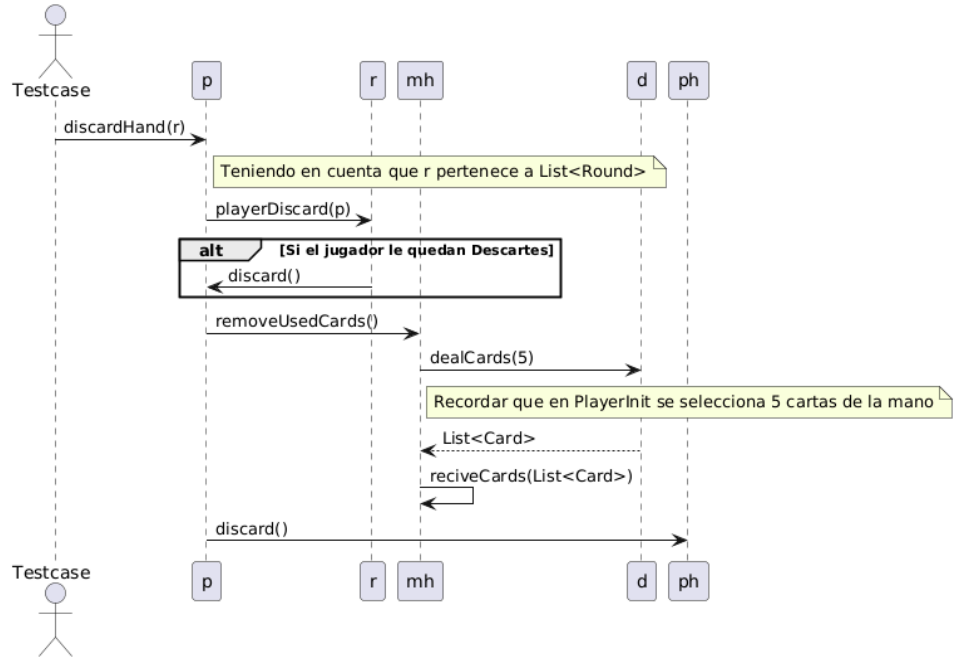
Parseo e Inicialización de Round y Deck usando archivo semilla json.



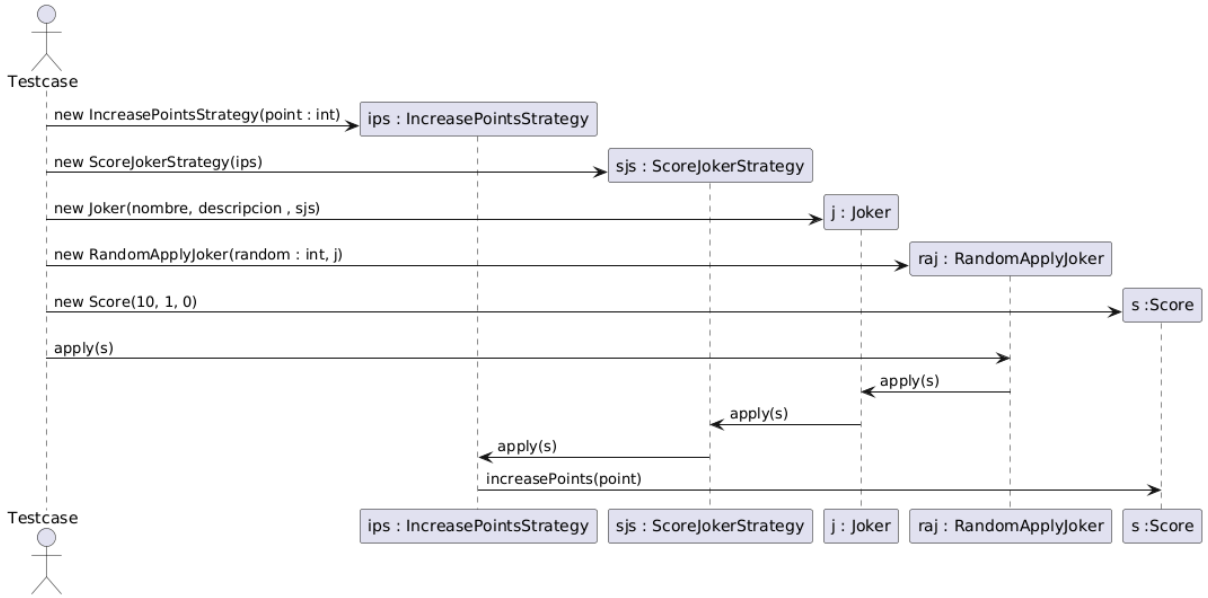
Creación de Player y Chequeo de tipo de Mano seleccionada.



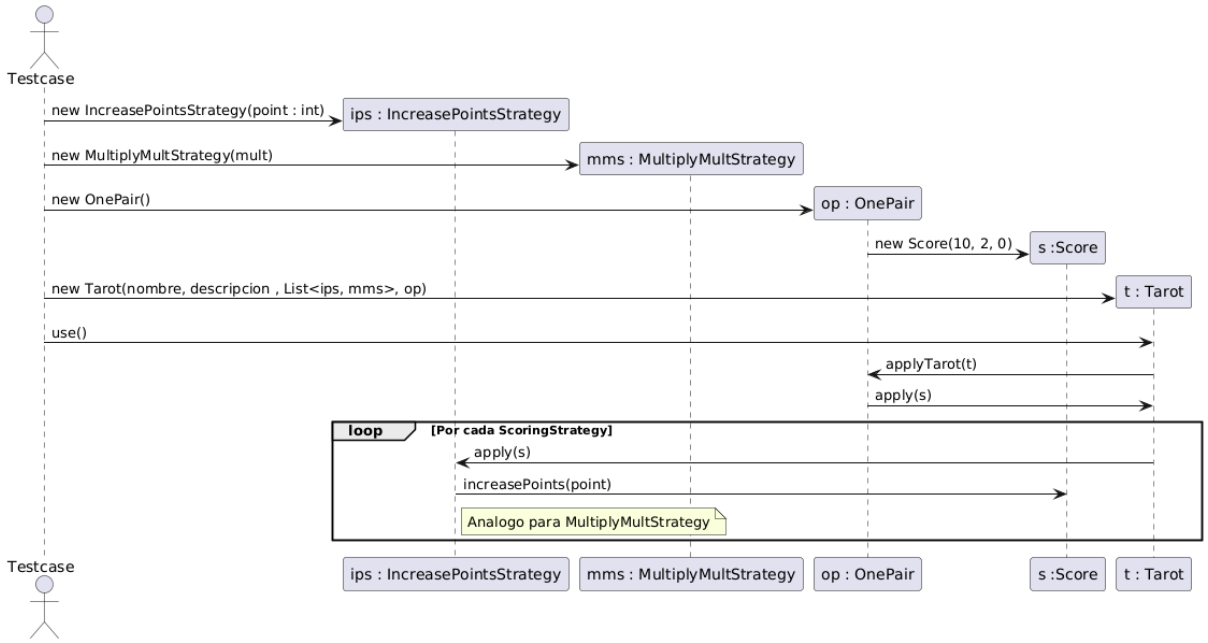
Secuencia de Jugar una mano calculando su puntaje total.



Secuencia de Descarte de una mano.

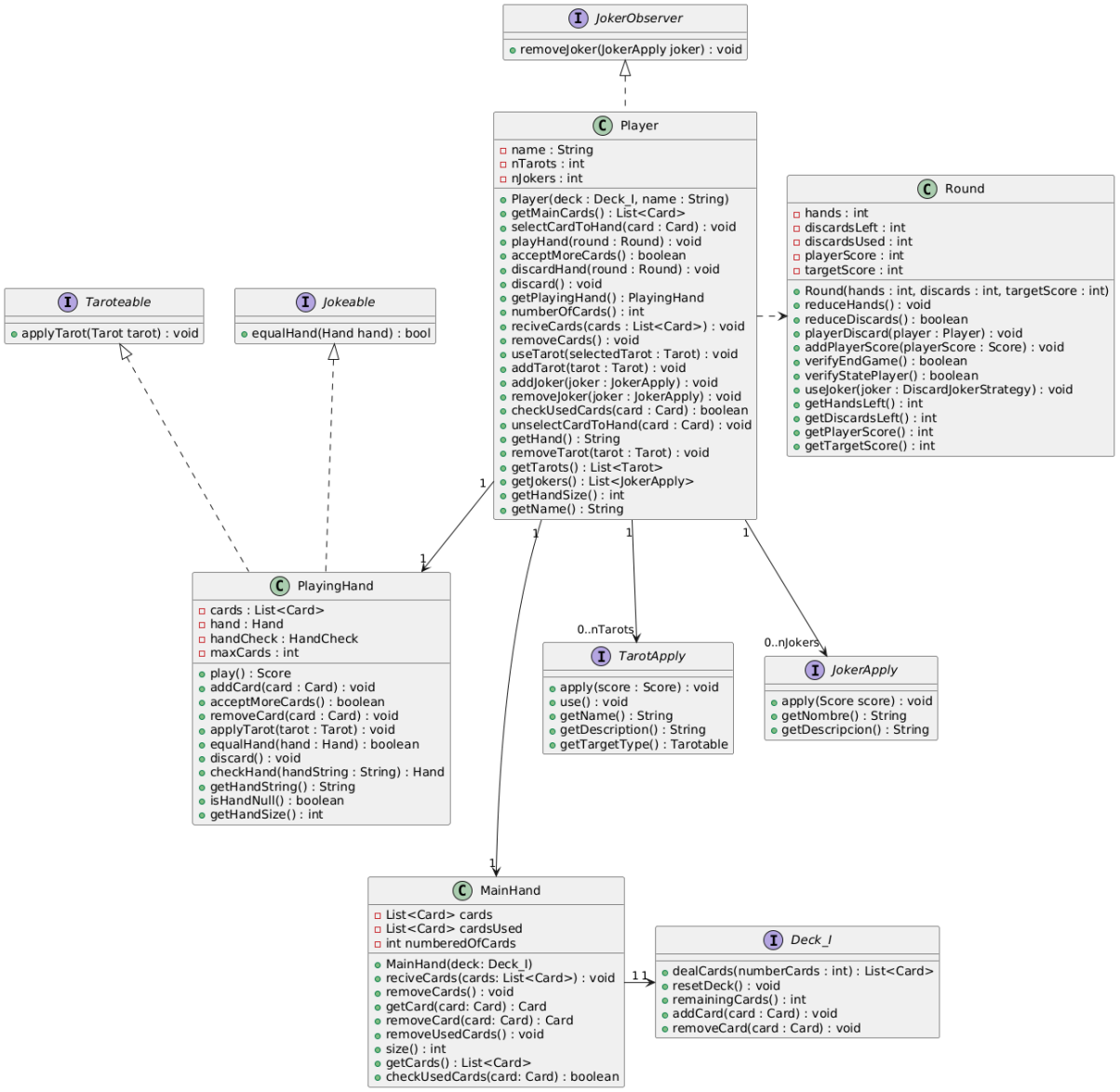


Creacion y aplicacion de un Joker sobre un Score.

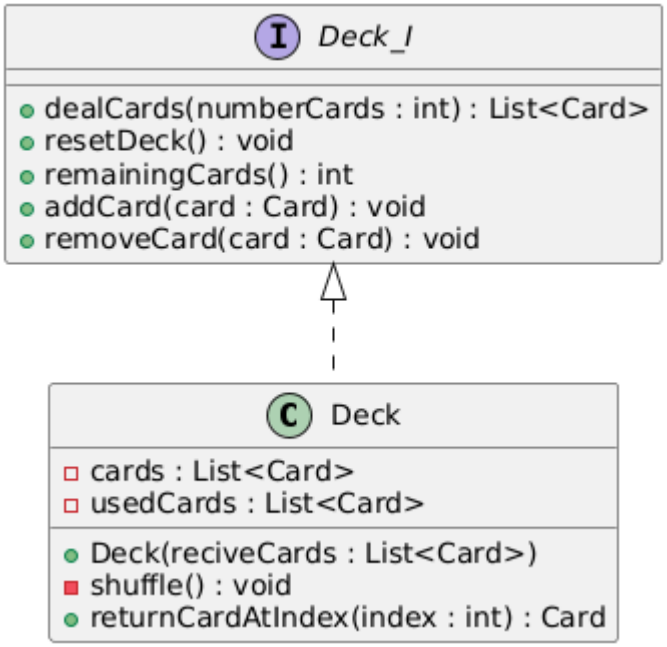


Creacion y aplicacion de un Tarot sobre una Hand.

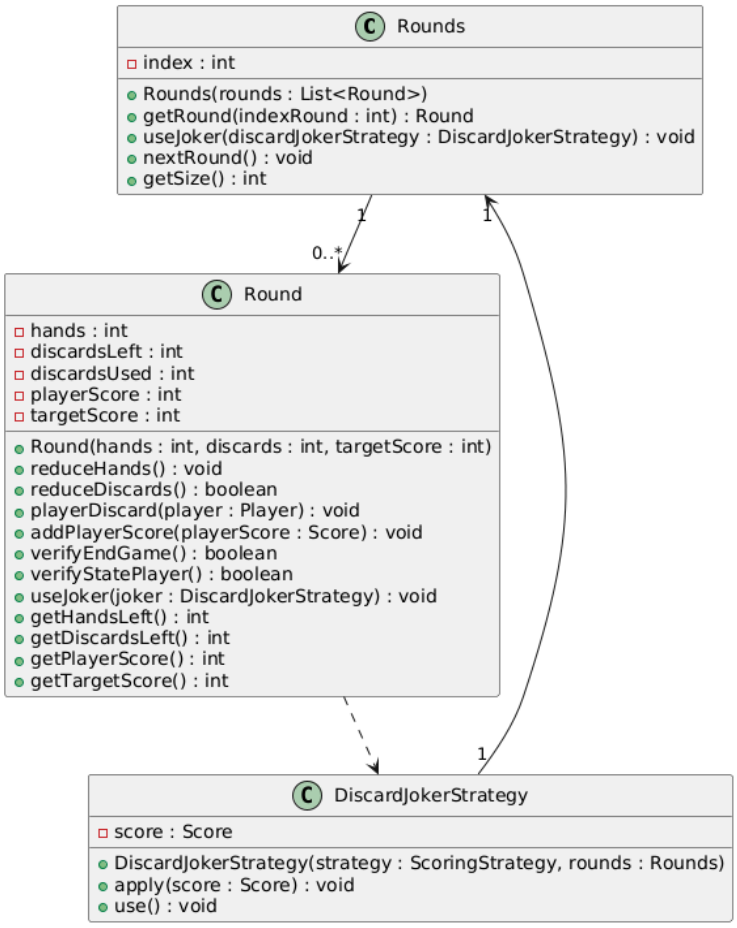
Diagramas de clase:



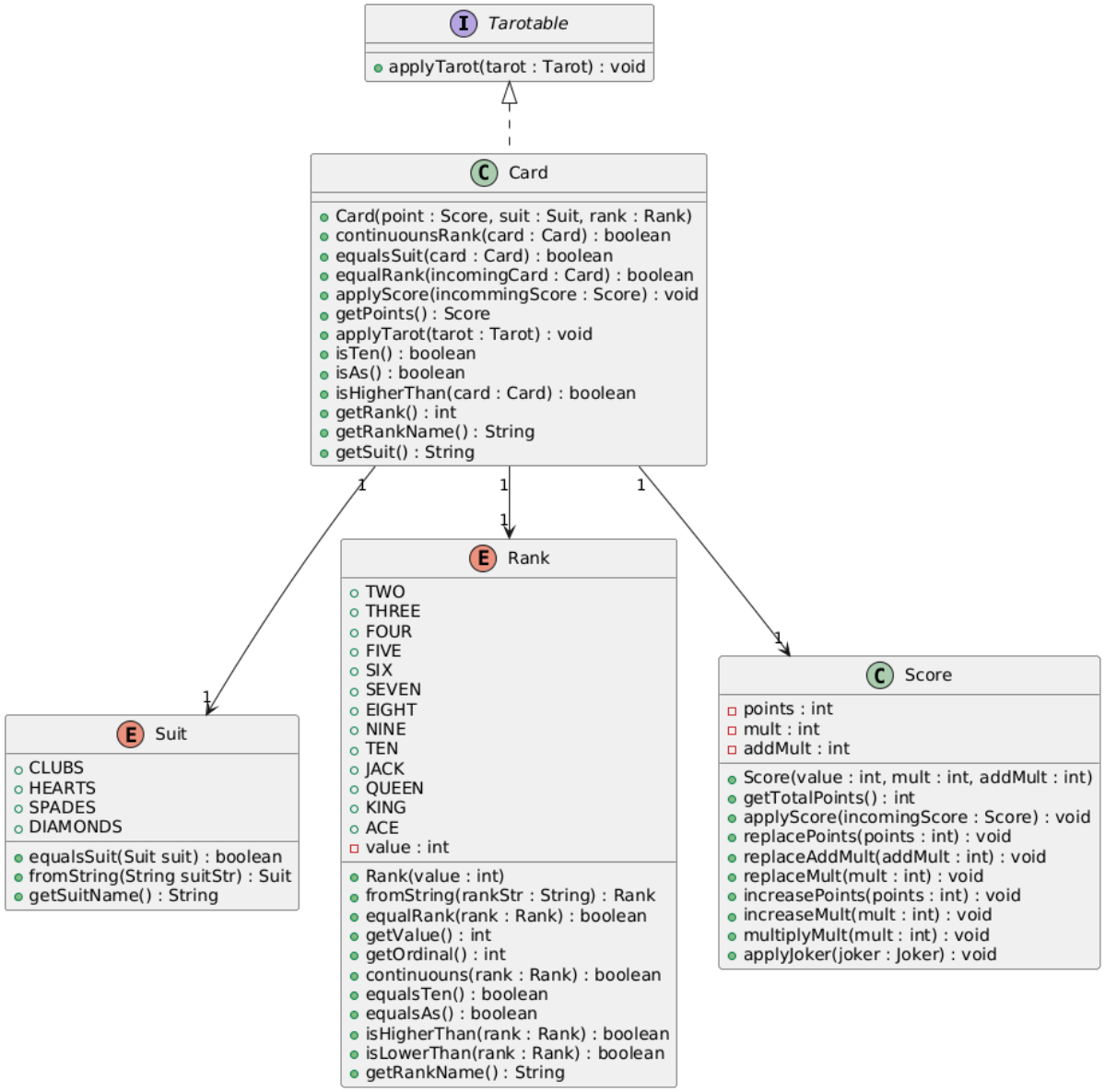
Clase Player.



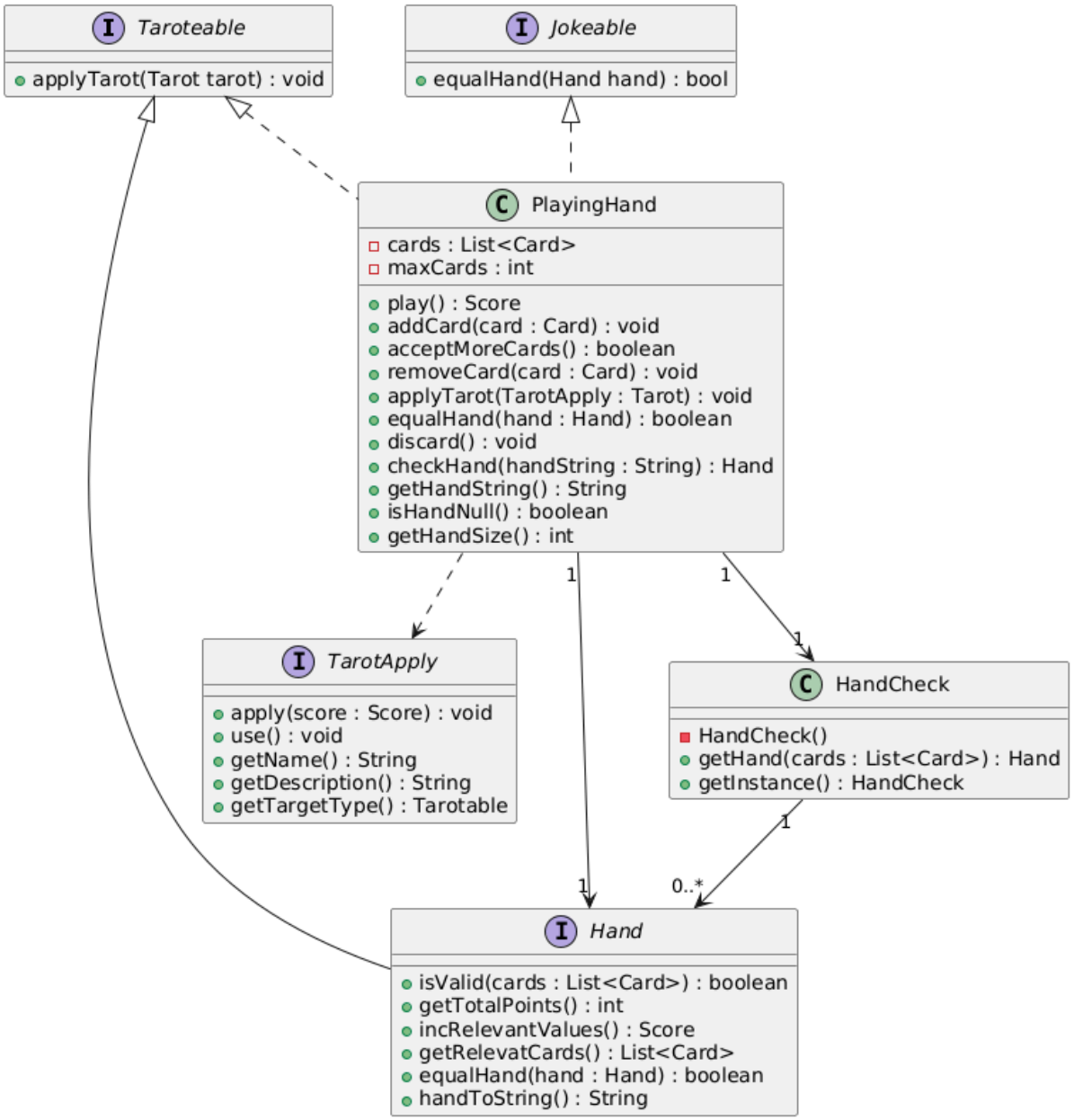
Clase Deck.



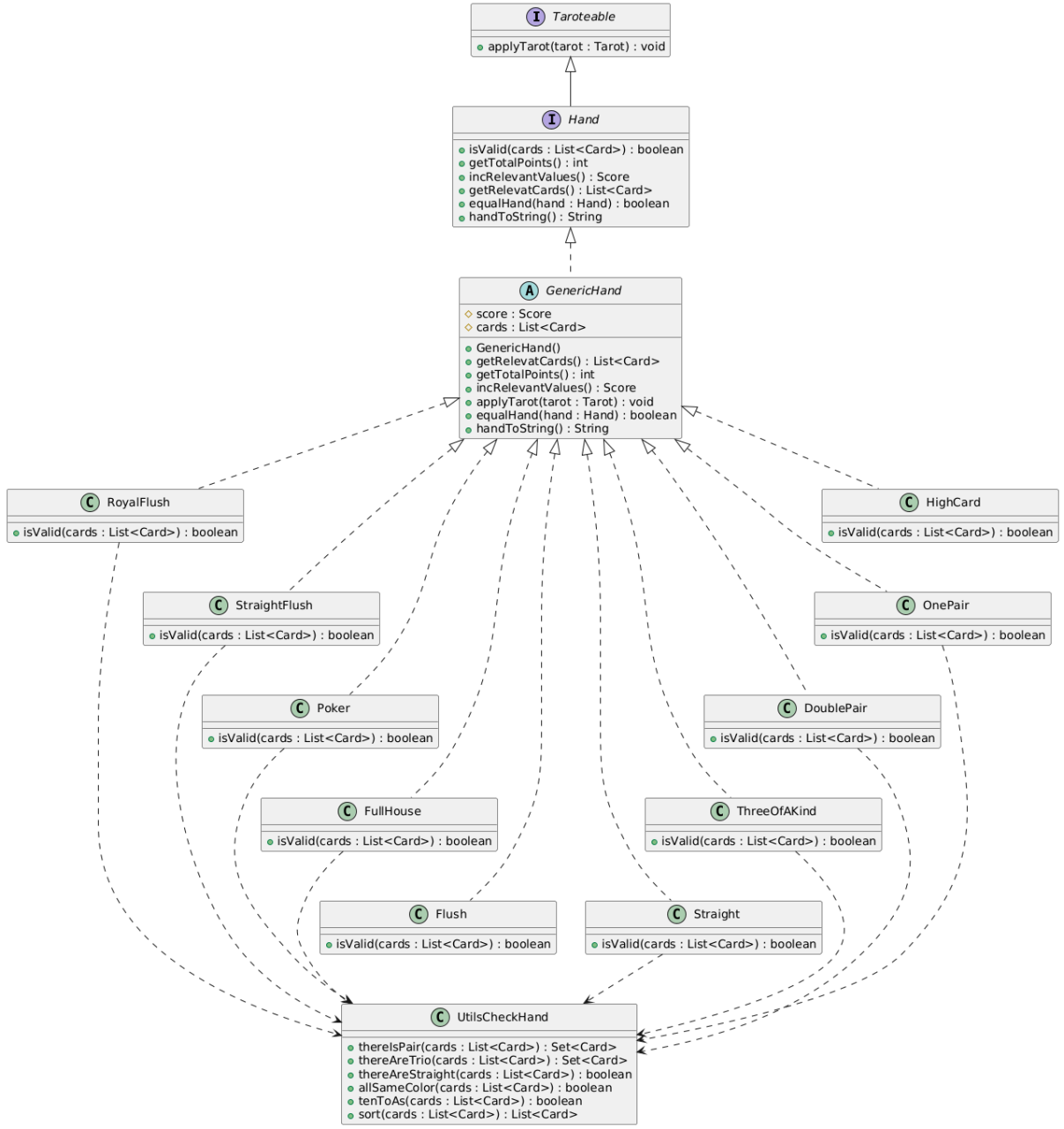
Clase Round.


Tarotable
applyTarot(tarot : Tarot) : void
Card
Card(point : Score, suit : Suit, rank : Rank)
continuounsRank(card : Card) : boolean
equalsSuit(card : Card) : boolean
equalRank(incomingCard : Card) : boolean
applyScore(incommingScore : Score) : void
getPoints() : Score
applyTarot(tarot : Tarot) : void
isTen() : boolean
isAs() : boolean
isHigherThan(card : Card) : boolean
getRank() : int
getRankName() : String
getSuit() : String
1
1
1
1
1
1
Suit
CLUBS
HEARTS
SPADES
DIAMONDS
equalsSuit(Suit suit) : boolean
fromString(String suitStr) : Suit
getSuitName() : String
Rank
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
NINE
TEN
JACK
QUEEN
KING
ACE
value : int
Rank(value : int)
fromString(rankStr : String) : Rank
equalRank(rank : Rank) : boolean
getValue() : int
getOrdinal() : int
continuouns(rank : Rank) : boolean
equalsTen() : boolean
equalsAs() : boolean
isHigherThan(rank : Rank) : boolean
isLowerThan(rank : Rank) : boolean
getRankName() : String
Score
points : int
mult : int
addMult : int
Score(value : int, mult : int, addMult : int)
getTotalPoints() : int
applyScore(incomingScore : Score) : void
replacePoints(points : int) : void
replaceAddMult(addMult : int) : void
replaceMult(mult : int) : void
increasePoints(points : int) : void
increaseMult(mult : int) : void
multiplyMult(mult : int) : void
applyJoker(joker : Joker) : void

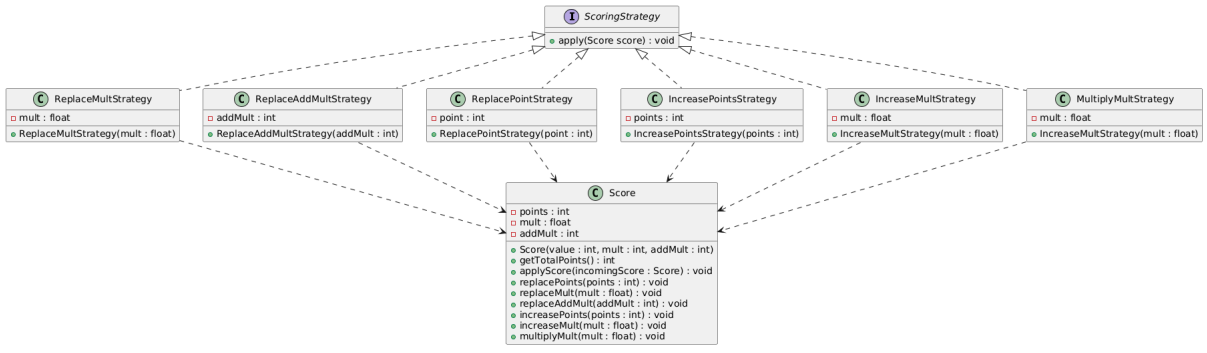
Clase Card.



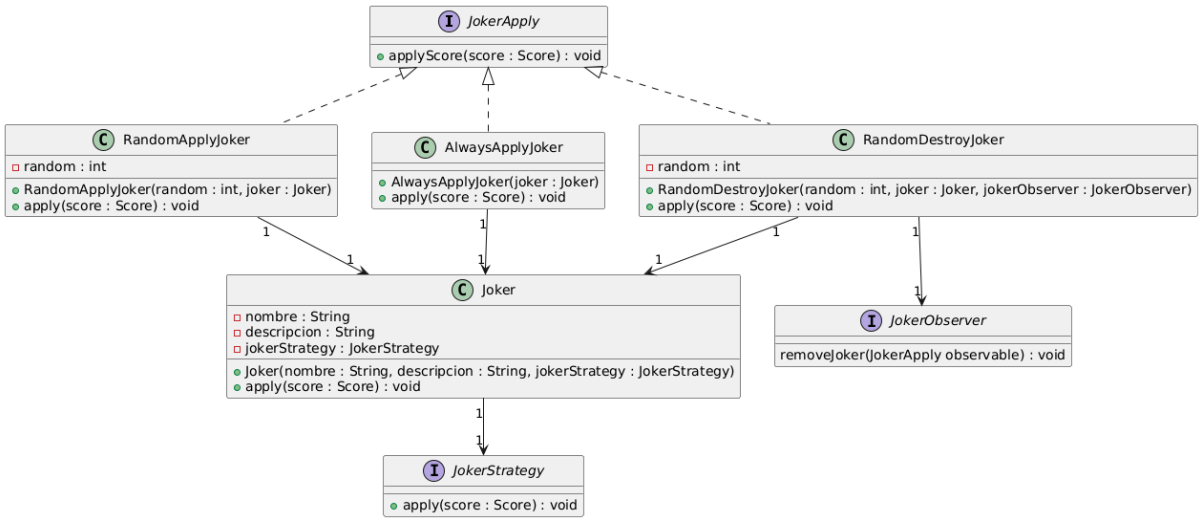
Clase PlayingHand y HandCheck.



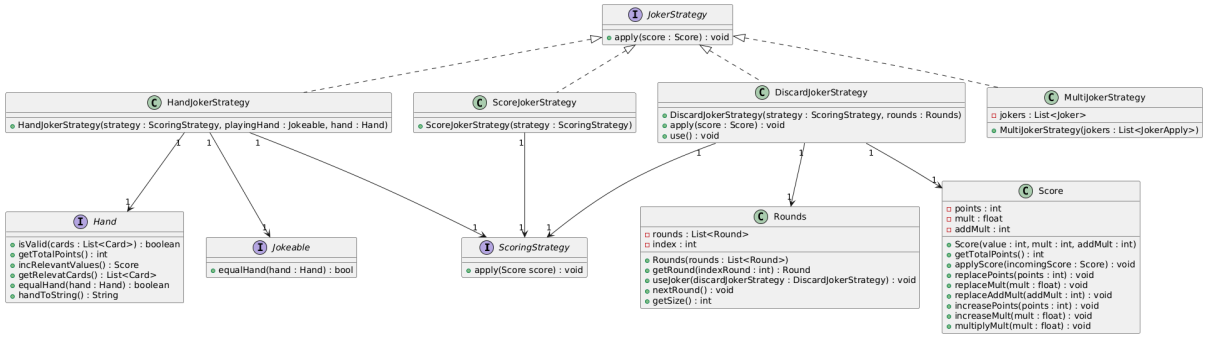
Interfaz Hand con todas las manos soportadas.



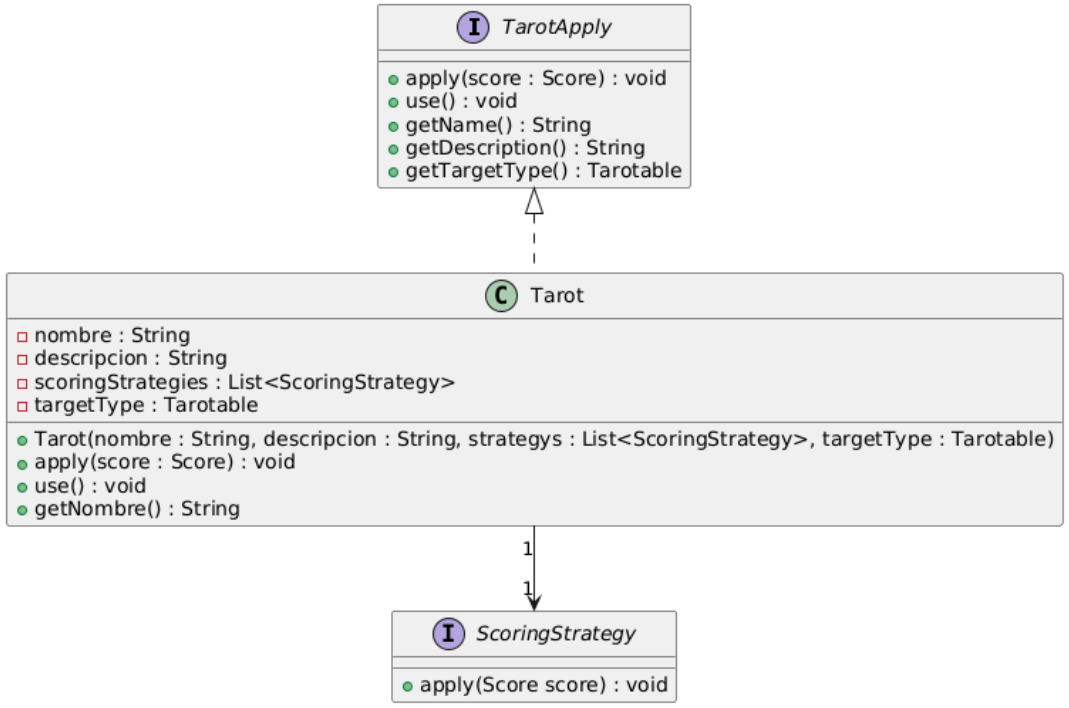
Clase Score.



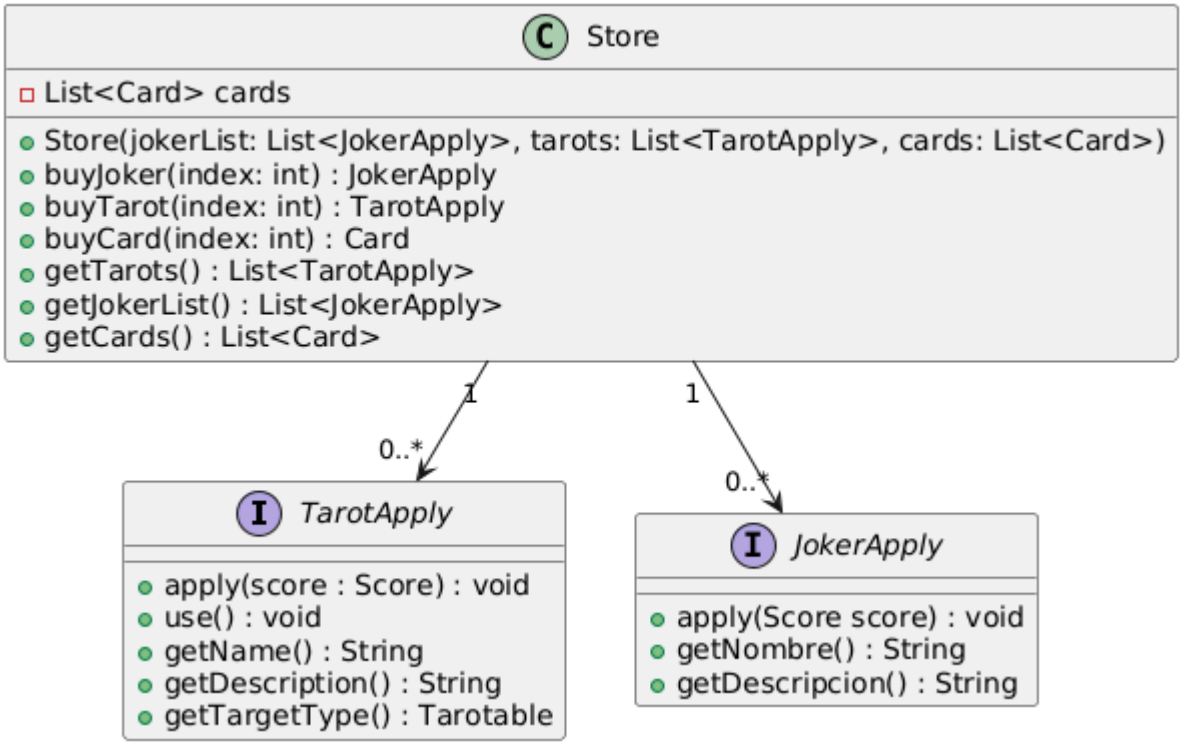
Clase Joker.



Sistema de Joker Strategy

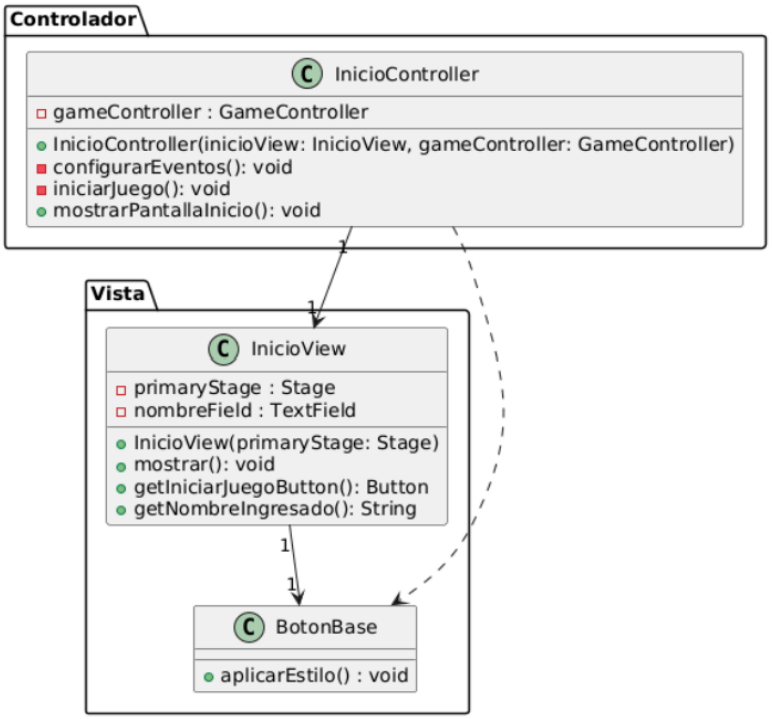


Clase Tarot.

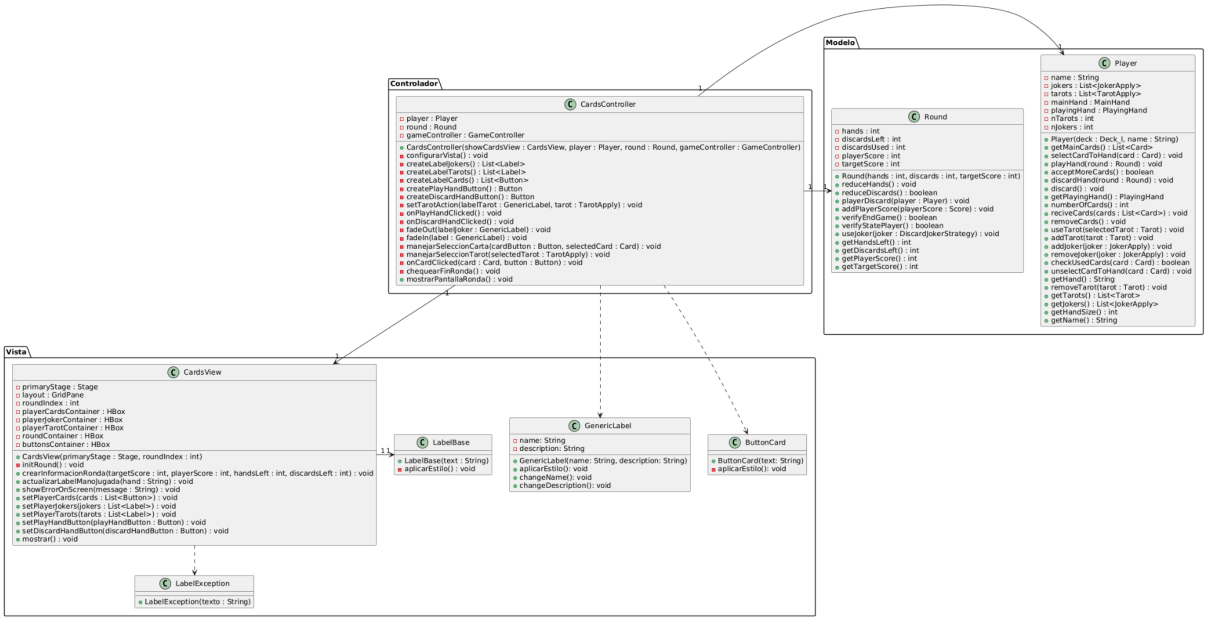


Clase Store.

Diagrama de paquetes:

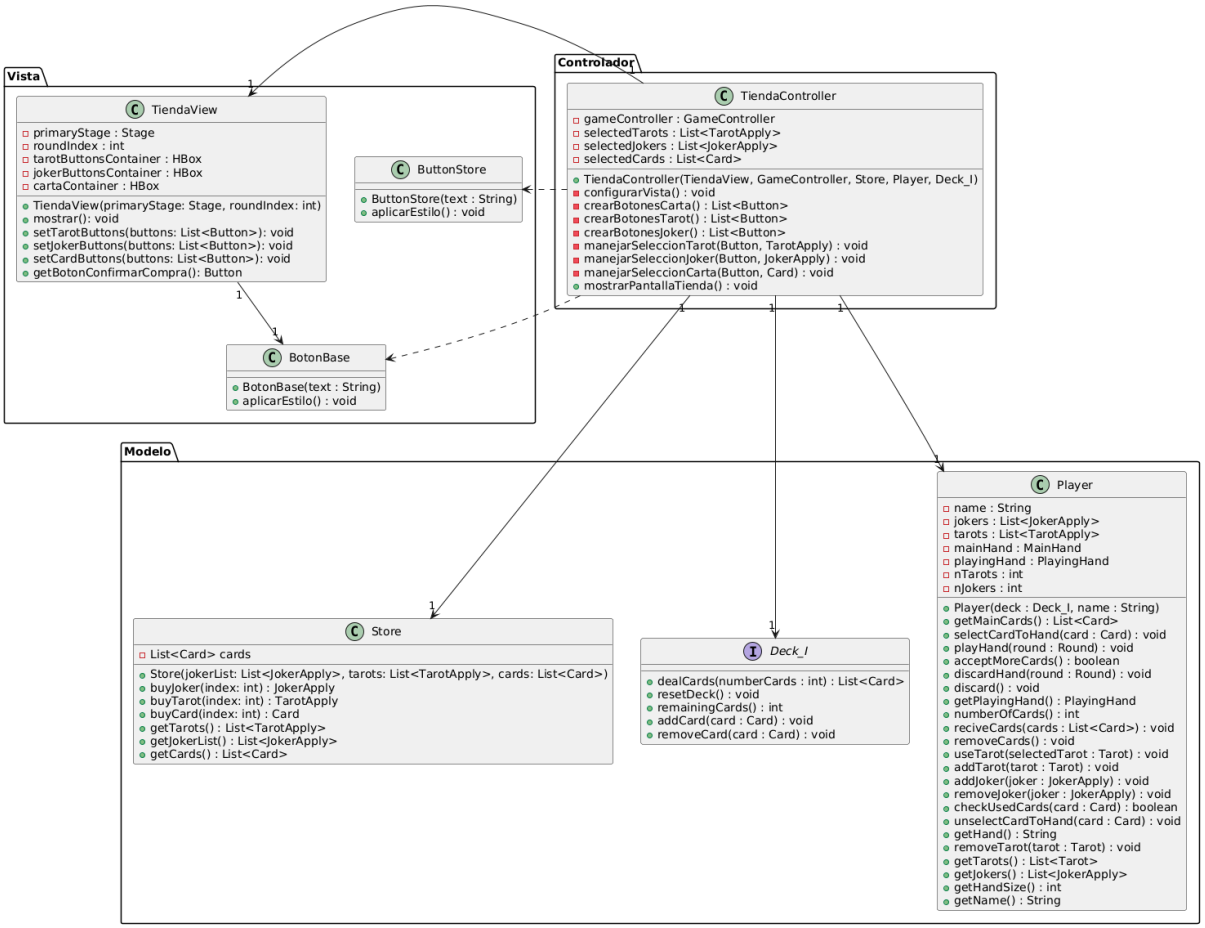


MVC del inicio del juego

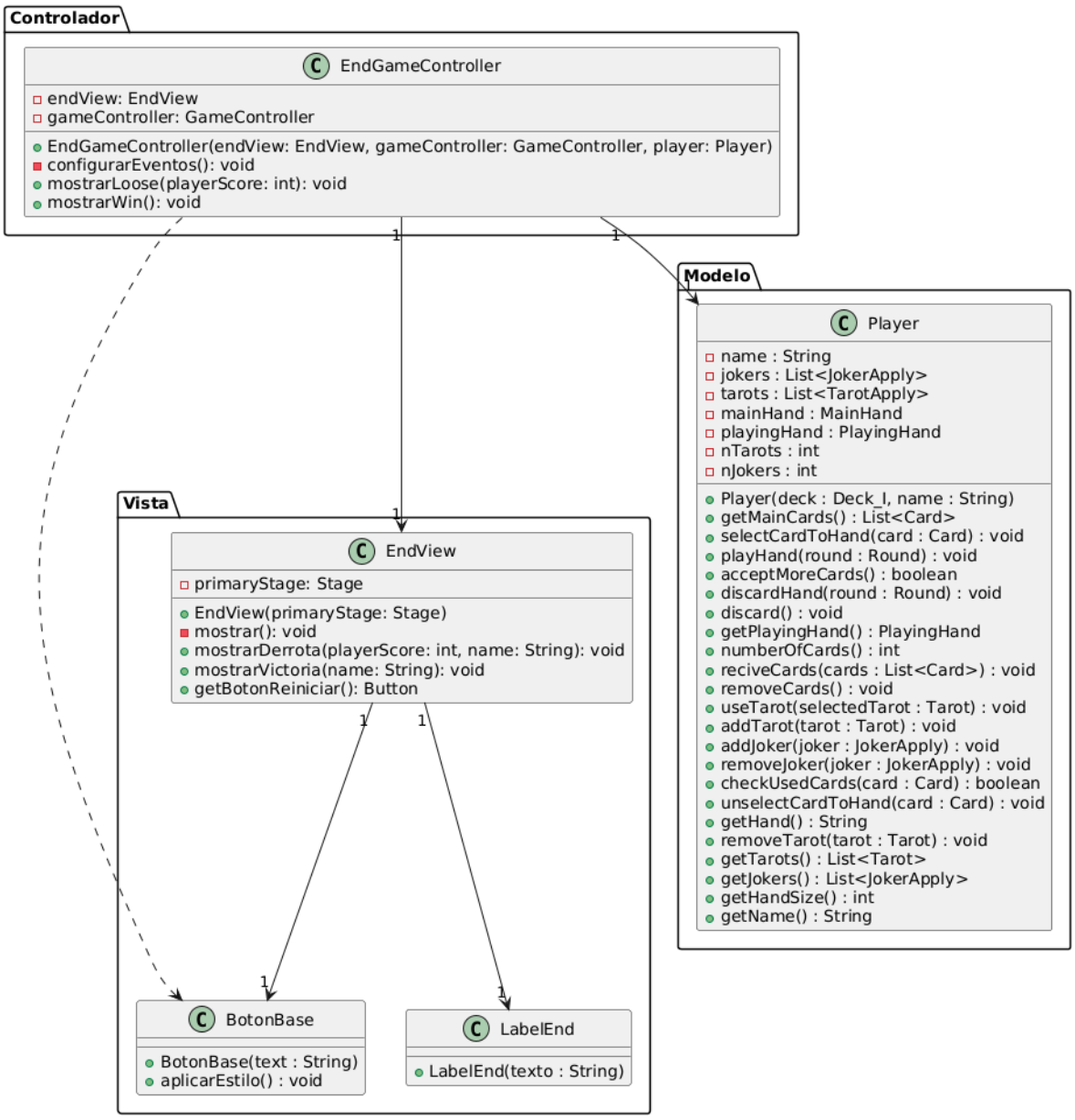


MVC de la partida

MVC Game Controller


Vista
TiendaView
- primaryStage : Stage
- roundIndex : int
- tarotButtonsContainer : HBox
- jokerButtonsContainer : HBox
- cartaContainer : HBox
+ TiendaView(primaryStage: Stage, roundIndex: int)
+ mostrar(): void
+ setTarotButtons(buttons: List<Button>): void
+ setJokerButtons(buttons: List<Button>): void
+ setCardButtons(buttons: List<Button>): void
+ getBotonConfirmarCompra(): Button
ButtonStore
+ ButtonStore(text : String)
+ aplicarEstilo() : void
BotonBase
+ BotonBase(text : String)
+ aplicarEstilo() : void
Controlador
TiendaController
- gameController : GameController
- selectedTarots : List<TarotApply>
- selectedJokers : List<JokerApply>
- selectedCards : List<Card>
+ TiendaController(TiendaView, GameController, Store, Player, Deck_I)
- configurarVista() : void
- crearBotonesCarta() : List<Button>
- crearBotonesTarot() : List<Button>
- crearBotonesJoker() : List<Button>
- manejarSeleccionTarot(Button, TarotApply) : void
- manejarSeleccionJoker(Button, JokerApply) : void
- manejarSeleccionCarta(Button, Card) : void
+ mostrarPantallaTienda() : void
Modelo
Store
- List<Card> cards
+ Store(jokerList: List<JokerApply>, tarots: List<TarotApply>, cards: List<Card>)
+ buyJoker(index: int) : JokerApply
+ buyTarot(index: int) : TarotApply
+ buyCard(index: int) : Card
+ getTarots() : List<TarotApply>
+ getJokerList() : List<JokerApply>
+ getCards() : List<Card>
Deck_I
+ dealCards(numberCards : int) : List<Card>
+ resetDeck() : void
+ remainingCards() : int
+ addCard(card : Card) : void
+ removeCard(card : Card) : void
Player
- name : String
- jokers : List<JokerApply>
- tarots : List<TarotApply>
- mainHand : MainHand
- playingHand : PlayingHand
- nTarots : int
- nJokers : int
+ Player(deck : Deck_I, name : String)
+ getMainCards() : List<Card>
+ selectCardToHand(card : Card) : void
+ playHand(round : Round) : void
+ acceptMoreCards() : boolean
+ discardHand(round : Round) : void
+ discard() : void
+ getPlayingHand() : PlayingHand
+ numberOfCards() : int
+ reciveCards(cards : List<Card>) : void
+ removeCards() : void
+ useTarot(selectedTarot : Tarot) : void
+ addTarot(tarot : Tarot) : void
+ addJoker(joker : JokerApply) : void
+ removeJoker(joker : JokerApply) : void
+ checkUsedCards(card : Card) : boolean
+ unselectCardToHand(card : Card) : void
+ getHand() : String
+ removeTarot(tarot : Tarot) : void
+ getTarots() : List<Tarot>
+ getJokers() : List<JokerApply>
+ getHandSize() : int
+ getName() : String

MVC Tienda



MVC Fin del juego

Detalles de implementación:

Joker

En general dividimos el comportamiento de Joker en 2 partes, por un lado manejamos la condición que necesita el Joker para ejecutarse, que incluye disparadores por Descartes, Mano jugada, aplicación constante al puntaje y cualquier combinación de los mismos. Por otro lado manejamos el comportamiento que tiene al aplicarse, ya que puede permitir su aplicación de forma aleatoria, después de aplicarse puede destruirse de forma aleatoria o puede ejecutarse Siempre. Esto nos permite crear todo tipo de jokers, incluso agregando nuevos

comportamientos o formas de aplicarse con la combinación que deseemos con mínimo esfuerzo, haciendo el diseño sumamente extensible y modular. A su vez el joker usa la implementación del puntaje que explicaremos a continuación en más detalle, aplicando sus efectos al puntaje que recibe. La diferenciación de ambas interfaces también nos permite asegurarnos de un orden de ejecución con las condiciones, delegando primero en la aplicación del joker la decisión de ejecutar o no la condición.También utilizamos la Interfaz Jokeable implementada por PlayingHand para poder comunicar los datos de la mano jugada al Joker para aplicar su efecto.

- Strategy:
    - Para la implementación las condiciones y de la aplicación del joker decidimos usar el patrón Strategy, que aunque no necesitamos el cambio dinámico de estrategias en tiempo de ejecución, nos permite la construcción de un modelo Extensible, encapsulado, polimórfico y modular, siguiendo los principios SOLID, especialmente Open/Closed, segregación de interfaces e inversión de dependencias. También es usado el Strategy de Score
- Observer
    - Utilizamos este patrón para el manejo de la destrucción de un Joker, ya que tiene que haber alguna forma de comunicarle al jugador que el RandomDestroyJoker .decidió destruirlo para que éste se elimine de la lista de Jokers pertenecientes al jugador manteniendo todo lo posible el encapsulamiento del sistema. Fue implementado con la creación de una interfaz JokerObserver implementado por el Player que contiene un método que le permite recibir el Joker para poder destruirlo si se cumple la condición en RandomDestroyJoker, que se enviaría a sí mismo para ser destruido.
- Partes
    - C Joker
        - I JokerStrategy
            - C MultiJokerStrategy
            - C DiscardJokerStrategy
            - C HandJokerStrategy
            - C ScoreJokerStrategy
        - I JokerApply
            - C RandomApplyJoker
            - C RandomDestroyJoker
            - C AlwaysApplyJoker
    - I Jokeable
    - I JokerObserver

Tarot

La clase tarot maneja el uso y la aplicación de Tarots sobre un objetivo cuyo puntaje (usando las estrategias de Score) queramos que sea modificado por el Tarot (Ya que los tarots solo hacen modificaciones de puntaje), estos objetivos están representados por las clases que implementan la interfaz Tarotable. Cada tarot sabe sobre qué tipo de objetivo se tiene que aplicar y cada objetivo usando la interfaz hace que el tarot se aplique sobre su puntaje.

- El Tarot usa el Strategy de Score para definir las formas en las que puede modificar el puntaje del objetivo. También seguimos el principio de Segregación de interfaces y de inversión de dependencias al hacer que el tarot interactúe con una interfaz Tarotable en vez de con una clase concreta. Tambien se usa mucho el polimorfismo, todas las partes del proceso usan el mismo mensaje apply, y cada uno sabe como aplicarse y todos son llamados y usados de forma general, a través de la interfaz o del Strategy
- Partes
    - Tarot
    - I Tarotable

Flujo de cartas para jugar

Main Hand se encarga de la administración de cartas del jugador, recibiendo el Deck, llevando la cuenta de las cartas repartidas y de las que que ya fueron usadas o descartadas por el jugador. Esto distribuye la carga de responsabilidades del jugador, delegando en MainHand. Llevamos registro de las cartas que han sido usadas para removerlas de la mano y para repartir la cantidad exacta de cartas que fueron usadas o descartadas, así nunca bajamos de 8 cartas (Hasta que el mazo se quede sin cartas y no tenga más que devolver). Para jugar una mano, el jugador selecciona cartas a Playinghand y luego coordina con MainHand para eliminar las cartas de la mano y repartirlas, igual al descartar. Esto permite separar la lógica entre la mano de cartas que el jugador tiene disponible de la responsabilidad de repartir las cartas y de la verificación de manos de una mano jugada, permitiendo el encapsulamiento.

- Partes
    - MainHand
        - Deck
        - Cards
        - Cards Used
        - numberOfCards

HandCheck

La forma que tenemos para identificar las manos de poker es una Clase HandCheck que inicializa todas las manos que nuestro juego usa y las agrega a una lista de Manos (Clases que implementan la clase mano). En este caso, todas las manos por defecto del juego las definimos a través de la clase abstracta GenericHand que tiene el comportamiento base extendido sobre lo que define la interfaz Hand. Cada clase mano puede recibir una lista de cartas y se encarga de verificar si esa lista corresponde, pero cómo evitamos que el sistema asuma que todas las manos son un high card o que un poker es un par? El orden de verificación es la clave, Al pasar el array por cada clase Mano en orden de mayor a menor complejidad a nivel de condiciones, nos aseguramos de que nunca una Mano pueda ser confundida por otra de menor valor, así podemos primero verificar si hay 4 cartas del mismo rango antes de verificar si hay un par, evitando conflictos y confusiones. Esto utiliza sobre todo el principio de diseño KISS, simplificando el proceso de verificación de cartas con un sistema que igual permite la extensibilidad, permitiendo simplemente agregar nuevos tipos de manos a la lista iterada, tomando en cuenta el orden. Las manos también utilizan el principio de Segregación de Interfaces y de Inversión de dependencias al usar la Interfaz Mano como abstracción.

- Estructura
    - I Hand
    - HandCheck
    - GenericHand
    - All The hands

Score

Score es la base de nuestro juego, el que maneja el puntaje y como es modificado el puntaje de todas las partes. Se divide en 3 atributos principales: Points, Mult y AddMult. Hicimos esta distinción entre Mult y AddMult ya que en el juego el puntaje consiste en base de puntos y multiplicador, pero entre los modificadores que se pueden tener para el mult existen directamente multiplicadores del valor base de mult pero también sumamos a este valor, que a su vez después de que se apliquen todos los modificadores a cada parte del Score, Points y Mult se multiplican para dar el puntaje final. El orden de cálculo del Score cuando se le aplica a otro Score es primero se le agrega al mult el addMult entrante, luego se multiplica el mult por mult entrante y por último se suman los points, así se lleva registro de todas los datos necesarios para al final hacer Points multiplicado por Mult para conseguir el score final. Tenemos también una interfaz ScoringStrategy que usamos para definir las diferentes formas que tienen los elementos del juego como los Jokers y Tarots para modificar los puntos, pudiendo desde reemplazar cada atributo del puntaje por el que ellos definan hasta incrementar o modificar el puntaje existente.

- Strategy: Aunque no necesitamos el cambio dinámico de estrategias en tiempo de ejecución (cosa que podría usarse luego para crear otros tipos de jokers que muten, por ejemplo), nos permite la construcción de un modelo Extensible, encapsulado, polimórfico y modular, siguiendo los principios SOLID, especialmente Open/Closed, segregación de interfaces e inversión de dependencias.
- Partes
    - Score
    - ScoringStrategy
        - IncreaseMultStrategy
        - IncreasePointStrategy
        - MultiplyMultStrategy
        - ReplaceAddMultStrategy
        - ReplaceMultStrategy
        - ReplacePointStrategy

Comentarios extra:

Tuvimos constantemente en cuenta los principios de diseño como SOLID, patrones de Diseño y los Pilares de POO, intentando hacer un sistema con alta cohesión y bajo acoplamiento. Tuvimos muchos exitos como la creación de Jokers, el manejo de Tarots y el chequeo de manos (aunque este se puede modularizar reestructurando las verificaciones para que usen partes como conjuntos de pares, tríos, escaleras genéricas que se podrían decorar, etc)

Igualmente quedan refactors y mejoras por hacer, desde cosas tan simples como hacer que el Player aparezca con una mano generada en vez de generarla externamente hasta un error que por alguna razón duplica algunas cartas y reduce el tamaño total de la mano si son usadas o descartadas (ARREGLADO, era que no se eliminaban las cartas que el player tenía en la mano antes de pasar de ronda, entonces se duplicaban las referencias a la misma instancia de una carta, como el sistema sabia que era la misma instancia, contaba como una sola al descartar o jugar pero ocupaba dos espacios). El Score también creo que es posible moldearlo distinto, soltando la necesidad de usar un atributo addMult, ya que describe un comportamiento que puede ser definido como una estrategia, que defina su forma de interpretar el mult. También el manejo de MainHand con player y deck capaz es muy complicado, llevando varios registros de cartas usadas.

Excepciones

- NoCardsInHandException: Lanzamos esta excepción cuando se intenta jugar sin tener cartas seleccionadas en la mano. Evita que el jugador pueda mandar una mano vacía e informa del error con un mensaje, recordando que tiene que seleccionar cartas.
- NoDiscardsLeftException: Lanzamos esta excepción cuando se intenta descartar una mano sin tener descartes restantes. Evita que el jugador pueda

descartar más veces que las estipuladas por el juego y la ronda e informa del error con un mensaje, recordando que tiene que seleccionar cartas.

- NoCardsInDiscardException: Lanzamos esta excepción cuando se intenta descartar sin tener cartas seleccionadas en la mano. Evita que el jugador pueda gastar un descarte con una mano vacía e informa del error con un mensaje, recordando que tiene que seleccionar cartas.
- CardTarotNeedsOnlyOneCardSelected: Lanzamos esta excepción cuando se intenta activar un tarot tipo carta sin tener cartas seleccionadas o seleccionando cartas de más. Evita que el jugador pueda gastar el tarot sin que sea aplicado a ninguna carta o que lo pueda aplicar a más cartas que las estipuladas por el juego e informa del error con un mensaje, recordando que tiene que seleccionar o deseleccionar cartas.