FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 - Balatro

[7507/9502] Paradigmas de la programación

Curso 01

Segundo cuatrimestre de 2024

Participantes:	
Joaquin Hernandez	110470
Lautaro Martin Sotelo	107472
Gaspar Amato	111137
Pablo Alcocer	106597
Bryan Serrantes	110158

Tutor:Maia Naftali

Nota Final:

INTRODUCCIÓN

El presente informe reúne la documentación sobre la resolución del segundo trabajo práctico realizado grupalmente. Para este trabajo práctico se desarrolló un juego llamado Balatro el cual podría decirse que es como un poker en solitario.

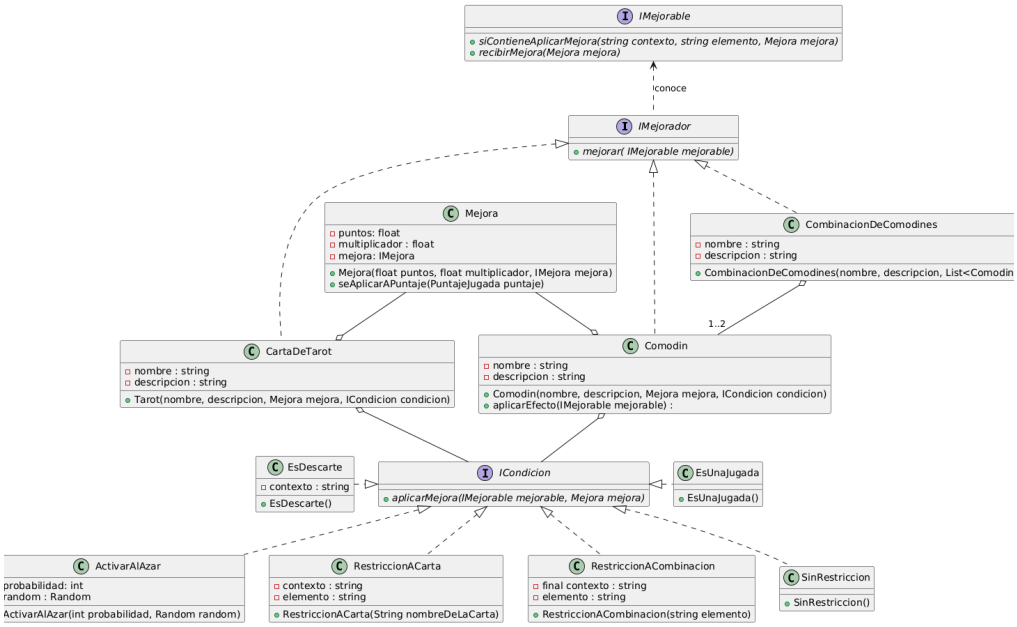
1. Supuestos

Los supuestos que hicimos para la realización del TP fueron:

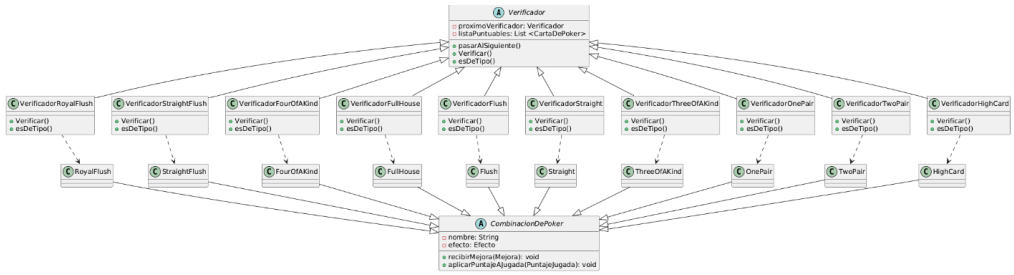
- El jugador solo puede elegir una carta modificadora en cada tienda con su respectiva ronda.
- Las Cartas de la Tienda son gratis para el Jugador, no dependen de fichas o un puntaje especial.
- Suponemos que el jugador conoce las combinaciones de poker existentes.
- Cada Acción del jugador se tomará como turno, sus acciones incluyen tanto el descarte como jugar la mano.Con ello, puede que en una ronda el jugador pierda antes de usar todos sus descartes o todas las manos jugables, el límite de acciones/turnos es de 5.

2. Diagramas de Clases

FACULTAD DE INGENIERIA
Universidad de Buenos Aires



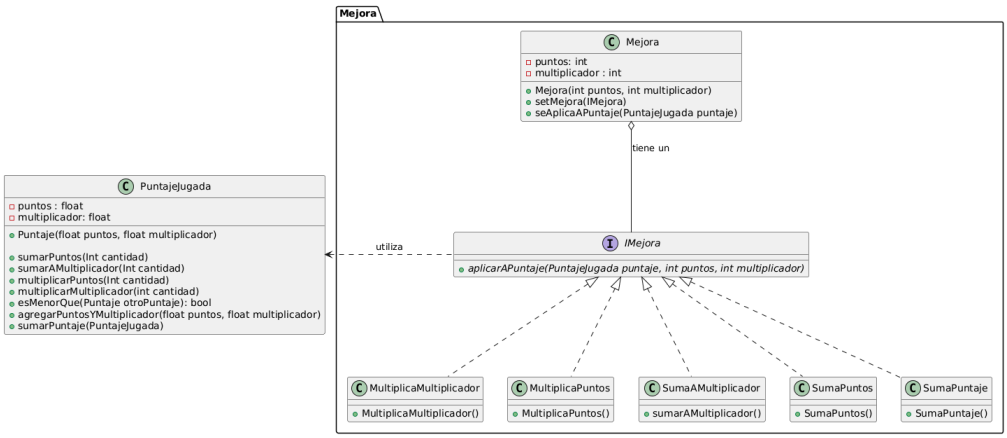
Este diagrama de clases describe las interfaces IMejorador e IMejorable. La interfaz IMejorable contiene el método siContieneAplicarMejora(), este se encarga de que si la clase que lo implemente contiene un contexto y/o elemento correspondientes, se aplicará la mejora pasada por parámetros, la razón de esta interfaz era para encapsular a las clases que podían ser mejoradas. Este contexto y elemento serán brindados por el IMejorador y se las pasará a una clase interna almacenada en sus atributos que se encargará de validar si se debe o no aplicar la mejora a la IMejorable. Este atributo almacena una clase que implementa a la interfaz ICondicion. Por último Descarte y Jugada implementan a IAccion, esta interfaz servirá para la clase ronda que se explicará después.



Esta clase verificador la utilizamos para determinar el tipo de mano en una jugada. Se le pasa una lista de cartas y devuelve si

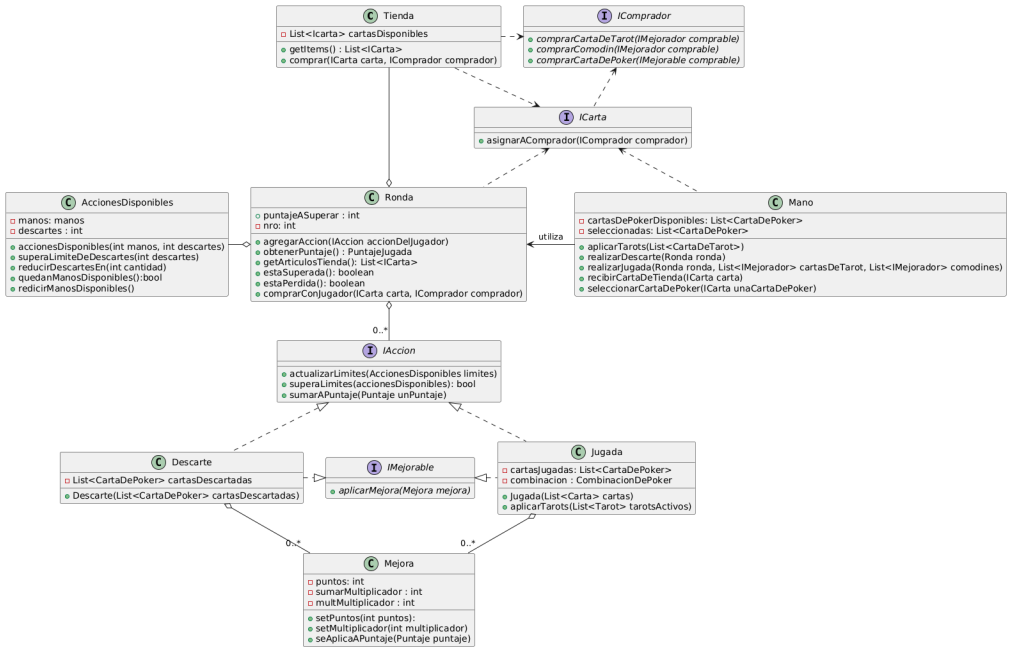
es Escalera, Trío, etc. Para el verificador, se empleó el patrón de diseño Chain of Responsibility, ya que permite estructurar mucho mejor las verificaciones, asignando a cada clase la tarea de identificar un tipo específico de mano. Esto facilita la extensibilidad, permitiendo agregar nuevas verificaciones sin necesidad de modificar las existentes, y mantiene el código más limpio y fácil de mantener al evitar un único bloque de condicionales complejo.

Además este verificador determina que cartas son las que efectivamente interactúan con el puntaje, ya que para nuestro modelo.Para hacer esto lo que hicimos fue utilizar el patrón de diseño Strategy, esto nos permite modificar el comportamiento de una carta externamente y que no haya necesidad de realizar verificaciones para saber si la cartas está o no seleccionada.Cabe destacar que cada carta tiene un estado, el cual será puntuable o no usada, cuando pase por el verificador, su estado cambiará a puntuable.



Este diagrama de clases muestra la relación entre los Mejoradores y su Mejora, así como su conexión con el PuntajeJugada. Al crear las clases Comodin y CartaDeTarot, se les asigna una mejora que solo aplicará cuando se calcule el puntaje de la jugada. Las IMejora representan cómo la Mejora interactúa con el PuntajeJugada; por ejemplo, si al crear una Mejora se introduce SumaAMultiplicador, el multiplicador de mejora se añadirá al multiplicador del PuntajeJugada. Al calcular el PuntajeJugada, los Comodines y los Tarots tienen las mejoras, y además estas últimas

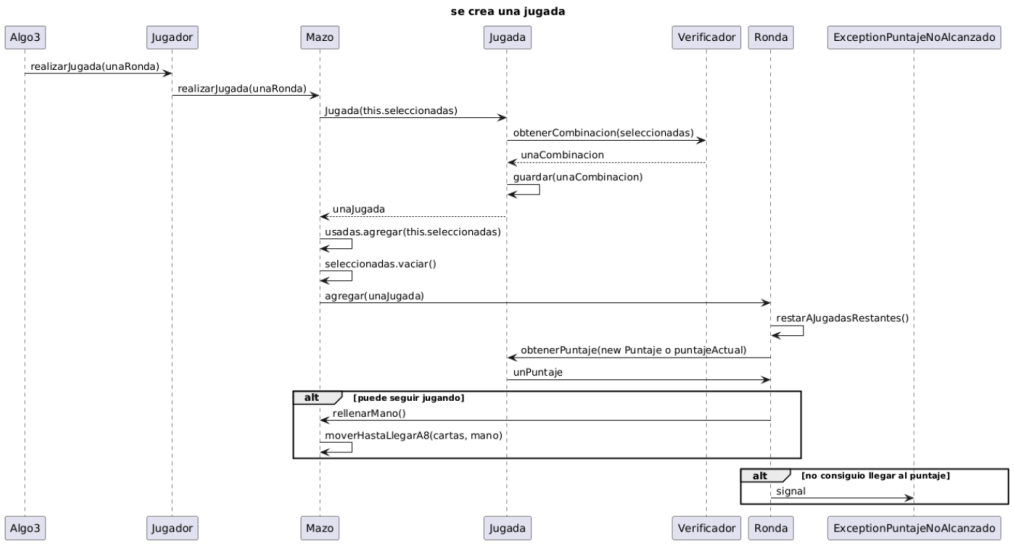
almacenan una carta objetivo. Si al calcular el puntaje de una Jugada el tarot pasa por el objetivo correcto, aplica su mejora al Puntaje Jugada.Diferencia esencial, el comodín afectará solo al final del turno, mientras que el tarot modificara la carta o la mano que tiene como ejemplar.

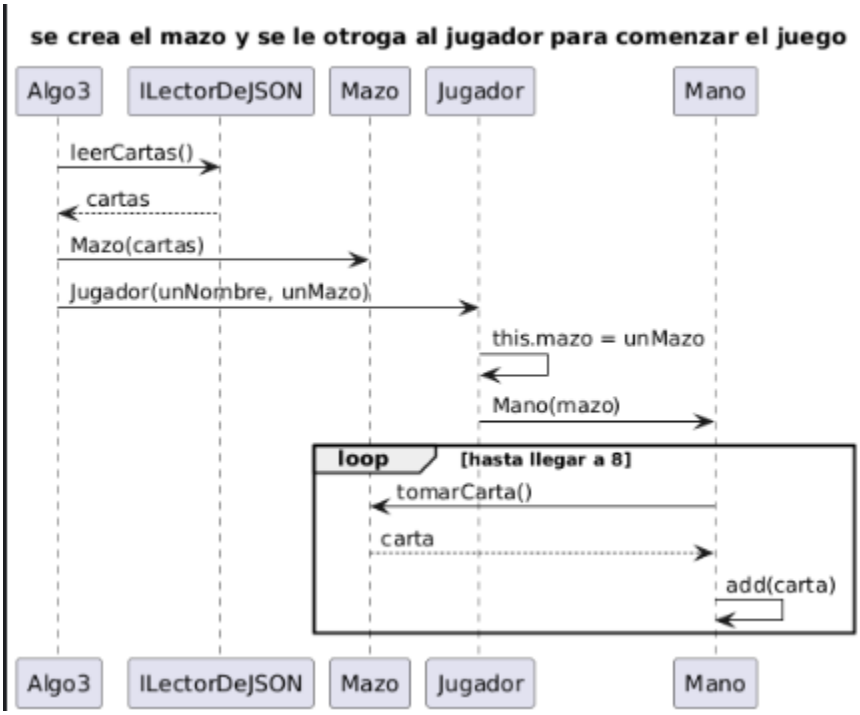


Cada ronda jugada tiene una cantidad de descartes y de manos que se pueden jugar máximas, la ronda determina si puede o no realizar una acción en base a si ha alcanzado el límite, esta verificación de si se ha alcanzado o no el límite está encapsulada en una clase llamada AccionesDisponibles. en caso de no poder agregar más.

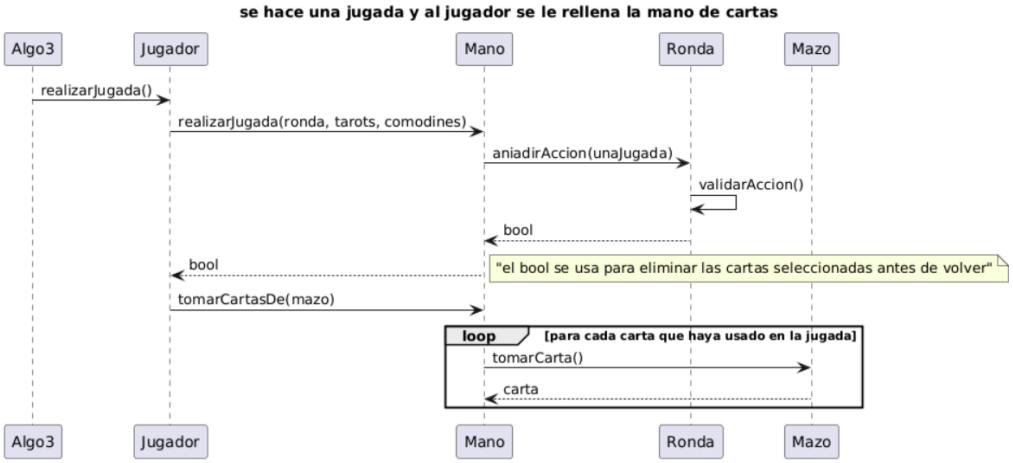
Cada vez que se realiza una jugada o un descarte la Ronda lo agrega a una lista interna que almacena toda las IAccion que se ejecutaron que luego serán utilizadas para calcular el puntaje de esa ronda. El método agregarAcción() realiza una verificación de con cada acción que se intenta agregar y devuelve un booleano en base a si se supera o no el límite.

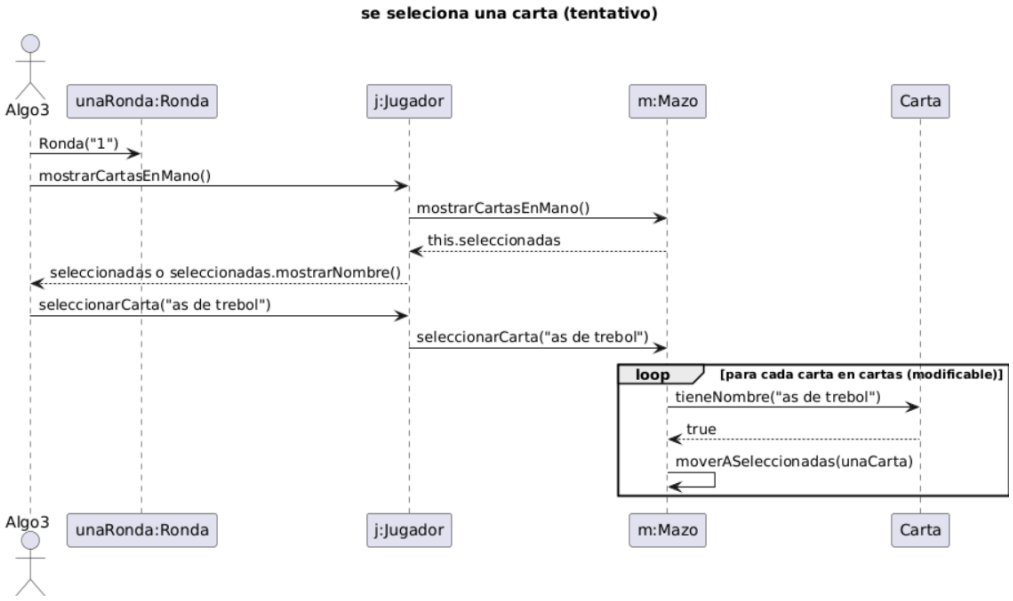
3. DIAGRAMAS DE SECUENCIAS



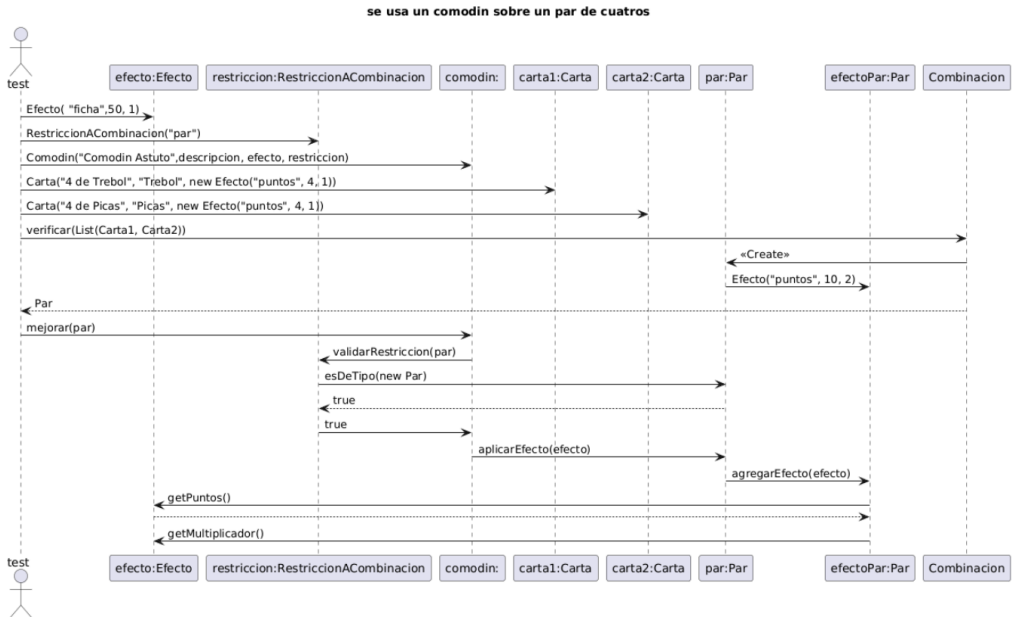


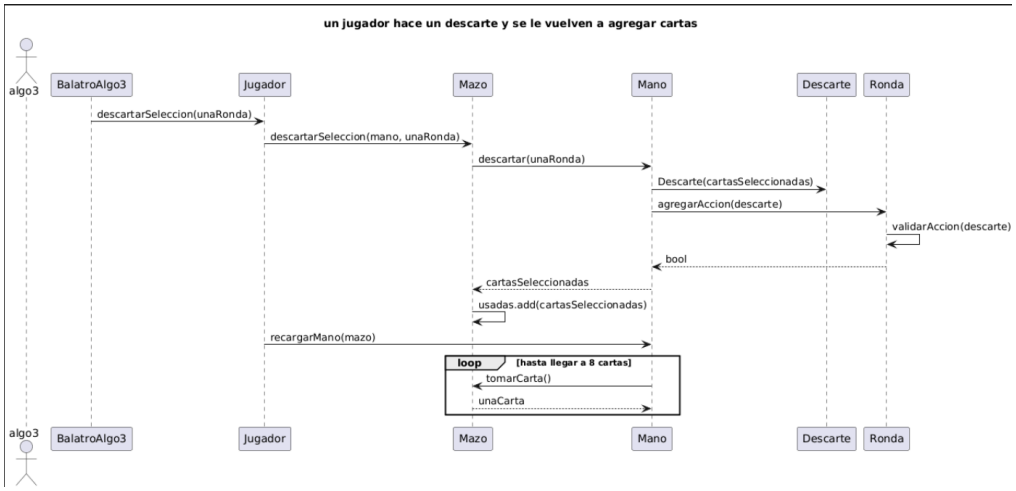
FACULTAD DE INGENIERIA
Universidad de Buenos Aires

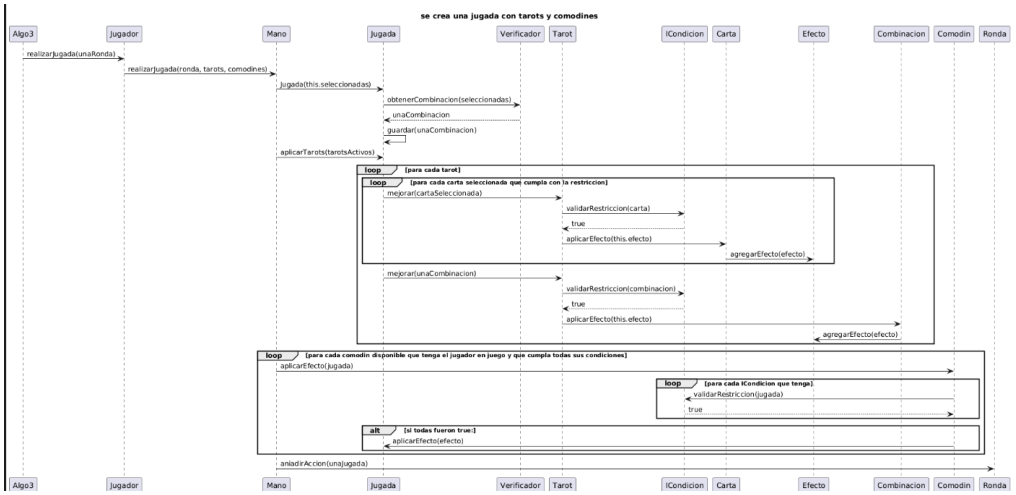




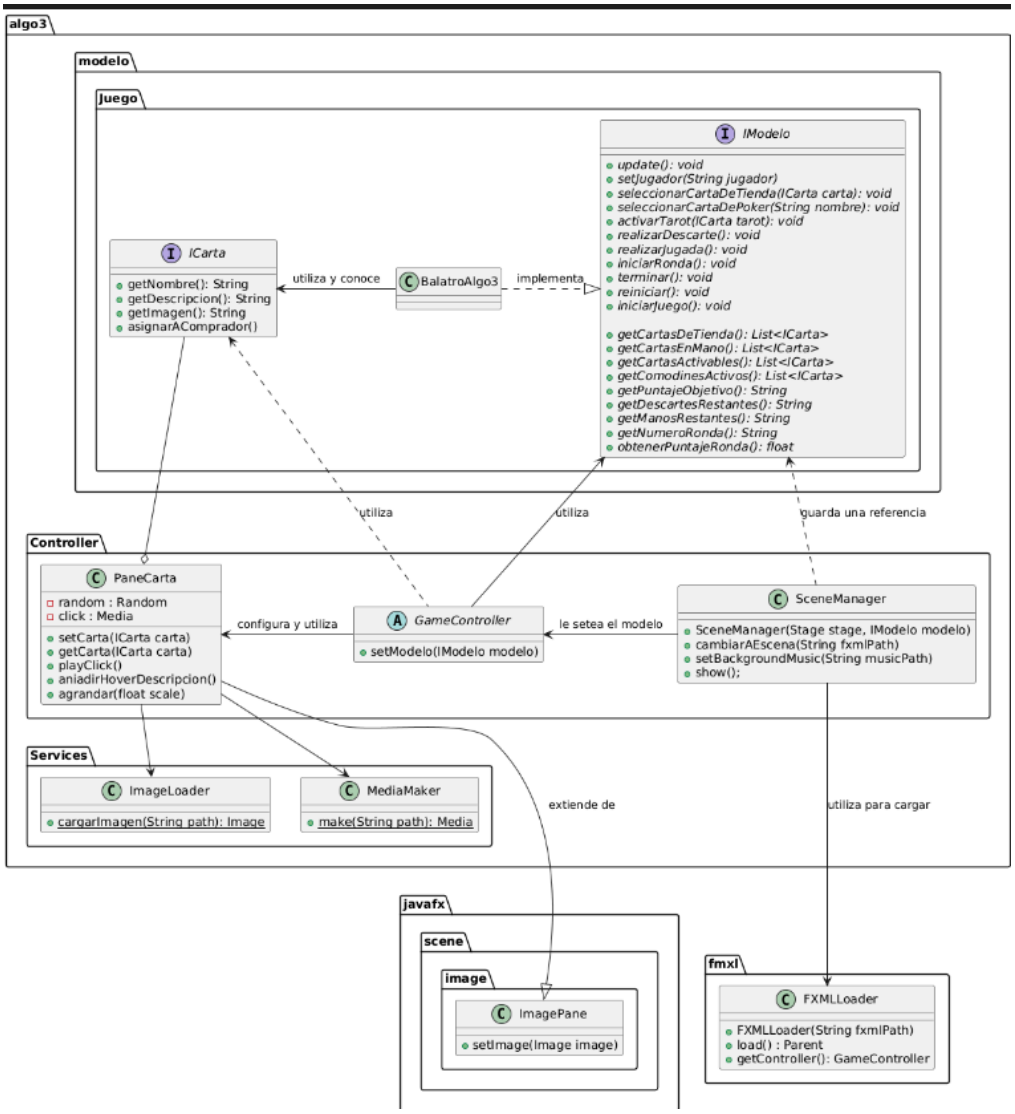
FACULTAD DE INGENIERIA
Universidad de Buenos Aires

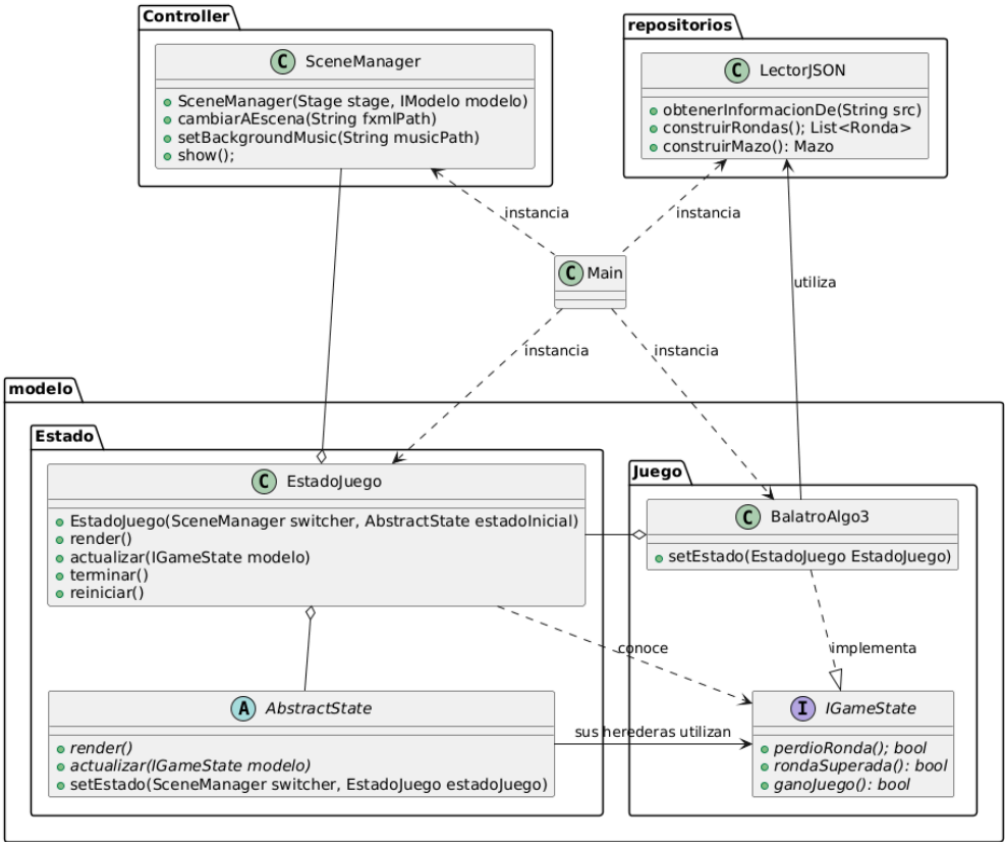






4. DIAGRAMAS DE PAQUETES





5. Excepciones

Las excepciones que utilizamos fueron:

a) **ErrorJugadaVacia:**

Esta excepción se lanza cuando a una Jugada se le ingresa un alista con 0 cartas, ya que no debería poder pasar esto.

b) **ErrorManoSeExcedioDeCartas:**

Este error es el contrario del anterior ya que se lanza cuando la jugada tiene más de las 5 cartas permitidas

c) **ErrorNoSePuedenSeleccionarCartasQueNoEstenEn LaMano:**

Esta excepción se lanza cuando se le pide a la clase Mano seleccionar una carta que no está en la lista mano.

d) **ErrorSeExcedenLosLimitesDeActivables :**

Excepción que se lanza en el caso que el Jugador compre cartas Tarots o Comodines y su activables (Efectos de cartas) excede el límite posible.

6.Detalles de Implementación

1. *Clase Balastro:*

Clase que se encarga de manejar el flujo del juego, y efectuar la comunicación entre las acciones que se realiza el jugador en la interfaz (mediante las vistas y los controladores) y trasladarlas al resto del modelo.Balastro se encarga de resolver cada Ronda, teniendo sus métodos para que el jugador seleccione las cartas o active tanto tarots como comodines.El mismo también puede cambiar de estado debido a la implementación del diseño de State , el cual permite mediante estados indicar si se termina o reinicia el juego.Actualiza el estado o pasa de ronda entre otros.

2. *Clase Ronda y Tienda:*

La Clase Ronda representa la ronda del juego, la misma contiene la información que tiene el jugador en el juego ,las cuales son las manos y descartes disponibles ,como el puntaje que debe superar el mismo.El mismo informara si el usuario cumplio con el puntaje deseado o no.Tambien cada ronda contra a una Clase Tienda, la misma contiene las cartas que el usuario puede comprar, y ronda se la asignará al usuario mediante la compra en Tienda.L

3. *Clase Acciones Disponibles y interfaz IAcción.*

Cada ronda guarda la información de rondas y descartes, la misma se delegará a Acciones Disponibles, al cual irá actualizando de acuerdo a cómo el usuario toma sus decisiones.

IAccion, actualiza las acciones y también aplicará puntajes de acciones, como en el caso de descartes que suman puntos (o fichas que es lo mismo) por cada descarte realizado, en sí una acción.

4. *Clase Jugador y Mano*

El jugador es el contenedor de su propio nombre identificador, como de los activables o comodines que puede utilizar, y el mazo del cual puede retroalimentar a su mano.El mismo puede seleccionar cartas y recargar su propia mano, como la posibilidad de realizar jugadas y descartes.Aquí el jugador decide sobre también que carta activar un tarot, el mismo Tarot dejará de ser un Activable y pasará a ser un Tarot Activo en la jugada.Cada activable de Tarot se actualiza con la compra de un Tarot en la tienda.

Por otro lado la mano es lo que posee el jugador , la mano en si se retroalimenta del mazo donde dentro un array de la Clase CartaDePoker, con un total de 8 cartas, donde luego el usuario seleccionara 5 de esas 8 y la mano actualizará su array de cartas seleccionadas.

5. *IMejorador ,CartaDeTarot,CombinacionDeComodines y Comodin*

IMejorador es una interfaz que utilizan tanto CartaDeTarot como CombinacionDeComodines y Comodin.El objetivo de las mismas es identificar cual es la clase que tiene el objetivo de mejorar elementos, en este caso **IMejorador** mejorará **IMejorable**. A través de su método mejorar, recibirá un mejorable y le aplicará una ***mejora***, con el efecto en este caso, que cualquiera de las 3 clases tenga y a la vez el contexto y el elemento que se busca mejorar. Así las mismas marcaran tanto una **CartaObjetivo** o puede ser una **CombinacionDePoker**

```
public interface IMejorador {

    void mejorar(IMejorable mejorable);

    //logica segun cada Joker o Tarot o
Combinacion

}
```

6. *IMejorable , CombinacionDePoker y CartaDePoker*

La interfaz **iMejorable** se utiliza para agrupar a todo aquello que pueda ser mejorable por efectos y condiciones traídas por las cartas que implementen **IMejorador**. Estas tienen los métodos *aplicarMejora* y *siContieneAplicarMejora.* La primera simplemente aplicará el efecto que se desea, siempre y cuando, *siContieneAplicarMejora* verifica que el elemento que se desea mejorar se encuentra.

- CartaDePoker:

  La carta de poker es contenedora de sus propios atributos (palo,multiplicador,puntos) como a su vez de un EstadoDeCarta, donde se utilizan estados para determinar y diferenciar los diferentes momentos que una carta está Seleccionada,Puntuable O Usada.

- CombinacionDePoker:

  La combinación de Poker es abstracta y implementa también la interfaz de IMejorable y es contenedora de sus atributos y del Puntaje de la Jugada. Aquí se evaluará qué tipo de mano el usuario ha jugado, a través del principio de diseño de Chain of Responsibility , así asignando qué tipo de combinación es.

```
public interface IMejorable {

   void siContieneAplicarMejora(String
contexto, String elemento, Mejora mejora);

//Logica segun cada mejorable,
Jugada,CartaDePoker,Descarte o
CombinacionDePoker.



}
```

7. *Clase PuntajeJugada*

PuntajeJugada es donde se encapsula todos los puntajes obtenidos mediante una Jugada del usuario. Aquí se calcularán los puntos obtenidos de cualquier jugada. El mismo es utilizado por la interfaz de *iMejora,*y las clases que la extendienden para calcular puntos.*D*eterminado en la las clases que usan *IMejorador* y se encargan de aplicar mejoras, en PuntajeJugada si debe multiplicarPuntos,SumarPuntos O SumarAMultiplicador según corresponda.

8. *Jugada*

Cada Clase **Jugada** es creada a partir de la selección de cartas por el usuario o en este caso *Jugador* .La misma es contenedora de una lista de la clase *CartaDePoker*, una lista de Mejoras que se fueron aplicando a los diferentes mejorables y una *CombinacionDePoker* con un objeto de la Clase Verificador. El verificador le dará el valor a la combinación a partir de la selección de cartas del usuario y su posterior evaluación. Así mismo este devolverá el puntaje final de toda la jugada a partir de las *CartasDePoke*r seleccionadas, la combinación de forman y las mejoras de la jugada.

```
public PuntajeJugada
aplicaPuntajeDeAccion(PuntajeJugada puntaje){


   for(CartaDePoker carta: this.cartas) {
       carta.sumarAPuntajeJugada(puntaje);
   }


   combinacion.sumarAPuntajeJugada(puntaje);
   for(Mejora mejora: this.mejoras) {
       mejora.seAplicaAPuntaje(puntaje);
   }
```

```
    return puntaje;
}
```

---

9. *Descarte*

La clase **Descarte** en comportamiento es similar a Jugada, mientras que una jugada se activa cada vez que el usuario gasta una mano, el **Descarte** , valga la redundancia, cada vez que el usuario gaste un Descarte posible.El mismo se construye a partir se una Lista de *CartaDePoker* seleccionada por el usuario, y al ser los descartes mejorables también , utiliza la interfaz *IMejorable*, evalúa que el contexto sea un descarte y mejorará el descarte (ya sea un descarte que sume puntos en cada turno o que aplique un modificador de multiplicador).

10. *Verificador*

Esta clase Abstracta es como se digo anteriormente la encargada mediante Chain of Responsability de verificar la selección de cartas y asignar los datos puntuables correspondientes, cuando se encuentra el tipo de Combinación que es, se cambiara el estado de cada carta a Puntuable.

```
public void agregarPuntuables(List
<CartaDePoker> cartasAPuntuar) {
    for (CartaDePoker carta : cartasAPuntuar) {
        carta.changeState(new Puntuable());

this.listaPuntuables.addAll(cartasAPuntuar);
    }
    ;
}
```

11. EstadoDeJuego

La clase EstadoJuego implementa el patrón de diseño State para gestionar el cambio de estados del y transicion juego. Aquí hay un resumen de su funcionamiento:

Atributos:

gameState: un objeto de tipo IGameState, que representa el estado del juego.

estadoActual: una instancia de AbstractState, que maneja el estado actual del juego.

switcher: una interfaz que se usa para cambiar entre los diferentes estados del juego.Principalmente, la representacion grafica de cada estado a partir de lso diferentes fxml.

Métodos:

Constructor: Inicializa el estadoActual con un estado inicial y configura la transición de estado a través de switcher.

*setModel(*IGameState modelo): Asigna un modelo al juego, gestionar la lógica del estado.Aunque en nuestro juego no se usa implicitamente, ya que usamos actualizar , este metodo podria ser util para implementar un observer en caso de necesitarlo.

cambiarA(AbstractState nuevoEstado): Cambia el estado actual a un nuevo estado y renderiza el estado.

render(): Llama al método render del estado actual, para mostrar información del juego en la interfaz.

actualizar(IGameState modelo): Actualiza el estado con el modelo proporcionado.

terminar(): Cambia al estado de inicio del juego (EstadoInicio).

reiniciar(): Cambia al estado de creación de partida (EstadoCreandoPartida).


FACULTAD
DE INGENIERIA
Universidad de Buenos Aires