FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Balatro

[7507/9502] Algoritmos y Programación III
Segundo Cuatrimestre de 2024

Alumno	**Número de Padrón**	**Email**
Brian Conde	110953	bconde@fi.uba.ar
Salvador Pérez Mendoza	110198	sperezm@fi.uba.ar
Atuel Sebastián Fullana	110247	afullana@fi.uba.ar
Francisco Russo	107480	frrusso@fi.uba.ar

Índice

1. Introducción

En el trabajo práctico se desea programar una copia simplificada de Balatro, siguiendo los principios de la POO y el TDD. Se desea crear código mantenible, de alta cohesión y bajo acoplamiento. Además, se busca hacer una interfaz de usuario intuitiva y accesible.

Balatro es un juego "roguelike", "deck-builder" hecho en Lua que se encuentra disponible en Steam. El juego consiste en sumar fichas a través de manos de póker, las cuales pueden ser modificadas por cartas especiales, llamados comodines y tarots. Para ganar, se debe superar un límite mínimo de fichas en una cierta cantidad de turnos.

2. Supuestos

- La carta que aparece en cada ronda, dentro del JSON de configuración es una carta que pertenece a la tienda de dicha ronda.
- Los comodines suman su multiplicador al puntaje acumulado, mientras que las cartas multiplican su multiplicador al puntaje acumulado.
- Los tarots setean el valor base y el multiplicador de sus objetivos.
- Las descripciones de las cartas especiales son solo descriptivas y no necesariamente tengan un efecto asociado en el juego. Por ejemplo, no se puede distinguir si los comodines suman o multiplican el multiplicador del puntaje acumulado. Tampoco existen las cartas de "piedra" o "cristal".
- El jugador puede y debe comprar una sola carta al comienzo de cada turno.
- El mazo se reestablece y se mezcla luego de cada ronda ganada.

3. Diagramas de clases

Para facilitar la comprensión del modelo, se hicieron distintos diagramas UML enfocados en distintos aspectos del juego. En primer lugar, está el diagrama de clases directamente relacionado a las cartas de Póker (figura 1). La clase Jugada es abstracta, y sus hijas son todas las jugadas válidas del Póker. Póker contiene un atributo del tipo Palo, el cual no está representado en el diagrama al no contener comportamiento especial.

El siguiente diagrama (figura 2) se enfoca en las cartas especiales: los comodines y los tarots. Los comodines fueron modelados usando el patrón Composite, el cual es explicado en detalle en los detalles de implementación. En el diagrama, se mantuvo un a representación simplificada de ellos.

Por último, existen otras clases de suma importancia para la configuración y el flujo del juego. Las mismas se ilustran en la figura 3.

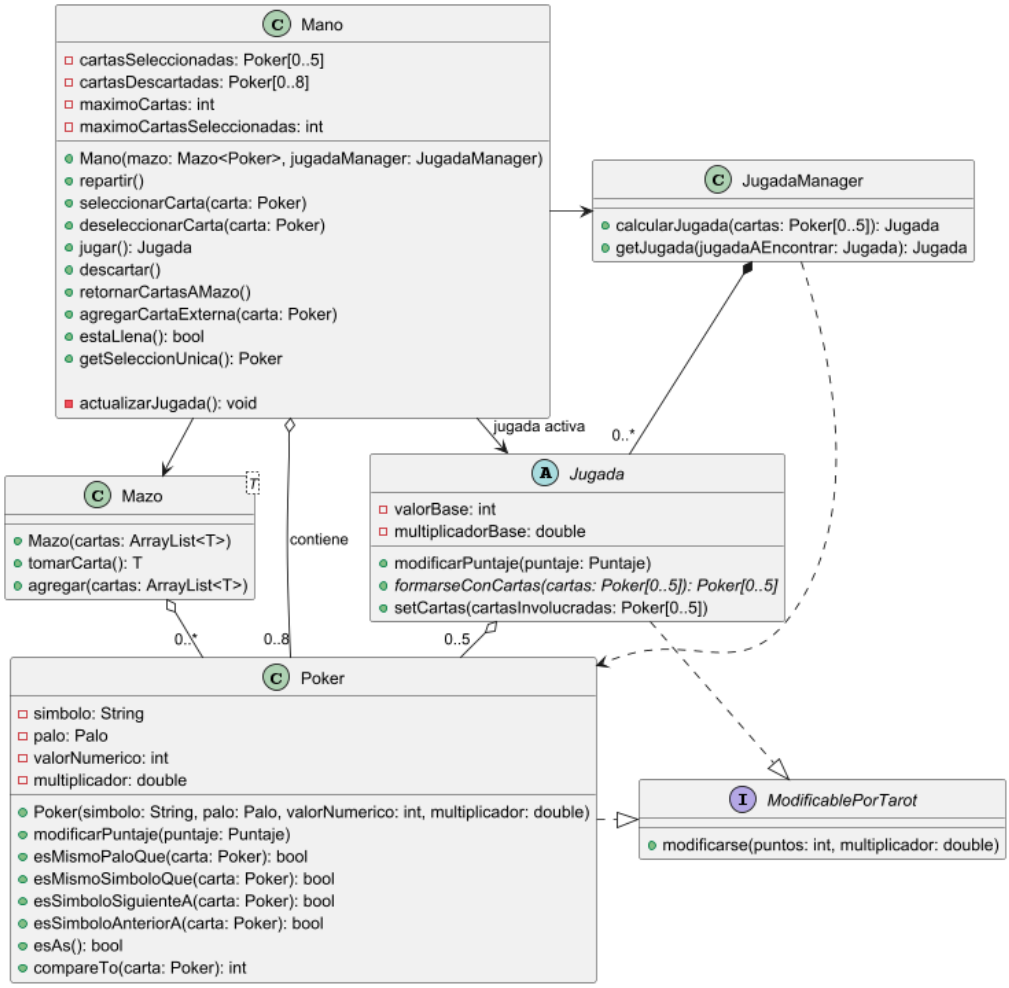

Figura 1: Diagrama de clases enfocado en Poker

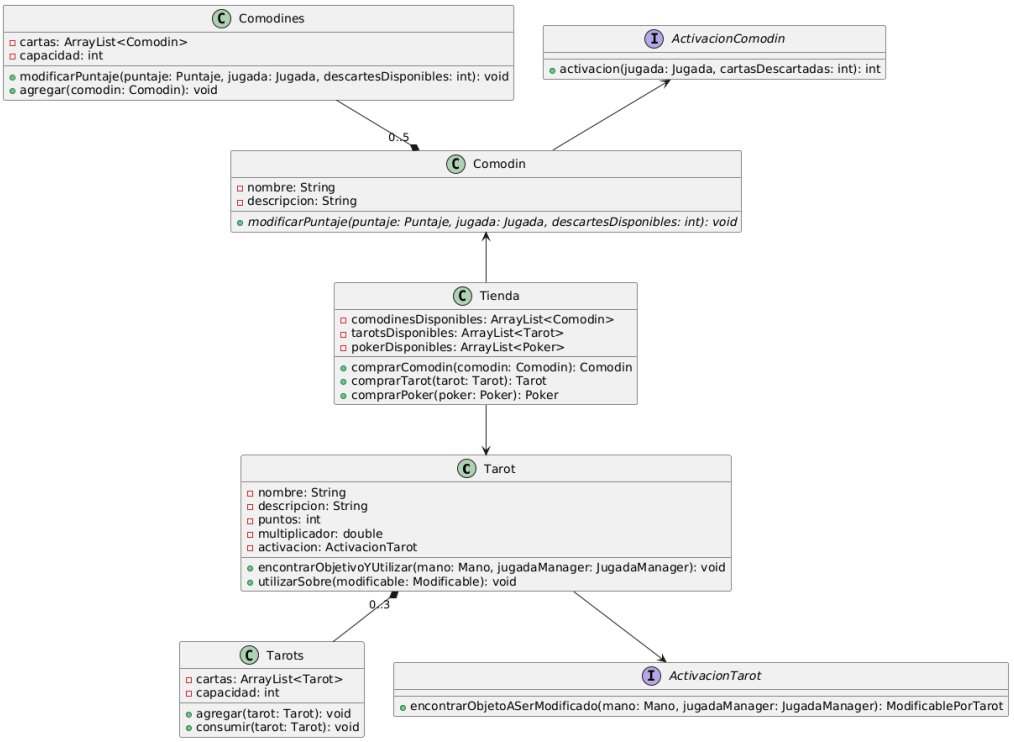

Figura 2: Diagrama de la clase Tienda y sus relaciones

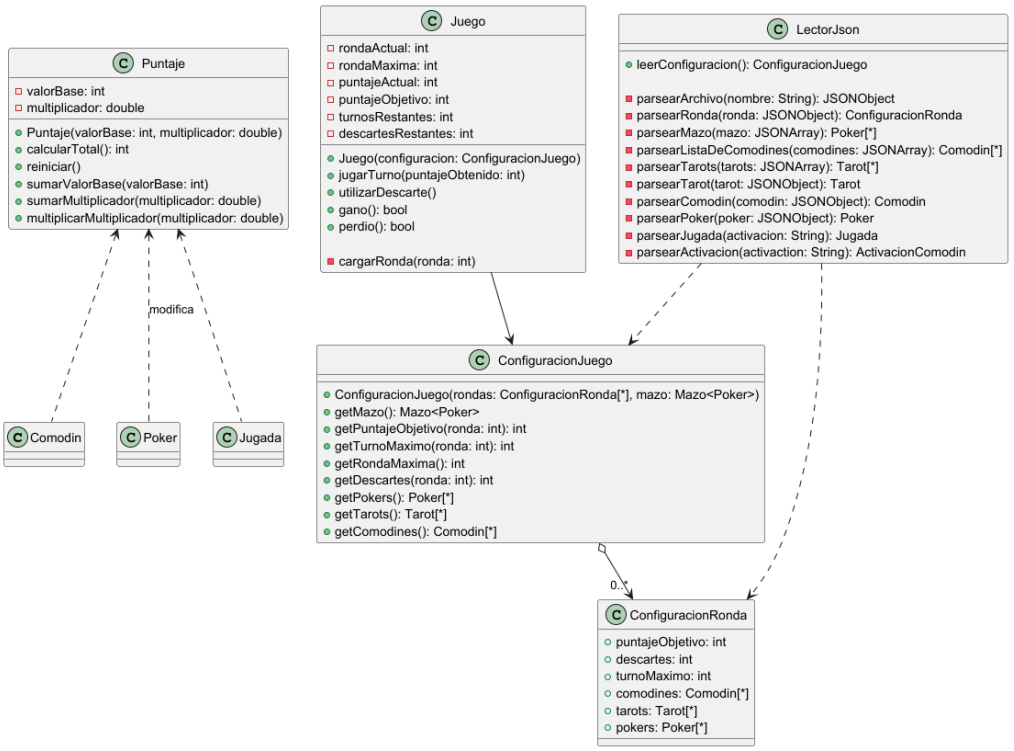

Figura 3: Diagrama de clases enfocado en el puntaje y el flujo del juego

4. Diagramas de secuencia

A continuación se muestran diferentes diagramas de secuencia para casos de uso interesantes o importantes.

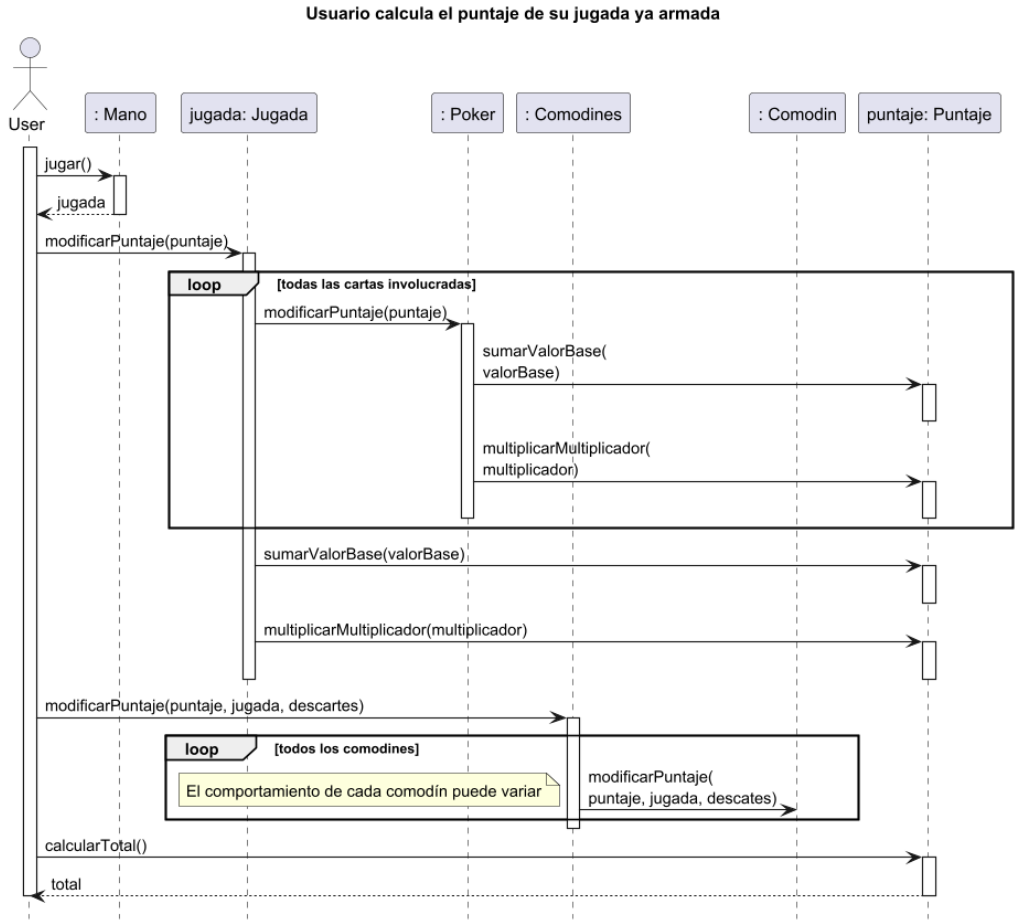

Figura 4: Usuario calcula el puntaje de su jugada ya armada

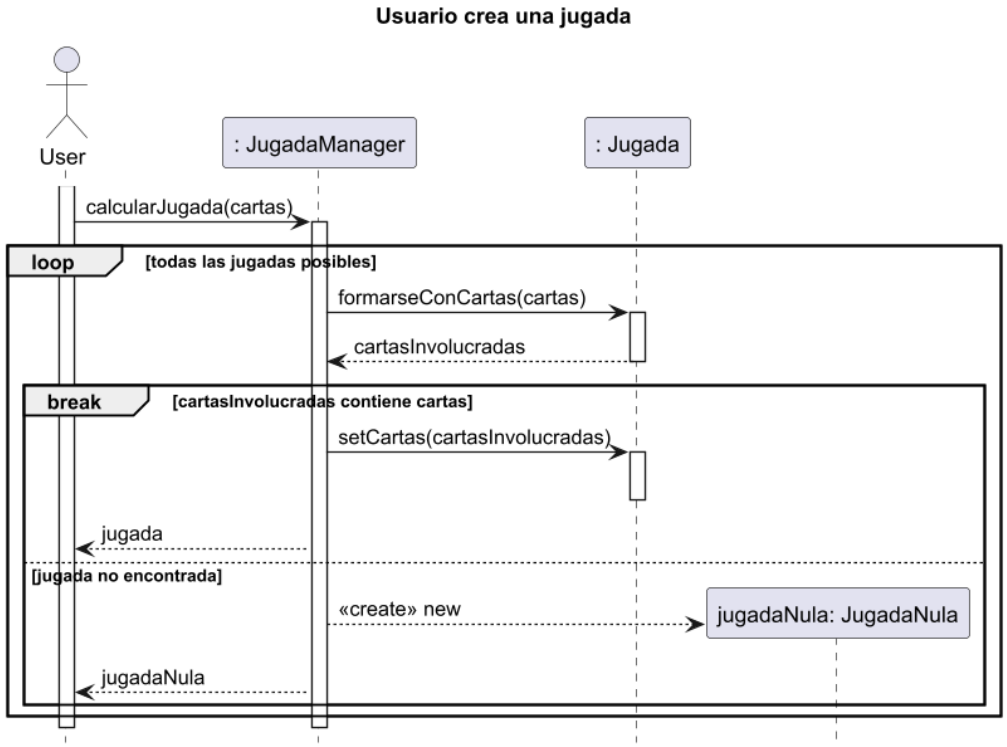

Figura 5: Usuario crea una jugada

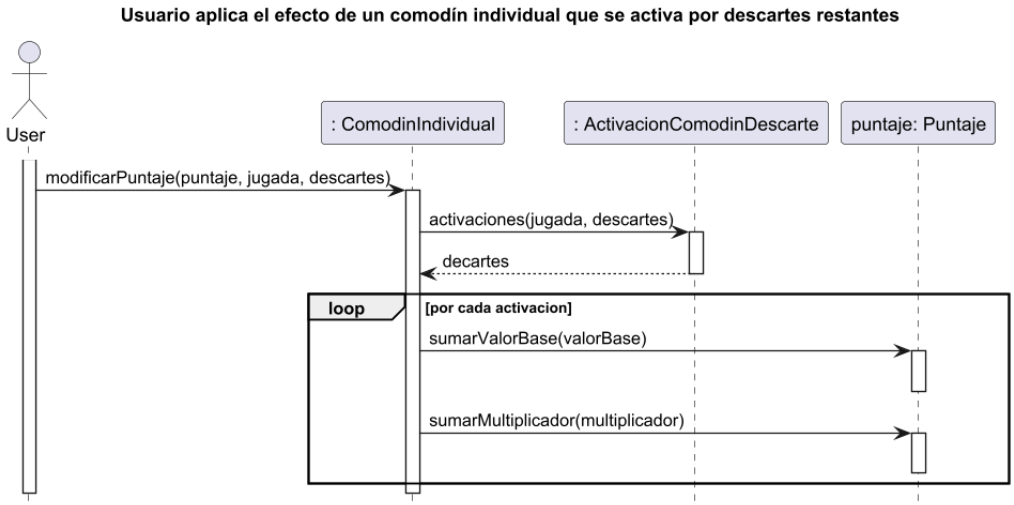

Figura 6: Usuario aplica el efecto de un comodín individual que se activa por descartes restantes

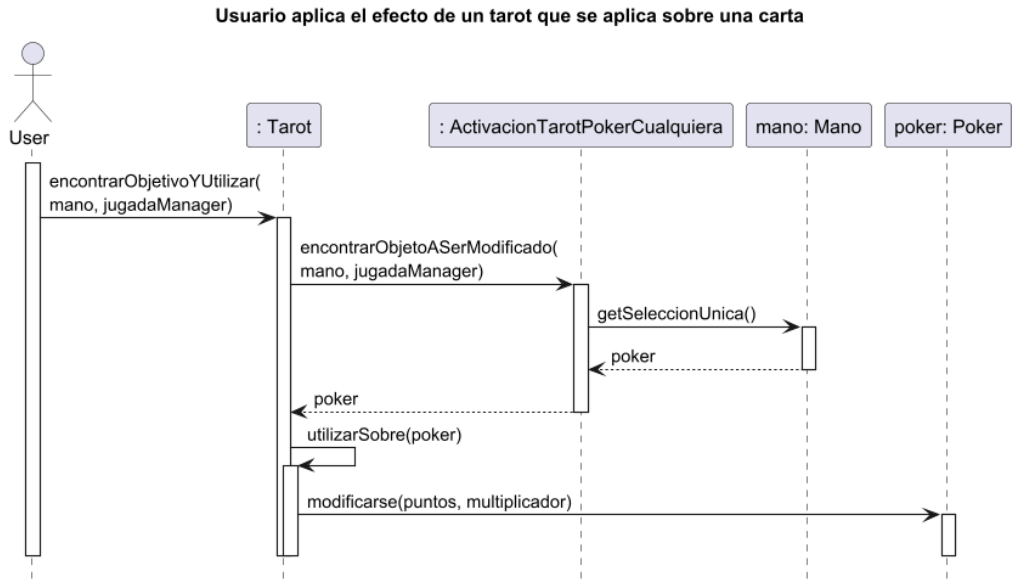

Figura 7: Usuario aplica el efecto de un tarot que se aplica sobre una carta

5. Diagramas de paquetes

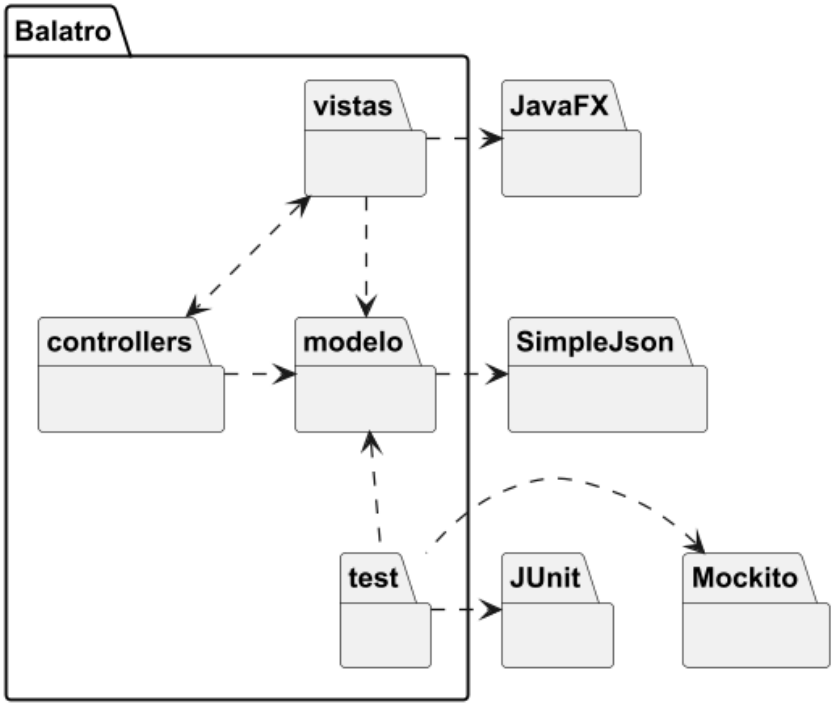

Figura 8: Diagrama de paquetes

6. Detalles de implementación

Es interesante recalcar ciertas decisiones de diseño e implementación, en particular el uso de patrones de diseño. A continuación se listan la mayoría de los patrones utilizados, y un análisis general de posibles áreas de mejora.

6.1. Patrón Observer

Para implementar la comunicación entre la vista y el modelo de una manera con bajo acoplamiento, se utilizaron la clase java.util.Observer y la interfaz java.util.Observable que permiten implementar de manera fácil el patrón en cuestión.

6.2. Patrón MVC

Para separar las distintas responsabilidades del juego, se utilizó el patrón Model-View-Controller, en donde el modelo existe de manera contenida (no depende de la vista ni de los controladores), mientras que la vista y los controladores dependen entre sí, y del modelo. Este patrón funciona muy bien en conjunto al patrón Observer, ya que de esta manera se desacopla el modelo y la vista.

6.3. Comodines y el patrón Composite

Existen ciertos comodines que son una combinación de otros comodines. Esta complejidad puede ser manejada de manera muy directa con el patrón Composite. A continuación se muestran snippets de código que ilustran cómo se utilizó el patrón para implementar los comodines.

```
public abstract class Comodin {
    String nombre;
    String descripcion;

    public abstract void modificarPuntaje(/* ... */);

    /* ... */
}

public class ComodinIndividual extends Comodin {
    int puntos;
    int multiplicador;
    ActivacionComodin activacionComodin;

    @Override
    public void modificarPuntaje(Puntaje puntaje, Jugada jugada, int
    descartesDisponibles) {
        int activaciones = activacionComodin.activaciones(jugada,
    cartasDescartadas);
        for (int i = 0; i < activaciones; i++) {
            puntaje.sumarValorBase(puntos);
            puntaje.sumarMultiplicador(multiplicador);
        }
    }

    /* ... */
}

public class ComodinCompuesto extends Comodin {
    ArrayList<Comodin> comodines;

    @Override
```

```
    public void modificarPuntaje(/* ... */) {
        for (Comodin comodin : comodines) {
            comodin.modificarPuntaje(/* ... */);
        }
    }

    /* ... */
}
```

Listing 1: Patrón Composite aplicado a comodines

El método modificarPuntaje recibe dos parámetros además del puntaje a ser modificado. Estos son requeridos para saber si el comodín se debe activar. Los mismos podrían ser agrupados en un objeto llamado "Acción de jugador".

6.4. Sonidos y el patrón Singleton

Los sonidos son todos manejados por la clase SonidoManager. La misma carga los sonidos a memoria, configura los volúmenes y permite darles play. Ya que los sonidos deben estar disponibles en todas las partes del juego, y todos deben compartir un volumen en común, se decidió usar el patrón ingleton y tener una sola instancia de SonidoManager. Esto también tiene el efecto de ser extremadamente fácil de usar en otras clases. A continuación se muestra código relevante:

```
public class SonidoManager {
    static SonidoManager instancia;
    Map<String, MediaPlayer> sonidos;
    double volumenMusica;
    double volumenEfectos;

    public static SonidoManager getInstancia() {
        if (instancia == null) {
            instancia = new SonidoManager();
        }

        return instancia;
    }

    /* ... */
}

public class Ejemplo {
    public class ejemplo() {
        SonidoManager.getInstancia().play("sonido_ejemplo");
        SonidoManager.getInstancia().setVolumenMusica(0.5);
    }
}
```

Listing 2: Patrón Singleton aplicado a los sonidos

6.5. Patrón Rival

La activación de tarots se implementó de manera muy similar al patrón Rival (Sánchez 2018). Los rivales serían las distintas clases que implementen ModificablePorTarot. En este caso, puede ser una carta de Póker o una Jugada. La responsabilidad de ActivacionTarot es elegir, dentro del "inventario" del jugador, un ModificablePorTarot al cual aplicar el efecto. A continuación se muestra el código de la clase.

```
public interface ActivacionTarot {
    ModificablePorTarot encontrarObjetoASerModificado(Mano mano, JugadaManager
     jugadaManager);
}

public class ActivacionTarotJugadaParticular implements ActivacionTarot {
    Jugada jugada;

    @Override
    public ModificablePorTarot encontrarObjetoASerModificado(Mano mano,
    JugadaManager jugadaManager) {
        return jugadaManager.getJugada(jugada);
    }

    /* ... */
}

public class ActivacionTarotPokerCualquiera implements ActivacionTarot {
    @Override
    public ModificablePorTarot encontrarObjetoASerModificado(Mano mano,
    JugadaManager jugadaManager) {
        return mano.getSeleccionUnica();
    }
}
```

Listing 3: Patrón Rival aplicado a objetivos de tarot

6.6. Timeline y Transition

Para animar partes de la vista se utilizaron los objetos Timeline y Transition provistos por JavaFX. Timeline permite hacer animaciones continuas, como las que se usan en la tienda o en el logo principal. Transition (y sus hijas) se utilizaron para animar las cartas cuando el usuario interactúa con ellas y para animar los números en el panel superior.

6.7. Posibles mejoras

Se identificaron ciertas partes de código problemáticas y que podrían mejorarse mediante refactors o aplicando algún patrón de diseño adicional.

- El único método de ActivacionTarot toma los parámetros Mano y JugadaManager, ya que estos objetos contienen los objetos que implementan ModificablePorTarot. Si en el futuro existiera un objeto adicional que pueda ser modificable por tarot, habría que modificar la firma de todas las clases hijas, lo cual viola el principio Open/Closed.
- Se utilizó el patrón Fachada de manera incompleta. El modelo tiene cierto nivel de acoplamiento, el cual permite a los controladores actuar sobre el modelo mediante unas pocas acciones. Sin embargo, no se implementó el patrón Fachada por completo, sino que existen partes separadas del modelo también. Lo ideal sería tener los objetos del modelo aún más separados, o crear una clase específicamente como Fachada.
- Las vistas están organizadas en pocas clases muy grandes y con muchos objetos cada una. Para poder mantener la vista más fácilmente, se debería modularizar aún más el código. Adicionalmente, se podría usar FXML y CSS para separar el layout, el estilo y la lógica de la vista.
- Algunas clases como Mano quedaron con muchas responsabilidades o funcionalidad. Las mismas podrían ser separadas en clases diferentes y más simples.

- Dentro de ControladorJugar existe un método muy grande y difícil de leer o modificar. Esto se debe a las pausas que se insertaron en forma de funciones anónimas. Esto podría mejorarse mediante una clase dedicada específicamente a concatenar acciones con pausas entre ellas. Además, se podría modificar el juego para que haya aún más pausas entre las acciones de las cartas individuales.

7. Excepciones

Existen varios tipos de excepciones en el programa, pero nunca se muestran al usuario de manera directa. Los controladores manejan las excepciones y las convierten a un sonido de error mediante el uso de try-catch.

- **CartaNoDisponibleError**: Se lanza dentro de Tienda cuando se intenta comprar una carta que no es parte de la tienda.
- **DescartesInsuficientesError**: Se lanza dentro de Juego cuando se intenta descartar, con 0 descartes disponibles.
- **ComodinesLlenoError**: Se lanza dentro de Comodines cuando se intenta agregar un 6to comodín.
- **ManoLlenaError**: Se lanza dentro de Mano cuando se intenta agregar una 9na carta.
- **TarotsLlenoError**: Se lanza dentro de Tarots cuando se intenta agregar un 4to tarot.
- **SinCartasError**: Se lanza dentro de Mazo cuando se intenta tomar una carta, pero el mazo está vacío.
- **SeleccionInvalidaError**: Se lanza dentro de Mano cuando se intenta seleccionar una 6ta carta.
- **JuegoYaTerminadoError**: Se lanza dentro de Juego cuando se intenta jugar un turno después de que el jugador haya perdido o ganado.
- **SeleccionParaTarotInvalidaError**: Se lanza cuando se intenta usar un tarot teniendo varias o ninguna carta seleccionada.
- **JugadaNoEncontradaError**: Se lanza cuando en un JSON no se encuentra la jugada indicada. Además, se lanza si JugadaManager no encuentra la jugada solicitada.
- **SobreNoEncontradoError**: Se lanza cuando en un JSON no se encuentra sobre que carta se aplica el tarot.

8. Conclusión

A partir de la metodología dictada por el TDD, se pudo lograr terminar un proyecto con funcionamiento correcto y código fácil de mantener. Aún así, es posible refactorizar el proyecto para disminuir la entropía en algunas porciones del trabajo. Al ser menos volátil, el modelo terminó siendo el paquete más testeado y más ordenado.

A pesar de no imitar al juego original por completo, se logró completar un juego con gameplay fluido e intuitivo. Adicionalmente, se pudo aumentar su atractivo y facilidad de uso mediante el uso de animaciones, sonidos y música.