Principios SOLID

1. S: Single Responsibility

Este principio se refiere a que una clase debería tener una y solo una, razón para cambiar. Como podemos observar a lo largo del proyecto este principio si se cumple, y podemos verlo en las diferentes clases implementadas (card, player, etc).

2. O: Open/Closed

Este principio se refiere a que se debería de poder extender el comportamiento de una clase, sin modificarla. De igual forma nuestro proyecto si cumple, ya que al estar tan generalizado el código, logramos hacer que una clase pueda extenderse con facilidad, y no necesariamente tener que modificarse.

3. L: Liskov Substitution

Este principio dice que las clases derivadas deben poder sustituirse por sus clases bases, el proyecto cumple con lo mencionado, esto lo podemos observar por ejemplo en las clases de Files y FIlesPanjpar, que podriamos sutituir la clase y el programa seguirá funcionando de forma perfecta, y así lo podemos ver en las diferentes clases derivadas.

4. I: Interface Segregation

Este principio dice que se deben de hacer interfaces que sean específicas para un cliente, o sea para una finalidad en concreto. Consideramos que este principio se cumple, ya que la interfaz creada apunta todo a un mismo objetivo y si bien es cierto que contiene muchos métodos, todos están funcionando y no hay ninguno que esté implementado en vano, por lo tanto el principio se cumple.

5. D: Dependency Inversion

Se refiere a que el código debería depender de abstracciones, y no de clases concretas. Si bien es cierto que en la mayoría del código dicho principio se cumple, no en la totalidad del código, por lo tanto en algunos puntos si se utilizan abstracciones, pero no en todas, por lo tanto el principio se cumple pero no al 100%.

Justificación del diseño

Para el desarrollo del framework y del proyecto en general se hizo uso del patrón Modelo-Vista-Controlador (MVC) en el que nuestro controlador es el encargado de comunicar a la vista con el modelo y actualizar la información de la vista.

Para la agrupación de cartas se realizó una clase general que utiliza el patrón de composición, ya que tanto los grupos de cartas como los grupos de cartas que hereden (En este caso DeckPanjpar) comparten una serie de métodos que hace puedan ser tratados de forma similar.

De igual manera realizamos el uso del patrón Método Plantilla como se puede ver en el controlador abstracto Game específicamente en el método run() que se encarga de ejecutar el juego ya que consideramos que cada juego necesita preparar el o los mazos de cartas, además de crear y mostrar las ventanas de la vista, entre otros, además también se podría mencionar que se utiliza en los métodos saveGame y loadGame de la clase Game, en el caso del Panjpar no se podría usar ese método plantilla ya que se debe actualizar la trumpCard que no comparte con otros juegos.

Con respecto al porqué decidimos realizar el código de esta manera se puede ver lo siguiente:

- Si se trata de agrupaciones de cartas, los jugadores pueden tener o no grupos de cartas (Ejemplo, hay juegos donde no tienen mano y otros en los que podrían tener varias o una zona de cartas descartadas personal), además de que objetivamente hablando un mazo es un grupo de cartas por lo que se podría utilizar esta clase como base para los mazos específicos de los juegos (Vease DeckPanjpar que hereda de GroupOfCards y solo tiene métodos específicos junto a un par de overrides para especificar la función dentro del juego). Para las agrupaciones de cartas para la vista se tienen GroupOfCardView y GroupOfBackView, el primer sirve como contenedor de un grupo de vistas frontales de las cartas y el segundo de la vista trasera, se decidió realizar esta clase ya que un juego podría tener más de un grupo de cartas que se tiene que mostrar o también puede tener diferentes grupos de cartas en las cuales unas se muestran pero otras no.
- Para el controlador abstracto, el validador y otras clases se hizo uso de plantillas para definir el tipo de los parámetros, esto debido a que en muchos casos va a ser necesario utilizar métodos específicos para los juegos y estos no podrían utilizarse en caso de tener parámetros de clases base (Véase llamar métodos específicos de PlayerPanjpar con un objeto de tipo Player).
- Y para finalizar, lo que son métodos de guardado y cargado del estado del juego se decidió hacer que los métodos sean abstractos ya que cada juego puede guardarse de forma diferente dependiendo de

valores específicos (Véase la necesidad de saber que jugador es el atacante y que jugador es el defensor en el Panjpar), pero los métodos para leer cartas, para serializar los jugadores y los grupos de cartas sí pueden ser compartidos entre todos los juegos sí se utiliza el formato definido para las cartas.