AZERTY

Trucardo

Documento de Arquitectura y Diseño

Autores:

- Colque Ventura, Santiago Agustín
- Fernandez, Ignacio Javier
- Lopez Paviolo, Franco Marcelo

Fecha:

- 12 de junio de 2020

Versión 1.0.0
Grupo AZERTY
Año 2020

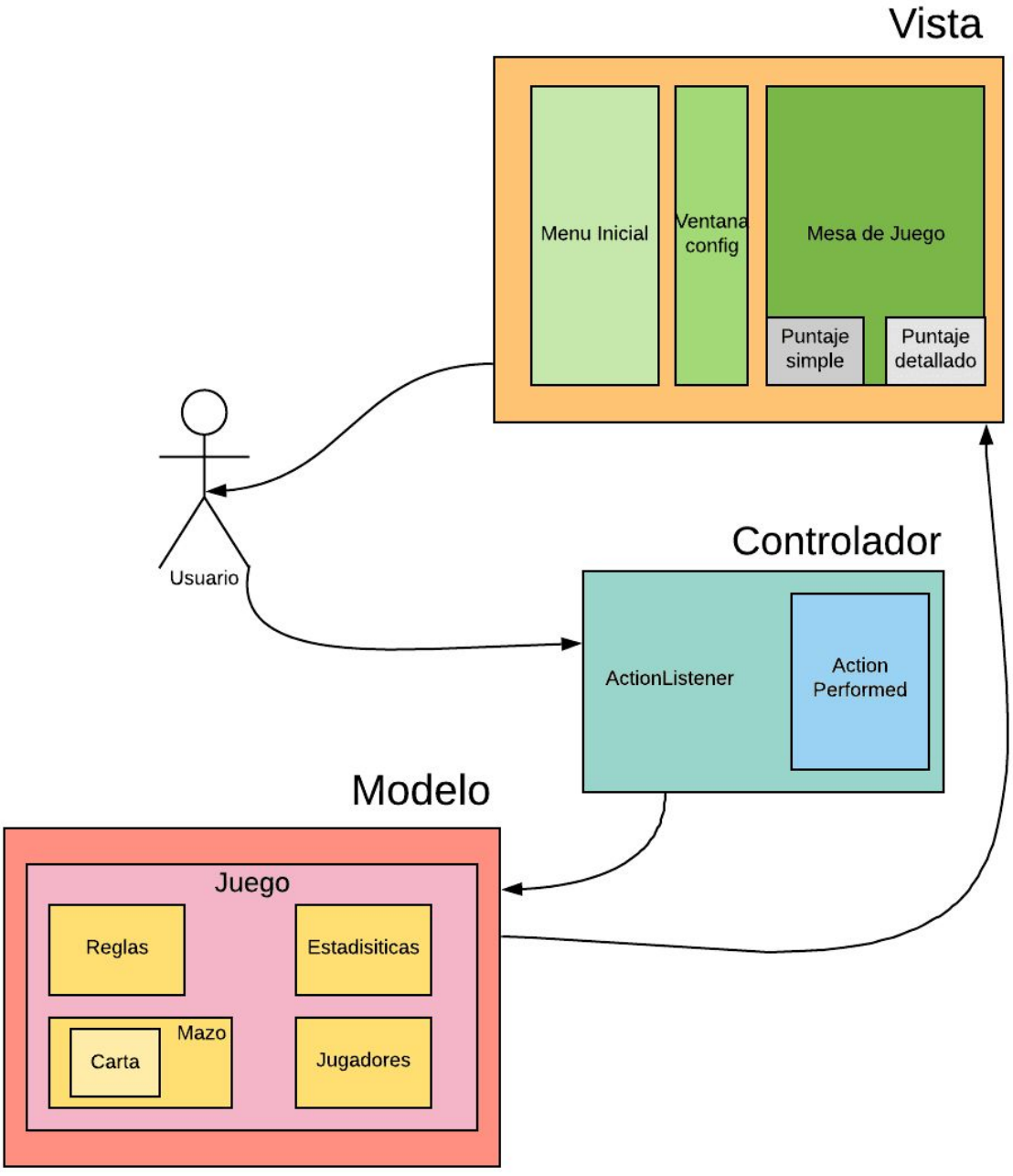
HISTORIAL DE CAMBIOS

Versión	Fecha	Cambios
1.0.0	12 junio 2020	Primera versión

1 INTRODUCCIÓN

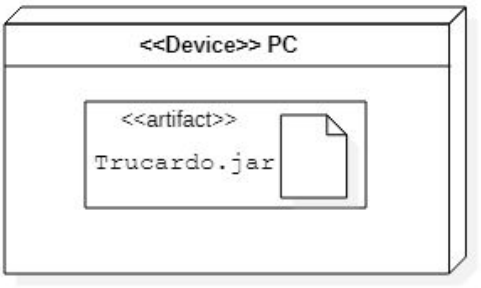
En este documento se detallarán el diseño e implementación de nuestro proyecto además de las pruebas realizadas sobre el sistema. Se realiza la explicación del patrón de Arquitectura elegido mostrando a su vez los patrones de diseño utilizados. Además, se encontrarán a lo largo del documento diagramas UML que acompañan a la explicación del proyecto. por otra parte, Se incluyen las pruebas de integración continua y las pruebas unitarias.

2 PATRÓN DE ARQUITECTURA



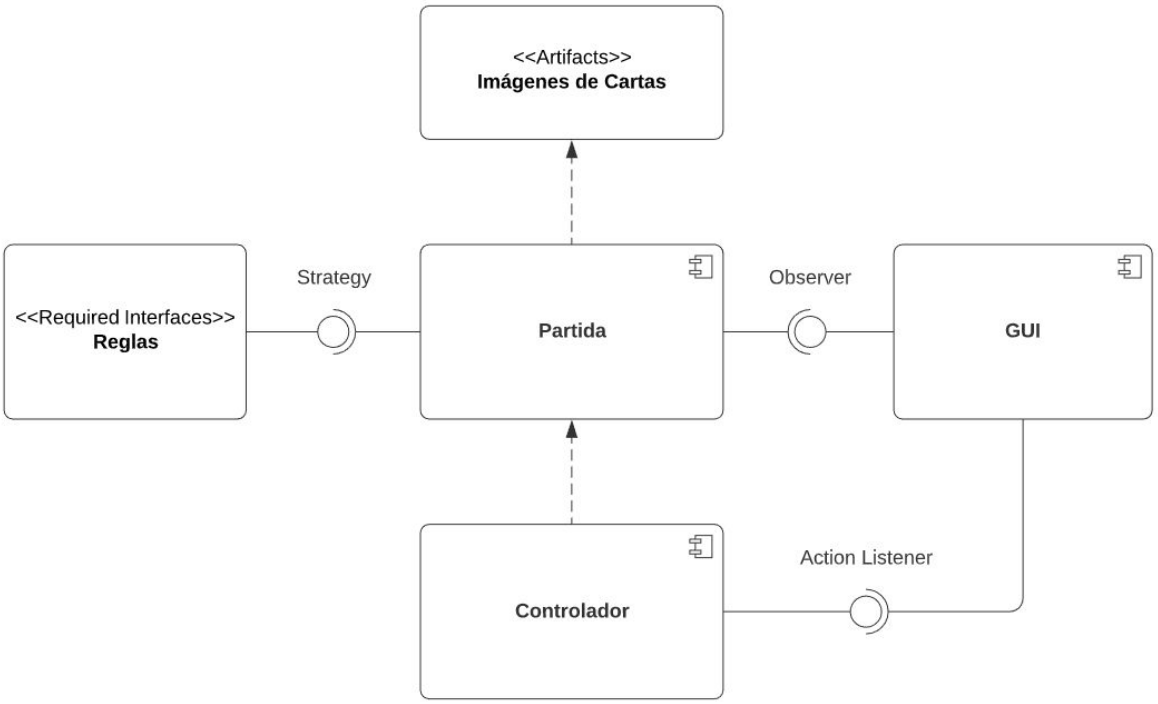
El patrón de arquitectura elegido para Trucardo es el MVC. Se puede observar en la imagen anterior las componentes que caracterizan este patrón de arquitectura. El usuario o jugador (actor externo) actúa sobre el controlador, en nuestro caso presionando botones que representan la acción que quiere realizar, por ejemplo: cantar truco. El controlador manipula al modelo, que fue implementado con un módulo o clase llamado Juego. Este módulo lleva las estadísticas de la partida, cuáles son los jugadores, las reglas de juego, etc. A partir de ahí, el modelo actualiza la vista (todos los componentes que sean necesarios).
Este patrón de arquitectura hace uso de dos patrones de diseño: Observer y Strategy que serán explicados en el apartado de diseño de sistema. Este patrón de arquitectura nos facilita cumplir con el requerimiento no funcional RN1, que especifica que se deben usar los patrones de diseño Observer y Strategy. El MVC nos permite implementar el patrón Observer ya que al tener módulos separados del modelo (la partida) y las vistas (la GUI) se necesita que la segunda se vaya actualizando segun se realizan cambios en el modelo, específicamente se utilizará Observer para actualizar los puntajes en la Vista, entre otras cosas.

Diagrama de despliegue:



Debido a que nuestro proyecto no contiene componentes de hardware ya sea como base de datos, servidores u otros, nuestro diagrama de despliegue se resume en nuestro medio de ejecución, la PC, y nuestro programa el cual se ejecuta en ese ambiente.

Diagrama de componentes:



El proyecto cuenta con tres componentes, la Partida (Modelo), la GUI (Vista) y el Controlador.
El módulo Partida hace uso de un set de reglas provistos mediante el patrón Strategy para definir las reglas que regirán la partida. Las imágenes de cartas son archivos que se utilizarán para mostrar en la pantalla el diseño correspondiente a las cartas.
El módulo de GUI mediante el patrón de diseño observer obtiene los datos que necesita de la Partida que se encuentra en curso, mientras que el Controlador manipula la Partida con la información que recibe por ser un Action Listener de la GUI.

2.1 PRUEBAS DE INTEGRACIÓN

Los siguientes casos de test están pensados para que corran automáticamente.

Test case ID: TI 001

Descripción: Se probará si funciona la integración entre la partida, los jugadores y el mazo.

Función a testear: Integración de componentes en partida.

Preparaciones previas: N/A.

Ejecución del test(pasos):

1. Crear un objeto de clase Partida
2. Iniciar una mano (reparte cartas)
3. Comprobar las cartas de los jugadores

Resultado esperado: Los jugadores deberían tener 3 cartas cada uno, y estas cartas deberían tener valores dentro de las reglas del Truco.

Test case ID: TI 002

Descripción: Se probará si funciona la integración de las reglas de la partida. Las reglas se implementan con el patrón de diseño Strategy y se pueden cambiar dinámicamente durante la partida.

Función a testear: Integración de partida con set de reglas. (Strategy)

Preparaciones previas: N/A.

Ejecución del test(pasos):

1. Crear un objeto de clase Partida.
2. Setear como reglas las Reglas Tradicionales.
3. Pasar 2 cartas como parámetro al set de reglas (cartas con distinto número y distinto palo).
4. El set de reglas devuelve cuál es la carta más alta.
5. Cambiar el set de reglas a Reglas Alternativas
6. Pasar 2 cartas como parámetro al set de reglas (cartas que en el set de reglas tradicionales tengan otro orden de jerarquía).
7. El set de reglas devuelve cuál es la carta más alta.

Resultado esperado: Primeramente el set de reglas debe devolver cual es la carta más alta de acuerdo al valor asignado en las reglas tradicionales de truco. Luego en el segundo intento éste debe devolver cual es la carta más alta de acuerdo a las reglas alternativas definidas.

Test case ID: TI 003

Descripción: Se testea que el observer funcione y el observador GUI Mesa se actualice con resultados correctos.

Función a testear: Integración de observador Mesa con partida. (Observer)

Preparaciones previas: N/A.

Ejecución del test(pasos):

1. Crear un objeto Partida
2. Registrar un objeto Mesa como observador de la partida
3. Modificar el puntaje de un jugador de la partida.
4. Actualizar los observadores del objeto partida
5. Comprobar que el puntaje mostrado en la mesa se haya actualizado correctamente

Resultado esperado: El label que muestra el puntaje en la GUI se debería actualizar correctamente. Si modificamos el puntaje de un jugador a 1, entonces el texto del label debe ser "1".

Test case ID: TI 004

Descripción: Se testea que el observer funcione y el observador Puntaje Detallado se actualice con resultados correctos.

Función a testear: Integración de observador Puntaje Detallado con partida. (Observer)

Preparaciones previas: N/A.

Ejecución del test(pasos):

6. Crear un objeto Partida
7. Registrar un objeto puntaje detallado como observador de la partida
8. Modificar el puntaje de un jugador de la partida.
9. Actualizar los observadores del objeto partida
10. Comprobar que el puntaje mostrado en la ventana de Detalle se haya actualizado correctamente.

Resultado esperado: El label que muestra el puntaje en la ventana de Detalle se debería actualizar correctamente. Si modificamos el puntaje de un jugador a 1 punto de truco, se debe observar en la ventana el punto detallado.

Test case ID: TI 005

Descripción: Se testea que la partida realice correctamente una jugada a través del controlador.

Función a testear: Integración de partida con controlador.

Preparaciones previas: N/A.

Ejecución del test(pasos):

1. Pasarle al controlador que el usuario quiere jugar una carta
2. El controlador manipula la partida para que el jugador juegue la carta
3. Las cartas disponibles en la mano del jugador disminuyen en 1 carta

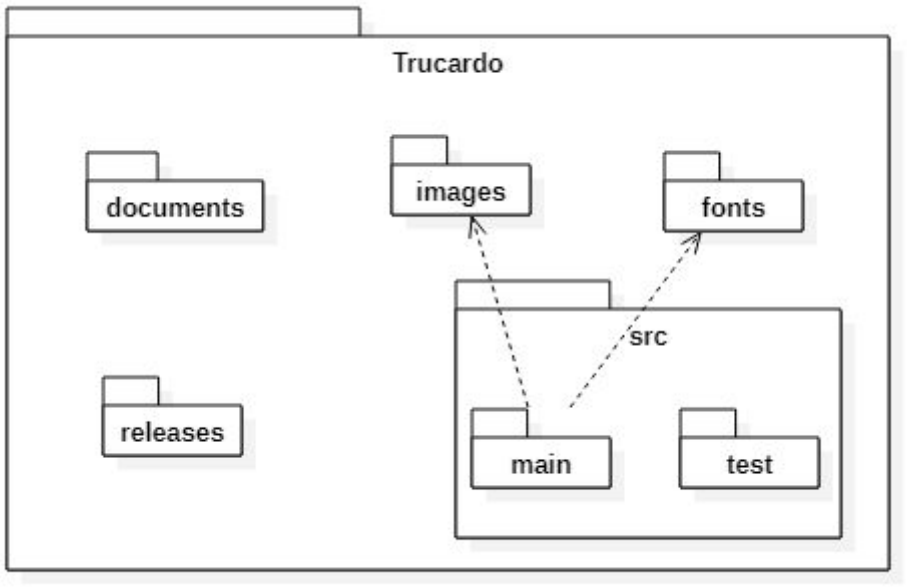
Resultado esperado: Una vez realizado el test, el tamaño de la lista de cartas disponibles que tiene el jugador pasa a ser igual a dos.

3 DISEÑO DEL SISTEMA

El patrón Strategy es implementado en el modulo Reglas. Este módulo será implementado con una Interfaz de la que luego dos clases implementarán y de esta manera se podrá cambiar dinámicamente en tiempo de ejecución el comportamiento de la partida. Una de las clases será de Reglas Convencionales, donde las cartas tendrán los valores descritos en las reglas básicas del truco original; mientras que la otra será una de Reglas Alternativas, que se regirá con una jerarquía de cartas distinto al convencional.

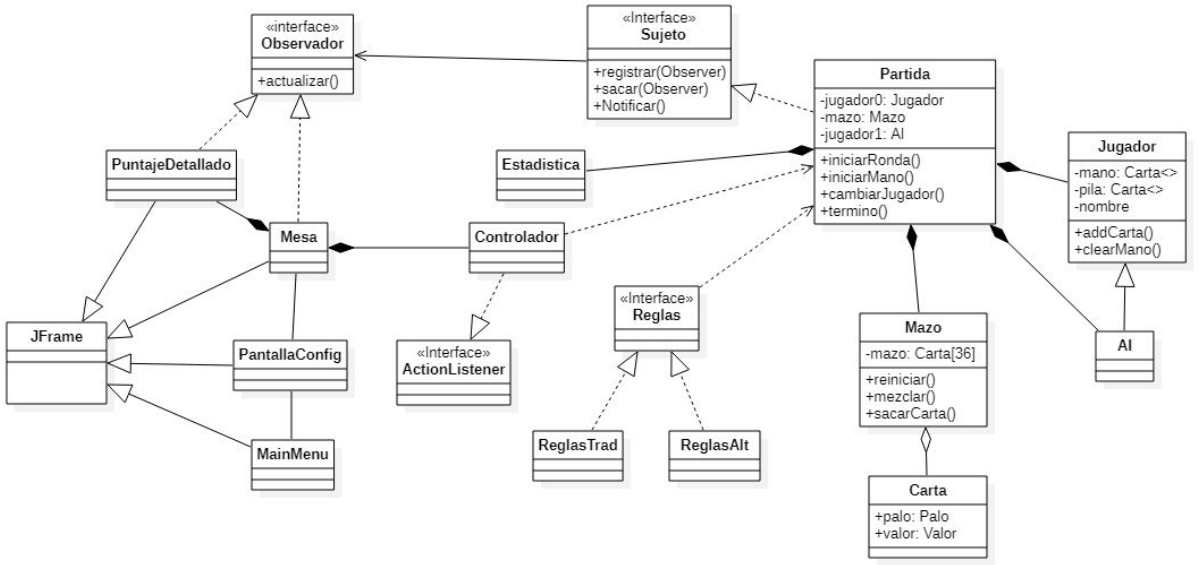
Por otro lado, el patrón Observer está implementado de manera que las vistas (Puntajes, vista principal, etc.) implementan de la interfaz Observer y tienen el método update(), de esta manera el modelo actualiza las vistas siempre que se modifique algo.

3.1 DIAGRAMA DE PAQUETES

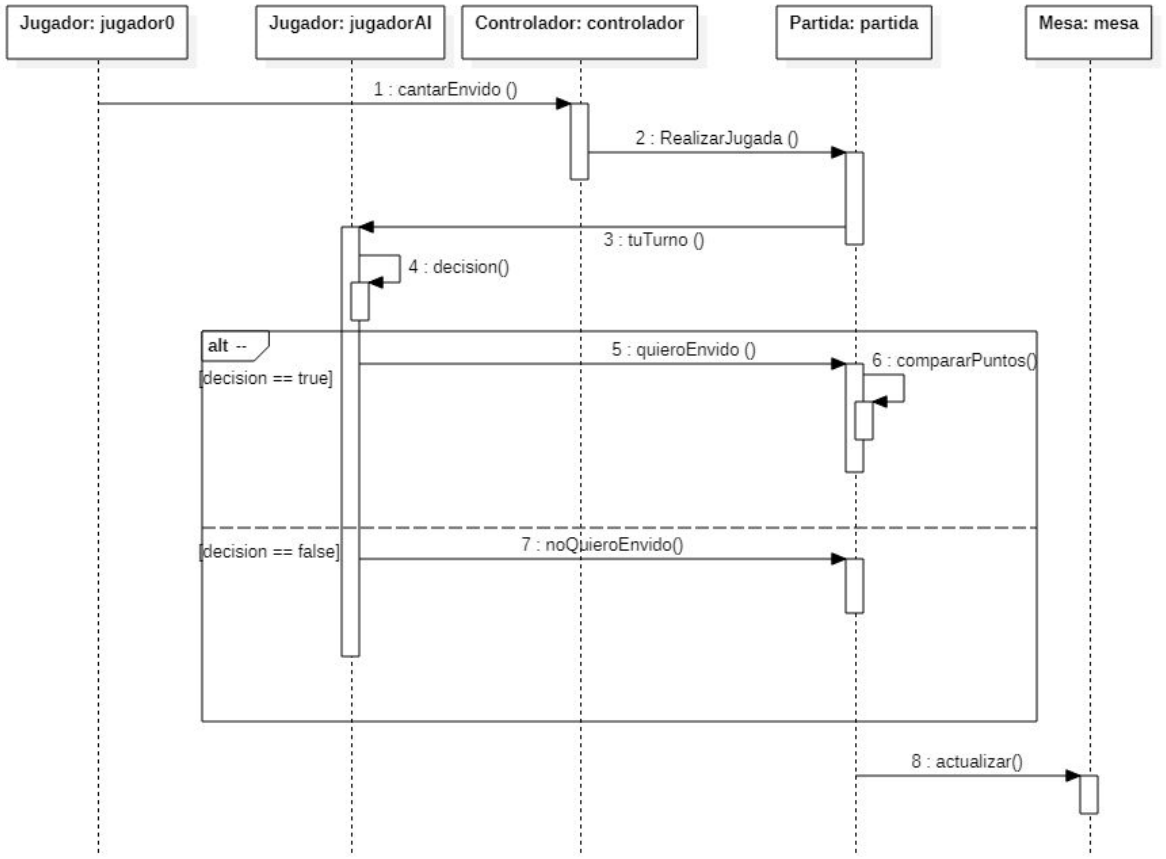


En el anterior diagrama se muestra la estructura que tendría el repositorio y las dependencias formadas dentro de este.

3.2 DIAGRAMA DE CLASES

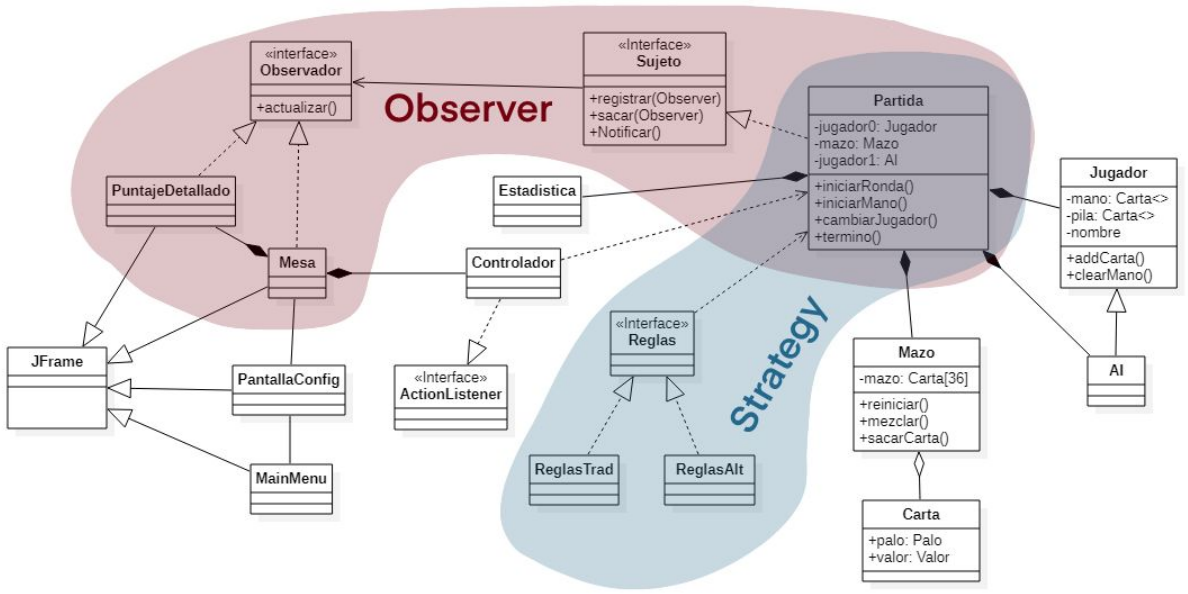


3.3 DIAGRAMA DE SECUENCIA



En este diagrama mostramos a alto nivel lo que sería el funcionamiento del sistema cuando en una partida empezada se realiza un canto de envido. De igual manera, el resto de los cantos tendrían una estructura parecida al anterior diagrama.

3.4 OBSERVER Y STRATEGY



Como se puede observar en el diagrama de clase sombreado, separamos las clases que se ven incluidas en cada uno de los patrones de diseño.

Para el caso del observer, están incluidas obviamente las interfaces de Sujeto y Observador las cuales son requeridas para implementar este patrón de diseño. De la interfaz de Observador tenemos dos Clases que la implementan: PuntajeDetallado y Mesa. Estas dos clases son ventanas de nuestro programa las cuales necesitan actualizarse constantemente con cada cambio que se de en la partida. Particularmente, PuntajeDetallado necesitará los cambios que se den en los puntajes de los jugadores al igual que la clase Mesa, que también necesita actualizar las cartas que debe mostrar. Por otro lado tenemos el Sujeto, la clase Partida, la cual se encargará de avisar a sus observadores de cuando se ha producido algún cambio tanto en los puntajes como en las cartas. Por esta razón el Observer es muy beneficioso ya que no es necesario que los "observadores" pregunten constantemente si algo cambio en la Partida, sino que esta última les avisa cuando así sea.

Para Strategy se ven incluidas la interfaz Reglas y las clases que lo implementan: RegrasTrad y ReglasAlt, y por supuesto la clase Partida. La clase Partida tiene seteado un conjunto de reglas a utilizar para determinar la carta más alta y lo que permite este patrón de diseño es que en tiempo de ejecución se pueda cambiar de forma dinámica el conjunto de reglas al que se le desea consultar.

3.5 PRUEBAS UNITARIAS

En este apartado se detallaran algunas pruebas unitarias, como así también como correrlas y verificar su estado.
Para la codificación se utiliza el framework JUnit. Estas pruebas se encontrarán en la clase AppTest dentro del paquete "test" que se puede observar en el diagrama de paquetes. Para correr estas pruebas localmente, simplemente debemos ejecutar el comando "mvn test" en el GIT Bash dentro de nuestro repositorio local. De esta forma se realiza un Build, se corren todas las pruebas definidas en esa clase, y se obtienen los resultados por consola.
Además de correrlas localmente para verificar que no se hayan roto algunas funcionalidades, gracias a la herramienta de Integración Continua "CircleCI", cada vez que se hace un push al repositorio en la nube se realiza un Build y se corren todas las pruebas automáticamente. En caso de que haya algún error en alguna de las pruebas, la herramienta notifica por mail automáticamente a todos los programadores que trabajan en el proyecto. Esto permite evitar problemas cuando se realizan merges que no tienen un conflicto en su compilación, pero si tienen algún error en algún método en la ejecución.
Algunos ejemplos de pruebas unitarias son los siguientes:

Test case ID: TU 001

Descripción: Probar funcionalidad de contar puntos para envido de la clase Jugador.

Función a testear: Contar puntos de envido.

Preparaciones previas: N/A.

Ejecución del test(pasos):

1. Crear un objeto de la clase jugador
2. Setear la mano del jugador con las cartas: 3 de espada, 5 de espada y 11 de copa. (Estos valores son arbitrarios pero importantes para la verificación del funcionamiento del método)
3. Ejecutar el método que devuelve cuantos puntos tiene el jugador.

Resultado esperado: El método debe devolver 28 puntos.

Test case ID: TU 002

Descripción: Limpiar mano de jugador.
Función a testear: Limpiar mano.

Preparaciones previas: N/A.

Ejecución del test(pasos):

1. Crear objeto de clase Jugador.
2. Ejecutar addCarta(Carta carta) 3 veces.
3. Ejecutar clearMano().

Resultado esperado: La lista utilizada para contener las cartas de la mano del jugador debe tener un tamaño igual a 0 (cero)

Test case ID: TU 003

Descripción: Testeo del método registrar() en la implementación de la interfaz Sujeto (Clase Partida)

Función a testear: Registrar Observador.

Preparaciones previas: N/A.

Ejecución del test(pasos):

1. Crear un objeto de clase Partida.
2. Comprobar que la lista de observadores de ese objeto tenga un tamaño igual a 0.
3. Crear un objeto de tipo Observador.
4. Ejecutar el método registrar(Observador) del objeto de Partida creado.

Resultado esperado: La lista que contiene los observadores registrados al sujeto debería tener un tamaño igual a 1 (uno)

De la misma forma que los Test anteriores, se intentará cubrir todas las funcionalidades con algún Test Unitario que compruebe su buen funcionamiento. A medida que se avance con el código se definirán mas Tests para abarcar así más funcionalidades.