2021

Testigos de Turing

Documento de arquitectura y diseño

Autores:

- Ciordia Cantarella , Francisco
- Martinez, Brenda Sofia
- Fernandez, Santiago
- Rao, Maximiliano
- Villane, Santiago

Tabla de contenido

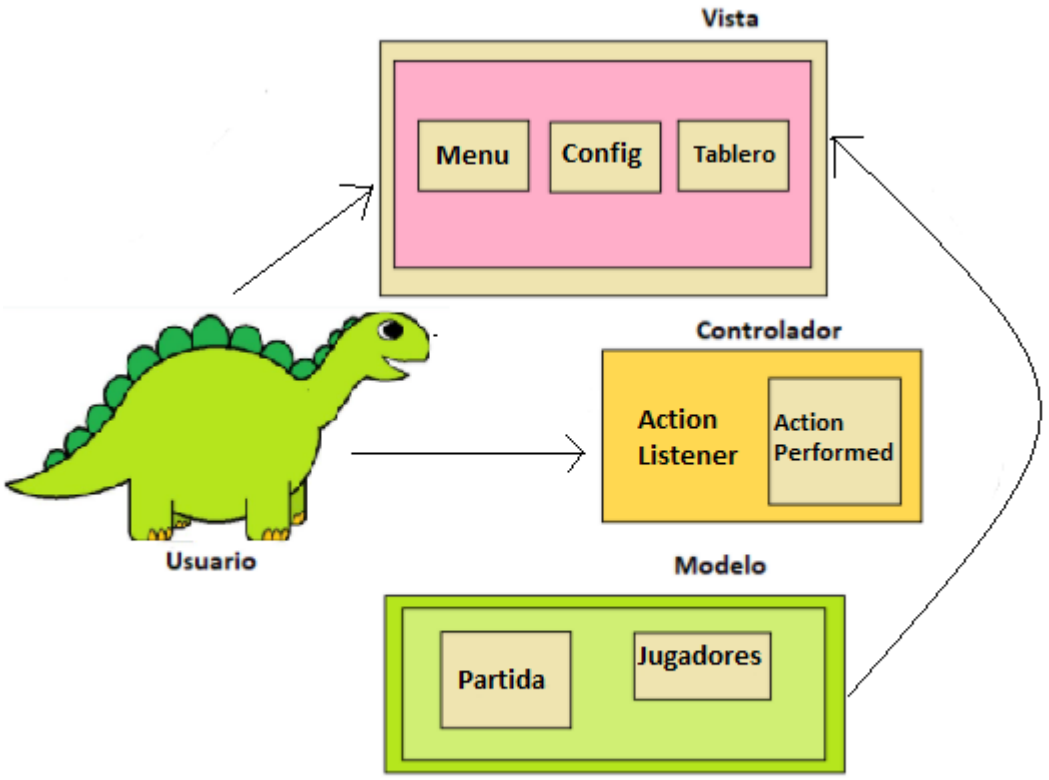
HISTORIAL DE REVISIONES

Versión	Fecha	Resumen de cambios	Autores
1.0.0	03/06/21	Creación de documento	Testigos de Turing
1.1.0	05/06/21	Se incorporan diagrama de clases y secuencia	Testigos de Turing
1.1.1	11/06/221	Correcciones 1#	Francisco Ciordia

1. INTRODUCCIÓN

En este documento se observará el diseño e implementación del proyecto. Se explica el patrón de arquitectura mostrando a su vez los patrones de diseño utilizados. Además las pruebas realizadas sobre el sistema.

2. PATRÓN DE ARQUITECTURA

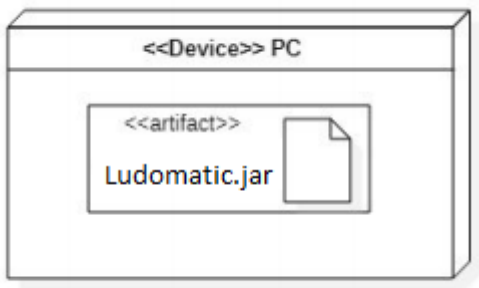


El modelo de arquitectura elegido por Ludomatic es MVC. Puede ver los componentes que caracterizan este patrón arquitectónico en la figura anterior. El usuario o jugador (participante externo) opera el controlador, en nuestro caso presionando un botón que representa la operación que desea realizar. Modelo de operación del controlador, que es implementado por un módulo o clase llamado Juego. Este modelo lleva las características del juego, a saber, jugadores, fichas, etc. A partir de ahí,el modelo actualiza la vista (todos los componentes que sean necesarios).

Este patrón arquitectónico utiliza dos patrones de diseño: Observer y Strategy , que se explicarán en la sección de diseño del sistema. MVC nos permite implementar el patrón de observador porque necesitamos módulos separados para el modelo (juego) y la vista (GUI) Cuando se realicen cambios en el modelo, se actualizará el segundo. El Observer se utilizará exclusivamente para actualizar la posición de la pieza, ilustrar sus estados y para el historial de jugadas.

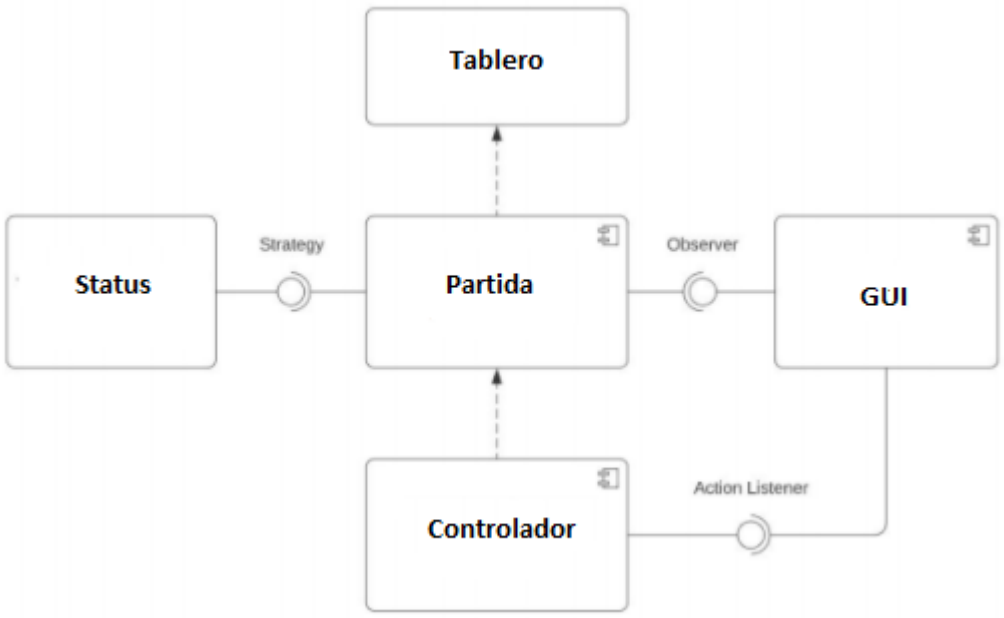
Por otro lado; mencionó por qué se usa MVC porque nos permite separar los componentes de nuestra aplicación en función de sus responsabilidades, lo que significa que cuando hacemos cambios en algunas partes del código, esto no afectará a la otra parte. Por ejemplo, si modificamos el comportamiento de la pieza, solo deberíamos modificar el modelo responsable de las piezas, mientras que el resto de la aplicación debería permanecer sin cambios. Esto respeta el principio de responsabilidad exclusiva. En otras palabras, parte del código no debe saber qué está haciendo toda la aplicación, solo debe ser responsable de una.

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 Status provistos mediante el patrón Strategy para definir el estado de cada pieza del tablero.
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 tablero.

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 partida (tira dados)
3. mover al jugador

Resultado esperado: Los jugadores deberían deberían poder tirar el dado y moverse los casilleros correspondientes.

ID: TI 002

Descripción: Se verificará que el cambio de status de la pieza se haga.

Función a testear: Integración de la interfaz Status y Ficha .

Preparaciones previas: N/A.

Ejecución del test(pasos):
1. Se hace una jugada que cambie el estado de la pieza.
2.Se mueve la pieza en cuestión.

Resultado esperado: Al verificar el status de la pieza, esta debería ser distinta al previo a la jugada.

ID: TI 003

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 tirar dado
2. El controlador manipula la partida para que el jugador juegue la ficha 1. Las fichas disponibles en la mano del jugador disminuyen a 2.

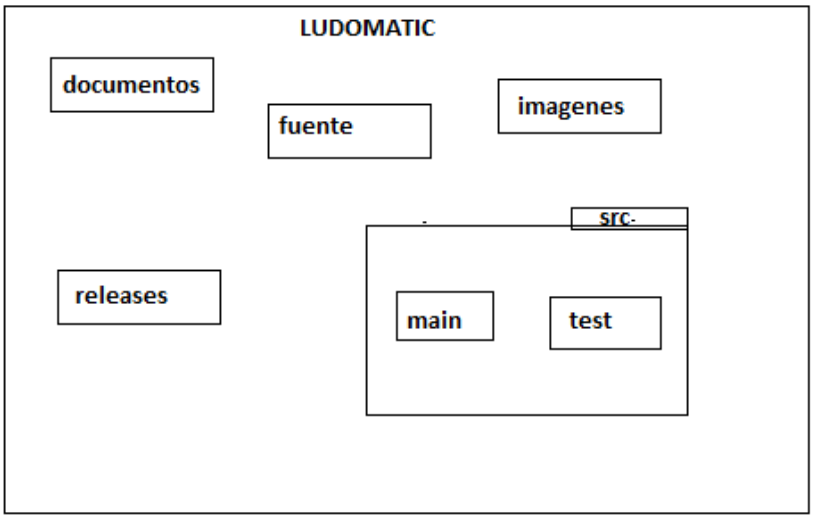
Resultado esperado: Una vez realizado el test, el tamaño de las fichas disponibles que tiene el jugador pasa a ser igual a dos.

3. DISEÑO DEL SISTEMA

El patrón Strategy se implementa en el módulo de Status. El módulo se implementará con una interfaz y luego se implementarán 4 clases para que el comportamiento del juego se pueda cambiar dinámicamente en tiempo de ejecución. Una de las clases será del estado "Safe", donde las piezas en este estado no pueden ser tomados por otras piezas de otro color; la otra será "Win", que será el estado de las piezas que lleguen exitosamente al centro; en por otro lado, está "Moving", cuál es el estado de las piezas en el tablero, y el "Wait" son para aquellas piezas que todavía están esperando en la posición inicial a la espera de un 6. El método responsable de cambiar el estado está en la clase Fichas.

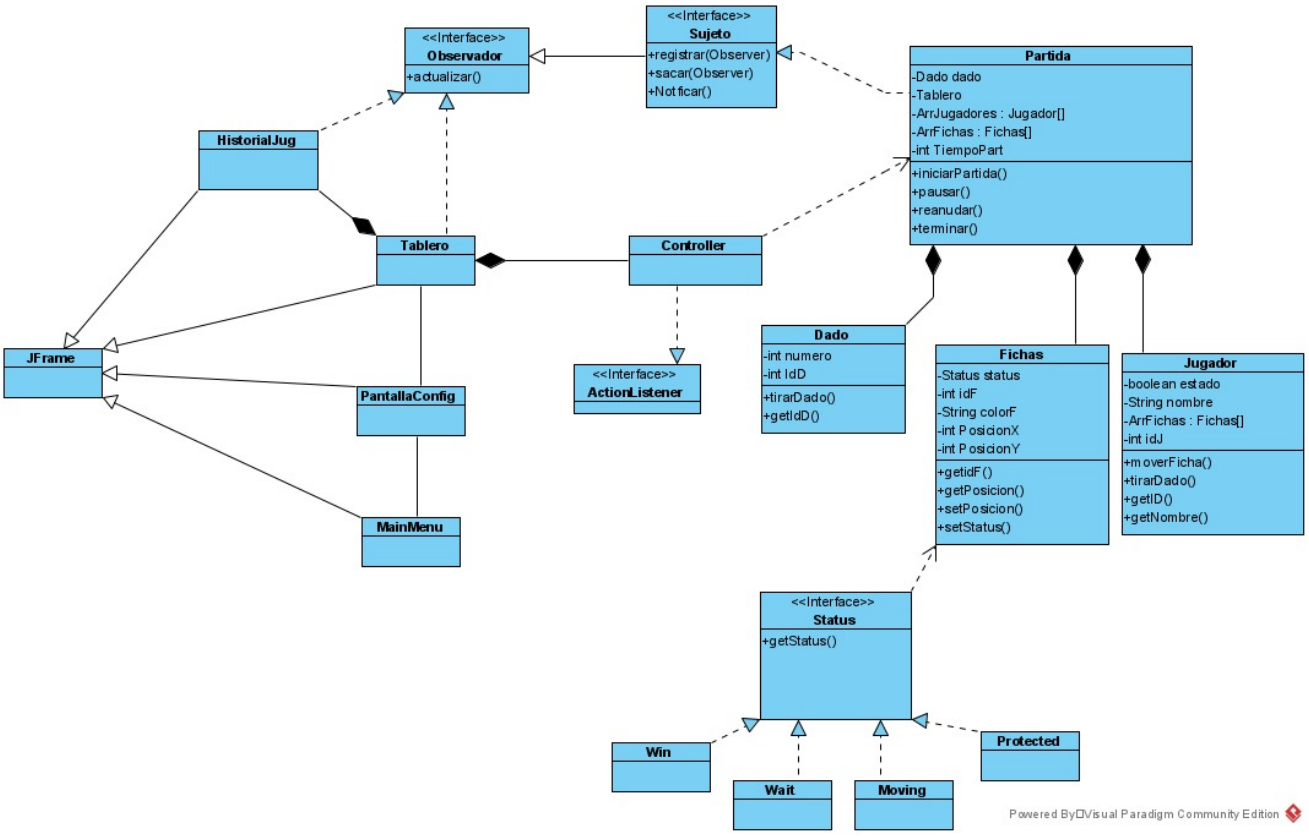
Por otro lado, la forma de implementar el patrón Observer en que la vista (historial de jugadas, vista del tablero) implemente la interfaz del observador y tenga un método update (), de modo que el modelo actualice la vista cuando se modifica algún contenido.

3.1. DIAGRAMA DE PAQUETES

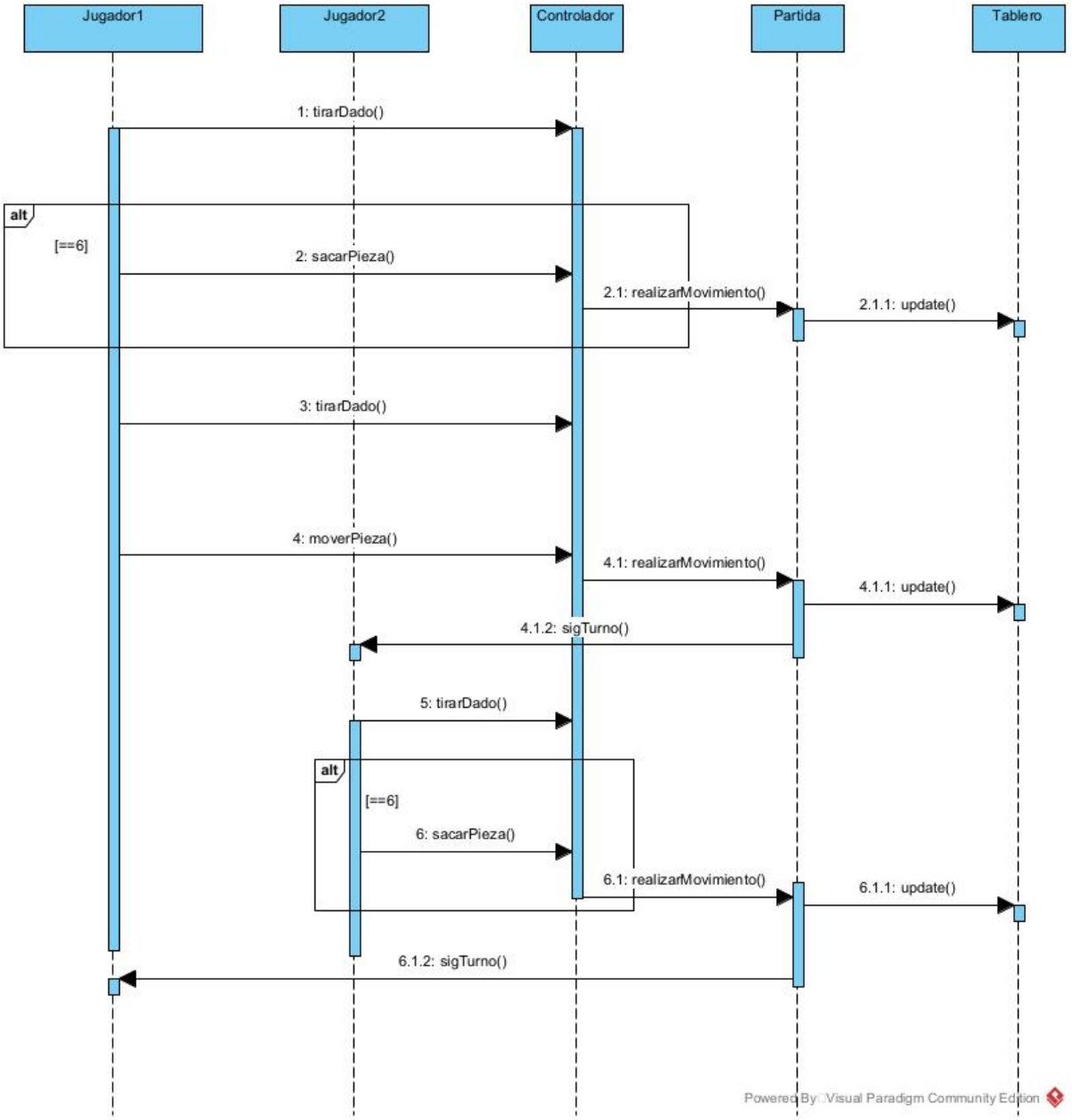


En el anterior diagrama se muestra la estructura que tendría el repositorio y las dependencias formadas dentro de este.

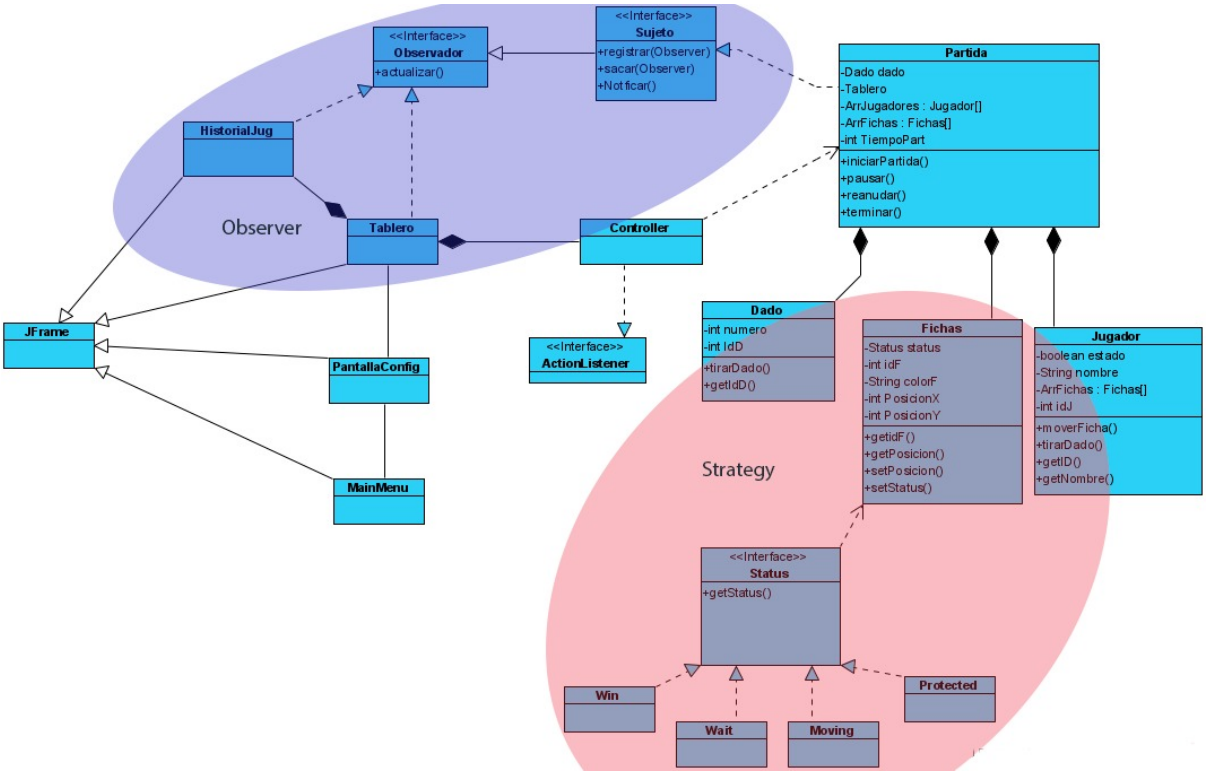
3.2. DIAGRAMA DE CLASES



3.3. DIAGRAMA DE SECUENCIA



3.4. OBSERVER Y STRATEGY



Nota: En la sección 3. se describió una clase “Safe”, nos vimos obligados a cambiar protected que se ve en la imagen para evitar conflictos con las palabras reservadas del IDE.

Como podemos observar en el diagrama de clase sombreado, separamos las clases que se ven en cada uno de los patrones de diseño.
Para el caso del observer, están incluidas las interfaces de Sujeto y Observador ya que son requeridas para implementar este patrón de diseño. De la interfaz de Observador tenemos dos Clases que la implementan: HistorialJug y Tablero. Estas dos clases son ventanas que deben actualizarse constantemente con cada cambio que se de durante la partida. HistorialJug necesitará los cambios que se den en los movimientos de los jugadores al igual que la clase Tablero, que también se actualiza acorde un jugador avance por los casilleros.
También tenemos el Sujeto, la clase Partida, la cual se encargará de avisar a sus observadores de cuando se ha producido algún cambio durante la partida ya sea en el historial de movimientos, como los cambios que se observen en el tablero. Debido a esto el Observer es de gran utilidad, ya que no es necesario que cada “observador” pregunte constantemente si hay algún cambio en la Partida, sino que esta última les avisa cuando así sea.
Para Strategy usamos la interfaz “Status” y las clases que lo implementan: Win, Wait,Moving y Protected,además de la clase Fichas.
La clase Fichas tiene seteada la posición de la ficha y su estado, las cuales nos permiten saber en qué parte del tablero nos encontramos y en qué estado se encuentra.

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 “GitLab”, 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.
Algunos ejemplos de pruebas unitarias son los siguientes:

Test case ID: TU 001

Modulo: Clase partida.

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

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)

Test case ID: TU 002

Modulo: Clase Dado

Descripción: Testeo del método Tirar dado

Función a testear: Obtener número aleatorio

Preparaciones previas: N/A.

Ejecución del test(pasos):
1. Crear un objeto dado
2. Ejecutar metodo tirar Dado.

Resultado esperado: El método deberá devolver un número aleatorio entre 1 y 6.

Test case ID: TU 003

Modulo: Clase Ficha

Descripción: Testeo del casillero protegido del tablero.

Función a testear: La ficha deberá quedar en Status Protected cuando se sitúe encima de este casillero

Preparaciones previas: N/A.

Ejecución del test(pasos):
1. Hacer set posición en una posición “Safe” (protegida) del tablero
2. La ficha deberá cambiar automáticamente su estado a Protected

Resultado esperado: La ficha estará en ese casillero, se ejecutará el método getStatus de la misma y deberá devolver “Protected”.

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 más Tests para abarcar así más funcionalidades.