PRÁCTICA 2 TICTACTOEGAME

AMPLIACIÓN DE INGENIERÍA DEL SOFTWARE

DANIEL AGUDO RIVAS

DANIEL LORENZO ALONSO

Repositorio de GitHub: https://github.com/DanielLorenzo/d.agudor-d.lorenzoal.git

Contenido

Introducción: 2

Pruebas Unitarias: Clase Board. 2

BoardTest 2

Pruebas con Dobles: clase TicTacToeGame. 2

TicTacToeGameTest 2

Pruebas de sistema 3

TicTacToeWebTest 3

Pruebas de integración continua con Jenkins 4

Introducción:

En esta memoria vamos a abordar el desarrollo de las pruebas realizadas a la aplicación **TicTacToe** donde se simula el juego de las tres en raya. Se han desarrollado diferentes pruebas unitarias, dobles y de sistema.

Las pruebas se han realizado con tecnología JUnit 5 y Eclipse Sts.

Pruebas Unitarias: Clase Board.

BoardTest

Para llevar a cabo esta prueba hemos declarado un objeto *board* de la clase Board así como dos jugadores en este caso Strings en el que cada uno de ellos representa el *label o símbolo* que introduce en el tablero, siendo para el jugador 1 "x" y para el jugador 2 "o".

Hemos elaborado 3 **@test** para cada una de las situaciones que se nos presentaban: `ganaJugador1(), ganaJugador2()` y `empatanJugadores()`. Para ello hemos "simulado" el juego de varias partidas para ello hemos asignado a las diferentes celdas el label de cada jugador con el método de la clase Board `getCell( )`. A continuación, hemos guardado en un array, para cada jugador, las celdas obtenidas al llamar al método de la clase Board `getCellsIfWinner()` finalmente hemos realizado las aserciones como se pedían en los requisitos utilizando `assertNotNull() y assertNull()` para el caso en el que gana sea uno u otro jugador respectivamente.

Este procedimiento lo hemos realizado tanto para probar el caso en el que gana el jugador 1 como el 2.

Para el caso de la simulación del empate se ha realizado una jugada de la misma forma explicada anteriormente donde ambos jugadores empatan y comprobamos usando la aserción `assertTrue()` con el método `checkDraw()` de la clase Board.

Todas las pruebas realizadas han devuelto los valores esperados.

La reutilización de código en esta prueba es escasa debido a las diferentes situaciones que se nos dan donde las posiciones que simulamos varían. Solo hemos podido automatizar la inicialización de los jugadores con el método `iniciarJugadores()` donde dados dos jugadores nos inicializan sus respectivos labels, así como el método `obtenerCeldas()` que pasado un jugador como parámetro nos devuelve el array obtenido al llamar al método de la clase Board `getCellsIfWinner().`

Pruebas con Dobles: clase TicTacToeGame.

TicTacToeGameTest

En esta prueba se han realizado diferentes comprobaciones atendiendo a los requisitos de estas. Primeramente, hemos creado el objeto TicTacToeGame y creado los dobles de las conexiones con mockito: Connection conexion1 = *mock*(Connection.class).

Se han creado los objetos Player que se corresponden con los dos jugadores que van a simular la partida. En nuestro caso, hemos creado un array de jugadores donde vamos a meter a los mismos.

Esta prueba admite gran modularidad por lo que vamos a indicar los métodos que utilizamos y el desarrollo de estos:

- `establecerConexion(),` este método recibe el objeto TicTacToeGame, las 2 conexiones que hemos creado (dobles), ambos jugadores y el array que vamos a utilizar para almacenar ambos jugadores. En este método usamos el método `addconnection()` de la clase TicTacToeGame para añadir ambas conexiones al juego.
  Añadimos el jugador 1 al array de jugadores y al juego y hacemos el verify con el evento JOIN GAME como se pide en los requisitos de la prueba Verify(c1).sendEvent(eq(EventType.JOIN_GAME), eq(arrayJugadores)). Esto se realiza para ambas conexiones. Se procede a hacer lo mismo con el jugador 2.
  Después de realizarse los *verify* se procede a comprobar los turnos para ambas conexiones del jugador 1 (arrayJugadores.get(0)).
  verify(c1).sendEvent(eq(EventType.SET_TURN),eq(arrayJugadores.get(0))).
  Para finalizar este método, aprovechamos la modularidad para llamar a otro método que hemos creado el cual recibe dos conexiones y procede a resetearlas resetearConexiones(c1,c2).
- `verificarCambioTurno()` que recibe ambas conexiones, el array de jugadores, así como un entero que hemos utilizado como times para hacer los verify. Este método se encarga de verificar que se envía el evento SET_ TURN y se procede por tanto al cambio de turno, se verifica para ambas conexiones, así como para ambos jugadores como se pide en la prueba.
- `verificarCaptor()` el cual se encarga de verificar los parámetros que se pasan.

Posteriormente implementamos tres **@test** `GanaJugador1()`, `GanaJugador2(),` `Empatan().`

En los tres métodos simulan jugadas llamando al método `mark()` de la clase TicTacToeGame, en cada caso se produce a una jugada diferente y sus respectivos cambios de turno, obtenemos el valor del captor con el método anterior y procedemos a realizar las aserciones como se solicita en la prueba, finalizado el proceso se resetean las conexiones. Para el @test de empate, el proceso es el mismo a excepción de la comprobación final en la que comprobamos si el valor del captor es **null** en cuyo caso se produce el empate, así como un AssertEquals que verifica si el método de la clase TicTacToeGame `checkDraw()` devuelve un true.

Todas las pruebas realizadas han devuelto el valor esperado.

Pruebas de sistema

TicTacToeWebTest

Para llevar a cabo las pruebas de sistema hemos declarado varios métodos que nos permiten aprovechar la modularidad del código y evitar por tanto la reutilización del mismo, estos métodos son:

- `inicializarNavegadores()` – Este método añade dos jugadores que se pasan como parámetro a un array de jugadores. Utilizamos dos WebDriver para inicializar los dos navegadores necesarios para llevar a cabo las pruebas, concretamente con la dirección http: localhost:8080. Posteriormente este método utiliza 4 WebElement (2 para cada navegador) para localizar dentro de la pagina los elementos que buscamos por id n1-finElement(By.id("nickname")) que son, el nombre del jugador y el botón para poder cargar el nombre en el campo y pulsar el botón. Se hace con `sendKeys().`
- `analizaAlert()` – Este método espera a que salga el mensaje de alerta indicando el estado de la partida, gana jugador1/2 o empate, hemos establecido dos minutos(120''), este tiempo se podría adaptar y hacer que espere más o menos. Analiza

el texto del Alert con el método navegador1.switchTo().alert().getText() proporcionado, creando un array de Strings para recuperar el valor de dicho alert. Devolvemos el array.

En el setupClass incializamos los dos navegadores (Google Chrome) como `ChromeDriver()` y procedemos a los test de sistema. Se simulan jugadas para cada navegador pulsando los botones que simulan las mismas, utilizamos el método `click().` Para comprobar quien gana se analiza el contenido del array que devuelve el método analizaAlert, recogemos el valor y hacemos las aserciones pertinentes. Todas las pruebas realizadas han devuelto el valor esperado.

Pruebas de integración continua con Jenkins

Para probar la integración continua hemos utilizado Jenkins y pipeline.

El pipeline que hemos creado lo guardaremos en un **Jenkinsfile**, el cual subimos a nuestro repositorio de Github junto a la aplicación TicTacToe para que lo ejecute cuando vayamos hacer las pruebas. Los pasos que hemos seguido para la realización son los siguientes:

1. En Jenkins creamos una nueva tarea e iremos a Pipeline.
2. Una vez en Pipeline, marcamos la opción "Pipeline script from SCM", marcamos la opción Git e incluimos nuestro repositorio de GitHub.
3. Una vez realizados los pasos anteriores, podemos comprobar si los test de la aplicación TicTacToe han tenido éxito o si, por el contrario, se han producido errores.

Una vez comprobamos que los test podemos comprobar en Jenkins el resultado obtenido de forma visual, nos ofrece información como el tiempo que ha tardado en realizar cada tarea, así como las ejecuciones que ha hecho del pipeline.

Si se hubiesen obtenido errores, tenemos la posibilidad de visualizarlos por consola.

EL fichero **Jenkinsfile** define las fases de las que consta nuestro flujo, por ejemplo, instalación de Maven, el jdk, cargar el repositorio de GitHub, compilar y verificar código, y realizar los test.

El Pipeline se define de la siguiente forma:

- **Tools** - Carga las herramientas que vamos a utilizar para pasar los test.
- **Agent** -Al poner *'any'*, hacemos que el pipiline se ejecute siempre.
- **Stages** - Definiremos los estados que tendrá nuestro pipeline.
  - Comando **checkout scm** para cargar el repositorio de GitHub.
- **Stage** - Definiremos las tareas que vamos a realizar.
  - Compile, compilar utilizando la herramienta de Maven.
  - Test, verificaremos que pasan los test, y los ejecuta, también utilizando Maven.
  - Install, realiza la misma operación Test guardando una copia del resultado de forma local.
- **Post** - se encarga de recoger los resultados tanto satisfactorios como los reportes negativos.