Proyecto Tictactoe

Actividad Individual

Pregunta: Explica el funcionamiento de los siguientes código dentro del sprint1.

```
public class TestBoardConsole {
   private Board board;

   @Before
   public void setUp() throws Exception {
         board = new Board();
   }

   @After
   public void tearDown() throws Exception {
   }

   @Test
   public void testEmptyBoard() {
         new Console(board).displayBoard();
   }

}
```

```
public class Console {
   private Board board;

   public Console(Board board) {
         this.board = board;
   }

   public void displayBoard() {
         for (int row = 0; row<3; row++) {
               System.out.println("-------");
               System.out.print("|"+ board.getCell(row, 0));
               System.out.print("|"+ board.getCell(row, 1));
               System.out.print("|"+ board.getCell(row, 2));
               System.out.println("|");
         }
         System.out.println("-------");
   }
}
```

Respuesta:

El primer código se tiene la clase TestBoardConsole, el cual realiza pruebas a las clases Board y Console. Para ello define el método setUp, el cual es llamado antes de cada prueba. Este método instancia un objeto de la clase Board, con el que inicializa el campo board. Este campo es usado en el método testEmptyBoard, al ser pasado como argumento al constructor de Console para instanciar un objeto de la clase Console. Sobre este objeto se llama al método displayBoard, el cual muestra el tablero vacío en la consola.

El segundo código es la implementación de la clase Console. Tiene un constructor, al cual debe pasarsele un atributo tipo Board con el que se inicializa su campo board. Se define el método displayBoard, el cual imprime en la consola el tablero 3x3, usando para ello el campo board y llamando su método getCell para obtener el valor de cada celda.

Pregunta: ¿se necesita refactorización?

Respuesta: Luego de implementar los tests para cuando se haga referencia a una fila o columna no válida, se tuvo que refactorizar el método getCell de la clase Board para que solo devuelva el valor de la celda cuando la fila y columna estén entre 0 y 2. Para otros valores se retorna -1 para indicar que la fila o columna son inválidas.

Pregunta: Realiza la cobertura de código. Explica tus respuestas.

Respuesta: Al realizar las pruebas unitarias usando la clase TestEmptyBoard2 se obtiene una covertura del 85% de las líneas (6 de 7). Esto se debe a que en las pruebas nunca se ejecuta la línea "return -1" del método getCell, ya que no se llama a este método con valores inválidos de fila o columna.

Pregunta (V/F): La secuencia de cuatro movimientos, `X (0,0), O (1,1), X (0,1), O (1,0)` no cumple la necesidad.

Respuesta: Falso. Si cumple la necesidad, ya que no existen XXX u OOO aún y es el turno de X.

Pregunta:

Para hacer que `testXWon` pase, `updateGameState` se enfoca en los escenarios `CROSS_WON`. Indica al menos tres pruebas para `AC4.1` para cubrir tres X seguidas de manera horizontal, vertical y diagonal.

Respuesta:

```
public void testXWonHorizontal(){
    board.makeMove(1, 0);
    board.makeMove(2, 1);
    board.makeMove(1, 1);
    board.makeMove(2, 2);
    board.makeMove(1, 2);

     assertEquals(“ ”, board.getGamesState(); GameState. CROSS_WON);
     new GUI(board);
     try {
          Thread.sleep(2000);
       } catch (InterruptedException e) {
           e.printStackTrace();
      }
}
```

```
```
public void testXWonVertical(){
    board.makeMove(0, 0);
    board.makeMove(1, 1);
    board.makeMove(1, 0);
    board.makeMove(1, 2);
    board.makeMove(2, 0);

     assertEquals(" ", board.getGamesState(); GameState. CROSS_WON);
     new GUI(board);
     try {
          Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
      }
}
```
```
public void testXWonDiagonal(){
    board.makeMove(0, 0);
    board.makeMove(1, 0);
    board.makeMove(1, 1);
    board.makeMove(1, 2);
    board.makeMove(2, 2);

     assertEquals(" ", board.getGamesState(); GameState. CROSS_WON);
     new GUI(board);
     try {
          Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
      }
}
```

**Pregunta:** Muestra que el método `testXWon` anterior ha cubierto `AC4.2` y `AC 4.4` y que el juego continuó hasta la jugada ganadora `board.makeMove (0, 2)`.

**Respuesta:** El método testXWon cubre AC 4.2 porque luego que X hace el primer movimiento (0,0) aún no se ha formado XXX y el juego continúa y cambia el turno a O.
También cubre AC 4.4 porque cuando O hace su primer movimiento (1, 1) aún no se ha formado OOO y el juego continúa y cambia el turno a X.

**Pregunta:** ¿`AC4.3` es similar a `AC 4.1` ?. ¿Se trata de los escenarios `NAUGHT_WON`?.

**Respuesta:** Son similares pero con la diferencia que AC 4.1 es el escenario CROSS_WON y AC 4.3 es el escenario NAUGHT_WON
```

Pregunta: ¿Toda las pruebas para `AC4.1-AC4.5` permitirán completar la clase de `Board`?.

Respuesta: Si, las pruebas AC4.1-AC4.5 permiten completar la clase Board con métodos que implementen los diferentes escenarios por los que el juego puede terminar.

Pregunta: ¿ Cuál es el problema de initialBoard y por que le cambiamos el nombre a `resetGame`.

Respuesta: Porque al ejecutarse el juego vuelve a su estado inicial, es decir hace un reset, por lo que lo más adecuado sería llamarlo resetGame.

Pregunta: Verifica esto en el código del paquete del proyecto TicTacToe entregado.

Respuesta: Se verifica que se ha creado una clase AutoTicTacToe que extiende TicTacToeGame, la cual implementa el que el jugador pueda jugar contra una computadora. Cuando la computadora empieza, esta juega con X y realiza un movimiento aleatorio válido. Sin embargo no están implementados los métodos makeWinningMove y blockOpponentWinningMove (solo retornan false), los cuales deberían realizar el movimiento ganador si es posible y bloquear una jugada ganadora del jugador. Por lo que cuando la computadora le toca realizar un movimiento solo verifica que lo realize en una celda vacía.