Othello

Introducción y reglas

En este informe se explica como ha sido el método para testear el juego Othello (Reversi) y como se ha desarrollado el juego a raíz de programarlo mediante el uso de la técnica Test Driven Development (TDD).

Para probar el juego se debe compilar el Cliente.

Las reglas de juego son las siguientes:

-Se empieza con 4 fichas en el centro, 2 de cada color. Cada jugador coloca una ficha de su color cada turno.

-Solo se puede colocar ficha en una casilla vacía y si al colocarla en esa posición se flanquea alguna del otro jugador.

-Al colocar una ficha las fichas flanqueadas en vertical, horizontal y diagonal por el otro jugador pasan a ser fichas tuyas

-La partida termina cuando ningún jugador puede hacer colocar ficha o se han jugado 60 turnos, con lo cual se consigue llenar el tablero y nadie puede tirar. Si un jugador no puede tirar se le pasara el turno al otro.

Cuando la partida termina se cuentan las fichas y gana el que más fichas tenga de su color en el tablero.

Test Driven Development

Para realizar este proyecto se ha realizado mediante el método test driven development por lo cual los test, en su mayoría ya que en algunos casos al empezar a codificar el juego se vio que faltaban casos de prueba o algunos no funcionaban correctamente , fueron escritos antes de empezar a codificar nada sobre el juego. Esto ayudó a que el juego ya fuese escrito a prueba de la gran mayoría de los ataques ya que en los tests ya teníamos en mente todos los posibles ataques que podria sufrir mediante exploratory testing y esos ataques que debian de dar fallos ya estaban solucionados en la primera versión del juego. Esto también se debe en gran parte a la experiencia de codificar los test para los dos juegos anteriores siguiendo este mismo método.

Tests:

Hemos realizado tests de caja negra para todos los métodos que tienen inputs i/o outputs. Hemos tenido en cuenta casos especiales y hemos estudiado las partes equivalentes con tal de testear todos los valores límite y frontera.

Hemos realizado tests de caja blanca para todos los métodos más complejos que lo permitían. Hemos realizado path coverage y decision coverage de todo el proyecto. En este documento sólo añadimos los diagramas de flujo de los métodos más complejos. También hemos realizado Loop Testing de todos los loops que nos parecieron interesantes y que podría haber algún problema.

También hemos utilizado el método de automatización. Hemos automatizado la función que se encarga de jugar la partida, con tal de generar partidas aleatorias o definiendo un desenlace deseado.

El método de exploratory testing también ha sido usado, aunque como explicamos, la mayor parte de éste fue prevenido en versiones tempranas del código o fue realizado en una versión anterior a la actual, por lo que la mayoría de problemas encontrados gracias a ello fueron resueltos.

Por último, con tal de facilitar muchos de los tests explicados, hemos usado Mock objects para simular varias cosas. Por un lado lo hemos usado para simular aleatoriedad y user inputs, también para generar escenarios manualmente y poder testear casos especiales o estados tediosos de conseguir. También ha sido de gran ayuda para el exploratory testing y la automatization.

Classes

En este apartado comentaremos los test realizados clase por clase y método por método especificando que esta testeado en cada uno.

Class Menu:

Esta clase permite al cliente poder acceder al juego mostrando un menú en consola pidiendo al cliente entradas por teclado.

MockObjects

```
public class MockMenu extends Menu{

        List<Integer> askIntReturn = new ArrayList<Integer>();
        public int index= 0;
        public int partidas=0;


        public int askInt() {
            int intReturn = askIntReturn.get(index);
            askIntReturn.remove(index);
            return intReturn;
        }

        public void startGame(){
            partidas++;
        }
    }
```

Este mockObject del menú nos permite poder simular las entradas de teclado del cliente sin la necesidad de que se entren a mano haciendo que el método que pide los datos devuelva los datos que nosotros queramos en el orden que queramos ademas de poder saber el numero de partidas “jugadas”.

CheckInput()

Esta función devuelve un booleano true si la opción pasada a la función está dentro de los límites establecidos, en este caso solo devuelve true si es un 1 o un 2.

```java
public boolean checkInput(int input) {
    if (input <= 0 || input > 2)
    {
        return false;
    }
    return true;

}
```

Codigo de Test

```java
/**
 * Checks if the input entered is valid
 * @return Returns true if the value is valid
 */
@Test
public void testCheckInput() {

    //Initialize a menu
    Menu menu = new Menu();

    //Equivalent Partition -inf to 0 -> Invalid input
    boolean res_0 = menu.checkInput(-23);
    assertFalse(res_0);

    boolean res_1 = menu.checkInput(-1);
    assertFalse(res_1);

    boolean res_2 = menu.checkInput(0);
    assertFalse(res_2);

    //Equivalent Partition 1 to 2 -> Valid input
    boolean res_3 = menu.checkInput(1);
    assertTrue(res_3);

    boolean res_4 = menu.checkInput(2);
    assertTrue(res_4);

    //Equivalent Partition 3 to inf -> Invalid input
    boolean res_7 = menu.checkInput(3);
    assertFalse(res_7);

    boolean res_8 = menu.checkInput(25);
    assertFalse(res_8);


}
```

Caja negra

Comprobamos valores límite, frontera , valores dentro de las particiones y el 0.

askInt()

Exploratory Testing

Tanto en el método de la clase Menu como en el de la clase Game, hemos usado el método de exploratory testing (con Mock Objects) para encontrar los posibles errores o problemas relacionados con el input. En versiones tempranas de nuestro juego, lo usamos y encontramos varias fallas en el tratamiento de los datos entrados.

A raíz de ello, la mayoría de los problemos encontrados ya se han corregido en la versión actual. Concretamente, los fallos que encontramos fueron el de no tratar correctamente la entrada de diferentes tipos de datos como strings y en una versión posterior, el problema de recibir nuevas líneas como input.

De todas formas, cabe decir que con exploratory testing hemos podido comprobar que ahora tratamos todo el input que hemos testeado correctamente.

Esto nos llevo a encontrar un problema con el overflow, por un tamaño demasiado grande de datos, pero no en nuestro programa, sino en la consola de Eclipse, la cual parece no ser capaz de mostrar más de 13026 caracteres. Esto se puede comprobar en la versión Oxygen, nuestro test todavía cuenta con esta parte.

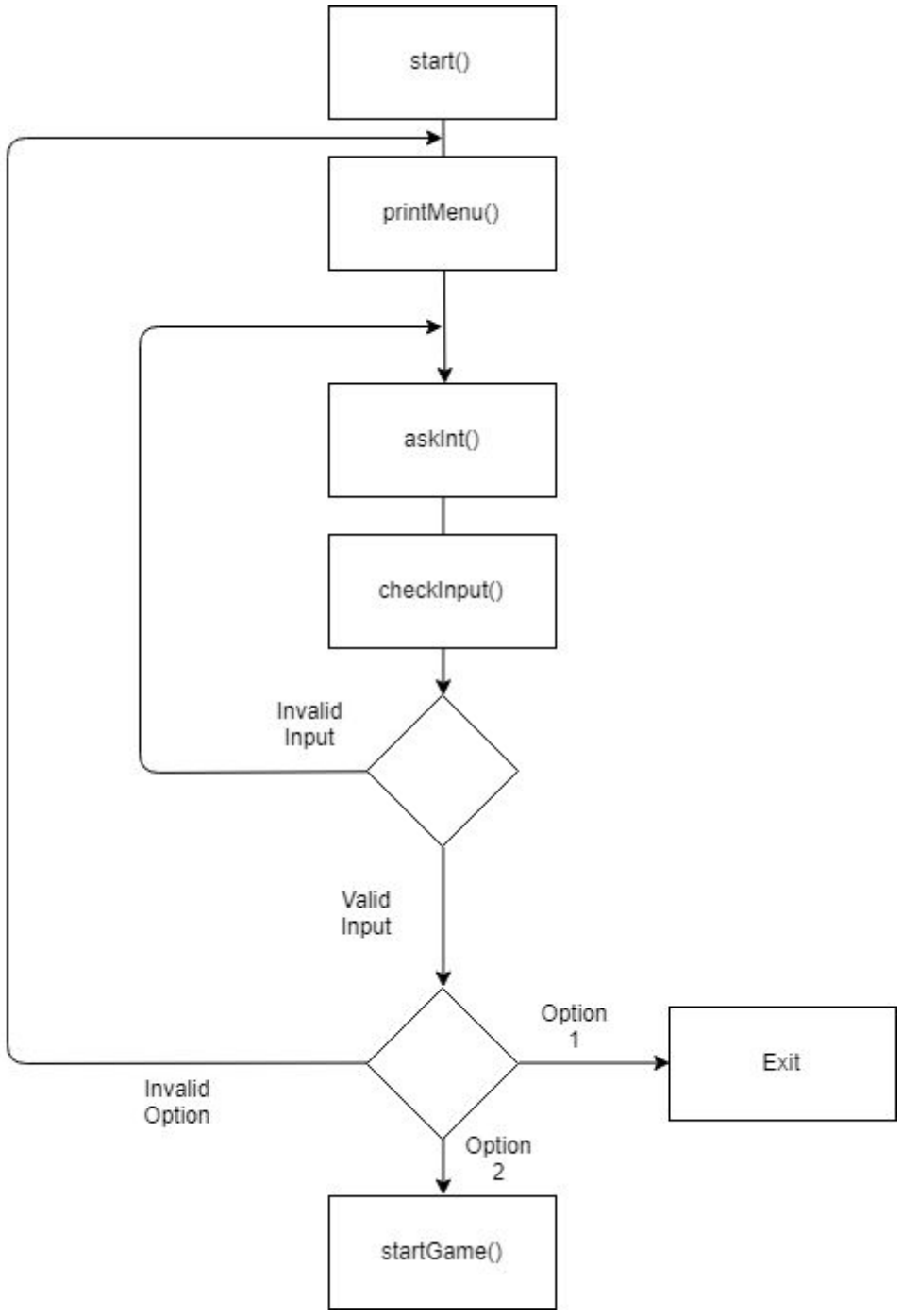
```
                //invalid characters
                 + "'\'" + System.getProperty("line.separator")

                    //invalid characters
                 + "%" + System.getProperty("line.separator")

                    //invalid characters
                 + "^@" + System.getProperty("line.separator")

                    //invalid characters
                 + "^Z" + System.getProperty("line.separator")

                    //invalid characters
                 + "&$|" + System.getProperty("line.separator")

                 //invalid Int
                 + "2147483648" + System.getProperty("line.separator")

                //valid Int, invalid option
                 + "8" + System.getProperty("line.separator")

                 //valid Int, valid option
           + "1" + System.getProperty("line.separator");

        InputStream savedStandardInputStream = System.in;
        System.setIn(new ByteArrayInputStream(simulatedUserInput.getBytes()));

        test.askInt();

        System.setIn(savedStandardInputStream);
```

Start()

Esta función pide un valor al cliente, comprueba si es correcto con la función CheckInput(), si este es false volverá a pedirlo hasta que sea correcto. Si este es un 1 empezara el juego y si es un 2 el programa termina.

```
public void start(){
    int op;
    do{
        this.printMenu();
        do{
            op=askInt();
            if(!this.checkInput(op)){
            System.out.println("Invalid Input");
            }
        }while(!this.checkInput(op));

        if(op==1){
            startGame();
        }
    }while(!(op==2));
}
```

Diagrama de flujo



Codigo de Test

```
public void testStart() {
	MockMenu test1=new MockMenu();

	//Path Coverage
	//The value its correct the first time and its 1

	test1.askIntReturn.add(1);
	test1.askIntReturn.add(2);
	test1.start();
	assertEquals(test1.partidas,1);
	//The value its correct the first time and its 2
	test1.askIntReturn.add(2);
	test1.start();
	assertEquals(test1.partidas,1);
	//The value its not correct the first time and one value its 1
	test1.askIntReturn.add(4);
	test1.askIntReturn.add(1);
	test1.askIntReturn.add(2);
	test1.start();
	assertEquals(test1.partidas,2);
	//The value its correct the first time and and no one value its 1

	test1.askIntReturn.add(6);
	test1.askIntReturn.add(2);
	test1.start();
	assertEquals(test1.partidas,2);

	//LoopTest
	//while(!this.checkInput(op));
	//0 times
	test1.askIntReturn.add(2);
	test1.start();
	assertEquals(test1.partidas,2);
	//1 times
	test1.askIntReturn.add(4);
	test1.askIntReturn.add(2);
```

```
test1.start();
assertEquals(test1.partidas,2);
//2 times
test1.askIntReturn.add(4);
test1.askIntReturn.add(4);
test1.askIntReturn.add(2);
test1.start();
assertEquals(test1.partidas,2);
//3 times
test1.askIntReturn.add(4);
test1.askIntReturn.add(4);
test1.askIntReturn.add(4);
test1.askIntReturn.add(2);
test1.start();
assertEquals(test1.partidas,2);
//7 times
test1.askIntReturn.add(4);
test1.askIntReturn.add(4);
test1.askIntReturn.add(4);
test1.askIntReturn.add(4);
test1.askIntReturn.add(4);
test1.askIntReturn.add(4);
test1.askIntReturn.add(4);
test1.askIntReturn.add(2);
test1.start();
assertEquals(test1.partidas,2);

//while(!(op==2));
//0 times
test1.askIntReturn.add(2);
test1.start();
assertEquals(test1.partidas,2);
//1 times
test1.askIntReturn.add(1);
test1.askIntReturn.add(2);
test1.start();
assertEquals(test1.partidas,3);
//2 times
```

```
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(2);
        test1.start();
        assertEquals(test1.partidas,5);
        //3 times
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(2);
        test1.start();
        assertEquals(test1.partidas,8);
        //7 times
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(1);
        test1.askIntReturn.add(2);
        test1.start();
        assertEquals(test1.partidas,15);

}
```

Caja blanca

Path Coverage

Comprobamos los diferentes caminos:

-El valor es correcto a la primera y es un uno
-El valor es correcto a la primera y es un dos
-El valor no es correcto a la primera y es un uno
-El valor no es correcto a la primera y es un dos

Loop Testing

-while(!this.checkInput(op));
-while(!(op==2));

Comprobamos ambos bucles entrando en ellos 0,1,2,3 y 7 veces.

Class Game:

Esta clase, como dice su nombre, gestiona el juego en si controlando el número de fichas puestas , el número de turnos realizados y va realizando los turnos para los jugadores y cuando termina el juego evalúa quien ha ganado.

MockObjects

```
public class MockGame extends Game{

        List<Integer> askIntReturn = new ArrayList<Integer>();
        public int index= 0;
        public int posX;
        public int posY;

        public int askInt() {
            int intReturn = askIntReturn.get(index);
            askIntReturn.remove(index);
            return intReturn;
        }
    }
```

Este mockObject de Game nos permite poder simular las entradas de teclado del cliente sin la necesidad de que se entren a mano haciendo que el método que pide los datos devuelva los datos que nosotros queramos en el orden que queramos.

```
public class MockGame2 extends Game{
    Random rn = new Random();

    public int askInt() {
    return rn.nextInt() % 8;
    }

    public void restart(){
        turnojugador=false;
        table =new Board();
        turnos=0;
        fichas=4;
        gameOver=false;
    }
}
```

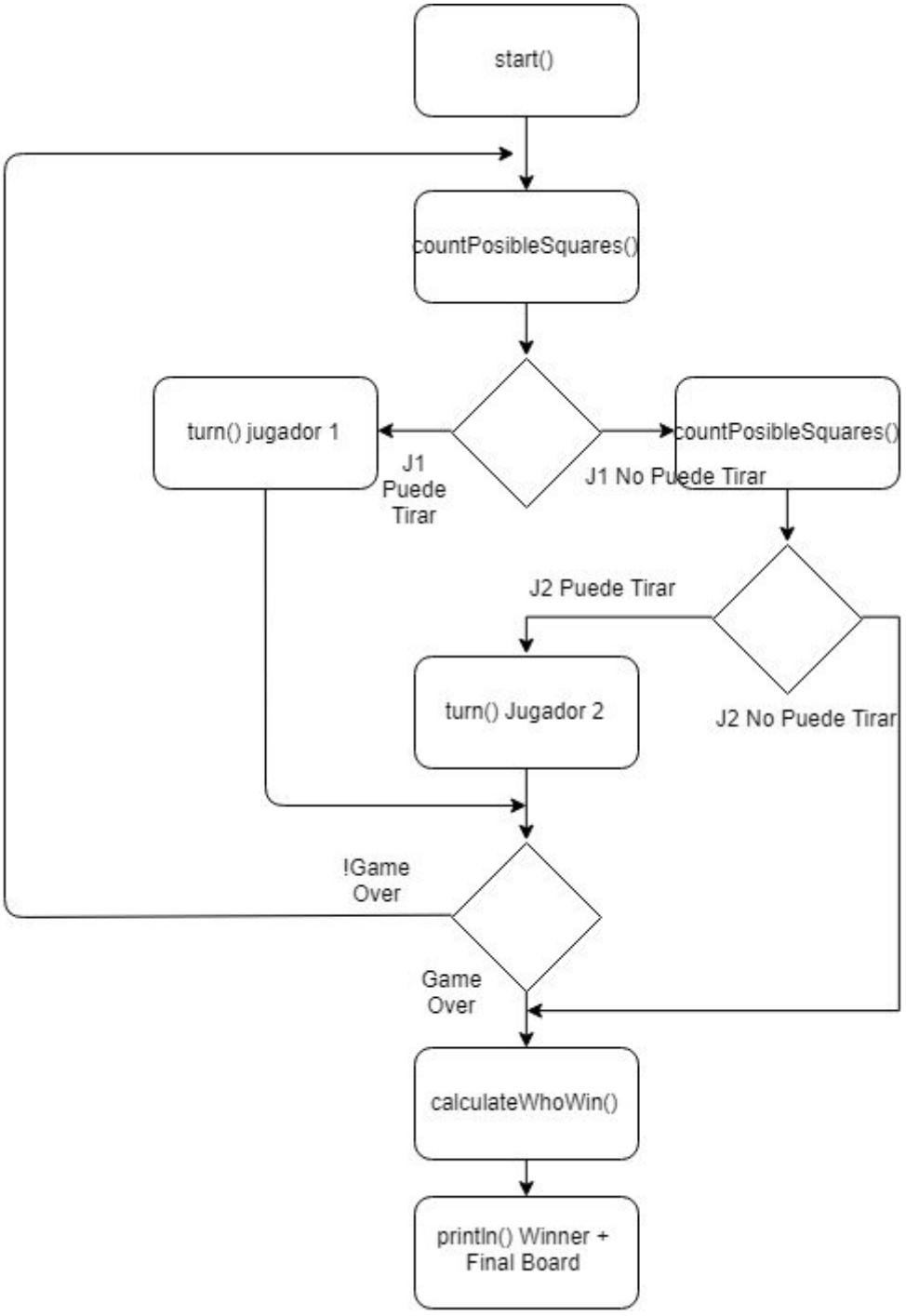
Este mockObject de Game nos permite simular una partida de valores randoms y volver la classe a su estado inicial sin tener que crear un nuevo Game, este mockObject lo usamos para realizar un ataque al juego jugando el numero de partidas que queramos seguidas.

Start()

Este método es el que empieza el juego en si ejecutando todas las funciones de Game y Board haciendo que empiece el turno de un jugador, dependiendo si le toca o si el otro no puede tirar. Cuando el número de turnos llega a 60 o si countPosiblesSquares() tanto pasandole un true o un false da 0. Cuando esto sucede cuenta el número de negras con CalculateWhoWin(), restando el número que devuelve al número de fichas totales obtenemos el numero de blancas y dependiendo cual sea mayor al otro o si son iguales habrá ganado uno o el otro o habran empatado

```
public void start(){

        gameOver=false;

        do{
            turnojugador=!turnojugador;
            if(this.table.countPosibleSquares(turnojugador)>0){
                gameOver=turn(turnojugador);
                fichas++;
            }else{
                turnojugador=!turnojugador;
                if(this.table.countPosibleSquares(turnojugador)>0){
                    gameOver=turn(turnojugador);
                    fichas++;
                }else{
                    gameOver=true;
                }
            }


        }while(!gameOver);

            win=table.calculateWhoWin();
            blancas=fichas-win;

            if(blancas==win){
                System.out.println("\n\n          IT'S A DRAW \n\n");
            }
            else if (win>blancas) {
                System.out.println("\n\n          BLACK WINS \n\n");
            }else{
                System.out.println("\n\n          WHITE WINS \n\n");
            }


        table.drawBoard();

    }
```

Diagrama de flujo



Codigo de Test

```
@Test
public void testStart() {

    //AUTOMATIZATION TEST

    //PLAY 100 RANDOM GAMES
    MockGame2 testAuto=new MockGame2();
    for(int i=0;i<100;i++){
        testAuto.start();
        testAuto.restart();
    }

    //PLAY RANDOM GAMES until DRAW
    MockGame2 testAuto2=new MockGame2();
    do{
         testAuto2.restart();
         testAuto2.start();
    }while(!(testAuto2.blancas==testAuto2.win));

    //PLAY RANDOM GAMES until BLACK WINS
    MockGame2 testAuto3=new MockGame2();
    do{
        testAuto3.restart();
        testAuto3.start();
    }while(!(testAuto3.blancas<testAuto3.win));


    //PLAY RANDOM GAMES until WHITE WINS
    MockGame2 testAuto4=new MockGame2();
    do{
        testAuto4.restart();
        testAuto4.start();
    }while(!(testAuto4.blancas>testAuto4.win));

  // Scenarios Testing

  //Case where game ends because no player can move anymore
  MockGame mockGame1=new MockGame();
  mockGame1.getBoard().changeSquare(3, 4);
  mockGame1.getBoard().changeSquare(4, 3);
  mockGame1.start();
  assertTrue(mockGame1.gameOver);
  assertEquals(mockGame1.fichas,4);
  assertEquals(mockGame1.turnos,0);
  assertEquals(mockGame1.blancas,0);
  assertEquals(mockGame1.win,4);

  //Case where game ends and Black Tabs player wins
  MockGame mockGame2=new MockGame();
  mockGame2.askIntReturn.add(2);
  mockGame2.askIntReturn.add(4);
  mockGame2.askIntReturn.add(4);
  mockGame2.askIntReturn.add(5);
  mockGame2.askIntReturn.add(5);
  mockGame2.askIntReturn.add(4);
  mockGame2.turnos=57;
  mockGame2.start();
  assertTrue(mockGame2.gameOver);
  assertEquals(mockGame2.fichas,7);
  assertEquals(mockGame2.turnos,60);
  assertEquals(mockGame2.blancas,2);
  assertEquals(mockGame2.win,5);
```

```
    //Case where game ends and White Tabs player wins
    MockGame mockGame3=new MockGame();
    mockGame3.askIntReturn.add(4);
    mockGame3.askIntReturn.add(2);
    mockGame3.askIntReturn.add(5);
    mockGame3.askIntReturn.add(4);
    mockGame3.askIntReturn.add(4);
    mockGame3.askIntReturn.add(5);
    mockGame3.askIntReturn.add(3);
    mockGame3.askIntReturn.add(2);
    mockGame3.turnos=56;
    mockGame3.start();
    assertTrue(mockGame3.gameOver);
    assertEquals(mockGame3.fichas,8);
    assertEquals(mockGame3.turnos,60);
    assertEquals(mockGame3.blancas,5);
    assertEquals(mockGame3.win,3);

    //Case where only White player can move (turn skip)
    MockGame mockGame4=new MockGame();
    mockGame4.getBoard().getSquare(3, 2).setColor(true);
    mockGame4.getBoard().getSquare(3, 2).setTab(true);
    mockGame4.getBoard().getSquare(4, 2).setColor(true);
    mockGame4.getBoard().getSquare(4, 2).setTab(true);
    mockGame4.getBoard().getSquare(5, 2).setColor(true);
    mockGame4.getBoard().getSquare(5, 2).setTab(true);
    mockGame4.getBoard().getSquare(5, 3).setColor(true);
    mockGame4.getBoard().getSquare(5, 3).setTab(true);
    mockGame4.getBoard().getSquare(5, 4).setColor(true);
    mockGame4.getBoard().getSquare(5, 4).setTab(true);
    mockGame4.getBoard().changeSquare(3, 4);
    mockGame4.askIntReturn.add(5);
    mockGame4.askIntReturn.add(4);
    mockGame4.turnos=59;
    mockGame4.start();
    assertTrue(mockGame4.gameOver);
    assertEquals(mockGame4.fichas,5);
    assertEquals(mockGame4.turnos,60);
    assertEquals(mockGame4.blancas,-2);
    assertEquals(mockGame4.win,7);
```

```
//Case where only Black player can move (turn skip)
MockGame mockGame6=new MockGame();
mockGame6.getBoard().getSquare(5, 3).setColor(false);
mockGame6.getBoard().getSquare(5, 3).setTab(true);
mockGame6.getBoard().getSquare(5, 4).setColor(false);
mockGame6.getBoard().getSquare(5, 4).setTab(true);
mockGame6.getBoard().getSquare(3, 5).setColor(false);
mockGame6.getBoard().getSquare(3, 5).setTab(true);
mockGame6.getBoard().getSquare(4, 5).setColor(false);
mockGame6.getBoard().getSquare(4, 5).setTab(true);
mockGame6.getBoard().getSquare(5, 5).setColor(false);
mockGame6.getBoard().getSquare(5, 5).setTab(true);
mockGame6.getBoard().changeSquare(3, 3);
mockGame6.askIntReturn.add(4);
mockGame6.askIntReturn.add(2);
mockGame6.turnos=59;
mockGame6.start();
assertTrue(mockGame6.gameOver);
assertEquals(mockGame6.fichas,5);
assertEquals(mockGame6.turnos,60);
assertEquals(mockGame6.blancas,2);
assertEquals(mockGame6.win,3);

//Case where game ends with a draw
MockGame mockGame5=new MockGame();
mockGame5.askIntReturn.add(2);
mockGame5.askIntReturn.add(4);
mockGame5.askIntReturn.add(4);
mockGame5.askIntReturn.add(5);
mockGame5.turnos=58;
mockGame5.start();
assertTrue(mockGame5.gameOver);
assertEquals(mockGame5.fichas,6);
assertEquals(mockGame5.turnos,60);
assertEquals(mockGame5.blancas,3);
assertEquals(mockGame5.win,3);


// Loop Testing

//Number of turns, test with initial Board setup
//1 Turns
MockGame mockGame7=new MockGame();
mockGame7.askIntReturn.add(2);
mockGame7.askIntReturn.add(4);
mockGame7.turnos=59;
mockGame7.start();
assertTrue(mockGame7.gameOver);
assertEquals(mockGame7.fichas,5);
assertEquals(mockGame7.turnos,60);
assertEquals(mockGame7.blancas,1);
assertEquals(mockGame7.win,4);

//2 Turns
MockGame mockGame8=new MockGame();
mockGame8.askIntReturn.add(2);
mockGame8.askIntReturn.add(4);
mockGame8.askIntReturn.add(4);
mockGame8.askIntReturn.add(5);
mockGame8.turnos=58;
mockGame8.start();
assertTrue(mockGame8.gameOver);
assertEquals(mockGame8.fichas,6);
assertEquals(mockGame8.turnos,60);
assertEquals(mockGame8.blancas,3);
assertEquals(mockGame8.win,3);
```

```
//3 Turns
MockGame mockGame9=new MockGame();
mockGame9.askIntReturn.add(2);
mockGame9.askIntReturn.add(4);
mockGame9.askIntReturn.add(4);
mockGame9.askIntReturn.add(5);
mockGame9.askIntReturn.add(5);
mockGame9.askIntReturn.add(4);
mockGame9.turnos=57;
mockGame9.start();
assertTrue(mockGame9.gameOver);
assertEquals(mockGame9.fichas,7);
assertEquals(mockGame9.turnos,60);
assertEquals(mockGame9.blancas,2);
assertEquals(mockGame9.win,5);

//4 Turns
MockGame mockGame0=new MockGame();
mockGame0.askIntReturn.add(2);
mockGame0.askIntReturn.add(4);
mockGame0.askIntReturn.add(4);
mockGame0.askIntReturn.add(5);
mockGame0.askIntReturn.add(5);
mockGame0.askIntReturn.add(4);
mockGame0.askIntReturn.add(2);
mockGame0.askIntReturn.add(3);
mockGame0.turnos=56;
mockGame0.start();
assertTrue(mockGame0.gameOver);
assertEquals(mockGame0.fichas,8);
assertEquals(mockGame0.turnos,60);
assertEquals(mockGame0.blancas,5);
assertEquals(mockGame0.win,3);
```

Caja blanca

Path Coverage

Comprobamos los diferentes caminos:

-Partida en la cual nadie puede tirar
-Partida donde ganan las negras
-Partida donde ganan blancas
-Partida donde empatan
-Situación donde solo blancas pueden tirar
-Situación donde solo negras pueden tirar

Loop Testing

-while(!gameOver);

Comprobamos el bucle simulando partidas que duran 1,2,3,4 turnos lo que hace que se entre en el bucle 0,1,2,3 respectivamente.

Automatization

En este método usamos la automatization para simular partidas. Para ello tambien nos ayudamos de un MockObject que se encarga de simular el input del usuario cada turno, en este caso haciendo tiradas aleatorias.

Gracias a ello, realizamos los siguientes test:

- Simular 100 partidas para encontrar posibles problemas o casos especiales no detectados
- Simular partidas hasta que ganen las negras o las blancas
- Simular partidas hasta que las partidas acaben en empate

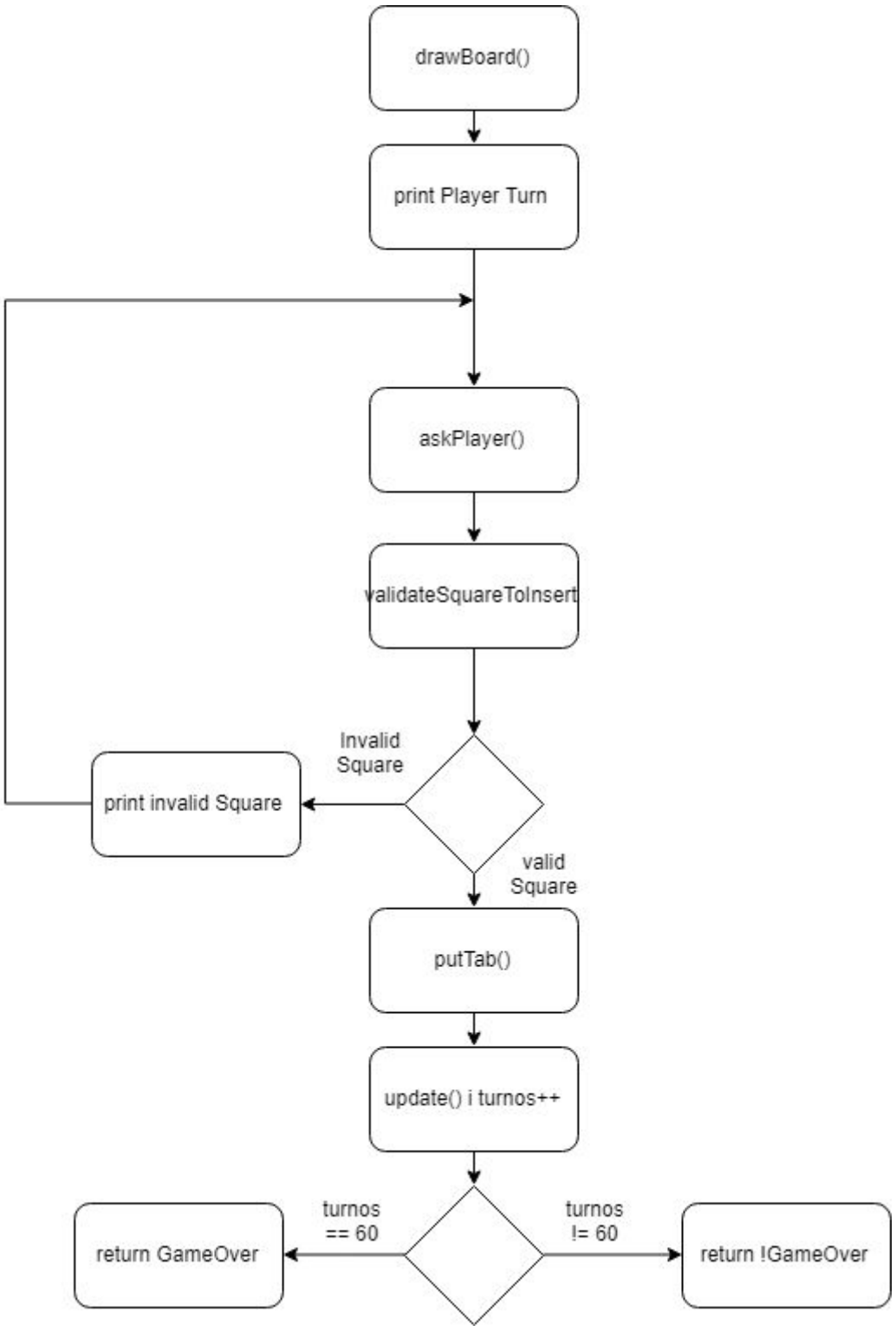
```
//AUTOMATIZATION TEST

//PLAY 100 RANDOM GAMES
MockGame2 testAuto=new MockGame2();
for(int i=0;i<100;i++){
    testAuto.start();
    testAuto.restart();
}

//PLAY RANDOM GAMES until DRAW
MockGame2 testAuto2=new MockGame2();
do{
     testAuto2.restart();
     testAuto2.start();
}while(!(testAuto2.blancas==testAuto2.win));

//PLAY RANDOM GAMES until BLACK WINS
MockGame2 testAuto3=new MockGame2();
do{
    testAuto3.restart();
    testAuto3.start();
}while(!(testAuto3.blancas<testAuto3.win));


//PLAY RANDOM GAMES until WHITE WINS
MockGame2 testAuto4=new MockGame2();
do{
    testAuto4.restart();
    testAuto4.start();
}while(!(testAuto4.blancas>testAuto4.win));
```

Turn()

Función que coloca ficha respecto a las entradas del cliente, volviendolas a pedir si estas no son correctas y actualiza el tablero respecto a la nueva ficha puesta cambiando de color las fichas flanqueadas del rival. Esta función devuelve false normalmente, solo devuelve true en el caso que la variable turnos sea 60 lo cual quiere decir que el tablero se a llenado.

```
public boolean turn(boolean turno){
    getBoard().drawBoard();
    boolean gameOver=false;
    if(turno){
    System.out.println("\n\n BLACK TURN ");
    }else{
    System.out.println("\n\n WHITE TURN ");
    }
    int coor[];

    do{
        coor=askPlayer();

        if(!table.validateSquareToInsert(coor[0], coor[1],turno)){
        System.out.println("Square already occupied or out of bounds");
        System.out.println("Insert valid coordinates");
        }
    }while(!table.validateSquareToInsert(coor[0], coor[1],turno));

    int x=coor[0];
    int y=coor[1];

    table.putTab(x,y,turno);
    table.update(x,y);
    turnos++;

    if(turnos==60){
        gameOver=true;
    }
    return gameOver;
}
```

Diagrama de flujo



Codigo de Test

```
@Test
public void testTurn() {

    //Initialize Set up Game
    MockGame mockGame=new MockGame();

    //Loop Testing

    //Ending turn by Black Tabs (out of movements)
    mockGame.askIntReturn.add(4);
    mockGame.askIntReturn.add(2);
    assertFalse(mockGame.turn(true));

    //Ending turn by White Tabs (out of movements)
    mockGame.askIntReturn.add(5);
    mockGame.askIntReturn.add(4);
    assertFalse(mockGame.turn(false));

    //Ending turn by Black Tabs Repeating 1 time the Insertion Loop
    mockGame.askIntReturn.add(0);
    mockGame.askIntReturn.add(0);
    mockGame.askIntReturn.add(4);
    mockGame.askIntReturn.add(5);
    assertFalse(mockGame.turn(true));

    //Ending turn by White Tabs Repeating 2 times the Insertion Loop
    mockGame.askIntReturn.add(0);
    mockGame.askIntReturn.add(0);
    mockGame.askIntReturn.add(0);
    mockGame.askIntReturn.add(0);
    mockGame.askIntReturn.add(3);
    mockGame.askIntReturn.add(2);
    assertFalse(mockGame.turn(false));

  //Ending turn by Black Tabs Repeating 3 times the Insertion Loop
  mockGame.askIntReturn.add(0);
  mockGame.askIntReturn.add(0);
  mockGame.askIntReturn.add(0);
  mockGame.askIntReturn.add(0);
  mockGame.askIntReturn.add(0);
  mockGame.askIntReturn.add(0);
  mockGame.askIntReturn.add(2);
  mockGame.askIntReturn.add(4);
  assertFalse(mockGame.turn(true));

  //Ending turn by White Tabs (out of turns)
  mockGame.turnos=59;
  mockGame.askIntReturn.add(4);
  mockGame.askIntReturn.add(1);
  assertTrue(mockGame.turn(false));

  //Ending turn by Black Tabs (out of turns)
  mockGame.turnos=59;
  mockGame.askIntReturn.add(3);
  mockGame.askIntReturn.add(1);
  assertTrue(mockGame.turn(true));
```

Caja blanca

Path Coverage

Comprobamos los diferentes caminos:

-Turno que no termina la partida de las negras
-Turno que no termina la partida de las blancas
-Turno que termina la partida de las negras
-Turno que termina la partida de las negras

Loop Testing

-while(!table.validateSquareToInsert(coor[0], coor[1],turno));

Comprobamos el bucle saltando directamente en tests del path coverage y luego fallando al introducir valores incorrectos 1,2,3 veces

AskPlayer()

Función que capta las dos entradas del cliente las cuales van a representar las coordenadas de X y Y donde se va a colocar la pieza y las pedira hasta que los valores entrados sean entre 0 y 8 con la función ValidCoor().

```
public int[] askPlayer(){
    Scanner scanner = new Scanner(System.in);
    int coordinates[]={-1,-1};
        do {
            System.out.println("\nEnter coordinate X");
            coordinates[1] = askInt();
        }while(!validCoor(coordinates[1]));

        do {
            System.out.println("\nEnter coordinate Y");
            coordinates[0] = askInt();
        }while(!validCoor(coordinates[0]));

    return coordinates;
}
```

Codigo de Test

```
public void testAskPlayer() {

    //Initialize Game
    MockGame mockGame=new MockGame();

    //Test First loop: Valid Option
    int[] res1 = {2, 1};
    mockGame.askIntReturn.clear();
    mockGame.askIntReturn.add(1);
    mockGame.askIntReturn.add(2);
    assertArrayEquals(mockGame.askPlayer(), res1);

    //Test First loop: Invalid Option x1
    int[] res2 = {7, 2};
    mockGame.askIntReturn.clear();
    mockGame.askIntReturn.add(15);
    mockGame.askIntReturn.add(2);
    mockGame.askIntReturn.add(7);
    assertArrayEquals(mockGame.askPlayer(), res2);

    //Test First loop: Invalid Option x2
    int[] res3 = {1, 1};
    mockGame.askIntReturn.clear();
    mockGame.askIntReturn.add(-2);
    mockGame.askIntReturn.add(23);
    mockGame.askIntReturn.add(1); // Valid Option
    mockGame.askIntReturn.add(1); // Valid Option
    assertArrayEquals(mockGame.askPlayer(), res3);

    //Test First loop: Invalid Option x5
    int[] res4 = {2, 1};
    mockGame.askIntReturn.clear();
    mockGame.askIntReturn.add(-28);
    mockGame.askIntReturn.add(10);
    mockGame.askIntReturn.add(25);
    mockGame.askIntReturn.add(100);
    mockGame.askIntReturn.add(-1);
    mockGame.askIntReturn.add(1); // Valid Option
    mockGame.askIntReturn.add(2); // Valid Option
    assertArrayEquals(mockGame.askPlayer(), res4);
```

```
//Test First loop: Invalid Option x10
Othello/src/Board.java
, 2};
mockGame.askIntReturn.clear();
mockGame.askIntReturn.add(-28);
mockGame.askIntReturn.add(10);
mockGame.askIntReturn.add(25);
mockGame.askIntReturn.add(100);
mockGame.askIntReturn.add(-1);
mockGame.askIntReturn.add(-28);
mockGame.askIntReturn.add(10);
mockGame.askIntReturn.add(25);
mockGame.askIntReturn.add(100);
mockGame.askIntReturn.add(-1);
mockGame.askIntReturn.add(2); // Valid Option
mockGame.askIntReturn.add(2); // Valid Option
assertArrayEquals(mockGame.askPlayer(), res5);



//Test Second loop: Valid Option
int[] res11 = {2, 1};
mockGame.askIntReturn.clear();
mockGame.askIntReturn.add(1);
mockGame.askIntReturn.add(2);
assertArrayEquals(mockGame.askPlayer(), res11);

//Test Second loop: Invalid Option x1
int[] res12 = {3, 2};
mockGame.askIntReturn.clear();
mockGame.askIntReturn.add(2);
mockGame.askIntReturn.add(15);
mockGame.askIntReturn.add(3);
assertArrayEquals(mockGame.askPlayer(), res12);

//Test Second loop: Invalid Option x2
int[] res13 = {1, 1};
mockGame.askIntReturn.clear();
mockGame.askIntReturn.add(1); // Valid Option
mockGame.askIntReturn.add(-2);
mockGame.askIntReturn.add(23);
mockGame.askIntReturn.add(1); // Valid Option
assertArrayEquals(mockGame.askPlayer(), res13);
```

```
//Test Second loop: Invalid Option x5
int[] res14 = {2, 1};
mockGame.askIntReturn.clear();
mockGame.askIntReturn.add(1); // Valid Option
mockGame.askIntReturn.add(-28);
mockGame.askIntReturn.add(10);
mockGame.askIntReturn.add(25);
mockGame.askIntReturn.add(100);
mockGame.askIntReturn.add(-1);
mockGame.askIntReturn.add(2); // Valid Option
assertArrayEquals(mockGame.askPlayer(), res14);

//Test Second loop: Invalid Option x10
int[] res15 = {2, 2};
mockGame.askIntReturn.clear();
mockGame.askIntReturn.add(2); // Valid Option
mockGame.askIntReturn.add(-28);
mockGame.askIntReturn.add(10);
mockGame.askIntReturn.add(25);
mockGame.askIntReturn.add(100);
mockGame.askIntReturn.add(-1);
mockGame.askIntReturn.add(-28);
mockGame.askIntReturn.add(10);
mockGame.askIntReturn.add(25);
mockGame.askIntReturn.add(100);
mockGame.askIntReturn.add(-1);
mockGame.askIntReturn.add(2); // Valid Option
assertArrayEquals(mockGame.askPlayer(), res15);
```

Caja blanca

Loop Testing

-while(!validCoor(coordinates[1]));

Comprobamos haciendo que se repita el bucle 0,1,2,5,10 veces.

-while(!validCoor(coordinates[0]));

Comprobamos haciendo que se repita el bucle 0,1,2,5,10 veces.

ValidCoor()

Devuelve true si el parametro entrado se encuentra entre 0 y 7, sino devuelve false.

```
public boolean validCoor(int input){
     if (input < 0 || input > 7)
     {
         return false;
     }
     return true;
}
```

Codigo de Test

```
@Test
public void testValidCoor() {

    //Initialize a game
    Game game = new Game();

    //Test checking if selected Square Coordinate is valid
    //Equivalent Partition -inf to -1 -> Invalid input
    boolean res_0 = game.validCoor(-29);
    assertFalse(res_0);

    boolean res_1 = game.validCoor(-1);
    assertFalse(res_1);

    //Equivalent Partition 0 to 7 -> valid input
    boolean res_2 = game.validCoor(0);
    assertTrue(res_2);

    boolean res_3 = game.validCoor(1);
    assertTrue(res_3);

    boolean res_4 = game.validCoor(2);
    assertTrue(res_4);

    boolean res_5 = game.validCoor(6);
    assertTrue(res_5);

    boolean res_6 = game.validCoor(7);
    assertTrue(res_6);

    //Equivalent Partition 8 to inf -> Invalid input
    boolean res_7 = game.validCoor(8);
    assertFalse(res_7);

    boolean res_8 = game.validCoor(19);
    assertFalse(res_8);
}
```

Caja negra

Comprobamos valores límite, frontera , valores dentro de las particiones y el 0.

Class Board:

Esta clase representa en si el tablero de juego así como sus cambios y las iteraciones del juego con el tablero proporcionando la información que necesita el juego para funcionar.

MockObjects

```
private class MockBoard1 extends Board{

    public boolean validateSquareToInsert(int x, int y,boolean turn){
        if(this.getSquare(x, y).getTab()){
            return false;
        }else{
            return true;
        }
    }

    public void cleanBoard(boolean color){
        for(int i=0;i<8;i++){
            for(int j=0;j<8;j++){
                this.getSquare(i, j).setTab(false);
                this.getSquare(i, j).setColor(color);
            }
        }
    }
```

Este mockObject de Board nos permite poder poner fichas donde queramos (siempre que no haya ya uno) para así poder generar facilmente escenarios de test y poder limpiar el tablero de fichas pero ponerle a esas casillas el color que se quiera.

```
private class MockBoard2 extends Board{

    private Square[][] table;

    public MockBoard2(){
        this.table=new Square[8][8];
        for(int i=0;i<8;i++){
            for(int j=0;j<8;j++){
                this.table[i][j]=new Square(true);
            }
        }
    }

    public boolean validateSquareToInsert(int x, int y,boolean turn){
        if(this.getSquare(x, y).getTab()){
            return false;
        }else{
            return true;
        }
    }

    public void restartBoard(){
        for(int i=0;i<8;i++){
            for(int j=0;j<8;j++){
                putTab(i,j,false);
                getSquare(i,j).setColor(false);
            }
        }
    }


    public void setCorners(){
        getSquare(0,0).setColor(true);
        getSquare(0,7).setColor(true);
        getSquare(7,0).setColor(true);
        getSquare(7,7).setColor(true);
    }

}
```

Este mockObject de Board nos permite poder poner fichas donde queramos (siempre que no haya ya uno) para así poder generar facilmente escenarios de test y además implementa algunas funciones para poner las fichas de las esquinas negras y poner fichas blancas en todo el tablero. Además al crear este mockObject el tablero estará totalmente vacío de fichas, no como el normal que se inicializa con 4 en el centro de ambos colores.

```
private class MockBoard3 extends Board{


        public void putTab(int x, int y,boolean turn){
            if(!this.getSquare(x, y).getTab()){
                getSquare(x,y).setColor(turn);
                getSquare(x,y).setTab(true);
            }
        }
    }
```

Este mockObject de Board nos permite poder poner fichas donde queramos (siempre que no haya ya uno) para así poder generar facilmente escenarios de test pero esta vez se modifica directamente el método que pone ficha y no la validación.

Update()

Método que eligiendo una casilla da la vuelta a la fichas flanqueadas por la casilla seleccionada respecto al color de la seleccionada.

```
public void update(int x, int y){

    if(!(x>7 || y>7 || x<0 || y<0 )){
        for(int i=-1;i<2;i++){
            for(int j=-1;j<2;j++){
                if(!(i==0 && j==0)){
                    if(!(getSquare(x+i,y+j).getColor()==getSquare(x,y).getColor()) && (getSquare(x+i,y+j).getTab())){
                        int z=i;
                        int t=j;
                        while(!(getSquare(x+z,y+t).getColor()==getSquare(x,y).getColor()) && (getSquare(x+z,y+t).getTab())){
                            z=z+i;
                            t=t+j;

                        }

                        if(getSquare(x+z,y+t).getColor()==getSquare(x,y).getColor() && (getSquare(x+z,y+t).getTab())){
                            z=z-i;
                            t=t-j;
                            while(!(z+x==x && t+y==y)){
                                changeSquare(z+x,t+y);
                                z=z-i;
                                t=t-j;
                            }

                        }
                    }
                }
            }
        }
    }
}
```

Codigo de Test

```
@Test
public void testUpdate() {

    //Black Box

    //Initialize Board
    MockBoard2 test=new MockBoard2();

    //Equivalent Partition Outside of the Board -> invalid

    test.restartBoard();
    test.update(8, 8);

    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            assertFalse(test.getSquare(i,j).getColor());
        }
    }

    test.restartBoard();
    test.update(-1, -1);

    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            assertFalse(test.getSquare(i,j).getColor());
        }
    }

    test.restartBoard();
    test.update(-1, 6);

    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            assertFalse(test.getSquare(i,j).getColor());
        }
    }
```

```
test.restartBoard();
test.update(5, -1);

for(int i=0;i<8;i++){
    for(int j=0;j<8;j++){
        assertFalse(test.getSquare(i,j).getColor());
    }
}

test.restartBoard();
test.update(9, -1);

for(int i=0;i<8;i++){
    for(int j=0;j<8;j++){
        assertFalse(test.getSquare(i,j).getColor());
    }
}

test.restartBoard();
test.update(-1, 9);

for(int i=0;i<8;i++){
    for(int j=0;j<8;j++){
        assertFalse(test.getSquare(i,j).getColor());
    }
}
```

```
//Equivalent Partition Inside Board, maximum flank length (8) -> valid

test.restartBoard();
test.setCorners();
test.update(0, 0);

for(int i=0;i<8;i++){
    assertTrue(test.getSquare(i,0).getColor());
    assertTrue(test.getSquare(0,i).getColor());
    assertTrue(test.getSquare(i,i).getColor());
}

test.restartBoard();
test.setCorners();
test.update(0, 7);

for(int i=0;i<8;i++){
    assertTrue(test.getSquare(i,7).getColor());
    assertTrue(test.getSquare(0,7-i).getColor());
    assertTrue(test.getSquare(0+i,7-i).getColor());
}

test.restartBoard();
test.setCorners();
test.update(7, 0);

for(int i=0;i<8;i++){

    assertTrue(test.getSquare(7-i,0).getColor());
    assertTrue(test.getSquare(7,i).getColor());
    assertTrue(test.getSquare(7-i,i).getColor());
}

test.restartBoard();
test.setCorners();
test.update(7, 7);
```

```
 for(int i=0;i<8;i++){

     assertTrue(test.getSquare(7-i,7).getColor());
     assertTrue(test.getSquare(7,7-i).getColor());
     assertTrue(test.getSquare(7-i,7-i).getColor());
 }

 //Equivalent Partition Inside Board, mid flank length (4) -> valid

 test.restartBoard();
 test.changeSquare(6, 6);
 test.changeSquare(1, 6);
 test.changeSquare(6, 1);
 test.changeSquare(1, 1);
 test.update(1, 1);

 for(int i=0;i<6;i++){
     assertTrue(test.getSquare(i+1,1).getColor());
     assertTrue(test.getSquare(1,i+1).getColor());
     assertTrue(test.getSquare(i+1,i+1).getColor());
 }

 test.restartBoard();
 test.changeSquare(6, 6);
 test.changeSquare(1, 6);
 test.changeSquare(6, 1);
 test.changeSquare(1, 1);
 test.update(1, 6);
```

```
    for(int i=0;i<6;i++){

        assertTrue(test.getSquare(i+1,6).getColor());
        assertTrue(test.getSquare(1,6-i).getColor());
        assertTrue(test.getSquare(1+i,6-i).getColor());

    }

    test.restartBoard();
    test.changeSquare(6, 6);
    test.changeSquare(1, 6);
    test.changeSquare(6, 1);
    test.changeSquare(1, 1);
    test.update(6, 1);


    for(int i=0;i<6;i++){

        assertTrue(test.getSquare(6-i,1).getColor());
        assertTrue(test.getSquare(6,i+1).getColor());
        assertTrue(test.getSquare(6-i,i+1).getColor());
    }

    test.restartBoard();
    test.changeSquare(6, 6);
    test.changeSquare(1, 6);
    test.changeSquare(6, 1);
    test.changeSquare(1, 1);
    test.update(6, 6);

    for(int i=0;i<6;i++){

        assertTrue(test.getSquare(6-i,6).getColor());
        assertTrue(test.getSquare(6,6-i).getColor());
        assertTrue(test.getSquare(6-i,6-i).getColor());
    }
```

```
//Equivalent Partition Inside Board, low flank length (2) -> valid

test.restartBoard();
test.changeSquare(5, 5);
test.changeSquare(2, 5);
test.changeSquare(5, 2);
test.changeSquare(2, 2);
test.update(2, 2);


for(int i=0;i<4;i++){
    assertTrue(test.getSquare(i+2,2).getColor());
    assertTrue(test.getSquare(2,i+2).getColor());
    assertTrue(test.getSquare(i+2,i+2).getColor());
}

test.restartBoard();
test.changeSquare(5, 5);
test.changeSquare(2, 5);
test.changeSquare(5, 2);
test.changeSquare(2, 2);
test.update(2, 5);


for(int i=0;i<4;i++){

    assertTrue(test.getSquare(i+2,5).getColor());
    assertTrue(test.getSquare(2,5-i).getColor());
    assertTrue(test.getSquare(2+i,5-i).getColor());

}

test.restartBoard();
test.changeSquare(5, 5);
test.changeSquare(2, 5);
test.changeSquare(5, 2);
test.changeSquare(2, 2);
test.update(5, 2);
```

```
  for(int i=0;i<4;i++){

      assertTrue(test.getSquare(5-i,2).getColor());
      assertTrue(test.getSquare(5,i+2).getColor());
      assertTrue(test.getSquare(5-i,i+2).getColor());
  }

  test.restartBoard();
  test.changeSquare(5, 5);
  test.changeSquare(2, 5);
  test.changeSquare(5, 2);
  test.changeSquare(2, 2);
  test.update(5, 5);

  for(int i=0;i<4;i++){

      assertTrue(test.getSquare(5-i,5).getColor());
      assertTrue(test.getSquare(5,5-i).getColor());
      assertTrue(test.getSquare(5-i,5-i).getColor());
  }

  //Equivalent Partition Inside Board, minimum flank length (0) -> valid

  test.restartBoard();
  test.changeSquare(4, 4);
  test.changeSquare(3, 4);
  test.changeSquare(4, 3);
  test.changeSquare(3, 3);
  test.update(3, 3);


  for(int i=0;i<2;i++){
      assertTrue(test.getSquare(i+3,3).getColor());
      assertTrue(test.getSquare(3,i+3).getColor());
      assertTrue(test.getSquare(i+3,i+3).getColor());
  }
```

```
    test.restartBoard();
    test.changeSquare(4, 4);
    test.changeSquare(3, 4);
    test.changeSquare(4, 3);
    test.changeSquare(3, 3);
    test.update(3, 4);


    for(int i=0;i<2;i++){

        assertTrue(test.getSquare(i+3,4).getColor());
        assertTrue(test.getSquare(3,4-i).getColor());
        assertTrue(test.getSquare(3+i,4-i).getColor());

    }

    test.restartBoard();
    test.changeSquare(4, 4);
    test.changeSquare(3, 4);
    test.changeSquare(4, 3);
    test.changeSquare(3, 3);
    test.update(4, 3);


    for(int i=0;i<2;i++){

        assertTrue(test.getSquare(4-i,3).getColor());
        assertTrue(test.getSquare(4,i+3).getColor());
        assertTrue(test.getSquare(4-i,i+3).getColor());
    }

    test.restartBoard();
    test.changeSquare(4, 4);
    test.changeSquare(3, 4);
    test.changeSquare(4, 3);
    test.changeSquare(3, 3);
    test.update(4, 4);

    for(int i=0;i<2;i++){

        assertTrue(test.getSquare(4-i,4).getColor());
        assertTrue(test.getSquare(4,4-i).getColor());
        assertTrue(test.getSquare(4-i,4-i).getColor());
    }
```

```
//Equivalent Partition Inside Board, maximum flank length-1 with same color tab at the end -> valid

MockBoard1 test2=new MockBoard1();
test2.cleanBoard(false);
for(int i=1;i<7;i++){
    for(int j=1;j<7;j++){
        test2.getSquare(i, j).setColor(true);
        test2.getSquare(i, j).setTab(true);
    }
}


for(int i=1;i<7;i++){
    for(int j=1;j<7;j++){
        test2.changeSquare(i, j);
        test2.update(i, j);
        for(int x=1;x<7;x++){
            for(int y=1;y<7;y++){

                if((i==x && j==y)){
                assertTrue(test2.getSquare(x,y).getTab());
                assertFalse(test2.getSquare(x,y).getColor());
                }else{
                assertTrue(test2.getSquare(x,y).getTab());
                assertTrue(test2.getSquare(x,y).getColor());
                }
            }
        }
        test2.changeSquare(i, j);
    }
}

//Equivalent Partition Inside Board, maximum flank length-1  -> valid


test2.cleanBoard(true);
for(int i=1;i<7;i++){
    for(int j=1;j<7;j++){
        test2.getSquare(i, j).setTab(true);
    }
}

for(int i=1;i<7;i++){
    for(int j=1;j<7;j++){
        test2.changeSquare(i, j);
        test2.update(i, j);
        for(int x=1;x<7;x++){
            for(int y=1;y<7;y++){

                if((i==x && j==y)){
                assertTrue(test2.getSquare(x,y).getTab());
                assertFalse(test2.getSquare(x,y).getColor());
                }else{
                assertTrue(test2.getSquare(x,y).getTab());
                assertTrue(test2.getSquare(x,y).getColor());
                }
            }
        }
        test2.changeSquare(i, j);
    }
}
```

Caja blanca

Path Coverage

Para conseguir esto tenemos que intentar hacer update tanto a casillas que que estén fuera del tablero y por lo cual comprobar que nada a cambiado en el tablero, tambien probar con casillas de dentro del tablero donde no pueda actualizar y otras donde actualice.

Loop Testing

-while(!(getSquare(x+z,y+t).getColor()==getSquare(x,y).getColor()) && (getSquare(x+z,y+t).getTab())){

Comprobamos este bucle en todas las direcciones por las que se va a actualizar haciendo que salga del bucle después de 0,2,4,6 y 8 veces y comprobamos que las fichas no han cambiado .

while(!(z+x==x && t+y==y)){

Comprobamos este bucle en todas las direcciones por las que se va a actualizar haciendo que salga del bucle después de 0,2,4,6 y 8 veces y comprobamos que las fichas esperadas han cambiado .

CalculateWhoWin()

Devuelve el número de fichas negras que hay en el tablero y para ello recorre todo el tablero.

```
public int calculateWhoWin(){

    int contadorNegras=0;
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(this.getSquare(i, j).getColor()){
                contadorNegras++;
            }
        }
    }

    return contadorNegras;
}
```

Codigo de Test

```
@Test
public void testCalculateWhoWin() {

    //Black Box
    //Initialize Board

    MockBoard2 test=new MockBoard2();
    test.restartBoard();
    int x=0;
    //Check every possible combination of Black-White Tabs
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            assertEquals(x,test.calculateWhoWin());
            test.changeSquare(i, j);
            x++;
        }
    }
    assertEquals(x,test.calculateWhoWin());
    assertEquals(64,test.calculateWhoWin());
}
```

Caja blanca

Path Coverage

Comprobamos los diferentes caminos:

Automatizamos para realizar la comprobación teniendo inicialmente todo el tablero con todo blancas y girando una por una las fichas y comprobando que los valores devueltos son correctos.

ValidateSquareToInsert()

Devuelve un true si la casilla seleccionada es correcta para colocar ficha del color pasado por parámetro. Para que sea correcta tiene que estar en el tablero , tiene que franquear alguna del otro color y tiene que tener una ficha ya colocada en esa casilla.

```
public boolean validateSquareToInsert(int x, int y,boolean turn){
    if(x>7 || y>7 || x<0 || y<0 ){
        return false;
    }

    if(getSquare(x,y).getTab()){
        return false;
    }

    for(int i=-1;i<2;i++){
        for(int j=-1;j<2;j++){
            if(!(i==0 && j==0)){
                if(!(getSquare(x+i,y+j).getColor()==turn) && (getSquare(x+i,y+j).getTab())){
                    int z=i;
                    int t=j;
                    while(!(getSquare(x+z,y+t).getColor()==turn) && (getSquare(x+z,y+t).getTab())){
                        z=z+i;
                        t=t+j;

                    }

                    if(getSquare(x+z,y+t).getColor()==turn && (getSquare(x+z,y+t).getTab())){

                        return true;
                    }

                }
            }
        }
    }
    return false;
}
```

Codigo de Test

```
@Test
public void testValidateSquareToInsert() {

    //Black Box

    //Equivalent Partition Insert Square next to opposite color tab, flanking -> valid
    //Equivalent Partition Insert Square next to opposite color tab, not flanking -> invalid
    //Equivalent Partition Insert Square next to same color tab, flanking -> valid
    //Equivalent Partition Insert Square next to same color tab, not flanking -> invalid
    //Equivalent Partition Insert Square next to no tab, not flanking -> invalid
    //Equivalent Partition Insert Square next to no color tab, not flanking -> invalid
    //Equivalent Partition Insert Square outside Board-> invalid

    //Initialize Board with Tabs in the center (initial setup) (2x2)
    MockBoard3 test=new MockBoard3();

    test.getSquare(3, 3).setColor(false);
    test.getSquare(4, 4).setColor(false);

    //Half-way game state, tabs in the center (4x4)
    for(int i=2;i<6;i++){
        for(int j=2;j<6;j++){
            test.putTab(i,j,true);
        }
    }

    //Check for true on the outside ring of the placed tabs and false on the rest, outside the board included.
    for(int i=-1;i<9;i++){
        for(int j=-1;j<9;j++){
            if((i==1 && j<=6 && j>=1) || (i==6 && j<=6 && j>=1)||(j==1 && i<6 && i>1) || (j==6 && i<6 && i>1)){
                assertTrue(test.validateSquareToInsert(i,j,false));
                assertFalse(test.validateSquareToInsert(i,j,true));
            }else{
                assertFalse(test.validateSquareToInsert(i,j,false));
                assertFalse(test.validateSquareToInsert(i,j,true));
            }
        }
    }

    //Very advanced game state, tabs in the center (6x6)
    for(int i=1;i<7;i++){
        for(int j=1;j<7;j++){
            test.putTab(i,j,false);
        }
    }

  //Check for true on the outside ring of the placed tabs and false on the rest, outside the board included.
  for(int i=-1;i<9;i++){
      for(int j=-1;j<9;j++){
          if((i==0 && j<=7 && j>=0) || (i==7 && j<=7 && j>=0)||(j==0 && i<7 && i>0) || (j==7 && i<7 && i>0)){
              assertFalse(test.validateSquareToInsert(i,j,false));
              assertTrue(test.validateSquareToInsert(i,j,true));
          }else{
              assertFalse(test.validateSquareToInsert(i,j,false));
              assertFalse(test.validateSquareToInsert(i,j,true));
          }
      }
  }



  //Finished game state, tabs in the center (8x8)
  for(int i=0;i<8;i++){
      for(int j=0;j<8;j++){
          test.putTab(i,j,true);
      }
  }

  //Check for true on the outside ring of the placed tabs and false on the rest, outside the board included.
  for(int i=-1;i<9;i++){
      for(int j=-1;j<9;j++){
          assertFalse(test.validateSquareToInsert(i,j,false));
      }
  }
```

Caja blanca

Path Coverage

Para ello tenemos que realizar este método sobre casillas que estén fuera del tablero, casillas que se encuentren dentro que ya tengan fichas, casillas de dentro que no flanqueen ninguna ficha del rival y casillas de dentro del tablero que flanqueen alguna ficha del rival.

Loop testing

while(!(getSquare(x+z,y+t).getColor()==turn)&&(getSquare(x+z,y+t).getTab()))

Para testear el loop realizamos este método en diferentes escenarios donde las casillas sobre las que se ejecuta el metodo no tienen ficha y están dentro del tablero pero estas flanquean fichas rivales y dependiendo del escenario flanquearan 2,4,6 fichas rivales entrando dichas veces en el bucle antes de salir.

ValidateSquareToChange()

Funcion que devuelve true si la casilla seleccionada es válida para cambiarla de color sino devuelve false. Para que eso sea posible la casilla seleccionada tiene que estar en el tablero y tener una ficha colocada, independientemente de su color.

```
public boolean validateSquareToChange(int x, int y){
    if(x>7 || y>7 || x<0 || y<0 ){
        return false;
    }
    return getSquare(x,y).getTab();
}

public int calculateWhoWin(){
```

Codigo de Test

```
@Test
public void testValidateSquareToInsert() {

    //Black Box

    //Equivalent Partition Insert Square next to opposite color tab, flanking -> valid
    //Equivalent Partition Insert Square next to opposite color tab, not flanking -> invalid
    //Equivalent Partition Insert Square next to same color tab, flanking -> valid
    //Equivalent Partition Insert Square next to same color tab, not flanking -> invalid
    //Equivalent Partition Insert Square next to no tab, not flanking -> invalid
    //Equivalent Partition Insert Square next to no color tab, not flanking -> invalid
    //Equivalent Partition Insert Square outside Board-> invalid

    //Initialize Board with Tabs in the center (initial setup) (2x2)
    MockBoard3 test=new MockBoard3();

    test.getSquare(3, 3).setColor(false);
    test.getSquare(4, 4).setColor(false);

    //Half-way game state, tabs in the center (4x4)
    for(int i=2;i<6;i++){
        for(int j=2;j<6;j++){
            test.putTab(i,j,true);
        }
    }

    //Check for true on the outside ring of the placed tabs and false on the rest, outside the board included.
    for(int i=-1;i<9;i++){
        for(int j=-1;j<9;j++){
            if((i==1 && j<=6 && j>=1) || (i==6 && j<=6 && j>=1)||(j==1 && i<6 && i>1) || (j==6 && i<6 && i>1)){
                assertTrue(test.validateSquareToInsert(i,j,false));
                assertFalse(test.validateSquareToInsert(i,j,true));
            }else{
                assertFalse(test.validateSquareToInsert(i,j,false));
                assertFalse(test.validateSquareToInsert(i,j,true));
            }
        }
    }


//Very advanced game state, tabs in the center (6x6)
for(int i=1;i<7;i++){
    for(int j=1;j<7;j++){
        test.putTab(i,j,false);
    }
}

//Check for true on the outside ring of the placed tabs and false on the rest, outside the board included.
for(int i=-1;i<9;i++){
    for(int j=-1;j<9;j++){
        if((i==0 && j<=7 && j>=0) || (i==7 && j<=7 && j>=0)||(j==0 && i<7 && i>0) || (j==7 && i<7 && i>0)){
            assertFalse(test.validateSquareToInsert(i,j,false));
            assertTrue(test.validateSquareToInsert(i,j,true));
        }else{
            assertFalse(test.validateSquareToInsert(i,j,false));
            assertFalse(test.validateSquareToInsert(i,j,true));
        }
    }
}



//Finished game state, tabs in the center (8x8)
for(int i=0;i<8;i++){
    for(int j=0;j<8;j++){
        test.putTab(i,j,true);
    }
}

 //Check for true on the outside ring of the placed tabs and false on the rest, outside the board included.
 for(int i=-1;i<9;i++){
     for(int j=-1;j<9;j++){
         assertFalse(test.validateSquareToInsert(i,j,false));
     }
 }
```

Caja blanca

Path Coverage

Comprobamos los diferentes caminos:

-Casillas fuera del tablero pasando por todas las combinaciones del if inicial

-Casillas de dentro del tablero.

ChangeSquare()

Si la casilla el valida para ser cambiada se le cambiara el color, se cambiará a negras si es blanca y viceversa.

```
public void changeSquare(int x, int y){
    if(validateSquareToChange(x,y)){
        if(getSquare(x,y).getColor()){
            getSquare(x,y).setColor(false);
        }else{
            getSquare(x,y).setColor(true);
        }
    }
}
```

Codigo de Test

```
@Test
public void testChangeSquare() {
    //Black Box
    //Initialize Board
    MockBoardl test=new MockBoardl();

    //Equivalent Partition Black to White -> valid
    //Equivalent Partition White to Black -> valid

    //Equivalent Partition Black to White on the edges -> valid
    //Equivalent Partition White to Black on the edges -> valid

    //Equivalent Partition Black to White on the edges outside -> invalid
    //Equivalent Partition White to Black on the edges outside -> invalid

    //Equivalent Partition Black to White outside -> invalid
    //Equivalent Partition White to Black outside -> invalid

    //Change initial Tabs to color black
    test.changeSquare(4,3);
    test.changeSquare(3,4);

    //Check that all tabs are black
    for(int i=-1;i<9;i++){
        for(int j=-1;j<9;j++){
            if(i>=3 && i<5 && j>=3 && j<5){
                assertTrue(test.getSquare(i,j).getColor());
            }else{
                test.changeSquare(i,j);
                assertFalse(test.getSquare(i,j).getColor());
            }
        }
    }
```

```
//Place Black Tabs in the center of the Board (4x4)
//Change to White Tabs and check color and placement
for(int i=-1;i<9;i++){
    for(int j=-1;j<9;j++){
        if(i>=2 && i<6 && j>=2 && j<6){
            test.putTab(i,j,true);
            test.changeSquare(i,j);
            assertFalse(test.getSquare(i,j).getColor());
        }else{
            test.changeSquare(i,j);
            assertFalse(test.getSquare(i,j).getColor());
        }
    }
}

//Place White Tabs in the center of the Board (6x6)
//Change to Black Tabs and check color and placement
for(int i=-1;i<9;i++){
    for(int j=-1;j<9;j++){
        if(i>=1 && i<7 && j>=1 && j<7){
            test.putTab(i,j,false);
            test.changeSquare(i,j);
            assertTrue(test.getSquare(i,j).getColor());
        }else{
            test.changeSquare(i,j);
            assertFalse(test.getSquare(i,j).getColor());
        }
    }
}

//Place Black Tabs in the center of the Board (whole board 8x8)
//Change to White Tabs and check color and placement
for(int i=-1;i<9;i++){
    for(int j=-1;j<9;j++){
        if(i>=0 && i<8 && j>=0 && j<8){
            test.putTab(i,j,true);
            test.changeSquare(i,j);
            assertFalse(test.getSquare(i,j).getColor());
        }else{
            test.changeSquare(i,j);
            assertFalse(test.getSquare(i,j).getColor());
        }
    }
}
```

Caja blanca

Path Coverage

Comprobamos los diferentes caminos:

- -Casilla válida negra
- -Casilla válida blanca
- -Casilla no válida

CountPosiblesSquares()

Devuelve el número de casillas donde es posible colocar una ficha del color pasado por parámetro.

```
public int countPosibleSquares(boolean turn){
    int count=0;
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(this.validateSquareToInsert(i, j, turn)){
                count++;
            }
        }
    }
    return count;
}
```

Codigo de Test

```
@Test
public void testCountPosibleSquares() {

    //Black Box

    //Initialize Board
    Board test=new Board();

    //Box of 2x2 Black Tabs -> 0 White/Black possible movements
    test.changeSquare(3, 4);
    test.changeSquare(4, 3);
    assertEquals(0,test.countPosibleSquares(false));
    assertEquals(0,test.countPosibleSquares(true));

    //Box of 2x2 Black Tabs with a ring around of White Tabs-> 20 Black possible movements, 0 White
    for(int i=2;i<6;i++){
        for(int j=2;j<6;j++){
            if((i==2 && j<=5 && j>=2) || (i==5 && j<=5 && j>=2)||(j==2 && i<5 && i>2) || (j==5 && i<5 && i>2)){
                test.getSquare(i, j).setTab(true);
                test.getSquare(i, j).setColor(false);
            }
        }
    }

    assertEquals(0,test.countPosibleSquares(false));
    assertEquals(20,test.countPosibleSquares(true));

    //Box of 2x2 Black Tabs with a ring around of White Tabs and another ring of Black Tabs
    //->    28 Black possible movements, 0 White
    for(int i=1;i<7;i++){
        for(int j=1;j<7;j++){
            if((i==1 && j<=6 && j>=1) || (i==6 && j<=6 && j>=1)||(j==1 && i<6 && i>1) || (j==6 && i<6 && i>1)){
                test.getSquare(i, j).setTab(true);
                test.getSquare(i, j).setColor(true);
            }
        }
    }

    assertEquals(28,test.countPosibleSquares(false));
    assertEquals(0,test.countPosibleSquares(true));
```

```
//Box of 2x2 Black Tabs with a ring around of White Tabs and another ring of Black Tabs and another ring of White Tabs
//->  0 White/Black possible movements

for(int i=0;i<8;i++){
    for(int j=0;j<8;j++){
        if((i==0 && j<=7 && j>=0) || (i==7 && j<=7 && j>=0)||(j==0 && i<7 && i>0) || (j==7 && i<7 && i>0)){
            test.getSquare(i, j).setTab(true);
            test.getSquare(i, j).setColor(false);
        }
    }
}

assertEquals(0,test.countPosibleSquares(false));
assertEquals(0,test.countPosibleSquares(true));
```

Caja negra

Pasamos por diferentes escenarios comprobando si el número de posibles casillas es igual que el esperado, tanto con blancas y con negras y teniendo más y menos fichas incluyendo el caso en que ni blancas y negras tengas casillas posibles

Caja blanca

Path Coverage

Comprobamos los diferentes caminos:

-Con casillas validas en el tablero

-Con casillas no validas

PutTab()

Si la casilla seleccionada es válida para insertar una ficha del color pasado a esta función por parametro se colocará una ficha de dicho color en la casilla seleccionada.

```
public void putTab(int x, int y, boolean color){
    if(validateSquareToInsert(x,y,color)){
    getSquare(x,y).setColor(color);
    getSquare(x,y).setTab(true);
    }
}
```

Codigo de Test

```
@Test
public void testPutTab() {
    //Black Box

    //Equivalent Partition Put Tab in center (initial setup) (2x2) -> valid
    //Equivalent Partition Put Tab in whole board -> valid
    //Equivalent Partition Put Tab in the inner edge of the board -> valid
    //Equivalent Partition Put Tab on the edges outside -> invalid
    //Equivalent Partition Put Tab outside of the board -> invalid

    //Initialize Board with tabs in the center (4x4)
    MockBoard1 test=new MockBoard1();
    for(int i=2;i<6;i++){
        for(int j=2;j<6;j++){
            test.putTab(i,j,true);
        }
    }
    //Check Correct placement
    for(int i=-1;i<9;i++){
        for(int j=-1;j<9;j++){
            if(i>=2 && i<6 && j>=2 && j<6){
                assertTrue(test.getSquare(i,j).getTab());
            }else{
                assertFalse(test.getSquare(i,j).getTab());
            }
        }
    }

    //Initialize Board with tabs in the center (6x6)
    for(int i=1;i<7;i++){
        for(int j=1;j<7;j++){
            test.putTab(i,j,true);
        }
    }
```

```
//Check Correct placement
for(int i=-1;i<9;i++){
    for(int j=-1;j<9;j++){
        if(i>=1 && i<7 && j>=1 && j<7){
            assertTrue(test.getSquare(i,j).getTab());
        }else{
            assertFalse(test.getSquare(i,j).getTab());
        }
    }
}

//Initialize Board with tabs in the whole Board (8x8)
for(int i=0;i<8;i++){
    for(int j=0;j<8;j++){
        test.putTab(i,j, true);
    }
}
//Check Correct placement
for(int i=-1;i<9;i++){
    for(int j=-1;j<9;j++){
        if(i>=0 && i<8 && j>=0 && j<8){
            assertTrue(test.getSquare(i,j).getTab());
        }else{

            assertFalse(test.getSquare(i,j).getTab());
        }
    }
}
```

Caja blanca

Path Coverage

Comprobamos los diferentes caminos:

- -Colocar ficha blanca en una casilla válida
- -Colocar ficha negra en una casilla válida
- -Colocar ficha blanca en una casilla no válida
- -Colocar ficha negra en una casilla no válida