

HUNDIR LA FLOTA

Introducción:

En este proyecto, se implementará un programa para crear el juego hundir la flota, usando la metodología TDD para el desarrollo del proyecto, que consiste en crear casos de tests de caja negra antes de crear cualquier función, intentando detectar posibles errores que nos pudieran surgir a lo largo del desarrollo del código, de manera que podamos corregirlos rápidamente y con facilidad. Una vez creado el código, implementaremos casos de tests de caja blanca para asegurarnos que el algoritmo funciona correctamente.

Tests de caja negra:

Funcionalidad: Introducción de coordenadas por teclado.

Localización: Archivo Directa.java, clase Directa, método setCoordenada().

Test: Archivo DirectaTest.java, clase DirectaTest, método setCoordenada(). En este test se han usado las técnicas: partición equivalente, valores límite y frontera y mockobject (MockTeclado()).

Funcionalidad: Introducción de la dirección hacia la que deseas poner el barco por teclado.

Localización: Archivo Directa.java, clase Directa, método setDireccion().

Test: Archivo DirectaTest.java, clase DirectaTest, método setDireccion(). En este test se han usado las técnicas: partición equivalente, valores límite y frontera y mockobject (MockTeclado()).

Funcionalidad: Introducción de la longitud del barco que deseas poner por teclado.

Localización: Archivo Directa.java, clase Directa, método setLongitud().

Test: Archivo DirectaTest.java, clase DirectaTest, método setLongitud(). En este test se han usado las técnicas: partición equivalente, valores límite y frontera y mockobject (MockTeclado()).

Funcionalidad: Constructor del tablero.

Localización: Archivo Tablero.java, clase Tablero, método Tablero().

Test: Archivo TableroTest.java, clase TableroTest, método constructor(). En este test se ha usado la técnica: partición equivalente. No he usado más porque las casillas no pueden tener otro valor que no sea el 0 a la hora de construir el tablero.

Funcionalidad: Asignar un valor a una casilla en concreto.

Localización: Archivo Tablero.java, clase Tablero, método setCasilla().

Test: Archivo TableroTest.java, clase TableroTest, método setCasilla(). En este test se han usado las técnicas: partición equivalente y valores límite y frontera.

Funcionalidad: Inicializar el array de direcciones a true.

Localización: Archivo Jugador.java, clase Jugador, método inicializarDirecciones().

Test: Archivo JugadorTest.java, clase JugadorTest, método inicializarDirecciones(). En este test se ha usado la técnica: partición equivalente.

Funcionalidad: Comprobación de que las direcciones se inician correctamente comprobando los límites del tablero y las longitudes de los barcos.

Localización: Archivo Jugador.java, clase Jugador, método comprobarDireccion().

Test: Archivo JugadorTest.java, clase JugadorTest, método comprobarDireccion(). En este test se han usado las técnicas: valores límite, frontera y partición equivalente.

Funcionalidad: Comprobación de que hayan o no obstáculos en el camino de las posibles posiciones hacia donde se pueden poner los barcos.

Localización: Archivo Jugador.java, clase Jugador, método comprobarObstaculos().

Test: Archivo JugadorTest.java, clase JugadorTest, método comprobarObstaculos(). En este test se han usado las técnicas: valores límite, frontera y partición equivalente y mockobject (MockTablero()).

Funcionalidad: implementar la funcionalidad de acción de disparo de un jugador hacia otro.

Localización: Archivo Jugador.java, clase Jugador, método disparar().

Test: Archivo JugadorTest.java, clase JugadorTest, método disparar(). En este test se han usado las técnicas: valores límite, frontera y partición equivalente, y mockobjects (MockJugador(), MockTabero() y MockTeclado()).

Funcionalidad: función que se encarga de poner un barco en una posición y hacia una dirección.

Localización: Archivo Jugador.java, clase Jugador, método ponerBarco().

Test: Archivo JugadorTest.java, clase JugadorTest, método ponerBarco(). En este test se han usado las técnicas: valores límite, frontera y partición equivalente, y mockobjects (MockJugador(), MockTabero() y MockTeclado()).

Funcionalidad: función que se encarga de ir poniendo los barcos que tiene el jugador sin poner.

Localización: Archivo Jugador.java, clase Jugador, método ponerBarcos().

Test: Archivo JugadorTest.java, clase JugadorTest, método ponerBarcos(). En este test se han usado las técnicas: valores límite, frontera y partición equivalente, y mockobjects (MockJugador(), MockTabero() y MockTeclado()).

Funcionalidad: función que se encarga de comprobar si quedan barcos en el tablero del jugador para saber si se ha acabado el juego o no.

Localización: Archivo Jugador.java, clase Jugador, método comprobarDerrota().





















Test: Archivo JugadorTest.java, clase JugadorTest, método comprobarDerrota(). En este test se han usado las técnicas: valores límite, frontera y partición equivalente, y mockobjects (MockJugador() y MockTeclado()).

Tests de caja blanca:

Funcionalidad: Algoritmo del juego.

Localización: Archivo Juego.java, clase Juego, método main().

Test: Archivo JuegoTest.java, clase JuegoTest, método Juego(). En este test se ha usado el mockobject MockTeclado() para simular una partida aprovechando para implementar los Statement, Decision y Condition coverage.

Element		Coverage	Covered Ins...	Missed Instr...	Total Instruc...
▼  hundirlaflota	<div><div></div></div>	23,8 %	1.724	5.533	7.257
▼  src/test/java	<div><div></div></div>	6,5 %	374	5.422	5.796
▼  test	<div><div></div></div>	6,5 %	374	5.422	5.796
>  JugadorTest.java	<div><div></div></div>	0,0 %	0	3.952	3.952
>  DirectaTest.java	<div><div></div></div>	0,0 %	0	1.136	1.136
>  JugadorTestCajaBlanca	<div><div></div></div>	0,0 %	0	143	143
>  TableroTest.java	<div><div></div></div>	0,0 %	0	126	126
>  MockTablero.java	<div><div></div></div>	0,0 %	0	38	38
>  MockJugador.java	<div><div></div></div>	0,0 %	0	27	27
>  JuegoTest.java	<div><div></div></div>	100,0 %	357	0	357
>  MockTeclado.java	<div><div></div></div>	100,0 %	17	0	17
▼  src/main/java	<div><div></div></div>	92,4 %	1.350	111	1.461
▼  hundirlaflota	<div><div></div></div>	92,4 %	1.350	111	1.461
>  Juego.java	<div><div></div></div>	0,0 %	0	91	91
>  Teclado.java	<div><div></div></div>	0,0 %	0	20	20
>  Barco.java	<div><div></div></div>	100,0 %	27	0	27
>  CoordenadaBarco.java	<div><div></div></div>	100,0 %	15	0	15
>  Directa.java	<div><div></div></div>	100,0 %	167	0	167
>  Jugador.java	<div><div></div></div>	100,0 %	1.089	0	1.089
>  Tablero.java	<div><div></div></div>	100,0 %	52	0	52

Aunque en el main salga que solo 92%, eso es porque he usado el JuegoTest y un MocTeclado para simular el juego. Podemos ver que la clase JuegoTest y MockTeclado están al 100% de coverage, así que sumando todo sale un 100% de Statement coverage de todo el algoritmo.

Para los Decision y condition:

```

140
141     int i = 0;
142
143     All 4 branches covered (i < 4 && !puedeponerse) {
144
145         if(this.direcciones[i]) { //comprobamos si hay alguna direccion posible para poner el barco
146             puedeponerse = true;
147         }
148
149         i++;
150     }
151
152     if(!puedeponerse) { //si no puede ponerse el barco en esa posición (contando posición y obstaculos) volvemos a pedir la coordenada
153         System.out.print("No puede ponerse este barco en estas coordenadas."
154             + " Vuelve a introducir la coordenada: \n");
155     }
156
157 }
158
159 //cuando tenemos una coordenada donde podemos poner el barco...
160 ArrayList<Integer> diraux = new ArrayList<Integer>();
161
162 for(int i = 0; i < direcciones.length; i++) { //guardamos en un array los índices de las direcciones correctas
163     if(direcciones[i]) {
164         diraux.add(i);
165     }
166 }
167

```

En todas las condiciones me sale que están comprobados todos los casos. Solo hay dos excepciones para esto, que son:

```

for(int i = 0; i < diraux.size(); i++) { //imprimimos la lista de direcciones correctas
    switch(diraux.get(i)) {

        case 0:
            System.out.print(i+1 + ": Arriba\n");
            break;

        case 1:
            System.out.print(i+1 + ": Derecha\n");
            break;

        case 2:
            System.out.print(i+1 + ": Abajo\n");
            break;

        case 3:
            System.out.print(i+1 + ": Izquierda\n");
            break;

    }
}

```

Los switch, ya que no entra en la opción default, ya que no existe porque controlo que los valores de los switch siempre sean uno de los case, así que no entra jamás en el default,

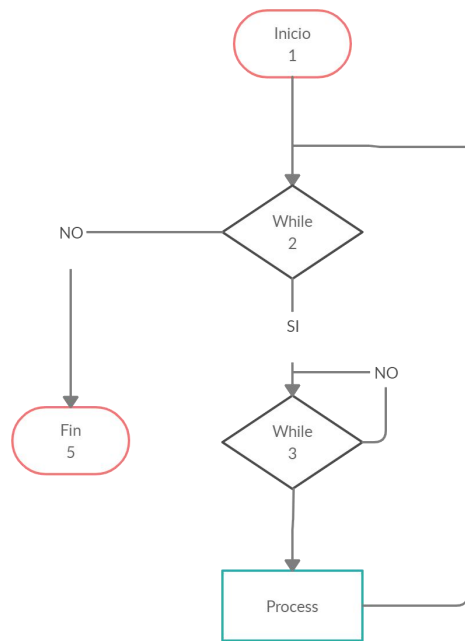
```

if(dir >= 1 && dir <= maximo) {
    direccion = dir;
} else {
    direccion = -1;
}

```

Y estos ifs, ya que es imposible matemáticamente que ambos sean false.

Para el path coverage, lo haremos sobre la función ponerBarcos(). El path es el siguiente



Así que los paths serán:

- 1- 1-2-5
- 2- 1-2-3-4-5
- 3- 1-2-3-3-4-5

Hay un while entre el 2 y el 3 que no he tenido en cuenta, porque si entra en el while 2, entrará seguro en el while que he ignorado, así que no he creído necesario introducirlo en el path para tomas de decisión.