

Gerard Asbert, 1603295

Albert Ceballos, 1604259

Horari de pràctiques: Dimecres, 10:30

SnakeGame

Github - <https://github.com/albertceballos0/SnakeGame>

Índice

1. Introducción	1
2. Funcionalidades del juego	2
SnakeBoardTest	2
SnakeFoodTest	4
SnakeObstacleTest	5
SnakePlayerTest	6
SnakeControllerTest	7
SnakeGameTest	9
3. Tests de caja blanca	11
3.1 Statement Coverage	11
3.2 Decision Coverage	11
3.3 Condition Coverage	12
3.4 Loop Testing	13
3.5 Path Coverage	13
4. Design by Contract	15
5. Mock Objects	15
6. CI/CD	16

1. Introducción

- Nuestro proyecto se basa en el famoso juego de Snake. Nos pareció que es un juego simple y a la vez complejo de programar, ya que se tiene que tener en cuenta muchas posibilidades, hay el factor random en el juego (en este caso el posicionamiento de la comida en cada turno), y creimos que sería divertido idear una forma diseñar la serpiente con listas de java.

Igualmente, después de programar las funcionalidades básicas, pensamos que no era suficiente, y que estaría bien añadir algunas funcionalidades extra para que el proyecto quede más completo. Al final nos decidimos por añadir-le obstaculos al juego, con los que si la serpiente toca uno de ellos, se acaba la partida.

Aunque no hayamos hecho un proyecto con java en el pasado, hemos decidido hacerlo en este lenguaje, ya que pensamos que es la oportunidad perfecta para aprender lo básico, y que nos sirva como proyecto introductorio al lenguaje.

2. Funcionalidades del juego

SnakeBoardTest

Funcionalidad: Inicialización del tablero de juego

- **Localización:** Archivo `Models/SnakeBoard` Clase `SnakeBoard`, método constructor (con parámetros `width` y `height`).
- **Test:** Archivo `Tests/SnakeBoardTest`, Clase `SnakeBoardTest`, método `testInitialization`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Particiones equivalentes y pruebas de valores límite (e.g., dimensiones iniciales y valores predeterminados).
- **Descripción:** Verifica que el tablero se inicialice correctamente con las dimensiones y valores esperados (ancho, alto, obstáculos y posición inicial de la comida).

Funcionalidad: Establecer y obtener la posición de la comida

- **Localización:** Archivo `Models/SnakeBoard` Clase `SnakeBoard`, métodos `setFood` y `getFood`.
- **Test:** Archivo `Tests/SnakeBoardTest`, Clase `SnakeBoardTest`, método `testSetAndGetFood`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Particiones equivalentes y verificación funcional.
- **Descripción:** Valida que la comida se pueda colocar en posiciones válidas y que estas se recuperen correctamente.

Funcionalidad: Añadir y obtener obstáculos

- **Localización:** Archivo `Models/SnakeBoard` Clase `SnakeBoard`, métodos `setObstacle` y `getObstacles`.
- **Test:** Archivo `Tests/SnakeBoardTest`, Clase `SnakeBoardTest`, método `testSetAndGetObstacles`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Particiones equivalentes y casos funcionales.
- **Descripción:** Comprueba que los obstáculos se añadan correctamente y puedan recuperarse con sus posiciones exactas.

Funcionalidad: Comprobar la presencia de comida en una posición

- **Localización:** Archivo `Models/SnakeBoard` Clase `SnakeBoard`, método `isFood`.
- **Test:** Archivo `Tests/SnakeBoardTest`, Clase `SnakeBoardTest`, método `testIsFood`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Casos positivos y negativos.
- **Descripción:** Valida si el tablero detecta correctamente la presencia de comida en una posición específica.

Funcionalidad: Comprobar la presencia de obstáculos en una posición

- **Localización:** Archivo `Models/SnakeBoard` Clase `SnakeBoard`, método `isObstacle`.
- **Test:** Archivo `Tests/SnakeBoardTest`, Clase `SnakeBoardTest`, método `testIsObstacle`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Casos positivos y negativos.
- **Descripción:** Verifica si el tablero detecta correctamente la presencia de obstáculos en una posición específica.

Funcionalidad: Limpiar todos los obstáculos del tablero

- **Localización:** Archivo `Models/SnakeBoard` Clase `SnakeBoard`, método `clearObstacles`.
- **Test:** Archivo `Tests/SnakeBoardTest`, Clase `SnakeBoardTest`, método `testClearObstacles`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Particiones equivalentes.
- **Descripción:** Comprueba que todos los obstáculos se eliminen correctamente y el contador se restablezca a 0.

Funcionalidad: Manejo de valores límite y excepciones

- **Localización:** Archivo `Models/SnakeBoard` Clase `SnakeBoard`, métodos `constructor`, `setFood`.
- **Test:** Archivo `Tests/SnakeBoardTest`, Clase `SnakeBoardTest`, método `testValorsFrontera`.
- **Tipo de test:** Caja blanca y negra.
- **Técnicas utilizadas:** Pruebas de valores límite y excepciones.
- **Descripción:**
 - Manejo de dimensiones inválidas (e.g., negativas, igual a 0 o mayores a 500).
 - Colocación de comida en posiciones inválidas.
 - Asegura que las dimensiones válidas no arrojan excepciones y funcionan correctamente.

SnakeFoodTest

Funcionalidad: Obtener la coordenada X de la comida

- **Localización:** Archivo `Models/SnakeFood`, Clase `SnakeFood`, método `getX`.
- **Test:** Archivo `Tests/SnakeFoodTest`, Clase `SnakeFoodTest`, método `testGetX`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Comprueba que el método `getX` retorna correctamente la coordenada X inicializada al crear el objeto `SnakeFood`.

Funcionalidad: Obtener la coordenada Y de la comida

- **Localización:** Archivo `Models/SnakeFood`, Clase `SnakeFood`, método `getY`.
- **Test:** Archivo `Tests/SnakeFoodTest`, Clase `SnakeFoodTest`, método `testGetY`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Comprueba que el método `getY` retorna correctamente la coordenada Y inicializada al crear el objeto `SnakeFood`.

Funcionalidad: Inicialización de un objeto `SnakeFood`

- **Localización:** Archivo `Models/SnakeFood`, Clase `SnakeFood`, constructor.
- **Test:** Archivo `Tests/SnakeFoodTest`, Clase `SnakeFoodTest`, método `testInitialization`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Valida que las coordenadas iniciales (X e Y) se asignen correctamente al crear un objeto `SnakeFood`.

Funcionalidad: Manejo de valores frontera en las coordenadas

- **Localización:** Archivo `Models/SnakeFood`, Clase `SnakeFood`, constructor.
- **Test:** Archivo `Tests/SnakeFoodTest`, Clase `SnakeFoodTest`, método `testValorsFrontera`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Pruebas de valores límite, manejo de excepciones.
- **Descripción:** Verifica que el constructor:
 - Lance una excepción cuando las coordenadas son negativas (X o Y).
 - Acepte coordenadas válidas, incluyendo casos frontera como 0, sin generar errores.

SnakeObstacleTest

Funcionalidad: Obtener la coordenada X de un obstáculo

- **Localización:** Archivo `Models/SnakeObstacle`, Clase `SnakeObstacle`, método `getX`.
- **Test:** Archivo `Tests/SnakeObstacleTest`, Clase `SnakeObstacleTest`, método `testGetX`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Comprueba que el método `getX` retorna correctamente la coordenada X inicializada al crear el objeto `SnakeObstacle`.

Funcionalidad: Obtener la coordenada Y de un obstáculo

- **Localización:** Archivo `Models/SnakeObstacle`, Clase `SnakeObstacle`, método `getY`.
- **Test:** Archivo `Tests/SnakeObstacleTest`, Clase `SnakeObstacleTest`, método `testGetY`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Comprueba que el método `getY` retorna correctamente la coordenada Y inicializada al crear el objeto `SnakeObstacle`.

Funcionalidad: Inicialización de un objeto `SnakeObstacle`

- **Localización:** Archivo `Models/SnakeObstacle`, Clase `SnakeObstacle`, constructor.
- **Test:** Archivo `Tests/SnakeObstacleTest`, Clase `SnakeObstacleTest`, método `testInitialization`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Valida que las coordenadas iniciales (X e Y) se asignen correctamente al crear un objeto `SnakeObstacle`.

Funcionalidad: Manejo de valores frontera en las coordenadas

- **Localización:** Archivo `Models/SnakeObstacle`, Clase `SnakeObstacle`, constructor.
- **Test:** Archivo `Tests/SnakeObstacleTest`, Clase `SnakeObstacleTest`, método `testValorsFrontera`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Pruebas de valores límite, manejo de excepciones.
- **Descripción:** Verifica que el constructor lance excepciones cuando se pasan valores negativos y que acepte valores válidos, incluyendo el caso de coordenadas 0.

SnakePlayerTest

Funcionalidad: Inicialización de un objeto `SnakePlayer`

- **Localización:** Archivo `Models/SnakePlayer`, Clase `SnakePlayer`, constructor.
- **Test:** Archivo `Tests/SnakePlayerTest`, Clase `SnakePlayerTest`, método `testInitialization`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Valida que un objeto `SnakePlayer` se inicialice correctamente con las coordenadas especificadas (X, Y) y con el tamaño del cuerpo igual a 1.

Funcionalidad: Movimiento de la serpiente

- **Localización:** Archivo `Models/SnakePlayer`, Clase `SnakePlayer`, método `move`.
- **Test:** Archivo `Tests/SnakePlayerTest`, Clase `SnakePlayerTest`, método `testMove`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Verifica que el método `move` actualice correctamente las coordenadas del jugador (x y y) según la dirección establecida (derecha, arriba, izquierda, abajo).

Funcionalidad: Cambio de dirección de la serpiente

- **Localización:** Archivo `Models/SnakePlayer`, Clase `SnakePlayer`, método `setDirection`.
- **Test:** Archivo `Tests/SnakePlayerTest`, Clase `SnakePlayerTest`, método `testChangeDirection`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Asegura que la serpiente no pueda moverse en la dirección opuesta a su dirección actual, y que el cambio de dirección se refleje correctamente.

Funcionalidad: Crecimiento de la serpiente

- **Localización:** Archivo `Models/SnakePlayer`, Clase `SnakePlayer`, método `grow`.
- **Test:** Archivo `Tests/SnakePlayerTest`, Clase `SnakePlayerTest`, método `testGrow`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Verifica que la serpiente crezca correctamente, agregando un segmento adicional al cuerpo de la serpiente en la última posición.

Funcionalidad: Colisiones de la serpiente

- **Localización:** Archivo `Models/SnakePlayer`, Clase `SnakePlayer`, métodos `collidesWithWall` y `collidesWithSelf`.
- **Test:** Archivo `Tests/SnakePlayerTest`, Clase `SnakePlayerTest`, método `testCollisions`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Verifica que la serpiente detecte colisiones con las paredes del tablero (usando un objeto `SnakeBoardMock` de `mock_classes/SnakeBoardMock`) y con su propio cuerpo después de moverse y crecer.

Funcionalidad: Verificación de la ocupación de las coordenadas por la serpiente

- **Localización:** Archivo `Models/SnakePlayer`, Clase `SnakePlayer`, método `occupies`.
- **Test:** Archivo `Tests/SnakePlayerTest`, Clase `SnakePlayerTest`, método `testOccupies`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional (partición equivalente).
- **Descripción:** Comprueba si la serpiente ocupa una posición dada en el tablero antes y después de moverse.

Funcionalidad: Manejo de valores frontera en las coordenadas y dirección

- **Localización:** Archivo `Models/SnakePlayer`, Clase `SnakePlayer`, constructor y método `setDirection`.
- **Test:** Archivo `Tests/SnakePlayerTest`, Clase `SnakePlayerTest`, método `testValorsFrontera`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Pruebas de valores límite, manejo de excepciones.
- **Descripción:** Verifica que se lance una excepción si se pasan valores negativos al inicializar las coordenadas, y que acepte valores válidos. También prueba que se lance una excepción al establecer una dirección fuera del rango permitido (de -1 a 3).

SnakeControllerTest

Funcionalidad: Inicialización de atributos en el controlador

- **Localización:** Archivo `Controllers/SnakeController`, Clase `SnakeController`, método `setAttributes`.
- **Test:** Archivo `Tests/SnakeControllerTest`, Clase `SnakeControllerTest`, método `testSetAttributes`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación funcional.

- **Descripción:** Verifica que los atributos del controlador (nombre del jugador, dimensiones del tablero, nivel de dificultad) se inicialicen correctamente al invocar el método `setAttributes`.

Funcionalidad: Reacción a la pulsación de teclas de dirección

- **Localización:** Archivo `Controllers/SnakeController`, Clase `SnakeController`, método `keyPressed`.
- **Test:** Archivo `Tests/SnakeControllerTest`, Clase `SnakeControllerTest`, método `testKeyPressed_ArrowKeys`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de interacción (mocking con Mockito).
- **Descripción:** Verifica que el controlador actualice correctamente la dirección del jugador cuando se presionan las teclas de dirección (arriba, derecha, abajo, izquierda).

Funcionalidad: Reacción a la pulsación de la tecla "Espacio"

- **Localización:** Archivo `Controllers/SnakeController`, Clase `SnakeController`, método `keyPressed`.
- **Test:** Archivo `Tests/SnakeControllerTest`, Clase `SnakeControllerTest`, método `testKeyPressed_SpaceKey`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de interacción (mocking con Mockito).
- **Descripción:** Verifica que al presionar la tecla "Espacio", se reinicie el juego correctamente si el juego ha terminado.

Funcionalidad: Reacción a la pulsación de la tecla "Escape"

- **Localización:** Archivo `Controllers/SnakeController`, Clase `SnakeController`, método `keyPressed`.
- **Test:** Archivo `Tests/SnakeControllerTest`, Clase `SnakeControllerTest`, método `testKeyPressed_EscapeKey`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de interacción (mocking con Mockito).
- **Descripción:** Verifica que al presionar la tecla "Escape", el juego se cierre correctamente y se vuelva al menú principal si el juego ha terminado.

Funcionalidad: Reacción a la pulsación de "Ctrl+C"

- **Localización:** Archivo `Controllers/SnakeController`, Clase `SnakeController`, método `keyPressed`.
- **Test:** Archivo `Tests/SnakeControllerTest`, Clase `SnakeControllerTest`, método `testKeyPressed_CtrlC`.

- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de interacción (mocking con Mockito).
- **Descripción:** Verifica que al presionar la combinación de teclas "Ctrl+C", el juego intente finalizarse (aunque no se realiza acción en el entorno de prueba debido a un bloque try-catch).

SnakeGameTest

Funcionalidad: Inicialización del juego

- **Localización:** Archivo `Models/SnakeGame`, Clase `SnakeGame`, método `testInitialization`.
- **Test:** Archivo `Tests/SnakeGameTest`, Clase `SnakeGameTest`, método `testInitialization`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de resultados esperados.
- **Descripción:** Verifica que la inicialización del juego establezca correctamente las dimensiones del tablero, la posición inicial del jugador, el puntaje y el número de obstáculos, asegurándose de que el juego no esté en estado "Game Over" al inicio.

Funcionalidad: Reinicio del juego

- **Localización:** Archivo `Models/SnakeGame`, Clase `SnakeGame`, método `testRestart`.
- **Test:** Archivo `Tests/SnakeGameTest`, Clase `SnakeGameTest`, método `testRestart`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de resultados esperados.
- **Descripción:** Verifica que al reiniciar el juego, la posición del jugador, el puntaje, el estado de "Game Over" y el número de obstáculos se restauren a sus valores iniciales, como si fuera un nuevo juego.

Funcionalidad: Actualización del juego sin colisión

- **Localización:** Archivo `Models/SnakeGame`, Clase `SnakeGame`, método `testUpdateWithoutCollision`.
- **Test:** Archivo `Tests/SnakeGameTest`, Clase `SnakeGameTest`, método `testUpdateWithoutCollision`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de resultados esperados.
- **Descripción:** Verifica que al actualizar el juego, el jugador se mueva correctamente sin colisiones y que el estado de "Game Over" siga siendo falso.

Funcionalidad: Actualización del juego con colisión

- **Localización:** Archivo `Models/SnakeGame`, Clase `SnakeGame`, método `testUpdateWithCollision`.

- **Test:** Archivo `Tests/SnakeGameTest`, Clase `SnakeGameTest`, método `testUpdateWithCollision`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de resultados esperados.
- **Descripción:** Verifica que al actualizar el juego, si el jugador colisiona con los límites del tablero, el estado de "Game Over" se activa correctamente, indicando el fin del juego.

Funcionalidad: Recolección de la comida

- **Localización:** Archivo `Models/SnakeGame`, Clase `SnakeGame`, método `testCheckFood`.
- **Test:** Archivo `Tests/SnakeGameTest`, Clase `SnakeGameTest`, método `testCheckFood`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de resultados esperados.
- **Descripción:** Verifica que al comer la comida, el puntaje del jugador se incremente y que la comida se reubique en el tablero, de acuerdo a las reglas del juego.

Funcionalidad: Establecimiento de obstáculos

- **Localización:** Archivo `Models/SnakeGame`, Clase `SnakeGame`, método `testSetObstacles`.
- **Test:** Archivo `Tests/SnakeGameTest`, Clase `SnakeGameTest`, método `testSetObstacles`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de resultados esperados.
- **Descripción:** Verifica que el número de obstáculos se ajuste correctamente según los parámetros establecidos en la configuración del juego, y que el valor máximo y mínimo de obstáculos sea validado correctamente.

Funcionalidad: Validación de condiciones límite

- **Localización:** Archivo `Models/SnakeGame`, Clase `SnakeGame`, método `testValorsFrontera`.
- **Test:** Archivo `Tests/SnakeGameTest`, Clase `SnakeGameTest`, método `testValorsFrontera`.
- **Tipo de test:** Caja negra.
- **Técnicas utilizadas:** Verificación de condiciones de borde (valores inválidos y válidos).
- **Descripción:** Verifica que se lancen excepciones cuando los valores de dificultad o el número de obstáculos no sean válidos, como cuando la dificultad es negativa o supera los valores permitidos, o cuando el número de obstáculos es inválido.













3. Tests de caja blanca

En la sección anterior se puede ver que para cada clase, en su respectiva Test Class, hemos creado una función TestValorsFrontera() donde hemos testado los valores frontera y límite de todas las variables que hemos creído necesarias. Por ende, como ya se muestra en la sección anterior, no hemos creído conveniente añadir otra sección para las pruebas de caja negra.

Para las pruebas de caja blanca tenemos los siguientes tests:

3.1 Statement Coverage

- Después de ejecutar los tests en Eclipse utilizando la extensión Eclemma, nos ha dado los siguientes resultados, en los que se puede observar que se ha llegado en todas las clases de modelo a casi el 100% de statement coverage.

Element	Coverage	Covered Instru...	Missed Instruct...	Total Instructio...
▼  models	 97,6 %	779	19	798
>  SnakeGame.java	 96,7 %	263	9	272
>  SnakeBoard.java	 96,6 %	141	5	146
>  SnakePlayer.java	 98,5 %	321	5	326
>  SnakeFood.java	 100,0 %	24	0	24
>  SnakeObstacle.java	 100,0 %	30	0	30

Resultados de análisis de Statement Coverage

3.2 Decision Coverage

- Para el Decision Coverage, mostramos estas dos capturas de pantalla. En la primera se puede observar que se pasa por todos los casos o decisiones del switch-case, mientras que en la segunda se puede observar que en alguna ocasión ha entrado al if, y en otra, no ha entrado y se ha saltado la excepción.

```

public void move() {
    int[] newHead = {body.get(0)[0], body.get(0)[1]};
    switch (direction) {
        case 0: newHead[1]--; break; // up
        case 1: newHead[0]++; break; // right
        case 2: newHead[1]++; break; // down
        case 3: newHead[0]--; break; // left
    }
    body.add(0, newHead);
    body.remove(body.size() - 1);
}

```

Primer ejemplo de Decision Coverage

```

public SnakePlayer(int x, int y) {
    // Design by contract
    if (x < 0 || y < 0) {
        throw new IllegalArgumentException("Coordenades del SnakePlayer no poden ser negatives");
    }
    body = new ArrayList<>();
    body.add(new int[]{x, y});
    direction = 1;
}

```

Segundo ejemplo de Decision Coverage

3.3 Condition Coverage

- Para el Condition Coverage volvemos a tener dos capturas de pantalla. En la primera se puede observar que para las 4 condiciones dentro del if, los 8 posibles resultados se han cubierto. En la segunda imagen, mostramos un caso menos favorecido, donde de los 8 posibles resultados, se han cubierto 6.

```

public SnakeBoard(int width, int height) {
    // Design by contract
    if (width <= 0 || height <= 0 || width >= 500 || height >= 500) {
        throw new IllegalArgumentException("Dimensions de la Board han de ser positives i menors a 500");
    }
    this.width = width;
    this.height = height;
    this.obstacles = new ArrayList<>();
    this.food = new SnakeFood(0, 0);
}

```

◆ All 8 branches covered.

Primer ejemplo de Condition Coverage

```

public boolean collidesWithWall(SnakeBoard board) {
    int[] head = body.get(0);
    return head[0] < 0 || head[0] >= board.getWidth() || head[1] < 0 || head[1] >= board.getHeight();
}

```

◆ 2 of 8 branches missed.

Segundo ejemplo de Condition Coverage

3.4 Loop Testing

- Para realizar loop testing, nos hemos fijado en el siguiente loop. Para testearlo, hemos aprovechado que tenemos una función donde testeamos el número de obstáculos añadidos a la clase SnakeGame. En este test, hemos utilizado como valores de prueba para el numero de obstáculos: {-1, 0, 5, 98, 100}

```
for (SnakeObstacle obstacle : obstacles) {  
    if (obstacle.getX() == x && obstacle.getY() == y) {  
        return true;  
    }  
}  
return false;
```

Resultados del Loop Testing

3.5 Path Coverage

- Para el path coverage, hemos tomado la función testCollisions() de la clase SnakePlayerTest y hemos hecho el diagrama de actividades de cuando testeamos el funcionamiento de la función CollidesWithSelf() de la clase SnakePlayer.

Como se puede ver en el código primero hacemos que la serpiente tenga la longitud suficiente para poder colisionar consigo misma y testeamos que la función CollidesWithSelf() devuelva False, ya que aún no hemos hecho que choque. Esto se puede observar en el diagrama a partir del path formado por las flechas rojas. En cambio después, si que se hace que la serpiente choque consigo misma cambiando su dirección y haciendo que se mueva. Esto se muestra en el diagrama a partir del path formado por las flechas azules.

```

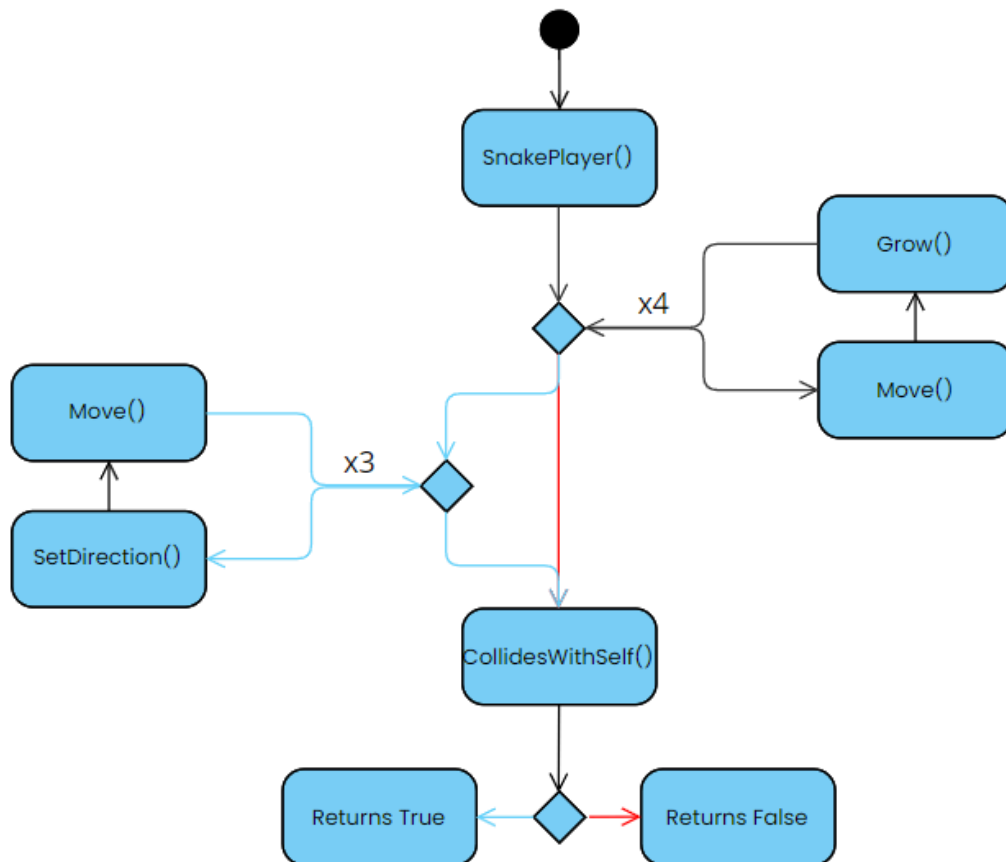
@Test
public void testCollisions() {
    // Use a mock SnakeBoard object
    SnakeBoardMock board = new SnakeBoardMock(10, 10);
    SnakePlayer snake = new SnakePlayer(0, 0);

    snake.setDirection(0); // Move up (out of bounds)
    snake.move();
    assertTrue(snake.collidesWithWall(board), "Wall collision no detectada");

    snake.restart(5, 5);
    snake.grow();
    snake.move();
    snake.grow();
    snake.move();
    snake.grow();
    snake.move();
    snake.grow();
    snake.move();
    assertFalse(snake.collidesWithSelf(), "No s'hauria de detectar Self-collision");
    snake.setDirection(2); // Down
    snake.move();
    snake.setDirection(3); // Left
    snake.move();
    snake.setDirection(0); // Up
    snake.move();
    assertTrue(snake.collidesWithSelf(), "Self-collision no detectada");
}

```

Código de Test para la función CollidesWithSelf()



4. Design by Contract

Para que se realice Design by Contract a medida que se ejecuta el código del cliente, hemos puesto varios asserts para comprobar que ciertas variables no tomen valores erróneos en producción. Esto se ha hecho sobre todo en variables que pueden ser peligrosas depende de que ponga el usuario.

Un ejemplo, es que si el usuario de alguna forma logra introducir un número extremadamente grande en el número de obstáculos, podría ser un peligro para el ordenador, ya que intentaría cargar demasiados objetos. Aquí abajo mostramos el assert usado para este caso.

```
public void setObstacles(int numObstacles) {  
    // Design by contract  
    if (numObstacles < 0 || numObstacles > board.getHeight()*board.getWidth() - 1) {  
        throw new IllegalArgumentException(s:"Numero de obstacles no pot ser menor a 0 o major al nombre de tiles");  
    }  
    for (int i = 0; i < numObstacles; i++) {  
        generateObstacle();  
    }  
}
```

Ejemplo de uso de asserts para la implementación de Design by Contract

5. Mock Objects

- Para realmente practicar el concepto de los Mock Objects, los hemos implementado de dos formas. Una más manual donde simplemente creamos una clase hija de la clase que queremos implementar, y otra más automática donde usamos la librería Mockito para generarla. Aquí abajo mostramos imágenes de los dos usos.

```
import models.SnakeBoard;  
  
public class SnakeBoardMock extends SnakeBoard{  
    private int width;  
    private int height;  
  
    public SnakeBoardMock(int width, int height) {  
        super(width, height);  
    }  
  
    public int getWidth() {  
        return width;  
    }  
  
    public int getHeight() {  
        return height;  
    }  
}
```

Implementación manual de Mock Objects

```

public class SnakeControllerTest {

    private SnakeController controller;
    private SnakeGame mockGame;
    private SnakeView mockView;

    @BeforeEach
    public void setUp() {
        controller = new SnakeController();

        // Mock objects creates amb Mockito
        mockGame = mock(SnakeGame.class);
        mockView = mock(SnakeView.class);
    }
}

```

Implementación automática de Mock Objects mediante Mockito

6. CI/CD

Usamos maven para ejecutar los tests mediante Github Actions, se realizan dos tests, uno de checkstyle predefinido y nuestro test, que está indicado en el archivo `.github/.workflows/test.yaml` y en el archivo `pom.xml` definiendo las reglas y dependencias.

Comandos que se ejecutan dentro de los archivos `.yaml`

```

run: mvn checkstyle:checkstyle

run: mvn test

```

Declaración de los archivos que se van a ejecutar a la hora de realizar el comando `mvn test`.

```

<build>
  <plugins>
    <!-- Configuración del maven-surefire-plugin para ejecutar las pruebas -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M5</version>
      <configuration>
        <!-- Especifica las clases de prueba a ejecutar -->
        <includes>
          <include>**/SnakeBoardTest.java</include>
          <include>**/SnakeFoodTest.java</include>
          <include>**/SnakeGameTest.java</include>
          <include>**/SnakeObstacleTest.java</include>
          <include>**/SnakePlayerTest.java</include>
        </includes>
      </configuration>
    </plugin>
  </plugins>
</build>

```

Demostración que los tests funcionan en cada pull request hacia la rama main.

albertceballos0 / SnakeGame

Q. Type to search

<> Code

Issues

Pull requests

Actions

Projects

Wiki

Security

Insights

Settings

SnakeGame

Private

Unwatch 1

Fork 0

Star 0

testingCI

5 Branches

0 Tags

Go to file

Add file

<> Code

About

This branch is 4 commits behind main.

Contribute

albertceballos

añadir archivo de prueba

21a11b · 5 minutes ago

58 Commits

.github/workflows

Agregar instalación de Maven y simplificar comandos en f...

2 hours ago

src

añadir código de prueba

21a11b · 5 minutes ago

README.md

añadir archivo de prueba

21a11b · 5 minutes ago

pom.xml

añadir archivo de prueba

21a11b · 5 minutes ago

prueba_CI.txt

añadir archivo de prueba

21a11b · 5 minutes ago

README

Snake Game

All checks have passed

2 successful checks

✓ Checkstyle / checkstyle (pull_request) Successful in 20s

✓ Run Tests / test (pull_request) Successful in 34s

Readme

Activity

0 stars

1 watching

0 forks

releases

0 releases published

create a new release

Packages

No packages published

Publish your first package

Contributors 2

GerardAsbert

albertceballos0

Languages

Java 100.0%