Practica 1 TQS: Queens TDD

Julen Cruz Gómez 1667663
Igor Ulyanov 1667150
Jueves 12:30-14:30

Introducción a la práctica: definición del Juego y resumen de funcionamiento

Reglas del Juego:

En esta práctica lo que se procura es realizar el juego llamado **queens**. Es una variante del **problema de la reinas**, en el cual se deben colocar reinas de manera que ninguna de ellas amenace a ninguna otra. En nuestro juego existen **reglas adicionales** y modificadas:

- Las **reinas** solo **atacan en su fila**, **columna** y una **casilla adyacente**, incluidas diagonales
- Existen **secciones** de colores en el tablero, solo puede haber **1 reina por sección**
- El juego **termina** cuando se han **posicionado todas las reinas** posibles (1 por fila/columna)

Implementación teórica:

La idea de implementación consiste en lo siguiente:

- **Generar** el puzzle (un tablero con secciones) con **al menos una solución posible**. De esta tarea **se ocupa la clase Board**, además de la **modificación y guardado** de las **secciones** e instancias de **casillas**
- Cada **casilla** guarda su **color** y estado de **disponibilidad** (atributo Available), debe ser un **int** ya que más de una reina puede estar inhabilitando la casilla y solo está **disponible cuando no hay ninguna reina deshabilitando la**.
- La clase **Game** se ocupa de **gestionar el juego**. Guarda un array de **reinas**, un **board** y un **visualizador**.
- Los objetos **reina** se ocupan de guardar sus **coordenadas** y tener los métodos correspondientes para **deshabilitar las casillas** adecuadas en el tablero. Lo hemos realizado así para tener la posibilidad de añadir reinas especiales pero no fue el caso.
- El objeto del **visualizador** se ocupa de **sacar información** pertinente por pantalla además de **pedir inputs** del jugador pero **no se ocupa de verificar** la información que recibe.

El bucle principal se compone de los siguientes pasos:

- Enseñar tablero corriente
- pedir coordenadas al jugador
- si la coordenadas pertenecen a una reina, quitarla
- si las coordenadas no pertenecen a una reina y es posible: poner la reina
- volver al inicio

Funcionalidad: Generación del tablero inicial

Posicionamiento de reinas para determinar la solución (con backtracking), asignación de colores a las reinas, generación de secciones a partir de las reinas y reset de los valores innecesarios del tablero para el comienzo correcto de la partida.

Es la clase Strategy del patrón de diseño strategy y además es un template. Este diseño permite añadir diferentes maneras de generar las secciones ya que la función createSections() es abstracta y se debe implementar en las extensiones concretas de la clase.

Localitzacion:

- src/developedCode/GenerationStrategy.java, clase GenerationStrategy, funcion = generate()
- src/developedCode/Board.java, clase board, funcion createSectionsList()
- src/developedCode/GusanilloFill.java clase GusanilloFill funcion = createSections()

Se compone de las llamadas a las funciones generateQueens(), assignColorToQueens(), fillBlanksAndReset() (concretas) y createSections() (Abstracta).

Test:

GusanilloFillTest.java contiene todos los tests de esta funcionalidad, a continuación los detallaremos por orden.

Para todos los siguientes test se utiliza el **objeto mock MockRNG**, creado por nosotros, su implementación está en testClasses/MockRNG.java. Esta clase tiene una función donde recibe un valor máximo y mínimo (para imitar la real) pero solo devuelve el siguiente valor de la matriz que se le pasa en el parámetro. Esto es imperativo para realizar test

GenerationStrategy -> testGenerateQueens()

Se realizan los tests con una matriz 4x4 donde solo existen 2 soluciones posibles (distribucion de reinas en ((0,1), (1,3), (2,0), (3,2)) o en ((0,2), (1,0), (2,3), (3,1))), en los tests se seleccionan valores de random para que sucedan los siguientes casos: se consiga la primera distribución, se consiga la segunda distribucion y se realiza la primera distribución con necesidad de backtracking.

GenerationStrategy -> testMarkAffected(), testUnmarkAfected()

Estos dos tests son muy similares ya que testean dos funciones iguales en concepto pero inversas en resultado. Lo que realizan es marcar como disponibles o quitar la disponibilidad de ciertas casillas una vez se ha colocado una posible reina en la función generateQueens() y se desmarca las casillas afectadas una vez se ha hecho backtracking y, por lo tanto, quitado la reina. Esto es importante para que la función generateQueens() escoja correctamente la posibles casillas donde puede poner una reina.

Hemos decidido realizar un condition y decisión coverage en este método ya que requiere de muchos ifs.

```
    private void unmarkAffectedPositions(Square[][] matrix, int row, int col) {
        if (row >= 0 && row < size && col >= 0 && col < size)
        {
            for (int i = 0; i < size; i++) {
                if (!matrix[row][i].hasQueen()) {
                    matrix[row][i].enable();
                }
            }

            for (int i = 0; i < size; i++) {
                if (!matrix[i][col].hasQueen()) {
                    matrix[i][col].enable();
                }
            }
            if (row - 1 >= 0 && col - 1 >= 0) {
                matrix[row - 1][col - 1].enable();
            }
            if (row - 1 >= 0 && col + 1 < size) {
                matrix[row - 1][col + 1].enable();
            }
            if (row + 1 < size && col - 1 >= 0) {
                matrix[row + 1][col - 1].enable();
            }
            if (row + 1 < size && col + 1 < size) {
                matrix[row + 1][col + 1].enable();
            }

            matrix[row][col].setAvailable(0);
        }
    }
```

```
    private void markAffectedPositions(Square[][] matrix, int row, int col) {
        if (row >= 0 && row < size && col >= 0 && col < size)
        {
            matrix[row][col].setAvailable(1000);
            for (int i = 0; i < size; i++) {
                if (!matrix[row][i].hasQueen()) {
                    matrix[row][i].disable();
                }
            }

            for (int i = 0; i < size; i++) {
                if (!matrix[i][col].hasQueen()) {
                    matrix[i][col].disable();
                }
            }
            if (row - 1 >= 0 && col - 1 >= 0) {
                matrix[row - 1][col - 1].disable();
            }
            if (row - 1 >= 0 && col + 1 < size) {
                matrix[row - 1][col + 1].disable();
            }
            if (row + 1 < size && col - 1 >= 0) {
                matrix[row + 1][col - 1].disable();
            }
            if (row + 1 < size && col + 1 < size) {
                matrix[row + 1][col + 1].disable();
            }
        }

    }
```

Para conseguir este resultado hemos utilizado los parametros de entrada (x,y) siguientes: (-1,-1)(-100, -1)(-1,-100)(100,100)(100,1)(1,100)(0,-1)(-1,0)(1,3)(0,0)(2,1) para mark afected y ademas (3,0)(3,3).

GenerationStrategy -> testAssignColorsToQueens()

En esta función se atribuyen colores de una lista de tipo static a las reinas que hay en el tablero. Recoge las coordenadas de una lista de reinas que se guarda internamente, copia la lista de colores y mediante un random selecciona qué color asignare, quitandolo de la lista. Debido a que hace un for iterando por todas las reinas (máximo 8) hemos decidido realizar un **loop testing de un for simple** con las cantidades de iteraciones siendo: 0,1,2,4,7,8;

En el test queda detallado con comentarios donde se hace el test de cada número de iteraciones del bucle. En cada teste, básicamente, se hace una clase Generation Strategy nueva donde se la pasa el mockRNG con los valores correctos y se añade una lista de reinos que dicta el número de iteraciones.

```
    protected Square[][] assignColorToQueens(Square[][] queenMatrix){
        ArrayList<String> colors = new ArrayList<>(Colors.colorArray);
        for(ArrayList<Integer> coord : queensPosition) {
            int index = rng.random(0, colors.size()-1);
            queenMatrix[coord.get(0)][coord.get(1)].setColor(colors.get(index));
            colors.remove(index);

        }
        return queenMatrix;
    }
```

GusanilloFill -> testCreateSections()

Esta función realiza la generación de las secciones según el método GusanilloFill, en el cual desde cada reina se escoge un número aleatorio de pasos (de 0 al tamaño de la matriz) . En cada paso, se miran las casillas posibles siguientes para pintar del mismo color que la reina y se escoge uno de manera aleatoria.

Ya que la función realiza un bucle anidado, hemos decidido realizar un **testing del bucle anidado**. Las iteraciones del bucle interior son determinadas por el valor que concede la clase Random y el bucle externo es una iteración de las reinas del tablero.

Primero se testea el bucle interno, después de hacer un setup (crear los objetos necesarios y asignar los valores al mockRNG) se realiza un for en el cual testea todos los valores posibles. Los valores escogidos para el loop testing son (iteraciones bucle externo, iteraciones bucle interno) → (1,0)(1,1)(1,2)(1,5)(1,7)(1,8)(0,1)(1,1)(2,1)(3,1)(5,1)(7,8). En este caso 8 es el valor máximo ya que hemos decidido que ese es el número máximo de reinas que puede haber en el tablero y por tanto la dimensión de tablero más grande que asumimos.

```
for (Integer value : valuesToTest) {
    nextColCorr = 7 - value;
    for (int i = 7; i >= nextColCorr; i--)
    {
        if (nextColCorr == -1) {
            correctInteriorMatrix[0][1].setColor(Colors.BACKGROUND_RED);
        } else {
            correctInteriorMatrix[i][0].setColor(Colors.BACKGROUND_RED);
        }

    }

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            resultInteriorMatrix[i][j] = new SquareDefault();
        }
    }

    resultInteriorMatrix[7][0].setAvailable(queen);
    resultInteriorMatrix[7][0].setColor(Colors.BACKGROUND_RED);

    resultInteriorMatrix = loopTester.callCreateSections(resultInteriorMatrix);

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            assertEquals(correctInteriorMatrix[i][j].getColor(), resultInteriorMatrix[i][j].getColor());
        }
    }
```

Antes de este bucle se asegura de que la clase GusanilloFill solo contiene 1 reina en su array para fijar las iteraciones del bucle externo. En cada iteración del bucle interno primero se actualiza la matriz contra la que compararemos, después de resetea y ejecuta la función sobre a la matriz de test. Finalmente se realiza una serie de asserts para comprobar el resultado

```
int col = 0;
for (Integer value: valuesToTest) {
    loopTester.setQueensPosition(exteriorQueens);
    while (exteriorQueens.size() < value){
        exteriorQueens.add(new ArrayList<>(Arrays.asList(7, col)));
        correctExteriorMatrix[7][col].setAvailable(queen);
        correctExteriorMatrix[7][col].setColor(Colors.colorArray.get(col));
        correctExteriorMatrix[6][col].setColor(Colors.colorArray.get(col));
        col++;
    }

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            resultExteriorMatrix[i][j] = new SquareDefault();
        }
    }

    for (int i = 0; i < value; i++)
    {
        resultExteriorMatrix[7][i].setColor(Colors.colorArray.get(i));
        resultExteriorMatrix[7][i].setAvailable(queen);
    }

    resultExteriorMatrix = loopTester.callCreateSections(resultExteriorMatrix);

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            assertEquals(correctExteriorMatrix[i][j].getColor(), resultExteriorMatrix[i][j].getColor());
        }
    }

}
```

Para el testing del bucle externo se fija el valor de iteraciones del externo (es decir el mockRNG siempre sacará el mismo valor siempre que se pida un número de casillas que pintar, 1 en este caso, y siempre se dará la misma opción para cual de los pasos se escoge, la primera en este caso). Después de haber hecho el setup correctamente para cada valor de cantidad de iteraciones se hacen los siguientes pasos: se añaden las reinas necesarias al array de reinas, se posicionan las reinas correctamente con su color adecuado en la matriz de resultado junto con la casilla de encima del mismo color, se resetea la matriz de test, se hace un setup de la matriz de test y finalmente se ejecuta la función y se hacen los asserts necesarios.

```java
    @Override
    protected Square[][] createSections(Square[][] coloredMatrix) {
        for(ArrayList<Integer> coords : queensPosition)
        {
            int row = coords.get(0);
            int col = coords.get(1);
            int stepsNumber = rng.random(0, size);
            String color = coloredMatrix[row][col].getColor();
            for (int i = 0; i < stepsNumber; i++)
            {

                ArrayList<ArrayList<Integer>> steps = new ArrayList<>();

                if (row-1 >= 0 && !coloredMatrix[row-1][col].hasQueen()) {
                    steps.add(new ArrayList<>(Arrays.asList(-1, 0)));
                }

                if (col+1 < size && !coloredMatrix[row][col+1].hasQueen()) {
                    steps.add(new ArrayList<>(Arrays.asList(0, 1)));
                }
                if (row+1 < size && !coloredMatrix[row+1][col].hasQueen()) {
                    steps.add(new ArrayList<>(Arrays.asList(1, 0)));
                }
                if (col-1 >= 0 && !coloredMatrix[row][col-1].hasQueen()) {
                    steps.add(new ArrayList<>(Arrays.asList(0, -1)));
                }

                if (steps.isEmpty()) {
                    break;
                }

                int direction = rng.random(0, steps.size()-1);

                row = row + steps.get(direction).get(0);
                col = col + steps.get(direction).get(1);
                coloredMatrix[row][col].setColor(color);
            }
        }
        return coloredMatrix;
    }

}
```

GenerationStrategy -> testFillBalnksAndReset()

Esta función rellena los posibles huecos sin color del tablero y además resetea el valor de “available” de cada casilla a 0 para poder tener el tablero preparado para jugar. Se testean los diferentes casos en los que se puede encontrar una casilla vacía (sin vecinos que tengan color, con 1, 2 ,3 o 4 vecinos con color). Además de comprobar los colores, se mira que el valor de “available” acabe en 0 en cada caso. Se detalla más en los comentarios de la función.

Board -> testCreateSectionsList()

Este metodo genera un array de secciones segun las coordenadas que le sean pasadas. En este caso se le pasa un array de coordenadas que coincide con las posiciones de las reinas escogidas en la generacion del tablero. Resulta en la modificacion de la propia clase board añadiendo las secciones necesarias.

Hemos decidido hacer un **loop testing simple** donde el valor maximo del loop es 8. Hemos escogido los valores: 0,1,2,5,7,8.

```
    private void createSections(ArrayList<ArrayList<Integer>> queensPosition) {
        sections.clear();
        String color = null;
        for (ArrayList<Integer> coords : queensPosition) {
            color = squares[coords.get(0)][coords.get(1)].getColor();
            sections.add(new Section(color));
        }
    }
```

testGenerate()

En este test se idea un tablero final que se hace manualmente, para esto se escogen unos valores muy concretos de random que permiten que la función realice los pasos necesarios. El test se compone de nada más que una generación de 1 matriz 4x4 ya que es un template method con tan solo llamas a funciones de alto nivel.

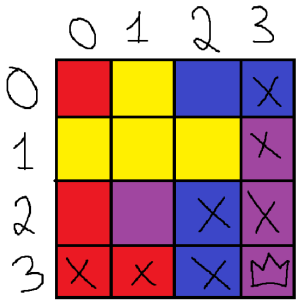
Funcionalidad: Mirar si una casilla está disponible para poner una reina

Localitzacion:

- src/developedCode/Board.java, clase Board , funcion = isSquareAvailable()

Test:

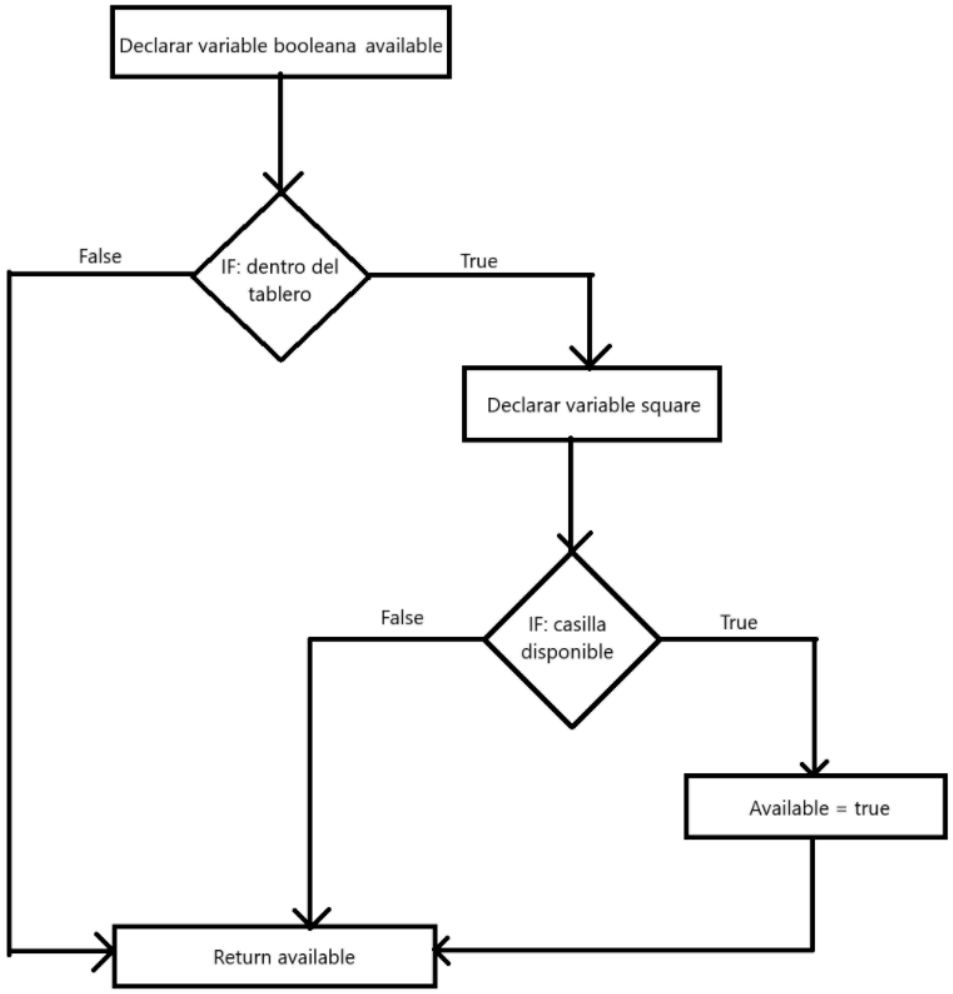
Para este test generamos un tablero de la siguiente manera para poder hacer el **path, condition y decision coverage**.



Lo primero que hemos realizado es el path coverage, el codigo:

```
public boolean isSquareAvailable(int x, int y) {
    boolean available = false;
    if (x >= 0 && x < size && y >= 0 && y < size) {
        Square square = squares[x][y];
        if(!square.isDisabled() && getSquaresSection(square) != null && !getSquaresSection(square).isDisabled()){
            available = true;
        }
    }
    return available;
}
```

Se puede representar según este diagrama de flujo:



Segun la formula aristas - nodos + 2 ⟶ 7-6 +2 = 3 caminos que debemos realizar para path coverage. Estos tres caminos se representan mirando la disponibilidad de las coordenadas (-1,-1) casilla fuera del tablero; (2,3): casilla dentro del tablero pero no disponible; (0,0) casilla disponible.

Ademas para realizar condition y decision coverage hemos utilizado los valores siguientes:
(3,2)(2,1)(1,1)(1,-10)(1,100)(4,4)(-10,-10)(-10,1)(100,1)(100,100)

Funcionalidad: Actualizar disponibilidad de sección cuando se coloca una reina.

Localizacion:

- src/developedCode/Board.java, clase Board , funcion = removeQueenInSection(int x, int y) & placeQueenInSection(int x, int y)
- auxiliar: src/developedCode/Board.java, clase Board , funcion = getSquaresSection(int x, int y)

Test:

Estas funciones (removeQueenInSection(int x, int y) & placeQueenInSection(int x, int y)) primero llaman a la funcion getSquaresSection() que devuelve el objeto de la sección a la que pertenece la casilla indicada en las coordenadas. Despues la deshabilita o habilita, segun la funcion, la seccion de vuelta.

testGetSquaresSection()

Se prueba recuperar la sección de 4 casillas diferentes, con secciones diferentes. Además se prueba de recuperar una casilla que tiene un color para el cual no existe sección y, porlo tanto, devuelve null (esto nunca debería) pasar en una ejecución real.

testRemoveQueenInSection() y placeQueenInSection()

Debido a que esta función siempre se ejecutará después de que se mire que una casilla está disponible tan solo se tiene que habilitar y deshabilitar la sección adecuada. En el test se prueba de habilitar/deshabilitar 4 secciones diferentes de una matriz 2x2.

Funcionalidad: Poner/Quitar una reina del tablero con un input válido del jugador

Localizacion:

- src/developedCode/Game.java, clase Game , funcion = placeOrRemoveQueen(int x, int y)
- auxiliar: src/developedCode/Game.java, clase Game , funcion = isQueenCoords(int x, int y)

Test:

Esta función se encarga primero de mirar si las coordenadas obtenidas pertenecen a un reina con la función auxiliar isQueenCoords(). Esta función devolverá la reina a la que pertenecen las coordenadas o, en caso de que no pertenezcan a ninguna, null. Según el resultado de esta anterior llamada, llamará a las funciones adecuadas para actualizar la disponibilidad de casillas en el tablero además de añadir o quitar la reina nueva/existente al array.

testIsQueenCoords()

En los tests para esta función se añaden dos reinas al objeto game y luego se mira si la función devuelve de manera correcta sus objetos cuando se le pasan sus coordenadas exactas por parámetro y que devuelva null cuando no sea el caso.

testPlaceOrRemoveQueen()

En este test verificamos que la función realice las llamadas correctas a las funciones de otros objetos que deberían mediante mock de board (mockito), (no testeamos su output ya que no es necesario, han sido testeadas por separado) además de mirar si añade/quita las reinas correctas de la lista cuando le toca.

Funcionalidad: Recibir un input válido del usuario

Localizacion:

- src/developedCode/Game.java, clase Game , funcion = getCoords()

Test:

Esta función llama al método del visualizador que pide un input del usuario hasta que el input sea una coordenada válida dentro del tablero. Para ello realizamos un mock con mockito del visualizador que devuelve valores preprogramados, tanto válidos como no. Comprobamos que la función sigue pidiendo inputs hasta que le devuelve uno válido y lo recibimos nosotros, realizando un assert.

Funcionalidad: Deshabilitar/habilitar casillas al poner una reina

Localizacion:

- src/developedCode/QueenDefault.java, clase QueenDefault , funcion = disableSquares() / ensableSquares()

Test:

Esta función hace llamadas del método disableSquare() de la instancia Game (que guarda). Por lo tanto hacemos un objeto mock Game (en el fichero src/testClasses/MockObjects/MockGame) que hace assert cada vez que se llama a su función disableSquare() con los parámetros x e y correctos.

Evidencias de pruebas:

Pairwise testing:

- **Test:** src/testCode/BoardTest.java, metodo = testDisableAndEnableSquare()

(-1,-1)(0,0)(1,1) → valor frontera inferior y sus limites

(2,2)(3,3)(4,4) → valor frontera superior y sus limites

(-10, -10)(-10,1)(-10,100) |

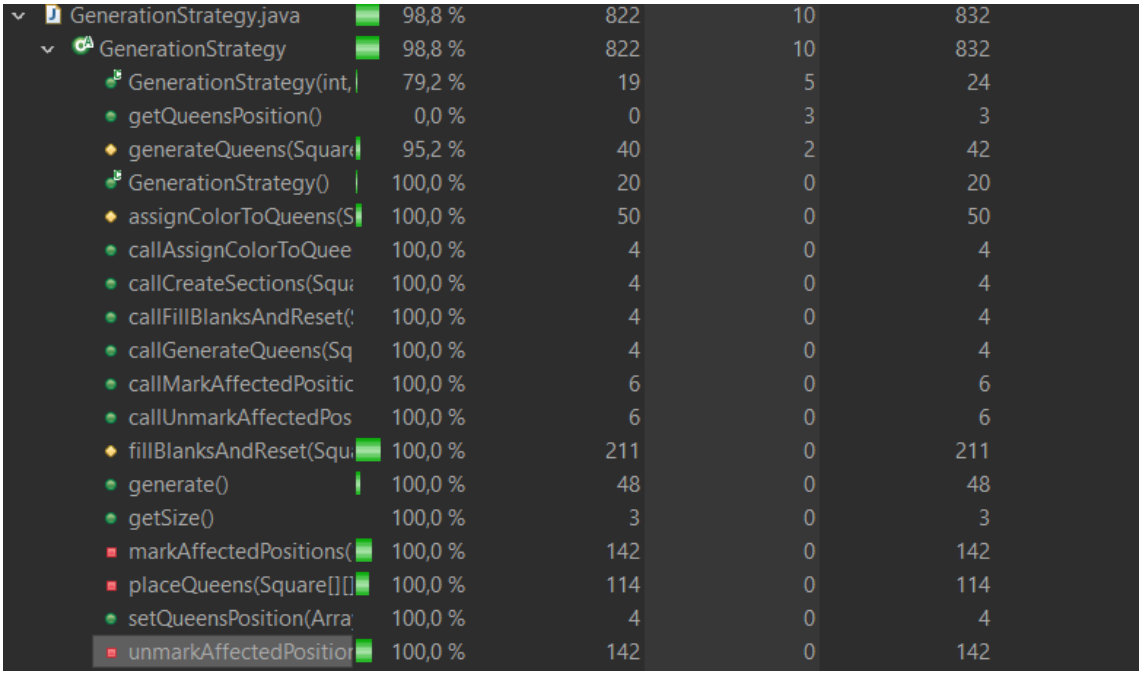
(1, -10)(1,1)(1,100) | → pairwise testing de x,y

(100, -10)(100,1)(100,100) |

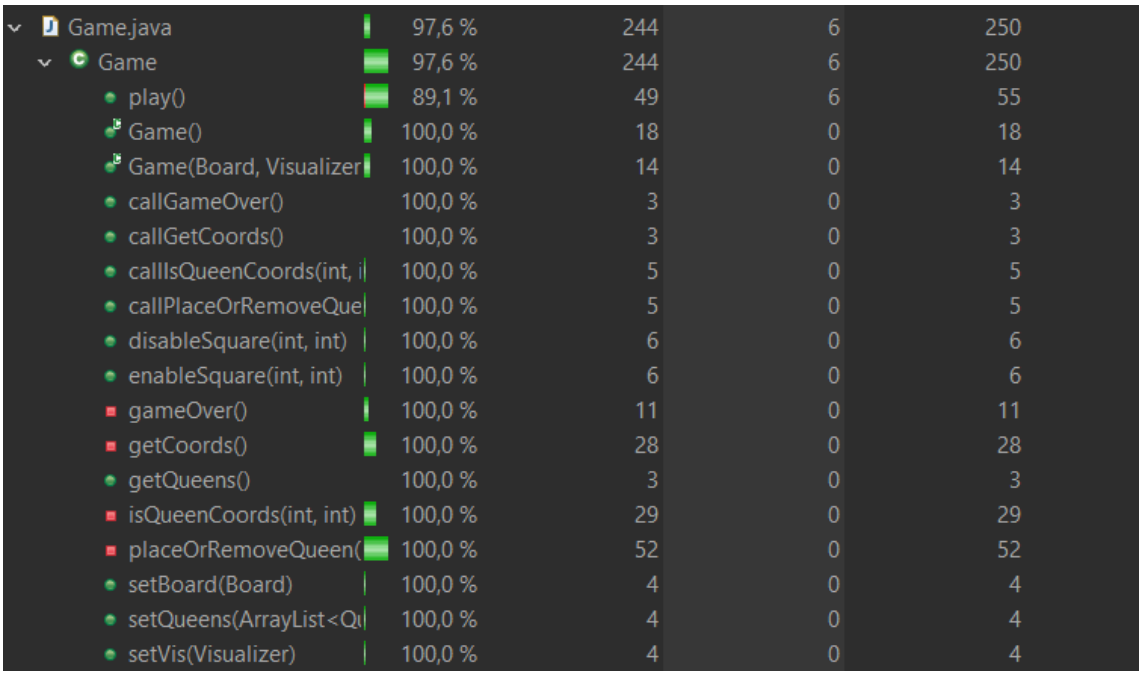
Statement Coverage:

coverage del proyecto completo

Coverage clase src/developedCode/GenerationStrategy.java

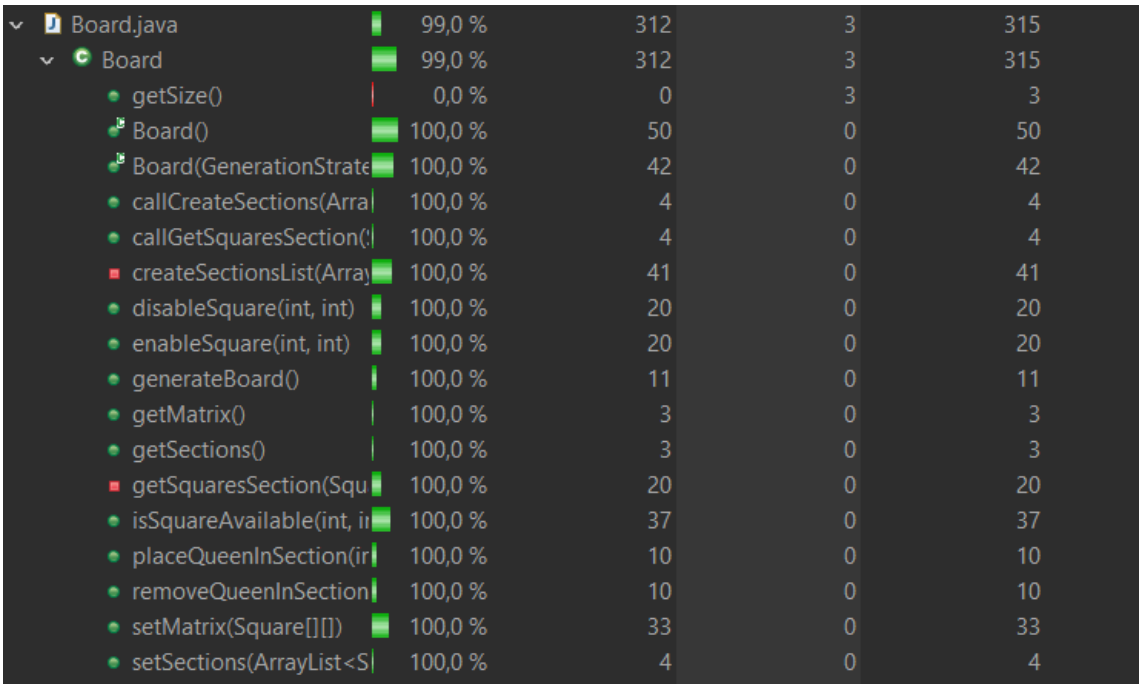


Coverage src/developedCode/Game.java



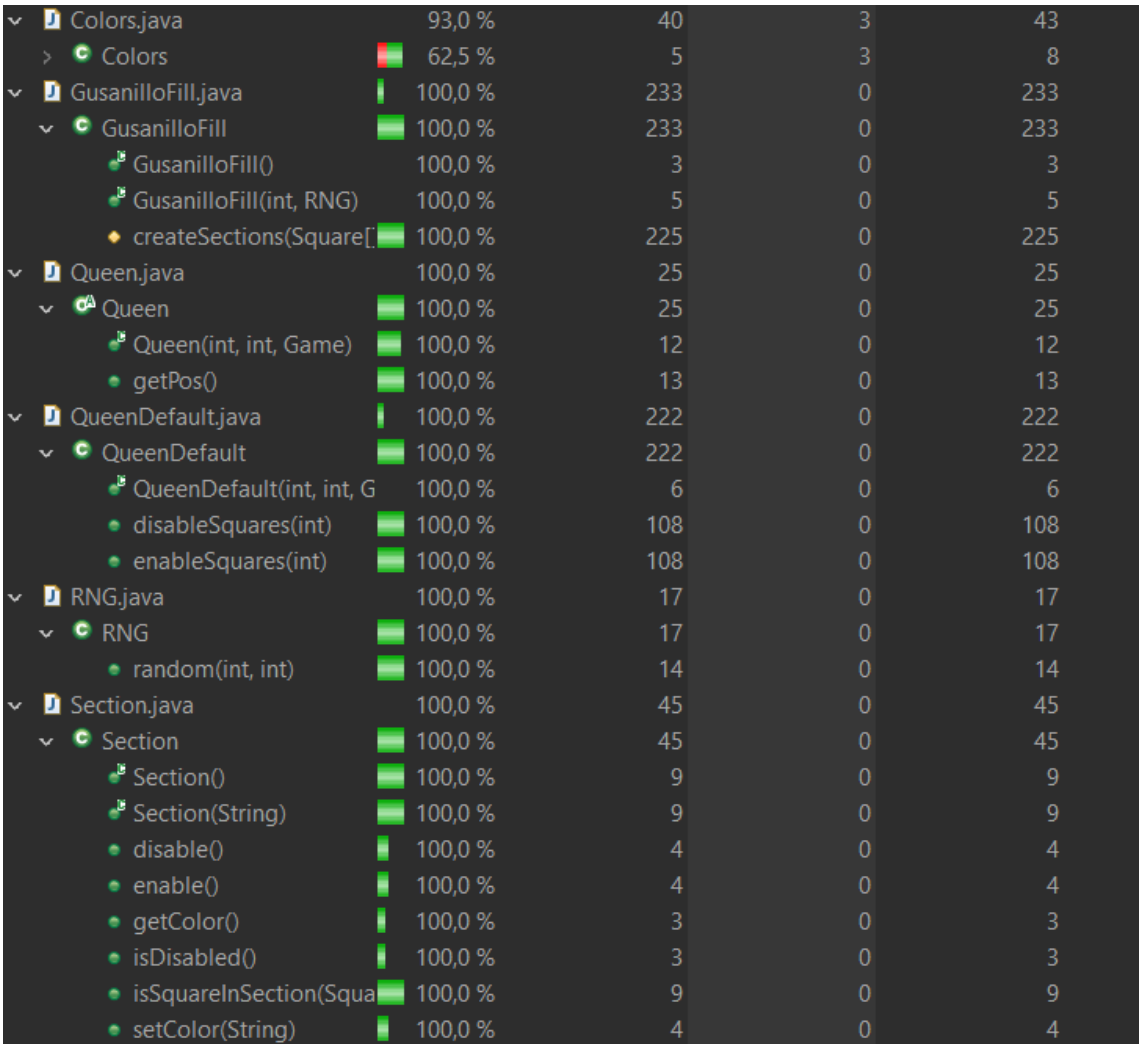
Game.java	97,6 %	244	6	250
Game	97,6 %	244	6	250
play()	89,1 %	49	6	55
Game()	100,0 %	18	0	18
Game(Board, Visualizer	100,0 %	14	0	14
callGameOver()	100,0 %	3	0	3
callGetCoords()	100,0 %	3	0	3
callIsQueenCoords(int, i	100,0 %	5	0	5
callPlaceOrRemoveQue	100,0 %	5	0	5
disableSquare(int, int)	100,0 %	6	0	6
enableSquare(int, int)	100,0 %	6	0	6
gameOver()	100,0 %	11	0	11
getCoords()	100,0 %	28	0	28
getQueens()	100,0 %	3	0	3
isQueenCoords(int, int)	100,0 %	29	0	29
placeOrRemoveQueen(	100,0 %	52	0	52
setBoard(Board)	100,0 %	4	0	4
setQueens(ArrayList<Q	100,0 %	4	0	4
setVis(Visualizer)	100,0 %	4	0	4

Coverage src/developedCode/Board.java

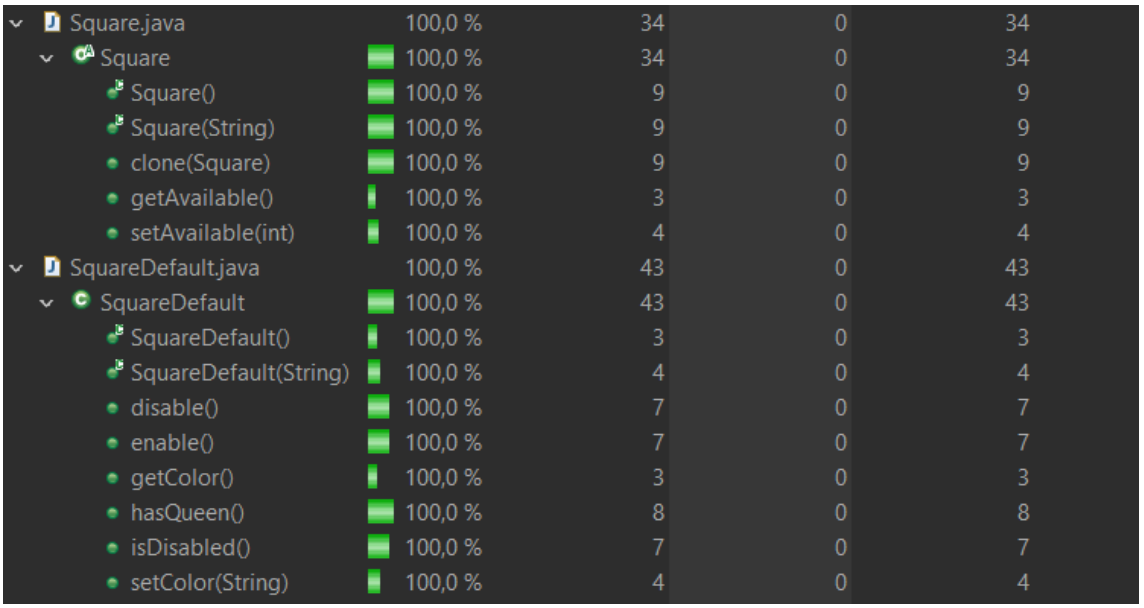


Board.java	99,0 %	312	3	315
Board	99,0 %	312	3	315
getSize()	0,0 %	0	3	3
Board()	100,0 %	50	0	50
Board(GenerationStrate	100,0 %	42	0	42
callCreateSections(Arra	100,0 %	4	0	4
callGetSquaresSection(	100,0 %	4	0	4
createSectionsList(Array	100,0 %	41	0	41
disableSquare(int, int)	100,0 %	20	0	20
enableSquare(int, int)	100,0 %	20	0	20
generateBoard()	100,0 %	11	0	11
getMatrix()	100,0 %	3	0	3
getSections()	100,0 %	3	0	3
getSquaresSection(Squ	100,0 %	20	0	20
isSquareAvailable(int, i	100,0 %	37	0	37
placeQueenInSection(in	100,0 %	10	0	10
removeQueenInSection	100,0 %	10	0	10
setMatrix(Square[][])	100,0 %	33	0	33
setSections(ArrayList<S	100,0 %	4	0	4

Coverage src/developedCode/Colors.java & Gusanillofill.java & Queen.java & QueenDefaul.java & RNG.java & Section.java & Square.java & SquareDefault.java



Colors.java	93,0 %	40	3	43
Colors	62,5 %	5	3	8
GusanilloFill.java	100,0 %	233	0	233
GusanilloFill	100,0 %	233	0	233
GusanilloFill()	100,0 %	3	0	3
GusanilloFill(int, RNG)	100,0 %	5	0	5
createSections(Square[	100,0 %	225	0	225
Queen.java	100,0 %	25	0	25
Queen	100,0 %	25	0	25
Queen(int, int, Game)	100,0 %	12	0	12
getPos()	100,0 %	13	0	13
QueenDefault.java	100,0 %	222	0	222
QueenDefault	100,0 %	222	0	222
QueenDefault(int, int, G	100,0 %	6	0	6
disableSquares(int)	100,0 %	108	0	108
enableSquares(int)	100,0 %	108	0	108
RNG.java	100,0 %	17	0	17
RNG	100,0 %	17	0	17
random(int, int)	100,0 %	14	0	14
Section.java	100,0 %	45	0	45
Section	100,0 %	45	0	45
Section()	100,0 %	9	0	9
Section(String)	100,0 %	9	0	9
disable()	100,0 %	4	0	4
enable()	100,0 %	4	0	4
getColor()	100,0 %	3	0	3
isDisabled()	100,0 %	3	0	3
isSquareInSection(Squa	100,0 %	9	0	9
setColor(String)	100,0 %	4	0	4



Square.java	100,0 %	34	0	34
Square	100,0 %	34	0	34
Square()	100,0 %	9	0	9
Square(String)	100,0 %	9	0	9
clone(Square)	100,0 %	9	0	9
getAvailable()	100,0 %	3	0	3
setAvailable(int)	100,0 %	4	0	4
SquareDefault.java	100,0 %	43	0	43
SquareDefault	100,0 %	43	0	43
SquareDefault()	100,0 %	3	0	3
SquareDefault(String)	100,0 %	4	0	4
disable()	100,0 %	7	0	7
enable()	100,0 %	7	0	7
getColor()	100,0 %	3	0	3
hasQueen()	100,0 %	8	0	8
isDisabled()	100,0 %	7	0	7
setColor(String)	100,0 %	4	0	4

Decision Coverage y Condition Coverage:

- src/developedCode/GenerationStrategy.java -> markAffected() & unmarkAffected (métodos muy similares, contado como 1):
- **Tests**: src/testCode/GusanilloFillTest.java -> testMarkAffected() & testUnmarkAffected

```
    private void markAffectedPositions(Square[][] matrix, int row, int col) {
        if (row >= 0 && row < size && col >= 0 && col < size)
        {
            matrix[row][col].setAvailable(1000);
            for (int i = 0; i < size; i++) {
                if (!matrix[row][i].hasQueen()) {
                    matrix[row][i].disable();
                }
            }

            for (int i = 0; i < size; i++) {
                if (!matrix[i][col].hasQueen()) {
                    matrix[i][col].disable();
                }
            }
            if (row - 1 >= 0 && col - 1 >= 0) {
                matrix[row - 1][col - 1].disable();
            }
            if (row - 1 >= 0 && col + 1 < size) {
                matrix[row - 1][col + 1].disable();
            }
            if (row + 1 < size && col - 1 >= 0) {
                matrix[row + 1][col - 1].disable();
            }
            if (row + 1 < size && col + 1 < size) {
                matrix[row + 1][col + 1].disable();
            }
        }

    }
```

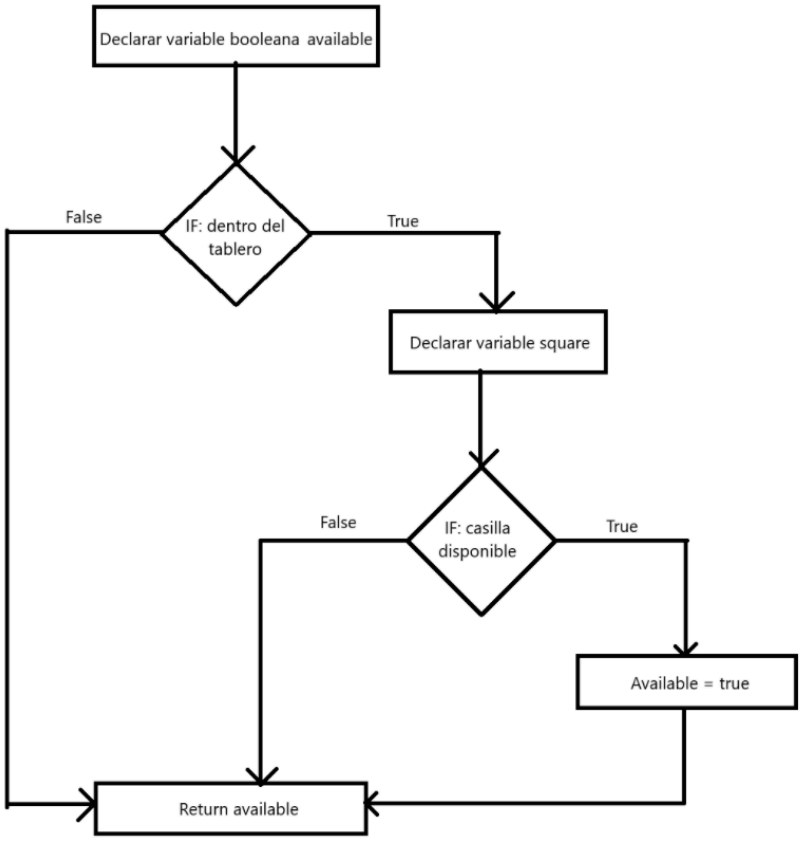
```
    private void unmarkAffectedPositions(Square[][] matrix, int row, int col) {
        if (row >= 0 && row < size && col >= 0 && col < size)
        {
            for (int i = 0; i < size; i++) {
                if (!matrix[row][i].hasQueen()) {
                    matrix[row][i].enable();
                }
            }

            for (int i = 0; i < size; i++) {
                if (!matrix[i][col].hasQueen()) {
                    matrix[i][col].enable();
                }
            }
            if (row - 1 >= 0 && col - 1 >= 0) {
                matrix[row - 1][col - 1].enable();
            }
            if (row - 1 >= 0 && col + 1 < size) {
                matrix[row - 1][col + 1].enable();
            }
            if (row + 1 < size && col - 1 >= 0) {
                matrix[row + 1][col - 1].enable();
            }
            if (row + 1 < size && col + 1 < size) {
                matrix[row + 1][col + 1].enable();
            }

            matrix[row][col].setAvailable(0);
        }
    }
```

- src/developedCode/Board.java -> isSquareAvailable()
- **Test:** src/testCode/BoardTest.java -> testIsSquareAvailable()

```java
public boolean isSquareAvailable(int x, int y) {
    boolean available = false;
    if (x >= 0 && x < size && y >= 0 && y < size) {
        Square square = squares[x][y];
        if(!square.isDisabled() && getSquaresSection(square) != null && !getSquaresSection(square).isDisabled()){
            available = true;
        }
    }
    return available;
}
```

Path coverage:

- src/developedCode/Board.java -> isSquareAvailable()
- **Test:** src/testCode/BoardTest.java -> testIsSquareAvailable()



-

Según la fórmula aristas - nodos + 2 ⟶ 7-6 +2 = 3 caminos que debemos realizar para path coverage. Estos tres caminos se representan mirando la disponibilidad de las coordenadas (-1,-1) casilla fuera del tablero; (2,3): casilla dentro del tablero pero no disponible; (0,0) casilla disponible.

Simple loop Testing:

- src/developedCode/GenerationStrategy.java -> assignColorsToQueens()
- **Test:** src/testCode/GusanilloFIllTest.java -> testAssignColorsToQueens()

```
    protected Square[][] assignColorToQueens(Square[][] queenMatrix){
        ArrayList<String> colors = new ArrayList<>(Colors.colorArray);
        for(ArrayList<Integer> coord : queensPosition) {
            int index = rng.random(0, colors.size()-1);
            queenMatrix[coord.get(0)][coord.get(1)].setColor(colors.get(index));
            colors.remove(index);

        }
        return queenMatrix;
    }
```

- src/developedCode/Board.java -> createSecrionsList()
- **Test:** src/developedCode/BoardTest.java -> testCreateSections();

```
    private void createSections(ArrayList<ArrayList<Integer>> queensPosition) {
        sections.clear();
        String color = null;
        for (ArrayList<Integer> coords : queensPosition) {
            color = squares[coords.get(0)][coords.get(1)].getColor();
            sections.add(new Section(color));
        }
    }
```

Nested loop Testing:

- src/developedCode/GusanilloFill.java -> createSection()
- **Test:** src/testCode/GusanilloFillTest.java -> testCreateSections()

```
    @Override
    protected Square[][] createSections(Square[][] coloredMatrix) {
        for(ArrayList<Integer> coords : queensPosition)
        {
            int row = coords.get(0);
            int col = coords.get(1);
            int stepsNumber = rng.random(0, size);
            String color = coloredMatrix[row][col].getColor();
            for (int i = 0; i < stepsNumber; i++)
            {

                ArrayList<ArrayList<Integer>> steps = new ArrayList<>();

                if (row-1 >= 0 && !coloredMatrix[row-1][col].hasQueen()) {
                    steps.add(new ArrayList<>(Arrays.asList(-1, 0)));
                }

                if (col+1 < size && !coloredMatrix[row][col+1].hasQueen()) {
                    steps.add(new ArrayList<>(Arrays.asList(0, 1)));
                }
                if (row+1 < size && !coloredMatrix[row+1][col].hasQueen()) {
                    steps.add(new ArrayList<>(Arrays.asList(1, 0)));
                }
                if (col-1 >= 0 && !coloredMatrix[row][col-1].hasQueen()) {
                    steps.add(new ArrayList<>(Arrays.asList(0, -1)));
                }

                if (steps.isEmpty()) {
                    break;
                }

                int direction = rng.random(0, steps.size()-1);

                row = row + steps.get(direction).get(0);
                col = col + steps.get(direction).get(1);
                coloredMatrix[row][col].setColor(color);
            }
        }
        return coloredMatrix;
    }

}
```

Mock Objects:

- src/testClasses/MockObjects/MockRNG.java → mock propio que se usa en → src/testCode/GusanilloFillTest.java en la funcion testGenerateQueen()

- src/testClasses/MockObjects/ MockGame.java → mock propio que se usa en → src/testCode/QueenTest.java en la funcion testDisableAndEnableSquares()

- src/testCode/GameTest.java en la función testPlaceOrRemoveQueen() → mock de mockito (mockBoard)

- src/testCode/BoardTest.java en la funcion testGenerateBoard() → mock del mockito (mockGenStat)