

SE 461 Software Testing and Quality

Group Project: JUnit and Test Coverage Criteria

Due on Sunday, May 5, 11:59PM.

JJ Javier

Dheeraj Singavarapu

Develop JUnit test cases to test a Snake video game written in Java. The source code is included in the Snake-Testing.zip file. Download the zip file, unzip it, and import the project into your Eclipse workspace.

There will be 11 groups in the class, and each group should have 2 members.

Your team must use at least one of the test coverage criteria that we have learned in SE 461 to develop JUnit test cases. This is similar to Question 1 of Assignment 3, except that we have a code base (instead of a single method) consisting of six classes to test.

Use JaCoCo (<https://www.jacoco.org/>), a Java code coverage tool, to generate a code coverage report, such as lines coverage, methods coverage, and branches coverage.

Write a project report that covers the following aspects:

1. **Test design:** use at least three examples to explain how your team developed JUnit test cases to satisfy the coverage criterion/criteria that your team used. The explanation should be specific and similar to your answers to Question 1 of Assignment 3 (e.g., include graphs, TRs, test cases, and test paths). For each test case, explain what your input is and what your expected output is.

EXAMPLE 1:

So for the first example I will showcase how we wrote a J Unit test for the drawtile() method using the edge coverage criterion. So to start off what we did was draw the CFG to be able to accurately see all the paths of the method as seen below:


```
@Test
public void testDrawTile_Path_1_2_9() {
    BufferedImage image = new BufferedImage(BoardPanel.COL_COUNT *
BoardPanel.TILE_SIZE,
        BoardPanel.ROW_COUNT * BoardPanel.TILE_SIZE, BufferedImage.TYPE_INT_ARGB);
    Graphics graphics = image.getGraphics();
    boardPanel.drawTile(0, 0, TileType.Fruit, graphics);
    assertTrue(isTileColorPresent(image, Color.RED));
}
```

Input:

Tile type: TileType.Fruit

Tile position: (0, 0)

Expected Output:

The color red (Color.RED) should be seen in the rendered image of the tile.

```
@Test
public void testDrawTile_Path_1_3_9() {
    BufferedImage image = new BufferedImage(BoardPanel.COL_COUNT *
BoardPanel.TILE_SIZE,
        BoardPanel.ROW_COUNT * BoardPanel.TILE_SIZE, BufferedImage.TYPE_INT_ARGB);
    Graphics graphics = image.getGraphics();
    boardPanel.drawTile(0, 0, TileType.SnakeBody, graphics);
    assertTrue(isTileColorPresent(image, Color.GREEN));
}
```

Input:

Tile type: TileType.SnakeBody

Tile position: (0, 0)

Expected Output:

The color green (Color.GREEN) should be seen in the rendered image of the tile.

```

@Test
public void testDrawTile_Path_1_4_5_9() {
    game.setDirection(Direction.North);
    BufferedImage image = new BufferedImage(BoardPanel.COL_COUNT *
BoardPanel.TILE_SIZE,
        BoardPanel.ROW_COUNT * BoardPanel.TILE_SIZE, BufferedImage.TYPE_INT_ARGB);
    Graphics graphics = image.getGraphics();
    boardPanel.drawTile(0, 0, TileType.SnakeHead, graphics);
    assertTrue(isTileColorPresent(image, Color.GREEN));
    assertTrue(isTileColorPresent(image, Color.BLACK));
}

```

Input:

Tile type: `TileType.SnakeHead`

Tile position: (0, 0)

Snake direction: `Direction.North`

Expected Output:

The color green (`Color.GREEN`) should be seen in the rendered image of the tile.

The color black (`Color.BLACK`) should be seen in the rendered image of the tile (which represents the eyes of the snake head facing North).

```

@Test
public void testDrawTile_Path_1_4_6_9() {
    game.setDirection(Direction.South);
    BufferedImage image = new BufferedImage(BoardPanel.COL_COUNT *
BoardPanel.TILE_SIZE,
        BoardPanel.ROW_COUNT * BoardPanel.TILE_SIZE, BufferedImage.TYPE_INT_ARGB);
    Graphics graphics = image.getGraphics();
    boardPanel.drawTile(0, 0, TileType.SnakeHead, graphics);
    assertTrue(isTileColorPresent(image, Color.GREEN));
    assertTrue(isTileColorPresent(image, Color.BLACK));
}

```

Input:

Tile type: `TileType.SnakeHead`

Tile position: (0, 0)

Snake direction: `Direction.South`

Expected Output:

The color green (Color.GREEN) should be present in the rendered image of the tile.
The color black (Color.BLACK) should be present in the rendered image of the tile
(representing the eyes of the snake head facing South).

```
@Test
public void testDrawTile_Path_1_4_7_9() {
    game.setDirection(Direction.West);
    BufferedImage image = new BufferedImage(BoardPanel.COL_COUNT *
BoardPanel.TILE_SIZE,
        BoardPanel.ROW_COUNT * BoardPanel.TILE_SIZE, BufferedImage.TYPE_INT_ARGB);
    Graphics graphics = image.getGraphics();
    boardPanel.drawTile(0, 0, TileType.SnakeHead, graphics);
    assertTrue(isTileColorPresent(image, Color.GREEN));
    assertTrue(isTileColorPresent(image, Color.BLACK));
}
```

Input:

Tile type: TileType.SnakeHead

Tile position: (0, 0)

Snake direction: Direction.West

Expected Output:

The color green (Color.GREEN) should be present in the rendered image of the tile.
The color black (Color.BLACK) should be present in the rendered image of the tile
(representing the eyes of the snake head facing West).

```

@Test
public void testDrawTile_Path_1_4_8_9() {
    game.setDirection(Direction.East);
    BufferedImage image = new BufferedImage(BoardPanel.COL_COUNT *
BoardPanel.TILE_SIZE,
        BoardPanel.ROW_COUNT * BoardPanel.TILE_SIZE, BufferedImage.TYPE_INT_ARGB);
    Graphics graphics = image.getGraphics();
    boardPanel.drawTile(0, 0, TileType.SnakeHead, graphics);
    assertTrue(isTileColorPresent(image, Color.GREEN));
    assertTrue(isTileColorPresent(image, Color.BLACK));
}

```

Input:

Tile type: `TileType.SnakeHead`

Tile position: (0, 0)

Snake direction: `Direction.East`

Expected Output:

The color green (`Color.GREEN`) should be present in the rendered image of the tile.

The color black (`Color.BLACK`) should be present in the rendered image of the tile (representing the eyes of the snake head facing East).

```

private boolean isTileColorPresent(BufferedImage image, Color color) {
    for (int x = 0; x < image.getWidth(); x++) {
        for (int y = 0; y < image.getHeight(); y++) {
            if (image.getRGB(x, y) == color.getRGB()) {
                return true;
            }
        }
    }
    return false;
}

```

This above method is used to check if the expected colors are present in the rendered image of the tile. It iterates over each pixel of the image and compares the RGB values with the expected color's RGB values.

EXAMPLE 2:

The next example we will be showcasing a test case within the snakegametest.java. This test case accounts for a single test path within the snakegame(). Below we will showcase the snakegame() CFG along with the corresponding TR's and Test paths. We will also present the path that is covered by the specific test case below:

```
@Test
    public void testGamePauseGameOver() throws IOException, InterruptedException,
AWTException {
    SnakeGame snakeGame = new SnakeGame();

        Thread gameThread = new Thread(() -> {
    snakeGame.startGame();
    });
    gameThread.start();

    Thread.sleep(1000);

    Robot robot = new Robot();

    robot.keyPress(KeyEvent.VK_ENTER);
    robot.keyRelease(KeyEvent.VK_ENTER);

    robot.delay(1000);
    Thread.sleep(1000);

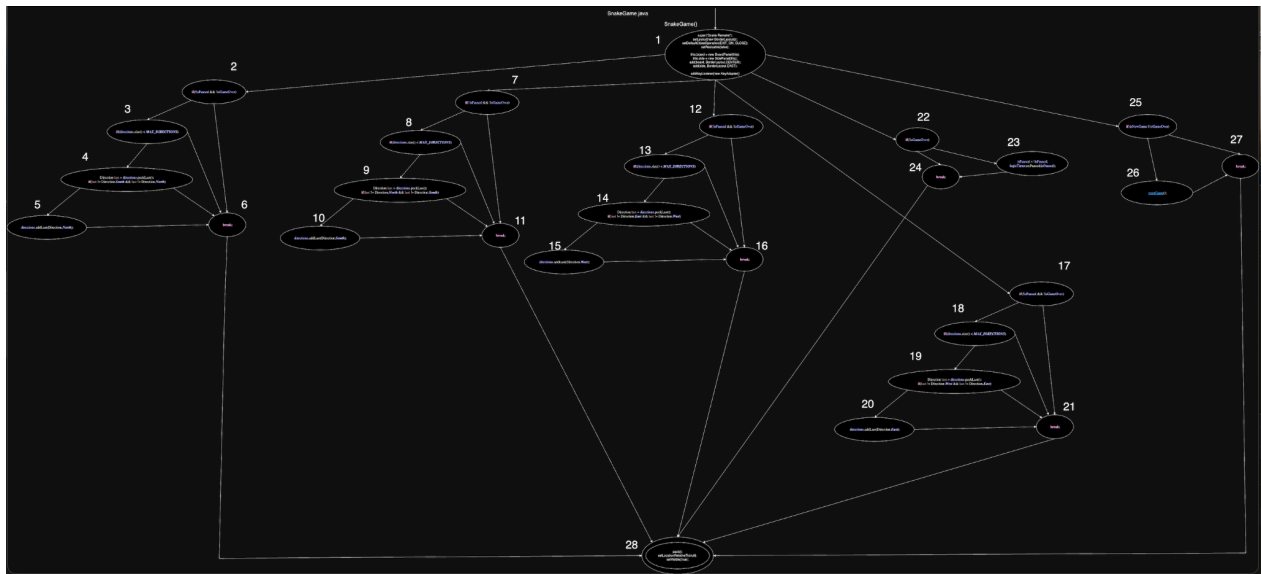
    robot.keyPress(KeyEvent.VK_P);
    robot.keyRelease(KeyEvent.VK_P);

    assertEquals(snakeGame.isPaused, false);

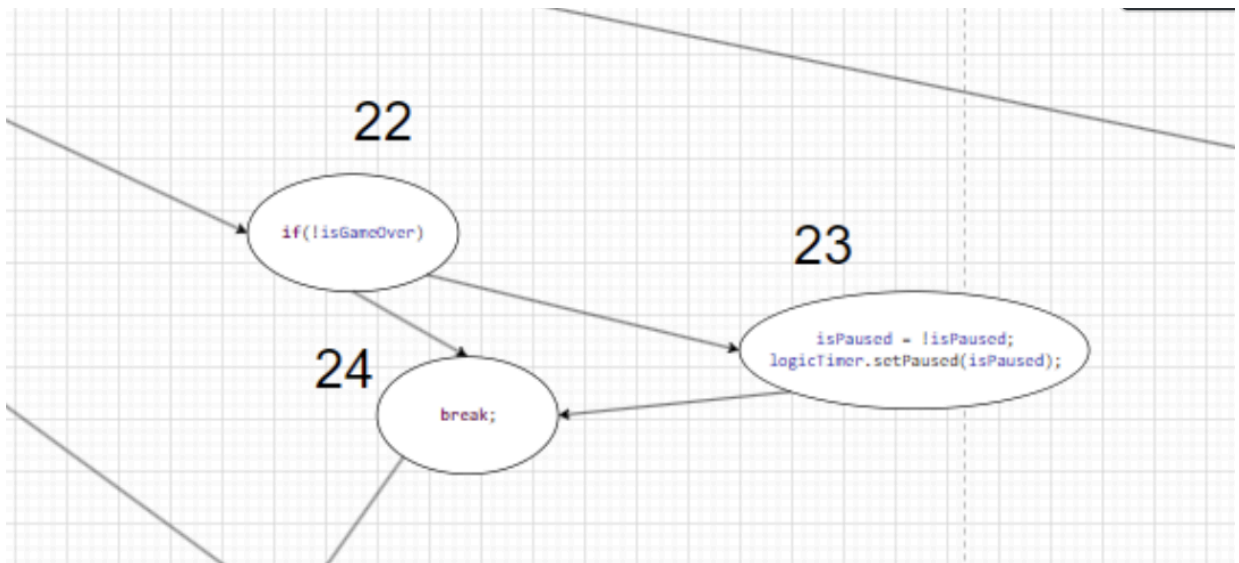
    }
```

CFG BELOW

Here is the CFG:



The CFG is the entire CFG for the snakegame() method. However the test case I provided only traverses these nodes in the CFG below:



Here are the TR's for snakegame() as a whole:

TR = {(1,2), (2,3), (3,4),(4,5), (5,6), (2,6), (3,6), (4,6), (6,28), (1,7), (7,8), (8,9),(9,10), (10,11), (7,11), (8,11), (9,11), (11,28), (1,12), (12,13), (13,14),(14,15), (15,16), (12,16), (13,16), (14,16), (16,28), (1,17), (17,18), (18,19),(19,20), (20,21), (17,21), (18,11), (19,11), (21,28), (1,22), (22,23), (23,24), (24,28), (22,24), (1,25), (25,26), (26,27), (27,28), (25,27)}

Here are the following test paths for snakegame() using edge coverage :

1,2,3,4,5,6,28
1,2,3,4,6,28
1,2,3,6,28
1,2,6,28
1,7,8,9,10,11,28
1,7,8,9,11,28
1,7,8,11,28
1,7,11,28
1,12,13,14,15,16,28
1,12,13,14,16,28
1,12,13,16,28
1,12,16,28
1,17,18,19,20,21,28
1,17,18,19,21,28
1,17,18,21,28
1,17,21,28
1,22,23,24,28
1,22,24,28
1,25,26,27,28
1,25,27,28

Now the J unit test case I showcased covers this test path of snake game :

1 -> 22 -> 24 -> 28(exit)

Input:

Starting the game/halving the robot click enter to start the game, and letting the snake collide with the sides, then the robot tries pressing P to make pause true, but it stays false since it cannot do it.

Expected Output:

We expect pause to stay false as it cannot do that.

EXAMPLE 3 :

The next example we will be showcasing a test case within the snakegametest.java again. This test case accounts for a single test path within the snakegame(). Below we will showcase the snakegame() CFG along with the corresponding TR's and Test paths. We will also present the path that is covered by the specific test case below:

@Test

```
public void testDirPeekLast3() throws InterruptedException, AWTException {
```

```
    SnakeGame snakeG = new SnakeGame();
```

```
    Thread gameThread = new Thread(() -> {
        snakeG.startGame();
    });
    gameThread.start();
```

```
    Robot robot = new Robot();
```

```
    robot.keyPress(KeyEvent.VK_ENTER);
    robot.keyRelease(KeyEvent.VK_ENTER);
```

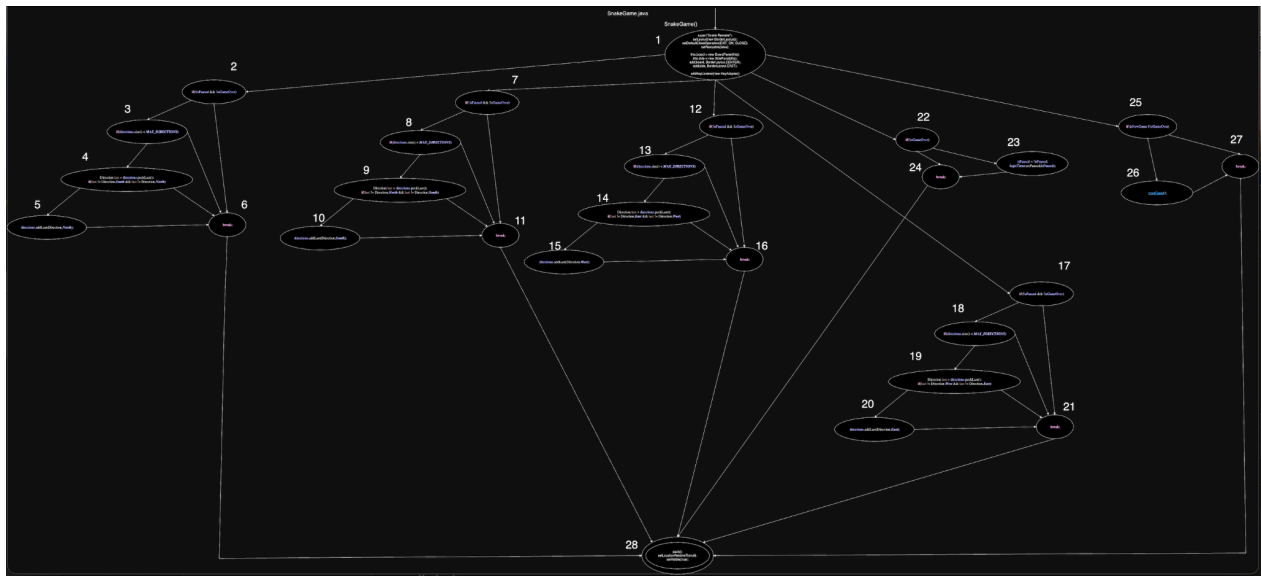
```
    robot.delay(500);
    Thread.sleep(500);
```

```
    snakeG.directions.addLast(Direction.East);
```

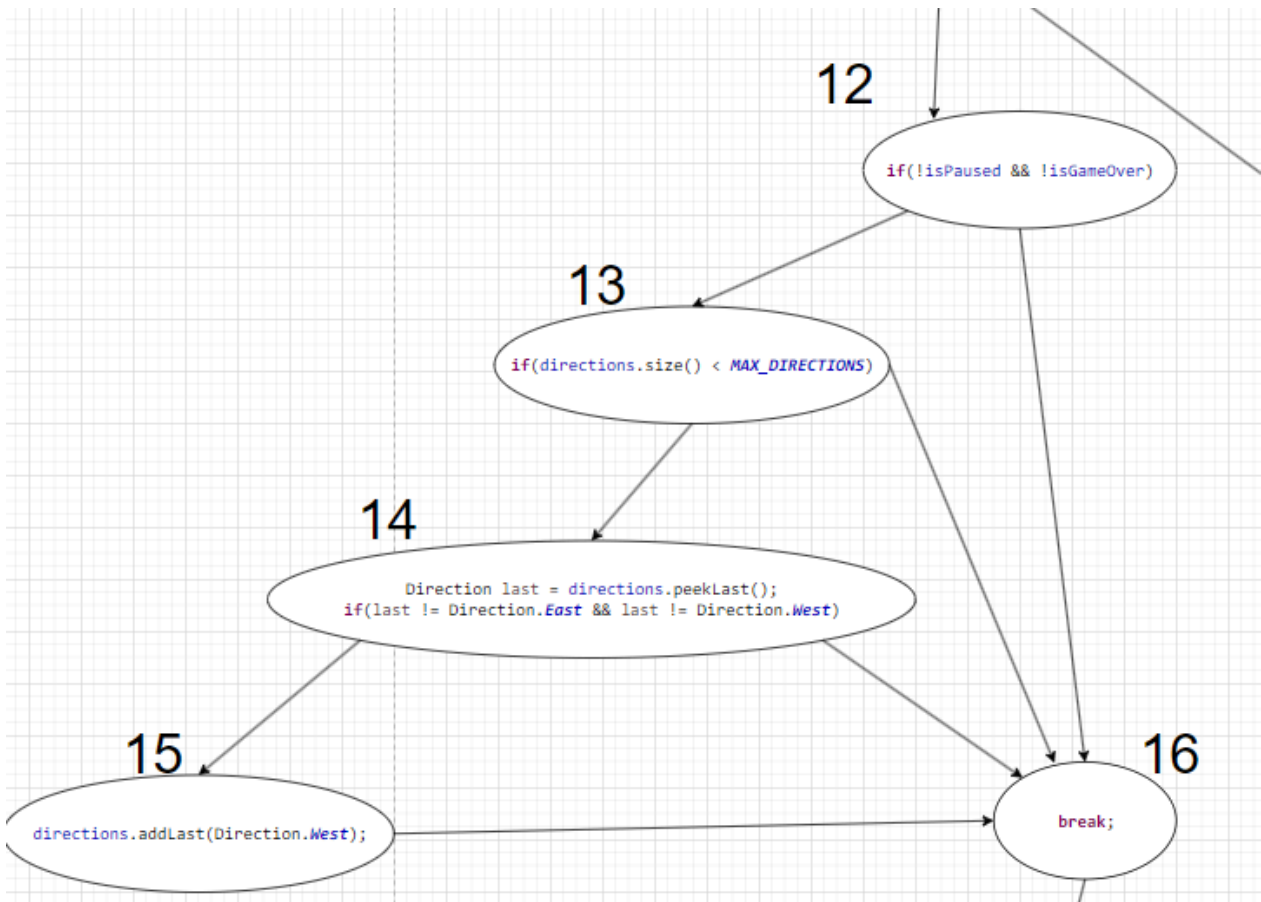
```
    robot.keyPress(KeyEvent.VK_D);
    robot.keyRelease(KeyEvent.VK_D);
    robot.delay(500);
    Thread.sleep(500);
```

```
}
```

CFG :



The CFG is the entire CFG for the snakegame() method. However the test case I provided only traverses these nodes in the CFG below:



Here are the TR's for snakegame() as a whole:

TR = {(1,2), (2,3), (3,4),(4,5), (5,6), (2,6), (3,6), (4,6), (6,28), (1,7), (7,8), (8,9),(9,10), (10,11), (7,11), (8,11), (9,11), (11,28), (1,12), (12,13), (13,14),(14,15), (15,16), (12,16), (13,16), (14,16), (16,28), (1,17), (17,18), (18,19),(19,20), (20,21), (17,21), (18,11), (19,11), (21,28), (1,22), (22,23), (23,24), (24,28), (22,24), (1,25), (25,26), (26,27), (27,28), (25,27)}

Here are the following test paths for snakegame() using edge coverage :

1,2,3,4,5,6,28
1,2,3,4,6,28
1,2,3,6,28
1,2,6,28
1,7,8,9,10,11,28
1,7,8,9,11,28
1,7,8,11,28
1,7,11,28
1,12,13,14,15,16,28
1,12,13,14,16,28
1,12,13,16,28
1,12,16,28
1,17,18,19,20,21,28
1,17,18,19,21,28
1,17,18,21,28
1,17,21,28
1,22,23,24,28
1,22,24,28
1,25,26,27,28
1,25,27,28

Now the J unit test case I showcased covers this test path of snake game :

1 -> 12 -> 13 -> 14-> 16 ->28(exit)

Input:

Creating snakegame object and starting the game, using the robot to simulate the game, but the main input is adding Direction East as the last element in the array of directions just before the robot inputs the D key as a case switch within snakegame to move the snake

Expected Output:

The expected output is getting that edge from node 14 to 16, because that checks whether or not the last direction used was either East or West, and thus because the input was Direction

East, it does not go into the if statement as true and goes directly to node 16, which is the break.

EXAMPLE 4 :

The next example we will be showcasing a test case within the snakegametest.java again. This test case accounts for a single test path within the StartGame(). Below we will showcase the startgame() CFG along with the corresponding TR's and Test paths. We will also present the path that is covered by the specific test case below:

```
@Test
public void testGameStart() throws IOException, InterruptedException, AWTException {
    SnakeGame snakeGame = new SnakeGame();

        Thread gameThread = new Thread(() -> {
            snakeGame.startGame();
        });
        gameThread.start();

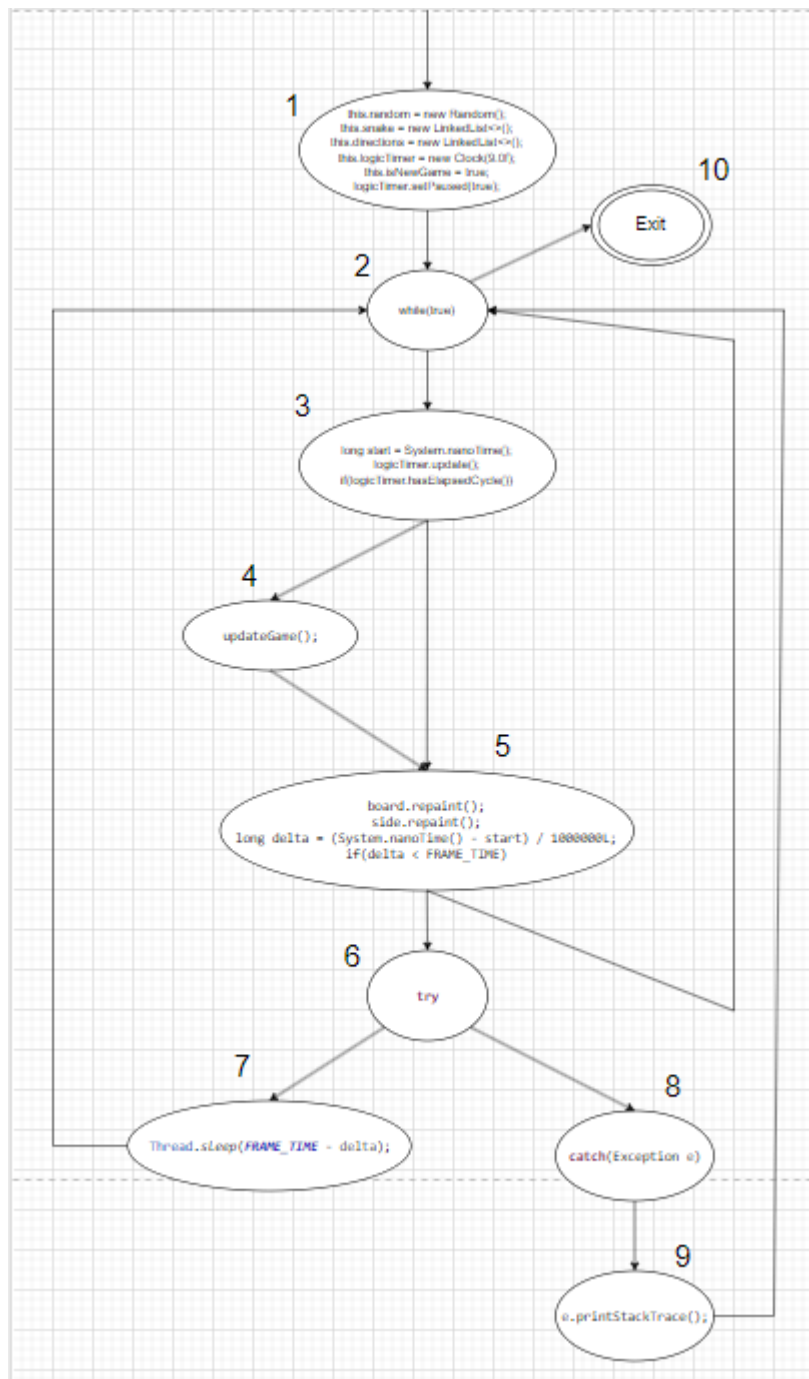
        Thread.sleep(1000);

        Robot robot = new Robot();

        robot.keyPress(KeyEvent.VK_ENTER);
        robot.keyRelease(KeyEvent.VK_ENTER);

        assertEquals(snakeGame.isNewGame, true);
}
```

CFG :



Here are the TR's for startgame() as a whole:

TR = {(1,2),(2,3),(3,4),(3,5),(4,5),(5,6),(6,7),(6,8),(8,9),(7,2),(9,2),(2,10)}

Here are the following test paths for snakegame() using edge coverage :

1,2,3,4,5,6,7,2,10

1,2,3,5,6,8,9,2,10

1,2,3,5,2,10

Now the J unit test case I showcased covers this test path of snake game :

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 2 -> 10(exit)

Input :

Creating a snakegame object in order to run startgame, but in order to do so we had to introduce multithreading. As we put startgame on a separate thread, we can start that thread separately then use the java robot to start the game by pressing enter

Expected Output:

We expected the output that the snakegame.isNewGame to be true, to which is asserted to be true as shown

EXAMPLE 5 :

The next example we will be showcasing a test case within the snakegametest.java again. This test case accounts for a single test path within the updateGame(). Below we will showcase the updateGame() CFG along with the corresponding TR's and Test paths. We will also present the path that is covered by the specific test case below:

@Test

```
public void testFruit() throws InterruptedException, AWTException {
```

```
    SnakeGame sg = new SnakeGame();
```

```
    sg.random = new Random();
```

```
    sg.snake = new LinkedList<>();
```

```
    sg.directions = new LinkedList<>();
```

```
    sg.logicTimer = new Clock(9.0f);
```

```
    sg.isNewGame = true;
```

```
    sg.snake.clear();
```

```
    Point head = new Point(5, 5);
```

```
    sg.snake.add(head);
```

```
    sg.board.setTile(head.x, head.y, TileType.SnakeHead);
```

```
    sg.board.setTile(head.x + 1, head.y, TileType.Fruit);
```

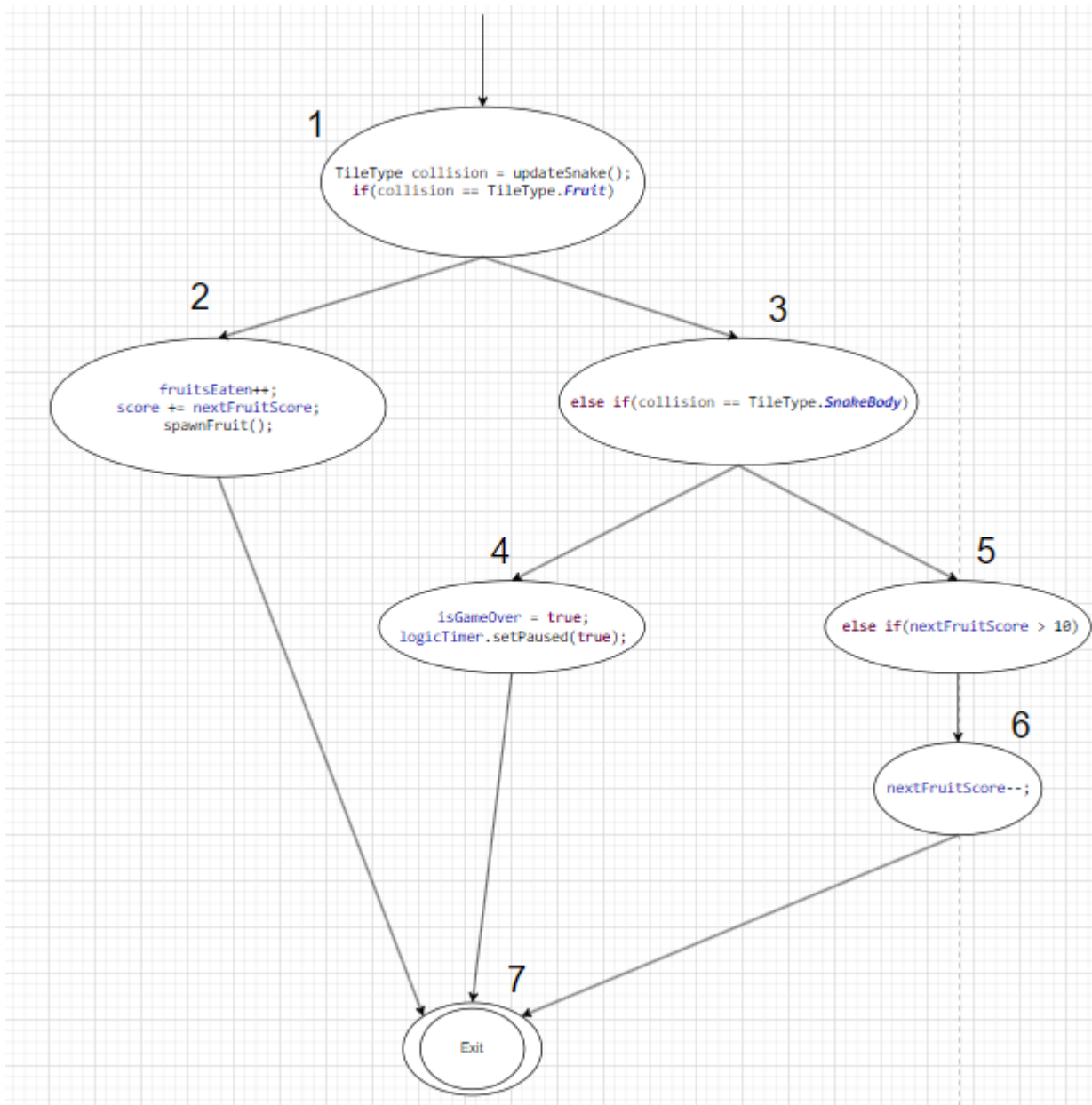
```
    sg.directions.add(Direction.East);
```

```
    sg.updateGame();
```

```
    assertEquals("Fruits eaten should increment", 1, sg.fruitsEaten);
```

}

CFG:



Here are the TR's for startgame() as a whole:

TR = {(1,2), (2,7), (1,3),(3,4),(3,5),(4,7),(5,6),(6,7)}

Here are the following test paths for snakegame() using edge coverage :

1,2,7

1,3,4,7

1,3,5,6,7

Now the J unit test case I showcased covers this test path of snake game :

1 -> 2 -> 7 (exit)

Input:

Creating a snakegame object, initiating it manually through setting a new random, snake, directions, logictimer, and setting a newgame. From there we cleared the snake in order to and created a new Point for the head to be, added it to the snake, and set the tile as a Snake Head Tiletype and then setted a tile 1 position to the right of the snake head as a Fruit TileType, set the direction of the snake head to east, the updated the game so the head collides with the fruit.

Expected output:

The expected output was the the collision type of the snake to be with the fruit and that to trigger the if statement within updateGame() so that the fruit counter goes up one, score += nextFruitScore, and it'll call spawnfruit, to which it does as shown.

EXAMPLE 6 :

The next example we will be showcasing a test case within the BoardPanel.java. This test case accounts for a single test path within the paintComponent(). Below we will showcase the paintComponent() CFG along with the corresponding TR's and Test paths. We will also present the path that is covered by the specific test case below:

@Test

```
public void testPaintComponent() {
    BufferedImage image = new BufferedImage(BoardPanel.COL_COUNT *
BoardPanel.TILE_SIZE,
    BoardPanel.ROW_COUNT * BoardPanel.TILE_SIZE, BufferedImage.TYPE_INT_ARGB);
    Graphics graphics = image.getGraphics();

    SnakeGame sg = new SnakeGame();
    sg.isGameOver = true;
    sg.isPaused = false;
    sg.isNewGame = false;

    sg.board.paintComponent(graphics);

    assertNotNull(sg.board);
}
```

```
sg.board.clearBoard();
```

```
sg.isNewGame = true;  
sg.isGameOver = false;  
sg.isPaused = false;
```

```
sg.board.paintComponent(graphics);  
assertNotNull(sg.board);
```

```
sg.board.clearBoard();
```

```
sg.isNewGame = false;  
sg.isGameOver = false;  
sg.isPaused = true;
```

```
sg.board.paintComponent(graphics);  
assertNotNull(sg.board);
```

```
sg.board.clearBoard();
```

```
sg.isNewGame = false;  
sg.isGameOver = false;  
sg.isPaused = false;
```

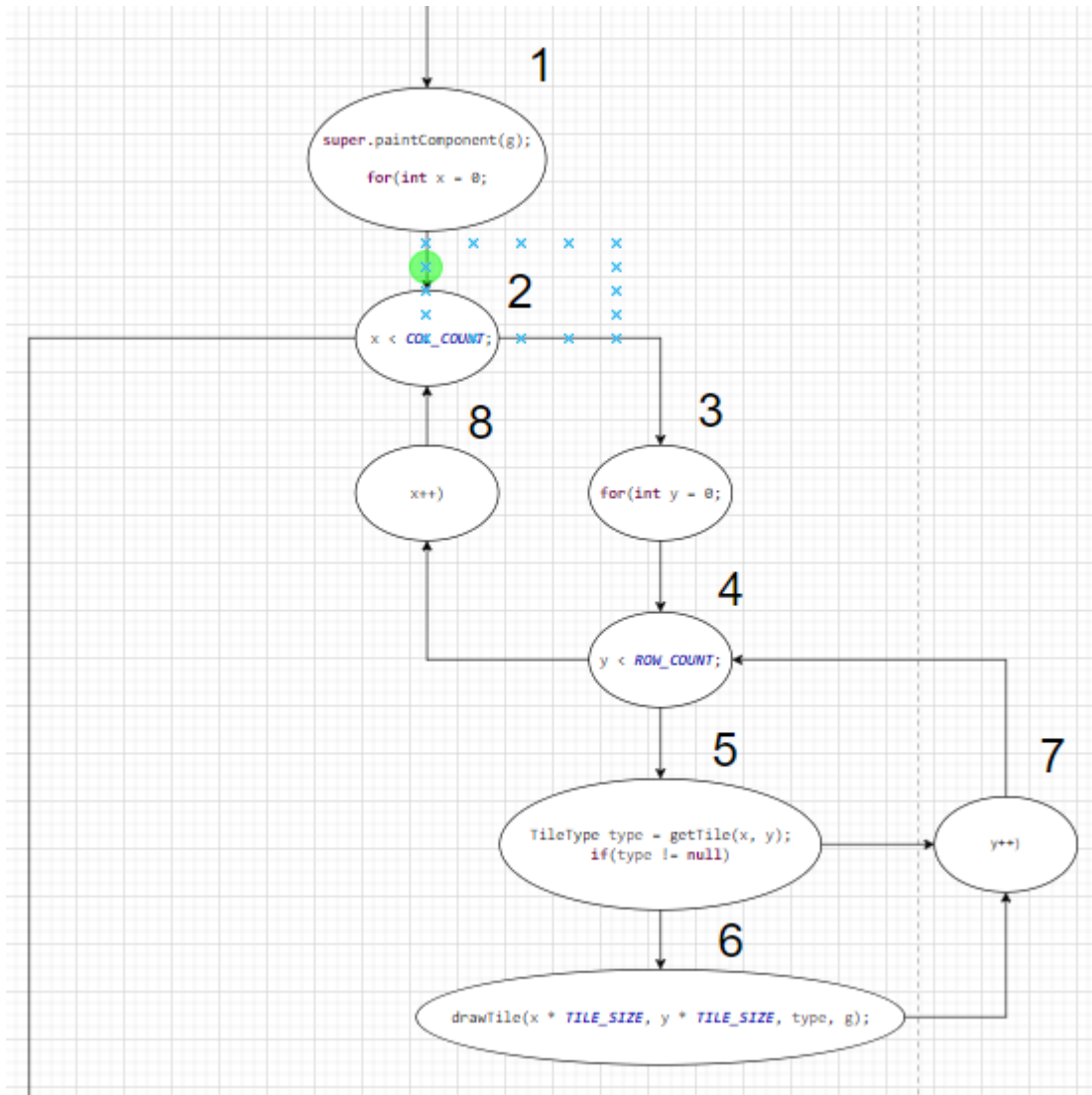
```
sg.board.paintComponent(graphics);  
assertNotNull(sg.board);
```

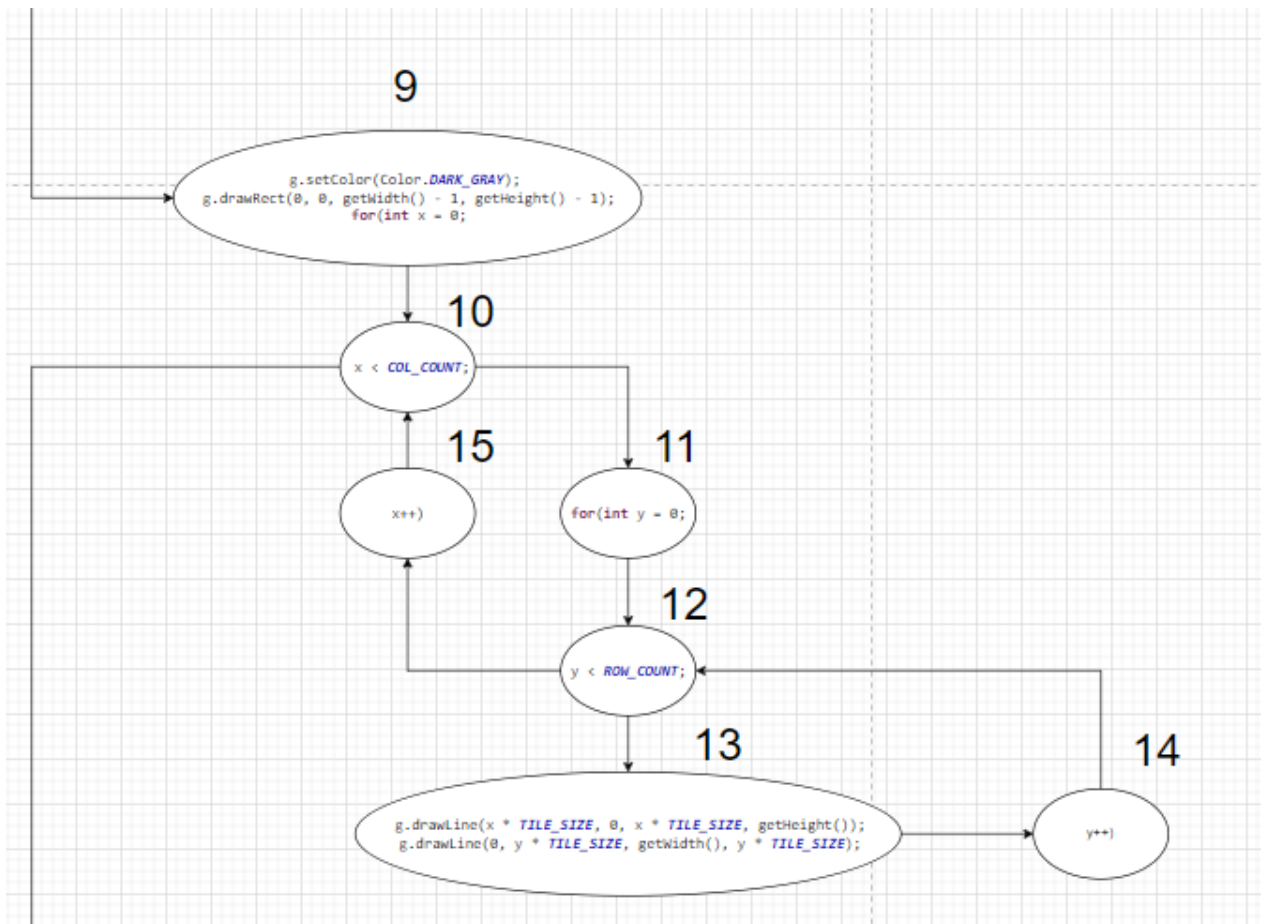
```
sg.random = new Random();  
sg.snake = new LinkedList<>();  
sg.directions = new LinkedList<>();  
sg.logicTimer = new Clock(9.0f);  
sg.isNewGame = true;
```

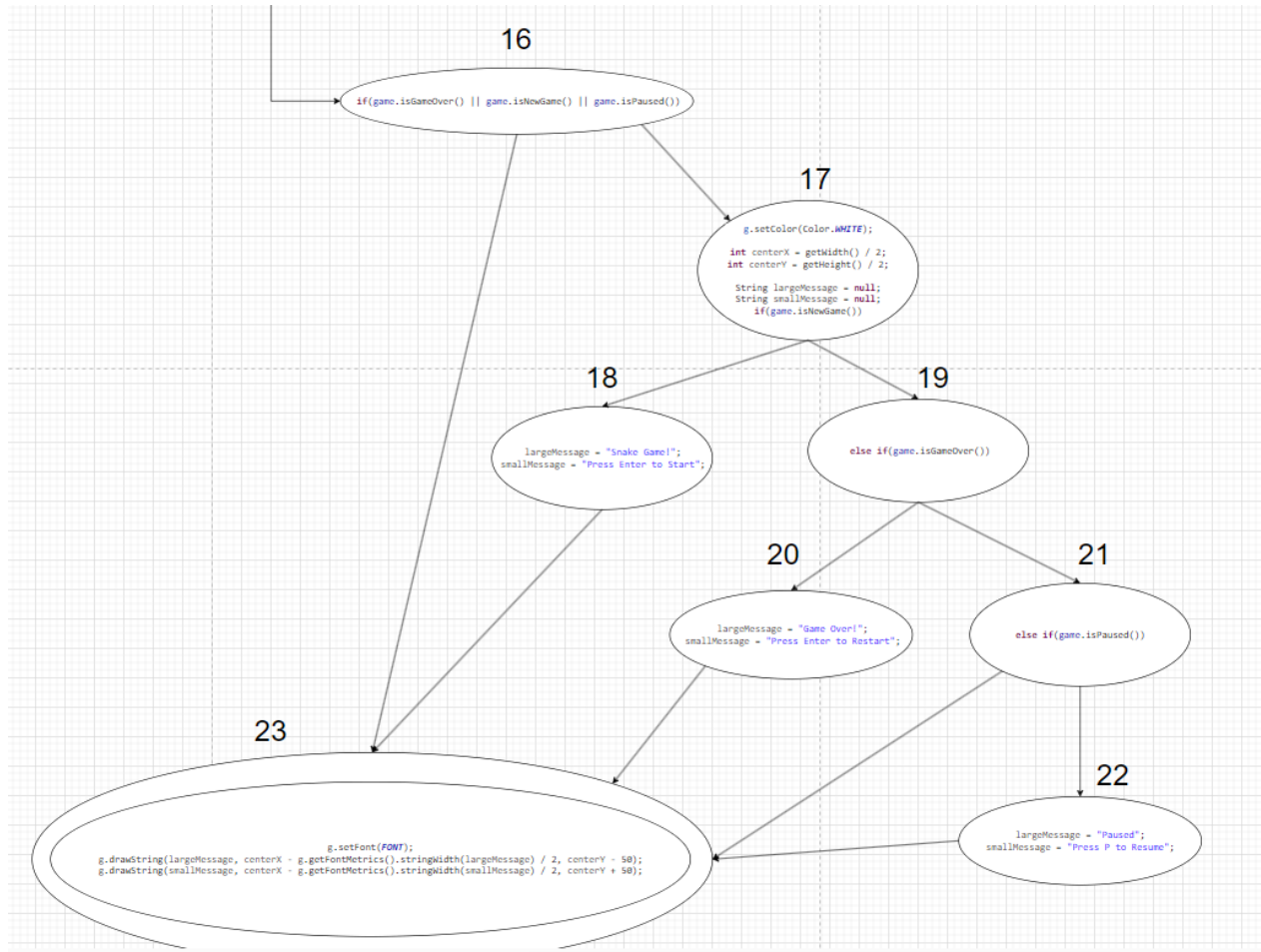
```
sg.snake.clear();  
sg.board.clearBoard();  
Point head = new Point(5, 5);  
sg.snake.add(head);  
sg.board.setTile(head.x, head.y, TileType.SnakeHead);  
sg.board.setTile(head.x + 1, head.y, TileType.Fruit);
```

```
}
```

CFG:







Here are the TR's for startgame() as a whole:

TR = {(1,2),(2,3),(3,4), (4,5),(5,6),(6,7),(5,7),(7,4),(4,8),(8,2),(2,9),
(9,10),(10,11),(11,12),(12,13),(13,14),(14,12),(12,15),(15,10),(10,16),(16,17),(16,23),(17,18),(17,
19),(18,23),(19,20),(19,21),(20,23),(21,22),(21,23),(22,23)}

Here are the following test paths for snakegame() using edge coverage :

1,2,3,4,5,6,7,4,8,2,9,10,11,12,13,14,12,15,10,16,17,19,21,22,23

1,2,3,4,5,7,4,8,2,9,10,16,17,18,23

1,2,3,4,5,7,4,8,2,9,10,16,17,19,20,23

1,2,3,4,5,7,4,8,2,9,10,16,17,19,21,23

Now the J unit test case I showcased covers this test path of snake game :

This covers all the edges except for one as the if statement checks all three isNewGame, isGameOver, and isPaused. This test case is like a combination of 3 different tests into one to cover all of paintcomponent. It's coded to get into the for loops and different edges of them and the different edges of the if statements

Input:

For each test within this test case, the input includes a new snakegame object, and a graphics object based on an BufferedImage based on the board of the snakegame. From there we coded to set isNewGame false/true, isGameOver false/true, and isPaused to false/true in order to get through all the different edges of the if statement

Expected Output:

The expected output was to reach those different outputs of the if statement which came out to happen as when it reached if(game.isNewGame()) then the output to be, largeMessage = "Snake Game!"; smallMessage = "Press Enter to Start";, then if it reached else if(game.isGameOver()), then the output to be, largeMessage = "Game Over!"; smallMessage = "Press Enter to Restart";, and finally if it reached, else if(game.isPaused()), then the output is to be largeMessage = "Paused"; smallMessage = "Press P to Resume";, to which all 3 came to be true and happened as the test covers those edges.





2. **Test results and analysis:** what problem(s) did you find in the code? For each problem, further explain how you found it (e.g., using which test case). What is your JaCoCo code coverage score? How did your team improve your coverage score? Include screenshots of JaCoCo test reports.

A problem we found in the code was that whenever the Tiletype head of the snake collided with its final snake body type at the end, considering it to be its tail, it wouldn't end the game like it should, but rather it went through it continuing the game as normal. This however wasn't discovered through the tests, but just manual gameplay.

Our overall Jacoco score was 99.0% of the provided source code. Our team improved our coverage score by first testing the big methods properly, going by creating tests to reach the edges we have created based on the CFGs mentioned earlier and the TRs we have made for EC in correspondence to the CFGs. And after making those we ran our code and saw what other edges weren't covered because of the Jacoco highlights of either green, yellow, and red. From there we were able to figure out what needed to be done and coded in order to reach those edges and turn them from either yellow to green or red to green. The main issue we would have when it came to Jacoco score coverage was the different branches that were possible from if statements and if the if statements had multiple arguments, we had to make sure we covered each possible argument within the statement so that those edges were reached appropriately.

AllTests (May 5, 2024 6:39:41 PM) > SnakeRemake

SnakeRemake

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
Test		96%		75%	22 69	40 578	19 63	1 5
src		99%		94%	9 114	5 278	1 40	0 7
Total	80 of 3,248	97%	11 of 147	92%	31 183	45 856	20 103	1 12

org.psnbtech

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
SnakeGame		97%		90%	5 36	5 102	1 14	0 1
SnakeGame.new KeyAdapter() {...}		98%		98%	1 32	0 33	0 2	0 1
BoardPanel		100%		90%	3 27	0 73	0 8	0 1
SidePanel		100%		n/a	0 3	0 28	0 3	0 1
Clock		100%		100%	0 14	0 30	0 11	0 1
Direction		100%		n/a	0 1	0 9	0 1	0 1
TileType		100%		n/a	0 1	0 4	0 1	0 1
Total	13 of 1,332	99%	8 of 135	94%	9 114	5 278	1 40	0 7

- 3. Reflection and lessons learned:** describe your experience of designing JUnit test cases in this project and how your test coverage criteria helped (or hindered?) your test design. Putting everything together, what worked, what did not? What are your overall thoughts about software testing, test coverage criteria, or Junit?

J unit test cases proved to be very difficult. We started off by disregarding the coverage criterion and just decided to test things method by method. This of course proved to be very inefficient and difficult. After discussing with the professor about how to correctly go about this testing we came to the conclusion that we need to first start by drawing CFGs for our methods and then choose a coverage criterion to come up with correct test paths to traverse the CFGS. After we started coming up with correct paths and using a specific coverage criterion we were actually able to much more efficiently test each of the classes within the snake game program. With the coverage criteria we were able to get a much higher percentage of JaCoCo coverage as well as correctly traverse the test paths using the coverage. What worked is the use of a coverage criterion to write J unit test cases instead of just free handing the test cases. What didn't work was trying to cover every single method individually. We learned that if done correctly you would only need to test a certain amount of the methods present to get coverage throughout the system as many methods rely on one another. All in all I think software testing is a very important part of the Engineering world. Obviously without testing we would have many catastrophes and issues throughout society. I think test coverage criteria is very important as it is a way to ensure you are efficiently and effectively covering the right test cases. It prevents you from doing both excessive work as well as gives you a correct method of testing that covers what is necessary. J unit is the first method of testing I learned so I will always remember using it however I don't think it is the best way of testing. I want to learn about other ways of testing and see if they are easier to use than J unit is.

Your project will be graded based on the following aspects.

1. Code coverage score that JaCoCo reported.
2. The number and quality of JUnit test cases that you have developed.
3. The completeness and quality of your project report.
4. The problems that you have found in the Snake game code.

5. Project presentation as described below.

At the end, each team should submit two files to Cougar Course: (1) project report and (2) the zipped project file that includes your test cases for the Snake application.

Additionally, each team will give an in-class presentation about their project. The presentation counts 20% towards your project grade.