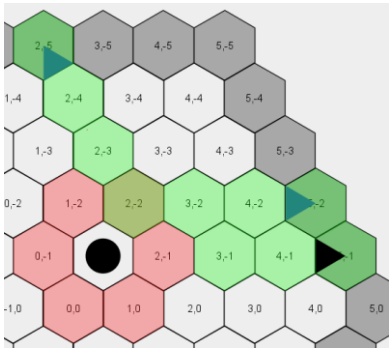
# Introduction

This sprint was focused on implementing game modes, which went very well. Game modes were fully implemented and even some new additions to the UI were added such as the finishing screen, and rules buttons in main menu.

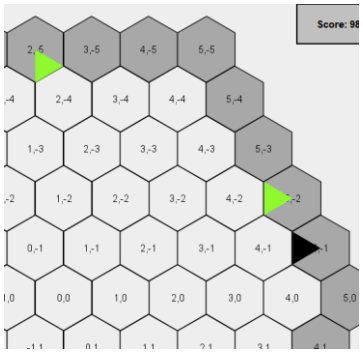
## Ray markers:

The ray markers are designed as triangles that appear along the border at the point from which the ray is sent (by mouseClick). This is and alternative to the circlular markers used in the square version which allows users to place multiple markers in a single border hexagon. The triangles snap to the nearest side clearly displaying entry and exit point/ direction making it the best option for our design.

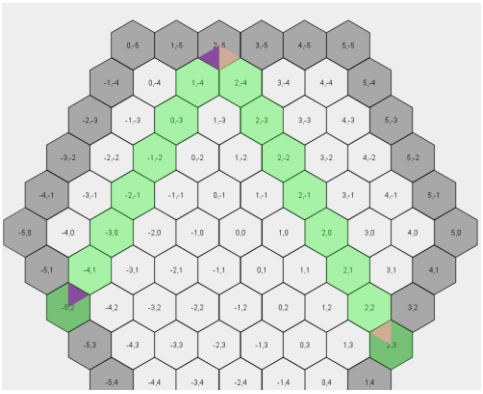
*Markers within sandbox game mode*



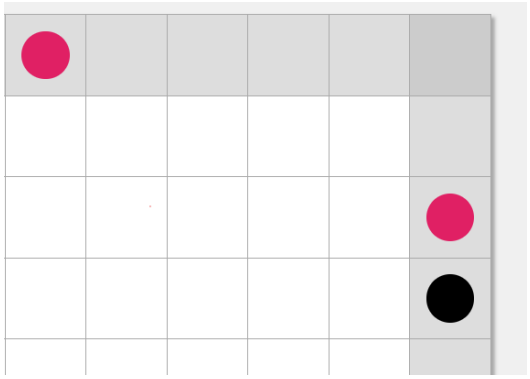
*Markers within 2 player game mode*



*Multiple markers in one bordering hexagon*



*Black box (square) markers*



## createMarker method:

A method in the hexBoard base class was created to render the ray markers.

The method takes an axial coordinate for a bordering hexagon (from which the ray is sent) and the “direction” in which the ray is sent. This is the minimum information needed to draw the markers in the correct place/direction.

```
public static final Point[] DIRECTIONS = new Point[] {//Directions array used to compute circle of influence
        new Point( x: 0,  y: 1), new Point( x: 0,  y: -1), new Point( x: -1,  y: 0),
        new Point( x: 1,  y: 0), new Point( x: -1,  y: 1), new Point( x: 1,  y: -1)
};
```

Formulas were deduced individually for each direction in which the ray marker could face. After doing so we were hoping to find some similarity so we could group cases together but we decided to leave it as is for readability.

*createMarker method*

```
294  @      protected Path2D createMarker(Point hexCenter, Point dir) {
295             Path2D path = new Path2D.Double();
296             hexCenter = axialToPixel(hexCenter.x,hexCenter.y);
297
298             //Calculate the vertices on the side from which the ray is coming (depending on direction)
299             Point vertex1 = new Point();
300             Point vertex2 = new Point();
301
302             //length from centre of a hexagon to the mid-point of any side.
303             double triangleHeight = HEX_SIZE * Math.sqrt(3) / 2;
304
305             if (dir.equals(new Point( x: 0,  y: 1))) {
306                 vertex1.x = hexCenter.x;
307                 vertex1.y = hexCenter.y + HEX_SIZE;
308                 vertex2.x = hexCenter.x + (int) triangleHeight;
309                 vertex2.y = hexCenter.y + HEX_SIZE/2;
310             } else if (dir.equals(new Point( x: 0,  y: -1))) {
311                 vertex1.x = hexCenter.x;
312                 vertex1.y = hexCenter.y - HEX_SIZE;
313                 vertex2.x = hexCenter.x - (int) triangleHeight;
314                 vertex2.y = hexCenter.y - HEX_SIZE/2;
315             } else if (dir.equals(new Point( x: -1,  y: 0))) {
316                 vertex1.x = hexCenter.x - (int) triangleHeight;
317                 vertex1.y = hexCenter.y + HEX_SIZE/2;
318                 vertex2.x = vertex1.x;
319                 vertex2.y = hexCenter.y - HEX_SIZE/2;
320             } else if (dir.equals(new Point( x: 1,  y: 0))) {
321                 vertex1.x = hexCenter.x + (int) triangleHeight;
322                 vertex1.y = hexCenter.y + HEX_SIZE/2;
323                 vertex2.x = vertex1.x;
324                 vertex2.y = hexCenter.y - HEX_SIZE/2;
325             } else if (dir.equals(new Point( x: -1,  y: 1))) {
326                 vertex1.x = hexCenter.x;
327                 vertex1.y = hexCenter.y + HEX_SIZE;
328                 vertex2.x = hexCenter.x - (int) triangleHeight;
329                 vertex2.y = hexCenter.y + HEX_SIZE/2;
330             } else if (dir.equals(new Point( x: 1,  y: -1))) {
331                 vertex1.x = hexCenter.x;
332                 vertex1.y = hexCenter.y - HEX_SIZE;
333                 vertex2.x = hexCenter.x + (int) triangleHeight;
334                 vertex2.y = hexCenter.y - HEX_SIZE/2;
335             }
336
337             path.moveTo(hexCenter.x, hexCenter.y); //Start from the center of the hexagon
338             path.lineTo(vertex1.x, vertex1.y); //First vertex on the hexagon side(from which the ray is coming from)
339             path.lineTo(vertex2.x, vertex2.y); //Second vertex on the hexagon side(from which the ray is coming from)
340             path.closePath(); //Close back to the center
341
342             return path;
343         }
```

*How it is used*

```
for(Ray ray:rays){
    if(ray.getType() == 1){
        g2d.setColor(new Color( r: 0, g: 0, b: 0));//black for absorbtion
    }
    else{
        g2d.setColor(new Color(ray.getR(),ray.getG(),ray.getB()));//other non absorbed
    }
    g2d.fill(createMarker(ray.getEntryPoint(),ray.getEntryDirection()));
    g2d.fill(createMarker(ray.getExitPoint(),new Point( x: ray.getDirection().x *-1, y: ray.getDirection().y*-1)));
}
```

Marker colours are random for all cases with the exception of the absorption case. This was done by adding r, g, b variables in the ray class. Thus each set of markers gets a random colour as they are directly linked to rays.

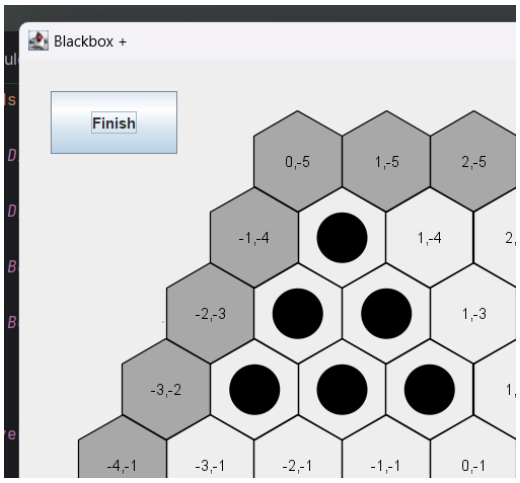
## Game Modes

Game modes were created by extending existing hexBoard class which now acts as the “sandbox” game mode used for testing and can be beneficial for users to figure out how the game mechanics work.

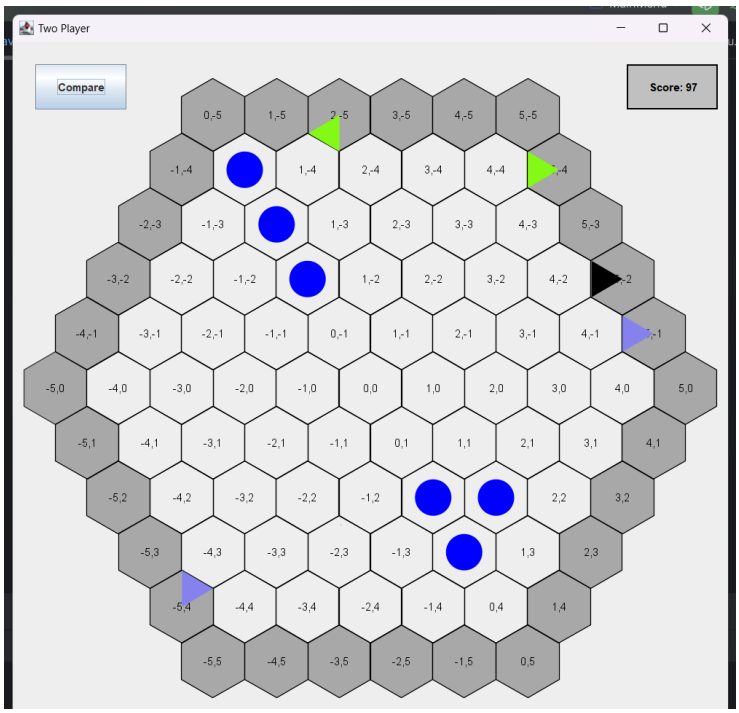
### Two player rules:

Rules:

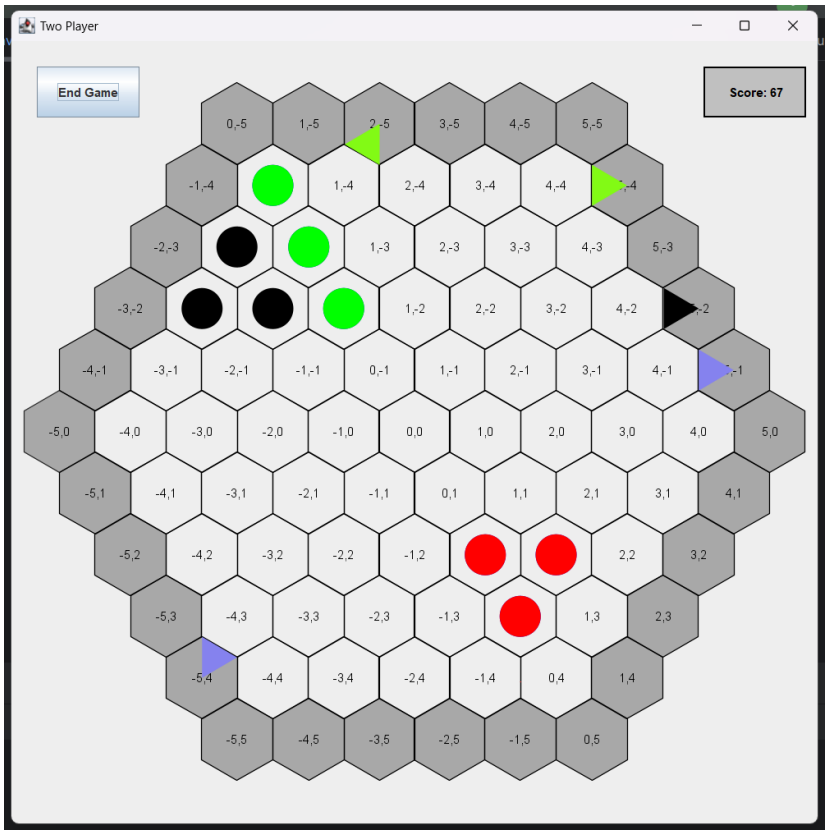
Player 1 goes first, they place atoms for the player 2 to guess. Once all 6 atoms are placed a finish button pops up, hiding the atoms for player 2.



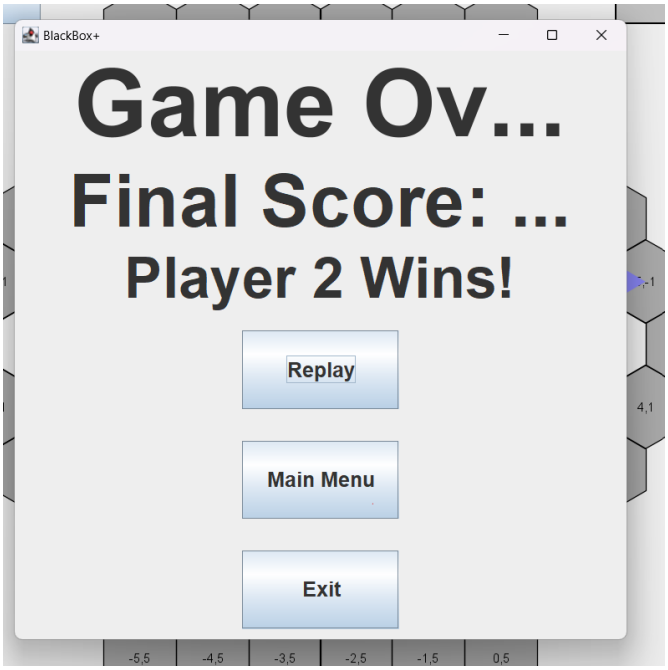

Now it is player 2s turn, player 2 attempts to guess where the atoms are by sending rays, and placing their guessed atoms(blue). Once they have placed 6 atoms a compare button pops up, this triggers the comparison part of the game, where player 1s hidden atoms are revealed, and score is generated. (-1 for every ray sent, -10 for every incorrectly guessed atom)



Guessed atoms are marked green (correctly guessed) and red (incorrectly guessed). While the not found atoms are left black. New score is displayed and option to end game is shown.




Upon ending game a pop up window shows up with some text (which will hold win details correctly in future) as well as buttons to main menu, replay current game mode or exit).



## Two Player Implementation:

The TwoPlayer class extends the HexBoard “sandbox” class.

```
public class TwoPlayer extends HexBoard {
```

Separate arrays used to store player data, and variable needed to keep track of game stage.

```
protected int currentPlayer; // 1 or 2 to indicate whose turn it is
13 usages
protected ArrayList<Atom> playerOneAtoms = new ArrayList<>();
8 usages
private ArrayList<Atom> playerTwoGuesses = new ArrayList<>();
2 usages
private ArrayList<Ray> playerTwoRays = new ArrayList<>();
```

The game is initialised in the constructor where the player is set to 1, score is initialised to 100. A Scoreboard and finish button are added to the panel but set invisible until it is player 2s turn.

The finish button remains the same but text within changes throughout gameplay (ie. Player 1 “finishes” placing atoms while player 2 is ready to “compare”, then “end game” finally).

```
public TwoPlayer() {
    currentPlayer = 1;
    score = 100;

    finishButton = new JButton( text: "Finish");
    finishButton.setBounds( x: 25,  y: 25,  width: 100,  height: 50);
    finishButton.addActionListener(e -> finishAction());

    this.setLayout(null); //Set layout to null for absolute positioning
    this.add(finishButton);
    finishButton.setVisible(false); //Initially hide the button
    drawRayPaths = false;

    //Initialize the score board label with a border and background
    scoreBoard = new JLabel( text: "Score: " + score, SwingConstants.CENTER);
    scoreBoard.setBounds( x: 675,  y: 25,  width: 100,  height: 50); // Adjust the size and position as needed
    scoreBoard.setOpaque(true); // Allow background coloring
    scoreBoard.setBackground(Color.LIGHT_GRAY); // Set background color
    scoreBoard.setForeground(Color.BLACK); // Set text color

    //Create a border
    Border border = BorderFactory.createLineBorder(Color.BLACK,  thickness: 2);
    scoreBoard.setBorder(border);

    //add to panel
    this.add(scoreBoard);
    scoreBoard.setVisible(false);
}
```

Finish action helper method is used to determine of what pressing the finish button does. Its called in the constructor. (Ie. If player 1 and button pressed change to player 2 if player 2 change to compare more and if compare mode and player 2 end game)

```
void finishAction() {
    if (currentPlayer == 1) {
        currentPlayer = 2;
        scoreBoard.setVisible(true);
        finishButton.setVisible(false);
    }
    else if(currentPlayer == 2 && compare){
        endGame =true;
    }else if (currentPlayer == 2) {
        compare = true;
    }
    repaint();

}
```

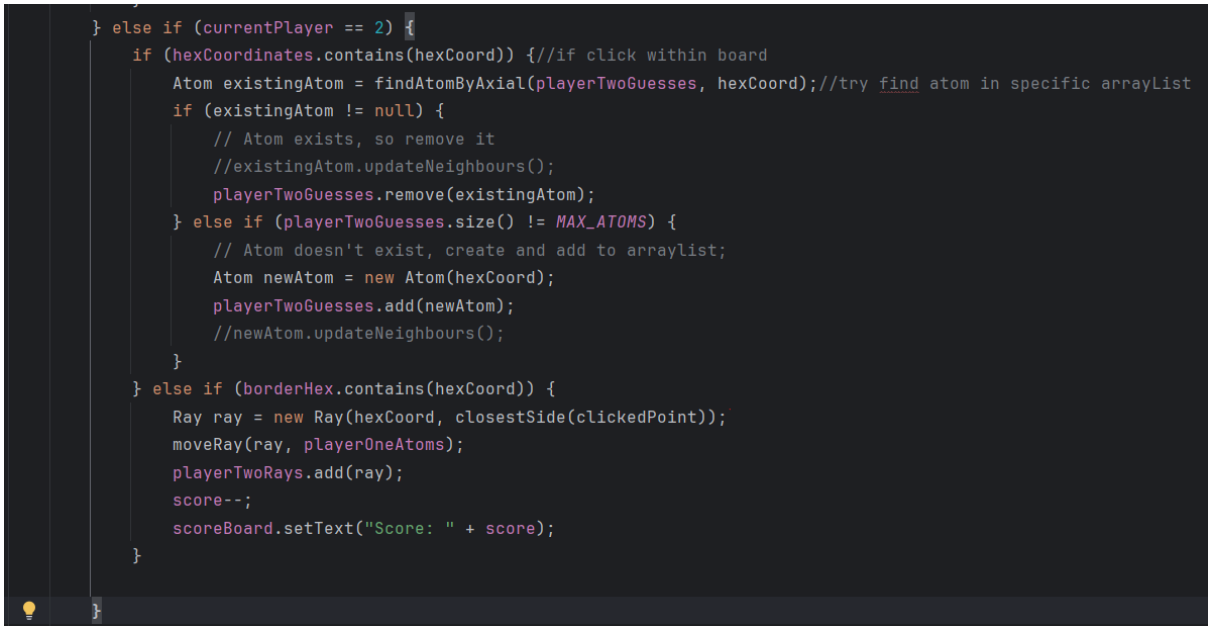
The handleMouseClick method from hexBoard is overridden to match the game modes functionality.

When it is player 1s turn for every mouse click an array containing all internal hexagon coordinates (axial) is traversed ie. Checks if a hexagon within the board is clicked. If it is, the programme then checks is there is already an atom there (contained within playerOneAtoms array) with the findAtomByAxial helper method located in hexBoard class. If there already is an atom there it is removed otherwise one is created and added to the array.

```
@Override
protected void handleMouseClick(Point hexCoord, Point clickedPoint) {
    if (currentPlayer == 1) {
        if (hexCoordinates.contains(hexCoord)) {//if click within board
            Atom existingAtom = findAtomByAxial(playerOneAtoms, hexCoord);//try find atom in specific arrayList
            if (existingAtom != null) {
                playerOneAtoms.remove(existingAtom);
            } else if (playerOneAtoms.size() != MAX_ATOMS) {
                // Atom doesn't exist, create and add to arraylist;
                Atom newAtom = new Atom(hexCoord);
                playerOneAtoms.add(newAtom);
                //newAtom.updateNeighbours();
            }

        }
```

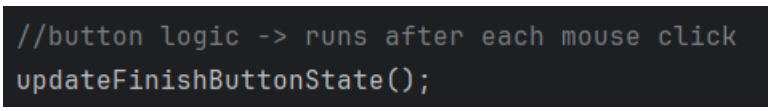
For player 2 the atom placing/removing mechanics is similar to player one with the exception that the playerTwoGuesses array is used to store the players guesses.

Ray mechanics are also incorporated here, being able to create and send a new ray if a bordering hexagon is pressed. For each send ray the score is reduced by 1 and scoreboard is updated.



```
    } else if (currentPlayer == 2) {
        if (hexCoordinates.contains(hexCoord)) {//if click within board
            Atom existingAtom = findAtomByAxial(playerTwoGuesses, hexCoord);//try find atom in specific arrayList
            if (existingAtom != null) {
                // Atom exists, so remove it
                //existingAtom.updateNeighbours();
                playerTwoGuesses.remove(existingAtom);
            } else if (playerTwoGuesses.size() != MAX_ATOMS) {
                // Atom doesn't exist, create and add to arraylist;
                Atom newAtom = new Atom(hexCoord);
                playerTwoGuesses.add(newAtom);
                //newAtom.updateNeighbours();
            }
        } else if (borderHex.contains(hexCoord)) {
            Ray ray = new Ray(hexCoord, closestSide(clickedPoint));
            moveRay(ray, playerOneAtoms);
            playerTwoRays.add(ray);
            score--;
            scoreBoard.setText("Score: " + score);
        }

    }
```

Finally within mouseClick handler the finish button state is update every time, as once 6 atoms are placed the state should change. (visible or invisible)



```
//button logic -> runs after each mouse click
updateFinishButtonState();
```

UpdateFinishButtonState(); helper method: simply changes text inside button.

```
private void updateFinishButtonState() {
    if (compare) {//if in comparison mode button used to end game
        finishButton.setText("End Game");
        finishButton.setVisible(true);
    } else if (playerOneAtoms.size() == MAX_ATOMS && currentPlayer == 1) {//if player 1 and 6 atoms placed button used to "finish" and move on
        finishButton.setText("Finish");
        finishButton.setVisible(true);
    } else if (playerTwoGuesses.size() == MAX_ATOMS && currentPlayer == 2) {//if player 2 placed 6 atoms button used to "compare"
        finishButton.setText("Compare");
        finishButton.setVisible(true);
    } else {
        finishButton.setVisible(false);
    }
```

PaintComponent is overridden for the player taking hexBoards method using super, this allows the board to be drawn.

If current player is 1 the atoms are drawn by traversing the playerOneAtoms array.

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g); // Call HexBoard's paintComponent to draw the base layer

    Graphics2D g2d = (Graphics2D) g;
    if (currentPlayer == 1) {
        //draw atom (oval)
        for (Atom atom : playerOneAtoms) {
            Point hex = atom.getPosition();
            Point pixelPoint = axialToPixel(hex.x, hex.y); // Convert axial back to pixel for drawing
            g2d.fillOval( x: pixelPoint.x - HEX_SIZE / 2,  y: pixelPoint.y - HEX_SIZE / 2, HEX_SIZE, HEX_SIZE);
        }
    }
```

If the current player is player 2. It draws its atoms in blue to show it is guess atoms, and ray markers depending on the type (black if absorption case, random for other).

```
if (currentPlayer == 2) {
    //draw atom (oval)
    for (Atom atom : playerTwoGuesses) {
        g2d.setColor(Color.blue);
        Point hex = atom.getPosition();
        Point pixelPoint = axialToPixel(hex.x, hex.y); // Convert axial back to pixel for drawing
        g2d.fillOval( x: pixelPoint.x - HEX_SIZE / 2,  y: pixelPoint.y - HEX_SIZE / 2, HEX_SIZE, HEX_SIZE);
    }


    for (Ray ray : playerTwoRays) {
        if (ray.getType() == 1) {
            g2d.setColor(new Color( r: 0,  g: 0,  b: 0));//black for absorbtion
        }
        else {
            g2d.setColor(new Color(ray.getR(), ray.getG(), ray.getB()));//other non absorbed
        }
        g2d.fill(createMarker(ray.getEntryPoint(), ray.getEntryDirection()));
        g2d.fill(createMarker(ray.getExitPoint(), new Point( x: ray.getDirection().x * -1,  y: ray.getDirection().y * -1)));
    }


}
```

When the “compare” button is pressed after player 2 is satisfied with their 6 guesses, it triggers the compare Boolean to true.

The guessed correctly array is used to compare atoms and display player ones atoms not found black.

The matchFound Boolean used to check if atom was correctly guessed and displays it green or black accordingly.

```
if (compare) {
    ArrayList<Point> guessedCorrectly = new ArrayList<>();

    //Check each guess against the original atoms
    for (Atom guess : playerTwoGuesses) {
        boolean matchFound = false;
        for (Atom original : playerOneAtoms) {
            if (original.getPosition().equals(guess.getPosition())) {
                guessedCorrectly.add(original.getPosition());
                matchFound = true;
                break; // Stop checking if a match is found
            }
        }

        //Draw the guess with the appropriate colour
        g2d.setColor(matchFound ? Color.green : Color.red);
        Point pixelPoint = axialToPixel(guess.getPosition().x, guess.getPosition().y);
        g2d.fillOval( x: pixelPoint.x - HEX_SIZE / 2,  y: pixelPoint.y - HEX_SIZE / 2, HEX_SIZE, HEX_SIZE);
    }
    score -= (6-guessedCorrectly.size())*10;
    scoreBoard.setText("Score: " + score); // Update the score display
```

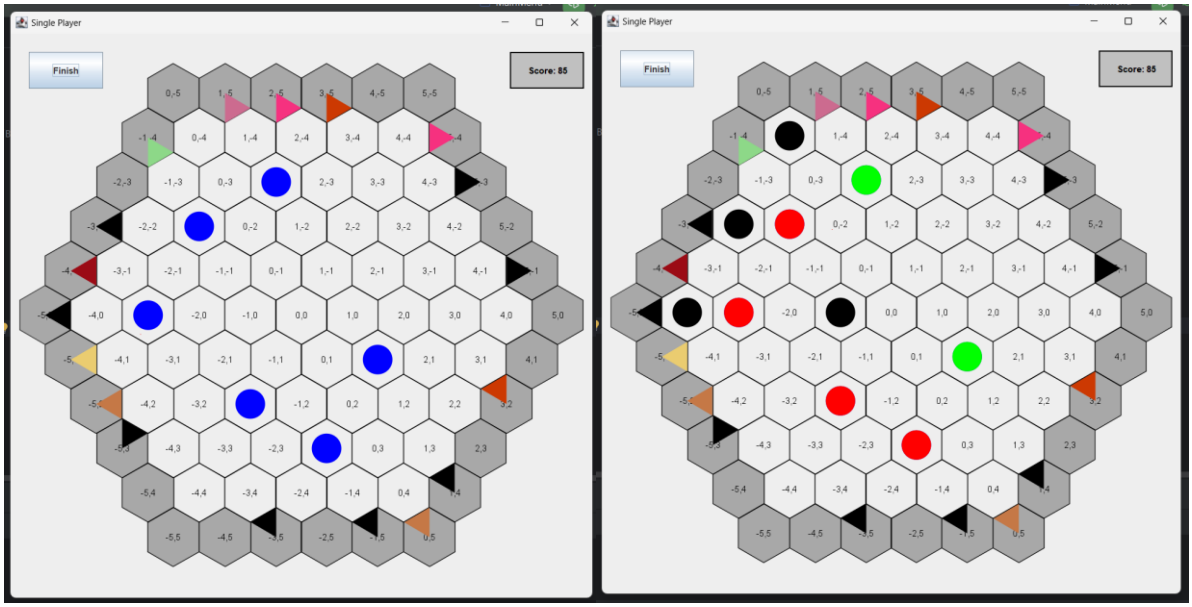
```
    //Draw original atoms that were not guessed correctly
    for (Atom original : playerOneAtoms) {
        if (!guessedCorrectly.contains(original.getPosition())) {
            g2d.setColor(Color.black);
            Point pixelPoint = axialToPixel(original.getPosition().x, original.getPosition().y);
            g2d.fillOval( x: pixelPoint.x - HEX_SIZE / 2,  y: pixelPoint.y - HEX_SIZE / 2, HEX_SIZE, HEX_SIZE);
        }
        //Correctly guessed atoms are already drawn in green so no need to redraw them here.
    }
    //Add end gamebutton
    updateFinishButtonState();
    finishButton.setVisible(true);
}
```

Finally upon ending game arrays are emptied and callFinishScreen method is called to display finish screen.

```
if(endGame){
    playerTwoGuesses.clear();
    playerOneAtoms.clear();
    callFinishScreen();
}
```

### Single player:

When the single player game mode is initiated the atoms are already hidden (randomly allocated and invisible to user) the user sends rays and places 6 atoms of where they think the actual atoms are hidden. After 6 atoms are placed a finish button appears and the game can then be concluded where the score is displayed and the hidden atoms are made visible.




## Single player Implementation:

Single player exnennds two players functionality, but player 1 is replaced by randomly allocated hidden atoms by the system. The player is two players player 2 in a sense and player 1 is the programme.

```java
public class SinglePlayer extends TwoPlayer {
    1 usage  artemkuc44
    public SinglePlayer() {
        super(); // Call the superclass constructor
        currentPlayer = 2; // Set currentPlayer to 2 since Player 1's actions are automated
        randomlyAllocateAtoms(); // Randomly allocate atoms for Player 1
        finishButton.setVisible(false); // Initially hide the finish button
        scoreBoard.setVisible(true);
        drawRayPaths = false;
    }
```

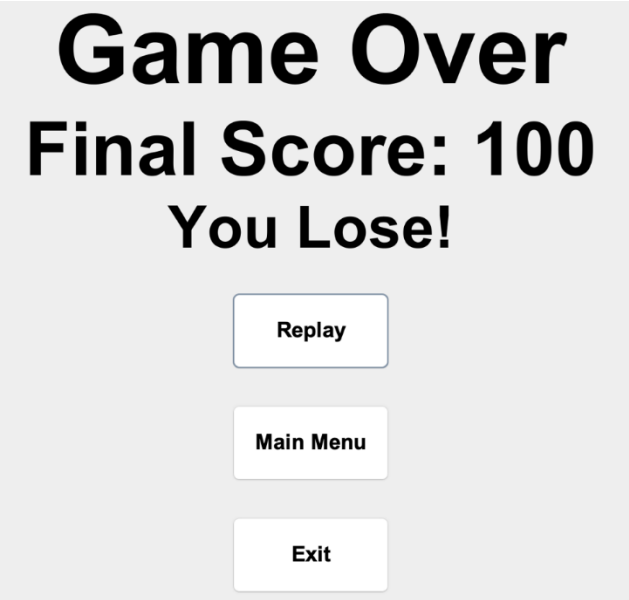
Most of the functionality is cover in two player class, thus this class is left simple with self explanatory methods.

```java
1 usage  artemkuc44
private void randomlyAllocateAtoms() {
    Random random = new Random();
    while (playerOneAtoms.size() < MAX_ATOMS) {
        int x = random.nextInt(DIAMETER_HEXAGONS) - DIAMETER_HEXAGONS / 2;
        int y = random.nextInt(DIAMETER_HEXAGONS) - DIAMETER_HEXAGONS / 2;
        Point randomPoint = new Point(x, y);
        // Ensure the random point is within hexCoordinates and not already occupied
        if (hexCoordinates.contains(randomPoint) && findAtomByAxial(playerOneAtoms, randomPoint) == null) {
            playerOneAtoms.add(new Atom(randomPoint));
        }
    }
}

2 usages  artemkuc44
@Override
protected void handleMouseClick(Point hexCoord, Point clickedPoint) {
    // Only handle clicks for Player 2's actions
    if (currentPlayer == 2) {
        super.handleMouseClick(hexCoord, clickedPoint);
    }
}
```

FinishScreen class:

Displays once the user compares the atoms placed vs the atoms guessed. The finishing screen consists of a simple display that is modelled off the revised MainMenu to give the user the option to replay, return to main menu or exit the game.

As of now it contains some place holder text which will be replaced by winning logic next sprint (this was not a requirement for this sprint).




```
// Add ActionListeners
ReplayButton.addActionListener(e ->
        replayGame(frame));
MMButton.addActionListener(e ->
        goToMainMenu(frame));
ExitButton.addActionListener(e ->
        System.exit( status: 0));
```

```
1 usage  new *
private void goToMainMenu(JFrame frame) {
    MainMenu.frame.dispose();

    MainMenu.displayMainMenu();
    frame.dispose();
}
```

```
private void replayGame(JFrame frame) {
    if(isSinglePlayer) {
        MainMenu.frame.dispose();
        frame.getContentPane().removeAll(); //when its presse

        SinglePlayer singlePlayerPanel = new SinglePlayer();
        frame.getContentPane().removeAll(); //when its presse
        frame.setSize( width: 800, height: 800);
        frame.setLocationRelativeTo(null); //makes it so when
        frame.add(singlePlayerPanel, BorderLayout.CENTER); //
        frame.setTitle("Single Player");

        frame.validate(); //validates
        frame.repaint(); //painting
    }else{
        MainMenu.frame.dispose();
        TwoPlayer twoPlayerPanel = new TwoPlayer();
        frame.getContentPane().removeAll(); //when its presse
        frame.setSize( width: 800, height: 800);
        frame.setLocationRelativeTo(null); //makes it so when
```

Rules Buttons added:

As well to enhance the users experience we implemented more effient displays of the rules for each more now the game has 3 separate buttons beside each mode on the main menu for the user to click and investigate each set of rules regarding each game. This is done using a simple

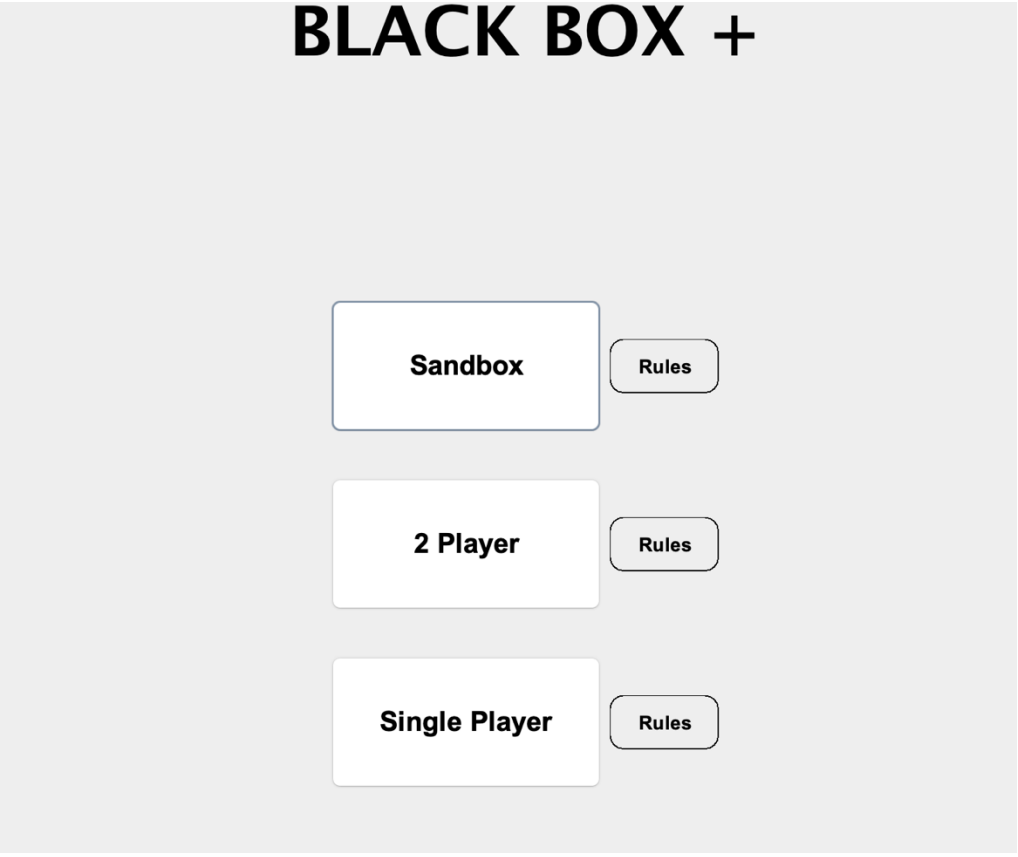
method called showRules(). It implements a switch statement to improve effientcy that switchs the message depending on the gamemode selected

```
switch(gameMode){
    case "Sandbox":


        RuleSet =     "The Sandbox is an area for you to experiment with the capability of our game an
                      "ATOMS:\nIn the Sandbox you can place the atoms and clearly see the circle of
                      "RAYS:\nThe ray markers have a random distinctive colour for each one of them
                      "ENJOY";

    case "2 Player":
       RuleSet =     "The 2 player mode is a great mode to play with friends. One places the atoms, th
                      "The Rules:\nPlayer 1 places 6 atoms\nThe game commences and Player 2 has to f
                      "The Winner:\nIt being a 2 player game there is two different ways of winning:
    case "Single Player":
      RuleSet = "The Single player mode will be the most popular for the people who want to master a g
                      "The Rules:Once game commences and the player has to find all 6 atoms by strat
        break;
    default:
```

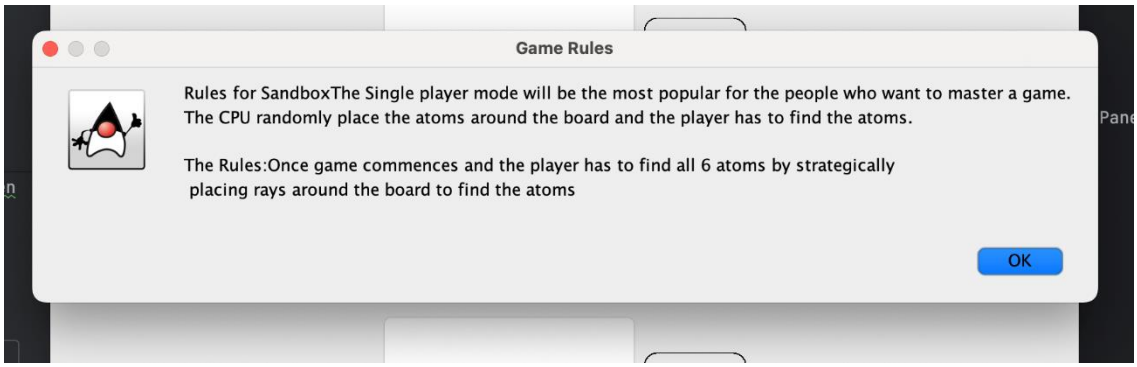
Rules messages in code.

```
String rulesContent = "Rules for " + gameMode + RuleSet;
JOptionPane.showMessageDialog(frame, rulesContent, title: "Game Rules", JOptionPane.INFORMATION_MESSAGE);
```

How the message is displayed using Info_message.



How its displayed in the mainmenu.




# Testing

```
@Test
void ValidDirection() { //checks valid direction
    assertDoesNotThrow(() -> new Ray(new Point( x: 5, y: 0), new Point( x: 0, y: 1)));
    assertDoesNotThrow(() -> new Ray(new Point( x: 5, y: 2), new Point( x: 1, y: 1)));
    assertDoesNotThrow(() -> new Ray(new Point( x: 1, y: 7), new Point( x: 1, y: 0)));
    assertDoesNotThrow(() -> new Ray(new Point( x: 3, y: 5), new Point( x: 1, y: 1)));
}

new *
@Test
void InvalidDirection() { //checks invalid direction
    assertThrows(IllegalArgumentException.class, () -> new Ray(new Point( x: 0, y: 0), new Point( x: 2, y: 0)));
    assertThrows(IllegalArgumentException.class, () -> new Ray(new Point( x: 0, y: 0), new Point( x: 4, y: 2)));
    assertThrows(IllegalArgumentException.class, () -> new Ray(new Point( x: 0, y: 0), new Point( x: 12, y: 0)));
    assertThrows(IllegalArgumentException.class, () -> new Ray(new Point( x: 0, y: 0), new Point( x: 2, y: 1)));
}
```

Aim: Make sure that rays can only be given direction in 45 degrees increments, ex. (1,0) would mean to go straight left.

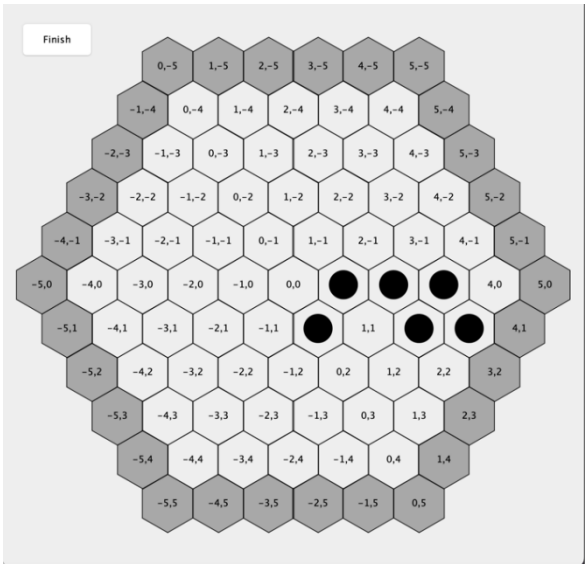
Process: creates new ray with location and given a direction. If it is given a correct direction, it should not output anything hence why “assertDoesNotThrow” is used for ValidDirection. “AssertThrows” is used for InvalidDirection as it is expecting an output as the directions are wrong.

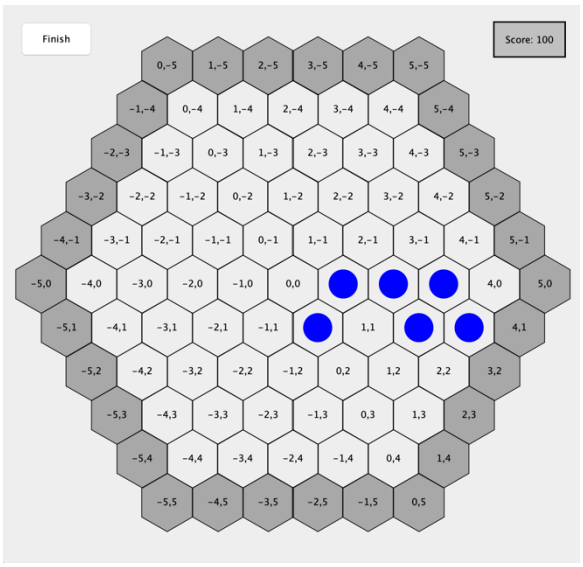
```
@Test
void CheckDirection() {
    assertDoesNotThrow(() -> ray.setdirection(new Point( x: 0,  y: -1)));
    assertEquals( expected: 0, ray.getDirection().x);
    assertEquals( expected: -1, ray.getDirection().y);
}
```

Aim: Make sure that getDirection() works correctly.

Process: sets ray direction as (0, -1) which would mean it is going south west direction and also makes sure an illegalArgument is not thrown. “AssertEquals” used to compare the numbers to what is being returned  by using getDirection() function.

```
@Test
void CorrectGuesses() {
    // Setup - Player 1 places atoms
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 0,  y: 1)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 1,  y: 0)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 2,  y: 1)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 2,  y: 0)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 3,  y: 1)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 3,  y: 0)));

    // Player 2 guesses
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 0,  y: 1)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 1,  y: 0)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 2,  y: 1)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 2,  y: 0)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 3,  y: 1)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 3,  y: 0)));

    assertTrue(FinishScreen.findWinner(),  message: "Player 2 wins if all guesses are correct.");
}
```

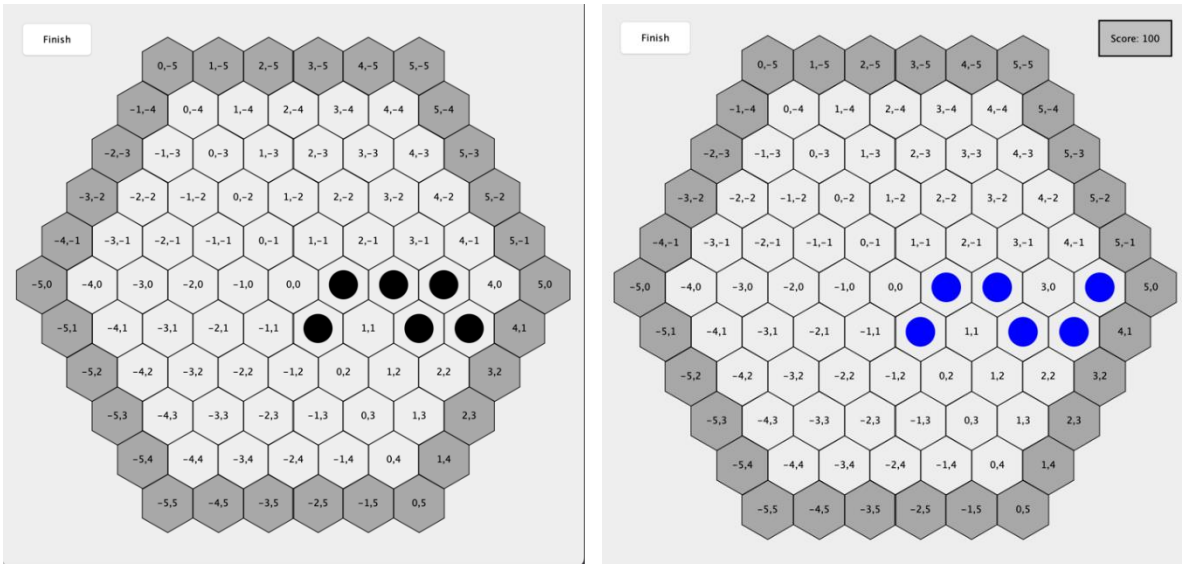



Aim: Make sure that the function “FindWinner()” works correctly when all guesses are correct.

Process: Adds 6 atoms from player one, and also adds the 6 guesses from player 2, as shown above.“AssertTrue” in runs the function findWinner() to find out if player 2 won or not. Black is the atoms inserted by player 1 and blue is by player 2. AssertTrue is used as the atoms placed are the same as shown above.

```
@Test
void IncorrectGuesses() {
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 0,  y: 1)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 1,  y: 0)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 2,  y: 1)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 2,  y: 0)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 3,  y: 1)));
    TwoPlayer.playerOneAtoms.add(new Atom(new Point( x: 3,  y: 0)));

    // Player 2 guesses
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 0,  y: 1)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 1,  y: 0)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 2,  y: 1)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 2,  y: 0)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 3,  y: 1)));
    TwoPlayer.playerTwoGuesses.add(new Atom(new Point( x: 4,  y: 0)));

    assertFalse(FinishScreen.findWinner(),  message: "Player 1 should win if Player 2 guesses incorrectly.");
}
```




Aim: Make sure that the function “FindWinner()” works correctly when all guesses are correct.

Process: Adds 6 atoms from player one, and also adds the 6 guesses from player 2, as shown above. “AssertFalse” runs the function findWinner() to find out if player 2 won or not. Black is the atoms inserted by player 1 and blue is by player 2. AssertFalse is used as it should output false as the atoms placed are not the same.

```
@Test
void testFinishActionTransitionsFromPlayerOneToTwo() {
    TwoPlayer game = new TwoPlayer();
    game.currentPlayer = 1;
    game.finishAction();
    assertEquals( expected: 2, game.currentPlayer,  message: "Should switch from player1 to player2.");
}
```

Aim: Make sure finishAction() transitions between players.

Process: Sets current player to 1, and then calls the function “finishAction(). AssertEquals compares 2 to the current player number.

```java
@Test
void testFinishActionConcludesGameAfterPlayerTwoFinishes() {
    TwoPlayer game = new TwoPlayer();
    game.currentPlayer = 2;
    game.finishAction();
    assertTrue(game.finish, message: "The game should be marked as finished after player 2 finishes.");
}
```

Aim: Make sure finishAction() works when game finished.

Process: Sets current player to 2, then calls the function finishAction(), when the function is called when its player two, the game should end. AssertTrue used to justify this.