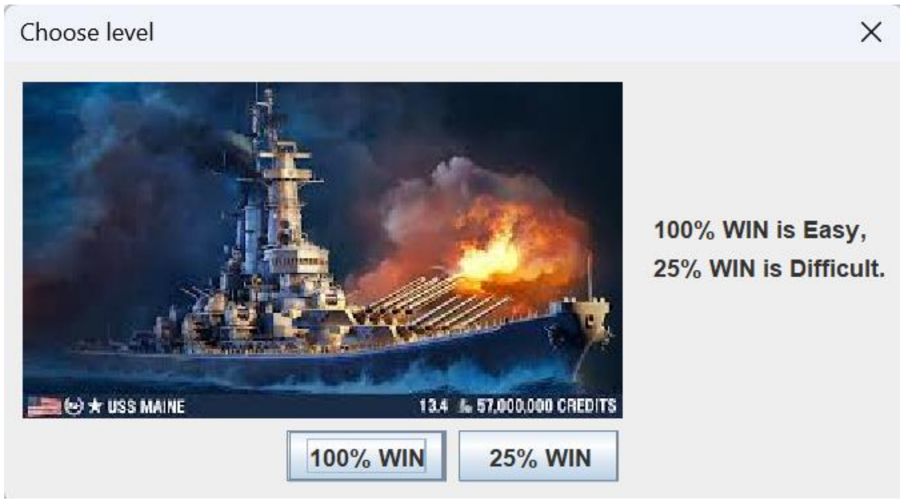
Student name: Tô Duy Thịnh

Student ID: ITCSIU22138

RULES

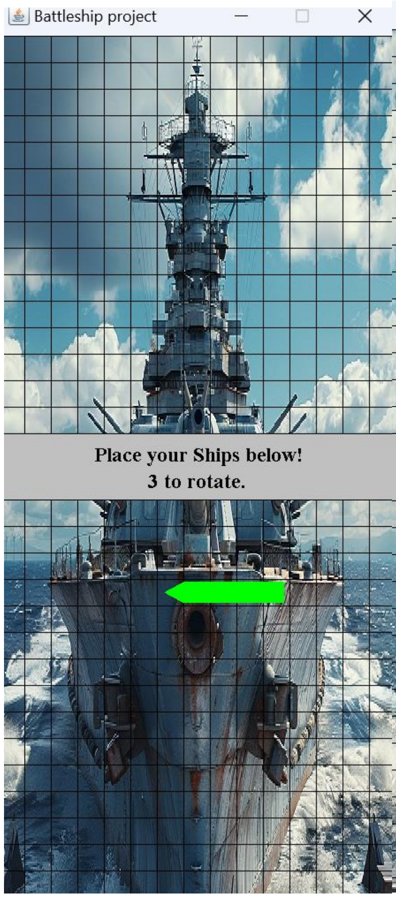
The chosen implementation of Battleship for this example game has included multiple levels of AI to make it a single played game. When the game begins the dialog shown below appears before the main game opens to allow selection of the difficulty.



Choose level

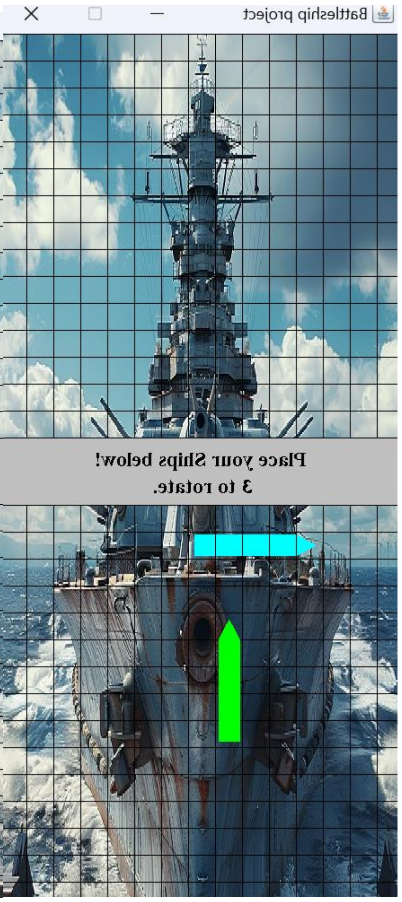
100% WIN is Easy,
25% WIN is Difficult.

After that you place the ships:



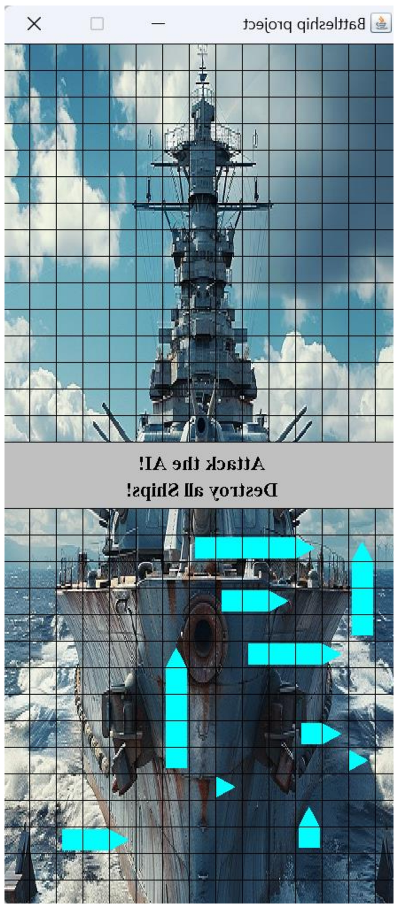
Battleship project

Place your Ships below!
3 to rotate.



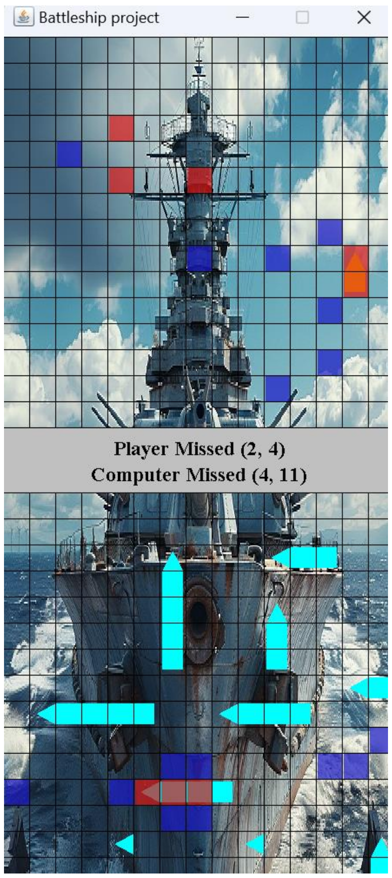
Battleship project

Place your Ships below!
3 to rotate.

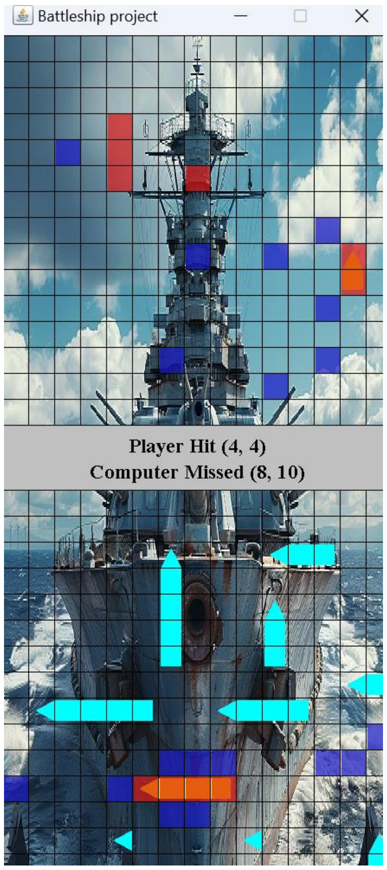


Battleship project

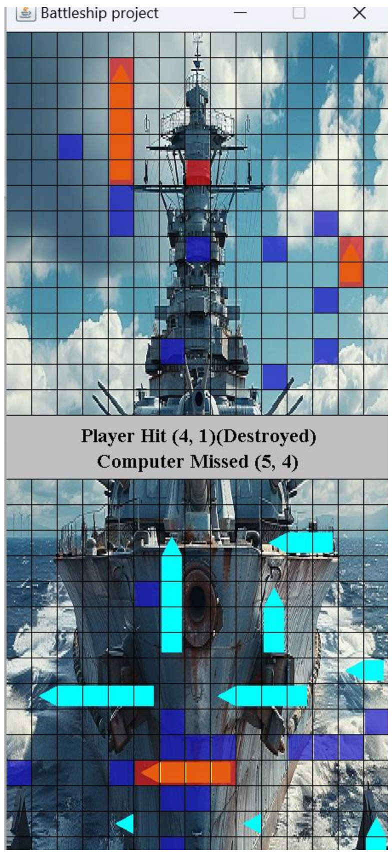
Attack the AI!
Destroy all Ships!



Player Missed (2, 4)
Computer Missed (4, 11)



Player Hit (4, 4)
Computer Missed (8, 10)



Player Hit (4, 1)(Destroyed)
Computer Missed (5, 4)

The controls can be summarised as the following.

- At any time:
  - 1 to quit.
  - 2 to restart.
  - 4 to activate debug mode to cheat and view the opponent's ships.
- During Placement Phase:
  - Click to place ship (only places the ship if it is a valid placement).
  - 3 to rotate between vertical and horizontal for placement.
- During Attack Phase:
  - Click on the enemy's grid in places that have not yet been marked to reveal the squares as hits or misses.

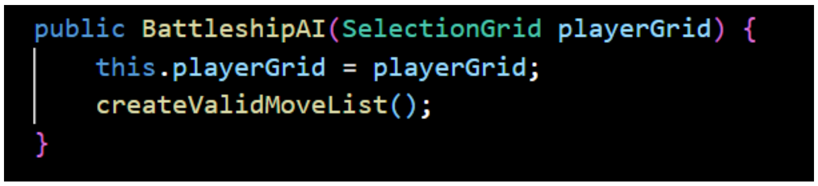
Battleship has a simple set of rules that create an enjoyable game through trying to find an opponent's ships before they find yours. The rules can be summarised as follow.

- Each player begins with a grid and place 10 ships on their grid. All ships are 1 unit wide, and with lengths of 5, 5, 4, 4, 3, 3, 2, 2, 1 and 1. These ships can be placed either horizontally or vertically.
- The game starts with the player and their opponent not knowing where the opponent’s ships are.
- After the preparation stage of placing ships the game alternates between players allowing them to select one new position on the opponent’s grid that has not yet been attacked.
- The positions are marked once they have been attacked, either with a blue marker indicating it was a miss in open water, or a red marker indicating a ship was hit.
- Once a ship has been destroyed from all grid positions having been hit, the ship is revealed as destroyed to the other player.
- The game ends when all ships for either player’s side have been destroyed. The winner is the player with ships still not destroyed.

EXPLAIN

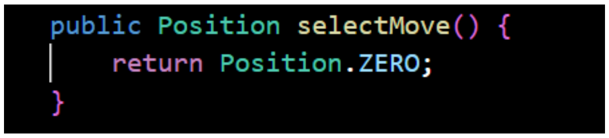
BattleshipAI.java

1. Data Structure:
   - SelectionGrid playerGrid:
     - This is a reference to the selection grid that the player is controlling. It can represent the grid of the Battleship game.
   - List<Position> validMoves:
     - This is a list of valid positions for the AI to attack. This structure helps manage the positions that can be attacked (can be filtered or updated after each attack).
2. Algorithm:
   - Initialization in the BattleshipAI(SelectionGrid playerGrid) constructor:



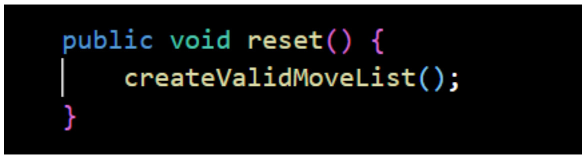
```java
public BattleshipAI(SelectionGrid playerGrid) {
    this.playerGrid = playerGrid;
    createValidMoveList();
}
```

     - createValidMoveList() will iterate over all coordinates on the board, from x = 0 to SelectionGrid.GRID_WIDTH and from y = 0 to SelectionGrid.GRID_HEIGHT. Then, Position objects (each containing a coordinate (x, y)) are added to the validMoves list.
     - Each iteration creates a new Position object representing each coordinate on the grid, resulting in a list containing all possible attack positions.
   - selectMove():



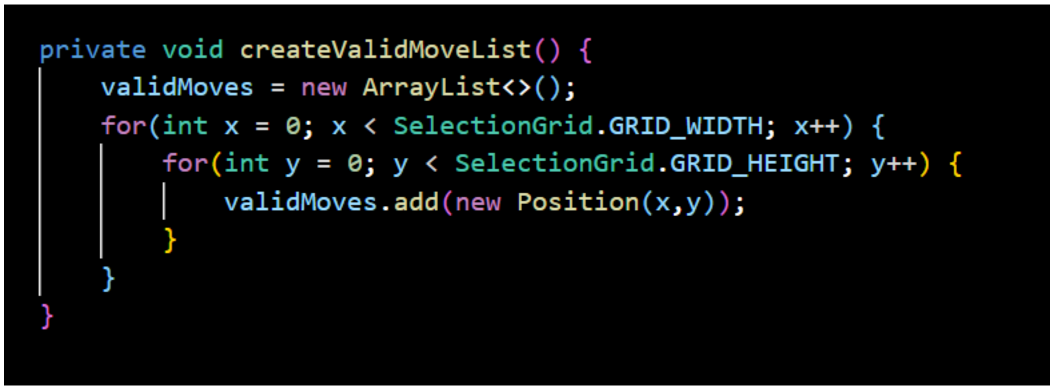
```java
public Position selectMove() {
    return Position.ZERO;
}
```

     - This is the default method for selecting the attack position, which returns Position.ZERO by default, i.e. coordinates (0, 0).
   - Reset():



```
public void reset() {
    createValidMoveList();
}
```

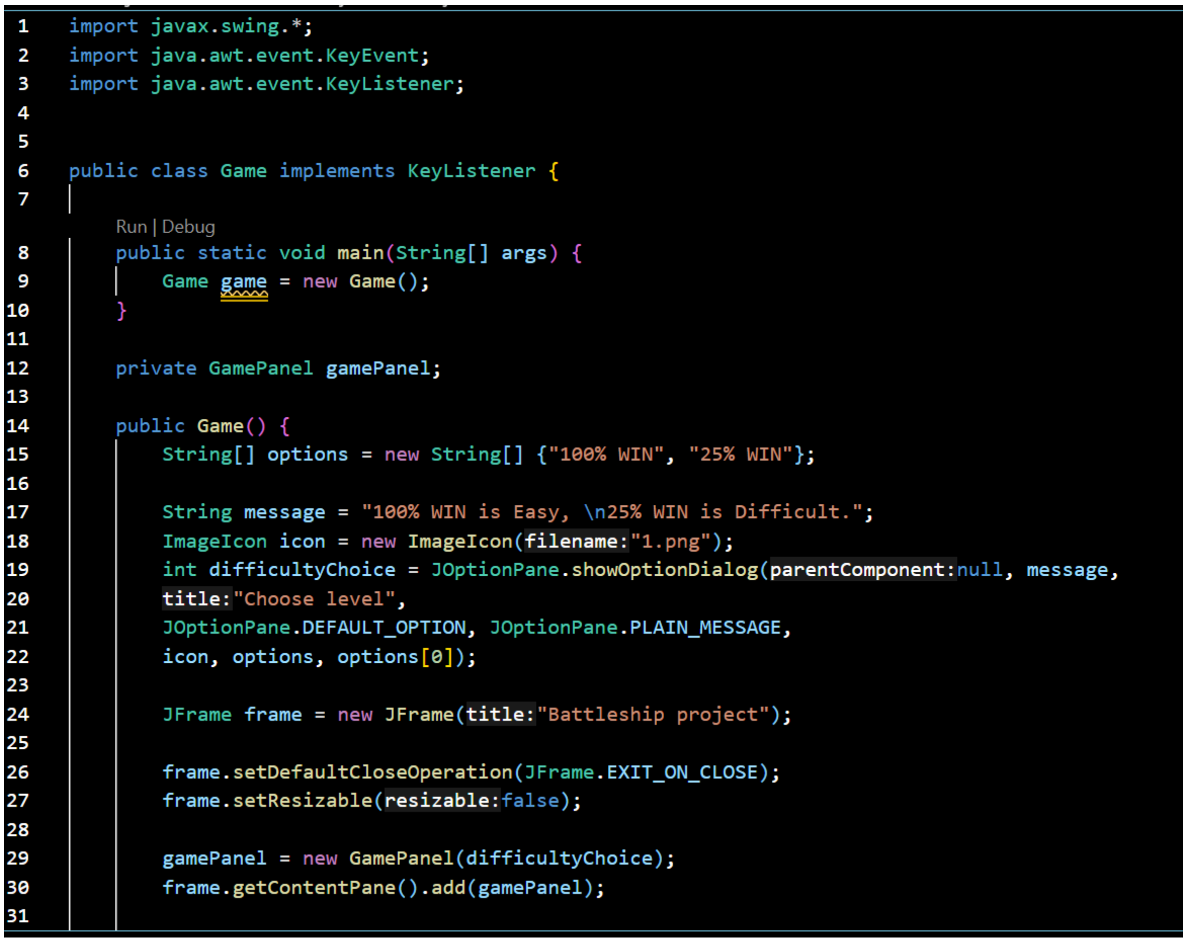
  - When this function is called, the validMoves list will be regenerated by calling createValidMoveList() again.
  - This method ensures that the list of valid positions is refreshed, which can be useful when valid positions change after each attack.
- createValidMoveList() function:



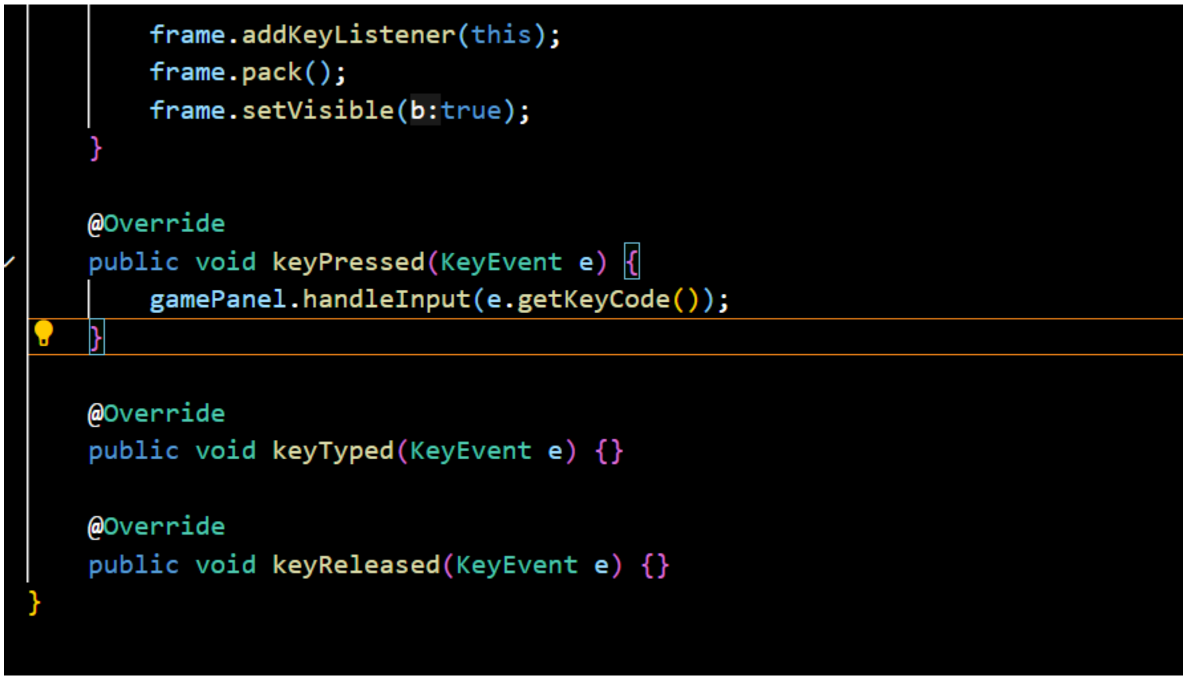
```
private void createValidMoveList() {
    validMoves = new ArrayList<>();
    for(int x = 0; x < SelectionGrid.GRID_WIDTH; x++) {
        for(int y = 0; y < SelectionGrid.GRID_HEIGHT; y++) {
            validMoves.add(new Position(x,y));
        }
    }
}
```

  - This function rebuilds the validMoves list by iterating over all possible coordinates on the board and creating Position objects.
  - This is an algorithm that traverses the grid to generate all valid coordinates and store them in a list. This algorithm has a time complexity of O(n * m), where n is the width of the grid (SelectionGrid.GRID_WIDTH) and m is the height of the grid (SelectionGrid.GRID_HEIGHT).

Game.java



```java
import javax.swing.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;


public class Game implements KeyListener {

    Run | Debug
    public static void main(String[] args) {
        Game game = new Game();
    }

    private GamePanel gamePanel;

    public Game() {
        String[] options = new String[] {"100% WIN", "25% WIN"};

        String message = "100% WIN is Easy, \n25% WIN is Difficult.";
        ImageIcon icon = new ImageIcon(filename:"1.png");
        int difficultyChoice = JOptionPane.showOptionDialog(parentComponent:null, message,
        title:"Choose level",
        JOptionPane.DEFAULT_OPTION, JOptionPane.PLAIN_MESSAGE,
        icon, options, options[0]);

        JFrame frame = new JFrame(title:"Battleship project");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setResizable(resizable:false);

        gamePanel = new GamePanel(difficultyChoice);
        frame.getContentPane().add(gamePanel);

```



```
        frame.addKeyListener(this);
        frame.pack();
        frame.setVisible(b:true);
    }

    @Override
    public void keyPressed(KeyEvent e) {
        gamePanel.handleInput(e.getKeyCode());
    }

    @Override
    public void keyTyped(KeyEvent e) {}

    @Override
    public void keyReleased(KeyEvent e) {}
}
```

1. Data Structure:
    - String[] options:
        - This is a string array (String[]) used to store the AI difficulty options, which include the values "100% WIN"and "25% WIN".
    - int difficultyChoice:
        - This is an integer value used to store the AI difficulty choice that the player chooses from the dialog box (JOptionPane). There are only three choices, so this value will be stored in an integer variable and corresponds to the index of the choice (0 for Easy, 1 for Medium, 2 for Hard).
    - GamePanel gamePanel:
        - This is a reference to the GamePanel object, which represents the main part of the player interface, where events such as input will be processed and drawn to the screen.
    - JFrame frame:
        - This is the main application window, using JFrame from the Swing library to create and display the user interface.
2. Algorithm:
    - The main(String[] args) method:
        - This method is the starting point of the program, initializing a new Game object.

- The Game() method (Constructor):
    - This is the constructor of the Game class, which:
    - Opens a dialog box (JOptionPane) for the player to choose the difficulty of the AI.
    - Based on the player's choice, a GamePanel object is created and assigned to the gamePanel.
    - The window (JFrame) is created and displayed with the gamePanel inside it. Keyboard events are also registered to call the keyPressed() method.
- KeyPressed(KeyEvent e) method:
    - This method is a piece of code that calls the handleInput() method in the gamePanel, passing the key code from the event to process.

- gamePanel.handleInput(e.getKeyCode()):
    - This means that the gamePanel must have a handleInput(int keyCode) method to handle the specific action corresponding to the key codes.
- The keyTyped(KeyEvent e) and keyReleased(KeyEvent e) methods:
    - Both of these methods are not used in the Game class. Therefore, they are only declarations without performing any action.

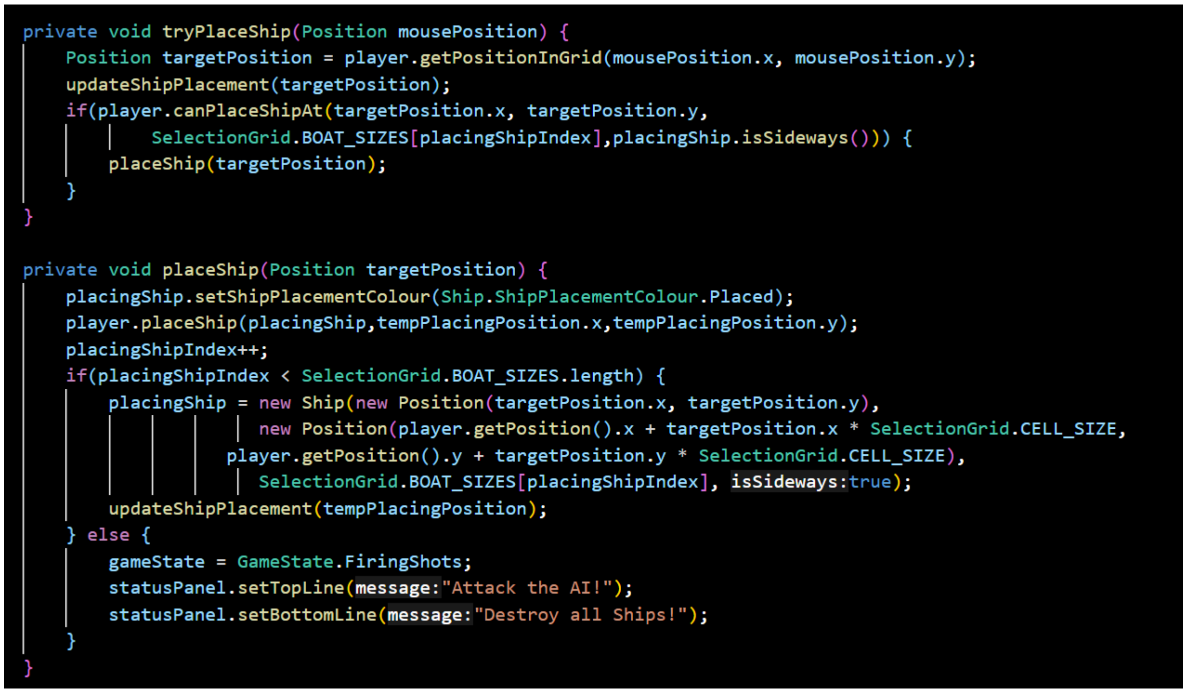
GamePanel.java

1. Data structuer:

- GameState Enum:
  - This enum defines the game states, including:
    - PlacingShips: The state when the player is placing a ship.
    - FiringShots: The state when the player is attacking the computer.
    - GameOver: The state when the game ends.
- Grids (computer, player):
  - SelectionGrid objects represent player and computer grids. Each grid will contain ships and allow position checks to determine if a ship has been hit.
- Ship (ship):
  - Ship objects are used to describe ships on the grid. This class has methods to check the position and state of the ship.

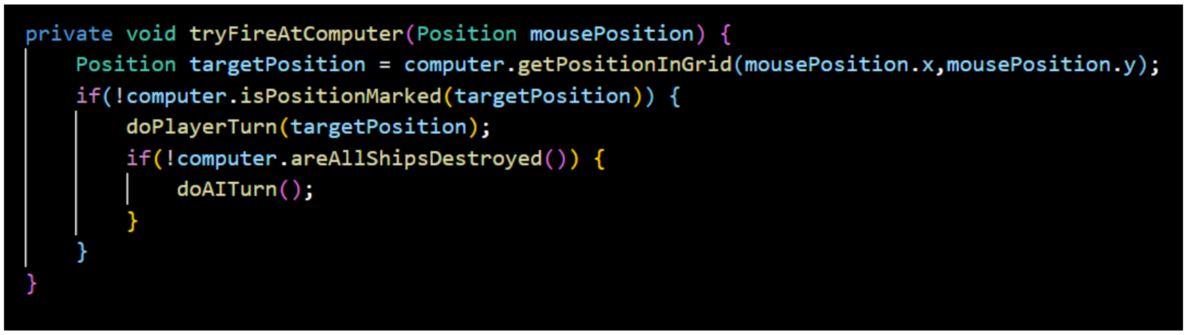
2. Algorithm

- Placing Ships:



```java
private void tryPlaceShip(Position mousePosition) {
    Position targetPosition = player.getPositionInGrid(mousePosition.x, mousePosition.y);
    updateShipPlacement(targetPosition);
    if(player.canPlaceShipAt(targetPosition.x, targetPosition.y,
            SelectionGrid.BOAT_SIZES[placingShipIndex],placingShip.isSideways())) {
        placeShip(targetPosition);
    }
}

private void placeShip(Position targetPosition) {
    placingShip.setShipPlacementColour(Ship.ShipPlacementColour.Placed);
    player.placeShip(placingShip,tempPlacingPosition.x,tempPlacingPosition.y);
    placingShipIndex++;
    if(placingShipIndex < SelectionGrid.BOAT_SIZES.length) {
        placingShip = new Ship(new Position(targetPosition.x, targetPosition.y),
                    new Position(player.getPosition().x + targetPosition.x * SelectionGrid.CELL_SIZE,
                    player.getPosition().y + targetPosition.y * SelectionGrid.CELL_SIZE),
                    SelectionGrid.BOAT_SIZES[placingShipIndex], isSideways:true);
        updateShipPlacement(tempPlacingPosition);
    } else {
        gameState = GameState.FiringShots;
        statusPanel.setTopLine(message:"Attack the AI!");
        statusPanel.setBottomLine(message:"Destroy all Ships!");
    }
}
```

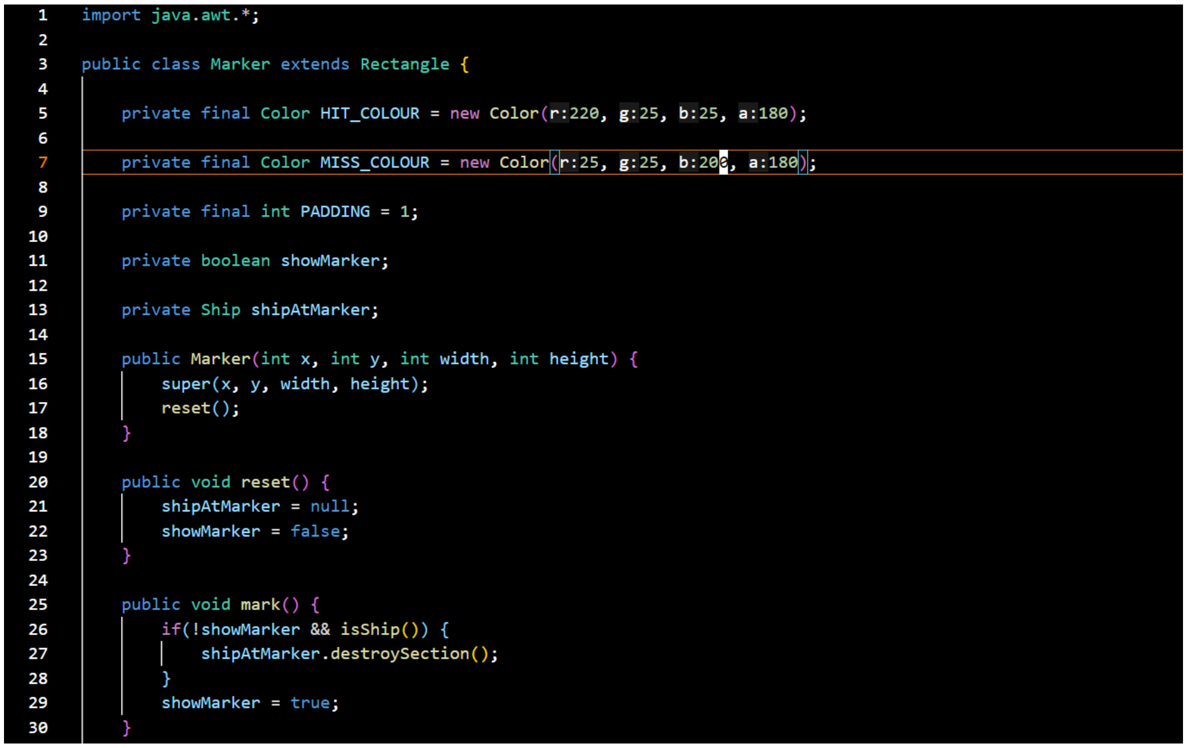
    - This algorithm handles the placement of ships by the player on the grid. It checks to see if the ship can be placed at the chosen location. If so, the ship is added to the grid. It then moves on to the next ship.
- Attack on the computer (Firing Shots):



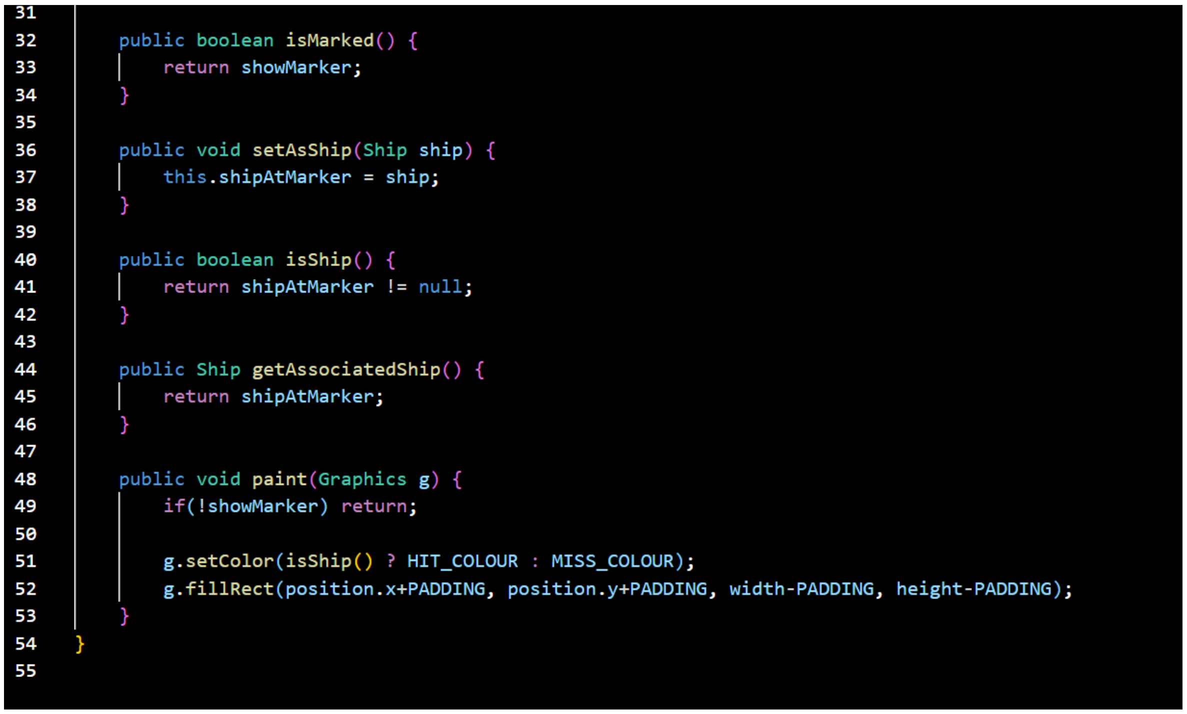
```
private void tryFireAtComputer(Position mousePosition) {
    Position targetPosition = computer.getPositionInGrid(mousePosition.x,mousePosition.y);
    if(!computer.isPositionMarked(targetPosition)) {
        doPlayerTurn(targetPosition);
        if(!computer.areAllShipsDestroyed()) {
            doAITurn();
        }
    }
}
```

    - This algorithm checks to see if the player has hit the computer ship and updates the game state based on that result.
- Computer AI:
    - BattleshipAI handles the computer's decisions in the game. There are two types of AI: SimpleRandomAI and SmarterAI.

Marker.java



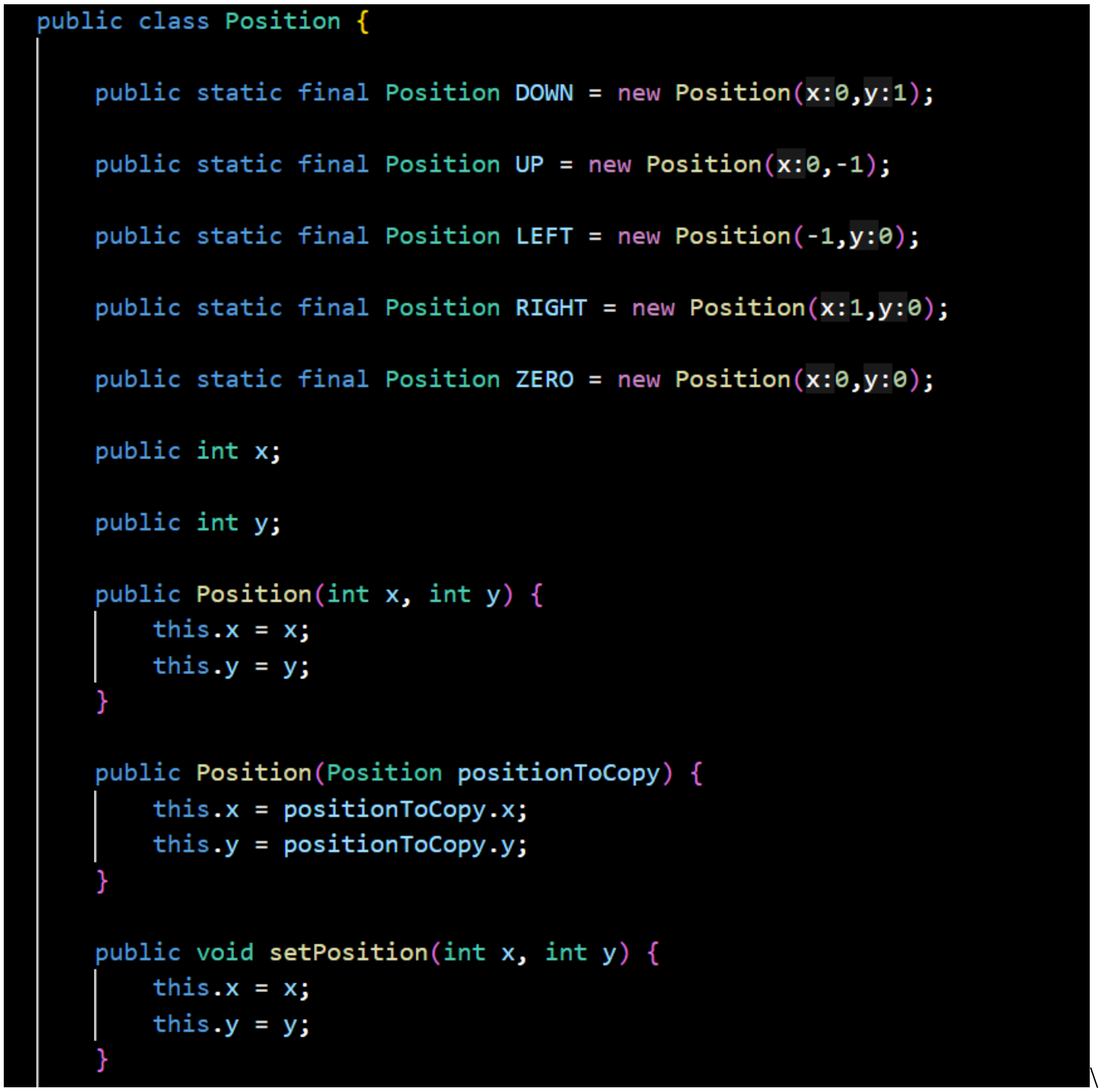
```java
import java.awt.*;

public class Marker extends Rectangle {

    private final Color HIT_COLOUR = new Color(r:220, g:25, b:25, a:180);

    private final Color MISS_COLOUR = new Color(r:25, g:25, b:200, a:180);

    private final int PADDING = 1;

    private boolean showMarker;

    private Ship shipAtMarker;

    public Marker(int x, int y, int width, int height) {
        super(x, y, width, height);
        reset();
    }

    public void reset() {
        shipAtMarker = null;
        showMarker = false;
    }

    public void mark() {
        if(!showMarker && isShip()) {
            shipAtMarker.destroySection();
        }
        showMarker = true;
    }
```



```java

    public boolean isMarked() {
        return showMarker;
    }

    public void setAsShip(Ship ship) {
        this.shipAtMarker = ship;
    }

    public boolean isShip() {
        return shipAtMarker != null;
    }

    public Ship getAssociatedShip() {
        return shipAtMarker;
    }

    public void paint(Graphics g) {
        if(!showMarker) return;

        g.setColor(isShip() ? HIT_COLOUR : MISS_COLOUR);
        g.fillRect(position.x+PADDING, position.y+PADDING, width-PADDING, height-PADDING);
    }
}

```

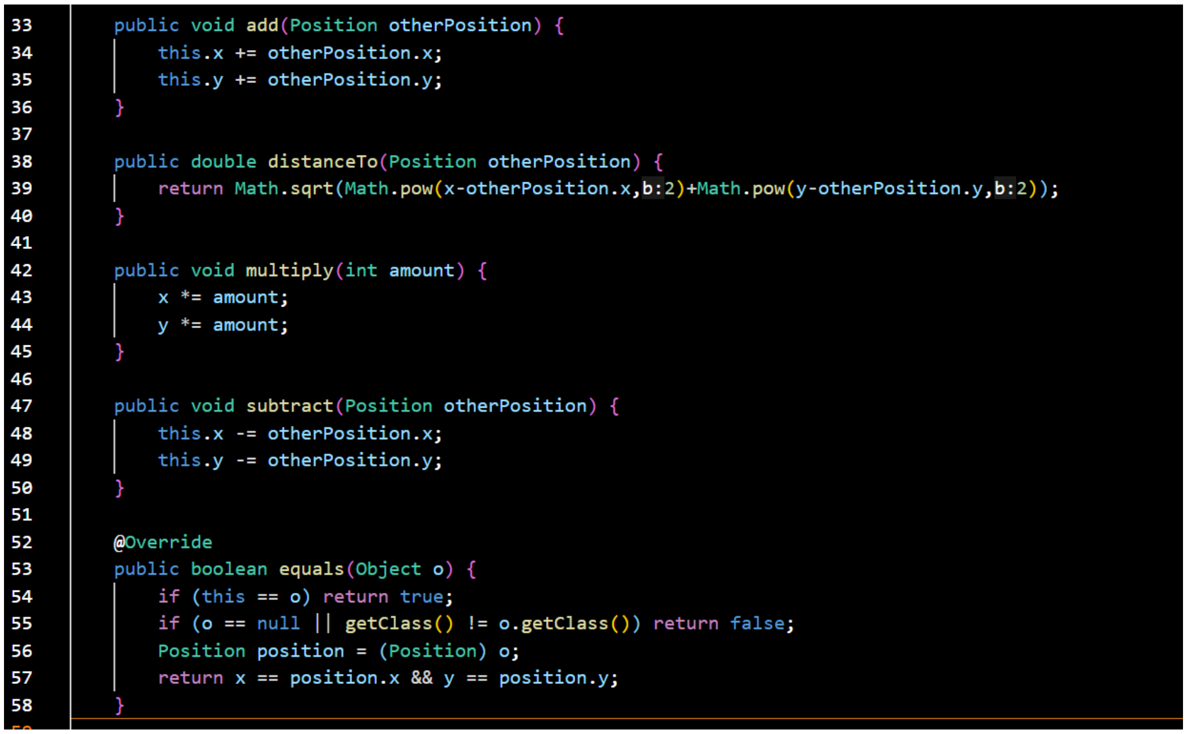
1. Data structure:
    - The Marker class uses a Ship object to attach a ship to a cell and uses boolean properties (showMarker) to determine the cell's state.
2. Algorithm:
    - The methods in the class mainly handle the state of the Marker and the attachment of ships to cells. Simple methods such as mark(), reset(), isShip() and paint() are basic operations that help control the state and image of markers on the interface.

Position.java

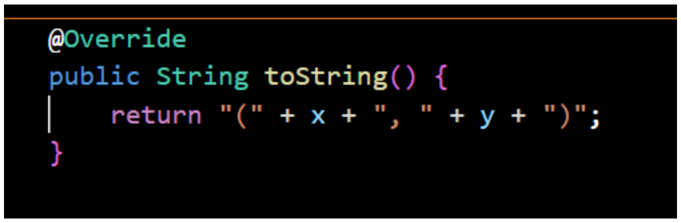


```java
public class Position {

    public static final Position DOWN = new Position(x:0,y:1);

    public static final Position UP = new Position(x:0,-1);

    public static final Position LEFT = new Position(-1,y:0);

    public static final Position RIGHT = new Position(x:1,y:0);

    public static final Position ZERO = new Position(x:0,y:0);

    public int x;

    public int y;

    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Position(Position positionToCopy) {
        this.x = positionToCopy.x;
        this.y = positionToCopy.y;
    }

    public void setPosition(int x, int y) {
        this.x = x;
        this.y = y;
    }
```

\



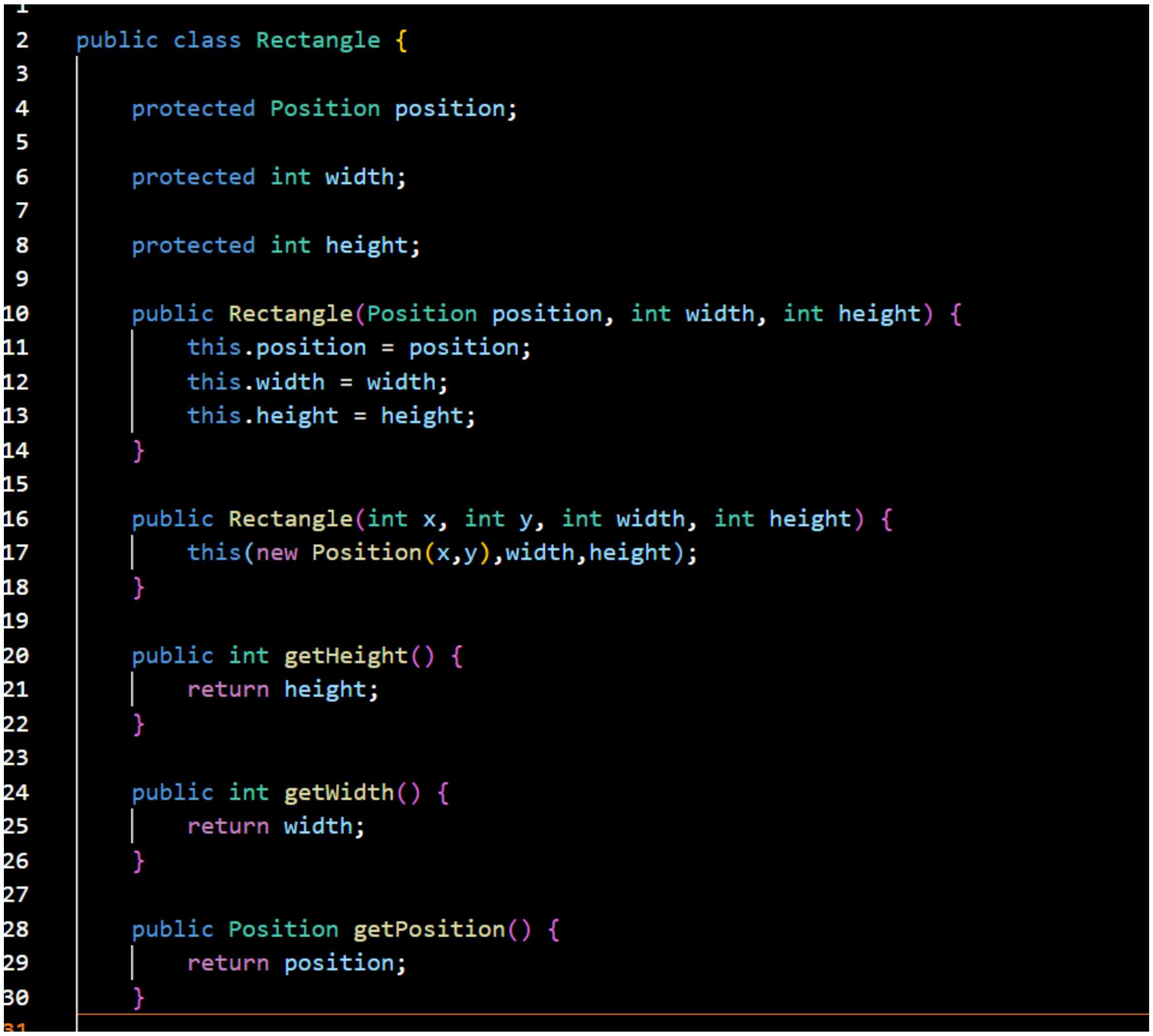
```java
    public void add(Position otherPosition) {
        this.x += otherPosition.x;
        this.y += otherPosition.y;
    }

    public double distanceTo(Position otherPosition) {
        return Math.sqrt(Math.pow(x-otherPosition.x,b:2)+Math.pow(y-otherPosition.y,b:2));
    }

    public void multiply(int amount) {
        x *= amount;
        y *= amount;
    }

    public void subtract(Position otherPosition) {
        this.x -= otherPosition.x;
        this.y -= otherPosition.y;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Position position = (Position) o;
        return x == position.x && y == position.y;
    }
```



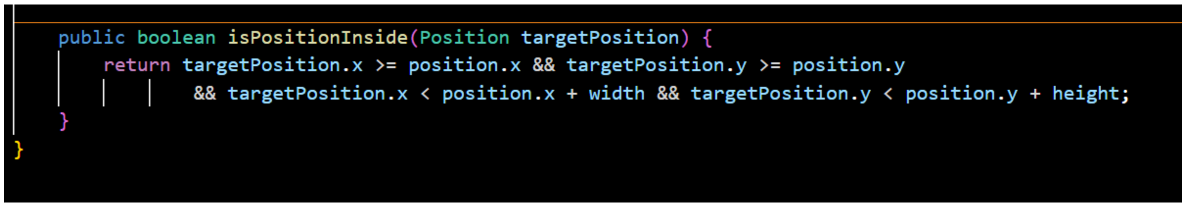
```java
@Override
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

1. Data Structure:
   - The Position class uses basic properties (int) to store coordinates in 2D space and methods to perform basic vector operations (addition, subtraction, multiplication, distance calculation).
2. Algorithms:
   - The algorithms in this class are mainly related to basic vector operations in 2D space. They are very simple and cannot be replaced by other algorithms without losing the purpose of the Position class. Operations such as addition, subtraction, multiplication and distance calculation are very basic and cannot be replaced without losing the accuracy of geometric operations.

Rectangle.java



```java

public class Rectangle {

    protected Position position;

    protected int width;

    protected int height;

    public Rectangle(Position position, int width, int height) {
        this.position = position;
        this.width = width;
        this.height = height;
    }

    public Rectangle(int x, int y, int width, int height) {
        this(new Position(x,y),width,height);
    }

    public int getHeight() {
        return height;
    }

    public int getWidth() {
        return width;
    }

    public Position getPosition() {
        return position;
    }

```



```java
    public boolean isPositionInside(Position targetPosition) {
        return targetPosition.x >= position.x && targetPosition.y >= position.y
                && targetPosition.x < position.x + width && targetPosition.y < position.y + height;
    }
}
```

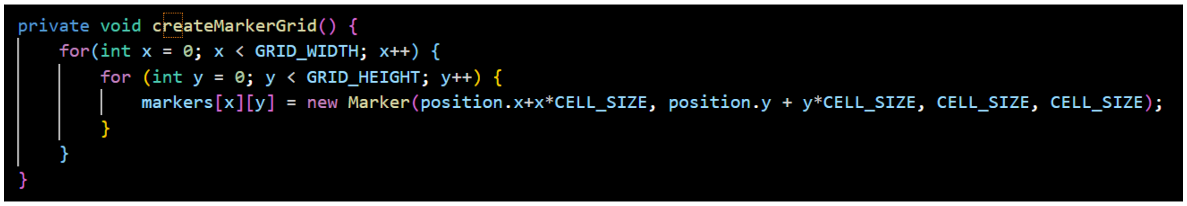
1. Data Structure:
    - The Rectangle class uses Position to store position information, and integer width and height values to store the dimensions of the rectangle.
2. Algorithm:
    - The methods in this class mainly perform simple calculations and comparisons.

- The isPositionInside method is a conditional check algorithm, and simple getter methods return property values.

SelectionGrid.java

Algorithms in the code

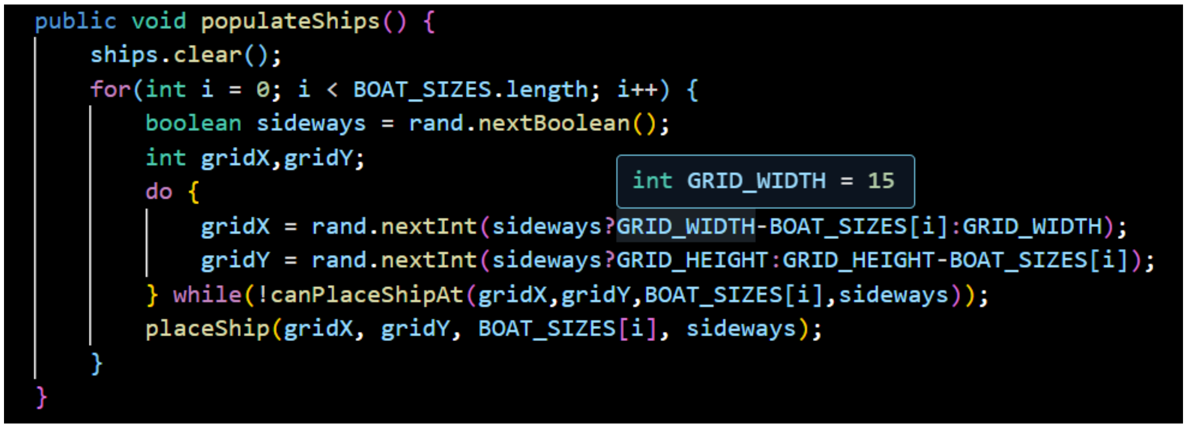
Initialize the markers grid:



```java
private void createMarkerGrid() {
    for(int x = 0; x < GRID_WIDTH; x++) {
        for (int y = 0; y < GRID_HEIGHT; y++) {
            markers[x][y] = new Marker(position.x+x*CELL_SIZE, position.y + y*CELL_SIZE, CELL_SIZE, CELL_SIZE);
        }
    }
}
```

This is an O(GRID_WIDTH * GRID_HEIGHT) algorithm to initialize all Marker objects in the grid. Each cell is initialized at position (x, y) and has a certain cell size.

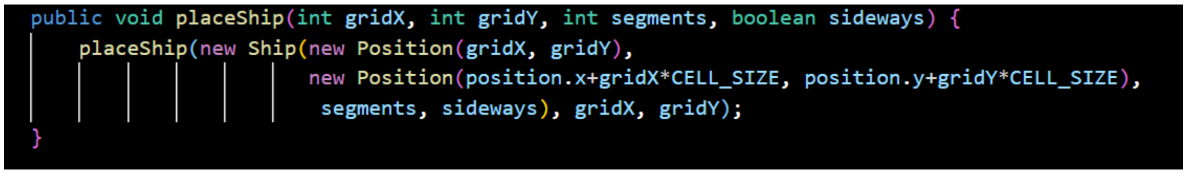
Place random ships:



```java
public void populateShips() {
    ships.clear();
    for(int i = 0; i < BOAT_SIZES.length; i++) {
        boolean sideways = rand.nextBoolean();
        int gridX,gridY;
        do {
            gridX = rand.nextInt(sideways?GRID_WIDTH-BOAT_SIZES[i]:GRID_WIDTH);
            gridY = rand.nextInt(sideways?GRID_HEIGHT:GRID_HEIGHT-BOAT_SIZES[i]);
        } while(!canPlaceShipAt(gridX,gridY,BOAT_SIZES[i],sideways));
        placeShip(gridX, gridY, BOAT_SIZES[i], sideways);
    }
}
```

This algorithm uses backtracking to try random positions for the ship. The ship position is randomly selected and retried if the ship cannot be placed on the grid. In case of failure, the algorithm continues to try until the ship is successfully placed.

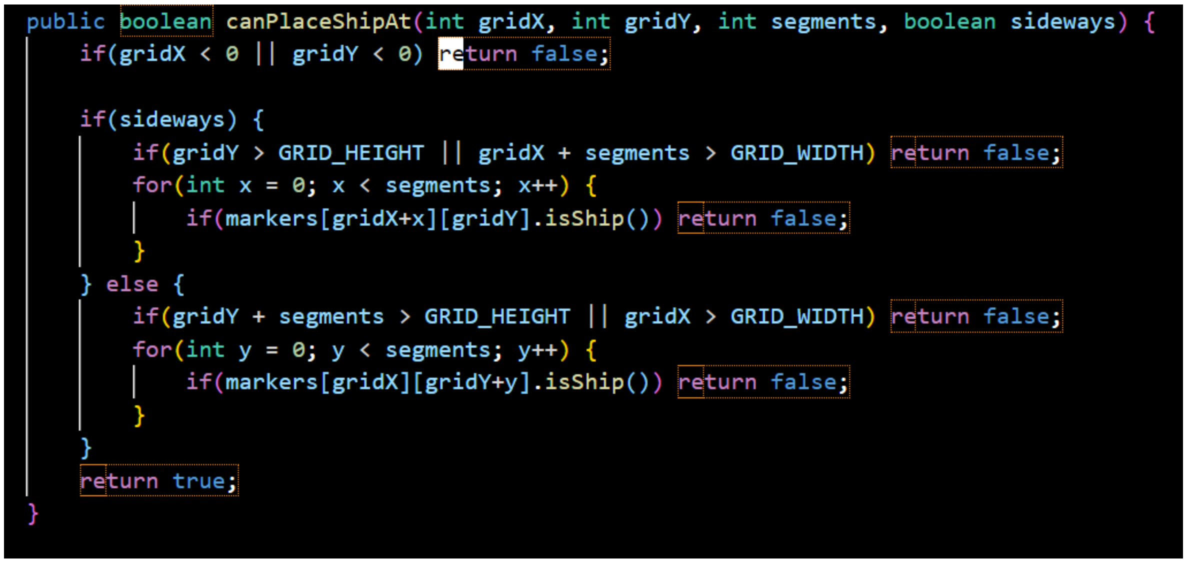
Place the ship on the grid:



```java
public void placeShip(int gridX, int gridY, int segments, boolean sideways) {
    placeShip(new Ship(new Position(gridX, gridY),
                        new Position(position.x+gridX*CELL_SIZE, position.y+gridY*CELL_SIZE),
                         segments, sideways), gridX, gridY);
}
```

This is an algorithm that places ships on the grid according to coordinate indices (gridX, gridY) and checks in the horizontal or vertical direction. The ship position is determined and then the corresponding cells are marked as "containing ships".

Check for valid train booking:



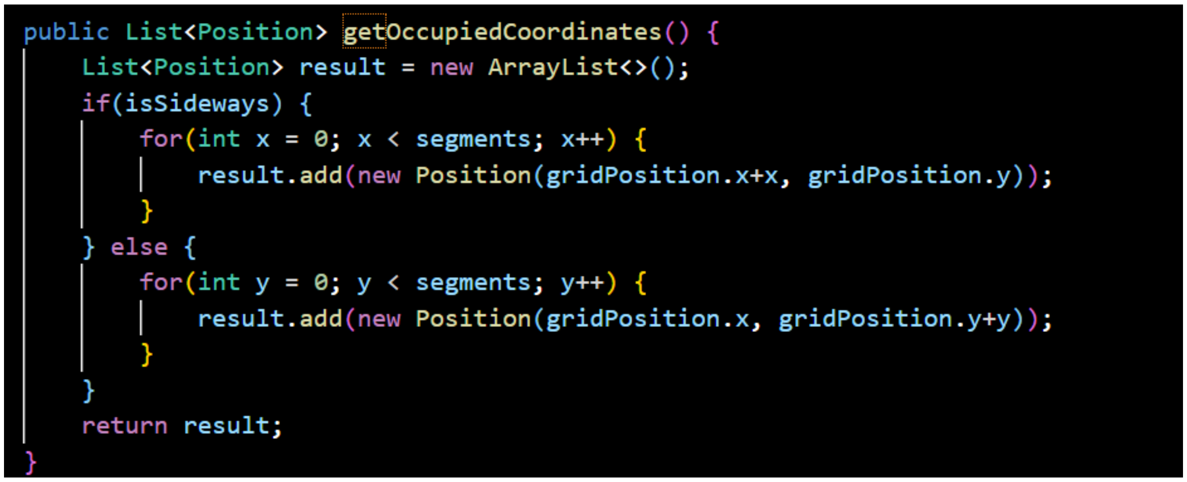
```java
public boolean canPlaceShipAt(int gridX, int gridY, int segments, boolean sideways) {
    if(gridX < 0 || gridY < 0) return false;

    if(sideways) {
        if(gridY > GRID_HEIGHT || gridX + segments > GRID_WIDTH) return false;
        for(int x = 0; x < segments; x++) {
            if(markers[gridX+x][gridY].isShip()) return false;
        }
    } else {
        if(gridY + segments > GRID_HEIGHT || gridX > GRID_WIDTH) return false;
        for(int y = 0; y < segments; y++) {
            if(markers[gridX][gridY+y].isShip()) return false;
        }
    }
    return true;
}
```

This algorithm checks whether the train can be placed at the given location. It is an algorithm that traverses the train's cells (O(segments)). Although this algorithm can be optimized a bit, it works very well in this case.

<u>Ship.java</u>

Algorithm:

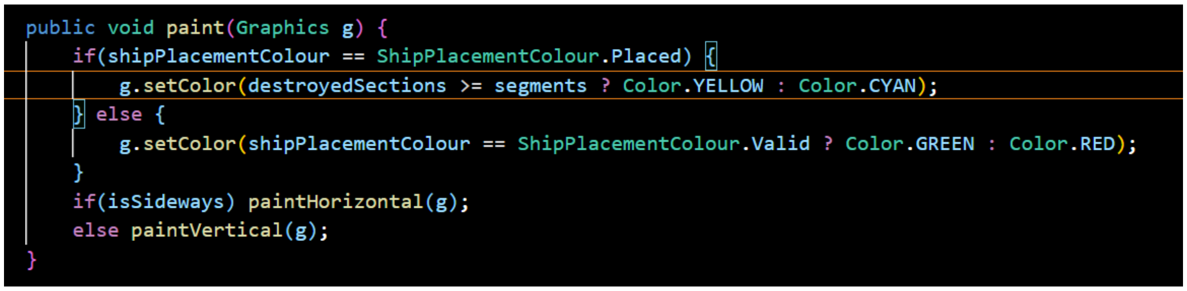
- GetOccupiedCoordinates():



```java
public List<Position> getOccupiedCoordinates() {
    List<Position> result = new ArrayList<>();
    if(isSideways) {
        for(int x = 0; x < segments; x++) {
            result.add(new Position(gridPosition.x+x, gridPosition.y));
        }
    } else {
        for(int y = 0; y < segments; y++) {
            result.add(new Position(gridPosition.x, gridPosition.y+y));
        }
    }
    return result;
}
```

  - This method iterates through each cell the ship occupies on the grid. If the ship is horizontal (isSideways is true), it creates a list of Positions starting from the initial position and incrementing horizontally.
  - If the ship is vertical (isSideways is false), it does the same but vertically.

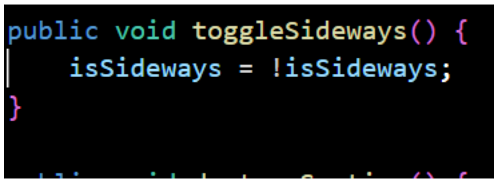
- Paint(Graphics g) method:



```java
public void paint(Graphics g) {
    if(shipPlacementColour == ShipPlacementColour.Placed) {
        g.setColor(destroyedSections >= segments ? Color.YELLOW : Color.CYAN);
    } else {
        g.setColor(shipPlacementColour == ShipPlacementColour.Valid ? Color.GREEN : Color.RED);
    }
    if(isSideways) paintHorizontal(g);
    else paintVertical(g);
}
```

  - This drawing algorithm changes the color and direction of the ship depending on its state (set valid, invalid, set).

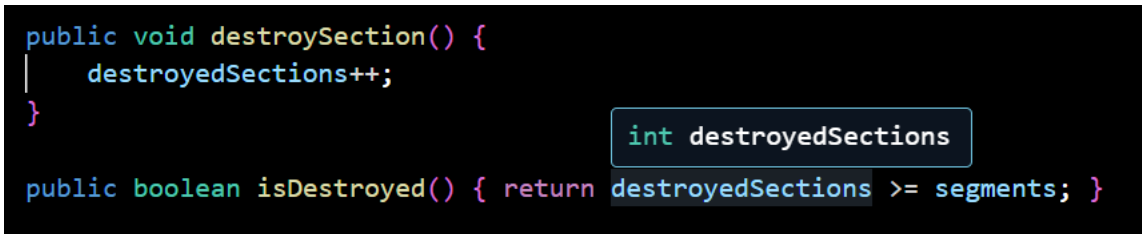
- ToggleSideways() method:



```
public void toggleSideways() {
    isSideways = !isSideways;
}
```

   - This method simply inverts the value of isSideways.

- DestroySection() and isDestroyed() methods:



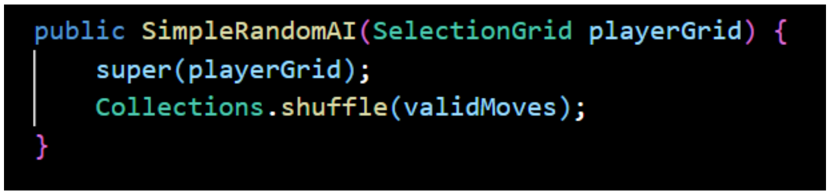
```
public void destroySection() {
    destroyedSections++;
}

public boolean isDestroyed() { return destroyedSections >= segments; }
```

   - When the ship is partially destroyed, destroyedSections is incremented. The isDestroyed() method checks whether the ship is completely destroyed by comparing destroyedSections with segments.

SimpleRandomAI.java

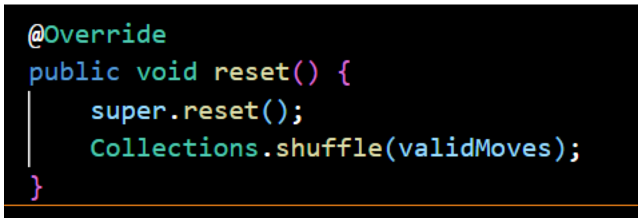
Algorithms:

- AI initialization (SimpleRandomAI constructor):



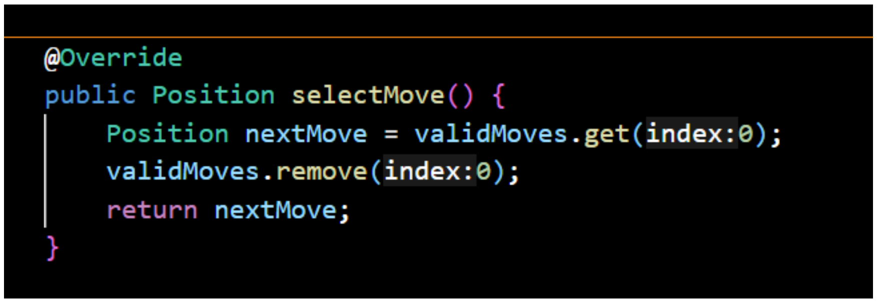
```java
public SimpleRandomAI(SelectionGrid playerGrid) {
    super(playerGrid);
    Collections.shuffle(validMoves);
}
```

  - When the SimpleRandomAI object is created, the Collections.shuffle(validMoves) method is called. This method randomly changes the order of the elements in the validMoves list — a list of valid moves that the AI can make.
  - Calling shuffle() ensures that each time the AI is initialized, it will have a list of moves in a random order, allowing the AI to make different moves each time.
- Reset() method:



```java
@Override
public void reset() {
    super.reset();
    Collections.shuffle(validMoves);
}
```

  - This method is called to "reset" the AI's state, restoring the initial values in the parent class (BattleshipAI), and then calling Collections.shuffle(validMoves) again to reshuffle the list of valid moves. This allows the AI to start over with a new sequence of moves, while still keeping the game from being repetitive.
- selectMove() method:

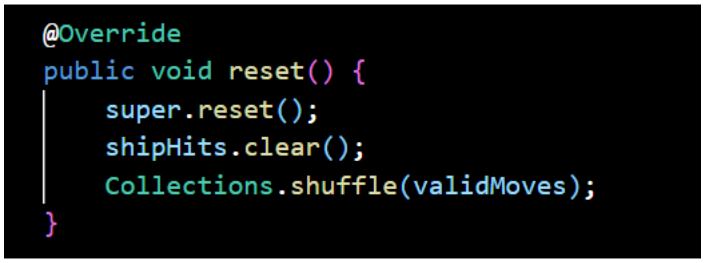


```java
@Override
public Position selectMove() {
    Position nextMove = validMoves.get(index:0);
    validMoves.remove(index:0);
    return nextMove;
}
```

- This method selects a position from the beginning of the validMoves list (the first position of the list) and then removes this position from the list, ensuring that the AI does not select that position again.
- This is how the AI chooses a simple move: select the first element, remove it, and make that move.

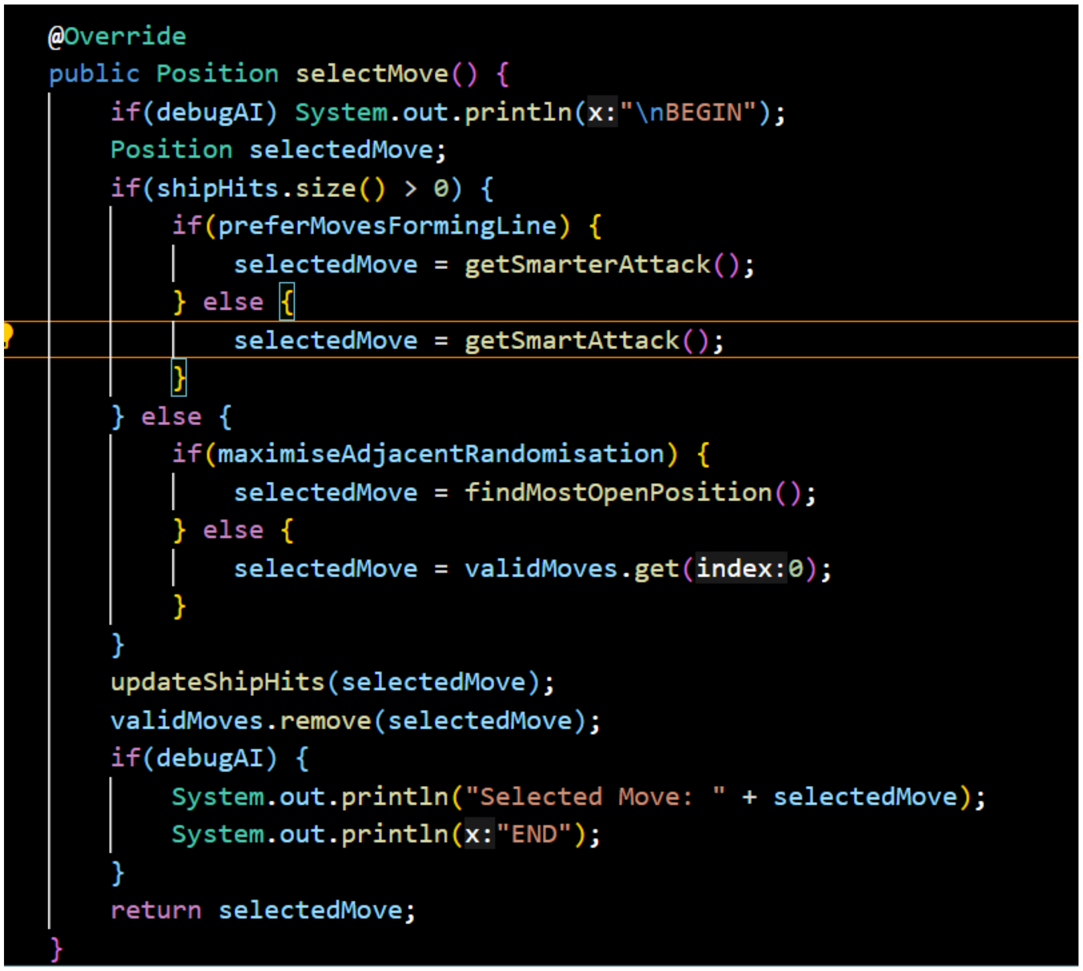
SmarterAI.java

1. List of shipHits (List<Position>):
   - Stores locations where ships have been hit but not completely destroyed.
   - Data structure: Use ArrayList because:
     - Accessing elements by index is fast (O(1)).
     - Convenient for traversing the list when dealing with hit locations.
2. Variable preferMovesFormingLine (boolean):
   - Adjusts how the AI prioritizes moves to form an attack path when a ship is found.
3. Reset method:



```java
@Override
public void reset() {
    super.reset();
    shipHits.clear();
    Collections.shuffle(validMoves);
}
```

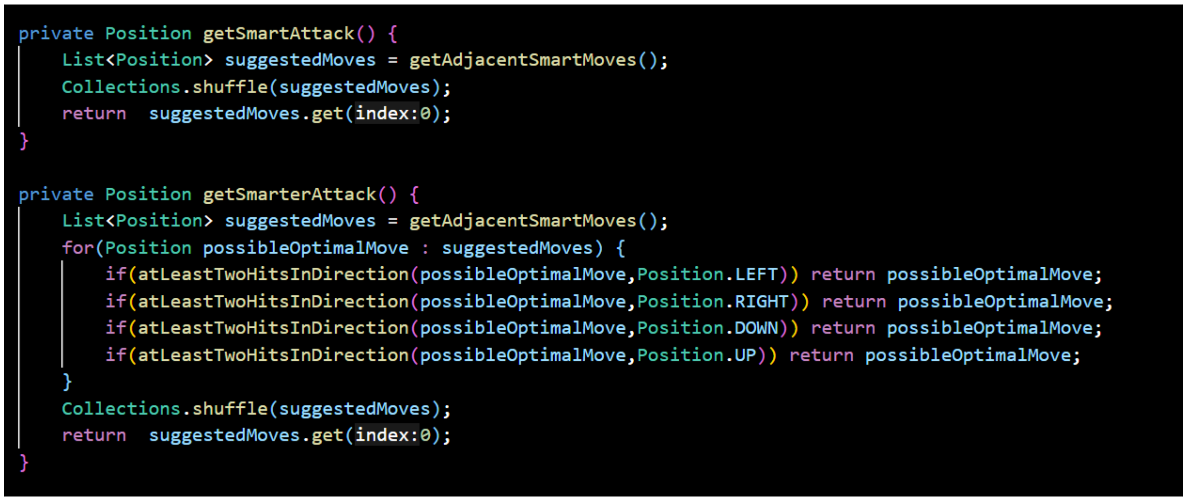
   - Resets the state of the AI when the game is restarted.
4. SelectMove method:



```java
@Override
public Position selectMove() {
    if(debugAI) System.out.println(x:"\nBEGIN");
    Position selectedMove;
    if(shipHits.size() > 0) {
        if(preferMovesFormingLine) {
            selectedMove = getSmarterAttack();
        } else {
            selectedMove = getSmartAttack();
        }
    } else {
        if(maximiseAdjacentRandomisation) {
            selectedMove = findMostOpenPosition();
        } else {
            selectedMove = validMoves.get(index:0);
        }
    }
    updateShipHits(selectedMove);
    validMoves.remove(selectedMove);
    if(debugAI) {
        System.out.println("Selected Move: " + selectedMove);
        System.out.println(x:"END");
    }
    return selectedMove;
}
```

- Check if shipHits is not empty (i.e. searching for ships).
- Use getSmarterAttack or getSmartAttack based on strategy.
- If no ships hit:
- Select a move using findMostOpenPosition or randomly.

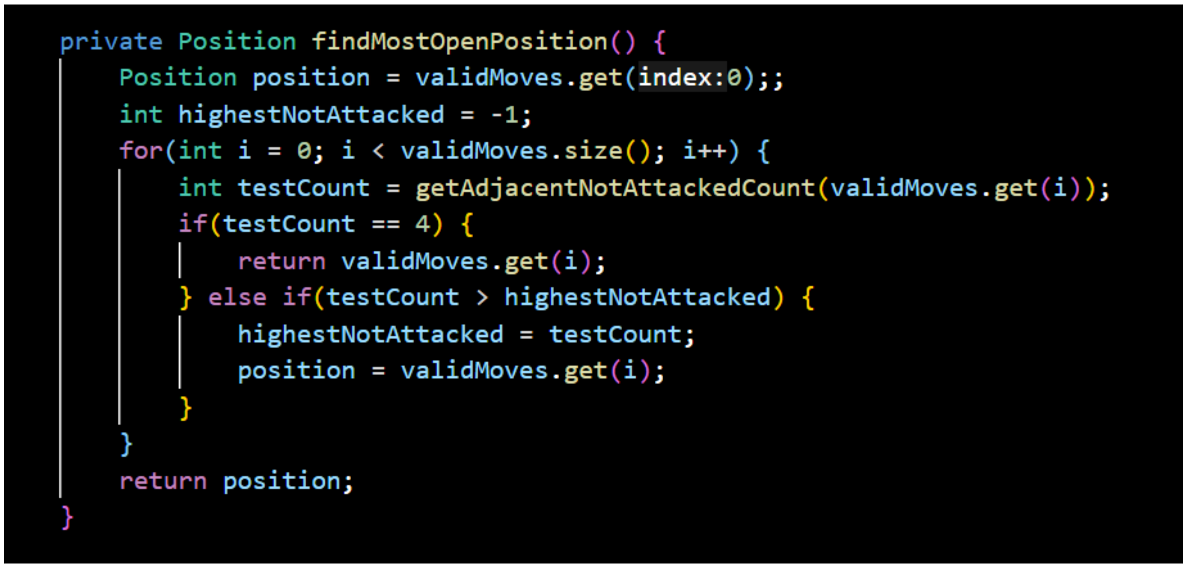
5. getSmarterAttack and getSmartAttack methods:



```java
private Position getSmartAttack() {
    List<Position> suggestedMoves = getAdjacentSmartMoves();
    Collections.shuffle(suggestedMoves);
    return  suggestedMoves.get(index:0);
}

private Position getSmarterAttack() {
    List<Position> suggestedMoves = getAdjacentSmartMoves();
    for(Position possibleOptimalMove : suggestedMoves) {
        if(atLeastTwoHitsInDirection(possibleOptimalMove,Position.LEFT)) return possibleOptimalMove;
        if(atLeastTwoHitsInDirection(possibleOptimalMove,Position.RIGHT)) return possibleOptimalMove;
        if(atLeastTwoHitsInDirection(possibleOptimalMove,Position.DOWN)) return possibleOptimalMove;
        if(atLeastTwoHitsInDirection(possibleOptimalMove,Position.UP)) return possibleOptimalMove;
    }
    Collections.shuffle(suggestedMoves);
    return  suggestedMoves.get(index:0);
}
```

- getSmartAttack: Get a list of valid moves around shipHits.
- Randomly select from this list.
- getSmarterAttack: Find the optimal move to form a path (search in 4 directions: UP, DOWN, LEFT, RIGHT).
- If not found, randomly select.

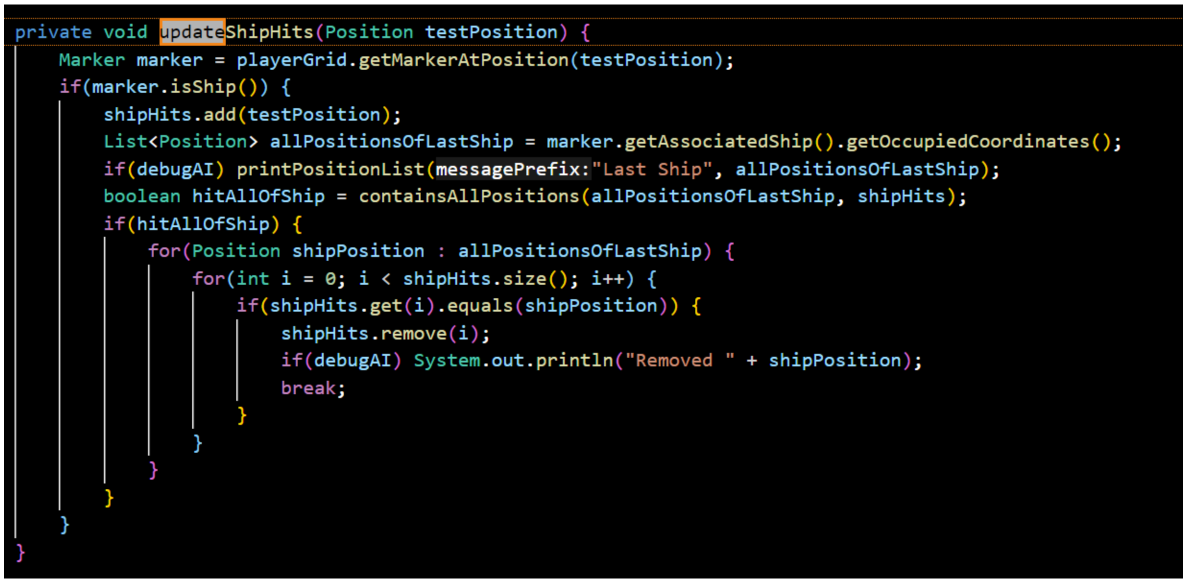
6. findMostOpenPosition method:



```java
private Position findMostOpenPosition() {
    Position position = validMoves.get(index:0);;
    int highestNotAttacked = -1;
    for(int i = 0; i < validMoves.size(); i++) {
        int testCount = getAdjacentNotAttackedCount(validMoves.get(i));
        if(testCount == 4) {
            return validMoves.get(i);
        } else if(testCount > highestNotAttacked) {
            highestNotAttacked = testCount;
            position = validMoves.get(i);
        }
    }
    return position;
}
```

- Evaluate each valid move to find the position with the most unattacked cells around.
- Choose the optimal position based on the number of unattacked cells.

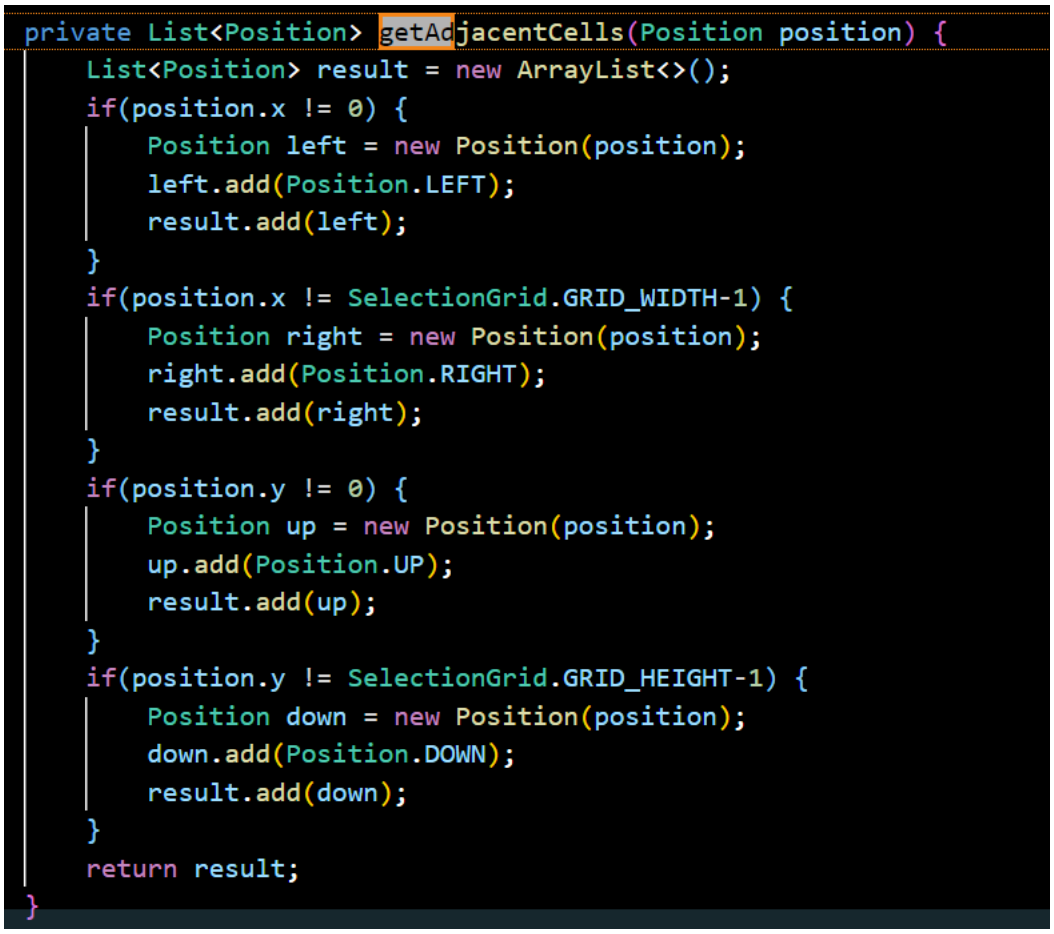
7. updateShipHits method:



```
private void updateShipHits(Position testPosition) {
    Marker marker = playerGrid.getMarkerAtPosition(testPosition);
    if(marker.isShip()) {
        shipHits.add(testPosition);
        List<Position> allPositionsOfLastShip = marker.getAssociatedShip().getOccupiedCoordinates();
        if(debugAI) printPositionList(messagePrefix:"Last Ship", allPositionsOfLastShip);
        boolean hitAllOfShip = containsAllPositions(allPositionsOfLastShip, shipHits);
        if(hitAllOfShip) {
            for(Position shipPosition : allPositionsOfLastShip) {
                for(int i = 0; i < shipHits.size(); i++) {
                    if(shipHits.get(i).equals(shipPosition)) {
                        shipHits.remove(i);
                        if(debugAI) System.out.println("Removed " + shipPosition);
                        break;
                    }
                }
            }
        }
    }
}
```

- If ship hit:
- Add to shipHits list.
- Check if all ships have been destroyed. If yes, remove ship data from list.

8. getAdjacentCells method:



```java
private List<Position> getAdjacentCells(Position position) {
    List<Position> result = new ArrayList<>();
    if(position.x != 0) {
        Position left = new Position(position);
        left.add(Position.LEFT);
        result.add(left);
    }
    if(position.x != SelectionGrid.GRID_WIDTH-1) {
        Position right = new Position(position);
        right.add(Position.RIGHT);
        result.add(right);
    }
    if(position.y != 0) {
        Position up = new Position(position);
        up.add(Position.UP);
        result.add(up);
    }
    if(position.y != SelectionGrid.GRID_HEIGHT-1) {
        Position down = new Position(position);
        down.add(Position.DOWN);
        result.add(down);
    }
    return result;
}
```

- Find adjacent cells around a position, remove cells beyond the boundary.

Main Activities:

1. Select Move (selectMove)
   - Main process when AI selects a move:
     - If the ship has been hit:
       - If preferMovesFormingLine = true: Select a move in the direction that forms a straight line.
       - Otherwise, select any adjacent cell around the hit coordinates.
     - If no ship has been hit:
       - If maximiseAdjacentRandomisation = true: Select a cell with multiple surrounding cells that have not been attacked.
       - Otherwise, select the first move in the validMoves list.
     - Update status:

        - Remove the selected move from validMoves.
        - Update the shipHits list if the ship has been hit.

2. Smart strategy:
    - getSmartAttack()
        - Select a random cell from adjacent moves around the coordinates in shipHits.
    - getSmarterAttack()
        - Attack in the preferred direction if there are at least 2 consecutive hits in a direction (left, right, up, down).
        - If there is no preferred direction, select randomly.
    - findMostOpenPosition()
        - Finds the cell in the validMoves list that has the most unattacked cells around it, increasing the chance of hitting a new ship.
3. Important support functions
    - getAdjacentCells(Position position)
        - Returns a list of adjacent cells (left, right, top, bottom) of a coordinate.
    - updateShipHits(Position testPosition)
        - If a ship is hit:
        - Adds the coordinates to shipHits.
        - Checks if all ships have been destroyed, removing all coordinates of that ship from shipHits.
        - If no ship is hit:
        - No update required.
    - atLeastTwoHitsInDirection(Position start, Position direction)
        - Checks if there are at least 2 consecutive cells hit in a direction from the starting coordinate.
    - containsAllPositions(List<Position> positionsToSearch, List<Position> listToSearchIn)
        - Check if all coordinates of a ship are in the shipHits list.
4. Debugging
    - debugAI:
        - When enabled, displays detailed information about the AI decision-making process, such as the selected move, the list of potential moves, the shipHits status.
    - printPositionList():
        - Supports printing a list of coordinates for easy tracking during debugging.
        - Illustrative workflow

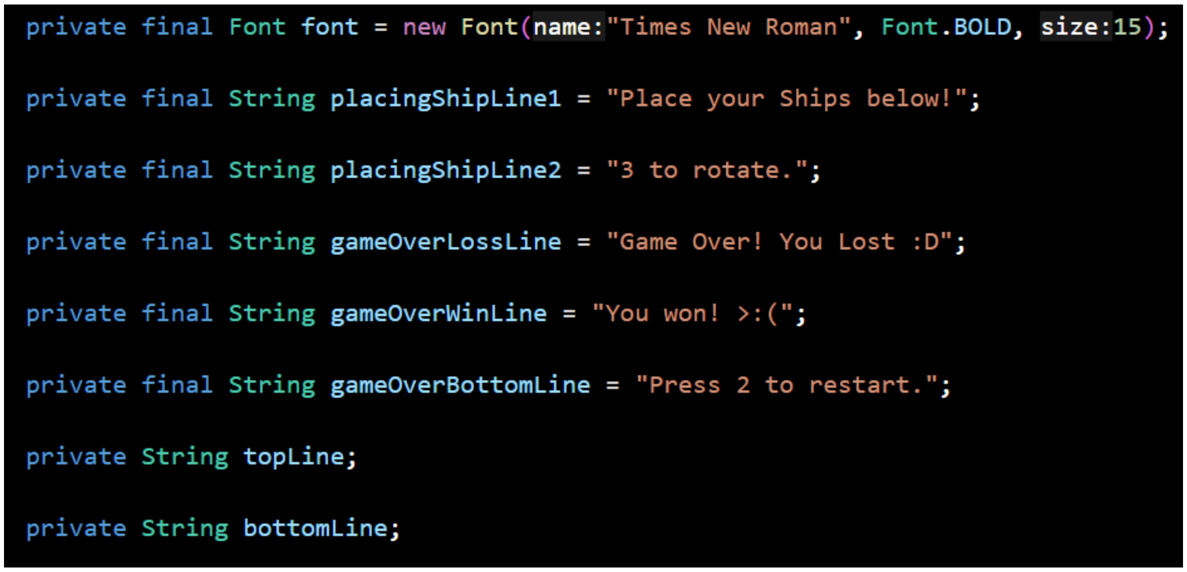
- Initial state:
    - validMoves: All squares on the board, randomly shuffled.
    - shipHits: Empty (AI has not hit any ship yet).

Example:

- Suppose AI hits at (3,3):
    - Add (3,3) to shipHits.
- On the next turn:
    - If preferMovesFormingLine = true: Find a direction with at least 2 consecutive hit squares (eg: (3,4)).
    - Otherwise, randomly select adjacent cells.
- When the ship is completely destroyed:
    - Remove all ship coordinates from shipHits.
- If the ship is not hit:
    - Continue searching for cells with many surrounding cells that have not been attacked (findMostOpenPosition).

StatusPanel.java

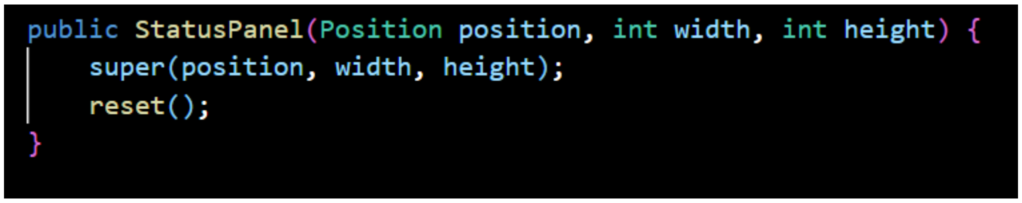
1/ Constant variables and status messages



```java
private final Font font = new Font(name:"Times New Roman", Font.BOLD, size:15);

private final String placingShipLine1 = "Place your Ships below!";

private final String placingShipLine2 = "3 to rotate.";

private final String gameOverLossLine = "Game Over! You Lost :D";

private final String gameOverWinLine = "You won! >:(";

private final String gameOverBottomLine = "Press 2 to restart.";

private String topLine;

private String bottomLine;
```

- Variables such as placingShipLine1, placingShipLine2, gameOverLossLine, gameOverWinLine, gameOverBottomLine are constants (final) used to store default messages in the game.

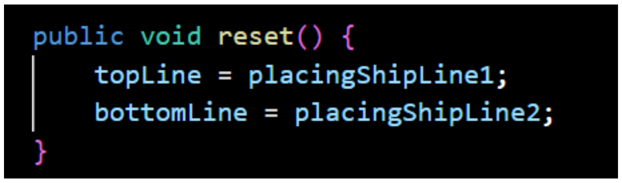
2/ Rectangle structure and interface position



```java
public StatusPanel(Position position, int width, int height) {
    super(position, width, height);
    reset();
}
```

- The StatusPanel class inherits from Rectangle, a rectangular data structure used to define the coordinates and dimensions of the interface displaying messages.
- Reasons for using Rectangle:
    - Rectangle simplifies the management of coordinates and dimensions (properties x, y, width, height).
    - Rectangle supports built-in methods such as contains(), intersects() to check for intersection, which can be useful if the functionality is extended.

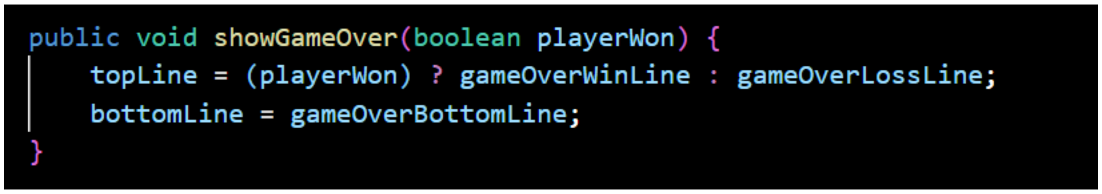
3/ Reset() method



```java
public void reset() {
    topLine = placingShipLine1;
    bottomLine = placingShipLine2;
}
```

- This method resets messages to the default state.
- Reasons for existence:
  - The method allows for a quick reset of the state in situations such as starting a new game.

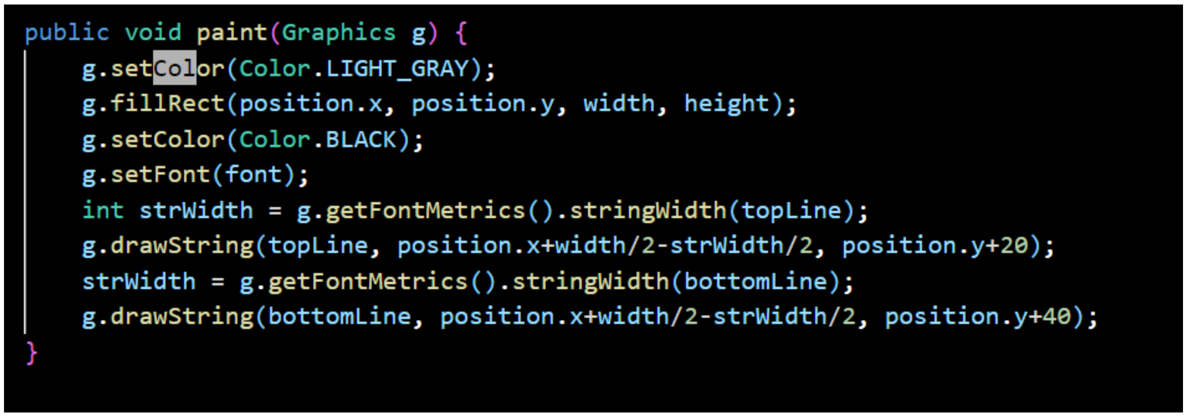
4/ showGameOver(boolean playerWon) method



```java
public void showGameOver(boolean playerWon) {
    topLine = (playerWon) ? gameOverWinLine : gameOverLossLine;
    bottomLine = gameOverBottomLine;
}
```

- This method checks the boolean variable playerWon to select the message to display when the game ends.

5/ paint(Graphics g) method



```java
public void paint(Graphics g) {
    g.setColor(Color.LIGHT_GRAY);
    g.fillRect(position.x, position.y, width, height);
    g.setColor(Color.BLACK);
    g.setFont(font);
    int strWidth = g.getFontMetrics().stringWidth(topLine);
    g.drawString(topLine, position.x+width/2-strWidth/2, position.y+20);
    strWidth = g.getFontMetrics().stringWidth(bottomLine);
    g.drawString(bottomLine, position.x+width/2-strWidth/2, position.y+40);
}
```

- This is the core method for drawing the interface, performing the following steps:
- Color the background.
- Display the message in the center of the panel by calculating the width of the character string (strWidth) to center it.
- Drawing algorithm:
  - Step 1: Fill the background with fillRect().

- Step 2: Get the width of the character string with g.getFontMetrics().stringWidth().
- Step 3: Draw the string at the center position according to the formula position.x + width/2 - strWidth/2.