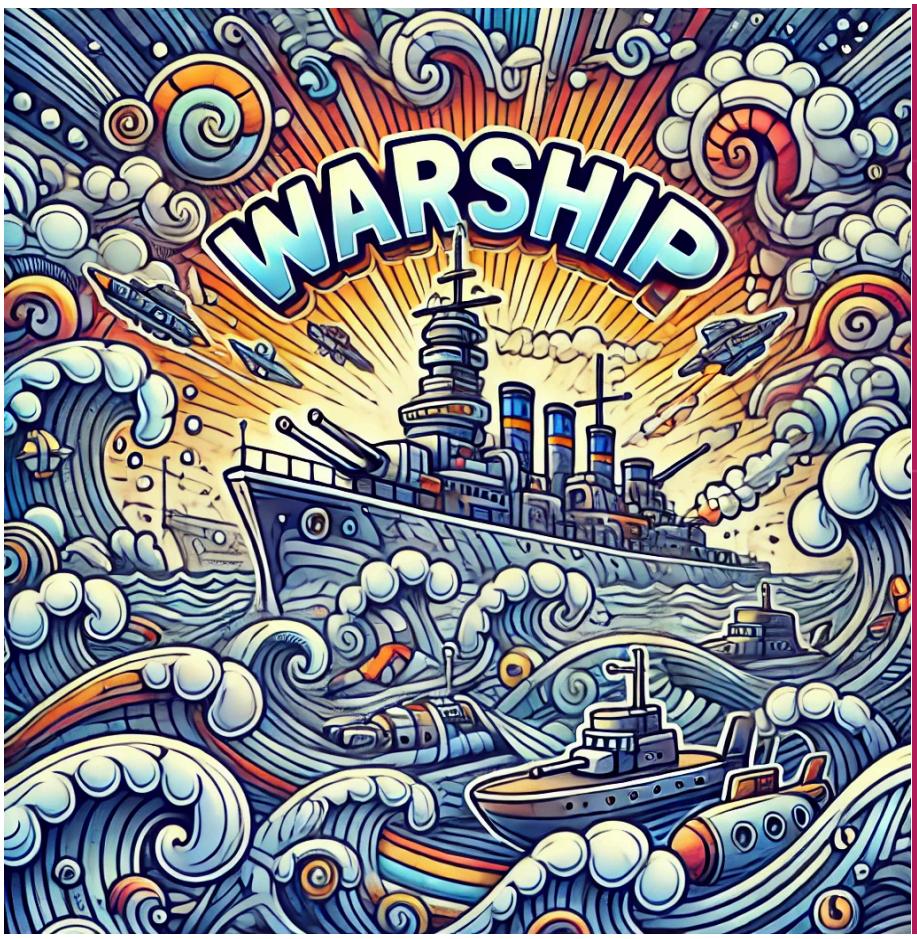


GAME DEVELOPER

Ho Thanh Tien

Nguyen The Vinh

Ngo Quang Hai

Ton Quang Tan

ADVISOR

Prof. Tran Thanh Tung

Nguyen Trung Nghia

Warship

FINAL PROJECT ORIENTED OBJECT PROGRAMMING

Warships (or **Battleships**) is a strategy-based guessing game designed for two players (in this case it is **Player vs CPU**). The game is played on ruled grids where each player's fleet of ships is positioned. These fleet locations remain hidden from the opponent. Players take turns to call out shots aimed at the opponent's ships. The primary objective is to strategically guess and destroy the entire fleet of the opposing player

Contents

CHAP 1:INTRODUCTION

1.1 Gaming in the Field:

In the ever-evolving domain of software engineering, the landscape is both familiar and distinct compared to other technological fields. What sets it apart is the convergence of diverse creative talents (art, music, acting, programming, etc.) working harmoniously to create unique experiences. Pursuing captivating gameplay often involves rigorous testing, feedback, and refinement cycles to ensure the final product resonates with players.

We are working on the Battleship game as our project for the "Object-Oriented Programming" course, a four-credit class and part of our degree program.

1.2 About the game project:

There are several board games similar to **Warships** (also known as **Battleships**). You can either play the physical version or find its digital version on various platforms like mobile or PC.

Our project involves developing a digital version of this beloved game on PC. By doing this project, we aim to:

- Improve our software development cycle.
- Enhance our Java programming skills.
- Apply theoretical knowledge from our classes to a practical, engaging project.

This game project not only serves as a means to practice and refine our technical skills but also provides an opportunity to understand and implement the principles of game design and user interaction. We look forward to creating a compelling and enjoyable digital version of Battleship that can be appreciated by players of all ages.

1.3 Our Battleship game:

We will add more features to our Battleship game to make the User get more excited while enjoying the game:

- o "Random button" for those don't want to arrange their ships
- o "Theme" for game
- Add some images and animations that fit the theme
- Apply Stack, singleton pattern, adapter pattern and some other OOP skills.

1.4 References:

- Images from DaleE (Open AI), google.com
- https://stackoverflow.com

1.5 Developer team:

Homies team is a Data Structure Algorithm project team. We have 4 members come from International University, Computer Science major:

Name	UID	Contribute
Hồ Thành Tiến	ITITIU22157	Report / CPU / Tester/ Fix Bug
Tôn Quang Tấn	ITITIU22205	Report / UML, Use Case diagram / Classes and Objects for location
Nguyễn Thế Vinh	ITITIU22184	Report / Gameplay Function / UI - UX
Ngô Quang Hải	ITCSIU22276	Report / UI-UX / Frame

CHAP 2:
SOFTWARE REQUIREMENTS

2.1 What we have:

- **Concept:** A simplified Battleship game with core mechanics such as ship placement, attack phases, and grid-based strategy.
- **Platform:** PC-only game designed to run on low-spec computers ("potato specs").

❖ **Development Tools:**

➢ **Java Environment:** Core programming language and platform.

➢ **Visual Studio Code:** Primary code editor.

➢ **ChatGPT:** Used for generating promotional materials, such as posters and images.

2.2 What we want:

Engaging Gameplay:

- The game should be challenging enough to hold the player's attention and allow them to take a break from daily routines.
- Players should feel a sense of accomplishment when defeating the AI or achieving milestones.

Core Features:

- **Grid-Based Gameplay:** A 10x10 grid where players place ships and strategize their moves.
- **AI Opponent:** Multiple difficulty levels to ensure a fair challenge for all players.
- **Randomized Ship Placement:** For both the player and AI, with an option for manual placement for advanced players.
- **Hit/Miss Feedback:** Visual indicators for hits and misses to enhance gameplay clarity.

User Experience:

- Simple, intuitive interface.
- Lightweight application optimized for low-performance hardware.

2.3 Working tools, platform:

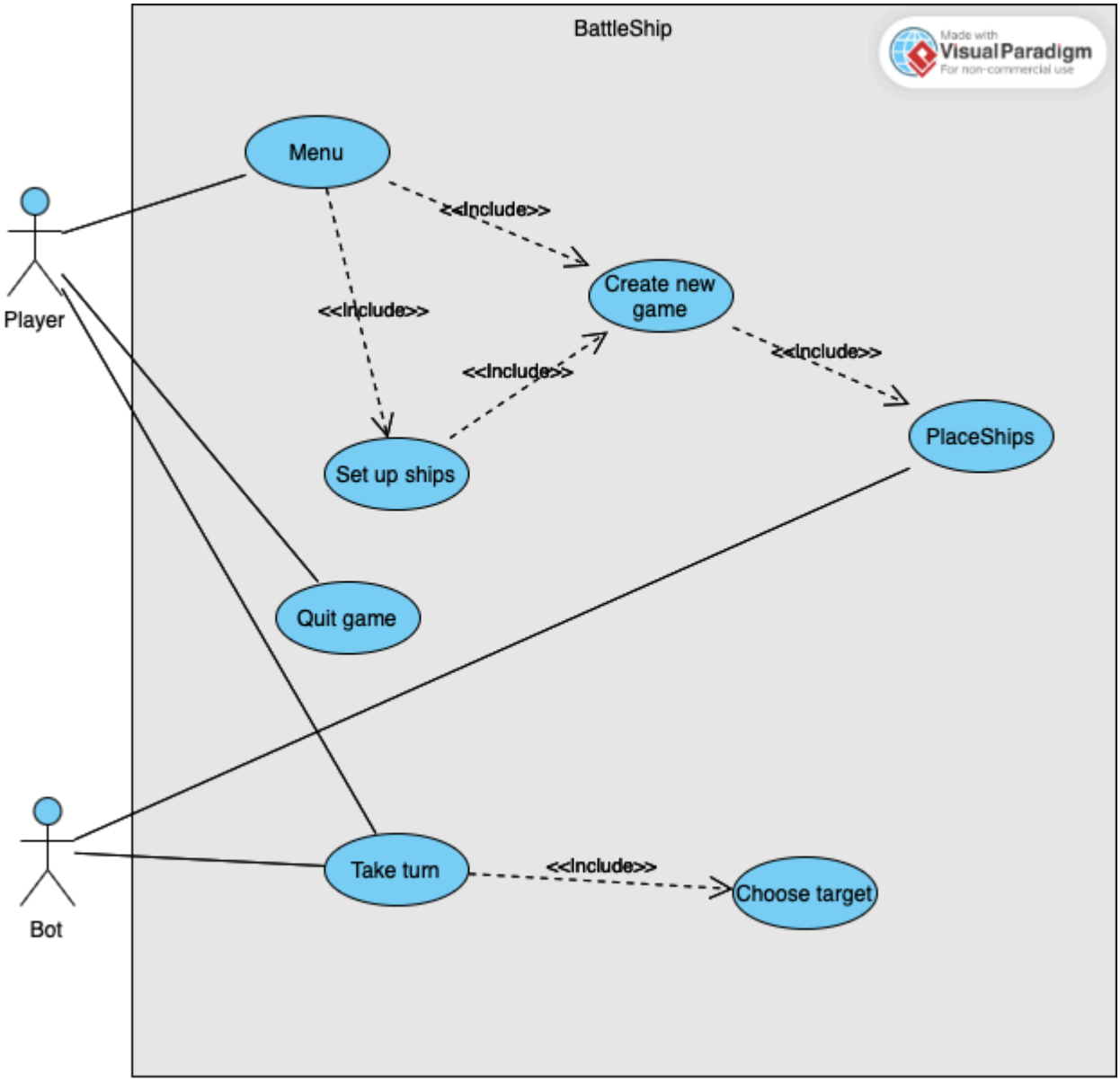
- **Programming Language:** Java
- **Development Environment:** Visual Studio Code

2.4 Use Case Scenario

We have created the use cases based on the UX view of the game.

UC	PLAY	Play the game
		Resume the game
		Exit the game
	SETTINGS	Set preferences
	PLACE SHIPS	Position ships on the grid
	TAKE TURN	Select target and attack
	QUIT	Exit the game

2.5 Use Case diagram



This diagram represents the primary functionalities and interactions between actors (Player and Bot) and the system (Battleship Game). It outlines the core use cases and their relationships, providing an overview of the gameplay flow and setup process.

Actors:

- **Player:**

Primary user interacting with the system.

Controls game setup, turns, and settings.

- **Bot**

Secondary automated player that takes turns during gameplay.

Use Cases and Descriptions:

- **Menu:**

    Provides options to load, create, or quit the game.

    Includes access to settings for customization.

- **Create New Game:**

    Starts a new game session.

    Includes:

        "PlaceShips" for setting up ships on the board.

        "Set up ships" for configuring game preferences.

- **Set up shipss:**

    Provides options to customize the game, such as ships location.

- **PlaceShips:**

    Enables players to position ships on the grid before starting the game.

- **Quit Game:**

    Allows the player to exit the game session.

- **Take Turn:**

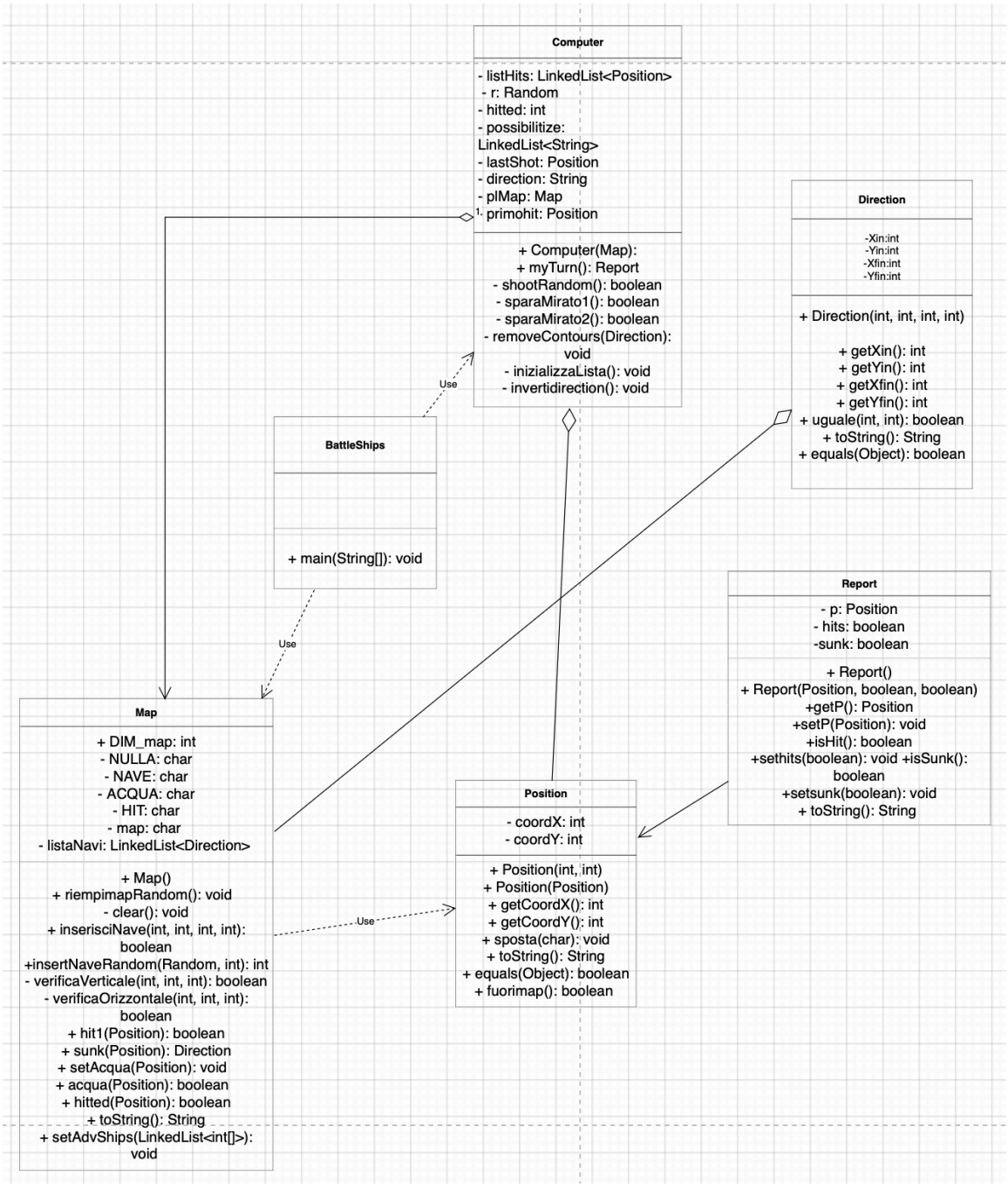
    Represents gameplay interaction where the player or bot takes a turn.

    Includes: "Choose target" for selecting a position to attack.

- **Choose Target:**

    Allows selection of a specific coordinate to target during a turn.

2.6 Class diagram



1. Computer

- Attributes:

    listHits: A linked list to store positions of hits.

    r: A random object for generating random values.

    hitted: An integer to count hits.

    lastShot: A Position object to track the last shot.

    direction: A string to track the shooting direction.

    piMAP: A Map object representing the game map.

    primoHit: A Position object marking the first hit of a ship.

- Methods:

    Computer(Map): Constructor to initialize the computer player with a map.

    myTurn(): Returns a Report, likely to represent the outcome of the turn.

    shootRandom(): Shoots randomly; returns a boolean indicating success.

    sparaMirato(): Executes targeted shots in different ways.

    removeContours(Direction): Removes potential move directions.

    inizializzalista(): Initializes a list, possibly possibilita.

    invertidirection(): Reverses the direction of shooting.

2. Direction

- Attributes:

    xInit, yInit, xFin, yFin: Integers for coordinates.

- Methods:

    Direction(int, int, int, int): Constructor for initializing directions.

    Getter methods for the coordinates.

uguale(int, int): Checks equality of direction values.

toString() and equals(Object): Standard object methods.

3. Map

- Attributes:

  DIM_map: Size of the map.

  Constants like NULLA, NAVE, ACQUA, HIT for map states.

  map: The map's state.

  listaNavi: A linked list of Direction objects, likely storing ship locations.

- Methods:

  Constructor Map().

  riempimapRandom(): Fills the map randomly with ships.

  Methods for inserting ships (inserisciNave), verifying placement (verificaVerticale, verificaOrizzontale), and checking hits (hitted, hit).

  Methods for managing ship states (sunk, setAcqua, setAdvShips).

4. Position

- Attributes:

  coordX, coordY: Integer coordinates of a position.

- Methods:

  Constructor Position(int, int) and a copy constructor.

  Getters for coordinates.

  sposta(char): Moves the position in a certain direction.

  Utility methods for checking equality (equals) and boundary constraints (fuorimapo).

5. Report

- Attributes:

p: A Position representing where the event occurred.

hits: Boolean indicating whether it was a hit.

sunk: Boolean indicating if a ship was sunk.

- Methods:

  Constructor Report(Position, boolean, boolean).

  Getter and setter for Position, hits, and sunk.

6. BattleShips

Purpose:

Contains the main entry point for the program (main(String[])).

Likely initializes and manages the game logic, connecting the components.

Relationships:

1. Computer Uses Map: The Computer class interacts with Map for tracking the game state.
2. Map Has Position and Direction: Positions and directions define the game grid and ship placement.
3. Report Contains Position: Reports are generated based on positions.
4. BattleShips Manages the Game: It ties all the components together.

CHAP 3: DESIGN & IMPLEMENTATION

RUNNING PLATFORM: Intellij

For UI design, we used 3 xml files: misc.xml, modules.xml and workspace.xml

misc.xml: it ensures the project uses JDK 23 features and libraries. Set version to 2, specifies the use of JDK type and set default configuration. Most important, it specifies the output directory for compiled artifacts (out).

modules.xml: indicate the project use version 4 in intellij, it tells IntelliJ IDEA where to find the .iml file that defines the module settings.

workspace.xml: configures the auto-import behavior, control how conflicts and dialogs are handled, tracks recently used file templates, ... In short, we make it to handle version control and manage templates and tasks.

FRAME: Most frames will import the following libraries:

```
import java.awt.Cursor;
 import java.awt.Dimension;
 import java.awt.Toolkit;
 import java.awt.event.ActionEvent;
 import java.awt.event.ActionListener;
 import java.awt.event.KeyEvent;
 import java.awt.event.KeyListener;
 import java.util.LinkedList;
 import java.util.StringTokenizer;

 import javax.swing.ImageIcon;
 import javax.swing.JButton;
 import javax.swing.JFrame;
 import javax.swing.JLabel;
 import javax.swing.JOptionPane;
 import javax.swing.JPanel;
 import javax.swing.Timer;
```

Along with all classes:

```
import battleship.Computer;
```

```
import battleship.Map;
import battleship.Ship;
import battleship.Position;
import battleship.Report;
```

CLASSES AND CODE EXPLANATION:

3.1 UI

UIJPanelBG Class

Key Component: paintComponent(Graphics g)

```
public void paintComponent(Graphics g) {
    g.drawImage(immagine, 0, 0, null);
}
```

- This method is responsible for drawing the background image (immagine) on the panel.
- It overrides the paintComponent method of JPanel, ensuring that the background image is rendered correctly whenever the panel is displayed or updated.
- Without this, the panel would lack a visual background, which is essential for creating an immersive game environment.

Key Component: createImageIcon(String path)

```
public static ImageIcon createImageIcon(final String path) {
    InputStream is = ImageLoader.class.getResourceAsStream(path);
    try {
        int length = is.available();
        byte[] data = new byte[length];
        is.read(data);
        is.close();
        return new ImageIcon(data);
    } catch (IOException e) {
        return null;
    }
}
```

- This method loads an image resource from the specified path and creates an ImageIcon object.
- It's crucial for dynamically setting background images and other graphical elements.

- The use of InputStream ensures compatibility with packaged resources, such as those within .jar files.

UImapnel Class

Key Component: drawShip(int[] dati)

```
void drawShip(int[] dati) {
    int x = dati[0];
    int y = dati[1];
    int dim = dati[2];
    int dir = dati[3];
    ImageIcon shipDim1orizz = new ImageIcon(
            getClass().getResource("/res/images/shipDim1orizz.png"));
    ImageIcon shipDim1vert = new ImageIcon(getClass().getResource("/res/images/shipDim1vert.png"));
    if (dim == 1) {
        buttons[x][y].setEnabled(false);
        if (dir == 0)
            buttons[x][y].setDisabledIcon(shipDim1orizz);
        else if (dir == 1)
            buttons[x][y].setDisabledIcon(shipDim1vert);
    } else {
        ImageIcon shipHeadLeft = new ImageIcon(
                getClass().getResource("/res/images/shipHeadLeft.png"));
        ImageIcon shipHeadTop = new ImageIcon(
                getClass().getResource("/res/images/shipHeadTop.png"));
        ImageIcon shipBodyLeft = new ImageIcon(
                getClass().getResource("/res/images/shipBodyLeft.png"));
        ImageIcon shipBodyTop = new ImageIcon(
```

```
        getClass().getResource("/res/images/shipBodyTop.png"));
    ImageIcon shipFootLeft = new ImageIcon(
        getClass().getResource("/res/images/shipFootLeft.png"));
    ImageIcon shipFootTop = new ImageIcon(
        getClass().getResource("/res/images/shipFootTop.png"));
```

- This method visually represents ships on the grid by setting custom icons for buttons.
- It accounts for the ship's size (dim) and orientation (dir), ensuring accurate placement.
- It's vital for gameplay, as it provides feedback to the player about the location and orientation of ships.

Key Component: **Grid Initialization in Constructor**

```
buttons = new JButton[numC][numC];
for (int i = 0; i < numC; i++) {
    for (int j = 0; j < numC; j++) {
        buttons[i][j] = new JButton(gray);
        buttons[i][j].setSize(dimC, dimC);
        sea.add(buttons[i][j]);
        buttons[i][j].setCursor(cursorHand);
        buttons[i][j].setBorder(border:null);
        buttons[i][j].setOpaque(isOpaque:false);
        buttons[i][j].setBorderPainted(b:false);
        buttons[i][j].setContentAreaFilled(b:false);
        buttons[i][j].setBounds(oroff, veroff, dimC, dimC);
        oroff += dimC + 2;
    }
    veroff += dimC + 2;
    oroff = 1;
}
```

- This initializes the 10x10 grid of buttons, which represent the game board.
- Each button corresponds to a cell in the grid, and their visual properties (e.g., size, transparency) are configured here.
- The grid is the core interactive element of the game, allowing players to place ships and make moves.

UIManagePanel

Key Component: Ship Selection and Orientation

```
ship = new JRadioButtonMenuItem[4];
radioButtonShip = new ButtonGroup();
direction = new JRadioButton[2];
ButtonGroup radioButtonDirection = new ButtonGroup();
```

- Ship allows users to select ships of different sizes visually, represented by radio buttons with ship icons.
- Direction enables players to choose the ship's orientation (horizontal or vertical).
- These components are essential for setting up the game, as they define how and where ships are placed on the grid.

Key Component: Action Buttons (random, reset, play)

```
random = new JButton(randomImg);
reset = new JButton(resetImg);
play = new JButton(playImg);
```

- **random:** Randomly places ships on the grid, providing convenience for quick gameplay.
- **reset:** Clears the grid, allowing players to start over with ship placement.
- **play:** Transitions to the game phase, becoming active once ship placement is complete.
- These buttons streamline user interaction, ensuring a smooth transition between setup and gameplay.

UIStatPanel Class

Key Component: Ship labels

```
        ships = new JLabel[10];
    for (int i = 0; i < ships.length; i++) {
        ships[i] = new JLabel();
        this.add(ships[i]);
    }
```

- Ships represents the fleet of ships, displayed as icons to indicate their status (e.g., available or sunk).
- This visual feedback is crucial for tracking progress during the game.

Key Component: Ship Icon and Positioning

```
ships[0].setIcon(new ImageIcon(getClass().getResource(name:"/res/images/ship4.png")));
ships[0].setBounds(25, 5, 135, 35);
```

- Each ship icon corresponds to a specific ship size and is positioned strategically on the panel.
- These icons provide an intuitive overview of ship availability, enhancing the player's situational awareness.

1. **FrameBattle implements ActionListener, KeyListener:**

· Create two new UIMapPanel : playerPanel and cpuPanel; using this class allow us to create two custom-imaged panels for player and cpu(computer) separately, using data image from our sources.

· Jframe and Jpanel are added to make frame and buttons, code will not allow the frame to change size however.

· Two UIStatPanel statPlayer and statCPU, will appear on the same background as UIMapPanel ones, they are to shown the number of ships lost or still on the battle.

· Create targetPanel for interacting with the player's map to aim and fire at the CPU's ships, providing a visual and interactive component for the targeting phase of the game.

· Coding makes sure no Panel overlaps on other Panels or Background.

- We use StringBuilder instead of String.
- We created crusol to interact with the panel data.

Methods:

FrameBattle (LinkedList<int[]> playerShips, Mappa map): Create map objects to store ship position , use map method fillMapRandom() to generate ship on the map with random positions. Set size for frame, set position for UIMapPanel and UIStatPanel variables, which metioned before exist alongside with each other. Adding keylistenr, centering the frame. Inside, there is a nested loop for actionListener when user click in the frame, the position[i][j] will be sent to actionCommand, and a for loop iterates through playerships to draw ship for playerPanel.

setBox(Report rep, boolean player):

- A void method
- Get result from player click on the cpuPanel
- Run a boolean method inside based on the position gotten, if hit the ship return specified icon, if miss, just return normal water icon

  The method create two int variable x and y to store position and then run through if statements to determine miss or hit state.

@override *actionPerformed*(ActionEvent e): The method handles the logic of processing a player's action in a game where they interact with a UI by clicking buttons.

- Check if its player turn or cpu turn, if its cpu, end the method.
- Retrieve the source of player click and create a new position from it, then check if hit or miss
- A report object is created to collect the position, run a method to see if miss or hit
- If it it, a method is run to check if it sunk(which typicall require multiple hits) if it sunk case, number of ship is decrease by one
- A nested if statement to check if all ships are sunk, if they are then a JoptionPanel appear with two button “New Game” or “Exit”, if exist end the code, if other, creates a new FrameManageship instance.

setSunk: is used to mark a ship as sunk on the player's map and update the user interface (UI) accordingly

- The method first checks which directions are possible to explore based on the current position p
- find the full length of the ship. Starting from the hit position p, it checks in one of the possible directions (North, South, West, or East). If it doesn’t find any other parts of the ship in those directions, it assumes the

ship is only one square and marks it sunk.

If it finds a ship part, it continues searching in that direction until it reaches the edge of the map or an empty space

- Once the full length of the ship has been determined, it marks each part of the ship as "sunk" in the UI. The method disables the buttons and changes their icon to a "wreck" image, signaling the ship has been sunk.
- dim int value tracks the number of squares that make up the sunk ship.
- After marking all parts of the ship, deleteShip(dim, statPlayer) is called to remove the ship from the player's state

  The method assumes that ships are either placed horizontally or vertically. The directions and checks are based on this assumption. If the ship is of length 1, it's immediately marked and deleted.

deleteShip: is used to disable a ship from the player's fleet(UIStatPanel) once it has been sunk.

- The method uses a switch statement to handle different cases depending on the size of the ship (dim). It disables a specific ship from the fleet by setting its setEnabled(false) based on its size.
- Depends on the value it get, the ship in the index of the ship Array will be removed

keyPressed, keyReleased, keyTyped : The provided code is an implementation of the KeyListener interface, which listens for key events (press, release, and typing)

GestoreTimer implements ActionListener: The class implements ActionListener and is designed to handle periodic actions triggered by a timer.

- Allow pause between turns, preventing immediate actions and allowing for the CPU's actions to complete.
- retrieve the result of the CPU's turn, which returns a Report object containing information about the shot
- If report indicate sunk state, the number of ship decrementes
- If the shot was a hit, the CPU gets another turn, and the timer is started again.

drawTarget: is responsible for drawing a target on a game board by setting the position and visibility of the target element

- The setBounds(x, y, width, height) method sets the position and size of the target component. 50, 50 is the width and height of the target.
- adds the target to a container (targetPanel). This is where the target will be displayed in the user interface.

- repaint() is called to update the UI and display the target in its new position.

2. **FrameManageship extends JFrame implements ActionListener, KeyListener**: The class manages the game's UI for placing ships, enabling random ship placement, and transitioning to the gameplay stage.

Variables:

- NUM_SHIP: The total number of ships a player must place.
- playerShips: A LinkedList holding the placed ships' positions and details (x, y, size, direction).
- shipInsert: Tracks the number of ships placed.
- counterShip: An array tracking the count of each type of ship
- choosePan and mapPanel: UI panels for managing ships and the map grid.

Methods:

Constructor: *FrameManageship*()

- invoke a method from father class(JFrame) to print out a String
- create a new map object
- setDefaultCloseOperation
- setResizable(false)
- UIMapPanel (mapPanel): Represents the grid where ships are placed. Displays UI controls like buttons for random placement, reset, and starting the game.
- Loops through all buttons in the UIMapPanel and assign ActionListener
- Initializes an empty LinkedList<int[]> (playerShips) to store the player's placed ship details.

actionPerformed(ActionEvent e): is an implementation of the ActionListener interface. It processes user interactions, specifically actions triggered by buttons or other components that generate ActionEvent objects.

- A new JButton object is created with name “source” served to retrieve the components when user click the button
- A string variable “test” is created, which will get corresponding string if the correct components are met.
- "reset": The game board is reset.

- "random": Ships are placed randomly on the board.
- "battle": The game transitions to the battle phase.
- A new StringTokenizer object is created with an action command int x and int y. The coordinates will be extracted.
- Run a nested for loop with choosePan array's length to pick which ship type is currently selected from the choosePan panel.
- For this, use a switch statement with different cases fro corresponding ship sizes
- Finally, map.insertShip() attempts to place the ship on the board at the specified location and orientation.
- If all ships are placed, updates the finish flag, disables further ship placement options, and enables the "battle" button.
- this.requestFocusInWindow(): Ensures the window regains focus after handling the action.

random(): automates the placement of ships on the game board. It ensures that all ships are placed in random positions

- First, if all ships (NUM_SHIP) have already been placed, the game board is reset to allow new random placement.
- Create new random variable "r", new integer array "data"
- r is created for generating random positions and orientations.
- data hold following attributes of the ship: x,y coordinates size and direction
- A nested for loop is created to generate all ships, Calls map.insertShipRandom() to generate valid random placement for the ship based on its size.
- If no error, then set the shipInsert counter to NUM_SHIP to indicate all ships are placed, and enable "battle" button.
- After that ensure that ship selection options and direction buttons in the UI are disabled.

reset(): is designed to reset the game state to its initial configuration. It clears any existing ship placements, re-enables UI components, and sets up the board for new ship placements.

- Iterates over the entire board grid (Map.DIM_MAP) and re-enables all buttons, allowing the user to place ships again.

- finish: Set to false to allow new ship placements

- Re-enables ship type selection buttons and direction selection buttons in the UI, allowing the user to choose ship types and placement orientations.
  - counterShip: Resets the counters for each ship type to their default values

battle(): method to visualize the Frame

keyPressed(KeyEvent arg0): handles keyboard input.

- Set all characters to lower case
- Captures the key code of the pressed key
- If character is “s” triggers the random() method to randomly place all ships on the board, also start the game
- Special keys like Delete or Backspace trigger restart() method, which restart the game
- Pressing Escape exits the application using System.exit(0).

3. **FrameMenu extends JFrame implements ActionListener, KeyListener:** is designed as a graphical menu interface for the game

New crusor object is created, allow interaction between user

Method:

Constructor FrameMenu():

- setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
- Fixing size
- Sets the icon for the application window using an image from the resources folder.
- Calculates the center position of the screen and sets the frame location accordingly.
- Creating new UIJPanelBG object, serving as the background for the game, The background spans the entire frame. Use getResource() from one of our images

4. **FrameSplashscreen extends JFrame:** is a splash screen designed for the Battleship game

Method:

- Exits the application when the window is closed.

- Fixing size
- Icon: Sets the application icon using a resource image
- Null Layout: Allows precise manual placement of components.
- The panel covers the whole splash screen
- The loadingLabel is positioned at the bottom-right corner (560x310) and overlays the splash panel.
- Ensures the splash screen is displayed when the object is created.
- Add functionality to load the main menu after the splash screen

3.2 Direction and Position

a. Class: Direction

Purpose:
Represents a ship on the game board, defined by its start and end coordinates.

Key Methods:

- **Constructor:**

```
public Direction(int xin, int yin, int xfin, int yfin) {
    this.xin = xin;
    this.yin = yin;
    this.xfin = xfin;
    this.yfin = yfin;
}
```

-

Initializes the ship with start (xin, yin) and end (xfin, yfin) coordinates.

- **Coordinate Check:**

```
public boolean uguale(int x, int y) {
    if (x <= xfin && x >= xin && y <= yfin && y >= yin) {
        return true;
    }
    return false;

}
```

Checks if a given position (x, y) is within the boundaries of the ship, used to determine

hits.

- **Equality Comparison:**

```java
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Direction other = (Direction) obj;
    if (xfin != other.xfin)
        return false;
    if (xin != other.xin)
        return false;
    if (yfin != other.yfin)
        return false;
    if (yin != other.yin)
        return false;
    return true;
}
```

Compares two ships to check if they occupy the same area on the grid.

- **String Representation:**

```java
public String toString() {
    return xin + "–" + yin + "  " + xfin + "–" + yfin;
}
```

Converts the ship's position into a readable string format.

b. Class Position

- **Purpose:**
  Represents a specific coordinate on the grid and handles movement.

- **Key Methods:**

  Constructor:

```java
public Position(int coordX, int coordY) {
    this.coordX = coordX;
    this.coordY = coordY;
}

public Position(Position p){
    this.coordX = p.coordX;
    this.coordY = p.coordY;
}
```

Initializes a position with the given X and Y coordinates.

Move Position:

```java
public void sposta(char dove){
    switch(dove){
        case 'N':
            coordX--;
            break;
        case 'S':
            coordX++;
            break;
        case 'O':
            coordY--;
            break;
        case 'E':
            coordY++;
            break;
    }
}
```

Moves the position in one of four directions—North ('N'), South ('S'), East ('E'), or West ('O'). Allows movement of positions, potentially for navigation or targeting purposes.

Boundary Check:

```
public boolean fuoriMappa(){
    if(coordX>=Mappa.DIM_MAPPA||coordY>=Mappa.DIM_MAPPA||coordX<0||coordY<0)
        return true;
    return false;
}
```

Checks whether the position is outside the boundaries of the grid. Prevents invalid positions from being used in gameplay.

Equality Comparison:

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Position other = (Position) obj;
    if (coordX != other.coordX)
        return false;
    if (coordY != other.coordY)
        return false;
    return true;
}
```

Compares two positions to determine if they have the same coordinates. Useful for detecting repeated positions, such as targeting the same cell.

3.3 Other:

a. **Computer:**

This class represents the CPU opponent in the game. It manages where the CPU will shoot.

- **Variables:**

```
5
6   public class Computer {
7       private LinkedList<Position> listHits;
8       private Random r;
9       private int hitted;
10      private LinkedList<String> possibilitize;
11      private Position lastShot;
12      private String direction;
13      private Map plMap;
14      private Position primohit;
```

  - **listHits:** A LinkedList to store all possible shooting positions.
  - **r:** A Random object for generating random numbers.
  - **hitted:** An integer to track the number of consecutive hits on a ship.
  - **possibilitize:** A LinkedList to store potential directions to shoot after a hit.
  - **lastShot:** A Position object to store the last shot fired by the computer.
  - **direction:** A String to store the current shooting direction (e.g., "N" for North).
  - **plMap:** A reference to the player's Map object.
  - **primohit:** A Position object to store the position of the first hit on a ship.
- **The Constructor:**

```
16
17      public Computer(Map mapAdversary) {
18          listHits = new LinkedList<Position>();
19          this.plMap = mapAdversary;
20          for (int i = 0; i < Map.DIM_map; i++) {
21              for (int j = 0; j < Map.DIM_map; j++) {
22                  Position p = new Position(i, j);
23                  listHits.add(p);
24              }
25          }
26          r = new Random();
27          hitted = 0;
28      }
```

Initializes the **listHits** with all possible positions on the map.

Stores a reference to the player's **mapAdversary** in the **plMap** variable.

Creates a new **Random** object.

Sets **hitted** to 0 initially.

- **myTurn:**

```
30      public Report myTurn() {
31
32          Report rep = new Report();
33          if (hitted == 0) {
34              boolean hit = shootRandom();
35              rep.setP(lastShot);
36              rep.sethits(hit);
37              Direction sunk;
38              if (hit) {
39                  hitted++;
40                  sunk = plMap.sunk(lastShot);
41                  if (sunk != null) {
42                      rep.setsunk(sunk:true);
43                      removeContours(sunk);
44                      hitted = 0;
45                      direction = null;
46                  } else {
47                      primohit = lastShot;
48                      possibilitize = new LinkedList<String>();
49                      inizializzaLista();
50                  }
51              }
52              return rep;
```

```
53          }
54          if (hitted == 1) {
55              boolean hit = sparaMirato1();
56              Direction sunk;
57              rep.setP(lastShot);
58              rep.sethits(hit);
59              rep.setsunk(sunk:false);
60              if (hit) {
61                  hitted++;
62                  possibilitize = null;
63                  sunk = plMap.sunk(lastShot);
64                  if (sunk != null) {
65                      rep.setsunk(sunk:true);
66                      removeContours(sunk);
67                      hitted = 0;
68                      direction = null;
69                  }
70              }
71              return rep;
```

```
73          if (hitted >= 2) {
74              boolean hit = sparaMirato2();
75              Direction sunk;
76              rep.setP(lastShot);
77              rep.sethits(hit);
78              rep.setsunk(sunk:false);
79              if (hit) {
80                  hitted++;
81                  sunk = plMap.sunk(lastShot);
82                  if (sunk != null) {
83                      rep.setsunk(sunk:true);
84                      removeContours(sunk);
85                      hitted = 0;
86                      direction = null;
87                  }
88              } else {
89                  invertidirection();
90              }
91              return rep;
92          }
93          return null;
94      }
```

This is the core method that determines the computer's next move.

It checks the value of hitted:

hitted == 0 (No previous hits):

Calls shootRandom() to select a random position.

Records the shot in lastShot and checks if it hits using plMap.hit1().

If hit:

Increments hitted.

Checks if the ship is sunk using plMap.sunk(). If sunk, removes surrounding positions from listHits using removeContours() and resets hitted and direction.

If not sunk, initializes the possibilitize list with potential directions using inizializzaLista().

hitted == 1 (One previous hit):

Calls sparaMirato1() to shoot in one of the four possible directions around the first hit.

Records the shot, checks for a hit, and handles sinking as in the hitted == 0 case.

hitted >= 2 (Two or more previous hits):

Calls sparaMirato2() to continue shooting in the current direction.

If the current direction is blocked, it calls invertidirection() to reverse the direction.

Records the shot, checks for a hit, and handles sinking as in the previous cases.

- **shootRandom**

```
private boolean shootRandom() {
    int tiro = r.nextInt(listHits.size());
    Position p = listHits.remove(tiro);
    lastShot = p;
    boolean hit = plMap.hit1(p);
    return hit;
}
```

Selects a random position from listHits, removes it, and fires a shot at that position.

- **sparaMirato1**

```
private boolean sparaMirato1() {
    boolean errore = true;
    Position p = null;
    do {
        int tiro = r.nextInt(possibilitize.size());
        String dove = possibilitize.remove(tiro);
        p = new Position(primohit);
        p.sposta(dove.charAt(index:0));
        direction = dove;
        if (!plMap.acqua(p)) {
            listHits.remove(p);
            errore = false;
        }
    } while (errore);

    lastShot = p;
    return plMap.hit1(p);
}
```

Selects a random direction from possibilitize, calculates the target position, and fires a shot.

- **sparaMirato2**

```
private boolean sparaMirato2() {
    boolean colpibile = false;
    Position p = new Position(lastShot);
    do {
        p.sposta(direction.charAt(index:0));

        if (p.fuorimap() || plMap.acqua(p)) {
            invertidirection();
        } else {
            if (!plMap.hitted(p)) {
                colpibile = true;
            }

        }
    } while (!colpibile);
    listHits.remove(p);
    lastShot = p;
    return plMap.hit1(p);
}
```

Continues shooting in the current direction until it hits or encounters a block.

- **removeContours**

Removes positions around a sunk ship from listHits to improve efficiency.

- **inizializzaLista**

```
private void inizializzaLista() {
    if (lastShot.getCoordX() != 0) {
        possibilitize.add(e:"N");
    }
    if (lastShot.getCoordX() != Map.DIM_map - 1) {
        possibilitize.add(e:"S");
    }
    if (lastShot.getCoordY() != 0) {
        possibilitize.add(e:"O");
    }
    if (lastShot.getCoordY() != Map.DIM_map - 1) {
        possibilitize.add(e:"E");
    }
}
```

Initializes the possibilitize list with potential directions around the first hit.

- **invertidirection**

```
private void invertidirection() {
    if (direction.equals(anObject:"N")) {
        direction = "S";
    } else if (direction.equals(anObject:"S")) {
        direction = "N";
    } else if (direction.equals(anObject:"E")) {
        direction = "O";
    } else if (direction.equals(anObject:"O")) {
        direction = "E";
    }
}
```

Reverses the current shooting direction.

b. Map:

This class represents Battleship's game board or map. It handles ship placement, tracks hits and misses, and provides methods for determining the game's state.

Constructor:

```
public Map() {
    listaNavi = new LinkedList<Direction>();
    map = new char[DIM_map][DIM_map];
    for (int i = 0; i < DIM_map; i++)
        for (int j = 0; j < DIM_map; j++)
            map[i][j] = NULLA;
}
```

Initializes the listaNavi LinkedList.

Creates a DIM_map x DIM_map 2D character array (map) and initializes all cells to NULL

Methods:

- **`riempimapRandom()`:**

```
public void riempimapRandom() {
    clear();
    Random r = new Random();
    insertNaveRandom(r, dimensione:4);
    insertNaveRandom(r, dimensione:3);
    insertNaveRandom(r, dimensione:3);
    insertNaveRandom(r, dimensione:2);
    insertNaveRandom(r, dimensione:2);
    insertNaveRandom(r, dimensione:2);
    insertNaveRandom(r, dimensione:1);
    insertNaveRandom(r, dimensione:1);
    insertNaveRandom(r, dimensione:1);
    insertNaveRandom(r, dimensione:1);
}
```

  - Clears the map.
  - Calls `insertNaveRandom()` to randomly place ships of different sizes on the map.
- **`clear()`:**
  - Resets the map by setting all cells to `NULL`.
- **`inserisciNave(x, y, dim, dir)`:**

- Places a ship of given dimension (`dim`) at the specified coordinates (`x`, `y`) and direction (`dir`).
- Checks for valid placement using `verificaOrizzontale()` or `verificaVerticale()`.
- Adds the ship's position and dimensions to the `listaNavi` list.
- Updates the `map` array to reflect the ship's placement.

- **`insertNaveRandom(Random random, int dimensione)`:**
  - Randomly generates coordinates and direction for a ship.
  - Checks for valid placement using `verificaOrizzontale()` or `verificaVerticale()`.
  - Places the ship on the map and adds it to the `listaNavi` list.
  - Returns an array containing the coordinates, dimension, and direction of the placed ship.
- **`verificaVerticale()` and `verificaOrizzontale()`:**
  - Check if a ship can be placed vertically or horizontally at the given coordinates without overlapping with other ships.
- **`hit1(Position p)`:**

```
public boolean hit1(Position p) {
    int riga = p.getCoordX();
    int colonna = p.getCoordY();
    if (map[riga][colonna] == NAVE) {
        map[riga][colonna] = HIT;
        return true;
    }
    map[riga][colonna] = ACQUA;
    return false;
}
```

  - Checks if a shot at the given `Position` hits a ship.
  - If hit, marks the position with `HIT` on the `map` and returns `true`.
  - If miss, marks the position with `ACQUA` on the `map` and returns `false`.
- **`sunk(Position p)`:**
  - Checks if the ship at the given `Position` has been completely sunk.
  - If sunk, removes the ship from the `listaNavi` list and returns the ship's `Direction` object.
  - If not sunk, returns `null`.
- **`setAcqua(Position p)`:**
  - Marks the given `Position` as `ACQUA` (missed shot) on the `map`.
- **`acqua(Position p)`:**
  - Checks if the given `Position` is marked as `ACQUA` (missed shot).

- **`hitted(Position p):`**

```
public boolean hitted(Position p) {
    return map[p.getCoordX()][p.getCoordY()] == HIT;
}
```

    - Checks if the given `Position` is marked as `HIT` (ship hit).
- **`toString():`**
    - Returns a string representation of the `map` for display.
- **`setAdvShips(LinkedList<int[]> advShips):`**
    - Sets the positions and dimensions of the opponent's ships on the map.
    - Used for debugging and testing purposes.

.

CHAP 4:
FINAL APP GAME

Source code (link github): https://github.com/TheeVinhh/oop-iu

```
https://github.com/TheeVinhh/oop-iu
```

Demo video:

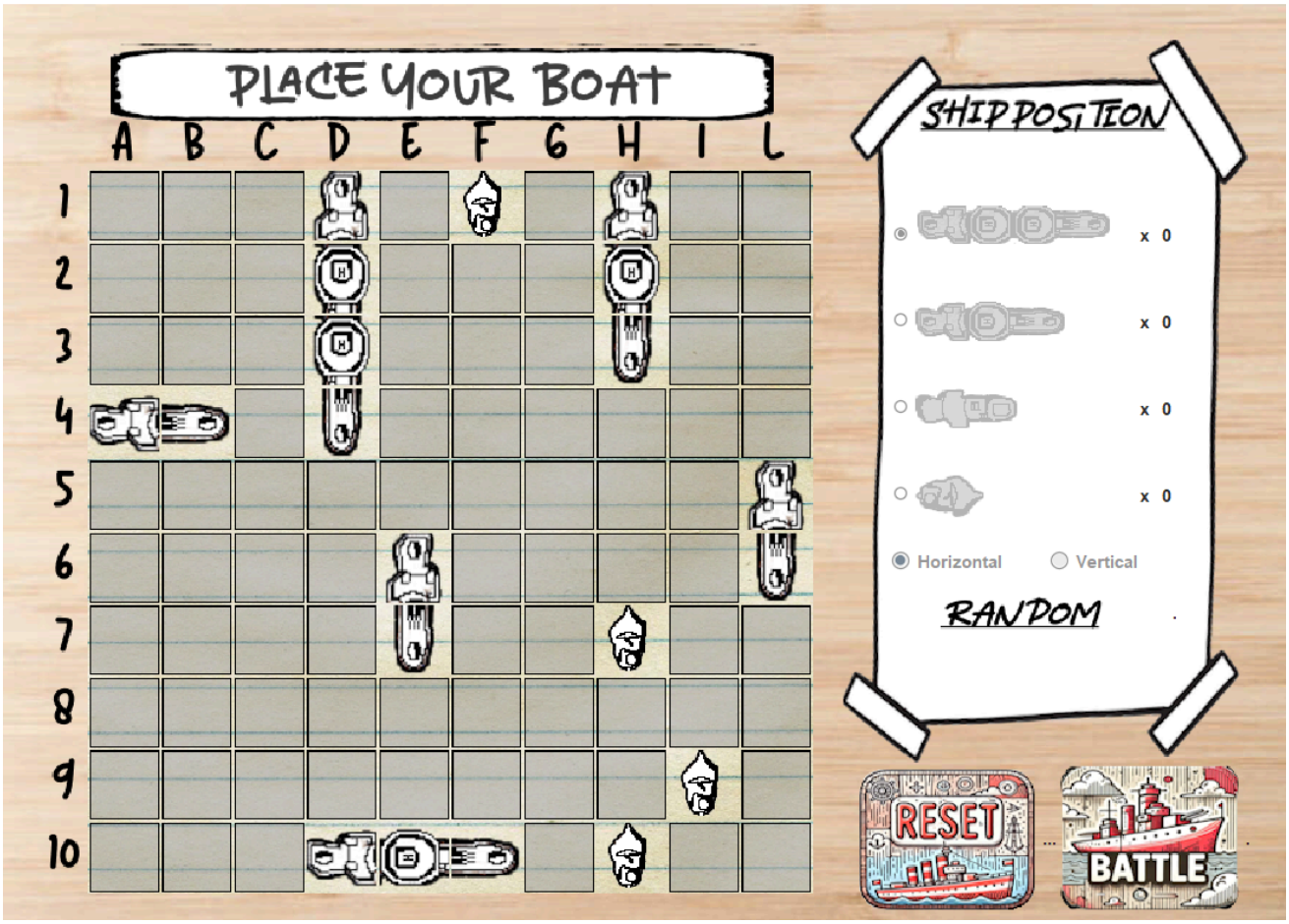
```
https://drive.google.com/drive/folders/1pR7dEPPd0IVGundtv7reT39ZH0o24veD?usp=drive_link
```

Instruction

1. Begin the game:

Click on the icon to open the game. Loading screen will appear.

2. Main Menu:



- **Central Grid (A1 to J10)**

Purpose: Allows players to position their ships manually.

- Players can click on specific grid cells to place ships, ensuring strategic placement. The grid highlights the selected area based on the ship's size and orientation.

- **Ship Selection Panel (Right Side)**

Ship Types:

x1: One ship with a size of 4 cells.

x2: Two ships with a size of 3 cells.

x3: Three ships with a size of 2 cells.

x4: Four ships with a size of 1 cell.

Purpose: Allows players to choose which ship to place next.

- Players select the type of ship they want to place, and the game ensures they follow the allowed quantity for each type.

- **Orientation Selector (Horizontal/Vertical)**

Purpose: Lets players choose the orientation of the ship they are placing.

- **Horizontal:** Places the ship horizontally across the grid.
- **Vertical:** Places the ship vertically down the grid.

- **Random Placement Button**

Purpose: Automatically places all ships randomly on the grid.

- With one click, the game populates the grid with ships in random positions and orientations, skipping manual placement.

- **"RESET" Button**

Purpose: Clears the grid and resets all ship placements.

- Returns the grid to its initial state, allowing players to restart the setup phase.

- **"BATTLE" Button**

Purpose: Finalizes the ship placement and starts the gameplay phase.

- Once clicked, the game transitions from the setup phase to the main battle phase.

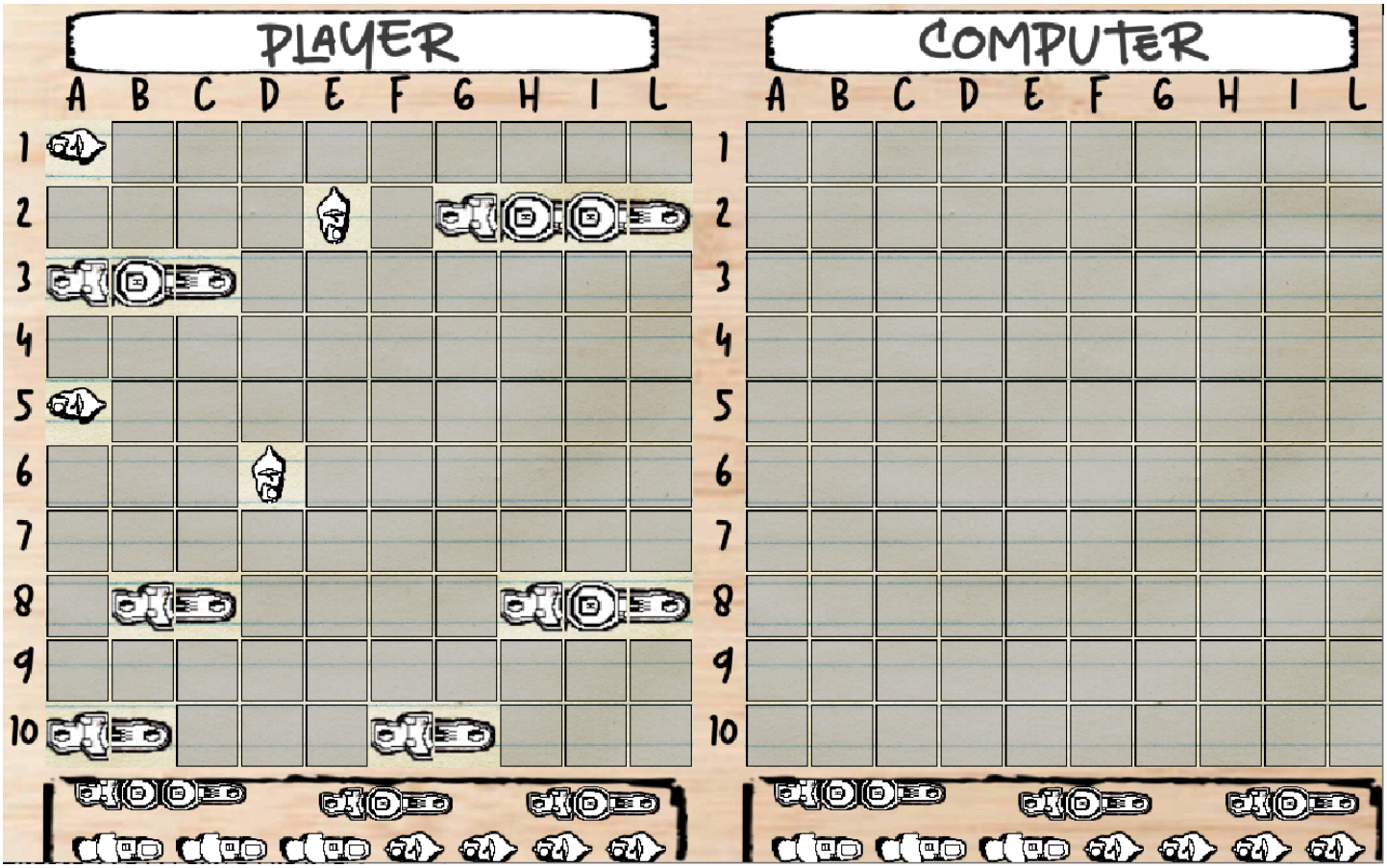
- **Header ("Place Your Boat")**

Purpose: Provides instructions to the player.

- Displays the current phase of the game (ship placement in this case)

-

3. Gameplay Panel



A. Player Grid (Left):

Labeled as "PLAYER" at the top.

- A 10x10 grid with rows numbered 1 to 10 and columns labeled A to J.
- Several ships of different sizes are placed on the grid, aligned either horizontally or vertically.
- Each ship occupies a specific number of squares based on its size.

B. Computer Grid (Right):

Labeled as "COMPUTER" at the top.

- Another 10x10 grid, also numbered and labeled identically to the player grid.
- The grid appears empty, as the computer's ship positions are hidden.

Ship Icons (Bottom):

- Displays the types of ships available for both the player and the computer.
- Each icon corresponds to a specific ship size, indicating how many squares it occupies.

4. What to Do:

Objective:

- The goal is to sink all of your opponent's ships before they sink yours.
- Gameplay Instructions:
- Players take turns guessing the location of their opponent's ships by calling out grid coordinates (e.g., "C5").

Mark hits or misses on the computer grid:

- A hit indicates part of a ship is at that location.
- A miss indicates no ship is present at that location.
- Use the player's grid to track your ships and hits/misses from the computer's guesses.

Winning the Game:

- A ship is considered "sunk" when all the squares it occupies have been hit.
- The first player to sink all of their opponent's ships wins the game.

CHAP 5: EXPERIENCE

This project has taught us a crucial lesson: just like a web server without content, a game without engaging content is ultimately meaningless. The quality of a game cannot be measured solely by the technical prowess of its software.

The project provided invaluable hands-on experience, allowing us to apply classroom learnings and grapple with numerous bugs. It also motivated us to actively seek knowledge beyond the curriculum, keeping our skills sharp and our understanding of the evolving IT landscape current.

We learned that self-directed learning is paramount in the field of Computer Science. The IT world moves at a rapid pace, demanding continuous self-improvement.

Moving forward, our team aims to further improve this game as our first published project and striving to consistently deliver high-quality and engaging experiences to users.

- END -