University of Exeter

UNIVERSITY OF EXETER · LUCEM SEQUIMUR

PROGRAMMING WITH PYTHON - COMM109J

BATTLESHIP BOARD GAME

Declaration

I want to acknowledge using ChatGPT as a tool to complete this project. Further, I acknowledge the use of GenAI tools in this assessment for the following:

[✓] For developing ideas.

[✓] To assist with research or gathering information.

[✓] To help me understand key theories and concepts.

[✓] To suggest a plan or structure for my work.

[✓] To give me feedback on a draft.

[✓] To generate images, figures or diagrams.

[✓] To proofread and correct grammar or spelling errors.

I declare that I have referenced all use of GenAI outputs within my assessment in line with the University referencing guidelines.

Battleship Game Technical Report

Introduction

Battleship Board Game is a classic two-player where each both the players place their ships in their board wisely and they will go on guessing the coordinated of opponent's players fleet to sink all of their ships. Milton Bradley first developed this game thought first it happened to be a pen and paper later it was designed to a board format.

I have developed a Python program for that game using the same rules here. It has a Graphic User Interface (GUI) using Tkinter, which has been integrated from the Command-Line Interface (CLI). The user can play in either of the modes he likes. This game uses Object-Oriented Programming (OOP) concepts to be modular, scalable, and easy to maintain, and concepts like Abstraction, Encapsulation, Inheritance, and Polymorphism are applied.

This game is more interesting because the computer player becomes a formidable opponent for the user to play against. The game is implemented using the Mode-View-Controller (MVC) pattern, which keeps the main game logic from the user interface. This makes the code extendable in the future.

Game Overview

Name of the Game: Battleship

YouTube Tutorial: https://youtu.be/RY4nAyRgkLo?si=tT6VBWy47nnAkQ1m

Gameplay Tutorial: https://tinyurl.com/yzpr86cm

Objective: The game aims to sink your opponent's ships before they sink all of yours.

Setup: Each player has an 8x8 grid and a fleet of ships. Players secretly place their ships on their grid either horizontally or vertically. Ships cannot overlap or be placed diagonally.

Gameplay: Players take turns calling out coordinates (e.g., B6) to hit their opponent's ships. The opponent responds with 'hit' if a ship occupies the coordinates or 'miss' if there is no ship. The player marks their board with '-' for a miss and 'X' for a hit. When all of the squares of a ship have been hit, the ship is sunk, and the player must announce which ship was sunk.

Winning: The game continues until one player has sunk all their opponent's ships. That player is declared the winner.

Termination: The game terminates when a player or computer successfully sinks all of their opponent's ships. After this point, the game cannot continue, and the winner is announced.

Milestones

Stage	Task	Completed?
1	Use object-oriented programming to structure the project using multiple classes.	Yes, refer to the Architecture Diagram
1	Implement a simple game loop	Yes
1	Handle user input for gameplay. (Command-Line Interface)	Yes, In the main Game File
1	The game progresses according to its basic rules	Yes
1	The game can detect and declare a winner	Yes
1	A requirements.txt file that enables code transferability	Yes, Added a requirement file
2	Implement unit tests covering core methods	Yes, Using Unit test
2	Implement appropriate code documentation	Yes, Used Docstrings
2	Implement error handling to prevent crashes	Yes, using Try-Catch
3	Read game settings from a configuration file	Yes, the JSON file has been added
3	Implement a graphical user interface (GUI)	Yes, using Tkinter
3	Additional features or elements would enhance the game design and/or presentability.	Yes, I Added colors for hit/miss, and in the GUI

Design Patterns

Class Name	Role in Object-Oriented Programming	Description
BoardDisplay	UI Helper Class	Manages the display of the game board.
ShipManager	Ship Handler Class	Handles ship creation, placement, and tracking
BoardValidator	Validation Class	Ensures ship placements and moves are valid.
BasePlayer	Abstract Player Class	Defines a standard interface for all players.
HumanPlayer	Derived Player Class	It inherits from Base Player and handles human-player interactions.
ComputerPlayer	Derived Player Class	Inherits from Base Player, implements AI logic for the computer
GameSetup	Setup Class	Initializes the game by setting up players, ships, and the board.

GameLoop	Main Game Controller	Manages the main game loop, processing turns, and maintains game flow.
CLIGamePlay	Game Logic Manager	In-game actions like attacks, turn switching, and conditions.
WindowManager	GUI Window Controller	Manages the main game window and GUI events.
GameDisplay	UI Rendering Class	Handles rendering of the game board and visual elements.
BattleshipGUI	Main GUI Class	Integrates game logic with the graphical user interface.

Object-Oriented Design Principles

1. **Abstraction**

- The concept of Abstraction is used in BasePlayer class, which controls the logic of players in this game.
- The classes HumanPlayer and ComputerPlayer are implemented from the BasePlayer class, focusing more on its class logic without revealing more information outside the class.

2. **Encapsulation**

- The ComputerPlayer class has hidden classes that encapsulate the AI logic and are isolated from other classes.
- The class ShipManger class shows only ship placing and checks the ships placement and it does not show how they are placed and how the tracking system works.

3. **Inheritance**

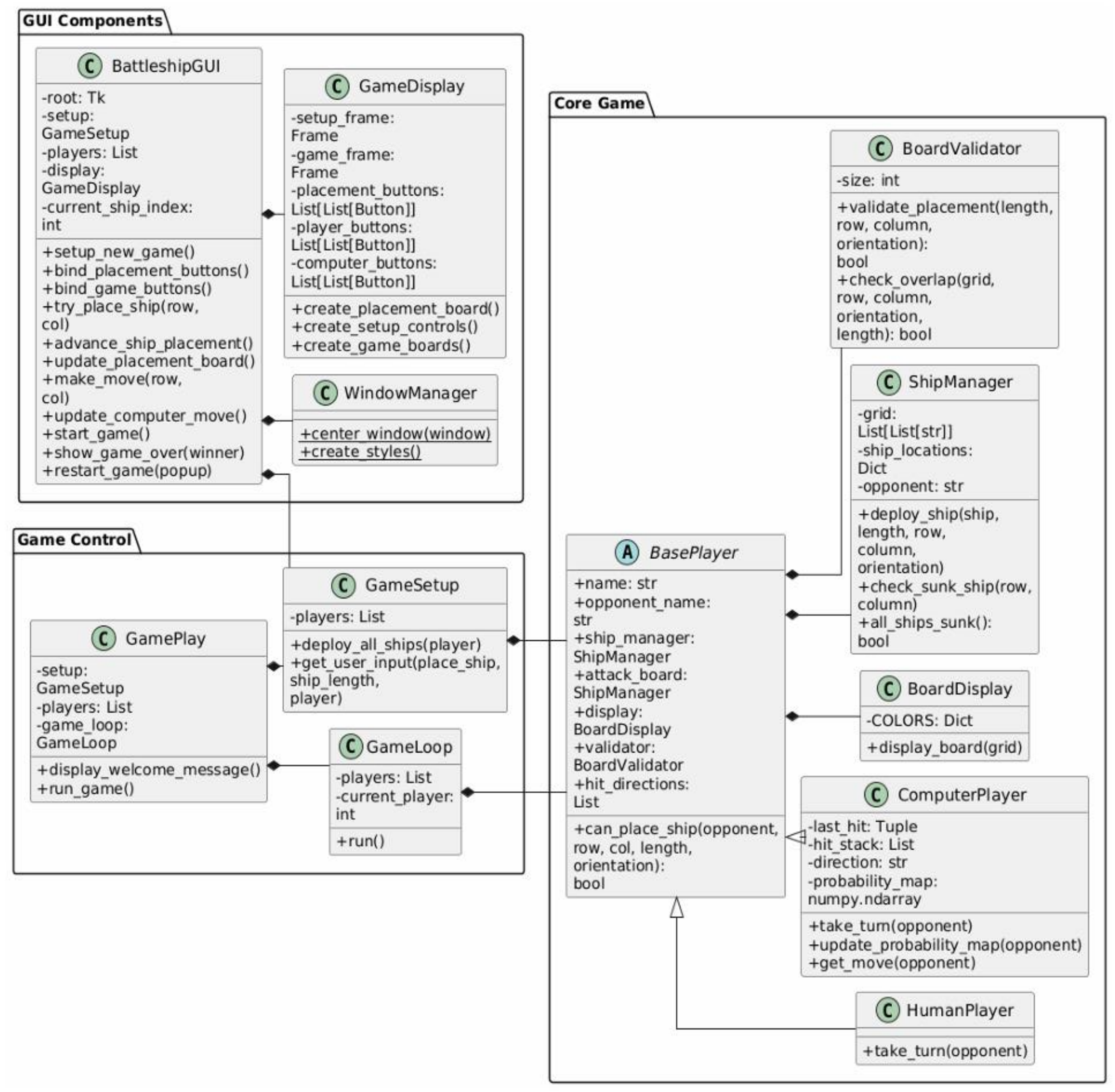
- Inheritance is used to reduce redundancy and promote code reuse.
- HumanPlayer and ComputerPlayer inherit from BasePlayer to avoid duplication of shared functionality like attack handling.

4. **Polymorphism**

- Polymorphism was implemented through the take_turn() function declared in BasePlayer and overridden in HumanPlayer and ComputerPlayer.
- The run-time game loop can support player turns regardless of whether the player is human or computer.

Architecture

The Battleship Game is designed based on a modular and scalable architecture with roots in object-oriented programming (OOP) principles. The system is divided into three main components that follow encapsulation, abstraction, and the Model-View-Controller (MVC) design pattern to separate concerns and ensure maintainability.



1. Game Control Layer (Controller - MVC)

- **Classes:** GameLoop, GameSetup, GamePlay
    - Manages game initialization, player setup, and game looping.
    - Controls the game flow turn alternating, attacking, and rule application.

- **OOP Concepts:**
    - The encapsulation technique is used in the GameLoop class to maintain game flow.
    - Single Responsibility Principle (SRP): Each Class has their function, i.e. GameRunSetup does the initial setup, GameLoop takes care of the gameplay and in-game tasks are handled by the GamePlay class.

2. Core Game Layer (Model - MVC)

- **Classes:** BasePlayer, HumanPlayer, ComputerPlayer, ShipManager, BoardValidator, BoardDisplay
    - These above classes manage the behaviour, ship placing checking and board
    - display validation.
    - The class ComputerPlayer inherits from BasePlayer has a probability-based targeting method which plays like a user.
- **OOP Concepts:**
    - Inheritance is used in the HumanPlayer and ComputerPlayer classes, which extend BasePlayer. Both classes use polymorphic methods take_turn() and receive_attack().
    - The BasePlayer class is the abstract class for the HumanPlayer class and ComputerPlayer class.
    - Probability-based targeting method is encapsulated in the ComputerPlayer class to present a modular design.

3. GUI Layer (View - MVC)

- **Classes:** BattleshipGUI, GameDisplay, WindowManager
    - These Classes maintain the overall functions of the GUI, i.e. input from the user, visual output, and response to the game events.
    - Visualizes colours for ship placing(Red), hit(Red) and misses(Blue) in the GUI board.
- **OOP Concepts:**
    - GUI rendering and event processing are two events encapsulated in the main game logic.
    - Here the methods are separated into independent and reusable components (Modularity) so that the GUI interacts with the game control and core layers does not know the exact logic behind it (Abstraction).

Methodology

This game is designed using an Instance-based Object-Oriented Approach, which mainly focuses on Modularity, Reusability and maintainability while developing this game.

1. Iterative Development Process

- The development cycle followed testable methods, like first building the base logic of how the game should work and proceeding according to the rules.
- The first task was to develop a working **Command-Line Interface (CLI)**, which adheres to the core game logic and manual typing of coordinates to place and attack the ship.
- When the base logic was set, the next task was to make the opponent play like a user and developing a **Graphical User Interface (GUI)** to make the game more interesting.

2. Object-Oriented Programming as Foundation

- **Encapsulation** was used in each class to handle its own data and methods to prevent tight coupling.
- **Inheritance and Polymorphism** allowed code reusability and made it simple to extend functionality (e.g., ComputerPlayer as an AI extension of BasePlayer).
- **Abstraction** simplified complex behaviours, e.g., game state management and AI decision-making, to simple interfaces.

3. MVC Design Pattern

- The project adhered to **the Model-View-Controller (MVC)** pattern to isolate the core game logic (Model), the graphical interface (View), and the game flow control (Controller).
- This separation of concerns improved maintainability and allowed the GUI and CLI versions to share the common game logic, indicating high modularity.

4. Justification for OOP Methodology

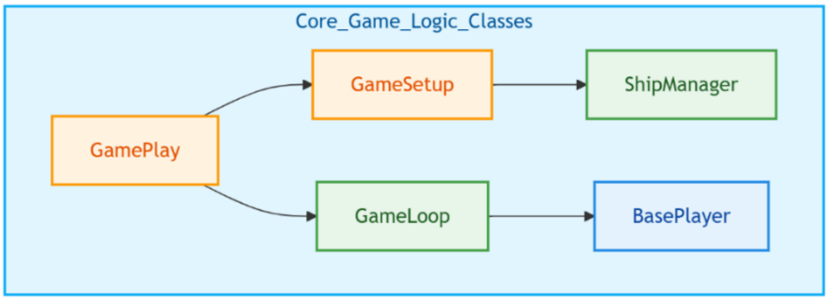
- **Scalability**: sing the concept of Object-Oriented Programming, let us to add features like changing the entire computer playing method, adding GUI without stepping down on the fundamental game logic.
- **Reusability**: Some of the standard methods were abstracted into base class to avoid duplication of code (Redundancy).
- **Maintainability**: While developing the classes their responsibilities was organized in a way to make it simple and can be extended for future.

Implementation

The game is separated in four main modules, i.e. The **Core Game Logic**, The **Board and Ship Management Classes**, The **Player Classes**, and The **Graphical User Interface (GUI)**.

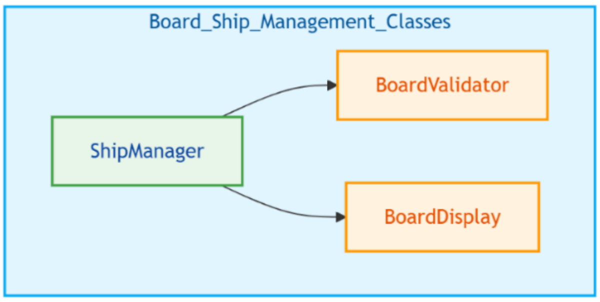
1. Core Game Logic Classes

The Core Game Logic is the base for the game's operation. It closely works with the initial game setup and looping of the game, following the players and ship manager classes. These classes combine to make the game flow uninterrupted.



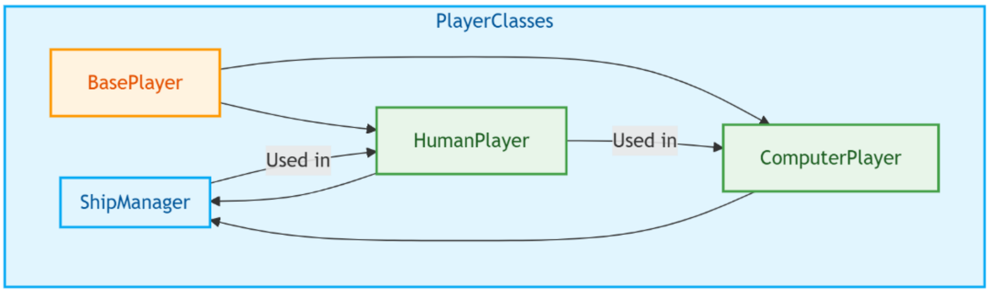
2. Board and Ship Management Classes

The Ship Manager class defines the board management while validating the user's placement of the ships and displays it both in CLI and GUI.



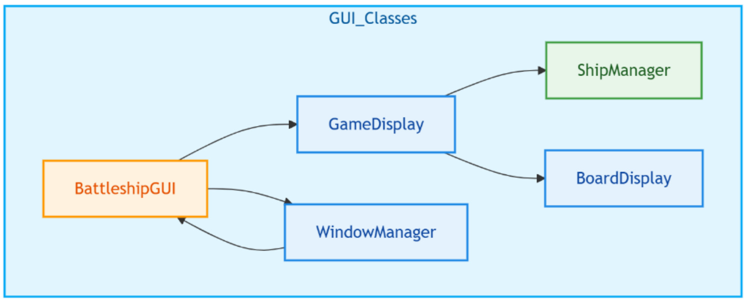
3. Player Classes

In this the Base Player is the main class which initiates the human and computer player. These two classes have their own methods of making the game move forward and do not collide with each other classes.



4. Graphical User Interface (GUI) Classes

The above classes are combined to create a GUI. Only the process of creating a GUI takes place, and the main logic is from the CLI.



Unit Testing

Unit testing uses Python's pre-defined module to ensure the functionality and accuracy of all various components in the Battleship game. The test cases are implemented for the BoardDisplay, ShipManager, BoardValidator, HumanPlayer, ComputerPlayer, and GameSetup classes.

1. **BoardDisplay Class (Black Box Testing)**
   - Colour code test initialization checks whether required keys like X, -, and RESET are in the COLORS dictionary.
2. **ShipManager Class (White Box & Black Box Testing)**
   - Initializes tests to verify proper initialization of the board grid and empty ship locations. (White Box Testing)
   - Verifies ship deployment by testing whether the grid is updated correctly and the ship is deployed as anticipated. (Black Box Testing)
   - Verifies ship sinking detection by striking all parts of a ship and confirming its removal from ship locations. (Black Box Testing)
   - Verifies overall sinking detection by deploying a ship and checking its status after sinking. (Black Box Testing)
3. **BoardValidator Class (White Box Testing)**
   - Checks ship placement within grid boundaries. (White Box Testing)
   - Checks for overlapping ships upon deployment by testing grid occupancy. (White Box Testing)
4. **HumanPlayer Class (White Box Testing)**
   - Test initialization to ensure components like ShipManager, BoardDisplay, and BoardValidator are correctly created and linked. (White Box Testing)

5. **ComputerPlayer Class (White Box & Gray Box Testing)**
    - Test initialization to ensure attributes like name, opponent_name, last_hit, hit_stack, and direction are correctly set. (White Box Testing)
    - Validates the probability map initialization to target ships. (Gray Box Testing)
6. **GameSetup Class (Black Box & Integration Testing)**
    - Verifies the initialization process to confirm that HumanPlayer and ComputerPlayer are correctly instantiated while setting up the game. (Black Box Testing & Integration Testing).
7. **WindowManager Class (GUI Testing)**
    - Window centring Test (Functional Test): Ensures the game window is correctly positioned
    - Style Creation Test (GUI Testing): Validates that UI styles are created without errors
8. **GameDisplay Class (GUI Testing)**
    - Initialization Test (GUI Testing): Verifies the correct setup of game frames and buttons.
    - Button Grid Size Test (GUI Testing): Ensures that UI button grids match the board size
9. **BattleshipGUI Class (GUI Testing & Integration Testing)**
    - GUI Initialization Test (Integration Testing): Ensures the entire GUI framework is correctly set up
    - GUI Mode Setting Test (GUI Testing): Confirms that both players have GUI mode enabled Testing)
10. **Additional Edge Case & Mock Testing**
    - Invalid Ship Placement: Tests ship deployment at invalid positions
    - Duplicate Moves: Ensures repeated attacks at the same position are handled correctly
    - Mocking GUI Inputs: Uses mocks for GUI-based tests to simulate user interactions

Challenges

While developing the game, many different obstacles were encountered. These challenges were resolved using the Advanced Object-Object Programming methods, which involve creating a modular, scalable and maintainable solution. The challenges are listed below.

1. Advanced Computer Player Logic Development with Object-Oriented Design

- **Challenge:** The opponent of the user, i.e. the computer, has to play like a user instead of making random guesses through the game. Random guesses made by the computer does not make the game exciting for the player.
- **Solution:**
  - The ComputerPlayer class was designed as a subclass of BasePlayer. Instead of random attacks, the AI remembers previous hits and responds with adaptive action on a probability-based targeting system.
  - The AI uses heatmap-based decision-making, assigning higher probabilities to the areas where ships are more likely to be. If a ship is attacked, the AI initially targets nearby spots before moving elsewhere.
- **Alternatives**: A Simplistic heuristic approach, i.e. making the computer to implement the checkers logic into the game was considered.
- **OOP Techniques:**
  - The ComputerPlayer class is extended from the BasePlayer
  - class (Inheritance) where the probability-based targeting technique is applied.
  - The ComputerPlayer class has hidden classes that encapsulate the AI logic and are kept away from other classes.

2. GUI and Game Logic Synchronization

- **Challenge:** The GUI had to be refreshed now and then so that it would not affect the game. It would become a big task if the game logic and GUI were mixed. It is hard to debug and modify.
- **Solution:** The Model-View-Controller (MVC) technique was used:
  - The model has classes GameSetup, ShipManager, and Stores game state.
  - The view has classes GameDisplay and BattleshipGUI. It Handles user input.
  - The controller has classes GameLoop, which Manages the game flow.

  Using this make the game more defined and enables to update the GUI without any change in the core logic.
- **Alternatives: Considered** to directly call the function to refresh the GUI and update it accordingly. But it would become more complex.
- **OOP Techniques:**
  - The core logic of the game is encapsulated in the GUI.
  - Every method is separated into independent and reusable components (Modularity).

3. Ensuring Valid Moves and Ship Placement

- **Challenge:** When the Player places the ships, the program should validate that they are not placed outside the grid and do not overlap any other ships on board, which makes the game worse and more challenging to maintain.

- **Solution:** This is done by the class BoardValidator to maintain all these logics. This class ensures:
    - That the ships placed are inside the boundaries of the board created.
    - It also checks that ship placed does not overlap with other ship.
- **Alternatives:** Manual verification of every condition within the ShipManager class. This, however would make ShipManager bulky and more difficult to maintain.
- **OOP Techniques**
    - The boardValidator method only performs validation and is encapsulated.
    - ShipManager method only manages the ships and does not perform validation. It is a single responsibility principle.

4. Modular and Maintainable Codebase

- **Challenge:** The game developed should be easily extended so that we can add additional features.
- **Solution:** Used inheritance to create a flexible player system:
    - BasePlayer (Abstract class) → Defines general player functionality.
    - HumanPlayer & ComputerPlayer (Derived classes) → Extend BasePlayer with specific behavior.
    - Used composition instead of deep inheritance wherever required.
    - The design allows new features to be added easily without rewriting the entire codebase.
- **OOP Technique:**
    - Following this, avoid redundant code by reusing the objects instead of modifying new classes.

5. Error Handling and User Input Validation

- **Challenge:** Preventing the game from crashing with incorrect user inputs (e.g., entering incorrect grid positions).
- **Solution:** Used exception handling (try-except) in HumanPlayer to catch and handle invalid input. The BoardValidator class game rules for placing ships. While placing the ships in board if ship is placed again in same position or if it is going to overlap another ship it will prompt to place the ship again to avoid crash.
- **OOP Techniques:**
    - Exception Handling: Prevents unexpected crashes.
    - To avoid crashing in between while playing Exception are used

Advanced Features

1. Probability-Based Targeting

- When the computer has a chance to hit the ships, it randomly hits possible positions until it gets a hit.
- After it gets the first hit, it switches to Probability-Based Targeting to hit the ships in the following positions.
- It then creates a heatmap to hit the positions with a higher probability (0,1) of ships.
- When it gets a hit, it targets nearby positions instead of randomly attacking.

Working:

- It uses arrays and probability mapping to decide which positions are more likely to contain ships.
- Implemented target tracking to focus on sinking ships efficiently.
- Refines itself with time based on previous hits/misses.

Why is it an Advanced Feature?

- The computer learns from the user's action
- instead of attacking random spots.
- The ComputerPlayer Class is encapsulated, and polymorphism is used to implement the take_turn() method inside that class.
- This feature makes the game engaging for the player, and they also create a strategy for playing it.

2. Graphical User Interface (GUI) with Tkinter

- Provides an interactive GUI instead of just a text-based interface.
- The player clicks buttons instead of typing commands.
- Ship placement, attacks, and hit/miss are much more manageable without typing the coordinated every time.

Working:

- Uses the Tkinter library to create a click-based user interface.
- The GUI is connected to the GameSetup and GameLoop classes to match the exact game logic with the key elements in the CLI.

Why is it an Advanced Feature?

- When playing the game in the CLI version, the user must input the coordinates to attack the ship. This is where the GUI comes in. The user can access the board and click on the grids to attack.
- While developing the GUI logic, the same MVC pattern is applied to improve the code's maintainability and allow us to add new features later in the game.

3. Color-coded Hit/Miss System

- Red = Ship placements and Successful hit
- Green = Missed shots
- Integrated these colours both on the CLI and GUI version

Working:

- Uses ANSI escape codes in CLI and Tkinter styles in GUI to dynamically change button colours.
- Updates real-time feedback after each player moves.

Why is it an Advanced Feature?

- It enhances game readability of the game since it becomes easy to distinguish between hits/misses.
- Enhanced users' accessibility through colour-based classification.

4. Configuration File (config.json)

- Stores game settings externally so players do not have to modify the code.
- Players can modify board size, ship types, and AI difficulty by modifying config.json.

Working:

- Reads settings from config.json using Python's JSON library.
- Loads ship sizes, grid size, AI behaviour settings, etc.

Why is it an Advanced Feature?

- Players can easily set the game rules using this file without changing the base code.
- The game settings
- are independent of the main game logic.

5. Unit Testing

- We use the base unit test module to test the methods in the class:
    - Ship place validation checks if the ships the user places are in the correct position.

    - Game logic involves declaring the winner and handling attacks by both players.
    - Working on Probability-based decision-making, which is used for computer

Working:

- The test cases check for core functional breaks in BoardValidator, ShipManager, GameLoop, and AI logic. If any issues are found, the tests automatically fail.

Why is it an Advanced Feature?

- This test validates code stability avoids bugs, and helps us perform automated testing of the game methods.

6. Dynamic Ship Placement & Validation

- These classes prevent the user from placing the ships on one another.
- The computer itself places the ships in an appropriate random position.

Working:

- Here, the BoardValidator class checks for ship placement before proceeding.
- The same goes for the computer if it is placed in a valid position that does not cause errors.

Why is it an Advanced Feature?

- It helps to re-validate possible future errors and ensure smooth gameplay.

7. Optimized Game Loop for Better Performance

- The game is designed in such a way so that is fast and more responsive by avoid unnecessary computations.

Working:

- It only executes probability map when its necessary and not on each turn.
- A lazy evaluation technique is used to reduce the performance overhead.

Why it is an Advanced Feature:

- It enhances the game's performance, especially when there are multiple simulations.

Conclusion

The Battleship Game project uses Object-Oriented Programming (OOP) principles to create a maintainable, scalable, and modular system through Abstraction, Encapsulation, Inheritance, and Polymorphism. Using the MVC pattern gives the game a clean architecture with some reusable features and a good interface that follows the game's logic.

Some features like Probability-based and heatmap implementation, a graphical user interface, and unit testing make the code more functional. This supports future improvements with minimal code changes.

Overall, this project highlights how OOP enables the development of robust and flexible programs in game design.

References

1. Tkinter Documentation. (n.d.). *Tkinter 8.5 reference: A GUI for Python*. Retrieved from https://docs.python.org/3/library/tkinter.html
2. Python Software Foundation. (n.d.). *unit test — Unit testing framework*. Retrieved from https://docs.python.org/3/library/unittest.html
3. NumPy Developers. (n.d.). *NumPy: The fundamental package for array computing with Python*. Retrieved from https://numpy.org/
4. Hasbro. (n.d.). *Battleship Game Rules & Instructions*. Retrieved from https://www.hasbro.com/common/instruct/Battleship.PDF
5. Python Software Foundation. (n.d.). *random — Generate pseudo-random numbers*. Retrieved from https://docs.python.org/3/library/random.html
6. Python Software Foundation. (n.d.). *Json — JSON encoder and decoder*. Retrieved from https://docs.python.org/3/library/json.html