Battleship Game Report

Summary

This is an interactive Python Battleship game with graphical user interface (GUI) and command-line interface (CLI). The game is played in accordance with standard Battleship rules, where players position their ships strategically and then take turns firing at the other player's board in an attempt to sink all of the other player's ships. Object-oriented concepts were used to achieve modularity, scalability, and maintainability.

The most important aspect of the project is the AI opponent that uses probability-based targeting for making decisions. The GUI enhances the game by providing an interactive session, making the game visually attractive and simple to use.

Its core features are input validation, error handling, and immediate feedback that provide smooth and efficient game flow. Its minimalistic codebase makes future additions easy, like difficulty levels or multiplayer. The project illustrates Python application in game development focusing on areas like GUI programming, AI, and event-driven programming.

Introduction

Battleship is a traditional strategy board game in which players deploy fleets and attack according to educated guesses of the location of the other player's ships. The goal of this project is to implement a Battleship game using Python, both command-line and graphical user interfaces.

The command-line edition uses libraries like numpy and json to model game logic, the board, and configuration handling. The GUI edition, developed using Tkinter, provides a visually intuitive, interactive experience. Object-oriented programming principles are used to encapsulate game logic, player interaction, and AI decision-making.

Why This Project? The reason behind this project is to illustrate practical uses of Python programming concepts like GUI programming, artificial intelligence, and object-oriented programming. The project also seeks to provide a fun and interactive user interface by presenting CLI and GUI versions of the game.

Objective

The main goals are to create an interactive Battleship game in Python both in text-based and graphical user interface (GUI) formats. The main goals are:

- Implement core gameplay mechanics: Ship placement, turn-based combat, and victory conditions.
- Develop an interactive interface: Provide both command-line and GUI versions.
- Create an AI opponent: Use probability-based decision-making to enhance gameplay difficulty.
- Ensure code modularity: Utilize object-oriented programming for scalability and maintainability.
- Provide input validation: Ensure smooth gameplay by preventing illegal moves and providing clear error messages.
- Enhance user experience: Include visual hints, animations, and feedback in the GUI.
- Improve game performance: Ensure smooth gameplay without lag or inefficiencies.
- Provide educational value: Demonstrate Python programming and game development principles.

By achieving these goals, the objective is to provide an enjoyable and interactive game and simultaneously demonstrate sound programming principles and software engineering principles.

Methodology

The implementation of the Battleship game was a systematic process, where there was a series of tools and techniques to reach an executable, interactive, and optimum implementation. In this chapter, the development process, tools, and techniques used in the project are explained.

The project was developed on an iterative construction model, wherein the features were progressively added in batches, tested, and refined. The development process had two general steps:

1. **Text-Based Version Implementation** - The initial task was to develop a full command-line based version of the game, including the fundamental mechanics like ship placement, attack calculation, and AI logic in a proper manner.
2. **Graphical User Interface (GUI) Development** - After completing the text-based version, the GUI was implemented with Tkinter to enhance user experience with a friendlier graphical interface.

Each phase followed an object-oriented design to ensure modularity and maintainability. The project was structured into distinct classes handling different game components, such as board management, player actions, AI logic, and display.

Implementation

The Battleship Game has been implemented in Python using an object-oriented programming approach, structured into multiple modules for better maintainability and clarity. The project is divided into three main components: the **Core Game Logic**, the **Board and Ship Management Classes**, the **Player Classes**, the **Graphical User Interface (GUI)**, and the **Unit Testing** suite. The implementation focuses on creating a user-friendly game with an AI opponent and options for both text-based and graphical interaction.

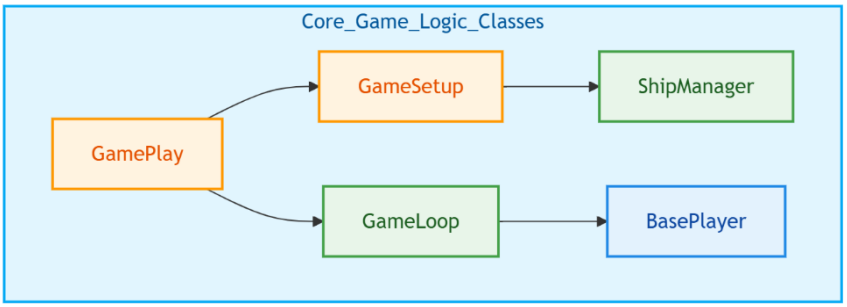
1. Core Game Logic Classes

The core game logic defines the fundamental behaviour of the game, including how players interact, how ships are managed, and how gameplay progresses. These classes work together to provide a smooth game flow.

GamePlay Class

- The GamePlay class is the main coordinator of the game. It creates the GameSetup and GameLoop instances and runs the complete game flow.

- The display_welcome_message method prints the game introduction and instructions to the user.
- The run_game method orchestrates the game setup and the main game loop.



GameSetup Class

- The GameSetup class handles the initialization and deployment of ships for both the player and the computer.
- The deploy_all_ships method manages the ship placement process for each player. It iterates through the available ship types and prompts the player or the computer to place the ships on the grid.
- For the player, it displays the game board and guides the user through the ship placement process, validating the input and ensuring no overlaps.
- For the computer, it randomly selects the ship orientation, row, and column, and deploys the ships on the grid, also validating the placement.

GameLoop Class

- The GameLoop class manages the main game loop, where players take turns attacking each other's ships.
- The run method executes the game loop until a winner is determined.
- In each turn, the current player's grid is displayed, and the player or the computer takes their turn by attacking a position on the opponent's grid.
- The game loop continues until one player's ships are all sunk, at which point the winner is declared.

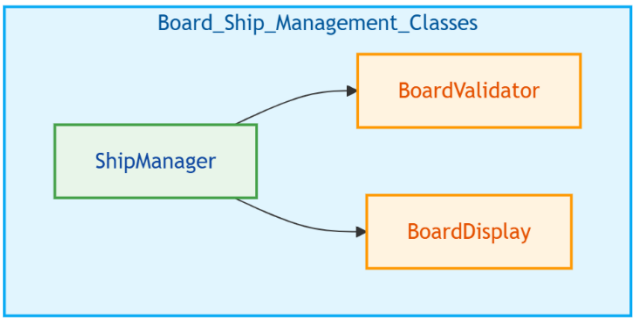
2. Board and Ship Management Classes

These classes manage the state of the game board, validate ship placements, and display the board in both textual and graphical formats.

ShipManager Class

- The ShipManager class manages the game board state and the placement of ships.

- The deploy_ship method places a ship on the board and records its location in the ship_locations dictionary.
- The check_sunk_ship method checks if a ship has been sunk and removes it from the tracking.
- The all_ships_sunk method checks if all of the player's ships have been sunk.



BoardValidator Class

- The BoardValidator class provides methods to validate ship placements and check for overlaps on the game board.
- The validate_placement method checks if a ship can be placed within the board boundaries.
- The check_overlap method checks if a ship placement overlaps with existing ships on the board.

BoardDisplay Class

- The BoardDisplay class is responsible for the visual representation of the game board, using colored output to differentiate between ships, hits, and misses.
- The display_board method takes a grid and formats it for display, using the defined color codes.

3. Player Classes

The player classes define the actions and behaviour for both human and computer players. They inherit common attributes and methods from a shared base class but have unique implementations tailored to their specific roles.

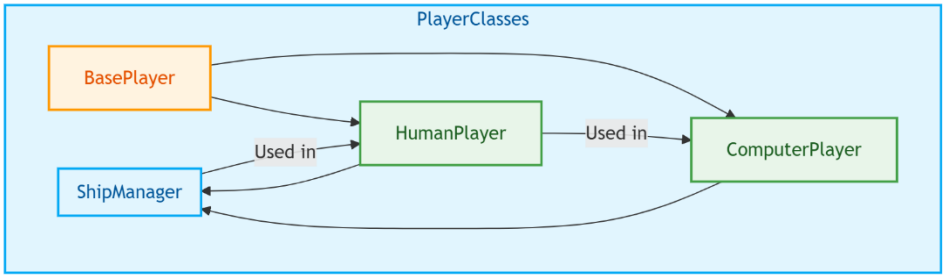
BasePlayer Class

- The BasePlayer class is the base class for both the human and computer players, providing common functionality.
- It initializes the player's name, opponent name, ship manager, attack board, display, and validator.

- The can_place_ship method checks if a ship can be placed on the opponent's grid without overlapping with existing ships.

HumanPlayer Class

- The HumanPlayer class inherits from the BasePlayer class and implements the human player's turn-taking logic.
- The take_turn method prompts the user for a position, validates the input, and processes the attack result (hit or miss).



ComputerPlayer Class

- The ComputerPlayer class inherits from the BasePlayer class and implements the computer player's intelligent targeting system.
- The take_turn method uses a combination of strategies to select the next attack position, including checking the hit stack, using information from the last successful hit, and updating a probability map to target the most likely ship positions.
- The update_probability_map method calculates a heat map of the board, assigning higher probabilities to positions where ships are more likely to be placed.
- The get_move method selects the next attack position based on the current targeting strategy.

4. Graphical User Interface (GUI) Classes

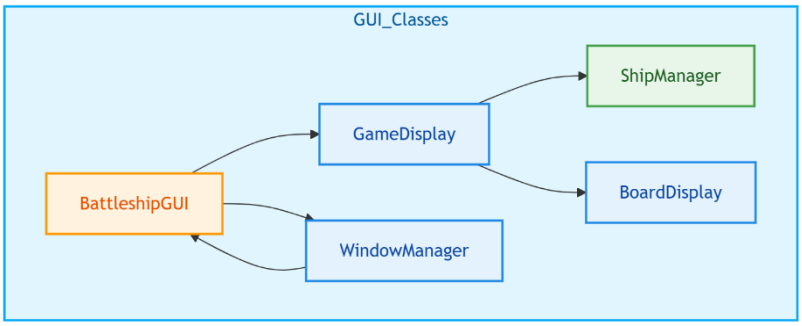
These classes implement the graphical interface of the game, offering an alternative to the console-based interaction by using the tkinter library.

BattleshipGUI Class

- InitializeGame: This initializes the main game, setting the window title, centering the window, creating the custom styles, and setting up a new game.
- TryPlaceShip: This handles the process of trying to place a ship on the placement board. It checks if all ships have been placed, retrieves the current ship and its length, gets the orientation, validates the placement, checks for overlaps, deploys

the ship on the player's grid, updates the placement board, and advances to the next ship placement.

- MakeMove: This handles the process of making a move during the game. It checks if the position has already been attacked, performs the player's attack, updates the player's attack board, updates the player's button style, checks if the attack hit a ship, checks if the computer's ships are all sunk, performs the computer's turn, updates the computer's board, and checks if the player's ships are all sunk.
- StartGame: This starts the game, deploying all ships for the computer, hiding the setup frame, showing the game frame, and updating the game message.
- RestartGame: This handles restarting the game, destroying the game over popup, removing the old game frames, and setting up a new game.



GameDisplay Class

- InitializeGameDisplay: This initializes the game display by creating the setup frame, placement board, setup controls, game frame, and game boards.
- CreatePlacementBoard: This creates the placement board, which is used during the ship placement phase of the game. It creates the placement frame and the individual placement buttons.
- CreateSetupControls: This creates the setup controls, including the orientation controls, message label, and start button.
- CreateGameBoards: This creates the player's and computer's game boards, including the individual buttons and the game message label.

WindowManager Class

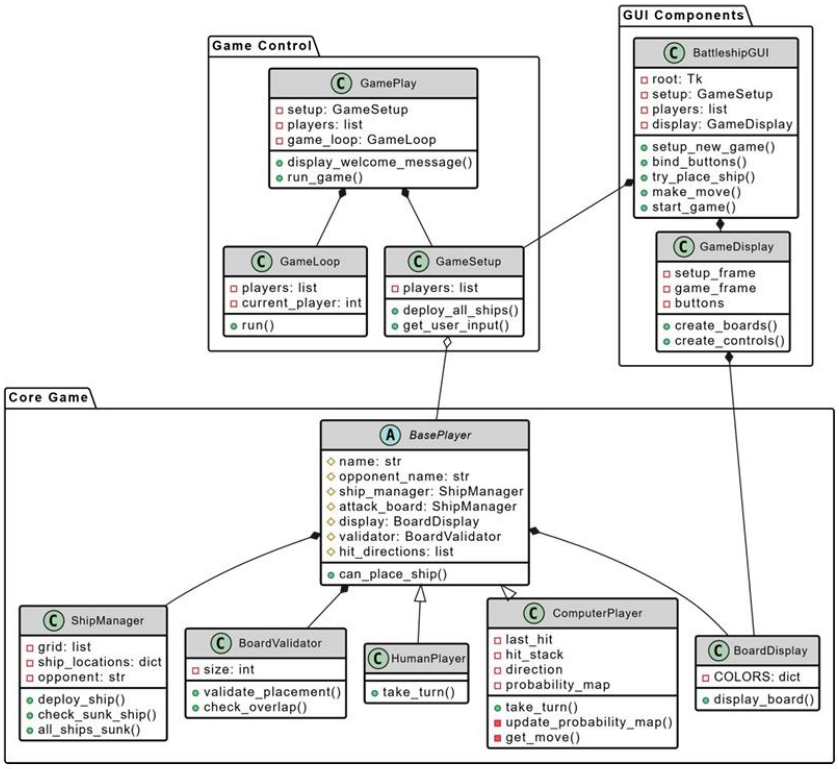
- CreateStyles: This subgraph is responsible for creating the custom styles for the Tkinter buttons, including the 'Ship.TButton', 'Hit.TButton', and 'Miss.TButton' styles.
- CenterWindow: This subgraph handles the centering of the main window on the user's screen. It updates the window's idle tasks, calculates the window's width and height, determines the appropriate x and y coordinates to center the window, and sets the window's geometry accordingly.

Architecture & Workflow

The architecture of the Battleship game is divided into two major components:

System Architecture Diagram

The System Architecture Diagram below illustrates the organization of the game's core modules, divided into three main parts:

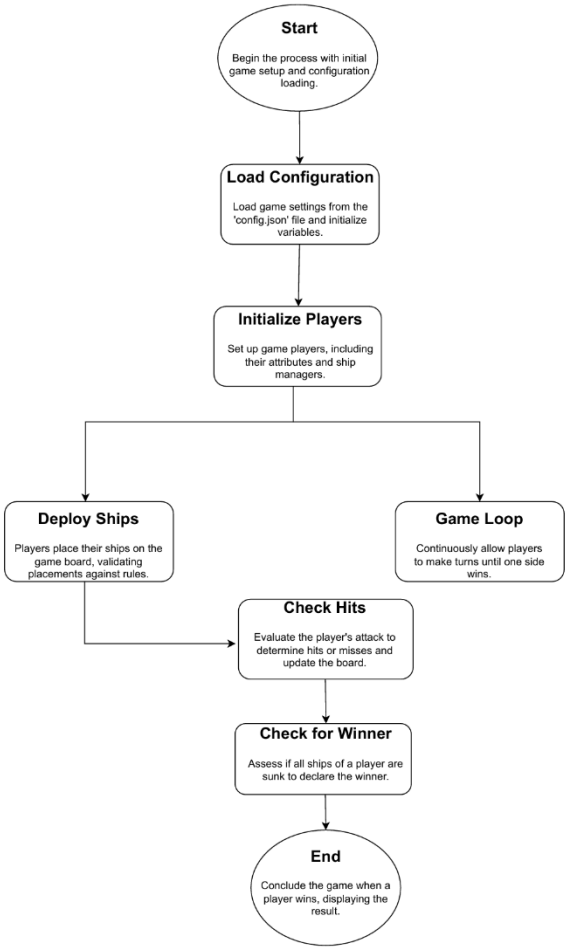


1. **Game Control**: Manages the main gameplay, game setup, and game loop functionalities. The GamePlay, GameLoop, and GameSetup classes coordinate the flow of the game, handle user inputs, and initiate AI turns.
2. **Core Game**: This section implements the logic for player interactions and board management. It includes classes like BasePlayer, HumanPlayer, ComputerPlayer, ShipManager, BoardValidator, and BoardDisplay. The AI's decision-making is handled by the ComputerPlayer class, which utilizes probability mapping for better targeting.
3. **GUI Components**: Uses BattleshipGUI and GameDisplay classes to provide an interactive interface via Tkinter. It manages graphical elements, player inputs, and board updates.

The relationships between classes show how the game logic is effectively separated from user interaction, promoting modularity and scalability.

Flowchart

The Flow Chart outlines the sequential process of the Battleship game from start to end, highlighting the following steps:



1. **Start**: Initializing game settings and configurations.
2. **Load Configuration**: Reading game settings from configuration files and setting initial variables.
3. **Initialize Players**: Setting up player details, including ship managers and ship deployment.
4. **Deploy Ships**: Players place their ships on the game board, validated against rules.
5. **Game Loop**: Continuously alternating between players until one wins.
6. **Check Hits**: Evaluating a player's attack to determine hits, updating the board accordingly.
7. **Check for Winner**: Assessing if all ships of a player are sunk to declare the winner.
8. **End**: Displaying the final result when a player wins.

The flow chart ensures a structured approach to gameplay, clearly defining interactions between user inputs, game states, and display updates.

Testing

The Battleship_Game_UnitTest.py file implements unit testing using Python's built-in unit test framework to ensure the functionality and correctness of various components within the Battleship game project. The classes tested include BoardDisplay, ShipManager, BoardValidator, HumanPlayer, ComputerPlayer, and GameSetup.

Techniques Used

1. **BoardDisplay Class (Black Box Testing)**
    - Tests the initialization of colour codes to verify if required keys like X, -, and RESET are present in the COLORS dictionary.
2. **ShipManager Class (White Box & Black Box Testing)**
    - Tests initialization to confirm proper creation of the board grid and empty ship locations. (White Box Testing)
    - Validates ship deployment by verifying if the grid is updated correctly and the ship is placed as intended. (Black Box Testing)
    - Checks ship sinking detection by hitting all parts of a ship and confirming its removal from ship_locations. (Black Box Testing)
    - Tests overall sinking detection by deploying a ship and then confirming its status after sinking. (Black Box Testing)
3. **BoardValidator Class (White Box Testing)**
    - Validates ship placement within grid boundaries. (White Box Testing)
    - Checks for overlapping ships during deployment by verifying grid occupancy. (White Box Testing)
4. **HumanPlayer Class (White Box Testing)**
    - Tests initialization, ensuring components like ShipManager, BoardDisplay, and BoardValidator are properly created and linked. (White Box Testing)
5. **ComputerPlayer Class (White Box & Gray Box Testing)**
    - Tests initialization to confirm attributes such as name, opponent_name, last_hit, hit_stack, and direction are properly set. (White Box Testing)
    - Validates the probability map initialization for targeting ships. (Gray Box Testing)
6. **GameSetup Class (Black Box & Integration Testing)**
    - Tests the initialization process to confirm that both HumanPlayer and ComputerPlayer are correctly instantiated during game setup. (Black Box Testing & Integration Testing)

Challenges

During the development process, several challenges were encountered:

1. **AI Logic Development:**

    - Implementing a probability-based targeting system which switched between random targeting and targeted search modes.
    - This was achieved through creating an intelligent heatmap system that gets updated in real-time continuously based on successful hits and game patterns.
    - Improving the AI's ability to locate ships by predicting their location and adhering to a standard attack routine was crucial for enhancing gameplay difficulty.

2. **GUI Development:**

    - Establishing a user-friendly and responsive interface with Tkinter was challenging in terms of linking graphical elements with the game logic.
    - Backend logic and user inputs needed to be synchronized properly in order to make the gameplay smooth.
    - Handling various events and updating the graphical interface without affecting performance needed to be designed and optimized carefully.

3. **Input Validation & Error Handling:**

    - Creating strong input validation routines to manage user errors and prevent illegal operations was a necessity.
    - Making sure that invalid inputs were handled gracefully and did not crash the program was important to provide a user-friendly experience.
    - Exceptional handling was implemented extensively to guide users appropriately throughout the gameplay.

4. **Code Organization:**

    - Maintaining modularity using object-oriented principles was a primary task that needed careful planning.
    - Splitting functionalities into various classes and modules ensured scalability and allowed the program to be extended or modified without causing disruptions.
    - This approach also simplified and made the debugging procedure more efficient.

The implemented AI significantly increases gameplay difficulty, providing a sophisticated and demanding experience to the game. The GUI version also supports user experience because it provides an interactive and more graphical interface.

Conclusion

Battleship game illustrates the use of Python programming concepts in building command-line and graphical interface versions of the well-known strategy board game. The utilization of artificial intelligence, graphical user interfaces using Tkinter, and object-oriented programming concepts resulted in a functional, interactive, and easy-to-use program.

The problems during the developmental stage, including AI logic creation, integration with the GUI, input validation, and modularity of the code, were appropriately resolved through thorough planning and testing in iterations. The implementation of the probability-based targeting system of the AI enhanced the game and made playing with the computer increasingly enjoyable and challenging for the user.

The project overall provides a solid foundation to build upon in the future. Some additions that could be made include improving the decision-making capabilities of the AI, adding multiplayer capability, polishing the graphical user interface, and adding different game modes or levels of difficulty.

Overall, this has been a valuable learning exercise that demonstrates how games are created using Python, and how every element of programming comes into play to create a complete working program. Everything learned from the experience of working out all the issues that were encountered will play a significant part in any future projects.

References

1. **Python** - The main programming language used for implementing the game logic and graphical interface.
2. **Tkinter** - A standard Python library used to create the graphical user interface (GUI) for the game.
3. **NumPy** - A powerful Python library used to enhance the AI's decision-making by employing probability-based calculations.
4. **Random Module** - Used for generating random numbers to randomly place ships and select moves for the computer opponent.
5. **JSON Module** - Used to store and load configuration settings for the game such as board size, ship types, and instructions.
6. **VS Code** - The integrated development environment (IDE) used for writing, debugging, and testing the code.
7. Various online tutorials, articles, and official documentation for understanding and implementing object-oriented programming, GUI design, and AI logic in Python.