Specifications and Test Cases for Battleship Game Methods

Below are the detailed specifications and test cases for the methods defined in the Battleship game implementation.

Table of Contents

32. addPotentialTarget
33. getBestArtilleryTarget
34. countUntargetedTilesInArtilleryArea
35. chooseTorpedoTarget
36. getSmokeScreenCoordinateForBot
37. handleEdgeCoordinates

1. initializePlayer

Specification

Purpose: Initializes a Player structure with default values, including setting up empty grids and resetting all counters and flags.

Prototype:

void initializePlayer(Player* player, bool isBot, DifficultyLevel difficulty);

Parameters:

- player: Pointer to the Player structure to initialize.
- isBot: Boolean indicating whether the player is a bot (true) or a human player (false).
- difficulty: The difficulty level for the bot (ignored for human players).

Behavior:

- Initializes the player's grid and tracking grid with empty water ('~').
- Resets all counters, including radar sweeps used, smoke screens used, ships sunk, and ships remaining.
- Sets the availability of special moves (artilleryAvailable, torpedoAvailable) to false.
- Initializes bot-specific fields like potential targets and last artillery coordinates.
- Resets the turn number for the bot.

Test Cases

1. **Test Case 1**: Initialize a human player.

Input:

Player player;

initializePlayer(&player, false, MEDIUM);

Expected Result:

  - player.isBot is false.
  - Grids are initialized with '~'.
  - All counters are set to zero.
  - Special moves are unavailable.
  - Difficulty level is set to MEDIUM.

2. **Test Case 2**: Initialize a bot player at HARD difficulty.

Input:

Player bot;

initializePlayer(&bot, true, HARD);

Expected Result:

  - bot.isBot is true.
  - Grids are initialized with '~'.
  - All counters are set to zero.
  - Special moves are unavailable.
  - Difficulty level is set to HARD.
  - Bot-specific fields are properly initialized.

---

2. initializeGrid

Specification

Purpose: Initializes a game grid by filling it with water ('~').

Prototype:

void initializeGrid(char grid[GRID_SIZE][GRID_SIZE]);

Parameters:

- grid: A 2D array representing the game grid to initialize.

Behavior:

- Sets every cell in the grid to '~'.

Test Cases

1. **Test Case**: Initialize a grid.

Input:

```
char grid[GRID_SIZE][GRID_SIZE];

initializeGrid(grid);
```

Expected Result:

   - All cells in grid contain '~'.

---

3. displayGrid

Specification

Purpose: Displays the game grid to the console, optionally showing ships.

Prototype:

```
void displayGrid(char grid[GRID_SIZE][GRID_SIZE], bool showShips);
```

Parameters:

- grid: The grid to display.
- showShips: If true, displays ships; if false, hides ship symbols.

Behavior:

- Prints the grid with column headers (A-J) and row numbers (1-10).
- Replaces ship symbols with '~' if showShips is false.

Test Cases

1. **Test Case 1**: Display grid without ships.

Input:

```
char grid[GRID_SIZE][GRID_SIZE];

initializeGrid(grid);

grid[0][0] = 'C'; // Place a ship symbol

displayGrid(grid, false);
```

Expected Result:

- The displayed grid shows '~' at position (0,0), hiding the ship.

2. **Test Case 2**: Display grid with ships.

Input:

```
displayGrid(grid, true);
```

Expected Result:

- The displayed grid shows 'C' at position (0,0), revealing the ship.

---

4. placeShips

Specification

Purpose: Allows a human player to place their ships on the grid.

Prototype:

```
void placeShips(Player* player, Fleet* fleet);
```

Parameters:

- player: Pointer to the player placing ships.
- fleet: Pointer to the player's fleet.

Behavior:

- Prompts the player to input coordinates and orientation for each ship.
- Validates placement and updates the player's grid accordingly.
- Clears the screen after successful placement.

Note:

- If player->isBot is true, calls placeShipsBot instead.

Test Cases

1. **Test Case**: Simulate ship placement (assuming valid inputs).

Input:

  - Player inputs: "A1 h" for Carrier, "B2 v" for Battleship, etc.

Expected Result:

  - Ships are placed at the specified coordinates with the correct orientation.
  - Grid is updated accordingly.

---

5. placeShipsBot

Specification

Purpose: Automatically places ships for a bot player randomly on the grid.

Prototype:

void placeShipsBot(Player* bot, Fleet* fleet);

Parameters:

- bot: Pointer to the bot player.
- fleet: Pointer to the bot's fleet.

Behavior:

- Randomly selects starting coordinates and orientations for each ship.

- Ensures ships do not overlap and are within grid boundaries.
- Updates the bot's grid with ship placements.

Test Cases

1. **Test Case**: Verify that ships are placed without overlapping.

Input:

Player bot;

Fleet fleet;

initializePlayer(&bot, true, MEDIUM);

placeShipsBot(&bot, &fleet);

Expected Result:

- All ships are placed on the bot's grid.
- No ships overlap.
- All ships are within the grid boundaries.

---

6. isValidPlacement

Specification

Purpose: Checks if a ship can be placed at the specified location without overlapping and within grid boundaries.

Prototype:

bool isValidPlacement(char grid[GRID_SIZE][GRID_SIZE], Coordinate coord, int size, char orientation);

Parameters:

- grid: The grid to check.

- coord: Starting coordinate for the ship.
- size: Size of the ship.
- orientation: 'h' for horizontal, 'v' for vertical.

Returns:

- true if the placement is valid.
- false otherwise.

Behavior:

- Verifies that the ship fits within the grid.
- Checks that the placement does not overlap with existing ships.

Test Cases

1. **Test Case 1**: Valid horizontal placement.

Input:

Coordinate coord = {0, 0};

int size = 5;

char orientation = 'h';

bool result = isValidPlacement(grid, coord, size, orientation);

Expected Result:

- result is true if positions (0,0) to (4,0) are empty.

2. **Test Case 2**: Invalid vertical placement (overlapping).

Input:

grid[1][0] = 'C'; // Existing ship

Coordinate coord = {0, 0};

int size = 4;

char orientation = 'v';

bool result = isValidPlacement(grid, coord, size, orientation);

Expected Result:

- result is false due to overlap at (0,1).

3. **Test Case 3**: Invalid placement (out of bounds).

Input:

Coordinate coord = {8, 0};

int size = 3;

char orientation = 'h';

bool result = isValidPlacement(grid, coord, size, orientation);

Expected Result:

- result is false because the ship would extend beyond column 9.

---

7. placeShipOnGrid

Specification

Purpose: Places a ship on the grid at the specified location and orientation.

Prototype:

void placeShipOnGrid(char grid[GRID_SIZE][GRID_SIZE], Coordinate coord, int size, char orientation, char symbol);

Parameters:

- grid: The grid to update.
- coord: Starting coordinate for the ship.
- size: Size of the ship.

- orientation: 'h' for horizontal, 'v' for vertical.
- symbol: Character symbol representing the ship.

Behavior:

- Updates the grid cells with the ship's symbol along the specified orientation and size.

Test Cases

1. **Test Case**: Place a ship on the grid.

Input:

Coordinate coord = {0, 0};

int size = 3;

char orientation = 'v'; char

symbol = 'D';

placeShipOnGrid(grid, coord, size, orientation, symbol);

Expected Result:

- Grid cells at positions (0,0), (0,1), (0,2) contain 'D'.

---

8. parseCoordinate

Specification

Purpose: Parses a coordinate string (e.g., "A5") into a Coordinate struct.

Prototype:

Coordinate parseCoordinate(const char* input);

Parameters:

- input: String representing the coordinate.

Returns:

- Coordinate with valid x and y if parsing is successful.
- Coordinate with x = -1 and y = -1 if invalid.

Behavior:

- Converts column letter to x index (0-9).
- Converts row number to y index (0-9).

Test Cases

1. **Test Case 1**: Valid coordinate.

Input:

Coordinate coord = parseCoordinate("A1");

Expected Result:

   - coord.x is 0, coord.y is 0.

2. **Test Case 2**: Invalid coordinate (out of bounds).

Input:

Coordinate coord = parseCoordinate("K11");

Expected Result:

   - coord.x is -1, coord.y is -1.

3. **Test Case 3**: Invalid format.

Input:

Coordinate coord = parseCoordinate("1A");

Expected Result:

   - coord.x is -1, coord.y is -1.

---

9. clearScreen

Specification

Purpose: Clears the console screen.

Prototype:

void clearScreen();

Behavior:

- Executes the appropriate system command to clear the console screen based on the operating system.

Test Cases

1. **Test Case**: Invoke clearScreen and verify that the console is cleared.

Input:

clearScreen();

Expected Result:

  - Console screen is cleared.

---

10. gameLoop

Specification

Purpose: Main game loop handling turns and checking for win conditions.

Prototype:

void gameLoop(Player* currentPlayer, Player* opponent, Fleet* currentFleet, Fleet* opponentFleet, bool hardMode);

Parameters:

- currentPlayer: Pointer to the player whose turn it is.
- opponent: Pointer to the opponent player.

- currentFleet: Pointer to the current player's fleet.
- opponentFleet: Pointer to the opponent's fleet.
- hardMode: Boolean indicating if hard mode is enabled (affects tracking grid visibility).

Behavior:

- Alternates turns between players.
- Invokes performMove or performBotMove based on the player type.
- Checks for win conditions after each turn.
- Ends the loop when a player wins.

Test Cases

1. **Test Case**: Simulate a game loop where a player wins after sinking all opponent's ships.

Input:

- Initialize players and fleets.
- Set up conditions for a quick game (e.g., small grid or pre-determined moves).

Expected Result:

- Game loop runs until checkWin returns true.
- The winning player's name is printed.

---

11. performMove

Specification

Purpose: Handles a human player's move, processing commands and executing actions.

Prototype:

void performMove(Player* player, Player* opponent, Fleet* opponentFleet, bool hardMode);

Parameters:

- player: Pointer to the player making the move.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- hardMode: Boolean indicating if hard mode is enabled.

Behavior:

- Displays the player's tracking grid.
- Prompts the player for a move (fire, radar, smoke, artillery, torpedo).
- Validates the input command and arguments.
- Executes the move and updates the game state.
- Provides feedback to the player about the result.

Test Cases

1. **Test Case 1**: Player fires at a valid coordinate and hits a ship.

Input:

  - Command: "fire A1"

Expected Result:

  - If a ship is at (0,0), reports "Hit!".
  - Updates tracking grid.

2. **Test Case 2**: Player enters an invalid command.

Input:

  - Command: "fly A1"

Expected Result:

  - Reports "Invalid command or command not available."
  - Player loses their turn.

12. performBotMove

Specification

Purpose: Handles the bot's move based on its difficulty level.

Prototype:

void performBotMove(Player* bot, Player* opponent, Fleet* opponentFleet, bool hardMode);

Parameters:

- bot: Pointer to the bot player.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- hardMode: Boolean indicating if hard mode is enabled.

Behavior:

- Determines the bot's action based on its difficulty level.
- Uses strategies appropriate for EASY, MEDIUM, or HARD difficulties.
- Executes special moves or fires at the player's grid.
- Updates the game state and provides feedback.

Test Cases

1. **Test Case**: Bot in EASY difficulty performs a random fire.

Input:

- Bot's difficulty set to EASY.

Expected Result:

- Bot selects a random untargeted coordinate.
- Updates tracking grid accordingly.

13. fire

Specification

Purpose: Fires at a specified coordinate, updating grids and ship statuses.

Prototype:

int fire(Player* player, Player* opponent, Fleet* opponentFleet, Coordinate coord, bool hardMode, char* sunkShipName);

Parameters:

- player: Pointer to the player making the fire.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- coord: Coordinate to fire at.
- hardMode: Boolean indicating if hard mode is enabled.
- sunkShipName: Buffer to store the name of a sunk ship, if any.

Returns:

- 0: Miss.
- 1: Hit.
- 2: Hit and sunk a ship.
- 3: Already targeted.
- -1: Invalid or ineffective shot.

Behavior:

- Checks if the coordinate is under a smoke screen.
- Updates opponent's grid based on the result.

- Updates player's tracking grid if appropriate.
- Updates ship hit counts and sunk status.

Test Cases

1. **Test Case 1**: Fire at an empty cell.

Input:

Coordinate coord = {0, 0};

int result = fire(player, opponent, opponentFleet, coord, false, sunkShipName);

Expected Result:

- result is 0 (Miss).
- Opponent's grid at (0,0) is updated to 'o'.

2. **Test Case 2**: Fire at a ship and sink it.

Input:

- Coordinate of the last remaining part of a ship.

Expected Result:

- result is 2 (Hit and sunk).
- sunkShipName contains the name of the sunk ship.
- Ship's sunk status is updated.

---

14. radarSweep

Specification

Purpose: Performs a radar sweep at the specified coordinate.

Prototype:

void radarSweep(Player* player, Player* opponent, Coordinate coord);

Parameters:

- player: Pointer to the player performing the radar sweep.
- opponent: Pointer to the opponent player.
- coord: Coordinate where the radar sweep is deployed.

Behavior:

- Checks a 2x2 area around the coordinate for enemy ships.
- Considers active smoke screens that may obscure the area.
- Reports to the player whether enemy ships were detected.
- If the area is obscured by smoke, the radar sweep is ineffective, and the smoke screen deactivates.

Test Cases

1. **Test Case**: Radar sweep detects enemy ships.

Input:

- ○ Radar sweep at coordinate where opponent has ships within the area.

Expected Result:

- ○ Reports "Radar detected enemy ships near the target area."
- ○ If the player is a bot, potential targets are added.

2. **Test Case**: Radar sweep area is covered by a smoke screen.

Input:

- ○ Radar sweep at coordinate covered by an active smoke screen.

Expected Result:

- ○ Reports "Radar sweep found no enemy ships (area obscured by smoke)."
- ○ Smoke screen deactivates.

15. smokeScreen

Specification

Purpose: Deploys a smoke screen at the specified coordinate.

Prototype:

```
bool smokeScreen(Player* player, Coordinate coord);
```

Parameters:

- player: Pointer to the player deploying the smoke screen.
- coord: Coordinate where the smoke screen is deployed.

Returns:

- true if successfully deployed.
- false otherwise.

Behavior:

- Checks if the player has smoke screens available.
- Deploys the smoke screen, activating it over a 2x2 area.
- Updates the player's smoke screen data.

Test Cases

1. **Test Case 1**: Deploy smoke screen successfully.

Input:

```
player.shipsSunk = 1;

player.smokeScreensUsed = 0;

bool result = smokeScreen(&player, (Coordinate){0, 0});
```

Expected Result:

  - result is true.
  - Smoke screen is active at the specified coordinate.

2. **Test Case 2**: Fail to deploy smoke screen (none available).

Input:

player.shipsSunk = 0;

player.smokeScreensUsed = 0;

bool result = smokeScreen(&player, (Coordinate){0, 0});

Expected Result:

  - result is false.
  - Reports "No smoke screens available."

---

16. artillery

Specification

Purpose: Performs an artillery strike at the specified coordinate.

Prototype:

void artillery(Player* player, Player* opponent, Fleet* opponentFleet, Coordinate coord, bool hardMode);

Parameters:

- player: Pointer to the player performing the artillery strike.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- coord: Coordinate where the artillery strike is deployed.
- hardMode: Boolean indicating if hard mode is enabled.

Behavior:

- Attacks a 2x2 area centered around the coordinate.
- Updates grids and ship statuses based on hits.
- Reports total hits and misses.
- If ships are sunk, updates the game state accordingly.

Test Cases

1. **Test Case**: Perform artillery strike and sink a ship.

Input:

- Artillery strike at coordinate covering the last parts of a ship.

Expected Result:

- Reports hits and misses.
- Sunk ship is reported.
- Ship's sunk status is updated.

---

17. torpedo

Specification

Purpose: Performs a torpedo attack on a specified row or column.

Prototype:

void torpedo(Player* player, Player* opponent, Fleet* opponentFleet, const char* input, bool hardMode);

Parameters:

- player: Pointer to the player performing the torpedo attack.
- opponent: Pointer to the opponent player.

- opponentFleet: Pointer to the opponent's fleet.
- input: String indicating the row (number) or column (letter) to attack.
- hardMode: Boolean indicating if hard mode is enabled.

Behavior:

- Determines if the input is a row or column.
- Attacks all cells in the specified row or column.
- Updates grids and ship statuses based on hits.
- Reports total hits and misses.
- If ships are sunk, updates the game state accordingly.

Test Cases

1. **Test Case 1**: Torpedo attack on a row.

Input:

torpedo(player, opponent, opponentFleet, "5", false);

Expected Result:

  - Attacks row 5.
  - Reports hits and misses.
  - Updates grids accordingly.

2. **Test Case 2**: Torpedo attack on an invalid column.

Input:

torpedo(player, opponent, opponentFleet, "K", false);

Expected Result:

  - Reports "Invalid column."
  - No action taken.

18. checkWin

Specification

Purpose: Checks if all ships in the fleet are sunk.

Prototype:

bool checkWin(Fleet* fleet);

Parameters:

- fleet: Pointer to the fleet to check.

Returns:

- true if all ships are sunk.
- false otherwise.

Behavior:

- Iterates through all ships in the fleet.
- Returns false if any ship is not sunk.

Test Cases

1. **Test Case**: Fleet with all ships sunk.

Input:

- All ships in fleet have sunk set to true.

Expected Result:

- checkWin(fleet) returns true.

2. **Test Case**: Fleet with at least one ship not sunk.

Input:

- At least one ship in fleet has sunk set to false.

Expected Result:

- checkWin(fleet) returns false.

---

19. updateShipStatus

Specification

Purpose: Updates the sunk status of a ship based on hits.

Prototype:

void updateShipStatus(Ship* ship);

Parameters:

- ship: Pointer to the ship to update.

Behavior:

- Sets ship->sunk to true if ship->hits >= ship->size.

Test Cases

1. **Test Case**: Ship has hits equal to its size.

Input:

ship.size = 3;

ship.hits = 3;

updateShipStatus(&ship);

Expected Result:

- ship.sunk is true.

2. **Test Case**: Ship has fewer hits than its size.

Input: ship.size

= 4;

ship.hits = 2;

updateShipStatus(&ship);

Expected Result:

- ship.sunk is false.

---

20. unlockSpecialMoves

Specification

Purpose: Unlocks special moves for the player based on game conditions.

Prototype:

void unlockSpecialMoves(Player* player, Player* opponent);

Parameters:

- player: Pointer to the player unlocking special moves.
- opponent: Pointer to the opponent player.

Behavior:

- Unlocks artilleryAvailable after sinking a ship.
- Unlocks torpedoAvailable when the opponent has only one ship remaining.
- Allows additional smoke screens based on the number of ships sunk.

Test Cases

1. **Test Case**: Player sinks a ship; artillery becomes available.

Input:

player.artilleryAvailable = false;

opponent.shipsRemaining = 3;

unlockSpecialMoves(&player, &opponent);

Expected Result:

- player.artilleryAvailable is set to true.

2. **Test Case**: Opponent has one ship remaining; torpedo becomes available.

Input:

player.torpedoAvailable = false;

opponent.shipsRemaining = 1;

unlockSpecialMoves(&player, &opponent); **Expected**

Result:

- player.torpedoAvailable is set to true.

---

21. displayTrackingGrid

Specification

Purpose: Displays the player's tracking grid, showing the results of their attacks on the opponent.

Prototype:

void displayTrackingGrid(Player* player, bool hardMode);

Parameters:

- player: Pointer to the player whose tracking grid is to be displayed.
- hardMode: Boolean indicating if hard mode is enabled (affects ship visibility).

Behavior:

- Calls displayGrid with the player's tracking grid.
- In hard mode, hides ship symbols; otherwise, shows them.

Test Cases

1. **Test Case 1**: Display tracking grid in normal mode.

Input:

displayTrackingGrid(&player, false);

Expected Result:

  - The tracking grid is displayed with ships visible (if any have been detected).

2. **Test Case 2**: Display tracking grid in hard mode.

Input:

displayTrackingGrid(&player, true);

Expected Result:

  - The tracking grid is displayed with ships hidden, showing '~' instead of ship symbols.

---

22. isValidCommand

Specification

Purpose: Validates if a command is valid and available for the player.

Prototype:

bool isValidCommand(const char* command, Player* player);

Parameters:

- command: String representing the command input by the player.
- player: Pointer to the player attempting to execute the command.

Returns:

- true if the command is valid and available.
- false otherwise.

Behavior:

- Checks if the command is among the allowed commands: "fire", "radar", "smoke", "artillery", "torpedo".
- Verifies if the player has the special move available (e.g., artillery is unlocked).

Test Cases

1. **Test Case 1**: Valid command "fire".

Input:

```
bool result = isValidCommand("fire", &player);
```

Expected Result:

- result is true.

2. **Test Case 2**: Command "torpedo" when torpedo is not available.

Input:

```
player.torpedoAvailable = false;

bool result = isValidCommand("torpedo", &player);
```

Expected Result:

- result is false.

3. **Test Case 3**: Invalid command "fly".

Input:

```
bool result = isValidCommand("fly", &player);
```

Expected Result:

- result is false.

---

23. getInput

Specification

Purpose: Safely gets input from the user, handling buffer overflows and input termination.

Prototype:

void getInput(char* input, int size);

Parameters:

- input: Buffer to store the input.
- size: Size of the input buffer.

Behavior:

- Reads a line from stdin into the input buffer.
- Removes the trailing newline character.
- Flushes the input buffer if the input exceeds the buffer size.

Test Cases

1. **Test Case**: User inputs "fire A1".

Input:

char input[50];

// Simulate user input "fire A1\n"

// (In actual unit testing, this may involve mocking stdin) getInput(input,

sizeof(input));

Expected Result:

  - input contains "fire A1".

---

24. coordinateToString

Specification

Purpose: Converts a Coordinate struct to a string representation (e.g., {0,4} -> "A5").

Prototype:

```
void coordinateToString(Coordinate coord, char* coordStr);
```

Parameters:

- coord: The coordinate to convert.
- coordStr: Buffer to store the string representation.

Behavior:

- Converts the x index to a column letter (A-J).
- Converts the y index to a row number (1-10).
- Stores the result in coordStr.

Test Cases

1. **Test Case**: Convert coordinate {2, 4}.

Input:

```
Coordinate coord = {2, 4};

char coordStr[5];

coordinateToString(coord, coordStr);
```

Expected Result:

  - coordStr contains "C5".

2. **Test Case**: Convert coordinate {9, 9}.

Input:

```
Coordinate coord = {9, 9};

char coordStr[5];

coordinateToString(coord, coordStr);
```

Expected Result:

- coordStr contains "J10".

---

25. toLowerCase

Specification

Purpose: Converts a string to lowercase.

Prototype:

void toLowerCase(char* str);

Parameters:

- str: The string to convert.

Behavior:

- Converts all uppercase letters in str to lowercase.

Test Cases

1. **Test Case**: Convert "Fire A1".

Input:

char str[] = "Fire A1";

toLowerCase(str); **Expected**

Result:

- str is "fire a1".

---

26. flushInputBuffer

Specification

Purpose: Flushes the input buffer to remove any extraneous input.

Prototype:

void flushInputBuffer();

Behavior:

- Reads and discards characters from stdin until a newline or EOF is encountered.

Test Cases

1. **Test Case**: Simulate overflowing input and ensure buffer is flushed.

Input:

// User inputs a long string exceeding buffer size

flushInputBuffer();

Expected Result:

- Input buffer is cleared.

---

27. getRandomNumber

Specification

Purpose: Generates a random number between min and max (inclusive).

Prototype:

int getRandomNumber(int min, int max);

Parameters:

- min: Minimum value.
- max: Maximum value.

Returns:

- A random integer between min and max.

Test Cases

1. **Test Case**: Generate random number between 1 and 10.

Input:

int num = getRandomNumber(1, 10);

Expected Result:

   - num is an integer between 1 and 10.

---

28. getRandomCoordinate

Specification

Purpose: Generates a random coordinate within the grid.

Prototype:

Coordinate getRandomCoordinate();

Returns:

- A Coordinate with x and y between 0 and GRID_SIZE - 1.

Test Cases

1. **Test Case**: Generate random coordinate.

Input:

Coordinate coord = getRandomCoordinate();

Expected Result:

   - coord.x and coord.y are integers between 0 and 9.

---

29. getNextTarget

Specification

Purpose: Selects the next target for the bot by choosing the coordinate with the highest probability based on the probability grid.

Prototype:

Coordinate getNextTarget(Player* bot, Fleet* opponentFleet);

Parameters:

- bot: Pointer to the bot player.
- opponentFleet: Pointer to the opponent's fleet.

Returns:

- A Coordinate representing the next target.

Behavior:

- Calculates a probability grid using calculateProbabilityGrid.
- Selects coordinates with the highest probability that haven't been targeted yet.
- If multiple coordinates have the same highest probability, selects one at random.

Test Cases

1. **Test Case**: Bot selects next target based on probability.

Input:

Coordinate target = getNextTarget(&bot, &opponentFleet);

Expected Result:

  - target is a valid coordinate.
  - The coordinate corresponds to a cell with the highest calculated probability.

---

30. calculateProbabilityGrid

Specification

Purpose: Calculates a probability grid representing the likelihood of each cell containing a ship.

Prototype:

```
void calculateProbabilityGrid(Player* bot, Fleet* opponentFleet, int
probabilityGrid[GRID_SIZE][GRID_SIZE]);
```

Parameters:

- bot: Pointer to the bot player.
- opponentFleet: Pointer to the opponent's fleet.
- probabilityGrid: 2D array to store the calculated probabilities.

Behavior:

- Considers the bot's tracking grid and remaining ships in the opponent's fleet.
- Assigns higher probabilities to cells where ships are more likely to be based on remaining ship sizes and previous hits.
- Increases probability for cells adjacent to hits.

Test Cases

1. **Test Case**: Probability grid reflects higher likelihood near hits.

Input:

```
// Assume bot's tracking grid has a hit at (5,5)
bot.trackingGrid[5][5] = '*';
int probabilityGrid[GRID_SIZE][GRID_SIZE];
calculateProbabilityGrid(&bot, &opponentFleet, probabilityGrid);
```

Expected Result:

    - Cells adjacent to (5,5) have higher probability values.
    - Cells with misses ('o') have zero probability.

31. addAdjacentTargets

Specification

Purpose: Adds adjacent tiles to the bot's potential target queue after a successful hit, considering ship orientation.

Prototype:

```
void addAdjacentTargets(Player* bot, Coordinate coord);
```

Parameters:

- bot: Pointer to the bot player.
- coord: Coordinate where the hit occurred.

Behavior:

- If aligned hits are found, continues targeting in that direction.
- Adds valid adjacent coordinates to bot->potentialTargets.

Test Cases

1. **Test Case**: Add adjacent targets after a hit.

Input:

```
// Assume bot hit at (5,5) Coordinate

hitCoord = {5, 5};

addAdjacentTargets(&bot, hitCoord);
```

Expected Result:

  - Coordinates (5,6), (6,5), (5,4), (4,5) are added to bot->potentialTargets if not already targeted.

2. **Test Case**: Extend search in a specific direction.

Input:

```
// Bot has hits at (5,5) and (5,6)

bot.trackingGrid[5][5] = '*';

bot.trackingGrid[5][6] = '*';

Coordinate hitCoord = {5, 6};

addAdjacentTargets(&bot, hitCoord);
```

Expected Result:

- Only adds coordinates in the same column (vertical direction), e.g., (5,7), to bot->potentialTargets.

---

32. addPotentialTarget

Specification

Purpose: Adds a new potential target to the bot's target queue if it hasn't been added already.

Prototype:

```
void addPotentialTarget(Player* player, Coordinate coord);
```

Parameters:

- player: Pointer to the player (bot).
- coord: Coordinate to add to the potential target queue.

Behavior:

- Checks if the coordinate is already in player->potentialTargets.
- Adds it to the queue if it's not already present.

Test Cases

1. **Test Case**: Add a new potential target.

Input:

Coordinate coord = {5, 5};

addPotentialTarget(&bot, coord);

Expected Result:

- coord is added to bot->potentialTargets.

2. **Test Case**: Attempt to add a duplicate potential target.

Input:

addPotentialTarget(&bot, coord); // coord already added

Expected Result:

- coord is not added again.
- bot->potentialTargetCount remains the same.

---

33. getBestArtilleryTarget

Specification

Purpose: Determines the optimal 2x2 area for deploying an artillery strike based on untargeted tiles.

Prototype:

Coordinate getBestArtilleryTarget(Player* bot);

Parameters:

- bot: Pointer to the bot player.

Returns:

- A Coordinate representing the top-left corner of the best artillery target area.

Behavior:

- Scans the grid for 2x2 areas with the highest number of untargeted ('~') tiles.
- Returns the coordinate of the area with the maximum untargeted tiles.

Test Cases

1. **Test Case**: Bot selects artillery target area with maximum untargeted tiles.

Input:

Coordinate target = getBestArtilleryTarget(&bot);

Expected Result:

  - target is the coordinate of a 2x2 area with the most untargeted tiles.

---

34. countUntargetedTilesInArtilleryArea

Specification

Purpose: Counts the number of untargeted tiles within a 2x2 artillery strike area.

Prototype:

int countUntargetedTilesInArtilleryArea(Player* bot, Coordinate coord);

Parameters:

- bot: Pointer to the bot player.
- coord: Coordinate representing the top-left corner of the artillery area.

Returns:

- Number of untargeted ('~') tiles in the specified area.

Behavior:

- Adjusts for grid boundaries.
- Counts untargeted tiles within the 2x2 area.

Test Cases

1. **Test Case**: Count untargeted tiles in a given area.

Input:

Coordinate coord = {5, 5};

int count = countUntargetedTilesInArtilleryArea(&bot, coord);

Expected Result:

- count reflects the number of untargeted tiles in the area starting at (5,5).

---

35. chooseTorpedoTarget

Specification

Purpose: Selects the best row or column to deploy a torpedo based on untargeted tiles.

Prototype:

bool chooseTorpedoTarget(Player* bot, Player* opponent, Fleet* opponentFleet, bool hardMode);

Parameters:

- bot: Pointer to the bot player.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- hardMode: Boolean indicating if hard mode is enabled.

Returns:

- true if a torpedo was successfully deployed.
- false otherwise.

Behavior:

- Evaluates all rows and columns to find the one with the most untargeted tiles.

- Deploys torpedo attack on the optimal row or column.

Test Cases

1. **Test Case**: Bot selects a row with the most untargeted tiles.

Input:

bool success = chooseTorpedoTarget(&bot, &opponent, &opponentFleet, false);

Expected Result:

- Torpedo is deployed on the selected row or column.
- success is true.

2. **Test Case**: No valid torpedo targets available.

Input:

// All rows and columns have been targeted

bool success = chooseTorpedoTarget(&bot, &opponent, &opponentFleet, false);

Expected Result:

- success is false.

---

36. getSmokeScreenCoordinateForBot

Specification

Purpose: Determines the best coordinate for the bot to deploy a smoke screen.

Prototype:

```
Coordinate getSmokeScreenCoordinateForBot(Player* bot);
```

Parameters:

- bot: Pointer to the bot player.

Returns:

- A Coordinate representing the best smoke screen location.
- { -1, -1 } if no suitable location is found.

Behavior:

- Scans the bot's grid for areas containing ships.
- Prefers areas where multiple ships are present.
- Returns the coordinate for deploying the smoke screen.

Test Cases

1. **Test Case**: Bot selects a coordinate over its ships.

Input:

```
Coordinate coord = getSmokeScreenCoordinateForBot(&bot);
```

Expected Result:

  - coord corresponds to a location where the bot has ships.

2. **Test Case**: No ships remaining; bot cannot deploy smoke screen.

Input:

```
// All bot's ships are sunk

Coordinate coord = getSmokeScreenCoordinateForBot(&bot);
```

Expected Result:

- coord is { -1, -1 }.

---

37. handleEdgeCoordinates

Specification

Purpose: Adjusts coordinate start and end values to prevent out-of-bounds access.

Prototype:

```
void handleEdgeCoordinates(int* start, int* end);
```

Parameters:

- start: Pointer to the starting index.
- end: Pointer to the ending index.

Behavior:

- Ensures that start is not less than 0.
- Ensures that end does not exceed GRID_SIZE - 1.

Test Cases

1. **Test Case**: Adjust coordinates at grid boundaries.

Input:

```
int xStart = -1;

int xEnd = 10;

handleEdgeCoordinates(&xStart, &xEnd);
```

Expected Result:

- xStart is adjusted to 0.
- xEnd is adjusted to GRID_SIZE - 1 (9).