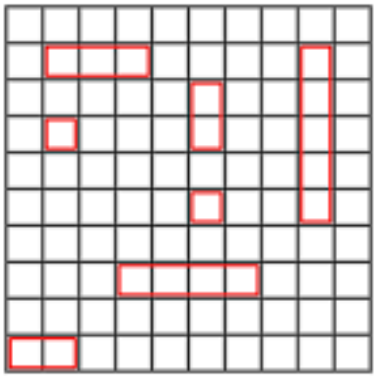
Programming Project 08

This assignment is worth 40 points (4.0% of the course grade) and must be completed and turned in before 11:59 on Monday, December 04, 2023 (2 weeks). After the due date, your score will be deducted by 25% for every 12 hours late or a fraction of it.

1 Assignment Overview

In this project, you will program the game of Battleship (Game Overview). The goal of this project is to develop an understanding of how to create, use, and organize classes to build software.

(a) Original Battleship Game



(b) 2D Version

1.1 Rules of Battleship

The game of Battleship is played between two players. Each player has an $n \times n$ sized board and starts with a specified number of ships of different sizes. (e.g., one ship of length 5, one ship of length 4, two ships of length 3, and one ship of length 2).

Part One In the first part of the game, each player places ships on their board. Ships may be placed horizontally or vertically (but not diagonally). Additionally, ships may not overlap each other.

Part Two Once both players have placed their ships on their boards, they will take turns making guesses. The goal of the game is to guess the positions of your opponent's ships, without being able to see the opponent's board. A single guess is a coordinate `(row, col)` of a position on the opponents board. A guess that is at the location of a ship is called a *hit*. Otherwise, the guess is a *miss*. Players cannot guess the same position twice (this would also be a waste of a turn). If a ship has been hit in all of its positions, then it is considered *sunk*. The game ends when one player sinks all of another player's ships.

2 Deliverables

The deliverables for this assignment are the following files:

- `board.py` – the source code for your `Ship` and `Board`, Classes
- `game.py` – the source code for your `Player` and `BattleshipGame` Classes
- `proj08.py` – the source code for your Python program which includes the main function

Be sure to use the specified file names and to submit all the 3 files for grading through Codio before the project deadline.

3 Instructions

For this project, you will implement four classes: `Ship`, `Board`, `Player`, and `BattleshipGame`. Each class is described next and implemented in a particular file.

Important Notes

- All class attributes are considered **public** unless stated otherwise.
- You must use the exact names specified for the attributes.
- Use `try-except` statements to print any exceptions raised by invalid input. Ex:

```
try:
    # code here
except Exception as e:
    print(e)
```

3.1 Ship Class

The `Ship` class represents a 'piece' in the Battleship game. The class should be implemented in the `board.py` file. You must implement the following attributes and methods for the `Ship` class:

`__init__() -> None`

This method creates a `Ship` piece. Only length, position, and orientation must be provided as arguments in this order when initializing this class (remember to also use `self`). You must define the following attributes (use these exact names) :

- `length: int` — Represents the length of the ship.
- `orientation: str` — Represents the orientation (`"h"` for horizontal or `"v"` for vertical) of the ship.
- `positions: list[tuple(int,int)]` — Represents the positions on the board occupied by the ship. This is a list of tuples. The length, position and orientation can be used to initialize the `positions` attribute of a ship. For example, if a Ship is initialized with position $(0, 1)$, length 3, and orientation 'h', then:

  `positions = [(0,1), (0,2), (0,3)]`

  if a Ship is initialized with position $(0, 1)$, length 3, and orientation 'v', then:

  `positions = [(0,1), (1,1), (2,1)]`
- `hit_count: int` — Represents the number of 'hits' on the ship. Initialize this to `0`.
- `is_sunk: bool` — Represents the status of the ship (if it is sunk or not). Initialize this to `False`. This attribute is `True` if all positions on the current ship are *hit*.

`def get_positions() -> list`

The `get_positions` method returns the list of positions of the Ship. The only parameter to use is `self`.

`def get_orientation() -> str`

The `get_orientation` method returns the orientation of the Ship. The only parameter to use is `self`.

`def apply_hit() -> None`

The `apply_hit` function increases the `hit_count` by 1 and checks if the ship is *sunk* (all positions are hit). If it is sunk, then you should update the `is_sunk` attribute. The only parameter to use is `self`.

3.2 Board Class

The `Board` class represents the board object. The class should be implemented in the `board.py` file. The board class will also keep track of the ships on the board, and assist with validating player moves. You must implement the following attributes and methods for the `Board` class:

`def __init__()`

This method creates a `Board` object. Only the `size` integer must be provided when creating this class as an argument in addition to self. You must define the following attributes (remember to use `self`):

- `size: int` — Represents the dimensions of the square board.
- `board: list[list[str]]` — Represents the data structure that will store the board state. The board is a list of lists, where each list will have `size` number of spaces (i.e., `" "`), and there will be a total of `size` number of such lists.
- `ships: list[Ship]` — Represents the list of ships on the current board. Initialize this to an empty list.

`def place_ship() -> None`

This function takes as a parameter `Ship` object (in addition to self) and updates the `ships` list. Additionally, this updates the `board` variable with the character `"S"` for each location occupied by the ship (hint: what function from the `Ship` class can help get a list of a ship's positions?). You may assume the ship has already been validated to be placed onto the board.

`def validate_ship_coordinates() -> None`

This method takes as a parameter a `Ship` object (in addition to self) and checks if it can be placed on the current board. You must ensure the following conditions are not violated:

1. Ship coordinates are valid positions on the board. If a Ship coordinate is not on the board, then you must raise the following exception:

   ```
   raise RuntimeError("Ship coordinates are out of bounds!")
   ```

2. Ship coordinates are not already occupied by another ship. If a Ship coordinate is already occupied, you must raise the following exception:

   ```
   raise RuntimeError("Ship coordinates are already taken!")
   ```

`def apply_guess() -> None`

This function takes as a parameter a tuple `guess` with two integers (row, column) of a player guess (in addition to self). Assume the guess has already been validated.

1. Check if the guess has hit any ship.
2. If there is a hit, update the ship's hit count and update the board at the location of the guess to the `"H"` character (since this was a hit, the location was previously an `"S"`). Finally, print out `"Hit!"`.
3. If the guess is not a hit, then update the location of the board with an `"M"` character and print out `"Miss!"`.

`def __str__() -> str`

Define the `__str__` method to return the current board as a string. Specifically, each line in the returned string will be one row in the current board. For each element in the row, place the character between `"[]"`. For example, a 5×5 board with 5 Ships (the longest has 3 hits) should look like this:

```
>>> print(board_instance)
[H][H][H][S][S]
[S][S][S][S][ ]
[S][S][S][ ][ ]
[S][S][S][ ][ ]
[S][S][ ][ ][ ]
```

3.3 Player Class

The `Player` class represents a single player of battleship. The class should be implemented in the `game.py` file. The `Player` class will be responsible for storing and updating each player's data, placing ships during Part One of the game, and for reading their guesses during each turn of the game (Part Two). You must implement the following attributes and methods for the `Player` class:

`def __init__()`

This method creates a `Player` object. You must provide a name, a board object, and a `ship_list` in this order as parameters. You must define the following attributes (remember to use `self`):

- `name: str` — Represents the name of the player (e.g. `"Player 1"` or `"Player 2"`).
- `board: Board` — Represents the `Board` object belonging to the current player.
- `guesses: list[tuple(int,int)]` — Represents a list of the current player's guesses. Initialize this to an empty list.
- `ship_list: list[int]` — Represents a list of integers representing the types of ships for the current game. For example, if `ship_list=[4,4,3,2]`, then the player will have two ships of length 4, one ship of length 3 and one ship of length 2. (Remember, this list is provided as input and will be used later).

`def validate_guess() -> None`

This method takes a `guess` tuple and checks if it is valid. A valid guess is one that has **not** already been guessed by the Player, and is an existing location on the board. If the guess is invalid, raise one of the following exceptions depending on the reason it is invalid:

```
raise RuntimeError("This guess has already been made!")
raise RuntimeError("Guess is not a valid location!")
```

`def get_player_guess() -> str`

This method will read a guess from the user using the following string: `"Enter your guess: "`. The guess will be in the format `"row, col"` (ex: `"3, 2"`). After obtaining the guess from the user, this method should check if the guess is valid (which method checks that a guess is valid?). If the guess is invalid, re-prompt the user until they have entered a valid guess. If the guess is valid, return the guess as a tuple (hint: the returned tuple will have two `int` values).

`def set_all_ships() -> None`

This function will place all ships for the player. For each ship size in `ship_list`, you must:

1. Get ship coordinates from the user.
2. Get ship orientation from the user.
3. You can assume user input will always be the correct format. However, it may not be possible to place a ship at the provided location with the provided orientation.
4. Initialize a ship and check if it can be placed on the board (hint: what function from the `Board` class can be used to check if a ship is valid?). If the ship is valid, add it to the players board. Otherwise, re-prompt the user until all ships have been placed. User the following strings to read user input:

   ```
   "Enter the coordinates of the ship of size {}: "
   "Enter the orientation of the ship of size {}: "
   ```

3.4 BattleshipGame Class

The `BattleshipGame` class is responsible for running the game, keeping track of turns, and checking if a player has won. The class should be implemented in the `game.py` file. You must implement the following attributes and methods for the `BattleshipGame` class:

`def __init__()`

This method creates a `BattleshipGame` object. The `BattleshipGame` class is initialized with two players. You must define the following attributes (remember to use `self`):

```
player1: Player
player2: Player
```

`def check_game_over() -> str`

This method checks if the game has ended (i.e. all ships belonging to a player have been sunk). If the game is over, return the name of the winning player. Otherwise, return `""`. (remember to use `self`)

`def display() -> None`

This method displays the current state of the game. A sample output will look like this before any ship has been placed (remember to use `self`):

```
Player 1's board:
[ ][ ][ ]
[ ][ ][ ]
[ ][ ][ ]

Player 2's board:
[ ][ ][ ]
[ ][ ][ ]
[ ][ ][ ]
```

`def play() -> None`

This method will run the entire game until one of the players has won (similar to a main function in previous projects). Specifically, this method will do the following (remember to use `self`):

1. (Part One) Each player will place their ships on the board. Each player will have their own board.
2. (Part Two) Then, players will start trying to sink each other's ships. The game should run until someone has won or chosen to exit the game:
   (a) Before any of the players take a turn, display the current state of the board.
   (b) Before each player's individual turn, print out `"{}'s turn."` with the corresponding player's name.
   (c) Get a guess for a player, update the appropriate variables, and check if the game is over. If the game is over, print out `"{} wins!"` with the corresponding player's name.
   (d) After each player has made a guess, ask the players if they want to continue playing (only ask this once per turn, after both players have guessed and the game has not ended). Use the following string to ask: `"Continue playing?: "`. If they enter `"q"`, exit the game, otherwise keep playing.

3.5 proj08.py

```
def main() -> None
```

The main function should be implemented in the `proj08.py` file. The nice thing about using classes is that most of the game logic should now be contained within a specific class, which simplifies the `main()` method. The main method should initialize each player's `Board`, each `Player` object, and then the `BattleshipGame` object. Finally, run the game.

4 Tests

There are 3 provided tests for individual classes:

`assert_board_class.py`
`assert_player_class.py`
`assert_ship_class.py`

Finally, the `assert_all.py` test will test the entire program. Running this file will ask for an input test number n (1 or 2). After executing, this test will generate an output file called `output{n}.txt`. You can compare this to `teacher_output{n}.txt` to confirm your output is correct. A helpful website for making this comparison is diffchecker.com.

5 Grading Rubric

General Requirements:
_ (4 pts) Coding Standard 1-9

Implementation:
_ (4 pts) `Ship` Class
_ (5 pts) `Board` Class
_ (4 pts) `Player` Class
_ (4 pts) `BattleshipGame` Class (no automatic test)
_ (13 pts) all hidden Class tests (same distribution as visible tests for each Ship, Player, and Board Classes)

_ (3 pts) Pass Test1
_ (3 pts) Pass Test2

If you hard code an answer, you will receive a zero for the whole project.