Your program will do things such as processesing text files containing `Ship` or attack strategy information, and you will process that data to build a 10X10 `GameBoard` of ships and perform attacks on the opponent's `GameBoard`. The goal of this project is to allow you to familiarize yourself with Ruby's built-in data structures, code blocks, and text processing capabilities.

- controllers
  - `game_controller.rb`: is provided to you. Don't make any changes to it. (Game Logic)
  - `input_controller.rb`: You will write your input processing code here. (Process Files)
- models
  - `game_board.rb`: You will implement the `GameBoard` class here.
  - `position.rb`: is provided to you. Don't make any changes to it.
  - `ship.rb`: is provided to you. Don't make any changes to it.
- `main.rb`: is provided to you. Don't make any changes to it.

You can run the game (If no attack strategy file provided, it will be randomly generated)

```
ruby src/main.rb test/public/inputs/player1.txt test/public/inputs/player2.txt
```

Or

```
ruby src/main.rb test/public/inputs/player1.txt test/public/inputs/player2.txt
test/public/inputs/attack1.txt test/public/inputs/attack2.txt
```

The format is as follows

```
ruby src/main.rb <first player ships information file> <second player ships
information file> <first player attack strategy file> <second player attack
strategy file>
```

Game Logic

We have provided you with the game logic. We call `read_ships_file` to create a `GameBoard` for each player; then, we call `read_attacks_file` to get an Array of `Position` objects which represent the attack strategy of each player. If no attack strategy file is provided, the game controller generates 35 random attack positions and uses them instead. Each player's `GameBoard`'s `attack_pos` method is called alternately with the other player's attacks (i.e., the `Position` objects), one player's attack, then the other's. The game ends on one of these two cases:

1. All of the ships of a player are sunk (as determined by the `GameBoard`'s `all_sunk?` method)
    - The winner is that player's opponent.
2. One of the players runs out of attack moves
    - Every time its the second player's turn, the game controller checks both the attack strategy Arrays to see if there is any valid attack position left. If there is nothing left for one of the two players, the game doesn't go to the next round of attacks. We do the following to determine the winner.
    - Each `GameBoard`'s `num_successful_attacks` method is called to compare success of each player. The player that has more successfull attacks will win the game.

You will see an output like the following:

```
....
....

Game result:
P1 success: 15,  P2 success: 7
All player 2 ships sunk!

-  `     ´ - -  `     ´ -
  ´     `      ´     `
Yoo-hoooo!
Player 1 has won the game!
-  `     ´ - -  `     ´ -
  ´     `      ´     `

====================
Player-1 GameBoard
STRING METHOD IS NOT IMPLEMENTED
====================
Player-2 GameBoard
STRING METHOD IS NOT IMPLEMENTED
```

Notice that the `to_s` implementation is optional, but it's highly recommended. It helps with debugging your code. Here is an example with a `to_s` implemented

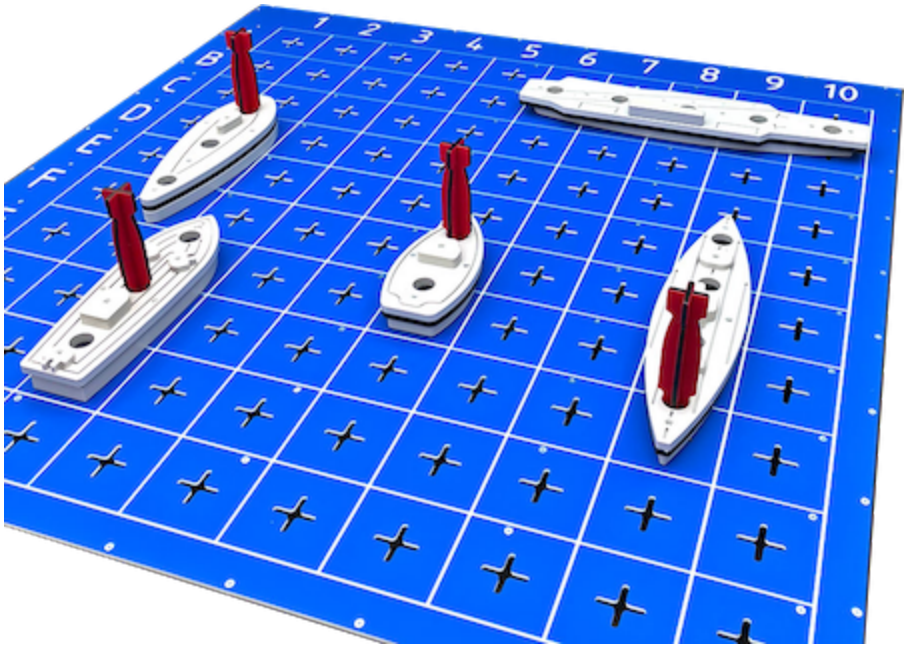
```
Game result:
P1 success: 15,  P2 success: 7
All player 2 ships sunk!

-  `     ´ - -  `     ´ -
  ´     `      ´     `
Yoo-hoooo!
Player 1 has won the game!
-  `     ´ - -  `     ´ -
  ´     `      ´     `
```

```
====================
Player-1 GameBoard
    1      2      3      4      5      6      7      8      9      10
1: -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, -
2: -, - | B, A | B, A | B, - | -, - | -, - | -, - | -, - | -, - | -, -
3: -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, -
4: -, - | B, - | B, - | -, - | -, A | -, - | -, - | -, - | -, - | -, -
5: -, - | B, - | B, - | B, - | B, A | -, - | -, - | -, - | -, - | -, -
6: -, - | B, - | B, - | B, - | B, A | B, - | -, A | -, A | -, - | -, -
7: -, - | B, A | B, A | B, - | B, A | B, - | -, - | -, - | -, - | -, -
8: -, - | -, A | -, A | -, - | -, A | -, - | -, - | -, - | -, - | -, -
9: -, - | -, - | -, - | -, - | -, A | -, - | -, - | -, - | -, - | -, -
10: -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, -
====================
Player-2 GameBoard
    1      2      3      4      5      6      7      8      9      10
1: -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, -
2: -, - | B, A | B, A | -, - | -, - | -, - | -, - | -, - | -, - | -, -
3: -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, -
4: -, - | -, - | -, - | -, - | B, A | -, - | -, - | -, - | -, - | -, -
5: -, - | -, - | -, - | -, - | B, A | -, - | -, - | -, - | -, - | -, -
6: -, - | -, - | -, - | -, - | B, A | -, - | B, A | B, A | B, A | -, -
7: -, - | B, A | B, A | -, - | B, A | -, - | -, - | -, - | -, - | -, -
8: -, - | B, A | B, A | -, - | B, A | -, - | -, - | -, - | -, - | -, -
9: -, - | -, - | -, - | -, - | B, A | -, - | -, - | -, - | -, - | -, -
10: -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, -
```

In this example, `B` represents the ship, and `A` represents the attack made. If there is nothing, `-` is shown.

Part 1

In this part, you will complete the `GameBoard` class located in `game_board.rb`. Please keep in mind that `Position` coordinates start from `1` in our tests (not `0`). Also, although the Battleship game require a 10X10 `GameBoard`, your code should work with any board size.



`initialize(max_row, max_column)`

- Description: `GameBoard` initializer.
- Please don't rename the instance variables as the game controller relies on this.
- For the Battleship game, pass 10 for both row and column since that's how the game works. Your code in this calss should work with any baord size.
- `max_row` and `max_column` define your `GameBoard` boundary. Use them accordingly.

`add_ship(ship)`

- Description: Add a `Ship` to your GameBoard. Return whether it was successful or not.
- Type: `(Ship) -> Boolean`
  Note that types of the parameters in the `Ship` constructor are
  - `start_position` : `Position`
  - `orientation` : `Orientation` where an `Orientation` is one of following `String`s
    - `"Up"`
    - `"Down"`
    - `"Left"`
    - `"Right"`
  - `size` : `Integer`
- First check to see if you can add `Ship` to GameBoard. If you can't add it to the `GameBoard`, return `false`.

- Check for the following rules
    - Ships can not overlap on the gameboard.
    - Ship must be inside the gameboard according to the starting_position and size
- If you can add it, add it to your `GameBoard` and return `true`

`attack_pos(position)`

- Description: Perform attack on the provided position. Return whether the attack was successful or not.
- Type: `(Position) -> Boolean`
- Check if `Position` is inside the boundary. If it's not, return `nil`.
- Update your `GameBoard` accordingly, to note whether the attack was successful. Return whether the attack was successful or not: If it hit a `Ship` in the GameBoard, return `true`. Return `false` otherwise. (Note: Attacking the same `Position` is allowed. Follow the same rules for it.)

`num_successful_attacks`

- Description: Return the number of succesfull attacks made by prior calls to `attack_pos`.
- Type: `() -> Int`
- Return the number of unique parts of ships that have been hit by attacks. In the example picture above, it'd be 4.

`all_sunk?`

- Description: Return a boolean, whether all the ships sunk or not
- Type: `() -> Boolean`
- If all the ships are sunk (i.e., every part of every ship has been successfully attacked), return `true`. Return `false` otherwise.

`to_s` *(optional)*

- Description: String representation of `GameBoard`
- Type: `() -> String`

Implementing the `to_s` is optional but *highly recommended*. Here is a string example:

```
    1      2      3      4      5      6      7      8      9      10
1: -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, -
2: -, - | B, A | B, A | -, - | -, - | -, - | -, - | -, - | -, - | -, -
3: -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, -
```

```
4: -, - | -, - | -, - | -, - | B, A | -, - | -, - | -, - | -, - | -, -
5: -, - | -, - | -, - | -, - | B, A | -, - | -, - | -, - | -, - | -, -
6: -, - | -, - | -, - | -, - | B, A | -, - | B, A | B, A | B, A | -, -
7: -, - | B, A | B, A | -, - | B, A | -, - | -, - | -, - | -, - | -, -
8: -, - | B, A | B, A | -, - | B, A | -, - | -, - | -, - | -, - | -, -
9: -, - | -, - | -, - | -, - | B, A | -, - | -, - | -, - | -, - | -, -
10: -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, - | -, -
```

-
    - In this example, `B` represents the ship, and `A` represents the attack made. If there is nothing, `-` is shown.

Part 2

In this part, you will work on the `input_controller.rb`, and you will complete `read_ships_file` and `read_attacks_file` functions. First, let's understand the input file formats. Again, remember the starting position in a board is (1,1), and it's located in the upper left of the grid.

Input File Formats

There are two types of file formats

- Ships Information
- Attack Strategy

Ships Information File Format

Example:

```
(2,2), Right, 2
(8,5), Up, 3
(6,9), Left, 3
(7,2), Down, 2
(7,6), Down, 2
```

Each line describes a ship: its starting position, orientation, and the size of the ship.

The format is as follows

```
[Starting position], [Orientation], [Size]
```

Each part is seperated by a comma followed by a single space.

- Starting Position
  - (#,#)
  - `Position` pair should be from 1 to `max_row` and 1 to `max_column`. For this part of the project, you may assume ithe 10X10 board size.
  - Notice there is no space after or before the comma, and the number can be more than one digit.
- Orientation
  - Either
    - `Up`
    - `Down`
    - `Left`
    - `Right`
- Size
  - An integer between 1 to 5

Attack Strategy file

```
(2,2)
(2,3)
(4,8)
(6,7)
(7,9)
(1,3)
(9,3)
(4,3)
(10,9)
(9,9)
```

Each line includes a single position that will be attacked.

Format is as follows

```
[Position]
[Position]
....
```

There is exactly one `Position` per line, and it follows the same format mentioned above. For this part of the project, assume that `max_row` and `max_col` is 10.

Input Controller

For reading the file, use the `read_file_lines` function provided. `read_file_lines` accepts a file path and a code block, and returns a boolean indicating if the file exists or not. The code block you provide will be called by the function for each line it reads from the file. Your code block will do the work of making sense of that line.

`read_ships_file(path)`

- Description: Returns a populated `GameBoard` object, or returns `nil` on errors
- Type: `(String) -> GameBoard`
- Use `read_file_lines` function to read the file. If a line doesn't follow the prescribed format, skip it. Create a `GameBoard` of size 10X10, and populate it using `add_ship` function and return the `GameBoard` object.
- Each player must have 5 ships (neither more nor fewer). If there weren't 5 valid ships to be added, return `nil`. Only add the *first* 5 valid ships, and ignore the rest of the file.

`read_attacks_file(path)`

- Description: Returns an `Array` of `Position` objects, or returns `nil` on errors.
- Type: `(String) -> Array`
- Use `read_file_lines` function to read the file. Add the `Position`s to an array. If a line doesn't follow the format provided above, skip it. The function returns `nil` if the file doesn't exist. Please note that attacking the same position is allowed, it's just a poor attack strategy.
- Don’t do boundary validation here; invalid `Position`s will be dropped when actually playing the game based on your `attack_pos` implementation.