Battleship Game
MSc AI Coursework

Oana Cocarascu & Luca Grillotti

Problem description

Battleship is a strategy type guessing game for two players. It is played on 10x10 square ruled grids on which each player's ships are marked. Players alternate turns to hit the other player's ships, the objective of the game being to destroy the opposing player's fleet.

In this coursework we look at a variation of the Battleship game. Each ship occupies a number of consecutive squares on the grid, arranged either vertically or horizontally. Ships cannot overlap (i.e. only one ship can occupy any given square in the grid). There are 5 types of ships that can span over: 1 cell, 2 cells, 3 cells, 4 cells, and 5 cells.

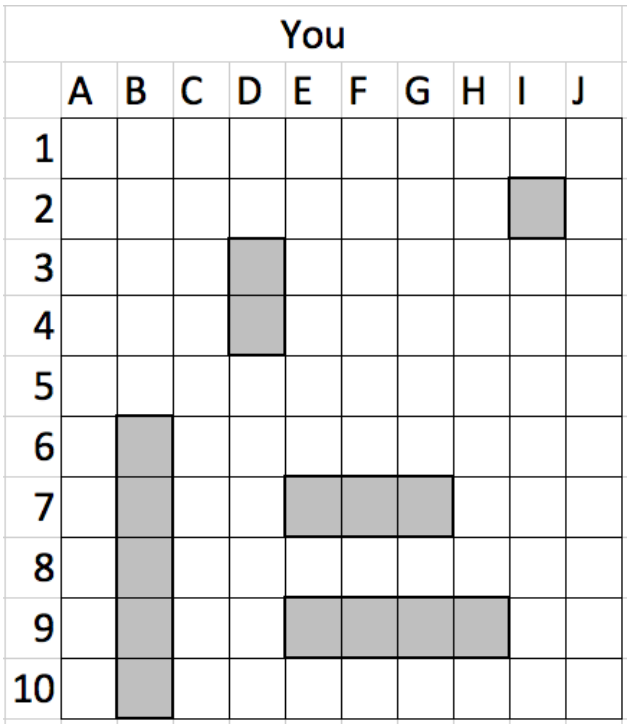


Figure 1: An example of a player's initial board. The opponent will have to guess the positions of the 5 ships on the board.

There are 2 grids per player: one on which the player arranges his ships and records the shots by the opponent and one on which the player records his own shots.

Players take rounds as follows:

- a player announces a target cell in the opponent's grid which is to be shot at
- the opponent announces whether or not the cell is occupied by a ship
  - the attacking player marks the hit/miss on his own 'tracking' grid
  - if it is occupied by a ship (i.e. it is a 'hit'), the player who is hit marks this on his own grid with an **X**
  - the attacking player can attempt another target until he misses
- when all cells of a ship have been hit, the ship's owner announces 'sink'
- if all of a player's ships have been sunk, the game is over and the opponent wins

Figure 2 shows an example of your board and the initial board associated to the opponent.

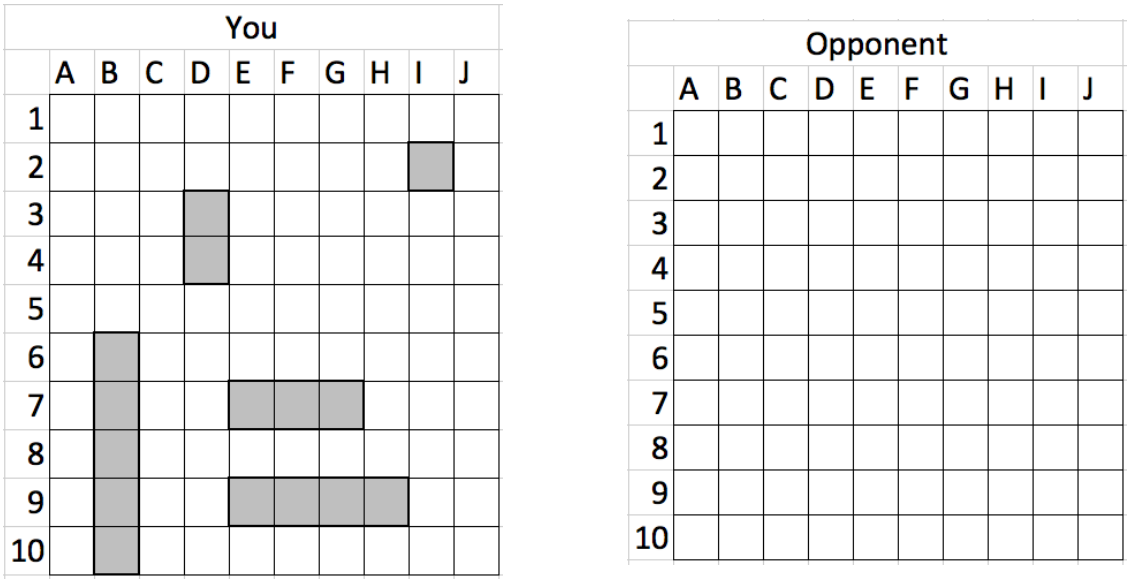
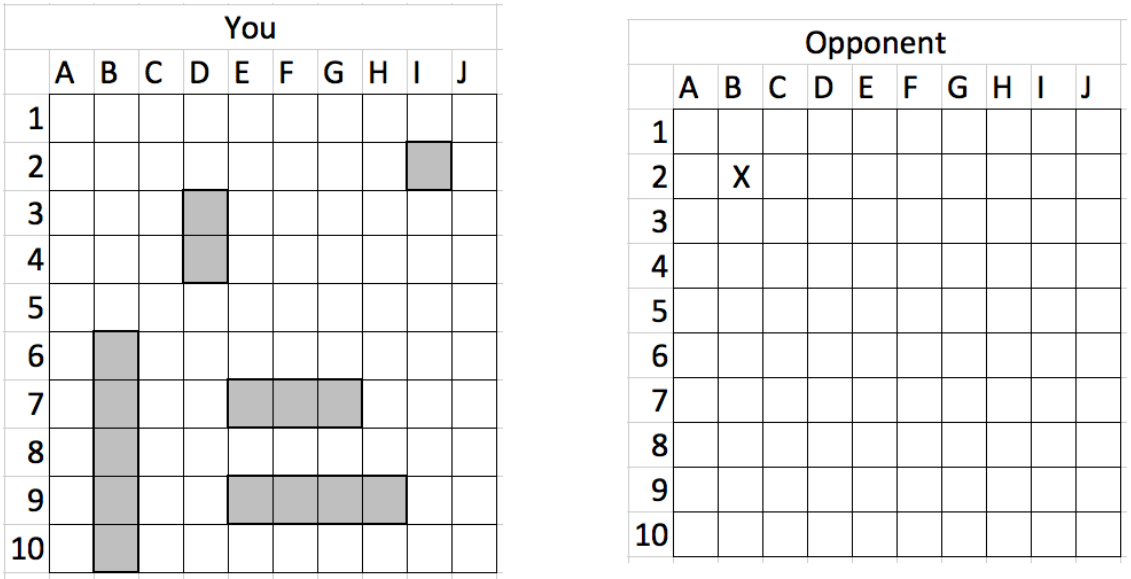


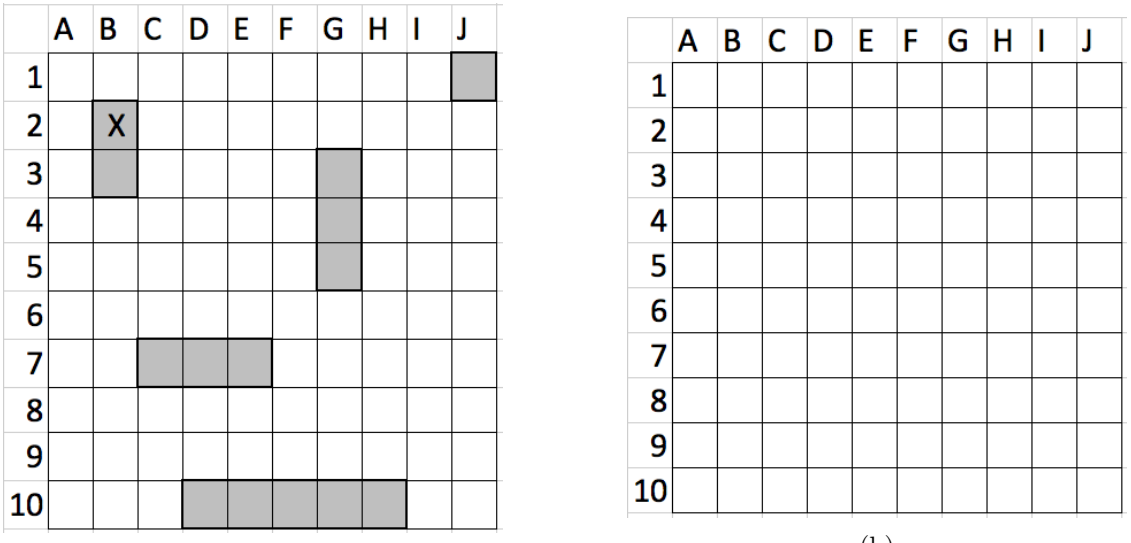
Figure 2: The board on the left contains your ships and where you record the opponent's shots. The board on the right is the one on which you record your own shots (i.e. guess the opponent's ships).

To start the game, you pick a target cell. For example, say you picked cell **B2**. You record the shot at B2 on the board associated to the opponent (Figure 3b). The opponent says the cell is occupied by a ship and marks the shot on his own board (Figure 4a). As you have hit a cell occupied by a ship, you can have another go.



(a) (b)

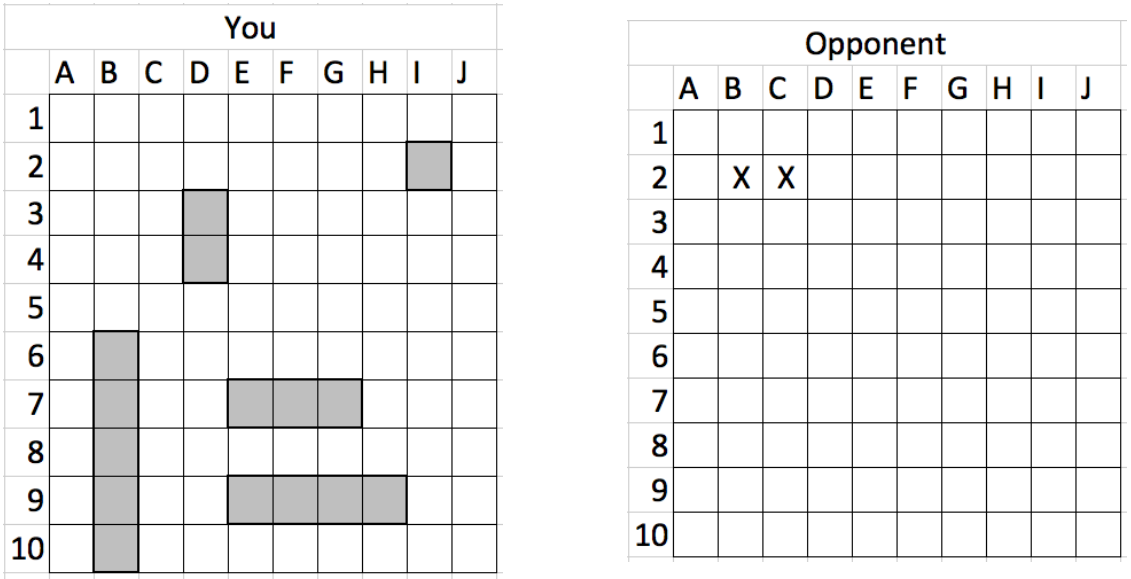
Figure 3: Your boards after the first round.



(a) (b)

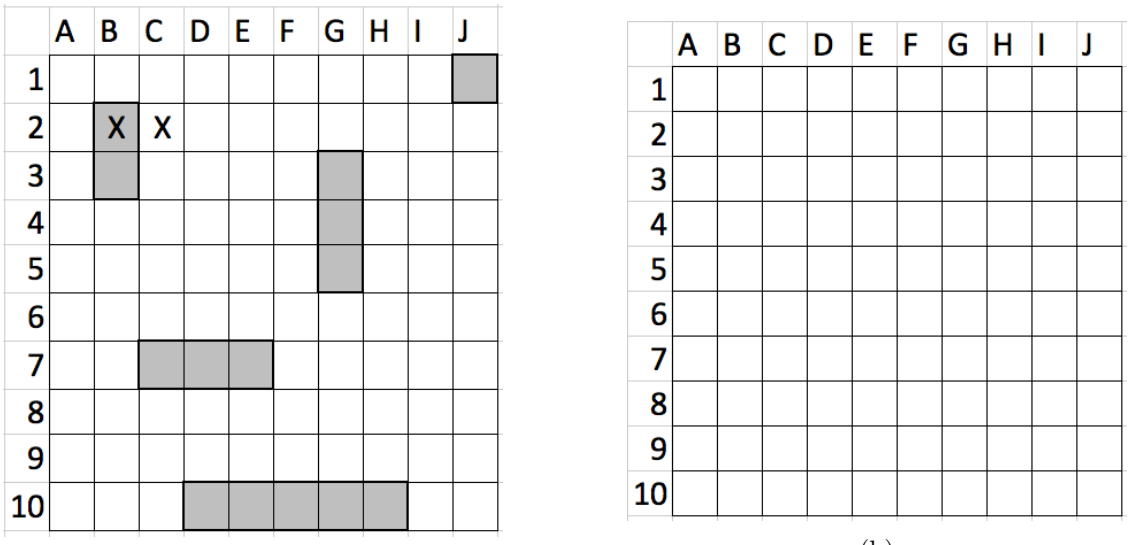
Figure 4: The opponent's boards after the first round.

Next, say you picked cell **C2**. You record the shot at C2 on the board associated to the opponent (Figure 5b). The opponent says the cell is not occupied by a ship and marks the shot on his own board (Figure 6a). Now it is the opponent's turn.



(a) (b)

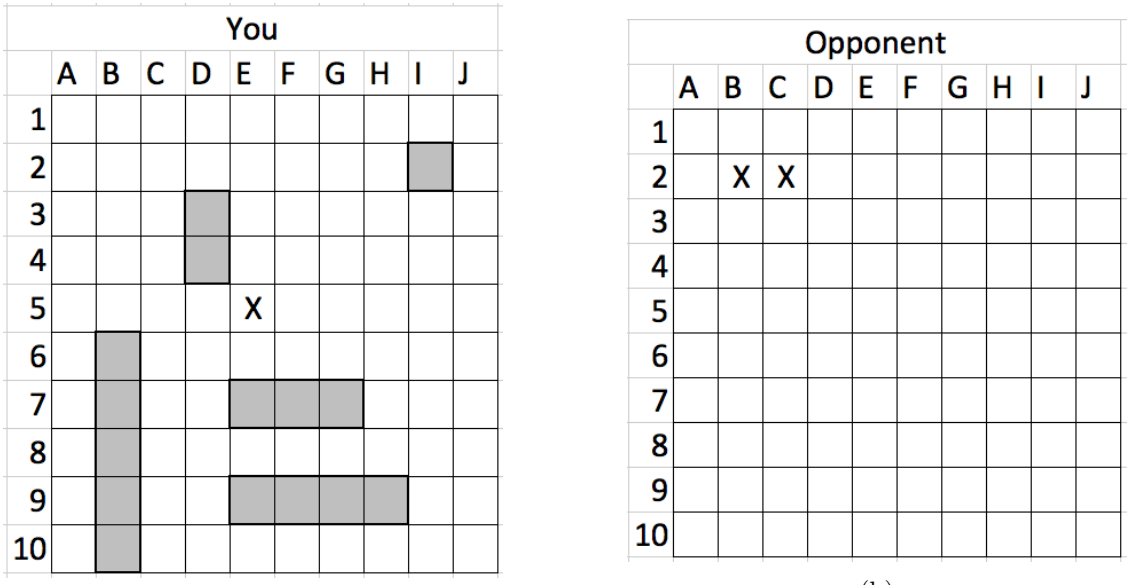
Figure 5: Your boards after the second round.



(a) (b)

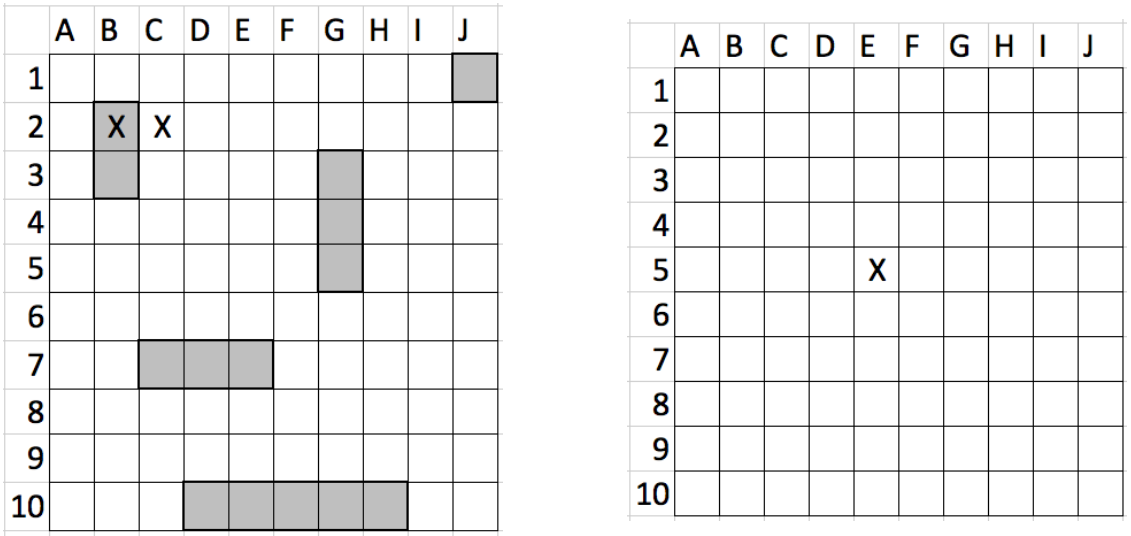
Figure 6: The opponent's boards after the second round.

The opponent hits at **E5**. You record the shot at E5 on your board (Figure 7a) which is not occupied by a ship and the opponent marks the shot on the board associated to you (Figure 8b). The game continues.



(a) (b)

Figure 7: Your boards after the third round.



(a) (b)

Figure 8: The opponent's boards after the third round.

Your task is to implement several functions for the Battleship game. To aid you in developing your program, you are supplied with a skeleton code, representing the classes needed for the game. We used type hinting so that you can better understand the type of object we expect from the functions. For example, the following definition of the function:
`def f(variable: str) -> int:`
states that the expected type of the `variable` argument is `str` and that the expected return type is `int`.

Specific tasks

You are provided with several files defining the classes: `Ship`, `Board`, `Player`, and `Game`, as well as some sub-classes. You will edit the `Ship`, `Board`, and `Player` classes, and some of the sub-classes.

Important:

- For the (sub-)classes that you will edit, do not change the function definitions nor the constructors of those classes.
- Do not modify the `Game` class as it contains the logic that allows you to play and visualise the game (and implicitly for you to analyse the output).
- Also do not modify class `PlayerUser`, a sub-class of `Player` that takes the coordinates to attack as input from the keyboard.
- We provide several functions to aid you in solving this exercise. It is important that you do not modify these functions.
  - `convert.py` contains the logic that converts your input representing the cell you are attacking (e.g. `E5`) to the actual coordinates of that cell on the board.
  - `board.py` contains several functions that display the board to the console.
- main.py contains functions from `examples_scripts.py` that define several types of players and boards to play the game. In order to play the game, you should run the `main.py` file.
- The files for ship, player, and board contain some print statements to help you test your code. For example, to run the code from file `ship.py`, type the following from the main directory: `python3 -m battleship.ship`. You will only be able to run these files after you implement `is_vertical` and `is_horizontal` functions.

Class `Ship` creates a ship given its start and end coordinates. There are two parameters in the constructor, `coord_start`, a tuple of 2 positive integers representing the starting position of the Ship and `coord_end`, a tuple of 2 positive integers representing the ending position of the Ship. Also in the constructor, we initialise the damage coordinates to an empty set and we determine all the cells the ship occupies using the function `get_all_coordinates`. Implement the following functions in class `Ship`:

- `is_vertical()` which, given the x coordinates returns `True` if the direction of the ship is vertical **(5 marks)**
- `is_horizontal()` which, given the y coordinates returns `True` if the direction of the ship is horizontal **(5 marks)**
- `length()` which returns the number of cells the ship occupies **(5 marks)**
- `is_on_coordinate(coord_x, coord_y)` which returns `True` if `(coord_x, coord_y)` is one of the coordinates of the ship **(5 marks)**
- `gets_damage_at(coord_damage_x, coord_damage_y)` which adds `coord_damage_x` and `coord_damage_y` coordinates to `set_coordinates_damages` if the ship contains these coordinates (hint: you can use the function `is_on_coordinate()`) **(5 marks)**
- `is_damaged_at(coord_x, coord_y)` which returns `True` if the ship is damaged at those coordinates **(5 marks)**
- `number_damages()` which returns the total number of coordinates at which the ship is damaged **(5 marks)**
- `has_sunk()` which returns `True` if ship is damaged at all its positions **(5 marks)** T
- `get_all_coordinates()` which returns a set of tuples where each tuple represents a cell (x, y coordinates) occupied by the ship **(5 marks)**
- `is_near_ship(other_ship)` which returns ~~`False`~~ True (typo) if there is a coordinate of `other_ship` (an object of class Ship) that is near the current ship (hint: use the helper function `is_near_coordinate(coord_x, coord_y)` which is provided to you) **(5 marks)**

Do these need be in order? i.e. can i use get_all_coordinates to write is_on_coordinate()?

Class `Board` creates the board of the player and acts as an interface between the player and its ships. The size (10x10) and the types and number of ships are already given to you. The constructor takes a list of ships and keeps track of the set of coordinates of previous shots, represented as a set. There are several checks in the constructor in order to make sure the required number of ships are initially generated on the board, with several functions being called and left for you to implement. Implement the following functions in class `Board`:

- `lengths_of_ships_correct()` which returns `True` if the board has the right number of ships of each length **(6 marks)** T
- `has_no_ships_left()` which returns `True` if all ships on the board have sunk **(5 marks)** T
- `is_attacked_at(coord_x, coord_y)` which returns a tuple of bool variables where the first variable is `True` if the coordinates are part of the opponent's ship and the second variable is `True` if that attack made the ship sink **(6 marks)** T
- `are_some_ships_too_close_from_each_other` which returns `True` if there are at least 2 ships on the board that are near each other **(6 marks)** T

Ask TA about difference between using for loop with counter and list comprehension in terms of memory, and speed (e.g. board.py, line 150)

Class `Player` creates a player. The player takes a board as a parameter and chooses where to perform an attack. Hint: Some of the functionality of this class can be delegated to the `Board`. Implement the following functions in class `Player`:

- `is_attacked_at(coord_x, coord_y)` which returns a tuple of bool variables where the first variable is `True` if the coordinates are part of the opponent's ship and the second variable is `True` if that attack made the ship sink **(5 marks)**
- `has_lost()` which returns `True` if all the ships on the board have sunk **(5 marks)**

You can now run `main.py` and play a game between two `PlayerUser`s. Check the Appendix to see what the output would look like.

Then, you can implement the final two functions.

`BoardAutomatic` is a sub-class of `Board` which randomly generates the ships on the board. Implement the following function from class `BoardAutomatic`, `generate_ships_automatically()` which generates ships and returns the ships in a list. You can define other attributes and methods apart from the existing ones in order to solve this exercise. You may find it useful to use the `random.randint` and `random.choice` functions from the `random` module. **(10 marks)**.

`PlayerAutomatic` is a sub-class of `Player`. Your task is to implement a simple AI player, defining its strategy by implementing the functions `select_coordinates_to_attack` of this class. Your strategy should be better than random. Document your function explaining the strategy of your player. One mark of the allocated ones for this exercise will be given for defining a function in the file `examples_scripts.py`, function that should play a game between two `PlayerAutomatic`. **(7 marks)**.

How to hand in

You are provided with a git repository. You will use git to push your code changes to the repository. You will find on CATe a short introduction to git version control.

To get the code, type the following command:

```
git clone git@gitlab.doc.ic.ac.uk:lab1920_autumn/cw1_<login>.git
```

replacing `<login>` with your Imperial username.

To submit on CATe, go to `https://teaching.doc.ic.ac.uk/labts` and click on the button `Submit to CATe` that is associated to the commit that contains the code you wish to submit.

How you will be marked

You will be assigned a mark according to:

- whether your program works or not;
- whether you have used meaningful names for variables;
- whether you have used a clear, appropriate and logical design.

Appendix

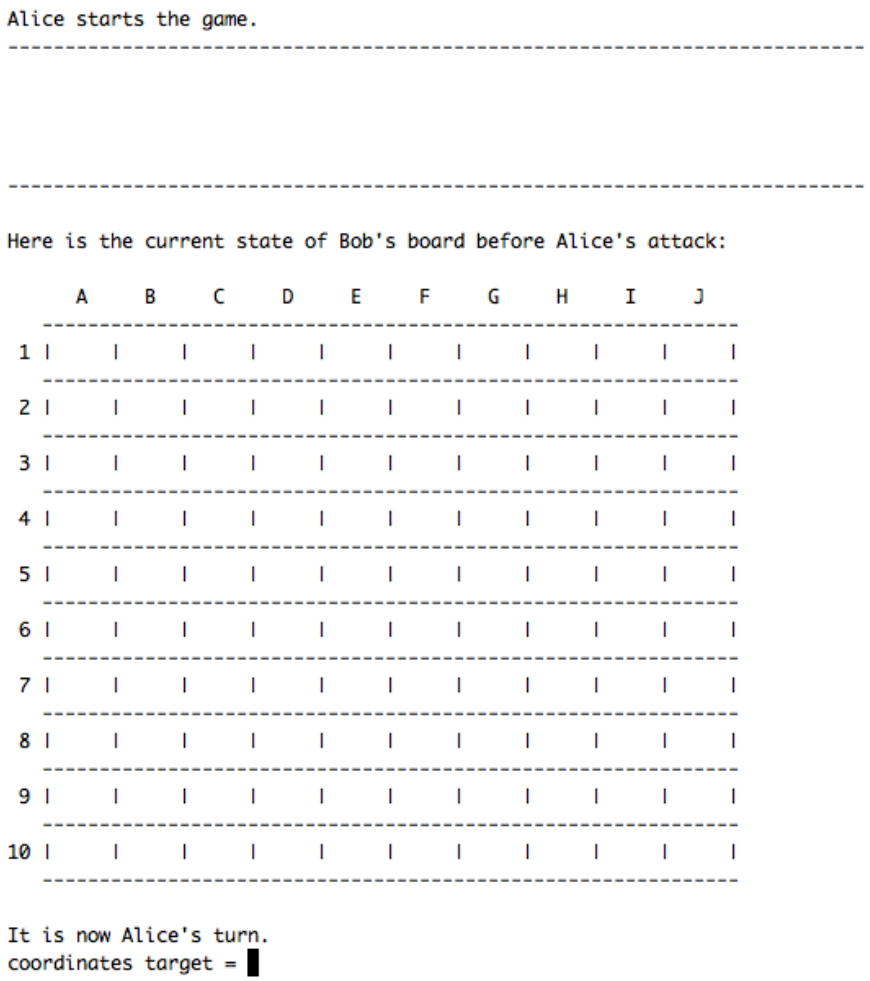


Figure 9: The initial output when starting a game with two `PlayerUser`s. Alice can see an empty board where she will record her shots.

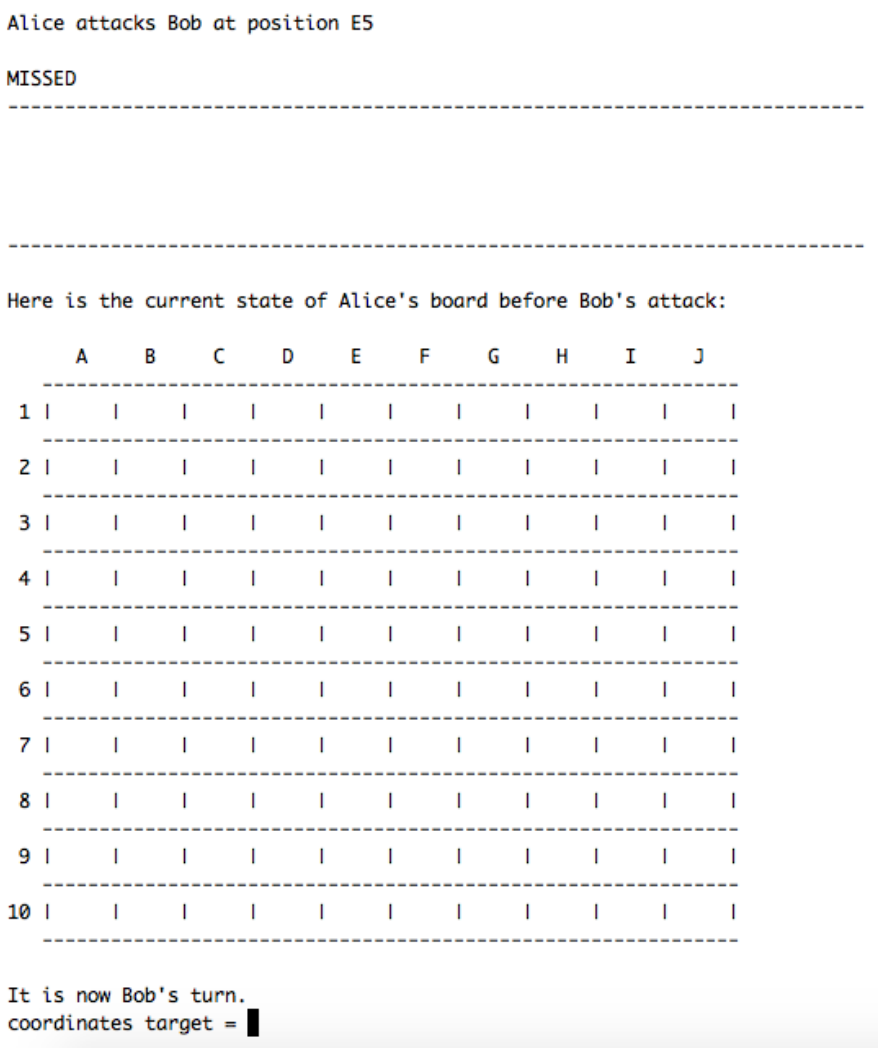


Figure 10: The cell to attack is `E5`. As it missed hitting a ship, Bob can now input the coordinates to attack. You can see an empty board where he will record his shots.

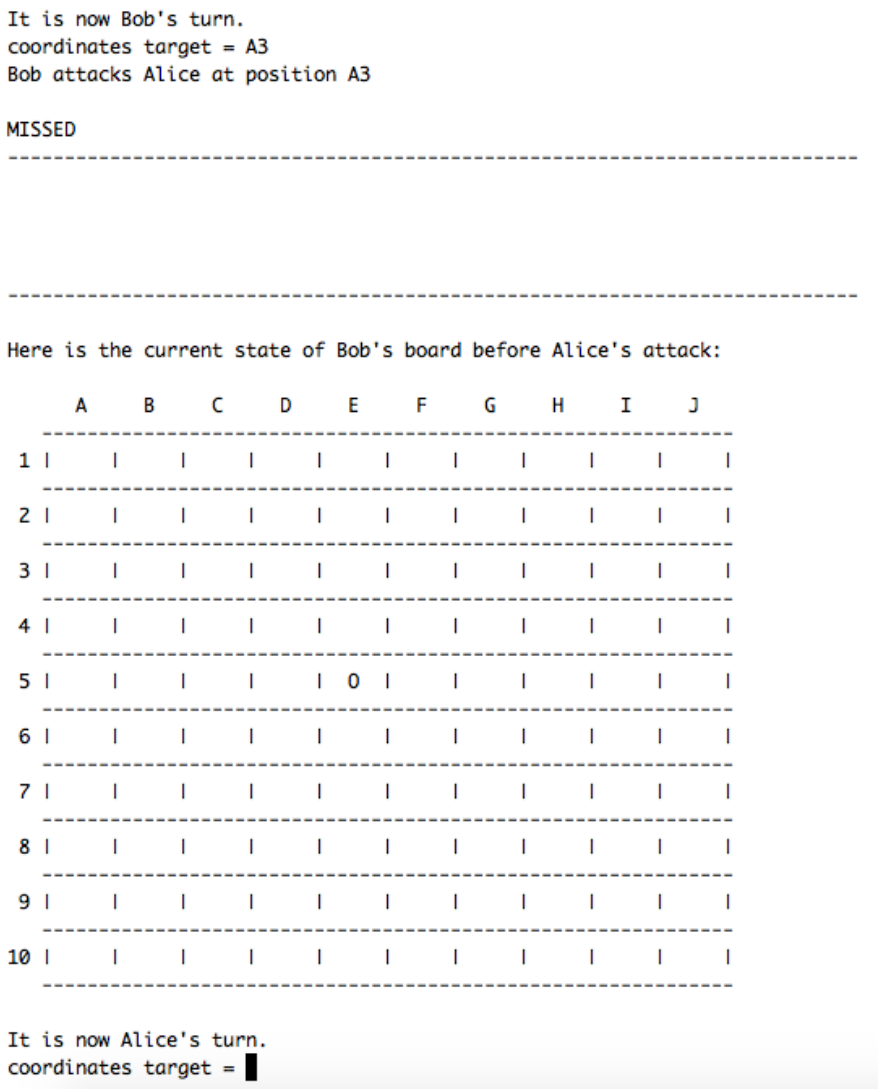


Figure 11: The cell to attack is `A3` and is a miss. Now, it is Alice's turn. You can see on the board the cell she attacked earlier and was a miss (represented with a `O`).

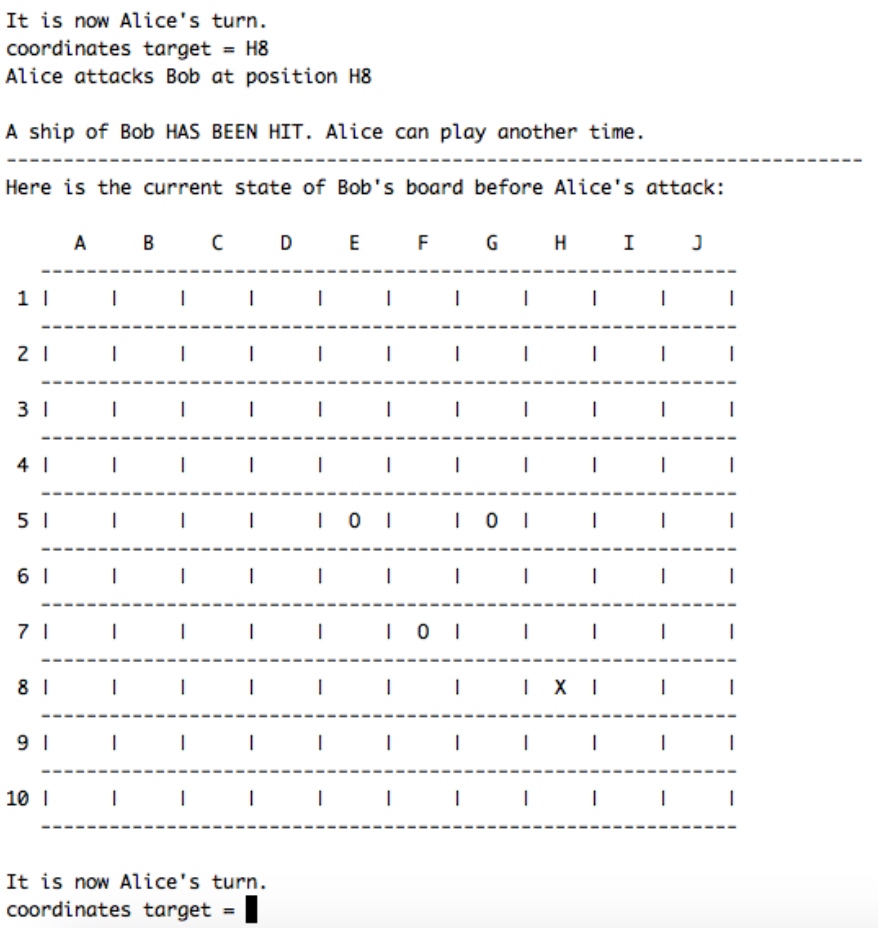


Figure 12: Several rounds later, it is again Alice's turn. She targets `H8` and hits a ship. The board now contains all her previous shots as well as the hit at `H8` represented with an `X`.

```
It is now Alice's turn.
coordinates target = G8
Alice attacks Bob at position G8

A ship of Bob HAS BEEN HIT. Alice can play another time.
------------------------------------------------------------------------------
Here is the current state of Bob's board before Alice's attack:

      A     B     C     D     E     F     G     H     I     J
   ------------------------------------------------------------
 1 |     |     |     |     |     |     |     |     |     |     |
   ------------------------------------------------------------
 2 |     |     |     |     |     |     |     |     |     |     |
   ------------------------------------------------------------
 3 |     |     |     |     |     |     |     |     |     |     |
   ------------------------------------------------------------
 4 |     |     |     |     |     |     |     |     |     |     |
   ------------------------------------------------------------
 5 |     |     |     |     |  O  |     |  O  |     |     |     |
   ------------------------------------------------------------
 6 |     |     |     |     |     |     |     |     |     |     |
   ------------------------------------------------------------
 7 |     |     |     |     |     |  O  |     |     |     |     |
   ------------------------------------------------------------
 8 |     |     |     |     |     |     |  X  |  X  |     |     |
   ------------------------------------------------------------
 9 |     |     |     |     |     |     |     |     |     |     |
   ------------------------------------------------------------
10 |     |     |     |     |     |     |     |     |     |     |
   ------------------------------------------------------------

It is now Alice's turn.
coordinates target =
```

Figure 13: Since ships can only be vertical or horizontal, Alice looks at the cells around `H8` and selects `G8`.

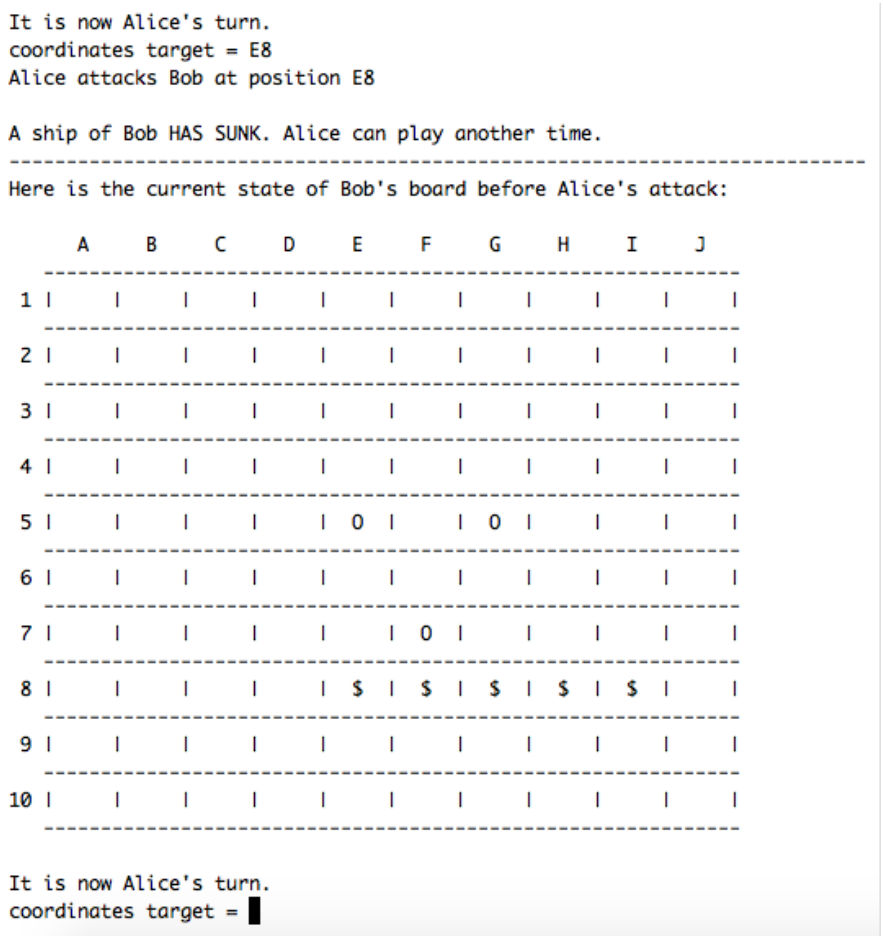


Figure 14: Once the ship sinks, its cells will be represented with $ sign.