COMS W1007 Homework 1

Submission instructions

All programs must compile and run on CUNIX to receive credit. Submit your electronic files via `http://courseworks.columbia.edu`. We prefer electronic submission of written portions, though you will not be penalized for paper submissions. (Do **not** print out your programs.) Include a separate README text file that explains exactly what each file in your submission is. Place all the files you want to submit into a submission directory with the following naming scheme.

```
<your_uni>_hw<number>
```

So if my UNI is `uni1234` am submitting homework 5, my directory would be `uni1234_hw5`. Archive your submission directory using zip, tar, or gzip format. In CUNIX, you can tar and gzip with the following command:

```
tar -czvf uni1234_hw5.tar.gz uni1234_hw5
```

and upload `uni1234_hw5.tar.gz` to courseworks. (You will probably need to first download the file to a local directory using an FTP program. See CUNIX tutorial for more info.)

Multiple Submissions: You can submit multiple times, but we will only consider the latest submission based on the timestamp in courseworks. Please give at least 1-2 minutes between two submissions so we can tell which is the newest submission.

Keep a pristine copy of your submission folder in case there is any submission error.

Programming Assignment

Description

Build a program that plays the board game Battleship! For those who may not have played the game before, the game is a classic two player pen-and-paper game that was converted into a popular board game by Milton Bradley. It is played on a grid, so needs no fancy interface and can be implemented using a text-only console interface.

The rules for Battleship are as follows. Each player keeps a 10x10 grid of squares onto which they each place a set of five ships. The ship pieces are each one square wide and varying lengths, and represent different types of naval vessels, such as aircraft carriers, battleships, and submarines. The standard ship lengths for the five ships are 5, 4, 3, 3, and 2. In the setup phase, the ship pieces must be placed horizontally or vertically on the grid. After the players have placed their ships on their boards, the main gameplay begins. At this point, each player does not know where the other has placed his or her ships.

The players take turns "firing shots" at the other player's board by announcing a grid location. The player being attacked must reveal whether one of his or her ships is at the attacked grid location. Once a ship has been hit on all its squares, it is considered destroyed. Players take turns attacking each other until one player has lost all his or her ships, at which point the other player wins.

To help strategize, each player also maintains a board marking where he or she has attempted shots and whether they hit or missed.

Your program should allow a full game of Battleship following all its standard rules against a computer opponent. This computer opponent should at least attack random grid locations that is has not already attacked, or if you prefer, have a smarter strategy[1]. The program will use a text interface for now. During the setup phase, the interface should allow the player to view the board and select a location to place each ship. At each turn, the program should display the two grids, and prompt the player for an attacking grid position. The program will detect when the game is over, announce the winner, and exit.

Any invalid entries during user input should result in an error message and allow the user to retry. The program should not crash. Ships must only be allowed to be placed in legal positions (not overlapping, must fit on the grid).

One reasonable way to display a grid in text is as follows. Different characters can be placed in the grid to represent hit, miss, and ship squares. You are welcome to come up with a better text-based grid interface if you want.

```
  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
--|---+---+---+---+---+---+---+---+---+---+
0 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
1 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
2 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
5 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
6 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
8 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
9 |   |   |   |   |   |   |   |   |   |   |
--|---+---+---+---+---+---+---+---+---+---+
```

[1]You won't receive extra credit for a smarter AI.

Process and Deliverables

You must submit the results of the following steps. Do these steps in order, but do not hesitate to return to a previous step when you realize you could have done things better as you are working on a later step (this will likely happen). Submit the final version of each part.

1. (10 points) The description above is not exactly a use case. Write use cases for (1) the setup phase, (2) for a turn of the game, and (3) for the end of the game. These use cases should describe the user interface. You must submit these three, but you are welcome to write more, as they will likely help your design.
2. (10 points) Identify classes and their responsibilities and submit CRC text describing each class. If you use index cards for CRC, copy the text into a document so we won't accidentally lose your index cards.
3. (10 points) Walk through placing a couple ships and a couple turns of the game using the classes you've decided on to make sure they make sense. Draw these walkthroughs as sequence diagrams. For this step, submit the sequence diagrams of the final version of your design.
4. (10 points for javadoc) Write up a class skeleton for your program with full javadoc commenting for each class and public method. Make sure the javadoc compiles. This is a good time to double check that your walkthrough makes sense. If any classes or methods seem like they will be too complicated, it might make sense to refactor them before you write too much code.
5. (50 points) Implement your program. I suggest starting with the class with the least dependencies, because it should compile with no errors once you are done. You can submit your final code for steps 4 and 5.
6. (10 points) Run your use cases from step 1 on the finished program. Copy and submit the text of the console interaction.

Resist the temptation to start from step 4 or 5.

Hints

- If you need to store the ships in a set, you can use an ArrayList. However, you may also consider ignoring the identity of the ships altogether once they have been placed, and instead just treat each grid cell as water or ship. Both strategies make sense, depending on how you prefer to represent the game state.
- Try to keep the user interface easily modifiable. For example, in the future, we may want to use the same game engine to display the game graphically. The rules of the game should run mostly independently of the interface. Allowing this, and other similar flexibility will help keep your code organized.
- Don't worry about efficiency for this program. We will be grading you on your design and not on whether you're using the most efficient data structures for storing and updating the game states. Just use the easiest, functional way you can think of to store the state.