Final Project – Game AI

This final project will push your limits in logic skills. To do your best, you will need to combine your knowledge of all the previous chapters. If you don't know the game Battleship, I suggest that you try the online version: http://www.knowledgeadventure.com/games/battleship/. Play it a few times, then figure out a strategy. Note the number of shots you fire in each game; you will want this number to be low for a good mark.

Your task is to create artificial intelligence to create a firing strategy requiring the fewest number of shots to sink all the enemy ships. All the rules have been programmed and you are not allowed to adjust the rules. You are also not allowed to examine the game board except through the given routines. You will be given hints on how to develop your code so that it is easier to debug even if you decide to use random numbers.

It is best to plan out your logic on paper before doing any coding. You should have at least a minimal state machine: one state would be a hunt mode (fire until you get a hit) and another state would be a destroy mode (fire in a pattern until the ship is sunk). You might possibly use arrays to make a list of strategic firing pattern. After coding your strategy, you may decide you need to handle more states to make your strategy more efficient.

BattleShip API

Ship – you are not allowed to use this class.

BattleShip::BattleShip – this is the constructor, but you won't need to use it.

BattleShip::clearBoard – this is to start a new game, but you won't need to use it. It resets counters and places ships randomly on the board.

BattleShip::printBoard – this prints the game board with all the positions fired on. You won't need to use it in your code. It is used in the main function and you are free to comment it in/out depending on your debugging needs.

BattleShip::placeShip – this is a method to place a ship on the board. It returns true if the ship can be legally placed. You may want to use this if you want to turn the game into player vs player or player vs computer.

BattleShip::fireShot – this is the method you must use. You are responsible for using valid coordinates (0,0) to (9,9). Using invalid coordinates results in a missed shot. This method returns a hit/miss status, so check the return value to optimize your strategy:

```
0 if bad shot
1 if missed shot
-1 if hit an unsunk ship
-2 if sunk
-100 if game complete
```

BattleShip::checkHitOrMiss – this is an optional method for you to use.

```
2 if bad shot
```

```
1 if missed shot or already sunk ship
0 if empty spot
-1 if hit
-2 if sunk ship
```

Main – there is not much to do with the main function other than turning debug on/off. Do not modify the logic at all. When you are first starting off, you will want to print the game board after each shot to verify the shots are going where you want them to. Later, you will only need to print out the game board after all the ships sunk to verify your firing logic.

Test you logic using the single game and try out different random seeds. When you are happy with the logic then you can proceed to the multi-game test to calculate your average, best, and worst games.

For most games, the main loop should be very minimal; it gives you an idea of how the game is constructed from a high level. The finer details are all handled by more specialized methods. This main loop won't look much different to handle an AI-vs-AI game, a 2 user game, or an AI-vs-user game.

Steps to Completion

You should follow the steps in the recommended order to successfully complete this project. Change the order at your own risk.

The default AI is not very sophisticated, but it shows you how to use the AI. It simply goes left to right from the top rows to the bottom rows. Your AI should perform better than this. If you have a bug in your AI, it is possible to perform worse than this – it will be because you are shooting off the board or repeating squares. To prevent eliminate duplicate shots on a square, you should call *BattleShip::checkHitOrMiss* and check the return value.

To optimize your algorithm, you may want to call *BattleShip::checkHitOrMiss* and check for -1. You may get a partially hit ship when you are in your destroy state.

To perfect your hunt algorithm, think about how the ships are allowed to be laid out (only horizontal and vertical). Also think about the spacing of shots to optimize your hunt algorithm.

Final Project – Game API

If you have successfully completed the previous project, you can write a Game API for Mastermind. Again, this will push the limits of your logic skills. Counting is not a trivial problem; it is easy to over- or under-count. If you don't know the game Mastermind, I suggest that you try the online version: http://www.puffgames.com/mastermind/. Play it a few times to understand the rules. Then you can use it as a comparison to see if you are implementing the rules correctly.