Battleship!

Programming Assignment 2

Due May 21, 2017

Overview

You will implement a simple text based implementation of Milton Bradley's Battleship! game. In this game, you and the computer will attempt to sink each others fleet of five ships by exchanging salvos.

Game Rules

The following rules apply to the physical game. We will need to slightly adjust them for computer play.

Objective

The object of Battleship is to try and sink all of the other player's before they sink all of your ships. All of the other player's ships are somewhere on their board. You try and hit them by calling out the coordinates of one of the squares on the board. The other player also tries to hit your ships by calling out coordinates. Neither you nor the other player can see the other's board so you must try to guess where they are. Each board in the physical game has two grids: the lower (horizontal) section for the player's ships and the upper part (vertical during play) for recording the player's guesses.

Start of Game

Each player places the 5 ships somewhere on their board. The ships can only be placed vertically or horizontally. Diagonal placement is not allowed. No part of a ship may hang off the edge of the board. Ships may not overlap each other. No ships may be placed on another ship.
Once the guessing begins, the players may not move the ships.
The 5 ships are: Carrier (occupies 5 spaces), Battleship (4), Cruiser (3), Submarine (3), and Destroyer (2).

Playing the Game

Player's take turns guessing by calling out salvo coordinates. The opponent responds with "hit" or "miss" as appropriate. Both players should mark their board with pegs: red for hit, white for miss. For example, if you call out F6 and your opponent does not have any ship located at F6, your opponent would respond with "miss". You record the miss F6 by placing a white peg on the lower part of your board at F6. Your opponent records the miss by placing.

When all of the squares that one your ships occupies have been hit, the ship will be sunk. You should announce ”hit and sunk”. In the physical game, a red peg is placed on the top edge of the vertical board to indicate a sunk ship.
As soon as all of one player’s ships have been sunk, the game ends.

Program Specifics

Your program will take care of all book keeping. Instead of white and red pegs, the text board will use O’s and *’s to represent misses and hits. The map below shows three hits at C7, C8, and D1, and lots of misses (A7, B4, etc.).

```
 0123456789
A       O
B    O
C      O**
D  *
E     O   O
F
G       O
H
I
J
```

When a ship is sunk, the program should announce the ship type after the fatal salvo (”Cruiser hit and sunk.”).
The program should start by asking the player where they want to place their ships. Locations include a letter-number coordinate and direction (U, D, L, R). The placement must be legal (no overlaps or dangling of the board). For example:

```
Where do you wish to place your Carrier?
H7U
Where do you wish to place your Battleship?
B2L
You cant place your Battleship there.
Where do you wish to place your Battleship?
B2R
```

The main interaction loop will consist of a status report, salvo map, and prompt for the next salvo location. The player does not need to see record of the opponents hits and misses, just a damage report for each ship. Once a fleet has been sunk, the game will end.
The computer will go first and fire at random locations.

```
A salvo landed at B3 and hit your Battleship.
Your fleet’s status:
Carrier       0 hits
Battleship    2 hits
Cruiser       1 hit
Submarine     Sunk
```

```
Destroyer     0 hits

4 enemy ships remaining

 0123456789
A       0
B    0
C      0**
D  *
E     0   0
F
G       0
H
I
J

Next salvo location: H3
Firing salvo. Woosh...miss
```

It is expected that you develop this program with an object oriented design. Think objects and messages, not sequences of operations. Please use one file per class. You will be expected to run unit tests.

Unit Tests

For each of your classes, there should be a unit test file. The convention will be to append *_test* to the file's basename (e.g. chair.rb and chair_test.rb). Please see
`https://en.wikibooks.org/wiki/Ruby_Programming/Unit_testing`
for more details.
Consider the class Chair

```
class Chair
  attr_accessor :legs
  def initialize legs=4
    @legs = legs
  end
end
```

Ruby ships with a unit testing framework that makes it easy to write and maintain regression tests. For example:

```
require_relative "chair"
require "test/unit"

class TestChair < Test::Unit::TestCase
  def test_default_constructor
    chair = Chair.new
    assert_equal 4, chair.legs
  end
  def test_constructor
    chair = Chair.new 3
    assert_equal 3, chair.legs
  end
  def test_assignment
    chair = Chair.new
    chair.legs = 3
    assert_equal 3, chairs.legs
  end
end
```

When you run the test program, you get a nice report.

```
$ ruby chair_test.rb
Loaded suite chair_test
Started
...
Finished in 0.000663 seconds.
-------------------------------------------------------------------------------------------
3 tests, 3 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed
-------------------------------------------------------------------------------------------
4524.89 tests/s, 4524.89 assertions/s
```

Now, lets assume we had `@legs = 4` in the constructor. Running the unit test catches this oversight.

```
$ ruby chair_test.rb
Loaded suite chair_test
Started
.F
===========================================================================================
Failure: test_constructor(TestChair)
chair_test.rb:11:in ‘test_constructor’
      8:   end
      9:   def test_constructor
     10:     chair = Chair.new 3
  => 11:     assert_equal 3, chair.legs
     12:   end
     13:   def test_assignment
     14:     chair = Chair.new
<3> expected but was
<4>
===========================================================================================
.
Finished in 0.011035 seconds.
-------------------------------------------------------------------------------------------
3 tests, 3 assertions, 1 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
66.6667% passed
```

```
-----------------------------------------------------------------------------------------
271.86 tests/s, 271.86 assertions/s
```