CS 5004: Object Oriented Design and Analysis

Northeastern University
College of Computer and Information Science

Overview | Course Map | Objectives

We extend, and finalize, our memory model for Java programs, and discuss garbage collection. We use our memory model to introduce Java programs that use mutation and circular data structures. We discuss the effects of mutation and circular data structure to our programs covering stack overflow exceptions and equality between objects.

We introduce class invariants and use pre-conditions and post-conditions along with invariants to reason about Java programs that use mutation.

Milestone 1 due date Thursday, February 15th @ 11:59pm
Milestone 2 due date Thursday, March 1st @ 11:59pm

Submission Criteria

Repository Contents

Your repositories must contain only the following

1. pom.xml file and
2. src folder with appropriate sub-folders with your code and test code only.

Code Criteria

1. Your code for this assignment must be in a package named `edu.neu.ccs.cs5004.assignment5.battleship`.
2. Your project should successfully build using maven generating all the default reports.
3. Your Javadoc generation should complete with no errors or warnings.
4. Your Checkstyle report must have no violations.
5. Your JaCoCo report must indicate 75% code coverage per package for "Branches" **and** "Instructions" or more (unless otherwise is specified).
6. Your FindBugs report must have no violations (except for violations categorized as PERFORMANCE or SECURITY, you can ignore those).
7. Your PMD report should have no violations.

Battleship for console

You and your partner are asked to design a program in Java to implement Battleship game for Console. The program will be played between a single player and a computer. This is an open - design project with **a few design requiemens/constraints outlined at the bottom of this assignment** .

Here is a reminder of the game setting

- The game is played between two players, each has a 2-dimensional map (board), where fleet of ships is placed (this is a fleet map).
- None of the players can see the opponent's fleet map.
- The objective of the game is to sink the entire fleet of the opponent, before the opponent sinks all your ships.
- Each player tries to hit ships of the opponent by guessing the position of one of the cells on the map.

- An opponent will reply if a ship was partially hit or not, or if was completely sunk.
- A player will maintain additional map, which records the correctness of guesses of opponent's fleet location (guesses' map)
- The fleet (per player) consists of 10 ships (1 Battleship, 2 Cruisers, 3 Submarines and 4 Destroyer).
- 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. Ships may NOT be placed on another ship.
- Since this is a **Russian** version, ships **cannot** occupy cells next to each other. In other words, ships cannot touch each other; there must be at least 1 cell that separates between two ships.
- Once the guessing begins, the players may not move the ships.
- If any of players guesses correctly, this player has an additional turn.
- As soon as all of one player's ships have been sunk, the game ends.

Part 1 - Ship placement:

- Each player has one fleet map (FleetMap) that should be populated by the ships.
- At the beginning of the game, both fleet maps are empty (filled in with approriate cells).
- There are two modes to determine the fleet configuration to be placed (your program should support BOTH):
    - DEBUG mode - the fleet configuration (the number and type of ships) is determined by the user.
    - GAME mode - the fleet configuration is determined by the game rules.
- Once setup part begins both players (a human and a computer) start placing ships.
- To place a ship on the fleet map, we need to provide:
    - The top left cell of the ship.
    - The direction (either horizontal or vertical).
- There are two ways to place ships (your program should support and test BOTH):
    - RANDOM placement - the program will randomly choose the top left cell of the ship and the direction (either horizontal or vertical) of the ship.
    - USER placement - the user will provide the input using a console (you define the exact interaction with user, be sure to ensure that it is user friendly and fail safe)
- At the end of the setup we will have 2 fleet maps with the correct fleet configuration :
    - In the DEBUG mode, both fleet maps, should be visible to the user.
    - In the GAME mode, only the human player's fleet map, should be visible to the user.

Part 2 - Playing the Game:

- To support the actual game, you are requested to design a battle map that will store the results of guesses (BattleMap).
    - Each player has one battle map, which is empty at the beginning of the game (populated with appropriate cells)
    - Every turn, the battle map will record the result of the player's attack on the opponent's fleet map.
- There are several strategies to attack an opponent:

    - RANDOM strategy - random choice of the not yet hit cell on the map (useful for computer player).
    - SMART strategy - design your own smart strategy for BONUS points.
    - USER strategy - user provides the cell location to be hit **using input via console**.
- Your program should support and test RANDOM and USER strategies (SMART is a bonus).
- Upon each turn in DEBUG mode, all the updated 4 maps (2 fleet maps and 2 battle maps) should be visible to the user.
- Upon each turn in the GAME mode, ONLY the human player maps are visible (one fleet map and one battle map).
- NOTE:
    - While the fleet position does NOT change during the game, the state of the cells on the fleet map changes (from not yet hit cells to hit or sunk cells). Thus the fleet map state should be kept updated and utilized to provide an automatic answer to the opponent's guesses (that is there is no need in user interaction to determine the outcome of the game).
    - The battle map state should be utilized by the strategy to come up with the next guess.
- Upon completion of the game, the console should display a message: "Game over, the winner is ...".

Design Guidelines and Constraints:

- You should utilize (with minor changes) the design of *cells*, *ships* and a *map* from your previous assignments.
- You should **code to interface**. If you are using existing libraries, use interface types. Whenever possible define an interface for your custom ADTs (for example *Strategy* etc.) and use it as a static type of your objects.
    - Team work: define all the necessary interfaces together and divide them, such that each partner is responsible for the implementation of certain interfaces
- Do NOT abuse *static* methods. Think in terms of nouns in the problem formulation above and transfer them into classes.
- Abstract out implementation of common behaviours and specify those in interfaces.
- Follow General Design rubric to come up with a clean code.
- Provide proper JavaDoc and tests of functionality (i.e., using `Assert`).
- **Use packages to represent modules** in your design (for example: you may have a package for cells' classes and interfaces, etc.).
- To get a full credit, you should **follow Model View Design Pattern**. That means your Console view can be easily subsituted by another view (for example Graphical view). Please review the class notes on MVC. If you successfully separate Controller from your View (and Model), you will gain BONUS points. Please be ready to explain during the codewalk, what classes represent the model and what classes represent the view.
    - Think about what changes should be made in your program in order to accommodate a different view.
    - Hint: ideally there should be an interface that your View implements and you code to this interface. Thus, any new view module would simply need to implement this interface.
- Classes that belong to Model should NOT share the same package with classes that belong to View.

- To get a full credit, you should use **Observer Pattern** (Push model) **at least once** in your program. Please be ready to explain this in the codewalk. Hints:
    - You may alter your existing design, such that a ship will have a list of cells it occupies. Once a ship is sunk, it notifies cells (**via Interface**) about that (and thus these cells become sunk).
    - Model (observable) should update View (observer) upon changes in its state.
- To get an input from user, please use ONLY `BufferedReader`. You can find here an example of how to use `BufferedReader` to read input from console. Please notice that some methods of `BufferedReader` may throw **Checked** exception. You will need to handle these exceptions, as illustrated in the example and explained in the lecture notes. You may also find it useful to read about *try with resources.*
- Note: it is challenging to achieve high test coverage of the code that uses `BufferedReader` (because of the `catch` and `finally` branches) - do your best. We will take this into account when checking your Jacoco reports.
- Using `Scanner` is NOT allowed in this assignment.

Milestones(please see deadlines at the top):

1. Submit UML diagram of your design (including all the interfaces you defined) and code for Part 1 (including tests for functionality).
2. Submit the entire game - Part 1 and Part 2 (tested and in working condition).