HW-1
Battleship (Морской бой)

Introduction

Games are a good source of ideas for software applications and an introduction to the challenges of software engineering. This assignment is based on Battleship game, a well-known game you probably played in childhood. The main goal is to improve your experience with the classes in Java and use of inheritance.

General idea

Battleship is usually a two-player game, where each player has a fleet of ships and an ocean where the ships are placed, but hidden from the other player, and both players compete to sink the other player's fleet first. You need to implement a variation of this game: only the computer player has its Fleet and arranges the ships, while the human player attempts to sink them.

The Ocean is the field of Battleship game. It is a rectangle (matrix) of square cells, which consists of M rows and N columns of cells.

Each ship of the Fleet might be of one of the following ship types, which differ with the number of vertically or horizontally adjacent Ocean cells in a line they occupy:

- Carrier (5 cells);
- Battleship (4 cells);
- Cruiser (3 cells);
- Destroyer (2 cell);
- Submarine (1 cell).

Gameplay

At the start of the application the user should define the size of the Ocean (M and N values) and the ships number (0 or above) for each ship type.

These values can be specified either through the command line arguments, or through the console dialog with the user, when no related command line arguments are provided. You need to document both these scenarios.

(For example, your program may suggest use of a comma-separated list of values in a console dialog to specify the number for each of the ship types, where these numbers are listed in the order which corresponds to ship types sizes (the largest first), then: “0, 1, 2, 3, 4” would mean: “no carrier, one battleship, two cruisers, three destroyers and four submarines”.)

Then the computer randomly places the defined combination of the ships in the Ocean with the following restrictions (adjacency rules): ships cannot overlap, and there are no ships, which occupy immediately adjacent cells, either horizontally, vertically, or diagonally.

If the computer is not able to arrange its Fleet in the Ocean and satisfy the adjacency rules and the Ocean and Fleet parameters, the user is invited to re-enter these parameters or quit the game.

Then the game consists of a series of turns in the form of a console dialog.

At the beginning of the game the human player does not know where the ships are located. The initial screen of the game prints the Ocean and displays its each cell as a “not-fired” (on the cell states see below).

The human player tries to hit the ships. At each game turn the user performs a shot to an Ocean cell and specifies the row and column number of it (in console dialog).

The computer responds with a single bit of information (in a user-friendly form): “hit” or “miss”.

A ship is "sunk" when all the cells it occupies are hit.

When a ship is hit, but it is not sunk, the program does ***not*** provide any information about what type of a ship was hit. However, when a ship is hit ***and*** sinks, the program prints out a message `"You just have sunk a ship-type."`

After each shot, the computer redisplays the current view of the Ocean. The representation of an Ocean cell for a human player is one of the following: “not-fired”, “fired-miss”, “fired-hit”, “sunk” (when a ship, which occupies it, is sunk). The program should use a distinct single character to display each of these cell states. Their choice is implementation dependent, but must be documented.

The user goal is to sink all the Fleet with as few shots as possible; the best possible score would be the total number of the cells occupied by the fleet. (Low scores are better.) When all the ships have been sunk, the program prints out a message that the game is over, and prints the total number of the shots were done.

Implementation details

Name your project `Battleship`, and your package `battleship`.

Enhanced functionality

For an excellent mark (>=8) you need to implement the following two additional features.

1. Torpedo firing mode

At the start of the game the user can define an additional parameter T – the number of torpedoes the user is granted for the game (int value in interval from 0 up to the number of the Fleet ships). If T parameter is greater than 0, the torpedo mode gets enabled.
When the torpedo firing mode is enabled, the user may mark a shot with symbol `T` (as a prefix before the shot coordinates), it means firing a torpedo. Each torpedo shot increments the total shots count as usual ***and*** decrements by one the number of the available torpedoes. When a ship is hit with a torpedo, it becomes sunk entirely, with no respect to its state and size. The shot marked with `T`, when there are no torpedoes available or torpedo firing mode is disabled, must be reported as an error with an appropriate message (“no torpedoes available”…).

2. Ship recovery mode

At the beginning of the game the user can enable ship recovery mode. When this mode is enabled and the user hits a ship, which is not sunk yet, the user has to hit the same ship with the next shot. Otherwise, the ship gets recovered to its initial state (that it had before the first hit in it). It means that to sink a ship the user has to hit it by every shot starting from the first hit and until the ship is sunk; otherwise the user needs to repeat the attack from the very beginning. (It looks like "the loss of tempo" in chess game or a "punishment" in biathlon). The representation of cell states and the shots counter behave as usual.

Obviously, the two modes do not contradict each other: when a ship is sunk by a torpedo it cannot be recovered, since it is already sunk (and a next shot cannot miss it…).

Additional requirements

- Every public method should have javadoc comment(s).
- The program source code should be properly formatted and satisfy the Java Code Style requirements.
- Every method should be short enough to see it entirely on the usual desktop screen (i.e. be "observable " and "readable").
- Any console dialog command must be documented with precise information about its syntax and semantics (in a `ReadMe.txt` file that describes them all).
- All methods, classes and variables should have names that reflect their essence.

Homework results requirements:

- The homework results is your IntelliJ IDEA project. You need to pack it into a ZIP archive. Important: output `.class` files should not be packed into the archive, since they can be recreated on the testing machine.
- You must follow the following pattern to name your archive:
  `<FamilyName>_<ShortName>_<GroupName>_HW1.zip`
- The project unpacked for estimation must be buildable and runnable on a testing machine with JDK 17 installed (without any additional libraries).
- The project must contain the proper artifact building settings to build the executable `Game.jar` with the required application functionality. The resulting Game.jar should be located in JAR subdirectory of the unpacked project.
- Any additional files are welcomed to explain details when it is needed.
- The archive with homework results must be uploaded to the SmartLMS server before the deadline defined below.

Due date-time: 25.10.2021, 23:00.