BATTLESHIP Sprint 2

Sprint 2 must upgrade sprint 1 by giving support to the following features:

- While in sprint 1 players took alternating turns to shoot, in this new sprint, the same player keeps shooting as long as it hits an opponent's ship.
- In Sprint2, the size of the board can be variable (different from 10 rows and columns as in sprint1).
- From Sprint2, when a ship is full of hits, the application must announce "Hit and sunk!" instead of just "Hit!". In addition, the way to display these ships will change.
- Sprint 2 may also improve sprint 1 by enabling placing ships in the boards in non-prefixed positions. That is, from now on, ships can be placed at different positions, probably, different for each fleet. Optionally, these positions can be randomly generated.

It will also bring code refactoring to avoid code repetition and provide a clearer and more maintainable implementation. Refactoring will be done mainly over the classes `TurnSelector`, `Player`, `Ship`, `Water` and `Board`, but this can provoke changes in other classes as well.

First, copy and paste your solution to sprint1 and name it as `surname1_surname2_name_battleship_sprint2`.

Together with this document, there is a `fleetLaunch.txt` file. It contains a piece of code that you can find helpful to implement some tests.

1 Description of the game

The board will change from containing int[][] to Square[][]. So, instead of representing the ocean as an array of integers, we will use an array of **Square** objects. There will be as many Square objects as cells in the array implementing the board. Each square will necessarily contain a reference to a **Ship** or a **Water object**, depending on whether BoardBuilder selects this square to place a Ship or not.

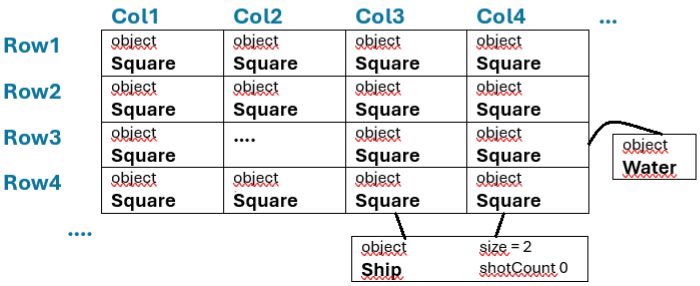


Figure 1 Overall picture of the board

As in sprint 1, ships come in various sizes and can be hit by opponent's shots. So, each ship object must record its **size** as well as the **shots** that hit it.

Turn management will also change. From now on, a player will **continue to fire** until it misses a shot.

Shooting an already shot is not allowed. When a user player types the coordinates of an already shot square, the game will keep asking new coordinates until a non-shot one is returned.

2 New package structure

As new classes are added, there will be a new package structure, as follows:

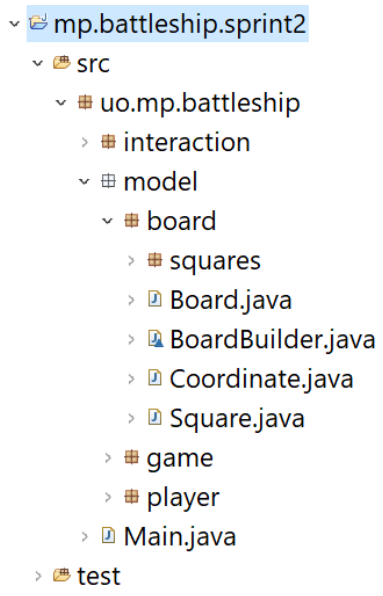


Figure 2: New package structure

3 New classes

3.1 Class Square

It represents each piece (cell) of the board. The **content** of each square will be an object (Ship or a Water). Each square will also store a **state**, that can be shot or not. By default, every square is not shot and will change when it is shot by the opponent player.

Methods:

`Damage shootAt()`

It **marks this square as shot** and **propagates the shot** to the Ship or Water referred by this square. It returns an enum, depending on the damage caused by the shot: **NO_DAMAGE**, **SEVERE_DAMAGE** and **MASSIVE_DAMAGE**. These values are deeply described in next classes.

`boolean isShot()`

If this square has been shot, it returns true and false otherwise.

`char toChar()`

Returns the character corresponding to the content: a Ship or Water. The character will also depend on the state of the square (shot or not). See table 1 below.

`Square setContent(Ship or Water object)`

This method sets the object referred by this square to be the one passed as argument. It could be a Ship or Water object. It returns the Square itself.

`boolean hasContent()`

Returns true if the content of this Square is set to a Ship or a Water object, false otherwise.

Next table shows the **characters to print** the boards. Notice new character '#'.

Displayed square	Description
BBBB	Battleship (each square a **B**)
CCC	Cruiser (each square a **C**)
DD	Destroyer (each square a **D**)
S	Submarine
*****	Hit

#	When the ship is destroyed, every * are replaced by #
Ø	Missed shot (\u00F8, empty set)
Blank space	In the left board, water squares; in the right, those not hit yet

*Table 1: Characters to **display** the content of the boards.*

3.2 Class Ship

It represents a ship that the application will place in the board. **There will be as many instances as ships in the fleet**. Ships can be different length as in sprint1. Apart from the attributes needed to represent a ship, the following is the set of methods to implement:

`Constructor public Ship(int shipSize)`

It creates a new Ship object; each ship must contain, at least, a **size** field and a **shotCount** field to count the number of shots in different positions of the ship.

`public Damage shootAt()`

It is hit by an opponent's shot, and it returns the damage caused.

- **SEVERE_DAMAGE**. Ship is hit but not sunk.
- **MASSIVE_DAMAGE**. Ship is hit and sunk.

`public int size()`

Returns the size of this ship.

`public char toChar()`

It returns the character corresponding to the ship, according to table 1 above.

`public char toFiredChar()`

It returns the character corresponding to hit or hit and sunk, according to table 1 above.

`public boolean isSunk()`

It returns true when ship is sunk; false, otherwise.

3.3 Class Water

It represents a piece of ocean where there is no Ship, i.e., every `Square` object not containing a Ship object must contain a Water object.

Public methods:

`public Damage shootAt()`

As it is not a Ship, it always returns **NO_DAMAGE**.

`public char toChar()`

It returns the character associated with a Water cell, according to table 1 above.

`public char toFiredChar()`

It returns the character associated with a missed shot, according to table 1 above.

Each Square object must contain a Ship or Water instance. Classes Ship and Water do not share common attributes or method implementations.

However, Square objects will need to invoke the same public methods in whatever instance inside.

So, it could be a good idea to design Ship and Water classes using a common interface, **Target**.

4 Refactoring classes

In this section, we describe some modifications to be done in existing classes. These modification may cause minor modifications in other classes, **not mentioned in this document** (for example, `Game::play` will change as the shot returns a Damage, instead of a Boolean).

4.1 Refactoring class Board

Rewrite class Board and **replace the array of integers by an array of Square**. This will certainly involve **changes in the implementation** of some methods, although most methods will not change **their signature**.

However, method **`shootAt`** will return the severity of damages and not only true/false as in sprint 1, the package constructor, will receive Square[][] instead int[][] and the package method will return Square[][] instead int[][].

Methods:

`constructor public Board(int size)`

It must create a fleet of ships, by default the same as in sprint1, and pass it to BoardBuilder. Then, BoardBuilder will create a Board of the given size and arrange the fleet there, as in Sprint 1.

`Board(Square[][] arg)`

It sets the inner array to a specific array passed as argument. This array does not need to comply neither the required size (between 10 and 20) not the float (4 submarines, etc) but it can be a reduced version, at your convenience. Useful for testing.

`public Damage shootAt(Coordinate coordinate)`

It records a shot in this square and returns the damage.

`Square[][] getInnerArray ()`

It returns a **copy** of the inner array of squares. Array type has a public method `clone()` that returns a duplicate copy of the same array. Useful for testing.

4.2 Refactoring class BoardBuilder

Since the array built by `BoardBuilder` changes, the implementation of the class itself needs also be reviewed.

`static Square[][] build(int size, List<Ship> fleet)`

Rewrite the method so it returns an array of Square. This method must

- create a 2D squared array, the size passed as argument, of Square objects,
- place the fleet passed as argument in the array according to some rules described later.
- Fill the array up with instances of class Water.

What follows are the rules to randomly place ships in the board. This is not a trivial problem so, if you find too difficult to implement this method, you may place the ships at fixed positions, the same as in sprint1. Or use the package protected constructor to set the ships at other positions at your convenience.

4.3 Rules to place ships in the board (OPTIONAL)

If you decide to implement the rules, method `build()` should be modified to **randomly** select locations for the ships, making sure this process follows **all the next rules to place the ships**:

- Each ship occupies several consecutive squares on the board. Ships must be placed **horizontally** or **vertically** (never diagonally) across board squares.
- Ships cannot be placed next to each other (at least one Water square all around the Ship).
- **Do not change the position** of any ships once the game has begun.

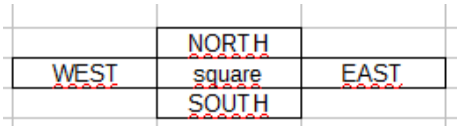
4.4 Refactoring class Coordinate

Let us add a new public method:

`Coordinate go(Direction direction)`

It returns a new Coordinate object, next to this coordinate, moving to the direction received as argument **NORTH**, **EAST**, **SOUTH,** or **WEST**. Notice that this method will **always return a Coordinate** object, even if it is out of the board.

For example, if square s = (col = 0, row = 0) `s.go(Direction.NORTH)` will return (col = 0, row = -1).



4.5 Refactoring classes HumanPlayer and ComputerPlayer

Refactor HumanPlayer and ComputerPlayer so they can be referred by **`Player`** superclass. **Common attributes and method implementations** must also be refactored.

Public methods:

`public Damage shootAt(Coordinate coordinate)`

As described in previous methods, shootAt will return the damage.

`public Coordinate makeChoice()`

No player is allowed to repeat shots. When a user player shoots to an already shot square, this input will be ignored, and player must keep typing coordinates until a non-shot one is returned. Look *public boolean contains(Object element)* in the data structure you are using to store already shot positions.

4.6 Refactoring TurnSelector class

As in the first sprint, it handles the turns to play. The first time it is always user's turn and when a player hits an opponent's ship, next turn will be for this same player again.

Package protected methods:

`Constructor TurnSelector(Player user, Player computer)`

Modify the constructor to receive players as arguments.

`Player next()`

It returns next player to play. It will return alternating players except if *repeat()* has been invoked just before, what means that player has hit an opponent's ship.

`void repeat()`

If it is run, *next()* will return current player again.

4.7 Refactoring the class ConsoleWriter

Change the signature of the following method:

`public void showShotMessage (Damage impact)`

It is very similar to the existent version, except that the parameter is a Damage and the messages will depend on this value (MISS, HIT!!. Continue, HIT AND SUNK!!. Continue)

4.8 Refactoring the class Game

First, add this new public constructor

`Constructor public Game (HumanPlayer human, ComputerPlayer computer, int size)`

It is very similar to the existent constructor, except by a new parameter, referring to the size of the boards to play instead of default value. It must be in range [10, 20]; otherwise, default size is used. Try to implement this new constructor avoiding code repetition.

The basic behavior of the class Game will remain the same. The **only differences** will be:

- When players hit an opponent's ship, they repeat turn.
- When a player hits a ship, the application prints *Hit! Continue*. However, if the player shot down a ship, the application prints *Hit and Sunk! Continue*

5 UML Class diagram

After reading the previous sections a couple of times and before implementing, draw a UML class diagram that models the relationships between the classes of the project including Main, Game, Board, Player, HumanPlayer, ComputerPlayer, Square, Target, Ship, Water, TurnSelector and Coordinate. Other classes are unnecessary. Show multiplicities.

For each class, also show attributes and public methods.

For the diagram:

- Use a tool (VisualParadigm or other) an export the diagram as a graphic format.
- Make sure that no two lines cross each other.
- Lines can be either horizontal or vertical or combine both with 90-degree bends.

6 Test

First, **review tests implemented in sprint 1. Some of them should be removed as repeated shots are not allowed in this sprint. The others must keep succeeding**. Then, implement the following tests.

6.1 Class Coordinate

Test method `go(Direction direction)`

Test Coordinate objects returned by this method. Use cases:

- Test a Coordinate at column A, direction West.
- Test a Coordinate at row 0, direction North.
- Test all directions with a Coordinate at row and column different from 0.

6.2 Class TurnSelector

Test method `next()`

Test that two consecutive calls to *`next()`* will return alternating players.

Test method `repeat()`

Test that, after running *`repeat()`*, *`next()`* will return the same player again.

6.3 Class Square

Test method `shootAt(Coordinate coordinate)`

- Shoot at Water, returns NO_DAMAGE.
- Shoot at Submarine, returns MASSIVE_DAMAGE.
- Shoot at a brand-new Destructor, returns SEVERE_DAMAGE.
- Shoot at an almost-sink Destructor, and sink it, returns MASSIVE_DAMAGE.
- Read **launchFleet.txt**. Use this fleet to write your tests.