Battleship sprint3

1 Main goal: reduce coupling in sprint2

In sprint 2 there are classes `ConsoleReader` and `ConsoleWriter` with public methods as `showGameStatus`, `showWinner`, `showTurn`, `readCoordinates`, etc. These methods are the ones used to implement user interaction.

Methods in ConsoleWriter are invoked from `Game` while method in ConsoleReader is invoked from `HumanPlayer`. These invocations cause undesirable dependencies between classes implementing the logic of the game (`Game` and `HumanPlayer`) and the ones implementing the user interaction.

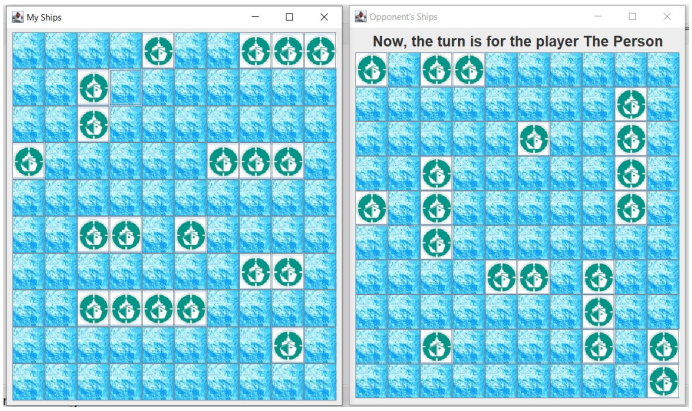
These dependencies as known as **coupling**, degree to which software components depend on each other.

For example, in the following picture a code snippet of class `Game` in sprint 2 is printed.

```
private ConsoleWriter interactor = new ConsoleWriter();


private Damage playTurn(Player player) {
    Coordinate target;
    Damage impact;
                                                    user interaction
    interactor.showGameStatus(human.getMyShips(), human.getOpponentShips(), gameMode);
    interactor.showTurn(player.getName());
    target = player.makeChoice();
    interactor.showShootingAt(target);
    impact = player.shootAt(target);      logic
    interactor.showShotMessage(impact);
    return impact;
}
```

Let us assume that we implement a class GUIWriter that implements the same methods as ConsoleWriter but in a graphical environment.



Then, changing the way output is done means **changing the code of class Game**, replacing:

```
private ConsoleWriter interactor = new ConsoleWriter();
```

by: `private GUIWriter interactor = new GUIWriter();`

And similarly, for ConsoleReader.

Using a user interface or another should not cause changes in the classes implementing the logic as `Game` or `HumanPlayer`. These classes should only be concerned with the game rules and not about how to print or read information.

The goal of this sprint is **decoupling the classes implementing the game model and rules from the classes implementing the user interface**, making it possible to play the same game, **without changes**, through different user interfaces.

2 Structure of sprint3

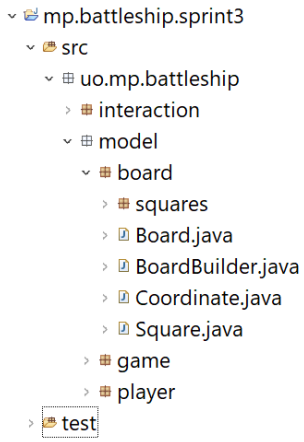
2.1 New projects

The game will be implemented by four different projects:

- **util:** A project containing auxiliary classes.
- **mp.battleship.sprint3** to implement the game (model and rules). This project **will not contain class `Main` because the game will not be launched from here anymore** but from one of the following projects acting as game interfaces.
- **mp.battleship.sprint3.console** implements a command-line interface (text-based interface). It will be half-implemented by students. It will contain a class `Main` to launch the game implemented in sprint3.
- **mp.battleship.sprint3.gui** implements a Graphical User Interface (GUI). It is fully implemented by the teaching staff. It will contain a class `Main` to launch the game implemented in sprint3.

2.2 Project battleship.sprint3 structure

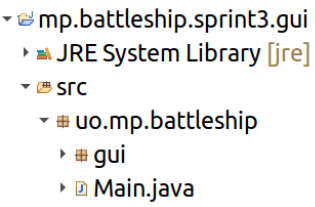
- package **battleship** contains java classes implementing the game. Notice there is no class Main anymore.
- package **battleship.board** contains java classes to model the game board (classes `Board`, `Coordinate`, `Square` …)
- package **battleship.game**: Java classes implementing basic rules. It contains classes `Game` and `TurnSelector`.
- package **battleship.player** contains classes to model the players. From this sprint, the constructor must validate the argument. It must be a non-null, non-empty and non-blank string or an IllegalArgumentException is thrown. There will not be a default name anymore.



2.3 Graphical user interface project structure

Project **mp.battleship.sprint3.gui**, fully implemented by the teaching staff, launches a graphical interface. It is possible to click over squares to shoot. The package structure is as follows:

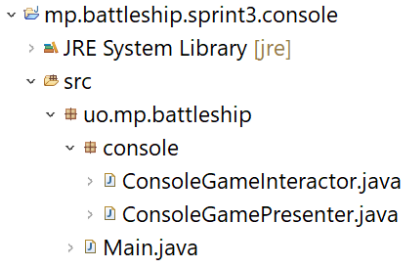
- package **gui** contains java classes needed to implement the GUI.
- class **Main** can be used to launch the game with a graphical interface.



2.4 Text-based interface project structure

Finally, project **mp.battleship.sprint3.console** contains

- a class `Main` to launch the game with a text-based interface like that in sprint1 and sprint2, and
- a package **console** where student must implement classes `ConsoleGamePresenter` and `ConsoleGameInteractor` described later in this document, as well as any auxiliary classes needed.



3 Implement separate classes for UI and logic game

Interaction classes should implement methods to communicate the game and the player (input/output) while **logic classes** are those implementing the rules of the game.

To run the game application using different interactors and presenters **without requiring any additional change in classes implementing the game (decoupling)**, we will:

- Write **two java interfaces, GamePresenter** and **GameInteractor,** containing methods for printing information to the user or getting information from the user, respectively. Implementation details are **hidden behind these interfaces.**
- Write **different implementations of that interfaces**.
    - Two implementations of interface GamePresenter: ConsoleGamePresenter and GUIGamePresenter. The first will print in console while the second will use a graphical output.
    - Three implementations of interface GameInteractor: ConsoleGameInteractor (read information from standard input), GUIGameInteractor (read information from graphical input) and RandomGameInteractor (generates random values, for the computer player).
- Use instances of interfaces GamePresenter and GameInteractor where the classes of the game need them. More specifically, class Game needs a presenter while each player needs an interactor.

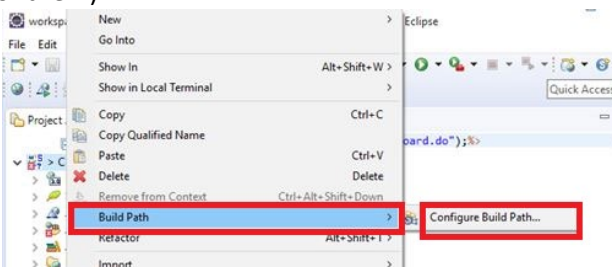
4 Implementation

4.1 Put all the projects together.

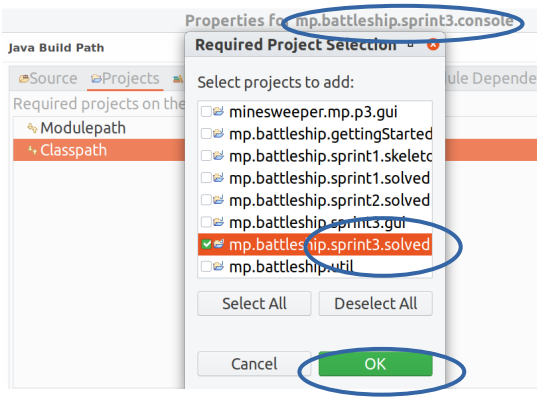
Projects **`mp.battleship.sprint3.gui`** and **`mp.battleship.sprint3.console`** are provided together with this document. Both contain Main classes to launch the game.

Below, there are some instructions to proceed (from any of them):

1. Import them in your workspace and for each
2. select Build Path → Configure Build path



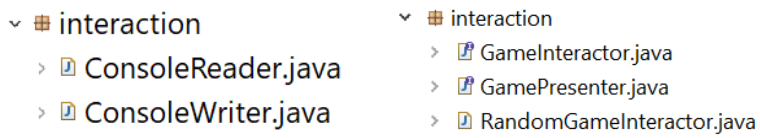
3. In **Projects** tab, **Add** project sprint3, click **OK** + **Apply and Close**



To launch the game, run any of the available `Main` classes in the projects.

4.2 Package interaction

There will be important changes in package ***`uo.mp.battleship.interaction`***. Compare sprint2 on the left with sprint 3 on the right:



`RandomGameInteractor` implements `GameInteractor` and produces a randomly generated coordinate.

4.3 Interface GamePresenter

Write it in package ***uo.mp.battleship*.interaction.**

`GamePresenter` interface consists of the following methods to **display information**:

`void showGameStatus(Board left, Board right, boolean gameMode)`
: It displays the state of the game including boards, the original fleet, and the remaining fleet. Both `left` and `right` parameters are references to the Board instances to be displayed. The third parameter is true when playing in debug mode; false otherwise.

`void showGameOver()`
: It informs the player the game is over by printing a message "GAME OVER!!!!"

`void showWinner(Player theWinner)`
: It displays the name of the player who won the match by printing a message as: "The winner is <player name>. Congratulations!!!". Notice the change in the type of the parameter.

`void showShotMessage(Damage impact)`
: It tells the user the result of the shot with messages as "Hit! Continue", "Hit and sunk! Continue" or "Miss!".

`void showTurn(Player player)`
: It tells the user whose turn is right now: "Now, the turn is for player <PLAYER NAME>". Notice the change in the type of the parameter.

`void showShootingAt(Coordinate coordinate)`
: It displays a message identifying target square: " Shoot at <coordinate in user-friendly format>"

There will be two different implementations of this interface with different implementations of the methods (**polymorphism**). Teaching staff provides `GuiGamePresenter` while student must implement `ConsoleGamePresenter` printing information in the standard output.

4.4 Interface GameInteractor

Write it in package **uo.mp.battleship.interaction.**

`GameInteractor` interface consists of just one public method:

`Coordinate getTarget()`
: It returns a coordinate to be shot.

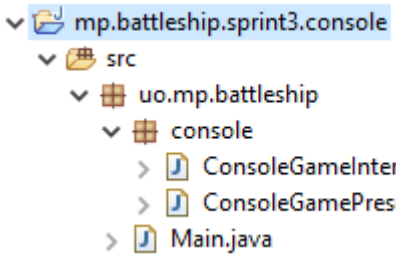
At least, three implementations of this interface will be needed: gui, console and random (for computer player). While the first is fully implemented by teaching staff, the student must implement the other two.

4.4.1 Class RandomGameInteractor

RandomGameInteractor implements GameInteractor and returns a randomly generated coordinate.

Notice that this code is already implemented in sprint2, specifically in ComputerPlayer::makeChoice().

4.5 Project mp.battleship.sprint3.console



The figure shows the final structure.

- `Main` is provided by teaching staff.
- Write a class `ConsoleGamePresenter` implementing the interface GamePresenter using the standard output. You may reuse most of the class `ConsoleWriter`.
- Write a class `ConsoleGameInteractor` implementing `GameInteractor` using standard input. You may reuse most of the method `makeChoice` in class `HumanPlayer`.
- Finally, update method Main::configure(). Create an instance of GamePresenter and inject it in game (`game.setPresenter`). Then, create an instance of `ConsoleGameInteractor` and an instance of `RandomGameInteractor` and inject them in the human player and the computer player instances, respectively (`setInteractor`).

4.6 Adapting the code of class Game

It is necessary to add the following public method to class `Game`:

```
void setPresenter( GamePresenter arg )
```

The argument will GamePresenter to communicate with the player (output). Null argument will produce an IllegalArgumentException

After creating a `Game` object, `setPresenter()` must be invoked before executing any method involving communication with the person playing the game. Otherwise, an IllegalStateException is thrown.

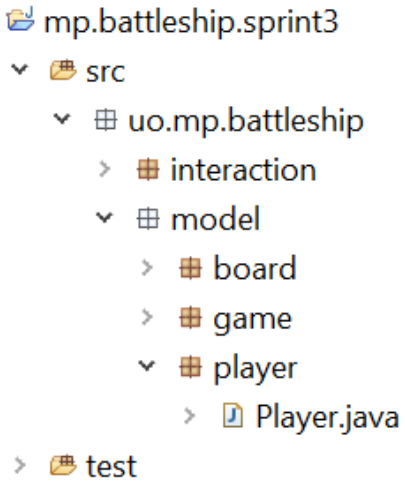
4.7 Changes in classes HumanPlayer and ComputerPlayer

It is necessary to add the following **public method** to class `Player`:

```
void setInteractor( GameInteractor arg )
```

The argument will be used to communicate with the player (input). Null argument will produce an IllegalArgumentException.

After creating the player object, ***`setInteractor()`*** must be invoked **before** executing any method involving input/output. Otherwise, an IllegalStateException will be thrown.



Then, the code used **to get the target coordinate must be replaced by an invocation to getTarget() in the `interactor`** object either random interactor, console interactor or gui interactor, depending on the argument passed in setInteractor.

With all the changes done in this and previous sections (move the code to generate a target coordinate from HumanPlayer and ComputerPlayer to GameInteractor and replace the code in HumanPlayer and ComputerPlayer by an invocation to this interactor), the only difference between classes HumanPlayer and ComputerPlayer disappears, and **we can replace them by a single class Player**.

5 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 and interfaces of the projects sprint3 and sprint3.console. Other

classes are unnecessary. Show multiplicities. Show attributes and public methods (**remember that you must never include those attributes already represented by association relationships**).

For the drawing:

- Make sure that no two lines cross each other.
- Lines can be either horizontal or vertical or combine both with 90-degree bends.
- Export it in a graphical format (png or jpeg).

6 Test

Review tests implemented in sprint 2. They must keep succeeding.

7 Final instructions

Remember to rename your projects before submitting by surname_surname_name.sprint3 and surname_surname_name.sprint3.console.

Keep in mind that it is essential to maintain names as well as package structure in your project.

Create a new folder named **UML** in project sprint3 and copy the UML diagram (png or jpeg) there.