Battleship sprint 4

1 Overview of the sprint4

Before submitting the project, we will add new features to the game as **game sessions**, **game ranking**, and **exception management**. However, not all will be fully implemented right now. Some of them will be completed in the next sprint.

Game session: A session is everything that happens from the moment the user launches the game until it finishes. In this version, the user must type a username first in the session and then, the user will be able to **play** one or more games, view the game **ranking** or **exit** the game.

Menu: Each time the application is launched and at the end of a game, the application will display **menus** with several options. Not all the options will be implemented in this sprint. Some of them will be unavailable.

Exception management: Exceptions thrown during program execution will be caught and managed.

2 Organize the code in several projects.

2.1 Project mp.battleship.sprint4.console structure

As in previous sprint, it will contain **classes needed to interact with the application through the console**: ***ConsoleSessionInteractor***, ***ConsoleGamePresenter*** and ***ConsoleGameInteractor***. It will also contain a *Main* class to launch the game using the console to communicate with player.

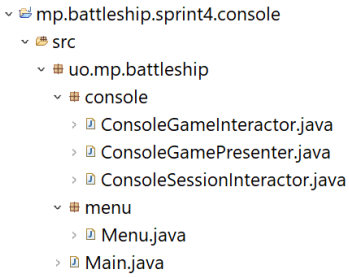

Figure 1 – Console project

Please note: Along with this statement, a **new `Main.java`** is provided to **replace** old one (package mp.battleship). This new Main.java must be copy **as it is**. Do not modify it.

2.2 Project mp.battleship.sprint4 structure

This project is the core of the game and must follow the package and class structure shown in Image 2.

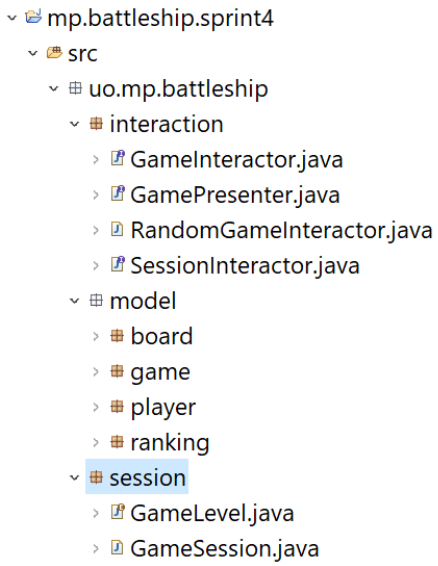

Image 2 – Package structure

3 UML class diagram

Please note: The **UML class diagram** shown in the image below, includes **fundamental relationships** between **fundamental classes**, but **it does not include methods, attributes, or utility classes**. It did not show many usage relationships either.

It is not intended to impose the classes in the diagram should be the only ones in the system. **You may create new classes, if necessary**, as long as you **stick to the names and relationships** shown in this and previous statements.

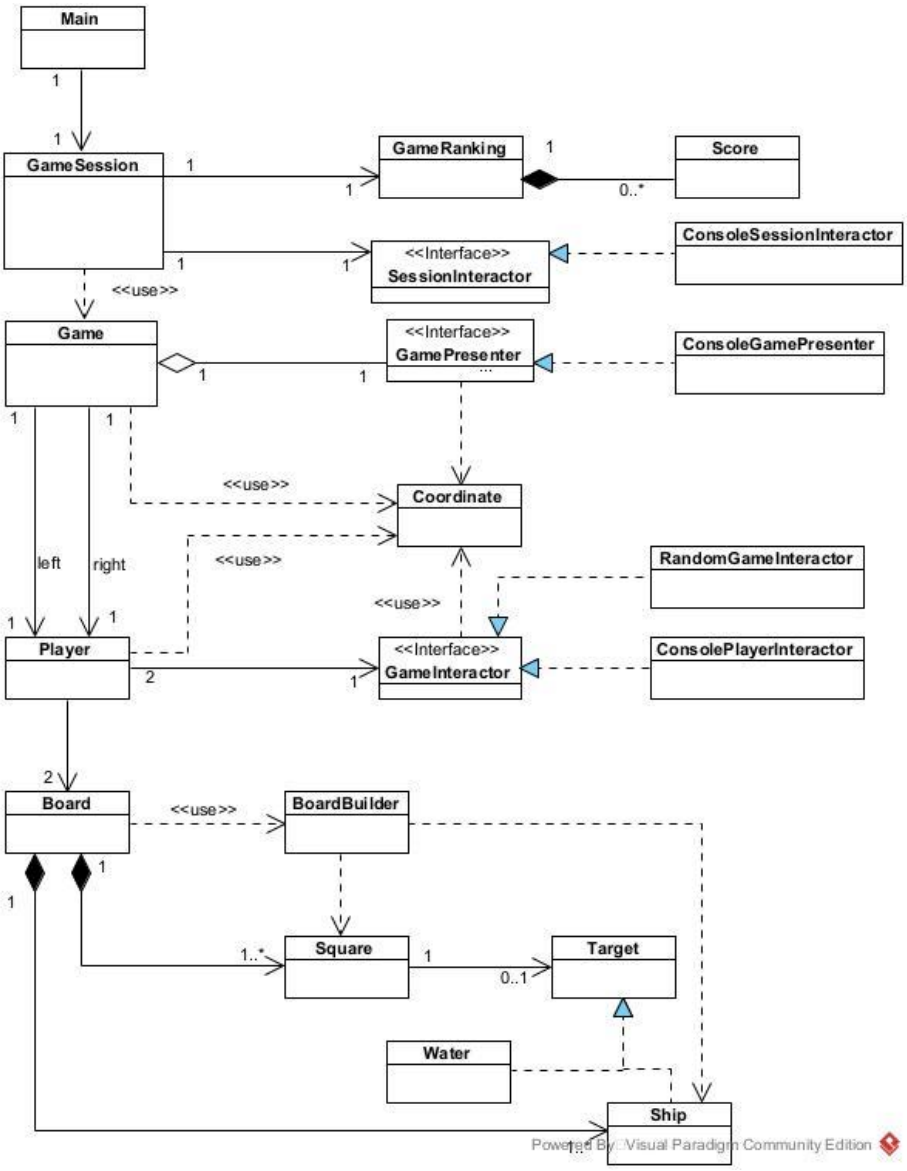

Image 3 - Simplified UML class diagram

4 Session implementation

To implement sessions, several classes and interfaces are needed.

4.1 Class GameSession

Class *`GameSession`* will implement a game session and **leads the interaction** between user and application.

Attributes

This class may need these **attributes**:

1. A ***`game interactor`*** object that implements the ***`GameInteractor`*** interface. It deals with getting user input data (next shot). At this stage, there are three implementations: random, gui and console.
2. A ***`game presenter`*** object that implements the ***`GamePresenter`*** interface and carries out actions to display data to the player as printing a welcome message, showing game board, or informing the user when game is over.

3. A ***session interactor*** object that implements the ***SessionInteractor*** interface and manages the **interaction between the user and the current session**, including requesting the username, display menu or execute a menu option.
4. A ***ranking*** object of type ***GameRanking***. It stores the scores achieved in the games, during a session.

Public methods

`void run()`
It runs the actions described below.

`void setSessionInteractor(SessionInteractor interactor)`
It sets field *SessionInteractor* to the argument.

`void setGameInteractor(GameInteractor interactor)`
It sets field *GameInteractor* to the argument.

`void setGamePresenter(GamePresenter presenter)`
It sets field *GamePresenter* to the argument.

`void setGameRanking(GameRanking ranking)`
It sets the *GameRanking* field to the argument.

Class GameSession will also contain as many **private methods** as necessary to produce clean and maintainable code. As a guideline, a good implementation of the run() method should not contain more than 7 lines.

Method run

This method implements the logic of the session. It performs the following actions:

- Firstly, it asks the user for a **username.**
- Secondly, it **creates player objects** (user and computer player). Now, the **user player can be fully configured** with the username and the same game interactor as in game session. The computer player needs also be configured with a game interactor (`class RandomGameInteractor`) but, unfortunately, at this stage, an instance of `RandomGameInteractor` cannot be created as the constructor needs the size of the board and it will be entered by the user later.
- Thirdly, it displays a **menu** with the following options:
    1. **Play** a game (see detailed description below).
    2. Print the **scores of all the players.**
    3. Print **personal scores**.
    4. **Exit**
- Finally, it **requests** the user an **option and execute** the selected one. Each time an option is processed, the menu will be printed again, unless the user selects exit; then, both session and program execution will finish. This means that, during a single session, several games can be played, and the scores are available (next sprint will make the game ranking persistent).

Playing a game

When the first option (**play a game**) is selected, the object *GameSession* should

- ask user to select the **difficulty level** and **debug mode**.
- The difficulty level determines the size of the board so now, an instance of **RandomGameInteractor** can be created and used to configure the computer player.
- create a new *Game* object with the players, the debug mode, and the size of the board, and
- set the game with the same presenter used in the game session.

Finally, method *play()* on this object *Game* is run and a **full battleship game will be played as in sprints 3**.

When **game is over** and **user player won**, she is **asked whether she wants to include the score in the ranking**. If so, a `Score` object is added to `GameRanking` with information about this game (see Section 5.1).

4.2 Enum GameLevel

The game can be configured to be played with different levels of difficulty: ***SEA***, ***OCEAN***, ***PLANET***. Each value will determine a size of the board: 10x10, 15x15 and 20x20, respectively. These three values will be represented by an **enum** called ***GameLevel***.

4.3 Interface SessionInteractor

`SessionInteractor` offers **methods to manage all the user interaction referred to the session** (not with the game itself). All input methods must return valid values so they must validate the string typed by the user and, if it doesn't match any expected values, must warn the user, and ask again.

```
GameLevel askGameLevel();
```
It asks the user for a level of difficulty and returns the answer with a `GameLevel` object.

```
String askUserName();
```
Prompts the user for a name and returns a String with the answer, which cannot be neither null nor empty.

```
void showMenu();
```
It displays the menu

```
int askNextOption();
```
It asks the user to choose an option. It returns an integer representing the selection. A value greater than zero will represent some of the available actions. A zero value will always represent the exit option.

```
boolean askDebugMode();
```
It asks the user to decide whether they want to execute the game in debug mode (y) or not (n). If the user answer y, the method returns true; otherwise, returns false.

```
boolean doYouWantToRegisterYourScore();
```
At the end of a game, it asks the user if they want to save their score (accepted answers are y, n). Returns true if the answer is affirmative and false otherwise.

```
void sayGoodbye();
```
It just print a goodbye message.

```
void showRanking(List<Score> ranking);
```
It receives a list of `Score` objects representing all the scores recorded in the system and displays all the information about them all (tabular format, one line for each score).

```
void showPersonalRanking(List<Score> ranking);
```
It receives a list of `Score` objects representing all the scores recorded in the system and displays all the information about them all (tabular format, one line for each score) except username (it is the current user own name).

```
void showErrorMessage(String message);
```
It displays an error message received as a parameter in the standard error. This method is for warning the user about recoverable errors.

```
void showFatalErrorMessage(String message);
```
It displays severe error messages received as parameter in the standard error. This method is for warning the user about unrecoverable errors.

4.4 ConsoleSessionInteractor class

Class ***ConsoleSessionInteractor*** will be implemented in mp.battleship.sprint4.console. It implements methods of *SessionInteractor* interface to work with the Java console.

In text mode, all the methods **requesting information** from the user must be preceded by a question, so user knows what their options are. Methods to **display information** must generate a string with an appropriate format. Some examples are shown below.

`String askUserName();`

```
Player name?  Player
```

`int askNextOption();`

```
Available options:
  1- Play a new game
  2- Show my results
  3- Show all results
  0- Exit
Option? 2
```

`GameLevel askGameLevel();`

```
Option? 1
Level (s)ea, (o)cean, (p)lanet ?
```

`boolean askDebugMode();`

```
Do you want to play in debug mode (y)es, (n)o ? y
```

`void showPersonalRanking();` (Fully provided)

```
Date            .Hour           .Level   .Time
2024-04-08      10:49:55        SEA       55
```

`void showRanking(List<Score> ranking);` (Fully provided)

A header and some lines like the following:

```
User name       .Date           .Hour           .Level   .Time
player          2024-04-08      10:49:55        SEA       55
```

`boolean doYouWantToRegisterYourScore();`

```
Do you want to store your score? (y)es, (n)o
y
```

Method `void sayGoodbye();`

```
Thank you for playing Battleship. Bye bye!
```

5 Ranking implementation

When a game finishes, **if user player won**, the app will ask they whether they want to save the score in the ranking. If the user agrees, the application will save some information about this game.

In this version, ranking is lost when session finishes, that is, every time the application execution ends. Next sprint it will become persistent, saving the results in a file that can keep information on different games played in different sessions.

Only won games will be saved and only if the user agrees.

5.1 Score class

It stores information of a single game.

- User player username
- Level of difficulty
- End date and time of the game
- Elapsed time

It contains these public methods:

`Score(String userName, GameLevel level, LocalDateTime initial, LocalDateTime end)`
Constructor.
- userName: name of the user playing the game.
- level: level of difficulty.
- initial, end: Initial and end times of the game.

The constructor will save the **username**, the **level**, the **end time** (LocalDateTime object) when the game finished **and elapsed time** (in seconds) that is the time between initial and end and will be calculated as `ChronoUnit.SECONDS.between(start, end);`

`String getUserName()`
Returns the value of userName field.

`long getTime()`
Returns the value of elapsed time field in seconds.

`LocalDateTime getEndTime()`
Returns the value of date field.

`GameLevel getLevel()`
Returns the value of level field.

5.2 GameRanking class

This class stores a list of *Score* objects representing finished games and it offers methods to query that list. It contains the following public methods:

`void append(Score score)`
It adds the argument to the end of the list of scores.

`List<Score> getRanking()`
It returns a copy of the list of scores.

`List<Score> getRankingFor(String userName)`
Returns a list containing only those scores whose username matches the argument.

6 Refactoring other classes in the application

6.1 New class Main.java for project console

As the one in sprint 3, the new *Main.java* also deals with launching the game. It creates an object of type `GameSession`, sets values for its fields and invokes its method `run( )`. A working version is provided.

6.2 Class Game

The original class Game must be added new attributes

- `LocalDateTime start`, that saves the moment the game starts, and
- `LocalDateTime end`, that saves the moment the game finishes.

And the following public methods:

`LocalDateTime getInitialTime()`
Returns the time when the game started

`LocalDateTime getEndTime()`
Returns the time when the game finished

6.3 Interface GamePresenter

A new public method must be added to the interface `GamePresenter`:

`void showErrorMessage(String message);`

It displays an error message received as a parameter in the standard error (`System.err`).

7 Exceptions

7.1 Programming errors

If, during program execution, an exception is thrown due to **programming errors** (**`RuntimeException`**), the procedure will be as follows:

- **`GameSession`** object will catch the exception.
- Program **execution will be finished without crashing**, even if the user has not selected exit option.
- User will be informed that an error happened.
  - Use **SessionInteractor::showFatalError (msg)** to warn the user before the application exits.
  - The message must be something as "**FATAL ERROR:"** followed by the message contained in the exception, followed by "The application must stop execution."

Programming errors are programmer fault, and they are only acceptable at development time. It should never happen at runtime as you (the programmer) should fix all programming errors before deployment.

7.2 User errors

Many user errors can happen during execution. Some of them have already been considered in **section 4.3 Session Interactor**; the user types some string that cannot be translated into an acceptable game element (blank strings, invalid options, etc.), the program will warn the user and ask again.

Apart from them, we are considering another user error: when the user types a coordinate that does not fit the board limits. Then, a user defined **`InvalidCoordinateException`** will be thrown containing the coordinate causing the error.

The method `Board::shootAt` must be refactored to throw the exception.

To handle it, the message must be printed out (use `GamePresenter::showErrorMessage(String s)`) and the game continues with the other player (players shooting outside loses their turn).

7.3 InvalidCoordinateException

It must hold an attribute Coordinate and add the following to the methods inherited from `Exception`:

`constructor InvalidCoordinateException(Coordinate c)`

Receives the coordinate causing the error and calls super with a message describing the error as:

`<coordinate-value> is out of the board. You lose your turn.`

It also saves the coordinate to use it later.

`Coordinate getCoordinate( )`

Returns the coordinate causing the failure.

8 Test

It may be necessary to add additional tests to the methods. Be **sure all tests, including existent ones, are green**.