Battleship sprint 4

1 Descripción general del sprint4

Vamos a añadir las siguientes características a nuestro proyecto battleship: sesiones, clasificación, manejo de excepciones y registro (logging). Sin embargo, no todo se implementará por completo en este momento y se completará en el próximo (y último) sprint.

Sesión de usuario: Una sesión de usuario es todo lo que sucede entre el jugador humano y la aplicación desde el momento en que el usuario inicia el juego hasta que finaliza. En esta versión, después de iniciar sesión (**log in)**, el usuario podrá **jugar** una o más partidas, ver la **clasificación** del juego o **salir** del juego.

Menú: Cada vez que se inicia la aplicación y al final de un juego, mostrará un **menú** con varias opciones. No se implementarán todas las opciones en este sprint. Algunas de ellas no estarán disponibles.

Manejo de excepciones: las excepciones lanzadas durante la ejecución del programa serán recogidas y manejadas.

Registro de errores: algunas excepciones se manejarán registrando (logging) la causa del error.

2 Organización del código en varios proyectos.

2.1 Estructura del proyecto mp.battleship.sprint4.console

Como en el sprint anterior, contendrá **clases que implementen las interfaces necesarias para interactuar con la aplicación a través de la consola: *`ConsoleSessionInteractor`***, ***`ConsoleGamePresenter`*** y ***`ConsolePlayerInteractor`***. También contendrá una clase *`Main`* para iniciar el juego usando la consola para comunicarse con el jugador.

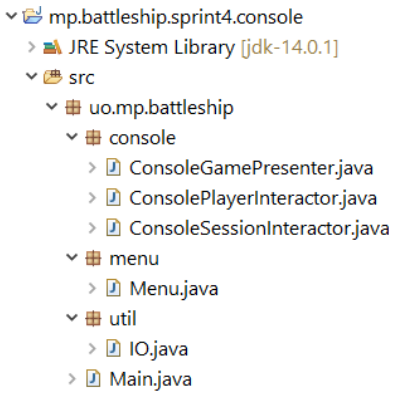

Image 1 – Console project

Importante: Se proporciona un nuevo fichero ***`Main.java`*** para *reemplazar* al antiguo (paquete mp.battleship). Este nuevo fichero *`Main.java`* debe copiarse tal cual, no se debe modificar.

2.2 Estructura del proyecto mp.battleship.sprint4

Este proyecto es el **núcleo del juego** y debe seguir la estructura de paquetes y clases que se muestra en la siguiente *Imagen 2*.

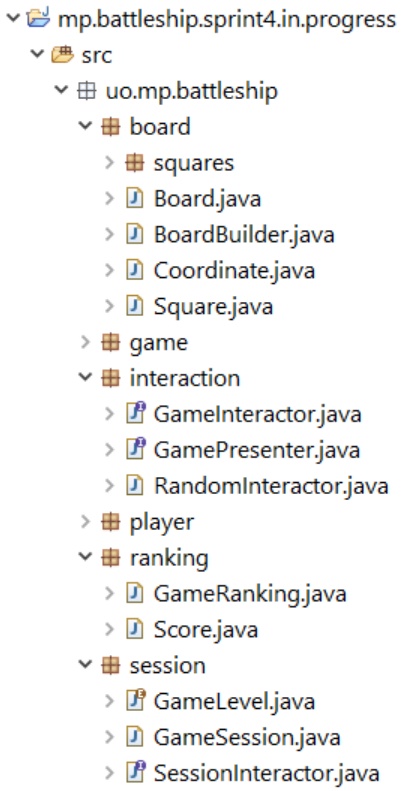

Image 2 - Packages

2.3 Estructura del proyecto mp.battleship.util

Puede incluir más clases que implementen funcionalidad reutilizable, si es necesario.

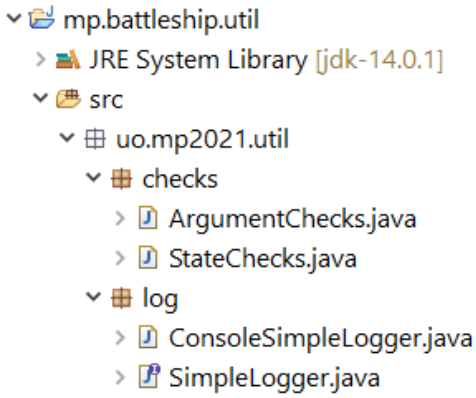

Image 3 –project util

3 Diagrama de clases UML

El **diagrama de clases UML** que se muestra en la imagen siguiente, **incluye relaciones entre clases fundamentales**, pero **no incluye métodos, atributos o clases de utilidad**. Tampoco se muestran algunas relaciones de uso.

No se pretende imponer que las clases que aparecen en el diagrama, sean las únicas del sistema. **Se pueden crear nuevas clases si es necesario**, siempre que se ciña a los nombres y las relaciones que se muestran en esta imagen y en las anteriores.

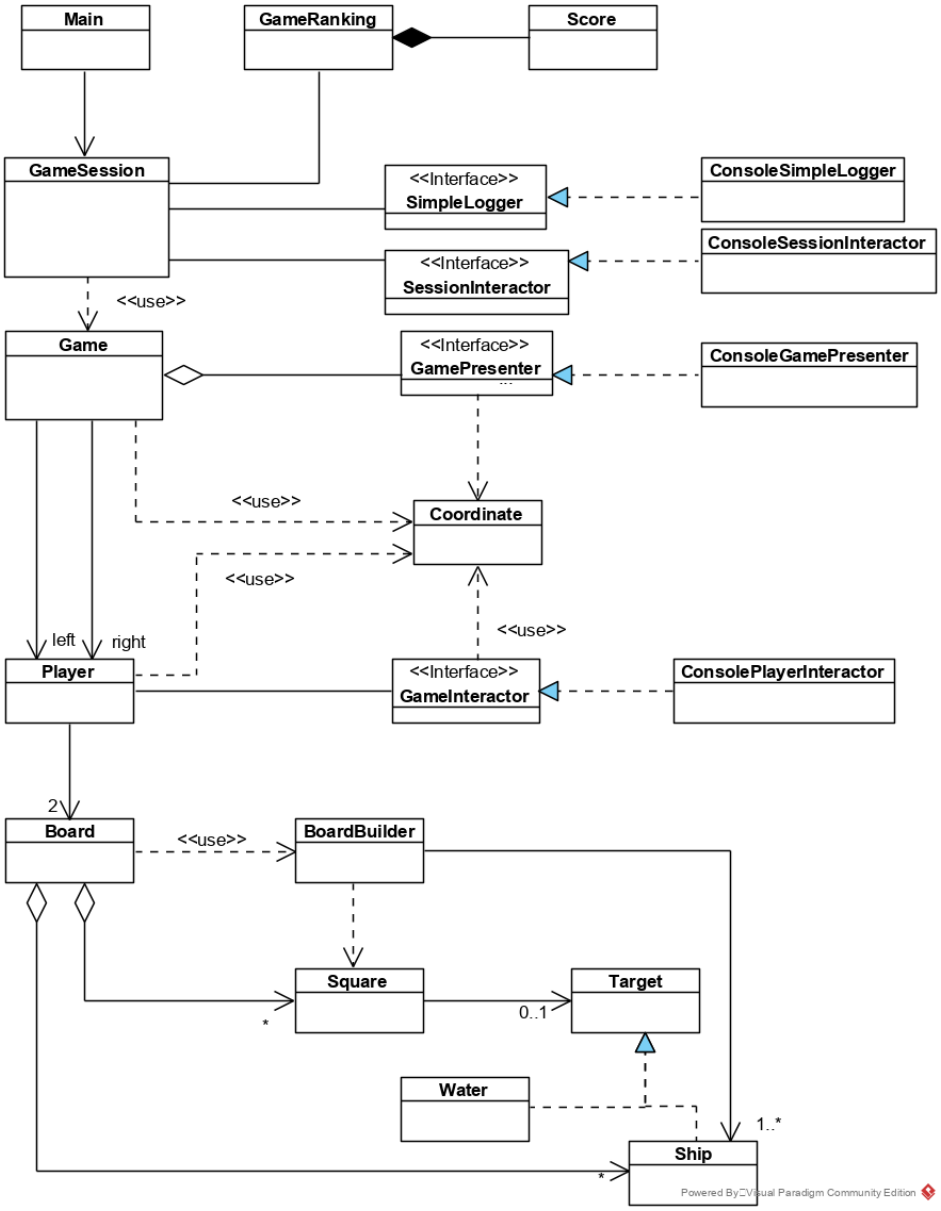

Image 4 - Simplified UML class diagram

3.1 Nueva clase Main.java

Como el del sprint 3, el nuevo *`Main.java`* también se ocupa del lanzamiento del juego. Crea un objeto de tipo *`GameSession`*, establece valores para sus campos e invoca al método *`run()`*.

3.2 GameSession

La clase *`GameSession`* determina la interacción entre el usuario y el juego. Este objeto debe realizar las siguientes acciones:

- **Cuando se inicia**, le pide al usuario un **nombre**.
- Luego, mostrará un **menú** con las siguientes opciones:
  - **Iniciar** un juego.
  - Mostrar las **puntuaciones de todos los jugadores.**
  - Mostrar **puntuaciones personales**.
  - **Salir**
- **Solicitará** al usuario **una opción** y se **ejecutará** la opción elegida.

Cada vez que se procesa una opción, el menú se mostrará nuevamente, a menos que el usuario seleccione salir; entonces, finalizará la ejecución de la sesión y del programa. Esto significa que, durante una sola sesión, se pueden jugar varias partidas y las puntuaciones están disponibles (el próximo sprint hará que la clasificación del juego sea persistente).

Es posible que *GameSession* necesite usar algunos de sus campos durante la ejecución:

1. Un objeto ***interactor*** del juego que implementa la interfaz ***GameInteractor***. Se trata de obtener datos de entrada del usuario (siguiente disparo). En esta etapa, hay tres implementaciones: aleatoria, interfaz gráfica de usuario y consola.

2. Un objeto ***presenter*** del juego que implementa la interfaz ***GamePresenter*** y lleva a cabo acciones para mostrar datos al jugador como imprimir un mensaje de bienvenida, mostrar el tablero del juego o informar al usuario cuando el juego ha terminado.

3. Un objeto ***interactor*** de sesión que implementa la interfaz ***SessionInteractor*** y gestiona la **interacción entre el usuario y la sesión actual**, incluyendo solicitar el nombre del usuario, mostrar el menú o ejecutar una opción de menú.

4. Un objeto ***logger*** que implementa la interfaz ***SimpleLogger***. Se ocupa de los estados de error o eventos del sistema **registrando el mensaje de error correspondiente en el log**.

5. Un objeto ***ranking*** de tipo ***GameRanking***. Almacena las puntuaciones obtenidas en los juegos, durante una sesión.

3.3 Jugar una partida

Cuando se selecciona **jugar una partida**, el objeto *GameSession* pedirá al usuario que seleccione el nivel de dificultad y el modo de depuración. Luego, creará los jugadores, pasándoles los parámetros necesarios. A continuación, creará un nuevo objeto *Game* pasándole los parámetros necesarios y establecerá el atributo **gamePresenter** en ***Game*** a ***gamePresenter*** en ***GameSession*** para que pueda interactuar con el usuario.

Finalmente, se ejecuta el método *play()* a través de este objeto *Game* y **se jugará una partida battleship completa como en sprints anteriores**.

Cuando **el juego termina** y el **jugador gana**, se le **pregunta si quiere incluir la puntuación en la clasificación**. Si es así, se crea un objeto *Score* y se añade a *GameRanking* con la siguiente información: nombre de usuario, fecha del juego, tiempo que ha tardado, resultado del juego (victoria o derrota) y nivel de dificultad.

3.4 Manejo de excepciones

Si, durante la ejecución del programa, se lanza una excepción debido a errores de programación o del sistema (***RuntimeException***), el procedimiento será el siguiente:

- *El objeto* ***GameSession*** recogerá la excepción y registrará su mensaje de error en el objeto logger.
- La **ejecución del programa terminará de forma controlada**, es decir, el bucle principal se detendrá, incluso si el usuario no ha seleccionado la opción de salida.
- Se le informará al usuario que ocurrió un error con un mensaje como "**FATAL ERROR:"** seguido del mensaje contenido en la excepción, y la ejecución del programa se detendrá.

3.5 El ranking

GameRanking contendrá una lista de objetos Score, cada uno de los cuales almacena diversa información asociada con un juego terminado. Aunque Score puede representar juegos perdidos, **solo se guardarán los juegos ganados y solo si el usuario está de acuerdo**.

En esta versión, la clasificación se pierde cuando finaliza la sesión, es decir, cada vez que finaliza la ejecución de la aplicación. El siguiente sprint se volverá persistente, guardando los resultados en un fichero y se podrá guardar información sobre diferentes juegos jugados en diferentes sesiones.

4 Implementación del Log

En esta sección, describiremos la interfaz **SimpleLogger** y la clase **ConsoleSimpleLogger** que la implementa. Sin embargo, **ambos elementos completamente implementados se proporcionan junto con este documento**. Por favor, **integre los ficheros** SimpleLogger.java y ConsoleSimpleLogger.java en su proyecto **sin modificarlos** en absoluto. Si aparece algún error de compilación, haga que su código sea compatible.

Por lo general, los programas escriben registros en ficheros de texto que almacenan una amplia gama de información: carga de datos, solicitudes, errores ...

Este sprint **registrará errores sin usar ficheros** (esto es para el próximo sprint). La interfaz de log ofrecerá un único método público y la implementación se registrará **escribiendo** contenido en la **salida estándar de errores**.

4.1 Interfaz SimpleLogger

SimpleLogger contiene un solo método

```
void log(Exception ex)
```
Escribe el mensaje de la excepción (ex.getMessage()).

4.2 Clase ConsoleSimpleLogger

Implementa la interfaz *SimpleLogger*. Imprime el mensaje en la salida estándar de errores ***System.err***

5 Implementación de la clasificación (Ranking)

5.1 Clase Score

Almacena la puntuación (y otra información) de un solo juego. Contiene estos métodos públicos:

```
Score(String userName, GameLevel level, long time)
```
Constructor.
- userName: nombre del usuario que jugó el juego.
- level: nivel de dificultad.
- time: duración del juego.

Nota: el constructor también guardará la **fecha** en la que finalizó el juego. En aras de la simplicidad, en lugar de pasar esta fecha como un parámetro al constructor, simplemente usaremos la fecha del sistema cuando se invoque al constructor. Simplemente se debe ejecutar ***Date date = new Date()*** para almacenar un objeto Date con la fecha actual (*java.util.Date*).

```
String getUserName()
```
Devuelve el valor del atributo userName

```
long getTime()
```

Devuelve el valor del atributo time.

`Date getDate()`
Devuelve el valor del atributo date.

`GameLevel getLevel()`
Devuelve el valor del atributo level.

`String toString()`
Devuelve una representación textual (o cadena) del objeto

5.2 Clase GameRanking

Esta clase almacena una lista de objetos *Score* que representan juegos terminados y ofrece métodos para consultar esa lista. Contiene los siguientes métodos públicos:

`void append(Score score)`
Añade el parámetro al final de la lista de puntuaciones.

`List<Score> getRanking()`
Devuelve una copia de la lista de puntuaciones.

`List<Score> getRankingFor(String userName)`
Devuelve una lista que contiene solo aquellas puntuaciones cuyo nombre de usuario coincide con el parámetro.

6 Implementación de la sesión

6.1 Enum GameLevel

El juego se puede configurar para jugar con diferentes niveles de dificultad: ***SEA***, ***OCEAN***, ***PLANET***. Cada valor determinará un tamaño del tablero: 10x10, 15x15 y 20x20, respectivamente. Estos tres valores estarán representados por una **enumeración** llamada ***GameLevel***

6.2 Interface SessionInteractor

SessionInteractor ofrece métodos para gestionar toda la interacción del usuario que tiene que ver con la sesión (no con el juego en sí).

`GameLevel askGameLevel();`
Le pide al usuario un nivel de dificultad y devuelve la respuesta con un objeto *GameLevel*.

`String askUserName();`
Solicita al usuario un nombre y devuelve una cadena con la respuesta, que no puede ser nula ni vacía.

`int askNextOption();`
Le pide al usuario que elija una opción del menú. Devuelve un número entero que representa la opción elegida. Un valor mayor que cero representará algunas de las acciones disponibles. Un valor cero siempre representará la opción de salida.

`boolean askDebugMode ();`
Le pide al usuario que elija sí o no.

`boolean doYouWantToRegisterYourScore();`
Al final de un juego, le pregunta al usuario si quiere guardar su puntuación. Devuelve verdadero si la respuesta es afirmativa y falsa en caso contrario.

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

Recibe una lista de objetos *Score* que representan todas las puntuaciones registradas en el sistema y muestra toda la información sobre todas ellas (formato tabular, una línea para cada puntuación).

`void showPersonalRanking(List<Score> ranking);`

Recibe una lista de objetos *Score* que representan todas las puntuaciones registradas en el sistema y muestra toda la información sobre todas ellas (formato tabular, una línea para cada puntuación) excepto el nombre de usuario (se entiende que es el usuario almacenado en la sesión).

`void showErrorMessage(String message);`

Muestra el mensaje de error, recibido como parámetro. Este tipo de errores no detienen la ejecución (errores recuperables).

`void showFatalErrorMessage(String message);`

Muestra mensajes de error graves al usuario. Este método debe invocarse para informar al usuario del error irrecuperable y que el programa va a detener su ejecución.

6.3 Clase ConsoleSessionInteractor

La clase ***ConsoleSessionInteractor*** se implementará en battleship.mp.p4.console. Implementa los métodos de interfaz *SessionInteractor* para trabajar con la consola Java.

En modo texto, todos los métodos que **solicitan información** al usuario deben ir precedidos de una pregunta para que el usuario sepa cuáles son sus opciones. Los métodos para **mostrar información** deben generar una cadena con un formato apropiado. Ejemplos:

Método `String askUserName();`

```
Player name?
Dani
```

Método `int askNextOption();`

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

Método `GameLevel askGameLevel();`

```
Option? 1
Level? (S)ea, (O)cean, (P)lanet
P
```

Método `boolean askDebugMode();`

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

Método `void showPersonalRanking`

```
Date      .Hour    .Level .Res .Time
09/03/2020 11:55:51 EASY    won  234
```

Método `showRanking(List<Score> ranking);`

Una cabecera y algunas líneas como las siguientes:

```
   r nane .Date      .Hour    .Level .Res .Time
Dani       09/03/2020 11:55:51 EASY    won  234
```

Método `boolean doYouWantToRegisterYourScore();`

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

6.4 Clase GameSession

La clase *GameSession* con su método público *run()* se encargará de la lógica de las sesiones:

- Al iniciar la sesión, le **pide al jugador un nombre de usuario.**
- Luego, comienza el bucle principal que consiste en desplegar un menú, recoger las opciones del usuario y ejecutarlas.
  - Cuando se selecciona **jugar una partida**, se le pide al usuario que introduzca un **nivel de dificultad** y un **modo de depuración**. Luego, se inicia un juego y, si el usuario así lo decide, **la puntuación se mantiene cuando termina**.
  - También puede **mostrar puntuaciones**.
- Siempre **recoge RuntimeException** y luego imprime un mensaje adecuado y finaliza la ejecución de forma controlada.

Además de *run()*, *GameSession* ofrecerá 4 *setters*:

`void run()`
Ejecuta las acciones descritas anteriormente.

`void setSessionInteractor(SessionInteractor interactor)`
Asigna al atributo *sessionInteractor* el parámetro `interactor`.

`void setGameInteractor(GameInteractor interactor)`
Asigna al atributo *gameInteractor* el parámetro `interactor`.

`void setGamePresenter(GamePresenter presenter)`
Asigna al atributo *gamePresenter* el parámetro `presenter`.

`void setLogger(SimpleLogger logger)`
Asigna al atributo *simpleLogger* en *GameSession* el parámetro `logger`.

`void setGameRanking(GameRanking ranking)`
Asigna al atributo *gameRanking* en *GameSession* el parámetro `ranking`.

La clase GameSession también contendrá tantos métodos privados como sea necesario para producir código limpio y mantenible. Como pauta, una buena implementación del método run() no debe contener más de 7 líneas.

7 Clase Game

A la clase de juego original se debe añadir el siguiente método público

`long getTime()`
Devuelve el tiempo transcurrido entre el inicio y la finalización del juego.

8 Excepciones

8.1 Errores irrecuperables

Los siguientes son posibles errores irrecuperables. Como se explica en el apartado 3.4, todos se gestionarán de la misma forma: imprimiendo un mensaje en consola. Entonces, la aplicación debe terminar de forma controlada.

Error	Message to print and log
Null argument when invoking setter.	Null value when setting <object name>.
Null argument when creating an instance of a class	Null <arg name> when creating a <class name>

*** NOTA:** Usa **SessionInteractor::showFatalErrorMessage (msg)** para advertir al usuario antes de que finalice la aplicación.

9 Test

No es necesario escribir **nuevos tests**. Solo **asegúrate de que todos los tests funcionan correctamente (color verde).**