Proyecto buscaminas. Sprint 4

1 Descripción general de la nueva versión del juego

En esta versión del proyecto, añadiremos tres nuevas características a nuestro buscaminas: **sesiones**, **ranking** y **logging**.

- **Sesión de usuario**: Una sesión de usuario es todo lo que ocurre entre un usuario y el juego desde que el usuario arranca el juego hasta que decide abandonarlo. En esta versión, dentro de una sesión, el usuario, después de **identificarse**, podrá **jugar** una o varias partidas, ver el **ranking del juego** o **salir** del juego.
- **Ranking**: El ranking contendrá **partidas ganadas** y, de cada partida, se almacenará la fecha en la que se ganó, el nivel de dificultad de la partida jugada, el nombre del jugador y el tiempo total de la partida.
- **Logging** o **registro de errores**: Se registrarán de forma muy simple las excepciones lanzadas durante la ejecución del programa.

IMPORTANTE: Junto con el enunciado de este proyecto **se proporción tres ficheros:** `Main.java`, `SimpleLogger.java` y `BasicSimpleLogger.java` que se describen en los apartados 3.1, 4.1 y 4.2 respectivamente. **Reemplaza `Main.java` del proyecto console (se encuentra en el paquete uo.mp.minesweeper) por este nuevo `Main.java`, tal cual está. No lo modifiques, tu código debe ser compatible** e incluye las clases `SimpleLogger.java` y `BasicSimpleLogger.java` en tu proyecto util23

2 Estructura de proyectos

2.1 Proyecto minesweeper_sprint4

El núcleo del buscaminas será este proyecto, con la estructura de paquetes y clases similar a la mostrada en la Imagen 1.

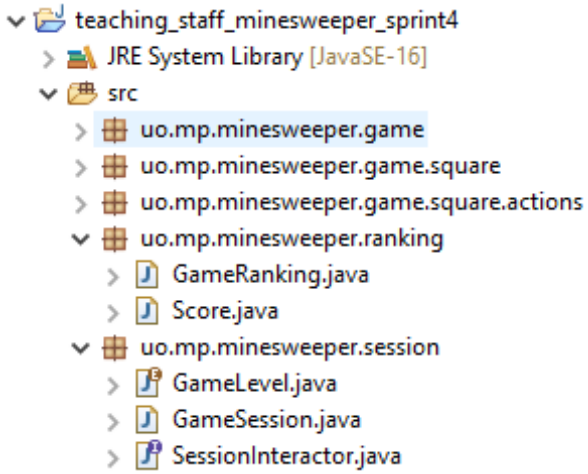

Imagen 1 - Proyecto principal

2.2 Proyecto minesweeper_sprint4_console

Al igual que en la sesión anterior, el proyecto **minesweeper_sprint4_console** contendrá la **implementación de aquellas interfaces** necesarias para interactuar con la aplicación a través de la **consola**: ***`ConsoleSessionInteractor`*** y ***`ConsoleGameInteractor`***.

Contendrá también una clase *`Main`* para lanzar la ejecución del juego utilizando la consola como vía de comunicación con el usuario.

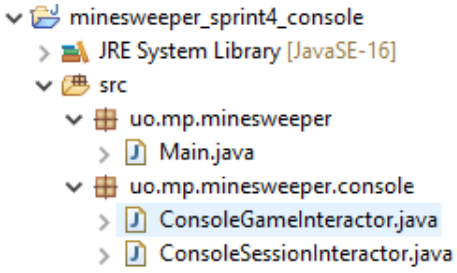

Imagen 2 - Proyecto Console

3 Visión general del sistema a través de su diagrama de clases

El **diagrama UML** de clases mostrado en este apartado incluye relaciones fundamentales entre la mayoría de clases del sistema, pero **no incluye métodos, atributos ni clases de utilidades**. Tampoco pretende imponer que las clases mostradas deban ser las únicas del sistema. **Puedes crear nuevas clases** para encapsular alguna funcionalidad si lo consideras necesario, siempre y cuando las **relaciones y nombres indicados** en este y anteriores enunciados **se respeten**.

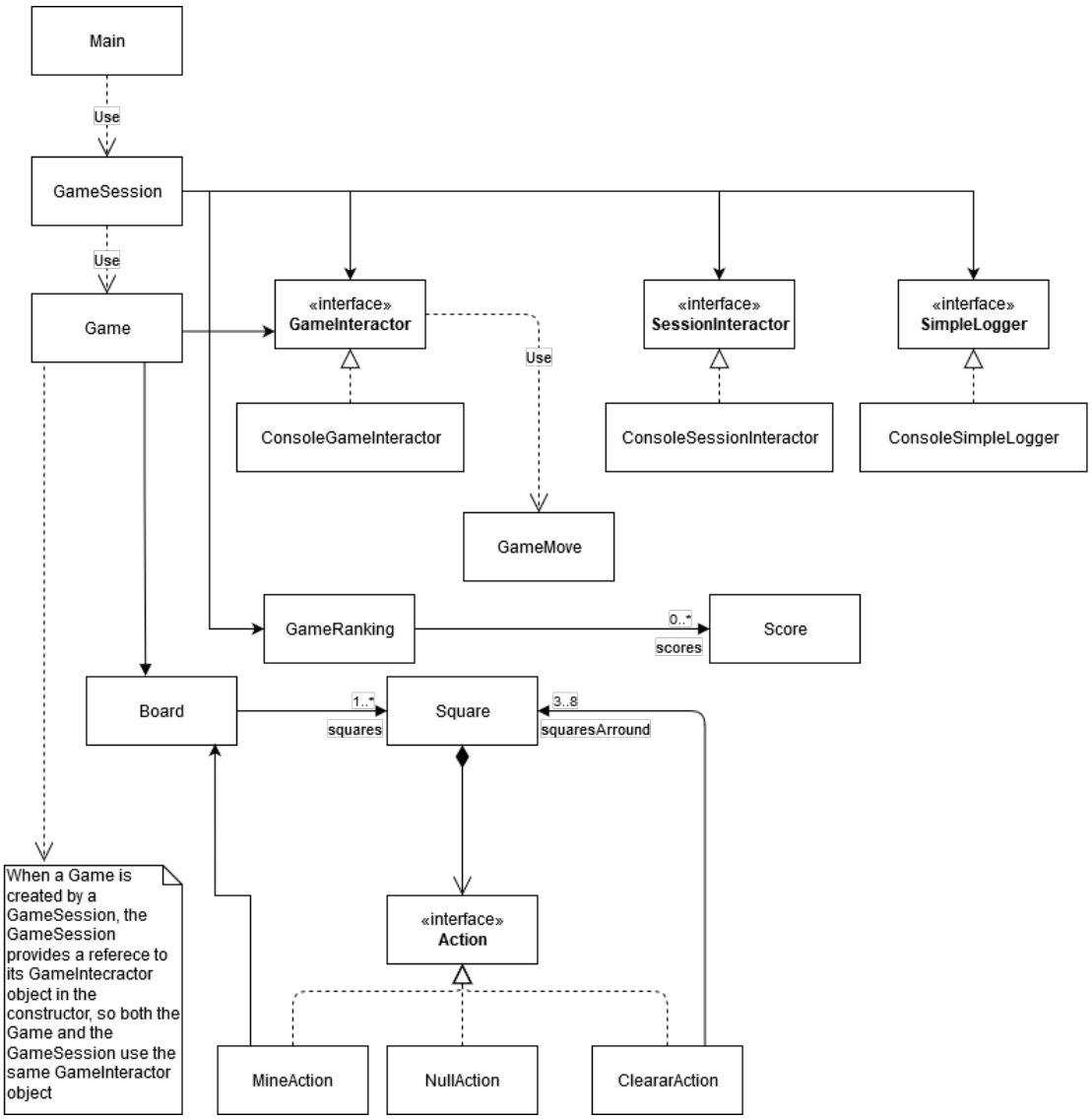

Imagen 3 - Diagrama de clases simplificado

3.1 El nuevo Main.

La nueva clase *Main* proporcionada, que reemplaza a la que se encontraba en el sprint 3, se ocupa igualmente de lanzar el juego. Para ello debe, fundamentalmente, crear un objeto de tipo *GameSession*, inicializarlo correctamente para interaccionar con el usuario y ejecutarlo.

3.2 GameSession

La clase *GameSession* determina el flujo de ejecución de acciones entre usuario y juego. Se requiere que este objeto funcione como sigue:

- **Al iniciar** la aplicación se solicita un **nombre de usuario**.
- Una vez introducidos los datos, se mostrará un **menú** con las siguientes opciones:
  - **Jugar** una partida.
  - Consultar las **puntuaciones de todos** los jugadores.
  - Consultar sus **propias puntuaciones**.
  - **Salir de sesión.**
- Se solicitará al usuario una opción y se ejecutará la opción elegida.

Cada vez que se **termine de procesar una opción** se **volverá a mostrar el menú**, salvo si se escoge salir, en cuyo caso terminará la sesión y la ejecución del programa. Esto quiere decir que en una misma sesión se podrán jugar varias partidas y se podrán consultar las puntuaciones tantas veces como se quiera sin necesidad de ejecutar el programa varias veces.

GameSession necesita 4 objetos para realizar su función:

1. Un objeto ***game interactor*** que implementa la interfaz ***GameInteractor***. Este objeto no varía respecto a la sesión anterior, se ocupa de la interacción (entrada/salida) que tiene que ver con la partida (el juego) en curso (dar la bienvenida a la partida, mostrar el tablero de juego, pedir siguiente movimiento o informar al usuario de que la partida ha terminado sea ganando o perdiendo). Existen dos implementaciones y, dependiendo de cuál se use, la salida será en la consola o en una ventana gráfica y la entrada se leerá desde el teclado o el ratón, respectivamente.

2. Un objeto ***session interactor*** que implementa la interfaz ***SessionInteractor***. Se ocupará de la interacción entre el **usuario** y la **sesión** actual (solicitar nombre del usuario que inicia la sesión, mostrar un menú, ejecutar una opción de menú, etc.).

3. Un objeto ***Logger*** que implementa la interfaz ***SimpleLogger***. Se ocupa de *loggear* estados de error o la ocurrencia de sucesos del sistema registrando el mensaje de error correspondiente en el log.

4. Un objeto ***ranking*** de tipo ***GameRanking***. Almacena las puntuaciones alcanzadas en las partidas.

3.3 Jugar una partida

Cuando el usuario seleccione jugar una partida nueva a través del menú, el objeto *GameSession* le preguntará qué nivel de dificultad desea y, a continuación, creará un nuevo objeto *Game* pasándole los parámetros que sean necesarios. Al **objeto *Game*** creado se le proporcionará la **referencia** al objeto ***GameInteractor*** **almacenado en *GameSession*** para que pueda interactuar con el usuario.

Una vez construido el objeto *Game*, se ejecutará su método *play()* y **se desarrollará una partida de buscaminas completa como en los sprints anteriores**.

Cuando **termine la ejecución de la partida**, en caso de victoria, se le **preguntará al usuario si desea guardar su puntuación**. En caso afirmativo, se crea un objeto *Score* que se almacena en la lista de *GameRanking* con la siguiente información: **nombre** del usuario que la jugó, **fecha** de la partida, **tiempo** que duró la partida, **resultado** de la partida (**victoria** o derrota) y **nivel de dificultad**.

3.4 Tratamiento de Errores

3.4.1 Errores de usuario/aplicación/lógicos

Se deben considerar los siguientes errores:

- Añadir un tamaño de tablero no permitido.
- El usuario se identifica con un nombre inválido.
- El nombre de usuario está repetido en el ranking.
- Pasar una casilla de estado abierto a cerrado.
- Pasar una casilla de estado abierto a abierto.
- Pasar una casilla de estado cerrado a cerrado.
- Pasar una casilla de estado flagged a flagged.

Estas situaciones elevarán una `GameException`.

3.4.2 Errores de Programación/Sistema

- Lista `null` o vacía para crear tablero.
- Una lista que contiene al menos una casilla inválida para crear tablero.
- Obtener un valor `null` al invocar el método `toString()` de la clase `Square`.
- Añadir un jugador `null` o un interactor `null`.

Estas situaciones elevarán una excepción `IllegalArgumentException` o una `IllegalStateException` y debe ser manejadas como se describe en la próxima sección 3.5

3.4.3 Errores de interacción del usuario

Si el usuario escribe una opción diferente a cualquiera de los caracteres permitidos en las distintas opciones del menú (mayúsculas o minúsculas), elevará una `UserInteractionException`.

3.5 Manejo de excepciones

1. `UserInteractionException`. Se ignora la entrada y se maneja la excepción informando al usuario a través de un mensaje en la pantalla y se puede volver a intentar la operación. Finalmente, se registrará su mensaje de error en el objeto `logger`.

2. `GameException. Se informa al usuario mostrando un mensaje en la pantalla y se volverá` a mostrar el menú para que el usuario pueda corregir el error y ejecutar nuevamente el juego.

3. `IllegalArgumentException` e `IllegalStateException`. Ambos indican un error de programación y, por lo tanto, ambos deben informar al usuario sobre la existencia de un error interno irrecuperable, se debe escribir la información en un fichero de registro (log) y detener la ejecución de forma controlada.

En caso de que, durante la ejecución del programa, se lance una excepción por errores de programación o de sistema, se actuará como sigue:

- El objeto ***GameSession*** capturará la excepción y registrará su mensaje de error en el objeto logger.
- Se **terminará** la **ejecución** del programa **de forma ordenada**. Es decir, se detendrá el bucle de procesar opciones, aunque el usuario no haya seleccionado la opción salir.
- Se informará al usuario de lo sucedido con un mensaje como "FATAL ERROR:" seguido por el mensaje contenido en la excepción.

3.6 El ranking

GameRanking contendrá una lista de objetos *Score*, cada uno de los cuales almacena diversa información asociada a una partida finalizada. A pesar de que *Score* puede representar partidas perdidas, **solo** se dará la opción de almacenar datos de **partidas ganadas**. Además, cada partida **se almacenará únicamente** si **el usuario**, al terminarla, indica que **quiere que su puntuación se almacene**.

En esta versión, el ranking es volátil, es decir, se pierde cada vez que termina la ejecución de la aplicación. En el próximo sprint 5 se hará persistente, guardando los resultados en un fichero que podrá mantener información de distintas partidas jugadas en diversas sesiones.

4 Implementación de Logging

En esta sección te describimos la interfaz **`SimpleLogger`** y la clase **`BasicSimpleLogger`** que la implementa. **No debes de modificar el código** de `SimpleLogger` ni de `BasicSimpleLogger`, tu código deberá ser compatible.

Cuando un programa produce logs suele guardar contenido en ficheros de texto. Los logs pueden almacenar información muy diversa: carga de datos, peticiones recibidas, errores…

En nuestro buscaminas, por el momento, vamos a hacer acciones de **log** muy sencillas y **sin guardar contenido en ficheros**. Utilizaremos una interfaz de log con un único método público y una clase que implementa esa interfaz **escribiendo** contenido en **la salida estándar de error**.

4.1 Interfaz SimpleLogger

SimpleLogger ofrece un único método público

```
void log(Exception ex)
```

Loguea (registra) información acerca de la excepción.

4.2 Clase BasicSimpleLogger

Implementa la interfaz *SimpleLogger*. Muestra el contenido a loguear en la salida de error estándar, es decir, ejecutando sentencias ***System.err.println().***

5 Implementación del ranking

5.1 Clase GameRankingEntry

Almacena la puntuación de una única partida. Contiene los siguientes métodos públicos:

```
GameRankingEntry(String userName, GameLevel level, long duration, boolean hasWon)
```

Constructor.

- `userName`: nombre del usuario que jugó la partida.
- `level`: nivel de dificultad.
- `duration`: tiempo de duración de partida
- `hasWon`: true si se ganó la partida, false en caso contrario.

IMPORTANTE: el constructor además guardará la **fecha** en la que terminó la partida. Por simplificar, en lugar de pasar esta fecha por parámetro del constructor, usaremos simplemente la fecha del sistema en el momento en que se llama al constructor: ejecutad ***`Date date = new Date()`*** para almacenar un objeto `Date` con la fecha actual (*java.util.Date*).

`String getUserName()`
Devuelve el valor de userName.

`long getDuration()`
Devuelve el valor de duration.

`boolean hasWon()`
Devuelve el valor de hasWon.

`Date getDate()`
Devuelve el valor de Date.

`GameLevel getLevel()`
Devuelve el valor de level.

`String toString()`
Devuelve una representación textual del contenido del objeto.

5.2 Clase GameRanking

Esta clase almacenará una lista de objetos *`GameRankingEntry`* representando partidas finalizadas y ofrecerá métodos para consultar dicha lista. Sus métodos públicos son los siguientes:

`void append(GameRankingEntry gameRankingEntry)`
Añade el objeto `GameRankingEntry` al final de la lista de `gameRankingEntries`.

`List< GameRankingEntry > getAllEntries()`
Devuelve una copia de la lista de `gameRankingEntries`.

`List< GameRankingEntry > getEntriesForUsername (String userName)`
Devuelve una lista que contiene solo aquellos `gameRankingEntries` cuyo usuario coincide con el *`userName`* recibido como parámetro.

6 Implementación de sesión

6.1 Enum GameLevel

El buscaminas podrá ser configurado para ser jugado con distintos niveles de dificultad: ***`EASY`***, ***`MEDIUM`***, ***`HARD`***. Estos 3 valores se representarán mediante un enumerado llamado ***`GameLevel`***.

6.2 Interfaz SessionInteractor

`SessionInteractor` ofrece métodos para llevar a cabo toda la interacción con el usuario que tiene que ver con la sesión (no con el juego en sí).

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

`String askUserName();`

Solicita al usuario su nombre y devuelve un *String* con la respuesta. El nombre que devuelve no puede ser vacío.

```
int askNextOption();
```
Solicita al usuario que introduzca una opción del menú de sesión. Devuelve un entero que representa la opción escogida de entre las posibles. Un valor mayor que cero representará alguna de las acciones disponibles. El valor cero representará siempre la opción salir.

```
boolean doYouWantToRegisterYourScore();
```
Al finalizar una partida, pregunta al usuario si quiere guardar su puntuación. Devuelve true si la respuesta es afirmativa, y false en caso contrario.

```
void showRanking(List<GameRankingEntry> ranking);
```
Recibe una lista de objetos `GameRankingEntry` representando todas las puntuaciones registradas en el sistema. Se muestra con su información completa (formato tabular, una línea por cada `GameRankingEntry`).

```
void showPersonalRanking(List<GameRankingEntry> ranking);
```
Recibe una lista de objetos `GameRankingEntry` representando todas las puntuaciones registradas en el sistema. Se muestra con su información completa (formato tabular, una línea por cada `GameRankingEntry`). Omite la información de quién es el usuario asociado a cada partida (se sobreentiende que es el usuario almacenado en la sesión).

```
void showGoodBye();
```
Muestra al usuario un mensaje de despedida cuando escoge finalizar la sesión.

```
void showErrorMessage(String message);
```
Comunica un mensaje de error al usuario. El mensaje a mostrar se recibe como parámetro.

```
void showFatalErrorMessage(String message);
```
Comunica mensajes de error graves al usuario. Este método ha de usarse para informar de que el error no puede ser solucionado y el programa finalizará su ejecución de forma inmediata.

6.3 Clase ConsoleSessionInteractor

La clase ***ConsoleSessionInteractor*** formará parte de **minesweeper_sprint4.console**. Implementa todos los métodos de la interfaz *SessionInteractor* para que funcionen con la consola de Java.

En modo texto, todos los métodos que solicitan cierta información al usuario deben ir precedidos de la pregunta pertinente para que el usuario comprenda cuáles son sus opciones. Los métodos cuya misión principal es la de mostrar información en lugar de solicitarla, deben generar texto con un formato adecuado. Ejemplos:

```
String askUserName();
```

```
Player name?
Dani
```

```
int askNextOption();
```

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

```
GameLevel askGameLevel();
```

```
Level? (e)asy, (m)edium, (h)igh
e
```

```
void showPersonalRanking();
```

Se espera que produzca una cabecera y líneas de información similar a las siguientes:

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

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

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

```
void showGoodBye();
```

```
Thanks for your session. Bye, bye!
```

```
boolean doYouWantToRegisterYourScore();
```

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

6.4 Clase GameSession

La clase *GameSession* será la encargada de llevar el peso de toda la lógica de sesión y de lanzar el juego cuando sea necesario. El método público principal de *GameSession* es *run()* que se encarga de lanzar toda la lógica de la sesión:

- **Pregunta el nombre** del usuario al iniciar sesión.
- Inicia el **bucle del menú** principal para recoger procesar las órdenes de usuario.
    a. Cuando el usuario selecciona j**ugar una partida** le pregunta el **nivel de dificultad** y la Inicia. Al finalizarla **conserva** sus **resultados** si el usuario así lo indica.
    b. **Muestra puntuaciones** al usuario si este lo solicita.
- **Captura** posibles ***RuntimeException*** para producir un **mensaje** adecuado para el usuario y **terminar la ejecución** de forma ordenada.

Además del *run()*, *GameSession* ofrecerá 4 *setters* públicos. Este es su catálogo completo de métodos públicos:

`void run()`
: Inicia toda la lógica principal de *GameSession*.

`void setSessionInteractor(SessionInteractor interactor)`
: Provoca que *GameSession* utilice el objeto *SessionInteractor* recibido como parámetro.

`void setGameInteractor(GameInteractor interactor)`
: Provoca que *GameSession* utilice el objeto *GameInteractor* recibido como parámetro.

`void setLogger(SimpleLogger logger)`
: Provoca que *GameSession* utilice el objeto *SimpleLogger* recibido como parámetro.

`void setGameRanking(GameRanking ranking)`
: Almacena como ranking el objeto recibido como parámetro.

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

7 Clase game

La clase Game original debe sufrir alguna modificación, además es necesario añadirle el método:

```
long getDuration ()
```

Devuelve el tiempo que duró la partida.

8 Test

No será necesario desarrollar **nuevos test** para este sprint. Asegúrate de que los **test implementados** anteriormente **siguen funcionando**.