Battleship sprint 2

1 Descripción general

Sprint 2 debe mejorar sprint 1 proporcionando soporte a las siguientes características:

- Mientras que en sprint 1 los jugadores realizaban turnos alternativos de disparos, en este nuevo sprint, los jugadores permanecen en el turno disparando mientras logren disparar sobre barcos del oponente.
- En sprint 2 se deberán colocar los barcos en el tablero en posiciones aleatorias. Es decir, los barcos se colocarán en posiciones aleatorias, no fijas y diferentes para cada flota.
- A partir de sprint 2, cuando un barco es hundido, la aplicación debe mostrar "Hit and Sunk" en vez de únicamente "Hit". Además, la forma de visualizar los barcos cambiará.

Además de estos cambios de funcionalidad, en sprint 2 se realizará una refactorización de código importante, para evitar la repetición de código y proporcionar una implementación más clara.

En primer lugar, importa el proyecto sprint 1, realiza una copia y cambia su nombre por

`nombre_apellido1_apellido2_battleship_sprint2`

A partir de aquí comienza a realizar los cambios que se proponen en Sprint 2. Mantendrás así **dos proyectos** sprint 1 y el nuevo proyecto sprint 2

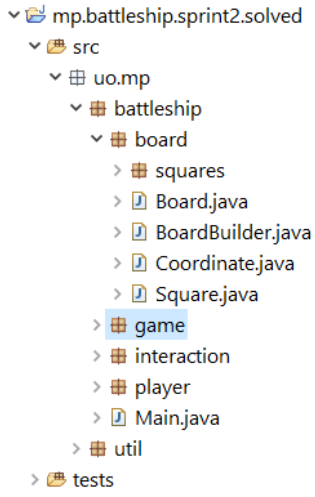
2 Diagrama UML

Dibuja un diagrama de clases para representar las clases principales del juego, sus atributos, métodos públicos y relaciones.

- Organizar las cajas en los diagramas para que no haya líneas cruzadas.
- Las líneas son verticales u horizontales (y se pueden combinar a 90º).
- Los nombres de las clases y métodos siguen las convenciones de UML (los mismos que en Java).

3 Nuevas clases

Dado que se añadirán nuevas clases, esta será la estructura de paquetes que debe tener el proyecto:



3.1 Clase Square

Esta clase contiene información de cada casilla en la que se divide el tablero, en lugar de enteros, como había en sprint 1. **Habrá tantos objetos `Square` como celdas o casillas haya en el array que implementa el tablero y cada una necesariamente contendrá una referencia a un objeto `Ship` o a un objeto Water,** ambas descritas más adelante en esta misma sección. El contenido de la casilla depende de cómo se construya el tablero en la clase `BoardBuilder`. Ten en cuenta que en total habrá 10 barcos (10 instancias de Ship. Si el barco es, por ejemplo, un crucero (que ocupa 3 posiciones del tablero) esas posiciones de la matriz tendrán una referencia al mismo barco. Por otro lado, hay tantas instancias de agua como casillas vacías haya (las que no tienen referencia a un barco).

Métodos públicos:

public int shootAt()

Marca esta casilla como disparada y propaga el disparo al barco o al agua cuya referencia se almacena en la casilla. Devuelve un entero, dependiendo del daño causado por el disparo:

- 0: no se ha producido daño alguno; no hay barco ocupando esa casilla en el tablero del oponente, sino agua.
- 1: Daño severo. Esta casilla tiene una referencia a un barco que no tiene todas sus casillas disparadas todavía.
- 2: Daño masivo. La casilla tiene una referencia a un barco, y ésta fue la última casilla del barco que aún no había sido disparada (el barco queda hundido).

public boolean isShot ()

Si esta casilla ha sido disparada, devuelve true, en otro caso devuelve false.

public boolean hasImpact ()

Devuelve true cuando el objeto contenido en la casilla es un barco y ya ha sido disparado (o incluso hundido) y false en otro caso. La clase `Square` **delega la implementación en el objeto contenido en** la casilla. Remítete a las clases `Ship` y `Water` para más detalles. Nótese que este método es diferente del anterior.

public char toChar ()

Devuelve el carácter correspondiente al contenido delegando esta operación en el objeto contenido en la casilla y dependiendo de si la casilla está o no disparada para que se muestre el carácter correspondiente de la tabla 1.

public Square setContent (objeto ship o Water**)**

Este método guarda en el atributo content de la casilla, el objeto recibido como parámetro. Podría ser un objeto `Ship` o `Water`.

public boolean hasContent ()

Devuelve true si el contenido de esta casilla está asignado a un objeto `Ship` o `Water`, y false en otro caso.

La siguiente tabla muestra los caracteres para visualizar los tableros.

Displayed square	Description
BBBB	Barco de Batalla (cada casilla una **B**)
CCC	Crucero (cada casilla una **C**)
DD	Destructor (cada casilla una **D**)
S	Submarino (cada casilla una **S**)
*	Tocado, golpeado (Hit)
#	Cuando el barco es hundido, el * se reemplaza por #
·	Disparo perdido (agua) (\u00B7, punto)
Blank space	En el tablero de la izquierda, casillas que contienen agua; en el tablero de la derecha aquellas que no han sido disparadas todavía.

*Tabla 1: Caracteres para **visualizar** el contenido de los tableros*

3.2 Clase Ship

Esta clase representa cualquier barco que pueda ser colocado en el tablero por la aplicación. Habrá tantas instancias como barcos haya en la flota. Un total de 10: uno de batalla, dos cruceros, tres destructores, y cuatro submarinos. Debe contener varios atributos para representar el barco y los siguientes métodos públicos a implementar:

Constructor Ship (int shipSize)

Crea un nuevo objeto `Ship`; cada barco debe contener, al menos, un campo **`size`** (recuerda la descripción de `ship` en sprint 1).

public int shootAt ()

El barco es golpeado por un disparo del oponente y devuelve el daño causado, como ha sido descrito en la clase `Square`.

public boolean hasImpact ()

Devuelve true si el barco ha sido tocado al menos una vez; false en caso contrario.

public int size ()

Devuelve el tamaño de este barco; es decir el número de casillas que ocupa.

public boolean isSunk ()

Devuelve true si todas las posiciones del barco han sido tocadas (golpeadas), es decir, si el barco está hundido; false en caso contrario.

public char toChar ()

Devuelve el carácter correspondiente al barco de acuerdo con la tabla 1 de arriba. Por ejemplo: Barco de Batalla (una **B**)

public char toFiredChar ()

Devuelve el carácter correspondiente a al barco ya disparado de acuerdo con la tabla 1 de arriba. Por ejemplo: un * si está tocado o un # si está hundido

3.3 Clase Water

Este nuevo tipo representa una pieza del tablero donde no hay barco, es decir, todos los objetos `Square` que no contienen un objeto `Ship` deben contener un objeto `Water`.

Métodos públicos:

public int shootAt ()

Como no es un barco, no se refiere al daño. Por tanto devuelve 0.

public boolean hasImpact ()

Como no es un barco, **siempre** devuelve false.

public char toChar ()

Devuelve el carácter asociado con una celda `Water`, de acuerdo con la tabla 1 de arriba (es decir, un blanco).

public char toFiredChar ()

Devuelve el carácter asociado con un disparo perdido, de acuerdo con la tabla 1 de arriba (es decir un ".").

4 Refactorización de clases

En esta sección, describiremos modificaciones hechas en clases existentes. Aquellas clases o métodos que permanecen iguales que en el sprint 1 no son mencionados.

4.1 Refactorización de las clases Ship y Water

Como ambas clases comparten solo un conjunto común de métodos, sería buena idea diseñarlas con una interface común, **`Target.`**

4.2 Refactorización de la clase Board

Reescribe la clase `Board` y **reemplaza el array de enteros por un array de `Square`**. Esto ciertamente provoca **cambios en la implementación** de sus métodos, pero **no en la interfaz de la clase**: constructores y métodos públicos no cambian (excepto para el método **`shootAt`** que devolverá el daño causado y no solo un valor true/false como hacía el sprint 1).

Métodos públicos:

Constructor Board (int size)

Construye un array de objetos **`Square`** de tamaño size. Además, crea una **flota** de diez barcos (uno de batalla, dos cruceros, tres destructores y cuatro submarinos). Finalmente, estos barcos serán colocados por la clase `BoardBuilder.`
Cuando se invoque el método build de `BoardBuilder` deberían haberse invocado previamente los métodos of(..) y forFleet(..). En otro caso, se debe lanzar una excepción `IllegalStateException`. Revisa `StateChecks.java`

`int shootAt (Coordinate coordinates)`

Almacena un disparo en esta casilla y devuelve un entero como se describió en la clase `Square`.

`Void setSquaresForTest (Square[][] arg)`

Asigna el array de Squares recibido como parámetro. Únicamente para utilizarlo en los tests, por lo que debe ser protegido.

4.3 Refactorización de la clase BoardBuilder

Dado que el array construido por `BoardBuilder` cambia, la implementación de la clase en si misma necesita también ser revisada.

Métodos públicos:

BoardBuilder forFleet (List<Ship> fleet)

Guarda la flota fleet que se recibe en la clase BoardBuilder. Este método debe ser invocado antes del método build.

Square[][] build( )

Reescribe el método para devolver un array de `Square` (no de int). Este método debe

- Crear un array de objetos `Square`
- Colocar una flota de barcos en el tablero en coordenadas aleatorias o fijas (según la opción elegida por el alumno) y que se describe en la siguiente sección.
- Rellenar el resto del tablero con instancias de la clase `Water`.
- *size* debería estar ahora en el rango [10,15].

4.4 Regla para colocar barcos en el tablero

Opción A

Los barcos se colocan de manera fija con en Sprint 1

Opción B

El método `build()` debe ser modificado para seleccionar posiciones aleatorias para los barcos.

En ambas opciones, se deben cumplir **todas las reglas para colocar los barcos que se describen a continuación**:

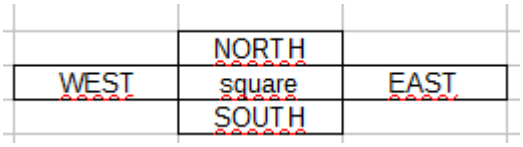
- Cada barco ocupa varias cuadrículas consecutivas en el tablero. Los barcos pueden ser colocados **horizontalmente** o **verticalmente** (nunca en diagonal) dentro del tablero.
- Los barcos no pueden ser colocados uno a continuación del otro.
- No coloques un barco de forma que cualquier parte de él solape el borde del tablero o de otro barco. Es decir, todo el borde del tablero estará libre de barcos y dos barcos no pueden estar pegados (en cuadrículas adyacentes ni tampoco pueden compartir cuadrícula (uno en horizontal y otro en vertical)
- **No se permite cambiar la posición** de los barcos una vez que el juego haya comenzado.

4.5 Refactorización de la clase Coordinate

Se añadirán nuevos métodos públicos:

Coordinate go (int direction)

Devuelve un objeto `Coordinate`, adyacente a la coordenada que invoca al método, en la dirección recibida como parámetro, donde 0 significa **Norte**, 1 significa **Este**, 2 significa **Sur** y 3 significa **Oeste**. Nótese que este método **siempre devolverá un objeto `Coordinate` válido** por la forma en la que se han colocado los barcos.



4.6 Refactorización de las clases HumanPlayer y ComputerPlayer

Refactoriza ambas clases, HumanPlayer y ComputerPlayer generalizando en un tipo común `Player`.

Aparte de esta importante modificación, cambia lo siguiente;

Métodos públicos:

`int shootAt (Coordinate coordinates)`

Como ya se ha descrito anteriormente, shootAt devolverá un entero con el nivel de daño sufrido.

`Coordinate makeChoice ()`

No se permiten disparos repetidos. Esto afecta solo a la clase HumanPlayer. ComputerPlayer nunca genera la misma coordenada dos veces. Es decir, cuando el usuario dispara a una casilla ya disparada, se deberá ignorar la entrada proporcionada por el usuario, y el jugador debe seguir introduciendo nuevas coordenadas hasta que dispare sobre una casilla aún no disparada. (Se puede usar el método `public boolean contains (Object element )` para buscar si un objeto coordenada se encuentra en la lista de posiciones ya disparadas).
Con esta implementación ambos jugadores se comportan de la misma manera.

4.7 Refactorización de la clase TurnSelector

Como en el sprint 1, realiza la gestión del turno para jugar. La primera vez siempre era el turno del Usuario, sin embargo, en este segundo sprint, cuando un jugador toca (golpea) un barco del oponente, el siguiente turno será para el mismo jugador de nuevo.

Métodos públicos:

Constructor TurnSelector (Player user, Player computer)

Modifica el constructor para recibir jugadores como argumentos.

Player next()

Devuelve el siguiente jugador que posee el turno.

En circunstancias normales será el oponente. Sin embargo, si el jugador ha disparado sobre un barco, el turno sigue siendo para el mismo jugado. En este caso se habrá llamado al método repeat

void repeat()

Se establece que el turno debe ser repetido.

4.8 Refactorización de la clase Game

El comportamiento básico del juego permanece igual. Las únicas diferencias serán:

- Cuando un jugador golpea (o toca) un barco del oponente, **repiten turno**.
- Cuando un jugador **golpea un barco**, la aplicación imprime **Hit!** Sin embargo, si el jugador **hunde un barco**, la aplicación imprime **Hit and Sunk!**
- Si no se golpea ningún barco, únicamente imprime **Miss!** como en sprint 1.

- Cuando a uno de los jugadores no le quedan barcos (es decir, todos los barcos del jugador han sido hundidos), el juego se termina.

5 Diagrama de clases

Tras leer las secciones anteriores y antes de empezar a implementar, dibuja un diagrama de clases UML que modele las relaciones entre las clases del proyecto: Main, Game, Board, Player, HumanPlayer, ComputerPlayer, Square, Ship, Water and Coordinate. No es necesario incluir ninguna otra clase. No olvide la multiplicidad.

Para cada clase, muestra, además de las relaciones, sus atributos y sus métodos públicos (no incluya aquellos atributos que estén incluidos en las relaciones representadas).

Respecto al dibujo:

- Asegúrese de que no se cruzan líneas.
- Las líneas podrán ser horizontales, verticales o combiner ambas doblándose siempre en ángulos de 90 grados.

6 Test

Revisa los test implementados en el sprint anterior. Deben seguir funcionando.

6.1 Clase Coordinate

Método de Test go (int direction)

Comprueba los objetos `Coordinate` devueltos por este método.

Casos de uso:

- Comprueba un `Coordinate` para columna A, dirección Oeste
- Comprueba un `Coordinate` para fila 0, dirección Norte
- Comprueba todas las direcciones con un `Coordinate` en fila y columna diferente a 0

6.2 Clase TurnSelector

Test method next ()

Comprueba que dos llamadas consecutivas a `next()`, devolverán jugadores alternados.

Test method repeat ()

Comprueba que, después de ejecutar `repeat()`, `next()` devolverá el mismo jugador de nuevo.

6.3 Clase Square

Test method shootAt (Coordinate coordinate)

Disparo a agua, devuelve 0

Disparo a un submarino, devuelve 2 (porque lo hunde)

Disparo a una posición de un destructor, devuelve 1

Disparo a la última posición del destructor, que lo hunde, devuelve 2

Nota. Utiliza el código del fichero `launuchFleet.txt` para escribir los test