Bataille Navale

A la « bataille navale »[1] (*battleship game*), le plateau de jeu est représenté par une grille rectangulaire de cases sur lesquelles on peut poser des bateaux. Les bateaux sont larges d'une case et longs d'un nombre variable de cases. Ils peuvent être posés verticalement ou horizontalement sur le plateau.
Le plateau est masqué au joueur qui « attaque » et celui-ci doit couler tous les bateaux du joueur « défenseur » (*defender*). Pour cela, il propose une position du plateau qui désigne la case sur laquelle il « tire ». Plusieurs cas sont alors possibles :

- si cette case n'est pas occupée par un bateau, le défenseur annonce *« dans l'eau »* (*missed*) ;
- dans le cas contraire,
  - le défenseur annonce « *touché* » (*hit*) si toutes les cases occupées par le bateau touché n'ont pas déjà été visées par l'attaquant,
  - le défenseur annonce « *coulé* » (*sunk*) si toutes les autres cases du bateau ont déjà été touchées.
- lorsqu'une case avait déjà été visée précédemment, la réponse est toujours « *dans l'eau* ».

On s'intéresse à certains aspects de la programmation en JAVA d'un ensemble de classes permettant la programmation de ce jeu. Les classes sont placées dans le paquetage `battleship`.

Les bateaux. La classe `Ship` permet de représenter les bateaux. Un objet bateau est défini par sa longueur. La longueur d'un bateau détermine son nombre de « points de vie » (*life points*) et est fixée à la création. Lorsqu'il est touché (méthode `beenHitting()`), ce nombre diminue. Un bateau est coulé quand son nombre de points de vie arrive à 0 (la méthode `hasBeenSunk()` permet de savoir si c'est le cas ou non).
Voici le diagramme de cette classe :

battleship::Ship
- lifePoints : int
+ Ship(length : int) + hasBeenSunk() : boolean + beenHitting() + getLifePoints() : int + toString() : String

Dans l'eau, touché, coulé. On appelle `Answer` le type permettant de représenter les 3 réponses possibles après une proposition d'un attaquant.

Q 1 . Comment proposez-vous de définir le type `Answer`.

Le plateau de jeu.
La classe représentant le plateau de jeu (« la mer ») s'appelle `Sea`. On décide de représenter son état par un tableau à 2 dimensions de « cases » qui sont des objets de type `Cell`.

La classe `Cell`.
La classe `Cell` permet de représenter les cases. Son diagramme de classe est donné ci-dessous.
Une case est initialement vide. Mais elle peut être occupée par un seul bateau (méthode `getShip()`). Il faut donc pouvoir poser un bateau sur une case (méthode `setShip()`).
L'attaquant peut « tirer » sur une case : méthode `shot()`. Si la case est occupée, lors du premier de ces tirs, cela a pour conséquence que le bateau correspondant est touché. Seul le premier tir sur une case compte, un bateau ne peut pas être touché deux fois par un tir sur la même case. La réponse fournie indique le résultat du tir.
Il est donc nécessaire de pouvoir savoir si une case a déjà été visée ou non par un tir de l'attaquant (peu importe qu'elle comporte initialement un bateau ou non). C'est ce que permet la méthode `hasBeenShot()`.

[1] Voir `https://fr.wikipedia.org/wiki/Bataille_navale_(jeu)`.

battleship::Cell
- ship : Ship - shot : boolean
+ Cell() + getShip() : Ship + setShip(ship : Ship) + hasBeenShot() : boolean + shot() : Answer

Q 2 . Comment proposez-vous de gérer la présence ou l'absence d'un bateau sur une case.

Q 3 . Donnez la ou les méthodes de tests qui permettent de vérifier la spécification suivante du comportement de la méthode `shot()` : « *L'attaquant peut « tirer » sur une case. Si la case est occupée, lors du premier de ces tirs, cela a pour conséquence que son bateau est touché. Seul le premier tir sur une case compte, un bateau ne peut pas être touché deux fois par un tir sur la même case. La réponse fournie indique le résultat du tir.* »

Q 4 . Donnez un code pour cette méthode `shot()`.

Positions.
Pour représenter les coordonnées des cases du plateau de jeu, on définit la classe `Position` dans le paquetage `battleship.util`. Voici son diagramme de la classe :

battleship::util::Position
- x : int - y : int
+ Position(x : int, y : int) + getX() : int + getY() : int + equals(o : Object) : boolean + toString() : String

La classe `Sea`.

Q 5 . Donnez le code définissant l'entête de déclaration de la classe `Sea`, ses attributs `Sea` ainsi que le constructeur sachant qu'initialement toutes les cases sont vides (pas de bateau) et que les dimensions du plateau de jeu sont fixées à la construction de l'objet.

On suppose que la classe `Sea` dispose d'une méthode :

```
public void addShip(Ship s, Position p)
```

qui permet d'ajouter le bateau `s` à la position `p` (en supposant la case à la position `p` vide). Cette méthode n'ajoute « qu'une unité » du bateau `s`. Pour respecter les règles du jeu, les « autres unités » du bateau devront être posées, horizontalement ou verticalement, sur les cases voisines de celle à la position `p`.

Q 6 . La méthode `shoot` de la classe `Sea` est utilisée lorsque le joueur attaquant vise une position `p`. Son résultat est la réponse, de type `Answer`, lorsque l'attaquant vise la case située à la position `p` sur le plateau de jeu. La position `p` visée est donc passée en paramètre de la méthode.

Si la position fournie n'est pas valide, c'est-à-dire hors des limites du plateau de jeu, une exception `InvalidShootException` est levée par la méthode. Cette exception est supposée définie dans le paquetage `battleship`.

Donnez la signature, la javadoc, les méthodes de test et le code de la méthode `shoot()` de `Sea`.