Bataille Navale

A la "bataille navale", le plateau de jeu est représenté par une grille rectangulaire sur laquelle 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". Pour cela, il propose une position du plateau. Si cette position n'est pas occupée par un bateau, le défenseur annonce *"dans l'eau"*, dans le cas contraire il annonce "*touché*" si toutes les cases occupées par le bateau touché n'ont pas déjà été visées par l'attaquant, et "*coulé*" si toutes les autres cases du bateau ont déjà été touchées. Lorsqu'une case est visée pour la seconde fois, la réponse est toujours "*dans l'eau*".

On s'intéresse à la programmation en JAVA d'un ensemble de classes permettant la programmation de ce jeu. Les classes sont à placer dans le paquetage `naval`.
Une archive de test de l'application à réaliser est fournie sur le portail.

Les bateaux. Un objet bateau est défini par sa longueur. La longueur d'un bateau détermine son nombre de "points de vie" et est fixée à la création. A chaque fois qu'il est touché, sur une case jamais touchée auparavant bien sûr, ce nombre diminue et un bateau est coulé quand son nombre de points de vie arrive à 0.

Q 1 . Donnez un diagramme de classe puis le code java d'une telle classe bateau.

Dans l'eau, touché, coulé. On appelle `Reponse` le type permettant de représenter les 3 réponses possibles après une proposition d'un attaquant : `DANS_LEAU`, `TOUCHE` et `COULE`.

Q 2 . Définissez le type `Reponse`.

Le plateau de jeu. La classe représentant le plateau s'appelle `Mer`. On décide de représenter son état par un tableau à 2 dimensions d'objets `Case`.
Une case peut être vide ou occupée par un bateau. Le code de la classe `Case` est donné en annexe. L'attribut `bateau` d'une instance de cette classe vaut `null` si aucun bateau n'occupe cette case. Pour une case, l'information `visee` permet de savoir si l'attaquant a déjà visé ou non cette case (peu importe qu'elle comporte initialement un bateau ou non).

Q 3 . Donnez le code définissant les attributs de la classe `Mer` ainsi que le constructeur sachant qu'initialement toutes les cases sont vides (pas de bateau) et que les dimensions du plateau sont fixées à la construction.

Q 4 . Donnez le code de la méthode, de la classe `Mer` :

```
public Reponse vise(Position p)
```

dont le résultat est la réponse lorsque l'attaquant vise la case représentée par la position `p`, supposée valide (c'est-à-dire dans les limites du plateau de jeu). Le diagramme de la classe `Position` est donnée en annexe.

Cette méthode doit gérer l'évolution de l'attribut `visee` de la case à la position `p` ainsi que, le cas échéant, la gestion des points de vie du bateau qui s'y trouve.

Q 5 . Ecrivez le code de la classe `Position`.

Affichage. On s'intéresse maintenant à l'affichage du plateau de jeu. On souhaite un affichage sur la sortie standard. Le plateau est affiché ligne par ligne et case par case. Cet affichage doit être différent selon que l'on est défenseur ou attaquant. On considère que la case de coordonnées (0,0) est la case en haut à gauche (la plus au nord-ouest) du plateau.
Pour le défenseur, le caractère affiché pour une case est `'~'` si la case est vide, `'B'` si elle est occupée par un bateau non touché et `'*'` si le bateau occupant cette case a été touché.
Pour l'attaquant, le caractère affiché est `'.'` pour une case qui n'a jamais été visée, il est `'~'` pour une case visée vide, et `'*'` pour une case occupée par un bateau touché.

Q 6 . Ecrivez le code de la méthode `getCaractere` de la classe `Case`. La signification du paramètre de cette méthode est la même que celui du paramètre de la méthode `affichage` (cf. ci-dessous).

Q 7 . Ecrivez le code JAVA de la méthode `affichage` de la classe `Mer`, documentée ci-dessous.

```
/** affiche le plateau ligne par ligne, case par case en
* proposant un affichage différencié pour le défenseur et
* l'attaquant.  Le paramètre defenseur détermine l'affichage
* à réaliser.
* @param defenseur true si l'affichage est pour le défenseur,
*  false si il est pour l'attaquant.
*/
public void affichage(boolean defenseur) {
   à compléter
}
```

Placement de bateaux. On s'intéresse maintenant au placement des bateaux sur le plateau.
Les bateaux ne peuvent être posés que selon des directions imposées. La méthode permettant de poser un bateau est la méthode `poseBateau` de la classe `Mer` dont la documentation est :

```
/** pose le bateau b à partir de la position p dans la direction d.
*  Le nombre de case occupée est bien sûr déterminé par la longueur
*  du bateau.
* @param b le bateau à poser
* @param p la position de la première case où l'on pose le bateau
* @param d la direction dans laquelle on pose b à partir de p
* @exception IllegalStateException si on ne peut poser b à
*  partir de p dans la direction d parce que b sortirait du plateau
*  ou parce qu'un autre bateau occupe déjà une des cases.
*/
```

Une **direction** correspond à un décalage de plus ou moins 1 dans le sens des `x` ou des `y` pour une position donnée (on rappelle que la case de coordonnées (0,0) est la case en haut à gauche, c'est-à-dire la plus au nord-ouest). On définit pour cela l'interface `Direction` ainsi :

```
public interface Direction {
   /** fournit la position suivante de p dans cette direction
    * @return la position suivante de p dans cette direction
    */
   public Position positionSuivante(Position p);
}
```

Q 8 . Ecrivez le code d'une classe `DirectionNord` qui définit une direction dans laquelle la "position voisine" est au nord de la position donnée (décalage selon les `y`).

Q 9 . Même question avec `DirectionEst` pour définir la direction "vers l'est" (décalage selon les `x`), puis avec `DirectionOuest` et `DirectionSud`.

Q 10 . Donnez le code JAVA de la méthode `poseBateau`. Cette méthode lance une `IllegalStateException` si en posant le bateau case par case à partir de la position de départ, on est amené à sortir des bornes du plateau ou à rencontrer une case déjà occupée.

Une manière de procéder est dans un premier temps de tester si la pose est possible, sans poser le bateau : il faut pour cela vérifier que chacune des cases nécessaires à la pose est vide et que l'on ne sort pas du plateau (on peut le tester en capturant l'exception `ArrayIndexOutOfBoundsException` déclenchée lorsque l'on sort des bornes d'un tableau). Si la pose est impossible on lance l'exception. Si elle est possible, dans un second temps, on pose effectivement le bateau en "modifiant" les cases concernées.

Q 11 . Utilisez la classe `naval.Jeu` fournie pour tester vos classes (en plus des tests que vous avez déjà dû réaliser). Créez ainsi une archive `jar` exécutable ayant cette classe comme "`main`" que vous rendrez sur PROF.

Annexes

La classe `naval.Case`.

```
package naval;
public class Case {
   private Bateau bateau;
   private boolean visee;
   public Case() {
      this.bateau = null;
      this.visee = false;
   }
   public Bateau getBateau() {
      return this.bateau;
   }
   public void setBateau(Bateau bateau) {
      this.bateau = bateau;
   }
   public boolean aEteVisee() {
      return this.visee;
   }
   public void visee() {
      this.visee = true;
   }
   public char getCaractere(boolean defenseur) {
      ... voir question 6
   }
}
```

Diagramme de la classe `naval.Position`

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