1 Description du jeu de bataille navale

À la bataille navale (*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.

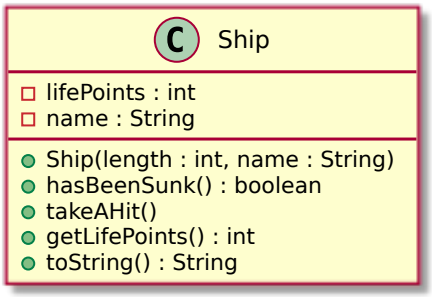
On s'intéresse à certains aspects de la programmation en Java d'un ensemble de classes permettant la programmation de ce jeu.

2 La classe `Ship`

2.1 Spécification de la classe `Ship`

La classe `Ship` permet de représenter les bateaux. Une instance de `Ship` est construite avec une longueur et un nom. La longueur d'un bateau détermine son nombre de points de vie initial (*life points*). Lorsqu'il est touché (méthode `takeAHit()`), ce nombre est décrémenté d'un. Un bateau est coulé quand son nombre de points de vie arrive à 0 (méthode `hasBeenSunk()`).

Voici le diagramme de cette classe :



2.2 Questions

Question 1 : Donner le code définissant les attributs de `Ship` ainsi que le constructeur de la classe.

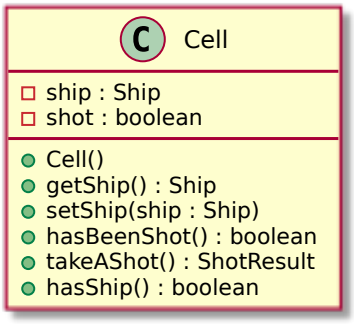
Question 2 : Donnez la méthode de tests qui permet de vérifier la spécification du comportement de la méthode `takeAHit()`.

3 La classe `Cell`

3.1 Spécification de la classe `Cell`

La classe `Cell` permet de représenter les cases du plateau de jeu. 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 `takeAShot()`). 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. 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) grace à la méthode `hasBeenShot()`.

On appelle `ShotResult` le type permettant de représenter les 3 réponses possibles après un tir d'un attaquant : `MISSED`, `HIT` et `SUNK`.



3.2 Questions

Question 3 : Comment peut-on définir le type `ShotResult` ?

Question 4 : Comment peut-on gérer la présence ou l'absence d'un bateau sur une case ?

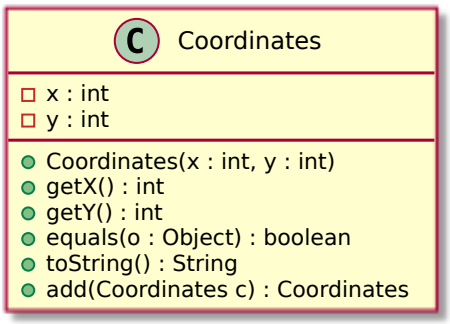
Question 5 : Donnez la ou les méthodes de tests qui permettent de vérifier cette spécification du comportement de la méthode `takeAShot()`.

Question 6 : Donnez le code de la méthode `takeAShot()`.

4 La classe `Coordinates`

4.1 Spécification de la classe `Coordinates`

Pour représenter les coordonnées des cases du plateau de jeu, on définit la classe `Coordinates`. La méthode `toString` renvoie un chaîne de caractère au format suivant : `(X, Y)` avec `X` la coordonnée en x et `Y` la coordonnée en y. Deux positions sont considérées égales si elles ont les mêmes coordonnées en x et en y.



4.2 Questions

Question 7 : Comment peut-on rendre les instances de `Coordinates` immuables ?

Question 8 : Donnez le code de la méthode `equals` de `Coordinates`.

5 La classe `Grid`

5.1 Spécification de la classe `Grid`

La classe représentant le plateau de jeu (la grille) s'appelle `Grid`. On décide de représenter son état par un tableau à deux dimensions de cases qui sont des objets de type `Cell`.

La méthode `shootAt` de la classe `Grid` est utilisée lorsque le joueur attaquant vise une case. Son résultat est la réponse lorsque l'attaquant vise la case située à la position `p` sur le plateau de jeu. La position `p` est passée en paramètre de la méthode.

5.2 Questions

Question 9 : Donnez le code définissant les attributs de `Grid` ainsi que le constructeur de la classe 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.

Question 10 : Dessinez le diagramme de classe de `Grid`.

Question 11 : Donnez la signature, la javadoc, les méthodes de test et le code de la méthode `shootAt` de `Grid`.

6 Rappels sur les tests

6.1 Méthodologie de test

Une classe de test par classe à tester. Une méthode de test par méthode ou cas à tester.
Le code d'une classe de test testant une classe `NameTestedClass` a le format suivant :

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

public class NameTestedClassTest {
  @Test
  void testTestMethod(){
    // code containing assertions
  }
}
```

6.2 Assertions (liste non-exhaustive)

Pour tester, on utilise des assertions qui doivent être vraies. Vous trouverez ci-dessous une listes des assertions les plus utiles.

- `assertThat(object).isNotNull()` : vérifie que la référence **n'**est **pas** `null`
- `assertThat(actual).isSameAs(expected)` : vérifie que les deux objets sont les mêmes (même référence : utilisation de `==`).
- `assertThat(condition).isTrue()` : vérifie que `condition` est vraie.
- `assertThat(condition).isFalse()` : vérifie que `condition` est faux.
- `assertThat(actual).isEqualTo(expected)` : vérifie que `expected` est égal à `actual` (en appelant `equals` sur `actual`).
- `assertThat(actual).isNotEqualTo(expected)` : vérifie que `expected` **n'**est égal **pas** à `actual` (en appelant `equals` sur `actual`).
- `assertThat(iterable).contains(element)` : vérifie que `iterable` (qui peut être une `List`, un tableau, ...) contient `element`.
- `assertThat(iterable).containsOnly(element)` : vérifie que `iterable` (qui peut être une `List`, un tableau, ...) contient seulement `element`.
- `assertThat(actual).isCloseTo(expected, within(delta))` : vérifie que $|expected - actual| \leq delta$ (comparaison de double).