Durée totale : **3h**. Tous documents autorisés. Nombre de pages : **5**
barème indicatif : spécification JML et test = 2/3, model-checking = 1/3

Dans tout le sujet les assertions JML sont supposées non activées ; les spécifications sont à écrire en JML (y compris la signature des méthodes) et à traiter dans l'esprit de la programmation défensive ; on ne demande pas de traiter dans les spécifications et les tests les cas récurrents de « levée de `NullPointerException` en cas de paramètre valant `null` » ; on s'autorisera à utiliser les constructions de Java 5 même si JML ne le permet normalement pas.

On rappelle :

- pour l'interface Java `List` : les méthodes `contains` et `equals` (qui appellent `equals` sur les éléments de la liste), la méthode `add` pour un ajout en fin, la méthode `size` ;
- en JML : les constructions `exists` et `forall`, par exemple pour une `boite` de type `List<Oeuf>` :

```
(\forall Oeuf o ; boite.contains(o) ; o.estIntact()) ; // boite OK
(\exists Oeuf o ; boite.contains(o) ; o.estCasse()) ; // boite KO
```

On souhaite spécifier et tester une toute petite partie d'un jeu de bataille navale (fig. 1). Chaque joueur dispose d'une grille sur laquelle il peut disposer ses bateaux et qui n'est pas visible par l'autre joueur. Le but du jeu est de «couler» les bateaux de l'adversaire en devinant leur position. Chaque joueur dispose de bateaux de différente longueur (données en nombre de cases) : par exemple des destroyers de longueur 3, et des frégates de longueur 2.

Chaque joueur place ses bateaux sur sa grille. Les bateaux sont placés horizontalement ou verticalement (pas de placement en diagonale) sans dépasser de la grille. Deux bateaux peuvent se toucher mais pas se chevaucher (chaque case est occupée par au plus un bateau).

À chaque tour, chaque joueur propose une case de la grille adverse sur laquelle il souhaite tirer. Son coup est évalué : si un bateau est placé sur cette case alors il est dit «touché» sur cette case ; si un bateau est touché sur toutes les cases qu'il occupe, il est dit «coulé» ; si le coup ne touche aucun bateau, il est «à l'eau».

Comme on le verra par la suite, la grille n'existe pas en tant que matrice, mais en tant que listes de cases associées aux bateaux qui les occupent. **On ne se préoccupera pas de la dimension de cette grille**.

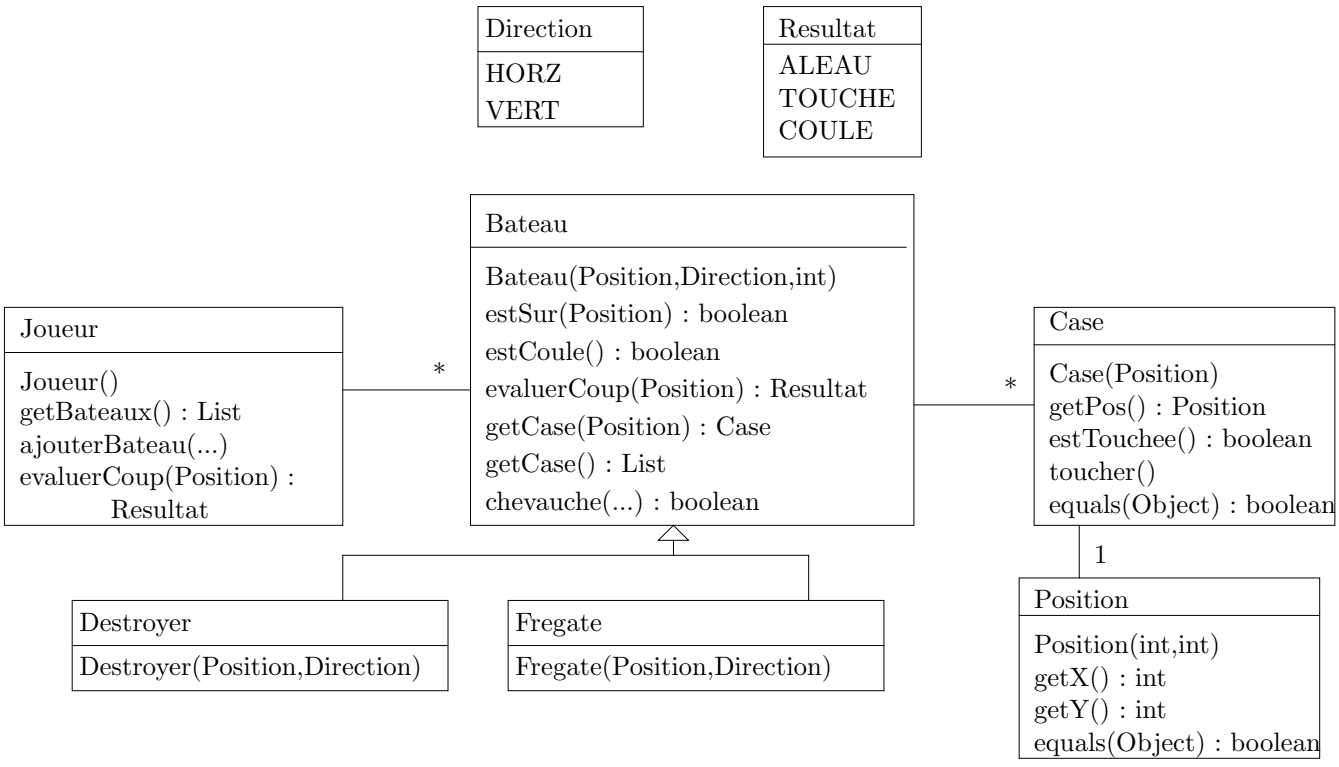

FIG. 1 – Diagramme pseudo UML très simplifié de la bataille navale

Exercice 1 : Classe `Position`, `Case`, types énumérés

Une `Position` est représentée par ses coordonnées cartésiennes. Deux `Position` sont égales si elles possèdent les mêmes coordonnées.

Q 1.1 : Donner la spécification de la méthode `equals` de `Position`. □

Une `Case` peut être touchée ou pas. À la construction elle n'est pas touchée. On indique qu'elle est touchée en appelant la méthode `toucher`. Les types énumérés `Direction` et `Resultat` s'utilisent classiquement (par exemple `Direction.HORZ`).

Exercice 2 : Classe `Bateau`

La classe abstraite `Bateau` représente un bateau par la liste des `Case` sur lequel il est placé. Un bateau peut être placé en n'importe quelle position (par exemple $(x = -3, y = 5)$). On construit un `Bateau` en donnant sa `Direction`, le nombre de cases qu'il occupe, et une de ses cases (la plus à gauche pour un placement horizontal, la plus haute pour un placement vertical). Les `Case` occupées sont créées dans le constructeur. On supposera que le constructeur ne lève pas d'exception. La méthode `getCase(Position)` retourne la `Case` associée à la position passée en paramètre, `null` si le bateau n'est pas sur cette position.

Les bateaux placés sur les grilles sont des instances des classes `Destroyer` et `Fregate`. Ces sous-classes ne contiennent qu'un constructeur qui appelle celui de `Bateau` avec la bonne taille.

Q 2.1 : On souhaite tester le constructeur de `Destroyer` (longueur 3) dans le cas nominal d'un bateau placé horizontalement en $(x = 2, y = 6)$. Donner un cas de test JUnit qui effectue ce test. □

Q 2. 2 : Pensez-vous que la conception choisie (construction des `Case`) dans le constructeur de `Bateau` soit pertinente ? Expliquer pourquoi. On conservera cette architecture par la suite. □

On suppose que la classe `Bateau` possède un attribut JML `lescases` de type `List<Case>` qui représente la liste des cases occupées par le bateau.

Q 2.3 : Donner la spécification de la méthode `estCoule`. □

Q 2.4 : Donner un cas de test JUnit qui vous semblent judicieux pour tester le comportement de la méthode `estCoule` dans le cas d'une `Fregate` (ce cas de test pourra contenir plusieurs assertions). □

La méthode `evaluerCoup` prend en paramètre une `Position` représentant une tentative pour viser ce bateau, répercute l'effet de ce coup sur la case du bateau touchée le cas échéant, et retourne `Resultat.COULE` si ce coup coule le bateau, `Resultat.TOUCHE` si ce coup touche le bateau sans le couler, et `Resultat.ALEAU` s'il ne le touche pas.

Q 2. 5 : Décrire en français les scénarios qui vous semblent judicieux pour tester les comportements de la méthode `evaluerCoup` sur une `Fregate`. Donne un cas de test JUnit qui correspond au scénario de votre choix. □

La méthode `chevauche` prend en paramètre un bateau et retourne vrai si ce bateau chevauche `this`.

Q 2. 6 : On souhaite tester `chevauche` sur des `Fregate`. Dessiner sur un morceau de grille les bateaux constituant les valeurs des données de test qui vous semblent judicieuses pour tester la méthode `chevauche` en cas de non chevauchement. □

Exercice 3 : Classe `Joueur`

Un `Joueur` est défini par la liste de ses bateaux. La méthode `ajouterBateau` prend en paramètre un bateau à ajouter à cette liste, et lève une exception de type `ChevauchementException` si ce bateau chevauche un des bateaux de la liste (il n'est alors pas ajouté). La méthode `evaluerCoup` prend en paramètre un coup représenté par une `Position` et répercute ce coup sur les bateaux du `Joueur`, en appelant sur les bateaux la méthode `evaluerCoup`.

On souhaite tester la classe `Joueur` indépendamment de la classe `Bateau`, par un test d'interaction nécessitant des mocks.

Q 3.1 : Indiquer comment modifier l'architecture de l'application pour permettre ce type de test. On adoptera cette nouvelle architecture par la suite. □

Q 3.2 : On suppose donné un attribut JML `lesbateaux` de type `List<...>` contenant les bateaux du `Joueur`. Donner la spécification de `ajouterBateau`. □

Q 3. 3 : On suppose donné un attribut privé Java `bateaux` de type `List<...>` contenant les bateaux du `Joueur`. Donner une mise en œuvre Java pour la méthode `ajouterBateau`. □

Q 3.4 : On souhaite tester la méthode `ajouterBateau` codée précédemment, avec le scénario suivant : ajout de 2 bateaux qui ne se chevauchent pas puis d'un 3ème bateau qui chevauche le second bateau. Décrivez un cas de test pour ce scénario en :

- précisant bien quels sont les objets du test et leur type ;
- donnant en Java les appels à l'objet testé et le ou les oracle(s) du test ;
- décrivant en français mais de manière précise le comportement des mocks (paramètres d'appel et retour de méthode).

□

Q 3.5 : Donner une mise en œuvre Java pour la méthode `evaluerCoup`. □

Q 3.6 : On souhaite tester la méthode `evaluerCoup` codée précédemment sur un joueur possédant 2 bateaux (par exemple `b1` et `b2` – on ne donnera pas les appels à `ajouterBateau` pour ne pas avoir à décrire les appels à `chevauche` qui en découlent). Décrire en français 3 scénarios qui vous semblent judicieux. Décrire plus précisément un de ces scénarios au choix en :

- donnant en Java les appels à l'objet testé et le ou les oracle(s) du test ;
- décrivant en français mais de manière précise le comportement des mocks (paramètres d'appel et retour de méthode).

□