TD3 : conception de jeu de tests - étude de cas « Bataille Navale »

De manière générale, les actions que **doit réaliser** une méthode sont de trois types :

- changement de l'état du système : mises à jour des attributs et des associations,
- changement des données pérennes (fichiers, base de données),
- valeur retournée.

Les actions **réalisées par** une méthode dépendent de 3 choses :

- les valeurs de ses paramètres,
- l'état de l'objet (valeurs des attributs, des associations) voir l'état des objets voisins,
- Les données pérennes (fichiers, base de données).

La conception de jeu de tests pour une **méthode** consiste à élaborer un jeu de tests en vue de vérifier que la méthode réalise bien toutes les actions qu'elle doit réaliser et ceci dans les divers cas d'usage (valeur de ses paramètres, état du système et des données pérennes, …).

Comment procéder ?

- Déterminer les divers cas d'usage et quels sont les éléments (attributs, association, …) qu'il faut tester car ils peuvent ou doivent être modifiés ;
- Puis élaborer un ensemble de scenarii permettant de tester **tous les cas d'usage** et écrire le code de la fonction qui testera un appel à la méthode à tester

Illustration de la procédure :

```
public class IntIterator {
      private int max = -1;
      private int current = -1;
      public IntIterator(int nbValeurs) {
                  if (nbValeurs > 0) max = nbValeurs-1;  }
      public boolean hasNext() {  return (current < max);  }
      public int next() {  if (current >= max)  return max;
                        current++; return current; }
      public int getCurrent() {return current; }
      public int getMax() {return max; }
}
```

La méthode `hasNext` a deux cas d'usage :

(1) cas où `current <= max` ; il faut tester que la valeur retournée est `true` ;
(2) cas où `current > max` ; il faut tester que la valeur retournée est `false`.

```
public void hasNextTest (IntIterator iter, boolean resExpected) {
      boolean res = iter.hasNext() ;
      assertEquals(res, resExpected); }
```

Les scenarii :

```
// iterateur avec 0 valeur
IntIterator iter = new IntIterator(-1);
hasNextTest(iter, false); hasNextTest(iter, false);
// iterateur avec 0 valeur
iter = new IntIterator(0);
hasNextTest(iter, false); hasNextTest(iter, false);
// iterateur avec 1 valeur
iter = new IntIterator(1);
hasNextTest(iter, true); iter.next();
hasNextTest(iter, false); hasNextTest(iter, false);
// iterateur avec 2 valeurs
iter = new IntIterator(2);
hasNextTest(iter, true); iter.next();
hasNextTest(iter, true); iter.next();
hasNextTest(iter, false); hasNextTest(iter, false);
// iterateurs avec 5 valeurs à 100 valeurs
for (int i = 5 ; i < 100 ; i++) {
     iter = new IntIterator(i);
     for (int j = 0; j < i; j++) {
          hasNextTest(iter, true); iter.next(); }
     hasNextTest(iter, false); hasNextTest(iter, false);
}
```

Question

1. Listez les cas d'usage à tester pour la méthode `next`, et pour chaque cas d'usage, listez les valeurs à tester.
2. Proposez des scénarii de tests permettant de tester tous les cas d'usage et réalisez la méthode `nextTest` qui permet de tester la méthode `next`.

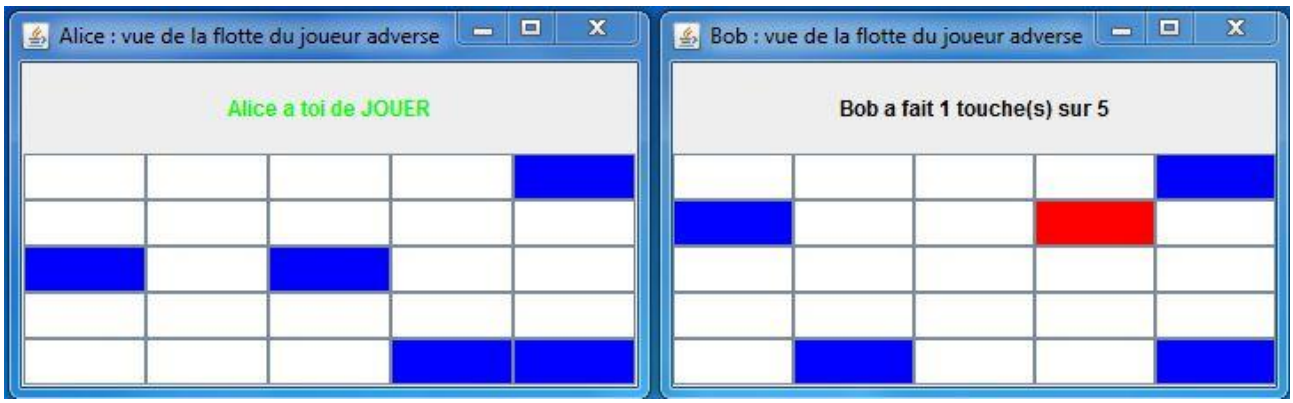
II Bataille navale

Comme son nom l'indique, la bataille navale est un jeu consistant à torpiller les navires de son adversaire. La flotte de chaque joueur est placée aléatoirement au début du jeu par l'application.

Ensuite, tour à tour, les joueurs lancent virtuellement des torpilles sur l'ennemi en indiquant les coordonnées d'un tir (par exemple B4).
L'application indique si l'un de ses bateaux a été atteint ou pas. Le gagnant est celui qui parvient à torpiller complètement les navires de l'adversaire avant que tous les siens ne le soient.

La taille des grilles de la bataille navale dépend du niveau (elle est plus importante dans les niveaux supérieurs).

Chaque joueur « voit » à une grille correspondant à la connaissance qu'il a acquise de la flotte de son adversaire. Par exemple, une case « vide » est une case non torpillée, une case bleue indique une torpille qui n'a pas touché un navire ; une case rouge indique une touche.



Quelques informations sur l'implémentation du jeu

```
public enum EtatCaseBN {  BATEAU_TOUCHE, A_L_EAU,  NON_DECOUVERTE ; }
```

L'attribut `torpillée` de `CaseBN` est à mis à `vrai`, une fois que la case est torpillée. A la construction de l'objet, l'attribut `aUnBateau` est mis à `vrai` si la case contient un bateau.

CaseBN
-x : int - y : int - aUnBateau : boolean - torpillée : boolean = false
+ getEtat() : EtatCaseBN + torpiller() : EtatCaseBN +CaseBN (x : int, y : int, aUnBateau : boolean) : void + getX() : int + getY() : int

La méthode `getEtat` retourne `NON_DECOUVERTE` tant que la case n'a pas été torpillée. Après que la case ait été torpillée, `getEtat` retourne `BATEAU_TOUCHE` si l'attribut `aUnBateau` est à `vrai`; sinon `getEtat` retourne `A_L_EAU`.
La méthode `torpiller` de `CaseBN` met l'attribut `torpillé` à `vrai`. Puis, elle retourne `BATEAU_TOUCHE` si l'attribut `aUnBateau` est à `vrai`; sinon elle retourne `A_L_EAU`.

Questions

3. Listez les cas d'usage de la méthode `getEtat` de la classe `CaseBN` et pour chaque cas d'usage, listez les valeurs à tester.
4. Listez les cas d'usage de la méthode `torpiller` de la classe `CaseBN` et pour chaque cas d'usage, listez les valeurs à tester.
5. Proposez les scénarios de tests permettant de tester tous les cas d'usage et réalisez la méthode `torpillerCaseTest` qui permet de tester la méthode `torpiller` de la classe `CaseBN`.
6. Proposez les scénarios de tester permettant de tester tous les cas d'usage et réalisez la méthode `getEtatTest` qui permet de tester la méthode `getEtat`.

Vous pouvez répondre aux deux questions précédentes simultanément.

La méthode `torpiller()` de la classe `JoueurBN` doit incrémenter la variable `nombreTirsRecus`. Si la case n'a pas été déjà été torpillée alors elle est torpillée et la variable `nombreToucheRecues` est incrémentée dans le cas où la case torpillée contient une partie d'un bateau.
`torpiller()` retourne l'état `EN_COURS` si le joueur torpillé n'a pas perdu (à savoir que `nombreToucheRecues` est inférieure au nombre de touches à faire).

Questions

7. Compléter avec des associations le diagramme de classes ci-dessus, si nécessaire.
8. Listez les cas d'usage de la méthode `torpiller` de la classe `JoueurBN` et pour chaque cas d'usage, listez les valeurs à tester.
9. Proposez un jeu de tests complet pour l'implémentation de la méthode `torpiller` de la classe `JoueurBN`.

La méthode `tirParJoueur(x, y, jx)` de la classe `ControleurBN` retourne `false` (elle n'appelle aucune méthode) si la partie est terminée, si `jx` n'est pas le joueur courant ou si `(x, y)` ne correspond pas au coordonnées d'une case.
Dans les autres cas, la méthode `torpiller(x, y)` du joueur non-courant (`jy`) est appelée. Si cette torpille lancée par `jx` résultant en la perte de la partie par `jy` alors `jx` est déclaré « gagnant » ; sinon le joueur courant change (`jy` devient le joueur courant).

Questions

10. Compléter le diagramme de classes ci-dessus avec des associations, si nécessaire.
11. Proposez un jeu de tests complet pour l'implémentation de la méthode `tirParJoueur` de la classe `ControleurBN`.

Diagramme de classes à compléter

```
public enum EtatJeu {  en_COURS, TERMINE; }
```