Université Paris Cité **Programmation Orientée Objet**
L2 **Année 2024-2025**

TD n° 9 - Correction

Modélisation – Bataille navale

1 Le jeu de bataille navale

Principe du jeu. La bataille navale se joue à deux joueurs, à l'aide de deux damiers rectangulaires de mêmes dimensions et un ensemble de figurines de bateaux pouvant couvrir une ou plusieurs cases. Au début du jeu, chaque joueur reçoit un damier, et les figurines sont équitablement réparties entre les deux joueurs. Chaque joueur dissimule sa grille à l'autre, et dispose ses figurines sur sa grille.

Les joueurs énoncent ensuite à tour de rôle les coordonnées d'une case sur le damier adverse. Si cette case est recouverte par l'une des tranches d'un bateau, la tranche est alors considérée comme détruite. Un bateau est coulé lorsque tous ses tranches ont été détruites. Dans tous les cas, le joueur adverse informe l'autre du résultat de son coup ("dans l'eau", "touché", "coulé").

Le jeu se termine lorsque tous les bateaux d'un joueur ont été détruits : le joueur adverse est alors déclaré gagnant.

Types de figurines et contraintes de placement. Les bateaux se divisent en deux grandes catégories : les bateaux de surface et les submersibles. Chaque bateau couvre entre 1 et 5 cases. Le placement des figurines de chaque joueur doit respecter les contraintes suivantes :

1. Deux bateaux ne peuvent recouvrir la même case.
2. Deux bateaux de surface ne peuvent être en contact (par bord ou par coin).
3. Un submersible peut être en contact avec un bateau de surface, mais pas avec un autre submersible.

2 Modélisation du jeu en Java

Proposer une, ou mieux, plusieurs manières de modéliser ce jeu en Java, en précisant les champs et les méthodes indispensables dans chaque classe. Votre modélisation devra en particulier examiner les points suivants :

1. Comment déterminer si un bateau est coulé ?
2. Lorsque l'on place un bateau sur le damier en début de partie, comment déterminer si ce placement est valide ?
3. Comment envisager l'ajout de nouvelles catégories de bateaux avec de nouvelles règles de placement ?

Correction : Une solution naïve pour représenter un damier consisterait à prendre un simple tableau de booléens, mais déterminer si un bateau respecte les règles de placement ou s'il est coulé seraient deux problèmes insolubles. En remplaçant les booléens par un type de bateau (surface ou submersible) couplé à un état courant (intact ou détruit), on pourrait envisager la vérification du placement, mais pas celle du fait qu'un bateau vient d'être entièrement détruit sans un examen laborieux du voisinage de la dernière case jouée.

Une meilleure solution consiste à représenter le damier comme une matrice de *tranches* de bateau, une tranche encapsulant une référence vers un bateau et un numéro de tranche, un bateau encapsule un tableau de booléens indiquant pour chacune de ses tranches si elle est détruite :

```
public class ShipSlice {
  private Ship ship;
  private int slice;

  public ShipSlice(Ship ship, int slice) {
    this.ship = ship;
    this.slice = slice;
  }

  public Ship getShip() {
    return ship;
  }
}
...  class Ship {
  private boolean[] hits;

  public Ship(int size){
    this.hits = new boolean[size];
  }
  ...
}
```

Reste le problème du respect des règles de placement. Si l'on souhaitait écrire dans `Ship` une méthode `boolean canTouch(Ship ship)`, comment l'implémenter sans utiliser `instanceof`, `getClass()`, la comparaison de tags, etc., uniquement en se servant de la liaison dynamique ?

Une solution possible est l'usage du design pattern "Visitor". La classe `Ship` est déclarée abstraite, et étendue en deux classes concrètes, une pour chaque sorte de bateau. La méthode peut s'implémenter ainsi :

```
public abstract class Ship {
  public abstract boolean canTouch(Ship ship);
  public abstract boolean canBeTouchedBy(SurfaceShip surfaceShip);
  public abstract boolean canBeTouchedBy(Submersible submersible);
}
```

```
public class SurfaceShip extends Ship {
  public boolean canTouch(Ship ship) {
    return ship.canBeTouchedBy(this);
  }
  public boolean canBeTouchedBy(SurfaceShip surfaceShip) {
    return false;
  }
  public boolean canBeTouchedBy(Submersible submersible) {
    return true;
  }
}
```

```
public class Submersible extends Ship {
  public boolean canTouch(Ship ship) {
    return ship.canBeTouchedBy(this);
  }
  public boolean canBeTouchedBy(SurfaceShip surfaceShip) {
    return true;
  }
  public boolean canBeTouchedBy(Submersible submersible) {
    return false;
  }
}
```

Dans le corps de `canTouch` de la classe `SurfaceShip` (resp. `Sumbersible`), `this` est une reférence de type `SurfaceShip` (resp. `Sumbersible`), et la descente vers la réponse de l'une ou ou l'autre des versions de `canBeTouched` dans la classe du `ship` argument de `canTouch` est déterminée par le type de `this`.

Aussi élégante que puisse être cette solution, elle présente un inconvénient de taille : la classe `Ship` devient entièrement dépendante de ses extensions, et l'ajout de nouvelles catégorie de bateaux supposerait d'ajouter toutes les méthodes aditionnelles nécessaires dans cette classe et toutes les autres.

Une autre solution consisterait à définir `Ship` comme une classe concrète en étiquetant ses instances par les éléments d'une énumération. Ce sont les instances de cette énumération à qui on déléguera la réponse de `canTouch` :

```
public enum ShipType {
  SURFACE_SHIP, SUBMERSIBLE;
  boolean canTouch(ShipType shipType) {
    return this != shipType;
  }
}
public class Ship {
  private ShipType shipType;
  private boolean[] hits;
  public Ship(ShipType shipType, int size) {
    this.hits = new boolean[size];
  }
  public boolean canTouch(Ship ship) {
    return shipType.canTouch(ship.shipType);
  }
}
```

L'ajout de nouvelles sortes de bateaux ne demanderait dans ce cas que la modification de l'énumération. Par exemple, si l'on voulait ajouter le type "amphibie", avec pour nouvelle règle "un amphibie ne peut être en contact avec aucune autre pièce", on pourrait modifier `ShipType` (et seulement cette énumération) ainsi :

```
public enum ShipType {
  SURFACE_SHIP, AMPHIBIAN, SUBMERSIBLE;

  private static boolean[][] allowed =
  {{false, false, true},
   {false, false, false},
   {true,  false, false}};

  boolean canTouch(ShipType shipType) {
    return allowed[ordinal()][shipType.ordinal()];
  }
}
```

Examinons enfin le problème du placement effectif des bateaux. Si l'on dispose d'une méthode permettant de déterminer si deux bateaux peuvent être en contact, ce problème peut être résolu ainsi (*c.f.* `Grid.java`) :

1. Vérifier que le bateau sera dans les limites du damier en le plaçant à la position souhaitée.
2. Verifier que les cases qui seront recouvertes par le bateau ne sont pas déjà occupées.
3. Vérifier que le voisinage de ces cases ne contient pas un bateau qui ne peut pas être en contact avec celui que l'on souhaite poser.