UE Conception Orientée Objet

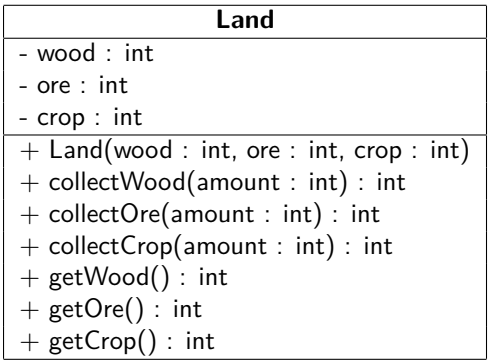
Devoir Surveillé
3h
Copie des diapositives de cours annotée autorisée.
Fiches « *design pattern* » du cours autorisées.
Dictionnaires de langue autorisés.
Autres documents interdits.

Sauf mention expresse, la javadoc et les tests des classes à écrire ne sont pas demandés.

Exercice 1 :
On se place dans le contexte d'un jeu de simulation au tour par tour où les joueurs doivent gérer un territoire et en exploiter les ressources à l'aide d'unités de personnages ayant différentes compétences. À chaque tour le joueur récupère des points en fonction de la collecte de ressources réalisée par les unités qu'il contrôle.
Dans cet exercice on ne s'intéressera pas directement à la logique du jeu mais à la gestion d'une partie des unités de personnage (appelées simplement *unités* par la suite).

Le terrain et les ressources

Le jeu considéré se déroule sur une carte divisée en cases de terrain (*land*). Au début du jeu chaque (case de) terrain possède un certain nombre de ressources. On identifie trois types de ressources : le bois (*wood*), le minerai (*ore*) et des cultures (*crop*). Les joueurs peuvent, petit à petit, collecter ces ressources grâce à leurs unités et obtiennent en retour un certain nombre de points. Une case de terrain est modélisée par le type `Land` suivant :



Les méthodes *`collectXxx`* sont les méthodes qui permettent de récolter *`amount`* ressources du type *`Xxx`* sur cette case. Les ressources disponibles sont diminuées d'autant. Le résultat de ces méthodes est le nombre de points reçus pour la collecte, il vaut 0 si la ressource collectée n'est plus disponible sur ce terrain.

Les outils

On le verra par la suite, certaines unités peuvent utiliser des outils. Ces outils sont représentés par l'interface `Tool` dont le code est fourni en annexe.
Une unité travaille sur un terrain grâce à son outil via la méthode `collect` de celui-ci. Chaque utilisation d'un outil peut en provoquer la casse avec une certaine probabilité. La méthode `isBroken` gère cette casse éventuelle et indique par son résultat si l'outil est cassé ou non.
Le jeu propose trois types d'outils : des haches (*axe*), pioches (*pick*) et faux (*scythe*) pour collecter du bois, des minerais ou des cultures. Le rendement d'un outil dépend, selon les cas, de la force et de l'habileté de l'unité qui l'utilise.
Un diagramme des classes pour les outils est fourni en annexe avec des extraits de code pour ces classes.

Les unités

Le joueur peut déplacer des unités pour les positionner sur les cases de terrain.
Toutes les unités sont caractérisées par une vitesse de déplacement représentée par un entier. De plus chaque unité dispose d'un « coût d'entretien » que doit payer le joueur à chaque tour pour pouvoir conserver cette unité. Cette information est un entier fourni par la méthode `cost()`.
Enfin, une unité peut travailler sur un terrain. Cela se fait grâce à la méthode `workOn` dont le résultat est le nombre de points obtenu suite à la collecte effectuée par l'unité :

```
public int workOn(Land land) throws BrokenToolException
```

Ce travail se fait grâce à la méthode `collect` de l'outil. L'exception est déclenchée si l'outil de l'unité est cassé (voir ci-dessus).

Le jeu permet au joueur de contrôler des unités simples (*single unit*) et des groupes d'unités (*group unit*). Le coût d'entretien d'une unité simple est fixé à 1.

Les unités simples.
Les unités simples disposent d'une force et d'une habileté, deux entiers. De plus, toutes les unités simples possèdent un outil qui leur permet de travailler sur un terrain.
Lors de la création d'une unité simple un nouvel outil est créé[1] et lui est affecté. En cas de casse de cet outil, on peut le remplacer par un nouvel outil grâce à un appel de la méthode :

```
public void changeTool()
```

de l'unité.

Il existe trois types d'unités simples « *de base* », spécialisés dans la récolte d'une ressource :

- les bûcherons (*lumberjack*) dont l'outil est une hache,
- les mineurs (*miner*) dont l'outil est une pioche,
- les fermiers (*farmer*) dont l'outil est une faux.

Chacune des unités simples se déclinent en des versions *expertes*. C'est-à-dire des unités simples plus fortes ou plus habiles. Les versions « plus fortes » ont une force doublée et les versions « plus habiles » ont leur habileté doublée. Le coût de ces unités est augmenté de 1. En plus de ce coût, seules les force ou habileté de ces unités changent par rapport à leur version « normales ». Evidemment on peut également avoir des unités « plus fortes et plus habiles » : leur force et leur habileté sont doublées et le coût est augmenté de 2.

Enfin toutes les unités simples (y compris les versions expertes) existent également en versions « à cheval » (*mounted unit*). Ces unités sont équipées d'un cheval, instance d'une classe `Horse` qui dispose d'une méthode `getSpeed()` qui fournit la vitesse du cheval[2]. La vitesse de ces unités est celle de leur cheval et leur coût est augmenté de 2 par rapport à la version « non montée ».

Les groupes d'unités.
Les groupes d'unités (simplement *groupe* par la suite) permettent au joueur de regrouper plusieurs unités simples en une seule unité. Les unités peuvent être ajoutées une par une au groupe. La vitesse d'un groupe est celle de la plus lente des vitesses des unités qui le composent et le coût d'un groupe est la somme des coûts des unités qui le composent. Le travail sur un terrain d'un groupe correspond au travail de chacune de ses unités sur ce terrain. Si, lors de ce travail, l'outil d'une des unités simples est cassé alors cet outil est immédiatement remplacé même si le travail de cette unité pour cette fois est perdu.

Q 1. Donnez un **ou plusieurs** diagrammes UML de classes détaillés qui présentent clairement votre proposition de conception pour gérer les différents types d'unités.

Dans votre diagramme, vous ferez apparaître clairement les attributs, constructeurs et méthodes (et leurs signatures), ainsi que les surcharges.

Faites des diagramme clairs et lisibles. N'hésitez pas à prendre votre copie au format paysage, ou mieux à faire votre diagramme sur les deux pages intérieures de votre copie.

[1] avec une puissance arbitraire fixée par une constante
[2] Pour simplifier, ce sera la seule méthode utile dans ce sujet pour la classe `Horse`.

Q 2. Quel(s) *design pattern(s)* avez-vous mis en œuvre dans votre solution ?

Justifiez clairement dans chaque cas en quelques phrases claires pourquoi l'utilisation de ce *design pattern* est pertinent dans votre solution.

Q 3. Donnez le code du constructeur de la classe de base des unités simples.

Q 4. Donnez les lignes de code qui, à partir de votre proposition, permettent de créer :

- une unité *bûcheron*,
- une unité *fermier plus habile*,
- une unité *mineur à cheval*,
- une unité *mineur plus fort à cheval*,
- une unité groupe regroupant les quatre unités précédentes.

Q 5. Donnez le code de la ou des méthodes de test pour vérifier le comportement de la méthode `collectWood` de la classe `Land`.

Q 6. Donnez le code de la ou des méthodes de test pour vérifier que la méthode `workOn` déclenche une exception si l'outil de l'unité est cassé.

Indiquez comment il faut faire pour que ce test soit vérifié pour chacune des classes d'unités simples en donnant le code nécessaire pour la classe des *bûcherons* et en précisant comment cela se généralise aux autres classes d'unités simples.

Q 7. Donnez le code de la ou des méthodes de test qui permettent de vérifier que la méthode `collect()` d'un outil est appelée exactement une fois par la méthode `workOn()` de l'unité associée, qu'une exception soit déclenchée ou non.

Q 8. Donnez le code de la ou des méthodes de test qui permet de vérifier que le coût d'une *unité à cheval* est augmenté de 2 par rapport à la même unité qui n'est pas à cheval.

Q 9. Pour tous les types d'unités où elles sont **définies ou redéfinies (surchargées)** donnez le code java des méthodes :

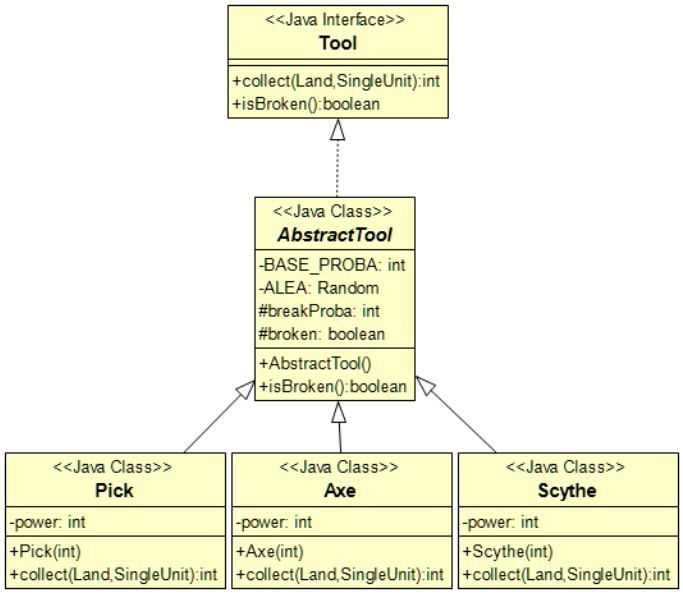
Q 9.1. `workOn()`

Q 9.2. `getSpeed()`

Précisez clairement à chaque fois la classe où se situe le code fourni.

Annexe

Les outils



La classe `AbstractTool` gère la probabilité de casse de l'outil dans la méthode `isBroken()`.

```java
/** tools used by units to collect resources */
public interface Tool {
  /** this tool is used by a collector to work on the given land and to produce points
  * @param land the worked land
  * @param collector the unit using this tool
  * @return the number of points received for this collecting
  */
  public int collect(Land land, SingleUnit collector);
  /**
  * @return true iff this tool is broken
  */
  public boolean isBroken();
}
```

```java
public class Axe extends AbstractTool {
  private int power;
  public Axe(int power) {
    this.power = power;
  }
  @Override
  public int collect(Land land, SingleUnit collector) {
    return land.collectWood(this.power * (collector.getStrength() + collector.getAbility()) / 2 );
  }
}
```

```java
public class Pick extends AbstractTool {
  ...
  @Override
  public int collect(Land land, SingleUnit collector) {
    return land.collectOre(this.power * collector.getStrength());
  }
}
```

```java
public class Scythe extends AbstractTool  {
  ...
  @Override
  public int collect(Land land, SingleUnit collector) {
    return land.collectCrop(this.power * collector.getAbility());
  }
}
```