TD - Séance n° 5

Héritage

Exercice 1 On considère comme lors du TD3 une classe `Personne`, avec des attributs légèrement modifiés :

```java
public class Personne {
  protected String nom;
  protected int argent;
  protected int pdv; // les points de vie de la personne

  public Personne(String nom, int argent, int pdv) {
    this.nom = nom;
    this.argent = argent;
    this.pdv = pdv;
  }

  public int getArgent() {
    return argent;
  }

  public void gain(int n) {
    this.argent += n;
  }

  public boolean perte(int n) {
    if (n > this.argent)
      return false;
    this.argent -= n;
    return true;
  }

  public void blessure(int n) {
    this.pdv -= n;
  }


  public void attaque(Personne p) {
    p.blessure(0);
  }

  public String toString() {
    return "Je m'appelle " + this.nom + ". J'ai "
      + this.argent + " unités monétaires, et "
      + this.pdv + " points de vie.";
  }
}
```

Comme dans le TD3, on a des classes `Noble`, `Pretre` et `Roturier` qui héritent de `Personne`.

Dans cet exercice, nous allons modéliser l'une des activités favorites de la noblesse au Moyen-âge : la guerre.

On créera pour cela une classe `Chevalier` (i.e les combattants de la noblesse) héritant de `Noble` ainsi que des classes `Archer` et `Fantassin` héritant de `Roturier`. La classe `Fantassin` possède un attribut `int degats` qui indique le nombre des points de vie à retirer lors d'une attaque. Les comportements militaires seront modélisés par une méthode `void attaque(Personne p)` et devront être implémentés par `Chevalier`, `Archer` et `Fantassin`.

1. Proposez des modifications à la méthode `blessure` pour éviter les valeurs négatives et afficher un message en cas de mort de la personne (lorsque ses points de vie tombent à zéro).
2. Redéfinissez la méthode `attaque` dans les classes `Chevalier`, `Archer` et `Fantassin` (l'argument de la méthode `attaque` représente la personne attaquée), sachant que :
   (a) un `Archer` tue la personne qu'il attaque (i.e il lui enlève tous ses points de vie).
   (b) Un `Chevalier` n'attaque une personne que si elle est elle-même une instance de `Chevalier` et dans ce cas le `Chevalier` attaqué est capturé (i. e. il perd sa liberté et il ne peut plus attaquer). On rajoute pour cela un attribut `Personne geolier` dans la classe `Chevalier` indiquant qui a capturé ce `Chevalier`, ou valant `null` s'il est libre. Pourquoi écrire une méthode `void attaque(Chevalier p)` ne répondrait pas à la question ?
   (c) un `Fantassin` capture la personne qu'il attaque si celle-ci est une instance de `Chevalier` et enlève `degats` points de vie à la personne si celle-ci n'est pas une instance de `Chevalier`.
3. Lorsqu'un chevalier est capturé, il a la possibilité de payer une rançon pour racheter sa liberté. Rajouter dans `Chevalier` une méthode `boolean acheteLiberte()` qui fonctionne de la façon suivante : si le chevalier capturé a les moyens de payer la rançon (i. e. son attribut `argent` est supérieur au montant de la rançon) alors il paye la rançon à son geôlier et regagne sa liberté. La méthode renvoie `true` si, et seulement si la rançon a été payée. Le montant de la rançon étant le même pour tous les chevaliers, vous le définirez dans un champ
   `private static final int prixLiberté = 50;`

Exercice 2 La guerre au Moyen-âge donnait rarement lieu à des batailles rangées mais consistait principalement en escarmouches et en pillages. Le but de cet exercice est de modéliser les pillages.

1. Créer une classe `Village` possédant un attribut `LinkedList<Roturier>`.

2. Créer une classe `Condottiere` héritant de `Personne` et possédant des attributs `LinkedList<Archer>` et `LinkedList<Fantassin>` (un condottiere est un chef d'armée au Moyen-Âge).
3. Créer une méthode `void attaque(Village v)` dans la classe `Condottiere`. Lorsqu'un `Condottiere` attaque un village, ce dernier est mis à sac (i. e. chaque villageois se fait voler la moitié de son argent). Les gains récupérés sont répartis pour moitié entre le condottiere et pour l'autre moitié uniformément entre les membres de sa compagnie.

Exercice 3 Pour des raisons techniques (sauvegarde d'un état par exemple), on souhaite avoir la possibilité de cloner les villages. Pour cela nous allons redéfinir la méthode `public Object clone()` que `Village` hérite de la classe `Object`. En effet cette méthode de clonage par défaut se contente d'une copie superficielle, souvent insuffisante.

1. Commencez par redéfinir la méthode `clone` dans la classe `Roturier`. Pourquoi cette redéfinition est-elle nécessaire ? Pensez à signaler à Java que la classe implémente l'interface `Cloneable` et que la méthode `clone` peut lever l'exception `CloneNotSupportedException`. N'hésitez pas à relire le cours correspondant.
2. Redéfinissez ensuite la méthode `clone` dans la classe `Village`. Soyez attentifs au clonage des habitants.

Exercice 4 On souhaite également pouvoir comparer des villages entre eux. Faites en sorte que la méthode `public boolean equals(Object o)`, initialement définie dans la classe `Object`, implémente les notions d'égalité suivantes.

1. Deux roturiers sont identiques lorsqu'ils ont le même nom, la même quantité d'argent et le même nombre de points de vie, ainsi que la même classe (i.e. `Archer` ou `Fantassin`) s'ils en ont une.
2. Deux villages sont identiques s'ils ont les mêmes habitants (pour simplifier on pourra dans un premier temps considérer qu'ils doivent avoir leurs habitants listés dans le même ordre pour être identiques).