Programmation avancée
TP 3 – L'appel de Cthulhu – Liaison dynamique

V. Padovani, PPS - IRIF

> *Les égouts de Londres sont peuplés de deux sortes de créatures : monstres, et chasseurs de monstres. Parmi les monstres, on trouve notamment les vampires et les loups-garous. Parmi les chasseurs de monstres, on trouve notamment les détectives et les savants fous.*

Si l'on vous demandait de modéliser en Java la situation précédente par une hiérarchie de classes, à la manière de l'exercice sur les Terminators – en implémentant par exemple, la capacité pour chaque créature modélisée de se présenter – votre solution commencerait vraisemblablement par quelque chose comme :

```
abstract class Creature {
  private String nom;
  Creature (String nom) {
    this.nom = nom;
  }
  void sePresente () {
    System.out.println ("Je m'appelle " + nom);
  }
}
```

```
abstract class Monstre extends Creature {
  Monstre (String nom) {
    super (nom);
  }
  void sePresente () {
    super.sePresente ();
    System.out.println ("Je suis un monstre.");
  }
}
```

```
class LoupGarou extends Monstre {
  LoupGarou (String nom) {
    super (nom);
  }
  void sePresente () {
    super.sePresente ();
    System.out.println ("Je suis aussi un loup-garou.");
  }
}
```

```
class Vampire extends Monstre {
  Vampire (String nom) {
    super (nom);
  }
  void sePresente () {
    super.sePresente ();
    System.out.println ("Je suis aussi un vampire.");
  }
}
```

Recopiez (par copier-coller, en l'indentant) le code ci-dessus dans un fichier `Chtulhu.java` de classe principale `Chtulhu` : cette modélisation est incomplète, mais suffira à faire quelques tests dans le `main` de sa classe principale. Il faudrait aussi, en principe, répartir les classes dans des fichiers distincts, mais peu importe pour ce seul exercice. Prevenez-moi lorsque vous serez tous prêts, nous ferons l'exercice suivant ensemble.

Exercice 1 : Interactif

Les instructions suivantes seront toutes supposées écrites dans `main`.

1. A-t-on le droit d'écrire ceci (voir le code) ?

   ```
   Creature cr = new Creature ("machin");
   ```

   Pourquoi ?

2. A-t-on le droit d'écrire ceci ? Dans tous les cas, à quoi pourrait bien servir une telle référence si l'on ne peut pas écrire l'initialisation précédente ?

   ```
   Creature cr;
   ```

3. Qu'afficheront précisement les instructions suivantes (voir le code) ?

   ```
   Vampire vp = new Vampire ("Vlad");
   vp.sePresente ();
   ```

4. Que donne le code suivant ? Une erreur ? Autre chose ?

   ```
   Creature cr;
   Vampire vp = new Vampire ("Vlad");
   cr = vp;
   cr.sePresente ();
   ```

   Testez-le. Qu'en concluez-vous ?

5. Donc, que devrait donner le code suivant ?

   ```
   Creature c;
   LoupGarou lg = new LoupGarou ("Wolfshead");
   cr = lg;
   cr.sePresente ();
   ```

6. Et ce code-là ?

```
Creature cr = null;
if(Math.random () > .5) {
  cr = new Vampire ("Vlad");
}
else {
  cr = new LoupGarou ("Wolfshead");
}
cr.sePresente ();
```

7. Et celui-ci ?

```
Monstre ms;
LoupGarou lg = new LoupGarou ("Wolfshead");
ms = lg;
ms.sePresente ();
```

8. Et celui-là ?

```
Monstre ms = new LoupGarou ("Wolfshead");
Creature cr = ms;
cr.sePresente ();
```

Observez à nouveau l'exemple n°6. Qu'entend-on ici par *liaison dynamique* ? Quelles nouvelles règles du langage pouvez-vous déduire de tous ces exemples ?

Exercice 2 : Copier-Coller-Modifier

Compléter rapidement votre programme en lui ajoutant une classe (abstraite) `Chasseur`, ainsi que les classes (concrètes) `Detective` et `SavantFou`.

Exercice 3 : Tableau de monstres

Tout détective possède un carnet répertoriant l'ensemble des monstres rencontrés au cours de ses pérégrinations.

1. Ajouter à la classe `Detective` un champ de type "tableau de références vers `Monstre` " réprésentant ce carnet, ainsi qu'un champ entier indiquant le nombre courant de monstres dans le carnet [1].
2. Initialiser ce champ dans le constructeur de `Detective`, à un tableau suffisamment grand (mettons, 100 cases).
3. Ajouter à `Detective` une méthode permettant d'ajouter un monstre au carnet :

   ```
   void ajouterMonstre (Monstre monstre)
   ```

4. Tester cette méthode dans `main` en l'invoquant sur une instance de `Detective`, en lui donnant en argument des instances de `Vampire` et de `LoupGarou`.
5. Ajouter à `Detective` une méthode `void lireCarnet()`. Son invocation présentera le contenu du carnet sur le modèle suivant :

1. Sur le modèle de la classe `Pile` du TD n° 1, au type du tableau près.

```
Page n. ..., le monstre s'est présenté ainsi :
Je m'appelle ...
Je suis un monstre.
Je suis aussi un ...
```

6. Peut-on dans `main` écrire ceci ?

```
Chasseur ch = new Detective ("Sherlock");
ch.ajouterMonstre (new Vampire ("Vlad"));
```

Pourquoi, *ie.* quelle est la règle ?

Exercice 4 : Métamorphoses et gratouilles

Le loup-garou est capable de se métamorphoser : il peut être soit dans l'état humain, soit dans l'état d'un loup. On peut, à ses risques et périls, tenter de gratouiller un loup-garou, mais l'effet produit dépendra de son état courant :

- S'il est dans l'état humain, il répondra : `"Hi! Hi!"`
- S'il est dans l'état d'un loup, il répondra : `"Grrr!....."`

On souhaite modéliser ce comportement sans se servir de champ booléen ni de `if...else`, uniquement en exploitant les propriétés de la liaison dynamique.

1. Définir une nouvelle classe abstraite `EtatLoupGarou`. Ajouter à cette classe deux méthodes abstraites :
   - `abstract EtatLoupGarou etatOppose ();`
   - `abstract void effetGratouille ();`
2. Définir deux extensions concrètes `EtatLoup` et `EtatHumain` de la classe précédente. L'implémentation de la méthode `etatOppose` de chaque classe devra renvoyer une nouvelle instance de l'autre[2] L'implémentation de la méthode `effetGratouille` devra afficher à l'écran la réponse aux gratouilles d'un loup-garou dans l'état correspondant.
3. Ajouter à la classe `LoupGarou` un champ de type `EtatLoupGarou`, initialisé dans le constructeur à une instance de `EtatHumain`.
4. Ajouter à la classe `LoupGarou` une méthode `seMetamorphose`. Cette méthode devra remplacer l'état courant d'un loup-garou par son état opposé : le nouvel état sera demandé à l'état courant via la méthode `etatOppose`.
5. Ajouter à la classe `LoupGarou` une méthode permettant de gratouiller un loup-garou. Son invocation devra demander à l'etat courant l'effet produit par cette gratouille.

2. Autre possibilité : chaque méthode peut renvoyer l'unique exemplaire d'une instance définie comme champ statique dans l'autre.