Le monde de l'objet : mise en pratique par le langage Java

Mutation d'objet

La mutation est un concept qui consiste à permettre à un objet d'alterner entre plusieurs comportements à l'exécution. Pourtant, si les comportements de cet objet se modifient, l'objet reste toujours à la même « adresse mémoire », c'est-à-dire qu'il ne doit pas être réalloué (avec un nouveau *new*) sinon ce ne serait plus le même objet.

La mutation consiste donc à trouver des mécanismes permettant de changer dynamiquement (i.e. à l'exécution) un comportement, c'est-à-dire que la réponse à un message ne sera plus la même.

On illustrera ces concepts ci-après au travers de la création d'un programme manipulant des personnages tels que des êtres humains, des loups, des chauves-souris, mais aussi des loup-garous et des vampires…

Exercice 1 : préambule et animaux

Créez un fichier *Entite.java*. Il contiendra une interface publique *Entite* qui déclarera trois méthodes :

- la méthode espece*()*, qui renvoie une chaîne de caractères ;
- la méthode *aimeLeJour()*, qui renvoie un booléen ;
- la méthode *mute()*, qui demande la transmutation d'une entité.

Créez ensuite trois classes publiques, dans trois fichiers séparés : *Humain*, *Loup* et *ChauveSouris*. Chacun renverra le nom de son espèce. L'*Humain* aime le jour, mais ni le *Loup*, ni la *ChauveSouris*. Enfin, aucune de ces trois classes ne fait quelque chose lorsqu'on lui demande de muter, donc le corps de la méthode d'instance *mute()* peut rester vide (mais la méthode doit exister).
Enfin, redéfinissez la méthode *toString*() de chacune de ces classes afin de pouvoir les passer directement en paramètre de *System.out.println*.

Testez vos codes dans une nouvelle classe publique *TestVivant*, qui contiendra par exemple le code suivant :

```
public static void main(String[] args)
{
	Entite[] tableauEntites = {new Humain(), new Loup(), new ChauveSouris()};

	for (int index = 0; index < tableauEntites.length; ++index)
	{
		System.out.println(tableauEntites[index]);
		tableauEntites[index].mute();
		System.out.println(tableauEntites[index]);
		System.out.println("");
	}
}
```

Exercice 2 : rationalisation

Si vous ne l'avez pas déjà fait, reprenez le code ci-dessus afin d'éliminer les redondances entre *Humain*, *Loup* et *ChauveSouris*. Astuce : pensez à la notion de classe abstraite, qui pourra vous aider fortement dans ce cas-là, et qui vous conduira sûrement à l'écriture d'une classe *Animal*.

Exercice 3 : de nouvelles entités

On veut maintenant ajouter deux nouvelles entités, qui ne sont pas de type *Animal*. Ce sont le *LoupGarou* et le *Vampire*.

Le *LoupGarou* alterne entre le comportement d'un *Humain* et d'un *Loup*, et ce à chaque fois que *mute()* est appelé. Il démarre sur un comportement d'*Humain*, et il ne devrait pas être possible de le différencier d'un *Humain* lorsqu'il adopte le comportement d'un *Humain*, et d'un *Loup* lorsqu'il adopte le comportement d'un *Loup*.

Le *Vampire* alterne entre le comportement d'un *Humain* et d'une *ChauveSouris*, et ce à chaque fois que *mute()* est appelé. Il démarre sur un comportement d'*Humain*. Pourtant, qu'il soit *Humain* ou *ChauveSouris*, il ne répondra jamais qu'il aime le jour…

Transformez la ligne allouant le tableau d'*Entite* en la ligne suivante, ce qui devrait suffire à vous permettre de tester votre nouveau code :

```
Entite[] tableauEntites = {new Humain(), new Loup(), new ChauveSouris(),
new LoupGarou(), new Vampire()};
```

Exercice 4 : rationalisation bis

De même qu'à l'exercice 2, peut-être pouvez-vous éviter la duplication de code entre *LoupGarou* et *Vampire* ?

Modifiez votre code, et testez-le comme précédemment.