IUT de Paris Descartes – Base de la Programmation Objet

Travaux Pratiques – Sujet n°10

Objectifs

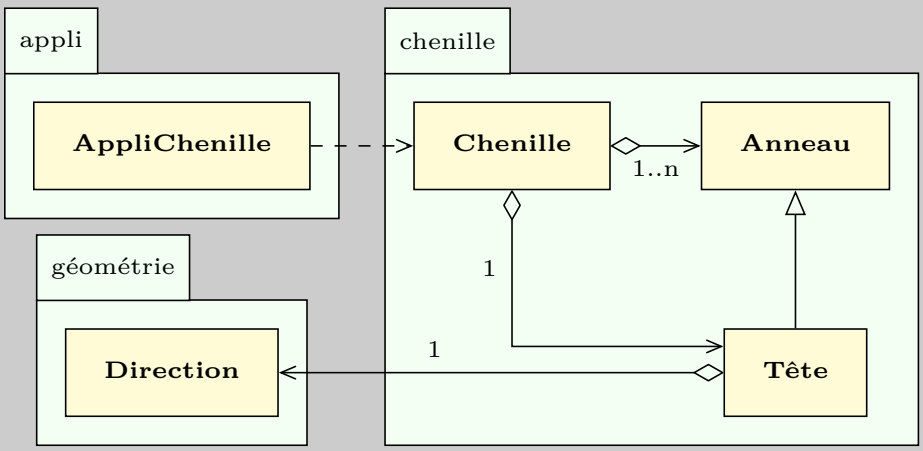
- Retravailler le code source non pas pour ajouter une fonctionnalité supplémentaire au logiciel mais pour améliorer sa qualité (sa lisibilité, simplifier sa maintenance, ou changer sa généricité).
- Favoriser le polymorphisme en évitant des dépendances vers les sous-classes d'une hiérarchie.
- Externaliser la création d'objet dans une fabrique statique.

Les chenilles revisitées

Lors d'une séance de travaux dirigés précédente, nous avons développé une application permettant l'animation des déplacements de chenille. L'architecture de notre application mettait en œuvre essentiellement l'héritage. Le but de cette séance est de retravailler notre code de manière à rendre la classe `Chenille` le plus possible indépendante de l'arbre d'héritage des anneaux. Cela va être réalisé par transformations successives. Le code résultant de la séance précédente est fournie sur le serveur commun.

1. Créez un projet, importez les fichiers sources nécessaires et étudiez les relations de dépendance (héritage, aggrégation et utilisation) qui lient les différentes classes. Dessinez le diagramme de classe (sans détailler les attributs et les méthodes des classes mais en vous focalisant sur leurs dépendances).

   Solution: Ces relations sont synthétisées dans le diagramme UML suivant.

2. L'objectif est de rendre la classe utilisatrice (la classe `Chenille`) indépendante des sous-classes de la hiérarchie (la classe `Tête`) de façon à ce que si cette hiérarchie est enrichie la classe cliente puisse en profiter sans être modifiée.

   La première chose à faire consiste à réorganiser les paquetages de l'application. Les classes `Anneau` et `Tête` (la hiérarchie) doit disposer de son propre paquetage. Créez un paquetage nommé `anneaux` et déplacez-y les deux classes.

3. Nous voulons que la classe `Chenille` n'emploie plus directement la classe `Tête` mais uniquement la classe `Anneau`. Pour cela, il est nécessaire que la chenille sache déplacer et dessiner les anneaux quelque soit leur type. Ces deux méthodes doivent avoir des prototypes communs dans les deux classes (l'une spécialisant l'autre).

   Étudiez les classes `Anneau` et `Tête` et déterminer les méthodes manquantes, les données qu'elles nécessitent et la façon d'unifier les prototypes des méthodes. Implémentez votre solution.

   Solution: Les deux classes partagent déjà la méthode `dessiner` et la classe `Tête` dispose d'une méthode `déplacer` ayant le prototype suivant : **`void`** `déplacer(`**`int`** `xMax, `**`int`** `yMax)`. La classe `Anneau` ne dispose pas de méthode pour le déplacement.

   Pour qu'un anneau, sache se déplacer par lui même, il est nécessaire qu'il connaisse la position de l'anneau qui le précède. Or, il ne dispose pas de cette connaissance. Deux solutions sont envisageables :

   1. Soit, nous faisons en sorte que tout anneau connaisse (dès la construction) l'anneau qui le précède,
   2. soit, cette connaissance doit être fournie par celui qui invoque la méthode de déplacement (la chenille).

   Les deux solutions sont égales. La tête n'a pas d'anneau qui le précède mais, dans les deux solutions, ce cas devra être traité à part. Parce qu'elle est plus facile à mettre en oeuvre nous choisissons la seconde solution.

   Au minimum, le prototype de la méthode de déplacement de la classe `Anneau` est donc le suivant : **`void`** `déplacer(Anneau précédent)`. Pour que cette méthode puisse être spécialisée dans la classe `Tête`, il faut y ajouter les paramètres déjà présents dans la méthode `déplacer` de cette classe. Ce qui donne : **`void`** `déplacer(Anneau précédent, `**`int`** `xMax, `**`int`** `yMax)`. Les anneaux n'emploieront que le premier paramètre et la tête uniquement les deux derniers.

   Seules les modifications faites dans les deux classes sont reportées dans ce qui suit :

```java
// Anneau.java
package anneaux;

public class Anneau {
  ...
  // ajout
  public void déplacer(Anneau précédent, int xMax, int yMax) {
    placerA(précédent.getX(), précédent.getY());
  }
}
```

```java
// Tête.java
package anneaux;

public class Tête extends Anneau {
  ...
  // modification : ajout d'un paramètre et du assert
  public void déplacer(Anneau précédent, int xMax, int yMax) {
    assert (précédent == null);
    if (getX() == 0 || getX() == xMax ||
      ...
  }
}
```

4. Corrigez le code de la méthode `déplacer` de la classe `Chenille`.

Solution:

```
public class Chenille {
  ...
  public void déplacer(int xMax, int yMax) {
    for (int i = anneaux.length - 1; i > 0; --i)
      anneaux[i].déplacer(anneaux[i - 1], xMax, yMax);
    anneaux[0].déplacer(tête, xMax, yMax);
    tête.déplacer(null, xMax, yMax);
  }
  ...
}
```

5. Faites en sorte que la classe `Chenille` ne déclare plus d'attribut de type `Tête` mais uniquement un tableau de `NbAnneaux` + 1 (pour la tête) références de type `Anneau`.

Solution:

```
public class Chenille {
  private Anneau[] anneaux;

  public Chenille(int nbAnneaux, int x, int y) {
    anneaux = new Anneau[nbAnneaux + 1]; // + 1 pour la tête
    anneaux[0] = new Tête(x, y);
    for (int i = 1; i < anneaux.length; ++i)
      anneaux[i] = new Anneau(x - i, y);
  }

  public void déplacer(int xMax, int yMax) {
    // de la queue à la tête
    for (int i = anneaux.length - 1; i > 0; --i)
      anneaux[i].déplacer(anneaux[i - 1], xMax, yMax);
    anneaux[0].déplacer(null, xMax, yMax);
  }

  public void dessiner(char[][] t) {
    // de la queue à la tête
    for (int i = anneaux.length - 1; i >= 0; --i)
      anneaux[i].dessiner(t);
  }
}
```

6. Seul le constructeur de la classe `Chenille` emploie encore directement la classe `Tête` lors de la création des objets la composant. Une solution courante consiste à déléguer (sous-traiter) la création des objets à une classe tiers.
   Créez une classe nommée `FabriqueAnneau` dans le paquetage `anneaux`. Cette classe disposera d'une seule méthode de classe (i.e. statique) retournant un anneau. Cet anneau sera créé aux coordonnées passées en paramètre. La méthode recevra aussi un numéro qui lui permettra de décider du type de l'anneau devant être créé. Un numéro égal à 0 donnera lieu à la création d'une tête. Tout autre numéro verra la création d'un simple anneau.

Solution:

```
// FabriqueAnneau.java
package anneaux;

public class FabriqueAnneau {
```

```
  public static Anneau getAnneau(int num, int x, int y) {
    if (num == 0)
      return new Tête(x, y);
    else
      return new Anneau(x, y);
  }
}
```

7. Employez cette nouvelle classe dans le constructeur de la classe `Chenille`.

Solution:

```
// Chenille.java
package chenille;

import anneaux.Anneau;
import anneaux.FabriqueAnneau;

public class Chenille {
  ...
  public Chenille(int nbAnneaux, int x, int y) {
    anneaux = new Anneau[nbAnneaux + 1]; // + 1 pour la tête
    for (int i = 0; i < anneaux.length; ++i)
      anneaux[i] = FabriqueAnneau.getAnneau(i, x - i, y);
  }
  ...
}
```

8. Développez une nouvelle sous-classe d'`Anneau` nommée `AnneauClignotant`. Ces anneaux se déplacent comme des anneaux standard mais se dessinent une fois sur deux en majuscule et l'autre fois en minuscule (voir les méthodes de classe de `java.lang.Character`).

Solution:

```
// AnneauClignotant.java
package anneaux;

public class AnneauClignotant extends Anneau {
  private boolean clignó;

  public AnneauClignotant(int x, int y) {
    super(x, y);
    clignó = false;
  }

  @Override
  public char getSymbole() {
    clignó = !clignó;
    char c = super.getSymbole();
    if (clignó)
      return Character.toUpperCase(c);
    return Character.toLowerCase(c);
  }
}
```

9. Modifiez la fabrique d'anneau de façon à ce qu'un anneau sur trois soit un clignotant. Relancez le programme.

Solution:

```
// FabriqueAnneau.java
public class FabriqueAnneau {
  public static Anneau getAnneau(int num, int x, int y) {
    if (num == 0)
      return new Tête(x, y);
    else if (num % 3 == 0)
      return new AnneauClignotant(x, y);
    else
      return new Anneau(x, y);
  }
}
```

10. Dessinez le nouveau diagramme de classe (sans détailler les attributs et les méthodes des classes mais en vous focalisant sur leurs dépendances).

Solution: