Université Paris Cité

Compléments en Programmation Orientée Objet
TP nº 4 : programmation à l'interface inversion de dépendance, fabriques abstraites, adaptateurs
(Correction)

1 Programmer à l'interface

Exercice 1 : Tris

Le tri à bulles est un algorithme classique permettant de trier un tableau. Il peut s'écrire de la façon suivante en Java :

```java
    static void triBulles(int tab[]) {
        boolean change = false;
        do {
            change = false;
            for (int i=0; i<tab.length - 1; i++) {
            if (tab[i] > tab [i+1]) {
                int tmp = tab[i+1];
                tab[i+1] = tab[i];
                tab[i] = tmp;
                change = true;
            }
            }
        } while (change);
    }
```

Cette implémentation du tri à bulles permet de trier un tableau d'entiers. Maintenant on veut pouvoir utiliser le tri à bulles sur tout autre type de données représentant une suite (séquence) d'objets comparables.
Pour cela, il faut programmer à l'interface. Ainsi, notre tri sera programmé pour les interfaces suivantes :

```java
public interface Comparable {
    public Object value(); // renvoie le contenu
    public boolean estPlusGrand(Comparable i);
}

public interface Sequencable {
    public int longeur(); // Renvoie la longueur de la sequence
    public Comparable get(int i); // Renvoie le ieme objet de la sequence
    public void echange(int i, int j); // Echange le ieme object avec le jieme objet
}

```

1. Écrivez une méthode `affiche()` dans l'interface `Sequencable` permettant d'afficher les éléments de la séquence du premier au dernier. (Utilisez la fonction `toString()` de Object.)

   Correction :

   ```java
       default void affiche() {
           String s=contenu(1).toString();
           for(int i=2; i< longeur(); i++) {
               s+=", " + contenu(i).toString();
           }
           System.out.println("[" + s + "]");
       }

   ```

2. Écrivez une méthode `triBulle` dans l'interface `Sequencable` qui effectue un tri à bulles sur la séquence.

Correction :

```
default void triBulle() {
    boolean change = false;
    do {
        change = false;
        for (int i = 1; i < longeur()-1; i++) {
            if ( contenu(i).estPlusGrand(contenu(i+1))) {
                echange(i, i+1);
                change = true;
            }
        }
    } while (change);
}

```

3. Écrivez une classe `MotComparable` représentant un mot et implémentant l'interface `Comparable` de tel sorte que `estPlusGrand(Comparable i)` :
   - quitte sur une exception (`throw new IllegalArgumentException();`) si `i.value()` n'est pas un sous-type de `String`,
   - retourne vrai si le contenu est plus grand lexicographiquement que `i.value()`, faux sinon.

   N'oubliez pas les constructeurs () et la méthode `toString()`.

   Correction : Voir le fichier « MotComparable.java ».

4. Écrivez une classe `SequenceMots` qui représente une séquence de `MotComparable` et qui implémente `Sequencable`.
   Écrivez un constructeur prenant un tableau de `String`.

   Correction : Voir le fichier « SequenceMots.java ».

5. Testez votre code.
   Vous pouvez passer en paramètre un tableau de chaînes aléatoires générées avec l'instruction `Integer.toString((int)(Math.random()*50000))` (ou utilisez un des générateurs de l'exercice précédent).

Exercice 2 : Générateurs de nombres

Attention : pour faire cet exercice, il faut savoir implémenter une interface (`Generateur`) et comprendre le polymorphisme par sous-typage (toute méthode retournant un `Generateur` a le droit de retourner une instance de classe implémentant `Generateur`).

Objectif : écrire la classe-bibliothèque `GenLib`, permettant de créer des générateurs de toute sorte (entiers au hasard, suites arithmétiques, suites géométriques, fibonacci, etc.), sans pour autant fournir d'autres types publics que la classe `GenLib` elle-même ainsi qu'une interface `Generateur` dans le package que vous livrez à vos utilisateurs.
Pour commencer, fixons l'interface générateur :

```
interface Generateur { int suivant(); }
```

Méthode : utiliser le schéma suivant : `GenLib` (classe non instanciable) contient une série de fabriques statiques permettant de créer les générateurs. Chaque appel à une fabrique instancie une classe imbriquée en utilisant les paramètres passés.
Attention : la classe `GenLib` n'implémente pas elle-même l'interface `Generateur` (ça n'aurait pas de sens, puisqu'elle n'est pas instanciable). Ses méthodes ne renvoient pas de `int` !

Exemple d'utilisation : pour afficher les 10 premiers termes de la suite de Fibonacci

```
Generateur fib = GenLib.nouveauGenerateurFibonacci();
for (int i = 0; i < 10; i++) System.out.println(fib.suivant());
```

Questions :

1. Programmez les méthodes statiques permettant de créer les générateurs suivants :
   - générateur d'entiers aléatoires (compris entre 0 et $m-1$, m étant un paramètre)
   - suite arithmétique : $0, r, 2r, 3r, ...$ (r étant un paramètre)
   - suite géométrique : $1, r, r^2, r^3, ...$ (r étant un paramètre)
   - suite de Fibonnacci : $1, 1, 2, 3, 5, 8, 13, ...$

   Contrainte : le type de retour doit être `Generateur` pour toutes les fabriques !

   Variez les techniques : montrez un exemple pour chaque genre de classe imbriquée (membre statique, membre non statique, locale, anonyme... mais une des 4 possibilités ne peut pas être utilisée ici, laquelle ?). Si vous vous rappelez comment on utilise les lambda-expressions, tentez aussi cette approche pour implémenter `Generateur` sans définir de classe.

Correction :

```
public class Generateurs {
       //avec le constructeur pas testé:
       private  Generateurs(){};

    interface Generateur {
        int suivant();
    }

    public static Generateur nouveauGenerateurEntiersAleatoires(int m) {
        return new Generateur() {
            @Override
            public int suivant() {
                return (int) (m * Math.random());
            }
        };
    }

    public static Generateur nouveauGenerateurGeometrique(int m) {
        class GenerateurGeometrique implements Generateur {
            int prochain = 1;

            @Override
            public int suivant() {
                int courant = prochain;
                prochain *= m;
                return courant;
            }
        }
        return new GenerateurGeometrique();
    }

    public static Generateur nouveauGenerateurArithmetique(int m) {
        class GenerateurArithmetique implements Generateur {
            int prochain = 0;

            @Override
            public int suivant() {
                int courant = prochain;
                prochain += m;
                return courant;
            }
        }
        return new GenerateurArithmetique();
    }

    static class Fibo implements Generateur {
        int eltSuivant = 1, eltCourant = 0;
```

```
48
49              @Override
50              public int suivant() {
51                  int nouveau = eltSuivant + eltCourant;
52                  eltCourant = eltSuivant;
53                  eltSuivant = nouveau;
54                  return eltCourant;
55              }
56          }
57
58          public static Generateur nouveauGenerateurFibonacci() {
59              return new Fibo();
60          }
61
62          public static void main(String args[]) {
63              Generateur fibo = nouveauGenerateurFibonacci();
64              for (int i = 0; i < 10; i++)
65                  System.out.println(fibo.suivant());
66          }
67      }
68
```

Correction : Réécriture du générateur aléatoire avec lambdas :

```
1           public static Generateur nouveauGenerateurEntiersAleatoires(int m) {
2               return () ->  (int) (m * Math.random());
3           }
4
```

On ne peut pas écrire les autres exemples de cette façon, car la syntaxe lambda-expression ne permet que de définir la méthode (unique) de l'interface implémentée par la classe anonyme. En outre, on ne peut pas ajouter/utiliser d'attribut. Donc cette syntaxe ne permet pas d'écrire un générateur où l'on a besoin de sauvegarder un état.

Correction : Pour répondre sur les différerents genres de classes imbriquées. Peuvaient convenir :

(a) les classes membres statiques (elles marchent comme les classes "normales", de premier niveau... qui pourraient d'ailleurs aussi convenir ici, bien qu'on perde l'avantage de l'encapsulation)

(b) les classes locales (et leur variante anonyme) : elles peuvent être créées dans toute méthode.

Pour utiliser les classes membres non statiques, il faudrait connaître une instance englobante. Or le genre de classe/bibliothèque que nous programmons n'est habituellement pas destinée à être instanciée.
Observons tout de même que les méthodes demandées sont de toute façon statiques et ne connaissent donc pas d'instance englobante implicite (`this`). La seule façon d'accéder à une classe membre non statique serait de fournir une instance explicite d'une façon où d'une autre (paramètre de la méthode, ou bien attribut)...

2. Pour aller plus loin, optimisez les fabriques de suites arithmétiques et géométriques pour qu'elles retournent un générateur constant (instance d'une classe spécialisée à cet effet) quand, respectivement, $r = 0$ ou $r = 1$ (on évite ainsi que la méthode `suivant` fasse une addition ou une multiplication inutile, vu qu'elle retourne toujours la même valeur). Pour les autre valeurs de r, on continue de faire comme avant.
3. Écrivez une méthode `int somme(Generateur gen, int n)` qui retourne la somme des `n` prochains termes du générateur `gen`.

4. Écrivez un `main()` qui demande à l'utilisateur de choisir entre les différents types de suite (et éventuellement d'entrer un paramètre), puis instancie le générateur de suite correspondant et en affiche ses 10 premiers termes et la somme des 5 suivants.
5. Maintenant vous vous mettez dans la peau de l'utilisateur de `GenLib`.

   Vous voulez utiliser la méthode `somme` pour calculer la somme des termes d'une collection Java (implémentant `java.util.Collection`)... qui évidemment n'implémente pas `Generateur`.

   Écrivez un adaptateur de `Collection` à `Generateur`, puis un programme utilisant l'adaptateur pour afficher la somme des termes de la liste `List.of(56, 23, 78, 64, 19)`.

2 Inversion de dépendance

Exercice 3 : Utilisation d'une bibliothèque pratiquant l'inversion de dépendance

Téléchargez le zip de l'exercice sur Moodle et décompressez le dans votre dossier de travail.
Vous trouverez une bibliothèque déjà programmée dans le sous-dossier/package `stats`.

1. D'abord, écrivez une classe `List2DataSeriesAdapter` qui implémente l'interface fournie `stats.DataSeries` en utilisant les éléments d'une liste (`java.util.List`) passée en paramètre. Comme son nom l'indique, cette classe met en œuvre le patron de conception Adaptateur.
2. Écrivez une méthode `main` dans une autre classe, qui affiche la moyenne et l'écart type de la liste `[64.51, 138.89, -25.5, 22.87]` en utilisant les outils de la classe fournie, `stats.Stats`, sur une instance de `List2DataSeriesAdapter`.

Exercice 4 : Écriture d'une bibliothèque pratiquant l'inversion de dépendance

Téléchargez le zip de l'exercice sur Moodle et décompressez le dans votre dossier de travail.
Vous trouverez un programme dans le sous-dossier/package `client`.
Il s'agit d'un programme exécutable qui ne fonctionne pas en l'état, car il dépend de la bibliothèque du package `logger` qui n'est pas encore programmée. Pour cause : ce sera à vous de le faire !
Implémentez la classe `logger.Logger` et les interfaces `logger.LogEntry` et `logger.LogEntryAbstractFactory` de la bibliothèque `logger` afin que le `main` de `client.main` fonctionne comme prévu, c'est à dire comme dans l'exemple d'éxécution ci-dessous :

```
> treu
Réessayer !
> true
> 56
> 6546
> print
true,
56,
6546,
> prettyprint
Booléen : true;
-------------
Entier : 56;
-------------
Entier : 6546;
-------------
>

Process finished with exit code 0
```

Explication : la classe `logger.Logger` stocke chaque valeur fournie par l'utilisateur dans le `main` dans une nouvelle instance appropriée de `logger.LogEntry`, qui est aussitôt ajoutée au journal (une liste).
Quand l'utilisateur demande un affichage du journal, l'instance de `logger.Logger`, lit la liste et demande l'affichage approprié (simple ou mis en page) pour chaque entrée sauvegardée.
Comme son nom l'indique, l'interface `logger.LogEntryAbstractFactory` implémentée par le client suit le patron de conception Fabrique Abstraite.