TP °4 : programmation à l'interface inversion de dépendance, fabriques abstraites, adaptateurs

I) Programmer à l'interface

Exercice 1 : 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 :

```
1 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

```
1 Generateur fib = GenLib.nouveauGenerateurFibonacci();
2 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 : $0, r, 2r, 3r, \ldots$ (r étant un paramètre)
   - suite géométrique : $1, r, r^2, r^3, \ldots$ (r étant un paramètre)
   - suite de Fibonacci : $1, 1, 2, 3, 5, 8, 13, \ldots$

   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.
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 autres 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.

II) Inversion de dépendance

Exercice 2 : 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 3 : 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'exécution 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.

III) Les dessous du transtypage

Exercice 4 : Transtypages primitifs

Voici un programme (`TranstypagesPrimitifs.java` sur Moodle) :

```java
public class TranstypagesPrimitifs {
   public static void main(String[] args) {
      int vint = 1234567891;
      short vshort = 42;
      float vfloat = 9.2E11f;
      System.out.println("vint = " + vint +
         ", vshort = " + vshort +
         ", vfloat = " + vfloat);
   }
}
```

1. Compilez et exécutez ce programme (assurez-vous de comprendre la notation `9.2E11f`).
2. Nous allons regarder superficiellement le code-octet produit : dans un terminal, allez dans le répertoire où se trouve `TranstypagesPrimitifs.class` et tapez la commande "`javap -c -v TranstypagesPrimitifs`". Le code-octet apparaît ainsi sous une forme désassemblée quasi lisible. Nous nous intéresserons en particulier au début de la partie `Code :`, qui correspond à la déclaration et l'initialisation de nos trois variables. On peut repérer l'appel à l'instruction suivante, `println`, par l'instruction **getstatic** dans le code-octet.

   Il n'y a donc que 6 ou 7 lignes à regarder. Constatez que certaines variables sont initialisées par une séquence d'instructions comme : **bipush** `42;`**istore_2**, alors que d'autres ont la séquence **ldc** suivie de **istore** ou **fstore** (le i ou le f désigne clairement un type)
3. Nous allons nous intéresser à la façon dont sont fait les transtypages. Ajoutez une ligne avant l'instruction d'affichage : `vint=vshort;` et interpréter les opérations `load,store` qui apparaissent.

   Avec les 3 variables présentes il y a théoriquement 6 transtypages, certains qu'il faut rendre explicites. Essayez les tous et complétez le tableau ci-dessous avec vos remarques.

   Le tableau :

=	vint	vshort	vfloat
vint	XXX		
vshort		XXX	
vfloat			XXX

   Relevez, notamment, dans chaque cas :
   - si ça compile directement, s'il ajouter un *cast* explicite, etc. ;
   - la nature des instructions ajoutées dans le code-octet (notez que les instructions de la forme `f2i` expriment un changement de type) ;
   - l'affichage produit après conversion.

Exercice 5 : Transtypages d'objets (sur machine)

Même exercice que le précédent mais sur le programme suivant :

```
1  public class TranstypagesObjets {
2     public static void main(String[] args) {
3        Object vObject = new Object();
4        Integer vInteger = 42;
5        String vString = "coucou";
6        System.out.println("vObject = " + vObject +
7               ", vInteger = " + vInteger +
8               ", vString = " + vString);
9     }
10 }
```

Différence, vous ne verrez plus l'ajout de l'instruction **u2t** mais parfois celle de **checkcast**. Dans quels cas ?
Dans certains cas vous aurez eu besoin, pour compiler, d'un *cast* explicite. Lesquels ? Est-ce les-mêmes que dans la question précédente ?
Dans certains cas, le programme quittera sur `ClassCastException`, lesquels ?

Exercice 6 : Transtypages mixtes (sur machine)

Faites le même travail sur le programme suivant. Remarquez les instructions, insérées par le compilateur, qui correspondent au boxing et à la vérification de types.

```
1  public class TranstypagesMixtes {
2     public static void main(String[] args) {
3        Object vObject = Integer.valueOf(9);
4        Integer vInteger = 42;
5        int vint = 111;
6        System.out.println("vObject = " + vObject +
7               ", vInteger = " + vInteger +
8               ", vint = " + vint);
9
10    }
11 }
```