UE Conception Orientée Objet

Devoir Surveillé

Mardi 18 décembre 2012 – 8h-11h
Copies des diapositives de cours autorisées
Fiches “design patterns” distribuées en TD autorisées
Une feuille recto-verso de notes personnelles
Dictionnaire de langue (papier ou électronique “dédié”) autorisé
Tout autre document interdit

Les exercices sont indépendants.
La clarté des réponses sera prise en compte dans l'évaluation, en particulier pour les diagrammes UML dans lesquels apparaitront toujours :

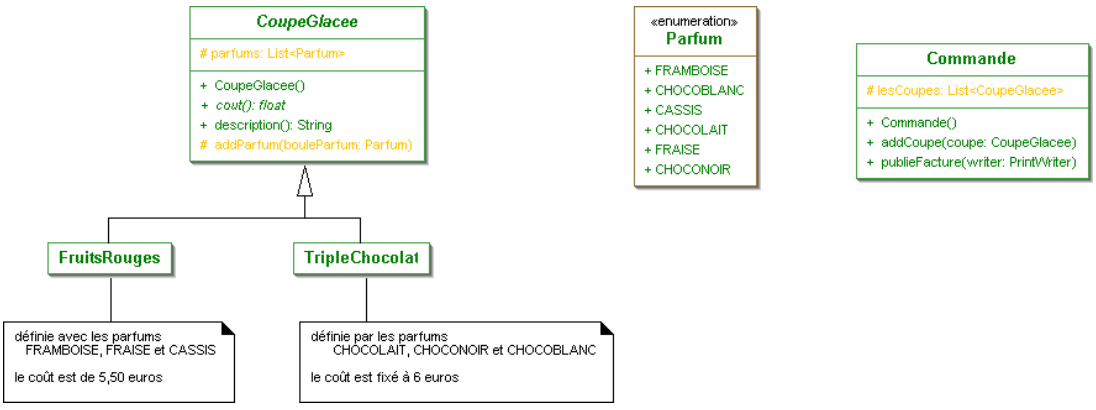
- *les liens d'héritage/implémentation entre types*
- *les noms et types de tous les attributs, ainsi que leur visibilité,*
- *les méthodes et constructeurs avec leurs paramètres et leurs types ainsi que les types des valeurs de retour*

Soignez la présentation de ces diagrammes en évitant autant que possible les ratures !

A titre purement indicatif, durée estimée et suggérée de traitement des exercices :
lecture du sujet : 15mn – exercice 1 : 1h05 – exercice 2 : 50mn – exercice 3 : 50mn

Exercice 1 : Glaces

Dans le cadre de l'informatisation d'un glacier, il est nécessaire de modéliser les différentes coupes glacées que celui-ci peut proposer. Cela permet notamment la gestion des commandes et factures des clients.
Dans une première version, seules des coupes glacées composées de plusieurs boules de différents parfums sont gérées. Dans cette première version, on trouve le diagramme de classes suivant. On peut imaginer ajouter d'autres coupes glacées :



Une partie du code associé est fourni en Annexe.

La méthode `publieFacture` de la classe `Commande` permet d'afficher une facture sur le flux de sortie voulu et défini par le `PrintWriter` passé en paramètre. En voici le code :

```
public void publieFacture(PrintWriter writer) {
   float total = 0;
   for (CoupeGlacee coupe :  this.lesCoupes) {
     writer.println(coupe.description()+" + "+coupe.cout());
     total = total + coupe.cout();
   }
   writer.println("   TOTAL : "+total);
}
```

Pour une commande composée de deux coupes “fruits rouges” et d'une coupe “triple chocolat”, la facture ainsi publiée ressemble à :

```
Coupe fraise framboise cassis + 5,50
Coupe fraise framboise cassis + 5,50
Coupe chocolat blanc chocolat au lait chocolat noir + 6,00
   TOTAL : 17,00
```

Q 1 . On décide d'appliquer le design pattern singleton aux classes `FruitsRouges` et `TripleChocolat` .

Q 1.1. Qu'est-ce qui justifie ce choix ?

Q 1.2. Donnez le code java de la classe `FruitsRouges`.

Evolution du logiciel. Suite à de nombreuses demandes de ses clients, le glacier décide qu'il sera possible pour chaque client de personnaliser ses coupes par l'ajout d'un ou plusieurs agréments (ou "topping") tels que de la *crème chantilly*, du *coulis chocolat*, du *coulis de fraise* etc. Ceci évidemmment contre un modique surcoût dépendant de l'ingrédient. La description de la coupe glacée consommée par le client doit bien entendu se trouver modifiée en conséquence.
Voici quelques toppings possibles, leur surcoût et la description associée :

topping	*surcoût*	*description*
chantilly	0,50	"chantilly"
sauce chocolat	0,70	"et sa délicieuse sauce chocolat"
coulis fraise	1	"au coulis de fraises fraiches"

Evidemment il est possible pour les gourmands de cumuler plusieurs toppings sur une même coupe glacée[1]. Dans ce cas les surcoûts se cumulent et les descriptions s'enrichissent en conséquence.
Il est évidemment nécessaire que cette évolution soit prise en compte au niveau des commandes et factures.
Voici donc un exemple de facture publiée, via `publieFacture`, dans cette seconde version :

```
Coupe fraise framboise cassis chantilly + 6,00
Coupe fraise framboise cassis chantilly au coulis de fraises fraiches + 7,00
Coupe chocolat blanc chocolat au lait chocolat noir  + 6,00
Coupe chocolat blanc chocolat au lait chocolat noir et sa délicieuse sauce chocolat chantilly  + 7,50
   TOTAL :   26,50
```

Q 2 . Proposez une solution élégante à l'ajout des classes permettant la gestion de coupes glacées avec "toppings" de la seconde version de l'application de gestion du glacier.

Votre proposition doit permettre

- de continuer à utiliser les classes existantes (`Commmande`, `CoupeGlacee`, etc.) **sans les modifier**.
- d'ajouter de nouveaux toppings sans modification d'une classe existante (respect du principe *ouvert-fermé*), ce qui n'est par exemple pas possible pour l'ajout de nouveaux parfums avec la conception actuelle à cause de l'`enum`.

Q 2.1. Formulez votre proposition sous la forme d'un diagramme UML qui vient compléter celui proposé ci-dessus.

Q 2.2. Donnez **tout le nouveau code** Java nécessaire à l'intégration du topping "*Chantilly*" dans l'application (classes et interfaces).

Q 2.3. Vous justifierez votre respect du principe ouvert-fermé en indiquant brièvement mais clairement ce qu'il faut faire pour ajouter un nouveau topping (*noisettes caramélisées* par exemple), sans donner de code.

Q 2.4. Donnez quelques lignes de code permettant de créer "*une Coupe chocolat blanc chocolat au lait chocolat noir et sa délicieuse sauce chocolat chantilly*".

Exercice 2 : Décryptage
On s'intéresse au décryptage de messages (cryptés). On suppose qu'existe une classe `Message` pour représenter un message et que celle-ci dispose d'une méthode :

```
public boolean estCrypte()
```

qui permet de savoir si un message est sous forme cryptée (valeur de retour *true*) ou décryptée (valeur de retour *false*).

On définit un processus de décryptage comme un processus prenant en entrée un message m_{in} (de type `Message`) et le transforme pour fournir en sortie un autre message m_{out} si l'algorithme a réussi l'opération de décryptage (c'est-à-dire si m_{out} est décrypté) ou lever une exception `EchecDecryptageException` sinon.

On suppose que l'on dispose de plusieurs processus de décryptage simples qui utilisent des algorithmes différents et que nous appellerons pour simplifier `Algo1`, `Algo2` et `Algo3`. Il peut en exister d'autres.

On appelle *multi-décryptage* un processus de décryptage qui s'appuie sur plusieurs processus de décryptage existants et les essaie successivement sur un message tant que celui-ci n'est pas décrypté (il est évidemment possible qu'aucun algorithme n'y parvienne). La liste des algorithmes de décryptage gérés par un tel processus n'est pas figée.

[1] y compris, soyons fous, plusieurs fois le même.

Q 1 . Faites une proposition de conception pour gérer tous ces processus de décryptage (simples et multiples). Vous donnerez votre réponse sous la forme d'un diagramme UML détaillé.

Q 2 . Donnez le code Java de la classe qui gère le processus de *multi-décryptage*.

Q 3 . Donnez le code Java de la classe `EchecDecryptageException`.

Exercice 3 : Refactoring

On découvre (avec stupeur) le code suivant :

--- Usine.java ---

```
package design;
import java.util.*;
public class Usine {
   private String marque;
   public Usine(String marque) {
      this.marque = marque;
   }
   public Cafetiere fabrique() {
      Cafetiere result;
      Date maintenant = Calendar.getInstance().getTime();
      if (this.marque.equals("seb")) {
         result = new Cafetiere("seb",maintenant);
      }
      else if (this.marque.equals("nespresso")) {
         result = new Cafetiere("nespresso",maintenant);
      }
      else result = null;
      return result;
   }
}
```

--- Machine.java ---

```
package design;
public enum Machine { FILTRE, CAPSULE; }
```

--- Cafetiere.java ---

```
package design;
import java.util.Date;
public class Cafetiere {
   private String marque;
   private Date dateFabrication;
   private Machine typeMachine;
   public Cafetiere(String marque, Date dateFabrication) {
      this.marque = marque;
      this.dateFabrication = dateFabrication;
      this.initTypeMachine();
   }
   public Machine getTypeMachine() {
      return this.typeMachine;
   }
   public String slogan() {
      if (this.marque.equals("seb")) {
         return "Seb, c'est bien !";
      }
      else if (this.marque.equals("nespresso")) {
         return "What else ?";
      }
      else return "aucun";
   }
   protected void initTypeMachine() {
      if (this.marque.equals("seb")) {
         this.typeMachine = Machine.FILTRE;
      }
      else if (this.marque.equals("nespresso")) {
         this.typeMachine = Machine.CAPSULE;
      }
      else  this.typeMachine = null;
   }
   public String toString() {
      return "cafetière "+ this.marque+ " fabrication le :"+this.dateFabrication;
   }
```

```
}
```

--- Main.java ---

```
package design;
import java.util.*;
public class Main {
   public static void main(String[] args) {
      Usine u1 = new Usine("seb");
      Usine u2 = new Usine("nespresso");

      List<Cafetiere> lesCafetieres = new ArrayList<Cafetiere>();

      lesCafetieres.add(u1.fabrique());
      lesCafetieres.add(u2.fabrique());

      for(Cafetiere k : lesCafetieres) {
         System.out.println(k+"\n"+k.slogan());
      }
   }
}
```

Q 1 . Quelle(s) critique(s) faites-vous de ce code sur le plan de sa conception objet ?

Q 2 . Faites une contre-proposition de conception corrigeant ces défauts. Donnez votre réponse sous la forme d'un diagramme UML détaillé.

Q 3 . En quoi votre proposition est-elle préférable à la version initiale ?

Q 4 . Donnez le code java correspondant à la gestion des objets *Usine* dans votre version.

Annexe

--- CoupeGlacee.java ---

```
package glace;
import java.util.*;
public abstract class CoupeGlacee {
    protected List<Parfum> parfums;
    public CoupeGlacee() {
       this.parfums = new ArrayList<Parfum>();
    }
    public String description() {
       String desc = "";
       for (Parfum parfum : this.parfums) {
           desc = desc + "␣" + parfum.toString();
       }
       return desc;
    }
    protected void addParfum(Parfum bouleParfum) {
       this.parfums.add(bouleParfum);
    }
    public abstract float cout();
}
```

--- Parfum.java ---

```
package glace;
public enum Parfum {
    FRAISE("fraise"), FRAMBOISE("framboise"), CASSIS("cassis"),
    CHOCOLAIT("chocolat␣au␣lait"), CHOCONOIR("chocolat␣noir"),
    CHOCOBLANC("chocolat␣blanc");

    private String description;
    private Parfum(String description) {
       this.description = description;
    }
    public String toString() {
       return this.description;
    }
}
```