POBJ – Rattrapage Janvier 2012

Durée : 2 heures Tous documents autorisés, appareils électroniques interdits

PROBLÈME : CRÊPES À LA DEMANDE

Le gérant d'une « crêperie à domicile » vous confie le développement du système de gestion de son établissement qui doit générer les menus, automatiser la prise de commandes depuis un poste internet distant, calculer le prix des commandes et émettre les factures. Les seules choses qui ne sont pas automatisées sont la confection des crèpes et la livraison à domicile.

Le « *business model* » de la crêperie est le suivant : le poste internet du client affiche un ensemble de « garnitures » (tomates, fromage, champignon, beurre, sucre, etc.) auxquelles sont associés des prix dépendant de la garniture. Le client peut composer les crêpes de son choix en associant toutes les garnitures qu'il désire à une crêpe de base (sans garniture). Dans sa commande, le client peut demander la même crêpe en plusieurs exemplaires (par exemple 3 crêpes chocolat anchois) et demande autant de crêpes différentes qu'il le souhaite.

Du point de vue du système, une crèpe possède juste un nom et un prix, on la définit avec l'interface suivante :

```
package pobj;
public interface Crepe {
	public double getPrix();
	public String getNom();
}
```

Barême classique à appliquer partout :

-25% s'il y a des petites erreurs de syntaxe : ;, {}, ()...

-25% pour les erreurs de visibilité (attributs public...)

-50% pour les fautes de prog.

On ne paye pas l'encre, il faut que ce soit (à peu près) juste pour avoir des points ;)

1 : Crêpes de base et garnitures

Q1.1 : Proposez dans un classe *CrepeBasique* une implantation pour les crêpes de base, dont le nom est simplement « Crepe » et le prix 1 euro.

```
package pobj;

public class CrepeBasique implements Crepe {
	public double getPrix() {
		return 1.0;
	}
	public String getNom() {
		return "Crepe";
	}
}
```

Evidemment, OK s'ils ont mis des attributs. Etre sans pitié pour les petites fautes.
Je ne sanctionne pas un final non demandé.

Les garnitures sont des produits que l'on ajoute aux crêpes. De même que les crêpes, elles sont caractérisées par un nom et un prix. Le nom et le prix ne sont pas susceptibles d'être modifiés dans l'application.

Q1.2 : Proposez une implantation pour les garnitures dans une classe *Garniture.*

```
package pobj;

public class Garniture {
	private final String nom;
	private final double prix;
	public Garniture(String nom, double prix) {
		this.nom = nom;
		this.prix = prix;
	}
	public String getNom() {
		return nom;
	}

	public double getPrix() {
		return prix;
	}
}
```

-50% pour implements Crepe (une garniture n'est pas une crêpe).
-25% s'il manque les final, c'est explicite dans l'énoncé.

2 : Crêpes garnies

On peut mettre n'importe quelle garniture dans n'importe quelle crêpe. Le nom d'une crêpe garnie est le nom de la crêpe de base suivi de la liste des garnitures (par exemple : « crêpe chocolat anchois »). Son prix est le prix de la crêpe de base plus le prix de ses garnitures.

Q2.1 : Dans le package *pobj*, proposez l'implantation des crêpes garnies dans une classe *CrepeGarnie* en utilisant le *Design Pattern Decorator.* Vous ferez notamment appel à une classe abstraite *CrepeDecoree* pour représenter un décorateur abstrait. Outre son constructeur, votre classe *CrepeGarnie* fournira une méthode *Crepe garnirCrepe(Crepe c, Garniture g)* qui ajoute une garniture à une crêpe.

Séparer en 2 notes de 100% : Q2.1.1 = CrepeDecoree, Q2.1.2 = CrepeGarnie

```
package pobj;

import java.util.Iterator;

public abstract class CrepeDecoree implements Crepe {
	private Crepe crepe ;

	public CrepeDecoree(Crepe crepe) {
		this.crepe = crepe;
	}

	public double getPrix(){
```

```
	return crepe.getPrix();
	}
	public String getNom() {
	return crepe.getNom();
	}

	public Crepe getCrepe(){
		return crepe;
	}

}
```

-50% s'il pas de constructeur qui prend une Crepe.

```
public class CrepeGarnie extends CrepeDecoree {
	private Garniture garniture ;

	public CrepeGarnie(Crepe crepe, Garniture garniture) {
		super(crepe);
		this.garniture = garniture;
	}
	public double getPrix(){
	return super.getPrix()+garniture.getPrix();
	}
	public String getNom() {
	return super.getNom()+" "+garniture.getNom();
	}
	public Crepe garnirCrepe(Crepe c, Garniture g){
		return new CrepeGarnie(c,g);
	}
	public Garniture getGarniture() {
		return garniture;
	}
}
```

Pas besoin du « implements Crepe » dans CrepeGarnie (sauf si CrepeDecoree pas fournie)

– 50% pour l'oubli de l'attribut garniture

– J'ai admis une solution à base d'arrayList de garnitures...

– Je n'ai pas sanctionné une solution (codée proprement) ou la Garniture est dans la CrepeDecoree

Une crêpe double est une crêpe garnie dont tous les ingrédients sont en quantité double et dont le prix est égal au prix de la crêpe simple équivalente, multiplié par 1,5. Le nom commence par « Double crêpe » puis vient la liste des garnitures.

Q2.2 : A l'aide d'une classe CrepeDouble, proposez une implantation pour les crêpes doubles.

```
package pobj;

public class CrepeDouble extends CrepeDecoree {

	public CrepeDouble(Crepe crepe) {
		super(crepe);
	}
```

```
	public double getPrix(){
		return super.getPrix()*1.5;
	}

	public String getNom() {
		return "Double" + super.getNom();
	}
}
```

On peut faire d'autres constructeurs.

Q2.3 : Dans une classe *CrepeMain*, écrivez une méthode standard *main()* qui crée une garniture « anchois » à 1 euro, une garniture « chocolat » à 0.8 euro, puis qui crée une crêpe contenant ces deux garnitures et une crêpe double « chocolat » et affiche leur nom et leur prix.

```
package pobj;

public class CrepeMain {
	public static void main(String[] args){

		Garniture g1 = new Garniture("anchois", 1.0);
		Garniture g2 = new Garniture("chocolat", 0.8);

		CrepeGarnie c2 = new CrepeGarnie(new CrepeGarnie(new
CrepeBasique(),g1),g2);
		CrepeDouble c3 = new CrepeDouble(new CrepeGarnie(new
CrepeBasique(),g2));
		System.out.println(c2.getNom()+":"+c2.getPrix());
		System.out.println(c3.getNom()+":"+c3.getPrix());
	}
}
```

Toujours pas de pitié pour les petites erreurs (-25%, -50% selon le cas...).

3 : Commandes

Une commande est constituée d'une liste de crêpes auxquelles sont associées des quantités. D'un point de vue pratique, la saisie d'une commande se passe de la façon suivante. Le client désigne sa liste de garnitures, il clique sur « crêpe terminée » et indique la quantité. Il peut alors cliquer sur « commande terminée » ou bien commencer la commande d'une autre crêpe et ainsi de suite jusqu'à ce qu'il clique sur « commande terminée ».

La commande est envoyée au serveur de la crêperie sous la forme d'une chaîne de caractères.

Le X sépare les différents types de crêpes. Pour chaque type de crêpe, le C sépare la quantité demandée de la « définition » de la crêpe. Cette définition contient les identifiants des différents ingrédients séparés par la lettre G. On ignore le cas des crêpes doubles. Par exemple, la chaîne « 3C2G5X1C0 » désigne 3 crêpes anchoix chocolat et une crêpe au sucre (0 est le numéro de la garniture « sucre », 2 est le numéro de la garniture « anchois » et 5 celle de la garniture « chocolat »).

Q3.1 : Donnez l'implémentation de la classe *Garnitures* qui permet de stocker l'ensemble des garnitures possibles que l'utilisateur peut commander. La classe devra porter une opération qui ajoute une garniture et rend son identifiant: *int ajouterGarniture(Garniture g)* ainsi que d'une opération *Garniture getGarniture(int index)* qui rend la garniture portant cet identifiant. On s'appuiera de préférence sur une java.util.List dans cette implantation.

```
package pobj;

import java.util.ArrayList;
import java.util.List;

public class Garnitures {
	private List<Garniture> garnitures = new ArrayList<Garniture>();

	public Garniture getGarniture(int index) {
		return garnitures.get(index);
	}

	public int ajouterGarniture(Garniture g){
		garnitures.add(g);
		return garnitures.size();
	}
}
```

-50% pour new List<...>

Q3.2 : Donnez l'implémentation de la classe *Commande* qui permet de représenter une commande quelconque. Cette classe dispose d'une méthode *getPrix()* qui renvoie le prix total d'une commande, d'une méthode *void ajouterCrepe(Crepe c, int quantite)* qui permet d'ajouter un type de crêpe en précisant la quantité, d'une méthode *int getNbTypes()* qui renvoie le nombre de types de crêpes, d'une méthode *int getNbCrepes()* qui renvoie le nombre total de crêpes contenues et d'une méthode *toString()* qui renvoie sous la forme d'une chaine de caractères les noms des crêpes et leur quantité. Pour stocker les crêpes et leur quantité, vous pourrez utiliser un *Map* ou toute autre structure de données qui vous semble appropriée. Vous fournirez aussi un itérateur qui permet de parcourir les types de crêpes de la commande.

```
package pobj;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class Commande implements Iterable<Crepe>{
	private HashMap<Crepe,Integer> crepes;
	private Reduction reduc;

	public Commande(Reduction r){
		reduc = r;
```

```
		crepes =  new HashMap<Crepe,Integer>();
	}

	public String toString(){
		String retour = "";
		for (Map.Entry<Crepe,Integer>  c : crepes.entrySet()){
			retour += c.getKey().getNom() + ":" + c.getValue() + "\n";
		}
		return retour;
	}
	public void ajouterCrepe(Crepe c, int  quantite){
		crepes.put(c, quantite);
	}
	public double getPrix(){
		double prix = 0;
		for (Map.Entry<Crepe,Integer>  c : crepes.entrySet()){
			prix += c.getValue()*c.getKey().getPrix();
		}
//plus tard
		if (reduc.verifCondition(this))return prix*reduc.getTaux();
		else return prix;
	}
	public int getNbCrepes(){
		int nb = 0;
		for (Map.Entry<Crepe,Integer>  c : crepes.entrySet()){
			nb += c.getValue();
		}
		return nb;
	}
	public int getNbTypes(){
		return crepes.size();
	}

	@Override
	public Iterator<Crepe> iterator() {
		return crepes.keySet().iterator();
	}
}
```

Notez séparément les 4 méthodes et la définition de la classe, si un élément est sans faute il rapporte 20.

Q3.3 : Donnez l'implémentation de la classe *CommandeBuilder* dont la méthode *Commande buildCommande(String chaine, Garnitures g)* permet de construire une commande à partir de la chaîne envoyée par le terminal du client et de la liste des garnitures. Pour découper la chaîne de caractère, vous utiliserez la méthode *split()* de la classe *String* dont la description est donnée en annexe. On pourra aussi s'appuyer sur l'opération de la classe *Integer : public static int parseInt (String)* qui extrait un entier d'une chaîne de caractères représentant un nombre entier.

```
package pobj;

public class CommandeBuilder {

	public Commande buildCommande(String comString, Garnitures garnitures,
Reduction reduc){
		Commande commande = new Commande(reduc);
		String[] chaineCrepe = comString.split("X");
		for (int cptCrepe=0; cptCrepe<chaineCrepe.length; cptCrepe++){
			String[] quantiteGarn = chaineCrepe[cptCrepe].split("C");
				Integer nb = Integer.parseInt(quantiteGarn[0]);
				String[] chaineGarnitures = quantiteGarn[1].split("G");
				Crepe crepe = new CrepeBasique();
				for (int z=0; z<chaineGarnitures.length; z++){
					Integer garn =
Integer.parseInt(chaineGarnitures[z]);
					crepe = new
CrepeGarnie(crepe,garnitures.getGarniture(garn));
				}
				commande.ajouterCrepe(crepe,nb);
			}
		return commande;
	}
}
```

Q3.4 : Dans une classe *CommandeMain*, écrivez une méthode standard *main()* qui crée une garniture « anchois » à 1 euro et une garniture « chocolat » à 0.8 euro puis qui crée une commande de crêpes utilisant ces garnitures (vous écrirez « à la main » la chaîne de caractères représentant la commande de votre choix contenant au moins deux types de crêpes simples) et affiche la commande et son prix.

```
package pobj;

public class CommandeMain {
	public static void main(String[] args){
		CommandeBuilder c= new CommandeBuilder();
		Garnitures g = new Garnitures();
		Garniture g1 = new Garniture("anchois", 1.0);
		Garniture g2 = new Garniture("chocolat", 0.8);
		Garniture g3 = new Garniture("sucre", 0.5);
		g.ajouterGarniture(g1);
		g.ajouterGarniture(g2);
		g.ajouterGarniture(g3);
		Commande com = c.buildCommande("3C2G1X12C0X1C0",g,new
ReductionAnchois());
		System.out.println(com);
		System.out.println(com.getPrix());
```

```
	}
}
```

4 : Réductions à gogo

Afin de dynamiser la demande, votre logiciel doit permettre d'appliquer diverses réductions si certaines conditions sont vérifiées par les commandes. Ces réductions sont renouvelées régulièrement, il faut qu'il soit facile d'appliquer de nouvelles réductions qui ne sont pas encore définies. A titre d'exemple, vous allez proposer deux façons de fixer le prix de la commande : la réduction de 10% « grand gourmand » qui s'applique si la commande contient au moins 10 crêpes, et la réduction de 15% « menu varié » qui s'applique si la commande contient au moins 5 crêpes avec des garnitures différentes. Si la commande ne vérifie pas la condition exigée par une réduction, alors le prix est calculé normalement. Par ailleurs, les réductions ne sont pas cumulables.

Vous utiliserez le *Design Pattern Strategy* en vous appuyant sur une interface *Reduction* définie comme suit :

```
package pobj;

public interface Reduction {
	public double getTaux();
	public boolean verifCondition(Commande c) ;
}
```

La réduction applicable sur une commande sera passée en paramètre supplémentaire de la méthode *buildCommande(...) définie à la question Q3.3.*

Q4.1 : Proposez une implantation pour la réduction « grand gourmand »

```
package pobj;

public class ReductionGrandGourmant implements Reduction {

	@Override
	public double getTaux() {
		return 0.9;
	}
	@Override
	public boolean verifCondition(Commande c) {
		return (c.getNbCrepes()>=10);
	}
}
```

Q4.2 : Proposez une implantation pour la réduction « menu varié ».

```
package pobj;

public class ReductionMenuVarie implements Reduction {

	@Override
	public double getTaux() {
		return 0.85;
	}
	@Override
	public boolean verifCondition(Commande c) {
		return (c.getNbTypes()>=5);
```

```
	}
}
```

Q4.3 : Indiquez les modifications à apporter à la classe *Commande* pour supporter l'application des réductions.

Ajout de l'attribut Reduction, du constructeur et modif de la méthode getPrix();

5 Itération sur les garnitures

On souhaite à présent représenter une réduction « 2 anchoix sinon rien » qui s'applique si une commande contient au moins deux fois la garniture « anchoix » et qui réduit le prix de 30%. Pour cela, on a besoin d'itérer sur les garnitures des crêpes contenues dans les commandes.

On propose à cette fin d'ajouter à la déclaration de l'interface *Crepe* : *implements Iterable<Garniture>.* Vous allez créer une classe spécifique *GarnIterator* pour itérer sur les garnitures. On l'initialise en lui passant la crêpe garnie sur laquelle on itère. Les crêpes de base doivent se comporter comme une liste vide, ne contenant aucune garniture.

Q5.1 : Implantez la classe *GarnIterator*.

```
package pobj;

import java.util.Iterator;

public class GarnIterator implements Iterator<Garniture>{
	private Crepe maCrepe;

	GarnIterator(Crepe c){
		maCrepe = c;
	}

	@Override
	public boolean hasNext() {
		return  !(maCrepe instanceof CrepeBasique);
	}

	@Override
	public Garniture next() {
		if (!(maCrepe instanceof CrepeBasique)){
		Garniture g = ((CrepeGarnie)maCrepe).getGarniture();
		maCrepe = ((CrepeGarnie)maCrepe).getCrepe();
		return g;
		}
		else return null;
	}
	@Override
	public void remove() {
		throw new UnsupportedOperationException();
	}
}
```

Q5.2 : Ajoutez à la classe *CrepeDecoree* le code nécessaire pour pouvoir itérer sur les garnitures.

cf. Q2.1

```
ajout du implements Iterable<Garniture>
```

```
	private GarnIterator myIterator = new GarnIterator(this) ;

// ajout plus tard
	@Override
	public Iterator<Garniture> iterator() {
		return myIterator;
	}
```

Q5.3 : Proposez une implantation de la classe *ReductionAnchois* qui réalise le calcul du prix avec la réduction « 2 anchoix sinon rien ».

```
package pobj;

import java.util.Iterator;

public class ReductionAnchois implements Reduction {

	@Override
	public double getTaux() {
		return 0.7;
	}

	@Override
	public boolean verifCondition(Commande c) {
		int nbAnchois = 0;
		Iterator<Crepe> it = c.iterator();
		while (it.hasNext()){
			CrepeGarnie crepe = (CrepeGarnie)it.next();
			Iterator<Garniture> git = crepe.iterator();
			while (git.hasNext()){
				if (git.next().getNom().equals("anchois")) nbAnchois++;
			}
		}
		return (nbAnchois>=2);
	}
}
```

Annexe

Présentation de la méthode *split()*

```
public String[] split(String regex)
```

Splits this string around matches of the given regular expression.

This method works as if by invoking the two-argument `split` method with the given

expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.

The string `"boo:and:foo"`, for example, yields the following results with these expressions:

Regex	*Result*
:	`{ "boo", "and", "foo" }`
o	`{ "b", "", ":and:f" }`

Parameters:

`regex` - the delimiting regular expression

Returns:

the array of strings computed by splitting this string around matches of the given regular expression

Throws:

`PatternSyntaxException` - if the regular expression's syntax is invalid

Since:

1.4

See Also:

`Pattern`