QROC A43 – Juillet 2009

Les documents distribués et les notes personnelles sont seuls autorisés.

NOM : Correction proposée

Prénom : **Filière** :

Compteur

On considère l'interface *ICompteur* qui correspond à la spécification suivante :

```
public interface ICompteur {
	public abstract void up();
	public abstract int getValue();
	public abstract void raz();
}
```

et on considère la classe *Compteur* implémentant cette interface.

```
public class Compteur implements ICompteur {
	protected int value;
	public Compteur() { raz(); }
	@Override
	public String toString() {	return ""+value; }
	public void up() { value++; }
	public int getValue() { return value; }
	public void raz() { value=0; }
}
```

Pour chaque section (indépendante) ci-dessous indiquer si elle est correcte ou non :

	Correcte	Incorrecte
ICompteur ic;	X	
Compteur c;	X	
ICompteur ic=new ICompteur();		X
ICompteur ic=new Compteur();	X	
Compteur c=new ICompteur();		X
Compteur c=new Compteur();	X	
Compteur c=new Compteur(); ICompteur ic=c;	X	
ICompteur ic=new Compteur(); Compteur c=ic;		X
ICompteur ic; Compteur c; c=ic;		X
ICompteur ic; Compteur c; ic=c;	X	

On teste cette classe *Compteur* avec le programme suivant :

```
public class Launcher {
	public static void main(String[] args) {
		Compteur c=new Compteur();
		System.out.println(c);
		System.out.print(c.getValue());
		for(int i=0;i<5;i++) {
			c.up();
			System.out.print(" - "+c.getValue());
		}
}
```

- L'instruction *System.out.println(c)* est correcte malgré le fait que println n'est pas définie pour le type spécifique *Compteur* auquel appartient *c*. Pouvez-vous expliquer cela ?

println attend un chaîne de caractères (*String*). Par commodité le compilateur invoquera *toString* sur l'objet passé en argument à println lorsque cet objet n'est pas une chaîne de caractère. La méthode *toString* est définie dans *Object* donc invocable sur n'importe quelle instance.

- Quel sera à l'écran le résultat d'exécution de ce programme de test (on rappelle que *print* et *println* diffère par le retour chariot généré en fin de traitement par *println*)

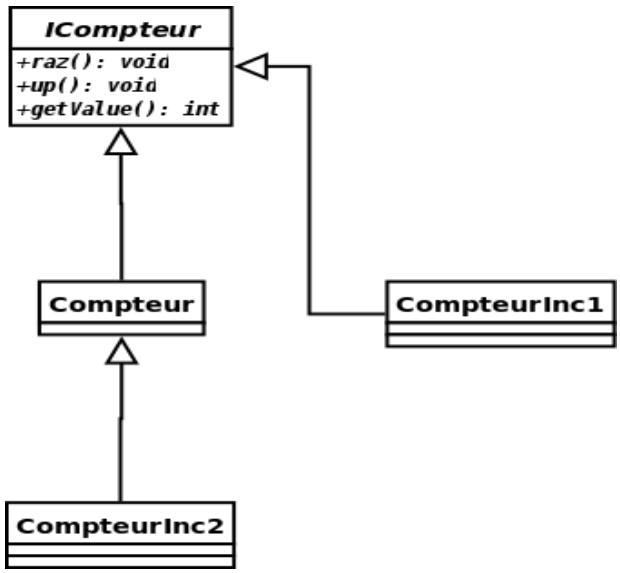
```
0
0 1 2 3 4 5
```

On souhaite maintenant implémenter un compteur à incrément. Un compteur à incrément est un compteur dont on fixe à la naissance la valeur de l'incrément. Chaque invocation de *up* incrémentera la valeur du compteur de la valeur de l'incrément. Le portion de code suivante illustre le comportement attendu de ce type de compteur nommé *CompteurInc* ci-dessous:

test	affichage

`CompteurInc ci=new CompteurInc(7);`
`System.out.println(ci);`
`for(int i=0;i<5;i++) ci.up();`
`System.out.println(ci);`
`ci.raz();`
`System.out.println(ci);`
`ci.up();`
`System.out.println(ci);`
	`0`
	`35`
	`0`
	`7`

Pour l'implémentation on hésite entre deux schémas d'héritage illustrés ci-dessous à travers les classes *CompteurInc1* et *CompteurInc2*.



Implémenter l'une et l'autre version de *CompteurInc*. Dans chaque cas faire figurer la clause d'héritage, le constructeur et faire appel à la super-invocation chaque fois que possible dans la définition des méthodes, ajouter la/les variables(s) membre(s) qui vous semble(nt) nécessaire(s).

- Version *CompteurInc1*

```
public class CompteurInc1 implements ICompteur {
	private int inc;
	private int value;
	public int getInc() { return inc; }
	public CompteurInc1(int inc) { this.inc=inc; }
	public void up() { value+=inc; }
	public int getValue() { return value; }
	public void raz() { value=0; }
	@Override
	public String toString() {
		return ""+value;
	}
}                                                    {
```

- Version *CompteurInc2*

```
public class CompteurInc2 extends Compteur {
	private int inc;
	public int getInc() { return inc; }
	public CompteurInc2(int inc) { this.inc=inc; }
	@Override
	public void up() {
		value+=inc;
	}
}
```

-

Relations entre classes

On considère la portion de code de test suivante où le nom d'origine des classes et des variables a été

remplacé par des lettres de façon à en cacher la signification :

```
public static void main(String[] args) {
	X x=X.createInstance();
	Y y=Y.createInstance();
	Z z=Z.createInstance();

	T t=new T(x);
	U u=new U(x, y);
	V v=new V(z,y,u);
	final W w=new W();
	w.add(t);
	w.add(u);
	w.add(v);
	Thread threadScrutation=new Thread(new Runnable() {
		public void run() {
			while (true) {
				sleep(5000);
				System.out.println(w);
			}
		}
	});
	threadScrutation.start();
}
```

La classe W évoquée ci-dessus est explicitée ci-dessous :

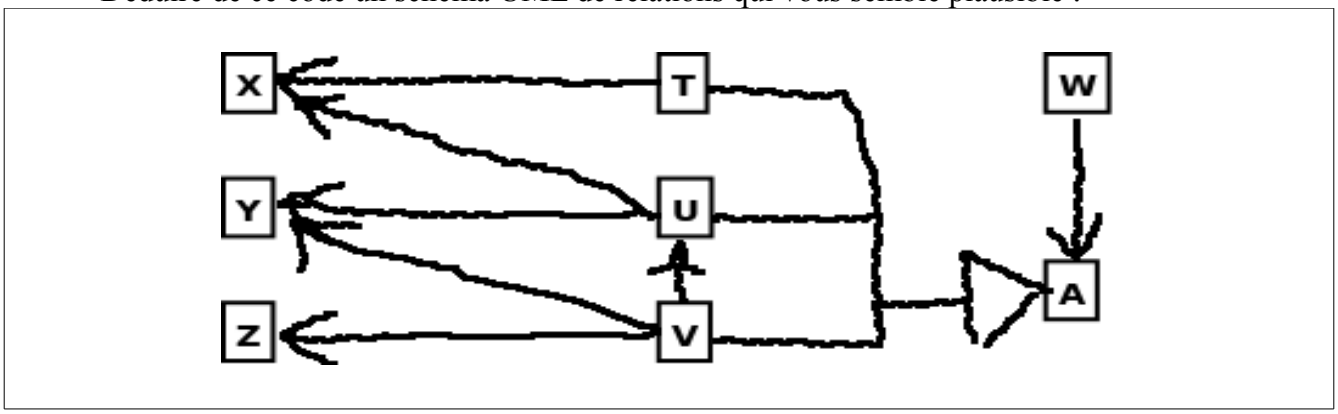
```
public class W {
	LinkedList<A> list;

	public W() { list = new LinkedList<A>(); }

	public void add(A a) { list.add(a); }

	@Override
	public String toString() {
		String s = "";
		for (A a : list) s += a + "\t";
		return s;
	}
}
```

- Déduire de ce code un schéma UML de relations qui vous semble plausible :



On s'intéresse à la classe *Z* évoquée dans le programme de test ci-dessus. Le code de cette classe a l'aspect suivant (on a mis des points (…) pour les portions de code qu'il n'est pas utile d'expliciter ici)

```
public class Z {
	private static Z z=null;
	....
	public static Z createInstance() {
		if (z==null) return z=new Z();
		else return z;
	}
	private Z() {
		....
	}
}
```

Dans le code de test la création d'une instance de Z s'effcetue par *Z z=Z.createInstance()*.

- Aurait-il été possible, dans le code de test, de créer une instance simplement par *Z z=new Z()* comme on le fait usuellement ? Pourquoi ?

En général non car le constructeur Z() est privé.

- Qu'est-ce qui autorise, dans le code de test, à invoquer une méthode (*createInstance*) en la préfixant par un nom de classe (*Z*) et non, comme c'est le cas normalement, par un nom d'objet ?

Parce que c'est une méthode statique

- Quel est l'intérêt finalement de la construction particulière choisie pour la classe *Z* ? Pour vous aider : que se passt-il si on execute la portion de code suivante :

```
Z [] tab=new Z[1000];
for(int i=0;i<1000;i++) tab[i]=Z.createInstance();
```

createInstance retourne une instance de Z qu'elle créé si elle n'existe pas déjà, sinon elle retourne l'instance déjà existante. Autrement dit createInstance permet d'être sûr qu'une seule instance de Z est créé. C'est ce qu'on appelle un singleton. Dans l'itération proposée une seule instance de Z est réellement créée (toutes les cases du tableau désignent donc le même objet)

Café

On souhaite modéliser une application OO embarquée dans un distributeur de café. Celui-ci permet au client d'obtenir une boisson chaude choisie (café- 0.5€, expresso-0.5€, chocolat-0.5€, thé-0.3€) et de lui ajouter optionnellement un certain nombre d'ingrédients (chantilly-0.2€, lait-0.1€, citron-0.1€, sucre-0,05€ cacao-0.1€). Le prix de la boisson dépend de la boisson choisie et du type et du nombre d'ingrédients supplémentaires choisies. Exemples :

Commande 1 : Un café simple peut par exemple se commander ainsi : un café, une dose de sucre, soit 0.5+0.05 = 0.55€

Commande 2 : Un capuccino très sucré peut se commander ainsi : un café, une dose de chantilly, deux dose de sucre, une dose de cacao, soit 0.5+0.2+0.05+0.05+0.1=0.90€

Commande 3 : Un thé au citron sucré peut se commander ainsi : un thé, une dose de citron, une dose de sucre, soit 0.3+0.1+0.05=0.45€

On souhaite prendre en compte les inévitables évolutions qui vont concerner les tarifs des boissons ou des ingrédients, ou encore l'introduction ou le retrait de nouvelles boissons ou ingrédients. Pour cela on décide d'utiliser un *design-pattern*, le *decorateur*, qui semble adapté dans le cas présent à cet objectif (et qu'il n'est pas nécessaire de connaître pour faire cet exercice).

L'analyse du problème nous montre que le produit finalement choisi par le client est composé d'une et une seule **boisson** et d'un certain nombre d'**ingrédients** supplémentaires. Pour respecter les principes objets on souhaite que chaque boisson soit prise en charge par une classe très simple et de même pour chaque ingrédient, mais pour ces derniers on souhaite capturer le fait qu'un ingrédient est toujours relatif à une boisson choisie (le client ne peut commander une chantilly seule par exemple)

Une boisson comporte une courte description et le tarif. La description est affichée sur l'écran à cristaux liquides du distributeur au fur et à mesure que le client complète sa commande. La classe *Cafe,* par exemple, est la suivante:

```
public class Cafe implements Commande {
	public double getCout() {
		return 0.5;
	}
	public String getDescription() {
		return "preparation en grain-robusta";
	}
}
```

Dans ce schéma la création des instances correspondant aux trois commandes ci-dessus serait la suivante :

```
	public static void main(String[] args) {
		// commande 1
		Commande commande1=new Cafe();
		// on ajoute un ingredient, sucre, à la commande
		commande1=new Sucre(commande1);
		// affichage du prix
		System.out.println(commande1.getCout()+" €");
		System.out.println(commande1.getDescription());

		// commande 2
		Commande commande2=new Cafe();
		// on ajoute un ingredient, sucre, à la commande
		commande2=new Sucre(commande2);
		// on ajoute un ingredient, sucre, à la commande
		commande2=new Sucre(commande2);
		// on ajoute un ingredient, chantilly, à la commande
		commande2=new Chantilly(commande2);
		// on ajoute un ingredient, cacao, à la commande
		commande2=new Cacao(commande2);
		// affichage du prix
		System.out.println(commande2.getCout()+" €");
		System.out.println(commande2.getDescription());
```

```
            // commande 3
            Commande commande3=new Sucre(new Citron(new The()));
            System.out.println(commande3.getCout()+" €");
            System.out.println(commande3.getDescription());
      }
```

L'exécution de cette portion de code de test donne le résultat suivant :

```
0.55 €
dose de 5g de sucre sur preparation en grain-robusta
0.9 €
dose de 3g de cacao sur emulsion de creme de normandie sur dose de 5g de sucre sur
                        dose de 5g de sucre sur preparation en grain-robusta
0.45 €
dose de 5g de sucre sur 1 cl de citron presse sur the origine ceylan
```

L'interface Commande est la suivante :

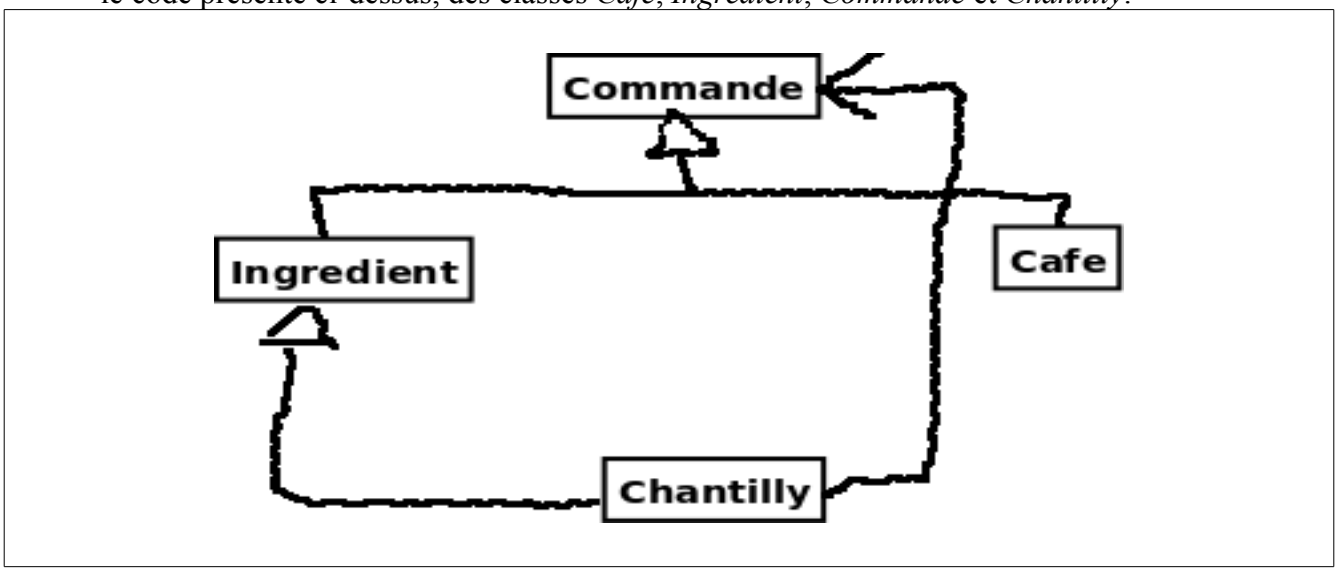
```
public interface Commande {
      abstract double getCout();
      abstract String getDescription();
}
```

La classe abstraite *Ingredient* est la suivante :

```
public abstract class Ingredient implements Commande{
      protected Commande commande;
      public Ingredient(Commande commande) {
            this.commande=commande;
      }
}
```

La classe concrète Chantilly hérite de *Ingredient*

- Proposer un schéma UML, en y faisant figurer les liens d'héritage et de dépendance induits par le code présenté ci-dessus, des classes *Cafe*, *Ingredient*, *Commande* et *Chantilly*.



- La classe abstraite *Ingredient* telle quelle pourrait-elle concrète ? Pourquoi ?

Non, par ce qu'elle hérite des deux méthodes abstraites *getCout* et *getDescription* sans être capable de les définir.

- La classe abstraite *Ingredient* telle quelle pourrait-elle une *interface* ? Pourquoi ?

Non parce qu'elle est en mesure de définir son constructeur et qu'elle contient une variable membre (Une interface ne définit rien et ne contient pas de varaible membre)

- Ecrire complétement la classe *Chantilly* en tenant compte de ce qui précède (lien d'héritage, constructeur, implémetnation des méthodes). Une invocation de *getCout* retourne le coût total de la commande dont *this* est un ingrédient. Même remarque pour *getDescription*.

```
public class Chantilly extends Ingredient  {
      public Chantilly(Commande commande) {
            super(commande);
      }
      public double getCout() {
            return commande.getCout()+ 0.2;
      }
      public String getDescription() {
            return "emulsion de creme de normandie sur " +
                                          commande.getDescription();
      }
}
```