HAI401I - Exercices sur :
Spécialisation, Polymorphisme, Sous-typage

1 Comptes bancaires : Résoudre des conflits d'héritage

Question 1 *Comptes bancaires*
Ecrivez en Java les types suivants :

- *Une interface* `CompteBancaire` *munie de 3 méthodes pour (1) connaître, (2) modifier le solde, (3) fermer le compte en affichant le solde qui sera rendu.*
- *Une interface* `CompteRemunere` *qui spécialise* `CompteBancaire` *et dans laquelle la fermeture du compte donne lieu à un ajout de 10% du solde puis à l'affichage du solde (spécialiser la méthode de* `CompteBancaire`*).*
- *Une interface* `CompteDepot` *qui spécialise* `CompteBancaire` *et dans laquelle la fermeture du compte donne simplement lieu à une réduction de 100, mais rien n'est affiché.*
- *Une interface* `CompteDepotRemunere` *qui spécialise* `CompteRemunere` *et* `CompteDepot`*. Que se passe-t-il si vous ne prévoyez pas de redéfinir la méthode de fermeture ? Ecrivez ensuite la méthode : la fermeture consiste à appeler la fermeture telle que décrite pour un compte de dépôt, puis telle que décrite pour un compte rémunéré.*
- *Une classe* `CDR` *qui implémente* `CompteDepotRemunere`*. Créez une instance de cette classe et appelez la méthode de fermeture.*

2 Couleurs : Reconnaître les schémas de redéfinition des méthodes

Question 2 *Schémas de définition des méthodes*
Le code suivant illustre différents schémas de définition/redéfinition des méthodes dans le cadre de la programmation par objets :

1. *héritage sans redéfinition*
2. *redéfinition avec masquage (une méthode définie dans une classe est complètement redéfinie dans une de ses sous-classes)*
3. *redéfinition avec spécialisation (on utilise l'appel* `super`*)*
4. *méthode abstraite*
5. *définition par généralisation (une méthode appelle une autre méthode dont les versions spécifiques pourront être trouvées dans les sous-classes)*
   (a) *avec appel d'une méthode abstraite*
   (b) *avec appel d'une méthode concrète*

Questions.

- *Rangez les différentes méthodes présentées ci-dessous dans ces catégories. Si vous observez des définitions problématiques, indiquez-les.*
- *Analyser le programme* `main` *(compilation et exécution).*

```
abstract class Pinceau{
    public void vert(){System.out.print("vert");}
    public void rose(){System.out.print("rose");}
    public void bleu(){System.out.print("bleu");}
    public void jaune(){System.out.print("jaune"); gamme(); }
    public void orange(){System.out.print("orange"); variante_orange();  }
    public void variante_orange(){System.out.print(" standard");}
    abstract void gamme();
}
abstract class PinceauClair extends Pinceau{
    @Override
    public void bleu(){System.out.print("bleu clair");}
}
class PinceauPastel extends PinceauClair{
    @Override
    public void bleu(){System.out.print("bleu pastel");}
    @Override
    public void rose(){super.rose(); System.out.print(" dragée");}
    @Override
    public void variante_orange(){System.out.print(" coquille d'oeuf");}
    public void mauve1(){this.rose(); System.out.print("---"); this.bleu();}
    public void mauve2(){super.rose(); System.out.print("---"); super.bleu();}
    @Override
    public void gamme() {System.out.print(" pastel");}
}
public class Couleurs {
    public static void main(String[] args) {
        System.out.println("------------- pinceau pp3 -----------------");
        PinceauPastel pp3 = new PinceauPastel();
        pp3.vert();         System.out.println();
        pp3.rose();        System.out.println();
        pp3.bleu();        System.out.println();
        pp3.jaune();      System.out.println();
        pp3.orange();      System.out.println();
        pp3.mauve1();  System.out.println();
        pp3.mauve2();  System.out.println();
        System.out.println("------------- pinceau p3 -----------------");
        Pinceau p3 = new PinceauPastel();
        p3.vert();  System.out.println();
        p3.rose();    System.out.println();
        p3.bleu();    System.out.println();
        p3.jaune();  System.out.println();
        p3.orange();      System.out.println();
        p3.mauve1();
        p3.mauve2();
        System.out.println("------------- pinceau pc3 -----------------");
        PinceauClair pc3 = new PinceauPastel();
        pc3.vert();    System.out.println();
        pc3.rose();   System.out.println();
        pc3.bleu();   System.out.println();
        pc3.jaune(); System.out.println();
        pc3.orange();      System.out.println();
        pc3.mauve1();
        pc3.mauve2();
    }
}
```

3 Bières : Réviser le principe de la liaison dynamique

QUESTION 3 *Liaison dynamique*

1. *Etudiez le programme suivant, mettez-le en place et exécutez-le. Indiquez quelles lignes du* **`main`** *compilent et ce qu'affiche le programme. Expliquez également quel est le type statique et le type dynamique qui sont en jeu et comment la liaison dynamique s'effectue dans chacun des cas.*
2. *Que pensez-vous de la méthode* **`description`** *de la classe* `MortSubiteKriek` *?*

```
public class Biere{
  public void description() {System.out.print("Biere ");this.couleur();}
  public void couleur() {System.out.println("variee");}
}
public class BiereAcidulee extends Biere{}
public class Lambic extends BiereAcidulee{
  public void description() {super.description(); System.out.println("Lambic ");}
  public void couleur() {System.out.println("blond cuivre");}
}
public class MortSubiteKriek extends Lambic{
  public void description()  {System.out.print("Biere Kriek ");
                              super.couleur(); System.out.println("Lambic ");}
  public void couleur() {System.out.println("rouge vif");}
}
public static void main(String[] args) {
 Biere biere1 = new Biere();   biere1.description();                    // main1

 Biere biere2 = new Lambic();  biere2.description();                    // main2

 Biere biereA1 = new MortSubiteKriek();  biereA1.description();         // main3

 BiereAcidulee biereA2 = new MortSubiteKriek();  biereA2.description(); // main4

 MortSubiteKriek biereA3 = new MortSubiteKriek();  biereA3.description();// main5
}
```

4 Bières : Savoir identifier une mauvaise conception et la faire évoluer

QUESTION 4 *Responsabilité des classes*

Le calcul de l'indice de coloration d'une bière (donné en unité EBC pour European Brewery Convention) se calcule à partir de l'indice de coloration procuré par les malts utilisés (MCU pour Malt Color Unit). Le MCU d'un malt se calcule d'après la couleur des grains du malt (EBC du malt) et leur dilution (poids de malt par volume de bière réalisé) :

$$MCU\ du\ malt = 4,24 * EBC\ du\ malt * Poids\ utilise\ (kg) / Volume\ de\ biere(L) \quad (1)$$

Pour une bière single malt (qui utilise un seul malt), l'EBC est donné par

$$EBC\ d'une\ biere\ single\ malt = 2,94 * (MCU\ du\ malt)^{0,6859} \quad (2)$$

Pour une bière comprenant plusieurs malts, l'EBC est donné par

$$EBC\ d'une\ biere\ avec\ plusieurs\ malts = 2,94 * (Somme\ des\ MCUs\ des\ malts)^{0,6859} \quad (3)$$

Un programmeur a rédigé le code des deux classes suivantes, qui décrivent les bières et les bières single malt (utilisant un unique malt). Ce code fonctionne mais nous allons analyser pourquoi il est écrit de manière inappropriée du point de vue de la responsabilité des classes.

```
public class Beer{
     public double beerEBC() {return 2.939*Math.pow(MCU(), 0.6859);}
     public double MCU() {
          if (this instanceof SingleMaltBeer){
               SingleMaltBeer smb = (SingleMaltBeer)this;
               return 4.23*smb.getMaltEBC()*
               smb.getMaltWeight()/smb.getBeerVolume();
          } else return 1;
     }
}

public class SingleMaltBeer extends Beer{
     private double maltWeight; //KG
     private double beerVolume; //L
     private double maltEBC; // en EBC
     public double getMaltWeight() {return maltWeight;}
     public double getBeerVolume() {return beerVolume;}
     public double getMaltEBC() {return maltEBC;}
     public SingleMaltBeer(double maltWeight, double beerVolume, double maltEBC) {
          this.maltWeight = maltWeight;
          this.beerVolume = beerVolume;
          this.maltEBC = maltEBC;
     }
}
```

1. *Identifiez ce qui est problématique dans ce programme et proposez quelques corrections évidentes.*
2. *Si vous deviez ajouter une sous-classe* `MultiMaltBeer` *de* `Beer` *pour décrire les bières composées d'un assemblage de plusieurs malts et que vous n'avez pas le droit de modifier le code des deux classes existantes, comment feriez-vous évoluer ce programme (écrivez le code correspondant) et comment jugez-vous le résultat ?*
3. *Revoyez le problème en profondeur, et discutez de différentes solutions en Java et en UML présentant les concepts de malt, de bière, de bière single malt, de bière composée de plusieurs malts, en prévoyant qu'il peut exister d'autres sortes de bières, notamment des bières sans malt.*
4. *Programmez l'une des modélisations en UML.*

5 Fournisseurs : Etudier la notion de visibilité

Le modèle de la figure 1 présente quelques classes pour un logiciel de vente. Des fournisseurs (`Supplier`*) offrent des opérations* `supply` *(fournir) et* `retailerList` *(liste des*

revendeurs). Des personnes (`Person`) et des enfants (`Child`) pourront faire des achats chez les revendeurs (`Retailer`). Les revendeurs, qui disposent d'une méthode de vente (`sell`) sont divisés en plusieurs catégories, revendeurs d'alcool (`RetailerOfAlcohol`) et revendeurs de bonbons (`RetailerOfCandies`) entre autres.

On s'intéresse à la mise en place d'accès à différentes méthodes, en considérant qu'un accès qui indique que `C1` peut accèder à `m` sur `C2` (noté par (`C1`, `m`, `C2`)) signifie : une méthode de `C1` peut contenir une expression `c2.m` avec `c2` une expression de type statique `C2`.

Différents accès sont prévus pour notre modèle :

- *Les revendeurs (quel que soit leur type) sont les seuls à accéder à la méthode `supply` des fournisseurs. Ils peuvent aussi accéder à la méthode `sell` sur n'importe quel type de revendeur.*
- *Les personnes (et les enfants) ne peuvent accéder qu'aux méthodes `sell` et `retailerList`. Elles peuvent accéder en général à `sell` sur n'importe quel type de revendeur. Mais il y a une exception, les enfants ne devraient accéder à la méthode `sell` que chez les revendeurs de bonbons.*

Question 5 *Faites un graphe schématisant les accès (`C1`, `m`, `C2`) autorisés.*

Question 6 *Puis réfléchissez sur la manière dont cela pourrait être mis en œuvre en Java. Vous ne pourrez pas reproduire exactement les accès prévus mais cherchez à les approcher. La question peut se poser de deux manières différentes : trouver la solution qui permet les accès demandés et le moins possible d'accès supplémentaires ; trouver la solution qui permet de n'avoir que des accès demandés et le moins possible d'accès manquants.*

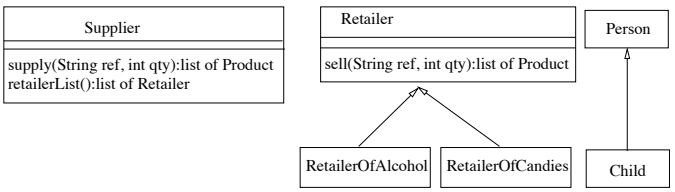

Figure 1 – Fournisseurs