AP3 – Programmation en Java
TD 02

Mickael Montassier et Arnaud Pêcher

8 septembre 2011

Résumé

Présentation des notions de classes abstraites et d'interfaces.

L'ensemble des fichiers de cette séance est disponible dans :
`/net/Bibliotheque/AP3/TD02/code/`

1 Cours

1.1 Classe abstraite

Quelques rappels :
- Une classe abstraite ne peut pas être instanciée.
- Toute classe contenant une méthode abstraite est abstraite et doit être déclarée comme tel.
- Une classe peut être définie comme abstraite sans contenir de méthodes abstraites (empêche l'instanciation). (une classe abstraite peut être implémentée en partie ou en totalité.)
- Une sous-classe peut être instanciée si et seulement si elle implémente toutes les méthodes abstraites de sa classe mère (abstraite).
- Une sous-classe qui n'instancie qu'en partie les méthodes abstraites de sa classe mère (abstraite) est abstraite.

1.2 Exemple : Shape

Nous allons considérer ici un ensemble de formes (rectangle ou cercle). Toute forme possède une aire, un périmètre. Nous voulons pouvoir calculer l'aire totale ou le périmètre total d'un ensemble de formes.

Voici comment déclarer la classe **Shape** abstraite. Le mot-clef à retenir est `abstract`.

```
public abstract class Shape{
    public abstract double area();
    public abstract double circumference();
}
```

Étant donnée une forme, il n'est possible de calculer l'aire ou le périmètre de la forme que lorsque l'on connait la forme. Les méthodes `area` et `circumference` ne sont donc pas implémentées et sont déclarées comme abstraites `abstract`.

Écrivons maintenant les classes **Rectangle** et **Cercle** étendant le classe **Shape** et implémentant les méthodes `area` et `circumference` :

```
public class Rectangle extends Shape{
    protected double height;
    protected double width;
    public Rectangle(double h, double w){
        height=h;
        width=w;
    }
    public Rectangle(){
        this(1.0,2.0);
    }
    public double getWidth(){return width;}
    public double getHeight(){return height;}
    public void setWidth(double w){
        width=w;
    }
    public void setHeight(double h){
        height=h;
    }
    public double area(){
        return height*width;
    }
    public double circumference(){
        return 2*(height+width);
    }
}

public class Circle extends Shape{
    protected double radius;
    protected static final double PI = 3.14159265358979323846;
    public Circle(){
        radius=1.0;
    }
    public Circle(double r){
        radius=r;
    }
    public double getRadius(){
        return radius;
    }
    public void setRadius(double r){
        radius=r;
    }
    public double area(){
        return PI*radius*radius;
    }
    public double circumference(){
        return 2*PI*radius;
    }
}
```

Enfin proposons un programme de tests :

```
public class ShapeApp{
    public static void main(String args[]){
        Shape [] s = new Shape [3];
        s[0]=new Circle(3.0);
        s[1]=new Rectangle(2.0,3.0);
        s[2]=new Rectangle();
```

```
        ((Rectangle)s[2]).setWidth(10);

        double area=0;
        double circumference=0;
        for(int i=0;i<s.length;i++){
            area+=s[i].area();
            circumference+=s[i].circumference();
        }
        System.out.println("Area          : " + area + "\n" +
                           "Circumference : " + circumference);
    }
}
```

Ce qui produit :

```
> java ShapeApp
Area          : 44.27433388230814
Circumference : 50.84955592153876
```

1.3 Interface

Supposons que l'on veuille maintenant implémenter des formes dessinables sur un écran.

Une solution possible est d'écrire une classe abstraite **DrawableShape** contenant par exemple les champs couleur, position et une méthode de dessin (abstraite) et d'ajouter deux classes **DrawableRectangle** et **DrawableCircle** implémentant la méthode de dessin (de DrawableShape).

Nous voulons en plus que ces formes dessinables supportent les méthodes `area` et `circumference`. Bien sûr, nous ne voulons pas réécrire du code !

Nous aimerions que **DrawableRectangle** hérite de **Rectangle**... mais c'est impossible : pas d'héritage multiple en java.

Une parenthèse sur l'héritage multiple En java, une classe peut étendre **une seule super-classe** : **modèle d'héritage simple**. L'**héritage multiple** est utile quand une nouvelle classe veut ajouter un nouveau comportement et garder la plupart ou tout l'ancien comportement. Mais quand il y a plus d'une super-classe, des problèmes surviennent quand le comportement d'une super-classe est hérité de deux façons. Par exemple, l'héritage diamant : X et Y héritent de W ; Z hérite de X et Y. Supposons que W contienne un champs publique `coucou` et soit `zref` une référence sur un objet Z. À quoi correspond `zref.coucou` ? coucou de X, coucou de Y ? Les problèmes d'héritage multiple surviennent d'un héritage multiple d'implémentation. Java fournit un moyen d'hériter un contrat sans hériter l'implémentation : **les interfaces**.

Une classe est un mélange de conception et d'implémentation. Une **interface** est une abstraction de pure conception : uniquement des constantes et des méthodes abstraites, **aucune implémentation**. Chacun peut donner alors sa propre implémentation d'une interface.

Revenons aux formes dessinables. Une solution possible est la suivante :

1. Créer une interface **Drawable** contenant les méthodes :
   `public void setColor(int color)`
   `public void setPosition(double x, double y)`
   `public void draw()`
2. Ajouter une classe **DrawableRectangle** sous-classant **Rectangle** et implémentant **Drawable**
3. Idem pour **Circle**

Voila ce que cela donne :

```
public interface Drawable {
    public void setColor(int color);
    public void setPosition(double x,double y);
    public void draw();
}

/*
  Interface :

  Toutes les méthodes sont implicitement abstraites.
  Les variables se doivent d'être static et final.

 */

public class DrawableRectangle extends Rectangle implements Drawable{
    protected int color;
    protected double abs;
    protected double ord;

    public DrawableRectangle(double h,double w,
                             double abs,double ord){
        super(h,w);
        this.abs=abs;
        this.ord=ord;
    }
    public void setColor(int color){this.color=color;}
    public void setPosition(double x,double y){abs=x;ord=y;}
    public void draw(){
        System.out.println("DrawableRectangle : draw");
    }
}
/*
  implements :
  toutes les méthodes de l'interface doivent être implémentées.

  implementer plusieurs interfaces est possible.
 */

public class DrawableCircle extends Circle implements Drawable{
    protected int color;
    protected double abs;
    protected double ord;

    public DrawableCircle(double r,
                          double abs,double ord){
        super(r);
        this.abs=abs;
        this.ord=ord;
    }
    public void setColor(int color){
        this.color=color;
    }
    public void setPosition(double x,double y){
        abs=x;
```

```
        ord=y;
    }
    public void draw(){
        System.out.println("DrawableCircle : draw");
    }
}

public class ShapeApp{
    public static void main(String args[]){
        Shape [] s = new Shape [2];
        Drawable [] d = new Drawable [2];

        DrawableCircle    dc = new DrawableCircle(3,10,11);
        DrawableRectangle dr = new DrawableRectangle(5,6,0,1);

        s[0] = dc;
        s[1] = dr;

        d[0] = dc;
        d[1] = dr;

        double area = 0;

        for(int i=0;i<s.length;i++){
            area+=s[i].area();
            d[i].draw();
        }
        System.out.println("area : " + area );
    }
}
```

À noter : Une classe peut implémenter plusieurs interfaces. Une interface peut étendre **plusieurs** interfaces.

1.4 Exercice : l'aquarium

Un aquarium contient toute sorte de choses : des pierres, des algues, des poissons (des requins, des sardines... notre aquarium est plutôt grand). Nous aimerions observer cet aquarium : les sardines et les requins nagent, les sardines se nourrissent d'algues, les requins chassent,...

1. Créer la classe **AquariumItem** représentant une chose de l'aquarium ; toute chose a une **Position**, une hauteur, une largeur. La classe **AquariumItem** contient une méthode `toString` permettant d'afficher l'ensemble des informations de l'item et une méthode `draw` de dessin de l'item.
2. Ajouter les classes permettant de représenter les algues, les pierres, les sardines, les requins.
3. Écrire une classe de tests permettant de remplir notre aquarium et de l'afficher.

Un peu d'aide :

```
// Position.java
public class Position{
    public int x;
    public int y;

    public Position(){
        this((int)(Math.random()*10),(int)(Math.random()*10));
    }
```

```
    public Position(int x,int y){
        this.x=x;
        this.y=y;
    }
    public String toString(){
        return "(" + x + "," + y + ")";
    }
}

// AquariumItem.java
public abstract class AquariumItem{
    protected Position position;
    protected int width;
    protected int height;

    public AquariumItem(Position position,int width,int height){
        this.position=position;
        this.width=width;
        this.height=height;
    }
    public String toString(){
        return
            "Position : " + position + "\n" +
            "Width    : " + width + "\n" +
            "Height   : " + height ;
    }
    public abstract void draw();
}

// Stone.java
public class Stone extends AquariumItem{
    String name;

    public Stone(Position position,int width, int height, String name){
        super(position,width,height);
        this.name=name;
    }
    public String toString(){
        return "Stone \n" +
            super.toString() + "\n" +
            "name     : " + name ;
    }
    public void draw(){
        System.out.println("Stone    : draw");
    }
}

// Seaweed.java
public class Seaweed extends AquariumItem{
    String type;

    public Seaweed(Position position,int width,int height,String type){
        super(position,width,height);
        this.type=type;
    }
    public String toString(){
        return "Seaweed" + "\n" +
            super.toString() + "\n" +
            "type     : " + type ;
    }
    public void draw(){
        System.out.println("Seaweed  : draw");
    }
}

// Fish.java
public class Fish extends AquariumItem {
    public Fish(Position position,int width,int height){
```

```
        super(position,width,height);
    }
    public String toString(){
        return getClass().getName() + "\n" + super.toString();
    }
    public void draw(){
        System.out.println("Fish : draw");
    }
}

// Shark.java
public class Shark extends Fish {
    public Shark(Position position,int width,int height){
        super(position,width,height);
    }
    public void draw(){
        System.out.println("Shark    : draw");
    }
}

// Sardine.java
public class Sardine extends Fish {
    public Sardine(Position position,int width,int height){
        super(position,width,height);
    }
    public void draw(){
        System.out.println("Sardine  : draw");
    }
}

// Aquarium.java
import java.util.*;

public class Aquarium{
   public static void remplir (Collection<AquariumItem> collection){
       for(int i=0;i<10;i++){
           int choix=(int)(Math.random()*4);
           switch(choix){
           case 0: collection.add(new Stone(new Position(),2,3,"calcaire"));
               break;
           case 1: collection.add(new Seaweed(new Position(),1,1,"flottante"));
               break;
           case 2: collection.add(new Shark(new Position(),5,3));
               break;
           case 3: collection.add(new Sardine(new Position(),2,2));
               break;
           }
       }
   }
   public static void afficher(Collection<AquariumItem> collection){
       for(AquariumItem i : collection){
           System.out.println(i);
           i.draw();
       }
   }
   public static void main(String[] args){
       Collection<AquariumItem> collection = new LinkedList<AquariumItem>();
       remplir(collection);
       afficher(collection);
   }
}
```

Ce qui donne (extraits) :

```
Shark
Position : (4,3)
Width    : 5
```

```
Height   : 3
Shark    : draw

Sardine
Position : (5,2)
Width    : 2
Height   : 2
Sardine  : draw

Stone
Position : (4,2)
Width    : 2
Height   : 3
name     : calcaire
Stone    : draw

Seaweed
Position : (2,7)
Width    : 1
Height   : 1
type     : flottante
Seaweed  : draw
```

Maintenant nos poissons se déplacent : ils choisissent une destination et s'y rendent.

4. Ajouter l'interface **Mobile** contenant les deux méthodes :
   `Position myTarget()`
   `void moveTo()`
5. Modifier votre hiérarchie en ajoutant la classe **MobileItem**.
6. Tester.

```
// Mobile.java
public interface Mobile{
    public Position myTarget();
    public void moveTo();
}

// MobileItem.java
public class MobileItem extends AquariumItem implements Mobile{
    Position target;

    public MobileItem(Position position,int width,int height,Position target){
        super(position,width,height);
        this.target=target;
    }
    public String toString(){
        return super.toString() + "\n" + "target   : " + target ;
    }
    public void draw(){
        System.out.println("MobileItem : draw");
    }
    public Position myTarget(){
        return new Position();
    }
    public void moveTo(){
        if(target.x > position.x) position.x++;
        if(target.x < position.x) position.x--;
        if(target.y > position.y) position.y++;
        if(target.y < position.y) position.y--;
    }
}

// Fish.java
public class Fish extends MobileItem {
```

```
    public Fish(Position position,int width,int height,Position target){
        super(position,width,height,target);
    }
    public String toString(){
        return getClass().getName() + "\n" + super.toString();
    }
    public void draw(){
        System.out.println("Fish : draw");
    }
}

// Sardine.java
public class Sardine extends Fish {
    public Sardine(Position position,int width,int height,Position target){
        super(position,width,height,target);
    }
    public void draw(){
        System.out.println("Sardine  : draw");
    }
}

// Shark.java
public class Shark extends Fish {
    public Shark(Position position,int width,int height,Position target){
        super(position,width,height,target);
    }
    public void draw(){
        System.out.println("Shark    : draw");
    }
}

// Aquarium.java
public class Aquarium{
    public static void remplir (Collection<AquariumItem> collection){
        for(int i=0;i<4;i++){
            int choix=(int)(Math.random()*4);
            switch(choix){
            case 0: collection.add(new Stone(new Position(),2,3,"calcaire"));
                break;
            case 1: collection.add(new Seaweed(new Position(),1,1,"flottante"));
                break;
            case 2: collection.add(new Shark(new Position(),5,3,new Position()));
                break;
            case 3: collection.add(new Sardine(new Position(),2,2,new Position()));
                break;
            }
        }
    }
    public static void afficher(Collection<AquariumItem> collection){
        for(AquariumItem i : collection){
            System.out.println(i);
            i.draw();
            System.out.println();
        }
    }
    public static void bouger(Collection<AquariumItem> collection){
        for(AquariumItem i : collection){
            if(i instanceof MobileItem)
                ((MobileItem)i).moveTo();
        }
    }
    public static void main(String[] args){
        Collection<AquariumItem> collection = new LinkedList<AquariumItem>();
        remplir(collection);
        afficher(collection);
        bouger(collection);
        afficher(collection);
    }
}
```

2 Collections

Dans l'exercice précédent, vous avez bien évidemment remarqué l'utlisation d'une liste chainée (objet `LinkedList`) dans la méthode `main` pour stocker les items de l'aquarium. De plus, compte tenu des signatures des méthodes de la classe `Aquarium`, `LinkedList` est une spécialisation d'un objet `Collection`. Nous allons aborder plus en profondeur dans cette section l'API des `Collections` en Java.

2.1 L'interface Collection

L'interface `Collection` est une des principales interfaces de Java. Elle sert pour le stockage et le parcours d'un ensemble d'objets, pas nécessairement de même type.

2.1.1 Prototype

Cette interface possède deux méthodes essentielles :

```
boolean add(Object obj)
Iterator iterator()
```

La méthode `add` ajoute un objet dans la collection. Elle renvoie `true` si l'ajout de l'objet a effectivement modifié la collection, `false` sinon.

La méthode `iterator` renvoie un objet qui implémente l'interface `Iterator`.

Cette interface possède trois méthodes principales :

```
Object next()
boolean hasNext()
void remove()
```

En appelant plusieurs fois la méthode `next`, vous pouvez parcourir tous les éléments de la collection un par un. Lorsque la fin de la collection est atteinte, la méthode `next` déclenche une exception `NoSuchElementException`[1]. Enfin, la méthode `remove` supprime l'élément renvoyé par le dernier appel à `next`.

```
Iterator i = c.iterator();
while (i.hasNext()){
Object o = i.next();
}
```

L'interface `Collection` déclare aussi ces méthodes, dont les prototypes sont suffisamment explicites :

```
int size()
boolean isEmpty()
boolean containsObject( Object )
boolean containsAll( Collection )
boolean equals( Object )
boolean addAll( Collection )
boolean remove( Object )
boolean removeAll( Collection )
void clear()
boolean retainAll( Collection )
Object[] toArray()
```

1. Les exceptions seront étudiées dans une prochaine séance

La classe `AbstractCollection` définit les méthodes fondamentales (`add`, `size` et `Iterator`) comme abstraites et implémente toutes les autres. Une classe de collection concrète peut donc se limiter à l'implémentation des méthodes fondamentales.

2.1.2 Exercice

Créer votre propre classe `MyCollection` implémentant une `Collection` et utilisez celle-ci pour le stockage des items de l'aquarium, i.e., le code de la méthode `main` sera maintenant :

```
public static void main(String[] args){
        Collection<AquariumItem> collection = new MyCollection();
        remplir(collection);
        afficher(collection);
        bouger(collection);
        afficher(collection);
    }
```

Naturellement, vous aurez également besoin de définir une classe `MyIterator` ...

2.1.3 Les listes doublement chainées

La classe `LinkedList` est l'implémentation Java des listes doublement chainées. Elle implémente l'interface `Collection`.

- *Avantages* : suppression, ajout d'un élément rapide ;
- *Inconvénients* : récupération d'un élément nécessite le parcours des éléments qui le "précèdent".

La méthode `add` fournie ajoute un élément en fin de liste.

Pour ajouter un élément au sein de la liste, il faut utiliser un objet `ListIterator` (une interface héritant de l'interface `Iterator`) qui permet l'insertion d'un élément à la position courante, via sa méthode `add`. La classe `ListIterator` ajoute également deux méthodes `previous` and `hasPrevious` qui permettent le parcours de la liste à rebours.

Exemple (ajout de la chaine "bob", en deuxième position)

```
LinkedList ll = new LinkedList();
...
ListIteratoi li = ll.listIterator();
li.next();
li.add("Bob");
```

La classe `ListIterator` possède aussi une méthode `set` qui permet de substituer un objet à l'élément courant. La classe `LinkedList` dispose d'une méthode `get( int i )` retournant l'élément d'indice i, mais si vous l'utilisez, c'est que vos données ne devraient pas être stockées sous forme d'une liste chainée ! ! !

2.1.4 les vecteurs ou tableaux dynamiques

La classe `ArrayList` encapsule un tableau classique `Object[]` dans un tableau dynamique. Elle implémente l'interface `List`. Les méthodes `get` et `set` sont donc disponibles pour accéder directement aux éléments d'un `ArrayList`. Il est préférable d'utiliser un `ArrayList` à chaque fois que vous n'avez besoin d'accéder à votre

tableau qu'à partir d'un seul processus, car les méthodes de la classe `Vector` sont toutes synchronisées.

2.1.5 Les tables de hachage

Les listes chaînées et les tableaux vous permettent de spécifier l'ordre (linéaire) dans lequel vous organisez vos éléments. Si l'ordre n'a pas d'importance, il existe des structures de données qui vous permettent de retrouver un élément beaucoup plus rapidement.

Une structure de données classique pour retrouver simplement un élément est la table de hachage. Une table de hachage calcule un nombre entier, appelé code de hachage, pour chacun des éléments. Elle est constituée d'un tableau de listes chaînées.

Un élément est stocké dans la liste correspondant à son code de hachage modulo le nombre de listes. Ainsi pour retrouver un élément, il suffit de réduire son code de hachage modulo le nombre de listes, et parcourir la liste à sa recherche.

Les tables de hachage peuvent servir à implémenter plusieurs structures de données. Par exemple, la classe `HashSet` modélise un ensemble. Un ensemble correspond simplement à une collection d'éléments ne figurant qu'une seule fois chacun dans la collection. La méthode `add` n' ajoute un élément que s'il n'est pas déjà présent.

L'insertion des éléments dans un `HashSet` repose sur les méthodes `HashCode`. Celle qui est fournitepour tout objet n'est pas très utile car elle repose uniquement sur l'adresse mémoire de l'objet. Il faut donc la redéfinir pour qu'elle ne dépende que du contenu de l'objet. Pensez également à redéfinir `equals`!

2.1.6 les collections ordonnées

La classe `TreeSet` repose également sur une table de hachage et permet de stocker des ensembles ordonnés, moyennant une petite pénalité.

La comparaison entre deux éléments repose sur la méthode `compareTo` de l'interface `Comparable`.

2.2 Les vues

Les méthodes des collections ne sont pas synchronisées, ce qui signifie que pour améliorer les performances, il n'y pas pas de protection gérant les accès concurrents de la part des processus. Il est possible néanmoins d'obtenir des vues synchronisées, en faisant appel à certaines méthodes de la classe `Collections`, comme dans l'exemple ci-dessous :

```
HashMap hm = new HashMap();
Map map = Collections.synchronizedMap(hm);
```

Vous pouvez également obtenir une collection en lecture seule (vue non modifiable), comme dans cet exemple

```
List l = new LinkedList();
List l2 = new Collections.unmodifiableList(l);
```

Il est aussi possible de travailler directement avec un sous-ensemble (vue restreinte) : ainsi le code

```
List groupe2 = etudiants.subList(10,20);
```

définit le groupe 2 comme les éléments d'indice 10 à 19.

2.3 Algorithmes

La méthode `Collections.sort` permet de trier une collection ordonnée (interface `List`). Par défaut, elle utilise comme critère de tri l'implémentation de `compare` de la collection à trier. Il est possible de lui préciser un autre critère de tri directement : par exemple

```
Collections.sort( cadres, new Comparator(){
public int compare(Object a, Object b){
double differenceSalaires = (Salarie) a.getSalaire() - (Salarie) b.getSalaire();
if (differenceSalaires < 0 ) return -1;
if (differenceSalaires > 0 ) return 1;
return 0;}
});
```

trie une liste `cadres` d'objets de type `Salarie` en fonction de leur salaire.

Ceci s'applique aussi aux méthodes génériques de

- calcul d'un plus grand élément : `Collections.max` ;
- recherche dichotomique d'un élément : `Collections.binarySearch` (à n'appliquer à des collections triées ;-))

2.4 Exercice

Trier (avec `Collection.sort`) les items de l'aquarium selon la somme de leurs coordonnées, en complétant la méthode `main` ci-dessous :

```
public static void main(String[] args){
List<AquariumItem> collection = new ArrayList();
remplir(collection);
afficher(collection);
bouger(collection);
afficher(collection);
System.out.println("tri selon la somme de leurs coordonnées ...");
Collections.sort(  collection, new Comparator(){
public int compare(Object a, Object b){
*** votre solution ***
}
});
afficher(collection);
}
```

3 Applications

L'application suivante est tirée de "Design Patterns : Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

Reprenons l'exemple du labyrinthe. Voici la nouvelle situation :

La classe **Lieu** est une classe abstraite contenant la méthode abstraite `entrer` (qui réagira en fonction du lieu). Un lieu est spécialisé en **Salle**, **Mur** et **Porte**. Une salle est

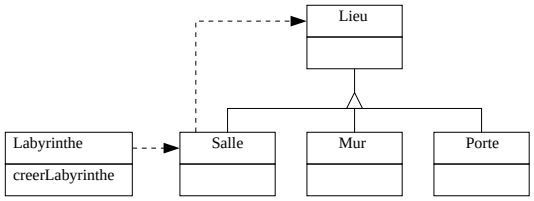

FIGURE 1 – Modèle de classes

composée de lieux : les murs qui l'entourent, les salles attenantes, des portes. Une **Porte** sépare deux **Salle** et peut être ouverte ou fermée. Si notre héros *entre* dans une Salle, la méthode `entrer` correspondante le déplacera. S'il entre dans un mur, la méthode `entrer` correspondante de réagira pas. S'il entre dans une porte, deux possibilités : si la porte est ouverte, alors le héros changera de salle ; sinon il ne bougera pas. Enfin, un labyrinthe est un collection de **Salle**.

1. Implementer les différentes classes. Créer un labyrinthe en implémentant la méthode `creerLabyrinthe` de **Labyrinthe**.

On peut observer que la fonction `creerLabyrinthe` est relativement compliquée. Elle contient l'instanciation de l'ensemble des éléments du labyrinthe et fait le lien entre chaque élément. Par exemple pour créer un labyrinthe contenant deux salles, on peut écrire :

```
public void creerLabyrinthe(){
    Salle s1 = new Salle();
    Salle s2 = new Salle();
    Porte p = new Porte(s1,s2);
    Mur m1 = new Mur();
    Mur m2 = new Mur();
    Mur m3 = new Mur();
    Mur m4 = new Mur();
    Mur m5 = new Mur();
    Mur m6 = new Mur();

    s1.ajouterLieuAdjacent(m1);
    s1.ajouterLieuAdjacent(m2);
    s1.ajouterLieuAdjacent(m3);
    s1.ajouterLieuAdjacent(p);
    s2.ajouterLieuAdjacent(m4);
    s2.ajouterLieuAdjacent(m5);
    s2.ajouterLieuAdjacent(m6);
    s2.ajouterLieuAdjacent(p);

    this.setSalleEntree(s1);
    this.setSalleEntree(s2);
    this.addSalle(s1);
    this.addSalle(s2);
}
```

Le problème de cette fonction membre est son absence de souplesse : elle codifie en dur le plan du labyrinthe ; modifier le plan revient à modifier cette fonction, soit en la modifiant (attention à la création d'erreurs), soit en la surchargeant (il faut tout réécrire).

Supposons que l'on souhaite conserver le **plan** du labyrinthe déjà existant pour un nouveau jeu. Mais dans ce nouveau jeu, le labyrinthe est constitué de nouveaux éléments comme des **SalleEnchantee** contenant par exemple un objet, des **PorteASesame** (porte verrouillée s'ouvrant à l'énoncé d'une formule magique). Comment changer `creerDedale` simplement afin qu'il crée des labyrinthes dotés des ces nouvelles classes d'objets ?

Dans un premier temps, nous allons créer la classe **FabriqueLabyrinthe** qui va créer les différents éléments du labyrinthe : elle construit les salles, les murs, les portes.

Voila :

```
public class FabriqueLabyrinthe{
    public Salle creerSalle(){
return new Salle()
    }
    public Mur creerMur(){
return new Mur();
    }
    public Porte creerPorte(Salle s1,Salle s2){
return new Porte(s1,s2);
    }
}
```

2. Modifier la fonction `creerLabyrinthe` en conséquence en passant la fabrique en paramètre.

On peut observer que, pour l'instant, les avantages de cette implémentation ne sont vraiment évidents : nous avons compléxifié la création d'un labyrinthe en passant par la fabrique. Revenons maintenant à la création de notre nouveau jeu : le labyrinthe est constitué de **SalleEnchantee** et de **PorteASesame** MAIS la plan du labyrinthe est le même (imaginez que nous avons déjà créé un labyrinthe contenant 10 000 lieux soit environ 2000 salles ; on comprend bien que l'on n'a pas envie (ou le temps) de recoder un nouveau labyrinthe). Nous allons voir que l'utilisation de la fabrique prend alors tout son sens.

3. Créer les classes **SalleEnchantee** et de **PorteASesame**. Voir Figure 2.

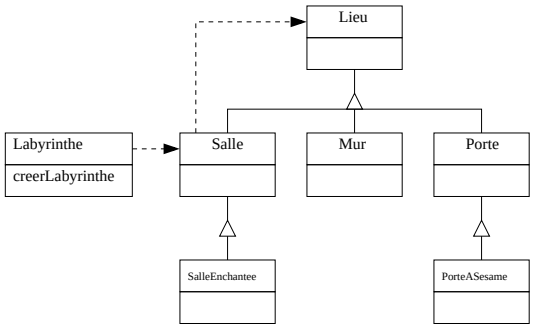

FIGURE 2 – Modèle de classes

4. Créer maintenant un labyrinthe enchanté contenant uniquement des salles enchantées et des portes à sésame ayant le même plan que le labyrinthe que vous avez précedemment créé.

Indice : Écrire la classe FabriqueLabyrintheEnchantee étendant la classe FabriqueLabyrinthe.

Observez que nous n'avons plus à modifier la fonction `creerLabyrinthe`. Il suffit de lui passer en paramètre une "fabrique enchantée" !

5. Compléter votre jeu de labyrinthe : ajouter Bob, etc...