Université Claude Bernard Lyon 1 4A , 2019–2020
Polytech Lyon - Département Informatique

ISI-3

TD3. Design Patterns(correction)

Laëtitia Matignon

1 Design Pattern Observateur/Observé

Soit le code suivant utilisant les classes représentées dans le diagramme de la figure 1 :

```
SujetConcret s = new SujetConcret("PopCorn", 1.29f);
NameObserver nameObs = new NameObserver();
PriceObserver priceObs = new PriceObserver();
s.addObserver(nameObs);
s.addObserver(priceObs);
s.setName("Frosties");
s.setPrice(4.57f);
s.setPrice(9.22f);
s.setName("Smacks");
```

On souhaite obtenir l'affichage suivant lors de l'exécution du code précédent :

```
Name changed to Frosties
Price changed to 4.57
Price changed to 9.22
Name changed to Smacks
```

Question 1 *Proposer une implémentation de sorte à obtenir l'affichage souhaité pour :*
- *les méthodes* `setName` *et* `setPrice` *de la classe* `SujetConcret`
- *les méthodes* `update` *pour chaque* `Observer`*,*

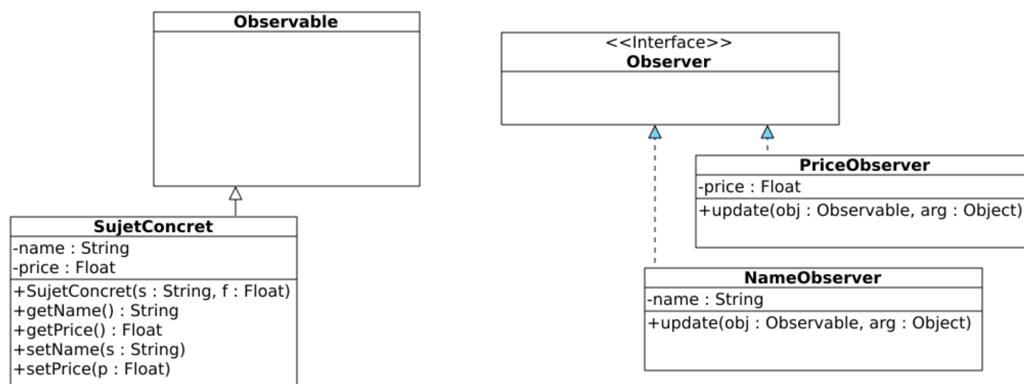


Figure 1 – Design Pattern Observateur Observé

Lorsque le sujet change de nom ou de prix (`setName` ou `setPrice`), il faut notifier l'observateur correspondant (méthode `notifyObservers` d'`Observable` qui appelle `update` sur chaque `Observer` (cf. slide 98 du CM2)). Les observateurs lorsqu'il sont notifiés vont afficher la phrase correspondante.
La difficulté ici est que vous avez 2 observateurs. A partir du moment où vous appelez `notifyObservers` dans `setName` et `setPrice`, **les 2 observateurs sont notifiés** et exécutent leur méthode `update`. Il faut donc trouver un moyen pour que seul `NameObserver` affiche quelque chose lorsque `setName` notifie, et seul `PriceObserver` affiche quelque chose lorsque `setPrice` notifie.

V1 : L'observé (`SujetConcret`) donne des informations (via la méthode `setName` et `setPrice`) qui sont de types différents (Float et String). On peut donc **pousser** des arguments via la méthode `notifyObservers` et son paramètre `Object arg`. Les arguments **poussés** sont récupérés dans la méthode `update` de l'observateur. Chaque observateur vérifie le type de l'argument poussé pour savoir s'il doit afficher quelque chose :

```
public void setName(String name) {
this.name = name;
setChanged();//NE PAS OUBLIER: TOUJOURS APPELER setChanged AVANT notify (cf.
    slide 98 du CM2): notify appele update si booleen changed mis a true ...
notifyObservers(name);// pousse argument recupere dans parametre arg de
    update
}
public void setPrice(float price) {
this.price = price;
setChanged();
notifyObservers(new Float(price));//pousse argument recupere dans parametre
    arg de update
}
public class NameObserver implements Observer {
 public void update(Observable obj, Object arg) {
  if (arg instanceof String) {//cet observateur verifie le type de l'argument
      pousse: si c'est un String, il affiche, sinon, il n efait rien
   name = (String)arg;
   System.out.println("Name changed to " + name);}}
public class PriceObserver implements Observer {
 public void update(Observable arg0, Object arg) {
   if (arg instanceof Float) {
    Float prix = (Float)arg; //cet observateur verifie le type de l'argument
        pousse: si c'est un Float, il affiche, sinon, il n efait rien
    System.out.println("Price change to "+ prix);}}}
```

V2 : la solution v1 fonctionne car les types des éléments poussés sont différents. Une autre solution consiste à ce que l'observateur tire des informations de l'observé, et vérifie si cette information a changé. Si oui, il devra afficher le nouveau prix ou nom. il faut donc que les observateurs :

- mémorisent des informations (ajout d'un attribut)
- récupérent l'instance qui les a notifié : cela se fait avec le paramètre `Observable o` dans la méthode `update`.

```java
public static void main(String[] args) {
      SujetConcret s = new SujetConcret("PopCorn", 1.29f);
      //modification pour utiliser le constructeur avec parametre
      NameObserver nameObs = new NameObserver(s.getName());
      PriceObserver priceObs = new PriceObserver(s.getPrice());
      s.addObserver(nameObs);
      s.addObserver(priceObs);
      s.setName("Frosties");
      s.setPrice(4.57f);
      s.setPrice(9.22f);
      s.setName("Smacks");
   }

public void setName(String name) {
this.name = name;
setChanged();
notifyObservers();//on ne pousse plus d'information ici contrairement a la V1
}
public void setPrice(float price) {
this.price = price;
setChanged();
notifyObservers();
}


public class NameObserver implements Observer {
   String name ;
   public NameObserver(String name) {
      super();
      this.name = name;
   }
   @Override
   public void update(Observable o, Object arg) {
      if (o instanceof SujetConcret){//recupere dans o celui qui l'a notifie,
           verifie si c'est SujetConcret
         String n = ((SujetConcret)(o)).getName(); //l'observateur tire des
             informations sur l'observable
         if (!n.equals(name) ){
            name = n;
            System.out.println("Name changed to "+n);
         }
      }
   }
}
```

```java
public class PriceObserver implements Observer {
   Float price;
      public PriceObserver(Float price) {
      super();
      this.price = price;
   }
      @Override
      public void update(Observable o, Object arg) {
         if (o instanceof SujetConcret){//recupere dans o celui qui l'a
             notifie, verifie si c'est SujetConcret
            Float p = ((SujetConcret)(o)).getPrice(); //l'observateur tire des
                 informations sur l'observable
            if (p != price ){
               price = p;
               System.out.println("Price changed to "+p);
            }
         }
      }
}
```

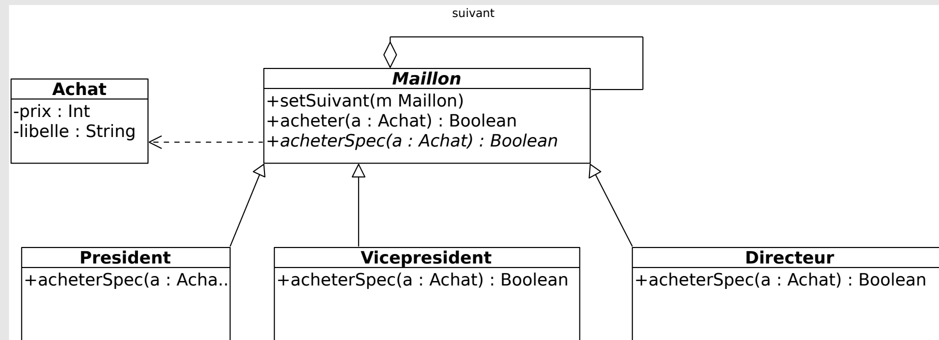
2 Design Pattern

Le code ci-dessous sert à tester l'énoncé suivant : *Le directeur s'occupe des achats. S'ils sont inférieur à 100 euro, il peut donner son accord. S'ils sont supérieurs à 100 euro mais inférieur à 100000 euro, il faut l'accord du vice-président. Au delà, il faut l'accord du président.*

```
President eric = new President();
Vicepresident antoine = new Vicepresident();
Directeur samuel = new Directeur();
samuel.setSuivant(antoine);
antoine.setSuivant(eric);
Achat p = new Achat(50, "Formation"); samuel.acheter(p);
p = new Achat(24000, "Voiture"); samuel.acheter(p);
p = new Achat(150000, "Maison"); samuel.acheter(p);}
```

Question 2 *Proposer un design pattern et son implémentation pour faire fonctionner le programme précédent.*

On utilise le design pattern **Chaine de responsabilité**. Plusieurs objets (`directeur, president, vice-president`) peuvent répondre à la même requête (`acheter`) mais cela n'est pas connu par le client (le client est ici le code ci-dessus) qui appelle la requête `acheter` toujours sur le même objet (le directeur). Avec la chaine de responsabilité, la requête passe de maillon en maillon automatiquement (diminution du couplage entre les classes capable de répondre à la requête : chaque maillon n'est couplé qu'au successeur ; on peut changer l'ordre d'exécution des maillons de la chaine sans modifier le client).




```
public abstract class Maillon { private Maillon suivant;
  public setSuivant(Maillon m){suivant = m;}
   public boolean acheter(Achat a) {//requete utilise par le client ; ici
       cette methode suit le design pattern "patron de methode": succession d'
       etapes/algo dont certaines parties peuvent etre differentes (ici l'
       etape acheterSpec sera differente dans chaque classe concrete)
        if(acheterSpec(a)) {return true; };
        if(suivant != null) { return suivant.acheter(a); }
        return false; }
  public abstract boolean acheterSpec(Achat a) ();
  }
public class Directeur extends Maillon {
    public boolean acheterSpec(Achat a) {
        if(a.getPrix() <= 100) {...; return true; }
        return false; }}
public class VicePresident extends Maillon {
    public boolean acheterSpec(Achat a) {
        if(a.getPrix() > 100 & a.getPrix() < 100000 ) {...; return true; }
        return false; }}
```

3 Design Pattern Fabrique

Soit le diagramme de classes de la figure 2. On peut, à partir de ces classes, créer différents types de labyrinthes.

Pour créer un labyrinthe classique, on utilise la méthode `createMaze()` de la classe `MazeGame` qui se charge d'instancier un labyrinthe avec des éléments classiques de type `Room`, `Door`, `Wall`. Par exemple on a le code suivant :

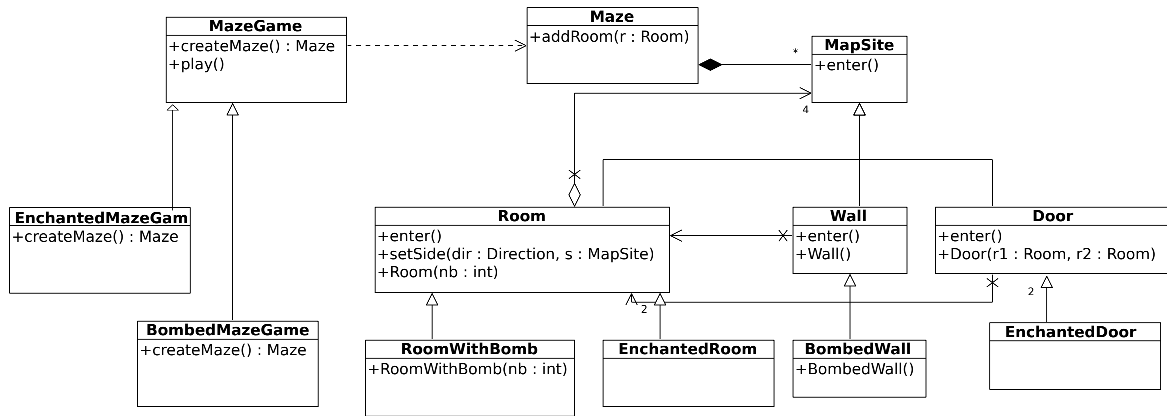


FIGURE 2

```java
class MazeGame {
void Play() {...}
public Maze createMaze() {
Maze aMaze =new Maze(); Room r1 = new Room(1);Room r2 = new Room(2)
Door theDoor = new Door(r1, r2); aMaze.addRoom(r1);aMaze.addRoom(r2)
r1.setSide(North, new Wall()); r1.setSide(East, theDoor);
r1.setSide(South, new Wall()); r1.setSide(West, new Wall());
r2.setSide(North, new Wall());r2.setSide(East, new Wall());
r2.setSide(South, new Wall());r2.setSide(West, theDoor);
return aMaze;}}
```

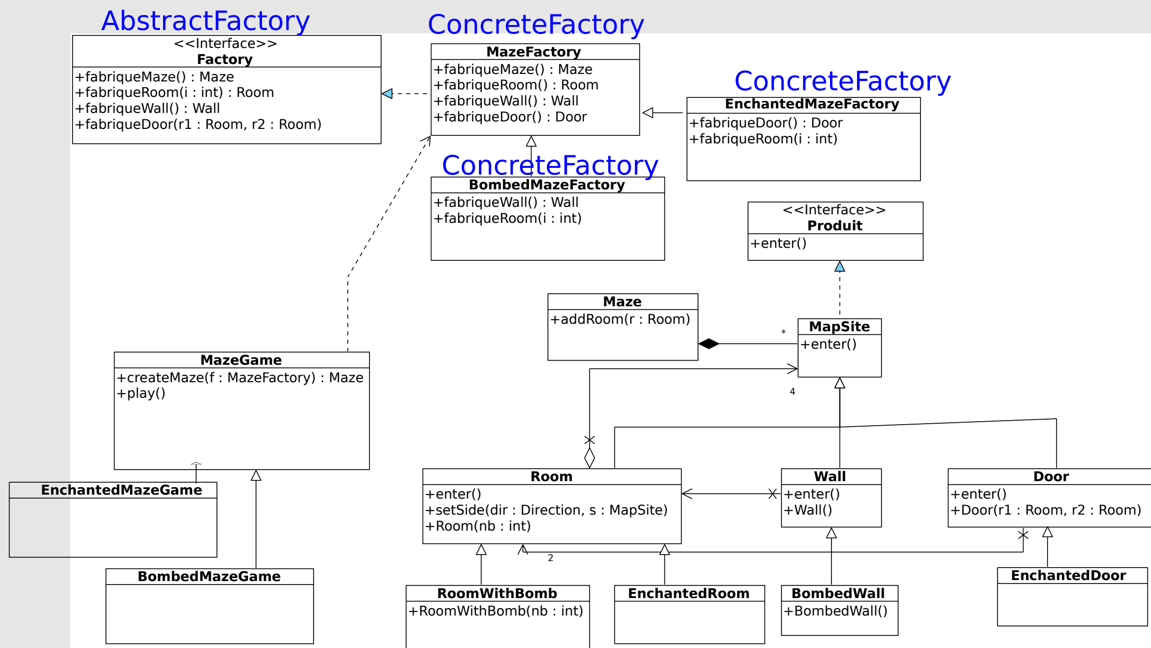
Pour créer un labyrinthe avec des bombes, on utilise la méthode `createMaze()` de la classe `BombedMazeGame` qui se charge d'instancier un labyrinthe avec des éléments de type `RoomWithABomb`, `Door`, `BombedWall`. Par exemple on a le code suivant :

```java
class BombedMazeGame extends MazeGame {
public Maze createMaze() {
Maze aMaze = new Maze();Room r1 = new RoomWithABomb(1);
Room r2 = new RoomWithABomb(2);Door theDoor = new Door(r1, r2);
aMaze.addRoom(r1);aMaze.addRoom(r2);
r1.setSide(North, new BombedWall());r1.setSide(East, theDoor);
r1.setSide(South, new BombedWall());r1.setSide(West, new BombedWall());
r2.setSide(North, new BombedWall());r2.setSide(East, new BombedWall());
r2.setSide(South, new BombedWall());r2.setSide(West, theDoor);
return aMaze;}}
```

De même pour un labyrinthe enchanté. **Dans tous les cas, l'organisation du labyrinthe est la même (nombre et position des éléments).**

Question 3 *Proposer une application du design pattern* ***Fabrique abstraite*** *permettant de n'avoir qu'une seule méthode* `createMaze`*. Cette méthode créera les éléments du labyrinthe sans connaitre leur type précis.*

La méthode `createMaze` de la classe `MazeGame` va se charger de créer un labyrinthe selon une organisation/squelette précise (nombre et position des éléments toujours les mêmes), mais les éléments créés pourront être différents car la méthode `createMaze` va utiliser des **méthodes fabrique** pour créer les éléments. Les méthodes fabrique vont instancier le type de chaque élément en fonction de la **fabrique passée en paramètre** de `createMaze`.




```
class MazeFactory implements Factory {//fabrique concrete , chaque methode instancie un element de chaque type (Wall, Room, Door, ...)
public Maze fabriqueMaze() { return new Maze();}
public Wall fabriqueWall() { return new Wall();}
public Room fabriqueRoom(int i) {return new Room(i);}
public Door fabriqueDoor(Room r1,Room r2) {return new Door(r1,r2);}
...}

class BombedMazeFactory extends MazeFactory {
public Wall fabriqueWall() {return new BombedWall();}
public Room fabriqueRoom(int i) {return new RoomWithABomb(i);}}

class EnchantedMazeFactory extends MazeFactory {
public Door fabriqueDoor(Room r1,Room r2) {return new EnchantedDoor(r1,r2);}
public Room fabriqueRoom(int i) {return new EnchantedRoom(i);}}

class MazeGame {
public Maze createMaze(MazeFactory f) {//squelette de creation du labyrinthe avec des methodes fabrique
Maze aMaze = f.fabriqueMaze();
Room r1 = f.fabriqueRoom(1);Room r2 = f.fabriqueRoom(2);//selon la fabrique passee en parametre (f), des pieces noramles ou enchantee ou autre seront instanciees ici
Door theDoor = f.fabriqueDoor(r1, r2);
aMaze.addRoom(r1);aMaze.addRoom(r2);
r1.setSide(North, f.fabriqueWall());r1.setSide(East, theDoor);
r1.setSide(South, fabriqueWall());r1.setSide(West, fabriqueWall());
...}
```