Université Claude Bernard Lyon 1
Polytech Lyon - Département Informatique

4A Apprentis, 2019–2020

ISI-3

TD2. Principes avancés de conception orientée objet et Design Pattern.

Laëtitia Matignon

---

1 Inversion des dépendances

Nous allons voir dans cet exercice 2 manières de casser les dépendances cycliques afin de respecter le **principe des dépendances acycliques entre packages** (cf. slide 99 du Cours 1).

1.1 Dépendances cycliques

Soit les 2 classes `Synchronisateur` et `Calculateur` représentées sur la figure 1.1. Un calculateur permet d'effectuer des calculs. Etant donné que n'importe qui peut demander à un calculateur d'effectuer des calculs, la classe `Synchronisateur` a été construite pour réguler les calculs.

Les personnes qui souhaitent demander la réalisation d'un calcul doivent passer par le synchronisateur (*via* l'opération `calculer()`). Celui-ci distribue les calculs aux différents calculateurs avec lesquels il est lié (c'est le synchronisateur qui appelle l'opération `calculer()` sur les calculateurs). Un calculateur connaît le synchronisateur auquel il est relié grâce à l'attribut `sync` de type `Synchronisateur`. Sa valeur doit être déterminée lors de la création des objets de type `Calculateur`.

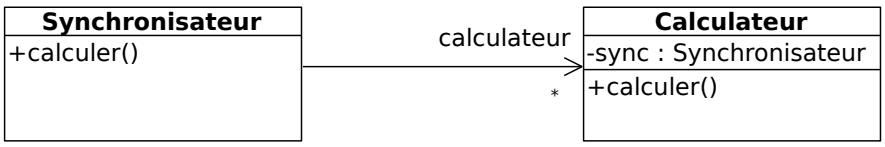


FIGURE 1 –

Question 1 *Exprimez en les justifiant les dépendances entre les classes* ***`Synchronisateur`*** *et* ***`Calculateur`****.*

1.2 Méthode 1

Nous souhaitons que les classes `Synchronisateur` et `Calculateur` soient dans deux packages différents. Dans ce cas, le **principe des dépendances acycliques entre**

packages (cf. slide 99 du Cours 1) n'est pas respecté. L'intérêt de ce principe est de **renforcer le couplage faible entre les classes**.

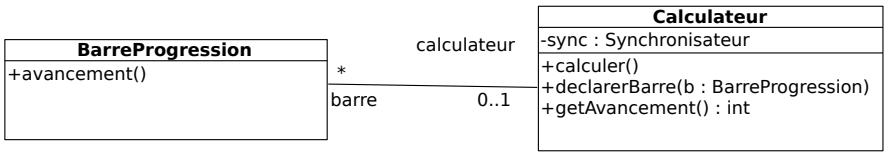
Question 2 *Proposez une solution respectant ce principe, en appliquant la méthode proposée au slide 100 du Cours 1.*

1.3 Méthode 2 : Application d'un patron de conception

Nous souhaitons ajouter à la classe `Synchronisateur` une opération `ajouterCalculateur()` qui permet d'assigner un calculateur à un synchronisateur, l'identité du calculateur étant un paramètre d'entrée de l'opération.

Question 3 *Compléter le diagramme UML obtenu à la question précédente en y ajoutant cette opération correctement définie.*

Nous souhaitons maintenant définir une classe représentant une barre de progression. Cette barre affiche l'état d'avancement du calcul (en pourcentage) d'un calculateur. Une barre de progression reçoit des messages d'un calculateur qui l'informe que l'état d'avancement du calcul a changé. Une barre se déclare auprès d'un `Calculateur` et est associée à au plus un `Calculateur`. Le `Calculateur` offre une opération permettant de connaître le pourcentage d'avancement du calcul.

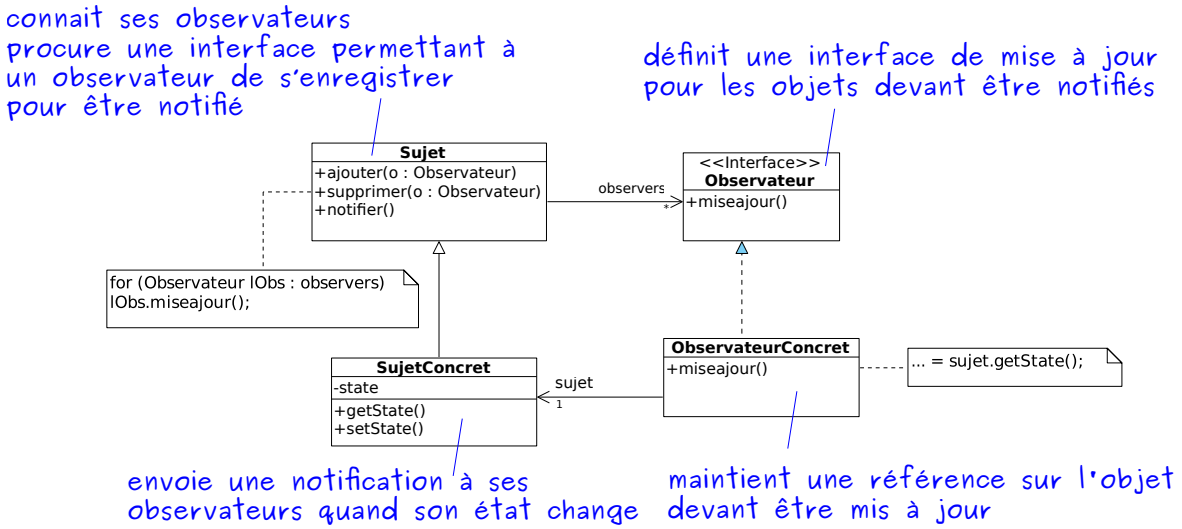


Question 4 *Nous souhaitons que ces deux classes soient dans deux packages différents. Proposez une solution pour respecter le principe des dépendances acycliques en appliquant le patron de conception* ***`Observer`*** *(cf. cours2 sur les design patterns et rappels en fin de ce TD).*

Rappels : Pattern Observateur/Observé (*Observer*)

Les objets observés (`Sujet`) enregistrent dynamiquement des observateurs (`Observateur`) et les notifient des changements lorsque cela est utile (`notifier()`).

- Faible couplage entre `ObservateurConcret` et `SujetConcret`
- Les données de `Sujet` peuvent être "poussées" (dans *notifier*) ou "tirées" (avec des getters)
- Se retrouve dans de nombreuses API Java



2 Design Pattern Observateur/Observé

Soit le code suivant utilisant les classes représentées dans le diagramme de la figure 2 :

```
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(4.575f);
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 5 *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`,

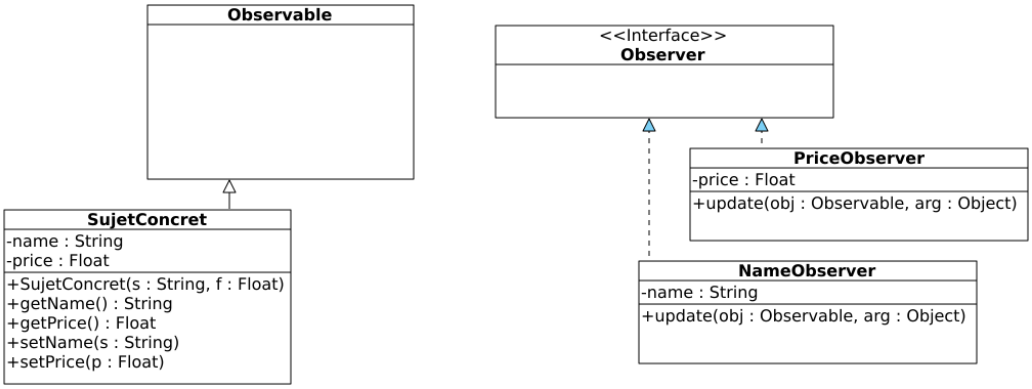


FIGURE 2 – Design Pattern Observateur Observé

3 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 6 *Proposer un design pattern et son implémentation pour faire fonctionner le programme précédent.*

4 Design Pattern Fabrique

Soit le diagramme de classes de la figure 3. 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 :

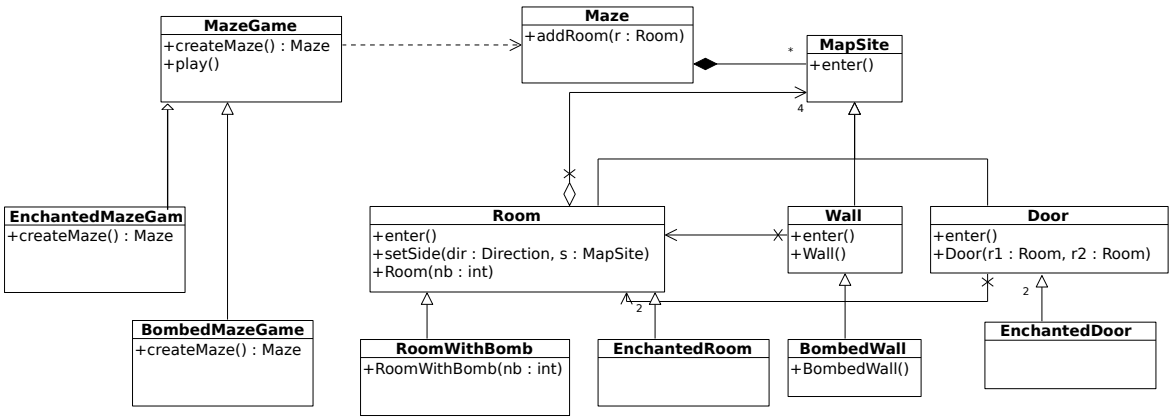


FIGURE 3 –

```
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 :

```
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 7 *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.*