Introduction aux design patterns

Michel Poriaux

HEPH Condorcet

Les Design Patterns c’est quoi ???

- Diagrammes UML qui forment une solution à un problème de POO connu et fréquent.
- Introduits en 1995 dans le livre « Design Patterns – Elements of Reusable Object-oriented Software » par le Gang of Four(GoF) :
  - Eric Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides
- Se subdivisent en 3 catégories:
  - Patterns de constuction ➔création d’objets
  - Patterns de structuration ➔ hiérarchisation et relation entre classes
  - Patterns de comportement ➔ interactions et répartition des traitements entre objets

Petit rappel

- Une application orientée objet est constituée d'objets qui s'envoient des messages.
- Par message on entend l'activation d'une méthode de l'objet auquel on s'adresse.
- Exemple :

Dans l'application « garage et voitures (1-1) », on désire que disposant d'un certain objet garage, on puisse indiquer l'immatriculation de la voiture qui s'y gare :

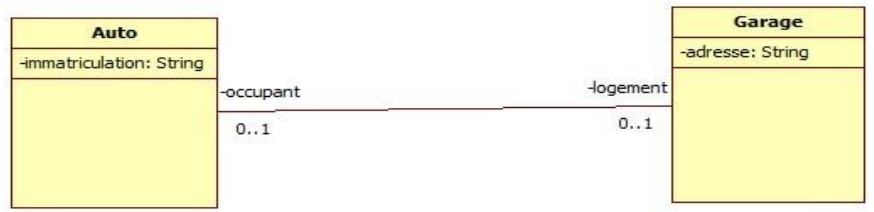
On a développé dans la classe Garage la méthode :

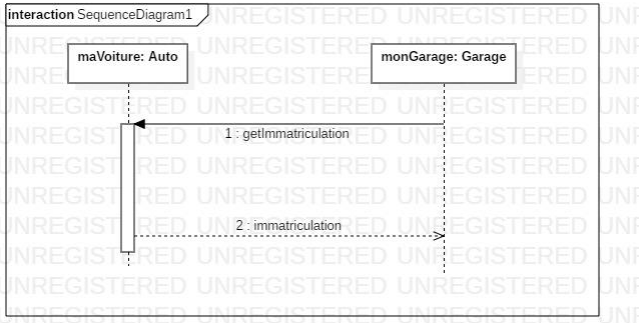
```
public String getInfosOccupant(){
  if(occupant !=null) return occupant.getImmatriculation();
   else return « pas d'occupant actuellement »;
}
```

main :

```
Garage monGarage = new Garage(« Mons »);
Auto maVoiture =new Auto(« ABC123 »);
monGarage.setOccupant(maVoiture);
println(monGarage.getInfosOccupant());// ➔ ABC123
```

- Cette manipulation n'a été possible que parce que l'objet garage dispose d'une référence,d'un lien (variable d'instance de type Voiture) vers l'objet représentant son occupant.
- En d'autres termes, il y a une association entre ces deux classes.
- Au niveau des diagrammes UML cette situation et cet envoi de message se représente comme suit :





Problème du couplage et de la dépendance

- Plus il existe d'associations (de couplage) entre des classes et plus on risque de devoir faire des mises à jour dans d'autres classes quand on apporte une modification à l'une d'entre elles(dépendance).
- Exemple :

L'immatriculation de la voiture devient un nombre entier

➔ le code de la classe Garage devient :

```
public int getInfosOccupant(){
  if(occupant !=null) return occupant.getImmatriculation();
   else return 0;
}
```

➔ Le code de l'application principale devient :

main :

```
int immatriculation = monGarage.getInfosOccupant();
if(immatriculation ==0) println(« aucun occupant actuellement »);
else println(imm);
```

- Remarque : du fait que la navigabilité à sens unique diminue le couplage entre les classes (pas de référence vers l'autre classe dans l'une des deux classes), elle permet de réduire les problèmes de dépendance.
- Conclusion :
  - par rapport aux associations issues de l'analyse conceptuelle, il faut dans la mesure du possible éviter de créer des associations supplémentaires entre les classes lorsque des messages doivent être échangés entre les objets qu'elles génèrent.
  - Les design patterns offrent des solution élégantes à cette contrainte.

Patterns de construction

Le pattern singleton 1/2

Problème à résoudre :

On désire enregistrer dans un objet journal toutes les créations d'objet ayant été réalisées dans l'application magasin. Puis, lorsque le programme se termine, afficher la liste des objets créés.

Mauvaise solution :

```
public class Journal {

    private List<String> lignes;

public Journal(){lignes=new ArrayList<>();}

   public List<String> getLignes() {return lignes;}

   }
```

```
public class Client {

private int idclient; etc

private Journal journal;

public Client(int idclient, String nom, String prenom, int cp, String
localite, String rue, String num, String tel) {

        this.idclient = idclient; etc …

        journal = new Journal();

        journal.getLignes().add("creation de "+ toString());

    }
```

```
public class Produit {

private int idproduit;etc

private Journal journal;

 public Produit(int idproduit,String numprod, String description, float
phtva, int stock)  {

        this.idproduit=idproduit;etc

journal = new Journal();

        journal.getLignes().add("creation de "+ toString());

}
```

```
 public static void main(String[] args) {

Journal journal=new Journal();

        Client cl=new Client(1,"Durand","Jean",1000,"BXL","de la
Senne","12A","0456/990088");

        Produit pr=new Produit(1,"P00001","écran",100f, 50);

        for(String l:journal.getLignes()){

            System.out.println(l);

        }
```

Résultat attendu :
Création de client 1 Durant etc
Création de produit 1 P00001 etc

Résultat obtenu :

L'objet journal n'est pas unique , chaque client, produit et même l'application principale en possède un exemplaire différent

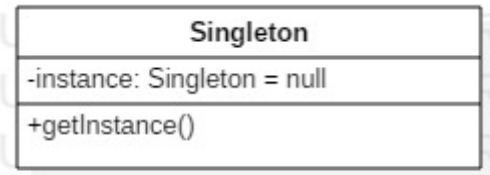
Le pattern singleton 2/2

- Le pattern singleton permet de s'assurer que la classe qui génère les objets demandés renvoie toujours le même objet.
- Bonne solution :

```
public class Journal {
    private List<String> lignes=new ArrayList<>()
    private static Journal journal;
    private Journal(){
        lignes=new ArrayList<>();
    }

    public static Journal getInstance(){
        if(journal==null) journal=new Journal();
        return journal;
    }

    public List<String> getLignes() {
        return lignes;
    }
}
 public Client(int idclient, String nom, String prenom, int cp, String
localite, String rue, String num, String tel) {
        this.idclient = idclient; etc
Journal.getInstance().getLignes().add("creation de "+this);
    }
 public Produit(int idproduit,String numprod, String description, float
phtva, int stock)  {
        this.idproduit=idproduit;etc
Journal.getInstance().getLignes().add("creation de "+this);

    }
```

```
public static void main(String[] args) {
        Client cl=new Client(1,"Durand","Jean",1000,"BXL","de la
Senne","12A","0456/990088");
        Produit pr=new Produit(1,"P00001","écran",100f, 50);
        for(String l:Journal.getInstance().getLignes()){
            System.out.println(l);
        }
    }
}
```

Résultat obtenu:
Création de client 1 Durant etc
Création de produit 1 P00001 etc



Le pattern builder 1/3

- Problème :

On désire prévoir plusieurs manières de construire un objet sachant que certains éléments à communiquer au constructeur sont obligatoires. Par exemple pour la classe client les paramètres nom,prénom et tel sont obligatoires mais les autres sont facultatifs.

- Mauvaise solution 1 : le constructeur « téléscopique »
  - public Client(String nom,String prenom,String tel)
  - public Client(String nom,String prenom,String tel,int cp)
  - public Client(String nom,String prenom,String tel,int cp,String localite)
  - Etc ➔ Ingérable !
- Mauvaise solution 2 : le constructeur avec informations de base obligatoires + setters

public Client(String nom,String prenom,String tel)

public void setNom(String nom) …

public void setPrenom(String prenom)…

public void setTel(String tel) …

public void setCp(int cp) … etc

main

Client cl = new Client(…,…,…);

cl.setCp(…);

➔ Problèmes:

- Comment s'assurer que les setters sont cohérents (exemple : indiquer un numéro de rue sans indiquer de rue).
- Comment permettre à l'utilisateur d'utiliser les setters dans l'ordre qu'il désire ?

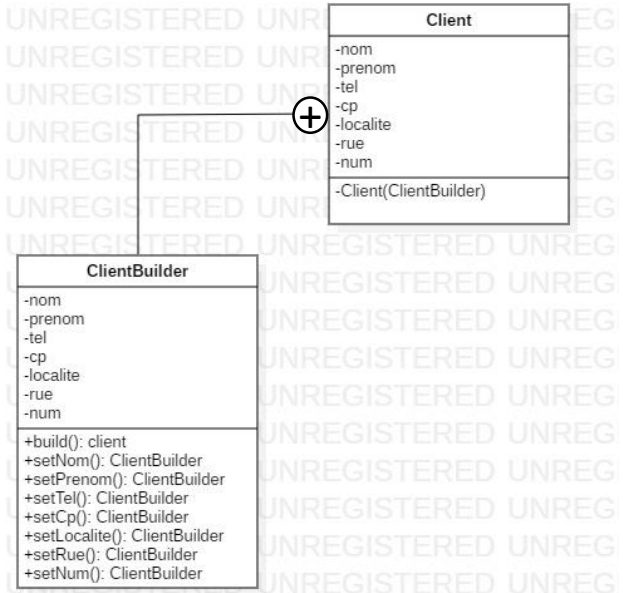
Le pattern builder 2/3

```
public class Client {
protected int idclient;
protected String nom;
protected String prenom;
protected int cp;
protected String localite;
protected String rue;
protected String num;
protected String tel;
 protected Set<ComFact> mesCommandes = new HashSet<>();
  private Client(ClientBuilder cb) {
        this.idclient = cb.idclient;
        this.nom = cb.nom;
        this.prenom = cb.prenom;
        this.cp = cb.cp;
        this.localite = cb.localite;
        this.rue = cb.rue;
        this.num = cb.num;
        this.tel = cb.tel;
    }

public int getIdclient() {
        return idclient;
    }
//tous les getters mais aucun setter



 public static class ClientBuilder{
protected int idclient;
protected String nom;
protected String prenom;
protected int cp;
protected String localite;
protected String rue;
protected String num;
protected String tel;
   public ClientBuilder setIdclient(int idclient) {
           this.idclient = idclient;
           return this;
        }
        public ClientBuilder setNom(String nom) {
           this.nom = nom;
           return this;
        }
        public ClientBuilder setPrenom(String prenom) {
           this.prenom = prenom;
           return this;
        }
        public ClientBuilder setCp(int cp) {
           this.cp = cp;
           return this;
        }
        public ClientBuilder setLocalite(String localite) {
           this.localite = localite;
           return this;
        }

        public ClientBuilder setRue(String rue) {
           this.rue = rue;
           return this;
        }

        public ClientBuilder setNum(String num) {
           this.num = num;
           return this;
        }

        public ClientBuilder setTel(String tel) {
           this.tel = tel;
           return this;
        }
            public Client build() throws Exception{
           if(idclient<=0 || nom==null || prenom==null || tel==null) throw new
Exception("informations de construction incomplètes");
           return new Client(this);
        }

 }
} //fin de la classe Client
```

Le pattern builder 3/3

```
public static void main(String[] args)  {

    try {
       Client cl1 = new Client.ClientBuilder().
             setIdclient(1).
             setNom("Durant").
             setPrenom("Eric").
             setTel("0455/332211").
             setLocalite("Mons").
             build();
       System.out.println(cl1);
    } catch (Exception e) {
       System.out.println("erreur "+e);
    }

       try {
       Client cl2 = new Client.ClientBuilder().
             setIdclient(1).
             setNom("Durant").
             setPrenom("Eric").
             setLocalite("Mons").
             build();
          System.out.println(cl2);
    } catch (Exception e) {
       System.out.println("erreur "+e);
    }
   }
```

Client{idclient=1, nom=Durant, prenom=Eric, cp=0, localite=Mons, rue=null, num=null, tel=0455/332211}
erreur java.lang.Exception: informations de construction incomplètes



Patterns de structuration

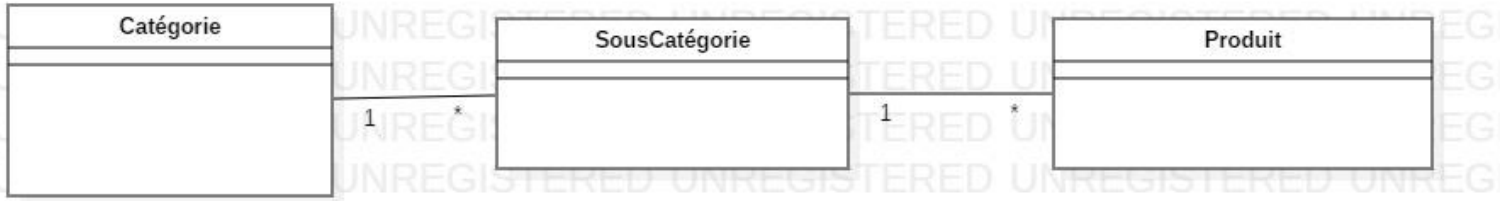
Le pattern composite 1/3

- Problème :

On désire gérer des produits organisés en catégories comprenant elles-mêmes des sous catégories, l'arborescence ayant une profondeur indéfinie. On désire notamment calculer facilement la valeur en stock de chaque produit mais aussi de chaque catégorie sur base des produits et des sous-catégories qu'elle comprend.

- Mauvaise solution :

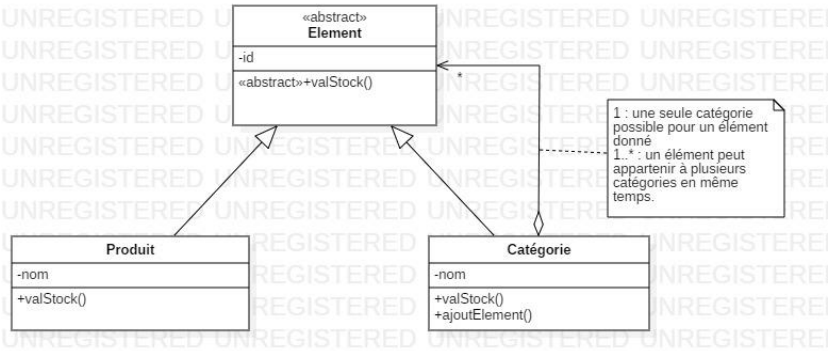
Créer une arborescence de profondeur fixe :



- On ne peut pas avoir des catégories comprenant à la fois des produits et des sous-catégories
- On ne peur pas étendre l'arborescence à des sous-niveaux supplémentaires

Le pattern composite 2/3

```
public class Categorie extends Element {
    private String nom;
    private Set<Element> elts=new HashSet<>();

    public Categorie(int id,String nom){
        super(id);
        this.nom=nom;
    }
    @Override
    public float valStock() {
        float somme=0;
       for(Element pc:elts){
           somme+=pc.valStock();
       }
       return somme;
    }

    public Set<Element> getElts() {
        return elts;
    } @Override
    public String toString() {
        StringBuilder aff= new StringBuilder(getId()+" "+nom+"\n");

        for(Element e:elts){
            aff.append(e+"\n");
        }
        return aff+"valeur totale " +nom +" = "+valStock()+"\n";
    }
   }
```




```
public abstract class Element {
    private int id;
    public Element(int id){
        this.id=id;
    }
    public int getId() {
        return id;
    }
  public abstract float valStock();
}
```

extends Element

```
 public class Produit {
  // voir avant
        @Override
     public float valStock(){
         return stock*phtva;
     }
```

Le pattern composite 3/3

```
public class Magasin {

    public static void main(String[] args) {
        Produit p1 = new Produit(1, "P00001", "écran", 200, 10);
        Produit p2 = new Produit(2, "P00002", "souris", 15, 100);
        Produit p3 = new Produit(3, "P00003", "clavier", 12, 10);
        Produit p4 = new Produit(4, "P00004", "clé usb", 20, 15);
        Categorie c1 = new Categorie(5, "générale");
        Categorie c2 = new Categorie(6, "cat1");
        Categorie c3 = new Categorie(7, "cat2");
        c1.getElts().add(p1);
        c1.getElts().add(c2);
        c1.getElts().add(c3);
        c2.getElts().add(p2);
        c2.getElts().add(p3);
        c3.getElts().add(p4);
        System.out.println(c1);

    }
}
```

```
5 générale
Produit{idproduit=1, numprod=P00001, description=écran, phtva=200.0, stock=10 valeur=2000.0}
6 cat1
Produit{idproduit=2, numprod=P00002, description=souris, phtva=15.0, stock=100 valeur=1500.0}
Produit{idproduit=3, numprod=P00003, description=clavier, phtva=12.0, stock=10 valeur=120.0}
valeur totale cat1 = 1620.0

7 cat2
Produit{idproduit=4, numprod=P00004, description=clé usb, phtva=20.0, stock=15 valeur=300.0}
valeur totale cat2 = 300.0

valeur totale générale = 3920.0
```

Patterns de comportement

Le pattern observer 1/3

- Problème :

On désire les clients qui le désirent soient « notifiés »de la modification du prix des articles qu'ils « suivent ».

- Mauvaise solution:

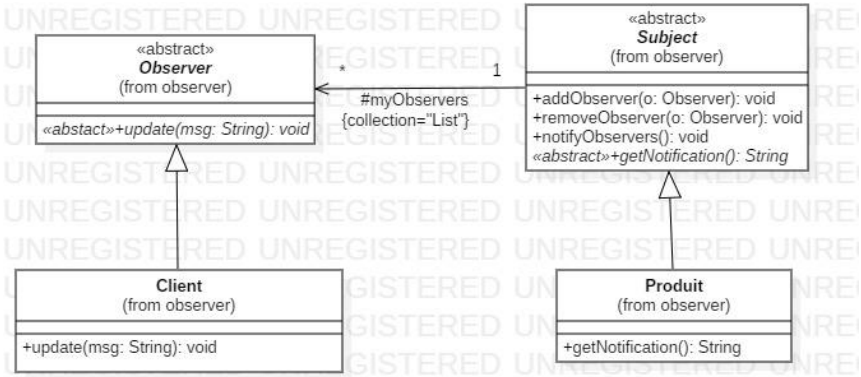
Créer une association entre les produits et les clients afin que les produits puissent envoyer des messages aux clients avec lesquels ils ont un lien.

➔On crée un dépendance entre les clients et les produits

➔Si on désire que d'autres objets issus d'autres classes soient avertis du changement, il faut créer une relation pour ces classes également.

Le pattern observer 2/3

```
public abstract class Observer {

    public abstract void update(String msg);
}


public abstract class Subject {
   protected List<Observer> myObservers = new ArrayList<>();

   public void addObserver(Observer o){
       myObservers.add(o);
   }
   public void removeObserver(Observer o){
       myObservers.remove(o);
   }
   public void notifyObservers(){
       String msg=getNotification();
       for(Observer o : myObservers) o.update(msg);
   }
   public abstract String getNotification();
 }



public class Client extends  Observer{

 //voir avant


    @Override
    public void update(String msg) {
        System.out.println(prenom+" "+nom+" a reçu le msg :"+msg);
    }

}
```




```
public class Produit extends Subject{

//voir avant

   public void setPhtva(float phtva) {
        this.phtva = phtva;
        notifyObservers();
    }
    @Override
    public String getNotification() {
        return "nouveau prix de "+numprod+" = "+phtva;
    }

}
```

Le pattern observer 3/3

```
public class Magasin {
    public static void main(String[] args) {
         Produit p1 = new Produit(1, "P00001", "écran", 200, 10);
         Produit p2 = new Produit(2, "P00002", "souris", 15, 100);
         Client cl1= new Client(1,"Durand","Jean",1000,"BXL","de la
Senne","12A","0456/990088");
         Client cl2= new Client(2,"Dupond","Annie",1000,"BXL","de la
bière","14","0451/441122");
         p1.addObserver(cl1);
         p1.addObserver(cl2);
         p2.addObserver(cl1);

         p1.setPhtva(210);
         p2.setPhtva(12);
    }
}
```

Jean Durand a reçu le msg :nouveau prix de P00001 = 210.0
Annie Dupond a reçu le msg :nouveau prix de P00001 = 210.0
Jean Durand a reçu le msg :nouveau prix de P00002 = 12.0

FIN