Conception et Programmation Objet Avancées
Modèles de conception

Petru Valicov
petru.valicov@umontpellier.fr

https://github.com/IUTInfoMontp-M3105

2019-2020

Contexte

- Des années d'expérience de la POO
- Des projets de très grande taille
- Une approche modulaire de la programmation
- Construction progressive d'une véritable expertise dans la réalisation d'architecture de programmes orientés objets

On veut capitaliser cette expérience et ne pas réinventer la roue.

Rule of thumb

Someone else has already solved your problem !

Modèles de Conception

- description des problèmes récurrents
- des solutions suffisamment générales et bien connues
- des solutions flexibles et extensibles
- des solutions indépendantes des langages de programmation

Définition

Un **modèle de conception** *(design patterns) décrit une structure commune et répétitive de composants en interaction qui résout un problème de conception dans un contexte particulier.*

23 modèles du *"Gang of Four" (GoF)* :

Design Patterns. Elements of Reusable Object Oriented Software.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Addison Wesley, 1995.

Vous en avez déjà utilisé sans (peut-être) le savoir

Modèle de Conception - formalisation

Un DP comporte différents éléments :

- le **nom** (un ou deux mots) - permet de reconnaître le modèle et indiquer son utilisation
- le **problème** - description de l'objectif du modèle et du contexte décrivant les circonstances d'utilisation du modèle
- la **solution** - décrit le schéma de conception résolvant le problème
- les **conséquences** - compromis espace/temps/complexité d'implémentation

Modèles de Conception - utilité

Un bon modèle de conception :

- résout un problème récurrent
- correspond à une solution éprouvée
- est généralement indépendant du langage de programmation
- favorise la réutilisabilité, l'extensibilité, etc.

Méfiez-vous :

- pas de magie $\Longrightarrow$ il faut quand-même réfléchir
- pas de solution universelle prête à l'emploi
- pas de bibliothèque de classes réutilisables

Les objectifs des DP

Objectif	Modèles
Construction	**BUILDER**, **FACTORY**, **ABSTRACT FACTORY**, MEMENTO, PROTOTYPE
Extensions	**DECORATOR**, **ITERATOR**, **VISITOR**
Interfaces	**ADAPTER**, BRIDGE, **COMPOSITE**, FACADE
Responsabilité	SINGLETON, **OBSERVER**, PROXY, MEDIATOR, FLYWEIGHT, CHAIN OF RESPONSABILITY
Opérations	TEMPLATE METHOD, STATE, **STRATEGY**, **COMMAND**, INTERPRETER

Modèles de création

Motivation :

- *simplifier* la création des objets
- *séparer* les mécanismes de création du reste
- *masquer* au client les détails de création

Principes généraux :

- renforcer l'*indépendance* entre l'utilisation des objets et leur création
- *encapsuler* l'information sur les classes concrètes utilisées
- *cacher* la façon dont les instances sont créées/assemblées
- rendre les objets *réutilisables*
- faciliter les modifications ultérieures

Builder - motivation

```
public class Employe {
    private int numeroId;
    private String nrINSEE, nom, prenom;
    private int echelon;
    private LocalDate dateNaissance;
    private double base, nbHeures;

    /* Le constructeur "principal" qui instanciera l'objet */
    public Employe(String nom, String prenom, String nrINSEE, LocalDate dateNaiss, int echelon,
           double base, double nbHeures, int numeroId) {
        this.nom = nom; this.prenom = prenom;
        this.nrINSEE = nrINSEE; this.dateNaissance = dateNaiss;
        this.echelon = echelon; this.base = base;
        this.nbHeures = nbHeures; this.numeroId = numeroId;
     }
}
```

Inconvénients de cette construction pour le client :

- constructeur trop lourd à utiliser
- des problèmes pour rendre des arguments optionnels :
  - constructeurs télescopiques $\Longrightarrow$??????
  - ajouter des modifieurs pour des attributs $\Longrightarrow$??????

Builder

Objectif : Encapsuler la construction d'un objet complexe et proposer la création par étapes

```java
public class Employe {
    private int numeroId;
    private String nrINSEE, nom, prenom;
    private int echelon;
    private LocalDate dateNaissance;
    private double base, nbHeures;

    private Employe(Builder builder) {
        numeroId = builder.numeroId;
        nrINSEE = builder.nrINSEE;
        nom = builder.nom;
        prenom = builder.prenom;
        dateNaissance = builder.dateNaissance;
        echelon = builder.echelon;
        base = builder.base;
        nbHeures = builder.nbHeures;

    }

    public static class Builder {
        /* ... le code ici à droite... */
    }
}
```

```java
Employe e= new Employe.Builder("Durand", "Jacques")
    .addNrINSEE("189599991234")
    .addDateNaissance(LocalDate.of(1997, Month.SEPTEMBER, 01))
    .addNumeroID(1214)
    .addEchelon(3)
    .build();
```

```java
public static class Builder {
    private int numeroId, echelon;
    private String nrINSEE, nom, prenom;
    private LocalDate dateNaissance;
    private double base, nbHeures;

    public Builder(String nom, String prenom){
        this.nom = nom;
        this.prenom = prenom;
    }

    public Builder addNumeroID(int n) {
        this.numeroId = n; return this;
    }

    public Builder addEchelon(int e) {
        this.echelon = e; return this;
    }

    public Builder addNrINSEE(String n) {
        this.nrINSEE = n; return this;
    }

    public Builder addDateNaissance(LocalDate d) {
        this.dateNaissance = d; return this;
    }

    public Employe build(){
        return new Employe(this);
    }
}
```


Méthode Fabrique (Factory method)

Motivation

Dans un framework générique :

- les abstractions définissent les objets et les liens entre eux
- il y a une multitude d'implémentations de ces abstractions
- le framework n'a pas à gérer les modalités de création

Processus

- encapsuler l'information sur le type concret d'objet à créer
- externaliser cette information $\Longrightarrow$ le framework est libre !

Méthode Fabrique – motivation

```java
public abstract class Voiture {
    private Color couleur;
    private double prixInitial = 10000;

    public Voiture(Color c){
        couleur = c;
    }

    public abstract void personnaliser();

    public void fixerPrix(double variableMagouille) {
        // un algorithme très complexe, par ex. :
        prixInitial *= variableMagouille;
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        Voiture v = new Limousine(Color.BLACK);
        v.fixerPrix(2.567); // même prix
        v.personnaliser();
        Voiture v1 = new TroisPortes(Color.RED);
        v1.fixerPrix(2.567); // même prix
        v1.personnaliser();

        /* du code utilisant toutes ces voitures */
    }
}
```

```java
public class TroisPortes extends Voiture {
    public TroisPortes(Color c) {
        super(c);
    }

    public void personnaliser(){
        /*le code pour personnaliser 3 portes*/
    }
}
```

```java
public class Limousine extends Voiture {
    public Limousine(Color c) {
        super(c);
    }

    public void personnaliser(){
        /*le code pour personnaliser limousine*/
    }
}
```

```java
public class QuatreQuatre extends Voiture {
    public QuatreQuatre(Color c) {
        super(c);
    }

    public void personnaliser(){
        /*le code pour personnaliser 4x4*/
    }
}
```

Des inconvénients pour le client ?

Vers la Méthode Fabrique : fabrique simple

```java
public class Client {
    public static void main(String[] args) {
        MagasinDeVoiture concessionnaire = new MagasinVoiture();
        Voiture v = concessionnaire.commanderVoiture("Limousine", Color.BLACK);
        Voiture v1 = concessionnaire.commanderVoiture("QuatreQuatre", Color.RED);
        Voiture v2 = concessionnaire.commanderVoiture("3portes", Color.WHITE);
        /* du code utilisant toutes ces voitures */
    }
}
```

```java
public class MagasinDeVoitures {
    public Voiture commanderVoiture(String type, Color couleur) {
        Voiture v;
        if (type.equals("Limousine"))
            v = new Limousine(couleur);
        else if (type.equals("QuatreQuatre"))
                v = new QuatreQuatre(couleur);
        else v = new TroisPortes(couleur);

        v.personnaliser();
        v.fixerPrix(2.567); // on s'assure qu'on fixe le même prix
        return v;
    }
}
```

Vers la Méthode Fabrique : fabrique simple

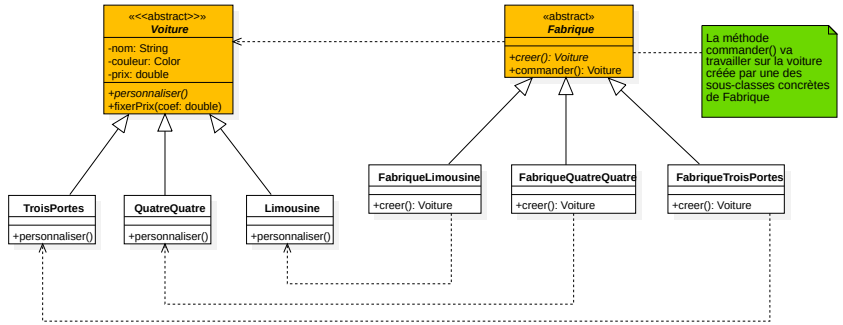
```
public class MagasinDeVoitures {
    public Voiture commanderVoiture(String type, Color couleur) {
        Voiture v = FabriqueSimple.produireVoiture(type, couleur);
        v.personnaliser();
        v.fixerPrix()
        return v;
    }
}
```

```
public class FabriqueSimple {
    public static Voiture produireVoiture(String type, Color couleur) {
        if (type.equals("Limousine"))
            return new Limousine(couleur);
        else if (type.equals("QuatreQuatre"))
                return new QuatreQuatre(couleur);
        return new TroisPortes(couleur);
    }
}
```

Le code de création est encapsulé dans la méthode de création

Est-ce qu'il y a autre chose ?

Méthode Fabrique

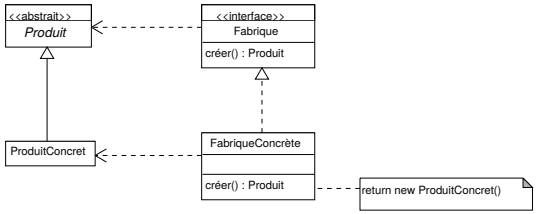


- `Fabrique` correspond au magasin de voitures générique
- Chaque fabrique concrète est une spécialisation du magasin
- Seule la fabrication est déléguée aux fabriques concrètes !

On peut ajouter d'autres spécialisations : par ex. en fonction de la marque et du type

Méthode Fabrique

Objectif : définir une classe abstraite (ou interface) de création d'objet et laisser aux sous-classes la tâche d'instanciation

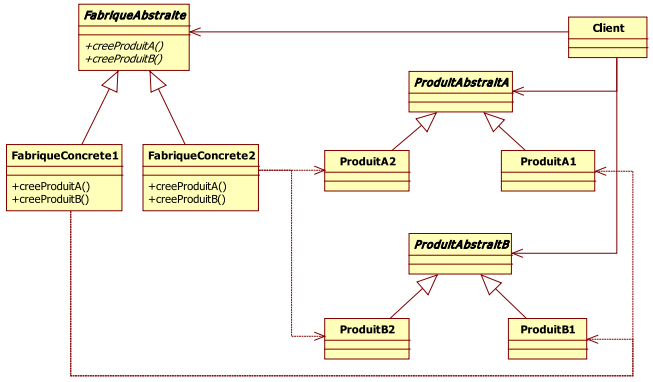


Conséquences :

- flexibilité pour la construction d'objets
- la classe est dissociée de l'instanciation de ses objets
  - hiérarchie des classes porteuses de données
  - hiérarchie des "fabriques" instanciant les objets de ces classes
- Est assez lourd à implémenter

Fabrique Abstraite

Objectif : créer une famille d'objets qui dépendent les uns des autres, sans que l'utilisateur de cette famille d'objets ne connaisse la classe exacte de chaque objet.



Le client ne voit que les classes abstraites `ProduitAbstraitA` et `ProduitAbstraitB`

Fabrique Abstraite - exemple

On souhaite un assemblage flexible d'ordinateurs :

```java
public interface DisqueDur {
   void enregistrerDonnees();
}
```

```java
public class PetitHDD implements DisqueDur {
   public void enregistrerDonnees(){
      System.out.println("J'enregistre peu de données");
   }
}
```

```java
public class GrosHDD implements DisqueDur {
   public void enregistrerDonnees(){
      System.out.println("J'enregistre beaucoup de données");
   }
}
```

```java
public interface Ecran {
   void afficherContenu();
}
```

```java
public class EcranFaibleResolution implements Ecran {
   public void afficherContenu(){
      System.out.println("J'affiche une petite résolution");
   }
}
```

```java
public class EcranGrandeResolution implements Ecran {
   public void afficherContenu(){
      System.out.println("J'affiche une belle résolution");
   }
}
```

```java
public interface FabriqueOrdinateur {
   public Processeur creerProc();
   public DisqueDur creerDiqueDur();
   public Ecran creerEcran();
}
```

```java
public class FabriqueOrdiCher implements FabriqueOrdinateur {
   public Processeur creerProc() { return new ProcesseurRapide(); }
   public DisqueDur creerDiqueDur() { return new GrosHDD(); }
   public Ecran creerEcran() { return new EcranGrandeResolution(); }
}
```

```java
public class FabriqueOrdiLowCost implements FabriqueOrdinateur {
   public Processeur creerProc() { return new ProcesseurLent(); }
   public Ecran creerEcran() { return new EcranFaibleResolution(); }
   public DisqueDur creerDiqueDur() { return new PetitHDD(); }
}
```

```java
public interface Processeur {
   void executerOperation();
}
```

```java
public class ProcesseurLent implements Processeur {
   public void executerOperation() {
      System.out.println("Je m'exécute lentement");
   }
}
```

```java
public class ProcesseurRapide implements Processeur {
   public void executerOperation(){
      System.out.println("Je m'exécute super-vite");
   }
}
```


Fabrique Abstraite - exemple

```java
public class Ordinateur {

   private Processeur processeur;
   private HDD hdd;
   private Ecran ecran;

   public Ordinateur(Processeur p, HDD hdd, Ecran e){
      processeur = p;
      this.hdd = hdd;
      ecran = e;
   }

   public void mettreEnMarche(){
      processeur.executerOperation();
      hdd.enregistrerDonnees();
      ecran.afficherContenu();
   }
}
```

```java
public class MagasinOrdinateurs {

   FabriqueOrdinateur usine;

   public MagasinOrdinateurs(FabriqueOrdinateur usine){
      this.usine = usine;
   }

   public Ordinateur assemblerOrdi(){
      Processeur processeur = usine.creerProc();
      DisqueDur hdd = usine.creerDiqueDur();
      Ecran ecran = usine.creerEcran();
      //on utilise les 3 pour assembler l'ordi
      return new Ordinateur(processeur, hdd, ecran);
   }

}
```

```java
public class App {

   public static void main (String args[]){
      //FabriqueOrdinateur factory = new FabriqueOrdiCher();
      FabriqueOrdinateur factory = new FabriqueOrdiLowCost();
      MagasinOrdinateurs magasin = new MagasinOrdinateurs(factory);
      Ordinateur maMachine = magasin.assemblerOrdi();
      maMachine.mettreEnMarche();
   }

}
```

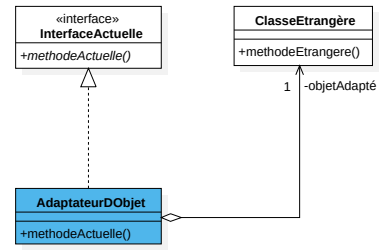

Modèles de structure

Principes généraux

- identifier une façon simple de réaliser les relations
- compositions de classe – décrire les relations à travers l'héritage
- compositions d'objets – obtenir des nouvelles fonctionnalités
- obtenir des nouvelles structures pour "*traiter*" de manière uniforme des objets uniques ou des regroupements d'objets

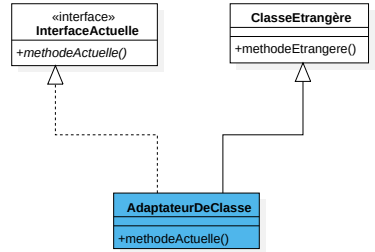
Adaptateur

Intention : collaboration des classes de types incompatibles




```java
public void methodeActuelle() {

    // utilisation par composition
    objetAdapté.methodeEtrangere();

}
```

blackbox reuse

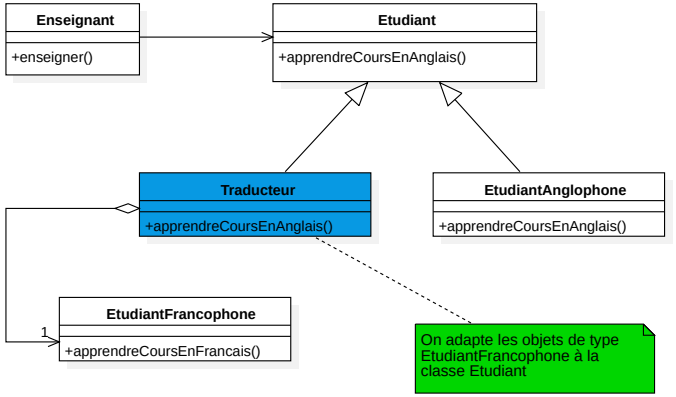



```java
public void methodeActuelle() {

    // utilisation par héritage
    methodeEtrangere();

}
```

whitebox reuse

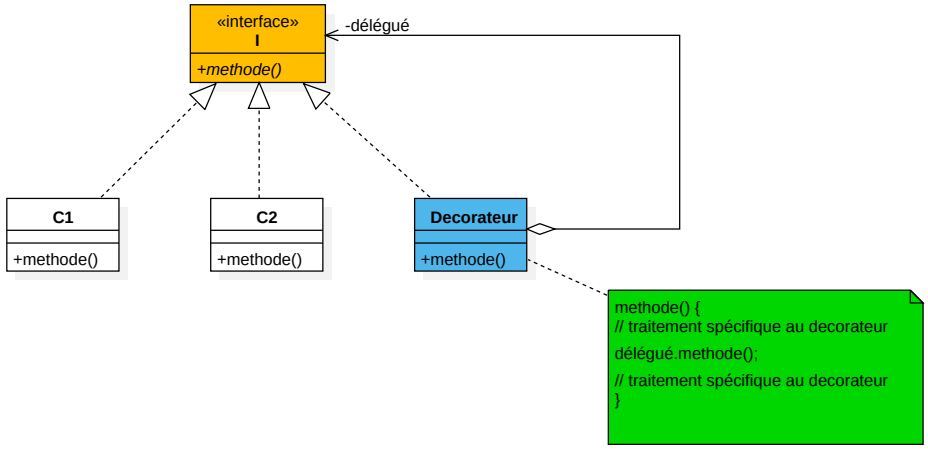
Adaptateur - exemple

- Un professeur avec des étudiants parlant des langues différentes
- Il faut traduire les cours en anglais
  - La **cible** : la classe `Etudiant`
  - L'**adaptateur** : la classe `Traducteur`
  - L'**adapté** : la classe `EtudiantFrancophone`



Décorateur

Objectif : alternative plus souple à l'héritage, pour étendre les fonctionnalités (parfois même plus efficace)



Décorateur - exemple

```java
public interface Voiture {
   public void demarrer();
   public void arreter();
   public void allumerPhares();
}
```

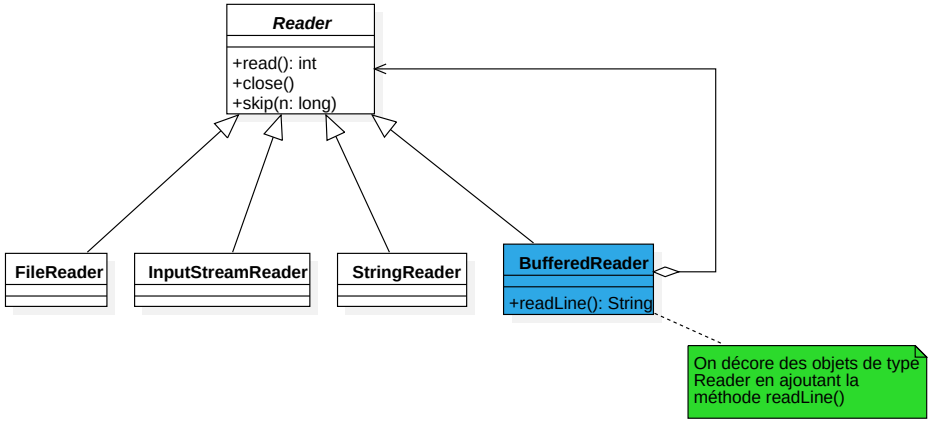
```java
public class VoitureElectrique implements Voiture {
   Voiture voitureADecorer;

   public VoitureElectrique(Voiture v) {
      voitureADecorer = v;
   }

   // nouvelle méthode -- décoration
   public void chargerVoiture() {
      System.out.println("Je récupère qqs km d'autonomie");
   }

   public void demarrer() {
      voitureADecorer.demarrer(); // réutilisation
   }

   public void arreter() {
      voitureADecorer.arreter(); // réutilisation
   }

   public void allumerPhares() {
      voitureADecorer.allumerPhares(); // réutilisation
   }

}
```

```java
public class Limousine implements Voiture {
   public void demarrer() {
      // du code ici
   }

   public void arreter() {
      // du code ici
   }

   public void allumerPhares() {
      // du code ici
   }
}
```

```java
public class TroisPortes implements Voiture {
   public void demarrer() {
      // du code ici
   }

   public void arreter() {
      // du code ici
   }

   public void allumerPhares() {
      // du code ici
   }
}
```


Décorateur - exemple en Java

La classe `java.io.BufferedReader` :

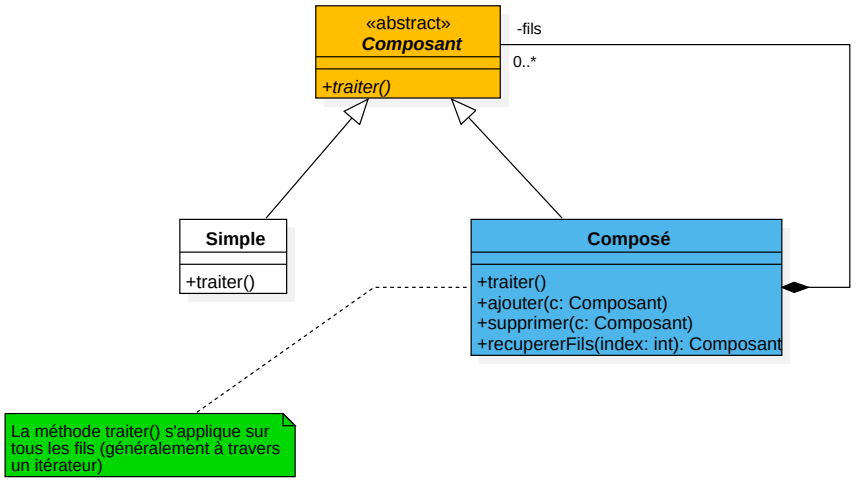


Utilisation :

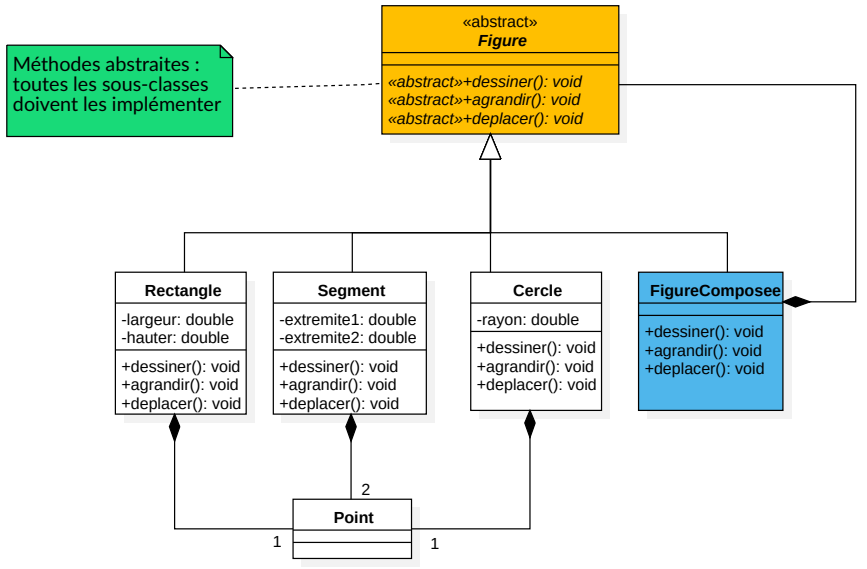
```java
BufferedReader b = new BufferedReader(new FileReader(new File("Fic.txt")));
b.readline();
```

Composite

Objectif : créer des objets simples ou composés avec des méthodes de traitement uniformes



Composite - exemple



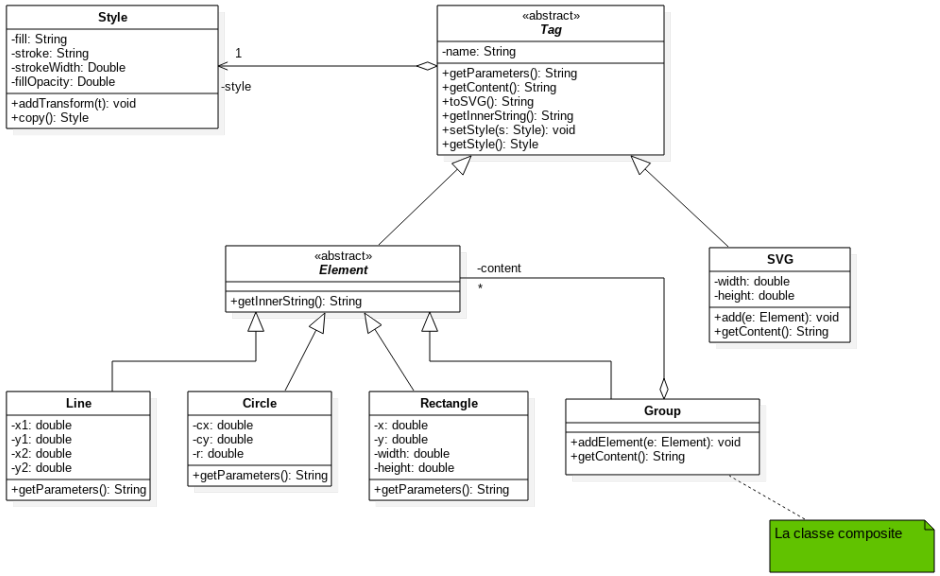
Composite - exemple

```java
public interface FichierGeneral {
   public void afficher();
}
```

```java
public class Fichier implements FichierGeneral {
   private String nom;
   public Fichier(String n){
      nom = n;
   }
   public void afficher() {
      System.out.println(nom);
   }
}
```

```java
import java.util.ArrayList;
public class Repertoire implements FichierGeneral {
   private String nom;
   private ArrayList<FichierGeneral> contenu = new ArrayList<FichierGeneral>();

   public Repertoire(String n){
      nom = n;
   }

   public void ajouter(FichierGeneral f){
      contenu.add(f);
   }

   public void afficher() {
      System.out.println("Le dossier "+nom+" contient :");
      for (FichierGeneral f : contenu){
         f.afficher();
      }
   }
}
```


Un exemple de l'an dernier : SVG



Décorateur vs Composite

La structure est similaire mais ...

- On utilise le modèle **Composite** quand on veut maintenir un groupe d'objets avec un comportement similaire à l'intérieur d'un autre objet
- Le **Décorateur** est utilisé quand on souhaite modifier les fonctionnalités de l'objet à l'exécution

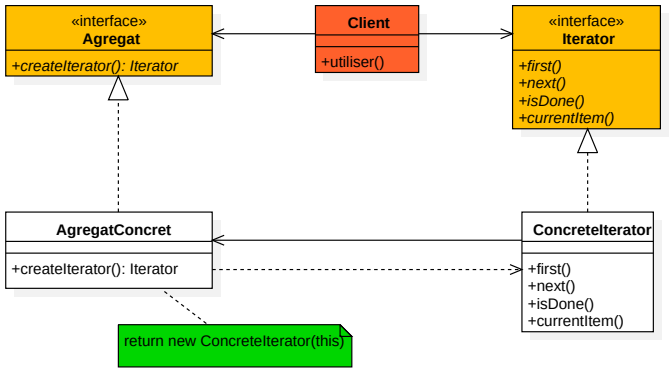
Modèles Comportementaux

Principes généraux

- améliorer la communication entre les objets :
  - définir les responsabilités de chaque objet
  - définir un modèle de communication entre les objets
- rendre les liens entre les objets communicants plus *souples*
- ... et donc augmenter la flexibilité de l'architecture

Itérateur

Objectif : fournir un moyen d'accès séquentiel aux éléments d'une agrégation d'objets, sans exposer la représentation interne de celle-ci



La classe cliente n'est pas concerné par la manière dont les objets sont gérés dans la collection.

Remarque : en Java l'interface *`Iterator`* est légèrement différente

Itérateur - exemple générique

```java
public class Entreprise {
    private String nom;
    private List<Personne> employes;
    public Entreprise(String nom) {
        this.nom = nom; employes = new LinkedList<Personne>();
    }

    public void embaucher(Personne p){ employes.add(p); }

    public Iterateur creerIterateur(){
        return new IterateurDePersonnes(employes);
    }
}
```

```java
public class Personne {
    private String nom, prenom;
    public Personne(String nom, String prenom) {
        this.nom = nom; this.prenom = prenom;
    }

    public String toString() {
        return "Personne [nom=" + nom + "," + "prenom=" + prenom + "]";
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        Entreprise maBoite = new Entreprise("Poire Mordue");
        maBoite.embaucher(new Personne("Laporte", "Marc"));;
        maBoite.embaucher(new Personne("Pain-Barre", "Cyril"));
        maBoite.embaucher(new Personne("Valicov", "Petru"));

        Iterateur it = maBoite.creerIterateur();
        for (it.premier(); !it.cestFini(); it.suivant()) {
            System.out.println(it.courant());
        }
    }
}
```

```java
public interface Iterateur {
    public Personne premier();
    public void suivant();
    public boolean cestFini();
    public Personne courant();
}
```

```java
public class IterateurDePersonnes implements Iterateur {
    private List<Personne> lesEmployes;
    private int position;

    public IterateurDePersonnes(List<Personne> employes) {
        lesEmployes = employes;
    }

    public void suivant() {
       position++;
    }

    public Personne premier() {
        position = 0;
        return lesEmployes.get(0);
    }

    public boolean cestFini() {
        if (position >= lesEmployes.size())
            return true;
        return false;
    }

    public Personne courant() {
        return lesEmployes.get(position);
    }
}
```

Itérateur - exemple avec java.util.Iterator

```java
public class Entreprise {
    private String nom;
    private List<Personne> employes;
    public Entreprise(String nom) {
        this.nom = nom; employes = new LinkedList<Personne>();
    }

    public void embaucher(Personne p){ employes.add(p); }

    public Iterator<Personne> createIterator(){
        return new IterateurConcret(employes);
    }
}
```

```java
public class Personne {
    private String nom, prenom;
    public Personne(String nom, String prenom) {
        this.nom = nom; this.prenom = prenom;
    }

    public String toString() {
        return "Personne [nom=" + nom + "," + "prenom=" + prenom + "]";
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        Entreprise maBoite = new Entreprise("Poire Mordue");
        maBoite.embaucher(new Personne("Laporte", "Marc"));
        maBoite.embaucher(new Personne("Pain-Barre", "Cyril"));
        maBoite.embaucher(new Personne("Valicov", "Petru"));

        for (Iterator<Personne> it =
                maBoite.createIterator();it.hasNext();){
            System.out.println(it.next());
        }
    }
}
```

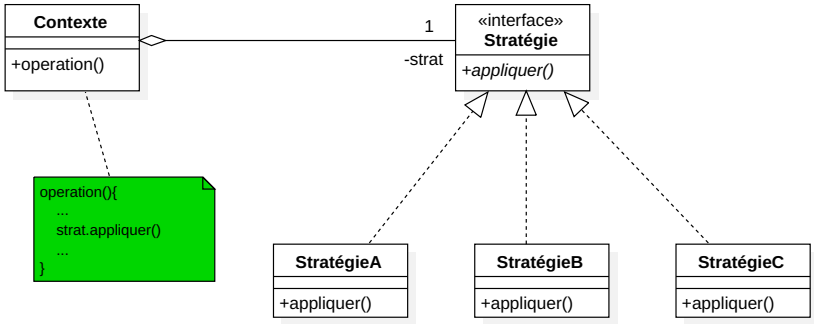
```java
// interface définie dans java.util
public interface Iterator<T> {

    // test de fin
    public boolean hasNext();

    //retourner courant + passer au suivant
    public T next();

    // optionnelle (a une implémentation par défaut)
    pubilc void remove();
}
```

```java
public class IterateurConcret implements Iterator<Personne>{

    private int position = 0;
    private List<Personne> lesEmployes;

    public IterateurConcret(List<Personne> employes) {
        lesEmployes = employes;
    }

    public boolean hasNext() {
        if (position >= lesEmployes.size())
            return false;
        return true;
    }

    public Personne next() {
        Personne p = lesEmployes.get(position);
        position++;
        return p;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```


Stratégie

Objectif :

- définir une famille d'algorithmes encapsulés dans des objets
- rendre ces algorithmes interchangeables dynamiquement
- masquer au client les données utilisées par les algos



Stratégie - exemple facile

```java
public interface Tri {
    public void trier(int[] tab);
}
```

```java
public class TriABulles implements Tri {
    public void trier(int[] tab) {
        // le code de l'algo
    }
}
```

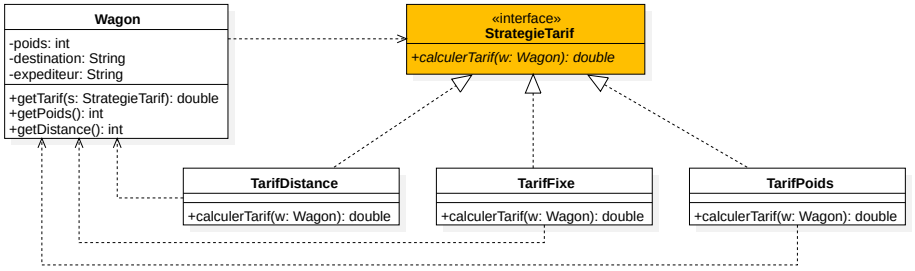
```java
public class Contexte {
    private Tri t;

    public void effectuerTri(int[] tab) {
        t.trier(tab);
    }

    public void changerTri(Tri typeTri) {
        t = typeTri;
    }

    public Tri retournerTri() {
        return t;
    }

}
```

```java
public class TriMinimum implements Tri {
    public void trier(int[] tab) {
        // le code de l'algo
    }
}
```

```java
public class TriParTas implements Tri {
    public void trier(int[] tab) {
        // le code de l'algo
    }
}
```

```java
public class ClasseCliente {

    public static void main(String[] args) {
        int[] liste = { 1, 2, 4, 7, 9, 3, 2, 1, 0, 22 };

        Contexte contexteCourant = new Contexte();

        contexteCourant.changerTri(new TriABulles());
        contexteCourant.effectuerTri(liste);

        for (int i = 0; i < liste.length; i++) {
            System.out.println(liste[i]);
        }
    }
}
```


Exemple que vous connaissez




```java
public class Wagon { // classe de Contexte
   // ...
    public void getTarif(StrategieTarif s) {
      // ...
      s.calculerTarif(this);
      // ...
    }
}
```

- l'interface `StrategieTarif` est une **stratégie abstraite**
- les méthodes `calculerTarif(...)` sont les **algos interchangeables**
- la classe `Wagon` est le **contexte**

Stratégie - conclusion

Avantages

- on peut changer les stratégies *à la volée*
- permet aux algorithmes d'évoluer *indépendamment* des clients qui les utilisent
- la classe `Contexte` peut avoir des sous-classes indépendantes des stratégies
- ce modèle est une alternative à l'héritage

Inconvénients (mineurs)

- surcoût en place mémoire (il faut créer les objets `Stratégie`)
- peut potentiellement casser l'encapsulation des objets sur lesquels les stratégies opèrent

Visiteur - exemple

```
public abstract class Article{
   private double prix;
   private int numero;

   public double getPrix() {
      return prix;
   }

   public int getNumero() {
      return numero;
   }

   public abstract double getCoutLivraison();
}
```

```
public class Livre extends Article{
  public double getCoutLivraison(){
     return super.getPrix()*0.05;
  }
}
```

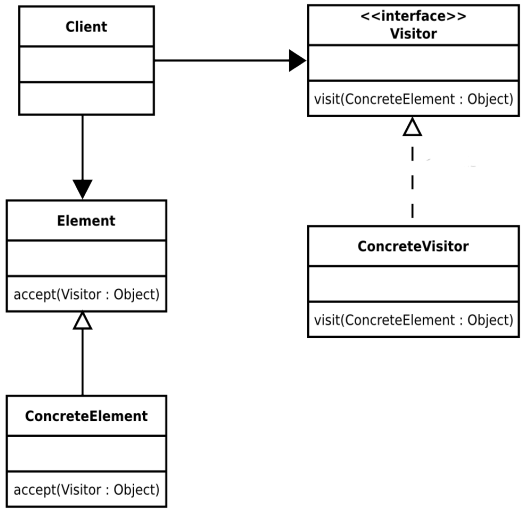
```
public class Moto extends Article{
  public double getCoutLivraison(){
     return super.getPrix()*0.09;
  }
}
```

```
public class Panier{
  private List<Article> contenu = new ArrayList<Article>();

  public void ajouter(Article a, int quantite){
     for (int i=0; i<quantite; i++){
        contenu.add(a);
     }
  }

  public double calculerTotalCoutLivraison(){
     double total = 0;
     for (Article a: contenu){
        total += a.getCoutLivraison();
     }
     return total;
  }
}
```

Et si le coût de la livraison varie en fonction de la région ?

Visiteur

Objectif : définir une nouvelle opération sans modifier les classes sur lesquelles elle opère.



Visiteur - exemple

```
public abstract class Article{
   private double prix;
   private int numero;

   public double getPrix() {
      return prix;
   }
}
```

```
public interface Visitable{
   public void accepter(Visiteur visiteur);
}
```

```
public interface Visiteur{
   public void visiter(Livre livre);
   public void visiter(Moto moto);
}
```

```
public class Panier{
  List<Visitable> contenu = new
        ArrayList<Visitable>();

  public double calculerTotalCoutLivraison(){
     VisiteurPostalPACA visiteur = new
           VisiteurPostalPACA();
     for(Visitable article: contenu) {
        article.accepter(visiteur);
     }
     return visiteur.getCoutTotalLivraison();
  }
}
```

```
public class Livre extends Article implements Visitable{
   public void accepter(Visiteur visiteur) {
      visiteur.visiter(this);
  }
}
```

```
public class Moto extends Article implements Visitable{
   public void accepter(Visiteur visiteur) {
      visiteur.visiter(this);
  }
}
```

```
public class VisiteurPostalPACA implements Visiteur {
   private double coutTotalLivraison;

   public void visiter(Livre livre) {
      //les livres chers sont livrés gratuitement
      // dans la region PACA
      if(livre.getPrix() < 20) {
         coutTotalLivraison += livre.getPrix()*0.05;
      }
   }

   public void visiter(Moto moto){...}

   //retourner l'etat interne
   public double getCoutTotalLivraison() {
      return coutTotalLivraison;
   }
}
```

Visiteur - conclusion

Avantages

- Permet l'ajout facile de nouvelles opérations
- Préserve l'intégrité des classes
- Les visiteurs peuvent changer d'état lors de la visite des éléments

Inconvénients

- "*Probablement le modèle le plus compliqué*"
- Ajouter des nouveaux éléments concrets est plus difficile
- Peut casser l'encapsulation (de l'élément visitable)

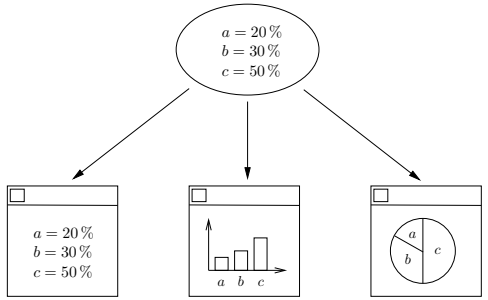
Conseil

Le Visiteur est très puissant, mais il faut l'utiliser avec discernement.

Observateur

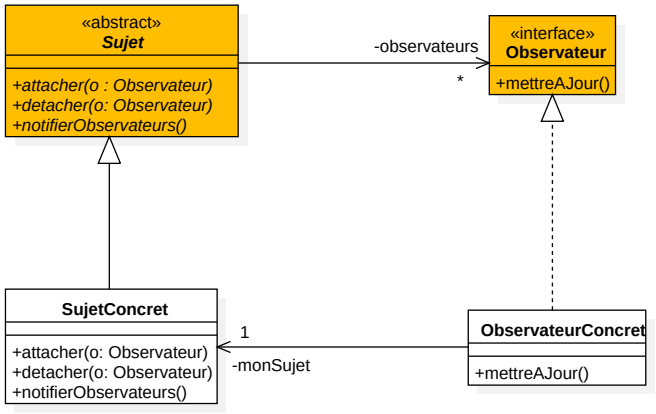
Objectif : lorsqu'un objet change, tous les objets liés sont notifiés et mis à jour

Exemple : une application peut comporter plusieurs représentations graphiques pour un même objet



Comment notifier les représentations graphiques des changements ?

Observateur - schéma général



- `Sujet` utilise les types `Observateur` pour les informer
- `ObservateurConcret` utilise des données fournies par `SujetConcret` pour effectuer la mise à jour

Observateur - exemple

```
public interface Observateur {
    public void update(double taux, String banque, String type);
}
```

```
public interface Sujet {
    public void enregistrerObservateur(Observateur obs);

    public void supprimerObservateur(Observateur obs);

    public void notifierObservateurs();
}
```

```
public class Journal implements Observateur {

    public void update(double taux, String banque, String type) {
        System.out.println("Journal: le taux " + type+ " a été mis à jour");
        System.out.println(" -- le nouvel taux : " + taux);
        System.out.println(" -- c'est la banque : " + banque);
    }
}
```

```
public class Internet implements Observateur {

    public void update(double taux, String banque, String type) {
        System.out.println("Internet: le taux a été mis à jour : ");
        System.out.println(" -- le nouvel taux : " + taux);
        System.out.println(" -- c'est chez la banque : " + banque);
    }
}
```

Observateur - exemple

```java
public class PretBancaire implements Sujet {

    private List<Observateur> observateurs = new LinkedList<Observateur>();
    private String type;
    private double taux;
    private String banque;

    public PretBancaire(String type, double taux, String banque) {
        this.type = type; this.taux = taux; this.banque = banque;
    }

    public void setInterets(double d) {
        this.taux = d;
        notifierObservateurs(); // la modification des taux doit être diffusée
    }

    public void enregistrerObservateur(Observateur obs) {
        observateurs.add(obs);
    }

    public void supprimerObservateur(Observateur obs) {
        observateurs.remove(obs);
    }

    public void notifierObservateurs() {
        for (Observateur ob : observateurs) {
            System.out.println("Notification des observateurs sur le changement du taux d'intérêt");
            ob.update(this.taux, this.banque, this.type);
        }
    }
}
```


Observateur - exemple

```java
public class ClasseDeTest {

    public static void main(String[] args) {
        Observateur journal = new Journal();
        Observateur leWeb = new Internet();

        PretBancaire sujetConcret = new PretBancaire("Immobilier", 1.48, "BNP ParisHaut");

        sujetConcret.enregistrerObservateur(journal);
        sujetConcret.notifierObservateurs();

        sujetConcret.enregistrerObservateur(leWeb);

        sujetConcret.setInterets(1.35); // ==> implique la notifiaction des observateurs

    }

}
```

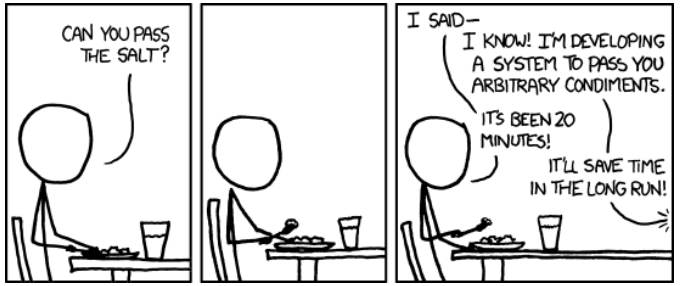

Observateur - Modèle-Vue-Contrôleur (MVC)

Le MVC est un exemple classique d'implémentation de Observateur :

- Le **modèle** : conserve toutes les données relatives à l'application (sous quelque forme que ce soit : base de données, fichiers...) et contient la logique métier de l'application.
- La **vue** : a pour rôle d'offrir une présentation du modèle (IHM par exemple). On peut avoir de nombreuses vues pour un même modèle.
- Le **contrôleur** : répond aux actions de l'utilisateur. Il traduit les événements de la vue en modifications du modèle et définit également la manière dont la vue doit réagir face aux interactions de l'utilisateur.

Les DP - Conclusion

- Les bons concepteurs/programmeurs **réutilisent** les solutions existantes
- N'utilisez pas les modèles de conception si vous ne les comprenez pas !
  - parfois la vie est plus simple sans patterns
  - utilisez les DP que si vous développez à long terme



Gare aux patrons abîmés et aux anti-patrons (`antipatterns`) !

https://en.wikipedia.org/wiki/Anti-pattern