Conception et programmation objet avancées

Modèles de conception

Petru Valicov
petru.valicov@univ-amu.fr

2017-2018

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

$\Longrightarrow$ Les patrons de conception sont nés de la capitalisation de cette expérience

Patrons de Conception

- on utilise souvent le terme anglais *Design Patterns*
- description des problèmes récurrents
- des solutions suffisamment générales et bien connues
- des solutions flexibles
- des solutions indépendantes des langages de programmation
- des solutions extensibles

Le "Gang of Four" (GoF) a formalisé 23 patrons :

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 mais ne le saviez pas

Patron de Conception - définition

- Abstrait, de haut niveau
- Reposant sur des interfaces / classes abstraites
- Orienté vers le bon découpage en package (modularité, flexibilité, réutilisabilité)
- Exprimé sous forme d'architecture reliant quelques classes très abstraites

Définition

Un patron de conception décrit une structure commune et répétitive de composants en interaction (la solution) qui résout un problème de conception dans un contexte particulier.

Patrons de Conception - définition

Un DP comporte différents éléments :

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

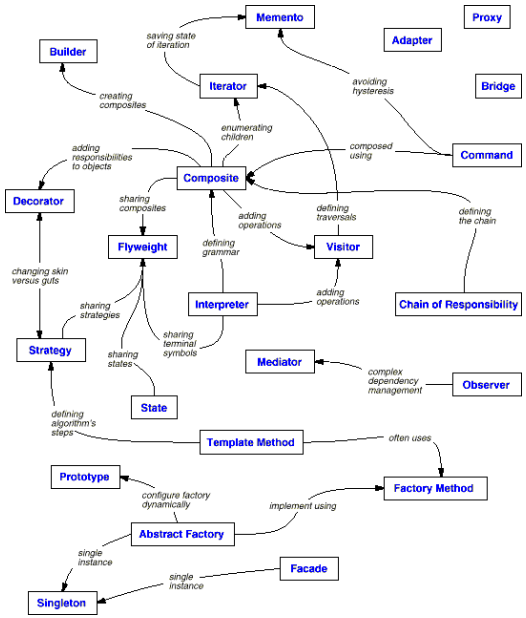
Patrons de Conception - utilité

Un bon patron 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 solution universelle prête à l'emploi
- pas de bibliothèque de classes réutilisables
- il faut quand-même réfléchir
- en gros : pas de magie



Les objectifs des DP

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

Patrons de création

Motivation :

- coder en dur l'instanciation des objets peut s'avérer dangereux
- comment définir des objets (briques de base) pour construire de nouveaux objets ?
- séparer les mécanismes de création du reste

Principes généraux :

- renforcer l'*indépendance* entre le système et création des objets
- *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

```java
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
- la création est complexe
- que faire si certains arguments sont optionnels ?

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
- souvent il peut être nécessaire de créer des instances des objets manipulés...
- ... et ça serait bien que le framework n'ait pas à savoir comment les créer

Solution

- encapsuler l'information sur le type d'objet à créer
- externaliser cette information
- le type n'est pas connu par le concepteur du Framework

Vers la Méthode Fabrique

On veut fabriquer des voitures de différents types et les utiliser :

```java
public abstract class Voiture {
    private String nom;
    private Color couleur;
    private double prix;

    public Voiture(String n, Color c){
        nom = n; couleur = c;
    }

    public abstract void personnaliser();

    public void fixerPrix(){
        /* un algorithme très complexe */
    }

}
```

```java
public class TroisPortes extends Voiture {

    public TroisPortes(String n, Color c) {
        super(n, c);
    }

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

}
```

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

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

}
```

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

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

}
```

Vers la Méthode Fabrique

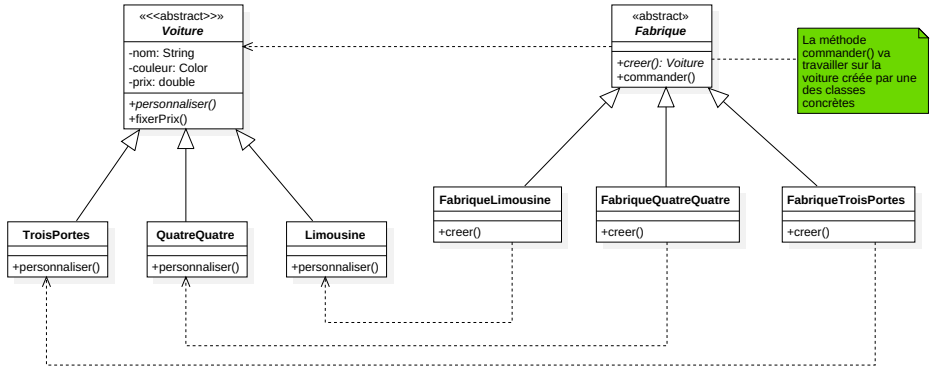
```java
public class Client {
    public static void main(String[] args) {
        Voiture v = MagasinDeVoiture.creerVoiture("Limousine", "Peugeot", Color.BLACK);
        Voiture v1 = MagasinDeVoiture.creerVoiture("QuatreQuatre", "Mercedes", Color.RED);
        Voiture v2 = MagasinDeVoiture.creerVoiture("3portes",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

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

        v.personnaliser();
        v.fixerPrix()
        return v;
    }
}
```

Qu'en pensez vous ?

Une fabrique simple est-elle suffisante pour spécialiser les magasins ?

Méthode Fabrique

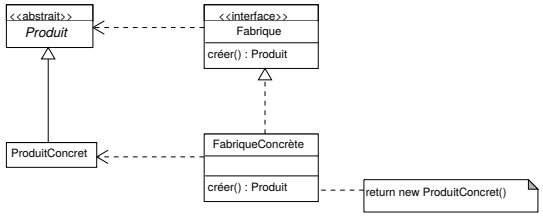


- La fabrique abstraite correspond au magasin de voitures générique
- Chaque fabrique concrète est une spécialisation du magasin
- Seule la fabrication est déléguée aux fabriques concrètes !

La classe client ?

Méthode Fabrique

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

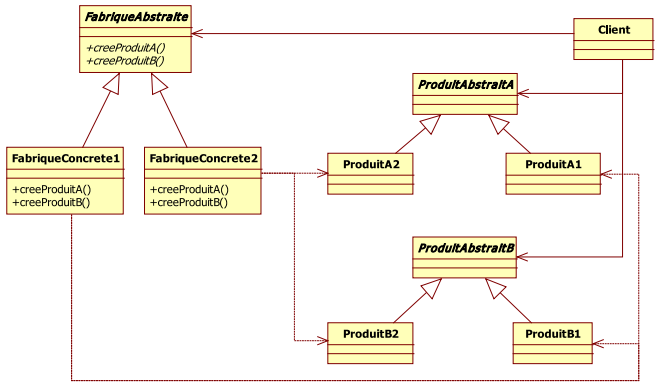


Conséquences :

- flexibilité pour la construction d'objets
- la classe est dissociée de l'instanciation de ses objets
- deux hiérarchies :
  - des classes porteuses de données
  - des "fabriques" permettant d'instancier les objets de ces classes
- Parfois peut être lourd (la fabrique en "cascade")

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 construire un ordinateur composé de Processeur, Disque Dur et Écran. Voici la fabrication :

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

```java
public class PetitHDD implements DisqueDur {
   public void enregistrerDonnees(){
      System.out.println("On peut enregistrer peu de données");
    }
}
```

```java
public class GrosHDD implements DisqueDur {
   public void enregistrerDonnees(){
       System.out.println("On peut enregistrer beaucoup de données");
    }
}
```

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

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

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

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

```java
public class EcranFaibleResolution implements Ecran {
   public void afficherContenu(){
       System.out.println("Qualité d'image faible");
    }
}
```

```java
public class EcranGrandeResolution implements Ecran {
   public void afficherContenu(){
       System.out.println("Meilleure qualité d'image");
    }
}
```

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

```java
public class ProcesseurLent implements Processeur {
   public void executerOperation() {
       System.out.println("L'exécution est lente");
    }
}
```

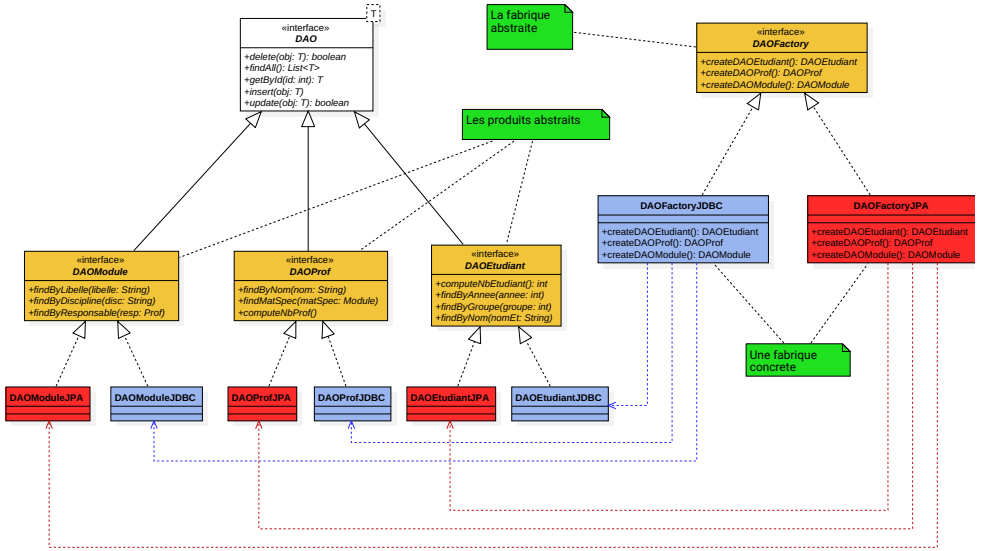
```java
public class ProcesseurRapide implements Processeur {
   public void executerOperation(){
       System.out.println("L'exécution est rapide");
    }
}
```

Fabrique Abstraite - exemple

```java
public class MagasinOrdinateurs {
   FabriqueOrdinateur categorie;

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

    public void assemblerOrdi(){
       Processeur processeur = categorie.creerProc();
       DisqueDur hdd = categorie.creerDiqueDur();
       Ecran ecran = categorie.creerEcran();

       //on utilise les 3 pour assembler l'ordi

       processeur.executerOperation();
       hdd.enregistrerDonnees();
       ecran.afficherContenu();
    }
}
```

```java
public class Client {
   public static void main (String args[]){
      //FabriqueOrdinateur factory = new OrdinateurCher();
      FabriqueOrdinateur factory = new OrdinateurPetitBudget();
      MagasinOrdinateurs magasin = new MagasinOrdinateurs(factory);
      magasin.assemblerOrdi();
   }
}
```

Fabrique Abstraite - exemple que vous connaissez



https://github.com/IUTInfoAix-M3106/TpJpa

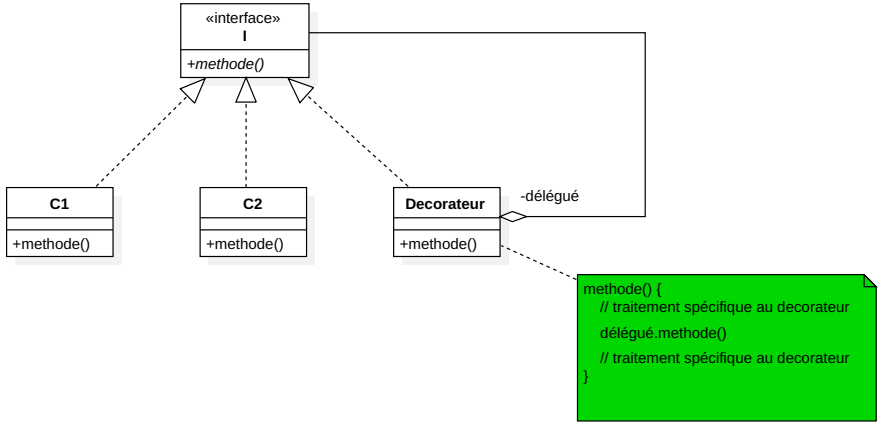
Patrons de structure

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

Décorateur

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



Décorateur - exemple

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

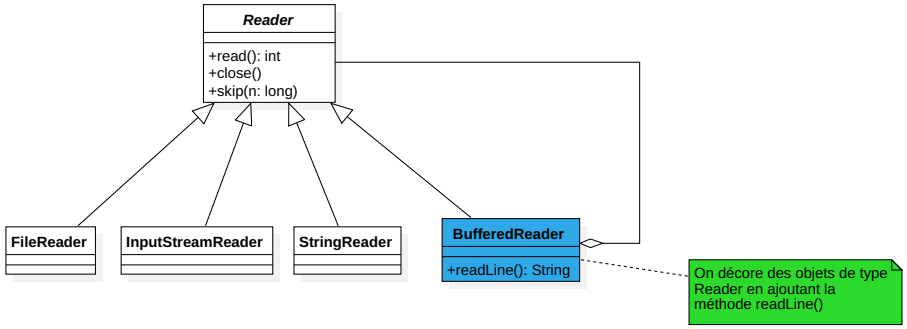
```
public class VoitureMusicale implements Voiture {
   Voiture voitureADecorer;

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

   // nouvelle méthode
   public void activerLaMusique() {
      System.out.println("Je mets la musique à fond");
   }

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

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

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

}
```

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

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

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

```
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` :

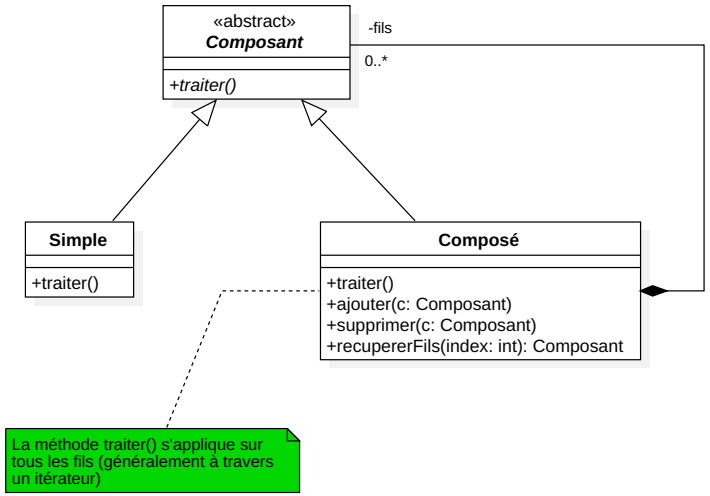


Utilisation :

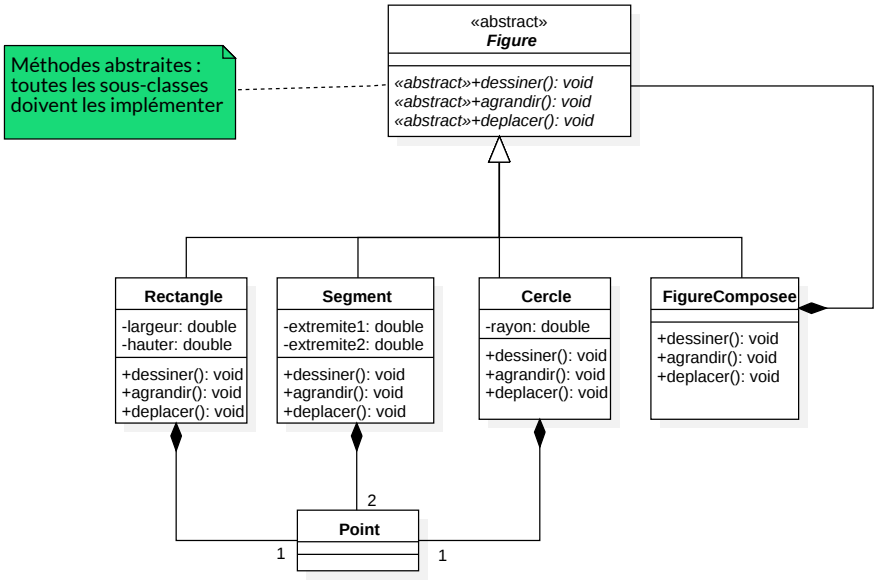
```
BufferedReader b = BufferedReader(new FileReader(new File("Fic.txt")));
b.readline();
```

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();
      }
   }
}
```

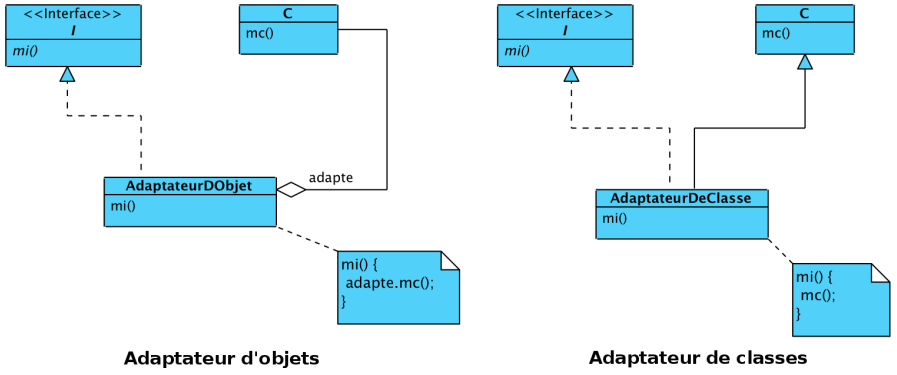
Décorateur vs Composite

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

Adaptateur

Intention : on ne refait pas, on adapte !
Permettre à des classes de collaborer, qui n'auraient pu le faire du fait d'interfaces incompatibles.

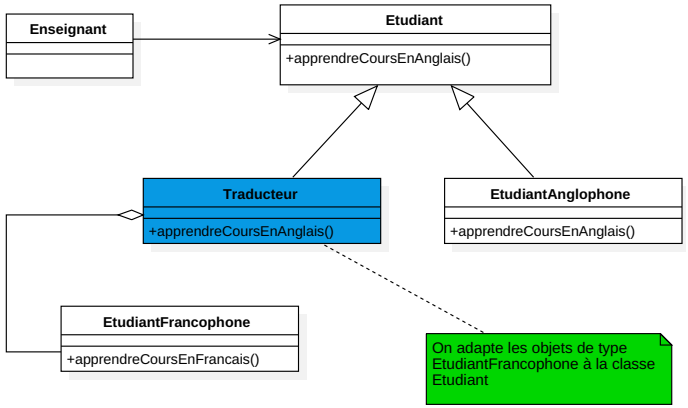


blackbox reuse

whitebox reuse

Adaptateur - exemple

- Un professeur avec deux types d'étudiants de nationalités différentes
- Il faut traduire les cours en anglais
- La cible : la classe mère `Etudiant`
- L'adaptateur : la classe `Traducteur`
- L'adapté : la classe `EtudiantFrancophone`



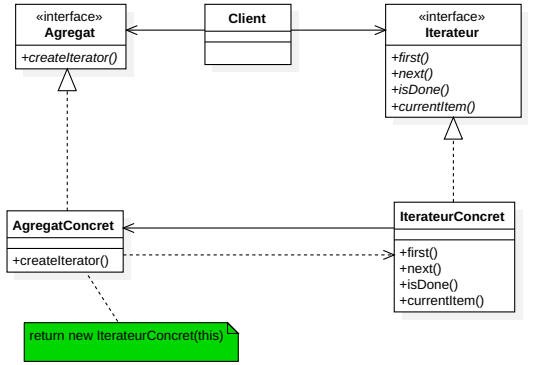
Patrons Comportementaux

Principes généraux :

- 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



Le client 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());
        }
    }
}
```

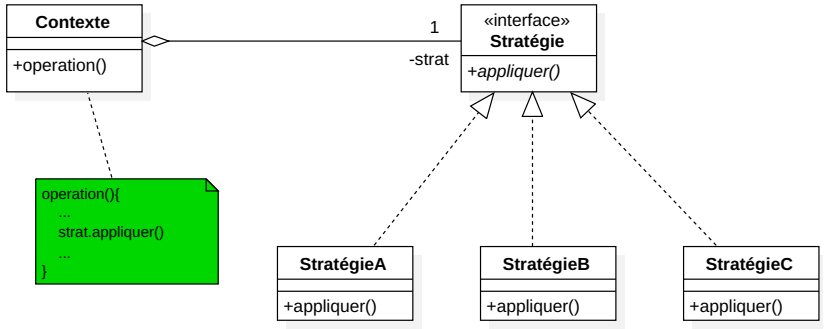
```java
// interface définie dans java.util
public interface Iterator<T> {

    // 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

```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 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 Contexte {
    private Tri t;

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

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

    public Tri retournerTri() {
        return t;
    }

}
```

La classe `Client` ?

Stratégie - exemple

```
public class ClasseCliente {

    public static void main(String[] args) {
        int[] liste = { 1, 2, 4, 7, 9, 3, 2, 1, 2, 0, 2, 10, 2, 22, 5 };

        Contexte contexteCourant = new Contexte();

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

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

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 :

- surcoût en place mémoire (il faut créer les objets `Stratégie`)
- surcoût en temps d'exécution à cause de l'indirection `strategie.appliquer()`

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;
  }
}
```

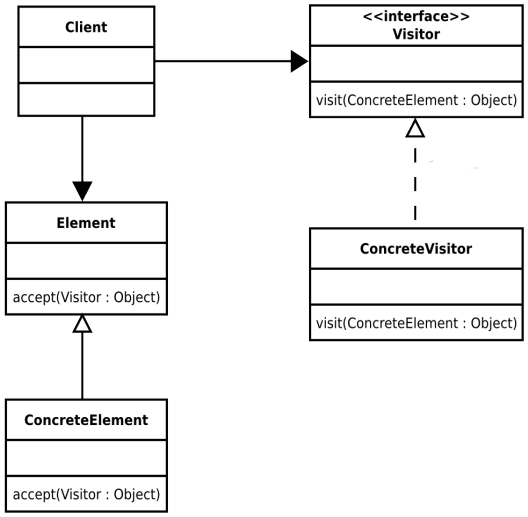
Et si le coût de la livraison varie en fonction de la région ?

Visiteur

1. Imaginez-vous un objet composite contenant des objets de **types très hétérogènes**
2. Vous devez ajouter des **fonctions très différentes** par nature mais qui s'appliquent sur tous les objets du composite
3. Vous devez fournir un moyen simple et centralisé d'ajouter ces fonctions

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 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 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 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 :

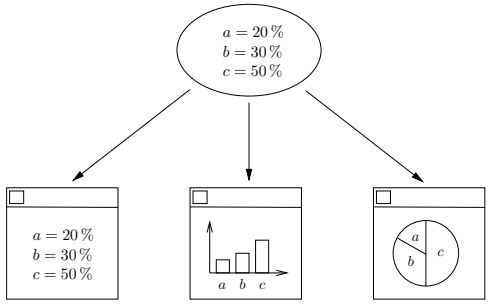
- "*Probablement le modèle le plus compliqué*"
- Ajouter des nouveaux éléments concrets est plus difficile
- Peut casser l'encapsulation (de l'élément visitable)

Le Visiteur est très puissant, mais il faut l'utiliser seulement là où il est nécessaire.

Observateur

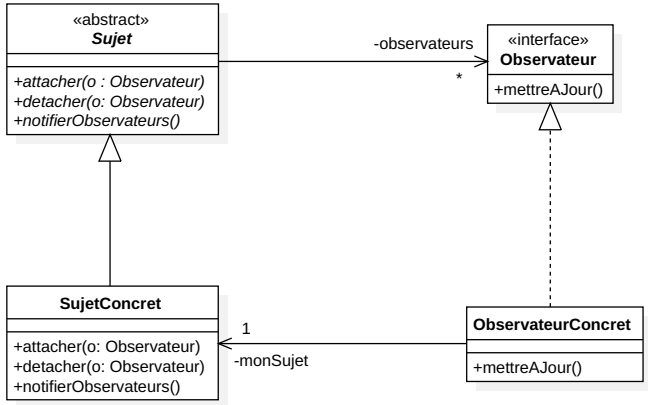
Objectif : lorsqu'un objet change, tous les objets liés sont notifiés et mis à jour

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 (mais `Observateur` n'a pas besoin de pouvoir accéder à `Sujet`)
- `ObservateurConcret` utilise des données fournies par `SujetConcret` pour effectuer la mise à jour

Observateur - méthodes importantes

- `ajouterObserveur(Observer o)`
- `deleteObserver(Observer o)`
- `hasChanged()`
- `clearChanged()`
- `notifyObservers()`

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

```
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 setInteret(double d) {
        this.taux = d;
        notifierObservateurs();
    }

    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

```
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.setInteret(1.35); // ==> implique la notifiaction des observateurs

    }

}
```

Observateur - Modèle-Vue-Contrôleur (MVC)

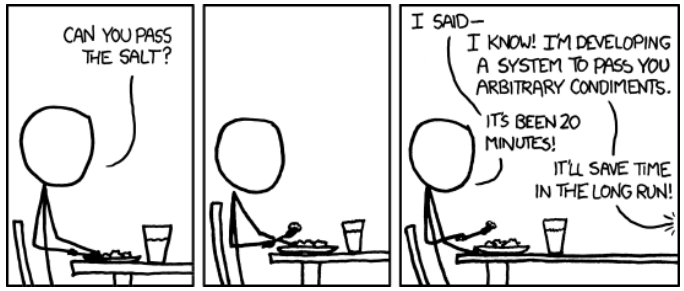
Le MVC est un exemple classique d'implémentation de Observateur :

- 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

- Il s'agit tout d'abord d'un travail de conception et pas de programmation
- Le métier de concepteur s'éloigne de plus en plus du code
- Ne pas utiliser les patrons de conception si on n'est pas sûr d'avoir clairement identifié le bon modèle
- Les bons concepteurs **réutilisent** les solutions existantes
- Gare aux patrons abîmés et anti-patrons (`antipatterns`) !
  `https://en.wikipedia.org/wiki/Anti-pattern`

Ne devenez pas parano !



- parfois il est plus simple de n'utiliser aucun pattern
- utilisez les DP que si vous développez à long terme