R3.04 – Qualité de développement

SOLID – Design Patterns

iut INSTITUT UNIVERSITAIRE DE TECHNOLOGIE VALENCIENNES • CAMBRAI • MAUBEUGE
Université Polytechnique HAUTS-DE-FRANCE
IUT département informatique MAUBEUGE

Principe de la conception logicielle

- L'objectif est de concevoir MAIS SURTOUT « BIEN » concevoir
- En modélisation (analyse/conception) : tout est ouvert…MAIS il faut FAIRE LES BONS CHOIX.

- Qu'est-ce qu'une mauvaise conception ?
  - Robert Martin : (http://www.objectmentor.com/omTeam/martin_r.html)
  - Rigide : chaque modification affecte trop d'autres parties du système
  - Fragile : quand une modification est apportée, des parties inattendues du systèmes ne fonctionnent plus
  - Immobile : difficile à réutiliser pour une autre application (trop de parties emmêlées (pire que la dépendance))

SOLID

- **S**ingle responsibility principle
- **O**pen close principle
- **L**iskov principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Principe de responsabilité unique

- Si une classe est chargée de plusieurs responsabilités
  - => quand on modifie une, ca peut avoir un impact sur les autres.
  - Ex. le serveur de mail doit-il être dépendant du format du contenu ?
    - Réponse : NON
- SOLUTION : 1 classe a 1 seule responsabilité.
- Principe simple et intuitif en théorie
  - En pratique ?
  - Utiliser une interface pour « gérer » le contenu
- Doit être appliqué lors de la définition des classes puis vérifié pour « assurer » l'évolutivité du système

Principe Ouvert-Fermé

- Les entités logicielles (ex. Classes, modules, fonctions, etc.) doivent être
  - **Ouvertes** aux extensions
    - On peut toujours ajouter des cas, des classes, des méthodes aux classes, etc.
  - ET
  - **Fermées** à la modification
    - On ne touche pas à ce qui est utilisé par les outils existants
- Préconise l'utilisation des classes abstraites
- L'utilisation des patterns Template et Strategy

Liskov Substitution Principle

- ***« Subtypes must be substituable for their base types. »***
- **Les sous classes doivent pouvoir être substituées à leur classe de base sans altérer le comportement de ses utilisateurs**
- Cela signifie **qu'il ne faut pas**
  - lever d'exception imprévue (comme UnsupportedOperationException par exemple), ou
  - modifier les valeurs des attributs de la classe principale d'une manière inadaptée, ne respectant pas le « contrat » défini par la méthode.

- Les cas de violation ne sont pas si fréquents nous concevons en général des modèles qui ne violent pas ce principe. Cependant, **cela peut se produire par précipitation ou méconnaissance des détails d'implémentation** des classes de base et sa détection est, la plupart du temps, difficile.

Principe de ségrégation d'interface

- Interface
  - les interfaces permettent de spécialiser différemment des comportements.
  - Une classe qui implémente une interface doit implémenter toutes ses méthodes.

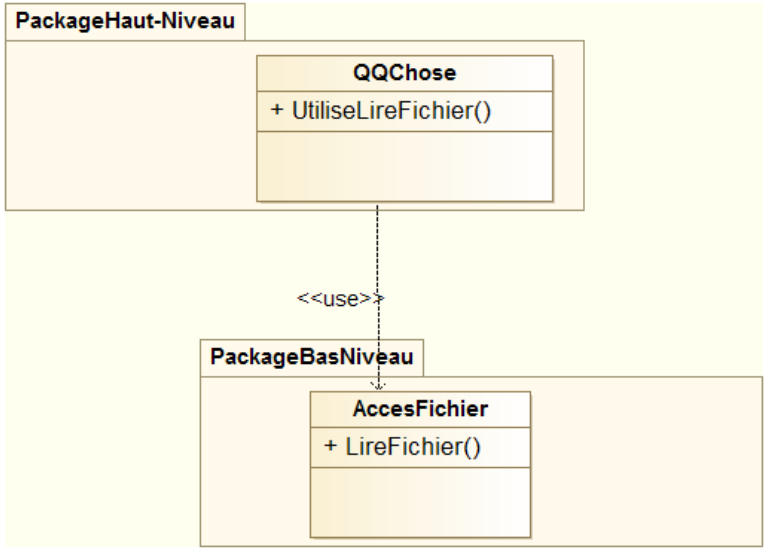
=> il faut qu'il y ait un sens à associer les méthodes placées dans les interfaces

  - Si une classe veut implémenter 1 méthode mais pas les autres, elle va les implémenter de manière factice…
- Solution :
  - proposer 2 (n) interfaces afin de ne pas obliger à développer du « factice »
  - Utiliser le pattern Adapter

Principe inversion de dépendance (1/2)

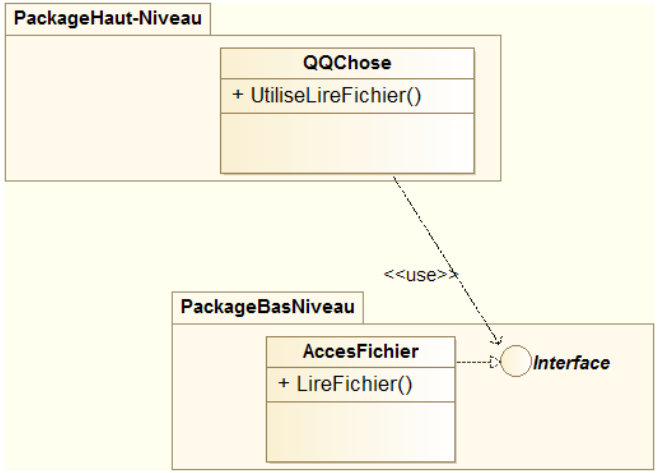
- Du bas niveau au haut niveau
  - Ex. une méthode de haut niveau (HN) dépend d'une méthode (class) de bas niveau
  - Si on souhaite faire EVOLUER la méthode HN afin d'utiliser un « utilitaire » plus performant
  - => Problème récurrent modification de la méthode HN + série de test à refaire !

- En général
  - des entités logicielles qui gèrent le bas niveau
    - Ex. lecture d'un fichier
  - des entités logicielles qui gèrent le haut niveau



Principe inversion de dépendance (2/2)

- Solution
  - Du haut niveau au bas niveau
    - Le haut niveau a des besoins
    - Il existe des « méthodes » bas niveau effectuant un « service » (ex. lecture d'un fichier)
    - On ajoute une interface qui propose le service sans avoir les détails d'implémentation
  - Utilisation de *Interface*
  - Le patron Template Method respecte ce principe



DESIGN PATTERN

iut INSTITUT UNIVERSITAIRE DE TECHNOLOGIE VALENCIENNES • CAMBRAI • MAUBEUGE
Université Polytechnique HAUTS-DE-FRANCE
IUT département informatique MAUBEUGE

Définition

Un *design pattern* (patron de conception) est une solution de conception commune à un **problème récurrent** dans un **contexte donné**.

⇒Un pattern est un couple **problème/solution** nommé et bien connu, qu'on peut appliquer à de nouveaux contextes.

⇒Il est accompagné de conseils sur la façon de l'appliquer et d'une présentation de ses inconvénients, implémentations, variantes, etc.

⇒Existe sous différentes formes d'abstraction

Historique

- Kent Beck (Extreme Programming)
  - Milieu des années 80
- 1994, livre de Gamma, Helm, Johnson et Vlissides, Design Patterns
  - Les auteurs étant 4, les patterns sont nommés patterns de la bande des 4 « GoF » (Gang of Four).
  - 23 patterns pour la conception objet
- GRASP (General Responsability Assignment Software Patterns)
  - 9 patterns de conception concernant l'affectation des responsabilités
- Autres Patterns dans d'autres domaines (ex. HCI)

23 patrons de conception orienté objet

PATTERNS GOF

Patterns GoF

- Patterns de construction
  - Abstract factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Patterns de structuration
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Patterns de comportement
  - Chain of responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template method
  - Visitor

Objectif : abstraire les mécanismes de création d'objets

Patterns encapsulent l'utilisation des classes concrètes et favorisent l'utilisation des interfaces.

⇒ Augmentent les capacités d'abstraction dans la conception globale du système

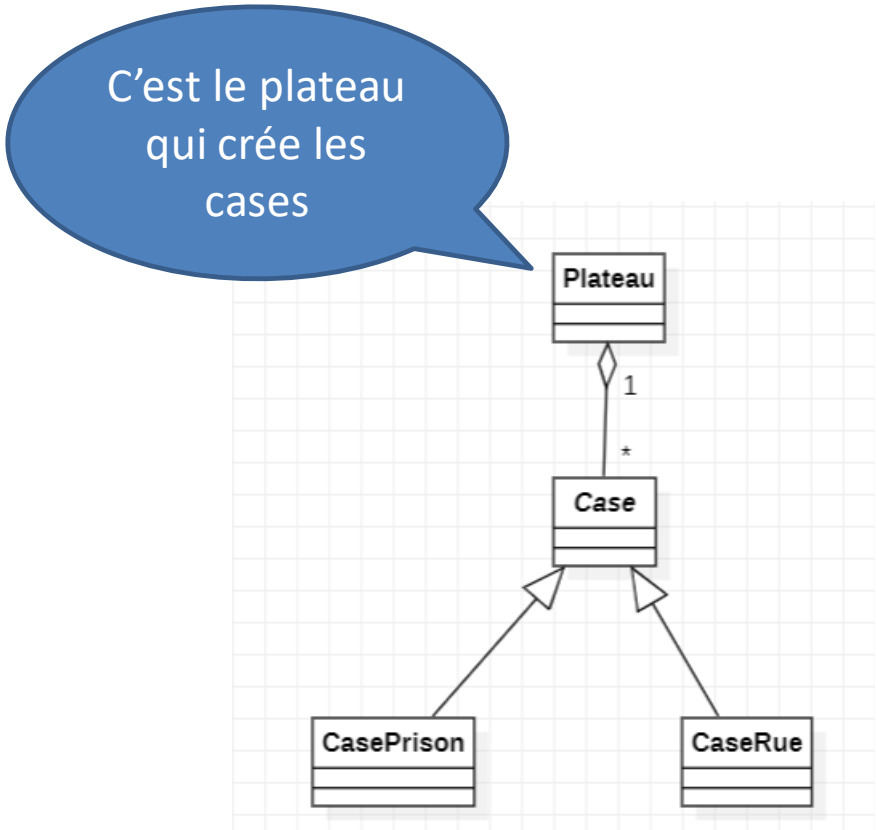
Les patrons :

1. Abstract factory,
2. Factory method,
3. Builder,
4. Prototype,
5. Singleton

PATTERNS DE CONSTRUCTION

Exemple Problème récurrent

- Exercice :
  - Créer un plateau de jeu contenant des cases (Prison et rue)
  - Exemple : Monopoly

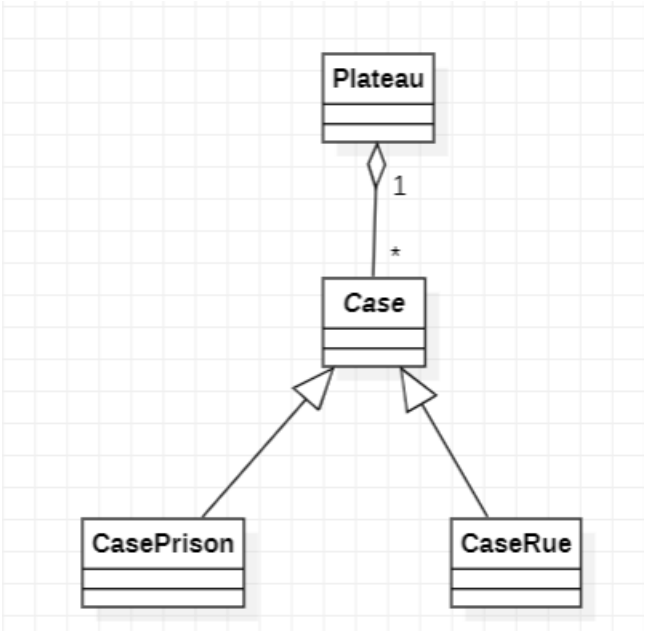


- Solution directe :
  - Case
    - = la classe abstraite
    - a des sous-classes concrètes :
      - Case rue, case e-, case prison
  - Plateau crée (instancie) les objets cases.

En Java

C'est le plateau qui crée les cases

```
public class Plateau {

public static void main(){
CasePrison prison1 = new CasePrison();
prison1.methodeCase();

CaseRue rue1 = new CaseRue();
rue1.methodeCase();
}
}
```




```
public abstract class Case {
protected abstract void
methodeCase();
}
```

```
public class CasePrison extends
Case {
protected void methodeCase(){
System.out.println("casePrison.
methodeCase()");
}
}
```

```
public class CaseRue extends
Case {

protected void methodeCase(){
System.out.println("caseRue.met
hodeCase()");
}
}
```

Exemple

- Exercice :
  - Créer un plateau de jeu contenant des cases (Prison et rue)
  - Exemple : Monopoly
  - Créer une nouvelle version du Monopoly avec d'autres types de cases !
  - Créer un autre jeu de plateau (Scrabble etc.)
  - => modification de Plateau => Comment le plateau sait ce qu'il doit mettre en argument ?

```
public class Plateau {

public static void main(){
CaseSDF sdf = new CaseSDF(0);
sdf.methodeCase();

CaseVille ville = new
CaseVille(1000);
ville.methodeCase();
}
}
```

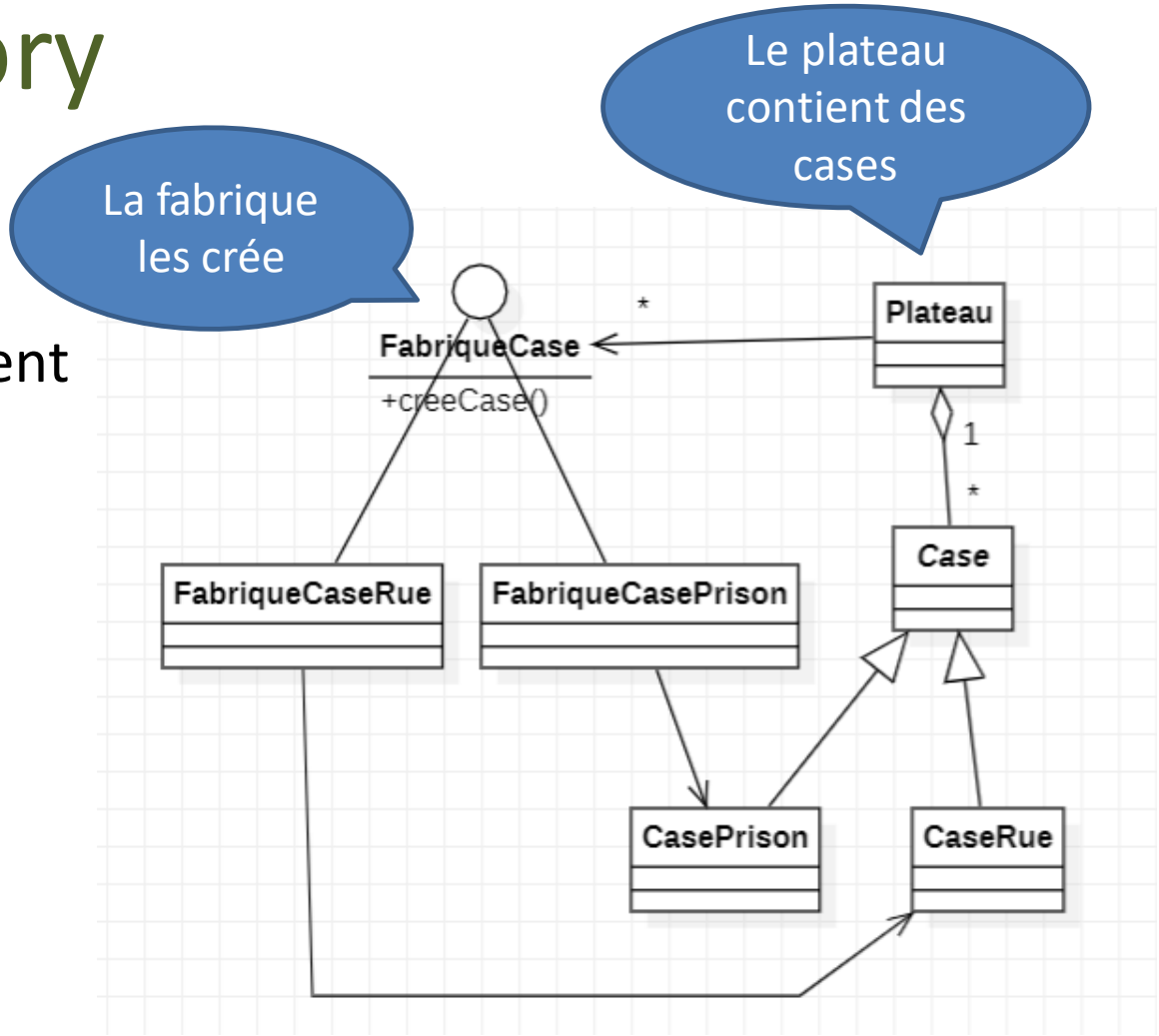
```
public class Plateau {

public static void main(){
CaseSimple caseLettre = new
CaseSimple (1);
caseLettre.methodeCase();

CaseCptDouble caseFois2= new
CaseCptDouble(2);
caseFois2.methodeCase();
}
}
```

Pattern abstract factory

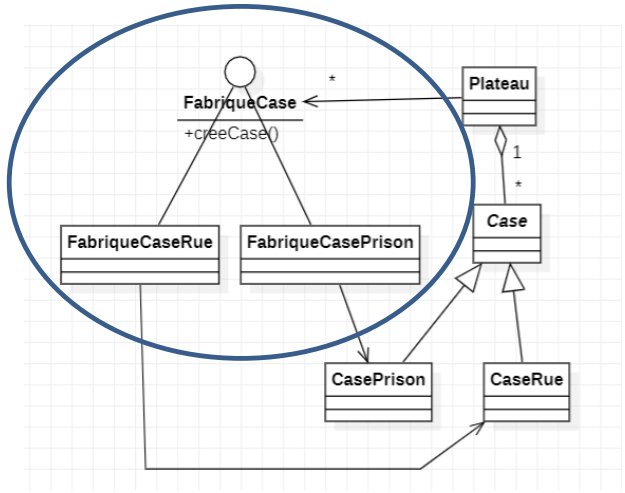
- Problème/Défaut : le plateau doit connaitre les cases (classes concrètes) et doit savoir comment les créer (détails).
- But du pattern : création d'objets regroupés en familles sans devoir connaitre les classes concrètes destinées à la création de ces objets

Pattern abstract factory

- Solution :
  - **interface** FabriqueCase qui contient la signature des méthodes pour définir chaque case.
  - Plateau est maintenant indépendant des types de cases.



Coté Fabrique




```
public interface FabriqueCase {
public Case creeCase() ;
}
```

```
public class FabriqueCasePrison
implements Fabrique {

public Case creeCase() {
return new CasePrison(222);
}

}
```

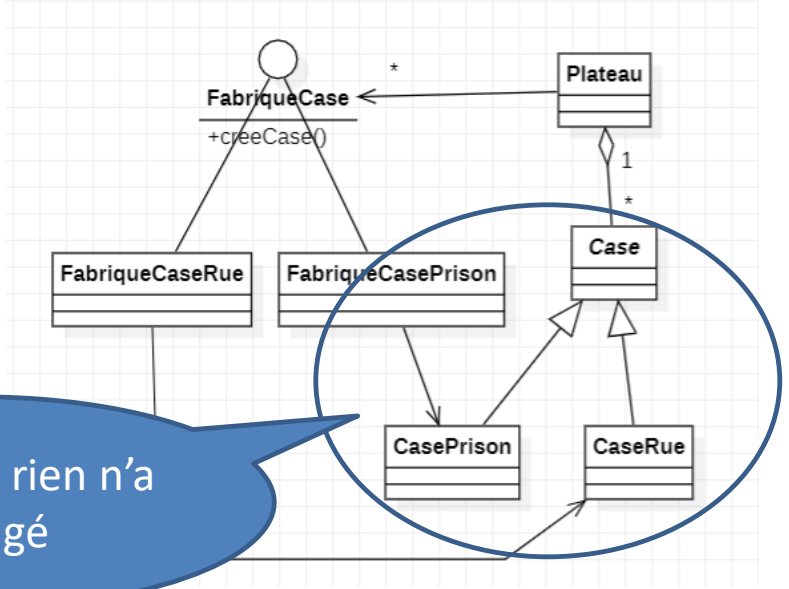
C'est la fabrique qui connait les détails de la case

```
public class FabriqueCaseRue
implements FabriqueCase {

public Case creeCase() {
return new CaseRue(100000);
}
}
```

C'est la fabrique qui connait les détails de la case

Coté Case

```
public abstract class Case {
public abstract void
methodeCase();
}
```

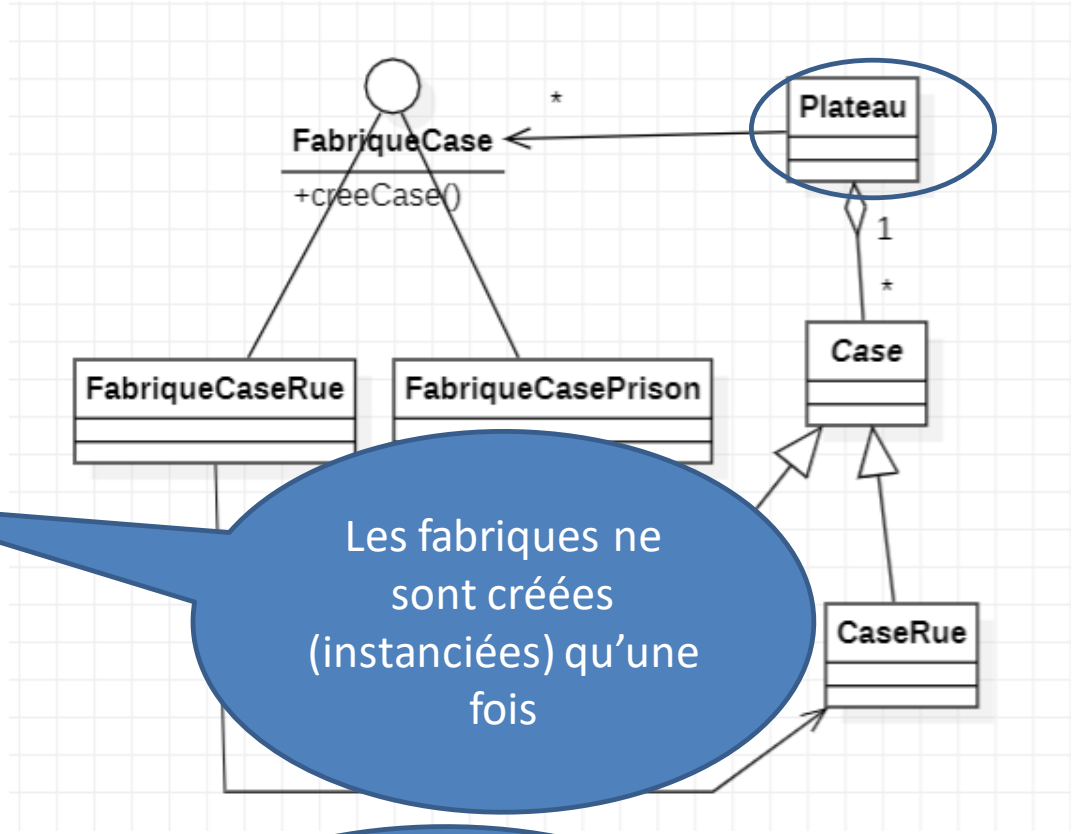



```
public class CasePrison extends Case {

public void methodeCase() {

System.out.println("CasePrison.methodeCase()");
}
}
```

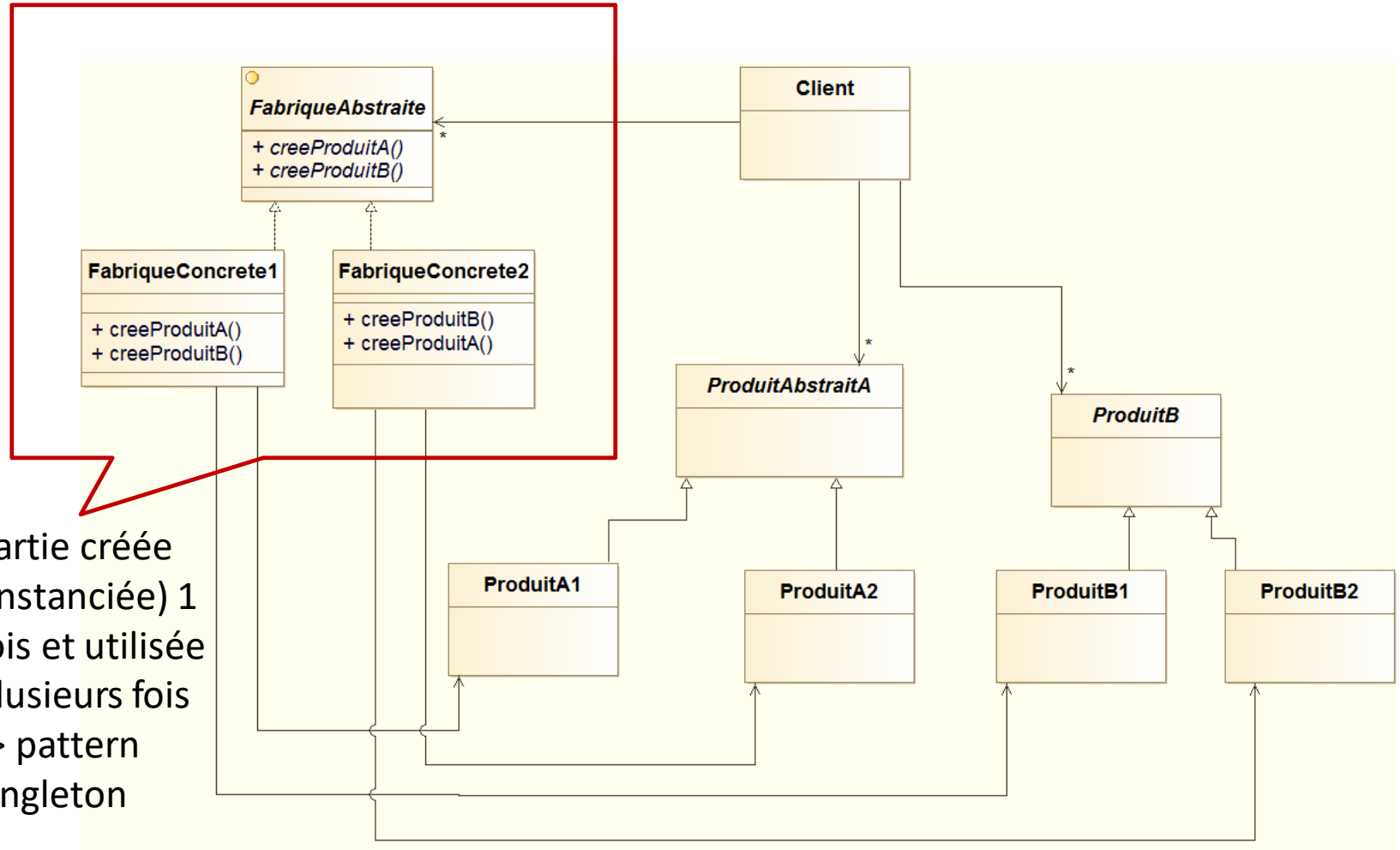
```
public class CaseRue extends Case {

public void methodeCase() {

System.out.println("CaseRue.methodeCase()");
}
}
```

Le Plateau

```
public class Client {
//le plateau du Monopoly
public static void main(String[] args) {
// les fabriques ne sont créées qu'une fois !
Fabrique fabriqueCasePrison = new FabriqueCasePrison();
Fabrique fabriqueCaseRue = new FabriqueCaseRue();
Case uneCase = null;

System.out.println("Utilisation de la premiere fabrique");
// on utilise la fabrique autant que l'on veut !
uneCase = fabriqueCasePrison.creeCase();
uneCase.methodeCase();

System.out.println("Utilisation de la seconde fabrique");
uneCase = fabriqueCaseRue.creeCase();
uneCase.methodeCase();
}
}
```

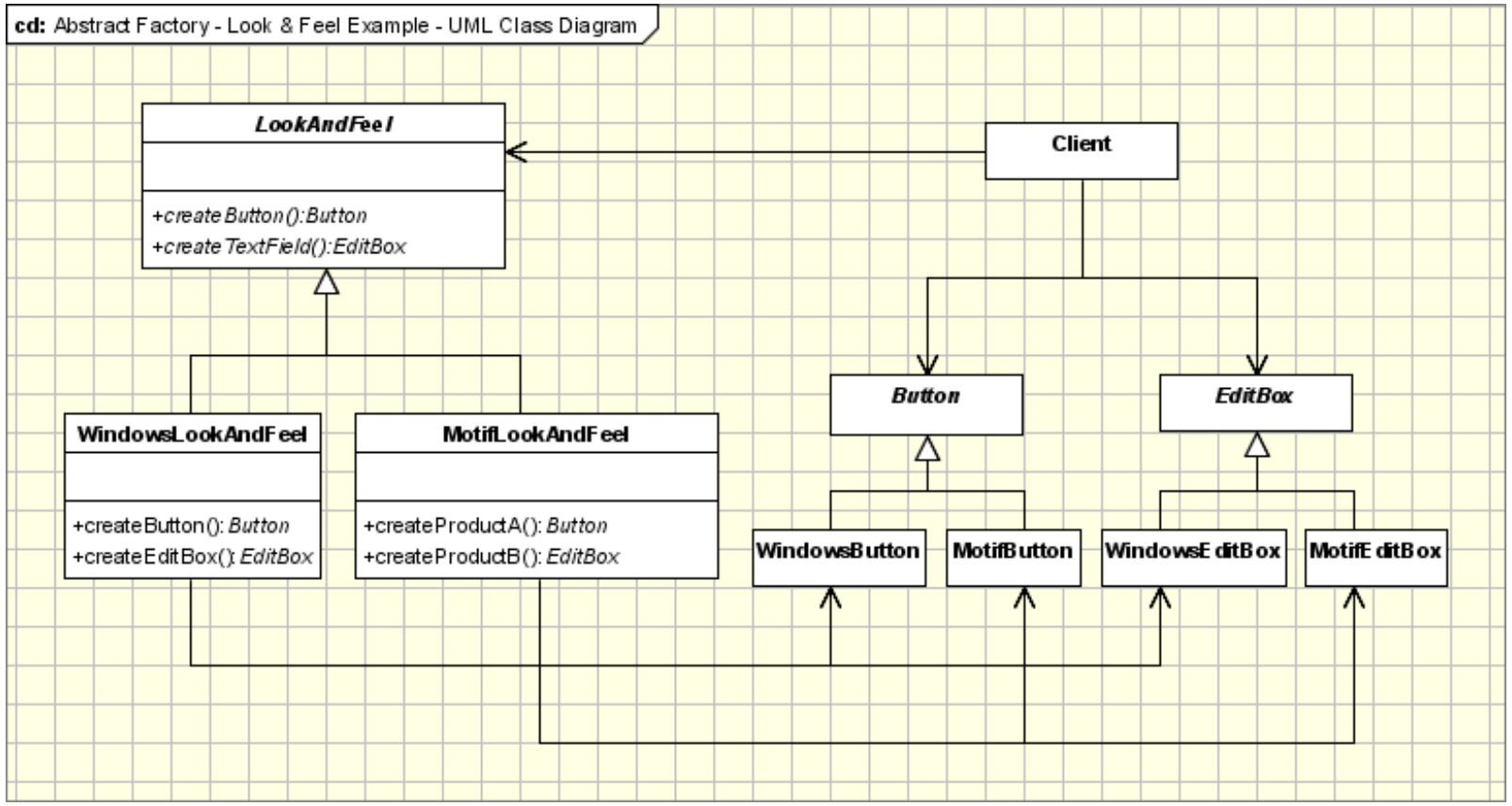


Pattern abstract factory



Partie créée (instanciée) 1 fois et utilisée plusieurs fois -> pattern singleton

Exemple de Pattern Abstract factory



http://www.oodesign.com/abstract-factory-pattern.html

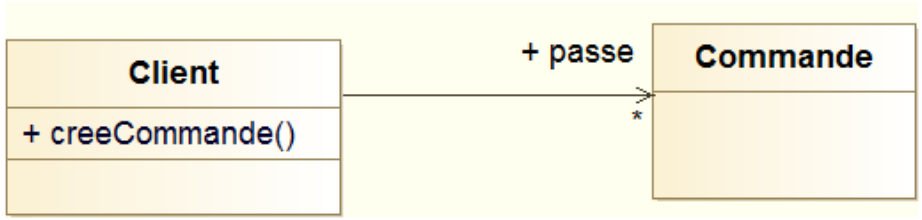
Pattern abstract factory

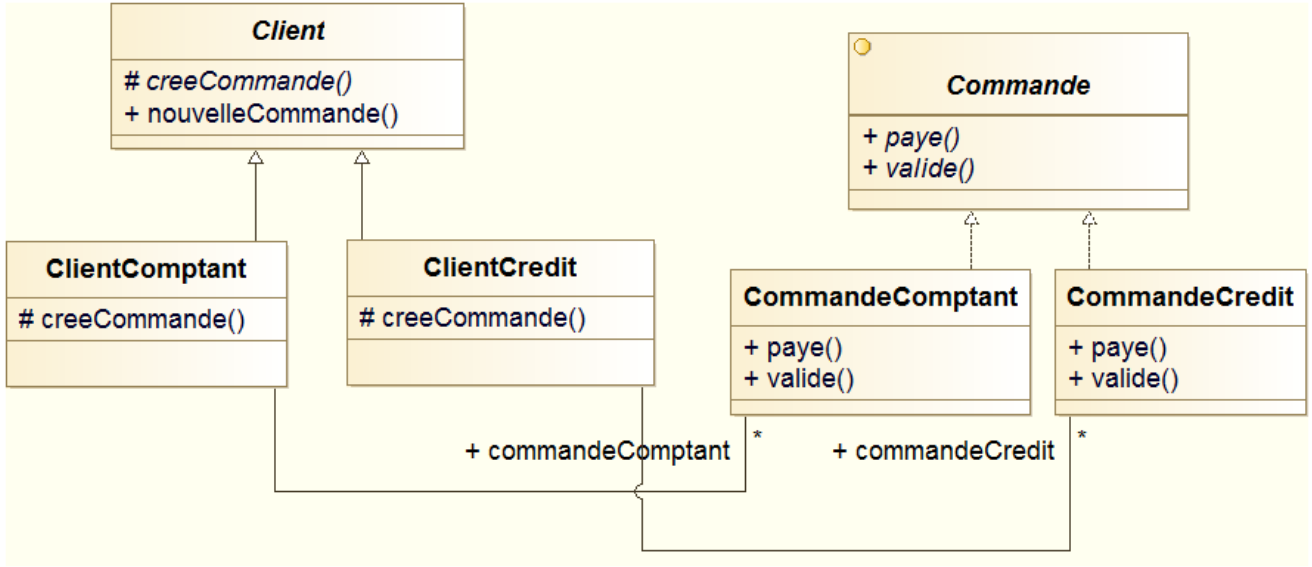
- Il est utilisable lorsque :
  - détermination du type d'objet à créer qu'à l'exécution
  - volonté de centraliser la création des objets
  - Le système a besoin d'indépendance entre le chemin d'accès et la création.
  - Le système est ou devrait être configuré pour travailler avec de multiples familles de produits.
  - …

Pattern Factory Method

- But : introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective.

- Modélisez :
  - Un Client passe une commande.
  - Certains clients payent la commande comptant, d'autres à crédit. On ne le sait pas à l'avance…
    - (concept de généricité)

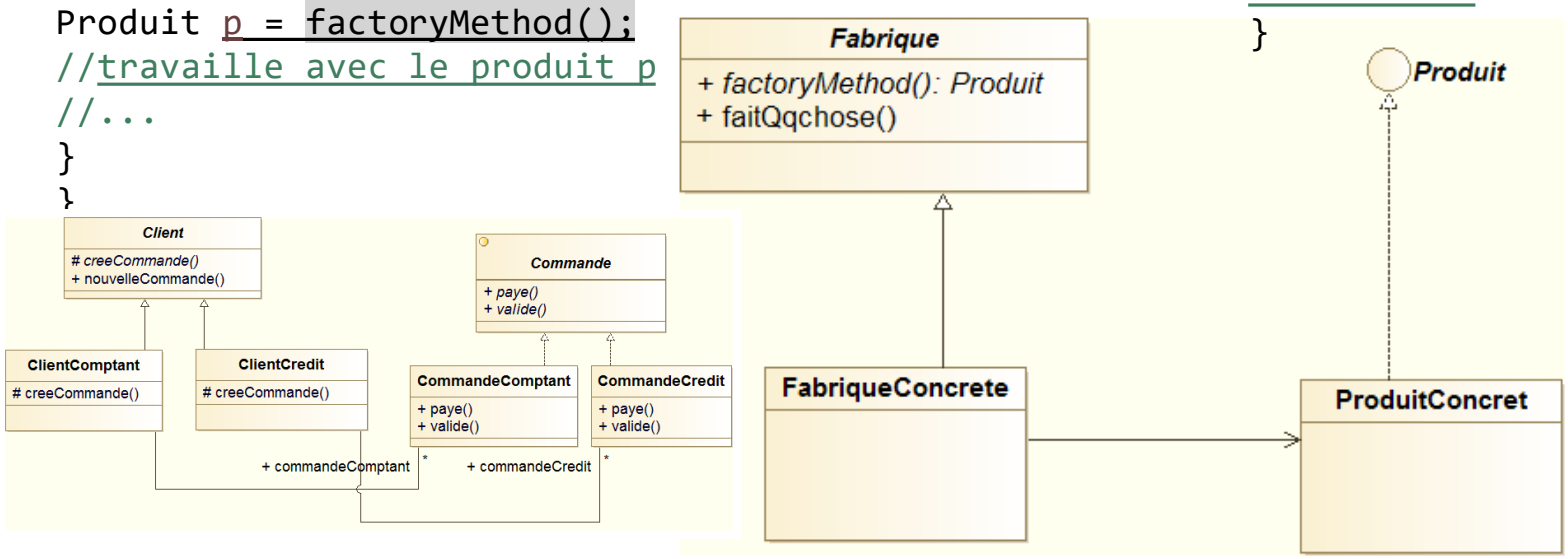




Pattern Factory Method

```
public abstract class Fabrique {
public abstract Produit factoryMethod();

public void faitQqchose(){
Produit p = factoryMethod();
//travaille avec le produit p
//...
}
}
```

```
public interface Produit {
//particularite d'un produit
// methodes fournies par
l'interface
}
```




```
public class FabriqueConcrete extends
Fabrique {
public Produit factoryMethod(){
return new ProduitConcret();
}
}
```

```
public class ProduitConcret implements
Produit {
//methodes de l'interface doivent etre
implementees
}
```

- http://www.oodesign.com/factory-method-pattern.html
- Favorise le polymorphisme

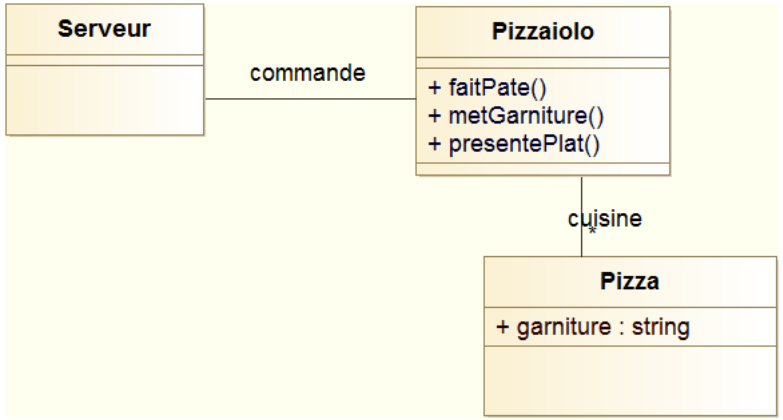
Pattern Builder

- Objectif : abstraire la construction d'objets complexes de leur implantation
  - Le client peut créer des objets complexes sans se préoccuper des différences d'implantation

Pattern Builder

- Exercice : Modéliser un SI de pizzeria
  - un serveur demande une pizza au pizzaiolo
  - à l'aide d'un diagramme de classes

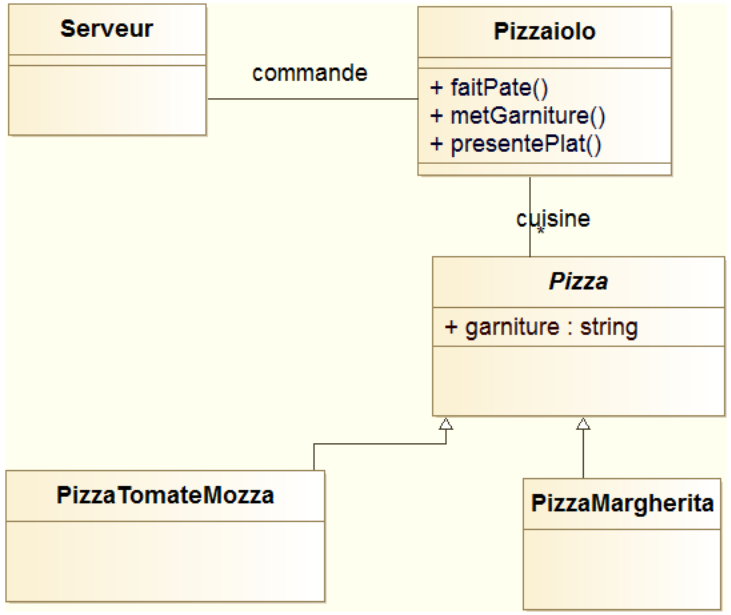
1ère version :



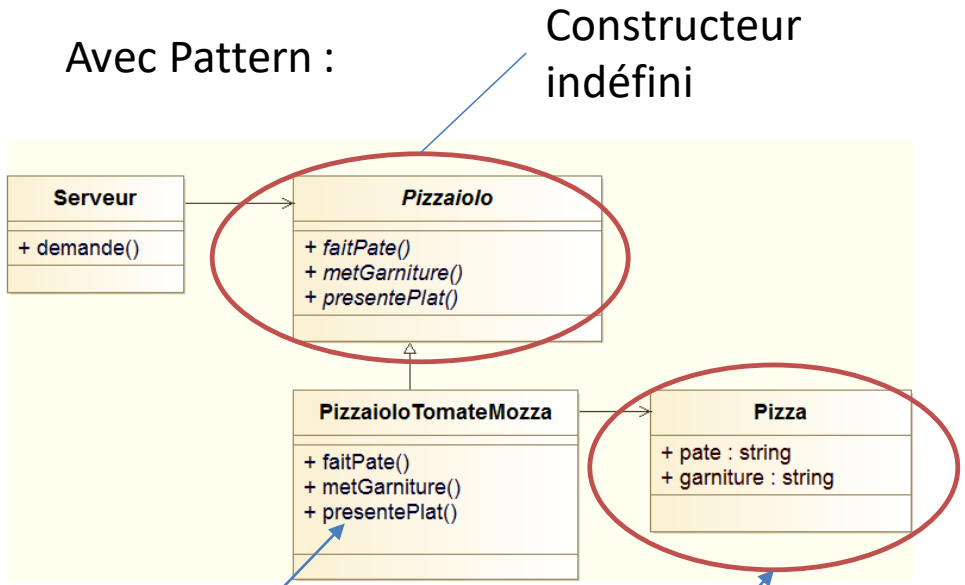
- En général : les pizzas font partie d'une liste proposée (tomate-Mozza, 4 fromages, saumon, bœuf épicé, vegetarienne)
- => spécialisation

Pattern Builder

Modèle conceptuel sans Pattern :



Avec Pattern :

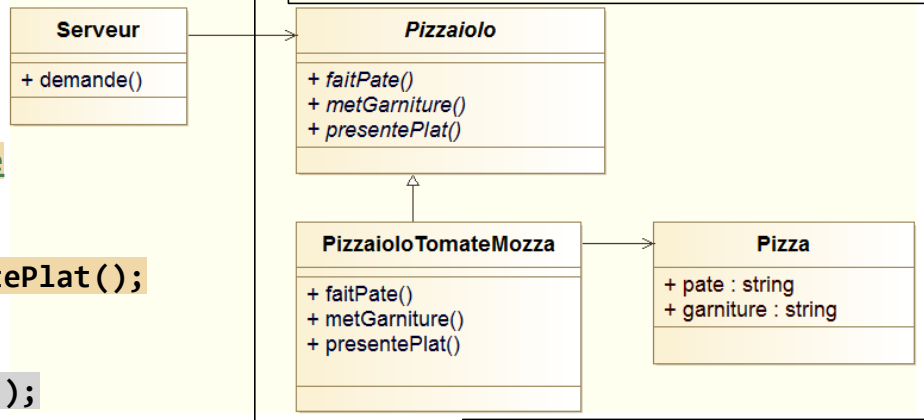


Rappel objectif : Le client peut créer des objets complexes sans se préoccuper des différences d'implantation

En Java

```
package patternBuilder;

public abstract class Pizzaiolo {

public abstract void faitPate();
public abstract void metGarniture();
public abstract Pizza presentePlat();
}
```

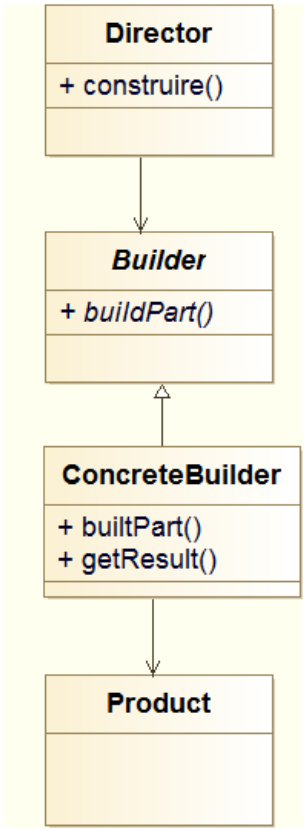
```
package patternBuilder;

public class Serveur {
private Pizza pCommandee;
//aurait pu etre plus generique
public void demande(){
pCommandee = (new
PizzaioloTomateMozza()).presentePlat();
}
public String getCommande(){
return pCommandee.getGarniture();
}
```




```
package patternBuilder;

public class PizzaioloTomateMozza extends Pizzaiolo{

PizzaTomateMozza pizzaTM;

PizzaioloTomateMozza(){
this.faitPate();
this.metGarniture();
this.presentePlat();
}
public  void faitPate(){
pizzaTM = new PizzaTomateMozza();
}
public  void metGarniture(){
System.out.println("Le pizzaiolo met la "+ pizzaTM.getGarniture()+ " sur la pate");
```

```
package patternBuilder;

public class PizzaTomateMozza extends Pizza{
protected String garniture;
protected String pate;

public PizzaTomateMozza(){
this.garniture = "TomateMozza";
this.pate = "standard";
}
public String getGarniture()
{
return(this.garniture);
}
}
```

Pattern Builder

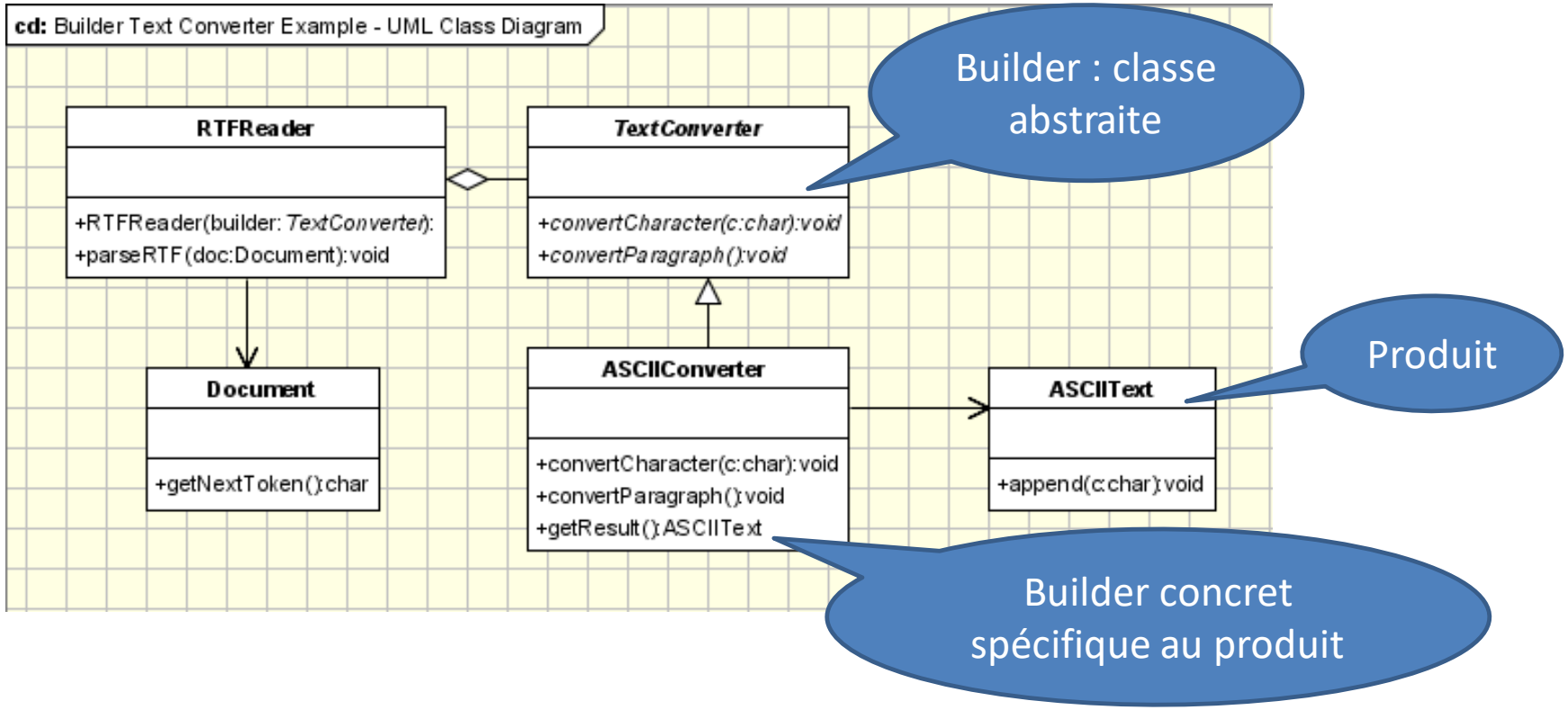


- Domaines d'utilisation :
  - Un client a besoin de construire des objets complexes
    - sans connaitre leur implantation
    - Ou
    - Ayant plusieurs représentations

- On met autant de classe concrete Builder que nécessaire
- Ex. doc.ppt / doc.pdf

Pattern Builder - exemple

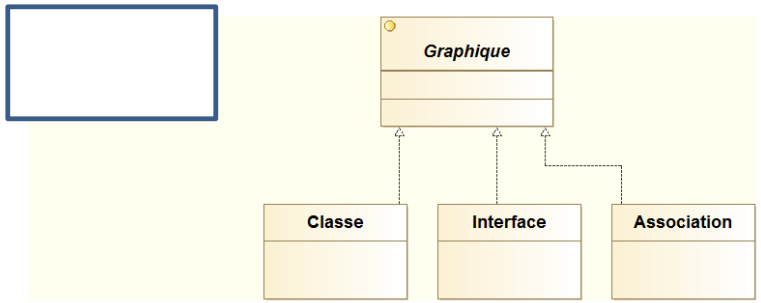
- Code sur http://www.oodesign.com/builder-pattern.html



Pattern Prototype

- Ex. Modéliser une palette d'outils permettant de créer un diagramme de classes, etc.)

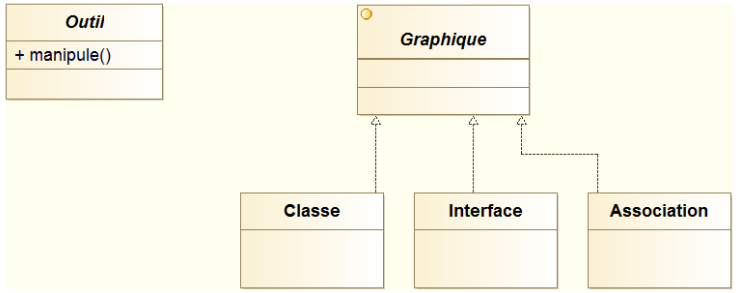
- Objectif : Spécifie le type des objets à créer à partir d'une instance de prototype et crée de nouveaux objets en copiant le prototype

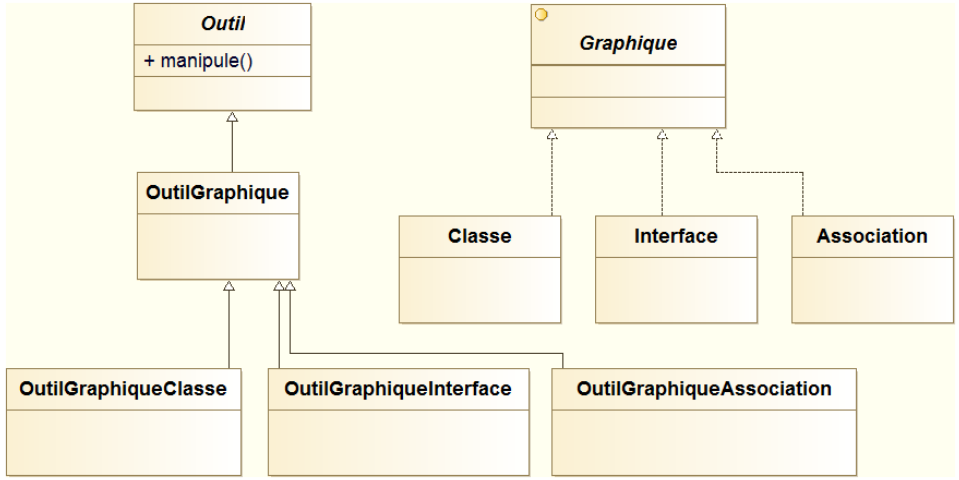


- Solution de l'exemple :
  - Classe abstraite Graphique : composants graphiques (dédiés au domaine, ici diagramme de classe)
  - Classe abstraite Outils (sélectionner, déplacer, manipuler, etc …) fournis par le framework (générique)

Pattern Prototype

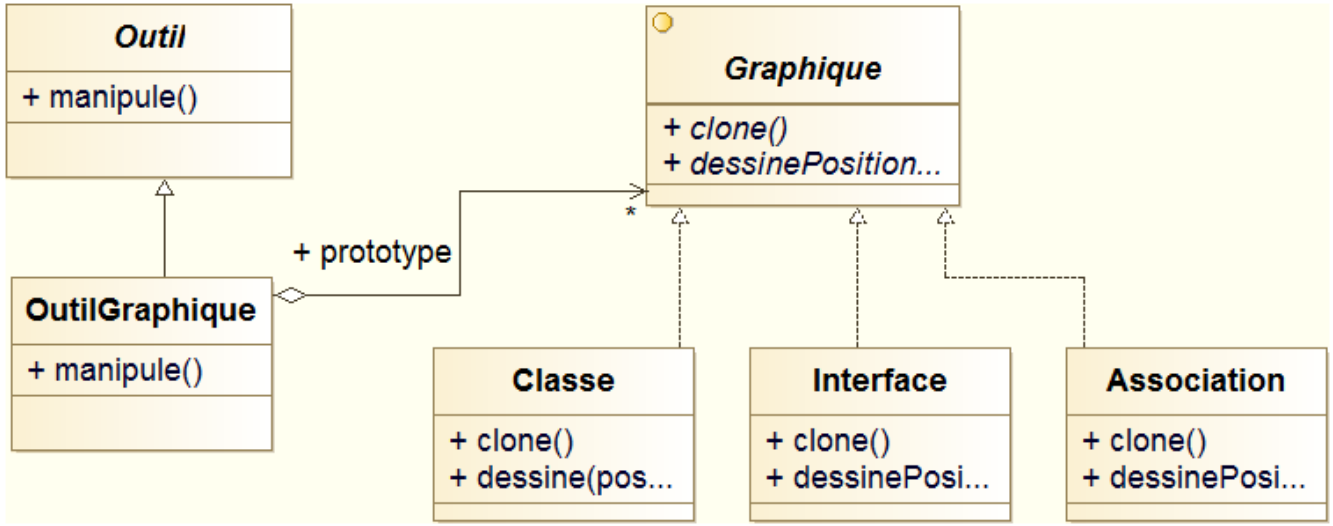
- Comment configurer les outils spécifiquement aux graphiques ?
  - Créer autant de sous-classes de OutilGraphique que d'objets graphiques
  - Beaucoup d'objets => beaucoup de sous-classes !





Pattern Prototype

- Solution : utiliser le prototype clonable !



Prototype en java

Classe.java | GraphiqueUML.java

```java
import java.util.ArrayList;
import java.util.List;

public class Classe implements GraphiqueUML{
    int x,y;
    List<String> attributs;
    List<String> methodes;

    Classe(){
        x =0;
        y =0;
        attributs = new ArrayList<String>();
        methodes = new ArrayList<String>();
    }
    @Override
    public Object clone() {
        return new Classe();
    }
```

.....

```java
    @Override
    public void dessine() {
        //ici on met le code pour dessiner

    }
    @Override
    public int getPositionX() {
        return x;
    }
    @Override
    public int getPositionY() {
        return y;
    }

}
```

Classe.java | GraphiqueUML.java

```java

public interface GraphiqueUML {
Object clone();
void dessine();
int getPositionX();
int getPositionY();
}
```

```java

public class Test {

    public static void main(String[] args) {
        Classe c1 = new Classe();
        Classe C2 = (Classe)c1.clone();

    }
```

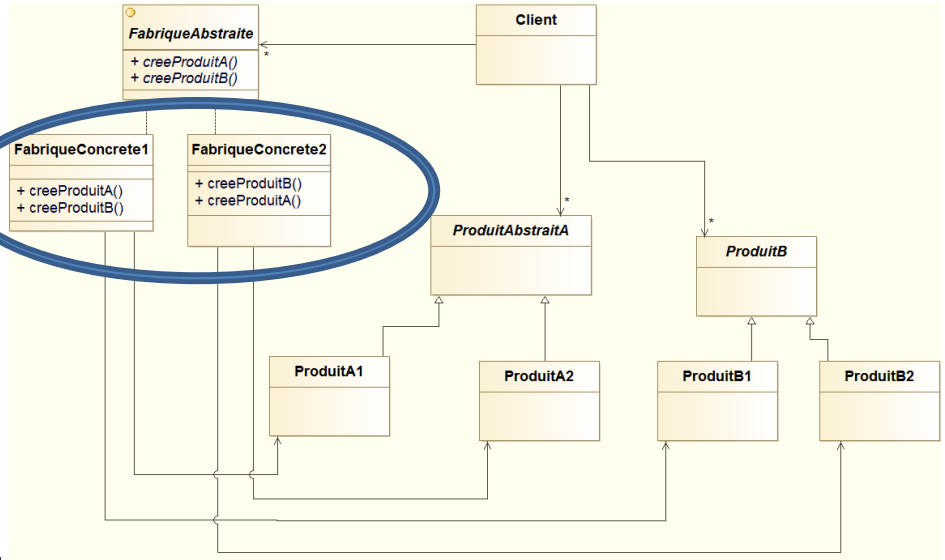
Pattern Prototype

- Indications d'utilisation :
  - Le système doit être indépendant de la manière dont ses produits son créés, composés et représentés ET
    - Si les classes sont spécifiées à l'exécution, ou
    - Pour éviter de construire une hiérarchie de classes fabriques qui réplique la hiérarchie de produits, ou
    - Si les instances d'une classe peuvent prendre un état parmi un petit nombre de combinaisons (faire une fois et dupliquer n plutôt que de faire n+1 fois).

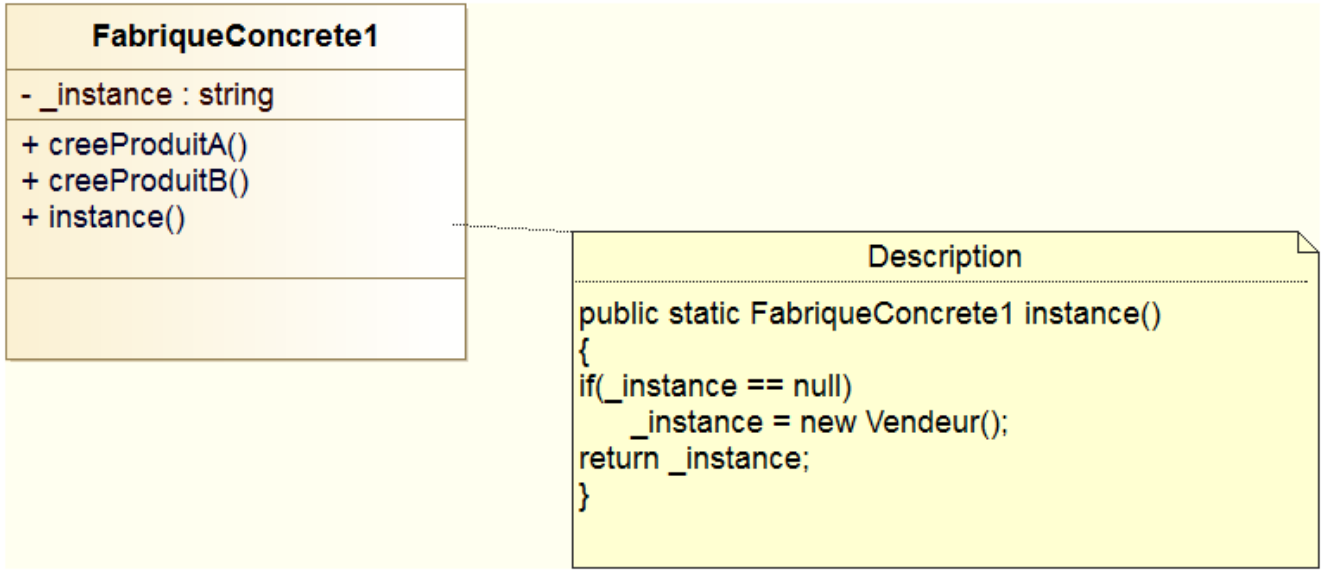
Pattern Singleton

- Cas de la fabrique de produit : on n'en crée qu'une instance

- But : assurer qu'une classe ne possède qu'une seule instance et de fournir une méthode de classe unique retournant cette instance.



Pattern Singleton



Adapter

Bridge

Composite

Decorator

Facade

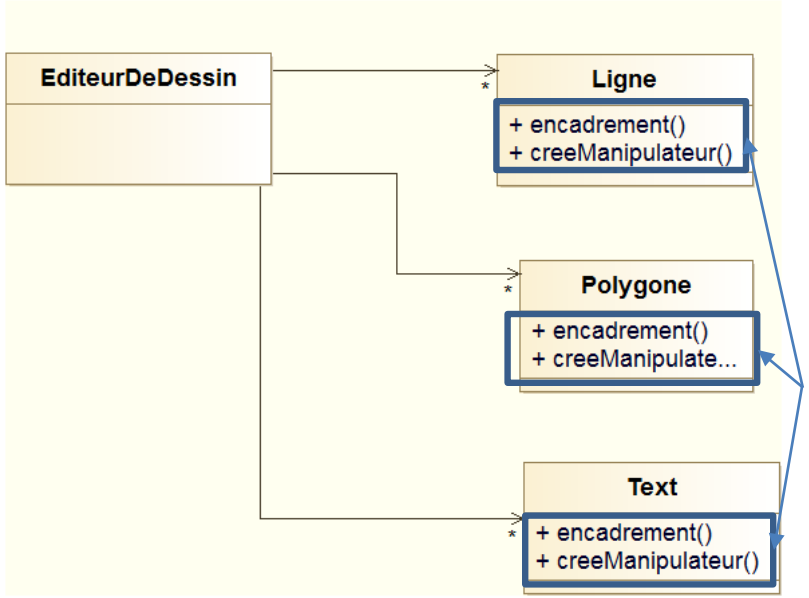
Flyweight

Proxy

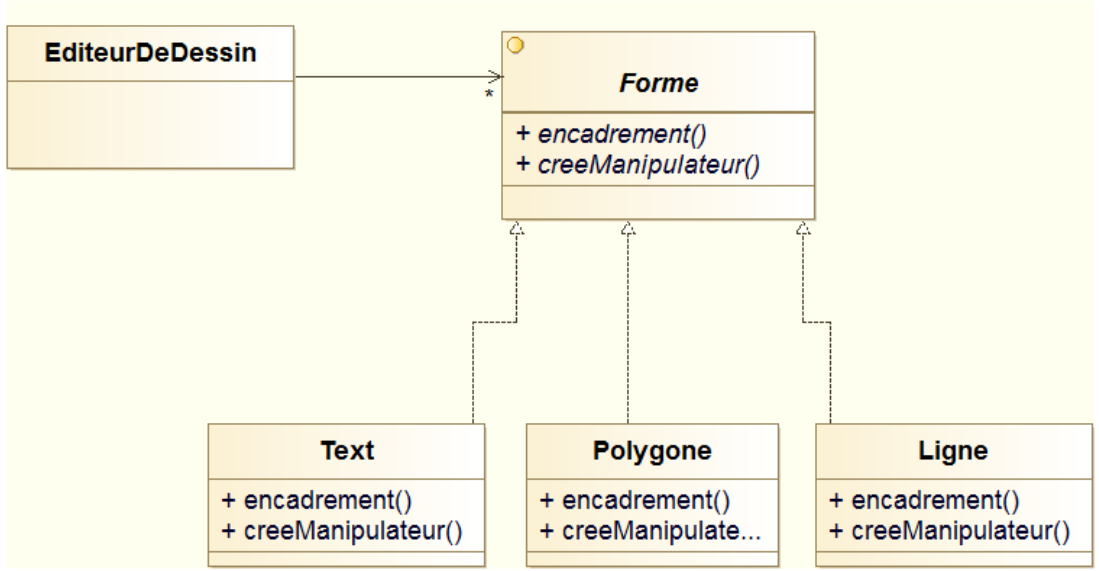
PATTERNS GOF : PATTERNS DE STRUCTURATION

Pattern Adapter

- Ex. Editeur de dessin
  - Manipule des objets graphiques
  - L'interface d'objets graphiques est appelée For
- Convertit l'interface d'une classe en une autre conforme à l'attente du client.
- Favorise la collaboration entre 2 classes

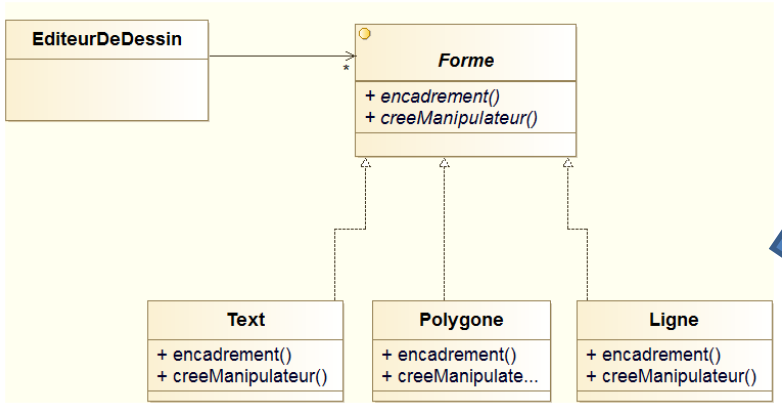


La Même structure de Méthodes

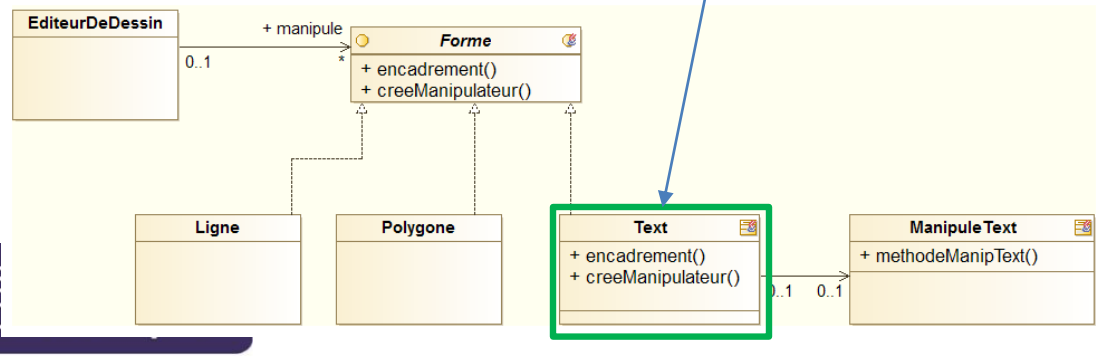


Pattern Adapter

- On dispose de ManipuleText qui fait ce qu'on recherche => on souhaite l'utiliser !
- Mais Forme de graphique et Forme texte sont différentes (interfaces différentes)
  - Solution 1 : modifier Forme texte ☹
  - Solution 2 : Modifier Forme graphique ☹
  - Solution 3 : Mettre un adapteur ☺

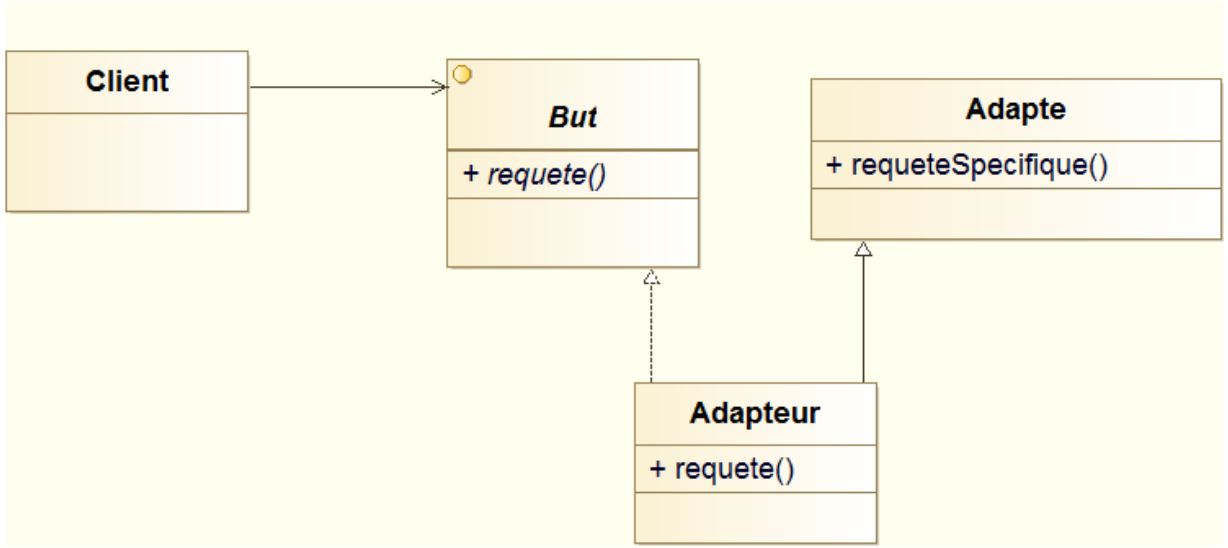



```
public class Text implements Forme {
public ManipuleText ;

public void encadrement() {
   }

public void creeManipulateur() {
Return new ManipuleText();
   }
```



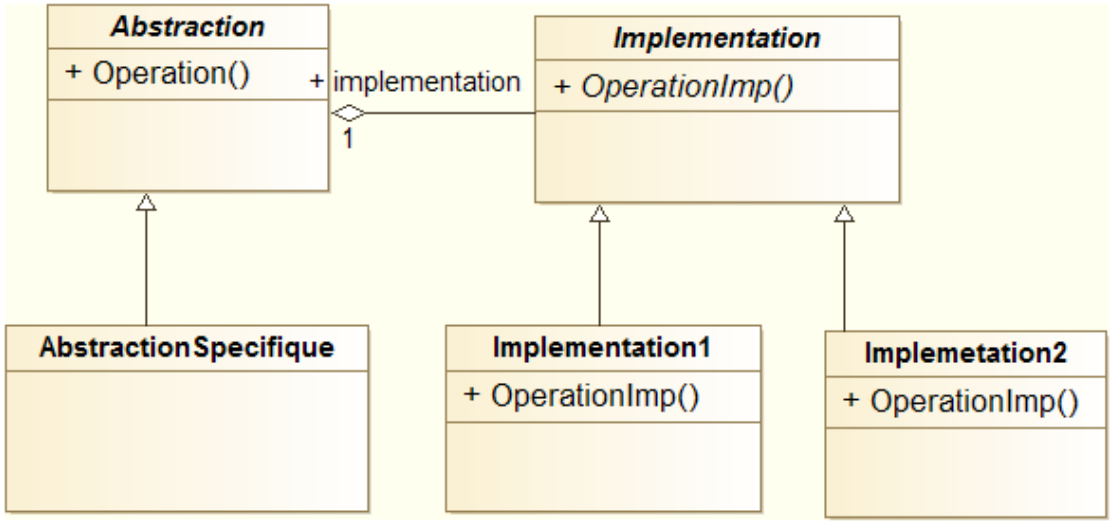
Pattern Adapter

- Indications d'utilisation :
  - On veut utiliser une classe existante mais dont l'interface ne coïncide pas avec celle escomptée
  - On souhaite créer une classe réutilisable qui collabore avec des classes sans relation avec elle et encore inconnues.
- Adaptateur de classes
- Adaptateur d'objets : composition d'objets



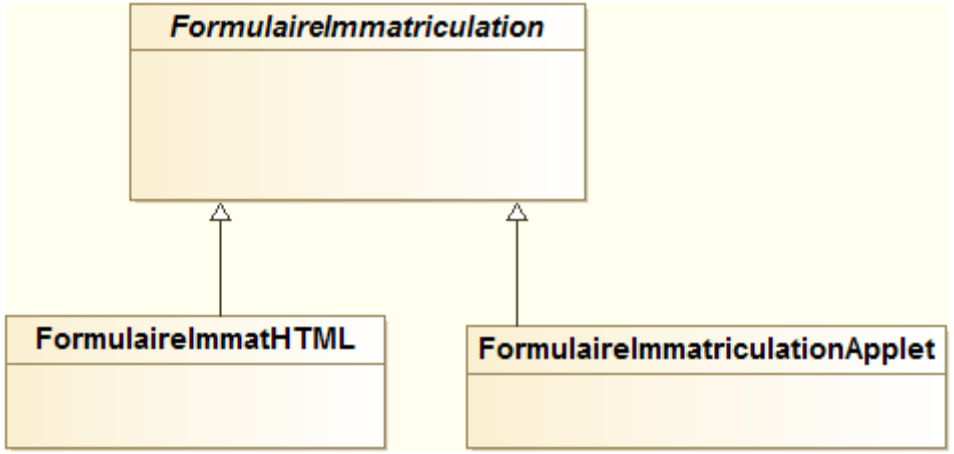
Pattern Bridge

- L'objectif de ce pattern est de découpler l'abstraction de l'implémentation
- Parfois une abstraction devrait avoir plusieurs implémentations



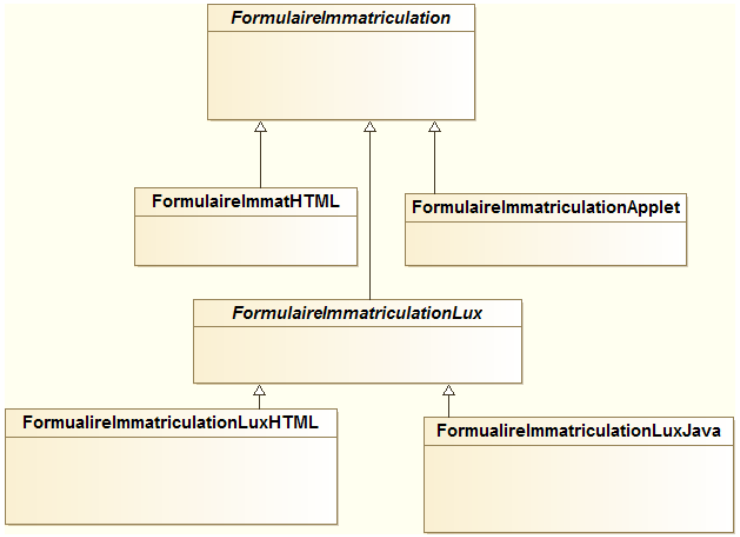
Exemple - sans Pattern Bridge

- Utilisations courantes en IHM:
  - Implémentations d'un formulaire d'immatriculation en HTML / Java /etc…
  - Solution habituelle (sans pattern) :

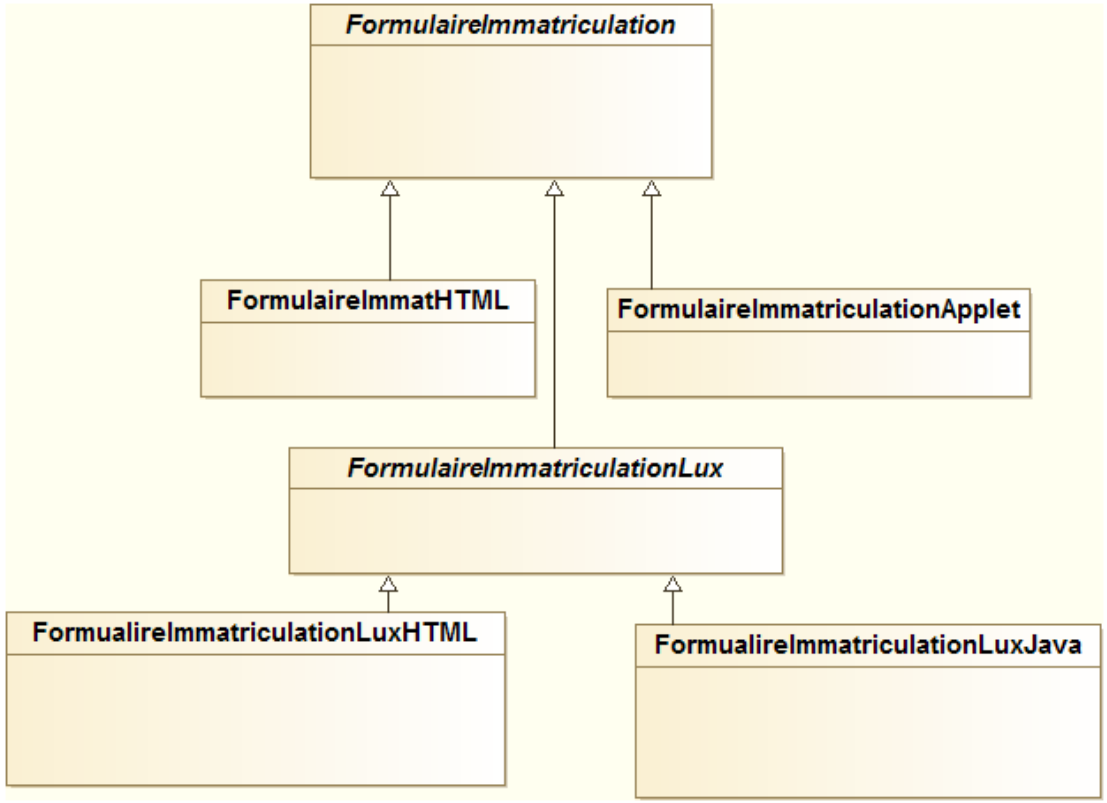


Exemple sans Pattern Bridge

- Evolution du système :
  - Les immatriculations peuvent être demandées du Luxembourg
  - Solution évoluée :

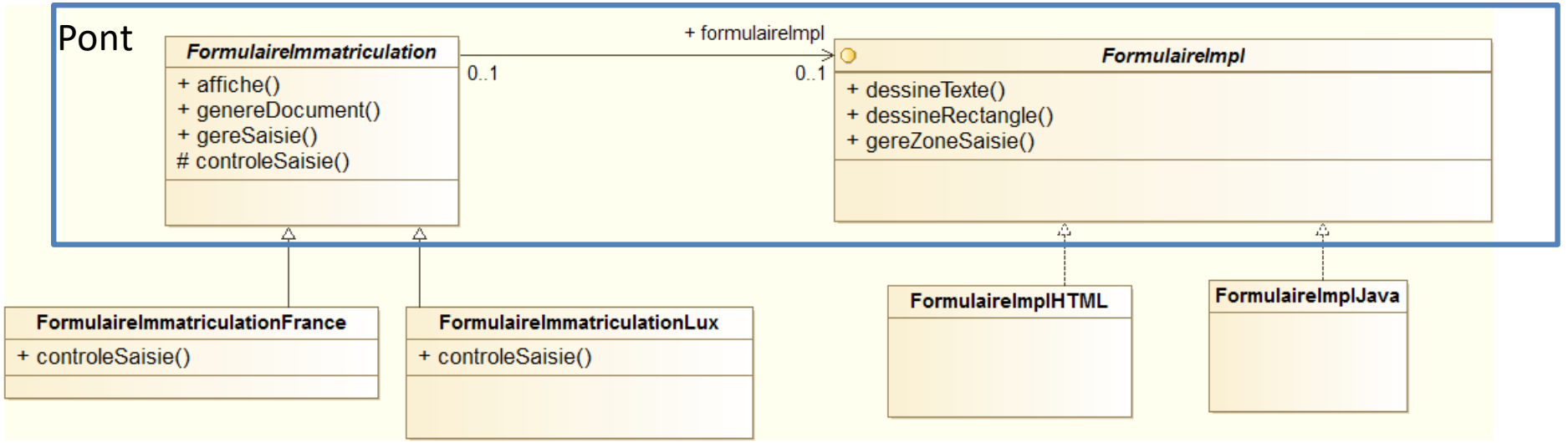


Exemple sans Pattern Bridge

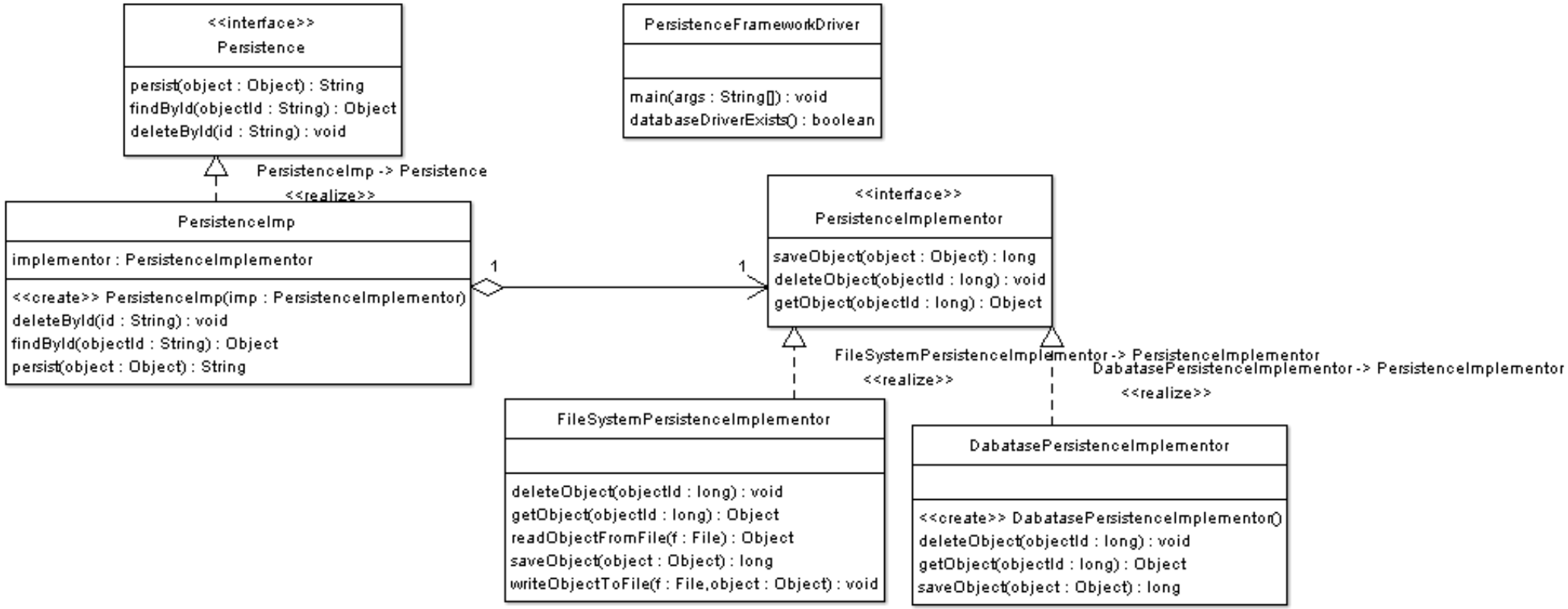


- Pbs de cette solution :
  - Mélange des sous-classes de représentation (Lux) avec les sous-classes d'implantation (HTML/Java)
  - Les clients sont dépendants de l'implantation, ils doivent interagir avec les classes concrètes de l'appli.

Exemple avec pattern Bridge



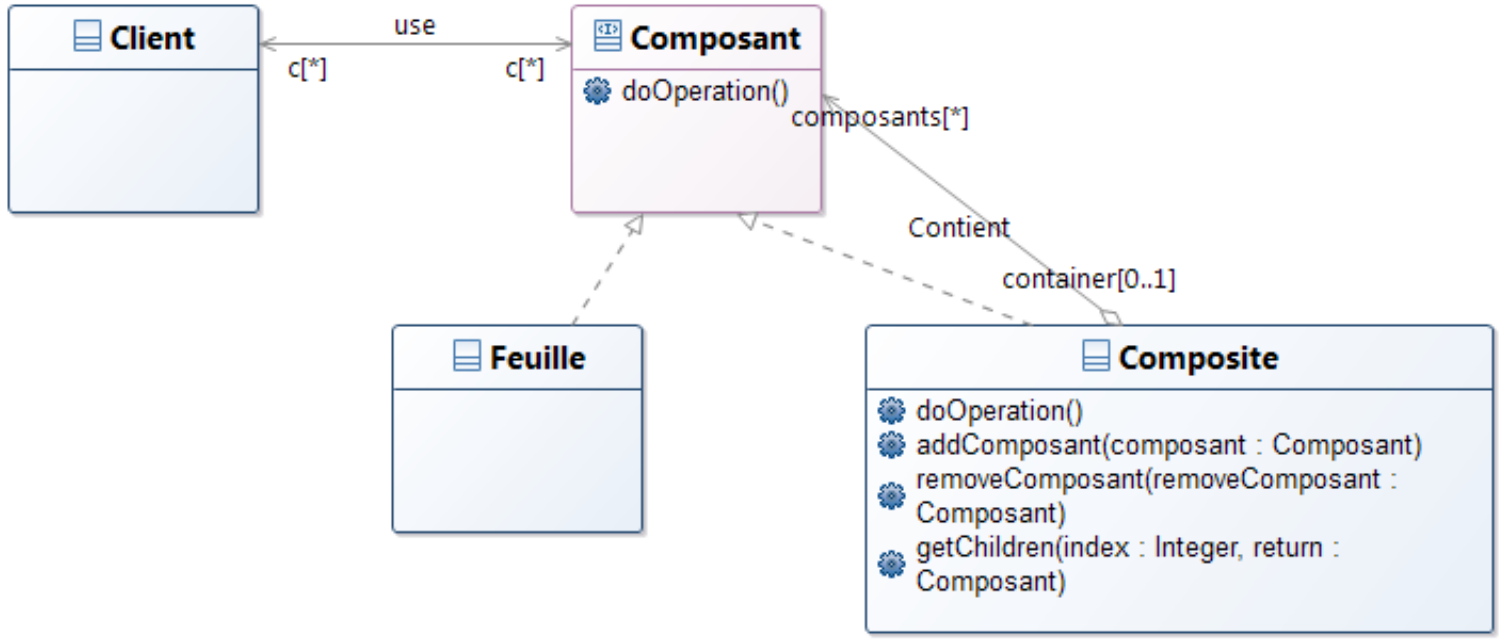
Pattern Bridge – un autre exemple



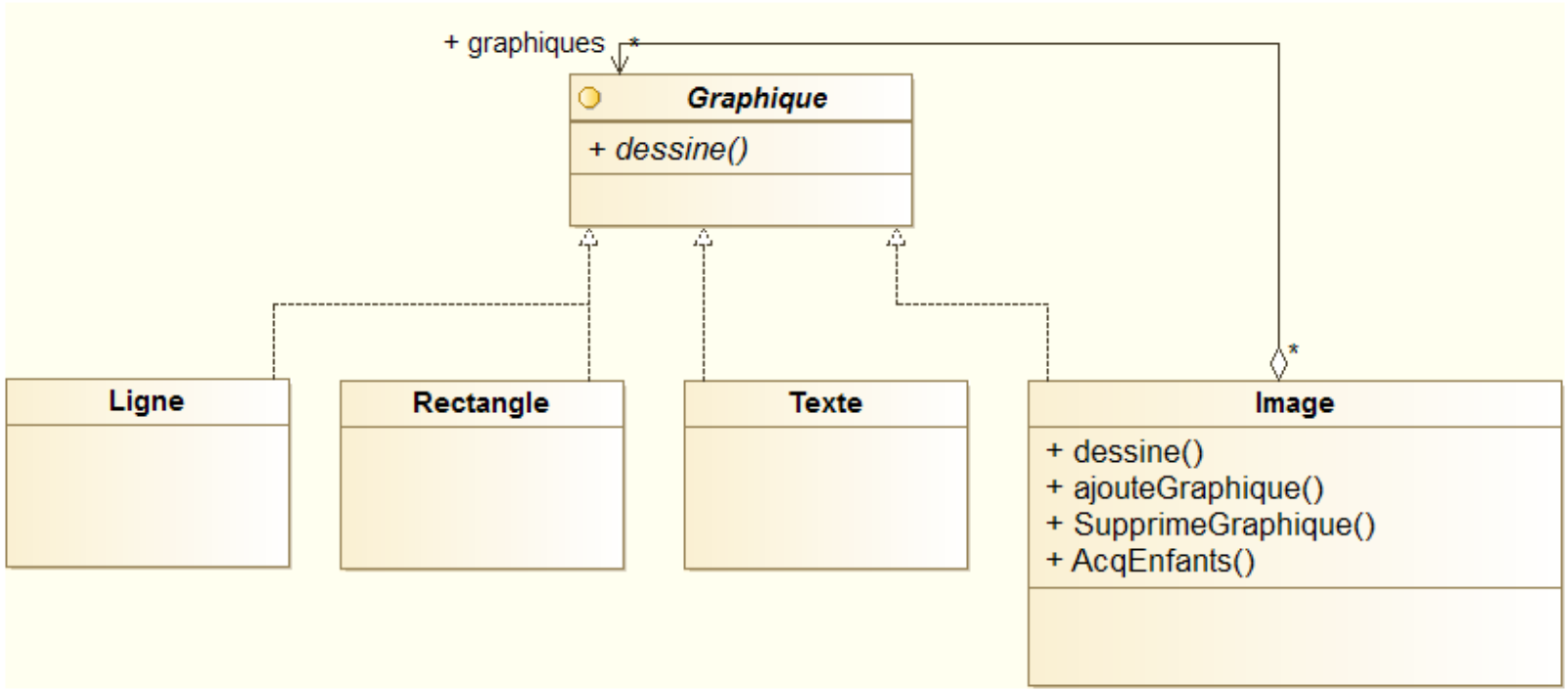
Source http://www.oodesign.com/bridge-pattern.html

Pattern Composite

- Objectif : compose des objets en structure arborescentes pour représenter des hiérarchies composant/composé

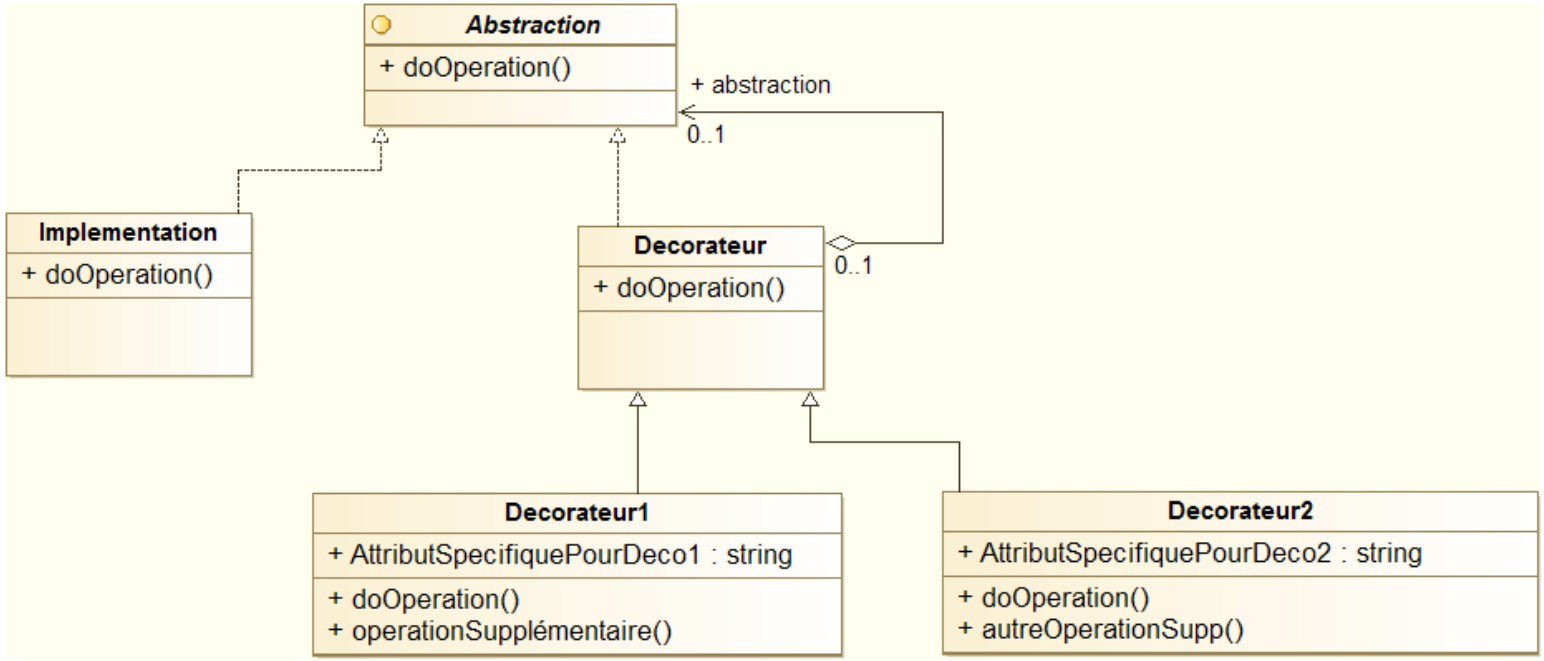


Exemple Pattern Composite



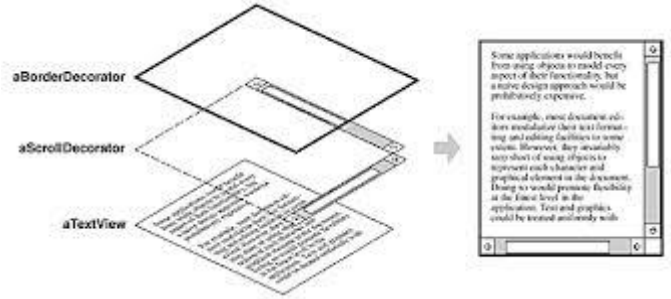
Pattern Decorator

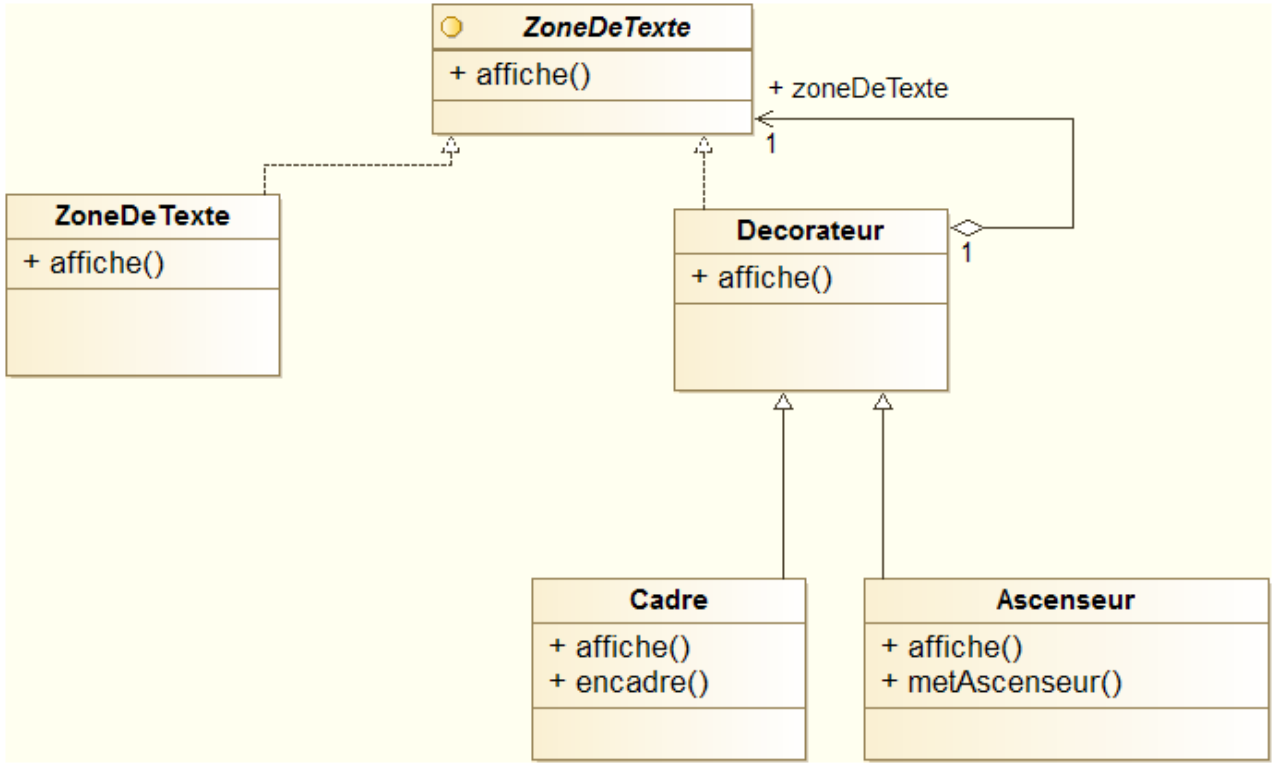
L'objectif est d'ajouter des responsabilités à une classe, sans que ce soit pour tous les objets de la classe.



Exemple Pattern Decorateur

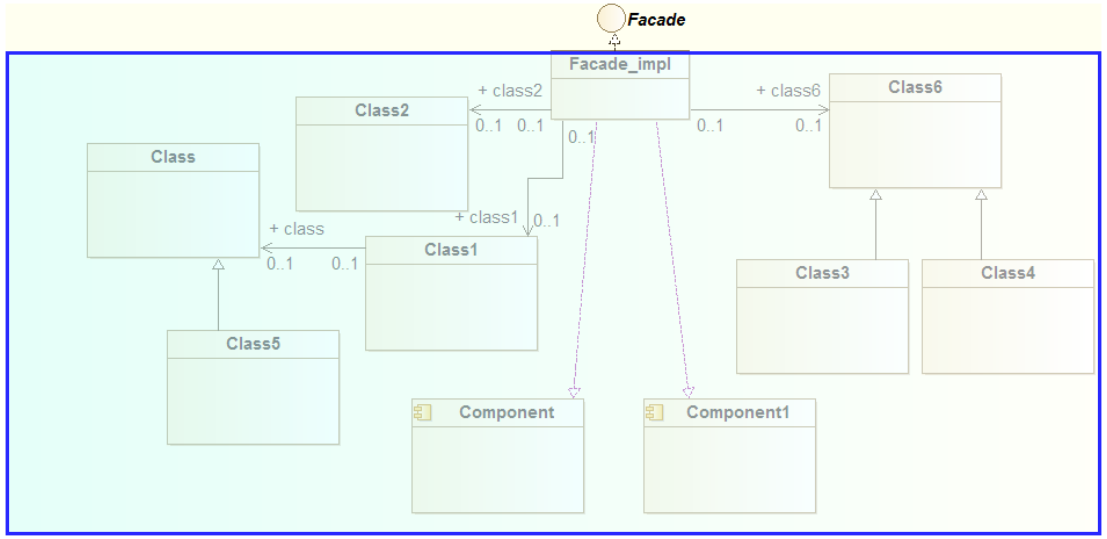
Ajout d'un ascenseur et d'une cadre au texte (mais pas à tous les textes)





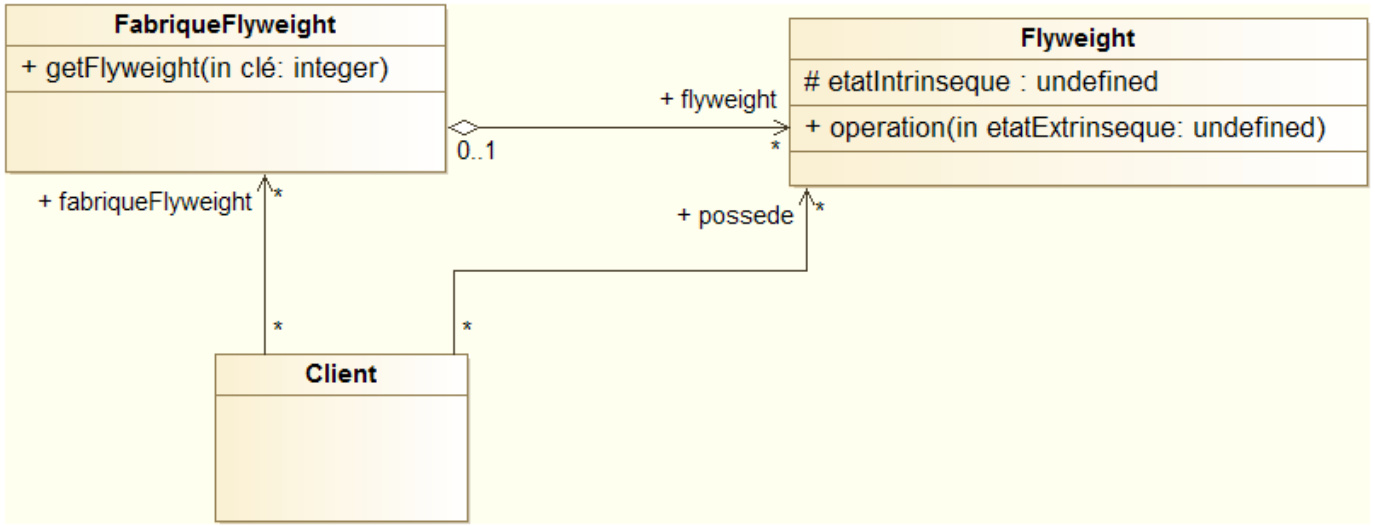
Pattern Façade

- Fournit une interface unifiée à l'ensemble des interfaces d'un sous-système. La façade fournit une « interface » de plus haut-niveau qui rend le sous-système plus facile à utiliser.



Pattern Flyweight

- Objectif : permettre le partage d'un ensemble important d'objet de grain fin (ou fine granularité).

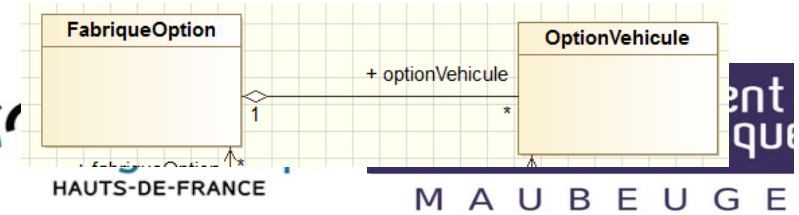


Pattern Flyweight

- Objectif : permettre le partage d'un ensemble important d'objet de grain fin (ou fine granularité).
- Exemple : Choix parmi de nombreuses options possible dans la configuration de matériel (voiture)

FabriqueOption.java | OptionVehicule.java | VehiculeCommande.java | Client.java

```java
import java.util.*;

public class FabriqueOption {

    protected Map<String, OptionVehicule> options = new TreeMap<Strin
    public OptionVehicule getOption( String nom){

            OptionVehicule resultat;
            if (options.containsKey(nom)){
                return options.get(nom);
        }
            else
        {
                resultat = new OptionVehicule(nom);
                options.put(nom,resultat);
        }
        return resultat;
    }
}
```

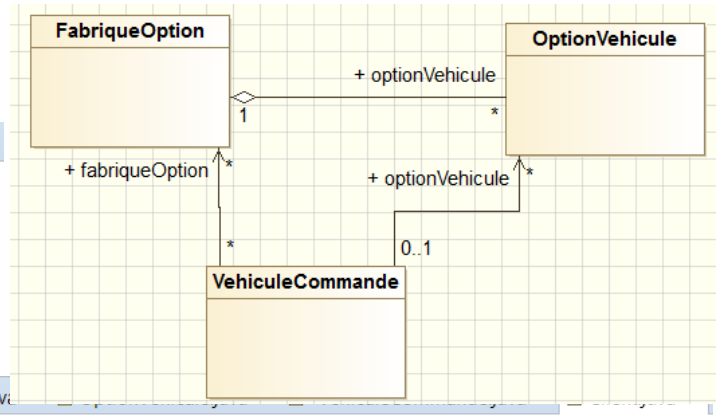


FabriqueOption.java | OptionVehicule.java | VehiculeCommande.java

```java

public class OptionVehicule {
protected String nom;
protected String description;
protected int prixStandard;

public OptionVehicule(String aNom){
    this.nom = aNom;
    this.description = "Description de " + aNom;
    this.prixStandard = 100;
}

public void affiche(int prixDeVente){
    System.out.println("Option");
    System.out.println("Nom : " + nom);
    System.out.println(description);
    System.out.println("prix standard : " + prixStandard);
    System.out.println("Prix de vente :" + prixDeVente);
}

}
```

Pattern Flyweight

- Exemple : Choix parmi de nombreuses options possible dans la configuration de matériel (voiture)



FabriqueOption.java | OptionVehicule.java | *VehiculeCommande.java | Client.java

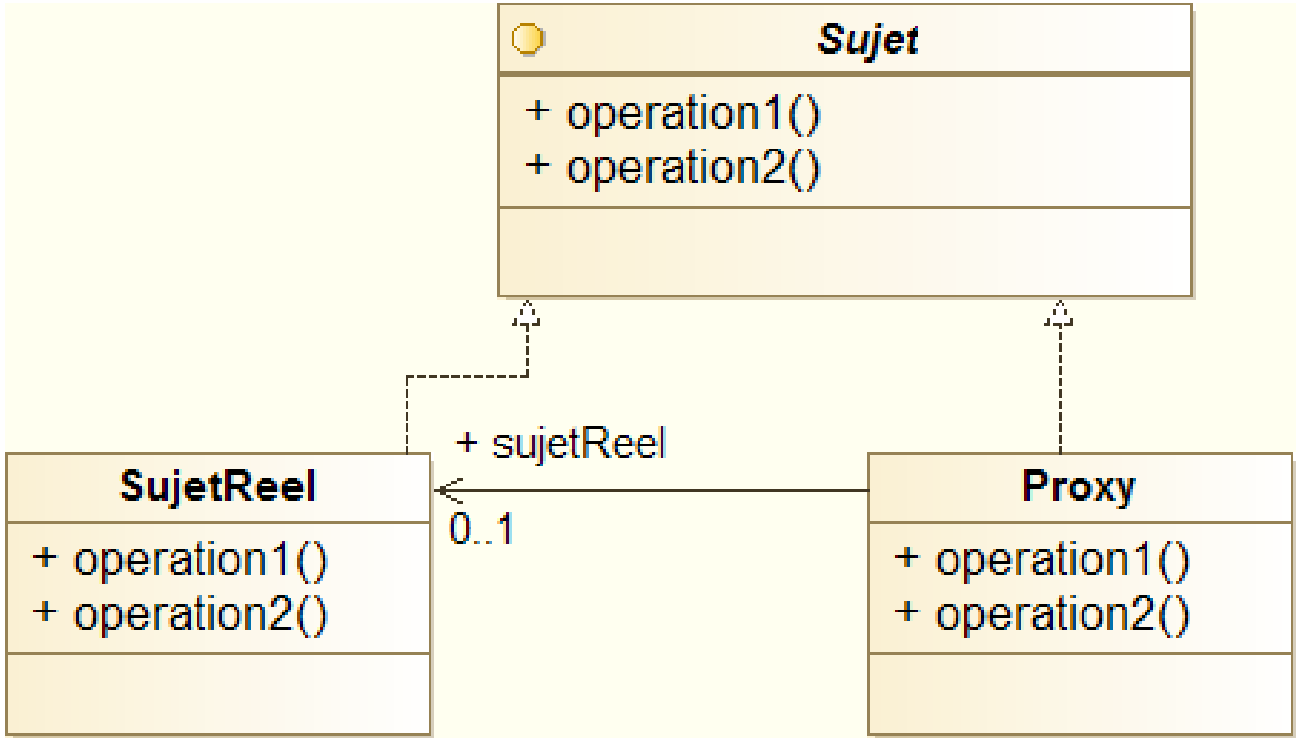
```java
import java.util.*;

public class VehiculeCommande {
protected ArrayList<OptionVehicule> options = new ArrayList<OptionVehicule>();
protected ArrayList<Integer> prixDeVenteOptions = new ArrayList<Integer>();

public void ajouteOptions(String nom, int prixDeVente, FabriqueOption fabrique){
    options.add(fabrique.getOption(nom));
    prixDeVenteOptions.add(prixDeVente);
}

public void afficheOptions()
{
    int index, taille;
    taille = options.size();
    for(index = 0;index < taille; index ++){
        options.get(index).affiche(prixDeVenteOptions.get(index));
        System.out.println();
    }
}

}
```

FabriqueOption.java

```java

public class Client {

    public static void main (String[] args)
    {
        FabriqueOption fabrique = new FabriqueOption();
        VehiculeCommande vehicule = new VehiculeCommande();
        vehicule.ajouteOptions("air bag",  80,  fabrique);
        vehicule.ajouteOptions("direction assistée", 90, fabrique);
        vehicule.ajouteOptions("vitres electriques", 85, fabrique);
        vehicule.afficheOptions();
    }

}
```

Pattern Proxy

- Objectif : concevoir un objet qui se substitue à un autre objet (le sujet) et qui en contrôle l'accès



Traitent les algorithmes et l'affectation des responsabilités

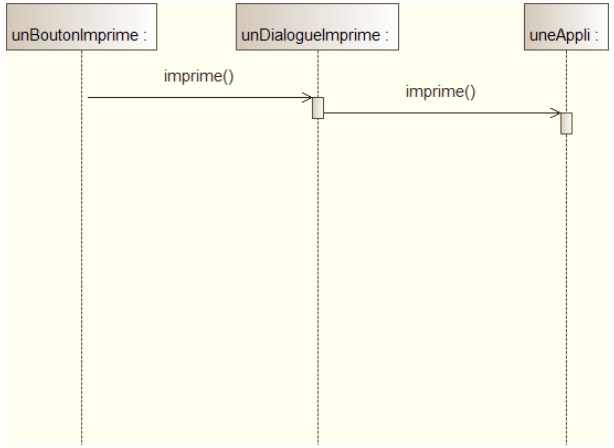
Décrivent Modèles de Classes MAIS AUSSI les communications (flux) entre les classes

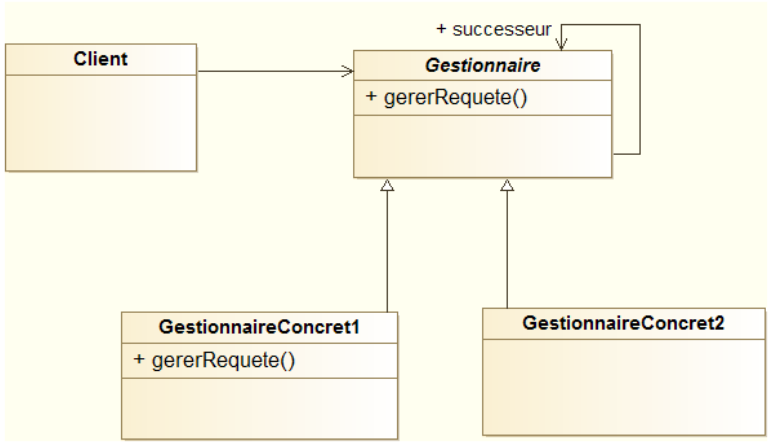
Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

PATTERNS GOF : PATTERNS DE COMPORTEMENT

Chain of responsability

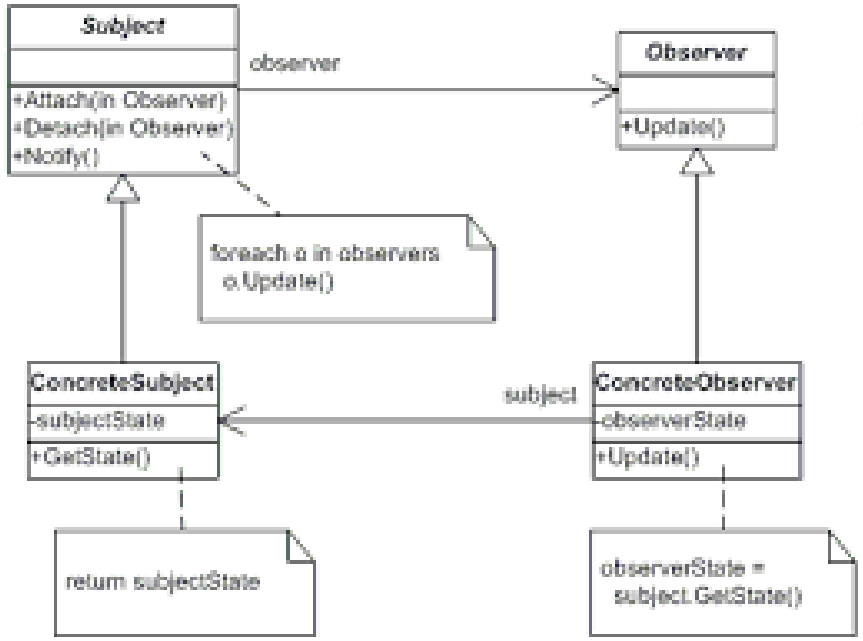
- Objectif : éviter le couplage entre l'émetteur d'une « requête » à ses récepteurs.
  - Plus d'un objet est en charge de « passer le message »
  - Chainage des objets jusqu'à ce qu'un objet traite la requête

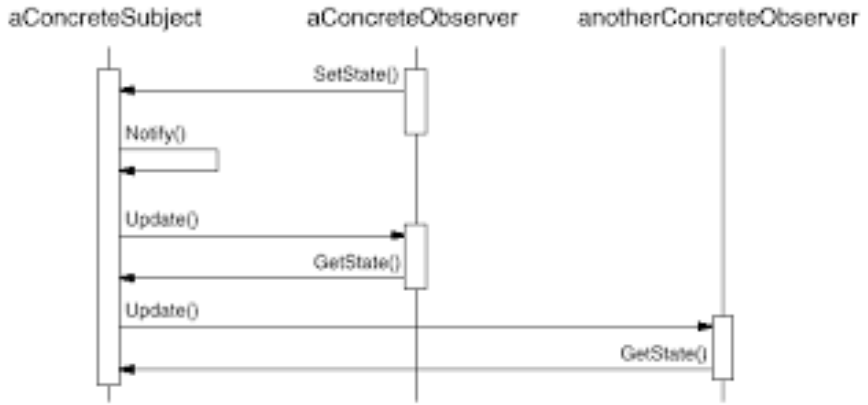




Pattern Observer

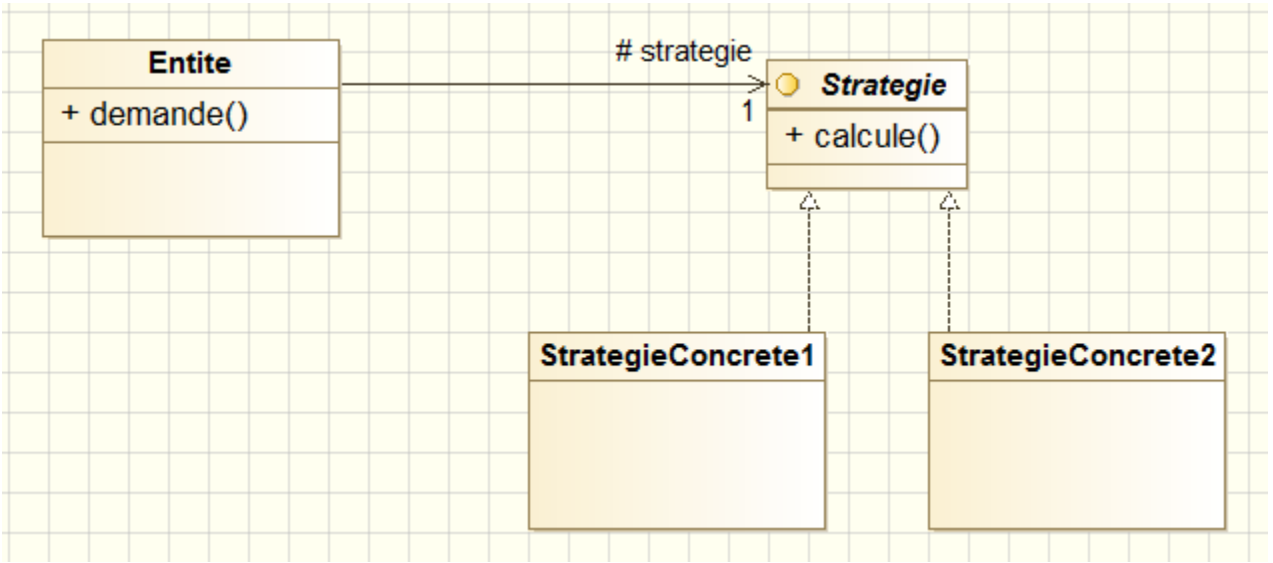
- Définit une interdépendance de type un à plusieurs, de façon telle que quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour





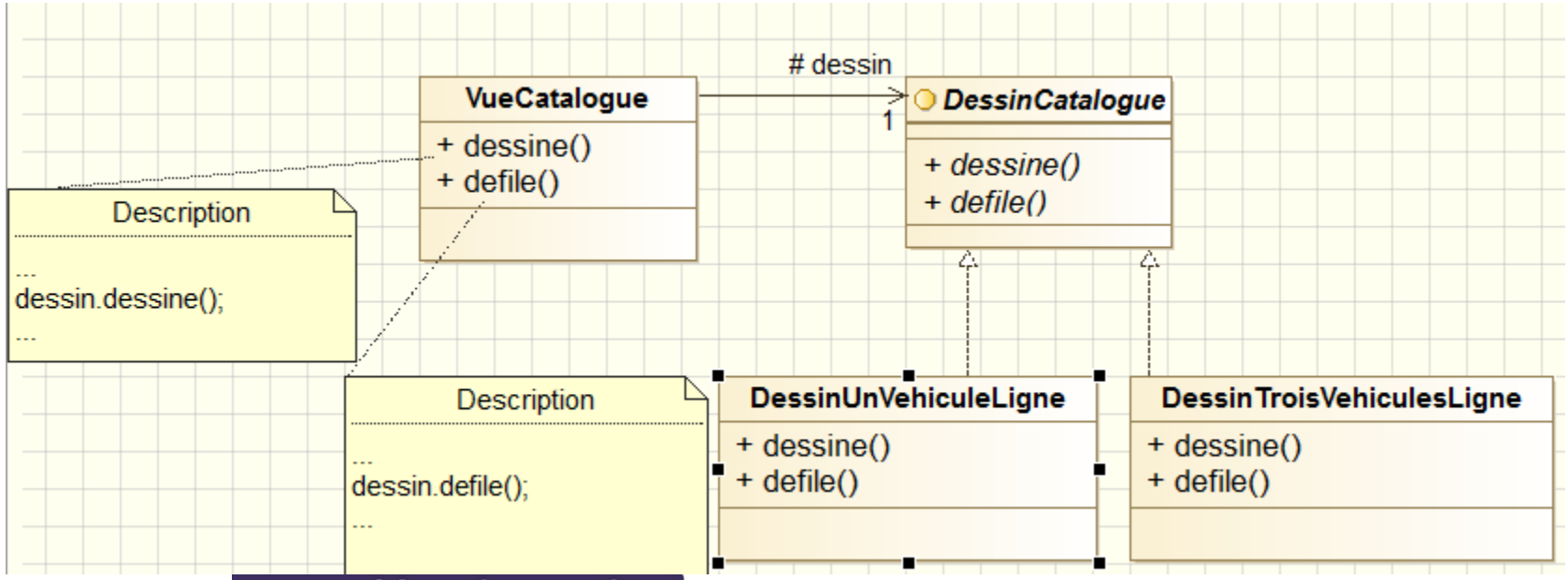
Strategy

- Définit une famille d'algorithme, encapsule chacun d'entre eux, et les rend interchangeables.
- Permet aux algos d'évoluer indépendamment des clients qui les utilisent



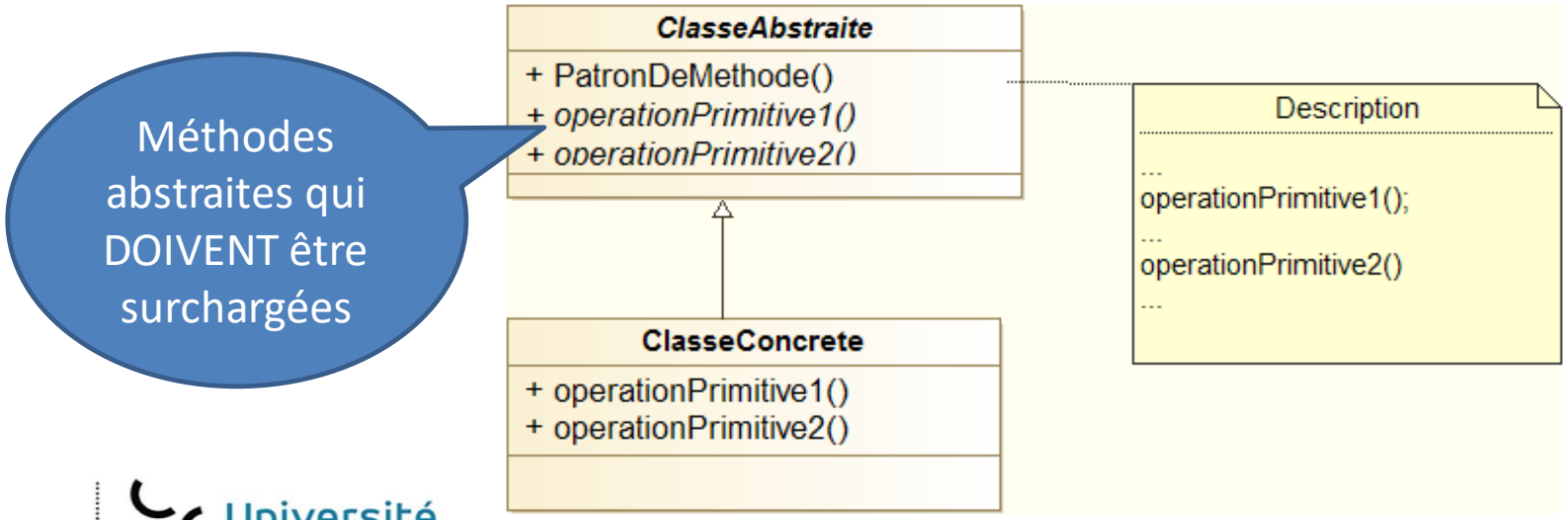
Strategy

- Définit une famille d'algorithme, encapsule chacun d'entre eux, et les rend interchangeables.
- Permet aux algos d'évoluer indépendamment des clients qui les utilisent



Template method

- Définit, dans une opération, le squelette d'un algorithme, en déléguant certaines étapes à des sous-classes.
- Objectif de bonne pratique :
  - sous-traitant, inversion de dépendance



Template method : exemple

- Gestion de circuit touristique
- Un voyage est composé :
  - un trajet aller
  - Activité en jour A
  - Activité en jour B
  - Activité en jour C
  - Un trajet retour

```
public class Trip {
    public final void performTrip(){
        doComingTransport();
        doDayA();
        doDayB();
        doDayC();
        doReturningTransport
    }
    public abstract void doComingTransport();
    public abstract void doDayA();
    public abstract void doDayB();
    public abstract void doDayC();
    public abstract void doReturningTransport();
}
```



```
public class TripA extends Trip {
    public void doComingTransport() {
        System.out.println("The turists are comming by air ...");
    }
    public void doDayA() {
        System.out.println("The turists are visiting the aquarium ...");
    }
    public void doDayB() {
        System.out.println("The turists are going to the beach ...");
    }
    public void doDayC() {
        System.out.println("The turists are going to mountains ...");
    }
    public void doReturningTransport() {
        System.out.println("The turists are going home by air ...");
    }
}
```

```
public class PackageB extends Trip {
    public void doComingTransport() {
        System.out.println("The turists are comming by train ...");
    }
    public void doDayA() {
        System.out.println("The turists are visiting the mountain ...");
    }
    public void doDayB() {
        System.out.println("The turists are going to the beach ...");
    }
    public void doDayC() {
        System.out.println("The turists are going to zoo ...");
    }
    public void doReturningTransport() {
        System.out.println("The turists are going home by train ...");
    }
}
```

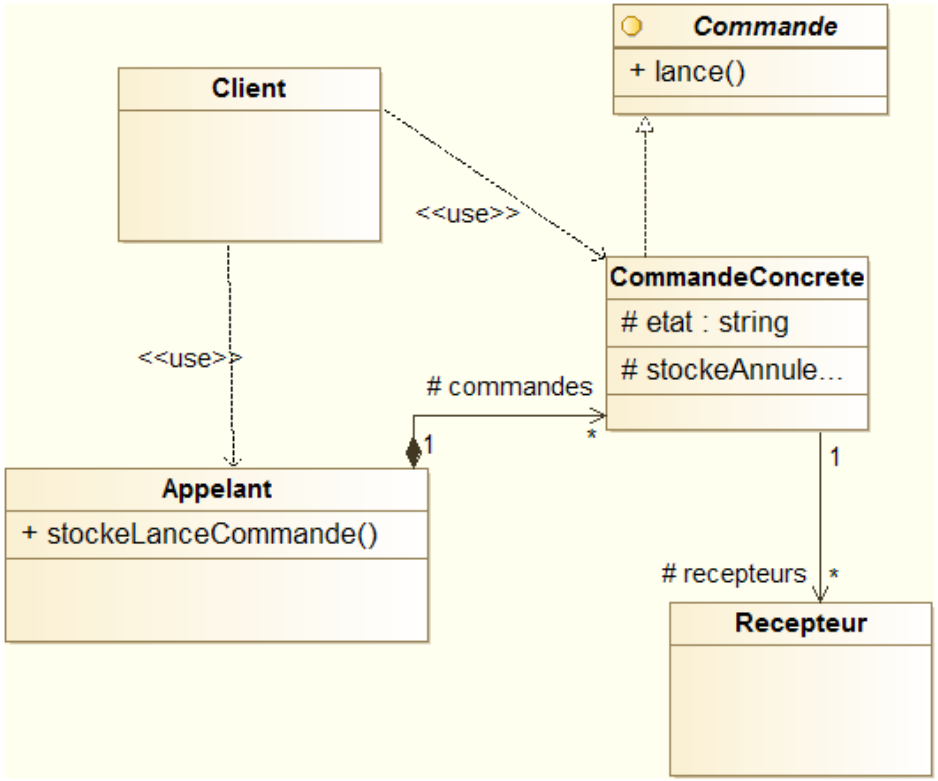
Source : http://www.oodesign.com/template-method-pattern.html

Pattern Command

- Objectif : encapsuler une requête comme un objet autorisant le paramétrage des clients par différentes requêtes et permettant la réversion des opérations
  - Classe commande abstraite, qui déclare une interface pour l'exécution des opérations.
  - La réversion
    - est utile par exemple pour les transactions (base de données), pour gérer le « Undo », pour gérer les paniers qui expirent…
    - Un objet pile (Appelant) gère la succession des appels

Pattern Command

- Sépare l'objet qui « demande » (Client) de celui qui exécute (Appelant).



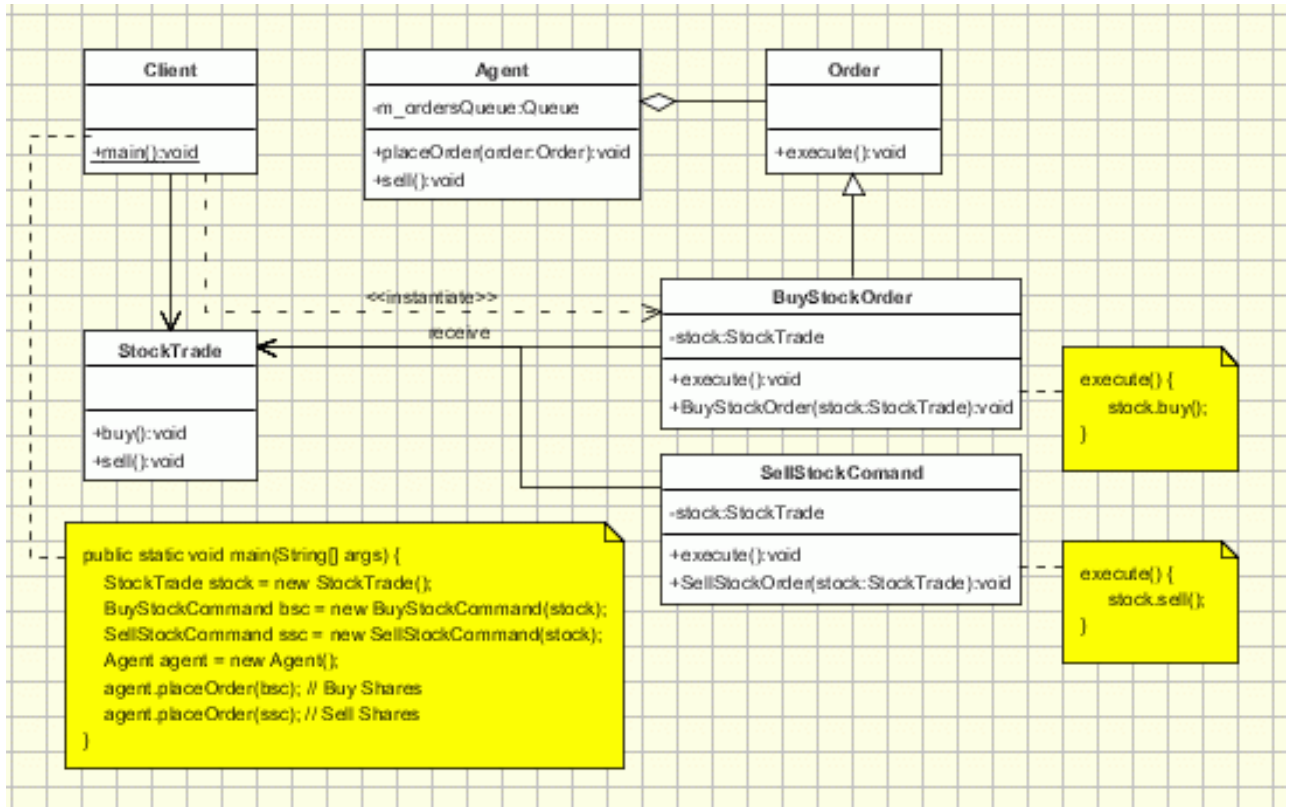
Un exemple avec un catalogue de Voiture [Debrauwer, 2009]

```java
public class Catalogue {
    protected List<Vehicule> vehiculesStock = new ArrayList<Vehicule>();
    protected List<CommandeSolder> commandes = new ArrayList<CommandeSolder>();

    public void LanceCommandeSolder (CommandeSolder commande){
        commandes.add(0,commande);
        commande.solde(vehiculesStock);
        }
public void annuleCommandeSolder(int ordre){
    commandes.get(ordre).annule();
    }
public void retablitCommandeSolder(int ordre){
    commandes.get(ordre).retablit();

}
public void ajoute (Vehicule vehicule)
{
    vehiculesStock.add(vehicule);
}
public void affiche(){
    for (Vehicule vehicule : vehiculesStock)
        vehicule.affiche();
}
}
```

```java
public interface Commande <T> {

    public void solde(List<T> asolder);
    public void annule();
    public void retablit();
}
```

```java
public class CommandeSolder implements Commande<Vehicule>{
protected List<Vehicule> vehiculesSoldes = new ArrayList<Vehicule>();
protected long aujourdhui;
protected long dureeStock;
protected double tauxRemise;

public CommandeSolder (long aujourdhui, long dureeStock, double tauxRemise){
    this.aujourdhui = aujourdhui;
    this.dureeStock = dureeStock;
    this.tauxRemise = tauxRemise;
}

    public void solde(List<Vehicule> vehicules){
        vehiculesSoldes.clear();
        for (Vehicule vehicule: vehicules)
            if (vehicule.getDureeStockage(aujourdhui) >= dureeStock)
                    vehiculesSoldes.add(vehicule);
            for(Vehicule vehicule : vehiculesSoldes)
                vehicule.modifierPrix(1.0 - tauxRemise);
        }
    public void annule()
    { for (Vehicule vehicule: vehiculesSoldes)
            vehicule.modifierPrix(1.0 - tauxRemise);
    }
    public void retablit(){
        for (Vehicule vehicule: vehiculesSoldes)
        vehicule.modifierPrix(1.0 - tauxRemise);
    }
}
```

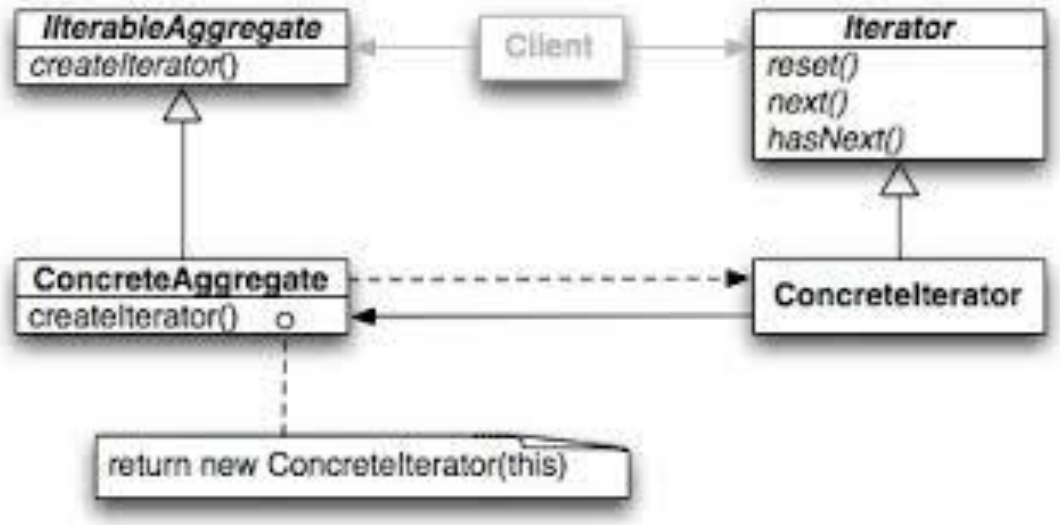
Exemple de Pattern Command or Producer-Consumer



http://www.oodesign.com/command-pattern.html

Pattern Iterator

- Fournit un moyen d'accès séquentiel aux éléments d'un agrégat d'objets sans mettre à découvert la représentation interne de celui-ci.



Iterator en java

java.util

Interface Iterator<E>

Type Parameters:

`E` - the type of elements returned by this iterator

All Known Subinterfaces:

ListIterator<E>, XMLEventReader

All Known Implementing Classes:

BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner

```
public interface Iterator<E>
```

An iterator over a collection. `Iterator` takes the place of `Enumeration` in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `ListIterator`, `Iterable`

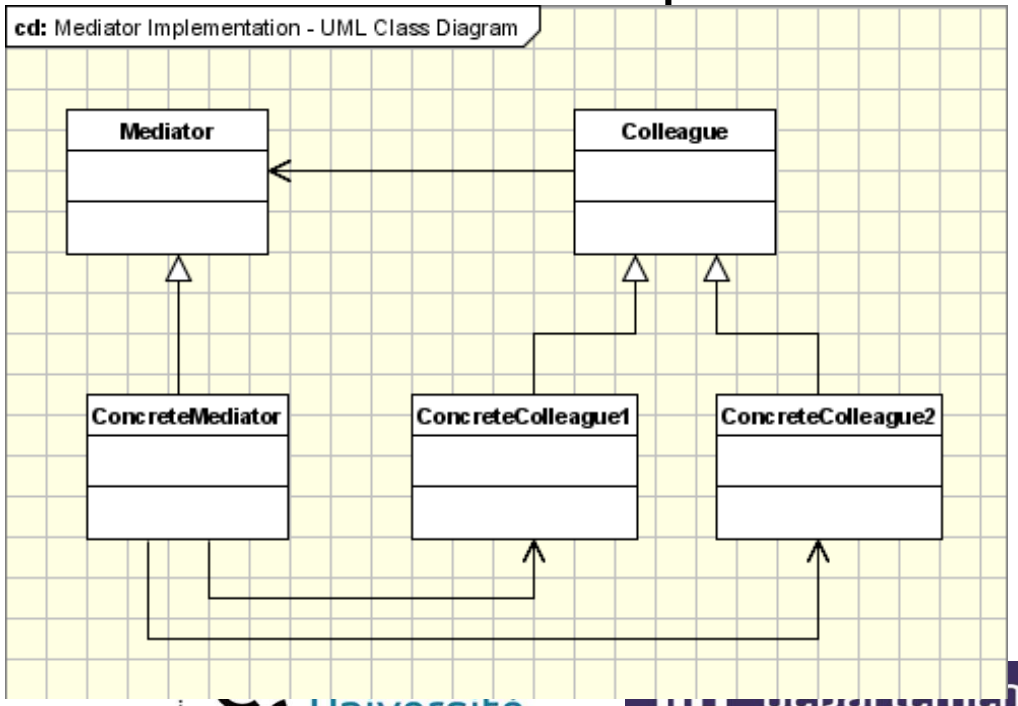
Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`void`	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

Pattern Mediator

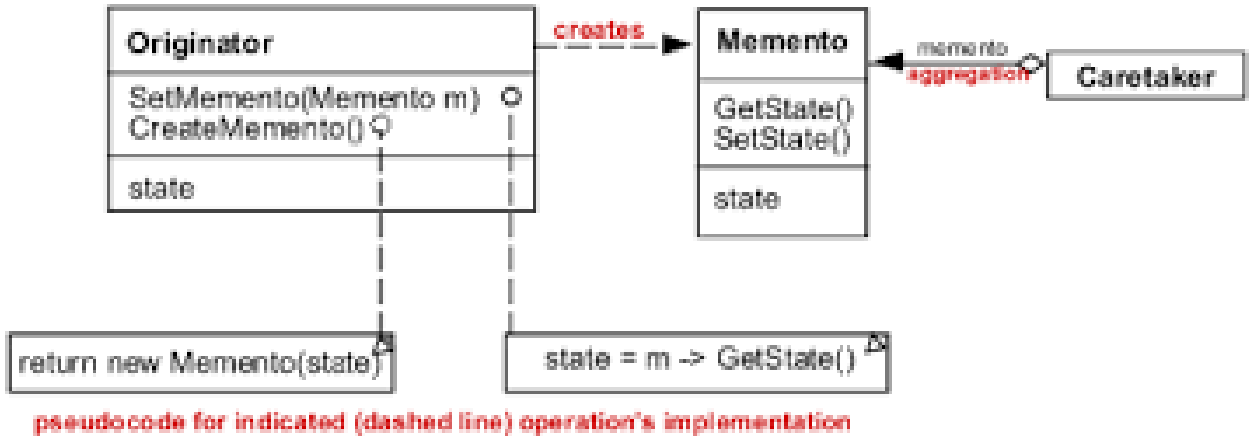
Objectif : définit un objet qui encapsule les modalités d'interaction d'un certain ensemble d'objets. Le médiateur favorise le couplage faible en dispensant les objets de se faire explicitement référence, et il permet donc de faire varier indépendamment les relations d'interaction



- MediateurConcret :
  - Réalise le comportement coopératif en coordonnant les objets collègues
  - Connait et gère ses collègues

Pattern Memento

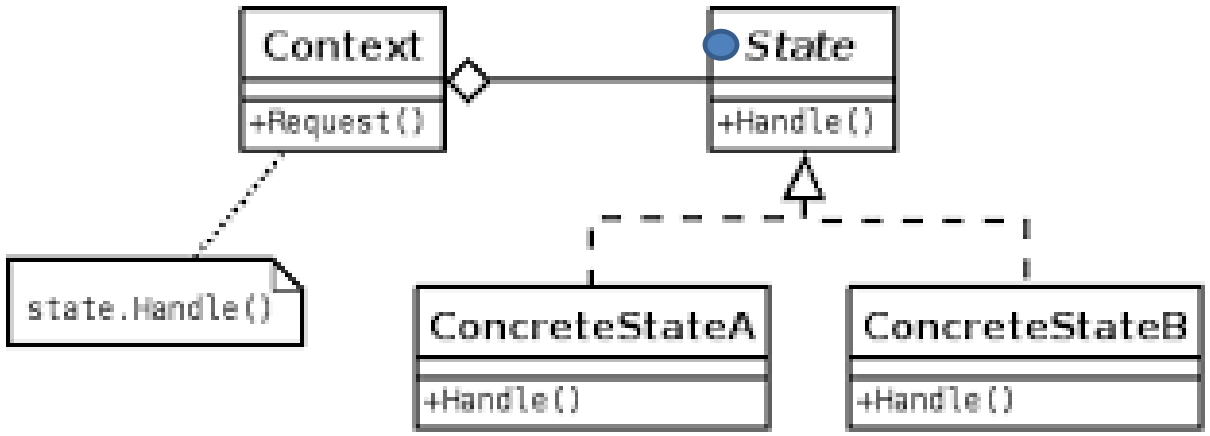
Objectif : saisir et transmettre à l'extérieur d'un objet l'état interne de celui-ci (sans violation de l'encapsulation), dans le but de pouvoir ultérieurement le restaurer dans cet état.



- Les mementos sont passifs, seuls le créateur « originator » affectera et récupérera son état.
- Le surveillant est responsable de la sauvegarde ; il n'agit pas sur le memento

Pattern State

Permet à un objet de modifier son comportement, quand son état interne change. Tout se passera comme si l'objet changeait de classe.

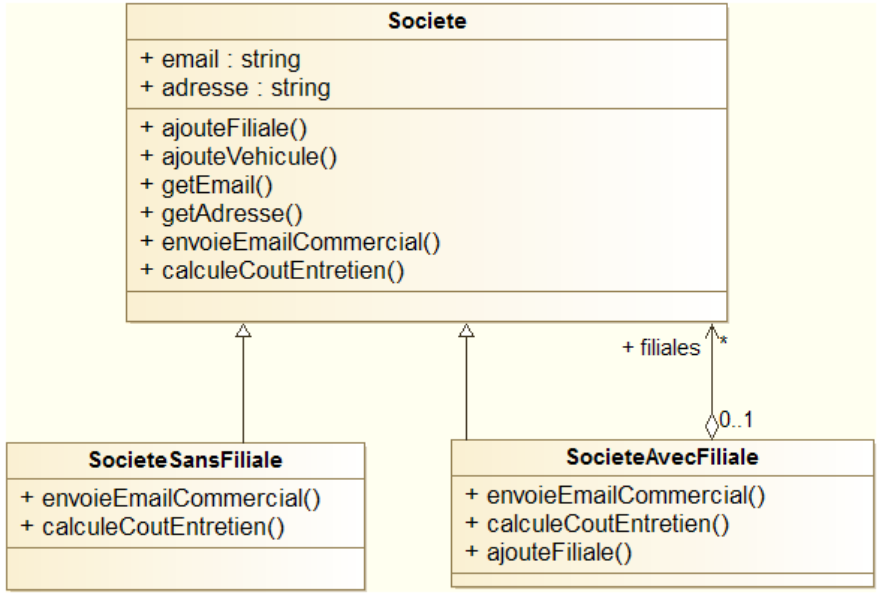


- Utile quand
  - Comportement d'un objet dépend de son état. Ce changement de comportement doit intervenir dynamiquement , en fonction de cet état

Visitor

- Fait la représentation d'une opération applicable aux éléments d'une structure d'objet. Il permet de définir une nouvelle opération, sans qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit.

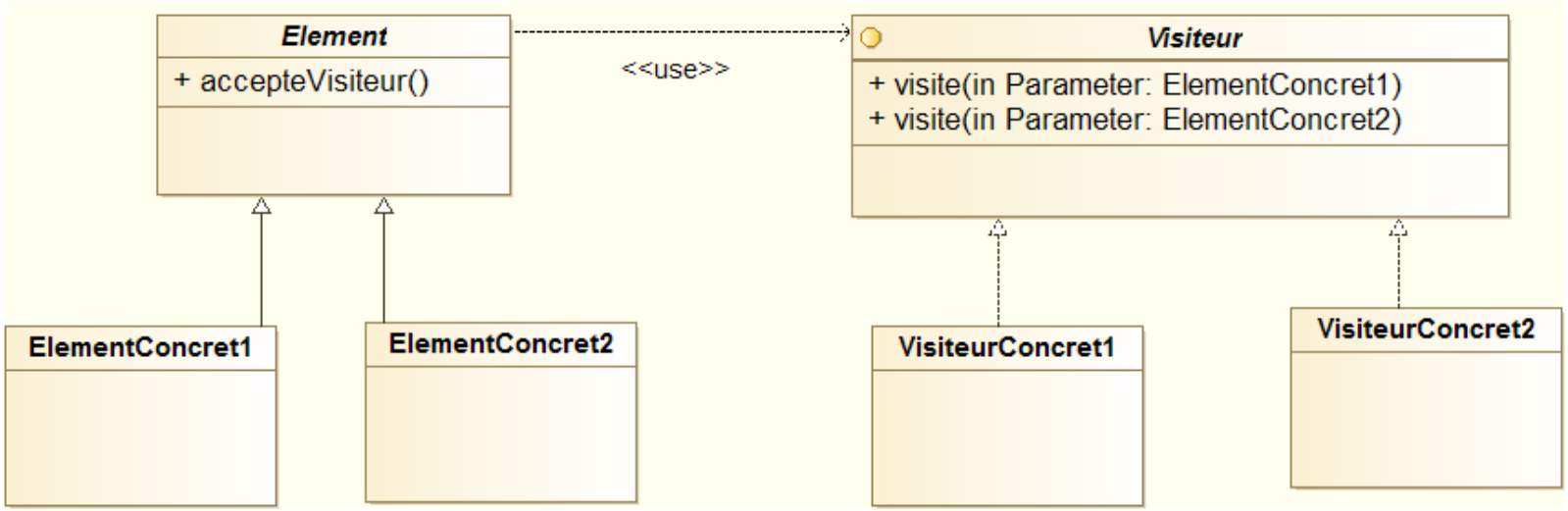
Conception sans Visitor



- Problème ?
  - Modèle valide si peu de méthodes (fonctionnalités)
  - Sinon beaucoup de méthodes
    - complexifie la compréhension de l'objet
    - Pas de relation entre les méthodes.
    - Manque de sens avec le « cœur » de l'objet

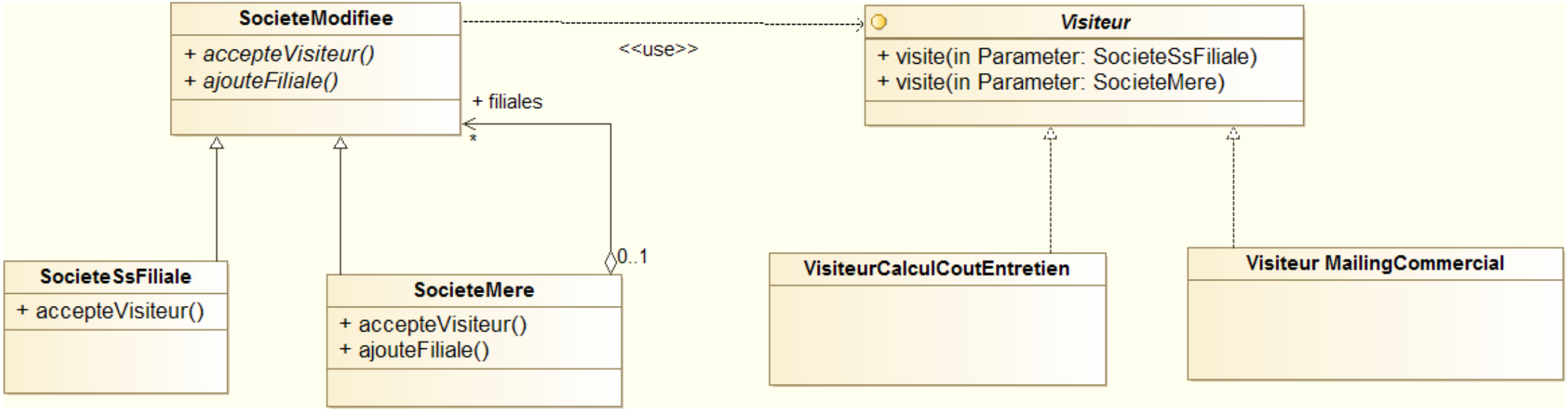
Pattern Visitor

- Objectif : externaliser les méthodes de l'objet qui n'ont pas de sens



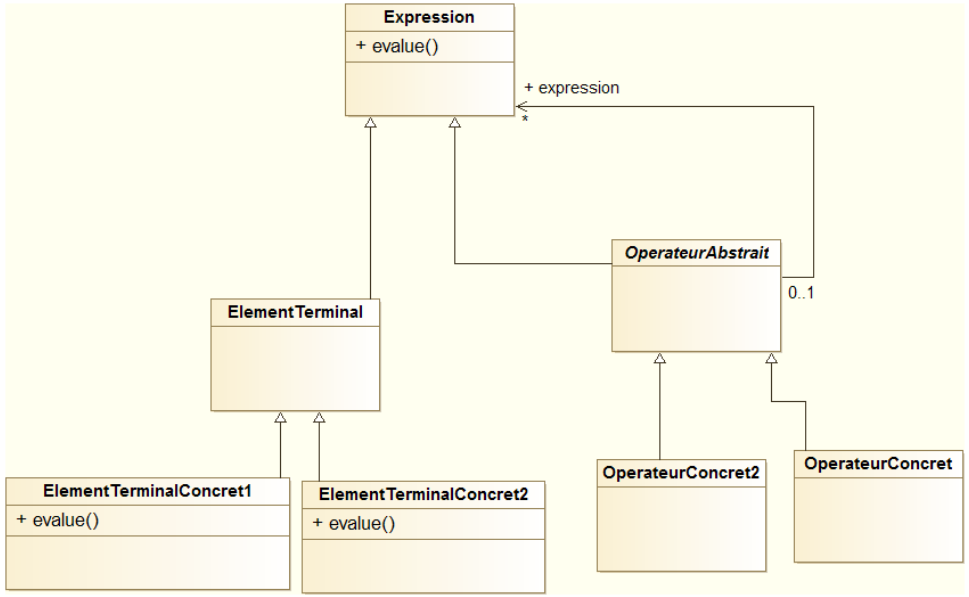
Exemple avec Visitor

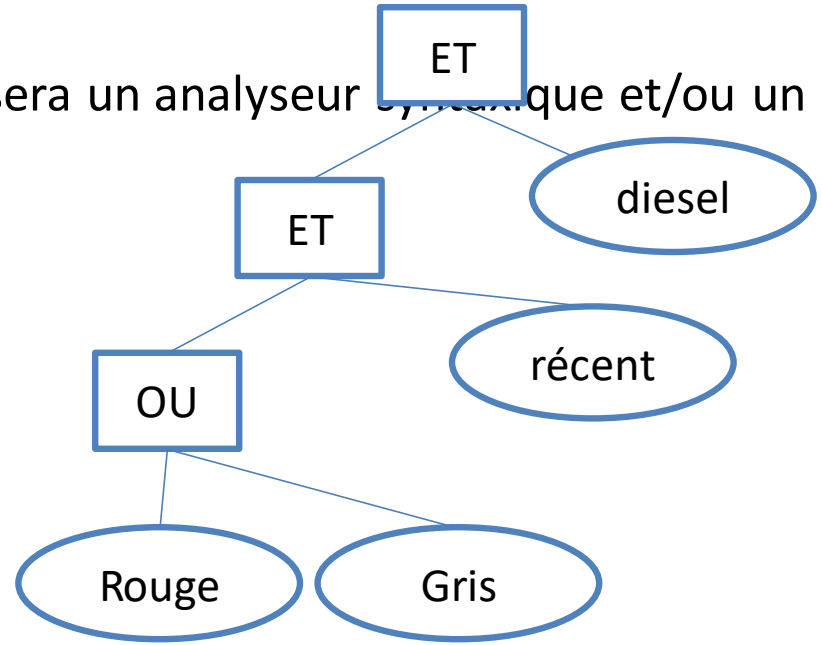
- Avec cette solution :
  - Les attributs et méthodes de Société sont bien en lien avec l'objet
  - Le Visiteur permet d'externaliser
  - Facilite l'ajout de méthodes (par de nouveaux visiteurs) et facilite la maintenance



Interpreter

- L'objectif est de composer des objets exécutables d'après un ensemble de règles de composition que vous définissez.
- Peut être utilisé pour définir des grammaires et faire des évaluations d'expression syntaxique
- Peu utilisé selon le site oodesign.com
  - Dans le cas d'une grammaire complexe on favorisera un analyseur syntaxique et/ou un compilateur.





Bibliographie

- Gamma, E., Helm R., Johnson R., Vlissides J. (1995) Design Patterns, catalogue de modèles de conception réutilisables, international thomson publishing France.
- Debrauwer L. 2007. Design Patterns Les 23 modèles de conception et solutions illustrées en UML2 et Java.edition ENI. ISBN 978-2-7460-3887-5.
- Debrauwer L. & Karam N. 2010. Design Patterns. Mise en Œuvre des modèles de conception en java. Edition ENI. ISBN 978-2-7460-5882-8
- Roques, P., Vallée, F. (2009), UML 2 en action, de l'analyse des besoins à la conception, 4ème edition, Eyrolles.

Autres pointeurs

- http://www.emse.fr/~boissier/enseignement/aco/pdf/2.DesignPattern.MSGL.4pp.pdf
- http://www.u-picardie.fr/~furst/docs/Patterns.pdf
- http://www.oodesign.com/
- http://www.eyrolles.com/Chapitres/9782212111392/Shalloway_Chap_7.pdf

- Plein d'exemples sur GitHub…