
# Génie Logiciel et projet de synthèse

## Design Patterns

Kévin Bailly
Institut des Systèmes Intelligents et de Robotique

kevin.bailly@upmc.fr

# Bibliographie

UPMC SORBONNE UNIVERSITÉS

- **Design Patterns: Elements of Reusable Object-Oriented Software,** E. Gamma, R. Helm, R. Johnson, J. Vlissides

- **Les Design Patterns en Java - Les 23 modèles de conception fondamentaux,** S. J. Metsker et W. C. Wake

- **Cours LI314 : Programmation par objets**

# Design Patterns

UPMC SORBONNE UNIVERSITÉS

- Elément de **solution orienté-objet réutilisable** à des **problèmes récurrents**
- Pas de l'algorithmique : réponse à un **problème de conception**
  - Flexibilité
  - Maintenabilité
  - Configurabilité
  - Robustesse…
- Décrit sous forme de **diagrammes** (statique et dynamique) un arrangement récurrent de **rôles** et **d'actions** joués par des **modules** d'un logiciel
- Issus de **l'expérience** des concepteurs de logiciels
- **Indépendant** des langages de programmation

# Design Patterns

UPMC SORBONNE UNIVERSITÉS

- Le GoF (Gang of Four) E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995). **Design Patterns: Elements of Reusable Object Oriented Oriented Software.**

- Définit **23 patterns**, dont la majorité sont devenus des standards

- Influence sur les langages développés après 1995. Par ex : librairie standard de Java

# Design Patterns

UPMC SORBONNE UNIVERSITÉS

- Un design pattern est défini par :
  - Son **nom**
  - Le **problème** : QUAND utiliser le DP
  - La **solution :** décrit l'agencement des classes et des objets qui permet de répondre au problème de conception
  - Les **conséquences** de l'utilisation de ce DP (en flexibilité, portabilité, espace mémoire, temps de calcul…)

# Principes communs aux DP

UPMC SORBONNE UNIVERSITÉS

- **Principe 1** : Favoriser la **composition** (lien dynamique, flexible) sur **l'héritage** (lien statique, peu flexible)
  - **Attention**: favoriser ne veut pas dire remplacer systématiquement, l'héritage est largement utilisé aussi

- **Principe 2** : Les clients programment en priorité pour des **abstractions** (interfaces, classes abstraites…) plutôt qu'en lien direct avec les **implémentations** (classes concrètes)

- **Principe 3 :** Privilégier une **encapsulation forte** pour permettre la substituabilité

# 3 types de Design Patterns

UPMC SORBONNE UNIVERSITÉS

- **Patterns de création/construction**
  - S'intéressent à la construction des objets (choix de la classe responsable des créations, choix du type créé)
  - Patterns Builder, FactoryMethod, AbstractFactory, Singleton

- **Patterns structuraux**
  - Liés au problèmes d'organisation des objets dans un logiciel (aspects statiques)
  - Composition des classes et des objets
  - Façade, Adapter, Decorator, Proxy, Composite

- **Patrons comportementaux (behavioral)**
  - liés au problèmes de communication entre les objets (aspects dynamiques)
  - Strategy, Iterator, Observer, Visitor

# Portée des DP (scope)

UPMC SORBONNE UNIVERSITÉS

- Portée de Classe
  - Focalisation sur les relations entre classes et leurs sous-classes
  - Réutilisation par héritage
- Portée d'Instance
  - Focalisation sur les relations entre les objets
  - Réutilisation par composition

# Les Design Patterns du GoF

UPMC SORBONNE UNIVERSITÉS

| | | Catégorie | | |
|---|---|---|---|---|
| | | **Création** | **Structure** | **Comportement** |
| **Portée** | **Classe** | Factory Method | Adapter | Interpreter |
| | | | | Template Method |
| | **Objet** | Abstract Factory | Adapter | Chain of Responsibility |
| | | Builder | Bridge | Command |
| | | Prototype | Composite | Iterator |
| | | Singleton | Decorator | Mediator |
| | | | Facade | Memento |
| | | | Flyweight | Observer |
| | | | Proxy | State |
| | | | | Strategy |
| | | | | Visitor |

# Les Design Patterns du GoF

UPMC SORBONNE UNIVERSITÉS

| | | Catégorie | | |
|---|---|---|---|---|
| | | **Création** | **Structure** | **Comportement** |
| **Portée** | **Classe** | Factory Method | Adapter | Interpreter |
| | | | | Template Method |
| | **Objet** | Abstract Factory | Adapter | Chain of Responsibility |
| | | Builder | Bridge | Command |
| | | Prototype | Composite | Iterator |
| | | Singleton | Decorator | Mediator |
| | | | Facade | Memento |
| | | | Flyweight | Observer |
| | | | Proxy | State |
| | | | | Strategy |
| | | | | Visitor |

UPMC
SORBONNE UNIVERSITÉS

Les Design Patterns

# DESIGN PATTERNS DE CONSTRUCTION

# Patterns de construction

- Définissent des mécanismes pour l'instanciation et/ou la l'initialisation d'objets

- Habituellement : utilisation d'un **constructeur** (opérateur new), mais :
  - Le client doit connaître la classe à instancier
  - Doit connaître les paramètres que le constructeur attend

- Pas toujours si facile, pourquoi ?

# Patterns de construction

- Définissent des mécanismes pour l'instanciation et/ou la l'initialisation d'objets

- Habituellement : utilisation d'un **constructeur** (opérateur new), mais :
  - Le client doit connaître la classe à instancier
  - Doit connaître les paramètres que le constructeur attend

- Pas toujours si facile, par exemple :
  - Un élément d'une interface graphique (ex: boutton) va **dépendre du matériel cible** (Smartphone ou grand écran)
  - Informations (valeurs initiale) **disponible uniquement au runtime** (fichier de configuration par exemple)
  - Constructeurs pas suffisant ➔ utilisation d'un DP

# FACTORY METHOD

UPMC SORBONNE UNIVERSITÉS

- **Objectif** :
  - Permet d'instancier des objets dont le type est dérivé d'un type abstrait.
  - La classe exacte de l'objet n'est donc pas connue par l'appelant.

- **Exemple** : les Itérateurs
  - L'interface `Collection` inclut une méthode `iterator()` ➔ pas besoin de connaître la classe à instancier !

# FACTORY METHOD

UPMC SORBONNE UNIVERSITÉS

```
package app.factoryMethod;
import java.util.*;

public class ShowIterator {
    public static void main(String[] args) {
      List<String> list= Arrays.asList({"riri",
                                "fifi", "loulou"});
      Iterator iter = list.iterator();
      while (iter.hasNext())
          System.out.println(iter.next());
    }
}
```

Quelle est la classe réelle de l'objet Iterator ?

# FACTORY METHOD

UPMC SORBONNE UNIVERSITÉS

```
package app.factoryMethod;
import java.util.*;

public class ShowIterator {
    public static void main(String[] args) {
      List<String> list= Arrays.asList({”riri”,
                                ”fifi”, “loulou”});
      Iterator iter = list.iterator();
      while (iter.hasNext())
          System.out.println(iter.next());
    }
}
```

Quelle est la classe réelle de l’objet Iterator ?

PEU IMPORTE !! Ce qui compte c’est l’interface

# FACTORY METHOD

UPMC SORBONNE UNIVERSITÉS

- D'autres exemples en Java :
  - toString()
  - clone()

- Autre exemple :
  - Sauvegarde dans un flux sortant : il peut s'agir du fichier ou d'une sortie sur le réseau
  - La classe possède une méthode qui retourne une instance d'objet qui implémente une interface commune au fichier et au réseau
  - Permet d'effectuer des opérations sur cet objet (écriture, fermeture) de manière transparente

# FACTORY METHOD

UPMC SORBONNE UNIVERSITÉS

- Dans une conception Factory Method, il peut y avoir :
  - **plusieurs classes** qui implémentent la **même opération**
  - qui retourne le même **type abstrait**
  - Mais la **classe instanciée dépend de l'objet factory qui recoit la requette**

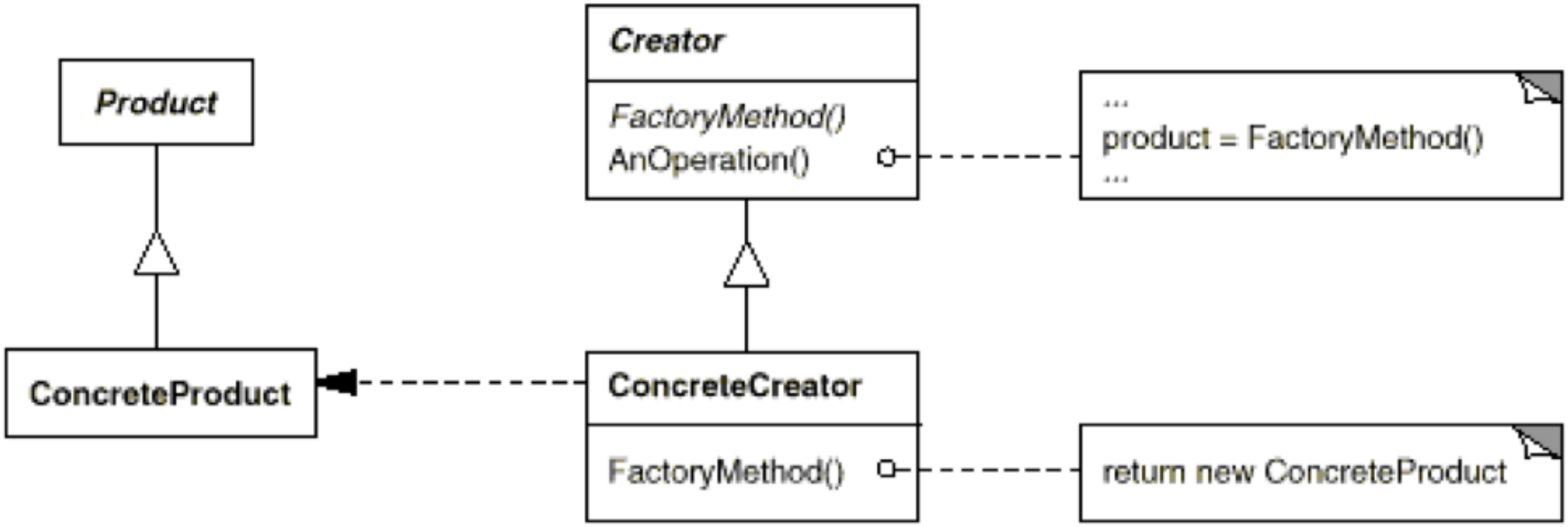
# Exercice

UPMC SORBONNE UNIVERSITÉS

- Représentation UML du Design Pattern pour un Iterateur sur une Collection en Java

# Un peu d'aide

- Représentation générique du DP Factory Method



# FABRIQUE ABSTRAITE

UPMC SORBONNE UNIVERSITÉS

- Encapsule un groupe de fabriques ayant une thématique commune

- Un client qui utilise une Fabrique :
  - Crée une instance concrète de la Fabrique
  - Utilise l'interface pour créer des objets concrets
  - Ne manipule ces objets concrets qu'au travers des interfaces

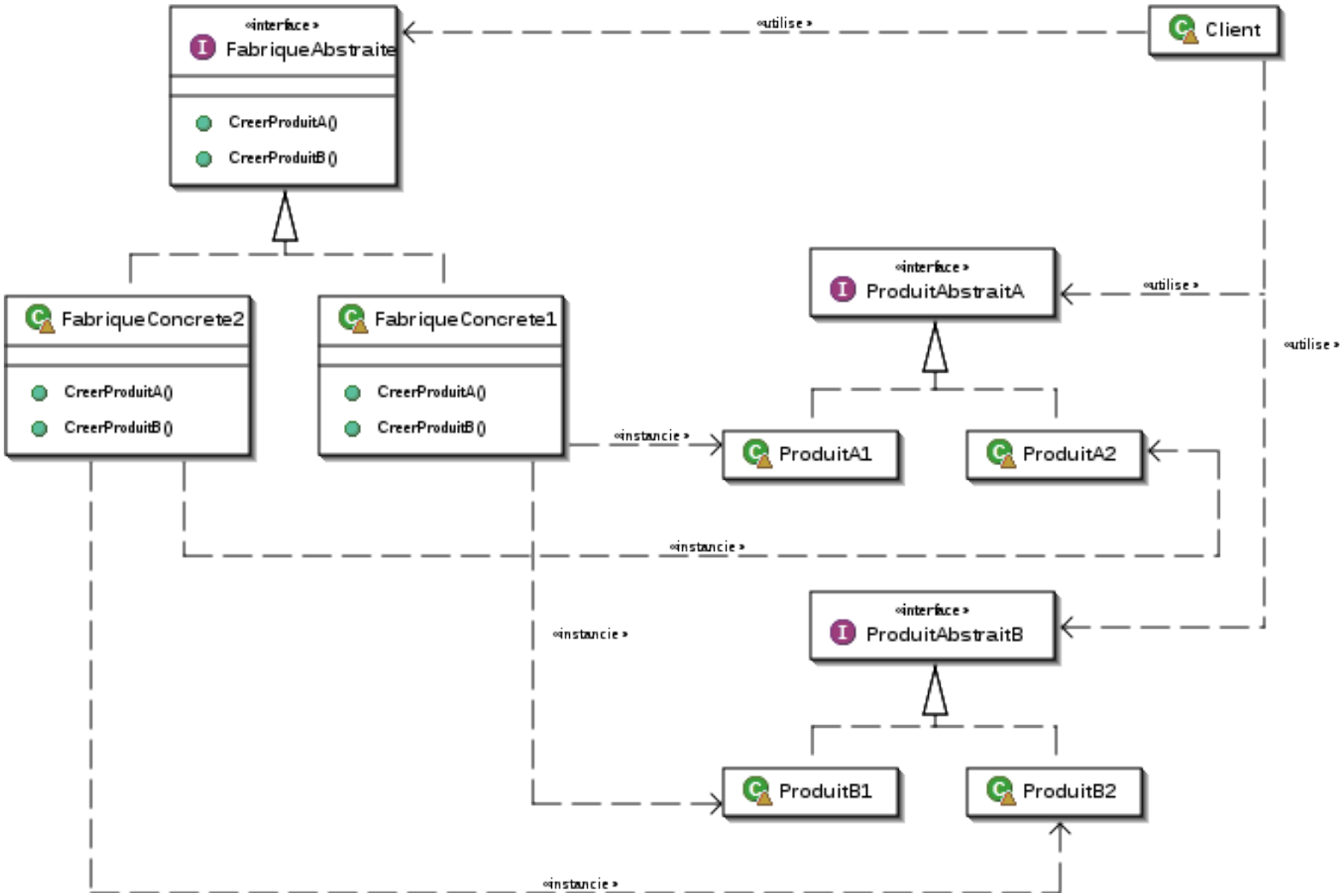
# Exemple 1

- Une classe ***`DocumentCreator`*** fournit une interface permettant de créer différents produits (e.g. **`createLetter`**() et **`createResume`**()
- Le système a des versions concrètes dérivées de la classe ***`DocumentCreator`***, comme **`ClassicDocumentCreator`** et **`ModernDocumentCreator`**
- Tous les fils de ***`DocumentCreator`*** ont une implémentation spécifique de **`createLetter`**() et **`createResume`**()
- Ces méthodes créent des objet concrets (**`ModernLetter`** ou **`ClassicResume`**) par exemple
- Le Client ne connaît que la classe abstraite de ces objets : Letter et Resume

# Exercice

- Dessiner le diagramme de classe de l'énoncé précédent

# Diagramme de classe général



# Exemple 2 : en Java

```
public abstract class GUIFactory{
   public static GUIFactory getFactory(){
    int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0)
            return(new WinFactory());
        return(new OSXFactory());
   }
   public abstract Button createButton();
}

class WinFactory extends GUIFactory {
    public Button createButton() {
        return(new WinButton());
    }
}

class OSXFactory extends GUIFactory {
    public Button createButton()
    {
        return(new OSXButton());
    }
}
```

```
public abstract class Button {
     private String caption;

     public String getCaption(){
         return caption;
     }

     public void setCaption(String caption){
         this.caption = caption;
     }

     public abstract void paint();
 }

 class WinButton extends Button {
     public void paint(){
            System.out.println("WinButton:  "+
getCaption());
     }
 }

 class OSXButton extends Button {
     public void paint() {
             System.out.println("OSXButton:  "+
getCaption());
     }
 }
```

Que fait ce code ?

# Exemple 2 : en Java

UPMC SORBONNE UNIVERSITÉS

Coté client

```
public class Application {
     public static void main(String[] args) {
         GUIFactory aFactory = GUIFactory.getFactory();
         Button aButton = aFactory.createButton();
         aButton.setCaption("Play");
         aButton.paint();
     }

     //output is
     //WinButton: Play
     //or
     //OSXButton: Play
 }
```

# Exemple 2 : autre solution

```
public class GUIFactory {
    public final static GUIFactory instance = null;

    public static GUIFactory getFactory() {
        if(instance == null)
            instance = new GUIFactory();
        return instance;
    }

    public Button createButton() {
        String productName = readFromConfigFile("BUTTON_TYPE");
        return (Button)Class.forName(productName).newInstance();
    }
}
```

Dans ce cas, la fabrique est complètement **indépendante** du produit concrêt qu'elle crée

http://www.adam-bien.com/roller/abien/entry/is_abstract_factory_vs_factory

# Factory Method vs Abstract Factory

UPMC SORBONNE UNIVERSITÉS

- Quelles sont les différences entre ces deux Design Patterns ?

# Factory Method vs Abstract Factory

UPMC SORBONNE UNIVERSITÉS

- Dans les deux cas : découple le système du client du type concrêt de l'objet (manipulation de l'objet au travers de son interface)

- Dans le DP Factory Method :
  - Une méthode d'une classe est responsable de la création de l'objet
  - **Cette classe peut réaliser d'autres opérations**
  - Lorsque l'on hérite de cette classe, on redéfinit cette méthode spécifique

- Dans le DP Abstract Factory
  - L'objectif de la fabrique est **uniquement de construire un ensemble d'objet** d'une meme famille
  - On délègue la responsabilité de la création de ces objets à une fabrique particulière

# En résumé

UPMC SORBONNE UNIVERSITÉS

- La fabrique abstraite :
  - Permet à un client de **créer des objets**
  - Ces objets sont reliés entre eux (**même famille** d'objets)
  - Par exemple : les composant d'une GUI
  - Le client n'a **pas besoin de connaitre le type concret** de l'objet
  - Le type concret de l'objet peut varier en fonction du **contexte**
  - Le type concret est définit **dynamiquement** à l'exécution (runtime)

# SINGLETON

UPMC SORBONNE UNIVERSITÉS

- Objectif :
  - Garantir qu'une classe ne possède qu'une seule instance
  - Fournir un point d'accès à cette classe

- Exemple :
  - Classe qui gère la connexion à une base de donnée
  - Driver d'un périphérique

# SINGLETON

UPMC SORBONNE UNIVERSITÉS

- Comment :
  - **Créer une classe avec au plus une seule instance ?**
  - Comment empécher d'autres développeurs de créer d'autres instances de cette classe ?

# SINGLETON

- **Comment** :
  - Créer une classe avec au plus une seule instance ?

- **Solution 1 :**
- Dans la classe `MonSingleton` que vous créez, définir un **champ static** :

```
private static MonSingleton singleton =
                                new MonSingleton()
```

- Creer une **méthode pour accéder** à cette instance

```
public static MonSingleton getMonSingleton()
```

# SINGLETON

UPMC SORBONNE UNIVERSITÉS

- Quel est l'inconvénient d'initialiser un objet lors de la déclaration du champ ?

# SINGLETON

UPMC SORBONNE UNIVERSITÉS

- Quel est l'inconvénient d'initialiser un objet lors de la déclaration du champ ?

- On ne dispose pas forcément de toutes les informations au moment de la création de cet objet

- Il n'est pas forcément judicieux d'initialiser un singleton au démarrage de l'application, en particulier si la création nécessite des ressources (connexion à une base de donnée par exemple)

# SINGLETON

UPMC SORBONNE UNIVERSITÉS

- Quelle autre solution ?

# SINGLETON

- Quel est l'inconvénient d'initialiser un objet lors de la déclaration du champ ?

- Initialisation tardive (lazy-initialisation)

```
Public static MonSingleton getSingleton() {
   if (singleton == null) {
      singleton = new MonSingleton();
   //…
   return singleton
}
```

# SINGLETON

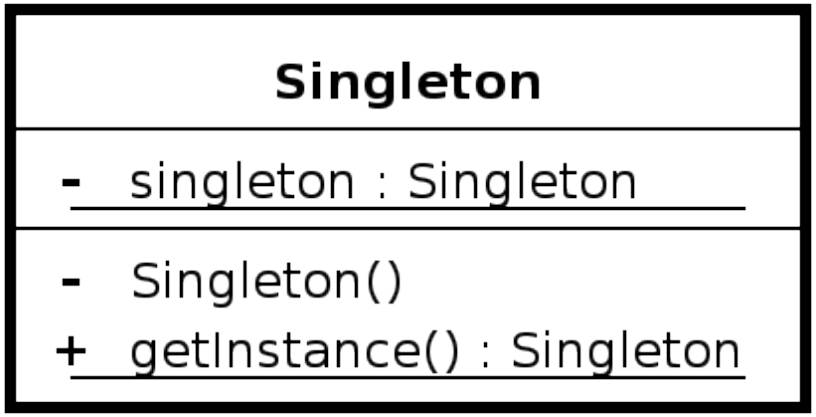
UPMC SORBONNE UNIVERSITÉS

- Comment :
  - Créer une classe avec au plus une seule instance ?
  - **Comment empêcher d'autres développeurs de créer d'autres instances de cette classe ?**

# SINGLETON

- Comment :
  - Comment empêcher d'autres développeurs de créer d'autres instances de cette classe ?

- Créer un seul constructeur avec accès privé !



# SINGLETON

UPMC SORBONNE UNIVERSITÉS

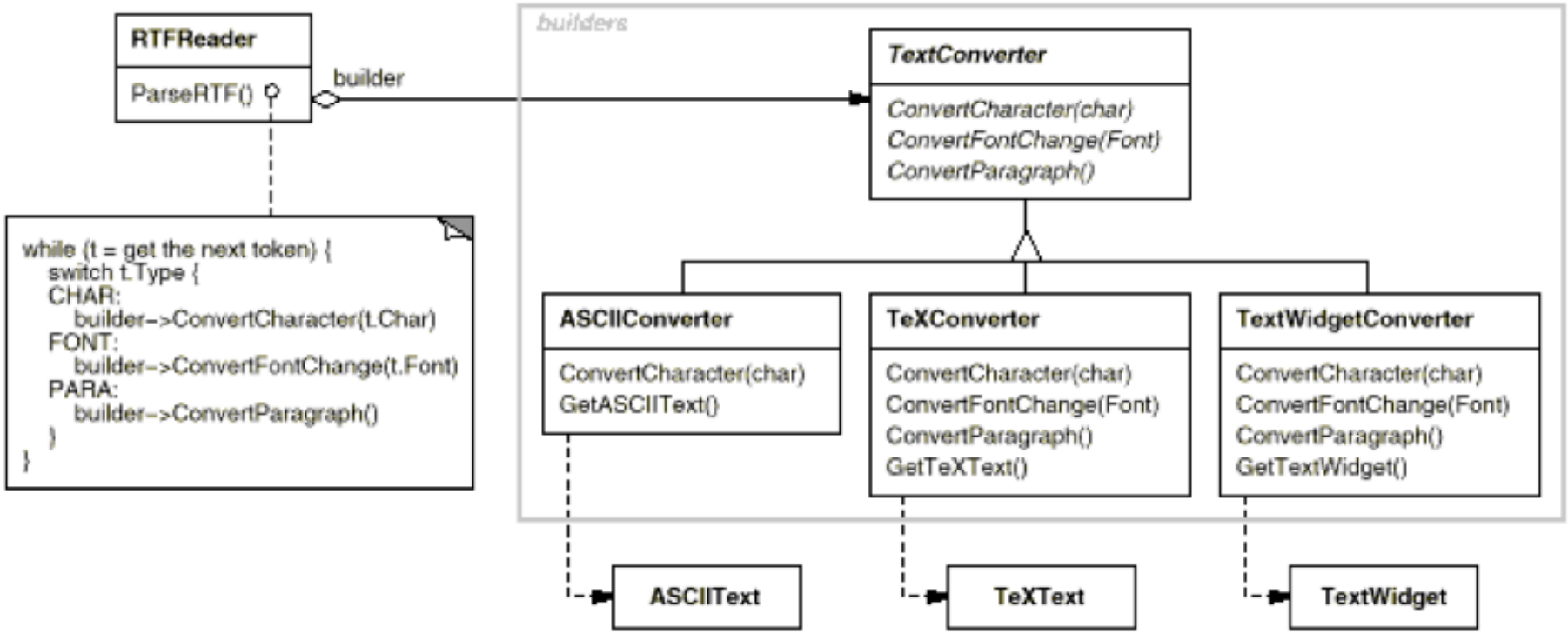
- Remarques :
  - SINGLETON ≠ Classe utilitaire : **pas d'instance d'une classe utilitaire**, uniquement des méthodes statiques (ex : Math.sqrt(), System.out.pintln()…)
  - Attention : ne pas l'utiliser pour créer des variables globales

# BUILDER (Monteur)

UPMC SORBONNE UNIVERSITÉS

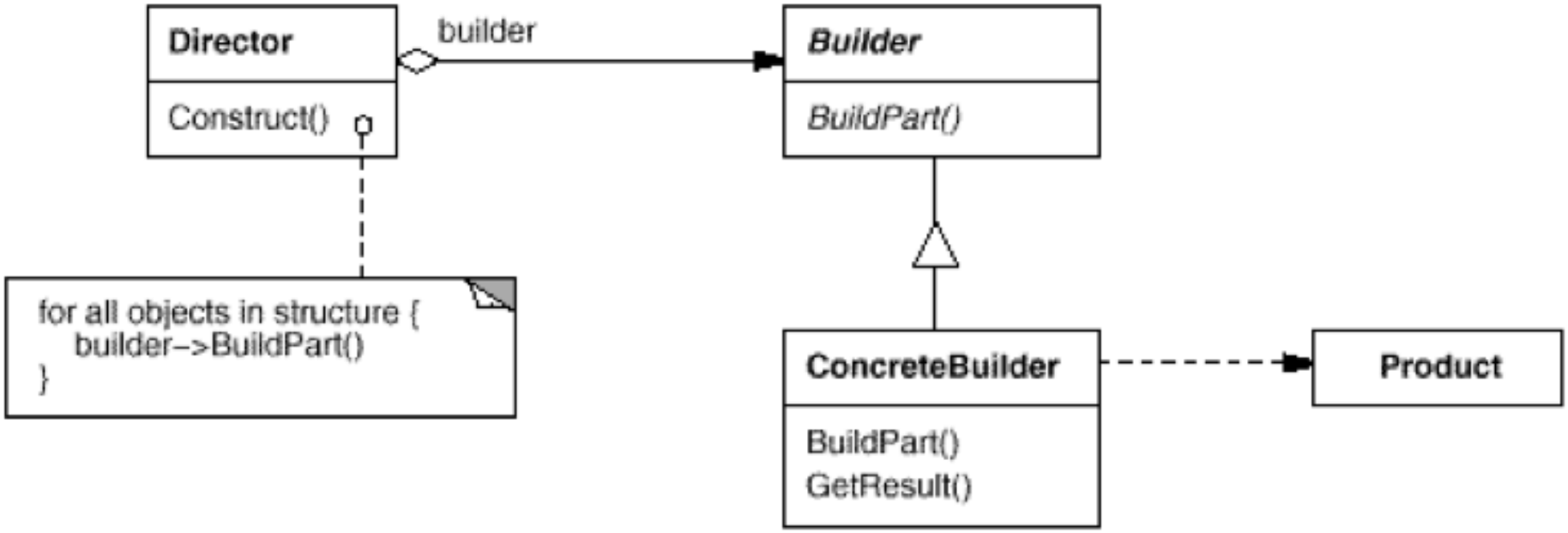
- Sépare la construction d'un objet complexe de sa représentation
  - Une classe se concentre sur la construction de l'objet (builder)
  - La classe principale se concentre sur l'utilisation d'un instance valide de l'objet
- Utile pour construire un objet progressivement (construction d'un objet à partir d'un fichier texte par exemple)
- Utile pour obtenir des **représentation différentes** avec le **même procédé de construction**

# BUILDER




- Un lecteur de fichier RTF doit être capable de convertir un format RTF dans d'autre formats : ASCII, Pdf, Tex
- Il faut pouvoir facilement ajouter de nouveau formats

# BUILDER (Monteur)



# BUILDER (Monteur)

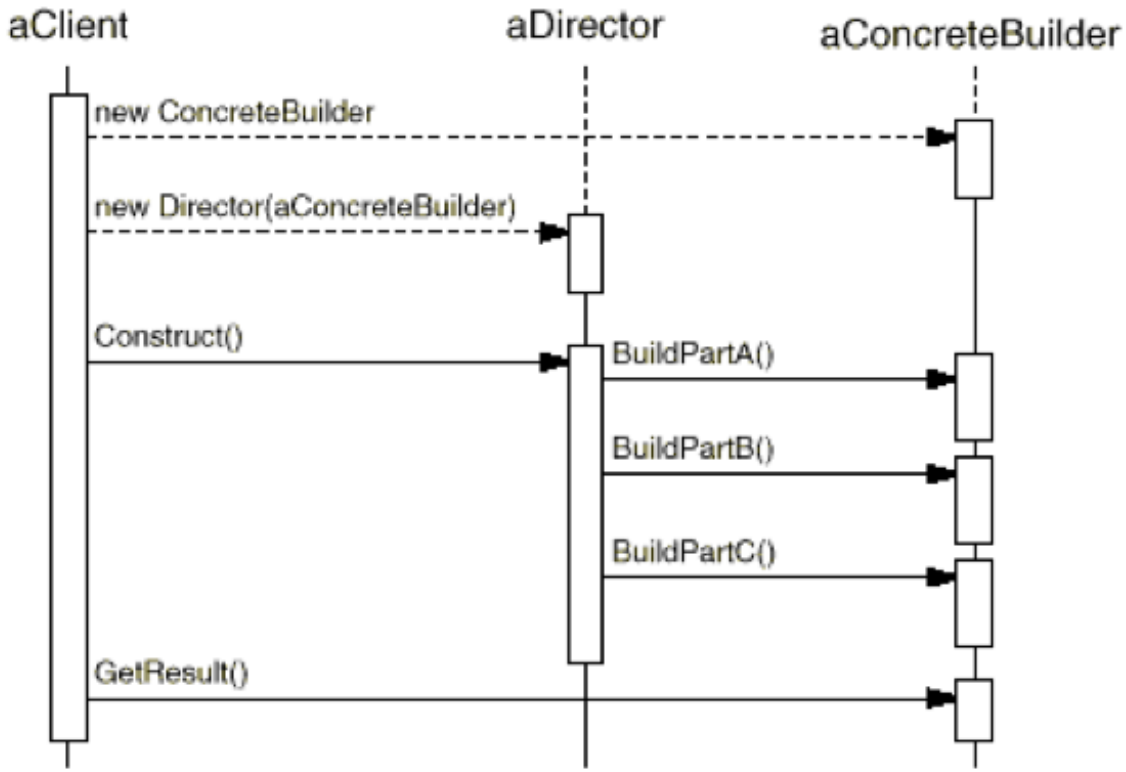



Diagramme d'interaction

# BUILDER

- Autres exemples d'application :
  - Visualisation d'un environnement virtuel VRML
  - Achat sur un site internet (l'objet "commande" et construit séquentiellement)
  - Système de réservation de salle
  - …

UPMC
SORBONNE UNIVERSITÉS

Les Design Patterns

# DESIGN PATTERNS DE STRUCTURE

# ADAPTER

UPMC SORBONNE UNIVERSITÉS

- **Objectif** :
  - Fournir une classe qui réponde au besoin du client (Interface client)
  - En utilisant les services d'une autre classe (interface différente)

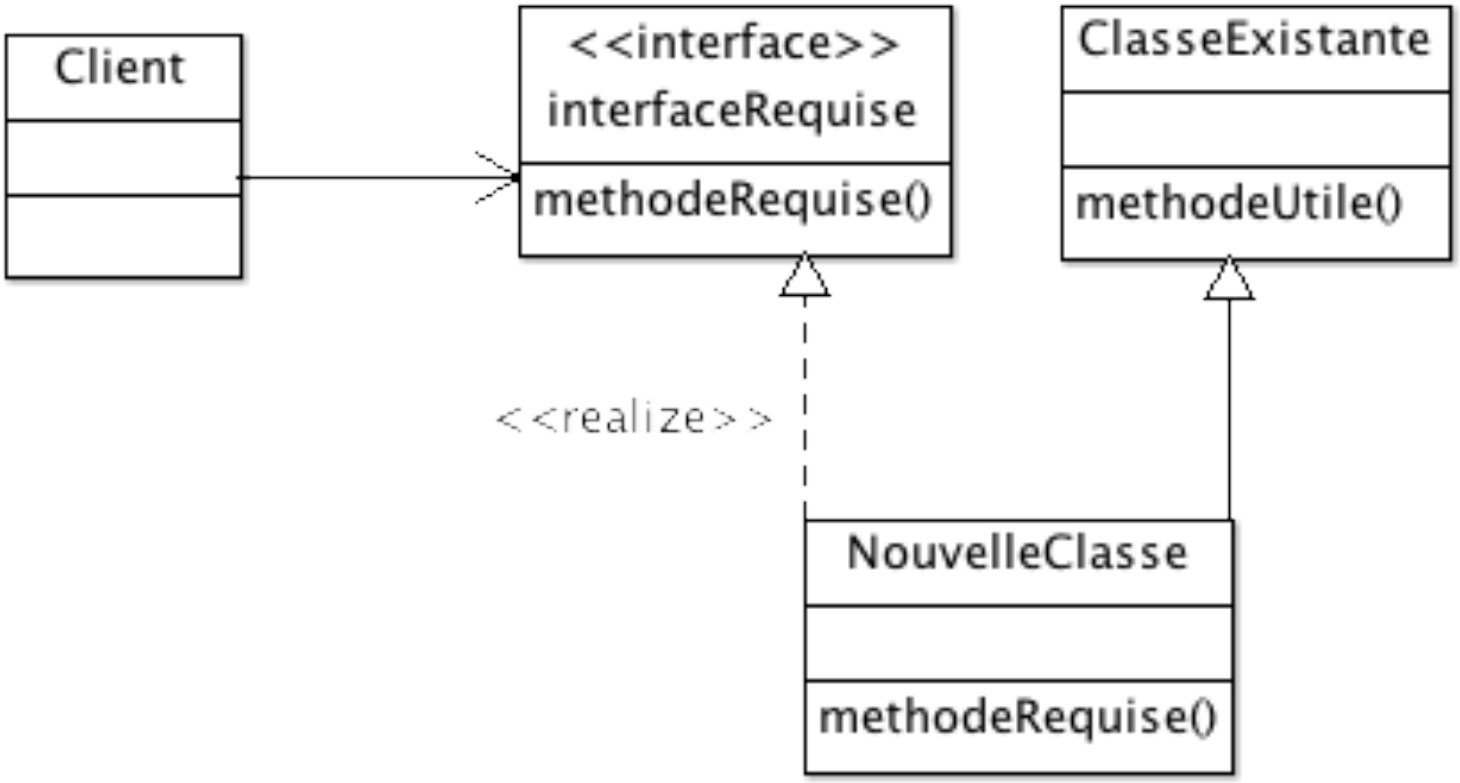
- **Pourquoi** :
  - On souhaite utiliser une API existante mais les signatures des méthodes ne conviennent pas.
  - On souhaite utiliser d'ancienne classe (fonctionnalité souhaitée) mais les conventions de nommage ne conviennent plus au nouveau standard
  - Pour interfacer deux systèmes existants
    - Ex : pilote d'un périphérique qui ne correspond pas à l'interface utilisée par le système pour les autres périphériques)
  - Mettre des données dans un format approprié (format de dates par exemple)

# ADAPTER

- Comment ?

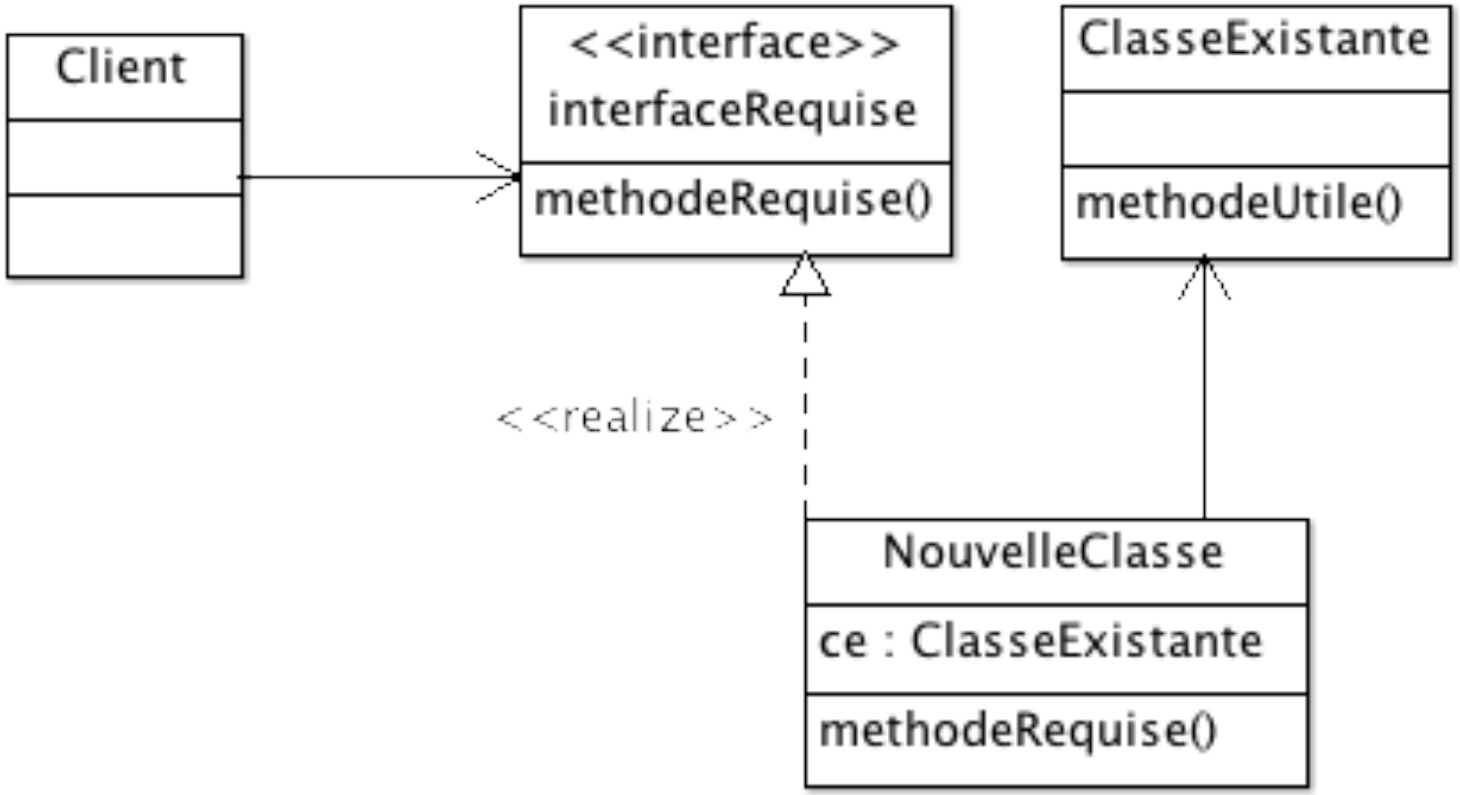
# ADAPTER

- Par héritage



# ADAPTEUR

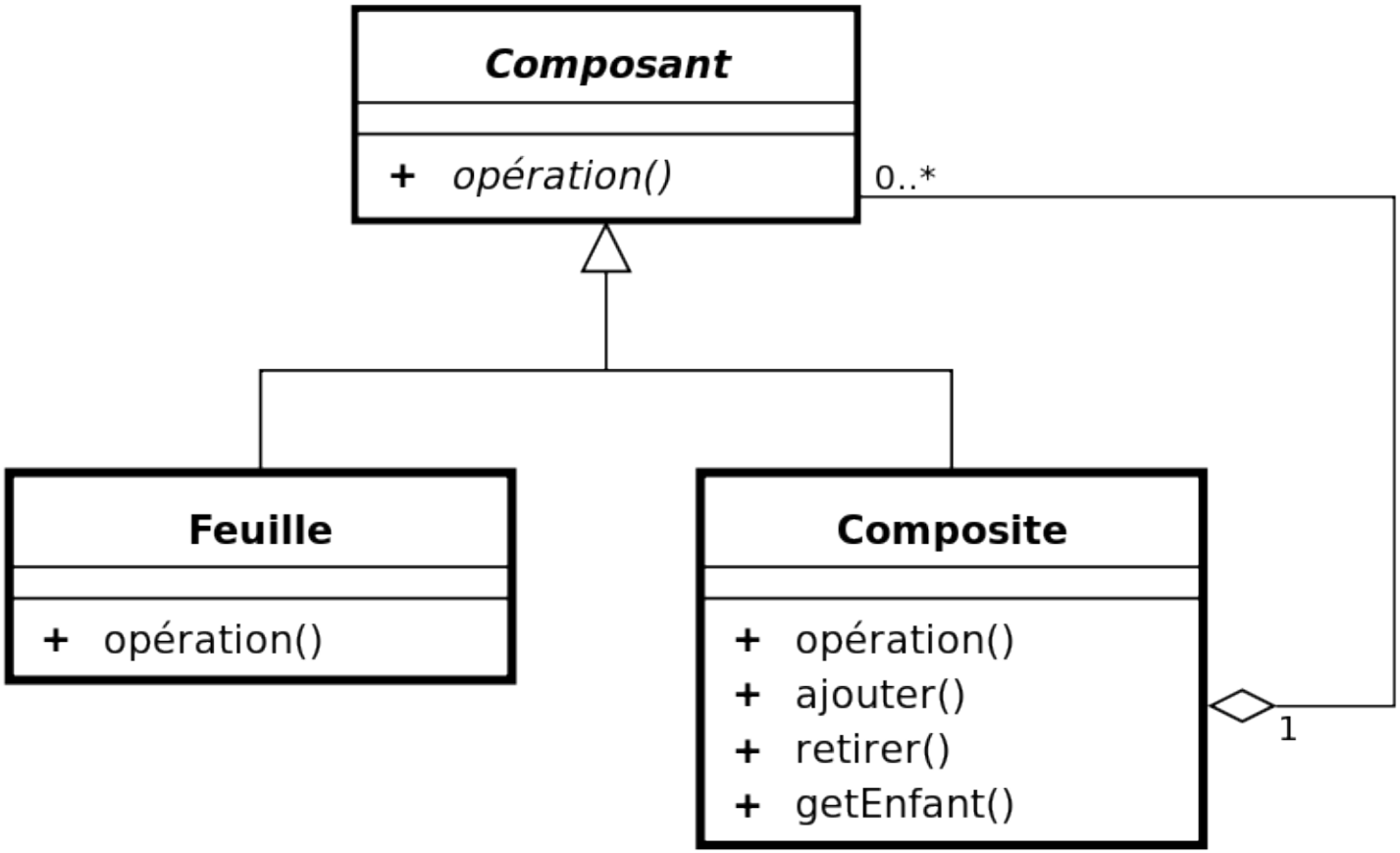
- Par délégation (composition)



# COMPOSITE

- Objectif
  - Organiser des objets en structure arborescente
  - Permettre aux clients de traiter de façon uniforme
    - des objets individuels
    - et des compositions d'objets
- Pourquoi ?
    - Utile pour manipuler des structures arborescentes de profondeur variable
    - Ex :
      - dans une application de dessin, un élément graphique peut être composé de plusieurs éléments graphiques
      - .

# COMPOSITE


Composant
+ opération()
0..*
Feuille
+ opération()
Composite
+ opération()
+ ajouter()
+ retirer()
+ getEnfant()
1

# Exemple en Java

```java
import java.util.ArrayList;

interface Graphic {

    //Imprime le graphique.
    public void print();

}

class CompositeGraphic implements Graphic {

    //Collection de graphiques enfants.
    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();

    //Imprime le graphique.
    public void print() {
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }

    //Ajoute le graphique à la composition.
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }

    //Retire le graphique de la composition.
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }

}

class Ellipse implements Graphic {

    //Imprime le graphique.
    public void print() {
        System.out.println("Ellipse");
    }

}
```

```java
public class Program {

    public static void main(String[] args) {
        //Initialise quatre ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialise trois graphiques composites
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes les graphiques
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);

        graphic2.add(ellipse4);

        graphic.add(graphic1);
        graphic.add(graphic2);

        //Imprime le graphique complet (quatre fois la chaîne "Ellipse").
        graphic.print();
    }
}
```

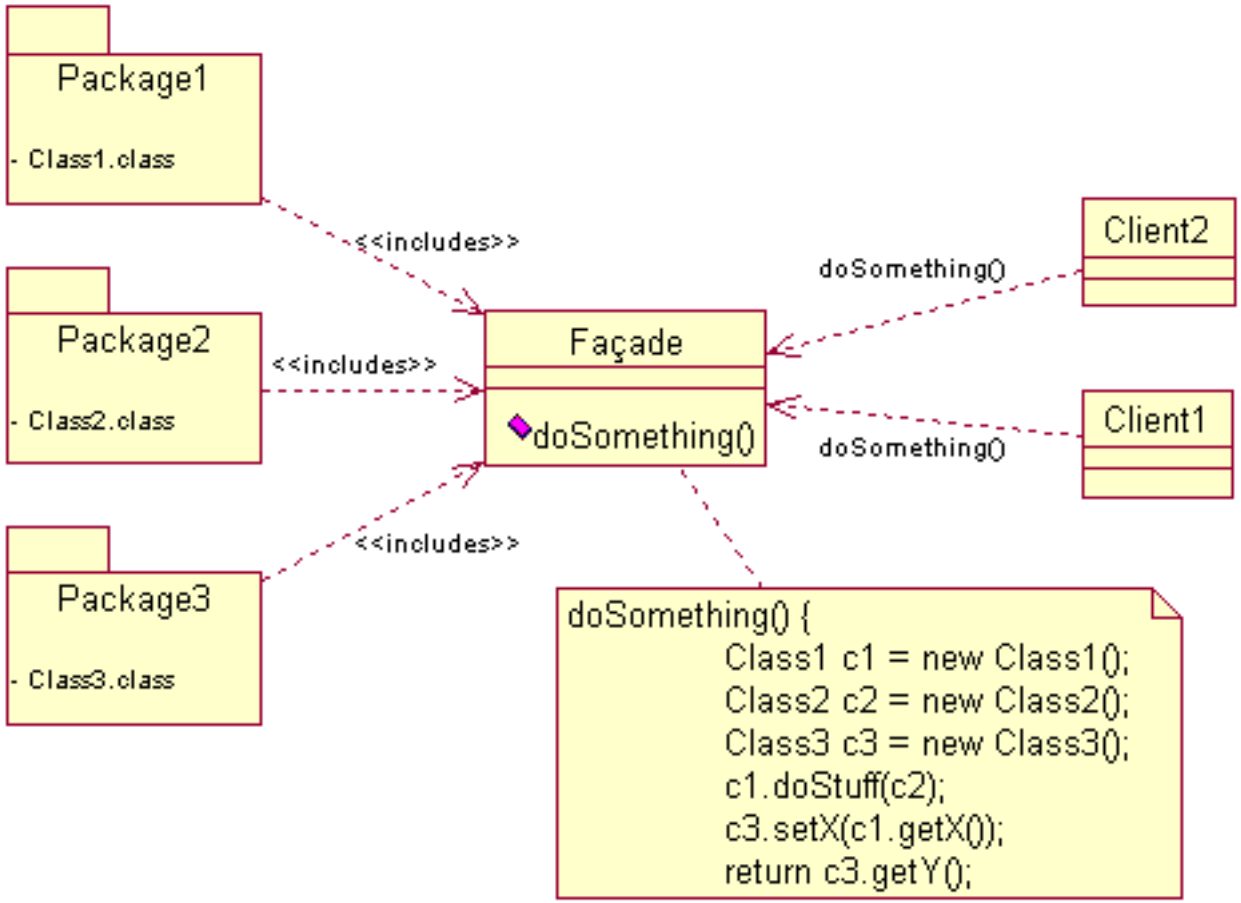
On peut facilement ajouter d'autres formes (rectangle, triangle…)

# FACADE

UPMC SORBONNE UNIVERSITÉS

- **Objectif**
  - fournir une interface **simplifiant** l'emploi d'un sous-système.

- **Exemples**
  - Pour **simplifier** l'utilisation d'une librairie complexe à utiliser
  - Pour **limiter** les fonctionnalités d'un système, et ne conserver que les partie utiles


Package1
- Class1.class
Package2
- Class2.class
Package3
- Class3.class
<<includes>>
<<includes>>
<<includes>>
Façade
doSomething()
Client2
doSomething()
Client1
doSomething()
doSomething() {
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
c1.doStuff(c2);
c3.setX(c1.getX());
return c3.getY();

# Exemple en Java

```java
import java.util.*;

// Façade
class UserfriendlyDate {
    GregorianCalendar gcal;

    public UserfriendlyDate(String isodate_ymd) {
        String[] a = isodate_ymd.split("-");
        gcal = new GregorianCalendar(Integer.parseInt(a[0]),
              Integer.parseInt(a[1])-1 /* !!! */, Integer.parseInt(a[2]));
    }

    public void addDays(int days) {
      gcal.add(Calendar.DAY_OF_MONTH, days);
    }

    public String toString() {
      return String.format("%1$tY-%1$tm-%1$td", gcal);
    }
}

// Client
class FacadePattern {
    public static void main(String[] args) {
        UserfriendlyDate d = new UserfriendlyDate("1980-08-20");
        System.out.println("Date : "+d);
        d.addDays(20);
        System.out.println("20 jours après : "+d);
    }
}
```

Date: 1980-08-20
20 jours après : 1980-09-09

UPMC
SORBONNE UNIVERSITÉS

Les Design Patterns

# DESIGN PATTERNS DE COMPORTEMENT

# ITERATOR

UPMC SORBONNE UNIVERSITÉS

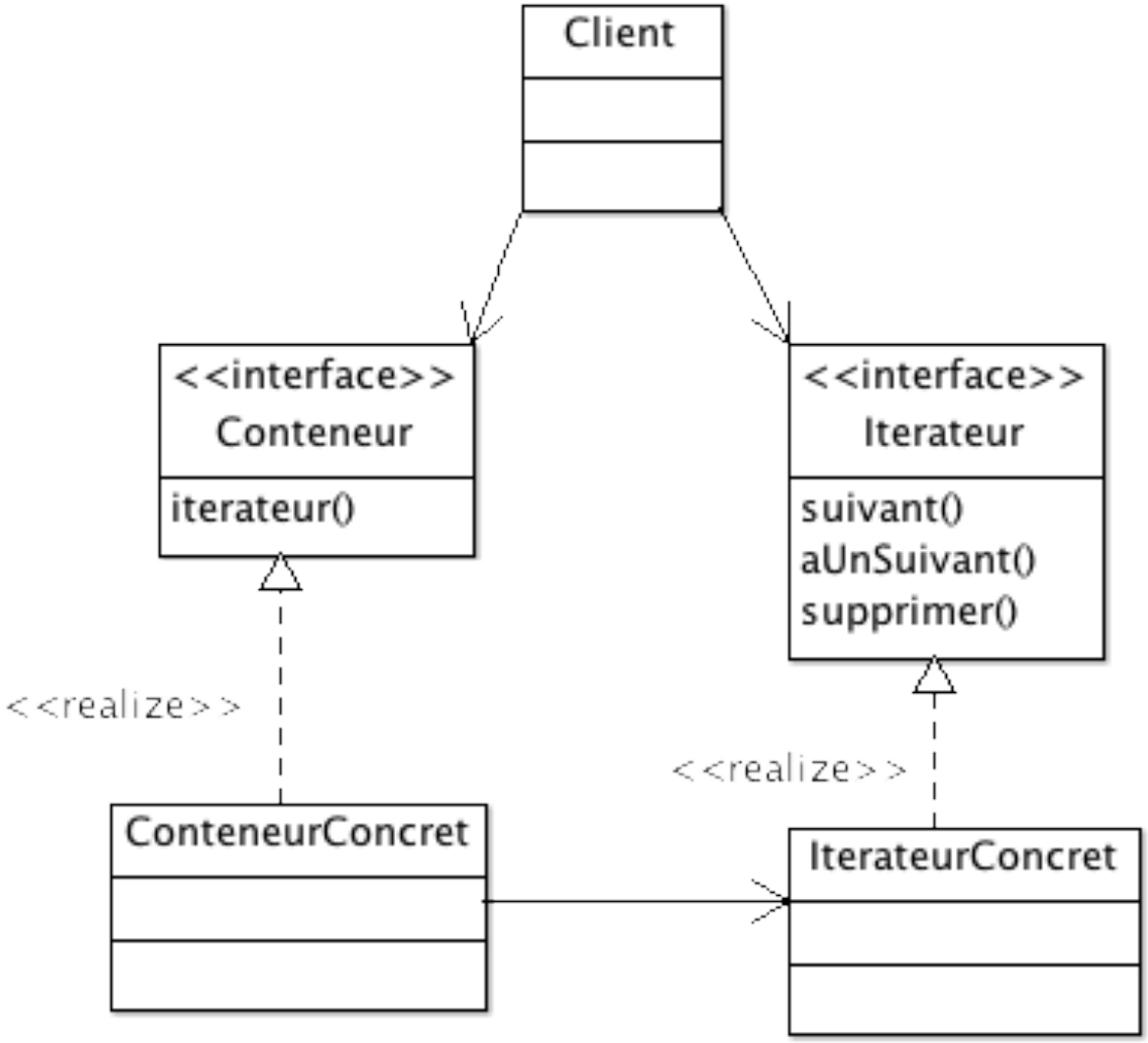
- Objectif :
  - Accéder de manière **séquentielle** aux éléments d'une collection

- Pourquoi l'utiliser
  - Lorsque l'on souhaite parcourir les éléments d'un **objet complexe**.
  - Et que la **structure de l'objet peut varier**
  - Que l'on ne souhaite pas **exposer la représentation interne** du conteneur

# Exercice

- Faire une représentation UML de ce Design Pattern

# ITERATOR


Client
<<interface>>
Conteneur
iterateur()
<<interface>>
Iterateur
suivant()
aUnSuivant()
supprimer()
<<realize>>
<<realize>>
ConteneurConcret
IterateurConcret

# Question

- Quels sont les avantages d'utiliser un itérateur plutôt que l'indexation ?

# Réponse

- Indexation pas toujours adaptée aux conteneurs
  - Si pas de méthode d'accès à un élément quelconque
  - Si l'accès à un élément quelconque est lent (dans une liste chaînée par exemple)
- Itérateur : manipulation indépendante de la structure de donnée
- L'itérateur peut ajouter des contraintes supplémentaire sur l'accès aux éléments (empêcher qu'un élément soit sauté par ex)

# OBSERVER

- **Habituellement** : un client qui souhaite obtenir des informations d'un objet → utilisation de méthodes

- **Mais** si **l'objet change**, comment le client peut-il prendre en compte ces changements ?

# OBSERVER

- Mauvaise stratégie
  - **L'objet prévient les clients lorsqu'une information qui les concerne change**

- Pourquoi est ce que ce n'est pas bien ?

# OBSERVER

- Mauvaise stratégie
  - **L'objet prévient les clients lorsqu'une information qui les concerne change**

- Pourquoi est ce que ce n'est pas bien ?

- Ce n'est pas la responsabilité de l'objet de savoir quelle information est pertinente pour chaque client et d'actualiser ce client

# OBSERVER

- Quelle solution est préférable ?

# OBSERVER

UPMC SORBONNE UNIVERSITÉS

- Quelle solution est préférable ?

- Les clients sont prévenus que l'état de l'objet a changé
- Les clients peuvent agir en conséquence : en demandant le nouvel état de l'objet par exemple

# OBSERVER

UPMC SORBONNE UNIVERSITÉS

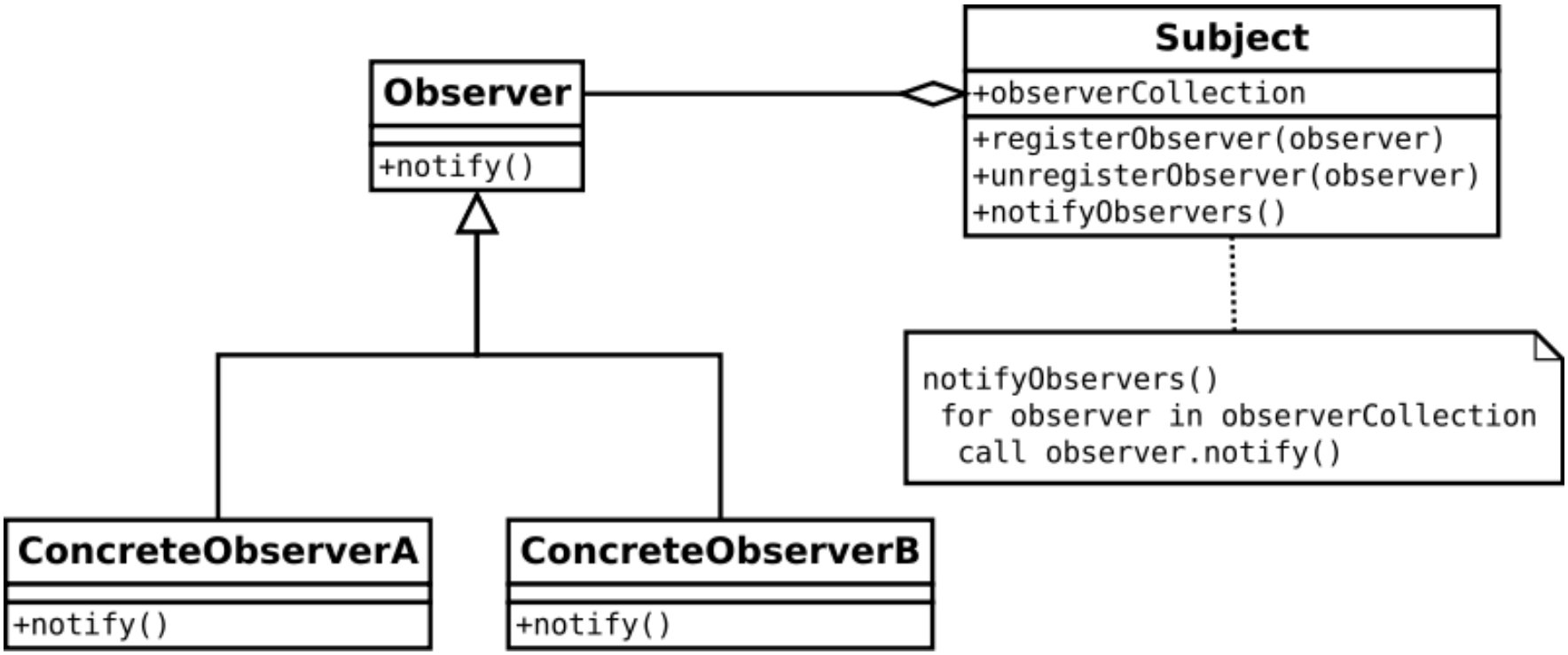
- Exemple
  - Programmation évènementielle : interface graphique utilisateur
  - Un robot doit adapter son comportement en fonction de son environnement

# Exercice

UPMC SORBONNE UNIVERSITÉS

- Faire le diagramme UML de ce Design Pattern

# OBSERVER


Observer
+notify()
Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
notifyObservers()
for observer in observerCollection
call observer.notify()
ConcreteObserverA
+notify()
ConcreteObserverB
+notify()

# STRATEGY

UPMC SORBONNE UNIVERSITÉS

- **Objectif** :
  - Définir des **stratégies** (algorithmes) **différents**
  - **Encapsuler** ces stratégies (classes distinctes)
  - Les rendre **interchangeables** (opération commune)

# STRATEGY

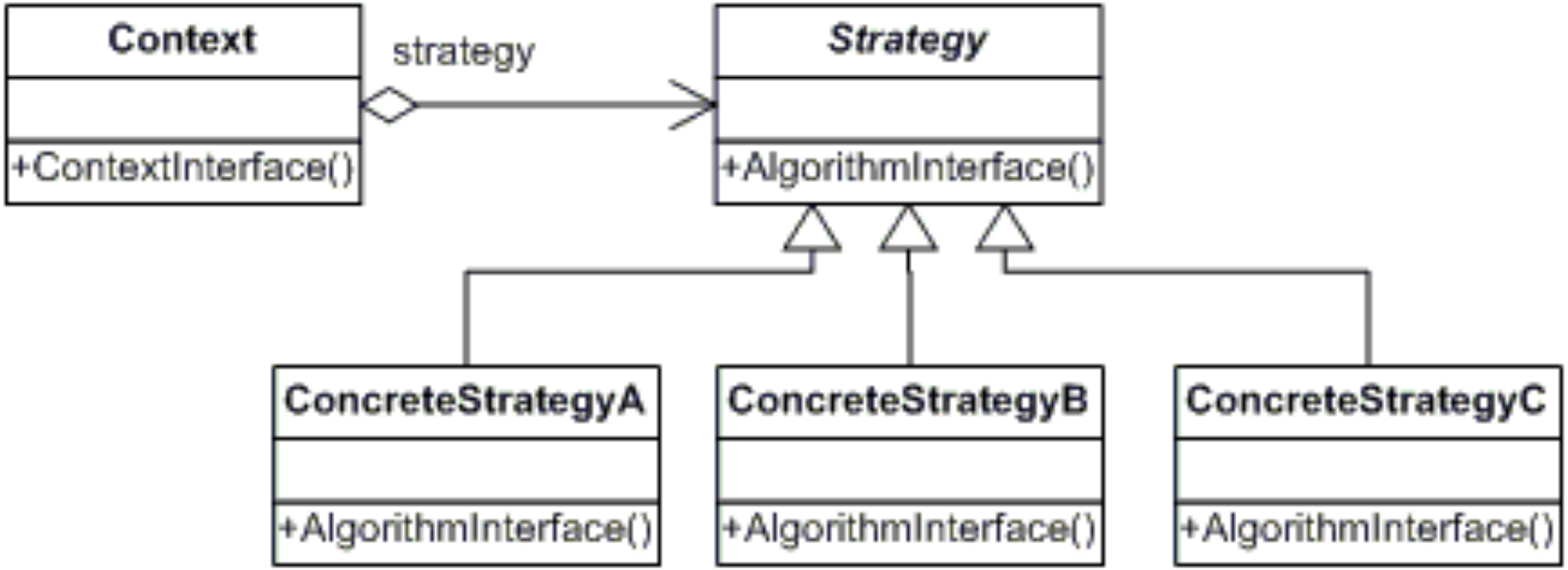
UPMC SORBONNE UNIVERSITÉS

- **Exemple**
  - Un moteur de recommandation sur un site internet :
    - Si la personne est enregistrée sur le site ➔ stratégie 1 (on regarde ce que les autres clients qui lui ressemblent ont acheté)
    - Sinon : suggestion sur la base de son historique de navigation
  - Un jeu video : la tratégie de l'ordinateur s'adapte au niveau du joueur
  - Un algorithme de traitement d'image qui est choisi dynamiquement en fonction du type d'image à traiter
  - Comportement d'avatars dans un monde virtuel…
  - Choix de l'algo

# STRATEGY

- Comment faire ?

# STRATEGY


Context
+ContextInterface()
strategy
Strategy
+AlgorithmInterface()
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()

# Design Patterns : Principes sous-jacents

UPMC SORBONNE UNIVERSITÉS

- Isoler et **encapsuler la partie variable**
  - Algorithme dans Strategy
  - Façon d'itérer dans Iterator
  - Création dans Factory
- Favoriser **composition et délégation** par rapport à l'héritage
  - **Héritage** : lien **statique**, très dépendant de l'implémentation → on ne peut pas hériter de classes différentes à des moments différents
    **Sensible aux changements** de la classe mère (changement d'un variable protected)
  - Réserver l'héritage pour traduire une relation « est un » statique

# Design Patterns : Principes sous-jacents

UPMC SORBONNE UNIVERSITÉS

- Programmer pour des **interfaces** plutôt que des classes concrètes
  - Plus souple et plus évolutif
- Toujours chercher le **couplage le plus faible** possible entre des parties indépendantes qui interagissent
  - exemple Observer
- Classes **ouvertes en extension, fermées en modification**
  - La réutilisation et la modification ou l'enrichissement doit se faire sans remettre en cause l'existant

# Design Patterns : Principes sous-jacents

UPMC SORBONNE UNIVERSITÉS

- **Ne parlez qu'à vos amis**
  - Limiter le nombre d'objets connus par un objet, réfléchir aux dépendances induites

- Ne m'appelez pas, je vous appellerai
  - **Relations asymétriques** dans les communications : les gros composants dépendent des plus petits, pas l'inverse.
  - **Eviter les cycles** de dépendances

- Une classe ne devrait avoir **qu'une seule responsabilité**
  - Permet une meilleure gestion des changements, distinguer les rôles et les capturer par des interfaces distinctes