

**ENSAF**

**3<sup>ème</sup> année Filière : Génie Informatique**

**Année universitaire 2019/2020**

# Design Patterns

**Mohammed Berrada**

**mohammed.berrada@gmail.com**

# Sommaire

- Rappel : POO
- Introduction au Design Patterns
- Designs de création
- Design de structure
- Design de comportement
- Références

# **Programmation Orientée Objet**

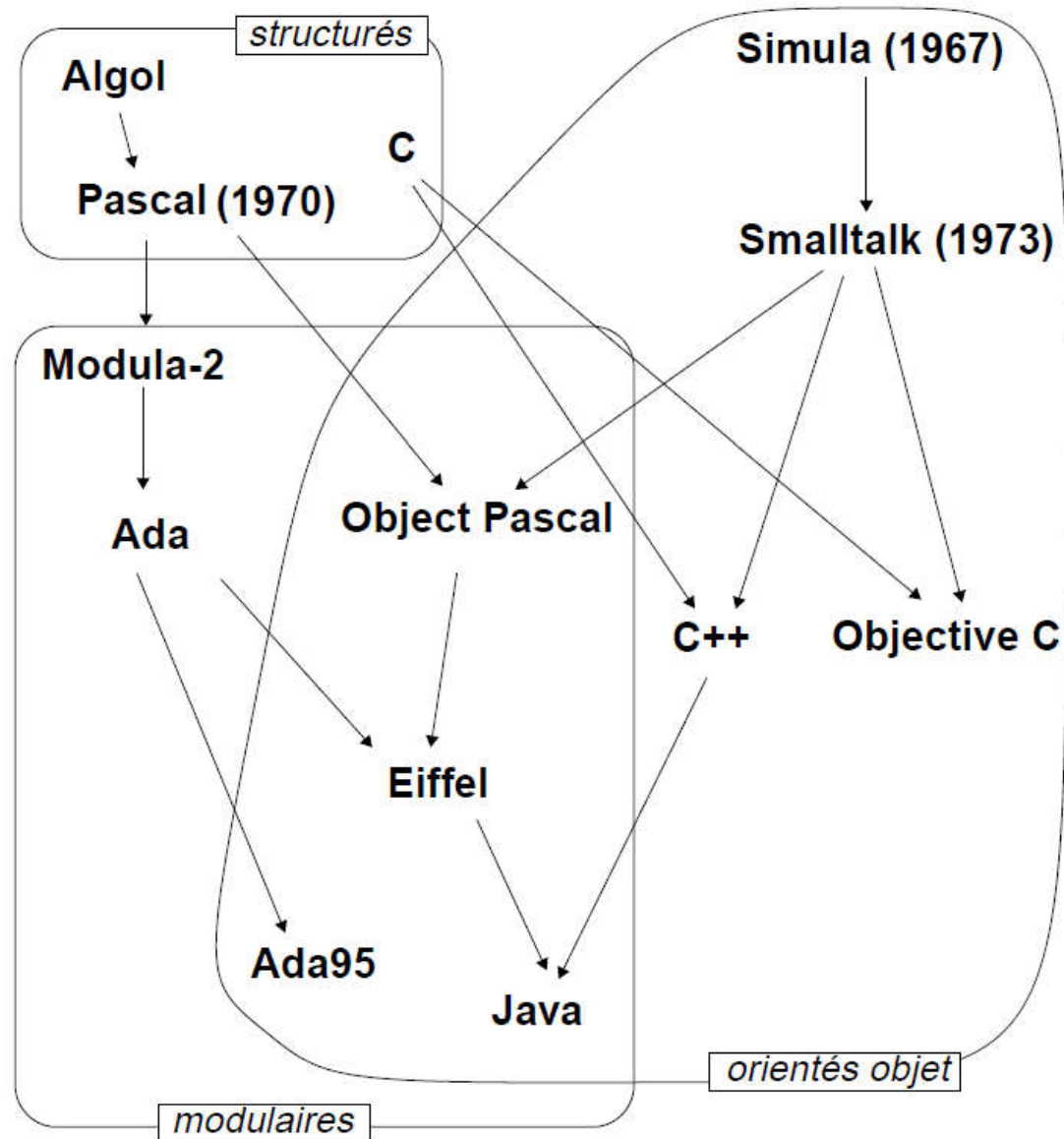
# Plan 1

- Concepts de l'approche Objet
- Origines des langages orientés objet
- Modularité et encapsulation
- Types abstraits
- Classes, objets et messages
- Exemples (langage Java)

# Concepts de l'approche Objet

- **Classe** : type de donnée abstrait
  - Caractéristiques : attribues et méthodes
- **Encapsulation** : masquer les détails d'implémentation d'un l'objet
  - Facilite l'évolution d'une application
  - Garantit l'intégrité des données
- **Héritage** : transmission les caractéristiques d'une classe vers une/plusieurs sous classe(s)
  - Spécialisation et généralisation
- **Polymorphisme**: faculté d'appliquer une méthode à des objets de classes différentes
- **Agrégation** : composition des objets, plus complexes, d'une classe à partir des objets d'autres classes de base

# Origines des langages Orientés Objet



# Modularité

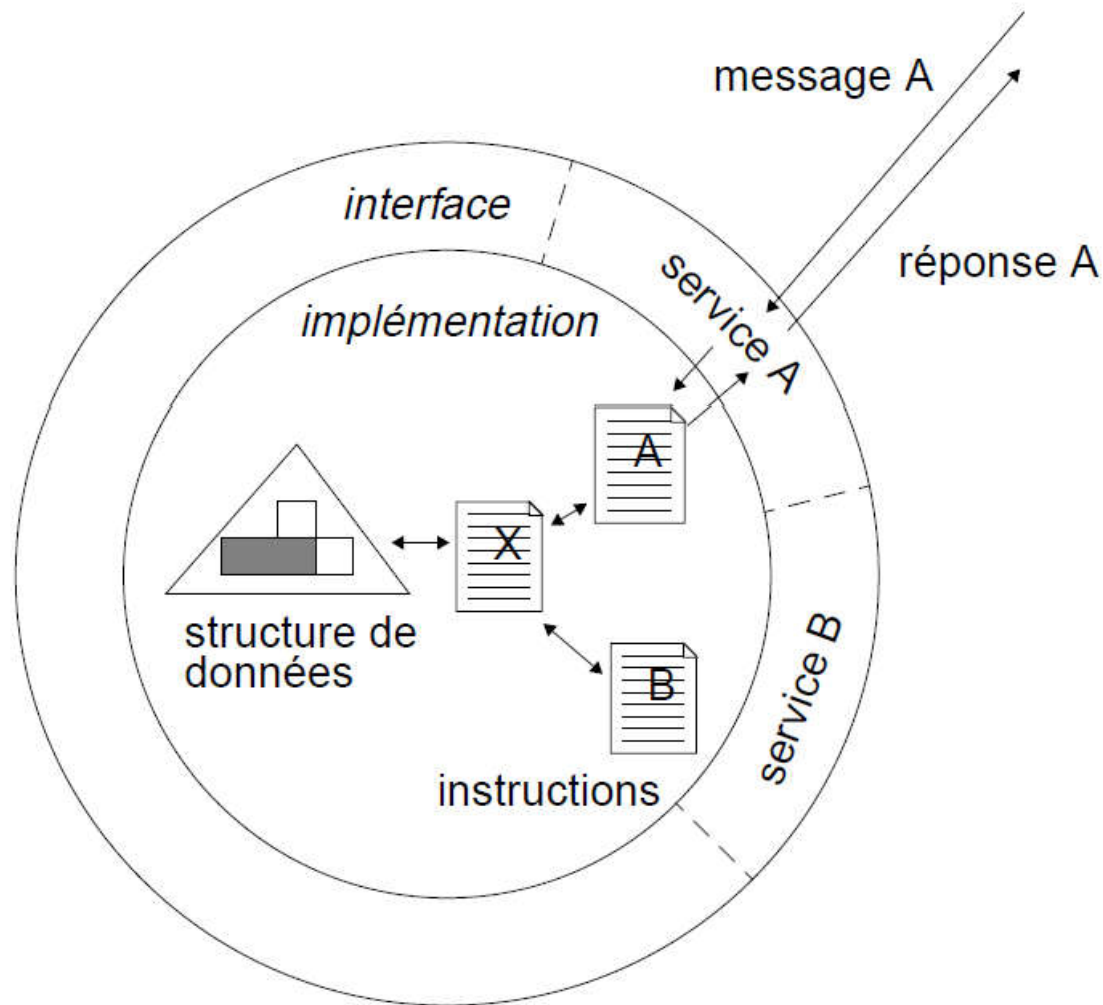
- technique de décomposition de systèmes
- réduire la complexité d'un système par un assemblage de sous-systèmes plus simples
- réussite de la technique dépend du degré d'indépendance entre les sous-systèmes (ou degré de couplage)
- on appelle module, un sous-système dont le couplage avec les autres est relativement faible par rapport au couplage de ses propres parties

# Encapsulation

- technique pour favoriser la modularité (l'indépendance) des sous-systèmes
- séparer l'interface d'un module de son implémentation,
- interface (partie publique): liste des services offerts (quoi)
- implémentation (partie privée): réalisation des services (comment)
  - structure de données
  - instructions et algorithmes
- protection des données par des règles (dans les instructions): les modules communiquent par messages, pas par accès aux données



# Un module



# Types abstraits

## Type “classique”

- ensembles de valeurs possibles

## Type abstrait

- ensemble d’opérations applicables

**L’abstraction** consiste à penser à un objet en termes d’actions que l’on peut effectuer sur lui, et non pas en termes de représentation et d’implantation de cet objet.

# Types abstraits (2)

## Exemple :TA rectangle

- Opération de construction:
  - rect: entier X, entier Y, entier L, entier H --> rectangle;
  - agrandissement: rectangle R, entier DL, entier DH --> rectangle.
- Opération d'accès
  - gauche: rectangle R --> entier;
  - haut: rectangle R --> entier;
  - largeur: rectangle R --> entier;
  - hauteur: rectangle R --> entier;

# Modules + Types abs. => Langage à objet

## Un objet est un module

- privé: variables d'instance, code des méthodes
- public: nom et paramètres des méthodes
- communication: invocation de méthodes, retour de résultats

## Classe = générateur d'objet

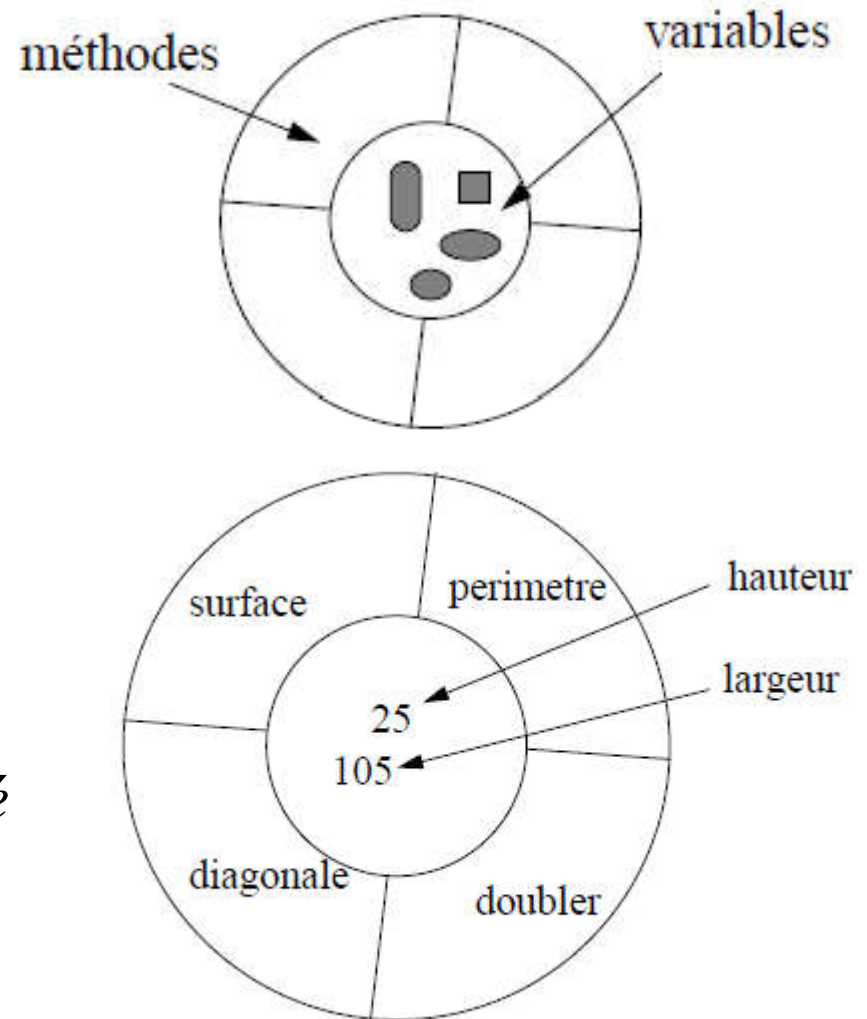
- génère des objets de même structure
- localise la définition de structure et action des objets
- définit la visibilité: private, public, etc.

## Classe --> type abstrait

- partie publique --> spécif. (partielle) TA

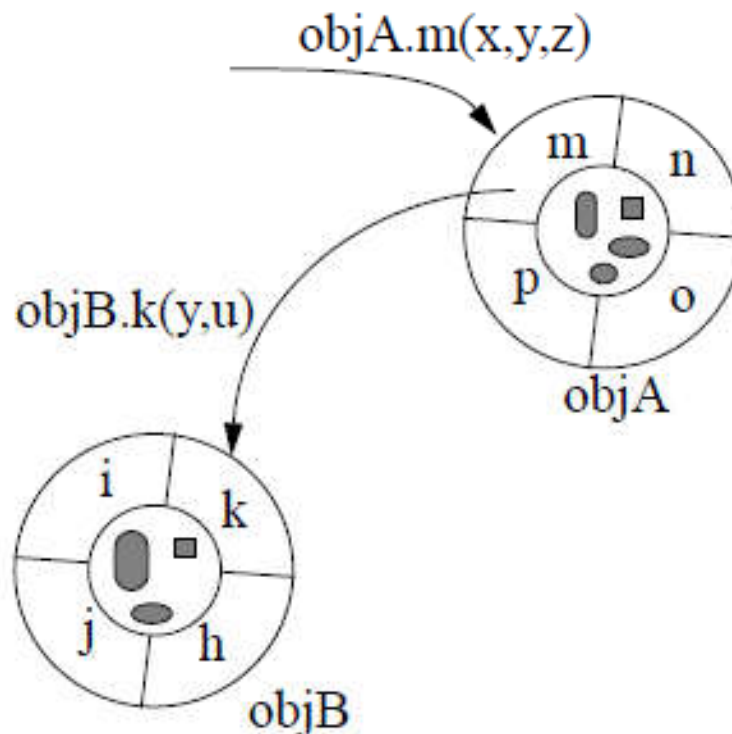
# Objet

- Une entité contenant des données (état) et des procédures associées (comportement)
  - **Exemple:** un objet *rectangle*
- ➔ Un objet possède une *identité* unique et invariable.



# Messages

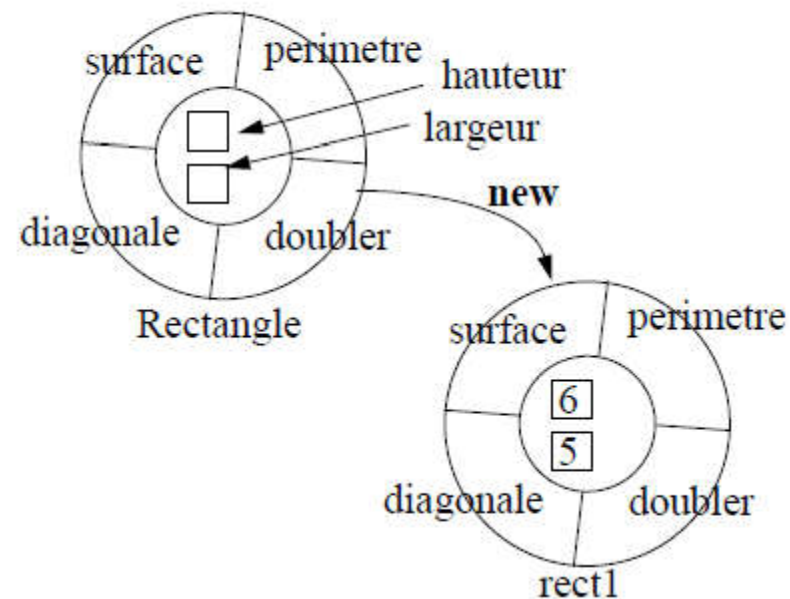
- Pour utiliser un objet on lui envoie des messages
- Un message déclenche l'exécution d'une méthode
- **Une méthode peut envoyer des messages à d'autres objets**



- Un système est constitué d'objets qui communiquent entre eux.

# Classes

- Une classe est un moule pour fabriquer des objets de même structure et de même comportement.
- Un objet est une instance d'une classe



- Une classe C définit un type C.
- Une variable de type C peut faire référence à un objet de la classe C.

# Déclaration d'une classe en Java

```
class Rectangle {  
    // variables d'instance  
    int largeur, hauteur;  
  
    // constructeur  
    Rectangle(int initialL, int initialH) {  
        largeur = initialL;  
        hauteur = initialH;  
    }  
    // méthodes  
    int perimetre() {  
        return 2 * (largeur+hauteur);  
    }  
    int surface() {  
        return largeur*hauteur;  
    }  
    void retaille(double facteur) {  
        largeur = (int) (largeur*facteur);  
        hauteur = (int) (hauteur*facteur);  
    }  
}
```



# Utilisation d'une classe Java

## Déclaration de variables

```
Rectangle r1, r2;
```

## Création d'objets (instantiation)

```
r1 = new Rectangle(50, 100);
```

```
r2 = new Rectangle(32, 150);
```

## Envoi de messages (utilisation)

```
r2.retaille(1.25);
```

```
System.out.println( r2.perimetre() );
```

# Références aux objets

Une variable de type C fait *référence* à un objet de la classe C.

```
Rectangle ra, rb;
```

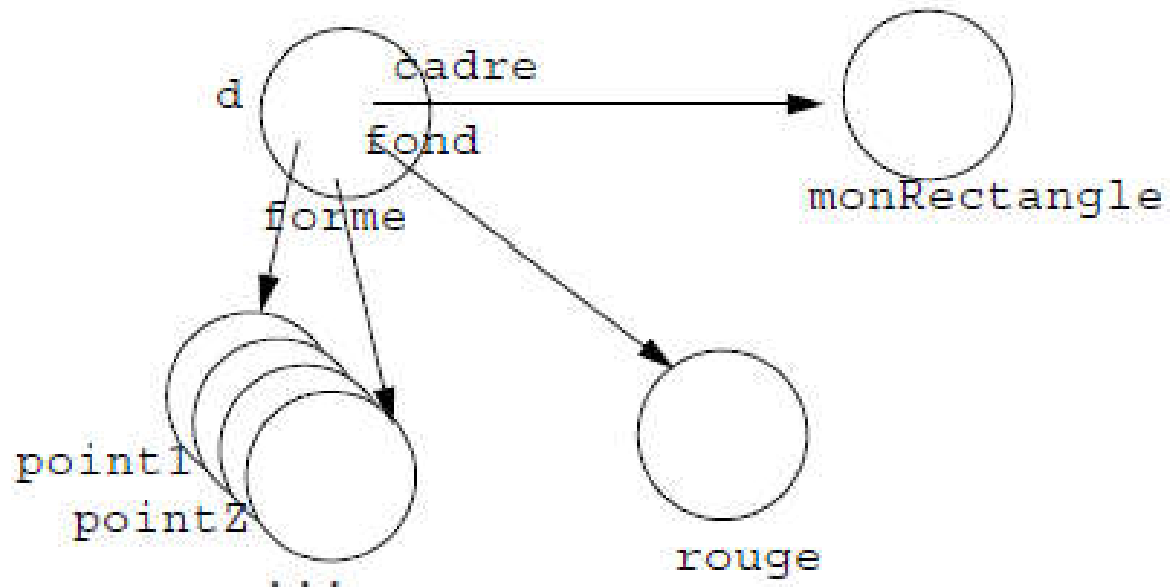
```
ra = new Rectangle(32, 125);
```

```
rb = ra // les deux variables font référence au même objet.
```

# Références aux objets (Exemple)

Les variables d'instance établissent des relations entre objets.

```
class Dessin {  
    Rectangle cadre;  
    Couleur fond;  
    Point[] forme;  
}  
..
```



```
d = new Dessin();  
d.cadre = monRectangle; d.fond = rouge;  
d.forme[0] = point1; d.forme[1] = pointZ; ...
```

# Identité et Égalité

Une des difficultés de la programmation O-O:

- **distinction objet / valeur  $\implies$  identité  $\neq$  égalité**

```
r1 = new Rectangle(128, 256);
```

```
r2 = new Rectangle(256, 128);
```

```
r3 = new Rectangle(128, 256);
```

```
(r1 == r2)  $\implies$  false
```

```
(r1 == r3)  $\implies$  false
```

```
r1.equals(r2)  $\implies$  false
```

```
r1.equals(r3)  $\implies$  true.
```

# Identité et Egalité (2)

## Test d'égalité

- (simple) comparaison des variables d'instance
- (complexe) calcul sur les variables d'instance

```
class Fraction {  
    private int numerateur, denominateur;  
    public boolean equals(Fraction f) {  
        return (this.numerateur * f.denominateur =  
            this.denominateur * f.numerateur); }  
}
```

# Egalité profonde et de surface

```
class ListeRect {  
    private Rectangle[] lesRectangles;  
    private int nbRectangles; }  

```

## Egalité de surface si

a.lesRectangles[0] == b.lesRectangles[0] **et**  
a.lesRectangles[1] == b.lesRectangles[1] **et**  
etc.

## Egalité profonde (ou récursive)

a.lesRectangles[0].equals (b.lesRectangles[0]) **et**  
a.lesRectangles[1].equals (b.lesRectangles[1]) **et**  
etc.

# Propriétés d'égalité

- **Symétrique:** `a.equals(b)` et `b.equals(a)` doivent toujours donner le même résultat;
- **Réflexive:** `a.equals(a)` doit toujours donner `true`;
- **Transitive:** si `a.equals(b)` et `b.equals(c)` donnent `true` alors `a.equals(c)` doit aussi donner `true`.

# La copie d'objets

## copie de surface

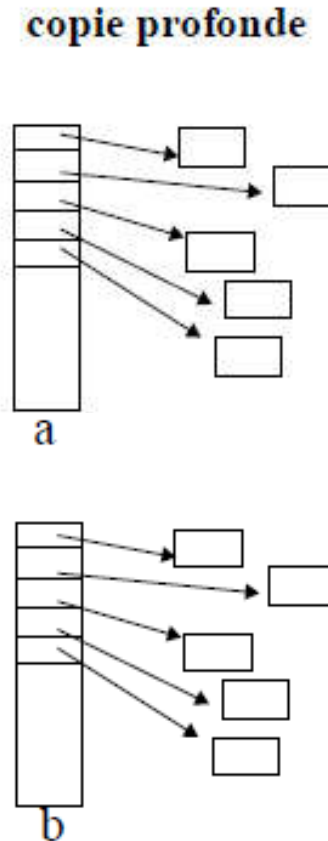
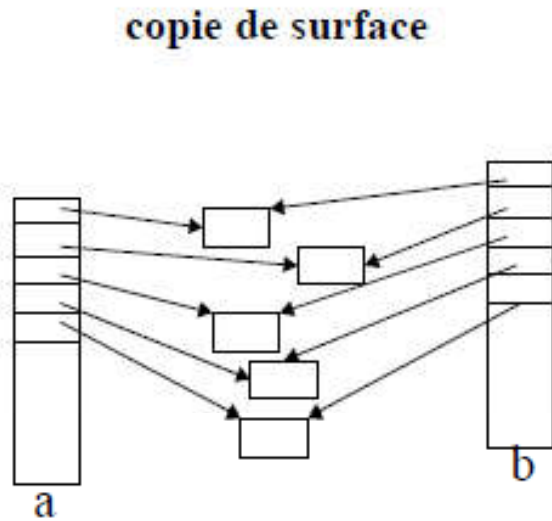
```
for (i=0; i<nbRectangles; i++)  
    {b.lesRectangles[i] = a.lesRectangles[i]; }
```

## copie profonde

```
for (i=0; i<nbRectangles; i++)  
    {b.lesRectangles[i] = a.lesRectangles[i].clone(); }
```



# La copie d'objets (2)



La copie de surface introduit du partage de structure

# Sous classes

- **Technique de réutilisation**

```
class Point {
    int x, y;
    public float distanceOrigine() {...}
    public float distance(Point pt) {...}
}

class PointTopographique extends Point {
    //hérité: int x, y;
    //hérité: public float distanceOrigine() {...}
    //hérité: public float distance(Point pt) {...}
    int alt;
    public int altitude() {...}
    public void ajusterAltitude(int a) {...}
}
```

- **Réutilisation souple** : On peut redéfinir des méthodes ou les surcharger

# Mécanisme de liaison dynamique

- Une variable ou un paramètre de type C peut faire référence à un objet de C ou d'une sous-classe de C.
- Le choix de la méthode à exécuter se fait dynamiquement lors d'exécution.rs de l'envoi d'un message.

**Exemple :** p.poidsTotal()

si p fait référence à une PieceSimple

exécute poidsTotal() de PieceSimple.

si p fait référence à une PieceComposée

exécute poidsTotal() de PieceComposée.

- La gestion du typage est laissée au système d'exécution

# Interfaces

- Comme une classe abstraite mais sans aucun code de méthodes
- Hétérarchie d'interfaces (héritage multiple)
- Une classe peut *implémenter une interface*, elle doit définir toutes les méthodes de l'interface.

# Exemple 1

```
interface Vendable {
    double prix();
    double rabais();
    void acheter(); }

class PieceSimple extends Piece implements Vendable {
    void affiche(){...}
    double prix(){return 7.5 * poidsTotal();}
    double rabais(){return 0.12 * prix();}
    void acheter(){...}
}

...
Vendable aVendre;
aVendre = maPieceSimple12;
x = aVendre.prix();
```

## Exemple 2 : Outil pour la généricité

```
// l'interface est un contrat: s'engager à être comparable
interface Comparable {
    boolean inferieur(Object c);
}

// classe capable de traiter tout objet respectant le
// contrat "Comparable".
class Tri {

    static void parEchange(Comparable[] x) {
        Comparable tmp;
        for (int i = 0; i < x.length - 1; i++) {
            for (int j = i + 1; j < x.length; j++) {
                if (x[j].inferieur(x[i])) {
                    tmp = x[i];
                    x[i] = x[j];
                    x[j] = tmp;
                }
            }
        }
    }
}
```

## Exemple 2 (suite)

```
class MotoTriable extends Moto
    implements Comparable {

    public boolean inferieur(Object c)
        { return this.prix() <= ((Moto)c).prix(); }
}

// Et maintenant trions des motos(triangles):
public static void main(String[] args) {

    MotoTriable motos[] =
        {new MotoTriable("Honda", 900, 18350),
         new MotoTriable("Kawasaki", 750, 14500),
         new MotoTriable("Gagiva", 650, 28300)};

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

# Exercice : Type Abstrait de Données (TAD)

- Un TAD est une collection d'informations structurées et reliées entre elles selon un graphe relationnel établi grâce aux opérations effectuées sur ces données.
- Un TAD contient une description des propriétés générales et des opérations qui décrivent la structure de données.
- Un TAD permet de spécifier des structures de données d'une manière générale sans avoir la nécessité d'en connaître l'implantation, ce qui est une caractéristique de la notion d'abstraction.



# TAD rationnel

## TAD rationnel

Utilise :  $\mathbb{Z}$

Champs : (Num , Denom ) appartient à  $\mathbb{Z} \times \mathbb{Z}^*$

Opérations :

Num :  $\text{---} \rightarrow \text{rationnel}$

Denom :  $\text{---} \rightarrow \text{rationnel}$

Reduire :  $\text{rationnel} \text{---} \rightarrow \text{rationnel}$

$+$  :  $\text{rationnel} \times \text{rationnel} \text{---} \rightarrow \text{rationnel}$

$/$  :  $\text{rationnel} \times \text{rationnel} \text{---} \rightarrow \text{rationnel}$

$*$  :  $\text{rationnel} \times \text{rationnel} \text{---} \rightarrow \text{rationnel}$

$-$  :  $\text{rationnel} \times \text{rationnel} \text{---} \rightarrow \text{rationnel}$

$-$  :  $\text{rationnel} \text{---} \rightarrow \text{rationnel}$

AffectQ :  $\text{rationnel} \text{---} \rightarrow \text{rationnel}$

Préconditions :

$x / y$  def ssi  $y.\text{Num} \neq 0$

## Finrationnel

# Spécifications opérationnelles du TAD rationnel

- **Reduire** : rendre le rationnel irréductible en calculant le **pgcd** du numérateur et du dénominateur, puis diviser les deux termes par ce pgcd.
- **AffectQ** : affectation classique d'un rationnel dans un autre rationnel.
- **Normalise** : rend le rationnel irréductible et met son signe au numérateur en rendant son dénominateur positif.
- **+** : addition de deux nombres rationnels par la recherche du plus petit commun multiple des deux dénominateurs et mise de chacun des deux rationnels au même dénominateur.

# Spécifications opérationnelles du TAD rationnel

- $*$  : multiplication de deux nombres rationnels, par le produit des deux dénominateurs et le produit des deux numérateurs.
- $/$  : division de deux nombres rationnels, par le produit du premier par l'inverse du second.
- $-$  : **(unaire)** renvoie l'opposé d'un rationnel dans un autre rationnel.
- $-$  : **(binaire)** soustraction de deux nombres rationnels par addition de l'opposé du second.

# TAD complexe

## TAD complexe

Utilise : **rationnel**

Champs : (part\_reel , part\_imag ) appartient à **rationnel** x **rationnel**

Opérations :

part\_reel : ---> **rationnel**

part\_imag : ---> **rationnel**

Charger : **rationnel** x **rationnel** ---> **complexe**

AffectC : **complexe** ---> **complexe**

+ : **complexe** x **complexe** ---> **complexe**

\* : **complexe** x **complexe** ---> **complexe**

- : **complexe** x **complexe** ---> **complexe**

- : **complexe** ---> **complexe**

Préconditions :

*aucune*

## Fin complexe

# Spécifications opérationnelles du TAD complexe

- **Charger** : remplit les deux champs `part_reel` et `part_imag` d'un nombre complexe.
- **AffectC** : affectation classique d'un complexe dans un autre.
- **+** : addition de 2 nombres complexes spécif. mathématique classique :  
 $z1 = x+iy$  et  $z2 = x'+iy' \Rightarrow z1+z2 = (x+x')+(y+y')i$ .

# Spécifications opérationnelles du TAD complexe

- $*$  : multiplication de 2 nombres complexes spécif. mathématique classique :  
 $z1 = x+iy$  et  $z2 = x'+iy' \Rightarrow z1 * z2 = (x.x' - y.y') + (x.y' + x'.y)i$ .
- $-$  : (binaire) soustraction de 2 nombres complexes spécif. mathématique classique :  
 $z1 = x+iy$  et  $z2 = x'+iy' \Rightarrow z1 - z2 = (x-x') + (y-y')i$ .
- $-$  : (unaire) pour un nombre complexe  $x+iy$  cet opérateur renvoie son opposé :  $-x -iy$

# Travail à effectuer

- Objectif : implanter les nombres rationnels et complexes
- **Construire les classes Rationnel et Complexe**
- **Adjoindre un mécanisme de robustesse par exception pour chacune des deux classes.**

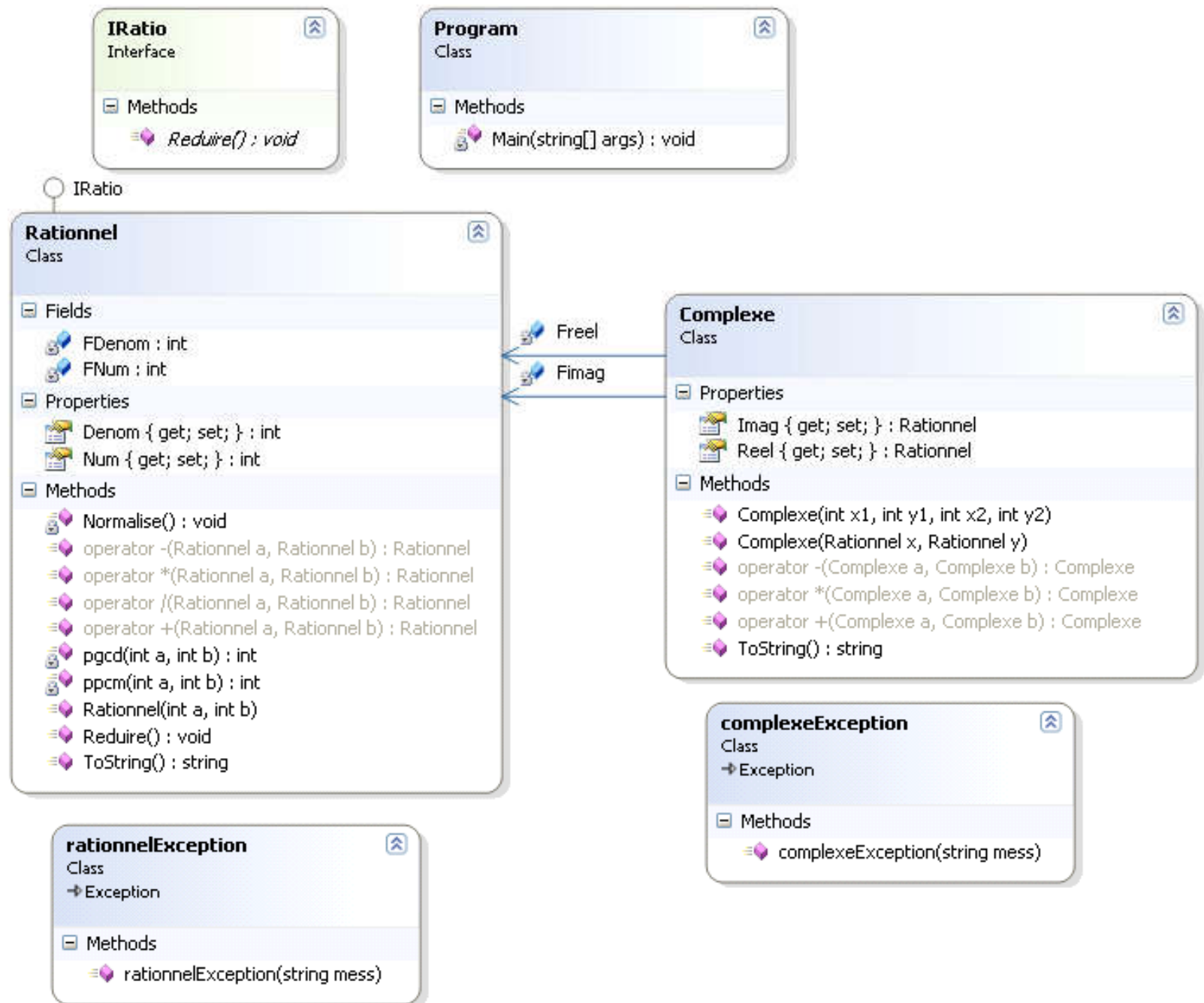
# Les classes de gestion d'exception

- Les classes `complexeException` et `rationnelException` :
- Ces classes servent à instancier une exception dans le cas du dénominateur nul dans une fraction et doivent s'adapter aux circonstances de survenues de la nullité du dit dénominateur en envoyant un message différent dans chaque cas.



# Exemples d'exception

- L'instruction qui suit doit engendrer une exception du type `rationnelException` :  
`z1 = new Complexe(new Rationnel(12, 0), new Rationnel(6, 9));`  
avec le message suivant : **"Denominateur nul (poursuite du calcul impossible)."**
- L'instruction qui suit doit engendrer une exception du type `rationnelException` : `z1.Reel.Denom = 0;`  
avec le message suivant : **"Denominateur nul (valeur 1 par défaut)."**
- L'instruction qui suit doit engendrer une exception du type `complexeException`: `z1 = new Complexe(12, 0, 6, 9);`  
avec le message suivant : **"Un coefficient rationnel n'a pas pu être réduit (denom = 0)."**



# **Introduction au Design Patterns**

# Plan 2

- Objectifs
- Définitions
- Avantages & Inconvénients
- Catégories de patterns
- Catégories de Design Patterns
- Portée des design patterns

# Objectifs

## Modularité

- Facilité de gestion (technologie objet)

## Cohésion

- Degré avec lequel les tâches réalisées par un seul module sont fonctionnellement reliées
- Une forte cohésion est une bonne qualité

## Couplage


- Degré d'interaction entre les modules dans le système
- Un couplage "lâche" est une bonne qualité

## Réutilisabilité

- Bibliothèques, frameworks (cadres)

# Cohésion : "mauvais" exemple

```
public class GameBoard {  
    public GamePiece[ ][ ] getState() { ... }  
    // Méthode copiant la grille dans un tableau temporaire, résultat de l'appel de la méthode.  
    public Player isWinner() { ... }  
    // vérifie l'état du jeu pour savoir s'il existe un gagnant, dont la référence est retournée.  
    // Null est retourné si aucun gagnant.  
    public boolean isTie() { ... }  
    //retourne true si aucun déplacement ne peut être effectué, false sinon.  
    public void display () { ... }  
    // affichage du contenu du jeu. Espaces blancs affichés pour chacune des  
    // références nulles  
}
```



**GameBoard est responsable des règles du jeu et de l'affichage**

# Cohésion : "bon" exemple

```
public class GameBoard {  
    public GamePiece[ ][ ] getState() { ... }  
    public Player isWinner() { ... }  
    public boolean isTie() { ... }  
}  
  
public class BoardDisplay {  
    public void displayBoard (GameBoard gb) { ... }  
    // affichage du contenu du jeu. Espaces blancs affichés pour chacune des  
    // références nulles.  
}
```

# Couplage : exemple

```
void initArray(int[] iGradeArray, int nStudents) {  
    int i;  
    for (i = 0; i < nStudents; i++) {  
        iGradeArray[i] = 0;  
    }  
}
```

Couplage entre client  
et initArray par le  
Paramètre "nStudents"

```
void initArray(int[ ] iGradeArray) {  
    int i;  
    for (i=0; i < iGradeArray.length; i++) {  
        iGradeArray[i] = 0;  
    }  
}
```

Couplage faible  
(et meilleure fiabilité)  
au travers de l'utilisation  
de l'attribut "length"



# Forte Cohésion

- Patron de conception Catégorie: GRASP
- Avoir des classes ayant des responsabilités spécialisées
- La cohésion mesure la compréhensibilité des classes. Une classe doit avoir des responsabilités cohérentes, et ne doit pas avoir des responsabilités trop variées. Une classe ayant des responsabilités non cohérentes est difficile à comprendre et à maintenir.
- La forte cohésion favorise :
  - la compréhension de la classe,
  - la maintenance de la classe,
  - la réutilisation des classes ou modules.

# Faible couplage

- Patron de conception Catégorie: GRASP
- Affaiblir la dépendances entre classes et modules
- Le couplage mesure la dépendance entre des classes ou des modules. Le faible couplage favorise :
  - la faible dépendance entre les classes,
  - la réduction de l'impact des changements dans une classe,
  - la réutilisation des classes ou modules.
- Pour affaiblir le couplage, il faut :
  - diminuer la quantité de paramètres passés entre les modules,
  - éviter d'utiliser des variables globales (par exemple, si une mauvaise valeur est assignée, détecter la fonction/classe incorrecte est plus difficile), il vaut mieux passer les valeurs en paramètres.

# Faible couplage (Exemple de Facturation)

- Dans un logiciel de gestion de vente, nous avons les classes suivantes :
  - **Facture** Contient un ensemble de produit facturé et est associée à un mode de paiement, **Paiement** Décrit un mode de paiement (espèces, chèque, carte bancaire, à crédit, ...), **Client** Effectue les commandes.
- On ajoute une méthode payer() à la classe Client. On étudie le couplage dans les deux cas suivants :
  1. La méthode payer() crée une instance de Paiement et l'assigne à l'objet Facture.
  2. La méthode payer() délègue l'action à la classe Facture qui crée une instance de Paiement et l'assigne.
- Le couplage est plus faible dans le deuxième cas car la méthode payer() de la classe Client n'a pas besoin de savoir qu'il existe une classe Paiement, c'est à dire qu'elle **ne dépend pas** de l'existence ou non de cette classe.

# Principes de conception

- Programmer une interface plus qu'une implémentation
- Utiliser des classes abstraites (interfaces en Java) pour définir des interfaces communes à un ensemble de classes
- Déclarer les paramètres comme instances de la classe abstraite plutôt que comme instances de classes particulières
- Préférer la composition d'objet à l'héritage de classes

# Définitions

- « **C. Alexander** » : Un patron décrit à la fois un **problème** qui se produit très fréquemment dans l'environnement et l'architecture de la **solution** à ce problème de telle façon que l'on puisse **utiliser** cette solution des milliers de fois sans jamais l'**adapter** deux fois de la même manière.
- « **Aarsten** » : Un groupe d'objets coopérants liés par des relations et des règles qui expriment les liens entre un contexte, un problème de conception et sa solution
- Traductions : patrons de conception, schémas de conception

# Ce que ce n'est pas . . .

- **Une règle** : un pattern ne peut pas s'appliquer mécaniquement ;
- **Une méthode** : ne guide pas une prise de décision ; un pattern est la décision prise.
- Les patrons sont des composants **logiques** décrits indépendamment d'un langage donné (solution exprimée par des modèles semi-formels)

# Avantages

- Un vocabulaire commun ;
- Capitalisation de l'expérience ;
- Un niveau d'abstraction plus élevé qui permet d'élaborer des constructions logicielles de meilleure qualité ;
- Réduire la complexité ;
- Guide/catalogue de solutions.

# Inconvénients

- Effort de synthèse ; reconnaître, abstraire . . . ;
- Apprentissage, expérience ;
- Nombreux...
  - lesquels sont identiques ?
  - De niveaux différents ... des patterns s'appuient sur d'autres...



# Catégories de Patterns

## Architectural Patterns

- schémas d'organisation structurelle de logiciels (pipes, filters, brokers, blackboard, MVC, SOA...)

## Design Patterns

- caractéristiques clés d'une structure de conception commune à plusieurs applications,
- Portée plus limitée que les « architectural patterns »

## Idioms ou coding patterns

- solution liée à un langage particulier

# Catégories de Patterns (2)

## Anti-patterns

- mauvaise solution ou comment sortir d'une mauvaise solution

## Organizational patterns

- Schémas d'organisation de tout ce qui entoure le développement d'un logiciel (humains)

# Patrons de conception GRASP

- GRASP signifie *General Responsibility Assignment Software Patterns/Principles*.
- Ces patrons de conception donnent des conseils généraux sur l'assignation de responsabilité aux classes et objets dans une application. Ils sont issus du bon sens de conception, intuitifs et s'appliquent de manière plus générale.
- Une responsabilité est vue au sens conception (exemples : création, détention de l'information, ...) :
  - elle est relative aux méthodes et données des classes,
  - elle est assurée à l'aide d'une ou plusieurs méthodes,
  - elle peut s'étendre sur plusieurs classes.
- En UML, l'assignation de responsabilité peut être appliquée à la conception des diagrammes de collaboration.

# Catégories de Design Patterns

- **Création**

- Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés
- Isolation du code relatif à la création, à l'initialisation afin de rendre l'application indépendante de ces aspects

- **Structure**

- Description de la manière dont doivent être connectés des objets de l'application afin de rendre ces connections indépendantes des évolutions futures de l'application
- Découplage de l'interface et de l'implémentation de classes et d'objets

- **Comportement**

- Description de comportements d'interaction entre objets
- Gestion des interactions dynamiques entre des classes et des objets

# Patterns de création

- ***Fabrique (Factory Method)*** : constructeur virtuel
  - Crée une instance parmi plusieurs classes dérivées
- ***Fabrique abstraite (Abstract Factory)*** : fabrique d'objets
  - Crée une instance parmi plusieurs familles de classes
- ***Monteur (Builder)*** :
  - Sépare la construction de l'objet de sa représentation
- ***Prototype (Prototype)*** : construction à partir de prototypes
  - Permet de copier (cloner) une instance entière d'un objet
- ***Singleton (Singleton)*** : instance unique
  - Une seule et unique instance peut être créée

# Patterns de structure (1/2)

- ***Adaptateur (Adapter)*** :
  - rendre un objet conformant à un autre
  - Obtenir un objet qui permet d'en utiliser un autre en conformité avec une certaine interface
  - Assortir des interfaces de différentes classes
- ***Pont (Bridge)*** :
  - Découplage de l'abstraction de l'implémentation pour lier une abstraction à une ou plusieurs implantations
  - Sépare l'interface d'un objet de son implémentation
- ***Objet composite (Composite)*** :
  - Une structure de type arbre pour les objets simples et composés
  - objets composés d'objets homogènes

# Patterns de structure (2/2)

- ***Décorateur (Decorator)*** :
  - Ajoute aux objets des responsabilités (services) d'une manière dynamique
  - ajout dynamique de comportements ou d'états à un composant ;
- ***Façade (Facade)*** :
  - Cache une structure complexe
  - Interface simple pour un système complexe
- ***Poids-mouche ou poids-plume (Flyweight)*** :
  - petits objets destinés à être partagés
- ***Proxy (Proxy)*** :
  - un objet en masque un autre
  - utiliser un composant intermédiaire pour accéder à un autre composant

# Patterns de comportement (1/3)

- ***Chaîne de responsabilité (Chain of responsibility) :***
  - Une manière pour passer une requête entre une chaîne d'objets.
- ***Commande (Command) :***
  - Encapsule une requête sous forme d'un objet.
- ***Interpréteur (Interpreter) :***
  - Un moyen pour inclure (ajouter) les éléments d'un langage dans un programme
- ***Itérateur (Iterator) :***
  - Accéder séquentiellement aux éléments d'une collection
- ***Médiateur (Mediator) :***
  - Définir une communication simplifiée entre les classes



# Patterns de comportement (2/3)

- *Mémento (Memento)* :
  - Capturer et restaurer l'état interne d'un objet
- *Observateur (Observer)*
  - Un moyen pour signaler un changement à un nombre de classes
  - Relation entre des vues et un composant observé
- *État (State)* :
  - Altérer le comportement d'un objet lorsque son état change
- *Stratégie (Strategy)* :
  - Encapsule un algorithme à l'intérieur d'une classe

# Patterns de comportement (3/3)

- *Patron de méthode (Template Method)* :
  - Différer les étapes exactes d'un algorithme à une sous classe
- *Visiteur (Visitor)* :
  - Définir une nouvelle opération à une classe sans changement

# Portée des Design Patterns (1/2)

## 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

# Portée des Design Patterns (2/2)

		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

# Présentation d'un Design Pattern

- **Nom du pattern** : utilisé pour décrire le pattern, ses solutions et les conséquences en un mot ou deux
- **Problème** : description des conditions d'applications. Explication du problème et de son contexte
- **Solution** : description des éléments (objets, relations, responsabilités, collaboration) permettant de concevoir la solution au problème ; utilisation de diagrammes de classes, de séquences, ...  
vision statique ET dynamique de la solution
- **Conséquences** : description des résultats (effets induits) de l'application du pattern sur le système (effets positifs ET négatifs)

# **Patterns de création**

# Plan 3

- Présentation
- Factory method
- Abstract Factory
- Builder
- Prototype
- Singleton

# Présentation (1/3)

- Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés
- Rendre le système indépendant de la manière dont les objets sont créés, composés et représentés
  - Encapsulation de la connaissance des classes concrètes à utiliser
  - Cacher la manière dont les instances sont créées et combinées
- Permettre *dynamiquement ou statiquement de préciser* **QUOI** (l'objet), **QUI** (l'acteur), **COMMENT** (la manière) et **QUAND** (le moment) de la création



# Présentation (2/3)


- *Fabrique (Factory Method)* : constructeur virtuel
  - Crée une instance parmi plusieurs classes dérivées
- *Fabrique abstraite (Abstract Factory)* : fabrique d'objets
  - Crée une instance parmi plusieurs familles de classes
- *Monteur (Builder)* :
  - Sépare la construction de l'objet de sa représentation
- *Prototype (Prototype)* : construction à partir de prototypes
  - Permet de copier (cloner) une instance entière d'un objet
- *Singleton (Singleton)* : instance unique
  - Une seule et unique instance peut être créée

# Présentation (3/3)

Deux types de motifs

1. Motifs de création de classe (utilisation de l'héritage) :  
Factory Method
2. Motifs de création d'objets (délégation de la construction à un autre objet) : Abstract Factory, Builder, Prototype

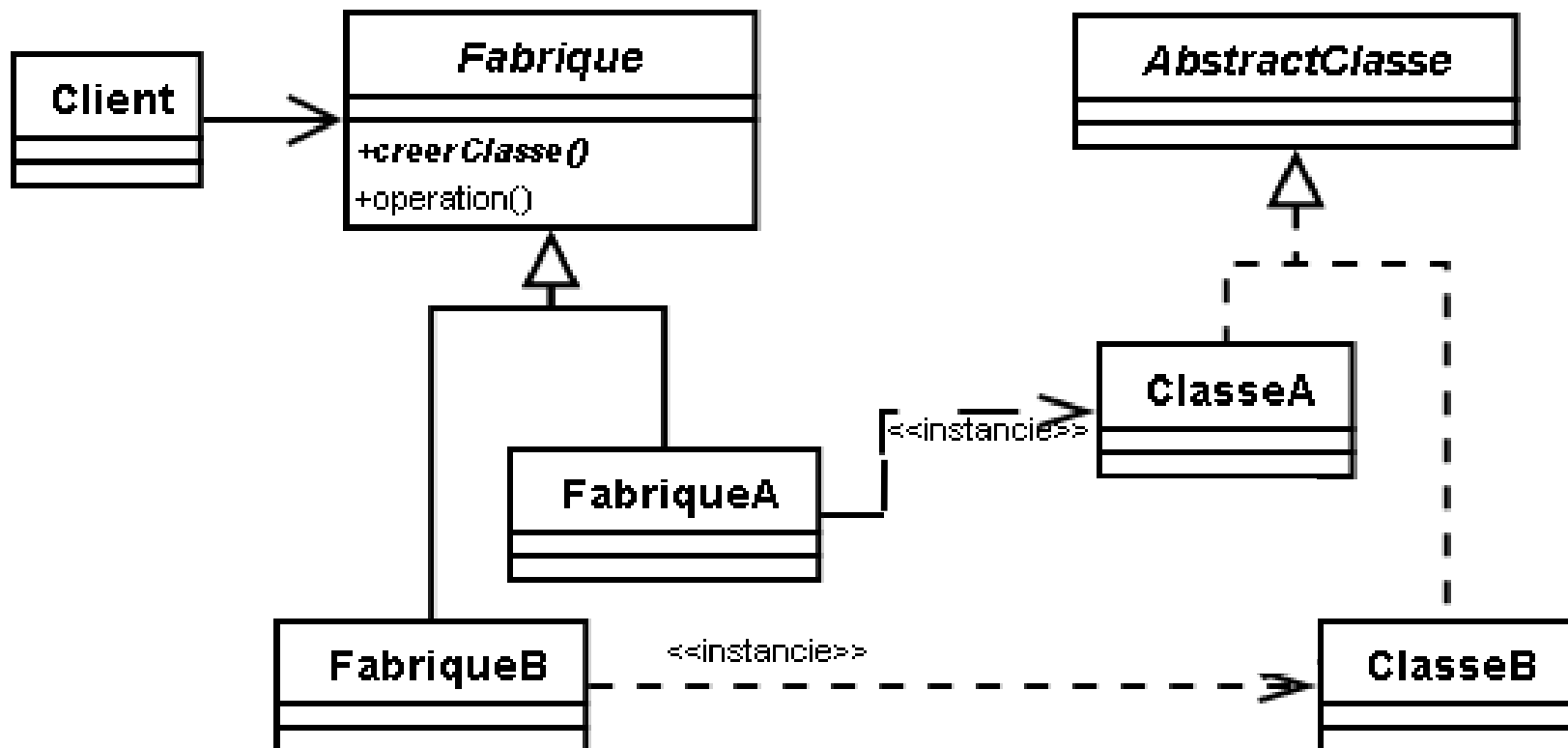
# Factory Method

- **Definition :** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Frequency of use:** high 
- **Objectifs**
  - Définir une interface pour la création d'un objet, mais laisser les sous-classes de décider quelle classe à instancier.
  - Déléguer l'instanciation aux sous-classes.
- **Résultat :** Le Design Pattern permet d'isoler l'instanciation d'une classe concrète.

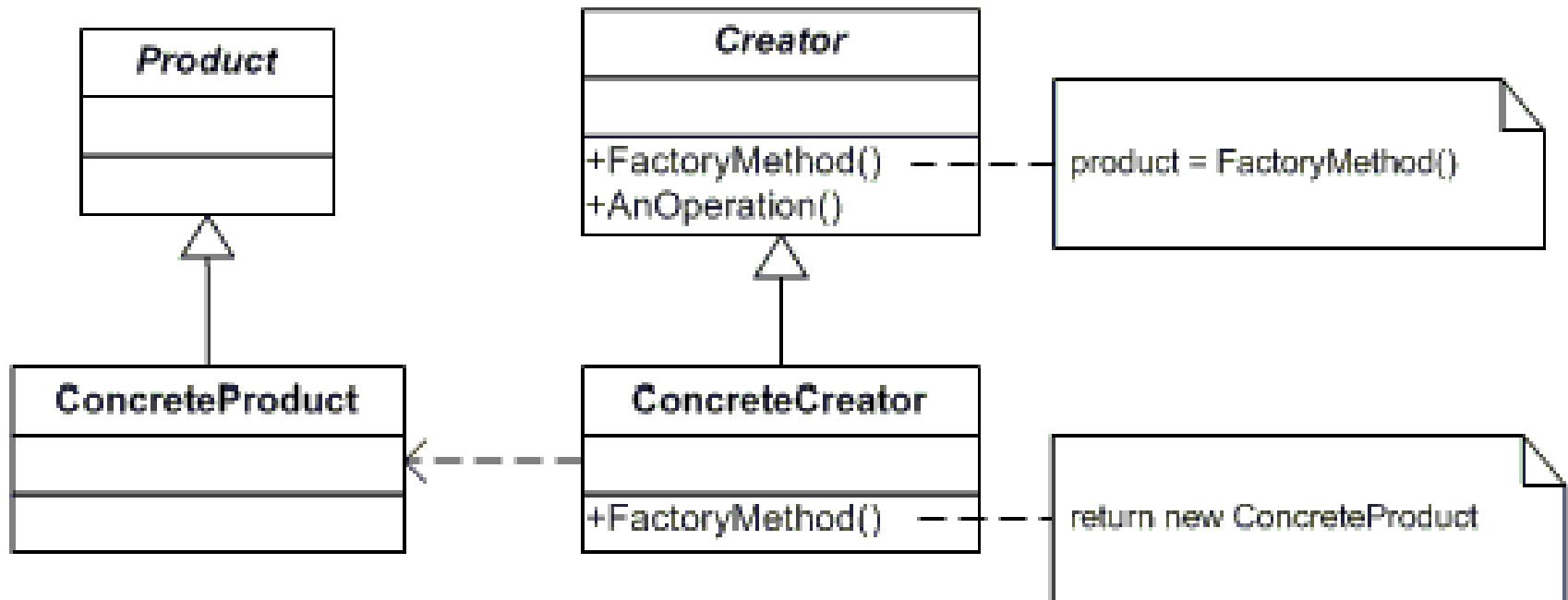
# Utilisation

- Dans le fonctionnement d'une classe, il est nécessaire de créer une instance. Mais, au niveau de cette classe, on ne connaît pas la classe exacte à instancier.
- Cela peut être le cas d'une classe réalisant une sauvegarde dans un flux sortant, mais ne sachant pas s'il s'agit d'un fichier ou d'une sortie sur le réseau.
- La classe possède une méthode qui retourne une instance (interface commune au fichier ou à la sortie sur le réseau). Les autres méthodes de la classe peuvent effectuer les opérations souhaitées sur l'instance (écriture, fermeture). Les sous-classes déterminent la classe de l'instance créée (fichier, sortie sur le réseau).
- Une variante du Pattern existe : la méthode de création choisit la classe de l'instance à créer en fonction de paramètres en entrée de la méthode ou selon des variables de contexte.

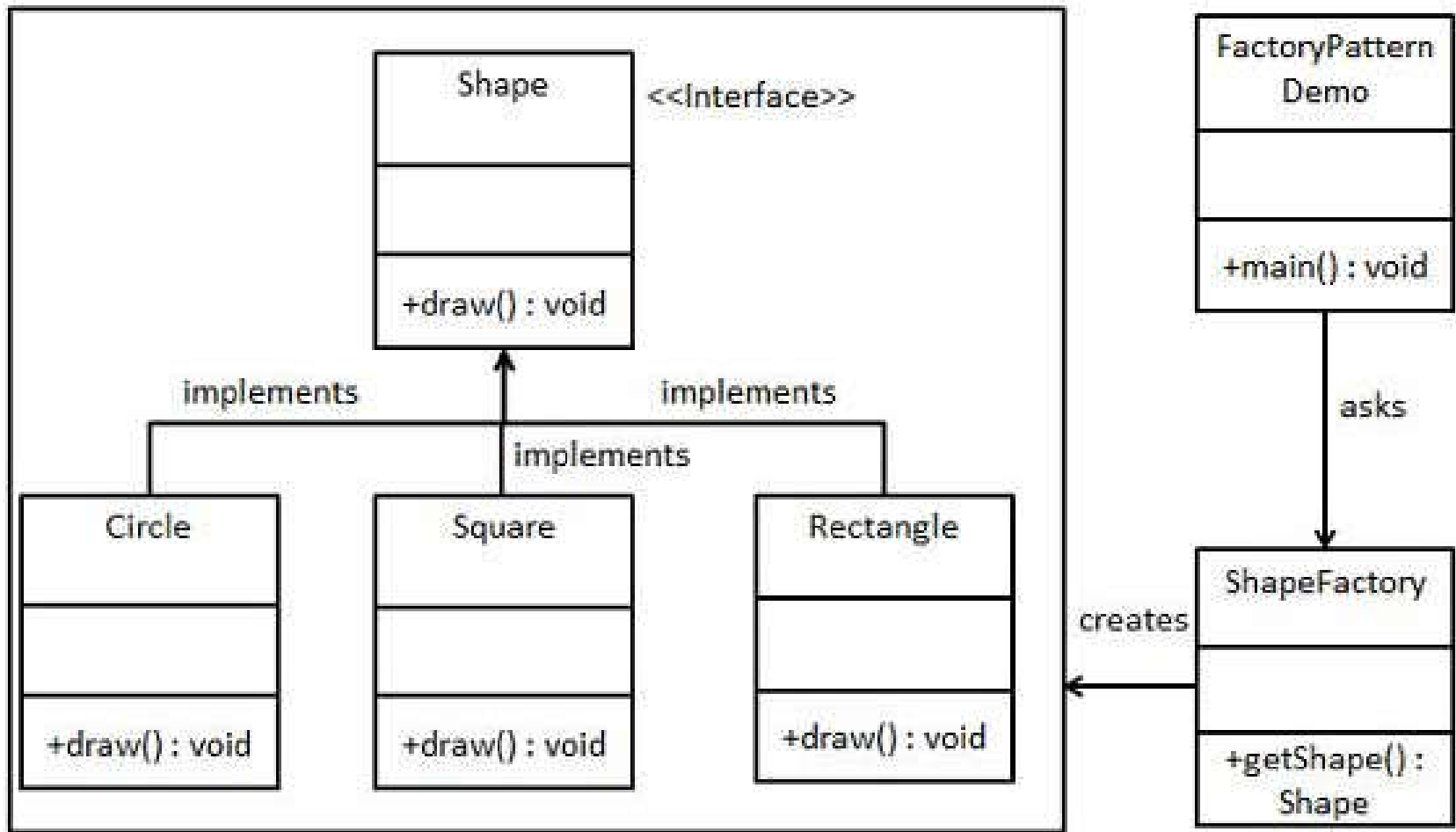
# Diagramme UML (1/2)



# Diagramme UML (2/2)



# Exemple 1



## Example 2

```
class Namer {  
    //a simple class to take a string apart into two names  
    protected String last; //store last name here  
    protected String first; //store first name here  
  
    public String getFirst() {  
        return first;           //return first name  
    }  
    public String getLast() {  
        return last;            //return last name  
    }  
}
```



# Les sous classes (1)

```
class FirstFirst extends Namer {    //split first last
    public FirstFirst(String s) {
        int i = s.lastIndexOf(" "); //find sep space
        if (i > 0) {
            //left is first name
            first = s.substring(0, i).trim();
            //right is last name
            last = s.substring(i+1).trim();
        }
        else {
            first = ""; // put all in last name
            last = s;    // if no space
        }
    }
}
```

# Les sous classes (2)

```
class LastFirst extends Namer {                                //split last, first
    public LastFirst(String s) {
        int i = s.indexOf(",");                                //find comma
        if (i > 0) {
            //left is last name
            last = s.substring(0, i).trim();
            //right is first name
            first = s.substring(i + 1).trim();                }
        else {
            last = s;                                          // put all in last name
            first = "";                                         // if no comma      }
        }
    }
}
```

# Création du Factory

```
class NameFactory {  
    //returns an instance of LastFirst or FirstFirst  
    //depending on whether a comma is found  
    public Namer getNamer(String entry) {  
        int i = entry.indexOf(","); //comma determines name order  
        if (i>0)  
            return new LastFirst(entry); //return one class  
        else  
            return new FirstFirst(entry); //or the other  
    }  
}
```

# Utilisation du Factory




```
NameFactory nfactory = new NameFactory();  
private void computeName() {  
    Namer namer = nfactory.getNamer(entryField.getText());  
    txFirstName.setText(namer.getFirst());  
    txLastName.setText(namer.getLast());  
}
```

# Example 3

- **participants**
  - **Product (Page)**
    - defines the interface of objects the factory method creates
  - **ConcreteProduct (SkillsPage, EducationPage, ExperiencePage, IntroductionPage, ResultsPage, Conclusionpage, SummaryPage, BibliographyPage)**
    - implements the Product interface
  - **Creator (Document)**
    - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
    - may call the factory method to create a Product object.
  - **ConcreteCreator (Report, Resume)**
    - overrides the factory method to return an instance of a ConcreteProduct.

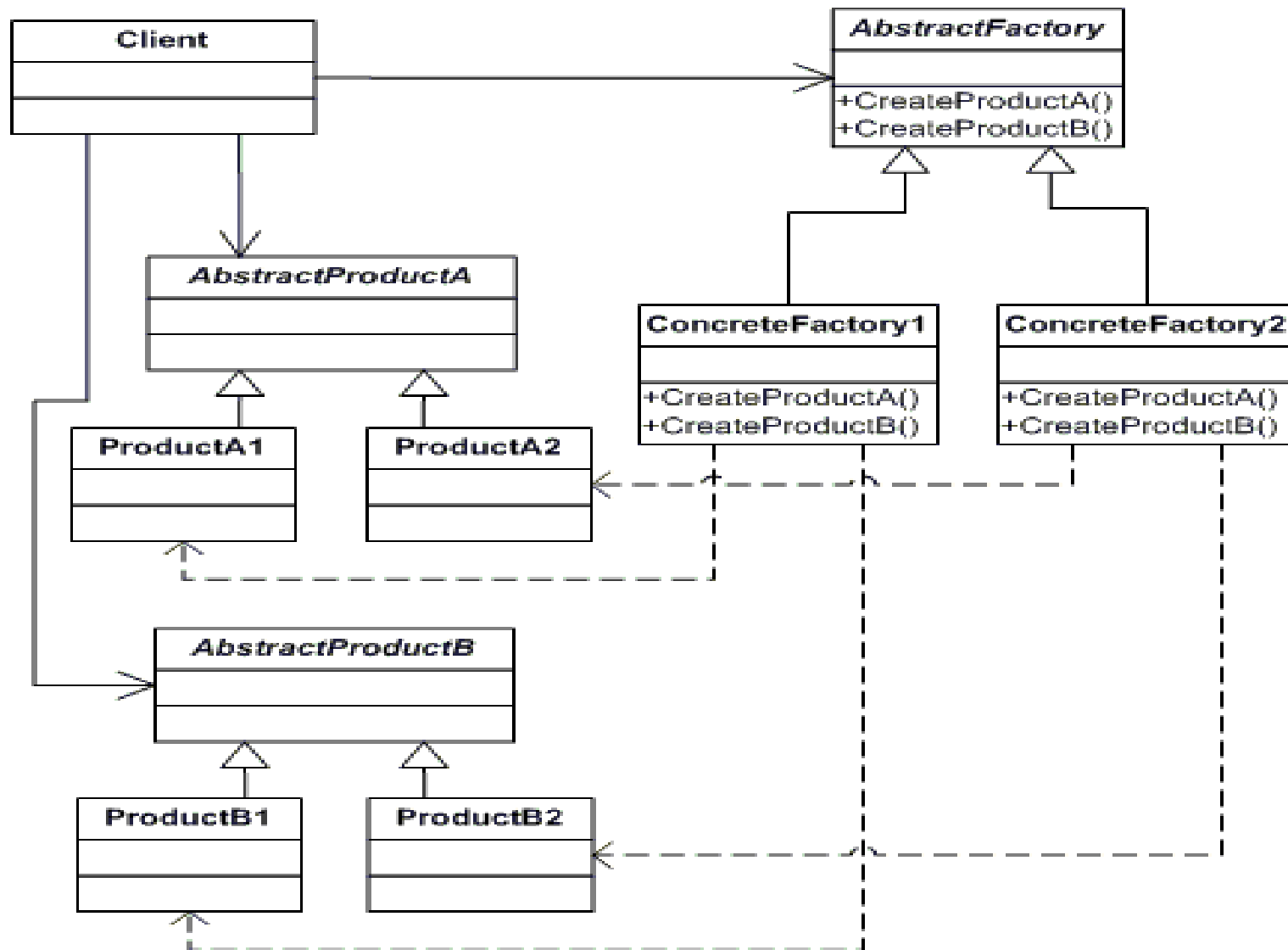
# Abstract Factory

- **Definition** : Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Frequency of use**: high 
- **Objectifs**: Fournir une interface pour créer des objets d'une même famille sans préciser leurs **classes concrètes**
- **Résultat** : Le Design Pattern permet d'isoler l'appartenance à une famille de classes.

# Utilisation

- Le système utilise des objets qui sont regroupés en famille. Selon certains critères, le système utilise les objets d'une famille ou d'une autre. Le système doit utiliser ensemble les objets d'une famille.
- Cela peut être le cas des éléments graphiques d'un **look and feel** : pour un look and feel donné, tous les graphiques créés doivent être de la même famille.
- La partie cliente manipulera les interfaces des objets ; ainsi il y aura une indépendance par rapport aux classes concrètes. Chaque fabrique concrète permet d'instancier une famille d'objets (éléments graphiques du même look and feel) ; ainsi la notion de famille d'objets est renforcée.

# Diagramme UML



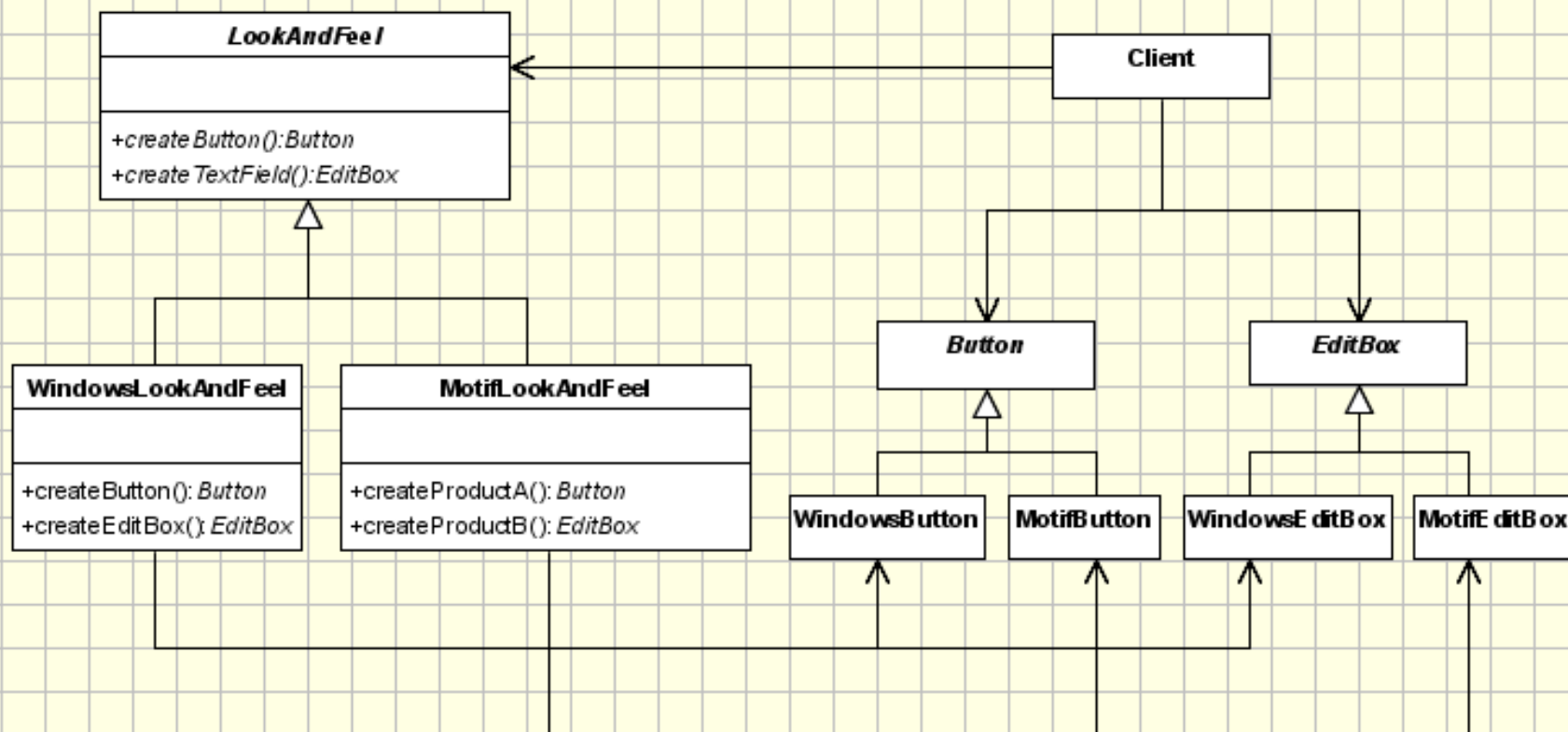


# Exemples en JDK

- `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`
- `javax.xml.xpath.XPathFactory#newInstance()`

# Exemple 1

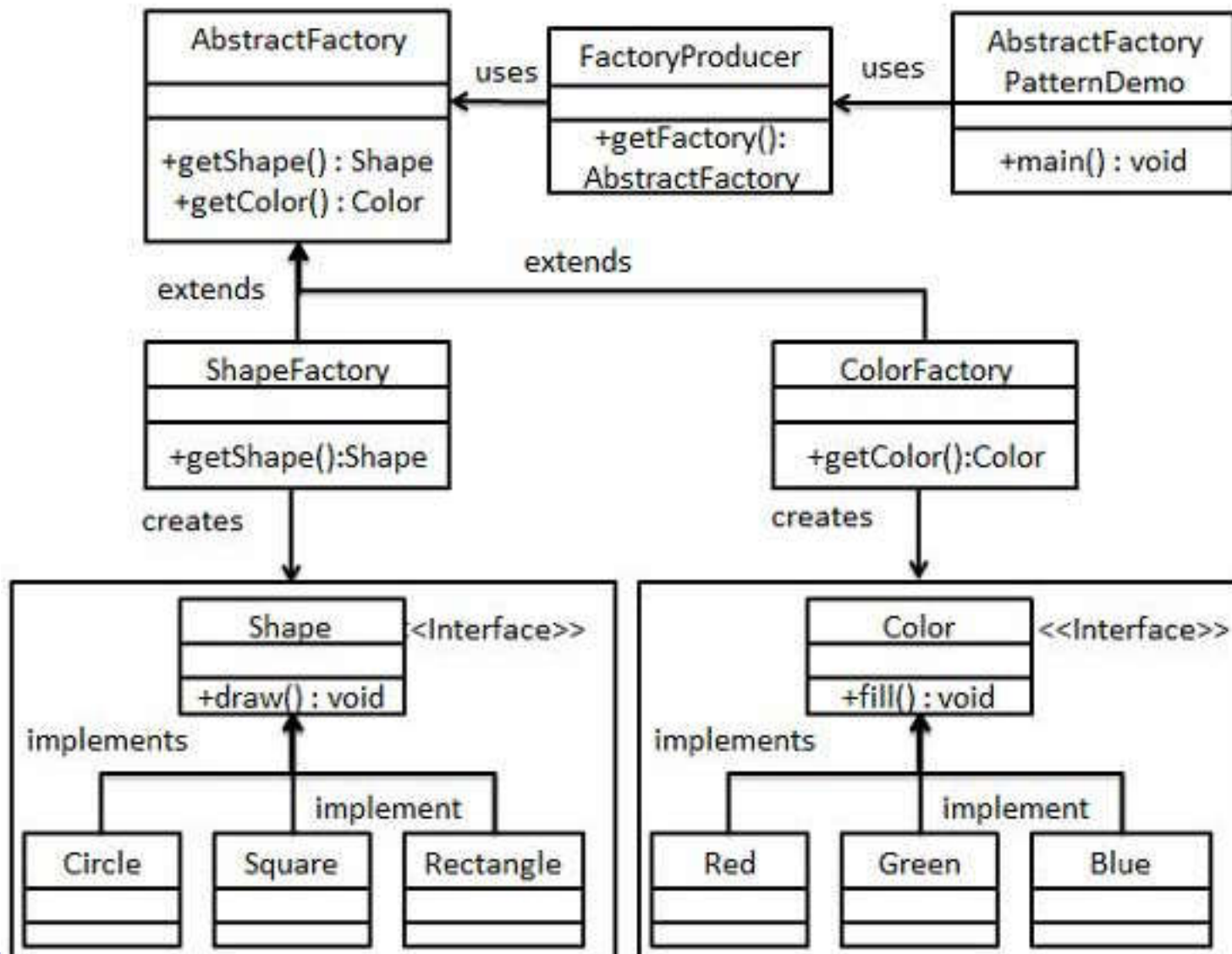
cd: Abstract Factory - Look & Feel Example - UML Class Diagram



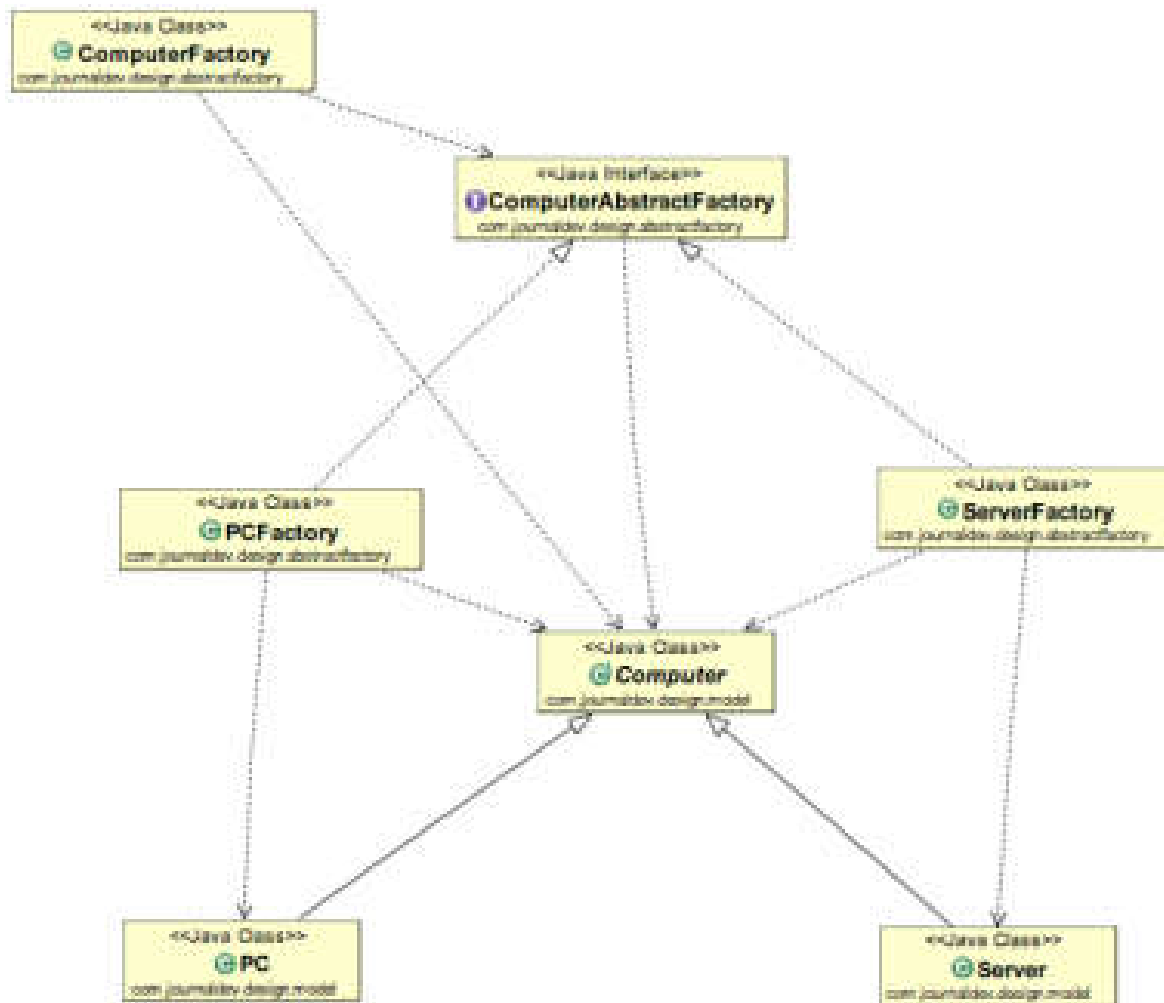
# Exemple 2

- **Participants**
  - **AbstractFactory (ContinentFactory)**
    - declares an interface for operations that create abstract products
  - **ConcreteFactory (AfricaFactory, AmericaFactory)**
    - implements the operations to create concrete product objects
  - **AbstractProduct (Herbivore, Carnivore)**
    - declares an interface for a type of product object
  - **Product (Wildebeest, Lion, Bison, Wolf)**
    - defines a product object to be created by the corresponding concrete factory
    - implements the AbstractProduct interface
  - **Client (AnimalWorld)**
    - uses interfaces declared by AbstractFactory and AbstractProduct classes


# Exemple 3



# Exemple 4



# Builder

- **Definition :** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Frequency of use:** medium low 
- **Objectifs :**
  - Séparer la construction d'un **objet complexe de sa représentation.**
  - Permettre d'obtenir des représentations différentes avec le même procédé de construction
- **Résultat :** Le Design Pattern permet d'isoler des variations de représentations d'un objet.

# Utilisation

- Le système doit instancier des objets complexes. Ces objets complexes peuvent avoir des représentations différentes.
- Cela peut être le cas des différentes fenêtres d'une IHM. Elles comportent des éléments similaires (titre, boutons), mais chacune avec des particularités (libellés, comportements).
- Afin d'obtenir des représentations différentes (fenêtres),
  - la partie cliente passe des monteurs différents au directeur.
  - Le directeur appellera des méthodes du monteur retournant les éléments (titre, bouton).
  - Chaque implémentation des méthodes des monteurs retourne des éléments avec des différences (libellés, comportements).

# Caractéristiques

## Problème

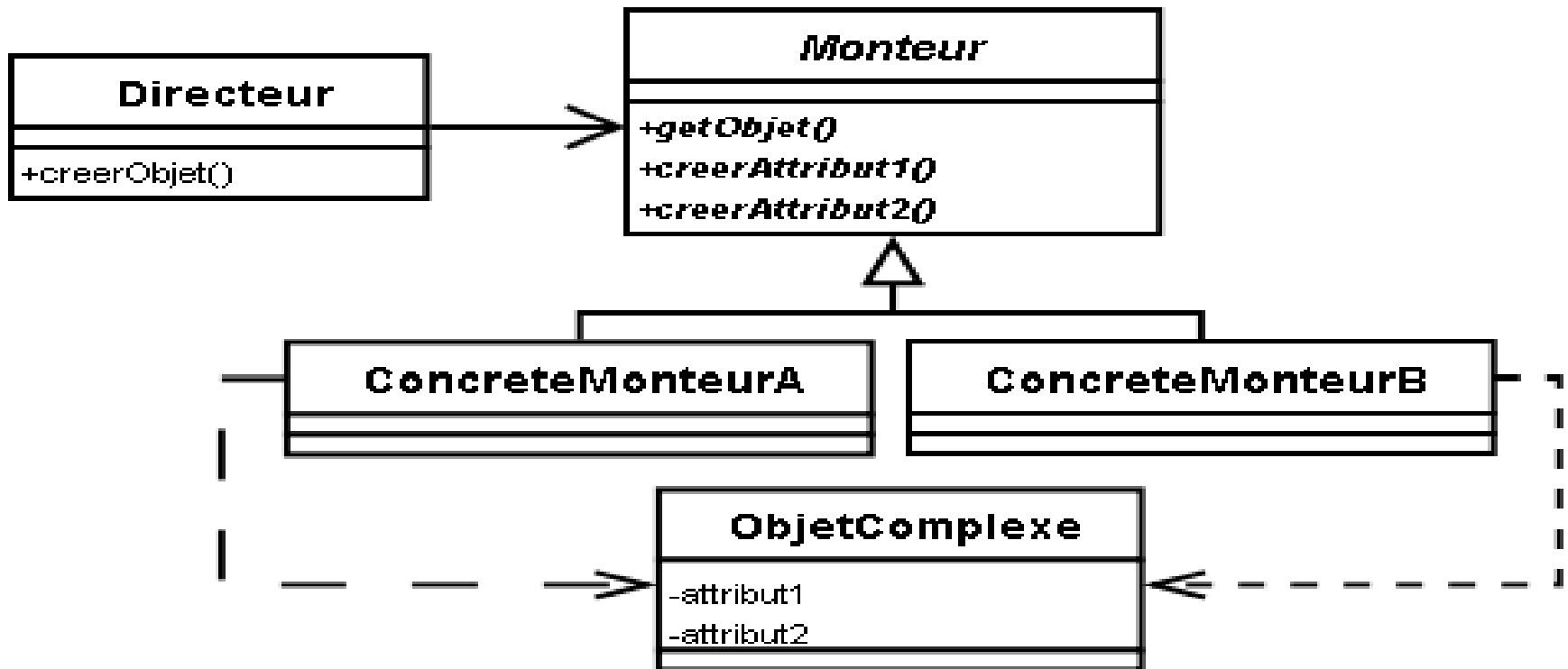
- ce motif est intéressant à utiliser lorsque l'algorithme de création d'un objet complexe doit être indépendant des constituants de l'objet et de leurs relations, ou lorsque différentes représentations de l'objet construit doivent être possibles

## Conséquences

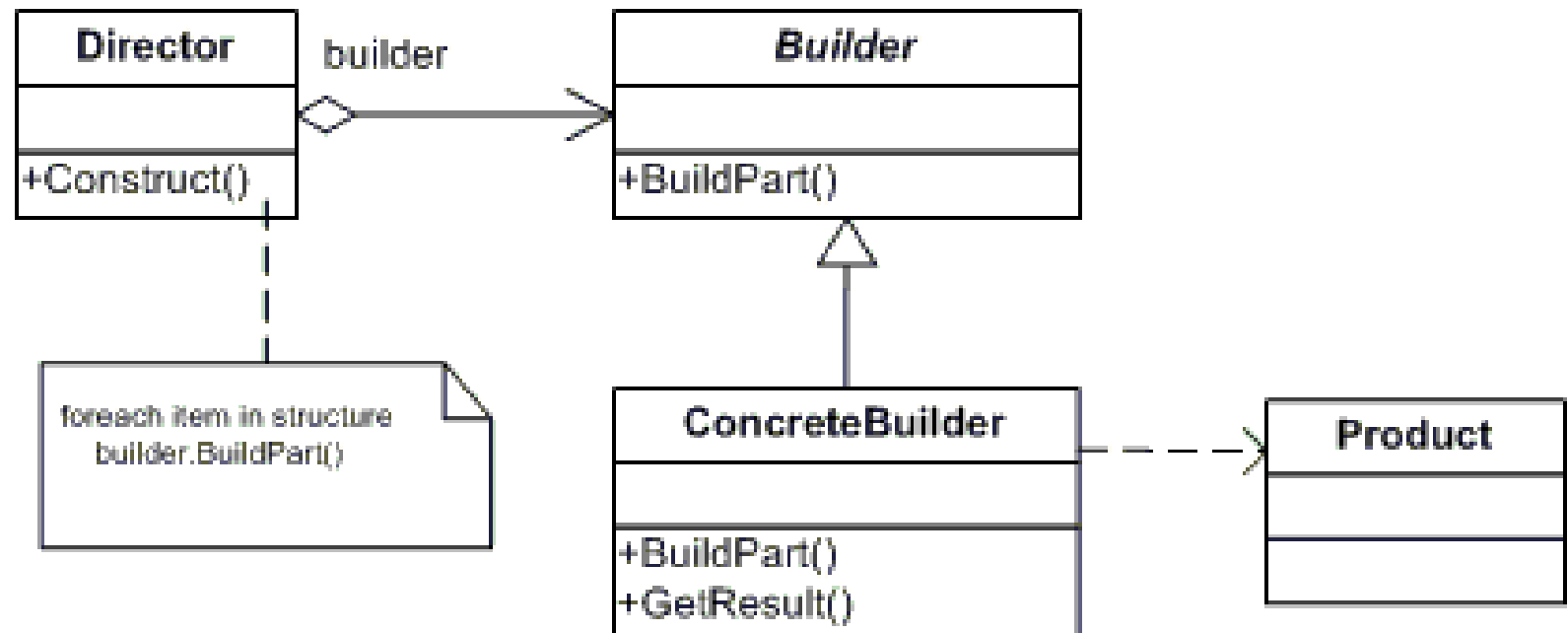
- Variation possible de la représentation interne d'un produit
  - l'implémentation des produits et de leurs composants est cachée au Director
  - Ainsi la construction d'un autre objet revient à définir un nouveau Builder
- Isolation du code de construction et du code de représentation du reste de l'application
- Meilleur contrôle du processus de construction



# Diagramme UML (1/2)



# Diagramme UML (2/2)



# Exemple 1

```
/* Produit */  
class Pizza {  
    private String pate = "";  
    private String sauce = "";  
    private String garniture = "";  
    public void setPate(String pate) { this.pate = pate; }  
    public void setSauce(String sauce) { this.sauce = sauce; }  
    public void setGarniture(String garniture) { this.garniture = garniture;  
    }  
}
```

# Exemple (suite)

```
/* Monteur */  
abstract class MonteurPizza {  
    protected Pizza pizza;  
    public Pizza getPizza() { return pizza; }  
    public void creerNouvellePizza()  
        { pizza = new Pizza(); }  
    public abstract void monterPate();  
    public abstract void monterSauce();  
    public abstract void monterGarniture();  
}
```

# Exemple (suite)

```
/* MonteurConcret */
```

```
class MonteurPizzaHawaii extends MonteurPizza {  
    public void monterPate() { pizza.setPate("croisée"); }  
    public void monterSauce() { pizza.setSauce("douce"); }  
    public void monterGarniture() {  
        pizza.setGarniture("jambon+ananas"); }  
}
```

```
/* MonteurConcret */
```

```
class MonteurPizzaPiquante extends MonteurPizza {  
    public void monterPate() { pizza.setPate("feuilletée"); }  
    public void monterSauce() { pizza.setSauce("piquante"); }  
    public void monterGarniture() {  
        pizza.setGarniture("pepperoni+salami"); }  
}
```

# Exemple (suite)

- `/* Directeur */`  
    `class Serveur {`  
        `private MonteurPizza monteurPizza;`  
        `public void setMonteurPizza(MonteurPizza mp) { monteurPizza`  
            `= mp; }`  
        `public Pizza getPizza() { return monteurPizza.getPizza(); }`  
        `public void construirePizza() {`  
            `monteurPizza.creerNouvellePizza();`  
            `monteurPizza.monterPate();`  
            `monteurPizza.monterSauce();`  
            `monteurPizza.monterGarniture();`  
        `}`  
    `}`

# Exemple (suite)

```
/* Un client commandant une pizza. */
class ExempleMonteur {
    public static void main(String[] args) {
        Serveur serveur = new Serveur();
        MonteurPizza monteurPizzaHawaii = new MonteurPizzaHawaii();
        MonteurPizza monteurPizzaPiquante = new MonteurPizzaPiquante();
        serveur.setMonteurPizza(monteurPizzaHawaii);
        serveur.construirePizza();
        Pizza pizza = serveur.getPizza();
    }
}
```

# Exemple 2

## Participants

- **Builder (VehicleBuilder)**
  - specifies an abstract interface for creating parts of a Product object
- **ConcreteBuilder (MotorCycleBuilder, CarBuilder, ScooterBuilder)**
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product
- **Director (Shop)**
  - constructs an object using the Builder interface
- **Product (Vehicle)**
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result



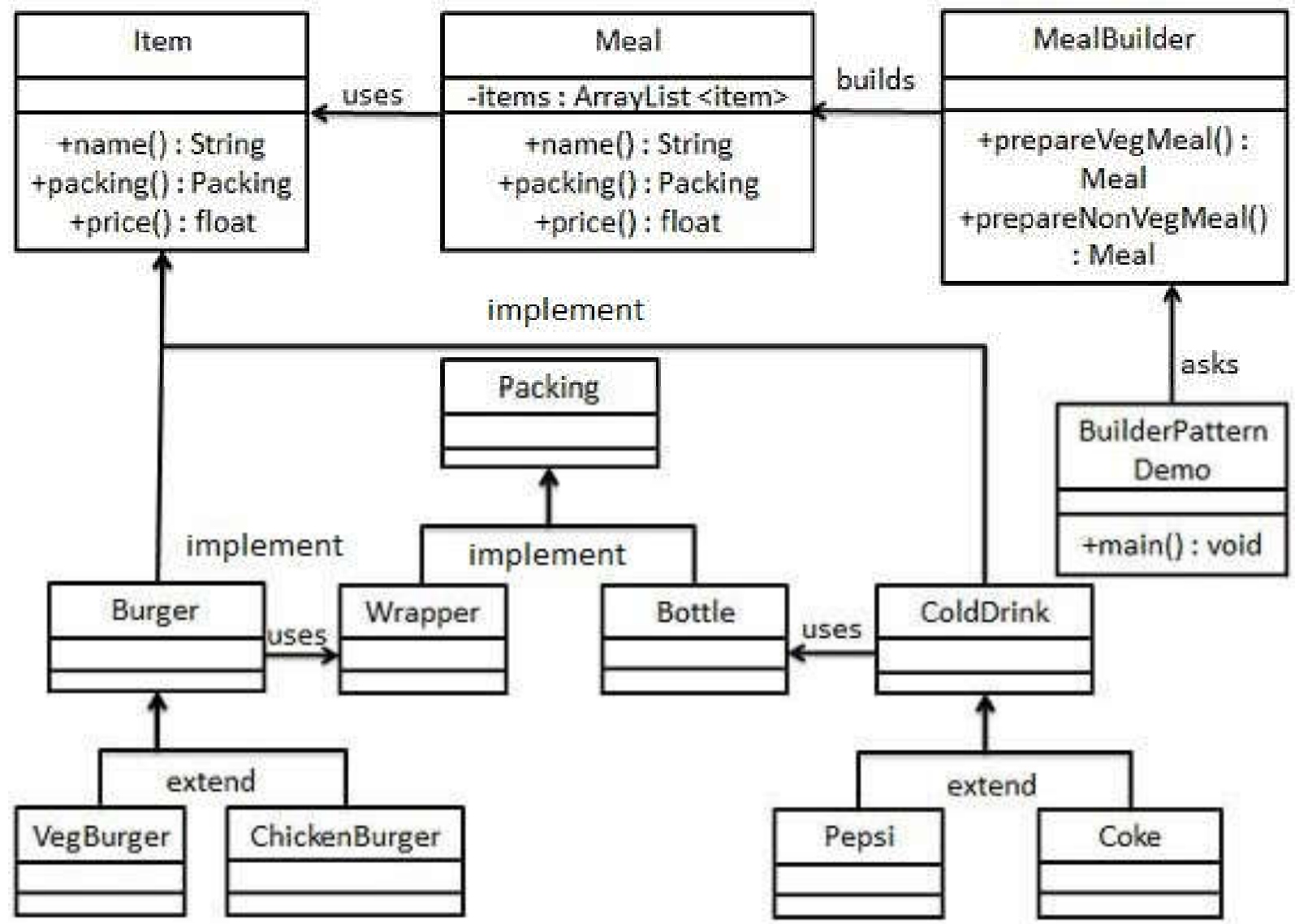
## Example 2 ( Test)

-----  
Vehicle Type: Scooter  
Frame : Scooter Frame  
Engine : none  
#Wheels: 2  
#Doors : 0


-----  
Vehicle Type: Car  
Frame : Car Frame  
Engine : 2500 cc  
#Wheels: 4  
#Doors : 4

-----  
Vehicle Type: MotorCycle  
Frame : MotorCycle Frame  
Engine : 500 cc  
#Wheels: 2  
#Doors : 0

# Exemple 3



# Prototype

- **Definition :** Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Frequency of use:** medium 

1	2	3	4	5
Red	Red	Red	Yellow	Yellow
- **Objectifs :**
  - Spécifier les genres d' objet à créer en utilisant une instance comme prototype.
  - Créer un nouvel objet en copiant ce prototype.
- **Résultat :** Le Design Pattern permet d'isoler l'appartenance à une classe.

# Utilisation

- Le système doit créer de nouvelles instances, mais il ignore de quelle classe. Il dispose cependant d'instances de la classe désirée.
- Cela peut être le cas d'un logiciel de DAO comportant un copier-coller. L'utilisateur sélectionne un élément graphique (cercle, rectangle, ...), mais la classe traitant la demande de copier-coller ne connaît pas la classe exacte de l'élément à copier.
- La solution est de disposer d'une duplication des instances (élément à copier : cercle, rectangle). La duplication peut être également intéressante pour les performances (la duplication est plus rapide que l'instanciation).

# Caractéristiques

## Problème

- Le système doit être indépendant de la manière dont ses produits sont créés, composés et représentés : les classes à instancier sont spécifiées au moment de l'exécution
- La présence de hiérarchies de Factory similaires aux hiérarchies de produits doivent être évitées. Les combinaisons d'instances sont en nombre limité

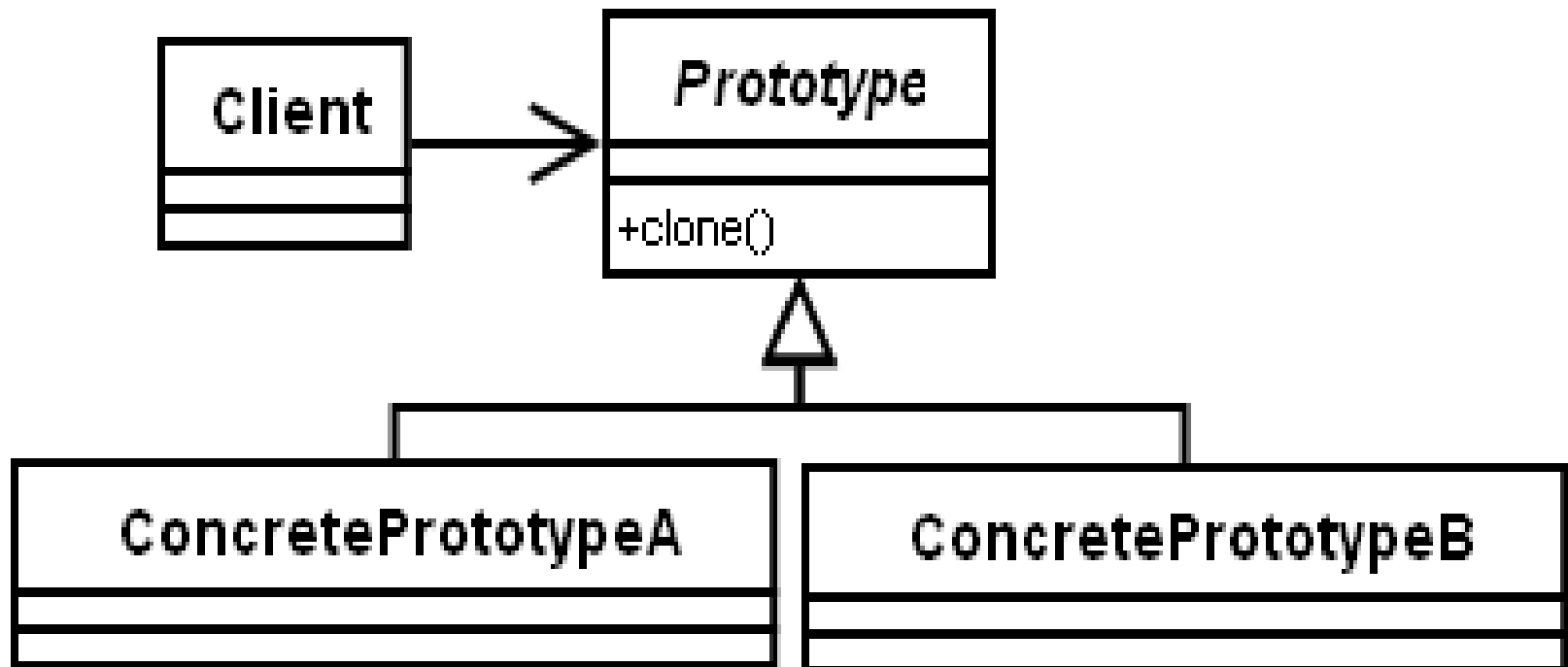
## Conséquences

- mêmes conséquences que Factory et Builder

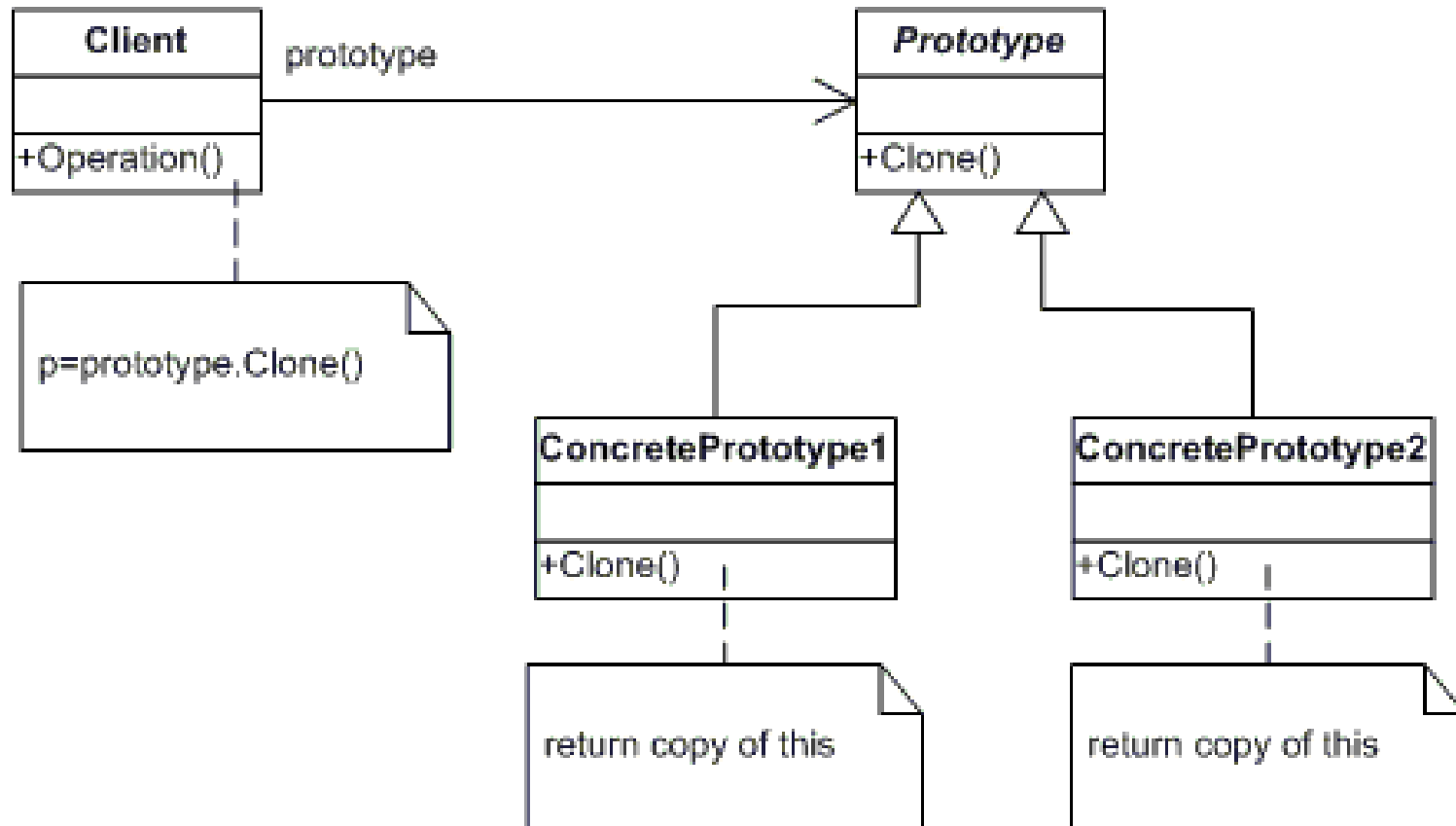
## Exemple

- `java.lang.Cloneable`

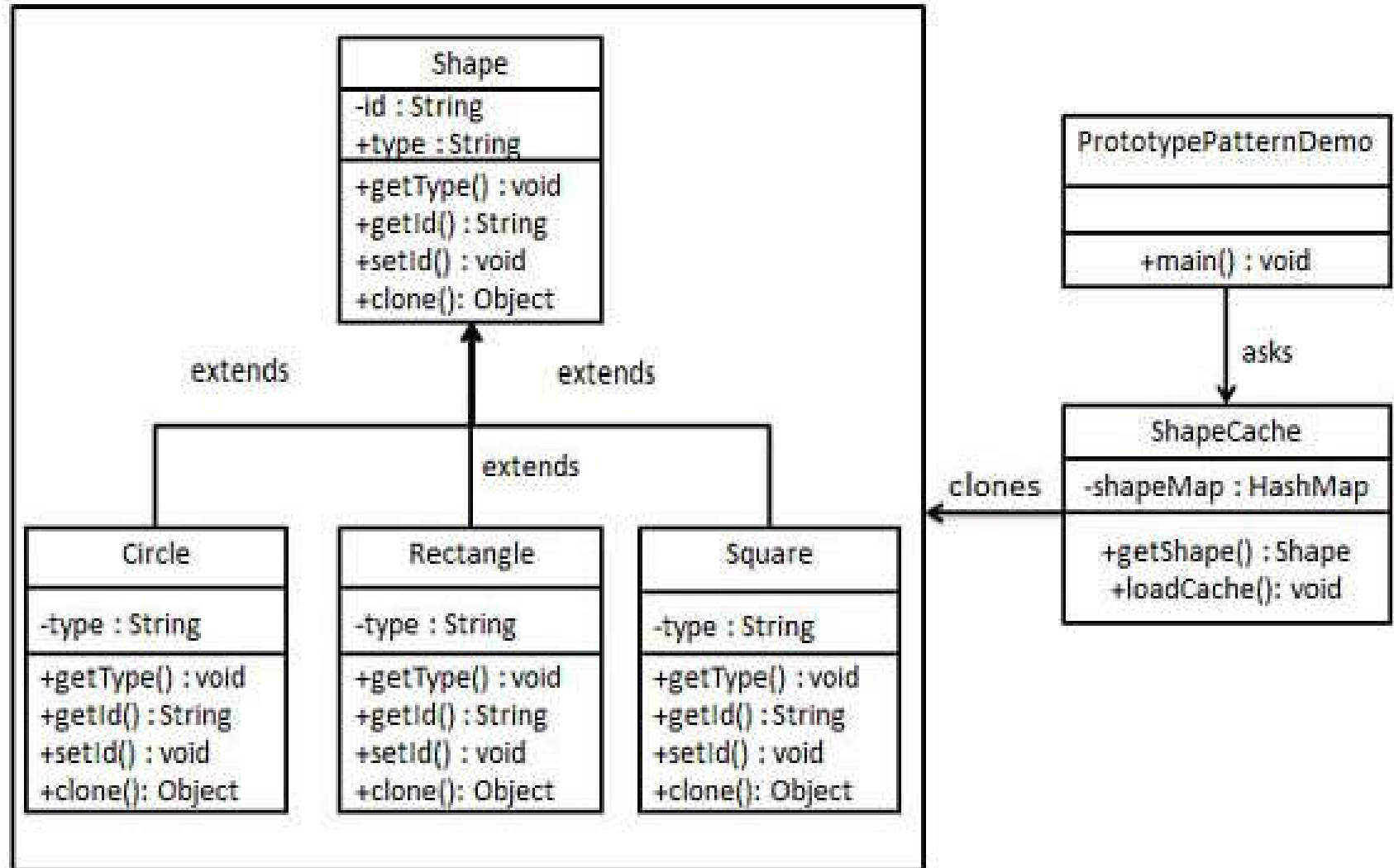
# Diagramme UML (1/2)



## Diagramme UML (2/2)



# Example1






## Exemple 2

### Participants:

- **Prototype (ColorPrototype)**
  - declares an interface for cloning itself
- **ConcretePrototype (Color)**
  - implements an operation for cloning itself
- **Client (ColorManager)**
  - creates a new object by asking a prototype to clone itself

# Singleton

- **Definition** : Ensure a class has only one instance and provide a global point of access to it.
- Frequency of use: medium high 

Singleton
-instance : Singleton
-Singleton() +Instance() : Singleton

# Singleton

## Problème

- avoir une seule instance d'une classe et pouvoir l'accéder et la manipuler facilement

## Solution

- une seule classe est nécessaire pour écrire ce motif

## Conséquences

- l'unicité de l'instance est complètement contrôlée par la classe elle même.
- Ce motif peut facilement être étendu pour permettre la création d'un nombre donné d'instances

## Example 1

```
class SingletonException extends RuntimeException
{
    //new exception type for singleton classes
    public SingletonException()
    {
        super();
    }
    //-----
    public SingletonException(String s)
    {
        super(s);
    }
}

class PrintSpooler
{
    //this is a prototype for a printer-spooler class
    //such that only one instance can ever exist
    static boolean
        instance_flag=false; //true if 1 instance

    public PrintSpooler() throws SingletonException
    {
        if (instance_flag)
            throw new SingletonException("Only one spooler allowed");
        else
            instance_flag = true;    //set flag for 1 instance
            System.out.println("spooler opened");
    }
    //-----
    public void finalize()
    {
        instance_flag = false;    //clear if destroyed
    }
}
```

# Exemple 1 (suite)

```
public class singleSpooler
{
    static public void main(String argv[])
    {
        PrintSpooler pr1, pr2;

        //open one spooler--this should always work
        System.out.println("Opening one spooler");
        try{
            pr1 = new PrintSpooler();
        }
        catch (SingletonException e)
        {System.out.println(e.getMessage());}

        //try to open another spooler --should fail
        System.out.println("Opening two spoolers");

        try{
            pr2 = new PrintSpooler();
        }
        catch (SingletonException e)
        {System.out.println(e.getMessage());}
    }
}
```

## Example 2

```
final class PrintSpooler
{
    //a static class implementation of Singleton pattern
    static public void print(String s)
    {
        System.out.println(s);
    }
}
//=====
public class staticPrint
{
    public static void main(String argv[])
    {
        Printer.print("here it is");
    }
}
```

## Exemple 3

```
class iSpooler
{
    //this is a prototype for a printer-spooler class
    //such that only one instance can ever exist
    static boolean instance_flag = false; //true if 1 instance

    //the constructor is privatized-
    //but need not have any content
    private iSpooler() { }
    //static Instance method returns one instance or null
    static public iSpooler Instance()
    {
        if (! instance_flag)
        {
            instance_flag = true;
            return new iSpooler(); //only callable from within
        }
        else
            return null; //return no further instances
    }
    //-----
    public void finalize()
    {
        instance_flag = false;
    }
}
```

## Exemple 4

```
// Only one object of this class can be created
class Singleton {
    private static Singleton instance = null;

    private Singleton() {
        ...
    }

    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
    ...
}

class Program {
    public void aMethod() {
        Singleton X = Singleton.getInstance();
    }
}
```



# Résumé

- Le **Factory Pattern** est utilisé pour choisir et retourner une instance d'une classe parmi un nombre de classes similaires selon une donnée fournie à la factory
- Le **Abstract Factory Pattern** est utilisé pour retourner un groupe de classes
- Le **Builder Pattern** assemble un nombre d'objets pour construire un nouvel objet, à partir des données qui lui sont présentées. Fréquemment le choix des objets à assembler est réalisé par le biais d'une Factory
- Le **Prototype Pattern** copie ou clone une classe existante plutôt que de créer une nouvelle instance lorsque cette opération est coûteuse
- Le **Singleton Pattern** est un pattern qui assure qu'il n'y a qu'une et une seule instance d'un objet et qu'il est possible d'avoir un accès global à cette instance

# **Patterns de structure**

# Plan 4

- Introduction
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

# Introduction

- Description de la manière dont doivent être connectés des objets de l'application afin de rendre ces connections indépendantes des évolutions futures de l'application
- Découplage de l'interface et de l'implémentation de classes et d'objets
- Abstraction de la manière dont les classes et les objets sont composés pour former des structures plus importantes.

# Présentation (1/2)

- **Adapter** : rendre un objet conformant à un autre
- **Bridge** : pour lier une abstraction à une implantation
- **Composite** : basé sur des objets primitifs et composants
- **Decorator** : ajoute des services à un objet
- **Facade** : cache une structure complexe
- **Flyweight** : petits objets destinés à être partagés
- **Proxy** : un objet en masque un autre

# Présentation (2/2)

Deux types de motifs :

- Motifs de structure de classes
  - Utilisation de l'héritage pour composer des interfaces et/ou des implémentations (ex : Adapter).
- Motifs de structure d'objets
  - composition d'objets pour réaliser de nouvelles fonctionnalités :
    - ajouter d'un niveau d'indirection pour accéder à un objet
    - ex : Adapter d'objet, Bridge, Facade, Proxy,
  - composition récursive pour organiser un nombre quelconque d'objets
    - ex : Composite

# Adapter

- **Definition :** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**Frequency of use:** medium high 

- **Objectifs :**
  - Convertir l'interface d'une **classe** dans une autre interface comprise par la partie cliente.
  - Permettre à des classes de fonctionner ensemble, ce qui n'aurait pas été possible sinon (à cause de leurs interfaces incompatibles).
- **Résultat:**
  - Le Design Pattern permet d'isoler l'adaptation d'un sous-système.

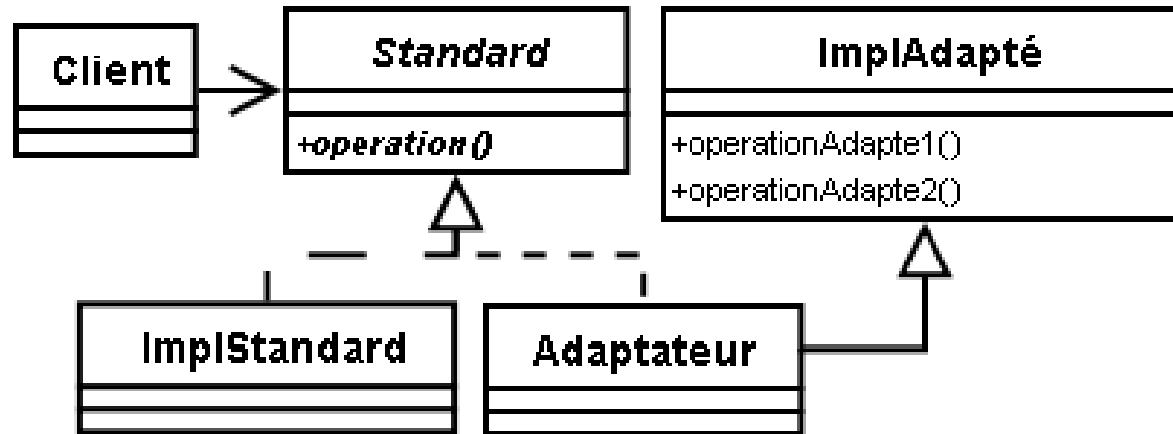
# Utilisation

- Le système doit intégrer un **sous-système** existant. Ce sous-système a une interface non standard par rapport au système.
- Cela peut être le cas d'un driver bas niveau pour de l'informatique embarquée. Le driver fournit par le fabricant ne correspond pas à l'interface utilisée par le système pour d'autres drivers.
- La solution est de masquer cette interface non standard au système et de lui présenter une interface standard. La partie cliente utilise les **méthodes** de l'Adaptateur qui utilise les méthodes du sous-système pour réaliser les opérations correspondantes.

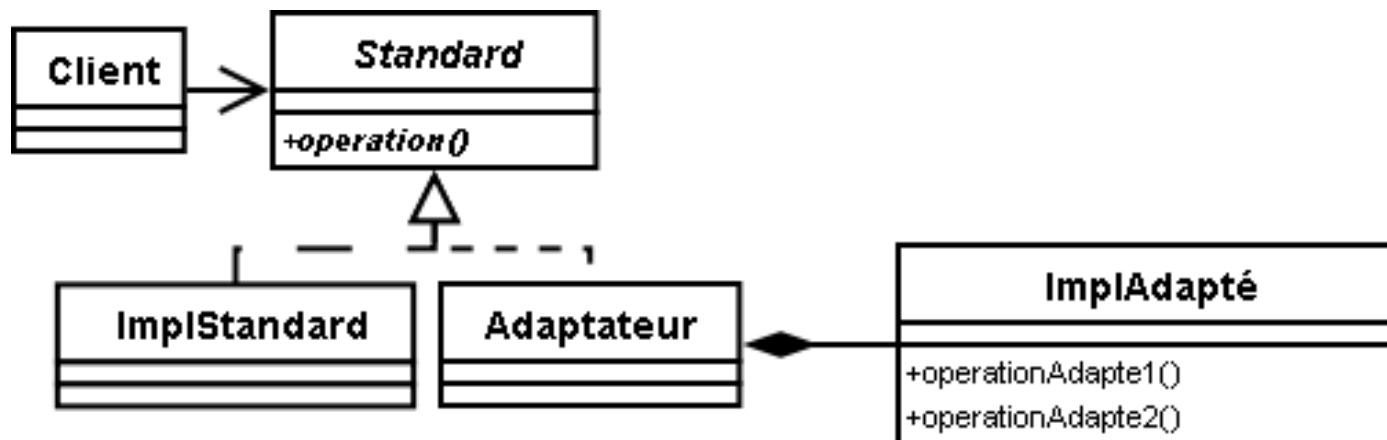


# Diagramme de classe

- Adaptateur par héritage



- Adaptateur par composition



# Caractéristiques

## Problème

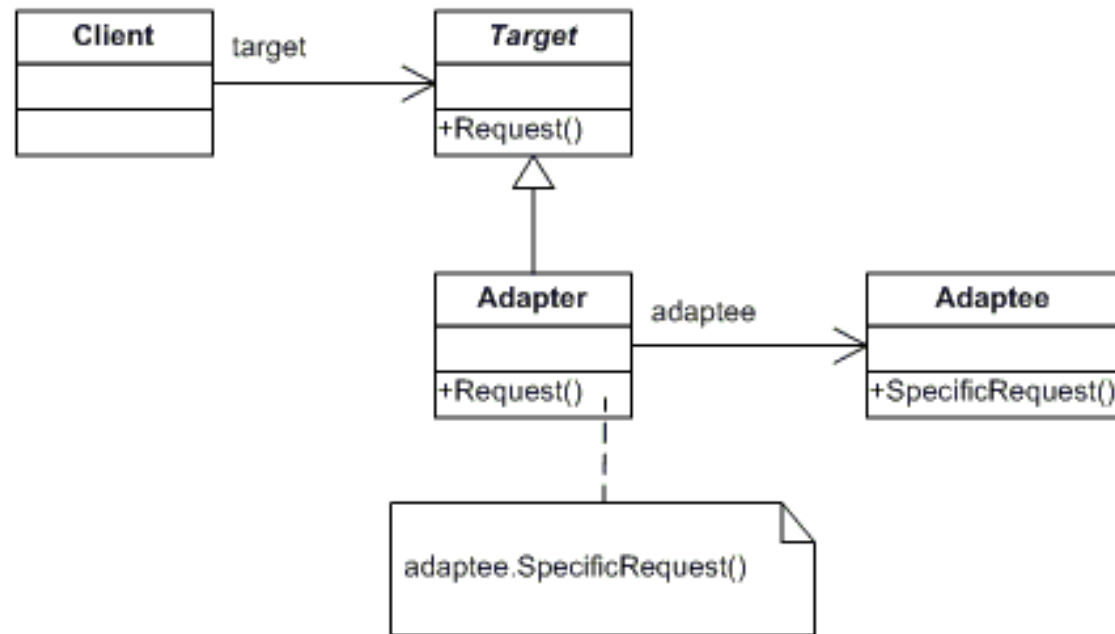
- Utilisation d'une classe existante dont l'interface ne nous convient pas (→ convertir l'interface d'une classe en une autre)
- Utilisation de plusieurs sous-classes dont l'adaptation des interfaces est impossible par dérivation (→ Object Adapter)

# Caractéristiques (2)

## Conséquences

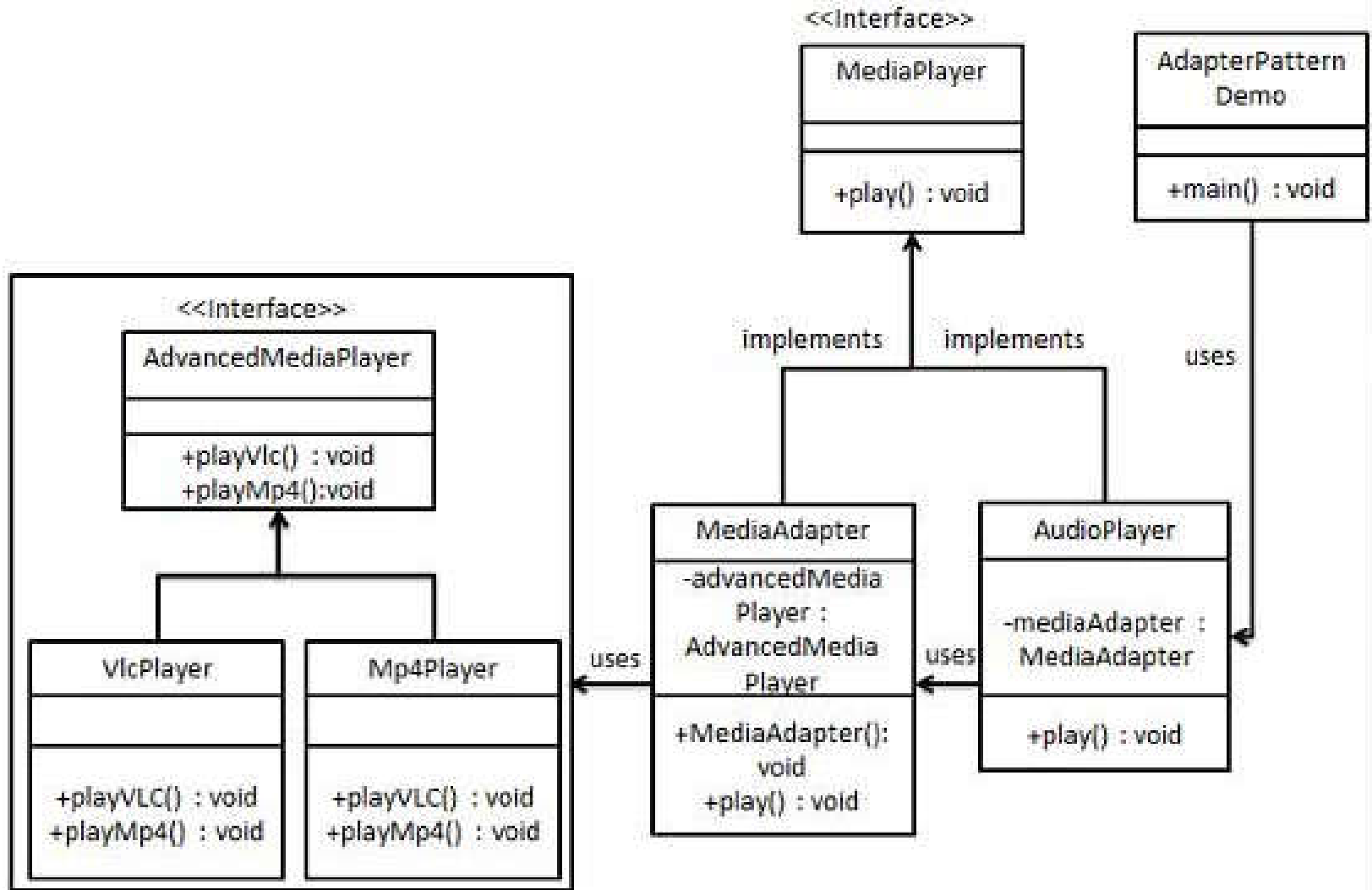
- Adapter de classe
  - + il n'introduit qu'une nouvelle classe, une indirection vers la classe adaptée n'est pas nécessaire
  - MAIS il ne fonctionnera pas dans le cas où la classe adaptée est racine d'une dérivation
- Adapter d'objet
  - + il peut fonctionner avec plusieurs classes adaptées
  - MAIS il peut difficilement redéfinir des comportements de la classe adaptée

# Exemple



- **Target (Compound)**
  - defines the domain-specific interface that Client uses.
- **Adapter (RichCompound)**
  - adapts the interface Adaptee to the Target interface.
- **Adaptee (ChemicalDatabank)**
  - defines an existing interface that needs adapting.
- **Client (AdapterApp)**
  - collaborates with objects conforming to the Target interface

# Exemple



# MediaPlayer & AdvancedMediaPlayer

*MediaPlayer.java*

```
public interface MediaPlayer {  
    public void play(String audioType, String fileName);  
}
```

*AdvancedMediaPlayer.java*

```
public interface AdvancedMediaPlayer {  
    public void playVlc(String fileName);  
    public void playMp4(String fileName);  
}
```

# VlcPlayer & Mp4Player-

*VlcPlayer.java*

```
public class VlcPlayer implements AdvancedMediaPlayer{
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file. Name: "+ fileName);
    }

    @Override
    public void playMp4(String fileName) {
        //do nothing
    }
}
```

*Mp4Player.java*

```
public class Mp4Player implements AdvancedMediaPlayer{

    @Override
    public void playVlc(String fileName) {
        //do nothing
    }

    @Override
    public void playMp4(string fileName) {
        System.out.println("Playing mp4 file. Name: "+ fileName);
    }
}
```

# MediaAdapter

*MediaAdapter.java*

```
public class MediaAdapter implements MediaPlayer {
    AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType){

        if(audioType.equalsIgnoreCase("vlc") ){
            advancedMusicPlayer = new VlcPlayer();

        }else if (audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer = new Mp4Player();
        }

    }

    @Override
    public void play(String audioType, String fileName) {

        if(audioType.equalsIgnoreCase("vlc")){
            advancedMusicPlayer.playVlc(fileName);
        }
        else if(audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer.playMp4(fileName);
        }

    }
}
```



# AudioPlayer

*AudioPlayer.java*

```
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileName) {

        //inbuilt support to play mp3 music files
        if(audioType.equalsIgnoreCase("mp3")){
            System.out.println("Playing mp3 file. Name: " + fileName);
        }

        //mediaAdapter is providing support to play other file formats
        else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        }

        else{
            System.out.println("Invalid media. " + audioType + " format not supported");
        }
    }
}
```

# Classe Client: AdapterDemo

*AdapterPatternDemo.java*

```
public class AdapterPatternDemo {  
    public static void main(String[] args) {  
        AudioPlayer audioPlayer = new AudioPlayer();  
  
        audioPlayer.play("mp3", "beyond the horizon.mp3");  
        audioPlayer.play("mp4", "alone.mp4");  
        audioPlayer.play("vlc", "far far away.vlc");  
        audioPlayer.play("avi", "mind me.avi");  
    }  
}
```

Verify the output.

```
Playing mp3 file. Name: beyond the horizon.mp3  
Playing mp4 file. Name: alone.mp4  
Playing vlc file. Name: far far away.vlc  
Invalid media. avi format not supported
```

# Bridge

- **Definition :** Decouple an abstraction from its implementation so that the two can vary independently.

Frequency of use: medium



- **Objectifs :**
  - Découpler l'abstraction d'un concept de son implémentation.
  - Permettre à l'abstraction et l'implémentation de varier indépendamment.
- **Résultat :**
  - Le Design Pattern permet d'isoler le lien entre une couche de haut niveau et celle de bas niveau.

# Utilisation

- Le système comporte une couche bas niveau réalisant l'implémentation et une couche haut niveau réalisant l'abstraction. Il est nécessaire que chaque couche soit indépendante.
- Cela peut être le cas du système d'édition de documents d'une application. Pour l'implémentation, il est possible que l'édition aboutisse à une sortie imprimante, une image sur disque, un document PDF, etc... Pour l'abstraction, il est possible qu'il s'agisse d'édition de factures, de rapports de stock, de courriers divers, etc...
- Chaque implémentation présente une interface pour les opérations de bas niveau standard (sortie imprimante), et chaque abstraction hérite d'une classe effectuant le lien avec cette interface (tracer une ligne). Ainsi les abstractions peuvent utiliser ce lien pour appeler la couche implémentation pour leurs besoins (imprimer facture).

# Caractéristiques

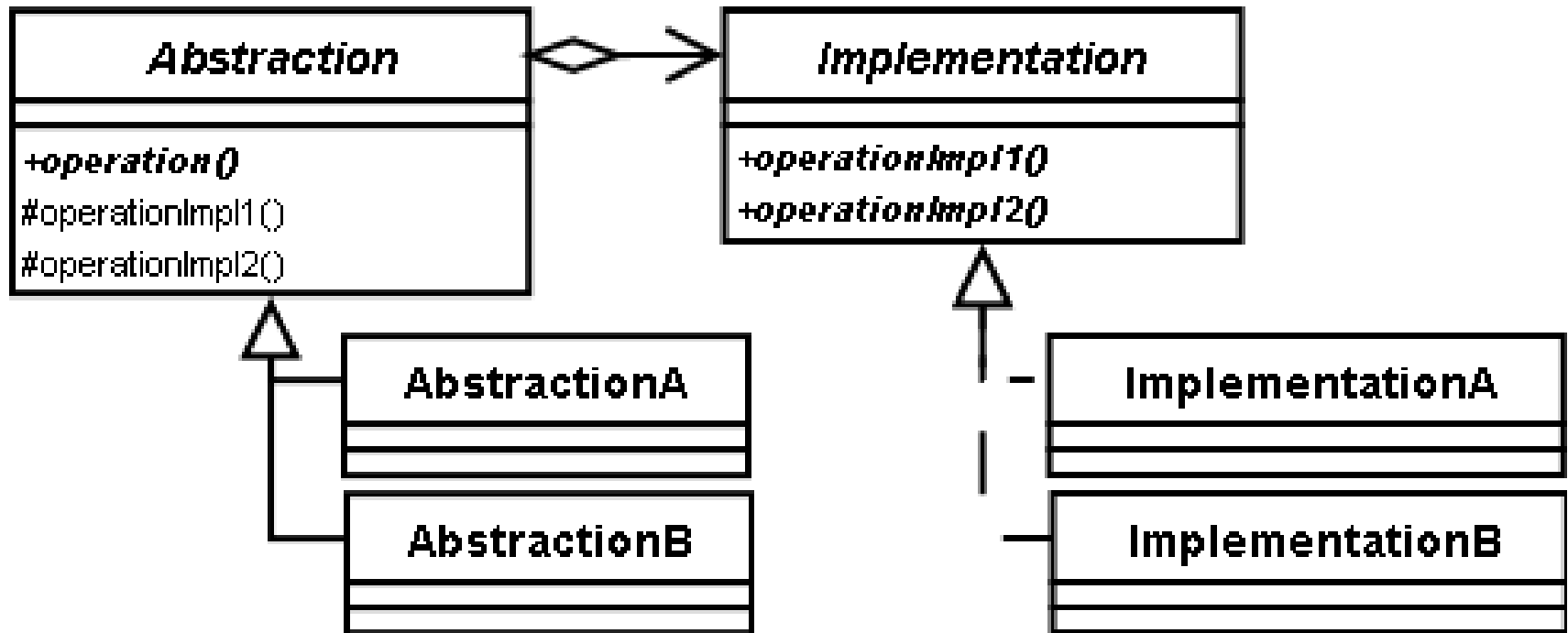
## Problème

- ce motif est à utiliser lorsque l'on veut découpler l'implémentation de l'abstraction de telle sorte que les deux puissent varier indépendamment

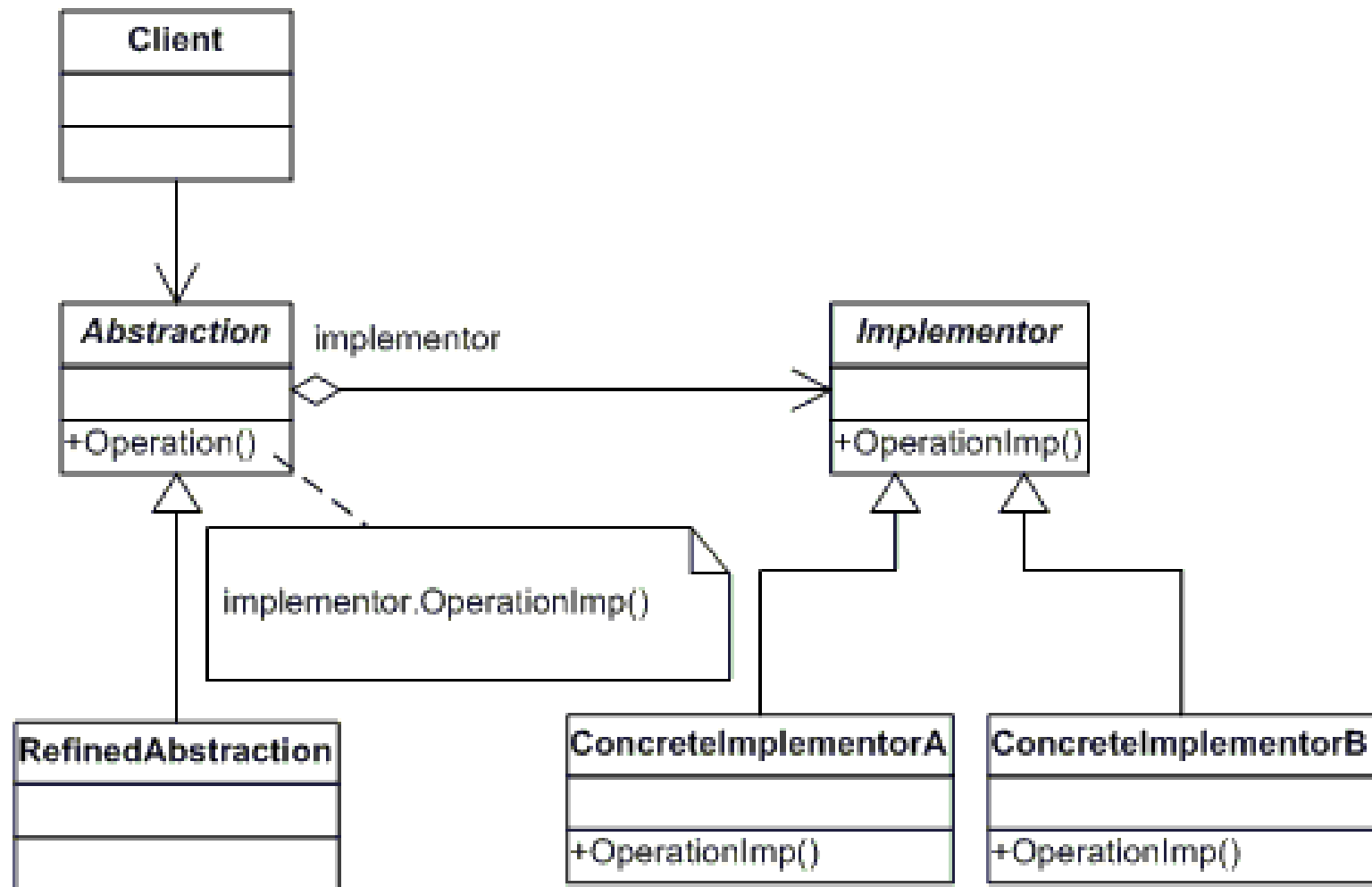
## Conséquences

- + interfaces et implémentations peuvent être couplées/découplées lors de l'exécution

# Diagramme de classe



## Diagramme de classe (2)

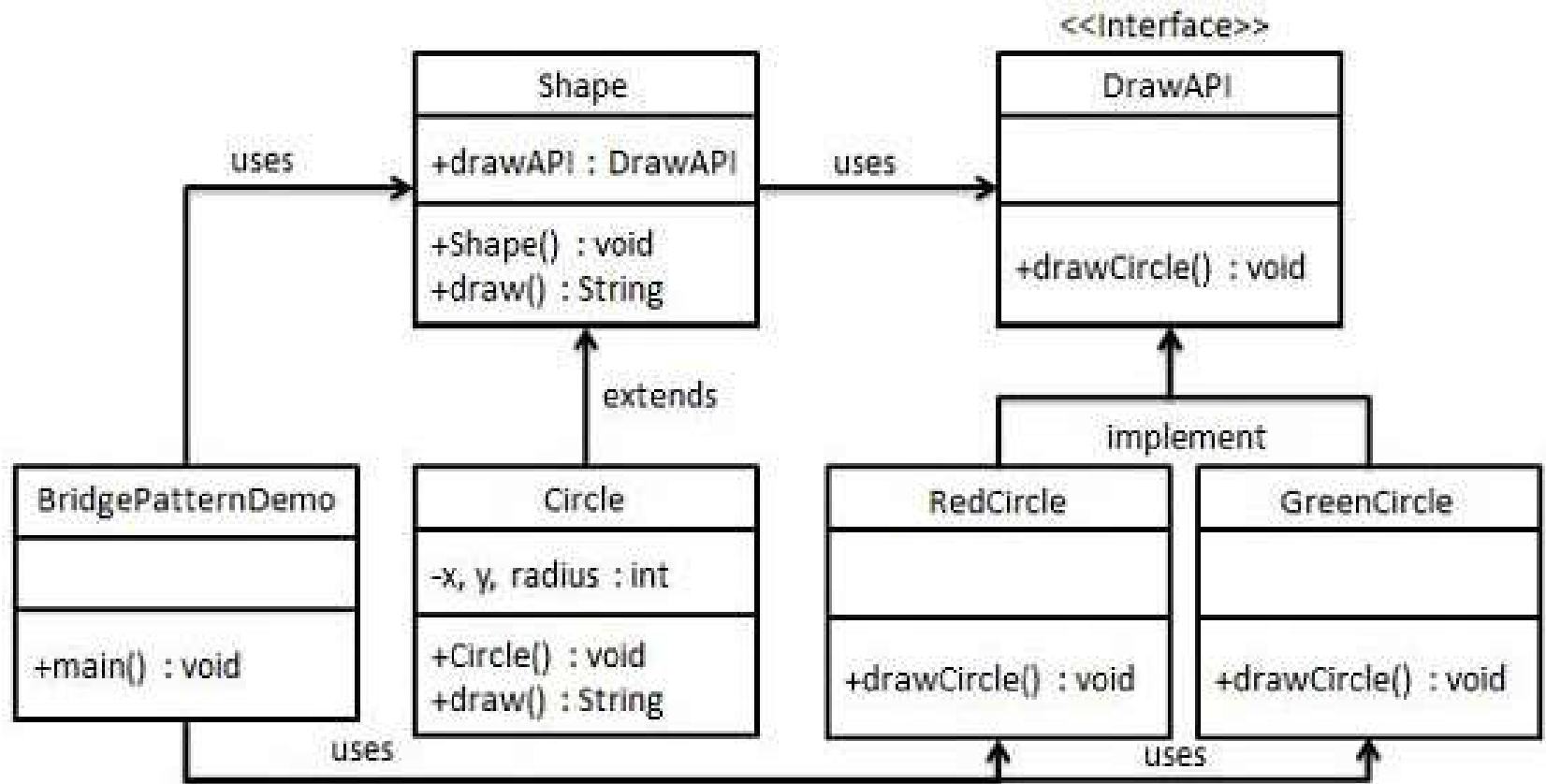


# Exemple

- **Abstraction (CustomersBase)**
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- **RefinedAbstraction (Customers)**
  - extends the interface defined by Abstraction.
- **Implementor (DataObject)**
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor (CustomersDataObject)**
  - implements the Implementor interface and defines its concrete implementation.



## Exemple 2



# Composite

- **Definition :** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**Frequency of use:** medium high



- **Objectifs :**
  - Organiser les objets en structure arborescente afin de représenter une hiérarchie.
  - Permettre à la partie cliente de manipuler un **objet unique** et un **objet composé** de la même manière
- **Résultat :**
  - Le Design Pattern permet d'isoler l'appartenance à un agrégat.

# Utilisation

- Le système comporte une hiérarchie avec un nombre de niveaux non déterminé. Il est nécessaire de pouvoir considérer un groupe d'éléments comme un élément unique.
- Cela peut être le cas des éléments graphiques d'un logiciel de DAO. Plusieurs éléments graphiques peuvent être regroupés en un nouvel élément graphique.
- Chaque élément est un composant potentiel. En plus des éléments classiques, il y a un élément composite qui peut être composé de plusieurs composants. Comme l'élément composite est un composant potentiel, il peut être composé d'autres éléments composites.

# Caractéristiques

## Problème

- établir des structures arborescentes entre des objets et les traiter uniformément

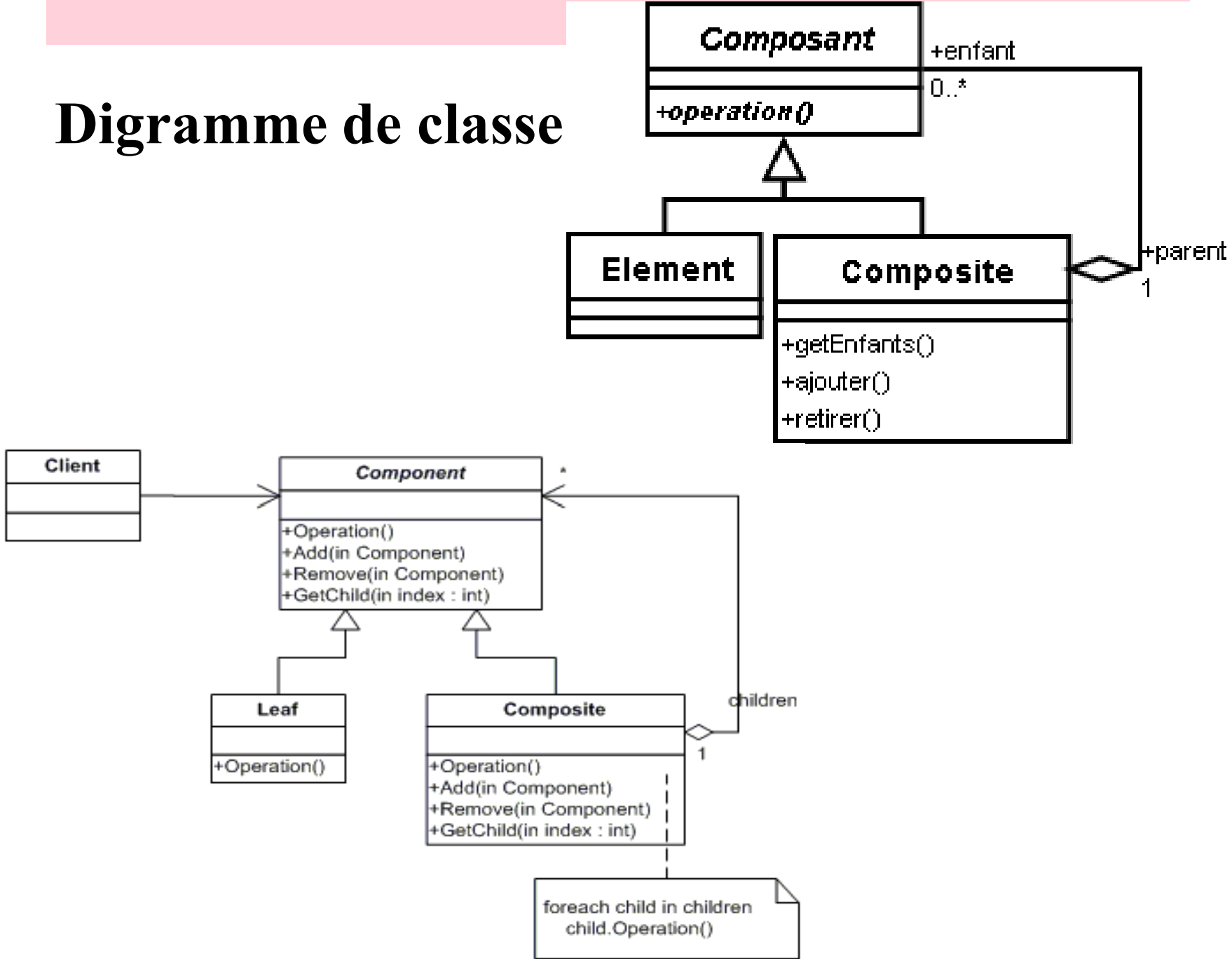
## Conséquences

- + hiérarchies de classes dans lesquelles l'ajout de nouveaux composants est simple
- + simplification du client qui n'a pas à se préoccuper de l'objet accédé
- MAIS il est difficile de restreindre et de vérifier le type des composants

## Exemple

- java.awt.Component
- java.awt.Container

# Digramme de classe



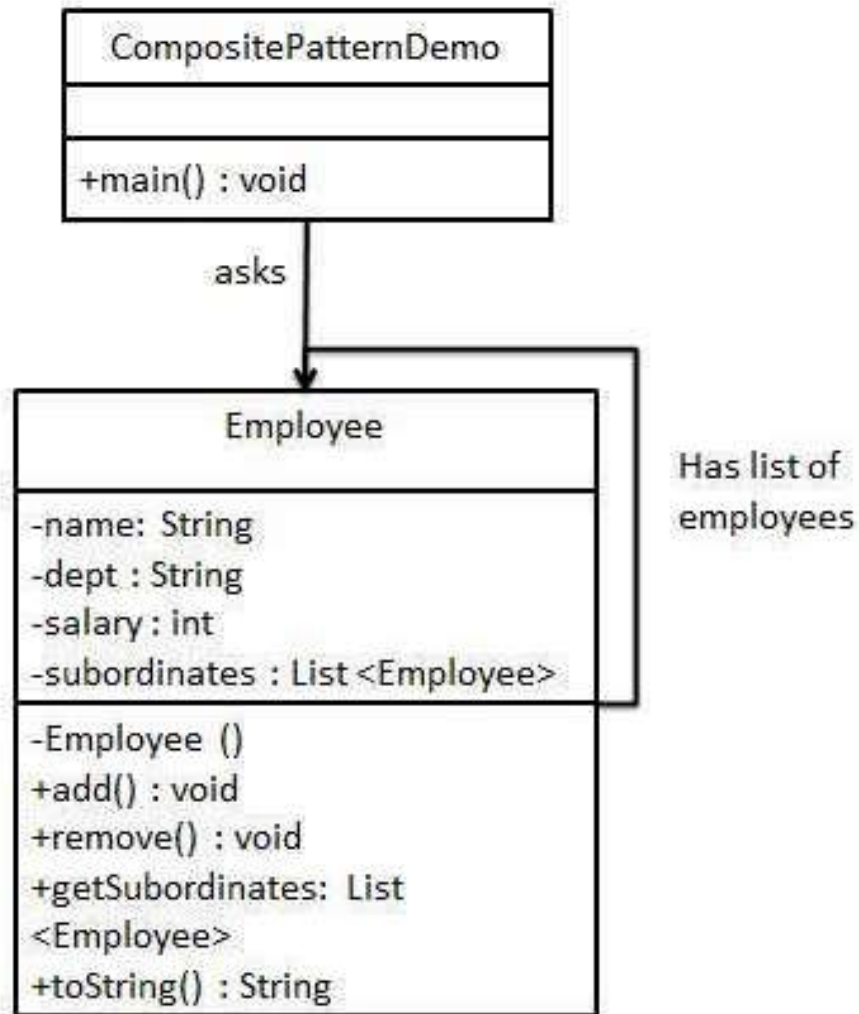
# Exemple

- **Component (DrawingElement)**
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Leaf (PrimitiveElement)**
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.

# Exemple (suite)

- **Composite (CompositeElement)**
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- **Client (CompositeApp)**
  - manipulates objects in the composition through the Component interface.

# Exemple





# Decorator

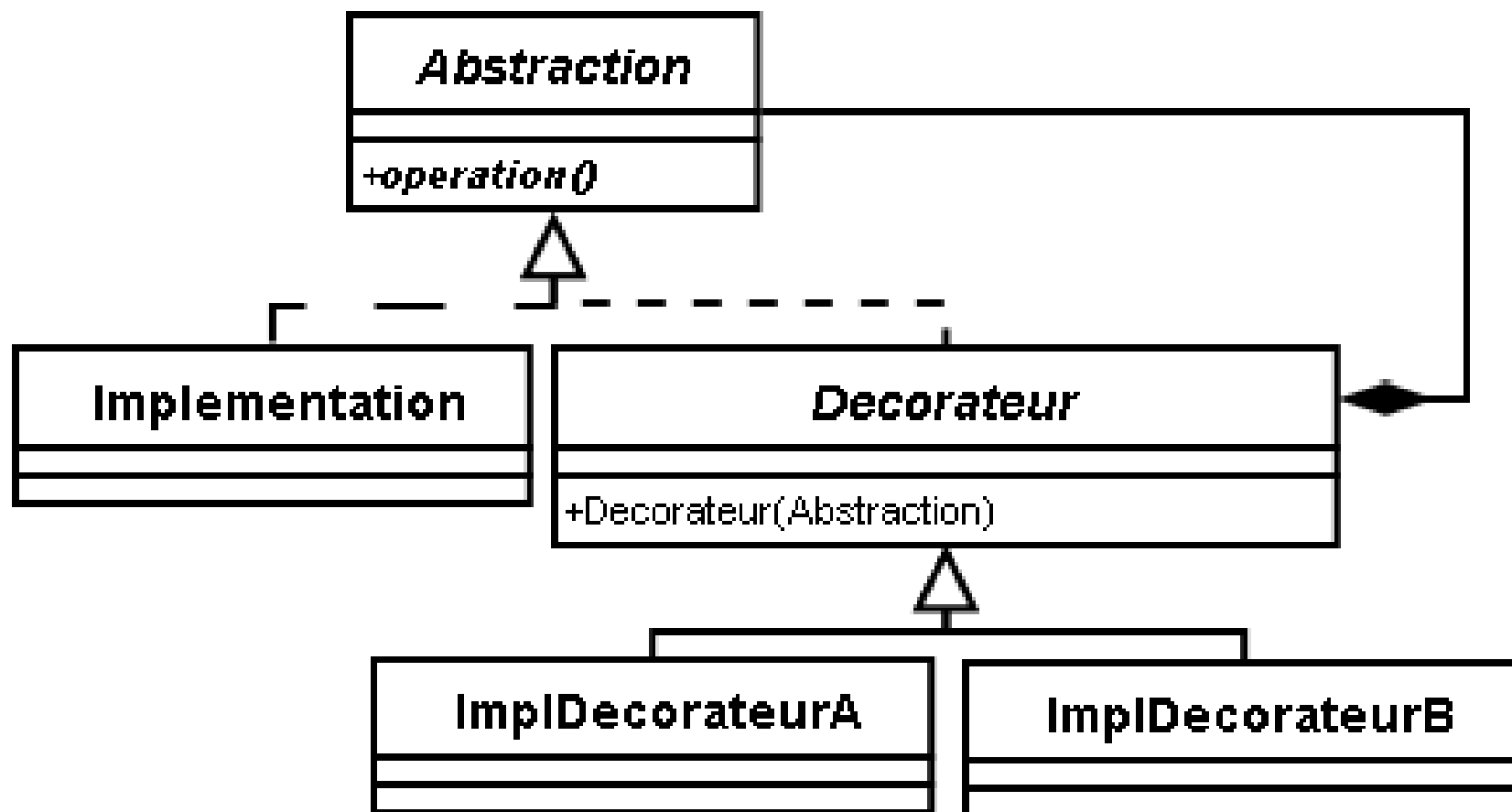
- **Definition :** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Frequency of use:** medium

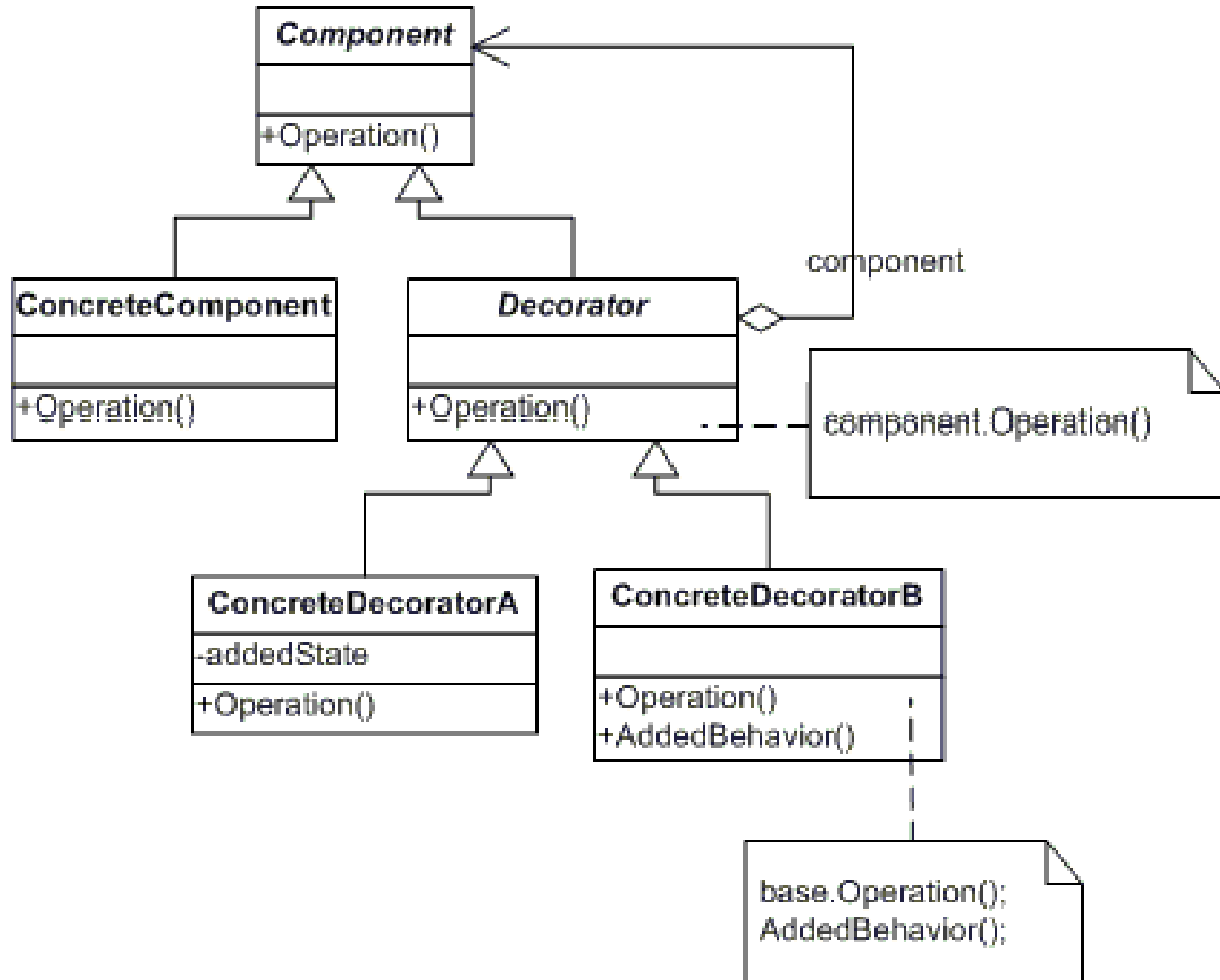


- **Objectifs :**
  - Ajouter dynamiquement des **responsabilités** (non obligatoires) à un objet.
  - Eviter de sous-classer la classe pour rajouter ces responsabilités.
- **Résultat :**
  - Le Design Pattern permet d'isoler les responsabilités d'un objet.

# Diagramme de classe



## Diagramme de classe (2)



# Utilisation

- Il est nécessaire de pouvoir étendre les responsabilités d'une **classe sans** avoir recours au sous-classage.
- Cela peut être le cas d'une classe gérant des d'E/S à laquelle on souhaite ajouter un buffer et des traces de log.
- La classe de départ est l'implémentation. Les fonctionnalités supplémentaires (buffer, log) sont implémentées par des classes supplémentaires : les décorateurs.
- Les décorateurs ont la même **interface** que la classe de départ. Dans leur **implémentation** des méthodes, elles implémentent les fonctionnalités supplémentaires et font appel à la méthode correspondante d'une **instance** avec la même interface. Ainsi, il est possible d'enchaîner plusieurs responsabilités supplémentaires, puis d'aboutir à l'implémentation finale.

# Caractéristiques

## Problème

- on veut ajouter/supprimer des responsabilités aux objets en cours de fonctionnement
- l'héritage est impossible à cause du nombre de combinaisons, ou à cause de droits d'accès

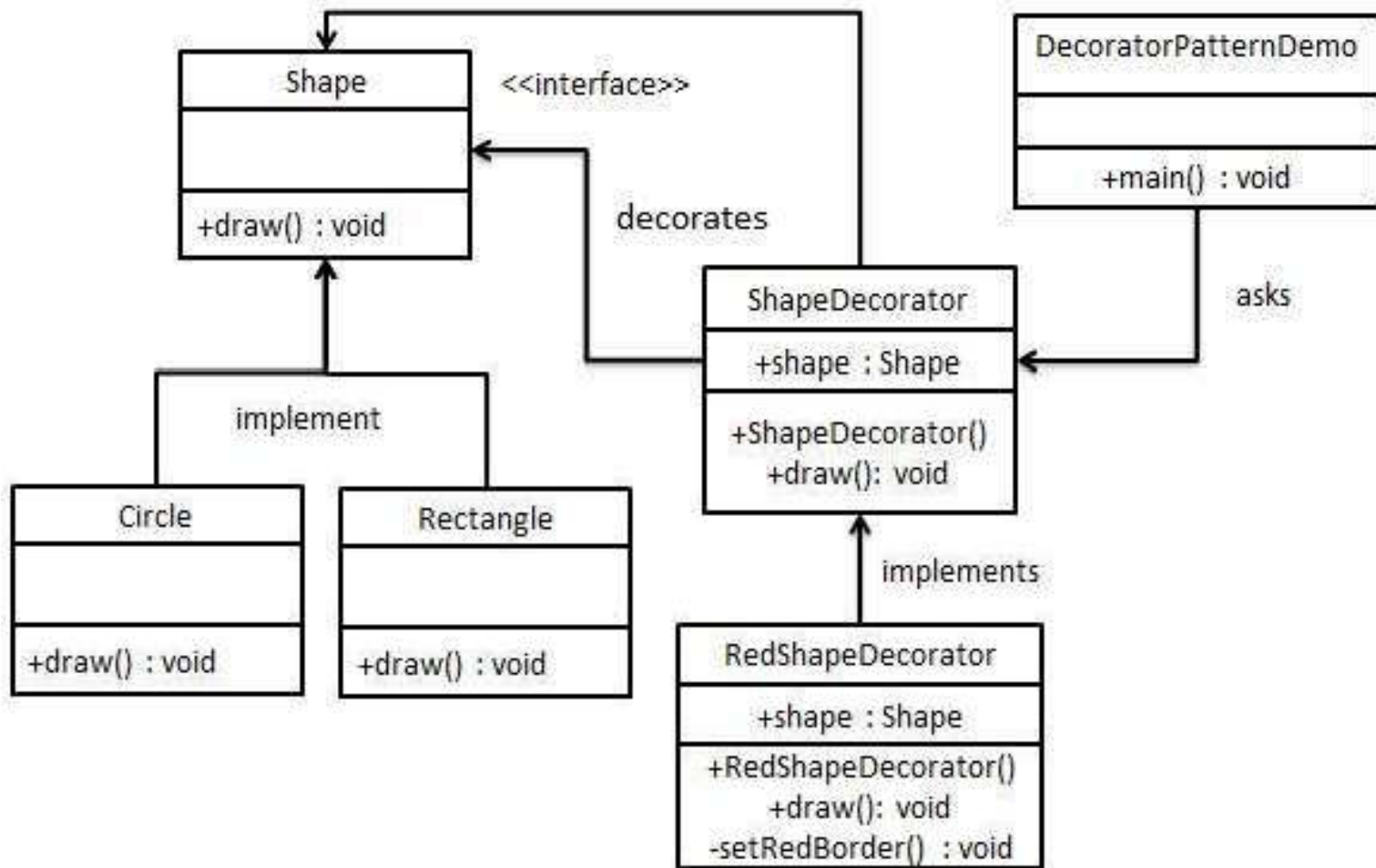
## Conséquences

- + plus de flexibilité que l'héritage
- + réduction de la taille des classes du haut de la hiérarchie
- MAIS beaucoup de petits objets

# Exemple

- **Component (LibraryItem)**
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent (Book, Video)**
  - defines an object to which additional responsibilities can be attached.
- **Decorator (Decorator)**
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator (Borrowable)**
  - adds responsibilities to the component.

## Exemple 2



# Facade

**Definition :** Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

**Frequency of use:** high



**Objectifs:**

- Fournir une interface unique en remplacement d'un ensemble d'interfaces d'un **sous-système**.
- Définir une interface de haut niveau pour rendre le sous-système plus simple d'utilisation.

**Résultat:**

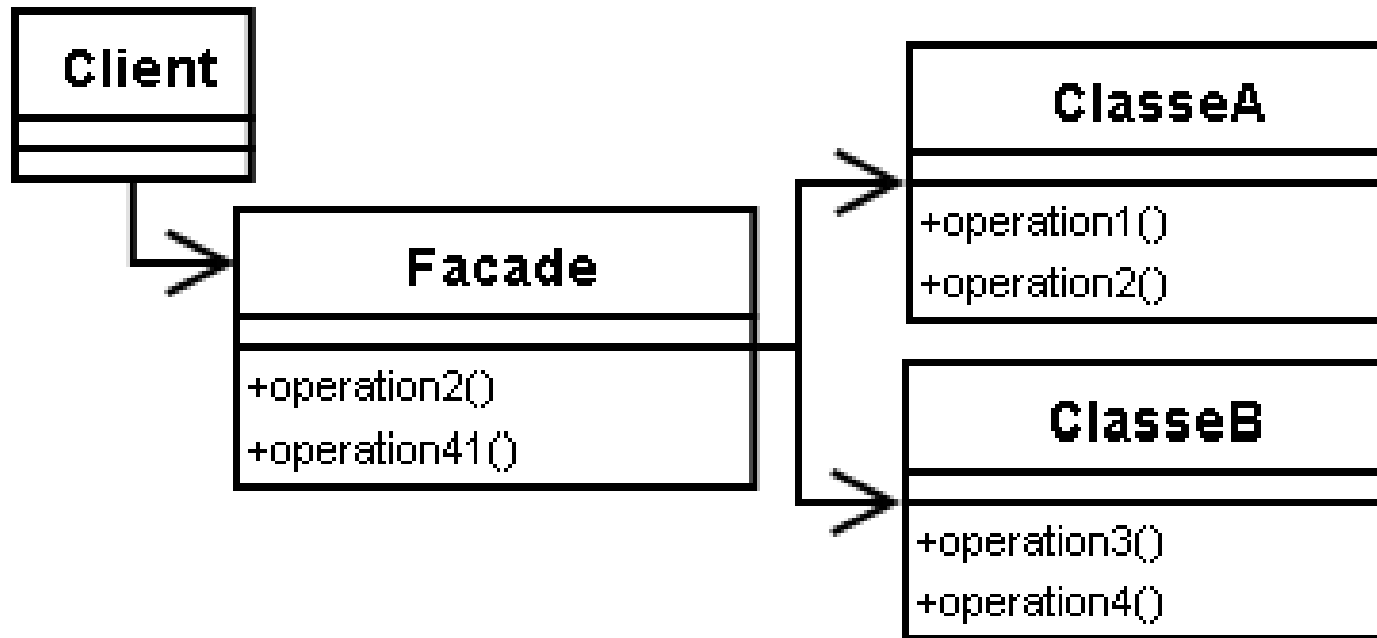
- Le Design Pattern permet d'isoler les fonctionnalités d'un sous-système utiles à la partie cliente.



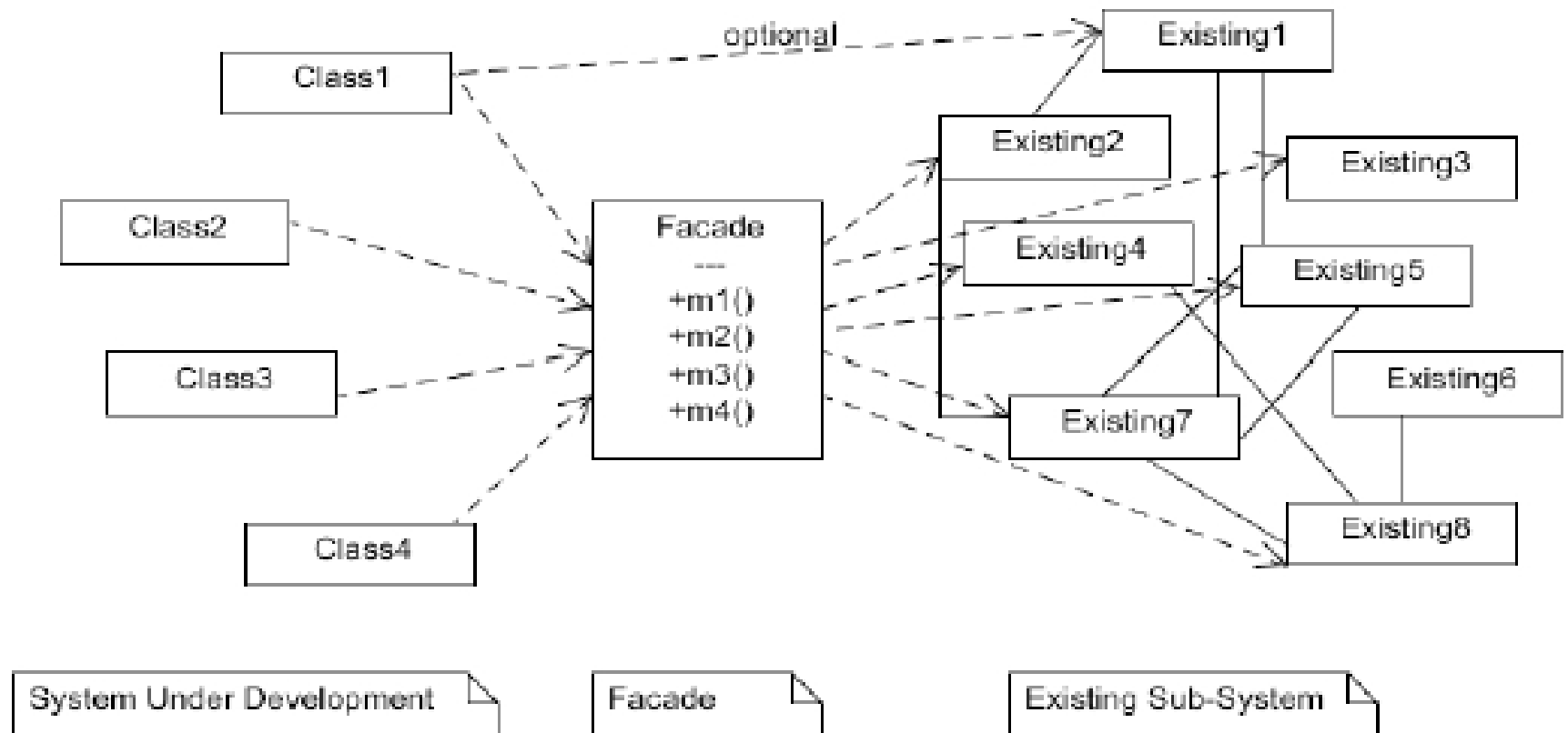
# Utilisation

- Le système comporte un sous-système complexe avec plusieurs interfaces. Certaines de ces interfaces présentent des opérations qui ne sont pas utiles au reste du système.
- Cela peut être le cas d'un sous-système communiquant avec des outils de mesure ou d'un sous-système d'accès à la base de données.
- Il serait plus judicieux de passer par une seule interface présentant seulement les opérations utiles. Une classe unique, la façade, présente ces opérations réellement nécessaires.
- *Remarque* : La façade peut également implémenter le Design Pattern **Singleton**.

## Diagramme de classe (1/2)



## Diagramme de classe (2/2)



# Caractéristiques

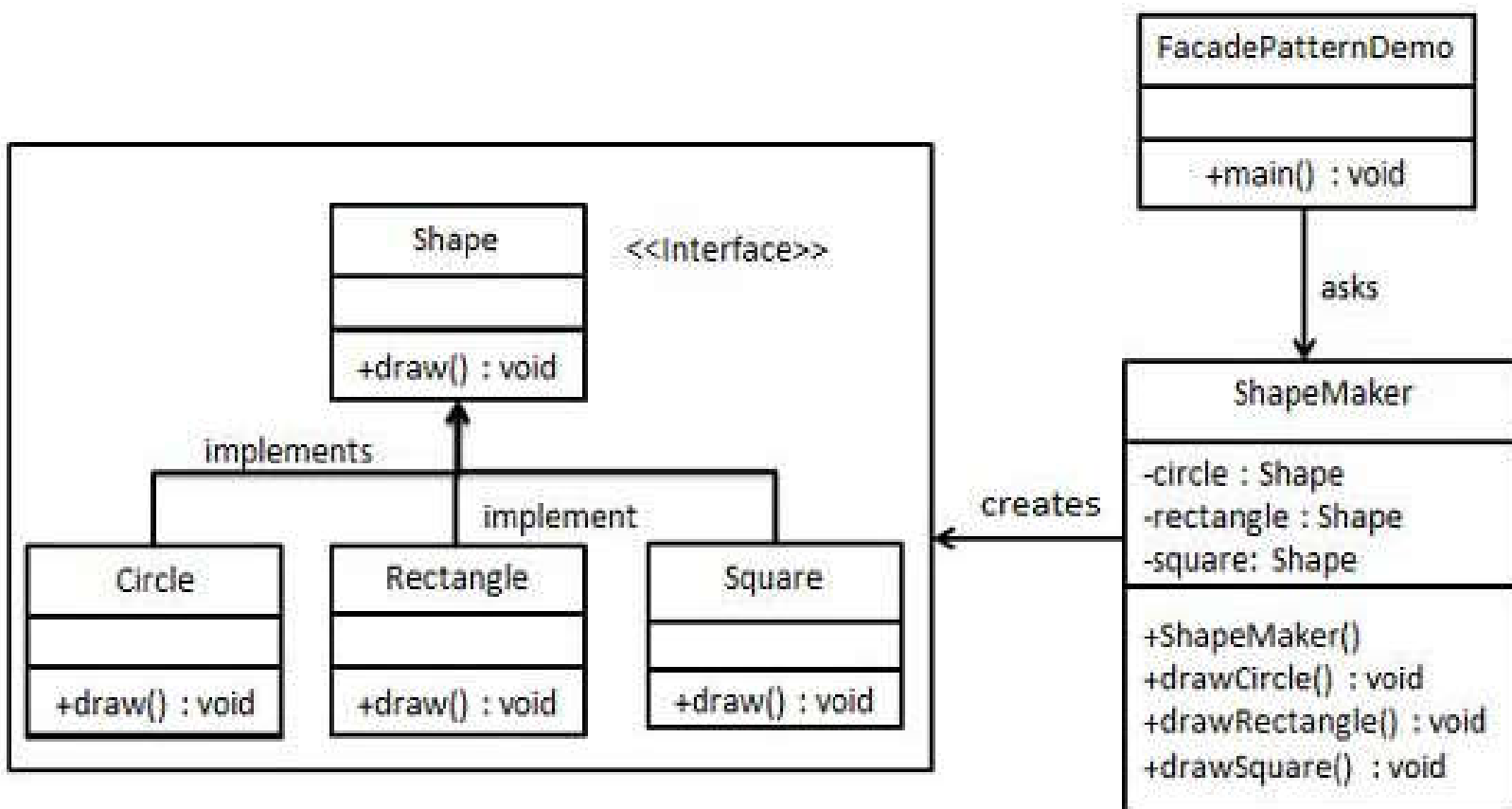
## Problème

- ce motif est à utiliser pour faciliter l'accès à un grand nombre de modules en fournissant une couche interface

## Conséquences

- + facilite l'utilisation de sous-systèmes
- + favorise un couplage faible entre des classes et l'application
- MAIS des fonctionnalités des classes interfacées peuvent être perdues selon la manière dont est réalisée la Façade

# Example 1



## Example 2

- `class CPU {  
    public void jump(long position) { ... }  
    public void execute() { ... }  
}`
- `class Memory {  
    public void load(long position, byte[] data) { ... }  
}`
- `class HardDrive {  
    public byte[] read(long lba, int size) { ... }  
}`

## Exemple 2 (suite)

```
/* Facade */
```

```
class Computer {  
    Memory memory;  HardDrive hardDrive; CPU cpu;  
  
    public void startComputer() {  
        memory.load(BOOT_ADDRESS,  
hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));  
        cpu.jump(BOOT_ADDRESS);  
        cpu.execute();  
    }  
}
```

## Exemple 2 (suite)

```
/* Client */
```

```
class You {  
    public static void main(String[] args) {  
        Computer facade = new Computer();  
        facade.startComputer();  
    }  
}
```



# Exemple 3

- **Facade (MortgageApplication)**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **Subsystem classes (Bank, Credit, Loan)**
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade and keep no reference to it.

# Flyweight

**Definition :** Use sharing to support large numbers of fine-grained objects efficiently.

**Frequency of use:** low



**Objectifs :** Utiliser le partage pour gérer efficacement un grand nombre d' **objets** de faible granularité.

**Résultat :**

Le Design Pattern permet d'isoler des objets partageables

# Utilisation

- Un système utilise un grand nombre d' **instances**. Cette quantité occupe une place très importante en mémoire.
- Chacune de ces instances a des **attributs** extrinsèques (propre au contexte) et intrinsèques (propre à l'objet).
- Cela peut être les caractéristiques des traits dans un logiciel de DAO, chaque trait possède :
  - une épaisseur (simple ou double), une continuité (continu, en pointillé), une ombre ou pas qui sont des **attributs intrinsèques**
  - des coordonnées qui sont des **attributs extrinsèques**.

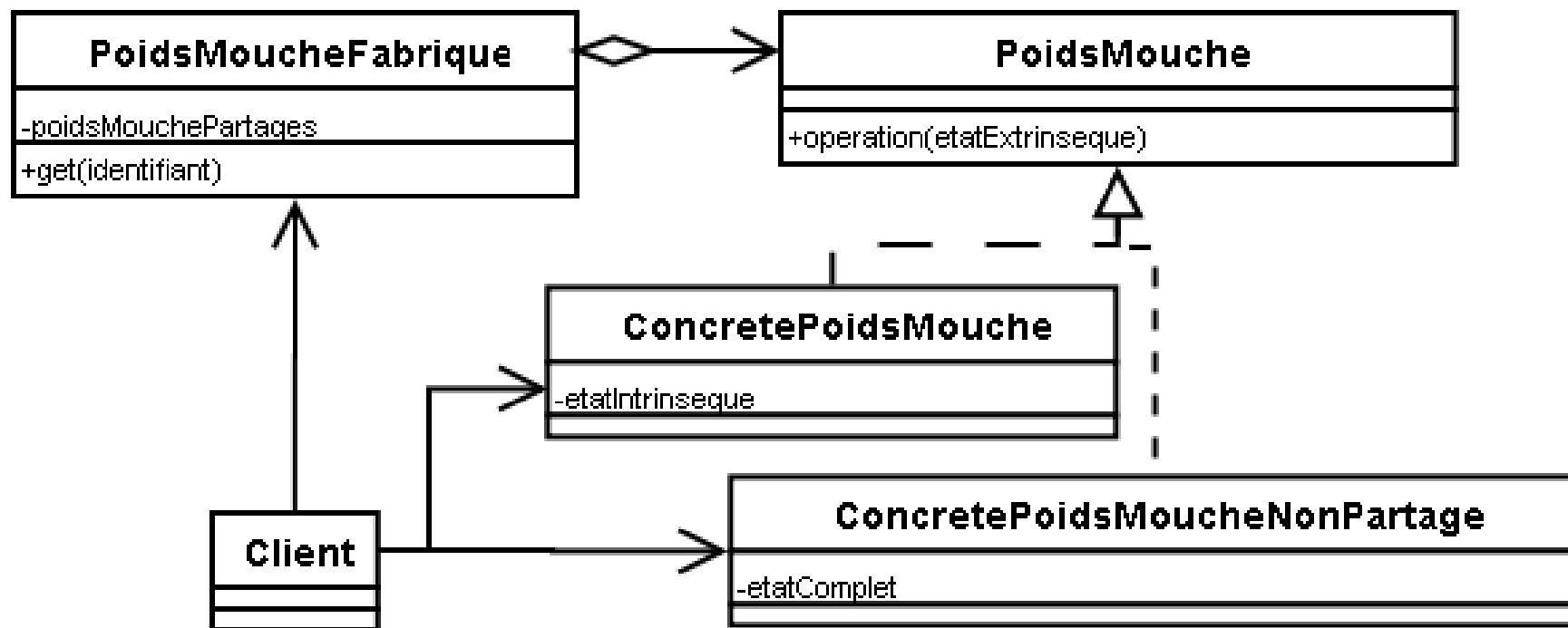
## Utilisation (suite)

- Plusieurs traits possèdent des épaisseurs, continuité et ombre similaires. Ces similitudes correspondent à des styles de trait.
- En externalisant les **attributs intrinsèques** des objets (style de trait), on peut avoir en mémoire une seule instance correspondant à un groupe de valeurs (simple-continu-sans ombre, double-pointillé-ombre).
- Chaque objet avec des **attributs extrinsèques** (trait avec les coordonnées) possède une référence vers une instance d'attributs intrinsèques (style de trait).

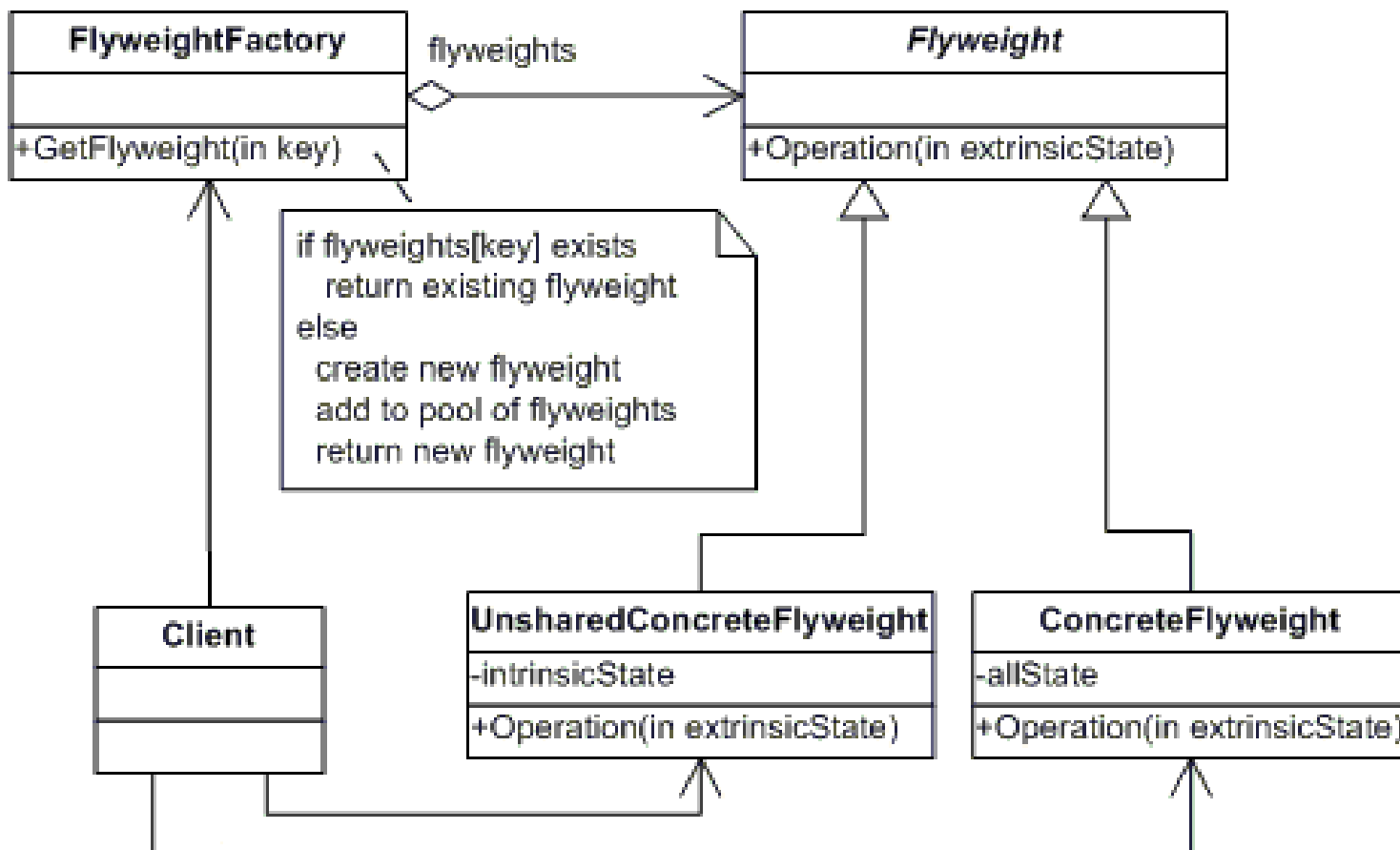
## Utilisation (suite)

- On obtient deux types de poids-mouche : les **poids-mouche partagés** (style de trait) et les **poids-mouche non partagés** (le trait avec ses coordonnées).
- La partie cliente demande le poids-mouche qui l'intéresse à la fabrique de poids-mouche.
  - S'il s'agit d'un **poids-mouche non partagé**, la fabrique le créera et le retournera.
  - S'il s'agit d'un **poids-mouche partagé**, la fabrique vérifiera si une instance existe. Si une instance existe, la fabrique la retourne, sinon la fabrique la crée et la retourne.

# Diagramme de classe



## Diagramme de classe (2)



# Caractéristiques

## Problème

- grand nombre d'objet
- le coût du stockage est élevé
- l'application ne dépend pas de l'identité des objets

## Conséquences

- + réduction du nombre d'instances
- coût d'exécution élevé
- plus d'états par objet



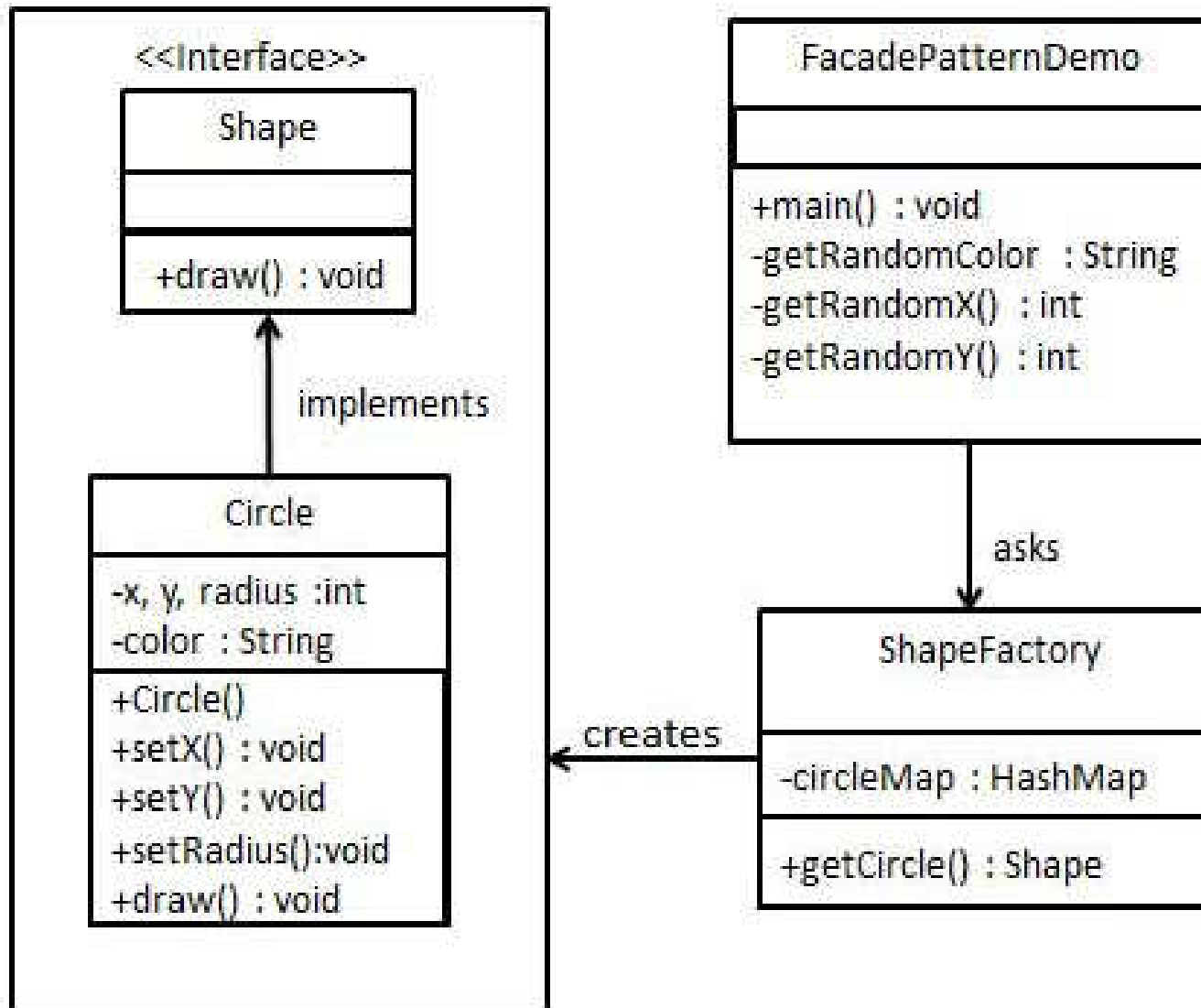
# Example 1

- **Flyweight (Character)**
  - declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight (CharacterA, CharacterB, ..., CharacterZ)**
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context.


# Exemple 1 (suite)

- **UnsharedConcreteFlyweight ( not used )**
  - not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing, but it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- **FlyweightFactory (CharacterFactory)**
  - creates and manages flyweight objects
  - ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects supplies an existing instance or creates one, if none exists.
- **Client (FlyweightApp)**
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).

## Exemple 2



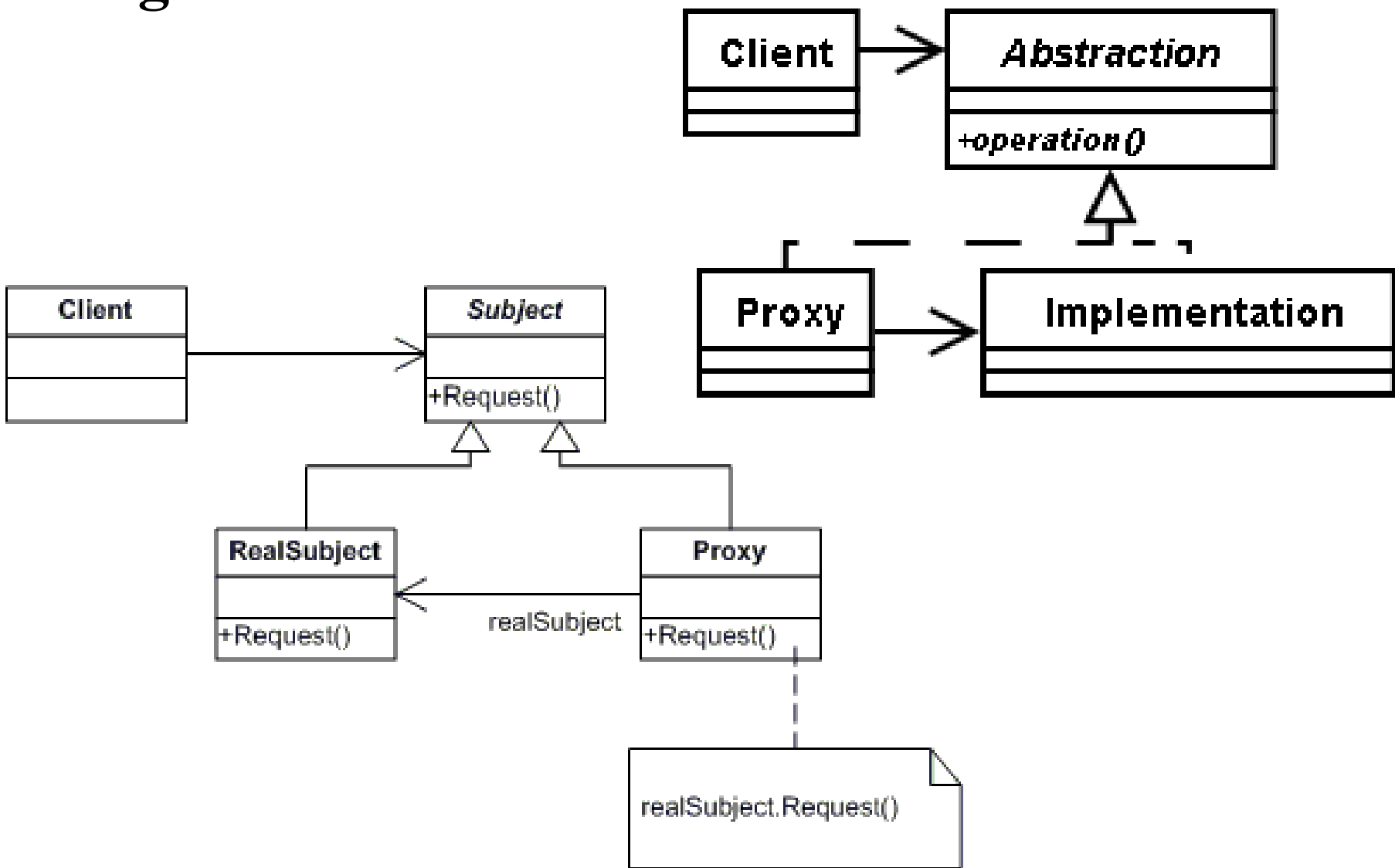
# Proxy

- **Definition :** Provide a surrogate or placeholder for another object to control access to it.
- **Frequency of use:** medium high 
- **Objectifs:** Fournir un intermédiaire entre la partie cliente et un **objet** pour contrôler les accès à ce dernier.
- **Résultat :** Le Design Pattern permet d'isoler le comportement lors de l'accès à un objet.

# Utilisation

- Les opérations d'un objet sont coûteuses en temps ou sont soumises à une gestion de droits d'accès. Il est nécessaire de contrôler l'accès à un objet.
- Cela peut être un système de chargement d'un document. Le document est très lourd à charger en mémoire ou il faut certaines habilitations pour accéder à ce document.
- L'objet réel (système de chargement classique) est l'implémentation. L'intermédiaire entre l'implémentation et la partie cliente est le proxy. Le proxy fournit la même interface que l'implémentation. Mais il ne charge le document qu'en cas de réel besoin (pour l'affichage par exemple) ou n'autorise l'accès que si les conditions sont satisfaites.

# Diagramme de classe



# Caractéristiques

## Problème

- ce motif est à utiliser pour agir par procuration pour un objet afin de contrôler les opérations qui lui sont appliquées
  - Masquer des problèmes d'accès (remote proxy )
  - Différer l'exécution d'opérations coûteuses (virtual proxy)
  - Contrôler les droits d'accès (protection proxy)

## Conséquences

- + ajout d'un niveau d'indirection lors de l'accès d'un objet permettant de cacher le fait que l'objet est dans un autre espace d'adressage, n'est pas créé, ...

## Exemple 1 : catalogue de véhicules

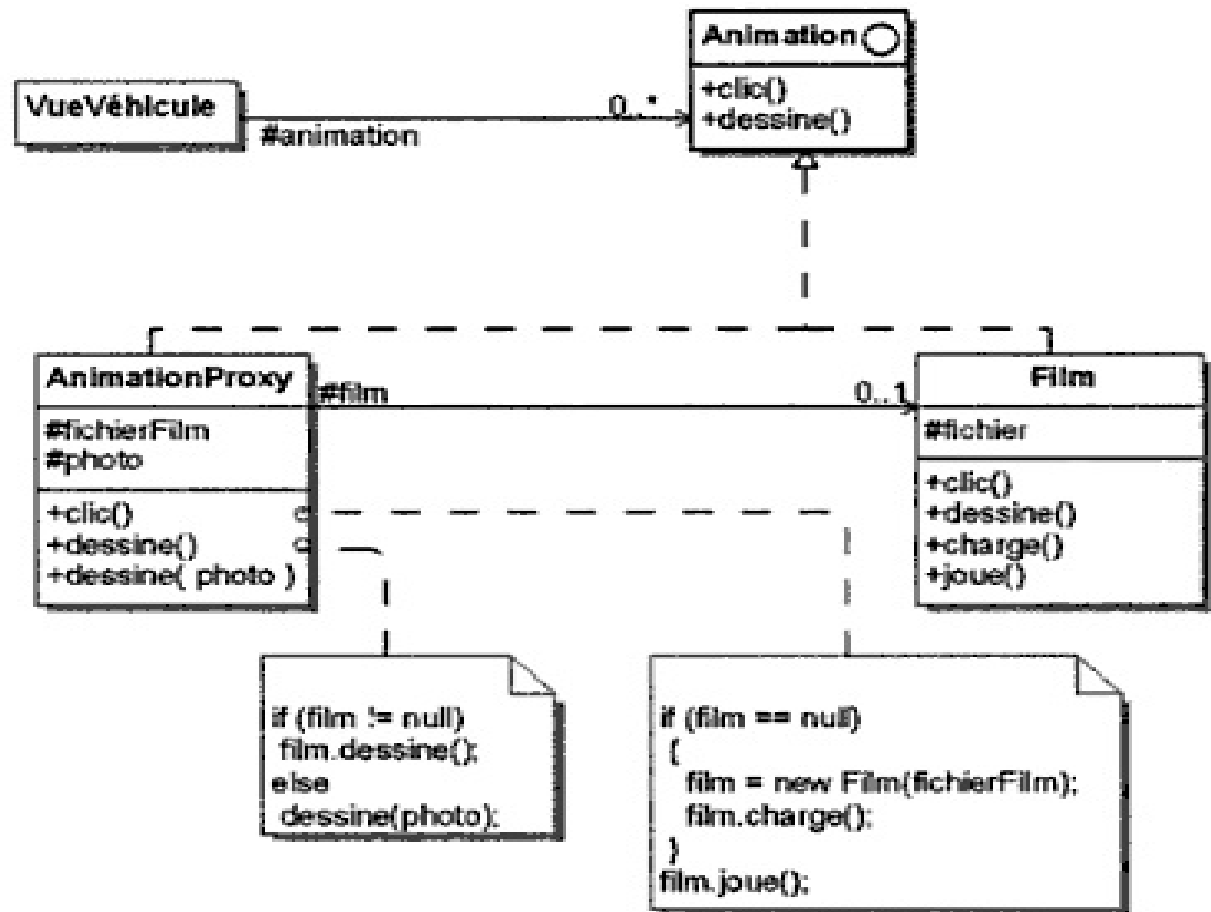
- Nous voulons offrir pour chaque véhicule du catalogue la possibilité de visualiser un film qui présente ce véhicule.
- Un clic sur la photo de la présentation du véhicule permet de jouer ce film.
- Une page du catalogue contient de nombreux véhicule et il est très lourd de créer en mémoire tous les objets d'animation car les films nécessitent une grande quantité mémoire et leur transfert au travers d'un réseau prend beaucoup de temps.

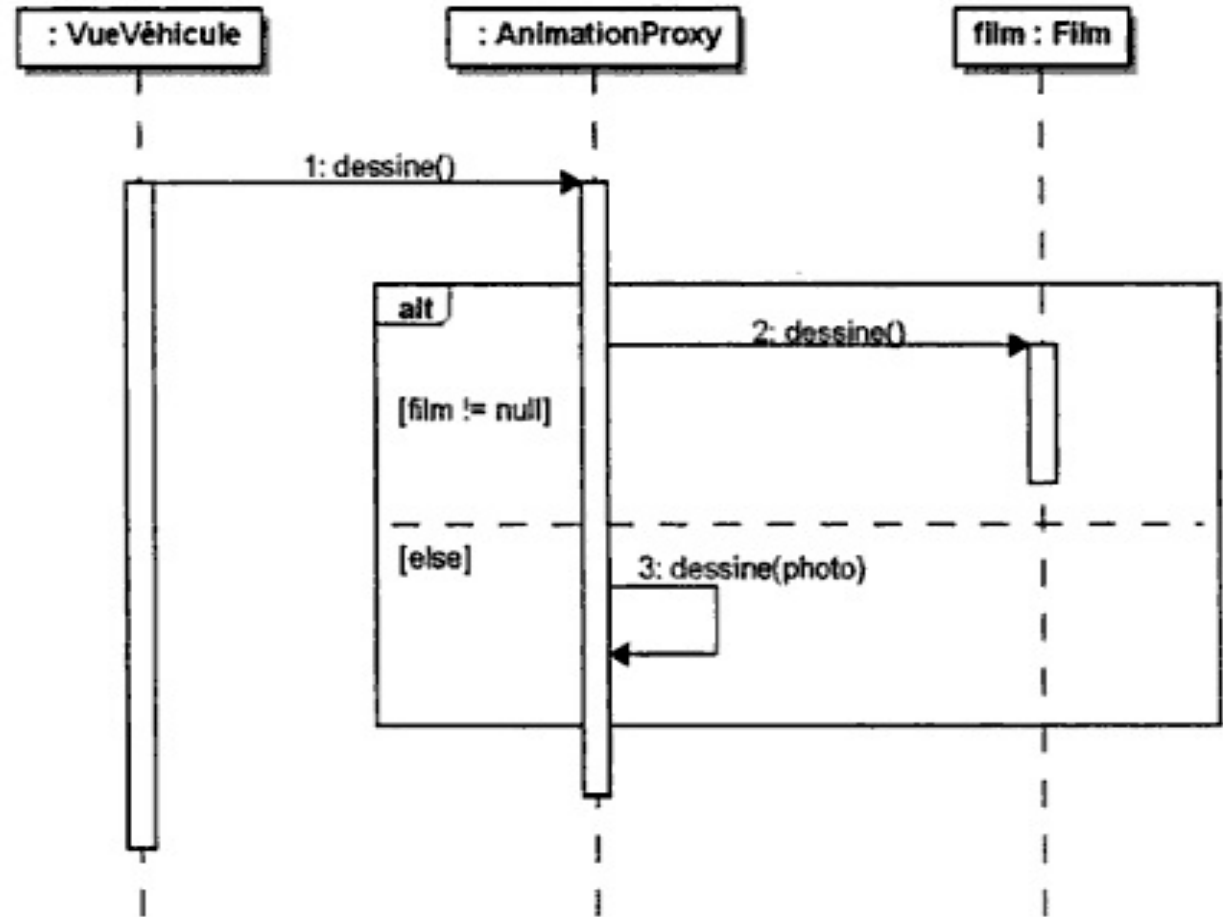


# Diagramme de classe

- L'objet photo est appelé le proxy du film. il se substitue au film pour l'affichage. il procède à la création du sujet uniquement lors du clic. il possède la même interface que l'objet film.

- Le pattern Proxy appliqué à l'affichage d'animation





- quand la proxy reçoit le message DESSINE , il affiche le film si celui-ci a déjà été créé et chargé . Quand le proxy reçoit le message CLIC , il joue le film après avoir préalablement créé et chargé le film .

## Exemple 2

- **Proxy (MathProxy)**

- maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
- provides an interface identical to Subject's so that a proxy can be substituted for for the real subject.
- controls access to the real subject and may be responsible for creating and deleting it.
- other responsibilities depend on the kind of proxy:
  - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
  - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images's extent.
  - *protection proxies* check that the caller has the access permissions required to perform a request.

## Exemple 2 (suite)

- **Subject (IMath)**
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject (Math)**
  - defines the real object that the proxy represents.

# Domaines d'applications

- Les proxy sont très utilisés en programmation par objets.il existe différents types de proxy
- **Proxy virtuel** : permet de créer un objet de taille importante
- **Proxy remote** : permet d'accéder à un objet s'exécutant dans un autre environnement. Ce type de proxy est mis en œuvre dans les systèmes d'objets distants (CORBA , Java RMI);
- **proxy de protection** : permet de sécuriser l'accès à un objet, par exemple par des techniques d'authentification.

# **Partie 3**

## **Patterns de comportement**

# Plan

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

# Patterns de comportement

- Description de structures d'objets ou de classes avec leurs interactions
- Deux types de motifs
  - Motifs de comportement de classes :
    - utilisation de l'héritage pour répartir les comportements entre des classes (ex : Interpreter)
  - Motifs de comportement d'objets avec l'utilisation de l'association entre objets :
    - pour décrire comment des groupes d'objets coopèrent ( Mediator)
    - pour définir et maintenir des dépendances entre objets (Observer)
    - pour encapsuler un comportement dans un objet et déléguer les requêtes à d'autres objets (Strategy, State, Command)
    - pour parcourir des structures en appliquant des comportements (Visitor, Iterator)



# Patterns de comportement

- **Chain of Responsibility** : A way of passing a request between a chain of objects
- **Command**: Encapsulate a command request as an object
- **Interpreter**: A way to include language elements in a program
- **Iterator**: Sequentially access the elements of a collection
- **Mediator**: Defines simplified communication between classes

## Patterns de comportement (2)

- **Memento:** Capture and restore an object's internal state
- **Observer:** A way of notifying change to a number of classes
- **State:** Alter an object's behavior when its state changes
- **Strategy:** Encapsulates an algorithm inside a class
- **Template Method:** Defer the exact steps of an algorithm to a subclass
- **Visitor:** Defines a new operation to a class without change

# Chain of Responsibility

**Definition :** Use sharing to support large numbers of fine-grained objects efficiently. Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

**Frequency of use:** medium low



## Objectifs :

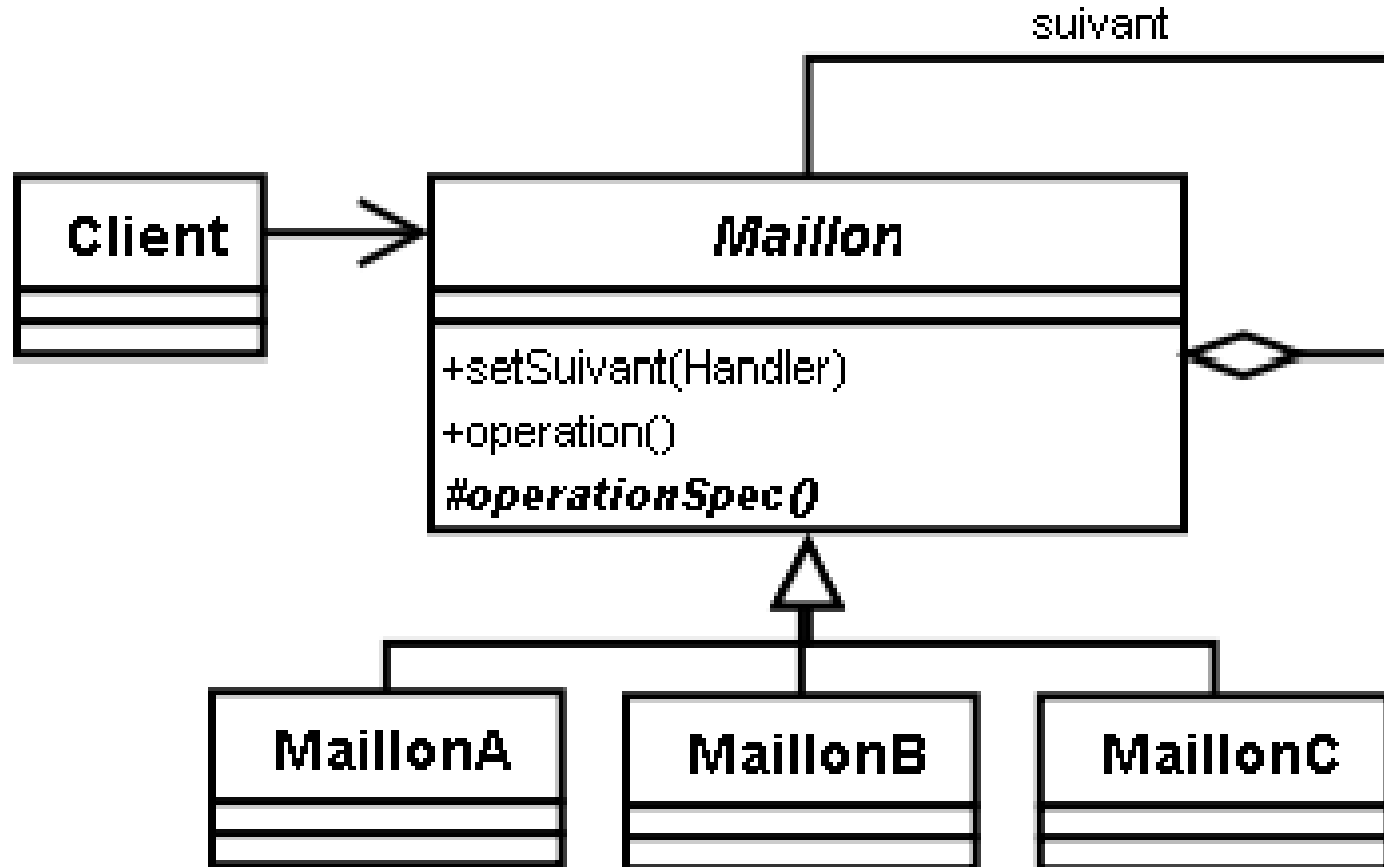
- Eviter le couplage entre l'**émetteur** d'une requête et son **récepteur** en donnant à plus d'un objet une chance de traiter la requête.
- Chaîner les objets récepteurs et passer la requête tout le long de la chaîne jusqu'à ce qu'un objet la traite.

**Résultat :** Il permet d'isoler les différentes parties d'un traitement.

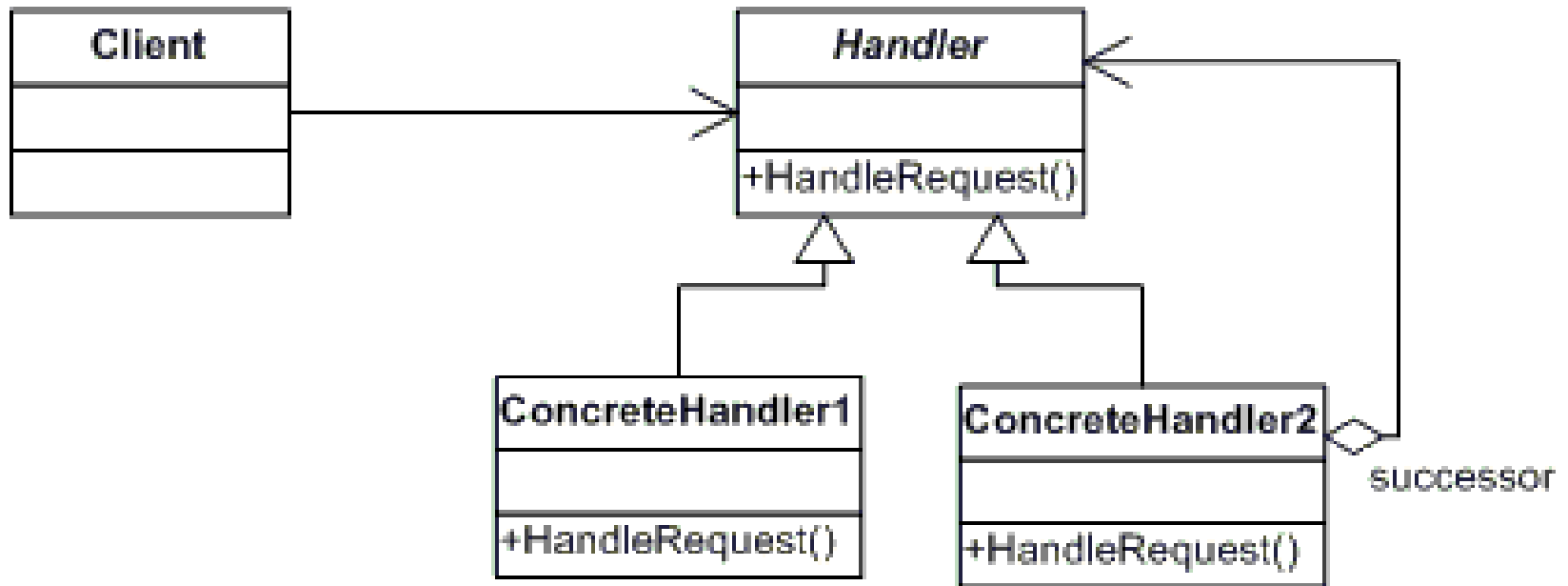
# Utilisation

- Le système doit gérer une requête. La requête implique plusieurs objets pour la traiter.
- Cela peut être le cas d'un système complexe d'habilitations possédant plusieurs critères afin d'autoriser l'accès. Ces critères peuvent varier en fonction de la configuration.
- Le traitement est réparti sur plusieurs objets : les maillons. Les maillons sont chaînés. Si un maillon ne peut réaliser le traitement (vérification des droits), il donne sa chance au maillon suivant. Il est facile de faire varier les maillons impliqués dans le traitement.

# Diagramme de classe



## Diagramme de classe (2)



# Caractéristiques

## Problème

- plus d'un objet peut traiter une requête et il n'est pas connu a priori
- l'ensemble des objets pouvant traiter une requête est construit dynamiquement

## Conséquences

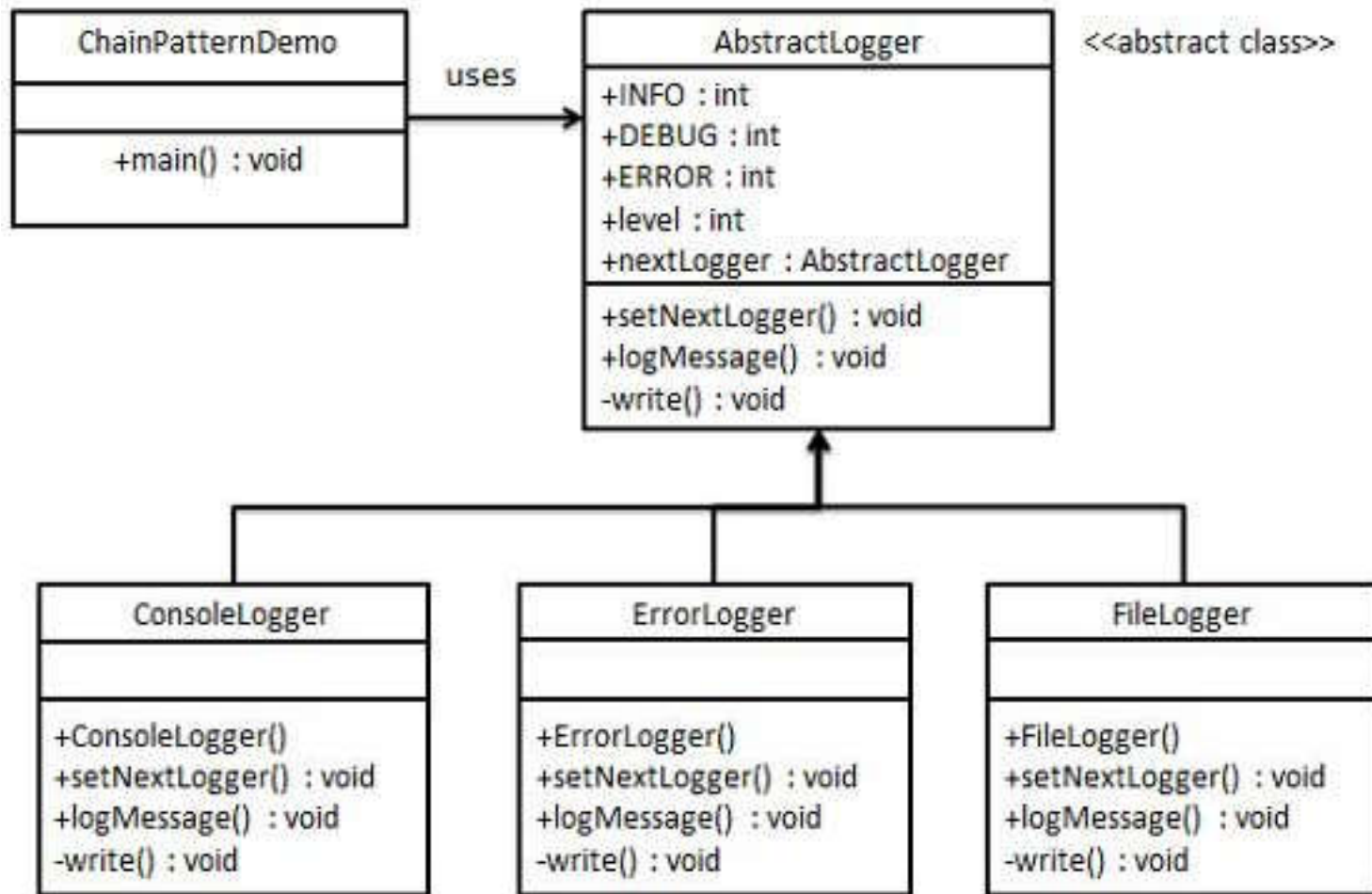
- + couplage réduit
- + flexibilité accrue pour l'assignation de responsabilités aux objets
- MAIS la réception n'est pas garantie

# Example 1

- **Handler (Approver)**
  - defines an interface for handling the requests
  - (optional) implements the successor link
- **ConcreteHandler (Director, VicePresident, President)**
  - handles requests it is responsible for
  - can access its successor
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- **Client (ChainApp)**
  - initiates the request to a ConcreteHandler object on the chain



## Exemple 2



# Abstract Logger

```
DESIGN PATTERNS

public enum Log{Info = 1, Debug, Error}
public abstract class AbstractLogger {
    protected int level;
    protected AbstractLogger nextLogger;
    public AbstractLogger(int level){
        this.level = level; }
    public void setNextLogger(AbstractLogger nextLogger){
        this.nextLogger = nextLogger;
    }
    public void logMessage(int level, String message){
        if(this.level <= level){ write(message); }
        if(nextLogger !=null){ nextLogger.logMessage(level, message); }
    }
    abstract protected void write(String message);
}
```

# ConsoleLogger, ErrorLogger & FileLogger

```
public class ConsoleLogger extends AbstractLogger {
    @Override
    protected void write(String message) {
        System.out.println("Standard Console::Logger: " + message); }
}
```

```
public class ErrorLogger extends AbstractLogger {
    @Override
    protected void write(String message) {
        System.out.println(" ErrorConsole::Logger: " + message); }
}
```

```
public class FileLogger extends AbstractLogger {
    @Override
    protected void write(String message) {
        System.out.println(" File::Logger: " + message); }
}
```

# Client

```
public class ChainPatternDemo {
    private static AbstractLogger getChainOfLoggers(){
        AbstractLogger errorLogger = new ErrorLogger(Log.Error);
        AbstractLogger fileLogger = new FileLogger(Log.Debug);
        AbstractLogger consoleLogger = new ConsoleLogger(Log.Info);
        errorLogger.setNextLogger(fileLogger);
        fileLogger.setNextLogger(consoleLogger);
        return errorLogger;
    }
    public static void main(String[] args) {
        AbstractLogger loggerChain = getChainOfLoggers();
        loggerChain.logMessage(Log.Info, "This is an information.");
        loggerChain.logMessage(Log.Debug, "This is an debug level information.");
        loggerChain.logMessage(Log.Error, "This is an error information.");
    }
}
```

# Command

**Definition :** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

**Frequency of use:** medium high



## Objectifs :

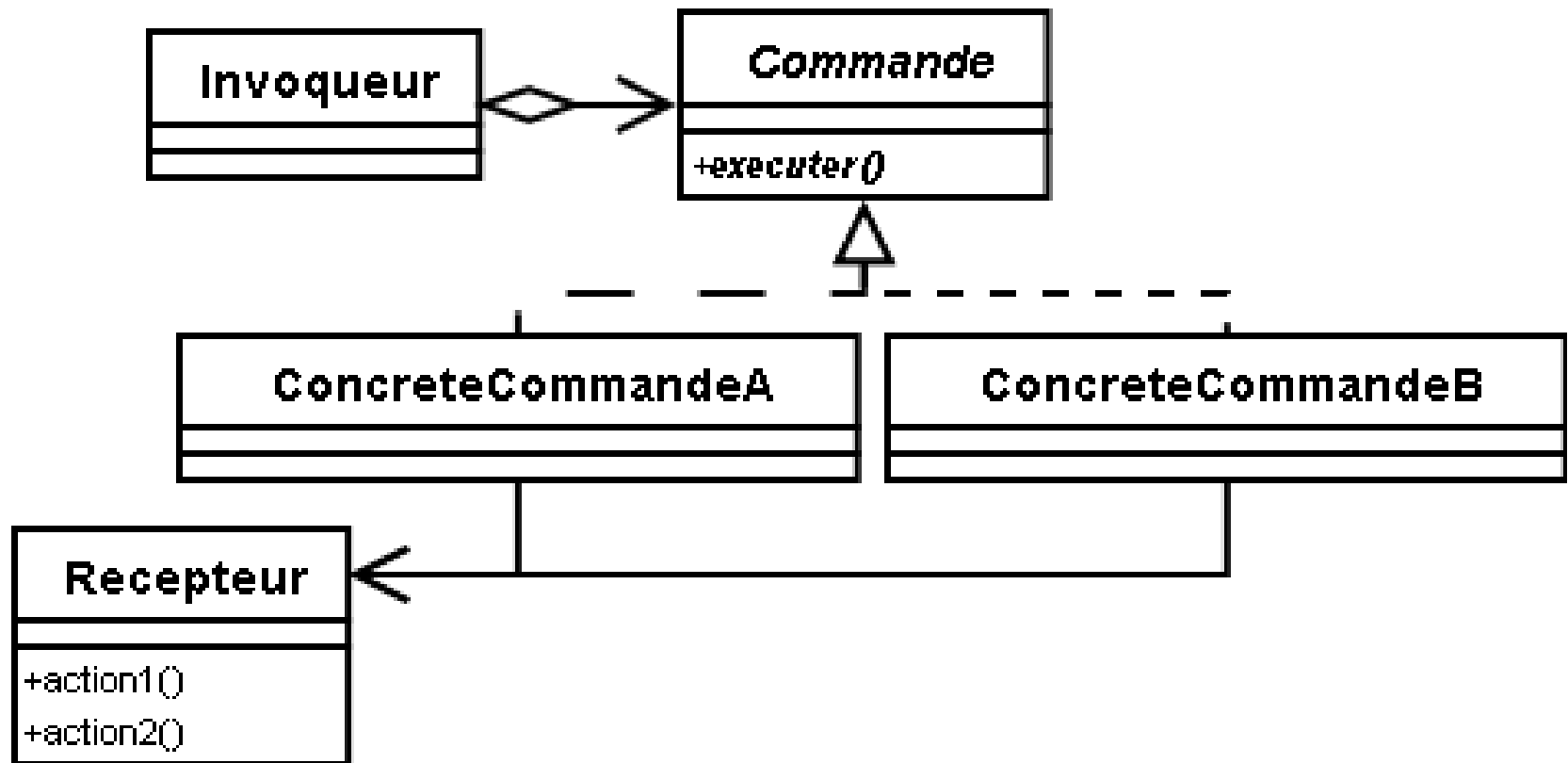
- Encapsuler une **requête** sous la forme **d'objet**.
- Paramétrer facilement des requêtes diverses.
- Permettre des **opérations** réversibles.

**Résultat :** Le Design Pattern permet d'isoler une requête.

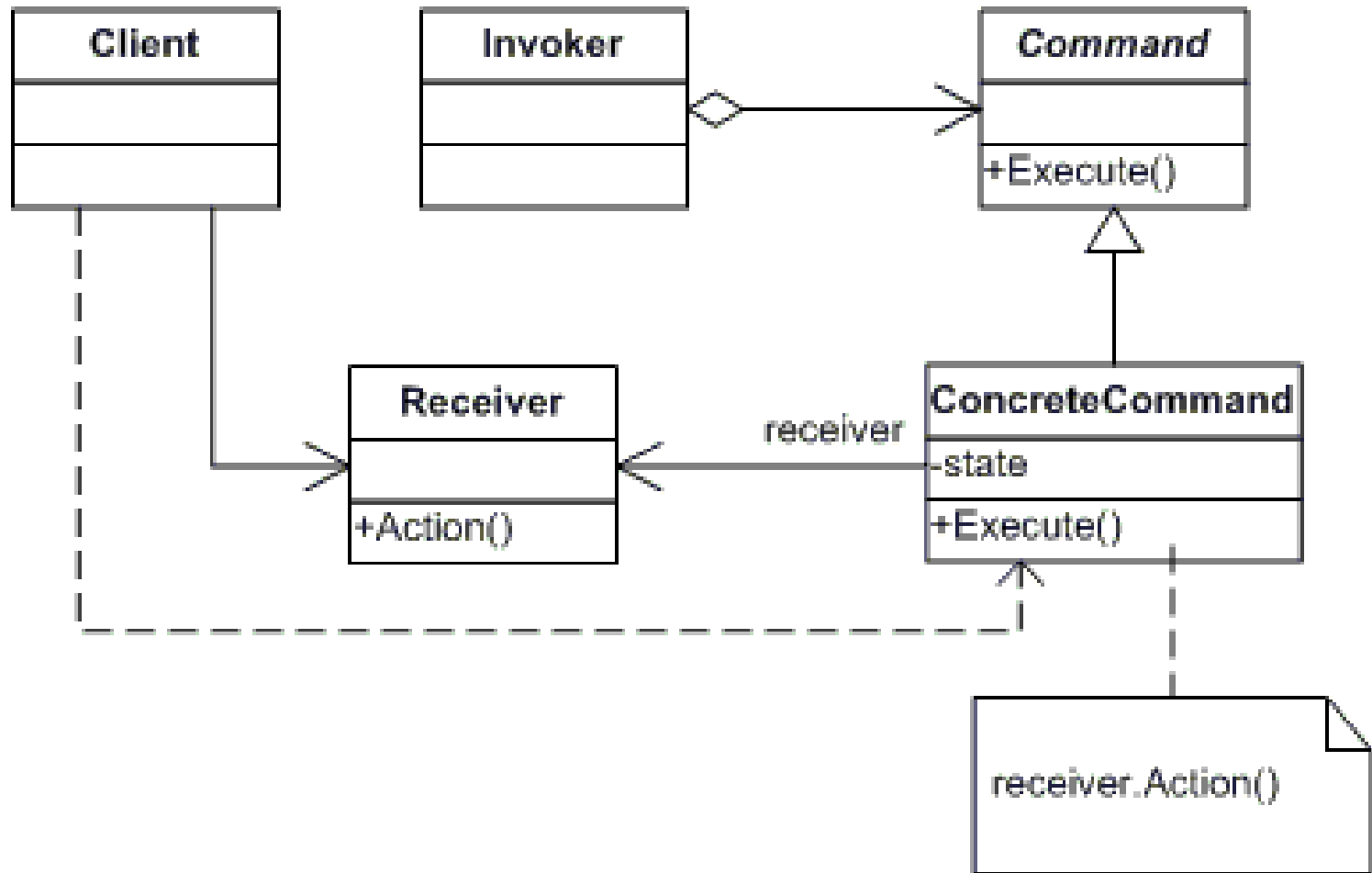
# Utilisation

- Le système doit traiter des requêtes. Ces requêtes peuvent provenir de plusieurs **émetteurs** qui peuvent produire la même requête. Les requêtes doivent pouvoir être annulées.
- Cela peut être le cas d'une IHM avec des boutons de commande, des raccourcis clavier et des choix de menu aboutissant à la même requête.
- La requête est encapsulée dans un objet : la **commande**.
- Chaque commande possède un objet qui traitera la requête : le **récepteur**.
- La commande ne réalise pas le traitement, elle est juste porteuse de la requête. Les émetteurs potentiels de la requête (éléments de l'IHM) sont des **invoqueurs** qui peuvent se partager la même commande.

# Diagramme de classe



## Diagramme de classe (2)





# Caractéristiques

## Problème

- on veut effectuer des requêtes sur des objets sans avoir à connaître leur structure

## Conséquences

- + découplage entre l'objet qui appelle et celui qui exécute
- + l'ajout de nouvelles commandes est aisée dans la mesure où la modification de classes existantes n'est pas nécessaire

# Receiver classes : FileSystemReceiver

```
public interface FileSystemReceiver {  
    void openFile();  
    void writeFile();  
    void closeFile();  
}
```

# Receiver classes : UnixFileSystemReceiver

```
public class UnixFileSystemReceiver implements
    FileSystemReceiver {
    public void openFile() {
        System.out.println("Opening file in unix OS");
    }
    public void writeFile() {
        System.out.println("Writing file in unix OS");
    }
    public void closeFile() {
        System.out.println("Closing file in unix OS");
    }
}
```

# Receiver classes: WindowsFileSystemReceiver

```
public class WindowsFileSystemReceiver implements
    FileSystemReceiver {
    public void openFile() {
        System.out.println("Opening file in windows OS");
    }
    public void writeFile() {
        System.out.println("Writing file in windows OS");
    }
    public void closeFile() {
        System.out.println("Closing file in windows OS");
    }
}
```

# Command classes : Command Interface

```
public interface Command {  
    void execute();  
}
```

# Command classes : OpenFileCommand

```
public class OpenFileCommand implements Command {  
    private FileSystemReceiver fileSystem;  
  
    public OpenFileCommand(FileSystemReceiver fs){  
        this.fileSystem=fs;  
    }  
  
    public void execute() {  
        //open command is forwarding request to openFile method  
        this.fileSystem.openFile();  
    }  
}
```

# Command classes : CloseFileCommand

```
public class CloseFileCommand implements Command {  
    private FileSystemReceiver fileSystem;  
  
    public CloseFileCommand(FileSystemReceiver fs){  
        this.fileSystem=fs;  
    }  
  
    public void execute() {  
        //close command is forwarding request to closeFile method  
        this.fileSystem.closeFile();  
    }  
}
```

# Command classes : WriteFileCommand

DESIGN PATTERNS

```
public class WriteFileCommand implements Command {  
    private FileSystemReceiver fileSystem;  
  
    public WriteFileCommand(FileSystemReceiver fs){  
        this.fileSystem=fs;  
    }  
  
    public void execute() {  
        //write command is forwarding request to writeFile method  
        this.fileSystem.writeFile();  
    }  
}
```



# Invoker classe : FileInvoker

```
public class FileInvoker {  
    public Command command;  
    public FileInvoker(Command c){  
        this.command=c;  
    }  
    public void execute(){  
        this.command.execute();  
    }  
}
```

## Factory method classe : FileSystemReceiverUtil

```
public static FileSystemReceiver getUnderlyingFileSystem() {  
    String osName = System.getProperty("os.name");  
    System.out.println("Underlying OS is:"+osName);  
    if(osName.contains("Windows")){  
        return new WindowsFileSystemReceiver();  
    }  
    else{  
        return new UnixFileSystemReceiver();  
    }  
}
```

## Client : FileSystemClient

```
public class FileSystemClient {  
    public static void main(String[] args) {  
        //Creating the receiver object  
        FileSystemReceiver fs = FileSystemReceiverUtil.getUnderlyingFileSystem();  
        //creating command and associating with receiver OpenFileCommand  
        Command openFileCommand = new OpenFileCommand(fs);  
        //Creating invoker and associating with Command  
        FileInvoker file = new FileInvoker(openFileCommand);  
        //perform action on invoker object  
        file.execute();  
        Command writeFileCommand = new WriteFileCommand(fs);  
        file = new FileInvoker(writeFileCommand);  
        file.execute();  
        Command closeFileCommand = new CloseFileCommand(fs);  
        file = new FileInvoker(closeFileCommand);  
        file.execute(); }  
}
```

# Test

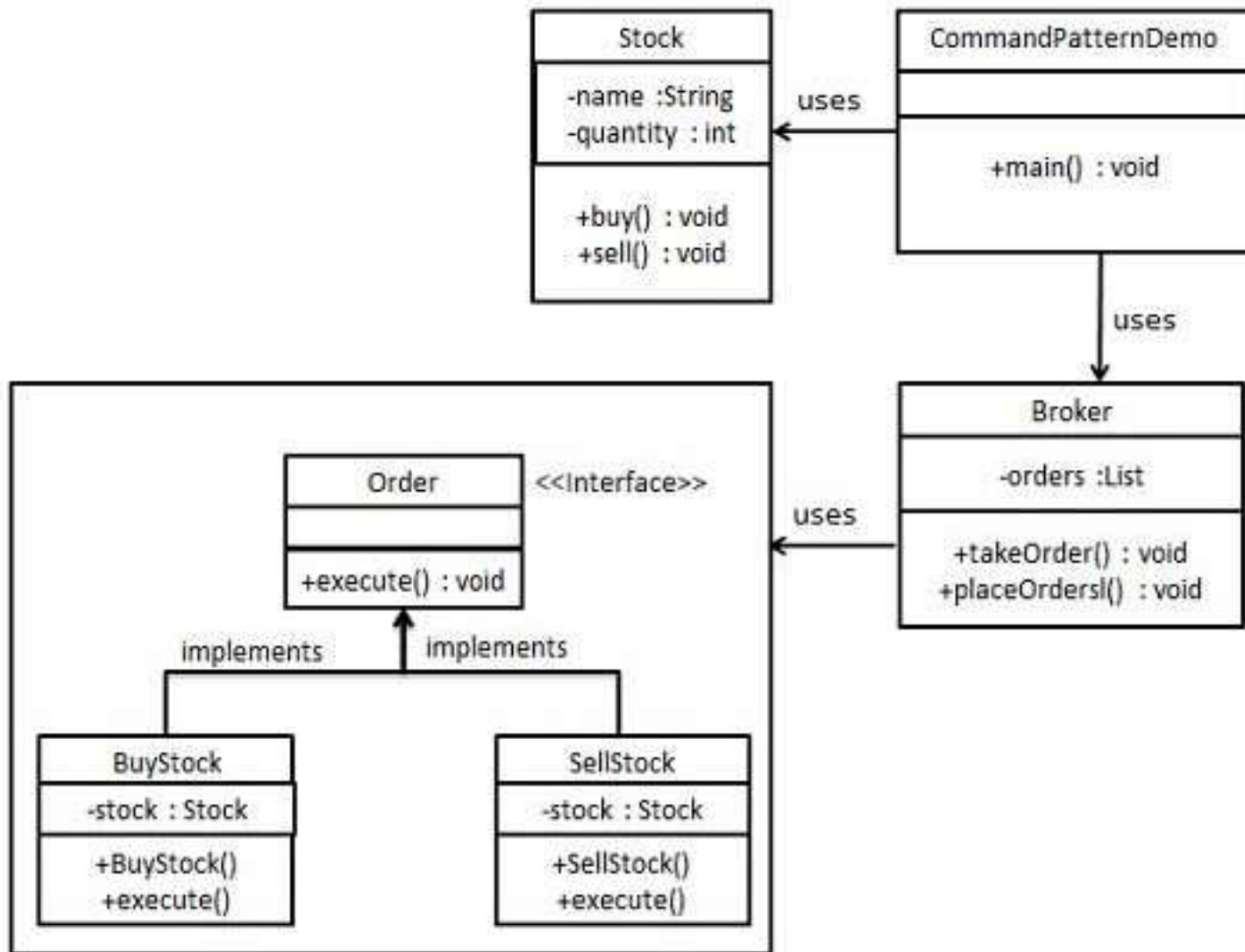
Underlying OS is: Windows OS

Opening file in Windows OS

Writing file in Windows OS

Closing file in Windows OS

## Exemple 2



# Exemple 3

- **Command (Command)**
  - declares an interface for executing an operation
- **ConcreteCommand (CalculatorCommand)**
  - defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operation(s) on Receiver
- **Client (CommandApp)**
  - creates a ConcreteCommand object and sets its receiver
- **Invoker (User)**
  - asks the command to carry out the request
- **Receiver (Calculator)**
  - knows how to perform the operations associated with carrying out the request.

# Interpreter

**Definition :** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

**Frequency of use:** low



## Objectifs :

- Définir une représentation de la grammaire d'un **langage**.
- Utiliser cette représentation pour interpréter les éléments de ce langage.

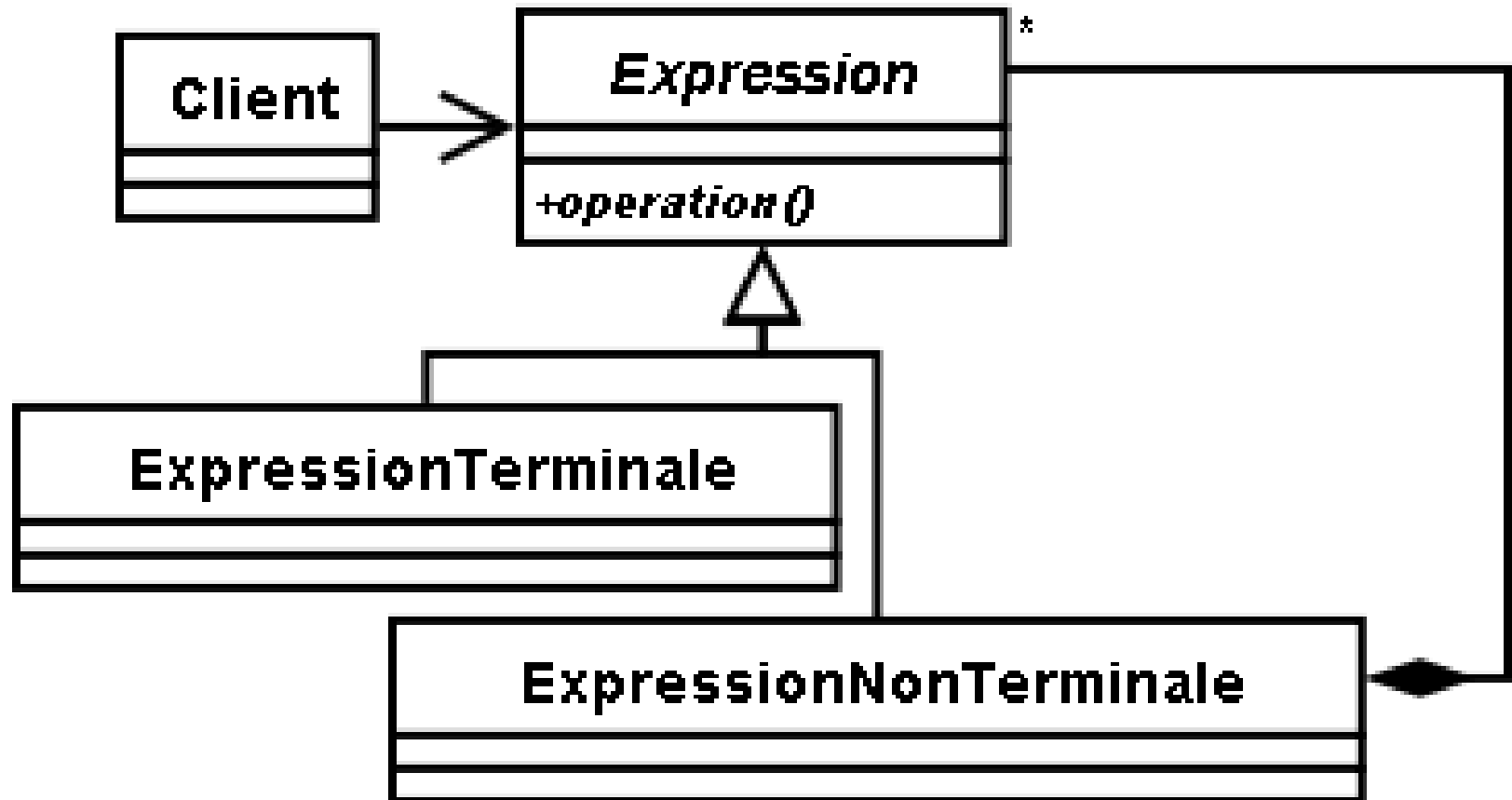
**Résultat :** Le Design Pattern permet d'isoler les éléments d'un langage.

# Utilisation

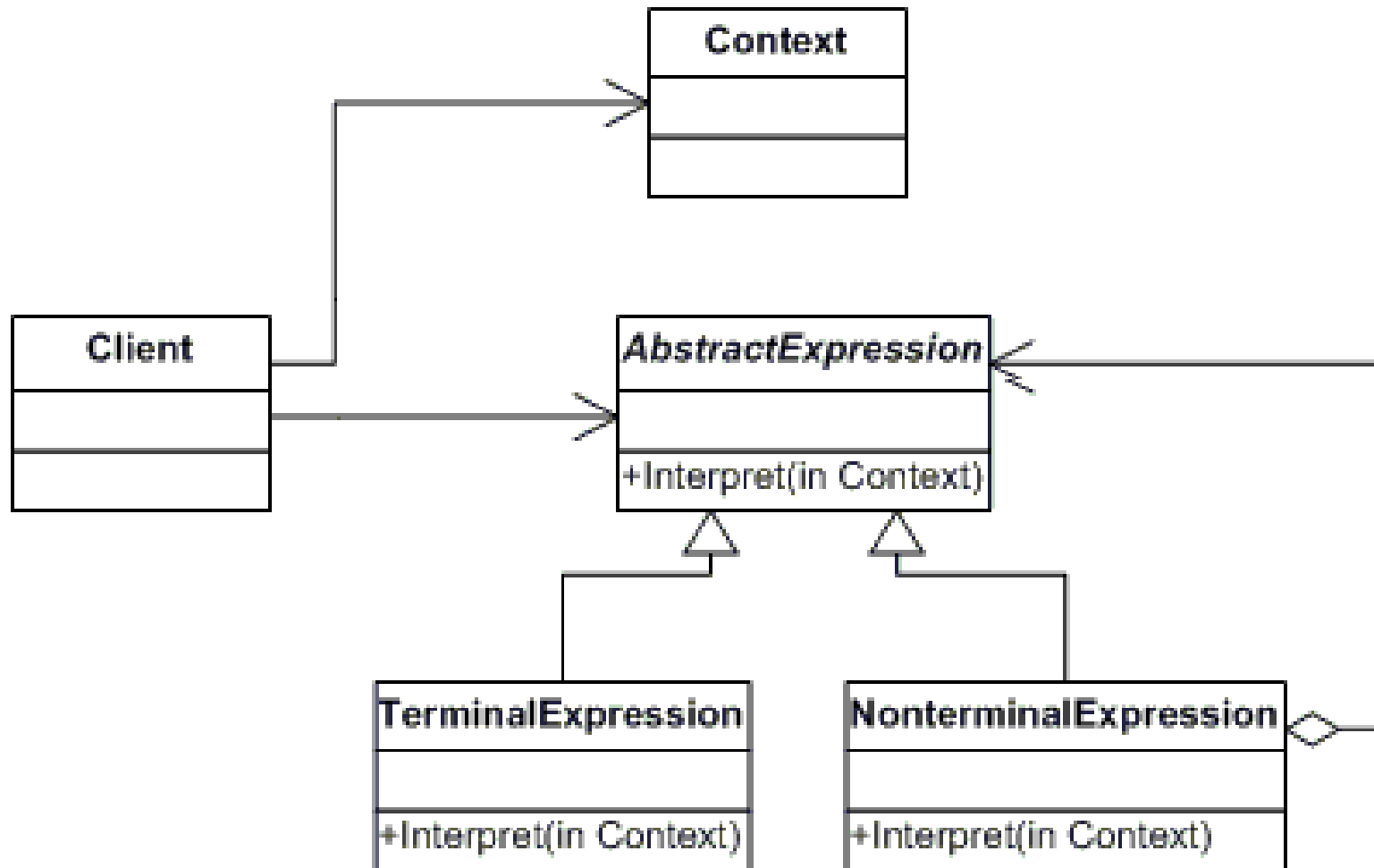
- Le système doit interpréter un langage. Ce langage possède une grammaire prédéfinie qui constitue un ensemble d'**opérations** qui peuvent être effectuées par le système.
- Cela peut être le cas d'un logiciel embarqué dont la configuration des écrans serait stockée dans des fichiers XML. Le logiciel lit ces fichiers afin de réaliser son affichage et l'enchaînement des écrans.
- Une structure arborescente peut représenter la grammaire du langage. Elle permet d'interpréter les différents éléments du langage.



# Diagramme de classe



## Diagramme de classe (2)



# Caractéristiques

## Problème

- ce motif est à utiliser lorsque l'on veut représenter la grammaire d'un langage et l'interpréter, lorsque :
  - La grammaire est simple
  - l'efficacité n'est pas critique

## Conséquences

- + facilité de changer et d'étendre la grammaire
- + l'implémentation de la grammaire est simple
- MAIS les grammaires complexes sont dures à tenir à jour

# Exemple

- **AbstractExpression (Expression)**
  - declares an interface for executing an operation
- **TerminalExpression (ThousandExpression, HundredExpression, TenExpression, OneExpression )**
  - implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in the sentence.

# Exemple (suite)

- **NonterminalExpression ( not used )**
  - one such class is required for every rule  $R ::= R_1 R_2 \dots R_n$  in the grammar
  - maintains instance variables of type AbstractExpression for each of the symbols  $R_1$  through  $R_n$ .
  - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing  $R_1$  through  $R_n$ .

# Exemple (suite)

- **Context (Context)**
  - contains information that is global to the interpreter
- **Client (InterpreterApp)**
  - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
  - invokes the Interpret operation

# Iterator

**Definition :** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Frequency of use:** high



**Objectifs :** Fournir un moyen de parcourir séquentiellement les éléments d'un **objet composé**.

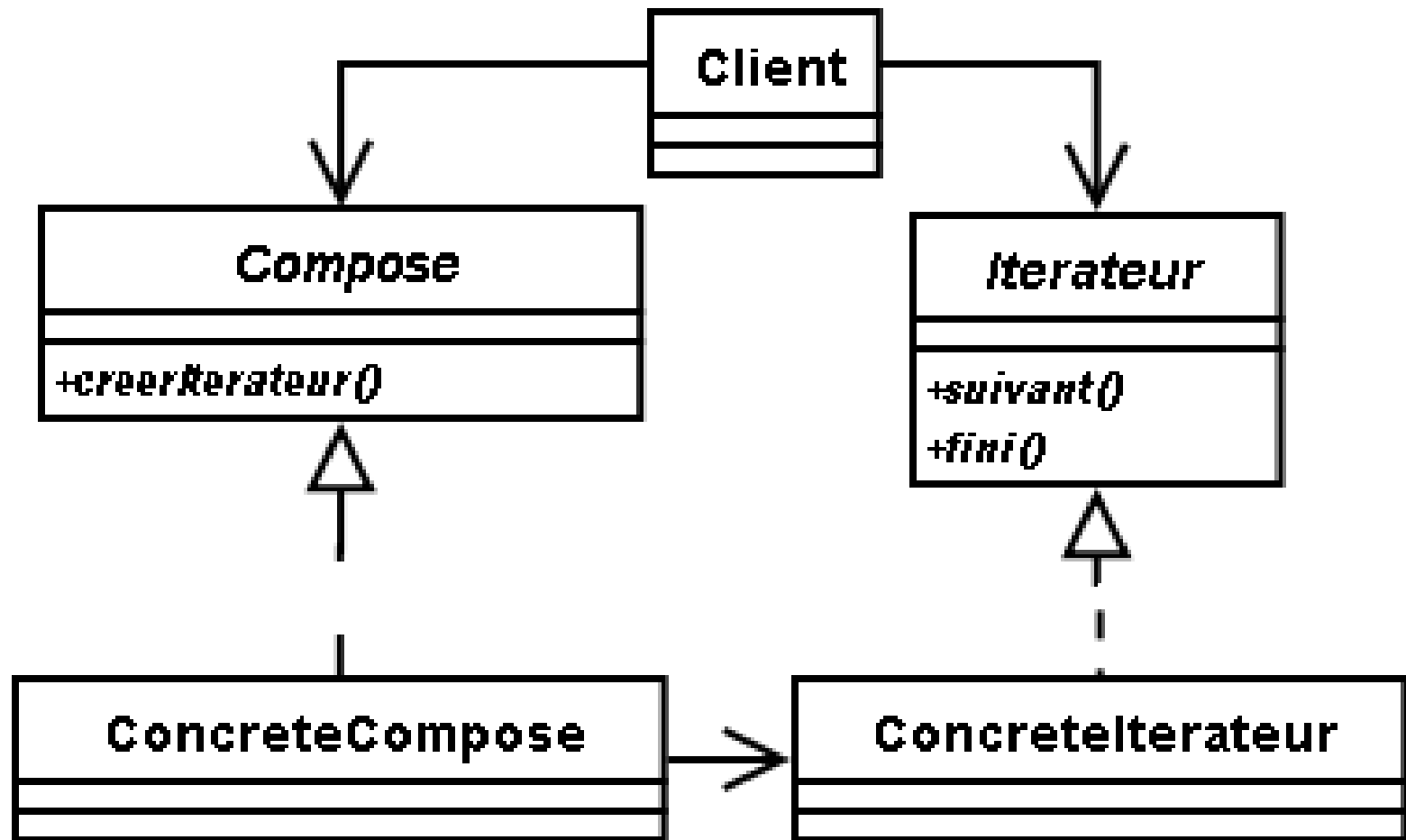
**Résultat :** Le Design Pattern permet d'isoler le parcours d'un agrégat.

# Utilisation

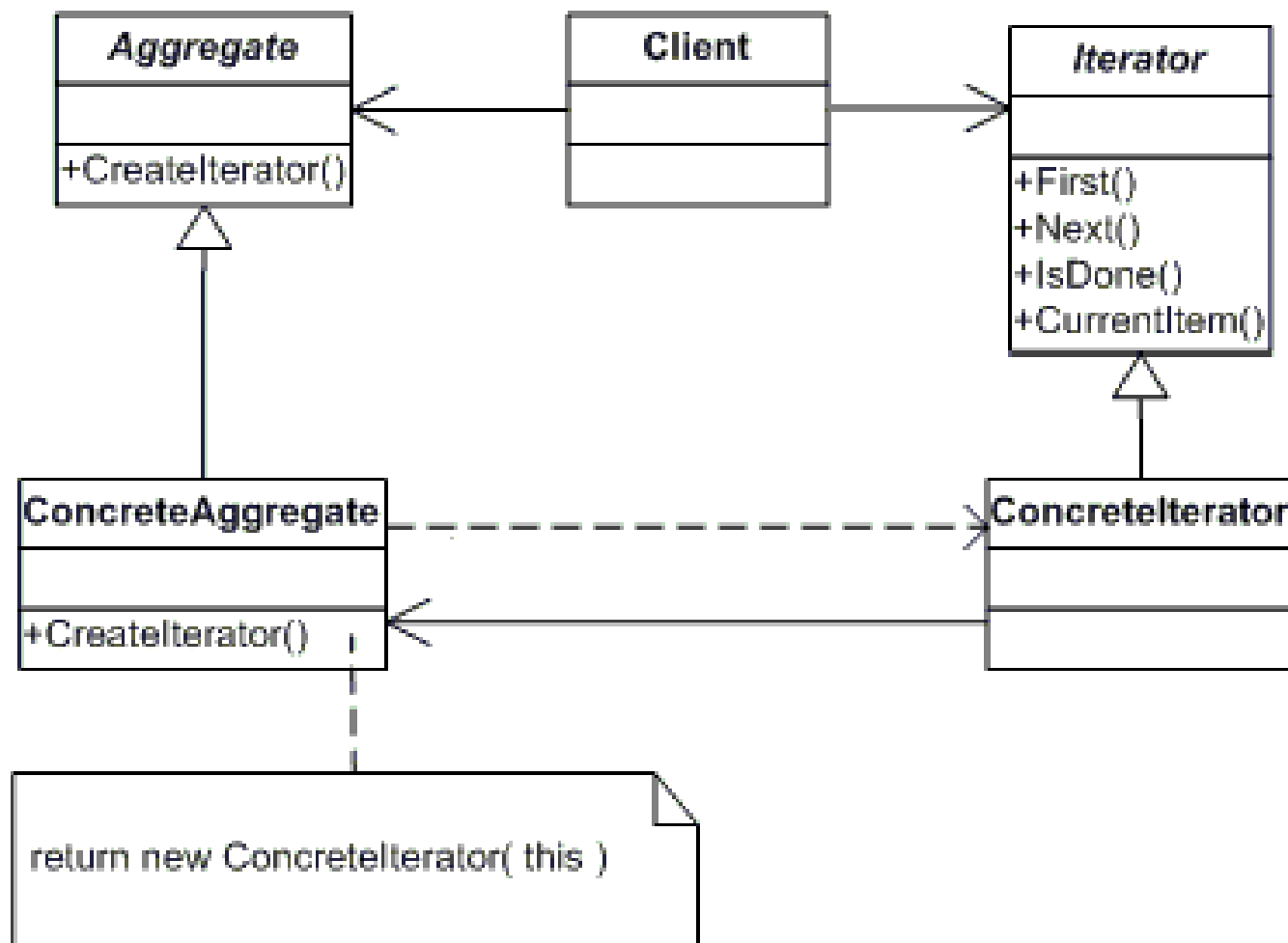
- Le système doit parcourir les éléments d'un objet complexe. La classe de l'objet complexe peut varier.
- Cela est le cas des classes représentant des listes et des ensembles en Java.
- Les classes d'un objet complexe (listes) sont des "composes". Elles ont une méthode retournant un itérateur, qui permet de parcourir les éléments. Tous les itérateurs ont la même **interface**. Ainsi, le système dispose d'un moyen homogène de parcourir les composes.



# Diagramme de classe



## Diagramme de classe (2)



# Caractéristiques

## Problème

- ce motif est à utiliser pour parcourir une collection d'éléments sans accéder à sa structure interne

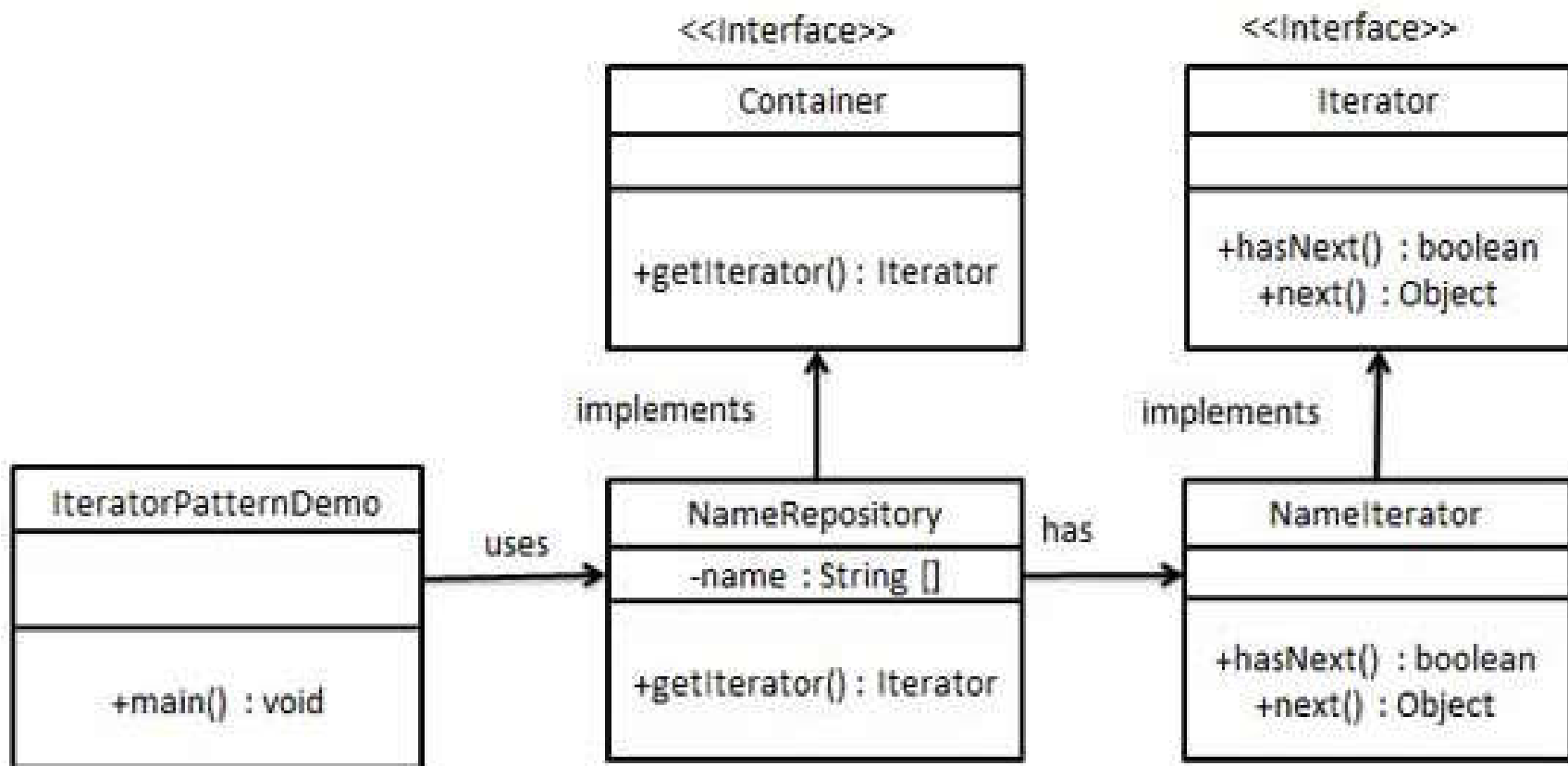
## Conséquences

- + des variations dans le parcours d'une collection sont possibles,
- + simplification de l'interface de la collection
- + plusieurs parcours simultanés de la collection sont possibles

# Exemple

- **Iterator (AbstractIterator)**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator (Iterator)**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate (AbstractCollection)**
  - defines an interface for creating an Iterator object
- **ConcreteAggregate (Collection)**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

## Exemple 2



# Mediator

**Definition :** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

**Frequency of use:** medium low



## Objectifs :

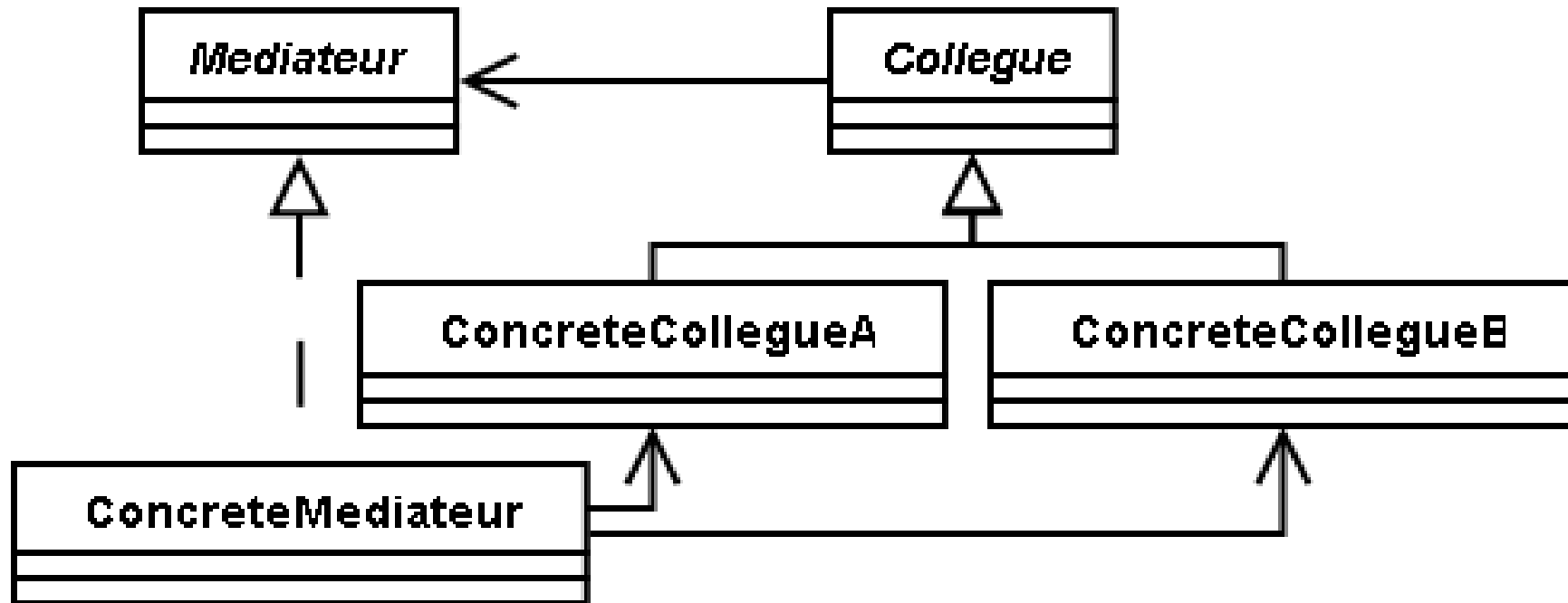
- Gérer la transmission d'informations entre des **objets** interagissant entre eux.
- Avoir un couplage faible entre les objets puisqu'ils n'ont pas de lien direct entre eux.
- Pouvoir varier leur interaction indépendamment.

**Résultat :** Il permet d'isoler la communication entre des objets.

# Utilisation

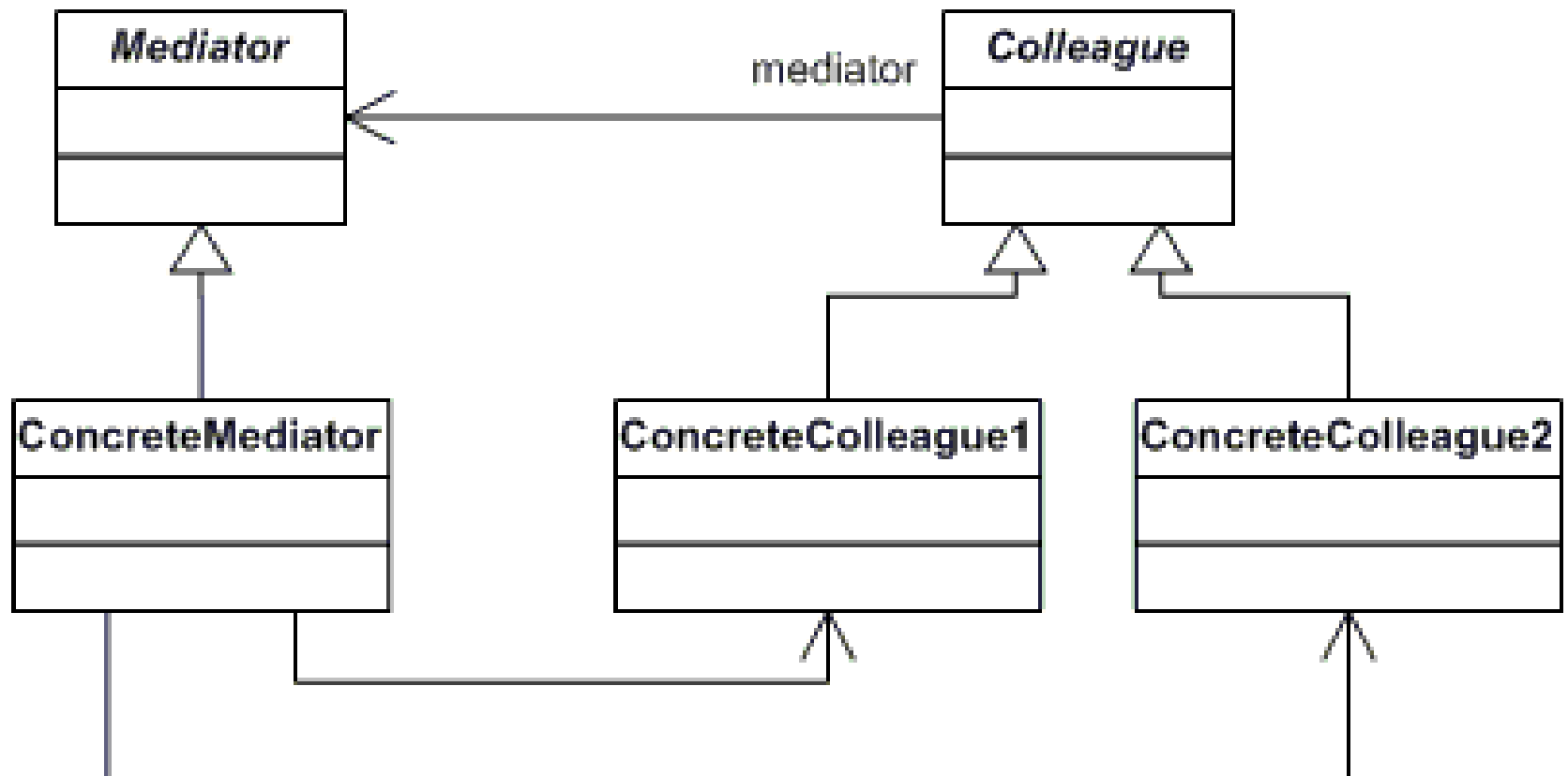
- Différents objets ont des interactions. Un événement sur l'un provoque une action ou des actions sur un autre ou d'autres objets.
- Cela peut être les éléments d'IHM. Si une case est cochée, certains éléments deviennent accessibles. Si une autre case est cochée, des couleurs de l'IHM changent.
- Si les **classes** communiquent directement, il y a un couplage très fort entre elles. Une classe dédiée à la communication permet d'éviter cela. Chaque élément interagissant (élément de l'IHM) sont des collègues. La classe dédiée à la communication est un médiateur.

# Diagramme de classe





## Diagramme de classe (2)



# Caractéristiques

## Problème

- Assurer l'interaction entre différents objets en assurant leur indépendance :
  - Les interactions entre les objets sont bien définies mais conduisent à des interdépendances difficiles à comprendre, ou
  - La réutilisation d'un objet est difficile de part ses interactions avec plusieurs objets

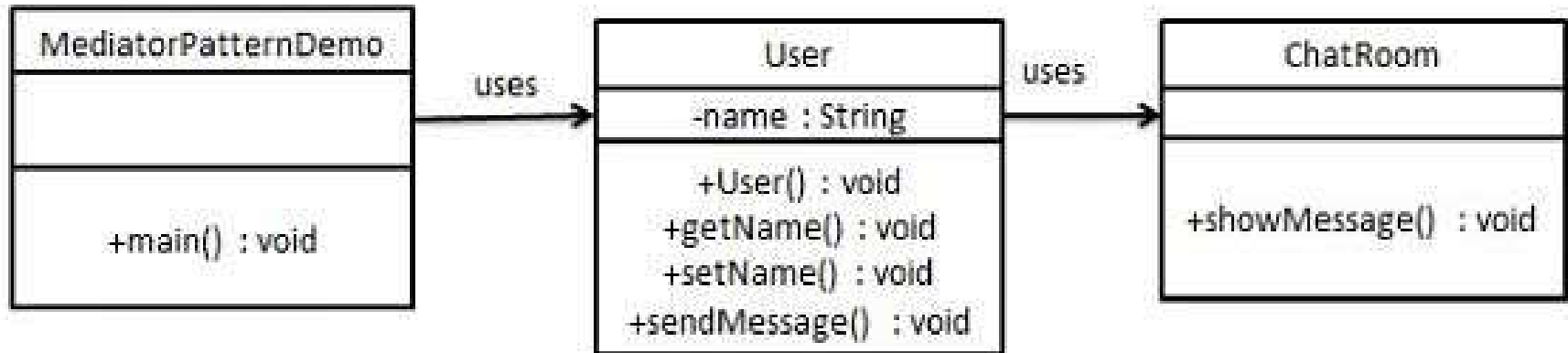
## Conséquences

- + limitation de la compartimentation : le médiateur contient le comportement qui serait distribué sinon entre les différents objets, indépendance des "Colleagues"
- + simplification des protocoles (many-to-many → one-to-many)
- + abstraction des coopérations entre objets
- MAIS centralisation du contrôle, complexité possible du Médiateur

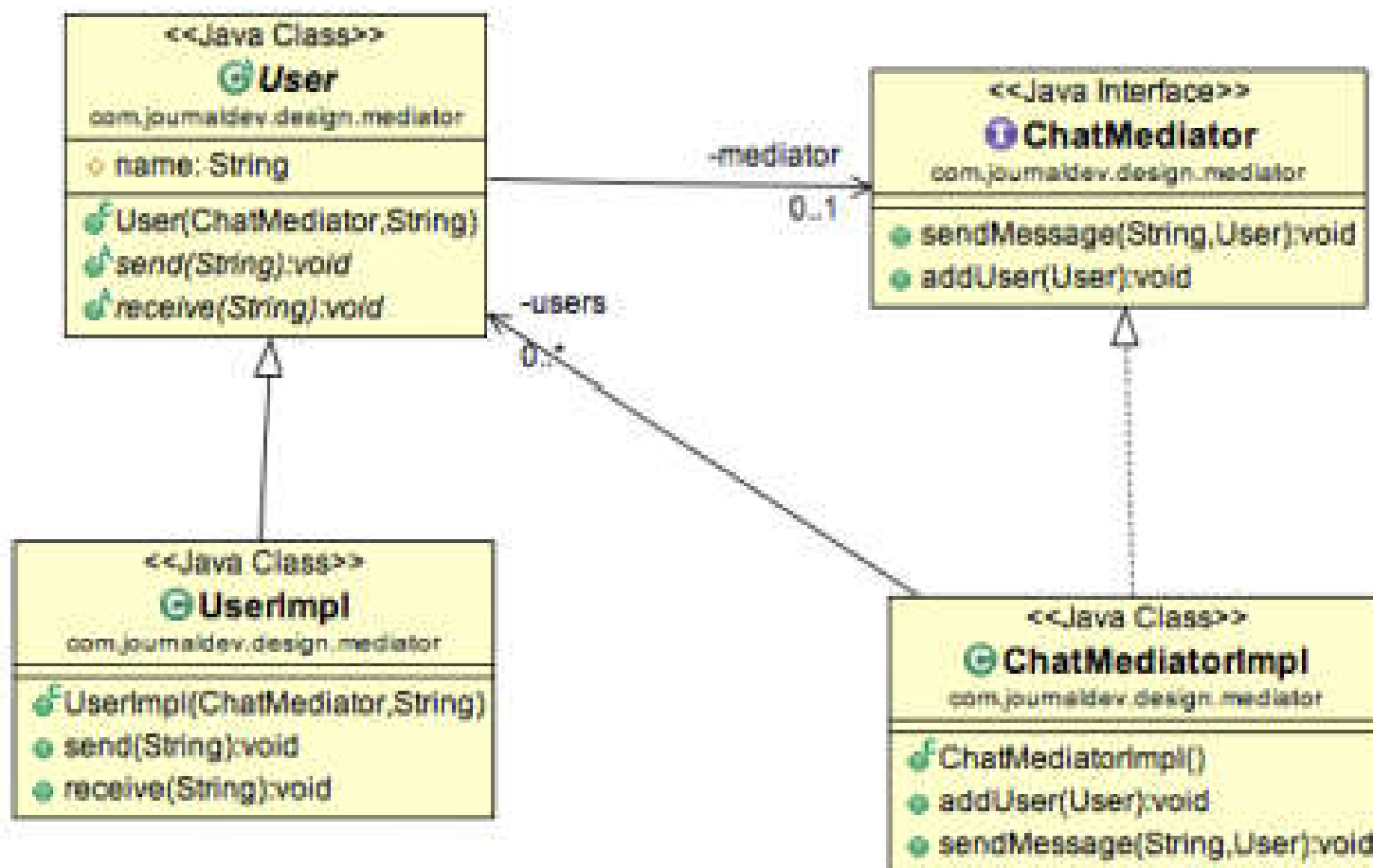
# Exemple

- **Mediator (IChatroom)**
  - defines an interface for communicating with Colleague objects
- **ConcreteMediator (Chatroom)**
  - implements cooperative behavior by coordinating Colleague objects
  - knows and maintains its colleagues
- **Colleague classes (Participant)**
  - each Colleague class knows its Mediator object
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

# Exemple 1



## Exemple 2



# Memento

**Definition :** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

**Frequency of use:** low



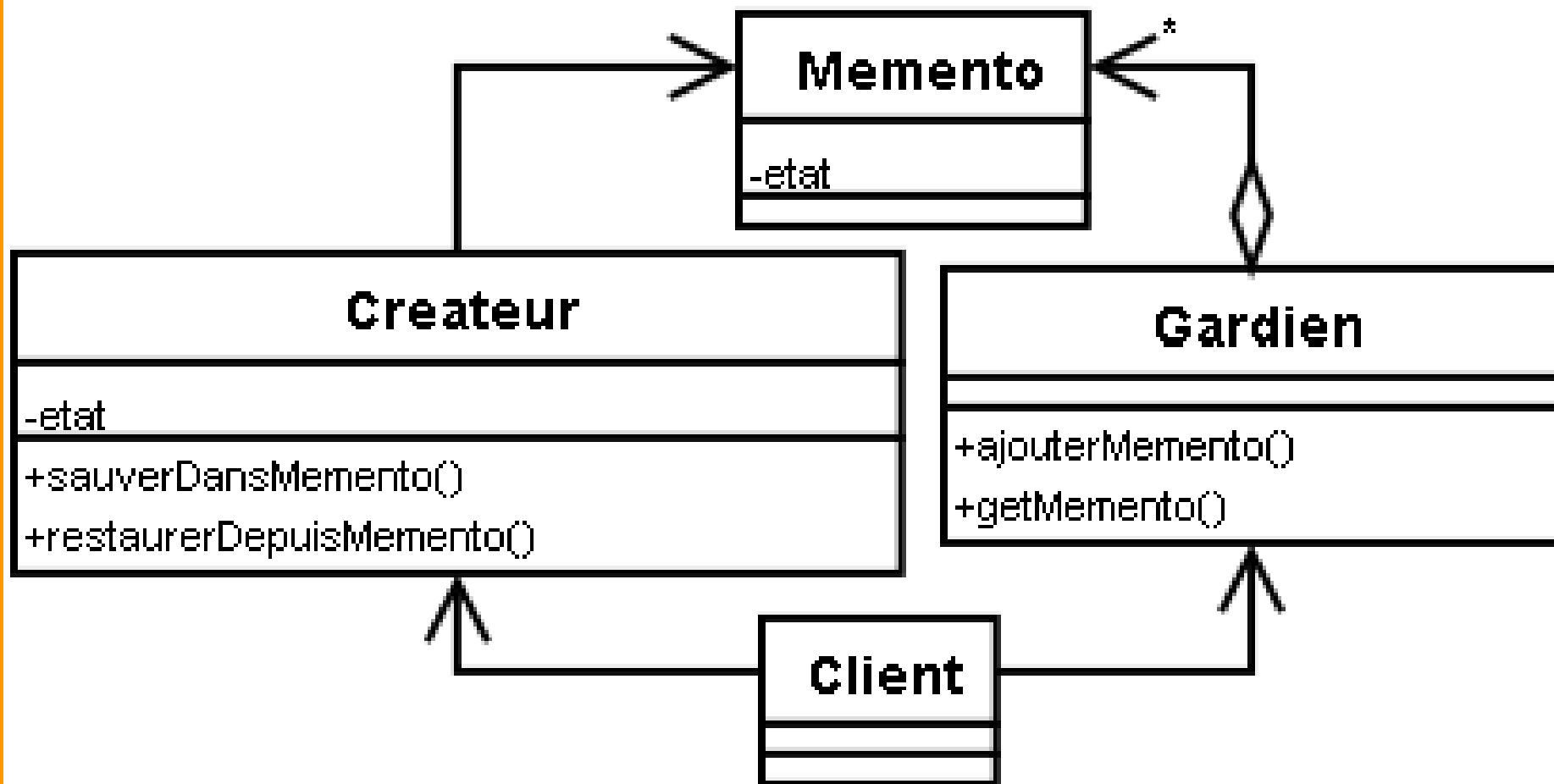
**Objectifs :** Sauvegarder l' **état interne** d'un objet en respectant l'**encapsulation**, afin de le restaurer plus tard.

**Résultat :** Il permet d'isoler la conservation de l'état d'un objet.

# Utilisation

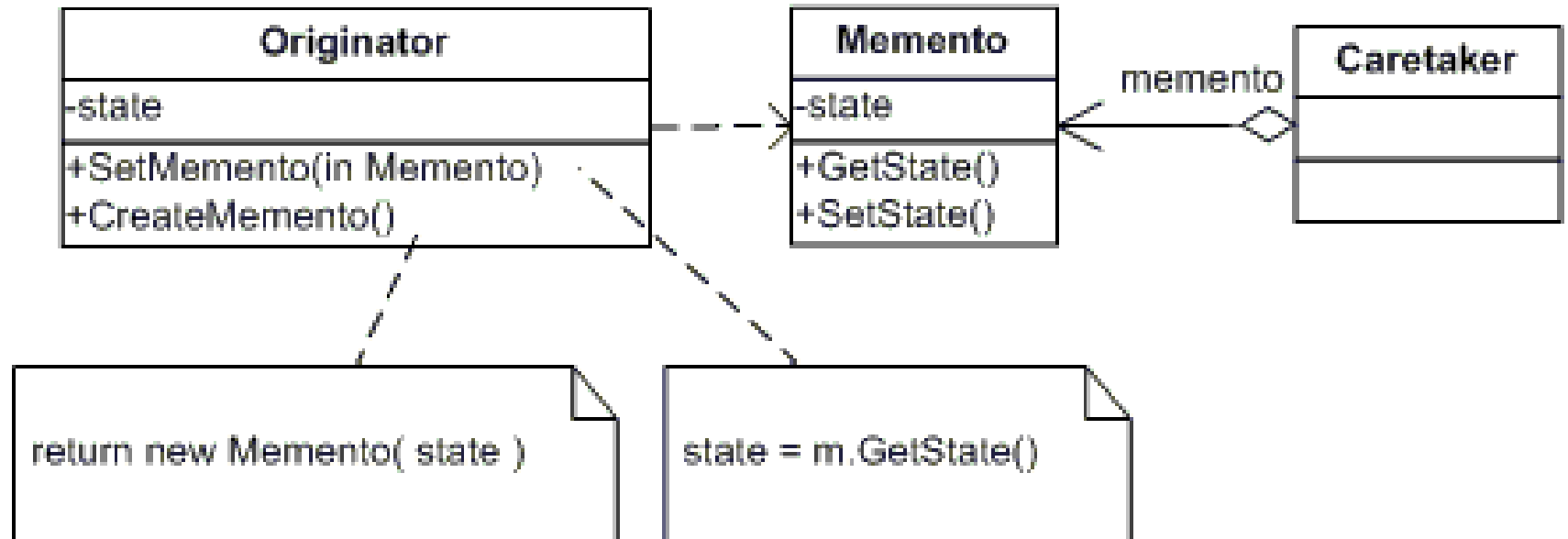
- Un système doit conserver et restaurer l'état d'un objet. L'état interne de l'objet à conserver n'est pas visible par les autres objets.
- Cela peut être un éditeur de document disposant d'une fonction d'annulation. La fonction d'annulation est sur plusieurs niveaux.
- Les informations de l'état interne (état du document) sont conservées dans un memento. L'objet avec l'état interne (document) est le créateur du memento. Afin de respecter l'encapsulation, les valeurs du memento ne sont visibles que par son créateur. Ainsi, l'encapsulation de l'état interne est préservée. Un autre objet est chargé de conserver les mementos (gestionnaire d'annulation) : il s'agit du gardien.

# Diagramme de classe





## Diagramme de classe (2)



# Caractéristiques

## Problème

- on veut sauvegarder l'état ou une partie de l'état d'un objet
- **sans** violer le principe d'encapsulation

## Conséquences

- + garder les limites de l'encapsulation
- peut-être coûteux en mémoire et en exécution

# Exemple

- **Memento (Memento)**

- stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
- protect against access by objects of other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento -- it can only pass the memento to the other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state.

# Exemple (suite)

- **Originator (SalesProspect)**
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state
- **Caretaker (Caretaker)**
  - is responsible for the memento's safekeeping
  - never operates on or examines the contents of a memento.

# Observer

**Definition :** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Frequency of use:** high



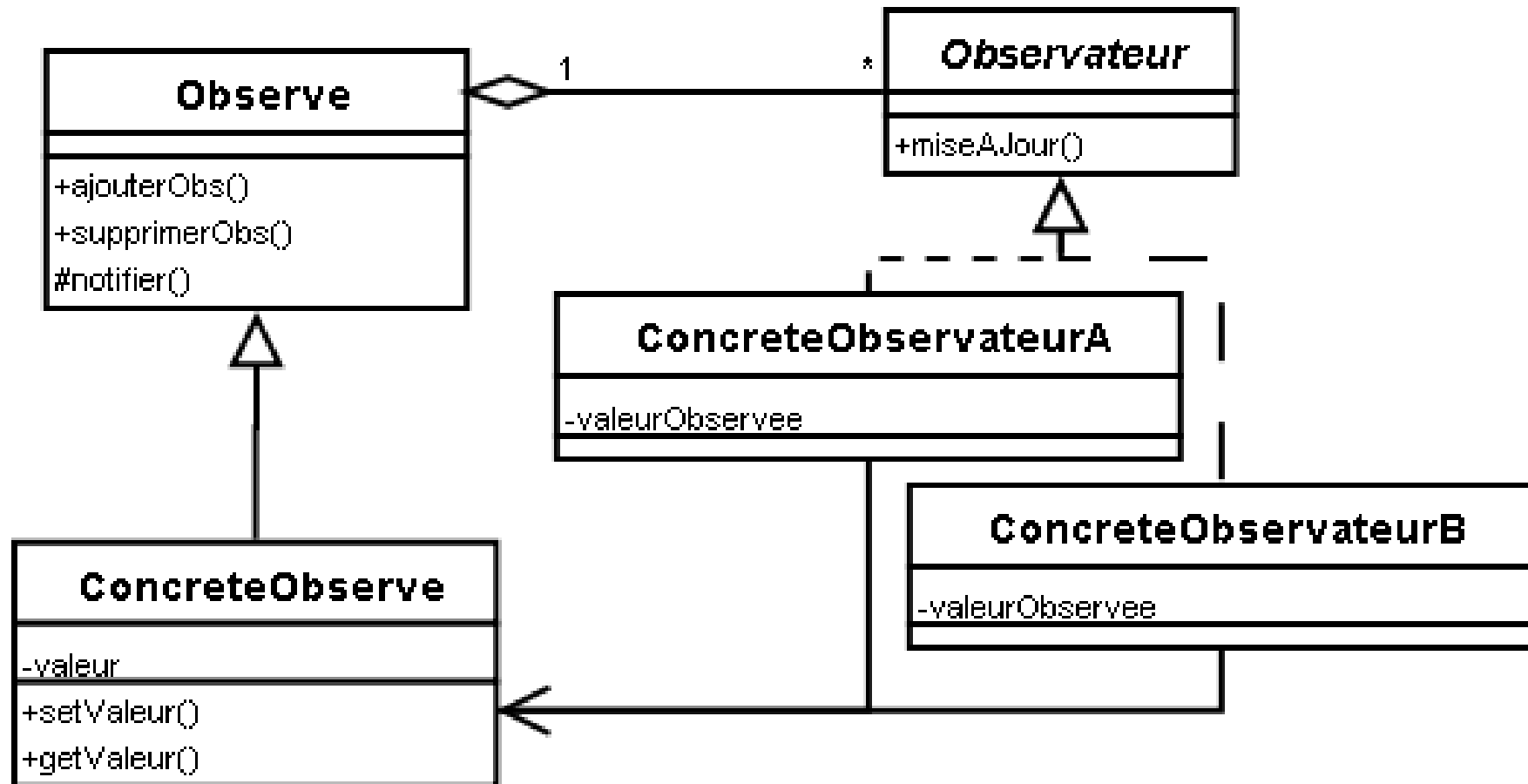
**Objectifs :** Prévenir des **objets** observateurs, enregistrés auprès d'un objet observé, d'un **événement**.

**Résultat :** Le Design Pattern permet d'isoler un **algorithme** traitant un événement.

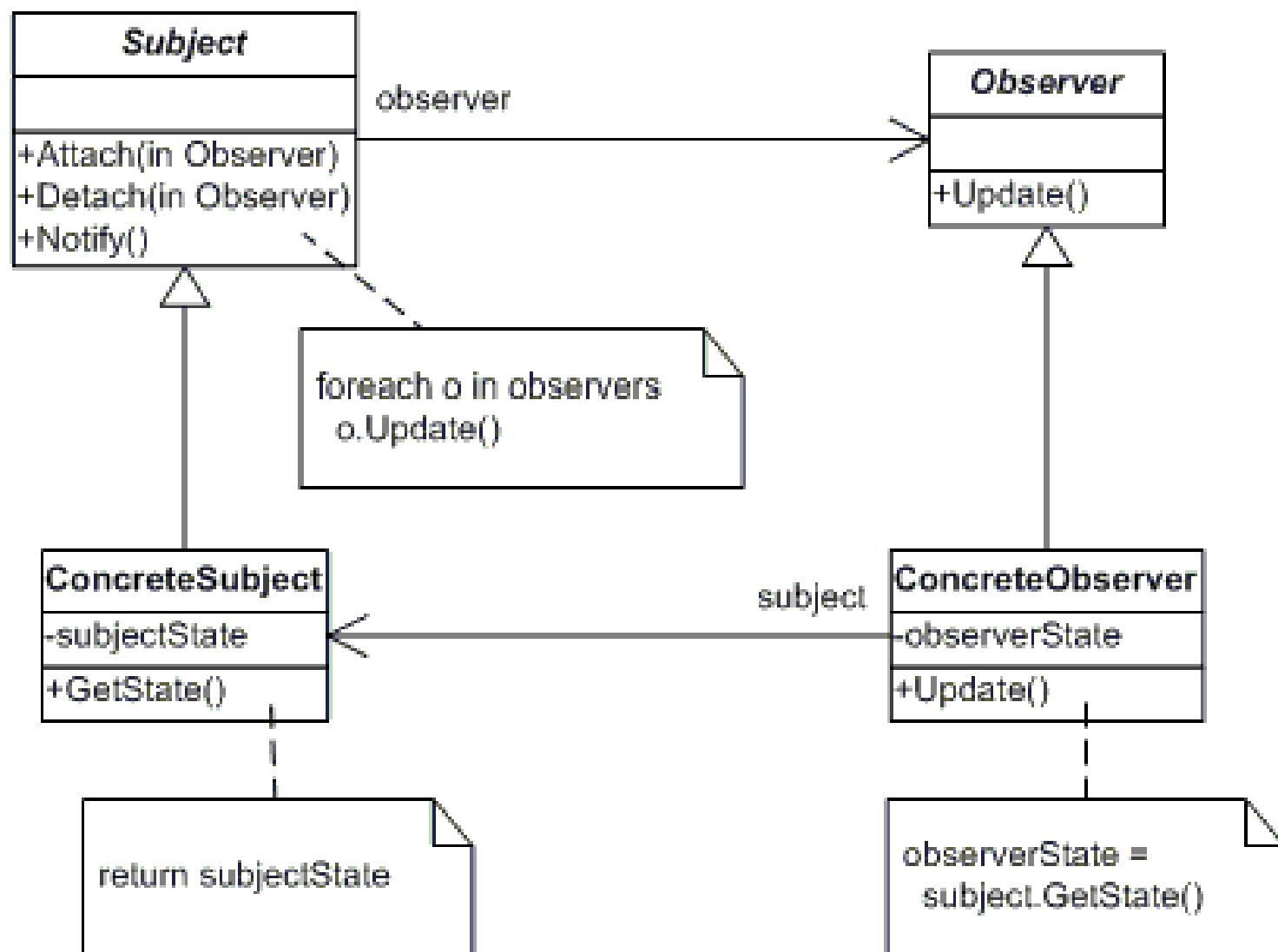
# Utilisation

- Un objet doit connaître les changements d'état d'un autre objet. L'objet doit être informé immédiatement.
- Cela peut être le cas d'un tableau affichant des statistiques. Si une nouvelle donnée est entrée, les statistiques sont recalculées. Le tableau doit être informé du changement, afin qu'il soit rafraîchi.
- L'objet devant connaître le changement (le tableau) est un observateur. Il s'enregistre en tant que tel auprès de l'objet dont l'état change. L'objet dont l'état change (les statistiques) est un "observe". Il informe ses observateurs en cas d'événement.

# Diagramme de classe



## Diagramme de classe (2)





# Caractéristiques

## Problème

- on veut assurer la cohérence entre des classes coopérant entre elles tout en maintenant leur indépendance

## Conséquences

- + couplage abstrait entre un sujet et un observateur, support pour la communication par diffusion,
- MAIS des mises à jour inattendues peuvent survenir, avec des coûts importants.

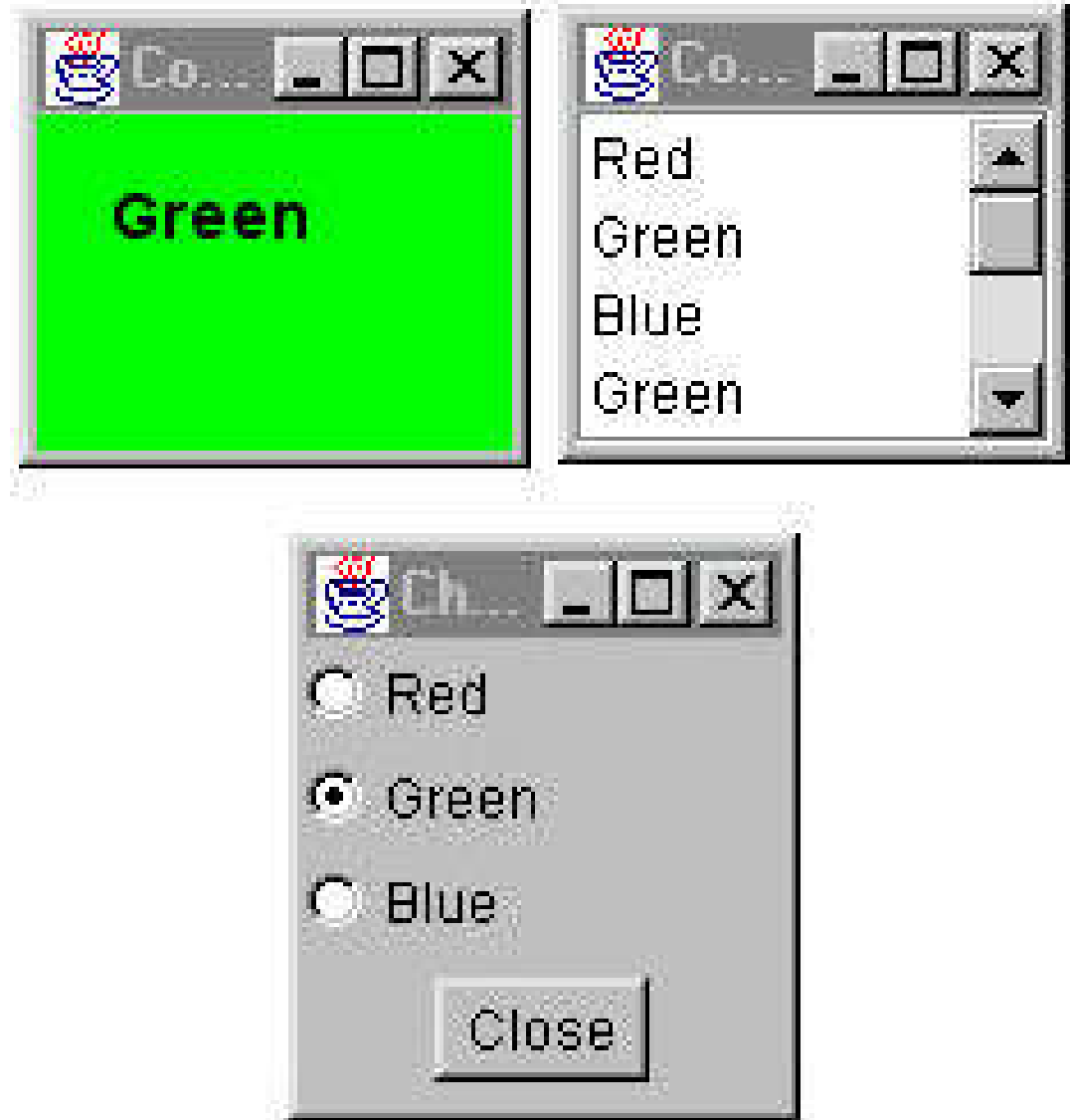
## Exemple

- `java.util.Observable`
- `java.util.Observer`

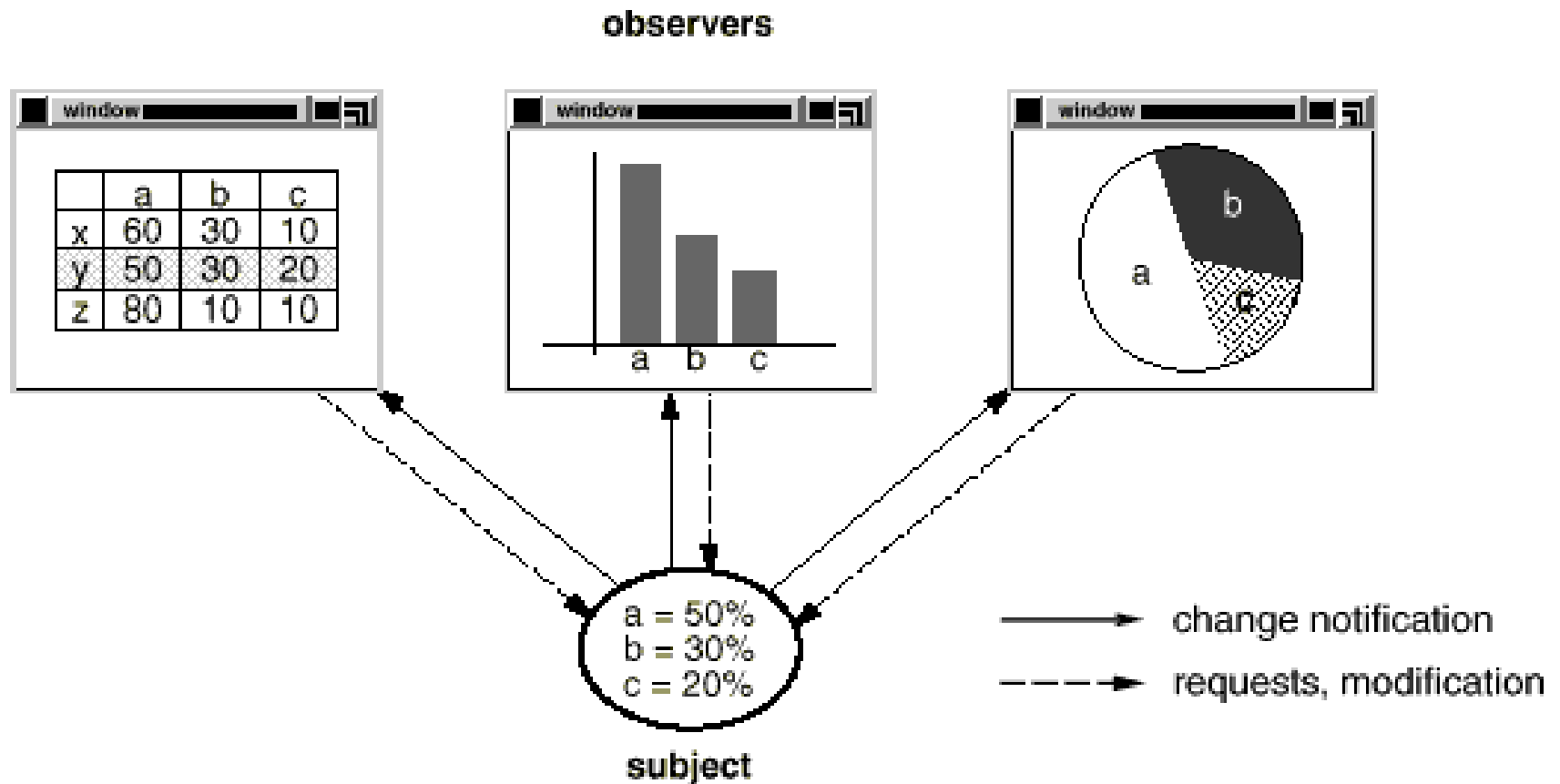
# Example 1

- **Subject (Stock)**
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects.
- **ConcreteSubject (IBM)**
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state changes
- **Observer (Investor)**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteObserver (Investor)**
  - maintains a reference to a ConcreteSubject object
  - stores state that should stay consistent with the subject's
  - implements the Observer updating interface to keep its state consistent with the subject's

## Exemple 2

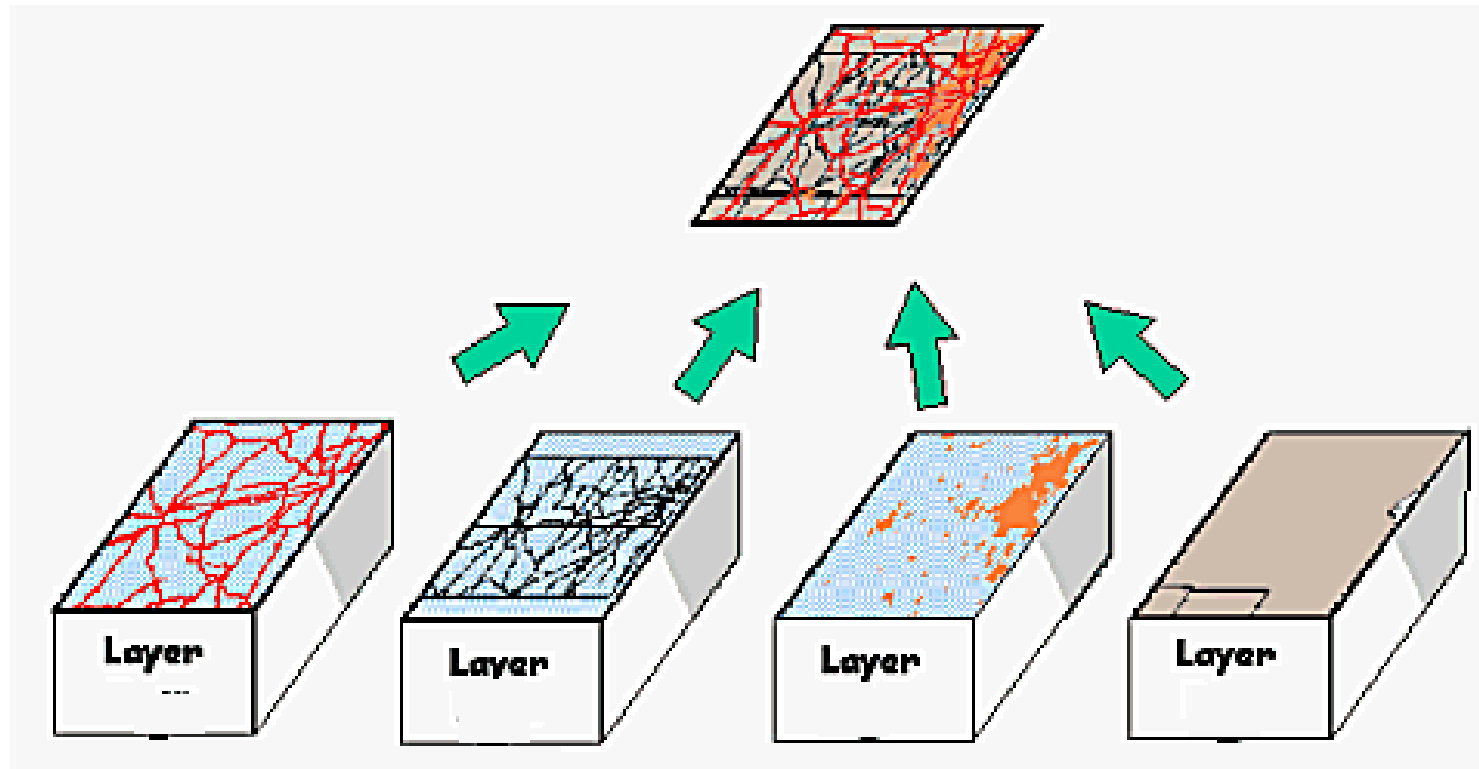


# Exemple 3



## Exemple 4

- Un observateur et plusieurs sources



# State

**Definition :** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

**Frequency of use:** medium



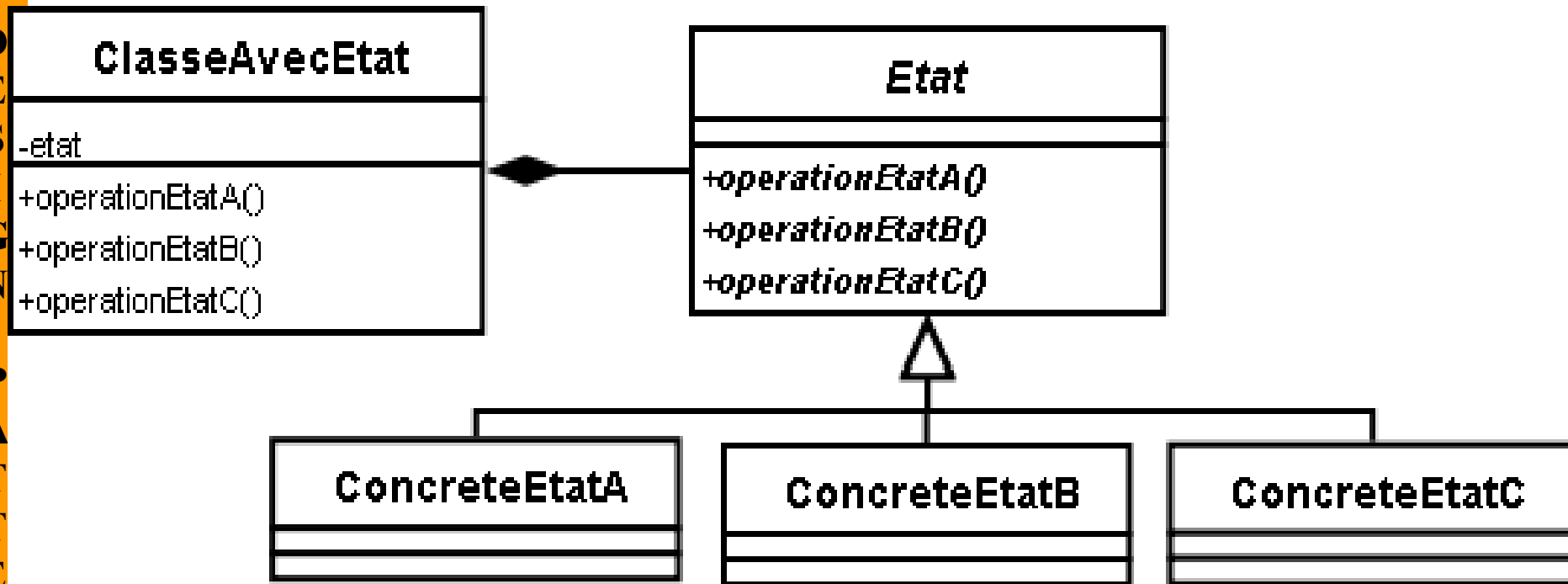
**Objectifs :** Changer le comportement d'un **objet** selon son **état interne**.

**Résultat :** Le Design Pattern permet d'isoler les **algorithmes** propres à chaque état d'un objet.

# Utilisation

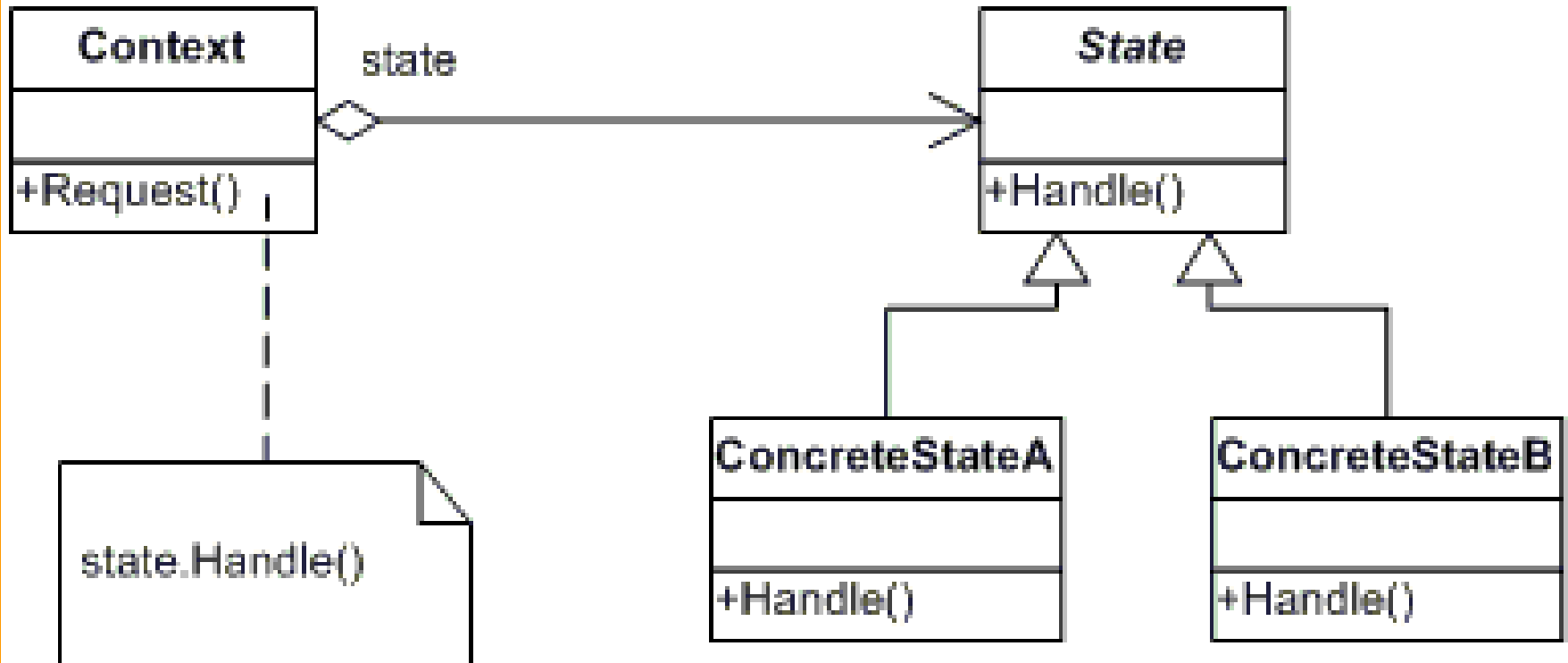
- Un objet a un fonctionnement différent selon son état interne. Son état change selon les **méthodes** appelées.
- Cela peut être un document informatique. Il a comme fonctions ouvrir, modifier, sauvegarder ou fermer. Le comportement de ces méthodes change selon l'état du document.
- Les différents états internes sont chacun représenté par une **classe** état (ouvert, modifié, sauvegardé et fermé).
- Les états possèdent des méthodes permettant de réaliser les opérations et de changer d'état (ouvrir, modifier, sauvegarder et fermer). Certains états bloquent certaines opérations (modifier dans l'état fermé). L'objet avec état (document informatique) maintient une référence vers l'état actuel. Il présente les opérations à la partie cliente.

# Diagramme de classe





# Diagramme de classe (2)



# Caractéristiques

## Problème

- ce motif est à utiliser lorsque l'on veut qu'un objet change de comportement lorsque son état interne change

## Conséquences

- + possibilité d'ajouter ou de retirer des états et des transitions de manière simple
- + suppression de traitements conditionnels
- + les transitions entre états sont rendues explicites

# Exemple

- **Context (Account)**
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State (State)**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **Concrete State (RedState, SilverState, GoldState)**
  - each subclass implements a behavior associated with a state of Context

# Strategy

**Definition :** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**Frequency of use:** medium high



## Objectifs :

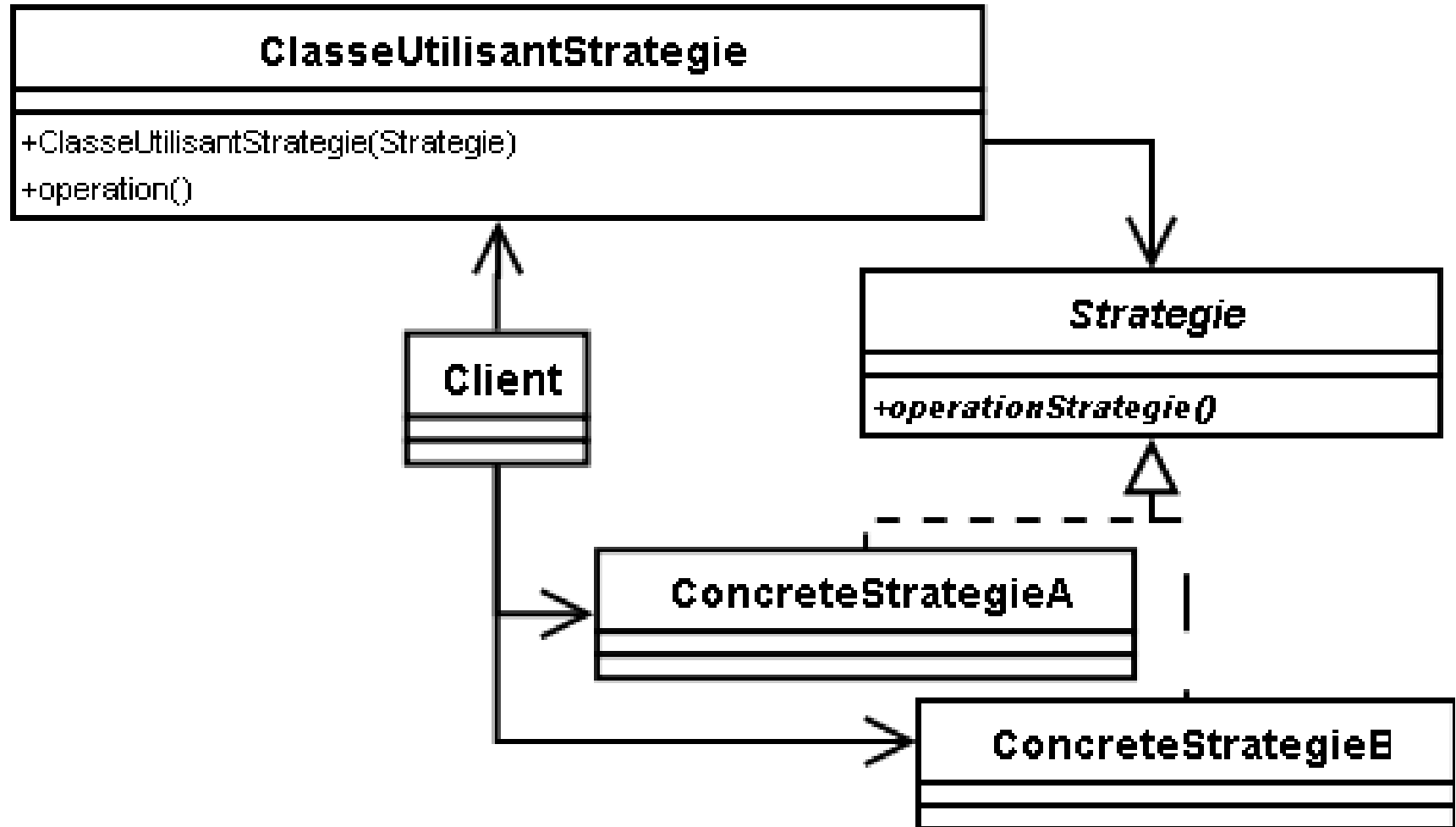
- Définir une famille d' **algorithmes** interchangeables.
- Permettre de les changer indépendamment de la partie cliente.

**Résultat:** Il permet d'isoler les algorithmes appartenant à une même famille d'algorithmes.

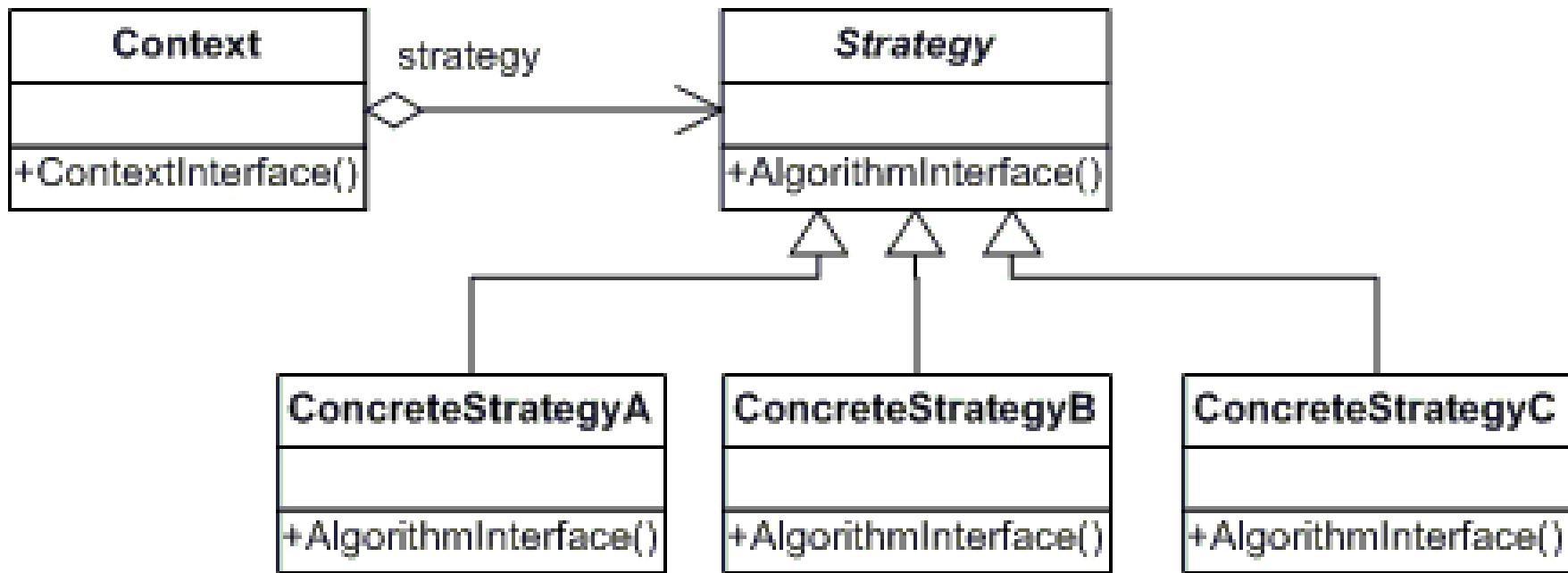
# Utilisation

- Un objet doit pouvoir faire varier une partie de son algorithme.
- Cela peut être une liste triée. A chaque insertion, la liste place le nouvel élément à l'emplacement correspondant au tri. Le tri peut être alphabétique, inverse, les majuscules avant les minuscules, les minuscules avant, etc...
- La partie de l'algorithme qui varie (le tri) est la stratégie. Toutes les stratégies présentent la même **interface**. La classe utilisant la stratégie (la liste) délègue la partie de traitement concernée à la stratégie.

# Diagramme de classe



## Diagramme de classe (2)



# Caractéristiques

## Problème

- on veut (i) définir une famille d'algorithmes, (ii) encapsuler chacun et les rendre interchangeables tout en assurant que chaque algorithme peut évoluer indépendamment des clients qui l'utilisent

## Conséquences

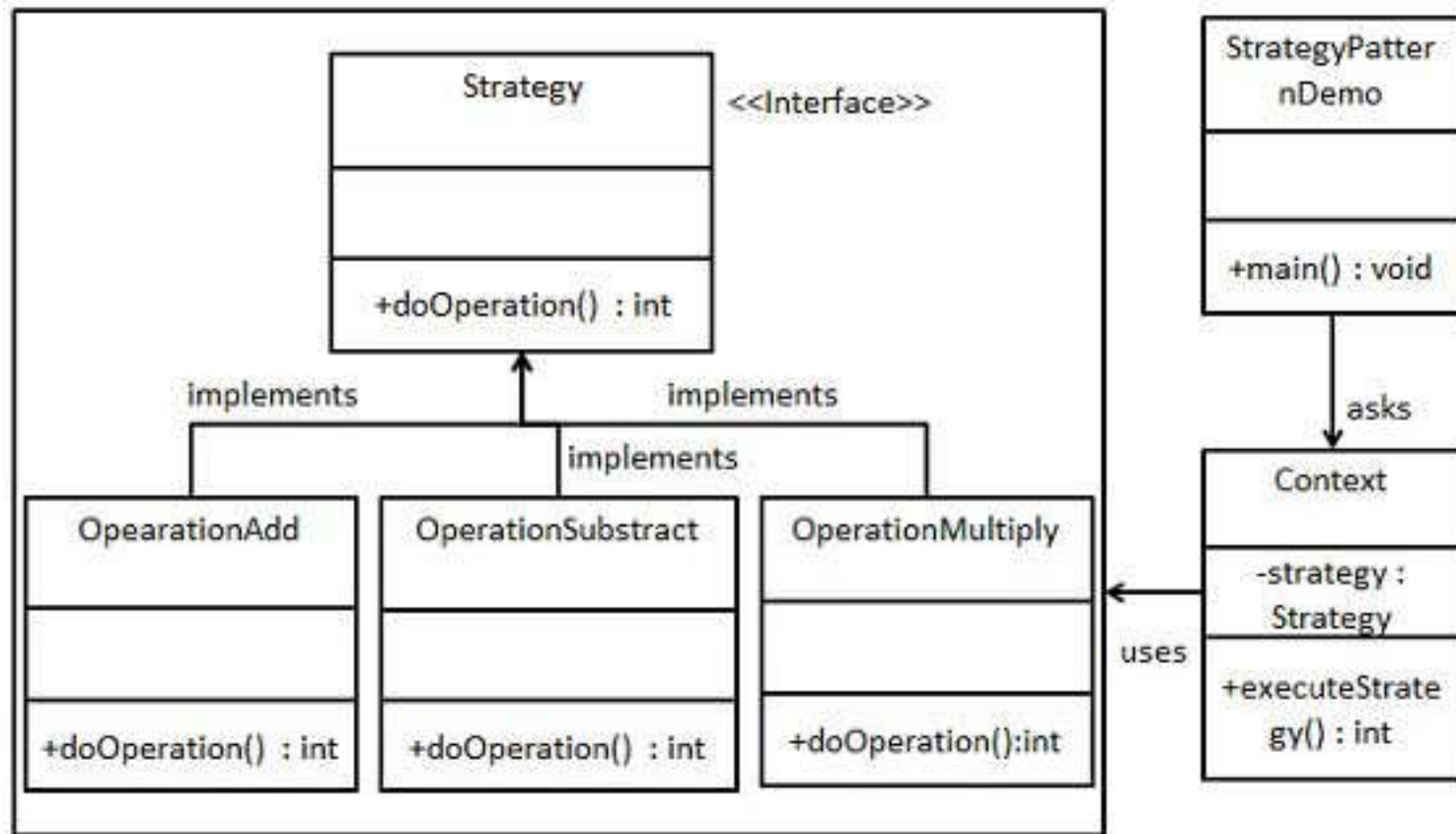
- + Expression hiérarchique de familles d'algorithmes, élimination de tests pour sélectionner le bon algorithme, laisse un choix d'implémentation et une sélection dynamique de l'algorithme
- Les clients doivent faire attention à la stratégie, surcoût lié à la communication entre Strategy et Context, augmentation du nombre d'objets



# Exemple

- **Strategy (SortStrategy)**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- **ConcreteStrategy (QuickSort, ShellSort, MergeSort)**
  - implements the algorithm using the Strategy interface
- **Context (SortedList)**
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object
  - may define an interface that lets Strategy access its data.

## Exemple 2



# Template Method

**Definition :** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

**Frequency of use:** medium



**Objectifs :** Définir le squelette d'un **algorithme** en **déléguant** certaines étapes à des **sous-classes**.

**Résultat :** Il permet d'isoler les parties variables d'un algorithme.

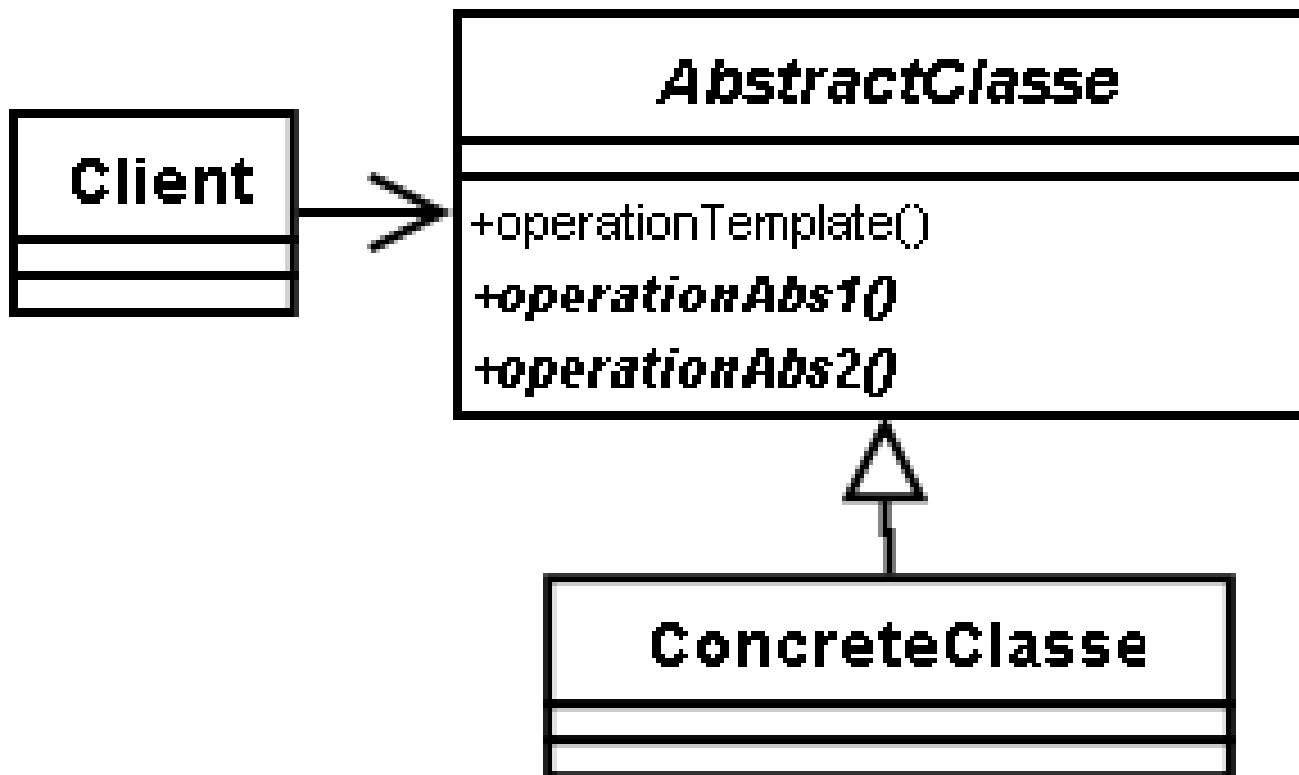
# Utilisation

- Une classe possède un fonctionnement global. Mais les détails de son algorithme doivent être spécifiques à ses sous-classes.
- Cela peut être le cas d'un document informatique. Le document a un fonctionnement global où il est sauvegardé.
- Pour la sauvegarde, il y aura toujours besoin d'ouvrir le fichier, d'écrire dedans, puis de fermer le fichier. Mais, selon le type de document, il ne sera pas sauvegardé de la même manière. S'il s'agit d'un document de traitement de texte, il sera sauvegardé en suite d'octets. S'il s'agit d'un document HTML, il sera sauvegardé dans un fichier texte.

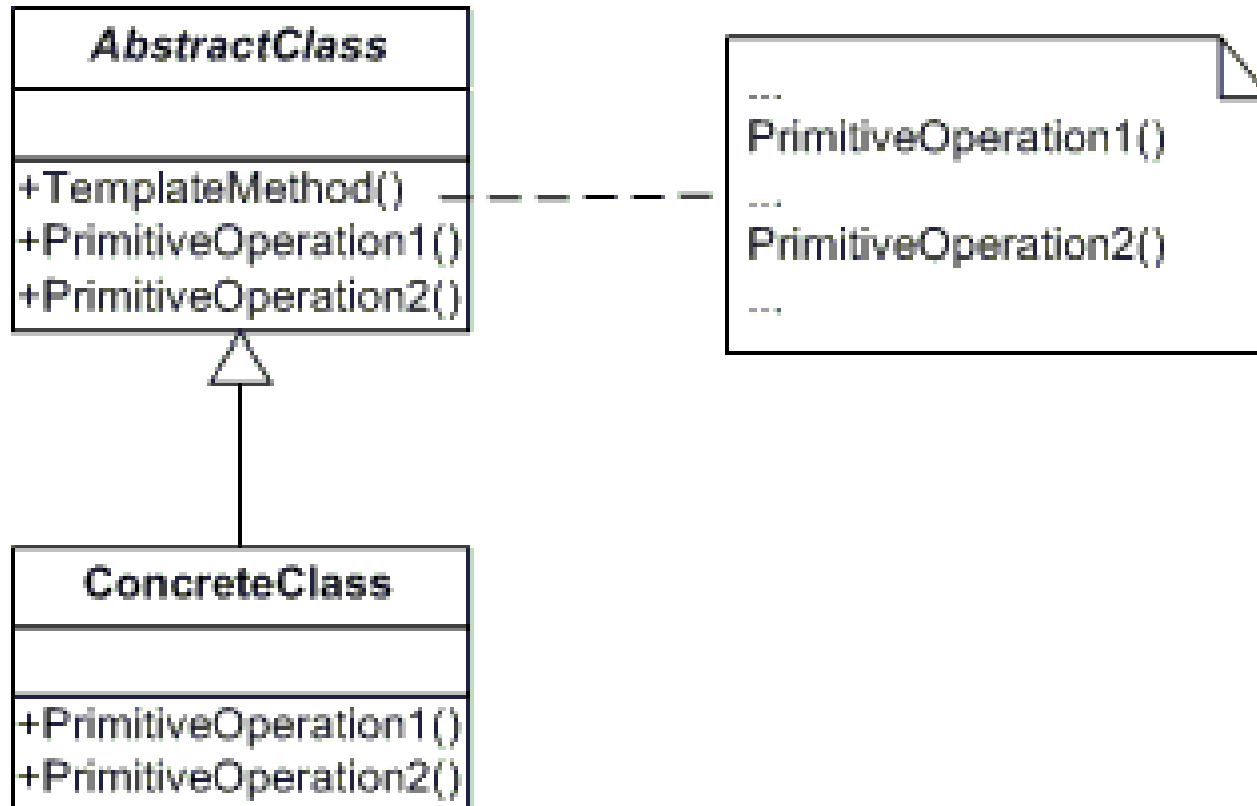
## Utilisation (suite)

- La partie générale de l'algorithme (sauvegarde) est gérée par la classe abstraite (document).
- La partie générale réalise l'ouverture, fermeture du fichier et appelle un méthode d'écriture.
- La partie spécifique de l'algorithme (écriture dans la fichier) est définie au niveau des classes concrètes (document de traitement de texte ou document HTML).

# Diagramme de classe



## Diagramme de classe (2)

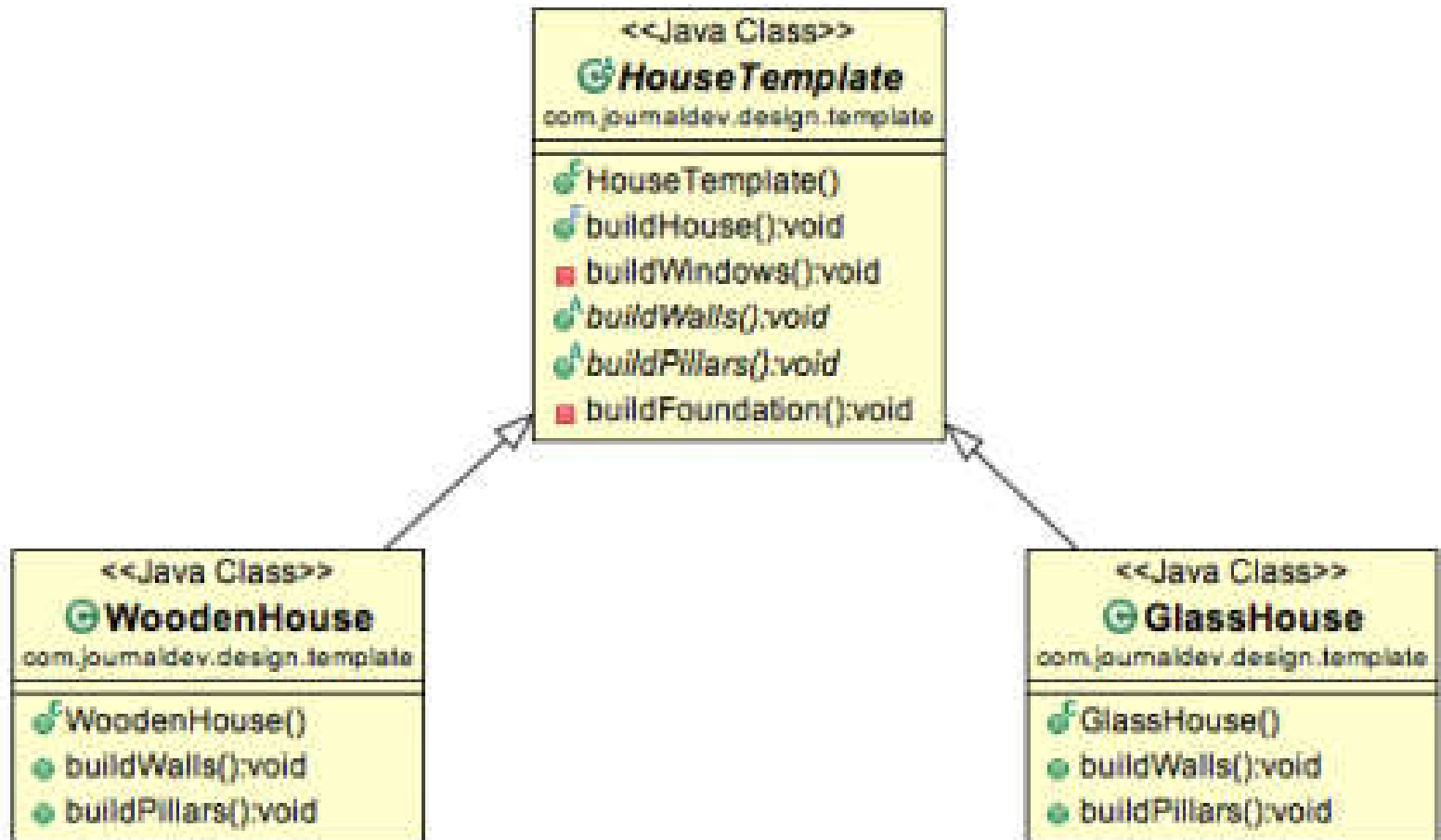


# Example

- **AbstractClass (DataObject)**
  - defines abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass (CustomerDataObject)**
  - implements the primitive operations to carry out subclass-specific steps of the algorithm



# Exemple



# Visitor

**Definition :** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

**Frequency of use:** low



**Objectifs :** Séparer un **algorithme** d'une structure de données.

**Résultat :** Il permet d'isoler les algorithmes appliquées sur des structures de données.

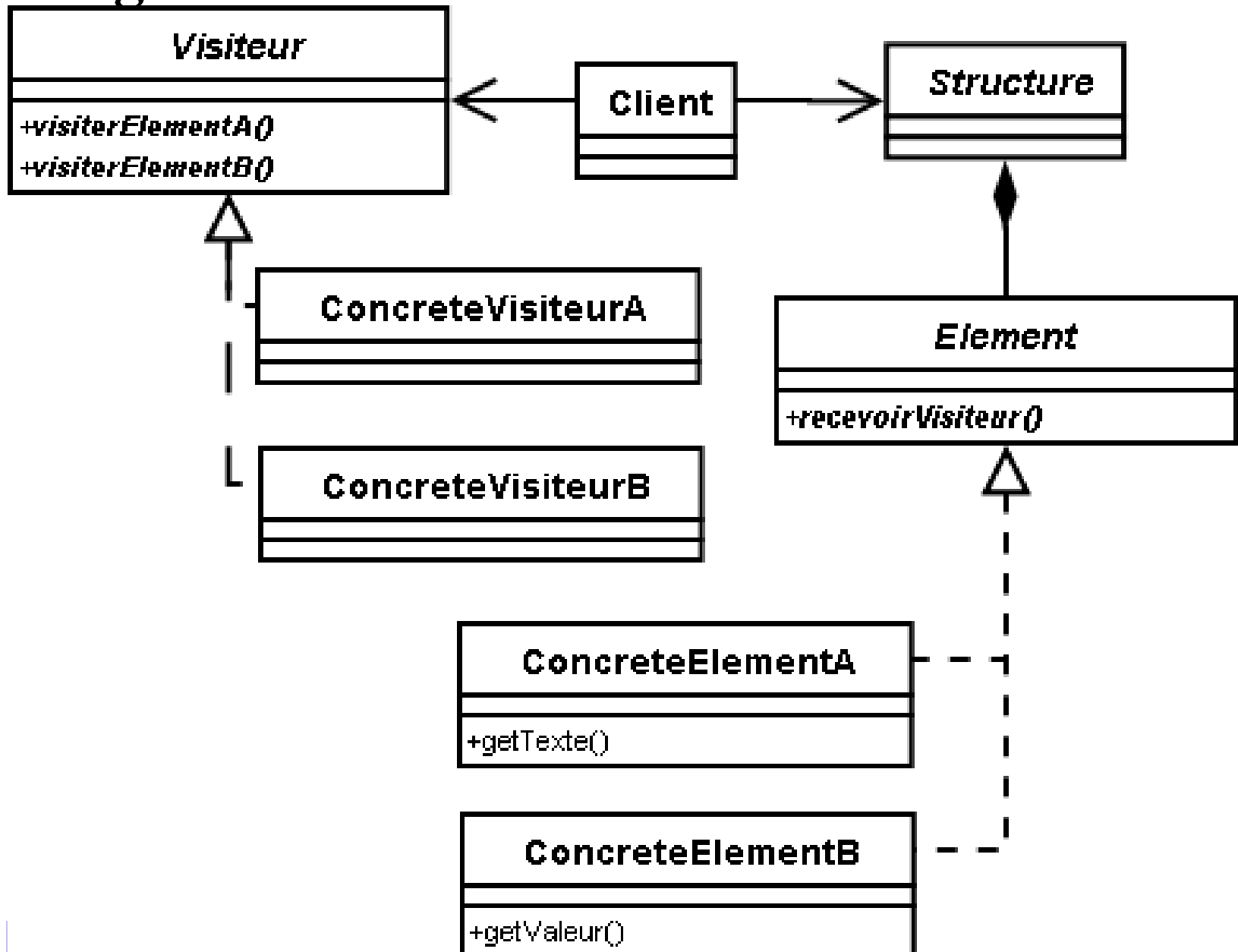
# Utilisation

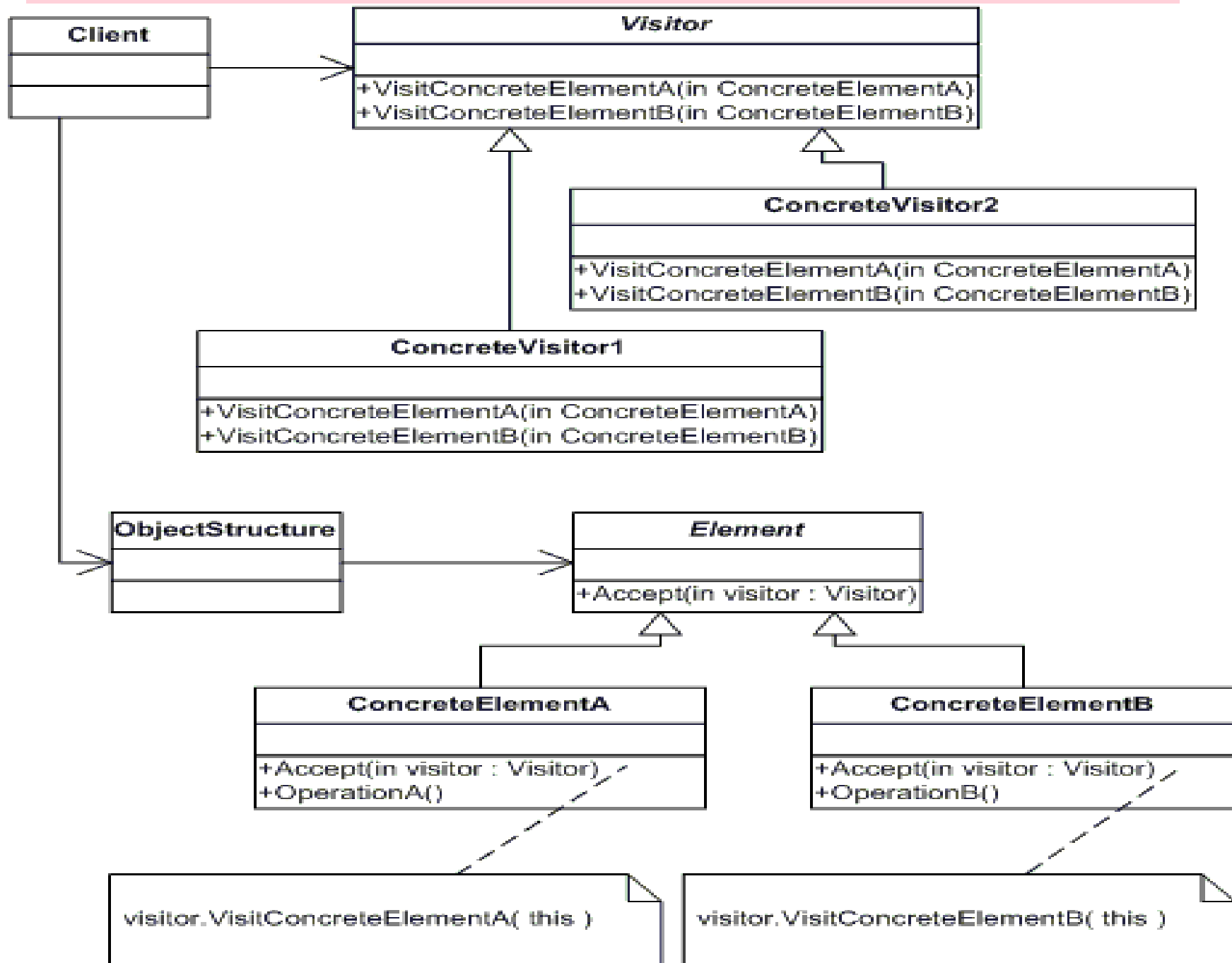
- Il est nécessaire de réaliser des **opérations** sur les éléments d'un objet structuré.
- Ces opérations varient en fonction de la nature de chaque élément et les opérations peuvent être de plusieurs types.
- Cela peut être le cas d'un logiciel d'images de synthèse.
- L'image est composée de plusieurs objets : sphère, polygone, personnages de la scène qui sont constitués de plusieurs objets, etc...
- Sur chaque élément, il faut effectuer plusieurs opérations pour le rendu : ajout des couleurs, effet d'éclairage, etc...

# Utilisation

- Chaque type d'opération (ajout des couleurs, effet d'éclairage) est implémenté par un visiteur.
- Chaque visiteur implémente une **méthode** spécifique (visiter un sphère, visiter un polygone) pour chaque type d'élément (sphère, polygone).
- Chaque élément (sphère) implémente un méthode d'acceptation de visiteur où il appelle la méthode spécifique de visite.

# Diagramme de classe





# Caractéristiques

## Problème

- Des opérations doivent être réalisées dans une structure d'objets comportant des objets avec des interfaces différentes
- Plusieurs opérations distinctes doivent être réalisées sur des objets d'une structure
- La classe définissant la structure change rarement mais de nouvelles opérations doivent pouvoir être définies souvent sur cette structure

## Conséquences

- + l'ajout de nouvelles opérations est aisé
- + union de différentes opérations et séparations d'autres
- MAIS l'ajout de nouvelles classes concrètes est freinée

# Example

- **Visitor (Visitor)**

- declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface

- **ConcreteVisitor (IncomeVisitor, VacationVisitor)**

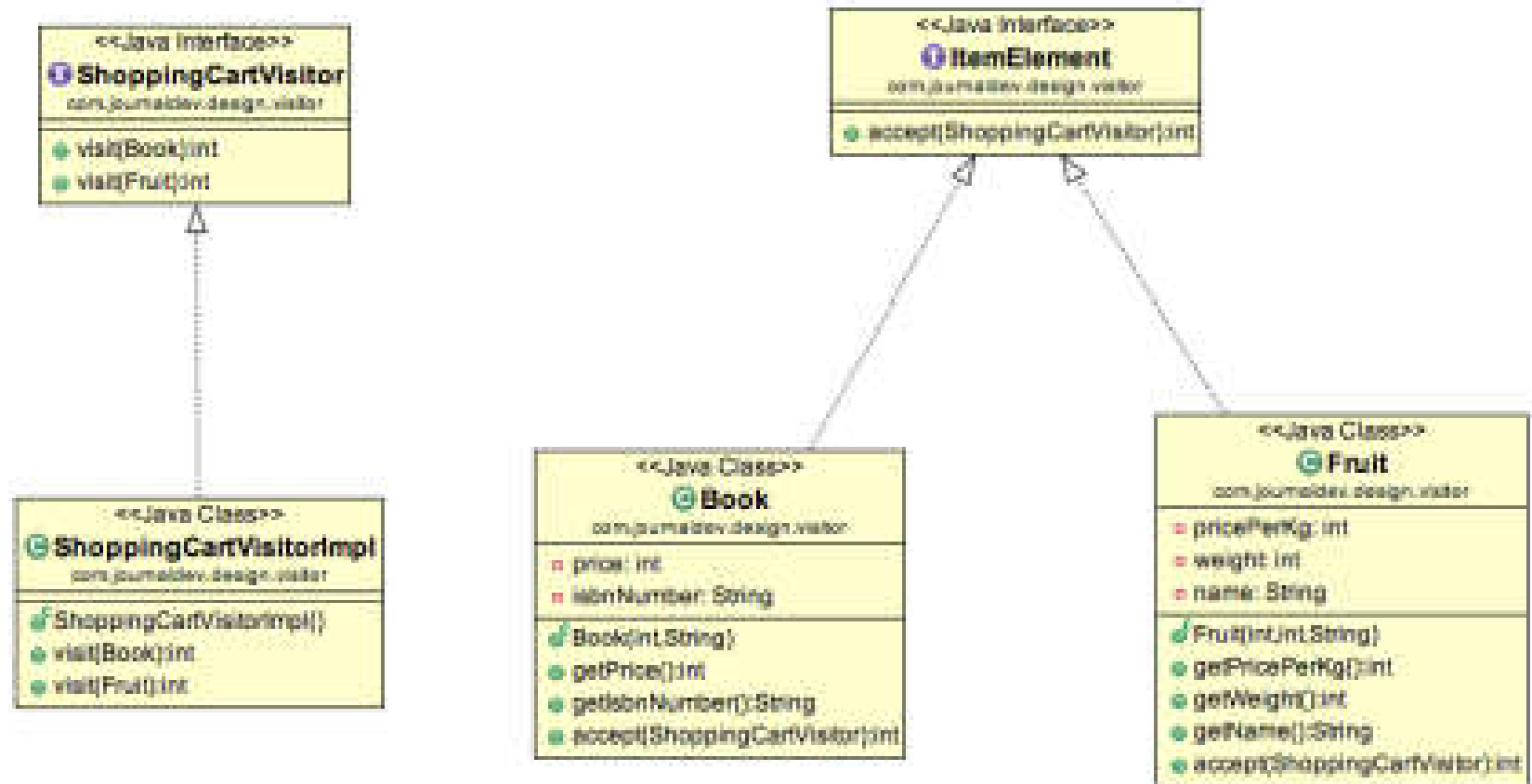
- implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.



# Exemple (suite)

- **Element (Element)**
  - defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement (Employee)**
  - implements an Accept operation that takes a visitor as an argument
- **ObjectStructure (Employees)**
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements
  - may either be a Composite (pattern) or a collection such as a list or a set

# Exemple 1



# **Partie 4**

# **Architectural Patterns**

## **MVC**

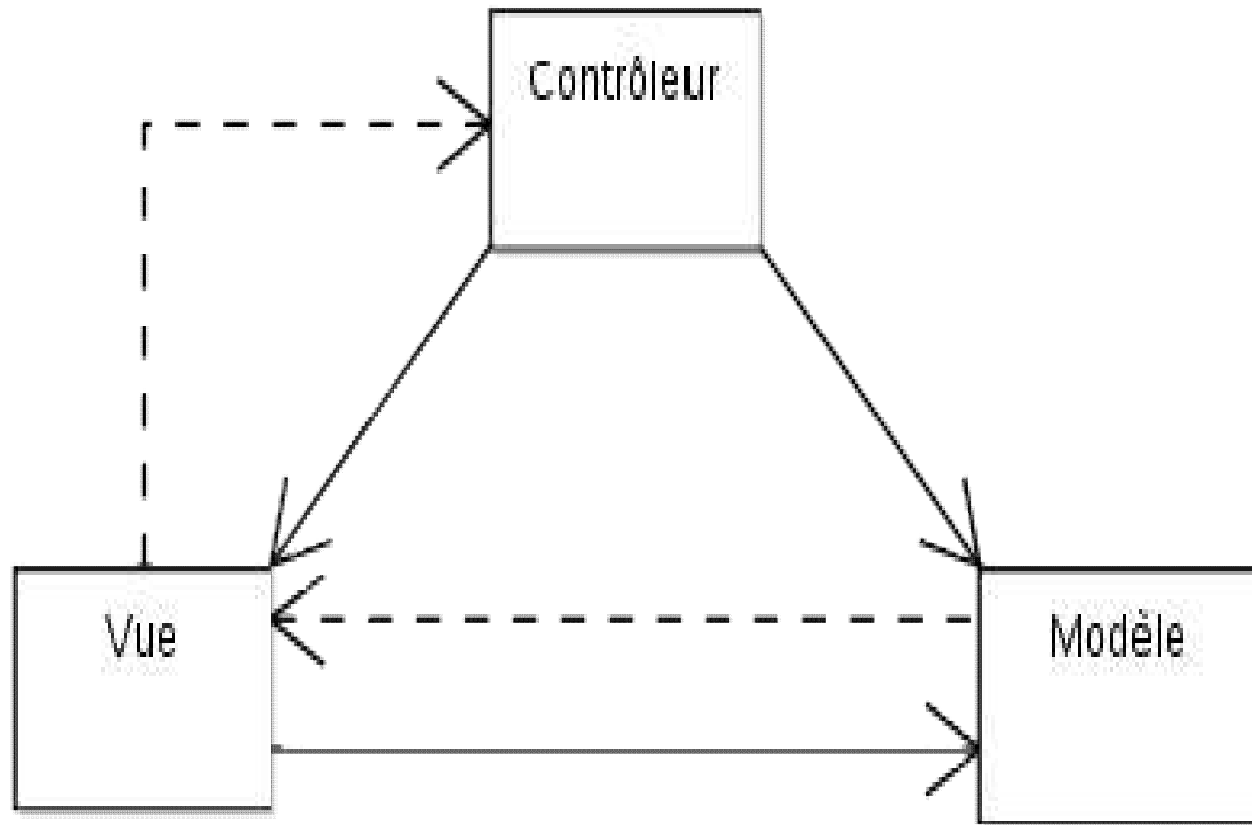
# Plan

- Architecture Modèle/Vue/Contrôleur .
  - Le modèle.
  - La vue.
  - Le contrôleur.
- Avantages et inconvénients .
- Démonstration.

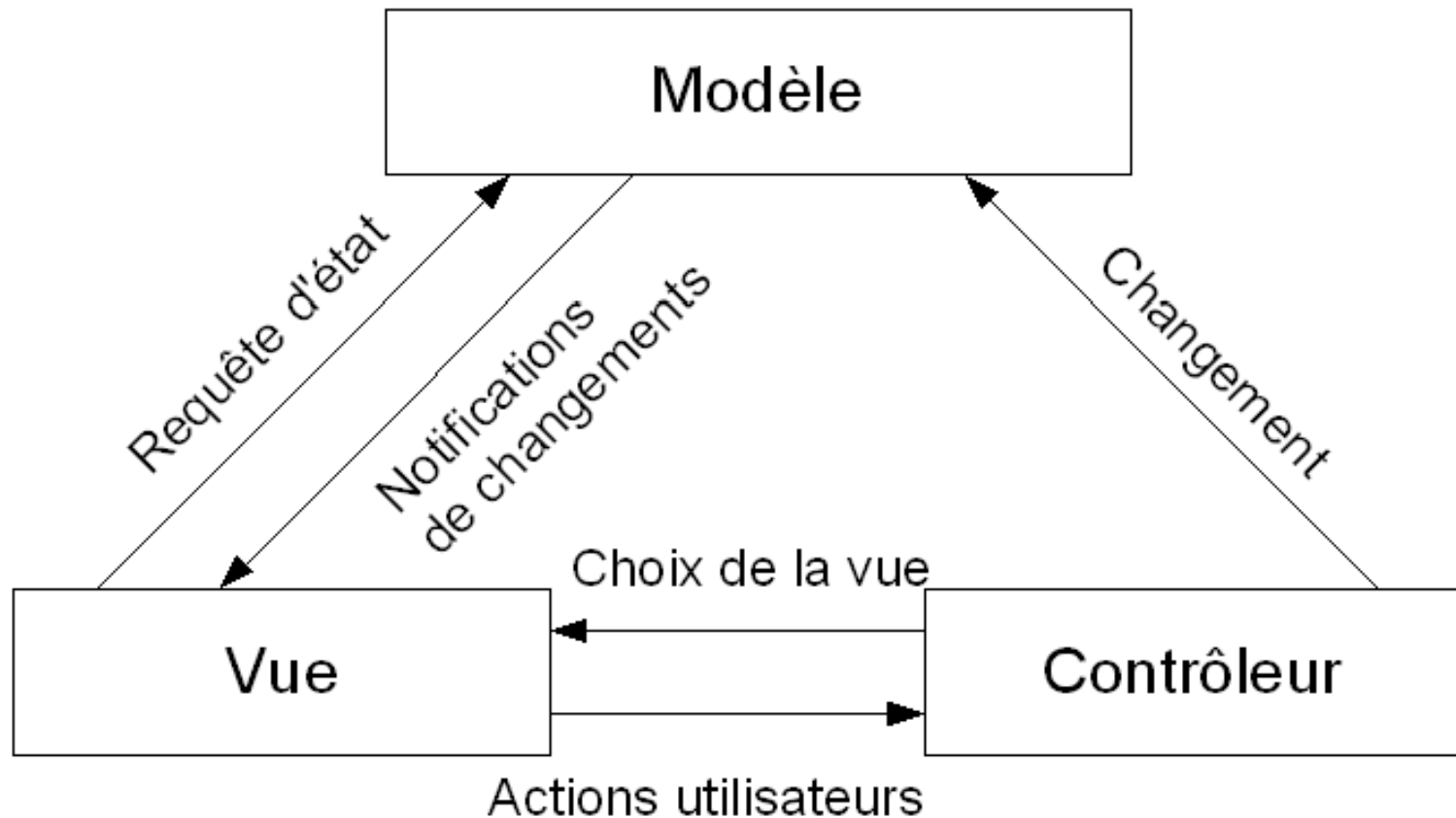
# Architecture Modèle/Vue/Contrôleur

- Ce modèle d'architecture impose la séparation entre les **données**, la **présentation** et les **traitements**, ce qui donne trois parties fondamentales dans l'application finale : le **modèle**, la **vue** et le **contrôleur**.
- Le motif MVC part du principe que toute application peut être décomposée en trois couches séparées :
  - **Modèle**, càd les données
  - **Vue**, càd la représentation des données
  - **Contrôleur**, càd le traitement sur les données en vue de leur représentation.
- Le patron MVC est une combinaison des patrons Observateur, Stratégie et Composite.

# Diagramme de classe



## Diagramme de classe (2)



# Le modèle

- Le modèle représente le comportement de l'application :
  - traitements des données,
  - interactions avec la base de données, ...
- Il assure la gestion de ces données et garantit leur intégrité.
- Le modèle offre des méthodes pour mettre à jour ces données (insertion, suppression, modification de valeur).
- Il offre aussi des méthodes pour récupérer ces données.
- Les résultats renvoyés par le modèle sont dénués de toute présentation.



# La vue

- La vue correspond à l'interface avec laquelle l'utilisateur interagit:
  - Sa première tâche est de présenter les résultats renvoyés par le modèle.
  - Sa seconde tâche est de recevoir toutes les actions de l'utilisateur (clic de souris, sélection d'une entrée, boutons...). Ces différents événements sont envoyés au contrôleur.
- La vue n'effectue aucun traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle.

# Le contrôleur

- Le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser.
- Il reçoit tous les événements de l'utilisateur et enclenche les actions à effectuer.
- Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle et ensuite avertit la vue que les données ont changé pour qu'elle se mette à jour.
- Certains événements de l'utilisateur ne concernent pas les données mais la vue. Dans ce cas, le contrôleur demande à la vue de se modifier.

# Interactions entre les couches

Ce modèle de conception permet principalement 2 choses :

- Le changement d'une couche sans altérer les autres. C'est-à-dire que comme toutes les couches sont clairement séparées, on doit pouvoir en changer une pour, par exemple, remplacer Swing par SWT sans porter atteinte aux autres couches. On pourrait aussi donc changer le modèle sans toucher à la vue et au contrôleur. Cela rend les modifications plus simples.
- La synchronisation des vues. Avec ce design pattern, toutes les vues qui montrent la même chose sont synchronisées.

# Avantages

- Le changement d'une couche sans altérer les autres. C'est-à-dire que comme toutes les couches sont clairement séparées, on doit pouvoir en changer une pour, par exemple, remplacer Swing par AWT sans porter atteinte aux autres couches.

# Inconvénients

- Il faut tout de même garder en mémoire, que la mise en œuvre de MVC dans une application n'est pas des plus simples.
- Ce pattern n'est donc à conseiller que pour les moyennes et grandes applications.

# Démonstration

```
public JFrameListVolume (VolumeController controller) {
```



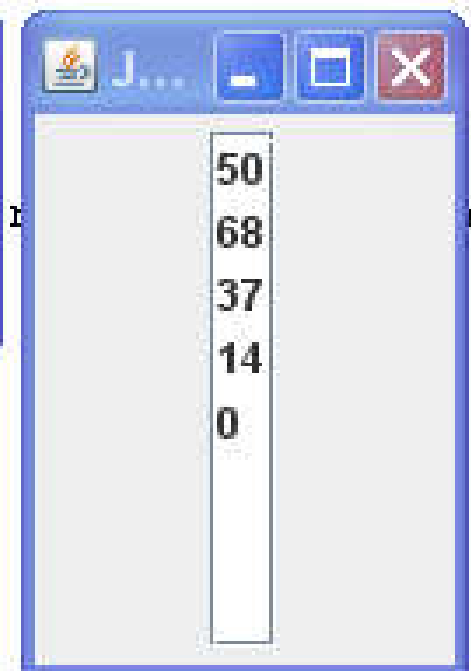
```
pub.
```



```
pri
```

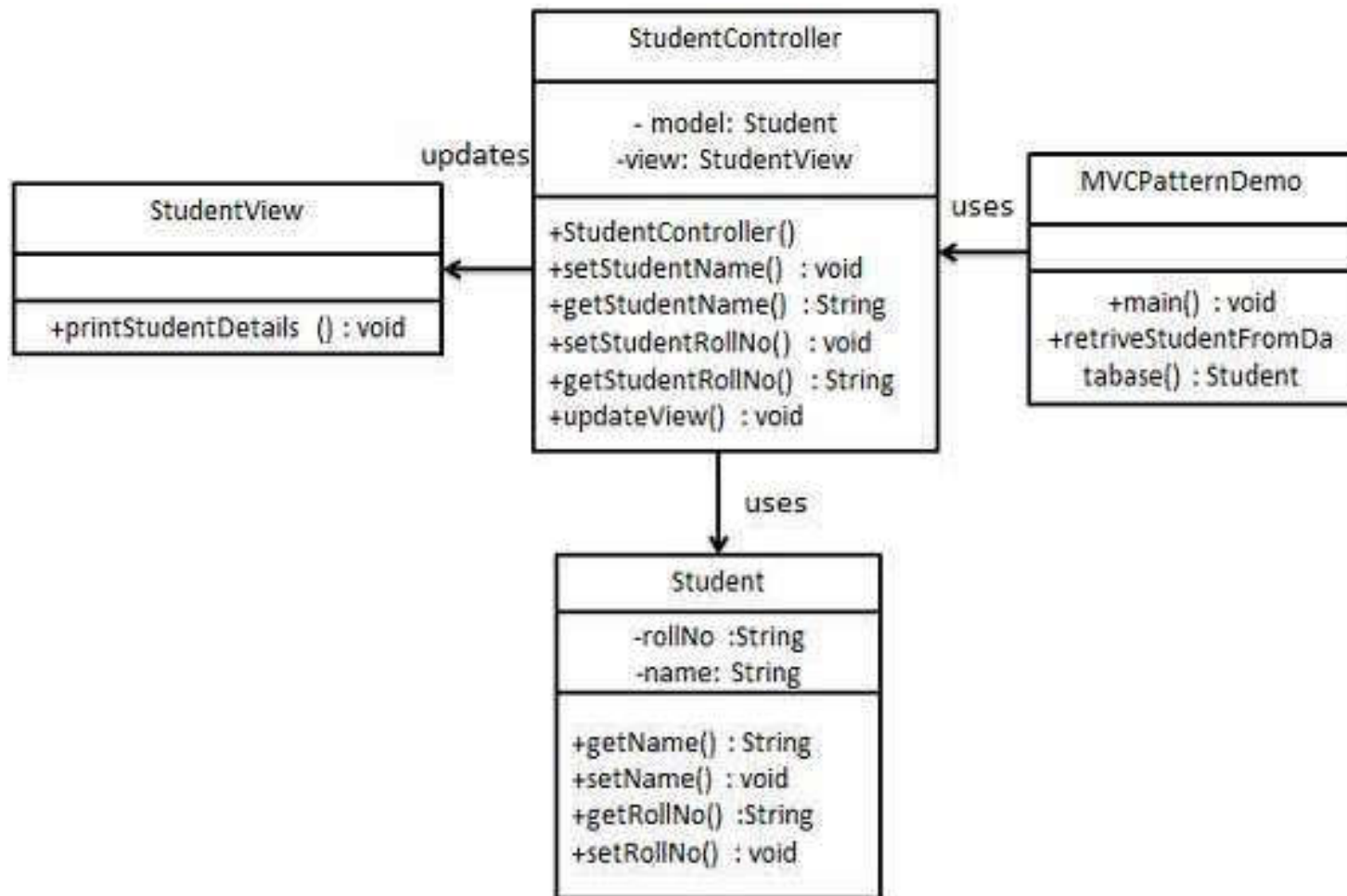


```
contentPane = new JPanel();
```



```
ne)
```

## Exemple 2



# Références



# Patterns de création (1)

- **AbstractFactory** : on passe un paramètre à la création qui définit ce qu'on va créer
- **Builder** : on passe en paramètre un objet qui sait construire l'objet à partir d'une description
- **FactoryMethod** : la classe sollicitée appelle des méthodes abstraites ...il suffit de sous-classer
- **Prototype** : des prototypes variés existent qui sont copiés et assemblés
- **Singleton** : unique instance

# Patterns de structure

- Description de la manière dont doivent être connectés des objets de l'application afin de rendre ces connections indépendantes des évolutions futures de l'application
- Découplage de l'interface et de l'implémentation de classes et d'objets

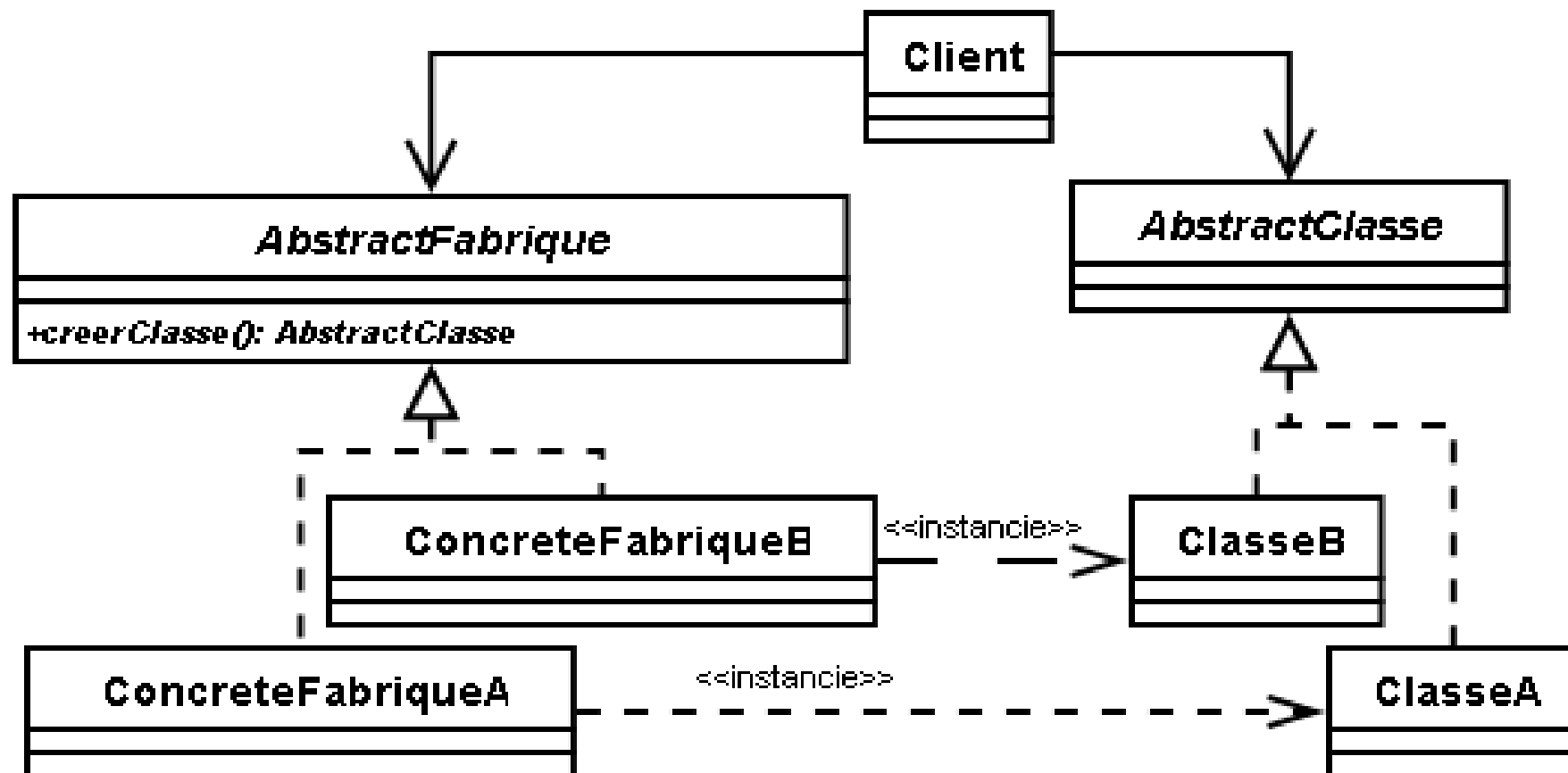
# Patterns de comportement

- Description de comportements d'interaction entre objets
- Gestion des interactions dynamiques entre des classes et des objets

# Participants

- **AbstractClasse** : définit l'interface de l'objet instancié.
- **ClasseA et ClasseB** : sont des sous-classes concrètes d'AbstractClasse. Elles sont instanciées par les classes Fabrique.
- **Fabrique** : déclare une méthode de création (**creerClasse**). C'est cette méthode qui a la responsabilité de l'instanciation d'un objet **AbstractClasse**. Si d'autres méthodes de la classe ont besoin d'une instance de **AbstractClasse**, elles font appel à la méthode de création. Dans l'exemple, la méthode **operation()** utilise une instance de **AbstractClasse** et fait donc appel à la méthode **creerClasse**. La méthode de création peut être paramétrée ou non. Dans l'exemple, elle est paramétrée, mais le paramètre n'est significative que pour la **FabriqueA**.
- **FabriqueA et FabriqueB** : substituent la méthode de création. Elles implémentent une version différente de la méthode de création.
- La partie cliente utilise une sous-classe de **Fabrique**.

# Diagramme UML



# Classes / Objets participants

- **AbstractFabrique** : définit l'interface des méthodes de création. Dans le diagramme, il n'y a qu'une méthode de création pour un objet d'une classe. Mais, le diagramme sous-entend un nombre indéfini de méthodes pour un nombre indéfini de classes.
- **ConcreteFabriqueA et ConcreteFabriqueB** : implémentent l'interface et instancient la classe concrète appropriée.
- **AbstractClasse** : définit l'interface d'un type d'objet instancié.
- **ClasseA et ClasseB** : sont des sous-classes concrètes d'**AbstractClasse**. Elles sont instanciées par les ConcreteFabrique.
- La partie cliente fait appel à une Fabrique pour obtenir une nouvelle instance d'**AbstractClasse**. L'instanciation est transparente pour la partie cliente. Elle manipule une **AbstractClasse**.