Design Patterns

Mourad MAHRANE

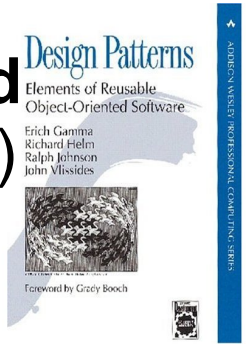
Objectifs

- Comprendre les bases de la philosophie des « formes de conception »
- Connaître le vocabulaire spécifique
- Connaître quelques « patterns »
- Concevoir l'objet différemment

Bibliographie

Les Design Patterns en Java
Les 23 modèles de conception fondamentaux,
Steven John Metsker, William C. Wake

Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series)
Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides (1995)



J2EE Design Patterns
William Crawford, Jonathan Kaplan

Plan

- Introduction
- Rappel de la Programmation Orientée Objets (POO) / Modélisation UML
- Classes de patterns et implémentation
- JEE Patterns

Introduction

Contexte

Modélisation d'objets dans une application

Besoins :

- Encapsuler des données sans en empêcher l'accès
- Trouver le bon niveau de granularité des objets
- Limiter les dépendances entre objets
- Concevoir des objets polyvalents, flexibles, réutilisables
- Simplicité d'utilisation
- Implémentation performante

Design pattern (patron de conception)

Schéma à objets qui forme une solution à un problème connu et fréquent

- Bonnes pratiques de la POO
  (retour d'expérience de programmeurs)
- Pas d'aspect théorique dans les patterns
  (ce n'est pas des algorithmes, ni des règles)
- Ne définit pas une méthode (ne guide pas une prise de décision ; un pattern est la décision prise)

Design pattern (2)

Exemple : comment garantir l'instanciation unique d'une classe ? :

- Mettre le constructeur en « private »
- Créer un instance statique dans la classe et gérer sa création dans une méthode de classe

  => Design pattern : Singleton

Historique

1995 : GoF « Gang of Four »

Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

Auteurs du livre de référence :

Design Patterns : Elements of Reusable Object-Oriented Software

- 23 patterns répertoriés dans 3 classes

- chaque pattern est décrit : nom, classification, intention/exemple de code, structure, conséquences ...

Avantages

- Un vocabulaire commun
- Capitalisation de l'expérience
- Un niveau d'abstraction plus élevé qui permet d'élaborer des constructions logicielles de meilleure qualité
- Réduction de 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...

Description d'un pattern

Chaque pattern est décrit à l'aide d'un langage de programmation et d'un langage de modélisation :

- Nom et description du pattern (problème/solution)
- Exemple d'implémentation - conséquences
- Structure générique du pattern :

  son schéma UML en dehors de tout contexte

  liste des participants au pattern

  les collaborations au sein du pattern

Rappel POO

/

Modélisation UML

Définition

L'orienté-objet = approche de résolution algorithmique de problèmes permettant de produire des programmes modulaires de qualité..

Objectifs :

- développer une partie d'un programme sans qu'il soit nécessaire de connaître les détails internes aux autres Parties;

- Apporter des modifications locales à un module, sans que cela affecte le reste du programme;

- Réutiliser des fragments de code développés dans un cadre différent.

Qu'est ce qu'un objet ?

Objet = élément identifiable du monde réel, soit concret (voiture, stylo,...), soit abstrait (entreprise, temps,...)

Un objet est caractérisé par :

- son état (les données de l'objet)
- son comportement (opérations : ce qu'il sait faire)

Qu'est ce qu'une classe ?

Une classe est une structure ayant des attributs (champs) et des méthodes permettant de **définir un nouveau type d'objets**

On peut construire plusieurs **instances** d'une classe.

```
public class NomClasse {
    // Attributs
    // Constructeurs
    // Méthodes
}
```

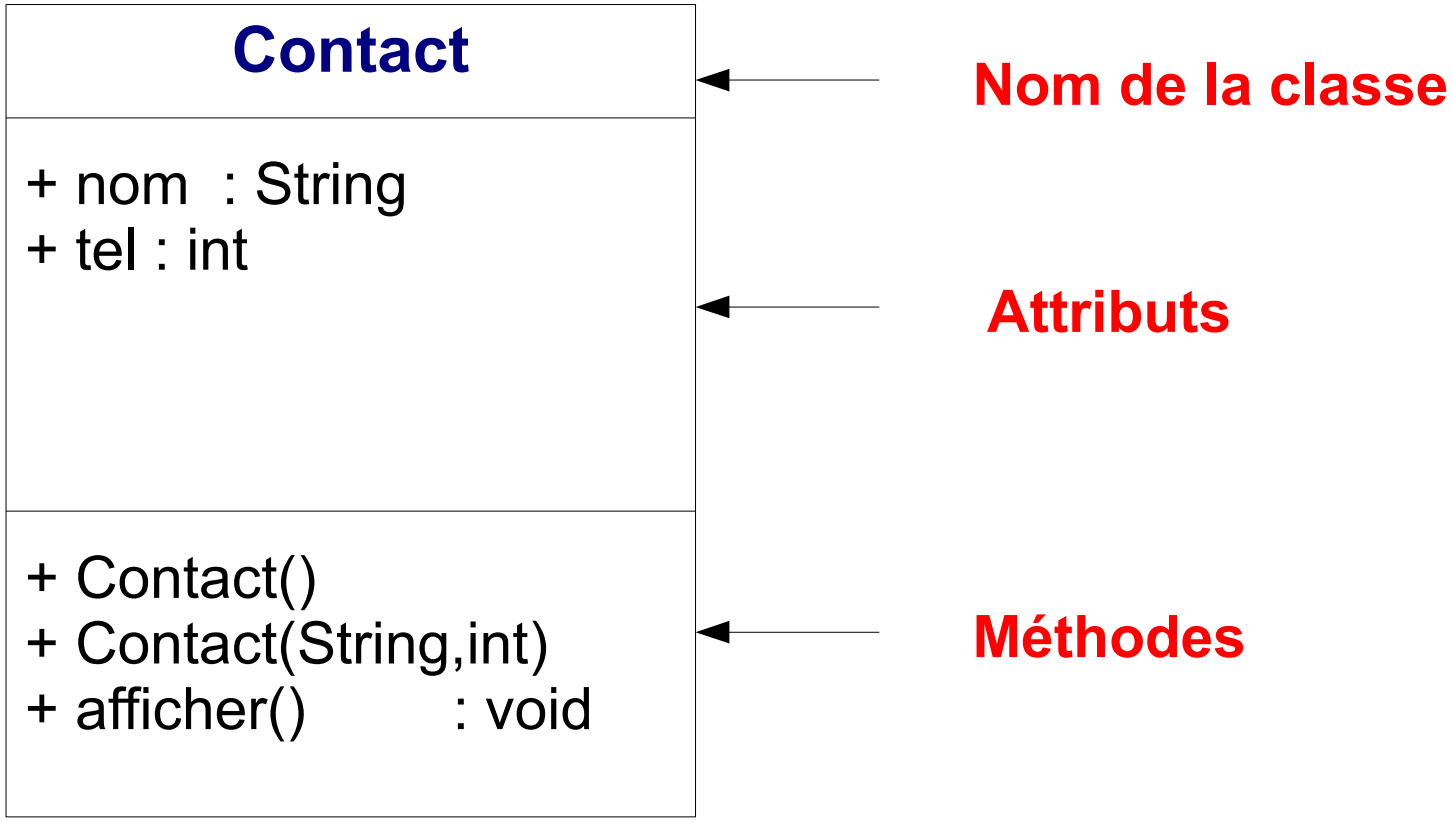
Unified Modeling Language

UML = Language pour la modélisation des classes, des objets, des intéractions etc...

UML 2.0 comporte ainsi treize types de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers du système d'information :

- **Diagrammes fonctionnels**
- **Diagrammes structuraux (statiques)**
- **Diagrammes comportementaux (dynamiques)**

Représentation UML d'une Classe



Les attributs ou les méthodes peuvent être précédés par un opérateur (+, #, -) pour indiquer le niveau de visibilité

Package (ou espace de noms)

Package = groupement de classes qui traitent un même problème pour former des « bibliothèques de classes ».

Une classe appartient à un package s'il existe une ligne au début renseignant cette option :

```
package nompackage;
```

Pour utilises une classe (au choix) :

- Être dans le même package
- Préfixer par le package (à chaque utilisation)
- Au début du fichier, importer la classe, ou le package entier :

```
import nompackage.*;
```

Instanciation d'objets

La création d'un objet/instance nécessite un appel au constructeur de la classe :

```
NomClasse nomObjet = new Contructeur(param.);
```

Constructeur = méthode spéciale dans la classe appelée à la création d'instances permettant d'initialiser les attributs

si on ne définit pas de constructeur, le compilateur en créera un par défaut sans paramètres qui initialisera les nombres à 0, les booléens à false et les objets à null.

- Un constructeur porte le nom de la classe
- Plusieurs constructeurs peuvent être déclarés.
- Il peut aussi y avoir un destructeur, appelé aussi automatiquement (`public void finalize(){...}`)

Variables d'instance / de classes

- **Variable d'instance** :
  - définit l'état de l'objet. La valeur d'un attribut est propre à chaque instance.
  - appel : nomObjet.nomVar

- **Variable de classe** :
  - variable déclarée « static »
  - partagée par toutes les instances (pas besoin d'instancier la classe)
  - appel : NomClasse.nomVar

Méthodes d'instance / de classe

- **Méthode d'instance** :
  - définit le comportement de l'objet.
  - appel : nomObjet.nomMéthode(arguments);

- **Méthode de classe** :
  - méthode déclarée « static »
  - définit un comportement globale
  - ne peut manipuler que des variables de classes
  - appel : NomClasse.nomVar

Mot-clé « this »

- Fait référence à l'objet en cours
- On peut l'utiliser pour :

  - Manipuler l'objet en cours

```
maMethode(this);
```

  - Faire référence à une variable d'instance

```
this.maVariable;
```

  - Faire appel au constructeur propre de la classe

```
this(param1,param2);
```

Agrégation et Accessibilité

- **Agrégation** = associer un objet avec un autre
  ex : Objet Propriétaire à l'intérieur de la classe Voiture

- **Accessiblité :** utilisation de facteurs de visibilité
  public:
    - Accessible par toutes les classes

  protected:
    - Accessible par toutes les sous-classes et les classes du même package

  "nothing" ou **default**:
    - Accessible seulement par les classes du même package.

  private:
    - Accessible seulement dans la classe elle-même

Encapsulation

Regroupement de code et de données

- Masquage d'information par l'utilisation d'accesseurs (getters et les setters) afin d'ajouter du contrôle .

- L'encapsulation permet de restreindre les accès aux membres d'un objet, obligeant ainsi l'utilisation des membres exposés.

Qu'est-ce qu'un JavaBean ?

Un type de classe ayant :

- un constructeur public sans arguments

(et éventuellement d'autres constructeurs)

- des getters et setters pour chaque attribut

- un moyen de sérialisation

(généralement, implémente java.io.Serializable)

Héritage

- L'héritage permet de créer la structure d'une classe à partir des membres d'une autre classe.
- Utilisation du mot-clé extends
- La sous-classe hérite de tous les attributs et méthodes (public ou protected) de sa classe mère.
- Pas d'héritage multiple en Java

Redéfinition

La **redéfinition** (***overriding***) consiste à réimplémenter une version spécialisée d'une méthode héritée d'une classe mère (les signatures des méthodes dans la classe mère et la classe fille doivent être identiques).

Utilisation de « final »

Le mot clé **final** permet :

- d'interdire l'héritage à partir de la classe.
- d'interdire la redéfinition d'une méthode
- de déclarer une constante

```
public final class C1 { …
}
public final void M1() { …
}
public static final int X = 3;
```

Polymorphisme

Le polymorphisme est la propriété d'une entité de pouvoir se présenter sous diverses formes. Ce mécanisme permet de faire collaborer des objets entre eux sans que ces derniers aient déclarés leur type exact.

Exemples :

- On peut avoir une voiture prioritaire avec le type Voiture

- On peut créer un tableau de Voitures et placer à l'intérieur des objets de type Voiture et d'autres de type VoiturePrioritaire

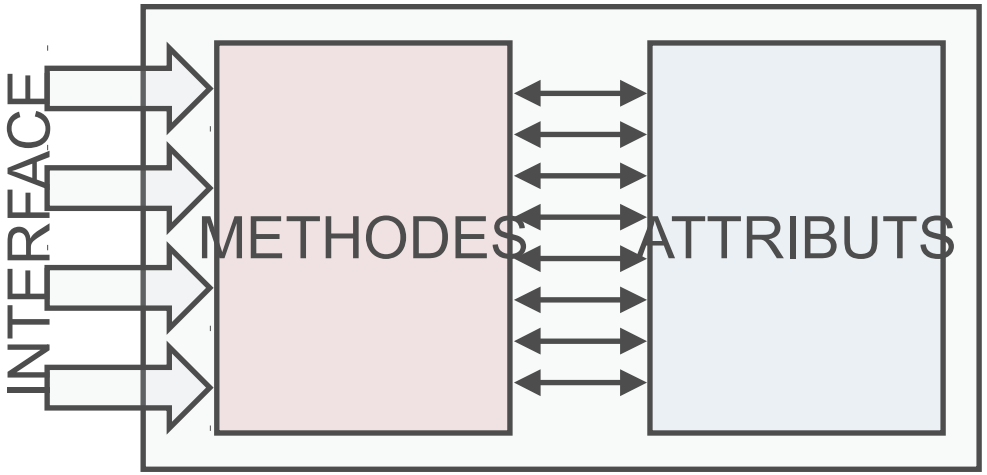
Classe abstraite

- Une classe qui ne peut être instanciée.
- Définit un type de squelette pour les sous-classes
- Si elle contient des méthodes abstraites, les sous-classes doivent implémenter le corps des méthodes abstraites.
- Déclarée avec le mot clé **abstract**.

Interface

Une pseudo classe abstraite marquée par le mot clé **interface** contenant juste des signatures de méthodes dans le but de montrer les capacités

Une classe peut implémenter une ou plusieurs interfaces (**implements**). Elle sera dans l'obligation de redéfinir les méthodes de ses dernières



Classes / Interfaces usuelles

- Serializable
- Cloneable
- Comparable
- Object
- ...

Classification des patterns

Présentation

Pour rappel, un pattern est un schéma de conception réutilisable (Organisation et hiérarchie de plusieurs modèles de classes réutilisable par simple implémentation, adaptation, extension)

Il existe 3 classes de patterns :

- **Patterns de création**
  Création d'objets sans instanciation directe d'une classe
- **Patterns de composition (de structure)**
  Composition de groupes d'objets
- **Patterns de comportement**
  Modélisation des communications inter-objets et du flot de données

Catalogue des patterns

		Classification		
		Créateurs	Structuraux	Comportementaux
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Catalogue de patterns (2)

- **Abstract Factory** : création d'objets regroupés en famille sans devoir connaître les classes concrètes destinées à la création de ces objets.
- **Builder** : séparation de la construction d'objets complexes de leur implantation de sorte qu'un client puisse créer ces objets avec des implantations différentes.
- **Factory Method** : introduction d'une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective
- **Prototype** : création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage.

Catalogue de patterns (3)

- **Singleton** : garantir l'instanciation unique d'une classe.
- **Adapter** : conversion de l'interface d'une classe existante pour garantir la collaboration entre clients de cette interface.
- **Bridge** : séparation des aspects conceptuels d'une hiérarchie de classes de leur implantation.
- **Composite** : offrir une profondeur variable à la composition d'objets ; conception basée sur un arbre
- **Decorator** : ajout dynamique de fonctionnalités supplémentaires à un objet
- **Facade** : regroupement des interfaces d'un ensemble d'objets pour le simplifié (interface unifiée)

Catalogue de patterns (4)

- **Flyweight** : faciliter le partage d'un ensemble important d'objets.
- **Proxy** : construction d'un objet se substituant à un autre objet et qui contrôle son accès.
- **Chain of responsability** : création d'une chaîne d'objets permettant de répondre à une requête.
- **Command** : transformation d'une requête en un objet pour faciliter les opérations d'annulation, suivi, mise en file...
- **Interpreter** : fournir un cadre objets pour évaluer/interpréter les expressions d'un langage.
- **Iterator** : parcours (accès séquentiel) d'une collection d'objets.
- **Mediator** : construction d'un objet pour la gestion et le contrôle des interactions d'un ensemble d'objets.

Catalogue de patterns (5)

- **Memento** : sauvegarder et restaurer l'état d'un objet
- **Observer** : construire une dépendance entre un sujet et des observateurs (avec notifications)
- **State** : adaptation du comportement d'un objet en fonction de son état interne.
- **Strategy** : adaptation du comportement et des algorithmes d'un objet en fonction d'un besoin sans changer les interactions avec le client.
- **Template Method** : report dans les sous-classes d'une étape de création d'un objet.
- **Visitor** : construction d'une opération à réaliser sur les éléments d'une collection d'objets ; ajout d'opérations sans modification de la classe de ses objets.

Comment choisir un pattern ?

- Se référer à la définition pour déterminer s'il existe un pattern dont la description s'approche de celle du problème
- Étudier en détail le pattern découvert (à travers un exemple et la structure générique) pour pouvoir l'adapter pour la résolution du problème
- Adaptation :

  - utilisation de la structure générique, renommage des classes et méthodes

  - changement de la structure générique en fonction des contraintes de l'application

Patterns de création

(Génération d'instances)

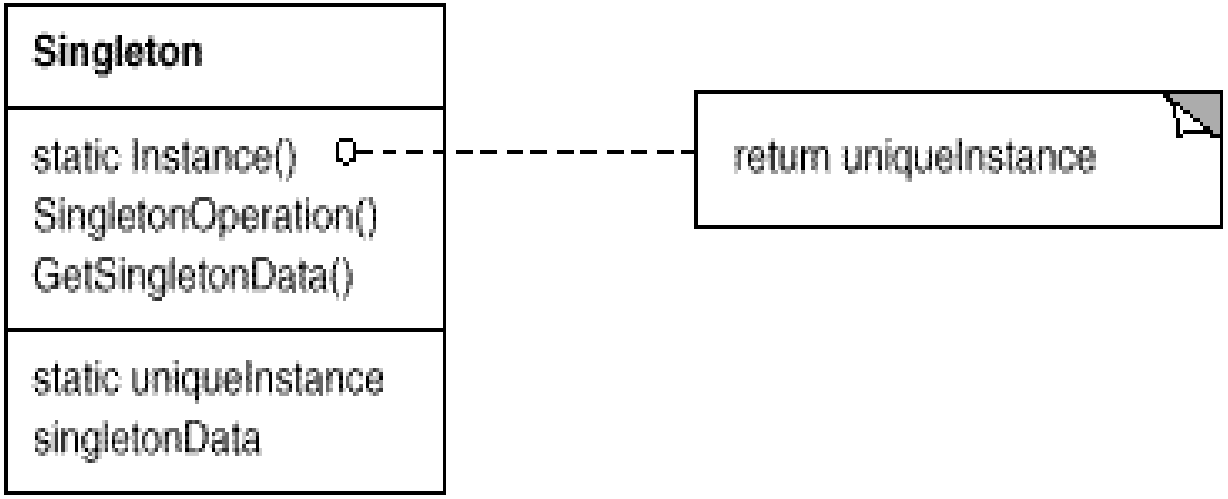
Abstraire le processus d'instanciation

- Rendre indépendant de la façon dont les objets sont créés, composés, assemblés, représentés.
- Encapsuler la connaissance de la classe concrète qui instancie.
- Cacher ce qui est créé, qui crée, comment et quand.

Singleton

Objectif : A tout moment une (et une seule) instance d'une classe existe pour une application

Solution : La classe possède une méthode statique getInstance(), qui renvoie une instance statique, et a son constructeur « protégé » (ou « privé »)



Singleton

```
public class Singleton {
     private static Singleton instance = null;

     /**
      * La présence d'un constructeur privé supprime
      * le constructeur public par défaut.
      */
     private Singleton() {}

     /**
      * Le mot-clé synchronized sur la méthode de création
      * empêche toute instanciation multiple même par
      * différents threads.
      * Retourne l'instance du singleton.
      */
     public synchronized static Singleton getInstance() {
         if (instance == null)
             instance = new Singleton();
         return instance;
     }
 }
```

Singleton

```
public class Singleton {
	/**
	 * Création de l'instance au niveau de la variable.
	 */
	private static final Singleton INSTANCE = new Singleton();

	/** La présence d'un constructeur privé supprime
	 * le constructeur public par défaut.
	 */
	private Singleton() {}

	/** Dans ce cas présent, le mot-clé synchronized n'est pas
 * utile. L'unique instanciation du singleton se fait           *
avant l'appel de la méthode getInstance(). Donc aucun         *
risque d'accès concurrents.
	 * Retourne l'instance du singleton.
	 */
	public static Singleton getInstance() {
		return INSTANCE;
	}
}
```

Factory Method

Objectif : Polymorphisme – créer une instance d'une classe ou d'une autre suivant des conditions

(Définir une interface de création, mais laisser les sous classes décider du type).

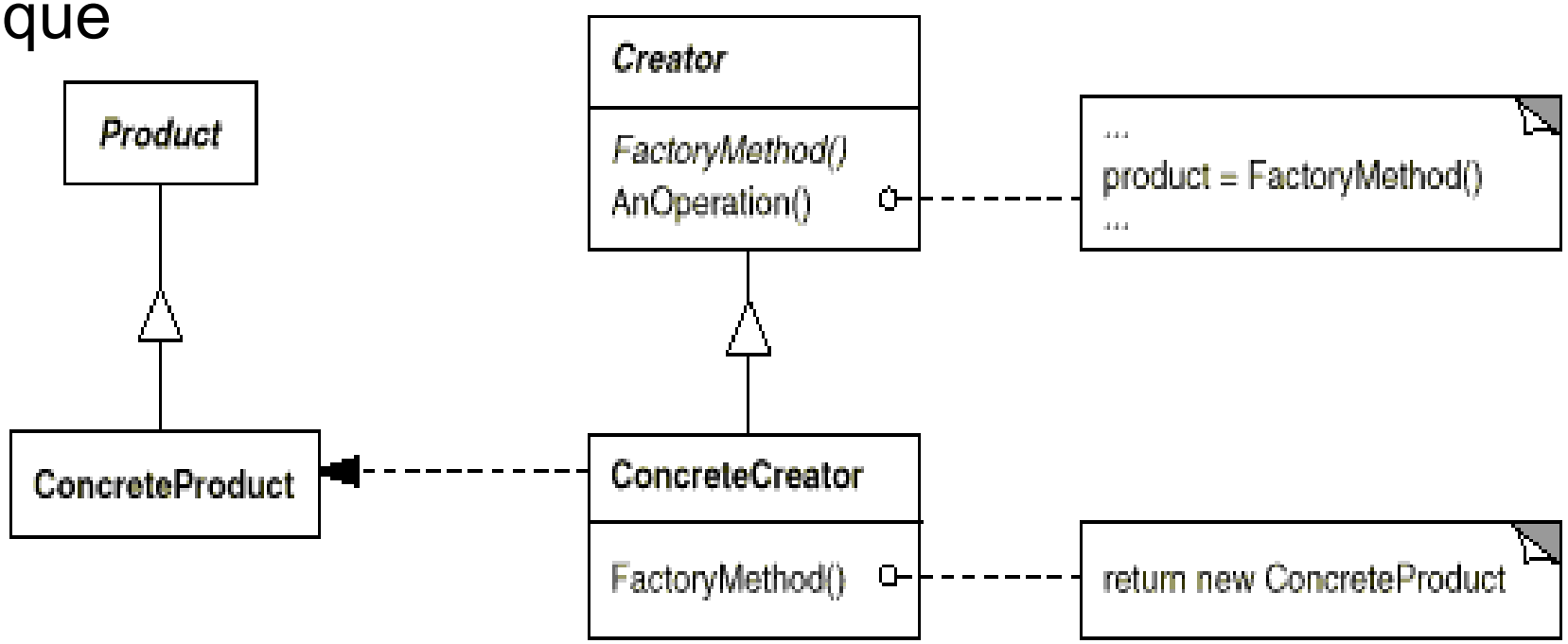
Solution : Classe supplémentaire (l'usine) qui possède une méthode qui renvoie un nouvel objet en fonction d'un paramètre.

Extension : Abstract Factory – Choisir l'usine elle-même suivant les conditions.

Factory Method

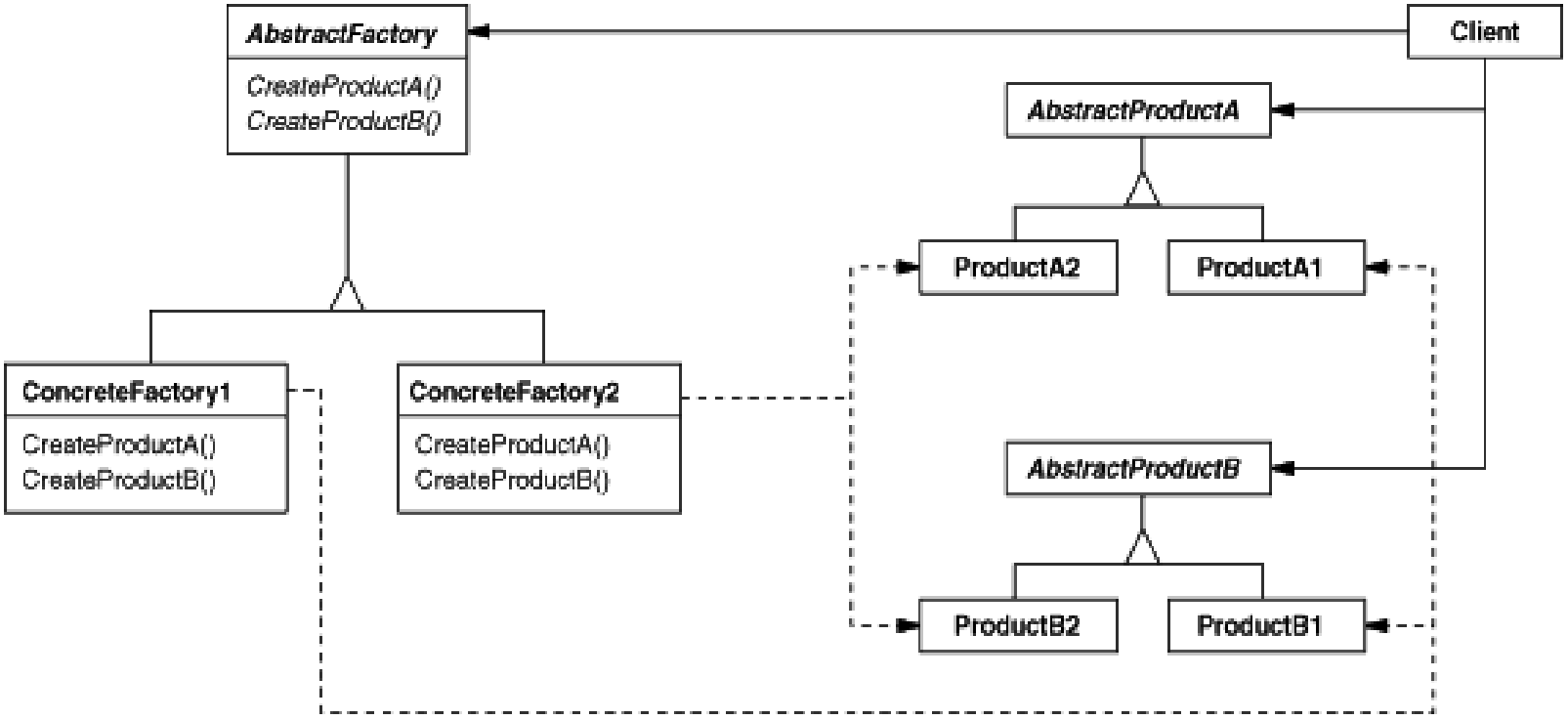
On utilise le FactoryMethod lorsque :

- une classe ne peut anticiper la classe de l'objet qu'elle doit construire
- une classe délègue la responsabilité de la création à ses sous-classes, tout en concentrant l'interface dans une classe unique



Abstract Factory

Objectif : obtenir des instances de classes implémentant des interfaces connues, mais en ignorant le type réel de la classe obtenue



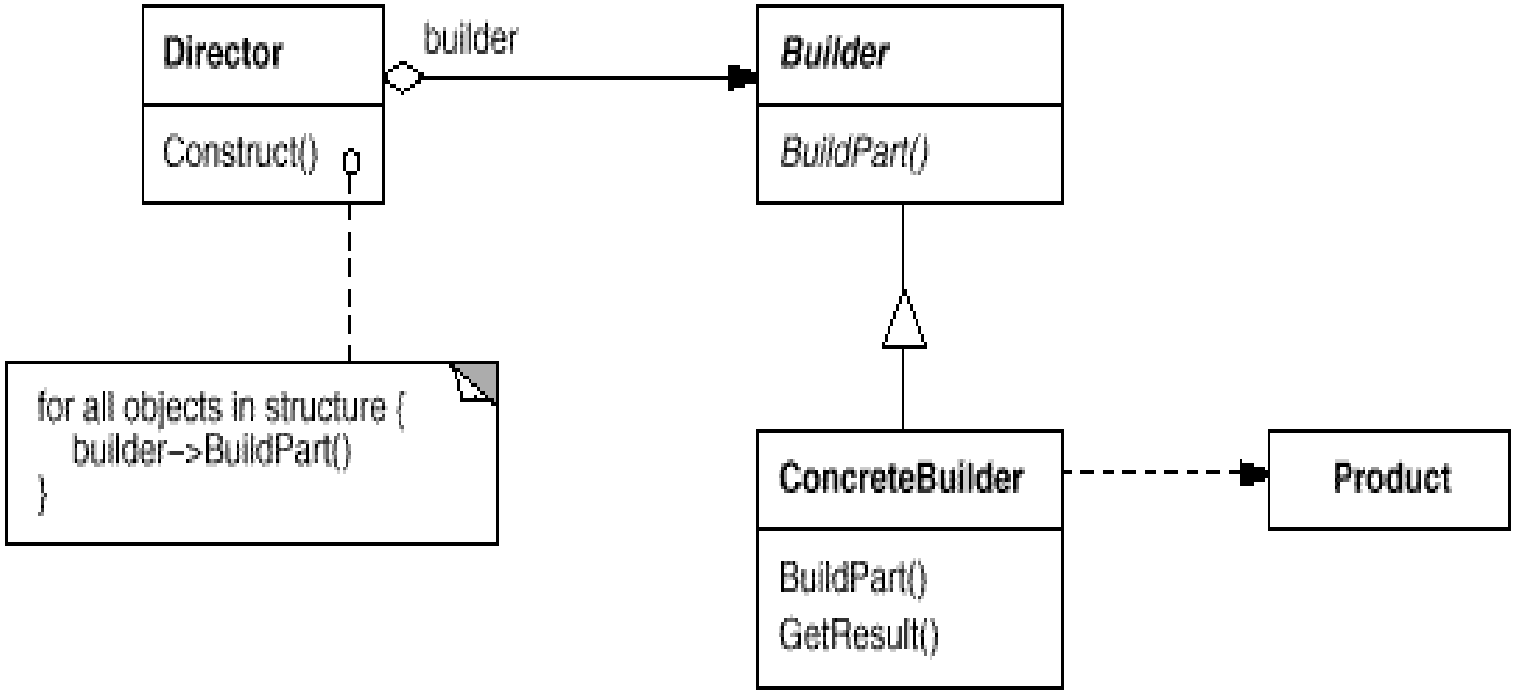
Abstract Factory

On utilise l'AbstractFactory lorsque :

- un système doit être indépendant de la façon dont ses produits sont créés, assemblés, représentés
- un système repose sur un produit d'une famille de produits
- une famille de produits doivent être utilisés ensemble, pour renforcer cette contrainte
- on veut définir une interface unique à une famille de produits concrets
- On souhaite changer globalement les types de containers utilisés par un programme (listes ou tableaux ou hashtables) par exemple

Builder

- Abstraire la construction d'objets complexes (séparer création et représentation)
- Permet de créer des composants morceaux par morceau progressivement (chaîne de montage)



Builder

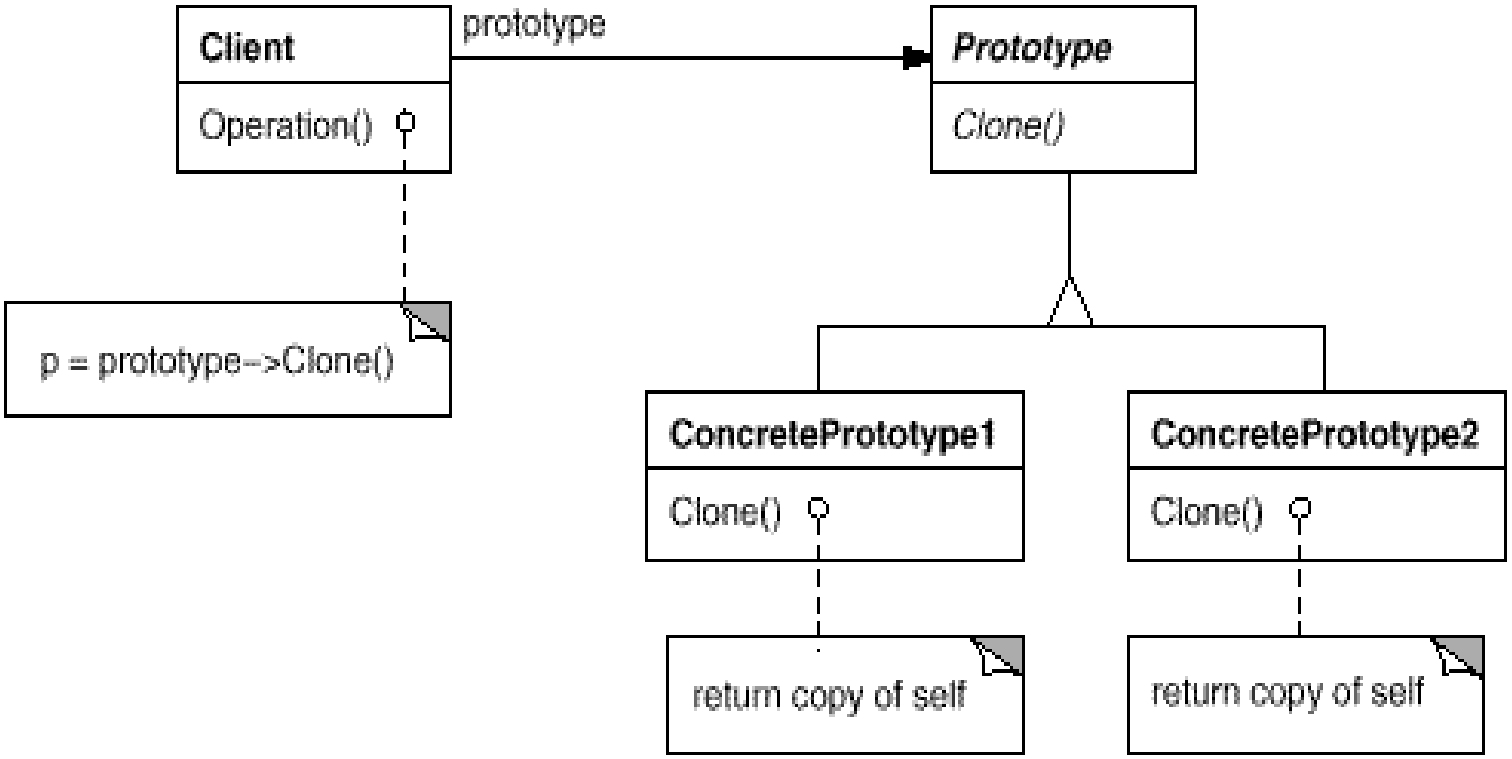
On passe en paramètre un objet qui sait construire l'objet à partir d'une description

On utilise Builder lorsque :

- l'algorithme pour créer un objet doit être indépendant des parties qui le compose et de la façon de les assembler
- le processus de construction permet différentes représentations de l'objet construit

Prototype

Création d'un prototype puis le dupliquer en créant des « clones », éventuellement de sous-classes différentes



Prototype

On utilise Prototype lorsque :

- un système doit être indépendant de la façon dont ses produits sont créés, assemblés, représentés
- quand la classe n'est connue qu'à l'exécution
- pour éviter une hiérarchie de Factory parallèle à une hiérarchie de produits

  (exemple de référence : mise en œuvre du copier/coller dans les interfaces graphiques)

Patterns de structure

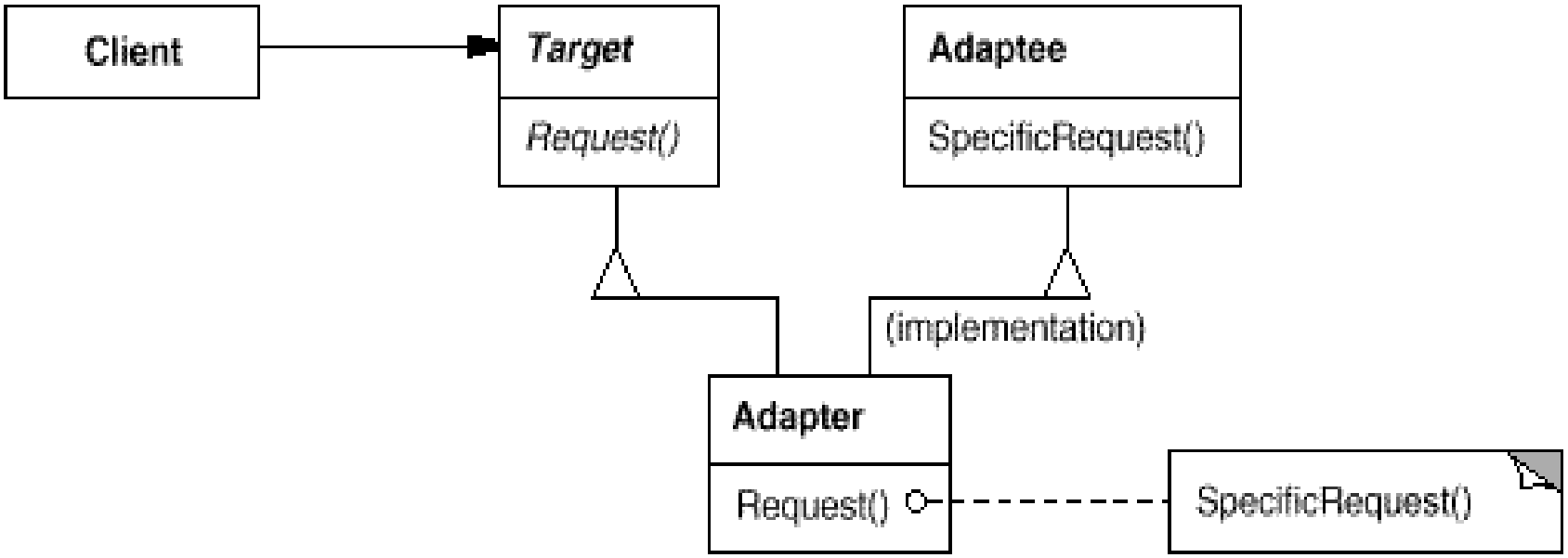
(Architecture statique de l'application)

Comment assembler des objets ?

Adapter

Objectif : Obtenir un objet qui permet d'en utiliser un autre en conformité avec une certaine interface

(convertir une interface en une autre pour réutilisation)



Adapter

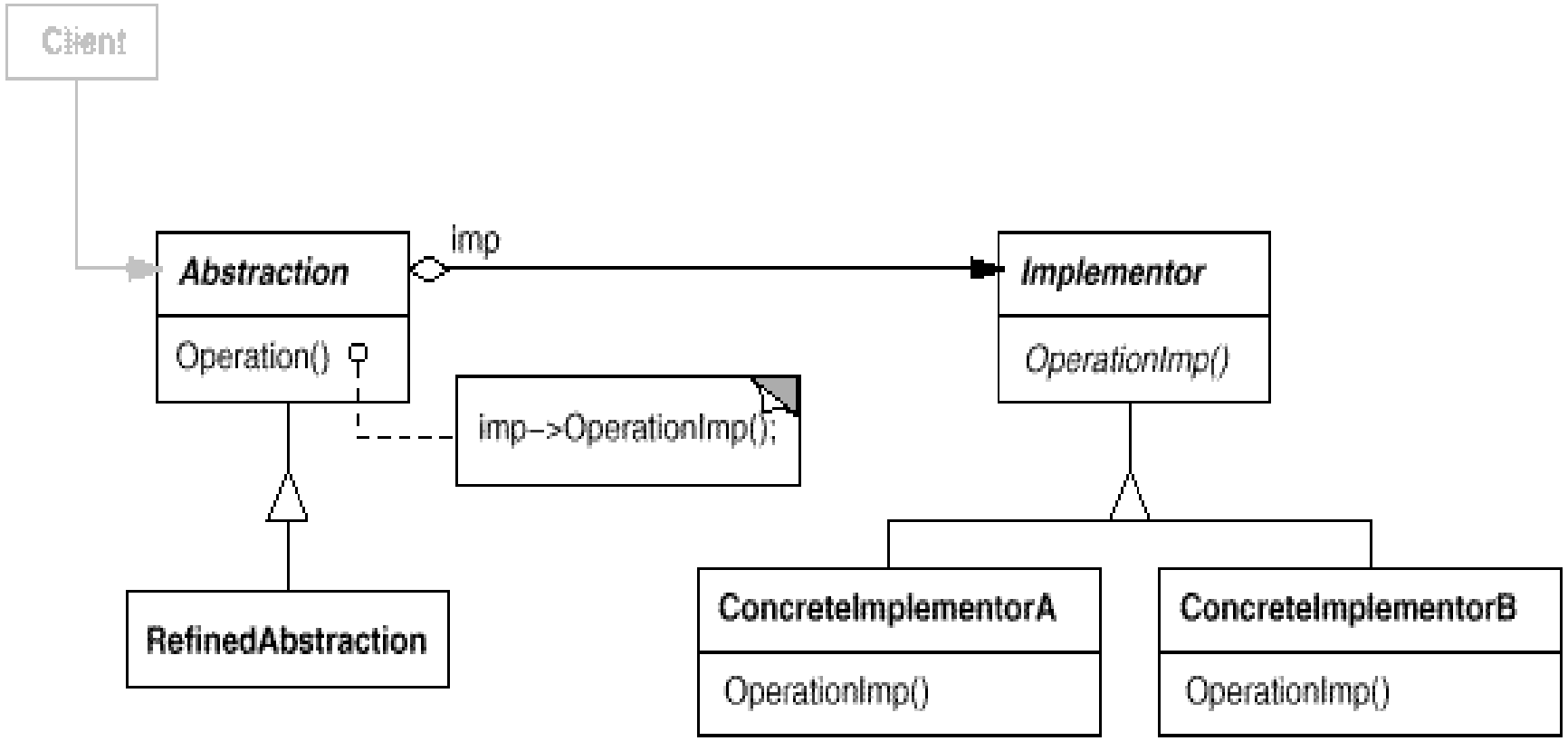
On utilise l'Adapter lorsque on veut utiliser :

- une classe existante dont l'interface ne convient pas
- plusieurs sous-classes mais il est coûteux de redéfinir l'interface de chaque sous-classe en les sous-classant. Un adapter peut adapter l'interface au niveau du parent.

Exemple : Mise en "conformité" de composants d'origines diverses : Pile à partir d'une liste

Bridge

Découpler une abstraction de son implantation



Bridge

On utilise Bridge lorsque :

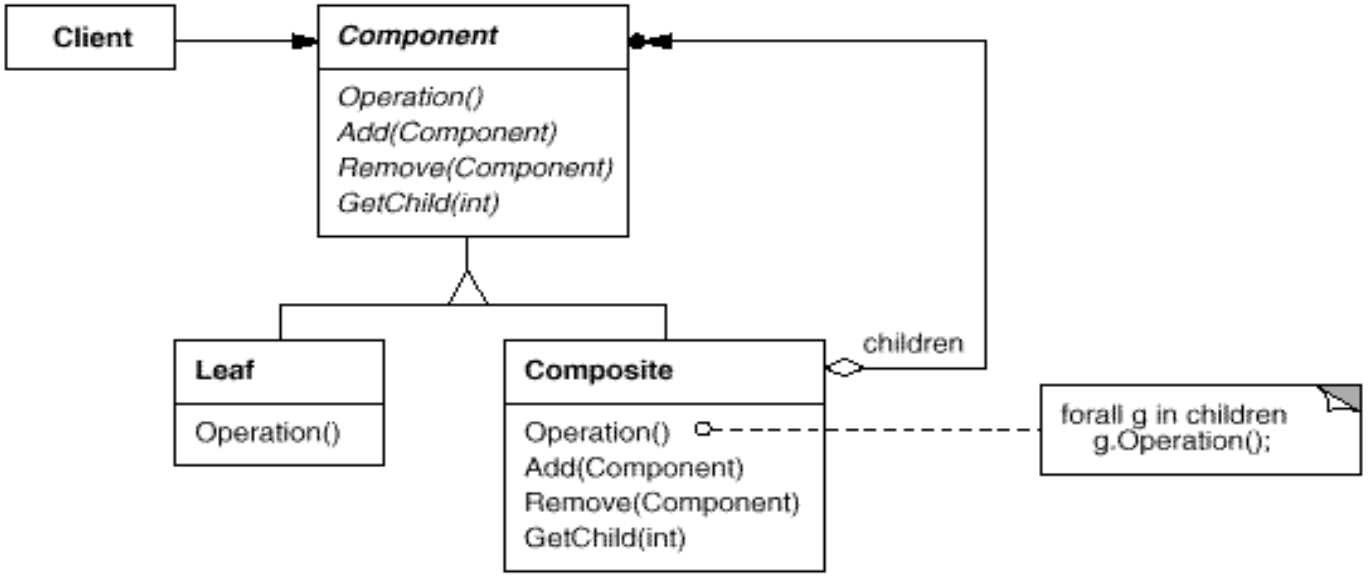
- on veut éviter un lien permanent entre l'abstraction et l'implantation (ex: l'implantation est choisie à l'exécution)
- l'abstraction et l'implantation sont toutes les deux susceptibles d'être raffinées
- les modifications subies par l'implantation ou l'abstraction ne doivent pas avoir d'impacts sur le client (pas de recompilation)

Composite

Objectif : Utilisation d'une liste/arbre pratique utilisant une classe A existante.

On utilise Composite lorsque on veut :

- représenter une hiérarchie d'objets
- ignorer la différence entre un composant simple et un composant en contenant d'autres. (interface uniforme)



Composite

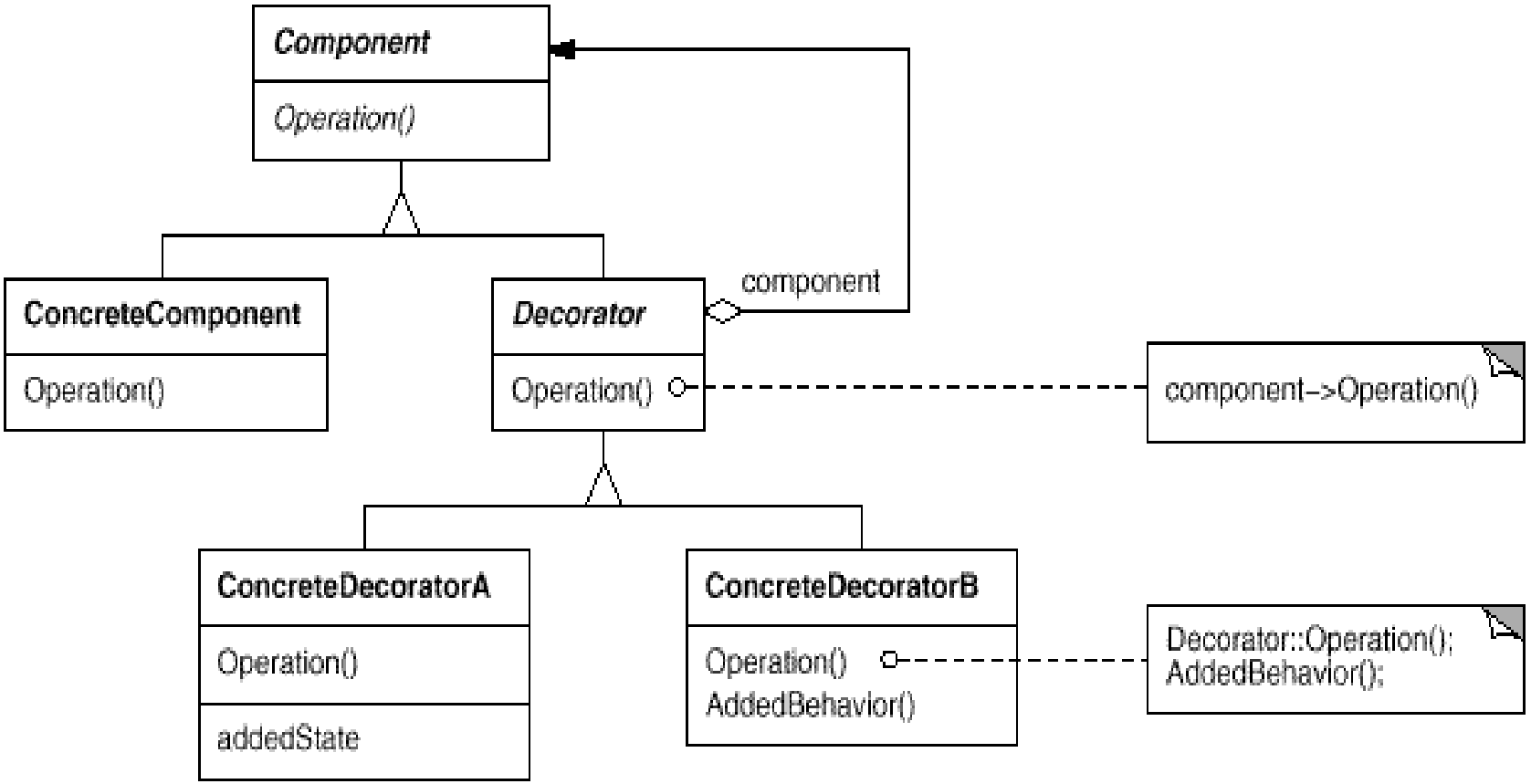
Solution : Réaliser une classe supplémentaire qui a la même signature que A, contient des A dans une collection et dispose de add(), remove(), iterator()...

Les méthodes sont réparties en :
Méthodes simples (dans la classe mère)
Méthodes récursives (abstraite dans la classe mère)
Méthodes de composition (dans la classe « branche », voire aussi dans la classe mère)

Exemple : Salariés de type employés (feuilles) et cadres (branches)

Decorator

Attacher des responsabilités dynamiquement



Decorator

On utilise Decorator lorsque :

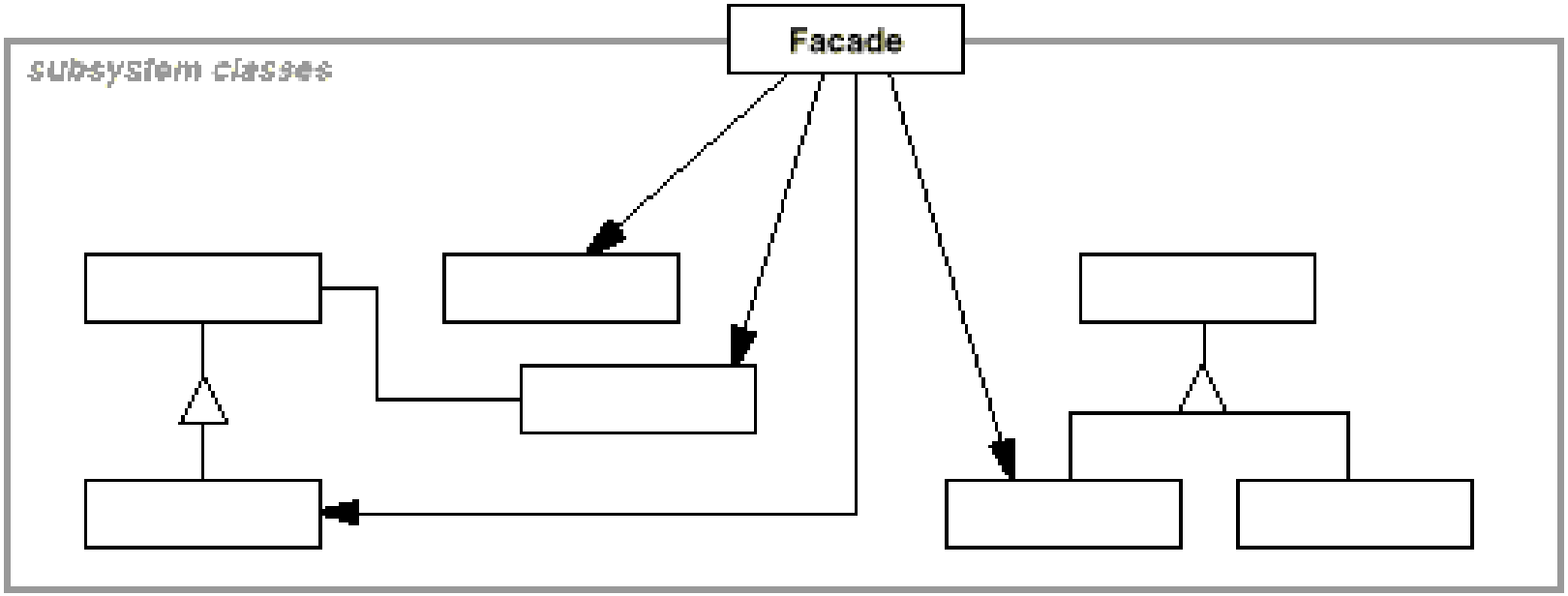
- il faut ajouter des responsabilités dynamiquement et de manière transparente
- il existe des responsabilités dont on peut se passer
- des extensions sont indépendantes et qu'il serait impraticable d'implanter par sous-classage

Pratique : dans les interfaces graphiques, l'affichage des "décorations" : barre de saisie, ascenseurs, transparence etc...

Facade

Objectif : Simplification de l'utilisation de nombreuses classes.

Solution : Réaliser une classe supplémentaire faisant un grand nombre d'appels à d'autres classes.



Facade

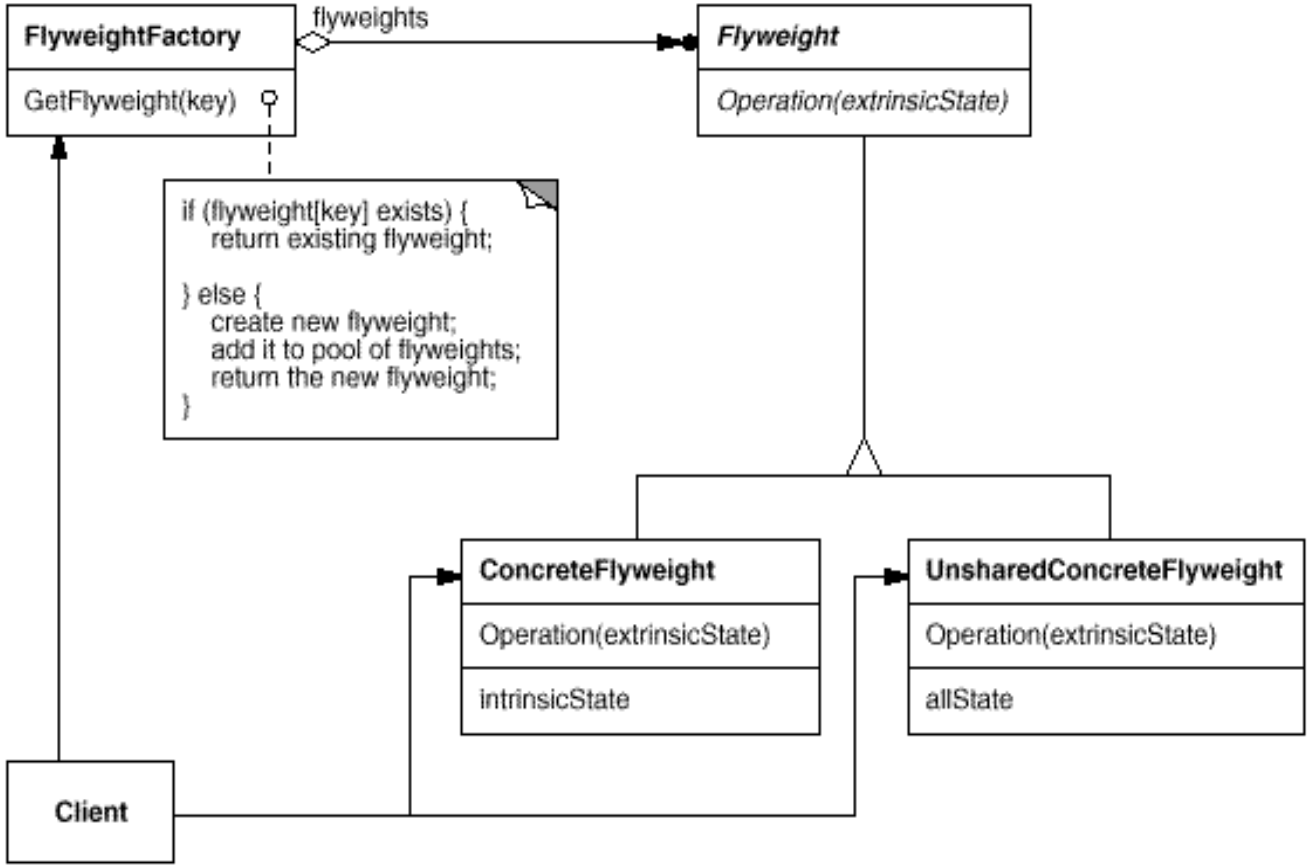
Utilisation :

- fournir une interface simple à un système complexe
- introduire une interface pour découpler les relations entre deux systèmes complexes
- construire le système en couche

Exemple : Corrections multiples sur un texte

Flyweight

Gérer des millions de pseudo objets associés à des données de base



Flyweight

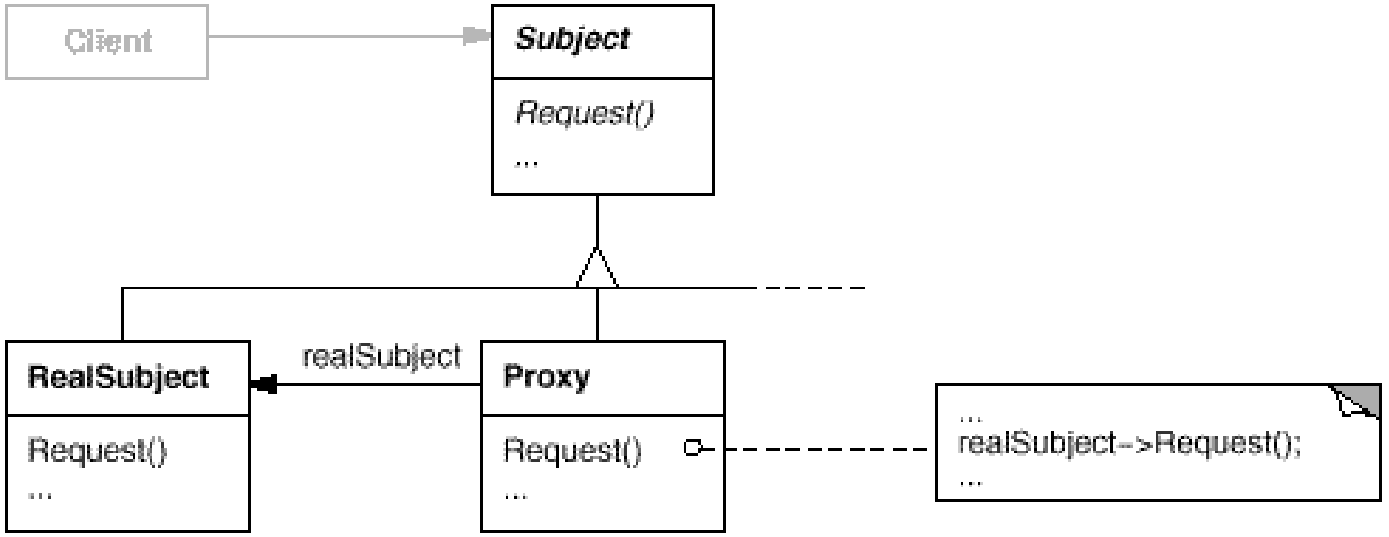
On utilise Flyweight lorsque :

- on utilise beaucoup d'objets, et les coûts de sauvegarde sont élevés,
- l'état des objets peut être externalisé et de nombreux groupes d'objets peuvent être remplacés par quelques objets partagés un fois que les états sont externalisés
- l'application ne dépend pas de l'identité des objets

Proxy

Objectif : Performance (optimisation d'un accès au système : processeur, mémoire, etc.) ou RMI (ex : EJB) ou protection(s) ou fonctionnalités (méthodes) supplémentaires, pour une classe A existante.

Obtenir un objet qui agit comme intermédiaire dans la communication avec un autre objet (un "passeur d'ordre").



Proxy

Utilisation : On utilise le Proxy lorsqu'on veut référencer un objet par un moyen plus complexe qu'un pointeur...

- remote proxy : ambassadeur
- protection proxy : contrôle d'accès
- référence intelligente

  - persistence

  - comptage de référence

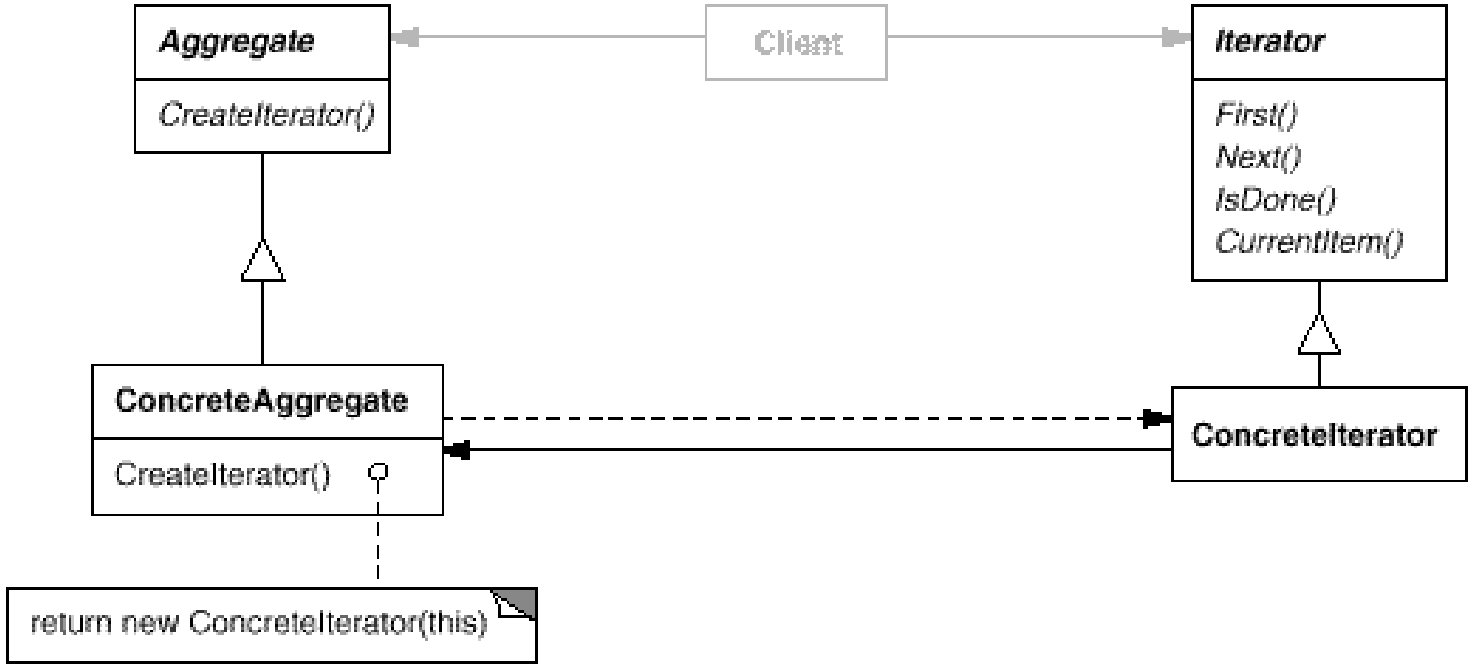
Solution : Réaliser une classe supplémentaire ayant la même signature que A et l'utilisant (agrégation).

Patterns de comportement
(Partie dynamique d'une application)

Design Patterns pour décrire :
- des algorithmes
- des comportements entre objets
- des formes de communication entre objet

Iterator

Objectif : permettre d'itérer de manière générique sur les éléments d'une collection, quelle que soit la nature des éléments ou de la collection



Iterator

Utilisation

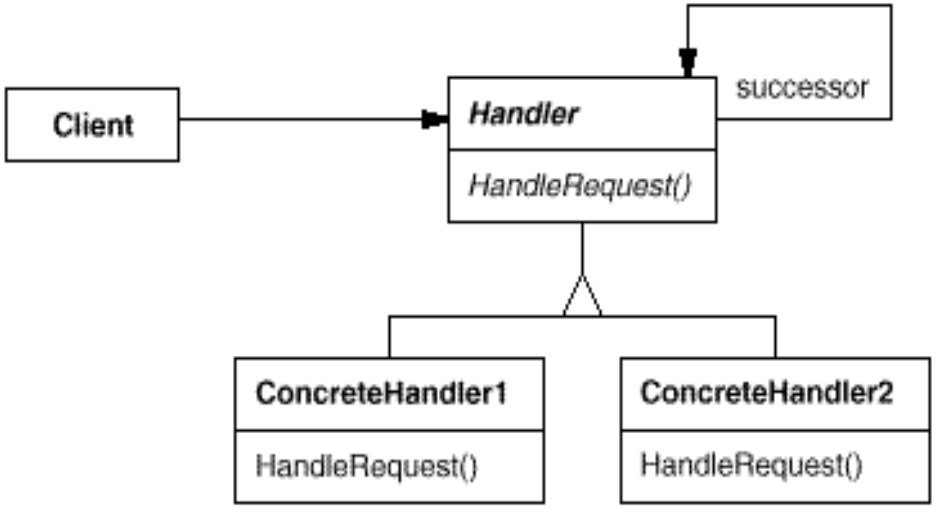
On utilise Iterator lorsque :

- pour accéder à un objet composé dont on ne veut pas exposer la structure interne
- pour offrir plusieurs manières de parcourir une structure composée
- pour offrir une interface uniforme pour parcourir différentes structures

Chain of responsibility

Objectif : Obtenir une hiérarchie dans la réponse à une interrogation.

Solution : Chaque classe, selon une même interface, répond à un appel ou bien passe l'appel à une classe « supérieure », selon une stratégie choisie.



Chain of responsibility

On utilise Chain of Responsibility lorsque :

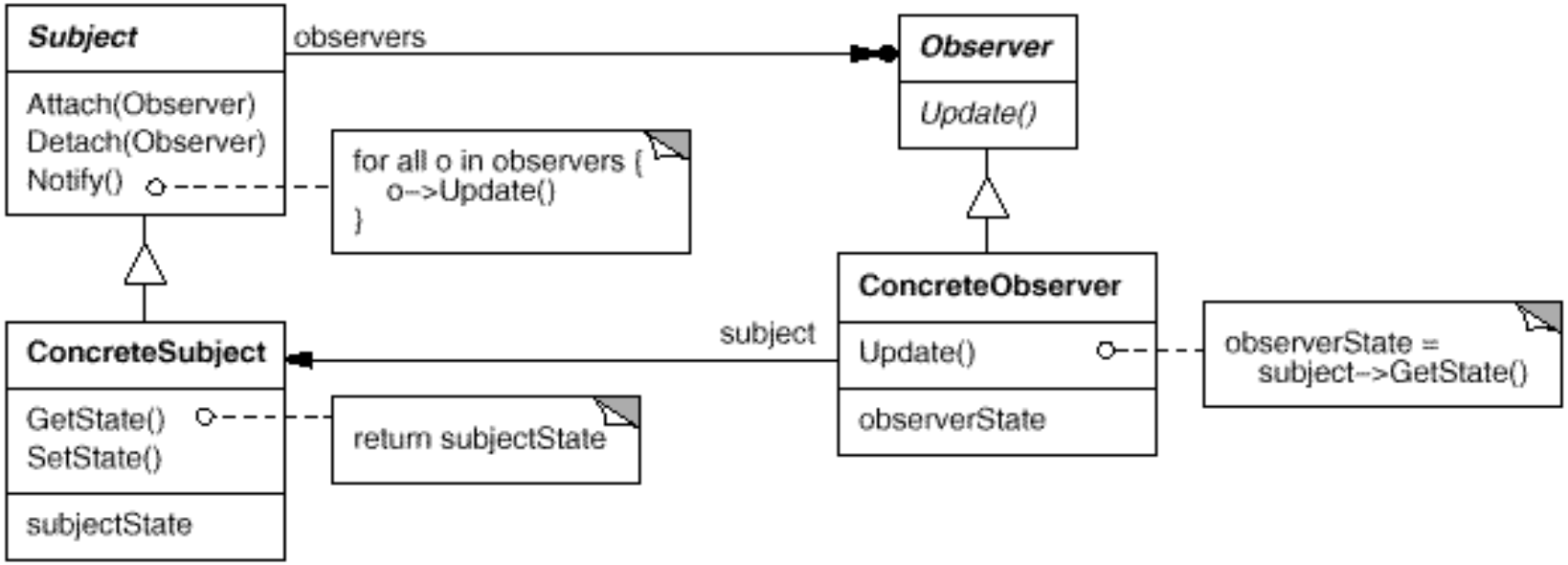
- plus d'un objet peut traiter une requète, et il n'est pas connu a priori
- l'ensemble des objets pouvant traiter une requète est construit dynamiquement

Exemple : Médiathèque avec catégories, sous-catégories, ... et recherche des favoris

Observer

Objectif : permettre à un objet d'informer d'autres objets qu'il ne connaît pas de l'évolution de son état interne.

Solution : Réaliser un observateur (l'observateur A s'enregistre auprès du sujet B, B signale des événements à A).



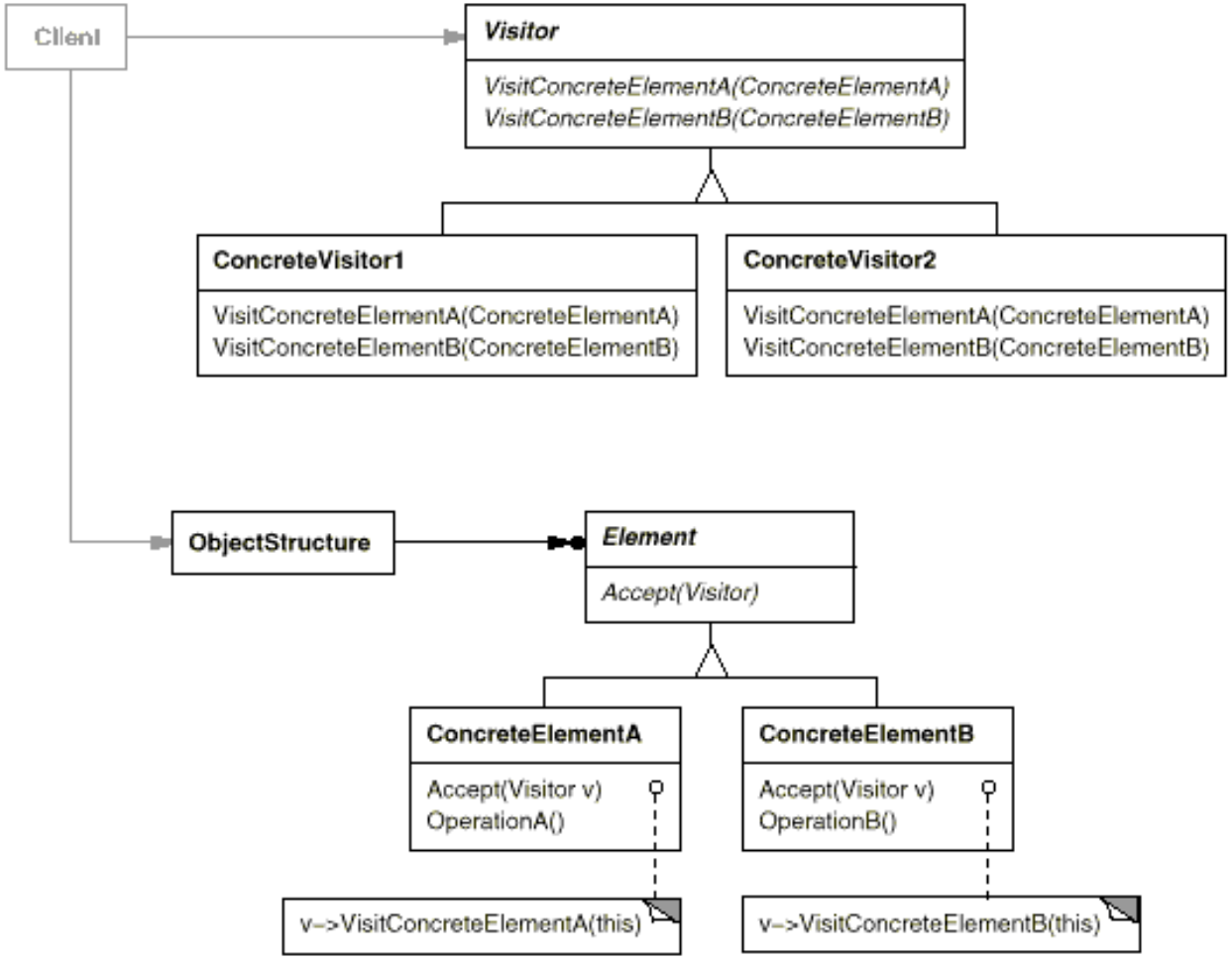
Observer

On utilise Observer lorsque :

- Une abstraction a plusieurs aspects, dépendant l'un de l'autre. Encapsuler ces aspects indépendamment permet de les réutiliser séparément.
- Quand le changement d'un objet se répercute vers d'autres.
- Quand un objet doit prévenir d'autres objets sans pour autant les connaître.

Visitor

Objectif : découpler une structure des opérations sur cette structure



Visitor

On utilise Visitor lorsque :

- une structure d'objets contient de nombreuses classes avec des interfaces différentes et on veut appliquer des opérations diverses sur ces objets.
- les structures sont assez stables, et les opérations sur leurs objets évolutives.

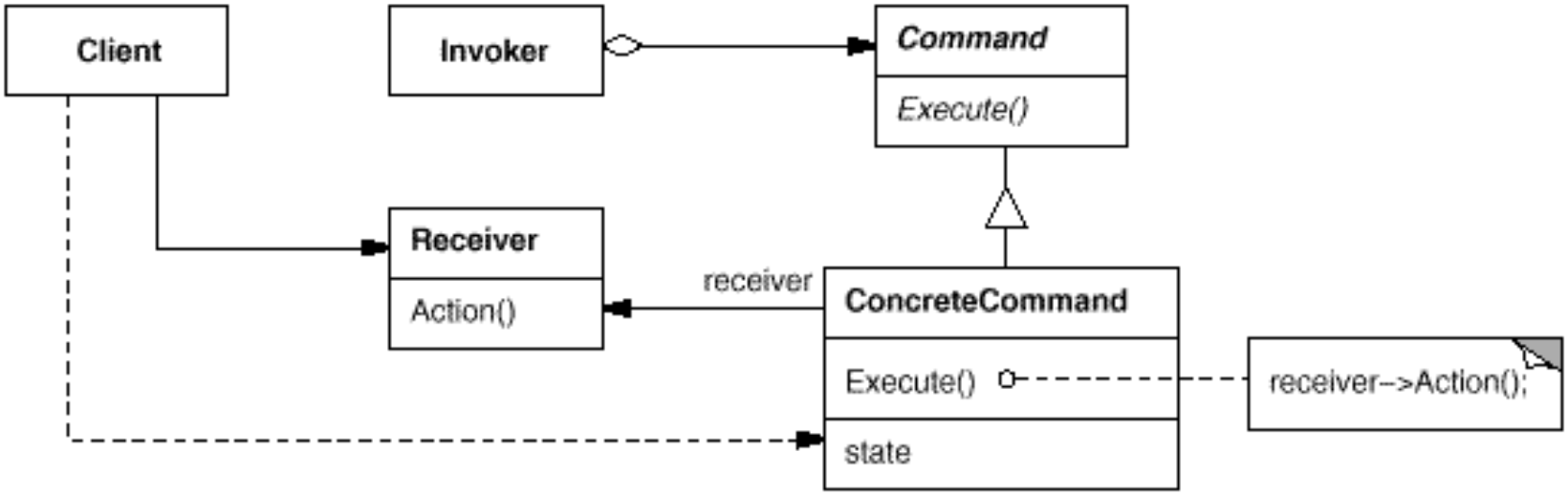
Solution : Réaliser une classe supplémentaire, le visiteur, disposant de visit(). Le visiteur agit sur d'autres classes, les visitées, qui disposent de accept() qui appelle visit().

Exemple : Employé et EmployéAncien : calcul des jours de congés

Command

Objectif : Simplifier un appel, le rendre plus facile à modifier.

Solution : Réaliser une classe A supplémentaire permettant d'exécuter une commande (série d'appels) sur une instance B, sauvegardée en tant qu'attribut de A.



Command

On utilise Command lorsque :

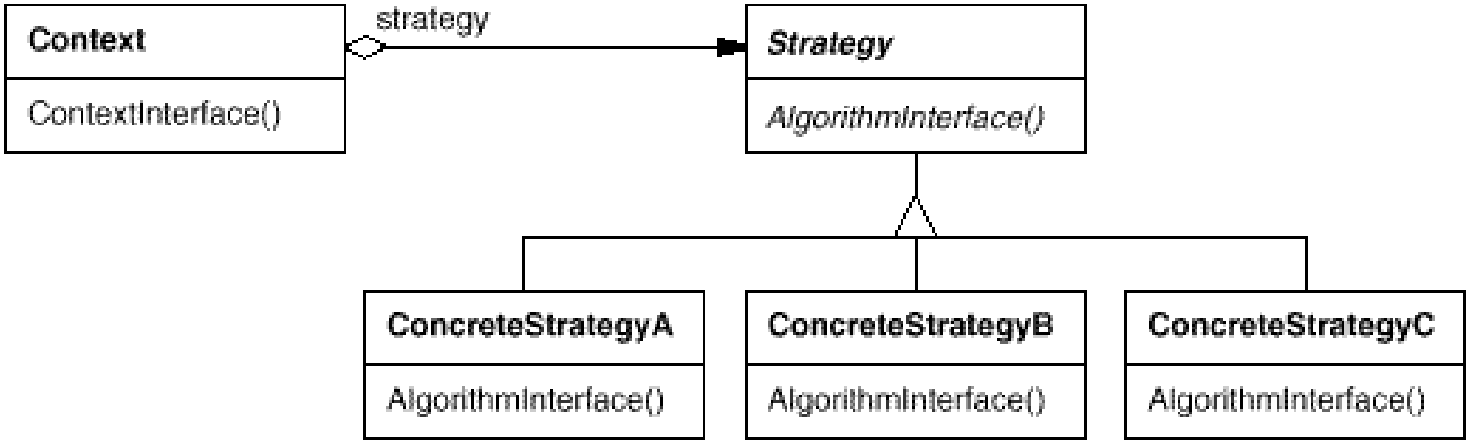
- spécifier, stocker et exécuter des actions à des moments différents.
- on veut pouvoir "défaire". Les commandes exécutées peuvent être stockées ainsi que les états des objets affectés...
- on veut implanter des transactions ;

  actions de "haut-niveau".

Strategy

Objectif : Permettre un choix durable entre des algorithmes similaires.

Solution : Réaliser une classe supplémentaire (« context ») qui permet de choisir entre plusieurs classes de même signature (« stratégies »).



Strategy

On utilise Strategy lorsque :

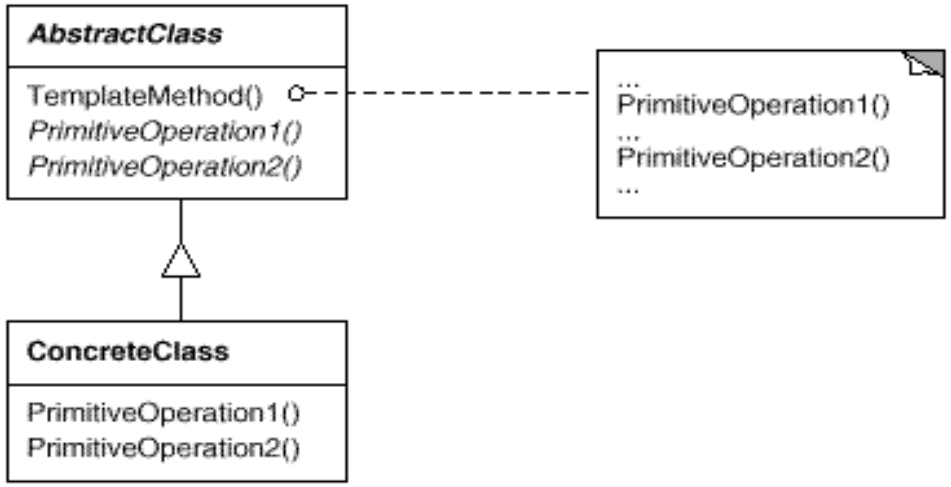
- de nombreuses classes associées ne diffèrent que par leur comportement. Stratégie offre un moyen de configurer une classe avec un comportement parmi plusieurs.
- on a besoin de plusieurs variantes d'algorithme.
- un algorithme utilise des données que les clients ne doivent pas connaitre.

Template Method

Prévoir un squelette de fonction, et compléter par les sous classes (raffinement de Strategy)

On utilise TemplateMethod pour :

- implanter une partie invariante d'un algorithme.
- partager des comportements communs d'une hiérarchie de classes.
- contrôler des extensions de sous-classe.

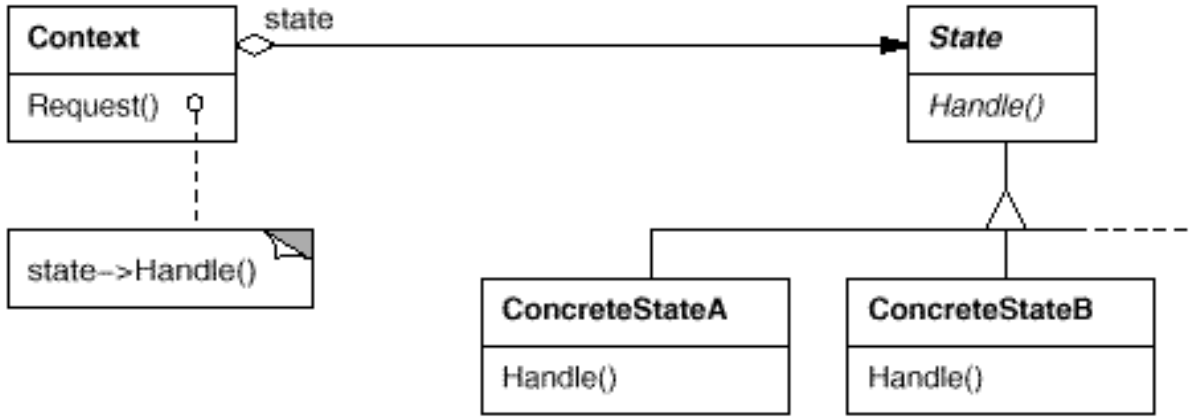


State

Changement de comportement en fonction de l'état de l'objet

On utilise State lorsque :

- Le comportement d'un objet dépend de son état, qui change à l'exécution
- Les opérations sont constituées de partie conditionnelles de grande taille (case)

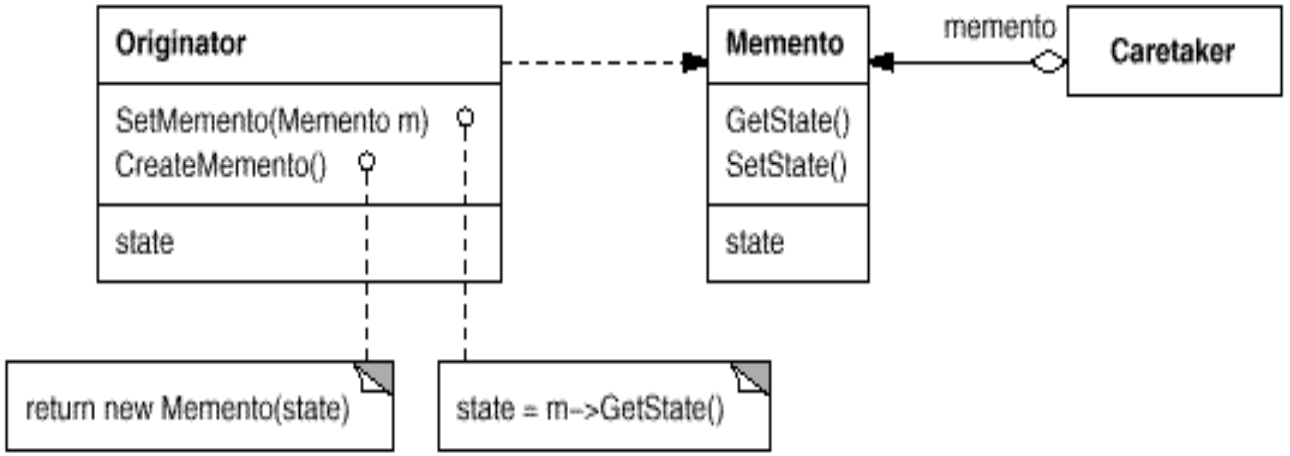


Memento

Sauver/ restaurer l'état d'un objet (persistance)

On utilise Memento lorsque :

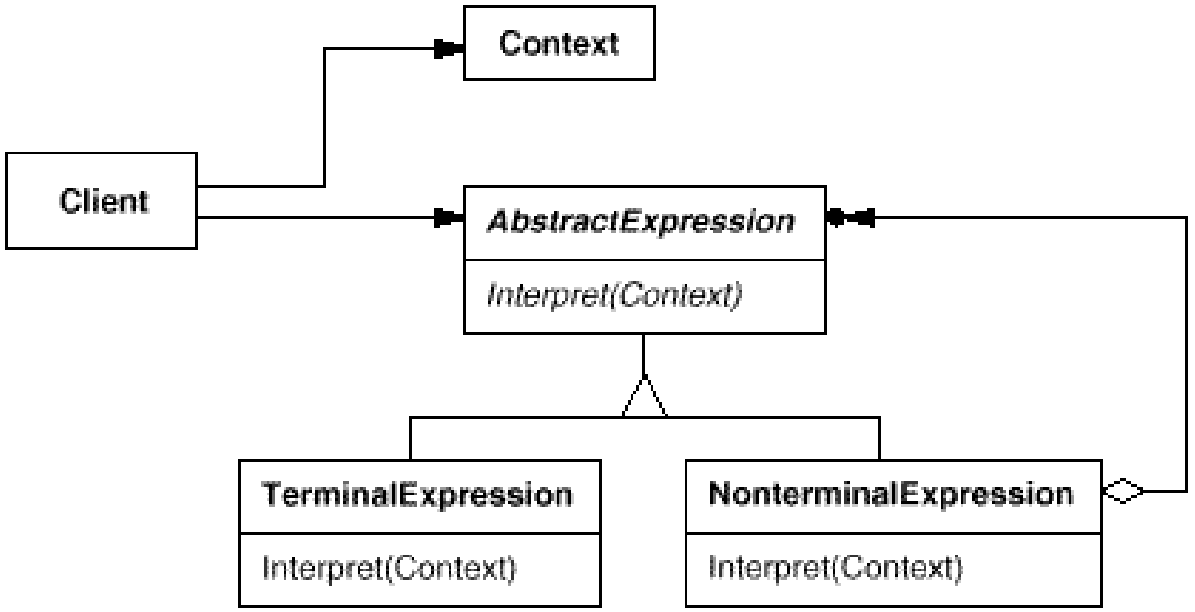
- on veut sauvegarder tout ou partie de l'état d'un objet pour éventuellement pouvoir le restaurer
- une interface directe pour obtenir l'état de l'objet briserait l'encapsulation



Interpreter

Représentation de la sémantique d'une grammaire

On utilise Interpreter lorsqu'il faut interpréter un langage et que la grammaire est simple et que l'efficacité n'est pas un paramètre critique

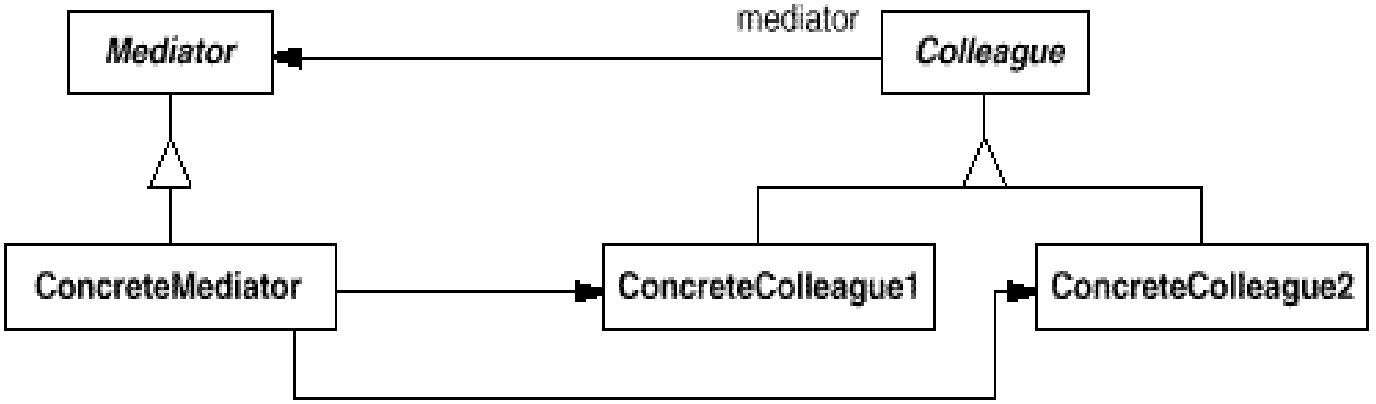


Mediator

Assure un couplage faible entre objets qui doivent Communiquer (Alléger le coût d'une communication nxn)

On utilise Mediator lorsque :

- de nombreux objets doivent communiquer
- la réutilisation d'un objet est délicate car il référence et communique avec de nombreux autres.



Synthèse

Design patterns

- Architecture de logiciel
- Des solutions-recettes adaptables pour des problèmes récurrents
- Pas réservé aux objets
- On les utilise sans le savoir ! Mais quand on en est conscient, on améliore sa réflexion.

En phase de conception

- Trouver les bons objets
- Bien choisir la granularité des objets
- Spécifier les interfaces des objets
- Spécifier l'implantation des objets
- Mieux réutiliser
  - héritage vs composition
  - délégation
- Compiled-Time vs Run-Time Structures
- Concevoir pour l'évolution

Trouver les bons objets

Les patterns proposent des abstractions qui n'apparaissent pas « naturellement » en observant le monde réel :

- Composite : permet de traiter uniformément une structure d'objets hétérogènes
- Strategy : permet d'implanter une famille d'algorithmes interchangeables
- State

Ils améliorent la flexibilité et la réutilisabilité

Granularité des objets

La taille des objets peut varier considérablement : comment choisir ce qui doit être décomposé ou au contraire regroupé ?

- Facade
- Flyweight
- Abstract Factory
- Builder

Spécifier les interfaces des objets

Qu'est-ce qui fait partie d'un objet ou non ?

- Memento : mémorise les états, retour arrière
- Decorator : augmente l'interface
- Proxy : interface délégué
- Visitor : regroupe des interfaces
- Facade : cache une structure complexe d'objet

Spécifier l'implantation des objets

Différence type-classe…

- Chain of Responsibility : même interface, mais implantations différentes
- Composite : les composants ont une même interface dont l'implantation est en partie partagée dans le Composite
- Command, Observer, State, Strategy ne sont souvent que des interfaces abstraites
- Prototype, Singleton, Factory, Builder sont des abstractions pour créer des objets qui permettent de penser en termes d'interfaces et de leur associer différentes implantations

Réutilisation

- Héritage vs Composition
  – white-box reuse ; rompt l'encapsulation - stricte ou non
  – black-box reuse ; flexible, dynamique
  "Préférez la composition à l'héritage"

- Délégation (redirection)
  – Une forme de composition...qui remplace l'héritage
  – Bridge découple l'interface de l'implantation
  – Mediator, Visitor, Proxy

Compiled-Time vs Run-Time Structures

- Agrégation
  - composition, is-part-of
  - dépendence, durée de vie liée, responsabilité
  - plus stable ~ compiled-time, statique

- Connaissance (accointance)
  - lien, association
  - relation fugitive, utilisation ponctuelle
  - dynamique, run-time

Concevoir pour l'évolution

Quelques raisons de "reengineering" :

- Création d'un objet en référençant sa classe explicitement...Lien à une implantation particulière...pour l'éviter : AbstractFactory, FactoryMethod, Prototype
- Dépendance d'une opération spécifique...pour rendre plus souple: Chain Of Responsibility, Command
- Dépendance d'une couche matérielle ou logicielle... AbstractFactory, Bridge
- Dépendance d'une implantation...pour rendre plus souple utilisez AbstractFactory, Bridge, Memento, Proxy
- Dépendance d'un algorithme particulier : Builder, Iterator, Strategy, TemplateMethod
- Couplage fort, relâcher les relations : AbstractFactory, Bridge, Chain Of Responsibility, Command, Facade, Mediator, Observer
- Etendre les fonctionnalités en sous-classant peut être couteux (tests, compréhension des superclasses, etc) utilisez aussi la délégation, la composition...Bridge, Chain Of Responsibility, Composite, Decorator, Observer, Strategy, Proxy
- Impossibilité de modifier une classe...absence du source, trop de répercussions : Adapter, Decorator, Visitor

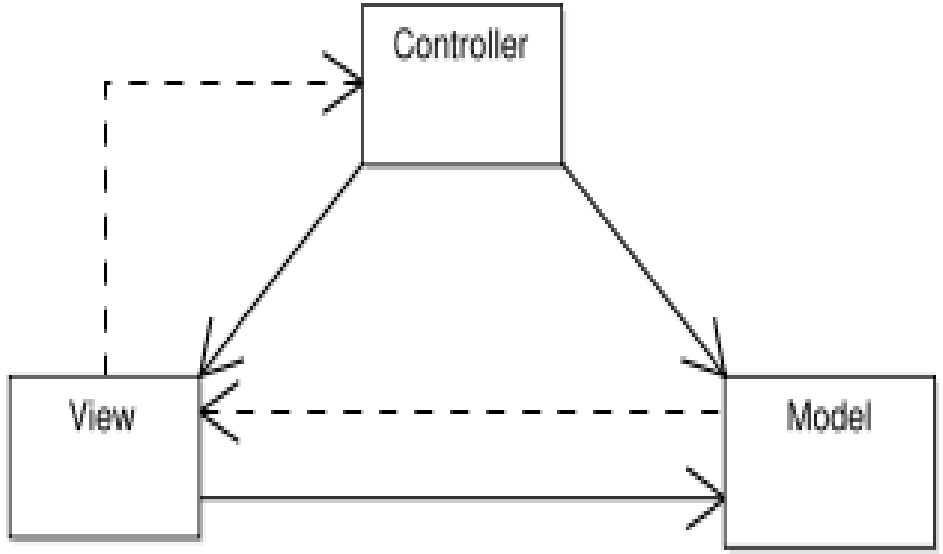
Autres Design Patterns

(Particularités du Web)

MVC

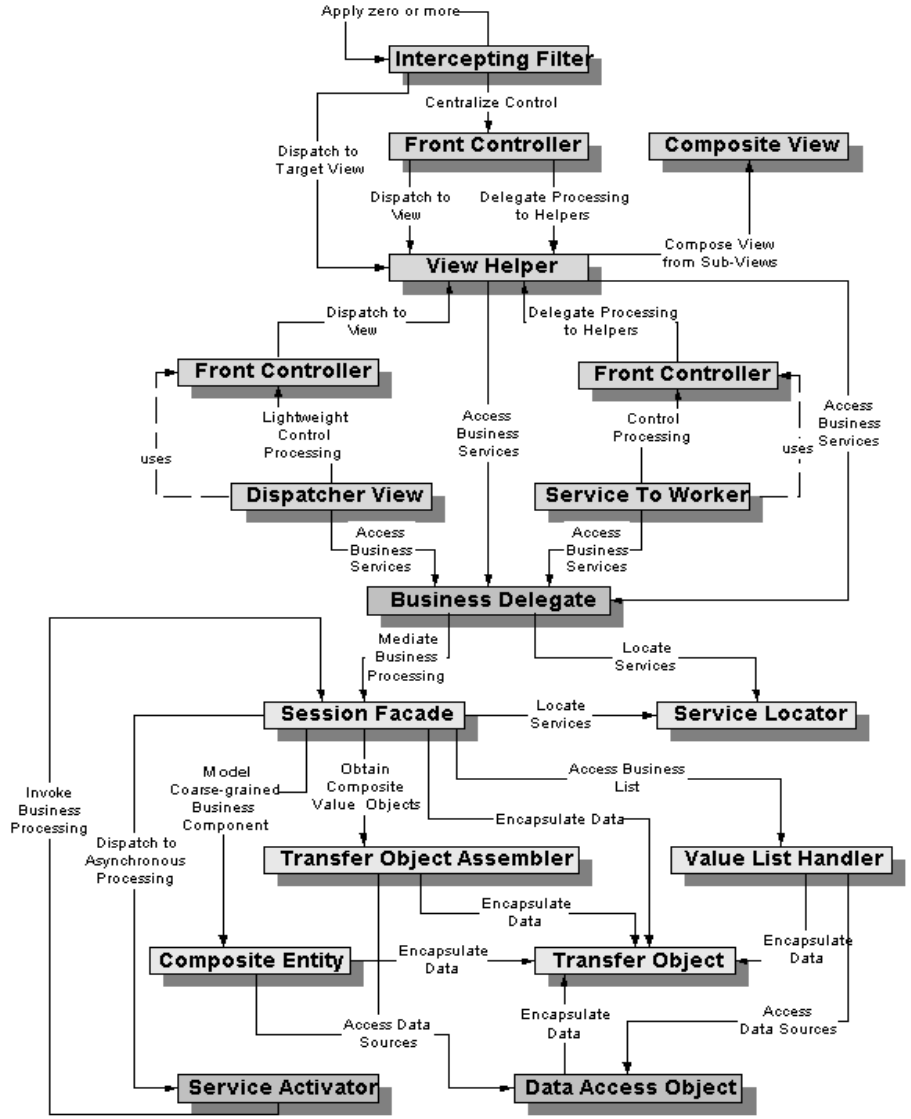
Le "Modèle Vue Contrôleur" est un pattern architectural souvent utilisé en programmation web.

Objectif : découpler l'accès de données et la logique métier de la présentation de données et de l'interaction utilisateur, en présentant un composant intermédiaire : le contrôleur.



JEE Design patterns

Sun propose, pour réaliser des applications JEE complète, l'application méthodique et systématique de 15 design patterns.



Common design patterns

J2EE design patterns	Software development patterns
Session Facade	Singleton
Value Object Assembler	Bridge
Service Locator Pattern	Prototype
Business Delegate	Abstract Factory
Composite Entity	Flyweight
Value List Handler	Mediator
Service Locator	Strategy
Composite Entity	Decorator
Value Object	State
Service to Worker	Iterator
Data Access Object	Chain of Responsibility
Intercepting Filter	Model View Controller II
View Helper	Memento
Composite View	Builder
Dispatcher View	Factory Method

Plus d'informations sur http://www.dawan.fr
Contactez notre service commercial au **0810.001.917** (prix d'un appel local)

DAWAN Paris, Tour CIT Montparnasse - 3,rue de l'Arrivée, 75015 PARIS
DAWAN Nantes, Le Sillon de Bretagne - 26e étage - 8, avenue des Thébaudières, 44800 ST-HERBLAIN
DAWAN Lyon, Le Britannia, 4ème étage - 20, boulevard Eugène Deruelle, 69003 LYON
DAWAN Lille, Parc du Chateau Rouge - 4ème étage, 276 avenue de la Marne, 59700 Marcq-en-Baroeul
formation@dawan.fr