

Programmation Orientée Objet Avancée

Arnaud Lewandowski

A series of horizontal lines in shades of blue and white, located on the right side of the slide, extending from the left edge of the text area.

Contenu du module

- Code conventions
- Refactoring
- Réflexivité, Introspection, Méta classes
- Modèles de conception (*Design patterns*)

Code Conventions

Code Conventions

- Pourquoi s'embêter avec ça ?!
 - 80% du coût de la durée de vie d'un soft est consacré à la maintenance
 - Rare sont les soft maintenus par leur auteur
 - Améliorent la lisibilité
(=> compréhension + rapide et + complète)
 - Produit de qualité, Professionnalisme

Code Conventions

- Références :
 - Java :
<http://www.oracle.com/technetwork/java/codeconv-138413.html>
 - Linux Kernel :
<http://www.kernel.org/doc/Documentation/CodingStyle>
 - GNU :
<http://www.gnu.org/prep/standards/standards.html>

Quelques conventions (1)

- Une seule classe ou interface publique par fichier
- Unité d'indentation = 4 espaces
- Lignes de 80 caractères max
 - Passage à la ligne
 - Après une virgule
 - Avant un opérateur
 - Indentation:
 - Au début de l'expression du même niveau
 - Ou 8 espaces

Quelques conventions (2)

- Commentaires
 - Dans le code, uniquement des infos nécessaires à la compréhension
 - Sinon = javadoc
- **Remarque** : si vous sentez le besoin d'ajouter un commentaire, réécrivez le code +simplement et +clairement

<http://www.codinghorror.com/blog/2008/07/coding-without-comments.html>

Quelques conventions (3)

- Déclarations
 - Variables
 - Uniquement en début de bloc
 - Une seule variable par ligne
 - Initialiser lors de la déclaration si possible
 - Méthodes
 - Pas d'espace entre un nom de méthode et la parenthèse (qui suit
 - Classes, if, for, while, ...
 - L'accolade { à la fin de la ligne
 - L'accolade } seule sur une nouvelle ligne

Quelques conventions (4)

- Un espace
 - Après la virgule
 - Entre un mot-clé et une parenthèse (
 - Autour de tous les opérateurs (sauf ++, --, .)
 - Avant/après une accolade
 - Après un cast

Quelques conventions (5)

```
public class MaBelleClasse extends Object {  
  
    static final int MA_CONSTANTE = 8;  
  
    int monAttribut;  
  
    public MaBelleClasse() {  
        // ... implémentation  
    }  
  
    public Object faitQuelqueChose() {  
        // ... implémentation  
    }  
}
```

Quelques conventions (6)

- Liste non exhaustive
 - => Voir la référence !
<http://www.oracle.com/technetwork/java/codeconv-138413.html>
- **Les programmes qui ne respectent pas les conventions de code ne seront pas corrigés**
- Pour vérifier : <http://checkstyle.sourceforge.net/>
- Voir aussi PMD (contrôle du respect des bonnes pratiques)

Refactoring

Le Refactoring (1)

- Principe :
 - Modification du code sans ajout de fonctionnalité
- Objectif :
 - Améliorer la lisibilité
 - Rendre le code plus maintenable/évolutif
- Réorganisation / restructuration du code

Le Refactoring (2)

- En principe
 - On ne change pas un code qui marche
 - Surtout si aucune fonctionnalité en plus
 - Risque d'introduction de bugs
- MAIS
 - Dev. itératif incrémental, corrections de bugs
 - Les modifs successives complexifient le code
 - On n'a pas le choix !

Le Refactoring (3)

- NB: Le refactoring n'est pas une solution miracle...
 - Utilisation de tests unitaires (non régression)

Les niveaux de Refactoring

- Modification de la présentation
 - Commentaires, mise en page, etc.
- Modification de l'algorithmique
 - Objectif : méthodes aussi simples que possible
- Relocalisation de procédures
- Refonte de la conception
 - Modification de la hiérarchie de classes

Activités de Refactoring (1)

- Suppression du code mort
 - Détection de code non utilisé: *grep*, analyseur de réf. croisées, outil de mesure de couverture de code
 - Le code commenté « au cas où »
- Ajout d'assertions
 - Règles à respecter
 - Facilitent le débogage, aident à la compréhension
- Renommage
 - Rôles des méthodes / classes
- Commentaires

Activités de Refactoring (2)

- Extraction (classe, méthode, variable, constante, interface)
- Conversion (variable locale vers attribut)
- Déplacement (vers une sous-classe, une super-classe, ou une autre classe)
- Encapsulation
- Généralisation
- etc.

Le Refactoring : Références

- <http://jmdoudoux.developpez.com/cours/developpons/eclipse/chap-eclipse-refactoring.php>
- <http://blog.excilys.com/2010/08/03/refactoring-par-la-pratique/>

TP/Tutoriel : découverte de PMD...

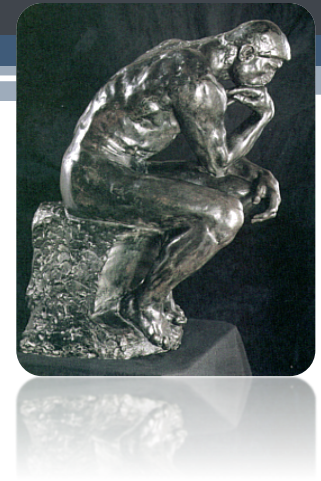
TP : refactoring

<http://www.cs.virginia.edu/~horton/cs494/s05/slides/lab-exercise-refactoring.htm>

Réflexivité, Introspection

La Réflexivité

- Capacité d'un programme à **examiner** et à éventuellement **modifier** sa structure, son état et/ou son comportement
- 2 catégories :
 - L'introspection
 - L'intercession



L'Introspection (1)

du latin « *introspectus* », action de regarder à l'intérieur

Utilité :

- Analyse des classes, découverte d'attributs, de méthodes
- Chargement dynamique de classes, création d'instances
- Inspecter des objets dynamiquement
- Manipuler des tableaux génériques, créer des méthodes génériques
- Débogage, profiling, complétion automatique
- Etc.

L'Introspection (2)

- Java maintient l'Identification de Type au Run-Time (RTTI) sur tous les objets
 - Permet de connaître la classe réelle d'un objet
 - Permet d'implémenter la liaison dynamique (quelle méthode est réellement appelée) :

```
ObjetGraphique o = new Cercle();  
o.draw(); // de Cercle ou de ObjetGraphique ?;
```

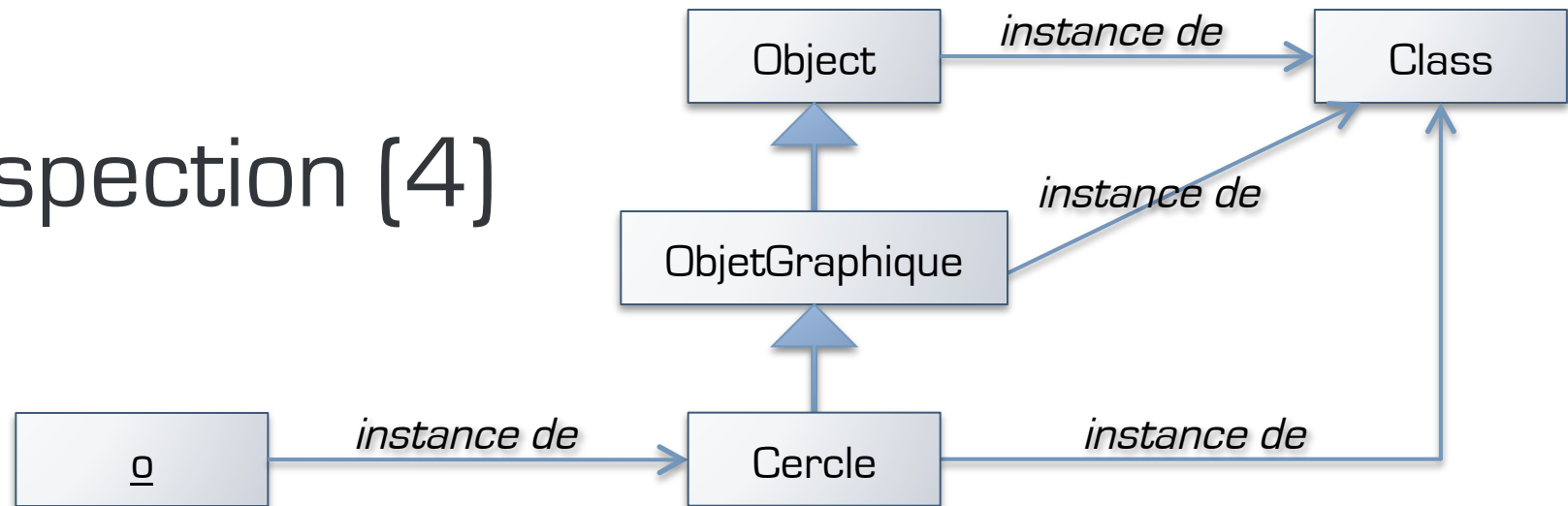
L'Introspection (3)

- Accès au RTTI grâce à l'API : `java.lang.reflect`
- On peut ainsi connaître la classe d'un objet :

```
Class c11 = o.getClass();  
Class c12 = Cercle.class;  
Class c13 = int.class;
```

- Un objet de type `Class` décrit un type, pas forcément une classe...

L'Introspection (4)



- Les représentations des classes sont des instances de Class et sont donc ... des objets !!
 - Possèdent des méthodes, des attributs
 - On peut les appeler
- La représentation de la classe Object est aussi une instance de Class (puisque Object est une classe...)
 - On peut donc l'utiliser de la même façon

L'Introspection (5)

- Exemple d'utilisation :

Utile pour instancer des classes dont le nom n'est connu qu'à l'exécution

```
// Sans utiliser l'introspection  
Foo foo = new Foo();  
foo.hello();
```

```
// En utilisant l'introspection  
Class cl = Class.forName("Foo");  
Method method = cl.getMethod("hello", null);  
method.invoke(cl.newInstance(), null);
```

Appel de méthode

Création d'une nouvelle instance

L'Intercession

- [ou Méta-programmation]
- Capacité d'un programme à modifier son code
- Non disponible en Java

Ex. en Python :

```
>>> class A:  
...     def hello(self):  
...         print "hello"  
...  
>>> A().hello()  
hello
```

```
>>> def hello2(self):  
...     print "hello arno"  
...  
>>> A.hello = hello2  
>>> A().hello()  
hello arno
```

TP : Analyse d'un objet inconnu

- Cf. énoncé de Y. Royon

Les Design Patterns

Modèles de Conception (Design Patterns)

- Introduction
- Les patterns de Création
- Les patterns de Structure
- Les patterns de Comportement
- Les anti-patterns

Les design patterns – Introduction

- En architecture
 - Description d'un pb récurrent et de sa solution
 - Solution peut être utilisée des millions de fois sans être 2 fois identique
 - **Forme, pattern, modèle, patron** de conception
 - Mur, porte, fenêtre ↔ objet, classe, interface

Les design patterns

- **CE QUE C'EST :**

- Une technique d'architecture logicielle
- Formalisation de bonnes pratiques
- Description d'une solution standard à un pb récurrent d'architecture ou de conception
- Décrit une partie de la solution
- Indépendant du langage de programmation

Les design patterns

- **CE QUE CE N'EST PAS :**
 - Une brique
 - Un pattern dépend de son environnement
 - Une règle
 - Un pattern ne peut pas s'appliquer mécaniquement
 - Une méthode
 - Il ne guide pas la prise de décision; un pattern *est* la décision prise
 - Nouveau

Les design patterns

- **AVANTAGES**

- Vocabulaire commun
- Capitalisation d'expérience
- Niveau d'abstraction +élevé
 - ➔ constructions logicielles de meilleure qualité
- Réduit la complexité
- Catalogue de solutions éprouvées

Les design patterns

- **INCONVÉNIENTS/DIFFICULTÉS**
 - Effort de synthèse : reconnaître, abstraire
 - Apprentissage, expérience
 - Les patterns se « dissolvent » en étant utilisés
 - Beaucoup de patterns !
 - Lesquels sont identiques ?
 - Lequel est le mieux adapté ?
 - Niveaux \neq , certains patterns s'appuient sur d'autres, ...

Les design patterns

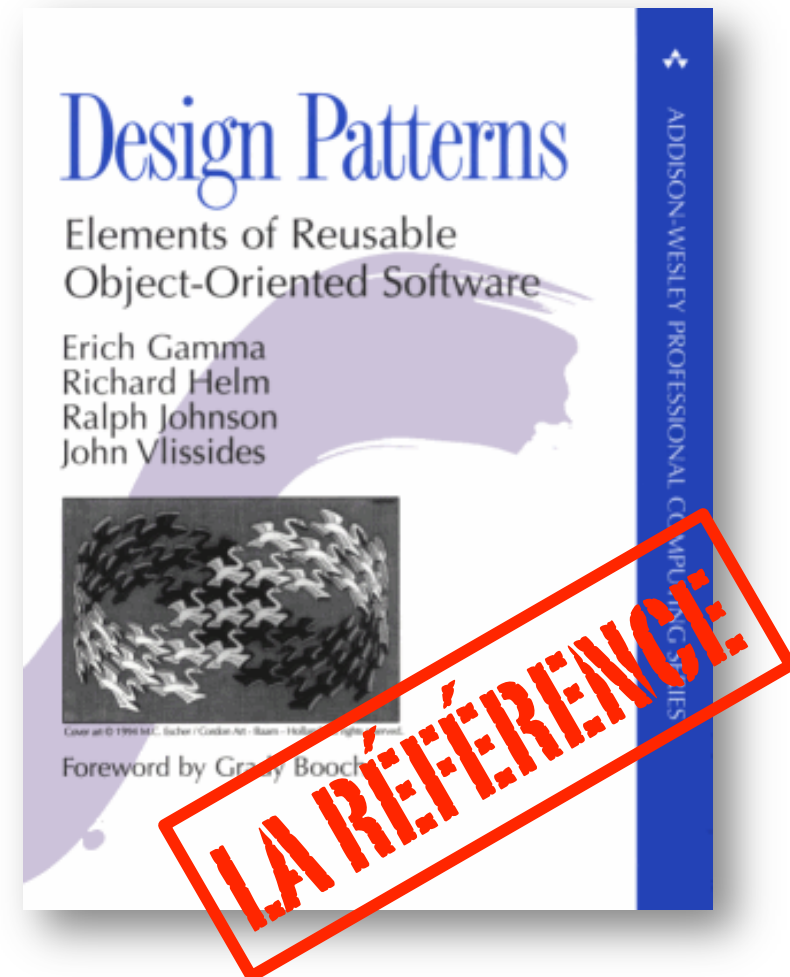
- Description d'un **patron** :
 - Nom
 - L'énoncé du problème
 - quand l'appliquer, dans quel contexte
 - Solution
 - Formalisée en UML
 - Les éléments de la solution, leurs relations, etc.
 - Conséquences
 - Résultats et compromis issus de l'application

Les design patterns

- Principe général : les patrons sont indépendants du langage de programmation
- Influence des langages sur les patrons
 - des langages implémentent des patrons de bas niveau
 - quelques patrons utilisent des concepts spécifiques à certains langages
 - quelques patrons ne sont pas indépendants du langage
 - certains langages forcent à tort des patrons compliqués lors de l'implémentation
- Influence des patrons sur les langages
 - Les patrons capitalisent l'état de réflexion courant sur les pratiques de programmation

Références

- Les designs patterns du **GoF** (Gang of Four)
 - Erich Gamma
 - Richard Helm
 - Ralph Johnson
 - John Vlissides



Références

- Sur le web, LE site :
 - <http://www.vincehuston.org/dp/>

Les design patterns

- 23 patrons répandus, appelés « patrons du GoF » classifié en 3 familles
 - Patrons de création (**Creational patterns**)
 - Comment instancier et configurer classes/objets
 - Patrons de structure (**Structural patterns**)
 - Comment organiser les classes (séparation interface/implémentation)
 - Patrons de comportement (**Behavioral patterns**)
 - Comment organiser les objets pour qu'ils collaborent (distribution des responsabilités)

| | Creational | Structural | Behavioral |
|--------|---|--|---|
| Class | Factory Method | Adapter (class) | Interpreter Template Method |
| Object | Abstract Factory Builder Prototype Singleton | Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy | Chain of Respons. Command Iterator Mediator Memento Observer State Strategy Visitor |

Les patterns de création

Les patterns de création

- **Ce qu'ils permettent**
 - Abstraire le processus d'instanciation
 - Rendre le système indépendant de la façon dont les objets sont créés, composés, représentés
- **Comment ils y parviennent**
 - En encapsulant la connaissance sur les classes concrètes utilisées par le système
 - En cachant la façon dont les instances de ces classes sont créées et assemblées (qui, quoi, quand et comment)

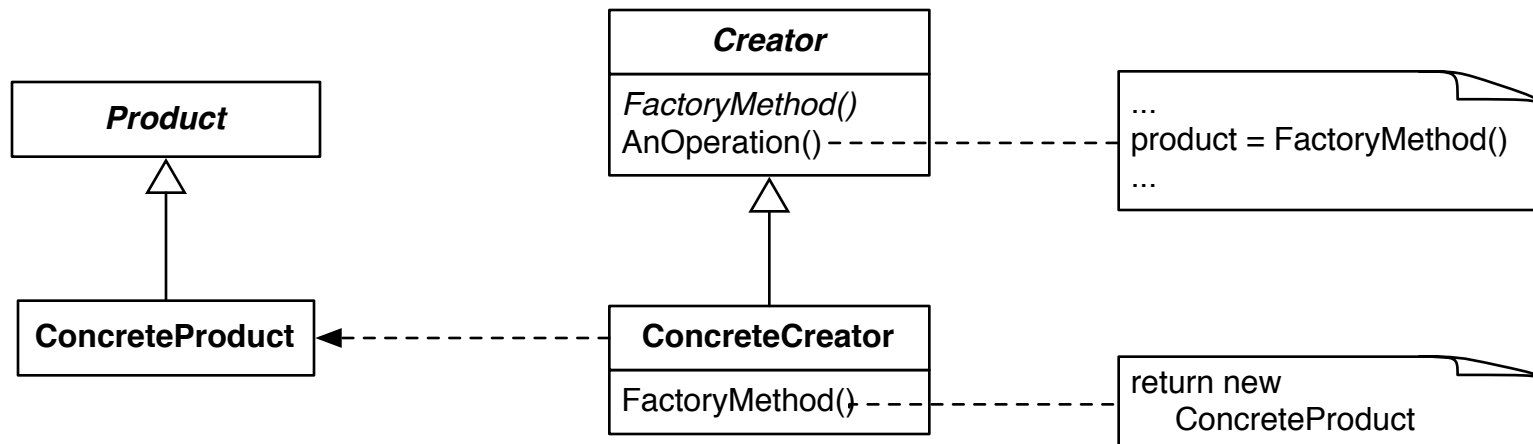
Les patterns de création

| | |
|-------------------------|---|
| Abstract Factory | Pour gérer différentes familles de produits |
| Builder | Sépare la construction d'un objet complexe de sa représentation |
| Factory Method | La classe sollicitée appelle des méthodes abstraites |
| Prototype | Création d'instances par clonage de prototypes |
| Singleton | Instance unique |

Factory Method

- **On l'utilise quand :**
 - une classe ne connaît pas à l'avance la classe des objets qu'elle doit créer
 - On veut facilement obtenir un objet prêt à l'emploi
 - une classe veut déléguer la responsabilité de création des objets à ses sous-classes

Factory Method



Factory Method

- **Conséquences**
 - Fournit des **hooks** pour les sous-classes
 - Permet de connecter des hiérarchies de classes « parallèles »

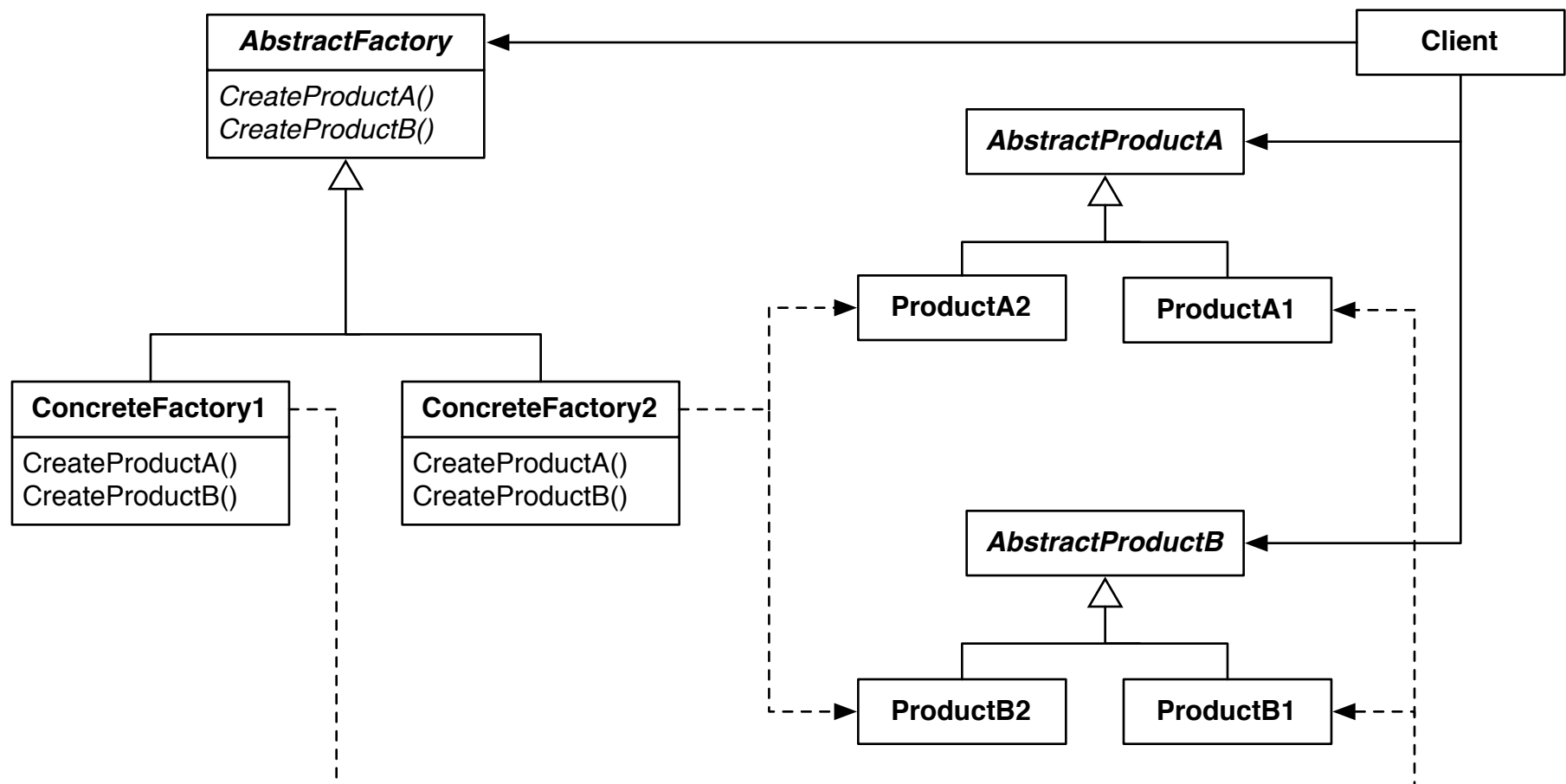
Abstract Factory

- **On l'utilise quand :**
 - un système doit être indépendant de la façon dont ses produits sont créés, composés et représentés
 - un système doit être configuré suivant une famille de produits parmi plusieurs familles
 - pour renforcer le fait que des produits d'une même famille doivent être utilisés ensemble
 - on ne veut afficher que les interfaces d'une librairie de produits, et cacher leur implémentation

Abstract Factory

- **Principe de fonctionnement**
 - C'est un regroupement de Fabriques (Factory Method)
 - La Fabrique détermine le type de l'objet concret à créer, mais retourne un pointeur abstrait sur l'objet créé
 - Le client est isolé de la création de l'objet
 - Le client est obligé de demander à une fabrique de créer l'objet du type abstrait désiré

Abstract Factory



Abstract Factory

- **Conséquences**

- Le client n'a aucune connaissance du type concret ;
- Les classes concrètes sont isolées
- Les objets concrets sont créés par la fabrique
- Le client n'interagit avec les objets concrets qu'au moyen de leur classe abstraite
- Ajouter de nouveaux types concrets se fait en spécifiant une nouvelle fabrique (Factory Method)
- On peut facilement changer de famille de produits
- Ajouter un nouveau type de produit est plus difficile

Builder

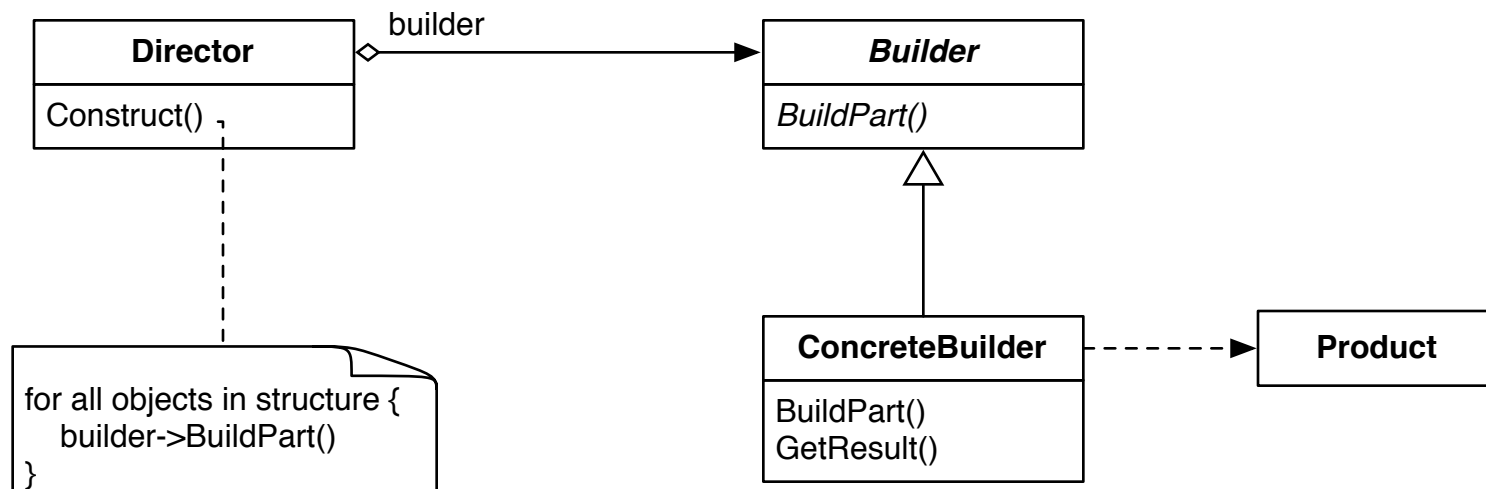
- **On l'utilise quand :**
 - L'algo de création d'un objet complexe doit être indépendant des parties qui constituent l'objet et de leur assemblage
 - Le processus de construction doit permettre plusieurs représentations de l'objet

Builder

- **Principe de fonctionnement**

- Le Client crée un objet Directeur et le configure avec le Builder désiré
- Le Directeur notifie le Builder lorsqu'une partie doit être construite
- Le Builder répond aux sollicitations du Directeur en créant les parties du Produit
- Le Client récupère le Produit en le demandant au Builder

Builder



Builder

- **Conséquences**

- On peut avoir plusieurs représentations d'un produit (il suffit d'ajouter un nouveau Concrete Builder)
- Le code pour la construction et la représentation est isolé (dans les Concrete Builder)
- Contrôle plus fin du processus de construction

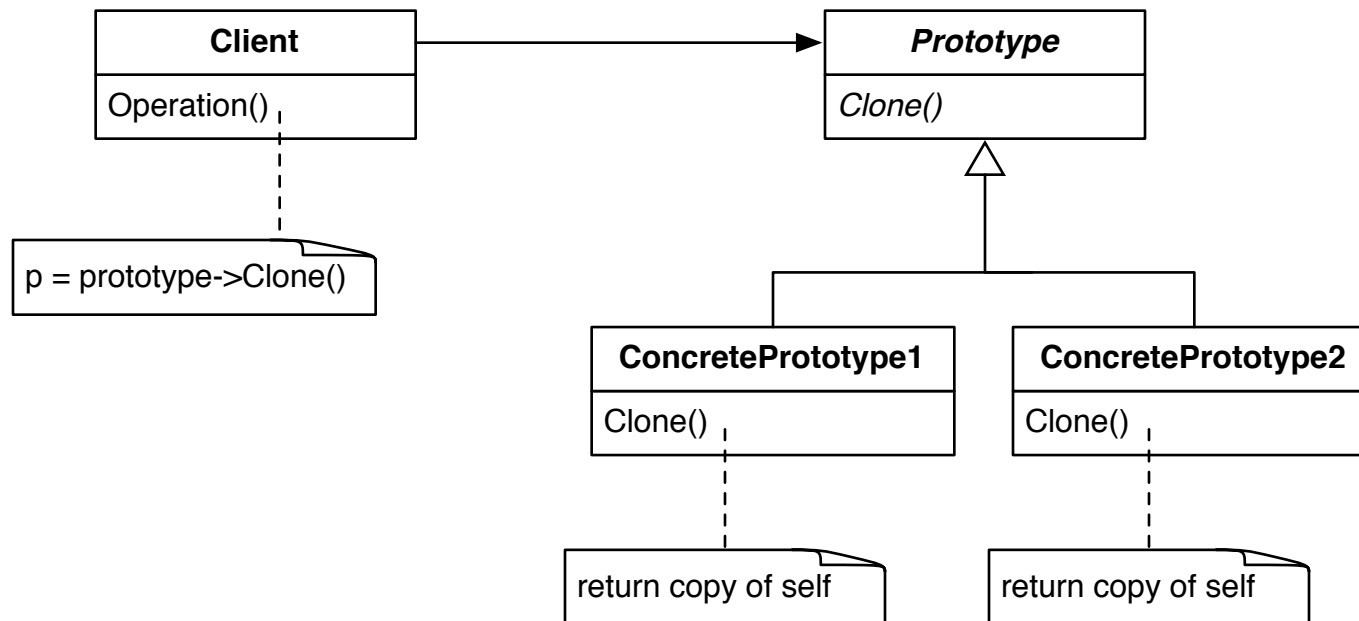
Prototype

- **On l'utilise quand :**
 - la classe à instancier n'est connue qu'au runtime
 - pour éviter d'avoir une hiérarchie de Factories qui gère en parallèle une hiérarchie de produits
 - les instances d'une classes peuvent avoir un état parmi plusieurs états identifiés
 - la création d'une instance est complexe ou consommatrice en temps

Prototype

- **Principe de fonctionnement**
 - Plutôt que créer plusieurs instances
 - Le client demande au prototype (la première instance) de se « cloner »
 - On n'a plus qu'à modifier le clone de façon appropriée

Prototype



Prototype

- **Conséquences**

- On peut ajouter ou retirer des produits à l'exécution (simplement si on connaît une instance qui servira de prototype)
- On peut spécifier des nouveaux objets en changeant les valeurs du clone
- On peut construire des objets composite plus complexe par assemblage de « clones » dans une librairie de prototypes
- Permet de s'affranchir d'un ensemble de Factory Method

Singleton

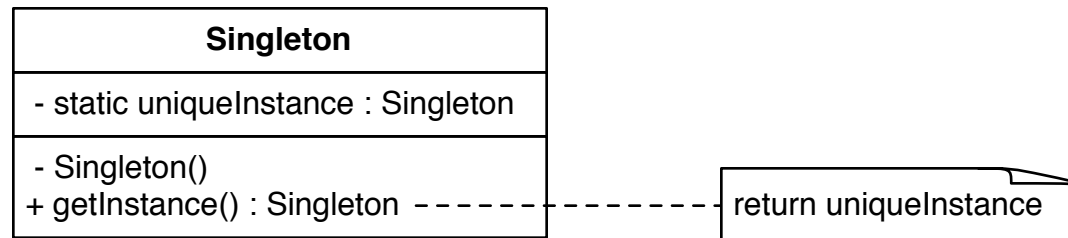
- **On l'utilise quand :**
 - il doit n'y avoir qu'une seule instance d'une classe
 - cette instance unique doit être accessible de manière connue
 - cette instance peut être sous-classée et utilisée par les clients sans qu'ils aient à modifier leur code

Singleton

- **Implémentation**

- La classe contient une méthode qui crée une instance, uniquement s'il n'en existe pas encore ; sinon, elle renvoie une référence vers l'instance qui existe déjà
- Attention :
 - penser à mettre les constructeurs privés ou protégés
 - contextes multi-threads
 - Incompatible avec le pattern Prototype

Singleton



Singleton

- **Conséquences**
 - On a le contrôle sur l'instance
 - On peut éventuellement contrôler plusieurs instances
 - On peut raffiner (en sous-classant)

Les patterns de structure

Les patterns de structure

- **Ce qu'ils permettent**
 - La composition, l'assemblage de classes et d'objets pour former des structures complexes
- **Comment ils y parviennent**
 - En utilisant l'héritage pour composer des interfaces ou des implémentations (**patterns de classe**)
 - En composant des objets pour construire des nouvelles fonctionnalités (**patterns d'objet**)

Les patterns de structure

| | |
|------------------|--|
| Adapter | Rendre un objet conforme à une autre interface |
| Bridge | Lier une abstraction à une implémentation |
| Composite | Objets composés d'objets primitifs ou de d'objets composites |
| Decorator | Ajouter des possibilités à une objet |
| Facade | Cacher une structure complexe |
| Flyweight | Limiter la prolifération de petits objets similaires |
| Proxy | Un objet en masque un autre |

Adapter

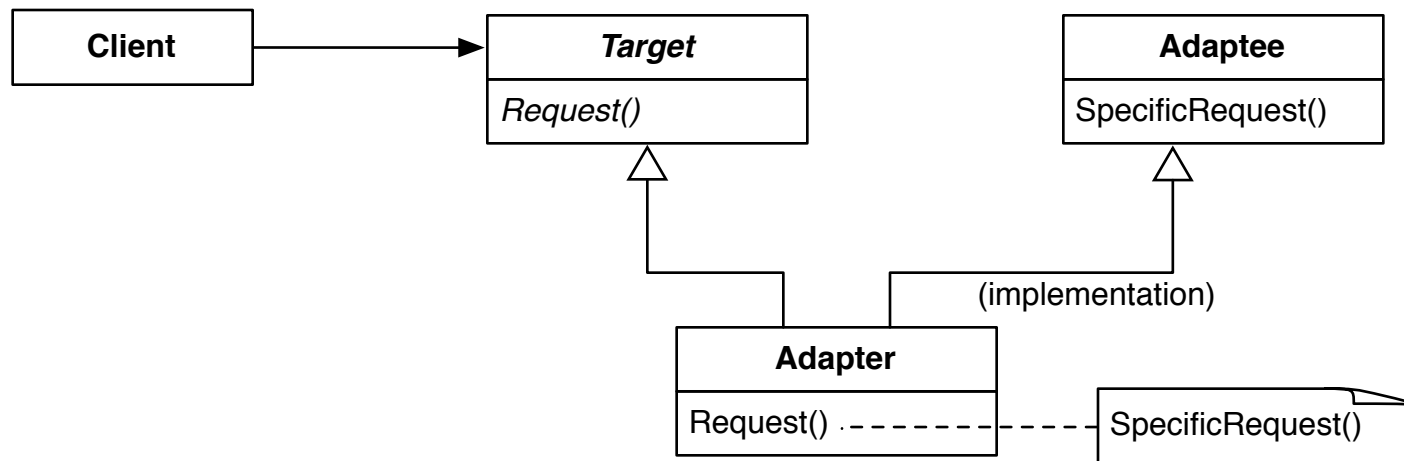
- Ou **Wrapper**
- **On l'utilise quand :**
 - on veut utiliser une classe existante, mais qui ne dispose pas de l'interface voulue
 - on veut créer une classe qui soit réutilisable par d'autres classes qui ne disposeront pas d'interface compatible
 - on veut utiliser plusieurs sous-classes mais qui ne disposent pas d'interface commune

Adapter

- **Principe de fonctionnement**
 - L'Adapter est conforme à l'interface de destination (Target)
 - Le client appelle les opérations sur une instance de cet Adapter
 - L'instance de l'Adapter contient une référence vers l'instance de la classe à adapter (Adaptee)
 - L'instance de l'Adapter appelle les opérations nécessaires sur l'instance de l'Adaptee

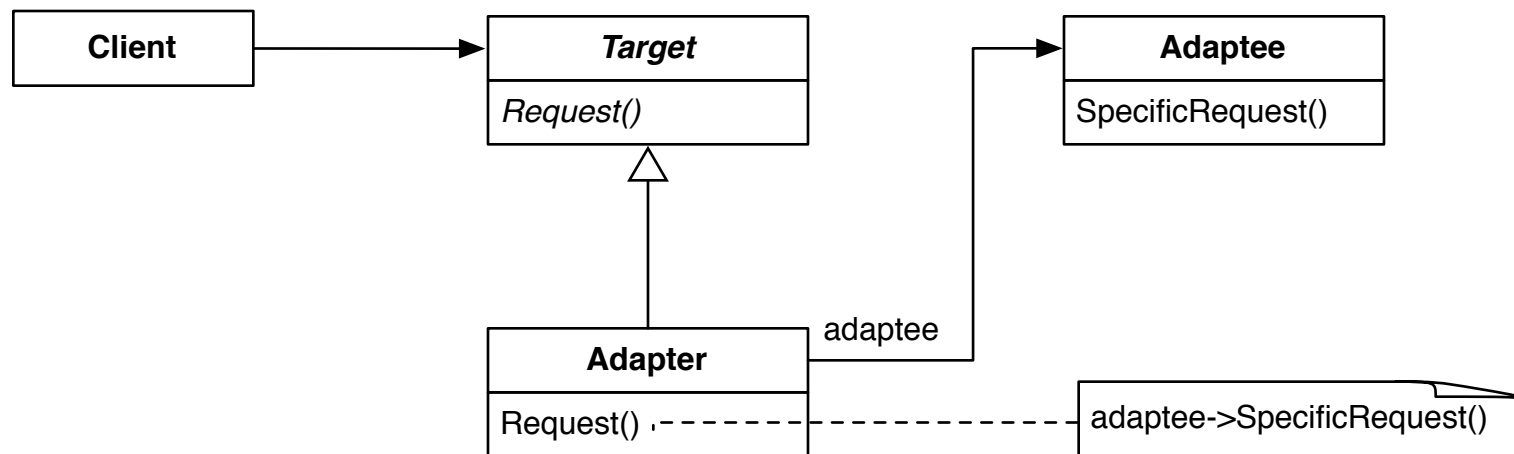
Adapter (class)

Version héritage multiple



Adapter (object)

Version composite



Adapter

- **Conséquences**

- La charge de travail de « traduction » dépend du gap entre les interfaces de l'Adaptee et du Target

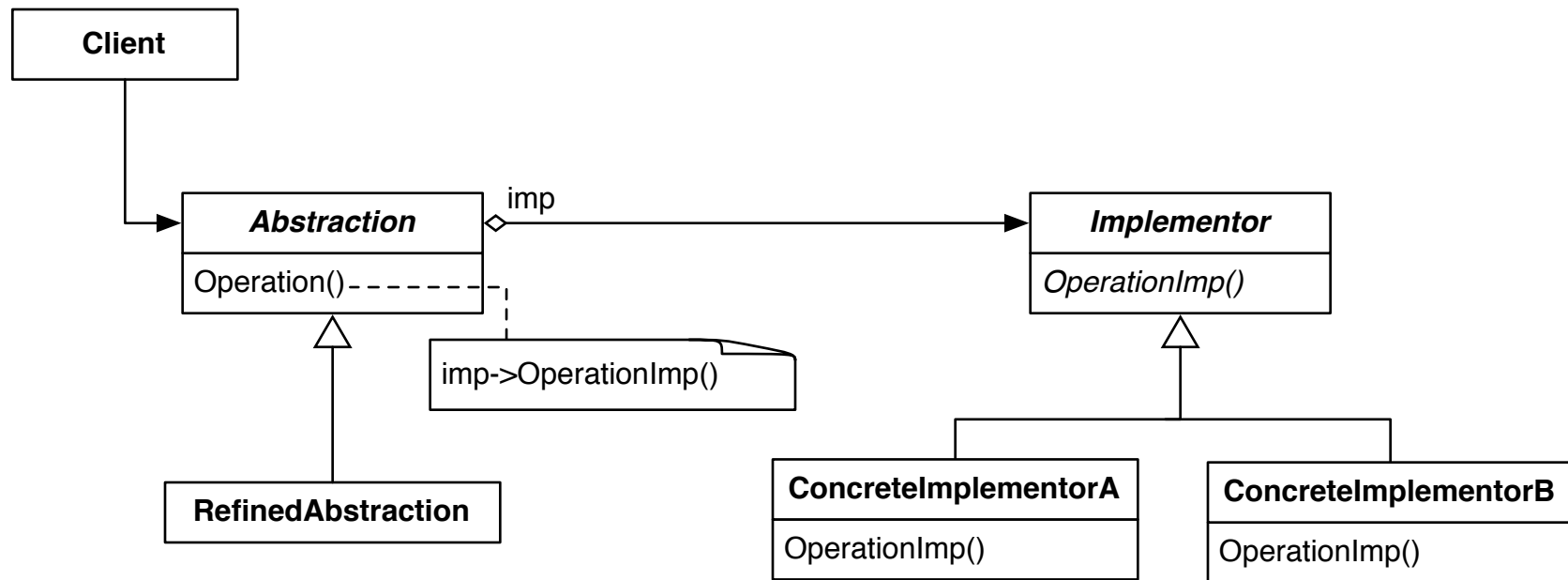
Bridge

- **On l'utilise quand :**
 - pour éviter un lien permanent entre l'abstraction et l'implémentation
(par ex., pour pouvoir choisir l'implémentation au moment de l'exécution)
 - l'abstraction et l'implémentation sont toutes deux susceptibles d'être sous-classées
 - les changements sur l'implémentation n'ont aucun impact sur le client (pas de recompilation)
 - on veut partager une même implémentation entre plusieurs objets, sans que cela soit visible pour le client

Bridge

- **Principe de fonctionnement**
 - L'abstraction contient une référence vers l'implémentation
 - L'abstraction forward les requêtes du client à l'implémentation

Bridge



Bridge

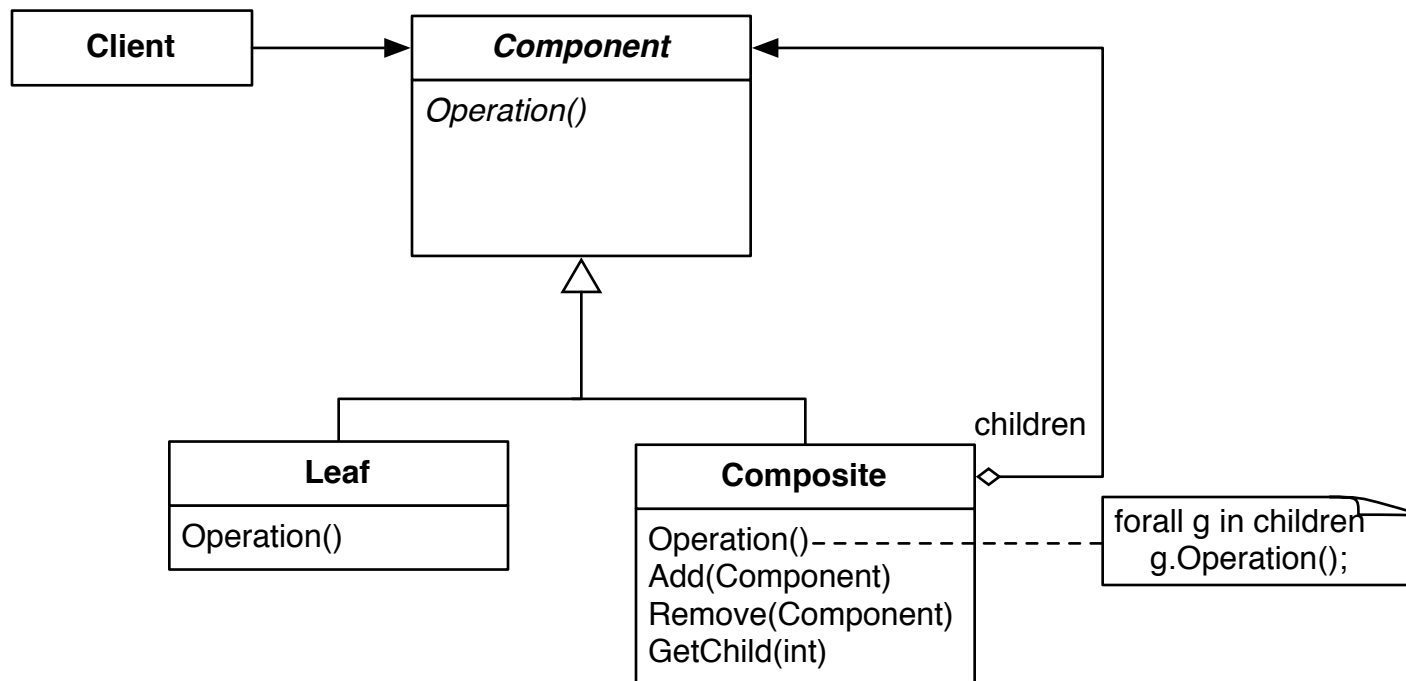
- **Conséquences**

- Découplage interface / implémentation
 - Force à mieux structurer
 - Assure compatibilité avec clients (pas besoin de recompiler après chgt de l'implémentation)
- Structure plus extensible
- Cache tous les détails d'implémentation aux clients

Composite

- **On l'utilise quand :**
 - on veut représenter des hiérarchies d'objets dans une structure arborescente
 - on veut que le client ne puisse pas distinguer un objet individuel d'un objet composé ; pour lui, tout est objet composé

Composite



Composite

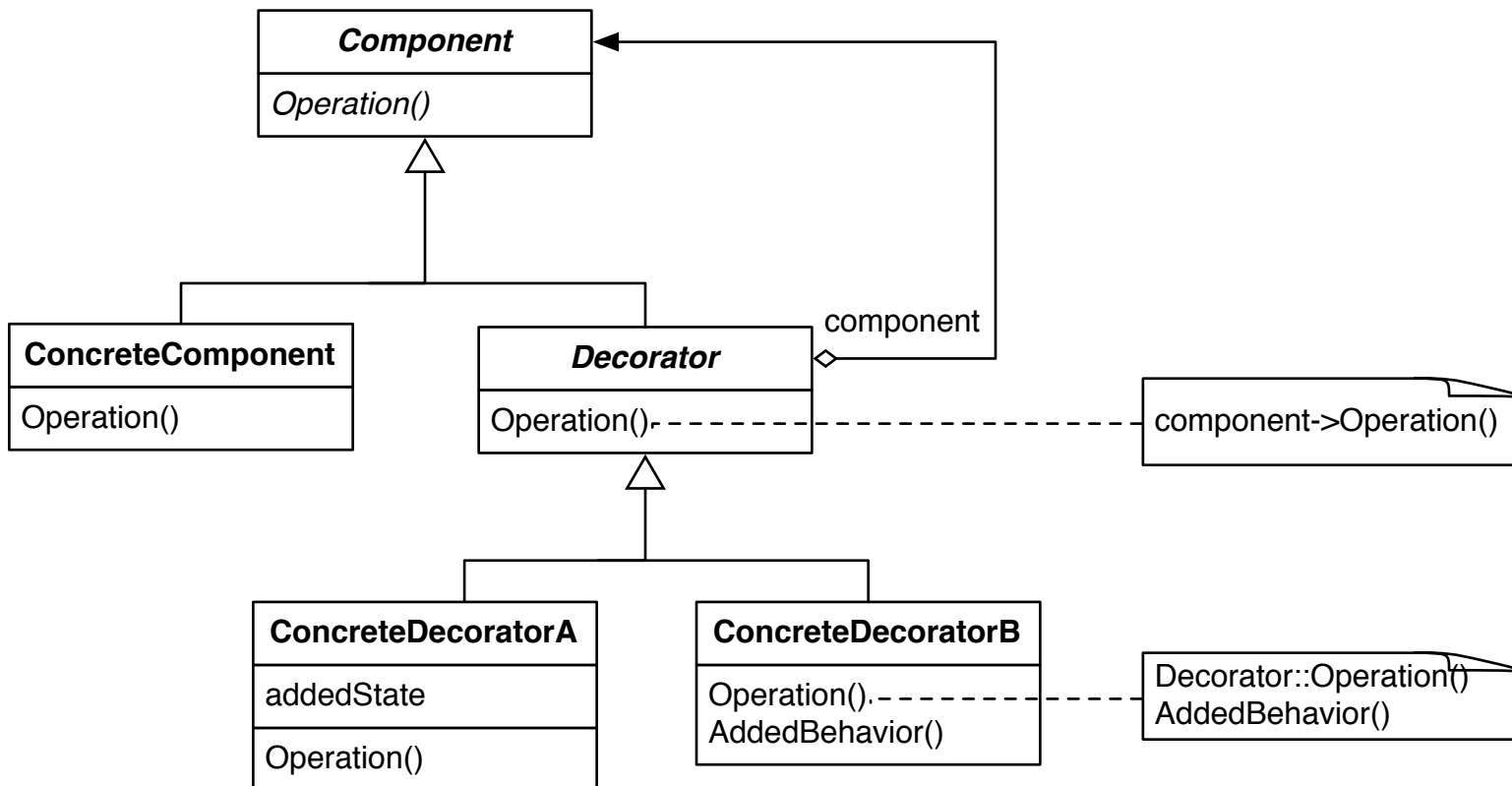
- **Conséquences**

- Simplifie le code du client
- Les répercussions aux composants sont transmises par chaque composé
- Facilite l'ajout de nouveaux types de composants
 - ➔ donc plus difficile de restreindre l'ajout de nouveaux types de composants...
- Difficile de restreindre le type des composants d'un composite

Decorator

- **On l'utilise quand :**
 - on veut rajouter des capacités à des objets de manière dynamique et transparente
 - pour des responsabilités dont on peut se passer
 - l'ajout de capacités par héritage n'est pas envisageable
- **Principe de fonctionnement**
 - Le composant à décorer avec de nouvelles fonctionnalités est passé en paramètre au Decorator
 - Le Decorator appelle les méthodes de décoration ainsi que celles de l'objet décoré

Decorator



Decorator

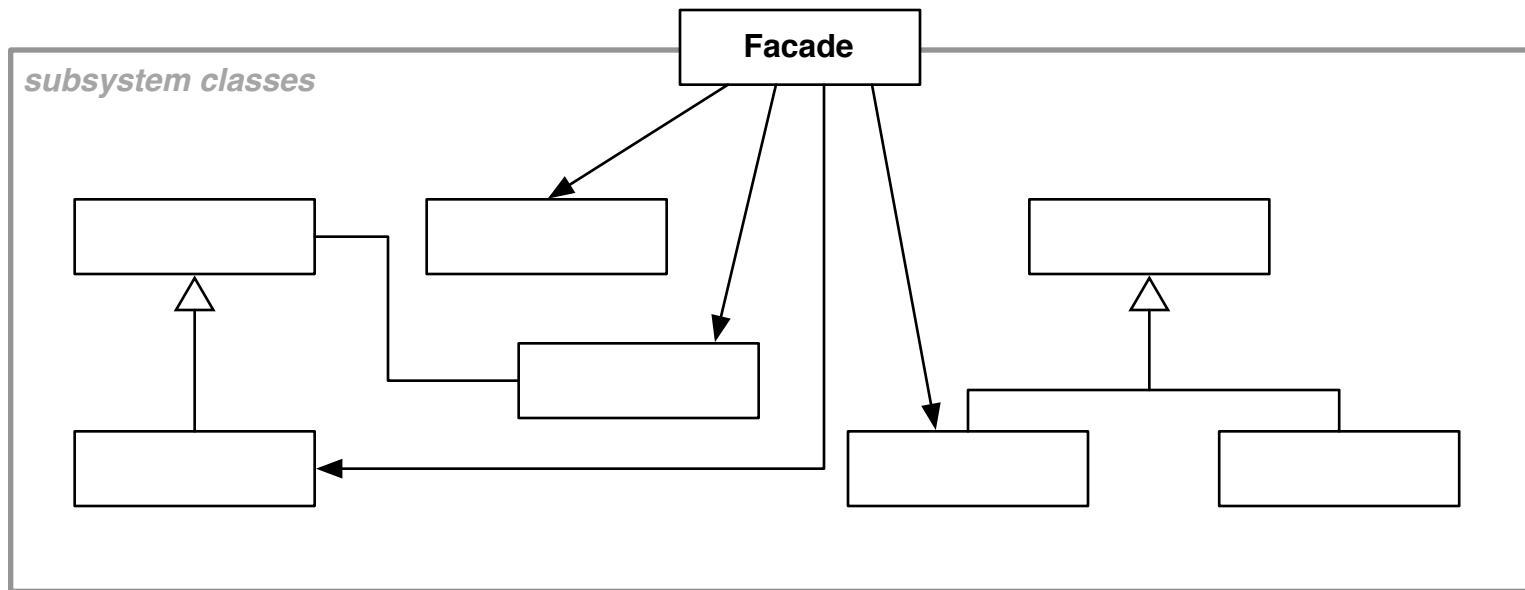
- **Conséquences**

- Plus de souplesse que l'héritage
 - ajout/retrait à l'exécution
 - possibilité d'ajouter 2x la même propriété
 - on peut mixer/mélanger les propriétés
- Lecture / déboguage plus difficile
- Attention, l'objet décoré \neq l'objet concret

Facade

- **On l'utilise quand :**
 - on veut fournir une interface simple à un système complexe
 - pour réduire les dépendances entre clients et les classes d'implémentation d'une abstraction
 - on veut construire un système en couches
(une façade par couche)

Facade



Facade

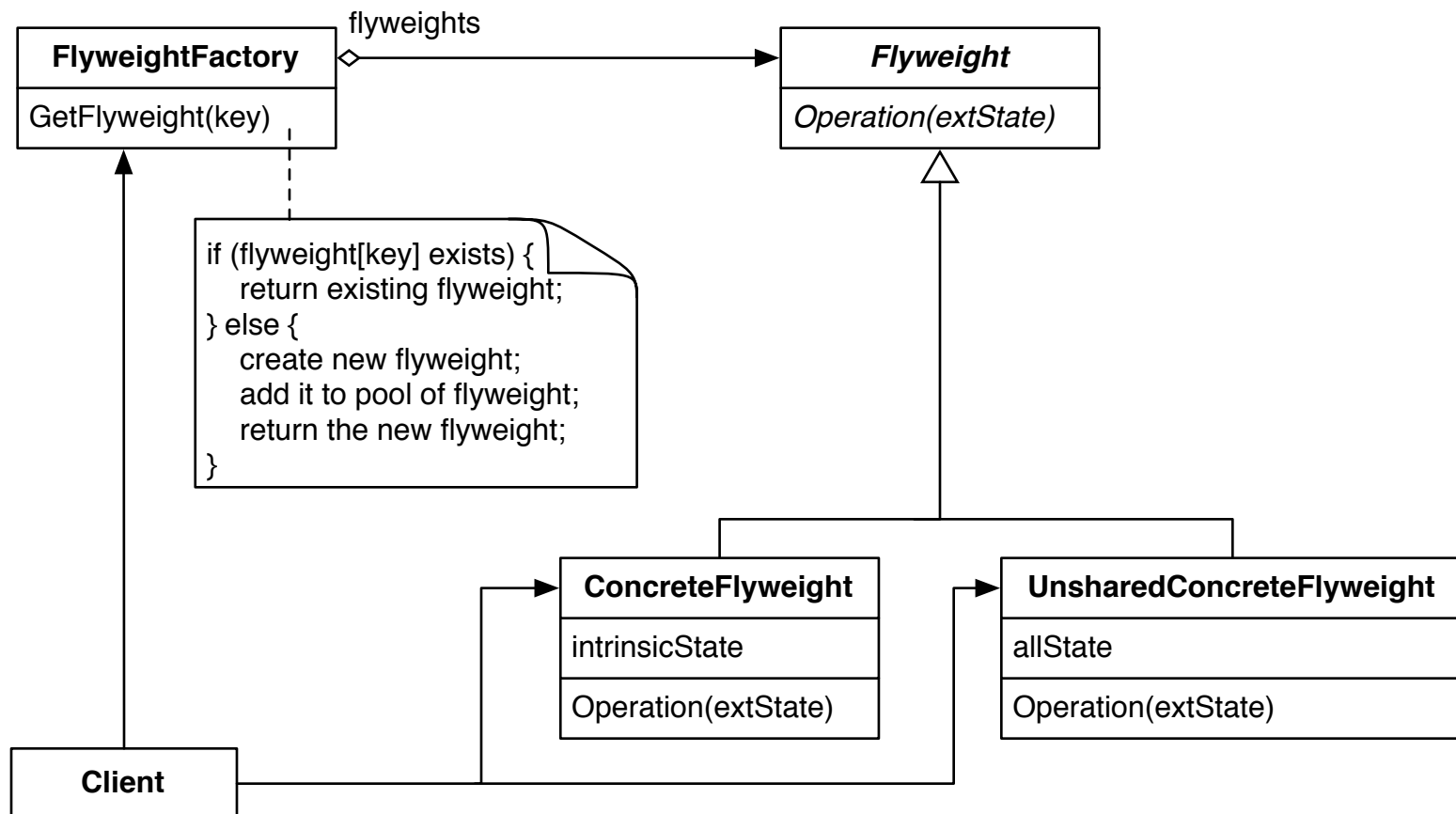
- **Conséquences**

- Simplifie l'utilisation du système par le client
- Favorise un faible couplage entre le système et ses clients
 - ➔ on peut faire évoluer le système sans toucher aux clients
- Mais rien n'empêche un client d'accéder aux sous-classes du système si besoin

Flyweight

- Ou **poids-mouche**
- **On l'utilise quand :**
 - on utilise un grand nombre de petits objets
 - mais qu'il serait trop coûteux en mémoire s'il fallait instancier tous ces objets
 - l'état des objets peut être externalisé
(les objets sont semblables à quelques paramètres près)
 - des groupes d'objets peuvent être remplacés par quelques objets partagés une fois que leur état est externalisé
 - l'application ne dépend pas de l'identité des objets

Flyweight



Flyweight

- **Conséquences**

- Le nombre d'instances diminue
- Mais
 - La gestion, le transfert, etc. des états externalisés engendre un coût
- Plus les objets flyweight sont partagés, plus on économise de la place

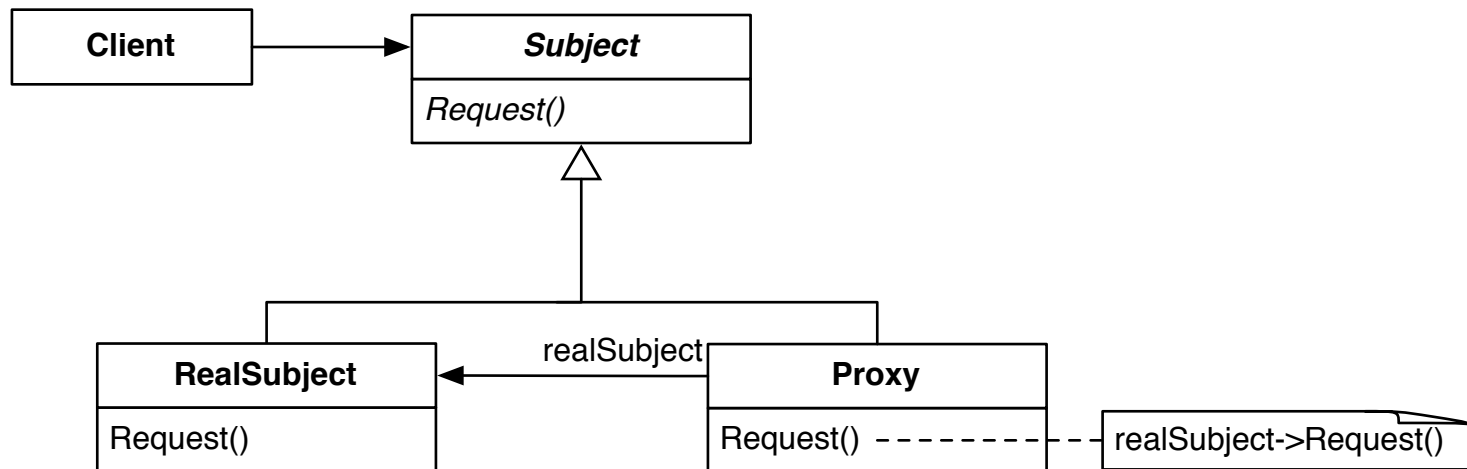
Proxy

- **On l'utilise quand :**
 - on a besoin de plus qu'une simple référence vers un objet
 - on veut un objet local représentant une instance distante (remote proxy)
 - on veut charger les objets à la demande (virtual proxy)
 - on veut contrôler l'accès à un objet (protection proxy)
 - on veut effectuer des actions lorsqu'un client accède à l'objet (smart reference)

Proxy

- **Principe de fonctionnement**
 - Le proxy et l'objet partagent la même interface
 - Le proxy se substitue à l'objet qu'il gère
 - L'utilisation du proxy ajoute une indirection à l'utilisation de la classe à substituer

Proxy



Proxy

- **Conséquences**

- Permet de cacher le fait qu'un objet est distant
- Permet de réaliser des optimisations (gestion de cache, copy-on-write, etc.)

Les patterns de comportement

Les patterns de comportement

- **Ce qu'ils permettent**
 - Décrire des algorithmes, des comportements et des formes de communication entre objets/classes
 - Se concentrer sur la façon dont les objets sont reliés
- **Comment ils y parviennent**
 - En utilisant l'héritage pour distribuer le comportement sur les classes (**patterns de classe**)
 - En composant des objets (**patterns d'objet**)

Les patterns de comportement (1)

| | |
|----------------|--|
| Chain of Resp. | |
| Command | |
| Interpreter | |
| Iterator | |
| Mediator | |
| Memento | |

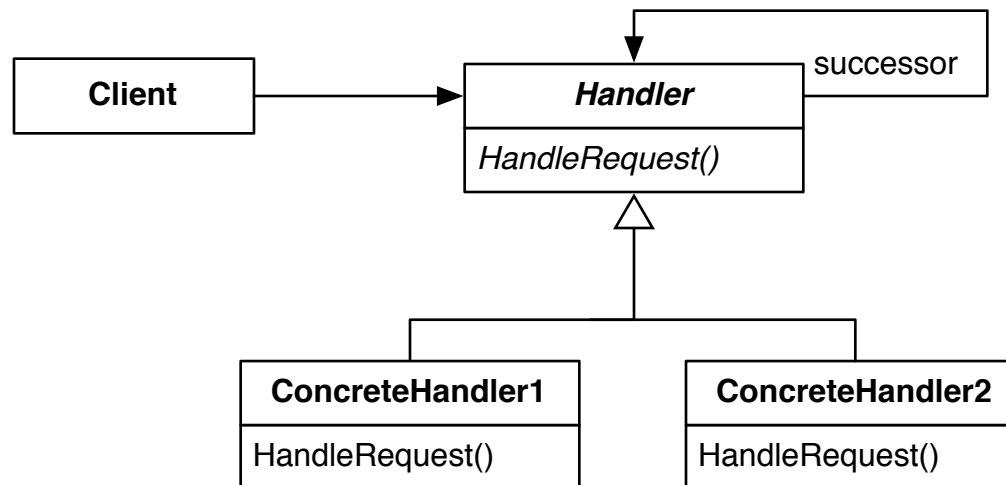
Les patterns de comportement (2)

| | |
|--------------------|--|
| Observer | |
| State | |
| Strategy | |
| Template Method | |
| Visitor | |

Chain of Responsibility

- **On l'utilise quand :**
 - plusieurs objets peuvent traiter une requête, mais on ne sait pas *a priori* lequel va effectivement le faire
 - les objets qui peuvent traiter la requête sont spécifiés dynamiquement
- **Principe de fonctionnement**
 - La requête envoyée par le client est propagée le long de la chaîne d'objets susceptibles de la traiter jusqu'à ce que l'un d'entre eux prenne la responsabilité de la traiter

Chain of Responsibility



Chain of Responsibility

- **Conséquences**

- Réduit le couplage (on ne connaît pas qui traite la requête)
- Ajouter/Modifier des responsabilités devient plus flexible
- Mais le traitement effectif de la requête n'est pas garanti

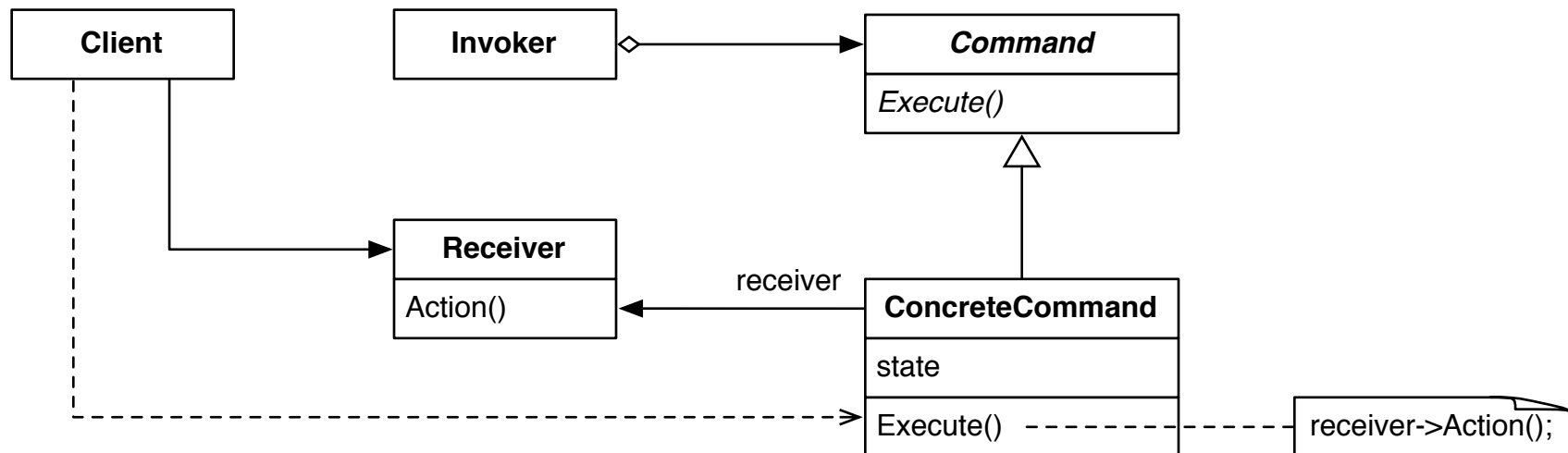
Command

- **On l'utilise quand :**
 - on veut spécifier, stocker, exécuter des actions à des moments différents
 - on veut supporter le « undo »
(on stocke les commandes exécutées et l'état des objets)
 - après un crash, pour récupérer le travail effectué
 - gérer des transactions (ensemble d'opérations primitives)

Command

- **Principe de fonctionnement**
 - Le **client** instancie la **commande** et lui indique l'objet cible (le **receiver** : document, application, ...)
 - L'**Invoker** (menu item, par ex) demande l'exécution de la commande

Command



Command

- **Conséquences**

- Découple l'objet qui invoque de celui qui effectue
- Les commandes sont instances de classes (peuvent être manipulées, étendues, etc.)
- On peut assembler des commandes
- On peut facilement ajouter des nouvelles commandes

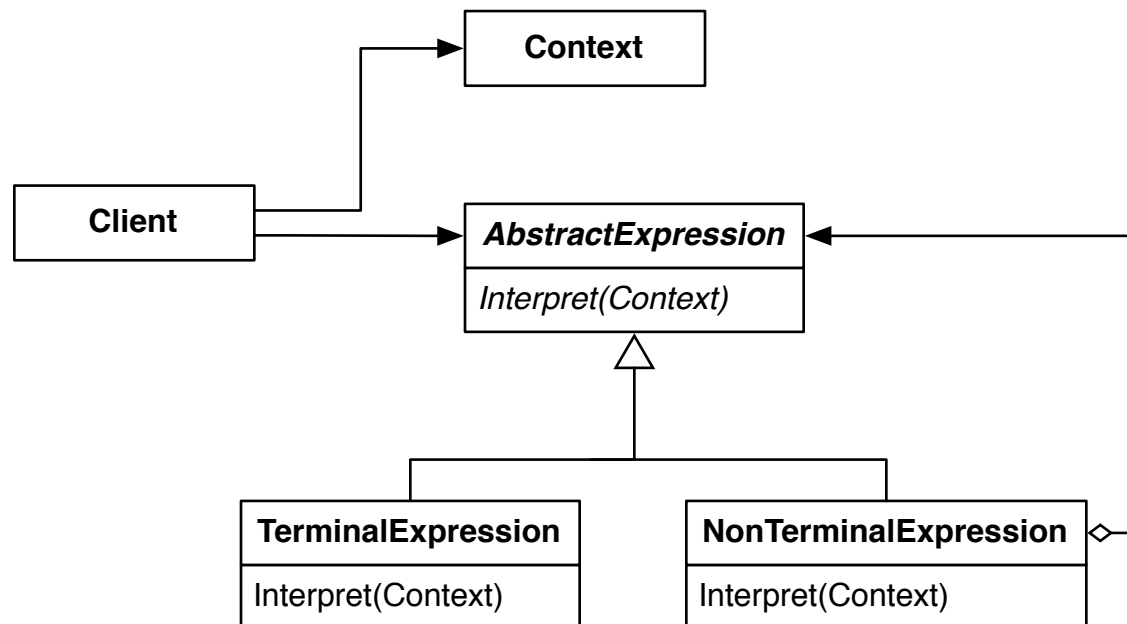
Interpreter

- **On l'utilise quand :**
 - pour interpréter/implémenter un langage spécialisé dont la grammaire est simple
 - l'efficacité n'est pas un critère primordial
 - Ex: interprétation d'un langage de script, d'une requête SQL, etc.

Interpreter

- **Principe de fonctionnement**
 - Une instance d'expression terminale pour chaque symbole (opérandes)
 - Une classe d'expression non-terminale pour chaque règle de la grammaire (opérateurs)
 - Le client construit un arbre représentant une expression dans le langage décrit par la grammaire, et déclenche le parcourt de l'arbre

Interpreter



Interpreter

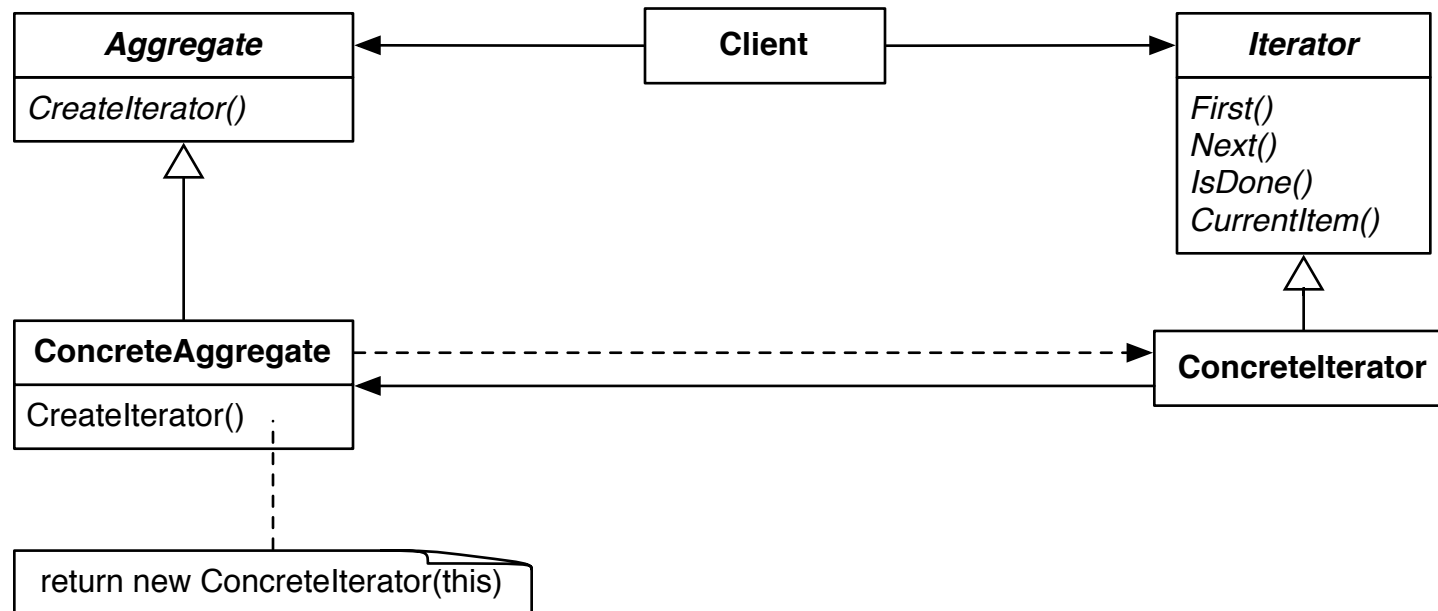
- **Conséquences**

- Il est facile de changer ou d'étendre la grammaire
- L'implémentation de la grammaire est facile
- Les grammaires complexes sont difficiles à gérer et à maintenir (→ autres techniques)
- On peut ajouter des nouvelles façons d'interpréter les expressions

Iterator

- **On l'utilise quand on veut :**
 - accéder aux composants d'un objet sans exposer sa représentation interne
 - implémenter plusieurs manières de parcourir des objets composés
 - fournir une interface uniforme pour parcourir différentes structures

Iterator



Iterator

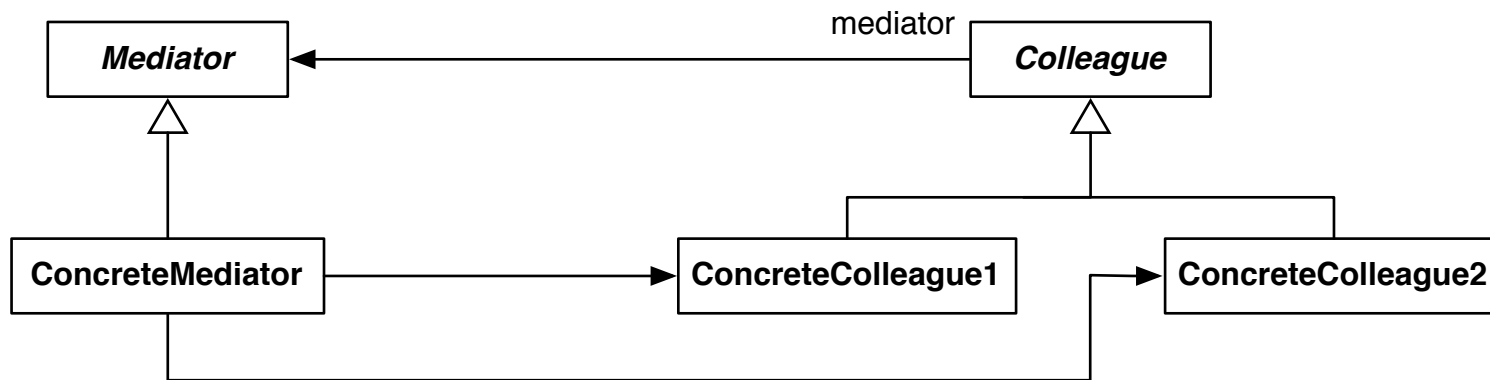
- **Conséquences**

- On peut implémenter plusieurs manières de traverser un agrégat (plusieurs sortes de parsers)
- Ça simplifie l'interface de l'agrégat
- On peut avoir plusieurs parcours simultanés du même agrégat (chaque itérateur mémorise sa position dans l'agrégat)

Mediator

- **On l'utilise quand :**
 - de nombreux objets doivent communiquer ensemble
 - réutiliser un objet est difficile à cause de ses nombreuses relations avec d'autres objets
- **Principe de fonctionnement**
 - Le médiateur est la seule classe ayant connaissance des interfaces des autres classes
 - ➔ une classe qui veut interagir avec une autre doit passer par le médiateur

Mediator



Mediator

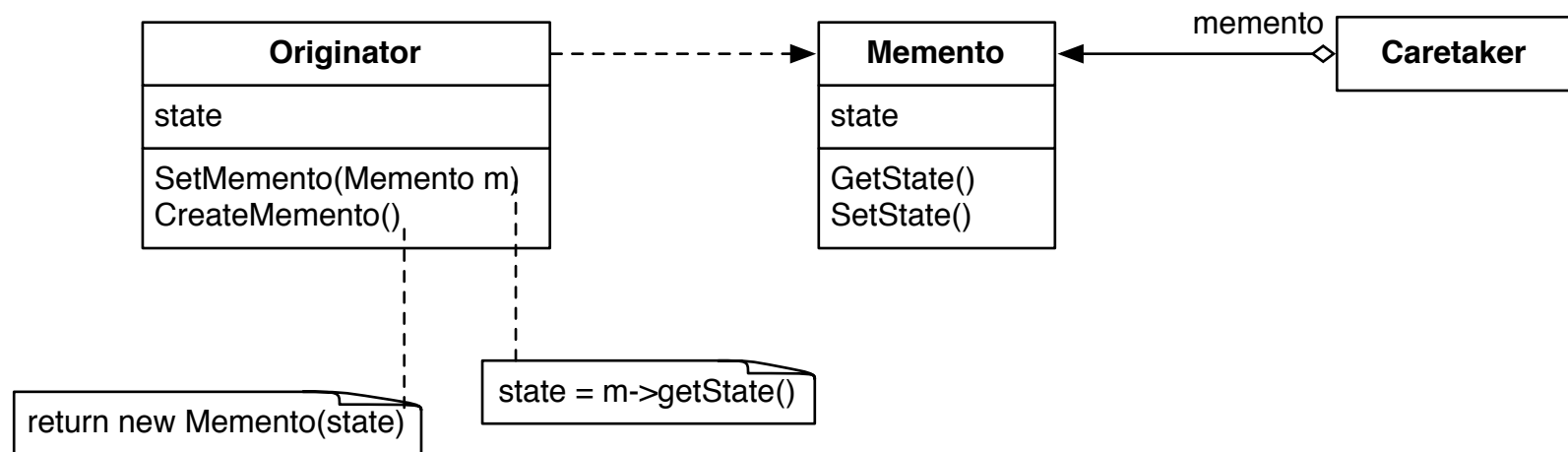
- **Conséquences**

- Spécialisation des objets plus difficile (comportement centralisé dans le médiateur)
- Les classes sont découplées
- Protocole de communication simplifié
- Contrôle centralisé

Memento

- **On l'utilise quand :**
 - on veut sauvegarder l'état de l'objet pour éventuellement pouvoir le restaurer (undo) et
 - on veut conserver le principe d'encapsulation
- **Principe de fonctionnement**
 - Le créateur est l'objet dont on veut sauvegarder l'état
 - Le gardien demande au créateur l'objet memento
 - Le memento, créé par le créateur, est retourné au gardien
 - L'objet memento est opaque (le gardien ne doit pas le modifier)

Memento



Memento

- **Conséquences**

- Le principe d'encapsulation est préservé
- Un memento opère sur un seul objet
- L'utilisation de mementos peut être coûteuse (si beaucoup d'infos, et souvent)
- Le gardien ne doit pas toucher au memento (difficile à assurer dans certains langages)

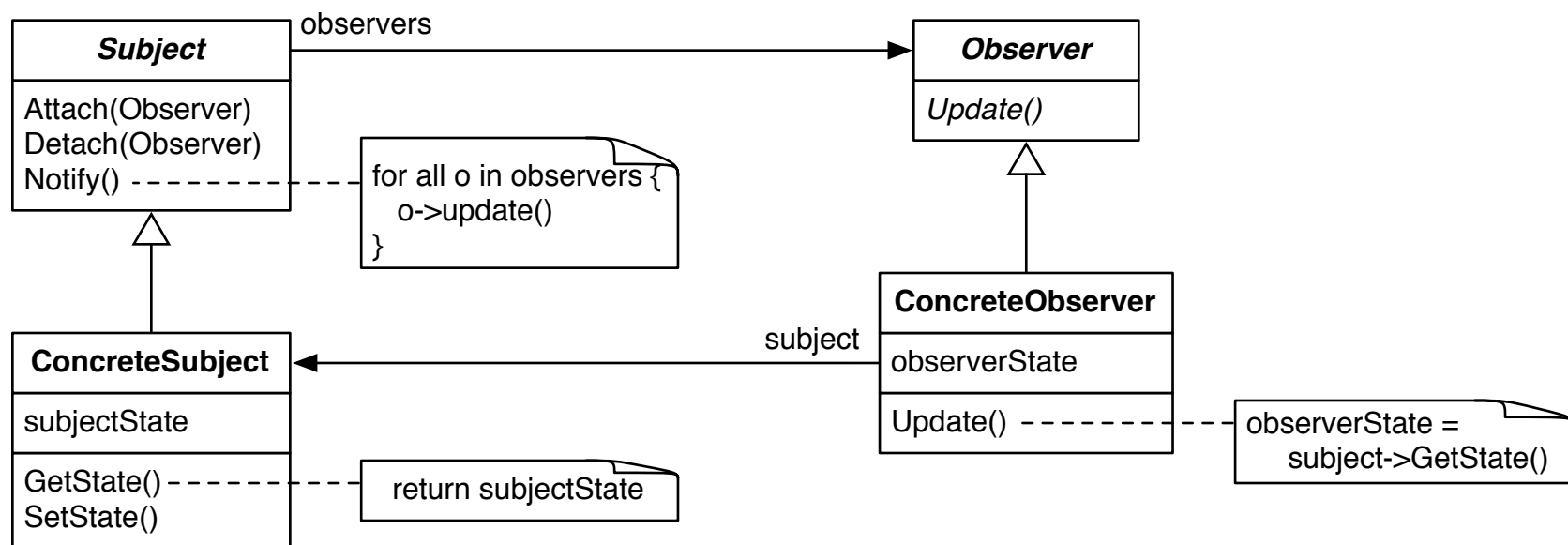
Observer

- **On l'utilise quand :**
 - une abstraction a plusieurs aspects dépendants
 - un changement sur un objet a des répercussions sur d'autres
 - un objet doit en prévenir d'autres sans pour autant les connaître

Observer

- Un **observé** (*subject*)
 - N'importe quelle instance qui est modifiée
- Les **observateurs** (*observer*) seront notifiés
 - à la modification de l'observé
 - synchrone
- Ajout et retrait dynamique d'observateurs

Observer




```
public interface Observable {  
    public void addObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
  
    public int getState();  
    public void setState(int state);  
}
```

```
public interface Observer {  
    public void update(Observable o);  
}
```

```
public class ConcreteObservable implements Observable {  
  
    private Collection<Observer> observers = new ...  
    private int state = 0;  
  
    public void addObserver(Observer o) {  
        observers.add(o);  
    }  
    public void removeObserver(Observer o) {  
        observers.remove(o);  
    }  
    public void notifyObservers(){  
        for(Observer obs : observers)  
            obs.update();  
    }  
  
    public int getState() { return this.state; }  
    public void setState(int state) {  
        this.state = state;  
        notifyObservers();  
    }  
}
```

```
Observable o = new ConcreteObservable();  
Observer obs = new ConcreteObserver();
```

```
o.addObserver(obs);
```

```
o.setState(5);
```

```
    // obs est « réveillé »,  
    // notifié par le changement sur o
```

Observer

- Ce patron est présent dans de nombreuses API
 - Ex: dans Swing
 - les ActionListener, MouseListener, etc.. sont des observers
 - Démo.

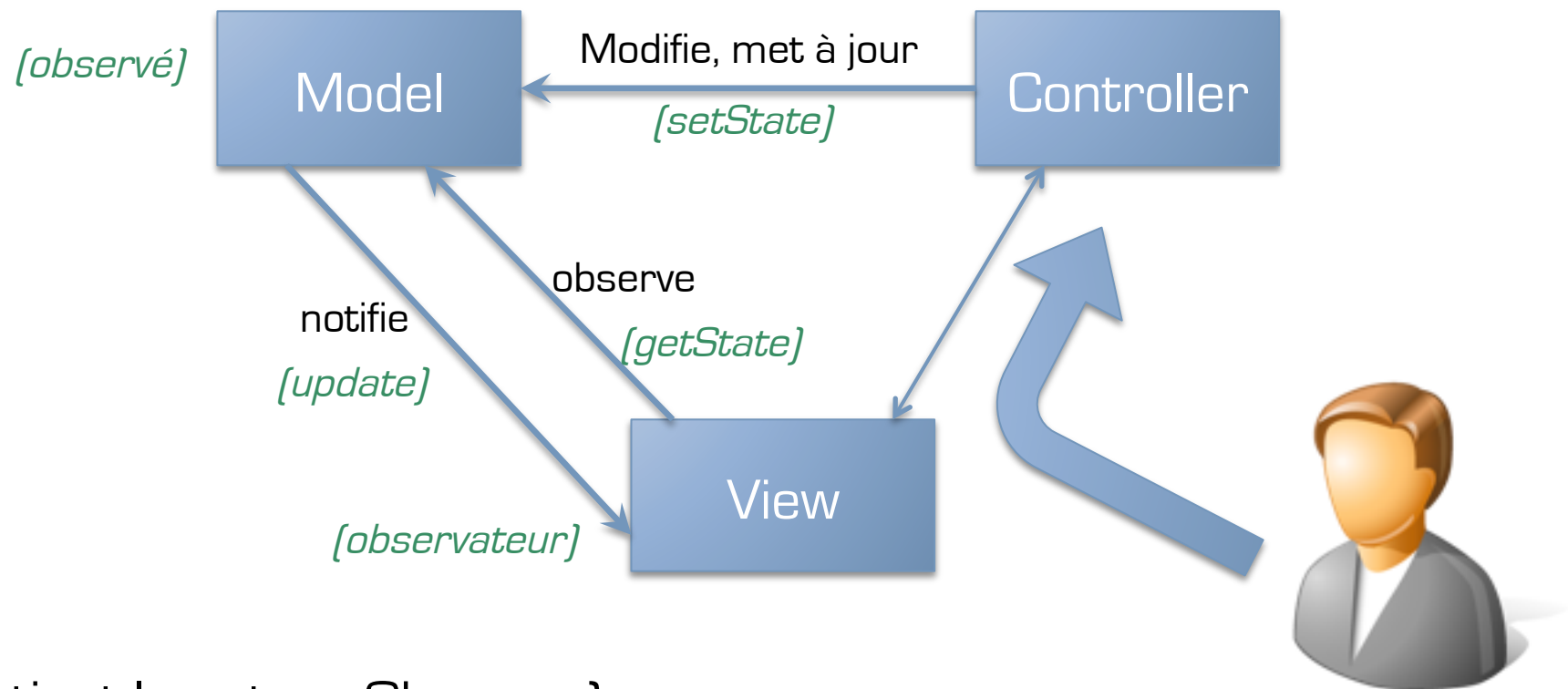
Observer

- **Conséquences**
 - Couplage abstrait et minimal entre Subject et Observer
 - Moyen de broadcast
 - Mises à jour inattendues

Modèle-Vue-Contrôleur (MVC)

- Le patron **MVC** permet de **séparer** :
 - Le **modèle** : la donnée à représenter,
 - La **vue** : la représentation externe,
 - et le **contrôleur** : l'agent responsable de la coordination
- Architecture claire, lisible, facile à maintenir

Modèle-Vue-Contrôleur (MVC)



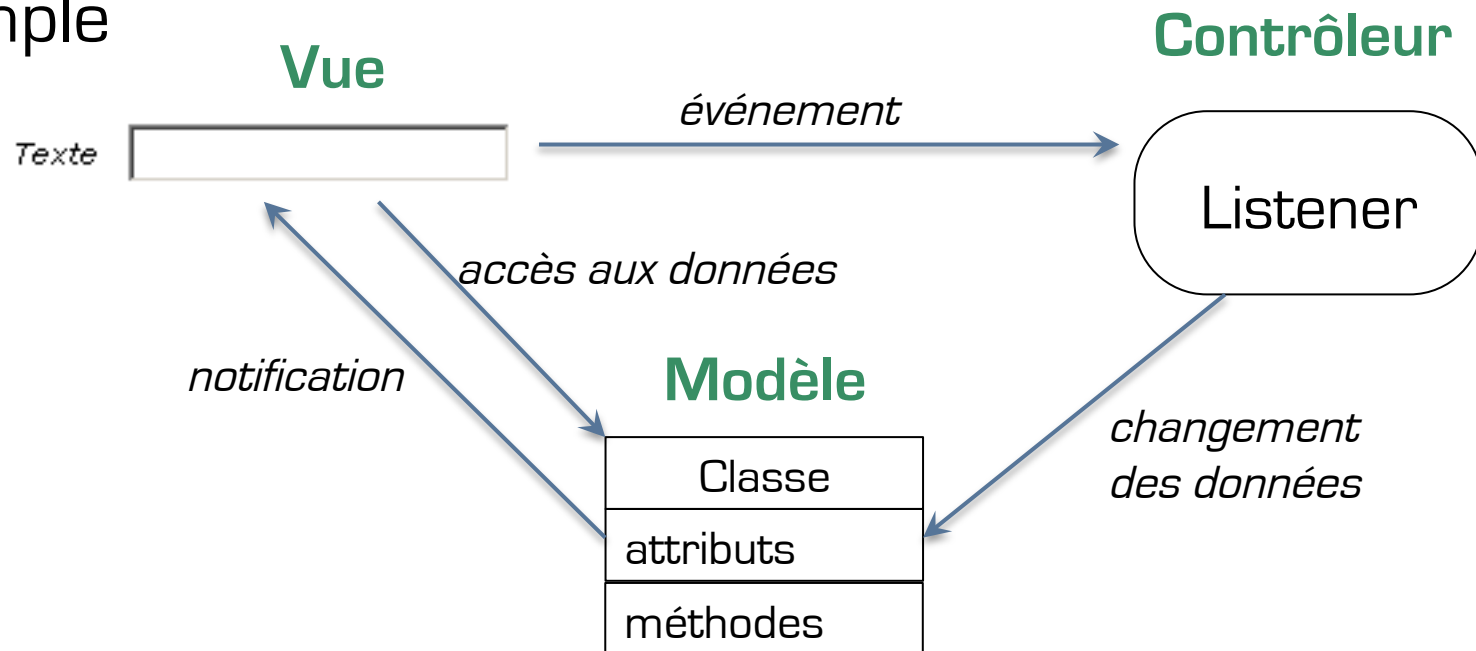
[Contient le patron Observer]

Modèle-Vue-Contrôleur (MVC)

- Fonctionnement
 - Abonnements entre vue(s), modèle(s), contrôleur(s)
 - Le contrôleur
 - Reçoit les événements
 - Interroge éventuellement la vue pour savoir à quoi il correspond
 - Puis avertit le modèle des événements pour savoir quoi faire
 - Le modèle
 - Met à jour la vue, qui sait quelles sont ses parties à mettre à jour

Modèle-Vue-Contrôleur (MVC)

- Exemple

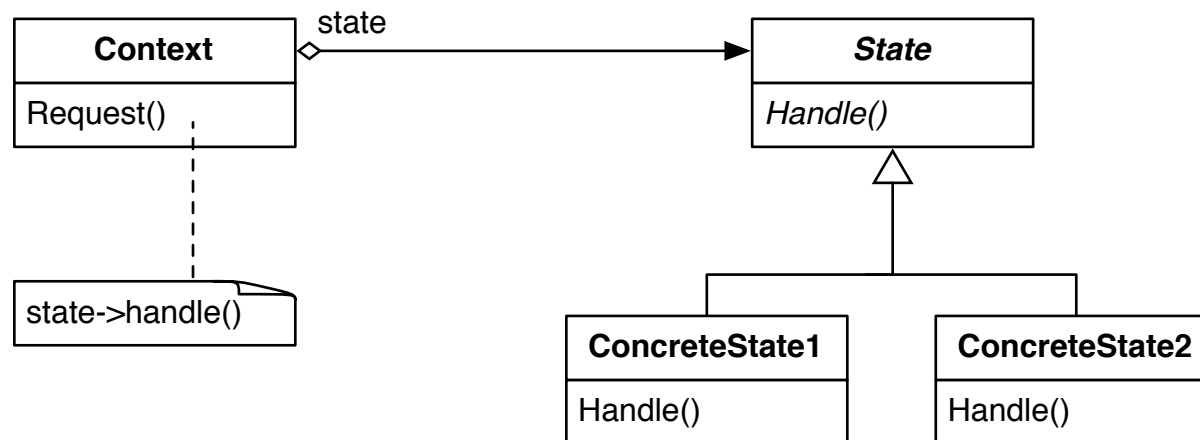


- Le modèle MVC peut s'appliquer de façon récursive

State

- **On l'utilise quand :**
 - le comportement d'un objet dépend de son état (qui peut changer à l'exécution)
 - plusieurs opérations contiennent les mêmes structures conditionnelles
- **Principe de fonctionnement :**
 - Les différents états, avec leurs comportements associés, sont externalisés

State



State

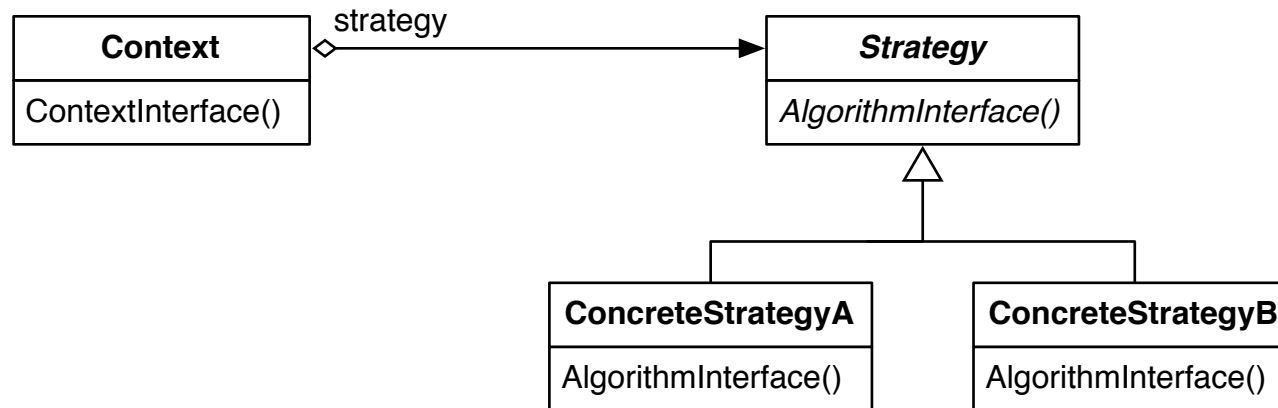
- **Conséquences**

- Les comportements spécifiques à chaque état sont localisés
- Les transitions entre états sont rendues explicites
- Les objets état peuvent être partagés (*flyweight*)

Strategy

- **On l'utilise quand :**
 - on a besoin de plusieurs variantes d'un algorithme
 - un algorithme utilise des données que le client n'est pas censé connaître
 - plusieurs classes se distinguent uniquement par leur comportement

Strategy



Strategy

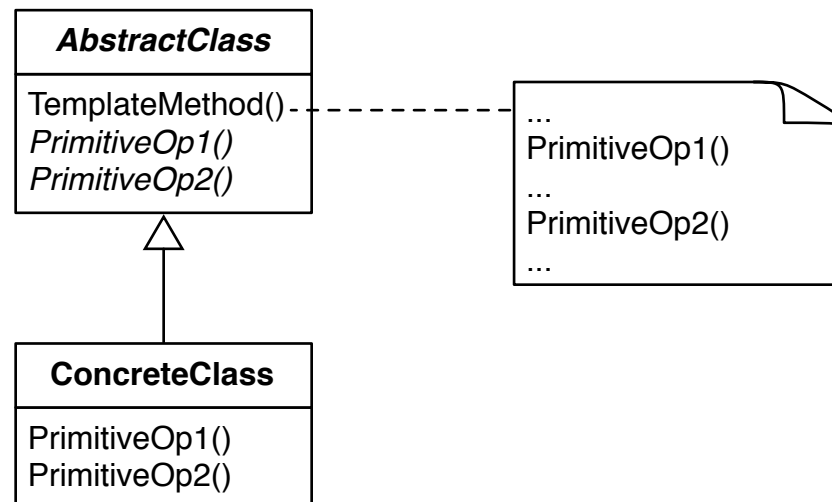
- **Conséquences**

- Familles d'algorithmes
- Alternative à la spécialisation
- Le client a le choix entre plusieurs implémentations
- Le client doit connaître à l'avance les différentes stratégies
- Nombre d'objets plus important (solution : les partager entre plusieurs contexte en éliminant l'état)

Template Method

- **On l'utilise quand :**
 - pour implémenter les invariants
 - pour éviter la duplication de code (factorisation des parties communes)
 - pour contrôler la spécialisation, fixer des comportements standards qui doivent être partagés par les sous-classes
- **Principe de fonctionnement**
 - Définit le squelette d'un algorithme à l'aide d'opérations *abstraites* dont le comportement concret se trouvera dans les sous-classes, qui implémenteront ces opérations

Template Method



Template Method

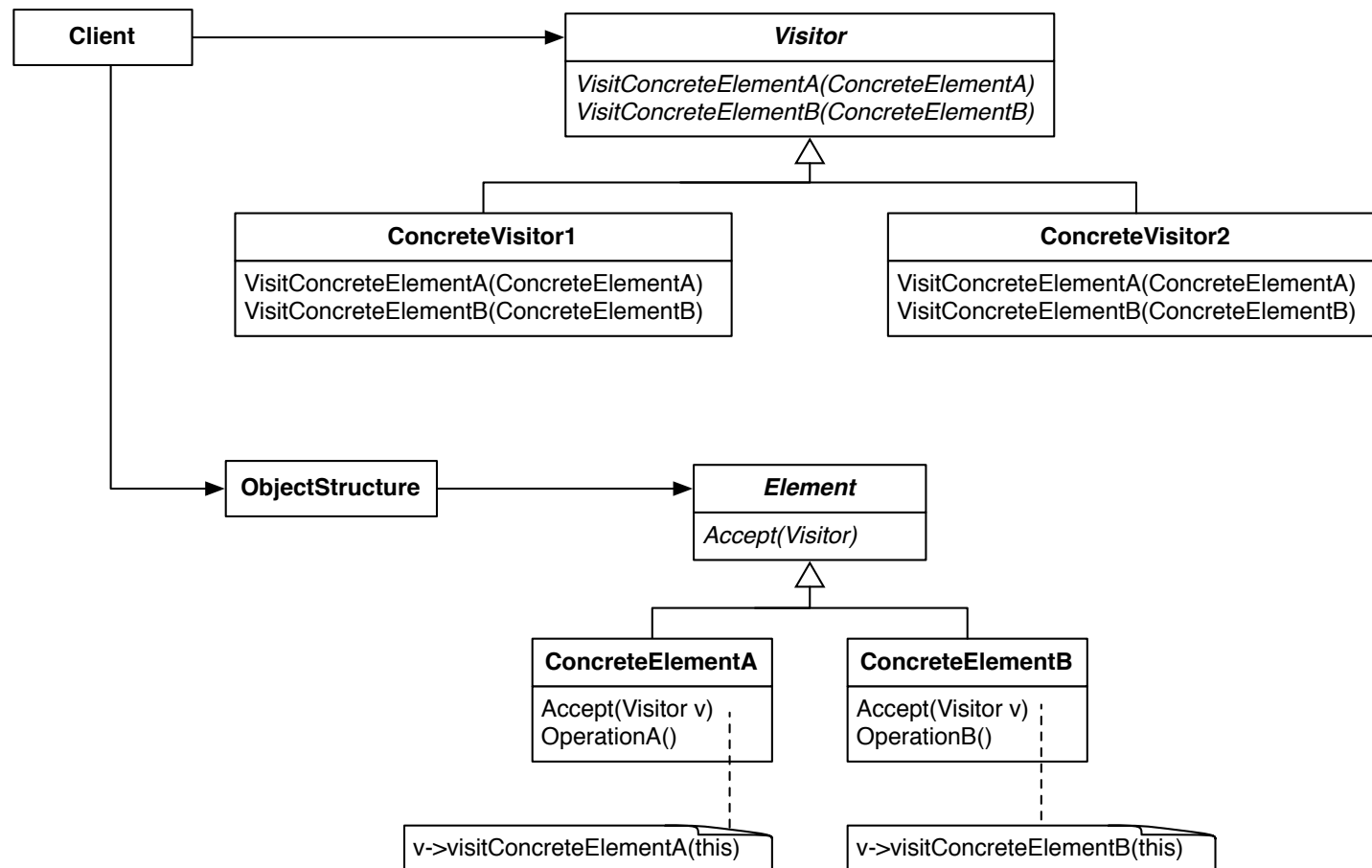
- **Conséquences**

- Principe d'Hollywood : « Don't call us, we will call you »
La classe parent appelle les opérations des sous-classes
- Utilisation des **hook** (opérations avec comportement par défaut que les sous-classes peuvent étendre)

Visitor

- **On l'utilise quand :**
 - une structure contient beaucoup de classes avec des interfaces \neq , et on veut appliquer des opérations sur leurs instances
 - on veut éviter de « polluer » les classes par toutes les opérations qu'on veut leur appliquer
 - la structure des classes est stable, mais pas les opérations

Visitor



Visitor

- **Conséquences**

- Ajouter des opérations devient plus facile (ajout d'un nouveau visitor)
- Les opérations liées sont regroupées, et séparées des autres
- Ajouter un nouvel élément à la structure est difficile (il faut changer chaque visiteur)
- Les visiteurs peuvent cumuler des états au fil des visites
- L'encapsulation des éléments de la structure est compromise

Choisir un design pattern

- Est-il une solution au pb ?
- Quels sont ses buts ?
- Existe-t-il des relations avec d'autres modèles ?
- Existe-t-il des modèles qui ont le même rôle ?
- Quels sont les motifs de refonte d'une conception ?
- Qu'est-ce qui doit rester libre dans la conception ?

Mise en œuvre d'un design pattern

- Lire complètement la description
attention aux sections **Indications d'utilisation** et **conséquences**
- Etudier en détail les sections **Structure**, **Constituants** et **Collaborations**
- Examiner la section **Exemple de code**
- Choisir des noms de constituant ayant un sens dans le contexte d'utilisation
- Définir les classes et leurs opérations
- Implémenter les opérations qui supportent les responsabilités et les collaborations

Ce qu'il ne faut pas attendre des DP

- Solutions universelles prêtes à l'emploi
- Bibliothèque de classes réutilisables
- Automatisation de la production du modèle de conception
- La disparition totale du facteur humain

Les anti-patterns

- Un **anti-pattern** ou **anti-patron** c'est :
 - une bonne solution à première vue
 - couramment mise en œuvre
 - mais qui finalement présente plus d'inconvénients et de problèmes que les bénéfices envisagés
- Ex d'anti-patterns sociaux :
 - terrorisme, meurtre, addiction à la drogue, etc.

Les anti-patterns

- Les **anti-patterns** logiciels sont des erreurs courantes de conception
 - Action, processus ou structure qui produit plus d'effets négatifs que le bénéfice qu'il était censé apporter
 - Erreurs courantes
 - Il existe une solution prouvée, documentée et applicable pour les résoudre
- Références :
 - W.J. Brown & al (1998) *Anti Patterns - Refactoring Software, Architectures, and Projects in Crisis*, Wiley.
 - <http://www.antipatterns.com/briefing>

Les anti-patterns

- Ils se caractérisent souvent par
 - Lenteur excessive du logiciel
 - Coûts de réalisation ou maintenance élevés
 - Comportements anormaux
 - Présence de bogues
- Le fait de les connaître permet d'éviter de tomber dans les mêmes « panneaux »...

Les anti-patterns

- Exemples d'anti-patterns
 - Le code spaghetti
 - L'attente active
 - L'objet divin
 - La coulée de lave
 - Réinventer la roue carrée
 - L'ancre de bateau
 - Le copy-and-paste programming
 - ...

