Les design patterns

M. Belguidoum

Université Mentouri de Constantine

Master2 Académique

Plan

1. Introduction
   - Définition
   - Principe
2. D'un problème à un pattern
3. Les patrons de GoF
4. Patrons pour l'intergiciel à objets répartis
   - Proxy
   - Factory
   - Adapter
   - Intercepter ou Observer
   - Comparaison des approches
5. Etude de cas : le Patron architectural MVC
6. Conclusion

Design pattern : définition

Définition1

"Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cIJur de la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière " . Christopher Alexander, 1977.

Définition2

Un Design Pattern est une solution à un problème récurrent dans la conception d'applications orientées objet. Un patron de conception décrit alors la solution éprouvée pour résoudre ce problème d'architecture de logiciel. Comme problème récurrent on trouve par exemple la conception d'une application où il sera facile d'ajouter des fonctionnalités à une classe sans la modifier. A noter qu'en se plaçant au niveau de la conception les Design Patterns sont indépendants des langages de programmation utilisés.

Historique

- Christopher Alexander, 1977 (architecture et urbanisme) : *Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.*
- Un exemple historique (1988) : le modèle MVC de Smalltalk (*Model-View-Controller*)
- Le catalogue de la bande des quatre (GoF design patterns) : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (1995)

Design pattern : principe

- Description d'une solution à un problème général et récurrent de conception dans un contexte particulier
- Description des objets communicants et des classes
- Indépendant d'une application ou spécifique
- Certains patterns sont relatifs à la concurrence, à la programmation distribuée, temps-réel
- Tous les patrons visent à renforcer la cohésion et à diminuer le couplage

Les avantages

- Résolution des problèmes de conception
- Utilisation d'un vocabulaire commun
- Capitalisation de l'expérience
- Niveau d'abstraction élevé pour permettre une conception de meilleure qualité
- Réduction de la complexité
- Catalogue de solutions

Les inconvénients

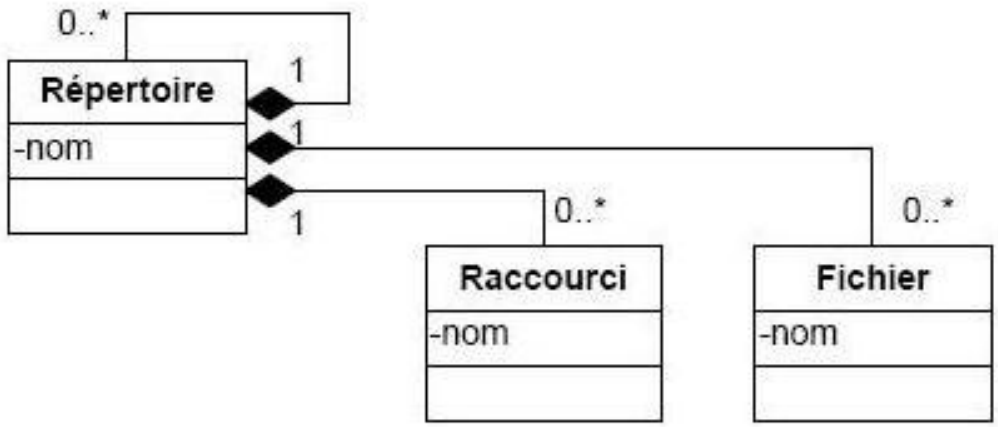
- Effort de synthèse : identification et application dans une conception
- Apprentissage et expérience nécessaires
- Dissolution des design patterns
- Nombreux design patterns : recoupement à des niveaux différents

D'un problème à un pattern

- **Exemple** : [UML 2 par la pratique, P. Rocques]
- **Énoncé** : proposer une solution élégante qui permette de modéliser le système de gestion de fichiers suivant :
  1. les fichiers, les raccourcis et les répertoires sont contenus dans des répertoires et possèdent un nom
  2. un raccourci peut concerner un fichier ou un répertoire
  3. au sein d'un répertoire donné, un nom ne peut identifier qu'un seul élément (fichier, sous-répertoire ou raccourci)

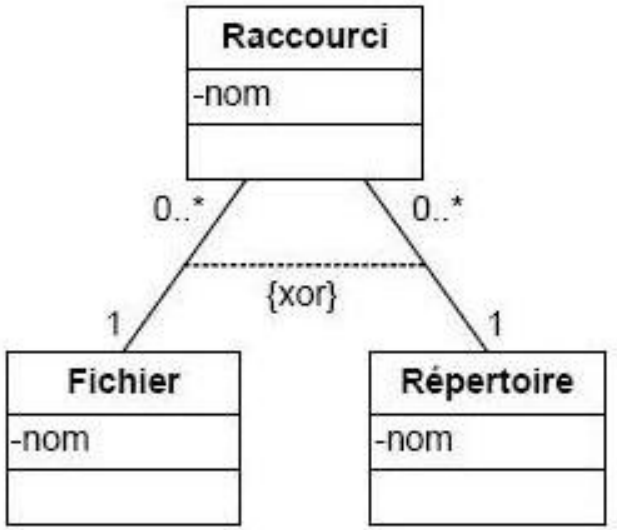
D'un problème à un pattern

- **1ère phrase**
  - 3 concepts = 3 classes
  - contenance modélisée par une composition
  - multiplicité côté contenant est égale à 1
  - destruction répertoire entraîne destruction de tout ce qu'il contient



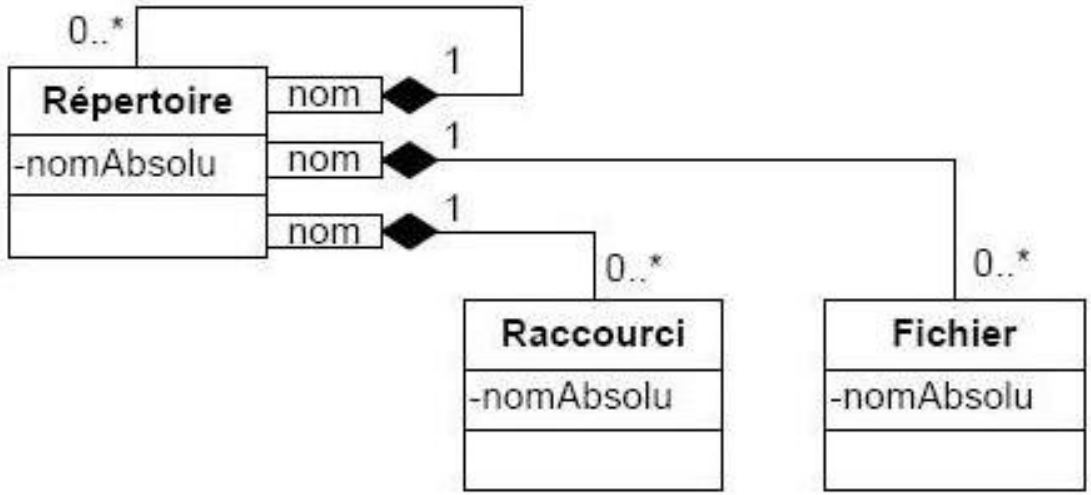
D'un problème à un pattern

- **2ère phrase** : un raccourci peut concerner un fichier ou un répertoire
  - 2 associations en exclusion mutuelle



D'un problème à un pattern

- **3ère phrase** : au sein d'un répertoire donné, un nom ne peut identifier qu'un seul élément (fichier, sous-répertoire ou raccourci)
  - **idée** : qualifier chacune des trois compositions avec un attribut nom



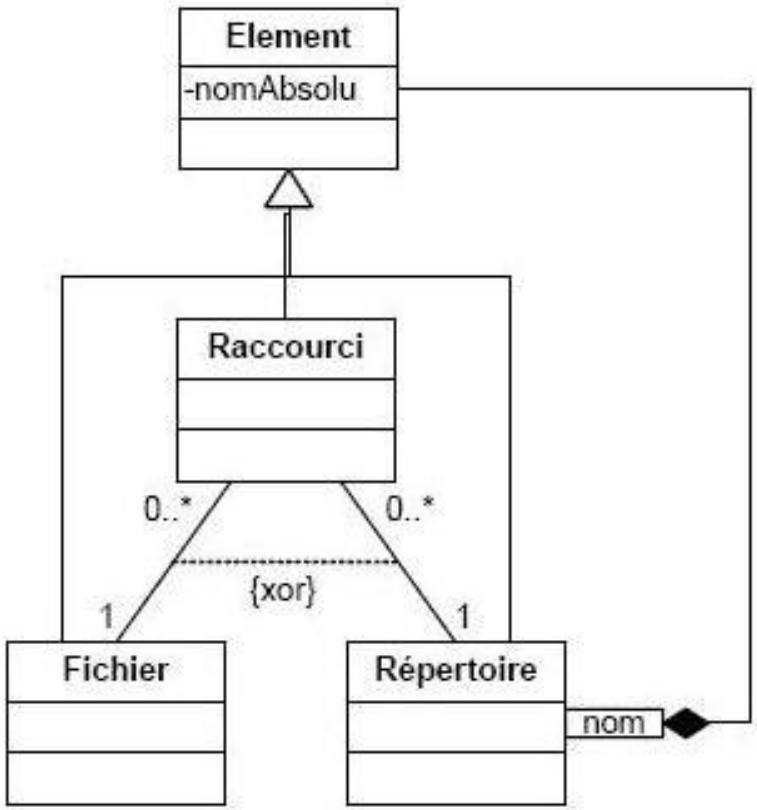
D'un problème à un pattern

- **Problème** : rien n'empêche qu'un fichier et qu'un raccourci aient le même nom
  - 3 compositions et 3 noms ne sont pas une si bonne idée que cela
  - il faut un qualificatif unique pour chaque type d'élément contenu dans un répertoire
- **Solution** : importance du terme élément
  - modifier le modèle afin de n'avoir plus qu'une composition à qualifier

D'un problème à un pattern

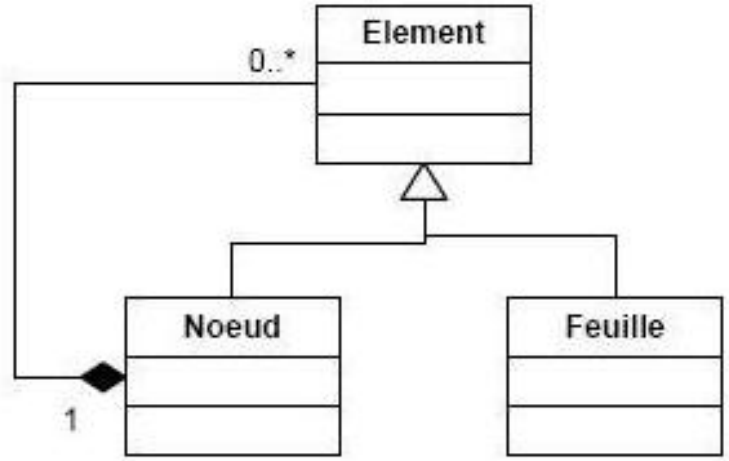
Solution

- double relation asymétrique entre Repertoire et Element
  - Répertoire contient des Element
  - Répertoire est un Element



D'un problème à un pattern

- **Le pattern Composite**
  - solution pour représenter des hiérarchies composant/composé
  - le client peut traiter de la même façon les objets individuels (feuilles) et leurs combinaisons (composites)



Les patrons de GoF

On distingue trois familles de patrons de conception selon leur utilisation :

- **de création** : création d'objets sans passer par l'instantiation directe d'une classe
- **structuraux** : assemblage de classes et d'objets.
- **comportementaux** : modélisation des interaction de classes et d'objets et de la répartition des responsabilités

Les patrons de GoF

Les Patrons de conception les plus répandus du GoF sont 23 :

- **Création**
  - **Fabrique abstraite (*Abstract Factory*)** : permet la création de familles d'objets ayant un lien ou interdépendants
  - **Monteur (*Builder*)** : déplace la logique de construction d'un objet en dehors de la classe à instantier, pour permettre une construction partielle ou pour simplifier l'objet
  - **Fabrique (*Factory Method*)** : laisse autre développeur définir l'interface permettant de créer un objet, tout en gardant un contrôle sur le choix de la classe à instancier.
  - **Prototype (*Prototype*)** : fournit de nouveaux objets par la copie d'un exemple
  - **Singleton (*Singleton*)** : garantit qu'une classe ne possède q'une seule instance, et fournit un point d'accès global à celle-ci

Les patrons de GoF

- **Structure**
  - **Adaptateur (*Adapter*)** : fournit l'interface qu'un client attend en utilisant les services d'une classe dont l'interface est différente.
  - **Pont (*Bridge*)** : découple une classe qui s'appuie sur des opérations abstraites de l'implémentation de ces opérations, permettant ainsi à la classe et à son implémentation de varier indépendamment.
  - **Objet composite (Composite)** : permet aux clients de traiter de façon uniforme des objets individuels et des compositions d'objets. Cela promeut un couplage lâche, évitant aux objets d'avoir à se référer les uns les autres, et permet de varier leur interaction indépendamment.
  - **Décorateur (*Decorator*)** : permet de composer dynamiquement le comportement d'un objet
  - **Façade (*Facade*)** : fournit une interface simplifiant l'emploi d'un sous-système
  - **Poids-mouche ou poids-plume (*Flyweight*)** : utilise le partage pour supporter efficacement un grand nombre d'objets à forte granularité
  - **Proxy (*Proxy*)** : contrôle l'accès à un objet en fournissant un intermédiaire pour cet objet.

Les patrons de GoF

- **Comportement**
  - **Chaîne de responsabilité (*Chain of responsibility*)** : évite de coupler l'émetteur d'une requête à son récepteur en permettant à plus d'un objet d'y répondre.
  - **Commande (*Command*)** : encapsule une requête dans un objet, de manière à pouvoir paramétrer des clients au moyen de divers types de requêtes
  - **Interpréteur (*Interpreter*)** : permet de composer des objets exécutables d'après un ensemble de règles de composition que vous définissez.
  - **Itérateur (*Iterator*)** : fournit un moyen d'accéder de façon séquentielle aux éléments d'une collection.
  - **Médiateur (*Mediator*)** : définit un objet qui encapsule la façon dont un ensemble d'objets interagissent
  - **Mémento (*Memento*)** : permet le stockage et la restauration de l'état d'un objet.

Les patrons de GoF

- **Comportement** (suite)
  - **Observateur (*Observer*)** : définit dépendance du type un à plusieurs (1,n) entre des objets de manière à ce que
  - **État (*State*)** : distribue la logique dépendant de l'état d'un objet à travers plusieurs classes qui représentent chacune un état différent
  - **Stratégie (*Strategy*)** : encapsule des approches, ou stratégies, alternatives dans des classes distinctes qui implémentent chacune une opération commune.
  - **Patron de méthode (*Template Method*)** : implémente un algorithme dans une méthode, laissant à d'autres classes le soin de définir certaines étapes de l'algorithme
  - **Visiteur (*Visitor*)** : permet de définir une nouvelle opération pour une hiérarchie sans changer ses classes.

Le catalogue GoF patterns

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

Formalisme de description

La description d'un patron de conception suit un formalisme fixe :

- **Nom**
- **Problème** : description du problème à résoudre
- **Solution** : les éléments de la solution, avec leurs relations. La solution est appelée patron de conception.
- **Conséquences** : résultats issus de la solution.
- **Autres** : implémentation, usages connus, etc.

Patrons pour l'intergiciel à objets répartis

- Les mécanismes d'exécution à distance reposent sur quelques patrons de conception [Gamma et al. 1994], [Buschmann et al. 1995], et [Schmidt et al. 2000]
- Nous mettons l'accent sur l'utilisation spécifique des patrons pour l'intergiciel réparti à objets :
  - Proxy
  - Factory
  - Pool
  - Adapter
  - Intercepter

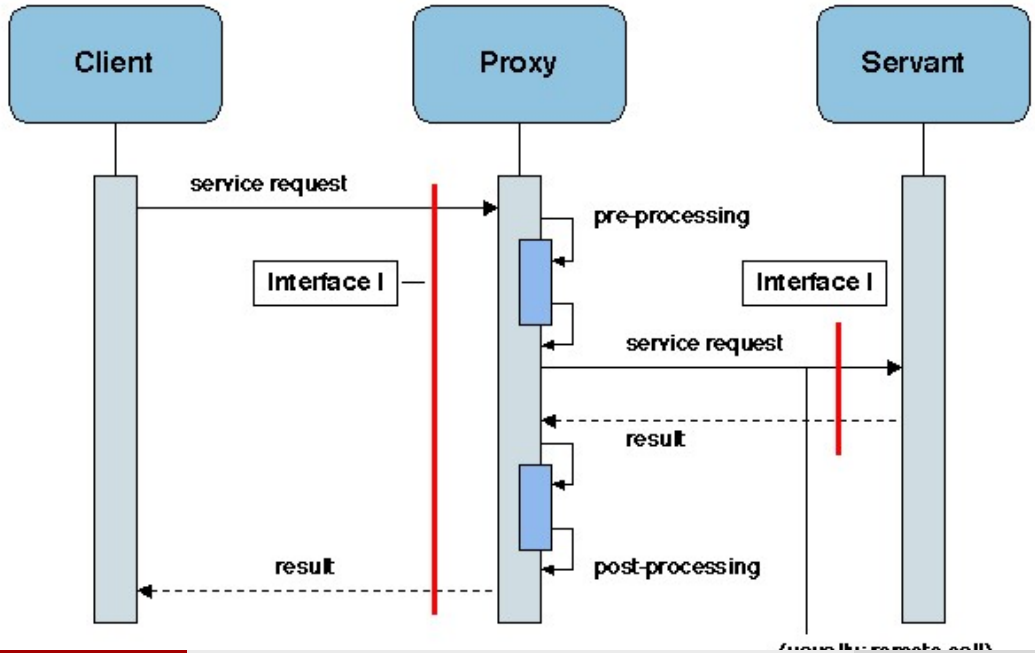
Proxy

- **But** : contrôle l'accès à un objet en fournissant un intermédiaire pour cet objet
- **Contexte** : le patron Proxy est utilisé pour des applications organisées comme un ensemble d'objets dans un environnement réparti, communicant au moyen d'appels de méthode à distance
- **Problème** : définir un mécanisme d'accès qui n'implique pas de coder l'emplacement du servant dans le code client, et qui ne nécessite pas une connaissance détaillée des protocoles de communication par le client.
- **Propriétés souhaitées** : l'accès doit être efficace à l'exécution. La programmation doit être simple pour le client ; il ne doit pas y avoir de différence entre accès local et accès distant (cette propriété est appelée transparence d'accès).
- **Contraintes** : la principale contrainte résulte de l'environnement réparti : le client et le serveur sont dans des espaces d'adressage différents.

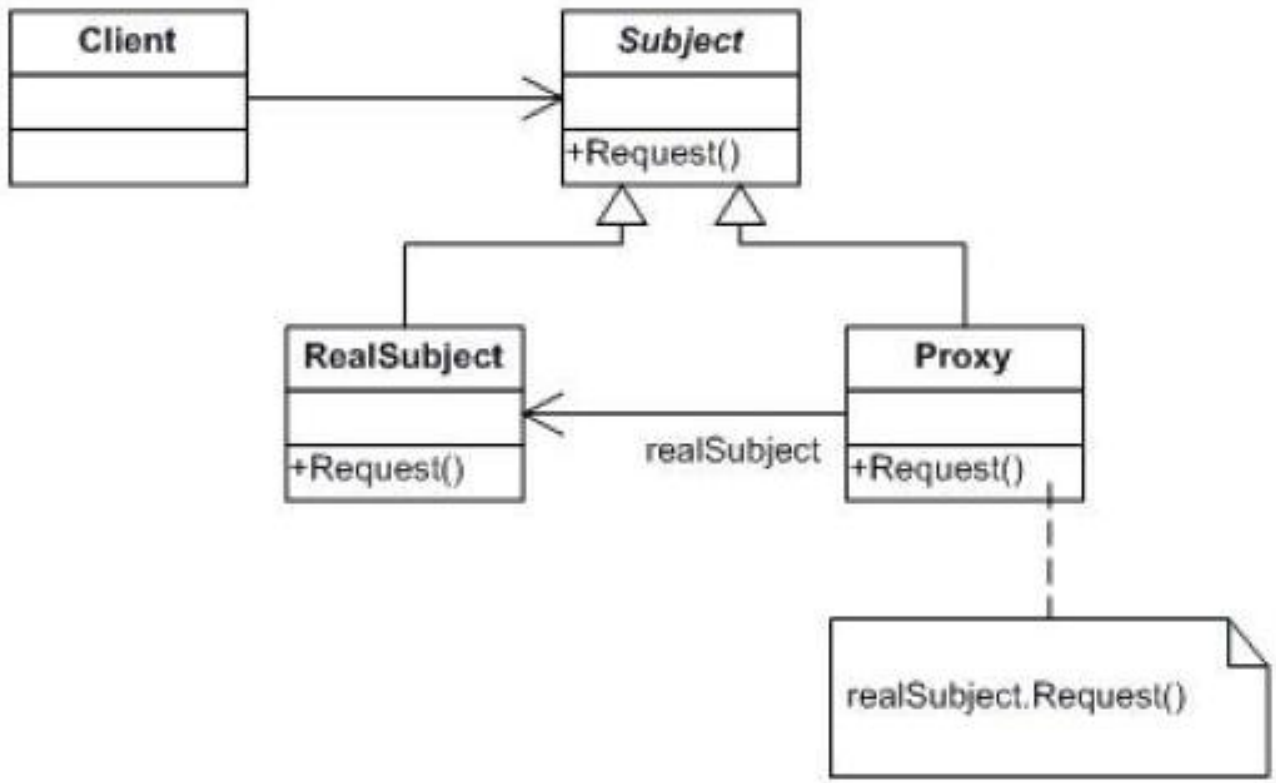
Proxy

La structure interne d'un mandataire suit un schéma bien défini, qui facilite sa génération automatique.

- pré-traitement : emballer les paramètres et à préparer le message de requête,
- l'appel effectif du servant, utilisant le protocole de communication pour envoyer la requête et pour recevoir la réponse,
- post-traitement : déballer les valeurs de retour.



Proxy : structure



Proxy : Participants

- **Participants** :
  - **Proxy** maintient une référence qui permet au Proxy d'accéder au RealSubject. Il fournit une interface identique à celle de Subject, pour pouvoir se substituer au RealSubject. Il contrôle les accès au RealSubject
  - **Subject** : définit une interface commune pour RealSubject et Proxy. Proxy peut ainsi être utilisé partout où le RealSubject devrait être utilisé
  - **RealSubject** : définit l'objet réel que le Proxy représente
- **Collaborations** : le Proxy retransmet les requêtes au RealSubject lorsque c'est nécessaire, selon la forme du Proxy

Proxy

- **Conséquences** :
  - L'inaccessibilité du sujet est transparente
  - Comme le proxy a la même interface que son sujet, il peuvent être librement interchangeable
  - le proxy n'est pas un objet réel mais simplement la réplique exacte de son sujet
- **Implémentation** : possibilité de nombreuses optimisations...
- **Usages connus** :
  - Les souches (stubs) et les squelettes utilisés dans RPC ou Java-RMI (représentants locaux pour des objets distants).
  - Des variantes des proxies contiennent des fonctions supplémentaires : les caches et l'adaptation côté client.
  - Pour l'adaptation coté client, le proxy peut filtrer la sortie du serveur pour l'adapter aux capacités spécifiques d'affichage du client (couleur, résolution, etc.).
  - De tels mandataires "intelligents" (smart proxies) combinent les fonctions standard d'un mandataire avec celles d'un intercepteur

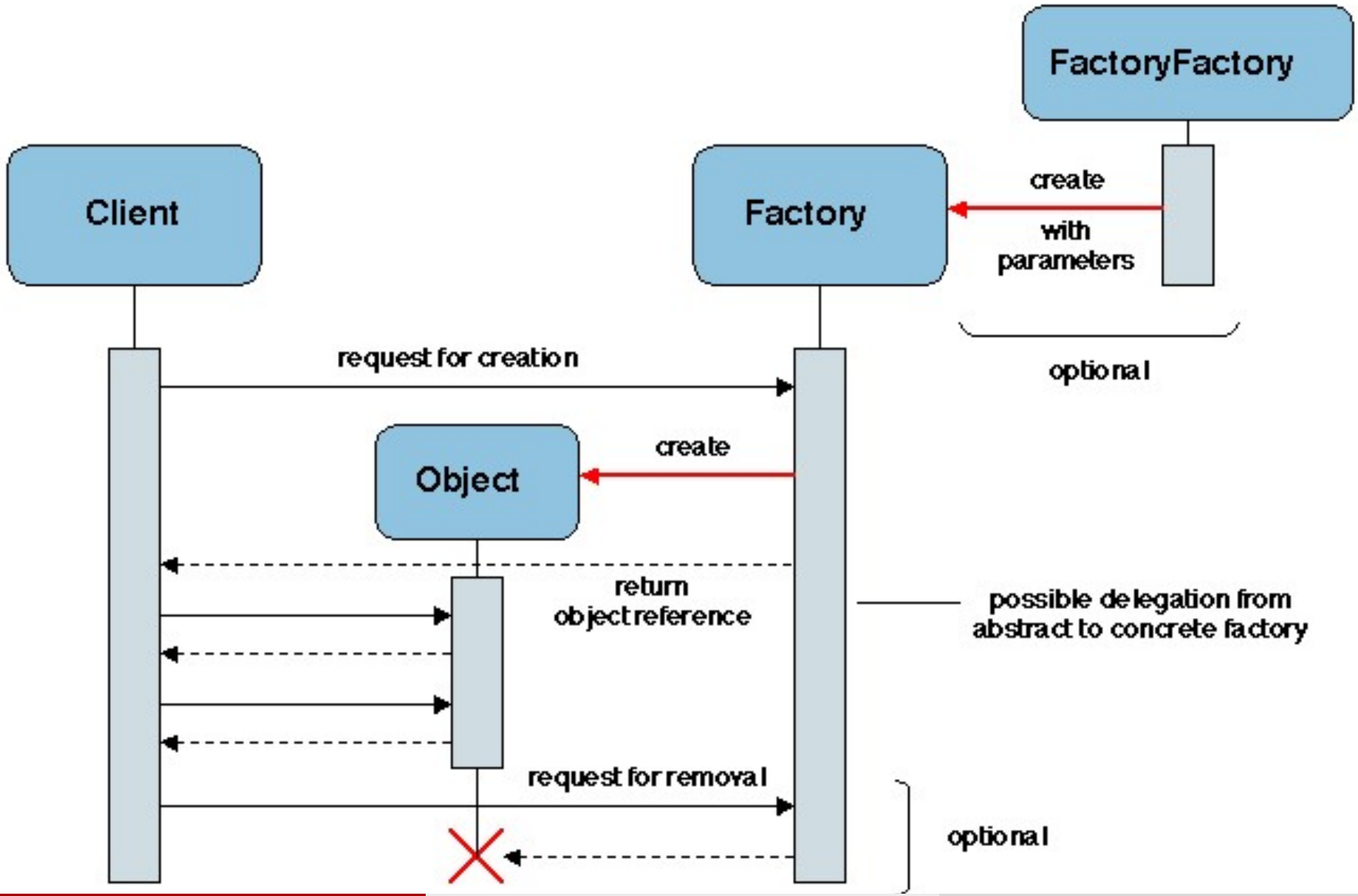
Factory

- **Contexte** : on considère des applications organisés comme un ensemble d'objets dans un environnement réparti
- **Problème** : on souhaite pouvoir créer dynamiquement des familles d'objets apparentés (par exemple des instances d'une même classe), tout en permettant de reporter certaines décisions jusqu'à la phase d'exécution (par exemple le choix d'une classe concrète pour réaliser une interface donnée).
- **Propriétés souhaitées** : les détails de réalisation des objets créés doivent être invisibles. Le processus de création doit pouvoir être paramétré. L'évolution du mécanisme doit être facilitée
- **Contraintes** : le client et le serveur sont dans des espaces d'adressage différents.

Factory

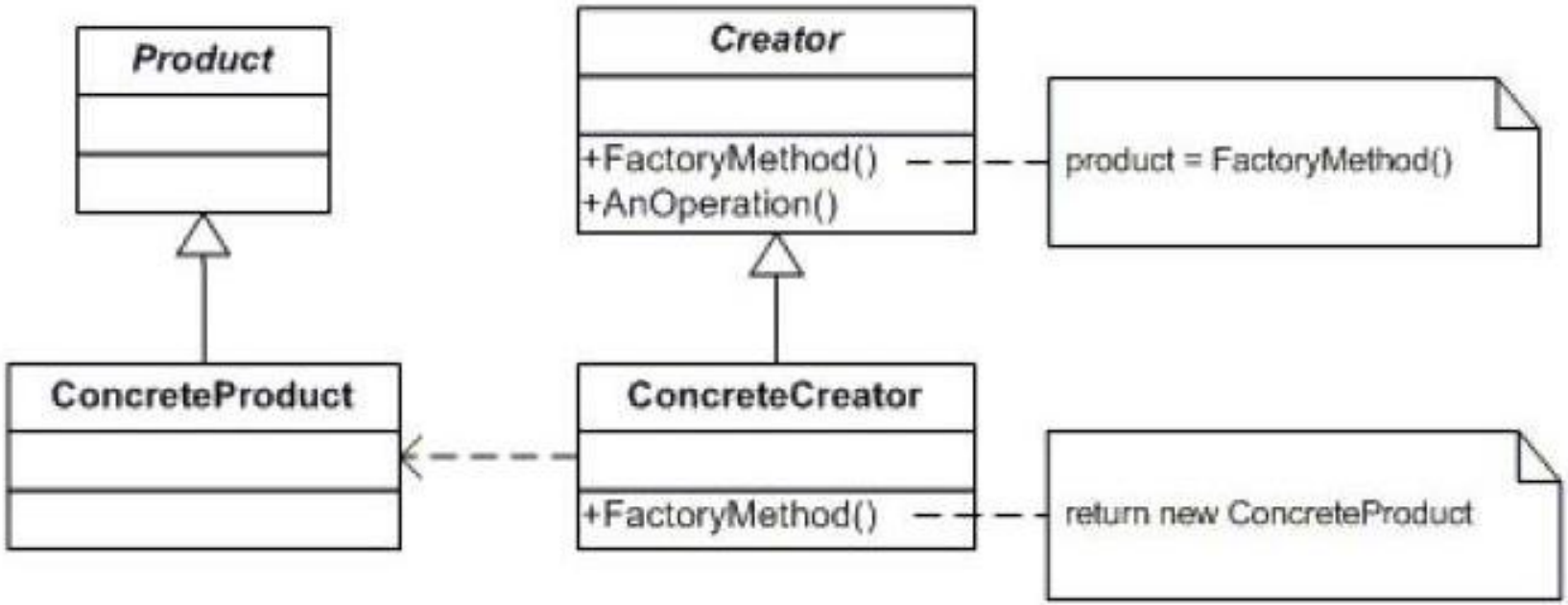
- **Solution** :
  - utiliser deux patrons corrélés : une usine abstraite Abstract Factory définit une interface et une organisation génériques pour la création d'objets ; la création est déléguée à des usines concrètes. Abstract Factory peut être réalisé en utilisant Factory Methods (une méthode de création redéfinie dans une sousclasse).
  - une autre manière d'améliorer la souplesse est d'utiliser une usine de fabrication d'usines (le mécanisme de création est lui-même paramétré).
  - une usine peut aussi être utilisée comme un gestionnaire des objets qu'elle a créés, et peut ainsi réaliser une méthode pour localiser un objet (en renvoyant une référence pour cet objet), et pour détruire un objet sur demande.
- **Usages connus** :
  - Factory est un des patrons les plus largement utilisés dans l'intergiciel.
  - il sert à la fois dans des applications (pour créer des instances distantes d'objets applicatifs) et dans l'intergiciel lui-même.
  - les usines sont aussi utilisées en liaison avec les composants

Factory


FactoryFactory
Client
Factory
create
with
parameters
optional
request for creation
Object
create
return
object reference
possible delegation from
abstract to concrete factory
request for removal
optional

Factory Method

Structure



Factory Method

- **Participants**
  - **Product** (Document) définit l'interface des objets créés par la fabrication
  - **ConcreteProduct** (MyDocument) implémente l'interface Product
  - **Creator** (Application) déclare la fabrication ; celle-ci renvoie un objet de type Product. Le Creator peut également définir une implémentation par défaut de la fabrication, qui renvoie un objet ConcreteProduct par défaut. Il peut appeler la fabrication pour créer un objet Product
  - **ConcreteCreator** (MyApplication) surcharge la fabrication pour renvoyer une instance d'un ConcreteProduct

Factory Method

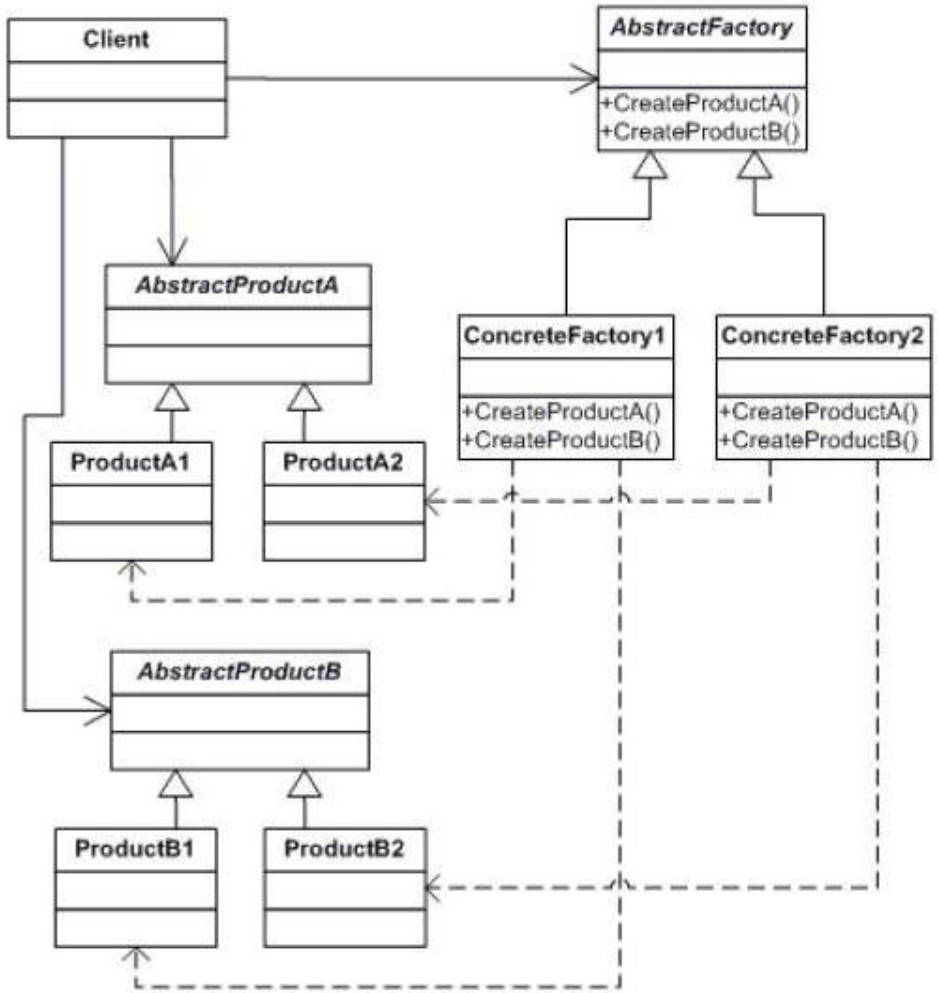
- **Conséquences**
  - la fabrication dispense d'avoir à incorporer à son code des classes spécifiques de l'application. Le code ne concerne que l'interface Product, il peut donc fonctionner avec toute classe ConcreteProduct définie par l'utilisateur
  - la création d'objets à l'intérieur d'une classe avec la méthode fabrication est toujours plus flexible que la création d'un objet directement

Factory Method

- **Implémentation**
  - 2 variantes principales :
    - la classe Creator est une classe abstraite et ne fournit pas d'implémentation pour la fabrication qu'elle déclare (les sous-classes définissent obligatoirement une implémentation)
    - la classe Creator est une classe concrète qui fournit une implémentation par défaut pour la fabrication
  - Fabrication paramétrée : la fabrication utilise un paramètre qui identifie la variété d'objet à créer
- **Usages connus** : applications graphiques, etc.

Abstract Factory

Structure



Abstract Factory

- **Participants** :
  - **AbstractFactory** déclare l'interface pour les opérations qui créent des objets abstraits
  - **ConcreteFactory** implémente les opérations qui créent les objets concrets
  - **AbstractProduct** déclare une interface pour un type d'objet
  - **ConcreteProduct** définit un objet qui doit être créé par la fabrique concrète correspondante et implémente l'interface AbstractProduct
  - **Client** utilise seulement les interfaces déclarées par AbstractFactory et par les classes AbstractProduct

Abstract Factory

- **Collaborations** :
  - une seule instance de fabrique concrète est créée à l'exécution. Cette fabrique crée les objets avec une implémentation spécifique. Pour créer différents sortes d'objets, les clients doivent utiliser différentes fabriques concrètes.
  - la fabrique abstraite défère la création des objets à ses sous-classes concrètes
- **Conséquences**
  - isolation des classes concrètes
  - échange facile des familles de produit
  - encouragement de la cohérence entre les produits
  - prise en compte difficile de nouvelles formes de produit

Abstract Factory

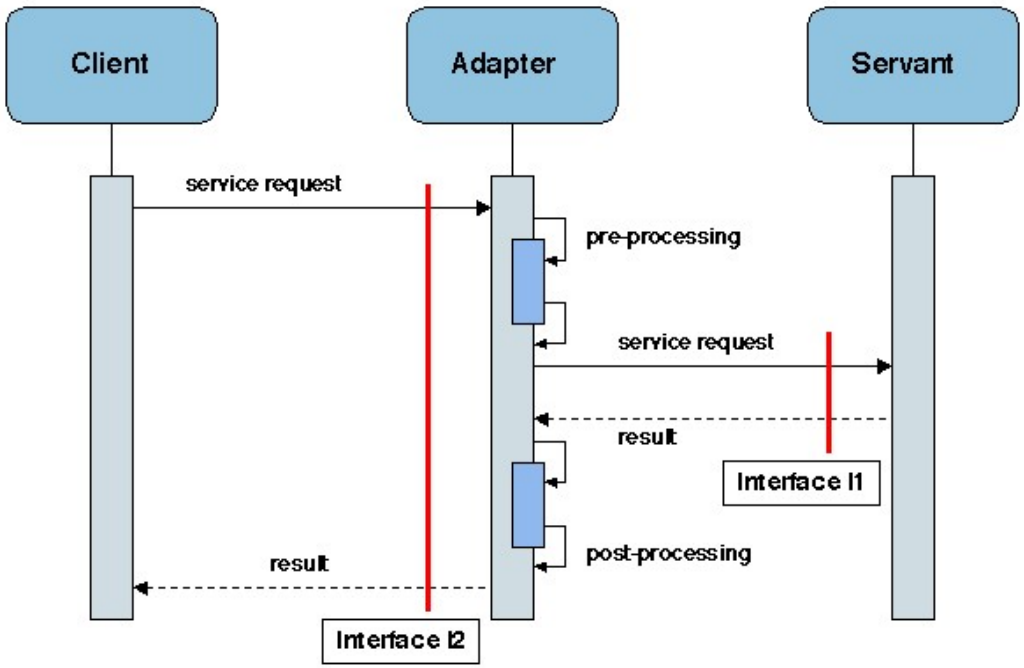
- **Implémentation**
  - Les fabriques sont souvent des singletons
  - Ce sont les sous-classes concrètes qui font la création, en utilisant le plus souvent une Factory Method
  - Si plusieurs familles sont possibles, la fabrique concrète utilise Prototype (créer de nouveaux objets en copiant un prototype existant).

Adapter

- **But** : fournit l'interface qu'un client attend en utilisant les services d'une classe dont l'interface est différente.
- **Contexte** : fourniture de services, dans un environnement réparti : un service est défini par une interface ; les clients demandent des services ; des servants, situés sur des serveurs distants, fournissent des services.
- **Problème** : réutiliser un servant existant en le dotant d'une nouvelle interface conforme à celle attendue par un client (ou une classe de clients).
- **Propriétés souhaitées** : le mécanisme de conversion d'interface doit être efficace à l'exécution. Il doit aussi être facilement adaptable, pour répondre à des changements imprévus des besoins. Il doit être réutilisable (générique).
- **Contraintes** : pas de contraintes spécifiques.

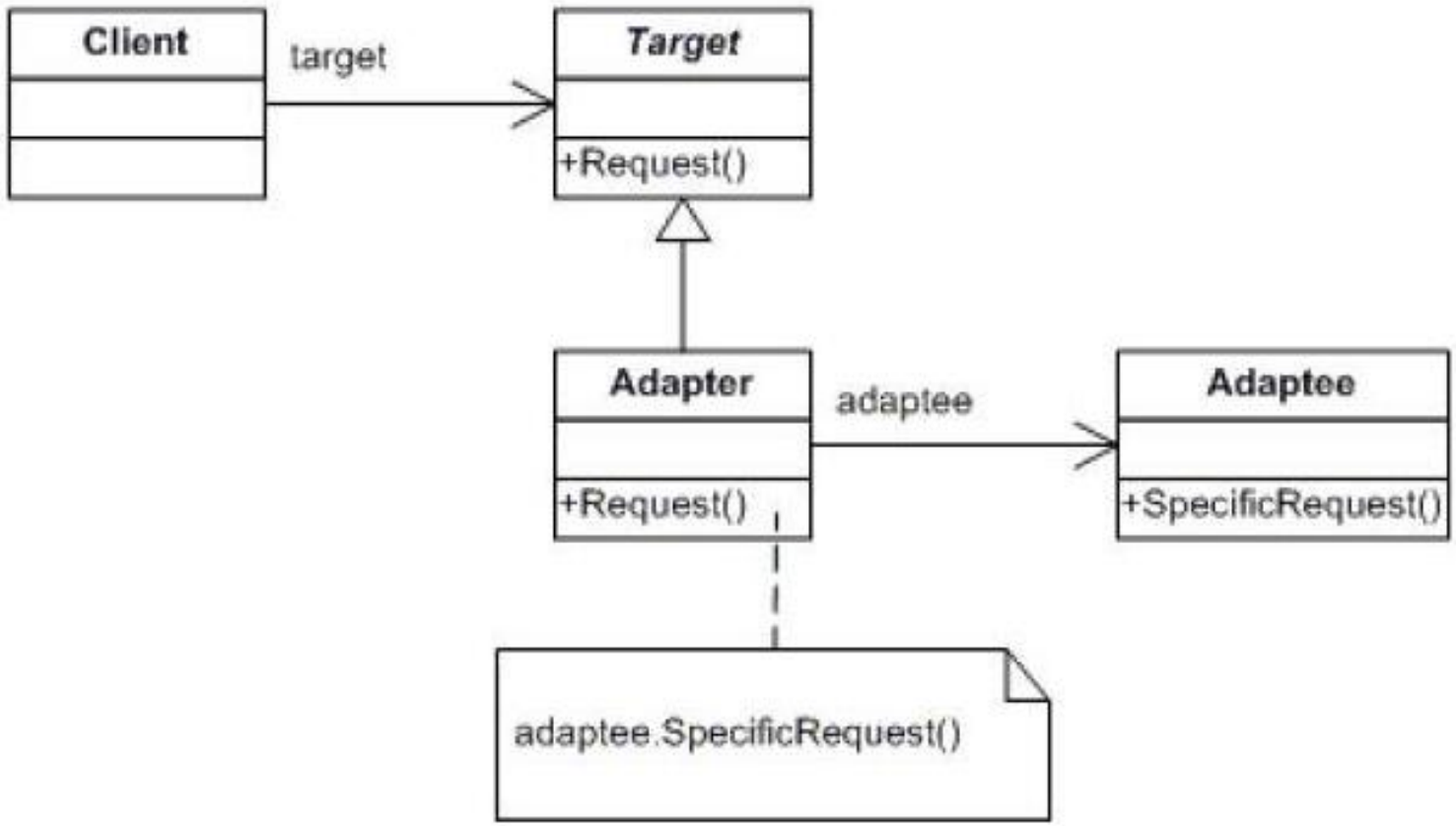
Adapter

- **Solution** : fournir un composant (adaptateur) qui isole le servant en interceptant les appels de méthode à son interface. Chaque appel est précédé par un prologue et suivi par un épilogue dans l'adaptateur. Il peut être engendré à partir d'une description des interfaces fournie et requise.



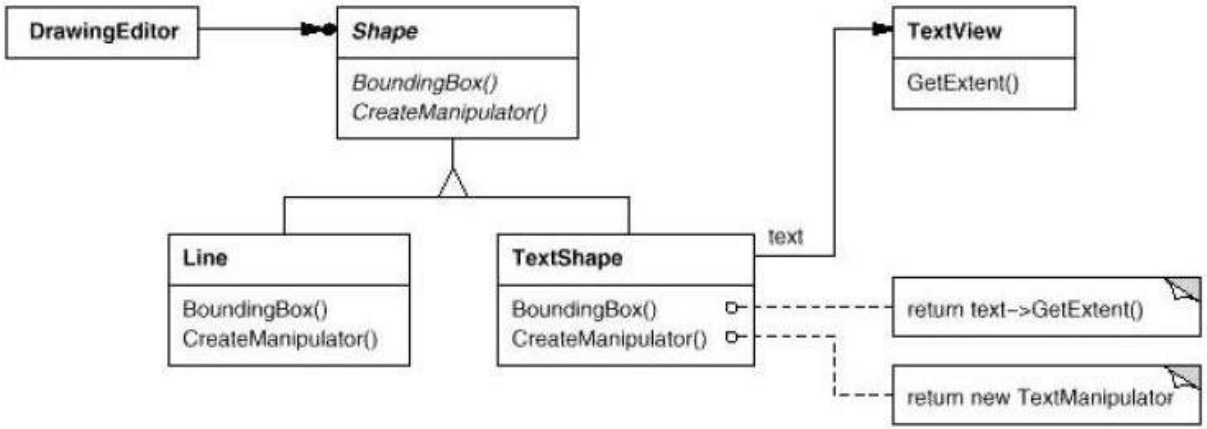
Adapter

- **Structure**



Adapter

- **Participants**
  - **Target (Shape)** : définit l'interface spécifique à l'application que le client utilise
  - **Client (DrawingEditor)** : collabore avec les objets qui sont conformes à l'interface de Target
  - **Adaptee (TextView)** : est l'interface existante qui a besoin d'adaptation
  - **Adapter (TextShape)** : adapte effectivement l'interface de Adaptee à l'interface de Target



Adapter

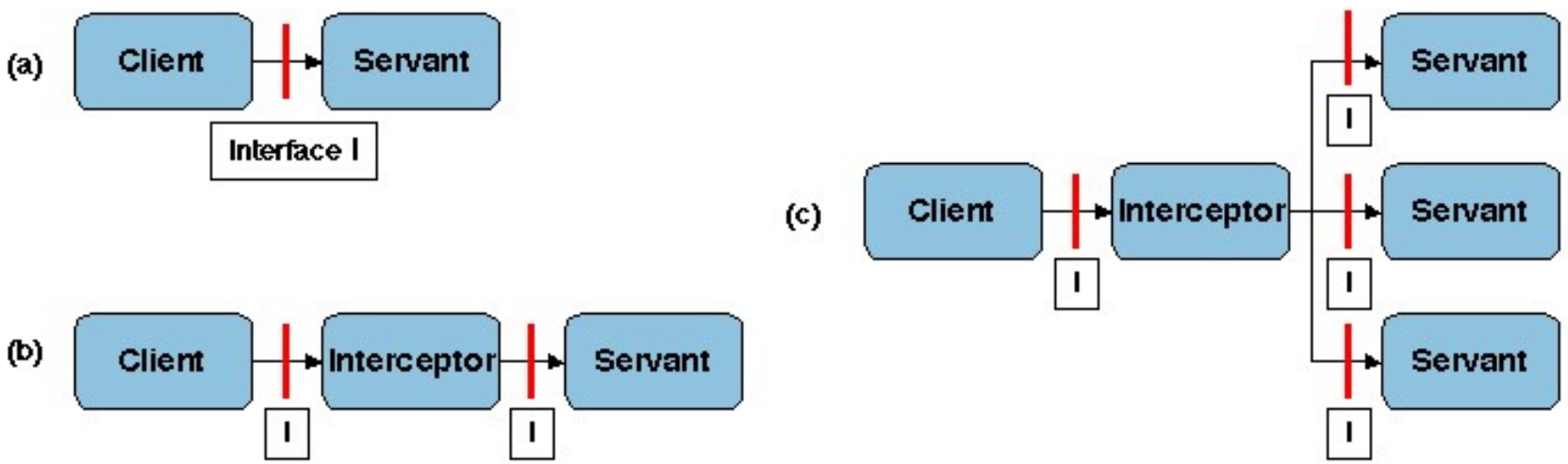
- **Collaborations** : le client appelle les méthodes sur l'instance d'Adapter. Ces méthodes appellent les méthodes d'Adaptee pour réaliser le service
- **Conséquences (adapter objet)**
  - un adapter peut travailler avec plusieurs Adaptees
  - plus difficile de redéfinir le comportement d'Adaptee
- **Conséquences (adapter classe)**
  - pas possible d'adapter une classe et ses sous-classes
  - mais redéfinition possible du comportement (sousclasse)
- **Usages connus** : Les adaptateurs sont largement utilisés dans l'intergiciel pour encapsuler des fonctions côté serveur. Des exemples sont le Portable Objet Adapter (POA) de CORBA et les divers adaptateurs pour la réutilisation de logiciel patrimoniaux (legacy systems), tel que Java Connector Architecture (JCA).

Intercepter ou Observer

- **Contexte** : fourniture de services, dans un environnement réparti : un service est défini par une interface ; les clients demandent des services ; les servants, situés sur des serveurs distants, fournissent des services. Il n'y a pas de restrictions sur la forme de la communication (uni- or bi-directionnelle, synchrone ou asynchrone, etc.).
- **Problème** : ajouter de nouvelles capacités à un service existant, ou fournir le service par un moyen différent.
- **Propriétés souhaitées** : le mécanisme doit être générique et doit permettre de modifier un service aussi bien statiquement que dynamiquement.
- **Contraintes** : les services peuvent être ajoutés ou supprimés dynamiquement.

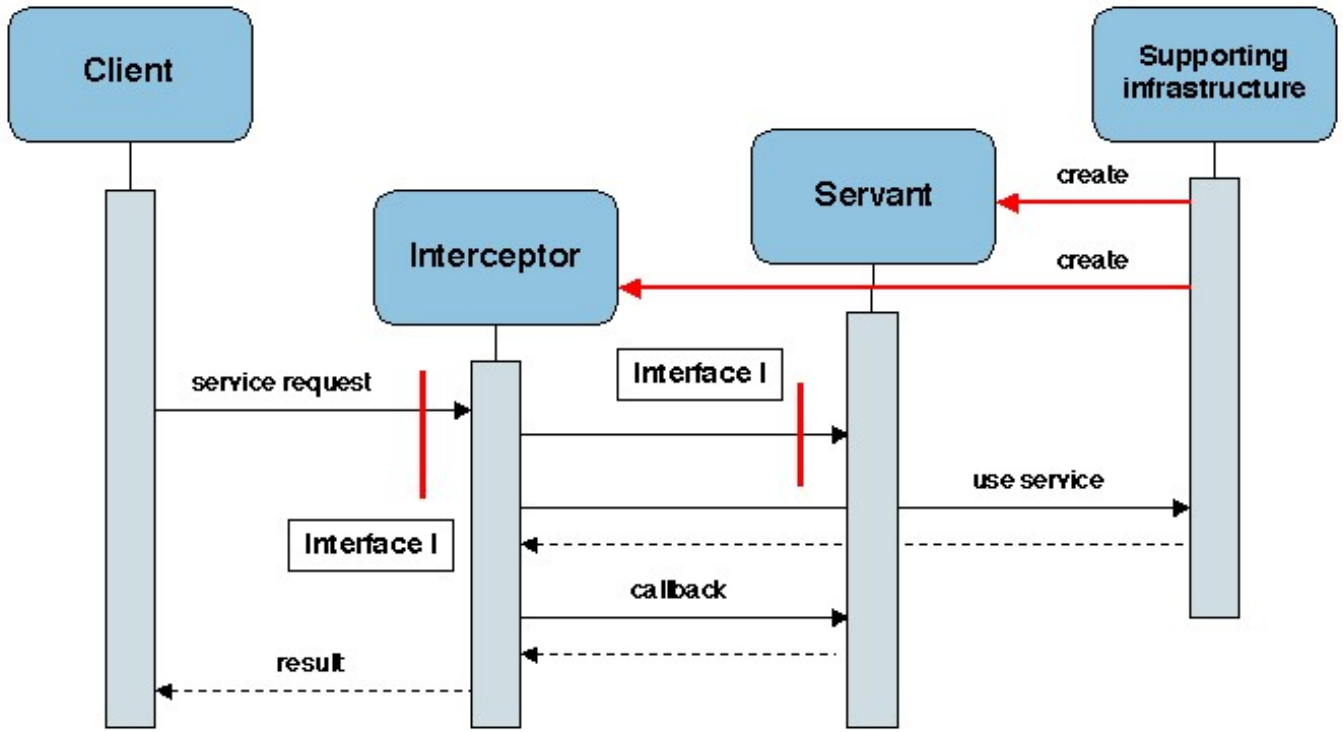
Intercepter ou observer

- **Solution** : créer (statiquement ou dynamiquement) des intercepteurs qui interceptent les appels (et/ou les retours) et insèrent un traitement spécifique et peuvent rediriger un appel vers une cible différente.
  - un intercepteur est un module qui est inséré à un point spécifique dans le chemin d'appel entre le demandeur et le fournisseur d'un service (Fig a et b).
  - Il peut aussi être utilisé comme un aiguillage entre plusieurs servants qui peuvent fournir le même service avec différentes options (Fig c).



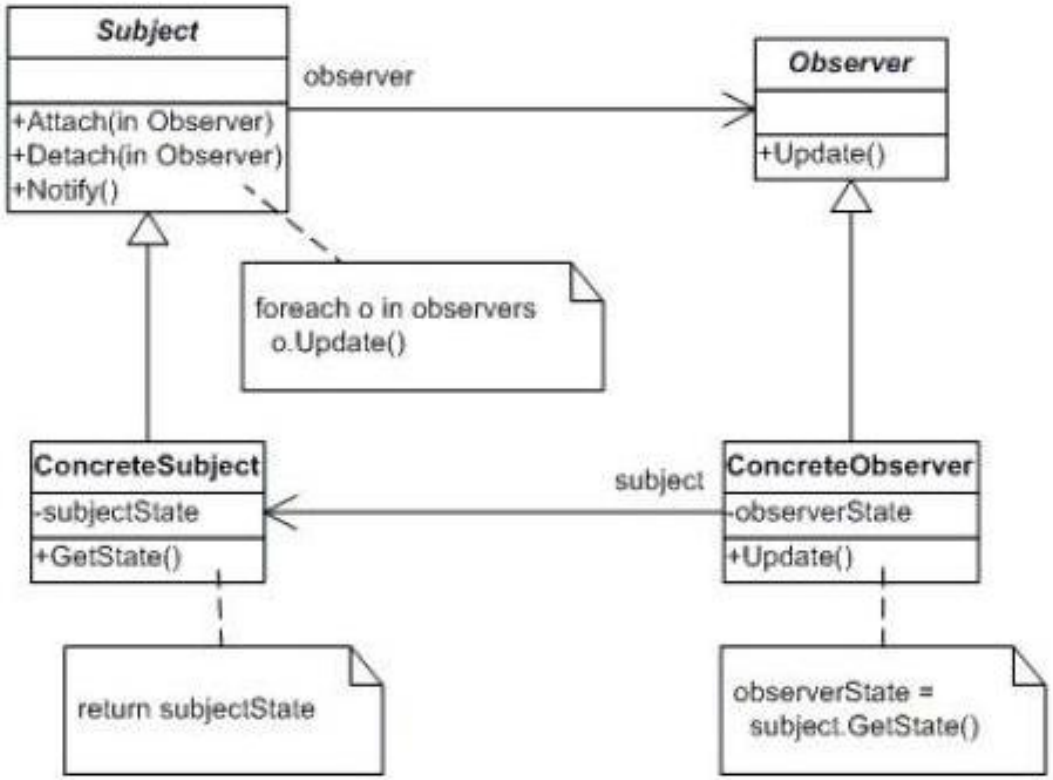
Intercepter ou observer

- **Solution** (suite) :
  - L'intercepteur utilise l'interface du servant et peut aussi s'appuyer sur des services fournis par l'infrastructure. Le servant peut fournir des fonctions de rappel utilisables par l'intercepteur.



Intercepter ou Observer

Structure



Intercepter ou Observer

- **Participants**
  - **Subject** : garde une trace de ses observateurs et propose une interface pour ajouter ou enlever des observateurs
  - **Observer** : définit une interface pour la notification de mises à jour
  - **ConcreteSubject** : est l'objet observé, il enregistre son état intéressant les observateurs et envoie une notification à ces observateurs quand il change d'état.
  - **ConcreteObserver** : observe l'objet concret et stocke l'état qui doit rester consistent avec celui de l'objet observé, il implémente l'interface de mise à jour permettant à son état de rester consistent avec celui de l'objet observé.

Intercepter ou Observer

- **Conséquences**
  - Variations abstraites des sujets et observateurs
  - Couplage abstrait entre le sujet et l'observateur
  - Prise en charge de communication broadcast
  - Mises à jour non prévues : protocole additionnel pour savoir ce qui a précisément changé
- **Implémentation**
  - Référence sur les observateurs, hash-table
  - Un observateur peut observer plusieurs sujets : étendre le update pour savoir qui notifie
  - faire attention : lors de la suppression des observés, à la consistance de l'observé avant notification et aux informations de notification

Intercepter ou Observer

- **Usages connus** : les systèmes intergiciels
  - pour ajouter des nouvelles capacités à des applications ou systèmes existants. Les *Portable Interceptors* de CORBA donnent une manière systématique d'étendre les fonctions du courtier d'objets (ORB) par insertion de modules d'interception en des points prédédéfinis dans le chemin d'appel. Un autre usage est l'aide aux mécanismes de tolérance aux fautes (par exemple la gestion de groupes d'objets).
  - pour choisir une réalisation spécifique d'un servant à l'exécution.
  - pour réaliser un canevas pour des applications à base de composants.
  - pour réaliser un intergiciel réflexif.

Comparaison et combinaison des approches

Les trois patrons : Proxy, Adapter et Intercepter reposent sur un module logiciel inséré entre le demandeur et le fournisseur d'un service.

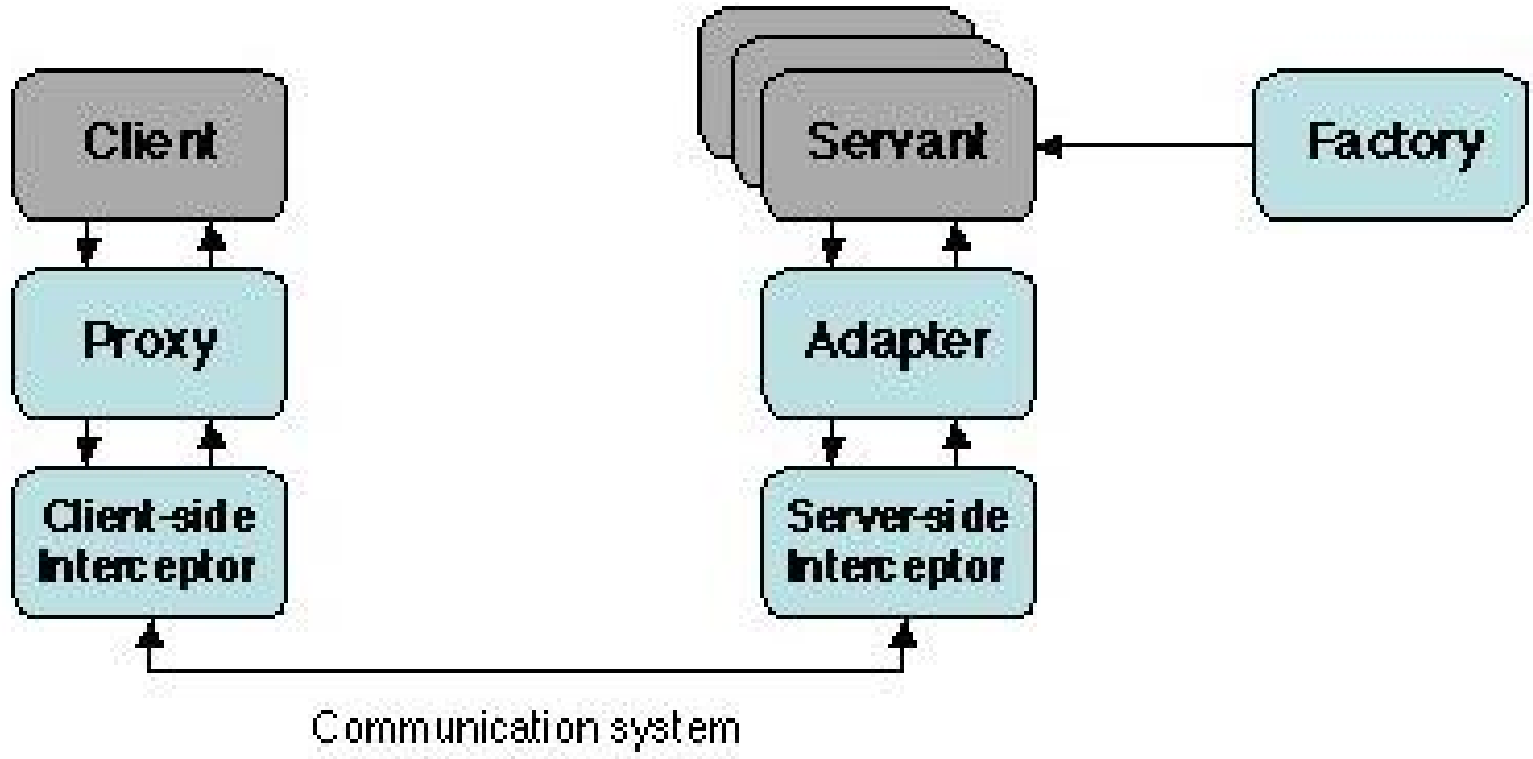
- **Adapter vs Proxy**
  - **analogies** : ont une structure semblable.
  - **différences** :
    - Proxy préserve l'interface, alors qu'Adapter transforme l'interface.
    - Proxy implique souvent (pas toujours) un accès à distance, alors que Adapter est généralement local.
- **Adapter vs Interceptor**
  - **analogies** : ont une fonction semblable : l'un et l'autre modifient un service existant.
  - **différences** : Adapter transforme l'interface, alors que Interceptor transforme la fonction (de fait Interceptor peut complètement masquer la cible initiale de l'appel, en la remplaçant par un servant différent).

Comparaison et combinaison des approches

- **Proxy vs Interceptor**
  - **analogies** : un Proxy peut être vu comme une forme spéciale d'un Interceptor, dont la fonction se réduit à acheminer une requête vers un servant distant, en réalisant les transformations de données nécessaires à la transmission, d'une manière indépendante des protocoles de communication.
  - **différence** : un proxy peut être combiné avec un intercepteur, devenant ainsi "intelligent" (fournissant de nouvelles fonctions en plus de la transmission des requêtes, mais laissant l'interface inchangée).

Utilisation de Patrons dans un ORB

Utilisation des patrons : Proxy, Adapter, Intercepteur et Factory pour tracer un premier schéma de l'organisation d'ensemble d'un ORB sans la représentation des liaisons et des communications



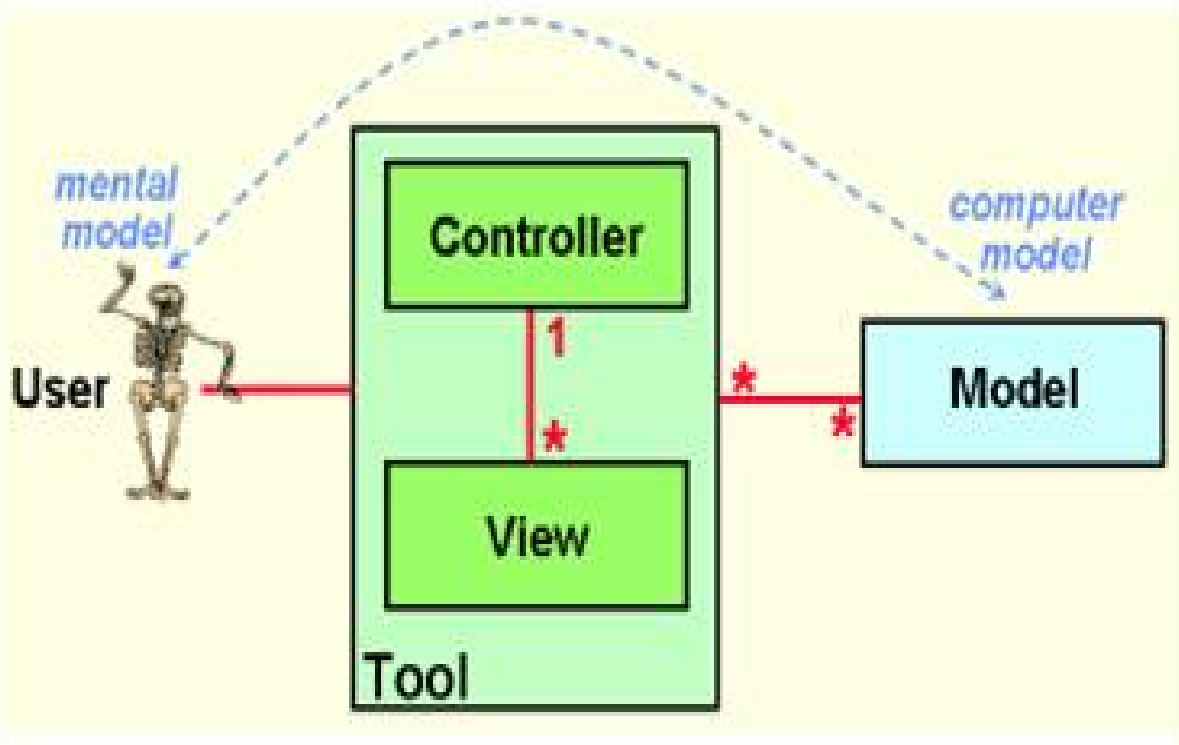
Etude de cas : le Patron MVC

Le patron MVC

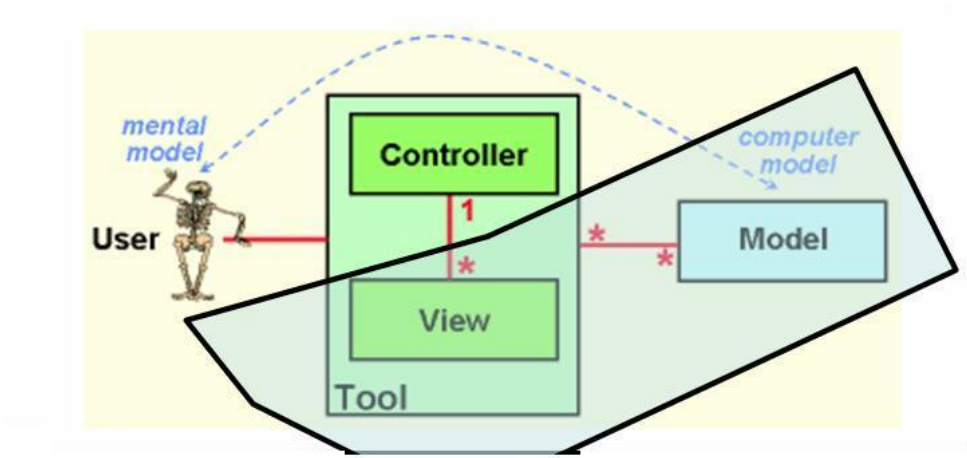
Modèle-Vue-Contrôleur est un pattern architectural qui sépare les *données* (**le modèle**), *l'interface homme-machine* (**la vue**) et la *logique de contrôle* (**le contrôleur**). MVC peut être vu comme une combinaison de design patterns :

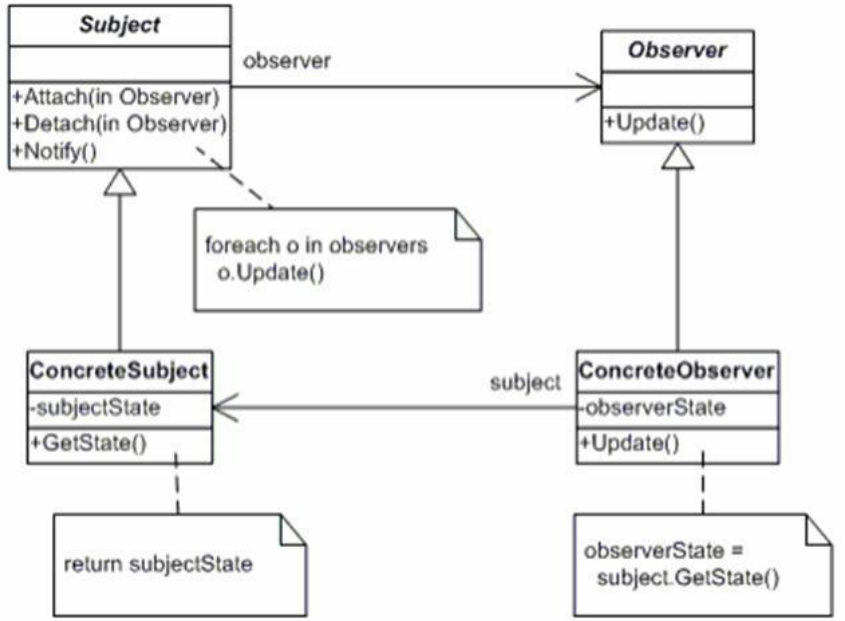
- les **vues** sont organisées selon **Composite**
- le lien entre **vues et modèles** est l'**Observer**.
- les **contrôleurs** sont des **Strategy** attachées aux vues.

Etude de cas : le Patron MVC

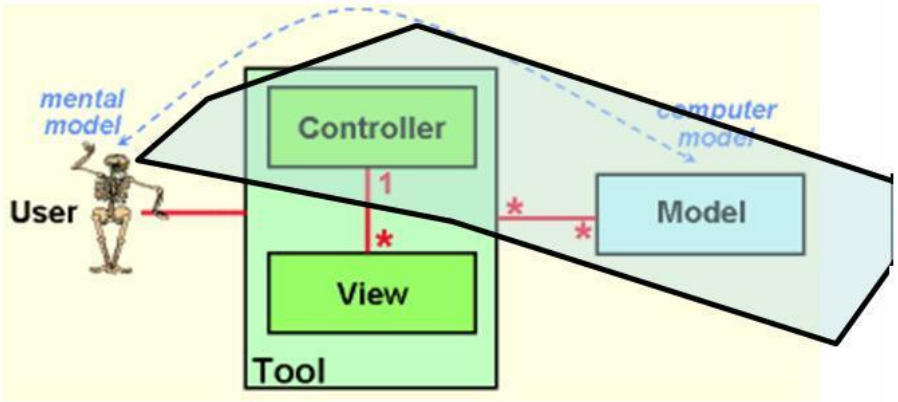


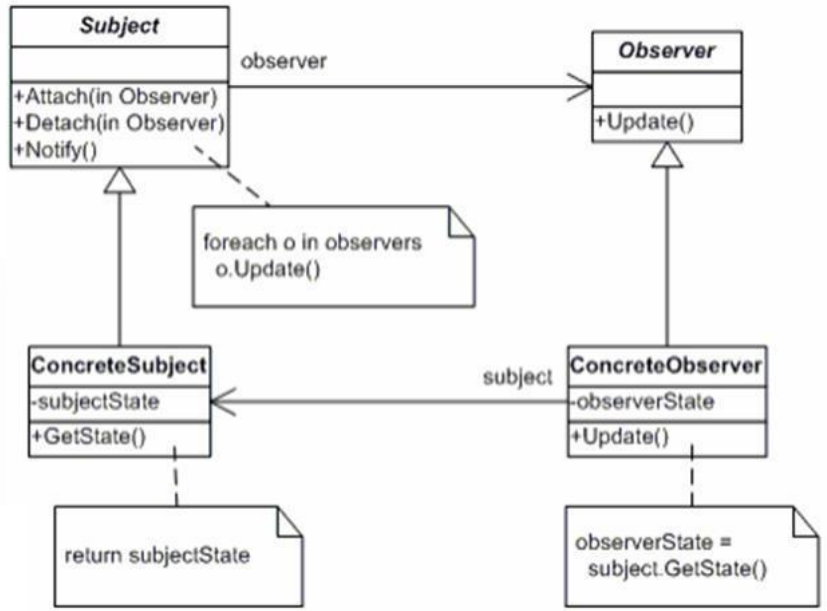
Le Patron MVC : le patron Observer (vue-modèle)



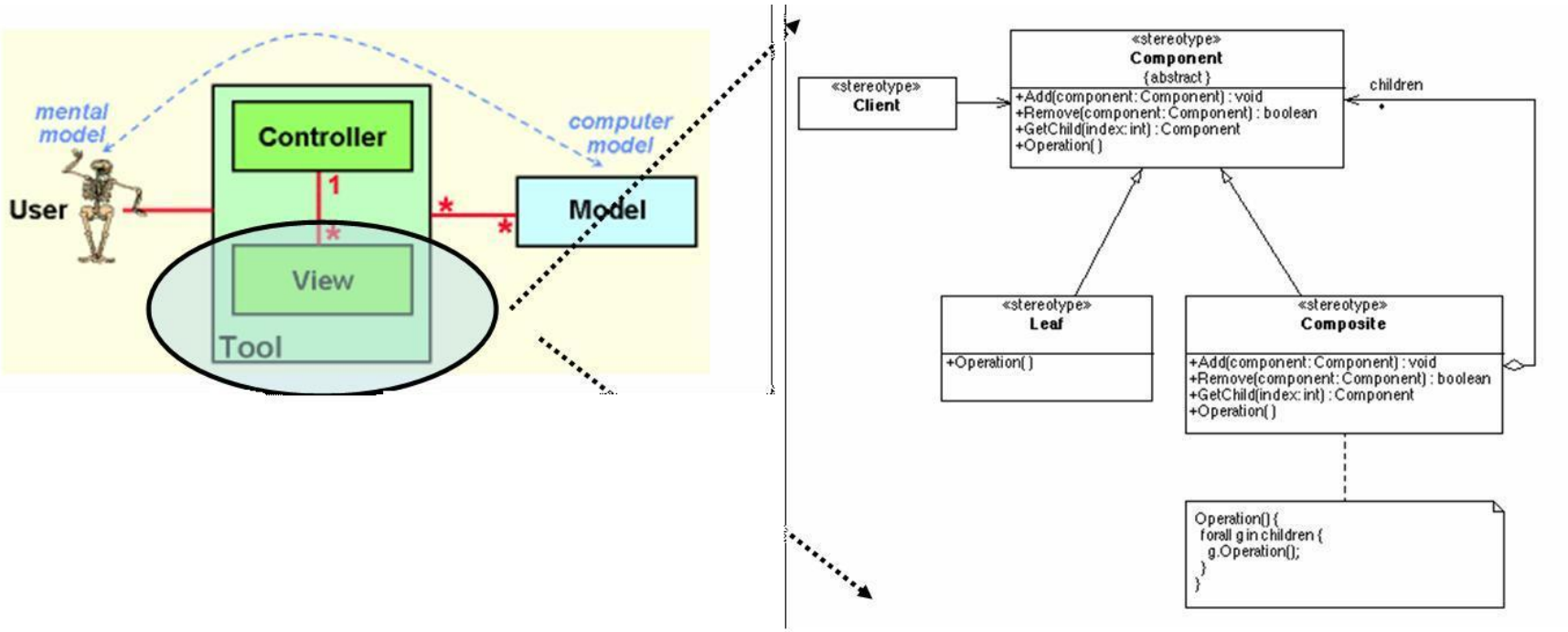


Le Patron MVC : le patron Observer (contrôleur-modèle)

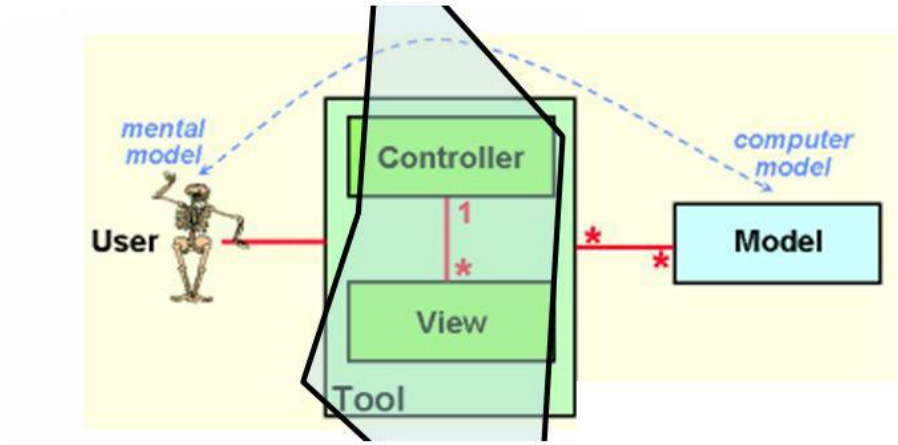


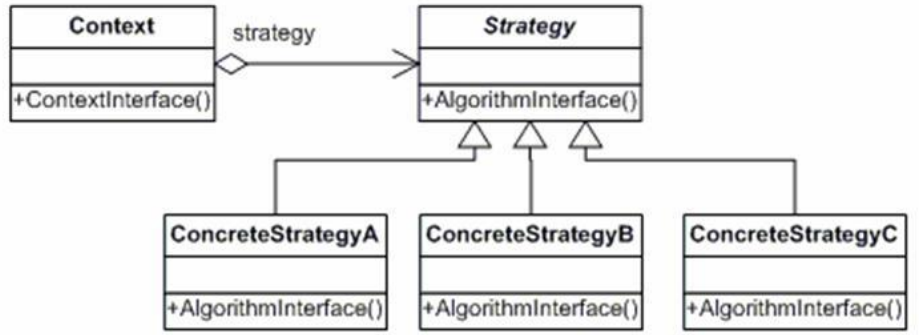


Le Patron MVC : le patron Composite (vue-vue)

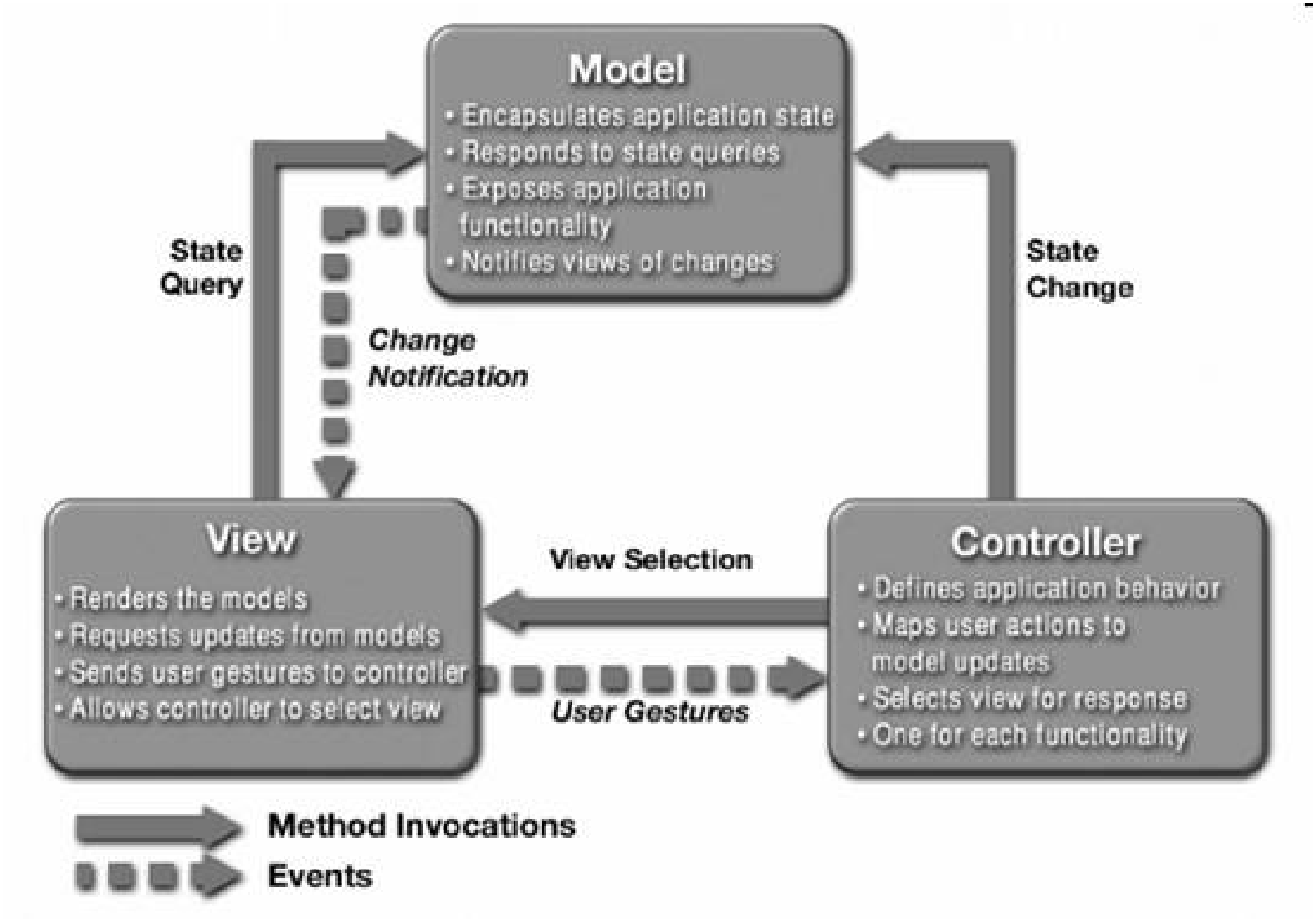


Le Patron MVC : le patron Strategy (vue-contrôleur)

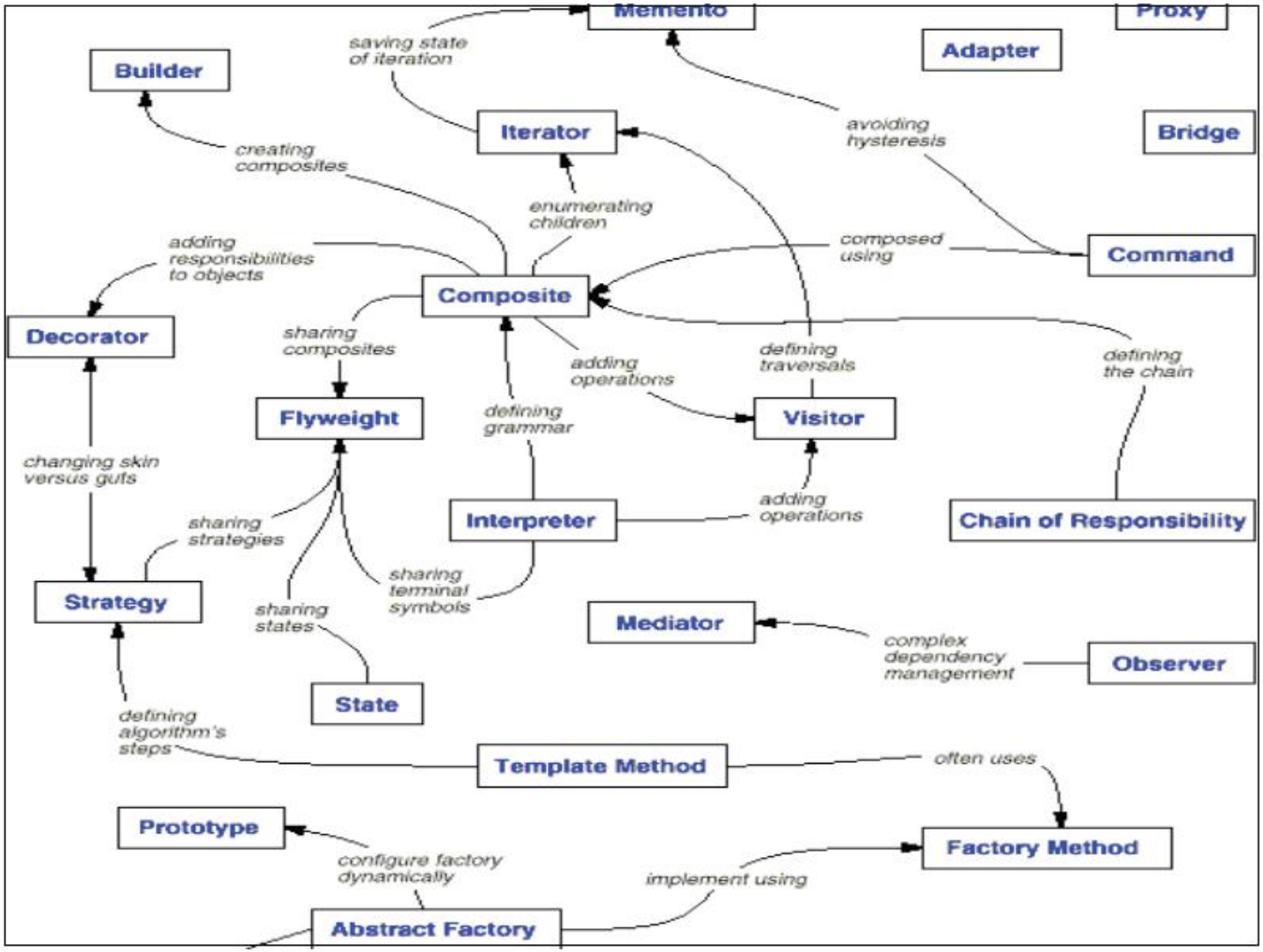




Le Patron MVC



Composition des Patrons



Conclusion

- les patrons représentent des solutions adaptables pour des problèmes récurrents
- les patrons représentent des propositions de base pour l'élaboration de solutions plus complexes
- les patrons permettent d'avoir des styles d'organisation du code pour
  - la création des objets
  - les structures de données
  - les comportements
- l'accent est mis sur l'utilisation spécifique des patrons pour l'intergiciel réparti à objets
- la génération automatique du code de certains patterns est possible

Références

- [Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns : Elements of Reusable Object Oriented Software*. Addison-Wesley. 416 pp.
- [Buschmann et al. 1995] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1995). *Pattern-Oriented Software Architecture, Volume 1 : A System of Patterns*. John Wiley & Sons. 467 pp.
- [Schmidt et al. 2000] Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2 : Patterns for Concurrent and Networked Objects*. John Wiley & Sons. 666 pp.
- Larman C. UML2 et les design patterns (2005).
- Metsker S. J et Wake W. Les design Patterns en Java : les 23 modèles de conception fondamentaux. CompusPress (2006)
- des figures et des transparents empruntés de L. Seinturier et P. Collet