Université Nice SOPHIA ANTIPOLIS

Université Nice SOPHIA ANTIPOLIS

Patrons de conception

Philippe Collet

Master 1 IFI

2013-2014

http://deptinfo.unice.fr/twiki/bin/view/Minfo/SoftEng1314

Plan

- Introduction
- Premier exemple
- Principes et classification
- Présentation des patrons les plus significatifs
- Etude de cas

Motivations

- Les besoins pour une bonne conception et du bon code :
  - Extensibilité
  - Flexibilité
  - Maintenabilité
  - Réutilisabilité
  - ☞ Les qualités *internes*
    - ☞ Meilleure spécification, construction, documentation

Motivations (suite)

- Augmenter la cohésion du code
  - La cohésion est le degré pour lequel les différentes données gérées dans une classe ou une fonction sont reliées entre elles. C'est le degré de corrélation des données entre elles.
- Diminuer le couplage
  - Le couplage est le nombre de liens entre les données de classes ou fonctions différentes. On peut faire un compte du couplage en comptant le nombre de références ou d'appels fait à l'objet d'un autre type.
- On minimise le couplage en maximisant la cohésion, et en créant des interfaces qui seront les points centraux d'accès aux autres données.

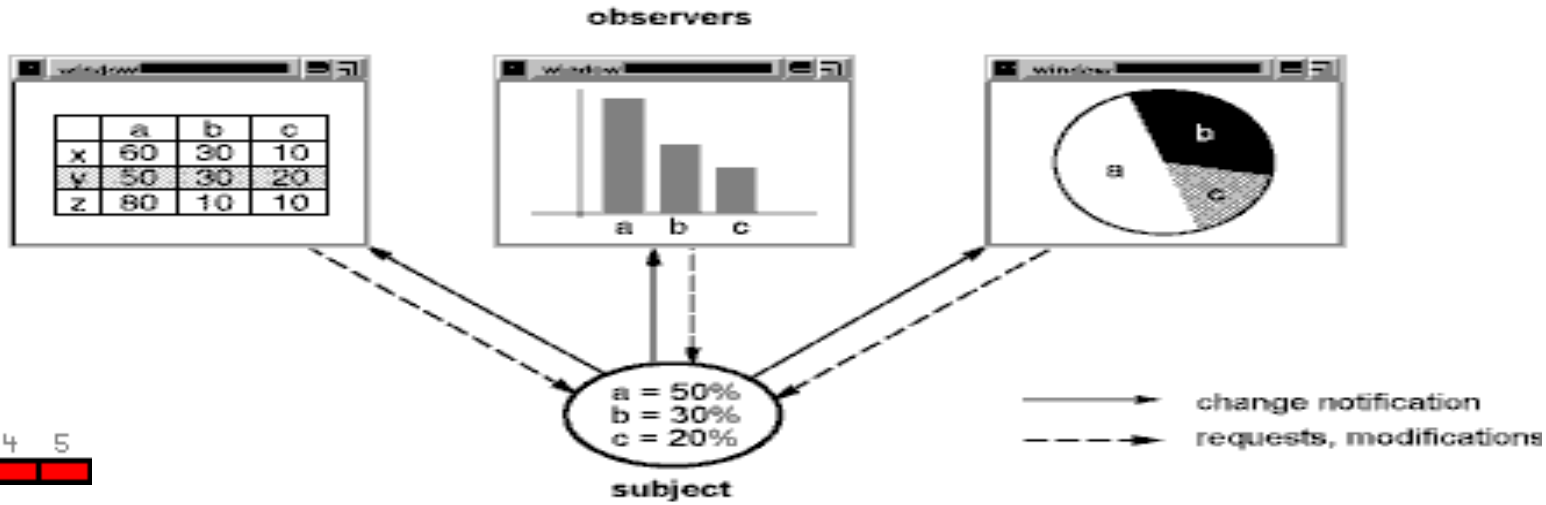
Design Pattern : principe

- 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
- *Traduction : patron de conception*
- Tous les patrons visent à renforcer la cohésion et à diminuer le couplage

Un exemple :

Observer (comportement)

- **Intention**
  - Définir une dépendance 1-N de telle façon que si l'objet change d'état tous ses dépendants sont prévenus et mis à jour automatiquement
- **Synonymes :** Dépendants, Publier/Abonner, Publish/Subscribe
- **Motivation**
  - Un effet secondaire de découper un logiciel en classes coopératives est la nécessité de maintenir la cohérence des objets associés. Le faire par des classes **fortement couplées** entrave leur réutilisabilité



Fréquence :

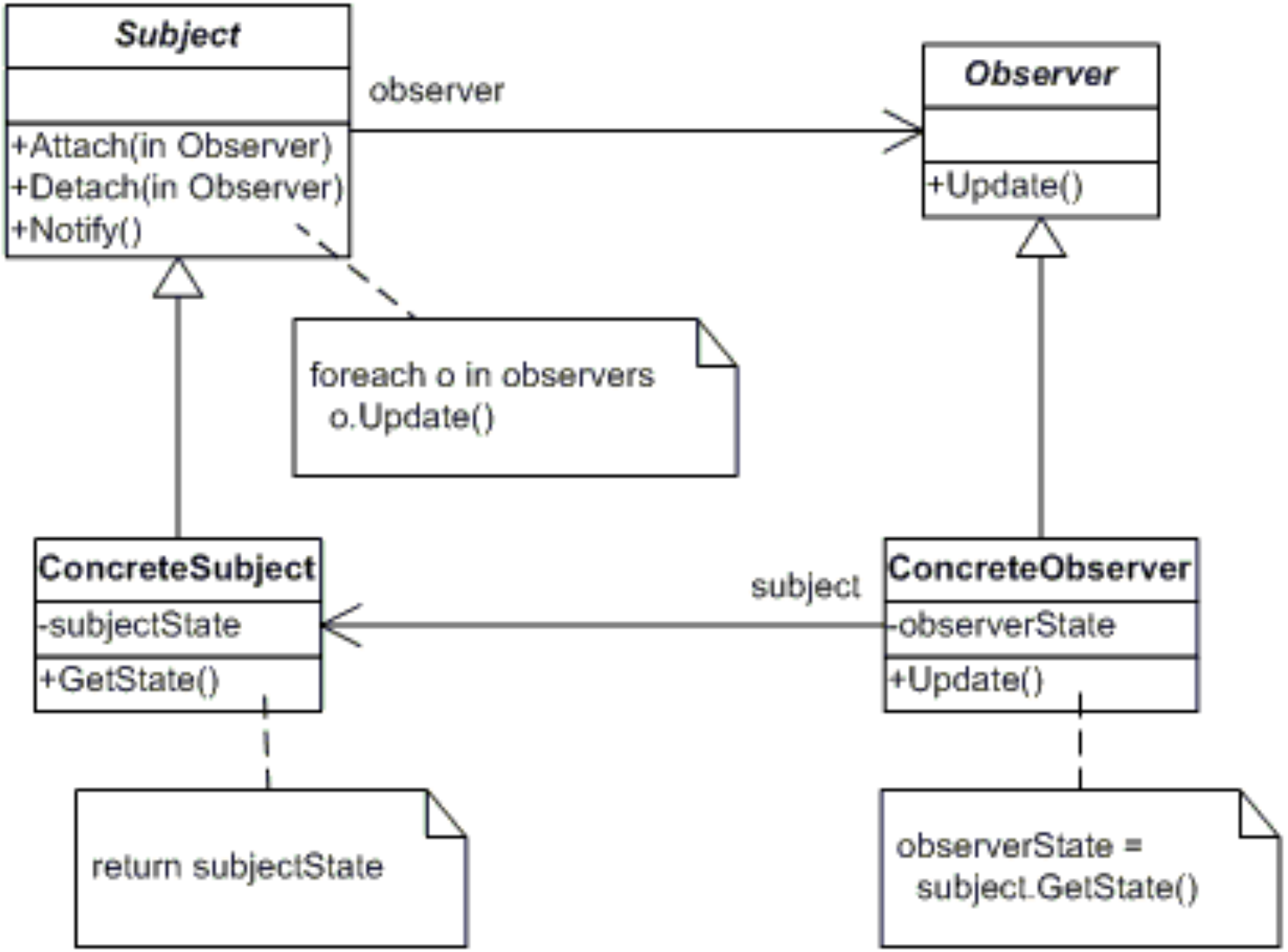
1 2 3 4 5

Observer (2)

- **Champs d'application**
  - Quand une abstraction a deux aspects, un dépendant de l'autre. Encapsuler ces aspects dans des objets séparés permet de les faire varier et de les réutiliser indépendamment
  - Quand un changement sur un objet nécessite de modifier les autres et qu'on ne peut savoir combien d'objets doivent être modifiés
  - Quand un objet doit être capable de notifier d'autres objets sans faire de suppositions sur qui sont ces objets, c'est-à-dire que les objets ne doivent pas être fortement couplés

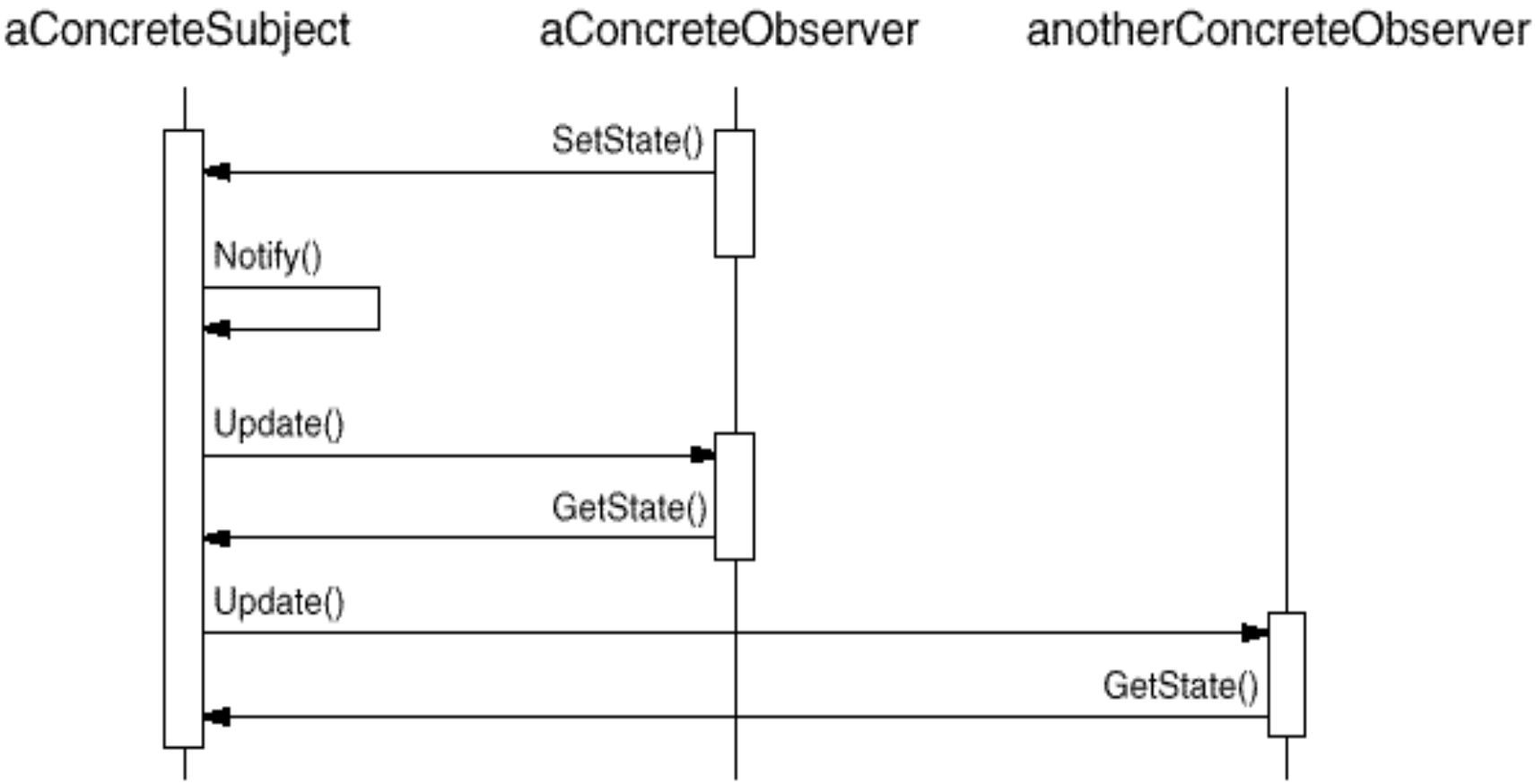
Observer (3)

- **Structure**



Observer (4)

- **Collaborations**



Observer (5)

- **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

Observer (6)

- **Implémentation**
  - Référence sur les observateurs, *hash-table*
  - Un observateur peut observer plusieurs sujets
    - Étendre le *update* pour savoir qui notifie
  - Attention lors de la suppression des observés
  - Attention à la consistance de l'observé avant notification
  - Attention aux informations de notification
- **Patterns associés**
  - Mediator, Singleton

Design Patterns : bénéfices globaux

- Standardisation de certains concepts en modélisation
- *Capture* de l'expérience de conception
- Réutilisation de solutions élégantes et efficaces vis-à-vis du problème
- Amélioration de la documentation
- Facilité de maintenance

Historique & définition

- *Gang of Four* : Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- *Design Patterns – Elements of Reusable Object-Oriented Software,* Addison Wesley, 1994

Un Design Pattern nomme, abstrait et identifie les aspects essentiels d'une structuration récurrente, ce qui permet de créer une modélisation orientée objet réutilisable

Classification des Patterns

		Objectif		
		Création	Structure	Comportement
Portée	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*

Université Nice SOPHIA ANTIPOLIS

Patrons de création

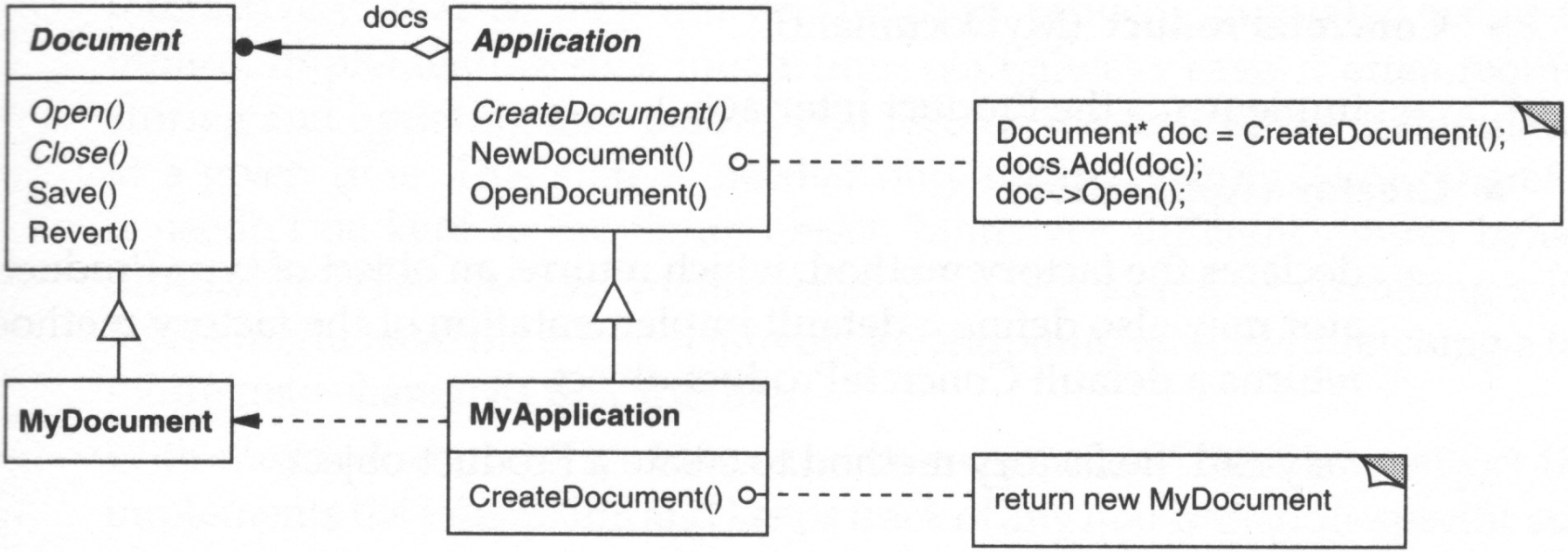
Factory Method (création)

- **Intention**
  - Définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier
  - Permet à une classe de déléguer l'instanciation à des sous-classes
- **Motivation**
  - instancier des classes, en connaissant seulement les classes abstraites
- **Synonymes**
  - Constructeur virtuel

Fréquence :

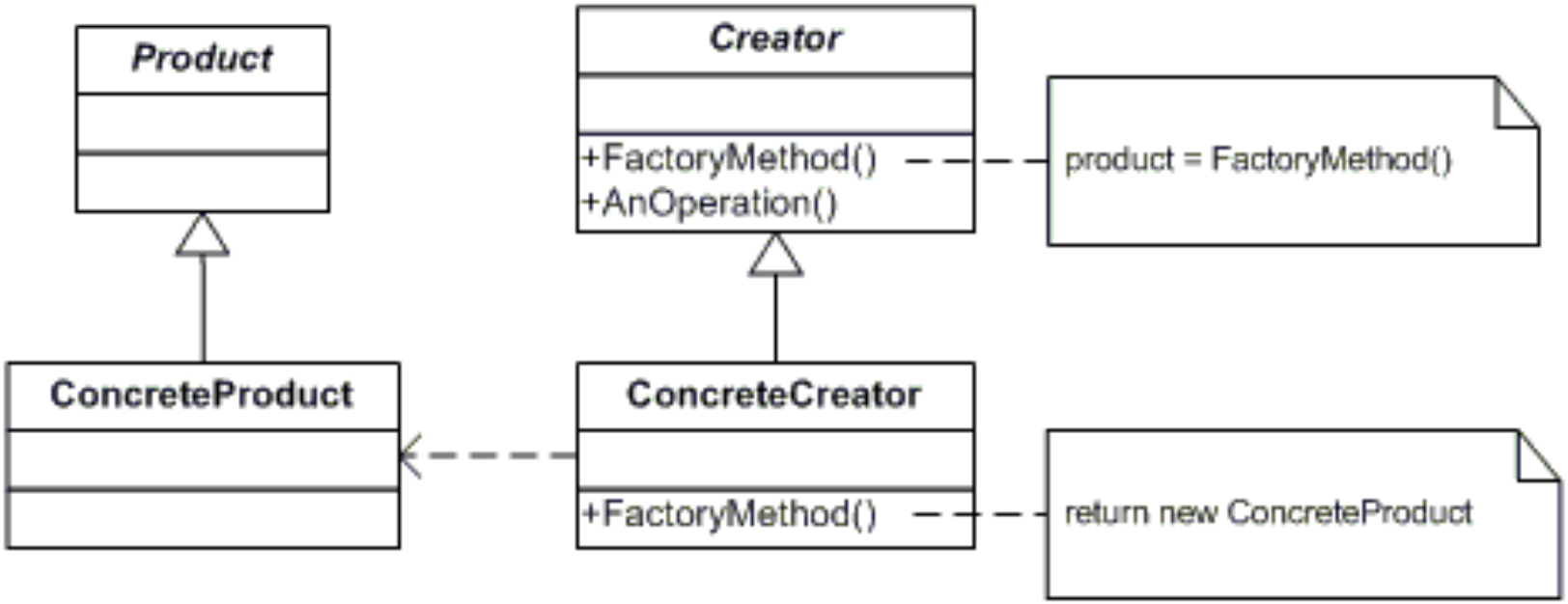
Factory Method (2)

- **Champs d'application**
  - 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



Factory Method (3)

- **Structure**



- **Participants**
  - **Product** (Document) définit l'interface des objets créés par la fabrication
  - **ConcreteProduct** (MyDocument) implémente l'interface Product

Factory Method (4)

- **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

- **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 (5)

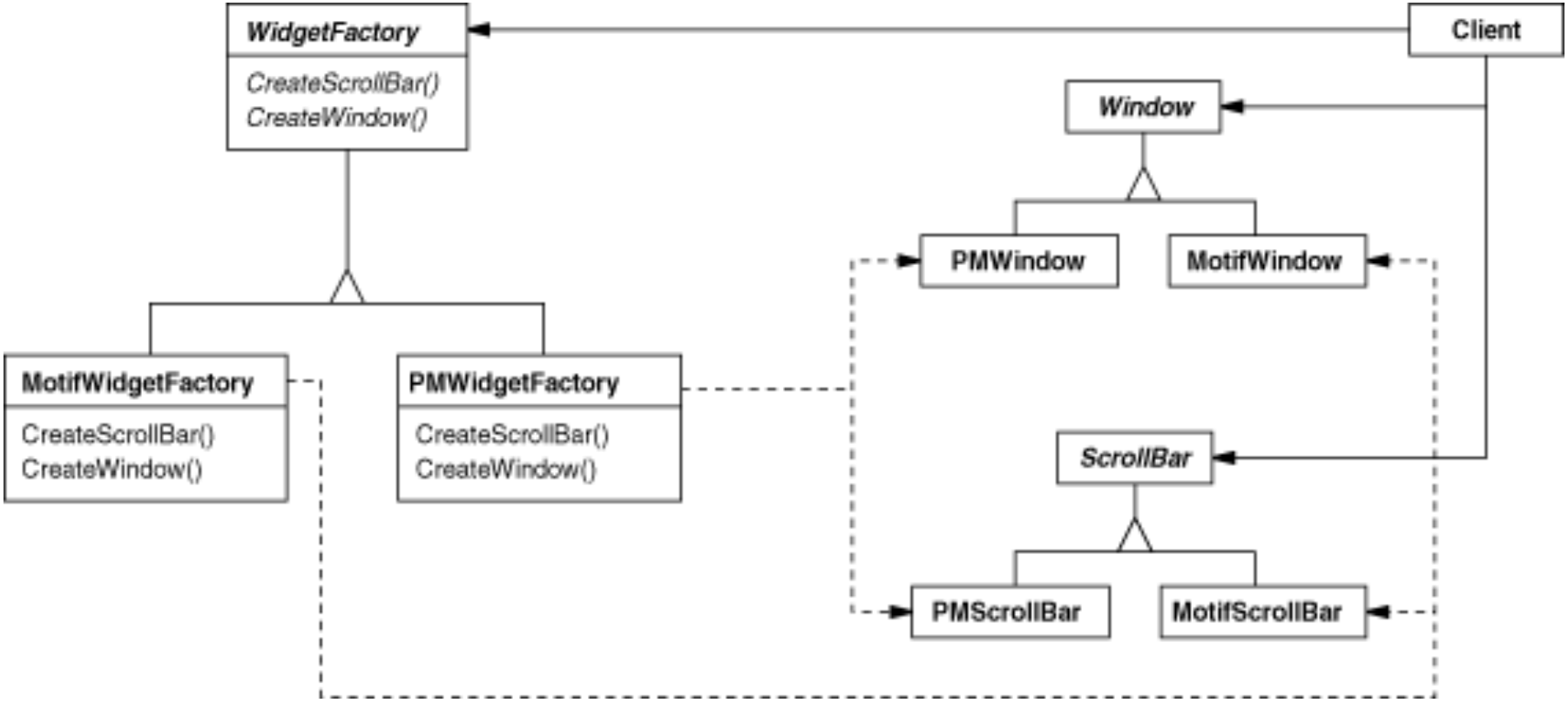
- **Implémentation**
  - 2 variantes principales :
    - La classe Creator est une classe abstraite et ne fournit pas d'implémentation pour la fabrication qu'elle déclare (les sous-classes définissent obligatoirement une implémentation)
    - La classe Creator est une classe concrète qui fournit une implémentation par défaut pour la fabrication
  - Fabrication paramétrée :
    - La fabrication utilise un paramètre qui identifie la variété d'objet à créer
- **Utilisations connues**
  - Applications graphiques, un peu partout…
- **Patterns associés**
  - Abstract Factory, Template Method, Prototype

Abstract Factory (création)

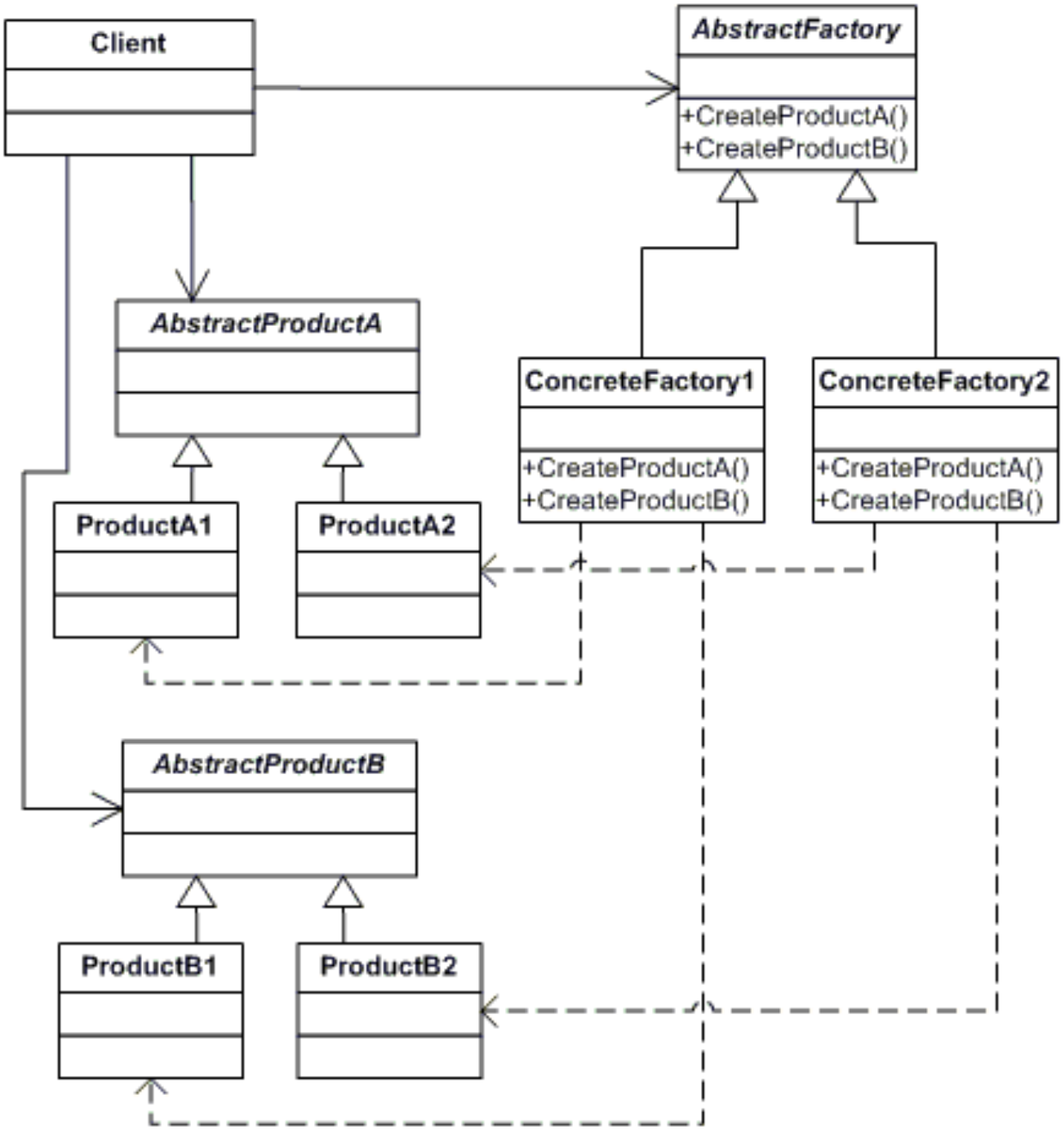
- **Intention**
  - Fournir une interface pour créer des familles d'objets dépendants ou associés sans connaître leur classe réelle
- **Synonymes :** Kit, Fabrique abstraite, Usine abstraite
- **Motivation**
  - Un éditeur qui va produire plusieurs représentations d'un document
- **Champs d'application**
  - Indépendance de comment les objets/produits sont créés, composés et représentés
  - Configuration d'un système par une instance d'une multitude de familles de produits

Abstract Factory (2)

- Conception d'une famille d'objets pour être utilisés ensemble et contrôle de cette contrainte
- Bibliothèque fournie avec seulement leurs interfaces, pas leurs implémentations



- **Structure**


Client
AbstractFactory
+CreateProductA()
+CreateProductB()
AbstractProductA
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2

Abstract Factory (4)

- **Participants**
  - *AbstractFactory* déclare l'interface pour les opérations qui créent des objets abstraits
  - *ConcreteFactory* implémente les opérations qui crée 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 déclarée par *AbstractFactory* et par les classes *AbstractProduct*

Abstract Factory (5)

- **Collaborations**
  - Normalement, 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**
  1. Isolation des classes concrètes
  2. Échange facile des familles de produit

Abstract Factory (6)

3. Encouragement de la cohérence entre les produits
4. Prise en compte difficile de nouvelles formes de produit

- **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
- **Patterns associés**
  - Singleton, Factory Method, Prototype

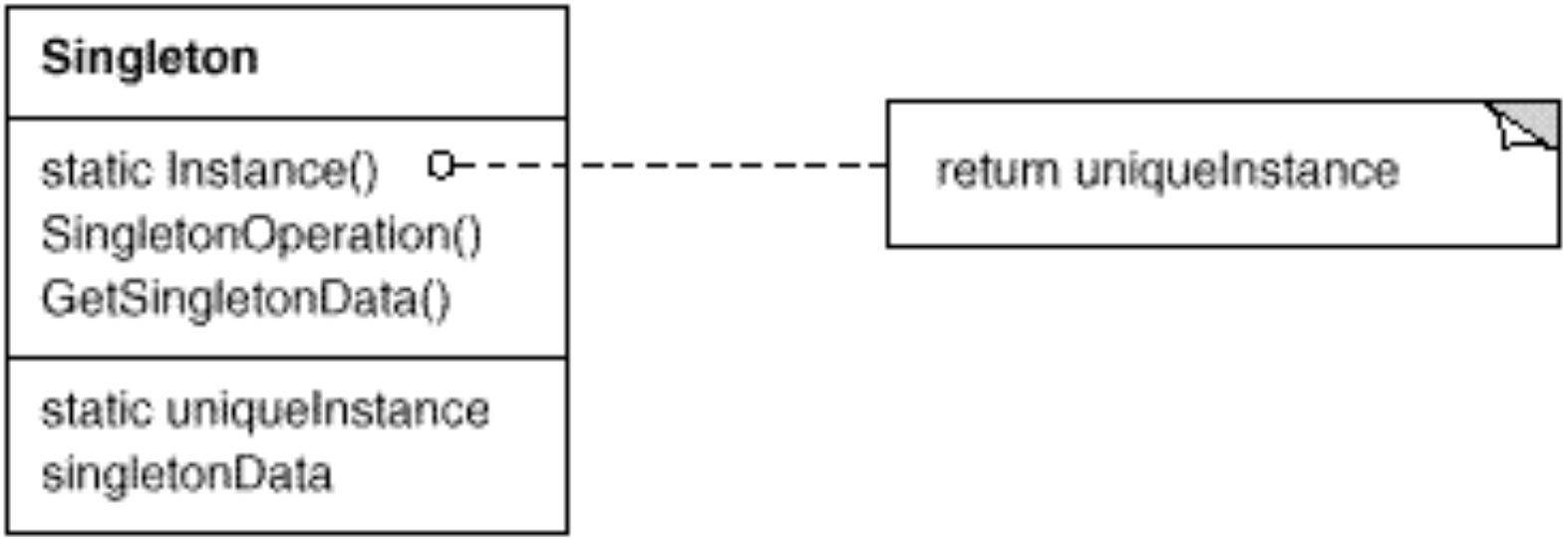
Singleton (création)

- **Intention**
  - S'assurer qu'une classe a une seule instance, et fournir un point d'accès global à celle-ci.
- **Motivation**
  - Un seul spooler d'imprimante / plusieurs imprimantes
  - Plus puissant que la variable globale
- **Champs d'application**
  - Cf. intention
  - Lorsque l'instance unique doit être extensible par héritage, et que les clients doivent pouvoir utiliser cette instance *étendue* sans modifier leur code

Fréquence :

Singleton (2)

- **Structure**



- **Participants**
  - instance() : méthode de classe pour accéder à l'instance

Singleton (3)

- **Collaborations**
  - Les clients ne peuvent accéder à l'instance qu'à travers la méthode spécifique
- **Conséquences**
  - Accès contrôlé
  - Pas de variable globale
  - Permet la spécialisation des opérations et de la représentation
  - Permet un nombre variable d'instances
  - Plus flexible que les méthodes de classe

Singleton (4)

- **Implémentation**
  - Assurer l'unicité
  - Sous-classer (demander quel forme de singleton dans la méthode instance())
- **Utilisations connues**
  - *DefaultToolkit* en AWT/Java et beaucoup de bibliothéques abstraite de *GUI*
- **Patterns associés**
  - Abstract Factory, Builder, Prototype

Autres patrons de création

- **Builder** 

  - Séparer la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction permette de créer différentes représentations de cet objet

- **Prototype** 

  - Indiquer le type des objets à créer en utilisant une instance (le prototype). les nouveaux objets sont des copies de ce prototype (clonage)
  - Principe vu en généricité dynamique

Université Nice SOPHIA ANTIPOLIS

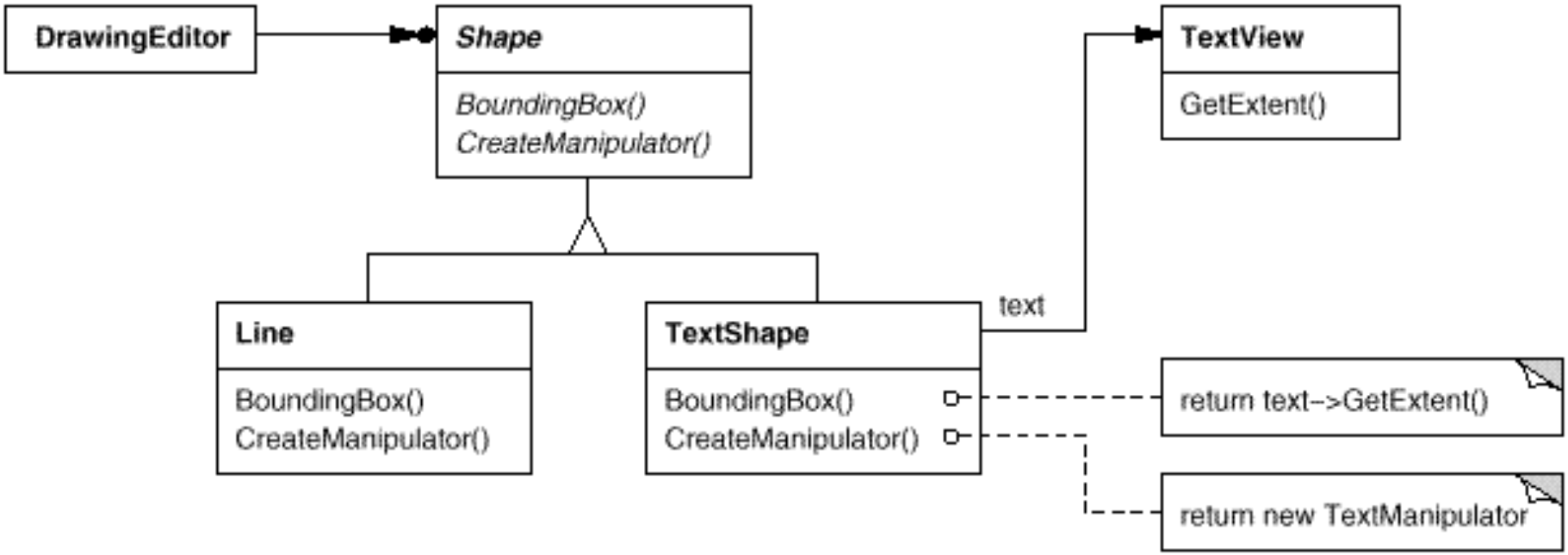
Patrons de structure

Adapter (structure)

- **Intention**
  - Convertir l'interface d'une classe en une autre interface qui est attendue par un client.
  - Permet de faire collaborer des classes qui n'auraient pas pu le faire à cause de l'incompatibilité de leurs interfaces
- **Synonymes :** Wrapper, Mariage de convenance
- **Motivation**
  - Une classe de bibliothèque conçue pour la réutilisation ne peut pas l'être à cause d'une demande spécifique de l'application

Fréquence :

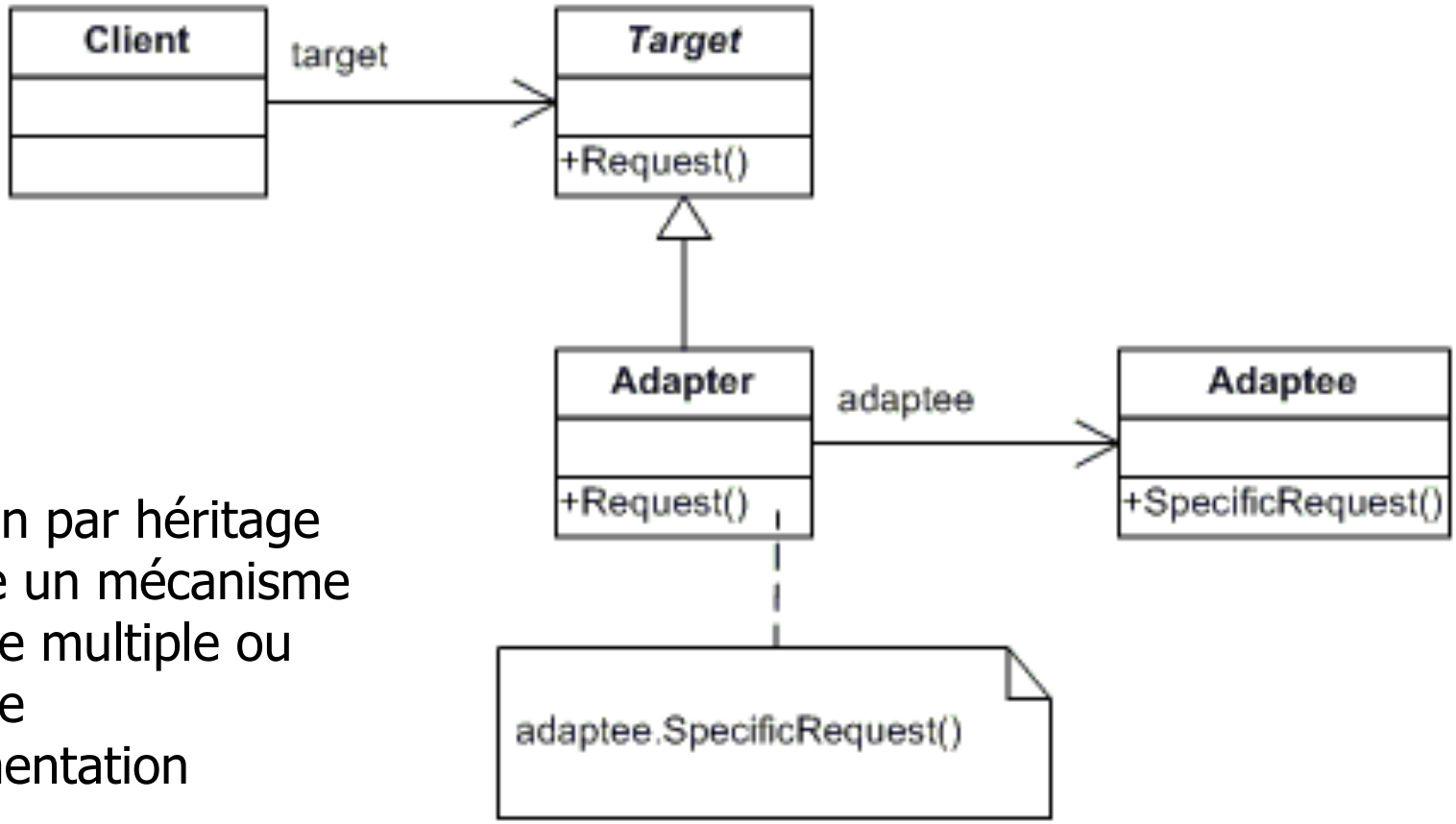
Adapter (2)



- **Champs d'application**
  - Volonté d'utiliser une classe, même si l'interface ne convient pas
  - Création d'une classe qui va coopérer par la suite…

Adapter (3)

- **Structure (version par délégation)**



  - La version par héritage nécessite un mécanisme d'héritage multiple ou d'héritage d'implémentation

Adapter (4)

- **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 (5)

- **Collaborations**
  - Le client appèle les méthodes sur l'instance d'Adapter. Ces méthodes appellent alors les méthodes d'Adaptee pour réaliser le service

- **Conséquences (adapter *objet*)**
  1. Un adapter peut travailler avec plusieurs Adaptees
  2. Plus difficile de redéfinir le comportement d'Adaptee (sous-classer puis obliger Adapter a référencer la sous-classe)

Adapter (6)

- **Conséquences (adapter *classe*)**
  1. Pas possible d'adapter une classe et ses sous-classes
  2. Mais redéfinition possible du comportement (sous-classe)

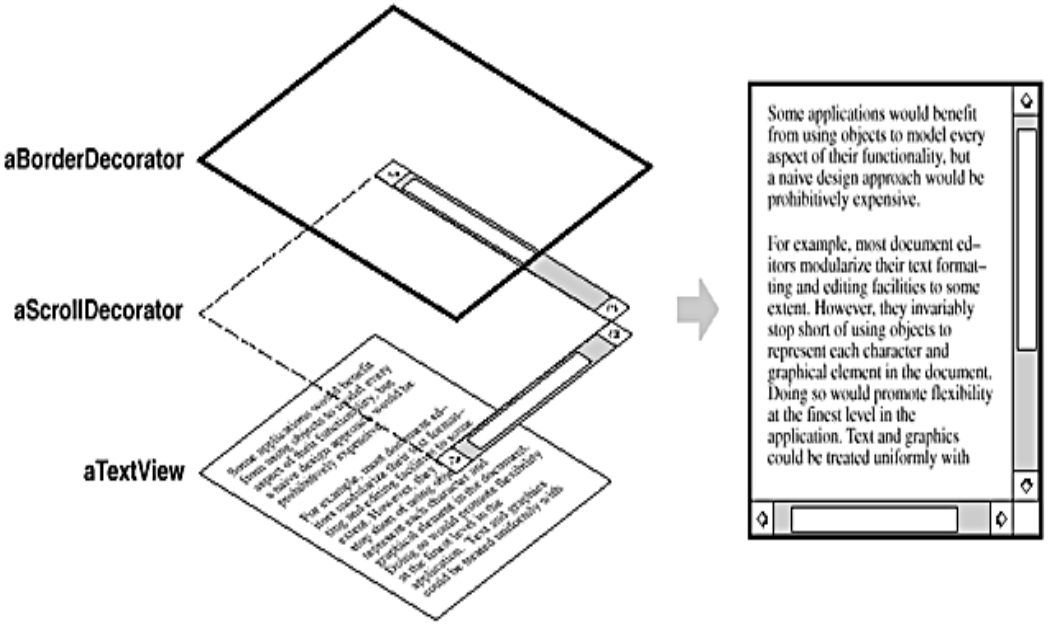
- **Implémentation**
  - En Java, utilisation combinée de extends/implements pour la version à classe
- **Patterns associés**
  - Bridge, Decorator, Proxy

Decorator (structure)

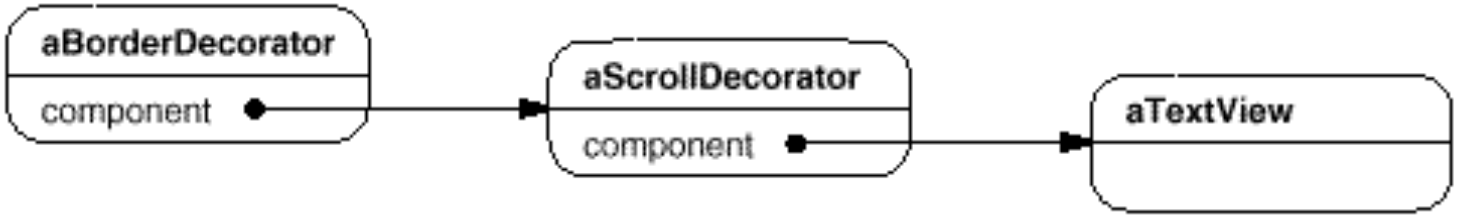
- **Intention**
  - Attacher dynamiquement des capacités additionnelles à un objet
  - Fournir ainsi une alternative flexible à l'héritage pour étendre les fonctionnalités
- **Synonymes :** Wrapper (attention !)
- **Motivation**
  - Ajout de capacités pour objet individuellement et dynamiquement
  - Englober l'objet existant dans un autre objet qui ajoute les capacités (plus que d'hériter)

Fréquence :

Decorator (2)

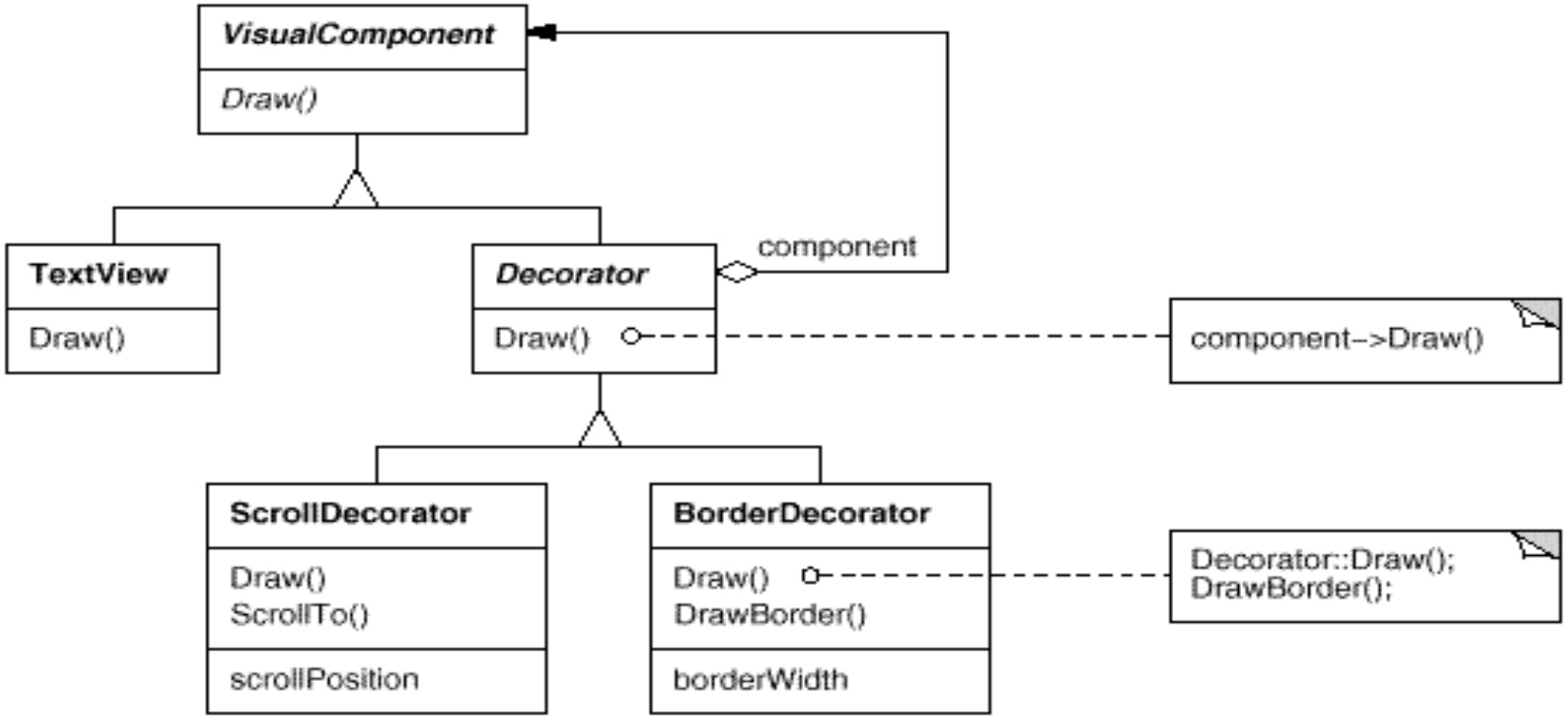


- **Champs d'application**
  - Pour ajouter des capacités de manière transparente
  - Pour des capacités qui peuvent être retirées



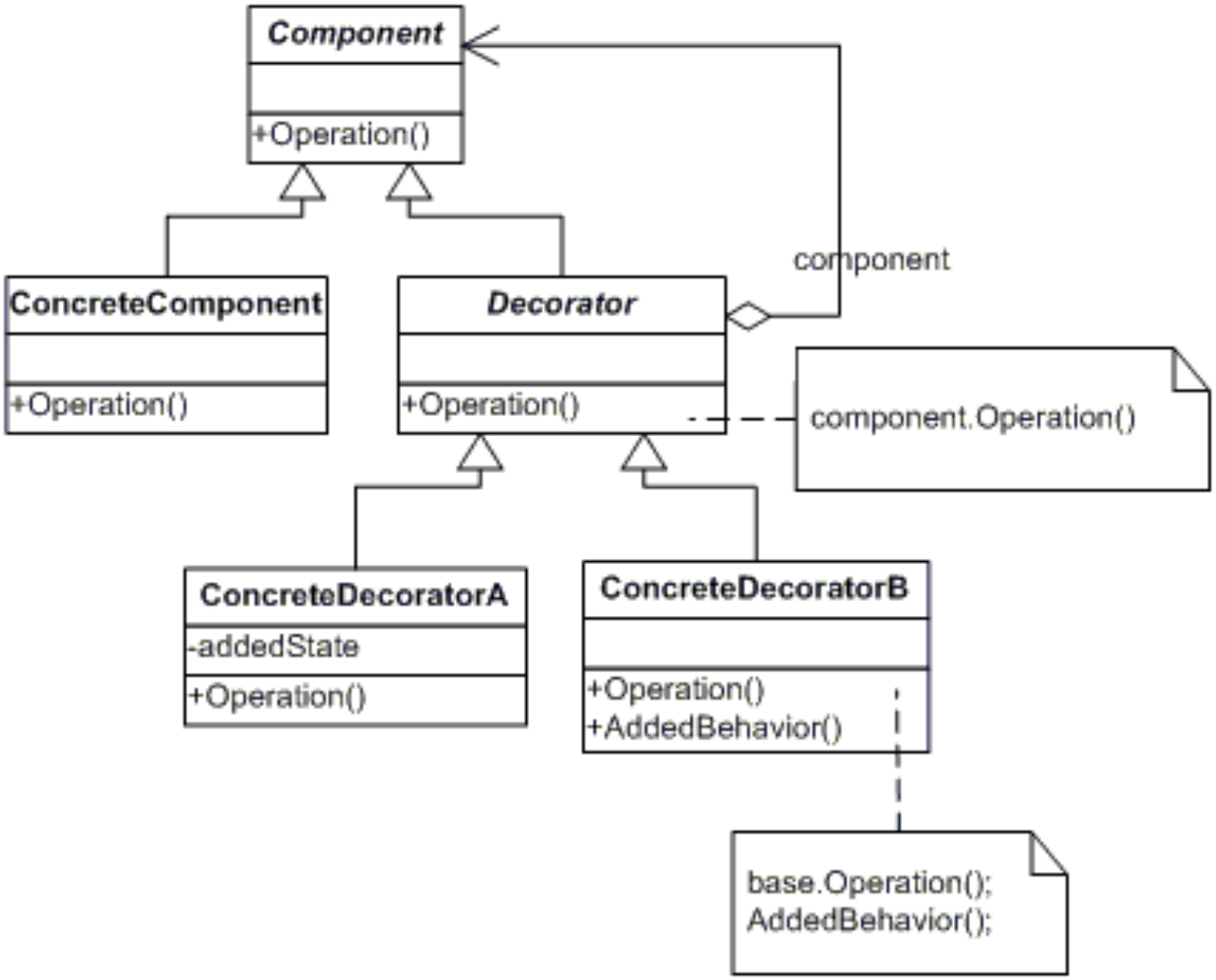
Decorator (3)

- Quand l'extension par héritage produirait un nombre trop important d'extensions indépendantes
- Quand l'héritage est interdit



Decorator (4)

- **Structure**



Decorator (5)

- **Participants**
  - **Component** (VisualComponent) définit l'interface des objets qui peuvent avoir des capacités dynamiques
  - **ConcreteComponent** (TextView) définit l'objet auquel on peut attacher de nouvelles capacités
  - **Decorator** maintient une référence à l'objet Component et définit une interface conforme à ce même Component
  - **ConcreteDecorator** (BorderDecorator, ScrollDecorator) ajoute les capacités à component
- **Collaborations**
  - Decorator transmets les demandes à son objet Component. Il peut aussi effectuer du travail en plus avant et après la transmission

Decorator (6)

- **Conséquences**
  1. Plus de flexibilité que l'héritage
  2. Évite la surcharge inutile de classe en service en haut de la hiérarchie *(PAY AS YOU GO)*
  3. Le décorateur et son composant ne sont pas identiques (identité)
  4. Plein de petits objets
- **Implémentation**
  - Java : utilisation d'interface pour la conformité
  - Pas forcément besoin d'un décorateur abstrait

Decorator (7)

  - Maintenir une classe de base légère
  - Decorator est fait pour le changement d'aspect, Strategy est fait pour le changement radical d'approche

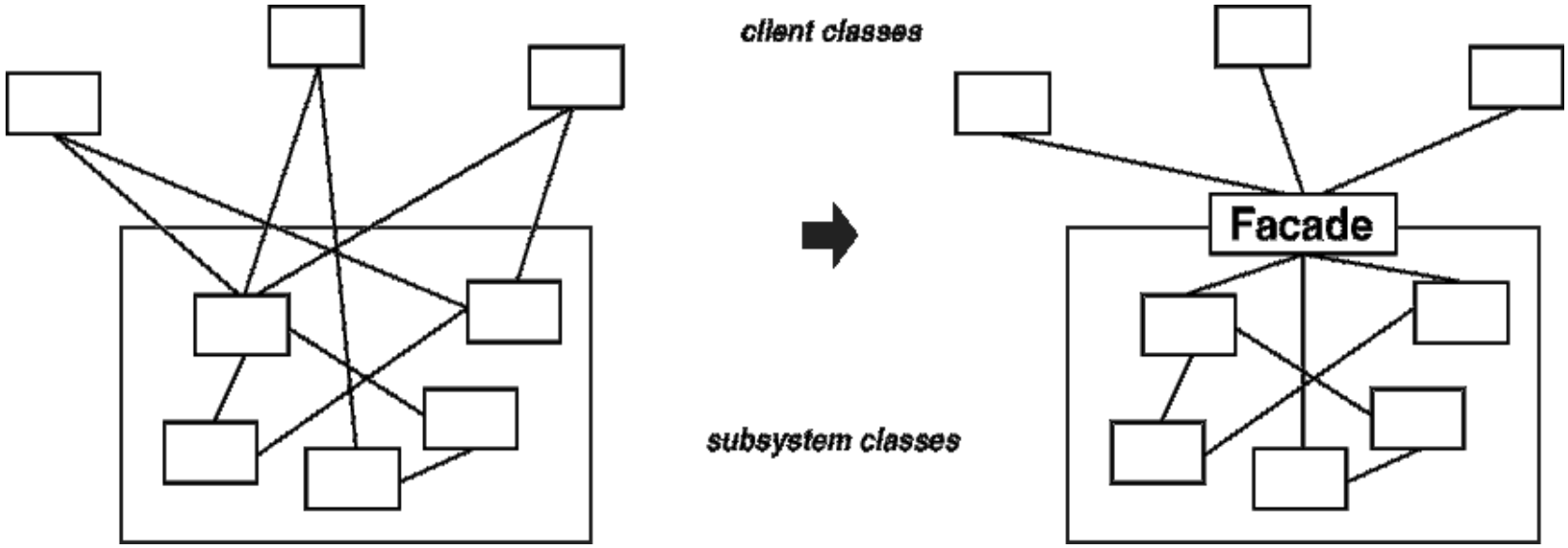
- **Utilisations connues**
  - Organisation de java.io
- **Patterns associés**
  - Adapter, Composite, Strategy

Façade (structure)

- **Intention**
  - Fournir une interface unique, simplifiée ou unifiée, pour accéder à un ensemble d'interfaces d'un sous-système complexe.
- **Motivation**
  - Réduire la complexité d'un système en le découpant en plusieurs sous-systèmes
  - Eviter la dépendance entre les clients et les éléments du sous-système

Fréquence :

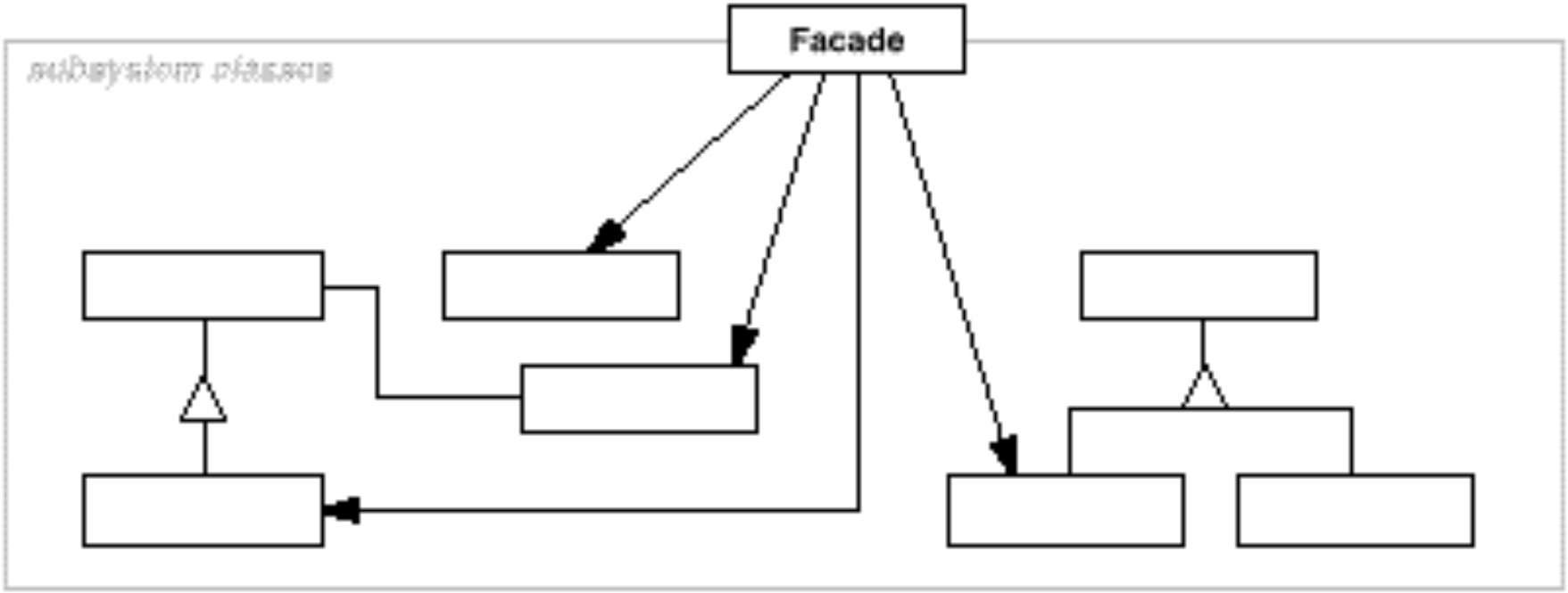
Façade (2)



- **Champs d'application**
  - Fournir une interface unique pour un système complexe
  - Séparer un sous-système de ses clients
  - Découper un système en couches (une façade par point d'entrée dans chaque couche)

Façade (3)

- **Structure**



Façade (4)

- **Participants**
  - La Façade connaît quelles classes du sous-système sont responsables de telle ou telle requête, et délègue donc les requêtes aux objets appropriés
  - Les classes sous-jacentes à la façade implémentent les fonctionnalités

    Le nombre de classes n'est pas limité
- **Collaborations**
  - Le client manipule le sous-système en s'adressant à la façade (ou aux éléments du sous-système rendus publics par la façade)
  - La façade transmet les requêtes au sous-système après transformation si nécessaire

Façade (5)

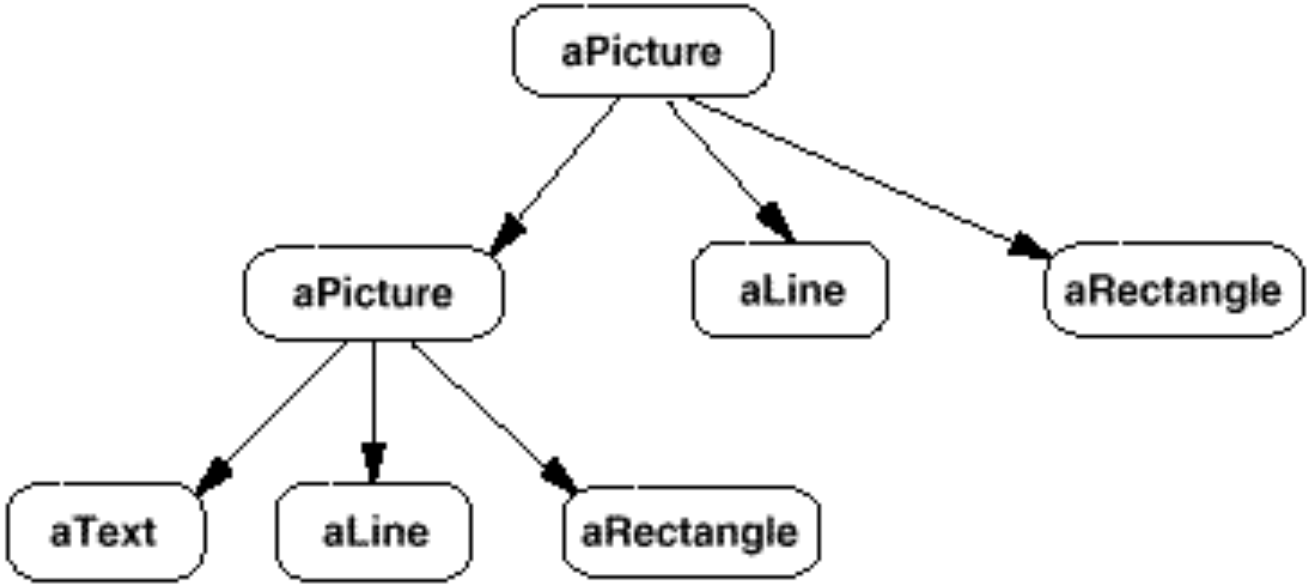
- **Conséquences**
  1. Facilite l'utilisation par la simplification de l'interface
  2. Diminue le couplage entre le client et le sous-système
  3. Ne masque pas forcément les éléments du sous-système (un client peut utiliser la façade ou le sous-système)
  4. Permet de modifier les classes du sous-système sans affecter le client
  5. Peut masquer les éléments privés du sous-système
  6. L'interface unifiée présentée par la façade peut être trop restrictive

Façade (6)

- **Implémentation**
  - Possibilité de réduire encore plus le couplage en créant une façade abstraite et des versions concrètes
  - Les objets de façade sont souvent des singletons
- **Utilisations connues**
  - JDBC…
- **Patterns associés**
  - Abstract Factory, Mediator, Singleton

Composite (structure)

- **Intention**
  - Composer des objets dans des structures d'arbre pour représenter des hiérarchies composants/composés
  - *Composite* permet au client de manipuler uniformément les objets simples et leurs compositions

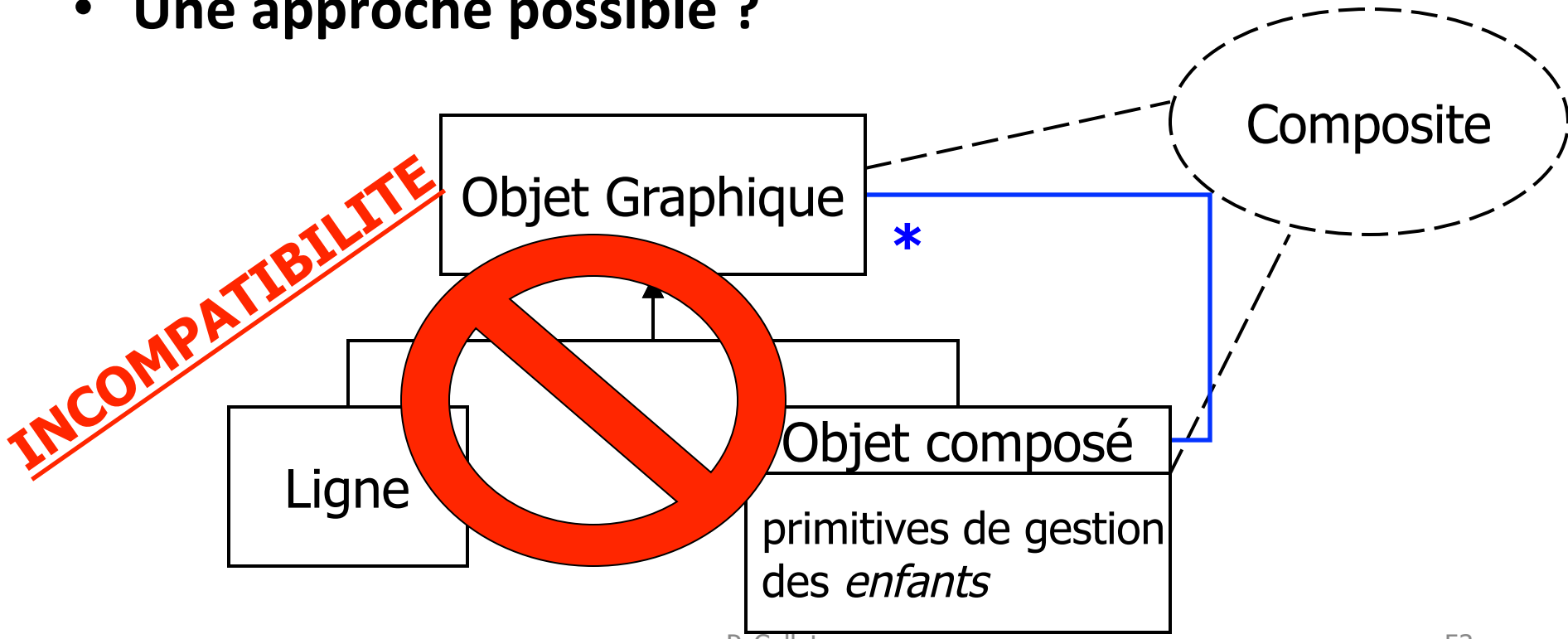


Fréquence :

1 2 3 4 5

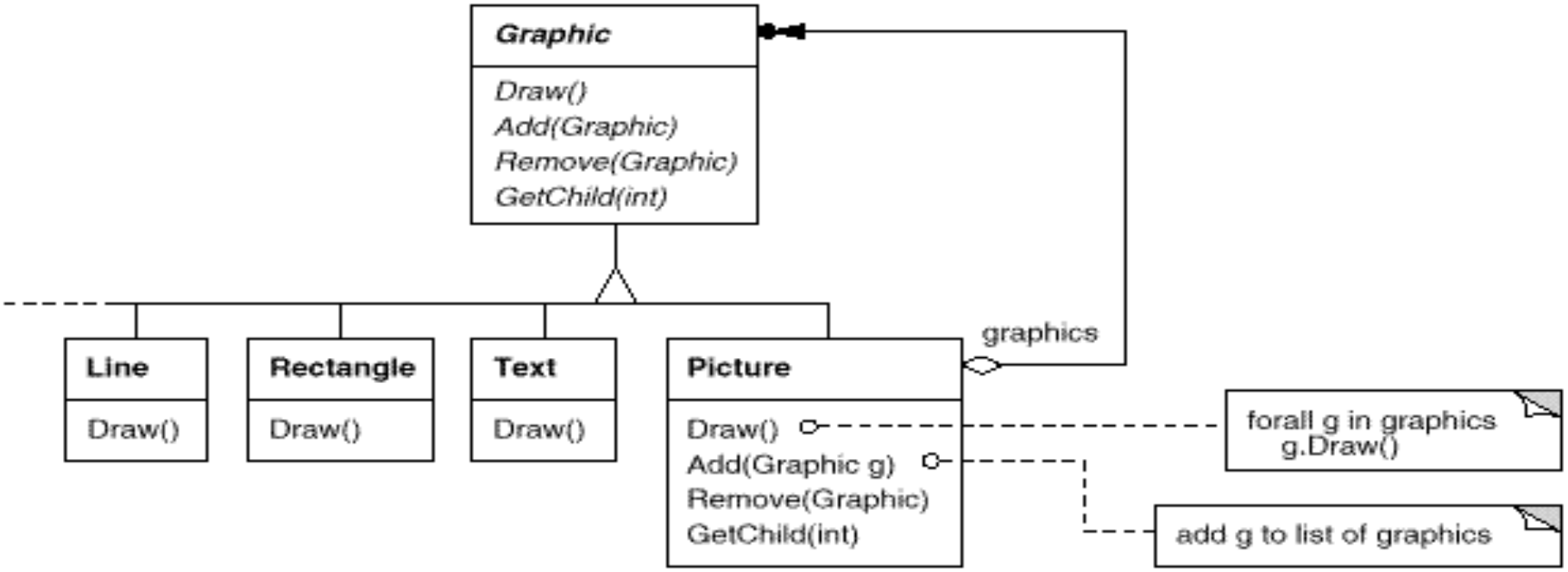
Composite (2)

- **Motivation**
  - Une classe abstraite qui représente à la fois les primitives et les containers
- **Une approche possible ?**

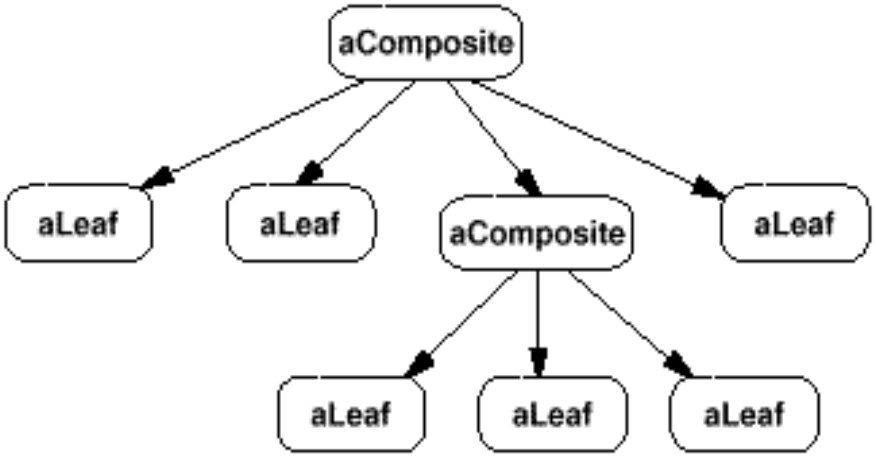


Composite (3)

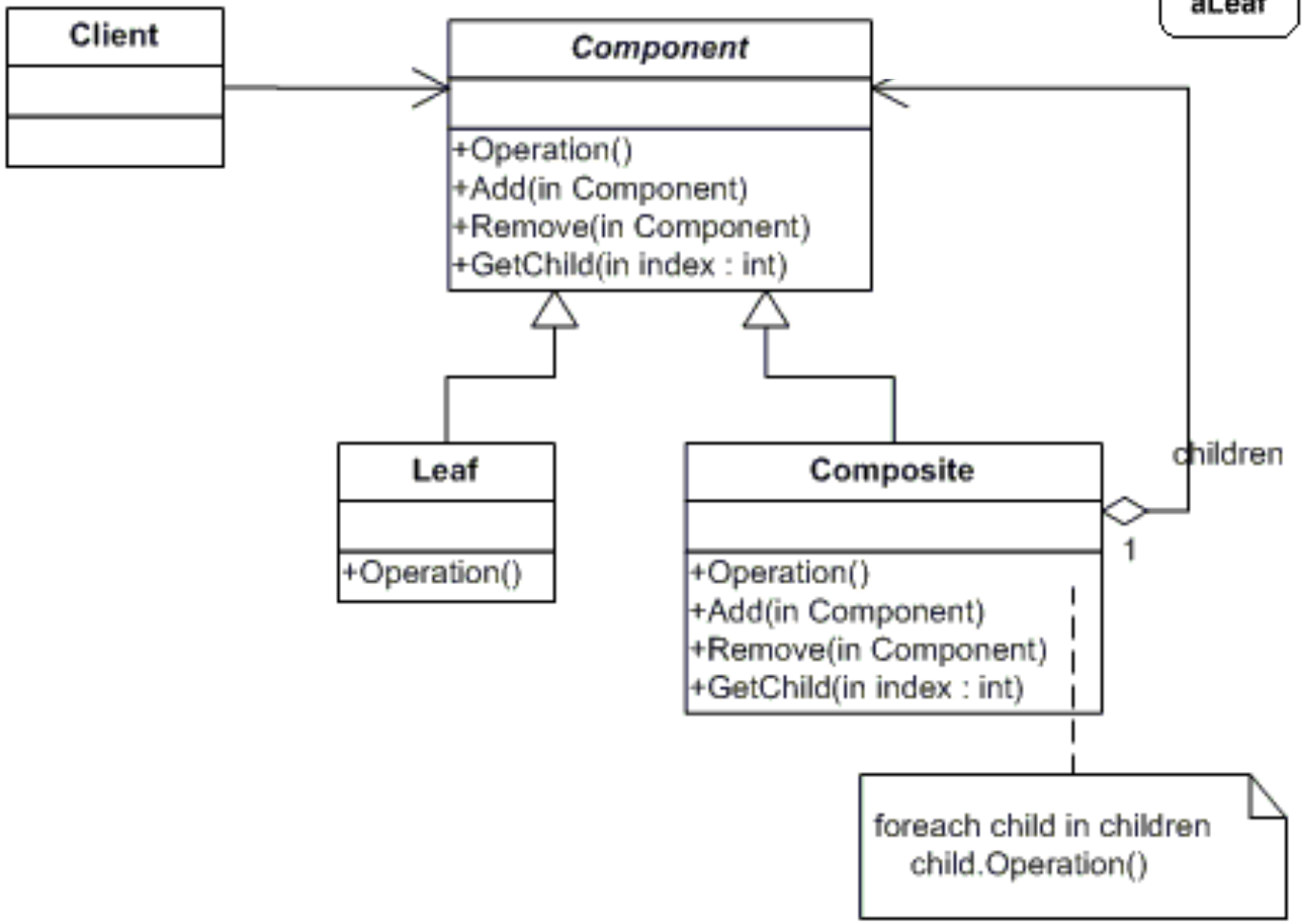
- **Champs d'application**
  - Représentation de hiérarchie composants/composés
  - Les clients doivent ignorer la différence entre les objets simples et leurs compositions (uniformité apparente)



Composite



- **Structure**



Composite (5)

- **Participants**
  - **Component** (Graphic)
    - déclare l'interface commune à tous les objets
    - implémente le comportement par défaut pour toutes les classes si nécessaire
    - déclare l'interface pour gérer les composants *fils*
    - Définit l'interface pour accéder au composant *parent* **(optionnel)**
  - **Leaf** (Rectangle, Line, etc.) représente une feuille et définit le comportement comme tel
  - **Composite** (Picture) définit le comportement des composants ayant des fils, stocke les fils et implémente les opérations nécessaires à leur gestion
  - **Client** manipule les objets à travers l'interface Component

Composite (6)

- **Collaborations**
  - Les clients utilise l'interface Component, si le receveur est une feuille la requête est directement traitée, sinon le Composite retransmet habituellement la requête à ses fils en effectuant éventuellement des traitements supplémentaires avant et/ou après
- **Conséquences**
  - Structure **hiérarchique**, **simple**, **uniforme**, **général** et **facile à étendre** pour de nouveaux objets

Composite (7)

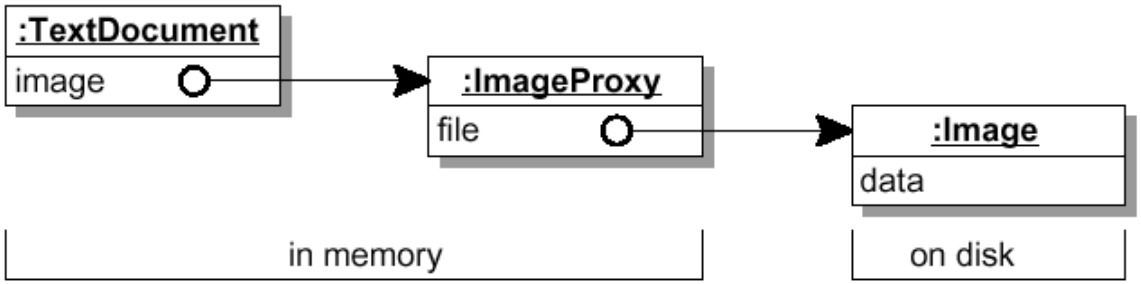
- **Implémentation**
  - Référence explicite aux parents ?
  - Partage des composants
  - Maximiser l'interface de *Component*
  - Déclaration des opérations de gestion des fils
  - Pas de liste de composants dans *Component*
  - Ordonnancement des fils ➡ Iterator
- **Utilisations connues :** Partout !
- **Patterns associés**
  - Chain of Responsibility, Decorator, Flyweight, Iterator, Visitor

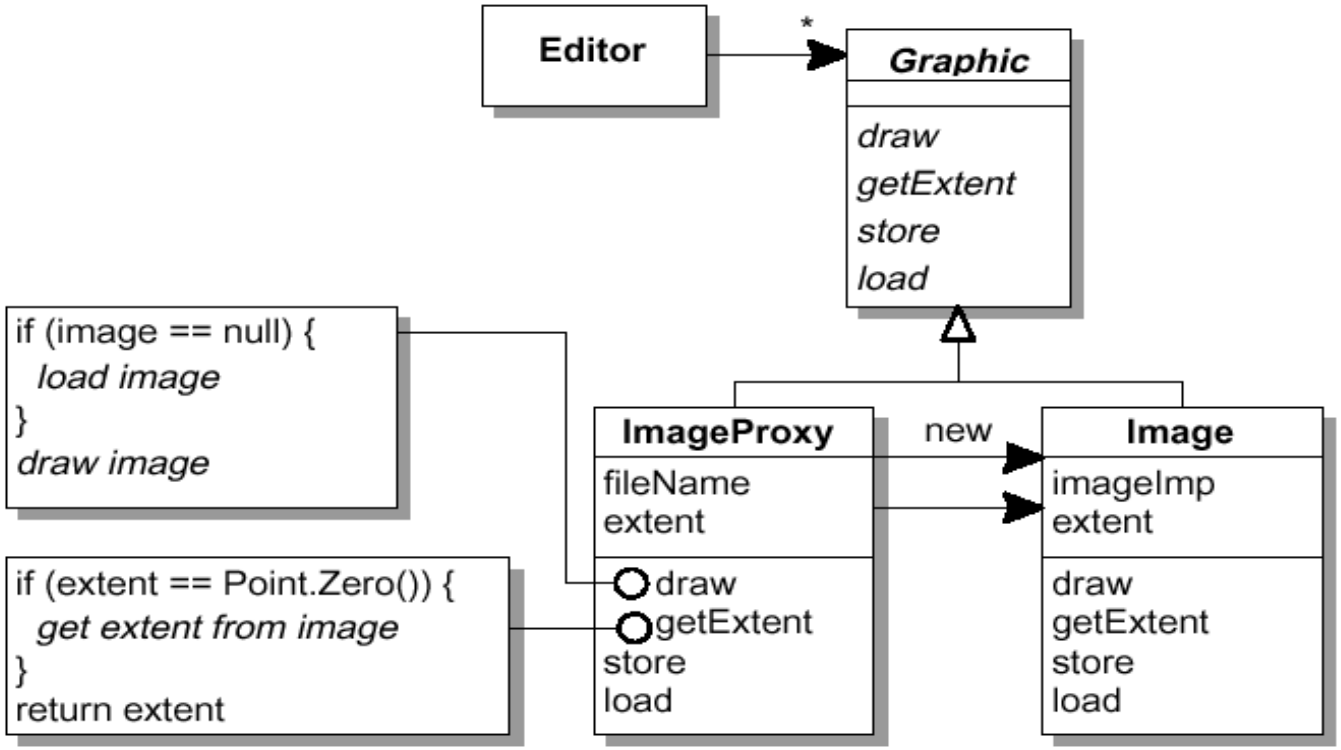
Proxy (structure)

- **Intention**
  - Fournir un substitut afin d'accéder à un autre objet souvent inaccessible.
  - Séparer l 'interface de l'implémentation
- **Synonyme**
  - Procuration, mandat, *surrogate*
- **Motivation**
  - Si un objet, comme une image volumineuse, met beaucoup de temps à se charger
  - Si un objet est situé sur une machine distante
  - Si un objet a des droits d'accès spécifiques

Fréquence :

Proxy (2)


:TextDocument
image
:ImageProxy
file
:Image
data
in memory
on disk

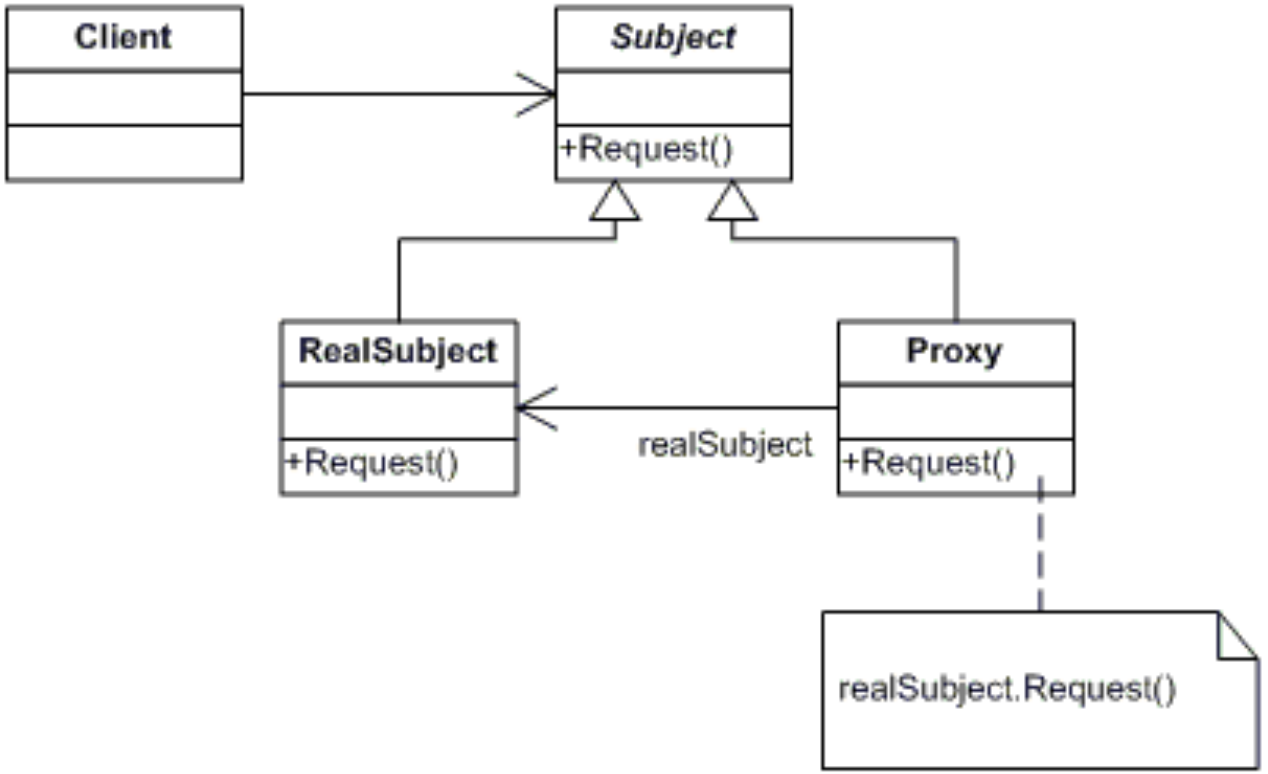

Editor
*
Graphic
draw
getExtent
store
load
if (image == null) {
load image
}
draw image
if (extent == Point.Zero()) {
get extent from image
}
return extent
ImageProxy
fileName
extent
draw
getExtent
store
load
new
Image
imageImp
extent
draw
getExtent
store
load

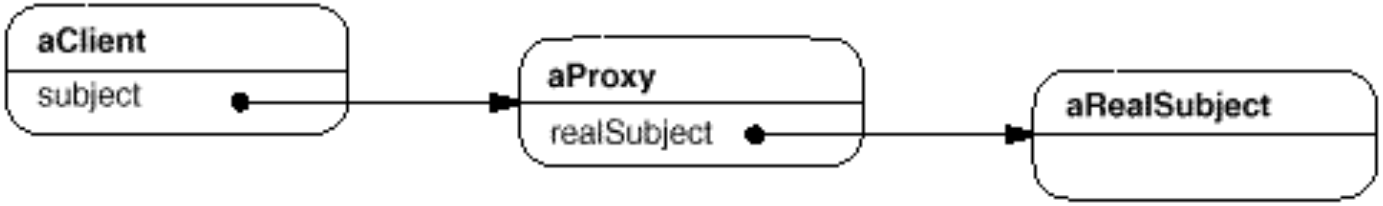
Proxy (3)

- **Champs d'application**
  - Dès qu'il y a un besoin de référencement sophistiqué ou polyvalent, autre qu'un simple pointeur
  - **Remote Proxy** est un représentant d'un objet situé dans un autre espace d'adressage
  - **Virtual Proxy** crée des objets « coûteux » à la demande
  - **Access Proxy** contrôle l'accès à un objet
  - **SmartReference** effectue un travail supplémentaire lors de l'accès à l'objet
    - Comptage de références (*smart pointers)*
    - Chargement d'objets persistants
    - Vérification de non verrouillage

Proxy (4)

- **Structure**



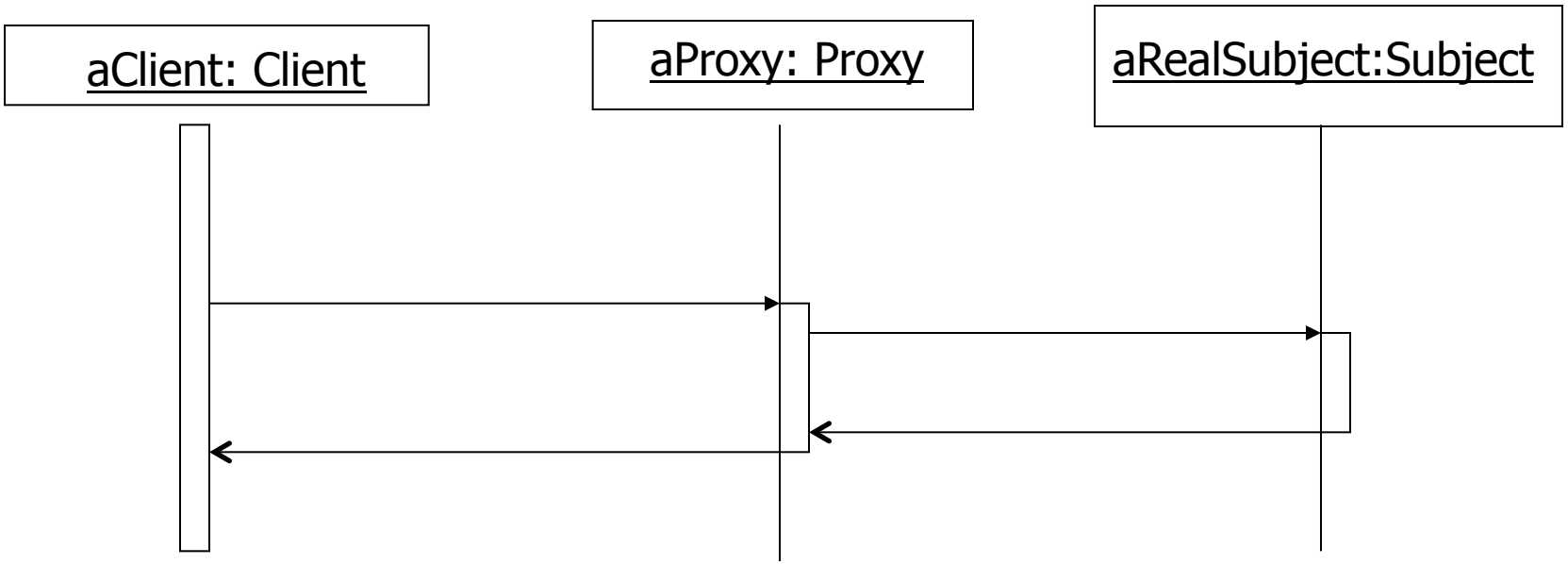


Proxy (5)

- **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

Proxy (6)

- **Collaborations**
  - Le Proxy *retransmet les requêtes au RealSubject lorsque c'est nécessaire, selon la forme du Proxy*



Proxy (7)

- **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…

Proxy (8)

- **Utilisations connues**
  - Utilisation courante (systématique…) pour faire apparaître un objet comme étant local dans les applications distribuées (CORBA, Java RMI)
- **Patterns associés**
  - Adapter, Decorator
  - Access Proxy, Remote Proxy, Virtual Proxy, SmartReference

Autres patrons de structure

- **Bridge** 

  - Découple l'abstraction de l'implémentation afin de permettre aux deux de varier indépendamment
  - Partager une implémentation entre de multiples objets
  - En Java, programmation par deux interfaces
- **Flyweight** 

  - Utiliser une technique de partage qui permet la mise en œuvre efficace d'un grand nombre d'objets de fine granularité
  - Distinction entre état intrinsèque et état extrinsèque

Université Nice SOPHIA ANTIPOLIS

Patrons de comportement

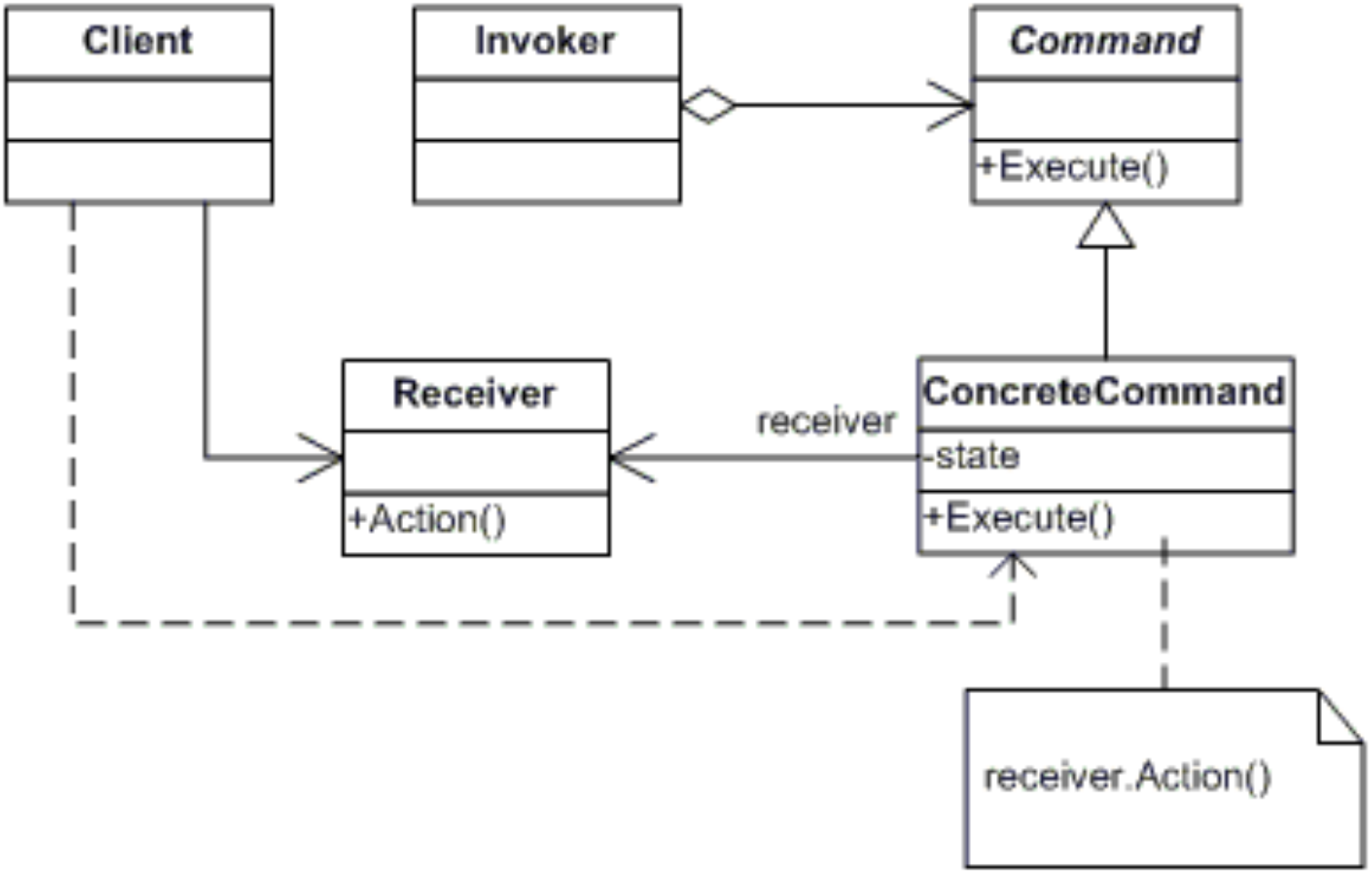
Command (comportement)

- **Intentions**
  - Encapsuler une requête comme un objet
  - Permettre de *défaire* des traitements (undo)
- **Synonymes**
  - Action, Transaction
- **Motivations**
  - Découpler les objets qui invoquent une action de ceux qui l'exécutent
  - Réaliser un traitement sans avoir besoin de savoir de quoi il s'agit et de qui va l'effectuer

Fréquence :

Command (2)

- **Structure**

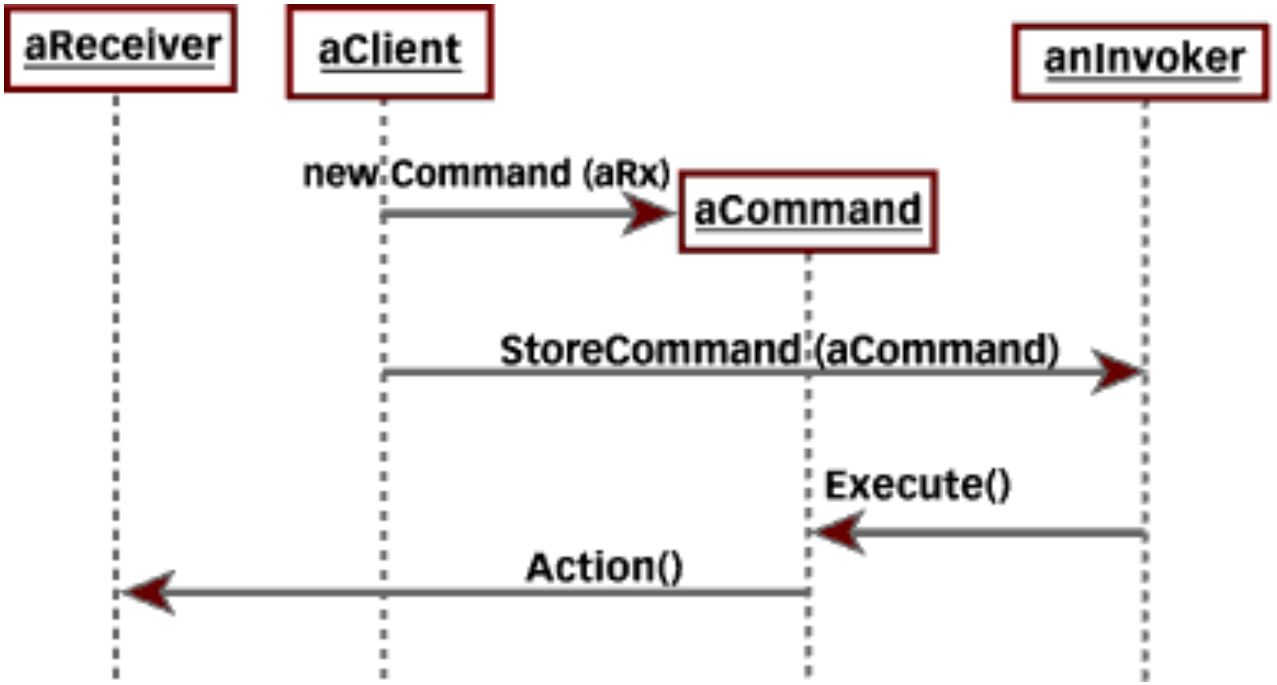


Command (3)

- **Participants**
  - **Command** déclare une interface pour exécuter une opération
  - **ConcreteCommand** définit une astreinte entre l'objet Receiver et l'action concrétise execute() en invoquant la méthode correspondante de l'objet Receiver
  - **Client** crée l'objet ConcreteCommand et positionne son Receiver
  - **Invoker** demande à l'objet Command d'entreprendre la requête
  - **Receiver** sait comment effectuer la ou les opérations associées à la demande

Command (4)

- **Champs d'application**
  - Défaire des requêtes (undo)
  - Paramétrer les objets par des actions
  - Manipuler les requêtes, les exécuter à différents moments
- **Collaboration**



Command (5)

- **Conséquences**
  - Découplage entre invocation et réalisation
  - Commandes manipulées comme n'importe quel autre objet
  - Assemblage composite => MacroCommande
  - Facilité d'ajouts de nouvelles commandes
- **Implémentation**
  - Pour supporter les undo et redo, il faut définir les opérations correspondantes et la classe ConcreteCommand doit stocker, pour cela, certaines informations
  - Un historique des commandes doit être conservé pour réaliser les undo à plusieurs niveaux
- **Patterns associés**
  - Composite, Memento, Prototype

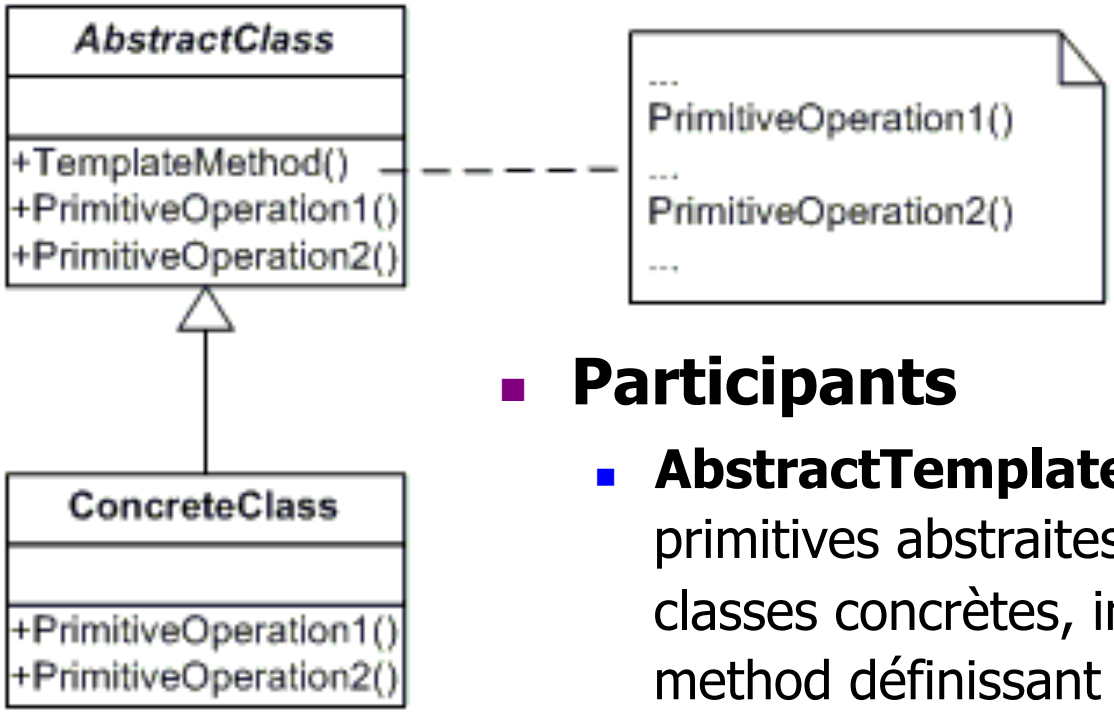
Template Method (comportement)

- **Intention**
  - Définir le squelette d'un algorithme dans une opération, et laisser les sous-classes définir certaines étapes
  - Permet de redéfinir des parties de l'algorithme sans avoir à modifier celui-ci
- **Motivation**
  - Faire varier le comportement de chaque opération de façon indépendante dans les sous-classes
- **Champs d'application**
  - Implanter une partie invariante d'un algorithme
  - Partager des comportements communs d'une hiérarchie de classes

Fréquence :

Template Method (2)

- **Structure**



- **Participants**
  - **AbstractTemplate** définit les opérations primitives abstraites implémentées par les sous-classes concrètes, implémente un template method définissant la trame d'un algorithme
  - **ConcreteTemplate** implémente les opérations primitives pour exécuter les étapes spécifiques de l'algorithme

Template Method (3)

- **Conséquences**
  - Réutilisation du code
  - Structure de contrôle inversé
  - Permet d'imposer des règles de surcharge
  - Mais il faut sous-classer pour spécialiser le comportement
- **Implémentation**
  - Réduction du nombre des opérations primitives
  - Convention de nomenclature (préfix do- ou faire-)
- **Patterns associés**
  - Factory Method, Strategy

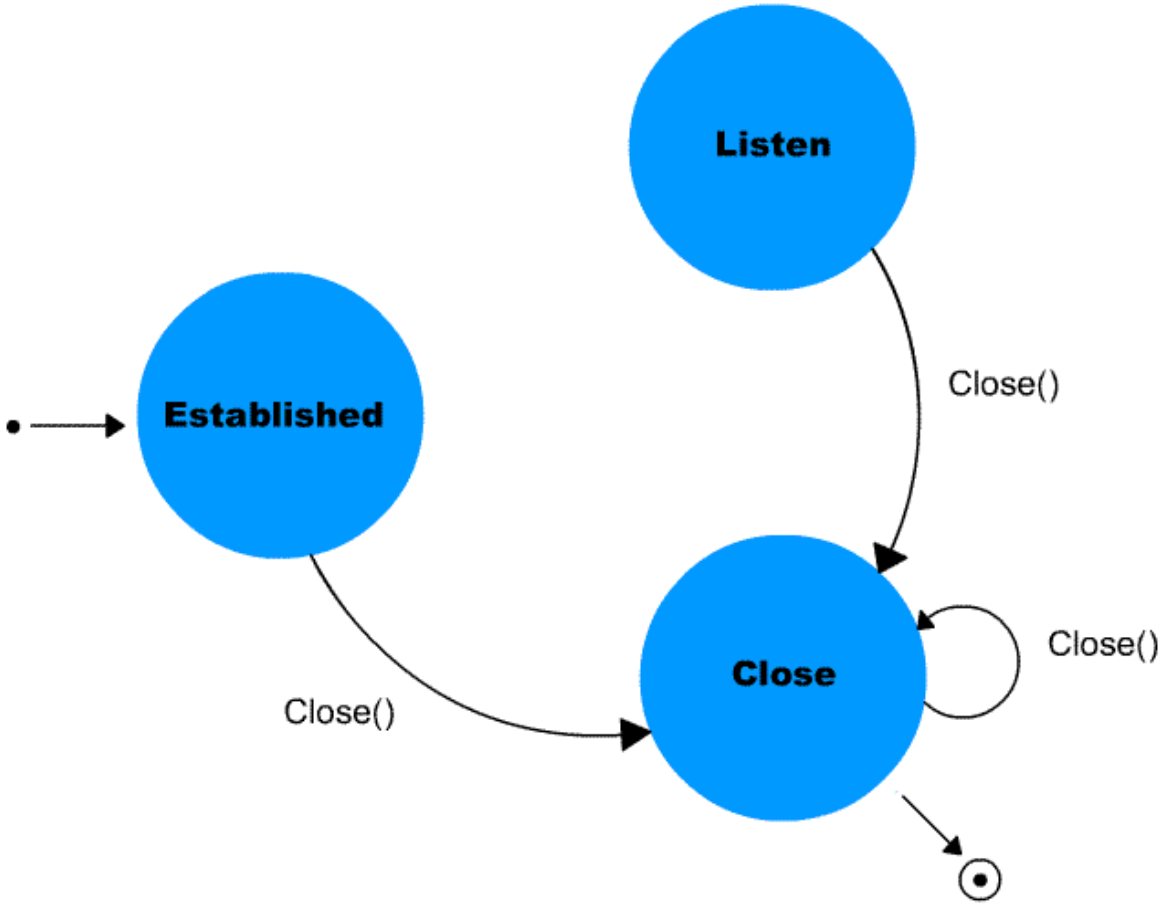
State (comportement)

- **Intention**
  - Modifier le comportement d'un objet quand son état interne change
  - Obtenir des traitements en fonction de l'état courant
  - Tout est mis en place pour donner l'impression que l'objet lui-même a été modifié
- **Motivation**
  - Eviter les instructions conditionnelles de grande taille (if then else)
  - Simplifier l'ajout et la suppression d'un état et le comportement qui lui est associé
- **Champs d'application**
  - Implanter une partie invariante d'un algorithme
  - Partager des comportements communs d'une hiérarchie de classes

Fréquence :

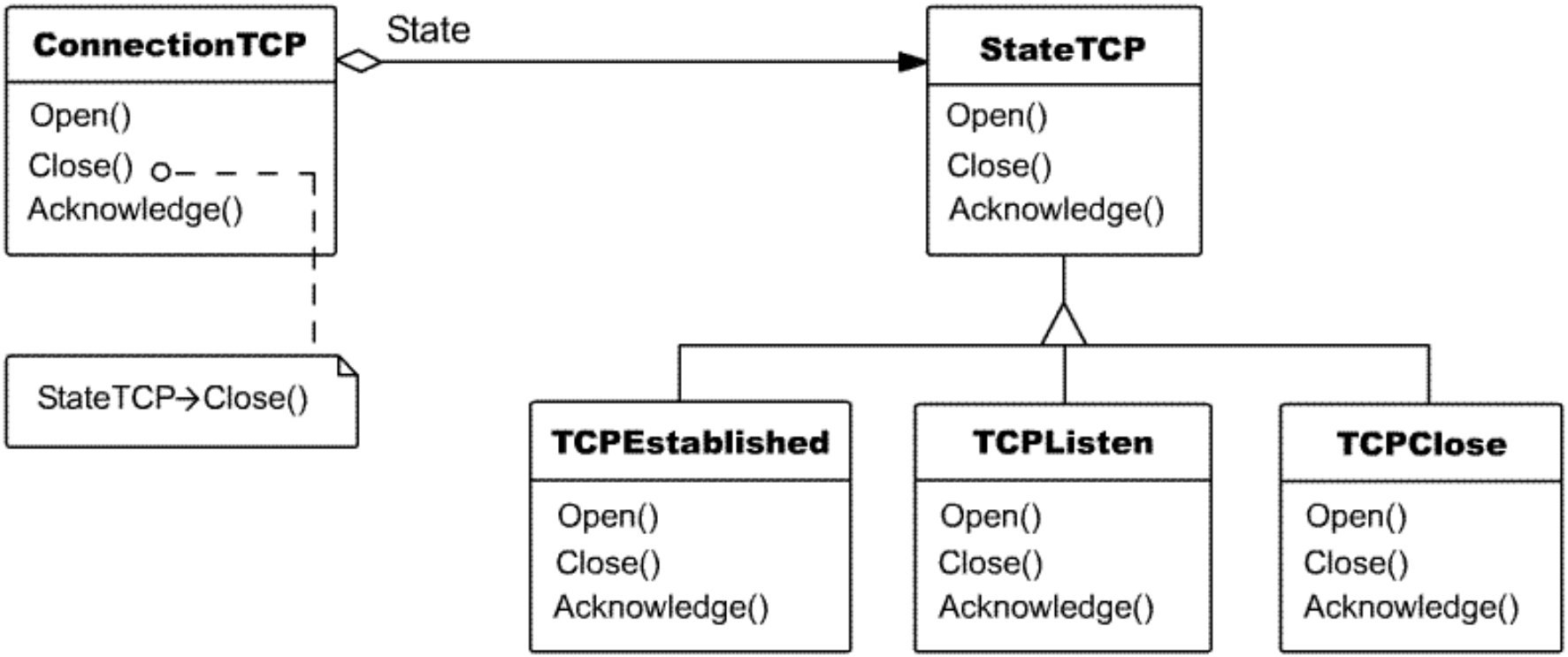
State (2)

- **Exemple : connexion TCP**



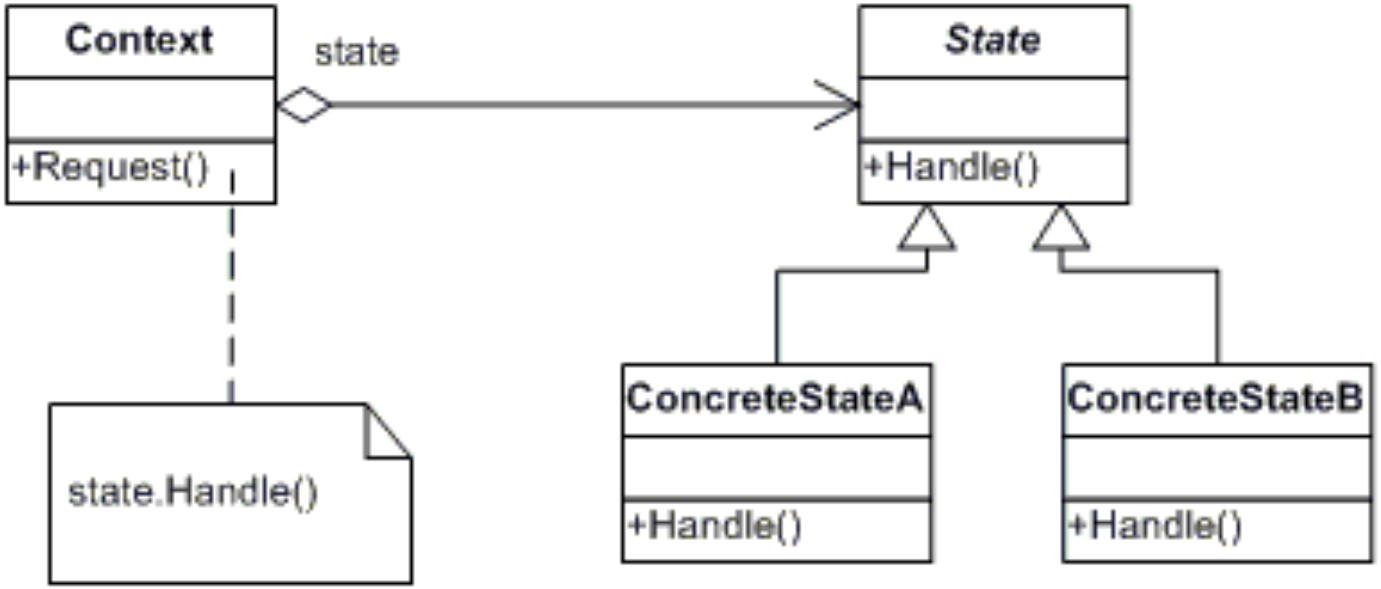
State (3)

- **Exemple : modélisation**



State (4)

- **Structure**



  - Créer des classes d'états qui implémentent une interface commune
  - Déléguer des opérations dépendantes des états de l'objet *Contexte* à son objet état actuel
  - Veiller à ce que l'objet *Contexte* pointe sur un objet état qui reflète son état actuel

State (5)

- **Participants**
  - **Context** (ConnectionTCP) est une classe qui permet d'utiliser un objet à états et qui gère une instance d'un objet ConcreteState
  - **State** (StateTCP) définit une interface qui encapsule le comportement associé avec un état particulier de Context
  - **ConcretState** (EstablishedTCP, ListenTCP, CloseTCP) implémente un comportement associé avec l'état de Context)
- **Collaborations**
  - Il revient soit à Context, soit aux ConcreteState de décider de l'état qui succède à un autre état
- **Conséquences**
  - Séparation des comportements relatifs à chaque état
  - transitions plus explicites

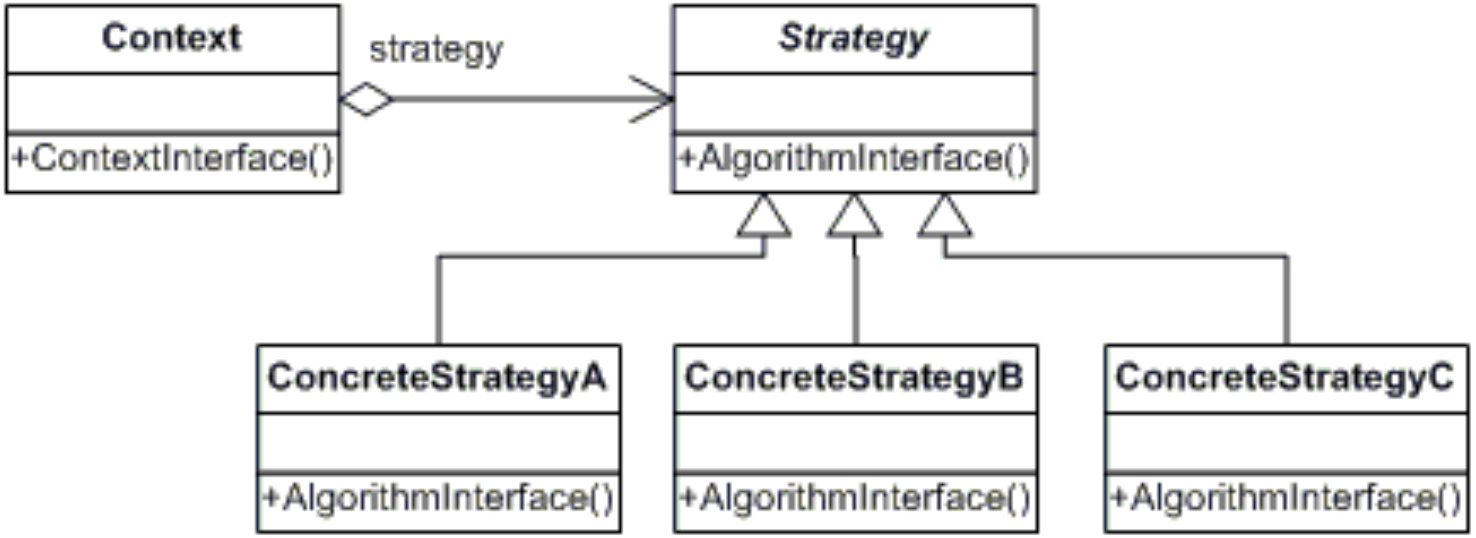
Strategy (comportement)

- **Intention**
  - Définir une hiérarchie de classes pour une famille d'algorithmes, encapsuler chacun d'eux, et les rendre interchangeables.
  - Les algorithmes varient indépendamment des clients qui les utilisent
- **Synonyme**
  - Policy
- **Motivation**
  - Cas des classes ayant le même comportement et utilisant des variantes d'un même algorithme :
    - Encapsuler ces algorithmes dans une classe (la stratégie)
    - Implémenter les "stratégies" dans les classes héritées
  - Evite les répétitions de code

Fréquence :

Strategy (2)

- **Champs d'application**
  - Lorsque de nombreuses classes associées ne diffèrent que par leur comportement
  - Lorsqu'on a besoin de plusieurs variantes d'un algorithme
  - Lorsqu'un algorithme utilise des données que les clients ne doivent pas connaître
  - Lorsqu'une classe définit plusieurs comportements
- **Structure**



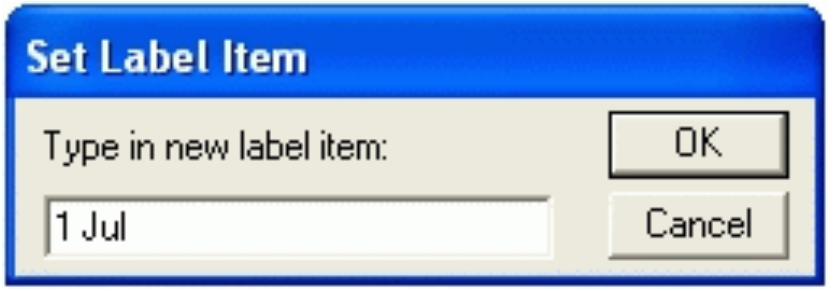
Strategy (3)

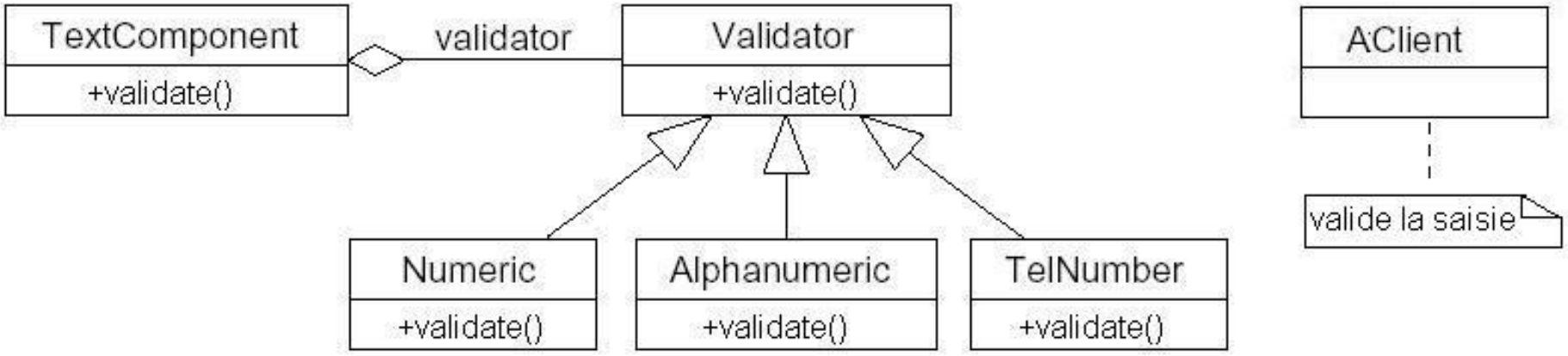
- **Participants**
  - **Context**
    - maintient une référence à l'objet Strategy
    - peut définir un interface qui permet à l 'objet Strategy d'accéder à ses données
  - **Strategy** déclare une interface commune à tous les algorithmes
  - **ConcreteStrategy** implémente l'algorithme en utilisant l'interface Strategy
- **Collaborations**
  - Le Context envoie les requêtes de ses clients à l'une de ses stratégies.
  - Les clients créent la classe concrète qui implémente l'algorithme.
  - Puis ils passent la classe au Context.
  - Enfin ils interagissent avec le Context exclusivement.

Strategy (4)

- ***Exemple :*** Validation des données dans un dialogue
  - Plusieurs stratégies de validation selon le type de données : numériques, alphanumériques,…

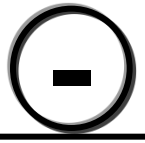
```
switch(data_type){
case NUMERIC : { //… }
case ALPHANUMERIC : { //… }
case TEL_NUMBER : { //…}
}
```





Strategy (5)

- **Conséquences**



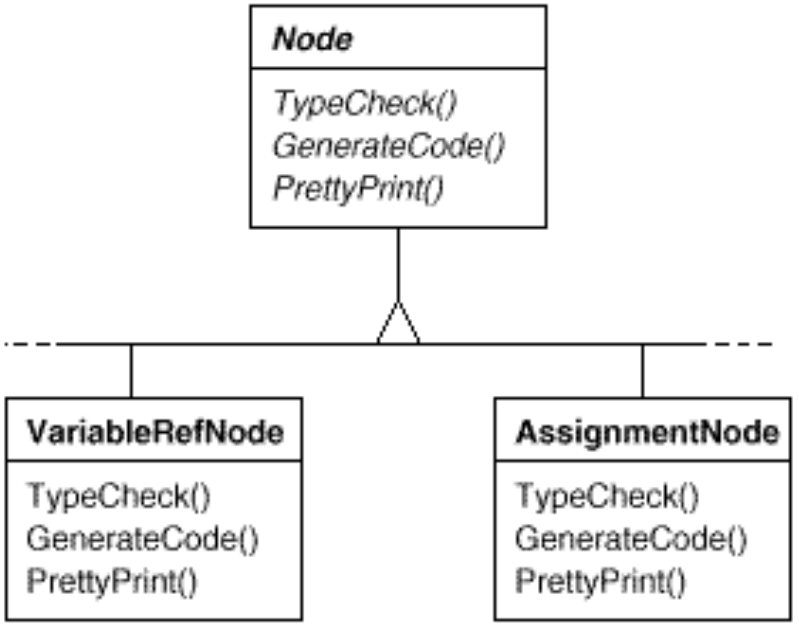
+ Simplification du code client + Élimination des boucles conditionnelles + Extension des algorithmes + Les clients n'ont pas besoin d'avoir accès au code des classes concrètes + Les familles d'algorithmes peuvent être rangées hiérarchiquement ou rassemblées dans une super-classe commune	- Le Context ne doit pas changer - Les clients doivent connaître les différentes stratégies - Le nombre d'objets augmente beaucoup - Imaginons une classe Strategy avec beaucoup de méthodes. Chaque sous-classe connaît ses méthodes mais peut être qu'elle ne les utilisera pas

Strategy (6)

- **Utilisations connues**
  - Validation des données en GUI
  - AWT et Swing en Java
- **Patterns associés**
  - Adapter, Flyweight, Template Method, State

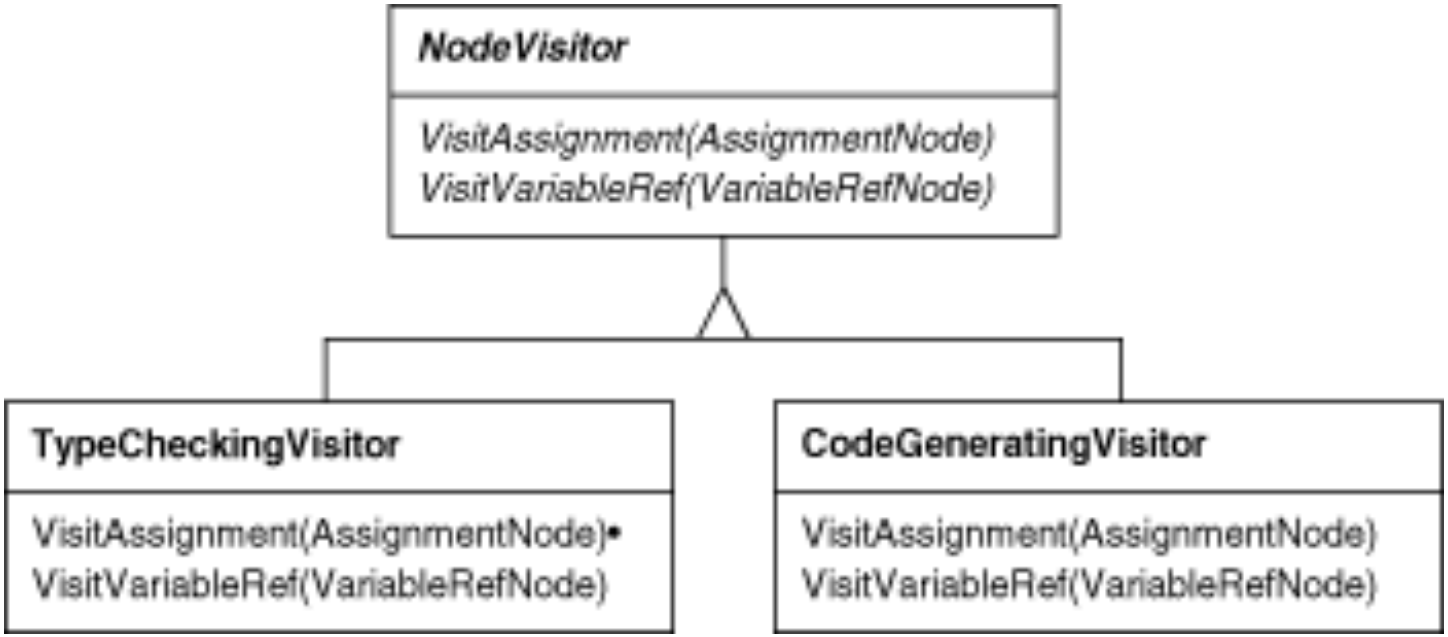
Visitor (comportement)

- **Intention**
  - Représenter **UNE** opération à effectuer sur les éléments d'une structure
  - Permet de définir une nouvelle opération sans changer les classes des éléments sur lesquels on opère
- **Motivation**
  - Un arbre de syntaxe abstraite pour un compilateur, un outil XML…
  - Différents traitement sur le même arbre : type check, optimisation, analyses…



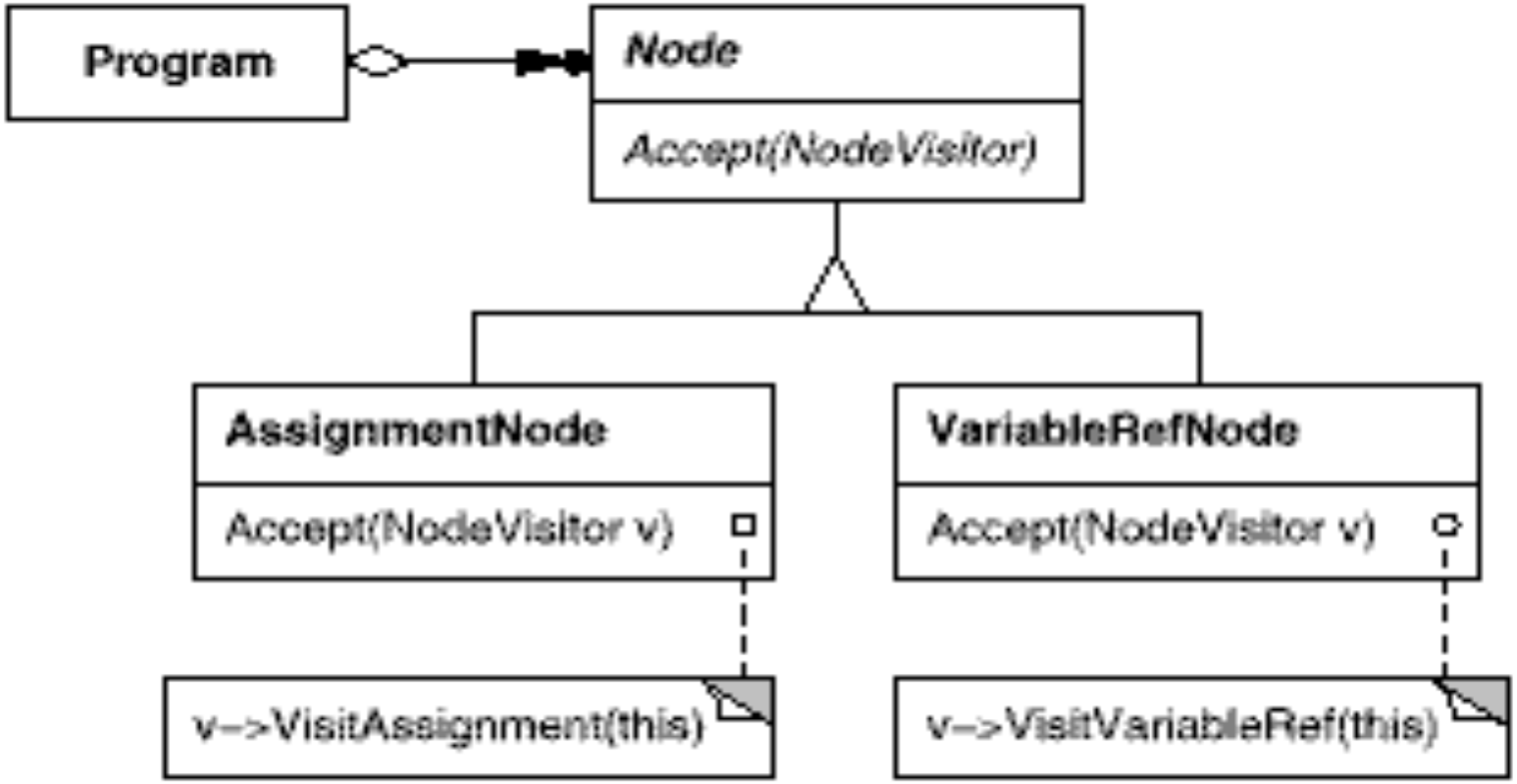
Fréquence :

Visitor (2)



- **Champs d'application**
  - Une structure contient beaucoup de classes aux interfaces différentes
  - Pour éviter la pollution des classes de la structure

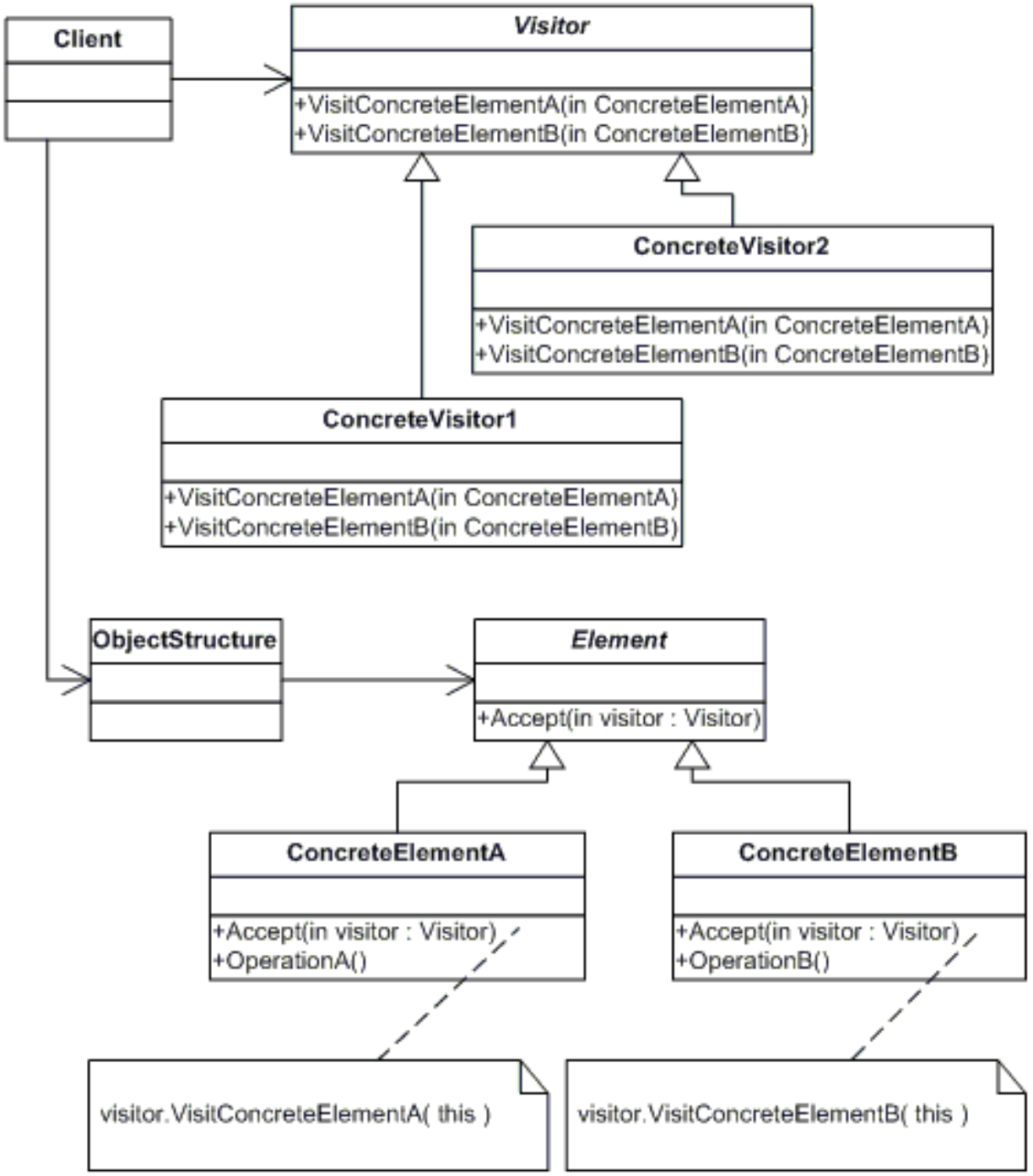
Visitor (3)



- **Champs d'application** (suite)
  - Les classes définissant la structure changent peu, mais de nouvelles opérations sont toujours nécessaires

Visitor

structure (4)

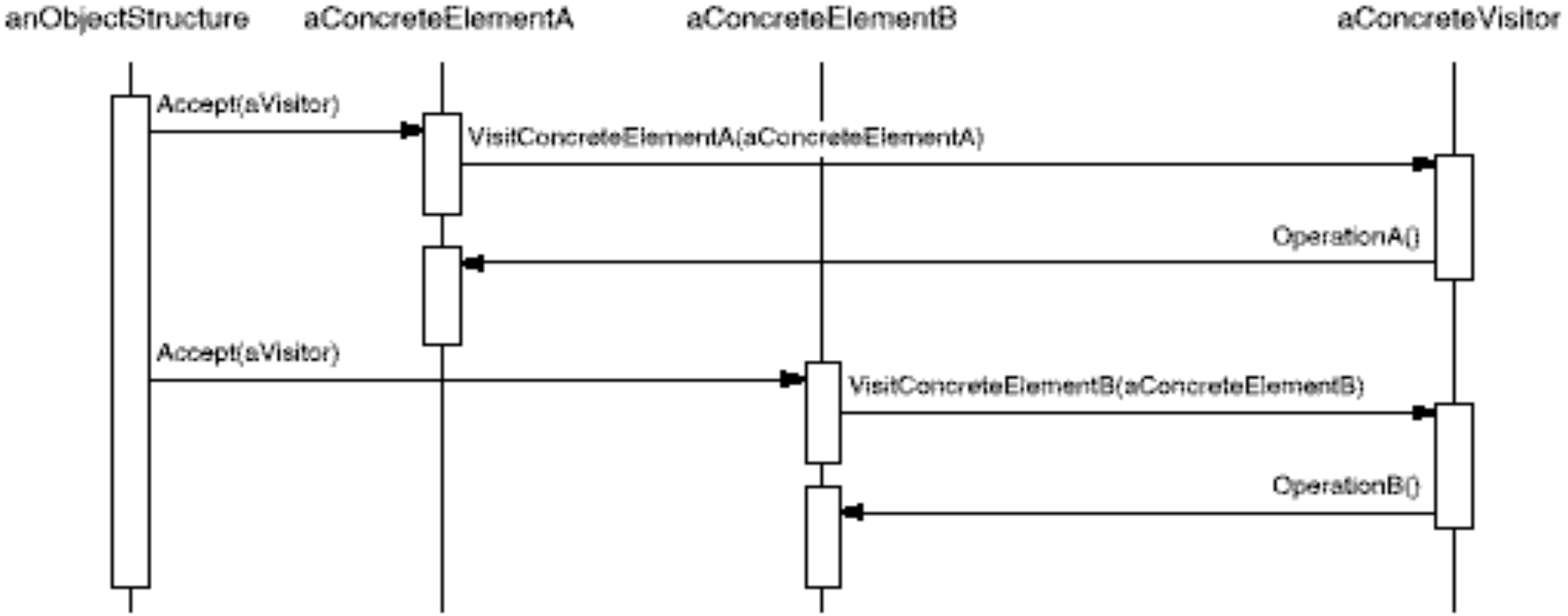


Visitor (5)

- **Participants**
  - **Visitor** (NodeVisitor) déclare l'opération de visite pour chaque classe de ConcreteElement dans la structure
  - ✓ Le nom et la signature de l'opération identifie la classe qui envoie la requête de visite au visiteur. Le visiteur détermine alors la classe concrète et accède à l'élément directement
  - **ConcreteVisitor** (TypeCheckingVisitor) implémente chaque opération déclarée par Visitor
  - ✓ Chaque opération implémente un fragment de l'algorithme, et un état local peut être stocké pour accumuler les résultats de la traversée de la structure

Visitor (6)

- **Element** (Node) définit une opération **Accept** qui prend un visitor en paramètre
- **ConcreteElement** (AssignmentNode, VariableRefNode) implémente l'opération **Accept**
- **ObjectStructure** (Program) peut énumérer ses éléments et peut être un Composite



Visitor (7)

- **Conséquences**
  1. Ajout de nouvelles opérations très facile
  2. Groupement/séparation des opérations communes (non..)
  3. Ajout de nouveaux ConcreteElement complexe
  4. Visitor traverse des structures où les éléments sont de types complètement différents / Iterator
  5. Accumulation d'état dans le visiteur plutôt que dans des arguments
  6. Suppose que l'interface de ConcreteElement est assez riche pour que le visiteur fasse son travail

     ➡ cela force à montrer l'état interne et à casser l'encapsulation

Visitor (8)

- **Implémentation**
  - Visitor = *Double dispatch* : nom de l'opération + 2 receveurs : visiteur + élément
    - ☞ C'est la clé du pattern Visitor
    - *Single dispatch* (C++, Java) : 2 critères pour une opération : nom de l'opération + type du receveur
  - Responsabilité de la traversée
    - Structure de l'objet ⇒ figée
    - Visiteur ⇒ flexible mais dupliquée
    - Itérateur ⇒ retour aux 2 cas précédents
- **Utilisations connues :** Compilateur, bibliothèques C++, Java…
- **Patterns associés :** Composite, Interpreter

Autres patrons de comportement

- **Chaîne de responsabilité**

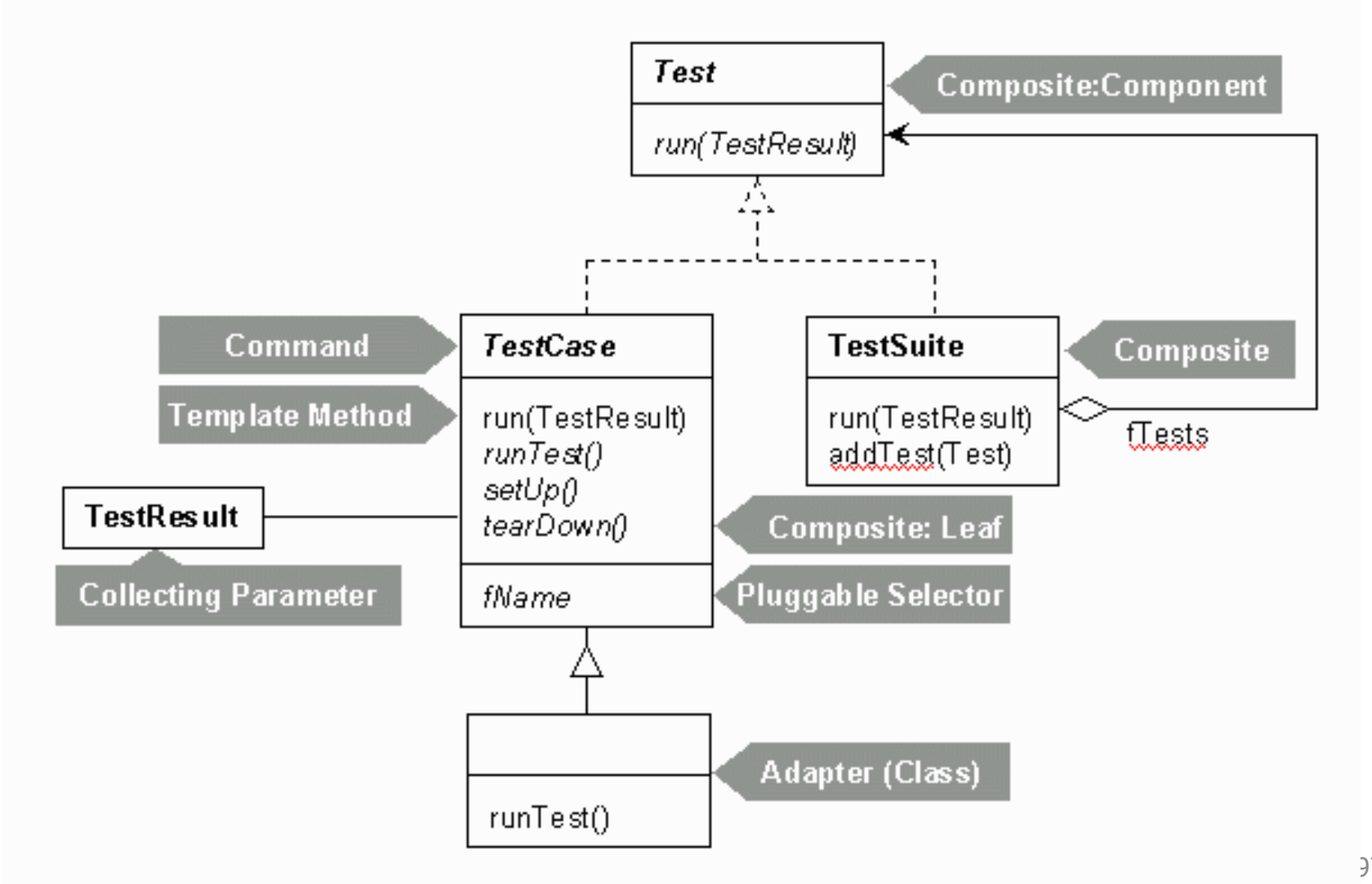
  - Envoyer une requête le long d'une chaîne d'objets de réception
  - Découplage entre expéditeur d'une requête et récepteur concret
- **Interprète**

  - Pour un langage donné, définir une représentation pour sa grammaire, fournir un interprète capable de manipuler ses phrases grâce à la représentation établie
- **Iterator**

- **Mediator**

  - Encapsule les modalités d'interaction d'un certain ensemble d'objets
  - Couplage faible en dispensant les objets de se faire explicitement référence
- **Mémento**
  - Externalise, enregistre (puis restaure) l'état d'un objet

Une étude de cas : Junit v3



Sources

- *Design Patterns – Elements of Reusable Object-Oriented Software,* Addison Wesley, 1994
- http://dofactory.com

- Autres références
  - http://sourcemaking.com/design_patterns
  - http://www.oodesign.com/
  - http://www.fluffycat.com/Java-Design-Patterns/