

Patrons (et anti-patrons)



de conception

Singleton (création)

- **Intention**

- S'assurer qu'une classe a une seule instance, et fournir un point d'accès global à celle-ci.

- **Motivation**

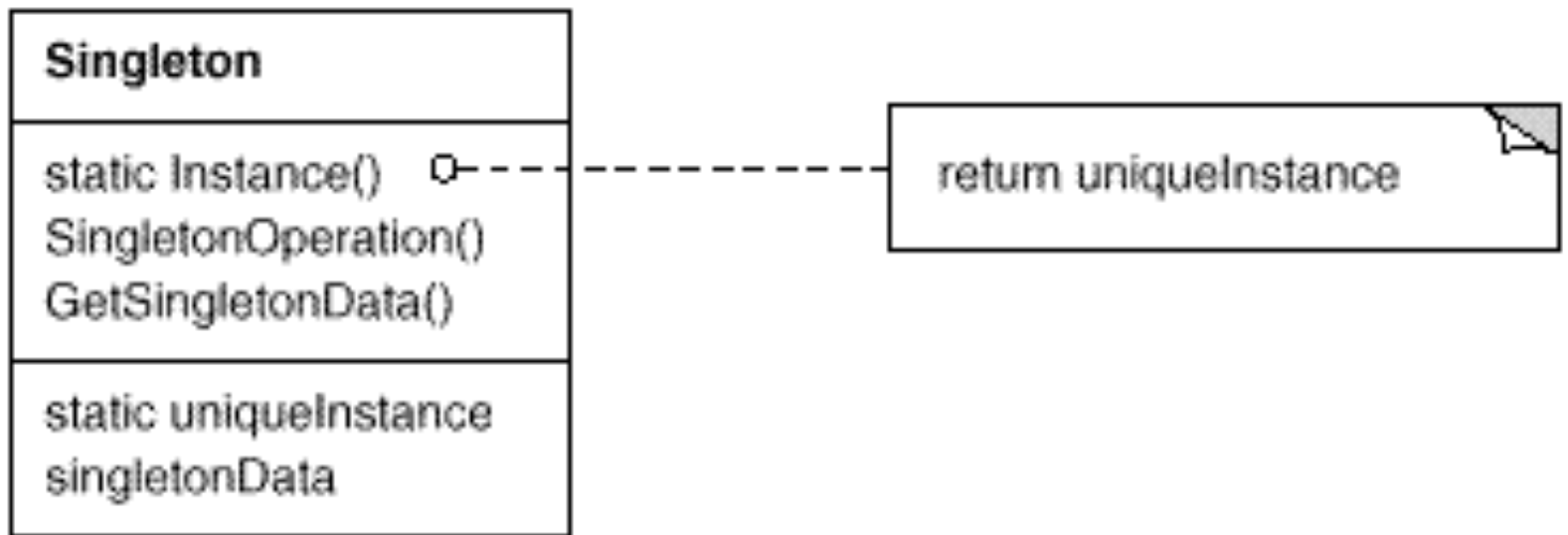
- Un seul spooler d'imprimante / plusieurs imprimantes
- Plus puissant que la variable globale

- **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

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

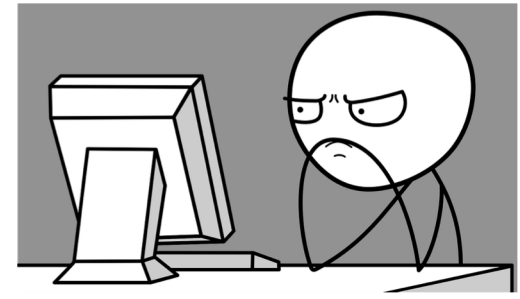
Singleton (4)

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

Singleton : un anti-patron ?

- Le bon singleton ? Il implémente une instance unique, mais c'est un bon singleton... NON
 - Gestion d'une seule instance avec une responsabilité unique
 - Pas d'état, sur la gestion de l'instance
 - Exemple : formater, cache, logger, interface d'accès à du matériel
- Mauvais singleton ?
 - Représentation d'un utilisateur qui vient de se logger
 - Représentation d'un plateau de jeu partagé
 - Facilité d'accès des valeurs dans plusieurs zones/couches de l'application

Why singletons suck...



- **Graphe de dépendances entre objets caché**
- **Difficiles à tester**
 - En fait, c'est un couplage fort : en étant globaux, c'est tout leur environnement qui doit gérer leur état
 - Ils sont difficiles à « mock », on doit écrire du code spécifique pour les tester
 - Ils ne sont pas vraiment extensibles par héritage
- **Pas bon pour la concurrence**
 - Ou pas *thread-safe*
 - Ou goulot d'étranglement en cas d'accès multiples et concurrents
- **Solution : Injection de dépendances**
 - cours ISA au S8 (ceci est un message publicitaire de Sébastien Mosser)

Autres patrons de création

- **Prototype**

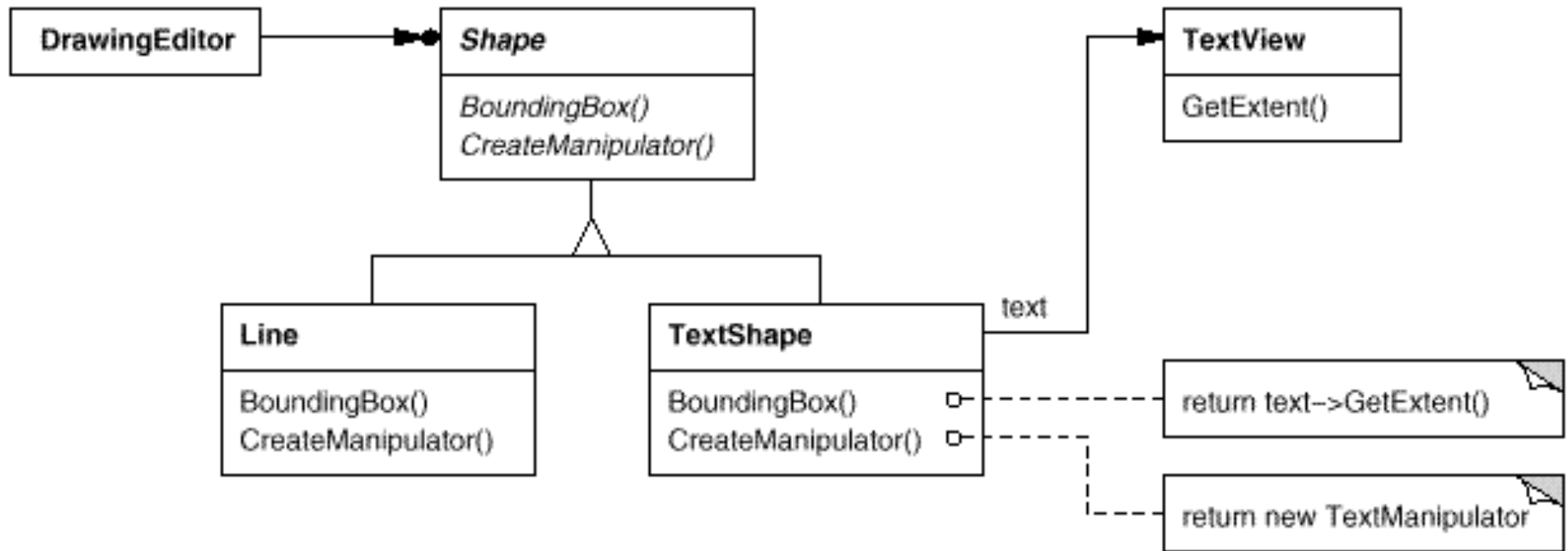


- Indiquer le type des objets à créer en utilisant une instance (le prototype). les nouveaux objets sont des copies de ce prototype (clonage)

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

Adapter (2)

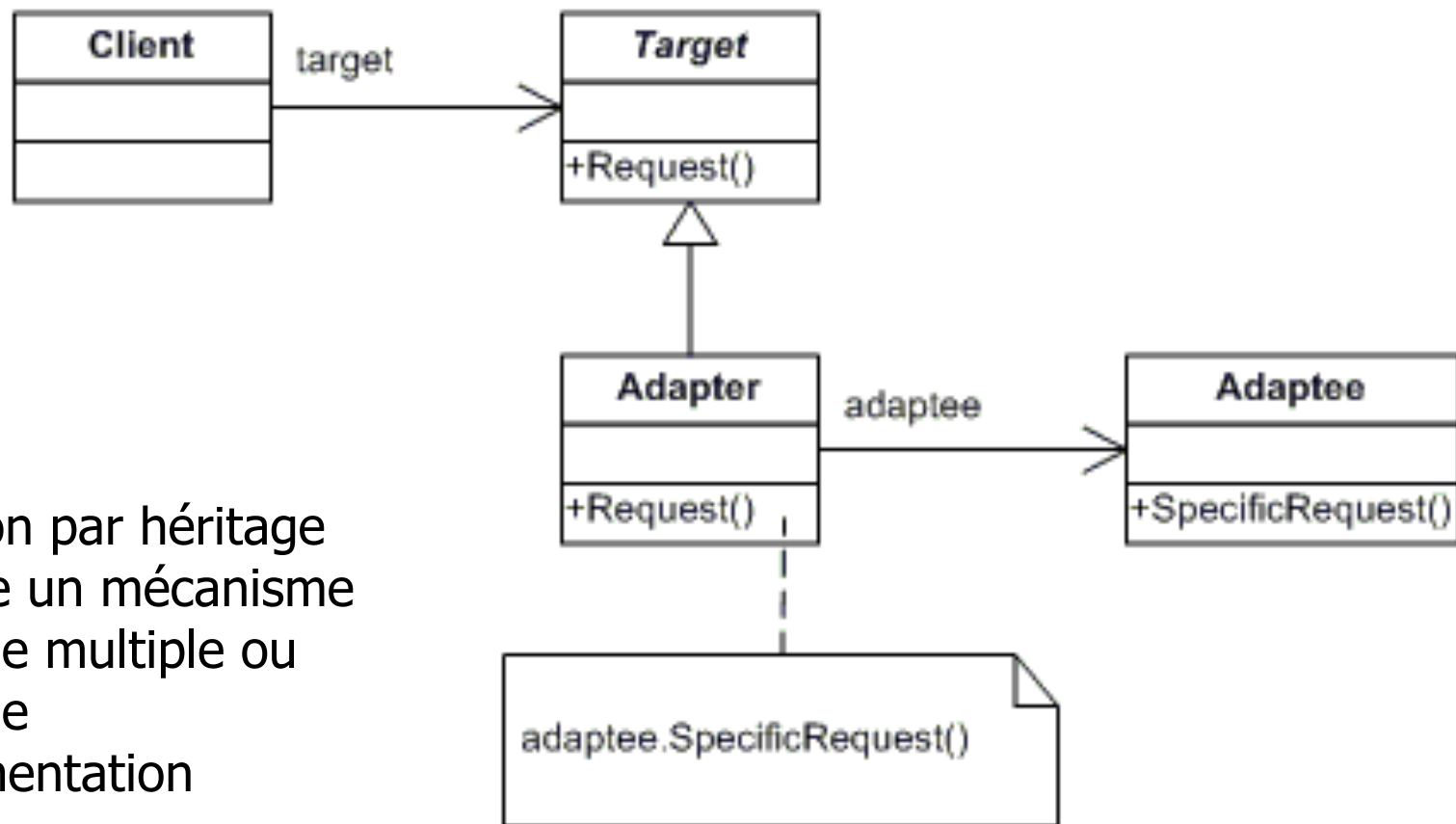


- **Champs d'application**

- 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 appelle 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

Façade (structure)

- **Intention**

- Fournir une interface unique, simplifiée ou unifiée, pour accéder à un ensemble d'interfaces d'un sous-système complexe.

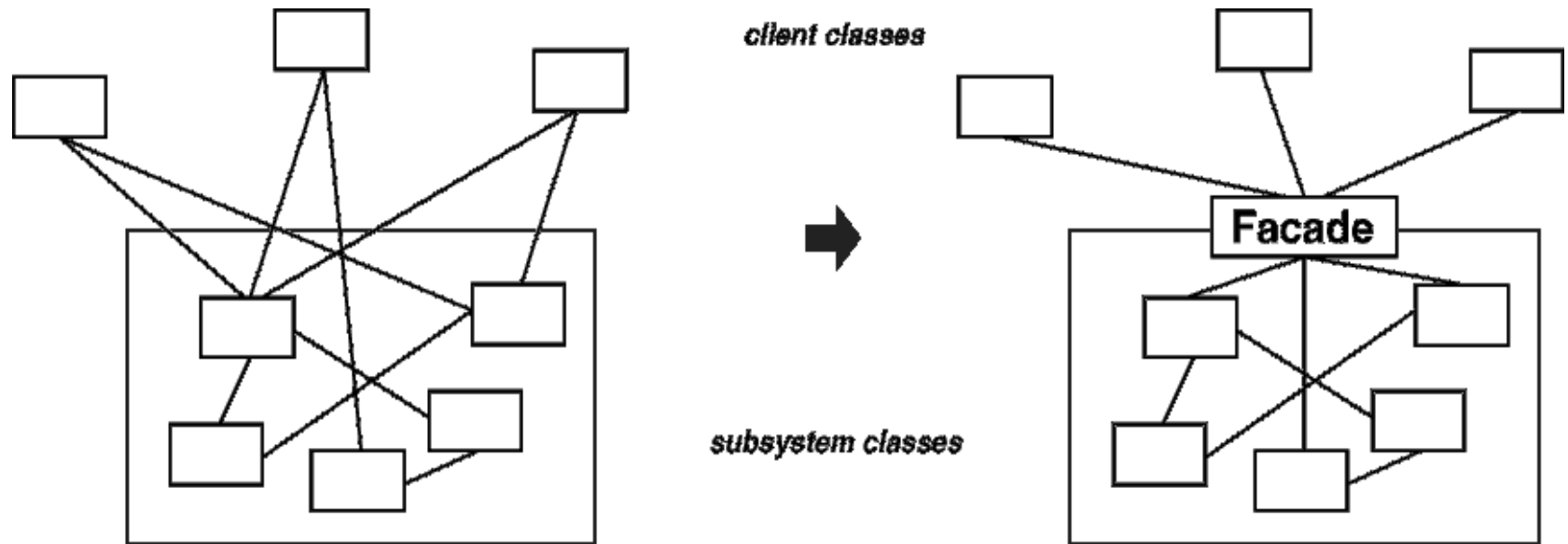
- **Motivation**

- 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)

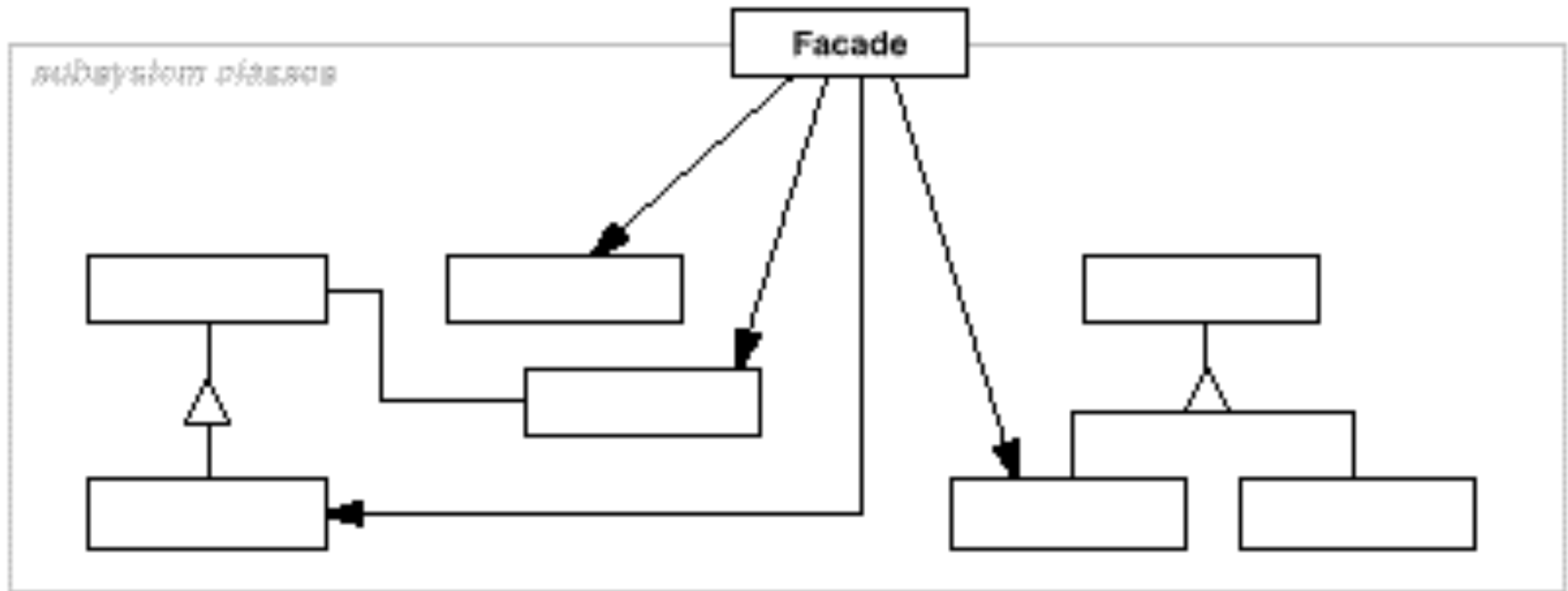


- **Champs d'application**

- 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

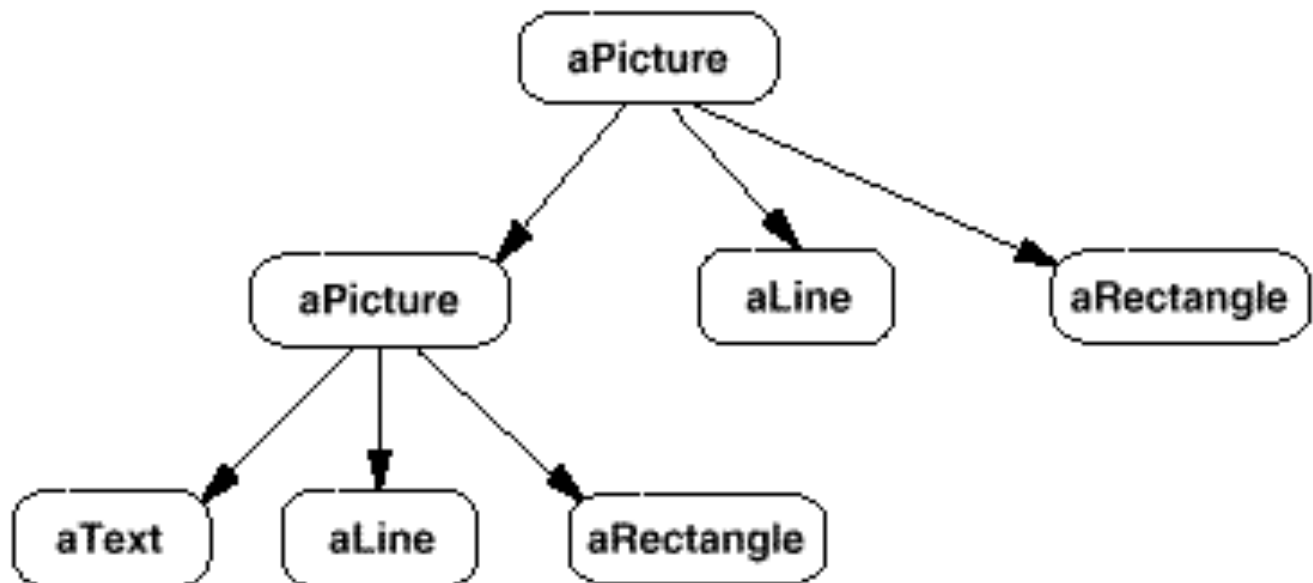
Façade (6)

- **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 :



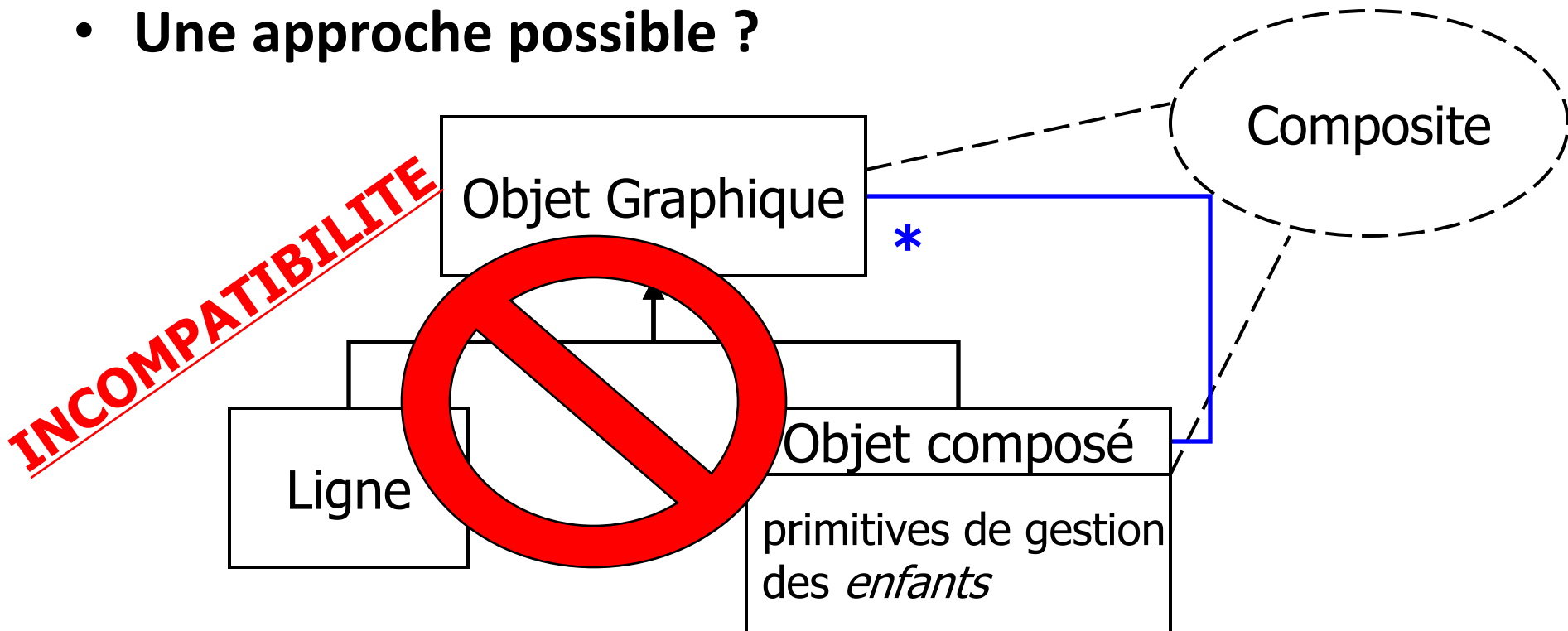
Composite

(2)

- **Motivation**

- Une classe abstraite qui représente à la fois les primitives et les containers

- **Une approche possible ?**

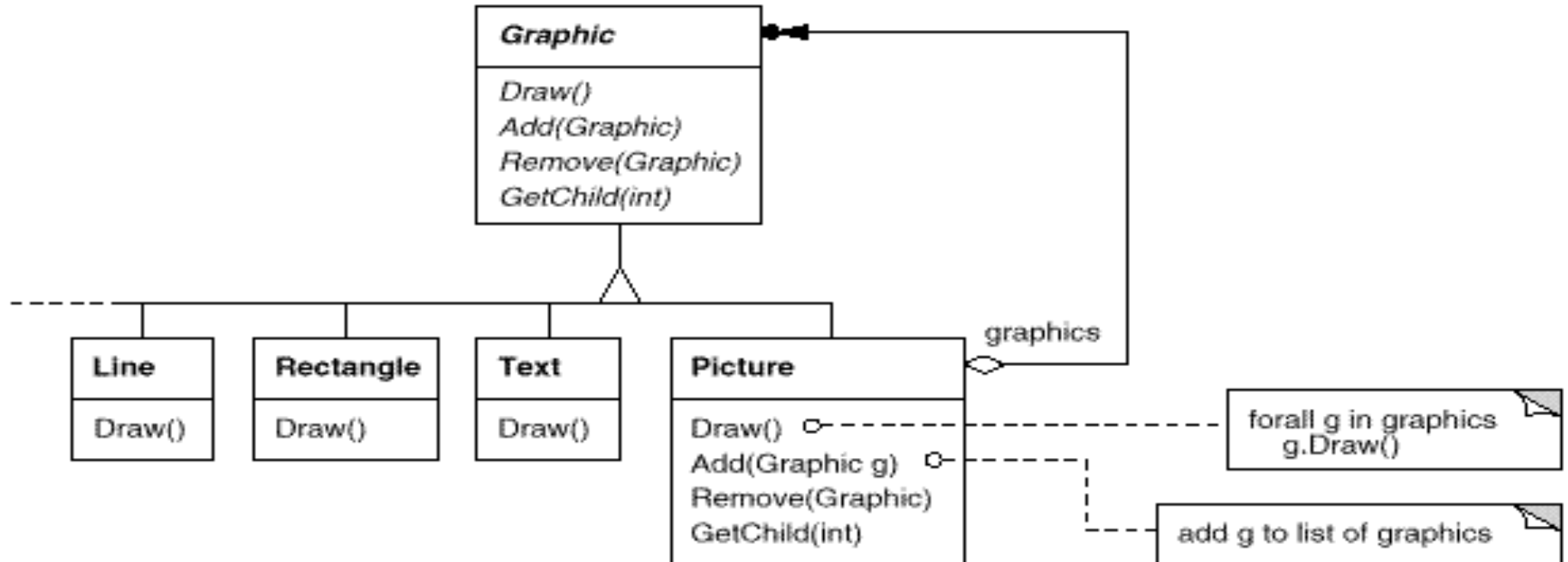


Composite

(3)

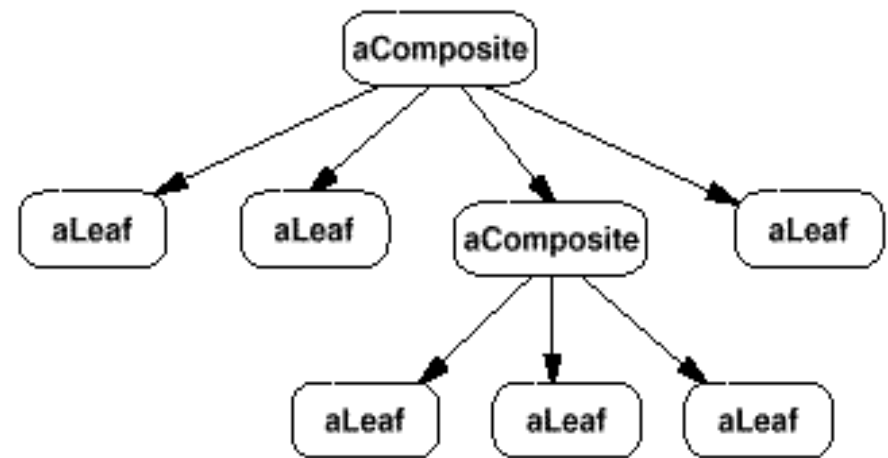
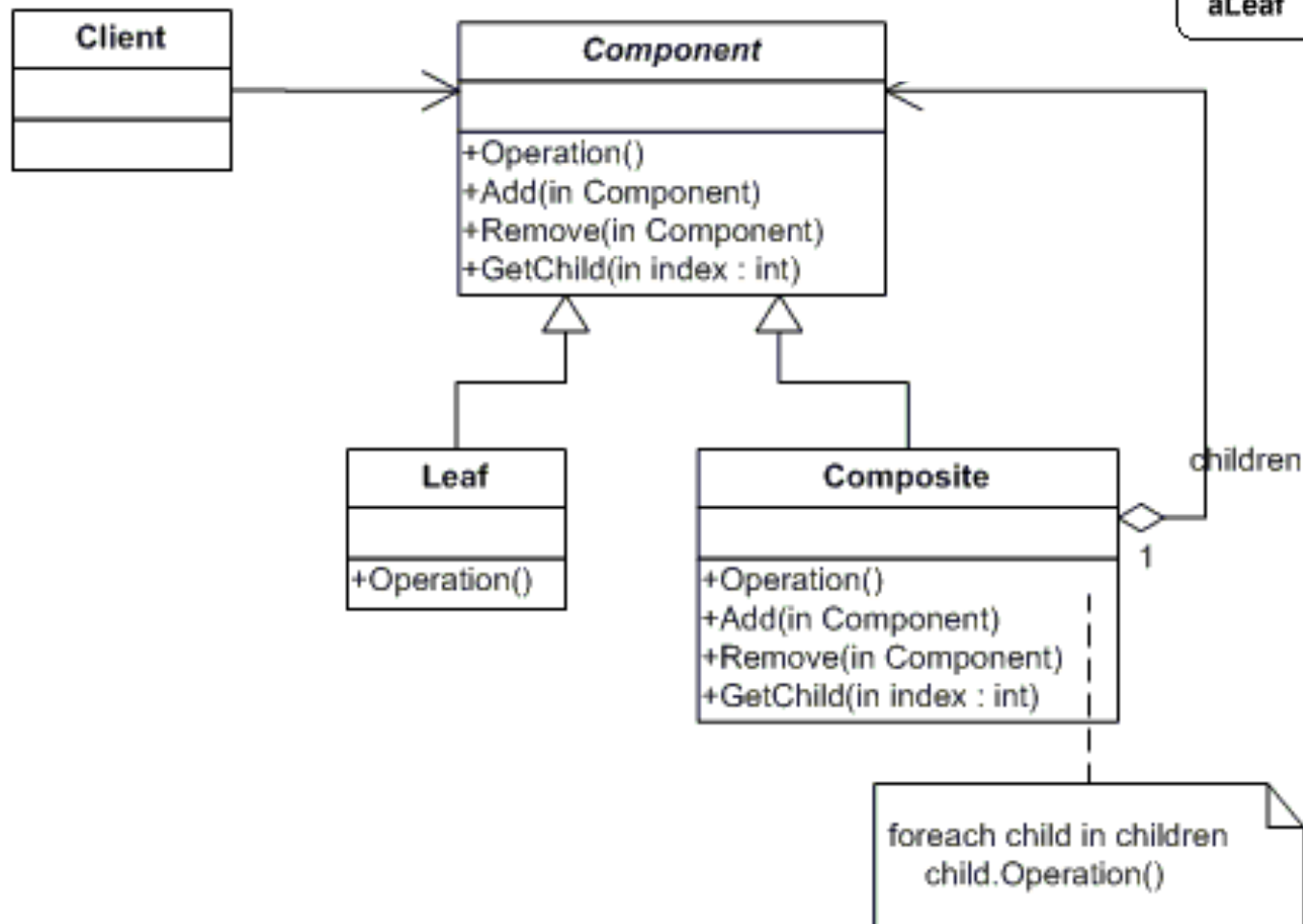
- **Champs d'application**

- 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

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

Chain of Responsibility (comportement)

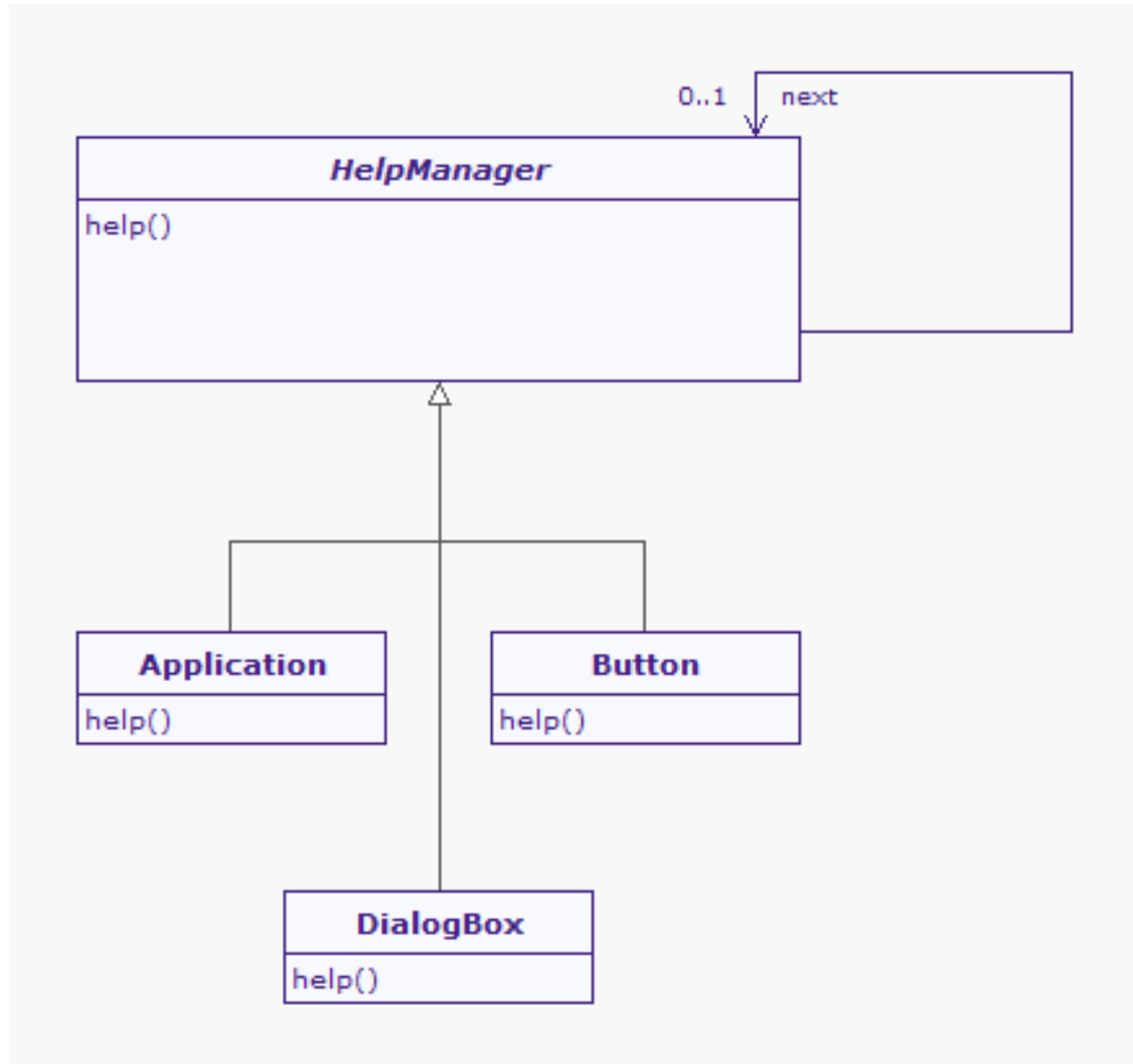
- **Intention**

- Permettre à un objet d'envoyer une instruction (requête) sans savoir quel objet va effectuer le traitement.
- Faire suivre une demande le long de la chaîne jusqu'à ce quelle soit traitée par un récepteur.

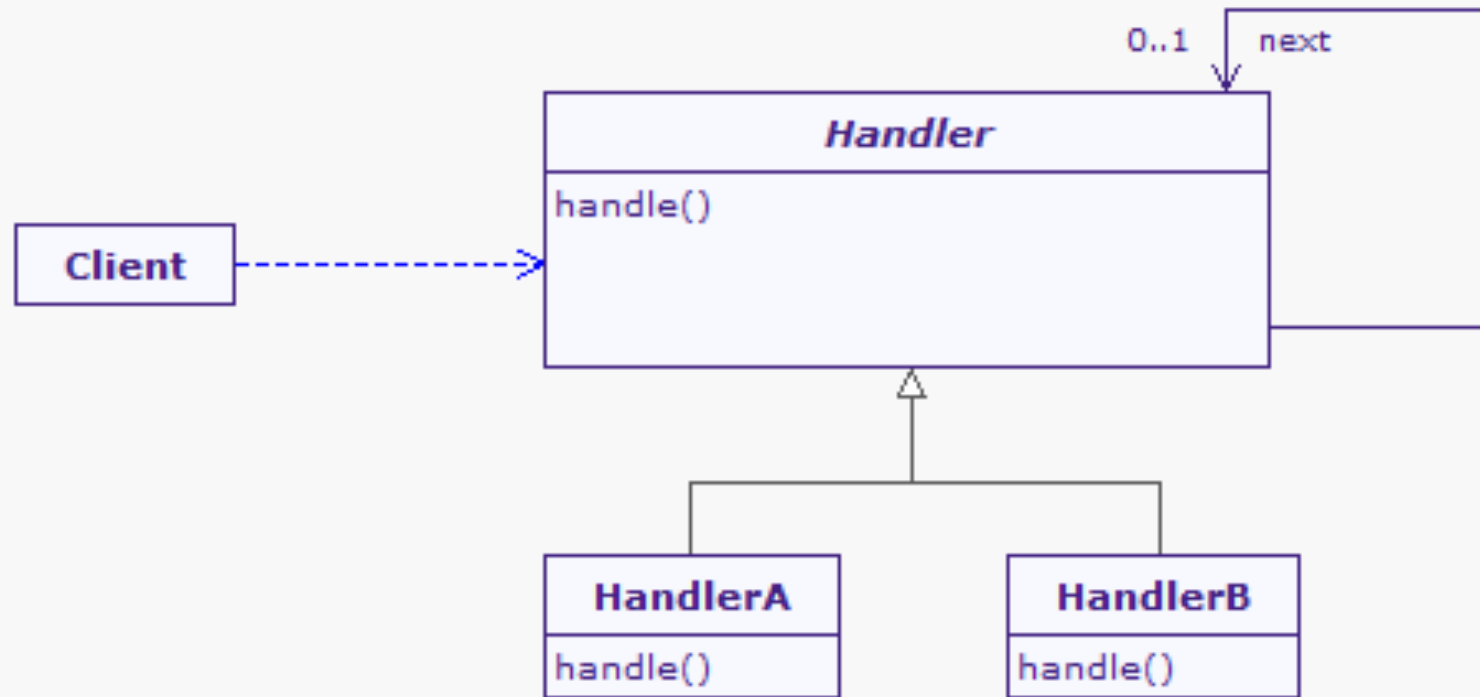
- **Motivation**

- Avoir un objet capable d'envoyer un ordre à un autre objet sans préciser le nom ni la nature du destinataire.
- Plus d'un objet peut être capable de recevoir et de gérer une requête, et il faut prioriser entre les objets de réception sans que le client ne gère cela directement.
- L'ensemble des objets qui peuvent traiter une requête doit être défini dynamiquement.

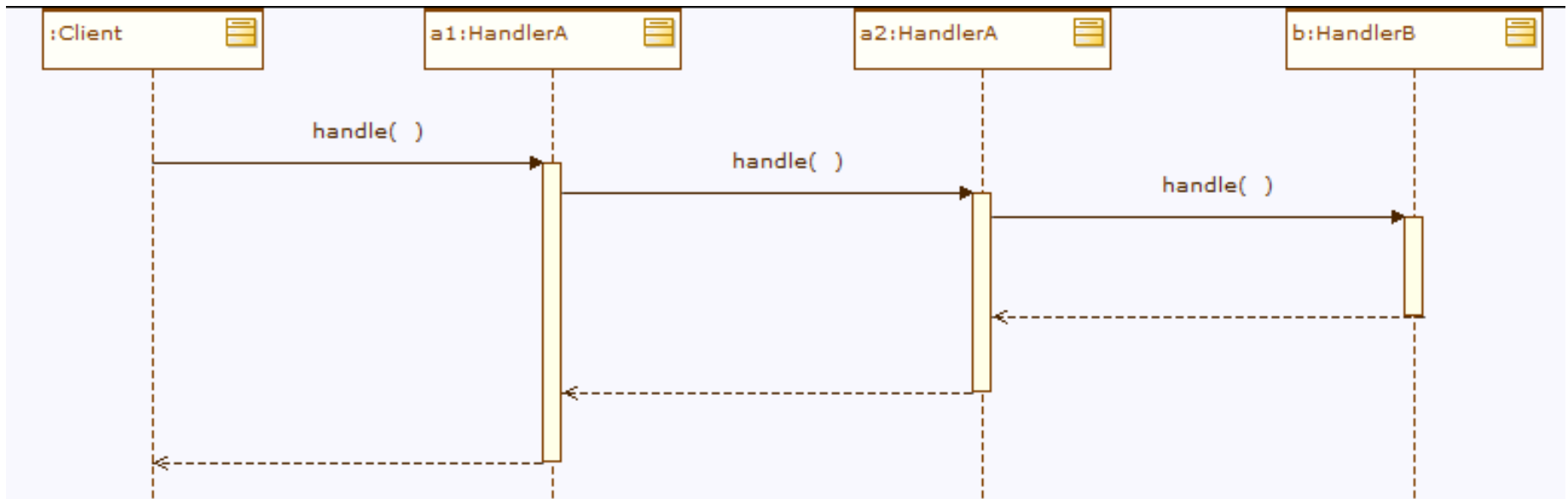
Chain of Responsibility (2)



Chain of Responsibility (3)



Chain of Responsibility (4)



Chain of Responsibility (5)

- **Avantages**

- Réduction du couplage
- Possibilité de modifier dynamiquement la façon de traiter une requête
- Souplesse accrue dans l'attribution des responsabilités aux objets (modification dans l'ordre à l'exécution)

- **Inconvénients**

- Pas de garantie que la requête va être traitée
- Si la chaîne est longue, des problèmes de performance peuvent apparaître

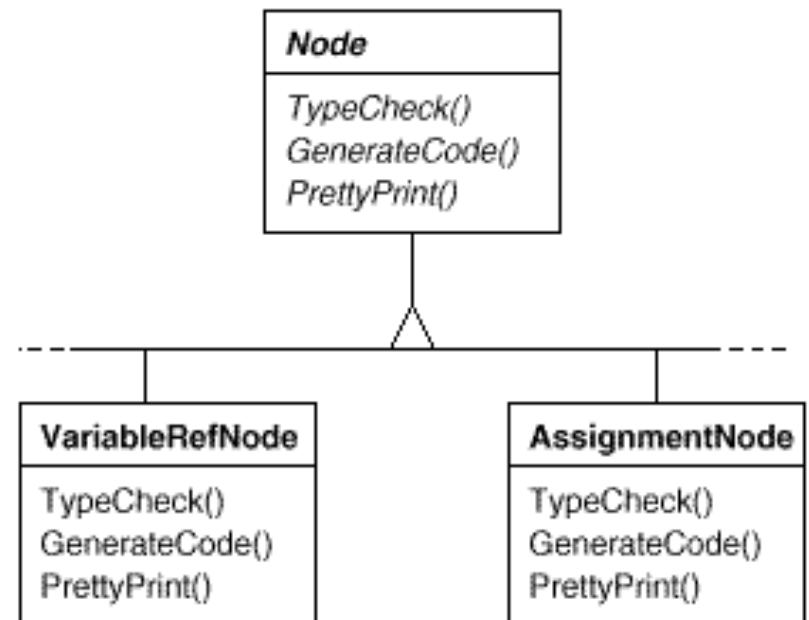
Visitor (comportement)

- **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**

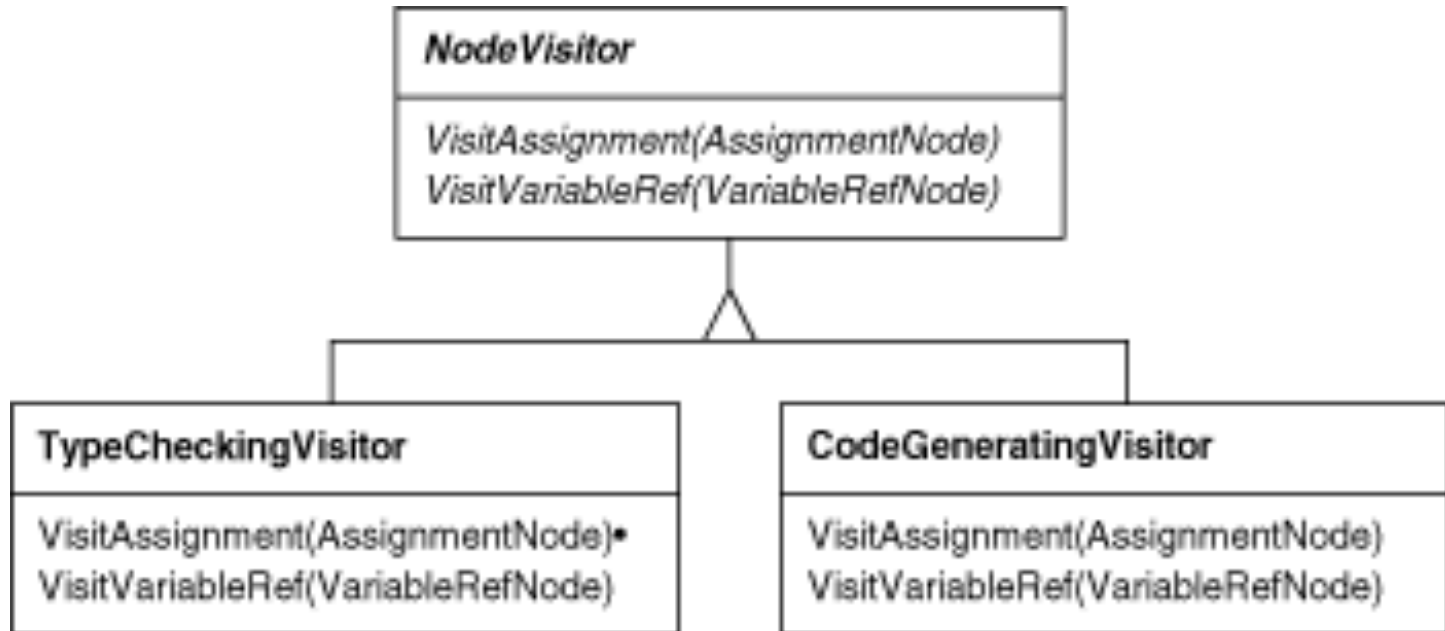
- Un arbre de syntaxe abstraite pour un compilateur, un outil XML...
- Différents traitements sur le même arbre : type check, optimisation, analyses...



Fréquence :



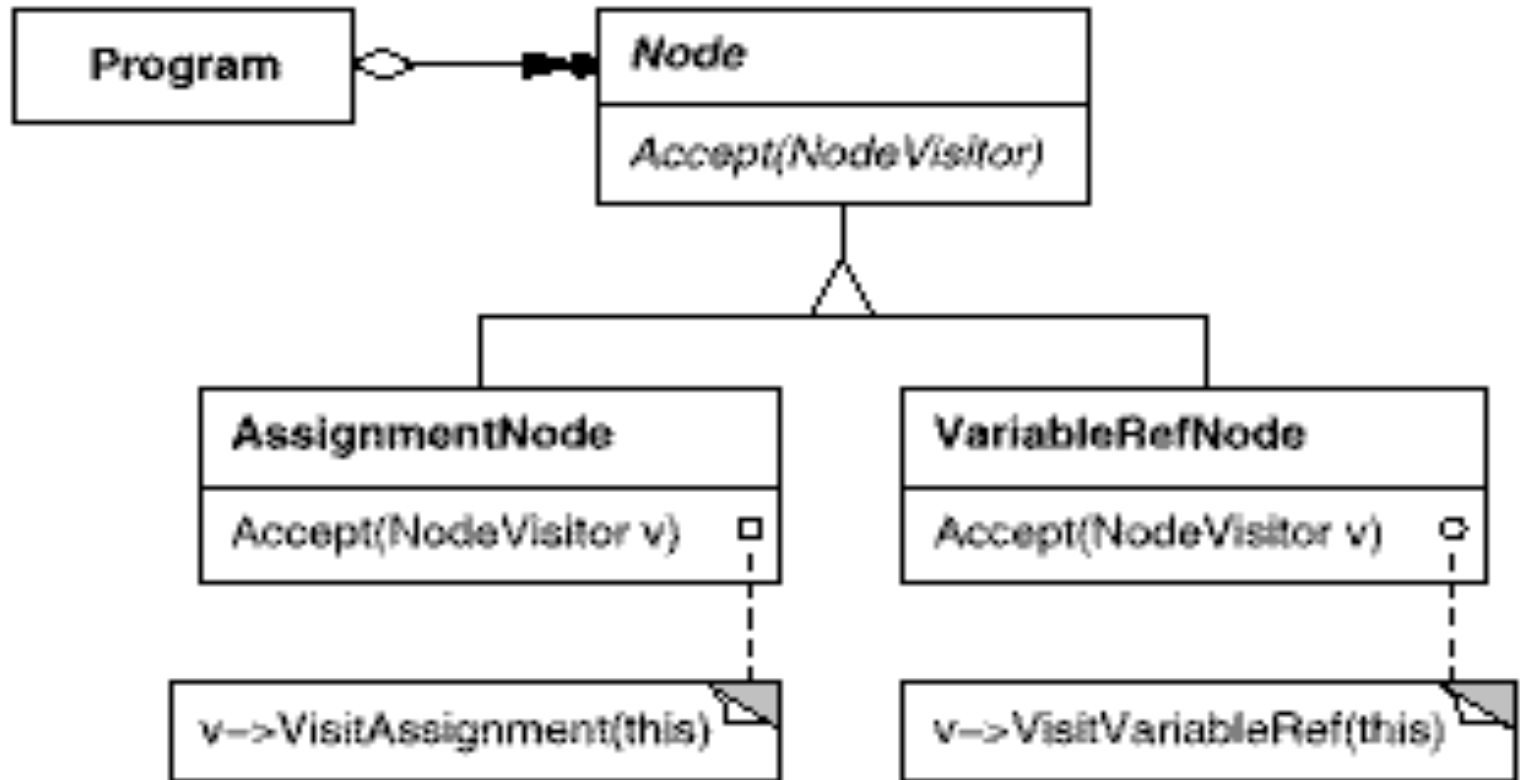
Visitor (2)



- **Champs d'application**

- Une structure contient beaucoup de classes aux interfaces différentes
- Pour éviter la pollution des classes de la structure

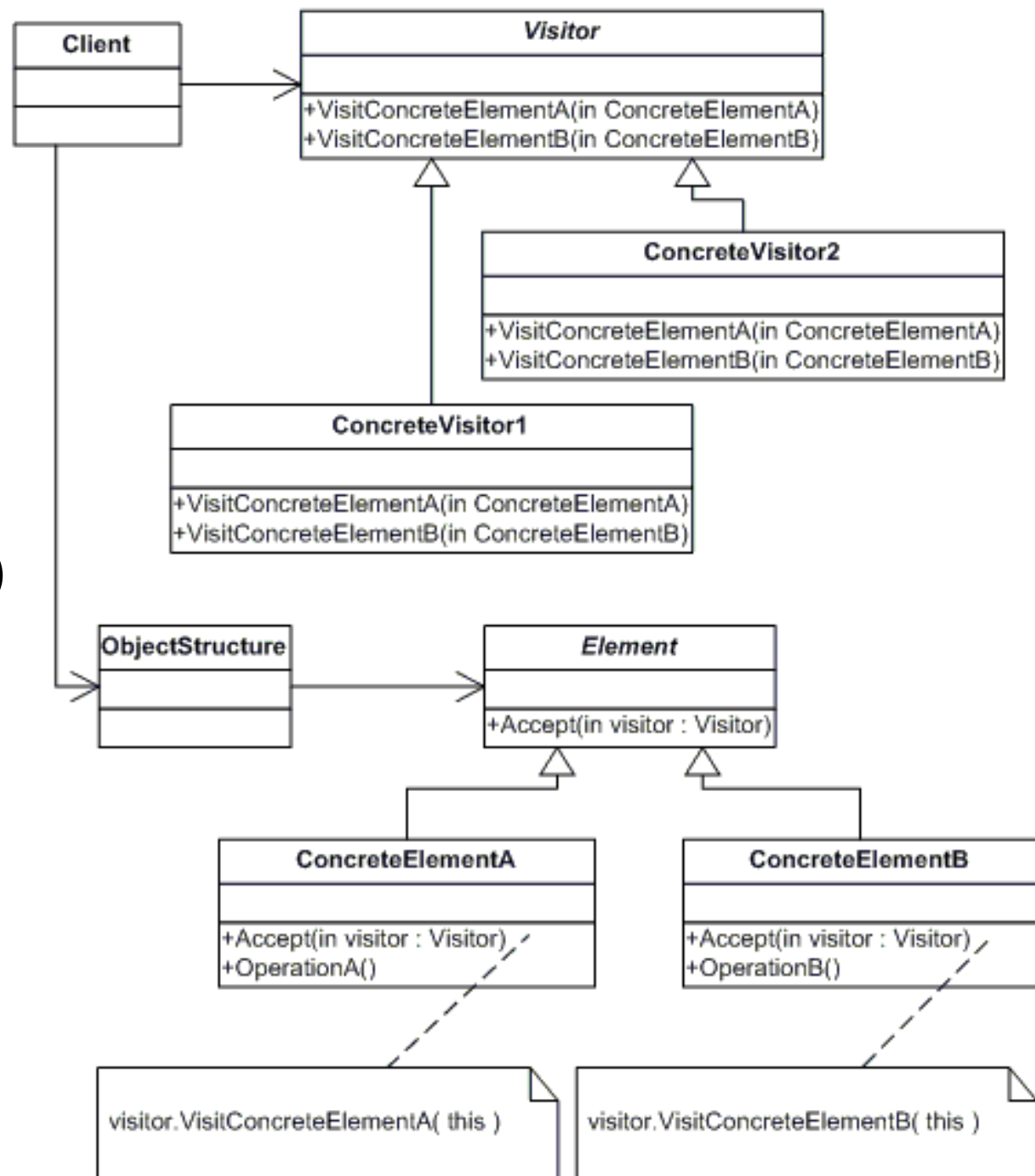
Visitor (3)



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

Visitor

structure (4)



Visitor

(5)

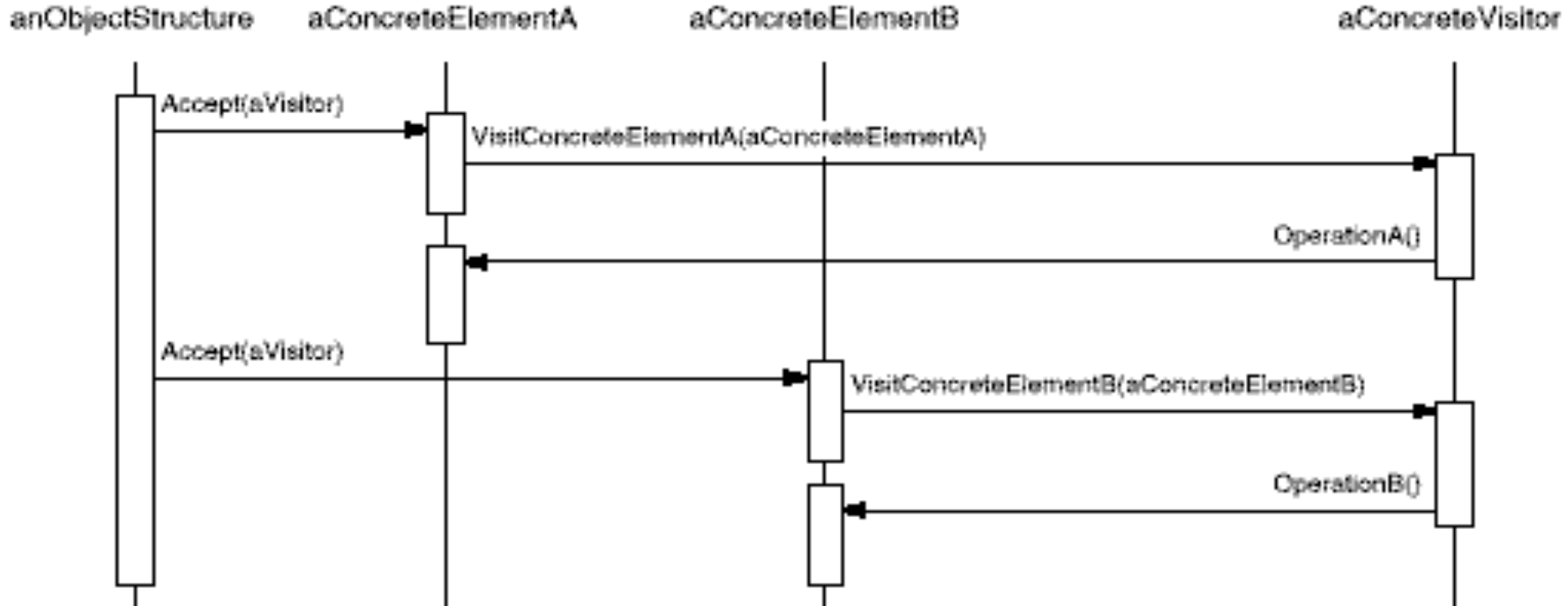
- **Participants**

- **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

- **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

- **Memento**



- Externalise, enregistre (puis restaure) l'état d'un objet