R5.A.08 : Qualité de développement
(Diapos I.Borne)

Cours Design Patterns 1 :

- Intention de construction : Builder
- Intention de responsabilité : Proxy
- Intention d'extension : Visitor

Classification des patterns selon leur intention

Intention	**Patterns**
Interfaces	Adapter, Facade, Composite, Bridge
Responsability	Singleton, Observer, Mediator, Proxy, Chain of Responsability, Flyweight
Construction	Builder, Factory Method, Abstract Factory, Prototype, memento
Operations	Template Method, State, Strategy, Command, Interpreter
Extensions	Decorator, Iterator, Visitor

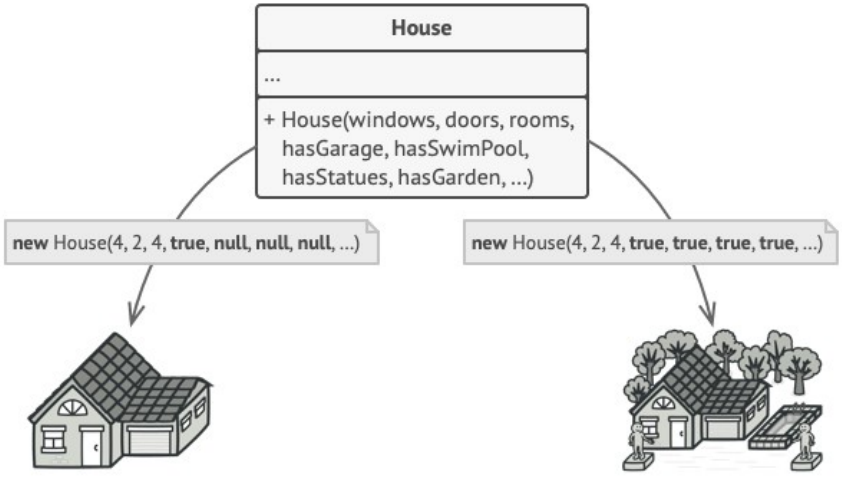
Intention de construction

- Construction ordinaire
  - Constructeur Java
  - Aspect important : collaboration que l'on peut orchestrer parmi les constructeurs
- Collaboration de superclasse
  - Chaînage des constructeurs
- Collaboration dans une classe
  - Interaction des constructeurs d'une classe

Le patron Builder – Intention et problème

Intention

Séparer la construction d'un objet complexe de sa représentation, afin qu'un même processus de création puisse créer différentes représentations



Problème

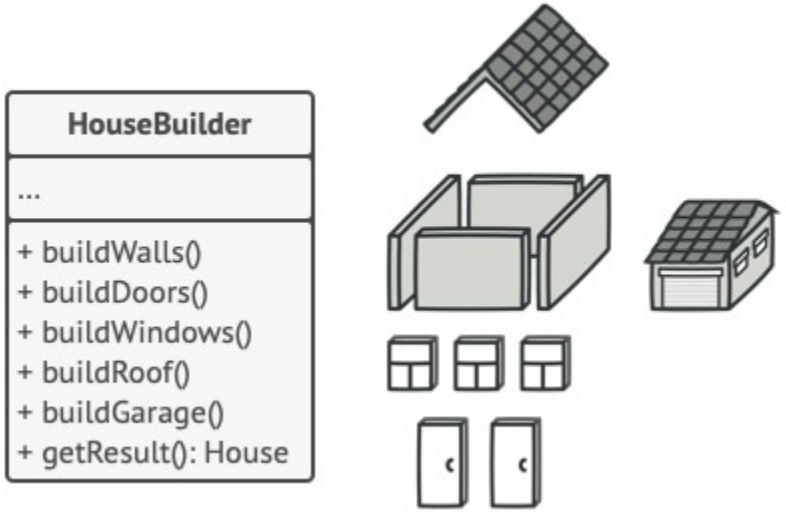
Un objet qui requiert une construction (étape par étape) pour initialiser tous ses attributs et objets internes.

Usuellement, on crée un grand grand constructeur avec de nombreux paramètres.

Pire encore : les étapes de constructions sont réparties dans le code client

Le patron Builder – Solution

Extraire la construction de l'objet et la mettre dans une classe Builder



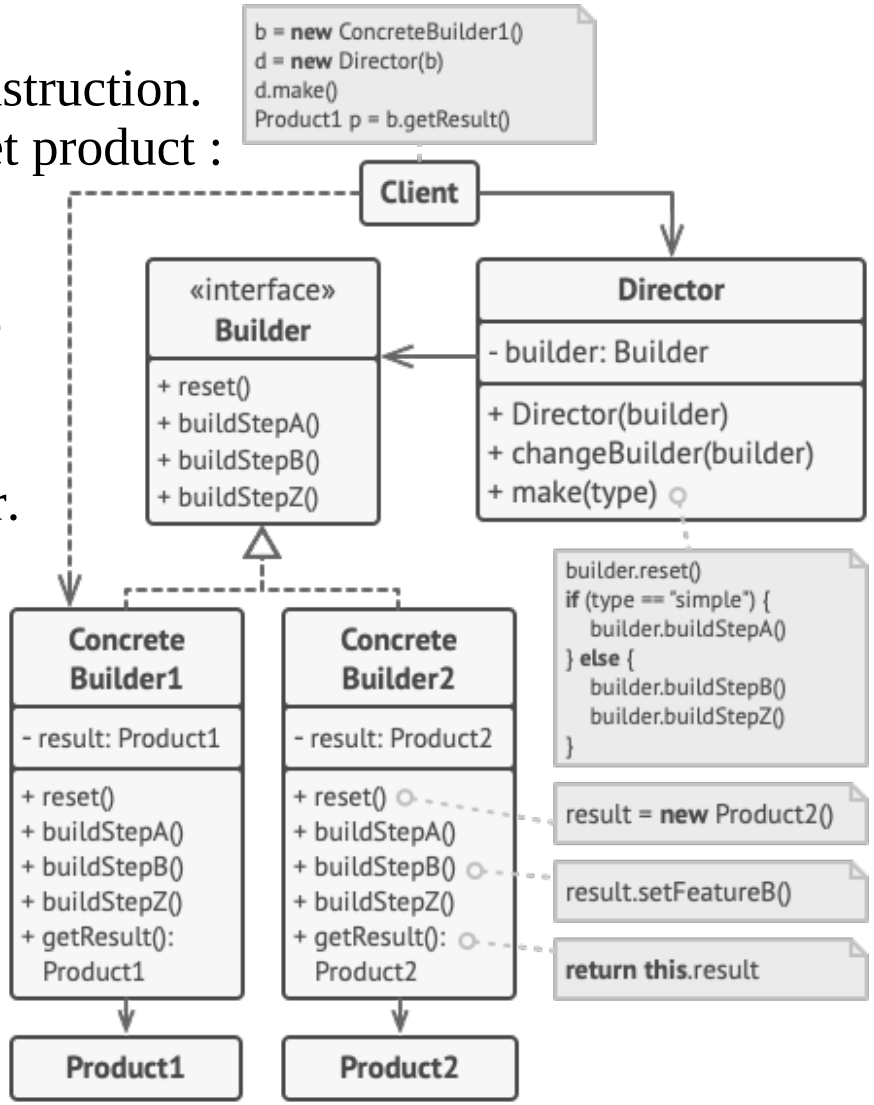
Dans le code client, on utilise un objet builder et on n'appelle que les méthodes (étapes) nécessaires à la création de l'objet souhaité (maison sans garage)

Quand utiliser le patron Builder

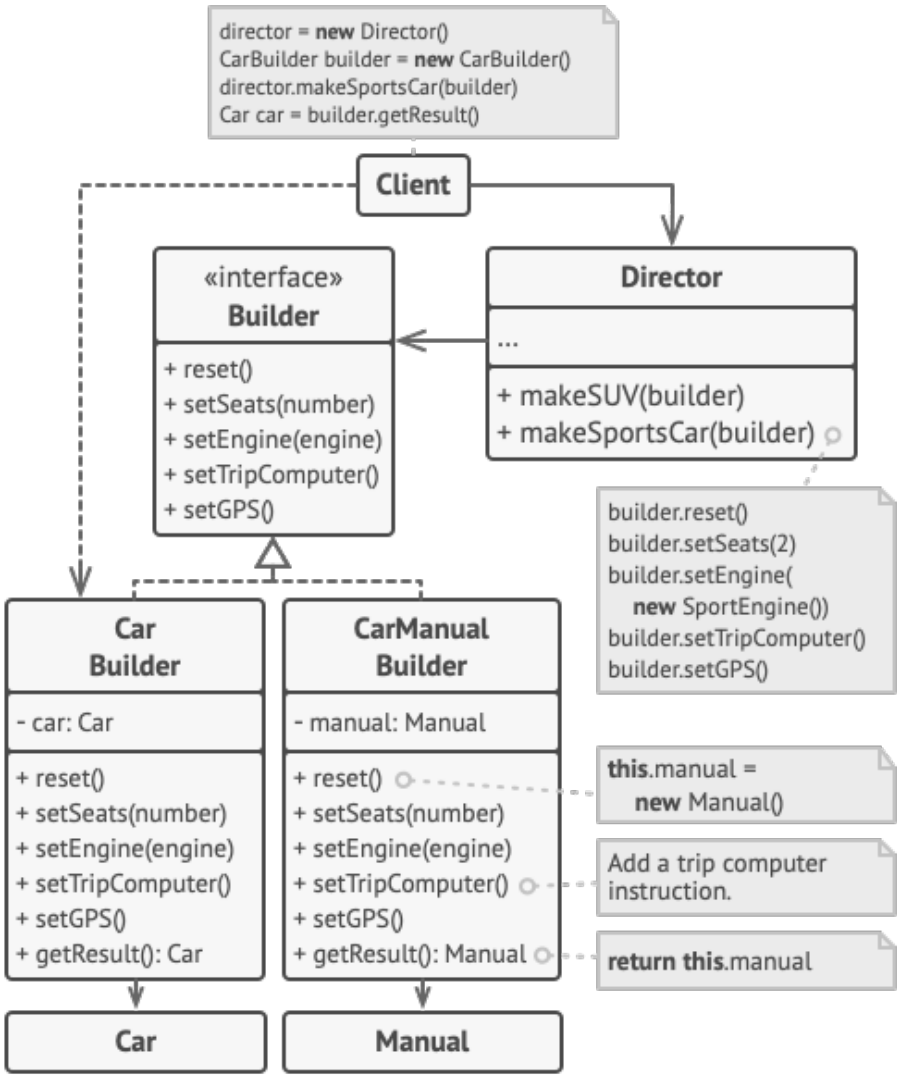
- Quand une classe a une structure interne complexe, par exemple avec un ensemble variable d'objets reliés.
- Quand l'algorithme, pour créer un objet complexe, doit être indépendant des parties qui forment l'objet et comment elles sont assemblées.
- Quand une classe a des attributs qui dépendent les uns des autres, par exemple un ordre de construction.
- Quand le processus de construction doit permettre différentes représentations de l'objet qui est construit.

Structure du pattern Builder

Builder : est responsable de la construction.
Il coordonne l'assemblage de l'objet product :
créer des ressources, stocker
des résultats intermédiaires
et fournir la structure fonctionnelle
pour la création.
Director : construit un objet
en utilisant une instance de Builder.
Il appelle les méthodes de création
sur son instance de builder.



Exemple de structure



Bénéfices et défauts de Builder

Bénéfice

- Rend plus facile la gestion du flot général pendant la création de l'objet

Défauts/conséquences

- Principal défaut : il y a un couplage fort parmi le builder, son produit et tout autre délégué utilisé

Variantes et Patterns reliés

- Variantes
  - Builder est soit une classe abstraite, soit une interface
  - Définir plusieurs méthodes de création pour le Builder pour fournir une variétés de façons d'initialiser la ressource construite.

- Patterns reliés :
  - Abstract Factory : cible plus les familles de produits.
  - Composite : le pattern Builder est souvent utilisé pour produire des objets composites

Intention de responsabilité

If your intent to	Apply the pattern
Centralize responsability in a single instance of a class	=> Singleton
Decouple an object from awareness of which other objects depend on it	=> Observer
Centralize responsability in a class that oversees how a set of other objects interact	=> Mediator
Let an object act on behalf of another objet	=> Proxy
Allow a request to escalate up a chain of objects until one handles it	=> Chain of Responsibility
Centralize responsability to shared fine-grained objects	=>Flyweight

Le patron Proxy

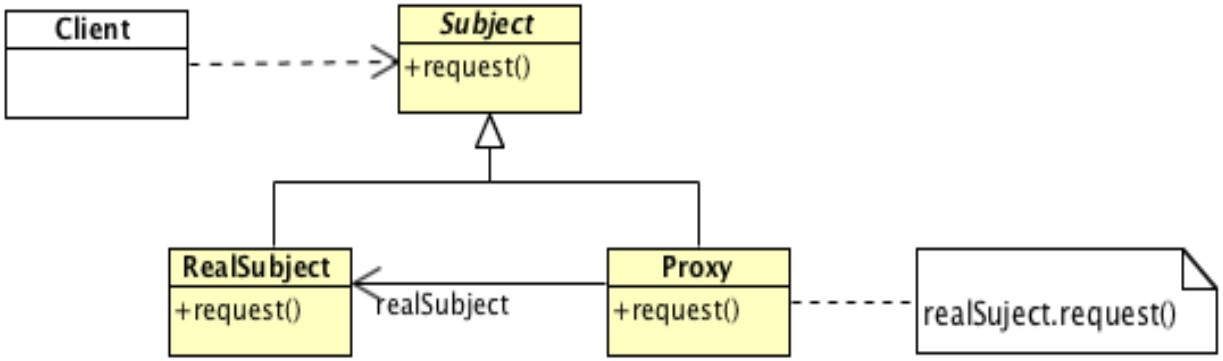
- Intention
  - Fournir un substitut afin d'accéder à un autre objet souvent inaccessible directement
- Motivation
  - Si un objet comme une image volumineuse met beaucoup de temps à se charger
  - Si un objet est situé sur une machine distante
  - Si un objet a des droits d'accès spécifiques

Les champs d'application

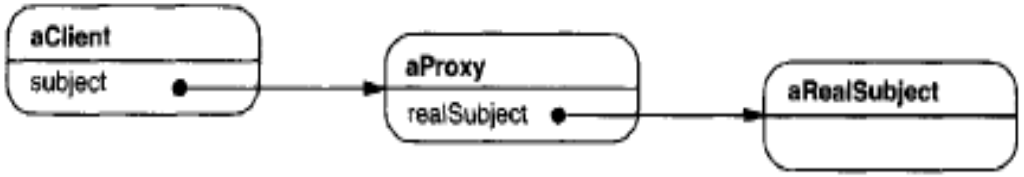
Dès qu'il y a un besoin de référencement sophistiqué ou polyvalent autre qu'un simple pointeur

- **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** pour contrôler l'accès à un objet
- **Smart reference** 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 : structure



Un diagramme d'objets possible d'une structure proxy pendant l'exécution



Proxy : Participants

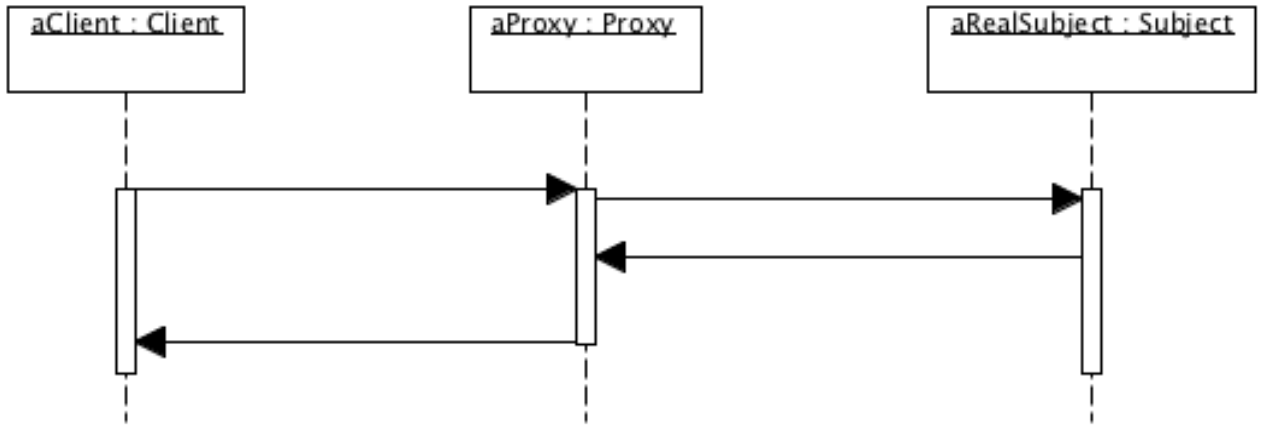
- **Proxy**
  - maintient une référence qui permet au Proxy d'accéder à l'objet `RealSubject`.
  - fournit une interface identique à celle de Subject, pour pouvoir se substituer au `RealSubject`.
  - contrôle les accès au `RealSubject` et peut être responsable de sa création ou destruction.
- **Subject**
  - 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

Les différents type de proxy : autres responsabilités

- Les remote proxies sont responsables de l'encodage d'une requête et de ses arguments afin d'envoyer la requête encodée au sujet réel dans un espace d'adresse différent.
- Les virtual proxies peuvent utiliser un cache pour l'information supplémentaire sur le sujet réel
- Les proxies de protection vérifient que l'appelant a les permissions d'accès requises pour exécuter la requête.

Proxy : collaborations

Le `Proxy` retransmet les requêtes au RealSubject lorsque c'est nécessaire, selon la forme du proxy



Conséquences

Le pattern Proxy introduit un niveau d'indirection lors de l'accès à un objet. L'indirection supplémentaire a beaucoup d'utilisations selon le type de proxy :

- Un *remote proxy* peut cacher le fait qu'un objet réside dans un espace d'adressage différent
- Un *virtual proxy* peut réaliser des optimisations telles que créer un objet sur demande
- Un *protection proxy* et les *smart references* permettent des tâches où il n'y a pas un objet réel mais simplement la réplique exacte de son sujet

Intention d'extensions(1)

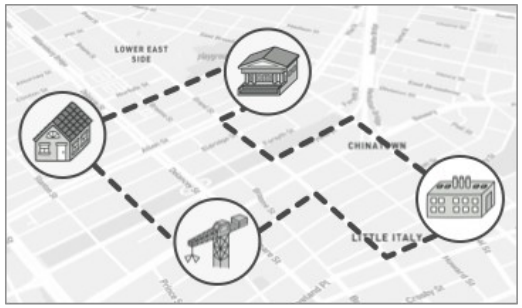
- Réutilisation comme une alternative à l'extension
  - Moins de code à maintenir
- Étendre en sous-classant
  - A la création d'une classe : garantir que c'est une extension logique et cohérente. Bien nommer les classes
- Le principe de substitution de Liskov
  - Une instance d'une classe doit fonctionner comme une instance de sa superclasse
- Extension par délégation
  - Quand une classe a des méthodes qui transmettent les appels d'opérations identiques fournies par une autre objet

Intention d'extensions(2)

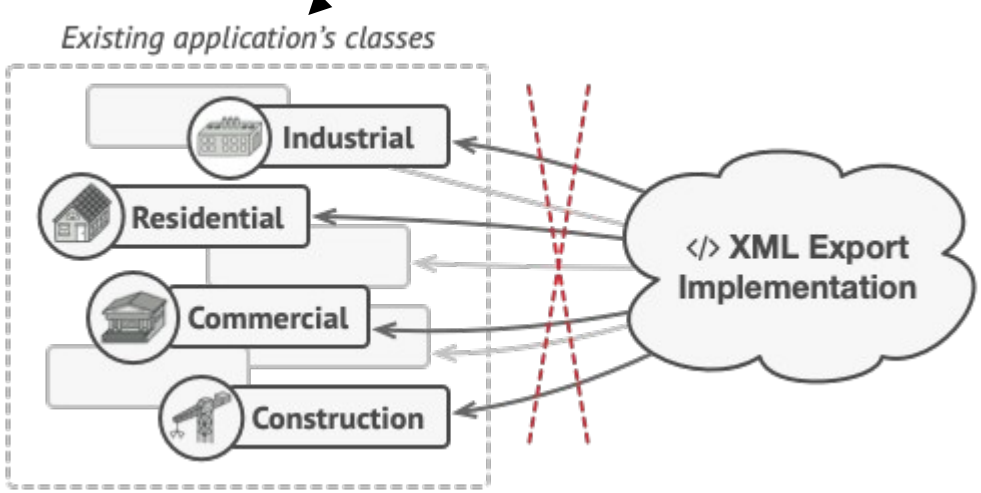
If your intent to	Apply the pattern
Allow a client to hook in an operation at a step in an algorithm	=> Template Method
Let a client outfit your code with an operation to execute in response to an event	=> Command
Attach additional responsibilities to an object dynamically	=> Decorator
Provide a way to access a collection of instances of a class that you create	=> Iterator
Allow for the addition of new operations to a class without changing the class	=> Visitor

Le pattern Visitor

Le pattern Visitor construit une opération à réaliser sur des éléments (ensemble d'objets issus de classes différentes). De nouvelles opérations peuvent ainsi être ajoutées sans modifier les classes de ces objets.

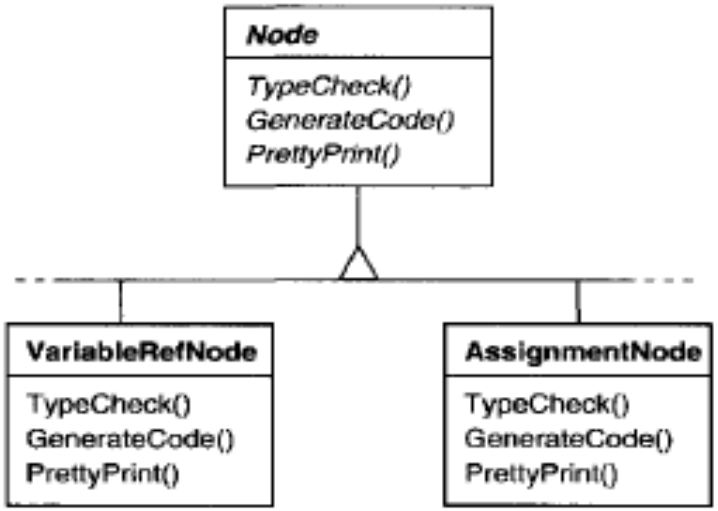


Classes stables qu'on ne voudrait pas modifier



Motivation (suite)

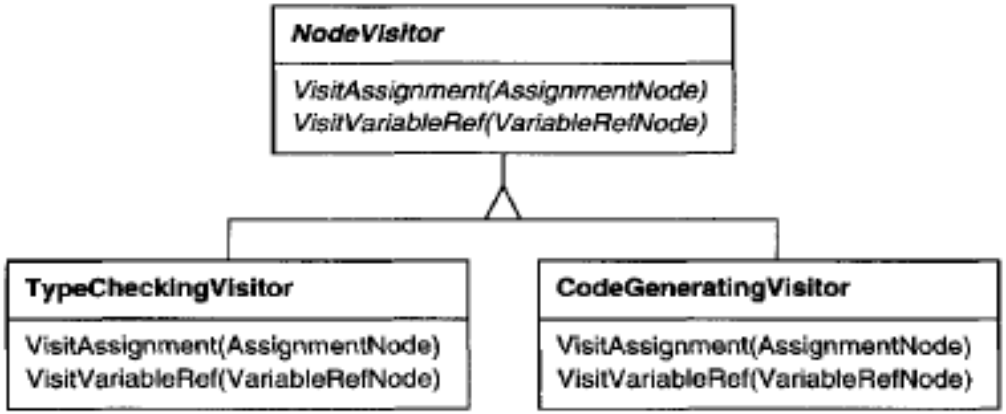
Autre exemple : un AST manipulé par un compilateur
La plupart des opérations (vérification de type, génération de code, …) auront besoin de traiter des nœuds d'un AST qui représentent de façons différentes les éléments : affectations, variables, expressions arithmétiques, ...
Ce diagramme montre une partie de la hiérarchie de la classe nœud d'un AST



Visitor

Pour faire marcher les visiteurs pour faire plus que la vérification de type, on a besoin d'une classe abstraite parente *NodeVisitor* pour tous les visiteurs d'un AST. *NodeVisitor* doit déclarer une opération pour chaque classe N*ode*.

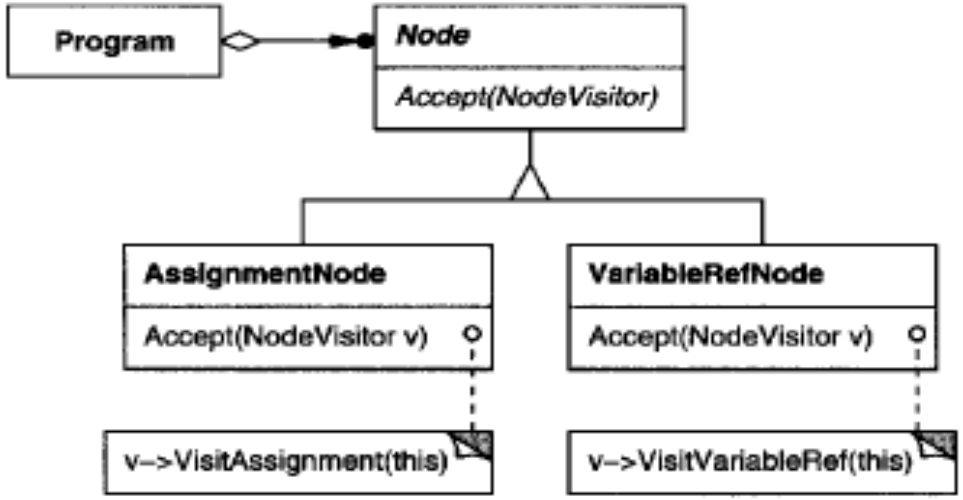
Le pattern Visitor encapsule les opérations pour chaque phase de la compilation dans un visitor associé avec cette phase.




```
class ExportVisitor implements Visitor is
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
    // ...
```

Description du pattern

Avec le pattern Visitor on définit deux hiérarchies de classes : une pour les éléments sur lesquels on opère (hiérarchie Node) et une pour les visiteurs qui définissent les opérations sur les éléments (hiérarchie NodeVisitor). On crée une nouvelle opération en ajoutant une sous-classe à la hiérarchie des visitor




```
// Client code
foreach (Node node in graph)
    node.accept(exportVisitor)

// City
class City is
    method accept(Visitor v) is
        v.doForCity(this)
    // ...

// Industry
class Industry is
    method accept(Visitor v) is
        v.doForIndustry(this)
    // ...
```

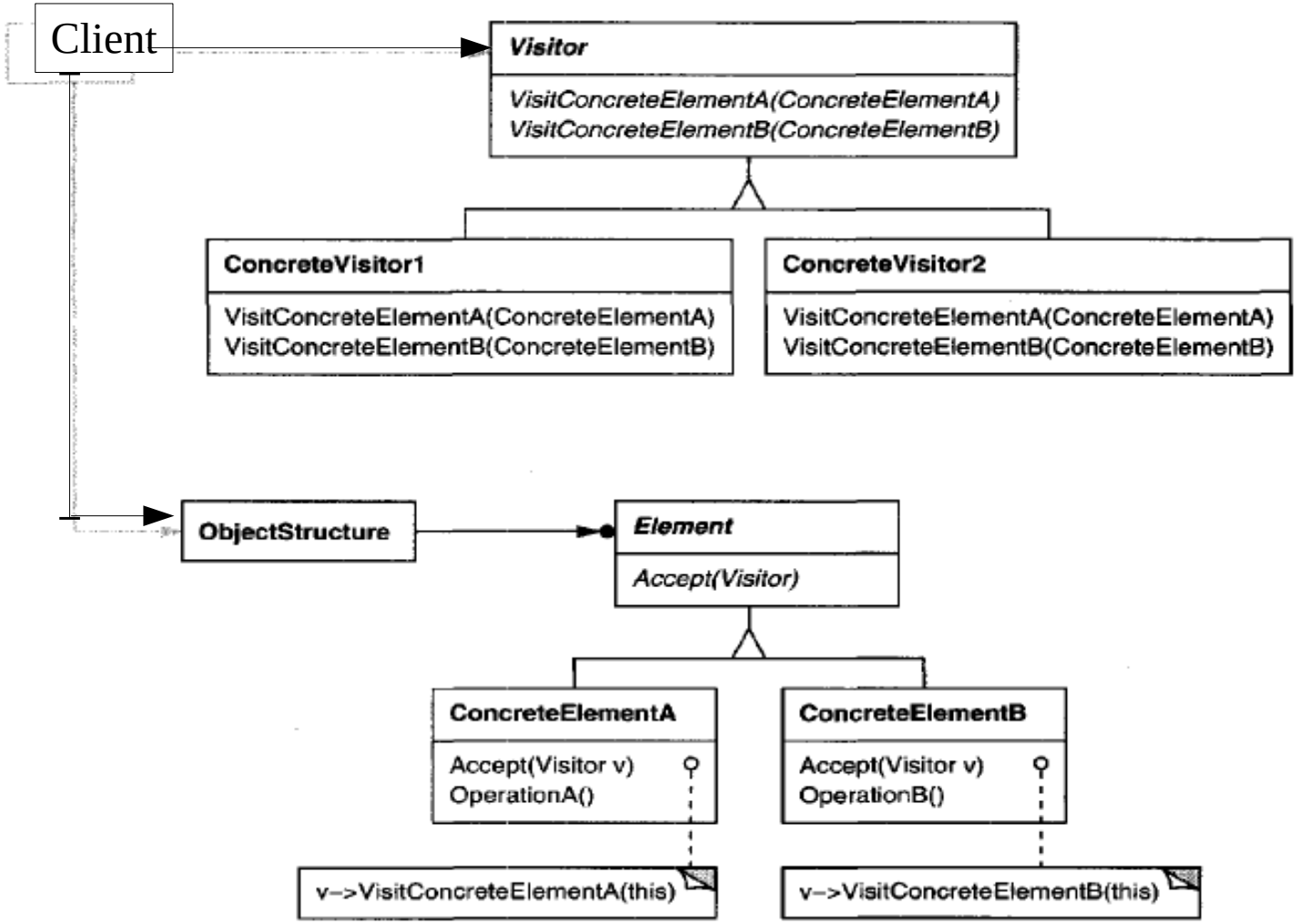
Quand utiliser Visitor ?

- Une structure d'objet contient beaucoup de classes d'objets avec des interfaces différentes, et on veut faire des opérations sur ces objets.

- Beaucoup d'opérations distinctes et non reliées ont besoin d'être réalisées sur des objets dans une structure et on veut éviter de polluer leurs classes avec des opérations. Visitor permet de garder les opérations ensemble en les définissant dans une classe.

- Les classes définissant la structure objet changent rarement, mais on veut définir souvent de nouvelles opérations dans la structure. Changer les classes de la structure demande de redéfinir l'interface de tous les visitors, ce qui coûte cher. Dans ce cas il est préférable de définir les opérations dans ces classes.

Structure de la solution de Visitor



Participants

Visitor (NodeVisitor)

- Déclare une opération visit pour chaque classe de `ConcreteElement`. Le nom de l'opération et la signature identifient la classe qui envoie la requête `visit` au visitor.

ConcreteVisitor(TypeCheckingVisitor)

- Implémente chaque opération déclarée par Visitor. Chaque opération implémente un fragment de l'algorithme défini par la classe d'objets correspondante dans la structure.
- Fournit le contexte pour l'algorithme et stocke l'état local qui souvent accumule les résultats durant la traversée de la structure.

Element (Node)

- Définit une opération `accept` qui prend un visiteur en argument

ConcreteElement (Assignment Node, VariableRefNode)

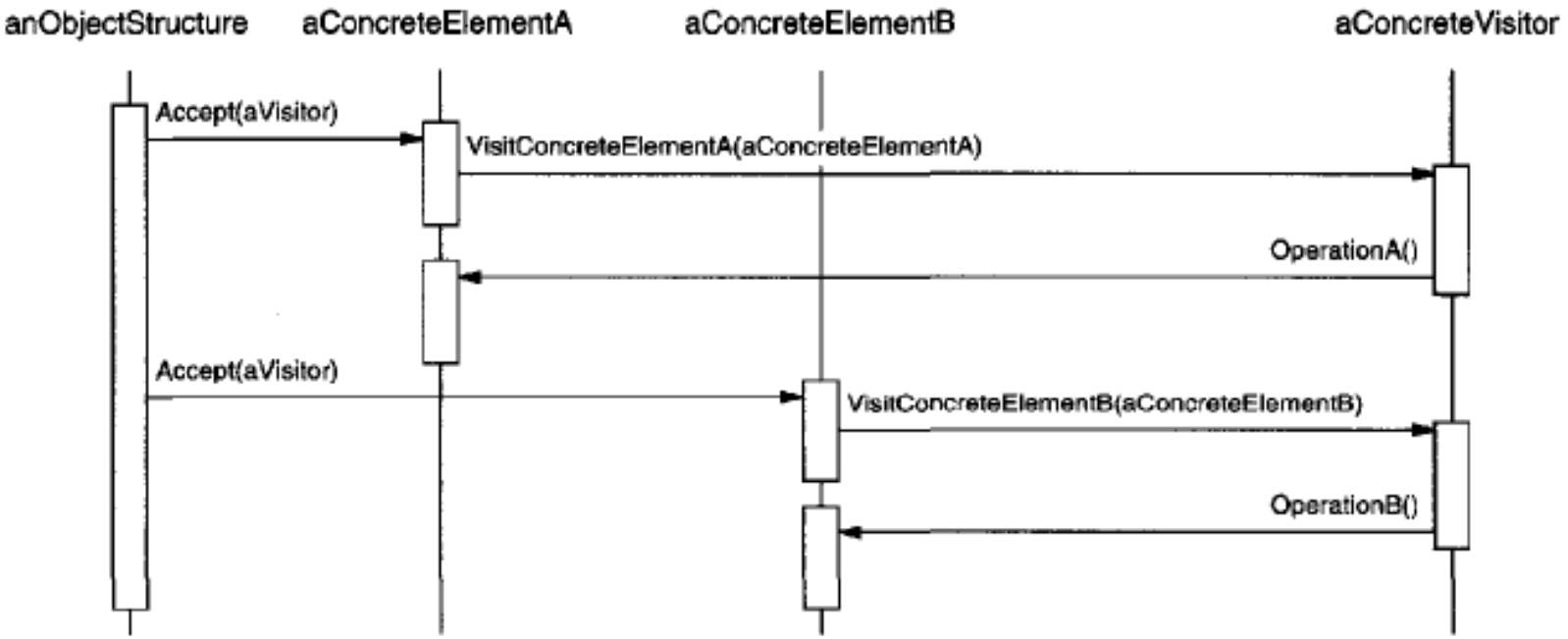
- implémente une opération `accept` qui prend un visiteur en argument

ObjectStructure (Program)

- Peut énumérer ses éléments
- Peut fournir une interface de haut-niveau pour permettre au visitor de visiter ses éléments
- Peut être soit un composite (voir pattern Composite) ou une collection telle qu'une liste ou un ensemble

Collaborations

Le client doit créer le ConcreteVisitor
Quand un élément est visité, il appelle l'opération Visitor qui correspond à sa classe
Il se passe lui-même en paramètre pour laisser le visitor accéder à son état.



Collaboration entre une structure d'objet, un visitor et deux éléments

Conséquences

- Ajout de nouvelles opérations : facile
- Ajout de novelles classes ConcreteElement : difficile
- Visitor traverse des structures où les éléments sont de types complètement différents, ce qui n'est pas possible avec un itérateur
- Accumulation d'états dans le visiteur plutôt que dans des arguments
- Suppose que l'interface de `ConcreteElement` est assez riche pour que le visiteur fasse son travail. Souvent la conséquence brise l'encapsulation car le pattern nous force à fournir des opérations publiques qui accèdent à l'état interne d'un élément.