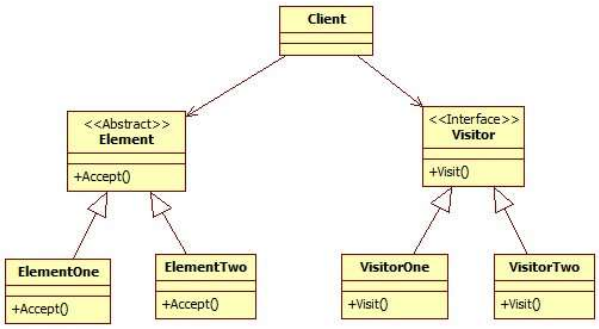
Le Pattern *Visiteur*

Le pattern **Visiteur** rentre dans les Patterns de **Comportement** du Gof. Il cherche à organiser les comportements des différents objets. Il vise à séparer le traitement des modèles ce qui garantit une maintenance viable dans le cas où les opérations s'ajoutent plus rapidement que n'évolue la classe grâce à une simplification de l'ajout d'opérations dans une structure large et complexe.

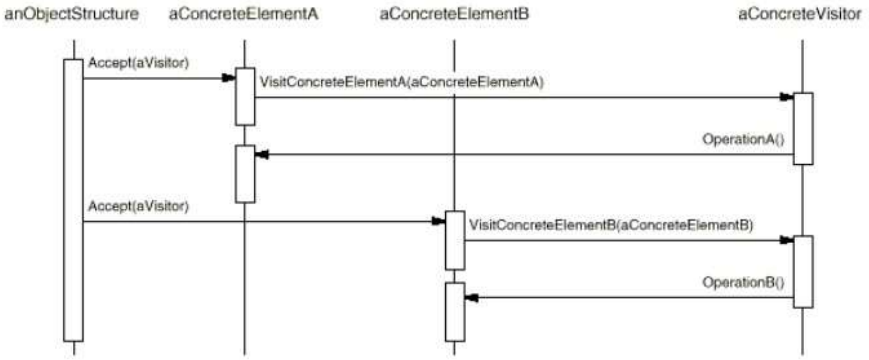
Sa structure :



Des visiteurs : Qui contiennent les opérations et implémentent une méthode ***visite*** pour chaque classe visitable. *Cette méthode prend en paramètre la classe visée.*

Des classes visitables : Qui contiennent les données Implémente une méthode ***accepte*** unique. Cette méthode prend en paramètre les visiteurs et fait appel à leurs méthodes ***visite***, passant son objet comme paramètre.

! Double Dispatch :



Le concept le plus important de ce cours. Cela permet que le choix de la méthode invoqué se fasse en fonction de ***deux objets à la fois***.

Il permet au programme de réagir en fonction de deux types d'objets en interaction : L'objet qui porte la méthode et l'objet passé en paramètre de cette méthode. Ceci est particulièrement utile dans le pattern ***Visiteur*** pour choisir la méthode appropriée à exécuter en fonction du type de l'élément visité et du type du ***Visiteur***.

Son point fort : SOLID	Ses limites
• **Single Responsibility Principle (SRP) :** Sépare la logique métier (les calculs, traitements, etc.) des objets eux-mêmes. Chaque classe a une responsabilité claire : les éléments concrets stockent les données et le visiteur implémente le comportement. • **Open/Closed Principle (OCP) :** Permet d'ajouter de nouveaux comportements sans modifier les classes des éléments déjà établis. Cela rend le système extensible sans devoir réécrire le code existant. • **Liskov Substitution Principle (LSP) :** Chaque visiteur concret peut être substitué par un autre sans affecter le fonctionnement du code. • **Interface Segregation Principle (ISP) :** Les méthodes ne sont pas implémentés dans les classes mais dans des interfaces spécifiques • **Dependency Inversion Principle (DIP) :** Repose sur une dépendance abstraite (l'interface Visiteur) plutôt qu'une dépendance directe à des implémentations concrètes.	• **Restreint l'encapsulation :** Peut parfois forcer l'ajout de getters sur des attributs qui ne le nécessitait pas à la base • **Augmente le couplage :** Les visiteurs dépendent fortement des types et de la structure des classes visitées • **Conception complexe :** Nécessite une complexité inutile pour des structures simples (en forme et e n nombre) en terme de classe et de type • **Maintenabilité des classes :** Complexifie la modification des classes visitables, impliquant une modification de l'interface et de ses implémentations