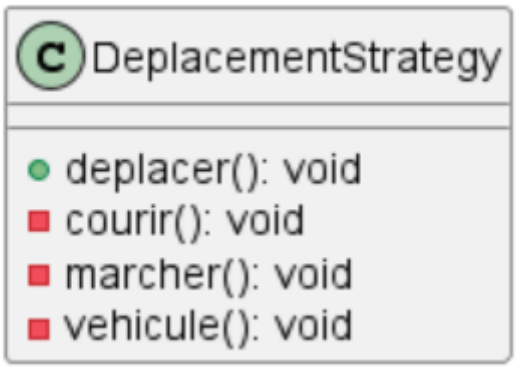
Pattern Strategy

Le pattern strategy appartient à la catégorie comportementale du modèle de conception GoF. Ce type de modèle fournit des solutions pour une meilleure interaction entre les objets.

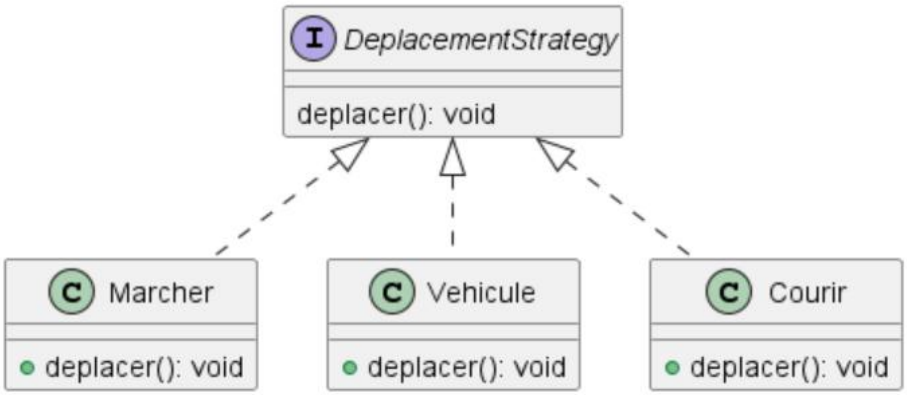
Il permet à un objet d'utiliser différentes méthodes pour accomplir une tâche, et de changer de méthode à la volée sans modifier l'objet qui l'utilise.

Quand l'utiliser ?

Le pattern Strategy est utilisé quand une classe utilise un nombre multiple de méthodes. Nous allons l'illustrer avec un exemple. Dans un jeu où le personnage peut se déplacer de différentes manières : marcher, courir, prendre un véhicule. On créait une classe 'DeplacementStrategy' avec la méthode 'déplacer()', 'courir()', 'marcher()' et 'vehicule()'.



Avec le pattern Strategy nous partageons cette classe en la transformant en interface et créons directement les classes: 'Marcher', 'Courir' et 'Vehicule'. Chacune de ces classes implémente l'interface 'DeplacementStrategy' ainsi que sa méthode 'deplacer' différemment.



Comme cela nous ne sur chargeons pas l'interface 'DeplacementStrategy'.

Diagramme de classe généralisé :

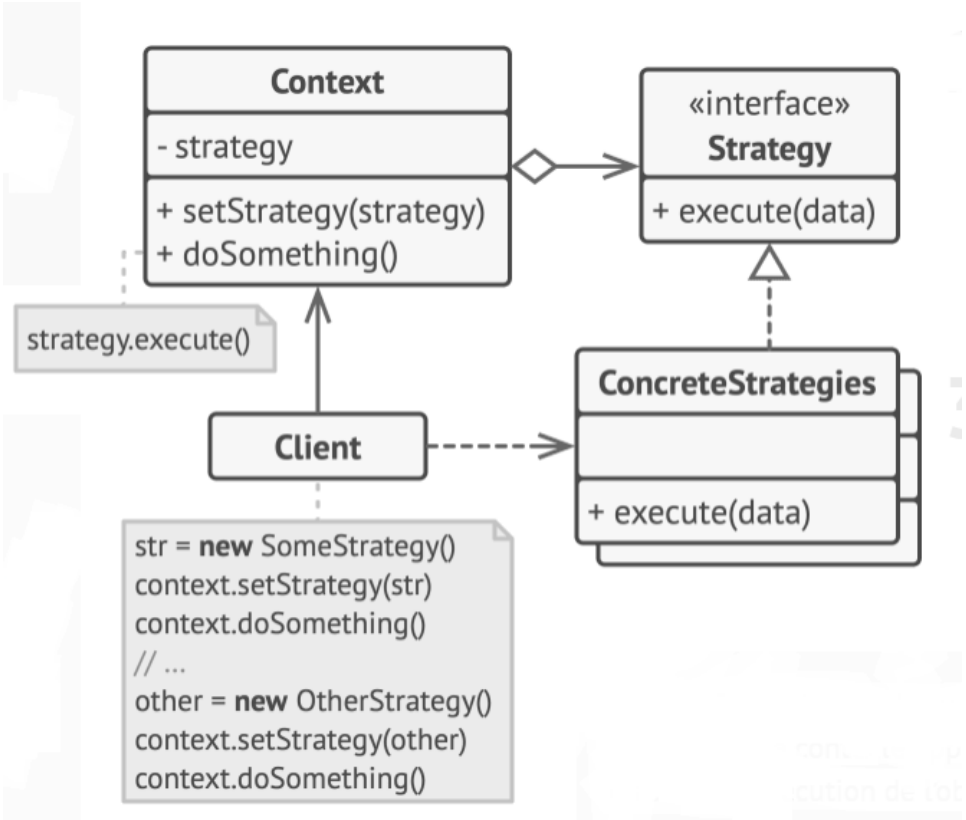
Le Contexte garde une référence vers une des stratégies concrètes et communique avec cet objet uniquement au travers de l'interface stratégie.

L'interface Stratégie est commune à toutes les stratégies concrètes. Elle déclare une méthode que le contexte utilise pour exécuter une stratégie.

Les Stratégies Concrètes implémentent différentes variantes d'algorithmes utilisées par le contexte.

Chaque fois qu'il veut lancer un algorithme, le contexte appelle la méthode d'exécution de l'objet stratégie associé. Le contexte ne sait pas comment la stratégie ne fonctionne ni comment l'algorithme est lancé.

Le Client crée un objet spécifique Stratégie et le passe au contexte. Le contexte expose un setter qui permet aux clients de remplacer la stratégie associée au contexte lors de l'exécution.



Principaux problèmes du Pattern :

Les principaux problèmes du pattern sont qu'on a plusieurs manières possibles d'accomplir une même action (ex. : calculer un tarif, appliquer une réduction, choisir un algorithme de tri, déterminer un chemin dans une carte...). Ensuite, sans le pattern Strategy, le code devient vite rempli de if/else ou de switch pour choisir quelle méthode utiliser. Et donc, chaque fois qu'on ajoute une nouvelle variante, il faut modifier le code existant, ce qui viole le principe Ouvert/Fermé (OCP) et augmente les risques de bugs.

Principe SOLID que le pattern Strategy suit:

Single Responsibility Principle (SRP) les responsabilités sont bien séparées dans différentes classes.

Open Closed Principle (OCP) les classes sont ouvertes aux extensions et fermées aux modifications.

Dependency Inversion Principle (DIP) les détails dépendent d'abstractions.