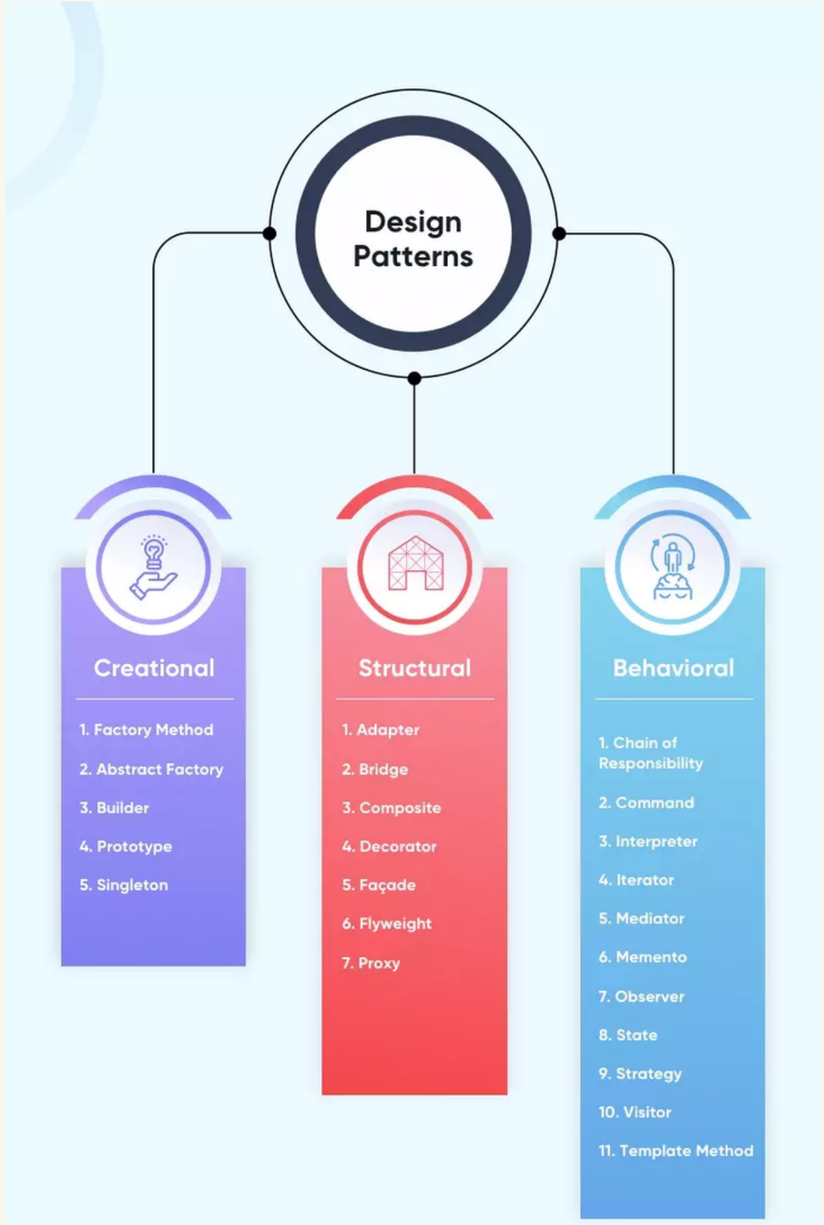
DESIGN PATTERN

Présenté par Delesvaux Allan, Gonzales Andy, PINEL-GUINARD Bertil, MONTERIN Maxime, CAZALS Mathias, LEINER Lucas



Sommaire

QU'ALLONS NOUS ABORDER

- **COMPOSITE :**
  - Mise en situation
  - Problème
  - Solution
  - Présentation de Composite
  - Explication
- **STRATEGIE :**
  - Mise en situation
  - Problème
  - Solution
  - Présentation de Stratégie
  - Explication

COMPOSITE

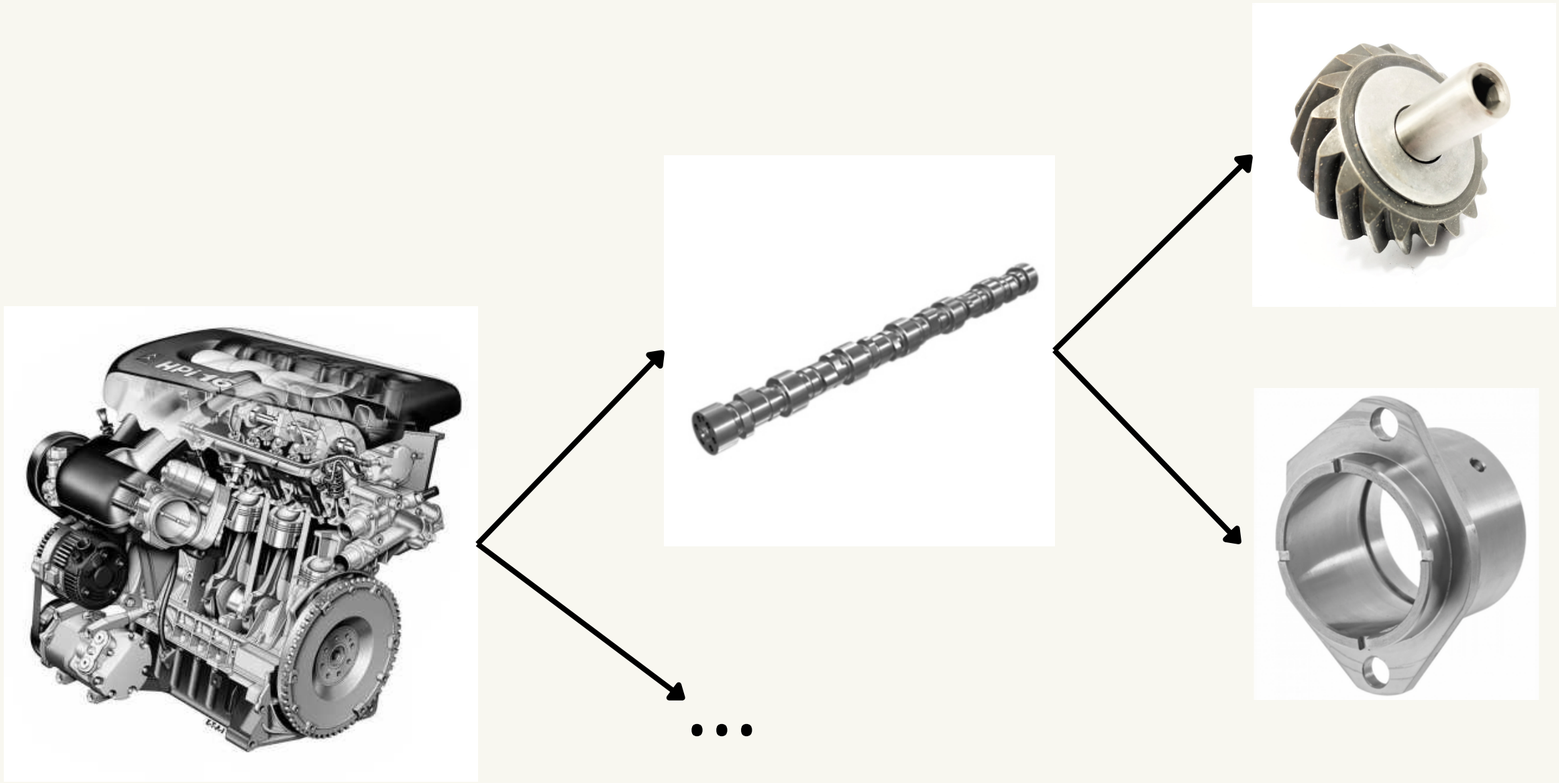
Imaginez... Un garagiste

Afin de facturer ses services, vous devez comptabiliser toutes les pièces internes que vous avez installé... La tâche est longue et pas très amusante, vous voulez donc l'automatiser en passant par un développeur (pas vous car vous êtes garagiste dans l'histoire).

Vous =>

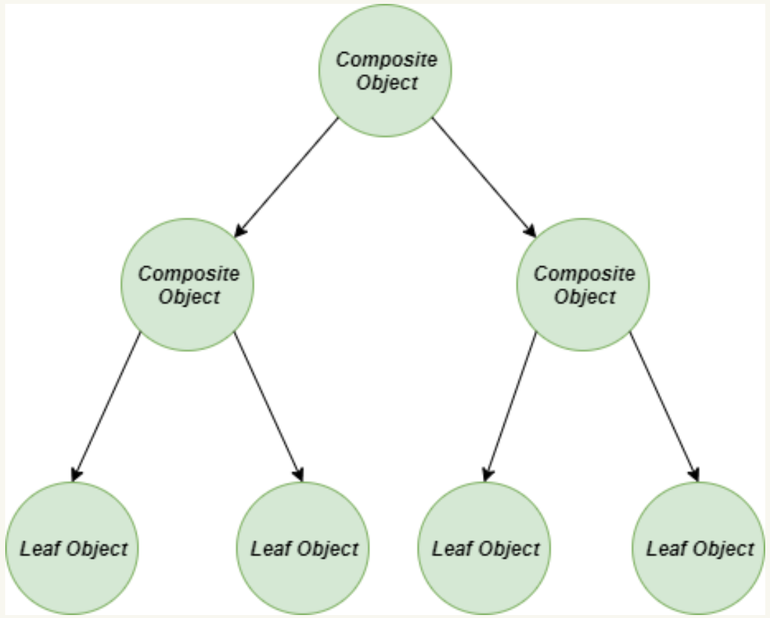
Problèmes

LA TÂCHE EST PLUS COMPLIQUE A DEVELOPPER QU'A FAIRE DANS LA VIE DE TOUS LES JOURS



Le garagiste est un flemmard c'est le cas de le dire, mais bon on ne refuse jamais un travail, le problème et que même s'il parait de simple de juste additionner les prix ce n'est pas autant simple car un moteur par exemple et composé d'autre objet qui ont eux aussi un prix.

Solution



INTERFACE

Une interface implémenté par chaque objet sans enfant (ici dans notre cas la plus petite pièce d'un mateur par exemple).

PRIX DU MOTEUR

A chaque fils sans enfant on demande leur prix, le prix d'un objet qui a plusieurs fils est la addition des objets sans fils.

COMPOSITE

Cette solution a bien évidemment un nom : Composite, sinon on ne ferait pas un oral sur les Design Pattern.

Présentation de Composite

Composite

Design Pattern de type structurel, permettant la représentation sous type d'arbre

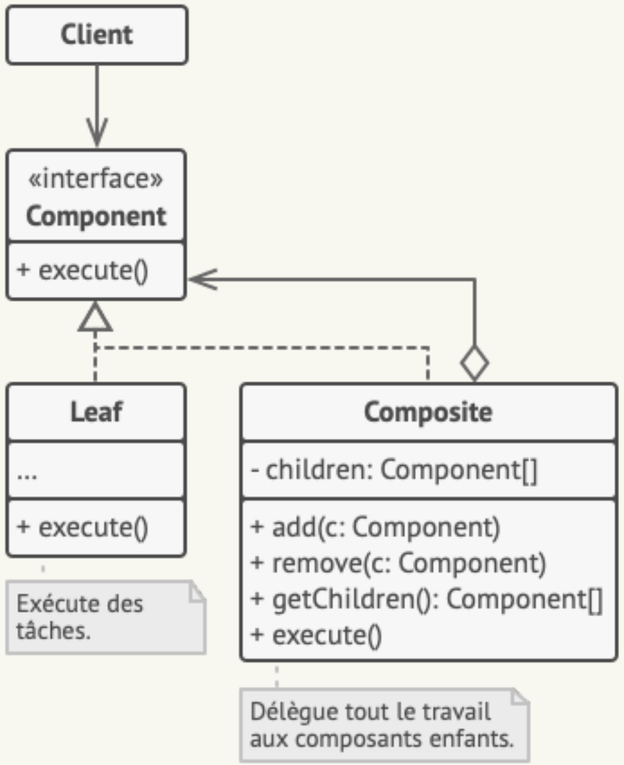
But

Permet de rendre flexible et extensible un projet, le rajout d'objet est facilement réalisable, tout réécrire sera pas nécessaire

Points faibles

Il ne faut pas implémenter ce Design Pattern n'importe où, il est parfois difficile de définir une interface commune à chaque objet sans la rendre trop abstraite

Explication



COMPONENT

Component est l'interface commune à tous les objets, c'est lui qui dire les opérations communes à chaque objet dans notre situation cela pourrait être : getPrice()

COMPOSITE

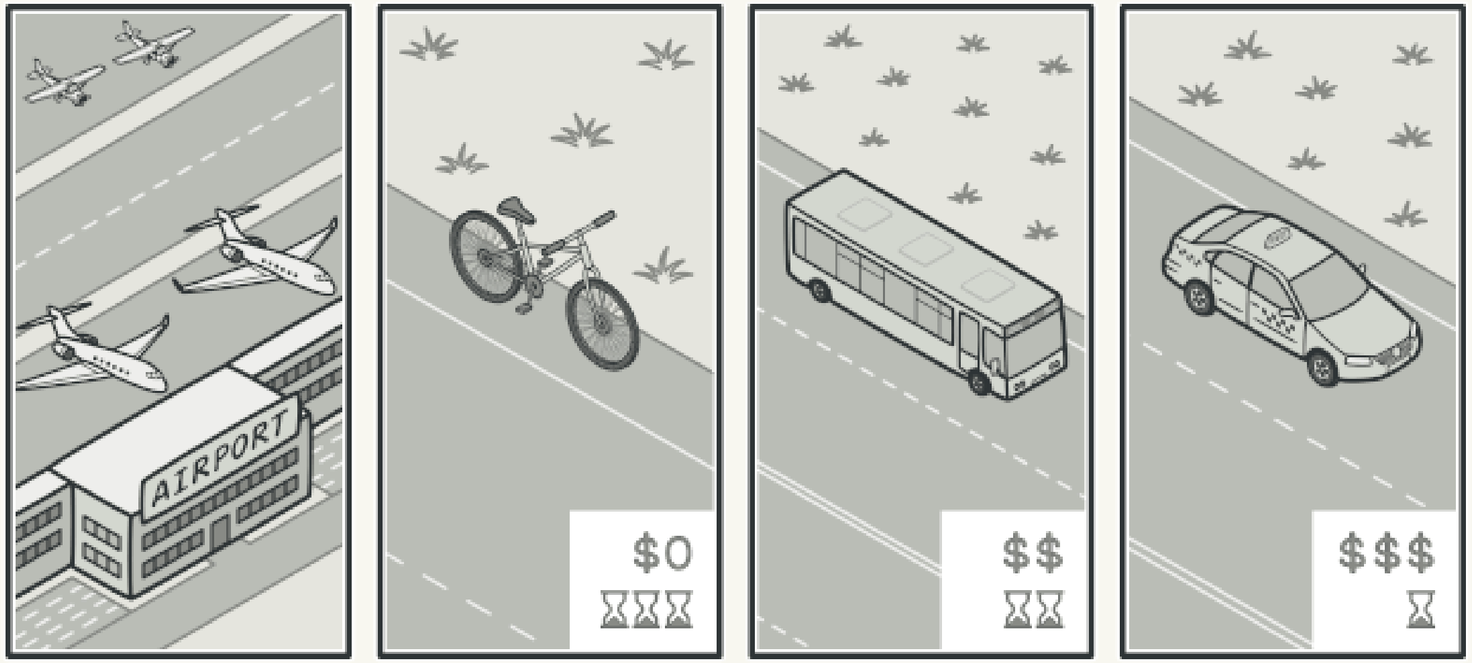
Composite agit comme un conteneur, il n'a pas besoin et d'ailleurs ne connait pas la classe de ses enfants il utilise Component pour avec intéragir avec eux, ici pour obtenir le prix de chacun.

LEAF

Les feuilles (Leaf) sont les objets sans fils, ce sont eux qui implémente Component.

STRATÉGIE

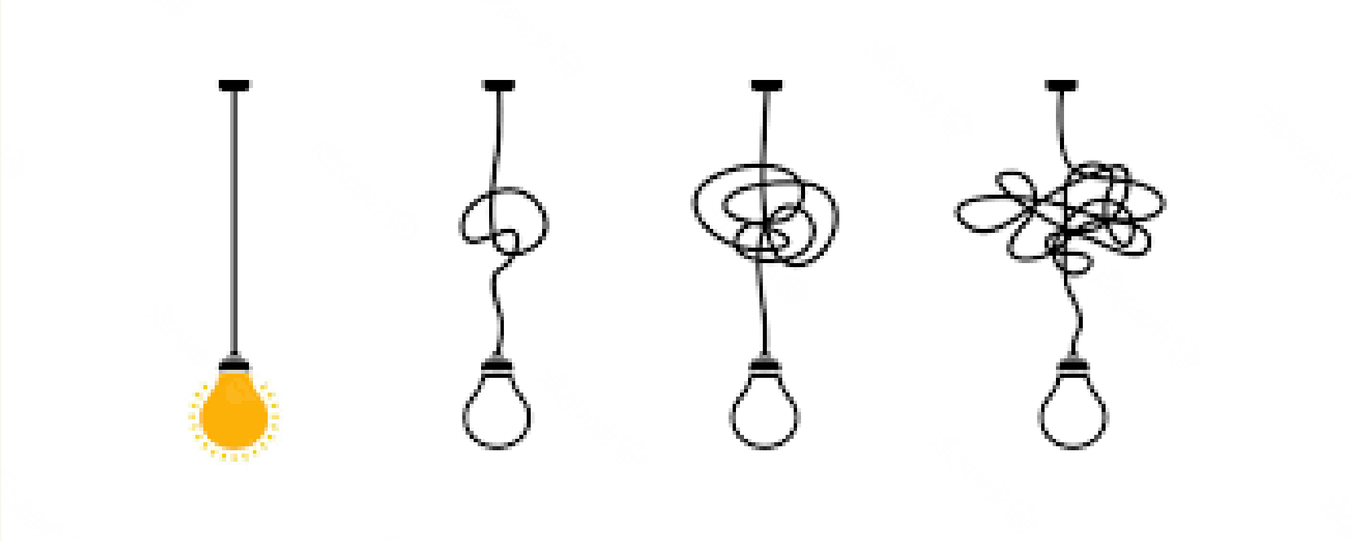
Mise en Situation



Création d'une application de Gps pour les voyages en voiture

Ajouts de fonctionnalité pour répondre aux besoins utilisateurs

Problème



Pour chaque ajout de fonctionnalité, cela demande un nouvel algorithme

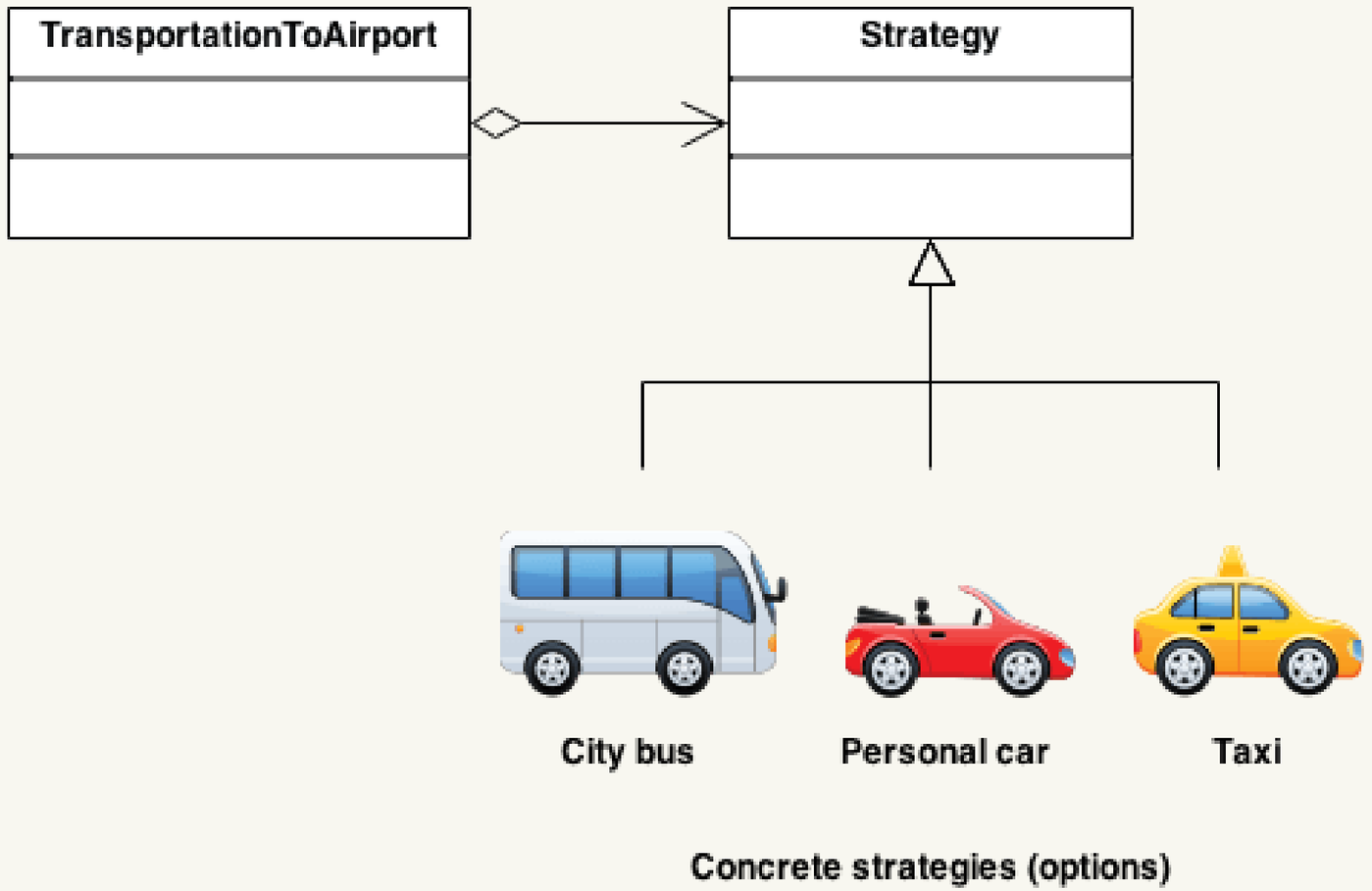
Tous ces ajouts d'algorithmes pour résoudre le même problème alourdi le code

La maintenance devient compliqué. Et chaque ajout augmente le risque de créer des bugs qui casseront le code.

Solution

On sépare tout les algorithmes qui se trouvaient dans un même classe dans différentes classes.

On garde un classe principale qui se chargera d'adopter la "stratégie" correspondant à la tâche demandée.



Présentation de Stratégie

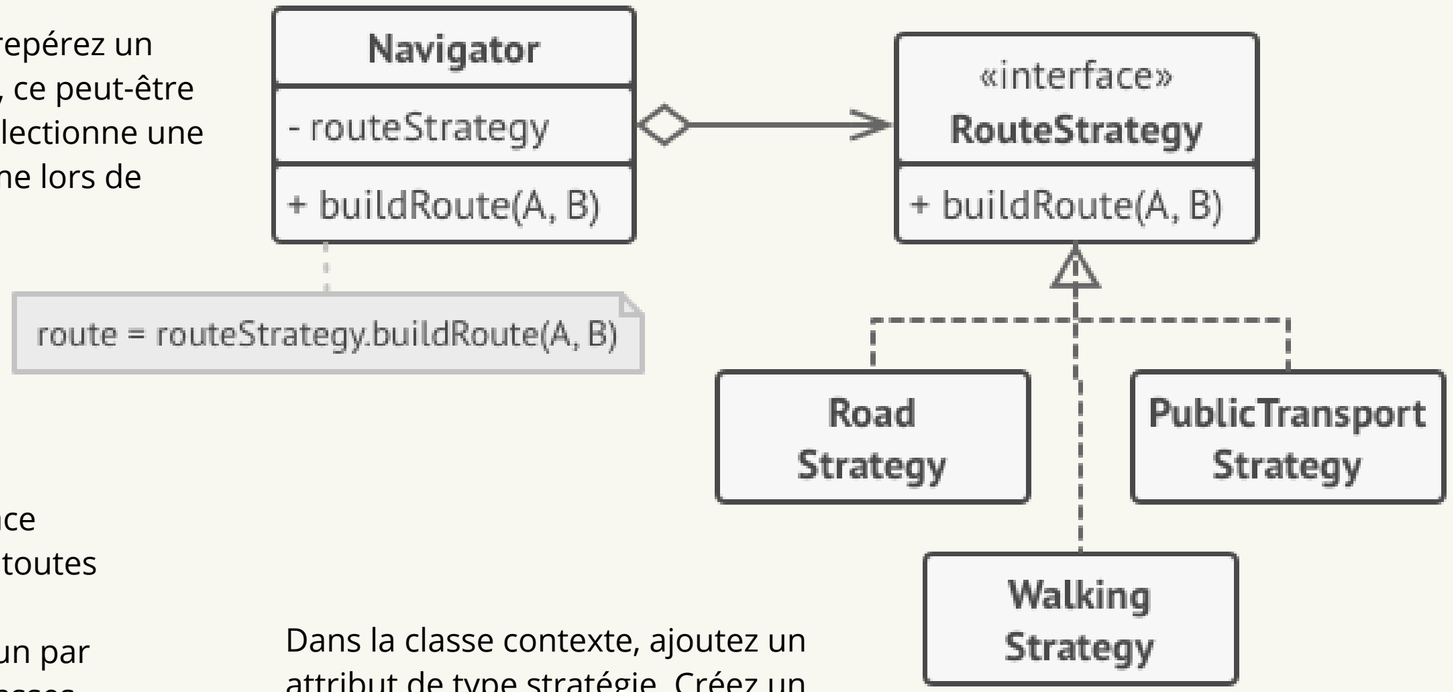
Stratégie est un patron de conception comportemental qui permet de définir un famille d'algorithmes que l'on classe séparément et dont leurs objets sont interchangeable.

BUT

La classe originale doit posséder un attribut référancant vers chacune des stratégies différentes et permettrant de déléguer les taches en fonction à celles-ci. Le client envoie donc la stratégie pour savoir comment se réalise la tâche.
On peut alors modifier n'importe quels algorithmes ou en ajouter facilement sans impacter le reste du programme que l'on ne veut pas modifier.

Explication

Dans votre classe contexte, repérez un algorithme qui change souvent, ce peut-être un gros bloc conditionnel qui sélectionne une variante du même algorithme lors de l'ecxécution.



Déclarez ensuite un interface "Stratégie" qui est commune à toutes les variantes.
Extrayez tous les algorithmes un par un et rangez les dans leurs classes respectives. Toutes ces classes doivent implémenter l'interface "Stratégie".

Dans la classe contexte, ajoutez un attribut de type stratégie. Créez un setter pour modifier cet attribut. La classe contexte ne doit pas manipuler l'objet stratégie uniquement à travers l'interface stratégie.

Enfin, les clients du contexte doivent l'associer avec une stratégie adaptée au comportement attendu.