Introduction aux Design Patterns (Suite)

Rezak AZIZ
rezak.aziz@lecnam.net

CNAM Paris

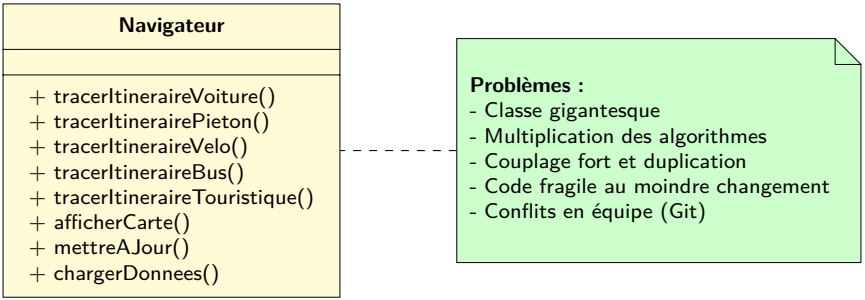
Les patterns de Comportement

Section 1

Strategy

Problème

Une seule classe **Navigateur** contient plusieurs variantes d'un même algorithme : itinéraire voiture, piéton, vélo, bus, etc. Cette classe devient énorme, difficile à maintenir et très fragile.



Stratégie — Intention

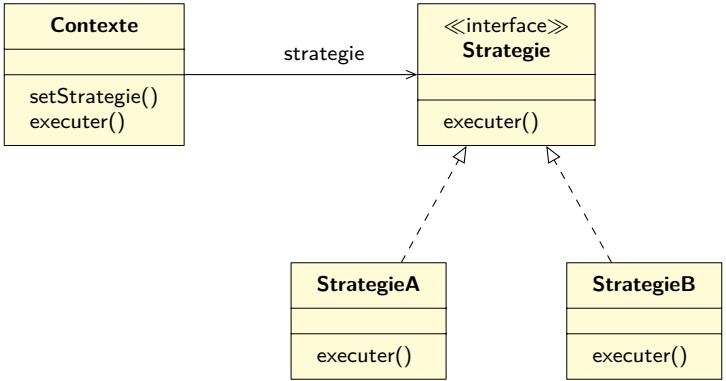
Intention générale

Permettre de définir une **famille d'algorithmes interchangeables**, les encapsuler dans des classes indépendantes et permettre au client de **changer dynamiquement** l'algorithme utilisé par le contexte.

Ce que l'on veut éviter

Les longues structures conditionnelles : `if (mode == voiture) ... else if (mode == velo) ...`

Stratégie — Structure



Stratégie — Possibilités d'application

Quand l'utiliser ?

- Plusieurs variantes d'un même algorithme (tri, compression, IA...).
- Comportement modifiable dynamiquement (jeu, UI, math).
- Remplacer un gros bloc de conditions par du polymorphisme.
- Isolation des algorithmes complexes.
- Respecter le principe *Ouvrir / Fermer*.

Stratégie — Avantages et inconvénients

Avantages

- + Algorithme interchangeable à l'exécution.
- + Réduction des conditions complexes.
- + Facile d'ajouter de nouveaux comportements.
- + Tests unitaires simplifiés.
- + Fortement découplé.

Inconvénients

- - Prolifération de classes.
- - Le client doit connaître la stratégie appropriée.
- - Peut sembler lourd pour de petits algorithmes.
- - Peut être remplacé par des lambdas dans certains langages modernes.

Section 2

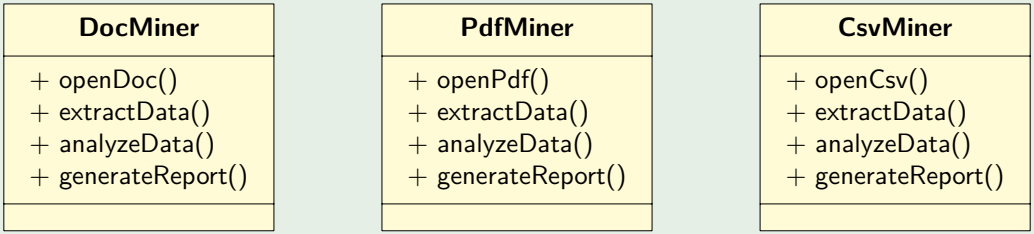
Template Method

Problème à résoudre

Observation

Dans de nombreuses applications, plusieurs classes implémentent un algorithme **presque identique**, avec seulement quelques différences.

Exemple : Data Mining (PDF, DOC, CSV)



Problèmes

Algorithmes presque identiques → **forte duplication de code**.

Intention du patron

Objectif principal

Définir le **squelette d'un algorithme** dans une classe mère, tout en laissant aux sous-classes la possibilité de **redéfinir certaines étapes**.

Idée clé : On *fixe l'enchaînement* des opérations, mais on laisse varier *leur contenu*.

Ce que l'on évite

```
if (format == PDF) ... else if (format == CSV) ...
```

Patron de Méthode — Solution

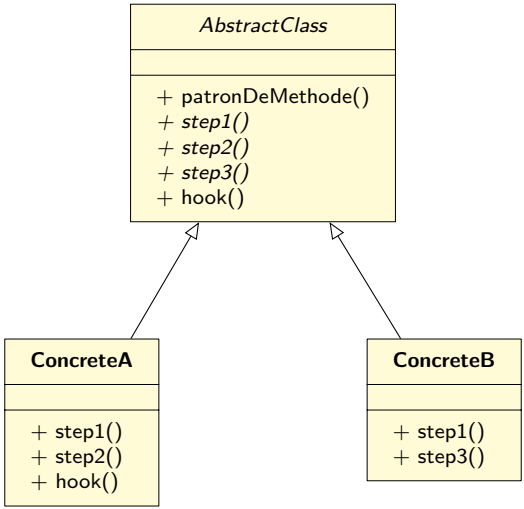
Principe

- Décomposer l'algorithme en étapes.
- Placer l'enchaînement de ces étapes dans une méthode **finale** : la méthode patron.
- Déclarer certaines étapes comme :
  - abstraites → à implémenter par les sous-classes ;
  - facultatives → avec une implémentation par défaut ;
  - crochets (hooks) → implémentation vide pouvant être redéfinie.

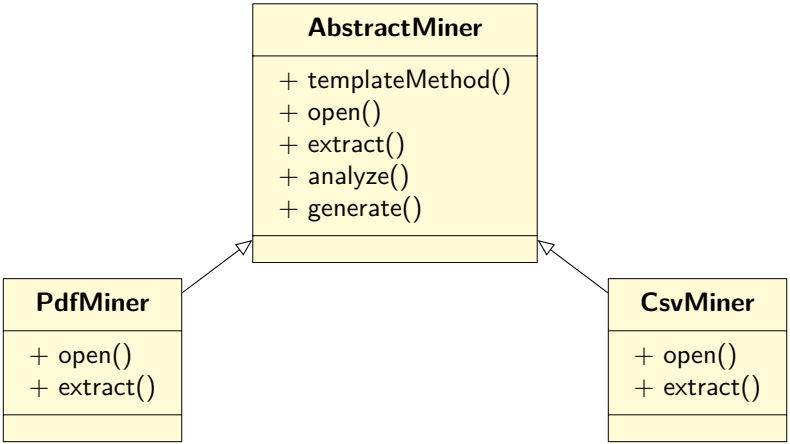
Conséquence

La structure de l'algorithme est **fixée**, mais ses étapes peuvent varier selon les sous-classes.

Patron de Méthode — Structure



Structure du Template Method



Patron de Méthode — Applications

Quand utiliser ce patron ?

- Lorsqu'un algorithme est identique dans plusieurs classes, avec seulement quelques variations.
- Lorsque vous voulez :
  - **partager du code** entre les sous-classes ;
  - **forcer un ordre d'exécution** ;
  - fournir des **points d'extension contrôlés**.
- Lorsqu'un client veut étendre seulement certaines étapes.
- Lorsque vous voulez remplacer un gros algorithme monolithique par des étapes bien définies.

Avantages

+ Code factorisé

Les étapes communes sont dans la classe mère.

+ Structure stable

L'algorithme est clair et figé.

+ Extensibilité contrôlée

Les sous-classes personnalisent seulement ce qui doit varier.

+ Polymorphisme

Le client n'a plus besoin de `if/else` ou `switch`.

Inconvénients

- Structure rigide

Impossible de modifier l'ordre des étapes.

- Risque de LSP violation

Si une étape "par défaut" n'a pas de sens pour une sous-classe.

- Peut devenir complexe

Beaucoup d'étapes $\rightarrow$ classe abstraite difficile à maintenir.