Design Pattern Strategy

Zacharie Dubrulle
Bastien Jacotin
Soane Crespin
Dorian Loizeau

Université de Limoges

Sommaire

Strategy, un membre du Gang Of Four

Définition : Un pattern est une solution générale et réutilisable à un problème courant dans un contexte donné.

On retrouve 3 grandes catégories de patterns :

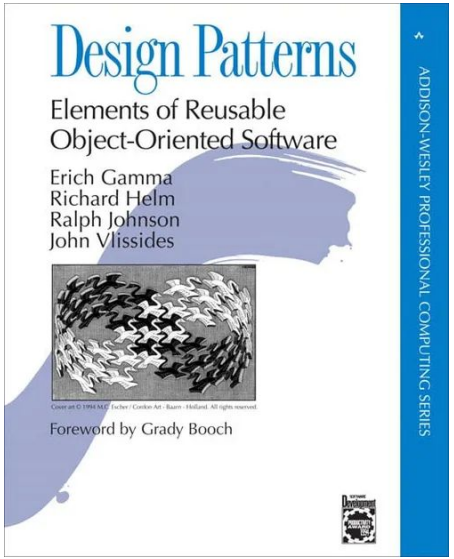
- Patterns de création (Builder, Abstract Factory..)
- Patterns de structure (Adapter, Composite..)
- Patterns de comportement (Etat, Strategy..)

Avantages :

- Meilleure organisation et structure du code
- Facilité de maintenance et d'évolution

Inconvénients :

- Risque de sur-ingénierie s'ils sont mal utilisés
- Compliqué à comprendre pour les débutants



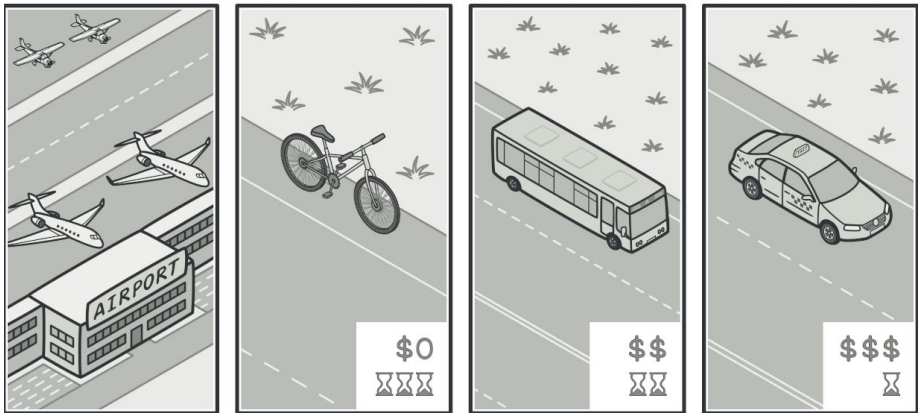
Énoncé du besoin selon une IA

Besoin principal :

Le besoin fondamental du pattern Strategy est de permettre la **variation du comportement** d'un objet de manière **flexible** et **extensible** sans toucher à sa structure. Il est particulièrement utile lorsque l'on doit effectuer des actions similaires de plusieurs façons différentes, et qu'il serait contre-productif de mélanger toutes ces variantes dans une même classe.

Introduction au pattern Strategy : Le pattern Strategy permet de définir un ensemble de classes qui implémentent une interface commune. Ces classes représentent des comportements spécifiques qui peuvent être sélectionnés et utilisés dynamiquement en fonction des besoins du contexte.

Pour illustrer l'utilité de ce pattern, voici un exemple simple :
Julien veut aller à l'aéroport mais il se demande quel moyen serait le plus rapide et le moins coûteux afin de ne pas arriver en retard ?

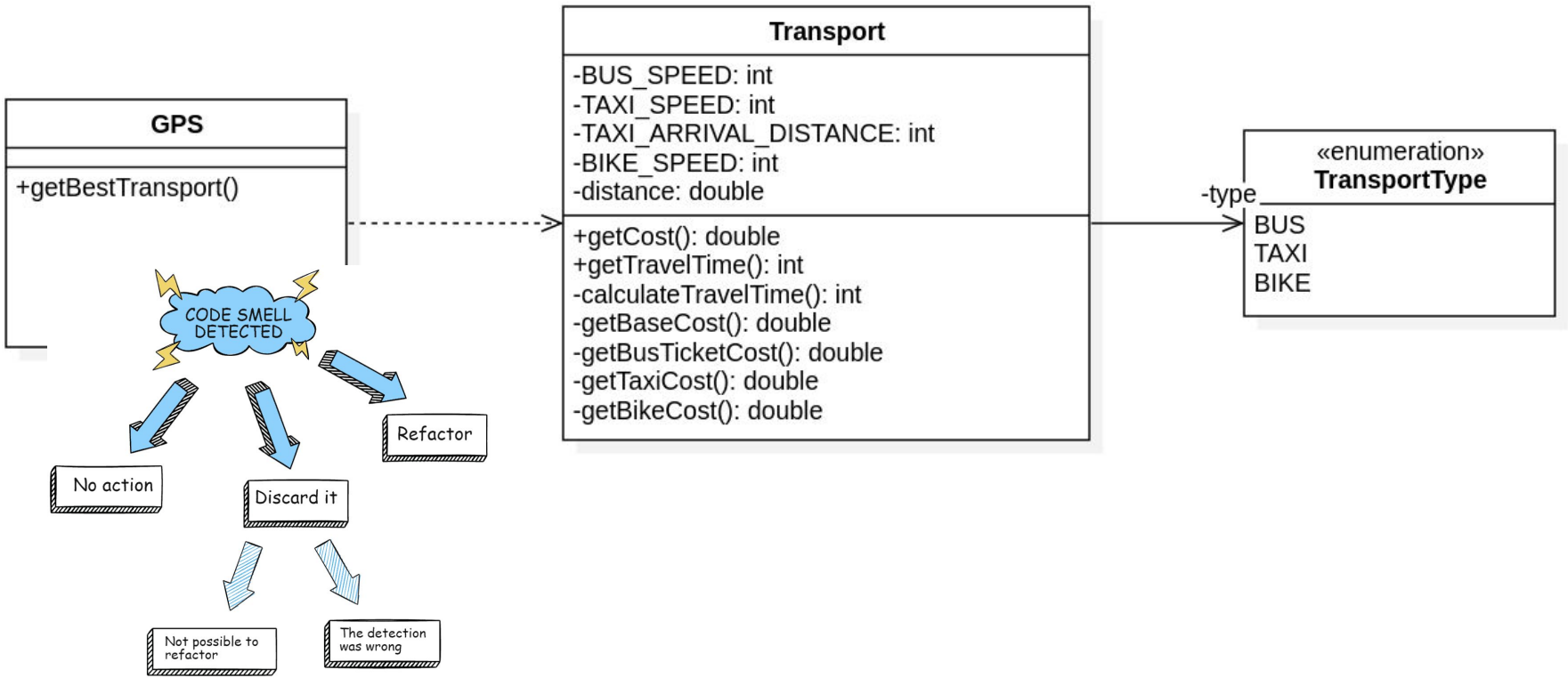


Aller à l'aéroport de diverses manières

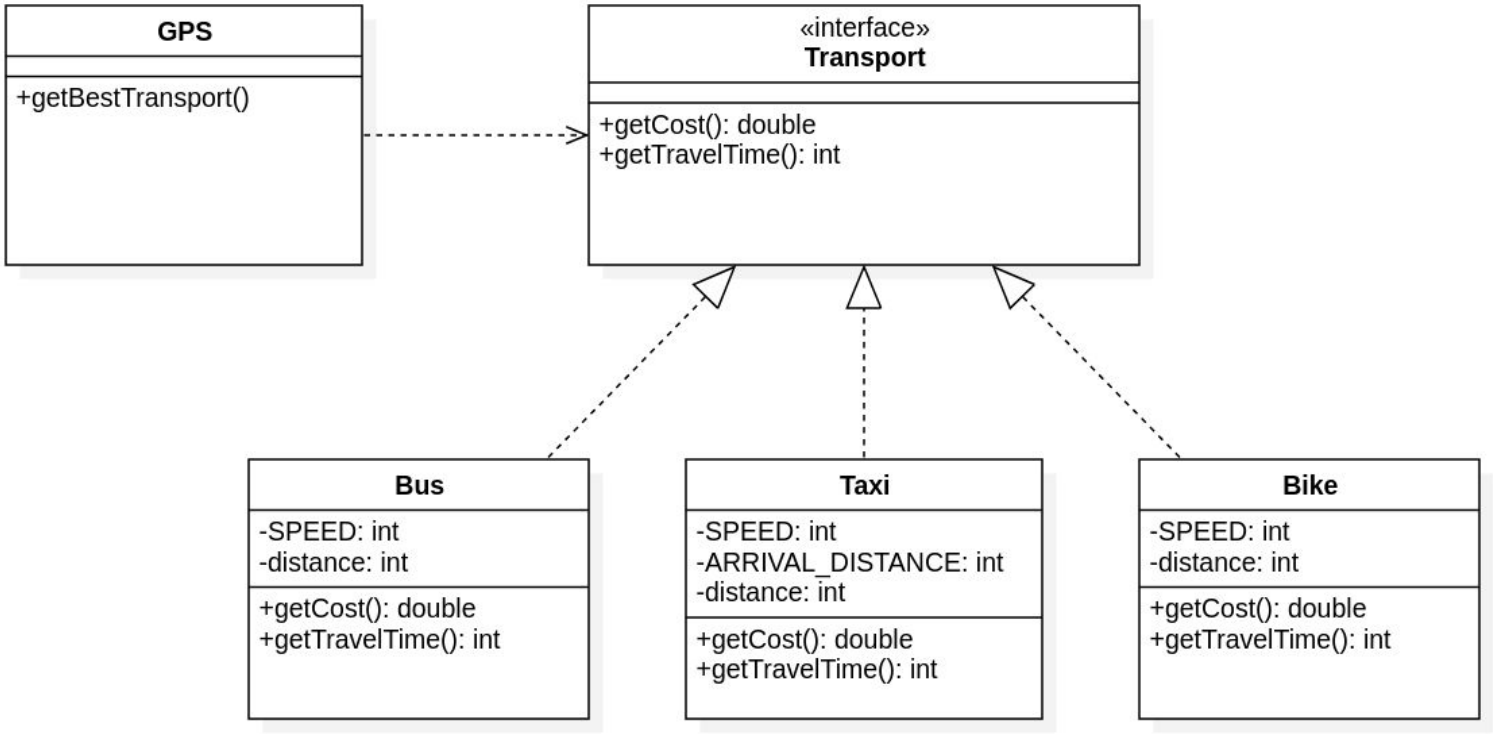
Problèmes de cet exemple

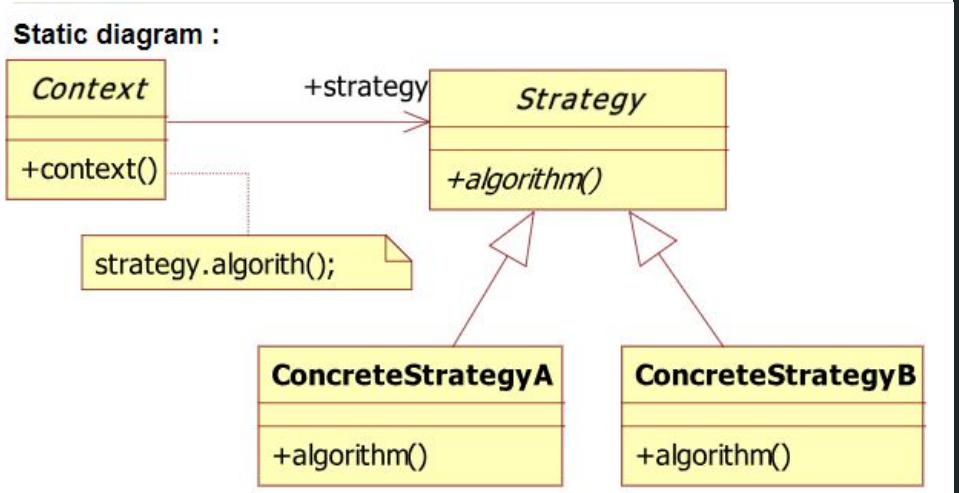
- Quel(s) moyen(s) Julien va-t-il utiliser pour aller à l'aéroport ?
- Sous quels critères va-t-il choisir son moyen de transport ?

Modélisations du cas de base par un diagramme UML



Solution par le pattern Strategy





Strategy, un pattern de comportement

Problématique de Strategy et son intention par ChatGPT

Problématique du Pattern Strategy

Dans certaines applications, il est nécessaire de **varier dynamiquement le comportement** d'un objet en fonction du contexte, sans pour autant alourdir ou complexifier la classe en utilisant des conditions multiples (`if-else` ou `switch-case`). Si les comportements varient souvent ou doivent être facilement extensibles, la gestion de ces variations peut rendre le code **difficile à maintenir, rigide et peu évolutif**.

Intention du Pattern Strategy

"Définir une famille de comportements interchangeables et encapsuler chaque comportement dans une classe séparée, afin que les objets puissent changer de comportement dynamiquement."

- **Séparer** les algorithmes dans des classes distinctes.
- **Interchanger** dynamiquement les algorithmes sans modifier les classes qui les utilisent.
- **Faciliter l'extension** en ajoutant de nouveaux comportements sans toucher au code existant.

Solutions du pattern

“Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:

- Clients that need linebreaking get more complex if they include the line-breaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.
- Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.
- It's difficult to add new algorithms and vary existing ones when linebreaking is an integral part of a client.

We can avoid these problems by defining classes that encapsulate different line-breaking algorithms. An algorithm that's encapsulated in this way is called a strategy.”

Détailler le rôle des classes participantes

Participants

Strategy (Compositor) :

- Déclare une interface commune à tous les algorithmes supportés. Le contexte utilise cette interface pour appeler l'algorithme défini par une stratégie concrète.

ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor) :

- Implémente l'algorithme en utilisant l'interface de la stratégie.

Context (Composition) :

- Est configuré avec un objet de type ConcreteStrategy.
- Maintient une référence à un objet de type Strategy.
- Peut définir une interface qui permet à la stratégie d'accéder à ses données.

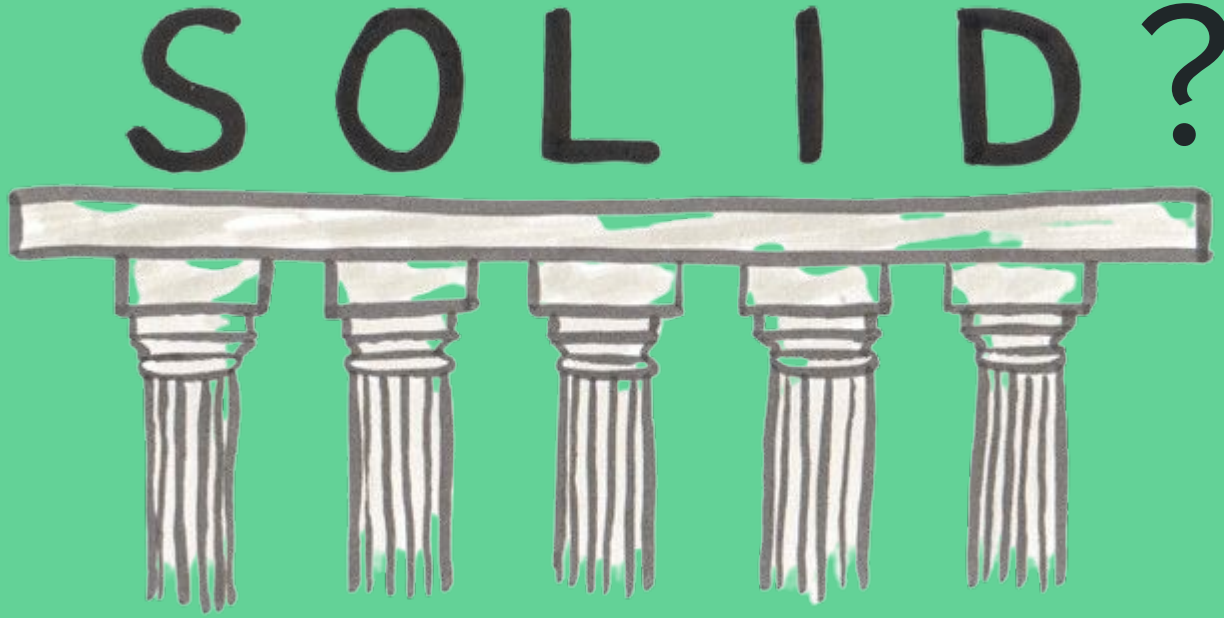
Quand doit-on utiliser Strategy ?

Le pattern Strategy simplifie une classe avec plusieurs responsabilités en déléguant les responsabilités à des classes distinctes, rendant le code plus clair et modulaire.

Les avantages du pattern Strategy incluent :

- Une responsabilité par classe,
- Ouverture à l'extension,
- Gestion des dépendances.
- Remplacer l’héritage par la composition
- Réduction des structures conditionnelles
- Rend extensible et modulable le code

Quels liens avec



Les différents principes liés à Strategy

- Single Responsability Principle (SRP)

Chaque stratégie à une responsabilité **UNIQUE**

- Open-Closed Principle (OCP)

Ce pattern permet d'implémenter des extensions **sans modifier** le programme en lui-même.

- Dependency Inversion Principle (DIP)

La classe principale dépend de classes abstraites de Strategy maintenant la **flexibilité** du code

Limites de Strategy

Inconvénients :

- Nécessité limitée : Dans le cas où il y a peu de changements de comportement, le pattern Strategy n’est pas la meilleure solution.
- Alourdissant : Dans un gros projet, le pattern Strategy n’est pas parfaitement efficace. La création des classes pour les différents comportements alourdit le code.
- Redondances : Ce pattern peut générer des redondances au sein d’un projet ce qui rejoint l’aspect alourdissant de Strategy.

Lien avec le pattern Etat

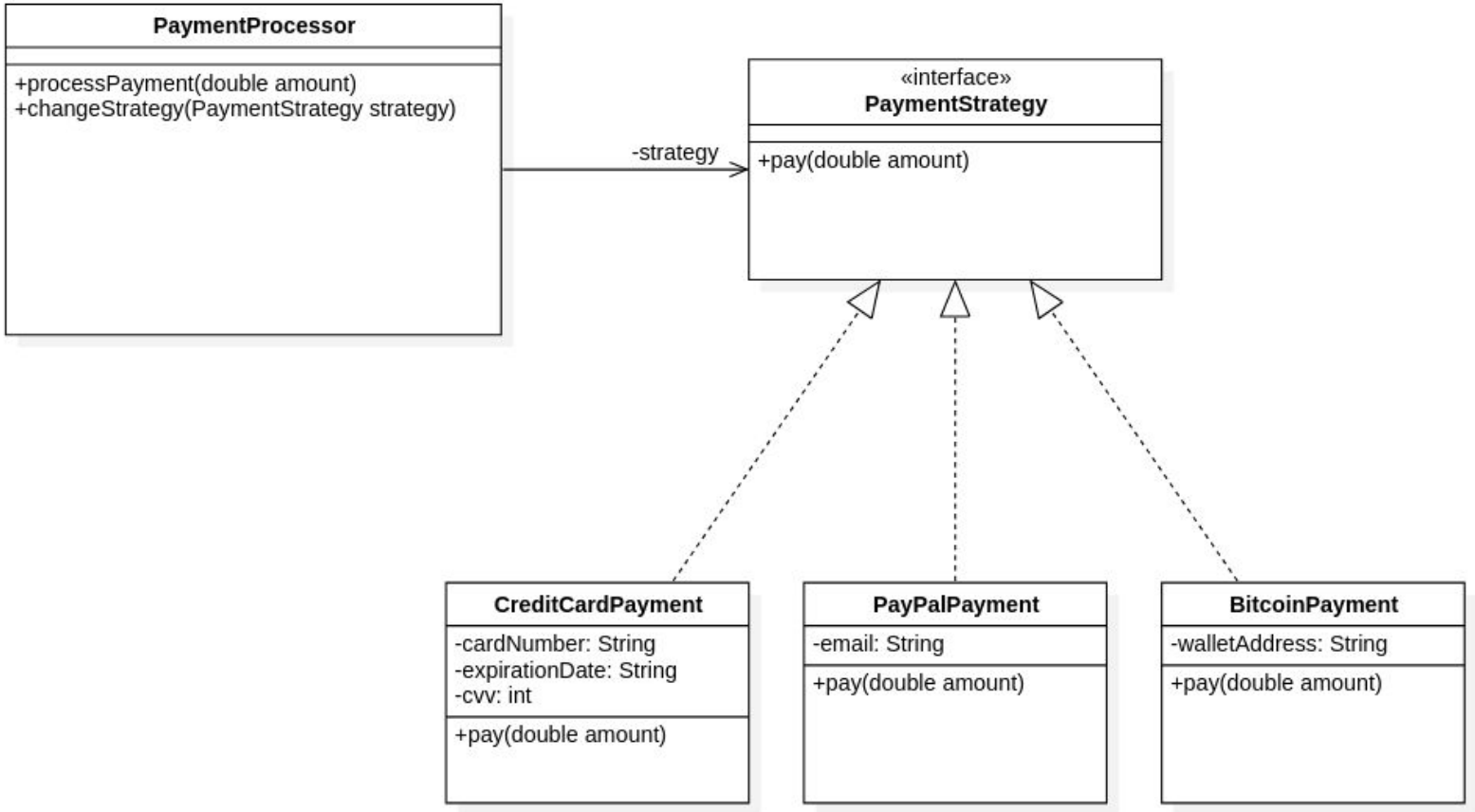
Les liens entre le pattern Strategy et le pattern Etat sont basés sur des concepts similaires, bien qu'ils aient des objectifs distincts. Voici les principaux liens entre les deux :

- Composition sur héritage
- Interchangeabilité des comportements
- Encapsulation de comportements spécifiques
- Utilisation d'une interface commune
- Réduction des structures conditionnelles

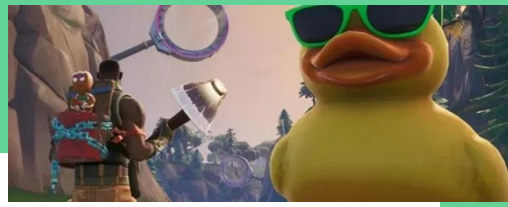
À suivre ...

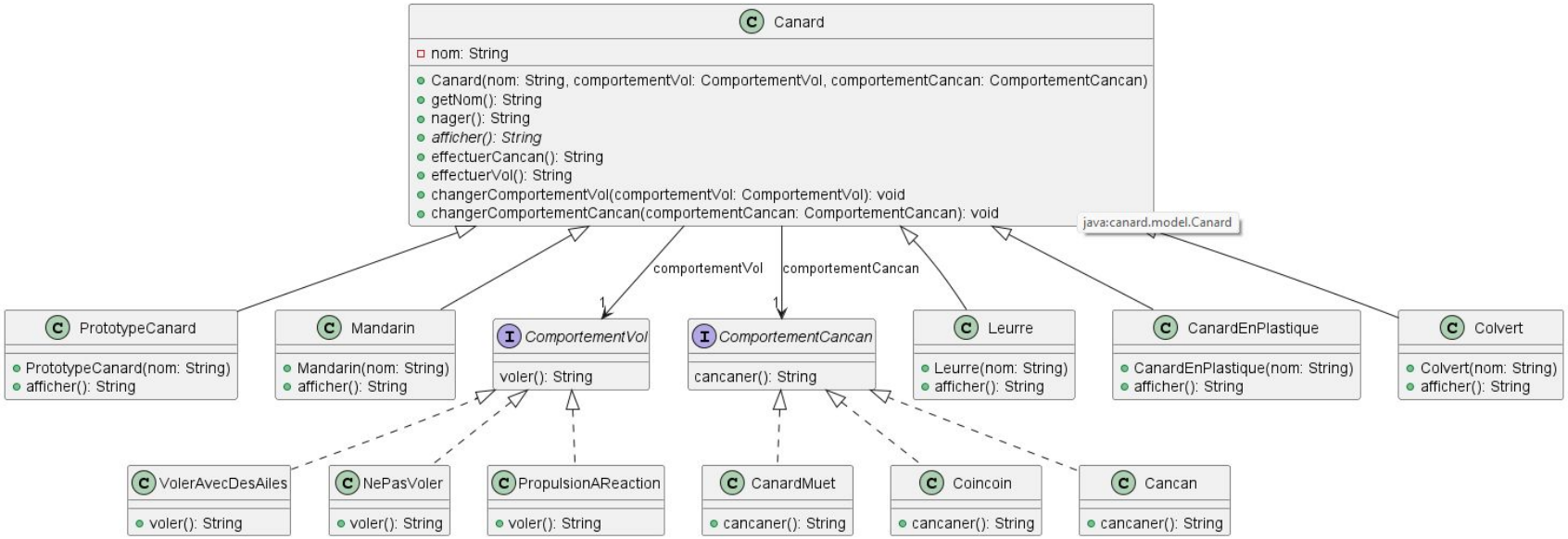
Live coding

https://youtu.be/AKBGRT1PBaA


PaymentProcessor
+processPayment(double amount)
+changeStrategy(PaymentStrategy strategy)
-strategy
«interface»
PaymentStrategy
+pay(double amount)
CreditCardPayment
-cardNumber: String
-expirationDate: String
-cvv: int
+pay(double amount)
PayPalPayment
-email: String
+pay(double amount)
BitcoinPayment
-walletAddress: String
+pay(double amount)

Des canards bien stratèges





Conclusion


WHEN YOU FINALLY UNDERSTAND THE
STRATEGY DESIGN
PATTERN AND FEEL LIKE A CODING GENIUS
Supermeme.ai

Bibliographie

https://fr.wikibooks.org/wiki/Patrons_de_conception/Stratégie

https://refactoring.guru/fr/design-patterns/strategy

https://java-design-patterns.com/patterns/strategy/

The Strategy Pattern Explained and Implemented in Java | Behavioral Design Patterns | Geekific

https://www.flaticon.com/fr/

https://chatgpt.com/