Design patterns

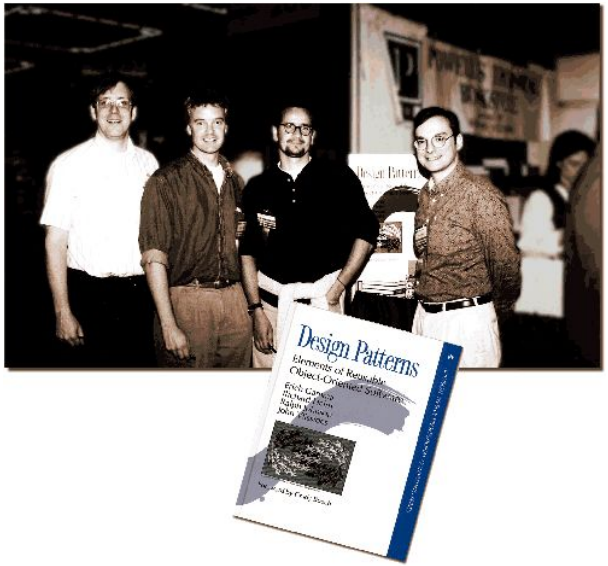
Introduction aux Patrons de Conception

Définition :

Les **patrons de conception** (ou **design patterns**) sont des solutions générales et réutilisables à des problèmes courants rencontrés dans la conception de logiciels. Ils ne sont pas des morceaux de code finis mais des descriptions ou des modèles de solution qui peuvent être appliqués dans différentes situations de programmation.

Un concept provenant de l'architecture

L'origine des patrons de conception remonte aux travaux de l'architecte Christopher Alexander dans les années 1970, qui a proposé des modèles récurrents dans la conception architecturale. Ces idées ont été adaptées à l'informatique dans les années 1990 par un groupe de quatre développeurs de logiciels, souvent appelés le **Gang of Four** (GoF) : Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. Leur livre, **"Design Patterns: Elements of Reusable Object-Oriented Software"** publié en 1994, a introduit 23 patrons de conception fondamentaux pour la programmation orientée objet.



Contexte Technique

Les patrons de conception sont nés de la nécessité de résoudre des problèmes récurrents de manière efficace dans la conception de logiciels. Ils sont particulièrement utiles dans le cadre de la programmation orientée objet, où les concepts d'héritage, de polymorphisme et d'encapsulation facilitent la mise en œuvre de solutions génériques et modulaires.

Le catalogue des design pattern

Ils sont généralement classé en trois catégories :

- Les **Patrons de création** fournissent des mécanismes de création d'objets, ce qui augmente la flexibilité et la réutilisation du code.
- Les **Patrons structurels** expliquent comment assembler des objets et des classes en de plus grandes structures, tout en les gardant flexibles et efficaces.
- Les **Patrons comportementaux** mettent en place une communication efficace et répartissent les responsabilités entre les objets.

Définitions extraites du site de référence : refactoring.guru

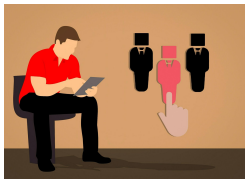
Structure d’un patron de conception

Les patrons de conception est décrit d’une manière assez normalisé, afin de pouvoir l’identifier et en comprendre le fonctionnement et les cas d’usage . Voici les éléments que l’on retrouve généralement dans leur description :

- Son **Nom** afin de pouvoir l’identifier.
- L’**Intention** du patron permet de décrire brièvement le problème et la solution.
- La **Motivation** explique en détail la problématique et la solution offerte par le patron.
- La **Structure** des classes montre les différentes parties du patron et leurs relations.
- Les **Conséquences** sont les résultats et les compromis de l'application du patron. Cela inclut les avantages, les inconvénients, et les effets potentiels sur la flexibilité, la réutilisabilité et la performance du système.
- L’**Exemple de code** écrit dans un des langages de programmation les plus populaires facilite la compréhension générale de l’idée derrière le patron.

Application d'un Patron de Conception

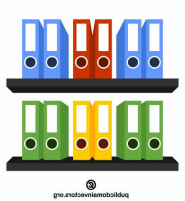
Identification du problème



Sélection du patron adapté

Adaptation au contexte

Evaluation



Documentation

Les patrons de création : gérer des instances de classes

Nous étudierons :

- Singleton
- Factory
- Abstract factory

Singleton

Intention

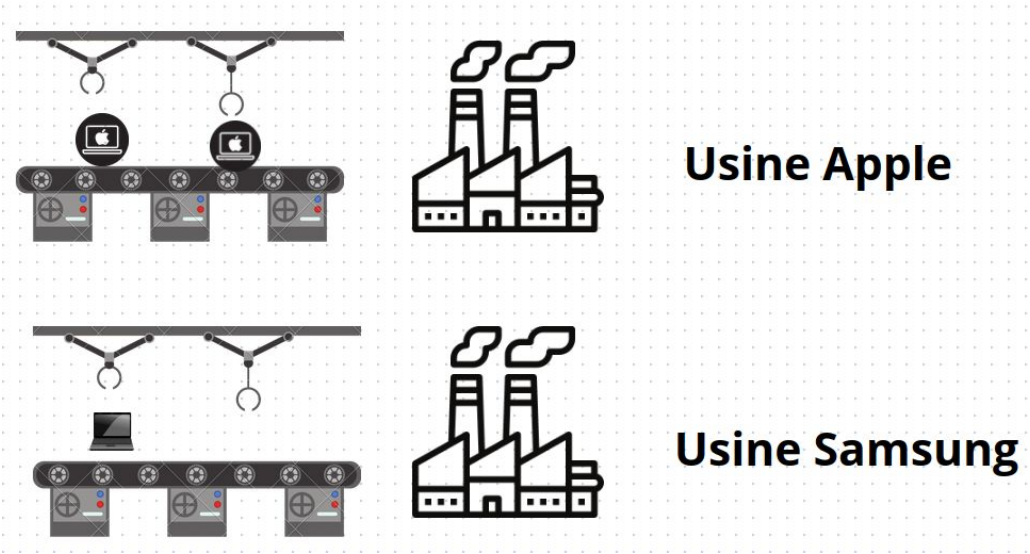
Garantir qu'une classe n'ait qu'une seule instance et fournir un point d'accès global à cette instance.

Motivation

Dans certains scénarios, il est crucial de s'assurer qu'une classe ait une seule instance. Par exemple, un programme peut avoir besoin d'un seul point d'accès pour une base de données ou une configuration globale. Le pattern Singleton garantit qu'il n'y ait qu'une seule instance de la classe en question et fournit un point d'accès global à cette instance.

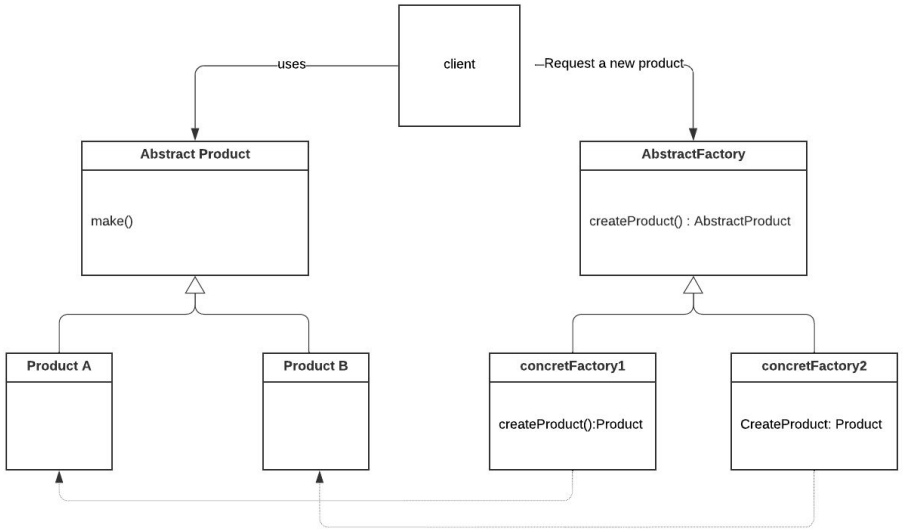
```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = object.__new__(cls)
        return cls._instance

    def __init__(self, value):
        if not hasattr(self, 'initialized'):
            self.value = value
            self.initialized = True
```

Factory method

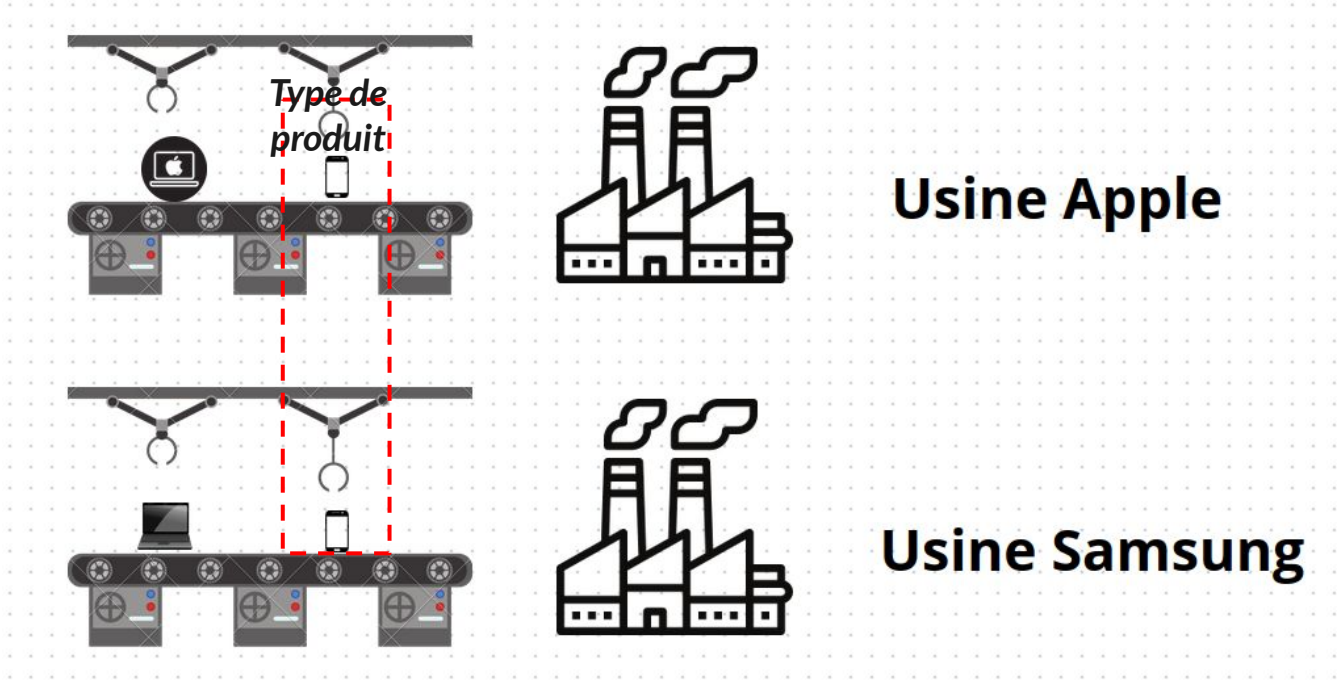


Factory method

Définir une interface pour créer des objets tout en gardant un contrôle sur le choix de la classe à instancier.

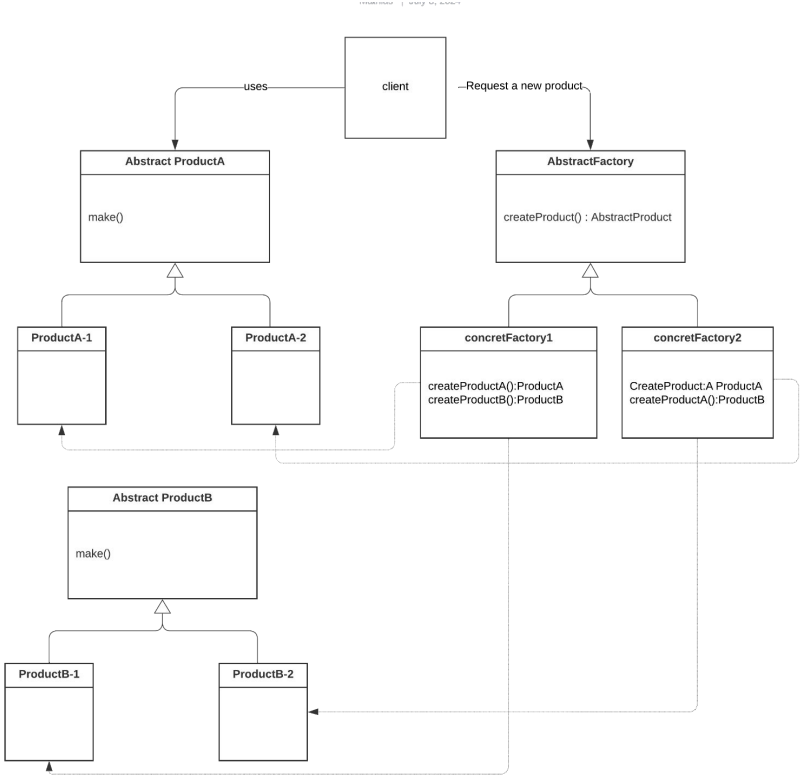


Abstract Factory



Abstract factory

- Créer des familles d'objets ayant un lien ou étant interdépendants.
- Obtenir plusieurs types d'objets retournés par chaque fabrique.



Les patrons structurels : structurer des données

Nous étudierons :

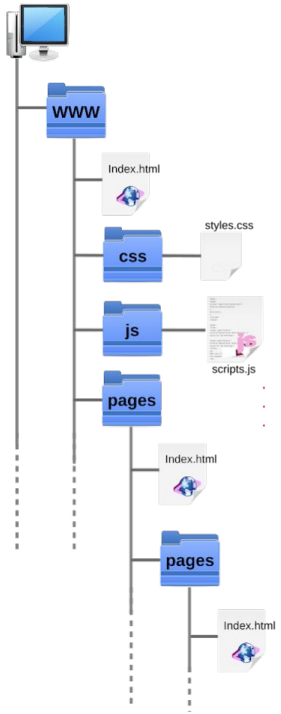
- Composite
- Proxy
- Adaptateur
- Façade

Composite

Objectif :

- Organise les objets en structure arborescente afin de représenter une hiérarchie.
- Permettre à la partie cliente de manipuler un objet unique et un objet composé de la même manière.

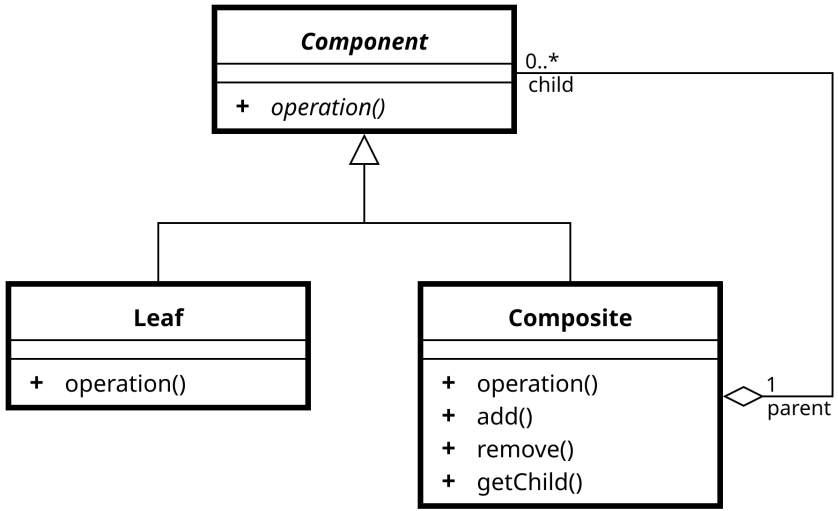
Exemple : Système de gestion de fichiers/dossiers au sein d'une arborescence



Composite : diagramme

- Component (abstrait)
- Leaf (élément terminal) et composite sont des dérivés de component .
- Composite se compose d'un ensemble de components (qui peuvent être des Leaf ou des composite).

Exemple : un dossier peut contenir plusieurs fichier et dossiers qui peuvent eux même être composés de dossiers et de fichiers etc.



Proxy

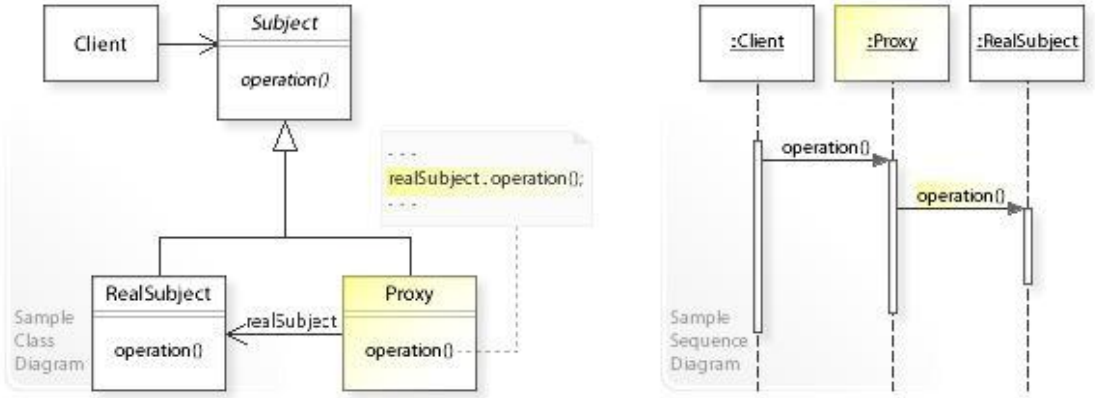
Objectif :

- Fournir un intermédiaire entre la partie cliente et un objet pour contrôler les accès à ce dernier.

Résultat :

- Le design pattern permet d’isoler le comportement lors de l’accès à un objet.

Proxy



- Le proxy implémente la même interface que l'objet cible.
- Le client appelle la méthode opération du proxy qui va elle même appeler la méthode opération de l'objet cible. Au préalable le proxy pourra effectuer éventuellement des vérification (cache, sécurité etc.)
- Usage : transaction(BDD)

Adapter

Objectif :

- Convertir l'interface d'une classe dans une autre interface comprise par la partie cliente.
- Permettre à des classes de fonctionner ensemble, ce qui n'aurait pas été possible à cause de leurs interfaces incompatibles.

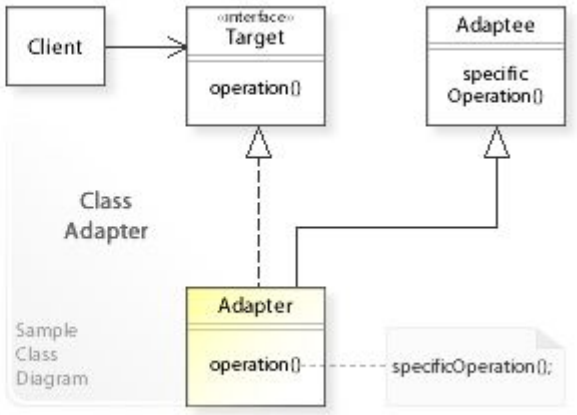
Résultat :

- Le design pattern permet d'isoler l'adaptation d'un sous-système.

Adapter, implémentation par l'héritage (classe)

Adapter par l'héritage de classe :

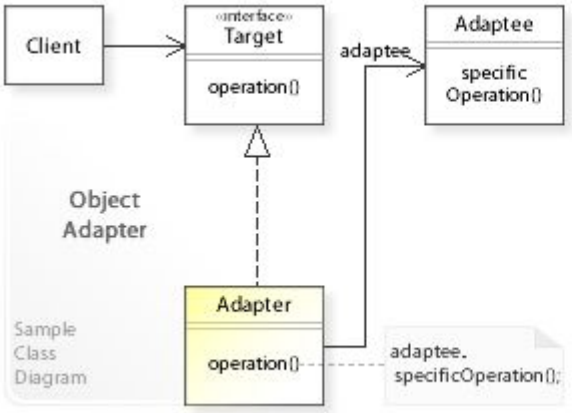
- Utilise la relation "est un" (is-a). L'Adapter hérite des deux interfaces Target et Adaptee.
- Peut être plus simple à implémenter lorsque les interfaces sont relativement simples.
- Peut entraîner une hiérarchie complexe si de nombreuses classes doivent être adaptées.



Adapter, implémentation par la composition (objet)

Composition:

- Utilise la relation “a un” (has-a). L’Adapter contient une instance de Adaptee.
- Offre une plus grande flexibilité car l’Adapter peut être modifié sans affecter les classes Target et Adaptee.
- Privilégié en général en raison de sa flexibilité et de la réduction de l’héritage multiple.



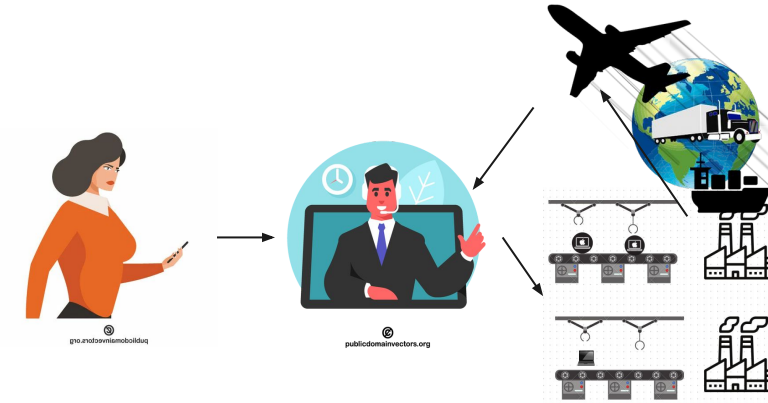
Façade

Objectif :

- Fournir un intermédiaire entre la partie cliente et un sous-système, souvent complexe, pour contrôler les accès à ce dernier. Elle en simplifie l'accès en ne proposant, par exemple, que les fonctionnalités intéressant le client.

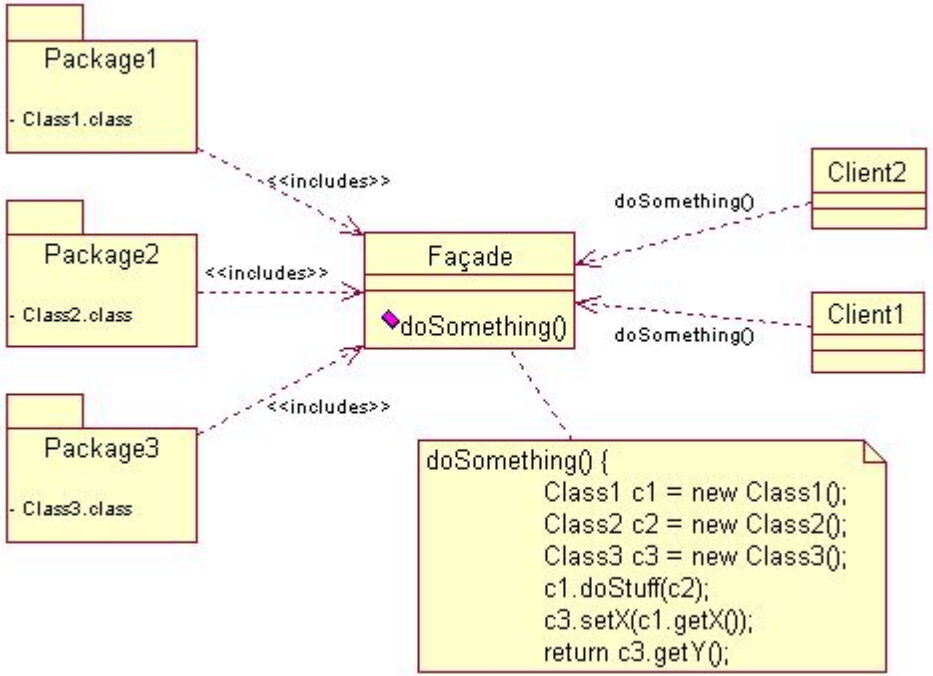
Résultat :

- Le design pattern permet d'isoler le comportement lors de l'accès à un objet.



Façade

La méthode doSomething de la façade, mise à la disposition des clients, prendra en charge une série d'opération complexe.



Patrons comportementaux : maîtrise du comportement

Les patrons comportementaux s'occupent des algorithmes et de la répartition des responsabilités entre les objets.

- Strategy
- Itérateur
- Observer
- Chaîne de responsabilité
- Visiteur

Strategy

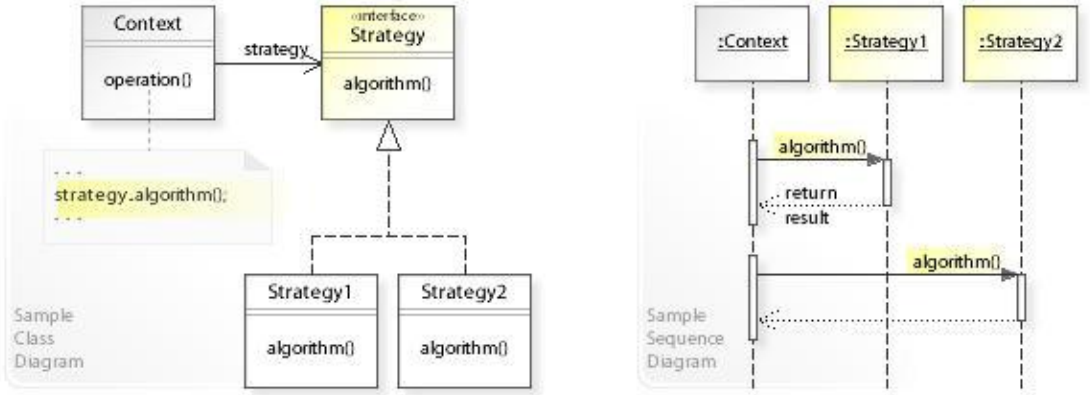
Objectifs :

Définir une famille d'algorithmes, les encapsuler dans de classe séparées et de les rendre interchangeable.

Résultat:

Le design pattern permet d’isoler des algorithmes appartenant à une même famille d’algorithme.

Strategy

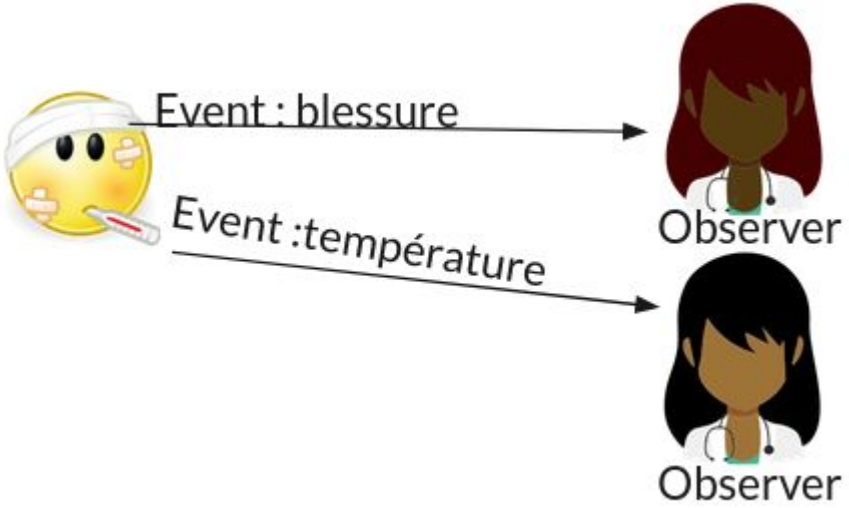


- Chaque strategy est une implémentation de l'interface Strategy.
- Le client est lié à l'interface Strategy, à un certain moment nous allons pouvoir lui affecter un objet Strategy1 et à un autre moment un objet Strategy2.

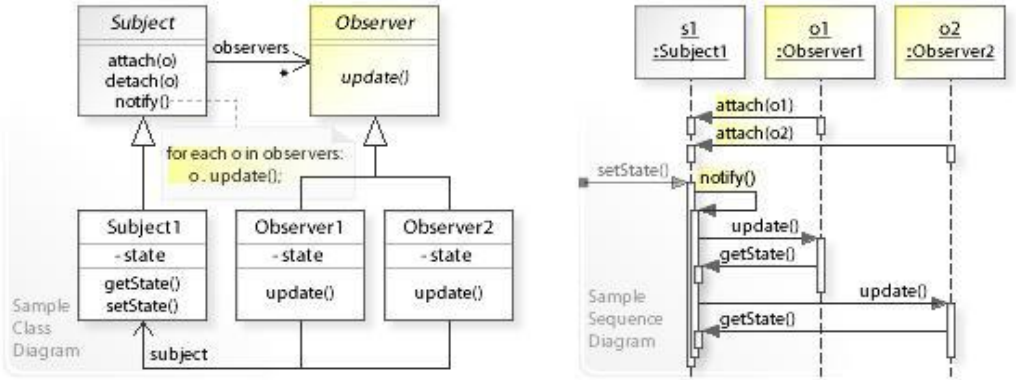
Observer

Objectif :

Mettre en place un mécanisme de souscription pour envoyer des notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent.



Observer



2 possibilités :

- **Push** : lorsque son état change l'observable (subject) envoi l'information à ses observateurs.
- **Pull** : lorsque l'état de l'observable change, il prévient ses observateurs, à charge pour eux de récupérer l'information (getState()).

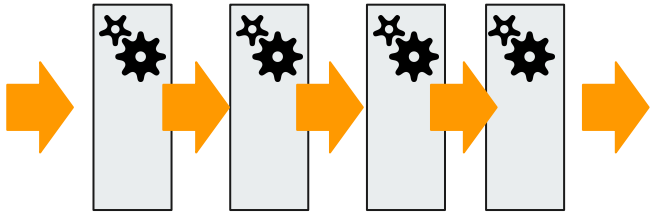
L'observable dispose généralement de 3 méthodes : **attach()** pour ajouter un observateur, **detach()** pour supprimer un observateur, et **notify()** pour informer les observers de son changement d'état.

Exemple : event en javascript

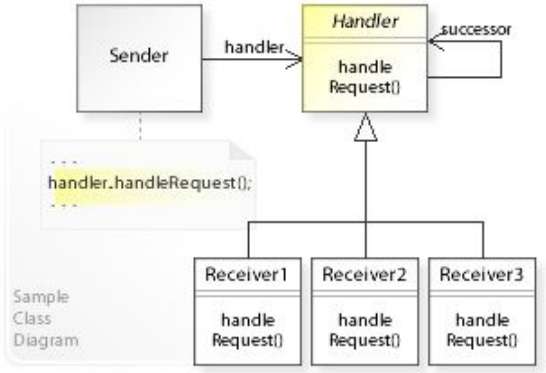
Chaîne de responsabilité

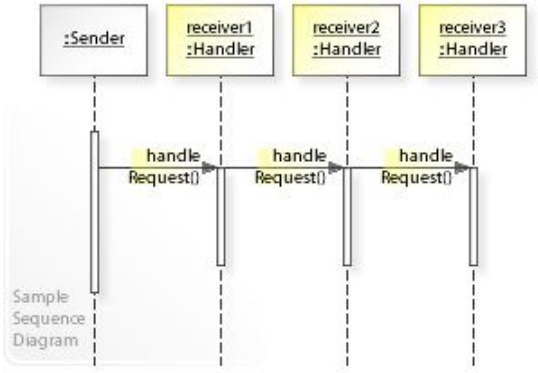
Objectif :

Faire circuler des demande d'un objet gestionnaire (handler) à un autre. Lorsqu'un gestionnaire reçoit une demande, il décide de la traiter ou pas et de l'envoyer au gestionnaire suivant.



Chaîne de responsabilité

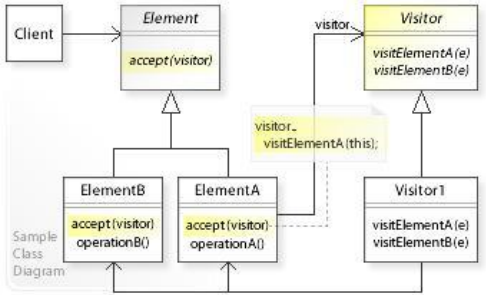


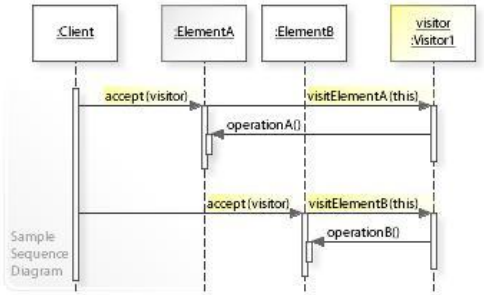


- Handler: une interface ou une classe abstraite qui déclare une méthode pour traiter les demandes et une méthode pour définir le prochain gestionnaire dans la chaîne.
- ConcreteHandler (ici receiver1, reveiver2 et receiver3): des classes concrètes qui implémentent les méthodes pour traiter les demandes. Elles appellent le prochain gestionnaire si elles ne peuvent pas traiter la demande.
- Sender: envoie les demandes à la chaîne de gestionnaires.

autres design patterns (à terminer)

Visiteur


Client
Element
accept(visitor)
visitor
Visitor
visitElementA(e)
visitElementB(e)
visitor.
visitElementA(this);
ElementB
accept(visitor)
operationB()
ElementA
accept(visitor)
operationA()
Visitor1
visitElementA(e)
visitElementB(e)
Sample
Class
Diagram


:Client
:ElementA
:ElementB
visitor
:Visitor1
accept(visitor)
visitElementA(this)
operationA()
accept(visitor)
visitElementB(this)
operationB()
Sample
Sequence
Diagram

Itérateur