

Patron de conception => la meilleure solution connue à un problème de conception récurrent

→ Par :Erich Gamma. Richard Helm. Ralph Johnson et John Visside

Patron d'architecture => agencement des packages les uns par rapport aux autres

Patron de conception => agencement des classes les unes par rapport aux autres

Structurelles => Agence les classes : résout le problème en agençant les classes

Comportementaux => structurelle + ajoute comportement aux classes : agencement des classes + ordonnancement des appels de méthode

Créateurs => gère la création des instances d'objets

SOLID :

S => Single Responsibility principle : une classe doit avoir une seule et unique responsabilité, donc elle qu'une seule raison de changer → faciliter la maintenance et l'évolution.

O => Open closed principle : ouverte à l'extension, fermé à la modification → Maintenir une bonne encapsulation et éviter les régressions

L => Liskov substitution principle : toute classe mère est substituable une de ses classes filles
→ rend le code plus robuste, évolutif et maintenable

I => Interface ségrégation principle : 1 interface = 1 fonctionnalité

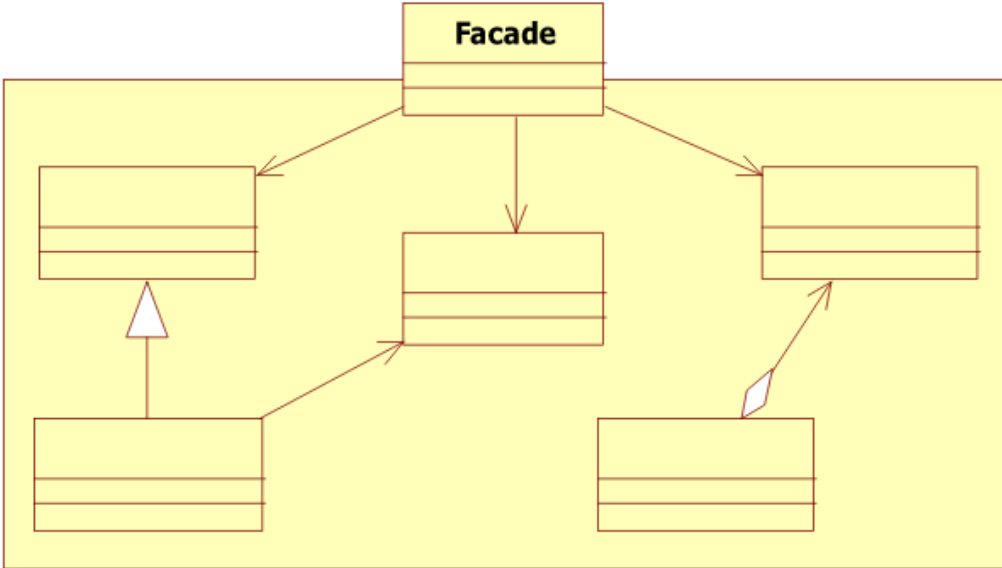
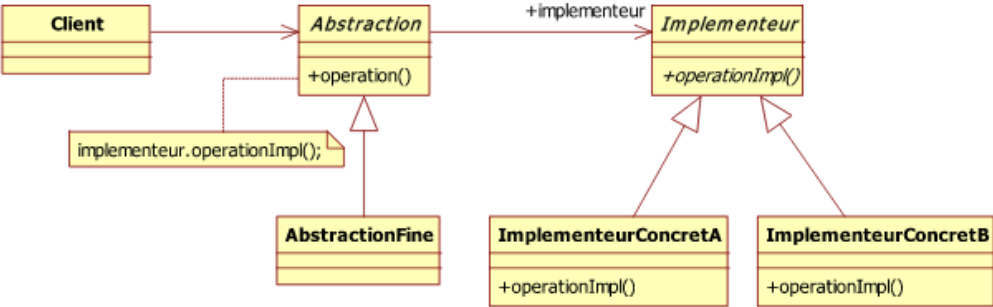
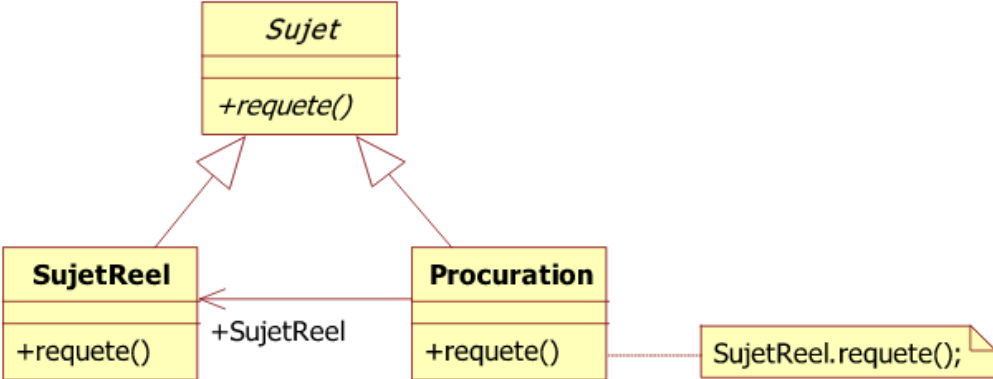
D => Dependency inversion principle : chaque classe doit pointer vers l'abstraction de son type de base
→ les classes doivent dépendre d'une interface ou d'une classe abstraite

Interface fonctionnelles → Interface ne possèdent qu'une méthode (pas besoin de faire une nouvelle classe fille pour l'utiliser)

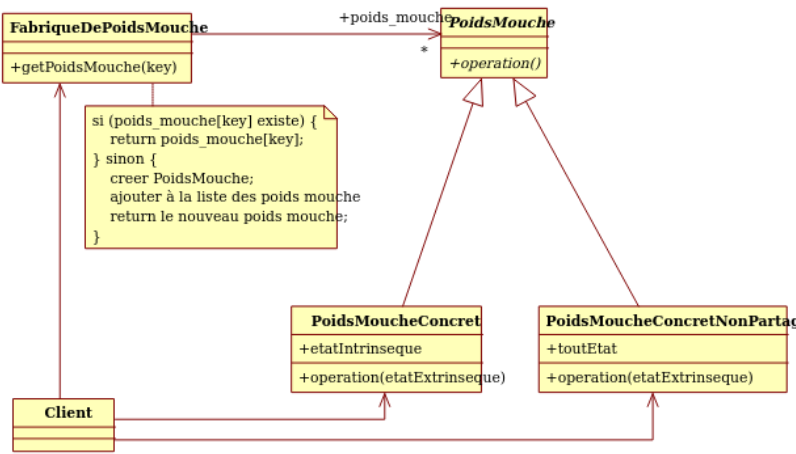
Binding → utilise les observateurs

Thread → Un processus qui lance des flux d'exécution (pile avec tas partagé)

Nom	structure	but
Adaptateur	<pre>classDiagram class Client class Target { +request() } class Adapter { +request() } class Adaptee { +specificRequest() } Client --> Target Adapter -- > Target Adapter --> Adaptee : +adaptee note for Adapter "+request() implementation: return adaptee.specificRequest();" style Client fill:#ffffcc style Target fill:#ffffcc style Adapter fill:#ffffcc style Adaptee fill:#ffffcc</pre>	Comment adapter le protocole d'une méthode à une autre
Composite	<pre>classDiagram class Client class Composant { +opération() +ajouter(x: Composant) +retirer(x: Composant) +getEnfant(x: int) } class Feuille { +opération() } class Composite { +opération() +ajouter(x: Composant) +retirer(x: Composant) +getEnfant(x: int) } Client --> Composant Feuille -- > Composant Composite -- > Composant Composite --> Composant : * +enfants note for Composite "Note : structurellement identique à l'Interprète"</pre>	Comment construire des arborescences, des structures hiérarchique et des emboîtements d'objet Note : structurellement identique à l'Interprète
Décorateur	<pre>classDiagram class Composant { +operation() } class ComposantConcret { +operation() } class Decorateur { +operation() } class DecorateurConcret_A { +etat +operation() } class DecorateurConcret_B { +operation() +ajouterComportement() } ComposantConcret -- > Composant Decorateur -- > Composant DecorateurConcret_A -- > Decorateur DecorateurConcret_B -- > Decorateur Decorateur --> Composant : +composant note for Decorateur "+operation() implementation: composant.operation();" note for DecorateurConcret_B "+operation() implementation: super.operation(); ajouterComportement();" style Composant fill:#ffffcc style ComposantConcret fill:#ffffcc style Decorateur fill:#ffffcc style DecorateurConcret_A fill:#ffffcc style DecorateurConcret_B fill:#ffffcc</pre>	Comment ajouter des fonctionnalités inconnues à un objet

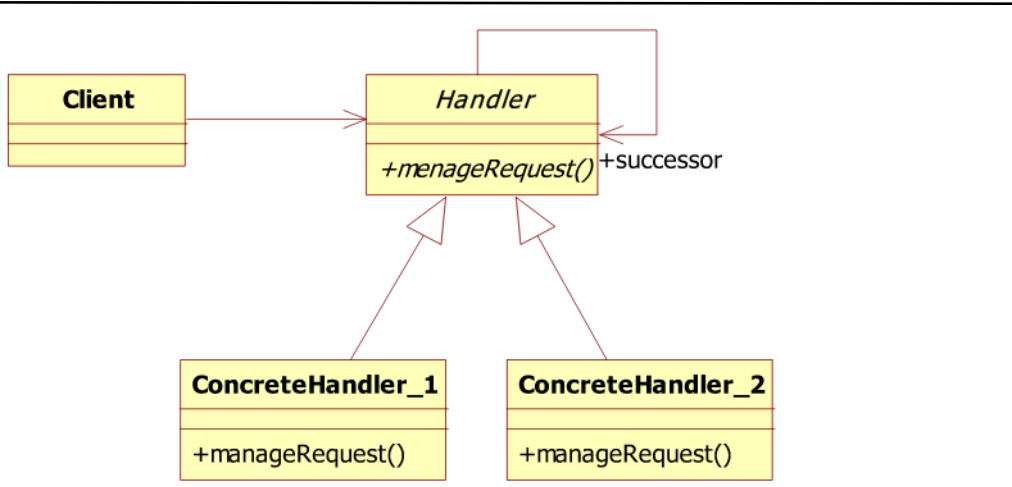
<p>Façade</p>		<p>Comment faciliter l'exécution d'un système et l'accès à un sous système</p>
<p>pont</p>		<p>Comment séparer 2 concepts liés de manière de le faire évoluer indépendamment l'un des autres.</p> <p>Découple une abstraction de son implémentation afin que les 2 éléments puissent être modifiés indépendamment l'un de l'autre</p>
<p>procuration</p>		<p>Comment faire passer un objet par un autre</p>

poids-mouche



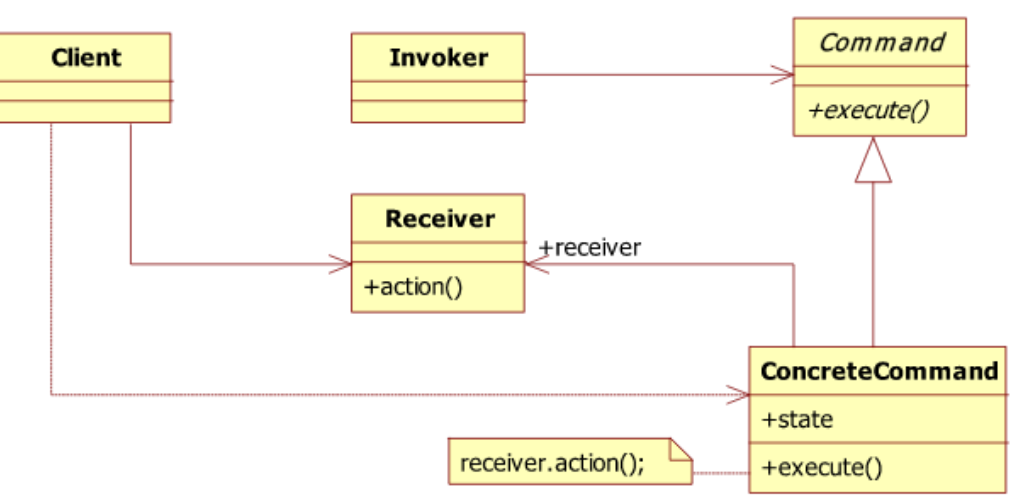
Comment limiter le nombre d'objets en mémoire

Chaîne de responsabilité



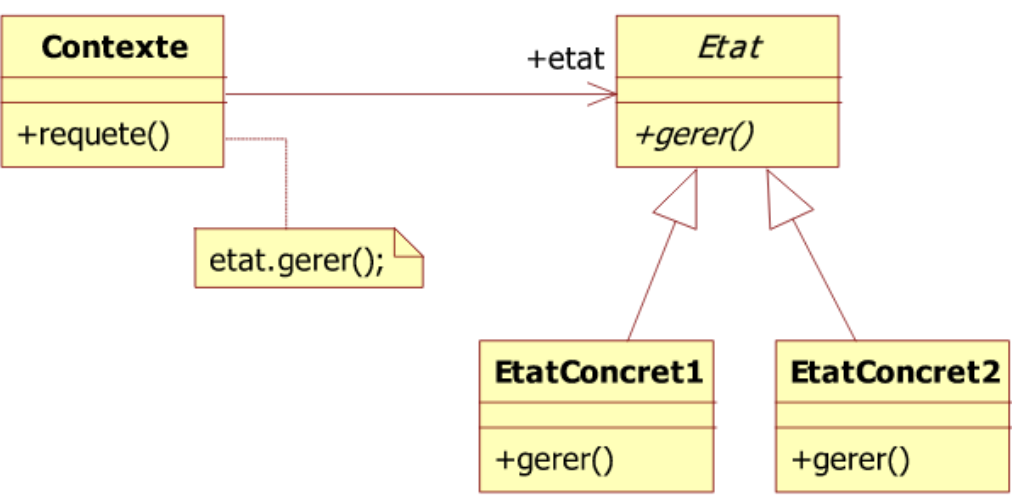
Comment masquer à l'utilisateur la résolution d'un problème

Commande



Comment mémoriser un appel de méthode continue

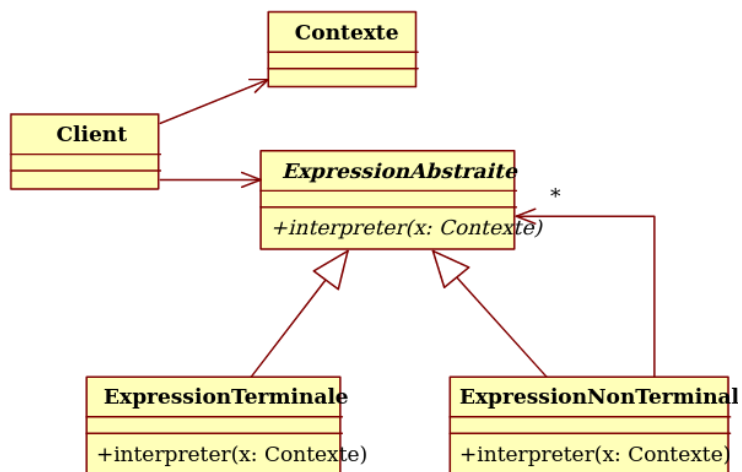
Etat



Comment changer le comportement d'un objet en fonction de son état (= changer l'état d'un objet à l'exécution)

Note : structurellement identique à Stratégie

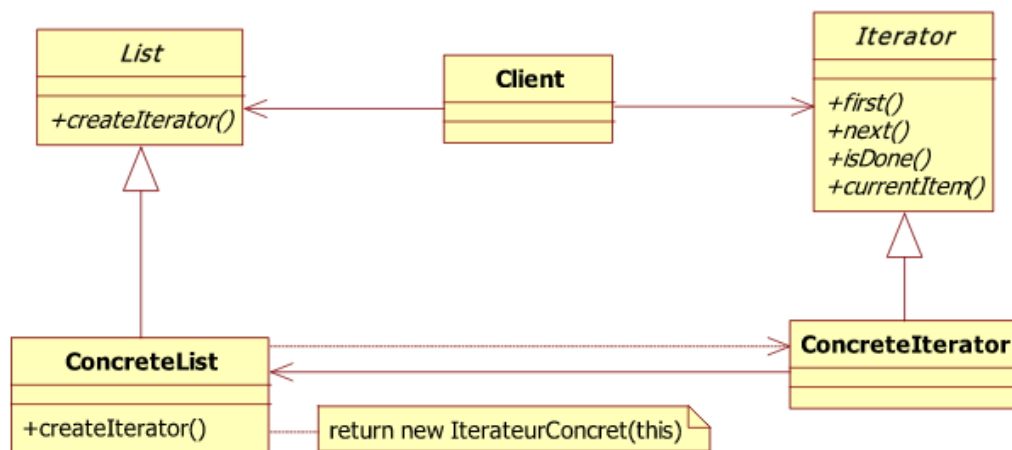
interprète



Comment interpréter un langage

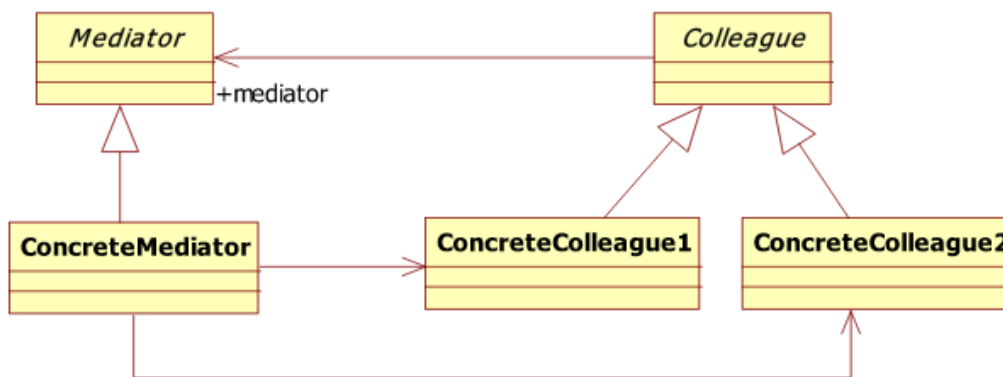
Note : structurellement identique au Composite

Itérateur



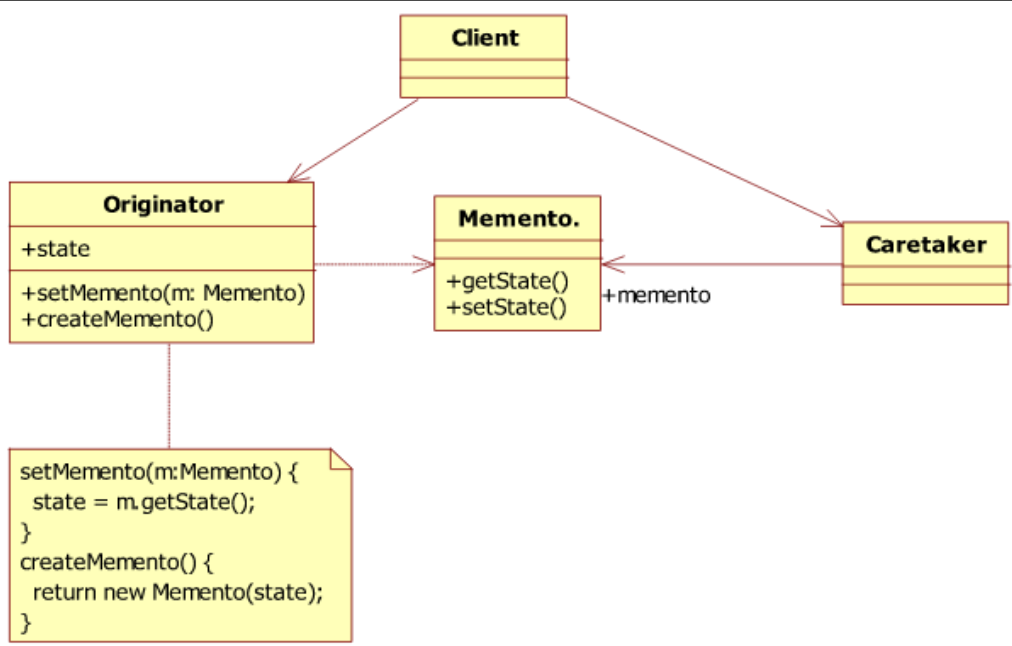
Comment faire pour contrôler l'itération d'une collection sans que la collection n'ait à la gérer elle-même.

Médiateur



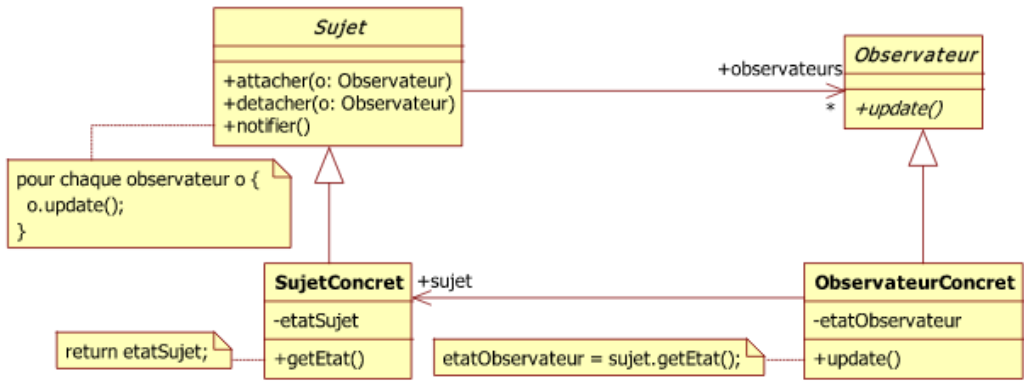
Comment réduire le couplage entre les objets

Memento



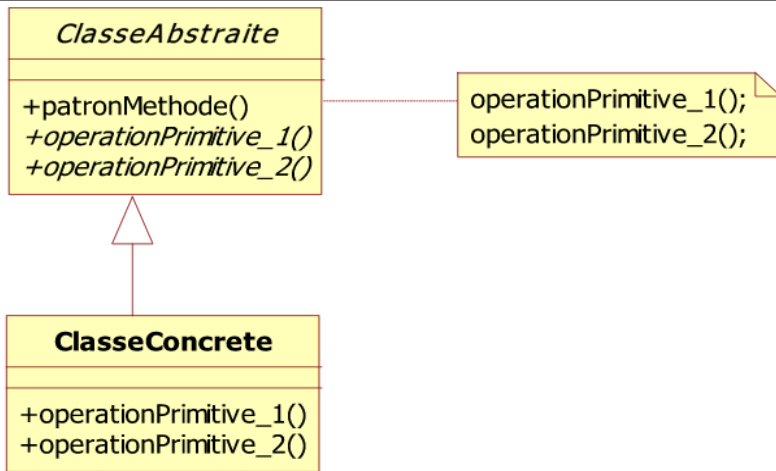
Comment mémoriser l'état d'un objet

Observateur



Comment observer le changement d'état d'un objet

patron de méthode

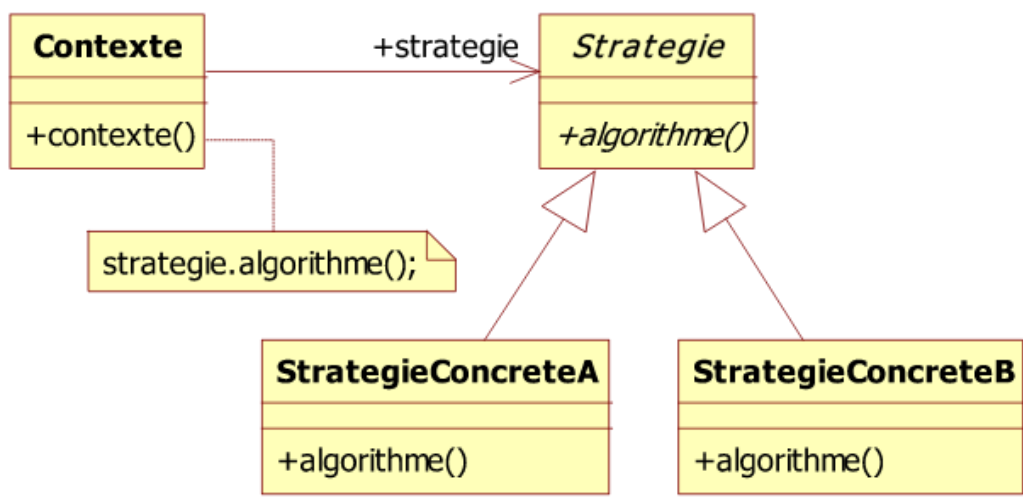


Comment abstraire le comportement d'une méthode

Comment spécialiser à l'exécution une partie de l'algorithme d'une méthode

ATTENTION : « patron » fait parti de son nom

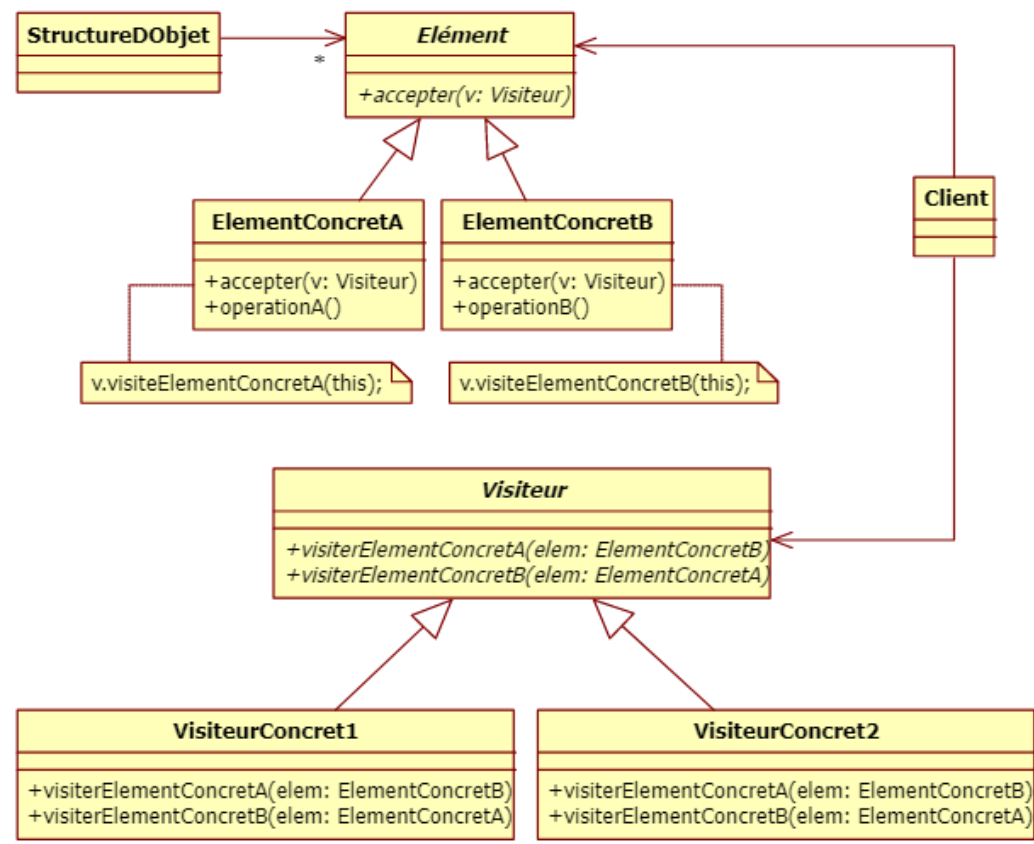
Stratégie



Comment changer dynamiquement l'algo d'une méthode (à l'exécution)

Note : structurellement identique à Etat

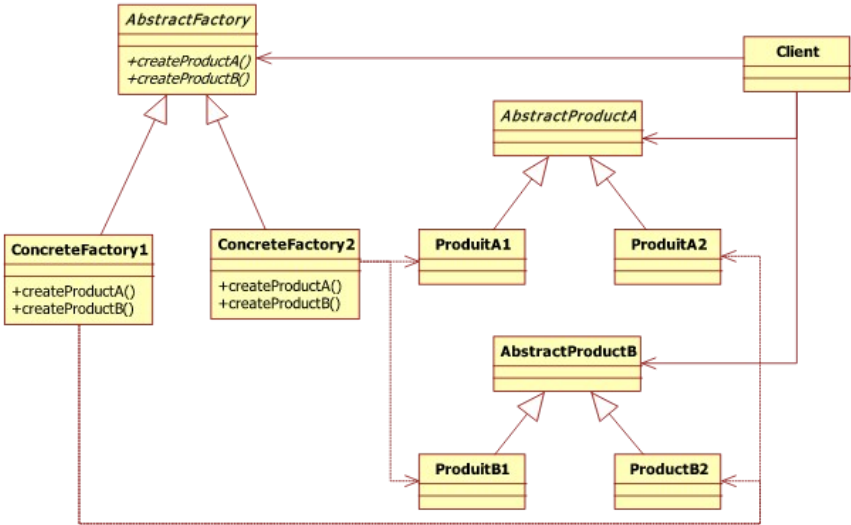
Visiteur



Comment faire faire à une classe quelque chose qu'elle ne savait pas comment faire

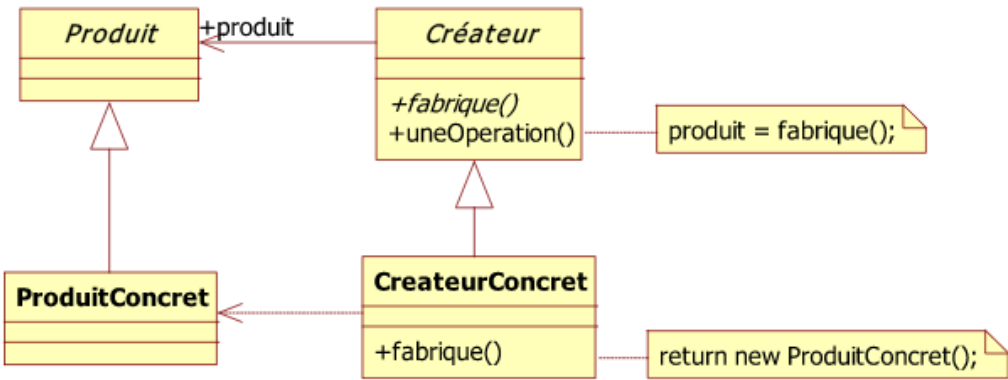
Note : enfreint le principe O

Fabrique abstraite



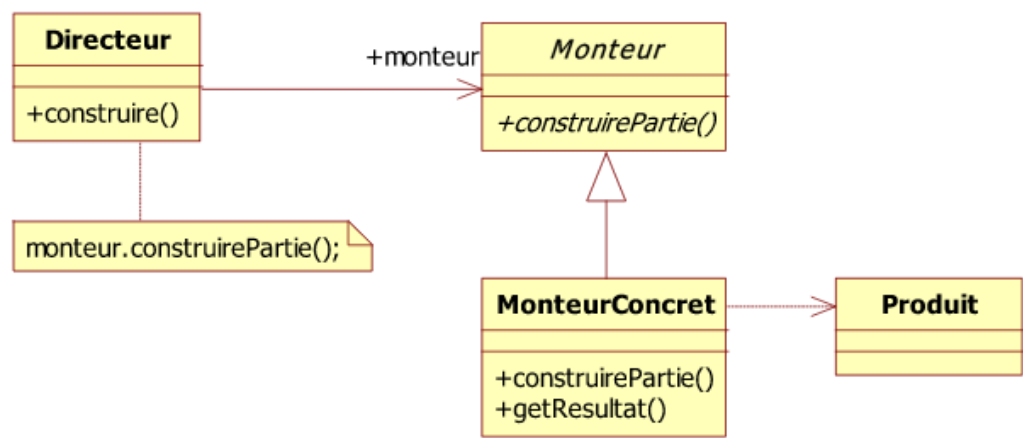
Comment s'assurer que l'environnement d'un objet est créé avec lui

Fabrique simple



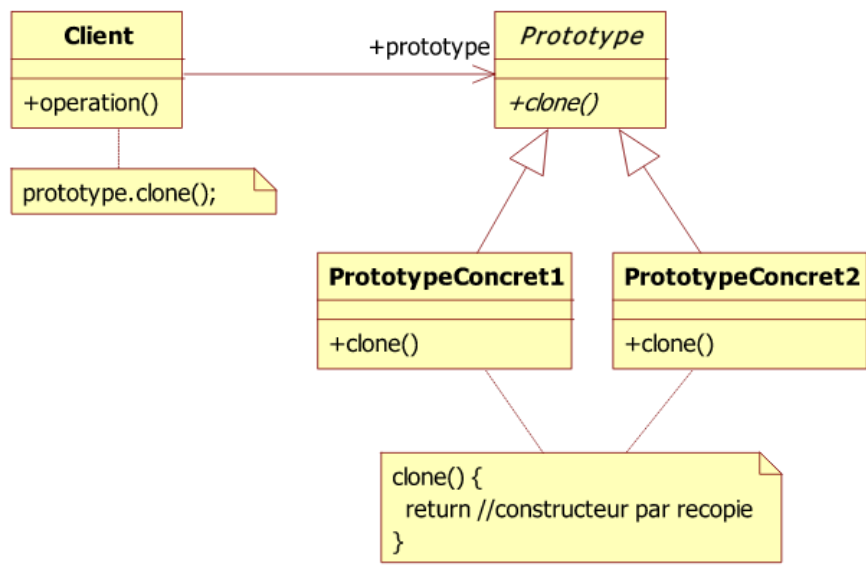
Comment contrôler la création d'un objet

Monteur



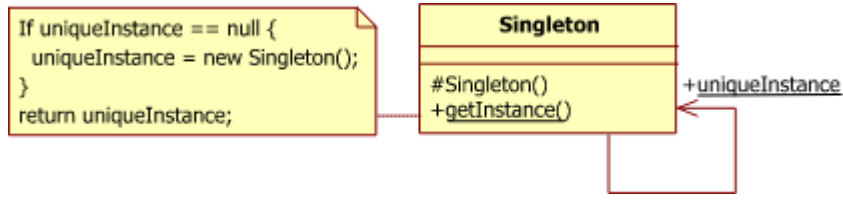
Comment dissocier la construction d'un objet de sa représentation

prototype



Comment cloner un objet depuis l'extérieur

Singleton



Comment garantir l'instanciation unique d'une classe