Représentation des patterns avec UML

Pierre-Alain Muller

ENSISA

pa.muller@uha.fr

03.89.33.69.65

Les micro-architectures (patterns)

La réutilisation du savoir et du savoir-faire

Définition des patterns

- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” C. Alexander

Description des patterns

- Le nom (augmenter le niveau d'abstraction)
- Le problème (contexte, pre/post conditions)
- La solution (indépendante de la réalisation)
- Les conséquences (flexibilité, extensibilité, portabilité)

Patterns et analyse

- Formalisation de savoir
- Représenter le domaine d’un métier
- Réutilisation d’expertise

Patterns et conception

- Formalisation de savoir faire
- Augmenter le niveau d’abstraction
- Eviter les objets trop liés au monde réel
- Rechercher des conceptions de plus haut niveau, éventuellement moins performantes, mais plus flexibles

Design Patterns

- Elements of Reusable Object-Oriented Software
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Addison-Wesley 1995

Idiomes

- Les idiomes (particularités liées à un langage)
- Combinaisons de caractéristiques existantes afin d'en définir de nouvelle
- Une technique donnée peut-être un idiome dans un langage, et une construction syntaxique dans un autre

Exemple d’idiomes

- Copy d’une chaîne de caractères
  - `While (*the_copy++ = *original++)`
- Préfixage d’une unité de compilation
  - `Standard.Mailbox.Open`

C++ et les idiomes

- C++ est un langage de bas niveau : beaucoup de techniques de programmation relèvent des idiomes
  - Gestion mémoire, entrées-sorties, initialisation, ...
- C++ est plus riche en méta-techniques que en techniques de base

Construire des patterns

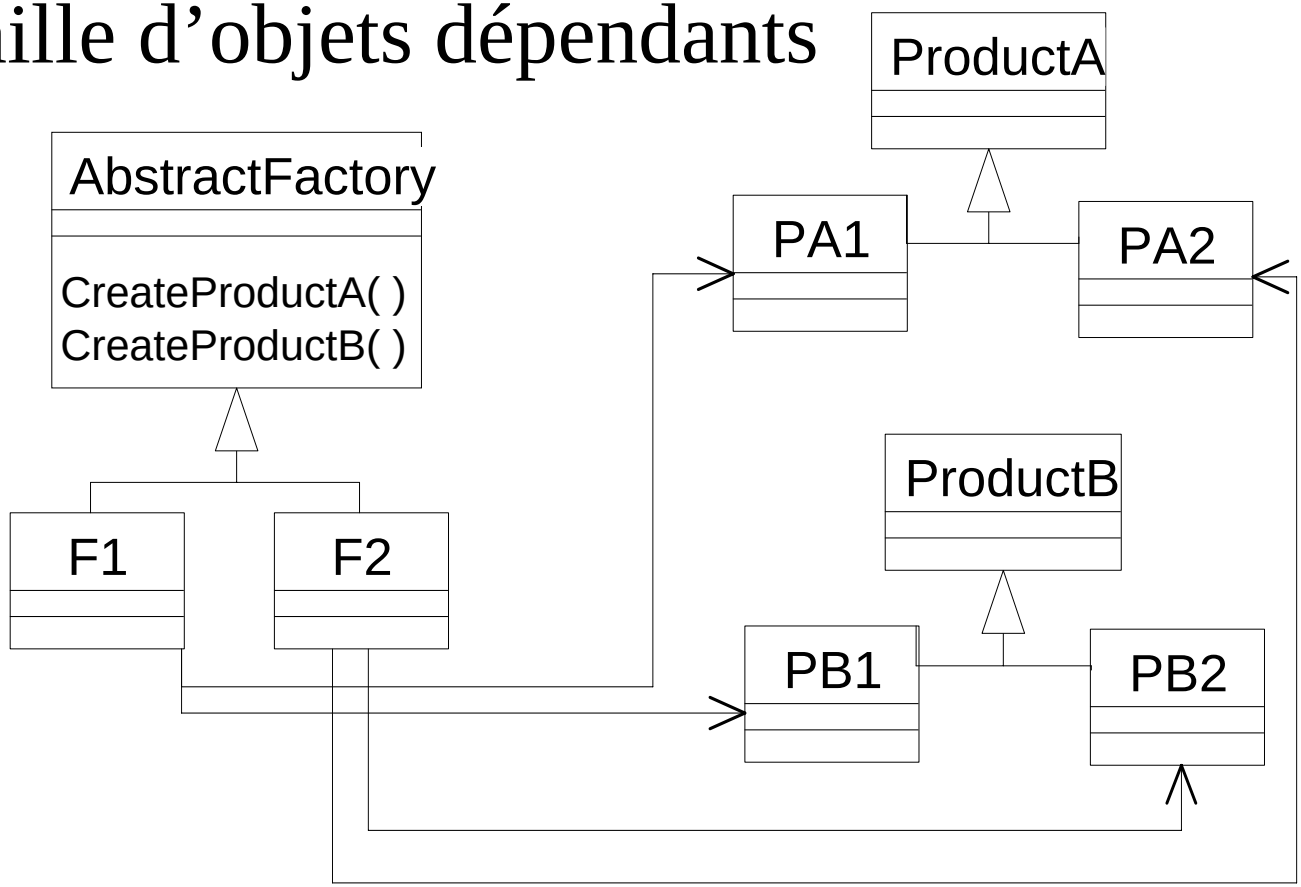
- Trouver les bons objets
- Il faut décomposer pour mieux recomposer
- Concilier : encapsulation, granularité, couplage, flexibilité, performance, évolution, réutilisabilité, ...
- Identifier les abstractions non évidentes, qui n'appartiennent pas au domaine, mais relèvent de la conception

Familles de patterns

- Création
- Structure
- Comportement

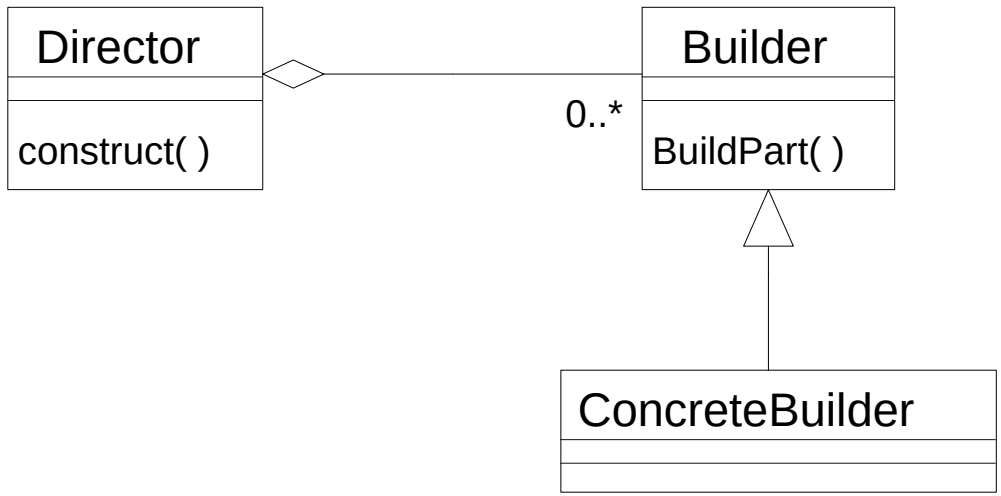
Abstract factory

- Famille d'objets dépendants



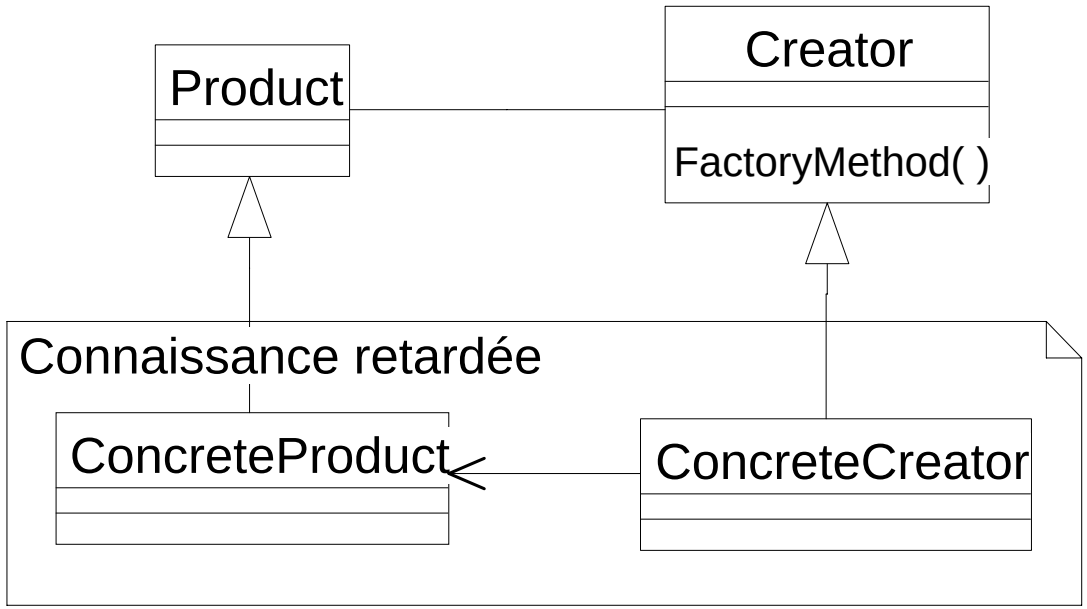
Builder

- L’algorithme de création est indépendant de la structure



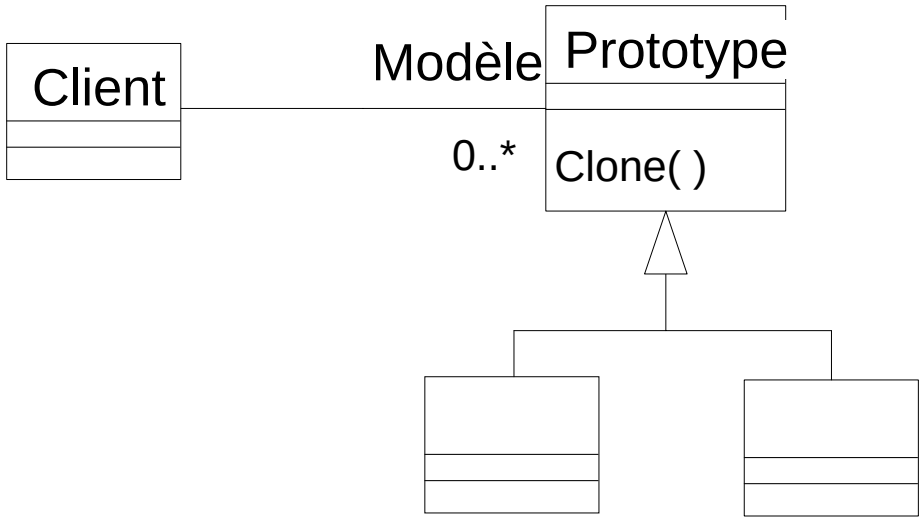
Factory method

- Permettre à une classe de créer des objets dont elle ne connaît pas la classe



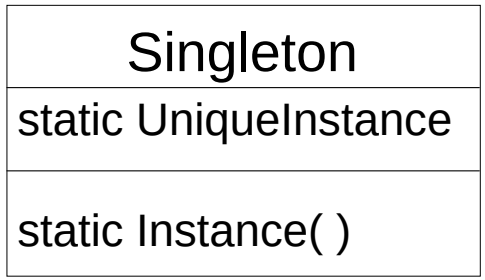
Prototype

- Création de nouveaux objets par copie d'un modèle



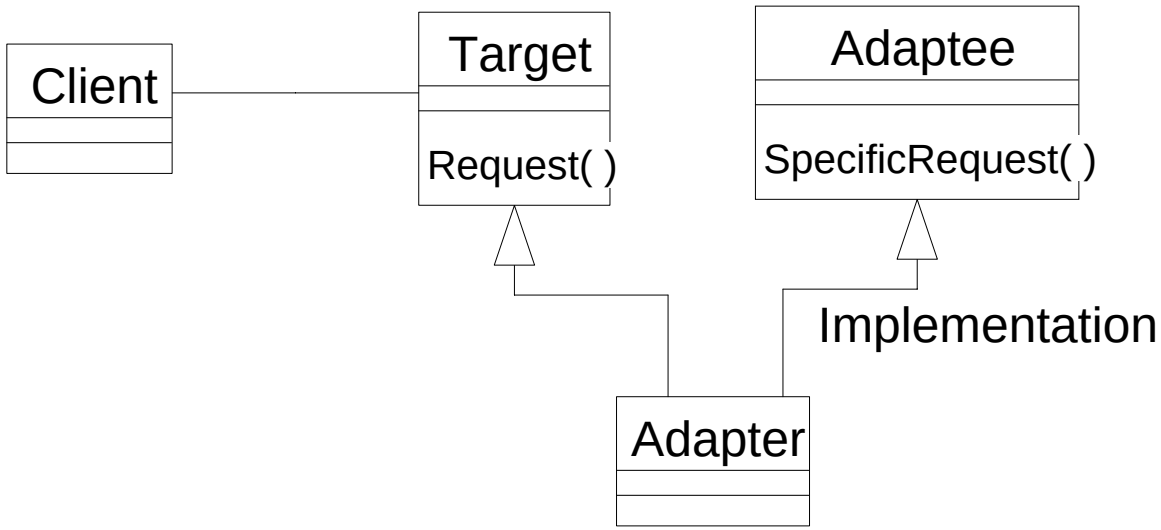
Singleton

- Classe qui ne donne qu'une seule instance



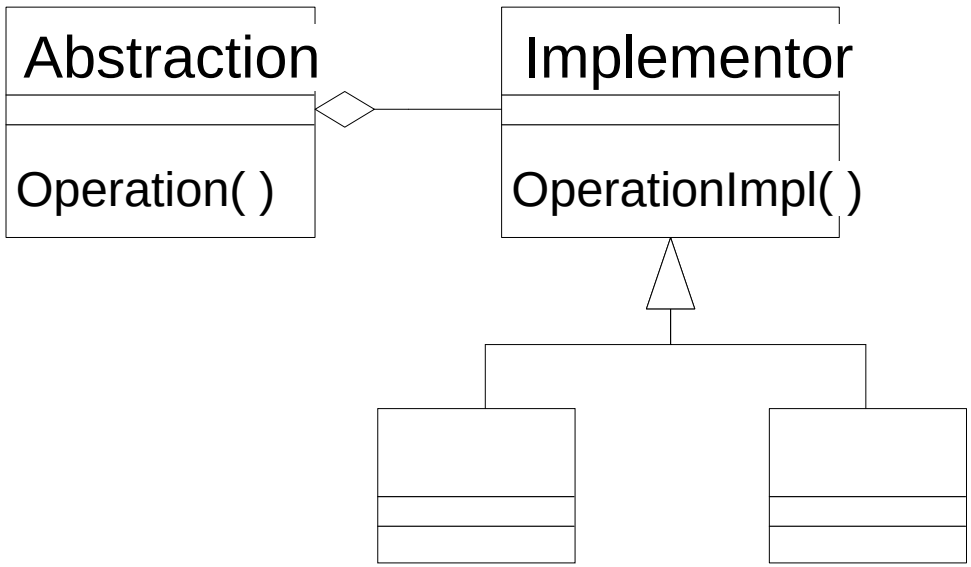
Adapter

- Convertir l’interface d’une classe



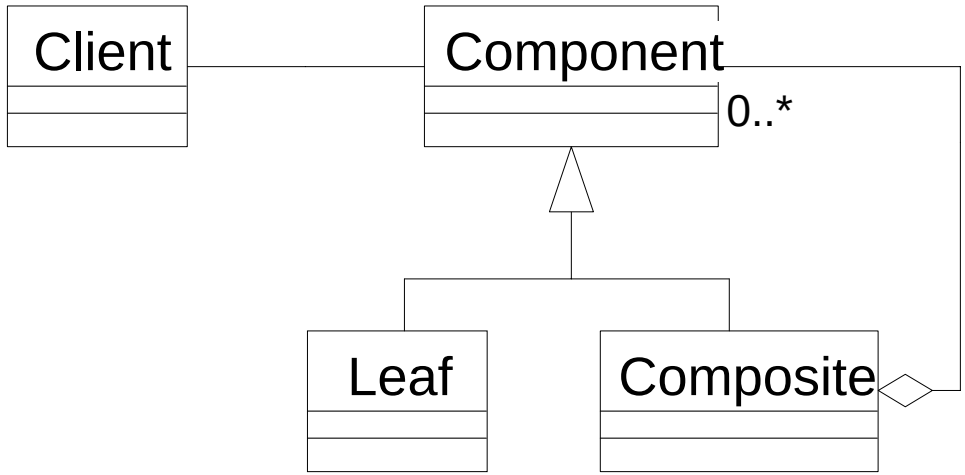
Bridge

- Decoupler le contrat et l'implémentation



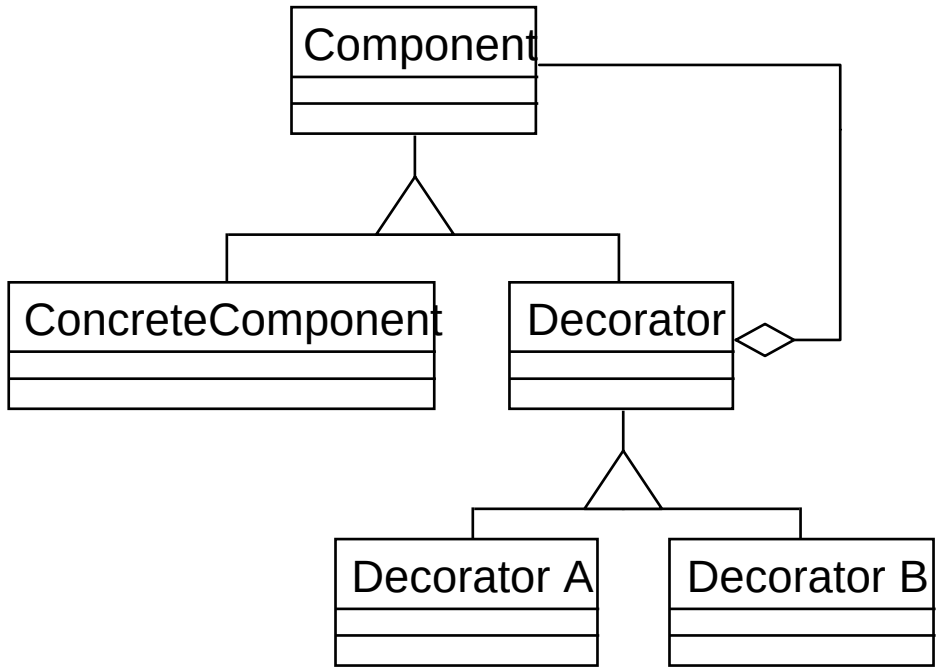
Composite

- Représentation de hiérarchies d'objets vus de manière uniforme par le client



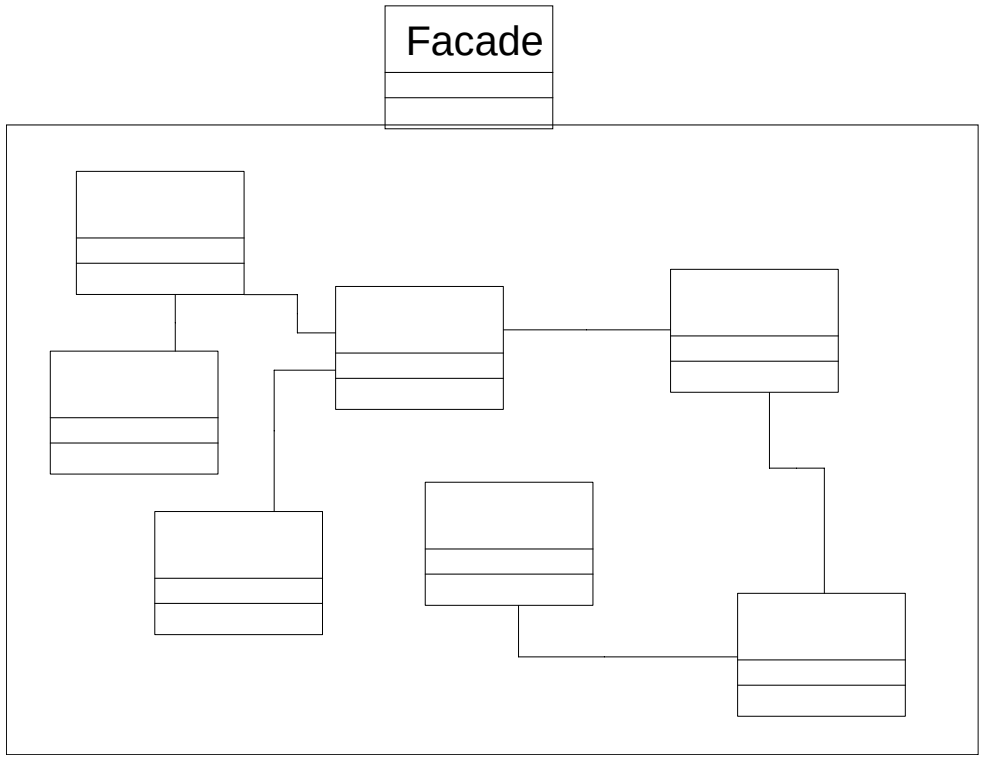
Decorator

- Ajouter des responsabilités dynamiquement



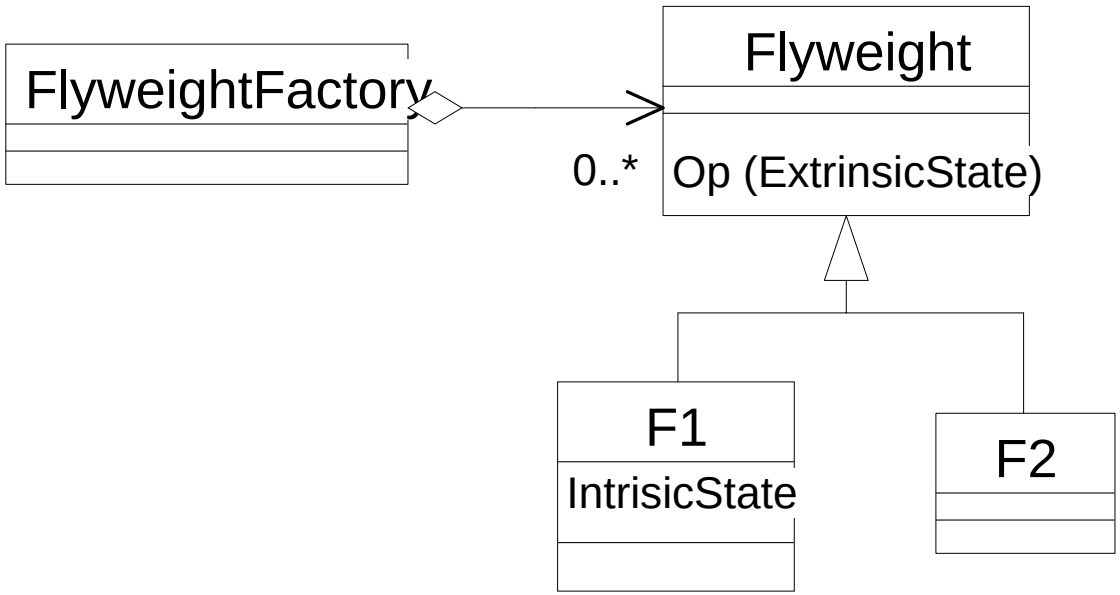
Façade

- Interface de sous-système simplifiée



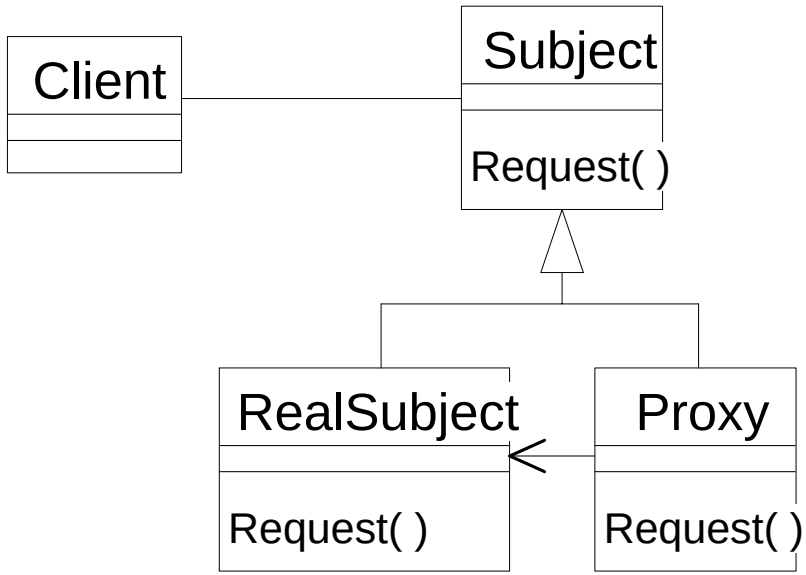
Flyweight

- Partager l’état extrinsèque



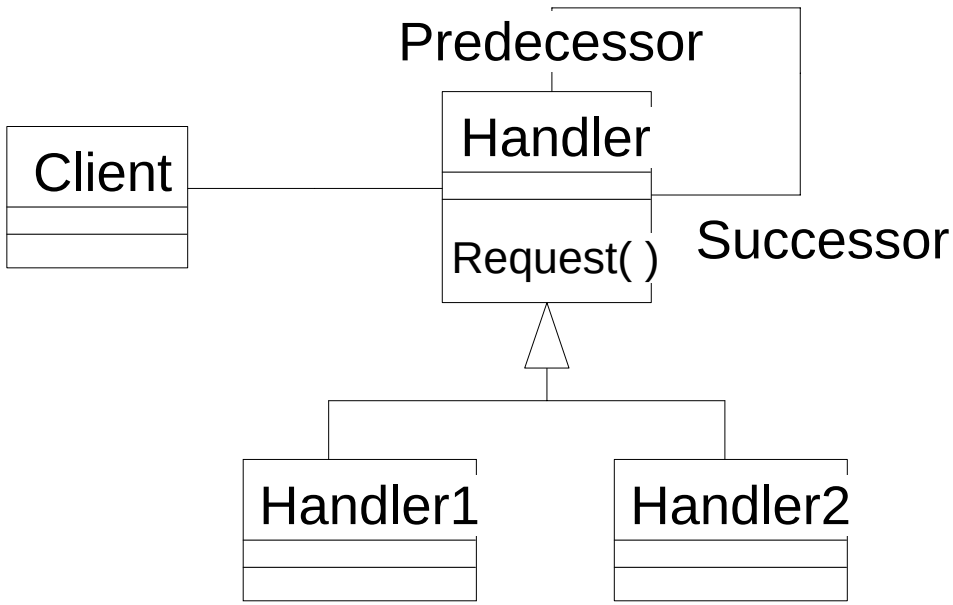
Proxy

- Objet miroir d'un autre objet plus lointain



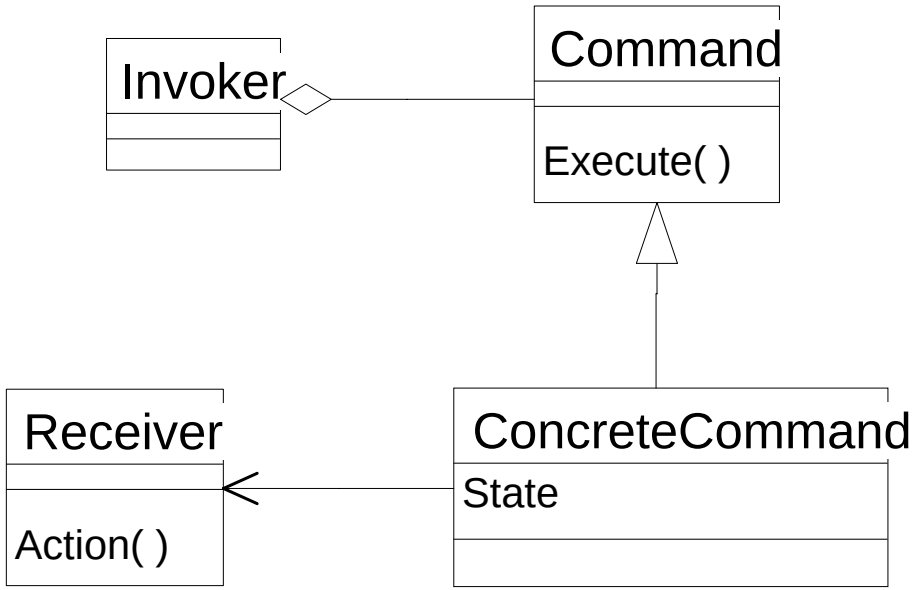
Chain of responsibility

- Découpler le client et le fournisseur



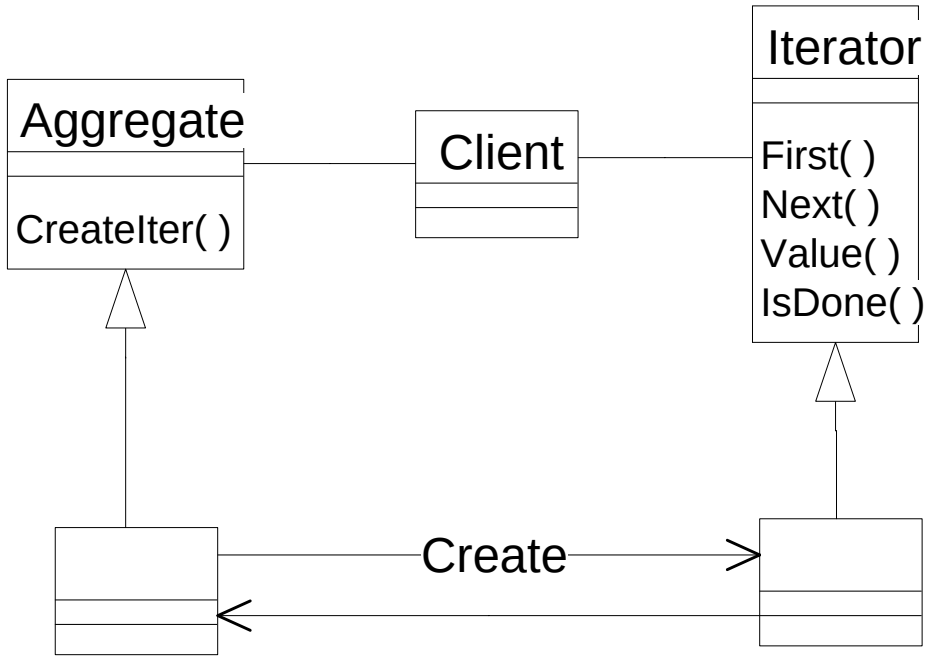
Command

- Réifier une commande, découpler le client du fournisseur



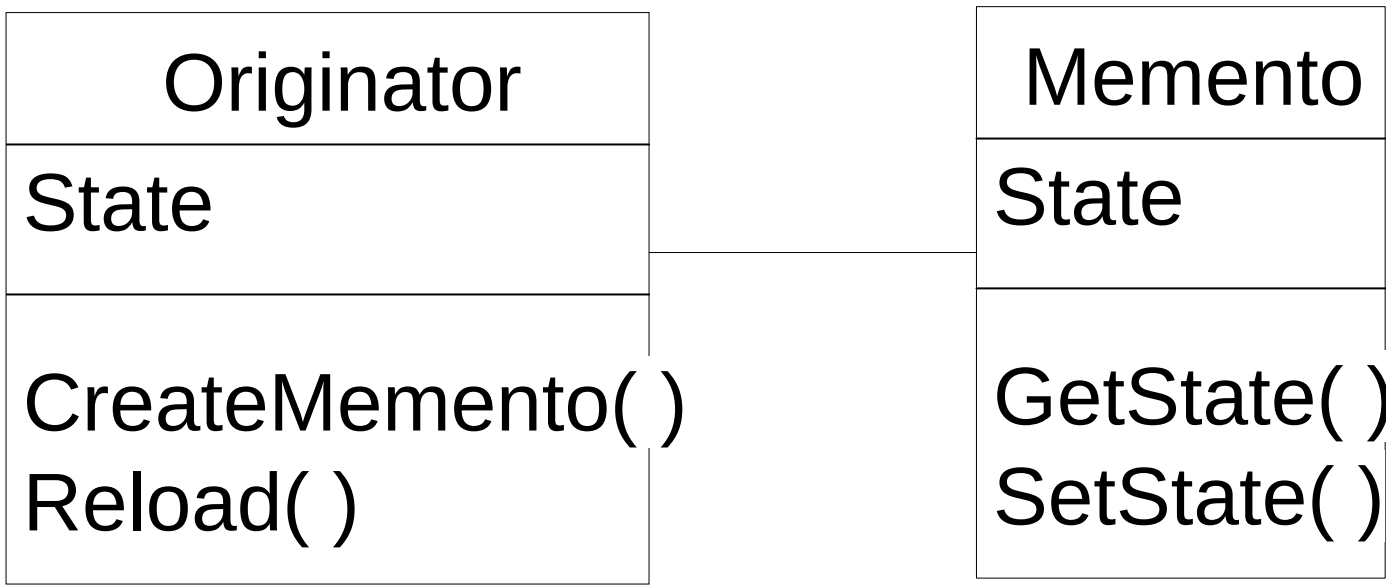
Iterator

- Visiter un objet sans connaître sa structure



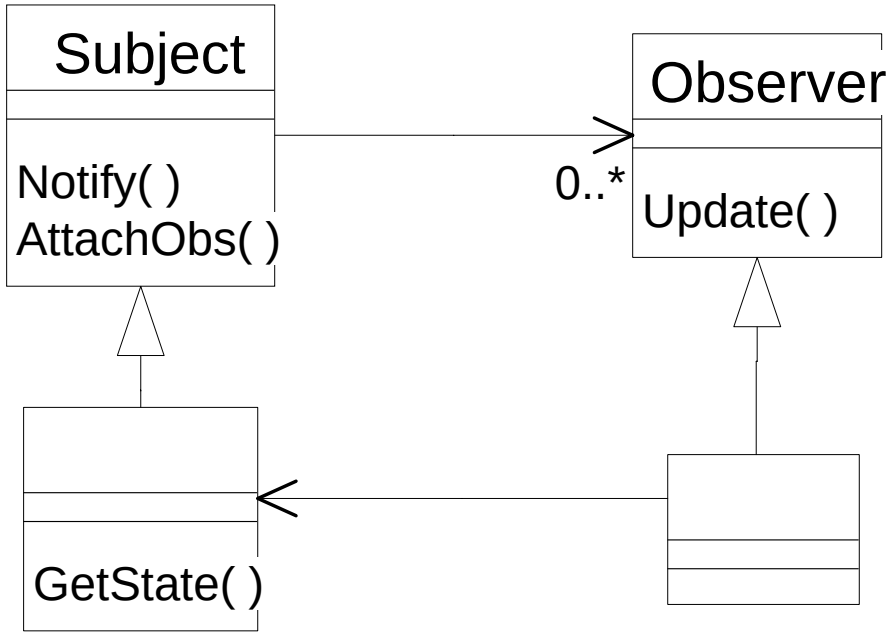
Memento

- Enregistrer l'état d'un objet sans briser l'encapsulation



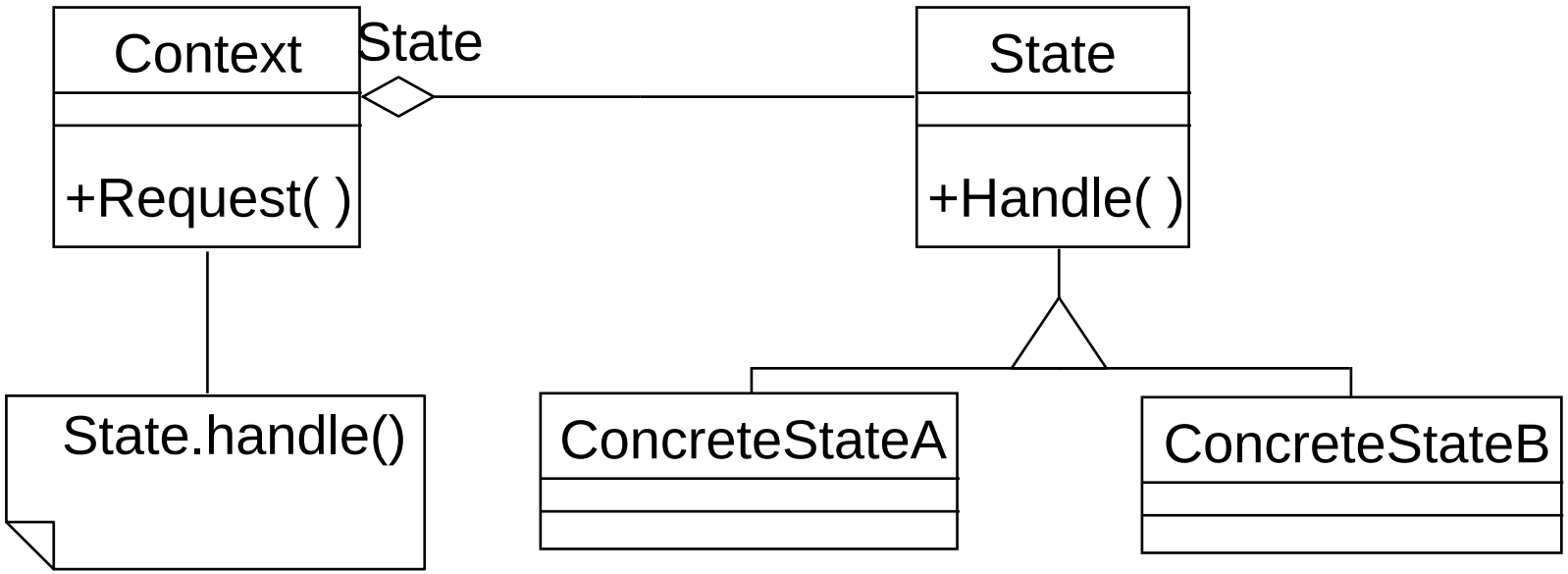
Observer

- Dépendance Publish-Subscribe



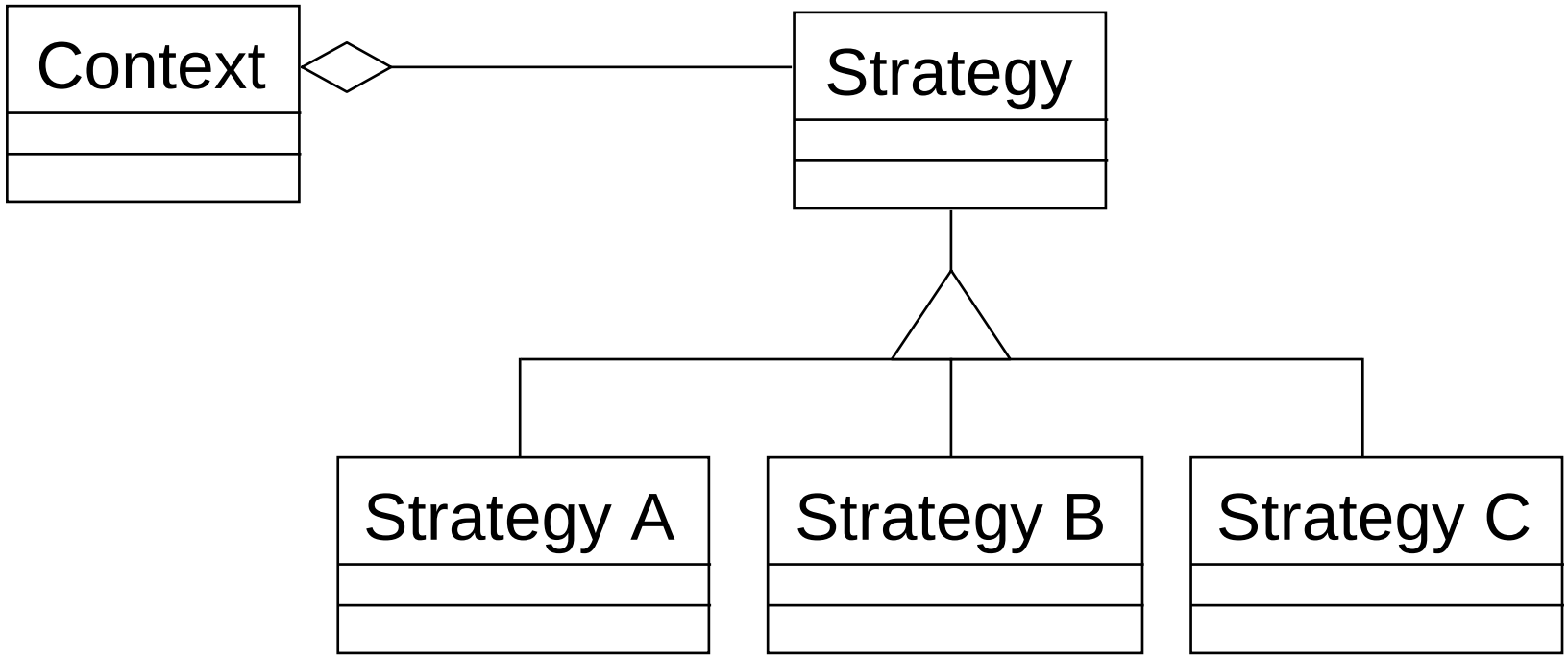
State

- Changer de comportement, “changer” de classe



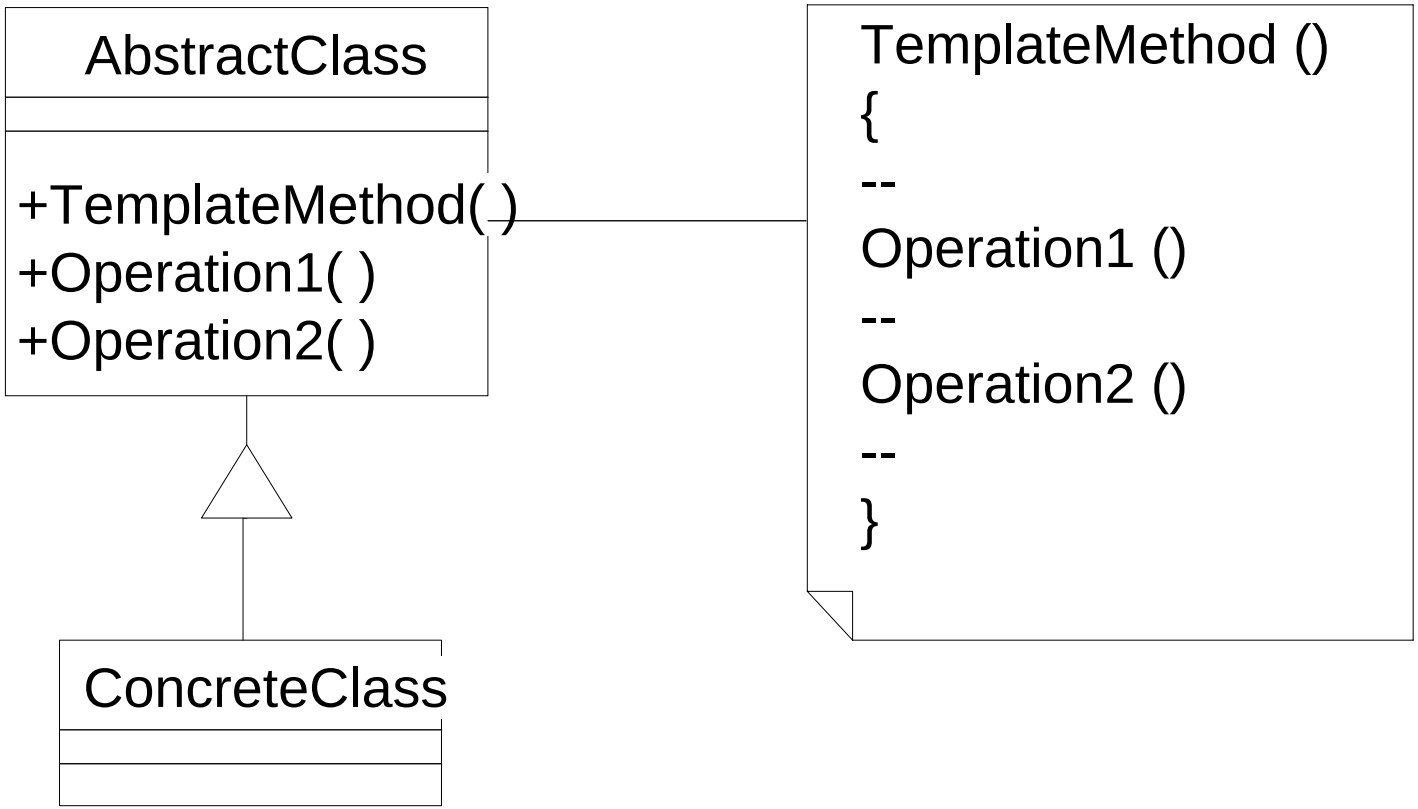
Strategy

- Famille d'algorithmes interchangeables



Template method

- Squelette d'algorithme



Conclusion

- Réutilisation de savoir faire objet
- Indépendance par rapport aux langages
- Composants plus abstraits
- Vers des macros-architectures (frameworks) plus faciles à comprendre