Introduction aux *GOF design patterns*
Extraits du catalogue de Gamma, Helm, Johnson et Vlissides

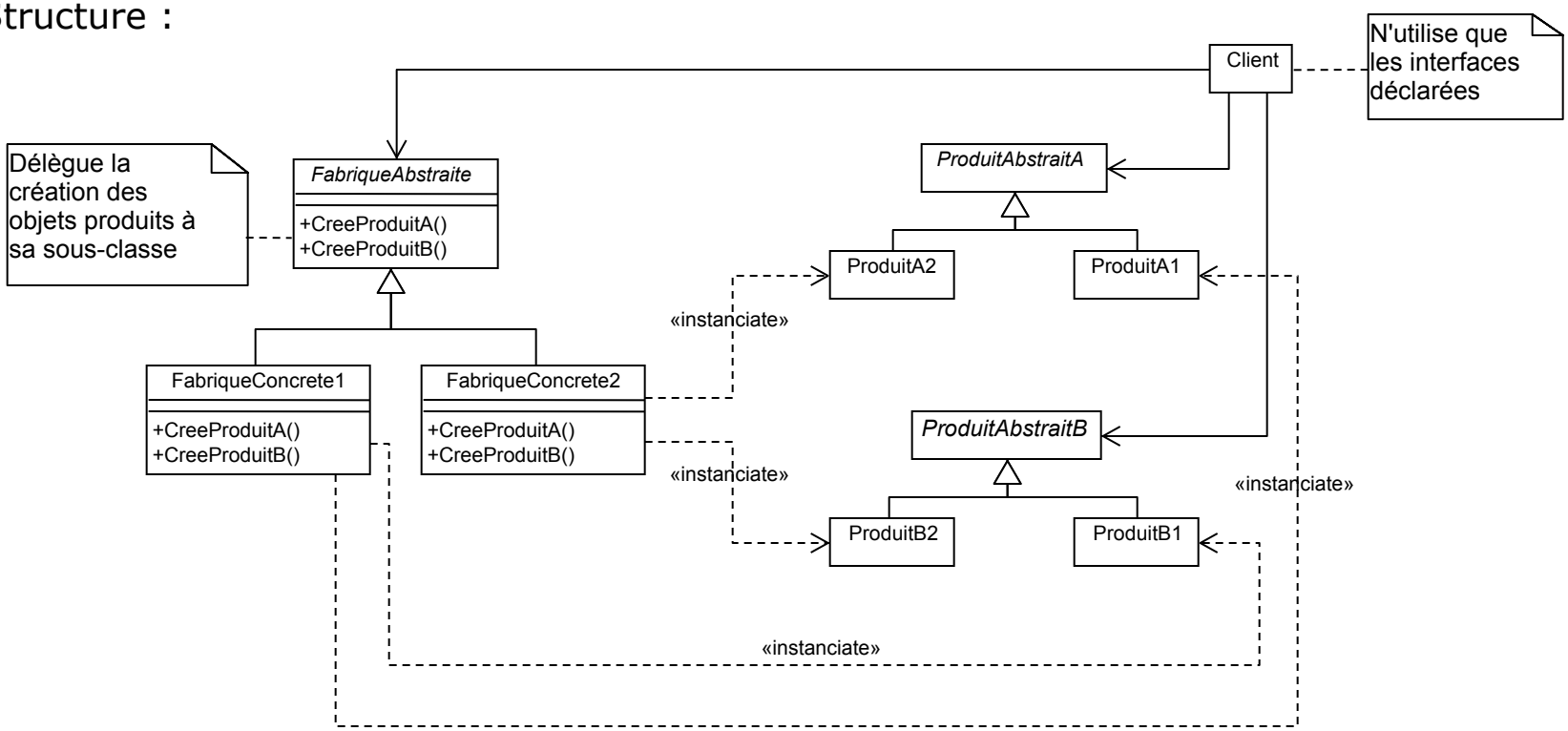
	Créateurs	Structuraux	Comportementaux
Classe	Factory Method	Adapter (classe)	Interpreter Template Method
Objet	Abstract Factory Builder Prototype Singleton	Adapter (objet) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Abstract Factory (Fabrique abstraite)

- Intention :

  Fournir une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes

- Utilisations :
  - Un système doit être indépendant de la façon dont ses produits sont créés, combinés et représentés
  - Un système doit être constitué à partir d'une famille de produits, parmi plusieurs
  - On souhaite renforcer le caractère de communauté d'une famille d'objet produits conçus pour être utilisés ensemble
  - On souhaite fabriquer une bibliothèque de classes de produits, en n'en révélant que l'interface et non l'implémentation
- Structure :

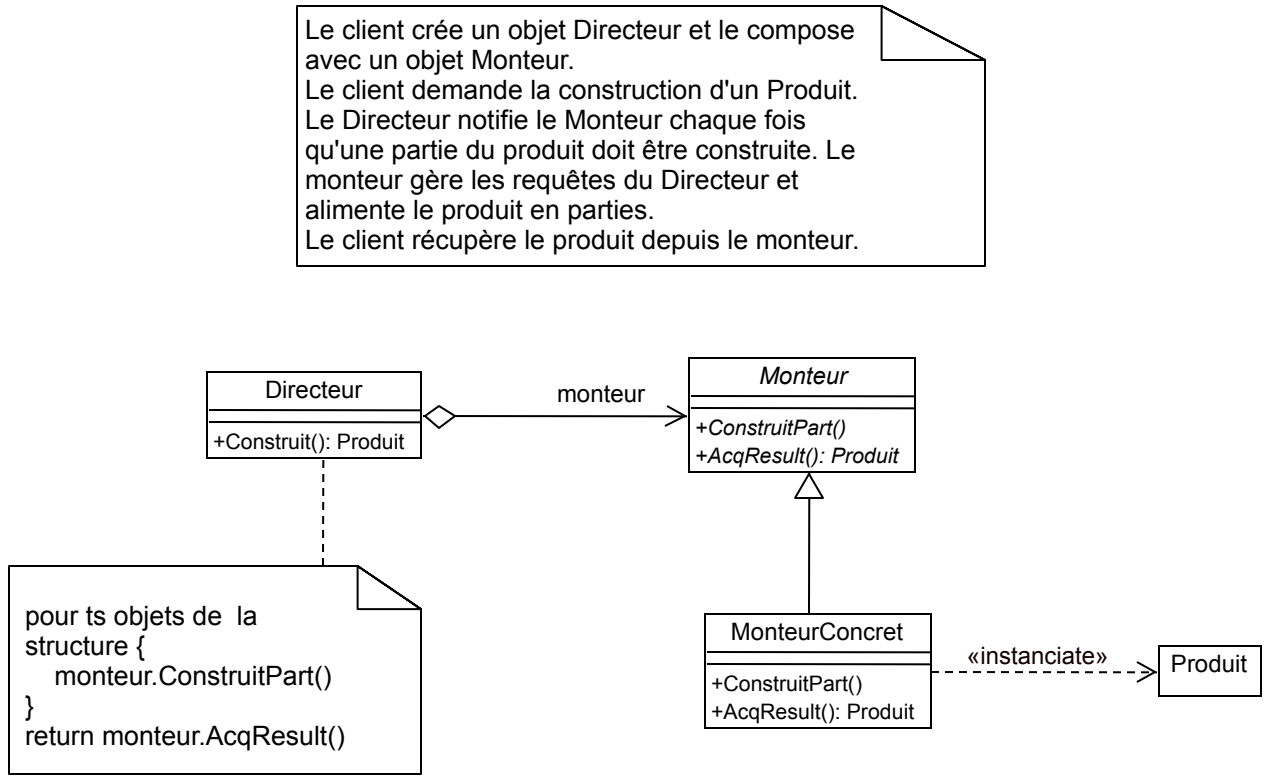


Builder (Monteur)

- Intention :

  Dissocie la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction permette des représentations différentes

- Utilisations :
  - L'algorithme de création d'un objet complexe doit être indépendant des parties qui composent l'objet et de la manière dont ces parties sont agencées
  - Le processus de construction doit autoriser des représentations différentes de l'objet en construction
- Structure :

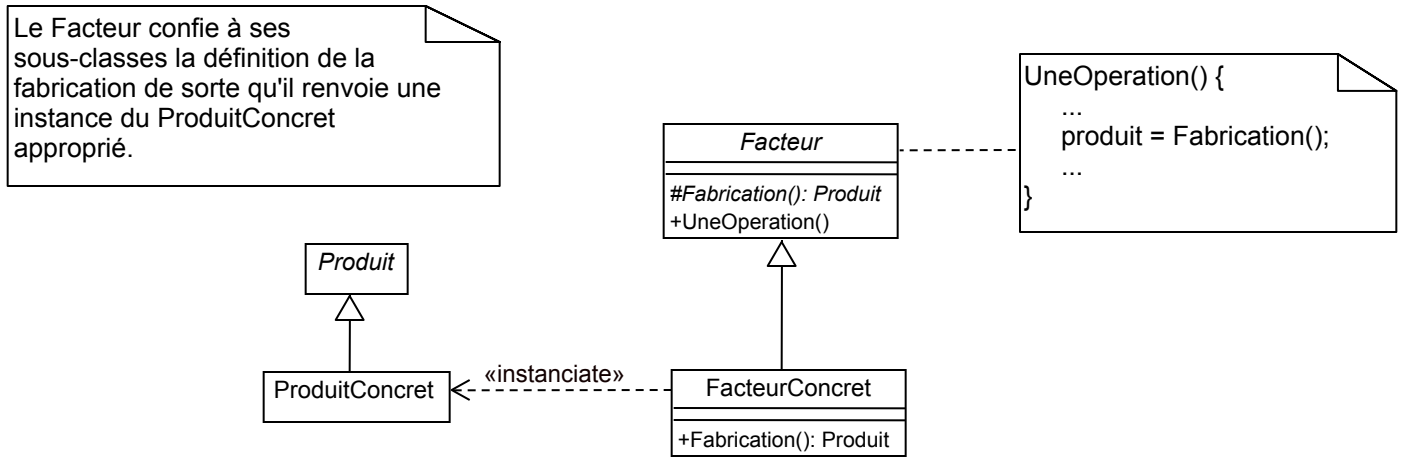


Factory Method (Fabrication)

- Intention :

  Définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier. Ce pattern permet à une classe de déléguer l'instanciation à des sous-classes

- Utilisations :
  - Une classe ne peut prévoir la classe des objets qu'elle aura à créer.
  - Une classe attend de ses sous-classes qu'elles spécifient les objets qu'elles créent.
  - Les classes délèguent des responsabilités à une de leurs nombreuses sous-classes assistances, et l'on veut disposer localement de l'information permettant de connaître la sous-classe assistante qui a reçu cette délégation.
- Structure :



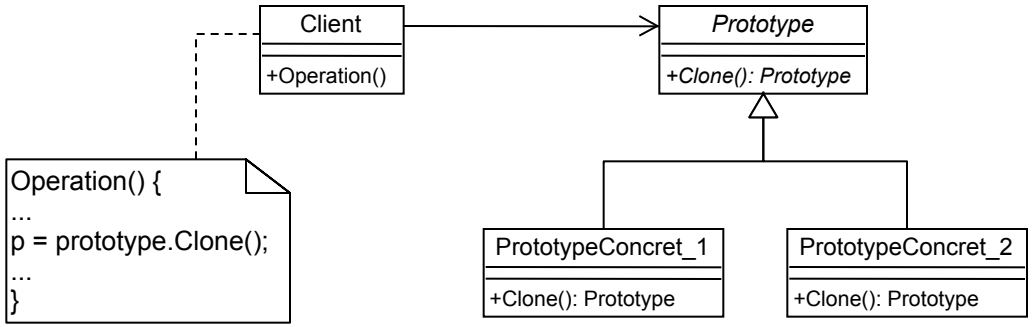
Prototype

- Intention :

  Spécifie le type des objets à créer à partir d'une instance de prototype, et crée de nouveaux objets en copiant ce prototype

- Utilisations :
  - On utilisera le pattern Prototype lorsqu'un système doit être indépendant de la manière dont ses produits sont créés, composés et représentés; et :
    - si les classes à instancier sont spécifiées à l'exécution, par exemple, par chargement dynamique; *ou bien*
    - pour éviter de construire une hiérarchie de classes de fabriques, qui réplique la hiérarchie de classes de produits; *ou encore*
    - si les instances d'une classe peuvent prendre un état parmi un petit nombre de combinaisons. Il peut être approprié d'installer le nombre requis de prototypes et d'en faire des clones, plutôt que d'instancier chaque fois la classe manuellement avec l'état correspondant.

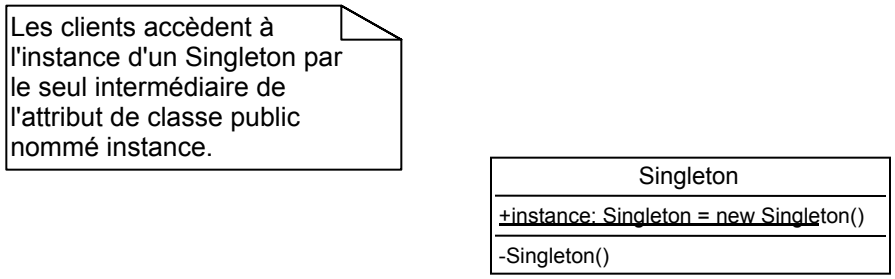
- Structure :



Singleton

- Intention :

  Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès de type global à cette classe
- Utilisations :
  - S'il doit n'y avoir exactement qu'une instance d'une classe, qui, de plus, doit être accessible aux clients en un point bien déterminé.
  - Si l'instance unique doit être extensible par dérivation en sous-classe, et si l'utilisation d'une instance étendue doit être permise aux clients, sans qu'ils aient à modifier leur code.
- Structure :



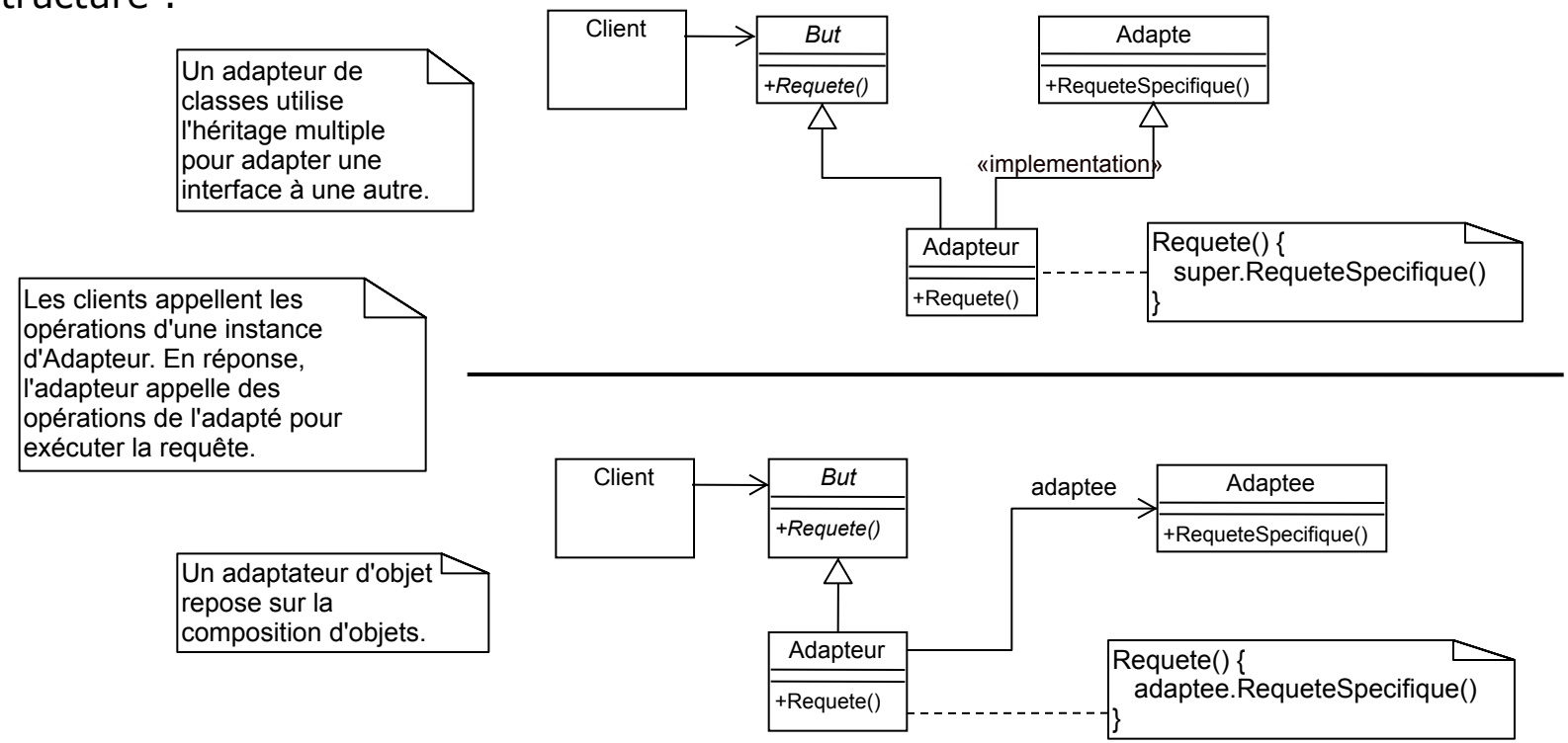
Adapter (class and object)

- Intention :

  Convertit l'interface d'une classe en une autre conforme à l'attente du client. Ce pattern permet à des classes de collaborer, qui n'auraient pu le faire du fait d'interfaces incompatibles

- Utilisations :
  - On veut utiliser une classe existante mais dont l'interface ne coïncide pas avec celle escomptée.
  - On souhaite créer une classe réutilisable qui collabore avec des classes sans relations avec elle et encore inconnues, c'est-à-dire avec des classes qui n'auront pas nécessairement des interfaces compatibles.
  - (Pour le cas *adapter object* uniquement) On a besoin d'utiliser plusieurs sous-classes existantes, mais l'adaptation de leur interface par dérivation de chacune d'entre elles est impraticable. Un adapter objet peut adapter l'interface de sa classe parente.

- Structure :



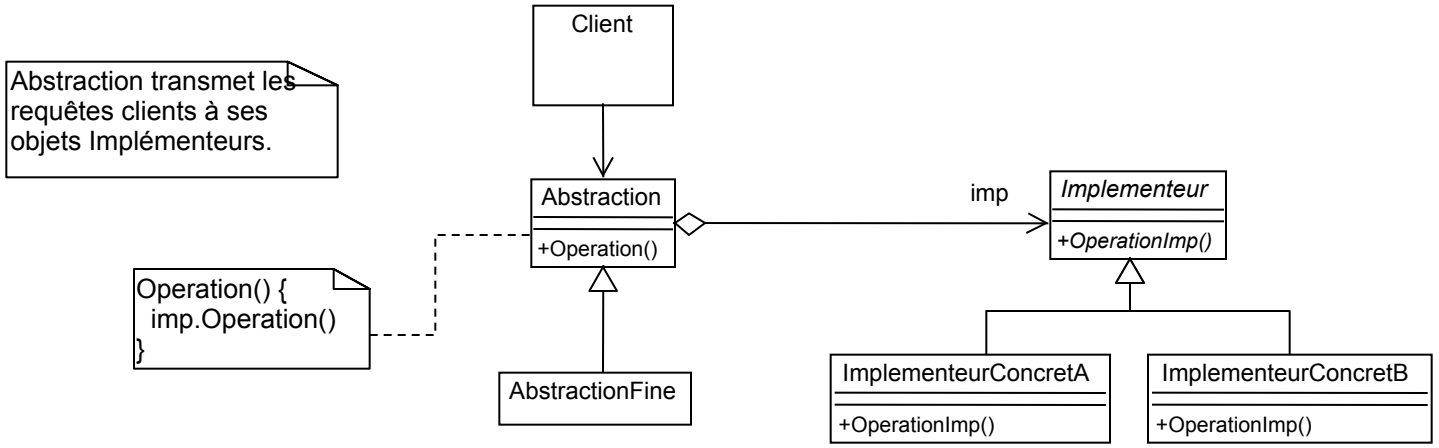
Bridge (Pont)

- Intention :

  Découple une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre.

- Utilisations :
  - On veut éviter un lien permanent entre l'abstraction et l'implantation (ex: l'implantation est choisie à l'exécution).
  - L'abstraction et l'implantation sont toutes les deux susceptibles d'être raffinées.
  - Les modifications subies par l'implantation ou l'abstraction ne doivent pas avoir d'impacts sur le client (pas de recompilation).

- Structure :

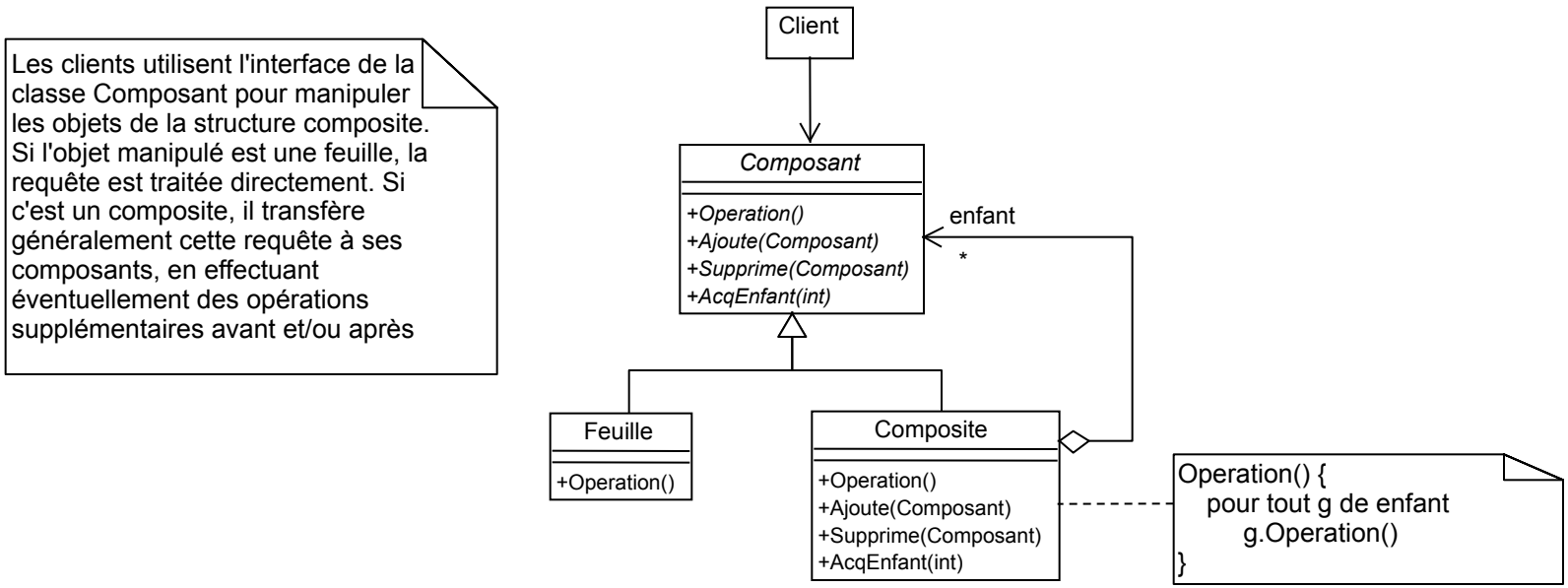


Composite

- Intention :

  Le pattern Composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même et unique façon les objets individuels et les combinaisons de ceux-ci.

- Utilisations :
  - On souhaite représenter des hiérarchies de l'individu à l'ensemble.
  - On souhaite que le client n'ait pas à se préoccuper de la différence entre combinaisons d'objets et objets individuels. Les clients pourront traiter de façon uniforme tous les objets de la structure composite.
- Structure :

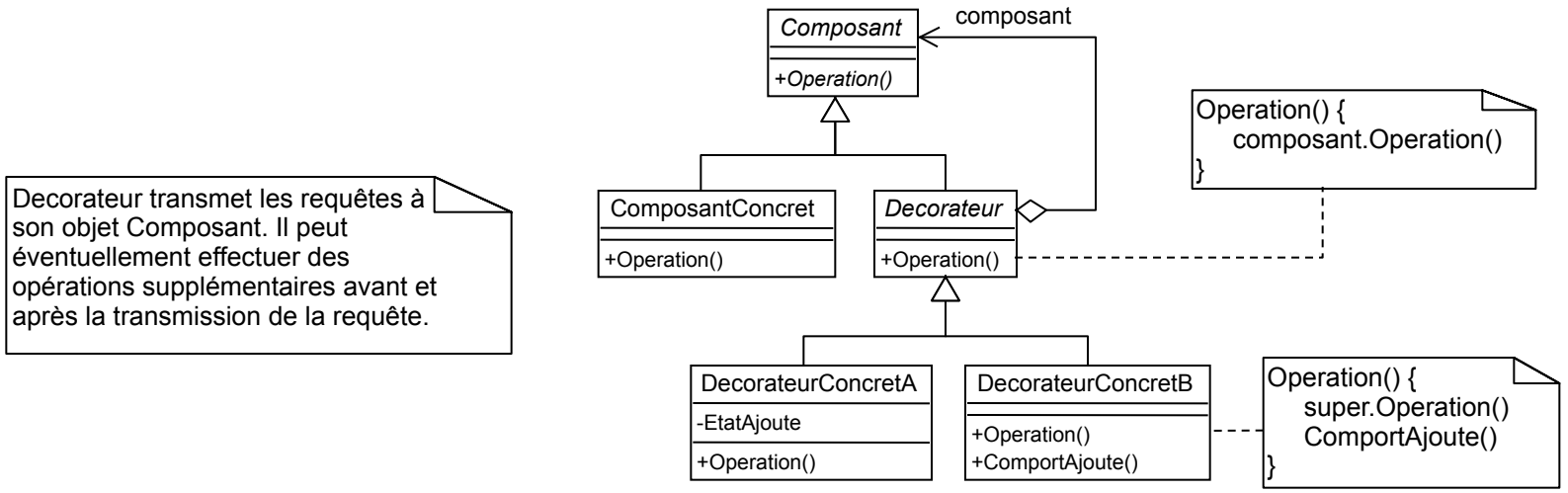


Decorator (Décorateur)

- Intention :

  Attache dynamiquement des responsabilités supplémentaires à un objet. Les décorateurs fournissent une alternative souple à la dérivation, pour étendre les fonctionnalités.

- Utilisations :
  - Pour ajouter dynamiquement des responsabilités à des objets individuels, ceci d'une façon transparente, c'est-à-dire, sans affecter les autres objets.
  - Pour des responsabilités qui doivent pouvoir être retirées.
  - Des extensions sont indépendantes et il serait impraticable de les implanter par dérivation.
- Structure :

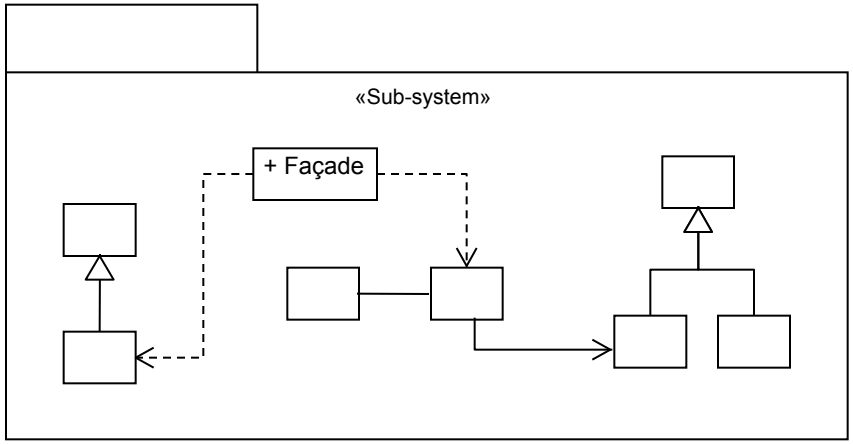


Facade (Façade)

- Intention :

  Fournit une interface, à l'ensemble des interfaces d'un sous-système. La façade fournit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.
- Utilisations :
  - Fournir une interface simple à un système complexe.
  - Introduire une interface pour découpler les relations entre deux systèmes complexes.
  - Construire le système en couche
- Structure :

Les clients communiquent avec le le sous-système
en envoyant des requêtes à la façade, qui répercute celles-ci aux objets appropriés du sous-système. Bien que les objets du sous-système effectuent réellement le travail, la façade peut avoir à charge, en propre, le travail de transcription de son interface à ceux du sous-système.
Le clients qui utilisent la façade n'ont pas à accéder directement aux objets de son sous-système.



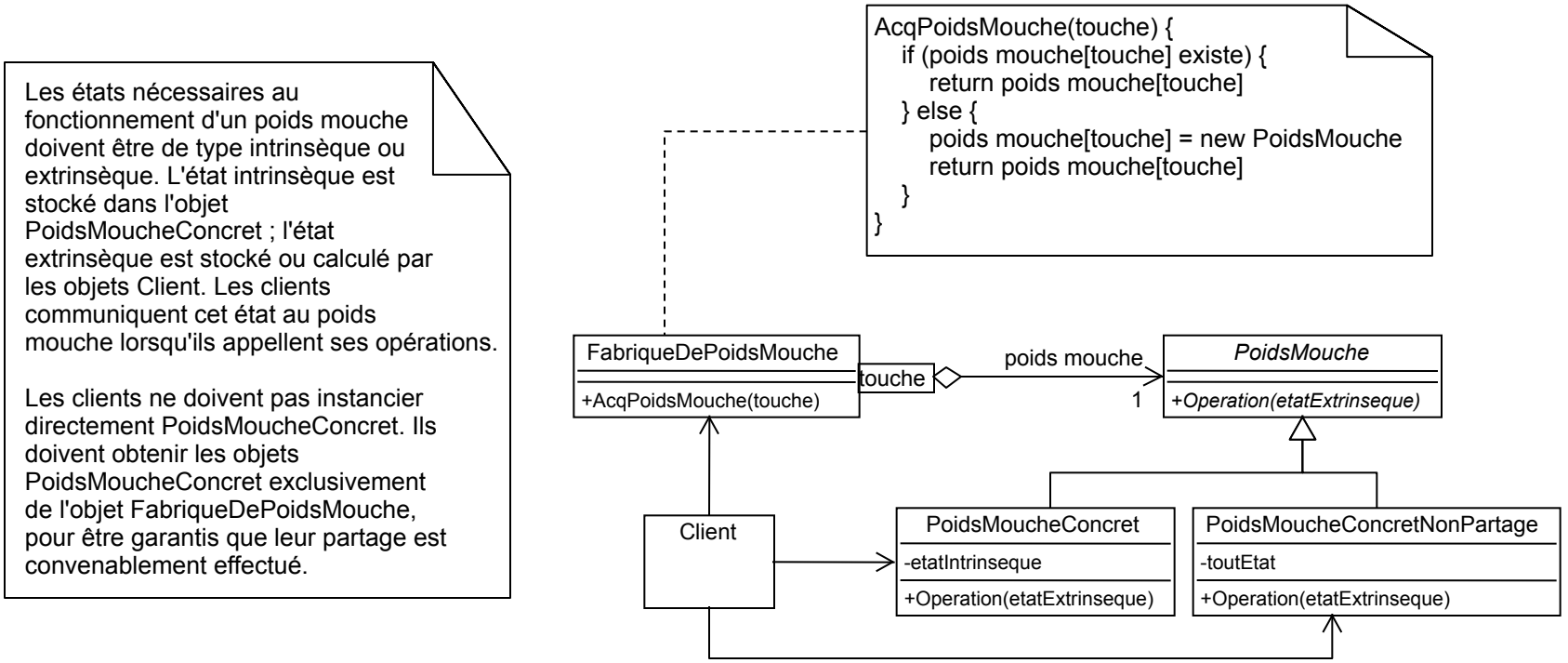
Flyweight (Poids Mouche)

- Intention :

  Le pattern flyweight utilise une technique de partage qui permet la mise en œuvre efficace d'un grand nombre d'objets de fine granularité.

- Utilisations :
  - L'application utilise un grand nombre d'objets.
  - Les coûts de stockage sont élevés du fait d'une réelle quantité d'objets.
  - L'état des objets peut être externalisé.
  - De nombreux groupes d'objets peuvent être remplacés par quelques objets partagés une fois que les états sont externalisés.
  - L'application ne dépend pas de l'identité des objets.

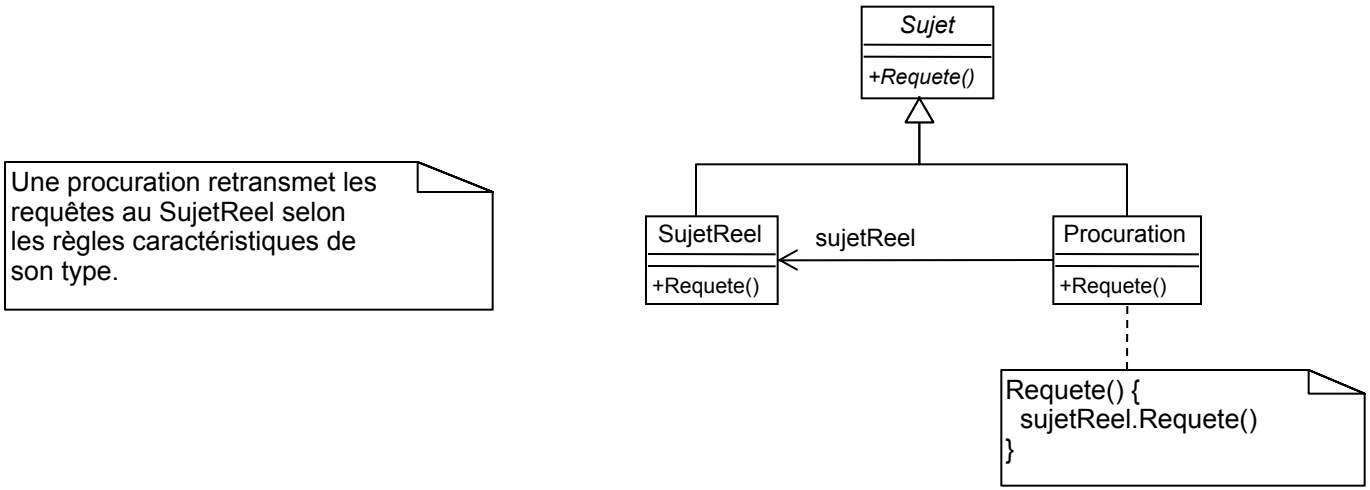
- Structure :



Proxy (Procuration)

- Intention :

  Fournit à un tiers objet un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.
- Utilisations :
  - On utilise le Proxy lorsqu'on veut référencer un objet par un moyen plus complexe qu'un pointeur :
    - Remote proxy : ambassadeur
    - Protection proxy : contrôle d'accès
    - Référence intelligente (persistance, comptage de référence, …)
- Structure :

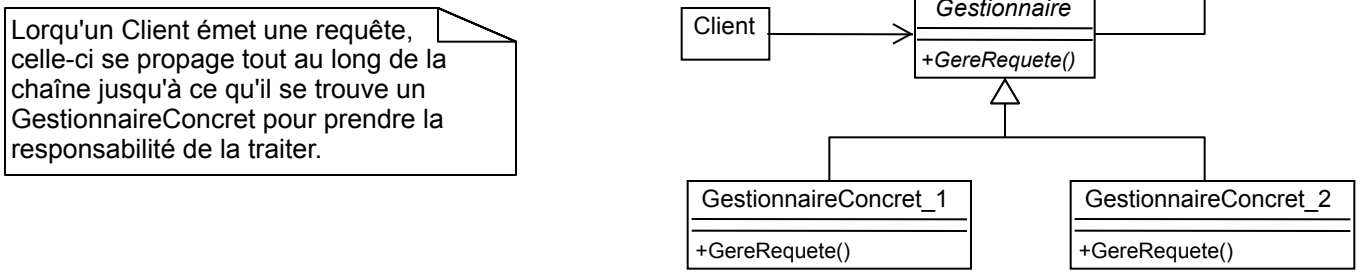


Chain of Responsability (Chaîne de Responsabilités)

- Intention :

  Eviter le couplage de l'émetteur d'une requête à ses récepteurs, en donnant à plus d'un objet la possibilité d'entreprendre la requête. Chaîner les objets récepteurs et faire passer la requête tout au long de la chaîne, jusqu'à ce qu'un objet la traite.

- Utilisations :
  - Une requête peut être gérée par plus d'un objet à la fois, et que le gestionnaire n'est pas connu a priori. Ce dernier doit être déterminé automatiquement.
  - On souhaite adresser une requête à un ou plusieurs objets, sans spécifier explicitement le récepteur.
  - L'ensemble des objets qui peuvent traiter une requête doit être défini dynamiquement.
- Structure :



Command (Commande)

- Intention :

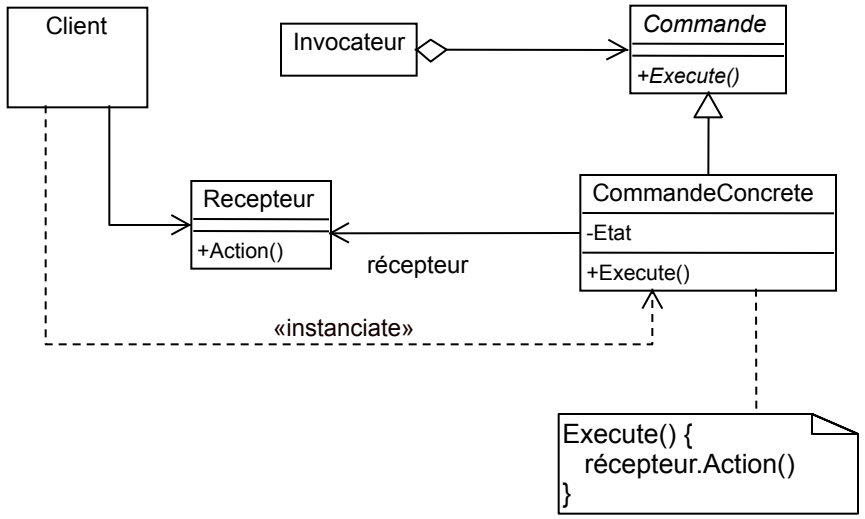
  Encapsuler une requête comme un objet, autorisant ainsi le paramétrage des clients par différentes requêtes, files d'attente et récapitulatifs de requêtes, et de plus, permettant la réversion des opérations

- Utilisations :
  - Spécifier, stocker et exécuter des actions à des moments différents.
  - On veut pouvoir « défaire » (*undo*). Les commandes exécutées peuvent être stockées ainsi que les états des objets affectés…
  - On veut implanter des transactions; actions de « haut-niveau ».
- Structure :

Le client crée un objet CommandeConcrete et établit les spécifications de son récepteur.
Un objet Invocateur stocke l'objet CommandeConcrete.

L'invocateur lance une requête en appelant la fonction exécute de la commande. Lorsque les commandes doivent pouvoir être inversées, CommandeConcrete, avant d'invoquer Execute, stocke l'état qui permettra de renverser l'exécution de la commande.

L'objet CommandeConcrete invoque les opérations de son récepteur pour satisfaire la requête.



Interpreter (Interpréteur)

- Intention :

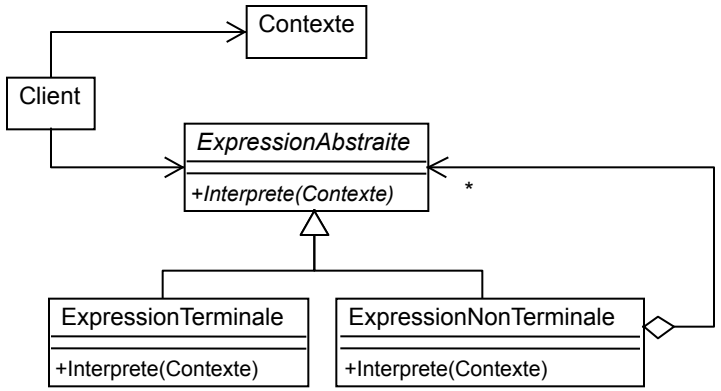
  Pour un langage donné, définir une représentation de sa grammaire, en même temps qu'un interpréteur utilisant cette représentation pour interpréter les phrases du langage.

- Utilisations :
  - La grammaire est simple
  - L'efficacité n'est pas un paramètre critique.
- Structure :

Le Client construit (ou reçoit tout construit) un arbre syntaxique abstrait composé d'instances de ExpressionNonTerminale et de ExpressionTerminale. Il initialise ensuite le contexte et invoque l'opération interprète.

Chaque noeud du type ExpressionNonTerminale définit Interprete en termes d'Interprete pour chaque sous-expression. L'opération Interprete de chaque ExpressionTerminale constitue le plancher de la récursion.

Les opérations Interprete de chaque noeud utilisent le contexte pour stocker et accéder à l'état de l'Interpréteur.

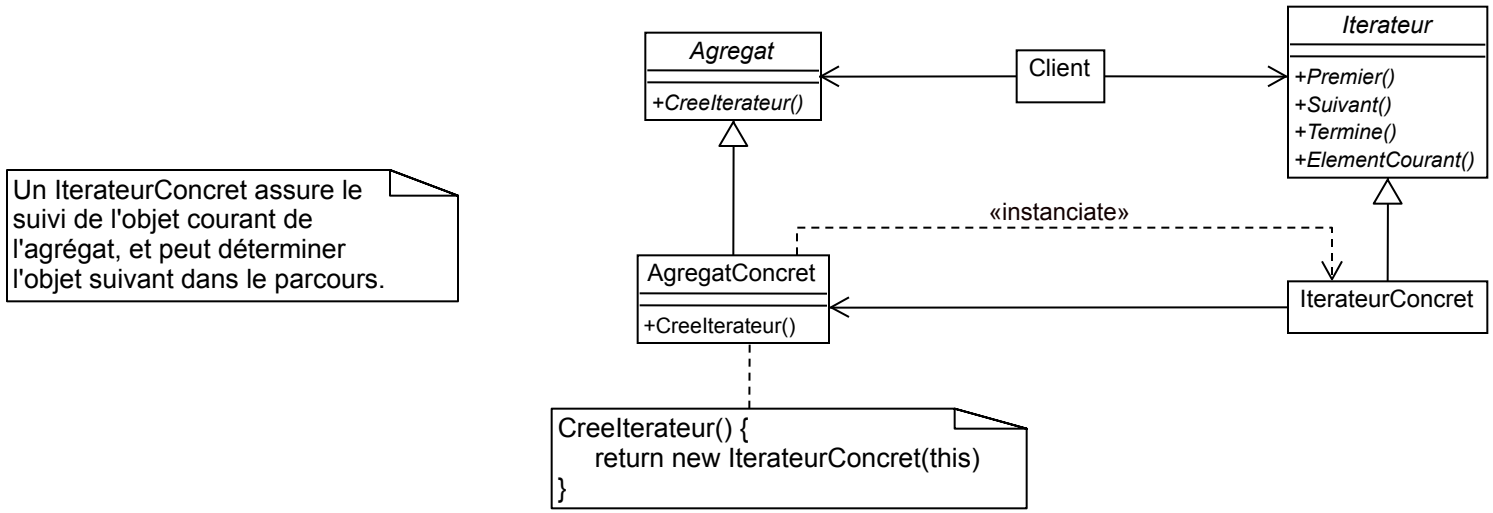


Iterator (Itérateur)

- Intention :

  Fournit un moyen d'accès séquentiel aux éléments d'un agrégat d'objets, sans mettre à découvert la représentation interne de celui-ci.

- Utilisations :
  - Pour accéder au contenu d'un objet d'un agrégat sans en révéler la représentation interne.
  - Pour gérer simultanément plusieurs parcours dans des agrégats d'objets.
  - Pour offrir une interface uniforme pour les parcours au travers de diverses structures agrégats (c'est-à-dire, pour permettre l'itération polymorphe).
- Structure :

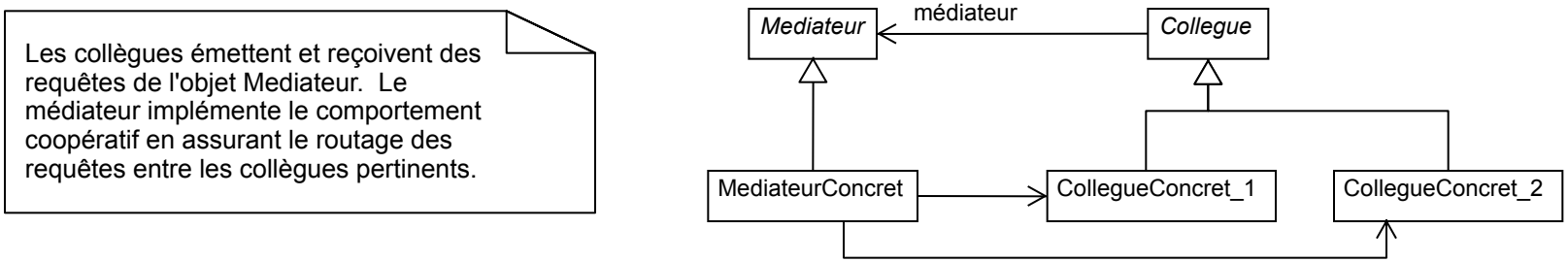


Mediator (Médiateur)

- Intention :

  Définit un objet qui encapsule les modalités d'interaction d'un certain ensemble d'objets. Ce pattern favorise le couplage faible en dispensant les objets de se faire explicitement référence, et il permet donc de faire varier indépendamment les relations d'interaction.

- Utilisations :
  - Les objets d'un ensemble communiquent d'une façon bien définie mais très complexe. Le résultat des interdépendances est non structuré et difficile à appréhender.
  - La réutilisation d'un objet est difficile, du fait qu'il fait référence à beaucoup d'autres objets et communique avec eux.
  - Un comportement distribué entre plusieurs classes doit pouvoir être spécialisé sans une pléthore de dérivations.
- Structure :

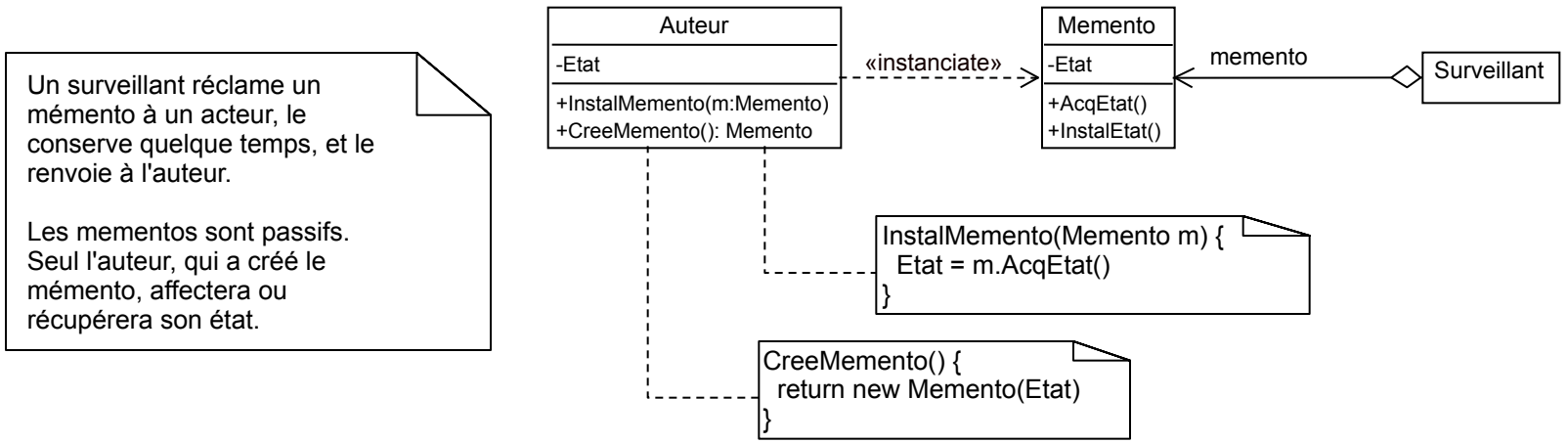


Memento

- Intention :

  Sans violation de l'encapsulation, saisir et transmettre à l'extérieur d'un objet, l'état interne de celui-ci, dans le but de pouvoir ultérieurement le restaurer dans cet état.

- Utilisations :
  - On veut sauvegarder tout ou partie de l'état d'un objet pour éventuellement pouvoir le restaurer.
  - Une interface directe pour obtenir l'état de l'objet briserait l'encapsulation.
- Structure :

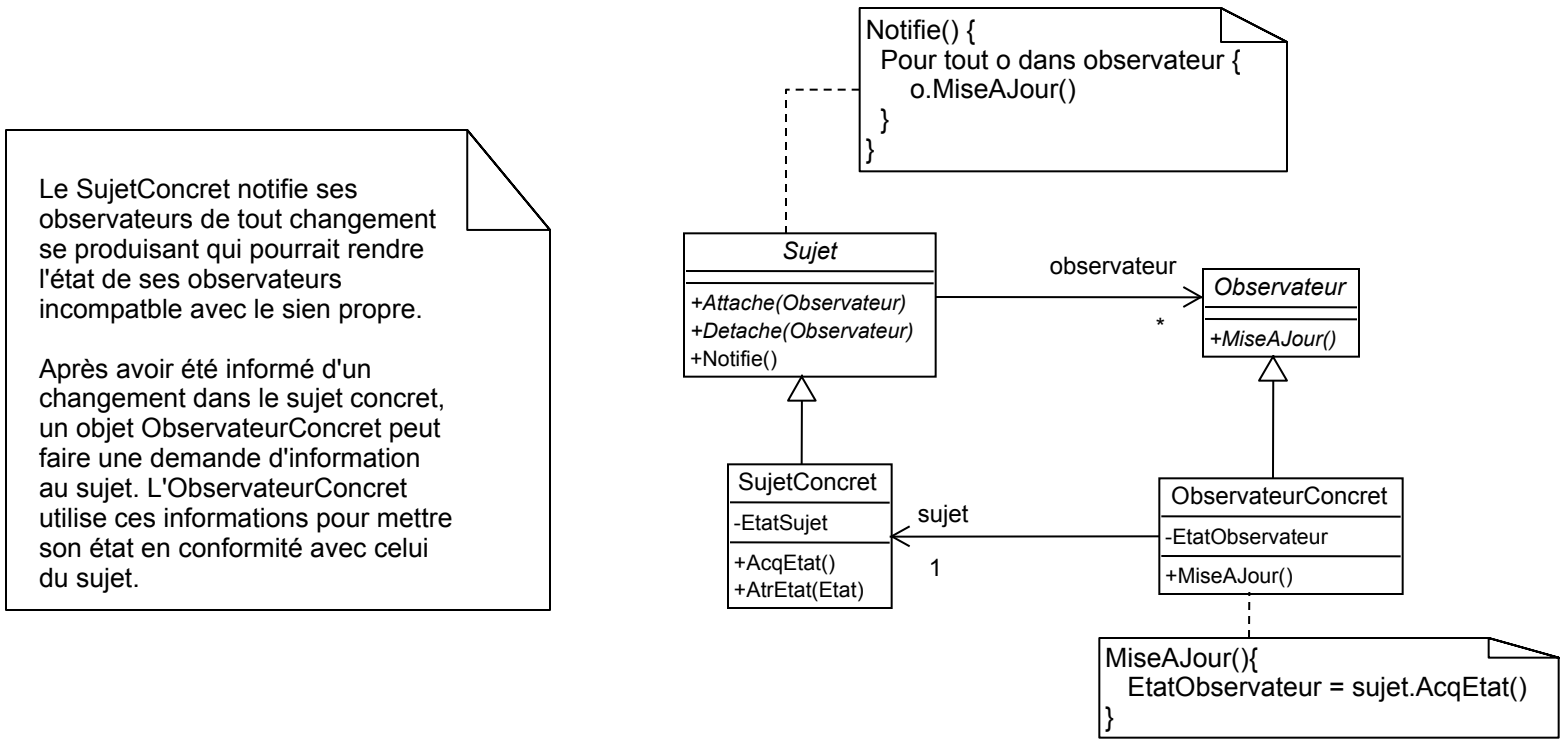


Observer (Observateur)

- Intention :

  Définit une interdépendance de type un ou plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour.

- Utilisations :
  - Une abstraction a plusieurs aspects, dépendant l'un de l'autre. Encapsuler ces aspects indépendamment permet de les réutiliser séparément.
  - Quand le changement d'un objet se répercute vers d'autres objets.
  - Quand un objet doit prévenir d'autres objets sans pour autant les connaître.
- Structure :



State (Etat)

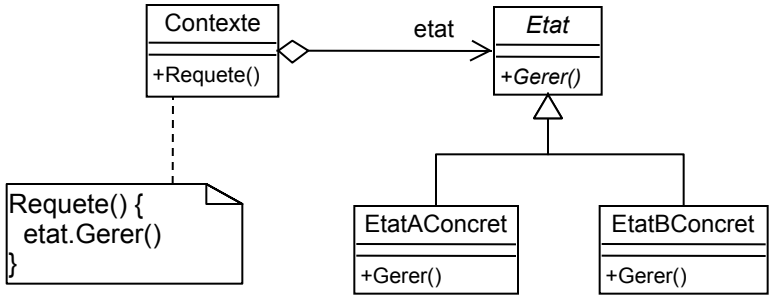
- Intention :

  Permet à un objet de modifier son comportement, quand son état interne change. Tout se passera comme si l'objet changeait de classe.

- Utilisations :
  - Le comportement d'un objet dépend de son état, qui change à l'exécution.
  - Les opérations sont constituées de partie conditionnelles de grande taille.
- Structure :

Le Contexte délègue les requêtes spécifiques d'état à l'objet EtatConcret courant.

Contexte est l'interface primaire pour les clients. Ceux-ci peuvent composer un contexte à l'aide d'objets Etat. Une fois qu'ils ont configuré un contexte, ils n'ont plus à "traiter" directement avec les objets Etat.



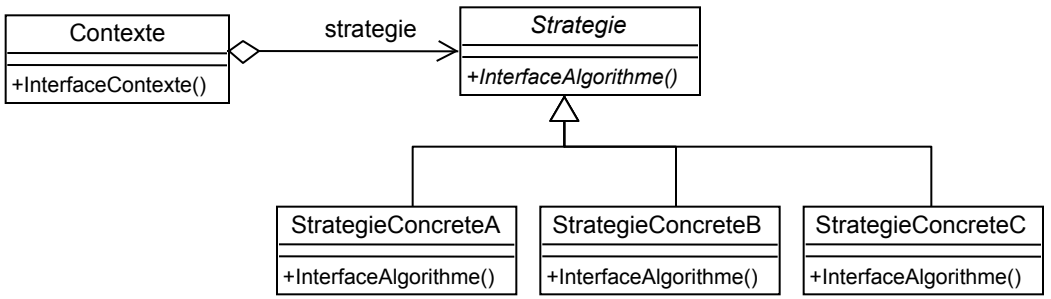
Strategy (Stratégie)

- Intention :

  Définit une famille d'algorithmes, encapsule chacun d'entre eux, et les rend interchangeables. Ce pattern permet aux algorithmes d'évoluer indépendamment des clients qui les utilisent.
- Utilisations :
  - De nombreuses classes associées ne diffèrent que par leur comportement. Strategy offre un moyen de configurer une classe avec un comportement parmi plusieurs.
  - On a besoin de plusieurs variantes d'algorithme.
  - Un algorithme utilise des données que les clients ne doivent pas connaître.
- Structure :

La classe Contexte fournit à la stratégie toutes les données que requiert cet algorithme.
Un contexte transmet les requêtes de ses clients à sa stratégie.

Un client crée un objet StrategieConcrete et le passe au contexte. Par la suite le client n'interagit qu'avec le contexte.

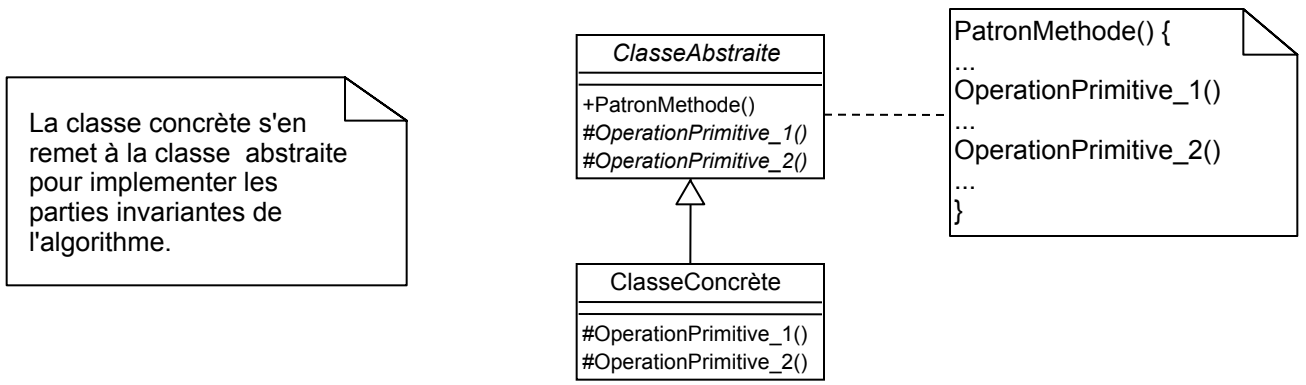


Template Method (Patron de Méthode)

- Intention :

  Définit dans une opération le squelette d'un algorithme, en en déléguant certaines étapes à des sous-classes. Ce pattern permet de redéfinir par des sous-classes, certaines parties d'un algorithme, sans avoir à modifier la structure de ce dernier.

- Utilisations :
  - Pour implanter une partie invariante d'un algorithme
  - Pour partager des comportements communs d'une hiérarchie de classes
  - Pour contrôler des extensions de sous-classe
- Structure :



Visitor (Visiteur)

- Intention :

  Le visiteur fait la représentation d'une opération applicable aux éléments d'une structure d'objet. Il permet de définir une nouvelle opération, sans qu'il soit nécessaire de modifier des éléments sur lesquels elle agit.

- Utilisations :
  - Une structure d'objets contient de nombreuses classes avec des interfaces différentes et on veut appliquer des opérations diverses sur ces objets.
  - Les structures sont assez stables et les opérations sur les objets évolutives.
- Structure :