Patrons de conception
(Design Patterns)

Xavier Crégut
<Prenom.Nom@enseeiht.fr>

Département Télécommunications & Réseaux
ENSEEIHT

Sommaire

Sommaire

Pourquoi les patrons ?

- Historique
- Motivation
- Définition
- Rubriques de la description
- Classification
- Utilisation

Historique

- Notion de « patron » d'abord apparue en architecture :
  - Christopher Alexander : « The Timeless Way of Constructing », 1979
  - Définition de « patrons » pour :
    - l'architecture des bâtiments
    - la conception des villes et de leur environnement
  - Christopher Alexander :
    « Chaque modèle [patron] décrit un **problème qui se manifeste constamment** dans notre environnement, et donc **décrit le cœur de la solution** de ce problème, d'une façon telle que l'on peut **réutiliser cette solution** des millions de fois, sans **jamais le faire deux fois de la même manière**. »
- Idée : appliquer la notion de patron à du logiciel : « design patterns »
  - premiers patrons à partir de 1987 (partie de la thèse de Erich Gamma)
  - puis Richard Helm, John Vlissides et Ralph Johnson (« Gang of Four, GoF »)
  - 1er catalogue en 1993 : *Elements of Reusable Object-Oriented Software*
- Vocabulaire :
  - design patterns
  - modèles de conception
  - patrons de conception
  - micro-architectures

Pourquoi les patrons ?

- Historique
- Motivation
- Définition
- Rubriques de la description
- Classification
- Utilisation

Intérêt des patrons de conception

Pourquoi définir des patrons de conception

- Construire des systèmes plus extensibles, plus robustes au changement
- Capitaliser l'*expérience collective* des informaticiens
- Réutiliser les solutions qui ont fait leur preuve
- Identifier les avantages/inconvénients/limites de ces solutions
- Savoir quand les appliquer

Complémentaire avec les API

- une API propose des solutions directement utilisables
- un patron explique comment structurer son application, une API

Patron de conception dans le cycle de développement

- intervient en conception détaillée
- reste indépendant du langage d'implantation

Pourquoi les patrons ?

- Historique
- Motivation
- Définition
- Rubriques de la description
- Classification
- Utilisation

Qu'est ce qu'un patron de conception ?

- **Définition :** Un *patron de conception* (*design pattern*) décrit une structure commune et répétitive de composants en interaction (la **solution**) qui résout un **problème de conception** dans un **contexte particulier**.
- **Quatre éléments essentiels**
  - **nom :** un ou deux mots pour décrire le problème de conception considérée, ses solutions et leurs conséquences.
  - **problème :** situation où le problème s'applique
  - **solution :** éléments de la conception, leurs relations et collaborations.
    - la solution n'est pas forcément précise : idée d'architecture.
    - plusieurs variantes peuvent être possibles.
  - **conséquences :** effets résultants et compromis induits
    - Les conséquences peuvent être positives ou négatives (arbitrage).
- Un **bon patron de conception** :
  - résout un problème
  - correspond à une solution éprouvée
  - favorise la réutilisabilité, l'extensibilité, etc.
  - inclut une composante subjective : utilité, esthétique, etc.

Pourquoi les patrons ?

- Historique
- Motivation
- Définition
- Rubriques de la description
- Classification
- Utilisation

Description d'un patron de conception

Rubriques utilisées

- Nom
- Intention
- Alias
- Motivation
- Indications d'utilisation
- Structure
- Constituants
- Collaborations
- Conséquences
- Implantation
- Exemples de code
- Utilisations remarquables
- Patrons apparentés

Description d'un patron de conception

Identification et compréhension du patron

- **Nom :** nom de référence du patron
  - étend le vocabulaire du concepteur
- **Intention :** courte description de :
  - ce que fait le patron de conception ;
  - sa raison d'être ou son but ;
  - cas ou problème particulier de conception concerné.
- **Alias :** Autres noms connus pour le patron
- **Motivation :**
  - scénario qui illustre un cas de conception
  - montre l'architecture en classes et objets de la solution
  - aide à comprendre les descriptions plus abstraites du modèle
- **Indications d'utilisation :**
  - Quels sont les cas qui justifient l'utilisation du patron ?
  - Quelles situations de conception peuvent tirer avantage du patron ?
  - Comment reconnaître ces situations ?

Description d'un patron de conception

Description de la solution proposée par le patron

- **Structure :** description de la solution sous forme de :
  - un diagramme de classe pour l'architecture ;
  - des diagrammes d'interaction pour la dynamique.
- **Constituants :** classes/objets de la solution avec leurs responsabilités
- **Collaborations** entre les constituants pour assumer leurs responsabilités
- **Conséquences :**
  - compromis induits par l'utilisation du patron
  - impacts sur l'architecture de conception
  - gains en terme de diminution du couplage dans la solution
- **Implantation :** Solutions types, techniques, pièges et astuces.
- **Exemples de code :** extraits de code illustrant la mise en œuvre du patron
- **Utilisations remarquables :** exemples issus de systèmes existants
- **Patrons apparentés :**
  - patrons similaires et différences essentielles
  - utilisation conjointe avec d'autres patrons

Pourquoi les patrons ?

- Historique
- Motivation
- Définition
- Rubriques de la description
- Classification
- Utilisation

Classification des patrons de conception

Fondamental	Créateur	Structurel	Comportement
	Fabrique	Adaptateur	Interprète
Délégation	Fabrique abstraite	Adaptateur	Patron de méthode
Interface	Monteur	Pont	Chaîne de responsabilités
Classe abstraite	Prototype	Composite	Commande
Immuable	Singleton	Décorateur	Itérateur
Interface de marquage		Façade	Médiateur
		Poids mouche	Mémento
		Procuration	Observateur
			État
			Stratégie
			Visiteur

- **fondamental :** application directe des concepts objets ;-)
- **créateur :** processus de création d'objets
- **structurel :** architecture statique du système
- **comportemental :** interactions entre objets et répartition des responsabilités

Pourquoi les patrons ?

- Historique
- Motivation
- Définition
- Rubriques de la description
- Classification
- Utilisation

Choix d'un patron de conception

- Est-il une solution au problème ?
- Quels sont ses buts ?
- Quelles sont les relations avec les autres patrons de conception ?
- Est-ce que d'autres patrons jouent le même rôle ?

Mise en œuvre d'un patron de conception

- Lire complètement la description, en particulier les sections *indications d'utilisation* et *conséquences*.
- Étudier en détail les sections *Structure*, *Constituants* et *Collaborations*
- Regarder la section *Exemple de code*
- Choisir des noms de constituants ayant un sens dans le contexte d'utilisation !

Ce qu'il ne faut pas attendre des patrons de conception

- Une solution universelle prête à l'emploi
- Une bibliothèque de classes réutilisables
- L'automatisation totale de l'instanciation d'un patron de conception
- La disparition du facteur humain

Sommaire

Patrons fondamentaux

- **Remarque :** Les patrons fondamentaux découlent souvent directement des concepts présents dans les langages objets.
  $\Longrightarrow$ Suivant les auteurs, ils sont souvent considérés comme implicites
- **Intérêt :**
  - Donnent un autre éclairage aux concepts objet
  - Sont utilisés par les autres patrons

Patrons fondamentaux

- Délégation
- Interface
- Classe abstraite
- Interface et Classe abstraite
- Immuable
- Interface de marquage

Délégation

- **Contexte :** Parfois, utiliser l'héritage conduit à une mauvaise conception. La délégation est alors un mécanisme plus général (même si plus lourd).
- **Cas où ne pas utiliser l'héritage (mais la délégation) :**
  - *Une classe métier veut réutiliser une classe « utilitaire ».*
    - ★ Évolutions de l'« utilitaire » compatibles avec la classe « métier » ?
  - *Une classe veut cacher des éléments de la superclasse* (Anti-patron !).
    - ★ C'est impossible à faire (si héritage implique sous-typage, ex. Java) ! Exemple : Pile ne doit pas hériter de ArrayList.
  - *Les sous-classes modélisent des rôles différents de la superclasse.*
    Exemple : Modéliser un joueur de foot.
    Spécialiser Joueur en Gardien, Stoppeur, Attaquant... ou définir une délégation sur Poste qui est spécialisée en Gardien, Stoppeur, Attaquant...
    Un joueur peut changer de poste, voire occuper plusieurs postes.

Remarque : L'héritage correspond à « est une sorte de » mais n'est pas adapté pour représenter « est un rôle joué par » car, sinon, le rôle ne peut pas changer pendant la durée de vie de l'objet.

Patrons fondamentaux

- Délégation
- Interface
- Classe abstraite
- Interface et Classe abstraite
- Immuable
- Interface de marquage

Interface

- **Intention :** On souhaite qu'un client reste indépendant de la classe qui fournit effectivement le service.
- C'est justement l'objectif des interfaces ! (patron fondamental)
- **Exemple :** Un client veut accéder à un fournisseur JDBC pour accéder à une base de données. Le fournisseur effectif dépendra de la BD considérée. Le code client n'a pas à le connaître, seulement les interfaces qui définissent les services JDBC.
- **Intérêt :** Le fournisseur de service peut changer sans impact sur le code client.

Patrons fondamentaux

- Délégation
- Interface
- Classe abstraite
- Interface et Classe abstraite
- Immuable
- Interface de marquage

Classe abstraite

- **Intention :**
  - Garantir que la logique commune à plusieurs classes est implantée de manière cohérente pour chaque classe
  - Éviter le code redondant (et les efforts de maintenance associés)
  - Faciliter l'écriture de nouvelles classes implantant la même logique
- **Solution :**
  - Utiliser une classe abstraite pour factoriser le code commun implantant la logique
  - Les méthodes peuvent éventuellement être définies comme **final**
- **Exemple :** La classe abstraite java.util.AbstractCollection définit toutes les opérations de Collection sauf size et iterator.
  - Collection concrète non modifiable : définir seulement size et iterator
  - Collection concrète modifiable : définir aussi add (et remove sur l'iterator).

Exemple 2 : La classe TestCase de JUnit (cf *Patron de méthode*, T. 109)

Patrons fondamentaux

- Délégation
- Interface
- Classe abstraite
- Interface et Classe abstraite
- Immuable
- Interface de marquage

Interface et classe abstraite

Idée :

- Pourquoi choisir entre Interface et Classe abstraite ?
- Faire les deux !

Intérêts :

- on cumule les avantages des deux patrons
- le client (surtout en Java) est libre de choisir entre :
  - réaliser l'interface (et dupliquer le code mais hériter d'une autre classe)
  - hériter de la classe abstraite (récupérer le code) mais sans pouvoir hériter d'une autre classe

Exemple : L'API des collections en Java :

- Collection et AbstractCollection,
- List et AbstractList, etc.

Patrons fondamentaux

- Délégation
- Interface
- Classe abstraite
- Interface et Classe abstraite
- Immuable
- Interface de marquage

Immuable

- **Intention :**
  - Augmenter la robustesse des objets partageant les mêmes objets
  - Diminuer le coût de la gestion du parallélisme.
- **Intérêts :**
  - facilite la gestion des objets partagés (personne ne peut les modifier !)
  - permet de réaliser une composition sans avoir à faire de copie des objets
  - pas de besoin de synchonisation (lecture seule)
  - MAIS toute « modification » nécessite la création d'un nouvel objet
- **Exemple :**
  - String (immuable) par opposition à StringBuffer (modifiable).

Patrons fondamentaux

- Délégation
- Interface
- Classe abstraite
- Interface et Classe abstraite
- Immuable
- Interface de marquage

Interface de marquage

- **Intention :** Définir une propriété sémantique booléenne sur une classe
- **Solution :** Hériter d'une interface de marquage (interface vide).
- **Exemples :**
  - Cloneable indique que les objets d'une classe peuvent être clonés.
  - Serializable indique qu'un objet peut être sérialisé[1]
- **Forces :**
  - Tester une propriété d'un objet/classe sans connaître la classe réelle

```
1  Object copie = null;
2  if (o instanceof Cloneable) {
3      copie = o.clone();
4  } else {
5      throw new CloneNotSupportedException();
6  }
```

  - MAIS La propriété ne peut jamais être supprimée/désactivée (obligatoirement héritée par les sous-classes)

[1] sérialiser : encoder l'information en mémoire sous forme d'octets ou autres pour la sauvegarder (persistance), la transporter à travers un réseau, etc.

Sommaire

Patrons créateurs

But : Définir un mécanisme de création de classes ou d'objets.
Caractéristiques clés :

- le patron encapsule (masque) la connaissance des classes concrètes
- le patron masque comment leurs instances sont créées et combinées.

Conséquence : Un modèle créateur offre une grande liberté concernant :

- quels objets doivent être créés
- qui doit les créer
- comment les créer
- quand les créer

Patrons créateurs

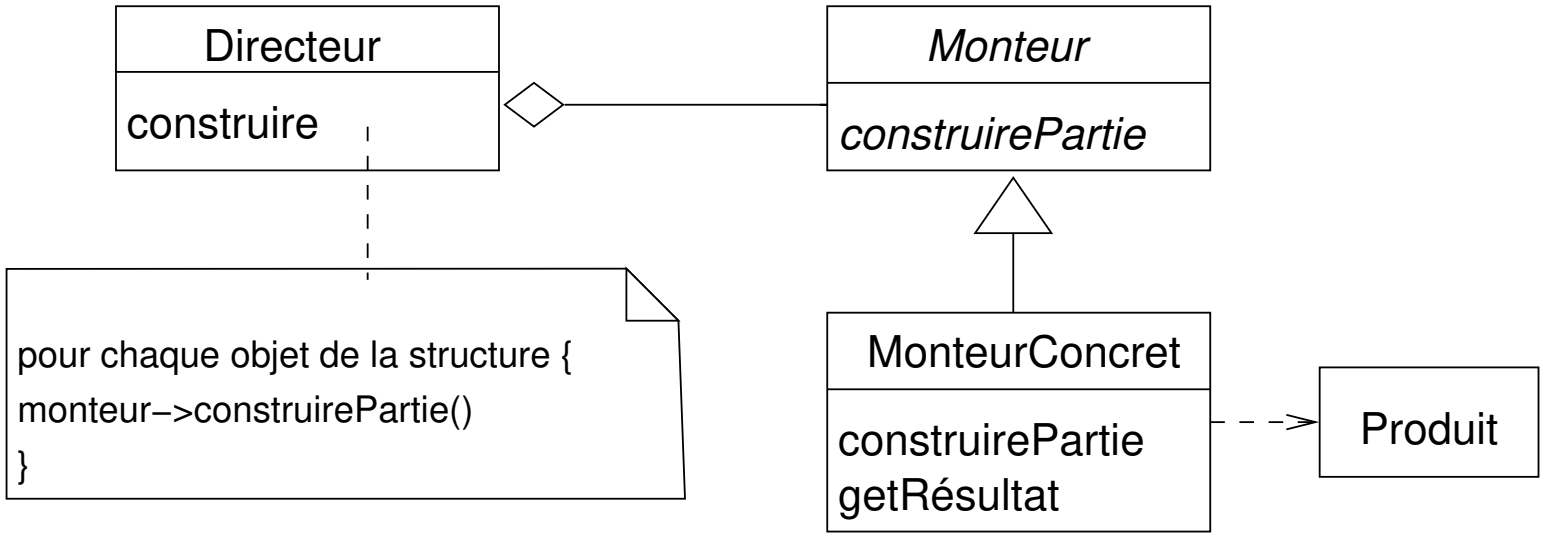
- Monteur
- Fabrique
- Fabrique abstraite
- Prototype
- Singleton

Monteur (Builder)

- **Intention**
  Dissocier la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction permette des représentations différentes.
- **Alias :** —
- **Indications d'utilisation**
  - 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
- **Exemple :**
  - Créer un message électronique à partir du destinataire, de l'expéditeur, de l'objet, du texte, des fichiers attachés, etc.

Monteur (Builder)

Diagramme de classe



Exemple : Créer un message électronique (Produit) en précisant le destinaire, l'objet, le texte, les fichiers attachés, etc. (construire les parties).

MessageBuilder
setTo(String) *setObject(String)* ... *getMessage(): Message*

Patrons créateurs

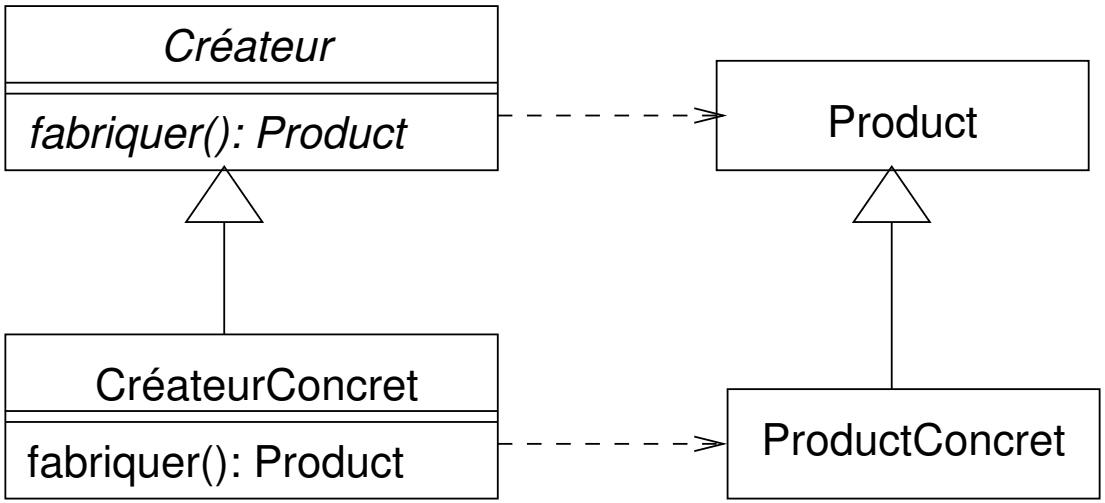
- Monteur
- Fabrique
- Fabrique abstraite
- Prototype
- Singleton

Fabrique (Factory method)

- **Intention**
  - définir une interface pour la création d'un objet mais en laissant à des sous-classes le choix des classes à instancier.
  - permettre à une classe de déléguer la création d'objets à des sous-classes.
- **Alias :** Fabrication, Constructeur polymorphe (ou virtuel)
- **Indications d'utilisation**
  - une classe ne peut pas prévoir les classes des objets qu'elle aura à créer
  - une classe attend de ses sous-classes qu'elles spécifient les objets qu'elles créent
- **Exemples :**
  - Créer un document qui peut être texte, dessin ou présentation en fonction du nom du type de document.
    On veut pouvoir ajouter de nouveaux types de documents.
  - Obtenir un manipulateur sur des objets graphiques.
- **Conséquence :**
  - Il procure un gîte pour les sous-classes (objets élargis)
  - Il inter-connecte des hiérarchies parallèles de classes.

Fabrique (Factory method)

Diagramme de classe



Exemple : Créer une pizza en fonction de son nom : régina, calzone, savoyarde...

Autre exemple : La méthode iterator() de Iterable.

Patrons créateurs

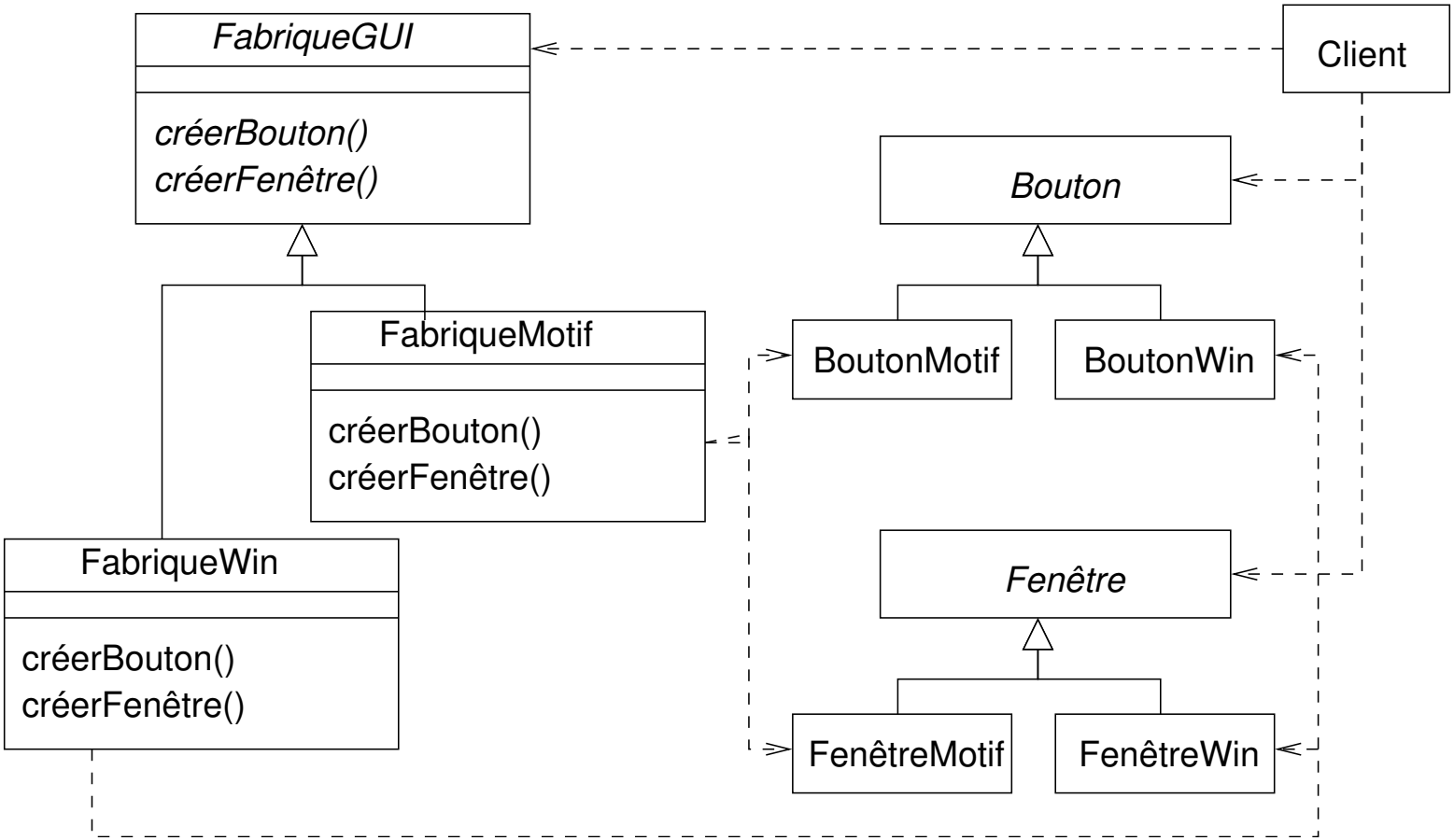
- Monteur
- Fabrique
- Fabrique abstraite
- Prototype
- Singleton

Fabrique abstraite (*Abstract Factory*)

- **Intention**
  Fournir une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes.
- **Alias :** Kit
- **Indications d'utilisation**
  - un système doit être indépendant de la manière dont ses produits ont été construits, combinés ou représentés
  - un système est composé à partir d'une famille de produits, parmi plusieurs
  - on souhaite renforcer le caractère de communauté d'une famille de produits conçus pour être utilisés ensemble
- **Exemples :** boîte à outils proposant plusieurs « look-and-feel » (décors).
- **Solution (principe) :** définir une interface Fabrique qui déclare les méthodes créerAscenceur, créerFenêtre, etc.

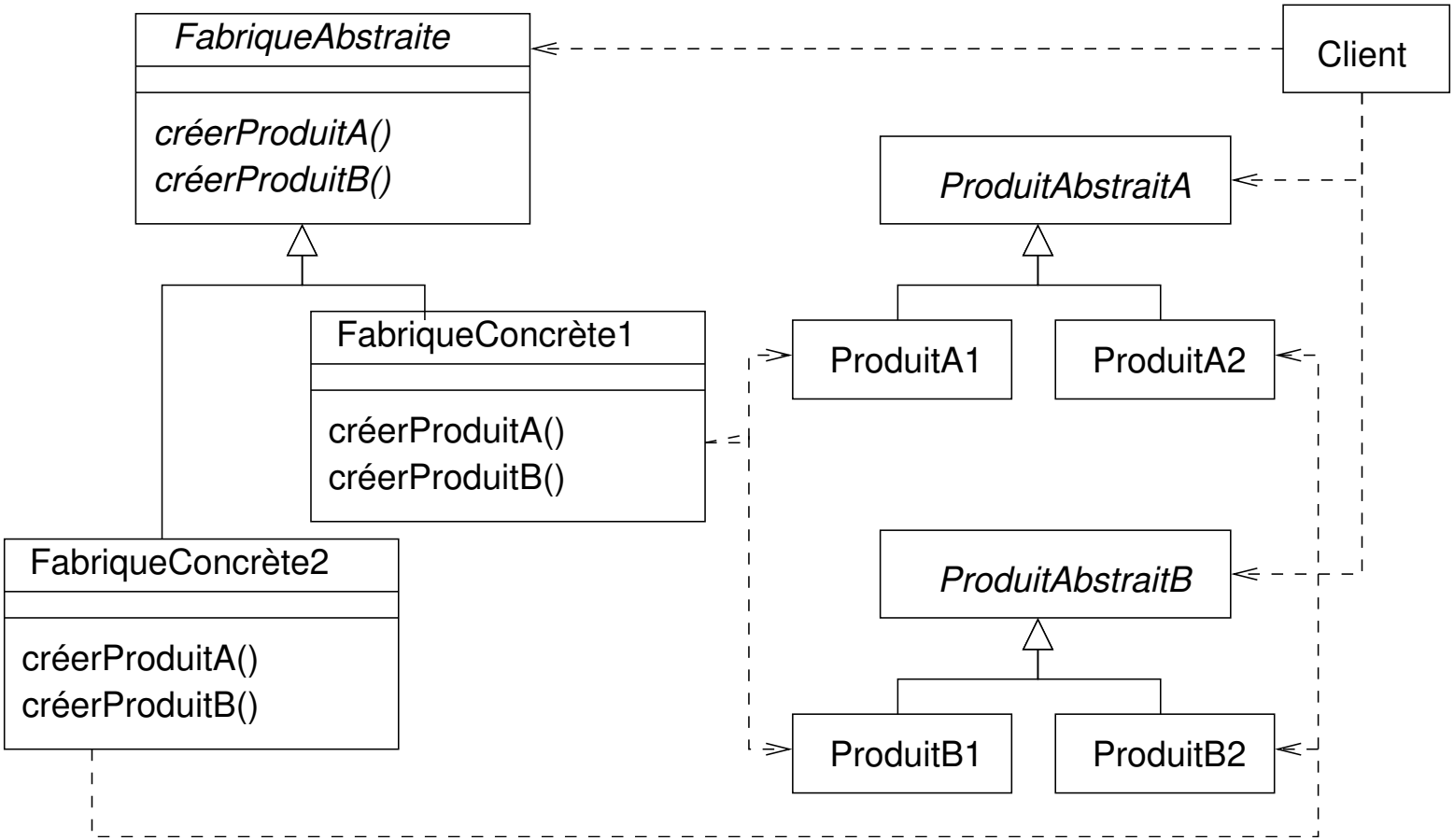
Fabrique abstraite (*Abstract Factory*)

Exemple : toolkit pour interfaces graphiques utilisateur (GUI)



Fabrique abstraite (*Abstract Factory*)

Diagramme de classe



Patrons créateurs

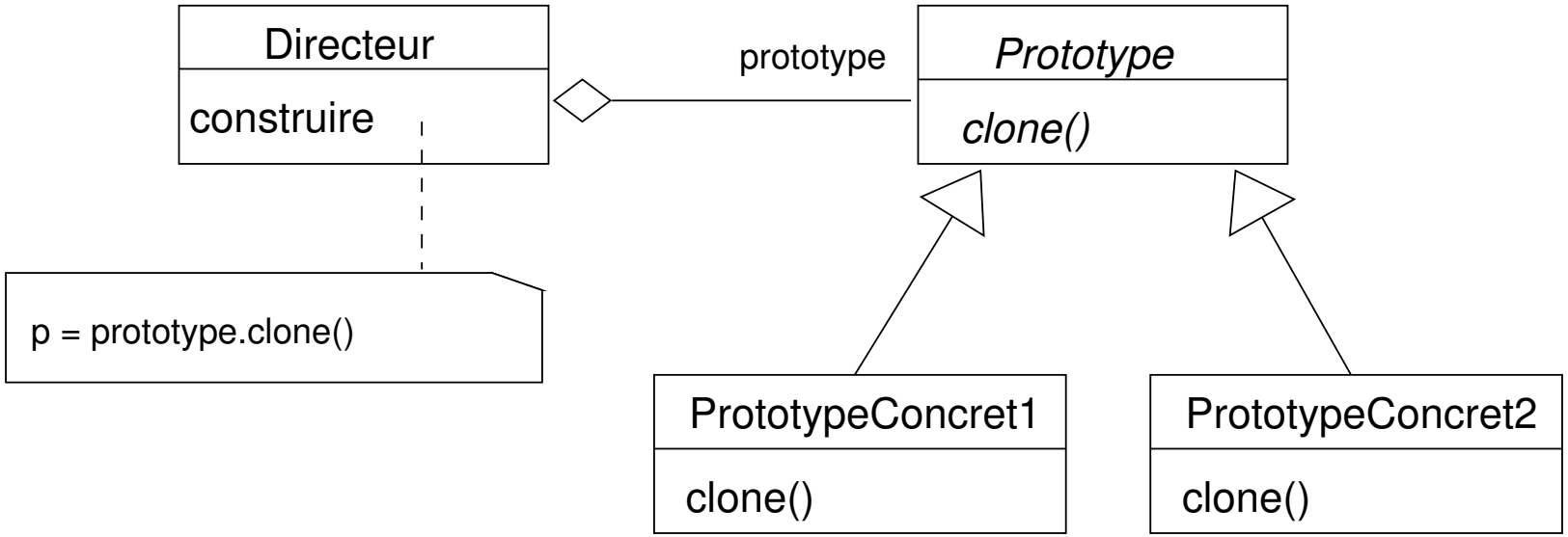
- Monteur
- Fabrique
- Fabrique abstraite
- Prototype
- Singleton

Prototype

- **Intention**
  Spécifie le type des objets à créer par une instance (le prototype) et crée de nouveaux objets en copiant ce prototype (clonage).
- **Alias :** —
- **Indications d'utilisation**
  - classes à instancier spécifiées à l'exécution (p.ex. chargement dynamique).
  - éviter de construire une hiérachie de classes Fabrique
  - classes pouvant prendre un nombre réduit d'états (un prototype par état)
- **Exemples :**
  - Éditeur de musique : les notes sont disponibles et copiées avant utilisation
  - Éditeur UML : les éléments apparaissent dans un menu et sont copiés pour les ajouter sur le dessin (puis les adapter).

Prototype

Diagramme de classe



Exemple vu : Pour ajouter des commandes réversibles aux menus textuels, il ne faut pas exécuter directement la commande obtenue du menu mais une copie de cette commande (mémorisation des informations pour pouvoir annuler).

Patrons créateurs

- Monteur
- Fabrique
- Fabrique abstraite
- Prototype
- Singleton

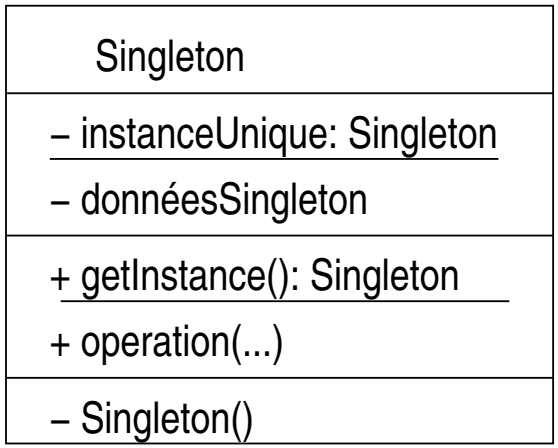
Singleton

- **Intention**
  Garantir qu'une classe n'a qu'*une seule instance* et fournir un *point d'*accès global à cette instance.
- **Alias :** —
- **Indications d'utilisation**
  - Il doit y avoir exactement une instance d'une classe
  - Cette instance doit être accessible globalement
  - Le type du singleton peut-être sous-classé

  $\Longrightarrow$ la classe elle-même est responsable de gérer l'unicité de l'objet et son accès
- **Exemples :** gestionnaire de fenêtres, file d'impression, etc.

Singleton

Diagramme de classe




```
1  /** Manière simple et traditionnelle de faire
2   * un Singleton en Java.
3   * Attention : non sûre si utilisation de threads
4   */
5  public class Singleton {
6      private final static Singleton INSTANCE =
7                      new Singleton();
8
9      // Interdire toute création d'instance
10     // depuis une classe extérieure
11     private Singleton() {}
12
13     // Accès à l'instance unique
14     public static Singleton getInstance() {
15         return INSTANCE;
16     }
17
18 }
```

Sommaire

Patrons structurels

- Définir la façon d'assembler des classes et des objets pour réaliser des structures complexes.
- Agrégat d'objets conçus comme des « macro-composants ».

Patrons présentés :

- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids mouche
- Procuration
- Discussion

Patrons structurels

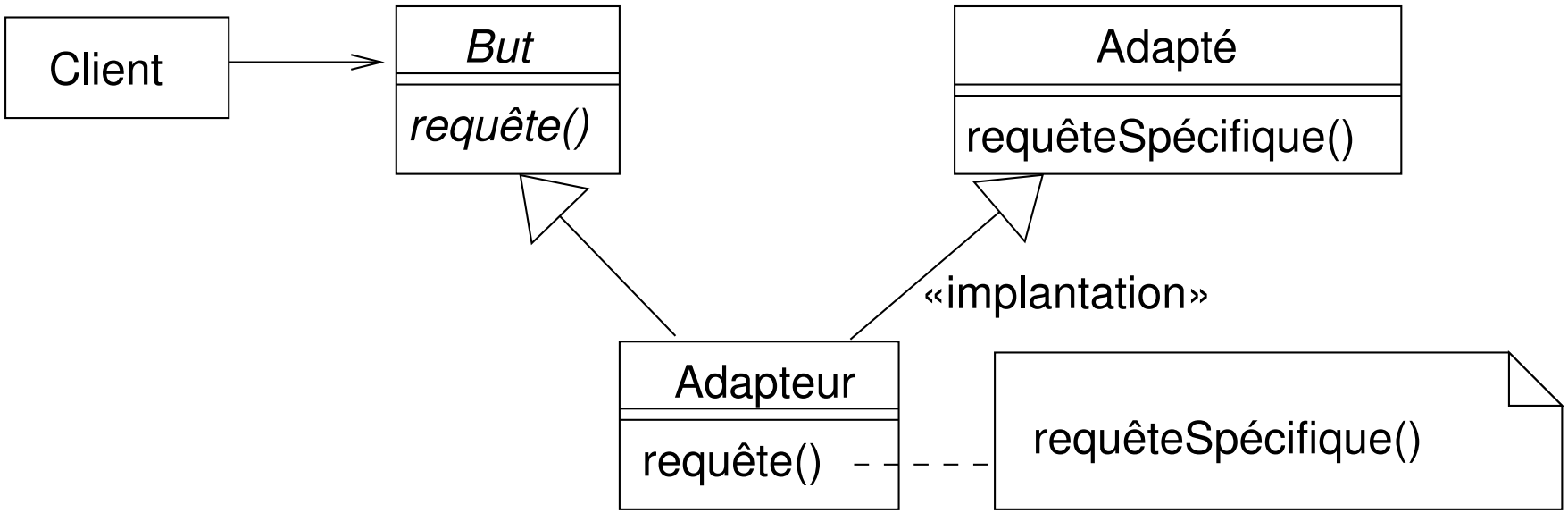
- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids mouche
- Procuration
- Discussion

Adaptateur (Adapter)

- **Intention**
  Convertit l'interface d'une classe en une autre conforme à l'attente d'un client.
  Permet de faire collaborer des classes aux interfaces incompatibles.
- **Alias :** Empaqueteur (Wrapper)
- **Indications d'utilisation**
  - On veut utiliser une classe dont l'interface ne coïncide pas avec celle escomptée
  - Prévoir l'ajout de classes non encore connues
- **Exemples :**
  - Utiliser un ArrayList pour réaliser une Pile (Adaptation d'objet).
  - Restructuration de Vector avec List (Adaptation de classe)

Adaptateur (Adapter)

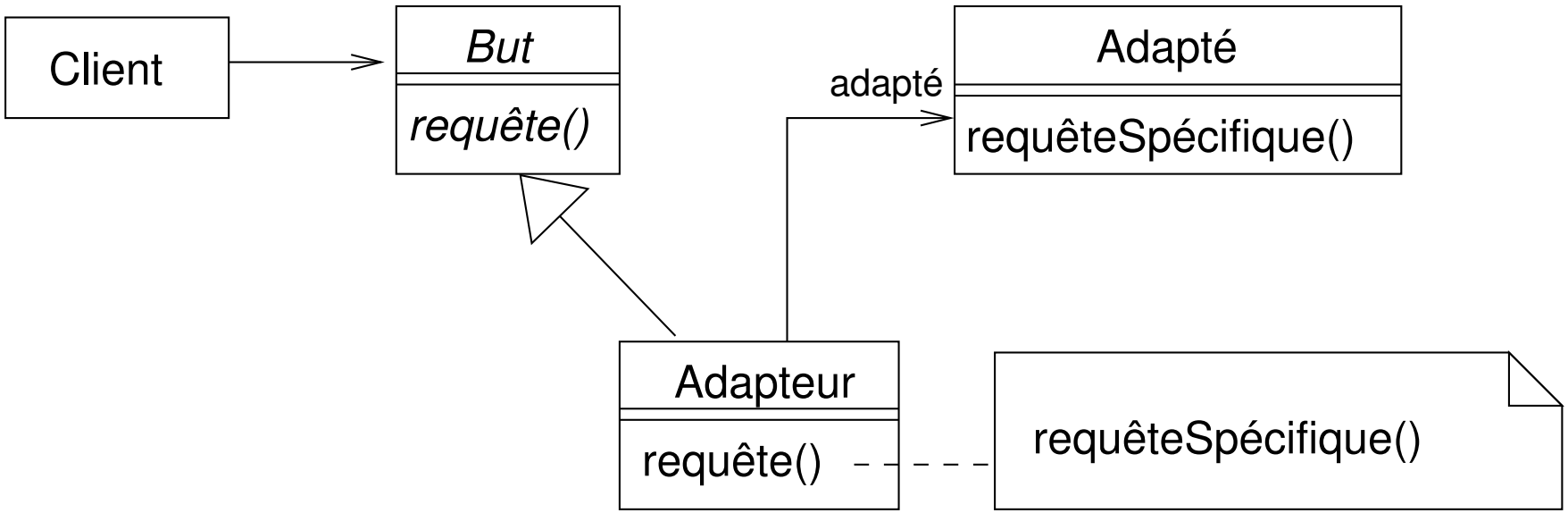
Diagramme de classe pour l'Adaptateur de Classe



Exemple : SousMenu dans l'éditeur avec menu textuel

Adaptateur (Adapter)

Diagramme de classe pour l'Adaptateur d'Objet



Exemple : CommandeGérerMenu dans l'éditeur avec menu textuel

Patrons structurels

- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids mouche
- Procuration
- Discussion

Pont (Bridge)

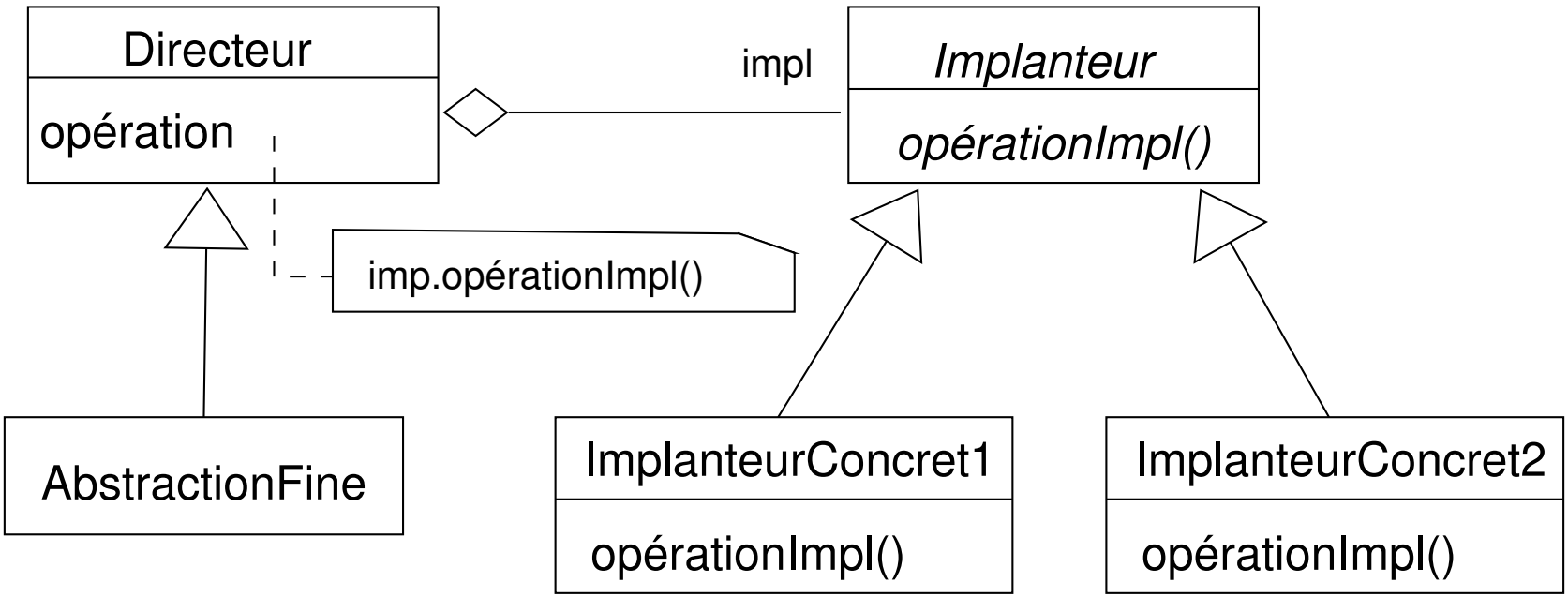
- **Intention**

  Découple une abstraction de sa réalisation afin que les deux puissent être modifiées indépendamment de l'autre

- **Alias :** Poignée/Corps (Handler/Body)
- **Indications d'utilisation**
  - éviter un lien définitif entre une abstraction et son implantation
  - permettre la spécialisation des abstractions **et** des implantations
  - un changement de l'implantation ne doit pas avoir d'impact sur les clients
  - plusieurs objets partagent la même implantation mais ceci est transparent pour les clients (compteur de références)
- **Exemples :**
  - Objets géométriques variés devant être affichés sur différents écrans.
  - L'abstraction Voiture doit pouvoir rouler sur une route (implantation). Tout type de voiture doit pouvoir rouler sur tout type de route.
- **Conséquences :**
  - découplage entre abstraction et implantation
  - capacité d'extension accrue
  - dissimulation des détails d'implantation aux clients

Pont (Bridge)

Diagramme de classe



Exemple : Dessiner (opération) les objets géométriques (abstraction) sur un afficheur (implantation).

Patrons structurels

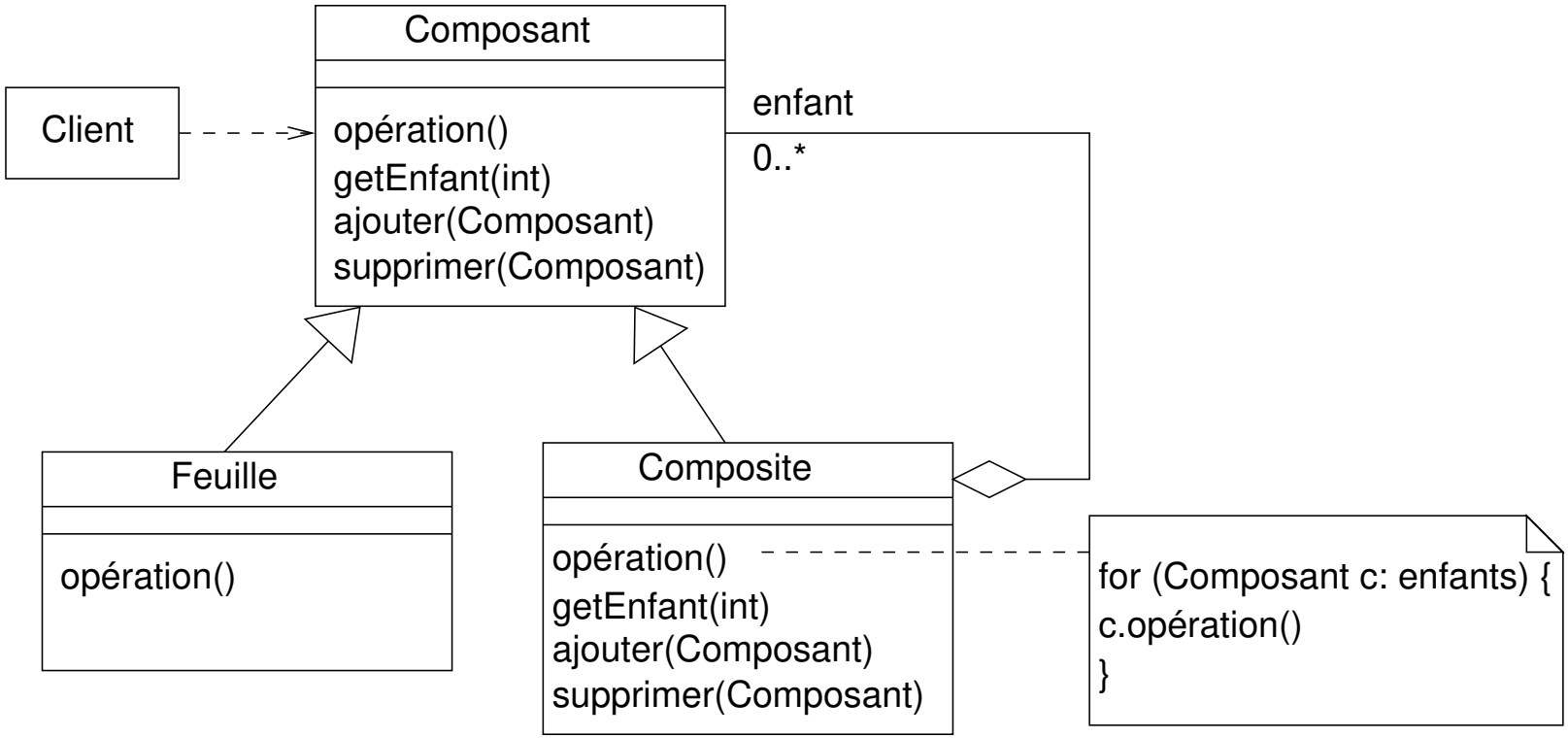
- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids mouche
- Procuration
- Discussion

Composite

- **Intention**
  - Compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé.
  - Permet au client de traiter uniformément un objet individuel ou les combinaisons de ceux-ci
- **Alias :** —
- **Indications d'utilisation**
  - représentation de structures récursives d'éléments hétérogènes
  - traitement uniforme d'un objet individuel ou d'une combinaison d'objets
- **Exemples :** Groupe dans éditeur de schémas mathématiques.
- **Conséquences :**
  - \+ facilite l'ajout de nouveaux types de composants
  - – difficile d'imposer des contraintes sur les compositions possibles

Composite

Diagramme de classe



Attention : les opérations manipulant les composés peuvent ou non être remontées au niveau du composant : compromis entre sécurité et transparence (pour le client)

Exemple : Groupe d'objets géométriques.

Patrons structurels

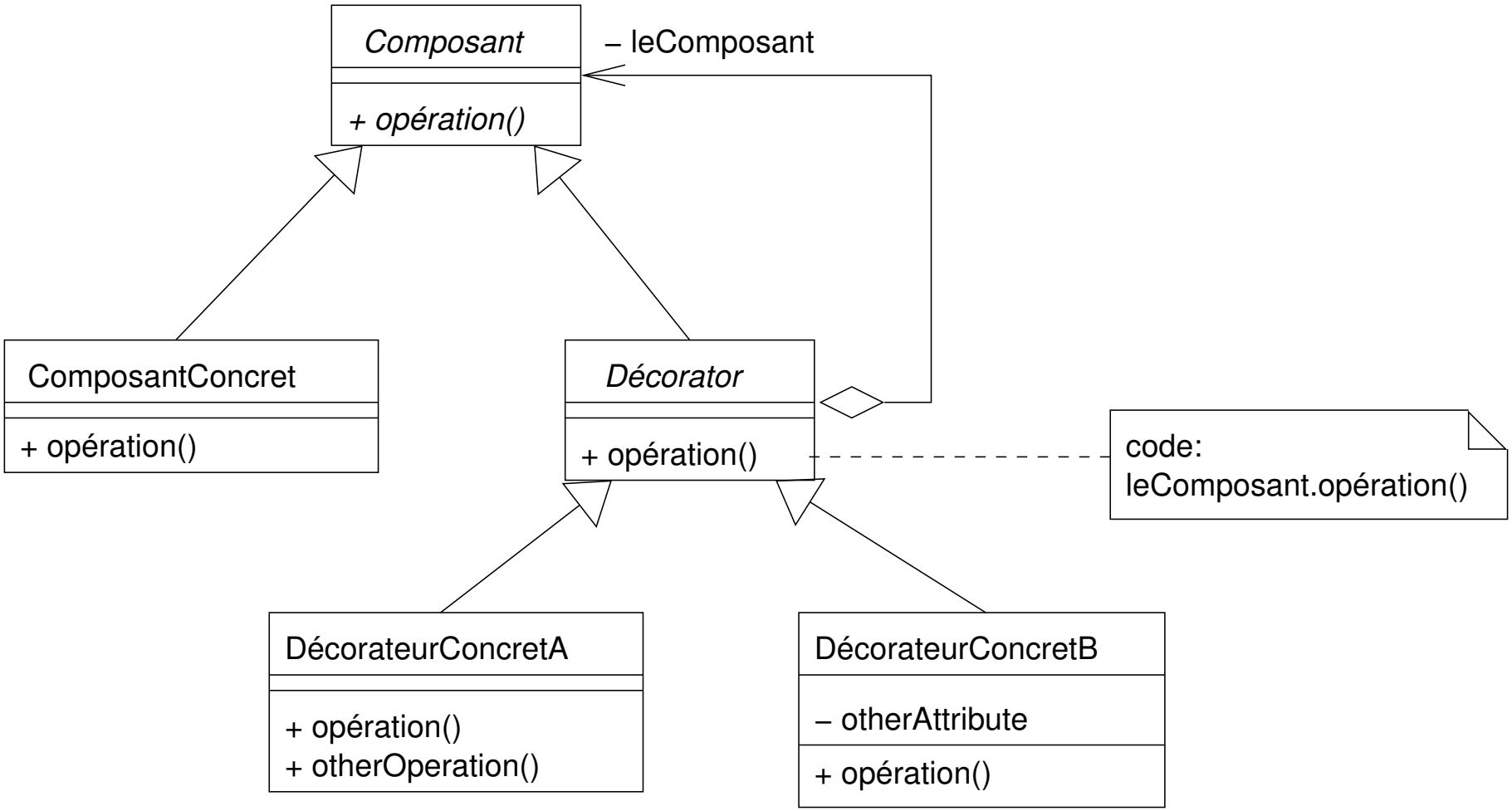
- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids mouche
- Procuration
- Discussion

Décorateur (Decorator)

- **Intention**
  - Attache dynamiquement des responsabilités supplémentaires à un objet
  - Alternative souple à l'héritage pour ajouter de nouvelles fonctionnalités
- **Alias :** Emballeur (Wrapper)
- **Indications d'utilisation**
  - ajouter dynamiquement de nouvelles fonctionnalités, de manière transparente (sans changement d'interface)
  - définir des responsabilités qui peuvent être ajoutées/retirées
  - éviter un héritage impossible à cause du trop grand nombre d'extensions *indépendantes* possibles
- **Exemples :**
  - Éléments optionnels (bord, ascenseur...) sur un composant graphique
  - Flots d'entrée/sorties en Java (aussi appelé Filtre dans ce cas).
- **Conséquences :**
  - plus de souplesse que l'héritage (qui est statique)
  - évite de surcharger en fonctionnalités les classes de haut niveau
  - composant et composant décoré n'ont pas la même identité (adresse) !
  - de nombreux petits objets !

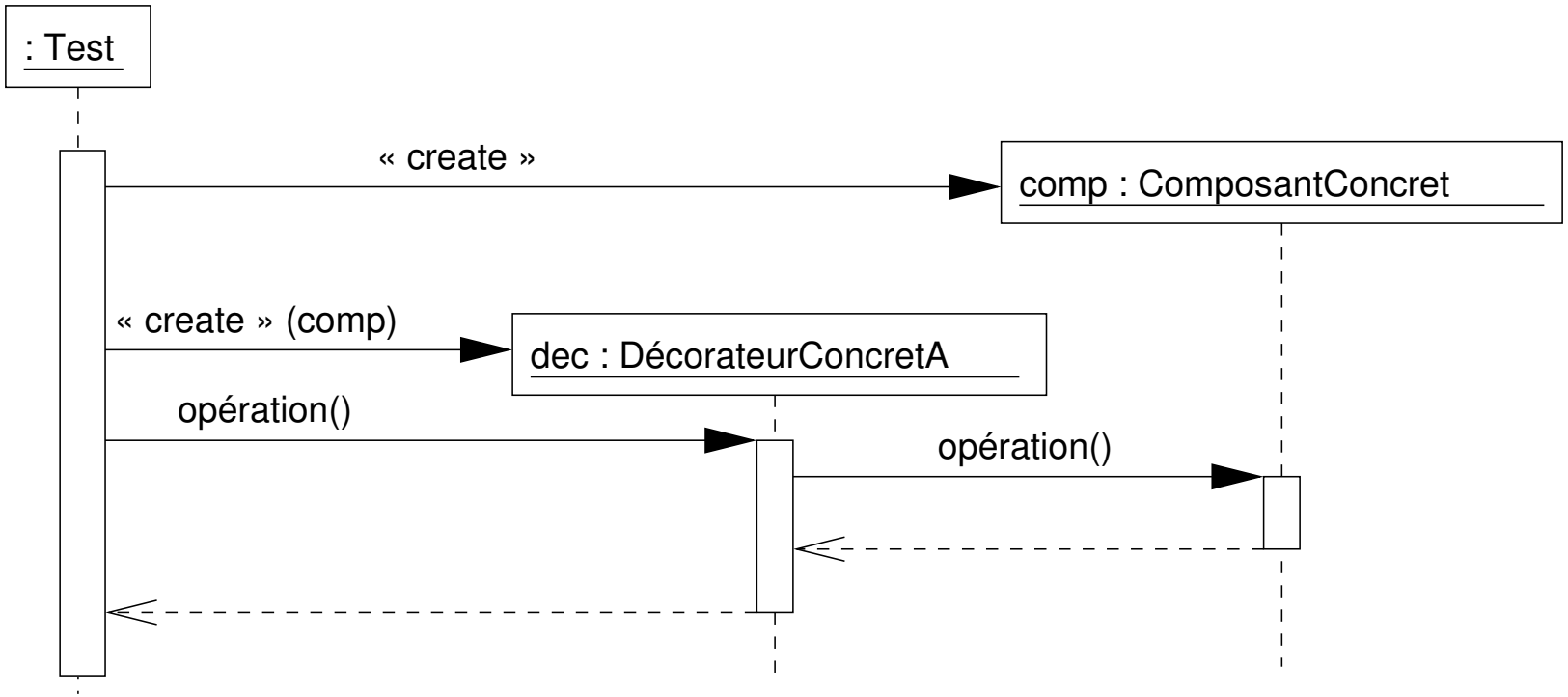
Décorateur (Decorator)

Diagramme de classe



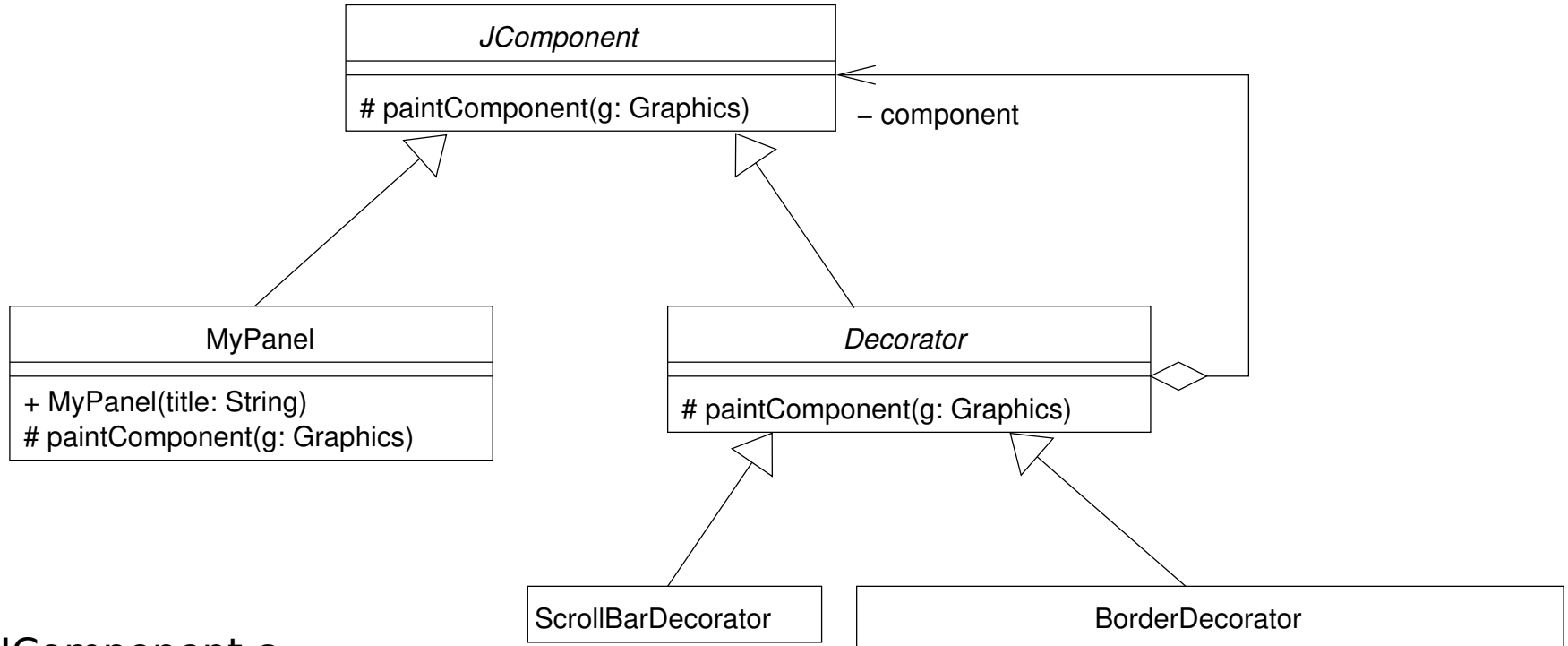
Décorateur (Decorator)

Diagramme de séquence



Décorateur (Decorator)

Exemple : élément graphique avec décorations




```
1 JComponent c =
2         new BorderDecorateur(
3             new ScrollBarDecorateur(
4                 new MyPanel()));
```

Patrons structurels

- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids mouche
- Procuration
- Discussion

Façade (Facade)

- **Intention**

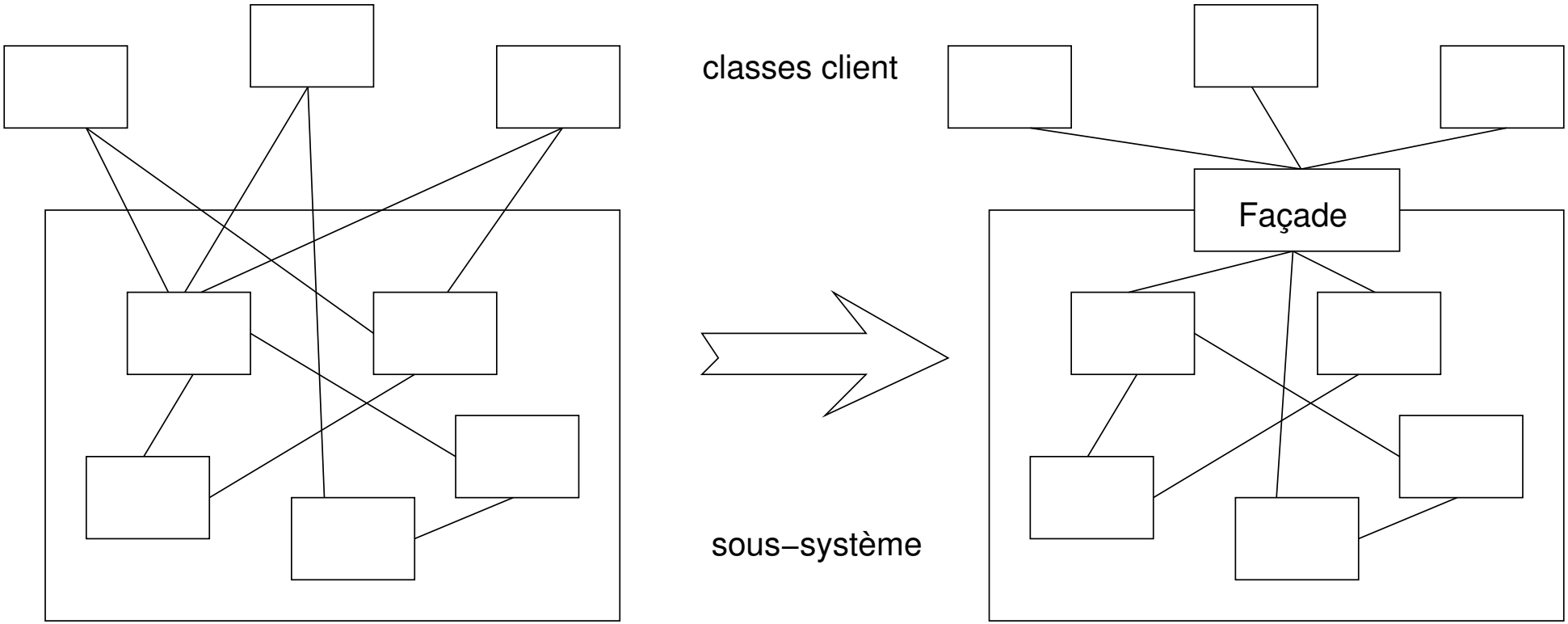
  Fournit une interface unifiée pour un sous-système, interface de haut niveau rendant le système plus facile à utiliser (couplage réduit).

- **Alias :** —
- **Indications d'utilisation**
  - On souhaite disposer d'une interface simple pour un système complexe
  - diminuer le couplage entre un sous-système et les clients
  - structuration d'un sous-système en niveaux
- **Exemple :**

  Un compilateur (compiler) qui utilise l'analyseur lexical, l'analyseur syntaxique, l'analyseur sémantique, une table des symboles, engendre du code pour plusieurs architectures, etc.

Façade (Facade)

Diagramme de classe



Patrons structurels

- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids mouche
- Procuration
- Discussion

Poids mouche (Flyweight)

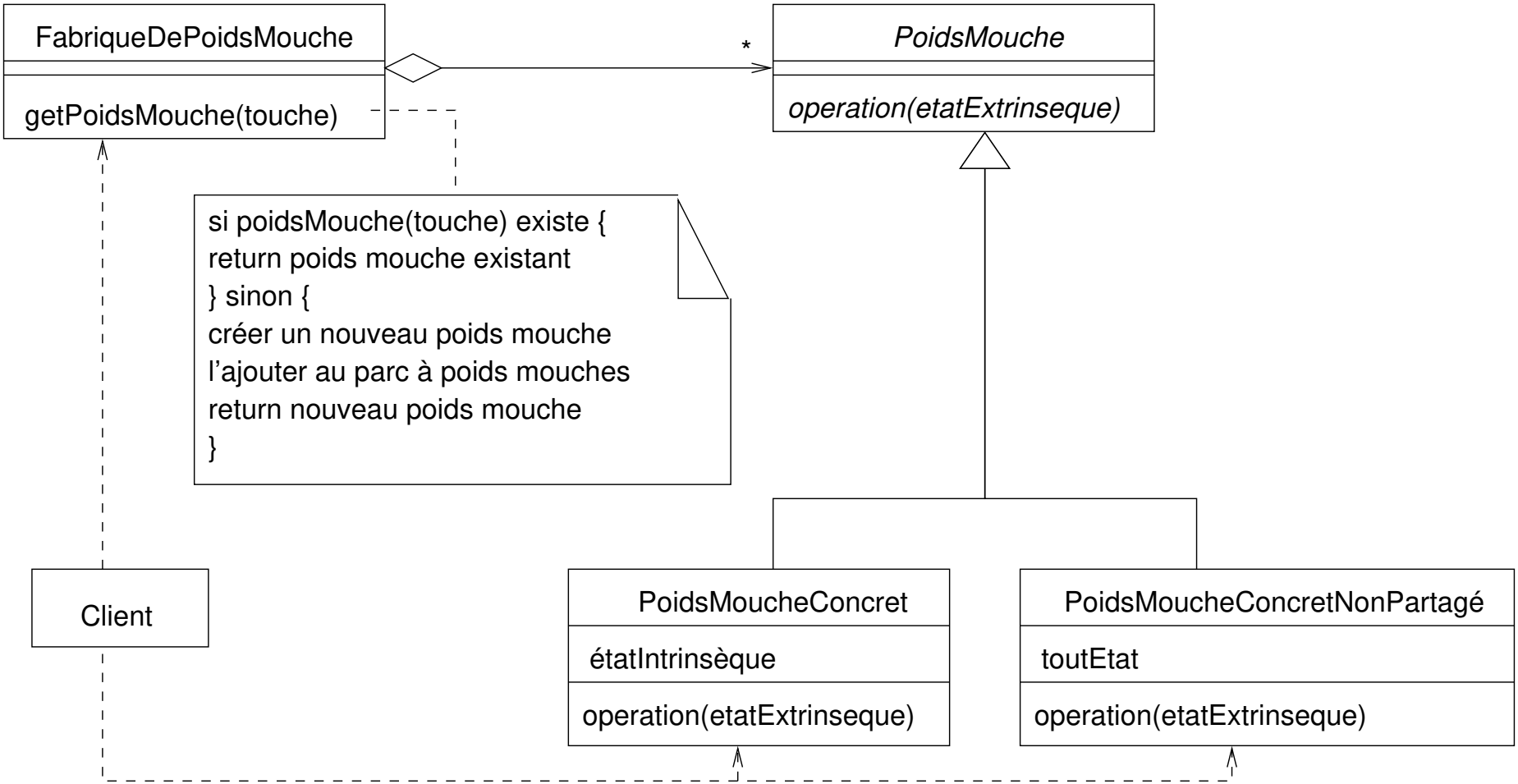
- **Intention**

  Utilise une technique de partage permettant la mise en œuvre efficace d'un grand nombre d'objets de granularité fine.

- **Alias :** —
- **Indications d'utilisation**
  - Détérioration des performances due à un trop grand nombre de petits objets (coût de stockage élevé)
  - Distinction possible entre état intrinsèque et extrinsèque (fonction du contexte client)
  - Partage possible des états intrinsèques.
- **Exemples :**
  - Analyse lexicale : *unités lexicales* gérées comme des poids mouches
    - ★ état intrinsèque : la chaîne
    - ★ état extrinsèque : ligne et colonne du premier caractère
- **Implantation :**
  - déportation des états extrinsèques
  - gestion des objets partagés

Poids mouche (Flyweight)

Diagramme de classe



Patrons structurels

- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids mouche
- Procuration
- Discussion

Procuration (Proxy)

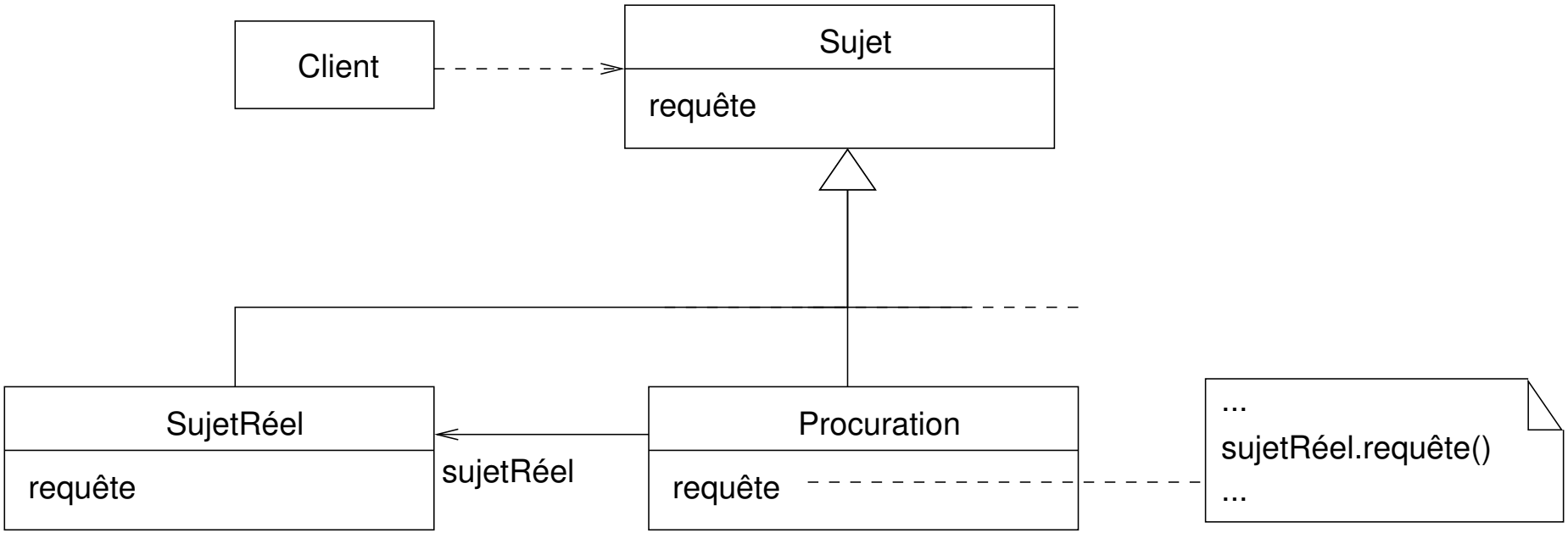
- **Intention**

  Fournit à un client un mandataire ou un remplaçant pour contrôler l'accès à un objet fournisseur

- **Alias :** Subrogé (Surrogate), Mandataire, Procuration
- **Indications d'utilisation**
  - procuration à distance : représentant local d'un objet distant
  - procuration virtuelle : création d'objet à la demande
  - procuration de protection : contrôle les accès au fournisseur
  - référence intelligente (*smart pointer*) : remplaçant d'un pointeur brut qui ajoute compteur de références, chargement en mémoire d'un objet persistant...

Procuration (Proxy)

Diagramme de classe



Patrons structurels

- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids mouche
- Procuration
- Discussion

Patrons structurels

Discussion

- Similitudes entre ces patrons mais des intentions différentes !
- Mécanismes communs :
  - héritage : pour les patrons dont le domaine est classe
  - délégation : pour les patrons dont le domaine est objet
- **Adapteur** et **Pont** : indirection et ré-expédition de requêtes entre une interface et une réalisation
- **Façade** : construction d'une nouvelle interface
- **Composite** et **Décorateur** : agrégats récursifs

Sommaire

Patrons comportementaux

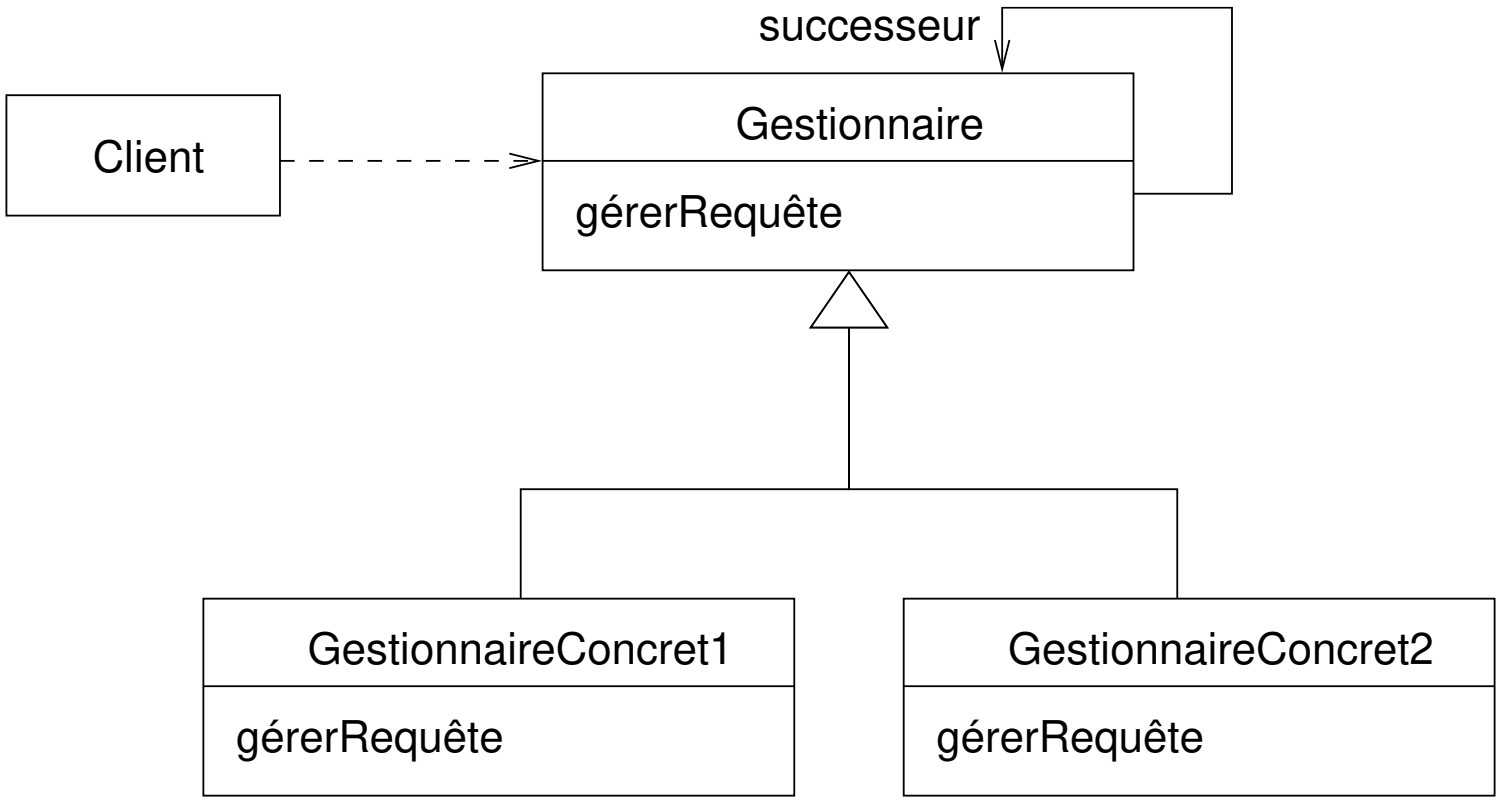
- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

Chaîne de responsabilités (Chain of Responsibility)

- **Intention**
  - Évite le couplage entre l'émetteur d'une requête et ses récepteurs
  - Chaîne les récepteurs qui se passent la requête jusqu'à ce qu'un la traite (le gestionnaire)
- **Alias :** —
- **Indications d'utilisation**
  - Une requête peut être gérée par plus d'un objet à la fois et le gestionnaire n'est pas connu *a priori* (déterminé dynamiquement)
  - On souhaite adresser une requête à plusieurs objets, sans spécifier explicitement le récepteur
  - Les objets qui traitent la demande doivent être définis dynamiquement
- **Exemples :**
  - même principe que les exceptions Java : récepteurs = **catch**
- **Conséquences :**
  - Réduction du couplage
  - Souplesse accrue dans l'attribution des responsabilités
  - Pas de garantie de traitement de la requête !

Chaîne de responsabilités (Chain of Responsibility)

Diagramme de classe



Patrons comportementaux

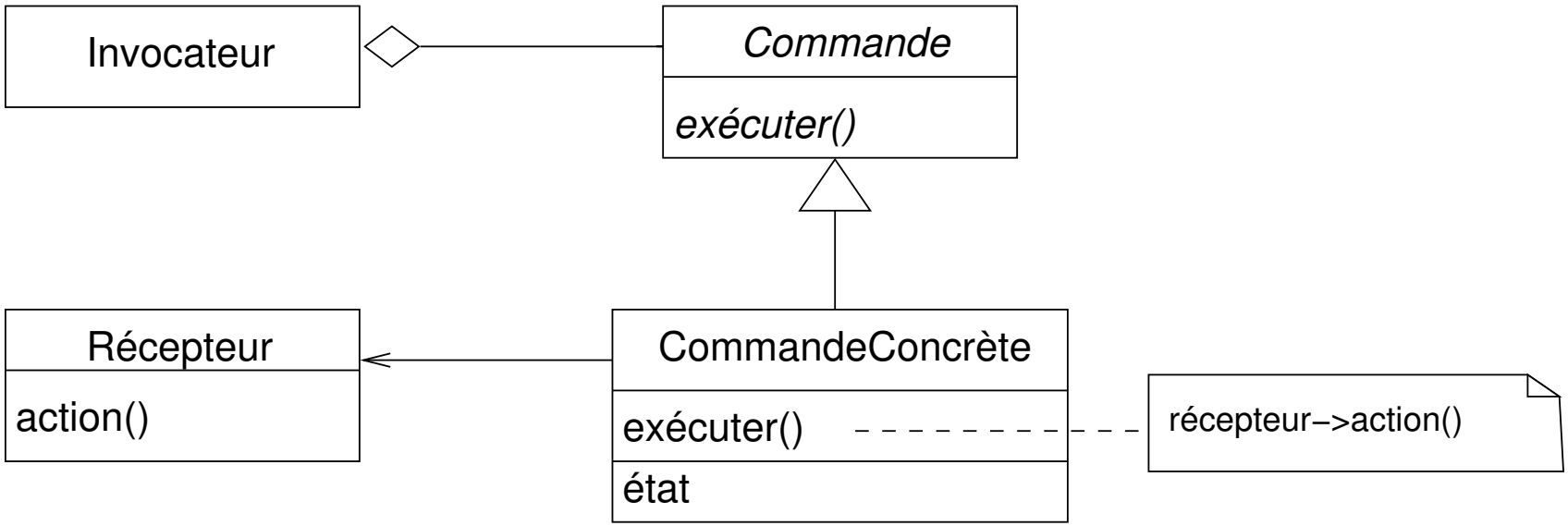
- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

Commande (Command)

- **Intention**
  - encapsuler une requête (ou un traitement) comme un objet (version objet des *callbacks*)
  - paramétrer le contexte client
  - gestion possible des requêtes/traitements en FIFO, annulation possible.
- **Alias :** Action, Transaction
- **Indications d'utilisation :**
  - réaliser un principe de *callback*
  - permettre de mémoriser des modifications
  - structurer le système en opérations de haut niveau (transaction)
  - voir aussi *Intention*

Commande (Command)

Diagramme de classe



Patrons comportementaux

- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

Interpréteur (Interpreter)

- **Intention**
  Définit une représentation de la grammaire d'un langage simple ainsi qu'un interpréteur.
- **Alias :** —
- **Indications d'utilisation**
  - la grammaire du langage doit être simple
  - l'efficacité n'est pas un souci majeur
- **Exemples :** Évaluer une expression arithmétique
- **Conséquences :**
  - les grammaires complexes sont difficiles à maintenir
    $\Longrightarrow$ utiliser des générateurs de compilateur (javacc, java_cup, antlr, etc.)

Patrons comportementaux

- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

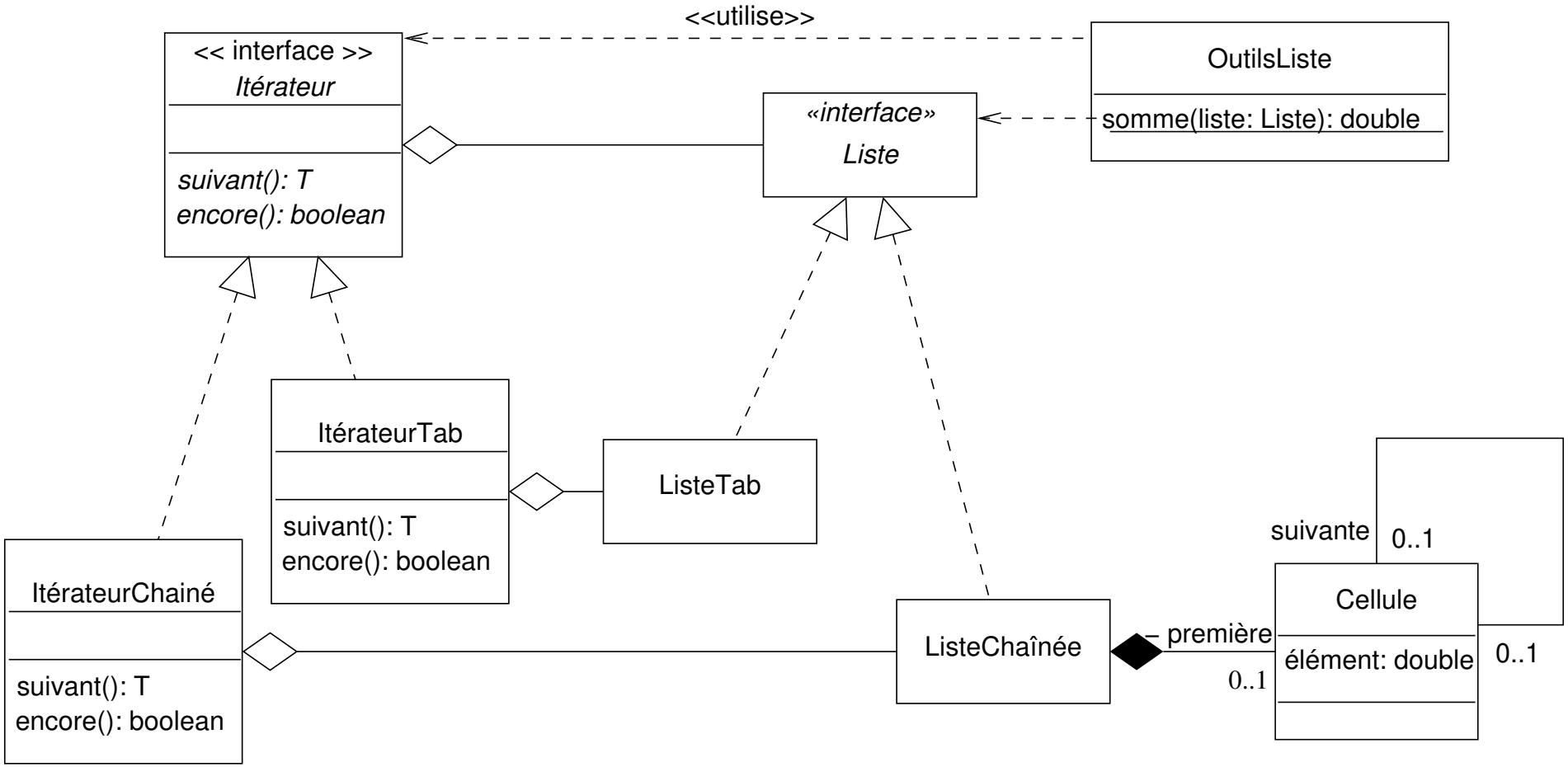
Itérateur (Iterator)

- **Intention**

  Fournit un moyen pour accéder séquentiellement à chacun des éléments d'un agrégat d'objets sans révéler la représentation interne de l'agrégat
- **Alias :** Curseur (Cursor)
- **Indications d'utilisation**
  - accéder aux éléments d'un agrégat sans révéler sa structure interne
  - gérer simultanément plusieurs parcours sur des agrégats
  - offrir une interface uniforme pour parcourir différents types d'agrégats
- **Exemples :**
  - Les itérateurs des collections Java
- **Conséquences :**
  - possibilité de définir plusieurs parcours (infixe et préfixe par exemple)
  - simplification de l'interface de l'agrégat
  - parcours simultanés possibles sur un même agrégat

Itérateur (Iterator)

Diagramme de classe



Patrons comportementaux

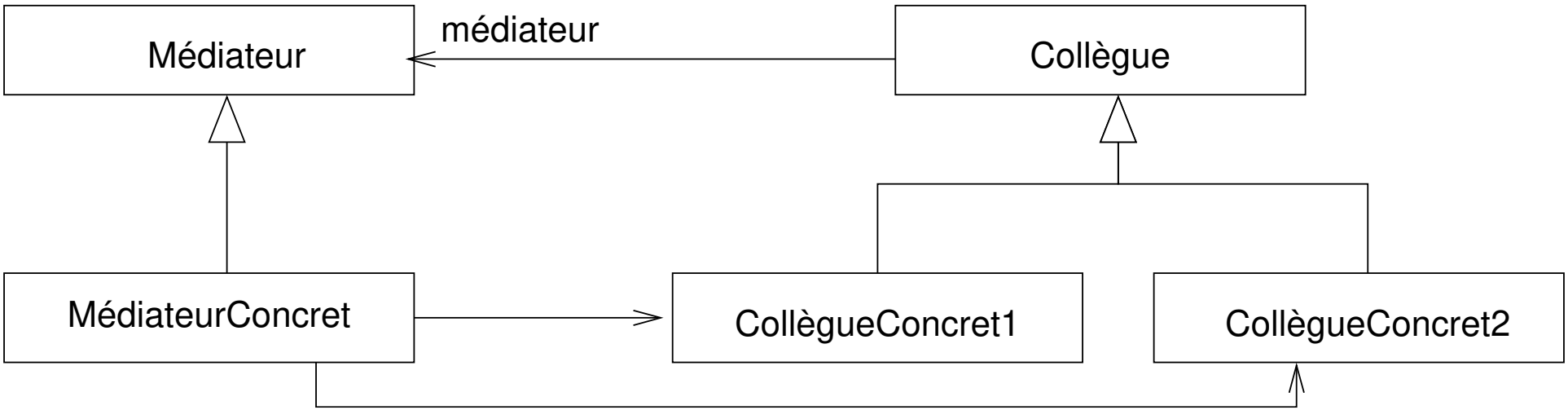
- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

Médiateur (Mediator)

- **Intention**
  - Définit un objet encapsulant les modalités d'interaction d'autres objets
  - Limite le couplage car évite une référence explicite entre ces objets
- **Alias :** —
- **Indications d'utilisation**
  - Les objets d'un ensemble communiquent d'une façon bien définie mais complexe : inter-dépendances non structurées et difficiles à appréhender
  - Réutilisation difficile d'un objet car fait référence à beaucoup d'objets
  - Un comportement distribué entre plusieurs classes doit pouvoir être spécialisé sans multiplier les sous-classes
- **Exemple :** médiateur entre les éléments graphiques d'une boîte de dialogue
- **Conséquences :**
  - limite la création de sous-classes (seulement du médiateur, pas des collègues)
  - réduit le couplage entre collègues
  - simplifie les protocoles objet (interaction 1–plusieurs)
  - formalise la coopération des objets
  - centralise le contrôle

Médiateur (Mediator)

Diagramme de classe



Patrons comportementaux

- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

Mémento (Memento)

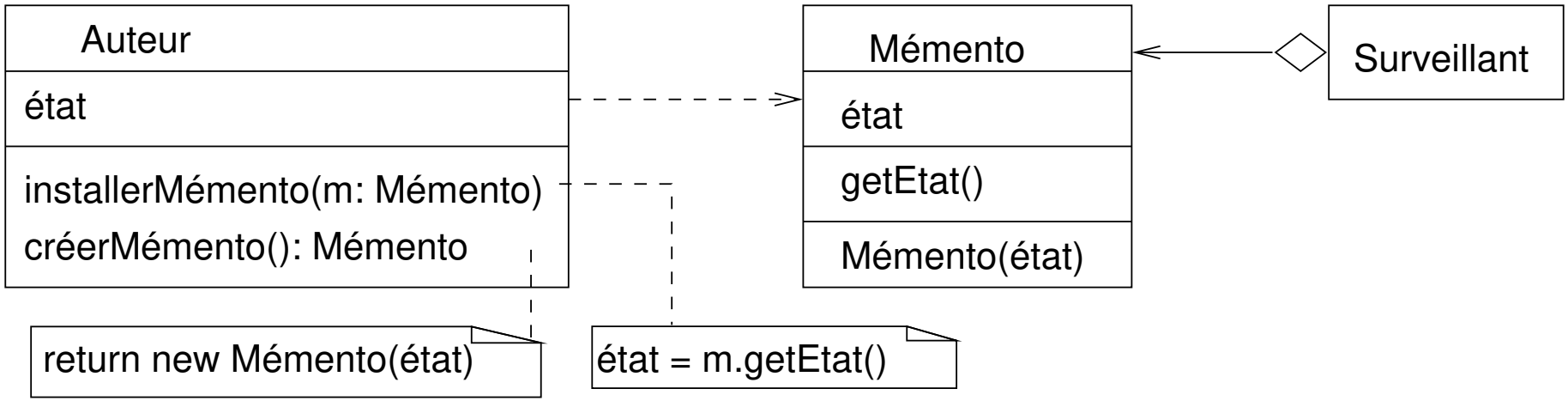
- **Intention**

  Transmettre à l'extérieur d'un objet son état interne **sans violation de l'encapsulation** dans le but de restaurer ultérieurement sont état.

- **Alias :** Jeton
- **Indications d'utilisation**
  - un instantané de tout ou partie d'un objet doit être mémorisé
  - et l'utilisation d'une interface directe pour atteindre l'état conduirait à rompre l'encapsulation
- **Exemples :**
  - Pouvoir passer les niveaux déjà atteints dans un jeu
  - Déplacer des objets graphiques reliés. Classe Résolution des contraintes. Nécessité de collaborer avec Résolution pour annuler.
- **Conséquences :**
  - préservation des frontières de l'encapsulation
  - simplification de l'auteur (car encapsulation préservée)
  - l'utilisation d'un mémento peut être coûteuse
  - peut être difficile de garantir que seul l'auteur aura accès au mémento

Mémento (Memento)

Diagramme de classe



Patrons comportementaux

- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

Observateur (Observer)

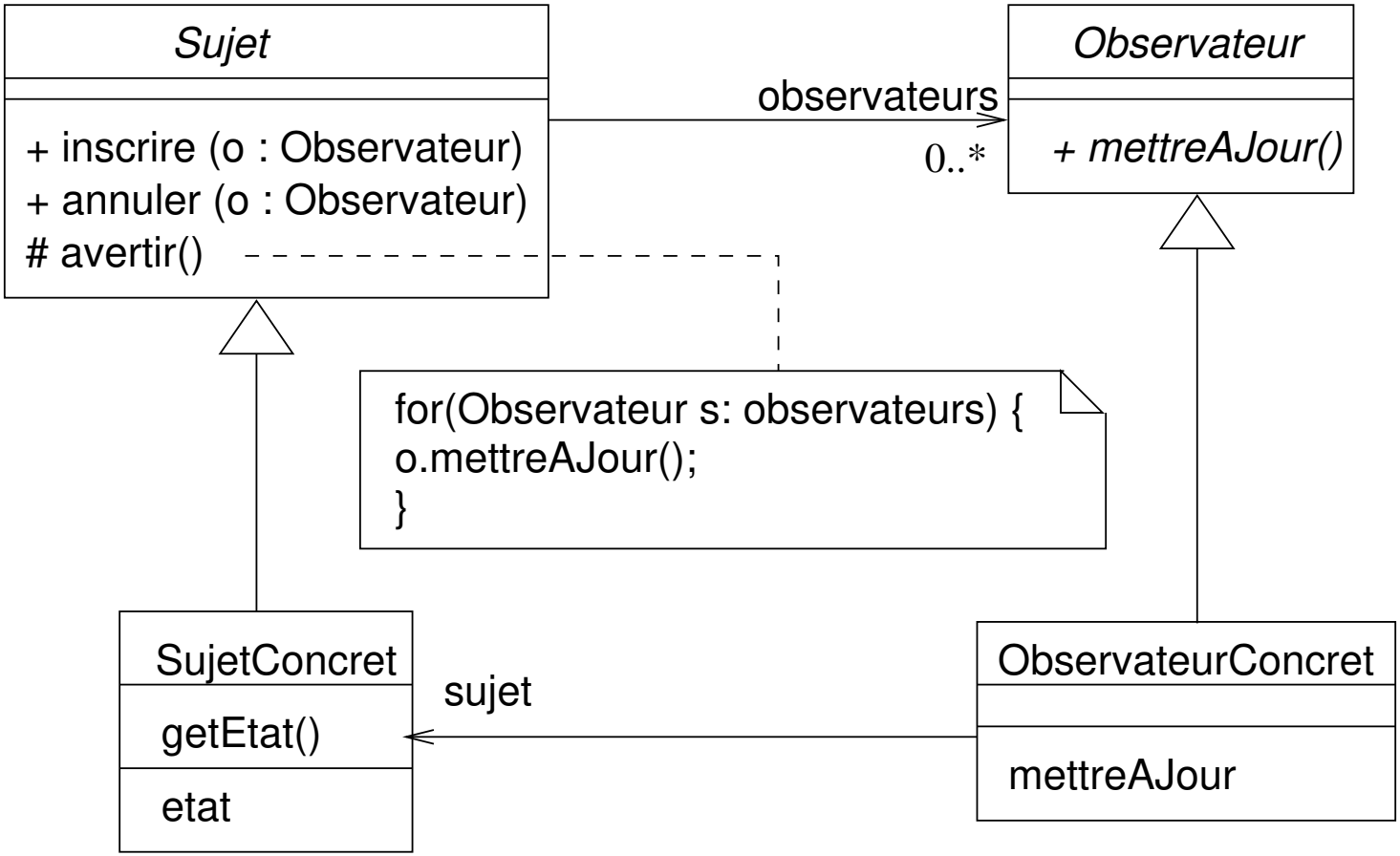
- **Intention**

  Définit une interdépendance de type un à plusieurs de sorte que quand un objet change d'état, tous ceux qui en dépendent en soient avertis et automatiquement mis à jour

- **Alias :** Dépendants, Diffusion–Souscription (Publish–Subscribe)
- **Indications d'utilisation**
  - Pour faciliter la réutilisation de deux représentations inter-dépendantes d'un même concept
  - Quand la modification d'un objet nécessite de modifier un nombre indéterminé d'autres objets
  - Quand un objet doit notifier d'autres objets sans faire d'hypothèses sur la nature de ces objets (faible couplage)
- **Exemples :** Plusieurs vues sur un même modèle

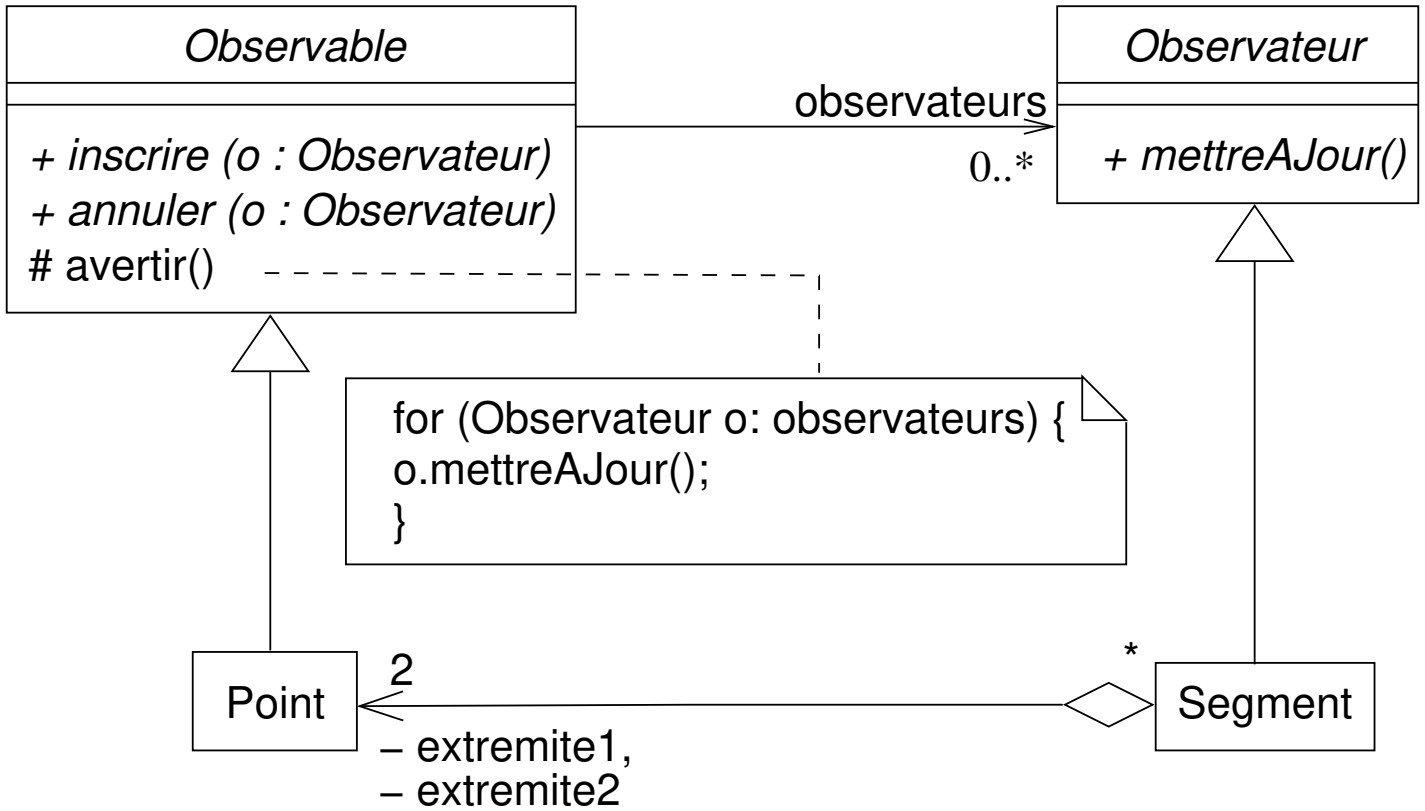
Observateur (Observer)

Diagramme de classe



Observateur (Observer)

Exemple



Patrons comportementaux

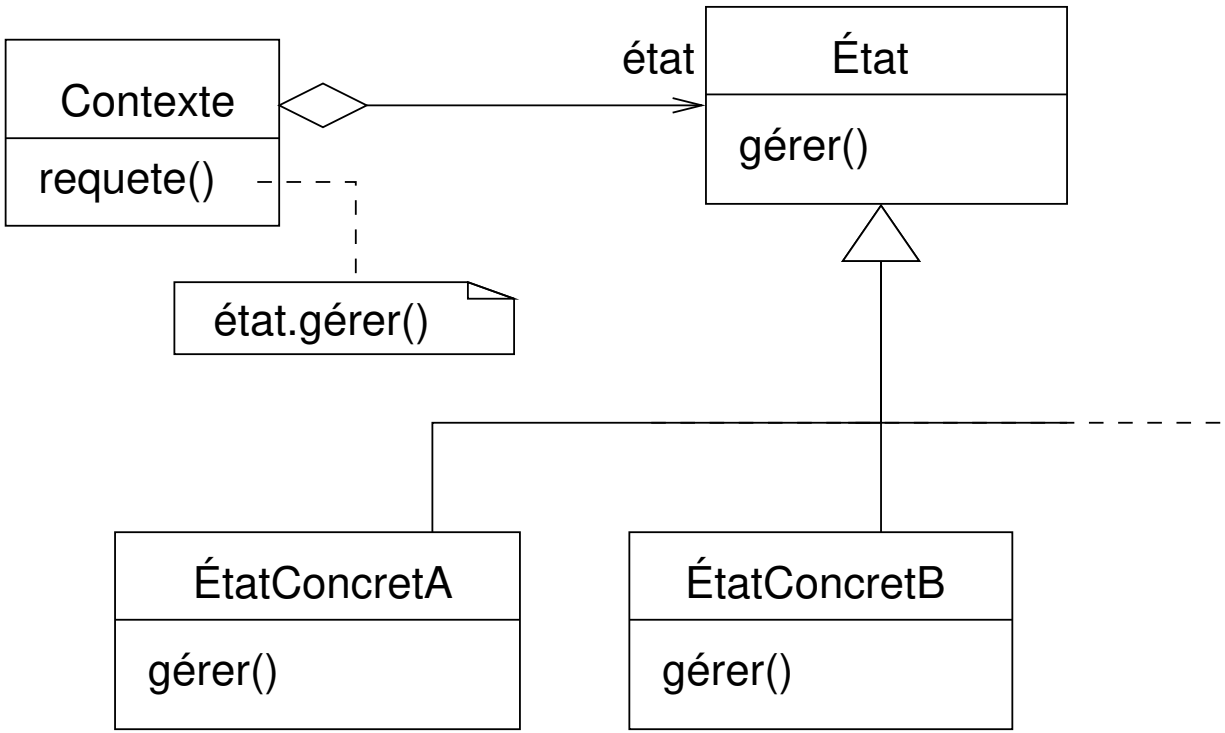
- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

État (State)

- **Intention**
  Permet à un objet de modifier son comportement quand son état interne change. Donne l'impression que l'objet change de classe.
- **Alias :** —
- **Indications d'utilisation**
  - le comportement d'un objet dépend de son état (changement dynamique)
  - les opérations contiennent des conditionnelles en fonction de l'état de l'objet
- **Exemples :**
  - Le joueur d'une équipe de foot (presque !)
  - Une ConnexionTCP qui répondra différemment à une demande d'ouverture suivant son état (établie, écoute, fermée).
- **Conséquence :**
  - Partionnement des différents comportements, état par état
  - Rend explicite les transitions d'état
  - Les objets État peuvent être partagés (si pas d'attributs d'instance) : poids mouche

État (State)

Diagramme de classe



Patrons comportementaux

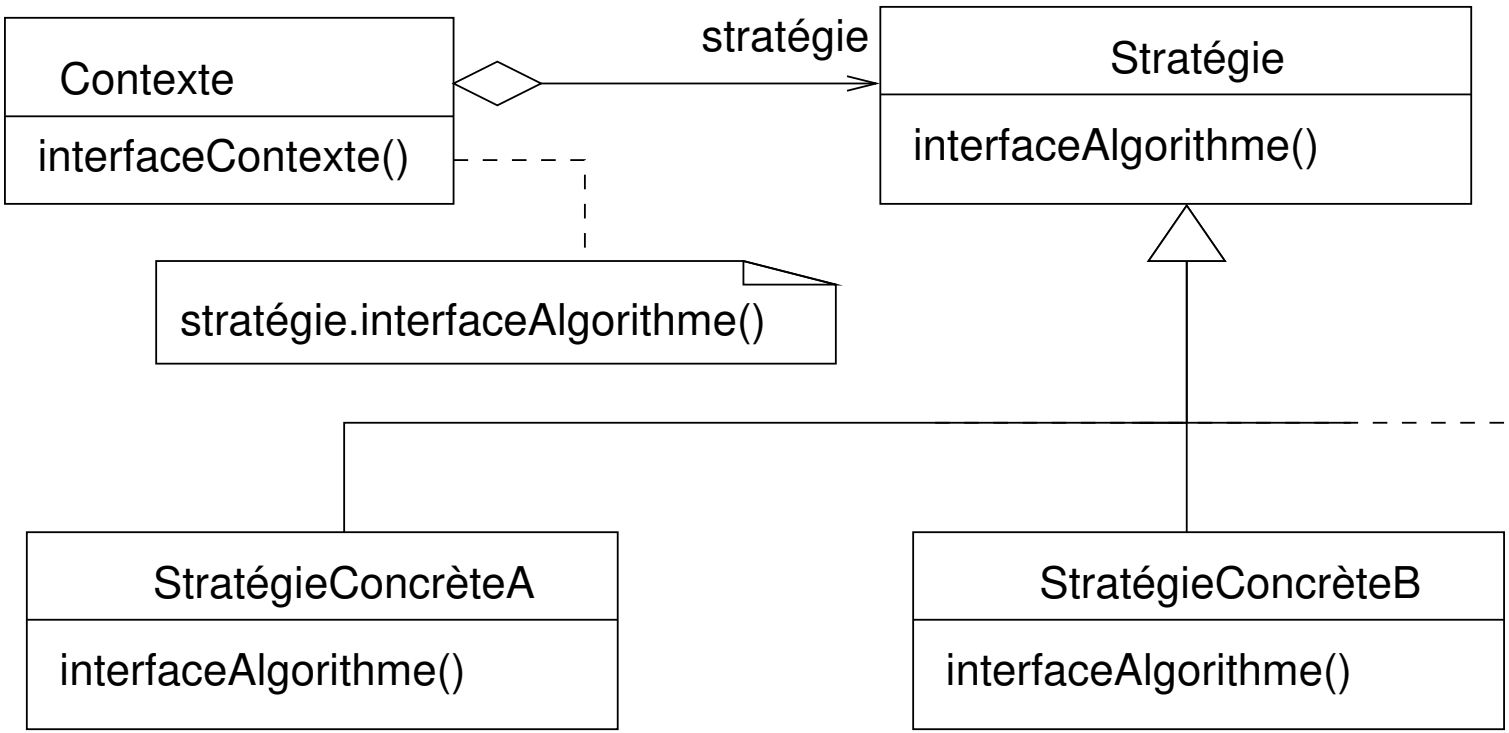
- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- **Stratégie**
- Patron de méthode
- Visiteur

Stratégie (Strategy)

- **Intention**
  - Définit une famille d'algorithme, encapsule chacun d'entre eux et les rend interchangeables.
  - Permet aux algorithmes d'évoluer indépendamment de leurs clients
- **Alias :** Politique
- **Indications d'utilisation**
  - Plusieurs classes apparentées ne différent que par leur comportement
  - On a besoin de diverses variantes d'un algorithme.
  - Un algorithme utilise des données que les clients n'ont pas à connaître (masquer les structures complexes)
  - Une classe définit de nombreux comportements figurant dans des conditionnelles multiples : faire autant de classes Stratégie
- **Exemples :**
  - Gérer les coupures de fin de ligne
  - trier les éléments d'une collection

Stratégie (Strategy)

Diagramme de classe



Patrons comportementaux

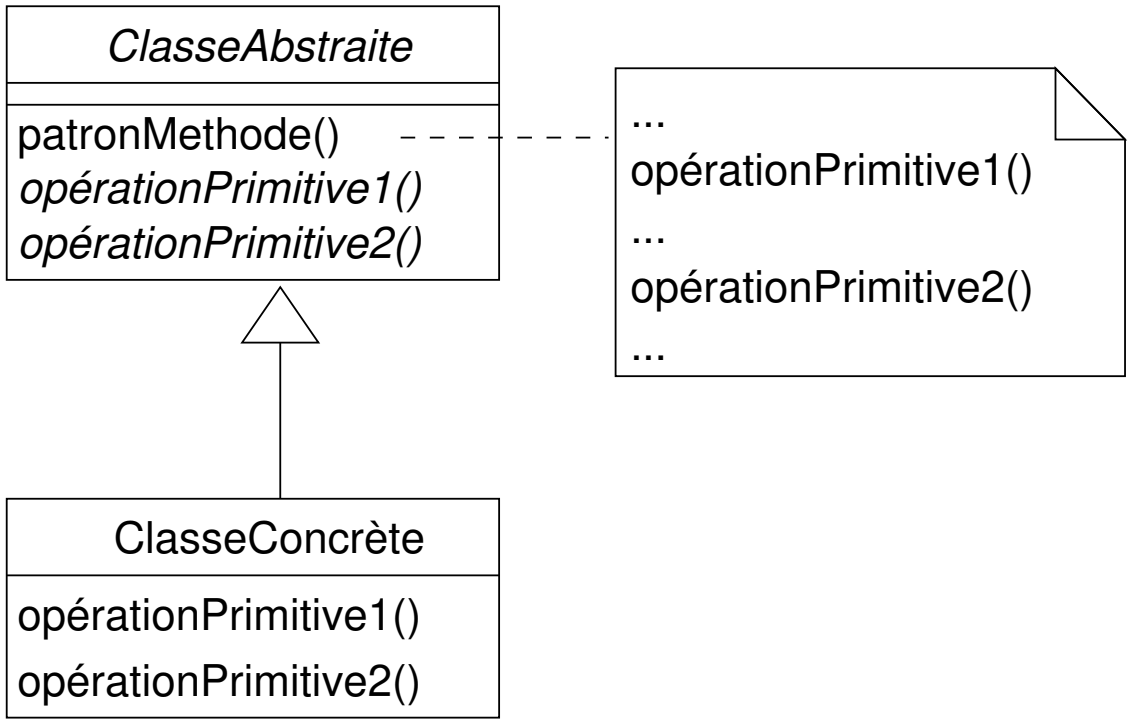
- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

Patron de méthode (Template Method)

- **Intention**
  Définit, dans une opération, le squelette d'un algorithme en en déléguant certaines étapes à des sous-classes.
- **Alias :** —
- **Indications d'utilisation**
  - Pour implanter une fois pour toutes les parties invariantes d'un algorithme
  - Pour isoler et factoriser le comportement comment à des sous-classes
  - contrôler les extensions des sous-classes (garantir un certain comportement)
- **Exemples :**
  - exécuter un test unitaire avec préparer (setUp), tester (test*) et comptabiliser le résultat du test et nettoyer (tearDown).

Patron de méthode (Template Method)

Diagramme de classe



Patrons comportementaux

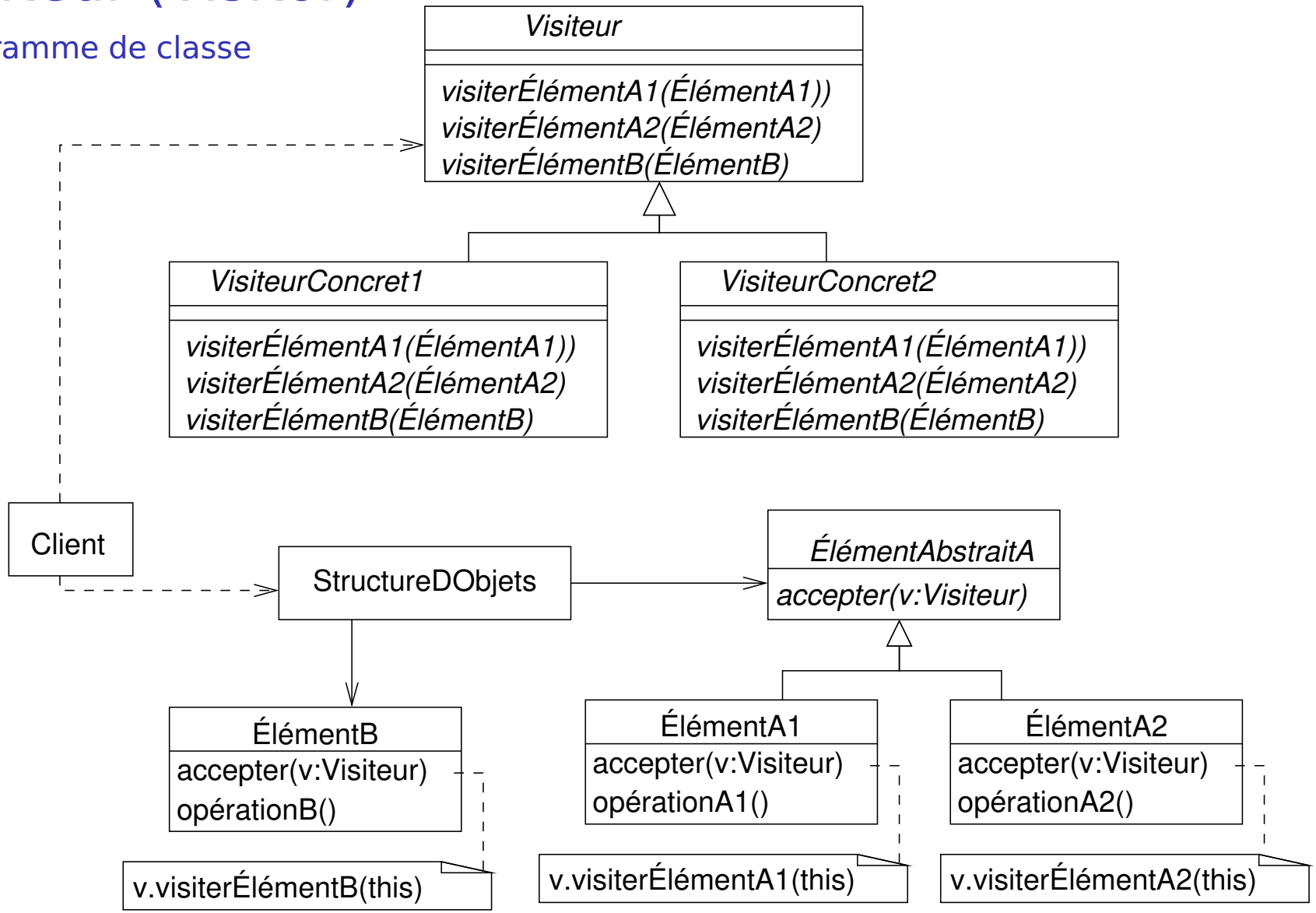
- Chaîne de responsabilités
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Mémento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

Visiteur (Visitor)

- **Intention**
  Modélise par une classe une opération applicable aux éléments d'une structure d'objets et permet de définir de nouvelles opérations sans modifier les classes de la structure
- **Alias :** —
- **Indications d'utilisation**
  - Une structure d'objets contient beaucoup de classes différentes d'interfaces distinctes et l'on veut réaliser des traitements qui dépendent de leurs classes concrètes.
  - Il s'agit de réaliser plusieurs traitements distincts sans relation entre eux, sur les objets de la structure, sans polluer leurs classes
  - Les classes qui définissent la structure d'objet changent rarement mais on doit souvent définir de nouvelles opérations sur cette struture.
- **Exemples :** Afficher en infixe, préfixe et postfixe une expression entière, calculer sa valeur, etc.

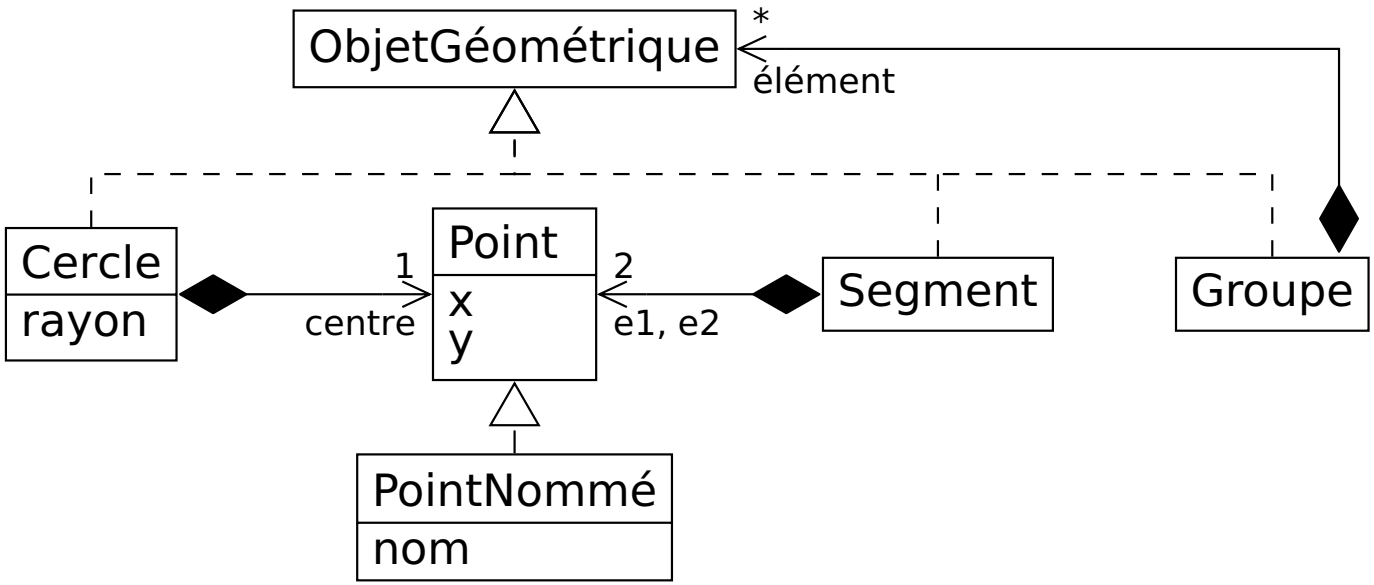
Visiteur (Visitor)

Diagramme de classe



Visiteur appliqué aux objets géométriques : problème posé

- Diagramme de classe structurel des objets géométriques :

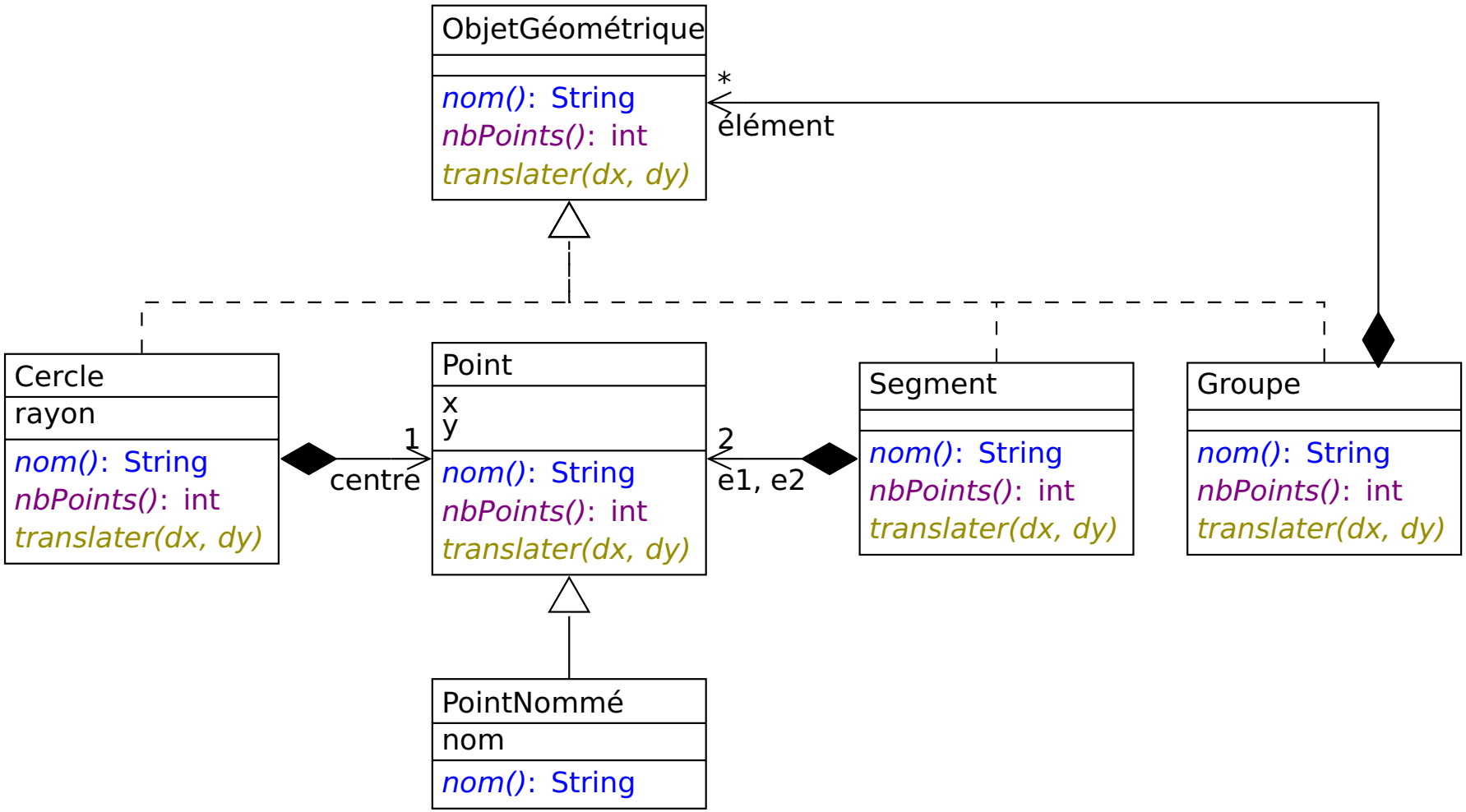


- Traitements à implanter :

1. obtenir le nom en français d'un élément : segment, point...
2. obtenir le nom en anglais
3. nombre de points utilisés pour caractériser un élément
4. afficher un élément
5. dessiner un élément
6. translater un élément
7. obtenir la *bounding box* d'un élément

Solution objet « classique »

Principe : Pour chaque opération, mettre une méthode sur les types généraux (ObjetGéométrique et Point) et la (re)définir dans les sous-classes.



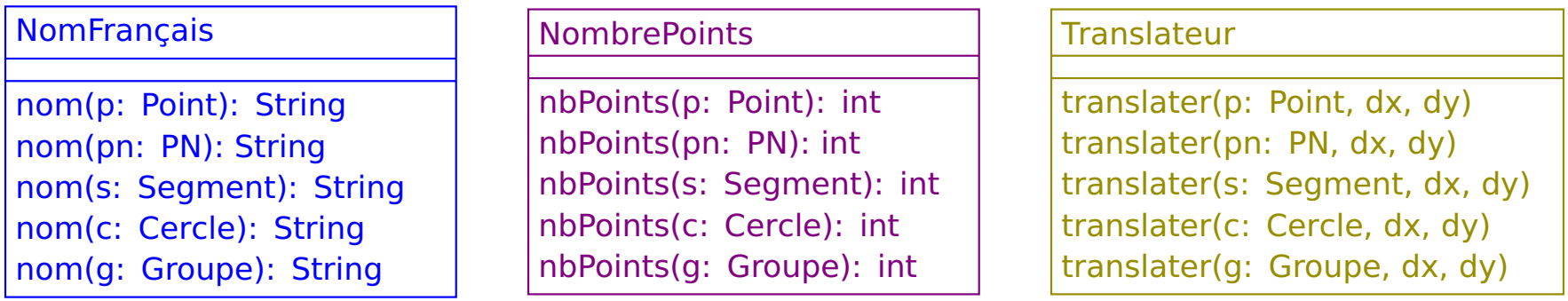
Problèmes induits par cette solution

Solution opérationnelle mais :

- chaque traitement est éclaté sous l'ensemble de la structure de classes
- la structure est polluée par les différents traitements
- il faut pouvoir modifier les classes pour ajouter de nouveaux traitements

Principe du visiteur

- **Principe :** Écrire une classe par traitement.
- **Moyen :**
  - regrouper dans une classe traitement les méthodes implantant ce traitement
  - mettre un paramètre explicite correspond à la classe traitée
  - rendre accessibles les informations internes de la structure (accesseurs)
- **Nouveau diagramme de classe :** diagramme structurel initial +



Traitements NomFrancais et NomAnglais : pas de problème

```
1  public class NomFrancais {
2      public String nom(Point p)        { return "point"; }
3      public String nom(PointNomme pn) { return "point␣nommé"; }
4      public String nom(Segment s)   { return "segment"; }
5      public String nom(Cercle c)       { return "cercle"; }
6      public String nom(Groupe g)     { return "groupe"; }
7  }
```

```
1  public class NomAnglais {
2      public String nom(Point p)        { return "point"; }
3      public String nom(PointNomme pn) { return "named␣point"; }
4      public String nom(Segment s)   { return "segment"; }
5      public String nom(Cercle c)       { return "circle"; }
6      public String nom(Groupe g)     { return "group"; }
7  }
```

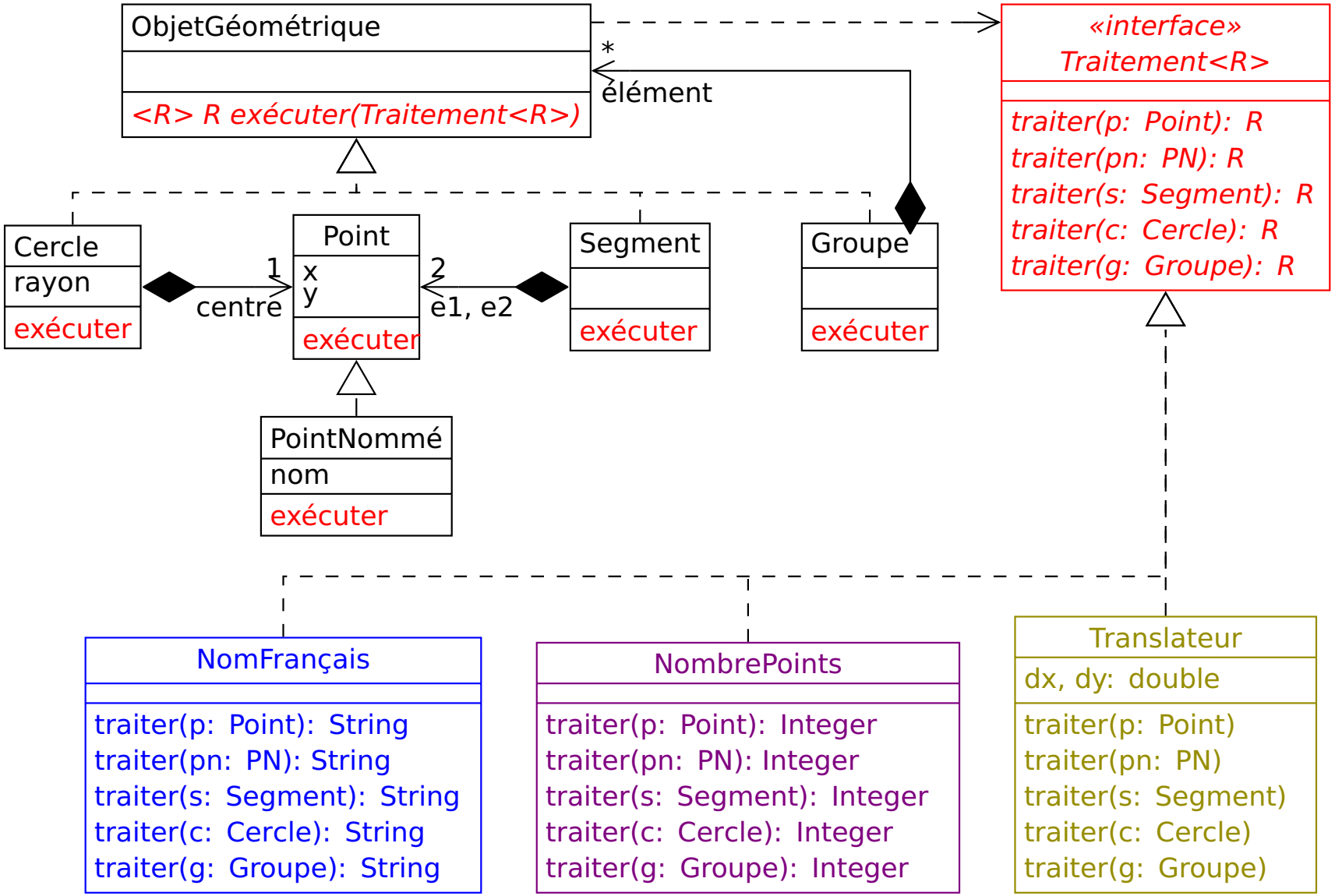
Traitement NombrePoints : pas si facile !

```
1  public class NombrePoints {
2    public int nbPoints(Point p)        { return 1; }
3    public int nbPoints(PointNomme pn) { return 1; }
4    public int nbPoints(Segment s)    { return 2; }
5    public int nbPoints(Cercle c)       { return 1; }
6    public int nbPoints(Groupe g) {
7      int somme = 0;
8      for (ObjetGeometrique og : g.elements()) {
9        if  (og instanceof Segment) {
10         somme += nbPoints((Segment) og);
11       } else if (og instanceof Cercle) {
12         somme += nbPoints((Cercle) og);
13       } else if (og instanceof Groupe) {
14         somme += this.nbPoints((Groupe) og);
15       } else {
16         throw new RuntimeException("Erreur dans le traitement par cas !");
17       }
18     }
19     return somme;
20   }
```

Critiques et... une solution

- **Ça marche mais :**
  - Faire des **instanceof** est une mauvaise idée !
    $\Longrightarrow$ voir l'exception levée...
  - De nombreux traitements nécessitent ce traitement par cas : afficher, translater, dessiner, bounding box, etc.
- **On sait comment faire pour l'éviter !**
  - Il suffit de mettre une méthode correspondant au traitement fait dans le type général et la redéfinir dans les sous-classes....
  - **MAIS** on retombe sur la solution initiale :(
- **Solution :**
  - C'est bien en ajoutant une méthode sur les classes de la structure que l'on peut faire la sélection de la bonne méthode sans **instanceof**
  - L'idée est de ne définir qu'une seule méthode, et non une par traitement
  - Il faut généraliser les différents traitements : c'est le Traitement/Visiteur !
  - Il faut unifier le nom des méthodes : traiter/visiter
  - Le type de retour change ? La généricité
  - Et la méthode sur ObjetGéometrique ?
    exécuter(Traitement)/accepter(Visiteur)

L'architecture du Visiteur



Un peu de code...

- **l'interface Traitement**

```
1  public interface Traitement<R> {
2      R traiter(Point p);
3      R traiter(PointNomme pn);
4      R traiter(Segment s);
5      R traiter(Cercle c);
6      R traiter(Groupe g);
7  }
```

- **Le code de exécuter dans les interfaces et classes abstraites :**

```
1  abstract public <R> R exécuter(Traitement<R> t);
```

- **Le code de exécuter dans les classes concrètes :**

```
1  public class Cercle {
2      public <R> R exécuter(Traitement<R> t) {
3          return t.traiter(this);
4      }
5      ...
6  }
```

Un peu de code : traitement NombrePoints

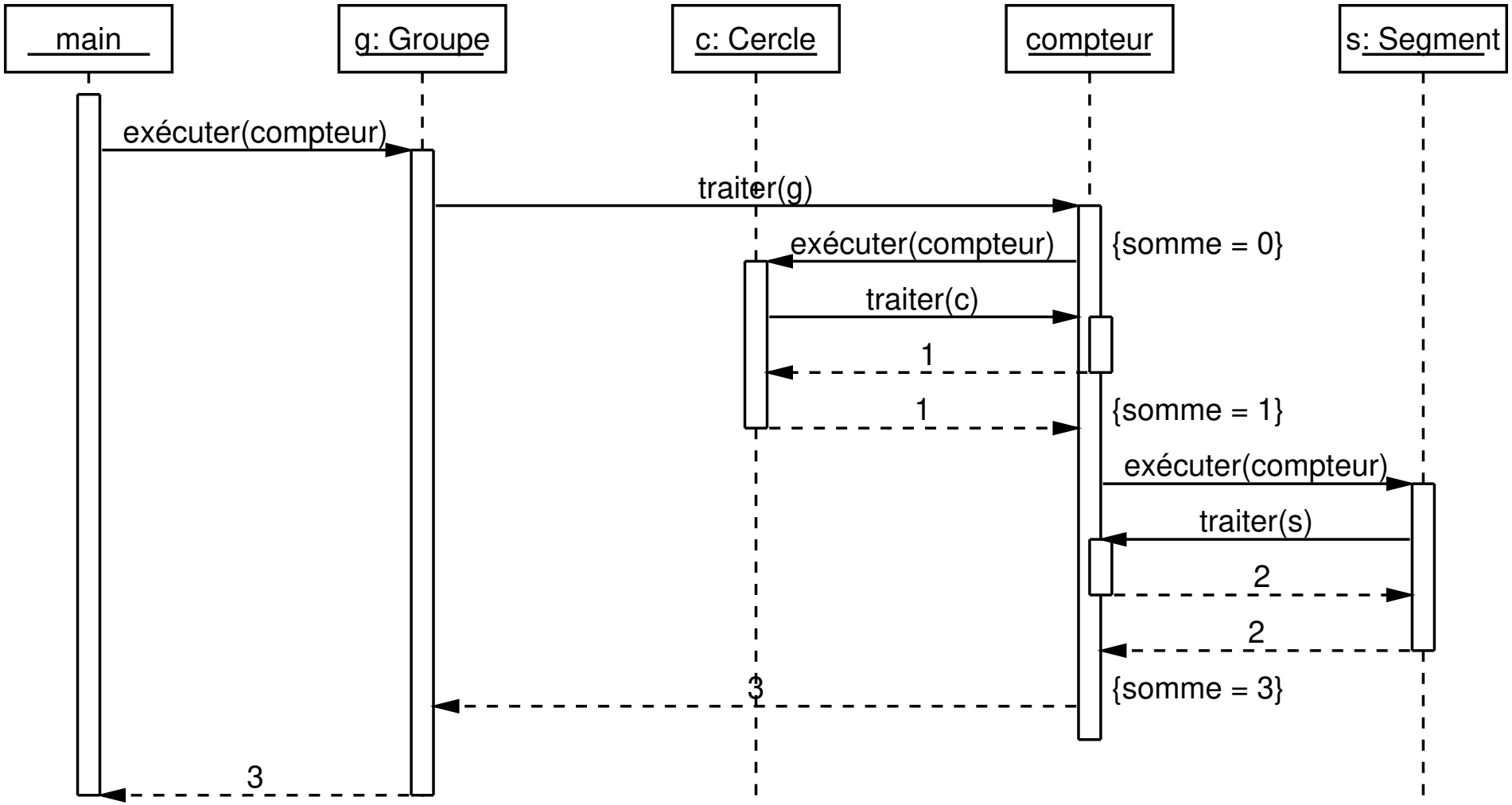
```
1  public class NombrePoints implements Traitement<Integer> {
2      Integer traiter(Point p)              { return 1; }
3      Integer traiter(PointNomme pn)  { return 1; }
4      Integer traiter(Segment s)          { return 2; }
5      Integer traiter(Cercle c)           { return 1; }
6      Integer traiter(Groupe g) {
7          int somme = 0;
8          for (ObjetGeometrique og : g.elements()) {
9              somme += og.executer(this);
10         }
11         return somme;
12     }
13 }
```

Utiliser un visiteur

```
1  Point p1 = new Point(10, −2);
2  Point p2 = new PointNomme(3, 4, "A");
3  Segment s = new Segment(p1, p2);
4  Cercle c = new Cercle(new Point(1, 2), 10);
5  Groupe g = new Groupe();
6  g.ajouter(s);
7  g.ajouter(c);
8  ObjetGeometrique og = g;
9  Traitement<Integer> compteur = new NombrePoints();
10 int nbPoints;
11 nbPoints = g.executer(compteur);   // ???
12 nbPoints = og.executer(compteur); // ???
13 nbPoints = compteur.traiter(g);       // ???
14 nbPoints = compteur.traiter(og);      // ???
15 Traitement<String> nf = new NomFrancais();
16 String nom;
17 nom = p2.executer(nf); // ???
18 nom = nf.traiter(p2); // ???
```

Visiteur appliqué aux objets géométriques

Diagramme de séquence de g.exécuter(compteur)



Visiteur (Visitor)

conséquences et implantation

- **Conséquences :**
  - il facilite l'ajout de nouveaux traitements
  - un visiteur rassemble les opérations d'un même type (Visiteur concret)
  - l'addition de nouvelles classes à la structure est difficile
  - se promener dans une hiérarchie de classe : itérateur (partiel).
  - thésaurisation des informations d'état dans un visiteur
  - rupture d'encapsulation (interface des Éléments riche)
- **Implantation**
  - Double aiguillage : deux critères déterminent l'opération à effectuer, le Visiteur et l'Élément.
  - Qui organise le parcours de la structure d'objet :
    - ★ le visiteur
    - ★ la structure d'objet (souvent choisi)
    - ★ un itérateur à part

Patrons comportementaux : synthèse

- **Encapsultation des parties variables**
  - **Stratégie :** : un algorithme
  - **Patron de méthode :** les étapes d'un algorithme
  - **État :** un comportement dépendant de l'état
  - **Médiateur :** un protocole
  - **Itérateur :** le parcours d'un agrégat
- **Relations statiques supportant la communication**
  - **Commande**, **Observateur**, **Médiateur**, **Chaîne de responsabilités** : découplage entre émetteur et récepteur
  - **Médiateur :** communication encapsulée
  - **Observateur :** communication distribuée


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns.
Addison-Wesley Professional, January 1995.


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns, Catalogue de modèles de conceptions réutilisables.
Vuibert, October 1999.


Mark Grand.
Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, volume 1.
Wiley, 2 edition, 2002.


Wikipedia : Design Pattern.
`http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29`.