Patterns de création

- Fabrique Abstraite
- Constructeur
- Méthode Fabrique
- Prototype
- Singleton

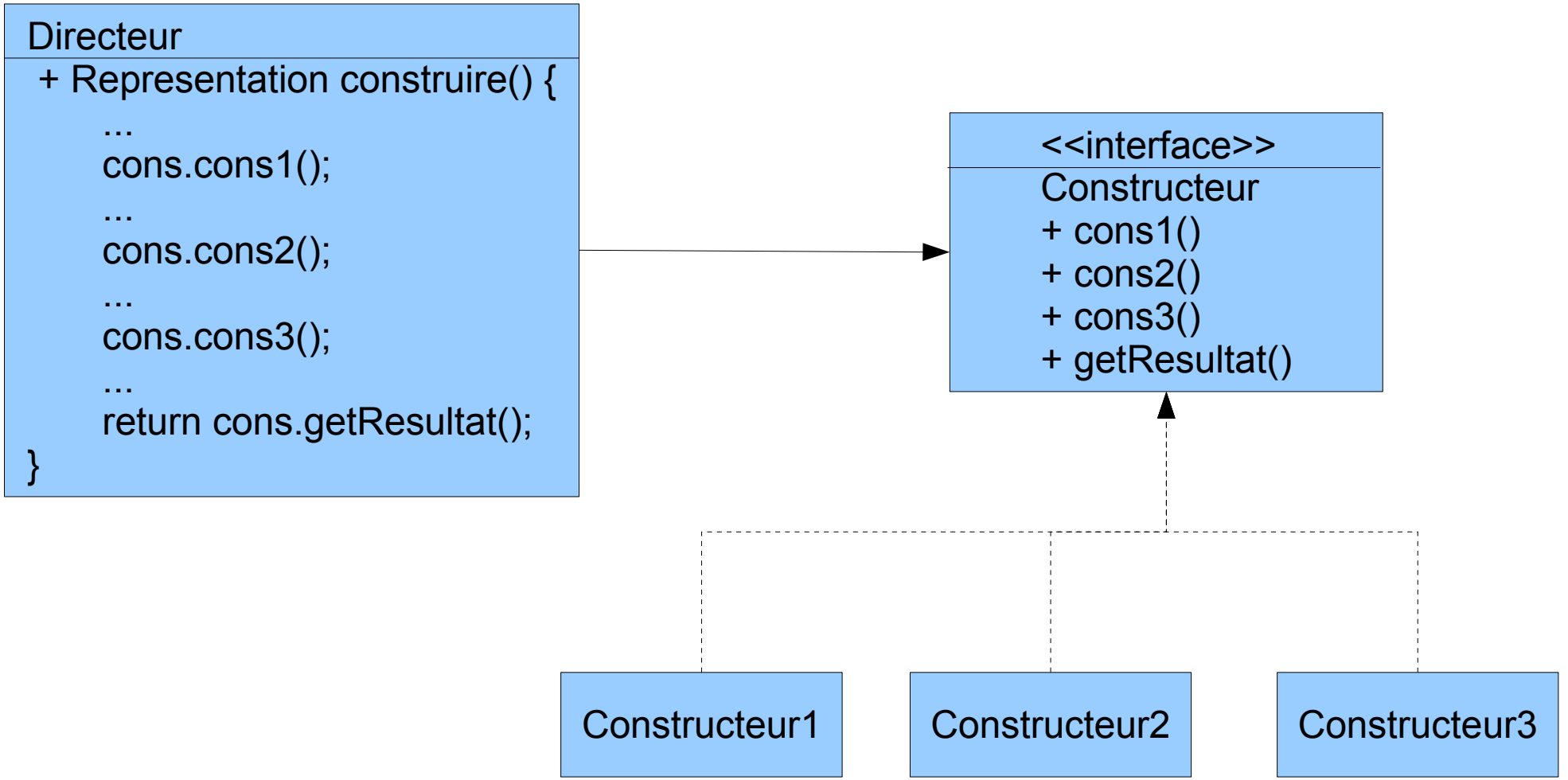
Fabrique Abstraite (Abstract Factory)

Voir cours de base

Constructeur (Builder)

- But
  - Séparer la construction d'un objet complexe de sa représentation de façon à ce que le même processus de construction puisse créer différentes représentations
- Application
  - On veut pouvoir créer différentes représentations d'un objet
  - L'algo général pour créer une représentation ne dépend pas de la représentation

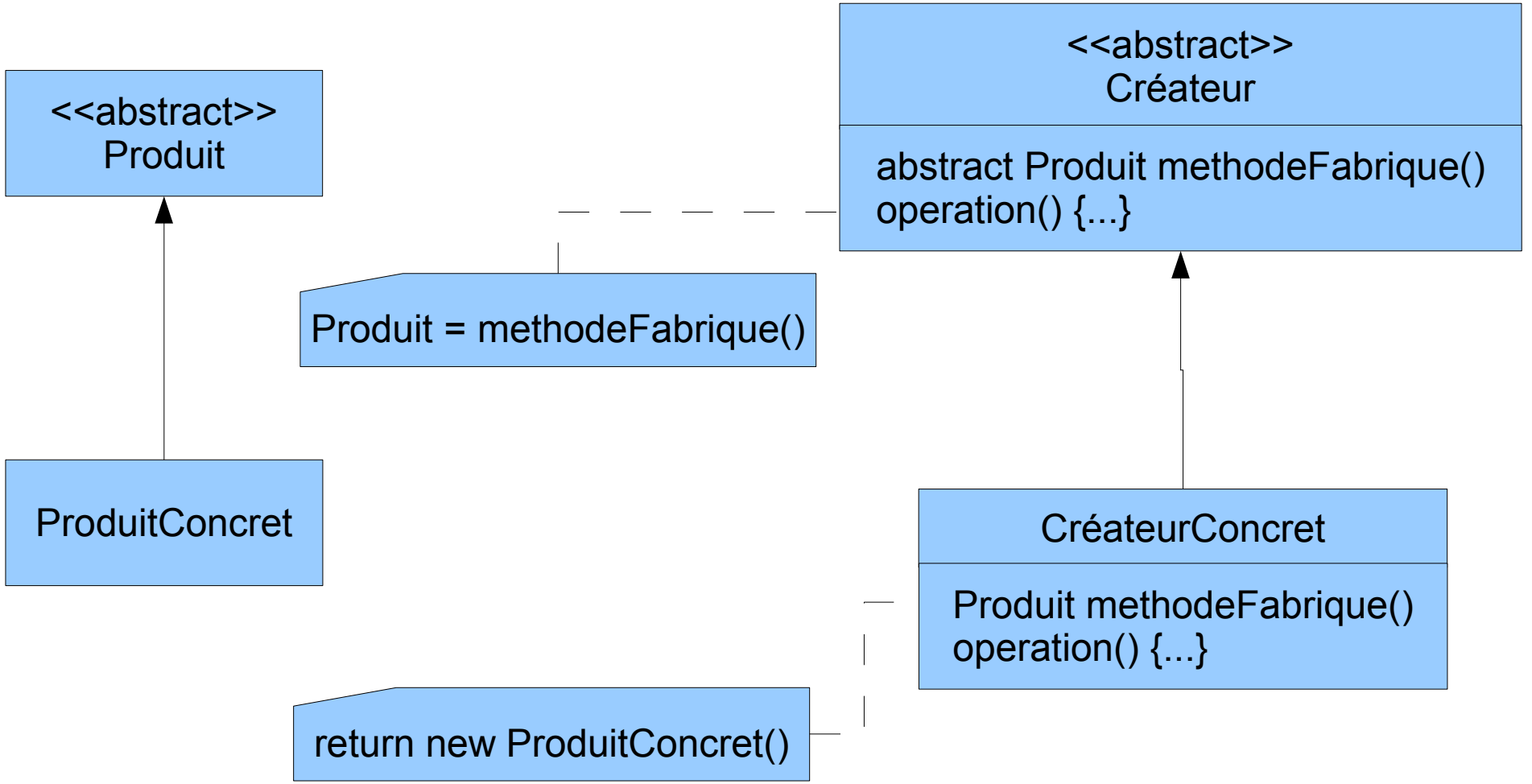
Constructeur (Builder) – 2



Méthode Fabrique (Factory Method)

- But
  - Définir une interface pour créer un objet, mais laisser les sous-classes décider quelle classe instancier

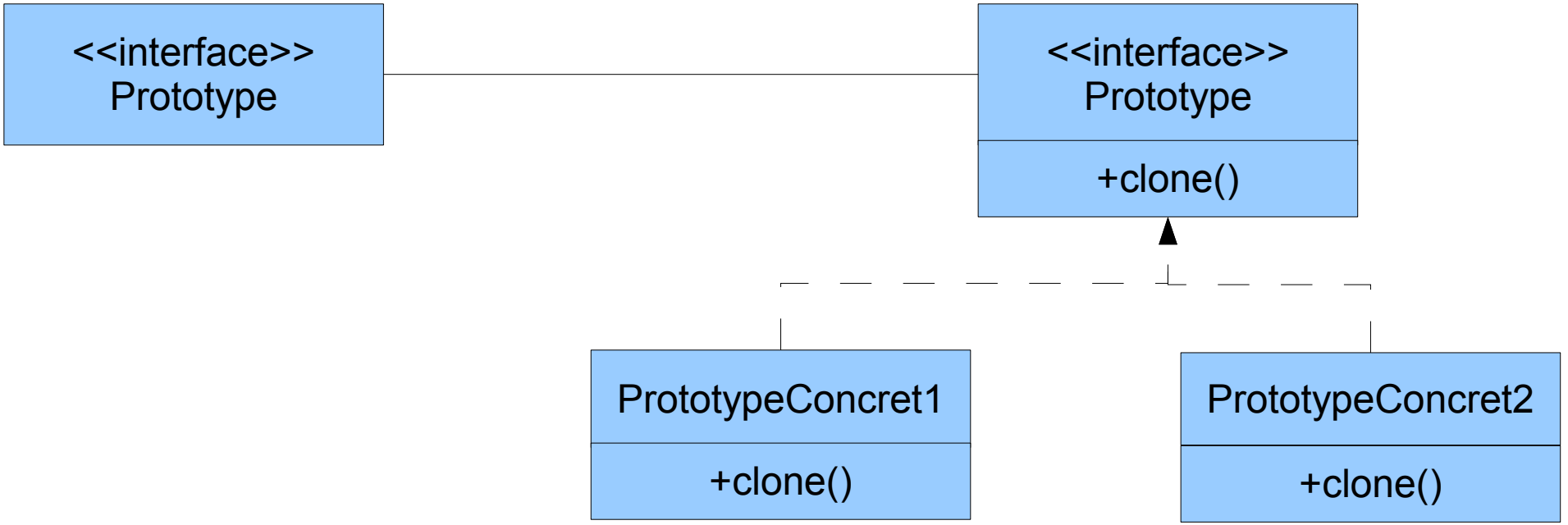
Méthode Fabrique (Factory Method) – 2



Prototype

- But
  - De nombreuses données à manipuler ne sont que de simples variations l'une de l'autre
  - On veut limiter le nombre de sous-classes d'une classe mère
- Principe
  - On définit un ou plusieurs objets prototype par classe
  - On crée les nouveaux objets essentiellement par copie des objets prototypes

Prototype – 2


<<interface>>
Prototype
<<interface>>
Prototype
+clone()
PrototypeConcret1
+clone()
PrototypeConcret2
+clone()

Singleton

Voir Cours de base

Patterns structuraux

- Adaptateur
- Pont
- Composite
- Décorateur
- Façade
- Poids Mouche
- Proxy

Adaptateur

Voir cours de base

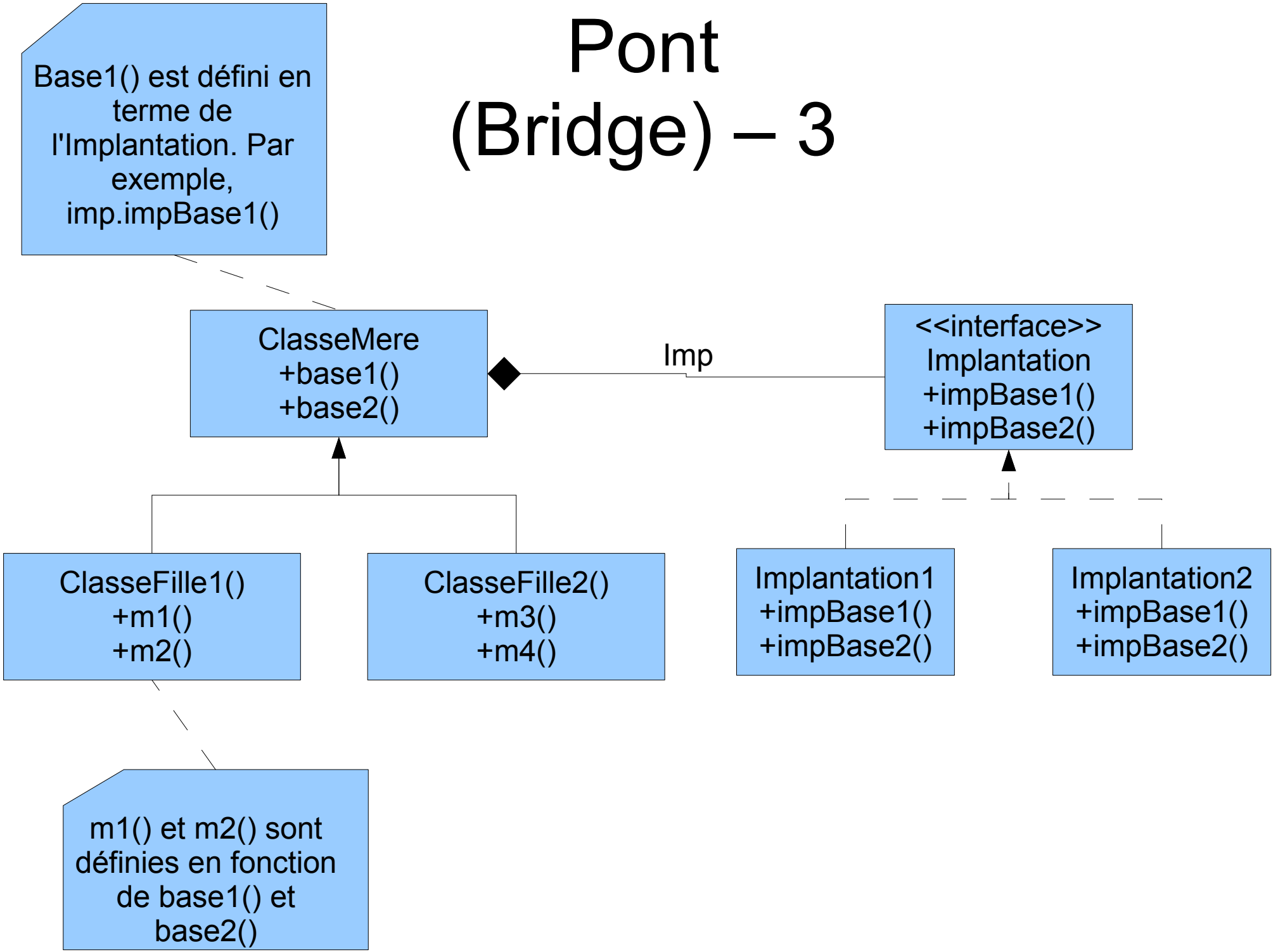
Pont
(Bridge)

- But
  - Découpler une abstraction de son implantation pour que les 2 puissent varier indépendamment
- Problème
  - Une hiérarchie de concepts
  - Plusieurs implantations possibles
- Exemple
  - Hiérarchie d'objets graphiques
  - Plusieurs implantations possibles selon le gestionnaire graphique

Pont (Bridge) – 2

- Solution
  - On définit la hiérarchie des objets graphiques normalement
  - Les opérations de base sont définies dans la super-classe
  - Les opérations des sous-classes sont définies en terme des opérations de base
  - La super-classe contient un objet de type "gestionnaireGraphique"
  - Chaque gestionnaire graphique implante l'interface gestionnaireGraphique

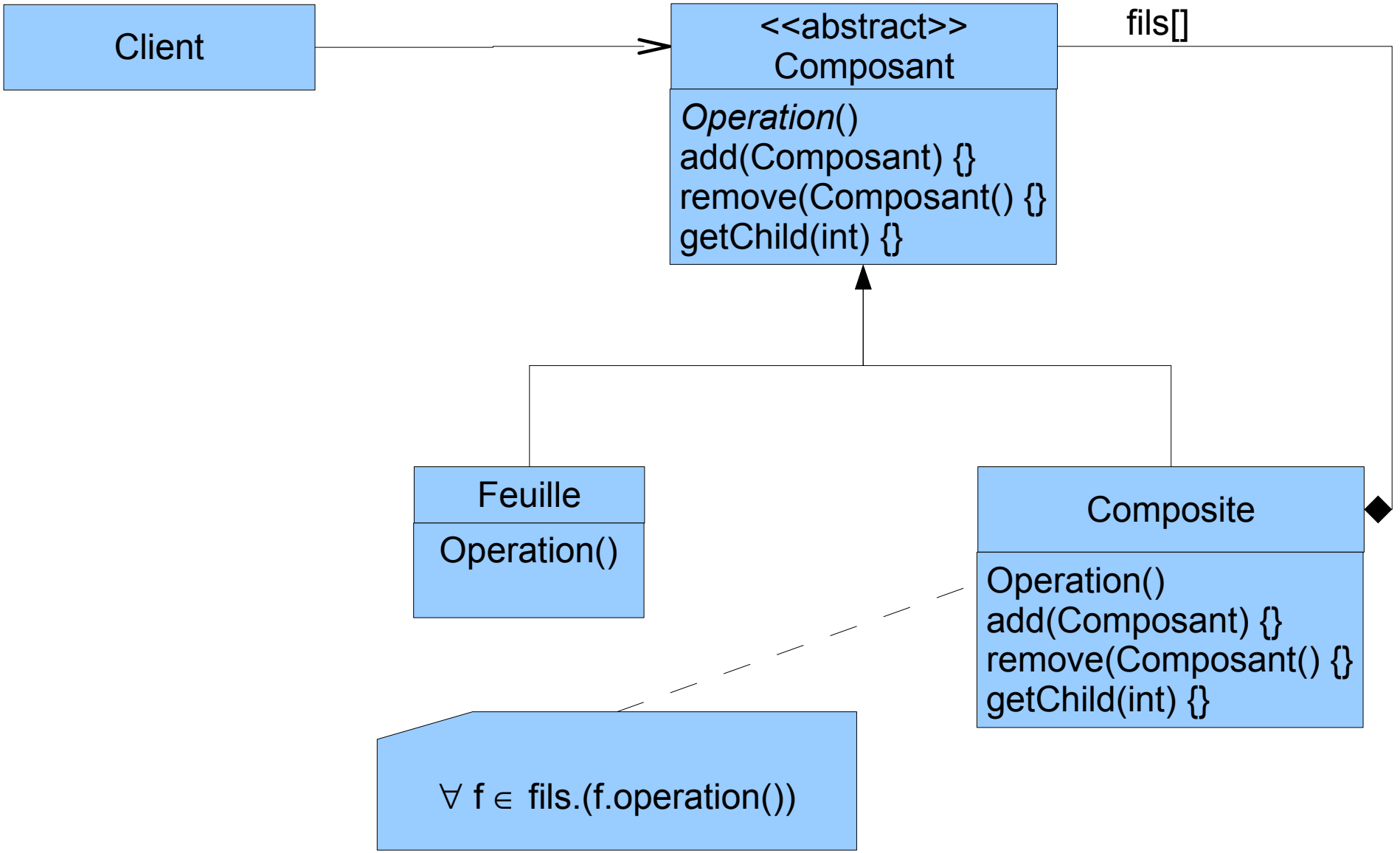
Pont (Bridge) – 3



Composite

- But
  - Pouvoir créer des objets constituer d'autres objets et pouvoir les manipuler comme des objets standards
- Exemple
  - Le groupage d'objets graphiques dans des applications type dessin vectoriel

Composite – 2


Client
<<abstract>>
Composant
Operation()
add(Composant) {}
remove(Composant() {}
getChild(int) {}
fils[]
Feuille
Operation()
Composite
Operation()
add(Composant) {}
remove(Composant() {}
getChild(int) {}
∀ f ∈ fils.(f.operation())

Decorateur

Voir Cours de base

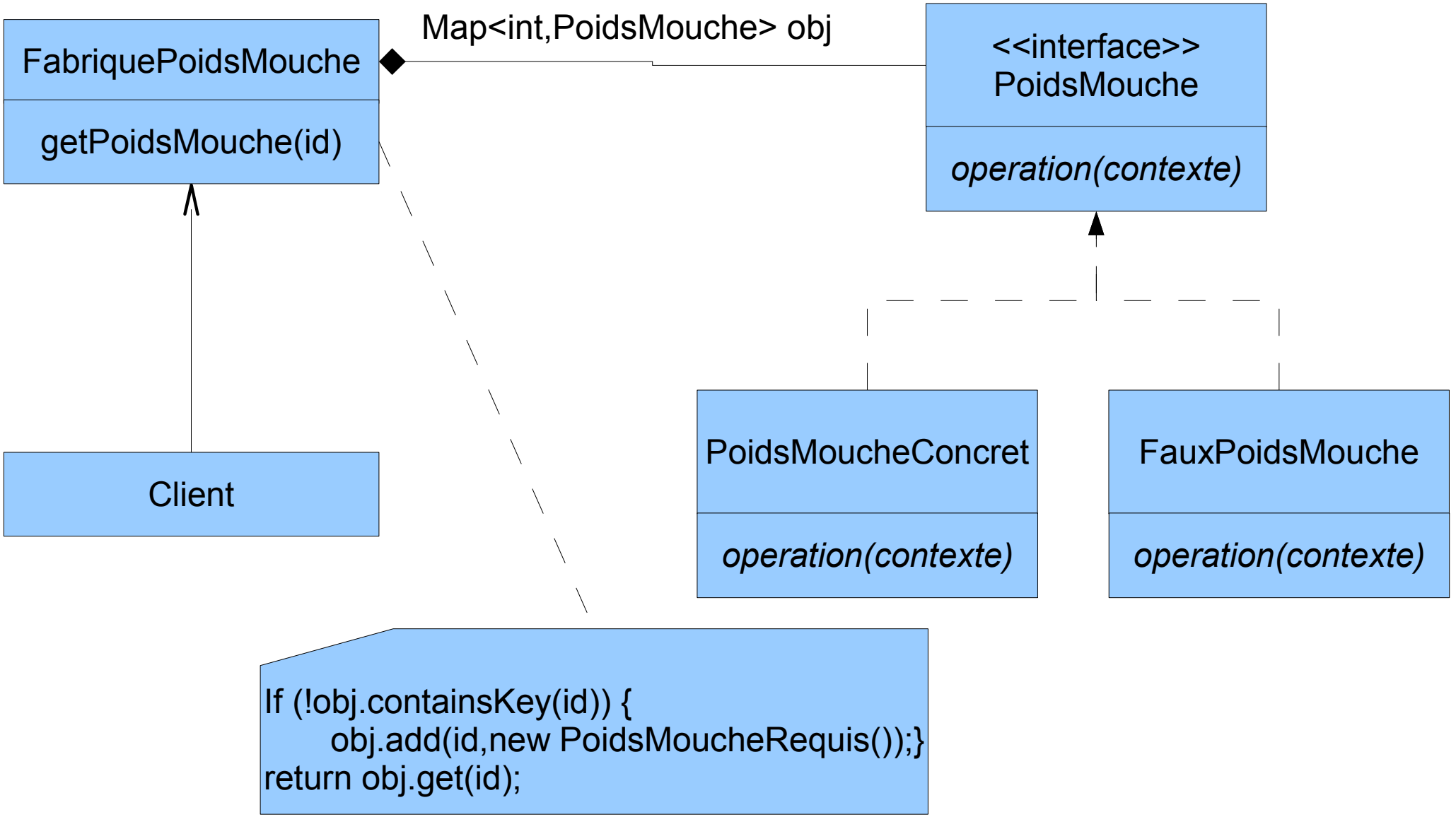
Façade

Voir Cours de base

Poids mouche (Flyweight)

- Problème
  - Beaucoup d'objets à gérer simples à gérer
  - De nombreux objets ne diffèrent que par leur contexte d'utilisation
- Solution
  - Ne créer qu'un objet pour chaque groupe d'objets (uniquement lorsque nécessaire)
  - Passer le contexte en paramètre lorsque c'est nécessaire
- Exemple : éditeur de texte
  - un objet par (lettre,police,taille)
  - La position pour le dessin est donnée dans le contexte

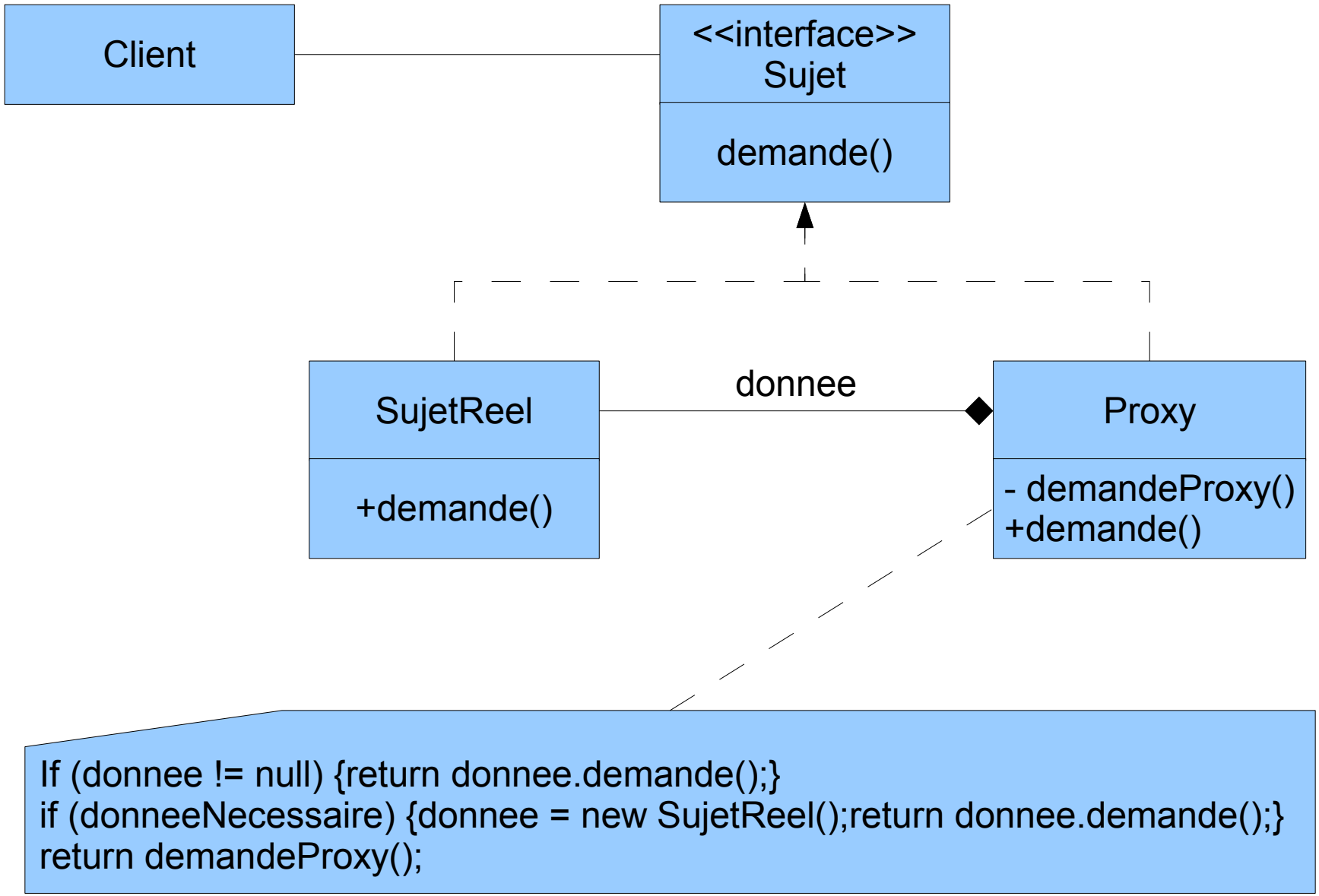
Poids mouche (Flyweight) – 2



Proxy

- But
  - Utiliser un *représentant* d'un autre objet pour contrôler les accès à cet objet
- Exemple
  - Dans un outil de traitement de texte, il n'est pas nécessaire de charger une image tant qu'elle n'est pas affichée ; seule ses dimensions comptent
- Solution
  - Un objet proxy fournit la même interface que l'objet cible. Suivant les cas, renvoie sa propre réponse ou celle de l'objet cible

Proxy – 2


Client
<<interface>>
Sujet
demande()
SujetReel
+demande()
donnee
Proxy
- demandeProxy()
+demande()
If (donnee != null) {return donnee.demande();}
if (donneeNecessaire) {donnee = new SujetReel();return donnee.demande();}
return demandeProxy();

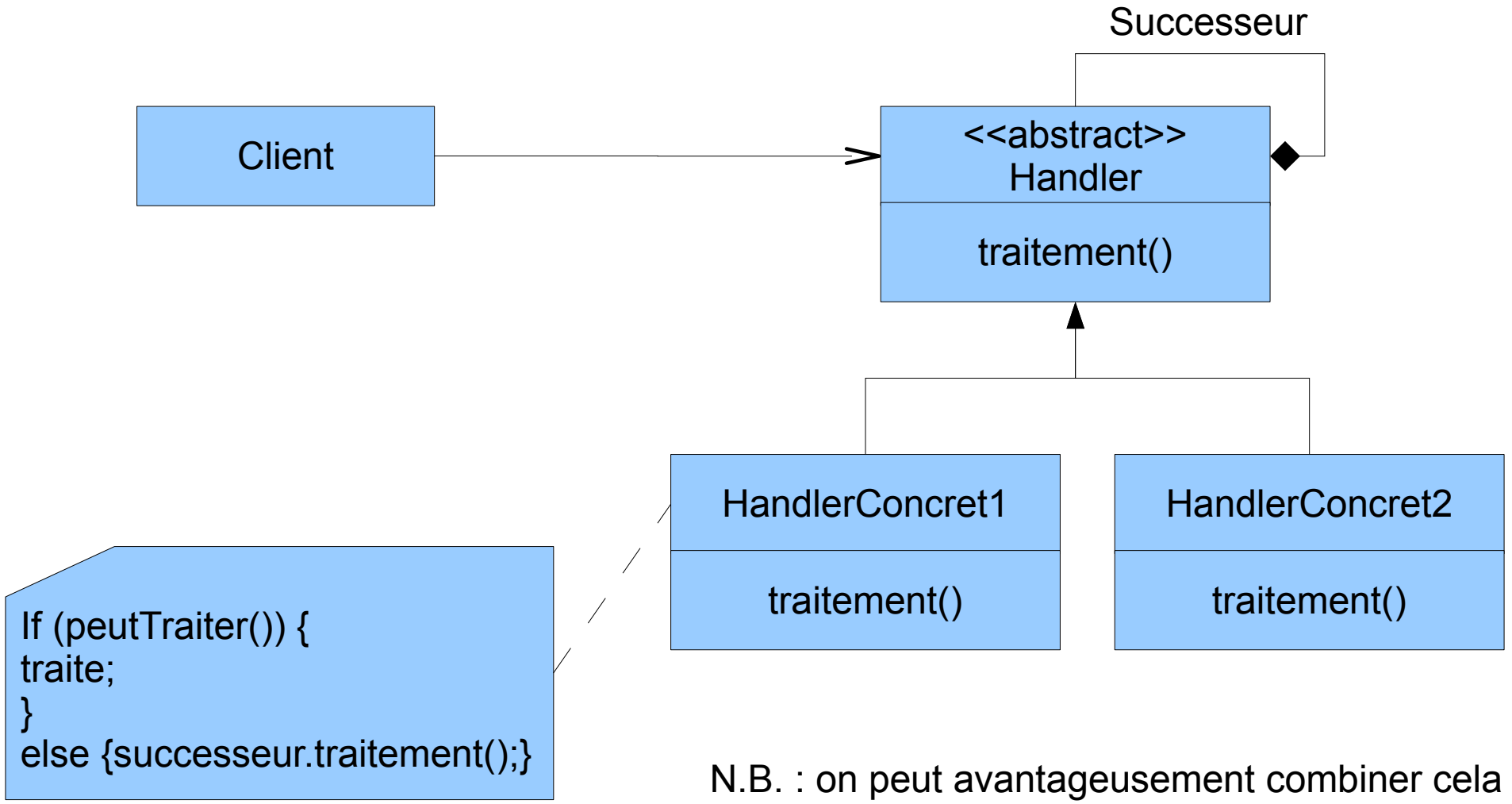
Patterns comportementaux

- Chaîne de responsabilité
- Commande
- Interpréteur
- Itérateur
- Médiateur
- Memento
- Observateur
- État
- Stratégie
- Patron de méthode
- Visiteur

Chaîne de responsabilité (Chain of Responsibility)

- But
  - Éviter de coupler fortement l'émetteur d'un événement avec l'objet qui va le traiter et permettre à plusieurs objet de l'intercepter, selon une chaîne d'objets

Chaîne de responsabilité (Chain of Responsibility) – 2

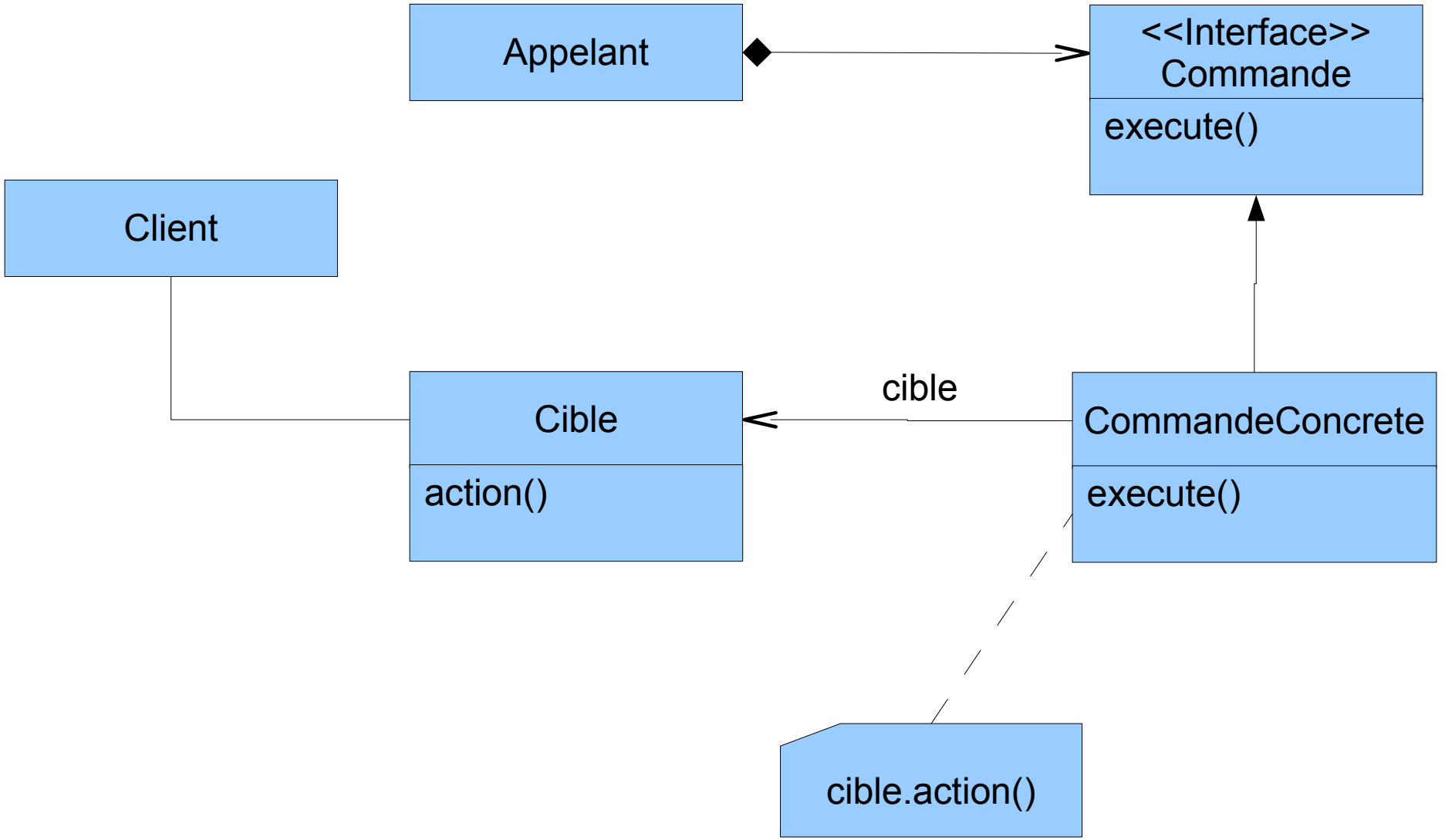


N.B. : on peut avantageusement combiner cela avec le pattern *Patron de méthode*

Commande (Command)

- But
  - Définir les actions comme des objets ; il devient alors facile de donner plusieurs moyens d'accéder à une même action
- Exemple
  - Classe Action de Swing

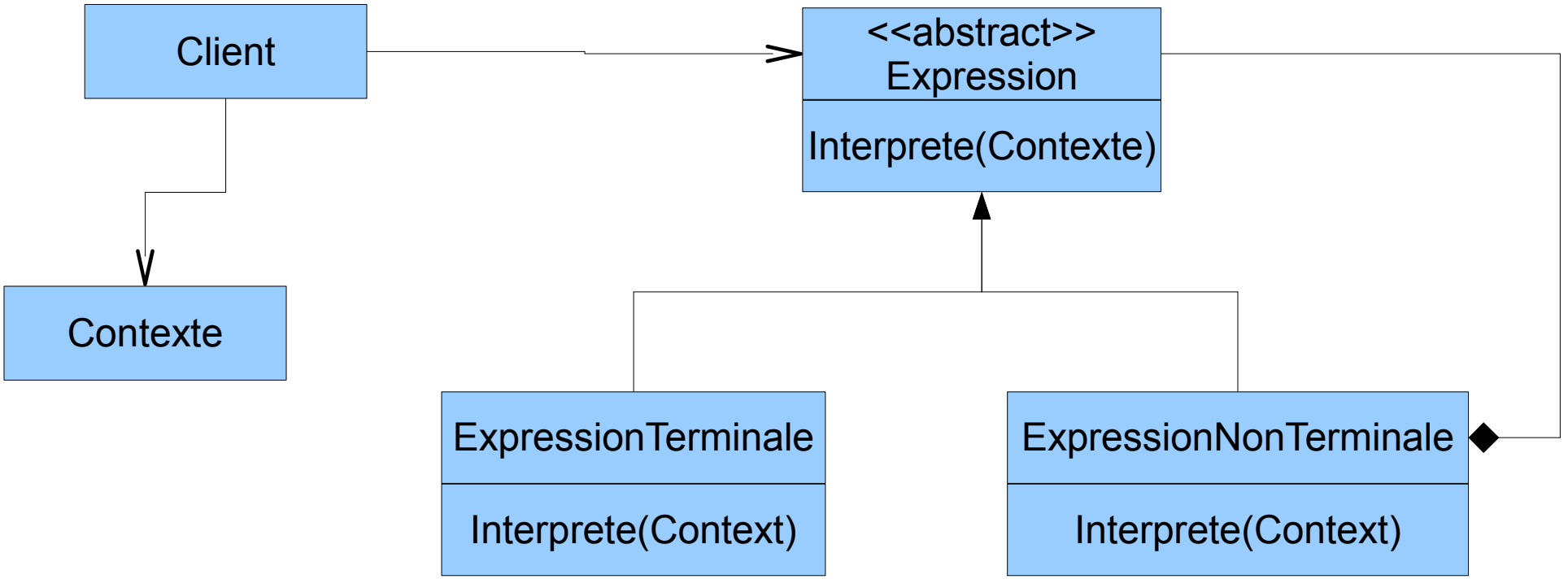
Commande (Command) – 2



Interpréteur (Interpreter)

- But
  - Définir une structure objet représentant la grammaire d'un langage pour interpréter des phrases du langage
- Solution
  - Une classe pour chaque non-terminal/règle
  - Une classe pour chaque terminal (avec éventuellement design pattern *Prototype*)

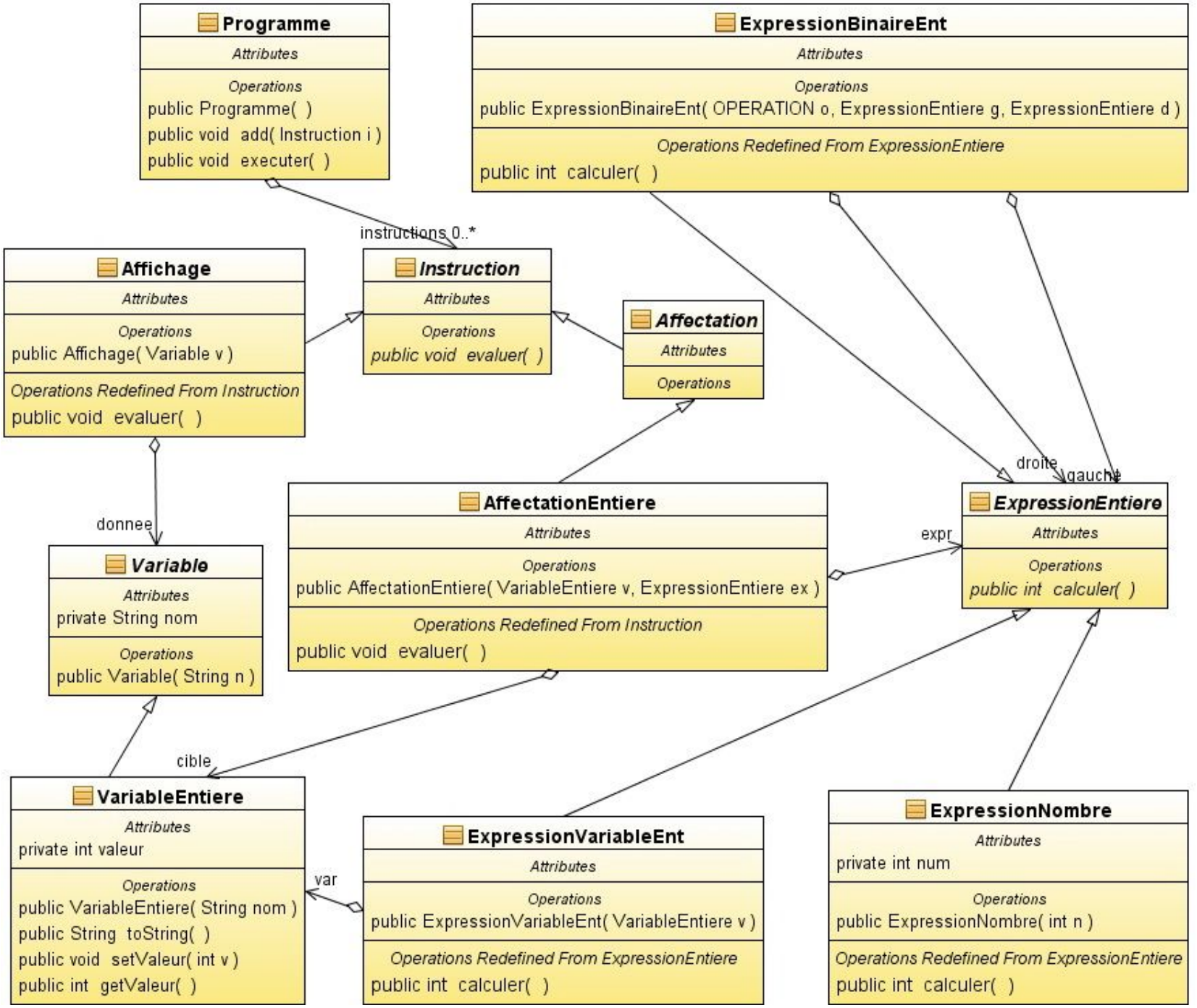
Interpréteur (Interpreter) – 2



Interpréteur (Interpreter) – 3

Programme :: Instruction | Instruction Programme
Instruction :: Affichage | Affectation
Affichage :: print Variable
Variable :: VariableEntiere
VariableEntiere :: nom
Affectation :: AffectationEntiere
AffectationEntiere :: VariableEntiere :: ExpressionEntiere
ExpressionEntiere :: ExpressionNombre | ExpressionVariableEntiere
| ExpressionBinaireEntiere
ExpressionNombre :: nombre
ExpressionVariableEntiere :: VariableEntiere
ExpressionBinaireEntiere :: ExpressionEntiere '+' ExpressionEntiere
| ExpressionEntiere '-' ExpressionEntiere
| ExpressionEntiere '*' ExpressionEntiere
| ExpressionEntiere '/' ExpressionEntiere

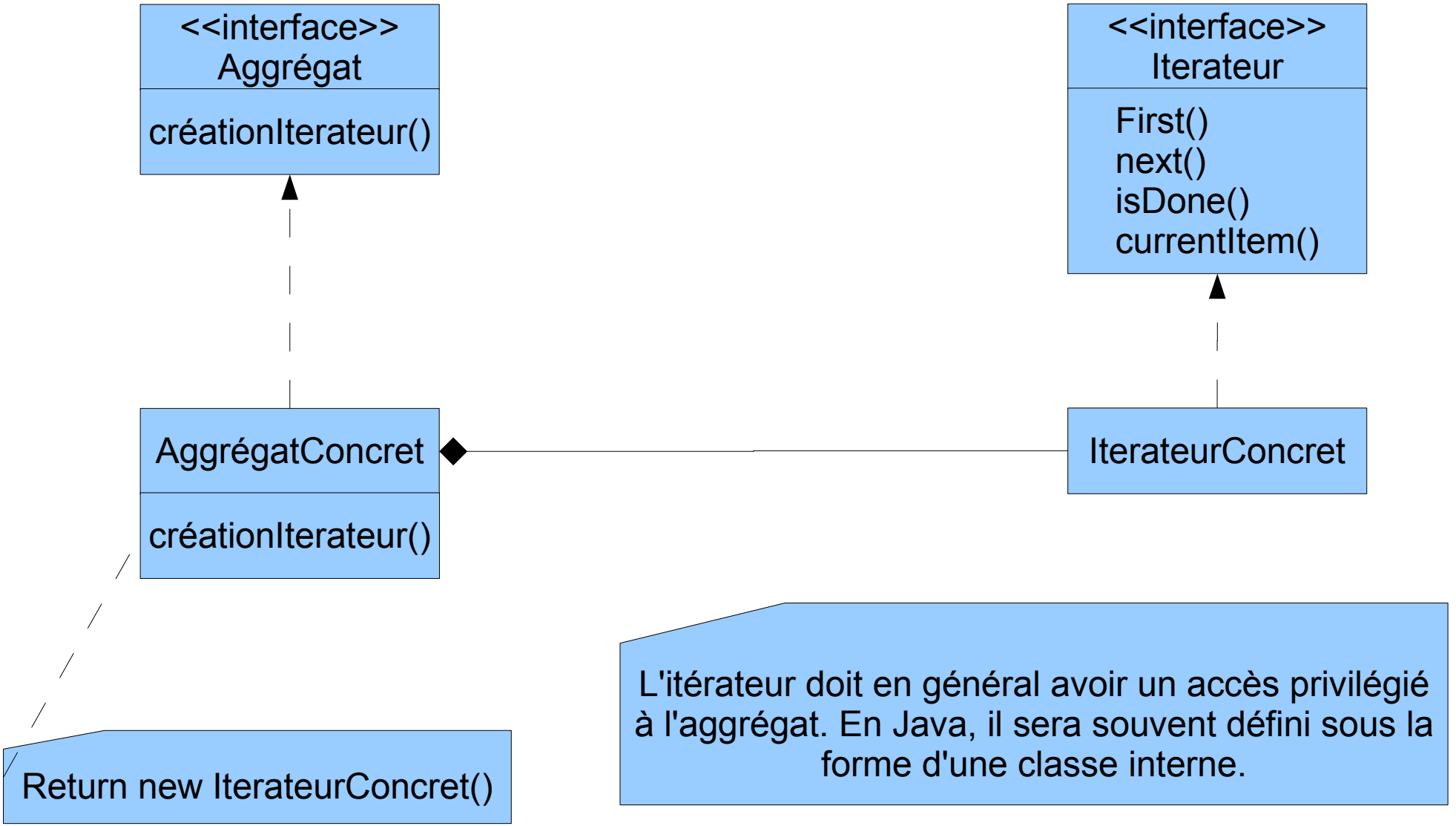
Interpréteur (Interpreter) – 4



Itérateur
(Iterator)

- But
  - Permettre de parcourir les éléments d'un objet composé sans dévoiler la représentation interne
- Exemple
  - Iterator du Java Collection Framework

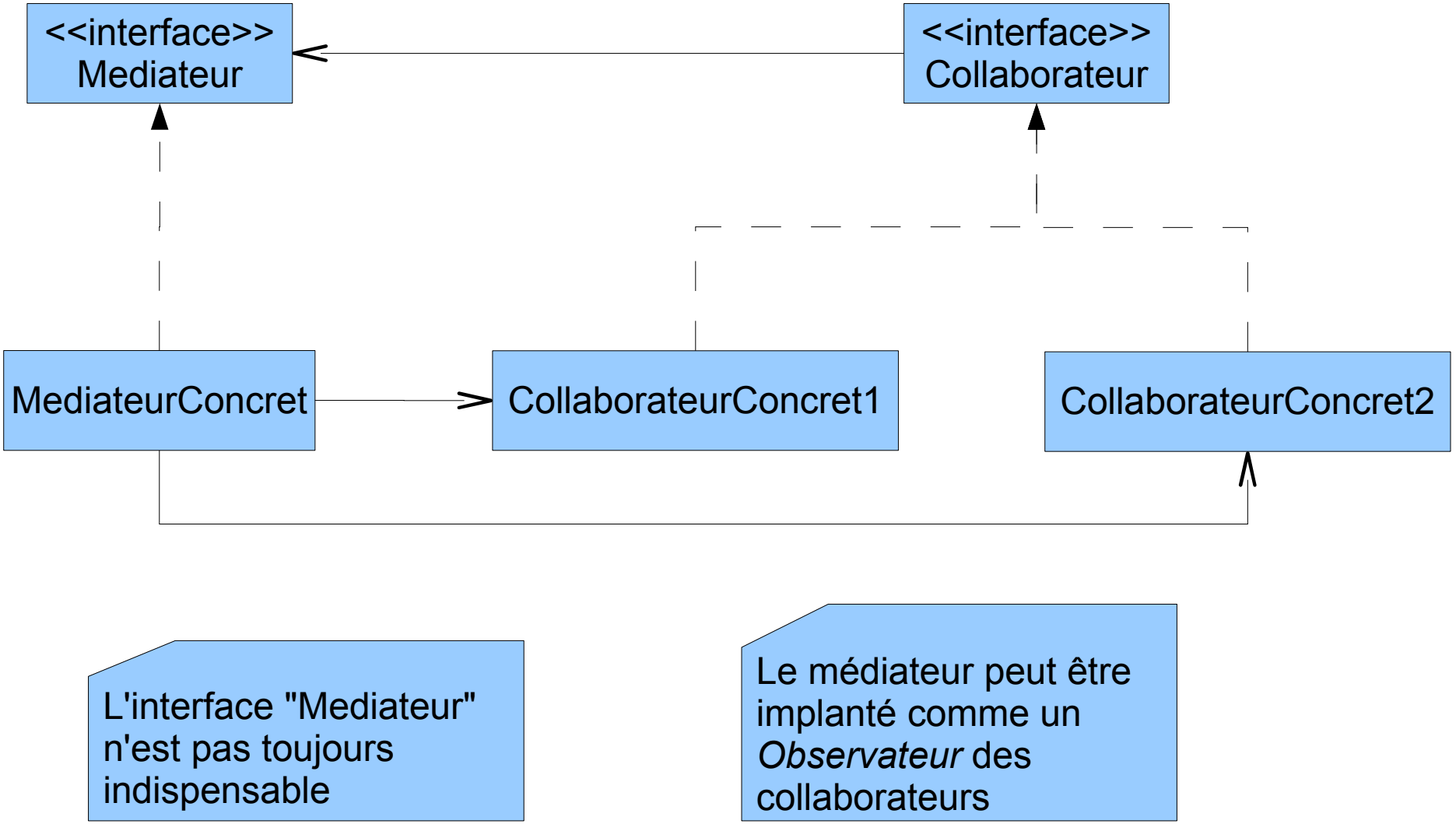
Itérateur (Iterator) – 2



Médiateur (Mediator)

- But
  - Lorsque plusieurs objets doivent interagir les uns avec les autres, centraliser les interactions dans une seule classe pour rendre les objets plus indépendants
- Exemple
  - Classe java.awt.CheckboxGroup pour gérer les boutons radios
- Principe
  - Chaque objet est enregistré auprès du médiateur
  - A chaque changement d'état, les objets préviennent le médiateur
  - Le médiateur modifie les états des autres objets si nécessaire

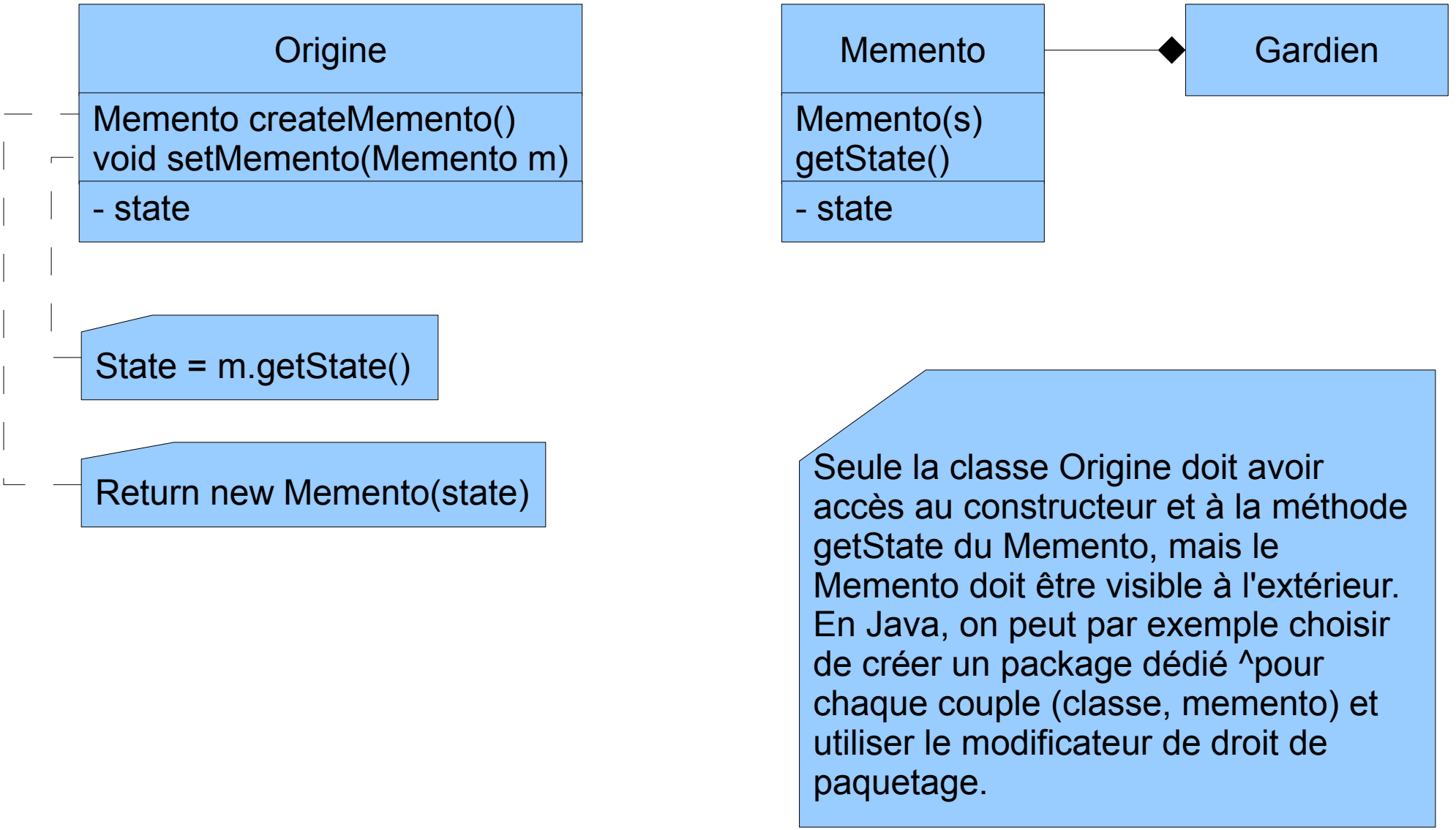
Médiateur (Mediator) – 2



Memento

- But
  - Sans viole le principe d'encapsulation, capturer et sauver l'état interne d'un objet pour pouvoir le restaurer plus tard
- Application
  - Les systèmes Undo/Redo de nombreuses applications

Memento – 2


Origine
Memento createMemento()
void setMemento(Memento m)
- state
State = m.getState()
Return new Memento(state)
Memento
Memento(s)
getState()
- state
Gardien
Seule la classe Origine doit avoir accès au constructeur et à la méthode getState du Memento, mais le Memento doit être visible à l'extérieur. En Java, on peut par exemple choisir de créer un package dédié ^pour chaque couple (classe, memento) et utiliser le modificateur de droit de paquetage.

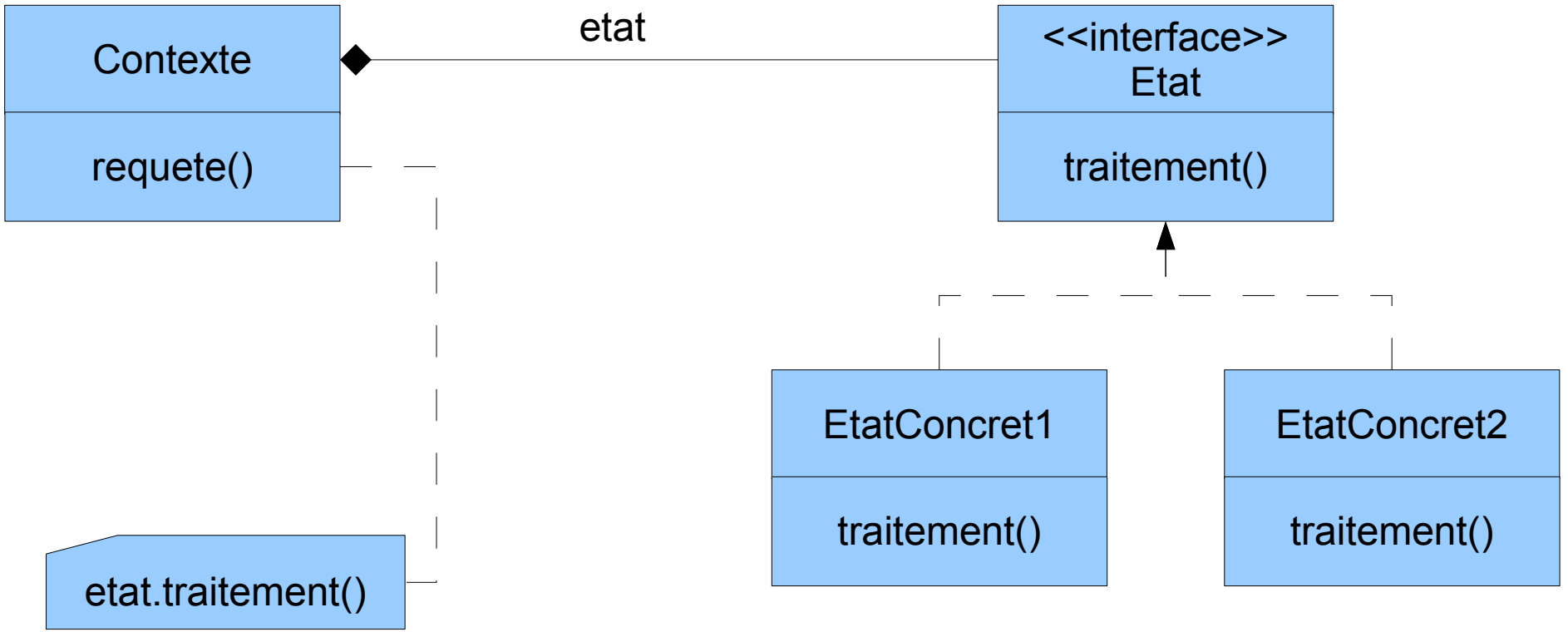
Observateur (Observer)

Voir Cours de base

État
(State)

- But
  - Permettre à un objet de modifier son comportement en fonction de son état
- Solution
  - Implanter les parties modifiables selon l'état dans des classes différentes implantant une interface commune

État (State) – 2



Les changements d'état peuvent être implantés directement dans les états ou bien dans l'objet Contexte.

Stratégie
(Strategy)

Voir Cours de base

Patron de méthode (Template Method)

Voir Cours de base

Visiteur
(Visitor)

- But
  - Représenter une opération qui doit être effectuée sur tous les objets internes d'une structure.
  - Permettre d'ajouter de nouveaux traitements sans modifier les classes de la structure.
- Solution
  - Un parcours générique est défini récursivement dans la structure
  - Une classe Visiteur est créée pour chaque traitement
  - Chaque Visiteur définit une méthode par type d'élément de la structure

Visiteur (Visitor) – 2