Les « Design Patterns »

Bruno Mermet
Université du Havre
2020

Introduction

Origine

Design Patterns, Elements of Reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1994.

Définition

Un design pattern est un "motif de conception". Il est là pour répondre à un problème récurrent. Il ne s'agit pas d'une solution toute faite mais d'une architecture générale

Caractéristiques essentielles

Nom

Problème

Solution

Exemples

Classification

(d'après A. Beugnard, ENST Bretagne)

	Créateurs	Structuraux	Comportementaux
Classe	Fabrication	Adaptateur (class)	Interpréteur Patron de méthode
Objet	Fabrique abstraite Constructeur Prototype Singleton	Adaptateur (objet) Pont Composite Decorateur Facade Poids mouche Proxy	Chaîne de responsabilité Commande Iterateur Mediateur Memento Observateur Etat Stratégie Visiteur

Un premier exemple...

Le Designe Pattern *Façade*

Pattern Façade (1)

Objectif

- simplifier l'utilisation d'un système existant
- avoir sa propre interface

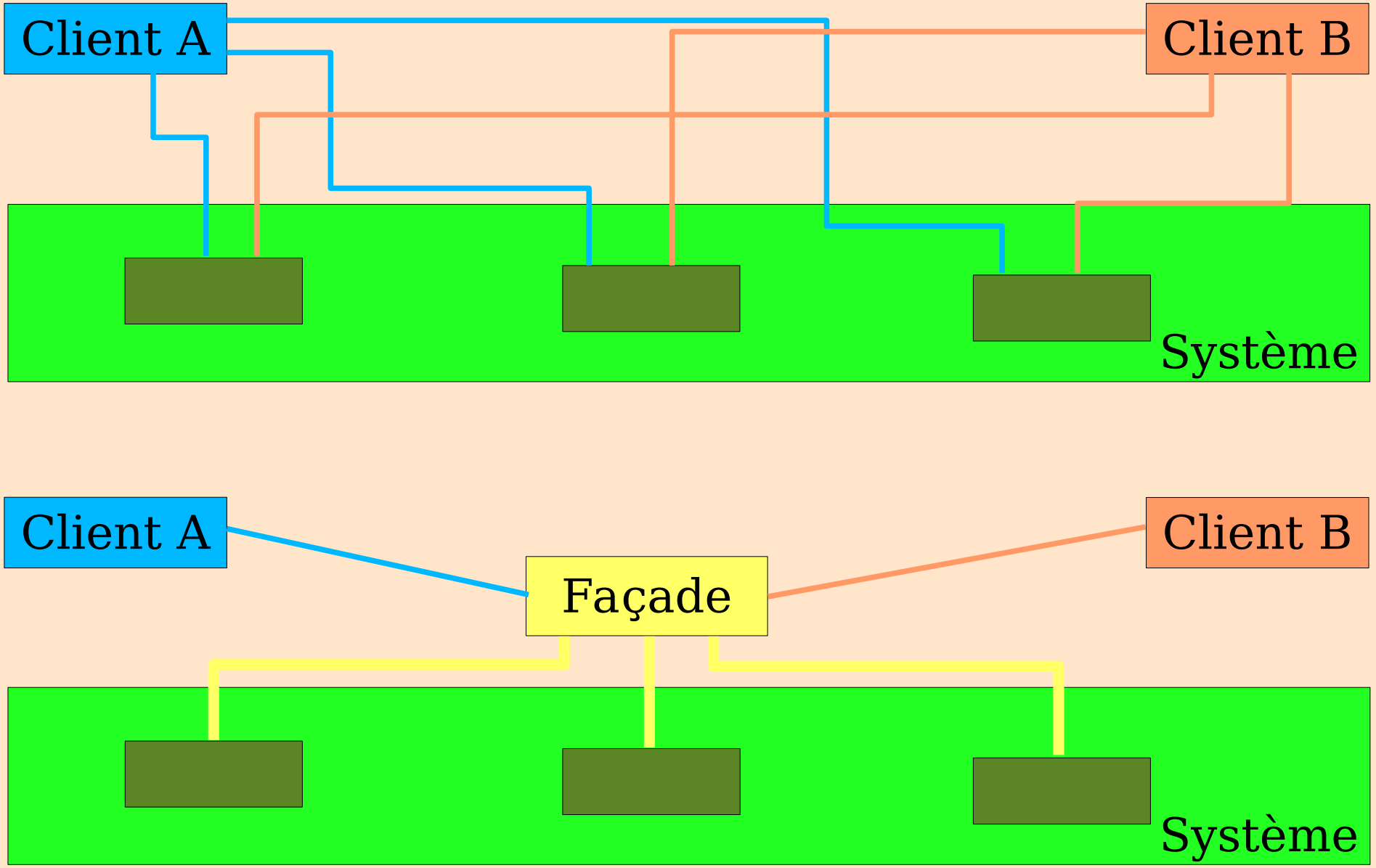
Problème

- besoin limité à un sous-ensemble d'un système complexe
- manipuler un système uniquement d'une façon spécifique

Solution

- façade fournit une nouvelle interface au système existant

Pattern Façade (2)



Façade – exercice du TP

- On veut proposer une classe « Pile » simple à utiliser (4 méthodes), sachant que la classe `ArrayDeque` propose les méthodes nécessaires, mais en propose beaucoup plus que nécessaires (une cinquantaine)

Patterns de création

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

Pattern Fabrique Abstraite (1)

Objectif

- Utiliser des familles ou jeux d'objets pour des clients/cas donnés

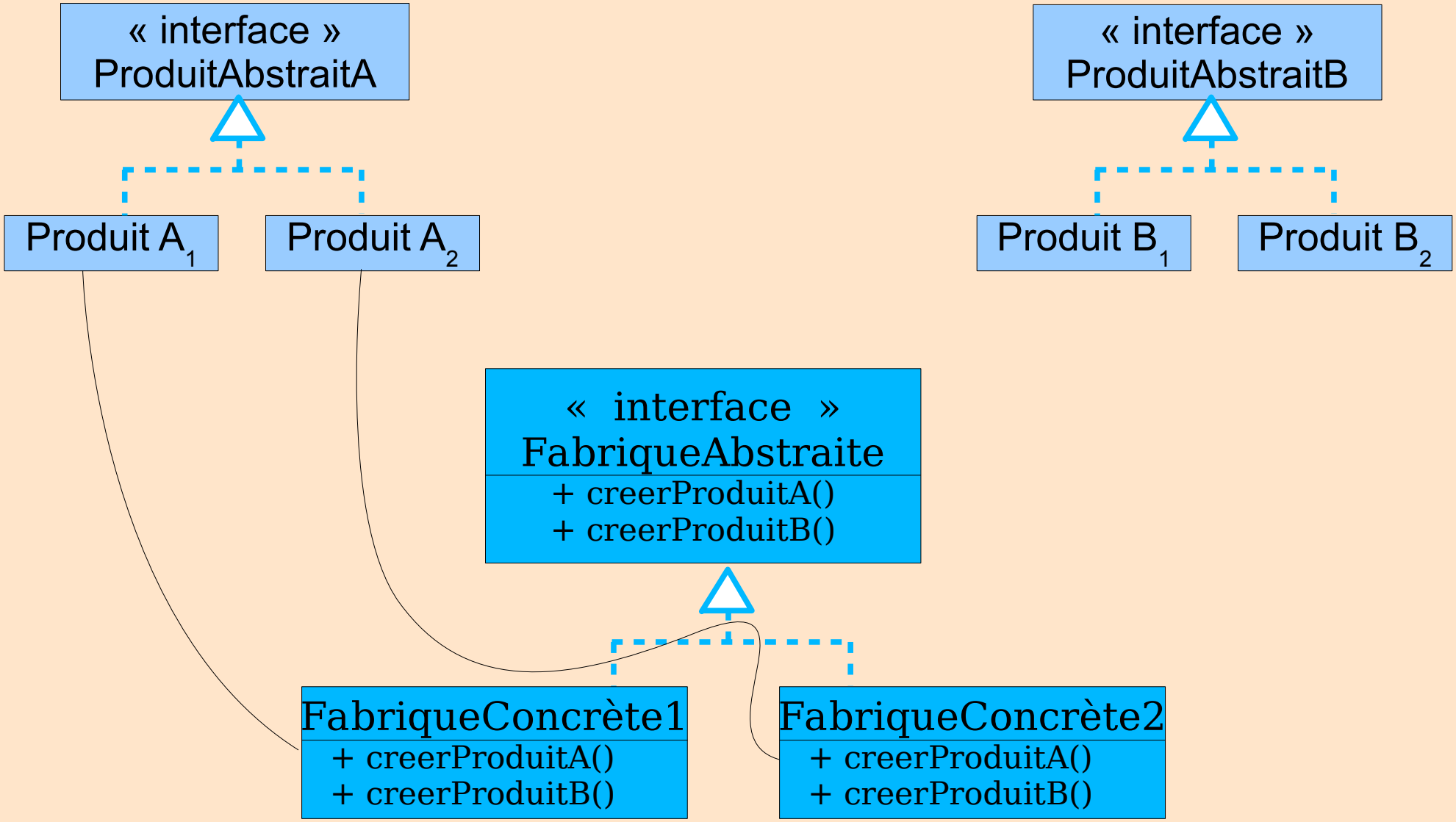
Problème

- Des familles d'objets associés doivent être instanciés

Solution

- Coordonner la création de familles d'objets
- Extraire les règles d'instanciation de l'objet client

Pattern Fabrique Abstraite (2)



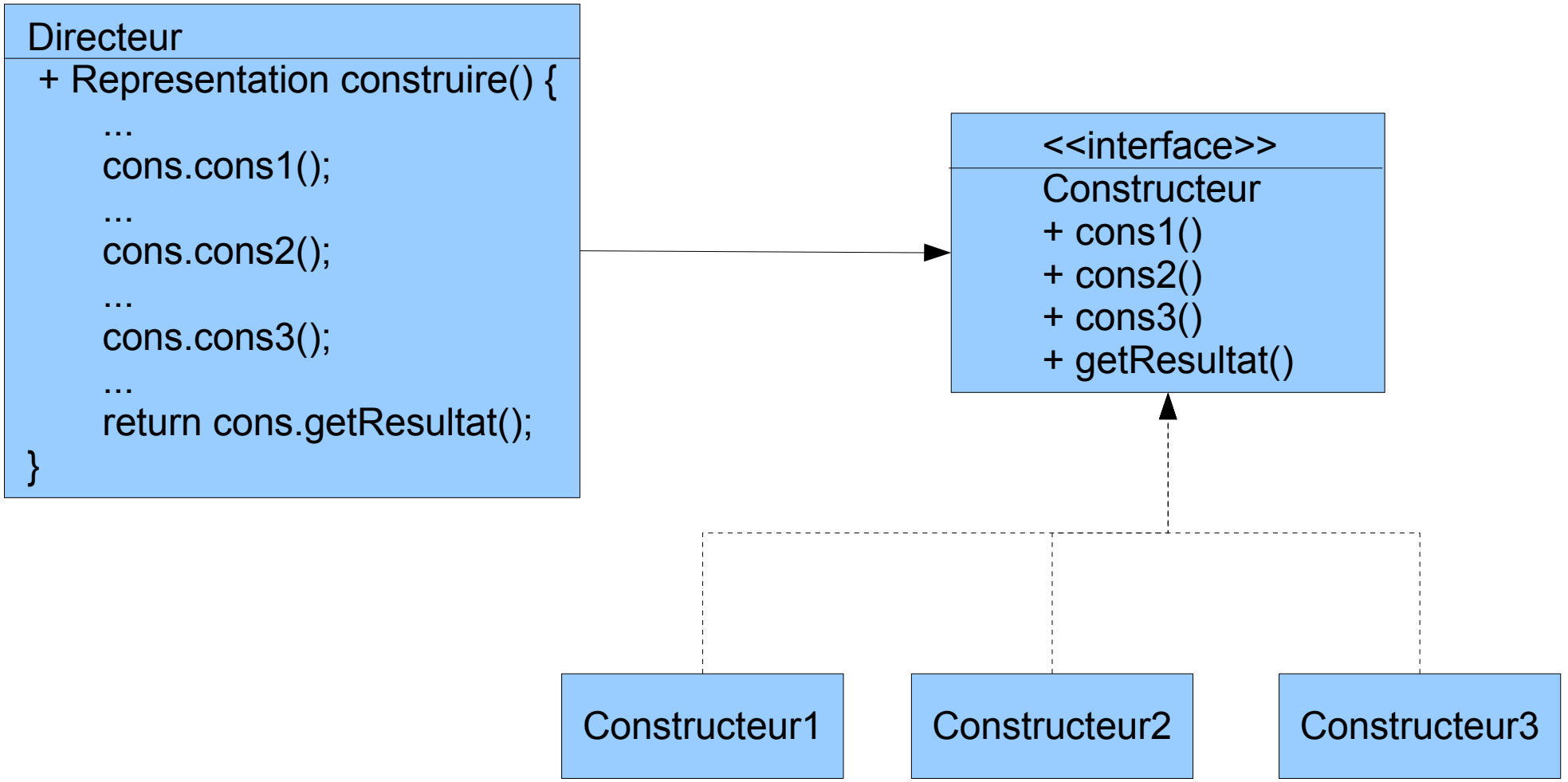
Fabrique abstraite – exercice du TP

- On doit sauvegarder des listes de joueurs et de jeux, et la sauvegarde peut être effectuée soit en XML, soit en JSON, avec des classes dédiées (joueur → XML, joueur → JSON, jeu → XML, jeu → JSON), et on veut que la sauvegarde soit uniforme (tout en XML ou tout en JSON)

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

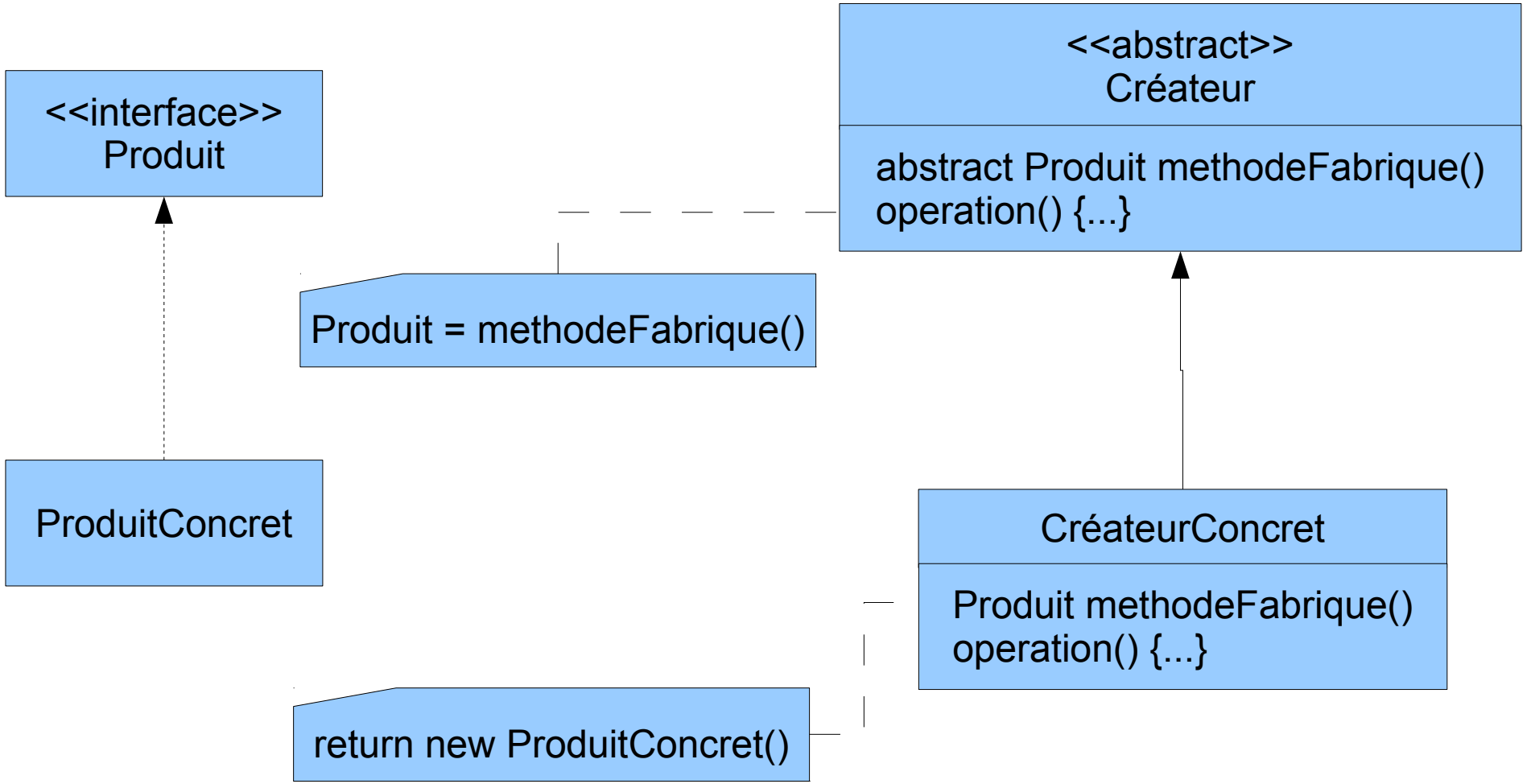
Constructeur (Builder) – 2



Méthode Fabrique (Factory Method)

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

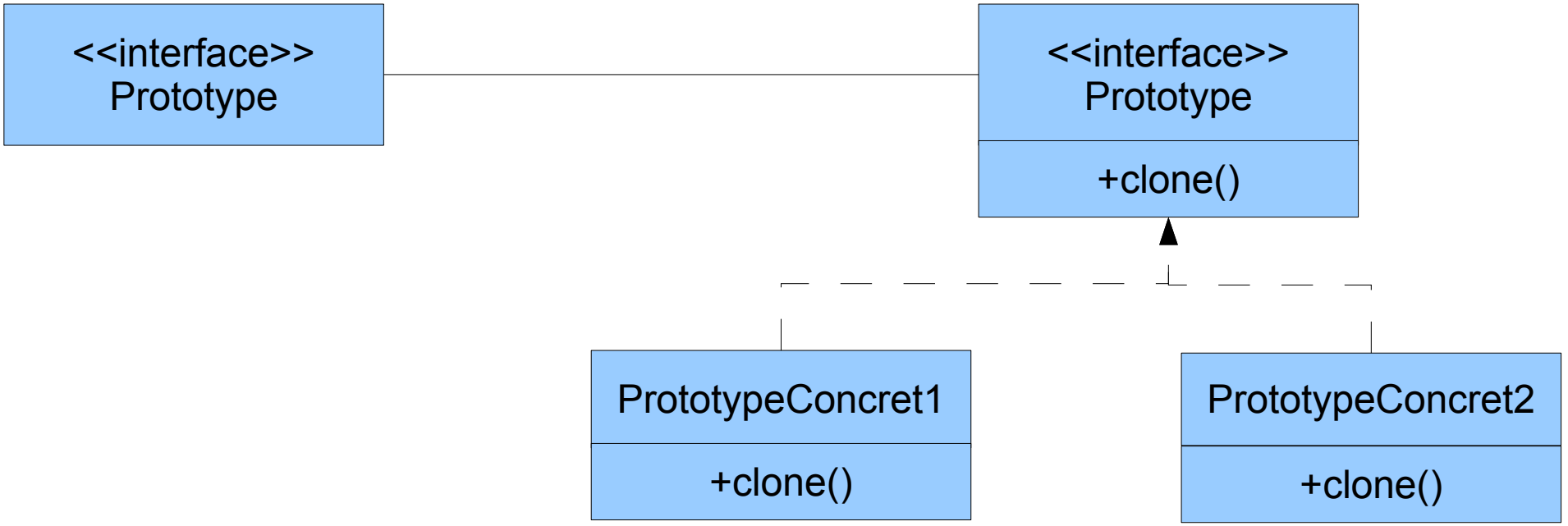
Méthode Fabrique (Factory Method) – 2



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()

Pattern Singleton (1)

Objectif

- Garantir qu'une classe n'est pas instanciée plus d'une fois

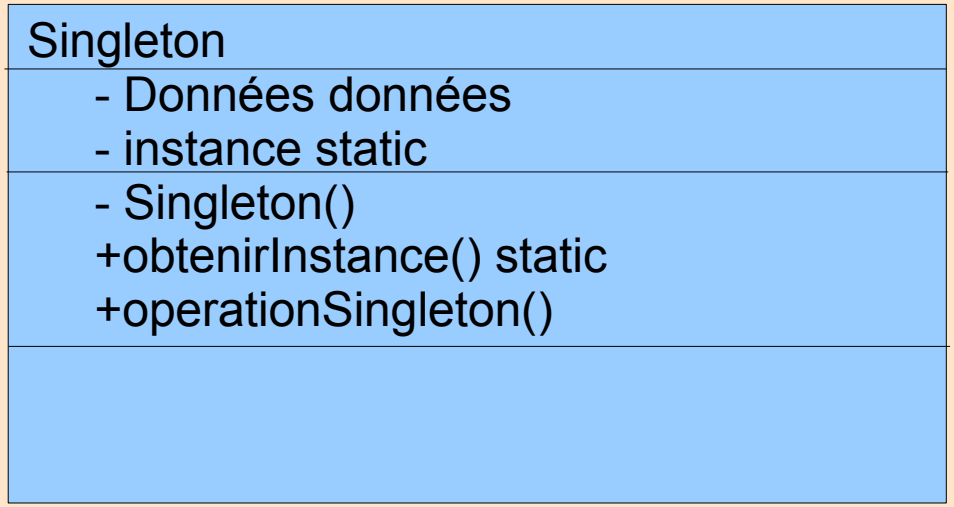
Problème

- Deux objets doivent se référer à la même chose
- Être sûr qu'il n'en existe qu'une instance

Solution

- S'assurer qu'il n'existe qu'une instance

Pattern Singleton (2)



En Java (version non thread-safe) :

```
public class Singleton {
  private Données données;
  private static Singleton instance = null;
  private Singleton() {
     ...
  }
  public static Singleton getInstance() {
    if (instance == null) {
      {instance = new Singleton();}
    return instance;
  }
}
```

Pattern Singleton (3)

- Version Java « Thread-Safe »

```
public class Singleton {
  private Donnees donnees;
  private Singleton() {...}
  private static class Detenteur {
    private static final Singleton instance = new
Singleton();
  }
  public static Singleton getInstance() {
    return Detenteur.instance;
  }
}
```

Patterns structuraux

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

Pattern Adaptateur (1)

Objectif

Faire correspondre à une interface donnée un objet existant que l'on ne contrôle pas

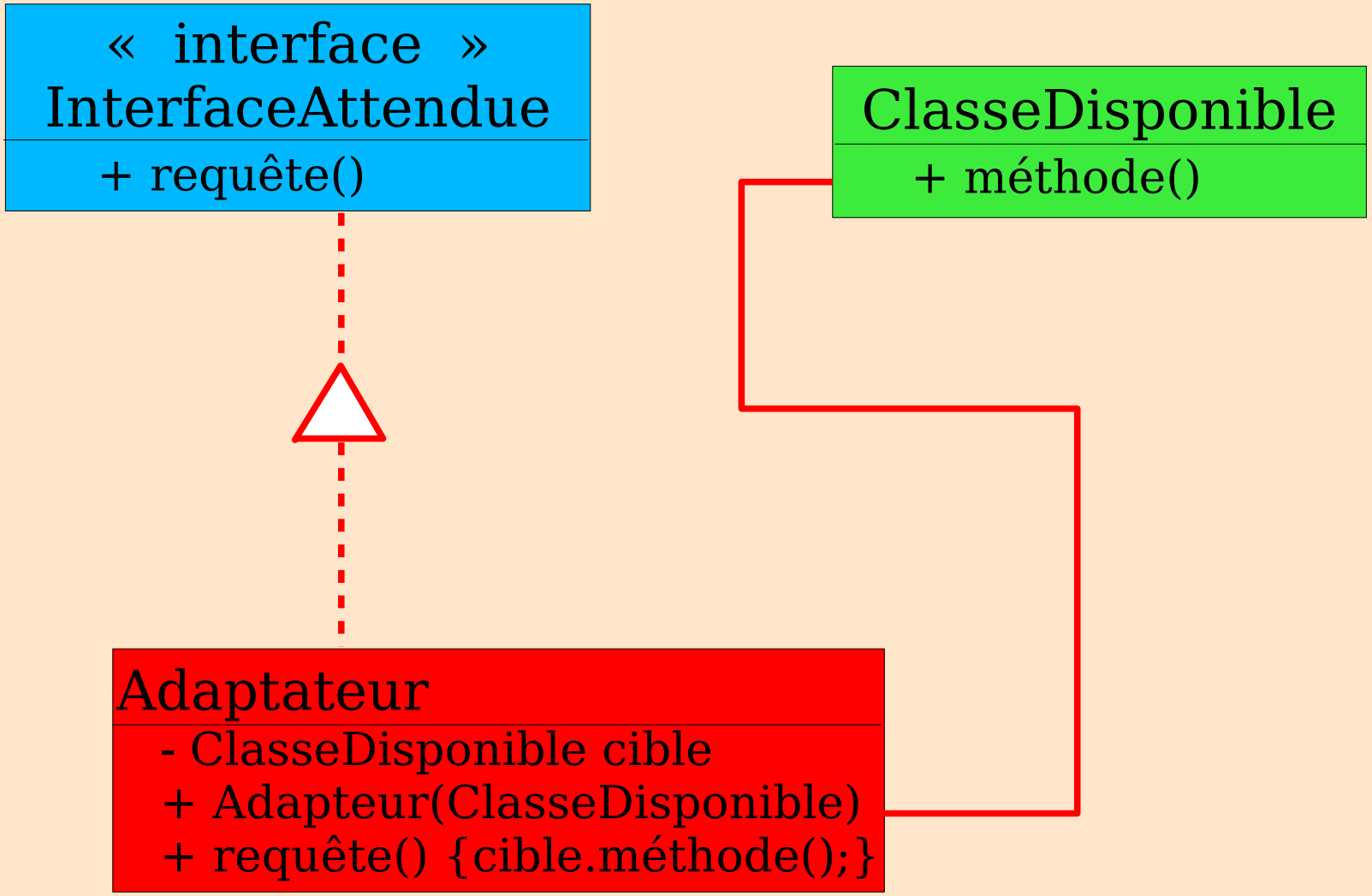
Problème

Le système a les bonnes données et les bonnes méthodes mais une mauvaise interface

Solution

Adaptateur = encapsulateur avec l'interface voulue

Pattern Adaptateur (2)



Adaptateur – exercice du TP

- On utilise une classe Horloge, qui propose de signaler les changements d'heure, et dispose de plusieurs classes permettant d'afficher une heure. On souhaite, sans modifier la classe Horloge ni les différentes classes de visualisation, permettre de disposer des différentes vues possible de l'heure courante

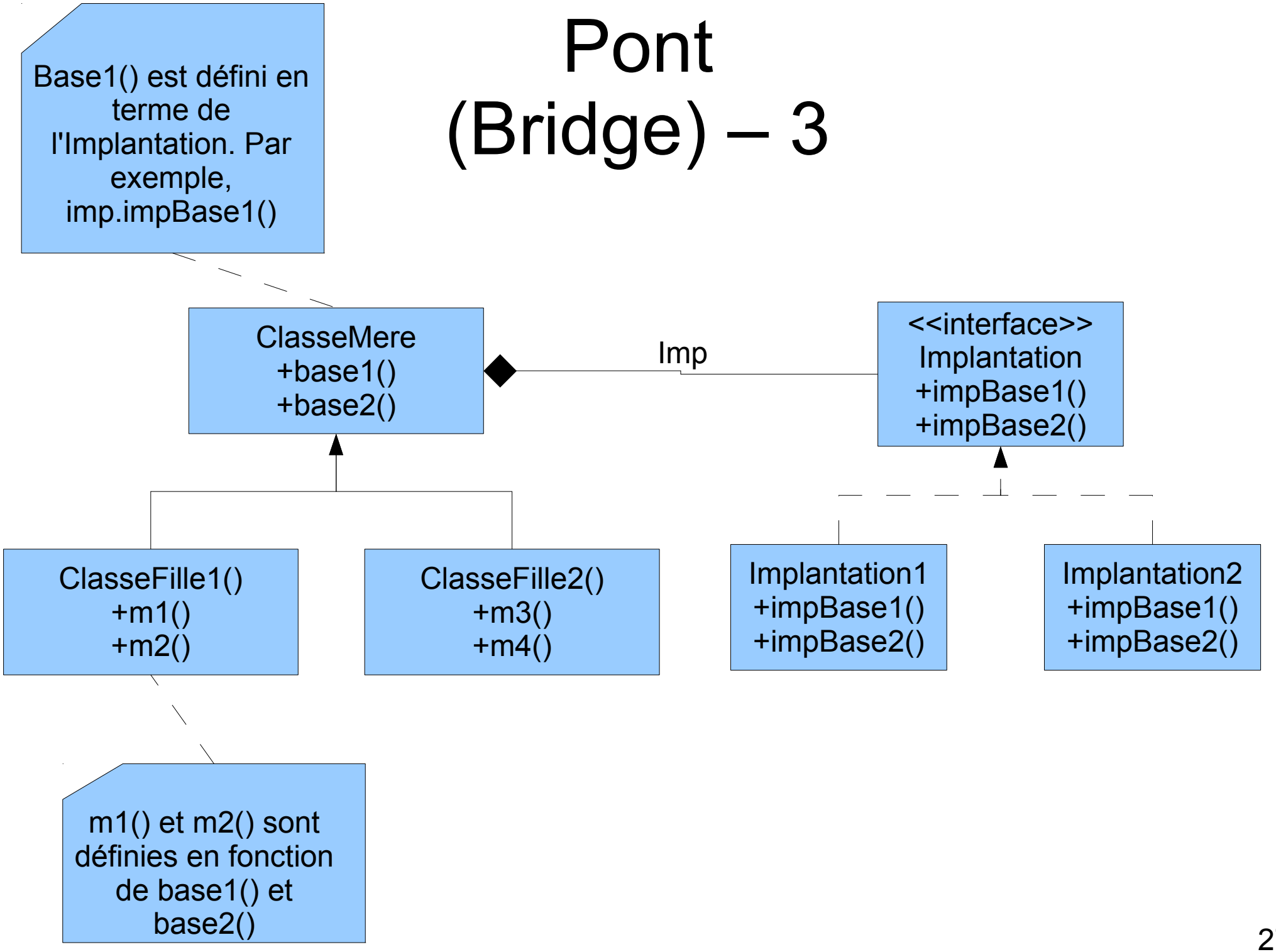
Pont
(Bridge)

- 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

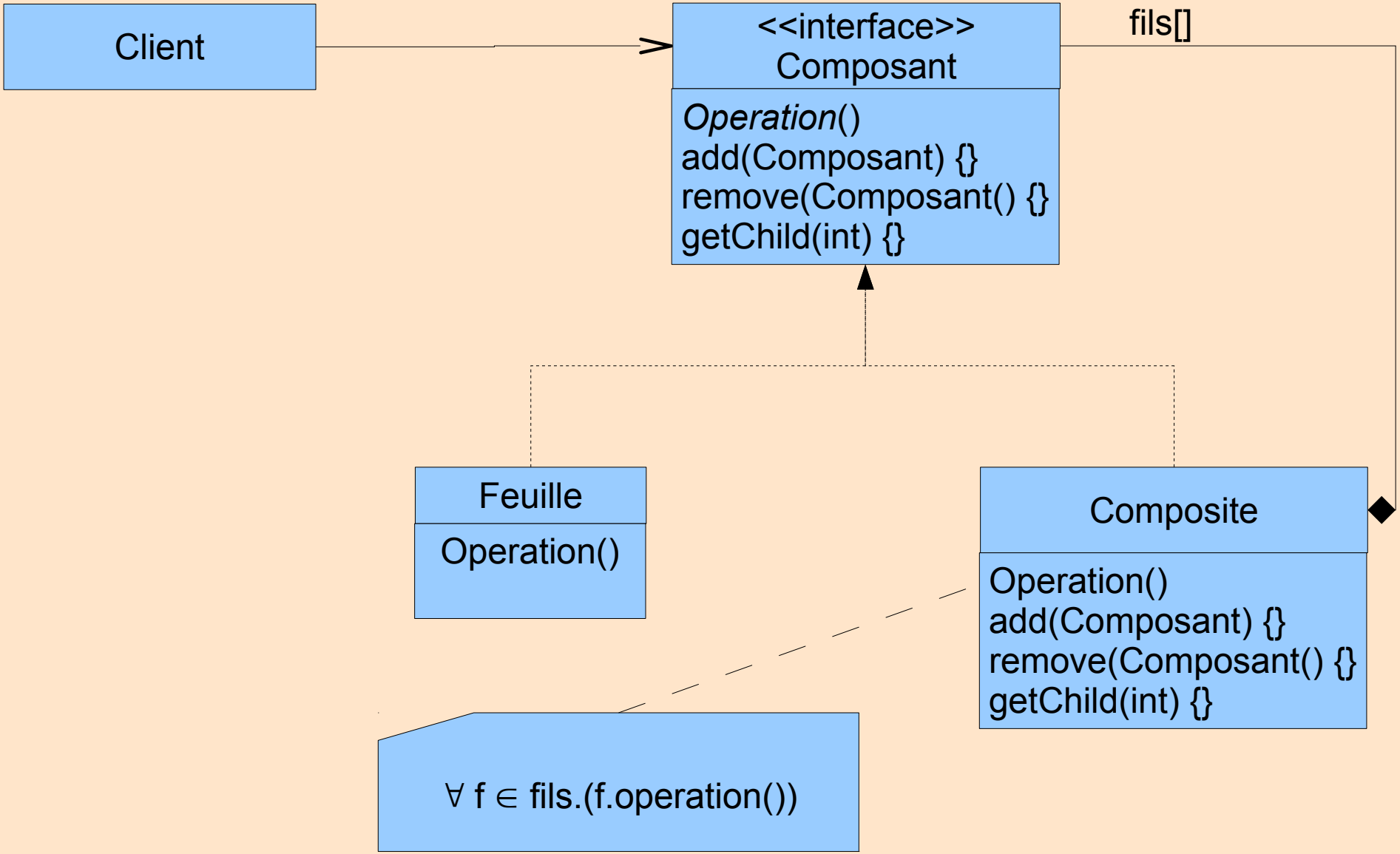
Pont (Bridge) – 3



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

Composite – exercice du TP

- Une application gratuite affiche différents cercle dont on peut modifier certains paramètres (taille, remplissage). On souhaite pouvoir appliquer ces traitements à plusieurs cercles s’ils font partie d’un même groupe

Pattern Décorateur (1)

Objectif

Associer dynamiquement des responsabilités supplémentaires à un objet

Problème

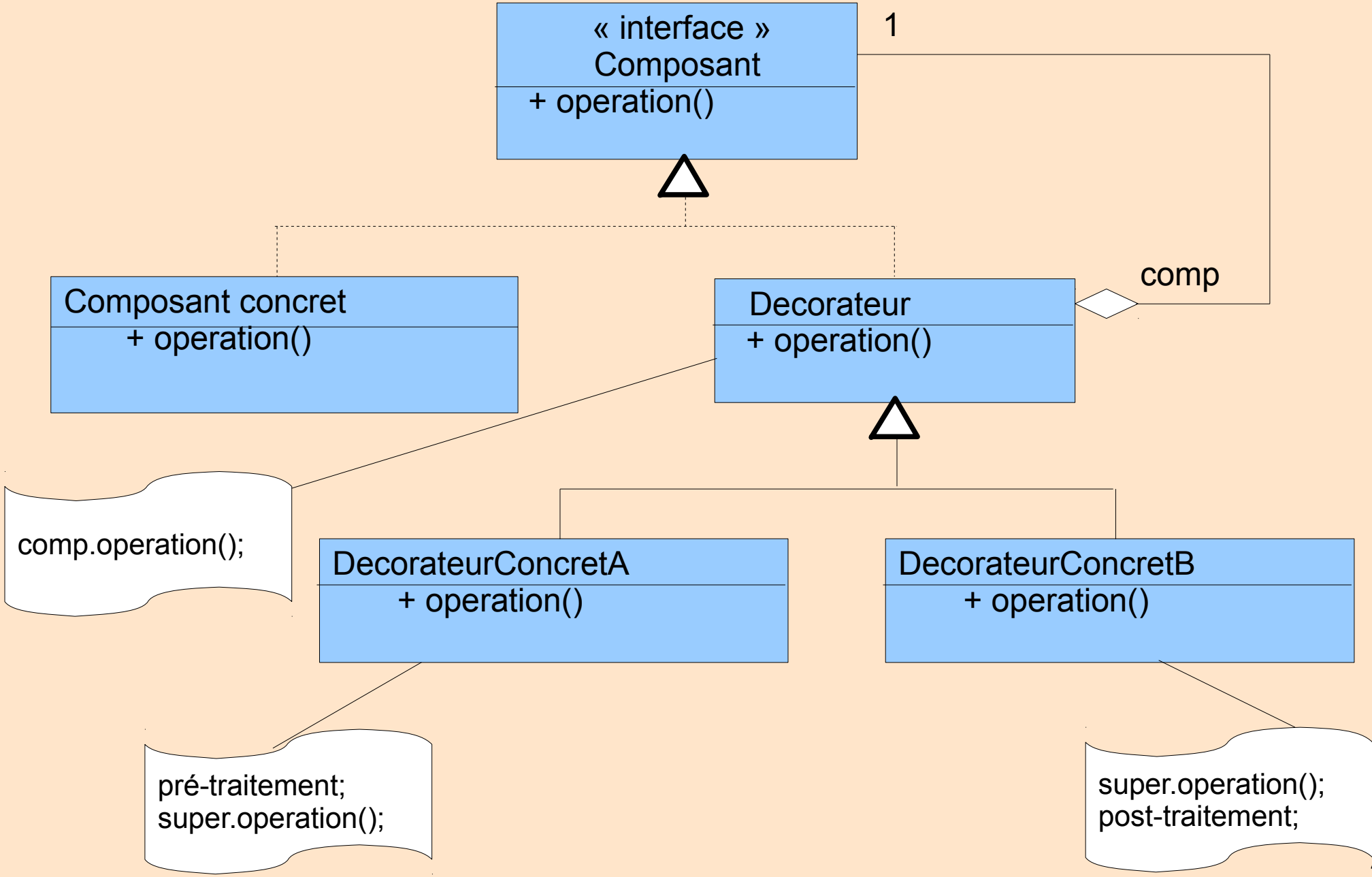
Objet à utiliser possède les fonctions de base recherchées

Nécessité de pouvoir ajouter dynamiquement des fonctionnalités supplémentaires s'appliquant avant ou après

Solution

Permettre l'extension de la fonctionnalité d'un objet sans recourir à des sous-classes

Pattern Décorateur (2)



Décorateur – exercice du TP

- On souhaite pouvoir ajouter des pré-traitements (en-tête par exemple) et des post-traitements (pied-de-page, encadrement) à des documents, à volonté

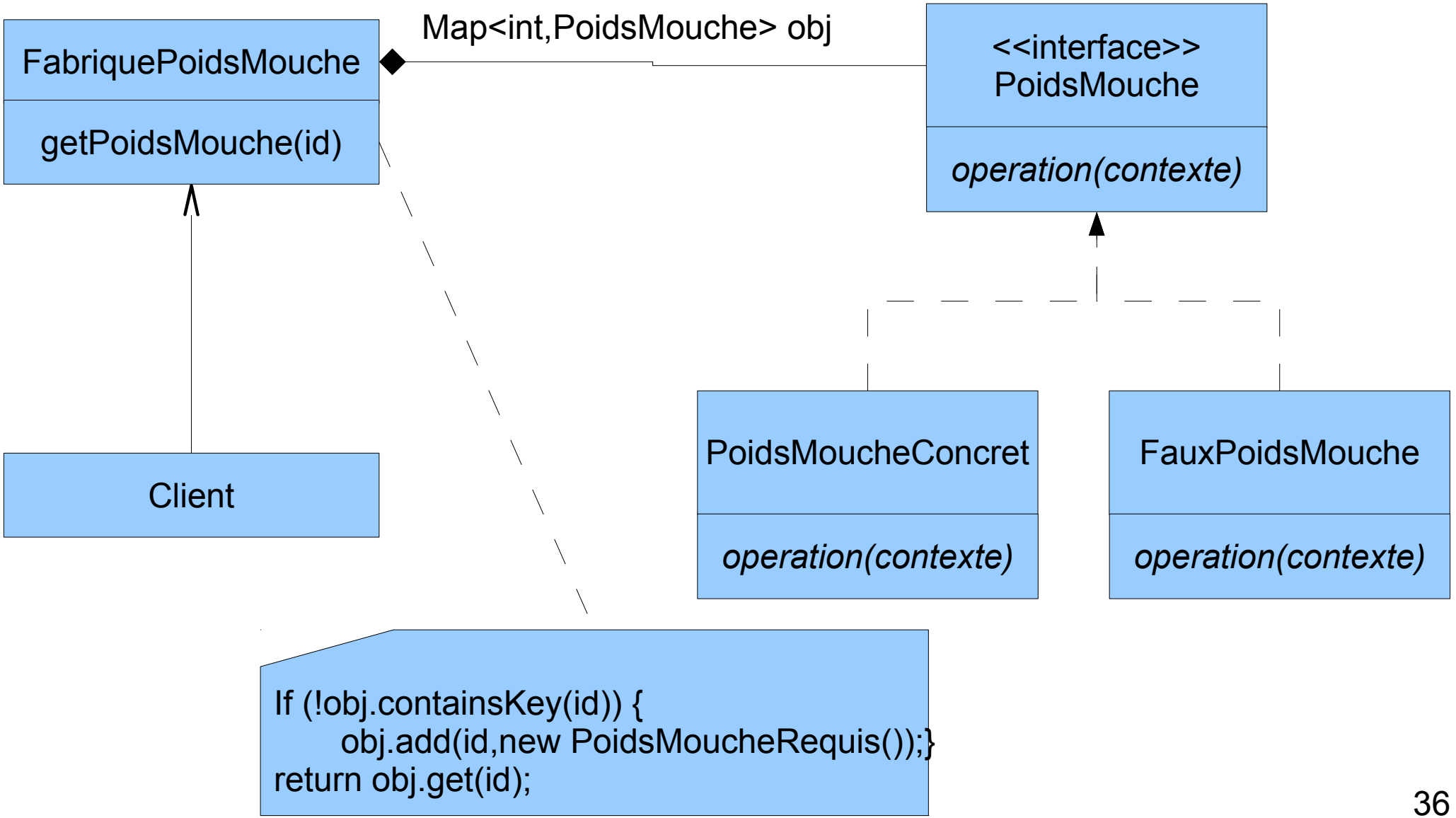
Façade

Voir Introduction

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

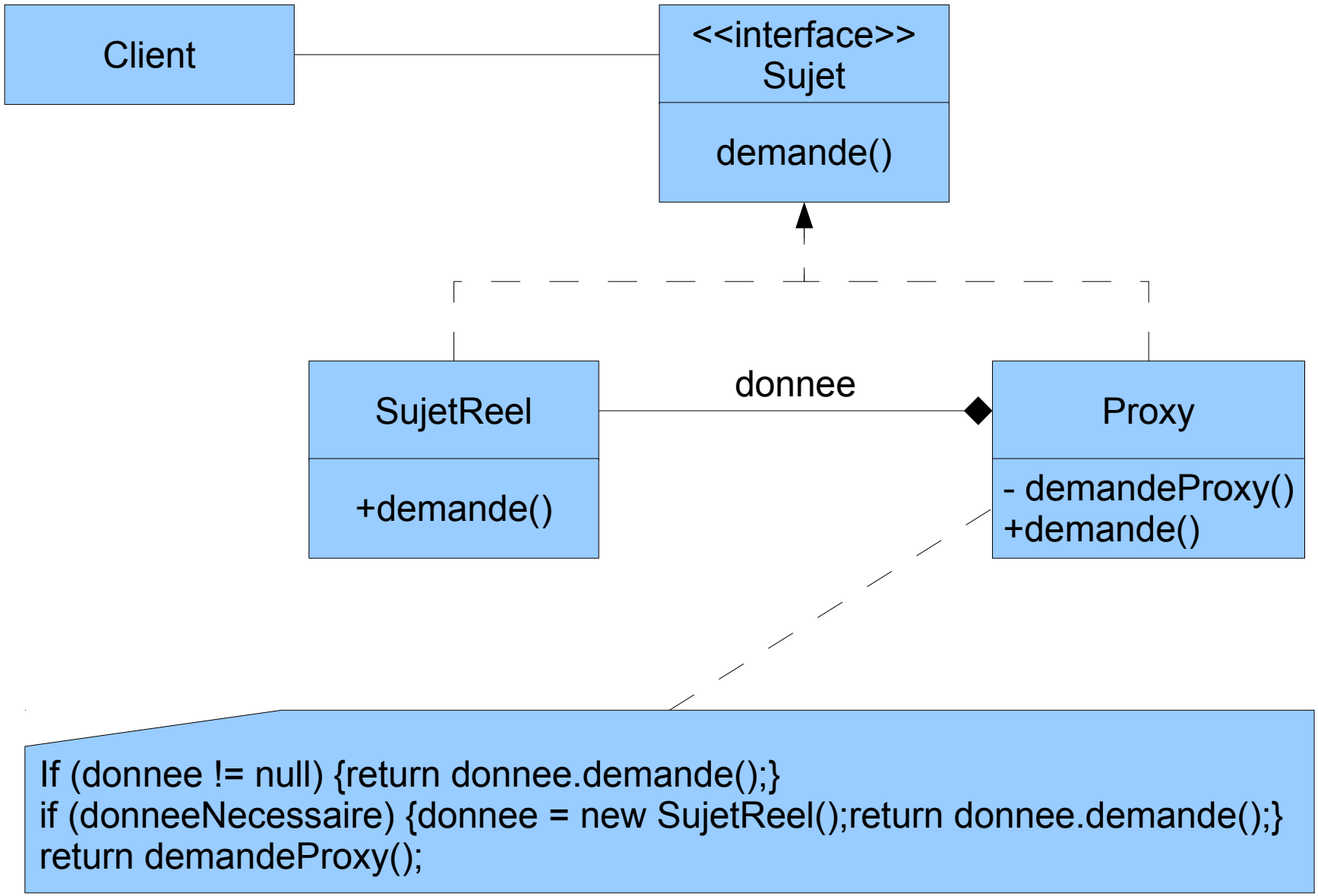
Poids mouche (Flyweight) – 2



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();

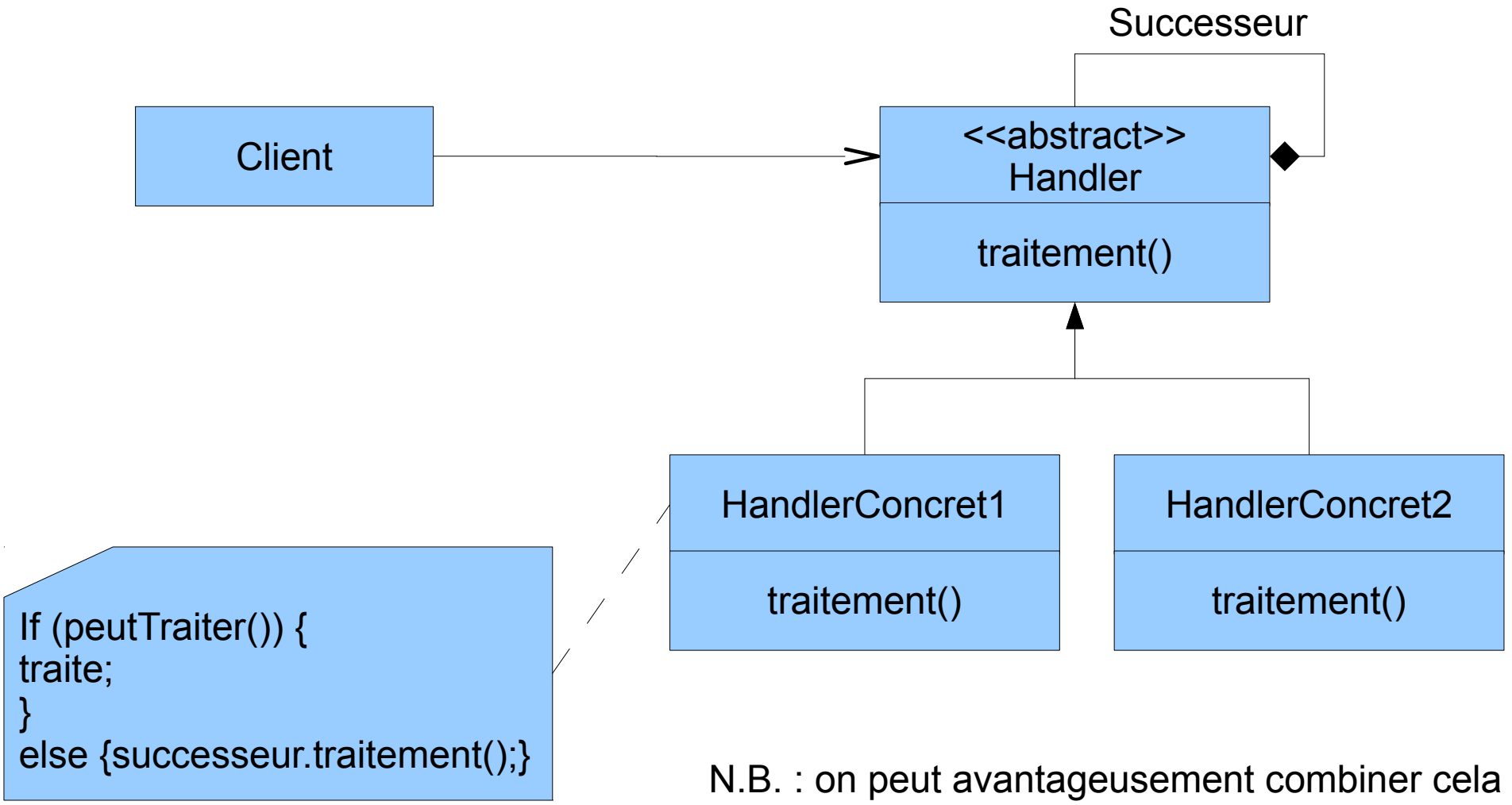
Patterns comportementaux

- 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

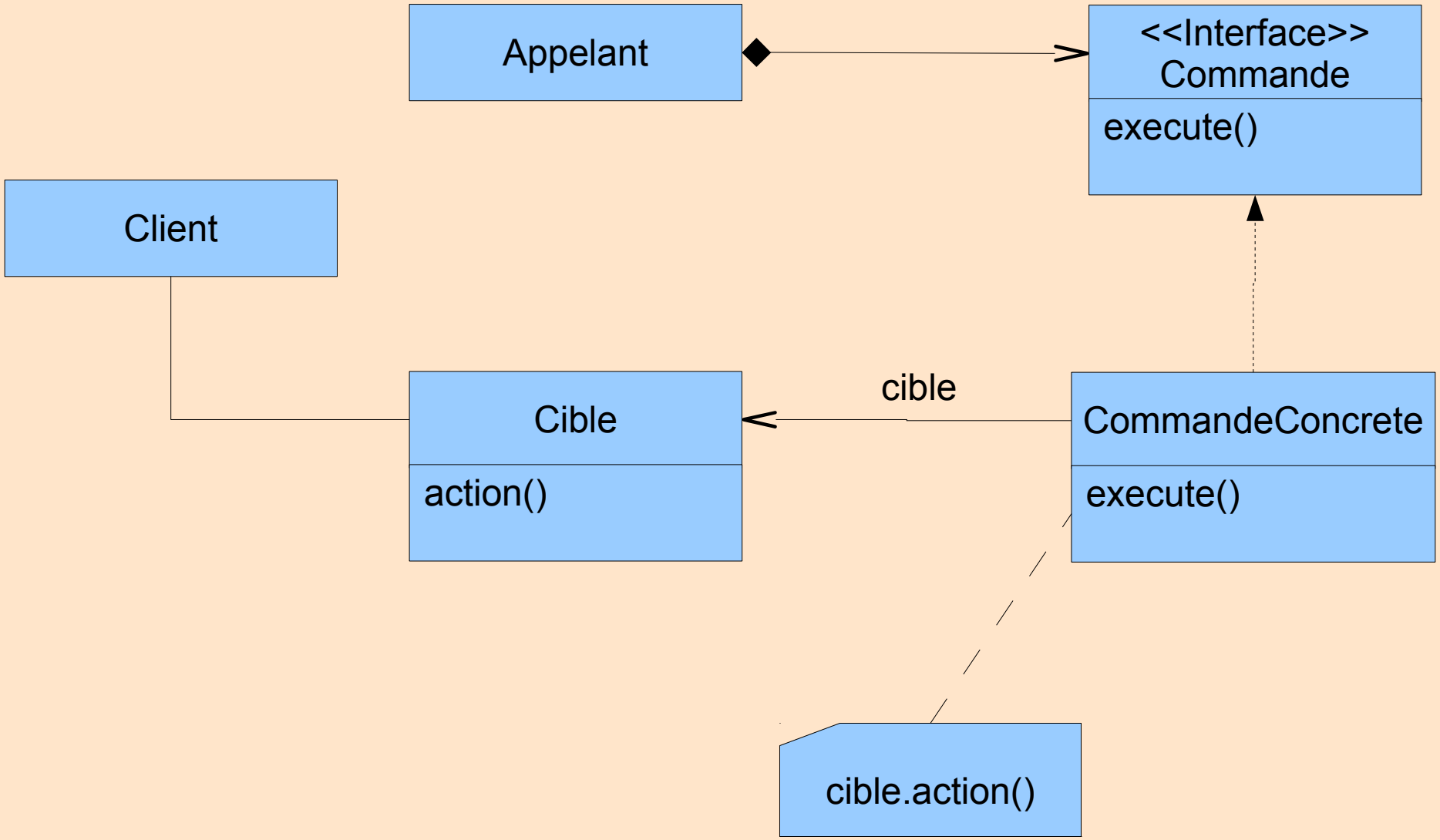


N.B. : on peut avantageusement combiner cela avec le pattern *Patron de méthode*

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

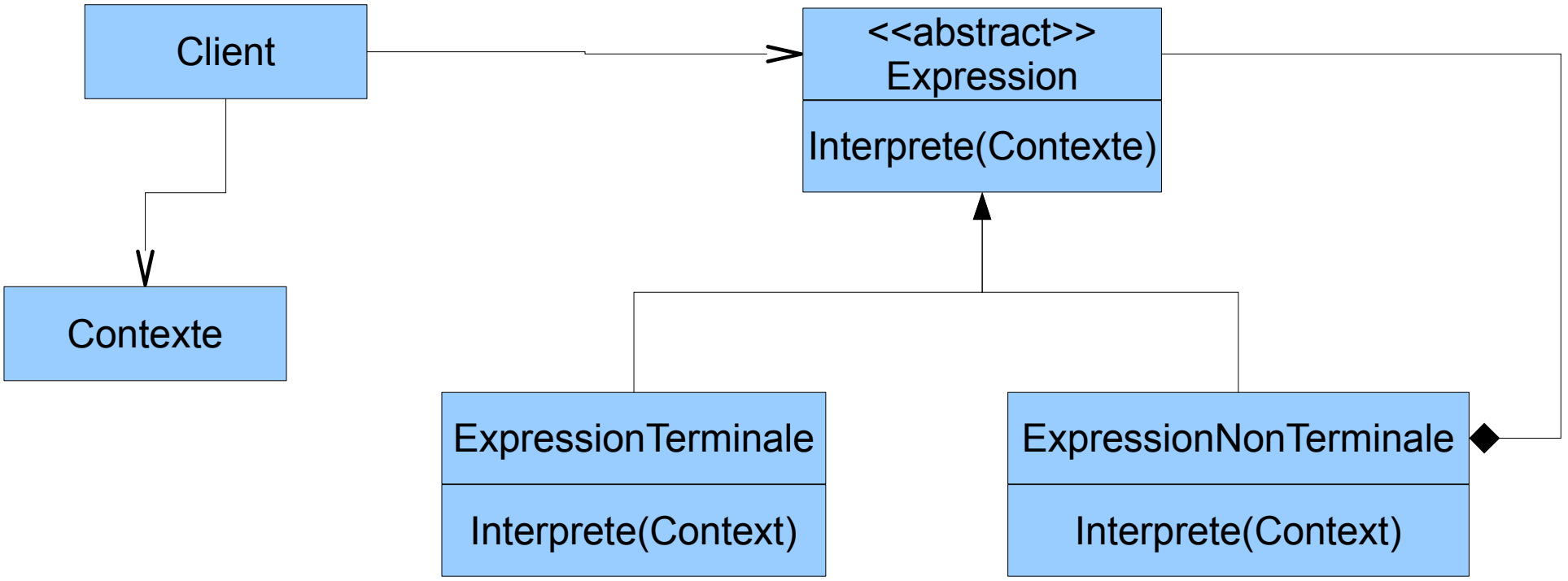
Commande (Command) – 2



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*)

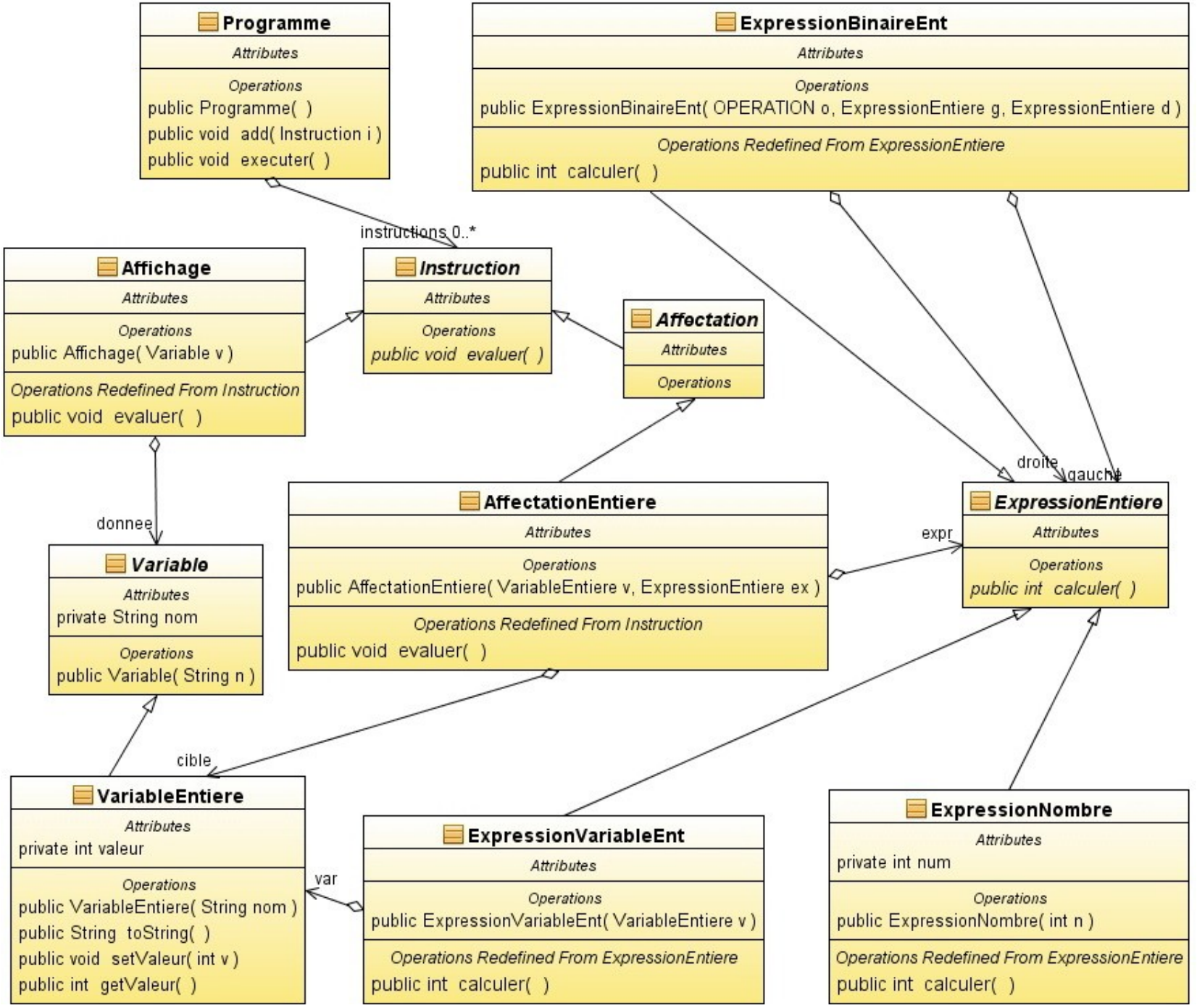
Interpréteur (Interpreter) – 2



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

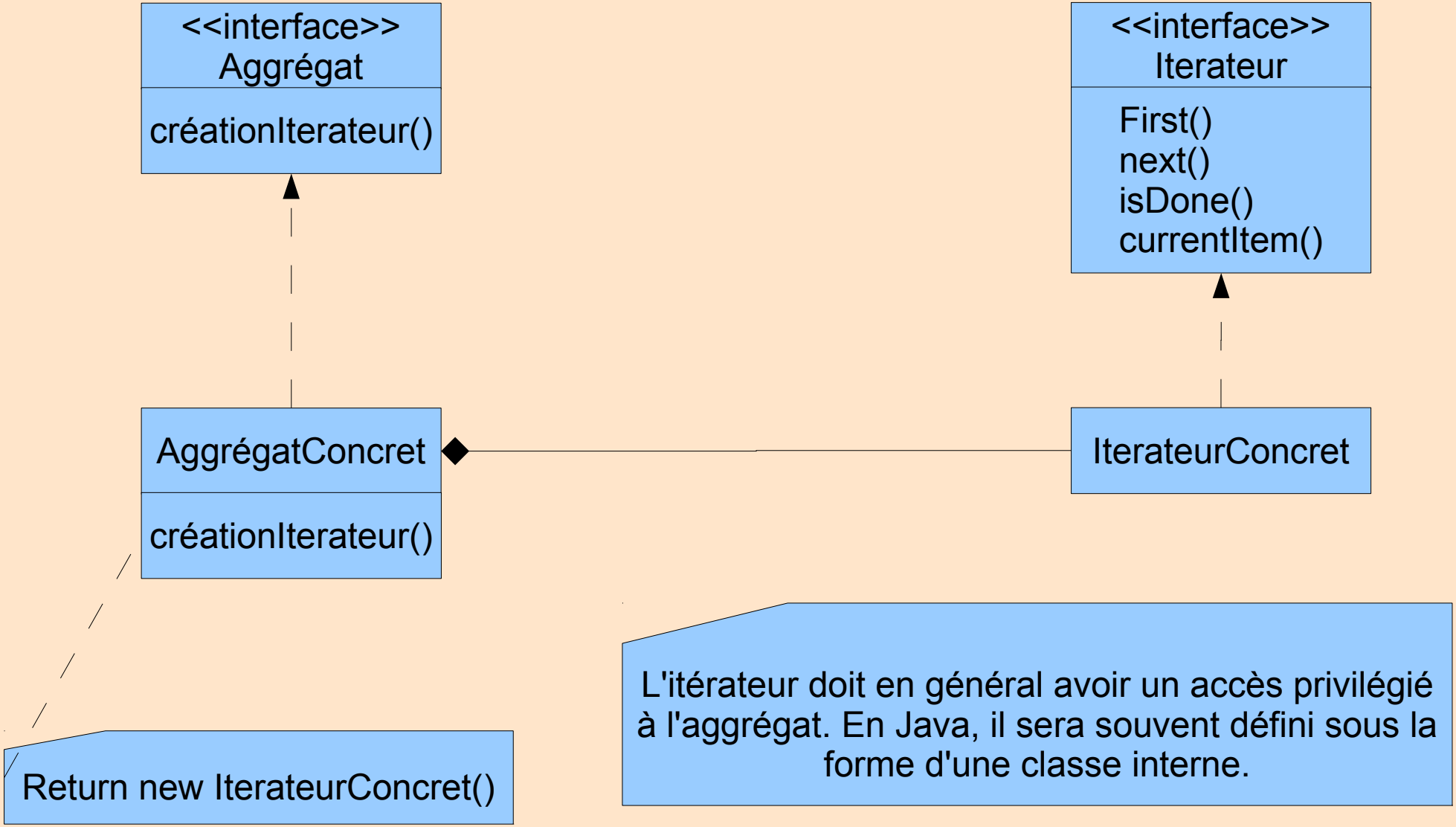
Interpréteur (Interpreter) – 4



Itérateur (Iterator)

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

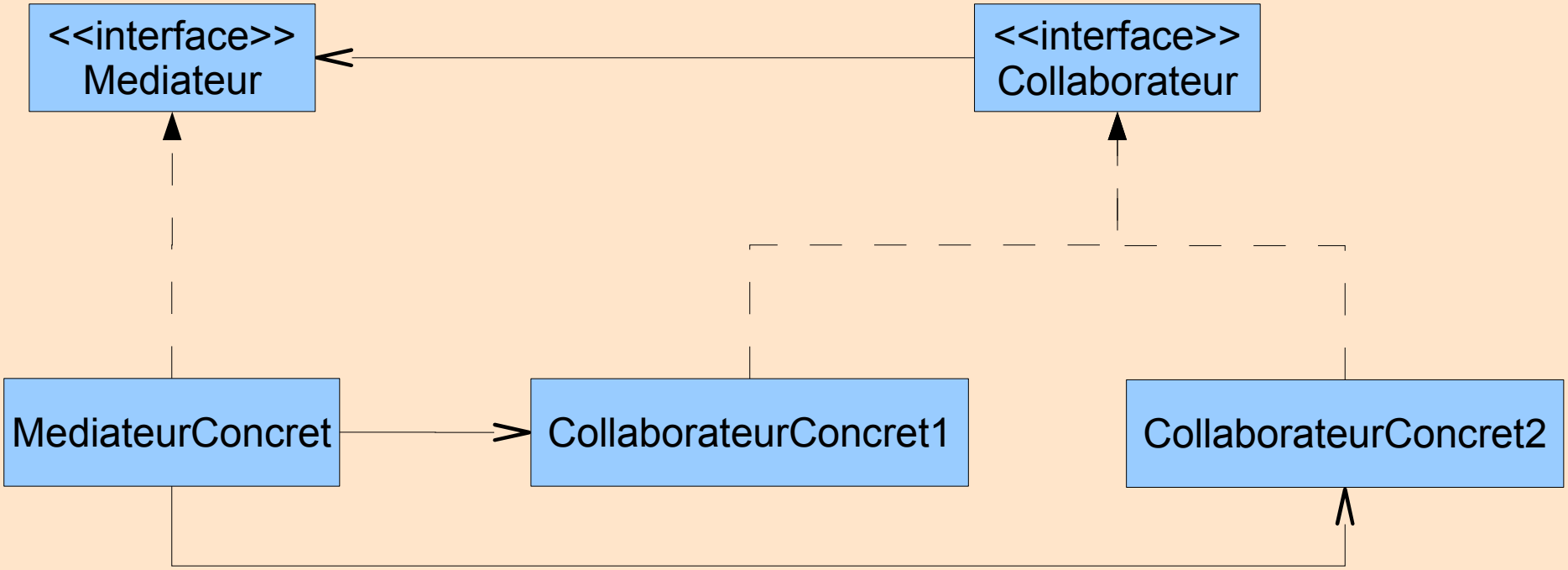
Itérateur (Iterator) – 2



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



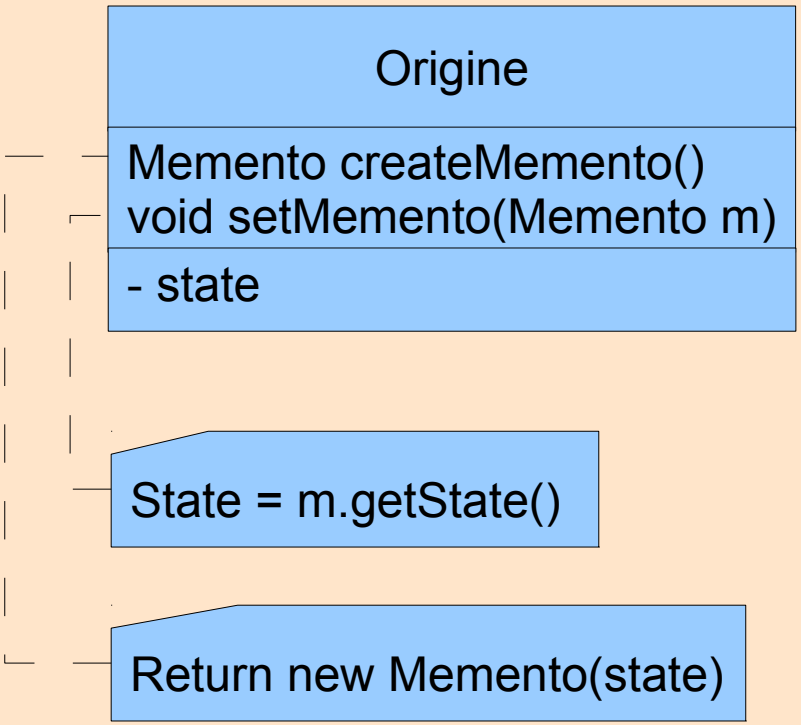
L'interface "Mediateur" n'est pas toujours indispensable

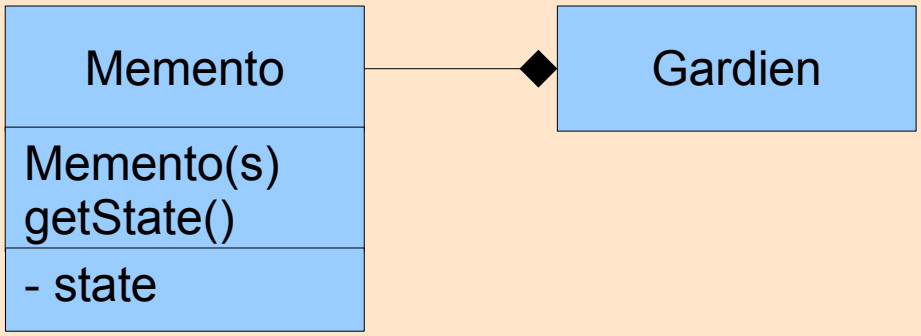
Le médiateur peut être implanté comme un *Observateur* des collaborateurs

Memento

- But
  - Sans violer 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





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.

Memento – exercice du TP

- On veut doter d’un mécanisme type « annuler/refaire » un petit éditeur de texte

Pattern Observateur (1)

Objectif

Etablir une dépendance 1-N entre des objets pour que ceux qui dépendent d'un objet modifié (ou créé) soient avertis du changement d'état et mis à jour automatiquement

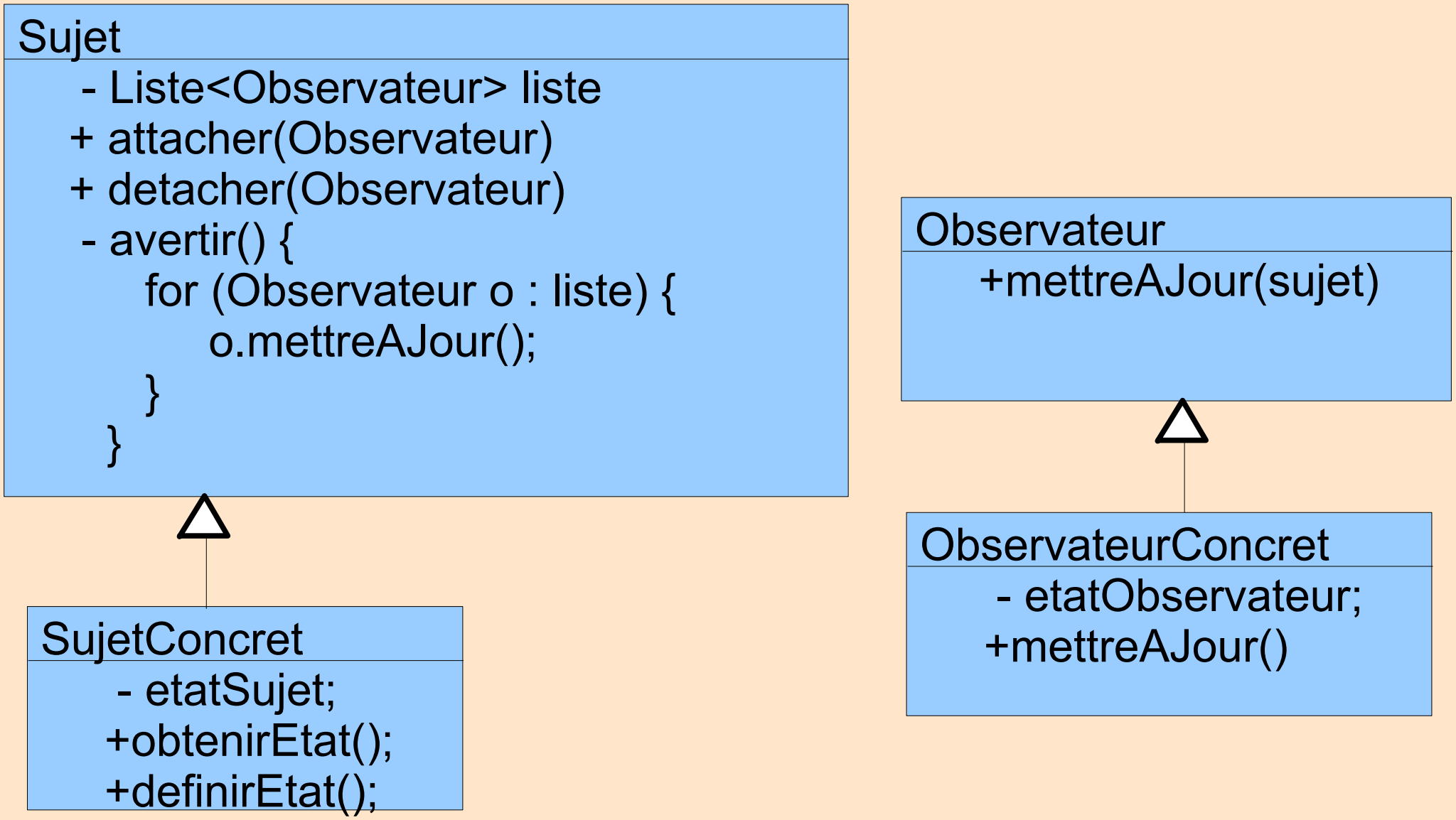
Problème

Avertir une liste variable d'objets qu'un événement a lieu

Solution

Les objets observateurs délèguent la responsabilité de contrôle d'un événement à un objet central, le *sujet*.

Pattern Observateur (2)



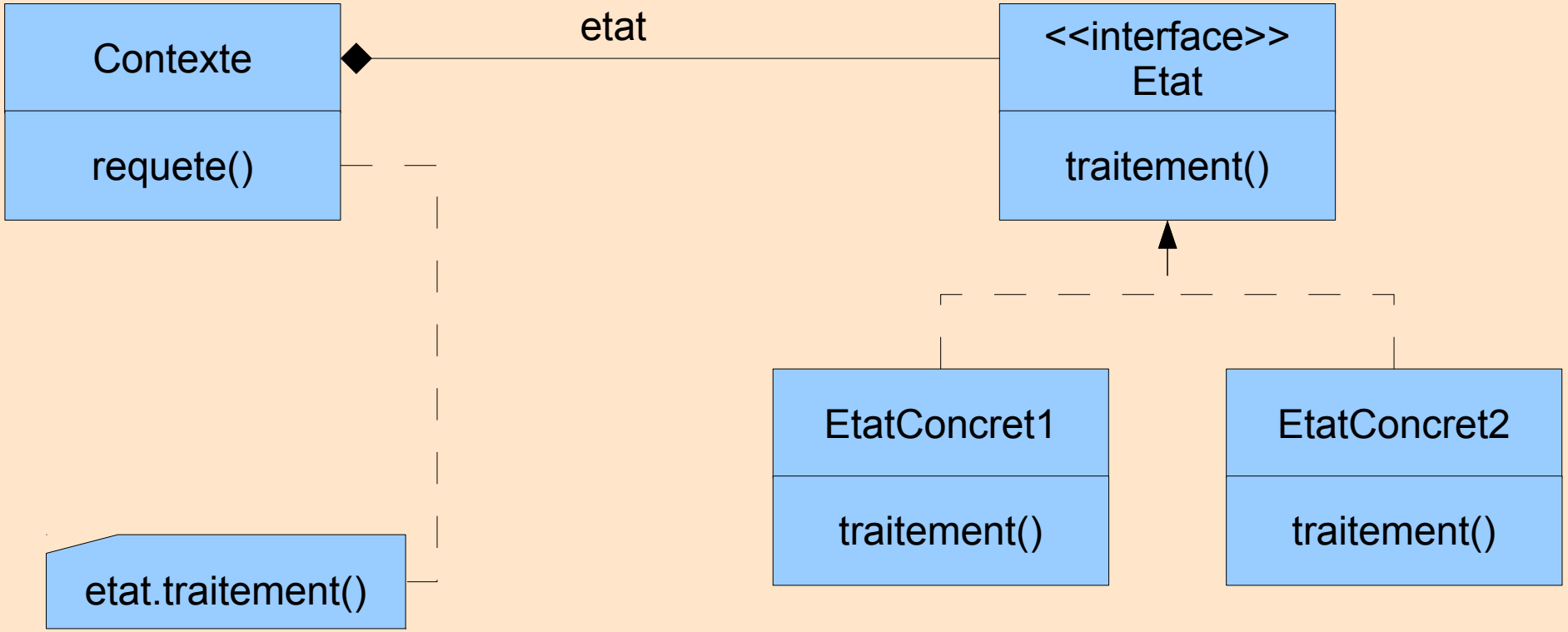
Observateur – exercice du TP

- On souhaite que les modifications de l’heure courante puisse être signalés à des objets intéressés

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.

Pattern Stratégie (1)

Objectif

Utiliser différents algorithmes ou règles métier, conceptuellement identiques, suivant le contexte

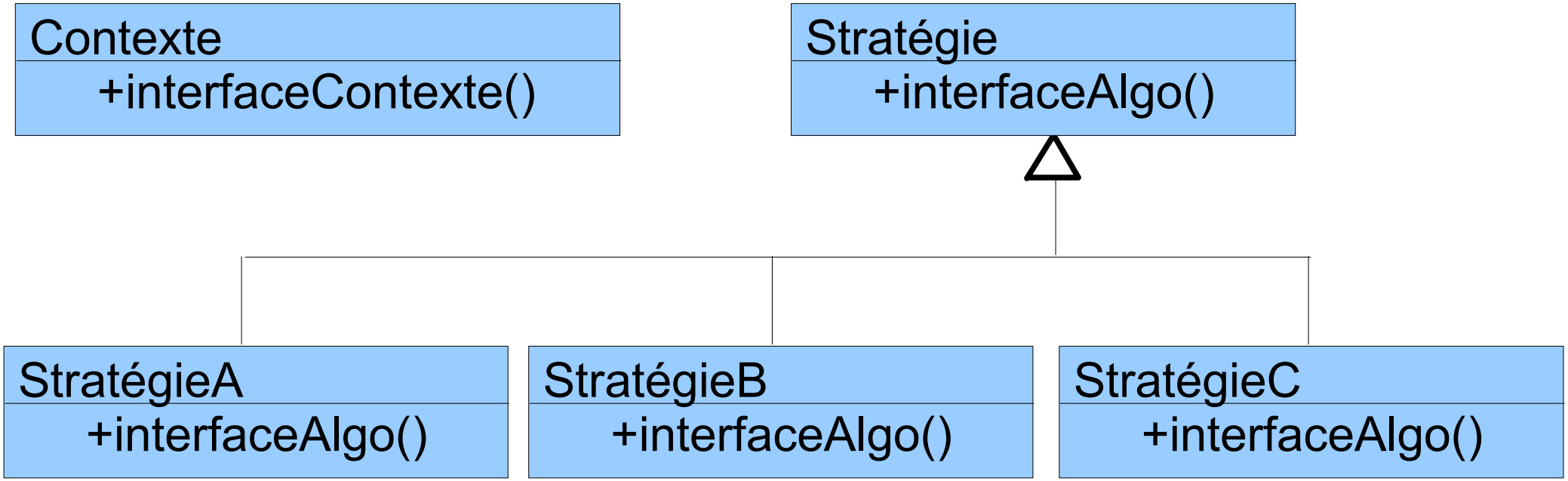
Problème

Sélection d'un algorithme dépend du client ou des données

Solution

Séparer la sélection de l'algorithme de son implantation

Pattern Stratégie (2)



Pattern Patron de méthode (1)

Objectif

- Définir une structure d'algorithme
- Déléguer la définition de certaines étapes aux sous-classes

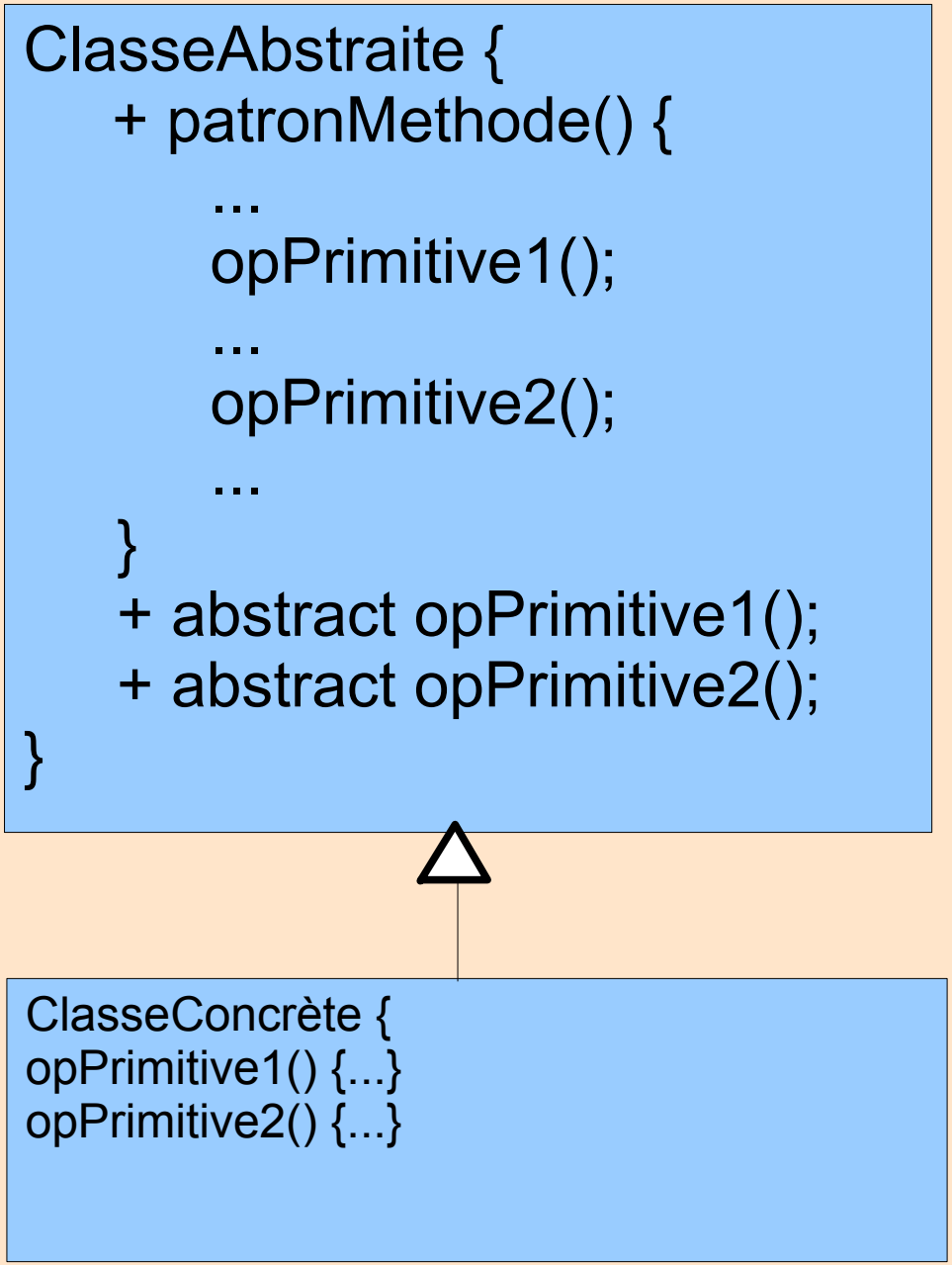
Problème

- Suivre une procédure cohérente à ce niveau de détail donné
- Implantation différée à un niveau plus détaillé

Solution

- Permettre la définition de sous-étapes variant tout en concervant un processus de base cohérent

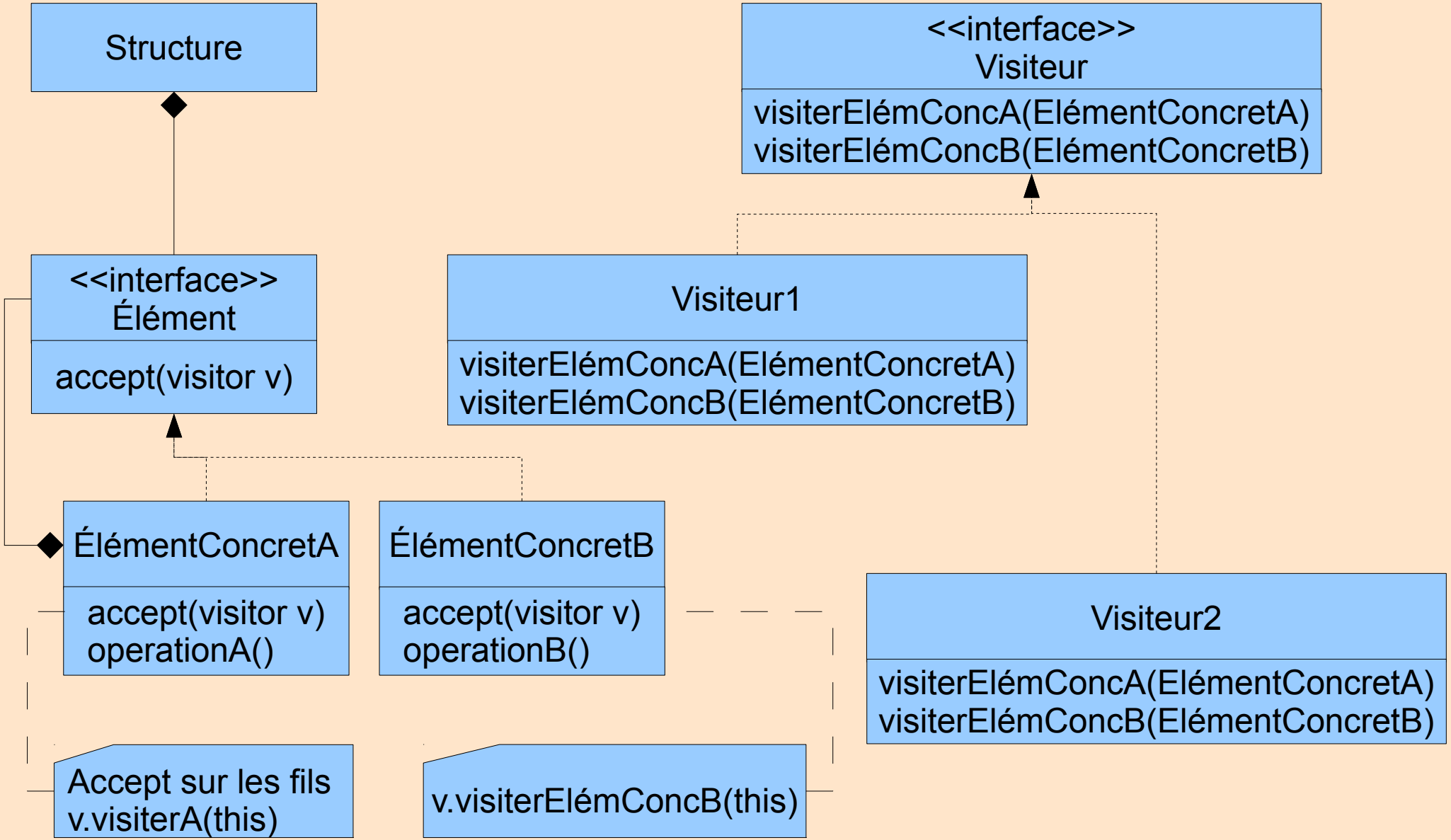
Pattern Patron de méthode (2)



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



Visiteur – exercice du TP

- On dispose d’une structure d’arbre binaire de recherche. On souhaiterait pouvoir effectuer des recherches et des ajouts dans cette structure sans que ces comportements n’affectent les classes représentant la structure