Les designs patterns (Patrons de conception)

3ème Année SI - GL

2020-2021

Dr. KOUAH SOFIA

Introduction aux designs patterns

Challenge de la programmation orientée objets

minimiser les interdépendances et maximiser la cohésion

Les objets (et les modules) sont des briques élémentaires permettant de réaliser ces architectures puisque :

- Ils aident à maximiser la cohésion grâce à l'encapsulation de leur état.
- Les classes abstraites (et l'héritage) permettent de minimiser les interdépendances en contrôlant le niveau de détails des services visibles par les clients d'un objet.

Le sujet d'aujourd'hui :

Comment agencer ces briques ?

- Deux difficultés

> La conception de composants réutilisables est une activité délicate.
>
> La programmation objet a des problèmes intrinsèques.

Introduction aux designs patterns

- En conception, il n'est pas rare de faire face à un problème qui a déjà été rencontré et résolu par d'autres personnes.
- Réutiliser les solutions trouvées par ces autres personnes permet de gagner non seulement du temps mais aussi de la qualité, pour peu que ces solutions aient été largement diffusées et corrigées d'éventuelles erreurs.
- Pour rendre cette idée concrète, **E. Gamma** a défini le concept de ***patron de conception***.
- **Un patron de conception** est une solution éprouvée qui permet de résoudre un problème de conception très bien identifié. Soulignons qu'un patron de conception est un **couple <problème/solution>.**
- **La solution définie par un patron** de conception n'est intéressante que si nous faisons face au même problème que celui traité par le patron. Il ne faut en aucun cas vouloir appliquer les solutions définies par les patrons de conception si nous n'en rencontrons pas les problèmes.

Référence Design Pattern

Gamma, Helm, Johnson, Vlissides
Design Patterns : Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995.

auteurs AKA "*Gang of Four*" (GoF)

Notion de design pattern

Un Design Pattern est une solution à un problème récurrent dans la conception d'applications orientées objet. Un patron de conception décrit alors la solution éprouvée pour résoudre ce problème d'architecture de logiciel. Comme problème récurrent on trouve par exemple la conception d'une application où il sera facile d'ajouter des fonctionnalités à une classe sans la modifier. A noter qu'en se plaçant au niveau de la conception les Design Patterns sont indépendants des langages de programmation utilisés.

Définition: Design pattern

Technique abstraite de résolution d'un problème récurrent

Description d'un design pattern

La description d'un patron de conception suit un formalisme fixe :

- **Nom**
- **Problème** : description du problème à résoudre
- **Solution** : les éléments de la solution, avec leurs relations. La solution est appelée patron de conception.
- **Conséquences** : résultats issus de la solution.
- **Autres** : implémentation, usages connus, etc.

Intérêt des designs patterns

Patron de conception = réutilisation

- Il est plus facile de réutiliser une solution de conception plutôt que du code
- Solution pour
  - décrire les bonnes pratiques de conception
  - Transmettre les connaissances acquises par l'expérience

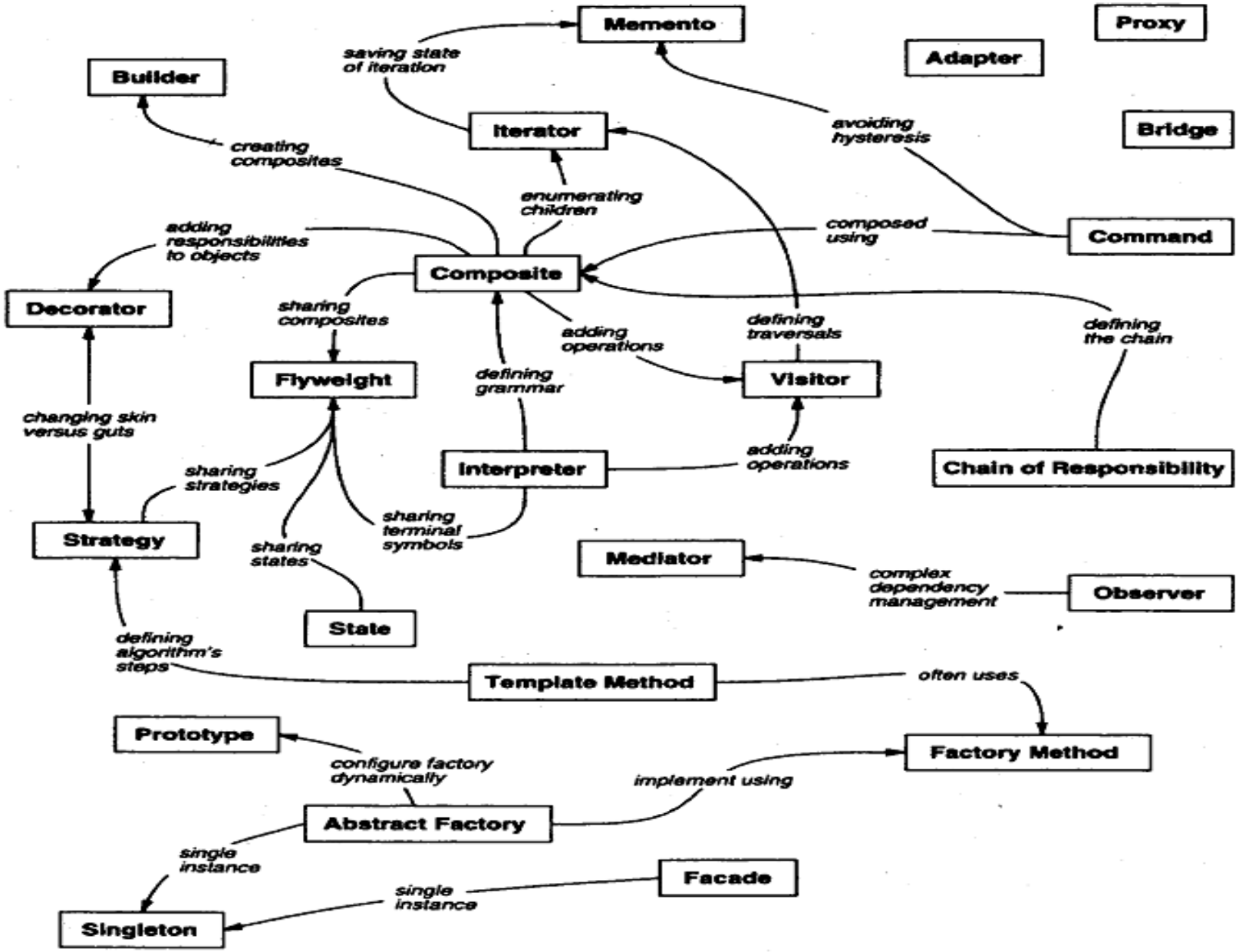
- Donc, ne pas réinventer la roue.
- Facilite la communication entre les développeurs.

Catégories des designs patterns

GoF ont explicité trois grandes classes de patrons dans leur livre, chacune spécialisée dans :

création	d'objets	(*creational patterns*)
structure	des relations entre objets	(*structural patterns*)
comportement	des objets	(*behavioral patterns*)

Catégories des designs patterns



Portée des designs patterns

	Créateur	Structurel	Comportement
Classe	Factory	Adaptor	Interpreter Template
Objet	Abstract Factory Builder Prototype Singleton	Adaptor Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Patrons de création

Les 5 patterns de création

Création

- Factory, AbstractFactory, Builder, Prototype, Singleton
-

Abstraction du processus de création.

Encapsulation de la logique de création.

On ne sait pas à l'avance ce qui sera créée ou comment cela sera créé.

Patrons structurel

Les 7 patterns de structure

- Structure

  Adaptator, Bridge, Composite, Decorator, Interface, Flyweight, Proxy

d

- Comment sont assemblés les objets.
- Découpler l'interface de l'implémentation.

Patrons comportementaux

Les 11 patterns comportementaux

Comportement

- Interpretor, Template method, Chain of responsability, Command, Iterator, Mediator, Memento, Observator, State, Strategy, Visitor

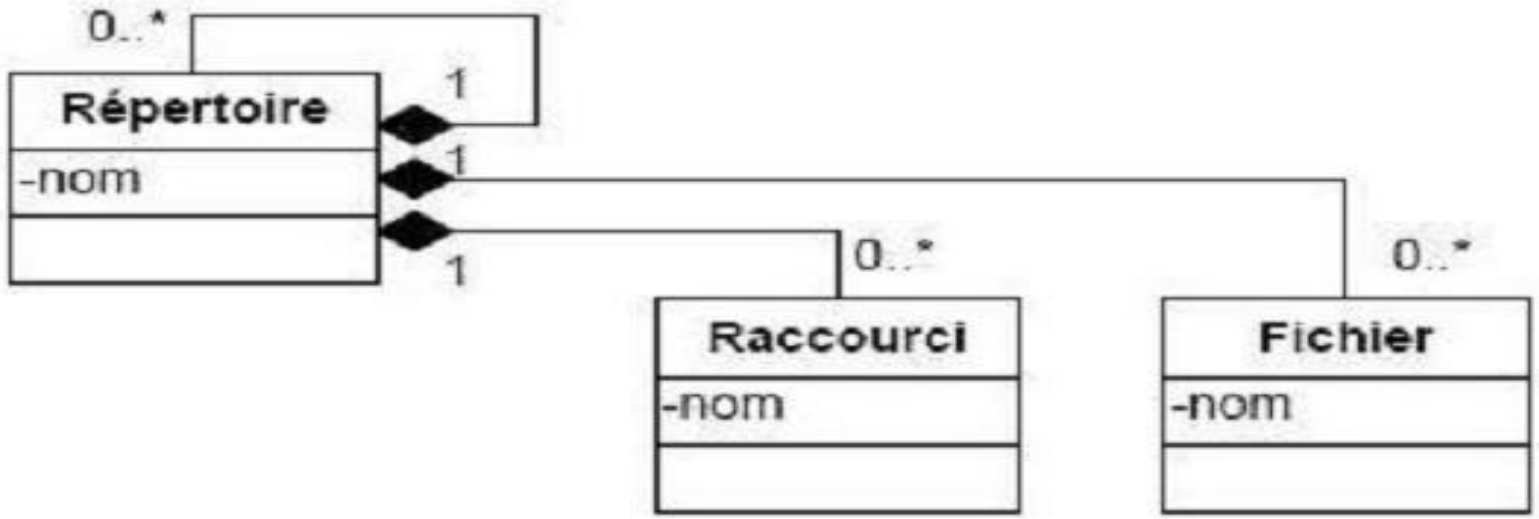
Mode de communication entre les objets

Exemple motivant les designs pattern

- **Exemple** : [UML 2 par la pratique, P. Rocques]
- **Énoncé** : proposer une solution élégante qui permette de modéliser le système de gestion de fichiers suivant :
  1. les fichiers, les raccourcis et les répertoires sont contenus dans des répertoires et possèdent un nom
  2. un raccourci peut concerner un fichier ou un répertoire
  3. au sein d'un répertoire donné, un nom ne peut identifier qu'un seul élément (fichier, sous-répertoire ou raccourci)

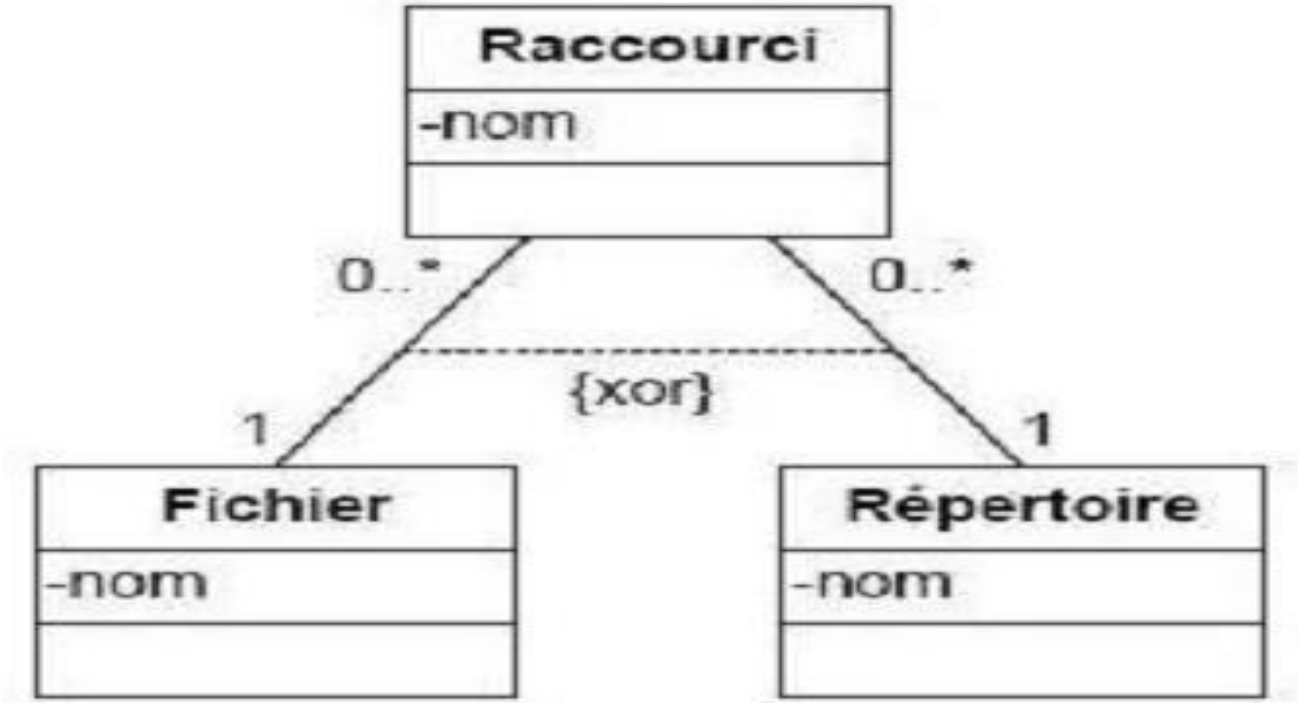
Exemple motivant les designs pattern

- **1ère phrase**
  - 3 concepts = 3 classes
  - contenance modélisée par une composition
  - multiplicité côté contenant est égale à 1
  - destruction répertoire entraîne destruction de tout ce qu'il contient



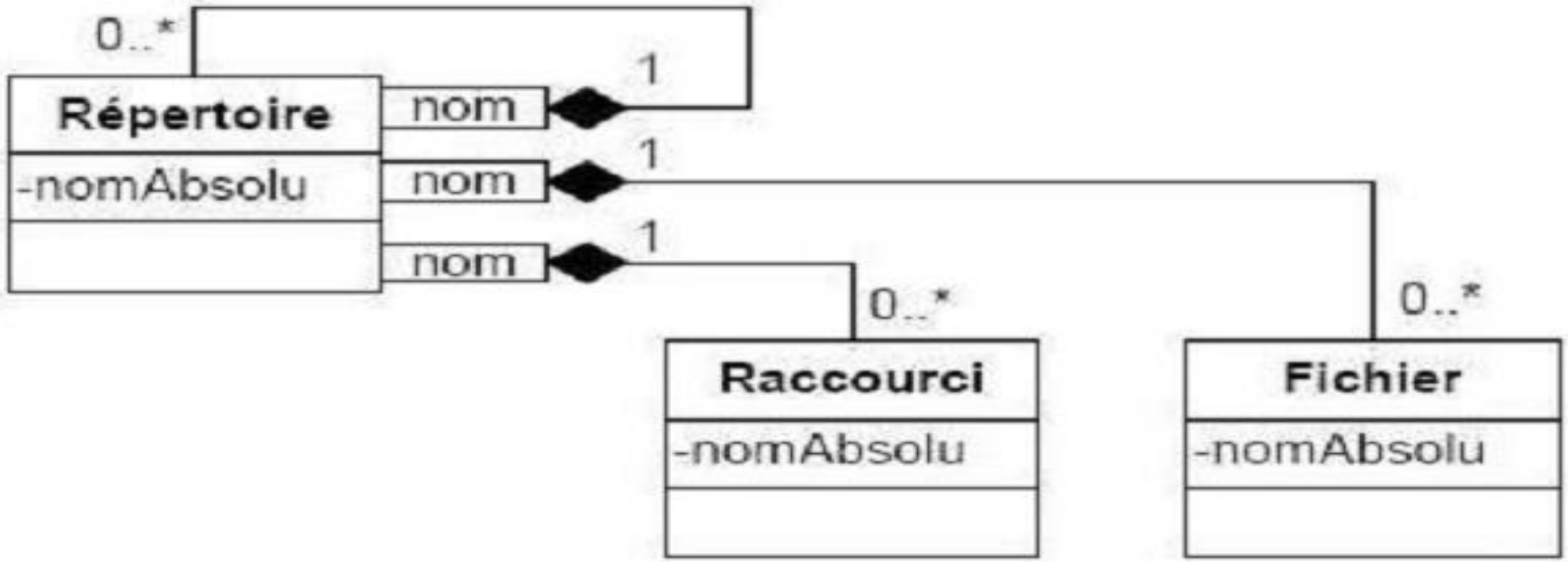
Exemple motivant les designs pattern

- **2ère phrase** : un raccourci peut concerner un fichier ou un répertoire
  - 2 associations en exclusion mutuelle



Exemple motivant les designs pattern

- **3ère phrase** : au sein d'un répertoire donné, un nom ne peut identifier qu'un seul élément (fichier, sous-répertoire ou raccourci)
  - **idée** : qualifier chacune des trois compositions avec un attribut nom



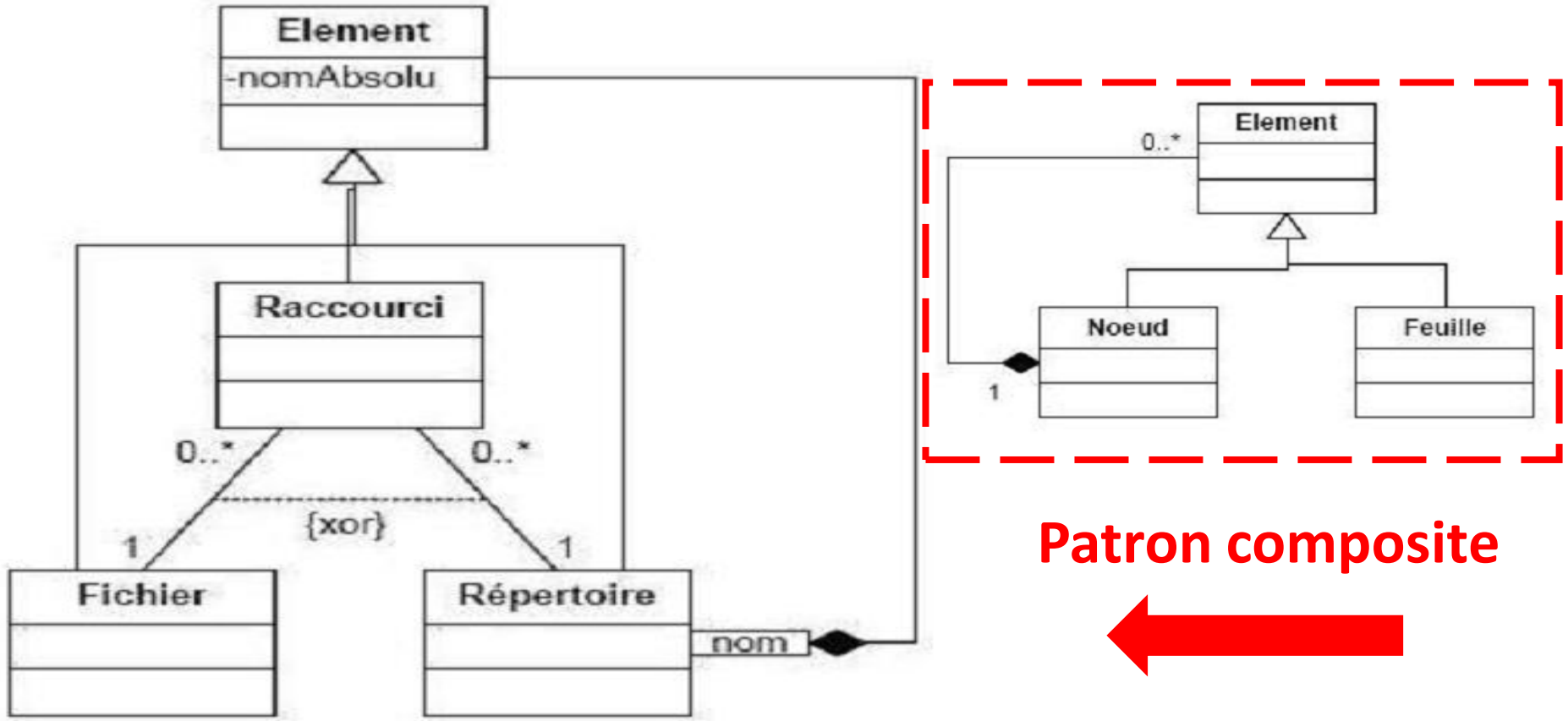
Exemple motivant les designs pattern

- **Problème** : rien n'empêche qu'un fichier et qu'un raccourci aient le même nom
  - 3 compositions et 3 noms ne sont pas une si bonne idée que cela
  - il faut un qualificatif unique pour chaque type d'élément contenu dans un répertoire
- **Solution** : importance du terme élément
  - modifier le modèle afin de n'avoir plus qu'une composition à qualifier

Exemple motivant les designs pattern

Solution

- double relation asymétrique entre Repertoire et Element
  - Répertoire contient des Element
  - Répertoire est un Element

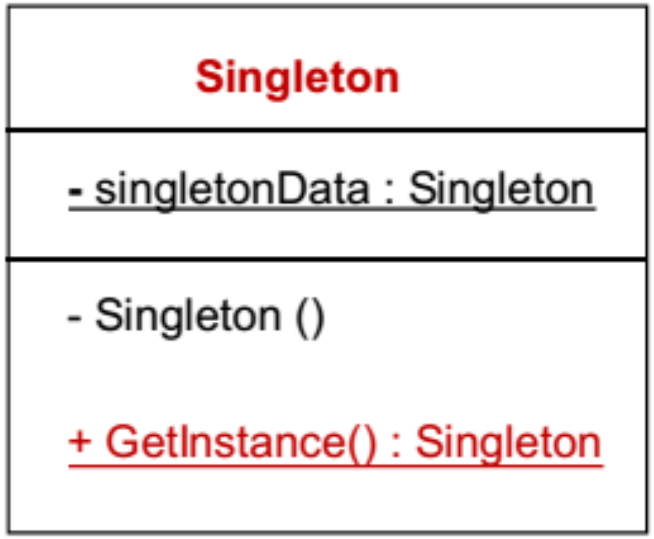


Patron composite

Singleton

- Le problème
  - Assurer qu'une classe n'a qu'une seule instance

- La solution
  - Le constructeur est privé
  - Une méthode ne crée une instance que si il n'en existe pas encore



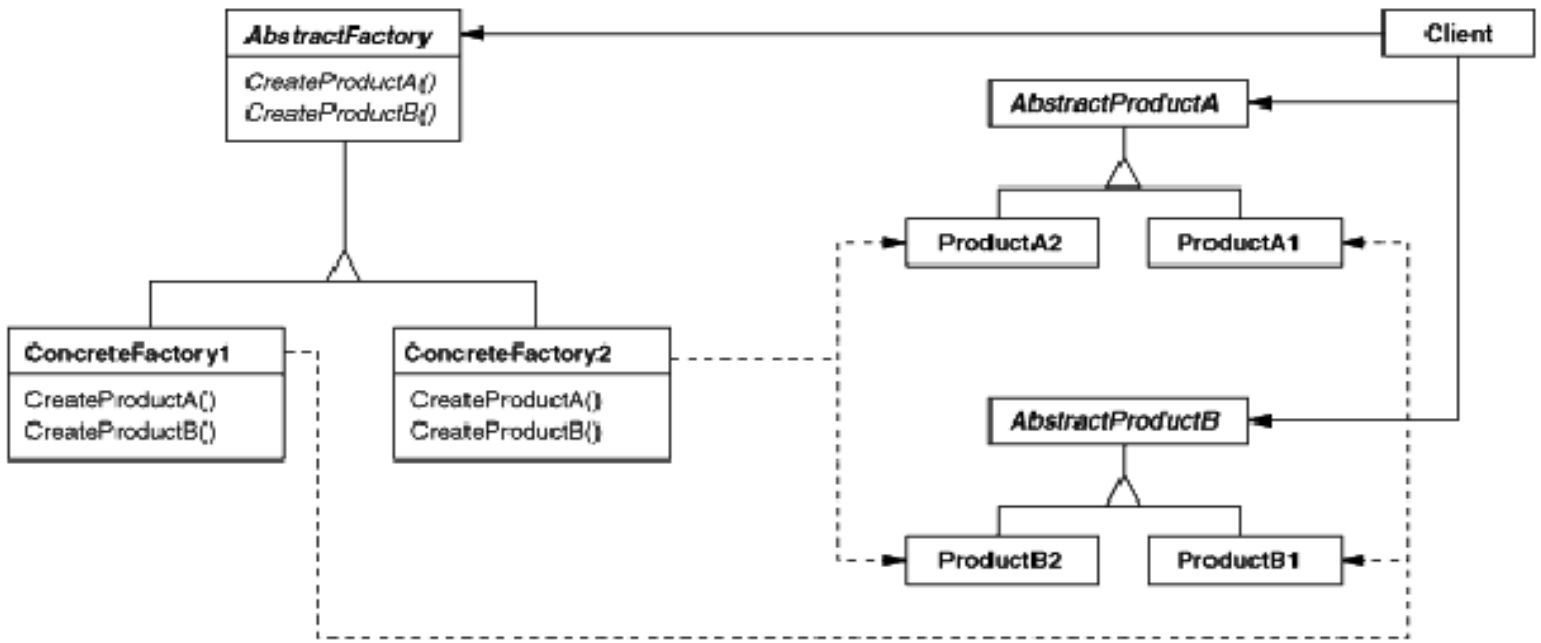
L'objectif du pattern `SINGLETON` est de garantir qu'une classe ne possède qu'une seule instance et de fournir un point d'accès global à celle-ci.

Singleton

```
// Only one object of this class can be created
class Singleton {
    private static Singleton instance = null;

    private Singleton() {
        ...
    }

    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
    ...
}

class Program {
    public void aMethod() {
        Singleton X = Singleton.getInstance();
    }
}
```

Abstract Factory (1)

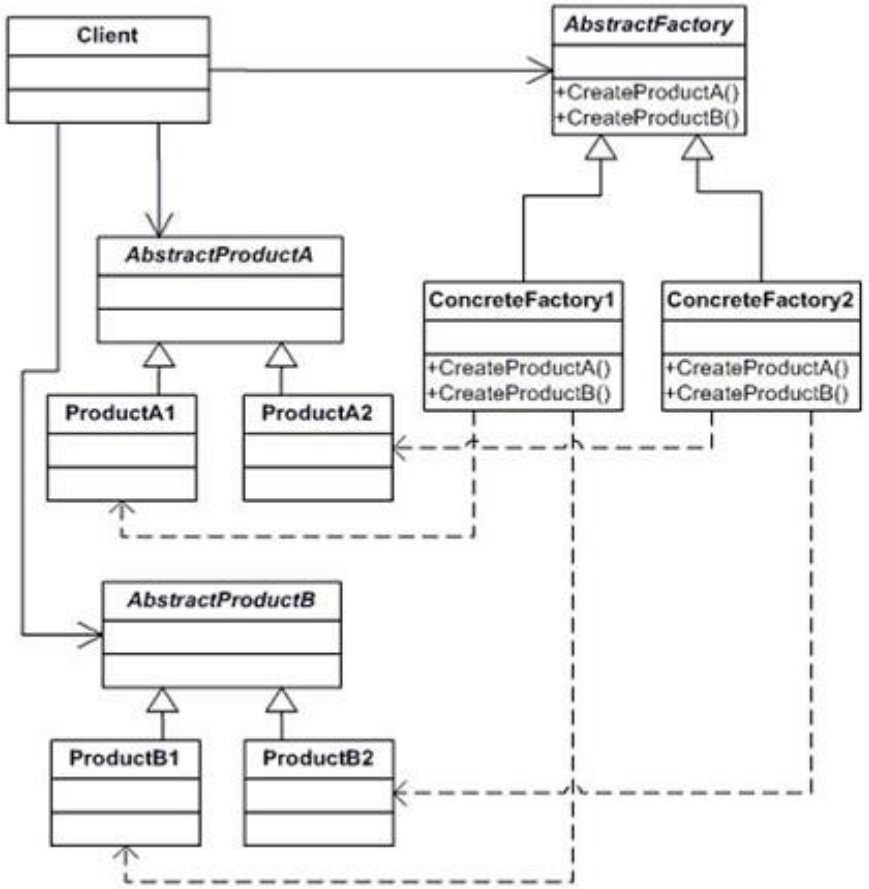
- Problème
  - On souhaite créer une instance d'une interface
  - mais quand on écrit le code
    $\Rightarrow$ on ne sait pas de quel type concret l'instance va être
  - Ex: documents XML, bordures de fenêtre...
- Solution
  - Fournit une interface pour la création d'objets d'une même famille en ignorant la classe concrète d'implémentation

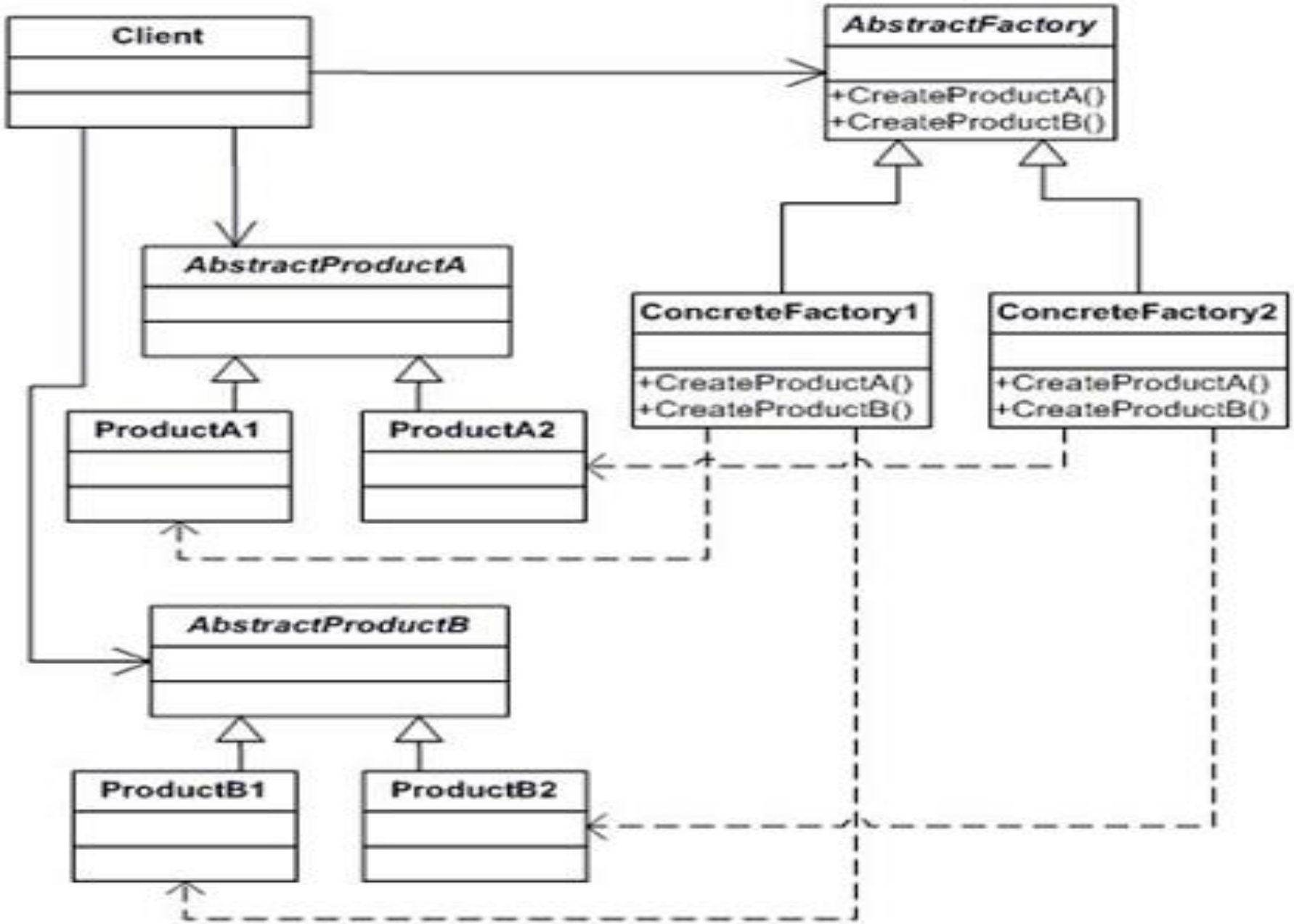


Abstract Factory (2)

Créer une famille d'objets sans spécifier leurs classes concrêtes

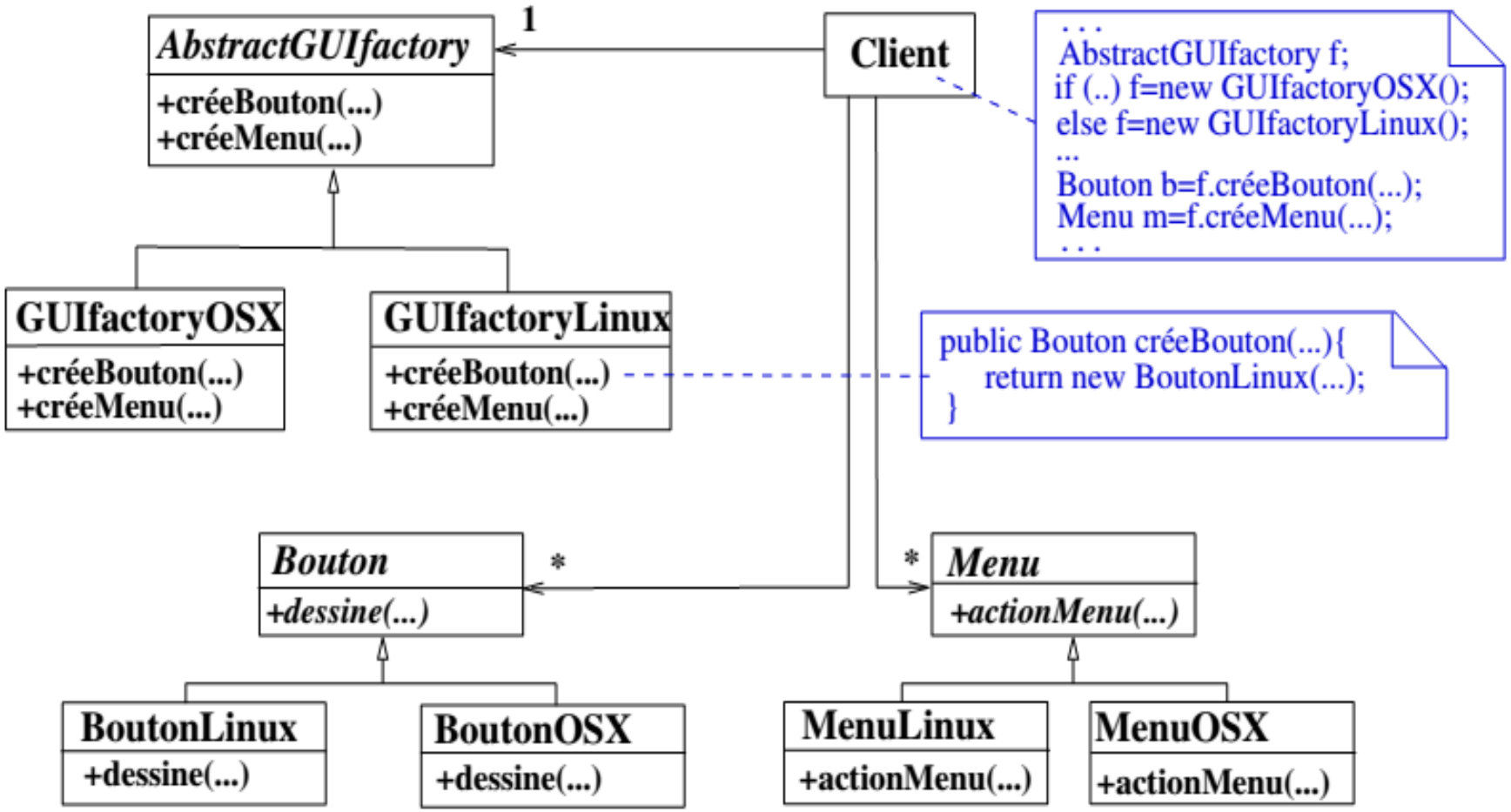
- Le problème
  - Créer un objet sans se préoccuper de son implémentation
  - Il existe plusieurs façons de construire un objet
- La solution
  - Séparation fabrique / élément
  - Séparation abstrait / concret
- Utilisation
  - Plateformes multiples

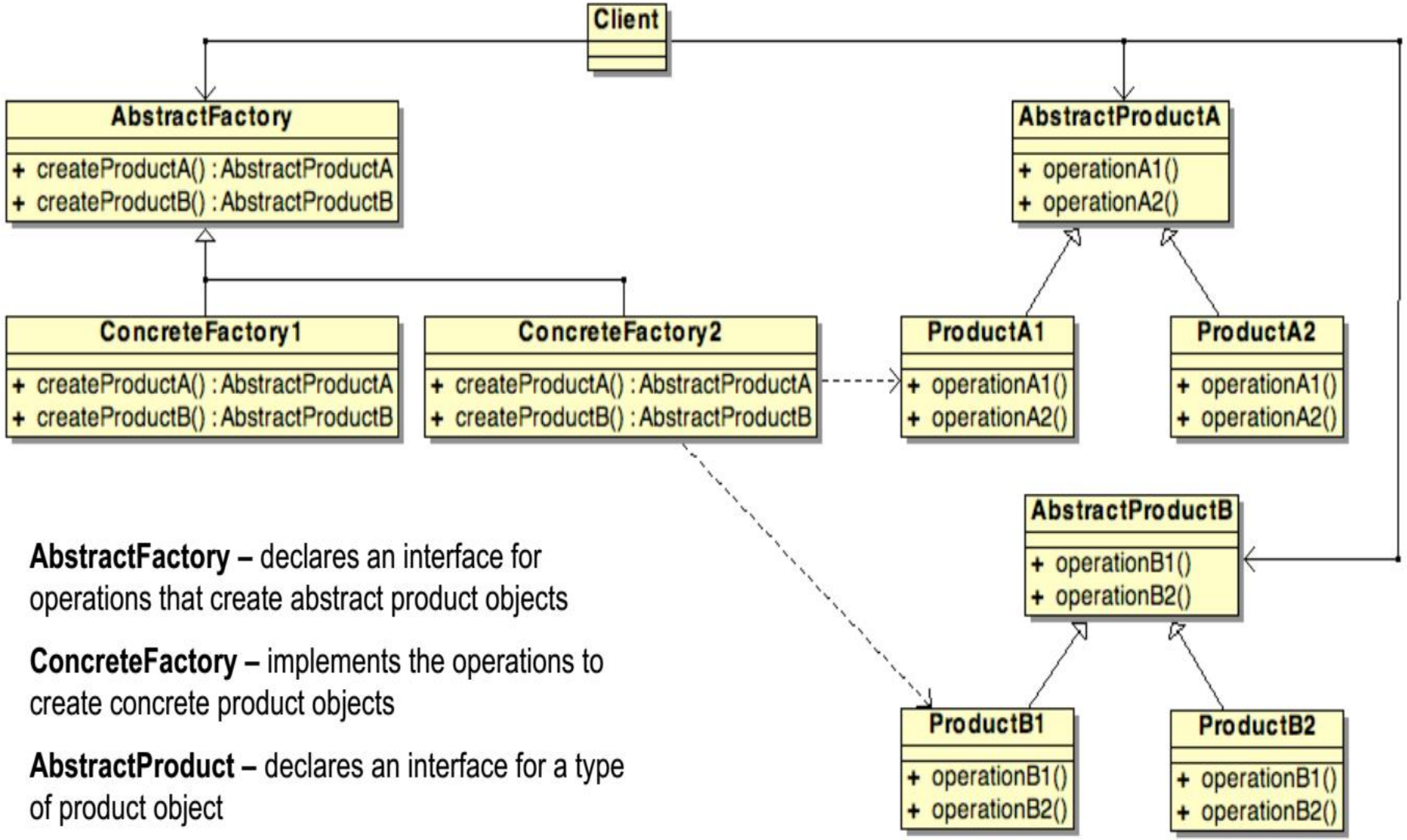



Client
AbstractFactory
+CreateProductA()
+CreateProductB()
AbstractProductA
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2

Exemple illustratif

- Créer une interface graphique avec widgets (boutons, menus, ...)
- Point de variation : OS (Linux, OSX, Windows)





AbstractFactory – declares an interface for operations that create abstract product objects

ConcreteFactory – implements the operations to create concrete product objects

AbstractProduct – declares an interface for a type of product object

ConcreteProduct

– defines a product object to be created by the corresponding concrete factory

– implements the AbstractProduct interface

Client – uses interfaces declared by AbstractFactory and AbstractProduct classes

Factory Method (1)

- Problème
  - Le même que pour le pattern précédent
- Solution
  - Mise en commun du code redondant, implémentation des différents constructeurs des classes concrètes

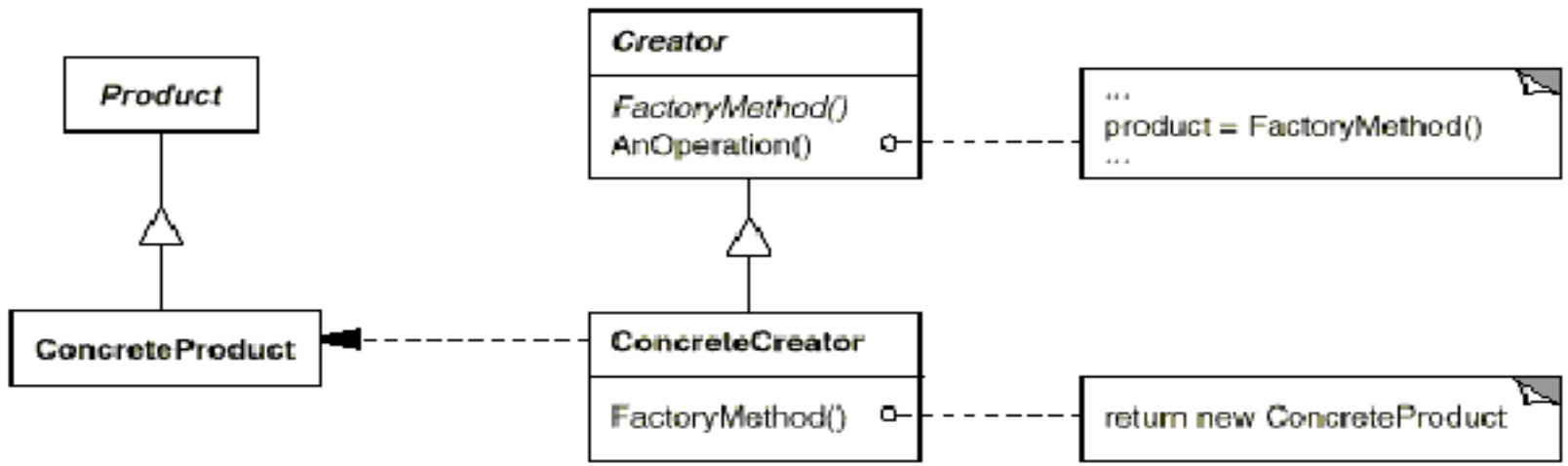
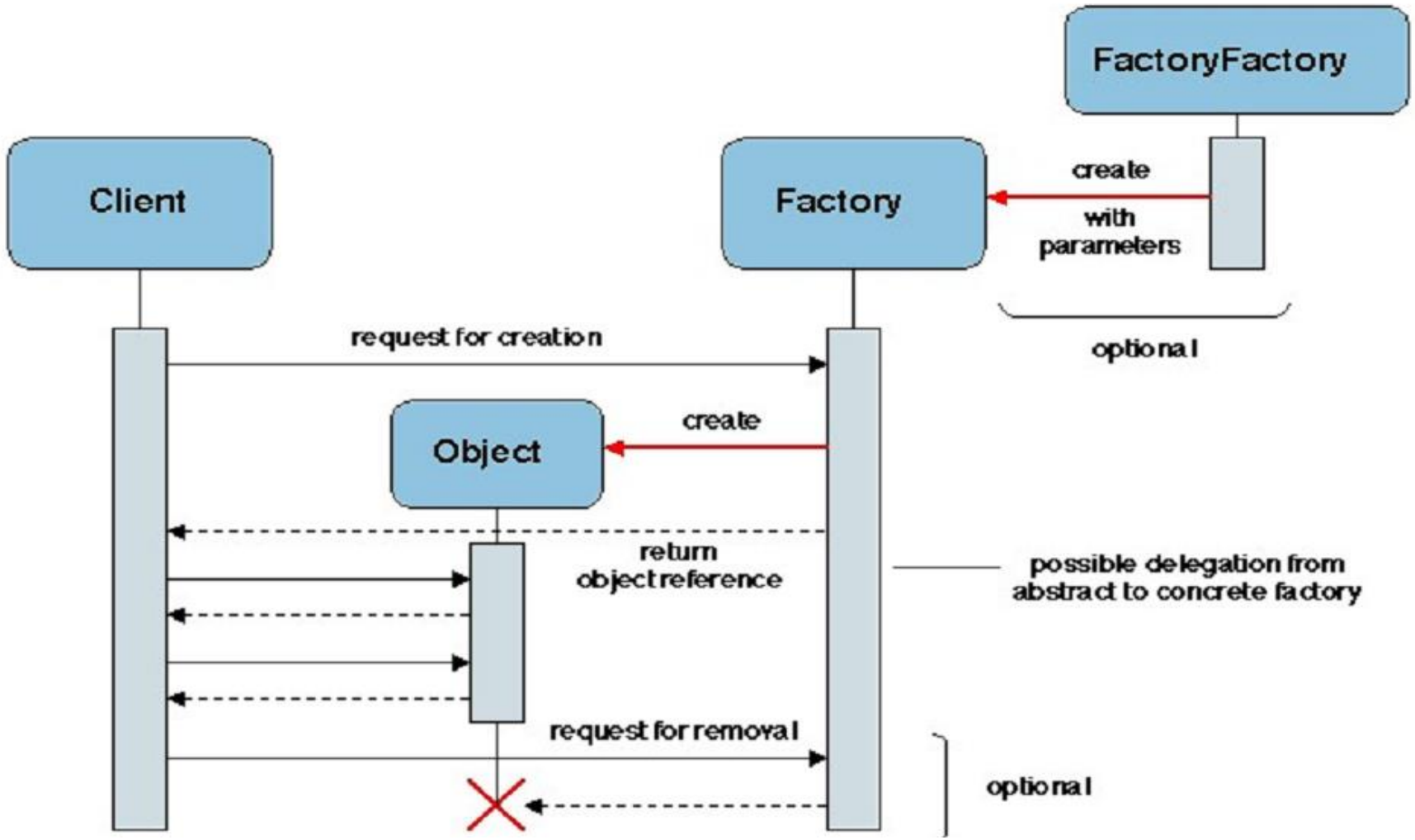


Figure: Factory Method

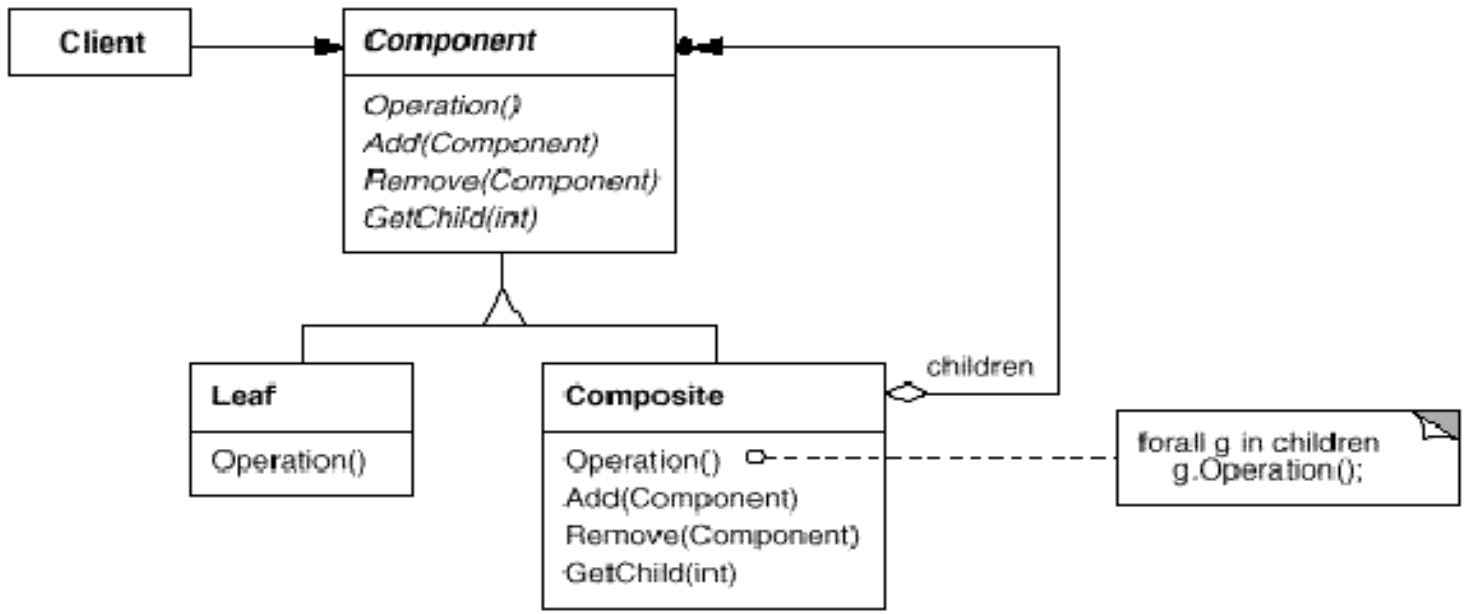
On peut donc créer *Product* sans lier le code client à un choix spécifique de classe qui l'implémente.

Factory Method (2)



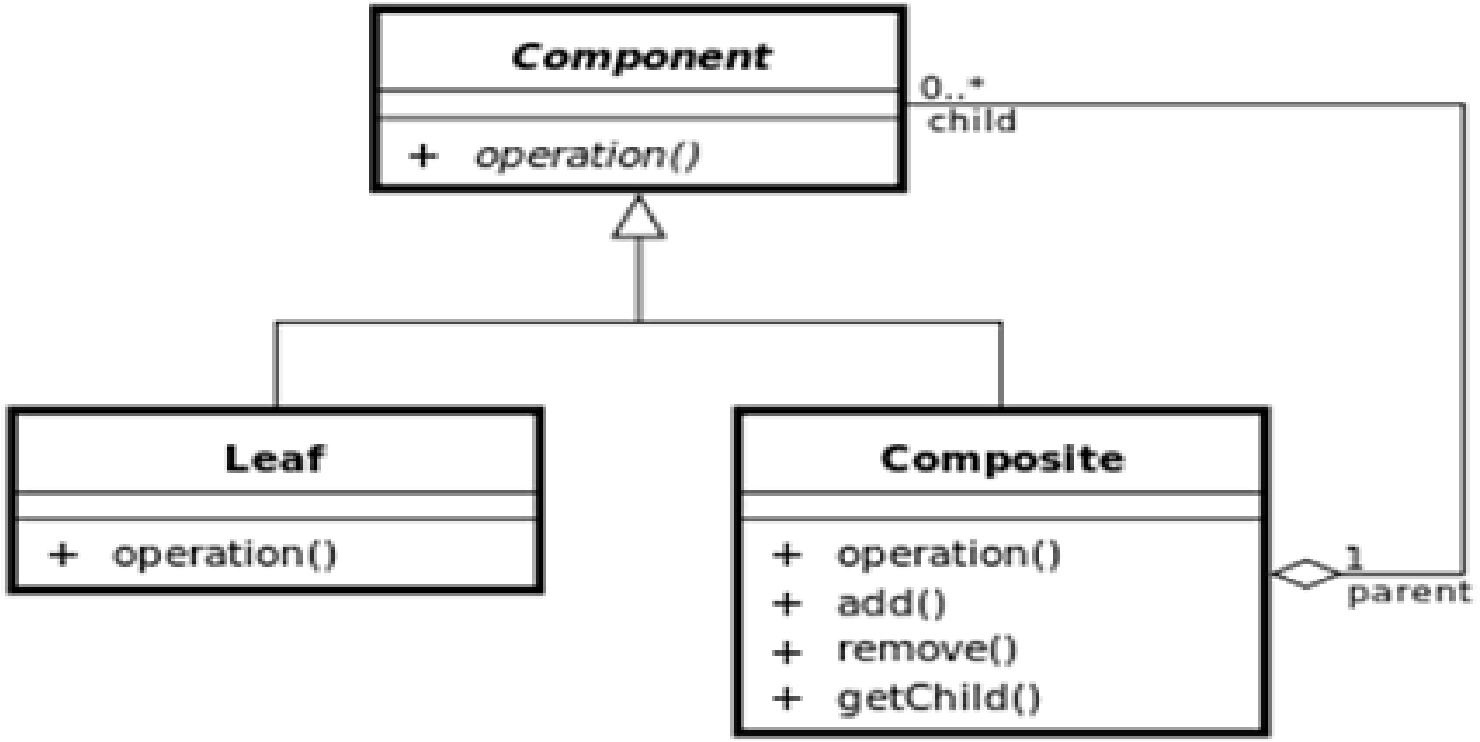
Composite (1)

- Problème
  - On souhaite créer des objets qui sont formés de plusieurs autres objets
  - Ex: logiciel de dessin, ou de cuisine
- Solution
  - On définit un objet composant qui est une interface pour tous les objets formant la composition
  - On déclare une classe permettant de gérer les enfants (add, remove, etc.)



Composite (2)

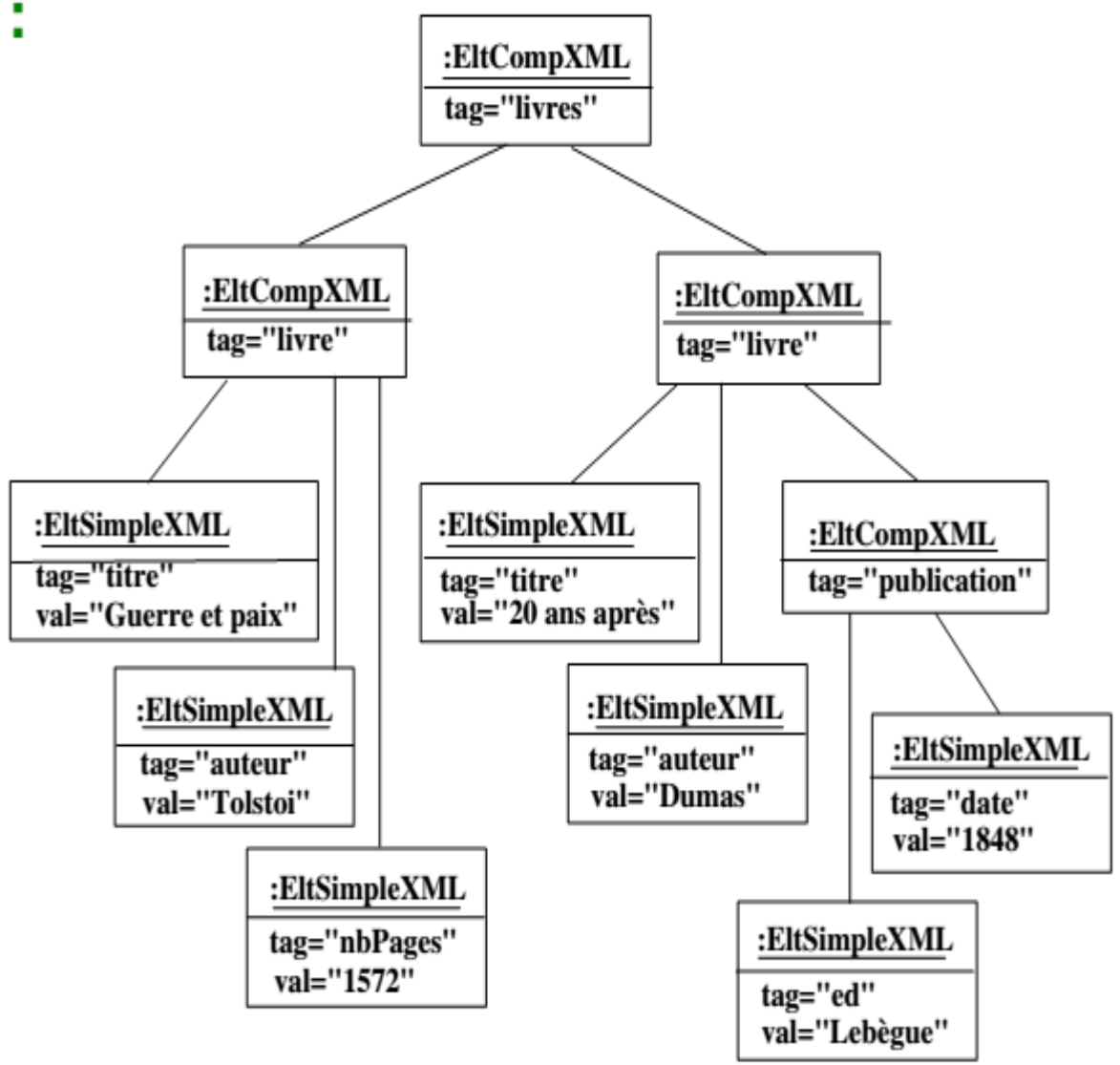
- Utilisations
  - Interface graphique
  - Bureautique
  - Souvent des versions dégradées



Composite (3)

Illustration sur un exemple :

```
< ?xml version="1.0" ?>
<livres>
    <livre>
        <titre>Guerre et paix</titre>
        <auteur>Tolstoï</auteur>
        <nbPages>1572</nbPages>
    </livre>
    <livre>
        <titre>20 ans après</titre>
        <auteur>Dumas</auteur>
        <publication>
            <ed>Lebègue</ed>
            <date>1848</date>
        </publication>
    </livre>
</livres>
```

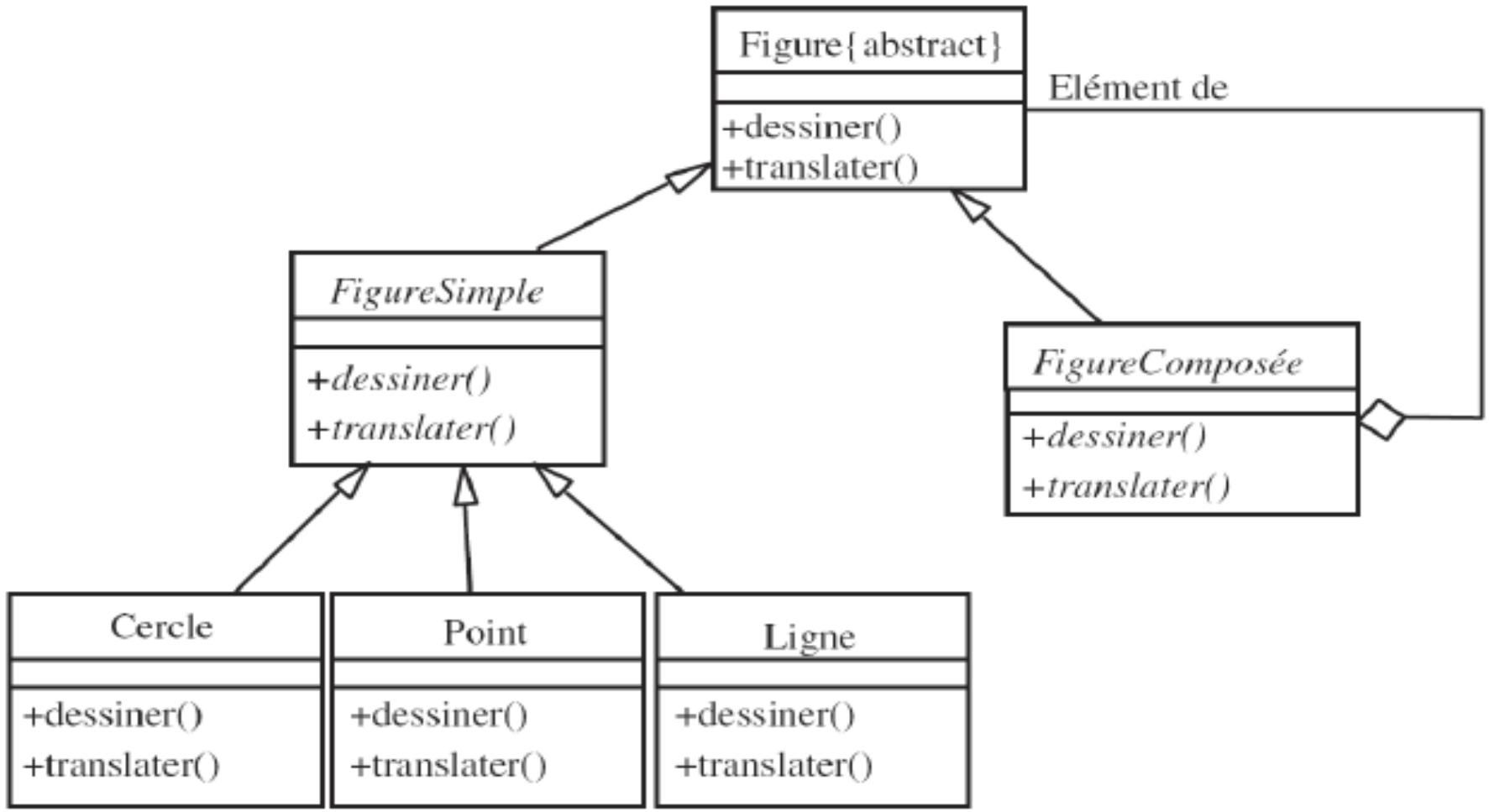


Application de composite

Une figure simple peut être un point, une ligne ou un cercle. Une figure peut être composée d'autres figures, simples ou elles-mêmes composées d'autres figures. Toutes les figures peuvent être dessinées ou translatées.

Question : Utilisez le pattern composite pour construire un diagramme de classe rendant compte de cette hiérarchie d'objets.

Solution


Figure{abstract}
+dessiner()
+translater()
Elément de
FigureSimple
+dessiner()
+translater()
FigureComposée
+dessiner()
+translater()
Cercle
+dessiner()
+translater()
Point
+dessiner()
+translater()
Ligne
+dessiner()
+translater()

Avantages & Inconvénients

- Un niveau d 'abstraction élevé
  - qui permet d 'élaborer des constructions logicielles de meilleure qualité
- Réduction de la complexité par séparation des préoccupations
- Capitalisation de l'expérience en conception de logiciel
  - Catalogue de solutions
- Nombreuses mises en œuvres dans les langages de programmation

- Apprentissage
  - Effort de synthèse : reconnaître, abstraire et appliquer
- Expérience de mise en œuvre
  - Choisir les « bons » Design Patterns à appliquer
    - Mise en œuvre dégradées
  - Nombreuses solutions
    - Composition de patterns …
- Complexité et efficacité du code
  - Beaucoup de couches, beaucoup de classes
  - Fondre les patterns dans le code