Design Pattern Composite

«Pattern Composite» : le problème

✓Problème :

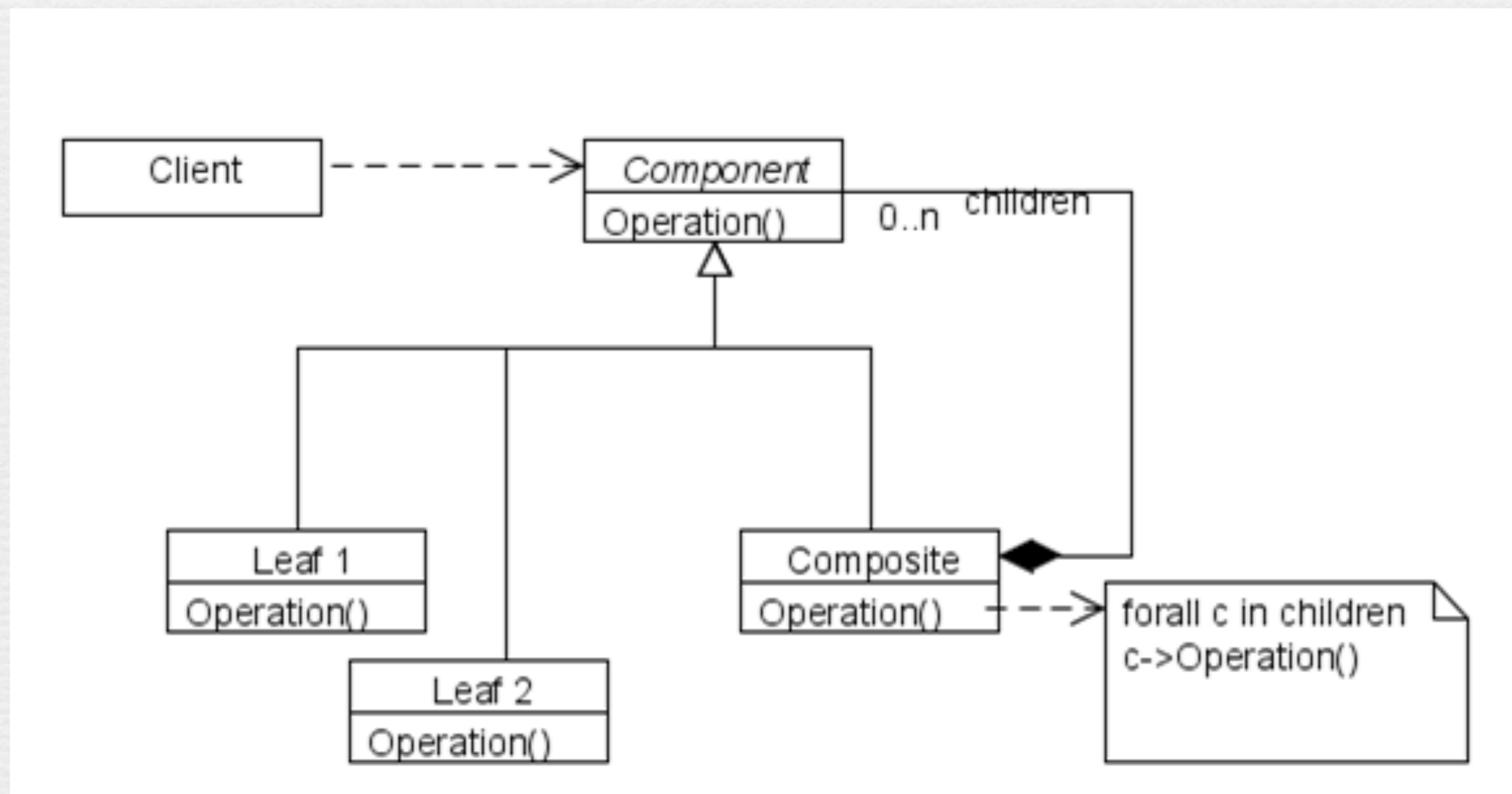
➡Une application doit manipuler des collections d'objets dont certains sont «primitifs» et d'autres «composites».

➡Ces objets doivent tous répondre à une méthode dont le comportement est différent selon que l'objet est composite ou non.

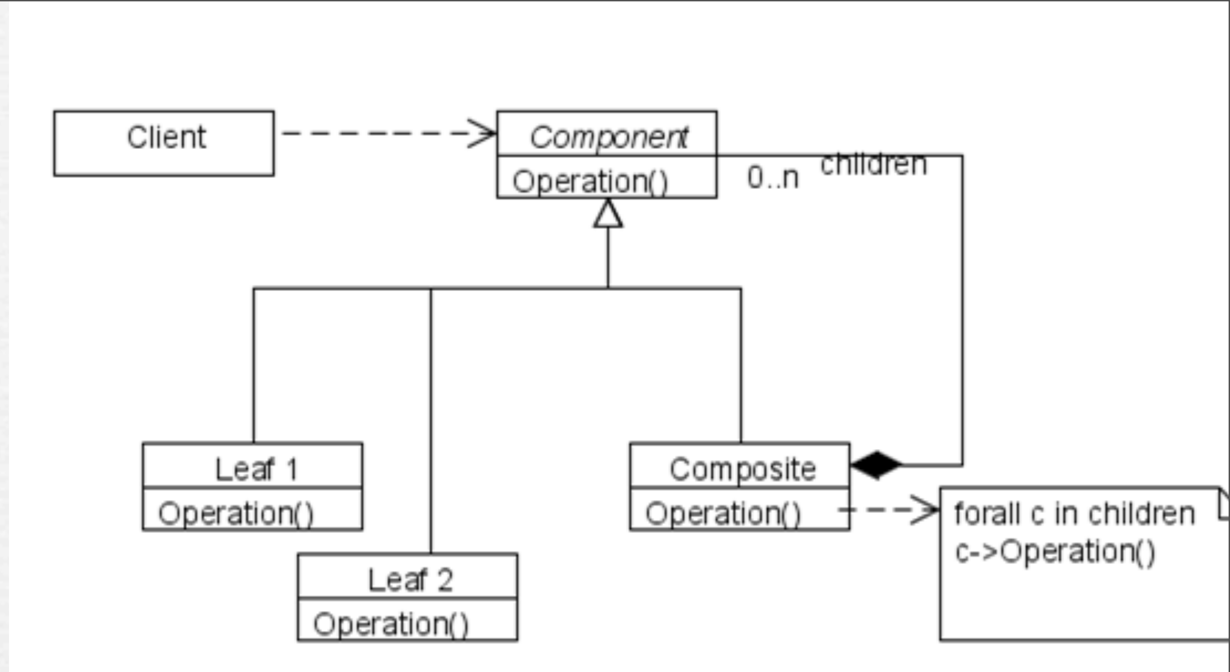
«Pattern Composite» : la solution

✓Organiser les objets dans une structure d'arbre qui capture la hiérarchie «partie-contenant».

✓Le client adresse alors tous les objets de manière uniforme. On parle de composition récursive.



«Pattern Composite» : les rôles



➡ **Component**

- declare l'interface des objets pris en compte dans la composition
- implémente le comportement par défaut des composants autant que possible

➡ **Composite**

- définit le comportement des composants ayant des «enfants» dans la hiérarchie de la composition
- référence les composants fils (ils peuvent eux-même être composites!)

➡ **Leaf**

- définit le comportement des objets primitifs dans la composition

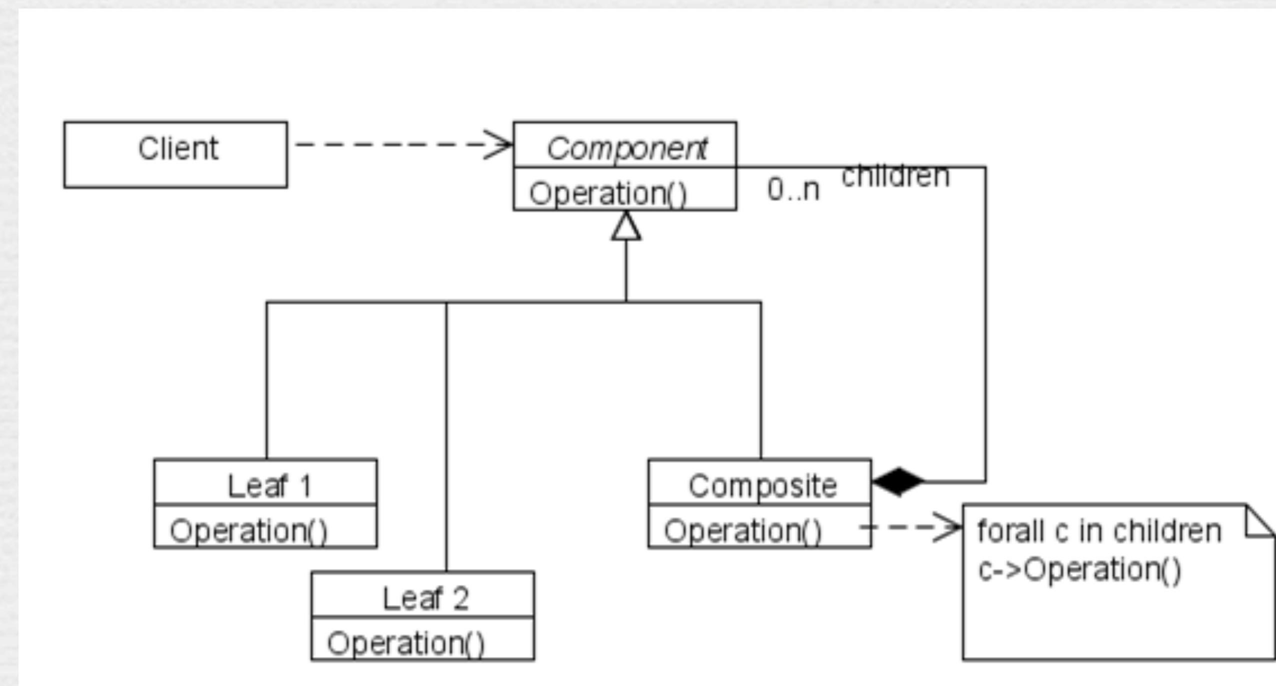
➡ Client

- manipule les objets de la composition au travers de l'interface Component

Design Pattern Composite en action

- ✓Notre entreprise vend des produits. Chaque produit a un prix, une référence et une description. Nous vendons des jeux videos. A certaines périodes de l'année nous vendons les jeux par lots. Le prix du lot est alors la somme des prix des jeux dans le lot réduite de 10%.
- ✓Nous construisons notre catalogue comme un ensemble de produits que l'on peut imprimer.
- ✓Tout produit créé a une référence qui est automatiquement déterminée à la création du produit.

A vous !



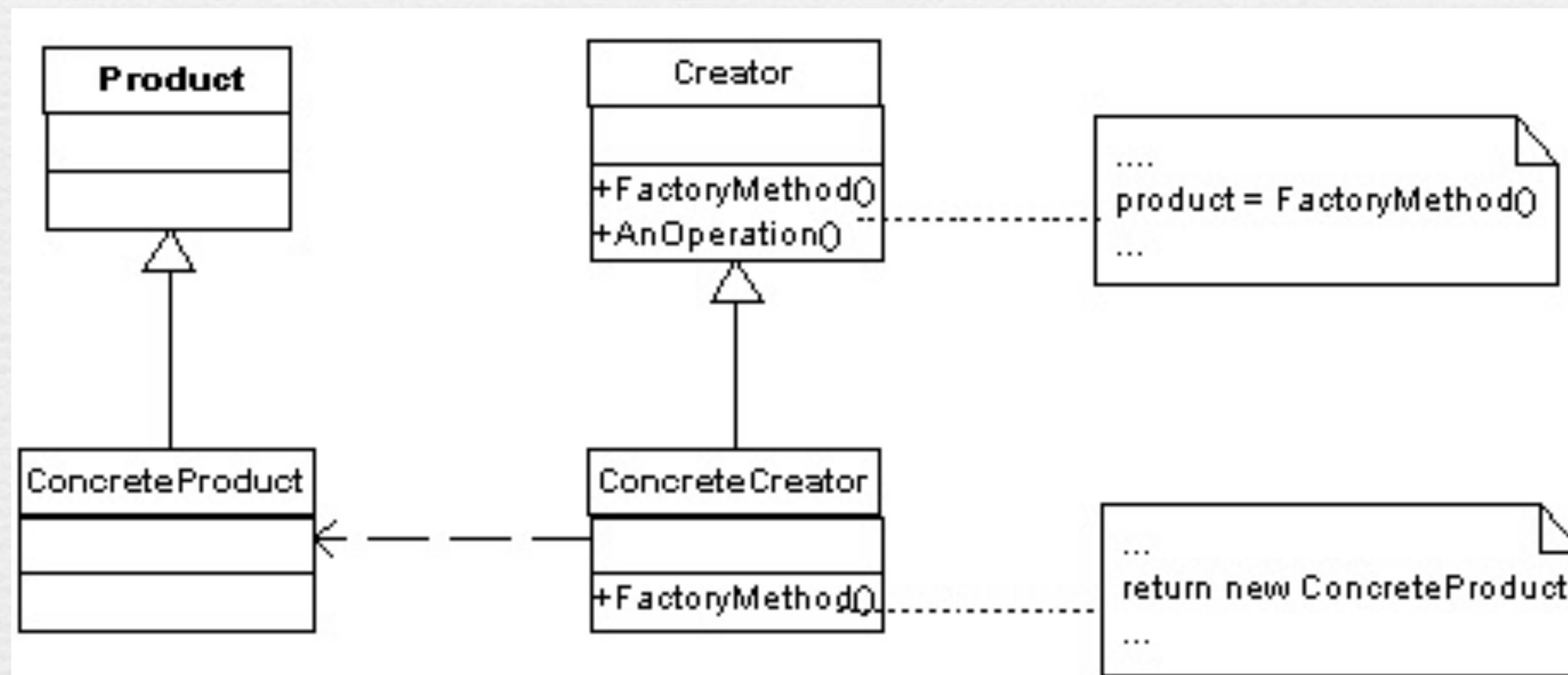
Design Pattern «Factory Method»

Factory Method Pattern : le problème

- ✓ Une application doit créer des objets sans connaître les classes de mises en oeuvre de ces objets.

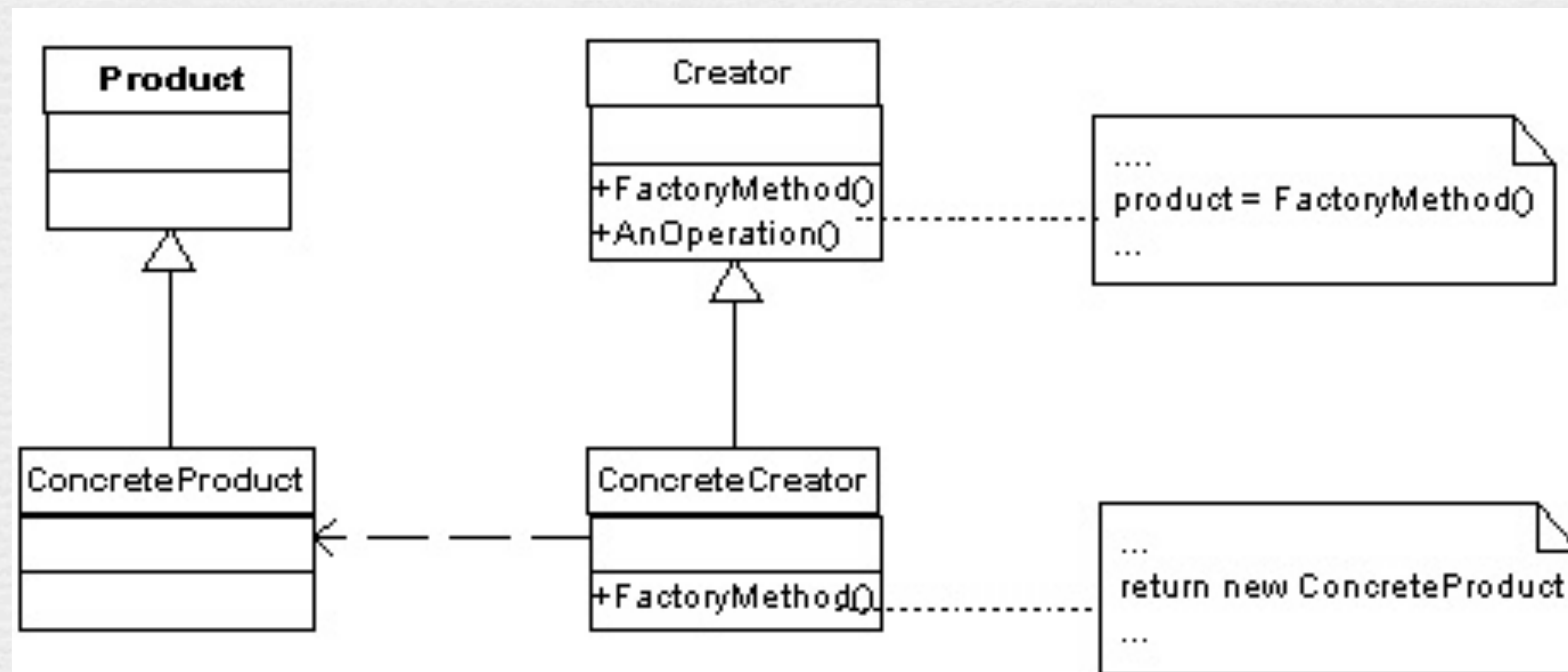
Factory Pattern : la solution

- ✓Définir une interface pour créer des objets, mais laisser les sous-classes décider quelle classe instancier.
- ✓Tous les objets créés se conforment à la même interface

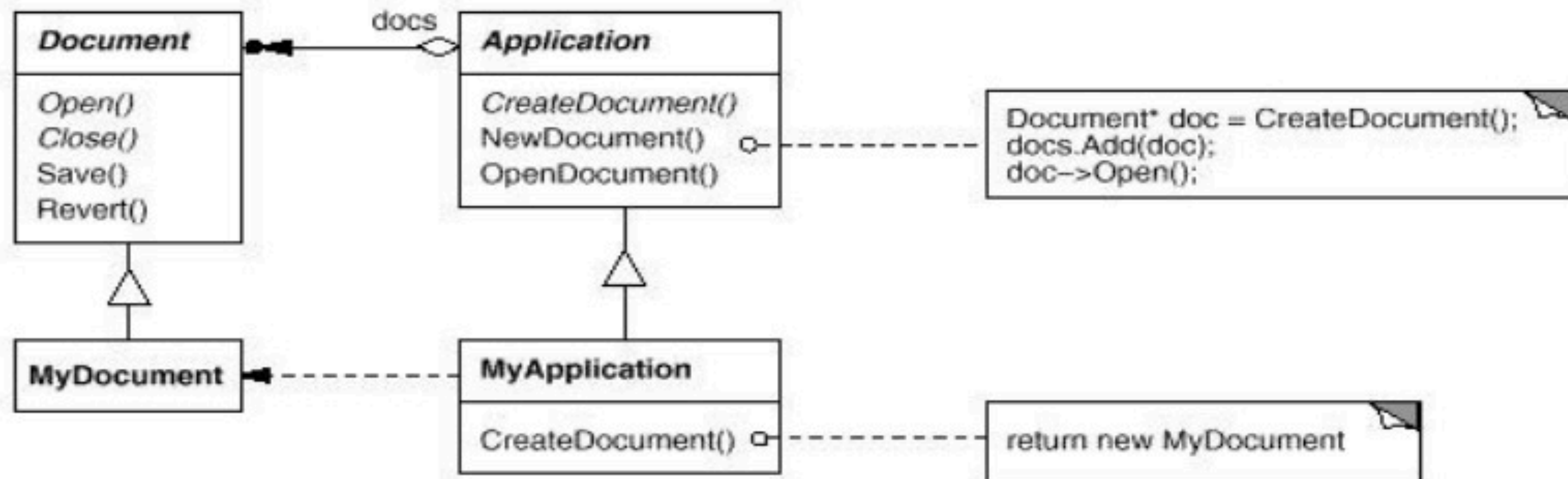


Factory Pattern : les rôles

- ✓ «Produit» définit l'interface des objets créés par la méthode de création.
- ✓ Les «Produits concrets» implémentent l'interface «Produit».
- ✓ La «fabrique (creator)» déclare la méthode de création qui retourne des objets «Produit».
- ✓ Les «fabriques concrètes» surchargent les méthodes pour créer des «Produits concrets».



Factory Method Pattern : un exemple

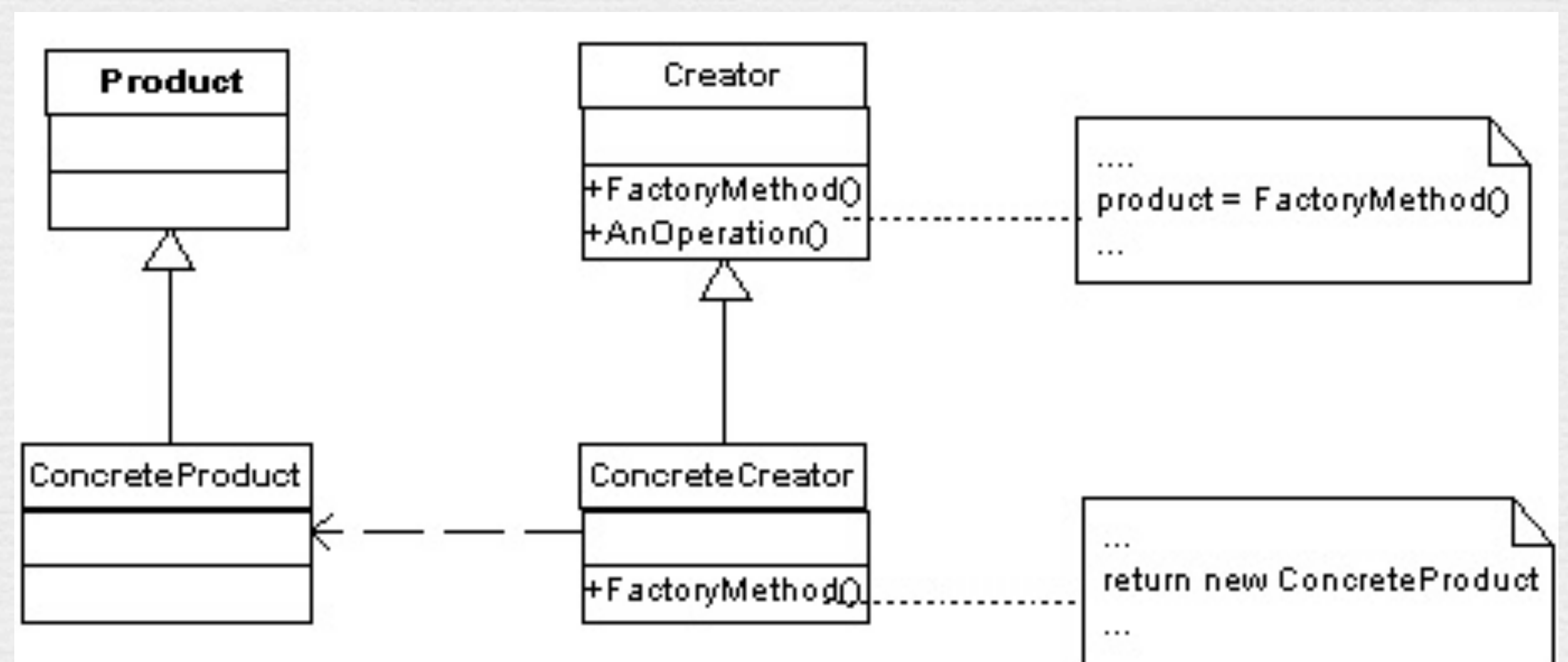


Quelle est la méthode «factory» ?

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

Method Factory Pattern : en action

- ✓ Un jeu est définie par des joueurs, des personnages.
- ✓ Il existe plusieurs variantes du jeu :
  - ➡Dans la version simple du jeu, les joueurs sont des humains et les personnages sont tirés au hasard.
  - ➡Dans la version «Disney», les personnages sont tirés au hasard dans un ensemble de Personnages de Disney
  - ➡Dans la version «apprentissage» les joueurs sont des «intelligences artificielles».
- ✓ Un joueur lance la partie en choisissant une version du jeu.

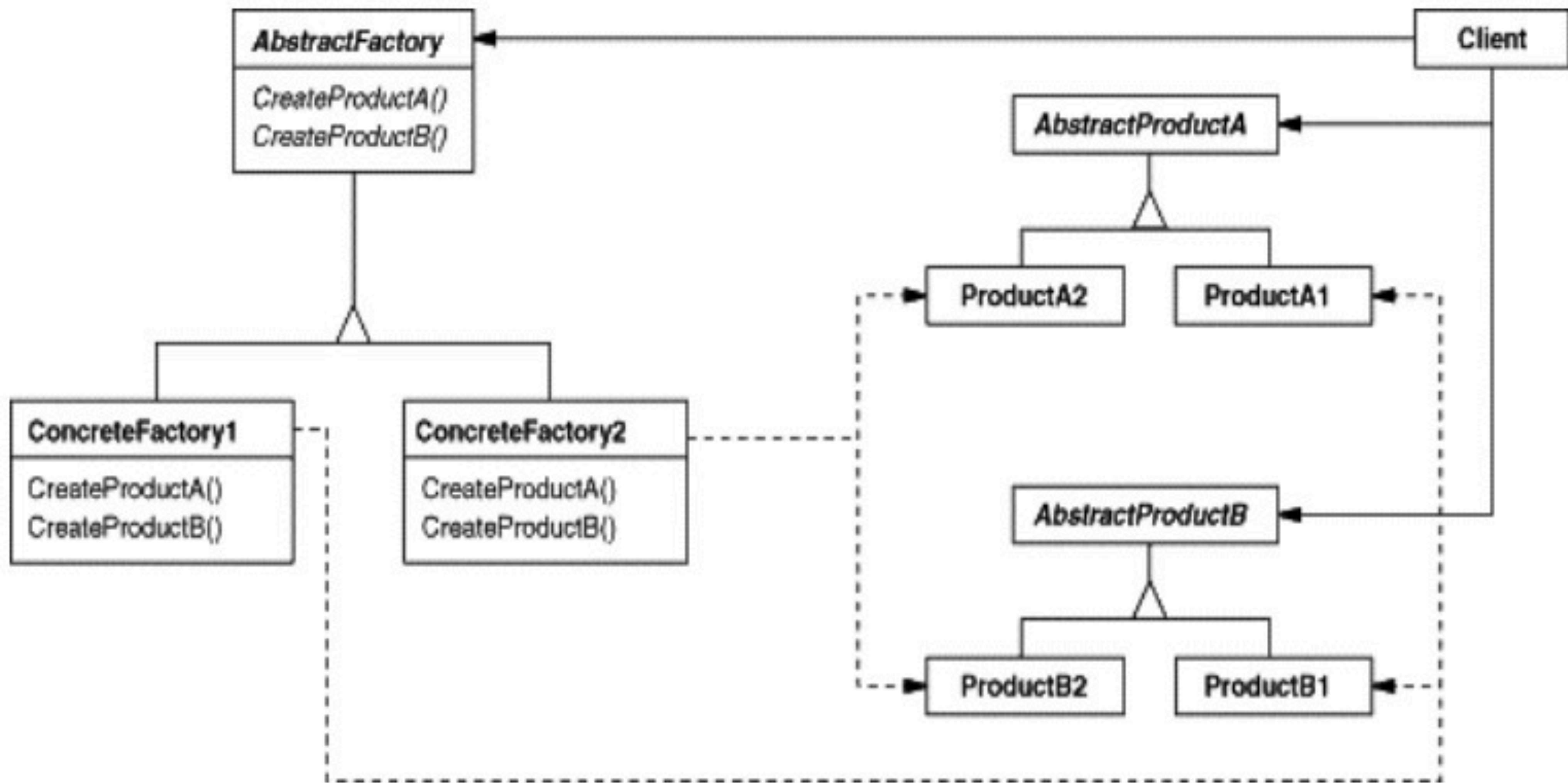


Design Pattern «Abstract Factory»

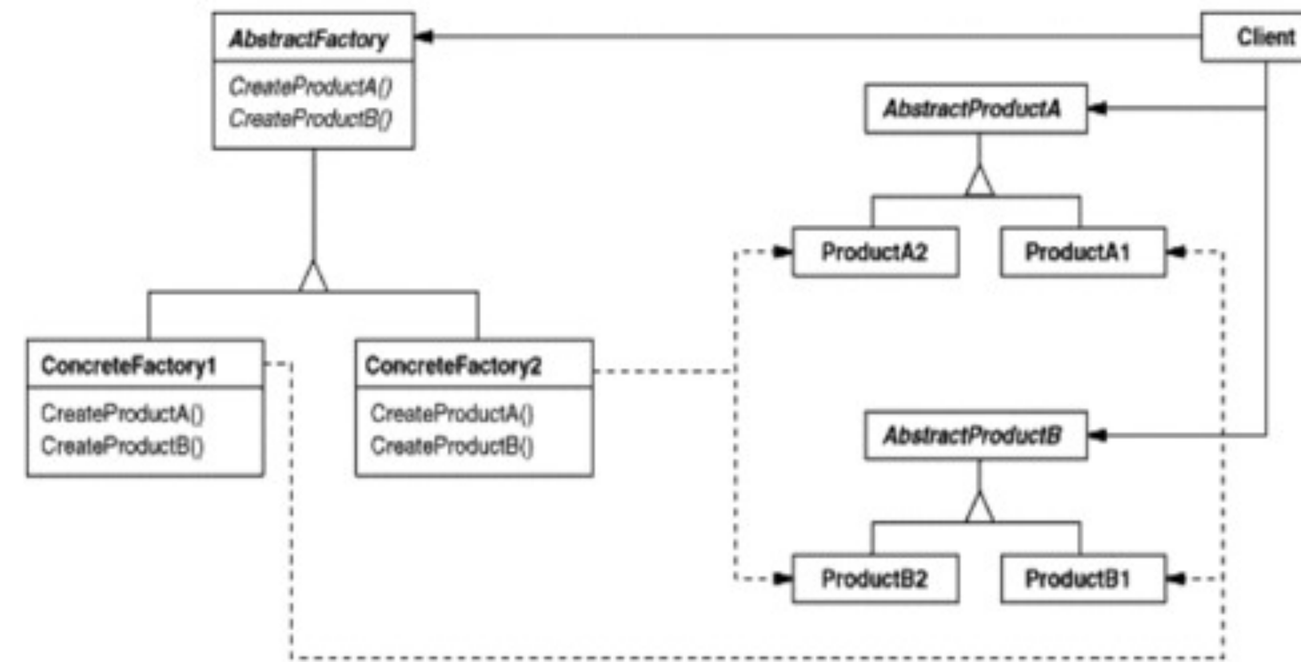
Abstract Factory Pattern : le problème

- ✓ Une application doit créer des familles d'objets dépendants sans spécifier leurs classes concrètes.
- ✓ *Le patron «Fabrique Abstraite» est très semblable au modèle «Method Factory».*

Abstract Factory Pattern : la solution



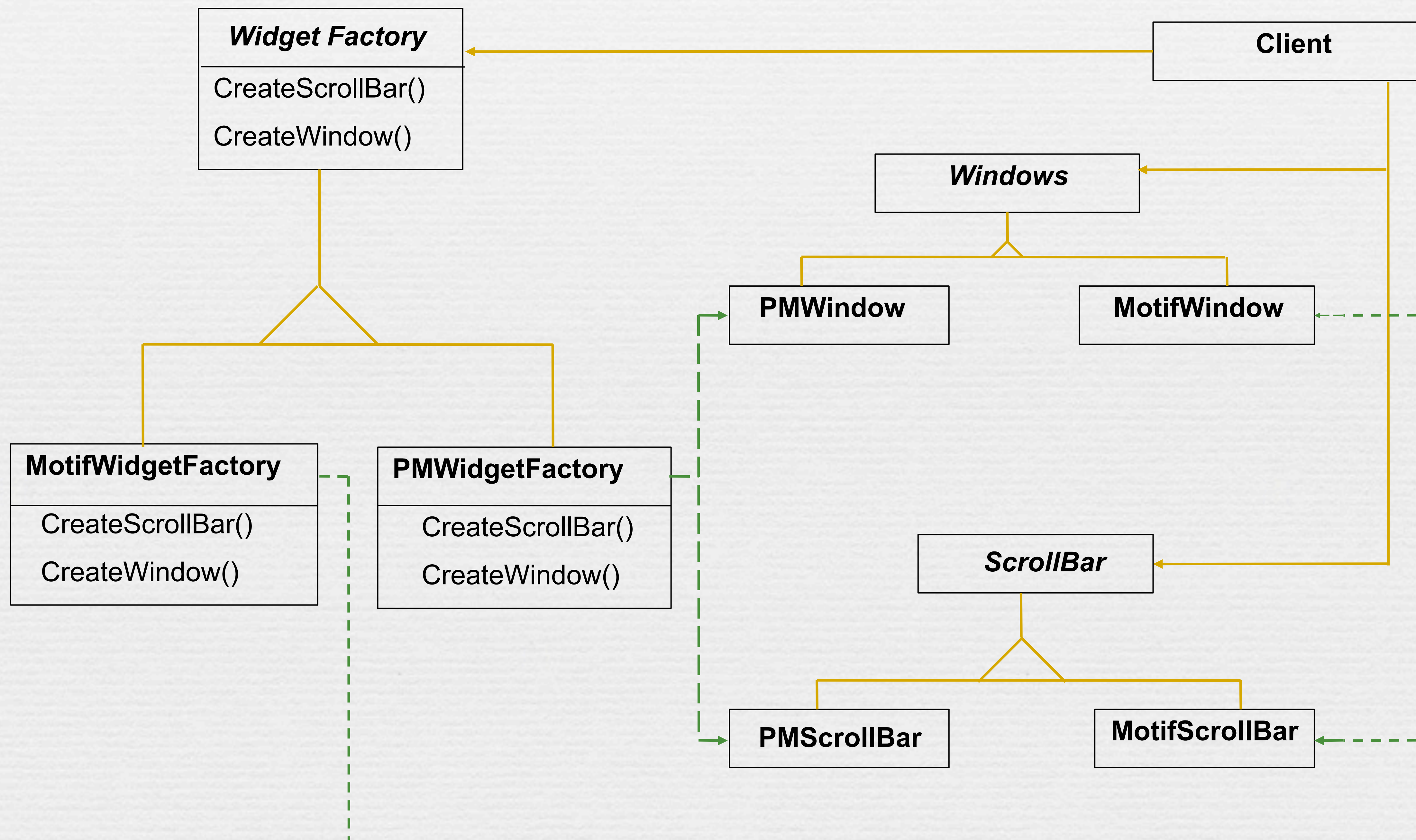
Abstract Factory Pattern : les rôles



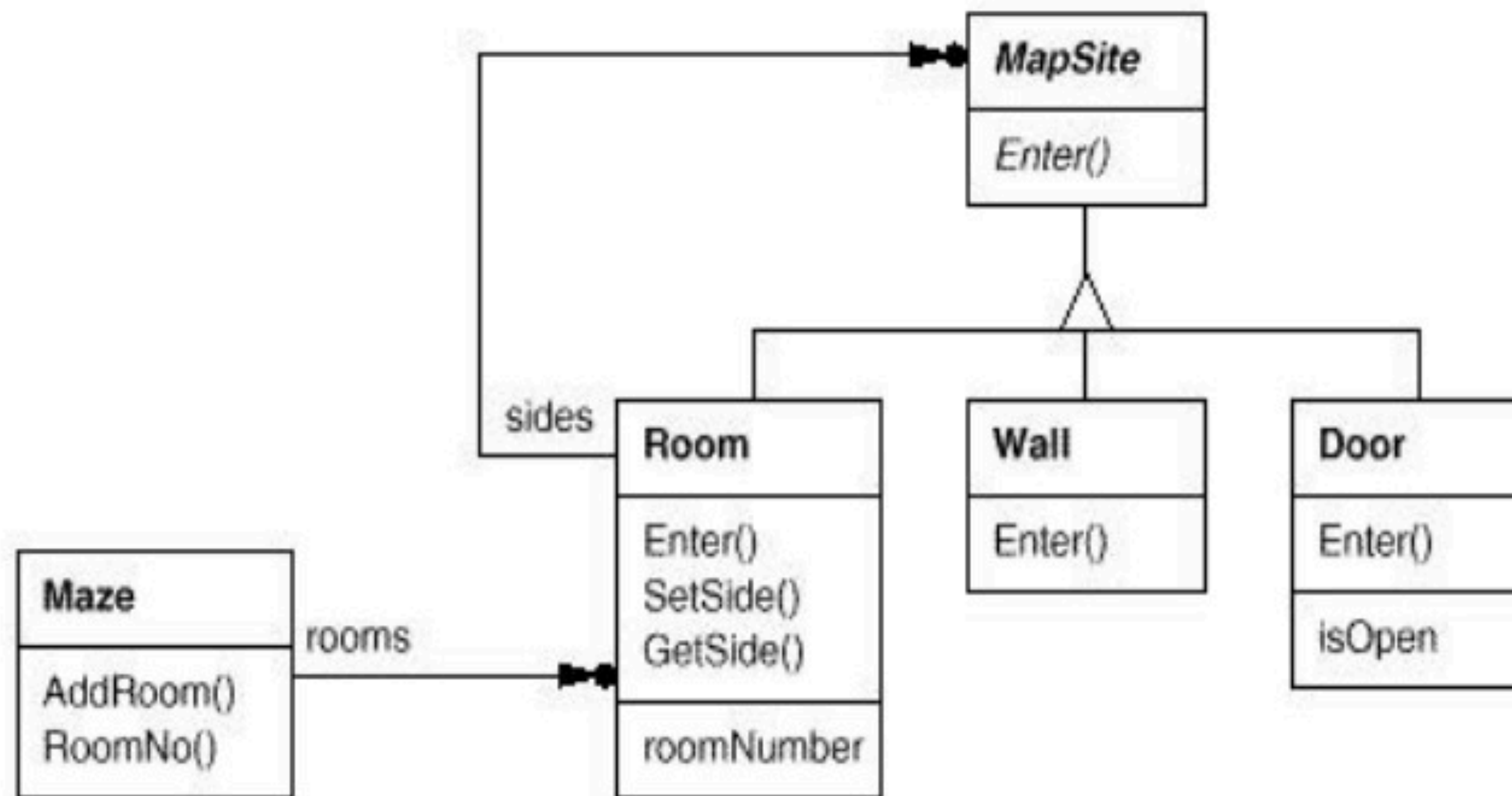
- ✓ AbstractFactory
  - ➡ Declares an interface for operations that create abstract product objects
- ✓ ConcreteFactory
  - ➡ Implements the operations to create concrete product objects
- ✓ AbstractProduct
  - ➡ Declares an interface for a type of product object
- ✓ ConcreteProduct
  - ➡› Defines a product object to be created by the corresponding concrete factory
  - ➡› Implements the AbstractProduct interface
- ✓ Client
  - ➡ Uses only interfaces declared by AbstractFactory and AbstractProduct classes

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

Abstract Factory Pattern : exemple



* Factory Pattern : exemple



Labyrinthe : une factory method (createMaze)

```
/**
 * MazeGame.
 */
public class MazeGame {
// Create the maze.
public Maze createMaze() {
   Maze maze = new Maze();
   Room r1 = new Room(1);
   Room r2 = new Room(2);
   Door door = new Door(r1, r2);
   maze.addRoom(r1);
   maze.addRoom(r2);
  r1.setSide(MazeGame.North, new Wall());
  r1.setSide(MazeGame.East, door);
  r1.setSide(MazeGame.South, new Wall());
  r1.setSide(MazeGame.West, new Wall());
r2.setSide(MazeGame.North, new Wall());
r2.setSide(MazeGame.East, new Wall());
r2.setSide(MazeGame.South, new Wall());
r2.setSide(MazeGame.West, door);
return maze;
}
```

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

```
/**
 * MazeGame.
 */
public class MazeGame {
 // Create the maze.
 public Maze createMaze() {
   Maze maze = new Maze();
   Room r1 = new Room(1);
  Room r2 = new Room(2);
   Door door = new Door(r1, r2);
   maze.addRoom(r1);
   maze.addRoom(r2);
  r1.setSide(MazeGame.North, new Wall());
  r1.setSide(MazeGame.East, door);
  r1.setSide(MazeGame.South, new Wall());
  r1.setSide(MazeGame.West, new Wall());
r2.setSide(MazeGame.North, new Wall());
r2.setSide(MazeGame.East, new Wall());
r2.setSide(MazeGame.South, new Wall());
r2.setSide(MazeGame.West, door);
return maze;
}
```

Labyrinthe … magique avec des pièces enchantées??

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

Factory Methods pour un Labyrinthe

```
//**
 * MazeGame with a factory methods.
 */
public class MazeGame {
    public Maze makeMaze() {
        return new Maze();}
    public Room makeRoom(int n) {
        return new Room(n);}
    public Wall makeWall() {
        return new Wall();}
    public Door makeDoor(Room r1, Room r2){
        return new Door(r1, r2);}
```


http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

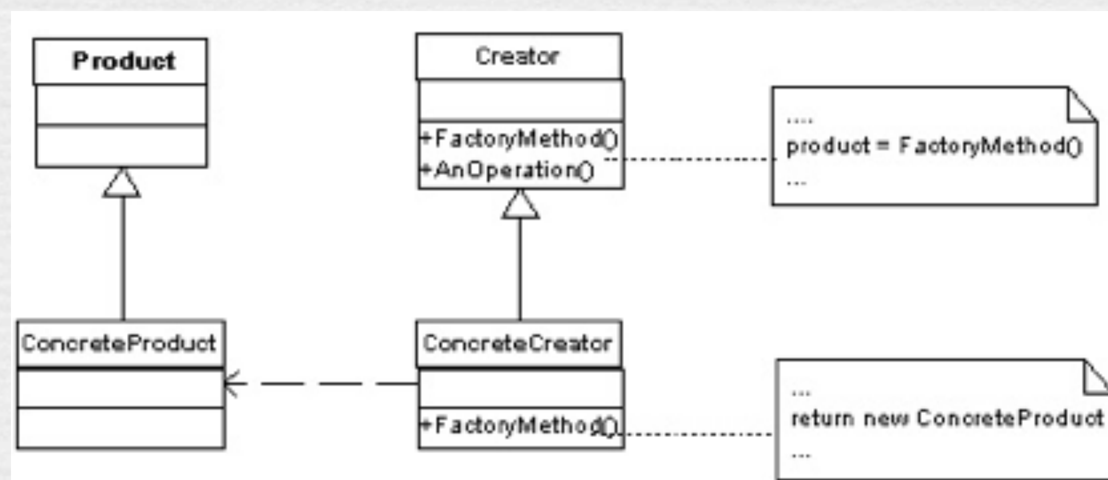
Factory Methods pour un Labyrinthe & usage

```
public Maze createMaze() {
  Maze maze = makeMaze();
  Room r1 = makeRoom(1);
  Room r2 = makeRoom(2);
  Door door = makeDoor(r1, r2);
  maze.addRoom(r1);
 maze.addRoom(r2);
 r1.setSide(MazeGame.North, makeWall());
 r1.setSide(MazeGame.East, door);
 r1.setSide(MazeGame.South, makeWall());
 r1.setSide(MazeGame.West, makeWall());
 r2.setSide(MazeGame.North, makeWall());
 r2.setSide(MazeGame.East, makeWall());
 r2.setSide(MazeGame.South, makeWall());
 r2.setSide(MazeGame.West, door);
 return maze;
 }
 }
```

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

Retour sur le labyrinthe enchanté

```
public class EnchantedMazeGame extends MazeGame {
    public Room makeRoom(int n) {
      return new EnchantedRoom(n);}
    public Wall makeWall() {
      return new EnchantedWall();}
    public Door makeDoor(Room r1, Room r2){
      return new EnchantedDoor(r1, r2);}
}
```



- ✓ Creator => MazeGame
- ✓ ConcreteCreator => EnchantedMazeGame (MazeGame is also a ConcreteCreator)
- ✓ Product => MapSite
- ✓ ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

```
// MazeFactory.
public class MazeFactory {
  public Maze makeMaze() {
       return new Maze();}
  public Room makeRoom(int n) {
     return new Room(n);}
public Wall makeWall() {
    return new Wall();}
public Door makeDoor(Room r1, Room r2) {
    return new Door(r1, r2);}
}
```

Pour créer des familles de labyrinthes

MazeFactory class est une collection de factory methods.

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

Pour créer des familles de labyrinthes

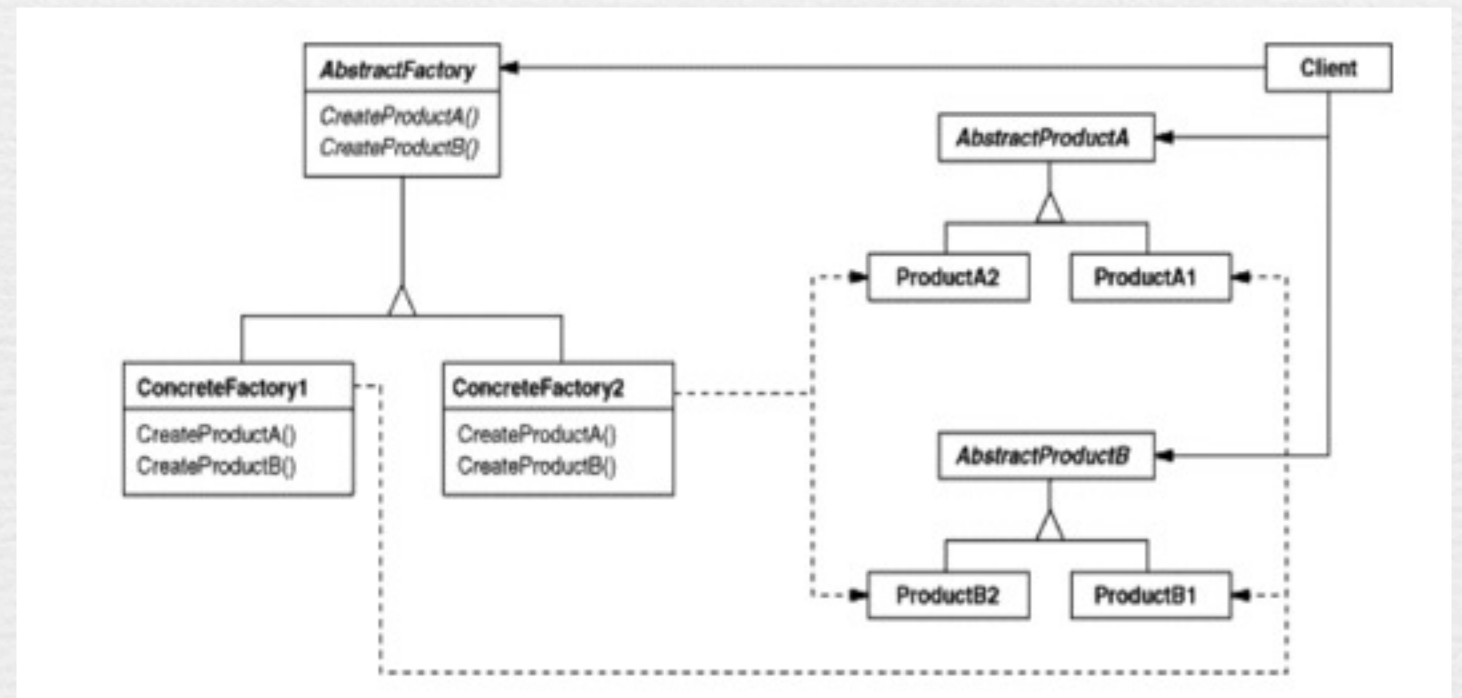
```
public class MazeGame {
  public Maze createMaze(MazeFactory factory) {
   Maze maze = factory.makeMaze();
   Room r1 = factory.makeRoom(1);
   Room r2 = factory.makeRoom(2);
   Door door = factory.makeDoor(r1, r2);
   maze.addRoom(r1);
   maze.addRoom(r2);
   r1.setSide(MazeGame.North, factory.makeWall());
   r1.setSide(MazeGame.East, door);
  ....
```

createMaze délégue la responsabilité de créer des labyrinthes à la factory.
MazeGame se focalise alors sur le jeu plus sur la construction des objets.

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

Pour créer des familles de labyrinthes

```
public class EnchantedMazeFactory extends MazeFactory {
  public Room makeRoom(int n) {
        return new EnchantedRoom(n);}
  public Wall makeWall() {
        return new EnchantedWall();}
  public Door makeDoor(Room r1, Room r2){
       return new EnchantedDoor(r1, r2);}
  }
....
```



- ✓AbstractFactory => MazeFactory
- ✓ConcreteFactory => EnchantedMazeFactory (MazeFactory is also a ConcreteFactory)
- ✓AbstractProduct => MapSite
- ✓ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor
- ✓Client => MazeGame

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

Design Pattern Decorator

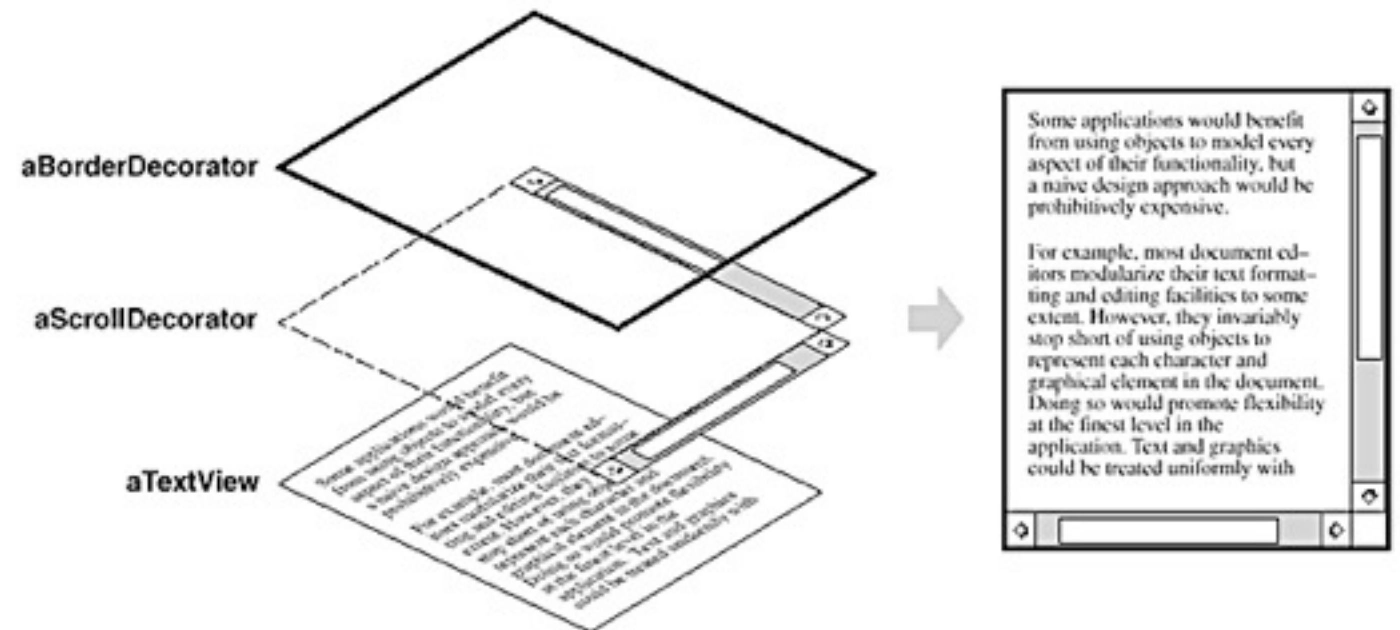
http://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

Patron Décorateur : le problème

- Il doit être possible d'ajouter dynamiquement des fonctionnalités à des objets.
- Le nombre de sous classes serait très grand si on devait définir autant de sous-classes que de variantes d'une classe ou bien elles doivent évoluer.

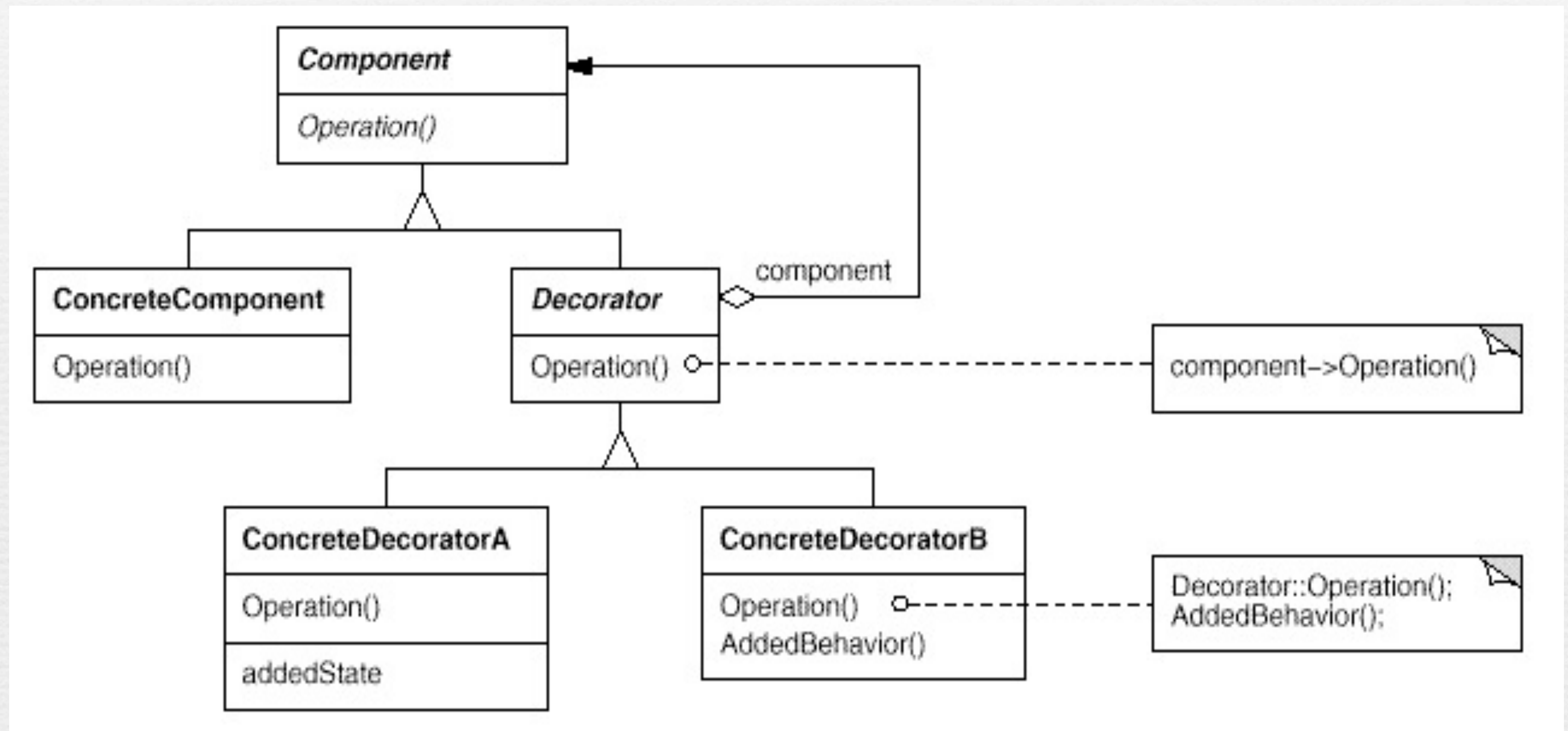
Motivation

- A TextView has 2 features:
  - borders:
    - 3 options: none, flat, 3D
  - scroll-bars:
    - 4 options: none, side, bottom, both



- How many Classes?
  - 3 x 4 = **12** !!!
    - e.g. TextView, TextViewWithNoBorder&SideScrollbar, TextViewWithNoBorder&BottomScrollbar, TextViewWithNoBorder&Bottom&SideScrollbar, TextViewWith3DBorder, TextViewWith3DBorder&SideScrollbar, TextViewWith3DBorder&BottomScrollbar, TextViewWith3DBorder&Bottom&SideScrollbar, ...

Patron Décorateur : la solution



Patron Décorateur : les rôles

✓Component

➡ définit l'interface des objets auxquels de nouvelles responsabilités peuvent être ajoutées dynamiquement.

✓ConcreteComponent

➡ un objet de base auquel de nouvelles responsabilités peuvent être ajoutées.

✓Decorator

➡définit une interface conforme à l'interface de «Component» ; il maintient une réference vers un objet composant

✓ConcreteDecorator

➡› ajoute des responsabilités à un «component»

Decorator en action

- On calcule le prix d'un café en fonction des ingrédients qui lui sont ajoutés : lait, sucre, .. en utilisant le pattern décorateur.

Design Pattern Adaptator

Patron Adapter : le problème

- " Permettre l'adaptation d'une classe à une autre interface qui est attendue par le client : autoriser ainsi des classes ayant des interfaces incompatibles à collaborer. "
- Synonyme : wrapper

motivation

Parfois, une bibliothèque de code ne peut pas être utilisée car son interface est incompatible avec l'interface requise par une application. Nous ne pouvons pas changer l'interface de la bibliothèque, puisque nous ne pouvons pas accéder ou modifier le code source .... De même il ne serait pas raisonnable d'avoir autant de versions d'une bibliothèque que de ses usages.

http://miageprojet2.unice.fr/User:SimonUrli/Patrons_de_Conception

Patron Adapter : solution