

Design Patterns

M. Blay-Fornarino & S. Urli

Motivations ... (toujours les mêmes)

- ✿ Faire une conception et un développement de qualité :
 - Extensibilité
 - Flexibilité
 - Maintenabilité
 - Réutilisabilité
 - Clarté



Martin Fowler

@martinfowler



Suivre

The same solution can be a good pattern in some contexts and an antipattern in others. Its value depends on the context that you use it in

Voir la traduction

RETTWEETS

483

J'AIME

381



13:12 - 8 juin 2016



483

381

...



Répondre à @martinfowler



Richard Bishop @bitsandhops · 8 juin

@martinfowler You mean I have to think? This is an outrage!



2

2

...



Lee Hawkins @therockertester · 9 juin

@martinfowler @charrett So you're suggesting there's no such thing as "best practices"? Tell that to those trying to standardize testing



2

2

...



The Guy @Guygies · 8 juin

@martinfowler also known as the "don't use your Lamborghini to transport horse manure" rule.

Toujours pas de recette «magique»

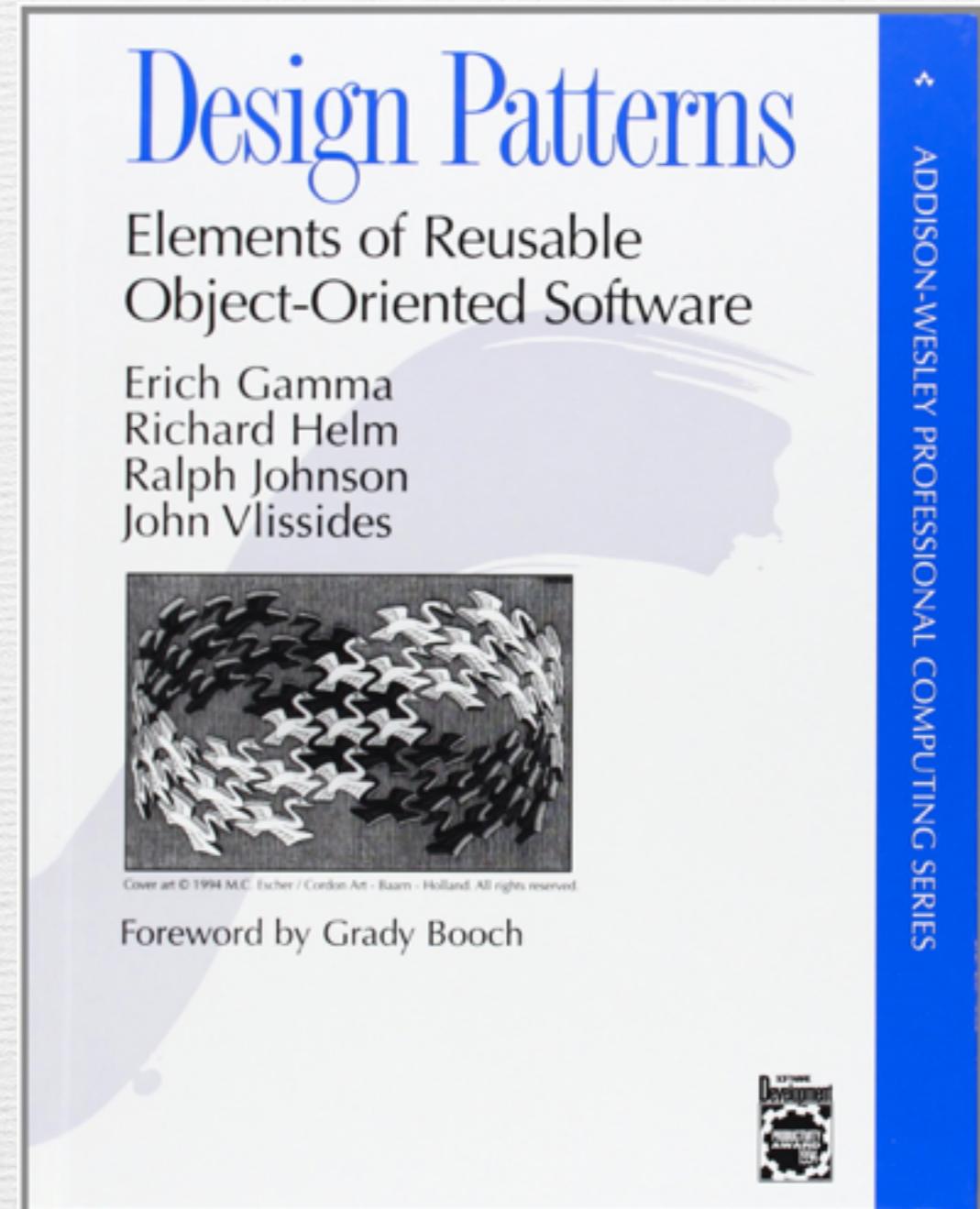
... mais outre les bonnes pratiques :

- KIS : Keep It Simple
- DRY : Don't Repeat Yourself
- YAGNI : You Ain't Gonna Need It
- SOLID

Patron de conception : un outil supplémentaire

Patrons de conception ?

- ❖ “ Généralisation d'une solution à un problème de conception récurrent par la description des classes et objets communicants ”



Patrons de conception? Comment sont-ils définis?

- ❖ Identification d'un problème de conception récurrent
 - Schématisation du problème de manière générique
 - Description d'une solution sous la forme d'un patron

Patrons de conception? Comment les utiliser?

- ❖ Identifier un problème dont le motif a fait l'objet d'une solution
- ❖ Rechercher le patron de conception adapté
- ❖ Appliquer et adapter la solution proposée par le patron de conception

Historique

- ~ 1977/79 : Christopher Alexander - Patron de conception pour l'architecture des villes et bâtiments
- ~ 1987 : Kent Beck et Ward Cunningham - Papier à OOPSLA sur l'utilisation de patrons de conception pour la programmation orientée objet
- ~ 1994 : le «Gang of Four» (GoF) (Gamma, Helm, Johnson and Vlissides) publie Design Patterns: Elements of Reusable Object-Oriented Software - Présentation des 23 patterns fondamentaux

Classification des patrons

❖ Patrons de création :

- dédiés à la création des objets.
- visent l'indépendance entre création et utilisation des objets.

❖ Patrons de structure :

- dédiés à la composition des objets.
- visent à conserver une bonne séparation des préoccupations.

❖ Patrons de comportement :

- dédiés à la communication entre les objets
- visent à conserver de la flexibilité dans les liens de communication.

Classification

❖ 23 Patrons fondamentaux

→ Patrons de création :

- *Abstract Factory*, Builder, Factory method,
- Prototype, Singleton

→ Patrons de structure :

- Adapter, Bridge, *Composite*, Decorator, Facade, Flyweight, Proxy

→ Patrons de comportement :

- Chain-of-responsibility, Command, Interpreter, Iterator, Mediator, Memento, *Observer*, State, Strategy, Template Method, Visitor

Autour des patrons de conception

- Patrons GRASP (General Responsibility Assignment Software Patterns)
- Principes SOLID
- Patrons d'architecture (ex : MVC, layers, etc)
- Patrons de gestion de la concurrence (pool de threads, etc)

Sources

- ❖ Les design patterns en Java - Steven John Metsker, William C. Wake
- ❖ Head first design patterns - Eric et Elisabeth Freeman
- ❖ The design pattern Smalltalk Companion
- ❖ Sherman R. Alpert, Kyle Brown, Bobby Woolf
- ❖ <http://blog.codinghorror.com>
- ❖ http://en.wikipedia.org/wiki/Software_design_pattern

Design Pattern State

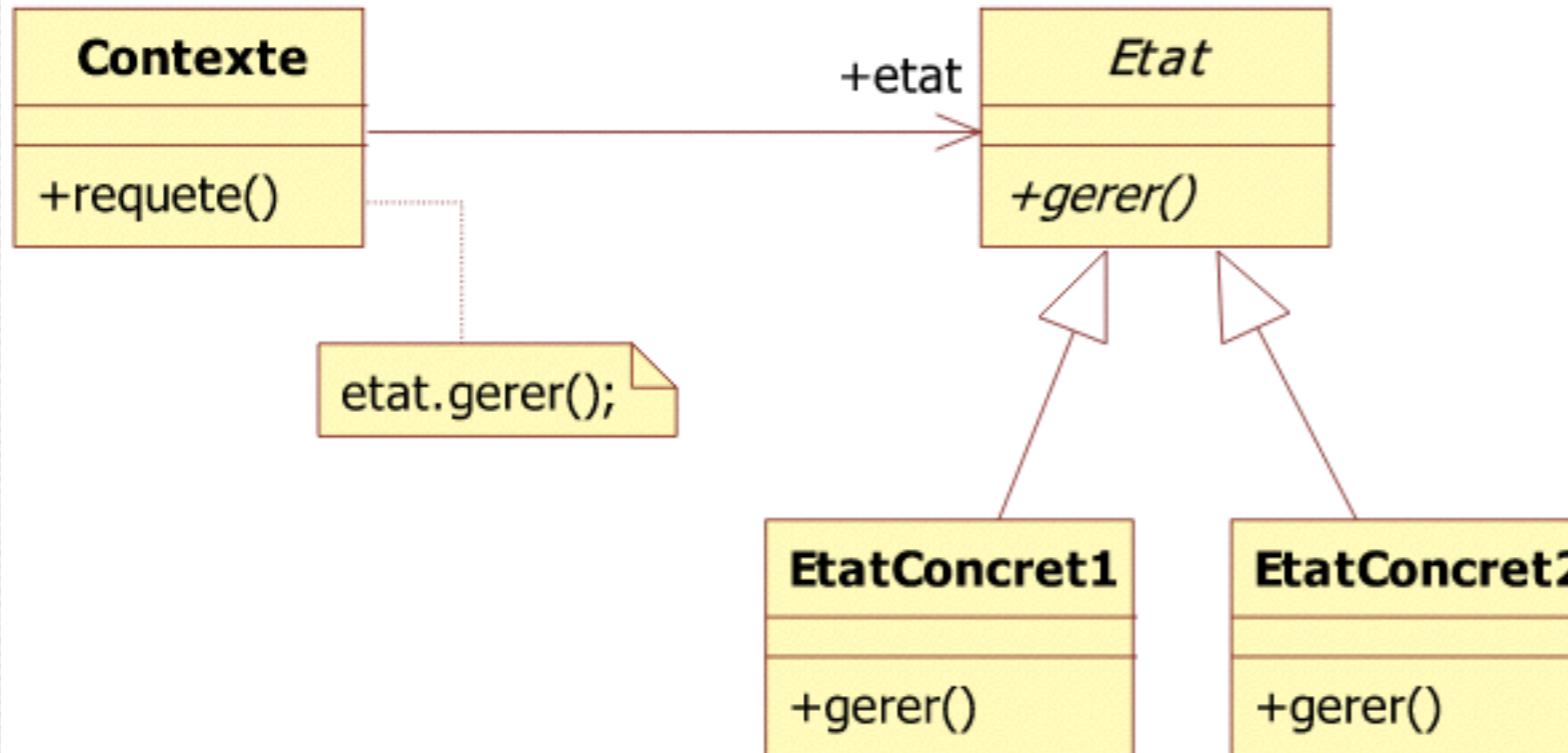
State Pattern : le problème par l'exemple

```
public class Pizza {  
  
    public final static int COOKED = 0;  
    public final static int BAKED = 1;  
    public final static int DELIVERED = 2;  
  
    private String name;  
  
    int state = COOKED;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getState() {  
        return state;  
    }  
    public void setState(int state) {  
        this.state = state;  
    }  
    public void bake() throws Exception {  
        if(state == COOKED) {  
            System.out.print("Baking the pizza...");  
            state = BAKED;  
        }  
        else if(state == BAKED) {  
            throw new Exception("Can't bake a pizza  
already baked");  
        }  
        else if(state == DELIVERED) {  
            throw new Exception("Can't bake a pizza  
already delivered");  
        }  
    }  
}
```

```
    public void deliver() throws Exception {  
  
        if(state == COOKED) {  
            throw new Exception("Can't deliver a  
pizza not baked yet");  
        }  
        else if(state == BAKED) {  
            System.out.print("Delivering the  
pizza...");  
            state = DELIVERED;  
        }  
        else if(state == DELIVERED) {  
            throw new Exception("Can't deliver a  
pizza already delivered");  
        }  
    }  
}
```

DANS CETTE VERSION
ON NE PEUT PAS
FACILEMENT RAJOUTER
UN ETAT ... UNE
METHODE DO SERAIT
MIEUX!!!

Diagramme de classes :



State Pattern :
la solution
généralisée

Permet à un objet de modifier son comportement, quand son état interne change. Tout se passe comme si l'objet changeait de classe.

- ✓ Extensibilité
- ✓ 1.1 Factorisation de protocole pour tous les états
- ✓ 1.2 L'ajout ou la suppression d'un état n'implique pas de modification de code
- ✓ Gestion simplifiée
 - 2.1 *Changement d'état possible à l'exécution sans destruction*
 - 2.2 *Découplage du comportement de chaque état*

```

public class Pizza {
    private String name;
    //State initialization
    private PizzaState state = new CookedPizzaState(this);

    public Pizza() { }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    public PizzaState getState() {
        return state;
    }

    public void setState(PizzaState state) {
        this.state = state;
    }

    public void bake() throws Exception {
        this.state.bake();
    }

    public void deliver() throws Exception {
        this.state.deliver();
    }
}

```

```

public abstract class PizzaState {
    protected Pizza pizza;

    public PizzaState(Pizza pizza){
        this.pizza = pizza;
    }
    abstract public void bake() throws Exception;
    abstract public void deliver() throws Exception;
}

public class BakedPizzaState extends PizzaState {
    public BakedPizzaState(Pizza pizza) {
        super(pizza);
    }
    public void bake() throws Exception {
        throw new Exception("Can't bake a pizza already baked");
    }
    public void deliver() throws Exception {
        System.out.print("Delivering the pizza...");
        pizza.setState(new DeliveredPizzaState(this.pizza));
    }
}

```

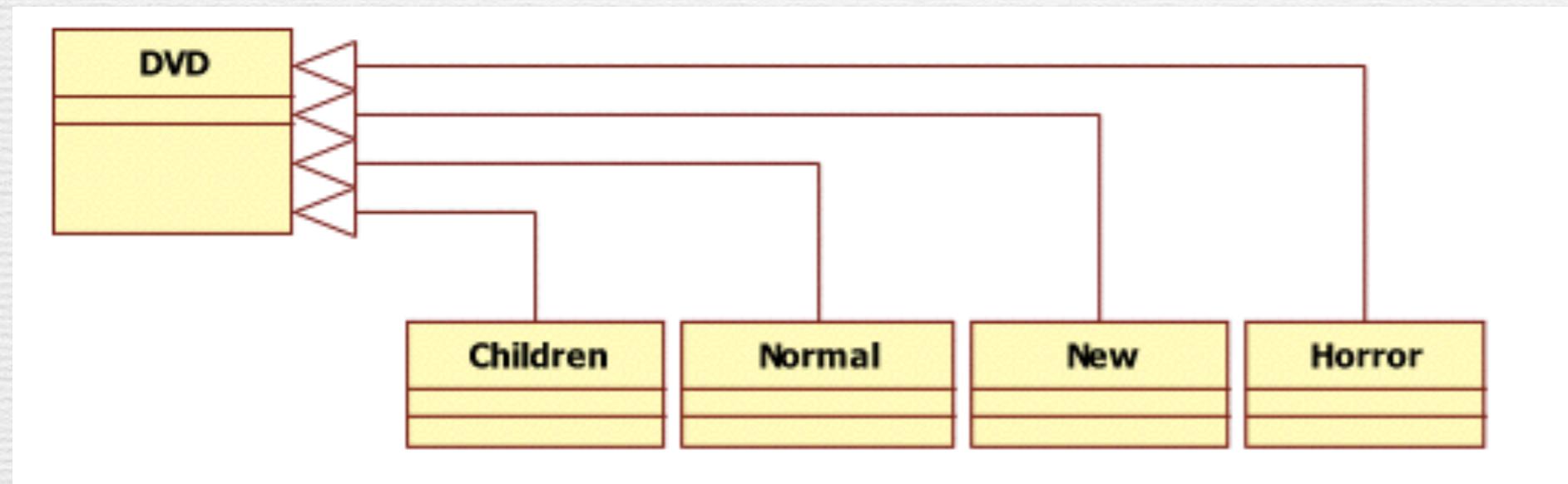
State Pattern : la solution

A vous...

**ATTENTION ICI ON NE GERE PAS LE
CHANGEMENT D'ETAT
AUTOMATIQUEMENT**

- ✓ Modéliser le fonctionnement d'une vidéothèque. La vidéothèque met à disposition de ses clients des DVD selon trois catégories : Enfant, Normal et Nouveauté. Un DVD est dans la catégorie Nouveauté pendant quelques semaines, puis passe dans l'une des autres catégories. Le prix des DVD dépend de la catégorie. Il est probable que le système évolue pour que la catégorie Horreur soit ajoutée.

Pattern Etat Abimé



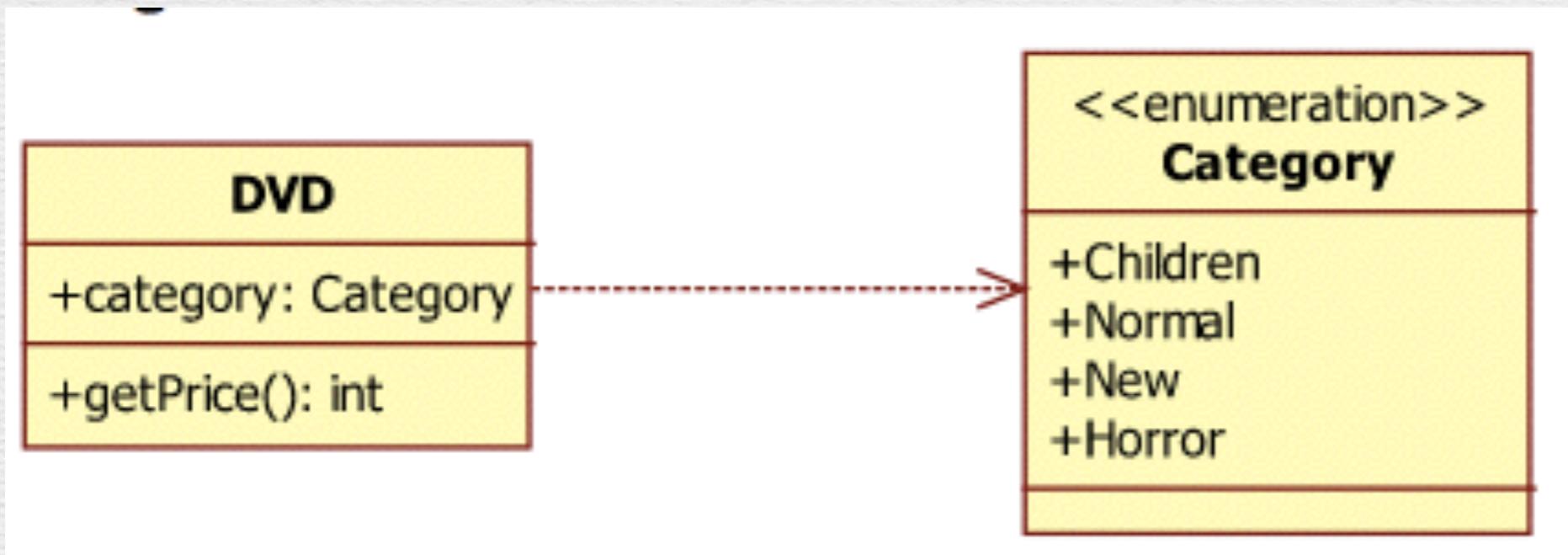
1. Extensibilité

- ✓ 1.1 Factorisation de protocole pour tous les états
- ✓ 1.2 L'ajout ou la suppression d'un état n'implique pas de modification de code

2. Gestion simplifiée

- ✗ 2.1 Changement d'état possible à l'exécution sans destruction
- ✗ 2.2 Découplage du comportement de chaque état

Pattern Etat Abimé



1. Extensibilité

- ✖ 1.1 Factorisation de protocole pour tous les états
- ✖ 1.2 L'ajout ou la suppression d'un état n'implique pas de modification de code

2. Gestion simplifiée

- ✓ 2.1 Changement d'état possible à l'exécution sans destruction
- ✖ 2.2 Découplage du comportement de chaque état

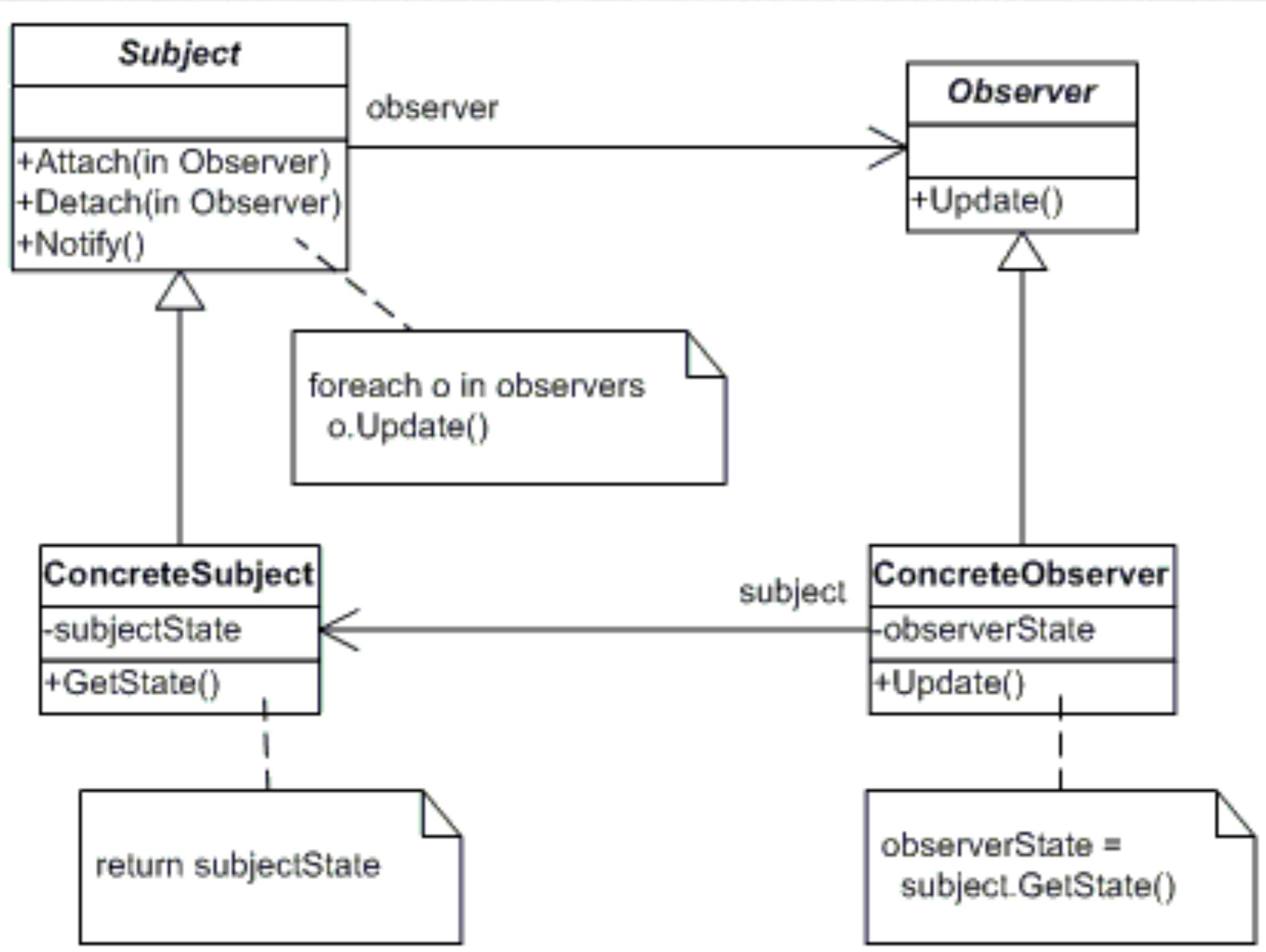
Design Pattern Observer

«Pattern Observer» : le problème

✓ Problème : Mettre en oeuvre une relation de «un vers plusieurs» objets afin que plusieurs objets puissent être notifiés du changement d'état d'un objet et puisse réagir.

Il est très utilisé en IHM, mais peut être appliqué dans bien d'autres cas.

Pattern Observer : Solution



«Pattern Observer» : les rôles

✓ Rôles : Un sujet et des «observers» (le sujet doit devenir «observable»)

✓ Responsabilités :

→ Le sujet :

- ▶ notifie les observeurs quand il «change»
- ▶ permet aux observeurs de s'(de-)enregistrer.

→ Les observeurs

- ▶ acceptent les notifications

Pattern Observer : Solution

→ Sujet Abstrait (Observable) :

- ▶ gère les observeurs (*AddObserver(Observer)*)
- ▶ notifie les observeurs (*notifyObservers*)

→ Sujet (Observable) :

- ▶ A chaque changement d'état, il «notifie» les observeurs (*notify*)
- ▶ Il peut donner son état (*getState*)

→ Observateur

- ▶ Se met à jour quand il est notifié (*update*)

→ Observateur (Abstrait)

- ▶ Peut être notifié (*update*)



«Pattern Observer» et autres en action

✓ Un forum

- On peut poster des messages dans le forum : un message à un titre.

✓ Des messages

- Certains messages sont des alertes, d'autres des annonces,...

✓ Des abonnés

- Un abonné peut recevoir des messages dans ses boites de messages.

✓ Dès qu'un message est posté sur le forum, tous les abonnés sont notifiés.

- Les abonnés enregistrent le message dans leur boite « inbox »
- Certains abonnés VIP filtrent les messages reçus en fonction du titre.

«Pattern Observer» en java

✓ Le sujet abstrait : classe abstraite
java.util.observable

✓ Le sujet concret :

- Votre classe qui hérite de observable
- *C'est à vous d'appeler notifyObservers*

G Observable	
■	changed
■	obs
●	Observable()
●	addObserver(Observer) : void
◆	clearChanged() : void
●	countObservers() : int
●	deleteObserver(Observer) : void
●	deleteObservers() : void
●	hasChanged() : boolean
●	notifyObservers() : void
●	notifyObservers(Object) : void
◆	setChanged() : void

✓ L'observateur abstrait : Interface
java.util.Observer

✓ L'observateur concret :

- Votre classe qui «implémente» Observer
- *doit implémenter update*

▼ I Observer
● update(Observable, Object) : void

«Pattern Observer» en java exemple

```
import java.util.Observable;

public class ObservableObject extends Observable
{
    private int n = 0;
    public ObservableObject(int n)
    {
        this.n = n;
    }
    public void setValue(int n)
    {
        this.n = n;
        setChanged();
        notifyObservers();
    }
    public int getValue()
    {
        return n;
    }
}
```

<http://www.javaworld.com/article/2077258/learn-java/observer-and-observable.html>

«Pattern Observer» en java exemple

```
import java.util.Observer;
import java.util.Observable;
public class TextObserver implements Observer
{
    private ObservableObject ov = null;
    public TextObserver(ObservableObject ov)
    {
        this.ov = ov;
    }
    public void update(Observable obs, Object obj)
    {
        if (obs == ov)
        {
            System.out.println(ov.getValue());
        }
    }
}
```

<http://www.javaworld.com/article/2077258/learn-java/observer-and-observable.html>

«Pattern Observer» en java exemple

```
public class Main
{
    public Main()
    {
        ObservableValue ov = new ObservableValue(0);
        TextObserver to = new TextObserver(ov);
        ov.addObserver(to);
    }
    public static void main(String [] args)
    {
        Main m = new Main();
    }
}
```

<http://www.javaworld.com/article/2077258/learn-java/observer-and-observable.html>

DP Observateur : Résumé

✓ Intention :

→ Définit une interdépendance de type un à plusieurs, de telle façon que quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour.

✓ Applicabilité : Utilisez l'Observateur dans les situations suivantes :

- Quand un concept a deux représentations, l'une dépendant de l'autre. Encapsuler ces deux représentations dans des objets distincts permet de les réutiliser et de les modifier indépendamment.
- Quand la modification d'un objet nécessite de modifier les autres, et que l'on ne sait pas combien sont ces autres.
- Quand un objet doit être capable de faire une notification à d'autres objets sans faire d'hypothèses sur la nature de ces objets. En d'autres termes, quand ces objets ne doivent pas être trop fortement couplés.

http://www.goprod.bouhours.net/?page=pattern&pat_id=16

Design Pattern Composite

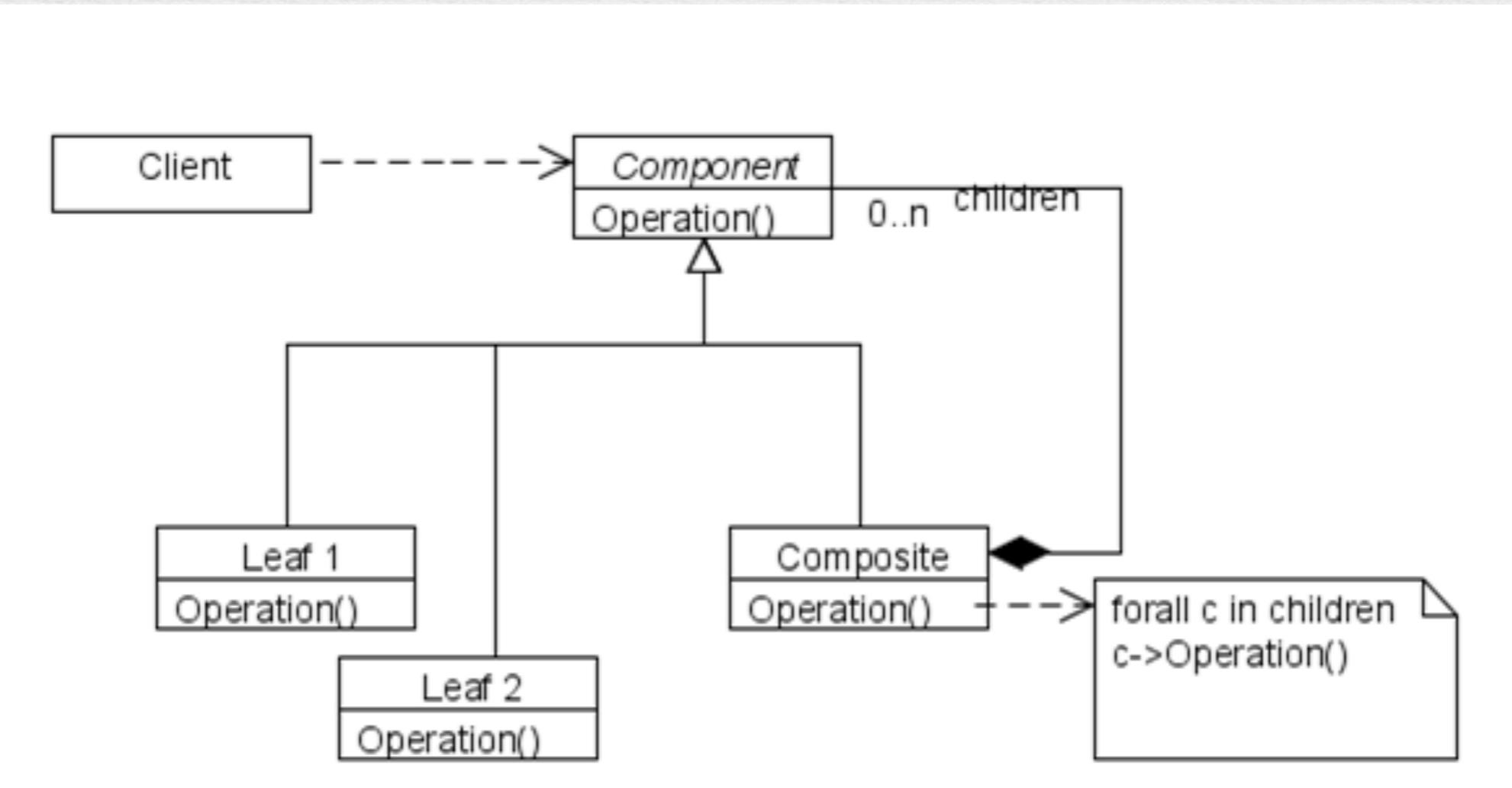
«Pattern Composite» : le problème

✓ 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

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

→ Composite

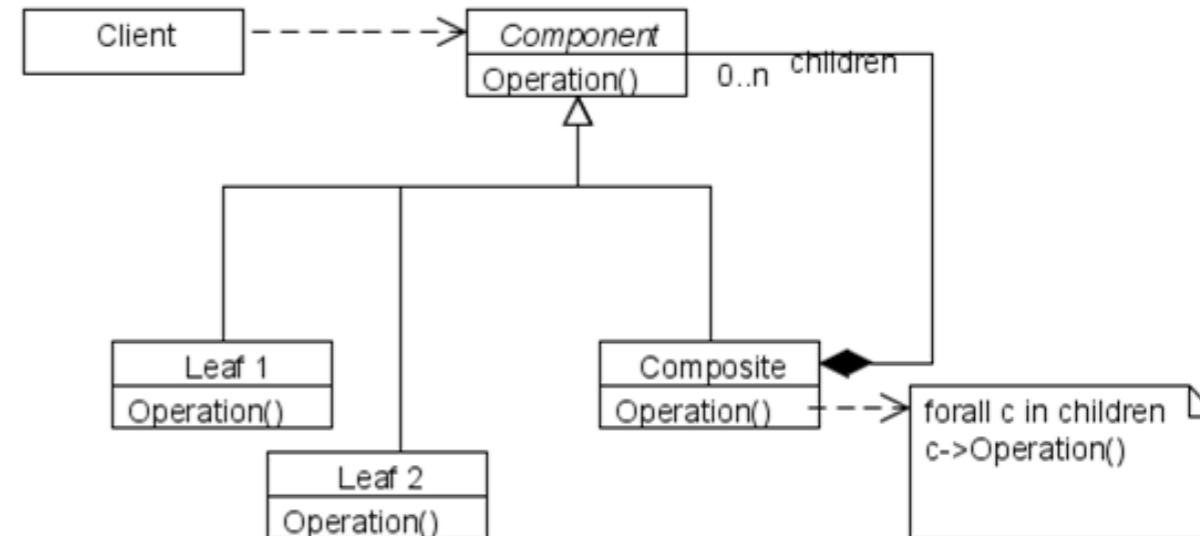
- ▶ définit le comportement des composants ayant des «enfants» dans la hiérarchie de la composition
- ▶ référence les composants fils (ils peuvent eux-même être composites!)

→ Leaf

- ▶ définit le comportement des objets primitifs dans la composition

→ Client

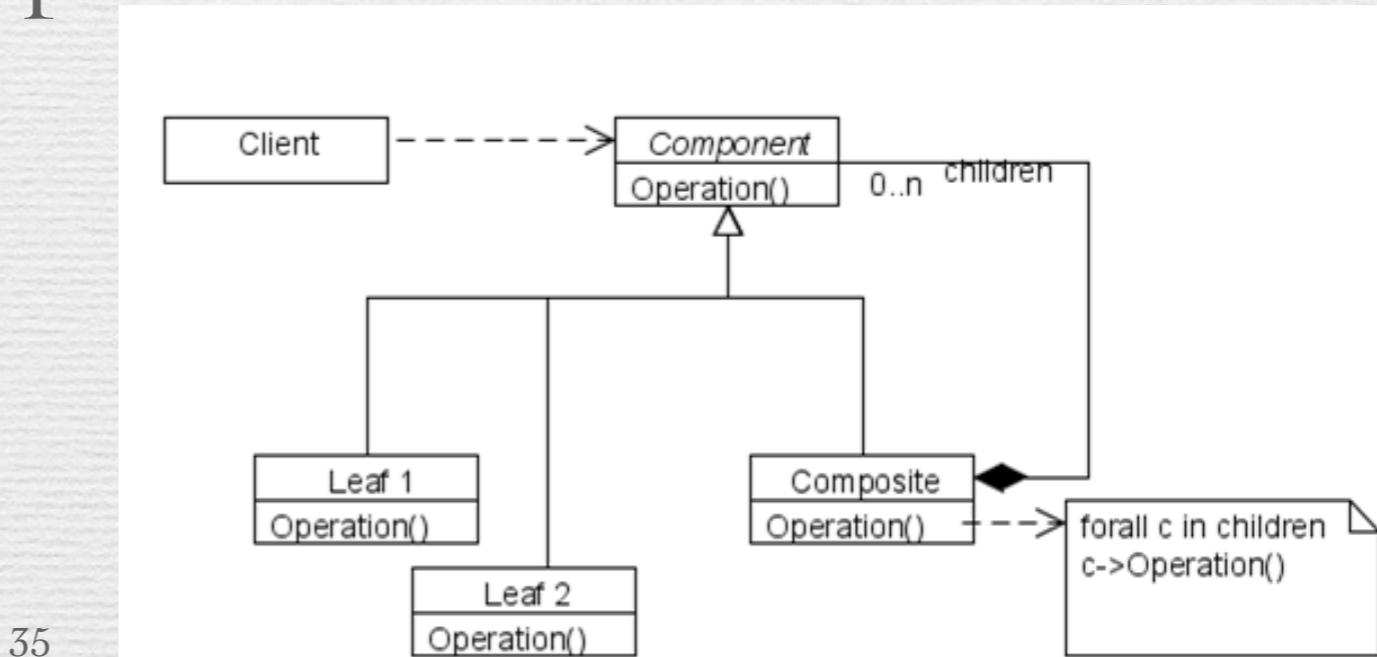
- ▶ manipule les objets de la composition au travers de l'interface Component



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 !



DP Composite : Résumé

✓ Intention :

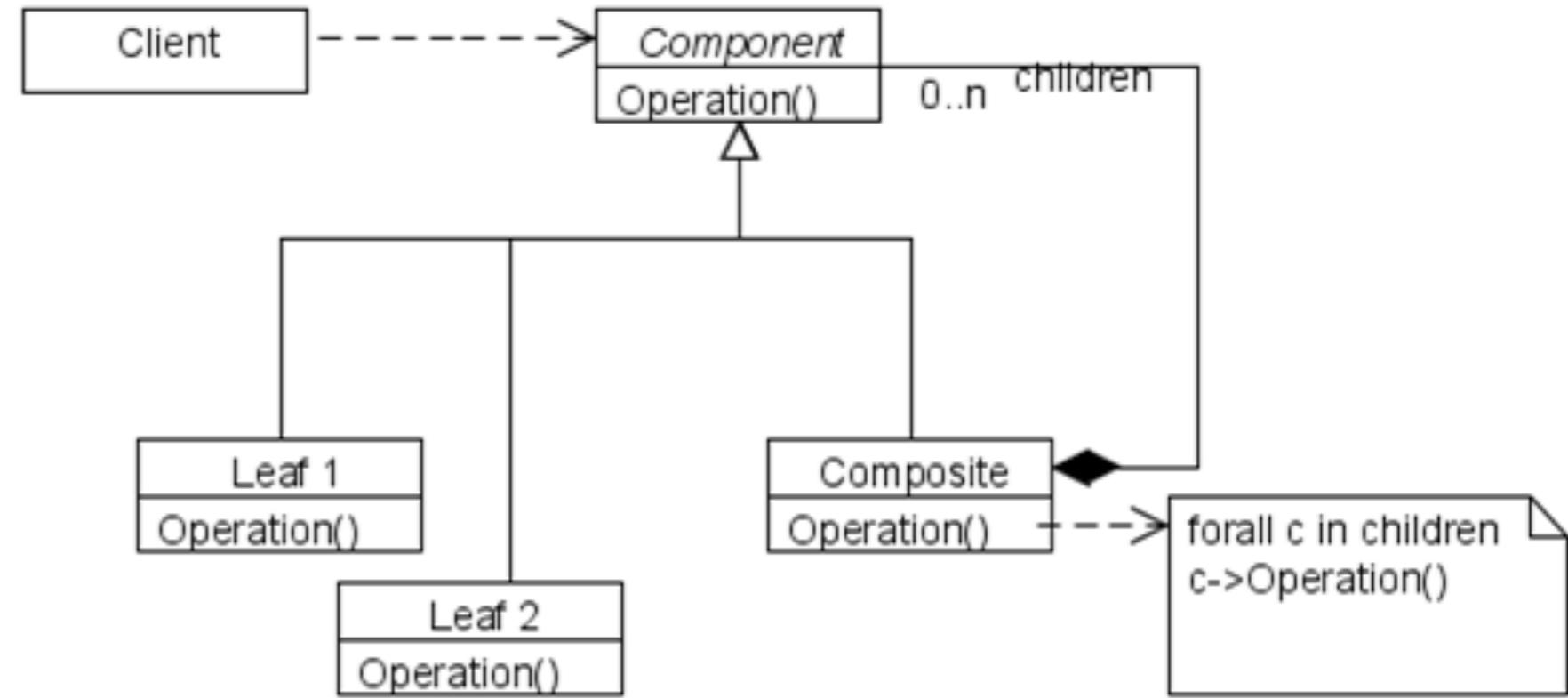
- Compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé.
- Permet au client de traiter d'une unique façon les objets et les combinaisons d'objets.

✓ Applicabilité : Utilisez le **Composite** lorsque :

- Vous souhaitez représenter des hiérarchies de l'individu.
- Vous souhaitez que le client n'ait pas à se préoccuper de la différence entre "combinaisons d'objets" et "objets individuels". Les clients pourront traiter de façon uniforme tous les objets de la structure composite.

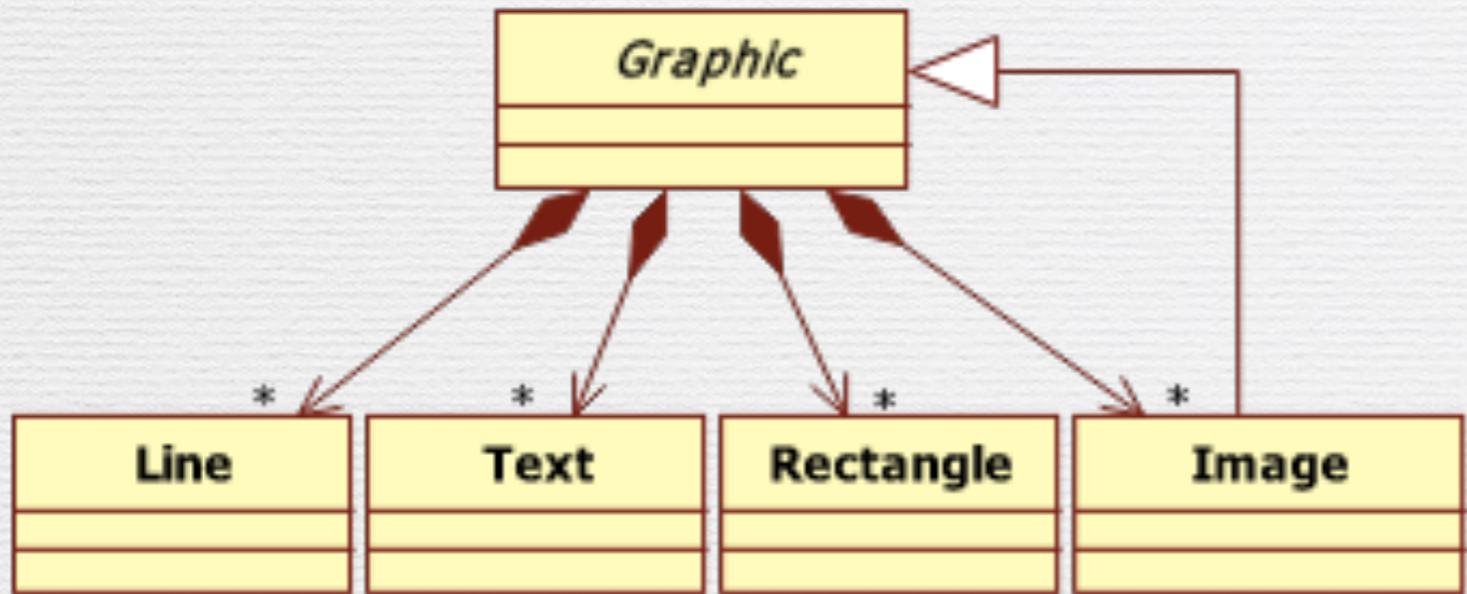
DP Composite : Résumé

Modéliser un système permettant de dessiner un graphique. Un graphique est composé de lignes, de rectangles, de textes et d'images, une image pouvant être composée d'autres images, de lignes, de rectangles et de textes.



✓ Points forts :

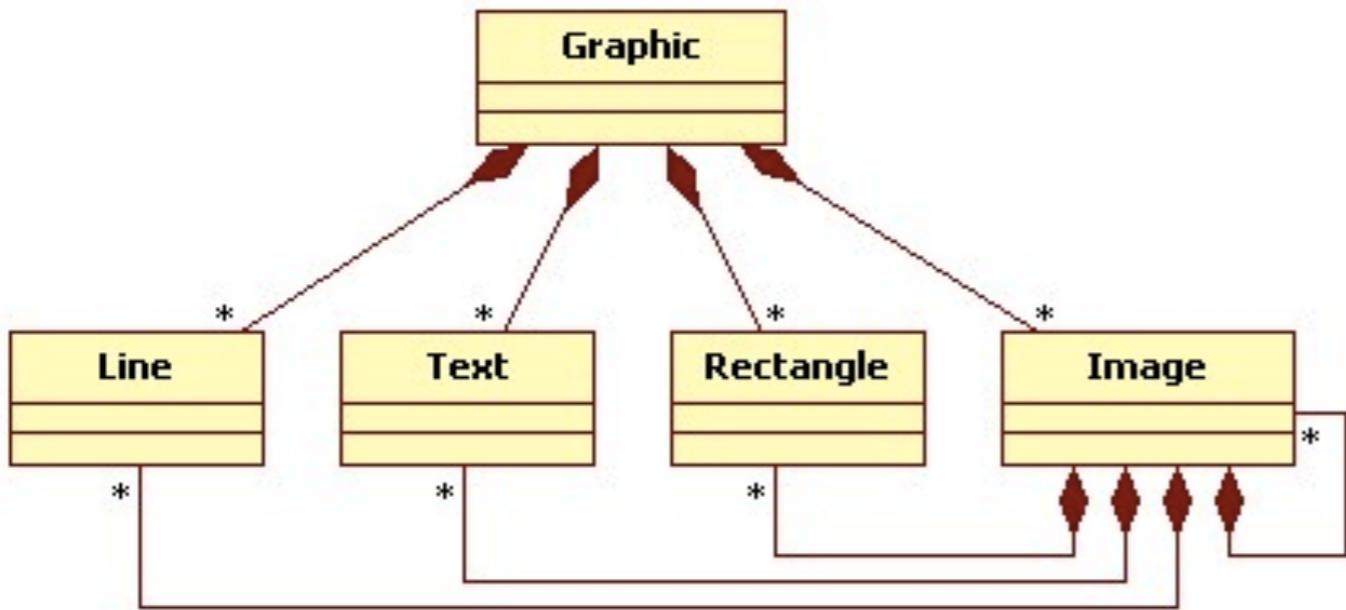
- ✓ 1. Découplage et extensibilité
 - ✓ 1.1 Factorisation maximale de la composition
 - ✓ 1.2 L'ajout ou la suppression d'une feuille n'implique pas de modification de code
 - ✓ 1.3 L'ajout ou la suppression d'un composite n'implique pas de modification de code
- ✓ 2. Protocole uniforme
 - ✓ 2.1 Protocole uniforme sur les opérations des objets composés
 - ✓ 2.2 Protocole uniforme sur la gestion de la composition
 - ✓ 2.3 Point d'accès unique pour la classe client



Quels sont les points faibles de ce modèle?

Modéliser un système permettant de dessiner un graphique. Un graphique est composé de lignes, de rectangles, de textes et d'images, une image pouvant être composée d'autres images, de lignes, de rectangles et de textes.

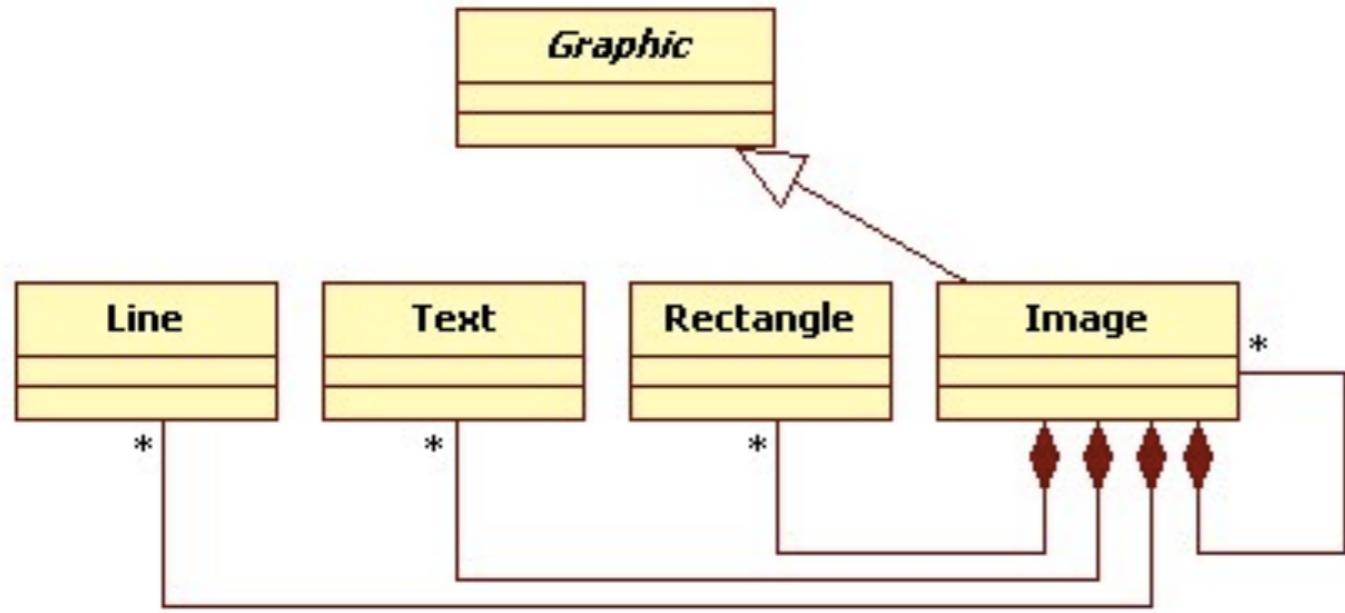
- 1. Découplage et extensibilité
- 1.1 Factorisation maximale de la composition
- 1.2 L'ajout ou la suppression d'une feuille n'implique pas de modification de code
- 1.3 L'ajout ou la suppression d'un composite ne n'implique pas de modification de code
- 2. Protocole uniforme
 - 2.1 Protocole uniforme sur les opérations des objets composés
 - 2.2 Protocole uniforme sur la gestion de la composition
 - 2.3 Point d'accès unique pour la classe client



Quels sont les points faibles de ce modèle?

Modéliser un système permettant de dessiner un graphique. Un graphique est composé de lignes, de rectangles, de textes et d'images, une image pouvant être composée d'autres images, de lignes, de rectangles et de textes.

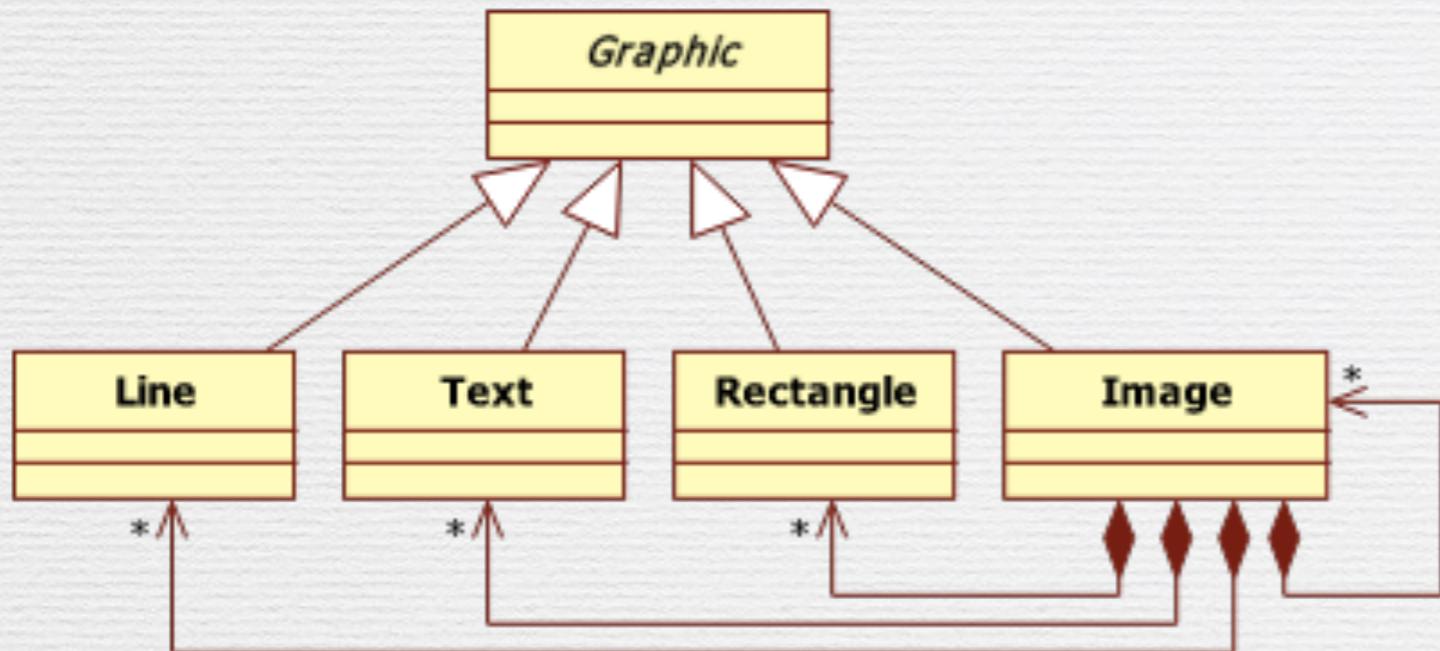
- 1. Découplage et extensibilité
- 1.1 Factorisation maximale de la composition
- 1.2 L'ajout ou la suppression d'une feuille n'implique pas de modification de code
- 1.3 L'ajout ou la suppression d'un composite ne n'implique pas de modification de code
- 2. Protocole uniforme
 - 2.1 Protocole uniforme sur les opérations des objets composés
 - 2.2 Protocole uniforme sur la gestion de la composition
 - 2.3 Point d'accès unique pour la classe client



Quels sont les points faibles de ce modèle?

Modéliser un système permettant de dessiner un graphique. Un graphique est composé de lignes, de rectangles, de textes et d'images, une image pouvant être composée d'autres images, de lignes, de rectangles et de textes.

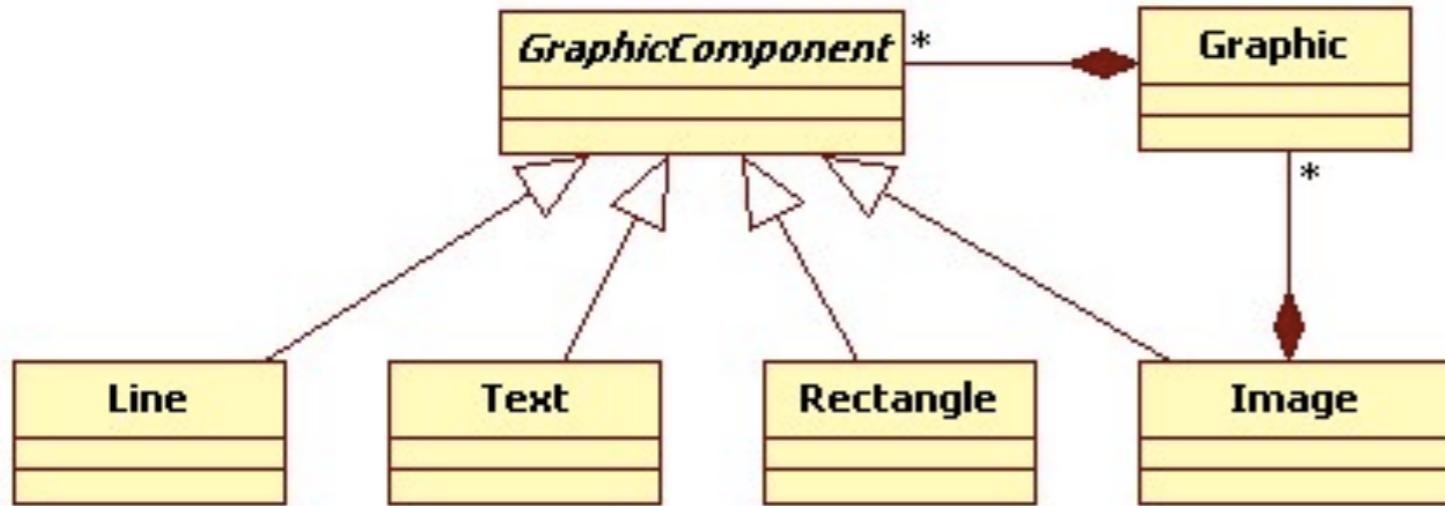
- 1. Découplage et extensibilité
- 1.1 Factorisation maximale de la composition
- 1.2 L'ajout ou la suppression d'une feuille n'implique pas de modification de code
- 1.3 L'ajout ou la suppression d'un composite ne n'implique pas de modification de code
- 2. Protocole uniforme
 - 2.1 Protocole uniforme sur les opérations des objets composés
 - 2.2 Protocole uniforme sur la gestion de la composition
 - 2.3 Point d'accès unique pour la classe client



Quels sont les points faibles de ce modèle?

Modéliser un système permettant de dessiner un graphique. Un graphique est composé de lignes, de rectangles, de textes et d'images, une image pouvant être composée d'autres images, de lignes, de rectangles et de textes.

- 1. Découplage et extensibilité
- 1.1 Factorisation maximale de la composition
- 1.2 L'ajout ou la suppression d'une feuille n'implique pas de modification de code
- 1.3 L'ajout ou la suppression d'un composite ne n'implique pas de modification de code
- 2. Protocole uniforme
- 2.1 Protocole uniforme sur les opérations des objets composés
- 2.2 Protocole uniforme sur la gestion de la composition
- 2.3 Point d'accès unique pour la classe client



Quels sont les points faibles de ce modèle?

Modéliser un système permettant de dessiner un graphique. Un graphique est composé de lignes, de rectangles, de textes et d'images, une image pouvant être composée d'autres images, de lignes, de rectangles et de textes.

- 1. Découplage et extensibilité
- 1.1 Factorisation maximale de la composition
- 1.2 L'ajout ou la suppression d'une feuille n'implique pas de modification de code
- 1.3 L'ajout ou la suppression d'un composite ne n'implique pas de modification de code
- 2. Protocole uniforme
- 2.1 Protocole uniforme sur les opérations des objets composés
- 2.2 Protocole uniforme sur la gestion de la composition
- 2.3 Point d'accès unique pour la classe client

Design Pattern Adaptor

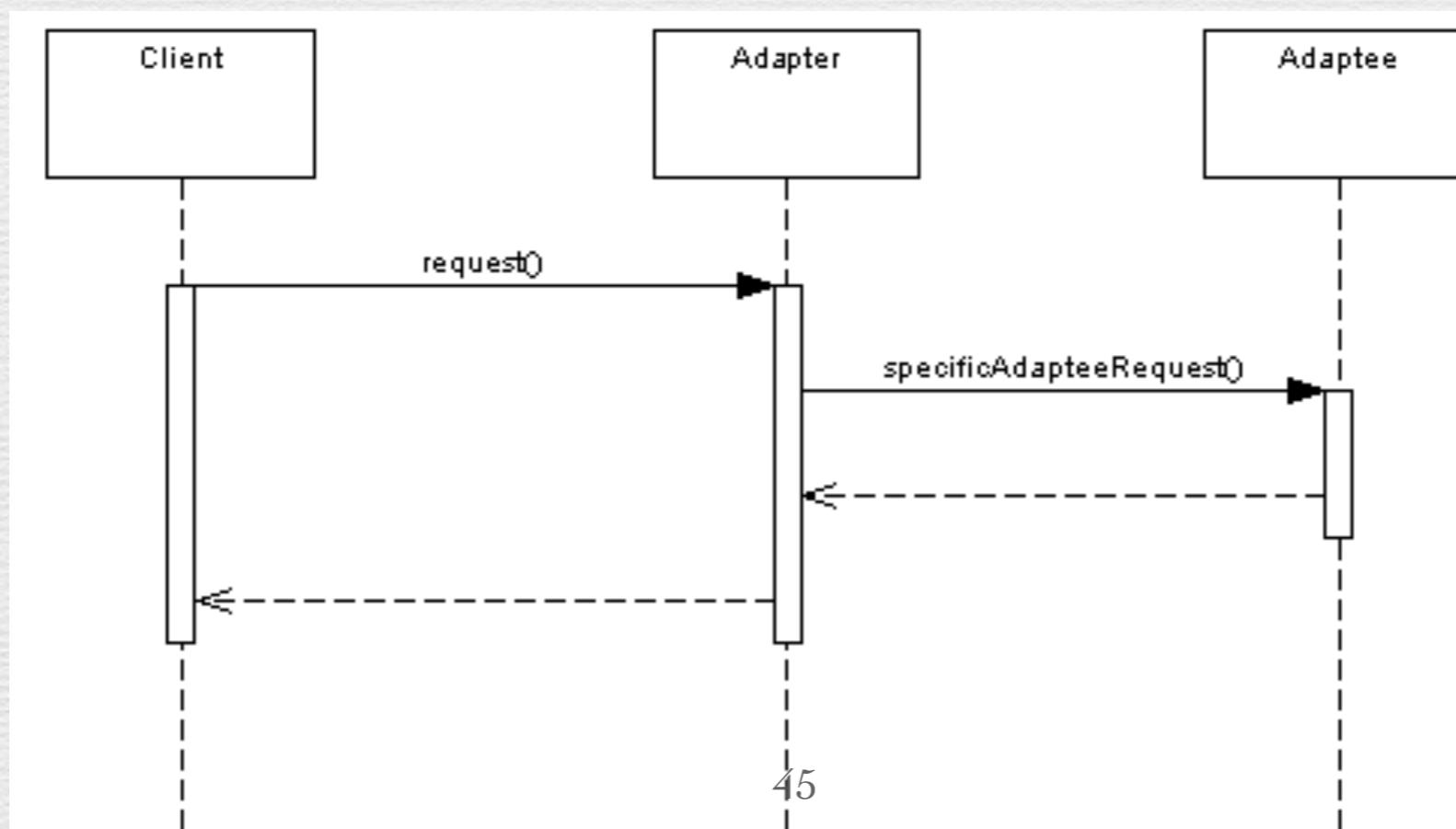
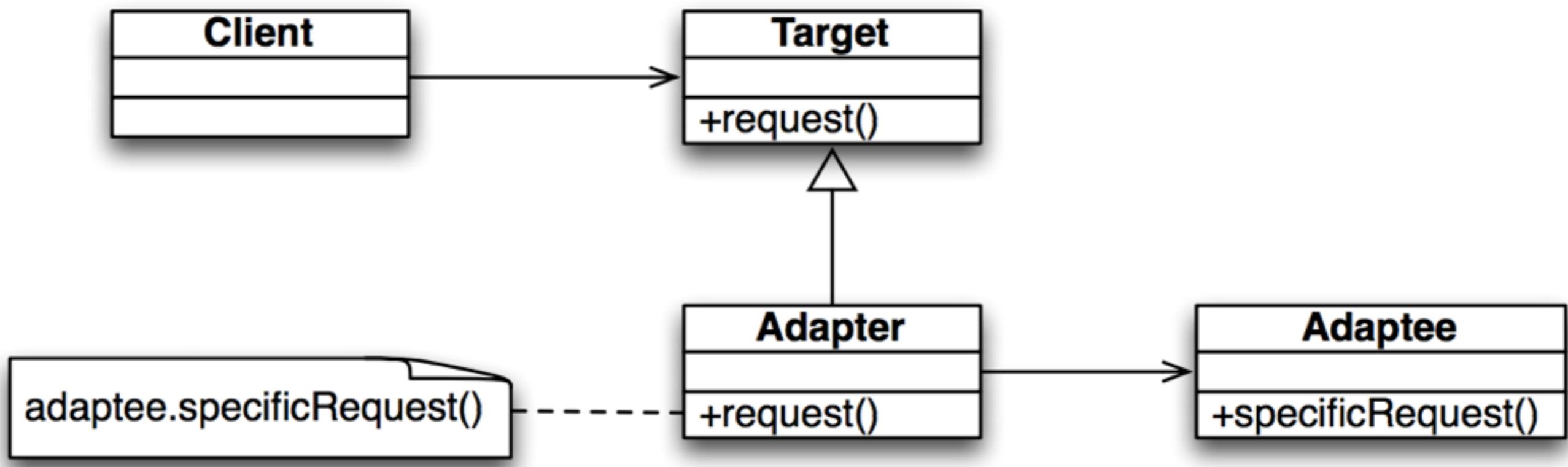
Patron Adapter : le problème

- ❖ “ 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.

Patron Adapter : solution



- ~ On a défini un jeu dans lequel des avatars peuvent avancer, être affecté à un joueur, ou se battre.
- ~ On voudrait pouvoir utiliser des objets présents dans un autre jeu.
- ~ On a déjà les codes suivants :

```
public class Robot {
    private String maitre;
    public void frapper() {...}
    public void marcher() {...}
    public void reagirAHumain(String maitre) {
        this.maitre = maitre; ...
    }
}

public interface Avatar {
    //Renvoie un nombre de Points
    int attaquer();
    void avancer();
    void affecterJoueur();
}
```

```
public class Tank {
    //Force du tir
    public int tirer() ... }

    public void rouler(int vitesse, Direction d) ...
}

public void setConducteur(String maitre) {
}
```

Design Pattern Decorator

[http://www.tutorialspoint.com/design pattern/
decorator pattern.htm](http://www.tutorialspoint.com/design_pattern/decorator_pattern.htm)

Traité à partir des codes

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.

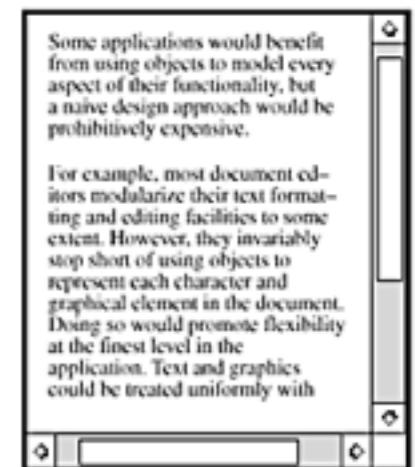
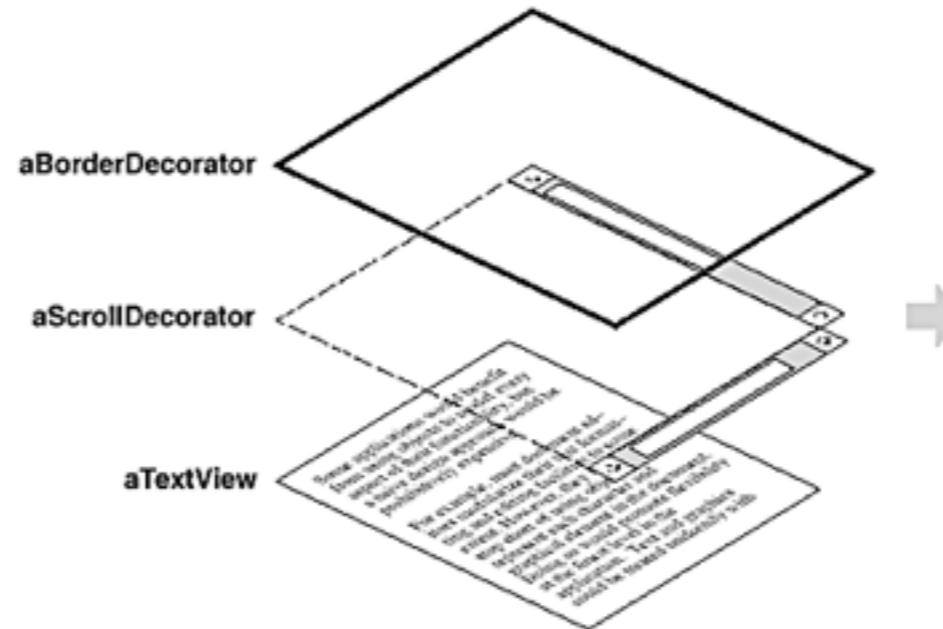
Motivation

■ A TextView has 2 features:

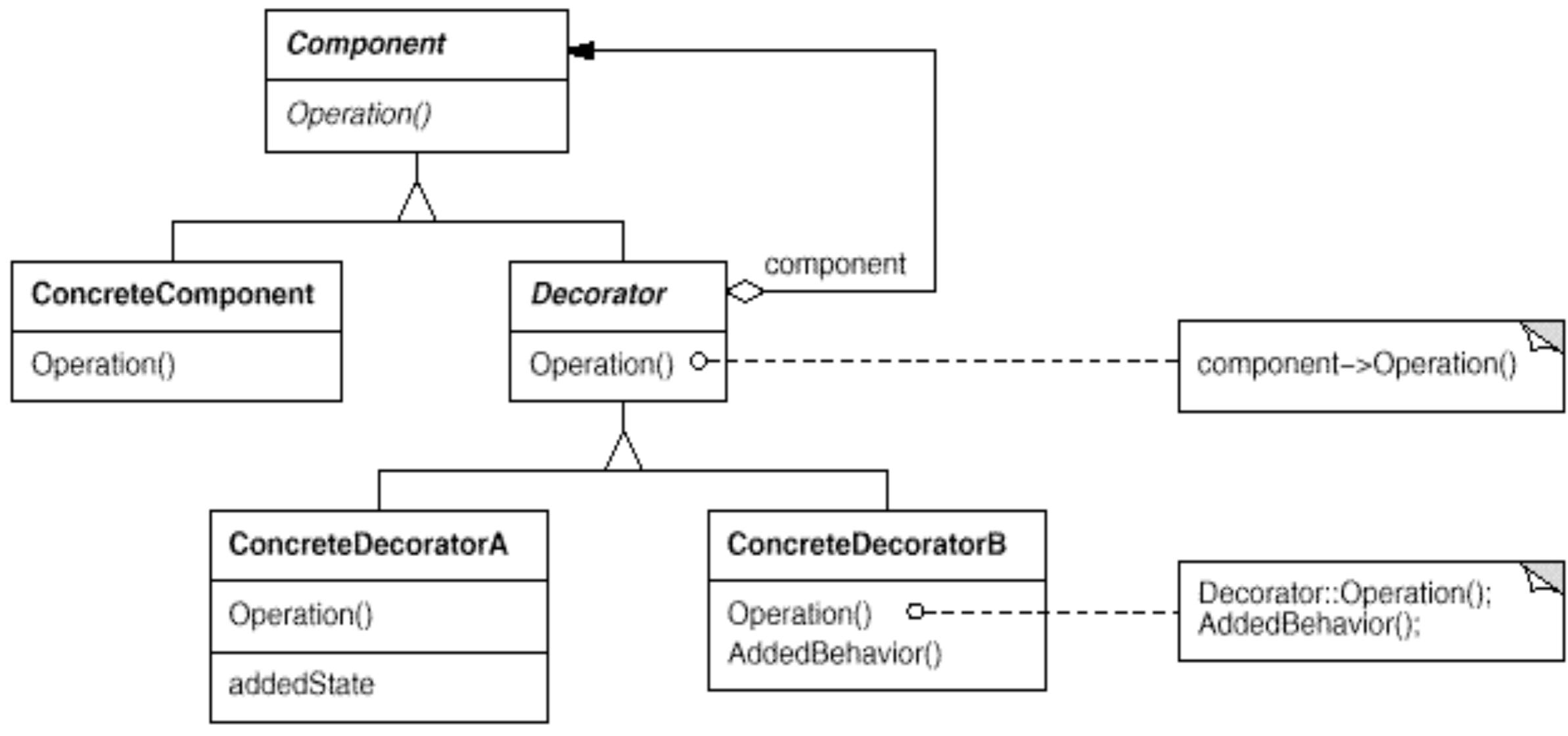
- borders:
 - 3 options: none, flat, 3D
- scroll-bars:
 - 4 options: none, side, bottom both

■ How many Classes?

- $3 \times 4 = 12 !!!$
 - 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 référence 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.

Simple Factory

Simple Factory Pattern : le problème par l'exemple

```
public class Jeu {  
  
    ArrayList<Personnage>personnages = new ArrayList<Personnage>();  
    public Jeu(){  
        Personnage principal = new Personnage();  
        personnages.add(principal);  
    }  
}
```

Simple Factory Pattern : le problème par l'exemple

```
public class Jeu {  
  
    ArrayList<Personnage>personnages = new ArrayList<Personnage>();  
    public Jeu(TypeJeu typeJeu) throws Exception{  
        Personnage principal;  
        if (typeJeu.equals(TypeJeu.guerre))  
            principal = new Monstre();  
        else  
            if (typeJeu.equals(TypeJeu.horreur))  
                principal = new Guerrier();  
            else throw new Exception("Jeu inconnu");  
        personnages.add(principal);  
    }  
}
```

Et si on ajoute ou retire des classes de Personnage ? ...

Simple Factory Pattern : Solution par l'exemple

Fabriquer un jeu c'est un peu plus compliqué... les personnages dans un jeu pour enfants sont différents d'un jeu de guerre...

Il nous faut une fabrique par type de jeu...

```

Personnage creerPersonnage(String s){
    Personnage principal;
    if (s.equals("Monstre"))
        principal = new Monstre();
    else
        if (s.equals("Guerrier"))
            principal = new Guerrier();
        else
            principal = new Improbable();
    return principal;
}

```

Simple Factory Pattern : Solution par l'exemple

```

Class<?> classe = Class.forName("dpSimpleFactory."+ s);
Personnage secondaire = (Personnage) classe.newInstance();

```

```

public class Jeu {

    ArrayList<Personnage>personnages = new ArrayList<Personnage>();

    FabriquePersonnages fabrique = new FabriquePersonnages();
    public Jeu(TypeJeu typeJeu) throws Exception{
        Personnage principal;
        if (typeJeu.equals(TypeJeu.guerre))
            principal = fabrique.creerPersonnage("Guerrier");
        else
            if (typeJeu.equals(TypeJeu.horreur))
                principal = fabrique.creerPersonnage("Monstre");
            else throw new Exception("Jeu inconnu");
        personnages.add(principal);
    }
}

```

Autre exemple

```
1 public class PizzaStore {  
2  
3     Pizza orderPizza(String type) {  
4  
5         Pizza pizza;  
6  
7         if (type.equals("cheese")) {  
8             pizza = new CheesePizza();  
9         } else if (type.equals("greek")) {  
10            pizza = new GreekPizza();  
11        } else if (type.equals("pepperoni")) {  
12            pizza = new PepperoniPizza();  
13        }  
14  
15        pizza.prepare();  
16        pizza.bake();  
17        pizza.cut();  
18        pizza.box();  
19  
20        return pizza;  
21    }  
22 }  
23  
24 Note: excellent example of "coding to an interface"
```

Creation

Creation code has all the same problems as the code earlier

Preparation

```
class Pizza {  
    String name;  
  
    public String getName() {  
        return name; }  
  
    public void prepare() {  
    }  
  
    public void box() {  
    }  
  
    public void cut() {  
    }  
  
    public void bake() {  
    }  
}  
  
class CheesePizza extends Pizza {}  
class GreekPizza extends Pizza {}  
class PepperoniPizza extends Pizza {}
```

```
package dpFactory;  
  
public class PizzaStore {  
  
    public Pizza orderPizza (String type) {  
        Pizza pizza = null;  
  
        if (type.equals("cheese")) {  
            pizza = new CheesePizza();  
        }  
        if (type.equals("greek")) {  
            pizza = new GreekPizza();  
        }  
        if (type.equals("pepperoni")) {  
            pizza = new PepperoniPizza();  
        }  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
  
        return pizza;  
    }  
}
```

```
package dpFactory;

public class PizzaStore {
    PizzaFactory factory;

    public PizzaStore(PizzaFactory factory) {
        this.factory = factory;
    }

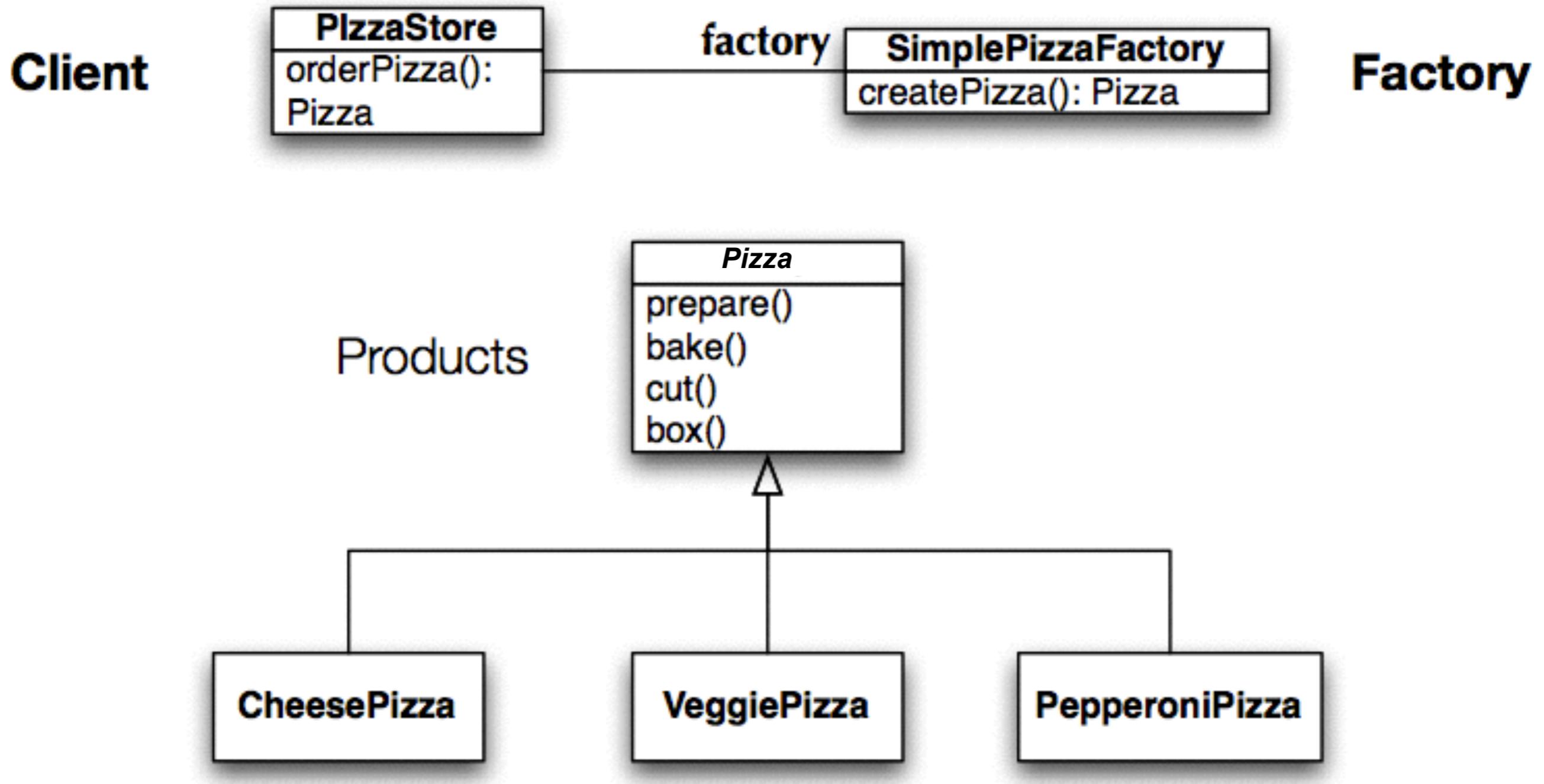
    public Pizza orderPizza(String type) {
        Pizza pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        return pizza;
    }
}

public interface PizzaFactory {
    Pizza createPizza(String type);
}
```

```
public class SimplePizzaFactory implements PizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if("cheese".equals(type)) {
            pizza = new CheesePizza();
        } else if("onion".equals(type)) {
            pizza = new OnionPizza();
        }
        return pizza;
    }
}
```



Class Diagram of New Solution



While this is nice, it is not as flexible as it can be: to increase flexibility we need to look at two design patterns: Factory Method and Abstract Factory



Des franchises de PizzaStore

```
1 public class NYStylePepperoniPizza extends Pizza {  
2  
3     public NYStylePepperoniPizza() {  
4         name = "NY Style Pepperoni Pizza";  
5         dough = "Thin Crust Dough";  
6         sauce = "Marinara Sauce";  
7  
8         toppings.add("Grated Reggiano Cheese");  
9         toppings.add("Sliced Pepperoni");  
10        toppings.add("Garlic");  
11        toppings.add("Onion");  
12        toppings.add("Mushrooms");  
13        toppings.add("Red Pepper");  
14    }  
15 }  
16  
17 }
```

```
1  
2 public class ChicagoStylePepperoniPizza extends Pizza {  
3  
4     public ChicagoStylePepperoniPizza() {  
5         name = "Chicago Style Pepperoni Pizza";  
6         dough = "Extra Thick Crust Dough";  
7         sauce = "Plum Tomato Sauce";  
8  
9         toppings.add("Shredded Mozzarella Cheese");  
10        toppings.add("Black Olives");  
11        toppings.add("Spinach");  
12        toppings.add("Eggplant");  
13        toppings.add("Sliced Pepperoni");  
14    }  
15  
16    void cut() {  
17        System.out.println("Cutting the pizza into square slices");  
18    }  
19 }
```

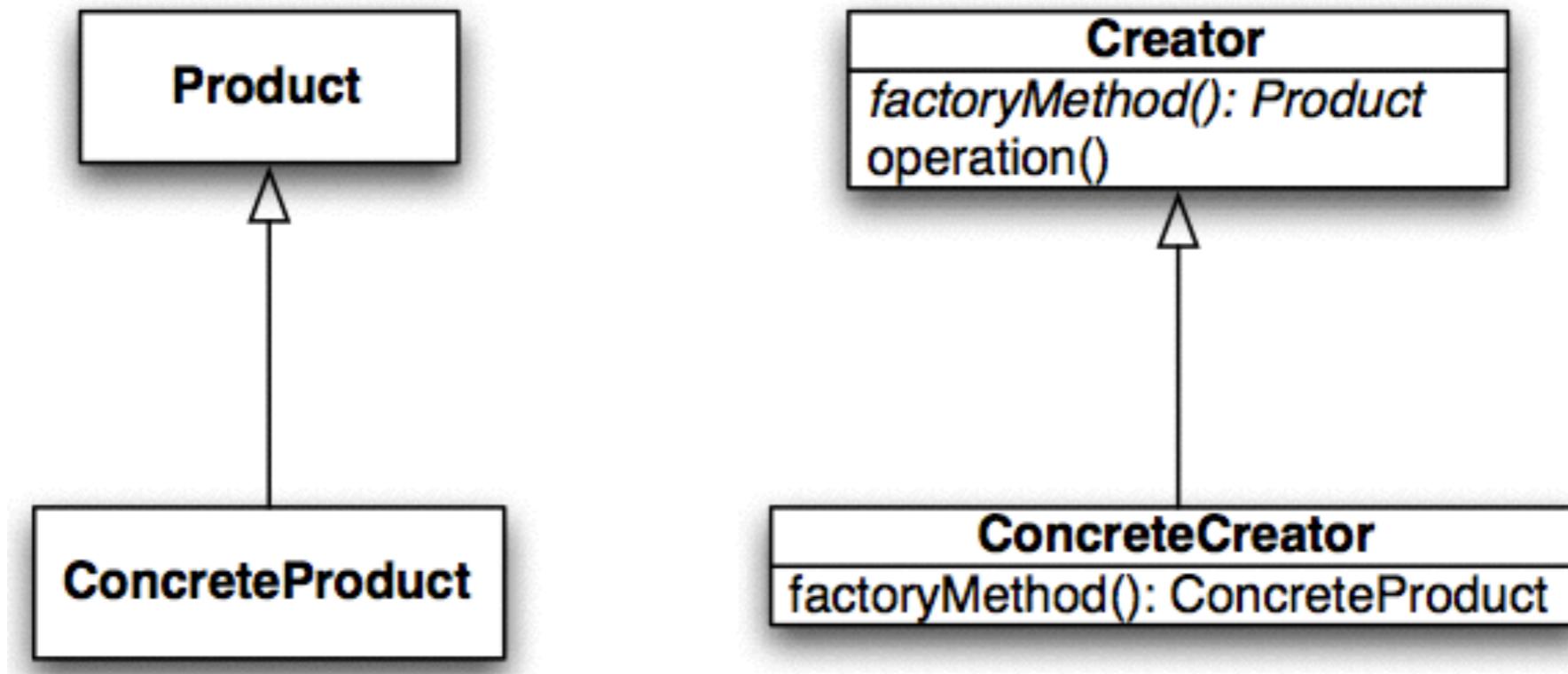
A couple of the concrete product classes

```
public class NewYorkPizzaFactory implements PizzaFactory {  
  
    public Pizza createPizza(String type) {  
        Pizza pizza = null;  
        if (type.equalsIgnoreCase("cheese")) {  
            pizza = new NYStyleCheesePizza();  
        } else if (type.equalsIgnoreCase("veggie")) {  
            pizza = new NYStyleVeggiePizza();  
        }  
        return pizza;  
    }  
}
```



Factory Method: Definition and Structure

- The Factory Method design pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Factory Method leads to the creation of parallel class hierarchies; ConcreteCreators produce instances of ConcreteProducts that are operated on by Creators via the Product interface



Moving On

- The factory method approach to the pizza store is a big success, allowing our company to create multiple franchises across the country quickly and easily
 - But (bad news): we have learned that some of the franchises
 - while following our procedures (the abstract code in PizzaStore forces them to)
 - are skimping on ingredients in order to lower costs and increase margins
 - Our company's success has always been dependent on the use of fresh, quality ingredients
 - So, something must be done!

```
public interface Dough {};  
  
public interface Cheese {};  
  
public interface Pepperoni {};  
  
  
public class ThinCrustDough implements Dough {  
    public String toString() { return "This Crust Dough"; }  
}  
  
  
public class ParmesanCheese implements Cheese {  
    public String toString() { return "Shredded Parmesan"; }  
}  
  
  
public class SlicedPepperoni implements Pepperoni {  
    public String toString() { return "Sliced Pepperoni"; }  
}
```

```
// abstract ingredient factory

public interface PizzaIngredientFactory {
    public Dough createDough();
    public Cheese createCheese();
    public Pepperoni createPepperoni();
}

public class NYPizzaIngredientFactory implements PizzaIngredientFactory {
    public Dough createDough() { return new ThinCrustDough(); }
    public Sauce createCheese() { return new ParmesanCheese(); }
    public Pepperoni createPepperoni() { return new SlicedPepperoni(); }
}
```

```
public class NYStyleCheesePizza extends Pizza {  
  
    Pizza pizza = null;  
    PizzaIngredientFactory ingredientFactory = new  
    NYPizzaIngredientFactory();  
  
    if (item.equalsIgnoreCase("cheese")) {  
        pizza = new CheesePizza(ingredientFactory);  
        pizza.setName("New York Style Cheese Pizza");  
    } else if (item.equalsIgnoreCase("pepperoni")) {  
        pizza = new PepperoniPizza(ingredientFactory);  
        pizza.setName("New York Style Pepperoni Pizza");  
    }  
    return pizza;  
}
```



Within Pizza Subclasses... (I)

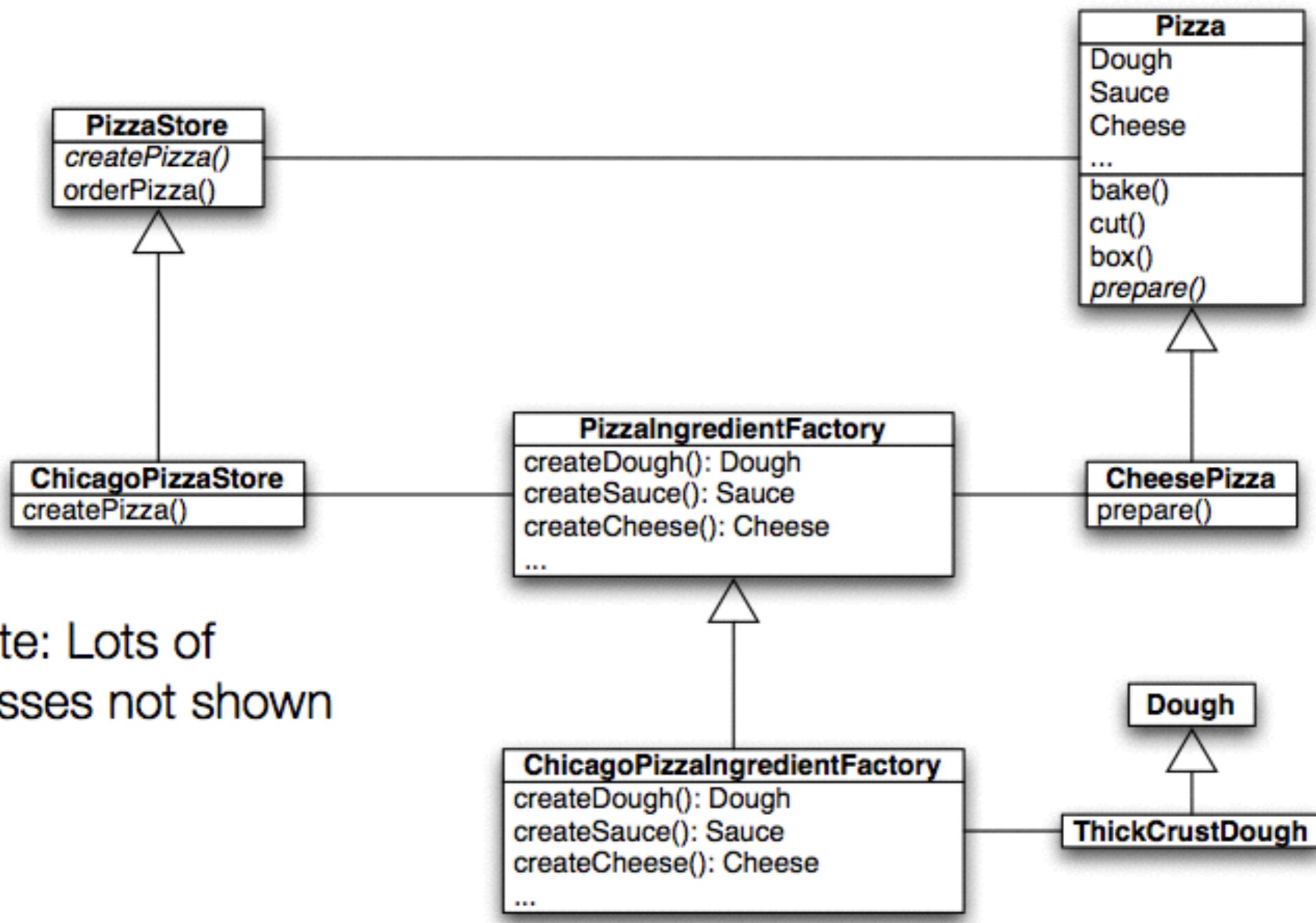
```
2 public abstract class Pizza {  
3     String name;  
4  
5     Dough dough;  
6     Sauce sauce;  
7     Veggie veggies[];  
8     Cheese cheese;  
9     Pepperoni pepperoni;  
.0     Clams clam;  
.1  
.2     abstract void prepare();  
.3  
.4     void bake() {  
.5         System.out.println("Bake for 25 minutes at 350");  
.6     }  
.7  
.8     void cut() {
```

First, alter the Pizza abstract base class to make the prepare method abstract...

```
public class CheesePizza extends Pizza {  
  
    PizzaIngredientsFactory ingredientFactory;  
  
    public CheesePizza(PizzaIngredientsFactory ingredientFactory){  
        this.ingredientFactory = ingredientFactory;  
    }  
  
    @Override  
    public void prepare() {  
  
        System.out.println("Preparing " + name);  
  
        dough = ingredientFactory.createDough();  
  
        sauce = ingredientFactory.createSauce();  
  
        cheese= ingredientFactory.createCheese();  
    }  
}
```



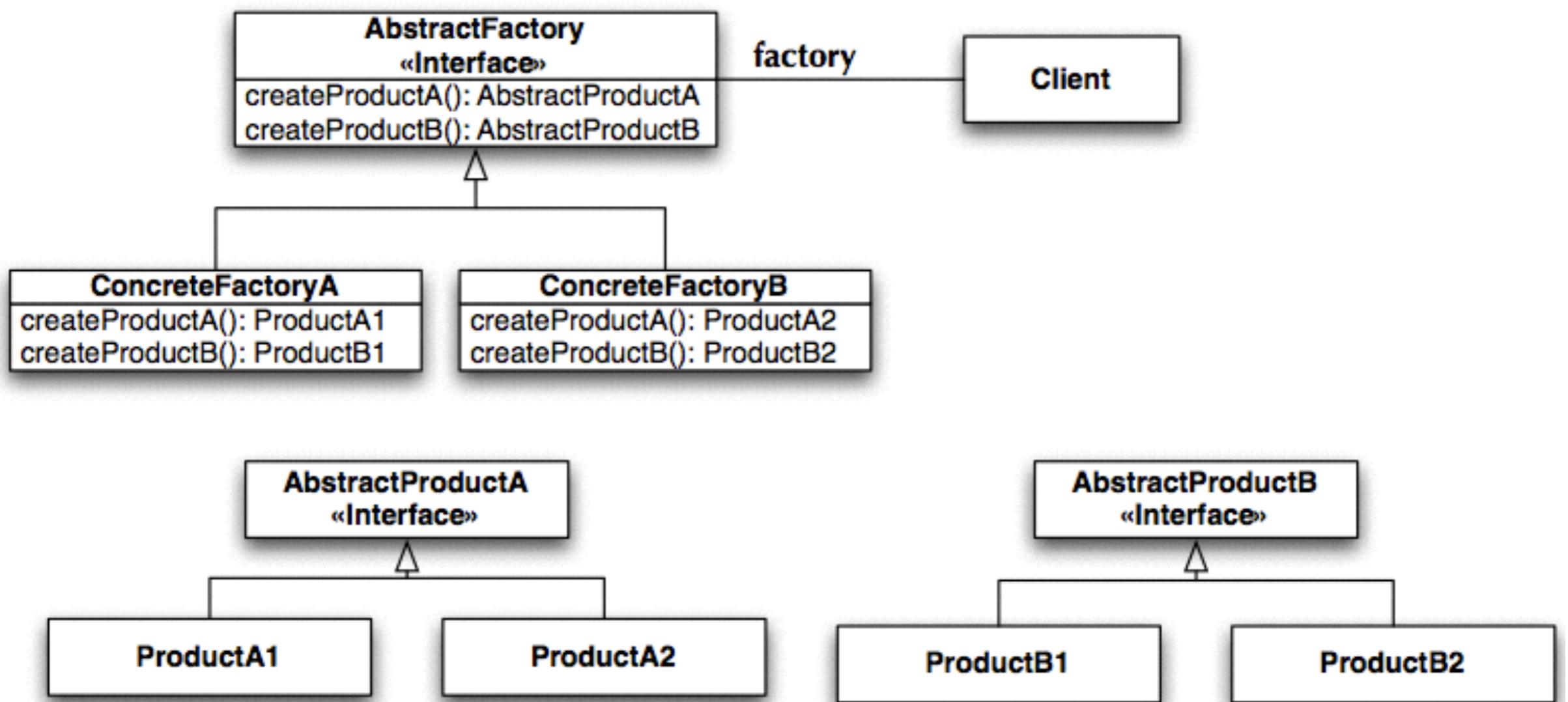
Class Diagram of Abstract Factory Solution





Abstract Factory: Definition and Structure

- The Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes





Abstract Factory Characteristics

- Isolates concrete classes
- Makes exchanging product families easy
- Promotes consistency among products

Design Pattern

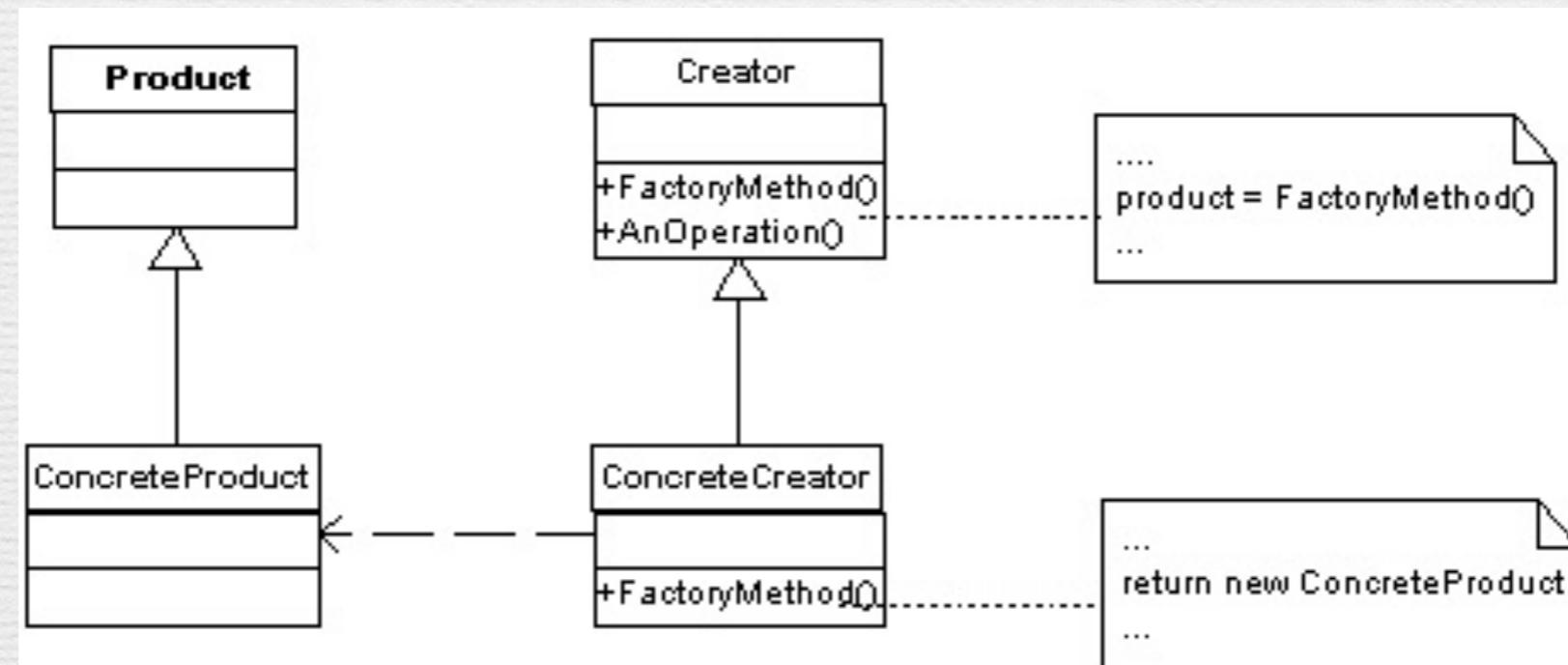
«Factory Method»

Factory Method Pattern : le problème

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

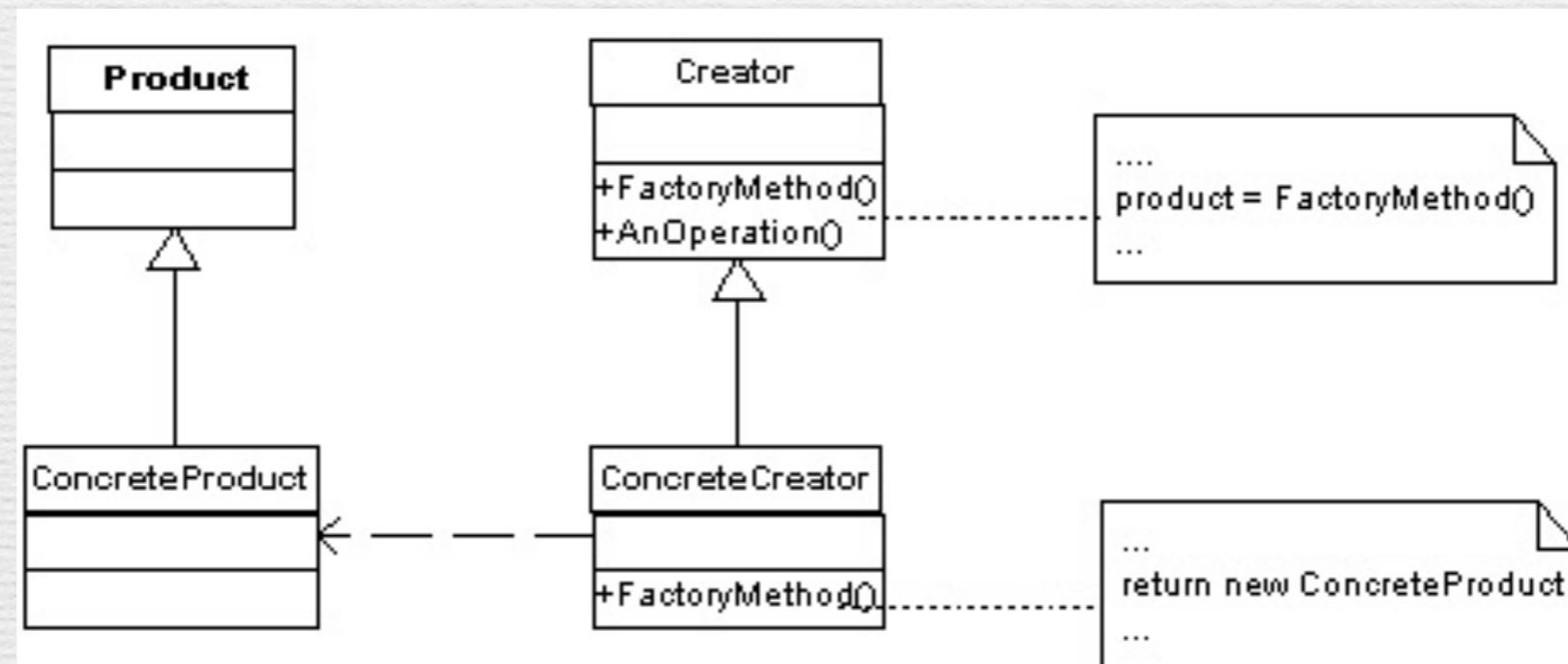
Factory Pattern : la solution

- ✓ 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

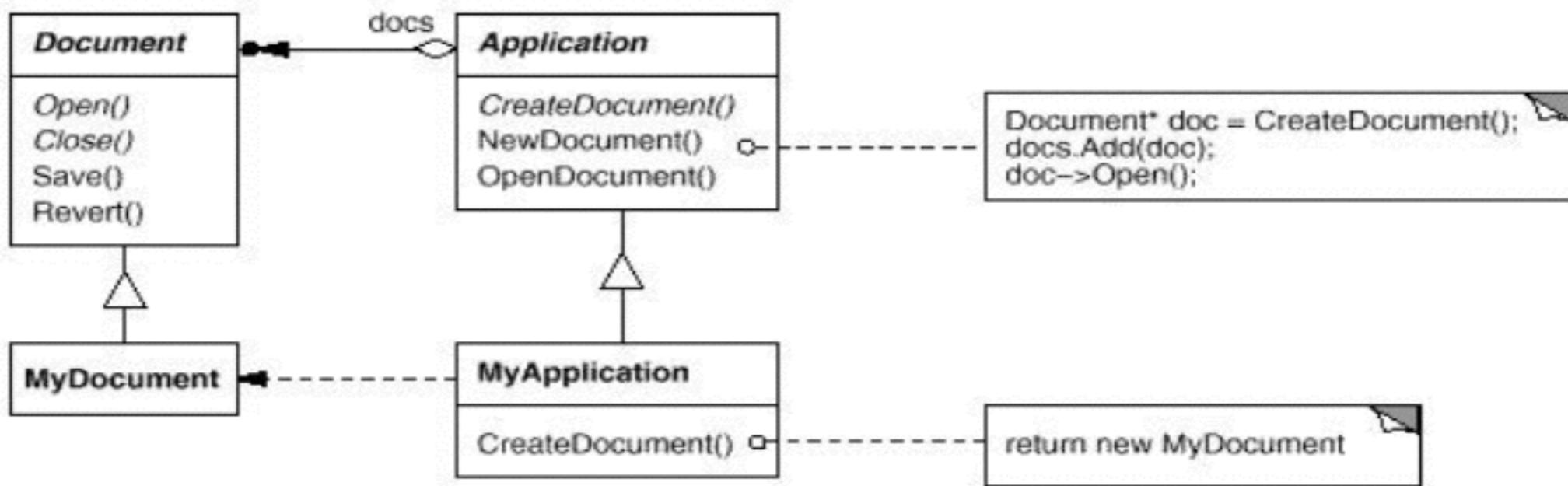


Factory Pattern : les rôles

- ✓ «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

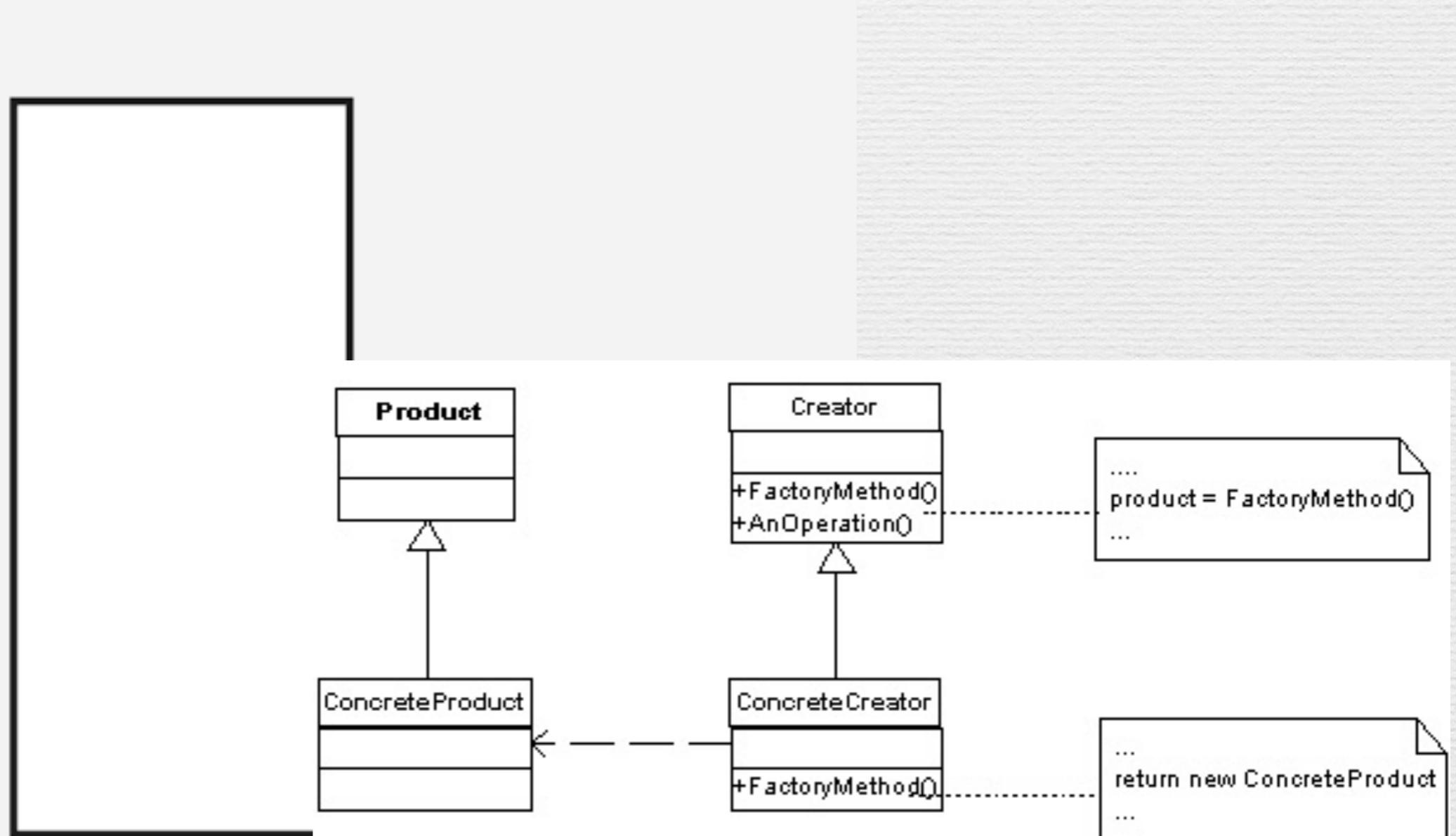


Quelle est la méthode «factory» ?

<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.



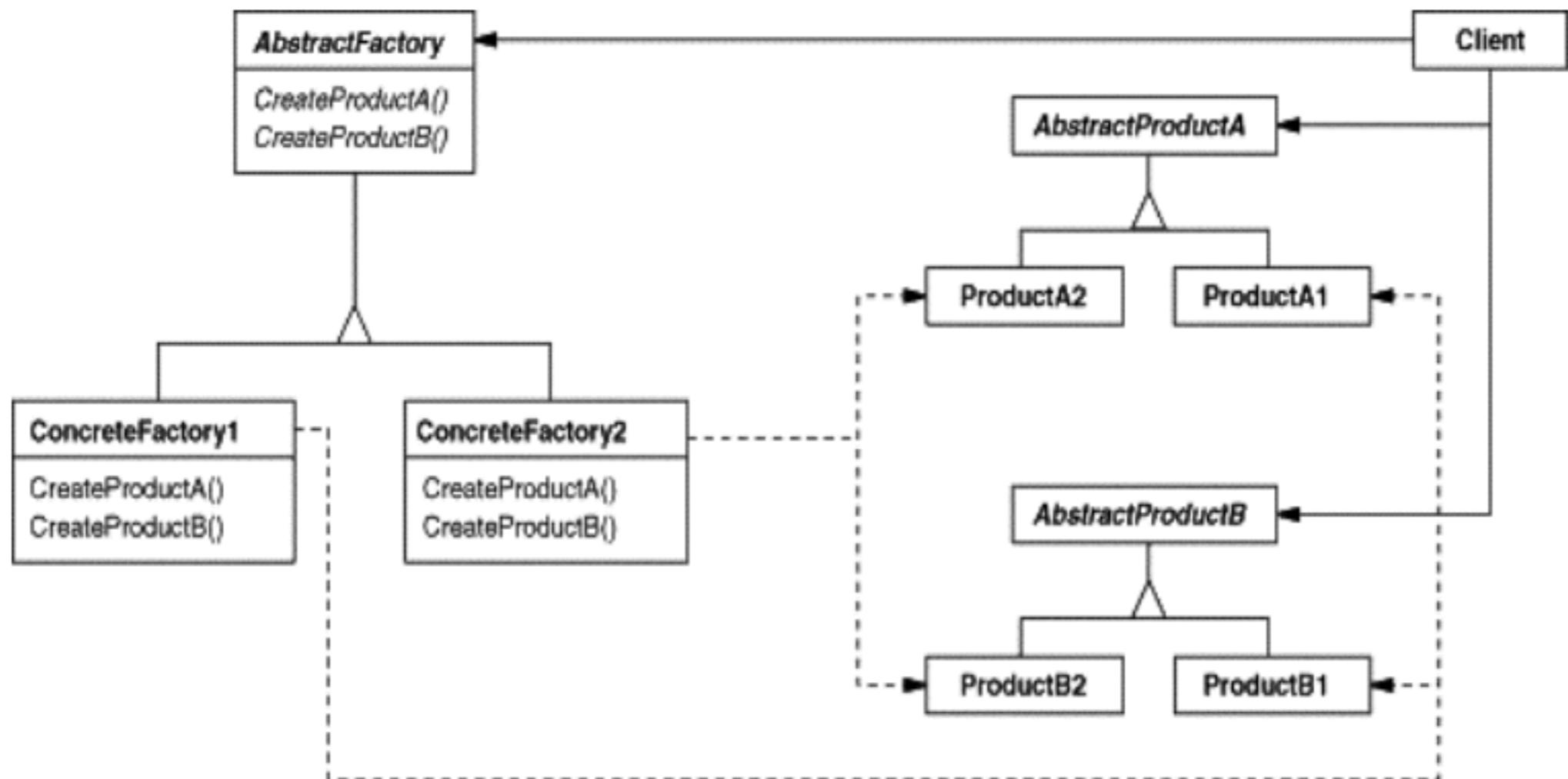
Design Pattern

«Abstract Factory»

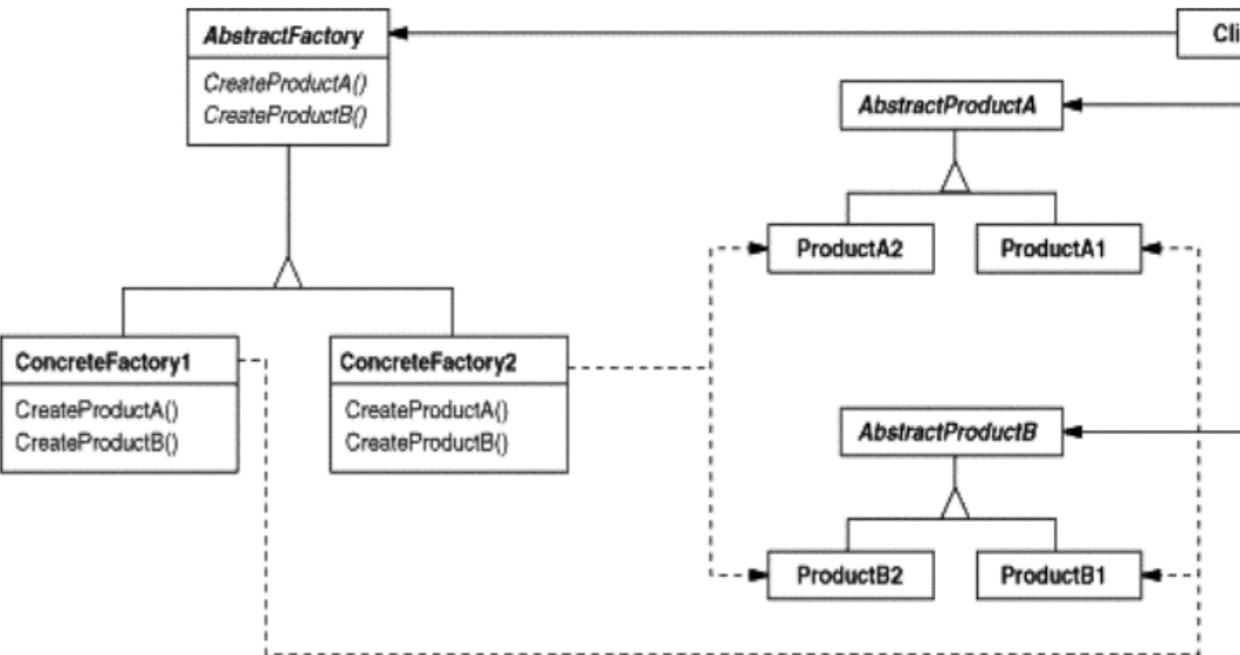
Abstract Factory Pattern : le problème

- ✓ Une application doit créer des familles d'objets dépendants sans spécifier leurs classes concrètes.

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

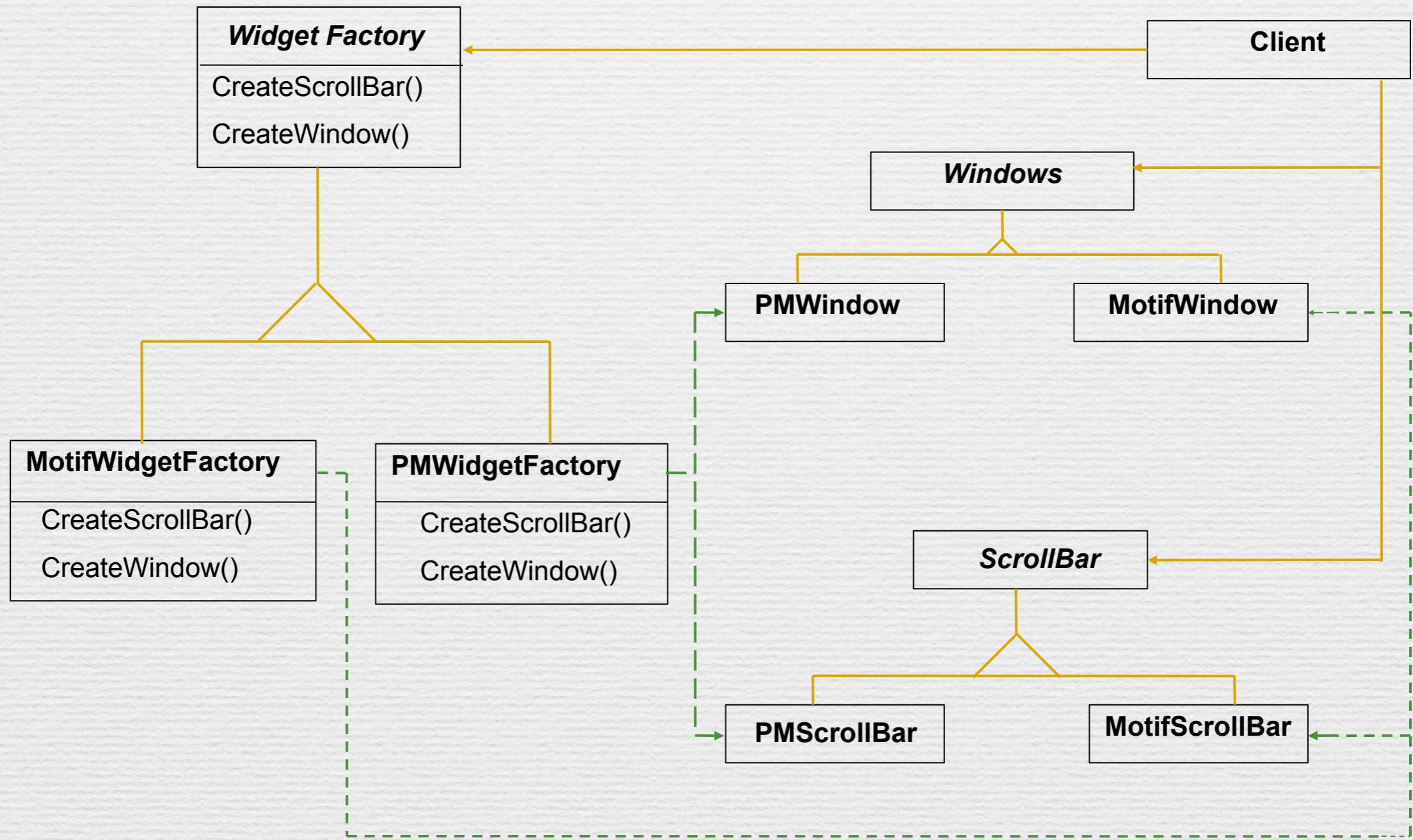
✓ ConcreteProduct

- Defines a product object to be created by the corresponding concrete factory
- Implements the **AbstractProduct** interface

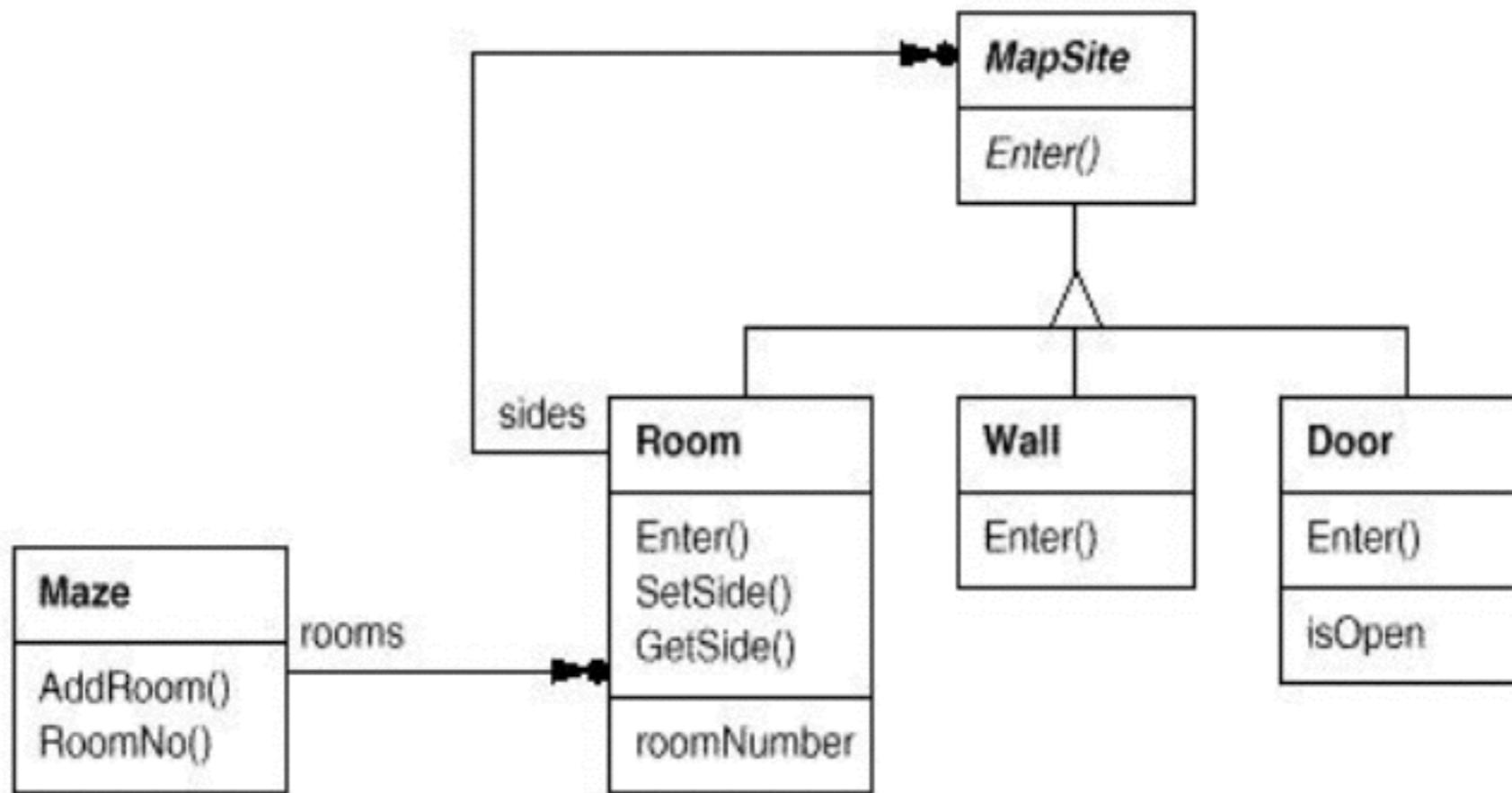
✓ Client

- Uses only interfaces declared by **AbstractFactory** and **AbstractProduct** classes

Abstract Factory Pattern : exemple



* Factory Pattern : exemple



```
/***
 * 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 : 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;
    }
}
```

Labyrinthe
... magique
avec des
pièces
enchantées??

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);
    }
}
```

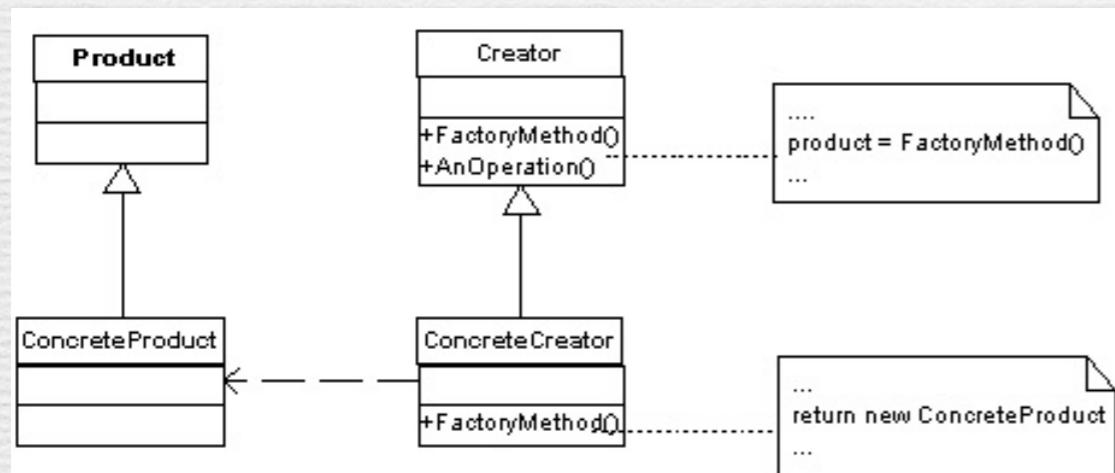
```
public Maze createMaze() {  
    Maze maze = makeMaze();  
    Room r1 = makeRoom(1);  
    Room r2 = makeRoom(2);  
    Door door = makeDoor(r1, r2);  
    maze.addRoom(r1);  
    maze.addRoom(r2);  
    r1.setSide(MazeGame.North, makeWall());  
    r1.setSide(MazeGame.East, door);  
    r1.setSide(MazeGame.South,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

Factory Methods pour un Labyrinthe & usage

```

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);
    }
}

```



Retour sur le labyrinthe enchanté

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

```
// 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.

```
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);  
....
```

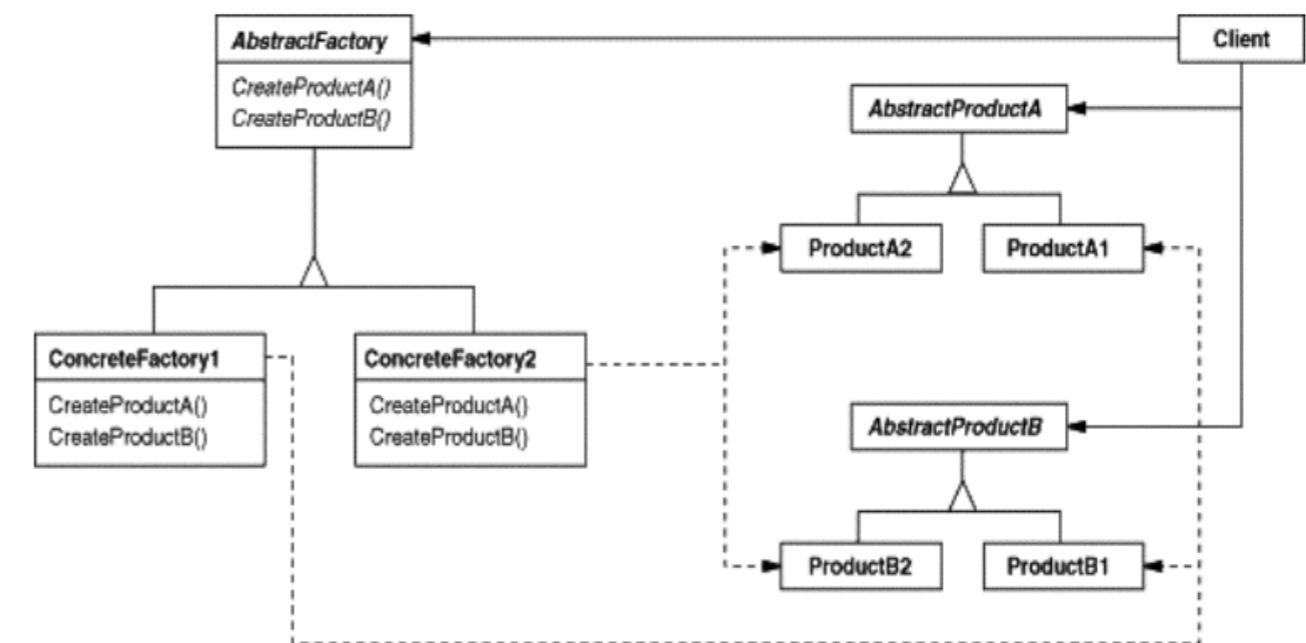
Pour créer
des familles
de
labyrinthes

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

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**

Vous avez le contrôle, ne soyez ni «fashion», ni «archaïque»

- (1) toutes les décisions en matière de conception, qu'elles relèvent de la conception logicielle ou architecturale, doivent être prises à la lumière des contraintes fonctionnelles, comportementales et sociales, et non selon des tendances aléatoires ;
- (2) lorsque vous ne maîtrisez qu'une façon de faire, tout à tendance à se ressembler.



« Il est tentant, si le seul outil que vous avez est un marteau, de traiter tout problème comme si c'était un clou. »

—Abraham Maslow, « The Psychology of Science »