



ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT
TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

المدرسة العليا لأساتذة التعليم التقني المحمدية

Design Patterns

2^{ème} année Cycle Ingénieur

GLSID 2, ICCN 2 & IIBDCC 2

Pr. SARA RETAL



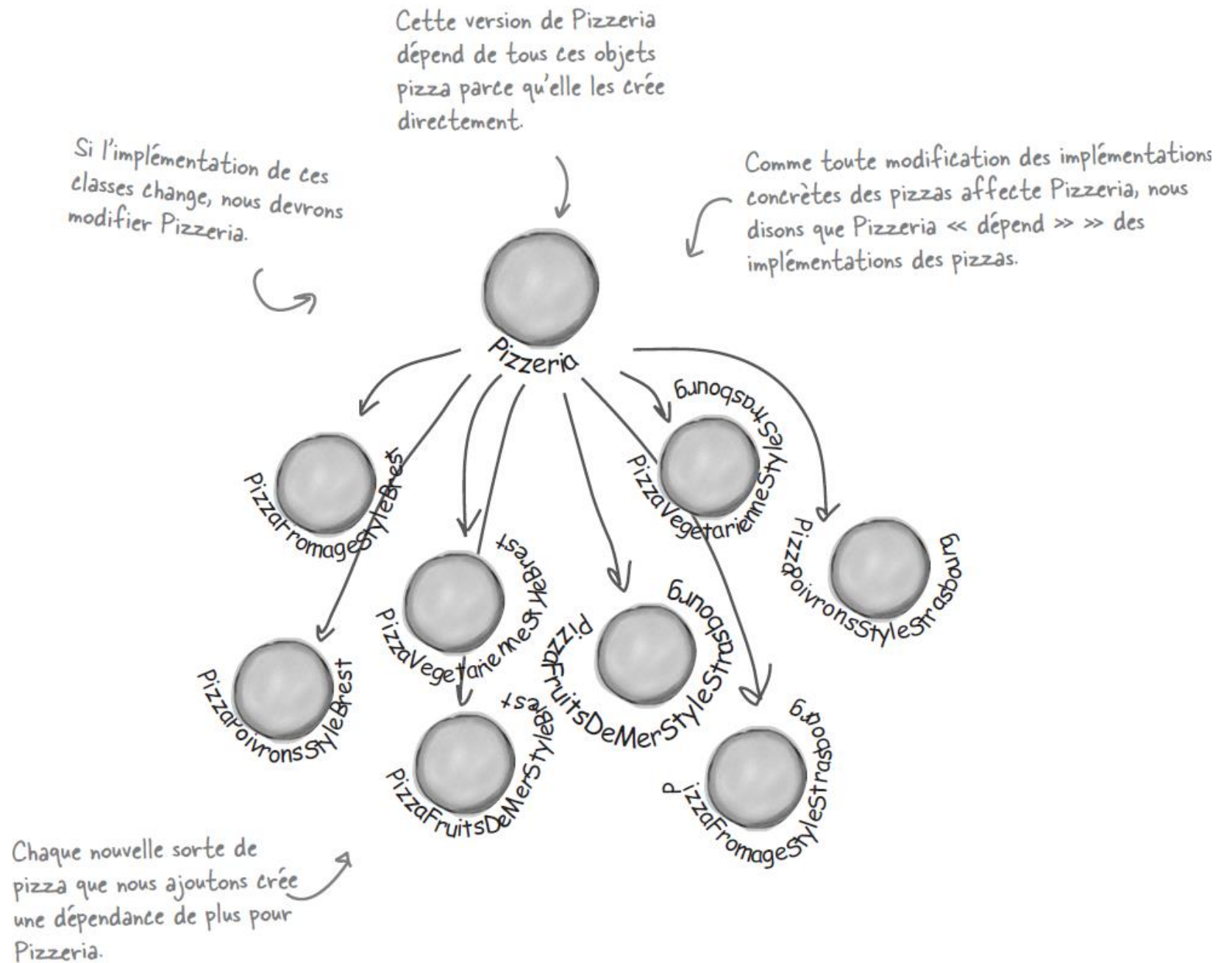
Problèmes de dépendance des objets

```
public class PizzeriaDependante {  
  
    public Pizza creerPizza(String style, String type) {  
        Pizza pizza = null;  
        if (style.equals("Brest")) {  
            if (type.equals("fromage")) {  
                pizza = new PizzaFromageStyleBrest();  
            } else if (type.equals("vegetarienne")) {  
                pizza = new PizzaVegetarienneStyleBrest();  
            } else if (type.equals("fruitsDeMer")) {  
                pizza = new PizzaFruitsDeMerStyleBrest();  
            } else if (type.equals("poivrons")) {  
                pizza = new PizzaPoivronsStyleBrest();  
            }  
        } else if (style.equals("Strasbourg")) {  
            if (type.equals("fromage")) {  
                pizza = new PizzaFromageStyleStrasbourg();  
            } else if (type.equals("vegetarienne")) {  
                pizza = new PizzaVegetarienneStyleStrasbourg();  
            } else if (type.equals("fruitsDeMer")) {  
                pizza = new PizzaFruitsDeMerStyleStrasbourg();  
            } else if (type.equals("poivrons")) {  
                pizza = new PizzaPoivronsStyleStrasbourg();  
            }  
        } else  
        {  
            System.out.println("Erreur : type de pizza invalide");  
            return null;  
        }  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.emballer();  
        return pizza;  
    }  
}
```

Gère toutes les pizzas
de style Brest

Gère toutes les
pizzas de style
Strasbourg

Problèmes de dépendance des objets

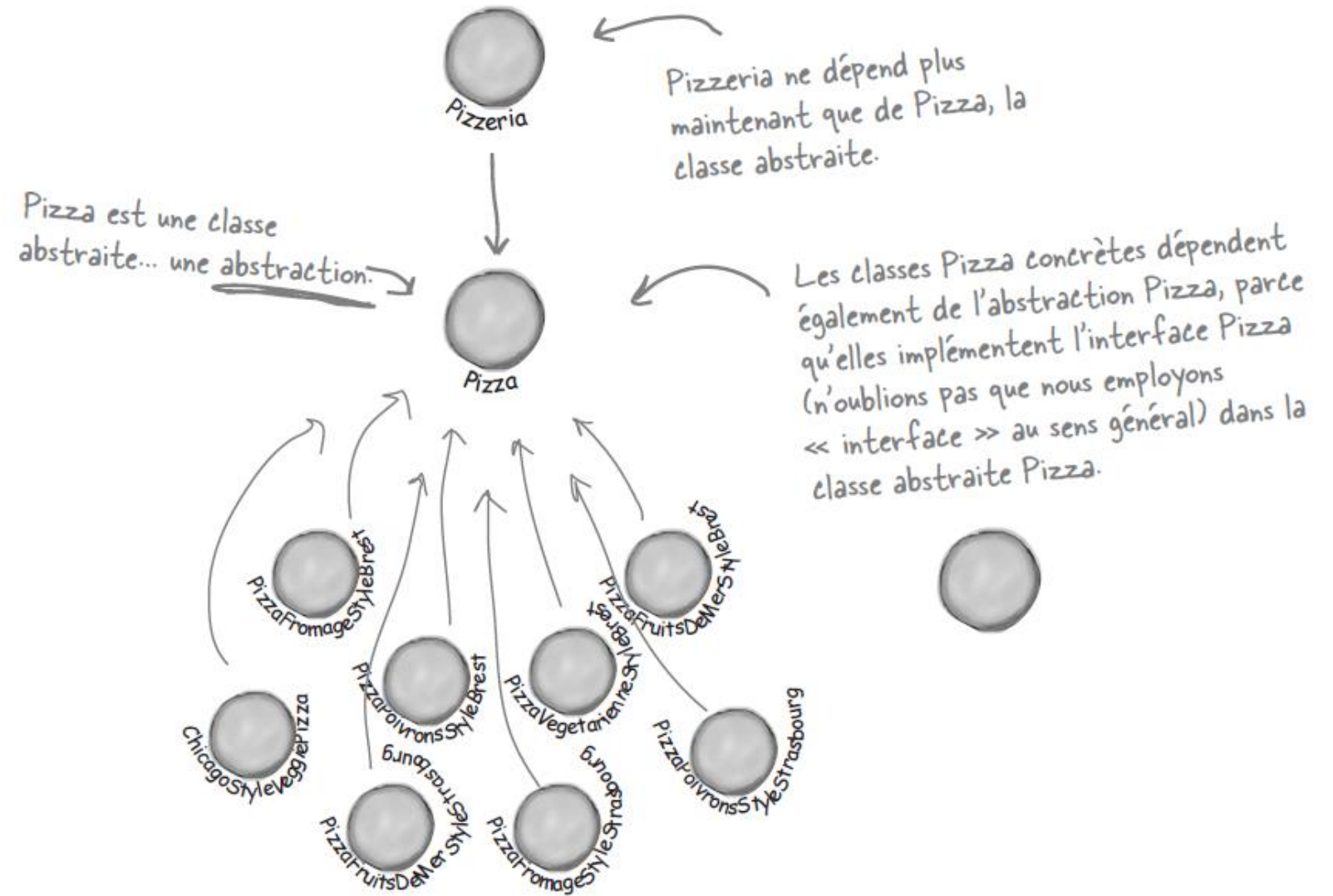


Principe de conception

Le principe d'inversion des dépendances :

- *Dépendez d'abstractions. Ne dépendez pas de classes concrètes*

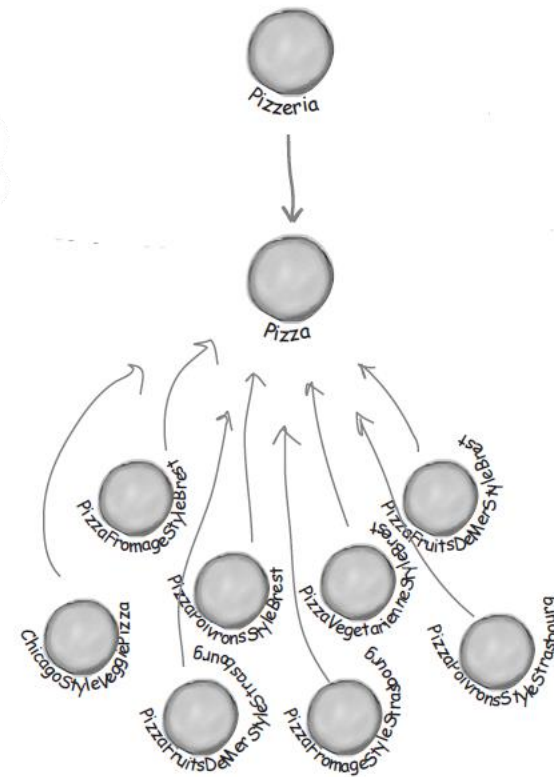
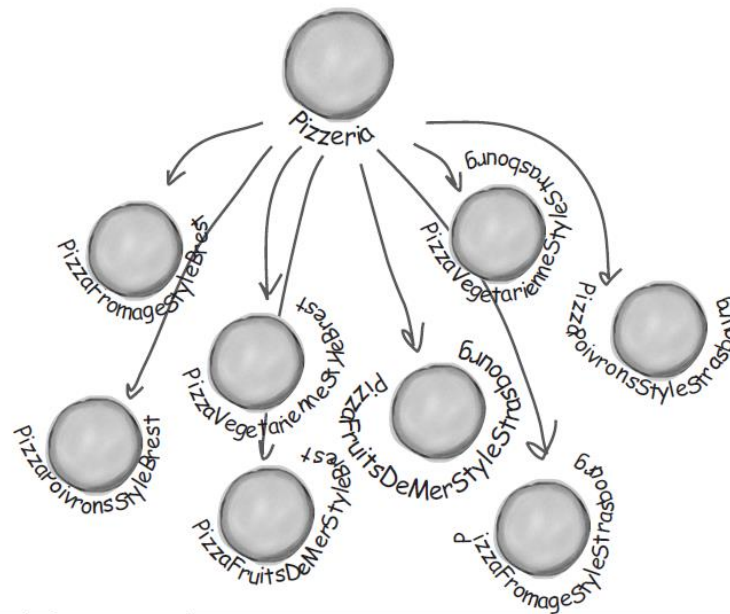
Appliquer le principe



Question

- Où est l' « inversion » dans le Principe d'inversion des dépendances ?

Question



Pizzeria

- Comment allez-vous donc veiller à ce que chaque franchise utilise des garnitures de qualité ? Vous allez construire une fabrique qui les produit et qui les leur expédie !
- Mais ce plan pose un problème : les franchises sont situées dans différentes régions et ce qui est une sauce tomate à Brest n'est pas une sauce tomate à Strasbourg.



Nous avons les
mêmes familles
de produits
(pâte, sauce,
fromage, légumes,
etc...) mais des
implémentations
différentes selon
la région.



Construire les fabriques d'ingrédients

```
public interface FabriqueIngredientsPizza {  
  
    public Pate creerPate ();  
    public Sauce creerSauce();  
    public Fromage creerFromage();  
    public Legumes[] creerLegumes();  
    public Poivrons creerPoivrons();  
    public Moules creerMoules();  
}
```

Pour chaque ingrédient, nous définissons une méthode de création dans notre interface.

Plein de nouvelles classes ici, une par ingrédient.

Si nous avons un « mécanisme » commun à implémenter dans chaque instance de fabrication, nous aurions pu créer à la place une classe abstraite...

Pizzeria

Voici ce que nous allons faire :

- ❶ Construire une fabrique pour chaque région. Pour ce faire, vous allez créer une sous-classe de `FabriqueIngredientsPizza` qui implémente chaque méthode de création.
- ❷ Implémenter un ensemble de classes ingrédients qui seront utilisées avec la fabrique, comme `Reggiano`, `PoivronsRouges` et `PateSoufflee`. Ces classes peuvent être partagées entre les régions en fonction des besoins.
- ❸ Puis nous assemblerons tout en incorporant nos fabriques dans l'ancien code de `Pizzeria`.

Pizzeria

```
public class FabriqueIngredientsPizzaBrest implements FabriqueIngredient-  
sPizza {
```

```
    public Pate creerPate() {  
        return new PateFine();  
    }
```

← Pour chaque ingrédient de la
famille, nous créons la version
Brest.

```
    public Sauce creerSauce() {  
        return new SauceMarinara();  
    }
```

```
    public Fromage creerFromage() {  
        return new Reggiano();  
    }
```

```
    public Legume[] creerLegumes() {  
        Legume legume [] = { new Ail(), new Oignon(), new Champignon(), new PoivronRouge() };  
        return legume;  
    }
```

```
    public Poivrons creerPoivrons() {  
        return new PoivronsEnRondelles();  
    }
```

↑ Pour les légumes, nous retournons un
tableau de Légumes. Ici, nous avons
codé les légumes en dur. Nous aurions
pu trouver quelque chose de plus
élaboré, mais comme cela n'ajouterait
rien à l'explication du pattern, nous
simplifions.

```
    public Moules creerMoules() {  
        return new MoulesFraiches();  
    }
```

```
}
```

```
}
```

↑ Comme Brest est sur la côte,
nous avons des moules fraîches.
Strasbourg devra se contenter
de moules surgelées.

Les meilleurs poivrons en
rondelles. Ils sont partagés
entre Brest et Strasbourg.
N'oubliez pas d'en utiliser
page suivante quand vous
implémenterez vous-même la
fabrique de Strasbourg

Pizzeria

```
public abstract class Pizza {  
    String nom;  
    Pate pate;  
    Sauce sauce;  
    Legume legume[];  
    Fromage fromage;  
    Poivrons poivrons;  
    Moules moules;  
  
    abstract void preparer();  
  
    void cuire() {  
        System.out.println("Cuisson 25 minutes à 180°");  
    }  
  
    void couper() {  
        System.out.println("Découpage en parts triangulaires");  
    }  
  
    void emballer() {  
        System.out.println("Emballage dans une boîte officielle");  
    }  
  
    void setNom(String nom) {  
        this.nom = nom;  
    }  
  
    String getNom() {  
        return nom;  
    }  
  
    public String toString() {  
        // code qui affiche la pizza  
    }  
}
```

Chaque pizza contient un ensemble d'ingrédients utilisés dans sa préparation.

Nous avons maintenant rendu la méthode `preparer()` abstraite.

C'est là que nous allons collecter les ingrédients nécessaires pour la pizza. Bien entendu, ils proviennent de la fabrication d'ingrédients.

Nos autres méthodes demeurent les mêmes : seule la méthode `preparer()` a changé.


Pizzeria

```
public class PizzaFromage extends Pizza {  
    FabriqueIngredientsPizza fabriqueIngredients;
```


```
    public PizzaFromage(FabriqueIngredientsPizza fabriqueIngredients) {  
        this.fabriqueIngredients = fabriqueIngredients;  
    }
```

```
    void preparer() {  
        System.out.println("Préparation de " + nom);  
        pate = fabriqueIngredients.creerPate();  
        sauce = fabriqueIngredients.creerSauce();  
        fromage = fabriqueIngredients.creerFromage();  
    }  
}
```

Maintenant, pour faire une pizza, nous avons besoin d'une fabrication qui fournit les ingrédients. Une fabrication est donc transmise au constructeur de chaque classe Pizza et stockée dans une variable d'instance.



← C'est là que c'est magique !



La méthode preparer() parcourt les étapes de la création d'une pizza au fromage. Chaque fois qu'elle a besoin d'un ingrédient, elle demande à la fabrication de le produire.

Pizzeria

```
public class PizzeriaBrest extends Pizzeria {  
  
    protected Pizza creerPizza(String item) {  
        Pizza pizza = null;  
        FabriqueIngredientsPizza fabriqueIngredients =  
            new FabriqueIngredientsPizzaBrest();  
        if (choix.equals("fromage")) {  
  
            pizza = new PizzaFromage(fabriqueIngredients);  
            pizza.setNom("Pizza au fromage style Brest");  
        } else if (choix.equals("vegetarienne")) {  
  
            pizza = new PizzaVegetarienne(fabriqueIngredients);  
            pizza.setNom("Pizza végétarienne style Brest");  
  
        } else if (choix.equals("fruitsDeMer")) {  
  
            pizza = new PizzaFruitsDeMer(fabriqueIngredients);  
            pizza.setNom("Pizza aux fruits de mer style Brest");  
        } else if (choix.equals("poivrons")) {  
  
            pizza = new PizzaPoivrons(fabriqueIngredients);  
            pizza.setNom("Pizza aux poivrons style Brest");  
  
        }  
        return pizza;  
    }  
}
```

La boutique de Brest est composée avec une fabrique d'ingrédients Brest. Elle sera utilisée pour produire les garnitures de toutes les pizzas style Brest.

Nous transmettons maintenant à chaque pizza la fabrique qu'il faut utiliser pour produire ses ingrédients.

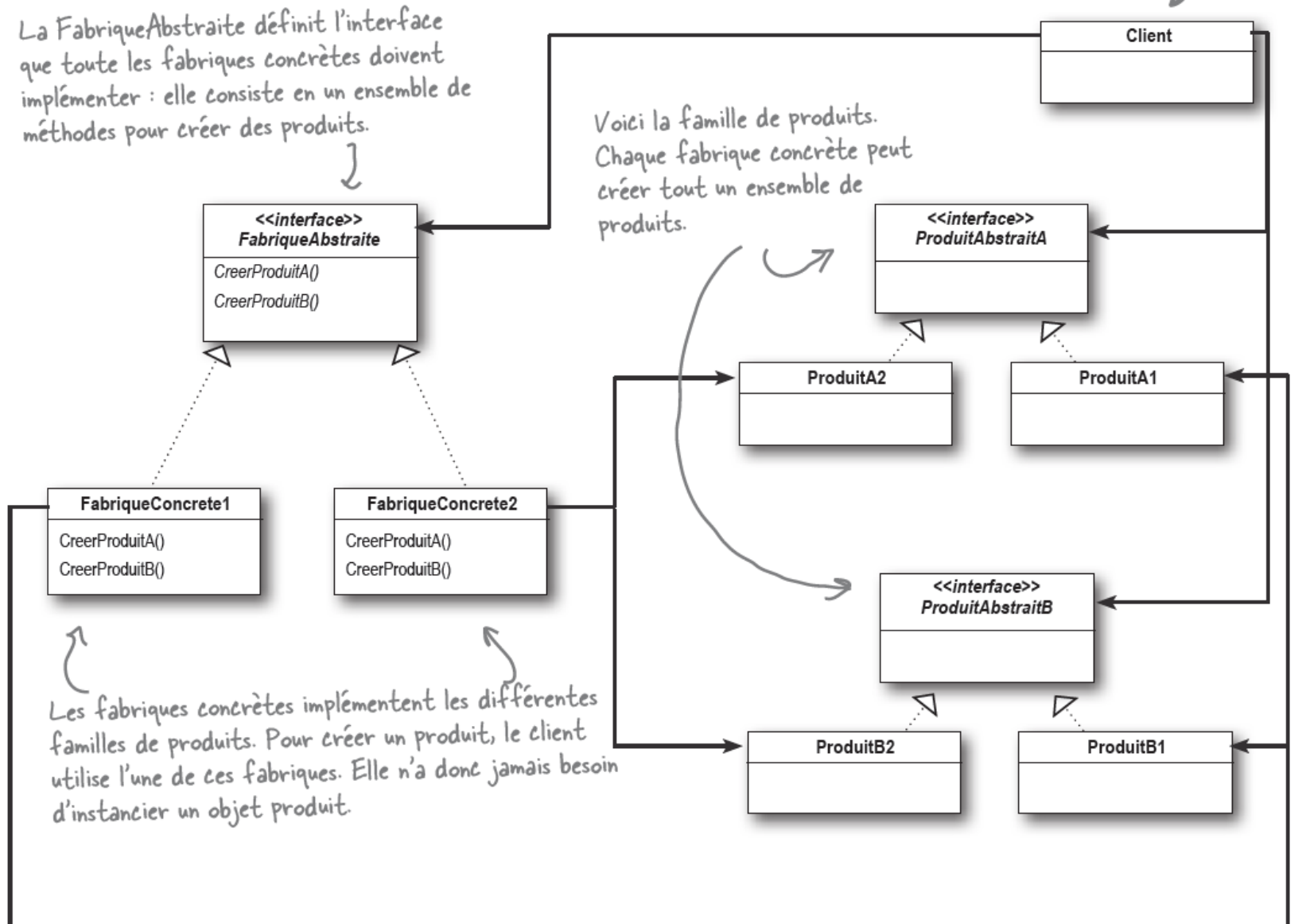
Revenez page précédente et vérifiez que vous avez compris comment la pizza et la fabrique collaborent !

Pour chaque type de Pizza, nous instancions une nouvelle Pizza et nous lui transmettons la fabrique dont elle a besoin pour obtenir ses ingrédients.

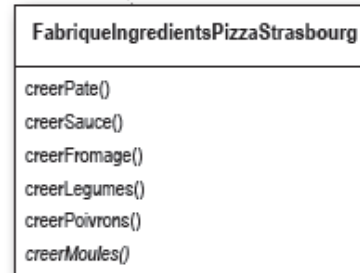
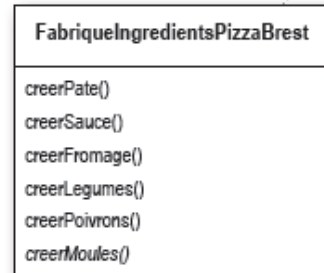
Le pattern Fabrique Abstraite : définition

Le pattern Fabrique Abstraite fournit une interface pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leurs classes concrètes..

Le pattern Fabrique Abstraite : définition

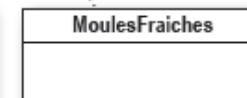
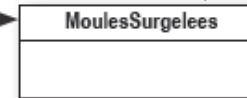
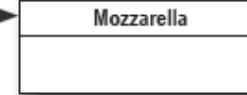
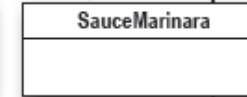
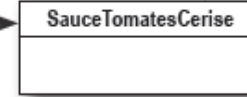
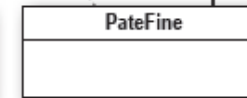
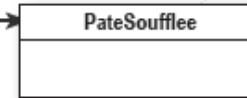
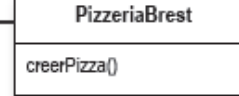
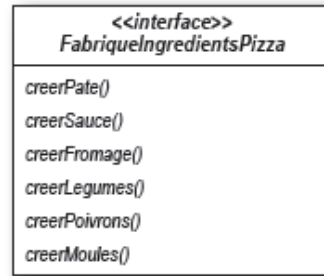


L'interface abstraite
FabriqueIngredientsPizza est l'interface
qui définit la façon de créer une famille de
produits apparentés – tout ce qu'il nous faut
pour confectionner une pizza.



La tâche des fabriques
concrètes consiste à
fabriquer les ingrédients
des pizzas. Chaque
fabrique sait comment
créer les objets qui
correspondent à sa région.

Chaque fabrique produit une implémentation
différente de la famille de produits.



Gestion des véhicules électriques et thermiques

- Vous devez développer une application pour une usine de fabrication de véhicules qui produit deux types de véhicules : électriques et thermiques. Chaque type de véhicule possède deux catégories : voiture et camion.
- Problème :
- Vous devez implémenter un système qui permet à l'usine de produire les véhicules en utilisant une architecture flexible et extensible. Les véhicules partagent des caractéristiques communes, mais les détails de production diffèrent en fonction du type (électrique ou thermique).

Indices

- Exigences :

Classes d'usines : Créez deux usines concrètes :

UsineElectrique pour les véhicules électriques.

UsineThermique pour les véhicules thermiques.

Produits concrets : VoitureÉlectrique et CamionÉlectrique pour l'usine électrique. VoitureThermique et CamionThermique pour l'usine thermique.

Abstract Factory : Déclarez une interface ou classe abstraite VehiculeFactory qui définit les méthodes creerVoiture() et creerCamion().

Interface Produit : Déclarez deux interfaces communes pour les véhicules : Voiture et Camion.