

PizzaStore :

Un PizzaStore dépendant (18/37)

- ▶ Hypothèse: On n'a jamais entendu parler du factory
 - ▶ Compter le nombre d'objets de pizzas concrètes dont cette classe dépend
 - ▶ Refaire le compte si on ajoute des pizzas de style bizertin

```
public class DependentPizzaStore {
    Pizza createPizza(String style, String type) {
        Pizza pizza=null;
        if (style.equals("Sfax")){
            if (type.equals("cheese")){
                pizza=new SfaxStyleCheesePizza();
            } else if (type.equals("pepperoni")){
                pizza=new SfaxStylePepperoniPizza();
            } else if (type.equals("clam")){
                pizza=new SfaxStyleClamPizza();
            }
        } else if (style.equals("Tunis")){
            if (type.equals("cheese")){
                pizza=new TunisStyleCheesePizza();
            } else if (type.equals("pepperoni")){
                pizza=new TunisStylePepperoniPizza();
            } else if (type.equals("clam")){
                pizza=new TunisStyleClamPizza();
            }
        } else {System.out.println("Erreur: type de pizza invalide");}
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

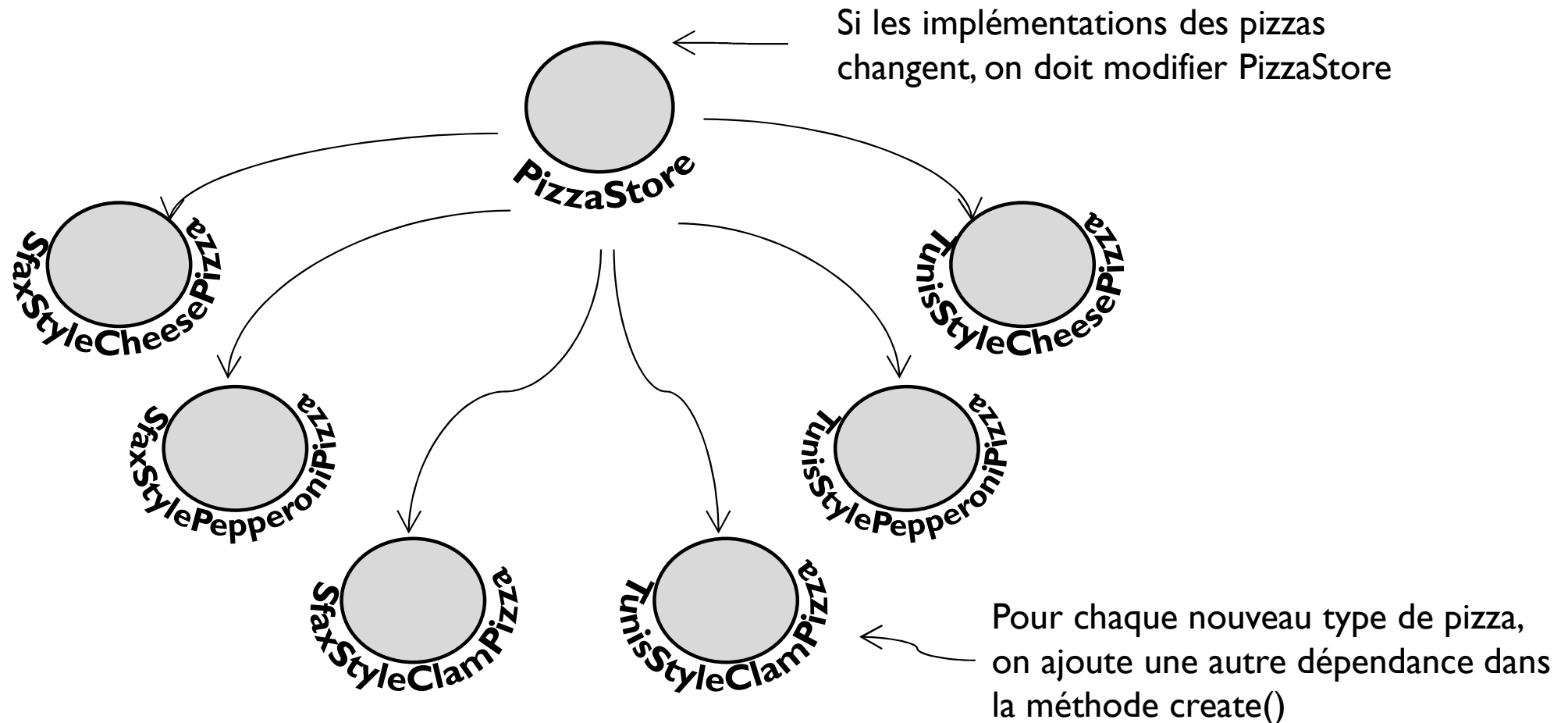
Gérer toutes les pizzas de style sfaxien

Gérer toutes les pizzas de style tunisois

PizzaStore :

La dépendance entre les objets (19/37)

- ▶ Cette version de PizzaStore dépend de tous les objets pizzas parce qu'elle les crée directement
- ▶ On dit que PizzaStore dépend des implémentations des pizzas parce que chaque changement des implémentations concrètes des pizzas, affecte le PizzaStore



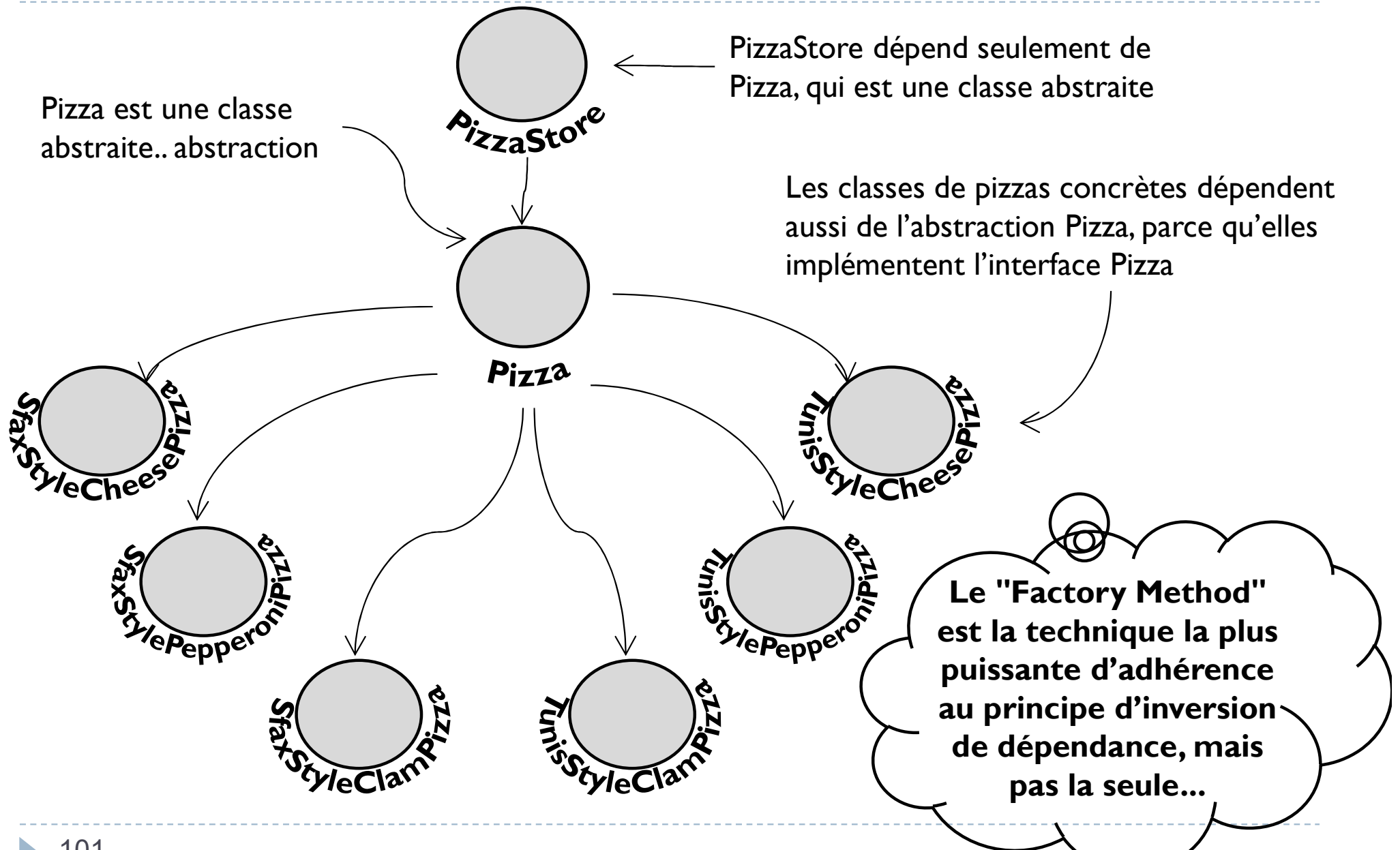
PizzaStore :

L'inversion de dépendance (20/37)

- ▶ Réduire les dépendances aux classes concrètes dans notre code, est une "bonne chose"
- ▶ Le principe qui formalise cette notion s'appelle "principe d'inversion de dépendance" :
 - ▶ **Règle 6:** Dépendre des abstractions. Ne jamais dépendre de classes concrètes.
- ▶ Ce principe prétend que nos "haut-niveau" composants ne doivent pas dépendre de nos "bas-niveau" composants; plutôt, les deux doivent dépendre des abstractions.
- ▶ Un composant de haut-niveau (PizzaStore) est une classe dont le comportement dépend des autres composants de bas-niveau(Pizza)

PizzaStore :

Appliquons ce principe (21/37)



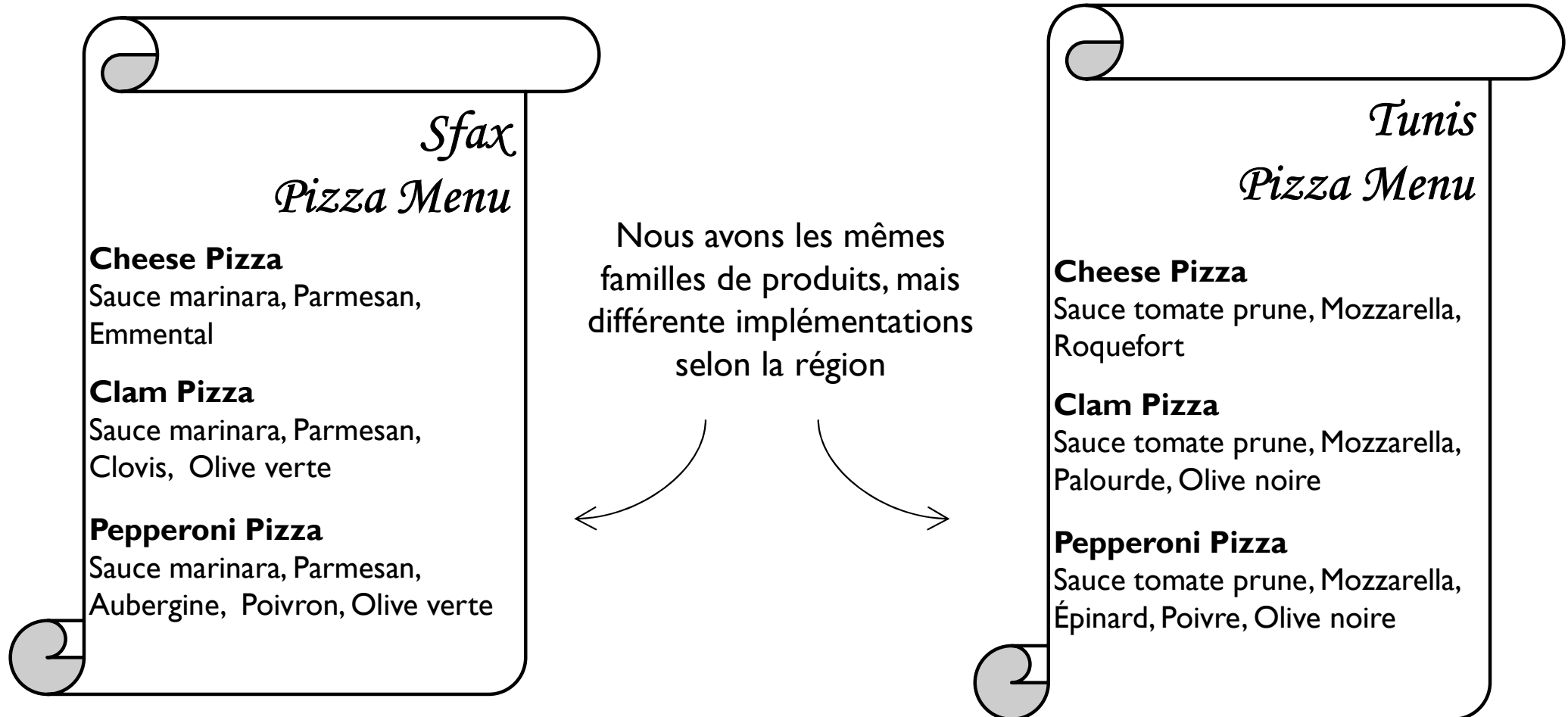
PizzaStore :

Les ingrédients des pizzas (22/37)

- ▶ Problème : quelques franchises n'ont pas utilisé la même procédure de préparation, et ce en substituant des ingrédients par d'autres de basse qualité, afin d'augmenter leur marge.
- ⚠ Il faut assurer la consistance des ingrédients
- ▶ Solution : créer un factory qui produit les ingrédients, et les transporter aux franchises
- ▶ Le seul problème avec ce plan : Ce qui est sauce rouge à Sfax, n'est pas sauce rouge à Tunis
 - ▶ Il y a un ensemble d'ingrédients à transporter à Sfax, et un autre ensemble à transporter à Tunis

PizzaStore :

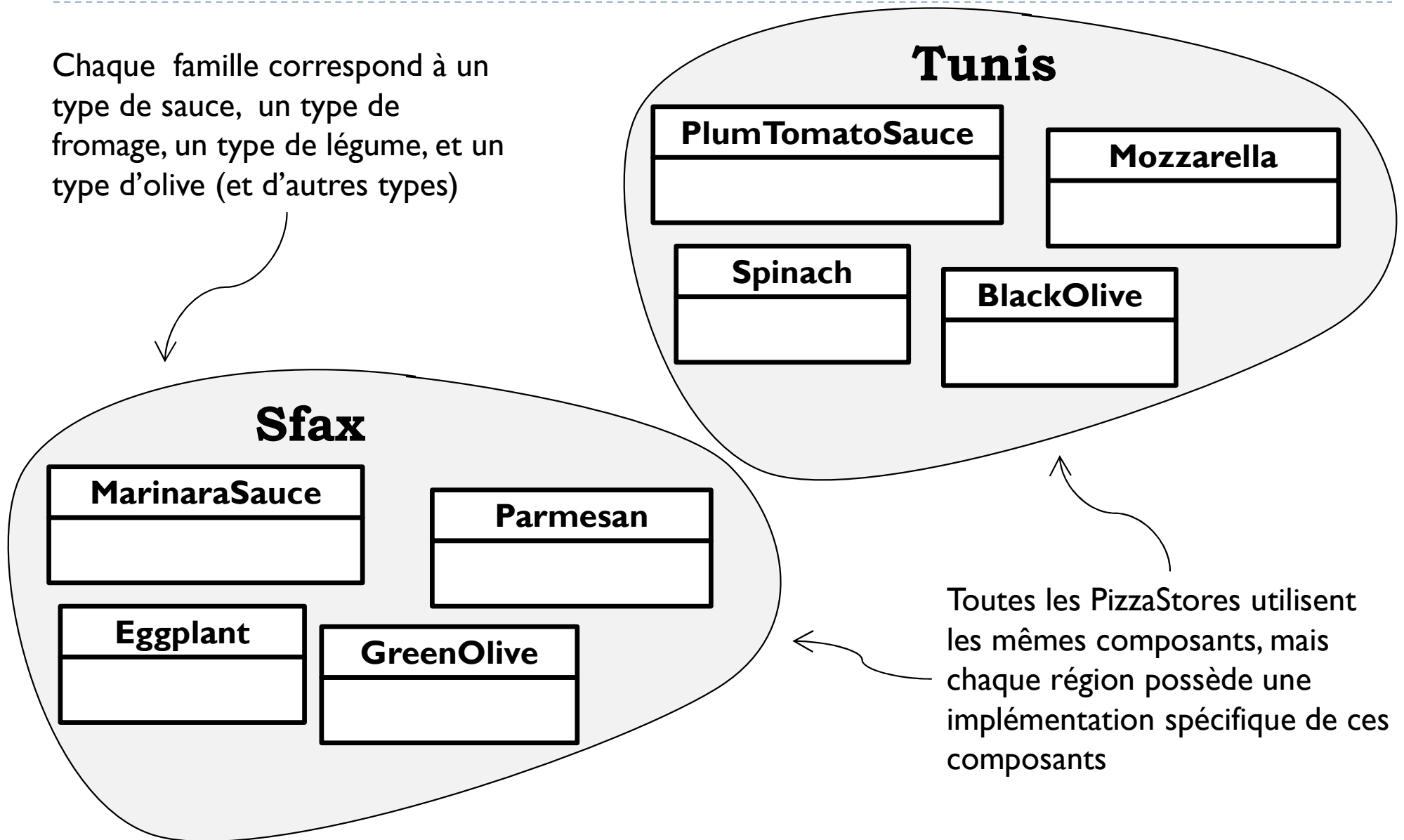
Les menus des pizzas (23/37)



PizzaStore :

Les familles des ingrédients (24/37)

Chaque famille correspond à un type de sauce, un type de fromage, un type de légume, et un type d'olive (et d'autres types)




PizzaStore :

Les factories des ingrédients (25/37)


- ▶ Le factory est le responsable de la création de la pâte, la sauce, le fromage, etc.

```
public interface PizzaIngredientFactory {  
    Dough createDough();  
    Sauce createSauce();  
    Cheese createCheese();  
    Veggies[] createVeggies();  
    Clam createClam();  
}
```

Pour chaque ingrédient, on crée une create() méthode dans notre interface



Beaucoup de nouvelles classes, une par ingredient



- ▶ A faire :
 - ▶ Construire un facotry pour chaque région: une sous-classe de PizzaIngredientFactory qui implémente chaque create() méthode
 - ▶ Implémenter un ensemble de classes d'ingrédients, à utiliser par les factories tels que: OliveVerte, Mozzarella, SauseMarinara, etc.
 - ▶ Lier ceci avec notre ancien code de PizzaStore, tout en travaillant nos factories d'ingrédients

PizzaStore :

Les factories de Sfax (26/37)

```
public class SfaxPizzaIngredientFactory implements PizzaIngredientFactory
{
    public Dough createDough() {
        return new ThinDough();
    }
    public Sauce createSauce() {
        return new MarinaraSauce();
    }
    public Cheese createCheese() {
        return new Parmesan();
    }
    public Veggies[] createVeggies() {
        Veggies veggies[] = {new Garlic(), new Onion(), new EggPlant()};
        return veggies;
    }
    public Clam createClam() {
        return new Clovis();
    }
}
```

Pour chaque famille d'ingrédient, on crée la version sfaxienne

Palourde() pour le cas de tunis

PizzaStore :

Les factories de Tunis (26'/37)

```
public class TunisPizzaIngredientFactory implements PizzaIngredientFactory
{
    public Dough createDough() {
        return new CrunchyDough();
    }
    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }
    public Cheese createCheese() {
        return new Mozzarella();
    }
    public Veggies[] createVeggies() {
        Veggies veggies[] = {new Garlic(), new Onion(), new Spinach()};
        return veggies;
    }
    public Clam createClam() {
        return new Palourde();
    }
}
```

Pour chaque famille d'ingrédient, on crée la version Tunisoise



PizzaStore :

Retravayillons la classe Pizza (27 / 37)

```
public abstract class Pizza {  
    protected String name;  
    protected Dough dough;  
    protected Sauce sauce;  
    protected Cheese cheese;  
    protected Veggies veggies[];  
    protected Clam clam;  
    public abstract void prepare();  
    public void bake() {  
        System.out.println("Cuire durant 25mn à 280°");  
    }  
    public void cut() {  
        System.out.println("Couper en morceaux à la diagonale");  
    }  
    public void box() {  
        System.out.println("Placer la pizza dans un boitier officiel");  
    }  
    public void setName(String s) {  
        name=s;  
    }  
    public String getName() {  
        return name;  
    }  
}
```

Les ingrédients d'une paizza
(liste non-exhaustive)

La collecte des ingrédients se fait dans cette
méthode (à travers un factory d'ingrédients)
qui sera définie par les classes dérivées

Les autres méthodes sont les
mêmes (à l'exception de prepare())

PizzaStore :

Retravajillons les classes des Pizzas (28/37)

Pour faire la pizza, on besoin d'un factory. Chaque classe Pizza prend le factory à travers son constructeur

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

Chaque fois que la méthode prepare()
a besoin d'ingrédient, elle appelle le
factory pour le produire

PizzaStore :

Retravajillons les classes des Pizzas (29/37)

Un factory pour chaque type de Pizza

```
public class ClamPizza extends Pizza {  
    private PizzaIngredientFactory ingredientfactory;  
    public ClamPizza(PizzaIngredientFactory ingredientfactory) {  
        this.ingredientfactory = ingredientfactory;  
    }  
    @Override  
    public void prepare() {  
        System.out.println("Préparons " + name);  
        dough = ingredientfactory.createDough();  
        sauce = ingredientfactory.createSauce();  
        cheese = ingredientfactory.createCheese();  
        clam= ingredientfactory.createClam();  
    }  
}
```

Si c'est le factory de sfax, on
va préparer des clovis

Pour faire une ClamPizza, la méthode
prépare les ingrédients
correspondants de son factory local.

PizzaStore :

Retravayillons les PizzaStores (30/37)

Le store de Sfax est composé d'un
factory sfaxien d'ingrédients/

```
public class SfaxPizzaStore extends PizzaStore {  
    private PizzaIngredientFactory sfaxIngredientfactory =  
        new SfaxPizzaIngredientFactory();
```

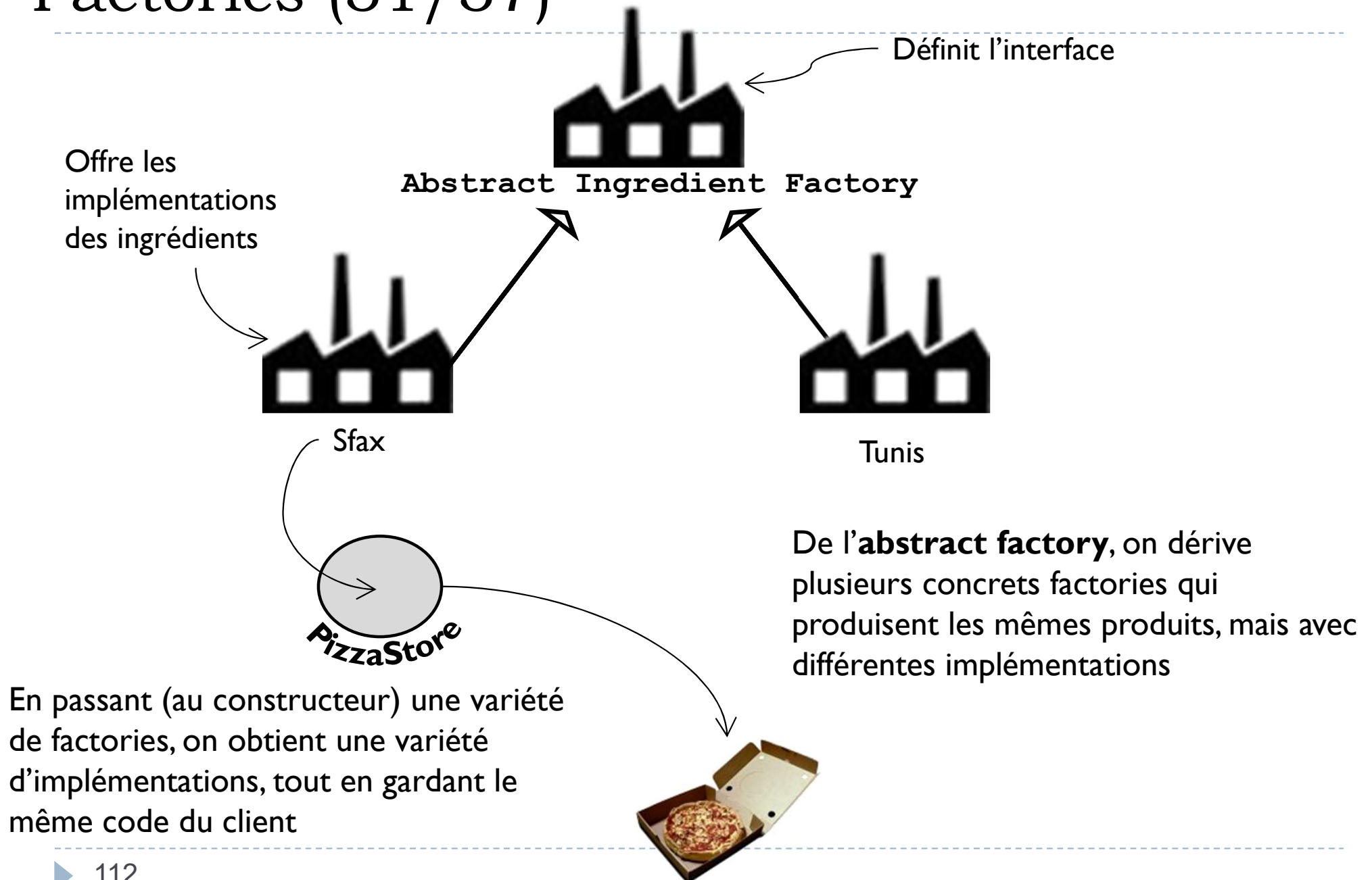
@Override

```
    public Pizza createPizza(String item) {  
        Pizza pizza = null;  
        if (item.equals("cheese")) {  
            pizza = new CheesePizza(sfaxIngredientfactory);  
            pizza.setName("Sfax Style Cheese Pizza");  
        } else if (item.equals("pepperoni")) {  
            pizza = new PepperoniPizza(sfaxIngredientfactory);  
            pizza.setName("Sfax Style Pepperoni Pizza");  
        } else if (item.equals("clam")) {  
            pizza = new ClamPizza(sfaxIngredientfactory);  
            pizza.setName("Sfax Style Clam Pizza");  
        }  
        return pizza;  
    }  
}
```

On passe à chaque pizza le factory
censé créer ses ingrédients

PizzaStore :

Factories (31/37)



PizzaStore :

Commander des pizzas (32/37)

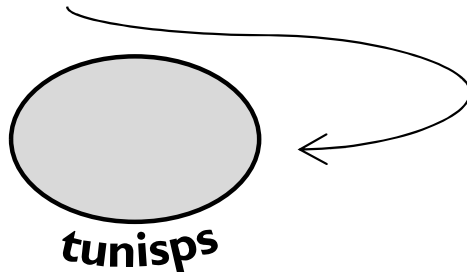
Je voudrai une pizza
de grande taille avec beaucoup
de fromage du style
tunisien

Je voudrai une pizza
de taille moyenne avec
peu de fromage et au thon
du style sfaxien

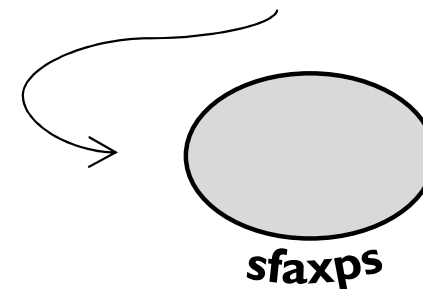


```
PizzaStore tunisps =  
new TunisPizzaStore();
```

```
PizzaStore sfaxps =  
new SfaxPizzaStore();
```



Instance du store spécifique



Prendre des commandes

```
tunisps.orderPizza("cheese");
```

```
sfaxps.orderPizza("cheese");
```

C'est une méthode de l'instance tunisps (respectivement
sfaxps), définie dans la classe PizzaStore

PizzaStore :

Commander des pizzas (33/37)

```
tunisps.orderPizza("cheese");
```

```
sfaxps.orderPizza("cheese");
```

La méthode orderPizza() appelle initialement createPizza()

```
Pizza pizza= createPizza("cheese");
```

```
Pizza pizza= createPizza("cheese");
```

La méthode createPizza() implique le factory d'ingrédients

```
Pizza pizza= new  
CheesePizza(tunisIngeredientFactory);
```

```
Pizza pizza= new  
CheesePizza(sfaxIngeredientFactory);
```

Chaque instance de pizza est associée à un factory d'ingrédients

La méthode prepare() est appelée et chaque factory est appelé pour produire les ingrédients de la région

```
void prepare() {
```

```
void prepare() {
```

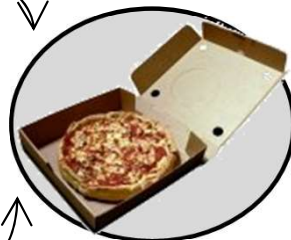
PizzaStore :

Commander des pizzas (34/37)

```
void prepare() {  
    dough = factory.createDough();  
    // Pâte coustillante  
    sauce = factory.createSauce();  
    // Sauce tomate prune  
    cheese = factory.createCheese();  
    // Mozzarella, Roquefort  
}
```

Elle prépare une pizza au
fromage avec les ingrédients
du style tunisien

```
pizza.bake();  
pizza.cut();  
pizza.box();
```



Pizza

De style tunisien

```
void prepare() {  
    dough = factory.createDough();  
    // Pâte mince  
    sauce = factory.createSauce();  
    // Sauce marinara  
    cheese = factory.createCheese();  
    // Parmesan, Emmental  
}
```

Elle prépare une pizza au
fromage avec les ingrédients
du style sfaxien

```
pizza.bake();  
pizza.cut();  
pizza.box();
```



Pizza

De style sfaxien

On termine la création

PizzaStore :

Le patron Abstract Factory (35/37)

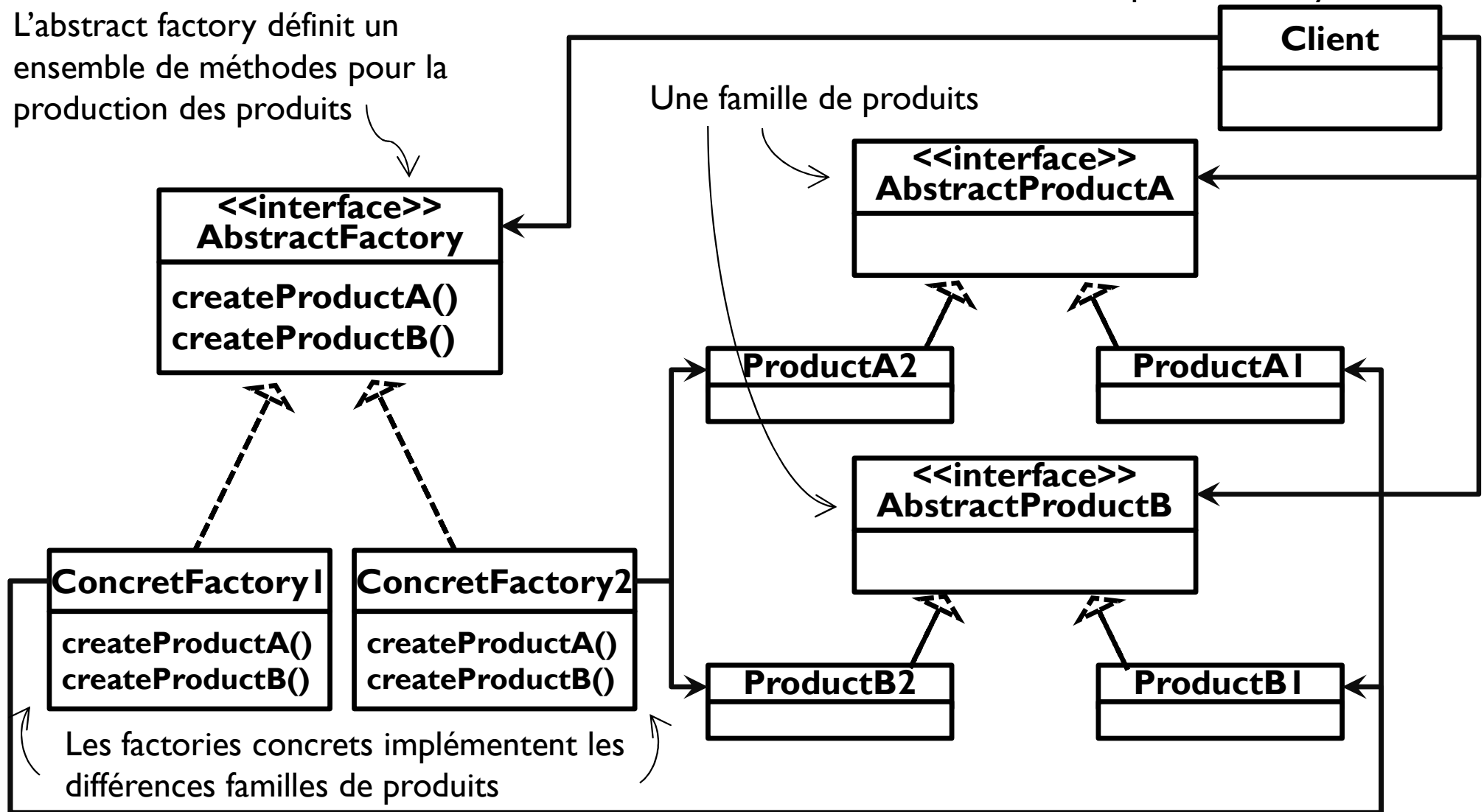
- ▶ Définition: **Abstract Factory**
 - ▶ Le **patron abstract factory** offre une interface de création de familles d'objets dépendants (en relation), sans spécifier leurs classes concrètes.

PizzaStore :

Le diagramme de classes du patron (36/37)

Le client est composé au moment de l'exécution par un factory concret

L'abstract factory définit un ensemble de méthodes pour la production des produits

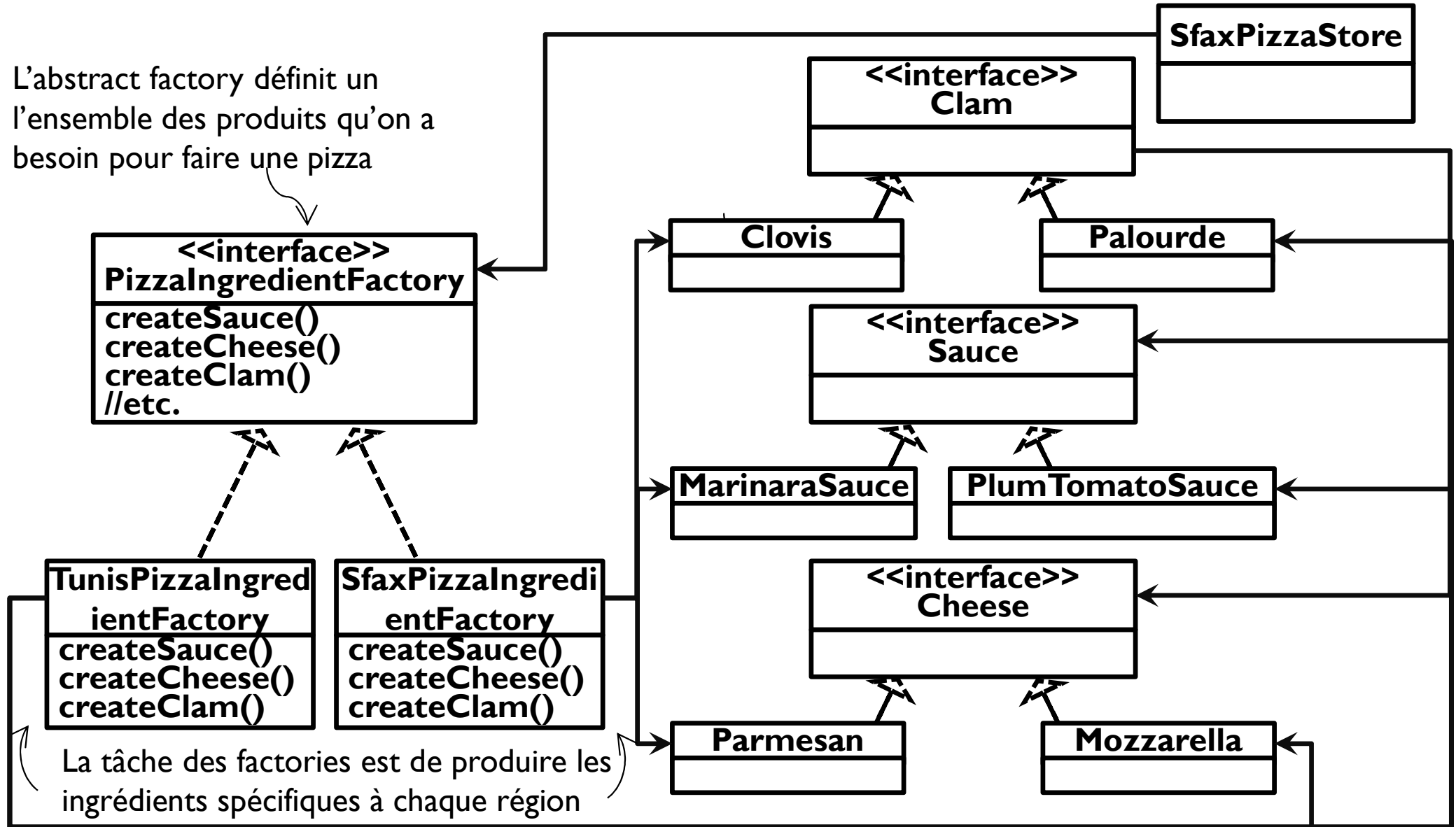


PizzaStore :

La conception finale (37/37)

Les clients de l'abstract factory sont les stores de Sfax et de Tunis

L'abstract factory définit un ensemble des produits qu'on a besoin pour faire une pizza



La tâche des factories est de produire les ingrédients spécifiques à chaque région

Chaque factory produit différent implémentation de chaque famille de produits

Récapitulatif (1 / 2)

- ▶ Bases de l'OO: Abstraction, Encapsulation, Polymorphisme & Héritage
- ▶ Principes de l'OO
 - ▶ Encapsuler ce qui varie
 - ▶ Favoriser la composition sur l'héritage
 - ▶ Programmer avec des interfaces et non des implémentations
 - ▶ Opter pour une conception faiblement couplée
 - ▶ Les classes doivent être ouvertes pour les extensions et fermées pour les modifications
 - ▶ **Dépendre des abstractions. Ne jamais dépendre de classes concrètes**
- ▶ Patron de l'OO
 - ▶ Strategy: définit une famille d'algorithmes interchangeables
 - ▶ Observer: définit une dépendance 1-à-plusieurs entre objets.
 - ▶ Decorator: attache des responsabilités additionnelles à un objet dynamiquement.
 - ▶ **Factory Method**: définit une interface de création des objets, et laisse les classes-dérivées décider de la classe de l'instanciation.
 - ▶ **Abstract Factory**: offre une interface de création de familles d'objets dépendants, sans spécifier leurs classes concrètes

Récapitulatif (2/2)

- ▶ Tous les factories encapsule la création des objets
- ▶ Malgré qu'il n'est un vrai patron, le "Simple Factory" est une manière simple de découplage de clients des concrètes classes
- ▶ Factory Method repose sur l'héritage: La création d'objet est déléguée aux sous-classes qui implémentent la méthode factory de création d'objets
- ▶ Abstract Factory repose sur la composition d'objets: La création d'objet est implémentée dans une méthode exposée dans l'interface du factory
- ▶ Tous les patrons factories soutiennent le faible couplage entre notre application et les classes concrètes.
- ▶ L'intension de Factory Method est de permettre à une classe de reporter l'instanciation à ses sous-classes.
- ▶ L'intension d'Abstract Factory est de créer une famille d'objets en relation sans dépendre de leurs classes concrètes
- ▶ L'inversion de dépendance nous guide afin d'éviter les dépendances des classes concrètes, et s'efforcer pour les abstractions
- ▶ Factories sont des techniques puissantes de codages des abstractions et non des classes concrètes