Objet et développement d'applications
Cours 7 - Patterns Factory Method et Abstract Factory

Patrons de conception pour POO

Florian Richoux

2014-2015

Polymorphisme

En POO, le polymorphisme nous permet de changer le comportement du programme dynamiquement, pendant son exécution.

Exemple

```
class A {
  public int f() { return 42; }
}

class B extends A {
  public int f() { return 24; }
}

class Main {
  public static void main( String[] args ) {
    A tab[] = new A[4];

    tab[0] = new A();
    tab[1] = new B();
    tab[2] = new A();
    tab[3] = new B();

    for( int i = 0; i < 4; ++i )
      System.out.println( tab[i].f() );
  }
}
```

Polymorphisme

En POO, le polymorphisme nous permet de changer le comportement du programme dynamiquement, pendant son exécution.

Et pour la création d'objet ?

A t-on la même flexibilité pour la création d'objet ?

Instancier un objet en Java

```
TypeVariable variable;
variable = new TypeObjet();
```

Instancier un objet en Java

```
TypeVariable variable;
variable = new TypeObjet();
```

Passage obligatoire

On doit bien avoir le mot-clé new pour instancier un objet, et derrière ce mot-clé doit apparaître un type qui doit être connu à la compilation.

Instancier un objet en Java

```
TypeVariable variable;
variable = new TypeObjet();
```

Passage obligatoire

On doit bien avoir le mot-clé new pour instancier un objet, et derrière ce mot-clé doit apparaître un type qui doit être connu à la compilation.

Différence fondamentale

On a donc une différence fondamentale (de souplesse) entre la création d'objet et l'invocation de méthodes.

So what?

Ok, mais quand on a besoin d'instancier un objet, c'est qu'on le connaît à l'avance, non ?

Pas forcément

- Laisser la connaissance de l'objet à créer à une autre entité (par exemple une méthode qui s'occupera de créer l'objet).
- Vous avez cette connaissance (sous forme d'une string, par exemple), mais pas accès au constructeur.

Instancier dynamiquement

Patterns Factory

Instancier dynamiquement, c'est possible en POO grâce aux patterns **Factory**. Avec Observer, ils sont parmi les patterns les plus utilisés.

Exemple de ce cours : la pizzeria

On va créer dynamiquement des pizzas.

Commander une pizza

Classe Pizzeria

```
class Pizzeria
{
   ...

   public Pizza commander()
   {
      Pizza pizza = new Pizza();

      pizza.preparer();
      pizza.cuire();
      pizza.couper();
      pizza.dansLaBoite();

      return pizza;
   }
}
```

Commander une certaine pizza

Pas n'importe quelle pizza

```
public Pizza commander( String type ) {
  Pizza pizza;

  if( type.equals( "Margherita" ) )
    pizza = new Margherita();
  else if( type.equals( "Royale" ) )
    pizza = new Royale();
  else if( type.equals( "Calzone" ) )
    pizza = new Calzone();

  pizza.preparer();
  pizza.cuire();
  pizza.couper();
  pizza.dansLaBoite();

  return pizza;
}
}
```

Changer le menu

Changements dans le menu

Vous ne vendez pratiquement aucune Margherita, et en contre partie vous avez une forte demande pour des pizzas végétariennes que vous ne proposez pas encore.

Modifications

Vous devez pas mal modifier votre classe Pizzeria !

- Supprimer la Margherita des if.
- Ajouter une nouvelle pizza végétarienne.

Ce qui change et ne change pas

Nouveau menu

```
    public Pizza commander( String type ) {
      Pizza pizza;

      if( type.equals( "Vegetarienne" ) )
        pizza = new Vegetarienne();
      else if( type.equals( "Royale" ) )
        pizza = new Royale();
      else if( type.equals( "Calzone" ) )
        pizza = new Calzone();

      pizza.preparer();
      pizza.cuire();
      pizza.couper();
      pizza.dansLaBoite();

      return pizza;
    }
}
```

Première approche

Rappel : bon principe de POO

Encapsuler ce qui change, ou est susceptible de changer.

Ce qui change

On extrait ce qui change (le bloc if/else) et on le case dans un méthode quelque part.

Ce qui ne bouge pas

On garde le reste (appels des méthodes preparer(), cuire(), ...) dans la classe Pizzeria.

Factory simple

Classe FactorySimple

```
class FactorySimple
{
  public Pizza fairePizza( String type )
  {
    Pizza pizza;

    if( type.equals( "Vegetarienne" ) )
      pizza = new Vegetarienne();
    else if( type.equals( "Royale" ) )
      pizza = new Royale();
    else if( type.equals( "Calzone" ) )
      pizza = new Calzone();

    return pizza;
  }
}
```

La nouvelle pizzeria

Nouvelle classe Pizzeria

```
class Pizzeria
{
  private FabriqueSimple fs_;
  public Pizzeria( FabriqueSimple fs ) { fs_ = fs; }
  ...

  public Pizza commander( String type )
  {
    Pizza pizza = fs_.fairePizza( type );

    //Remarque : plus de new dans la méthode !

    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.dansLaBoite();

    return pizza;
  }
}
```

So what 2?

Euh...

On n'a fait que de déplacer le problème dans une autre classe...

C'est vrai...

La solution n'est pas idéale, mais à quand même son utilité : si plusieurs classes font appel à la méthode `fairePizza(String)` de `FabriqueSimple`, on y gagne.

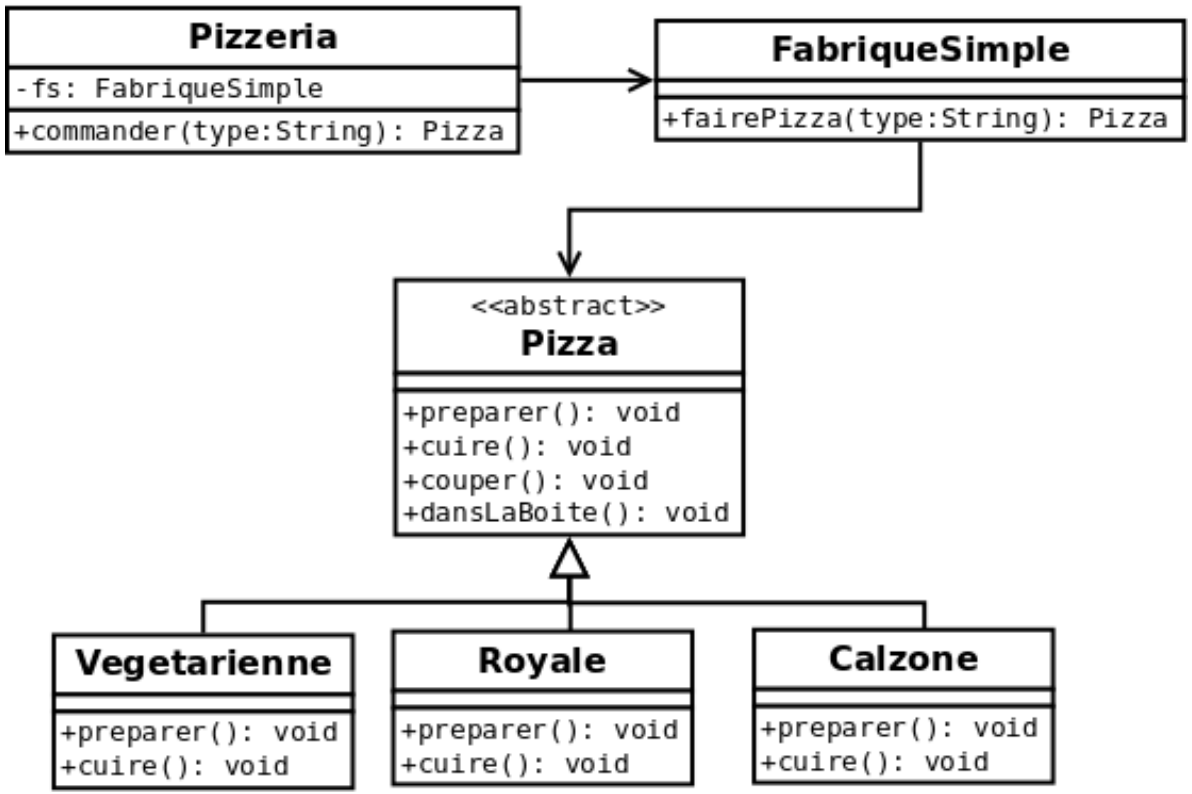
Exemple

On pourrait avoir une classe `Livraison` qui a aussi besoin d'instancier un objet Pizza, ou encore une classe `Menu`.

Trop simple ?

Simple Factory

La Simple Factory (fabrique simple) n'est communément pas admise comme un pattern. Il s'agit plus d'une astuce de programmation.



Revenons à nos pizzas

Votre pizzeria fait un carton !

Du coup, expansion mondiale : franchise à Nantes, New-York et Tokyo.

Vous souhaitez :

- que les enseignes suivent toutes la même procédure...
- ...mais adapter les pizzas au goût local.

Différentes pizzas du même type

```
tokyoPizzeria.commander( "Royale" );
```

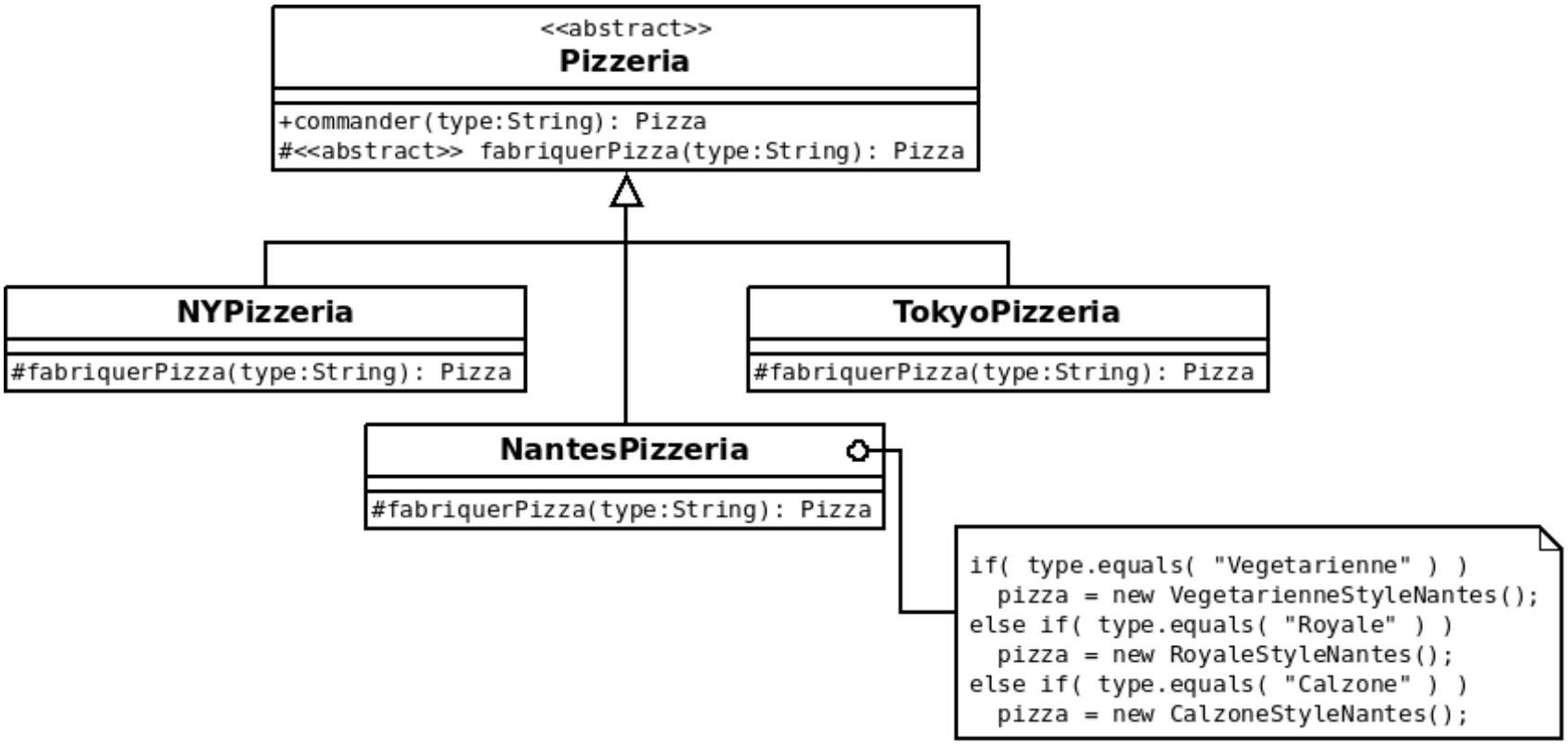
$\neq$

```
nantesPizzeria.commander( "Royale" );
```

Pizzeria abstraite

```
abstract class Pizzeria {
  public Pizza commander( String type ) {
    Pizza pizza = fabriquerPizza( type );

    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.dansLaBoite();

    return pizza;
  }

  protected abstract Pizza fabriquerPizza(String type);
}
```

Pizzeria abstraite



Itinéraire d'une pizza

1. Choix de sa pizzeria.

```
Pizzeria nantesPizzeria = new NantesPizzeria();
```

2. Commande de son type de pizza.

```
nantesPizzeria.commande( "Royale" );
```

3. Appel de la fabrique simple.

```
Pizza pizza = fabriquePizza( "Royale" );
```

4. Une fois la pizza créée, appels des méthodes communes à toutes les pizzerias et les pizzas.

```
pizza.preparer();     // Dans le code, on ne connait pas
pizza.cuire();        // la classe concrete de pizza.
pizza.couper();       // Pizzeria est donc decouplee
pizza.dansLaBoite(); // de la classe Pizza !
```

La classe Pizza

Classe Pizza

```
class abstract Pizza {
  protected String nom, pate, sauce;
  protected ArrayList<String> garnitures
                    = new ArrayList<String>();

  public void preparer() {
    print("Préparation d'une " + nom);
    print("Pâte : " + pate + "\nSauce : " + sauce);
    for( String g : garnitures )
    print(" - " + g);
  }

  public void cuire() { print("7 minutes."); }

  public void couper() { print("Coupée en 8."); }

  public void dansLaBoite() {
    print("Mise dans une boîte classique.");
  }
}
```

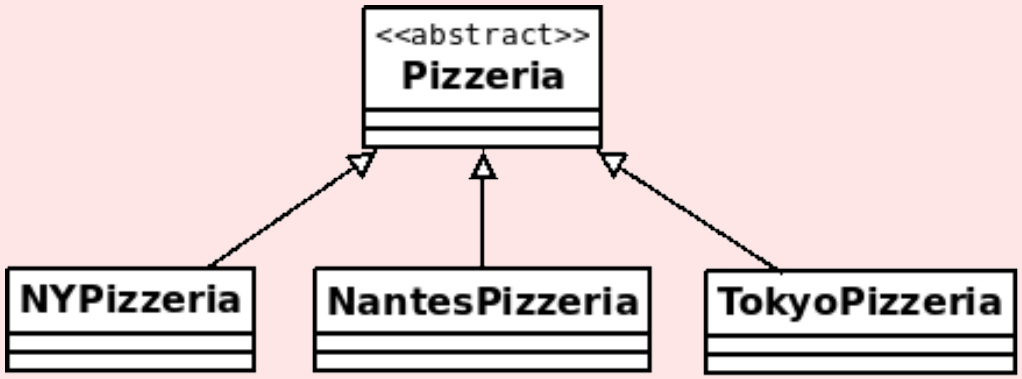
Pizza concrète

Classe RoyaleStyleNantes

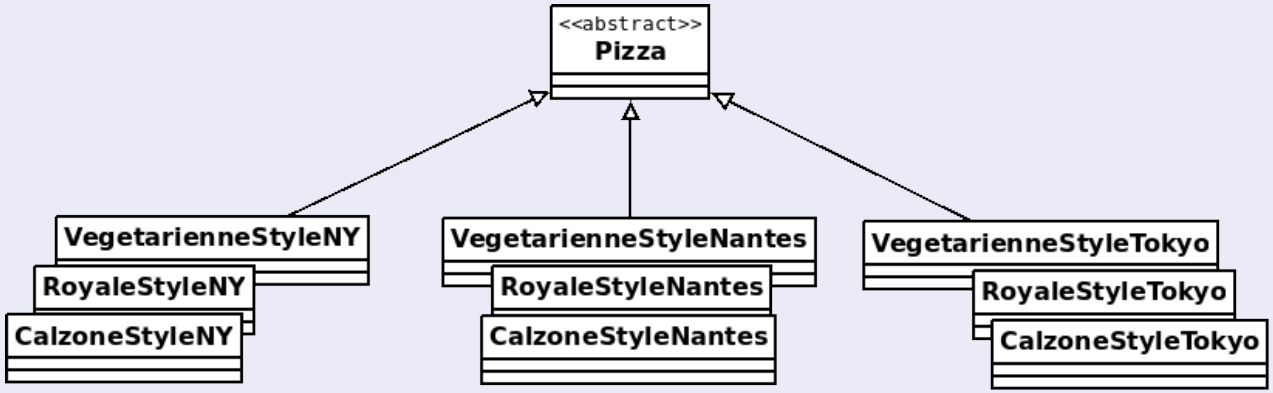
```
class RoyaleStyleNantes extends Pizza {
  public RoyaleStyleNantes()
  {
    nom = "Royale nantaise";
    pate = "Classique";
    sauce = "Tomates fraîches";
    garnitures.add("Champignon de Paris");
    garnitures.add("Épaule");
    garnitures.add("Fromage");
    garnitures.add("Oeuf");
  }

  // Cuire moins longtemps pour l'oeuf
  public void cuire() { print("6 minutes."); }
}
```

Le point sur ce que l'on connaît

Classes créatrices

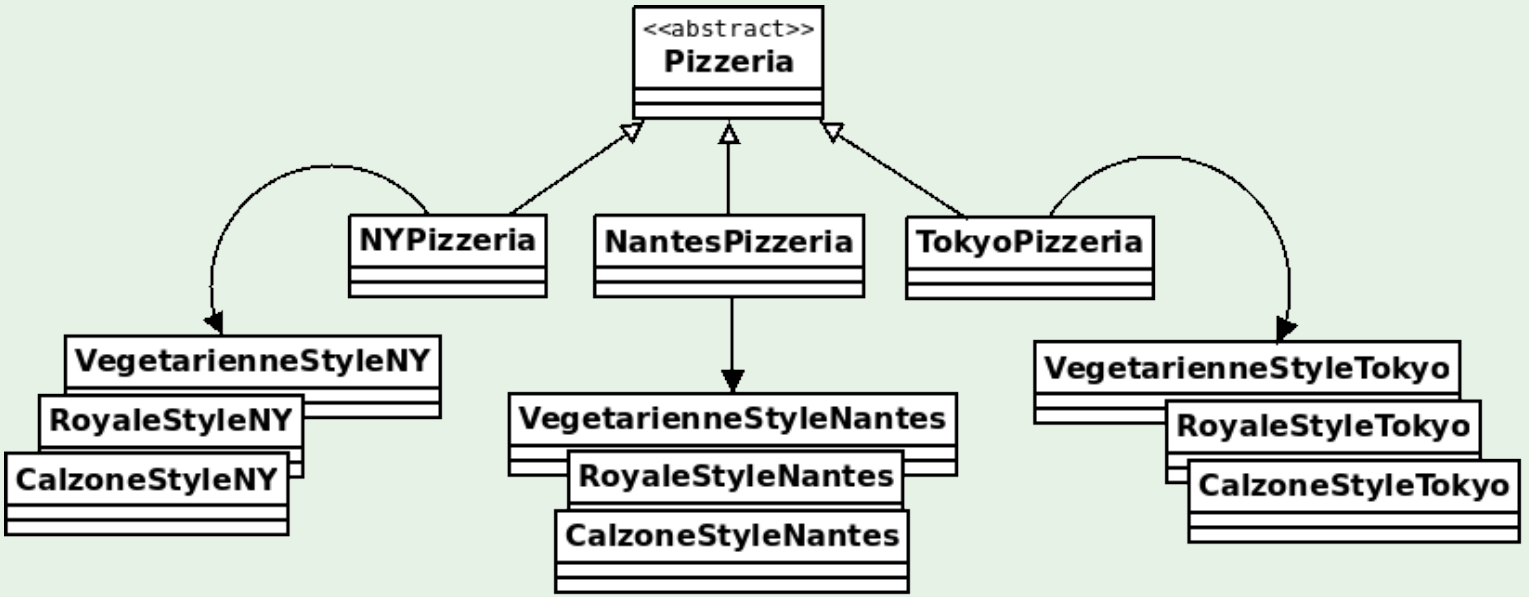


Classes produits



Pattern Factory Method

L'idée



Point-clé

Le point-clé des patterns Factory est l'encapsulation des connaissances pour la création des objets souhaités.

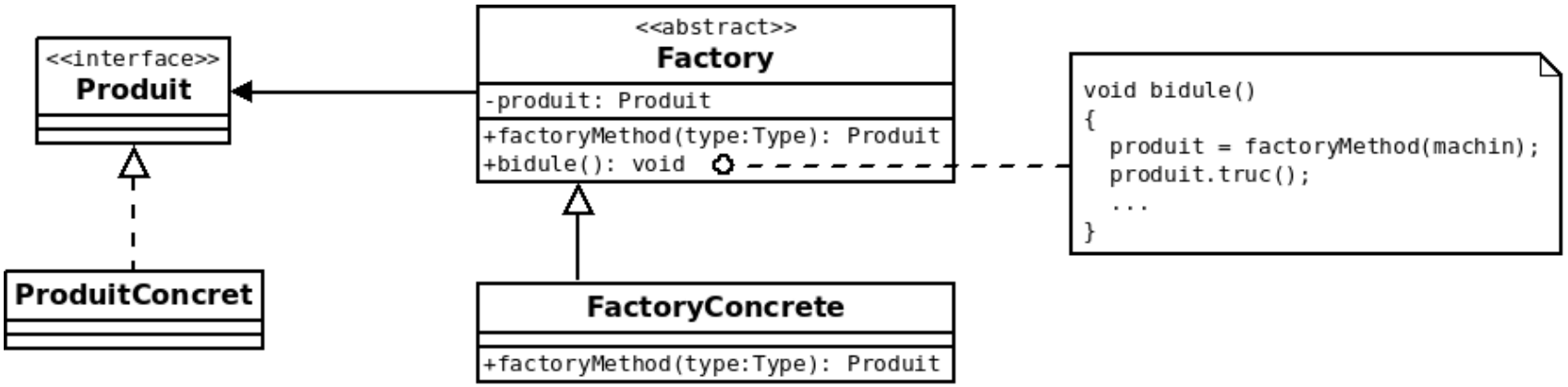
Pattern Factory Method

Définition

Le pattern Factory Method, aussi juste appelé Factory (pattern Fabrique ou Fabrication en français), définit une interface pour la création d'un objet, mais en laissant à des sous-classes le *choix* des classes à instancier. Factory Method permet à une classe de déléguer l'instanciation à des sous-classes.

Intuitivement

Pattern Factory Method = constructeur virtuel (autorisant le polymorphisme).



Pattern Factory Method

Flashback

. . . en laissant à des sous-classes le **choix** des classes à instancier.

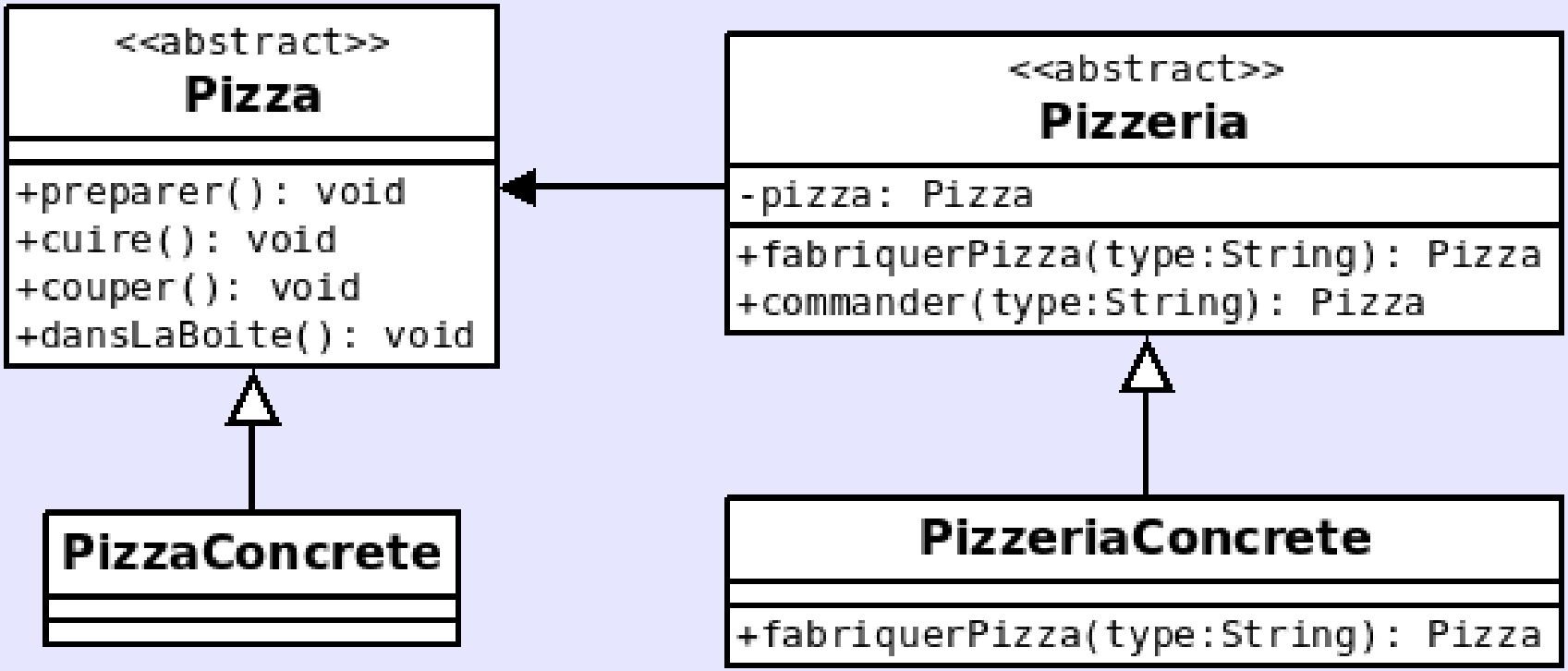
Kezako?

C'est bien sûr le programmeur ou l'utilisateur qui choisit les classes à instancier. Ce qu'il faut lire, c'est que l'interface de création ne connaît pas la classe concrète à instancier et laisse le soin aux sous-classes de s'en charger.

Exemple

`Pizzeria` sait qu'il faut fabriquer une pizza avec `fabriquerPizza("Royale")`, mais à aucune idée de ce qu'est une pizza royale, ni comment l'instancier. Par contre, `NantesPizzeria` sait faire des royales nantaises, elle !

Pattern Factory Method



Intérêts du pattern Factory Method

Encapsuler par Factory Method

L'encapsulation du code de création d'objets par une Factory Method permet :

- de pouvoir choisir dynamiquement le type de l'objet à instancier.
- de concentrer le code de création d'objets à un endroit, facilitant la maintenance.
- **d'avoir des classes clientes qui dépendent d'une interface pour créer des objets, et non de classes concrètes**

Nouveau bon principe de POO

Principe d'inversion de dépendance

Dépendez d'abstraction, pas de classes concrètes.

Vocabulaire

- Classe de bas niveau : classe définissant un comportement qui lui est propre. Ex: Pizza.
- Classe de haut niveau : classe dont le comportement dépend du comportement des classes qu'elle utilise. Ex: Pizzeria (qui dépend de Pizza).

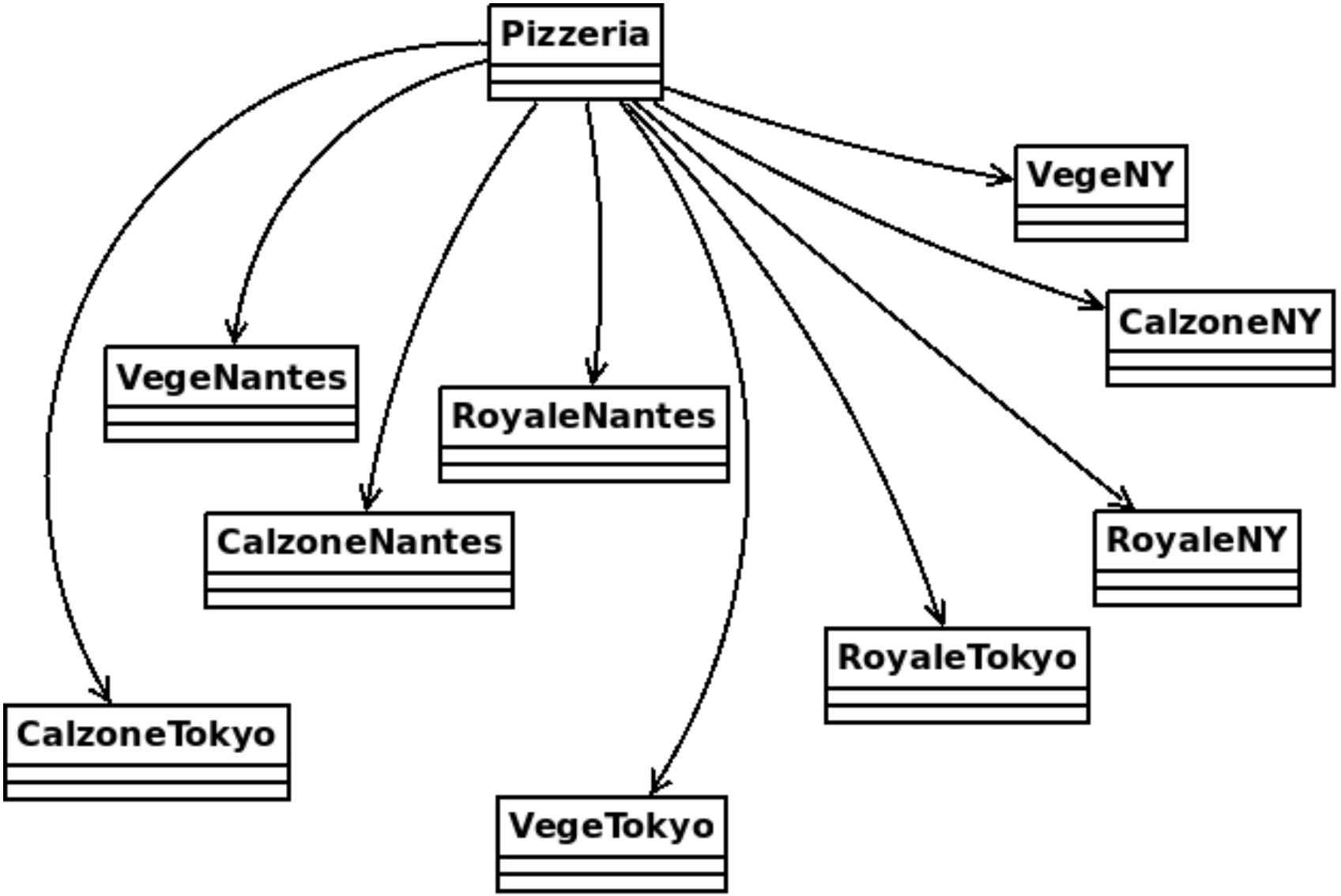
Principe d'inversion de dépendance

Une classe de haut niveau doit dépendre de classes de bas niveaux abstraites et non concrètes.

Forte dépendance de la classe Pizzeria

```
public Pizza commander( String type ) {
  Pizza pizza;

  if( type.equals( "VegetarienneStyleNantes" ) )
    pizza = new VegetarienneStyleNantes();
  else if( type.equals( "RoyaleStyleNantes" ) )
    pizza = new RoyaleStyleNantes();
  else if( type.equals( "CalzoneStyleNantes" ) )
    pizza = new CalzoneStyleNantes();
  else if( type.equals( "VegetarienneStyleNY" ) )
    pizza = new VegetarienneStyleNY();
  else if( type.equals( "RoyaleStyleNY" ) )
    pizza = new RoyaleStyleNY();
  else if ( type.equals( "CalzoneStyleNY" ) )
    pizza = new CalzoneStyleNY();
  ...
  }
}
```

Illustration



Illustration

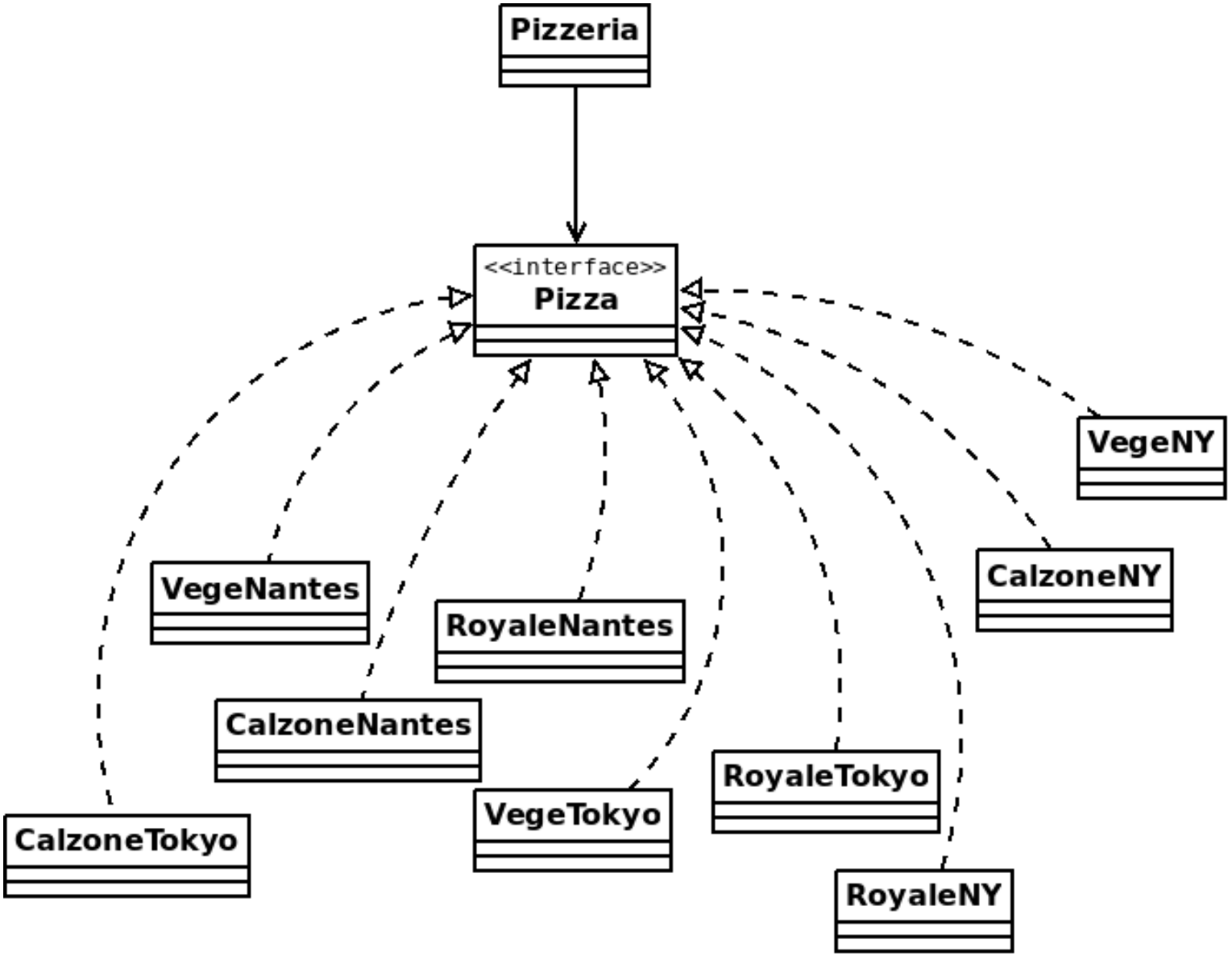
Mauvaise structure

Si une classe concrète change ou est supprimée (ou ajoutée), il faut modifier la classe de haut niveau Pizzeria !

Bonne structure

Avec une classe abstraite intermédiaire, on n'a plus ce problème. Observez dans le prochain schéma l'inversion des flèches (donc des dépendances) !

Illustration



Illustration

Ok, mais...

On a réglé le problème de dépendance de la classe de haut niveau, mais avant la pizzeria avait un accès à chaque pizza concrète, donc on pouvait instancier la pizza concrète que l'on souhaitait. Comment on fait maintenant ?

Illustration

Ok, mais...

On a réglé le problème de dépendance de la classe de haut niveau, mais avant la pizzeria avait un accès à chaque pizza concrète, donc on pouvait instancier la pizza concrète que l'on souhaitait. Comment on fait maintenant ?

Ben justement...

Le pattern Factory Method est là pour ça !

Approches **top-down** et **bottom-up**

Changez vos habitudes de programmation

Check-list pour vérifier le principe d'inversion de dépendance

1. Adopter l'approche bottom-up : définissez d'abord les classes de bas niveau, puis celles de haut niveau.
2. Avoir le moins possible de variables qui font référence à des objets.
3. Ne pas avoir de classe qui dérive d'une classe concrète.

Violation de la check-list

Bien entendu, vous aurez besoin de temps en temps de violer ces conseils. À vous de vous adapter.

Factory Method et Abstract Factory

Factory Method

Encapsule la création d'objets et permet à une classe de créer un objet en déléguant ce travail à ses sous-classes.

Abstract Factory

Encapsule la création d'objets et permet à une classe de créer une famille d'objets.

Factory Method vs. Abstract Factory

- Un objet **vs.** une famille d'objets.
- Délégation aux sous-classes **vs.** création en passant par de la composition.

Problème de notre code avec Factory Method

Problème

On doit écrire une classe concrète pour chaque type de produit de chaque fabrique !

Dans notre exemple

Classes `VegetarienneStyleNantes`, `RoyaleStyleNantes`, `VegetarienneStyleTokyo`, ...

Dommage

C'est d'autant plus bête que ces produits sont tous plus ou moins composés des mêmes choses (une pâte, une sauce, ...).

Refonte de la Pizza

Une famille d'objets

Famille : ensemble d'objets dont la fabrication est similaire, partageant les mêmes sous-éléments.

Dans notre exemple

Considérons la pizzeria de Nantes. Il est facile d'imaginer qu'elle va utiliser le même type de sauce tomate, de pâte, etc, pour chacune de ses pizzas.

Donc les pizzas de la pizzeria de Nantes formeront une famille.

Nouvelle classe Pizza

```
abstract class Pizza
{
   private String nom;
   private Pate pate;
   private Sauce sauce;
   private Fromage fromage;
   ...

   public abstract void prepare();

   ...
}
```

Classe CreerIngredients

Flexibilité

On souhaite pouvoir instancier une liste d'ingrédients pour nos pizzas, en laissant la possibilité de changer facilement un ingrédient (la pâte par exemple).

Classe CreerIngredients

```
interface CreerIngredients
{
  public Pate creerPate();
  public Sauce creerSauce();
  public Fromage creerFromage();
  ...
}
```

Fabrique concrète d'ingrédients

Classe IngredientsNantes

```
class IngredientsNantes implements CreerIngredients
{
  public Pate creerPate() {
    return new PateFine();
  }

  public Sauce creerSauce() {
    return new SauceTomatePiquante();
  }

  public Sauce creerFromage() {
    return new Mozzarella();
  }

  ...
}
```

Pizza générique aux fruits de mer

Classe PizzaFruitsDeMer

```
class PizzaFruitsDeMer extends Pizza
{
  private CreerIngredients ci_;

  public PizzaFruitsDeMer( CreerIngredients ci )
  {
    ci_ = ci;
  }

  public void prepare()
  {
    pate      = ci_.creerPate();
    sauce     = ci_.creerSauce();
    crevette  = ci_.creerCrevette();
    moule     = ci_.creerMoule();
    palourde  = ci_.creerPalourde();
  }
}
```

Créer des pizzas aux fruits de mer

Passer d'une pizza aux fruits de mer à l'autre

```
Pizza pizza;
CreerIngredients ci;

// pizza nantaise aux fruits de mer
ci = new IngredientsNantes();
pizza = new PizzaFruitsDeMer( ci );

// pizza parisienne aux fruits de mer
ci = new IngredientsParis();
pizza = new PizzaFruitsDeMer( ci );
```

Pizzas de la pizzeria de Nantes

Nouvelle classe PizzeriaNantes

```
class PizzeriaNantes extends Pizzeria {
  private CreerIngredients ci_;
  private Pizza pizza_;

  public PizzeriaNantes() {
    ci_ = new IngredientsNantes();
  }

  public Pizza commander( String type ) {
    if( type.equals( "Vegetarienne" ) )
      pizza_ = new PizzaVegetarienne( ci_ );
    else if( type.equals( "Royale" ) )
      pizza_ = new PizzaRoyale( ci_ );
    ...

    pizza.preparer();
    pizza.cuire();
    ...
  }
}
```

Notez bien !

Remarque importante

Tout comme pour la Factory Method, l'interface de création d'une Abstract Factory ne dépend pas des classes de bas niveaux.

Classe CreerIngredients

```
interface CreerIngredients
{
  public Pate creerPate();
  public Sauce creerSauce();
  public Fromage creerFromage();
  ...
}
```

Notez bien 2 !

Famille d'objets

Par définition de l'Abstract Factory : toutes les pizzas d'une même pizzeria doivent partager les mêmes types d'ingrédients (un même type de sauce, un même type de pâte, ...).

Familles isomorphiques

Par contre, les familles doivent être isomorphiques (de même forme). Une pizza aux fruits de mer doit contenir des crevettes, qu'elle soit de la famille de Nantes, New York ou Tokyo !

Bon choix ?

Si cela pose problème pour votre programme, Abstract Factory n'était pas le bon choix.

Classes concrètes des ingrédients

Potentiellement beaucoup de classes concrètes

Une classe concrète par ingrédients ! Même problème qu'avec Factory Method ?

Par exemple pour les sauces

Classes `SauceTomate`, `SauceTomatePiquante`, `CremeFraiche`, `SauceMarinara`, ...

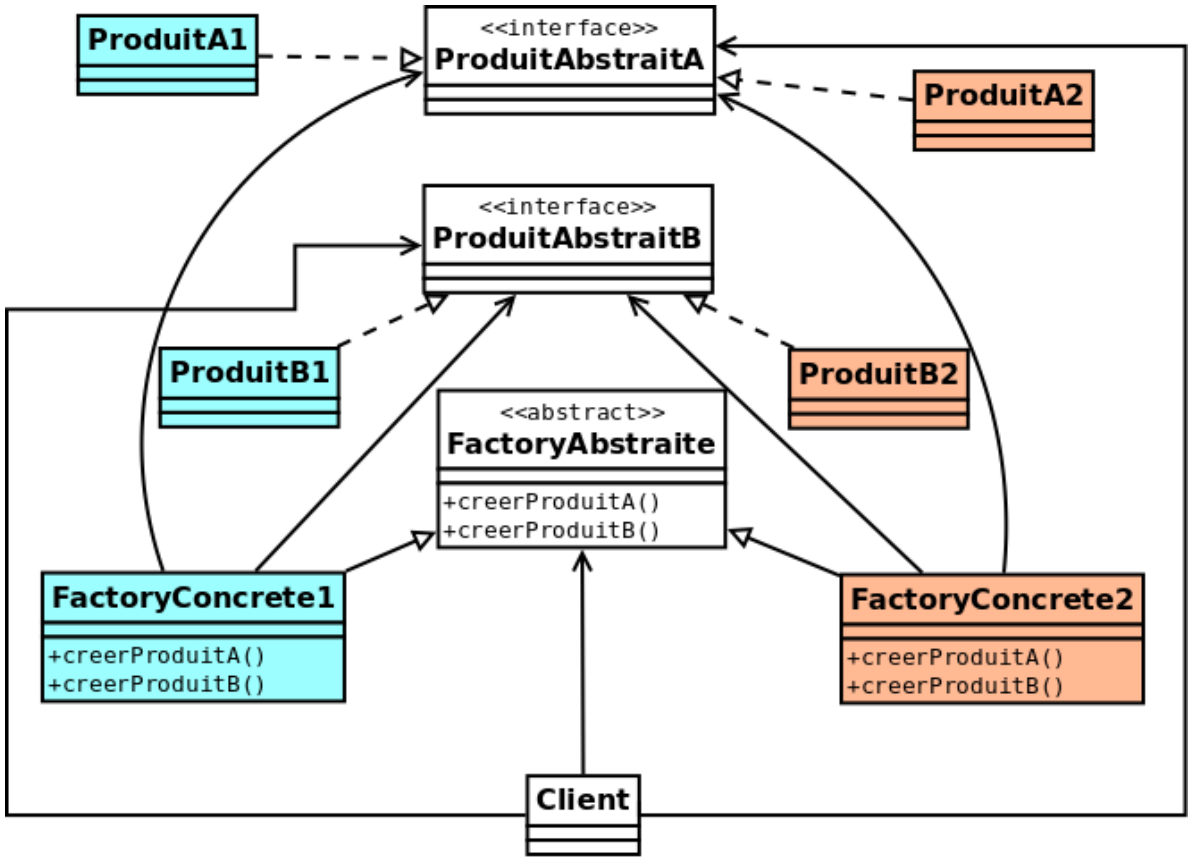
Réutilisation !

Cependant on peut réutiliser ces objets si des pizzas partagent certains ingrédients identiques.

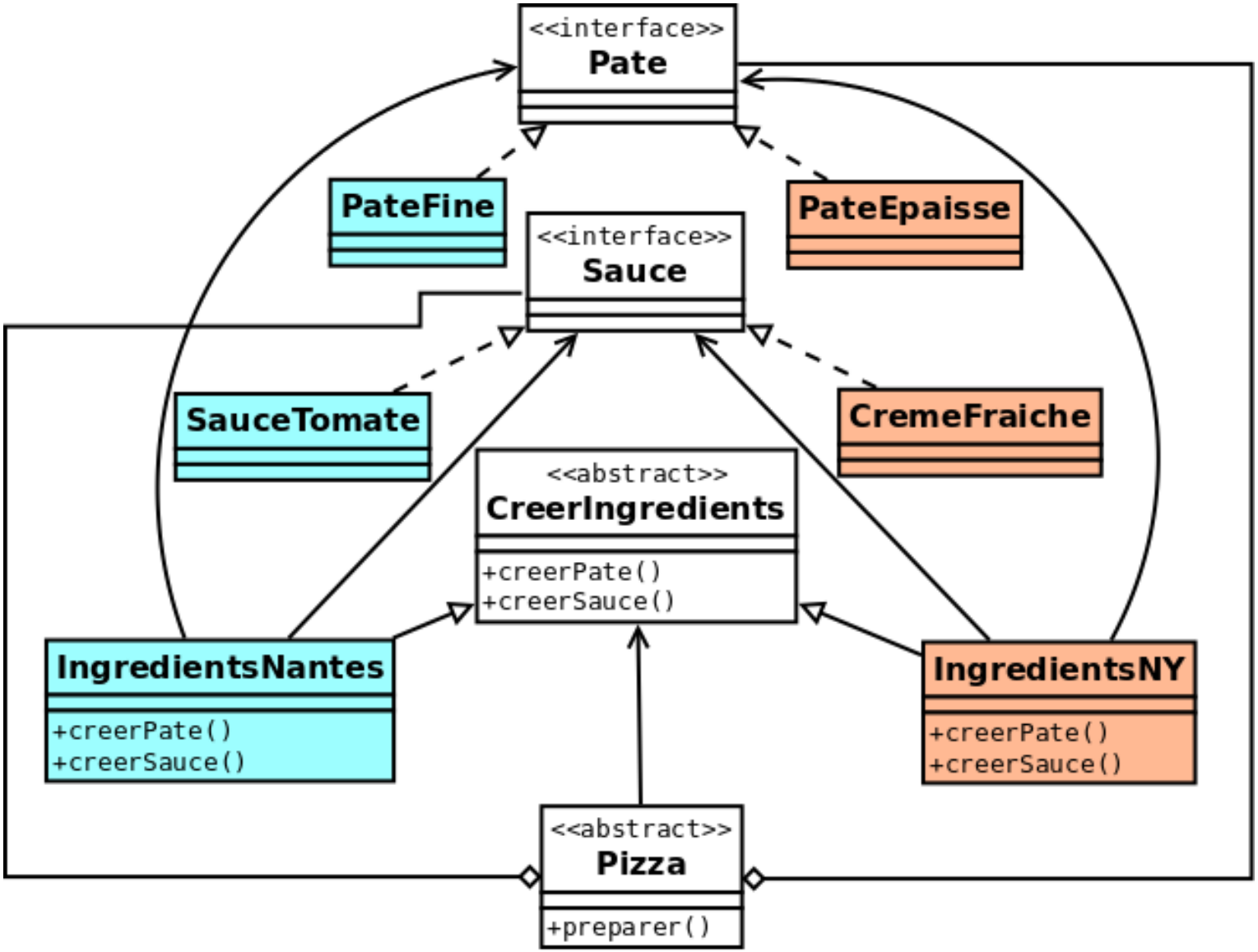
Pattern Abstract Factory

Définition

Le pattern Abstract Factory (Fabrique abstraite) fournit une interface pour créer des familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes.



Pattern Abstract Factory



Pattern Abstract Factory

Le nom

Pourquoi Abstract Factory s'appelle Abstract Factory ?

Pattern Abstract Factory

Le nom

Pourquoi Abstract Factory s'appelle Abstract Factory ?

L'implémentation

L'Abstract Factory est en quelque sorte plusieurs Factory Methods placées dans une classe abstraite (ou une interface). Voir CreerIngredients.

Classe CreerIngredients

```
interface CreerIngredients // classe ''abstraite''
{
  public Pate creerPate(); // factory method pour
                           // creer l'objet pate

  public Sauce creerSauce(); // factory method pour
                             // creer l'objet sauce

  public Fromage creerFromage(); //factory method
                                 // pour le fromage

  ...
}
```

Résumé

À retenir sur les deux patterns

- Les deux patterns Factory encapsulent la création d'objets.
- Les deux patterns Factory découplent les classes de haut niveau des classes de bas niveau.
- Factory Method est une sorte de constructeur virtuel pour créer un type d'objet.
- Factory Method se base sur l'héritage pour déléguer la création d'objets aux sous-classes.
- Abstract Factory permet d'instancier des familles d'objets.
- Abstract Factory se base sur la composition.

Pattern de création

Pattern de création

Factory Method et Abstract Factory sont des patterns de création. Ils sont là pour découpler des classes d'un programme la création d'objets.

La prochaine fois

On verra un autre pattern de création, **Singleton**. Il sera le dernier pattern que l'on étudiera en détail.