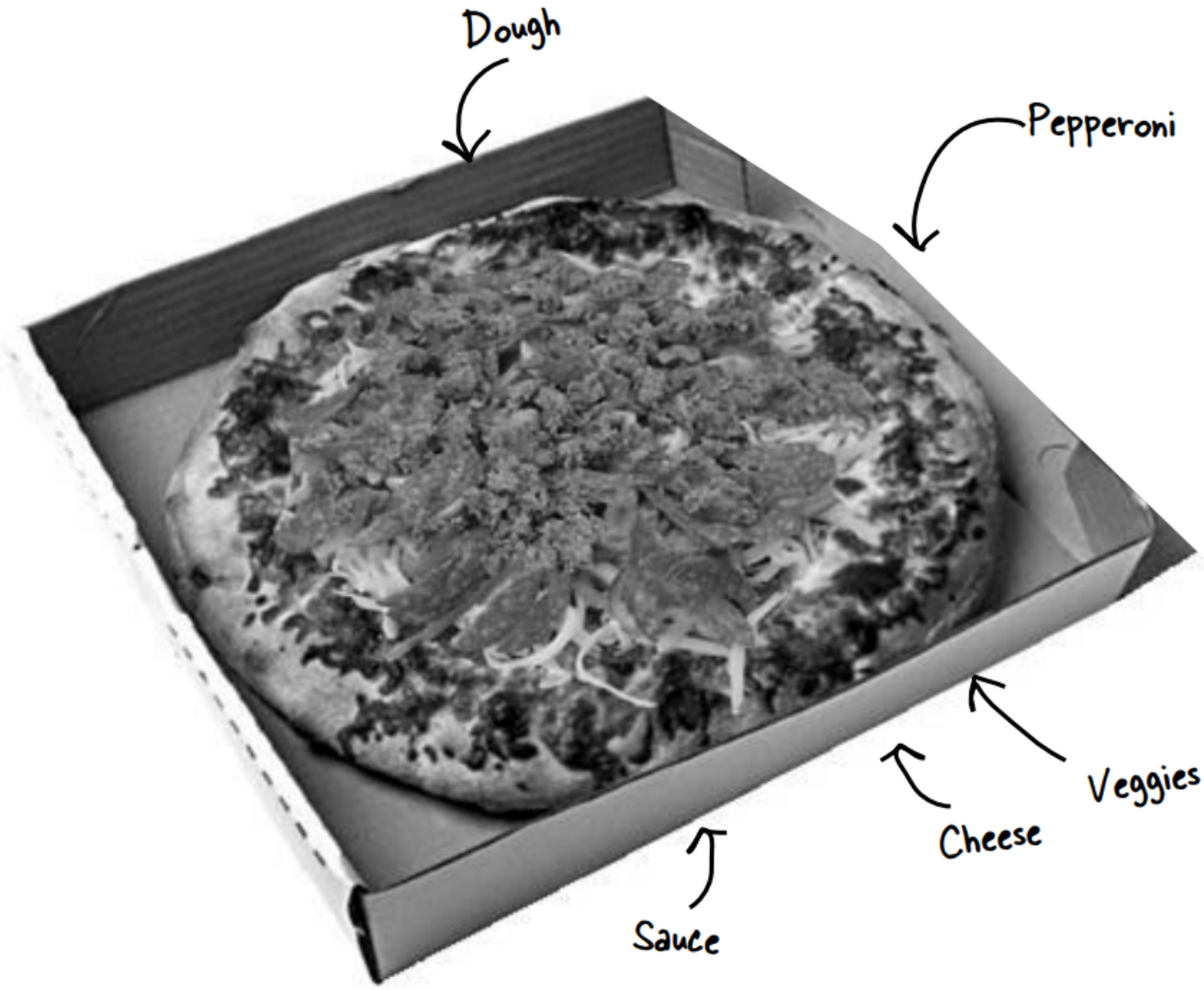
Chương 4: Factory Pattern & Abstract Factory Pattern (Phần 2)

Trong khi đó, trở lại PizzaStore ...

Thiết kế cho PizzaStore đang thực sự định hình: nó có một framework linh hoạt và nó thực hiện tốt việc tuân thủ các nguyên tắc thiết kế.



Giờ đây, chìa khóa thành công của Objectville Pizza luôn là những nguyên liệu tươi ngon, chất lượng và điều bạn đã khám phá là với khuôn khổ mới, những nơi được nhượng quyền của bạn đã tuân theo quy trình của bạn, nhưng một vài nhượng quyền đã thay thế các thành phần kém hơn trong bánh nướng của họ để hạ thấp chi phí và tăng tỷ suất lợi nhuận của họ. Bạn biết rằng bạn sẽ phải làm một cái gì đó, bởi vì về lâu dài điều này sẽ làm ảnh hưởng thương hiệu Objectville!

ĐẢM BẢO TÍNH NHẤT QUÁN TRONG THÀNH PHẦN CỦA PIZZA

Vì vậy, làm thế nào để bạn đảm bảo mỗi nhượng quyền đang sử dụng các thành phần chất lượng hay không? Bạn sẽ xây dựng một factory sản xuất chúng và chuyển chúng đến nhượng quyền của bạn!

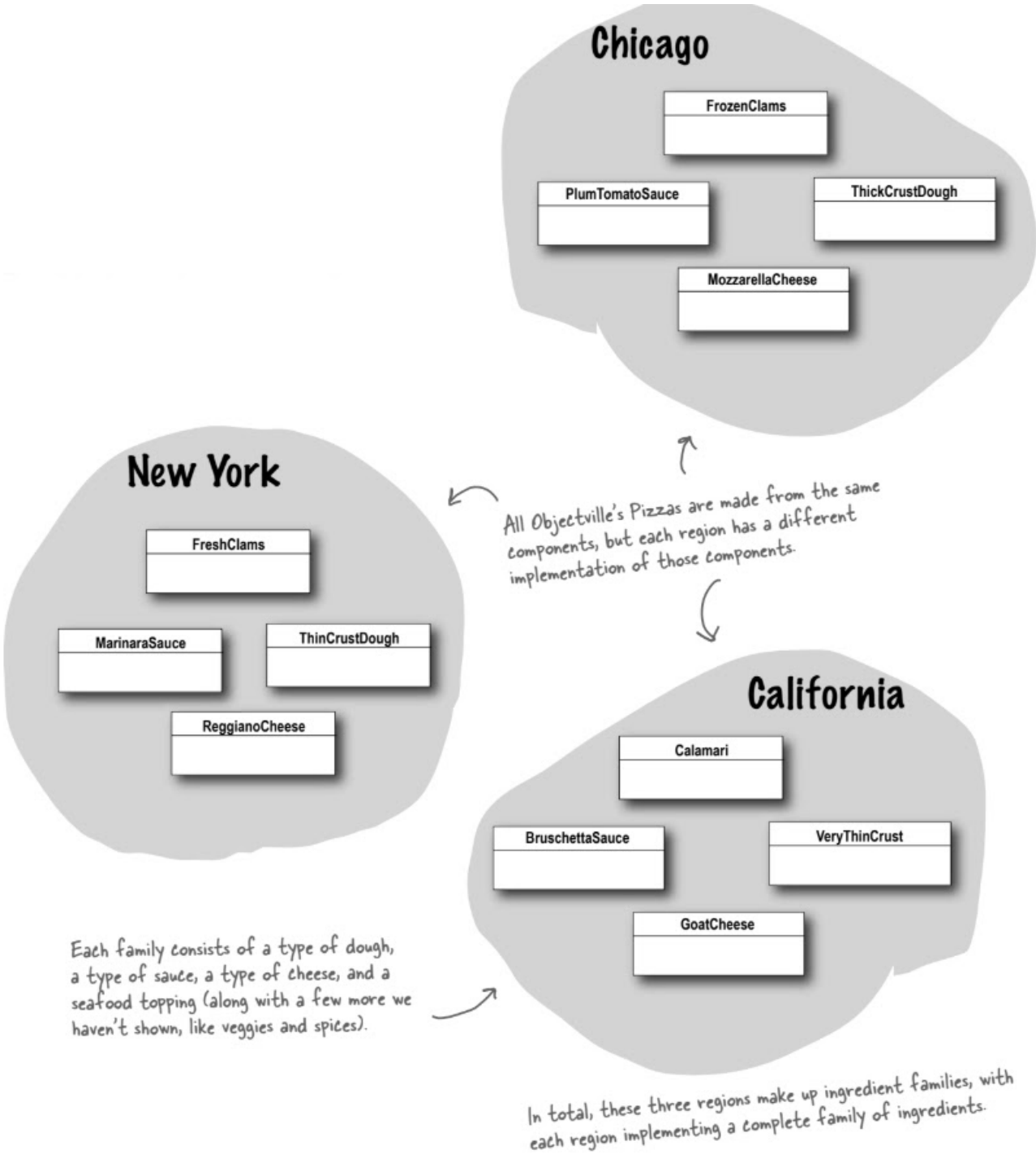
Bây giờ chỉ có một vấn đề với kế hoạch này: nhượng quyền thương mại được đặt ở các khu vực khác nhau và nước sốt đỏ ở New York không phải là nước sốt đỏ ở Chicago.

Vì vậy, bạn có một bộ nguyên liệu cần được chuyển đến New York và một bộ nguyên liệu khác cần chuyển đến Chicago. Hãy để xem xét kỹ hơn:

Bộ nguyên liệu …

New York sử dụng một bộ nguyên liệu và Chicago là một bộ khác. Với sự phổ biến của Objectville Pizza, sẽ không lâu nữa bạn cũng cần phải chuyển một bộ nguyên liệu đến những khu vực khác: California, Seattle…

Để làm việc này, bạn sẽ phải tìm ra cách xử lý "bộ nguyên liệu".



Xây dựng nhà máy nguyên liệu

Bây giờ chúng ta sẽ xây dựng một nhà máy để tạo ra các nguyên liệu của chúng ta; nhà máy sẽ chịu trách nhiệm tạo ra từng nguyên liệu trong "bộ tnguyên liệu". Nói cách khác, nhà máy sẽ cần tạo ra bột, nước sốt, phô mai, v.v … Bạn sẽ thấy cách chúng ta sẽ xử lý sự khác biệt trong từng khu vực ngay sau đây.

Hãy bắt đầu bằng cách xác định interface cho nhà máy sẽ tạo ra tất cả các nguyên liệu cho chúng ta:

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}
```

For each ingredient we define a create method in our interface.

Lots of new classes here, one per ingredient.

If we'd had some common "machinery" to implement in each instance of factory, we could have made this an abstract class instead...

Đây là những gì chúng ta sẽ làm:

1. Xây dựng nhà máy cho từng vùng. Để làm điều này, bạn sẽ tạo một lớp con của PizzaIngredientFactory và implement từng phương thức tạo.
2. Implement một tập hợp các lớp nguyên liệu sẽ được sử dụng với nhà máy, như ReggianoCheese, RedPeppers và ThickCrustDough. Các lớp này có thể được chia sẻ giữa các khu vực khi thích hợp.
3. Sau đó, chúng ta cần kết nối tất cả những điều này bằng cách đưa các nhà máy sản xuất nguyên liệu mới vào PizzaStore.

Xây dựng nhà máy sản xuất nguyên liệu ở New York

Được rồi, ở đây, việc triển khai cho nhà máy sản xuất nguyên liệu ở New York. Nhà máy này chuyên về sốt Marinara, phô mai Reggiano, Fresh Clams…

The NY ingredient factory implements the interface for all ingredient factories

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();
    }

    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    public Cheese createCheese() {
        return new ReggianoCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }
    public Clams createClam() {
        return new FreshClams();
    }
}
```

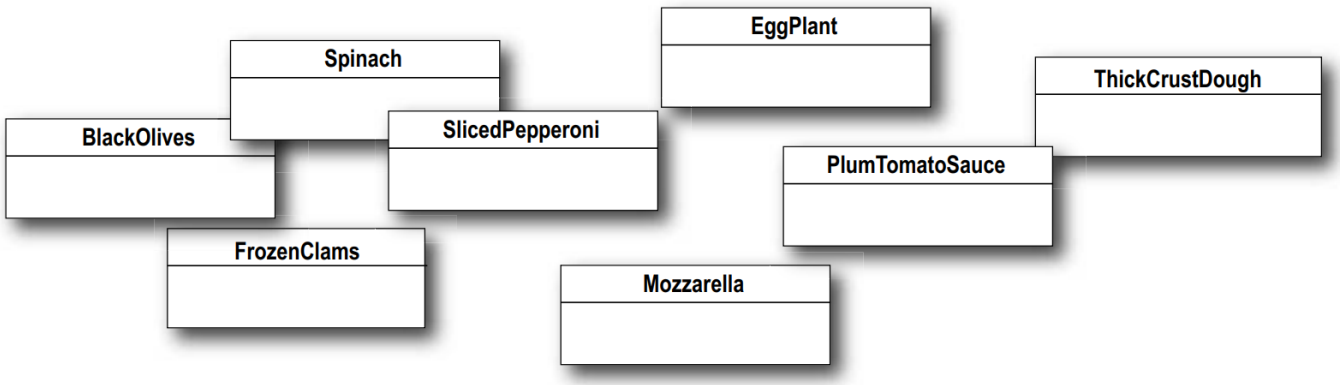
For each ingredient in the ingredient family, we create the New York version.

For veggies, we return an array of Veggies. Here we've hardcoded the veggies. We could make this more sophisticated, but that doesn't really add anything to learning the factory pattern, so we'll keep it simple.

New York is on the coast; it gets fresh clams. Chicago has to settle for frozen.

The best sliced pepperoni. This is shared between New York and Chicago. Make sure you use it on the next page when you get to implement the Chicago factory yourself

Bạn hãy tự viết cho lớp **ChicagoPizzaIngredientFactory**. Bạn có thể tham khảo các lớp dưới đây trong việc implement của bạn:



Làm lại lớp Pizza ...

Chúng tôi đã cho tất cả các nhà máy kém chất lượng của chúng tôi bị sa thải và sẵn sàng sản xuất các nguyên liệu chất lượng; bây giờ chúng tôi chỉ cần làm lại lớp Pizza để chúng chỉ sử dụng các nguyên liệu sản xuất tại nhà máy. Chúng tôi sẽ bắt đầu với lớp abstract Pizza của mình:

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

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

    String getName() {
        return name;
    }

    public String toString() {
        // code to print pizza here
    }
}
```

Each pizza holds a set of ingredients that are used in its preparation.

We've now made the prepare method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory.

Our other methods remain the same, with the exception of the prepare method.

Tiếp tục làm lại lớp Pizza …

Bây giờ bạn đã có một lớp abstract Pizza để làm việc, và bây giờ sẽ tạo ra những chiếc bánh pizza kiểu New York và Chicago – lúc này, chúng sẽ lấy nguyên liệu từ nhà máy.

Khi chúng tôi viết Factory Method code ở Phần 1 (https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-1.html), chúng tôi đã có một lớp **NYCheesePizza** và một lớp **ChicagoCheesePizza**. Nếu bạn nhìn vào hai lớp này, điều duy nhất khác biệt là việc sử dụng các nguyên liệu trong khu vực. Các loại pizza được làm giống nhau (bột + nước sốt + phô mai). Điều tương tự cũng xảy ra với các loại pizza khác: **Veggie**, **Clam**,… Tất cả đều theo các bước chuẩn bị giống nhau; chúng chỉ có các nguyên liệu khác nhau.

Vì vậy, những gì bạn có thể thấy là chúng tôi thực sự không cần hai lớp cho mỗi chiếc bánh pizza; nhà máy sản xuất nguyên liệu sẽ xử lý sự khác biệt trong khu vực cho chúng tôi.

Ở đây, **Cheese Pizza**:

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

To make a pizza now, we need a factory to provide the ingredients. So each Pizza class gets a factory passed into its constructor, and it's stored in an instance variable.

Here's where the magic happens!

The prepare() method steps through creating a cheese pizza, and each time it needs an ingredient, it asks the factory to produce it.

Lớp Pizza sử dụng factory mà nó đã được kết hợp để sản xuất các nguyên liệu được sử dụng trong pizza. Các nguyên liệu được sản xuất phụ thuộc vào nhà máy mà chúng tôi sử dụng. Lớp Pizza không quan tâm; Nó biết cách để làm pizza. Giờ đây, nó đã tách ra khỏi sự khác biệt về thành phần trong khu vực và có thể dễ dàng tái sử dụng khi có các nhà máy ở Rockies, Tây Bắc Thái Bình Dương và những vùng khác.

```
sauce = ingredientFactory.createSauce();
```

We're setting the Pizza instance variable to refer to the specific sauce used in this pizza.

This is our ingredient factory. The Pizza doesn't care which factory is used, as long as it is an ingredient factory.

The createSauce() method returns the sauce that is used in its region. If this is a NY ingredient factory, then we get marinara sauce.

Bình luận của người dịch:

Hãy nhớ lại lúc đầu, các nguyên liệu được tạo ra trong chính lớp Pizza, các lớp con kế thừa lớp Pizza và tùy ý gán giá trị cho các biến nguyên liệu.

Vấn đề đặt ra là lớp cha Pizza muốn quản lý quá trình tạo nguyên liệu luôn (không thể để các lớp con gán tùy ý), vì vậy bước tạo nguyên liệu được tách ra thành 1 factory **PizzaIngredientFactory** và khai báo tham chiếu đến interface này trong lớp cha Pizza. Chúng sẽ đảm bảo được rằng, các lớp con tạo nguyên liệu bằng cách dùng factory mà lớp cha Pizza cung cấp.

Hãy cùng khám phá ClamPizza:

```
public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public ClamPizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

ClamPizza also stashes an ingredient factory.

To make a clam pizza, the prepare method collects the right ingredients from its local factory.

If it's a New York factory, the clams will be fresh; if it's Chicago, they'll be frozen.

Xem lại các cửa hàng pizza của chúng ta

Chúng ta gần đến nơi rồi; chúng ta chỉ cần thực hiện một chuyến đi nhanh đến các cửa hàng nhượng quyền để đảm bảo rằng họ đang sử dụng đúng lớp Pizza. Chúng ta cũng cần cung cấp cho họ một tham chiếu (reference) đến các nhà máy sản xuất nguyên liệu của địa phương của họ (Chicago sẽ có **ChicagoPizzaIngredientFactory**, NY sẽ có **NYPizzaIngredientFactory**…):

```
public class NYPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NYPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("New York Style Cheese Pizza");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("New York Style Veggie Pizza");

        } else if (item.equals("clam")) {

            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("New York Style Clam Pizza");

        } else if (item.equals("pepperoni")) {
            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("New York Style Pepperoni Pizza");

        }
        return pizza;
    }
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas.

We now pass each pizza the factory that should be used to produce its ingredients.

Look back one page and make sure you understand how the pizza and the factory work together!

For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.

Sử dụng sức mạnh bộ não

So sánh phương thức ***createPizza()*** của phiên bản này với phương thức trong triển khai **Factory Method** trước đó trong Phần 1 (https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-1.html).

Chúng ta đã làm được gì?

Đó là một loạt các thay đổi về code; Chính xác thì chúng ta đã làm gì?

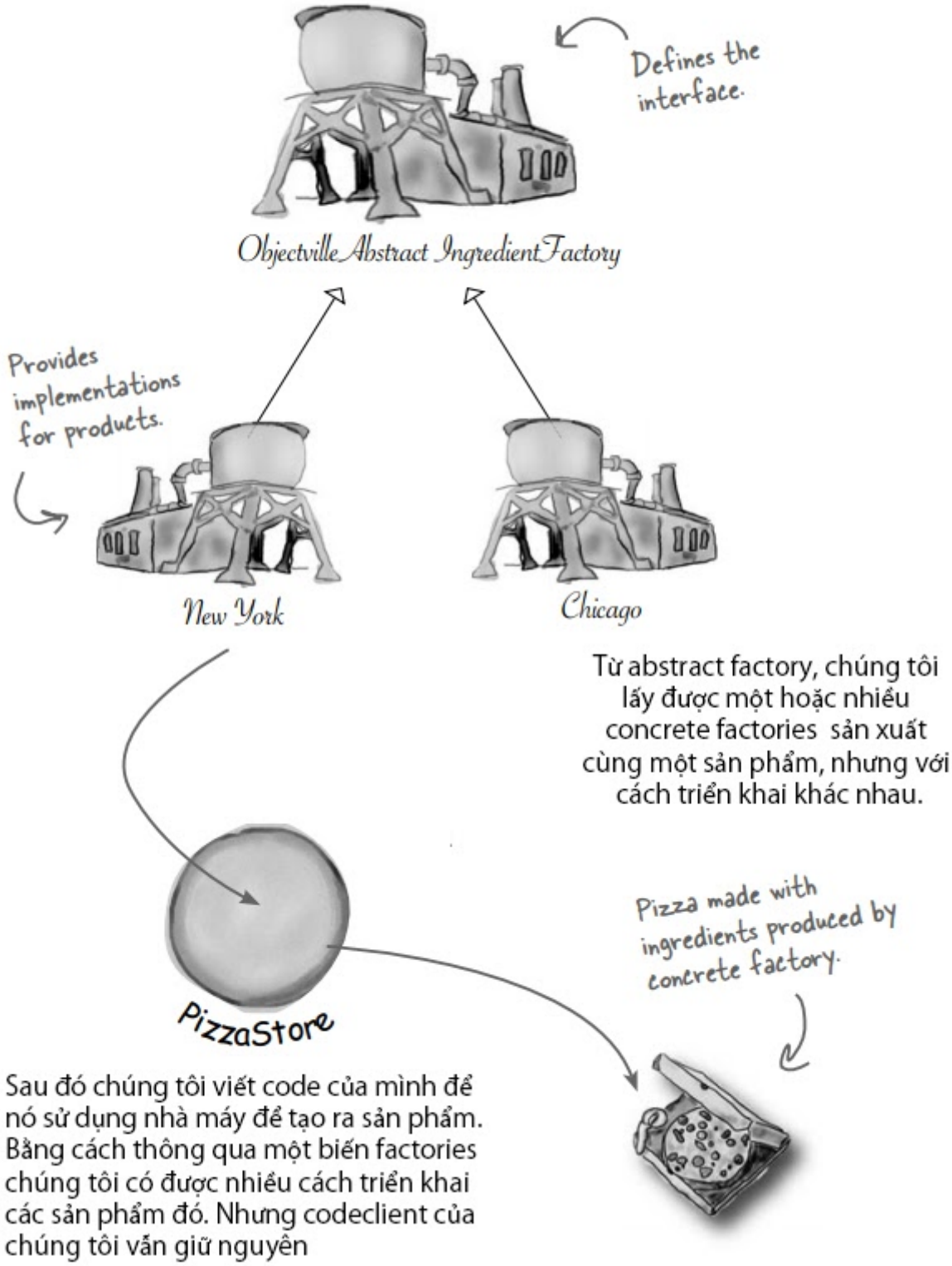
Chúng ta đã cung cấp một phương tiện để tạo ra một "bộ nguyên liệu" cho pizza bằng cách giới thiệu một loại nhà máy mới gọi là **Abstract Factory** – sử dụng **Abstract Factory Pattern (https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-1.html)**.

Một **Abstract Factory Pattern (https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-1.html)** cung cấp cho chúng ta một giao diện để tạo một "bộ sản phẩm". Bằng cách viết code sử dụng giao diện này, chúng ta tách code khỏi factory thực tế tạo ra các sản phẩm.

Điều đó cho phép chúng ta triển khai nhiều nhà máy sản xuất các sản phẩm dành cho các bối cảnh khác nhau – chẳng hạn như các khu vực khác nhau, hệ điều hành khác nhau hoặc giao diện khác nhau.

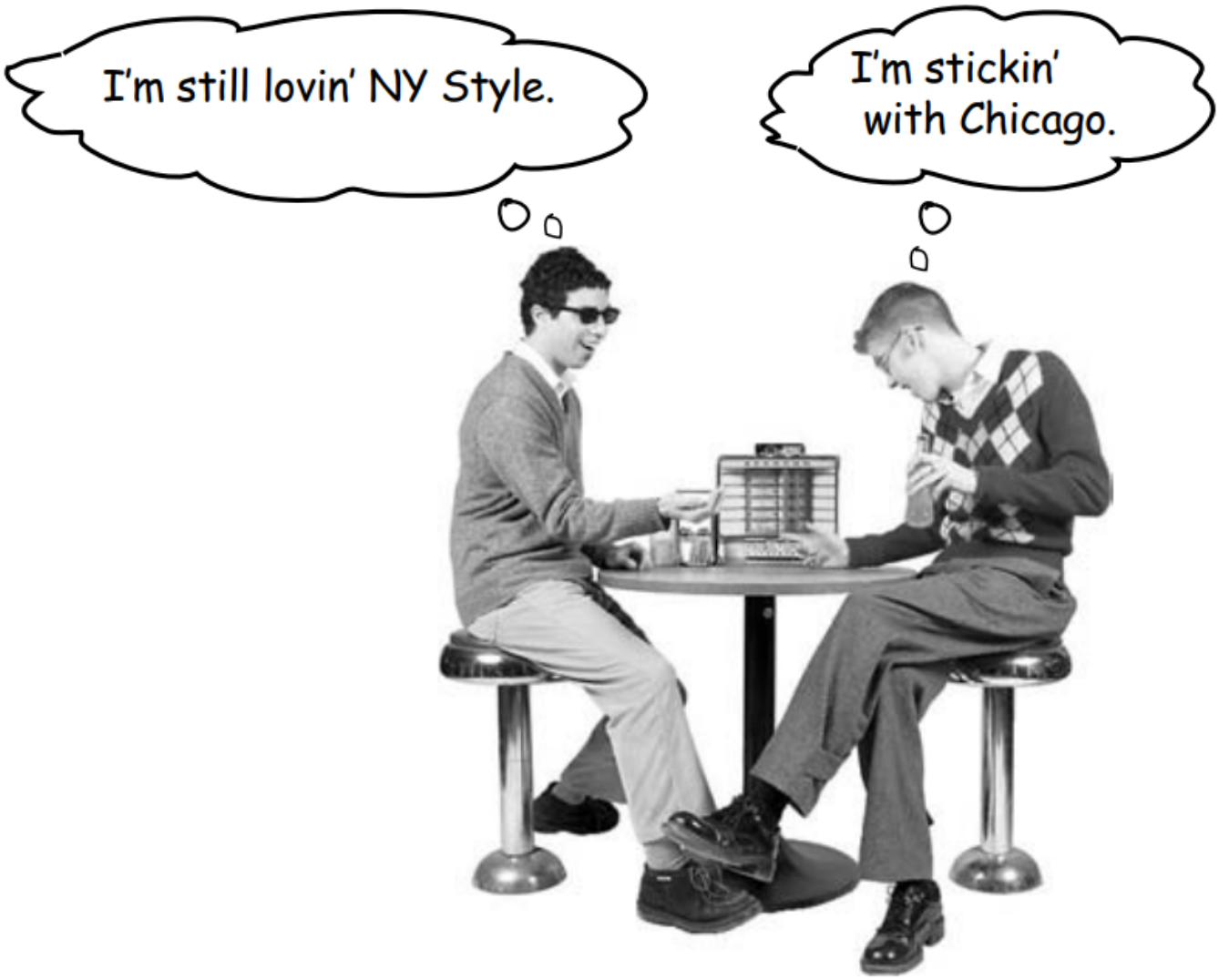
Vì code của chúng ta được tách rời khỏi các sản phẩm thực tế, chúng ta có thể thay thế các nhà máy khác nhau để có các hành vi khác nhau (như lấy marinara thay vì plum tomatoes).

Một Abstract Factory cung cấp một giao diện cho một bộ sản phẩm. Một bộ là gì? Trong trường hợp của chúng tôi, nó có tất cả những thứ chúng ta cần để làm một chiếc bánh pizza: bột, nước sốt, phô mai, thịt và rau.



Nhiều pizza hơn cho Ethan và Joel...

Ethan và Joel không thể có đủ Objectville Pizza! Điều họ không biết là bây giờ các đơn hàng của họ đang sử dụng các nhà máy sản xuất nguyên liệu mới. Vì vậy, bây giờ khi họ đặt hàng …



Phần đầu tiên của quy trình đặt hàng đã không thay đổi chút nào. Hãy để theo dõi đơn hàng của Ethan lần nữa:

1. Đầu tiên chúng ta cần một NY PizzaStore:

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Creates an instance of NYPizzaStore.

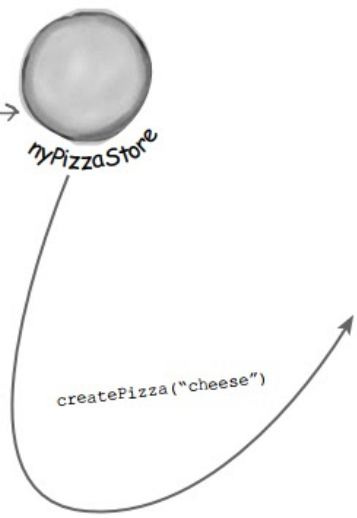
2. Bây giờ chúng tôi có một cửa hàng, chúng tôi có thể đặt hàng:

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

the orderPizza() method is called on the nyPizzaStore instance.

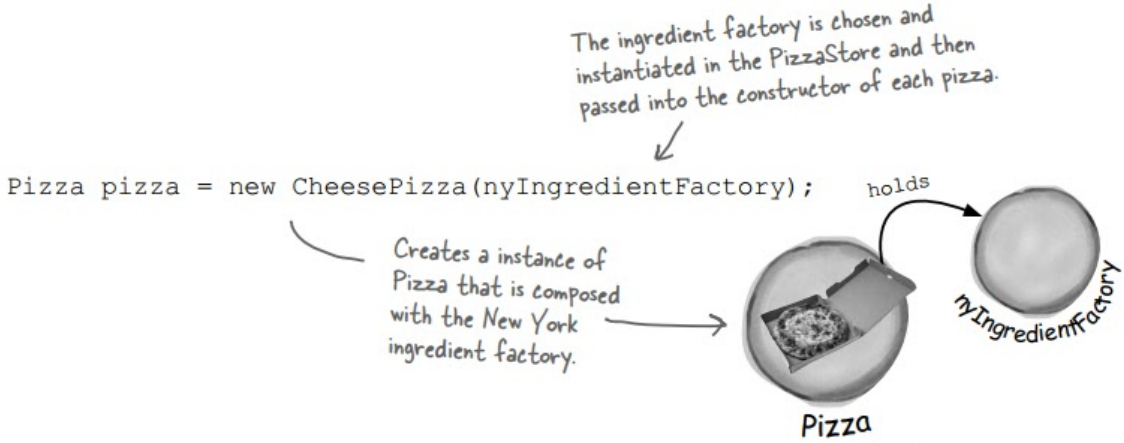
3. Phương thức orderPizza() trước tiên gọi phương thức createPizza ():

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

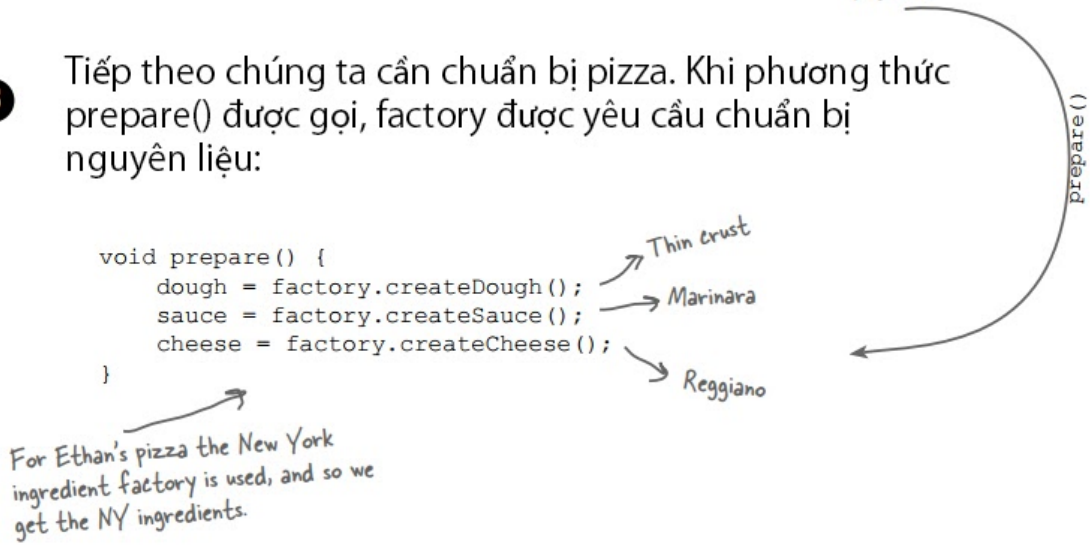


Từ đây mọi thứ thay đổi, bởi vì chúng tôi đang sử dụng một nhà máy sản xuất nguyên liệu

4. Khi phương thức createPizza () được gọi, đó là khi nhà máy sản xuất nguyên liệu của chúng tôi tham gia:



5. Tiếp theo chúng ta cần chuẩn bị pizza. Khi phương thức prepare() được gọi, factory được yêu cầu chuẩn bị nguyên liệu:



6. Cuối cùng, chúng tôi có bánh pizza đã được chuẩn bị trong tay và phương thức orderPizza () sẽ nướng, cắt và đóng hộp bánh pizza.

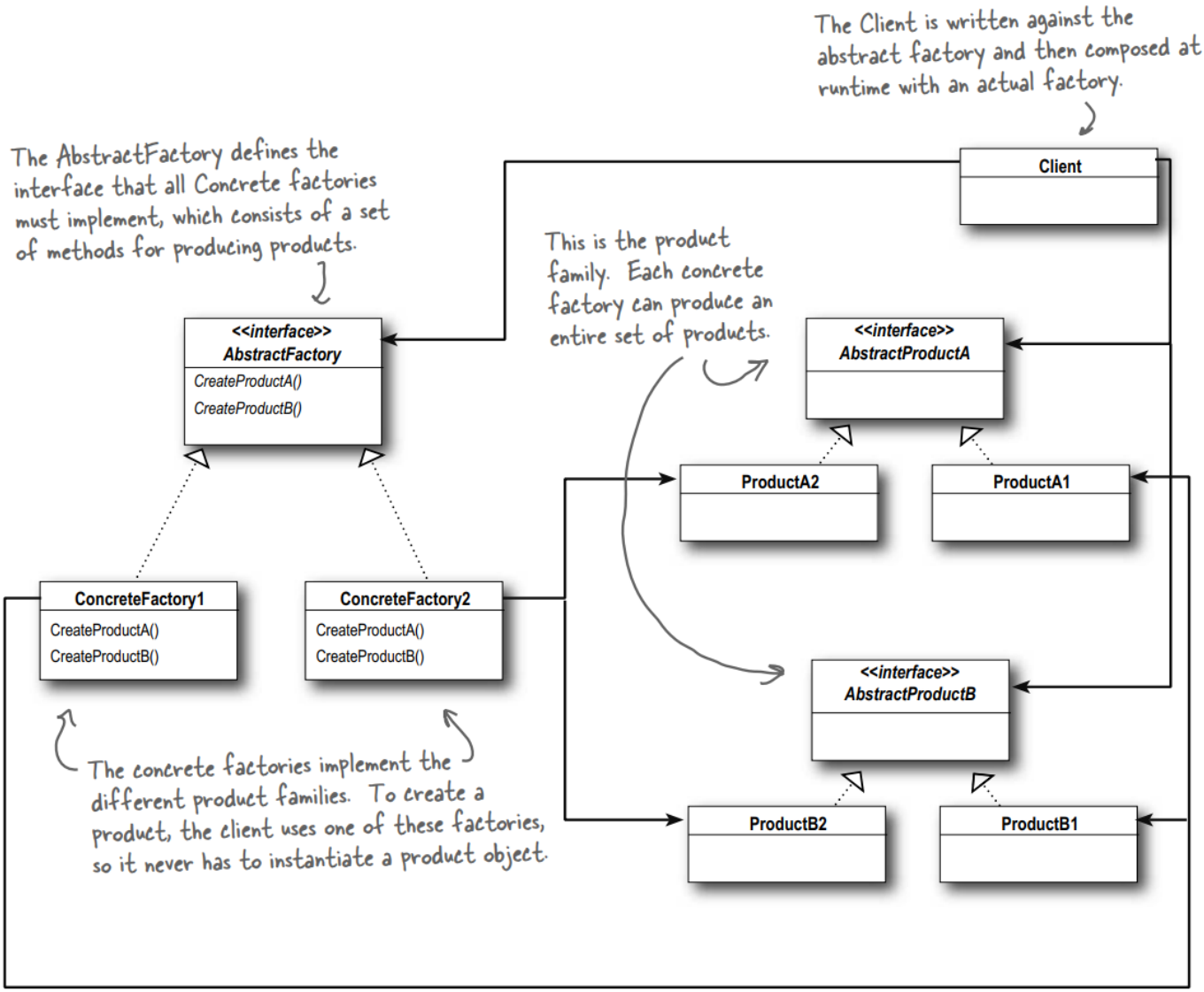
Định nghĩa Abstract Factory Pattern (https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-1.html)

Chúng tôi đã thêm một Factory Pattern (https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-1.html) khác vào "gia đình mẫu" của chúng tôi, một mô hình cho phép chúng tôi tạo ra các bộ sản phẩm. Hãy cùng kiểm tra định nghĩa chính thức cho mẫu này:

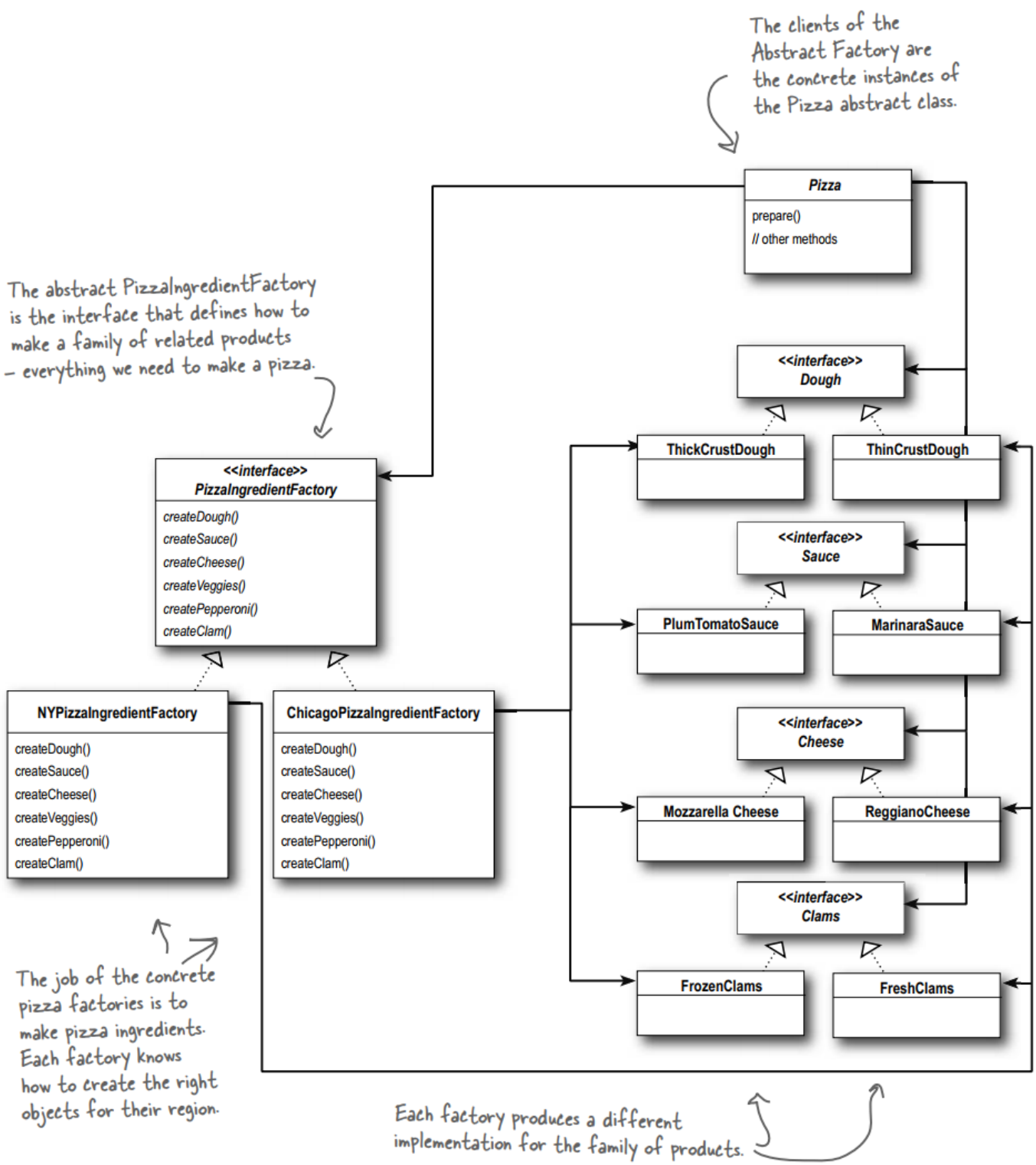
The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

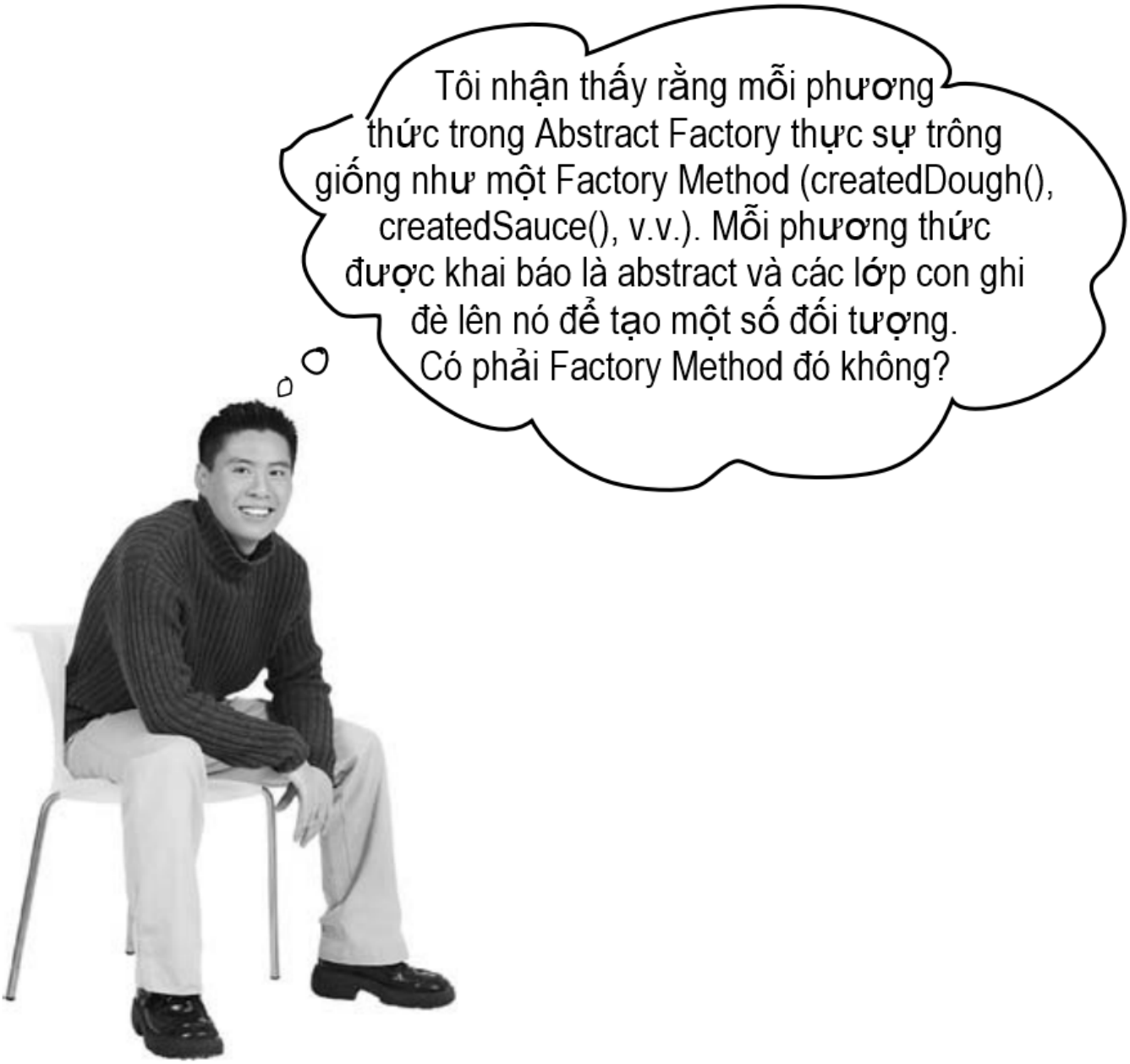
(Abstract Factory Pattern (https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-1.html) cung cấp một interface có chức năng tạo ra một tập hợp (https://vi.wikipedia.org/wiki/T%E1%BA%ADp_h%E1%BB%A3p) ***các đối tượng liên quan hoặc phụ thuộc lẫn nhau mà không chỉ ra đó là những lớp cụ thể nào tại thời điểm thiết kế.)***

Chúng tôi chắc chắn đã thấy rằng **Abstract Factory Pattern (https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-1.html)** cho phép client sử dụng giao diện trừu tượng để tạo ra một bộ sản phẩm liên quan mà không cần biết (hoặc quan tâm) về các sản phẩm cụ thể được tạo ra thực sự. Theo cách này, client được tách ra khỏi bất kỳ chi tiết cụ thể nào của concrete products. Chúng ta hãy nhìn vào sơ đồ lớp để xem tất cả những thứ này kết hợp với nhau như thế nào:



Đó là một sơ đồ lớp khá phức tạp; hãy cùng xem xét về khía cạnh **PizzaStore** của chúng tôi:


The clients of the Abstract Factory are the concrete instances of the Pizza abstract class.
Pizza
prepare()
// other methods
The abstract PizzaIngredientFactory is the interface that defines how to make a family of related products – everything we need to make a pizza.
<<interface>>
PizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
<<interface>>
Dough
ThickCrustDough
ThinCrustDough
<<interface>>
Sauce
PlumTomatoSauce
MarinaraSauce
<<interface>>
Cheese
Mozzarella Cheese
ReggianoCheese
<<interface>>
Clams
FrozenClams
FreshClams
NYPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
ChicagoPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
The job of the concrete pizza factories is to make pizza ingredients. Each factory knows how to create the right objects for their region.
Each factory produces a different implementation for the family of products.



Có phải đó là một Factory Method ẩn giấu bên trong Abstract Factory?

Nắm bắt tốt! Có, thường thì các phương thức của một Abstract Factory được triển khai như các factory methods. Nó có ý nghĩa, phải không? Công việc của một Abstract Factory là xác định một interface để tạo ra một bộ sản phẩm. Mỗi phương thức trong interface đó chịu trách nhiệm tạo ra một sản phẩm cụ thể và chúng tôi triển khai một lớp con của Abstract Factory để cung cấp các triển khai đó. Vì vậy, factory methods là một cách tự nhiên để implement product methods trong abstract factory của bạn (ý đoạn này khi Abstract Factory tạo ra được 1 product, product đó chính là 1 factory con) .

Buổi phỏng vấn của Factory Method và Abstract Factory Pattern

HeadFirst: Wow, một cuộc phỏng vấn với hai mẫu cùng một lúc! Thật vinh dự cho chúng tôi.

Factory Method: Vâng, tôi không chắc chắn rằng tôi thích được "gộp chung" với Abstract Factory, bạn biết đấy. Chúng tôi đều là mẫu "Factory" không có nghĩa là chúng tôi không có các buổi phỏng vấn của riêng mình.

HeadFirst: Đừng nói vậy, chúng tôi muốn phỏng vấn các bạn cùng nhau để có thể giúp làm sáng tỏ những sự nhầm lẫn về cái nào sẽ là mẫu dành cho độc giả. Các bạn có những điểm tương đồng, và tôi đã nghe nói rằng các bạn khiến mọi người đôi khi bối rối.

Abstract Factory: Đó là sự thật, đã có lần tôi đã bị nhầm với **Factory Method** và tôi biết bạn đã gặp vấn đề tương tự, đúng không **Factory Method**. Cả hai chúng tôi thực sự giỏi trong việc tách các ứng dụng khỏi các triển khai cụ thể; chúng tôi chỉ làm điều đó theo những cách khác nhau. Vì vậy, tôi có thể thấy tại sao mọi người đôi khi có thể khiến chúng ta bối rối.

Factory Method: Vâng, nó làm tôi nổi điên. Rốt cuộc, tôi sử dụng các lớp để tạo và bạn sử dụng các đối tượng; điều đó hoàn toàn khác nhau!

HeadFirst: Bạn có thể giải thích thêm về điều đó, Factory Method?

Factory Method: Chắc chắn rồi. Cả Abstract Factory và tôi đều tạo ra các đối tượng – đó là công việc của chúng tôi. Nhưng tôi làm điều đó thông qua thừa kế …

Abstract Factory: … và tôi thực hiện nó thông qua kết hợp (composition) đối tượng.

Factory Method: Đúng. Vì vậy, điều đó có nghĩa là, để tạo các đối tượng bằng Factory Method, bạn cần extends một lớp và override một phương thức factory.

HeadFirst: Và phương thức factory đó làm những gì?

Factory Method: Tất nhiên nó tạo ra các đối tượng! Ý tôi là, toàn bộ ý tưởng của Mẫu Factory Method là bạn sử dụng một lớp con để tạo ra đối tượng cho bạn. Bằng cách đó, người dùng sẽ chỉ cần biết đến lớp trừu tượng như gia cầm, và các lớp con cụ thể sẽ lo về các kiểu gà, kiểu vịt, kiểu ngan. Vì vậy, nói theo cách khác, tôi giúp chương trình độc lập với các kiểu (type) cụ thể đó.

Abstract Factory: Và tôi cũng làm điều đó, nhưng theo cách khác.

HeadFirst: Tiếp tục đi, Abstract Factory … bạn đã nói gì về kết hợp đối tượng (object composition)?

Abstract Factory: Tôi cung cấp một loại trừu tượng (abstract type) để tạo ra một "bộ sản phẩm" khác. Các lớp con của kiểu trừu tượng xác định cách thức các sản phẩm đó được tạo ra. Để áp dụng được ý tưởng của tôi, bạn phải tạo ra một instance của một trong các lớp con trên (instance này là 1 factory) và đưa nó vào chỗ cần thiết trong code. Vì thế, giống như **Factory Method**, những nơi sử dụng factory của tôi sẽ hoàn toàn độc lập với những produtcts cụ thể.

HeadFirst: Ồ, tôi hiểu rồi, vì vậy một lợi thế khác là bạn nhóm lại một bộ các sản phẩm liên quan.

Abstract Factory: Đúng vậy.

HeadFirst: Điều gì xảy ra nếu người ta cần bổ sung thêm một sản phẩm nữa vô "nhóm các sản phẩm" mà anh có thể tạo ra? Điều đó không yêu cầu thay đổi interface của bạn chứ?

Abstract Factory: Interface của tôi phải thay đổi nếu sản phẩm mới được thêm vào, điều mà tôi biết mọi người không thích làm ….

Factory Method: :)))

Abstract Factory: Bạn đang cười gì, Factory Method?

Factory Method: Ồ, thôi nào, đó là một vấn đề lớn! Thay đổi giao diện của bạn có nghĩa là bạn phải thay đổi giao diện của mọi lớp con (các lớp con ở đây là các factories)! Nghe có vẻ như rất nhiều việc ở đây.

Abstract Factory: Vâng, nhưng tôi cần một "giao diện lớn" vì tôi được sử dụng để tạo toàn bộ "bộ sản phẩm". Bạn chỉ tạo một sản phẩm, vì vậy bạn không thực sự cần một "giao diện lớn", bạn chỉ cần một phương thức duy nhất.

HeadFirst: Abstract Factory, tôi nghe đồn rằng bạn thường sử dụng nhiều hàm factory method theo cách của Factory Method Pattern để tạo các đối tượng bên trong những factories của anh, điều đó có đúng không?

Abstract Factory: Có, tôi thừa nhận điều đó, các factory con của tôi thường thực hiện một factory methods để tạo ra sản phẩm. Trong trường hợp của tôi, chúng được sử dụng thuần túy để tạo ra sản phẩm …

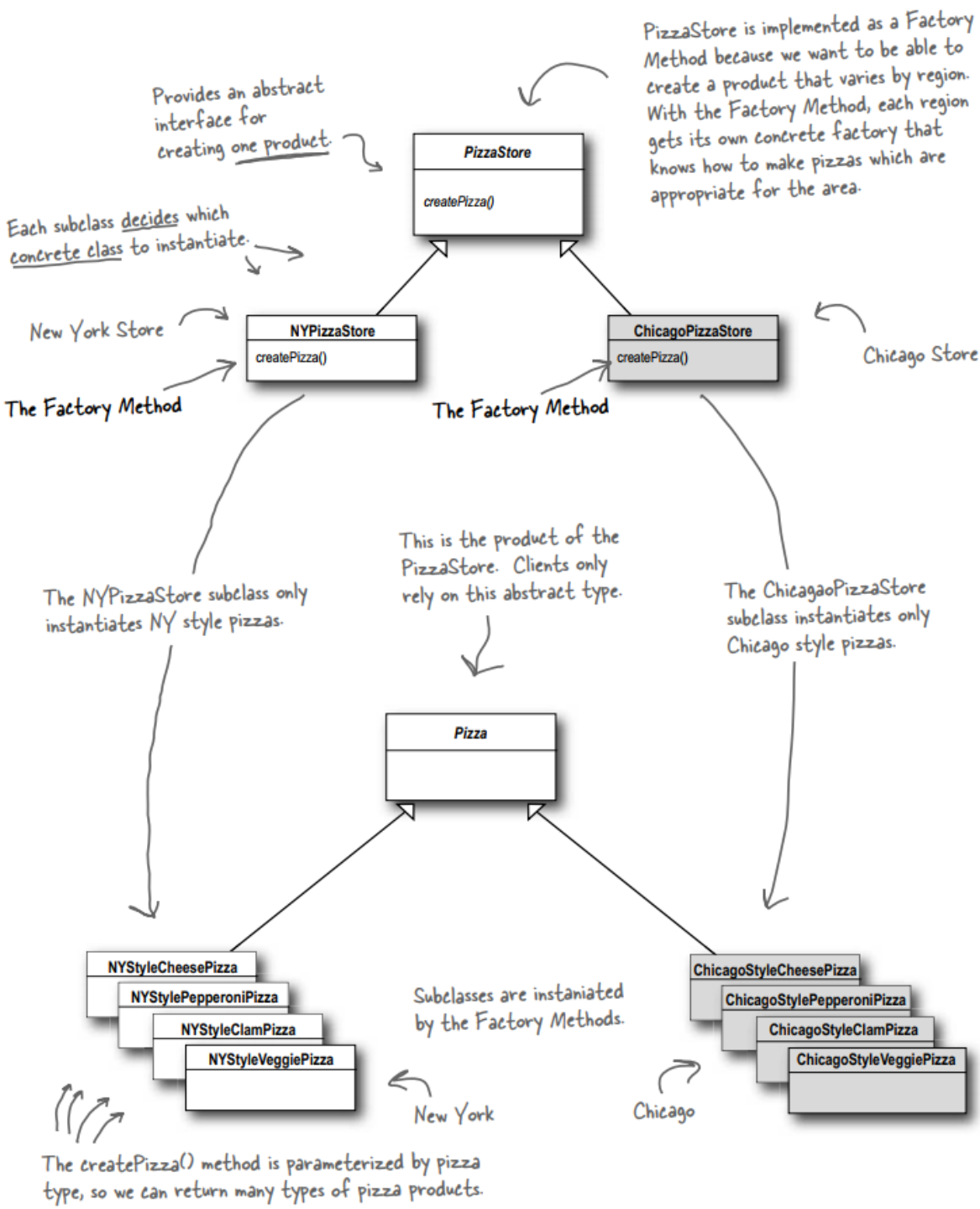
Factory Method: … trong trường hợp của tôi, tôi dùng hàm factory method để tạo ra product cụ thể mà người ta muốn, người dùng sẽ không biết cái gì được tạo ra, họ chỉ cần gọi hàm.

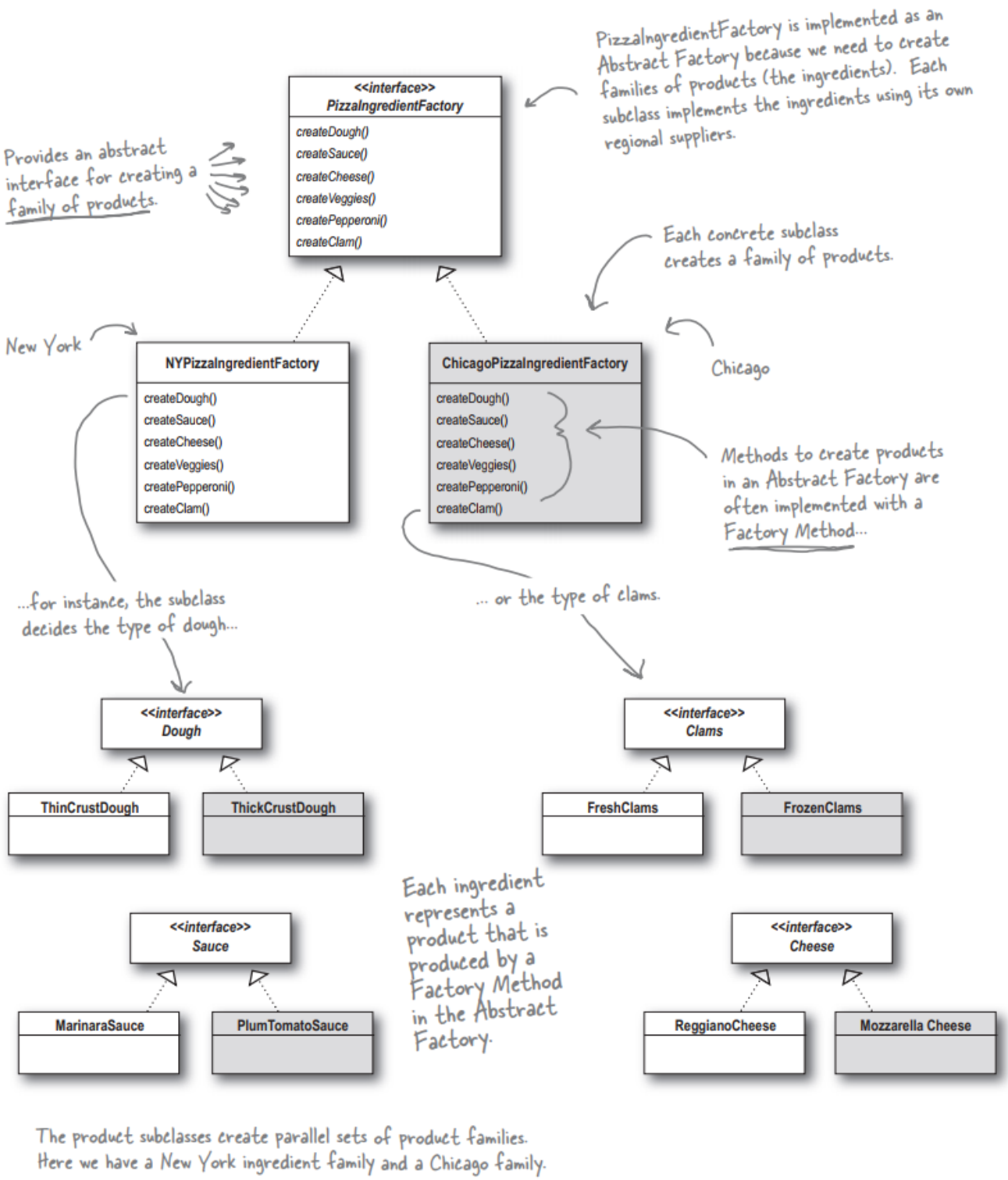
HeadFirst: Có vẻ như cả hai bạn đều giỏi trong những gì bạn làm. Tôi chắc rằng đọc giả đã có được lựa chọn của mình; Rốt cuộc, các factory rất hữu ích, họ sẽ muốn sử dụng chúng trong tình huống thích hợp. Cả hai bạn đều đóng gói việc tạo đối tượng để giúp chương trình độc lập và giảm phụ thuộc với những kiểu cụ thể, điều này thực sự tuyệt vời, cho dù bạn sử dụng **Factory Method** hay **Abstract Factory**. Hai bạn có lời gì trước khi chào tạm biệt đọc giả không ạ?

Abstract Factory: Cảm ơn. Hãy nhớ đến tôi, Abstract Factory và sử dụng tôi bất cứ khi nào bạn có "bộ sản phẩm" bạn cần tạo và bạn muốn đảm bảo client của bạn tạo ra các sản phẩm liên quan nhau.

Factory Method: Và Factory Method của tôi; sử dụng tôi để tách client code của bạn khỏi các lớp cụ thể mà bạn cần khởi tạo hoặc nếu bạn không biết trước tất cả các lớp cụ thể mà bạn sẽ cần. Để sử dụng tôi, chỉ cần tạo lớp con kế thừa tôi và implement factory method!

So sánh Factory Method và Abstract Factory Pattern


PizzaStore is implemented as a Factory Method because we want to be able to create a product that varies by region. With the Factory Method, each region gets its own concrete factory that knows how to make pizzas which are appropriate for the area.
Provides an abstract interface for creating one product.
PizzaStore
createPizza()
Each subclass decides which concrete class to instantiate.
New York Store
NYPizzaStore
createPizza()
ChicagoPizzaStore
createPizza()
Chicago Store
The Factory Method
The Factory Method
The NYPizzaStore subclass only instantiates NY style pizzas.
This is the product of the PizzaStore. Clients only rely on this abstract type.
The ChicagaoPizzaStore subclass instantiates only Chicago style pizzas.
Pizza
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza
Subclasses are instaniated by the Factory Methods.
ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza
New York
Chicago
The createPizza() method is parameterized by pizza type, so we can return many types of pizza products.



Tóm tắt

- Tất cả các factory đóng gói việc tạo đối tượng.
- Simple Factory không phải là một mẫu thiết kế thực sự, nó chỉ là một cách đơn giản để tách client code của bạn khỏi concrete classes.
- Factory Method dựa vào sự kế thừa: việc tạo đối tượng được ủy quyền cho các lớp con thực hiện phương thức Factory để tạo đối tượng.
- Abstract Factory dựa vào kết hợp đối tượng: việc tạo đối tượng được thực hiện theo các phương thức được hiển thị trong interface của factory.
- Tất cả các mẫu Factory thúc đẩy đạt tới "khớp nối lỏng lẻo" bằng cách giảm sự phụ thuộc của ứng dụng của bạn vào các lớp cụ thể.

- Mục đích của Factory Method là cho phép một lớp trì hoãn việc khởi tạo đối với các lớp con của nó.
- Mục đích của Abstract Factory là tạo ra một bộ của các đối tượng liên quan mà không phải phụ thuộc vào các lớp cụ thể của chúng.
- Nguyên tắc nghịch đảo phụ thuộc hướng dẫn chúng ta tránh phụ thuộc vào các loại cụ thể và cố gắng trừu tượng hóa.
- Factory là một kỹ thuật mạnh mẽ để đạt tới trình độ "*coding to abstractions*", không phải là "*coding to concrete*" nữa.