TRƯỜNG ĐẠI HỌC GIA ĐỊNH

KHOA CÔNG NGHỆ THÔNG TIN

\*\*\*\*\*

**🙜🙜 🕮 🙜🙜**



**BÁO CÁO CUỐI KỲ MÔN HỌC**

**LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG NÂNG CAO**

**ĐỀ TÀI: FACTORY DESIGN PATTERN**

GVHD: Ths. LÊ HUỲNH PHƯỚC

SVTH: NGUYỄN HỒ HẢI ANH 2008110020 – K14DCPM01

LÝ HOÀI LÂM 2008110053 – K14DCPM01

XÍCH VĂN HẬU 2008110008 – K14DCPM01

HOÀNG VIỆT 2008110321 – K14DCPM02

Tháng 8 – Năm 2022

**🙖 ✯✯✯ 🙐**

**MỤC LỤC**

[**1** **Giới thiệu Factory Pattern** 3](#_Toc112263352)

[1.1 Khái niệm 3](#_Toc112263353)

[1.2 Mục đích sử dụng Factory Pattern 3](#_Toc112263354)

[**2** **Thiết kế mẫu** 7](#_Toc112263355)

[2.1 Simple Factory (solution) 7](#_Toc112263356)

[2.1.1 Ý nghĩa 7](#_Toc112263357)

[2.1.2 Đặt vấn đề 7](#_Toc112263358)

[2.1.3 Thiết kế với SimpleFactory 11](#_Toc112263359)

[2.2 Factory Method 12](#_Toc112263360)

[2.2.1 Ý nghĩa: 12](#_Toc112263361)

[2.2.2 Cấu trúc mẫu 12](#_Toc112263362)

[2.2.3 Thiết kế Factory Method Pattern 14](#_Toc112263363)

[2.2.4 Tình huống áp dụng 20](#_Toc112263364)

[2.3 Abstract Factory Method Pattern 21](#_Toc112263365)

[2.3.1 Ý nghĩa: 21](#_Toc112263366)

[2.3.2 Cấu trúc mẫu: 21](#_Toc112263367)

[2.3.3 Thiết kế Abstract Factory Pattern 22](#_Toc112263368)

[2.3.4 Tình huống áp dụng 25](#_Toc112263369)

[**3** **Cài đặt & thử nghiệm** 26](#_Toc112263370)

[**4** **Đánh giá ưu, nhược điểm** 27](#_Toc112263371)

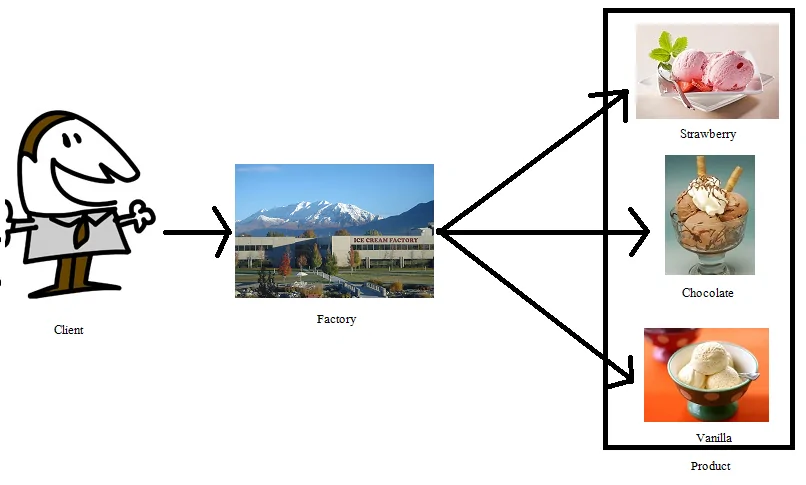
[4.1 Ưu điểm 27](#_Toc112263372)

[4.2 Nhược điểm 27](#_Toc112263373)

[**5** **Tài liệu tham khảo** 27](#_Toc112263374)

# **Giới thiệu Factory Pattern**

## ***Khái niệm***



Factory Pattern là một trong những Pattern phổ biến trong lập trình hướng đối tượng thuộc nhóm Creational Design Pattern (mẫu khởi tạo). Nhiệm vụ của mẫu này là quản lý và trả về các đối tượng theo yêu cầu, cung cấp một trong những cách tốt nhất để tạo một đối tượng. Giúp cho việc khởi tạo đối tượng một cách linh hoạt hơn

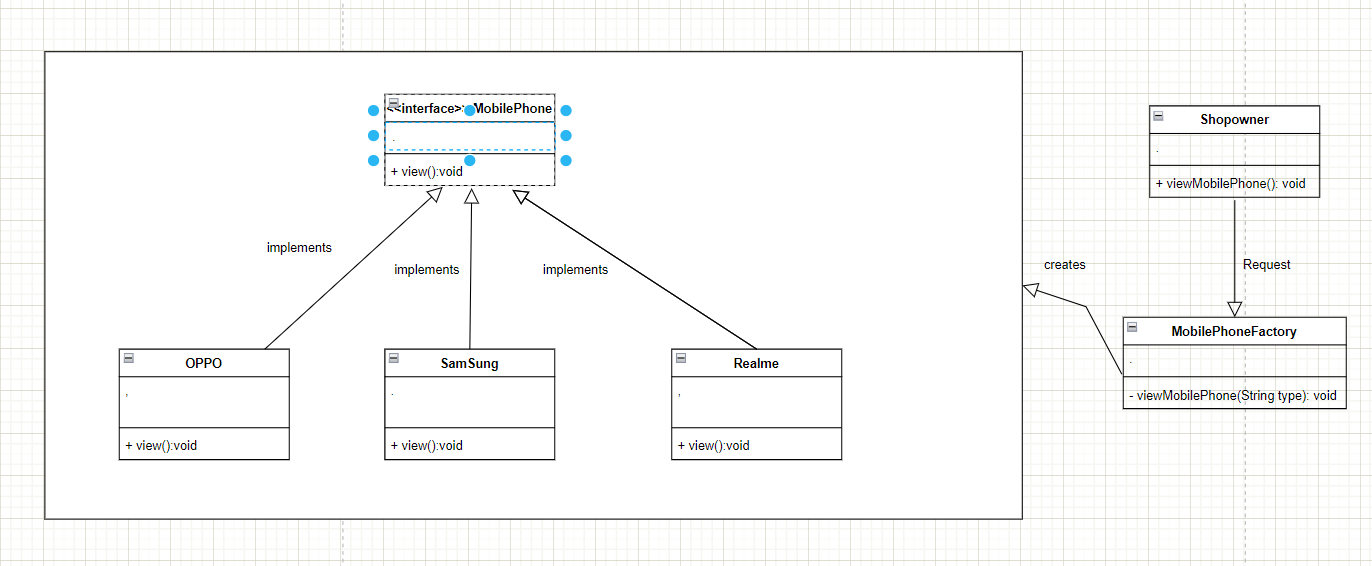
## ***Mục đích sử dụng Factory Pattern***

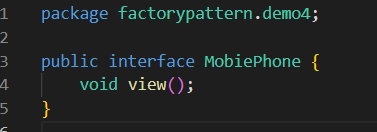
* Khi tạo ra 1 cách mới trong việc khởi tạo Object thông qua interface
* Khởi tạo các đối tượng để che giấu xử lý logic của việc khởi tạo đấy, trong trường hợp bạn đang muốn viết 1 thư viện để cho người khác use
* Giảm bớt sự phụ thuộc các module, các logic với các class cụ thể, mà chỉ phụ thuộc vào ỉnterface hoặc abstract class. Chính vì vậy dễ dàng cho việc mở rộng trong trường hợp bạn chưa biết chắc số lượng các đối tượng là đủ cho bài toán của mình chưa

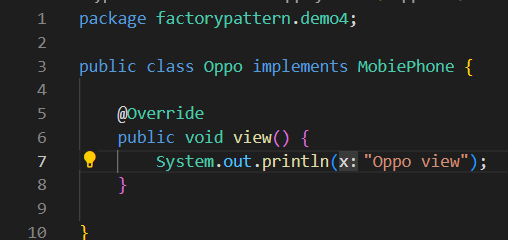
Ví dụ:

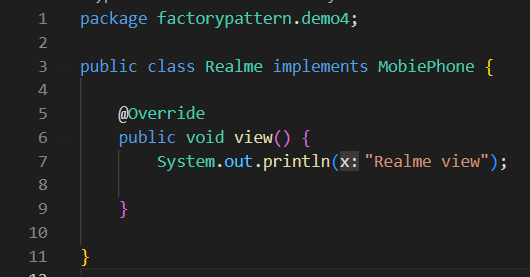
Giả sử bạn muốn mua một chiếc điện thoại, bạn sẽ phải đến các cửa hàng để xem xét các điện thoại khi mua. Chủ cửa hàng sẽ đưa điện thoại ra cho bạn xem

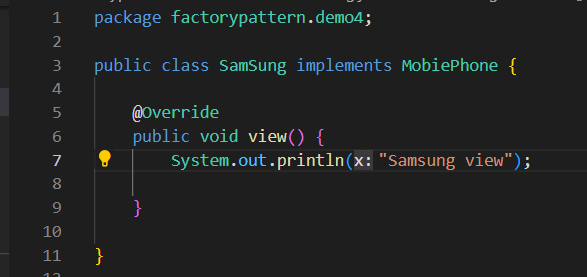
Sơ đồ class:

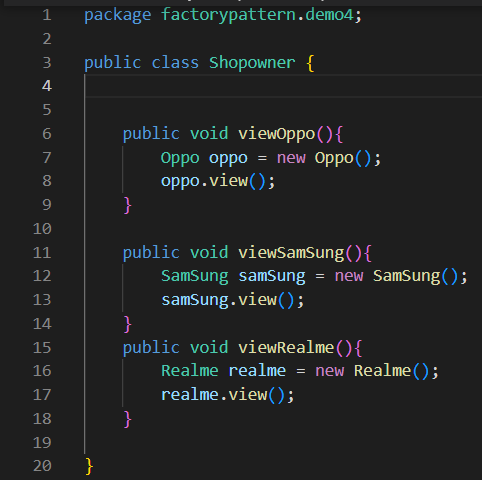






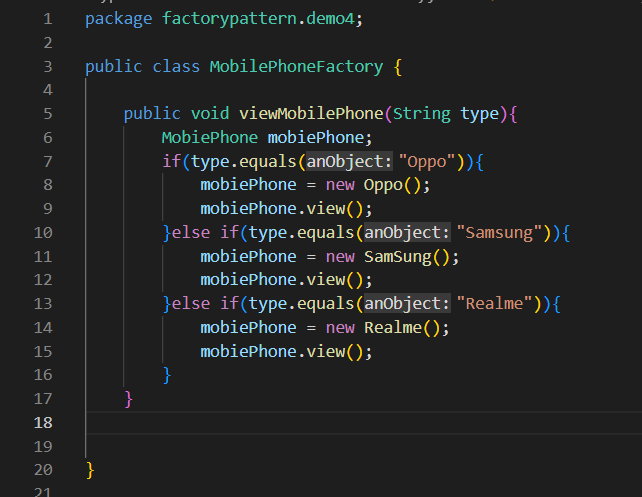




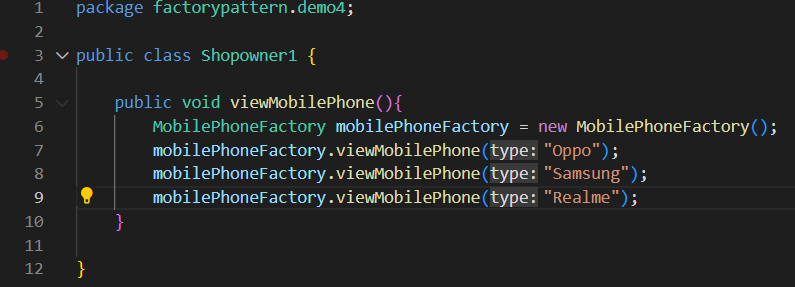


Khi tạo một ra một đối tượng của lớp con, ta phải dùng từ khóa new. Nhưng khi mà có sự thay đổi (thêm hoặc sửa) nó sẽ ảnh hưởng tới việc sử dụng new của chúng ta Nhưng bằng cách khác, khi “code với một interface”, bạn có thể bảo vệ mình thoát khỏi nhiều thay đổi có thể xảy ra với một hệ thống. Nếu code của bạn được dựa trên một interface, thì nó hoạt động với bất kỳ lớp nào mới nào implement interface đó thông qua đa hình. Tuy nhiên, khi bạn có code dựa trên lớp con, bạn sẽ gặp rắc rối vì code có thể phải thay đổi khi các lớp con được thêm vào

Việc làm này có vẻ mất thời gian và công sức của bạn khi đến từng cửa hàng để xem. Tuy nhiên có một cách khác đơn giản hơn, đó là đến một đại lý điện thoại có bán nhiều hãng để xem xét hết tất cả các điện thoại. Như vậy sẽ dễ dàng và nhanh chóng hơn



Lớp MobilePhoneFactory cho phép chúng ta tạo ra một địa điểm tập trung tất cả các đối tượng đã được tạo ra



# **Thiết kế mẫu**

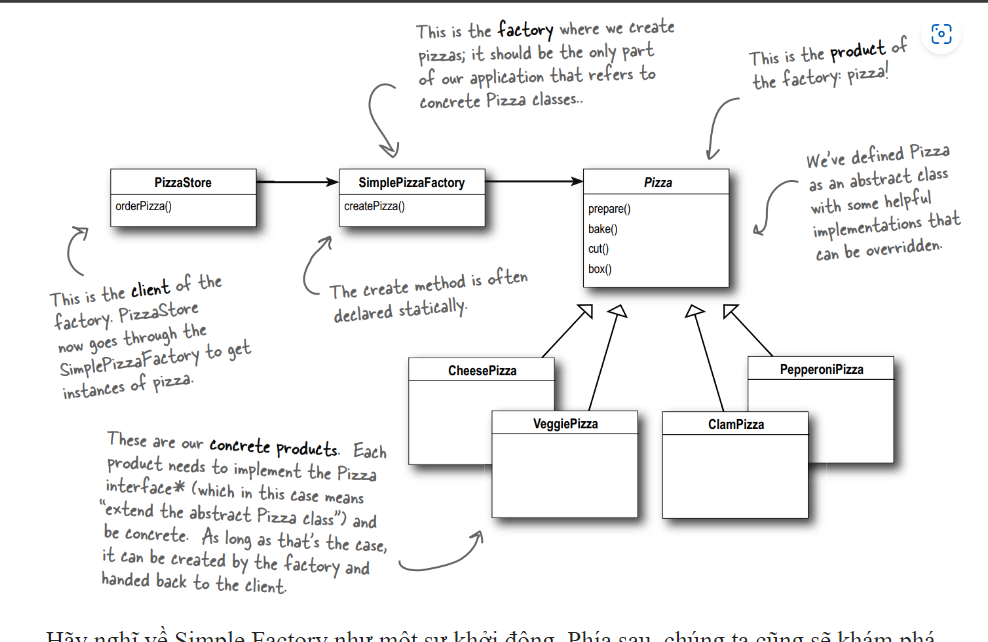
## ***Simple Factory (solution)***

### **Ý nghĩa**

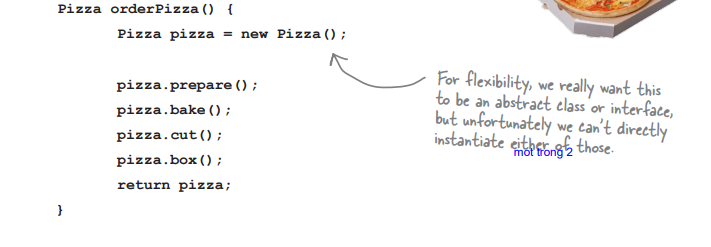
Simple Factory không thực sự là mẫu thiết kế thực sự, nó là một thành ngữ lập trình, nhưng nó thường được sử dụng nhưng nó là một giải pháp

### **Đặt vấn đề**

Sơ đồ:



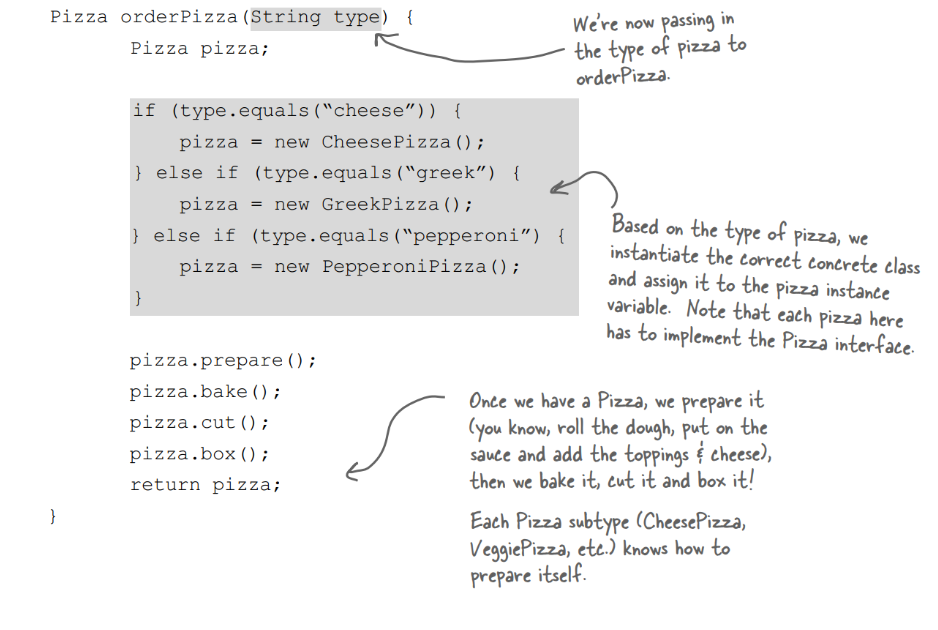
Khía cạnh: Bạn có một cửa hàng Pizza và là chủ cửa hàng Pizza tiên tiến ở Mỹ, bạn có thể viết code:



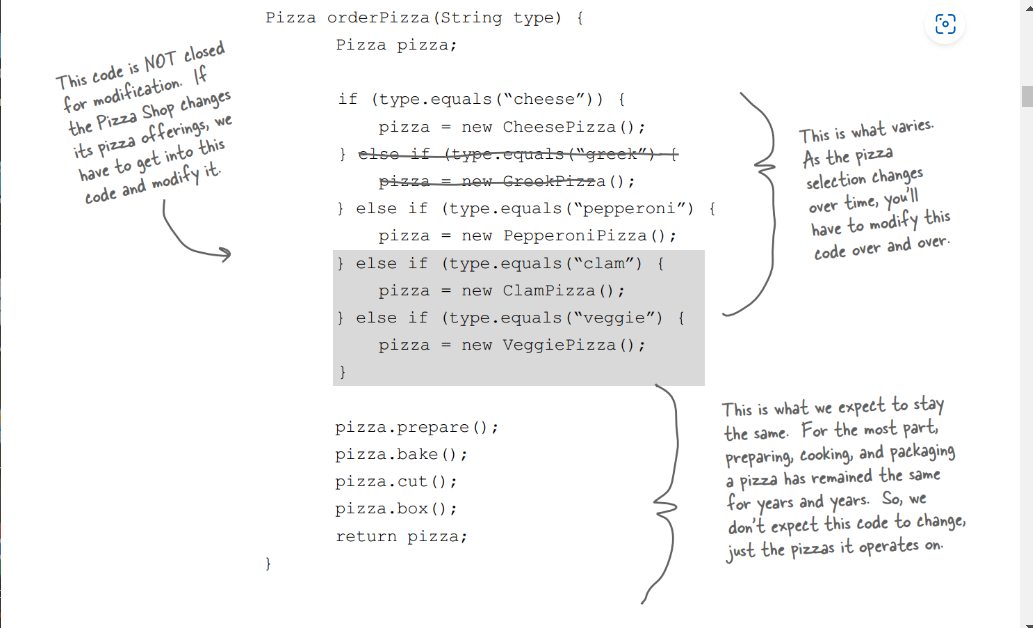
Để linh hoạt, chúng tôi thực sự muốn lớp Pizza() là một lớp abstract class hoặc interface, nhưng tiếc là không thể trực tiếp khởi tạo 1 trong hai lớp đó

Nhưng bạn cần nhiều loại pizza hơn...

Vì vậy, sau đó bạn thêm một dòng code định nghĩa loại pizza phù hợp và sau đó khởi tạo bánh pizza:

  
Nhưng phải thêm nhiều loại pizza nữa

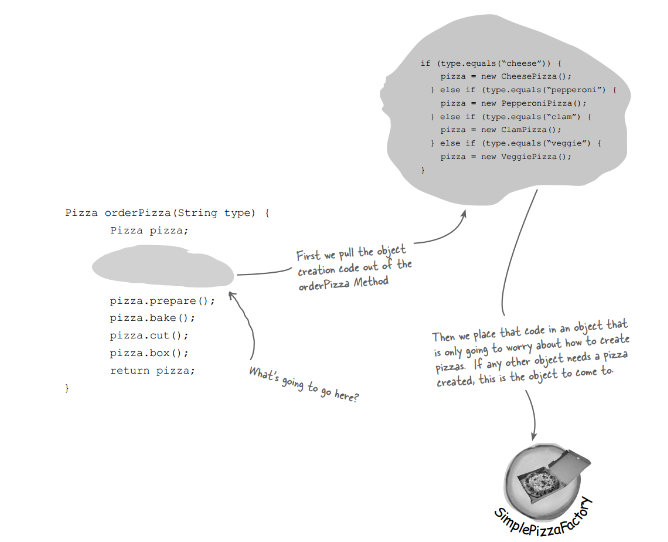
Bạn nhận ra rằng tất cả các đối thủ cạnh tranh của bạn đã thêm một vài loại pizza “hợp thời trang” vào thực đơn của họ: **Pizza Clam** và **Veggie Pizza**. Rõ ràng bạn cần theo kịp đối thủ, vì vậy bạn sẽ **thêm**các mục này vào thực đơn của mình. Và gần đây bạn không bán được nhiều bánh **Greek Pizzas**, vì vậy bạn quyết định **loại bỏ** nó ra khỏi thực đơn



Rõ ràng, việc xử lý khởi tạo trên thực sự gây rối cho phương thức **orderPizza()**của chúng ta và ngăn không cho nó “đóng cho việc sửa đổi”. Nhưng bây giờ chúng ta đã biết cái gì đang thay đổi và cái gì không, có lẽ đã đến lúc đóng gói nó.

Đóng gói việc khởi tạo đối tượng

Vì vậy, bây giờ chúng ta biết rằng sẽ tốt hơn khi chuyển việc tạo đối tượng ra khỏi phương thức **orderPizza()**. Nhưng bằng cách nào? Chà, những gì chúng tôi sẽ làm là lấy phần code tạo pizza và chuyển nó ra một đối tượng khác chỉ liên quan đến việc tạo ra pizza.



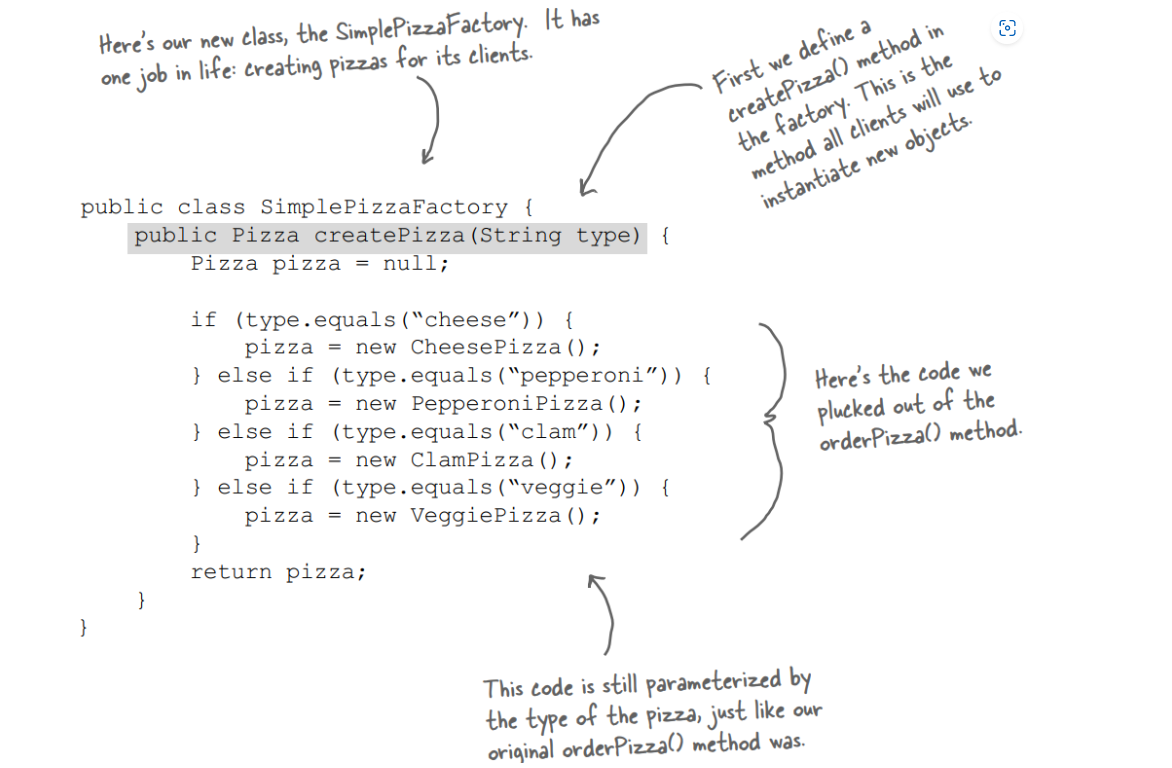
**Chúng tôi đã đặt tên cho đối tượng mới này: chúng tôi gọi nó là Factory.**

Các nhà máy Factory xử lý phần chi tiết của việc tạo đối tượng. Khi chúng ta có một **SimplePizzaFactory**, phương thức ***orderPizza()***của chúng ta sẽ trở thành một máy khách (client) của đối tượng đó. Bất cứ khi nào nó cần một chiếc bánh pizza, nó sẽ yêu cầu nhà máy sản xuất pizza làm một cái. Đã qua rồi cái thời mà phương thức ***orderPizza()***cần biết về loại pizza **Greek Pizzas** hay là**Clam pizzas**. Bây giờ phương thức ***orderPizza()*** chỉ quan tâm rằng nó nhận được một chiếc bánh pizza đang implement giao diện Pizza để nó có thể gọi ***prepare()***, ***bake()***, ***cut()***, và ***box()***.

Chúng tôi vẫn có một vài chi tiết để điền vào đây; chẳng hạn, phương thức ***orderPizza()*** thay thế code tạo của nó bằng gì? Hãy implement một nhà máy đơn giản cho cửa hàng pizza và tìm hiểu …

### **Thiết kế với SimpleFactory**

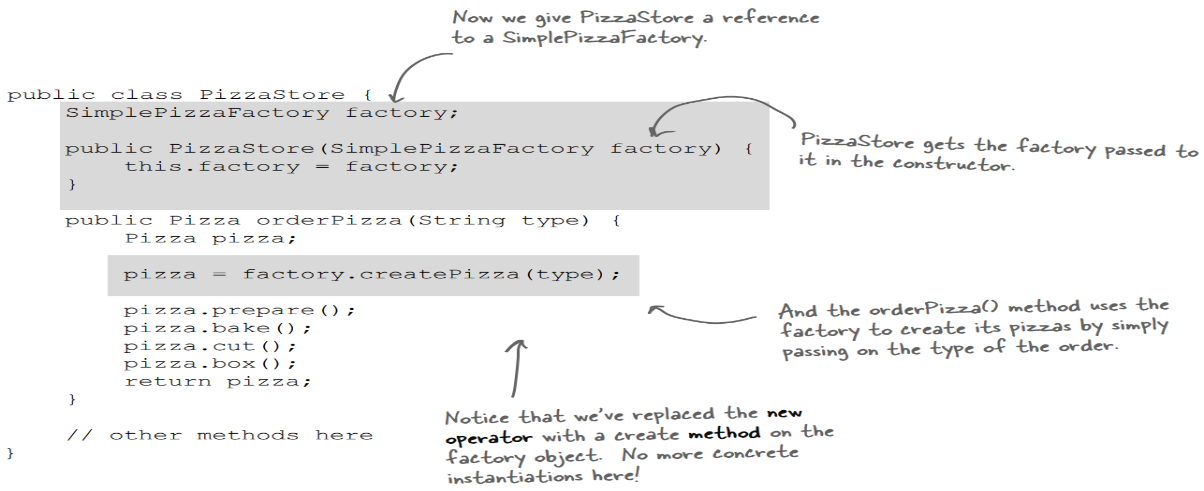
Chúng tôi sẽ bắt đầu với chính nhà máy. Những gì chúng tôi sẽ làm là định nghĩa một lớp gói gọn việc tạo đối tượng cho tất cả các loại pizza. Nó đây..



Làm lại lớp PizzaStore với Simple Factory

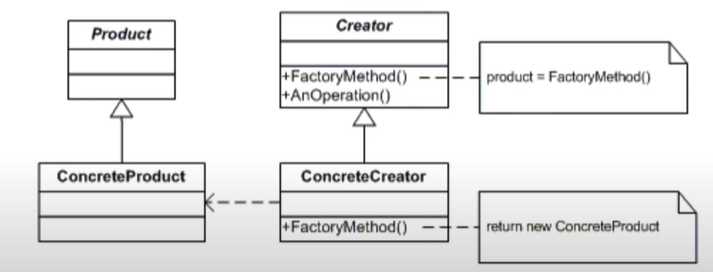
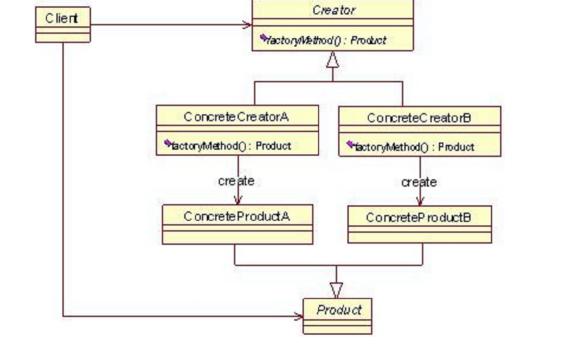
Bây giờ là thời gian để sửa code. Những gì chúng tôi muốn làm là dựa vào factory để tạo ra các loại pizza. Dưới đây là những thay đổi:

Bây giờ lớp PizzaStore (client) sẽ tham chiếu đến **SimplePizzaFactory** (factory) của chúng ta



## **Factory Method**

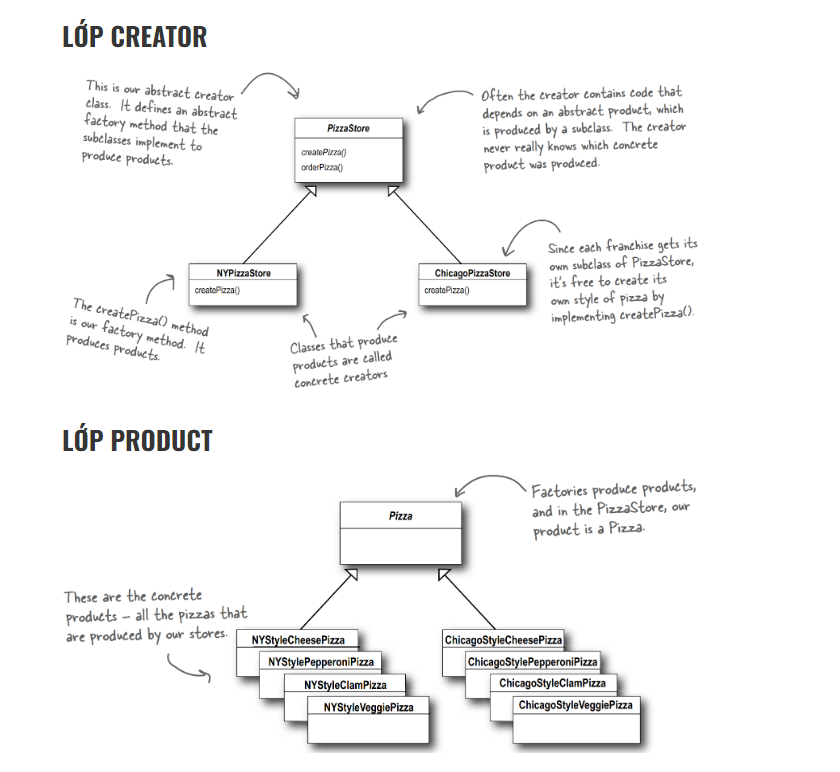
### **Ý nghĩa:**

Là một phương thức chuẩn để khởi tạo đối tượng, như là một phần của phương thức tạo, nhưng việc quyết định kiểu đối tượng nào được tạo ra thì phụ thuộc vào các lớp con

### **Cấu trúc mẫu**

Trong đó:

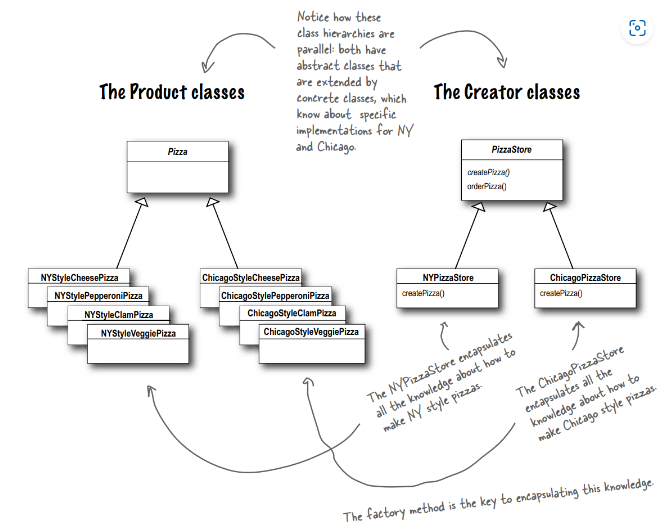
* Creator là lớp trừu tượng, khai báo phương thức factoryMethod() nhưng không cài đặt
* Product cũng là lớp trừu tượng
* ConcreteCreatorA và ConcreteCreatorB là 2 lớp kế thừa từ lớp Creator để tạo ra các đối tượng riêng biệt
* ConcreteProductA và ConcreteProductB là các lớp kế thừa của lớp Product, các đối tượng của 2 lớp này sẽ do 2 lớp ConcreteCreatorA và ConcreteCreatorB tạo ra



Hệ thống phân cấp lớp dạng song song

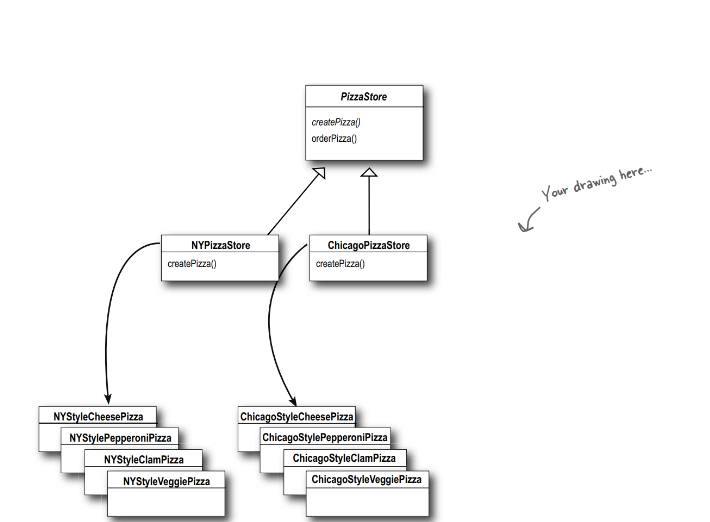
Chúng tôi đã thấy rằng factory pattern cung cấp một framework bằng cách cung cấp một phương thức **orderPizza()** được kết hợp với một factory method. Một cách khác để xem mẫu này như một framework là trong cách nó gói gọn “thông tin về sản phẩm” vào mỗi creator.

Hãy cùng xem xét hai hệ thống phân cấp lớp đặt song song và xem chúng liên quan như thế nào:

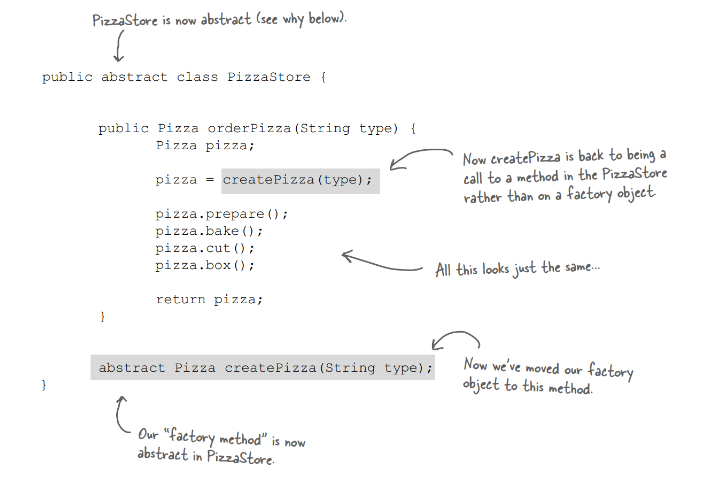


### **Thiết kế Factory Method Pattern**

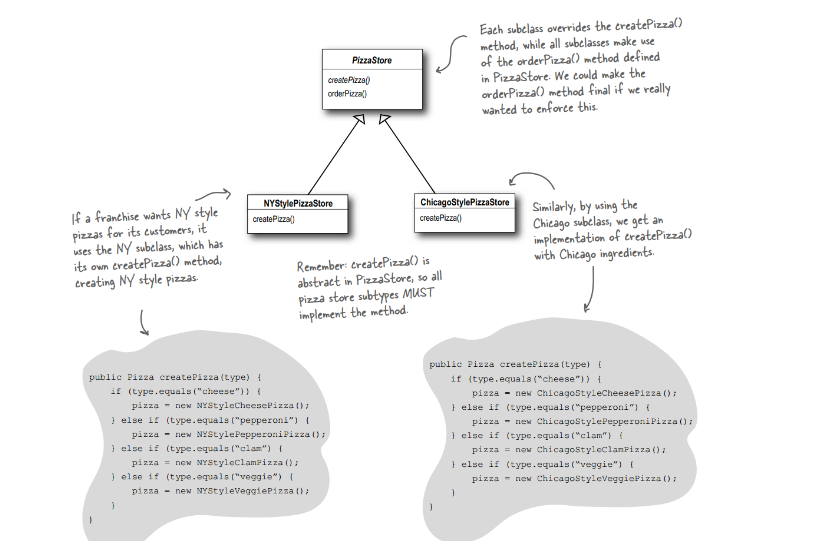
Chúng ta cần một loại bánh pizza khác cho những người California. Vẽ một tập hợp các lớp song song khác mà bạn cần để thêm một vùng California mới vào chuỗi cửa hàng **PizzaStore**của chúng tôi.



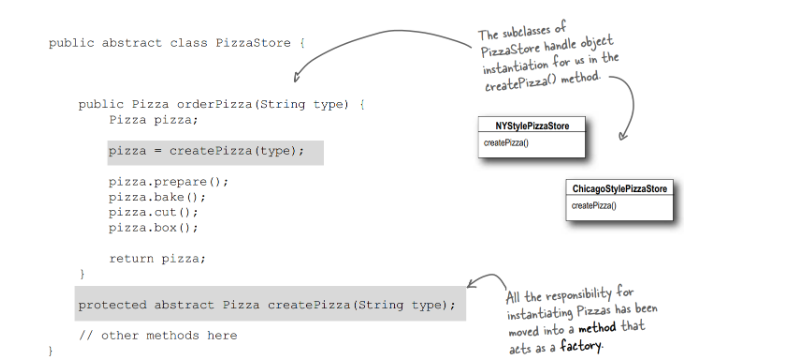
Đầu tiên, hãy xem các thay đổi của PizzaStore:



Bây giờ chúng tôi đã có một cửa hàng đang chờ các lớp con kế thừa; chúng tôi sẽ có các lớp con cho từng loại khu vực (**NYPizzaStore**, **ChicagoPizzaStore**, **CaliforniaPizzaStore**) và mỗi lớp con sẽ đưa ra quyết định về những gì tạo nên một chiếc bánh pizza. Hãy xem điều này sẽ hoạt động như thế nào.



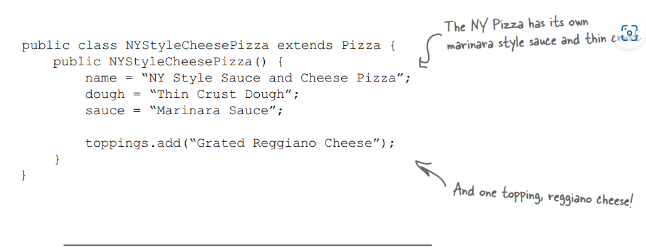
Chỉ với một vài biến đổi cho **PizzaStore**, chúng tôi đã chuyển từ việc chỉ có một đối tượng xử lý việc khởi tạo các lớp cụ thể của chúng tôi thành một tập hợp các lớp con hiện đang đảm nhận trách nhiệm đó. Hãy để xem xét kỹ hơn:

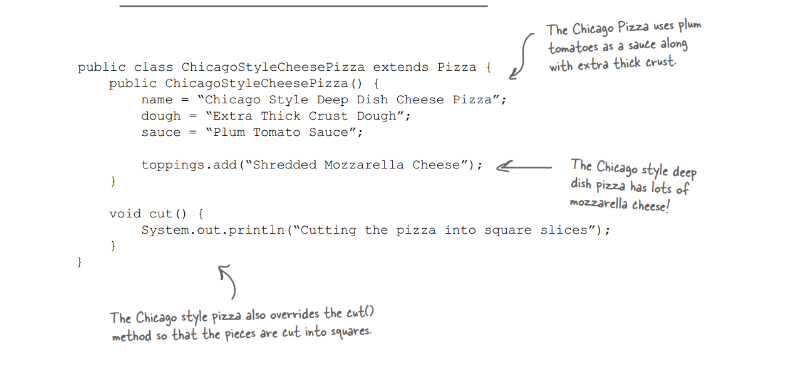


**MỘT SỐ LOẠI PIZZA, VÌ VẬY HÃY LÀM CHÚNG:**



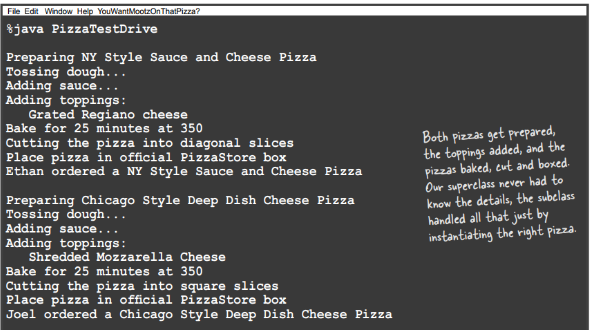
**BÂY GIỜ CHÚNG TA CHỈ CẦN MỘT SỐ LỚP CON CỤ THỂ … LÀM THẾ NÀO ĐỂ XÁC ĐỊNH CHEESE PIZZAS KIỂU NEW YORK HAY CHICAGO?**





**Bạn đã đợi đủ lâu, thời gian này cho một vài cái pizza!**





Như với mọi nhà máy, Mẫu **Factory Pattern**cho chúng ta một cách để gói gọn các khởi tạo của các loại cụ thể. Nhìn vào sơ đồ lớp bên dưới, bạn có thể thấy rằng abstract **Creator** cung cấp cho bạn một interface với phương thức khởi tạo các đối tượng, còn được gọi là “factory method”. Bất kỳ lớp nào khác implement lớp abstract **Creator**đều được viết để vận hành trên các sản phẩm (object) được tạo bởi factory method. Chỉ các lớp con thực sự thực hiện phương thức factory và tạo ra sản phẩm (object).

Như trong định nghĩa chính thức, bạn sẽ nghe các developer nói rằng **Factory Pattern**cho phép các lớp con quyết định lớp nào sẽ khởi tạo. Họ nói rằng “quyết định” không phải vì mô hình cho phép chính các lớp con quyết định trong runtime, mà bởi vì lớp creator.

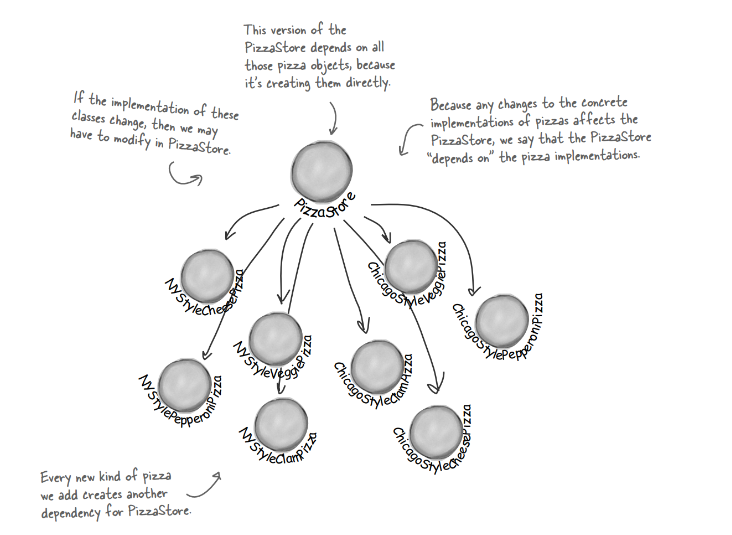
**Cửa hàng Pizza “rất phụ thuộc”**

Hãy giả vờ như bạn đã từng nghe nói về **Factory Pattern**. Ở đây, một phiên bản của PizzaStore không sử dụng factory pattern; hãy đếm số đối tượng con tạo pizza lớp này phụ thuộc. Nếu bạn đã thêm pizza kiểu California vào PizzaStore này, thì có bao nhiêu đối tượng sẽ phụ thuộc vào đó?



**Nhìn vào sự phụ thuộc đối tượng**

Khi bạn trực tiếp khởi tạo một đối tượng, bạn phụ thuộc vào lớp cụ thể của nó. Hãy xem lại **PizzaStore** rất nhiều sự phụ thuộc của chúng tôi. Nó tạo ra tất cả các đối tượng pizza ngay trong lớp **PizzaStore** thay vì ủy thác cho một lớp **Factory**.



### **Tình huống áp dụng**

* Khi bạn muốn tạo ra một framework có thể mở rộng, có nghĩa là nó cho phép tính mềm dẻo trong một số quyết định như chỉ ra loại đối tượng nào được tạo ra
* Khi bạn muốn 1 lớp con, mở rộng từ 1 lớp cha, quyết định lại đối tượng được khởi tạo
* Khi bạn biết khi nào thì khởi tạo một đối tượng nhưng không biết loại đối tượng nào được khởi tạo
* Bạn cần một vài khai báo chồng phương thức tạo với danh sách các tham số như nhau, điều mà Java không cho phép. Thay vì điều đó ta sử dụng các Factory Method với các tên khác nha

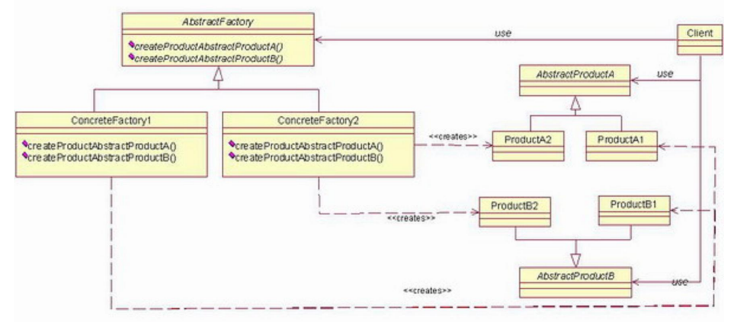
## **Abstract Factory Method Pattern**

### **Ý nghĩa:**

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**.**

Đóng gói một nhóm những lớp đóng vai trò “sản xuất” (factory) trong ứng dụng, đây là những lớp được dùng để tạo lập các đối tượng. Các lớp sản xuất này có chung một giao diện lập trình được kế thừa từ một lớp cha thuần ảo gọi là “lớp sản xuất ảo”

### **Cấu trúc mẫu:**



Trong đó :

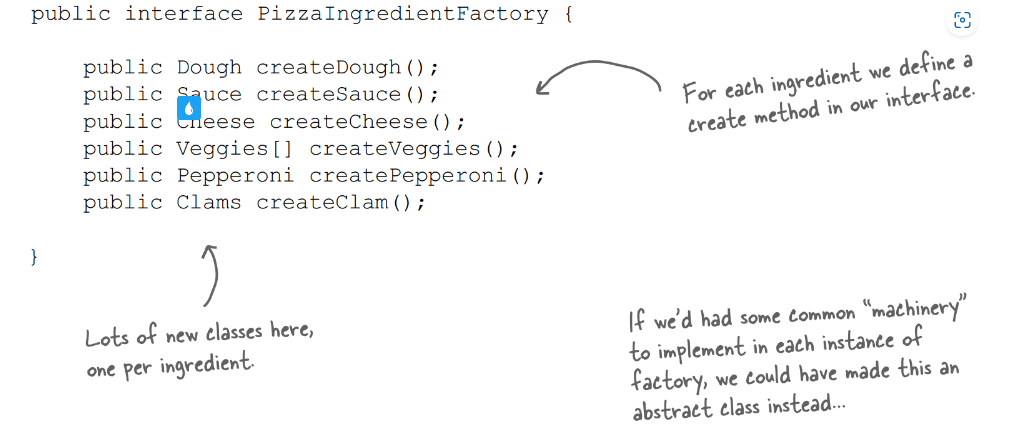
* **AbstractFactory:** là lớp trừu tượng, tạo ra các đối tượng thuộc 2 lớp trừu tượng là: AbstractProductA và AbstractProductB
* **ConcreteFactoryX:** là lớp kế thừa từ AbstractFatory, lớp này sẽ tạo ra một đối tượng cụ thể
* **AbstractProduct:** là các lớp trừu tượng, các đối tượng cụ thể sẽ là các thể hiện của các lớp dẫn xuất từ lớp này.

### **Thiết kế Abstract Factory Pattern**

**Xây dựng nhà máy nguyên liệu**

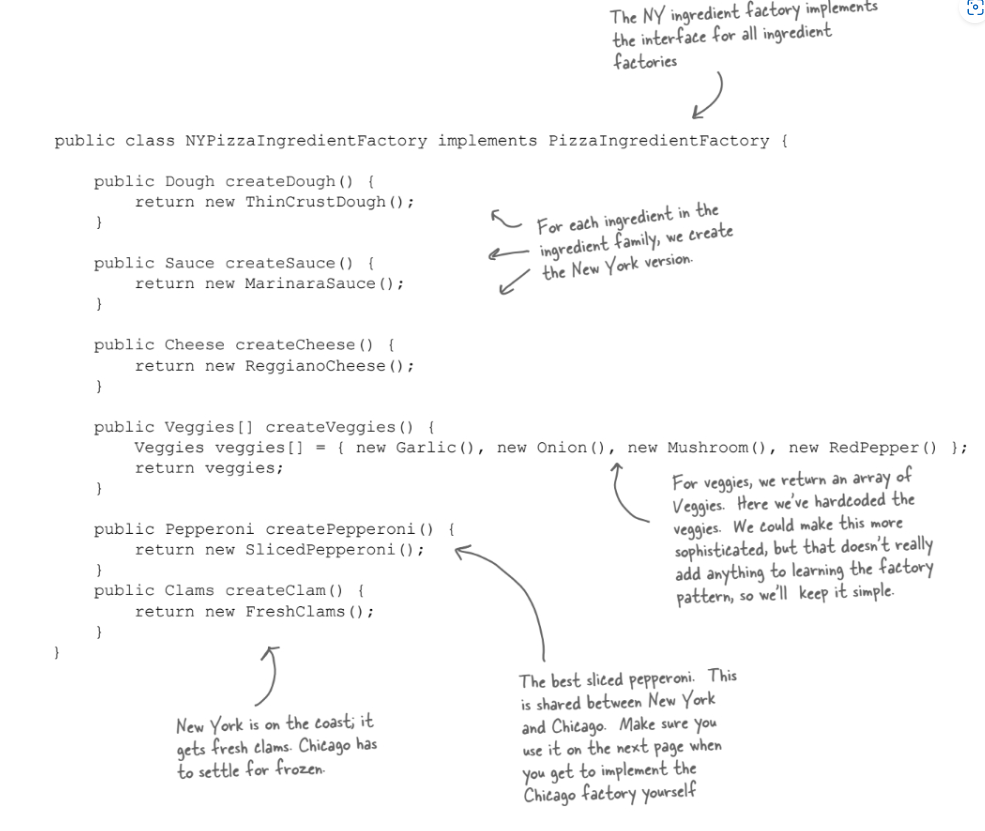
Bây giờ chúng ta sẽ xây dựng một nhà máy để tạo ra các nguyên liệu của chúng ta; nhà máy sẽ chịu trách nhiệm tạo ra từng nguyên liệu trong “bộ tnguyên liệu”. Nói cách khác, nhà máy sẽ cần tạo ra bột, nước sốt, phô mai, v.v … Bạn sẽ thấy cách chúng ta sẽ xử lý sự khác biệt trong từng khu vực ngay sau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

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

****

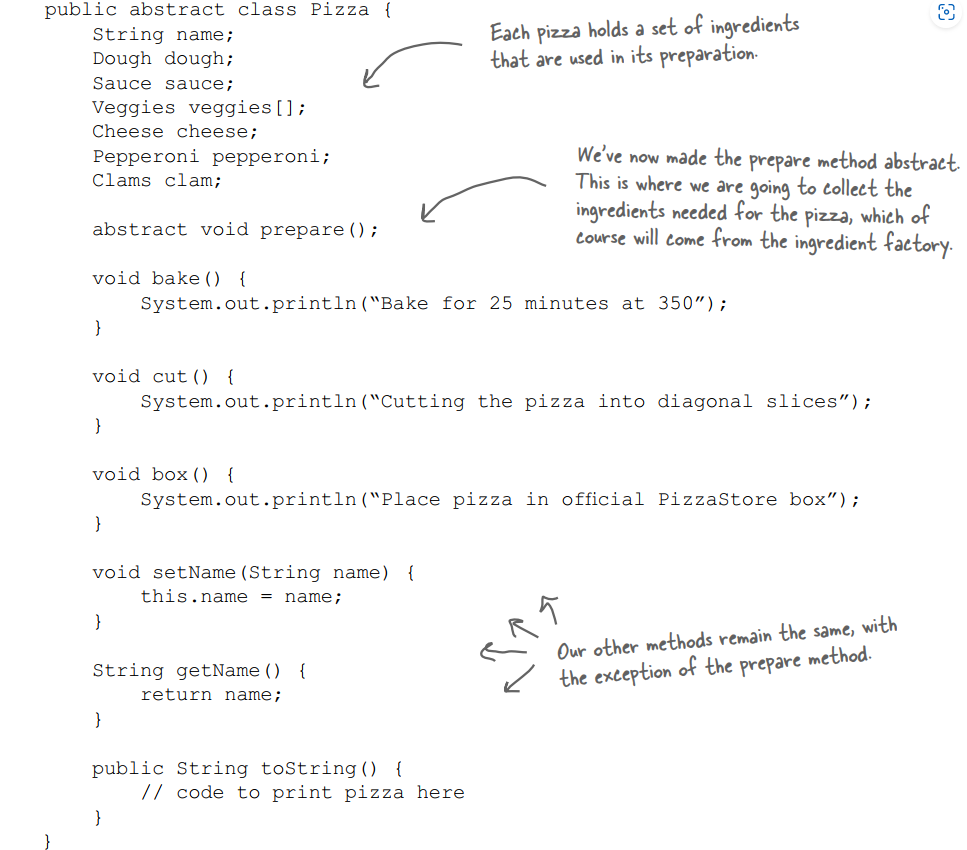
**Xây dựng nhà máy sản xuất nguyên liệu ở New York**

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…

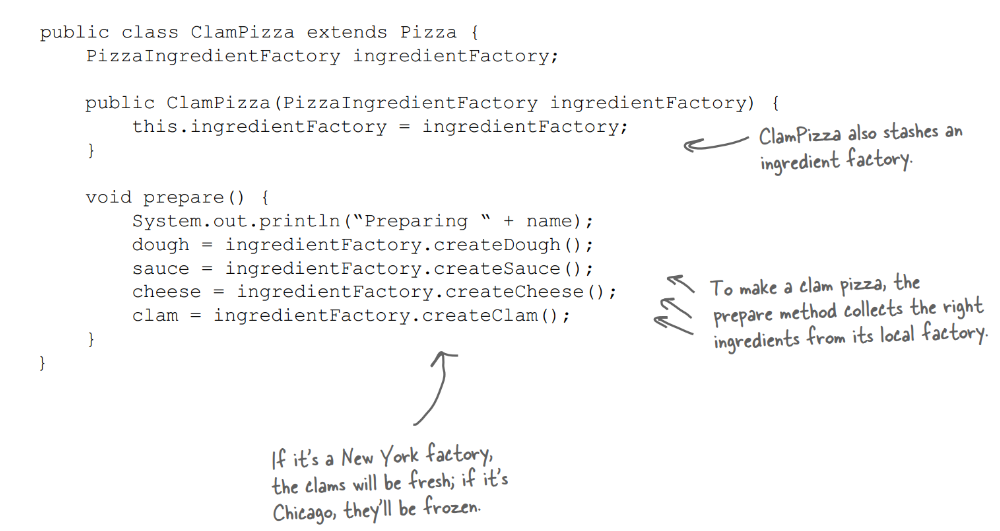
****

**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:

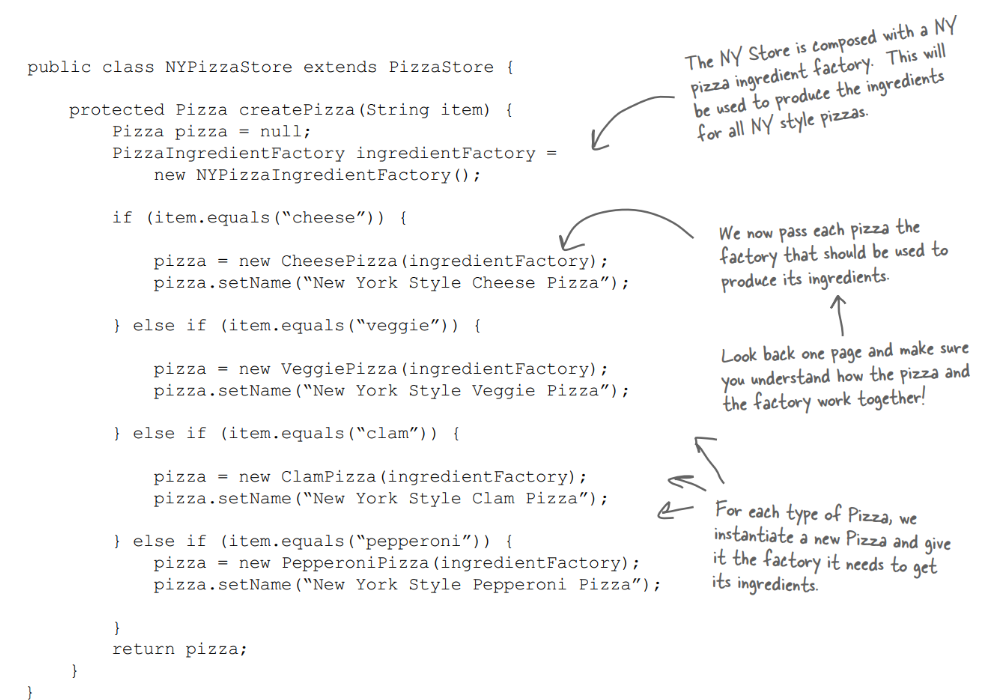
****

**Hãy cùng khám phá ClamPizza**

****

**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**…):

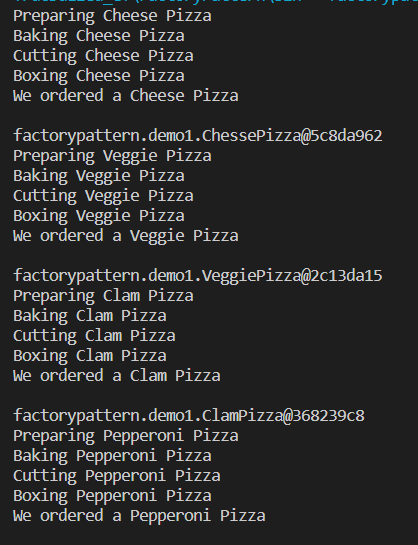
****

### **Tình huống áp dụng**

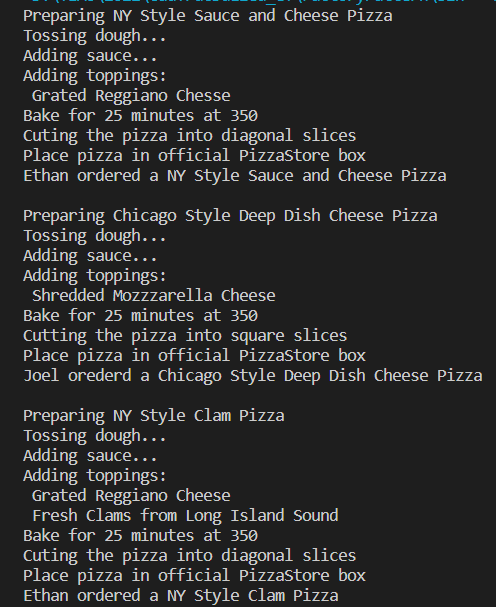
* Phía trình khách sẽ không phụ thuộc vào việc những sản phẩm được tạo ra như thế nào.
* Ứng dụng sẽ được cấu hình với một hoặc nhiều họ sản phẩm.
* Các đối tượng cần phải được tạo ra như một tập hợp để có thể tương thích với nhau.
* Chúng ta muốn cung cấp một tập các lớp và chúng ta muốn thể hiện các ràng buộc, các mối quan hệ giữa chúng mà không phải là các thực thi của chúng(interface).

# **Cài đặt & thử nghiệm**

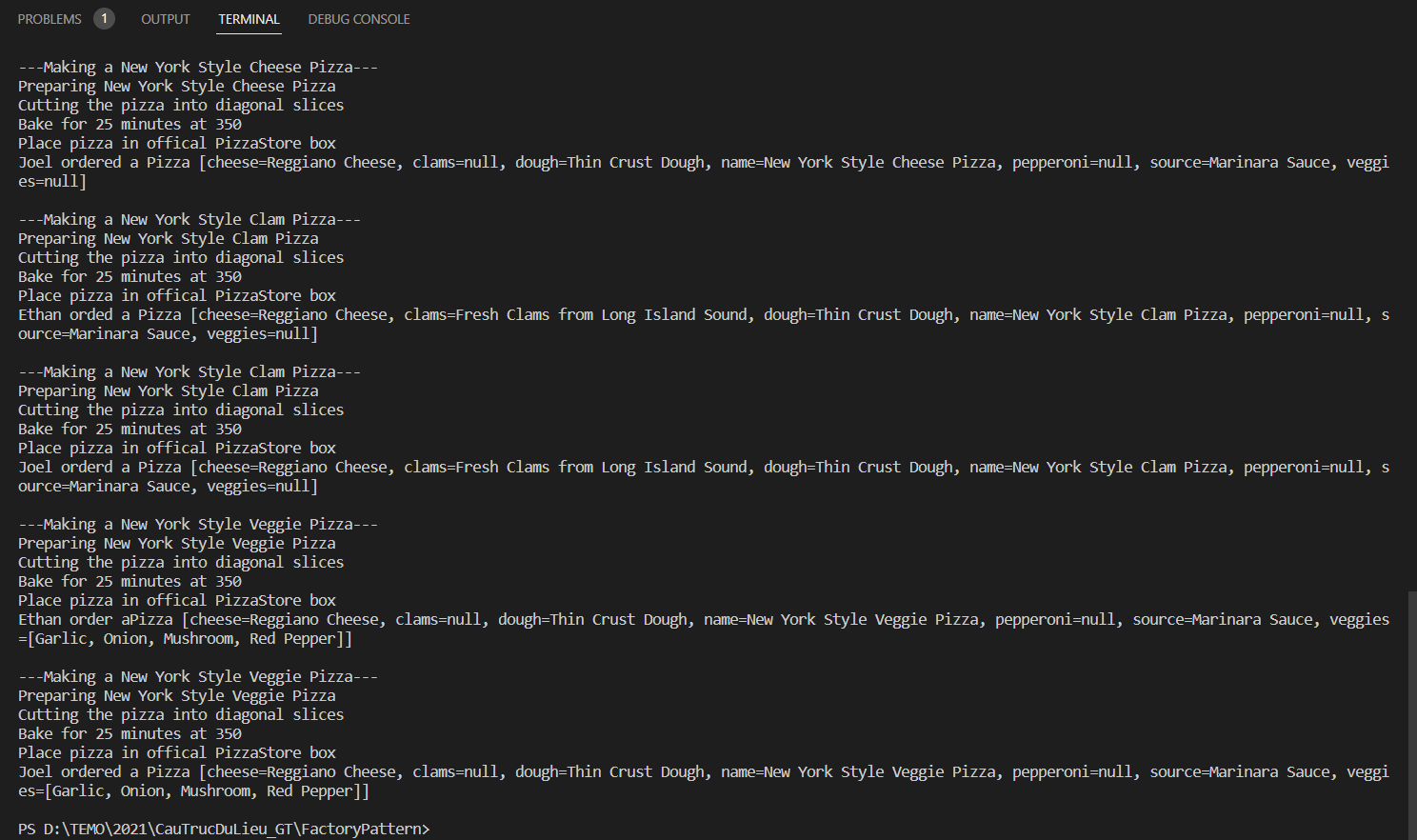
* Kết quả Màn hình SimpleFactory(solution)



* Kết quả màn hình Factory Method



* Kết quả màn hình Abstract Factory



# **Đánh giá ưu, nhược điểm**

## **Ưu điểm**

* Factory Method Pattern giúp hạn chế sự phụ thuộc giữa creator và concrete products
* Factory Method Pattern giúp gom các đoạn code tạo ra product vào một nơi trong chương trình, nhờ đó giúp dễ theo dõi và thao tác
* Với Pattern này, chúng ta có thể dễ dàng mở rộng, thêm những đoạn code mới vào chương trình mà không cần phá vỡ các đối tượng ban đầu

## **Nhược điểm**

Code có thể trở nên phức tạp hơn mức bình thường do đòi hỏi phải sử dụng nhiều class mới có thể cài đặt được pattern này

# **Tài liệu tham khảo**

1. <https://chiasekinang.com/factory-pattern/>
2. [Chương 4: Factory Pattern & Abstract Factory Pattern (Phần 1) (toihocdesignpattern.com)](https://toihocdesignpattern.com/chuong-4-factory-pattern-va-abstract-factory-pattern-phan-1.html?amp)
3. [Design Pattern - Factory pattern - MibeSoft](https://mibesoft.com/vn/blog/design-pattern-factory-pattern.html)