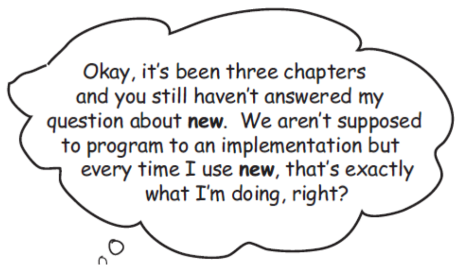
Factory Patterns


Okay, it's been three chapters and you still haven't answered my question about **new**. We aren't supposed to program to an implementation but every time I use **new**, that's exactly what I'm doing, right?

When you see "new", think "concrete".

Yes, when you use **new** you are certainly instantiating a concrete class, so that's definitely an implementation, not an interface. And it's a good question; you've learned that tying your code to a concrete class can make it more fragile and less flexible.

```
Duck duck = new MallardDuck();
```

We want to use interfaces to keep code flexible.

But we have to create an instance of a concrete class!

When you have a whole set of related concrete classes, often you're forced to write code like this:

```
Duck duck;

if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

We have a bunch of different duck classes, and we don't know until runtime which one we need to instantiate.

Here we've got several concrete classes being instantiated, and the decision of which to instantiate is made at runtime depending on some set of conditions.

When you see code like this, you know that when it comes time for changes or extensions, you'll have to reopen this code and examine what needs to be added (or deleted). Often this kind of code ends up in several parts of the application making maintenance and updates more difficult and error-prone.

What's wrong with "new"?

Technically there's nothing wrong with **new**, after all, it's a fundamental part of Java. The real culprit is our old friend CHANGE and how change impacts our use of **new**.

By coding to an interface, you know you can insulate yourself from a lot of changes that might happen to a system down the road. Why? If your code is written to an interface, then it will work with any new classes implementing that interface through polymorphism. However, when you have code that makes use of lots of concrete classes, you're looking for trouble because that code may have to be changed as new concrete classes are added. So, in other words, your code will not be "closed for modification." To extend it with new concrete types, you'll have to reopen it.

So what can you do? It's times like these that you can fall back on OO Design Principles to look for clues. Remember, our first principle deals with change and guides us to *identify the aspects that vary and separate them from what stays the same.*

Identifying the aspects that vary

Let's say you have a pizza shop, and as a cutting-edge pizza store owner in Objectville you might end up writing some code like this:

```
Pizza orderPizza() {
        Pizza pizza = new Pizza();

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

For flexibility, we really want this to be an abstract class or interface, but we can't directly instantiate either of those.

But you need more than one type of pizza...

So then you'd add some code that *determines* the appropriate type of pizza and then goes about *making* the pizza:

```
Pizza orderPizza(String type) {
       Pizza pizza;

       if (type.equals("cheese")) {
           pizza = new CheesePizza();
       } else if (type.equals("greek") {
           pizza = new GreekPizza();
       } else if (type.equals("pepperoni") {
           pizza = new PepperoniPizza();
       }

       pizza.prepare();
       pizza.bake();
       pizza.cut();
       pizza.box();
       return pizza;
}
```

We're now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce and add the toppings & cheese), then we bake it, cut it and box it!

Each Pizza subtype (CheesePizza, VeggiePizza, etc.) knows how to prepare itself.

But the pressure is on to add more pizza types

```
Pizza orderPizza(String type) {
       Pizza pizza;

       if (type.equals("cheese")) {
           pizza = new CheesePizza();
       } ~~else if (type.equals("greek") {~~
           ~~pizza = new GreekPizza~~();
       } else if (type.equals("pepperoni") {
           pizza = new PepperoniPizza();
       } else if (type.equals("clam") {
           pizza = new ClamPizza();
       } else if (type.equals("veggie") {
           pizza = new eggiePizza();
       }

       pizza.prepare();
       pizza.bake();
       pizza.cut();
       pizza.box();
       return pizza;
}
```

This code is NOT closed for modification. If the Pizza Shop changes its pizza offerings, we have to get into this code and modify it.

This is what varies. As the pizza selection changes over time, you'll have to modify this code over and over.

This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years and years. So, we don't expect this code to change, just the pizzas it operates on.

Clearly, dealing with *which* concrete class is instantiated is really messing up our orderPizza() method and preventing it from being closed for modification. But now that we know what is varying and what isn't, it's probably time to encapsulate it.

(Simple) Factory

Encapsulating object creation

So now we know we'd be better off moving the object creation out of the orderPizza() method. But how? Well, what we're going to do is take the creation code and move it out into another object that is only going to be concerned with creating pizzas.

The main essence

The Simple Factory pattern is the simplest class which has methods for creation other class instances. For example if you need to create a tons of instances for some specific class, it's much easier to use method which returns instances for you. That's the main idea. Factory helps us to keep all objects creation in one place and avoid of spreading `new` key value across codebase.

We've got a name for this new object: we call it a Factory.

Factories handle the details of object creation. Once we have a SimplePizzaFactory, our orderPizza() method just becomes a client of that object. Any time it needs a pizzam it asks the pizza factory to make one. Gone are the days when the orderPizza() method needs to know about Greek versus Clam pizzas. Now the orderPizza() method just cares that it gets a pizza, which implements the Pizza interface so that it can call prepare(), bake(), cut(), and box().

We've still got a few details to fill in here; for instance, what does the orderPizza() method replace its creation code with? Let's implement a simple factory for the pizza store and find out...

-> If you don't use any patterns and give the responsibility of creating instances to your classes, then you will end up with lots of classes having the same code: same switch cases (if – else) and using the new keyword. (as we saw by now in the pizza examples)

-> The main idea of the static / simple factory is that you put the responsibility of creating instances to one place (instead of many clients that use the same duplicate logic) and when you need to CHANGE (for example add a new pizza variant) you can just change it in one place: inside the factory class.

- This violates the Open/Closed principle because we are opening our class when we need to add something, instead of extending the current functionality.

```java
public class SimplePizzaFactory {

  public Pizza createPizza(String type) {
    Pizza pizza = null;

    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    }
    else if (type.equals("sucuk")) {
      pizza = new SucukPizza();
    }
    // more else if cases
    return pizza;
  }
```

This is the factory where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes..

```java
public class PizzaStore {

  SimplePizzaFactory pizzaFactory;

  public PizzaStore(SimplePizzaFactory pizzaFactory) {
    this.pizzaFactory = pizzaFactory;
  }

  public Pizza orderPizza(String type) {
    Pizza pizza = pizzaFactory.createPizza(type);

    // pizza.bake();
    // pizza.cut();
    // pizza.box();
    return pizza;
  }
}
```

This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.

Notice that we've replaced the new operator with a create method on the factory object. No more concrete instantiations here!

The Simple Factory isn't actually a Design Pattern; it's more of a programming idiom. But it is commonly used, so we'll give it a Head First Pattern Honorable Mention. Some developers do mistake this idiom for the "Factory Pattern," so the next time there is an awkward silence between you and another developer, you've got a nice topic to break the ice.

Q: What's the advantage of this? It looks like we are just pushing the problem off to another object.

SimplePizzaFactory may have many clients.

So, by encapsulating the pizza creating in one class, we now have only one place to make modifications when the implementation changes.

Don't forget, we are also just about to remove the concrete instantiations from our client code!

Franchising the pizza store

Your Objectville PizzaStore has done so well that you've trounced the competition and now everyone wants a PizzaStore in their own neighborhood. As the franchiser, you want to ensure the quality of the franchise operations and so you want them to use your time-tested code.

But what about regional differences? Each franchise might want to offer different styles of pizzas (New York, Chicago, and California, to name a few), depending on where the franchise store is located and the tastes of the local pizza connoisseurs.

You want all the franchise pizza stores to leverage your PizzaStore code, so the pizzas are prepared in the same way.

We've seen one approach...

If we take out SimplePizzaFactory and create three different factories, NYPizzaFactory, ChicagoPizzaFactory and CaliforniaPizzaFactory, then we can just compose the PizzaStore with the appropriate factory and a franchise is good to go. That's one approach.

Let's see what that would look like...

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("Veggie");
```

Here we create a factory for making NY style pizzas.

Then we create a PizzaStore and pass it a reference to the NY factory.

...and when we make pizzas, we get NY-styled pizzas.

But you'd like a little more quality control...

So you test marketed the SimpleFactory idea, and what you found was that the franchises were using your factory to create pizzas, but starting to employ their own home grown procedures for the rest of the process: they'd bake things a little differently, they'd forget to cut the pizza and they'd use third-party boxes.

Rethinking the problem a bit, you see that what you'd really like to do is create a framework that ties the store and the pizza creation together, yet still allows things to remain flexible.

In our early code, before the SimplePizzaFactory, we had the pizza-making code tied to the PizzaStore, but it wasn't flexible. So, how can we have our pizza and eat it too?

A framework for the pizza store

There *is* a way to localize all the pizza making activities to the PizzaStore class, and yet give the franchises freedom to have their own regional style.

What we're going to do is put the createPizza() method back into PizzaStore, but this time as an **abstract method**, and then create a PizzaStore subclass for each regional style.

First, let's look at the changes to the PizzaStore:

```
public abstract class PizzaStore {

  public final Pizza orderPizza(String type) {
    Pizza pizza = createPizza(type);

    // pizza.bake();
    // pizza.cut();
    // pizza.box();
    return pizza;
  }

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

Our "factory method" is now abstract in PizzaStore.

Now we've got a store waiting for subclasses; we're going to have a subclass for each regional type (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) and each subclass is going to make the decision about what makes up a pizza. Let's take a look at how this is going to work.

Allowing the subclasses to decide

Remember, the PizzaStore already has a well-honed order system in the orderPizza() method and you want to ensure that it's consistent across all franchises.

What varies among the regional PizzaStores is the style of pizzas they make – New York Pizza has thin crust, Chicago Pizza has thick , and so on – and we are going to push all these variations into the createPizza() method and make it responsible for creating the right kind of pizza. The way we do this is by letting each subclass of PizzaStore define what the createPizza() method looks like. So, we will have a number of concrete subclasses of PizzaStore, each with its own pizza variations, all fitting within the PizzaStore framework and still making use of the well-tuned orderPizza() method.

Each subclass overrides the createPizza() method, while all subclasses make use of the orderPizza() method defined in PizzaStore. We could make the orderPizza() method final if we really wanted to enforce this.

Let's make a PizzaStore

Being a franchise has its benefits. You get all the PizzaStore functionality for free. All the regional stores need to do is subclass PizzaStore and supply a createPizza() method that implements their style of Pizza. We'll take care of the big three pizza styles for the franchisees.

Here's the New York regional style:

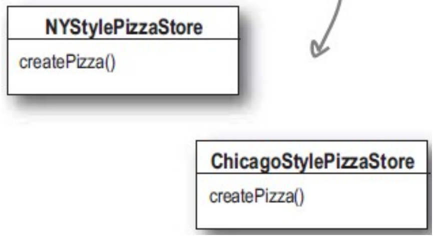
```java
public class NYPizzaStore extends PizzaStore {

  @Override
  protected Pizza createPizza(String type) {
    Pizza pizza = null;

    if (type.equals("cheese")) {
      pizza = new NYStyleCheesePizza();
    }
    else if (type.equals("sucuk")) {
      pizza = new NYStyleSucukPizza();
    }
    // more else if cases
    return pizza;
  }
```

All the responsibility for instantiating Pizzas has been moved into a **method** that acts as a **factory**.

Declaring a factory method

With just a couple of transformations to the PizzaStore we've gone from having an object handle the instantiation of our concrete classes to a set of subclasses that are now taking on that responsibility. Let's take a closer look:

The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.



A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

A factory method may be parameterized (or not) to select among several variations of a product.

```java
abstract Product factoryMethod(String type)
```

A factory method is abstract so the subclasses are counted on to handle object creation.

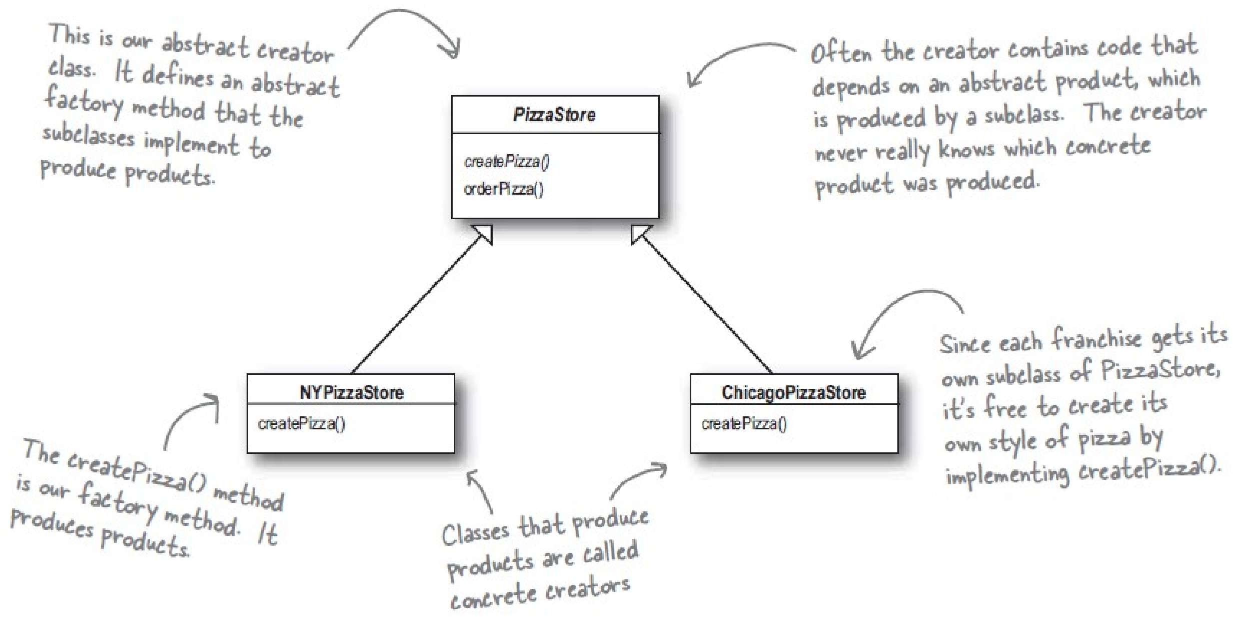
A factory method returns a Product that is typically used within methods defined in the superclass.

A factory method isolates the client (the code in the superclass, like orderPizza()) from knowing what kind of concrete Product is actually created.

Factory Method Pattern

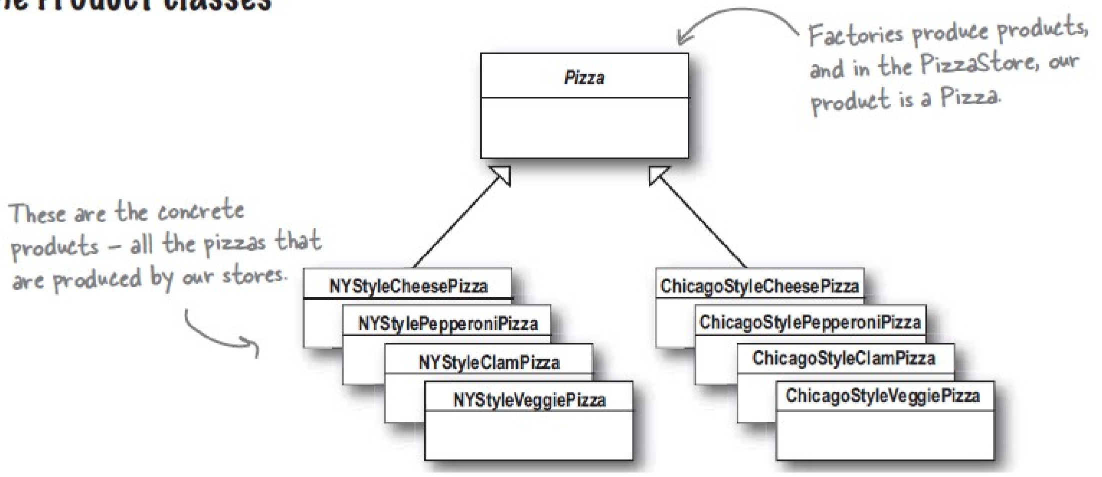
All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create. Let's check out these class diagrams to see who the players are in this pattern:

The Creator classes



This is our abstract creator class. It defines an abstract factory method that the subclasses implement to produce products.
PizzaStore
createPizza()
orderPizza()
Often the creator contains code that depends on an abstract product, which is produced by a subclass. The creator never really knows which concrete product was produced.
NYPizzaStore
createPizza()
ChicagoPizzaStore
createPizza()
Since each franchise gets its own subclass of PizzaStore, it's free to create its own style of pizza by implementing createPizza().
The createPizza() method is our factory method. It produces products.
Classes that produce products are called concrete creators

The Product classes



Pizza
Factories produce products, and in the PizzaStore, our product is a Pizza.
These are the concrete products – all the pizzas that are produced by our stores.
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza
ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza

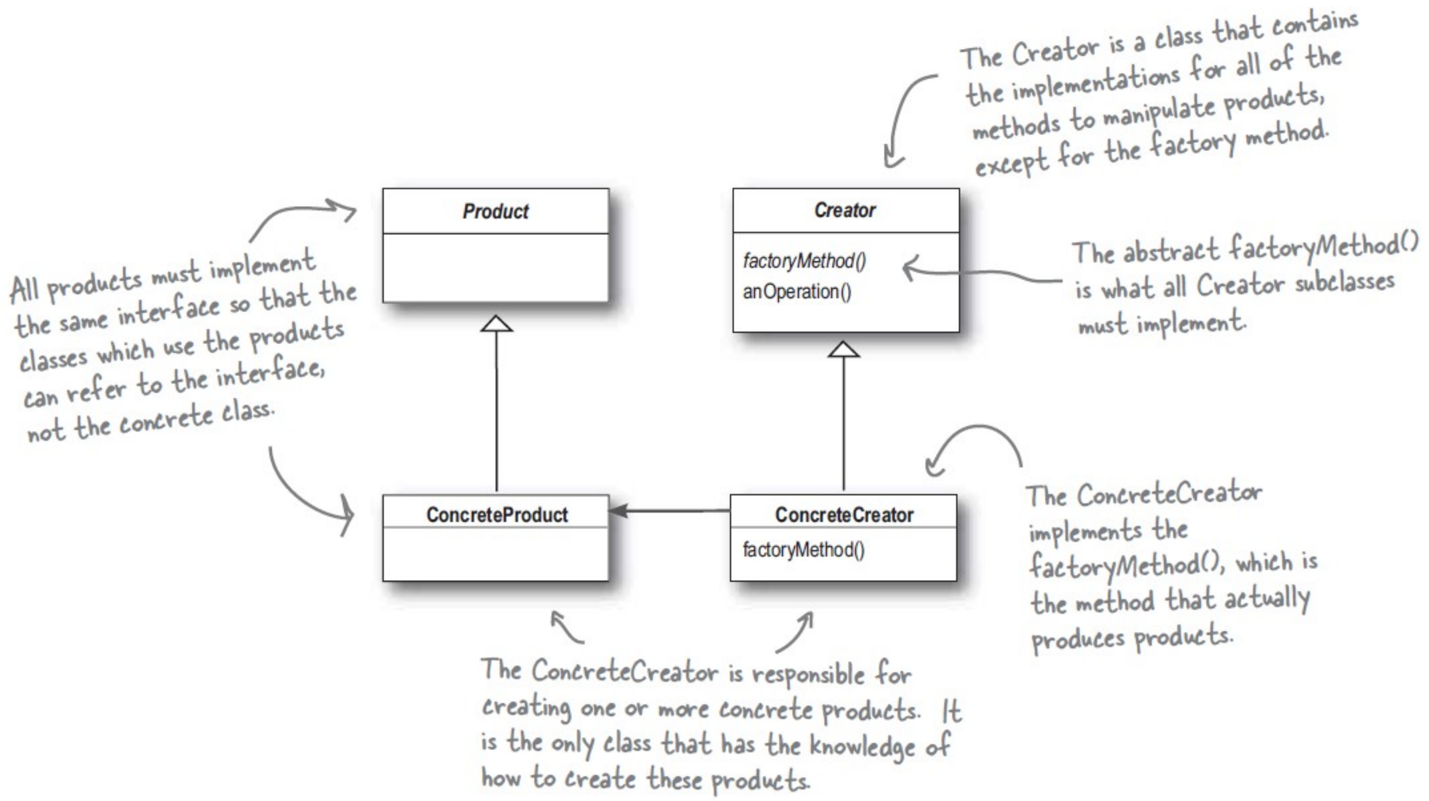
Factory Method Pattern defined

It's time to roll out the official definition of the Factory Method Pattern:

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. Looking at the class diagram below, you can see that the abstract Creator gives you an interface with a method for creating objects, also known as the "factory method." Any other methods implemented in the abstract Creator are written to operate on products produced by the factory method. Only subclasses actually implement the factory method and create products.

As in the official definition, you'll often hear developers say that the Factory Method lets subclasses decide which class to instantiate. They say "decides" not because the pattern allows subclasses themselves to decide at runtime, but because the creator class is written without knowledge of the actual products that will be created, which is decided purely by the choice of the subclass that is used.


The Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.
Product
Creator
factoryMethod()
anOperation()
The abstract factoryMethod() is what all Creator subclasses must implement.
All products must implement the same interface so that the classes which use the products can refer to the interface, not the concrete class.
ConcreteProduct
ConcreteCreator
factoryMethod()
The ConcreteCreator implements the factoryMethod(), which is the method that actually produces products.
The ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

there are no Dumb Questions

Q: What's the advantage of the Factory Method Pattern when you only have one ConcreteCreator?

A: The Factory Method Pattern is useful if you've only got one concrete creator because you are decoupling the implementation of the product from its use. If you add additional products or change a product's implementation, it will not affect your Creator (because the Creator is not tightly coupled to any ConcreteProduct).

Q: Would it be correct to say that our NY and Chicago stores are implemented using Simple Factory? They look just like it.

A: They're similar, but used in different ways. Even though the implementation of each concrete store looks a lot like the SimplePizzaFactory, remember that the concrete stores are extending a class which has defined createPizza() as an abstract method. It is up to each store to define the behavior of the createPizza() method. In Simple Factory, the factory is another object that is composed with the PizzaStore.

Q: Are the factory method and the Creator always abstract?

A: No, you can define a default factory method to produce some concrete product. Then you always have a means of creating products even if there are no subclasses of the Creator.

Q: Each store can make four different kinds of pizzas based on the type passed in. Do all concrete creators make multiple products, or do they sometimes just make one?

A: We implemented what is known as the parameterized factory method. It can make more than one object based on a parameter passed in, as you noticed. Often, however, a factory just produces one object and is not parameterized. Both are valid forms of the pattern.

Q: Your parameterized types don't seem "type-safe." I'm just passing in a String! What if I asked for a "CalmPizza"?

A: You are certainly correct and that would cause, what we call in the business, a "runtime error." There are several other more sophisticated techniques that can be used to make parameters more "type safe", or, in other words, to ensure errors in parameters can be caught at compile time. For instance, you can create objects that represent the parameter types, use static constants, or, in Java 5, you can use *enums*.

Q: I'm still a bit confused about the difference between Simple Factory and Factory Method. They look very similar, except that in Factory Method, the class that returns the pizza is a subclass. Can you explain?

A: You're right that the subclasses do look a lot like Simple Factory, however think of Simple Factory as a one shot deal, while with Factory Method you are creating a framework that let's the subclasses decide which implementation will be used. For example, the orderPizza() method in the Factory Method provides a general framework for creating pizzas that relies on a factory method to actually create the concrete classes that go into making a pizza. By subclassing the PizzaStore class, you decide what concrete products go into making the pizza that orderPizza() returns. Compare that with SimpleFactory, which gives you a way to encapsulate object creation, but doesn't give you the flexibility of the Factory Method because there is no way to vary the products you're creating.

This version of the PizzaStore depends on all those pizza objects, because it's creating them directly.

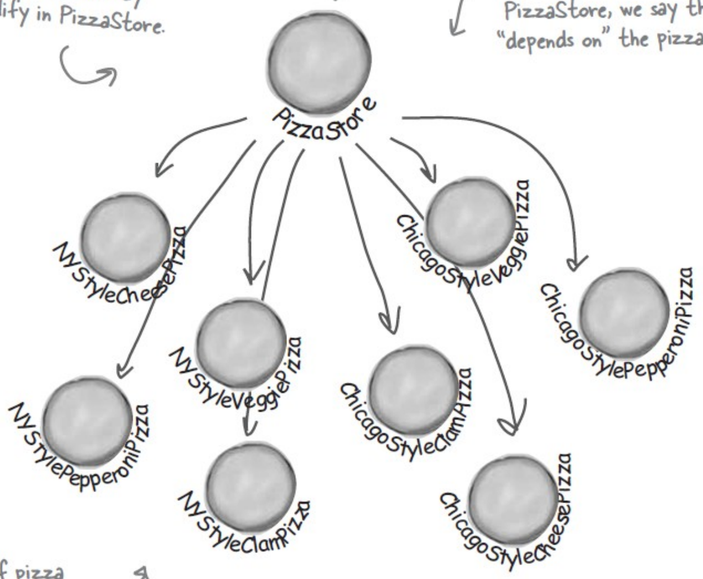
If the implementation of these classes change, then we may have to modify in PizzaStore.

Because any changes to the concrete implementations of pizzas affects the PizzaStore, we say that the PizzaStore "depends on" the pizza implementations.

Looking at object dependencies

When you directly instantiate an object, you are depending on its concrete class. Take a look at our very dependent PizzaStore one page back. It creates all the pizza objects right in the PizzaStore class instead of delegating to a factory.

If we draw a diagram representing that version of the PizzaStore and all the objects it depends on, here's what it looks like:



Every new kind of pizza we add creates another dependency for PizzaStore.

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
```

The Dependency Inversion Principle

It should be pretty clear that reducing dependencies to concrete classes in our code is a "good thing." In fact, we've got an OO design principle that formalizes this notion; it even has a big, formal name: *Dependency Inversion Principle*.

Yet another phrase you can use to impress the execs in the room! Your raise will more than offset the cost of this book, and you'll gain the admiration of your fellow developers.

Here's the general principle:

Design Principle

Depend upon abstractions. Do not depend upon concrete classes.

At first, this principle sounds a lot like "Program to an interface, not an implementation," right? It is similar; however, the Dependency Inversion Principle makes an even stronger statement about abstraction. It suggests that our high-level components should not depend on our low-level components; rather, they should *both* depend on abstractions.

A "high-level" component is a class with behavior defined in terms of other, "low level" components.

For example, PizzaStore is a high-level component because its behavior is defined in terms of pizzas – it creates all the different pizza objects, prepares, bakes, cuts, and boxes them, while the pizzas it uses are low-level components.

But what the heck does that mean?

Well, let's start by looking again at the pizza store diagram on the previous page. PizzaStore is our "high-level component" and the pizza implementations are our "low-level components," and clearly the PizzaStore is dependent on the concrete pizza classes.

Now, this principle tells us we should instead write our code so that we are depending on abstractions, not concrete classes. That goes for both our high level modules and our low-level modules.

But how do we do this? Let's think about how we'd apply this principle to our Very Dependent PizzaStore implementation...

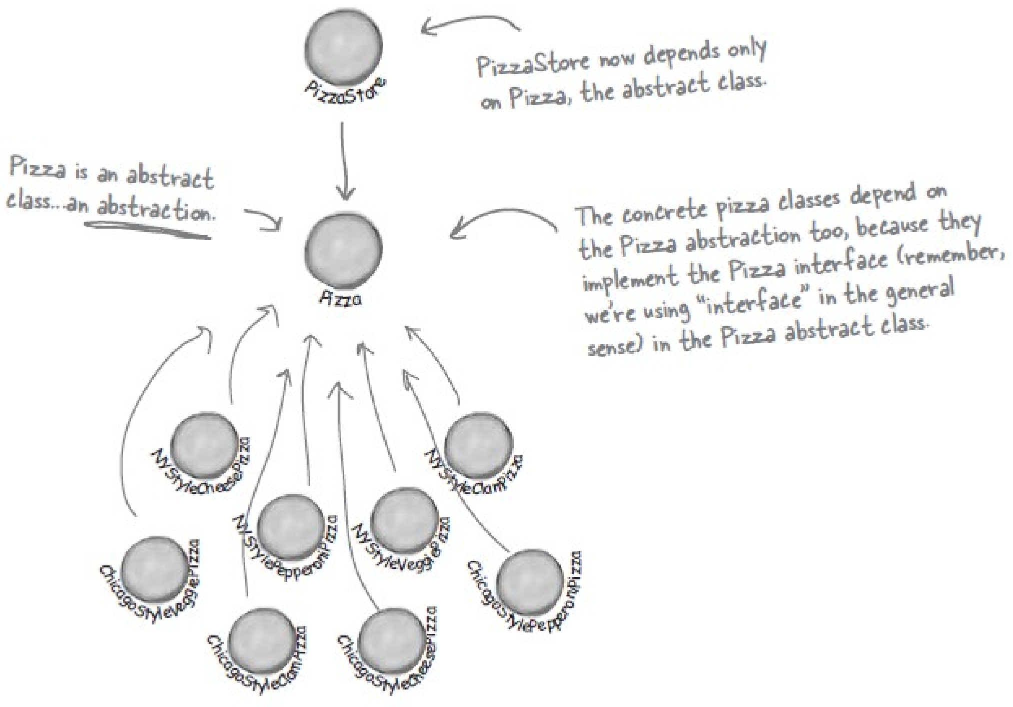
Applying the Principle

Now, the main problem with the Very Dependent PizzaStore is that it depends on every type of pizza because it actually instantiates concrete types in its orderPizza() method.

While we've created an abstraction, Pizza, we're nevertheless creating concrete Pizzas in this code, so we don't get a lot of leverage out of this abstraction.

How can we get those instantiations out of the orderPizza() method? Well, as we know, the Factory Method allows us to do just that.

So, after we've applied the Factory Method, our diagram looks like this:


PizzaStore
PizzaStore now depends only on Pizza, the abstract class.
Pizza is an abstract class...an abstraction.
Pizza
The concrete pizza classes depend on the Pizza abstraction too, because they implement the Pizza interface (remember, we're using "interface" in the general sense) in the Pizza abstract class.
NYStyleCheesePizza
NYStyleClamPizza
NYStylePepperoniPizza
NYStyleVeggiePizza
ChicagoStyleVeggiePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleCheesePizza

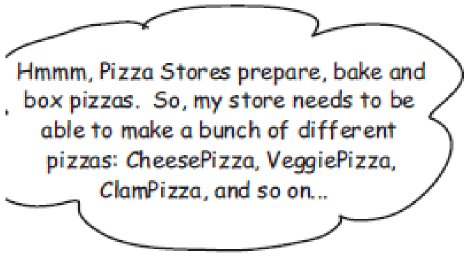
After applying the Factory Method, you'll notice that our high-level component, the PizzaStore, and our low-level components, the pizzas, both depend on Pizza, the abstraction. Factory Method is not the only technique for adhering to the Dependency Inversion Principle, but it is one of the more powerful ones.

Where's the "inversion" in Dependency Inversion Principle?

The "inversion" in the name Dependency Inversion Principle is there because it inverts the way you typically might think about your OO design. Look at the diagram on the previous page, notice that the low-level components now depend on a higher level abstraction. Likewise, the high-level component is also tied to the same abstraction. So, the top-to-bottom dependency chart we drew a couple of pages back has inverted itself, with both high-level and low-level modules now depending on the abstraction.

Let's also walk through the thinking behind the typical design process and see how introducing the principle can invert the way we think about the design...

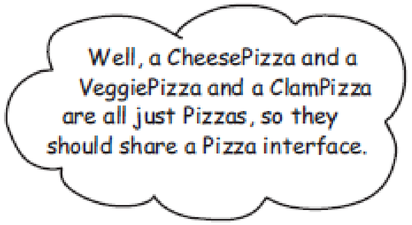
Inverting your thinking...


Hmmm, Pizza Stores prepare, bake and box pizzas. So, my store needs to be able to make a bunch of different pizzas: CheesePizza, VeggiePizza, ClamPizza, and so on...

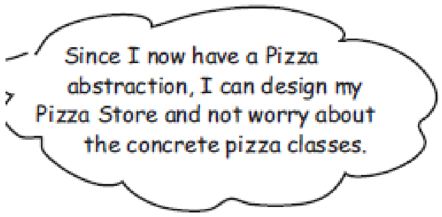
Okay, so you need to implement a PizzaStore. What's the first thought that pops into your head?

Right, you start at top and follow things down to the concrete classes. But, as you've seen, you don't want your store to know about the concrete pizza types, because then it'll be dependent on all those concrete classes!

Now, let's "invert" your thinking... instead of starting at the top, start at the Pizzas and think about what you can abstract.


Well, a CheesePizza and a VeggiePizza and a ClamPizza are all just Pizzas, so they should share a Pizza interface.

Right! You are thinking about the abstraction *Pizza*. So now, go back and think about the design of the Pizza Store again.


Since I now have a Pizza abstraction, I can design my Pizza Store and not worry about the concrete pizza classes.

Close. But to do that you'll have to rely on a factory to get those concrete classes out of your Pizza Store. Once you've done that, your different concrete pizza types depend only on an abstraction and so does your store. We've taken a design where the store depended on concrete classes and inverted those dependencies (along with your thinking).

A few guidelines to help you follow the Principle...

The following guidelines can help you avoid OO designs that violate the Dependency Inversion Principle:

- No variable should hold a reference to a concrete class.

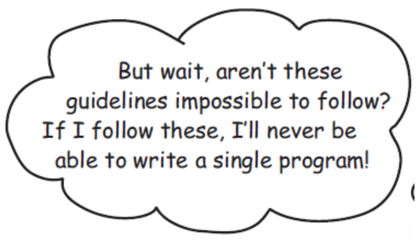
If you use **new**, you'll be holding a reference to a concrete class. Use a factory to get around that!

- No class should derive from a concrete class.

If you derive from a concrete class, you're depending on a concrete class. Derive from an abstraction, like an interface or an abstract class.

- No method should override an implemented method of any of its base classes.

If you override an implemented method, then your base class wasn't really an abstraction to start with. Those methods implemented in the base class are meant to be shared by all your subclasses.


But wait, aren't these guidelines impossible to follow? If I follow these, I'll never be able to write a single program!

You're exactly right! Like many of our principles, this is a guideline you should strive for, rather than a rule you should follow all the time. Clearly, every single Java program ever written violates these guidelines!

But, if you internalize these guidelines and have them in the back of your mind when you design, you'll know when you are violating the principle and you'll have a good reason for doing so. For instance, if you have a class that isn't likely to change, and you know it, then it's not the end of the world if you instantiate a concrete class in your code. Think about it; we instantiate String objects all the time without thinking twice. Does that violate the principle? Yes. Is that okay? Yes. Why? Because String is very unlikely to change.

If, on the other hand, a class you write is likely to change, you have some good techniques like Factory Method to encapsulate that change.

Meanwhile, back at the PizzaStore...

The design for the PizzaStore is really shaping up: it's got a flexible framework and it does a good job of adhering to design principles.

Now, the key to Objectville Pizza's success has always been fresh, quality ingredients, and what you've discovered is that with the new framework your franchises have been following your *procedures*, but a few franchises have been substituting inferior ingredients in their pies to lower costs and increase their margins. You know you've got to do something, because in the long term this is going to hurt the Objectville brand!

Ensuring consistency in your ingredients

So how are you going to ensure each franchise is using quality ingredients? You're going to build a factory that produces them and ships them to your franchises!

Now there is only one problem with this plan: the franchises are located in different regions and what is red sauce in New York is not red sauce in Chicago. So, you have one set of ingredients that need to be shipped to New York and a *different* set that needs to shipped to Chicago. Let's take a closer look:

We've got the same product families (dough, sauce, cheese, veggies, meats) but different implementations based on region.

Families of ingredients...

New York uses one set of ingredients and Chicago another. Given the popularity of Objectville Pizza it won't be long before you also need to ship another set of regional ingredients to California, and what's next? Seattle?

For this to work, you are going to have to figure out how to handle families of ingredients.

All Objectville's Pizzas are made from the same components, but each region has a different implementation of those components.

Each family consists of a type of dough, a type of sauce, a type of cheese, and a seafood topping (along with a few more we haven't shown, like veggies and spices).

In total, these three regions make up ingredient families, with each region implementing a complete family of ingredients.

Building the ingredient factories

Now we're going to build a factory to create our ingredients; the factory will be responsible for creating each ingredient in the ingredient family. In other words, the factory will need to create dough, sauce, cheese, and so on... You'll see how we are going to handle the regional differences shortly.

Let's start by defining an interface for the factory that is going to create all our ingredients:

```
public interface PizzaIngredientFactory {

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

Here's what we're going to do:

1. Build a factory for each region. To do this, you'll create a subclass of PizzaIngredientFactory that implements each create method

2. Implement a set of ingredient classes to be used with the factory, like ReggianoCheese, RedPeppers, and ThickCrustDough. These classes can be shared among regions where appropriate.

3. Then we still need to hook all this up by working our new ingredient factories into our old PizzaStore code.

Building the New York ingredient factory

Okay, here's the implementation for the New York ingredient factory. This factory specializes in Marinara sauce, Reggiano Cheese, 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

Reworking the pizzas...

We've got our factories all fired up and ready to produce quality ingredients; now we just need to rework our Pizzas so they only use factory-produced ingredients. We'll start with our abstract Pizza class:

```
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.

Reworking the pizzas, continued...

Now that you've got an abstract Pizza to work from, it's time to create the New York and Chicago style Pizzas – only this time around they will get their ingredients straight from the factory. The franchisees' days of skimping on ingredients are over!

When we wrote the Factory Method code, we had a NYCheesePizza and a ChicagoCheesePizza class. If you look at the two classes, the only thing that differs is the use of regional ingredients. The pizzas are made just the same (dough + sauce + cheese). The same goes for the other pizzas: Veggie, Clam, and so on. They all follow the same preparation steps; they just have different ingredients.

So, what you'll see is that we really don't need two classes for each pizza; the ingredient factory is going to handle the regional differences for us. Here's the 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.

The Pizza code uses the factory it has been composed with to produce the ingredients used in the pizza. The ingredients produced depend on which factory we're using. The Pizza class doesn't care; it knows how to make pizzas. Now, it's decoupled from the differences in regional ingredients and can be easily reused when there are factories for the Rockies, the Pacific Northwest, and beyond.

```
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.

```
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.

Revisiting our pizza stores

We're almost there; we just need to make a quick trip to our franchise stores to make sure they are using the correct Pizzas. We also need to give them a reference to their local ingredient factories:

```
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!

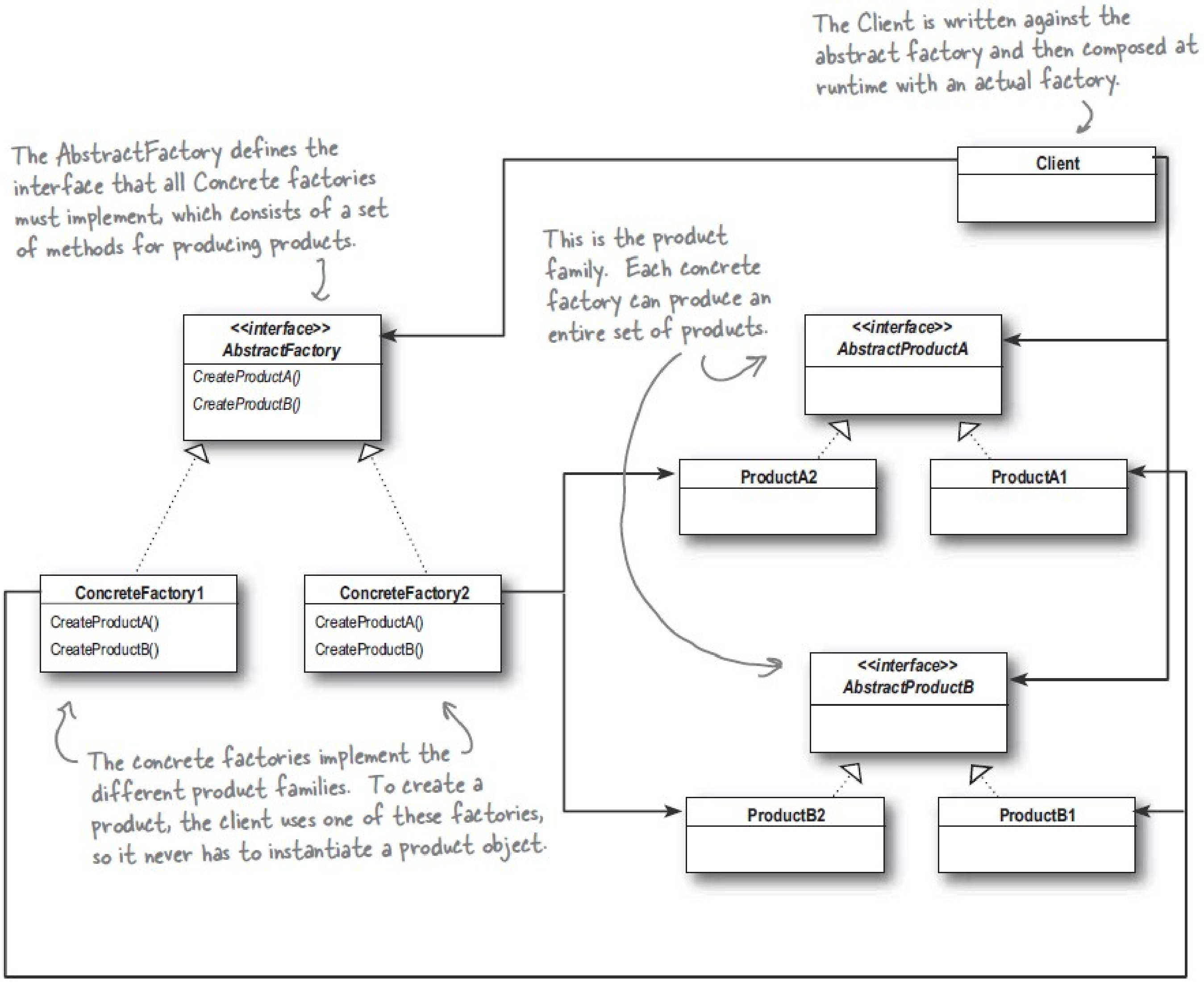
For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.

Abstract Factory Pattern defined

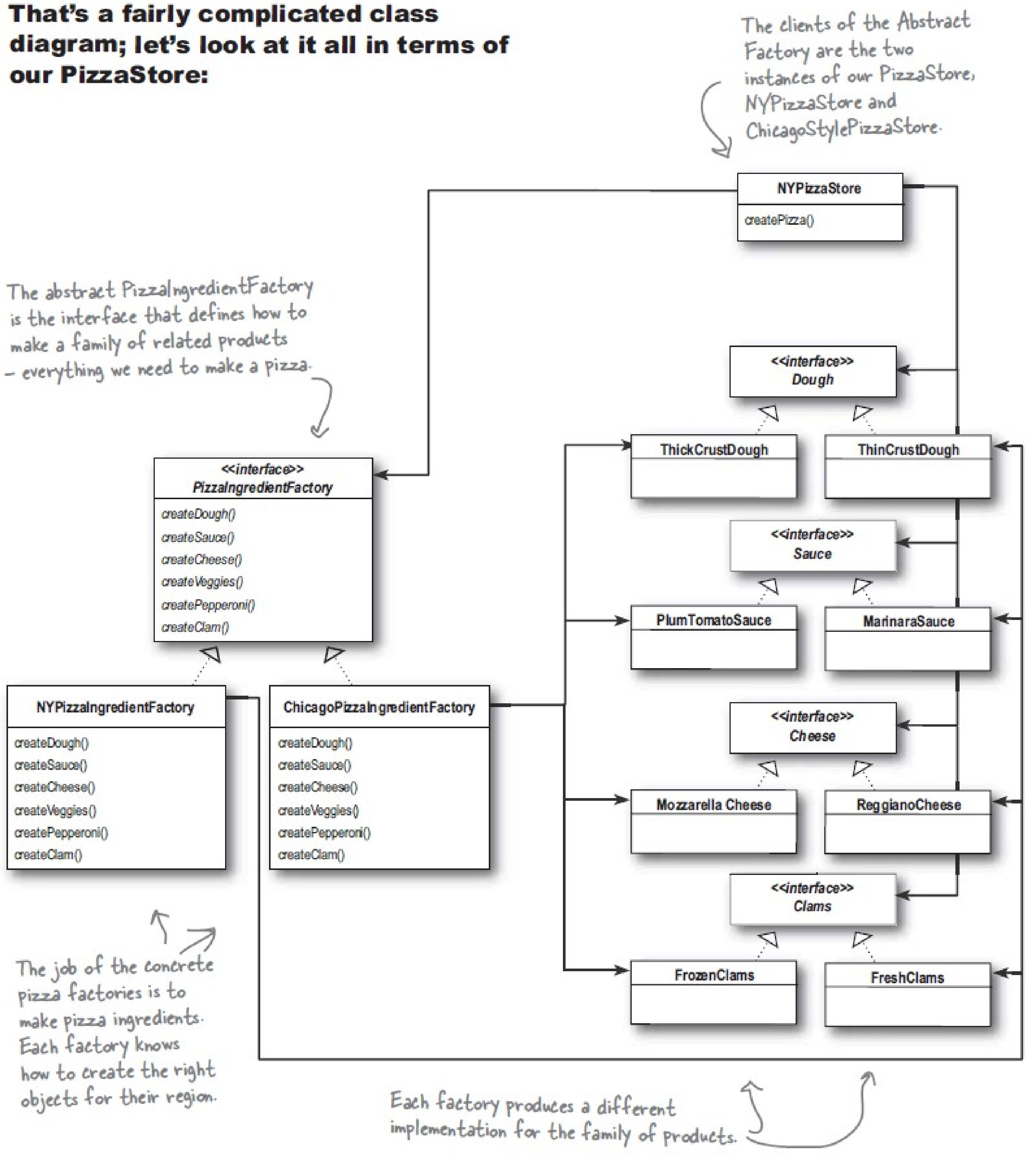
We're adding yet another factory pattern to our pattern family, one that lets us create families of products. Let's check out the official definition for this pattern:

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

We've certainly seen that Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products. Let's look at the class diagram to see how this all holds together:


The Client is written against the abstract factory and then composed at runtime with an actual factory.
The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.
This is the product family. Each concrete factory can produce an entire set of products.
Client
<<interface>> AbstractFactory
CreateProductA()
CreateProductB()
<<interface>> AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
<<interface>> AbstractProductB
ProductB2
ProductB1
The concrete factories implement the different product families. To create a product, the client uses one of these factories, so it never has to instantiate a product object.

That's a fairly complicated class diagram; let's look at it all in terms of our PizzaStore:


The clients of the Abstract Factory are the two instances of our PizzaStore, NYPizzaStore and ChicagoStylePizzaStore.
NYPizzaStore
createPizza()
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()
NYPizzaIngredientFactory
ChicagoPizzaIngredientFactory
<<interface>> Dough
ThickCrustDough
ThinCrustDough
<<interface>> Sauce
PlumTomatoSauce
MarinaraSauce
<<interface>> Cheese
Mozzarella Cheese
ReggianoCheese
<<interface>> Clams
FrozenClams
FreshClams
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.

Is that a Factory Method lurking inside the Abstract Factory?

Good catch! Yes, often the methods of an Abstract Factory are implemented as factory methods. It makes sense, right? The job of an Abstract Factory is to define an interface for creating a set of products. Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations. So, factory methods are a natural way to implement your product methods in your abstract factories.

The Difference Between The Two

The main difference between a "factory method" and an "abstract factory" is that the factory method is a single method, and an abstract factory is an object. I think a lot of people get these two terms confused, and start using them interchangeably. I remember that I had a hard time finding exactly what the difference was when I learnt them.

Because the factory method is just a method, it can be overridden in a subclass, hence the second half of your quote:

> ... the Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

The quote assumes that an object is calling *its own* factory method here. Therefore the only thing that could change the return value would be a subclass.

Difference between AbstractFactory and Factory design patters are as follows:

- **Factory Method** is used to create one product only but **Abstract Factory** is about creating families of related or dependent products.
- **Factory Method** pattern exposes a method to the client for creating the object whereas in case of **Abstract Factory** they expose a family of related objects which may consist of these Factory methods.
- **Factory Method** pattern hides the construction of single object where as **Abstract factory method** hides the construction of a family of related objects. Abstract factories are usually implemented using (a set of) factory methods.
- **AbstractFactory** pattern uses composition to delegate responsibility of creating object to another class while **Factory** design pattern uses inheritance and relies on derived class or sub class to create object.
- The idea behind the **Factory Method** pattern is that it allows for the case where a client doesn't know what concrete classes it will be required to create at runtime, but just wants to get a class that will do the job while **AbstractFactory** pattern is best utilised when your system has to create multiple families of products or you want to provide a library of products without exposing the implementation details.!

The main difference between Abstract Factory and Factory Method is that **Abstract Factory is implemented by Composition**; but **Factory Method is implemented by Inheritance**.

Yes, you read that correctly: the main difference between these two patterns is the old composition vs inheritance debate.

UML diagrams can be found in the (GoF) book. I want to provide code examples, because I think combining the examples from the top two answers in this thread will give a better demonstration than either answer alone. Additionally, I have used terminology from the book in class and method names.

Abstract Factory

1. The most important point to grasp here is that the abstract factory is *injected* into the client. This is why we say that Abstract Factory is implemented by Composition. Often, a dependency injection framework would perform that task; but a framework is not required for DI.
2. The second critical point is that the concrete factories here **are not** Factory Method implementations! Example code for Factory Method is shown further below.
3. And finally, the third point to note is the relationship between the products: in this case the outbound and reply queues. One concrete factory produces Azure queues, the other MSMQ. The GoF refers to this product relationship as a "family" and it's important to be aware that family in this case does not mean class hierarchy.

Real world example

> To create a kingdom we need objects with common theme. Elven kingdom needs an Elven king, Elven castle and Elven army whereas Orcish kingdom needs an Orcish king, Orcish castle and Orcish army. There is a dependency between the objects in the kingdom.

In plain words

> A factory of factories; a factory that groups the individual but related/dependent factories together without specifying their concrete classes.

Wikipedia says

> The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes

```
public interface Castle {
  String getDescription();
}
public interface King {
  String getDescription();
}
public interface Army {
  String getDescription();
}
```

```
// Elven implementations ->
public class ElfCastle implements Castle {
  static final String DESCRIPTION = "This is the Elven castle!";
  @Override
  public String getDescription() {
    return DESCRIPTION;
  }
}
public class ElfKing implements King {
  static final String DESCRIPTION = "This is the Elven king!";
  @Override
  public String getDescription() {
    return DESCRIPTION;
  }
}
public class ElfArmy implements Army {
  static final String DESCRIPTION = "This is the Elven Army!";
  @Override
  public String getDescription() {
    return DESCRIPTION;
  }
}

// Orcish implementations similarly...
```

```
public interface KingdomFactory {
  Castle createCastle();
  King createKing();
  Army createArmy();
}
```

```
public class ElfKingdomFactory implements KingdomFactory {
  public Castle createCastle() {
    return new ElfCastle();
  }
  public King createKing() {
    return new ElfKing();
  }
  public Army createArmy() {
    return new ElfArmy();
  }
}
```

```
public class OrcKingdomFactory implements KingdomFactory {
  public Castle createCastle() {
    return new OrcCastle();
  }
  public King createKing() {
    return new OrcKing();
  }
  public Army createArmy() {
    return new OrcArmy();
  }
}
```

```
  abstractFactoryDemo(new OrcKingdomFactory());
  abstractFactoryDemo(new ElfKingdomFactory());

}

private static void abstractFactoryDemo(KingdomFactory kingdomFactory) {
  System.out.println("\n----- Abstract Factory Pattern -----");
  System.out.println(kingdomFactory.getArmy().getDescription());
  System.out.println(kingdomFactory.getKing().getDescription());
  System.out.println(kingdomFactory.getCastle().getDescription());
```

```
----- Abstract Factory Pattern -----
Orc Army
Orc King
Orc Castle

----- Abstract Factory Pattern -----
Elven Army
Elven King
Elven Castle
```

Now, we can design a factory for our different kingdom factories. In this example, we created FactoryMaker, responsible for returning an instance of either ElfKingdomFactory or OrcKingdomFactory.
The client can use FactoryMaker to create the desired concrete factory which, in turn, will produce different concrete objects (Army, King, Castle).
In this example, we also used an enum to parameterize which type of kingdom factory the client will ask for.

```java
public class FactoryMaker {

  public enum KingdomType {
    ELF, ORC
  }

  public static KingdomFactory makeKingdomFactory(KingdomType type) {
    switch (type) {
    case ELF:
      return new ElfKingdomFactory();
    case ORC:
      return new OrcKingdomFactory();
    default:
      throw new IllegalArgumentException("KingdomType not supported.");
    }
  }
}
```

```java
KingdomFactory kingdomFactory = FactoryMaker.makeKingdomFactory(KingdomType.ELF);
System.out.println(kingdomFactory.getArmy().getDescription());
System.out.println(kingdomFactory.getKing().getDescription());
System.out.println(kingdomFactory.getCastle().getDescription());

kingdomFactory = FactoryMaker.makeKingdomFactory(KingdomType.ORC);
System.out.println(kingdomFactory.getArmy().getDescription());
System.out.println(kingdomFactory.getKing().getDescription());
System.out.println(kingdomFactory.getCastle().getDescription());
```

```
----- Abstract Factory Pattern -----
Elven Army
Elven King
Elven Castle
Orc Army
Orc King
Orc Castle
```

Applicability

Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed and represented
- a system should be configured with one of multiple families of products
- a family of related product objects is designed to be used together, and you need to enforce this constraint
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations
- the lifetime of the dependency is conceptually shorter than the lifetime of the consumer.
- you need a run-time value to construct a particular dependency
- you want to decide which product to call from a family at runtime.
- you need to supply one or more parameters only known at run-time before you can resolve a dependency.

Use Cases:

- Selecting to call the appropriate implementation of FileSystemAcmeService or DatabaseAcmeService or NetworkAcmeService at runtime.
- Unit test case writing becomes much easier