

“Factory method”

Problema: Pizzas!!!

Una pizzería nos pide realizar el diseño de su sistema, todas las pizzas que hace se preparan, se cocinan, se cortan y se ponen en una caja.



Primer diseño:

```
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.



¿Basta con hacer una 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.

¿Qué problema tiene este diseño?

¿Qué problema tiene este diseño?

La pizzería se dio cuenta que su competencia vende pizzas de almejas y vegetarianas por lo que quiere agregarlas a su menú, y además su pizza griega ya no se está vendiendo bien, por lo que quiere quitarla del menú.

```
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 VeggiePizza();  
    }  
  
    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.

Claramente, tratar con qué clase concreta se crea una instancia está realmente estropeando nuestro método `orderPizza()` e impidiendo que se cierre para su modificación. Pero ahora que sabemos qué varía y qué no, probablemente sea hora de encapsularlo.

Lo que vamos a hacer es tomar el código de creación y moverlo a otro objeto que solo se ocupará de crear pizzas.

Recuerden los principios de diseño:

- “Identifiquen los aspectos que cambian y sepárenlos de los que se quedan iguales”
- “Hay que crear código abierto a extensión pero cerrado a modificación”

```
Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

What's going to go here?

First we pull the object creation code out of the orderPizza Method

```
if (type.equals("cheese")) {  
    pizza = new CheesePizza();  
} else if (type.equals("pepperoni")) {  
    pizza = new PepperoniPizza();  
} else if (type.equals("clam")) {  
    pizza = new ClamPizza();  
} else if (type.equals("veggie")) {  
    pizza = new VeggiePizza();  
}
```

Then we place that code in an object that is only going to worry about how to create pizzas. If any other object needs a pizza created, this is the object to come to.



SimplePizzaFactory

Encapsulemos lo que cambia

Here's our new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

```
public class SimplePizzaFactory {  
    public Pizza createPizza(String type) {  
        Pizza pizza = null;  
  
        if (type.equals("cheese")) {  
            pizza = new CheesePizza();  
        } else if (type.equals("pepperoni")) {  
            pizza = new PepperoniPizza();  
        } else if (type.equals("clam")) {  
            pizza = new ClamPizza();  
        } else if (type.equals("veggie")) {  
            pizza = new VeggiePizza();  
        }  
        return pizza;  
    }  
}
```

First we define a createPizza() method in the factory. This is the method all clients will use to instantiate new objects.

Here's the code we plucked out of the orderPizza() method.

This code is still parameterized by the type of the pizza, just like our original orderPizza() method was.

Las fábricas manejan los detalles de la creación de objetos. Una vez que tenemos una SimplePizzaFactory, nuestro método orderPizza () simplemente se convierte en un cliente de ese objeto. Cada vez que necesita una pizza, le pide a la fábrica de pizza que haga una. Ahora, al método orderPizza() le importa que reciba una pizza, que implementa la interfaz Pizza para que pueda llamar a prepare(), bake(), cut() y box().

Now we give PizzaStore a reference to a SimplePizzaFactory.

```
public class PizzaStore {  
    SimplePizzaFactory factory;  
  
    public PizzaStore(SimplePizzaFactory factory) {  
        this.factory = factory;  
    }  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza;  
  
        pizza = factory.createPizza(type);  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
  
    // other methods here  
}
```

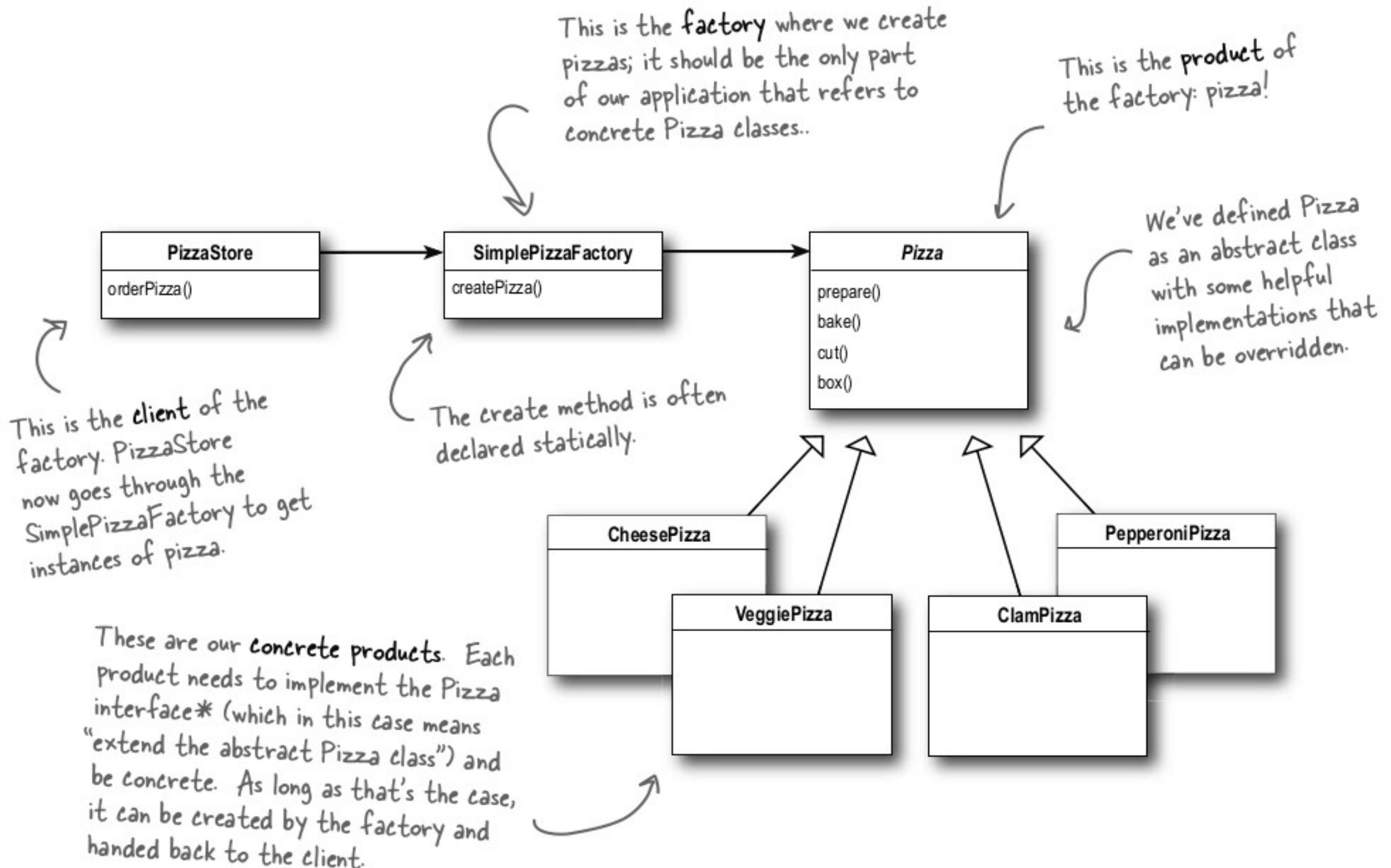
PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

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

“Simple Factory o Factory” algunos no lo consideran como un patrón de diseño; si no como un modismo de programación.

Algunos desarrolladores confunden este patrón con el Patrón de “Abstract factory”, por lo que la próxima vez que haya un silencio incómodo entre usted y otro desarrollador, tendrá un buen tema para romper el hielo.



¿Esto resuelve el problema?

¿Esto resuelve el problema?

- ✓ El código ahora encapsula las partes que cambian y las separa de las que son fijas.
- ✗ Sigue siendo abierto a modificación.

Más problemas

Otro problema

La pizzería va a extenderse y abrirá un nuevo local en Chicago, actualmente se encuentra en New York, pero requiere que las pizzas se hagan con el estilo del lugar, y que las de Chicago se hacen cuadradas y se cortan en cuadros más pequeños.

¿Puede el modelo hacer esto?

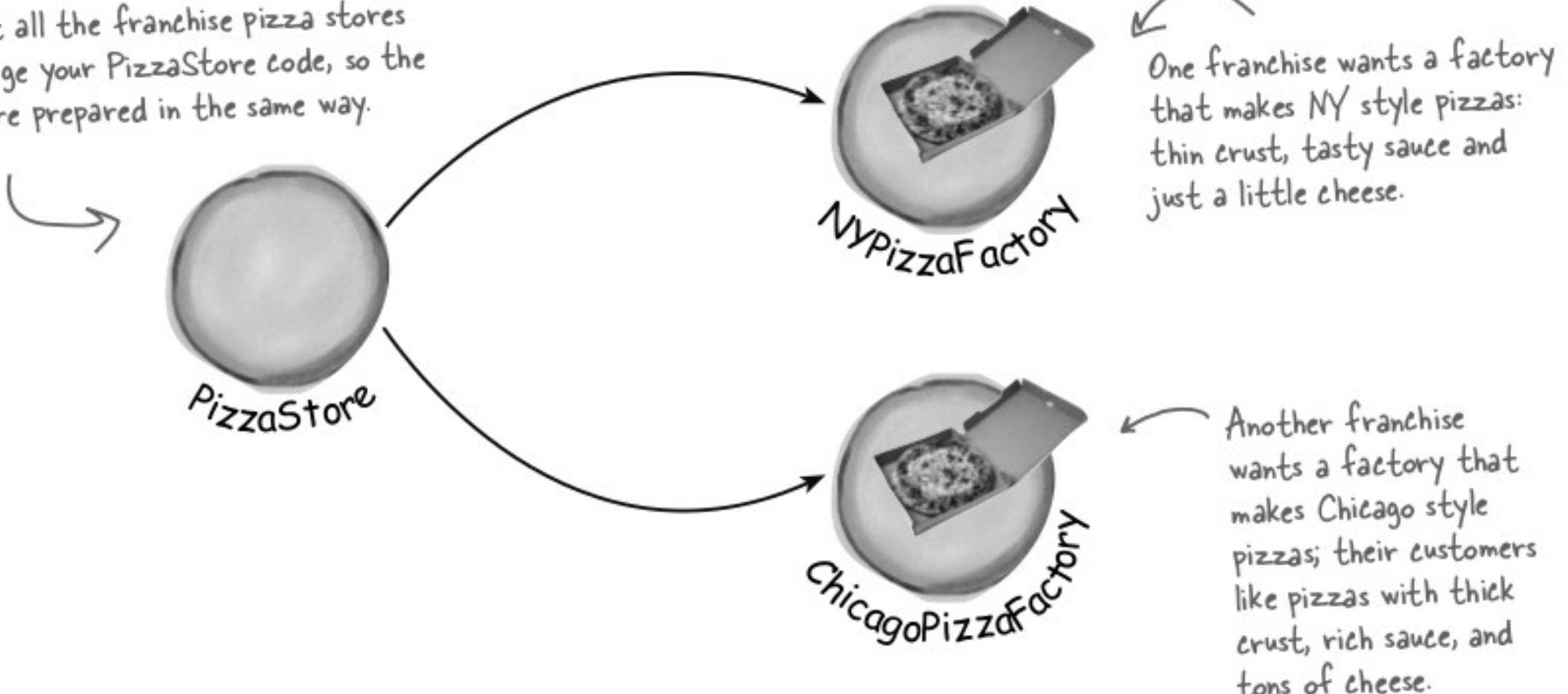
¿Puede el modelo hacer esto?

Al parecer nuestro modelo actual tampoco es tan abierto a extensión como parecía.

¿Qué necesitamos?

Queremos poder hacer nuevas “fabricas” de pizza sin cambiar las existentes.

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



PizzaStore is now abstract (see why below).



```
public abstract class PizzaStore {
```

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

```
        pizza = createPizza(type);
```

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

```
        return pizza;
```

```
}
```

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

```
}
```

Our "factory method" is now
abstract in PizzaStore.

Now createPizza is back to being a
call to a method in the PizzaStore
rather than on a factory object.

All this looks just the same...

Now we've moved our factory
object to this method.

`createPizza()` returns a `Pizza`, and the subclass is fully responsible for which concrete `Pizza` it instantiates

```
public class NYPizzaStore extends PizzaStore {  
    Pizza createPizza(String item) {  
        if (item.equals("cheese")) {  
            return new NYStyleCheesePizza();  
        } else if (item.equals("veggie")) {  
            return new NYStyleVeggiePizza();  
        } else if (item.equals("clam")) {  
            return new NYStyleClamPizza();  
        } else if (item.equals("pepperoni")) {  
            return new NYStylePepperoniPizza();  
        } else return null;  
    }  
}
```

The `NYPizzaStore` extends `PizzaStore`, so it inherits the `orderPizza()` method (among others).

← We've got to implement `createPizza()`, since it is abstract in `PizzaStore`.

← Here's where we create our concrete classes. For each type of `Pizza` we create the `NY` style.

* Note that the `orderPizza()` method in the superclass has no clue which `Pizza` we are creating; it just knows it can prepare, bake, cut, and box it!

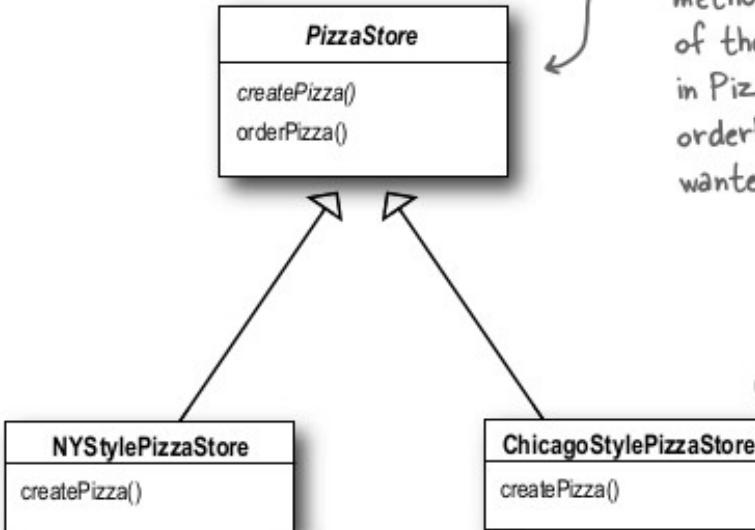
Ahora tenemos una tienda esperando subclases; tendremos una subclase para cada tipo regional (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) y cada una de las subclases tomará la decisión sobre lo que constituye una pizza.

Permitir que las subclases decidan

Recuerde, PizzaStore ya tiene un sistema de pedidos bien perfeccionado en el método `orderPizza()` y desea asegurarse de que sea coherente en todas las franquicias.

Lo que varía entre las PizzaStore regionales es el estilo de las pizzas que hacen - New York Pizza tiene una corteza delgada, Chicago Pizza tiene gruesas, y así sucesivamente - y vamos a aplicar **todas estas variaciones en el método createPizza ()** y hacerlo responsable de creando el tipo correcto de pizza. La forma en que hacemos esto es permitir que cada subclase de PizzaStore defina cómo se ve el método createPizza (). Por lo tanto, tendremos una cantidad de subclases concretas de PizzaStore, cada una con sus propias variaciones de pizza, que **se ajustarán al marco de PizzaStore y seguirán utilizando el método orderPizza ()** bien ajustado.

If a franchise wants NY style pizzas for its customers, it uses the NY subclass, which has its own `createPizza()` method, creating NY style pizzas.



Remember: `createPizza()` is abstract in `PizzaStore`, so all pizza store subtypes MUST implement the method.

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni")) {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam")) {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie")) {
```

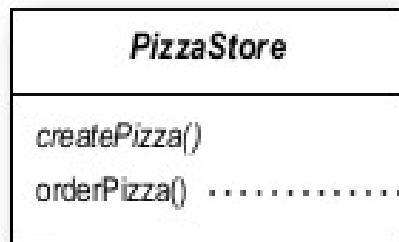
```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni")) {
        pizza = new ChicagoStylePepperoniPizza();
    } else if (type.equals("clam")) {
        pizza = new ChicagoStyleClamPizza();
    } else if (type.equals("veggie")) {
```

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.

Similarly, by using the Chicago subclass, we get an implementation of `createPizza()` with Chicago ingredients.



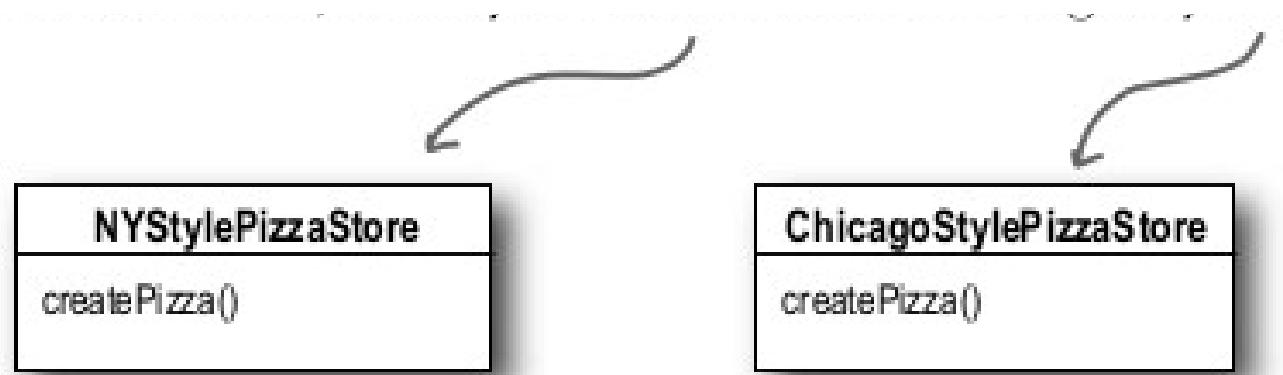
orderPizza() is defined in the abstract *PizzaStore*, not the subclasses. So, the method has no idea which subclass is actually running the code and making the pizzas.



```
 pizza = createPizza();
 pizza.prepare();
 pizza.bake();
 pizza.cut();
 pizza.box();
```

orderPizza() calls *createPizza()* to actually get a pizza object. But which kind of pizza will it get? The *orderPizza()* method can't decide; it doesn't know how. So who does decide?

Cuando `orderPizza()` llama a `createPizza()`, una de sus subclases se llamará para crear una pizza. ¿Qué tipo de pizza se hará? Bueno, eso es decidido por la elección de la tienda de pizzas que pides, `NYStylePizzaStore` o `ChicagoStylePizzaStore`.



Ejemplo

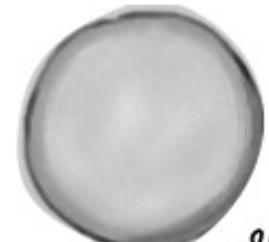
- 1) Primero, Joel y Ethan necesitan una instancia de PizzaStore. Joel necesita crear una instancia de ChicagoPizzaStore y Ethan necesita una NYPizzaStore.
- 2) Con PizzaStore en mano, Ethan y Joel llaman al método orderPizza () y pasan el tipo de pizza que desean (queso, verduras, etc.).
- 3) Para crear las pizzas, se llama al método createPizza (), que se define en las dos subclases NYPizzaStore y ChicagoPizzaStore. Tal como los definimos, NYPizzaStore crea una pizza de estilo NY, y ChicagoPizzaStore crea una pizza de estilo Chicago. En cualquier caso, la pizza se devuelve al método orderPizza().
- 4) El método orderPizza() no tiene idea de qué tipo de pizza se creó, pero sabe que es una pizza y la prepara, hornea, corta y empaqueta para Ethan y Joel.

1

Let's follow Ethan's order: first we need a NY PizzaStore:

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

Creates a instance of
NYPizzaStore.



2

Now that we have a store, we can take an order:

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

The orderPizza() method is called on
the nyPizzaStore instance (the method
defined inside PizzaStore runs).

nyPizzaStore

createPizza("cheese")

3

The orderPizza() method then calls the createPizza() method:

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

Remember, createPizza(), the factory
method, is implemented in the subclass. In
this case it returns a NY Cheese Pizza.



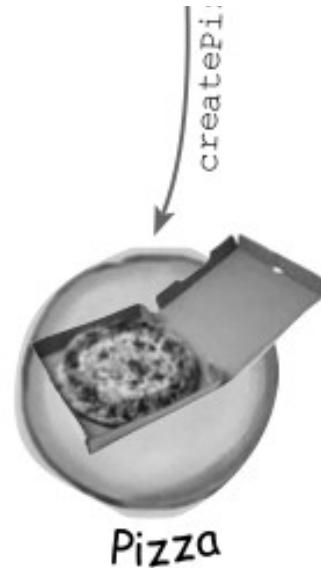
3

The orderPizza() method then calls the createPizza() method:

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



Remember, createPizza(), the factory method, is implemented in the subclass. In this case it returns a NY Cheese Pizza.



4

Finally we have the unprepared pizza in hand and the orderPizza() method finishes preparing it:

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

The orderPizza() method gets back a Pizza, without knowing exactly what concrete class it is.

All of these methods are defined in the specific pizza returned from the factory method createPizza(), defined in the NYPizzaStore.

We'll start with an abstract
Pizza class and all the concrete
pizzas will derive from this.

```
public abstract class Pizza {  
    String name;  
    String dough;  
    String sauce;  
    ArrayList toppings = new ArrayList();  
  
    void prepare() {  
        System.out.println("Preparing " + name);  
        System.out.println("Tossing dough...");  
        System.out.println("Adding sauce...");  
        System.out.println("Adding toppings: ");  
        for (int i = 0; i < toppings.size(); i++) {  
            System.out.println("    " + toppings.get(i));  
        }  
    }  
  
    void bake() {  
        System.out.println("Bake for 25 minutes at 350");  
    }  
  
    void cut() {  
        System.out.println("Cutting the pizza into diagonal slices");  
    }  
}
```

Each Pizza has a name, a type of dough, a
type of sauce, and a set of toppings.

The abstract class provides
some basic defaults for baking,
cutting and boxing.

Preparation follows a
number of steps in a
particular sequence.

Ahora solo necesitamos algunas subclases concretas ...
¿qué tal si definimos las pizzas de queso al estilo de Nueva York y Chicago?

```
public class NYStyleCheesePizza extends Pizza {  
    public NYStyleCheesePizza() {  
        name = "NY Style Sauce and Cheese Pizza";  
        dough = "Thin Crust Dough";  
        sauce = "Marinara Sauce";  
  
        toppings.add("Grated Reggiano Cheese");  
    }  
}
```

The NY Pizza has its own
marinara style sauce and thin crust.



And one topping, reggiano cheese!



```
public class ChicagoStyleCheesePizza extends Pizza {  
    public ChicagoStyleCheesePizza() {  
        name = "Chicago Style Deep Dish Cheese Pizza";  
        dough = "Extra Thick Crust Dough";  
        sauce = "Plum Tomato Sauce";  
  
        toppings.add("Shredded Mozzarella Cheese"); ←  
    }  
  
    void cut() {  
        System.out.println("Cutting the pizza into square slices");  
    }  
}
```

The Chicago Pizza uses plum tomatoes as a sauce along with extra thick crust.

The Chicago style deep dish pizza has lots of mozzarella cheese!

The Chicago style pizza also overrides the cut()
method so that the pieces are cut into squares.

```
public class PizzaTestDrive {  
    public static void main(String[] args) {  
        PizzaStore nyStore = new NYPizzaStore();  
        PizzaStore chicagoStore = new ChicagoPizzaStore();  
  
        Pizza pizza = nyStore.orderPizza("cheese");  
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");  
  
        pizza = chicagoStore.orderPizza("cheese");  
        System.out.println("Joel ordered a " + pizza.getName() + "\n");  
    }  
}
```

First we create two
different stores

Then use one store
to make Ethan's order.

And the other for Joel's.

File Edit Window Help YouWantMootzOnThatPizza?

%java PizzaTestDrive

Preparing NY Style Sauce and Cheese Pizza

Tossing dough...

Adding sauce...

Adding toppings:

Grated Regiano cheese

Bake for 25 minutes at 350

Cutting the pizza into diagonal slices

Place pizza in official PizzaStore box

Ethan ordered a NY Style Sauce and Cheese Pizza

Preparing Chicago Style Deep Dish Cheese Pizza

Tossing dough...

Adding sauce...

Adding toppings:

Shredded Mozzarella Cheese

Bake for 25 minutes at 350

Cutting the pizza into square slices

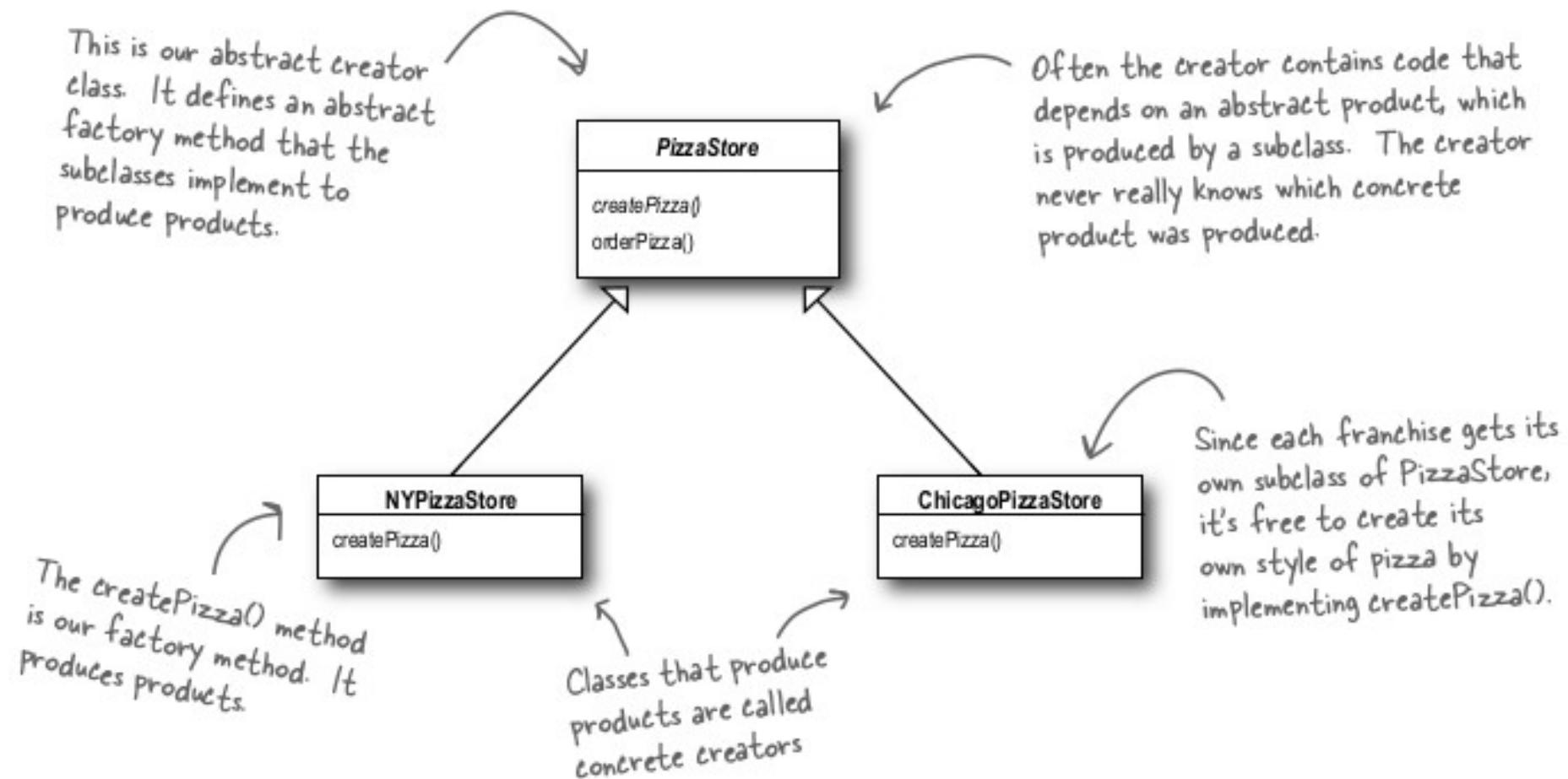
Place pizza in official PizzaStore box

Joel ordered a Chicago Style Deep Dish Cheese Pizza

Both pizzas get prepared,
the toppings added, and the
pizzas baked, cut and boxed.
Our superclass never had to
know the details, the subclass
handled all that just by
instantiating the right pizza.

El patrón de método de fábrica encapsula la creación de objetos permitiendo que las subclases decidan qué objetos crear.

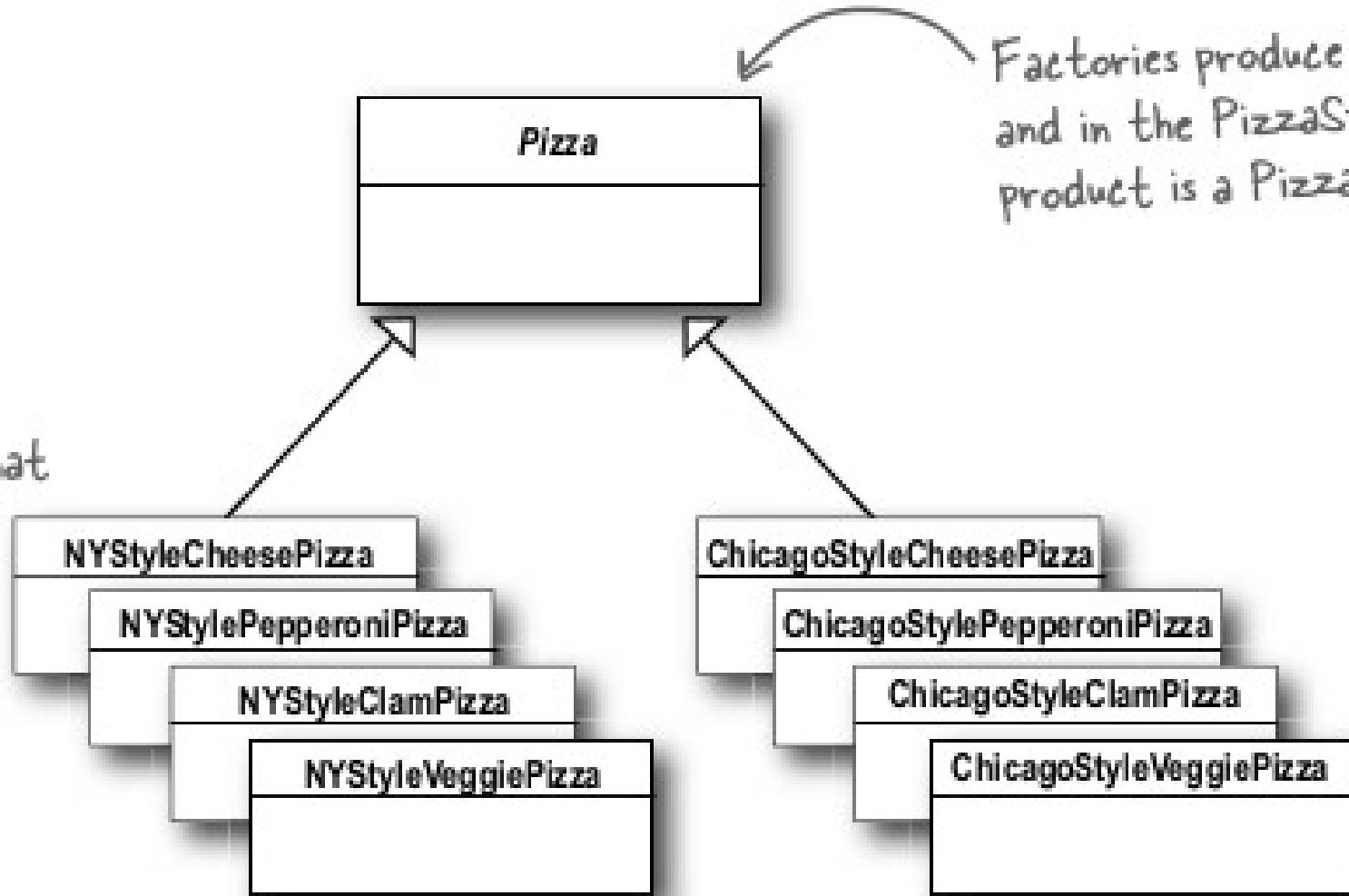
The Creator classes



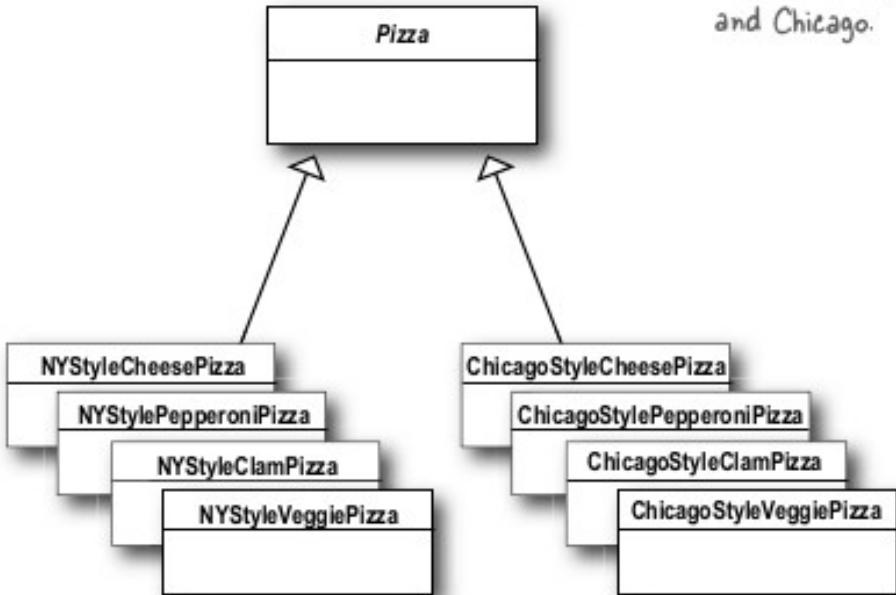
The Product classes

These are the concrete products – all the pizzas that are produced by our stores.

Factories produce products, and in the PizzaStore, our product is a Pizza.

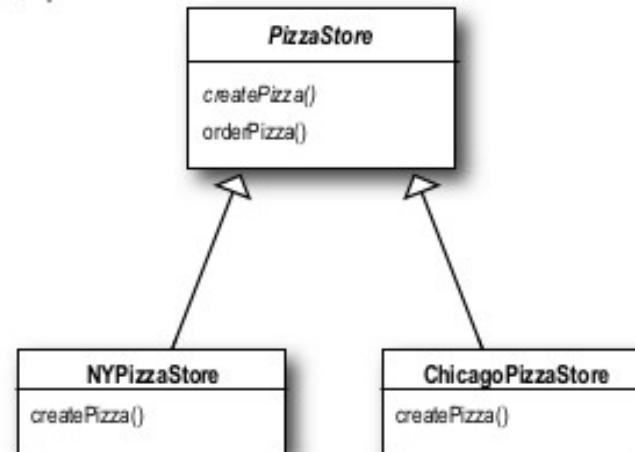


The Product classes



Notice how these class hierarchies are parallel: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago.

The Creator classes



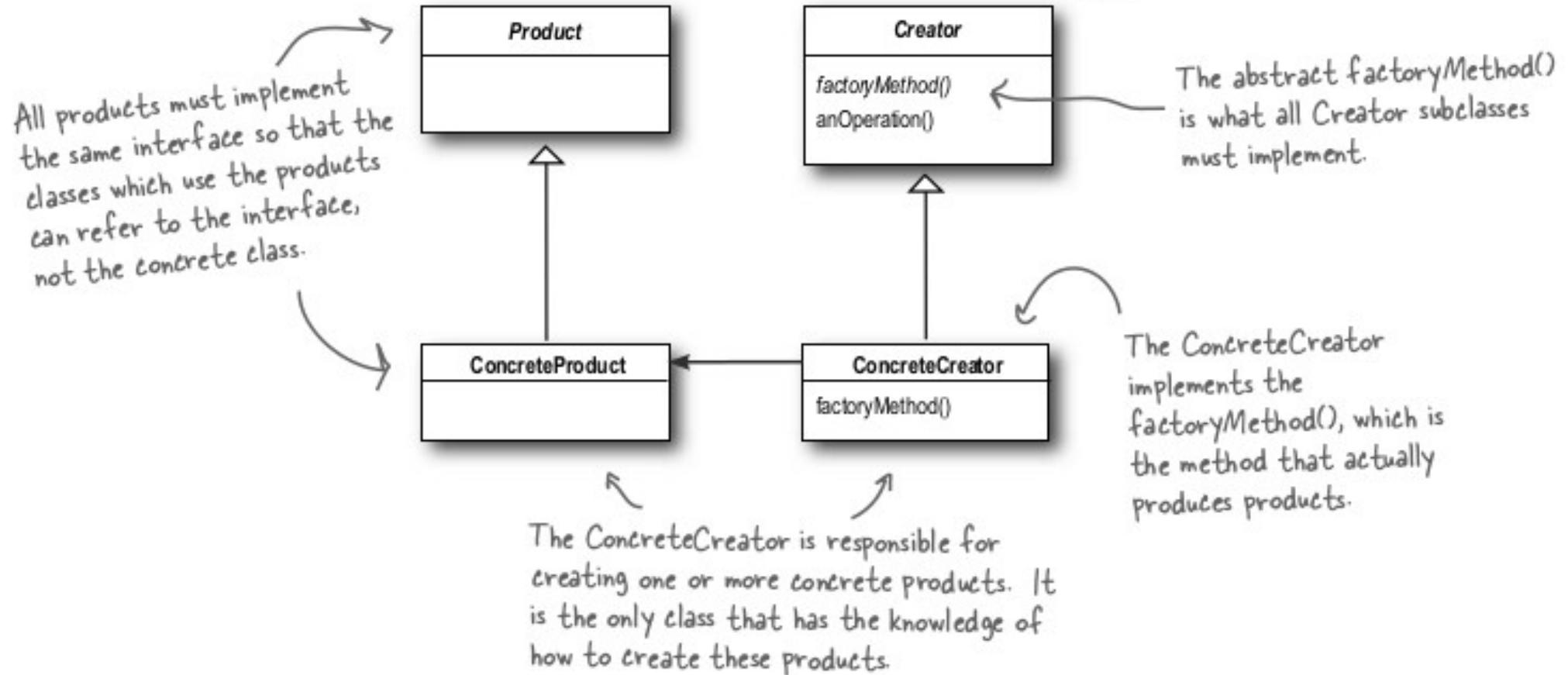
The **NYPizzaStore** encapsulates all the knowledge about how to make NY style pizzas.

The **ChicagoPizzaStore** encapsulates all the knowledge about how to make Chicago style pizzas.

The factory method is the key to encapsulating this knowledge.

FactoryMethod

El Patrón de Método de Fábrica define una interfaz para crear un objeto, pero permite que las subclases decidan qué clase instanciar. El método de fábrica permite que una clase difiera la instanciación a subclases.



En general

- Creamos una clase abstracta que es nuestra fábrica.
- Dentro de esta declaramos el método general donde se hacen las actividades en común, y dentro de este mismo método llamamos métodos de actividades que cambian.
- Los métodos cambiantes se declaran abstractos.
- Sus descendientes implementan estos métodos a su propia manera.
- Al llamar a algún descendiente de la fábrica abstracta solo llamamos al método general.

Implementación de alguien que no
escuchó la clase de hoy

```
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();  
            }  
        } else {  
            System.out.println("Error: invalid type of pizza");  
            return null;  
        }  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}
```

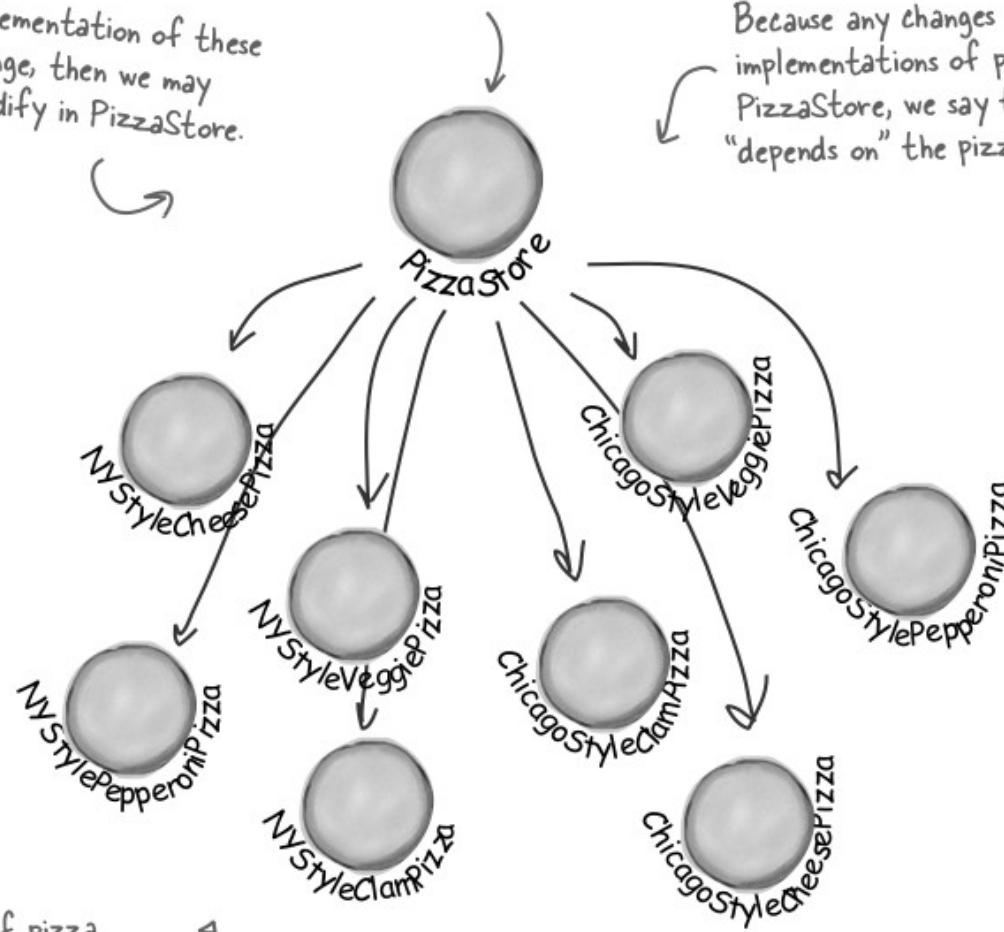
Handles all the NY
style pizzas

Handles all the
Chicago style
pizzas

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.



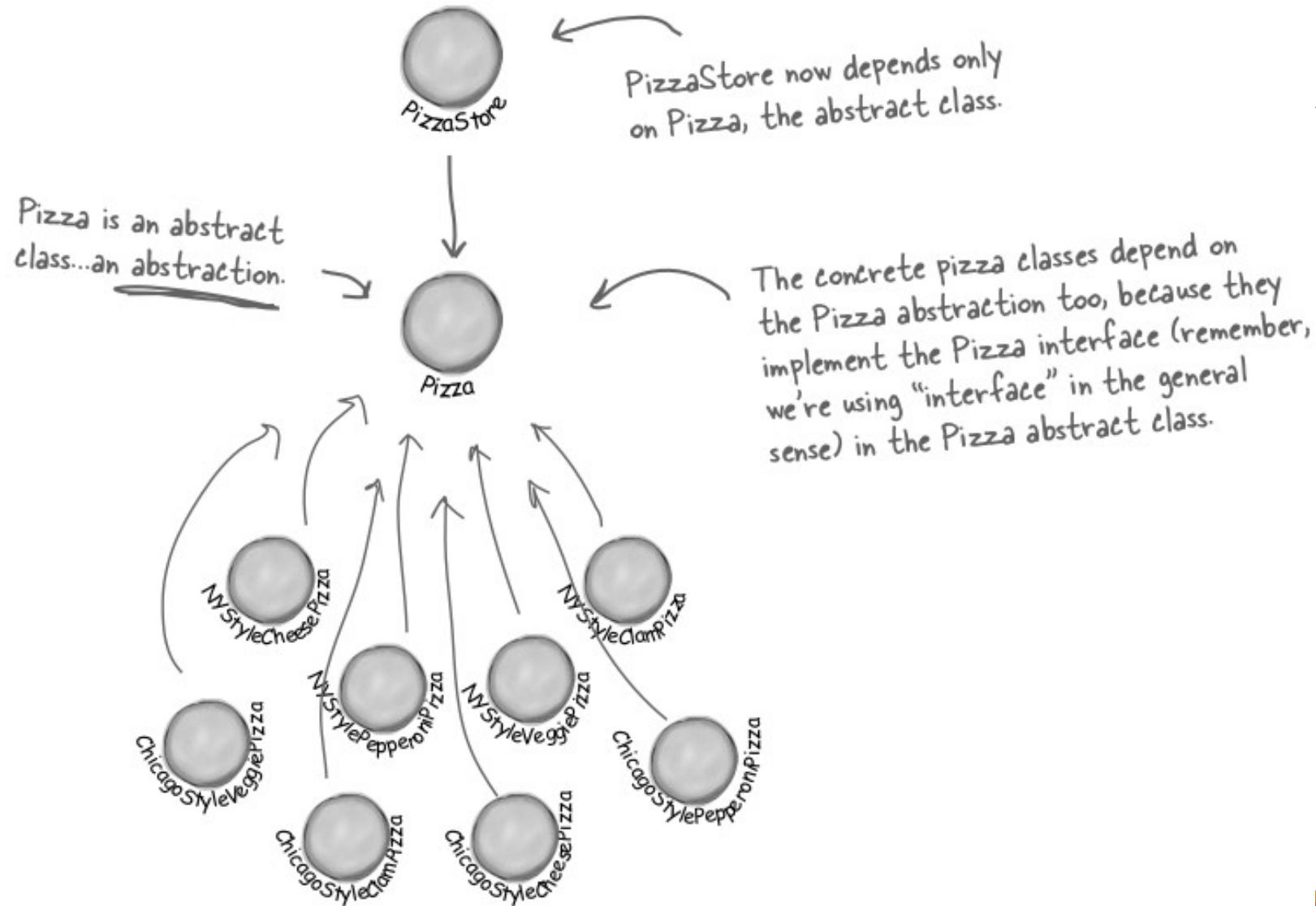
Every new kind of pizza we add creates another dependency for PizzaStore.

El Principio de Inversión de Dependencia

Debería estar bastante claro que reducir las dependencias a clases concretas en nuestro código es una "cosa buena". De hecho, tenemos un principio de diseño OO que formaliza esta noción; incluso tiene un gran nombre formal:

Depende de las abstracciones. No dependas de clases concretas

Aplicando el principio de diseño



Patrón de diseño “Abstract Factory”

Más problemas para la franquicia

Ahora, la clave del éxito de Objectville Pizza siempre ha sido los **ingredientes frescos y de calidad**, y lo que has descubierto es que con el nuevo marco tus franquicias han seguido tus procedimientos, pero algunas franquicias han estado **sustituyendo los ingredientes** por unos inferiores para reducirlos costos y aumentar sus márgenes. ¡Sabes que tienes que hacer algo, porque a largo plazo esto dañará la marca Objectville!

Asegurando consistencia en sus ingredientes

Entonces, ¿cómo vas a asegurarte de que cada franquicia utiliza ingredientes de calidad? ¡Vas a construir una fábrica que los produzca y los envíe a tus franquicias!

Ahora solo hay un problema con este plan: las franquicias están ubicadas en diferentes regiones y lo que es salsa en Nueva York no es la misma salsa en Chicago.

Por lo tanto, tiene un conjunto de ingredientes que deben enviarse a Nueva York y un conjunto diferente que debe enviarse a Chicago.



Chicago Pizza Menu

Cheese Pizza

Plum Tomato Sauce, Mozzarella, Parmesan, Oregano

Veggie Pizza

Plum Tomato sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives

Clam Pizza

Plum Tomato Sauce, Mozzarella, Parmesan, Clams

Pepperoni Pizza

Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives, Pepperoni

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



New York Pizza Menu



Cheese Pizza

Marinara Sauce, Reggiano, Garlic

Veggie Pizza

Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers

Clam Pizza

Marinara Sauce, Reggiano, Fresh Clams

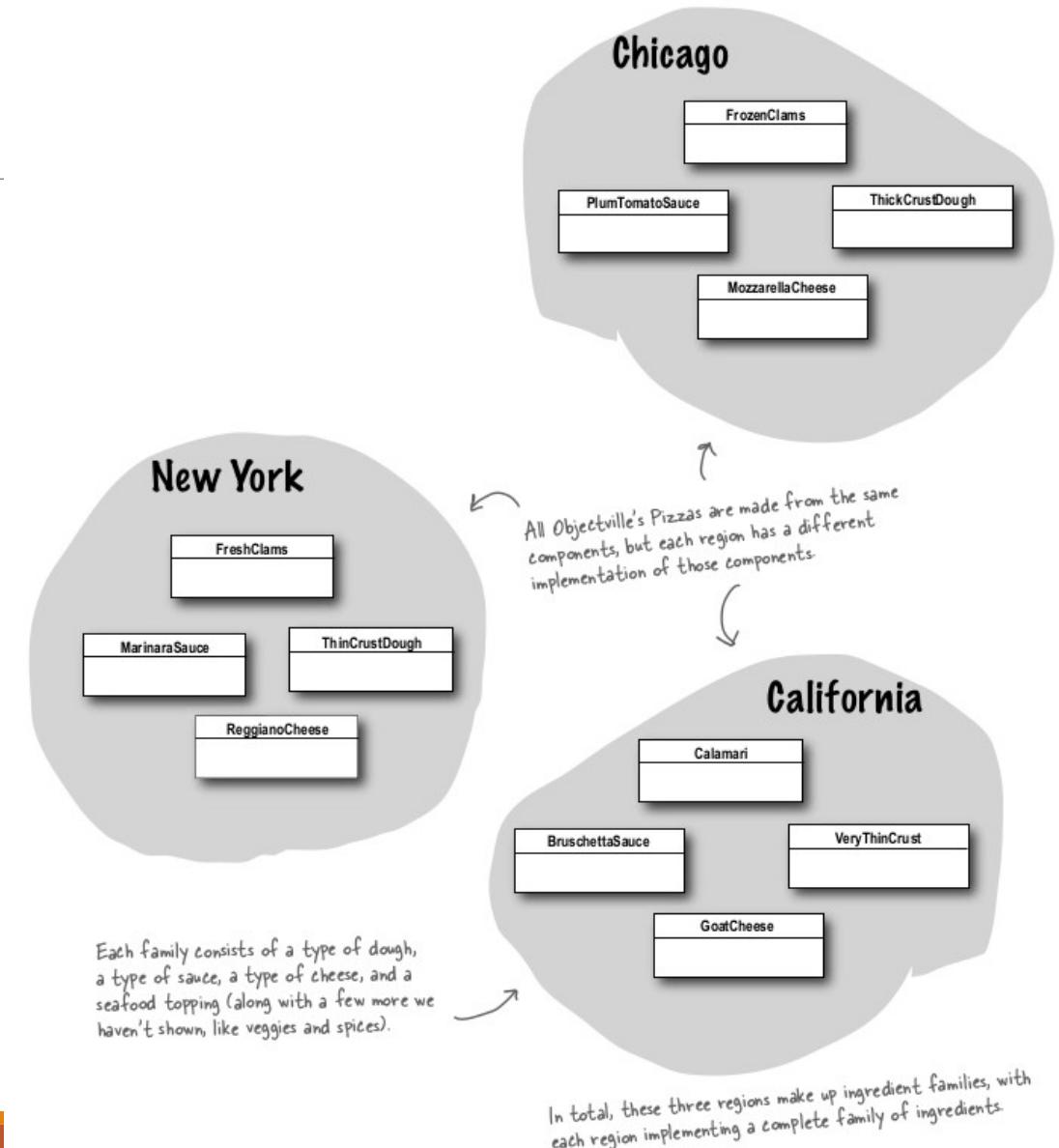
Pepperoni Pizza

Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers, Pepperoni

Familias de ingredientes ...

Nueva York usa un conjunto de ingredientes y Chicago otro.

Dada la popularidad de Objectville Pizza, no pasará mucho tiempo antes de que también necesite enviar otro conjunto de ingredientes regionales a California, y ¿qué sigue? ¿Seattle?



Construyendo las fábricas de ingredientes

Ahora vamos a construir una fábrica para crear nuestros ingredientes; la fábrica será responsable de crear cada ingrediente en la familia de ingredientes. En otras palabras, la fábrica necesitará crear masa, salsa, queso, etc. ... Verás cómo manejaremos las diferencias regionales

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

Lots of new classes here,
one per ingredient



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

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

1 Construye una fábrica para cada región. Para hacer esto, creará una subclase de PizzalngredientFactory que implementa cada método de creación

Construyendo la fábrica de ingredientes de Nueva York

```
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();  
    }  
}
```

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

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.

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.

2 Implemente un conjunto de clases de ingredientes para usar con la fábrica, como ReggianoCheese, RedPeppers y ThickCrustDough. Estas clases se pueden compartir entre regiones donde corresponda.

```
public interface Cheese {  
    public String toString();  
}  
  
public class ReggianoCheese implements Cheese {  
  
    public String toString() {  
        return "Reggiano Cheese";  
    }  
}
```

3 Entonces todavía tenemos que conectar todo esto en nuestras nuevas fábricas de ingredientes del antiguo código de PizzaStore.

Rehaciendo las pizzas

```
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.

Cuando escribimos el código de simple factory, tuvimos una clase NYCheesePizza y una ChicagoCheesePizza. Si miras las dos clases, lo único que difiere es el uso de ingredientes regionales. Las pizzas están hechas igual (masa + salsa + queso).

La fábrica de ingredientes manejará las diferencias regionales para nosotros

```
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.

Revisando nuestras tiendas de pizza

```
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.

¿Que hicimos?

Proporcionamos un medio para crear una familia de ingredientes para pizzas mediante la presentación de un nuevo tipo de fábrica llamada Abstract Factory.

Una fábrica abstracta nos da una interfaz para crear una familia de productos. Al escribir el código que usa esta interfaz, desacoplo nuestro código de la fábrica real que crea los productos.

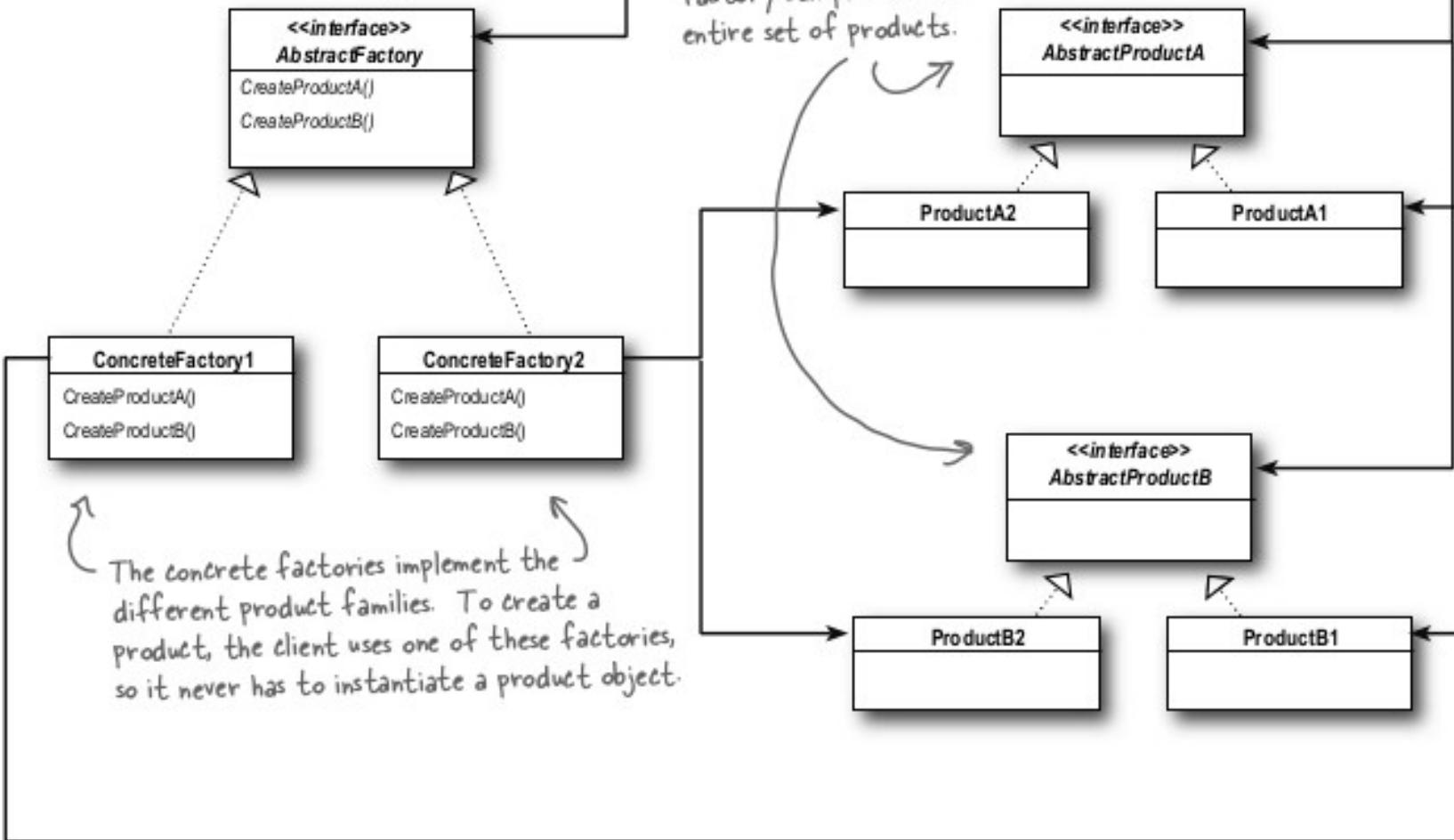
¿Que hicimos?

Eso nos permite implementar una variedad de fábricas que producen productos destinados a diferentes contextos, como diferentes regiones, diferentes sistemas operativos o diferentes aspectos.

Debido a que nuestro código está desacoplado de los productos reales, podemos sustituir diferentes fábricas para obtener diferentes comportamientos (como obtener masa delgada en lugar de gruesa).

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.

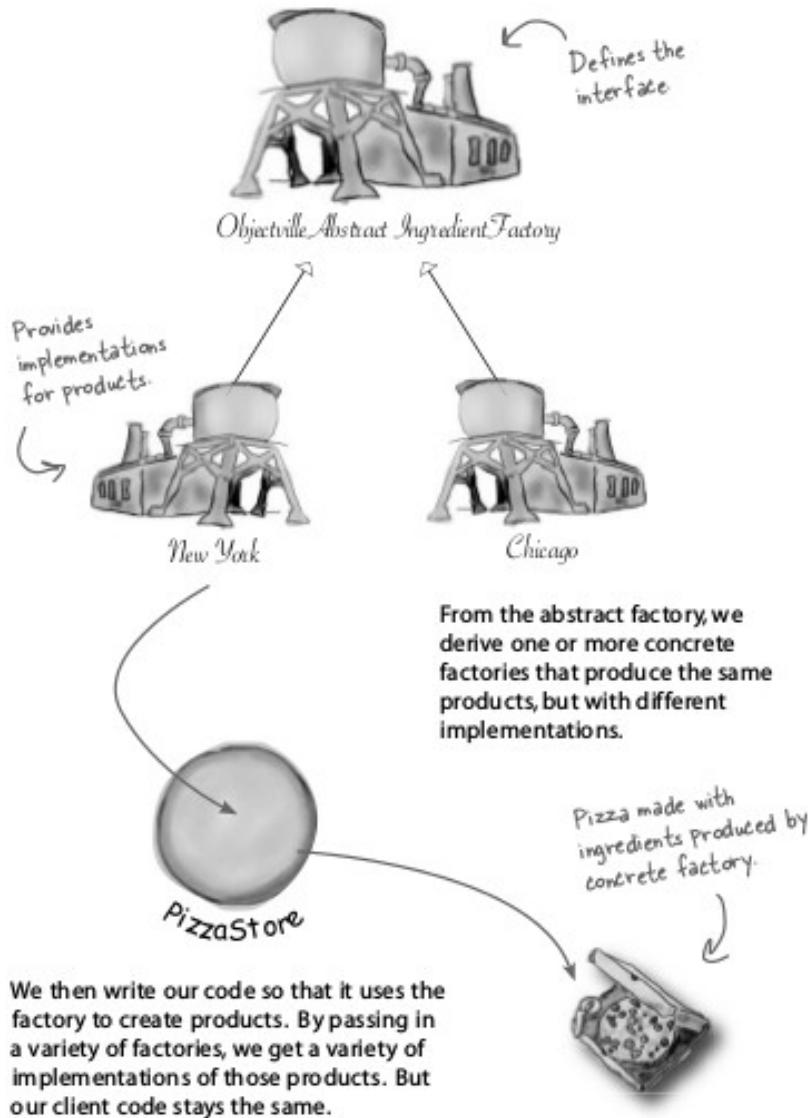


Una fábrica abstracta (Abstract Factory) proporciona una interfaz para una familia de productos. ¿Qué es una familia? En nuestro caso, son todas las cosas que necesitamos para hacer una pizza: masa, salsa, queso, carnes y verduras.

De la fábrica abstracta, derivamos una o más fábricas concretas que producen los mismos productos, pero con diferentes implementaciones.

Luego escribimos nuestro código para que use la fábrica para crear productos. Al pasar en una variedad de fábricas, obtenemos una variedad de implementaciones de esos productos. Pero nuestro código de cliente permanece igual.

An Abstract Factory provides an interface for a family of products. What's a family? In our case it's all the things we need to make a pizza: dough, sauce, cheese, meats and veggies.



Abstract Factory proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

