FACTORY METHOD PATTERN

Hongbin Lu

Design pizza creation

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;
}
```

```
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.

```
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 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.

Encapsulating object creation

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

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

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

What's going to go here?

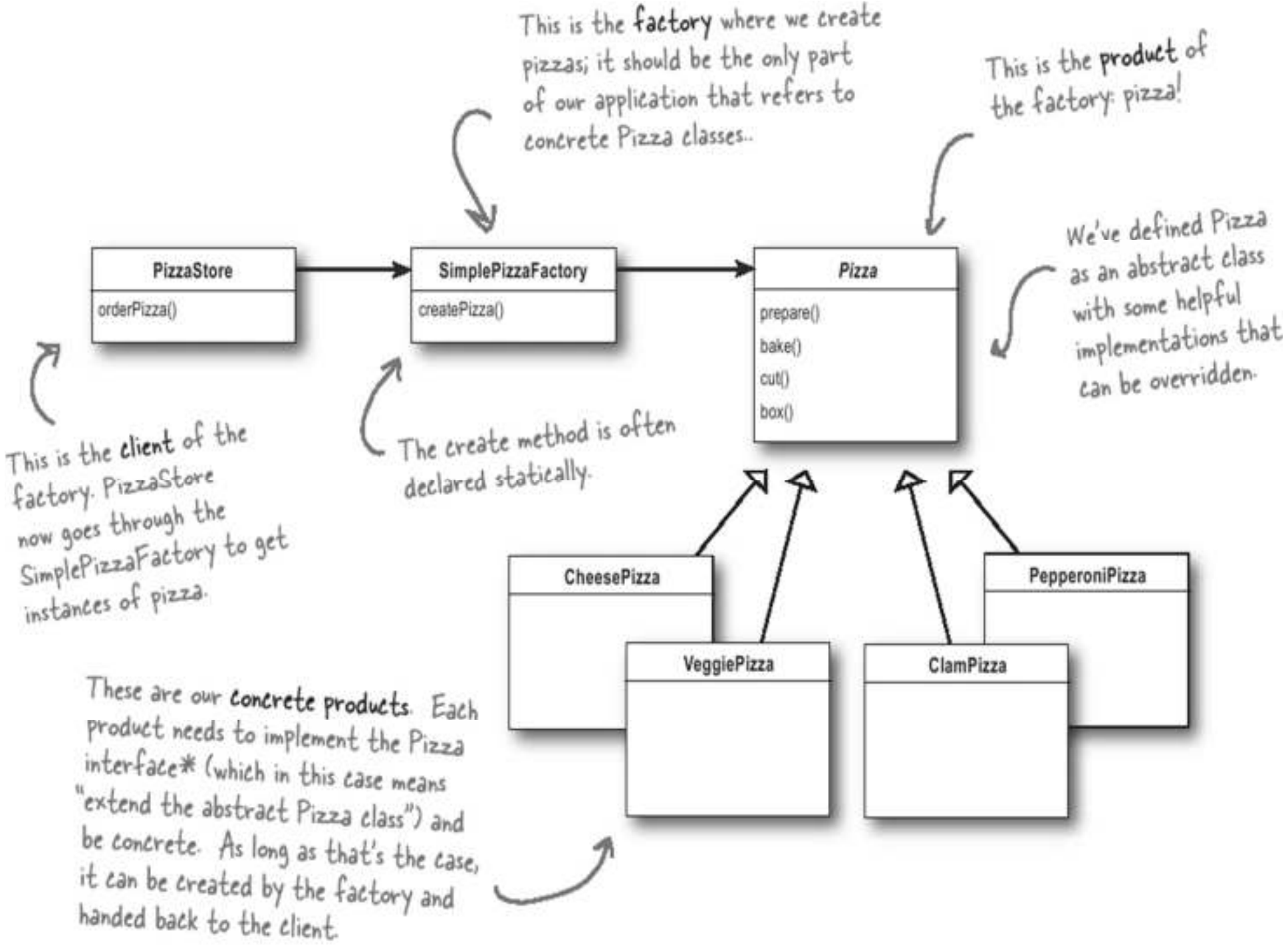
Where is the creational code?

- We should place the creational code in an object.
- That object is only going to worry about how to create pizzas.
- We call that “simple pizza factory”.

Our simple pizza factory

```
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;
    }
}
```

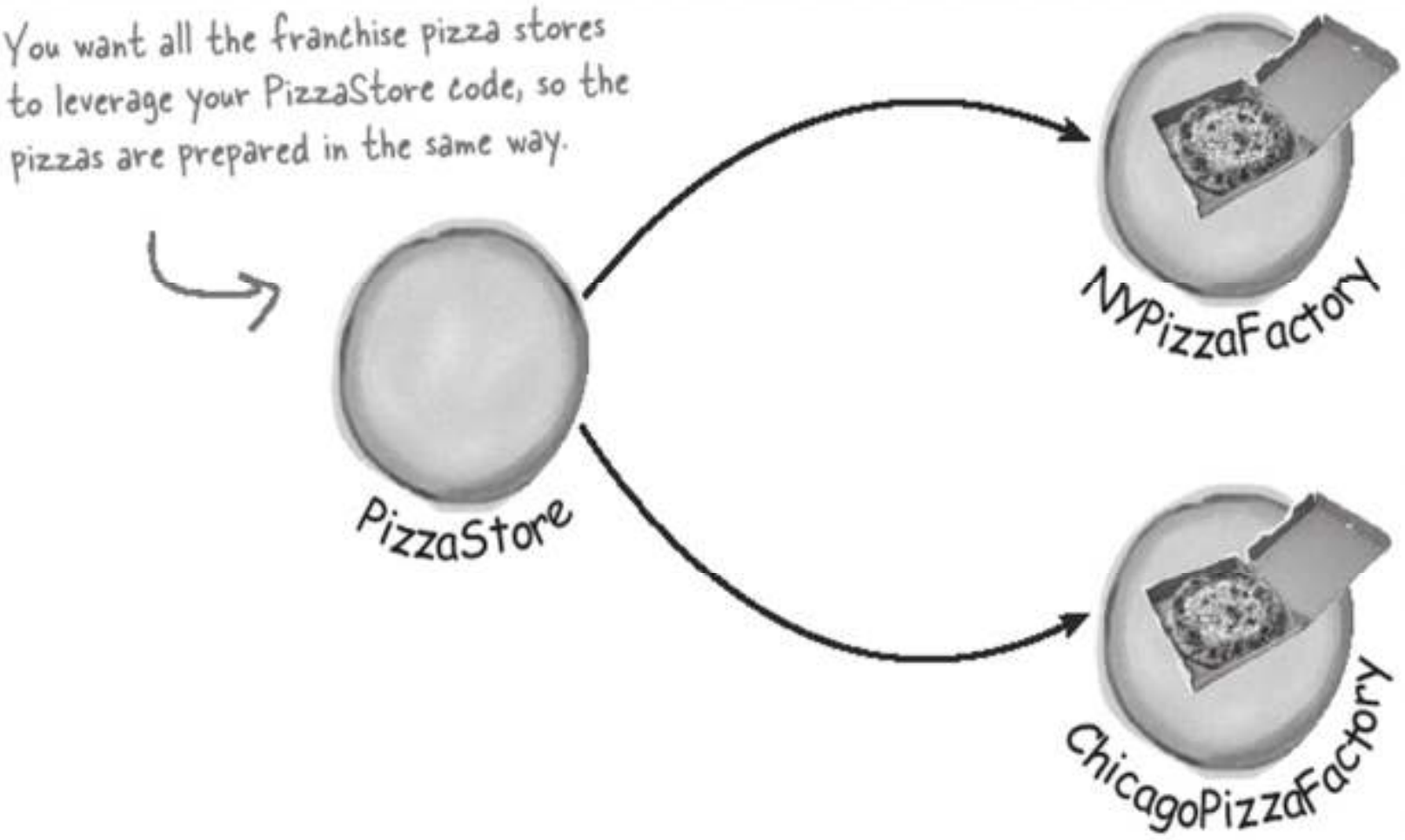
```
public class PizzaStore {
    SimplePizzaFactory factory;

    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;
    }
```


This is the factory where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes..
This is the product of the factory: pizza!
We've defined Pizza as an abstract class with some helpful implementations that can be overridden.
PizzaStore
orderPizza()
SimplePizzaFactory
createPizza()
Pizza
prepare()
bake()
cut()
box()
This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.
The create method is often declared statically.
CheesePizza
VeggiePizza
ClamPizza
PepperoniPizza
These are our concrete products. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

Franchising the pizza store

- NY style pizzas: thin crust, tasty sauce and just a little cheese.
- Chicago style pizzas: thick crust, rich sauce, and tons of cheese.
- We need two factories to produce different styles of pizzas.

Franchising the pizza store



Franchising the pizza store

New York:

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

Chicago:

```
ChicagoPizzaFactory chiFac = new ChicagoPizzaFactory();
PizzaStore chiStore = new PizzaStore(chiFac);
chiStore.order("Veggie");
```

A framework for pizza store

```
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);
}
```

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

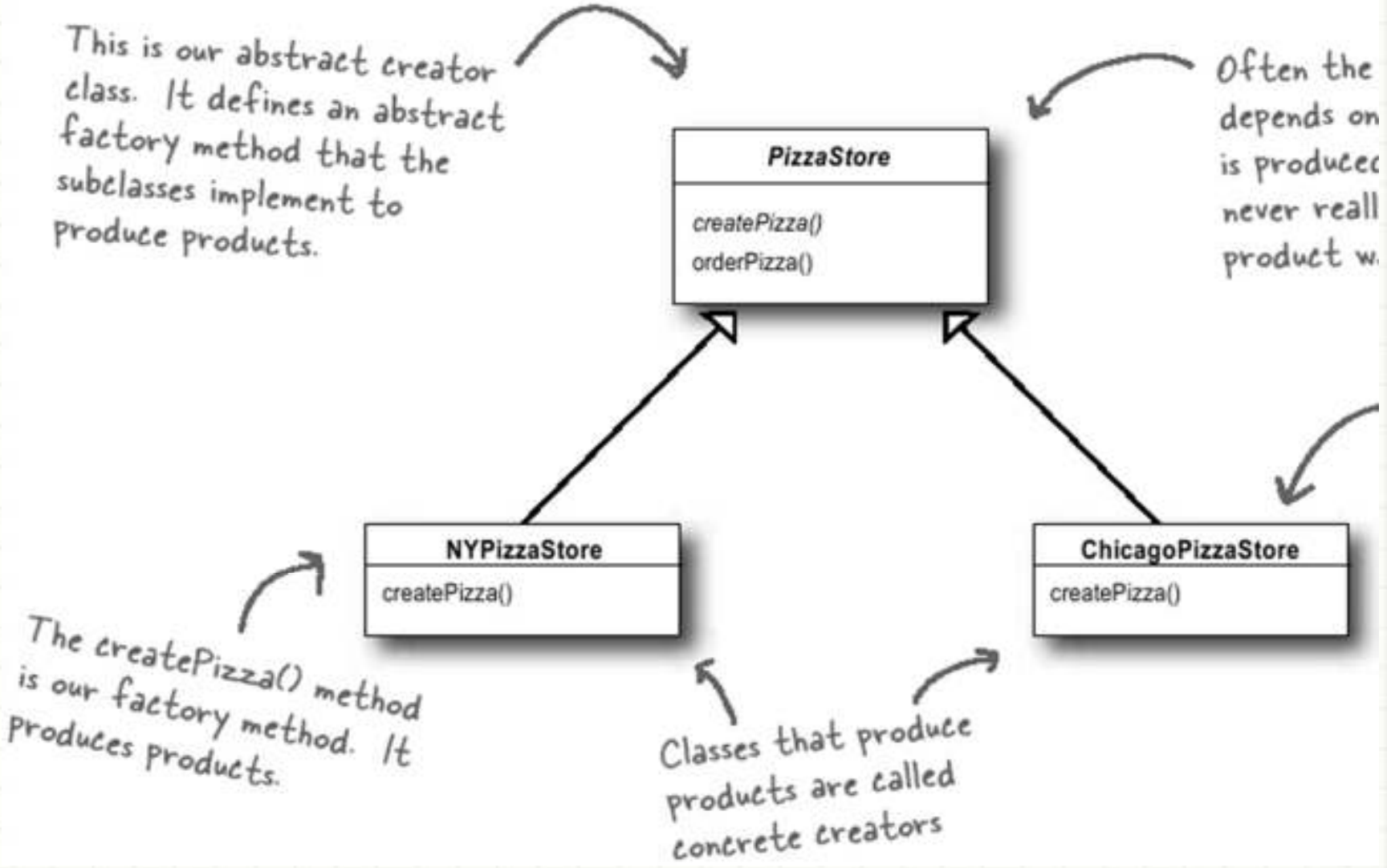
All this looks just the same...

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

Let's make a PizzaStore

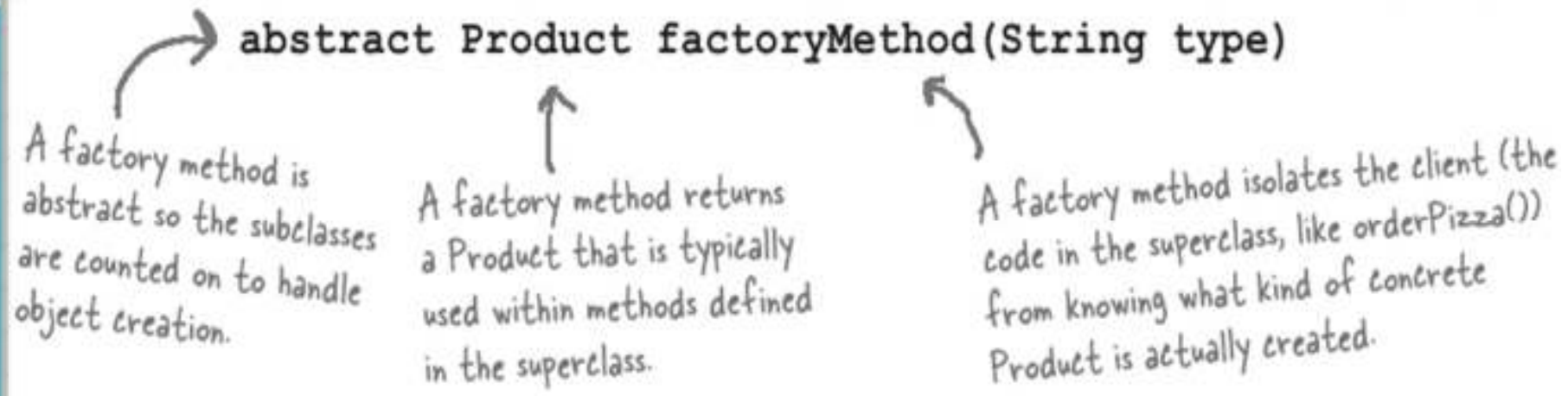
```
public class NYPizzaStore extends PizzaStore {
    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {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

The Creator classes



Look closer to the factory method

```
Public abstract class  PizzaStore {
   …
   Protected abstract Pizza createPizza(String type);
   ….
}
```



The Pizza class

```
Public abstract class Pizza {
   String name;
   String dough;
   String sauce;
   ArrayList toppins = new ArrayList();

   void prepare() {…}
   void bake() {…}
   void cut() {…}
   void box() {…}
}
```

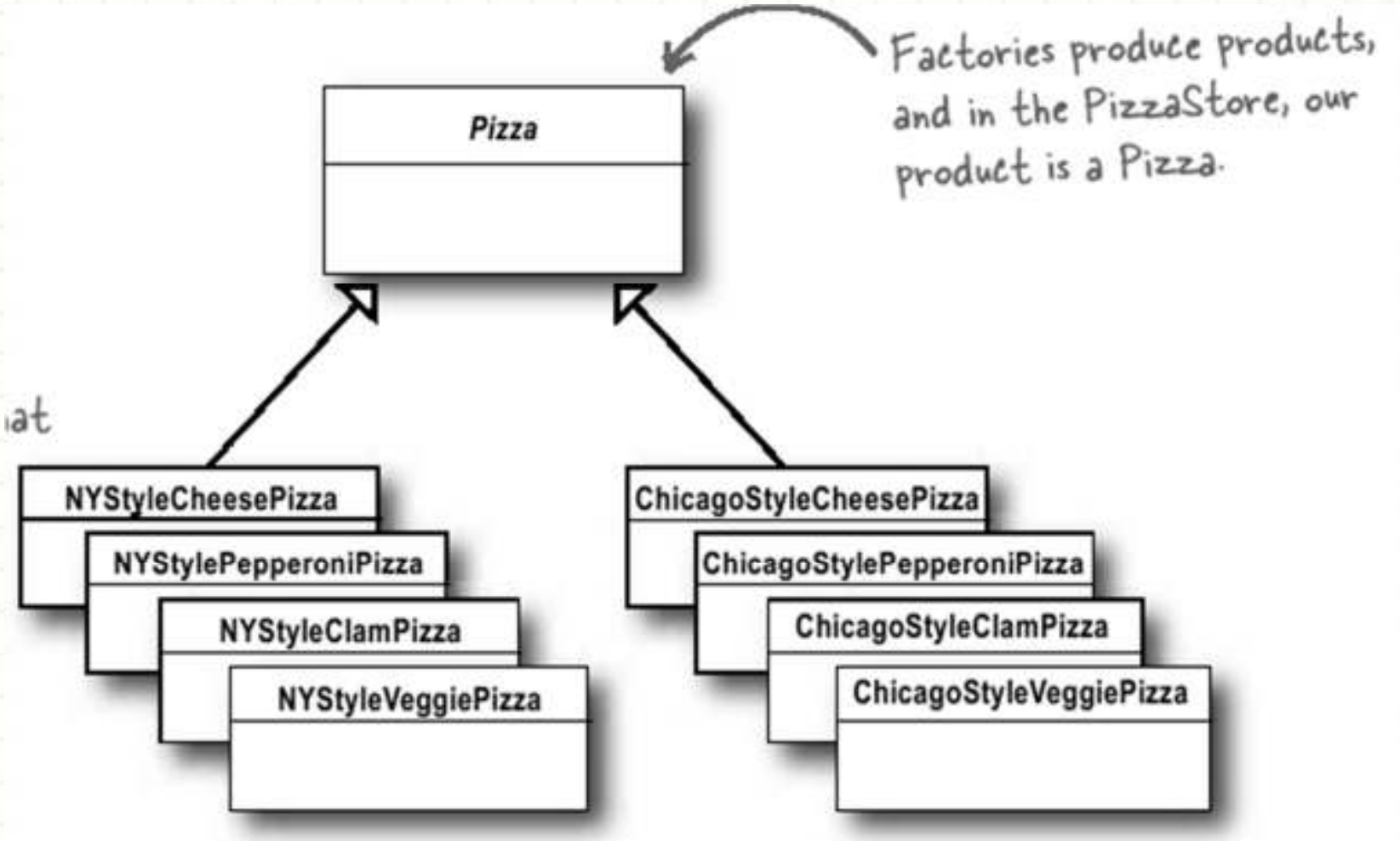
The NY Style Cheese Pizza

```
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 Chicago Style Pizza

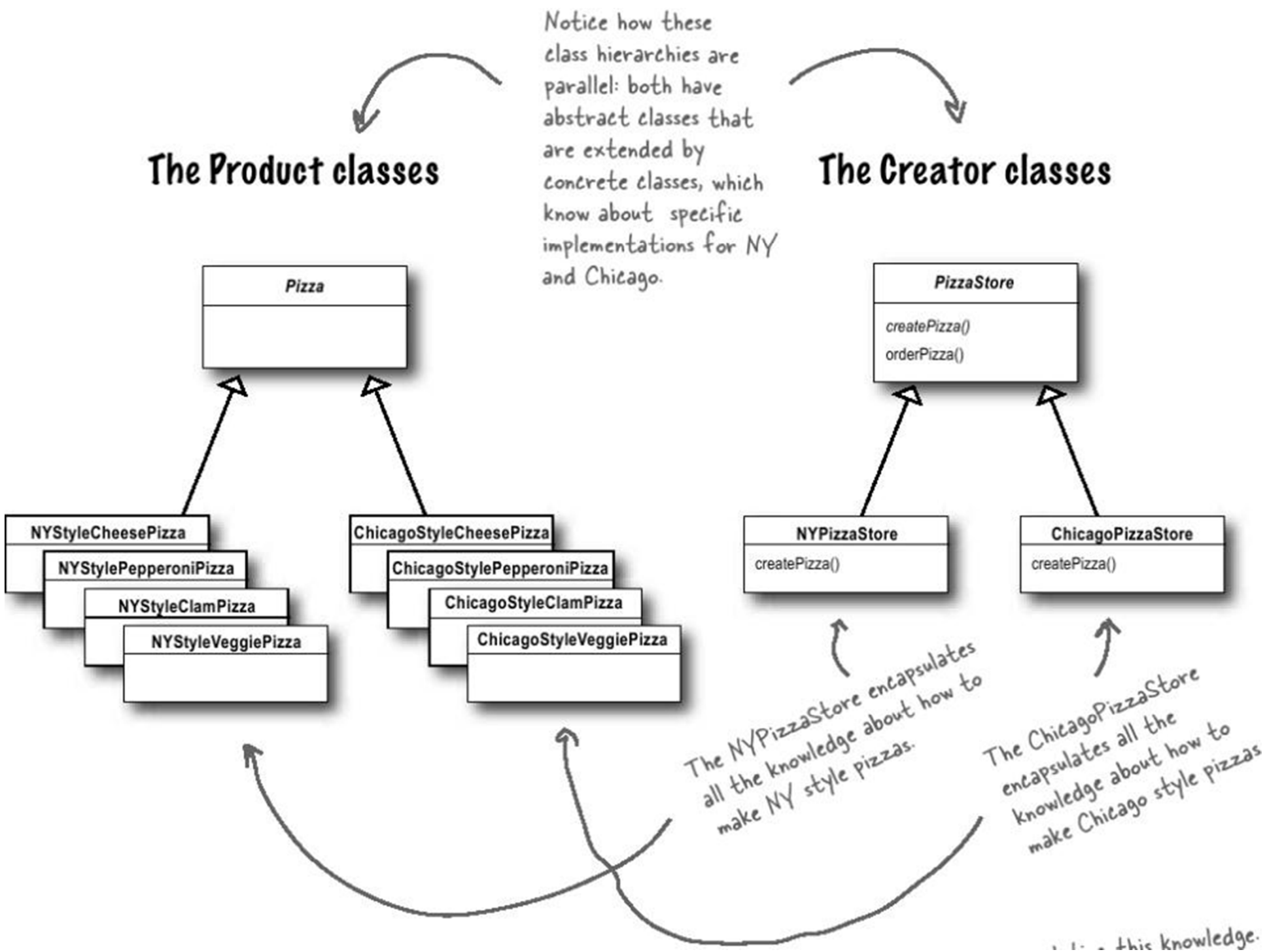
```
Public class ChicagoStyleCheesePizza extends Pizza {
  public ChicagoStyleChesePizza() {
    name = “Chicago Style Deep Dish Cheese Pizza”;
    dough = “Extra Thick Crust Dough”;
    sauce = “Plum Tomato Sauce”;
    toppings.add(“Shredded Mozzarella Cheese”);
  }

  void cut() {
    // Cutting the pizza into square slices
  }
}
```

The Product classes



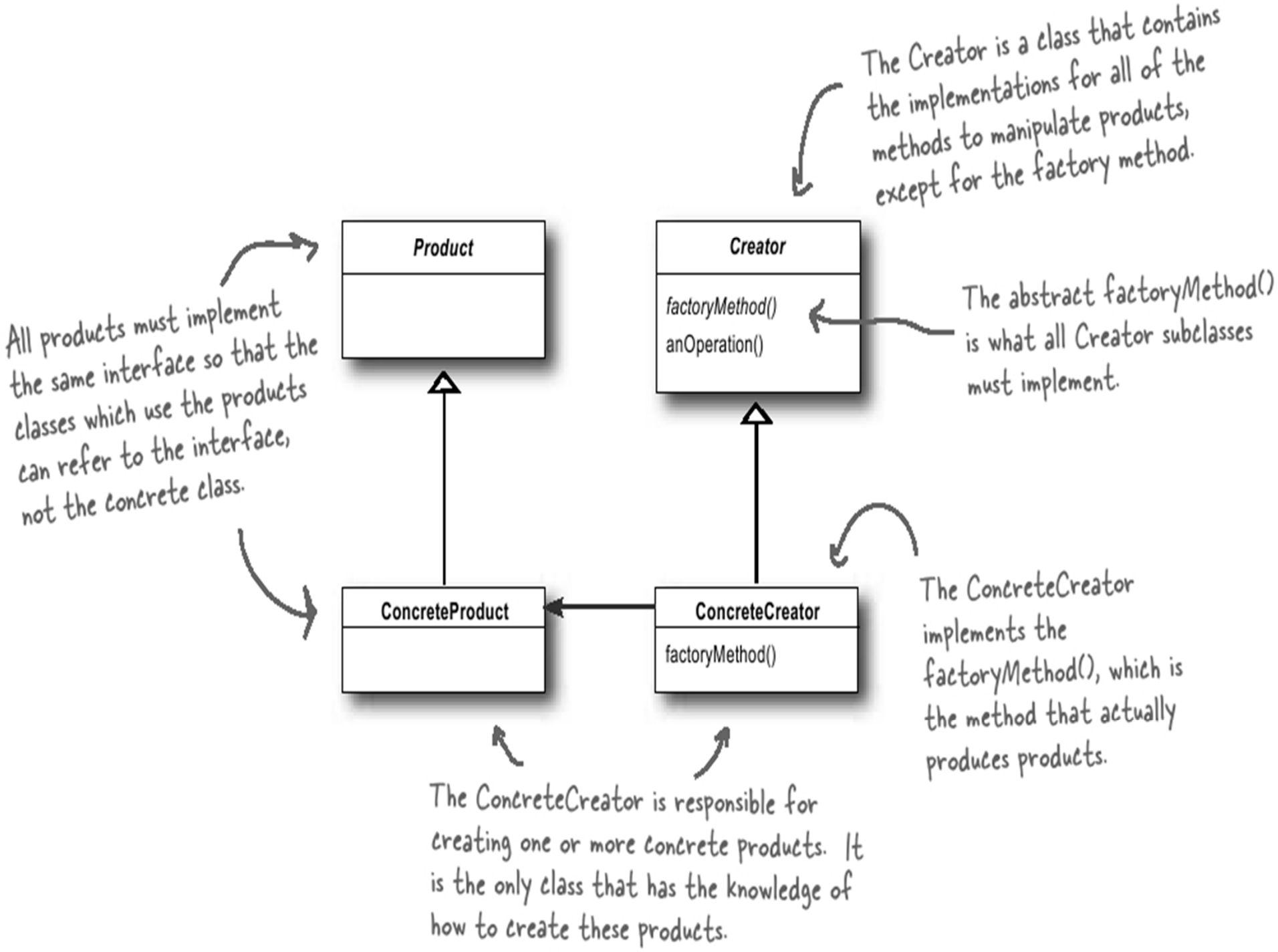
The Client

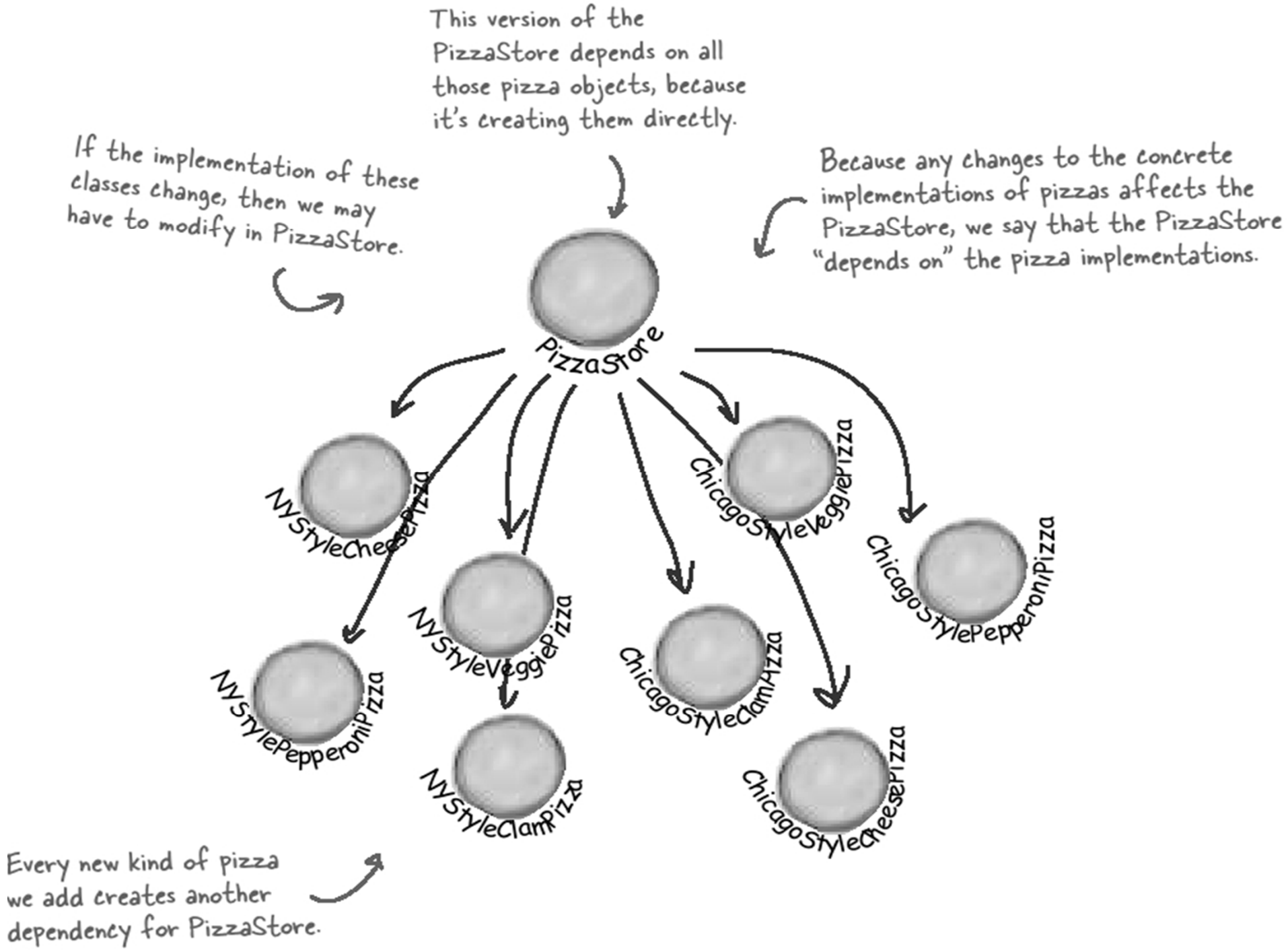
```
Public class PizzaClient {
   public static void main(String[] args) {
      PizzaStore nyStore = new NYPizzaStore();
      PizzaStore chicagoStore = new ChicagoPizzaStore();

      // order NY Style cheese
      Pizza pizza = nyStore.orderPizza("cheese");

      // order Chicago Style cheese
      Pizza pizza2 = chicagoStore.orderPizza("cheese");
   }
}
```

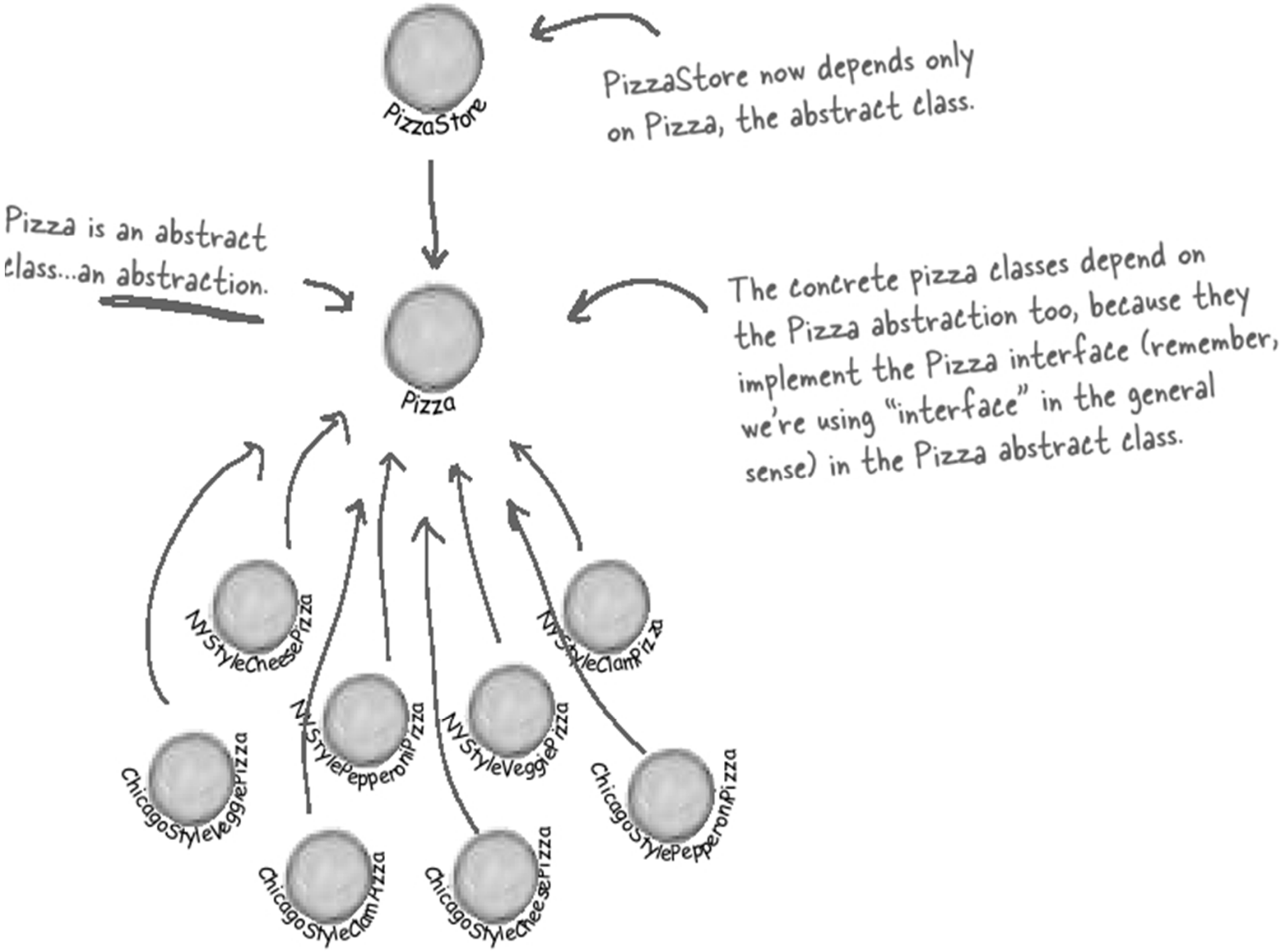

Notice how these class hierarchies are parallel: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago.
The Product classes
The Creator classes
Pizza
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza
ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza
PizzaStore
createPizza()
orderPizza()
NYPizzaStore
createPizza()
ChicagoPizzaStore
createPizza()
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

Definition

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


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.


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.
PizzaStore
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleVeggiePizza
NYStyleClamPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza
ChicagoStylePepperoniPizza
ChicagoStyleCheesePizza
Every new kind of pizza we add creates another dependency for PizzaStore.


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

QUESTIONS?

REFERENCES

Head First Design Patterns, Eric F, Elisabeth F, 2004