THE FACTORY PATTERNS

Chandan R. Rupakheti

Week 3-1

Today …

- The Factory Method Pattern
- The Dependency Inversion Principle
- The Abstract Factory Pattern

Object Creation

- There is more to making objects than just using the **new** operator

- Instantiation done in public and can often lead to *coupling problems*

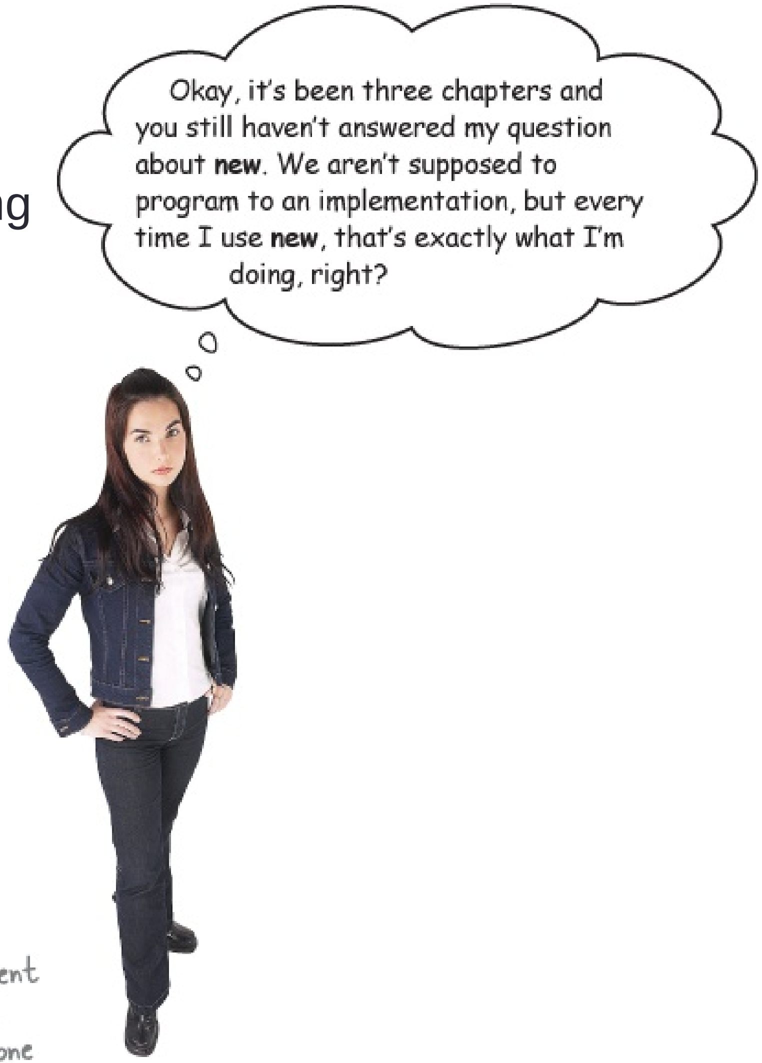
```
Duck duck = new MallardDuck();
```

We want to use interfaces to keep code flexible.

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

```
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.


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?

What’s wrong with “new”?

- Nothing is wrong with new, but it’s the change that impacts use of **new**.

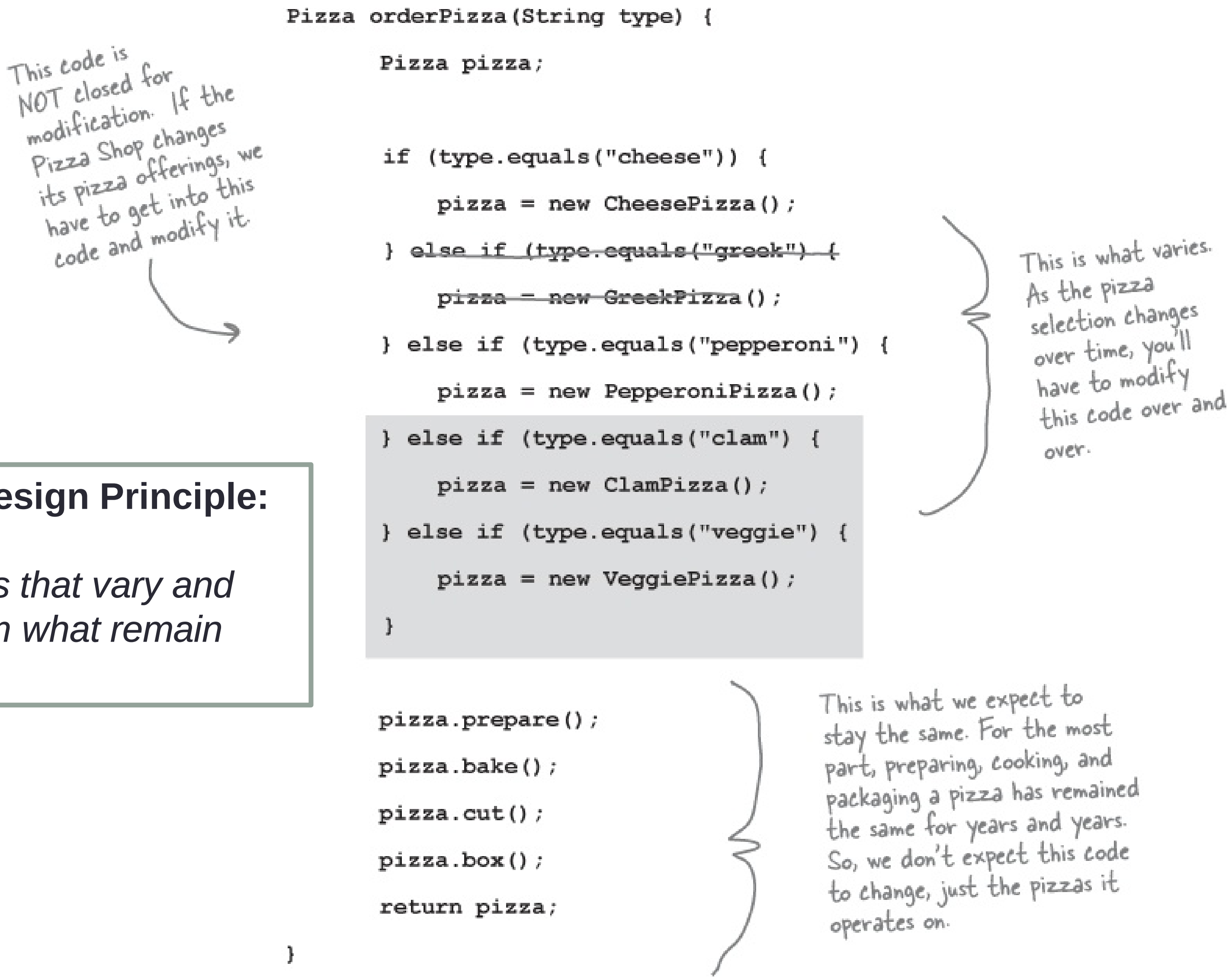
- Recall the DIP principle: “*Program to an interface not to an implementation.”*

But you have to create an object at some point and Java only gives us one way to create an object, right? So what gives?

```
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.

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 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.

Recall the First Design Principle:

Identify the aspects that vary and separate them from what remain unchanged.

Encapsulating object creation

```
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();
}
```

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



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

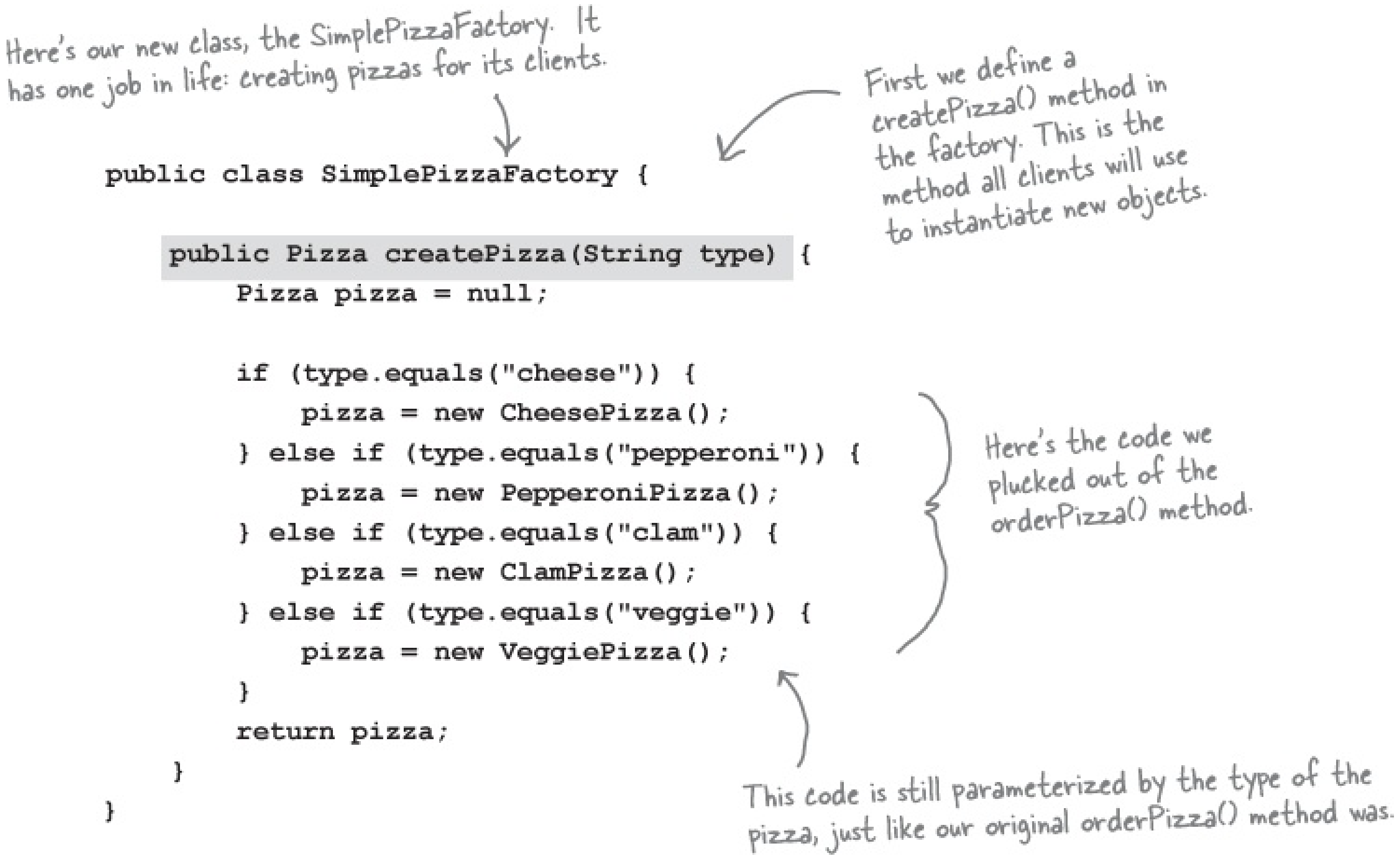
First we pull the object creation code out of the orderPizza() Method.

What's going to go here?

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.

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

Building a simple pizza factory



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

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

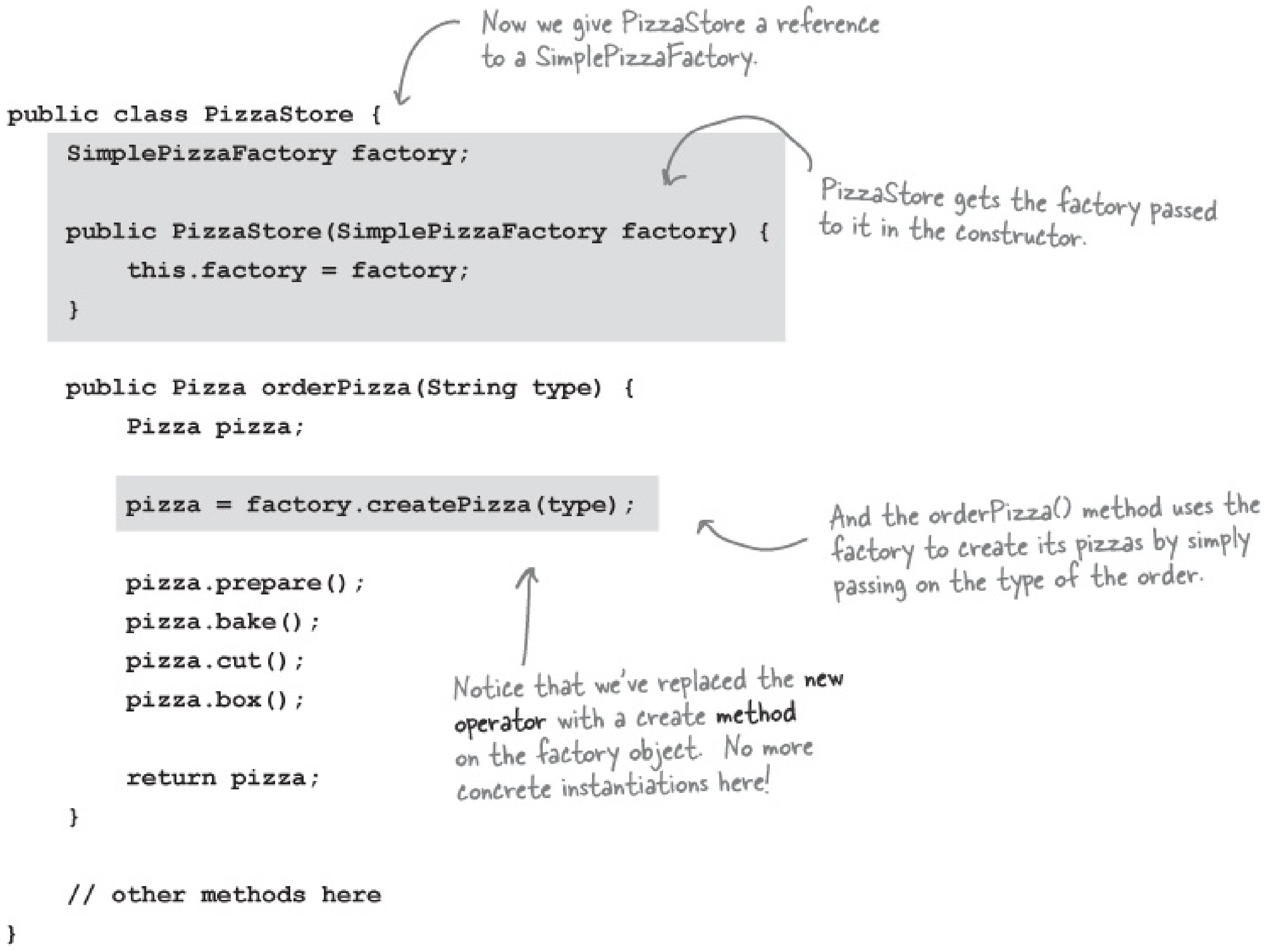
```
public class SimplePizzaFactory {

    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;
    }
}
```

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.

Q1

Reworking the PizzaStore class



```
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
}
```

Now we give PizzaStore a reference to a SimplePizzaFactory.

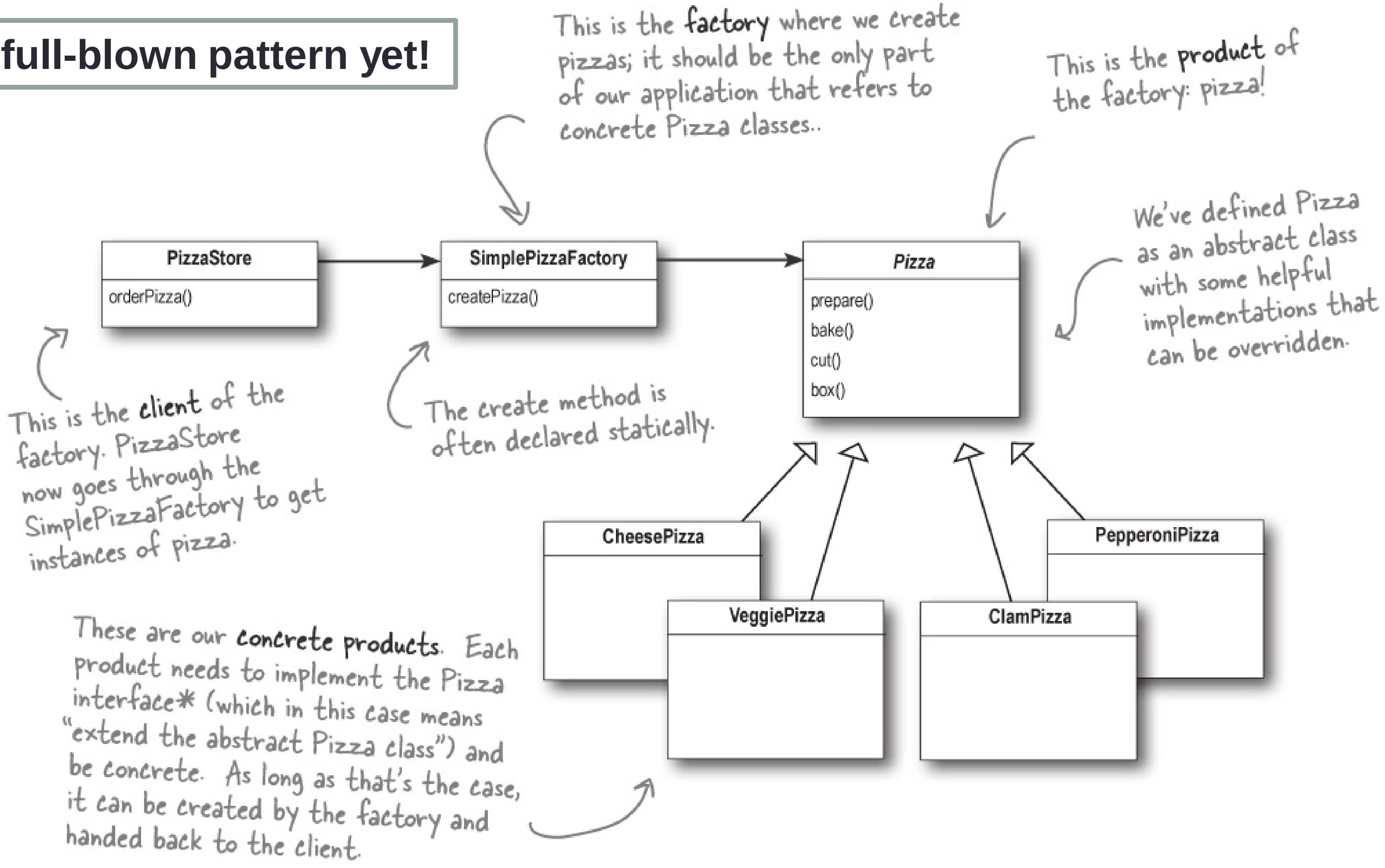
PizzaStore gets the factory passed to it in the constructor.

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!

Pattern Honorable Mention

Not a full-blown pattern yet!



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!

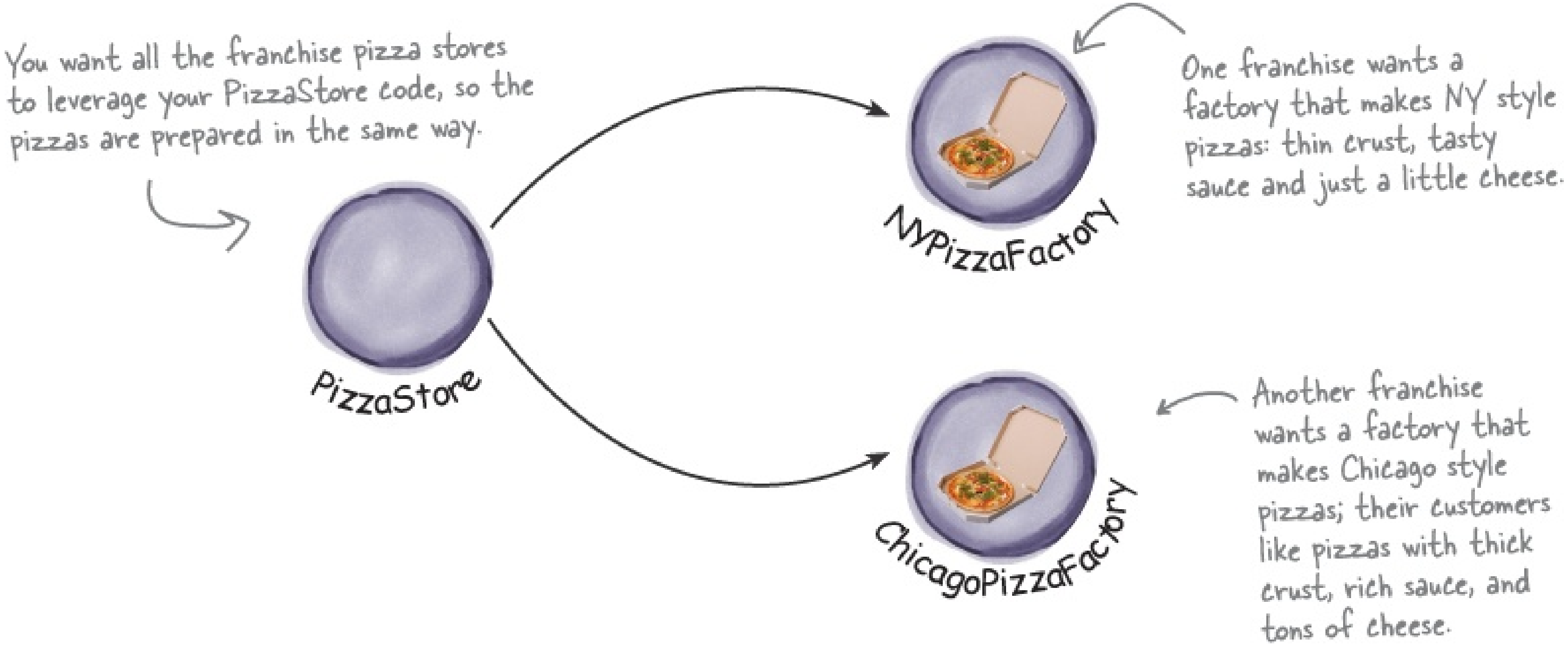
We've defined Pizza as an abstract class with some helpful implementations that can be overridden.

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

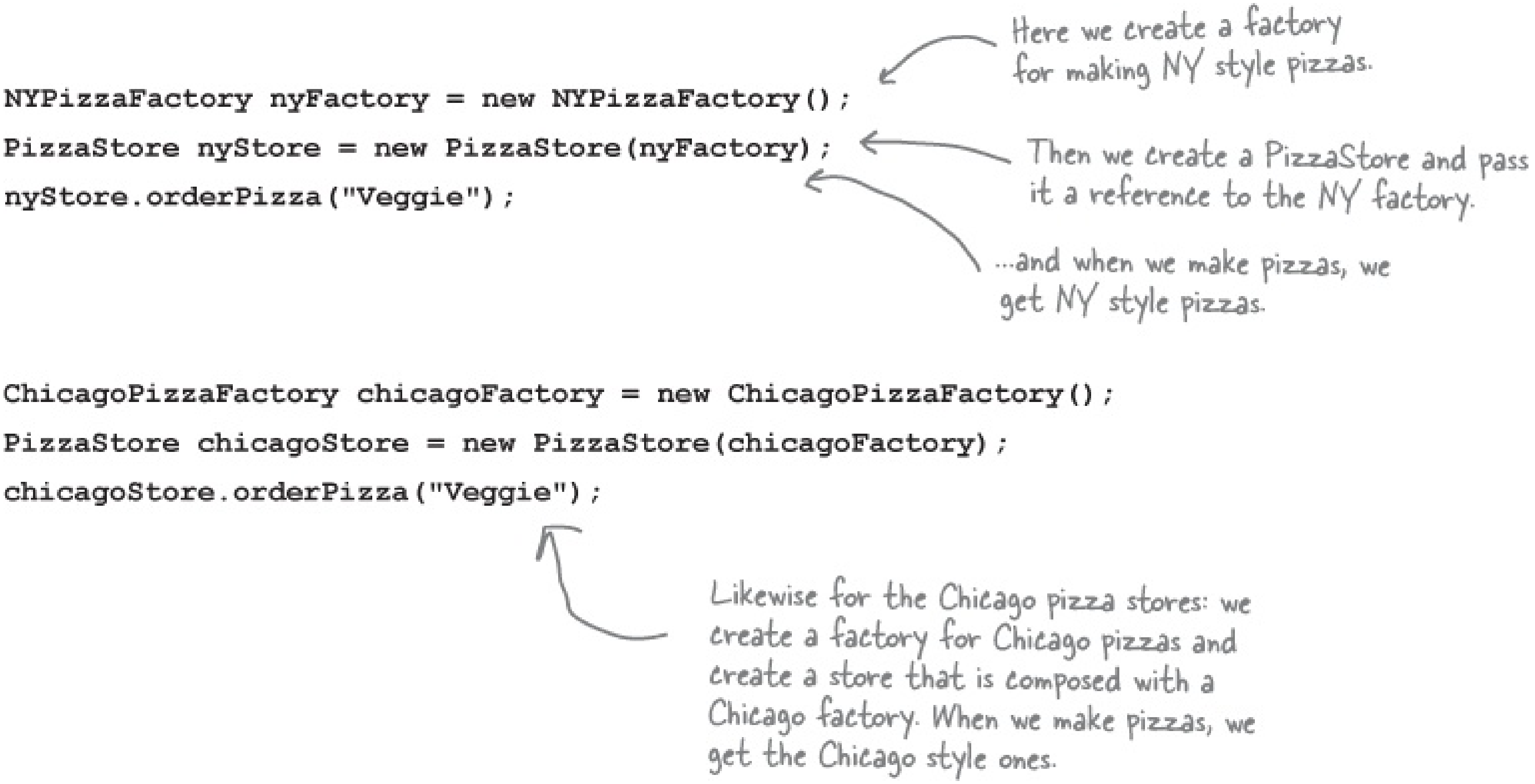
The create method is often declared statically.

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


You want all the franchise pizza stores to leverage your PizzaStore code, so the pizzas are prepared in the same way.
PizzaStore
NYPizzaFactory
One franchise wants a factory that makes NY style pizzas: thin crust, tasty sauce and just a little cheese.
ChicagoPizzaFactory
Another franchise wants a factory that makes Chicago style pizzas; their customers like pizzas with thick crust, rich sauce, and tons of cheese.

Let’s solve this with SimpleFactory



```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("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 style pizzas.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.orderPizza("Veggie");
```

Likewise for the Chicago pizza stores: we create a factory for Chicago pizzas and create a store that is composed with a Chicago factory. When we make pizzas, we get the Chicago style ones.

Now we like a little more quality control…

After the first test-launch, you found 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.

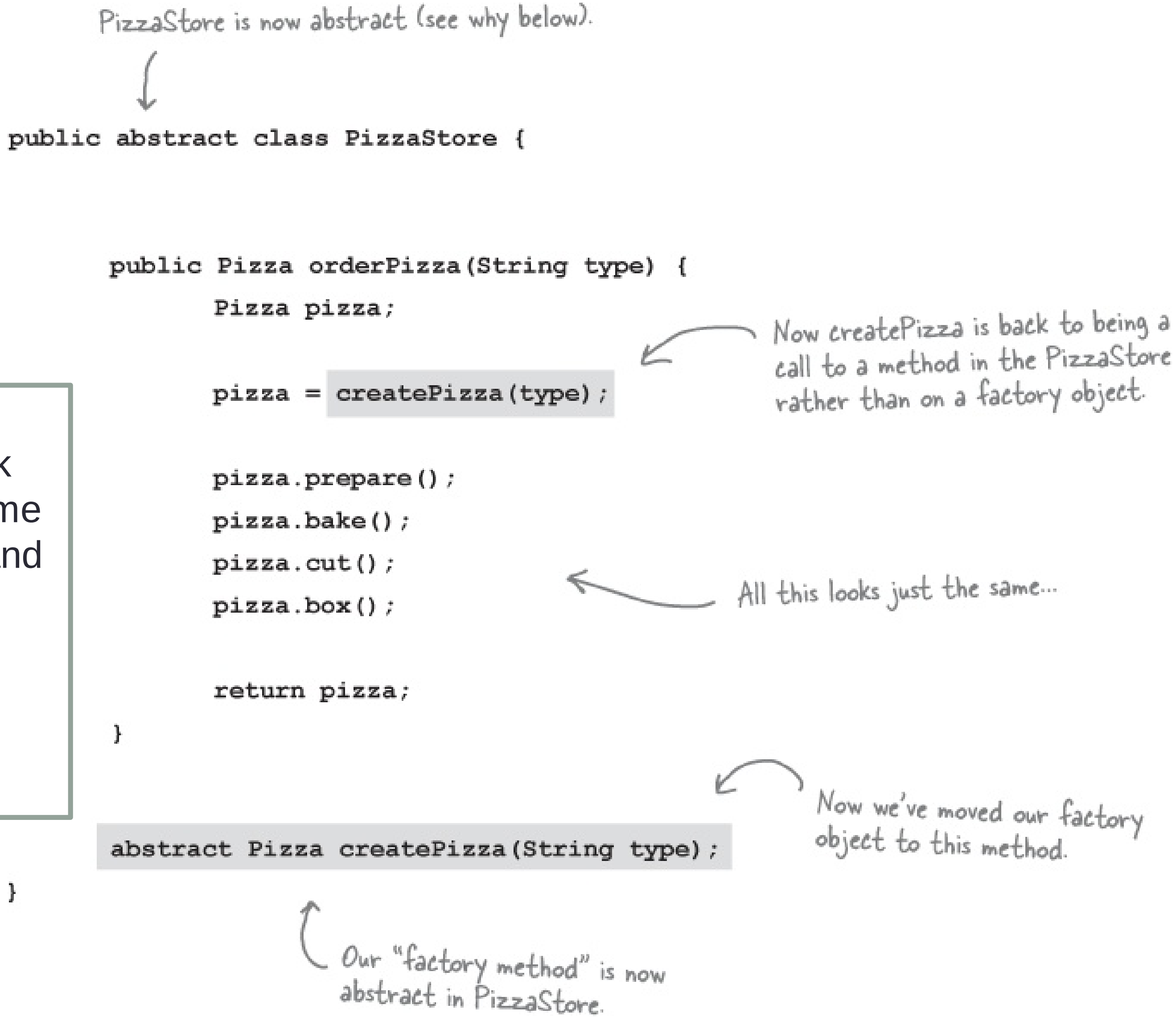
Rethinking the problem a bit, you see that what you'd really like to do is create **a framework that ties the store and the pizza creation together**, yet still allows things to remain flexible.

I've been making pizza for years so I thought I'd add my own "improvements" to the PizzaStore procedures...

Not what you want in a good franchise. You do NOT want to know what he puts on his pizzas.

A framework for the pizza store

We're going to put the createPizza() method back into PizzaStore, but this time as an **abstract method**, and then create a PizzaStore subclass for each regional style. E.g., NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore.



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);

}
```

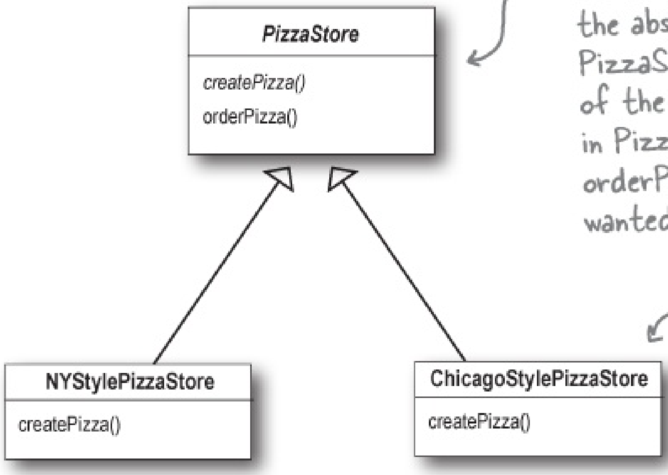
Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

All this looks just the same...

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

Our "factory method" is now abstract in PizzaStore.

Allowing the subclasses to decide



Each subclass provides an implementation of the createPizza() method, overriding the abstract createPizza() method in PizzaStore, 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.

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.

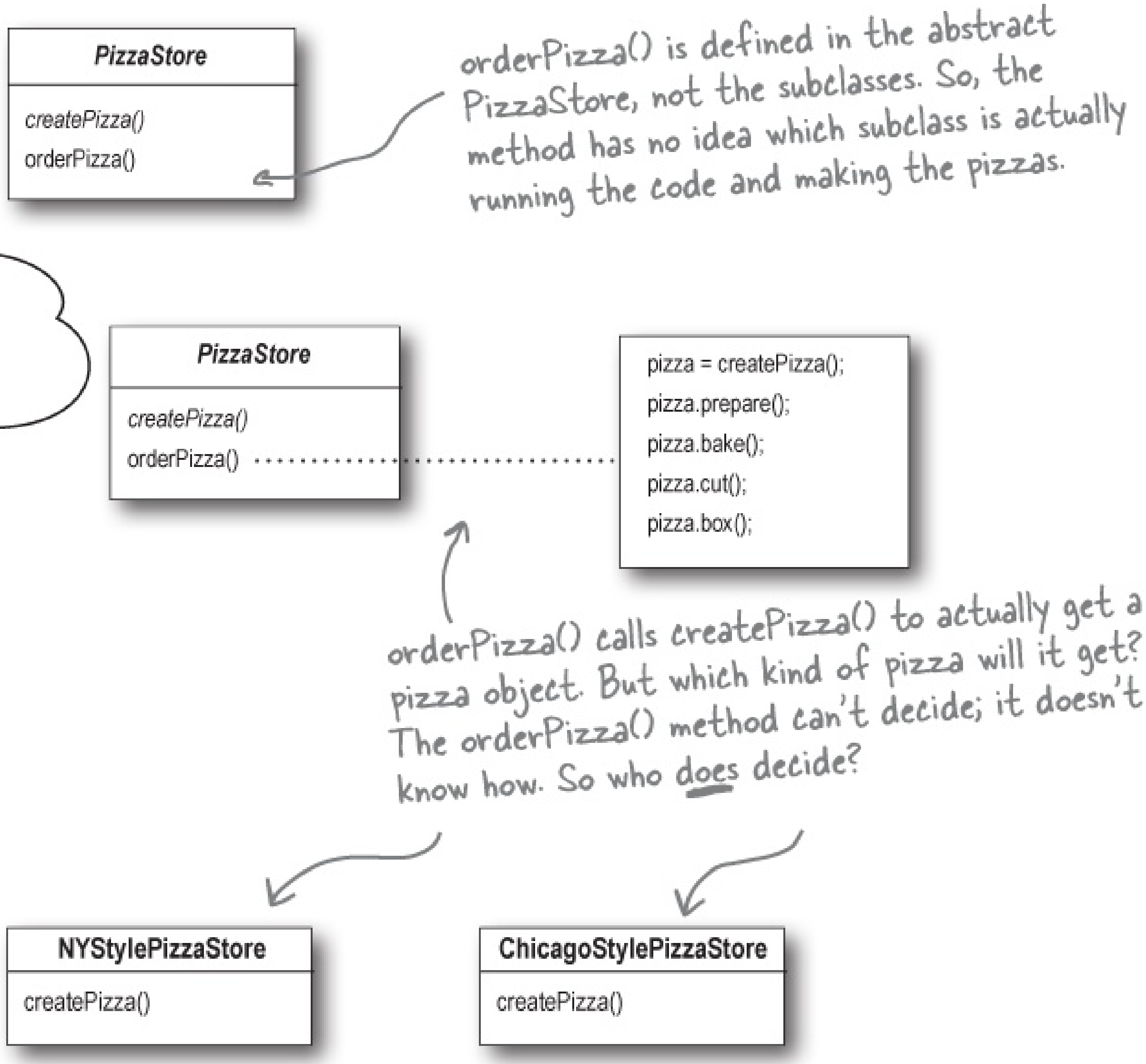
Similarly, by using the Chicago subclass, we get an implementation of createPizza() with Chicago ingredients.

```
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") {
        pizza = new NYStyleVeggiePizza();
    }
}
```

```
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") {
        pizza = new ChicagoStyleVeggiePizza();
    }
}
```

Confusion!

I don't get it. The PizzaStore subclasses are just subclasses. How are they deciding anything? I don't see any logical decision-making code in NYStylePizzaStore....



PizzaStore

createPizza()

orderPizza()

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.

PizzaStore

createPizza()

orderPizza()

```
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?

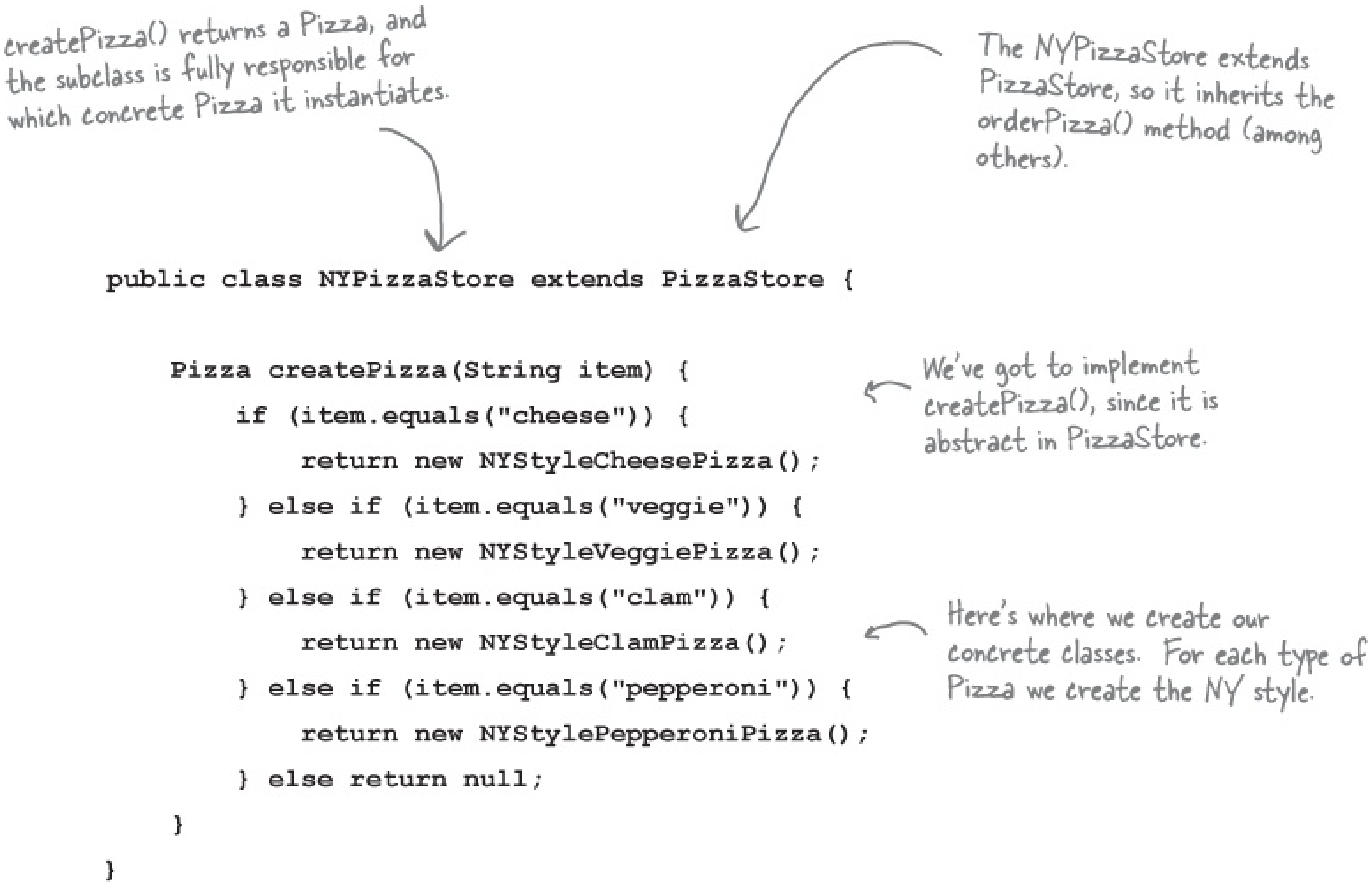
NYStylePizzaStore

createPizza()

ChicagoStylePizzaStore

createPizza()

Let’s make a PizzaStore



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

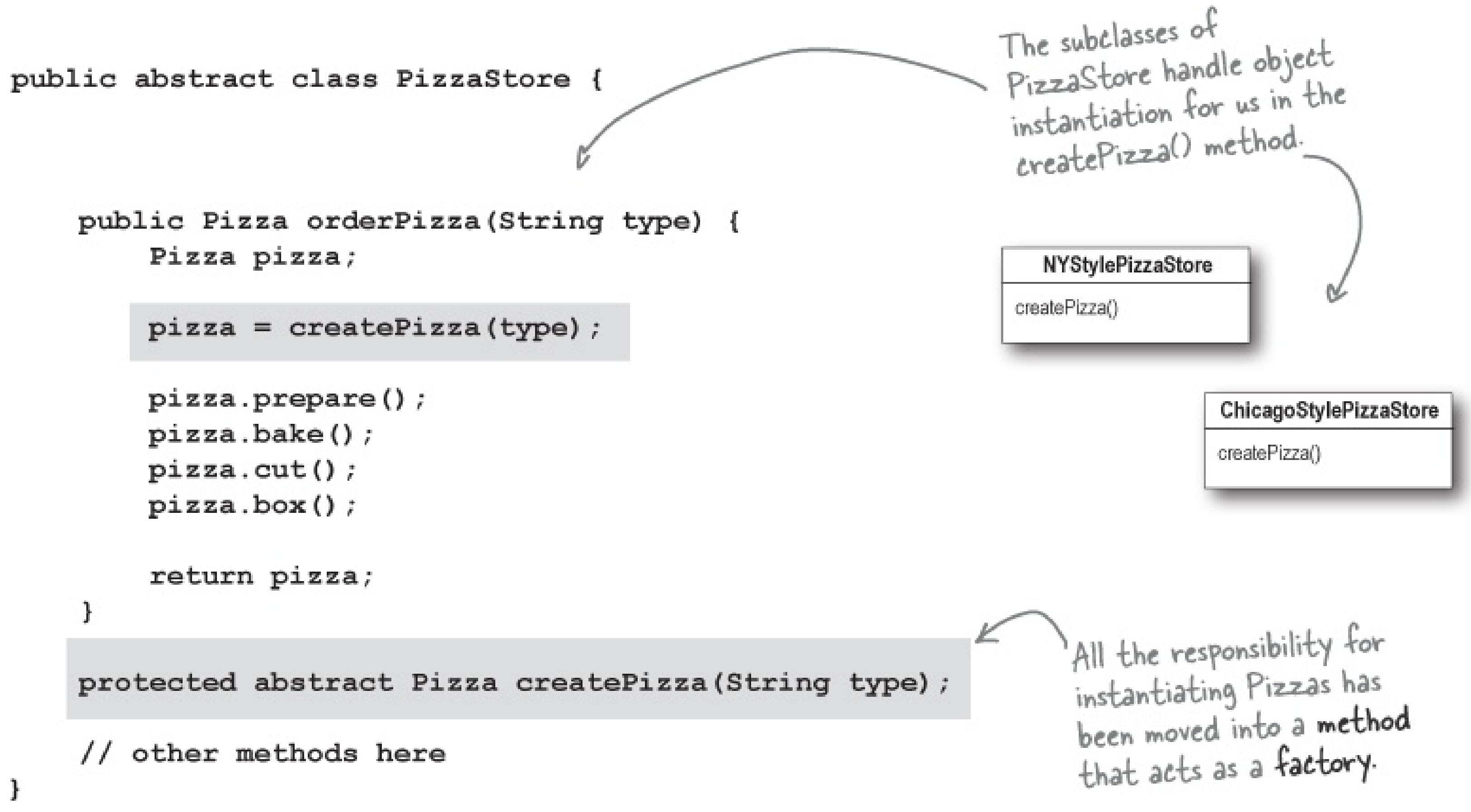
The NYPizzaStore extends PizzaStore, so it inherits the orderPizza() method (among others).

```
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;
    }
}
```

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.

Declaring a factory method



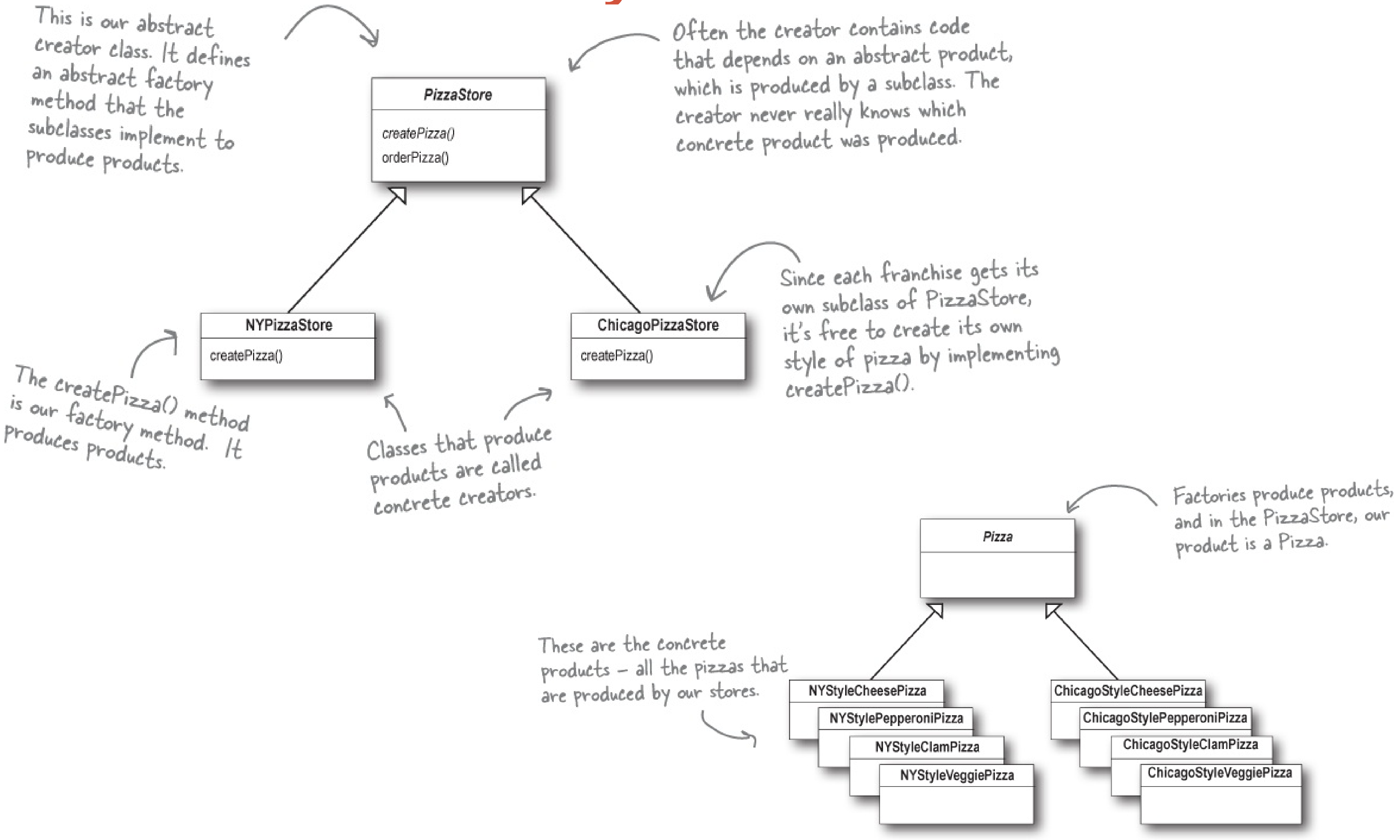
```
public abstract class PizzaStore {


    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

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

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

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

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

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.

Meet the Factory Method Pattern



This is our abstract creator class. It defines an abstract factory method that the subclasses implement to produce products.

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.

Since each franchise gets its own subclass of PizzaStore, it's free to create its own style of pizza by implementing createPizza().

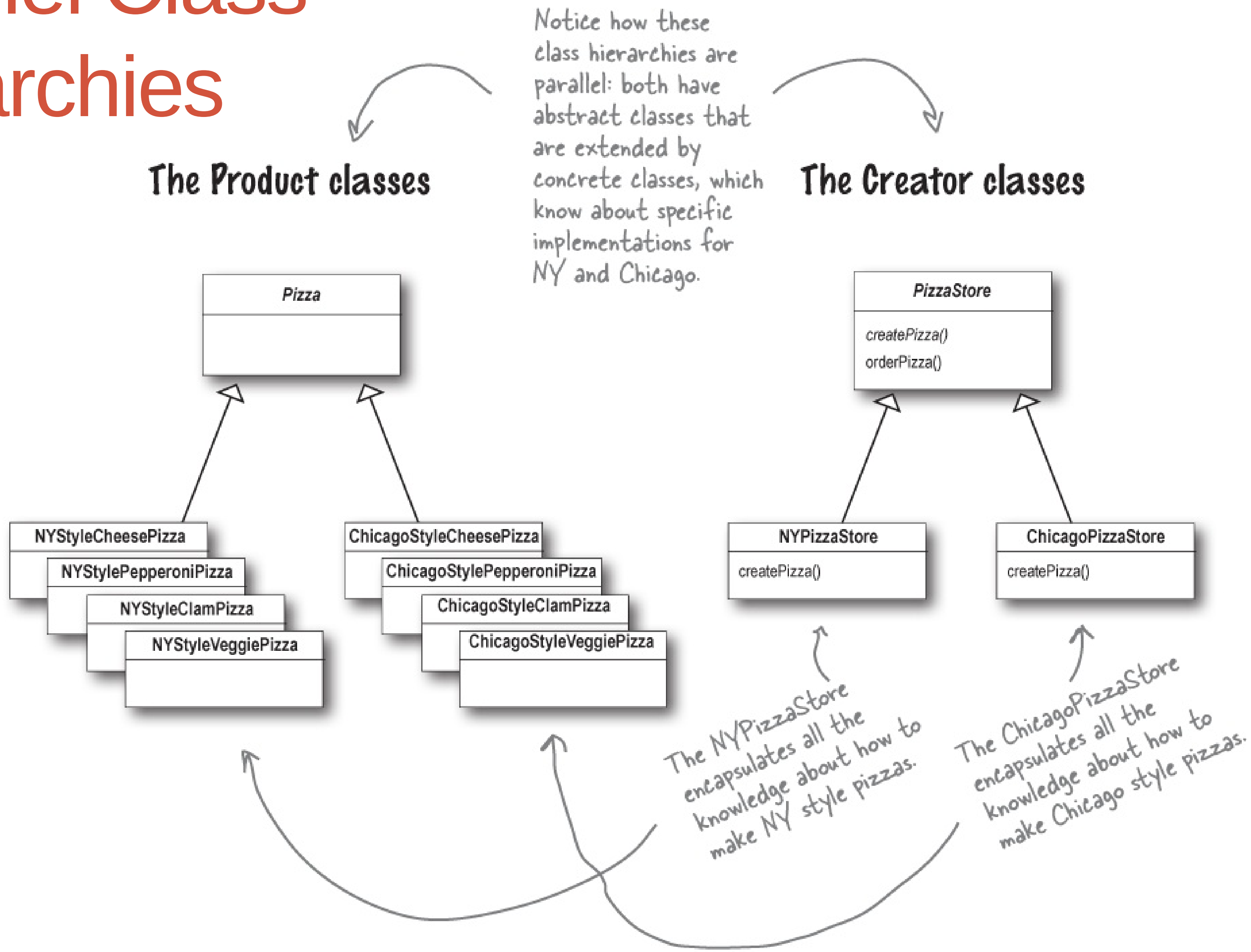
The createPizza() method is our factory method. It produces products.

Classes that produce products are called concrete creators.

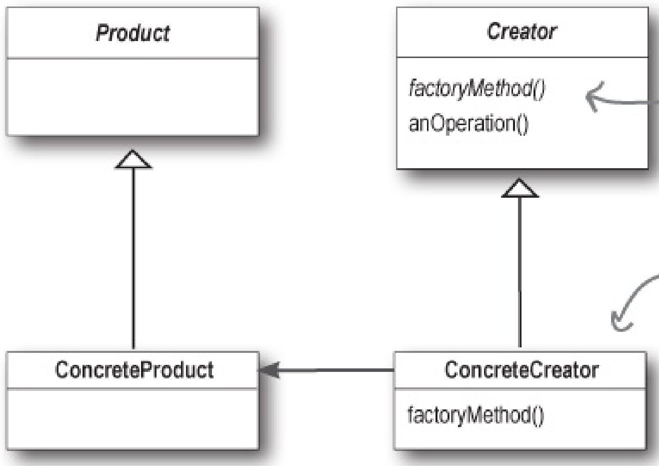
Factories produce products, and in the PizzaStore, our product is a Pizza.

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

Parallel Class Hierarchies


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.

The Factory Method Pattern Defined



The Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.

All products must implement the same interface so that the classes that use the products can refer to the interface, not the concrete class.

The abstract factoryMethod() is what all Creator subclasses must implement.

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.

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.

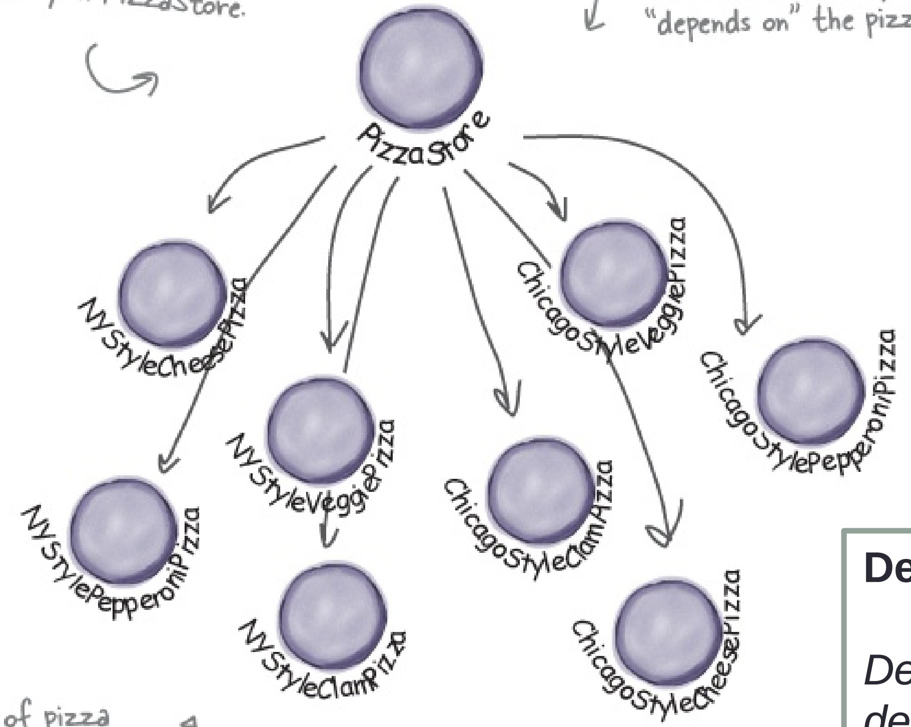
Q2

Dependency Inversion Principle

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

If the implementation of these classes changes, 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.

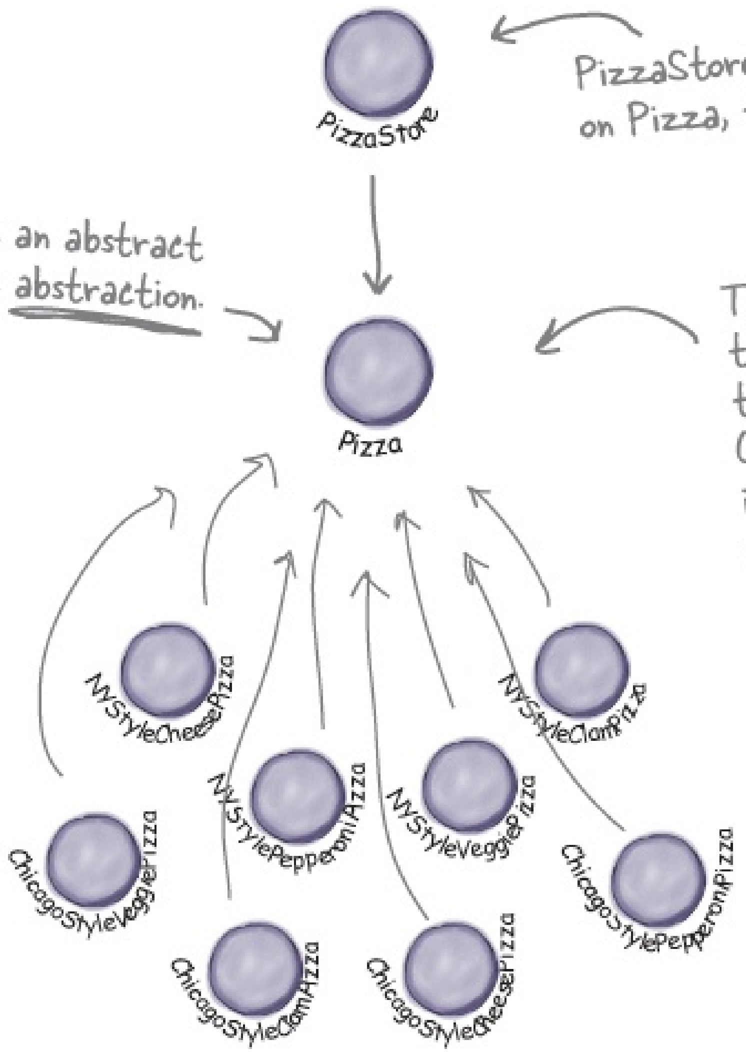


Every new kind of pizza we add creates another dependency for PizzaStore.

Design Principle (DIP):

Depend upon abstractions. Do not depend upon concrete classes.

Applying DIP



PizzaStore now depends only on Pizza, the abstract class.

Pizza is an abstract class...an abstraction.

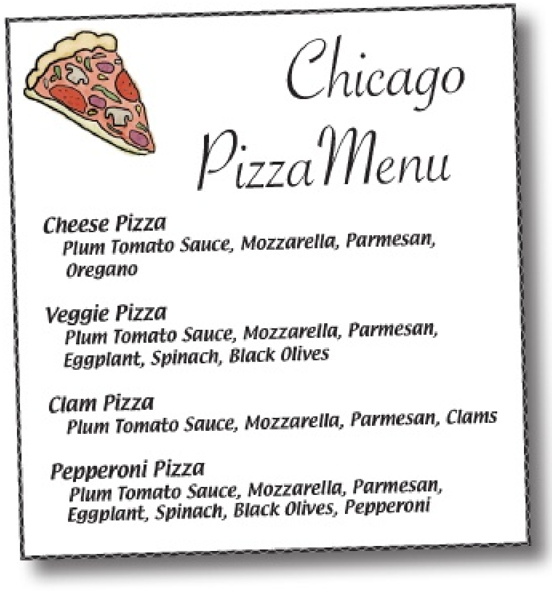
The concrete pizza classes depend on the Pizza abstraction too, because they implement the Pizza interface (remember, we're using "interface" in the general sense) in the Pizza abstract class.

Design Principle (DIP):

High-level components should not depend on our low-level components; rather, they should *both* depend on abstractions.

Ensuring consistency in our ingredients

You’ve discovered 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.



Chicago Pizza Menu

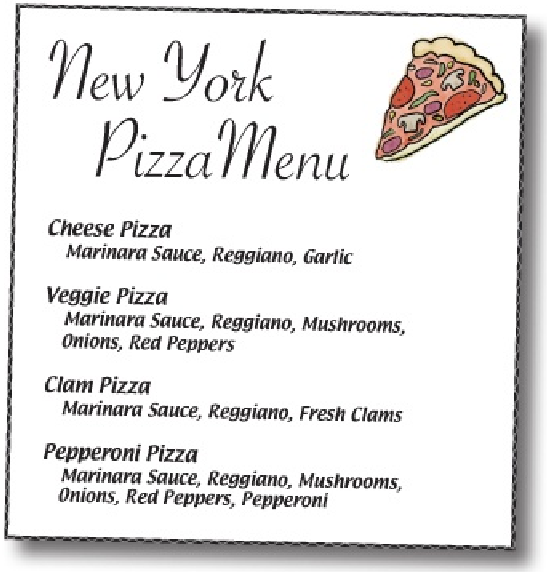
Cheese Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Oregano

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

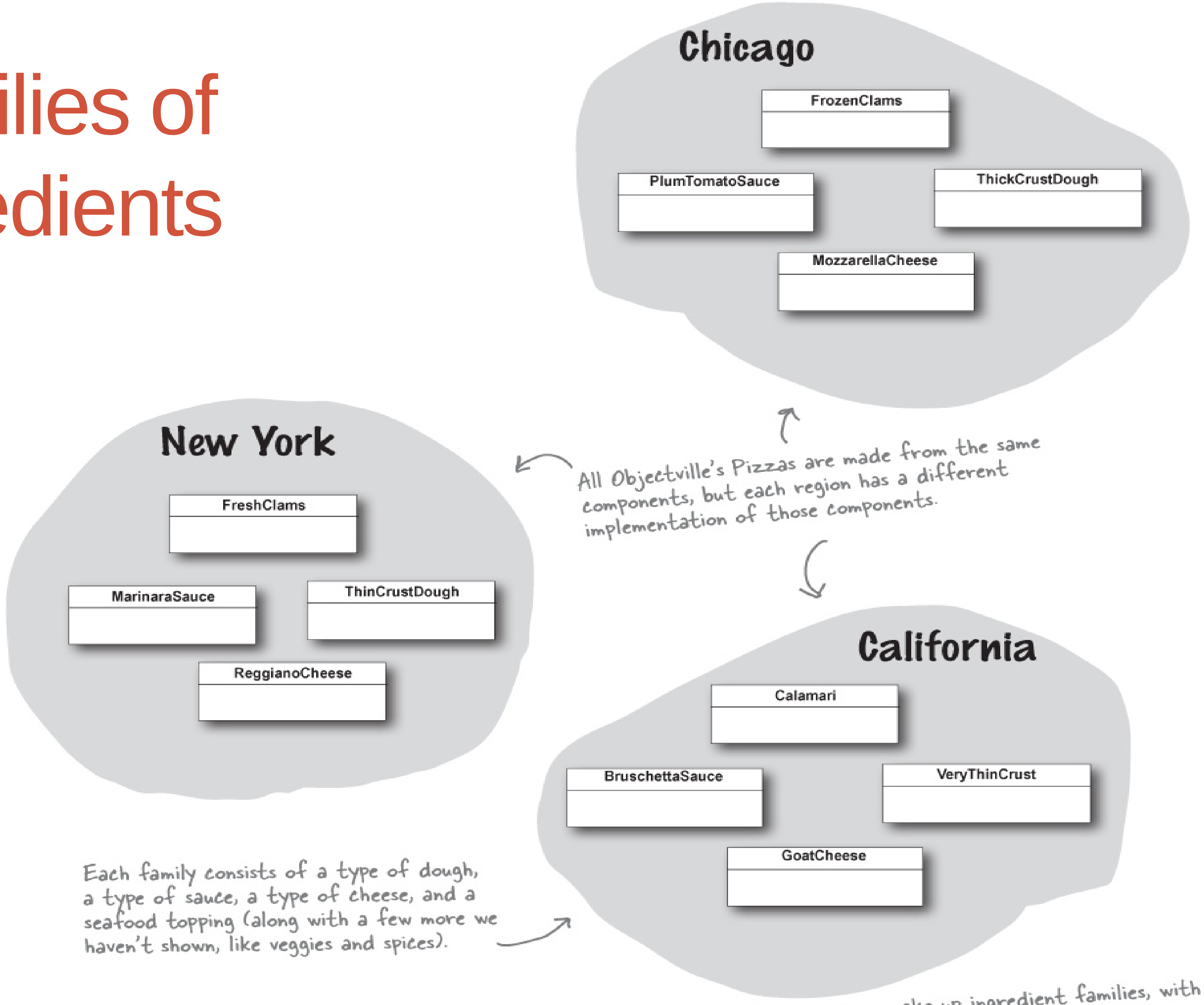
Cheese Pizza
Marinara Sauce, Reggiano, Garlic

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

Families of ingredients

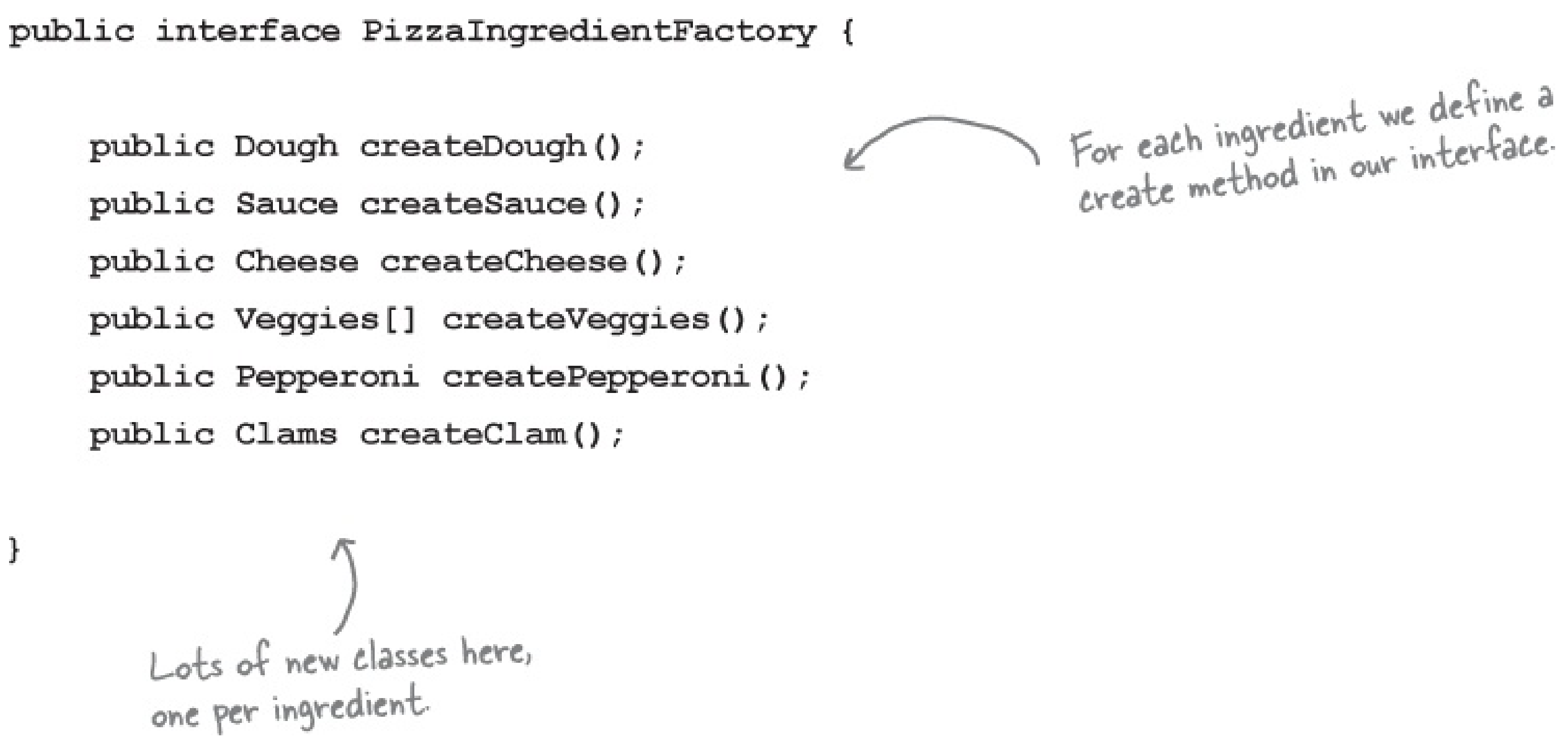


All Objectville's Pizzas are made from the same components, but each region has a different implementation of those components.

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

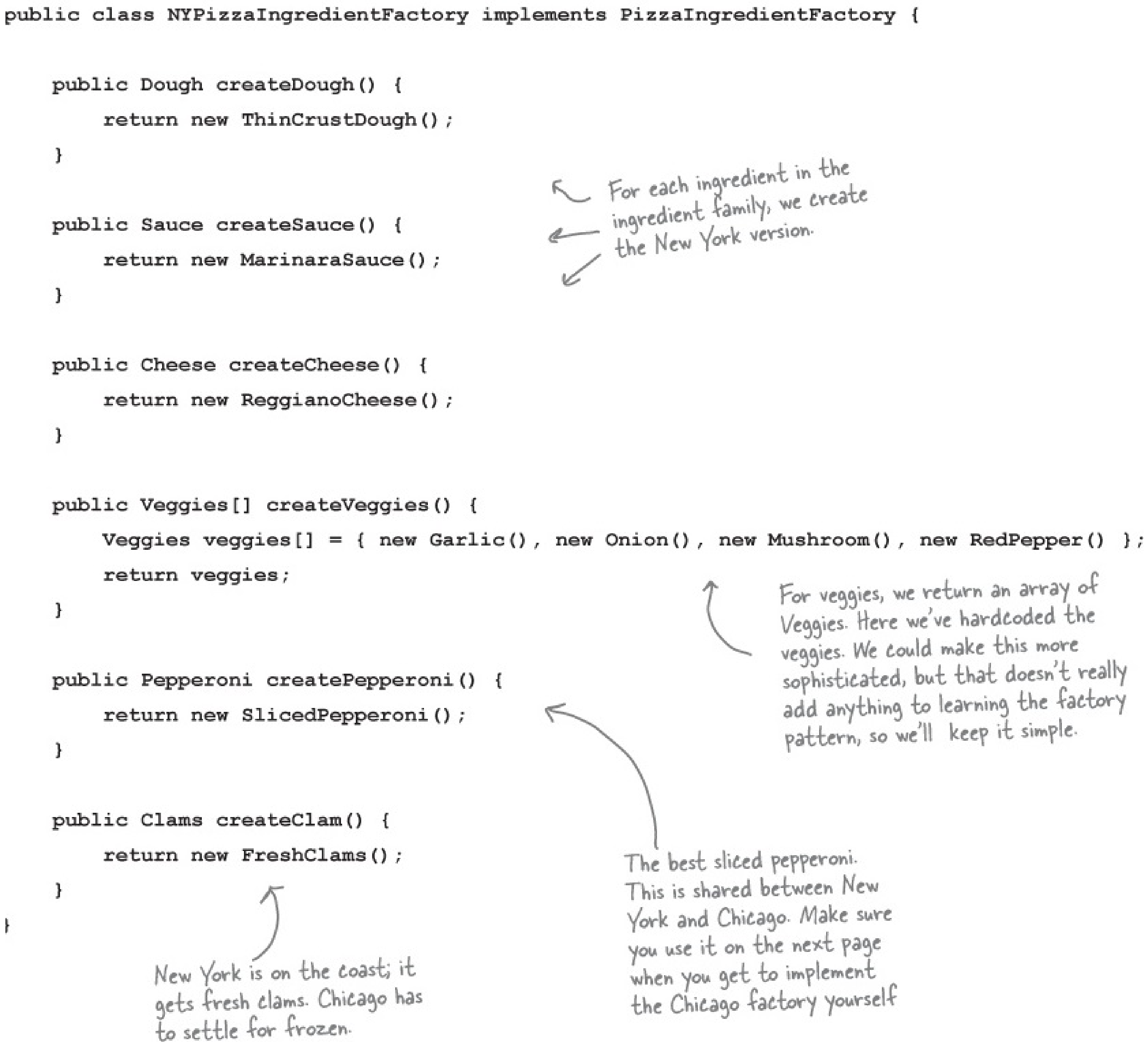


```
public interface PizzaIngredientFactory {

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

}
```

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

Lots of new classes here, one per ingredient.

NY Ingredient Factory



```
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();
   }
}
```

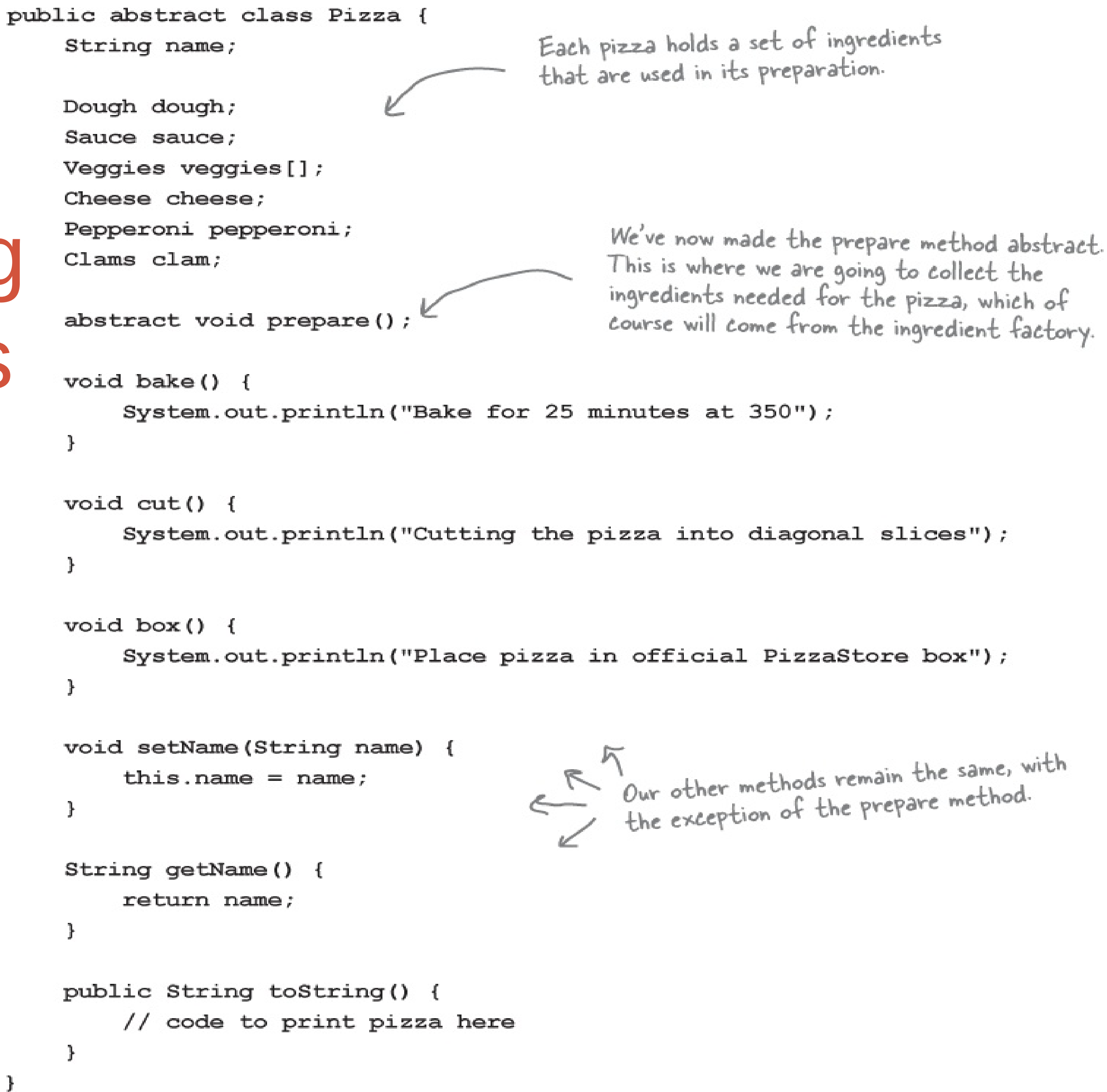
For each ingredient in the ingredient family, we create the New York version.

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.

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

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

Reworking the Pizzas



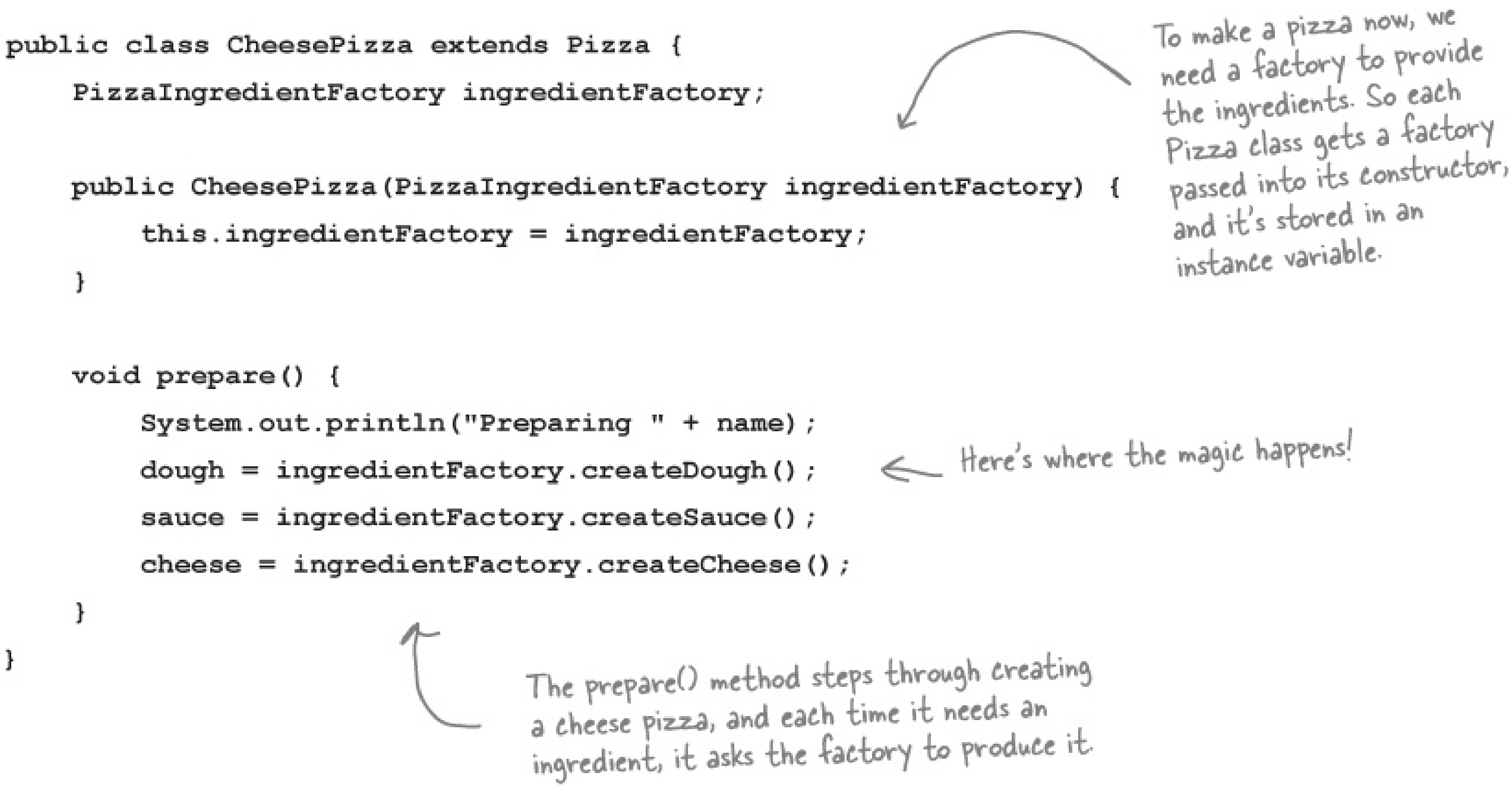
```
public abstract class Pizza {
    String name;

    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.

Let’s Rework CheesePizza



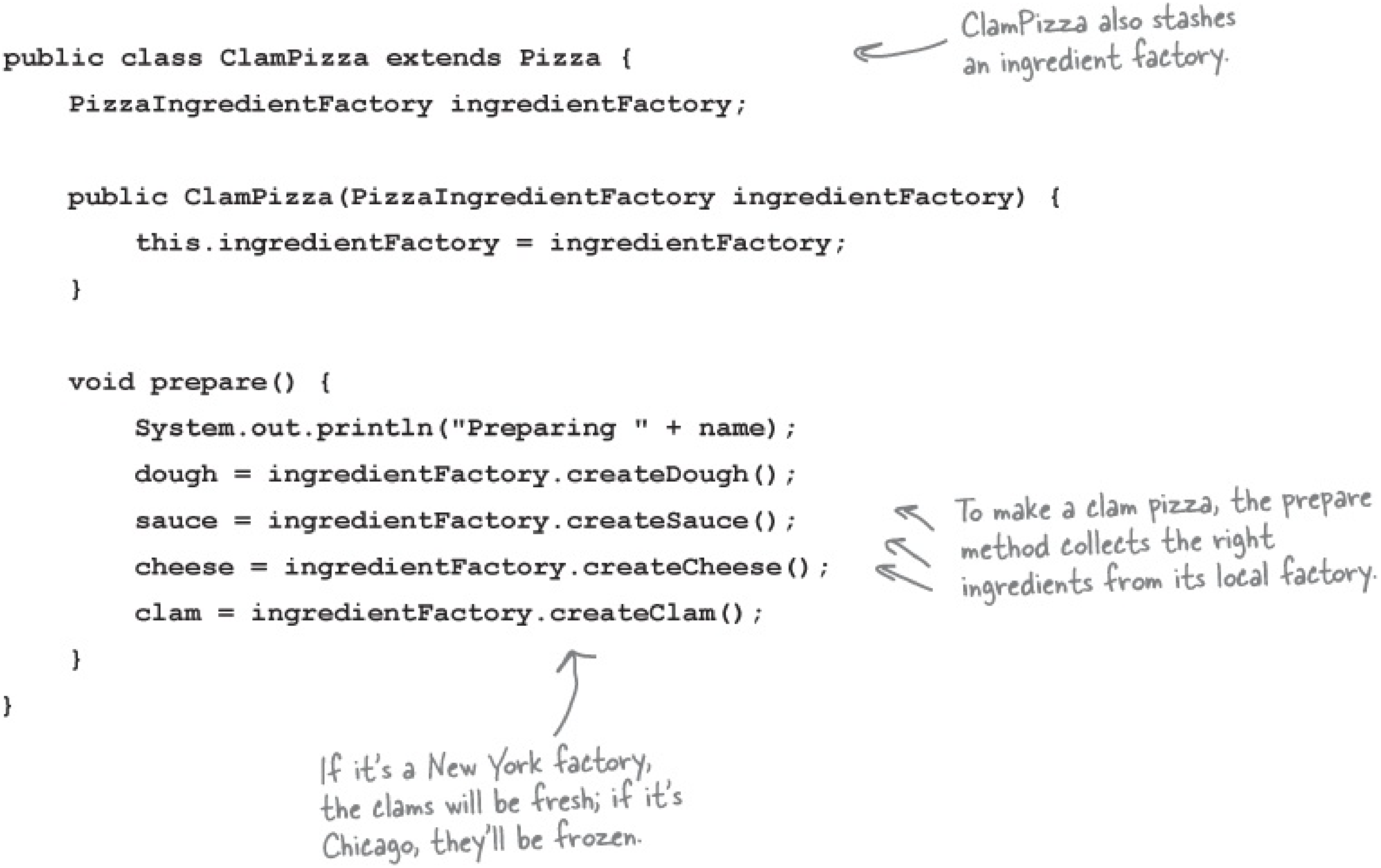
```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    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.

Another Example ClamPizza



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

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

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

ClamPizza also stashes an ingredient factory.

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

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

What have we done?

We provided a means of creating a family of ingredients for pizzas by introducing a new type of factory called an Abstract Factory.

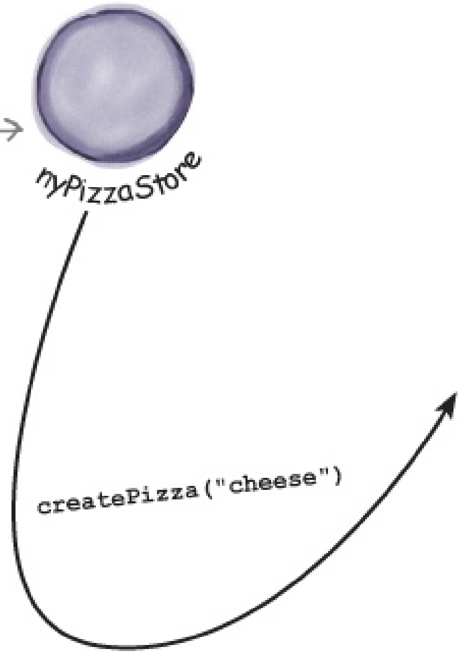
Because our code is decoupled from the actual products, we can substitute different factories to get different behaviors (like getting marinara instead of plum tomatoes)

The Order Process 1/2

1

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

Creates an instance of NYPizzaStore.



2

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

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

3

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

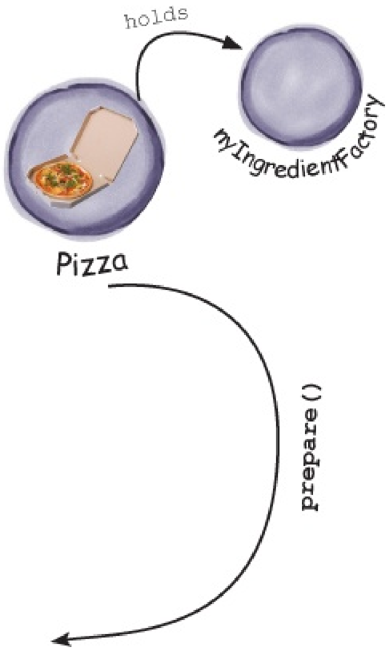
The Order Process 2/2

4

The ingredient factory is chosen and instantiated in the PizzaStore and then passed into the constructor of each pizza.

```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

Creates a instance of Pizza that is composed with the New York ingredient factory.



5

```
void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
```

Thin crust

Marinara

Reggiano

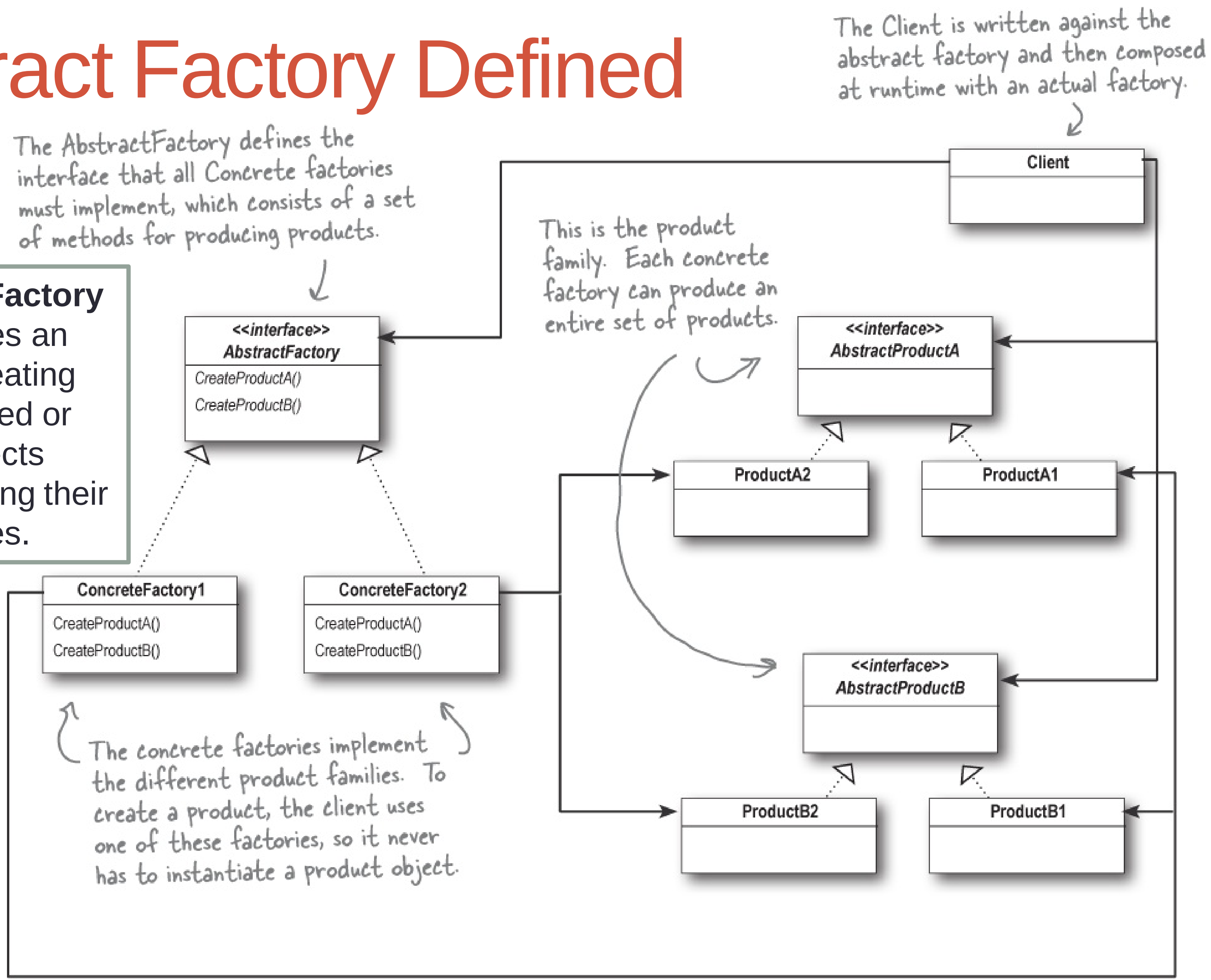
For Ethan's pizza the New York ingredient factory is used, and so we get the NY ingredients.

6

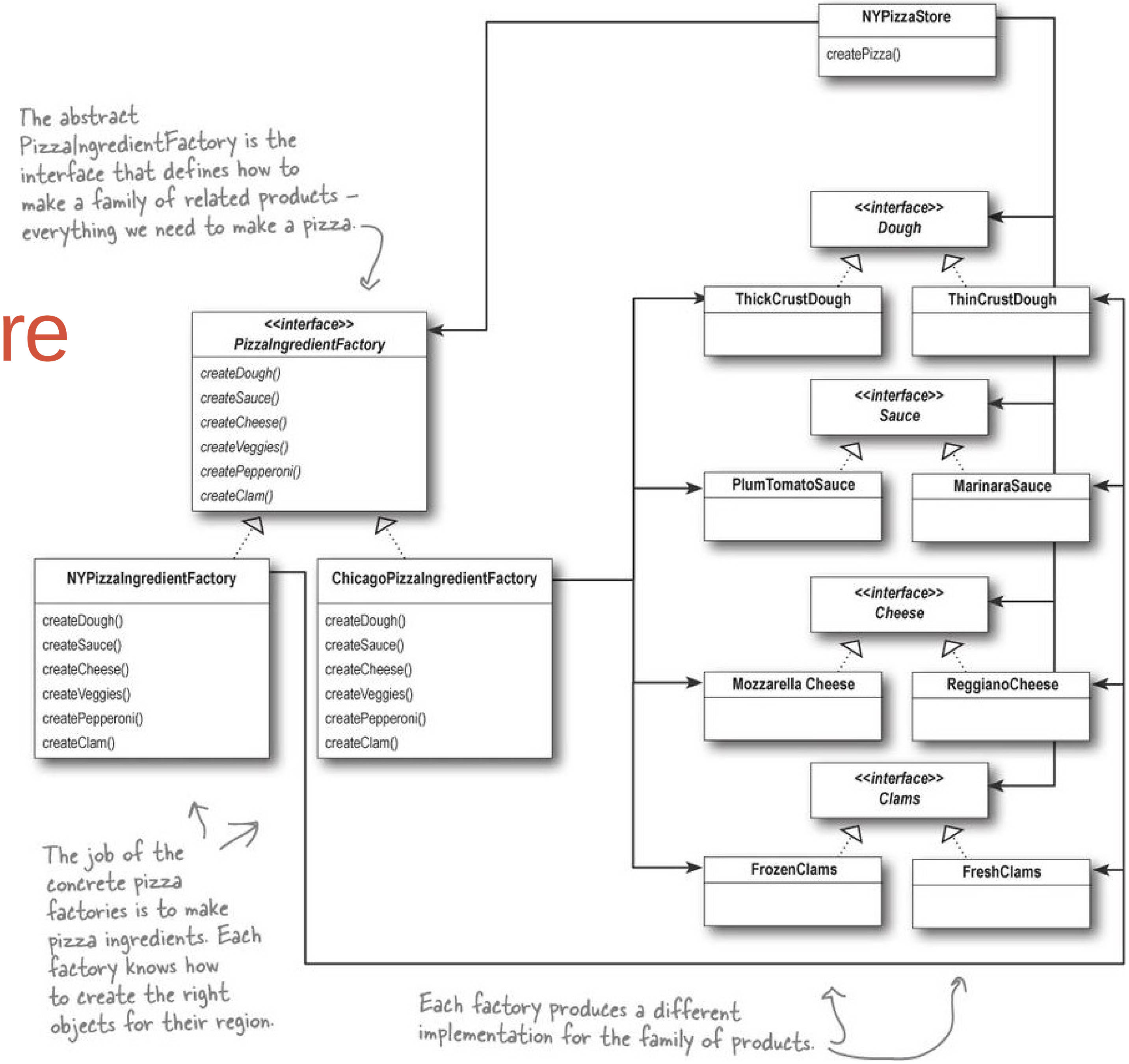
Finally, we have the prepared pizza in hand and the orderPizza() method bakes, cuts, and boxes the pizza.

Abstract Factory Defined

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


The Client is written against the abstract factory and then composed at runtime with an actual factory.
Client
The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.
<<interface>>
AbstractFactory
CreateProductA()
CreateProductB()
This is the product family. Each concrete factory can produce an entire set of products.
<<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.

Final PizzaStore Design



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
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()

ChicagoPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()

<<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.

Q3

Recap

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

The Abstract Factory Pattern provides mechanism to support a family of product. Because our code is decoupled from the actual products, we can substitute different factories to get different behaviors.