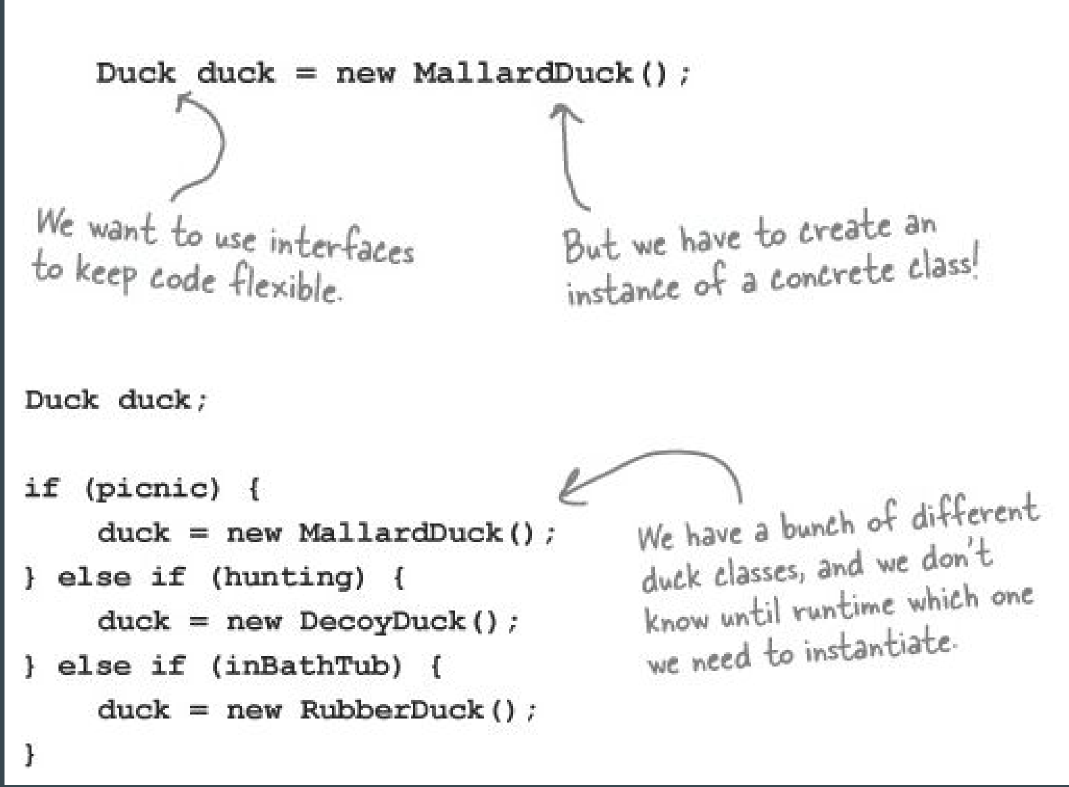
# Patterns 2

The Factory Pattern

# Another dependence

- In our previous pattern example, we programmed to an interface at the top (to retain flexibility), but we still have to instantiate these classes
- What if new types of ducks were added?
  - We'd have to go back into our code, which isn't good
- Wouldn't it be nice to stay abstract here?



```
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.

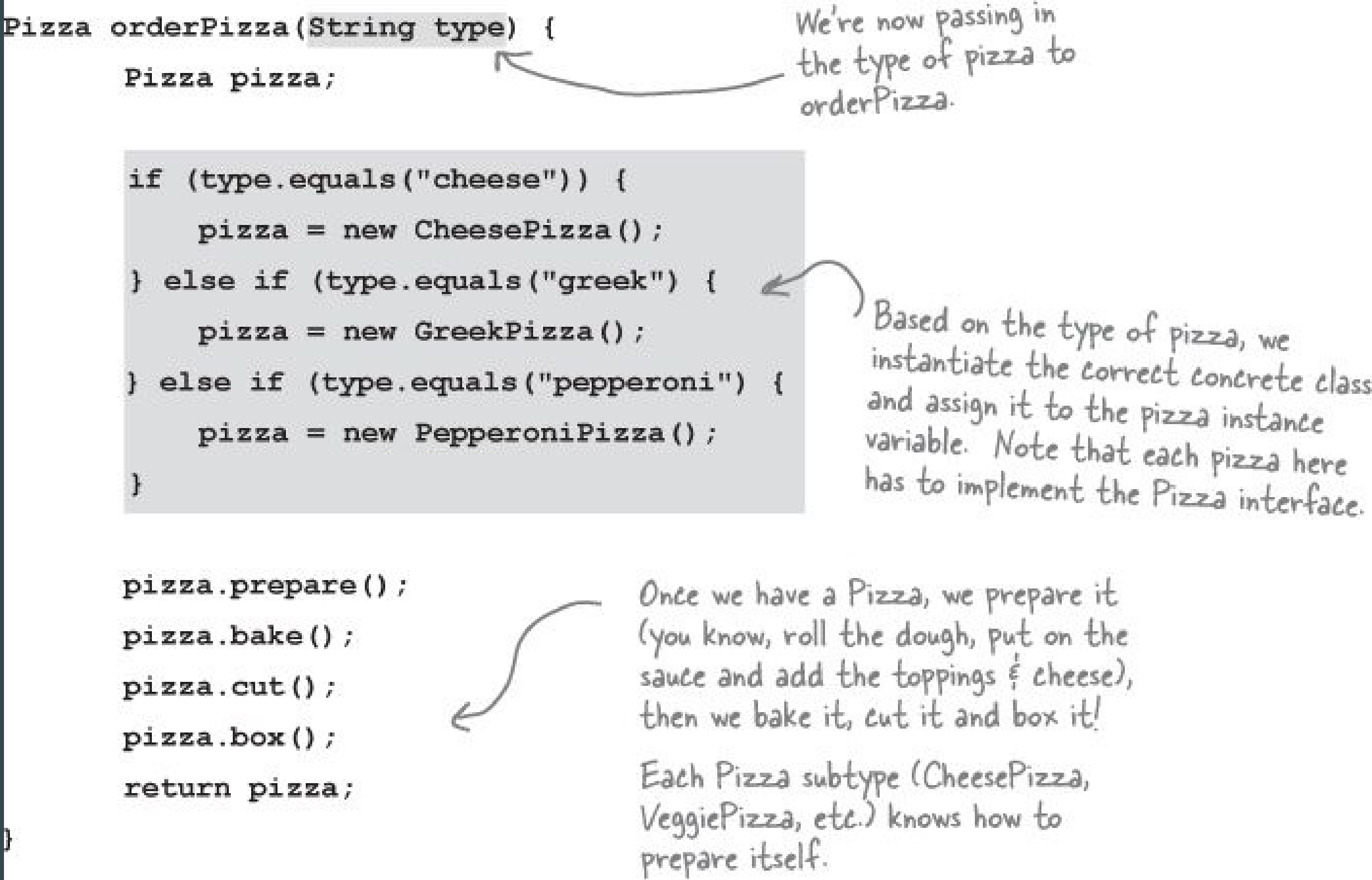
# A new example: PizzaStore

- PizzaStore is a class that has two main methods:
    - createPizza: creates pizzas
    - orderPizza: calls createPizza to get a pizza, then does various other operations before returning a pizza to the caller
    - orderPizza currently has the same problem as the Duck code we just examined

The store can be used in a program to order pizzas. In addition to this class, we start with pizza interfaces, and concrete instantiations of pizzas that can be served (more on these later). We will be adding to this to illustrate variations of the factory pattern.

# Insulating our code

- Identify what varies, and what doesn't, and separate them
- Consider a pizza store class with an orderPizza method that handles different pizza types
- What changes? What stays the same?
- What about clam pizzas?



```
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.

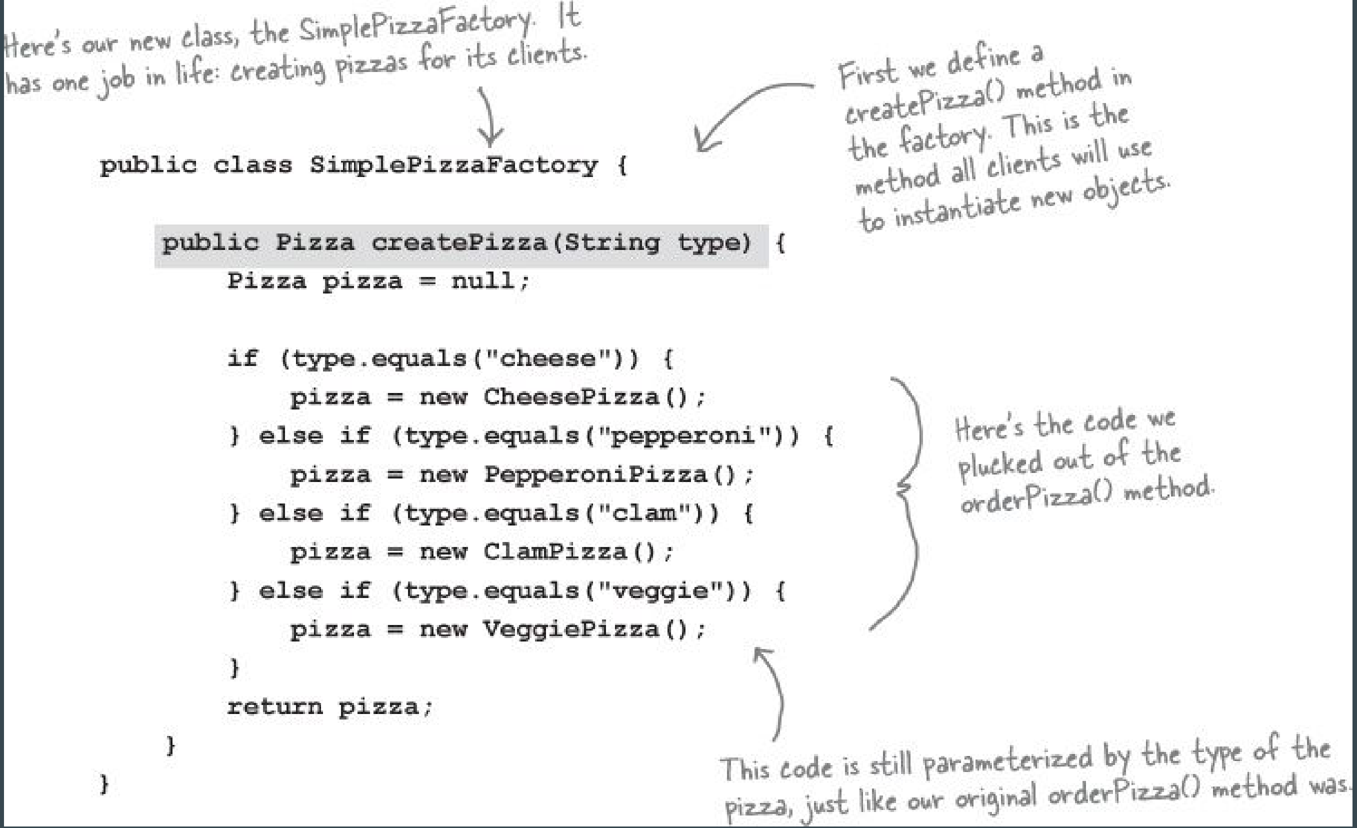
Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

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.

# First solution: a simple pizza factory

- This class will make the right kind of pizza for us, given a string
- This separates what changes (pizza creation) from what doesn't



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.

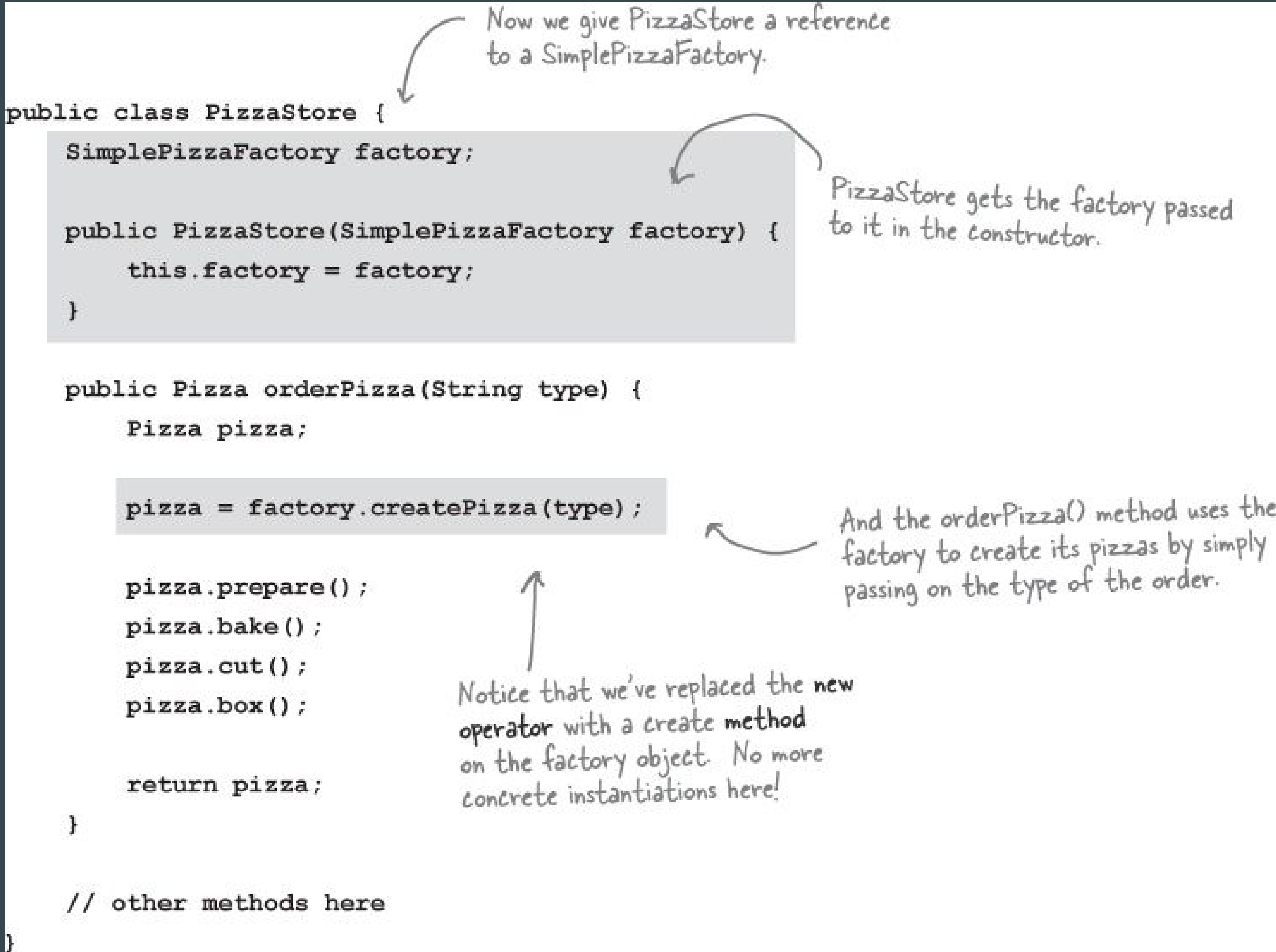
```
public class SimplePizzaFactory {

    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.

# The new PizzaStore class

- Another example of composition
- We can use the factory in other classes as well



```
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
}
```

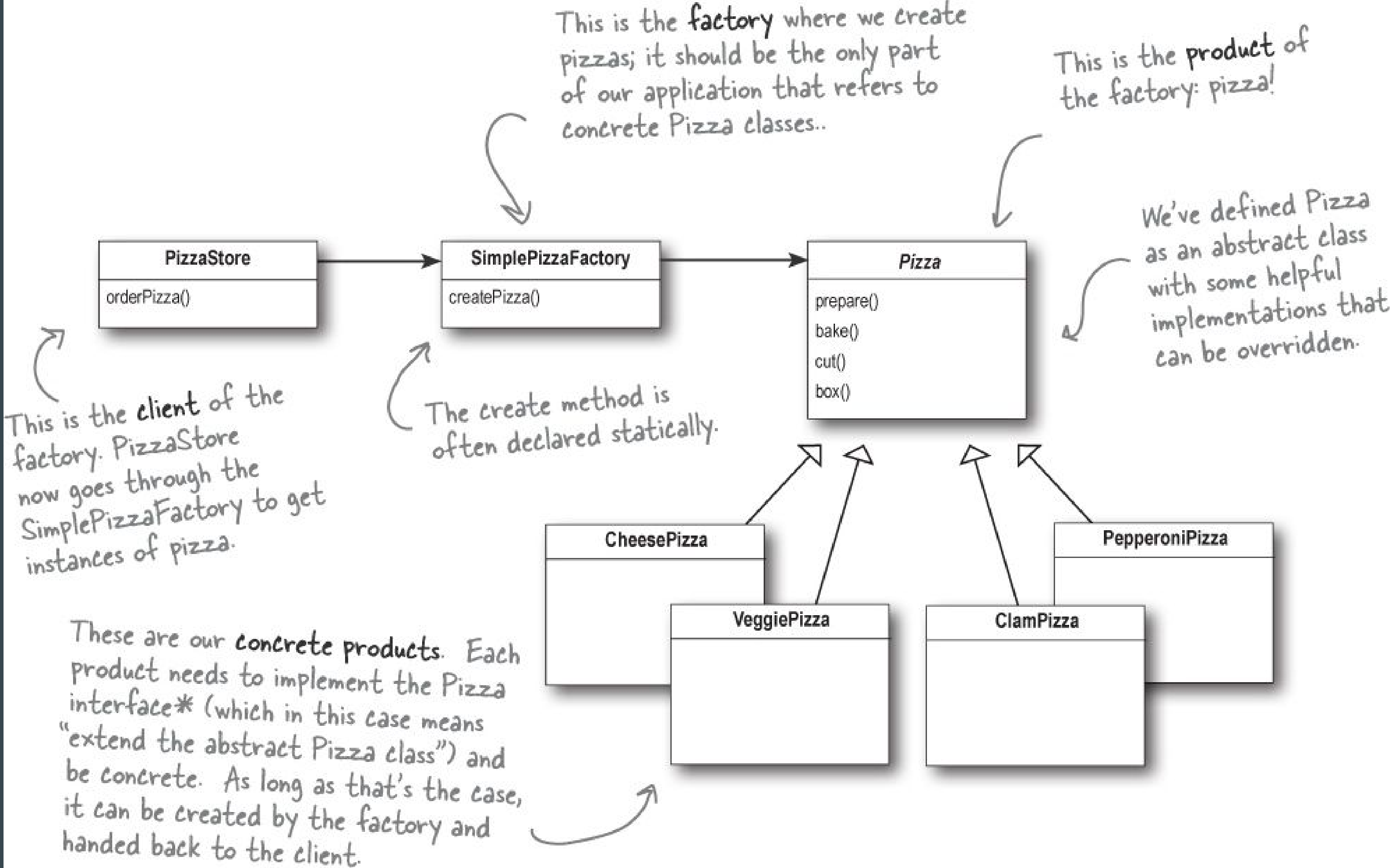
Now we give PizzaStore a reference to a SimplePizzaFactory.

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!

# Simple Factory Idiom UMLs



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.

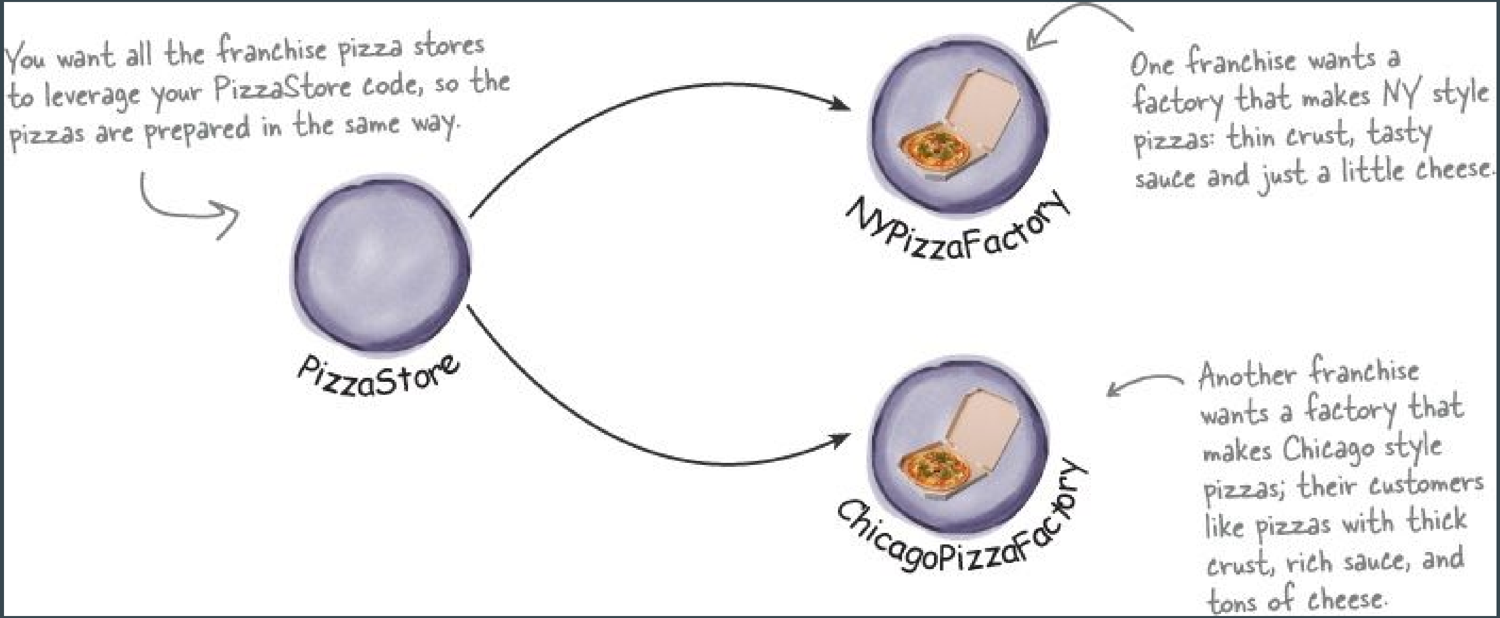
The create method is often declared statically.

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.

Note: it's not the factory pattern itself, but a commonly used idiom worth knowing

# Franchising the pizza store

- Different regions use different ingredients
- But we want them to use the same PizzaStore code

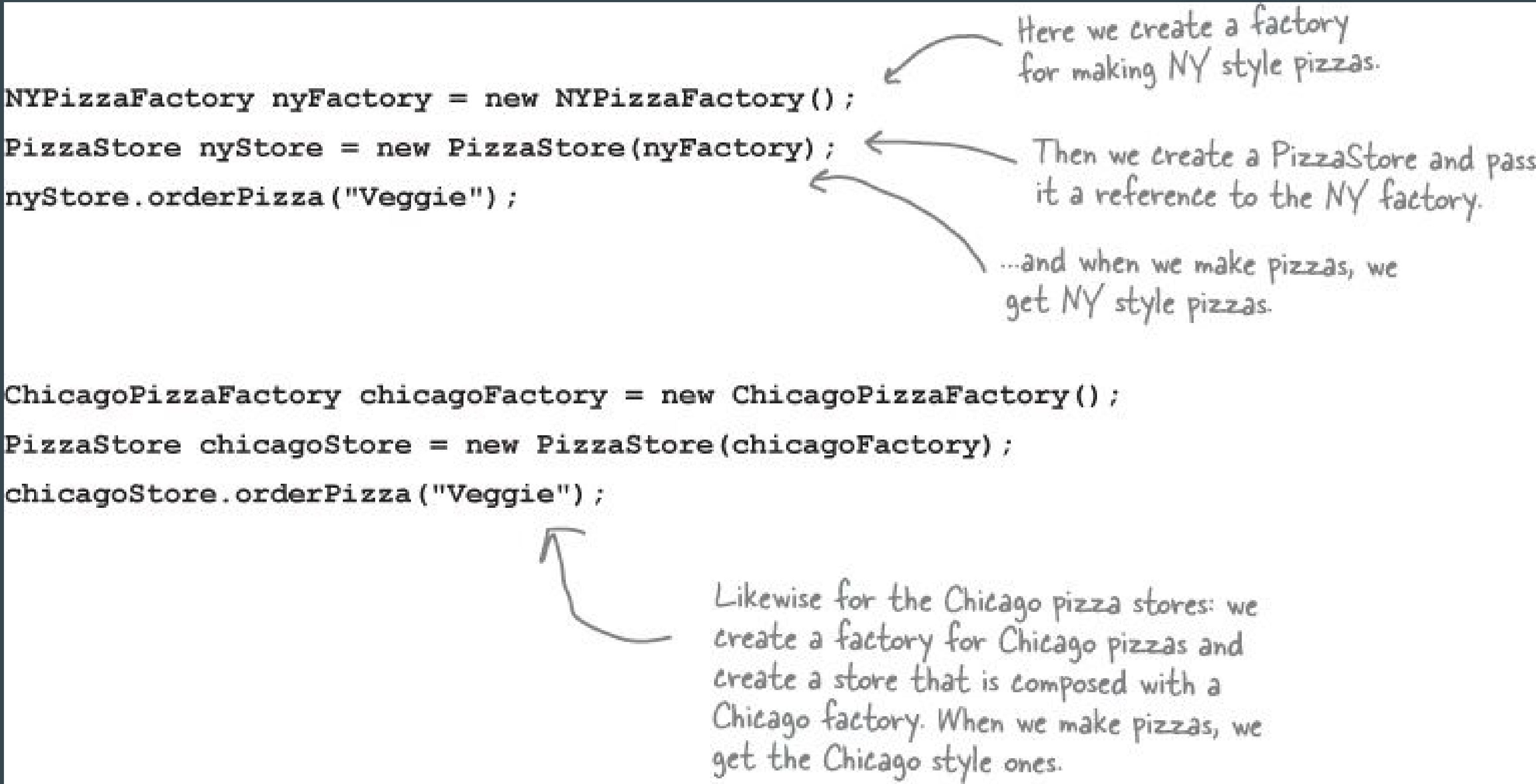


You want all the franchise pizza stores to leverage your PizzaStore code, so the pizzas are prepared in the same way.

One franchise wants a factory that makes NY style pizzas: thin crust, tasty sauce and just a little cheese.

Another franchise wants a factory that makes Chicago style pizzas; their customers like pizzas with thick crust, rich sauce, and tons of cheese.

# Using the simple factory idiom



```
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.

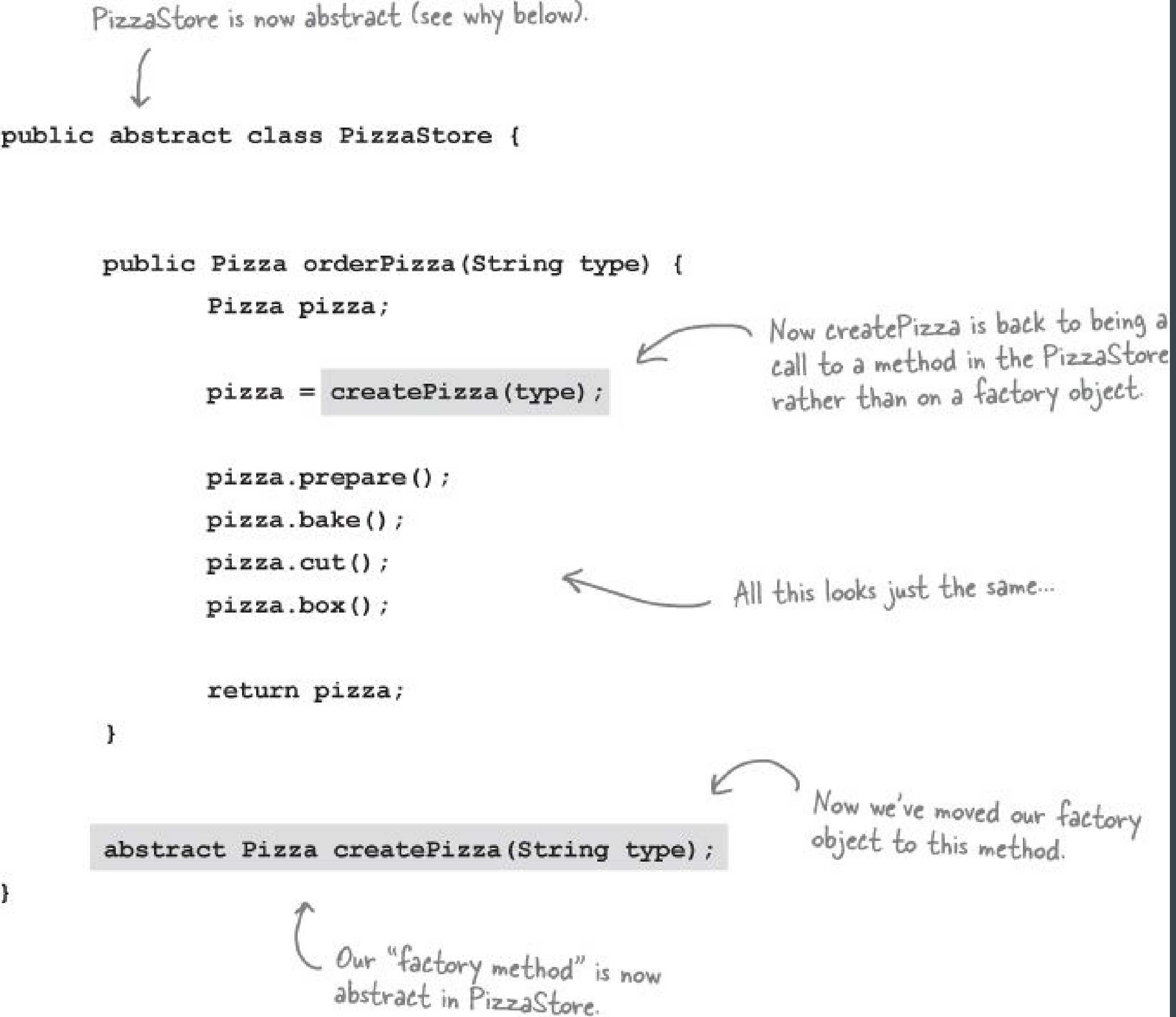
...and when we make pizzas, we get NY style pizzas.

```
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.

# Adding regional franchises

- We need now to manage franchises with different styles of ingredients
- Put createPizza back in PizzaStore, but make it abstract
- Let subclasses handle different regions



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);
}
```

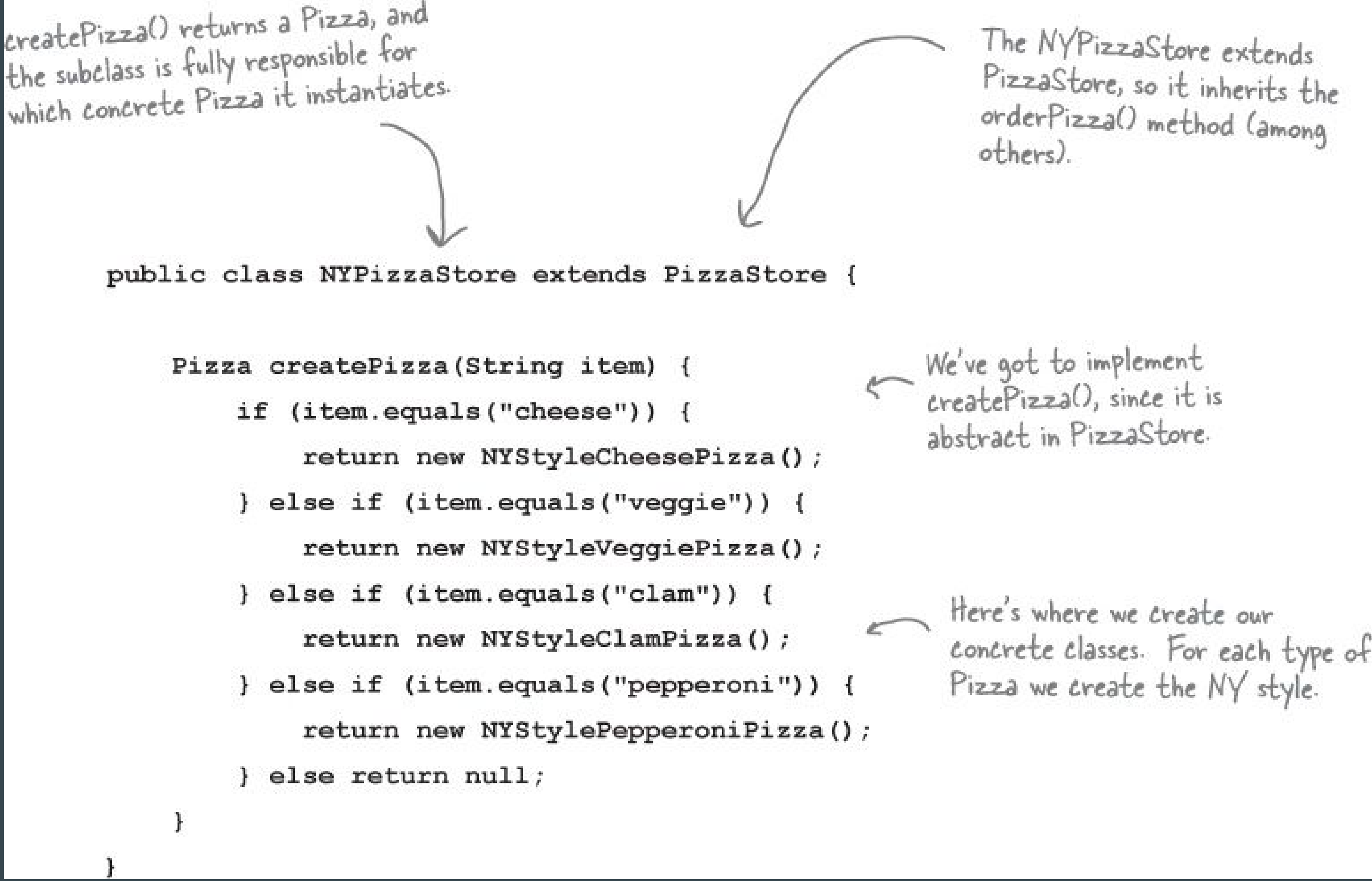
Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

All this looks just the same...

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

Our "factory method" is now abstract in PizzaStore.

# Letting a subclass determine which regional style to use



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;
    }
}
```

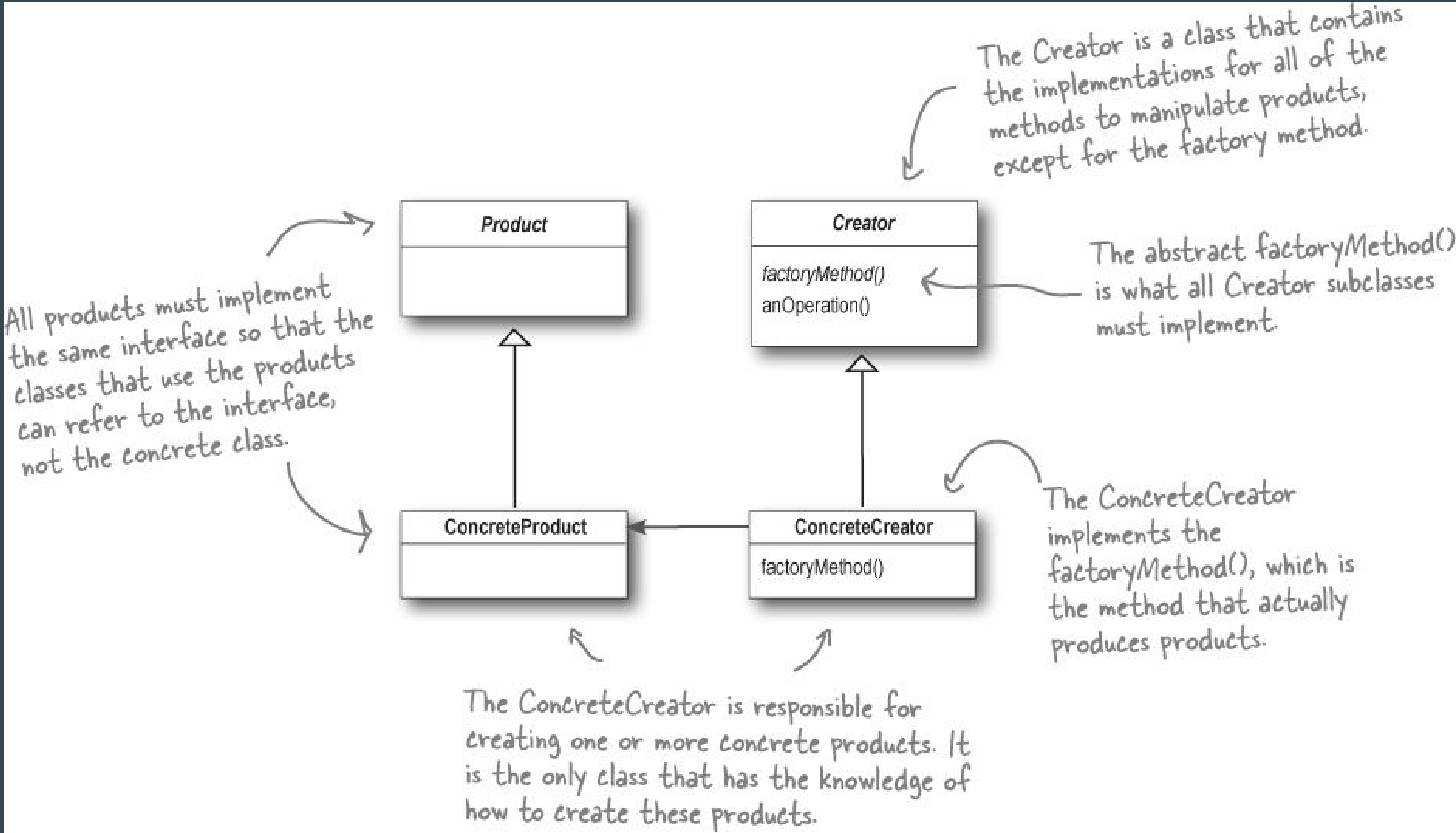
We've got to implement createPizza(), since it is abstract in PizzaStore.

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

# Factor Method Pattern Defined

**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.

# UMLs



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.

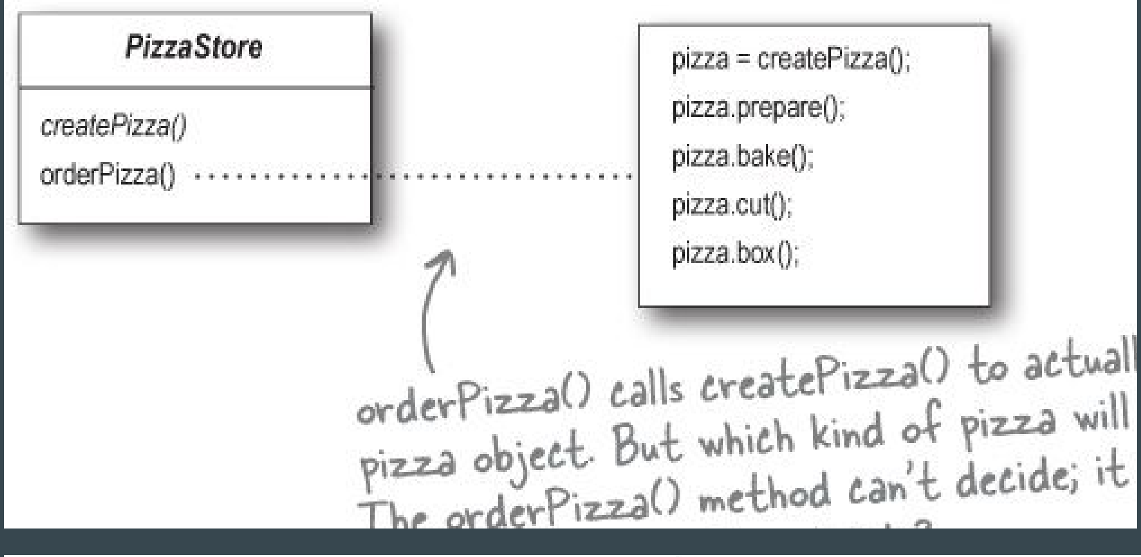
The abstract factoryMethod() is what all Creator subclasses must implement.

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.

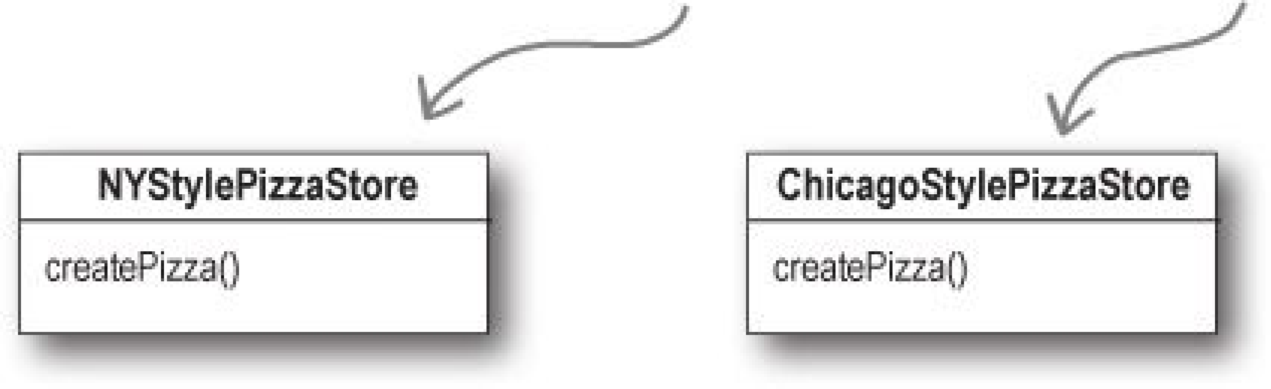
The subclasses below implement the abstract method in the abstract PizzaStore class

Which kind of pizza is made is decided by which subclass is used



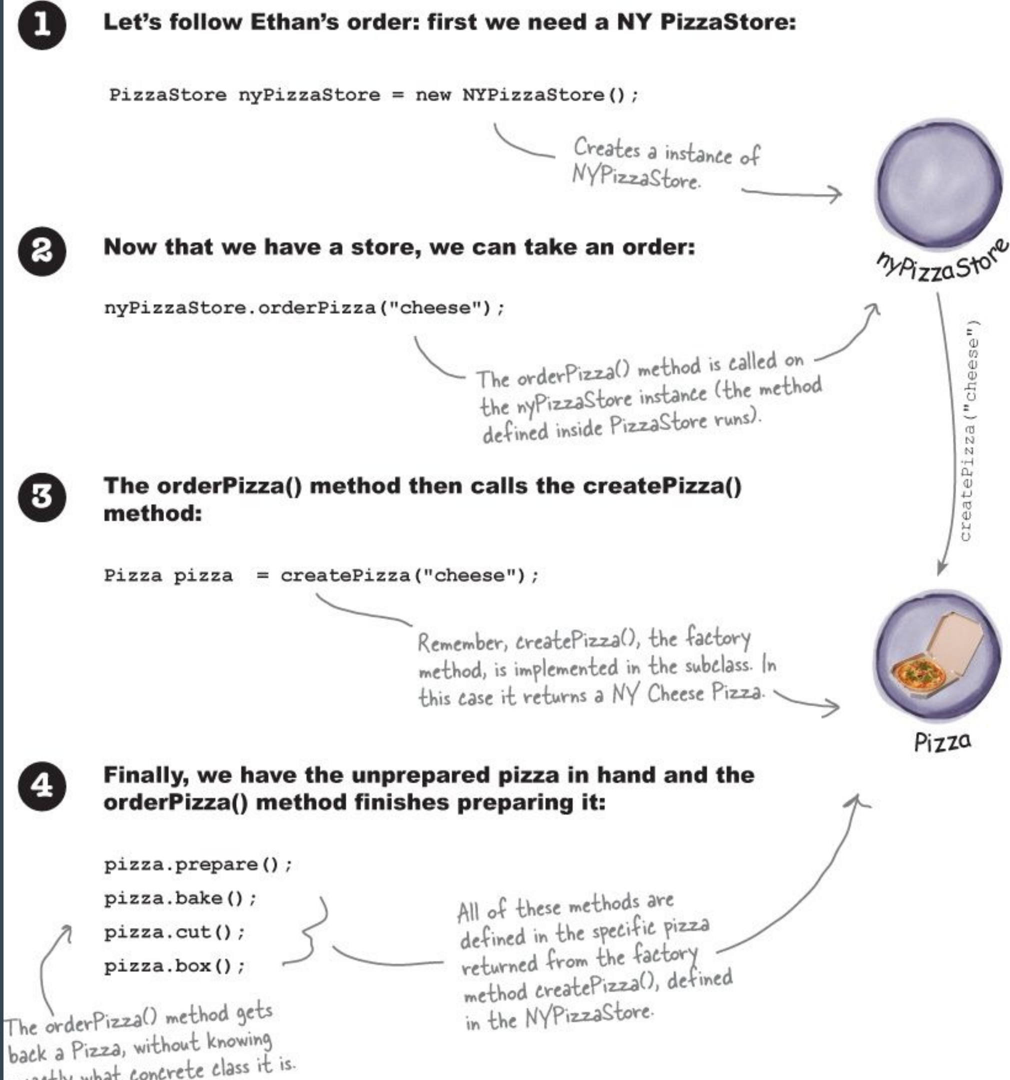
```
pizza = createPizza();
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
```

orderPizza() calls createPizza() to actuall
pizza object. But which kind of pizza will
The orderPizza() method can't decide; it



# Using the subclasses

The diagram on the right shows the order in which calls are made



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).

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();
```

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

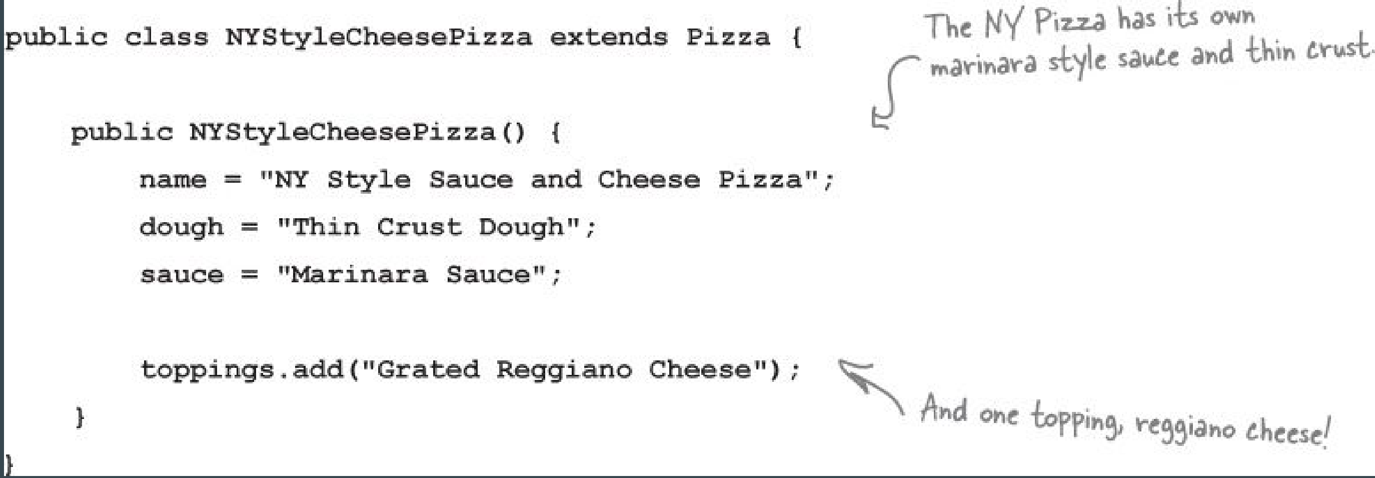
The orderPizza() method gets back a Pizza, without knowing exactly what concrete class it is.

# Pizza Classes

- Too much code for slides: see code example
- To download the code for all the examples in the textbook, go to http://www.headfirstlabs.com/books/hfdp/

# Pizza Subclasses:

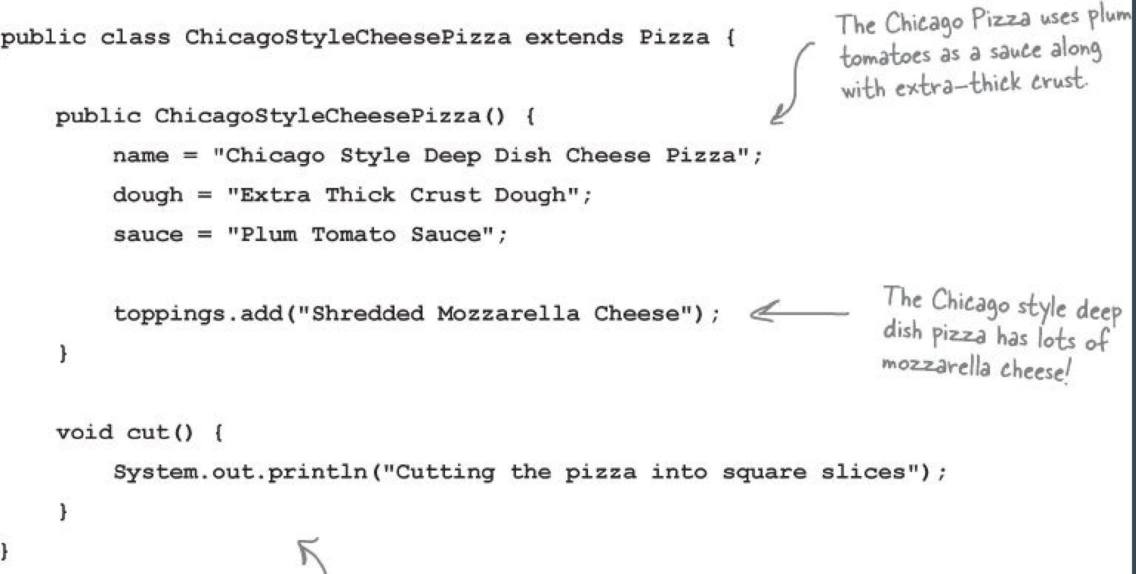
Each has different ingredients



```
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!

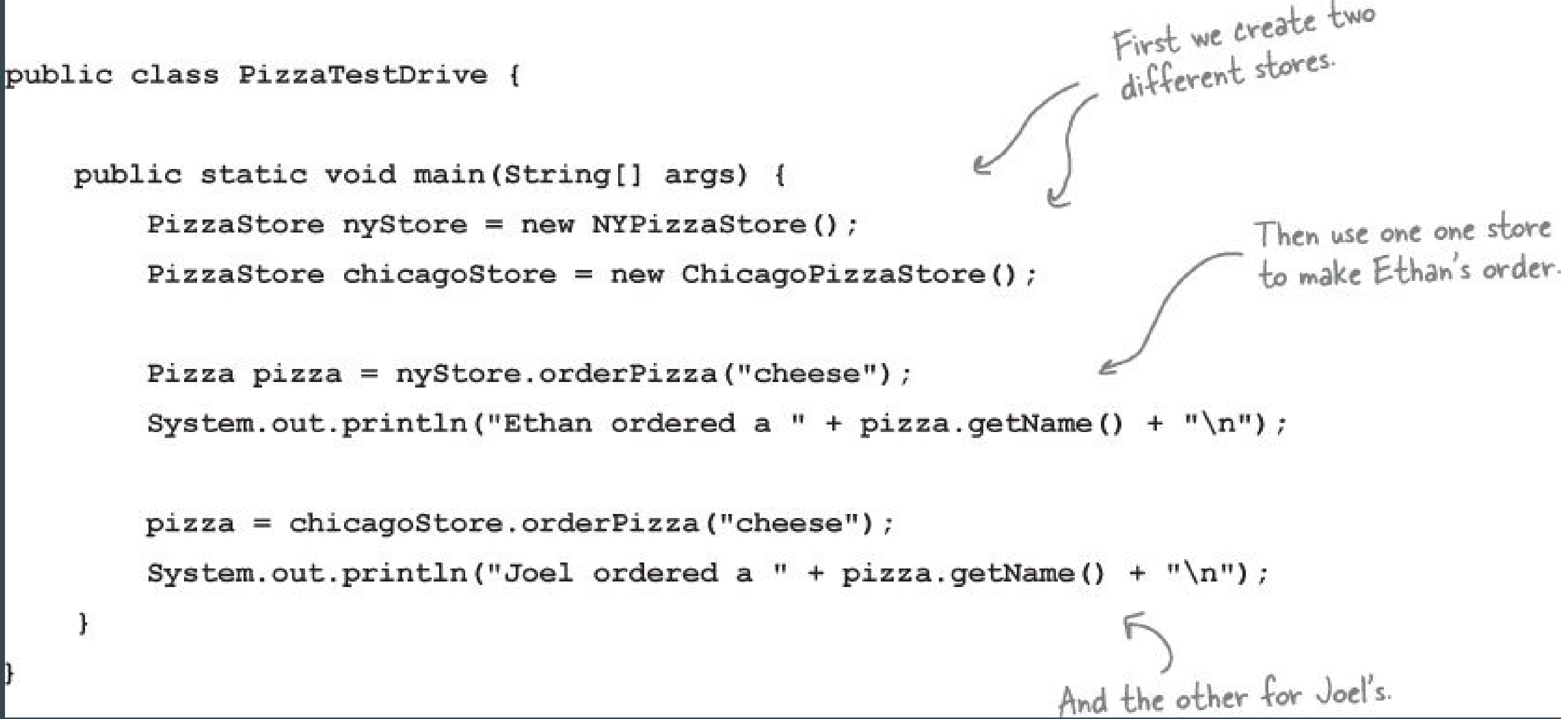


```
public class ChicagoStyleCheesePizza extends Pizza {

    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!

# A test program



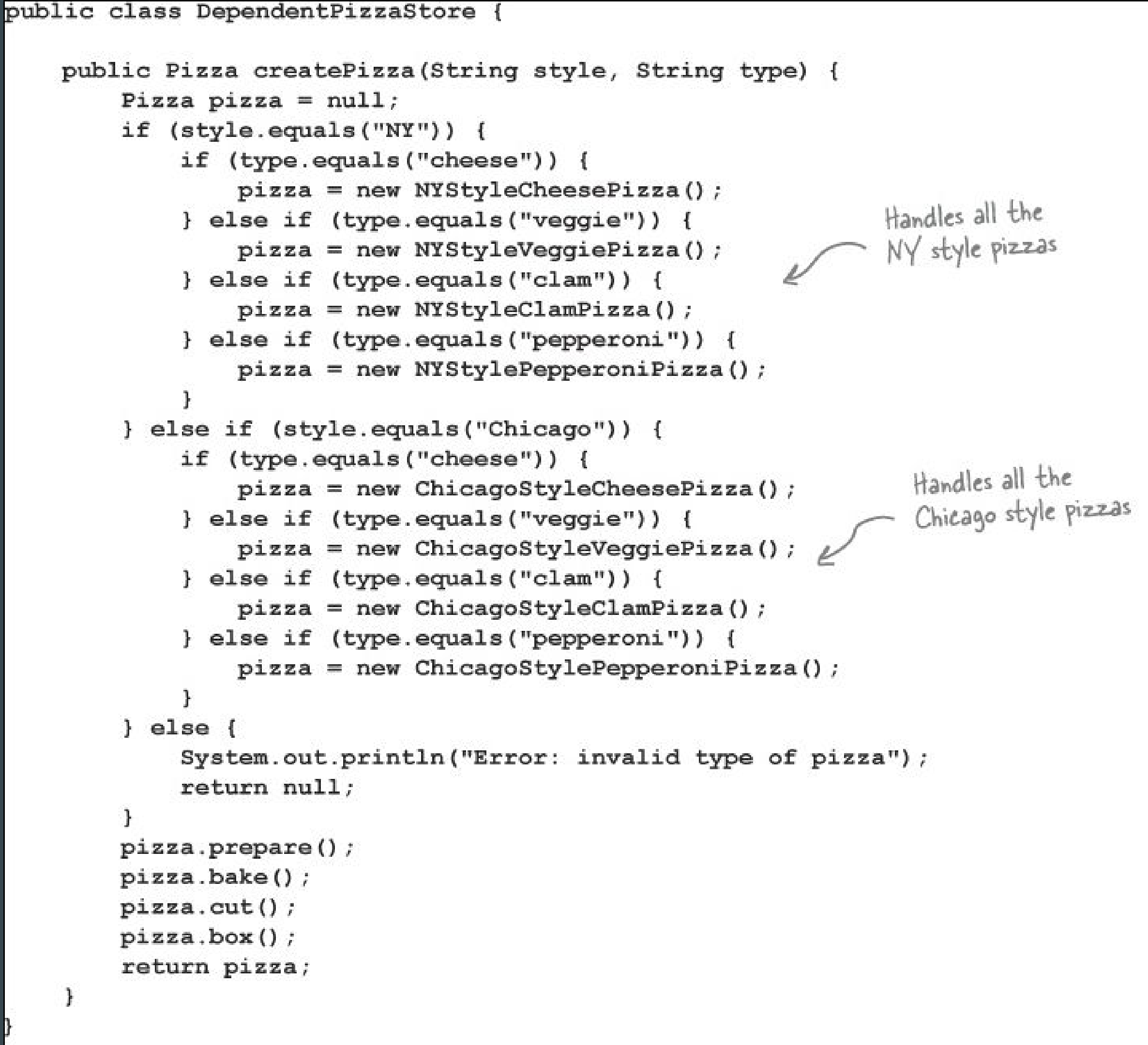
```
public class PizzaTestDrive {

    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 one store to make Ethan's order.

And the other for Joel's.

What if we didn’t follow the above pattern?

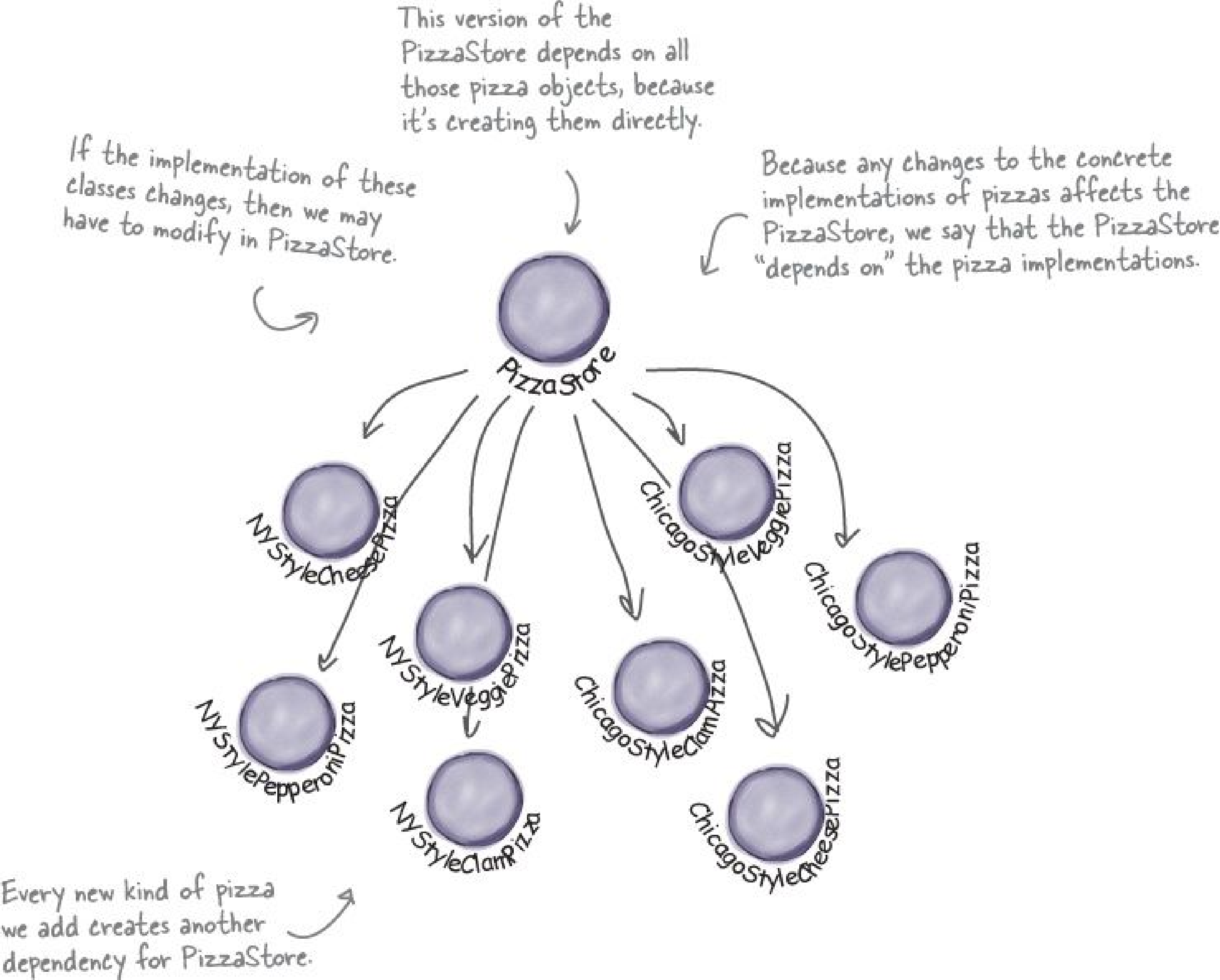


```
public class DependentPizzaStore {

    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

- The previous code creates dependencies in the PizzaStore on lots of more specific components
- If we add or subtract any of these, we have to change the code in PizzaStore



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.

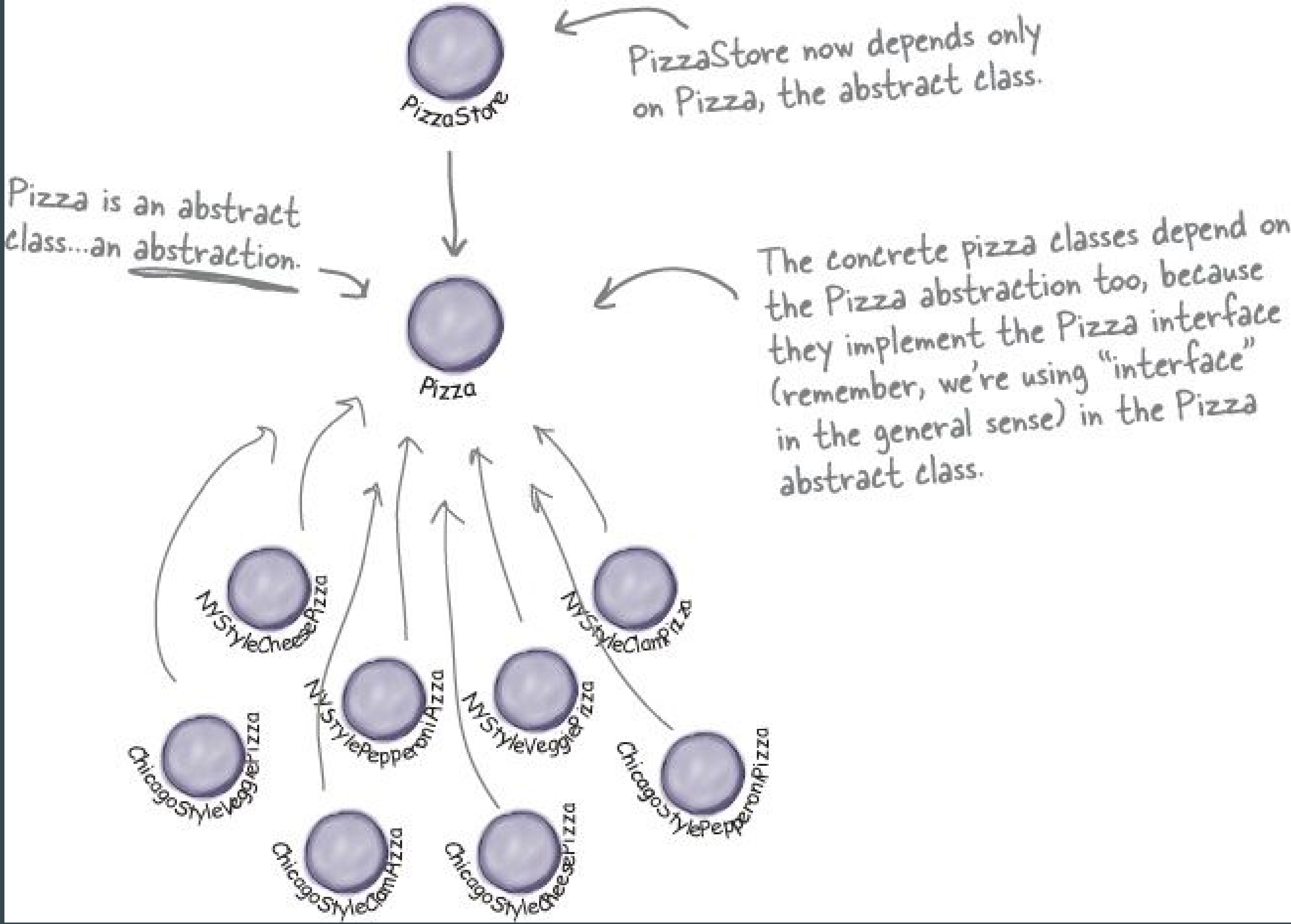
Every new kind of pizza we add creates another dependency for PizzaStore.

# Dependency Inversion Principle

- Depend upon abstractions. Do not depend on concrete classes.
- High level components (where behavior is defined in terms of low-level components) should not depend on *concrete* low level components; they should both depend on abstractions
- We do this for flexibility, realizing that code changes

(If code never changed, we wouldn’t need any of these patterns!)

Much better: All the components depend on Pizza, an abstraction (either an interface or abstract class)


PizzaStore now depends only on Pizza, the abstract class.
PizzaStore
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

# Some dependency inversion guidelines

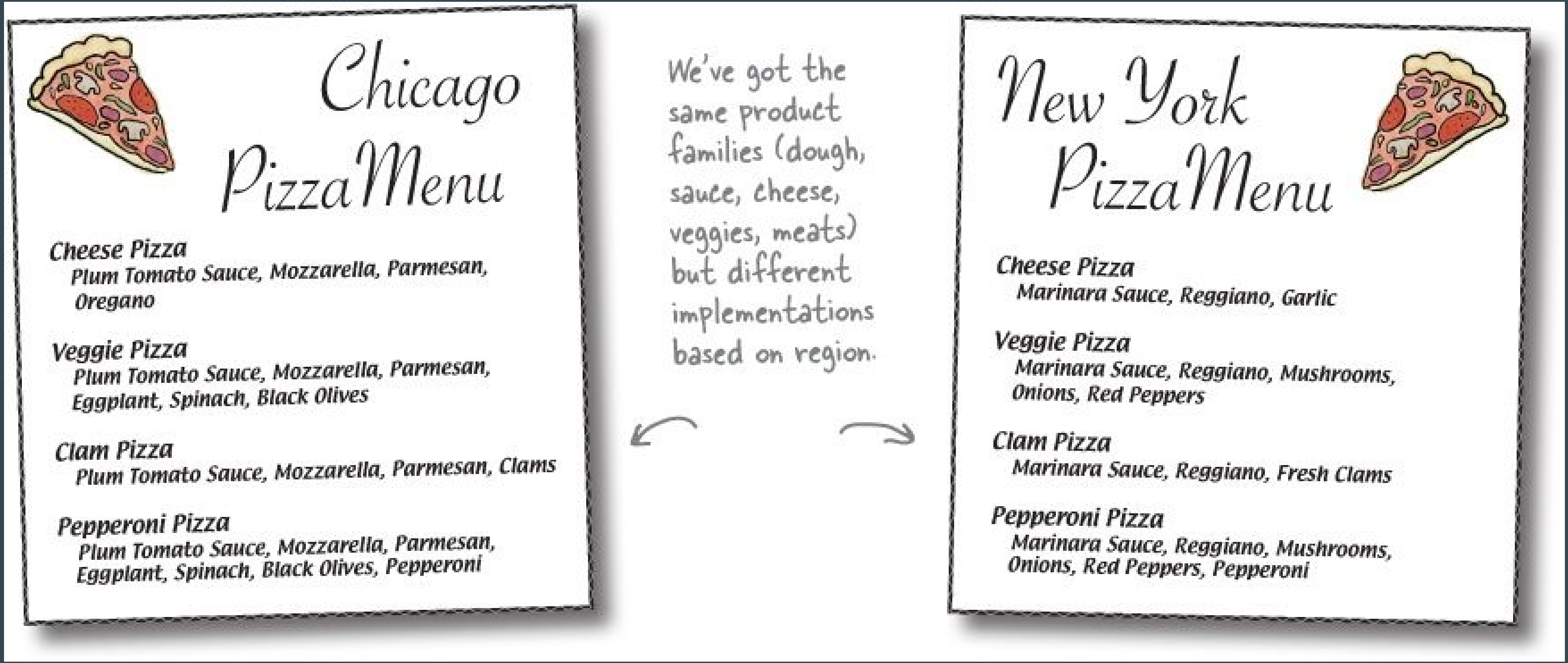
- No variable should hold a reference of a concrete class
- No class should derive from a concrete class
- No method should override an implemented method of any of its base classes

Note: you can’t actually follow all these, but try to as much as you can, especially when the classes are likely to change

# Another emergency

- Some of the local franchises have been substituting inferior ingredients into their pizzas to save money and increase their profits
- We need to build a factory that produces the ingredients and ships them to the franchises
- We will need a new pattern: the Abstract Factory

# All the regions have different versions of the same ingredients



Chicago PizzaMenu

**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 PizzaMenu

**Cheese Pizza**
Marinara Sauce, Reggiano, Garlic

**Veggie Pizza**
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers

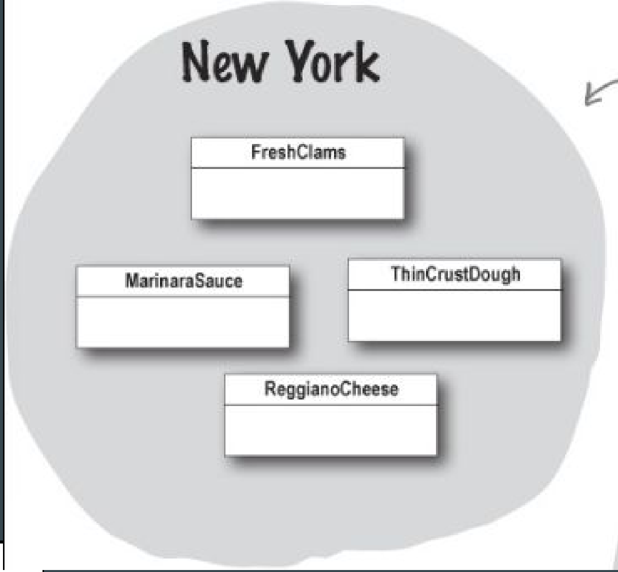
**Clam Pizza**
Marinara Sauce, Reggiano, Fresh Clams

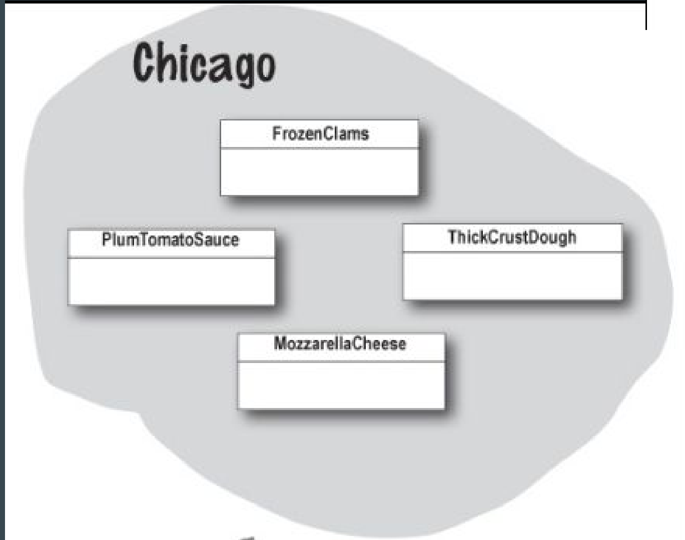
**Pepperoni Pizza**
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers, Pepperoni

# How to handle this

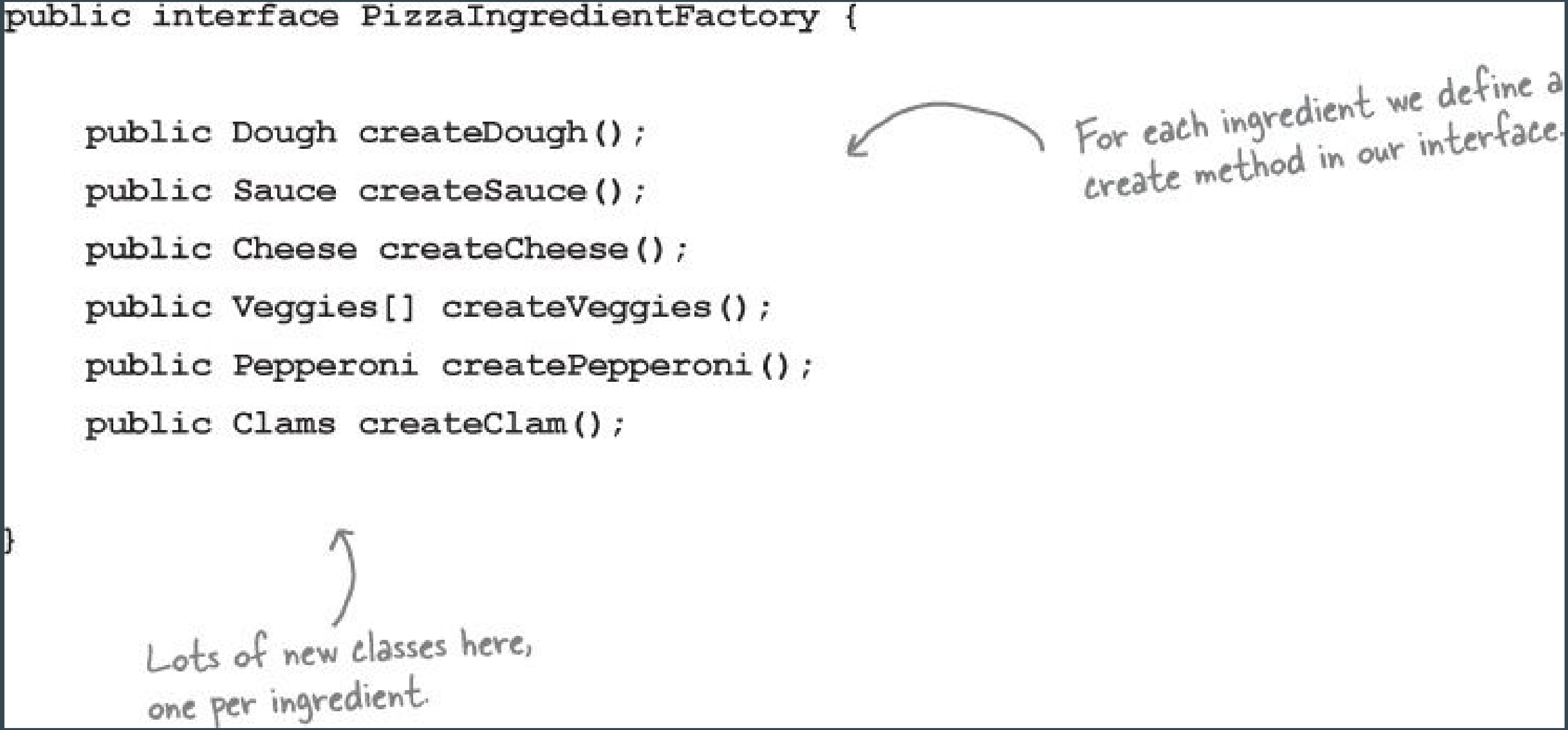
- Each has:
- Seafood
- Sauce
- Dough
- Cheese

Abstract these concepts





# Here’s the ingredient abstraction



```
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.

# The process

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.

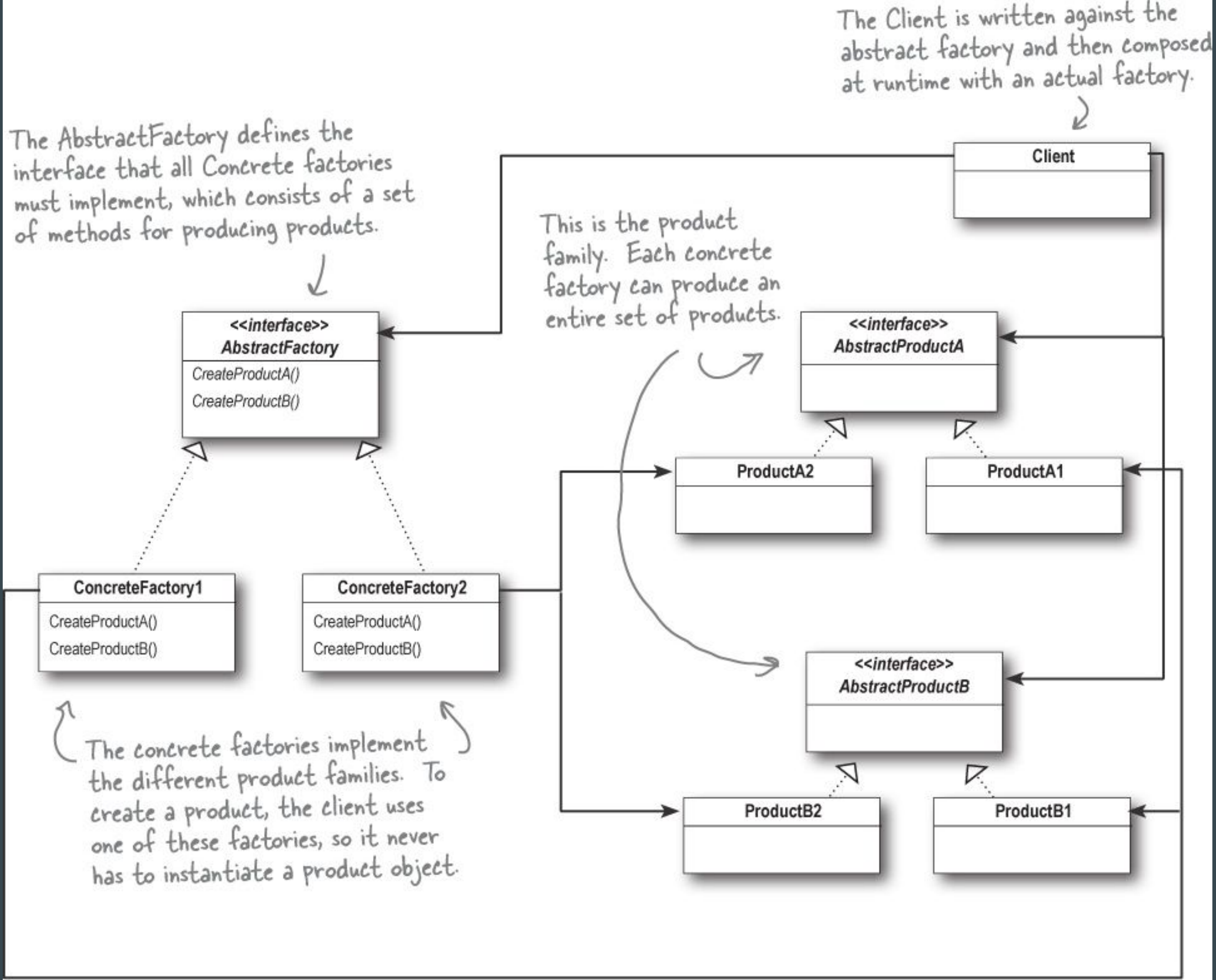
③ Then we still need to hook all this up by working our new ingredient factories into our old PizzaStore code.

# NYPizzaIngredientFactory, Pizza, and PizzaStore class changes

- Too much code for slides, but it basically implements the PizzaIngredientFactory and works the appropriate changes into Pizza by making prepare() abstract and having the ingredients set with implementations of this method. It also creates and the appropriate ingredient factory in the PizzaStore class
- See code demo
- Do the same for any other region with its own versions of these ingredients

# Abstract Factory Pattern defined

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.

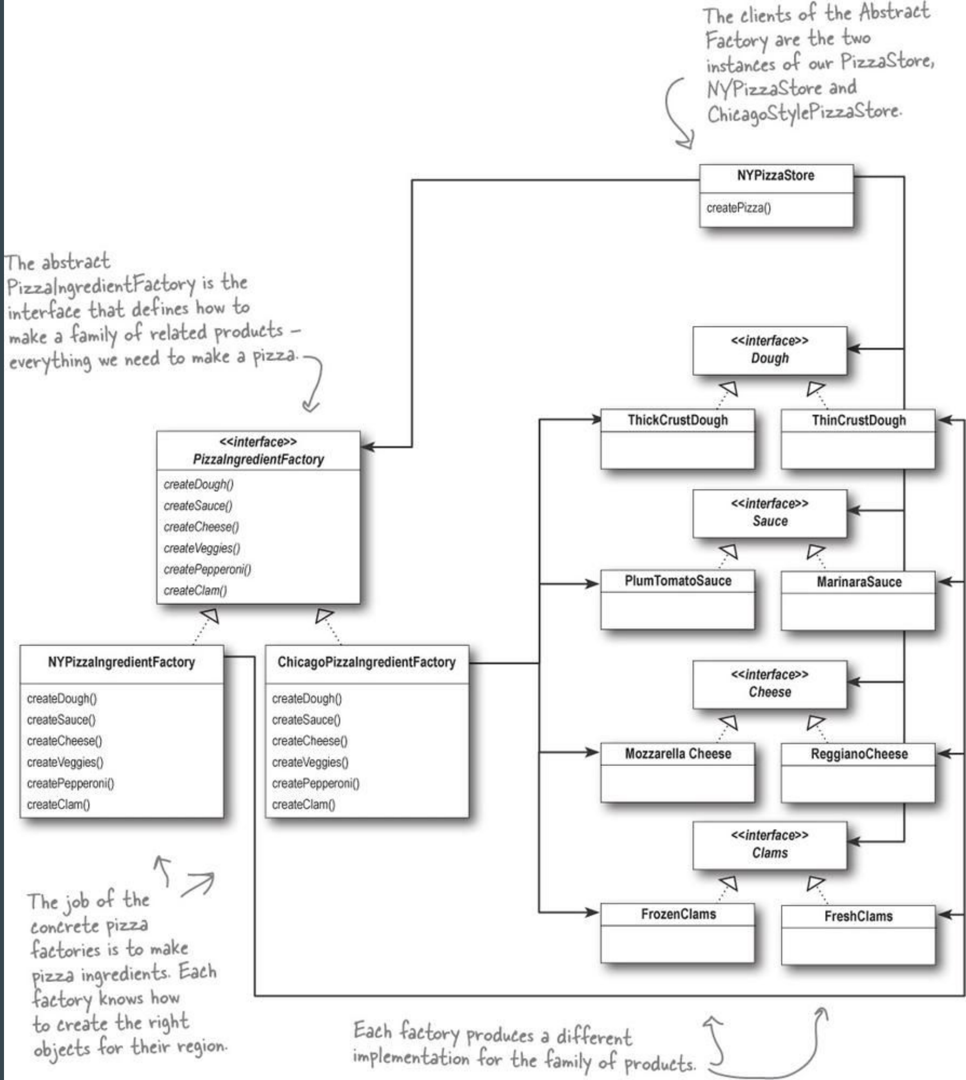
The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.

This is the product family. Each concrete factory can produce an entire set of products.

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.

# A pizza example of the previous diagram

This is useful when there are families of products that go together as a set, and you want to be able to switch these out quickly with a minimum of code



The clients of the Abstract Factory are the two instances of our PizzaStore, NYPizzaStore and ChicagoStylePizzaStore.

The abstract PizzaIngredientFactory is the interface that defines how to make a family of related products – everything we need to make a pizza.

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.

# How all this fits together: see code demo

Download the code yourself at: http://www.headfirstlabs.com/books/hfdp/