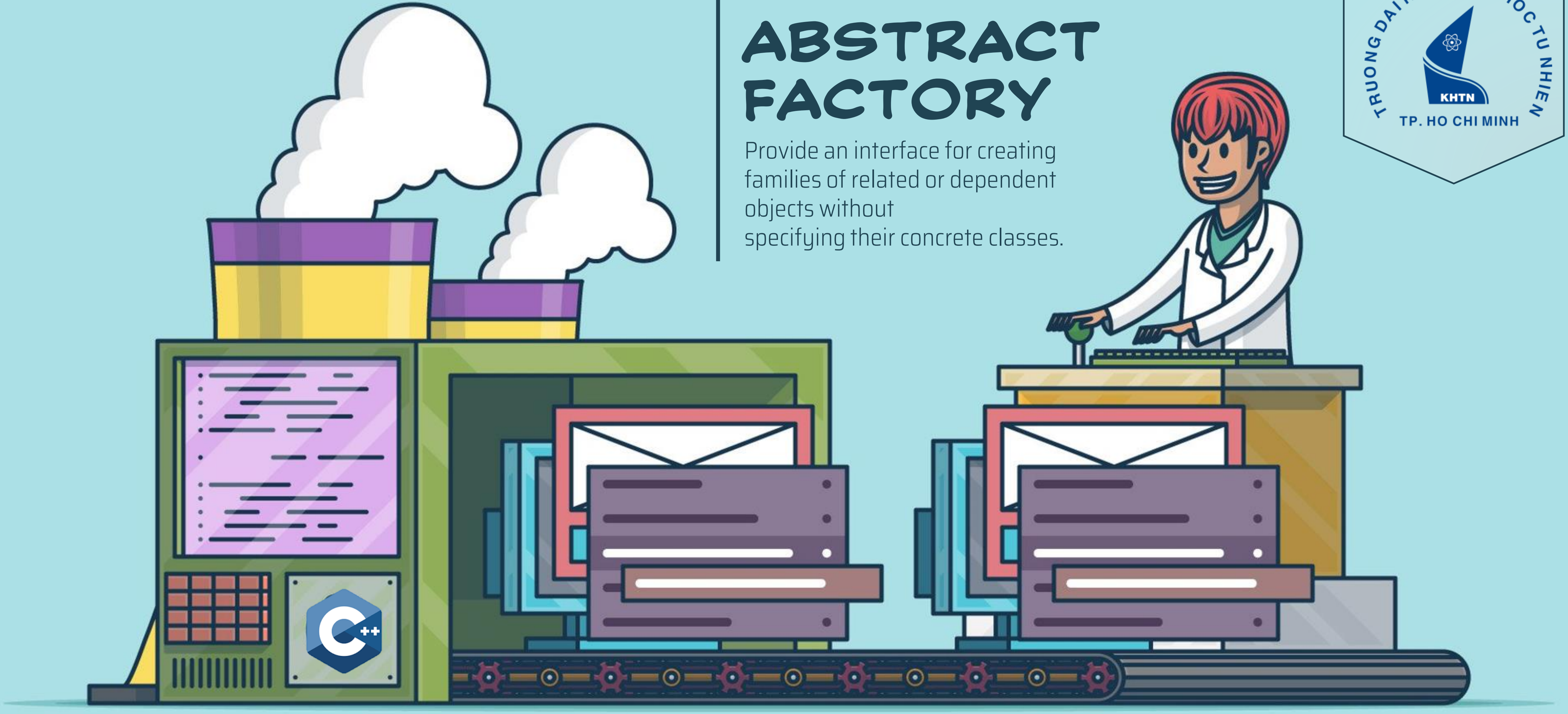
TRUONG DAI HOC KHOA HOC TU NHIEN
KHTN
TP. HO CHI MINH

ABSTRACT FACTORY

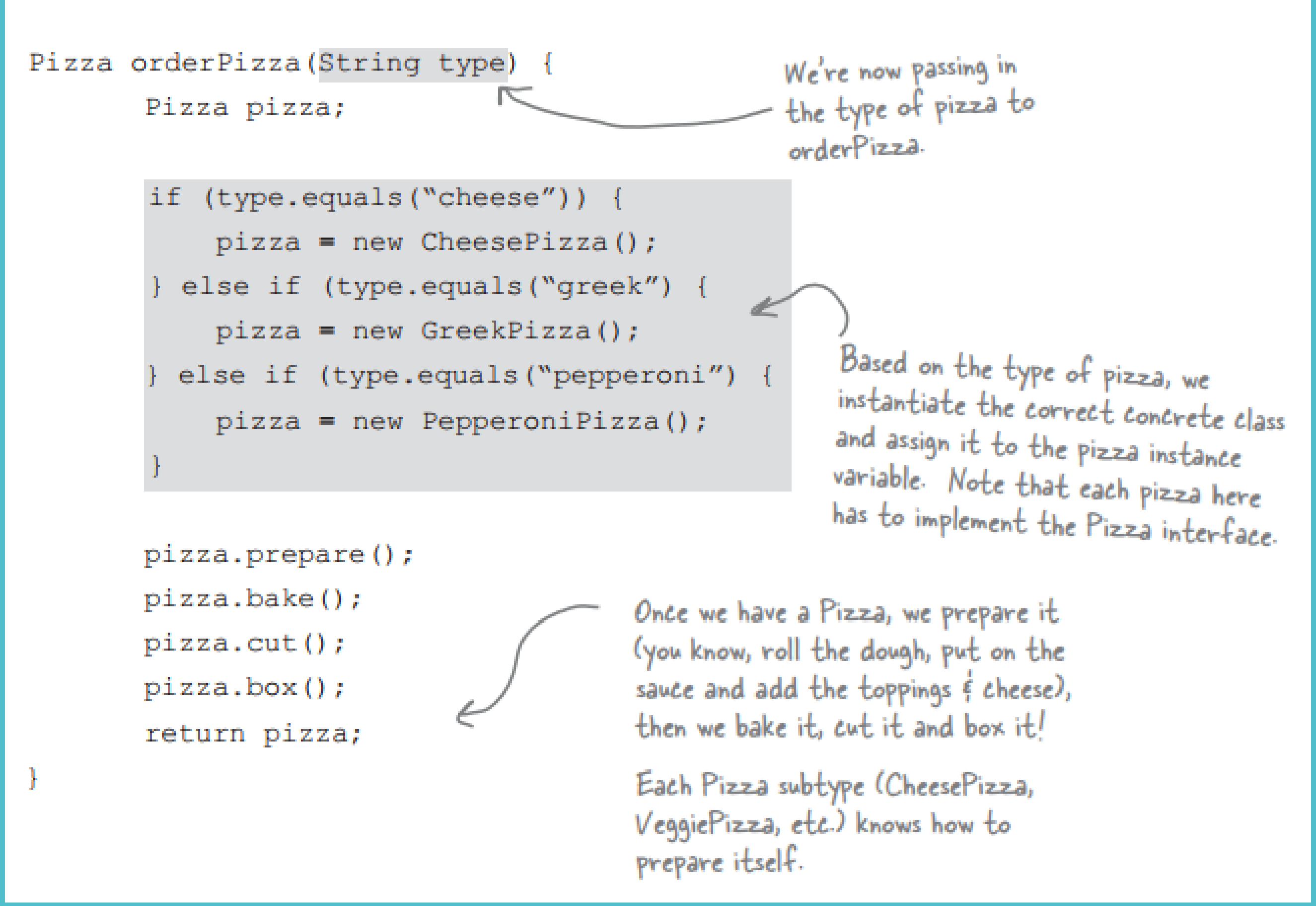
Provide an interface for creating families of related or dependent objects without
specifying their concrete classes.



Resources: bit.ly/OOAD_AF
Or: https://github.com/t3bol90/AbstractFactory

I. PIZZASTORE PROBLEM

Let’s say you have a pizza shop and as a cutting-edge pizza store owner, you have to handle with “The Pizza Store Process” (... create pizza -> prepare -> bake -> cut -> box ...). There are 3 types of Pizza: Cheese, Pepperoni, Greek.



```
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;
}
```

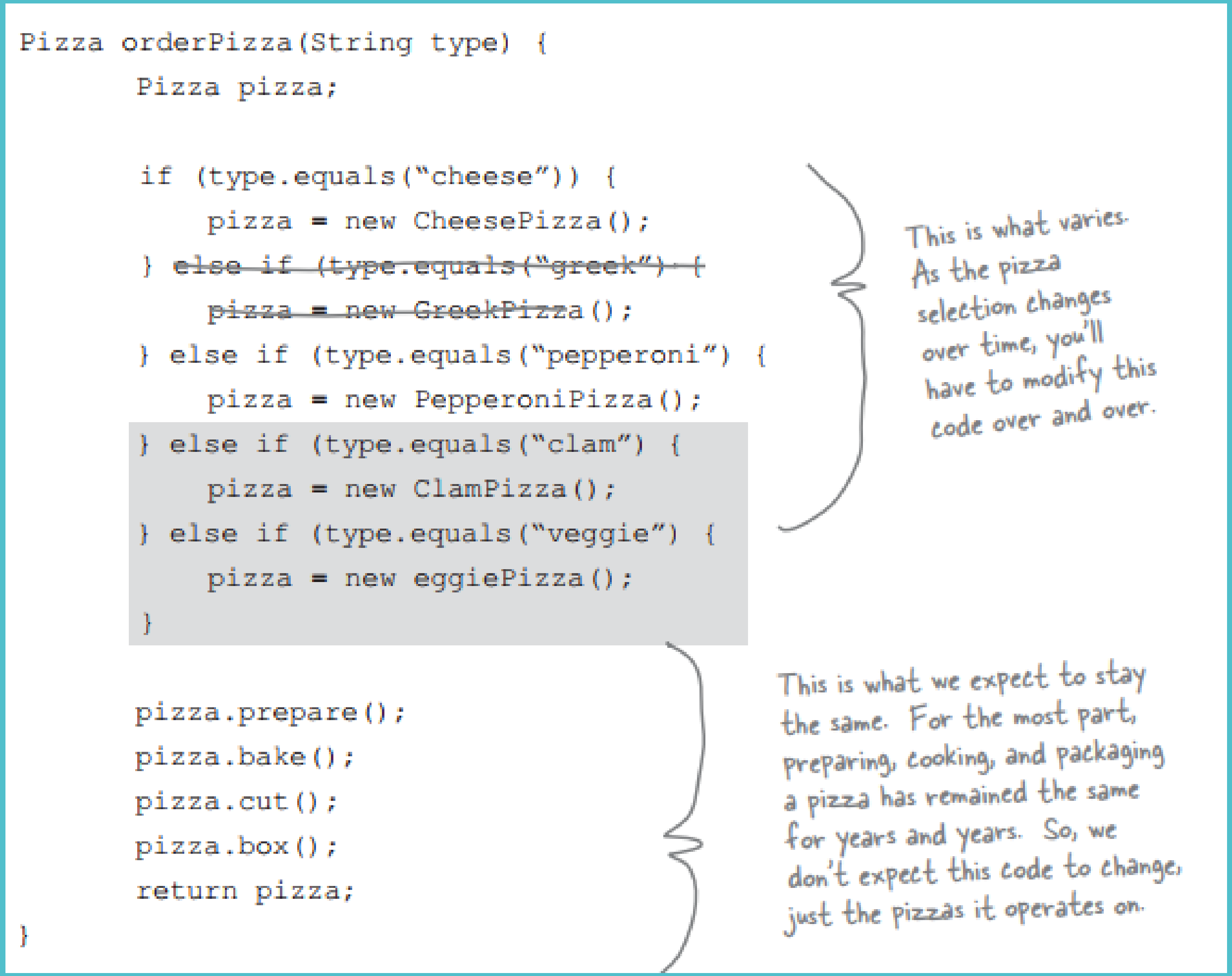
We’re now passing in the type of pizza to orderPizza.

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.

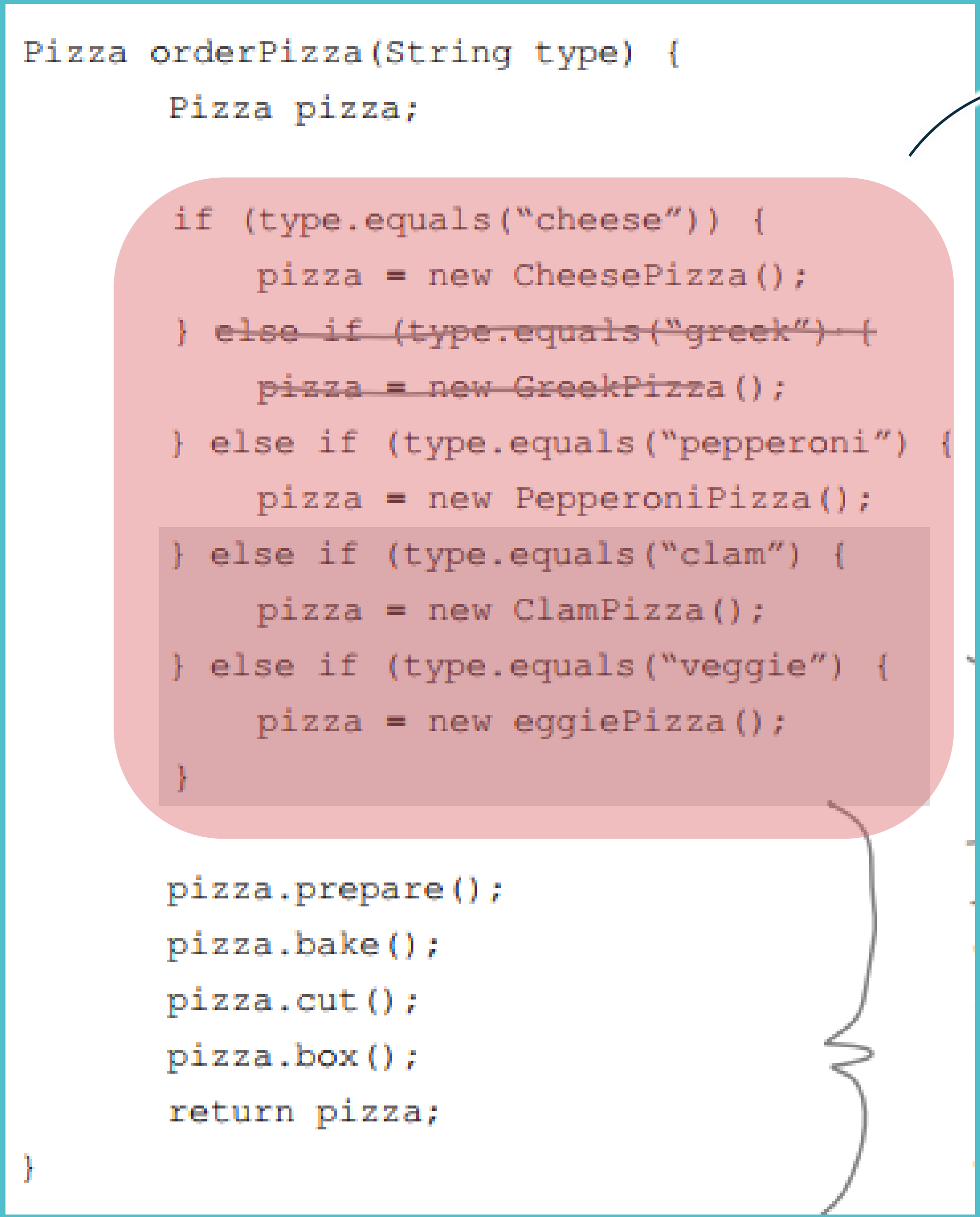
You realize that all of your competitors have added a couple of trendy pizzas to their menus: the Clam Pizza and the Veggie Pizza. Obviously you need to keep up with the competition, so you'll add these items to your menu. And you haven't been selling many Greek Pizzas lately, so you decide to take that off the menu.



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

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

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

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

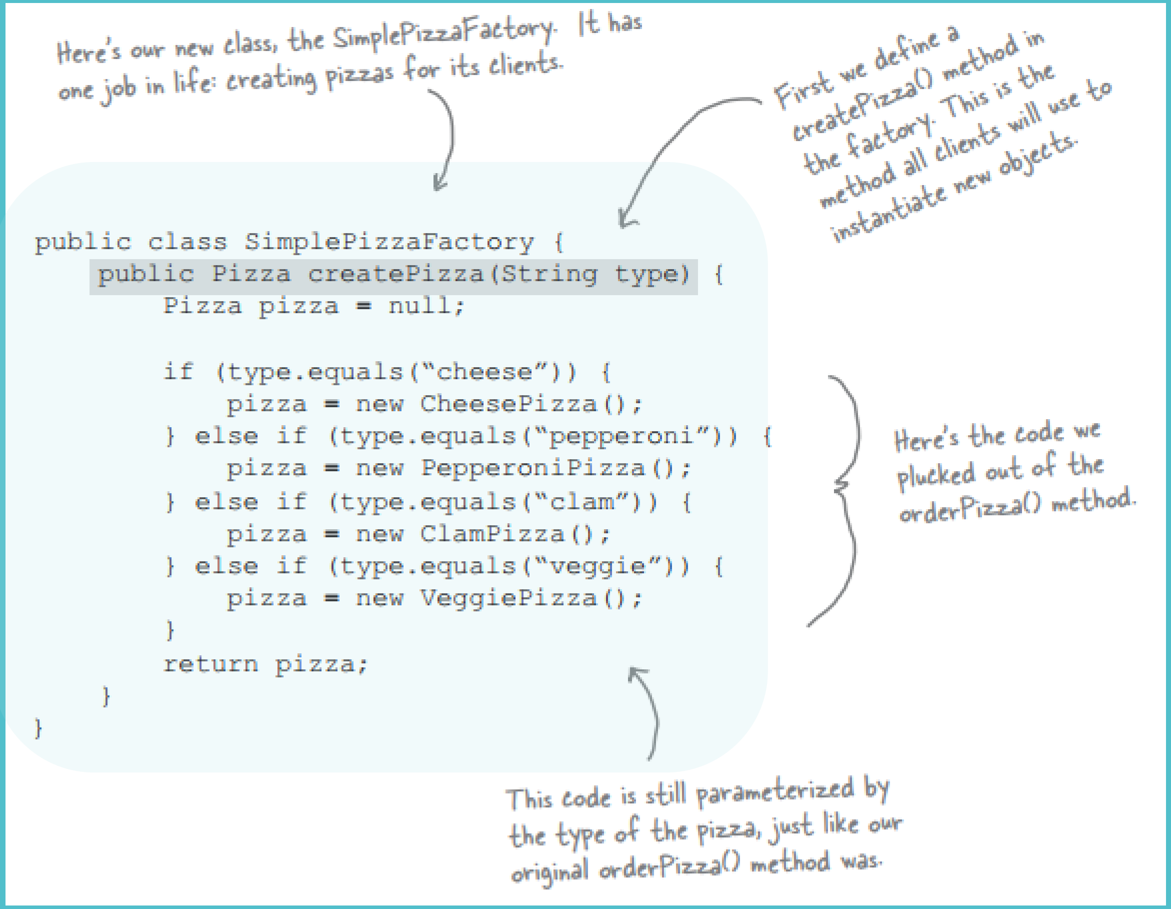
This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years and years. So, we don't expect this code to change, just the pizzas it operates on.



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

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

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

Pull the object creation code out of the orderPizza Method

We need a Factory to make pizza!



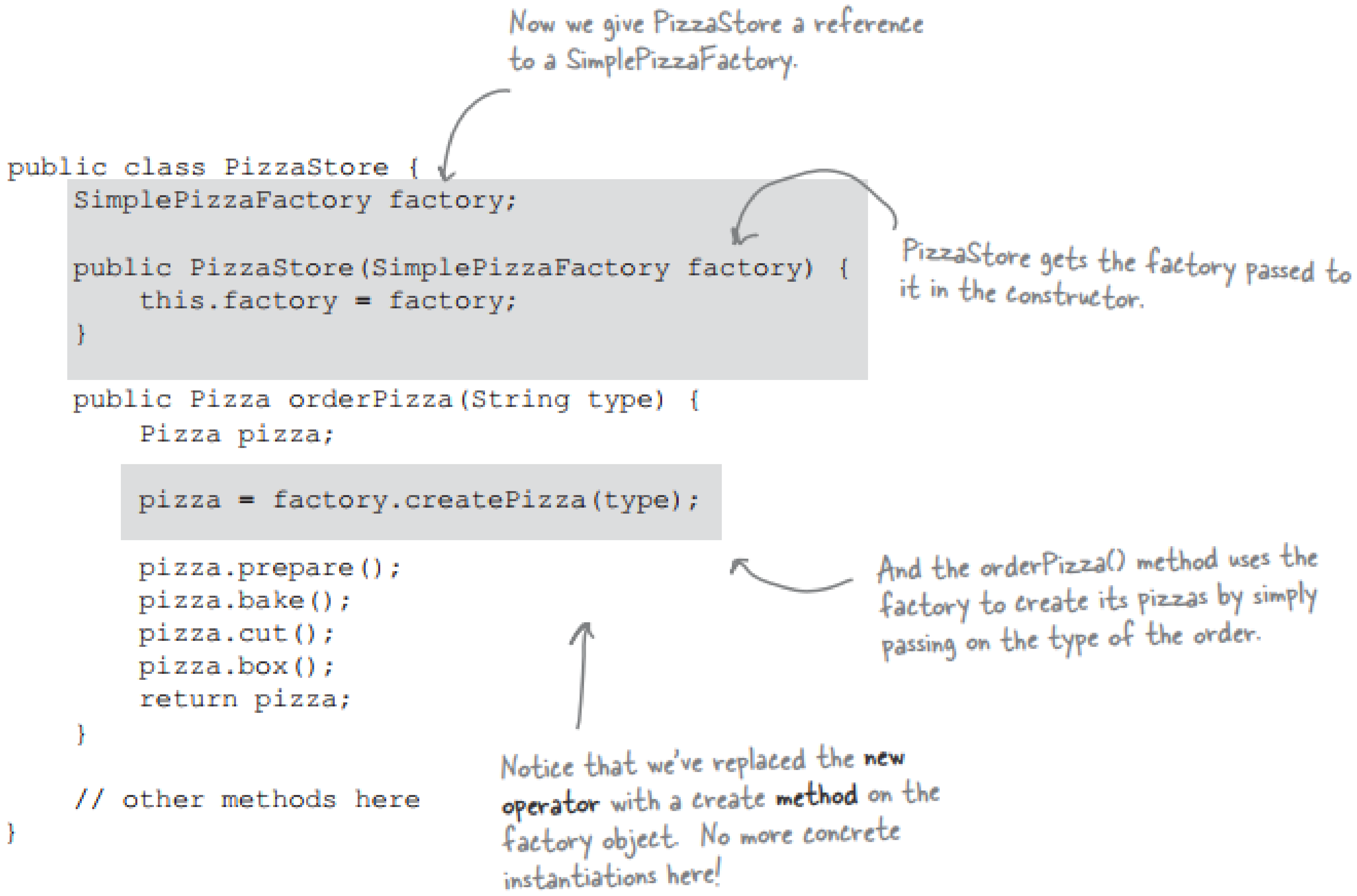
Here's our new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

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.



Now we give PizzaStore a reference to a SimplePizzaFactory.

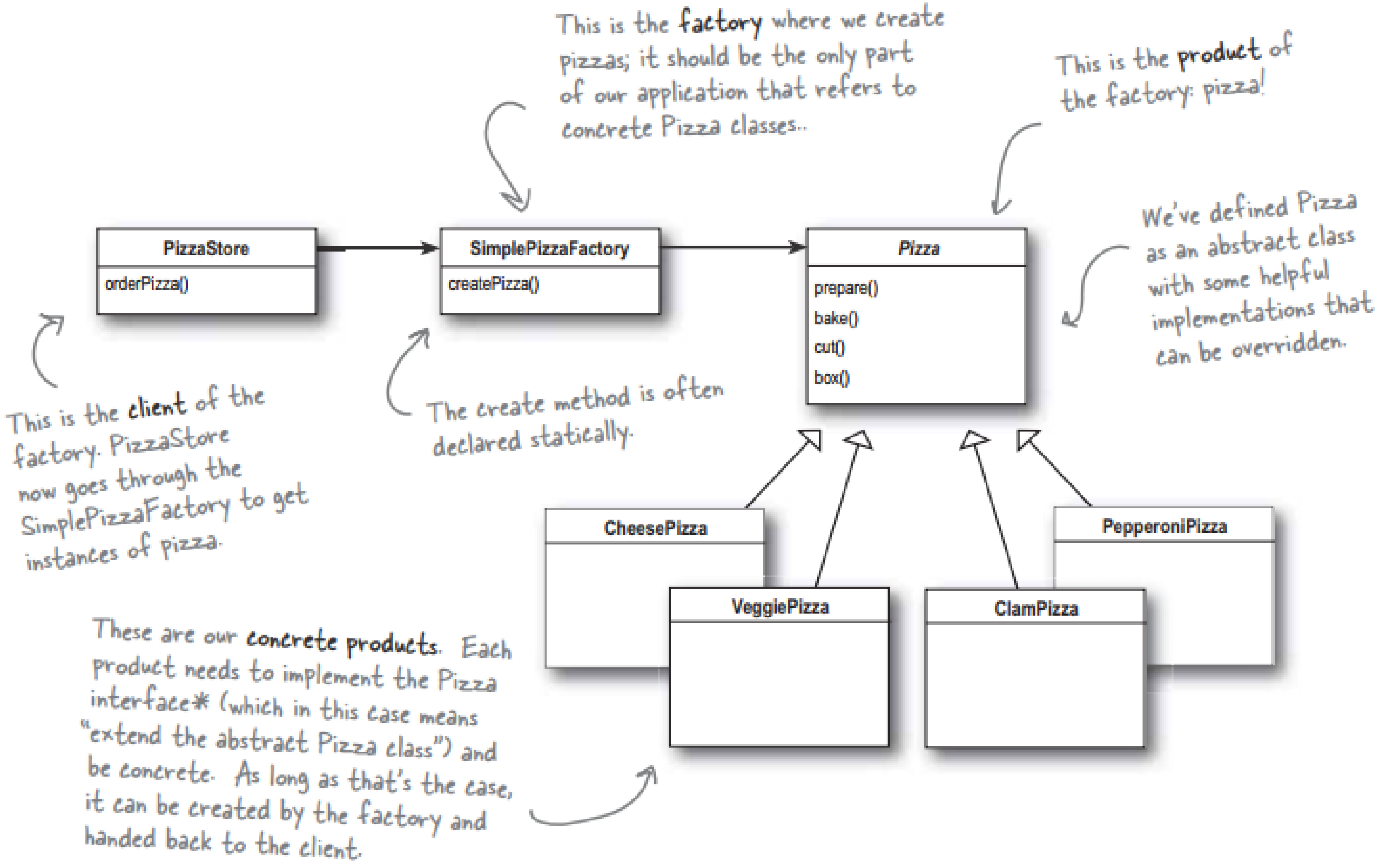
```
public class PizzaStore {
    SimplePizzaFactory factory;

    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!

Use our Factory in class PizzaStore



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.

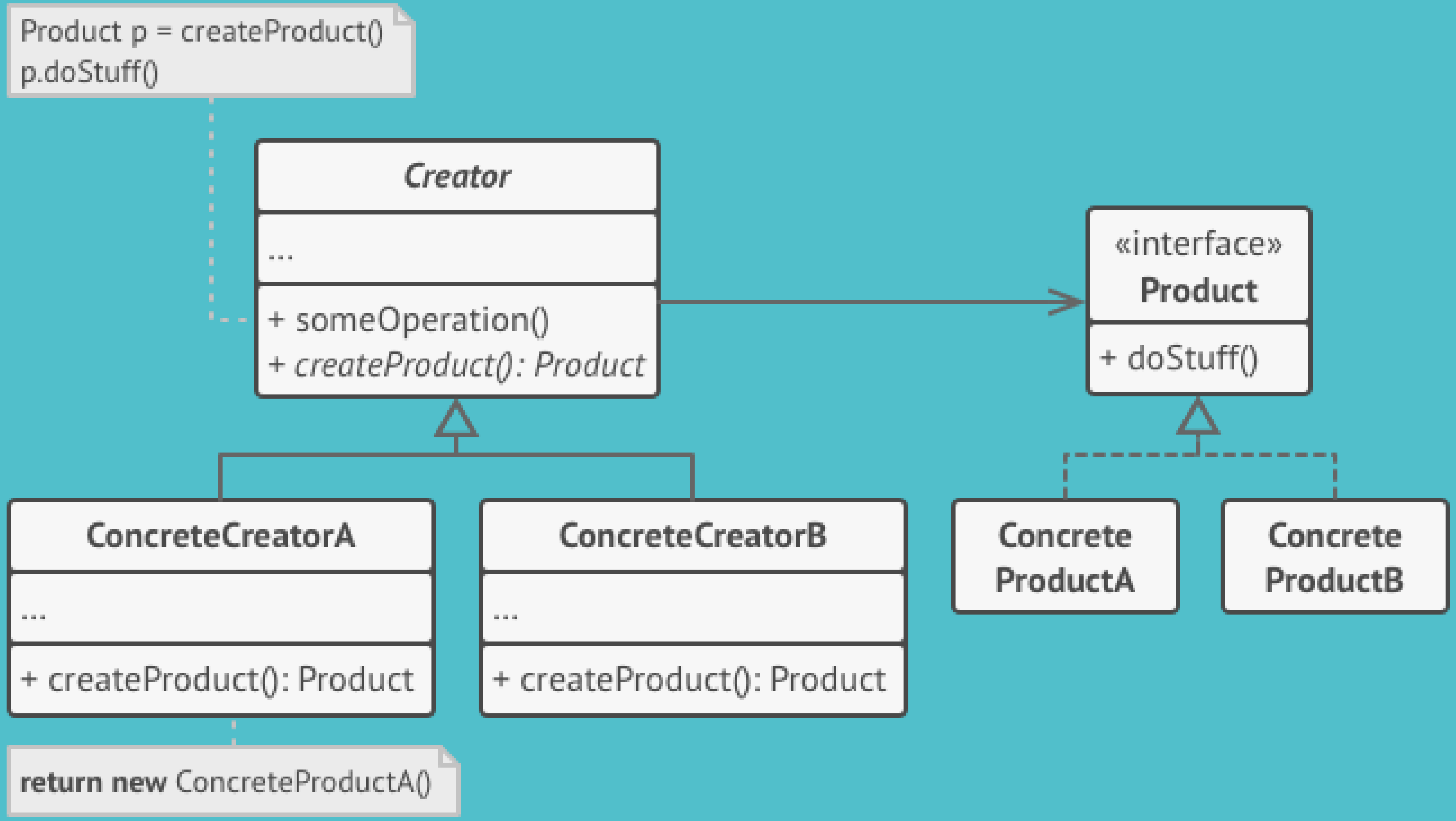
These are our **concrete products**. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

UML Diagram

*. REVIEW FACTORY METHOD

Factory class have such a function like:

```
Abstract Product factoryMethod(String type) //Java

Product* factoryMethod(string type) // C++
```



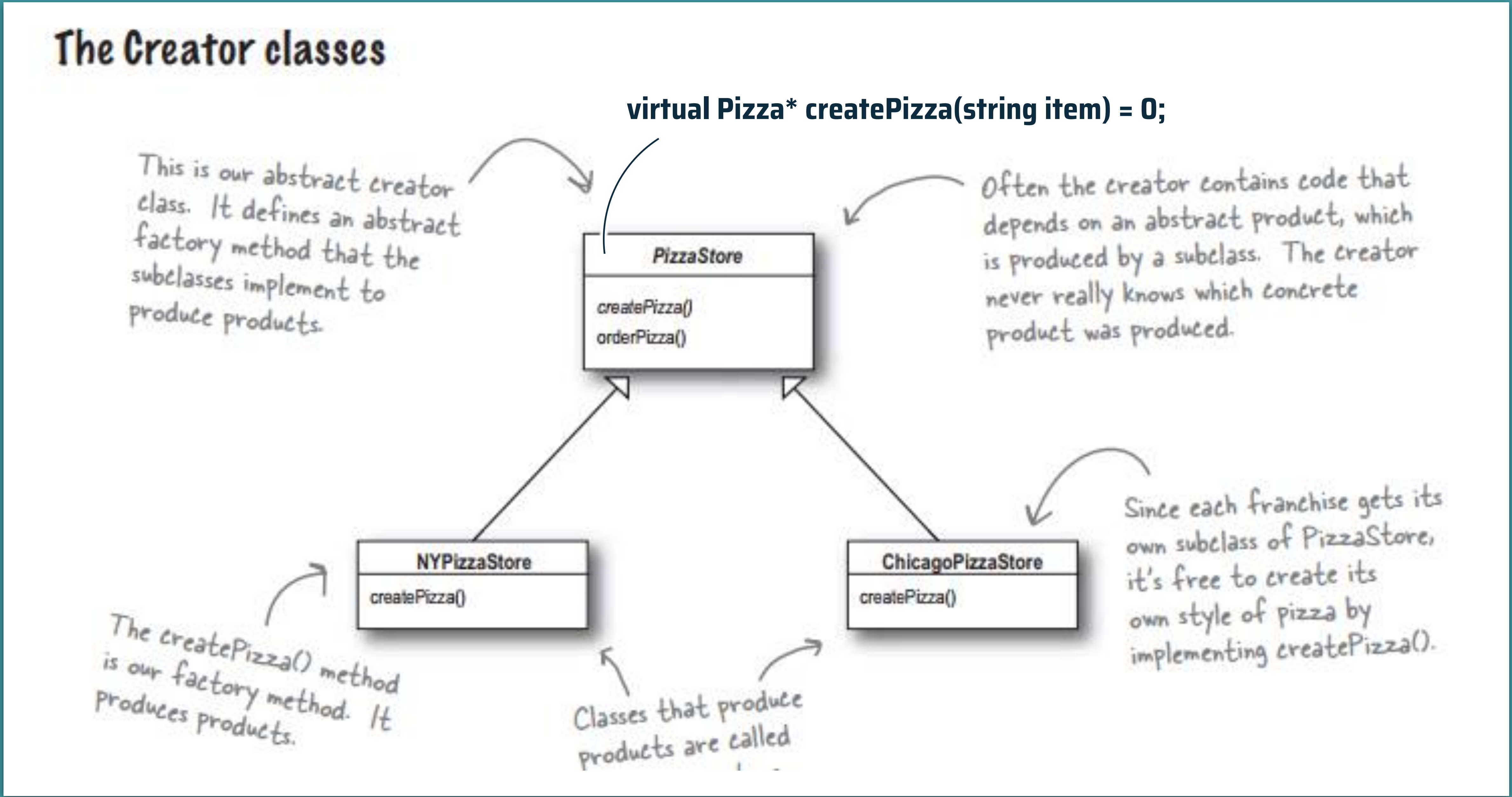
II. FRANCHISE PROBLEM

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

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

The Creator classes

```
virtual Pizza* createPizza(string item) = 0;
```



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.

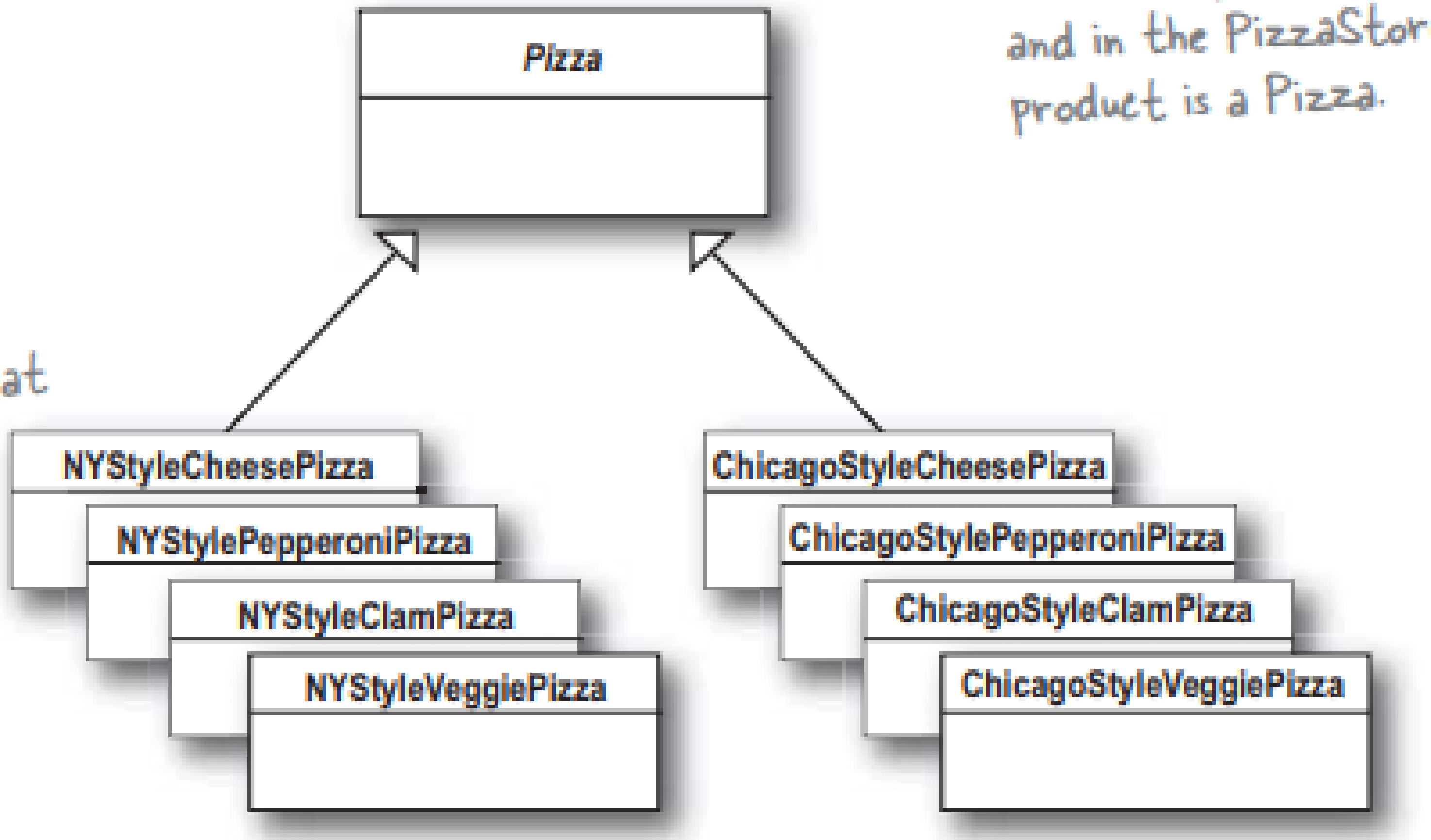
Since each franchise gets its own subclass of PizzaStore, it's free to create its own style of pizza by implementing createPizza().

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

Classes that produce products are called

The Product classes

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



Pizza

NYStyleCheesePizza

NYStylePepperoniPizza

NYStyleClamPizza

NYStyleVeggiePizza

ChicagoStyleCheesePizza

ChicagoStylePepperoniPizza

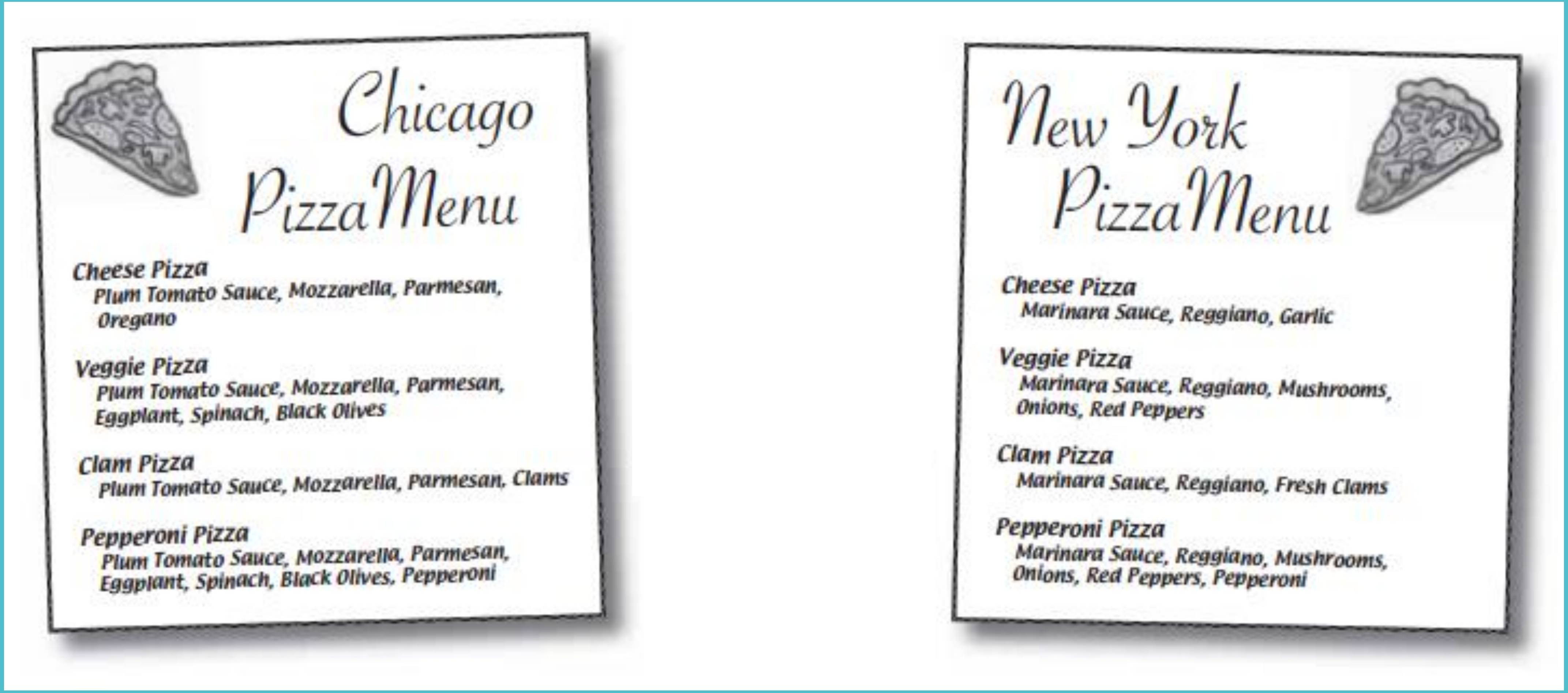
ChicagoStyleClamPizza

ChicagoStyleVeggiePizza

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

II. FRANCHISE PROBLEM

In different regions, they use different ingredients to lower costs and increase the margins. But some franchises use same ingredients for some types of their pizzas.



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

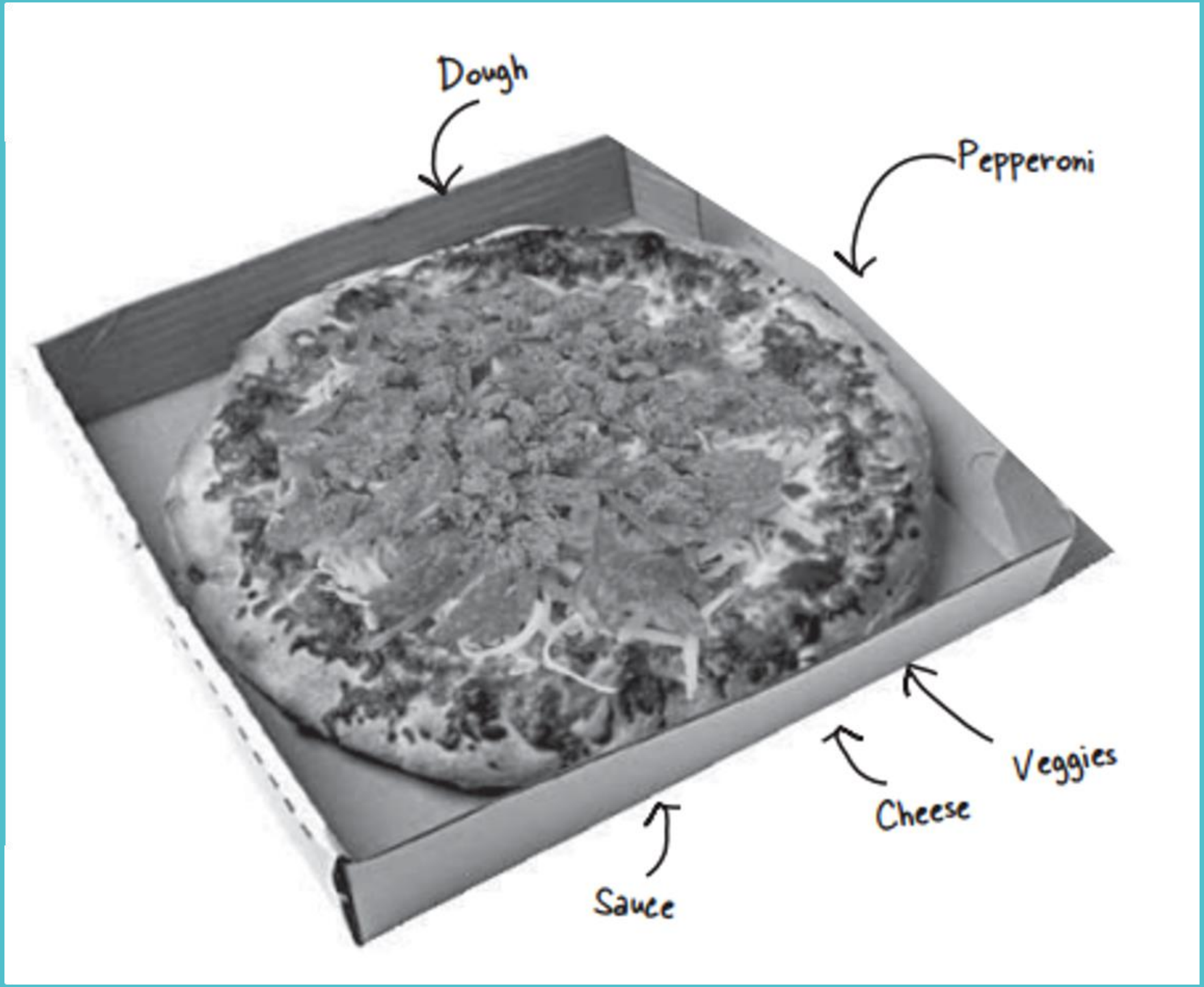
New York
Pizza Menu

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



We can split the pizza into some specific modules.

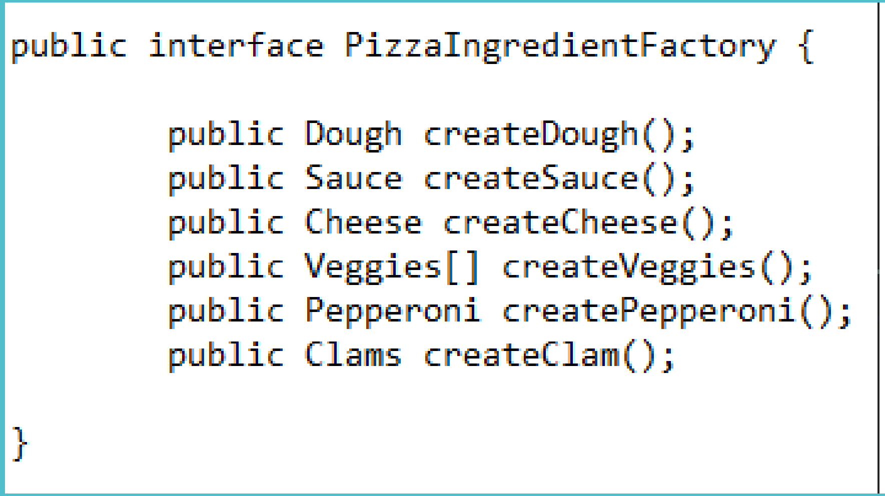
You are going to have to figure out how to handle families of ingredients.

BUILDING THE INGREDIENT FACTORY

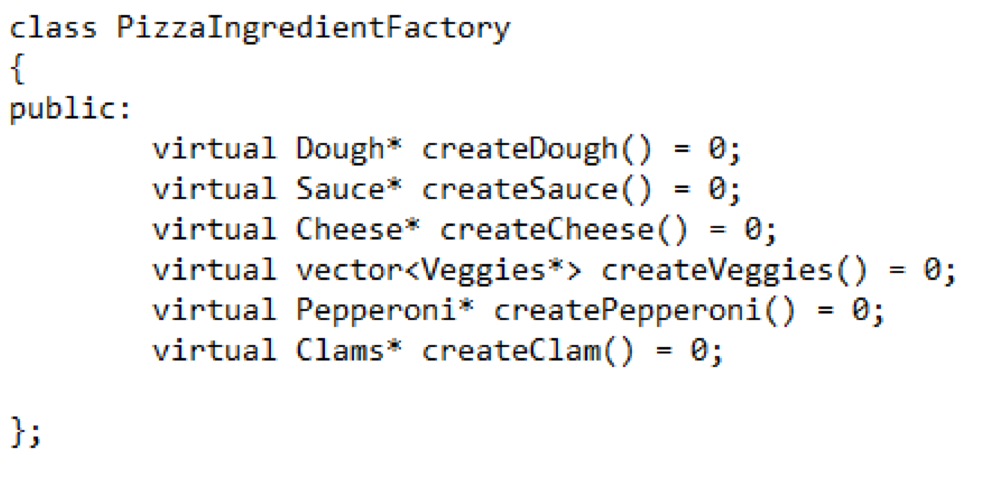
Now you are going to build a factory to create our ingredients; the factory will be responsible for creating each ingredient in the ingredient family. In other words, the factory will need to create dough, sauce, cheese, and so on...

1

Defining an factory that going to create our ingredient.



```java
public interface PizzaIngredientFactory {

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

}
```



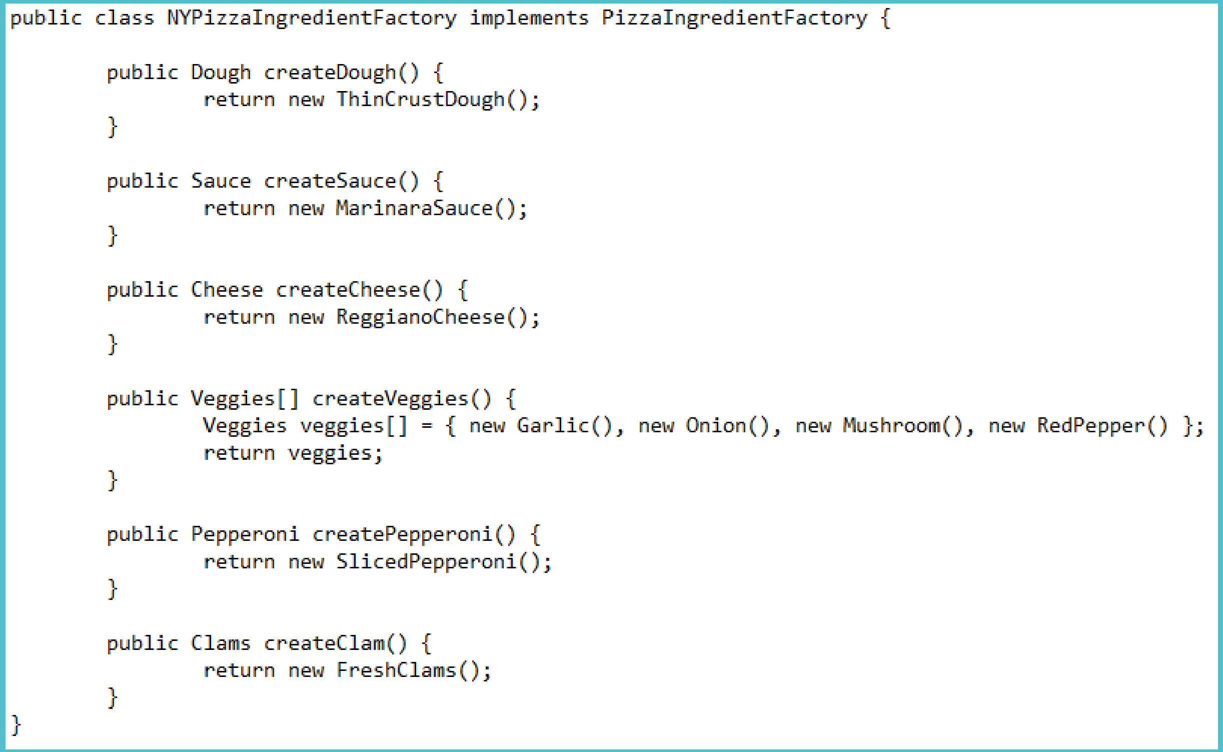
```cpp
class PizzaIngredientFactory
{
public:
        virtual Dough* createDough() = 0;
        virtual Sauce* createSauce() = 0;
        virtual Cheese* createCheese() = 0;
        virtual vector<Veggies*> createVeggies() = 0;
        virtual Pepperoni* createPepperoni() = 0;
        virtual Clams* createClam() = 0;

};
```

JAVA

C++

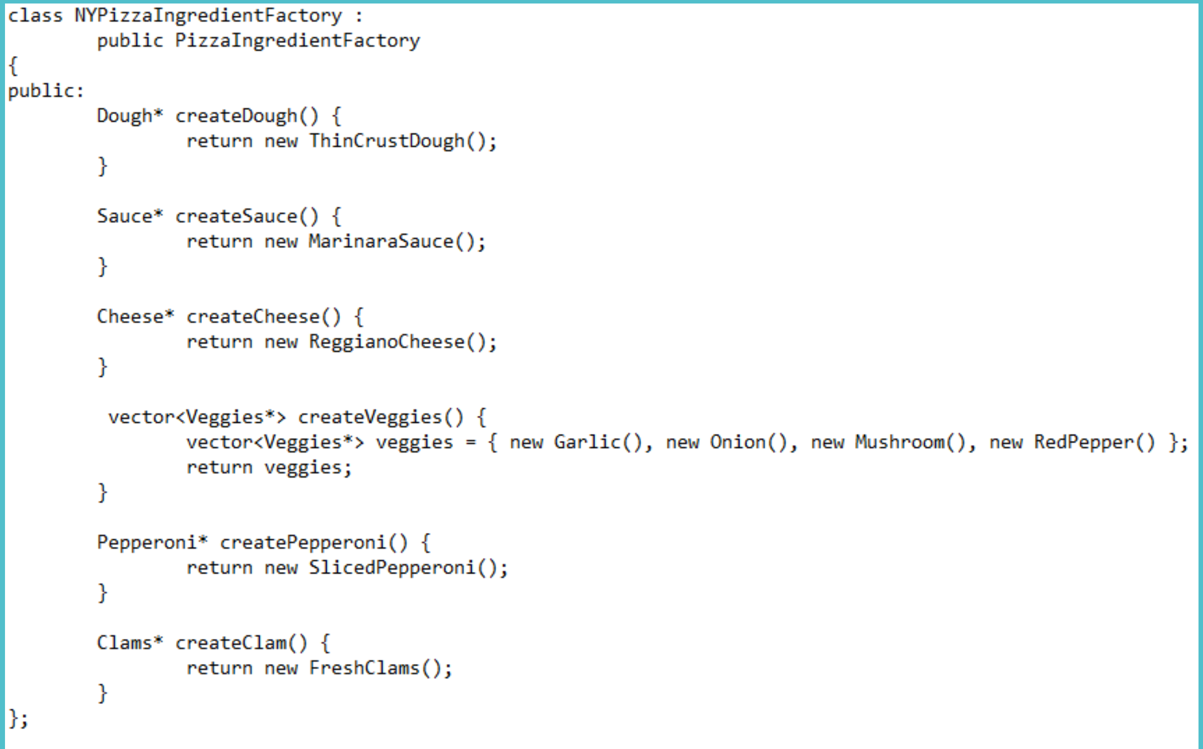
BUILDING THE INGREDIENT FACTORY

2 Build a factory for each region. Create a subclass inherit from our PizzaIngredientFactory



```java
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();
        }
}
```

JAVA

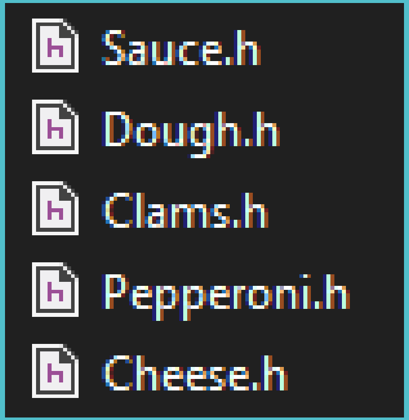


```cpp
class NYPizzaIngredientFactory :
        public PizzaIngredientFactory
{
public:
        Dough* createDough() {
                return new ThinCrustDough();
        }

        Sauce* createSauce() {
                return new MarinaraSauce();
        }

        Cheese* createCheese() {
                return new ReggianoCheese();
        }

         vector<Veggies*> createVeggies() {
                vector<Veggies*> veggies = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
                return veggies;
        }

        Pepperoni* createPepperoni() {
                return new SlicedPepperoni();
        }

        Clams* createClam() {
                return new FreshClams();
        }
};
```

C++

BUILDING THE INGREDIENT FACTORY

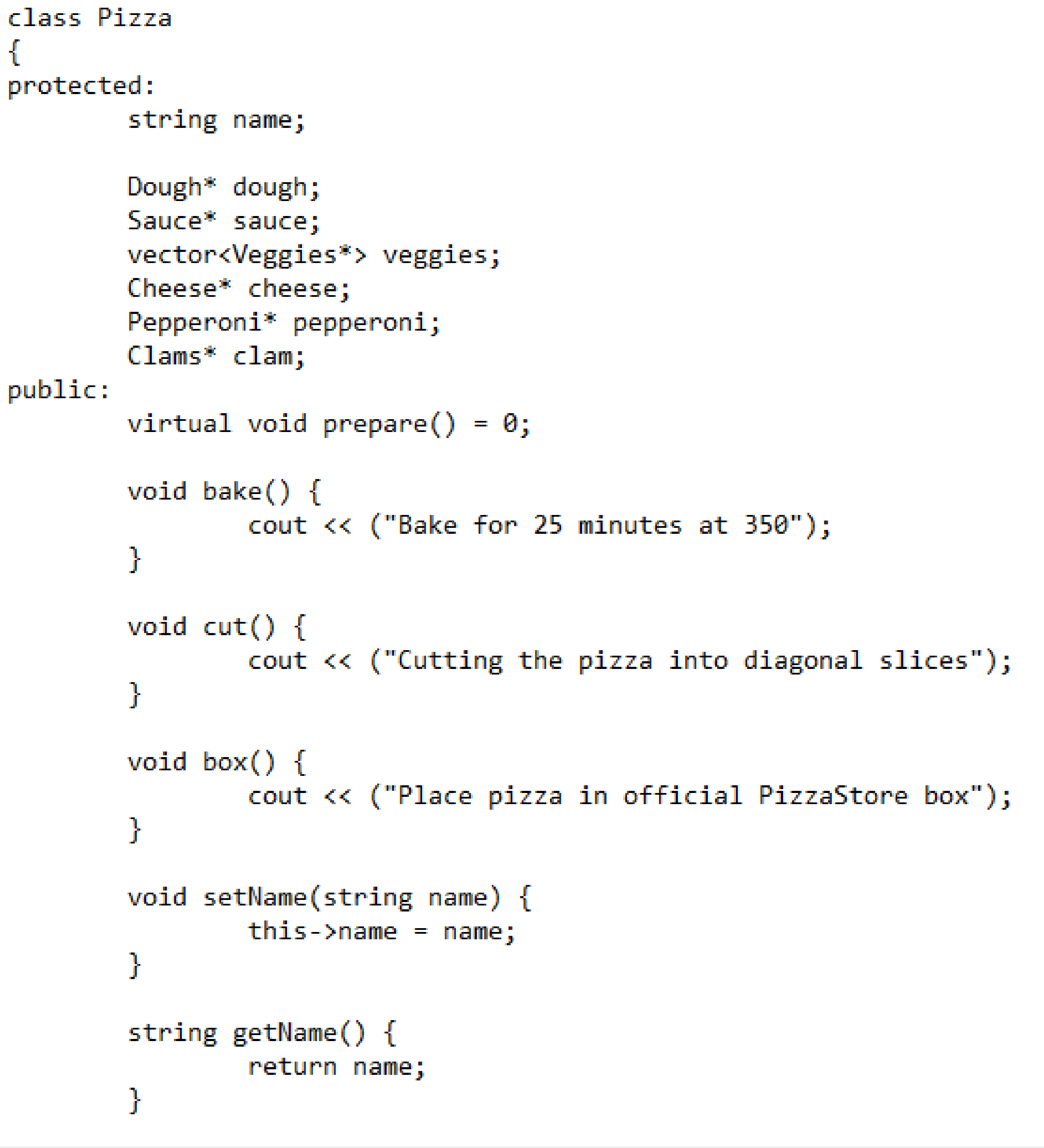
3 Build concrete class of our Ingredients.



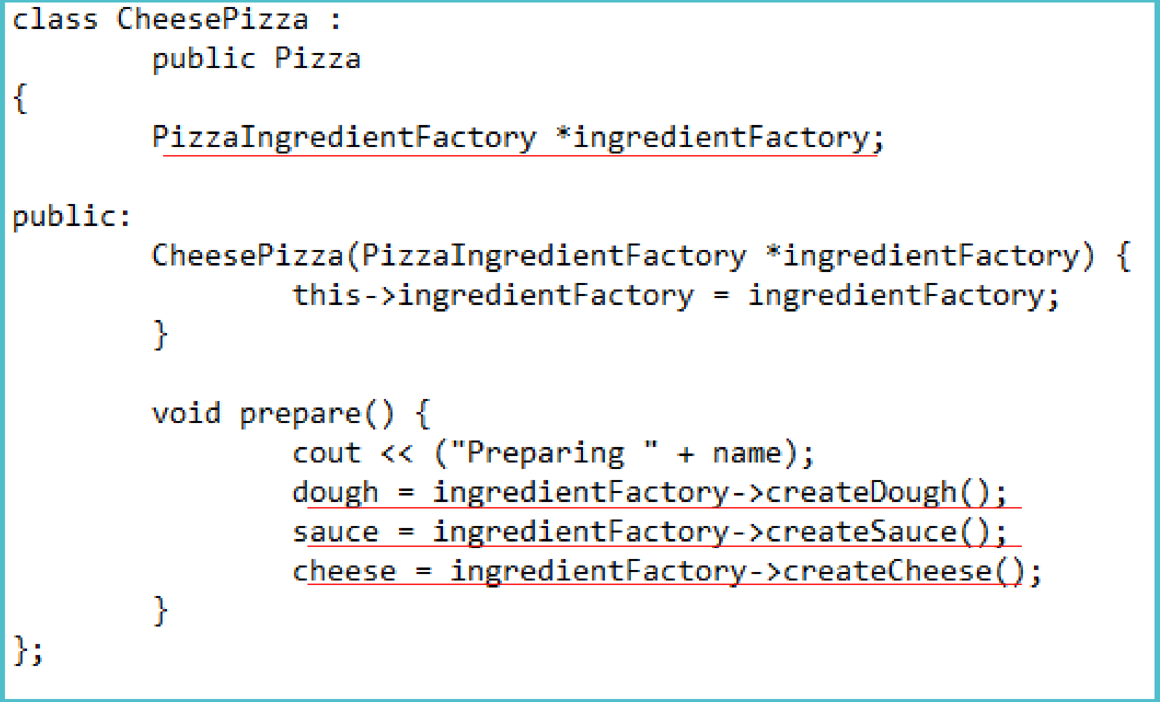
Sauce.h
Dough.h
Clams.h
Pepperoni.h
Cheese.h

4 Using it in our PizzaStore.

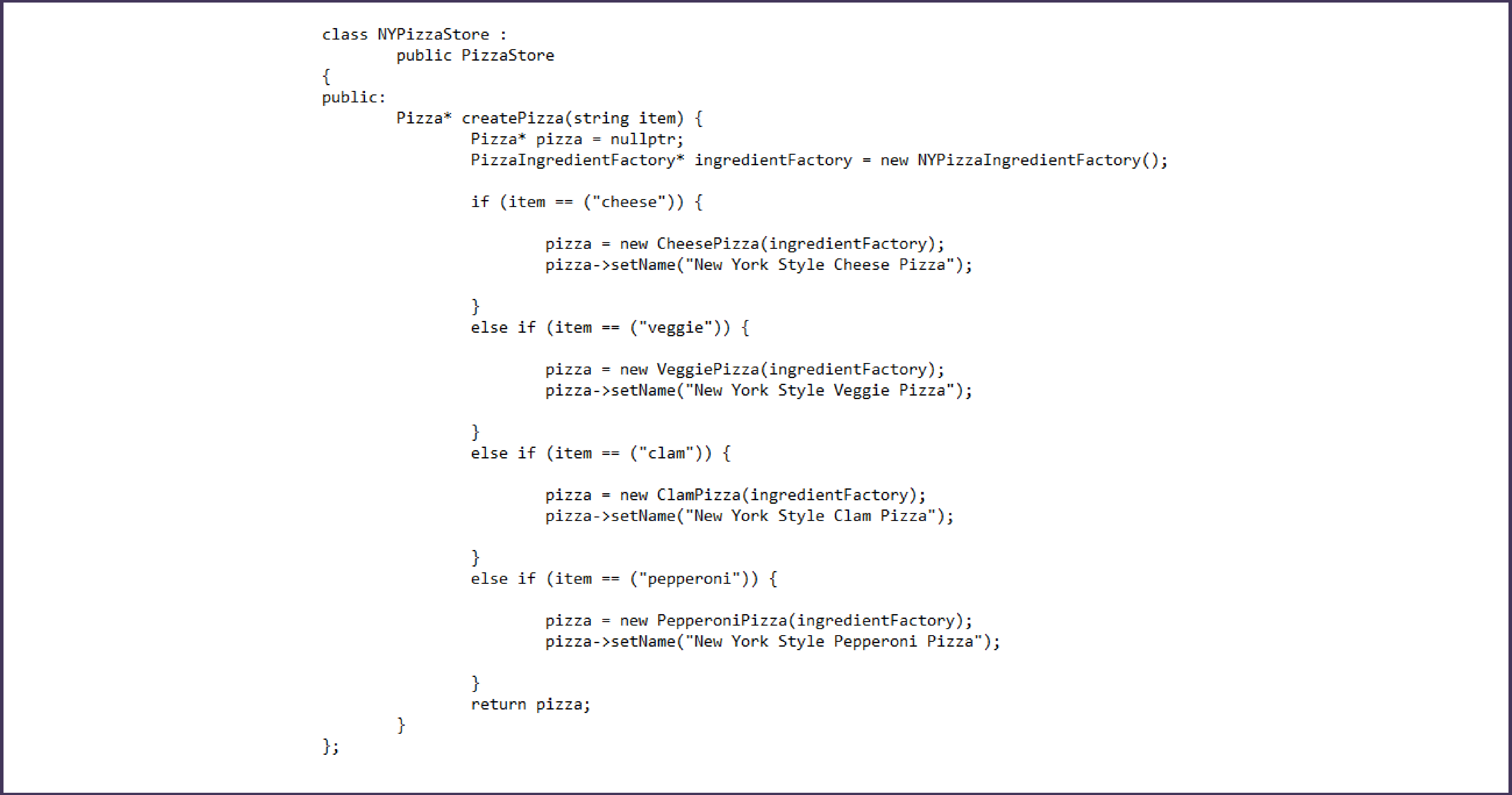
BUILD DA PIZZA CLASS AND SUBCLASSES



```
class Pizza
{
protected:
        string name;

        Dough* dough;
        Sauce* sauce;
        vector<Veggies*> veggies;
        Cheese* cheese;
        Pepperoni* pepperoni;
        Clams* clam;
public:
        virtual void prepare() = 0;

        void bake() {
                cout << ("Bake for 25 minutes at 350");
        }

        void cut() {
                cout << ("Cutting the pizza into diagonal slices");
        }

        void box() {
                cout << ("Place pizza in official PizzaStore box");
        }

        void setName(string name) {
                this->name = name;
        }

        string getName() {
                return name;
        }
```

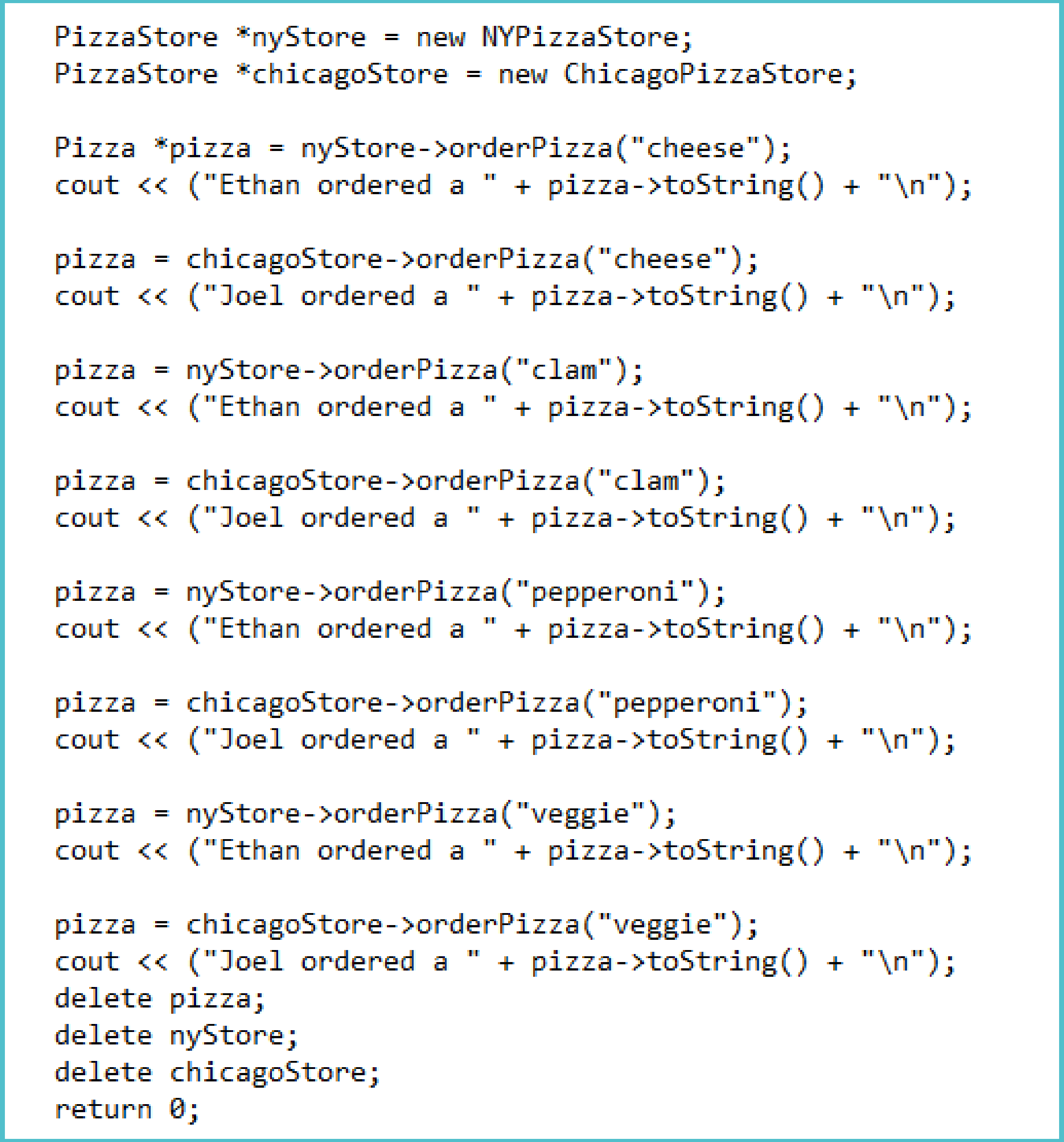


```
class CheesePizza :
        public Pizza
{
        PizzaIngredientFactory *ingredientFactory;

public:
        CheesePizza(PizzaIngredientFactory *ingredientFactory) {
                this->ingredientFactory = ingredientFactory;
        }

        void prepare() {
                cout << ("Preparing " + name);
                dough = ingredientFactory->createDough();
                sauce = ingredientFactory->createSauce();
                cheese = ingredientFactory->createCheese();
        }
};
```



```
class NYPizzaStore :
        public PizzaStore
{
public:
        Pizza* createPizza(string item) {
                Pizza* pizza = nullptr;
                PizzaIngredientFactory* ingredientFactory = new NYPizzaIngredientFactory();

                if (item == ("cheese")) {

                        pizza = new CheesePizza(ingredientFactory);
                        pizza->setName("New York Style Cheese Pizza");

                }
                else if (item == ("veggie")) {

                        pizza = new VeggiePizza(ingredientFactory);
                        pizza->setName("New York Style Veggie Pizza");

                }
                else if (item == ("clam")) {

                        pizza = new ClamPizza(ingredientFactory);
                        pizza->setName("New York Style Clam Pizza");

                }
                else if (item == ("pepperoni")) {

                        pizza = new PepperoniPizza(ingredientFactory);
                        pizza->setName("New York Style Pepperoni Pizza");

                }
                return pizza;
        }
};
```

NYPizzaStore – C++



```
PizzaStore *nyStore = new NYPizzaStore;
PizzaStore *chicagoStore = new ChicagoPizzaStore;

Pizza *pizza = nyStore->orderPizza("cheese");
cout << ("Ethan ordered a " + pizza->toString() + "\n");

pizza = chicagoStore->orderPizza("cheese");
cout << ("Joel ordered a " + pizza->toString() + "\n");

pizza = nyStore->orderPizza("clam");
cout << ("Ethan ordered a " + pizza->toString() + "\n");

pizza = chicagoStore->orderPizza("clam");
cout << ("Joel ordered a " + pizza->toString() + "\n");

pizza = nyStore->orderPizza("pepperoni");
cout << ("Ethan ordered a " + pizza->toString() + "\n");

pizza = chicagoStore->orderPizza("pepperoni");
cout << ("Joel ordered a " + pizza->toString() + "\n");

pizza = nyStore->orderPizza("veggie");
cout << ("Ethan ordered a " + pizza->toString() + "\n");

pizza = chicagoStore->orderPizza("veggie");
cout << ("Joel ordered a " + pizza->toString() + "\n");
delete pizza;
delete nyStore;
delete chicagoStore;
return 0;
```

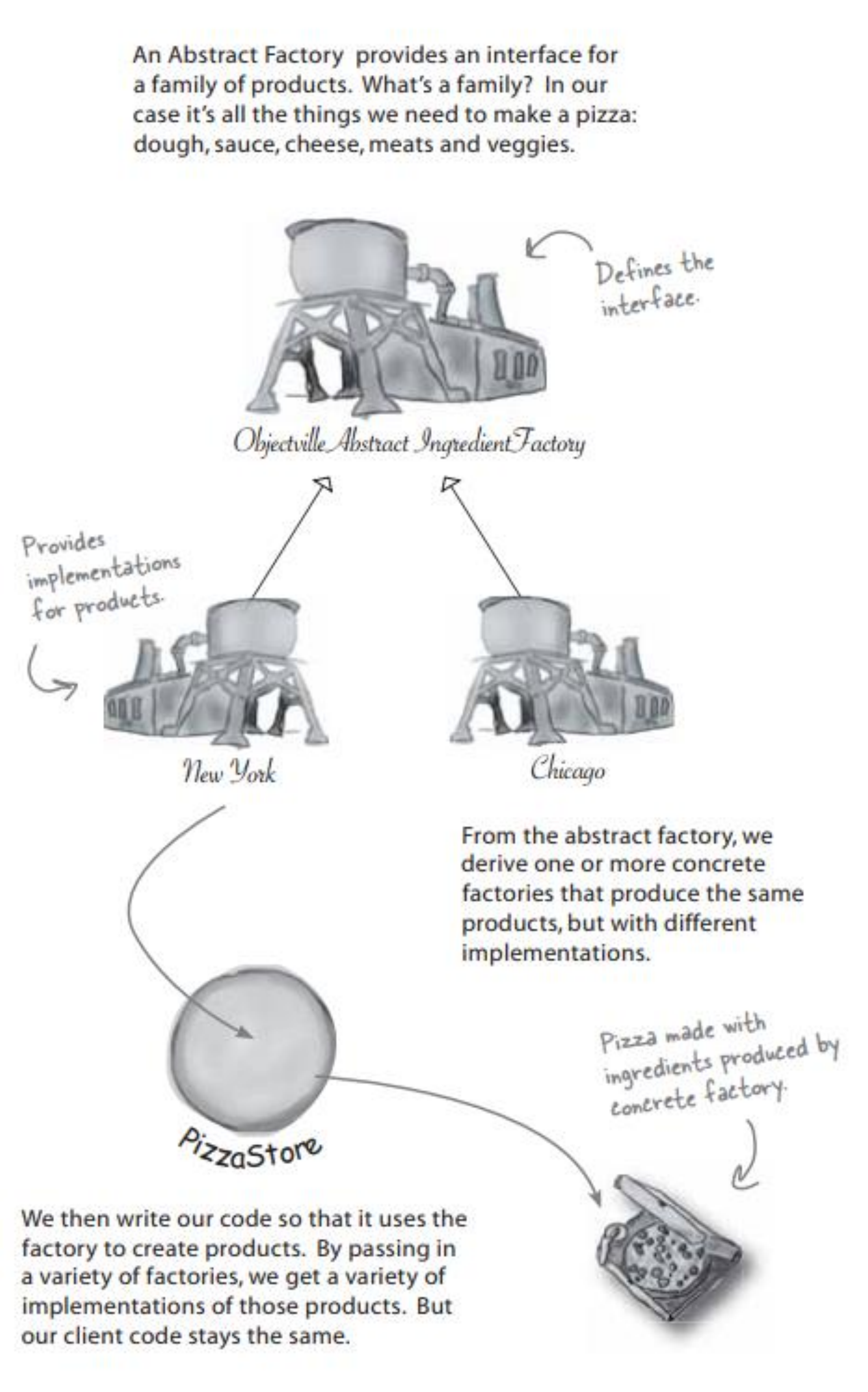
PizzaTestDrive.cpp

WHAT WE HAVE DONE?

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

An Abstract Factory gives us an interface for creating a family of products. By writing code that uses this interface, we decouple our code from the actual factory that creates the products. That allow us to implement a variety of factories that produce product meant for different contexts – such as different regions, different operating system, or different look and feels.

Because our code is decouple from the actual products, we can substitute different factories to get different behaviors.



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.

Defines the interface.

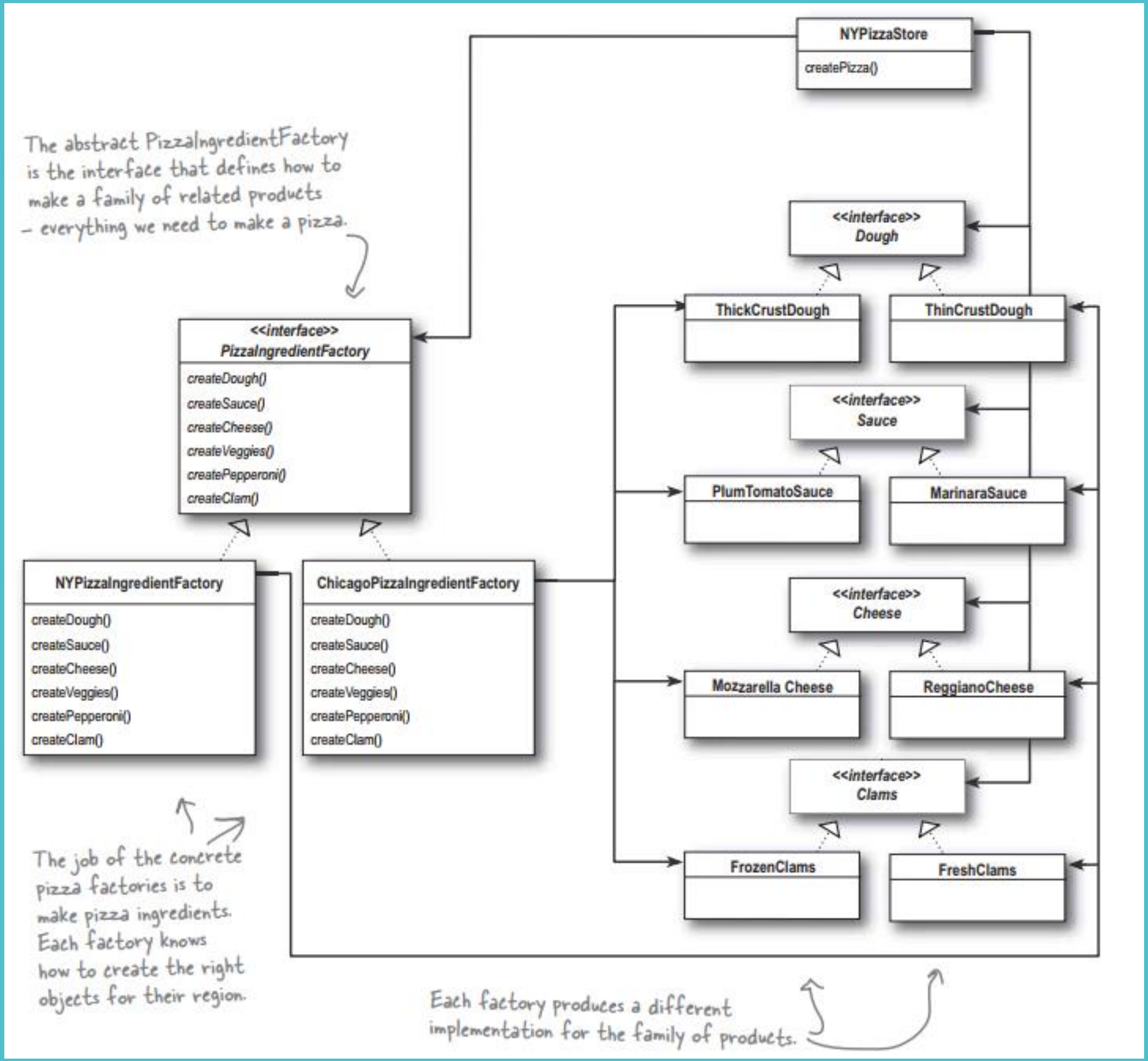
Provides implementations for products.

From the abstract factory, we derive one or more concrete factories that produce the same products, but with different implementations.

Pizza made with ingredients produced by concrete factory.

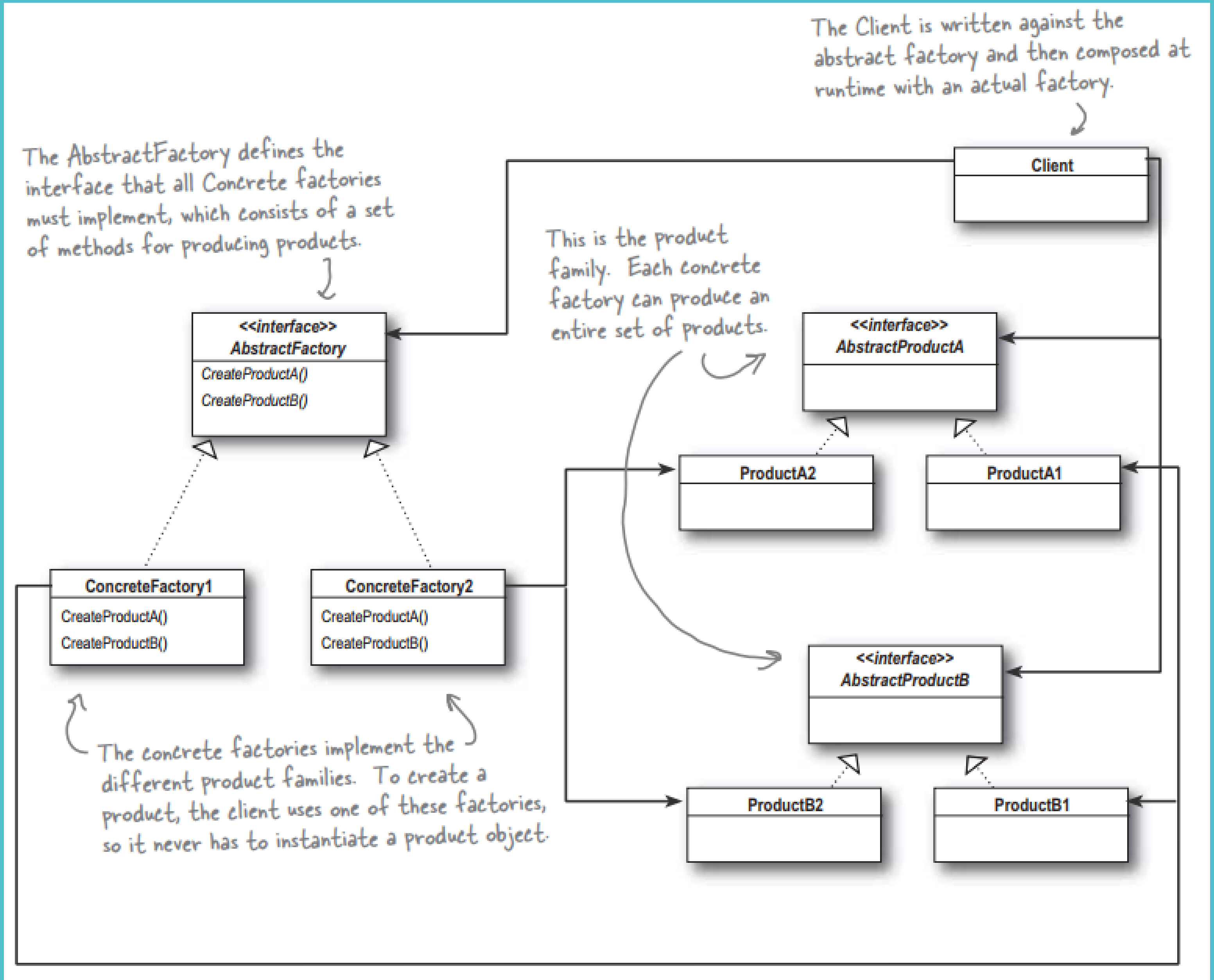
We then write our code so that it uses the factory to create products. By passing in a variety of factories, we get a variety of implementations of those products. But our client code stays the same.

CLASS DIAGRAM



THE ABSTRACT FACTORY PATTERN

Providing 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.

III. THE ABSTRACT FACTORY PATTERN

1. **Core Principle: “The Dependency Inversion Principle”:**

Depend upon abstractions. Do not depend upon concrete classes.

2. **Use Abstract Factory when:**

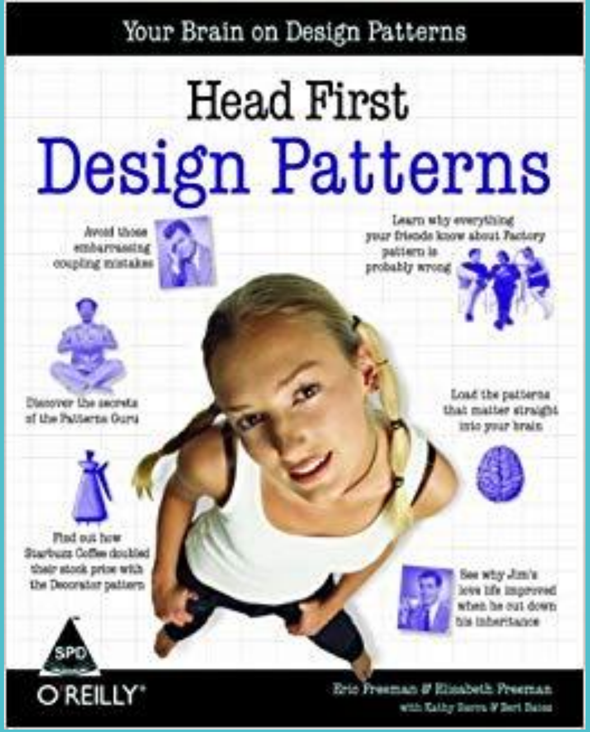
A system should be independent of how its products are created, composed, and represented.
A system should be configured with one of multiple families of products.
A family of related product objects is designed to be used together, and you need to enforce this constraint.
You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

3. Implement Abstract Factory:

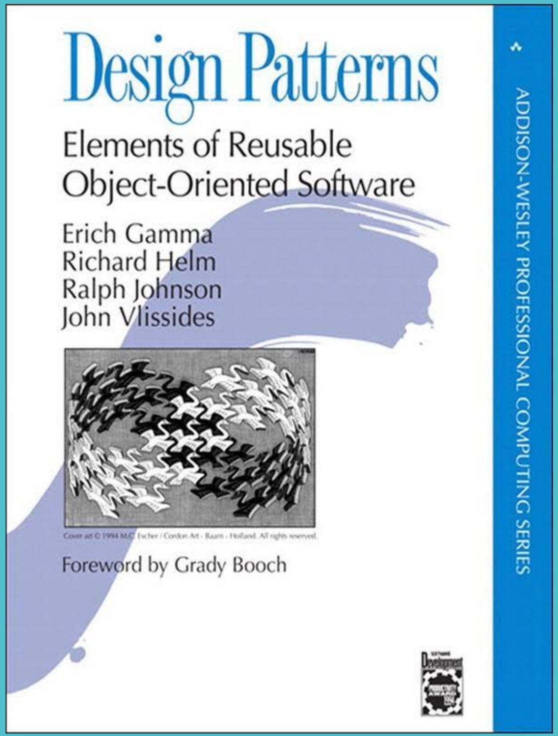
Abstract Factory classes are often implemented with factory methods Factory Method, but they can also be implemented using Prototype.
A concrete factory is often a Singleton.

IV. REFERENCES

Head First Design Patterns - By Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra


Your Brain on Design Patterns
Head First
Design Patterns
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
SPD
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

Design Patterns: Elements of Reusable Object-Oriented Software - By Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm



THANK YOU

Cảm ơn các bạn thân yêu đã lắng nghe.