The Factory Method Pattern
and
The Abstract Factory Pattern

Several slides in this presentation are taken from:

- https://www.cs.colorado.edu/~kena/classes/5448/f09/lectures/18-decoratorfactory.pdf

Baking with OO Goodness

The Problem with New?

- Example 1

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

- Is Duck an interface or an implementation?
- Is DecoyDuck an interface or an implementation?
- Which design principle is violated here?

The Problem with New?

- Example 2: Assume that you have code that checks a specific variable, and creates different class types accordingly

-
  ```
  Duck duck;
  if (picnic) {
        duck = new MallardDuck();
  } else if (hunting) {
        duck = new DecoyDuck();
  } else if (inBathTub) {
        duck = new RubberDuck();
  }
  ```
- What if you decide to add a new variable?
- Which design/OO principles are being violated here?

Pizza Store

- We have a pizza store program that wants to separate the process of creating a pizza with the process of preparing/ordering a pizza.
- Let’s say you have a pizza shop, 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 Store

- But … You need more than one type of 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;
```

Creation

Preparation

Each Pizza subtype (CheesePizza, VeggiePizza, etc.) knows how to prepare itself

Pizza Store

- How about you encapsulate what varies?
- A simple way to encapsulate this code is to put it in a separate class.
- That new class depends on the concrete classes, but those dependencies no longer impact the preparation code.
- Let us take a look.

Pizza Store

- First we pull the pizza creation code out of 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;
```

Creation

Preparation

Each Pizza subtype (CheesePizza, VeggiePizza, etc.) knows how to prepare itself

Pizza Store

- 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
-

```
if (type.equals(“cheese”)) {
    pizza = new CheesePizza();
} else if (type.equals(“greek”)) {
    pizza = new GreekPizza();
} else if (type.equals(“pepperoni”) {
    pizza = new PepperoniPizza();
}
```

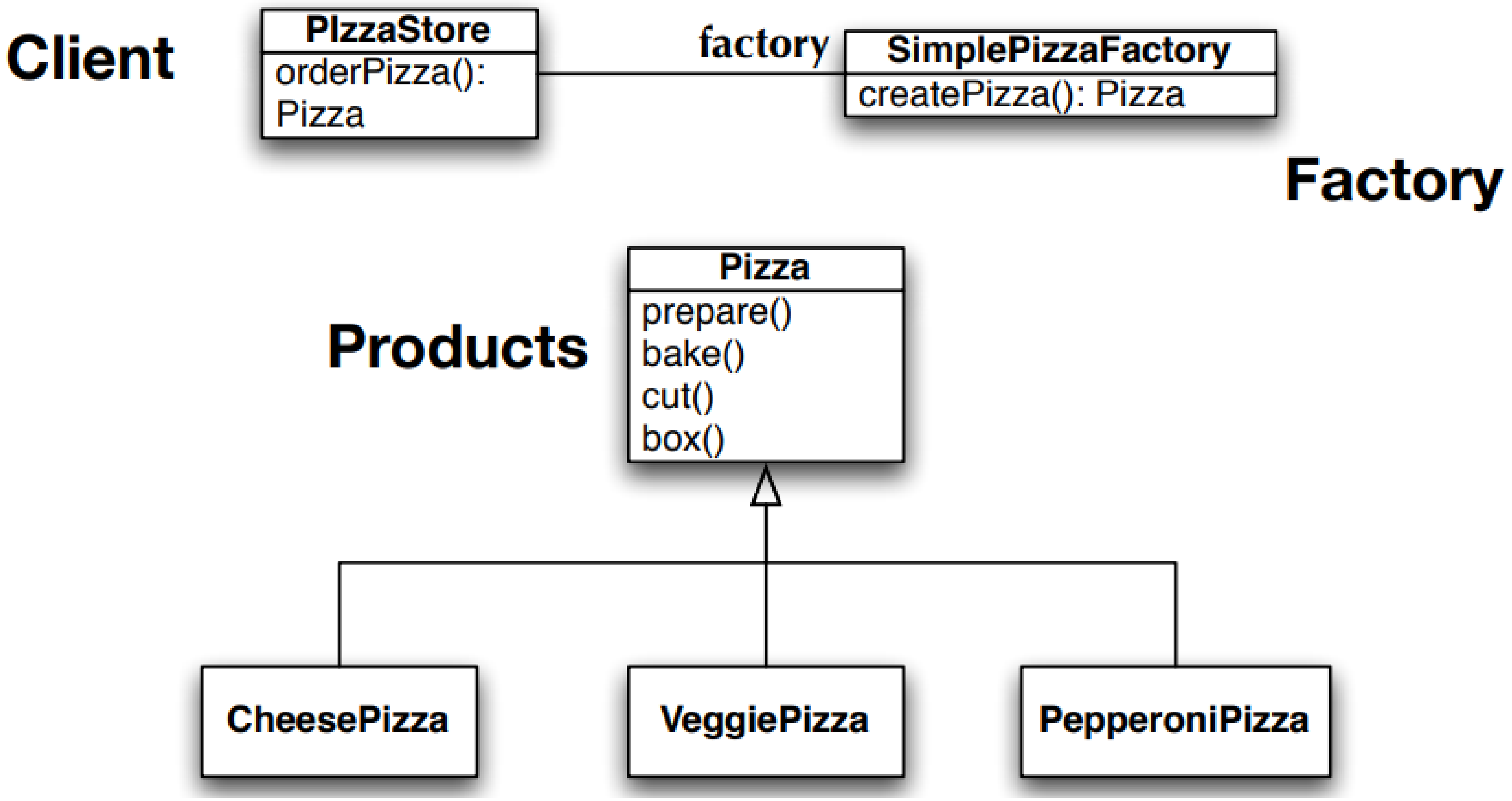
Pizza Store

```
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;
```

Pizza Store

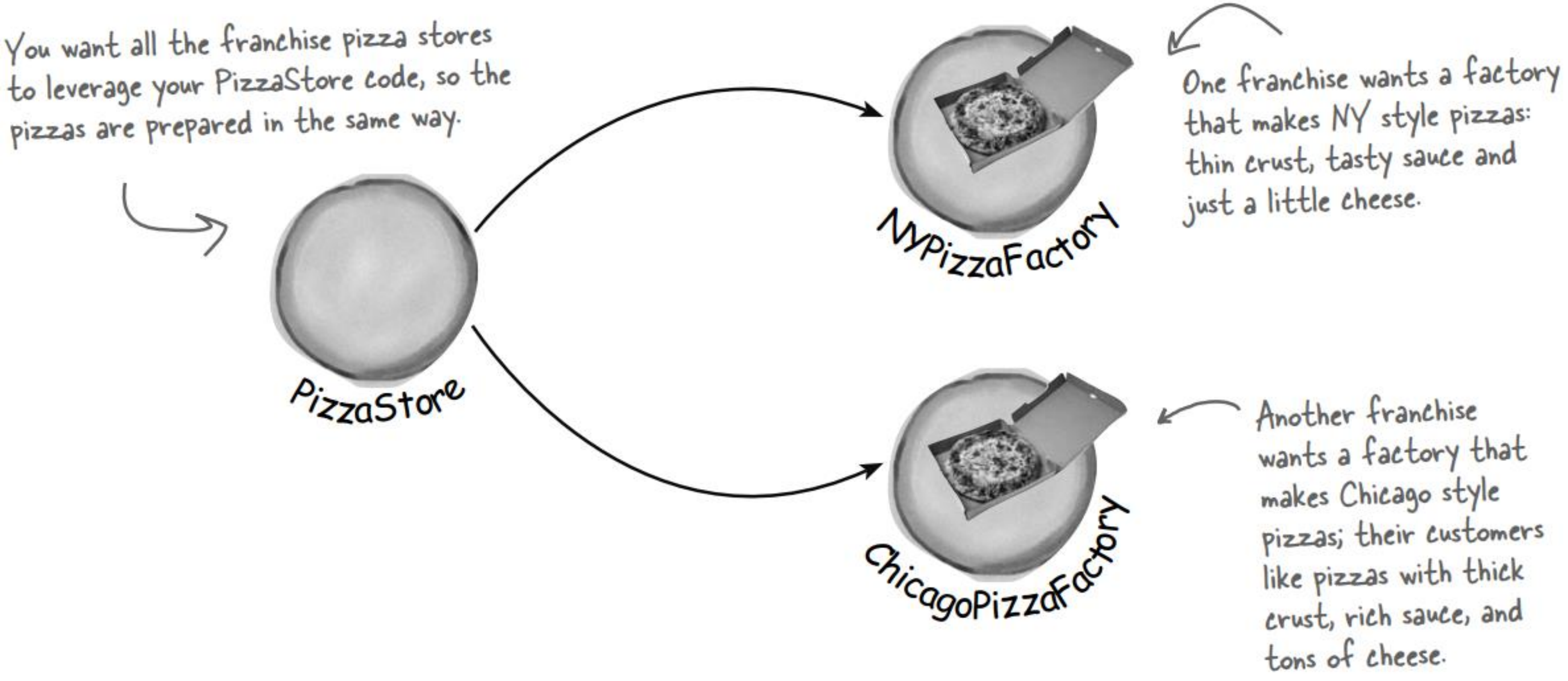
```
public class PizzaStore {
    SimplePizzaFactory factory;
    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory; }
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();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

The Simple Factory



- Problems with the simple factory?
- Factory method to the rescue!

More Pizza Stores ….Chicago! California!


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.

More Pizza Stores ….Chicago! California!

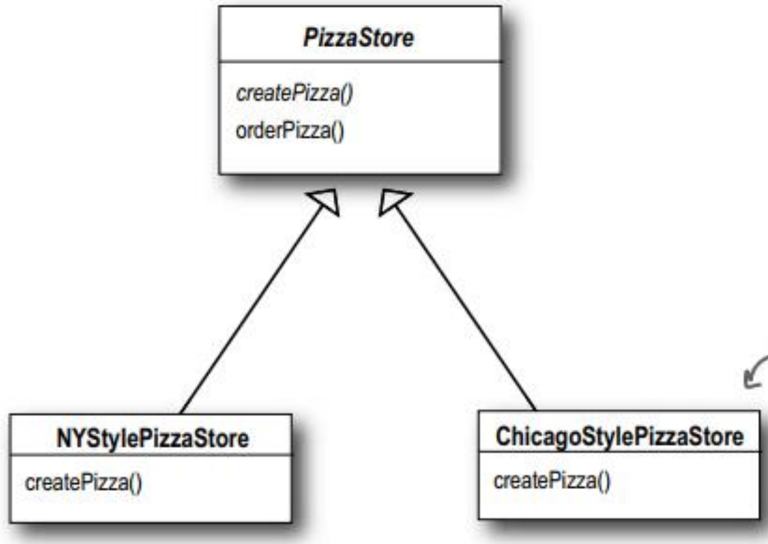
- The pizza store example evolves
  - to include the notion of different franchises that exist in different parts of the country (California, New York, Chicago)
- Each franchise will need its own factory to create pizzas that match the proclivities of the locals
  - However, we want to retain the preparation process of the Pizza Store
- The Factory Method Design Pattern allows you to do this by
  - placing abstract, “code to an interface” code in a superclass
  - placing object creation code in a subclass

More Pizza Stores ….Chicago! California!

```
public abstract class PizzaStore {
    protected abstract createPizza(String type);
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
         }
    }
```

More Pizza Stores ….Chicago! California!

```
public class NYPizzaStore extends PizzaStore {
    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new NYCheesePizza();
        } else if (type.equals("greek")) {
            return new NYGreekPizza();
        } else if (type.equals("pepperoni")) {
            return new NYPepperoniPizza();
        }
        return null;
    }
}
```



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.

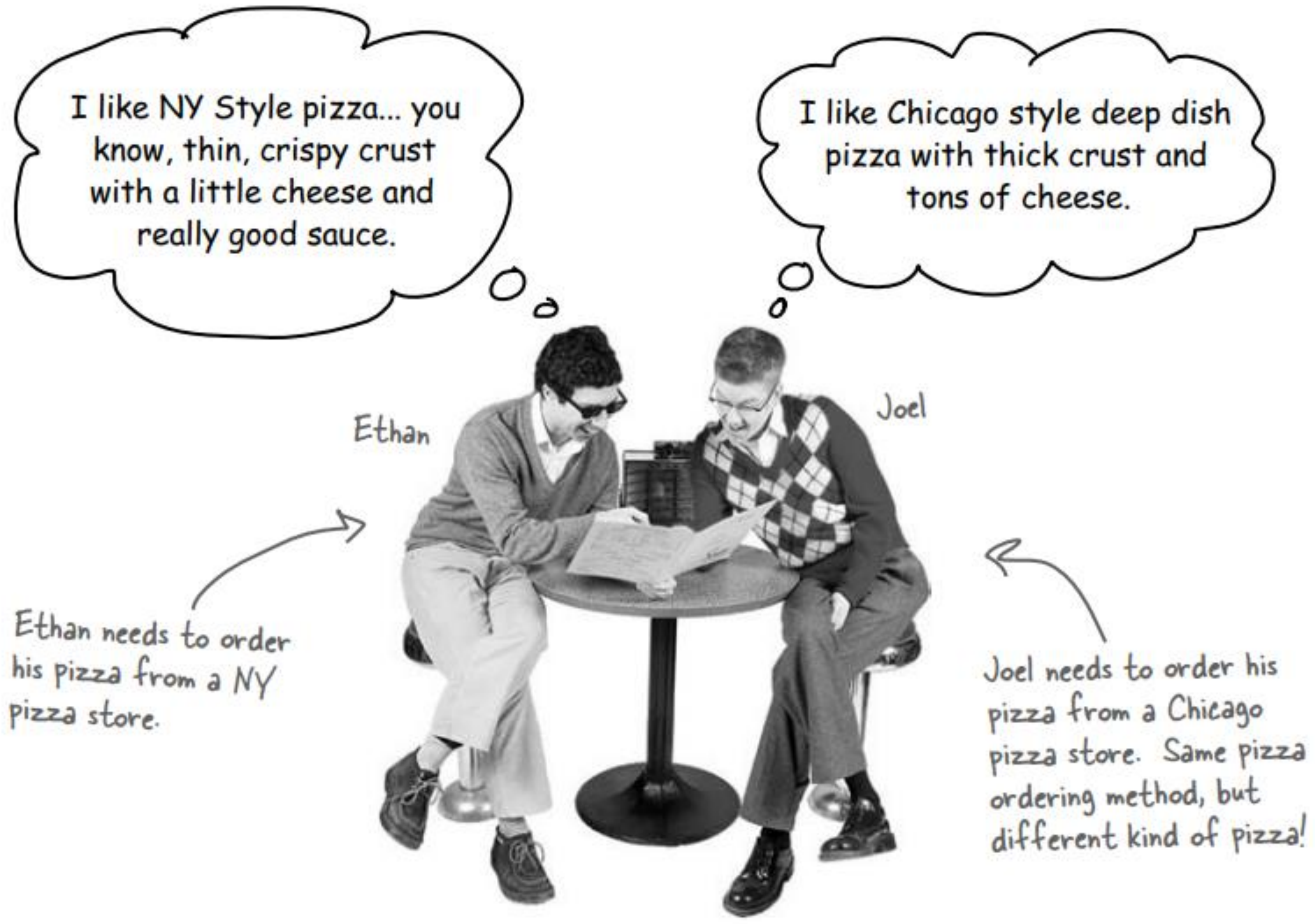
Remember: createPizza() is abstract in PizzaStore, so all pizza store subtypes MUST implement the method.

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();
    }
}
```

Let's see how it works: ordering pizzas with the pizza factory method


I like NY Style pizza... you know, thin, crispy crust with a little cheese and really good sauce.
I like Chicago style deep dish pizza with thick crust and tons of cheese.
Ethan
Joel
Ethan needs to order his pizza from a NY pizza store.
Joel needs to order his pizza from a Chicago pizza store. Same pizza ordering method, but different kind of pizza!

More Pizza Stores ….Chicago! California!

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:

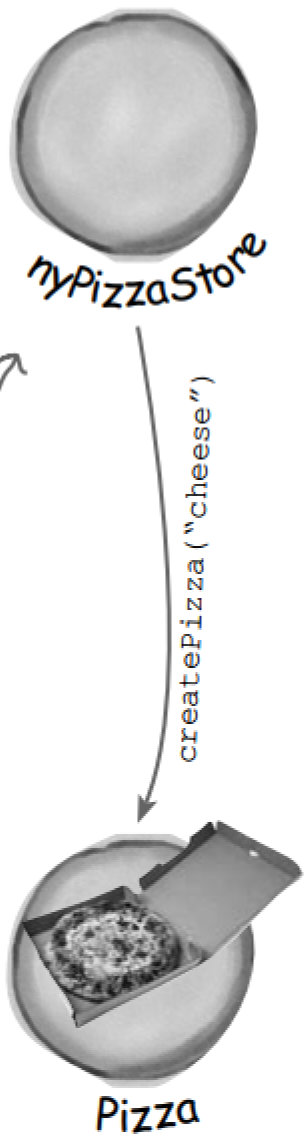
```
nyPizzaStore.orderPizza(“cheese”);
```

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.



More Pizza Stores ….Chicago! California!

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.

Anything Missing from that Design?

```
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”);
    }

    void box() {
        System.out.println(“Place pizza in official PizzaStore box”);
    }

    public String getName() {
        return name;
    }
}
```

```
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”);
    }
}
```

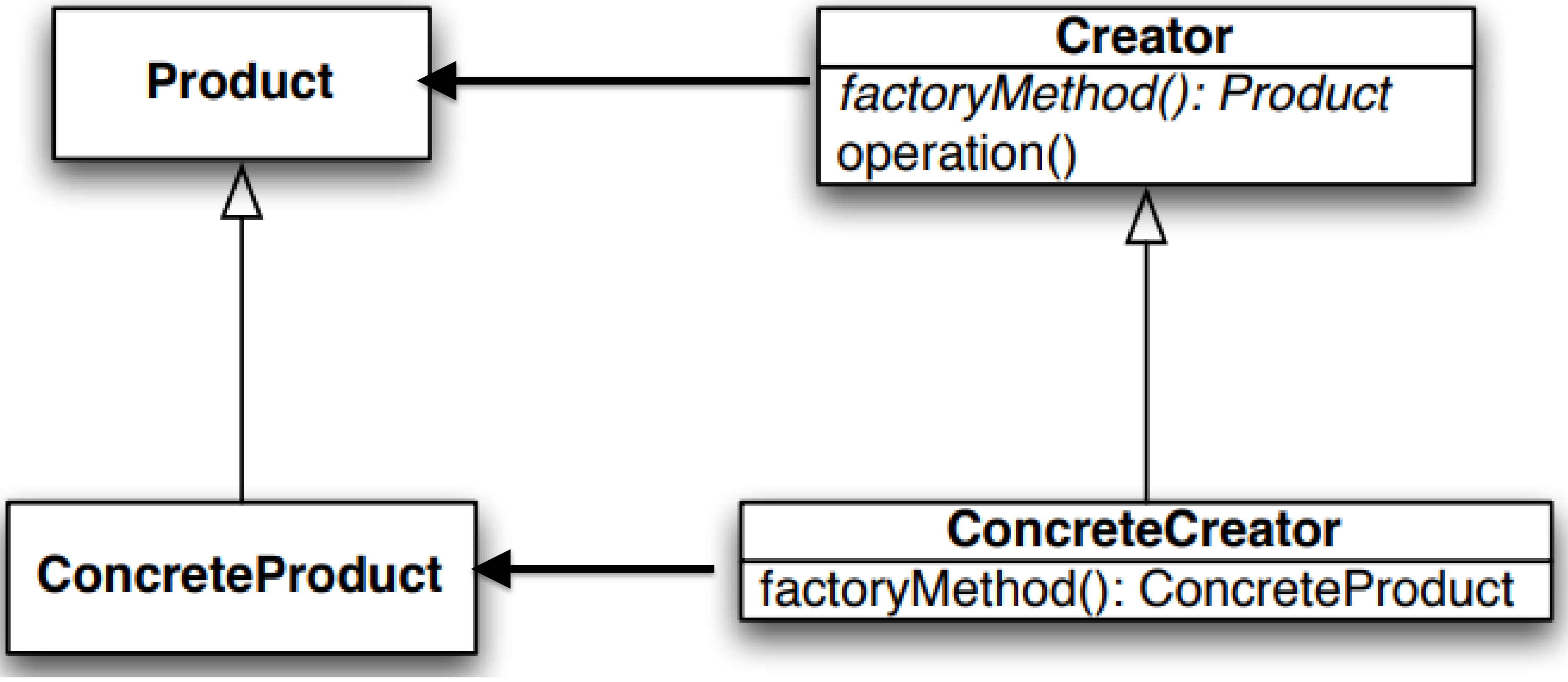
Lets make different concrete classes now

```
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”);
    }
}
```

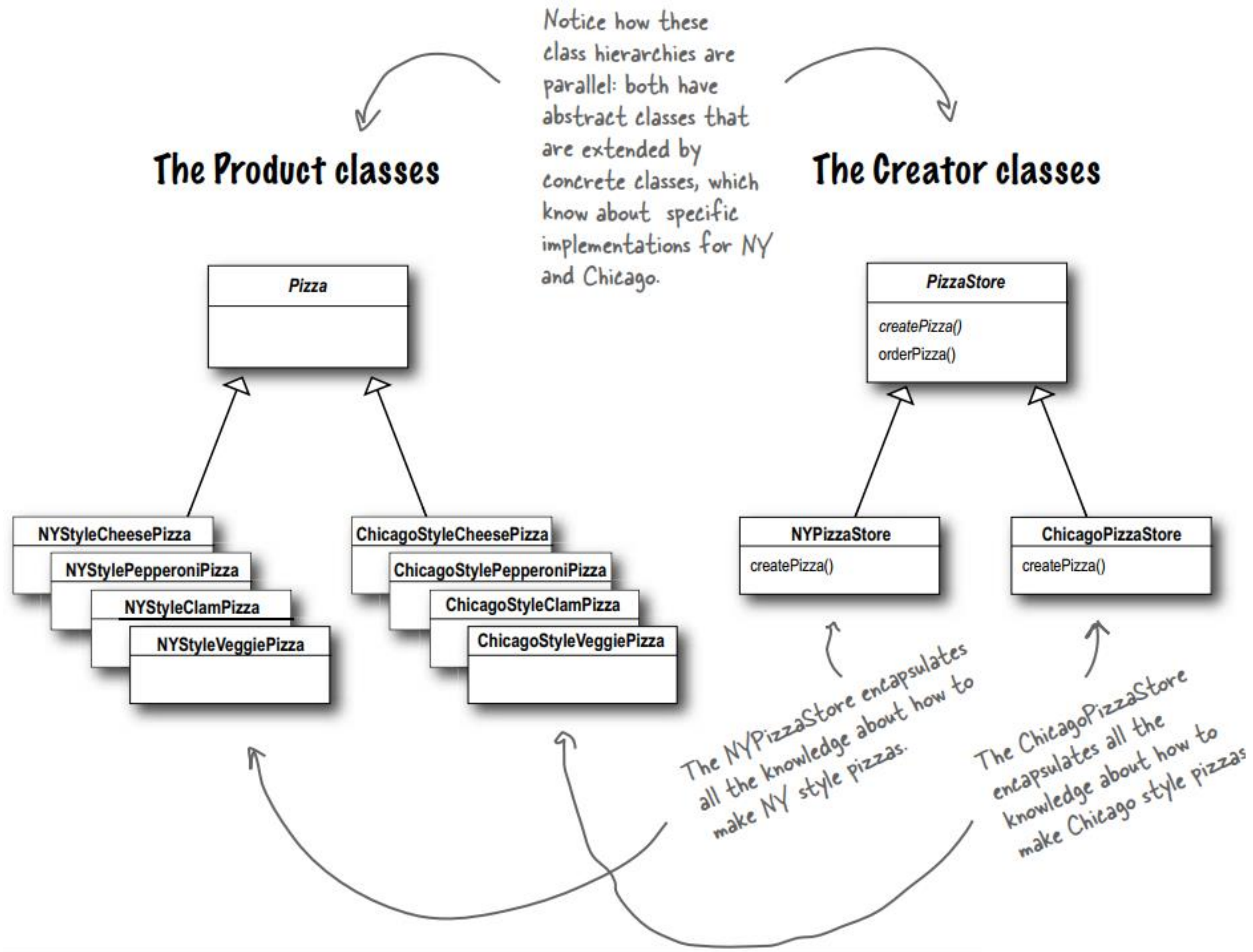
The Factory Method Pattern Defined

- **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.

The Factory Method Pattern Defined



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

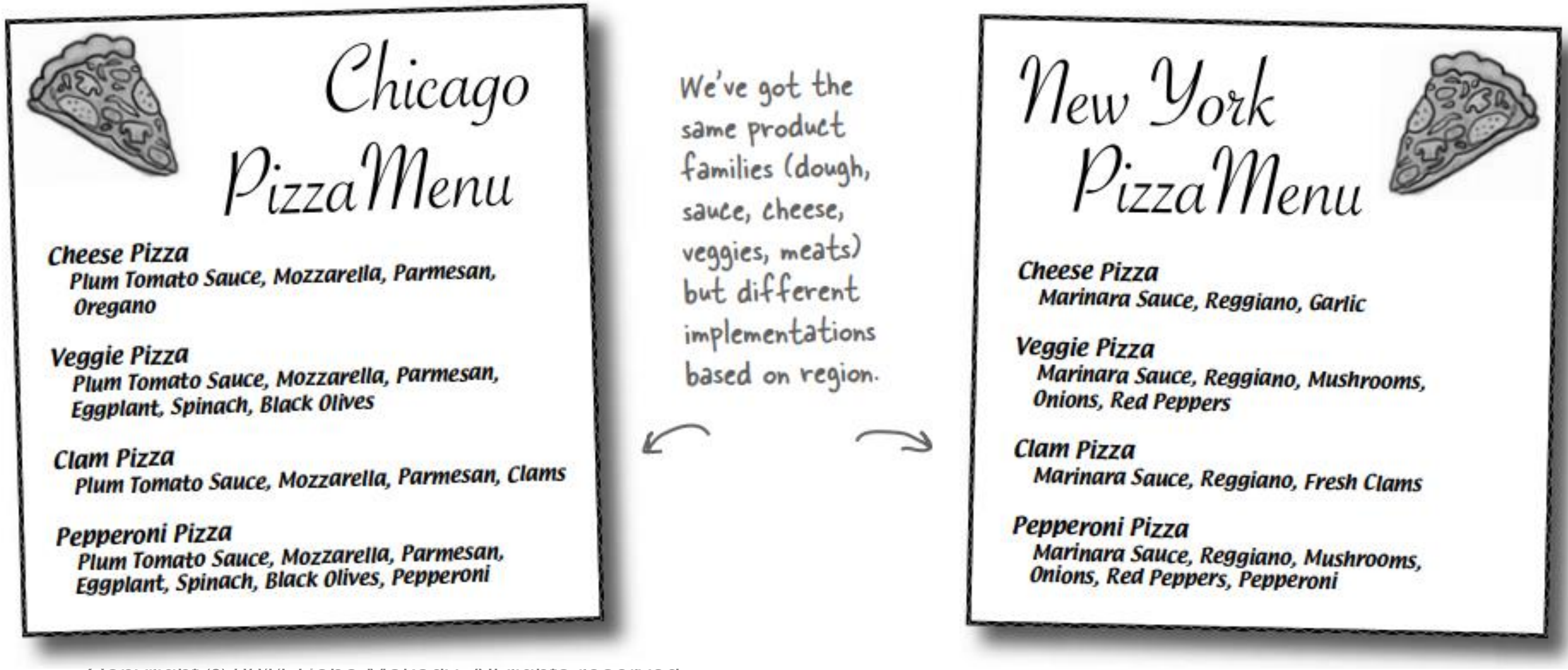
The ChicagoPizzaStore encapsulates all the knowledge about how to make Chicago style pizzas

Respecting the Ingredients?

- The factory method approach to the pizza store is a big success allowing our company to create multiple franchises across the country.
- But, bad news, we have learned that some of the franchises
  - while following our procedures (the abstract code in PizzaStore forces them to), are skimping on ingredients in order to lower costs and increase margins
- Our company's success has always been dependent on the use of fresh, quality ingredients • so "Something Must Be Done!"

Ensuring Consistency in Your Ingredients!

- So how are you going to ensure each branch set is using quality ingredients?
- You're going to build a factory that produces them and ships them to your branch sets!



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

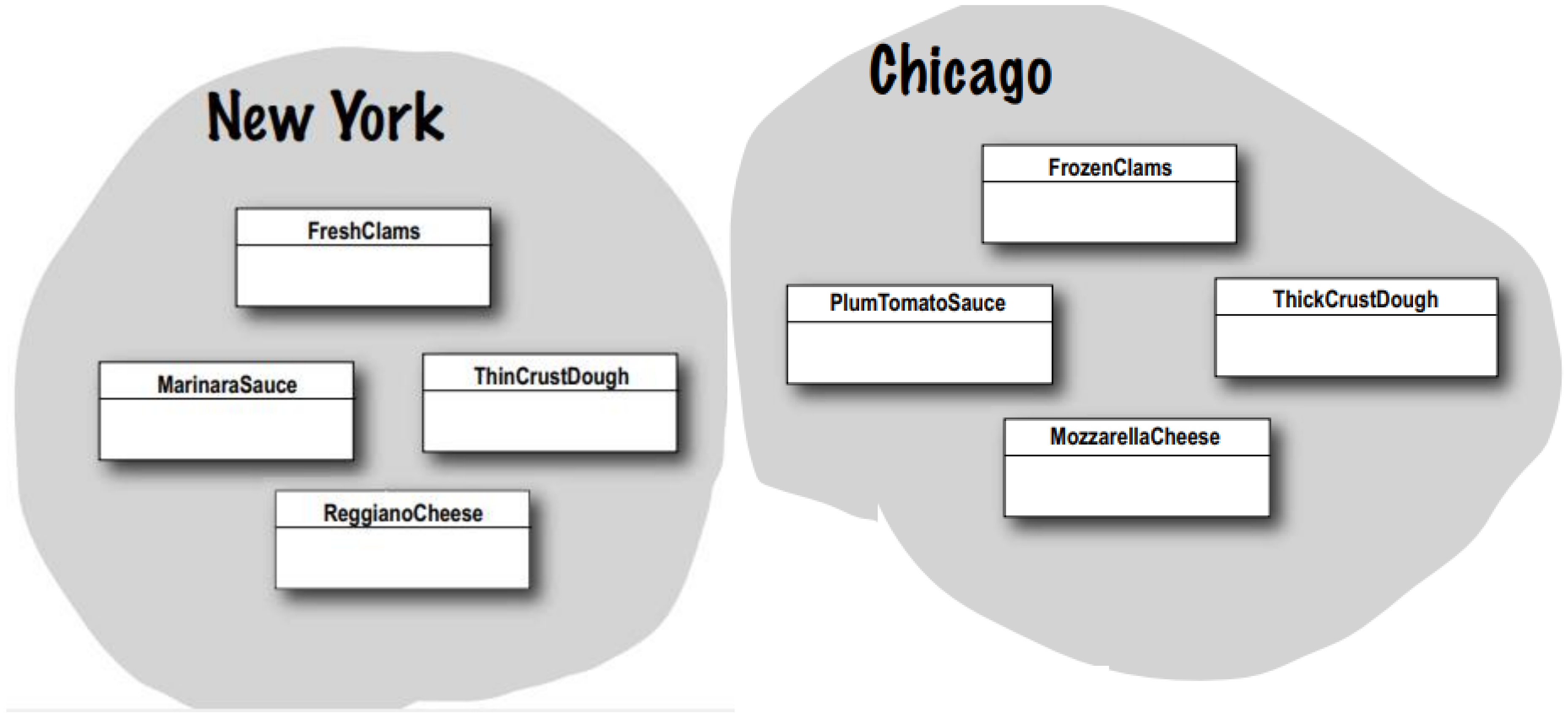
Veggie Pizza
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers

Clam Pizza
Marinara Sauce, Reggiano, Fresh Clams

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

Families of Ingredients

- New York uses one set of ingredients and Chicago another.
- For this to work, you are going to have to figure out how to handle families of ingredients



Abstract Factory to the Rescue!

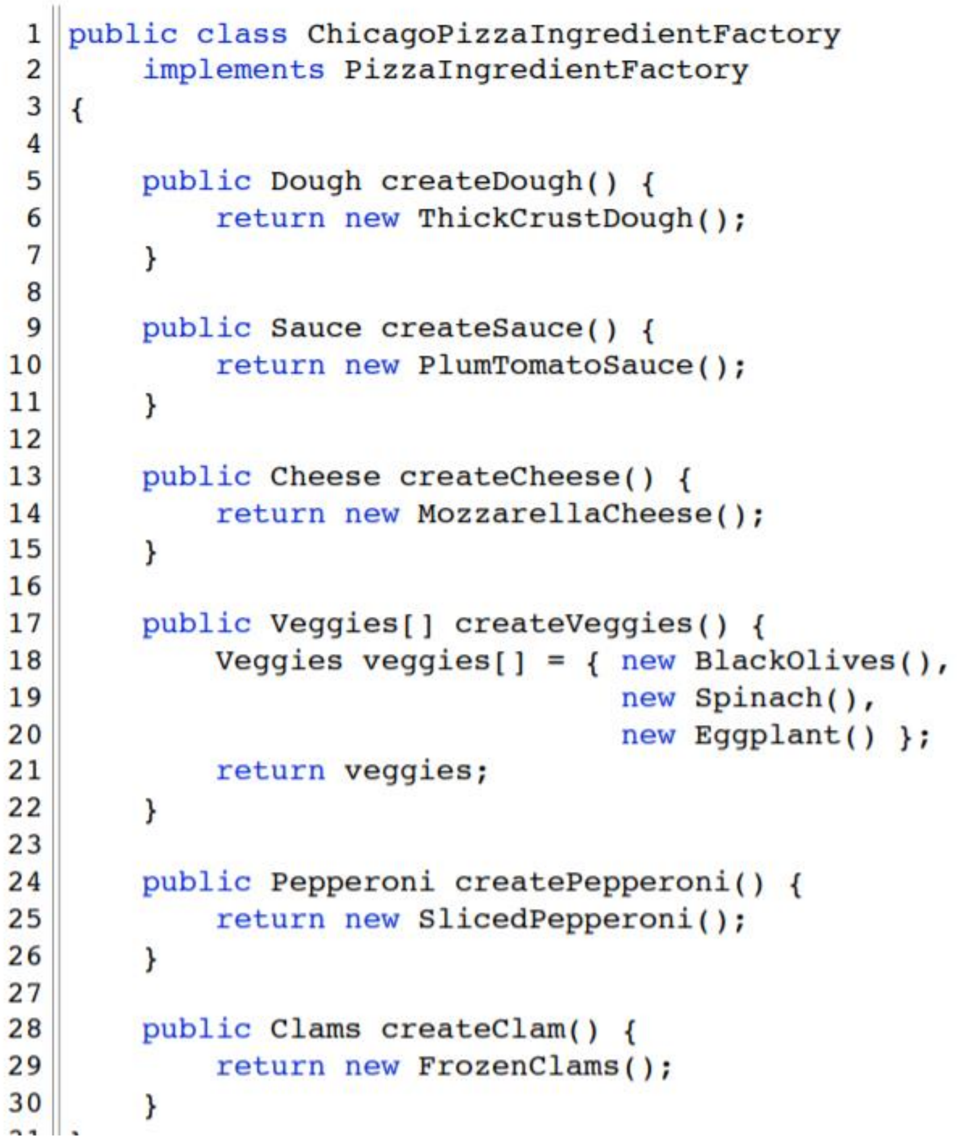
- We will alter our design such that a factory is used to supply the ingredients that are needed during the pizza creation process
  - Since different regions use different types of ingredients, we'll create region-specific subclasses of the ingredient factory to ensure that the right ingredients are used
  - But, even with region-specific requirements, since we are supplying the factories, we'll make sure that ingredients that meet our quality standards are used by all franchises
    - They'll have to come up with some other way to lower costs. ☺

First, We need a Factory Interface

```java
public interface PizzaIngredientFactory {

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

}

```

Note the introduction of more abstract classes: Dough, Sauce, Cheese, etc.

Second, We implement a Region-Specific Factory



```java
public class ChicagoPizzaIngredientFactory
    implements PizzaIngredientFactory
{

    public Dough createDough() {
        return new ThickCrustDough();
    }

    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
                              new Spinach(),
                              new Eggplant() };
        return veggies;
    }

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

    public Clams createClam() {
        return new FrozenClams();
    }
```

This factory ensures that quality ingredients are used during the pizza creation process…

… while also taking into account the tastes of people who live in Chicago

But how (or where) is this factory used?

Within Pizza Subclasses… (I)

```java
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() {
```

First, alter the Pizza abstract base class to make the prepare method abstract…

Within Pizza Subclasses… (II)

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

Then, update Pizza subclasses to make use of the factory! Note: we no longer need subclasses like NYCheesePizza and ChicagoCheesePizza because the ingredient factory now handles regional differences

One last step…

```java
public class ChicagoPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
        new ChicagoPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("Chicago Style Cheese Pizza");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("Chicago Style Veggie Pizza");
                        ...
```

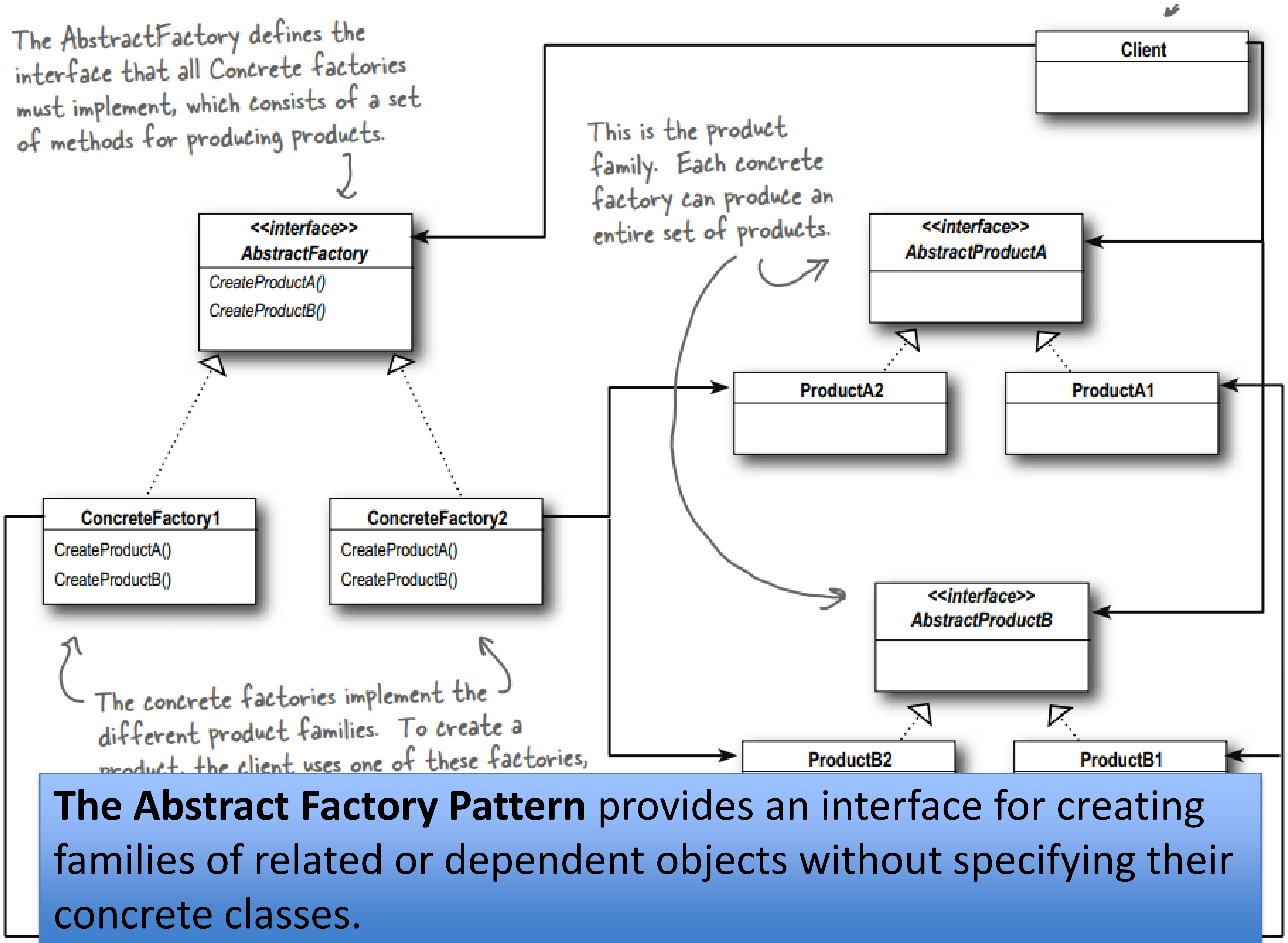
We need to update our PizzaStore subclasses to create the appropriate ingredient factory and pass it to each Pizza subclass in the createPizza factory method.

Summary: What did we just do?

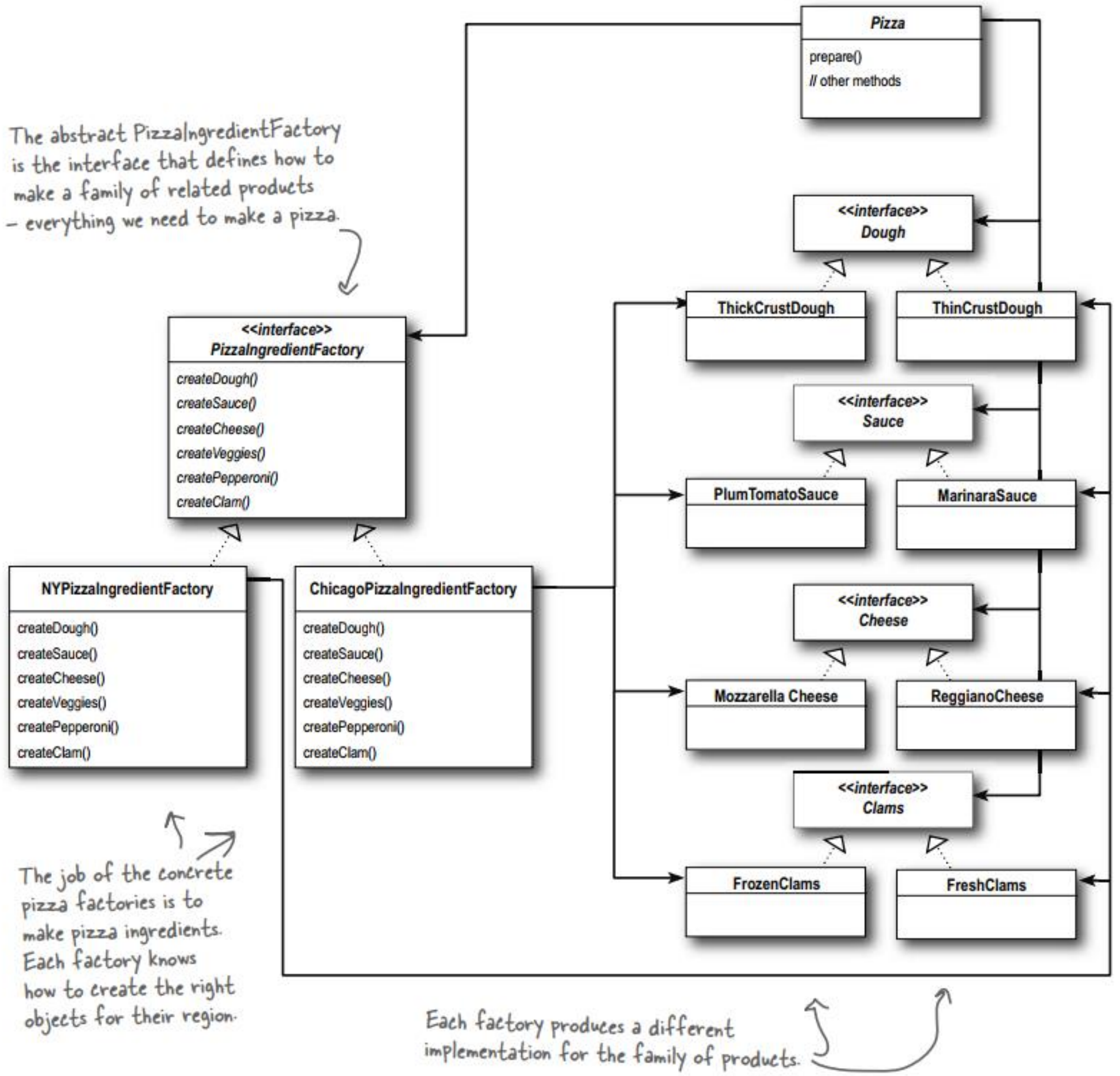
1. We created an ingredient factory interface to allow for the creation of a family of ingredients for a particular pizza
2. This abstract factory gives us an interface for creating a family of products

   The factory interface decouples the client code from the actual factory implementations that produce context-specific sets of products
3. Our client code (PizzaStore) can then pick the factory appropriate to its region, plug it in, and get the correct style of pizza (Factory Method) with the correct set of ingredients (Abstract Factory)

The Abstract Factory Pattern


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

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



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.

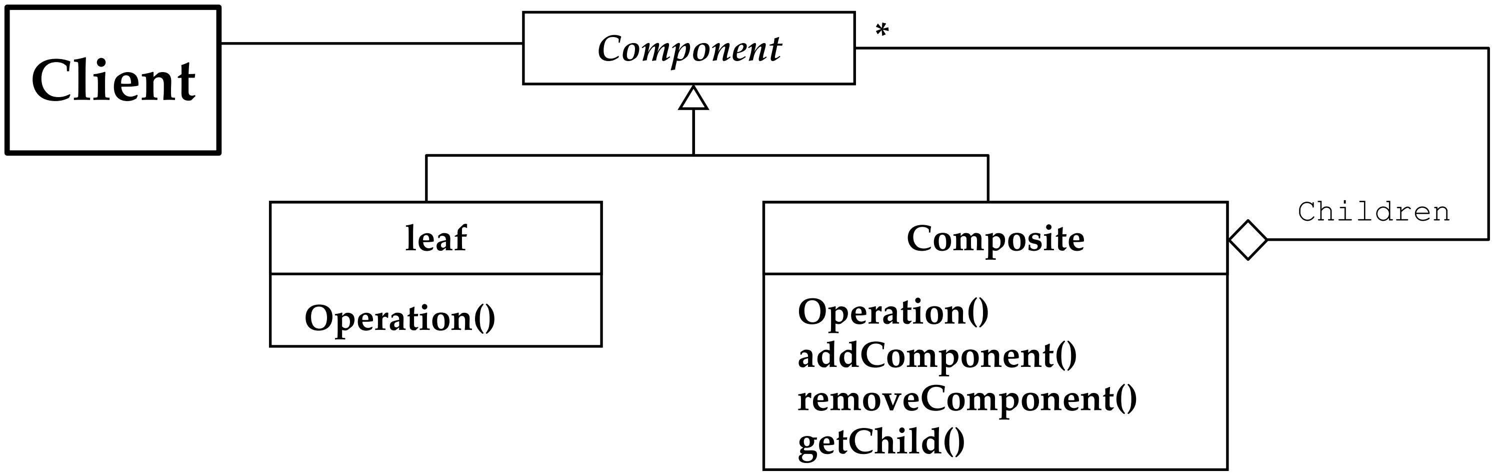
Each factory produces a different implementation for the family of products.

Abstract Factory vs. Factory

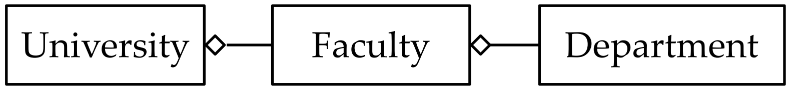
Composite Patterns

Compose objects into tree structures to

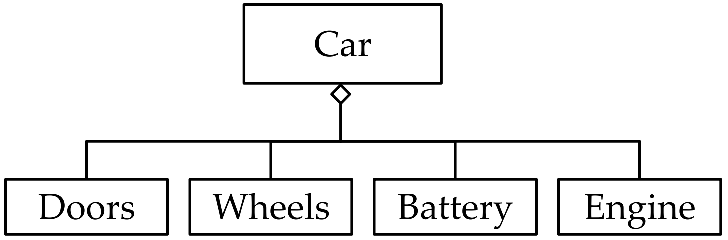
- represent part-whole hierarchies with arbitrary depth and width.
- Treat individual objects and compositions of objects uniformly



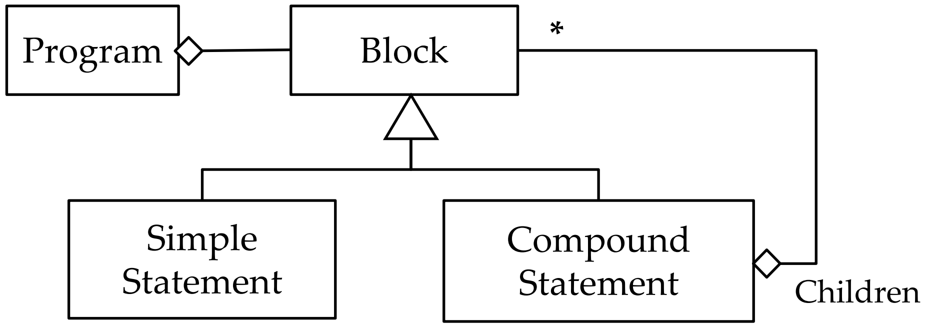
Modeling aggregates



Organization Chart (variable aggregate)



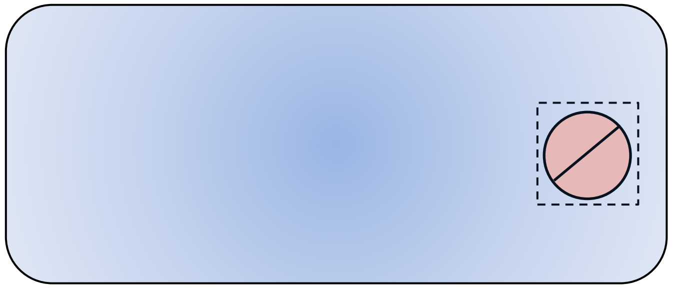
Fixed Structure

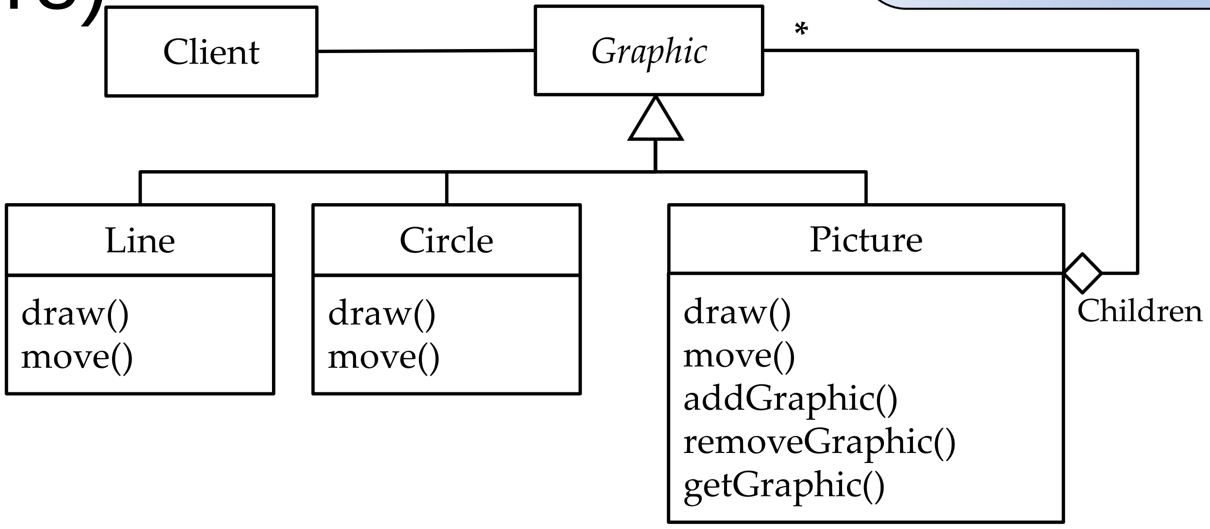


Dynamic tree (recursive aggregate)

Composite Pattern: Example

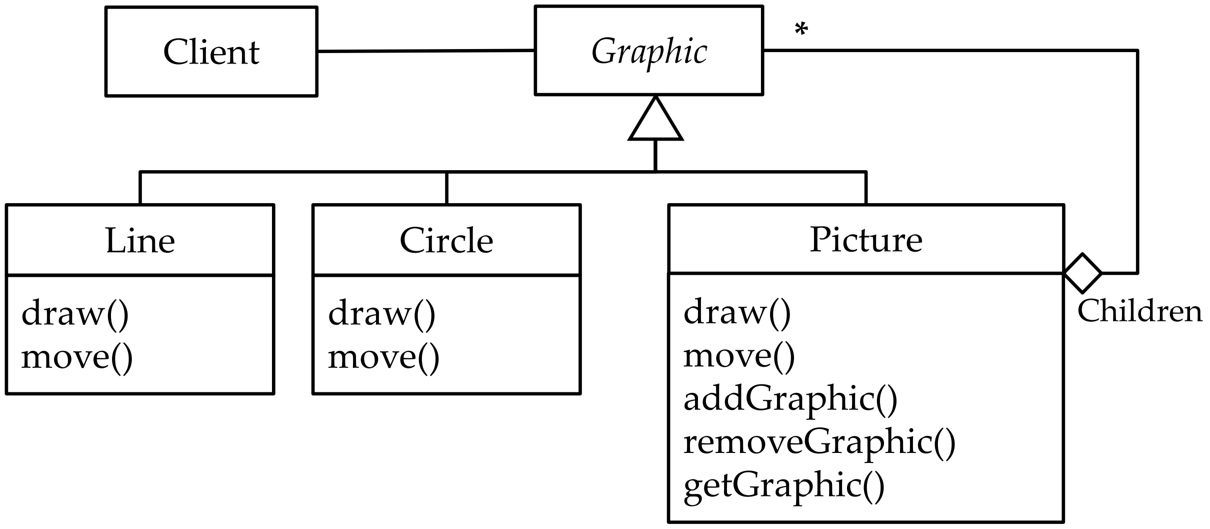
- The Graphic Class represents both primitives (Line, Circle) and their containers (Picture)





Composite Pattern: Example

- The Graphic Class represents both primitives (Line, Circle) and their containers (Picture)



Is the Composite Pattern SOLID?

- A leaf in the Composite Pattern implements the Component interface, including Add, Remove, and GetChild methods that a leaf is not going to use.
- That design violates the SOLID principles.
  - myLeaf.Add(someChild);
  - That call would have to throw a MethodNotSupportedException, return null or indicate in some other way to the caller that adding a child to a Leaf does not make sense.
- Did the GoF miss this issue in their design?

Is the Composite Pattern SOLID?

Discussion with the GoF book:

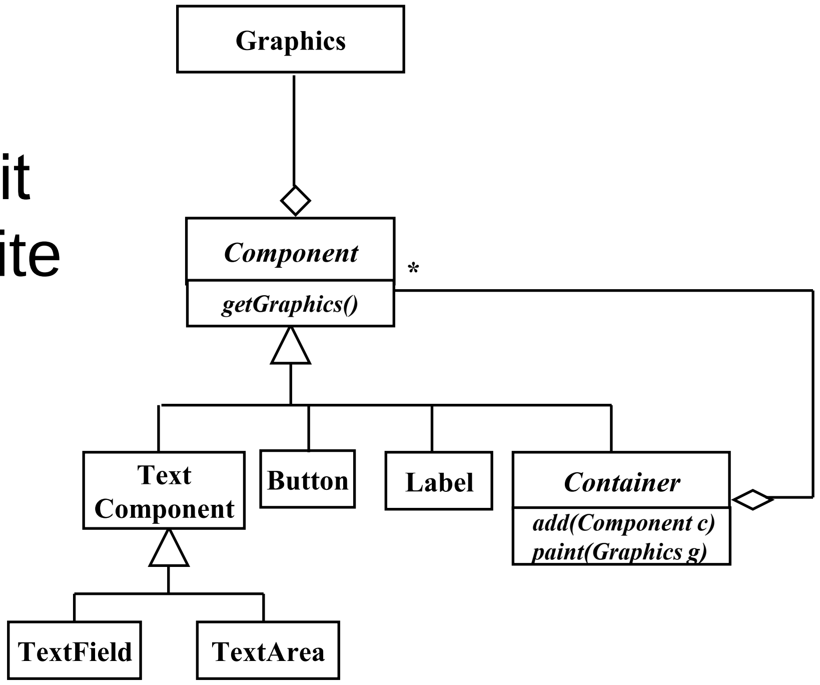
- Should we declare these operations in the Component, or should we declare and define them only in Composite?
- The decision involves a trade-off between safety and transparency:
  - Defining the child management interface at the root of the class hierarchy gives you **transparency**, because you can treat all components uniformly.
  - It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.

Is the Composite Pattern SOLID?

- Discussion with the GoF book:
- The decision involves a trade-off between safety and transparency:
  - Defining child management in the Composite class gives you **safety**, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language like C++.
  - But you lose transparency, because leaves and composites have different interfaces.
- The GoF have emphasized transparency over safety in this pattern.

Java AWT library

- Java's Abstract Window Toolkit can be modeled as a composite pattern



The Composite Pattern Defined

- **The Composite Pattern** allows you to compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
- Using a composite structure, we can apply the same operations over both composites and individual objects.

Required Readings

- Chapter 4 from Head First Design Patterns
- Chapter 9 [Composite pattern part]