Lecture 05: The Abstract Factory Pattern

SE313, Software Design and Architecture

Damla Oguz

Chapter 4: The Factory Pattern

- All factories encapsulate object creation.
- *Simple Factory (not an actual design pattern) (previous lecture)*
- *Factory Method Pattern (previous lecture)*
- Abstract Factory Pattern

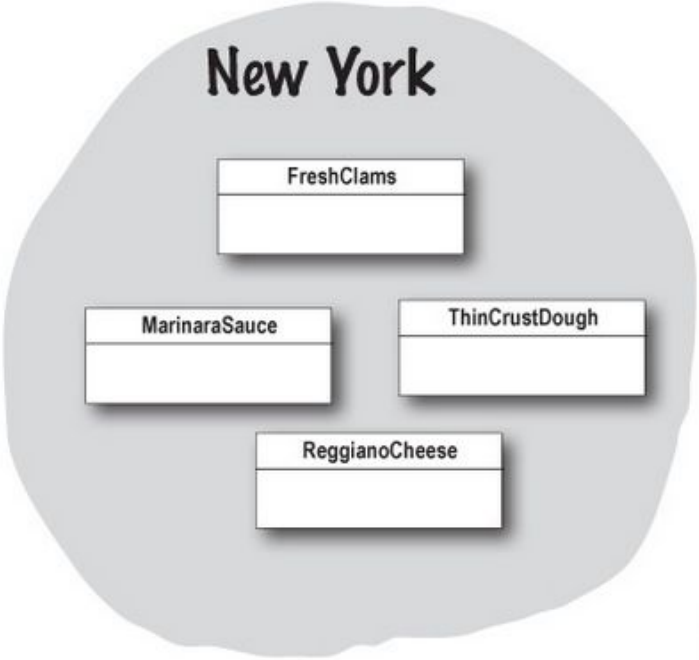
Back at the PizzaStore...

- We've discovered that a few franchises have been substituting inferior ingredients in their pies to lower costs and increase their margins.
- How are we going to ensure each franchise is using quality ingredients?
  - We're going to build a factory that produces them!

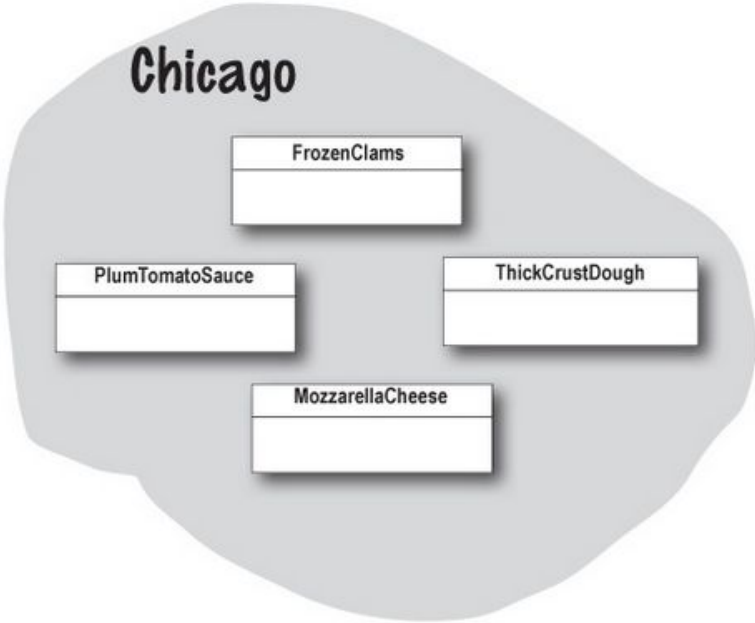
Ensuring consistency in your ingredients

- There is only one problem with this plan: the franchises are located in different regions so they use different types of ingredients.
- Actually, we've got the same product families (dough, sauce, cheese, veggies, etc.) but different implementations based on region.
- In brief, New York uses one set of ingredients and Chicago another (a different set).
- We can also need to provide another set of regional ingredients to California, Seattle, etc.
- So, we should figure out how to handle families of ingredients.

Families of ingredients...



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



Building the ingredient factories

- Now we're 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...
- We are going to handle the regional differences as well.
- Let's start by defining an interface for the factory that is going to create all our ingredients...

Building the ingredient factories (cont.)

```
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.

If we'd had some common "machinery" to implement in each instance of factory, we could have made this an abstract class instead...

Building the ingredient factories (cont.)

Here's what we're going to do:

1. Build a factory for each region. To do this, we'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.

Building the New York 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.

New York is on the coast; it gets fresh clams.

Reworking the pizzas...

- Now we just need to rework our Pizzas so they only use factory-produced ingredients.
- We'll start with our abstract Pizza class:

```
public abstract class Pizza {

    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;
```

Each pizza holds a set of ingredients that are used in its preparation.

```
    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
    }
}
```

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.

Reworking the pizzas, continued...

- Now that we've got an abstract Pizza to work from, it's time to create the New York and Chicago style Pizzas.
- This time around they will get their ingredients straight from the factory.
- When we wrote the Factory Method code, we had a NYCheesePizza and a ChicagoCheesePizza class.
- If you look at the two classes, the only thing that differs is the use of regional ingredients. The pizzas are made just the same (dough + sauce + cheese). The same goes for the other pizzas: Veggie, Clam, and so on.
- So, we really don't need two classes for each pizza; the ingredient factory is going to handle the regional differences for us.
- Here's the Cheese Pizza...

Reworking the pizzas, continued...

```
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.

Reworking the pizzas, continued...

```
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();
    }
}
```

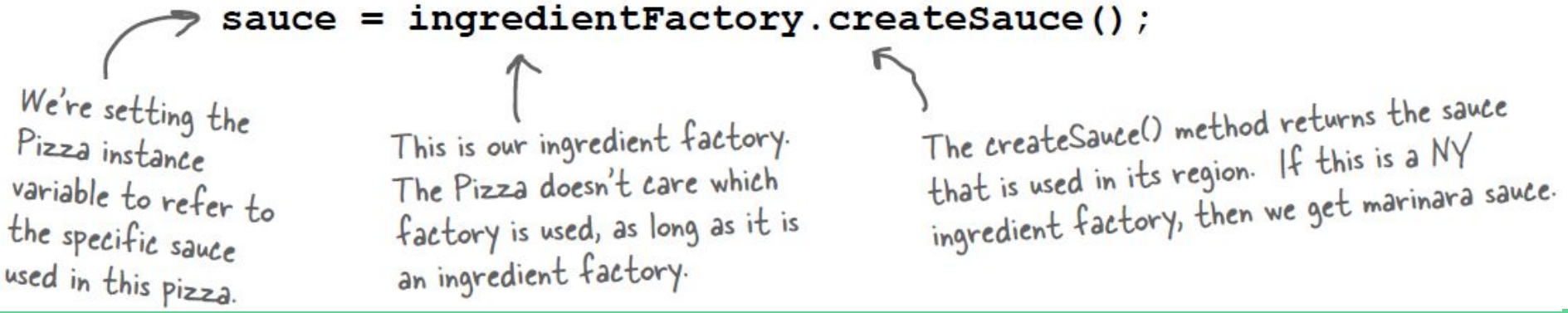
ClamPizza also stashes an ingredient factory.

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.

Code Up Close

- The Pizza code uses the factory it has been composed with to produce the ingredients used in the pizza.
- The ingredients produced depend on which factory we're using. The Pizza class doesn't care; it knows how to make pizzas.
- Now, it's decoupled from the differences in regional ingredients and can be easily reused when there are factories...



Revisiting our pizza stores

- We just need to to make sure our franchise stores are using the correct Pizzas.
- We also need to give them a reference to their local ingredient factories...

```
public abstract class PizzaStore {

    protected abstract Pizza createPizza(String item);

    public Pizza orderPizza(String type) {
      Pizza pizza = createPizza(type);
      System.out.println("--- Making a " + pizza.getName() + " ---");
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
    }
}
```

PizzaStore class is not changed.

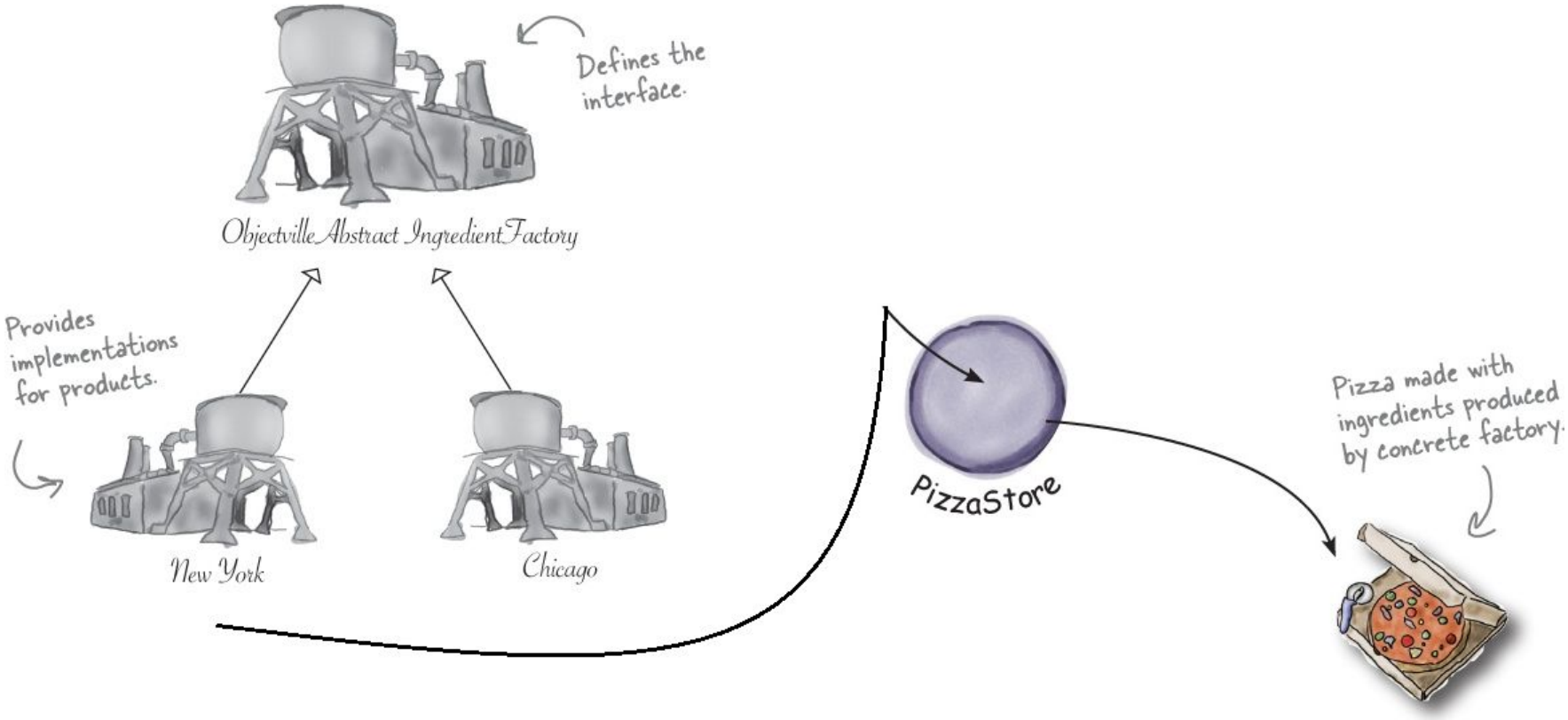
```
public class NYPizzaStore extends PizzaStore {

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

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

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

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

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

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

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

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

            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("New York Style Pepperoni Pizza");
      }
      return pizza;
    }
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas

We now pass each pizza the factory that should be used to produce its ingredients.

For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.

What have we done?



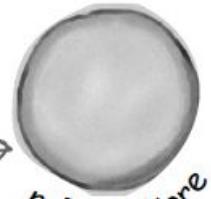
More pizza for Ethan and Joel...

- Ethan and Joel still love NY Style and Chicago Style pizzas, respectively.
- Let’s follow Ethan’s order again...

1 **First we need a NY PizzaStore:**

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

Creates an 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.

3 **The orderPizza() method first calls the cre-atePizza() method:**

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

createPizza("cheese")

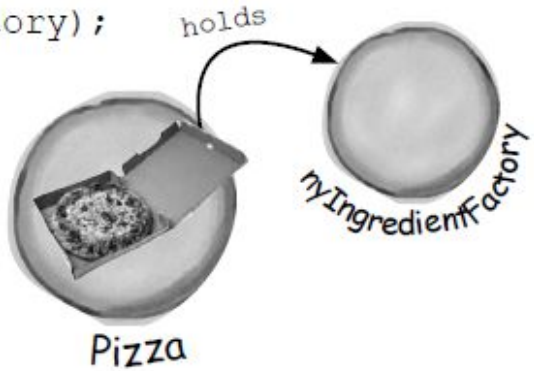
From here things change, because we are using an ingredient factory.

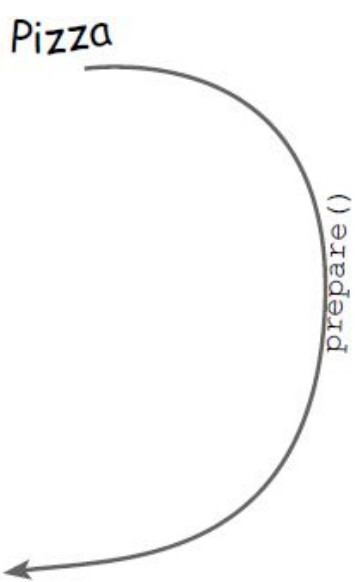
4 **When the createPizza() method is called, that's when our ingredient factory gets involved:**

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 **Next we need to prepare the pizza. Once the prepare() method is called, the factory is asked to prepare ingredients:**

```
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.

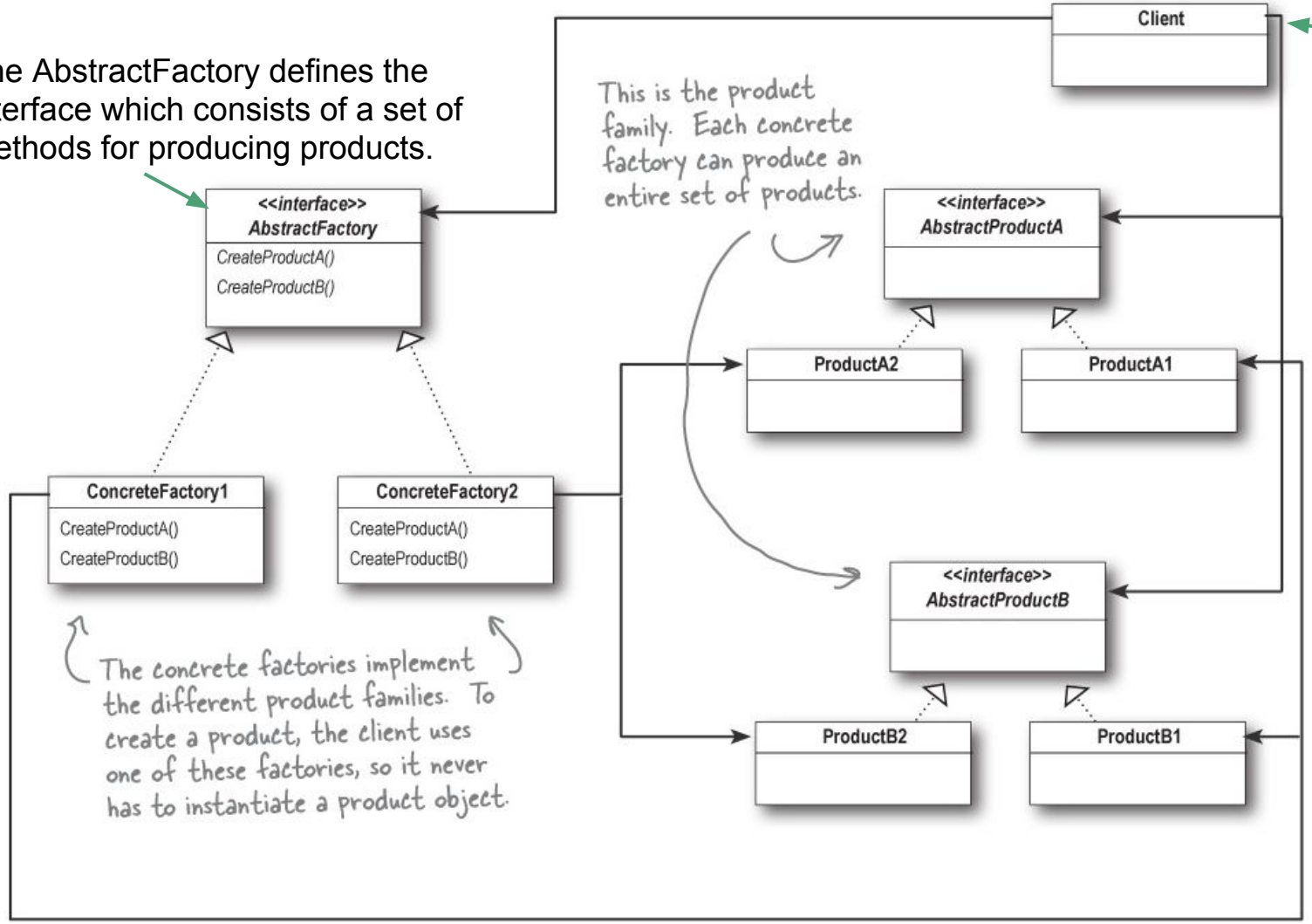
6 **Finally we have the prepared pizza in hand and the orderPizza() method bakes, cuts, and boxes the pizza.**

Abstract Factory Pattern defined

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

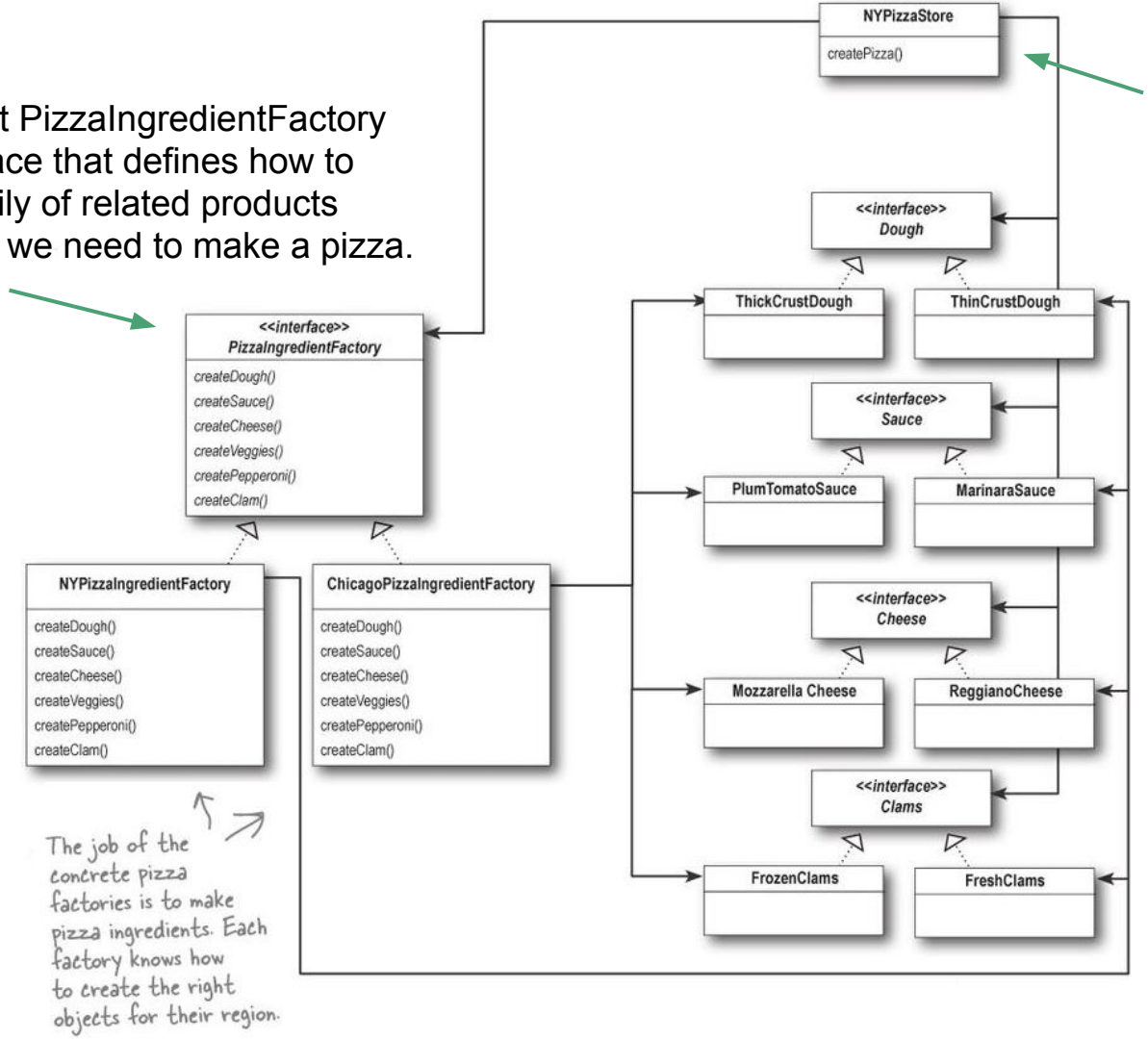
- Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products.
- Let’s look at the class diagram to see how this all holds together…

The AbstractFactory defines the interface which consists of a set of methods for producing products.



The Client is written against the abstract factory and then composed at runtime with an actual factory.

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



The clients of the Abstract Factory are the concrete instances of our PizzaStore, NYPizzaStore and ChicagoPizzaStore.

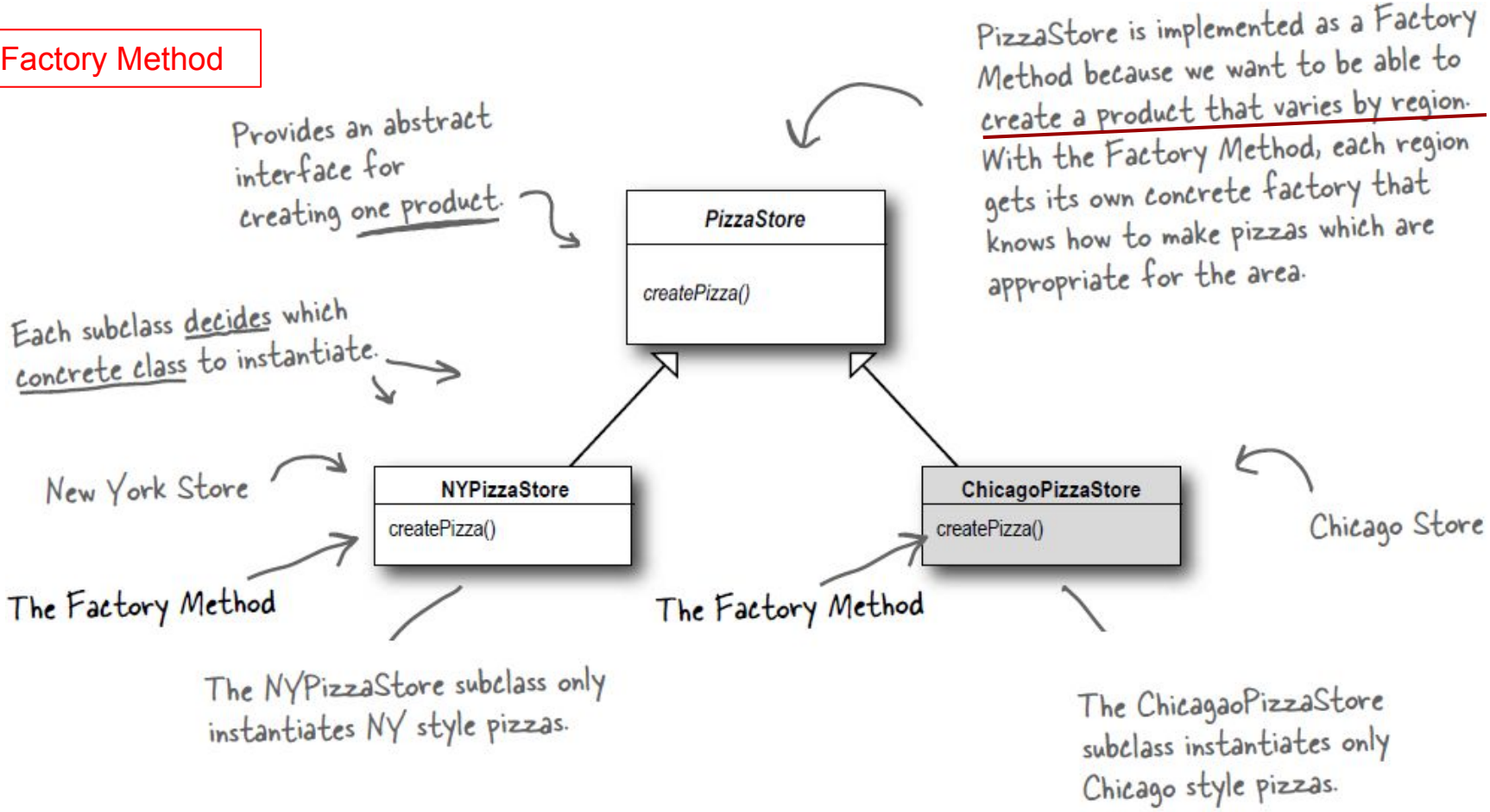
Each factory produces a different implementation for the family of products.

Factory Method and Abstract Factory compared

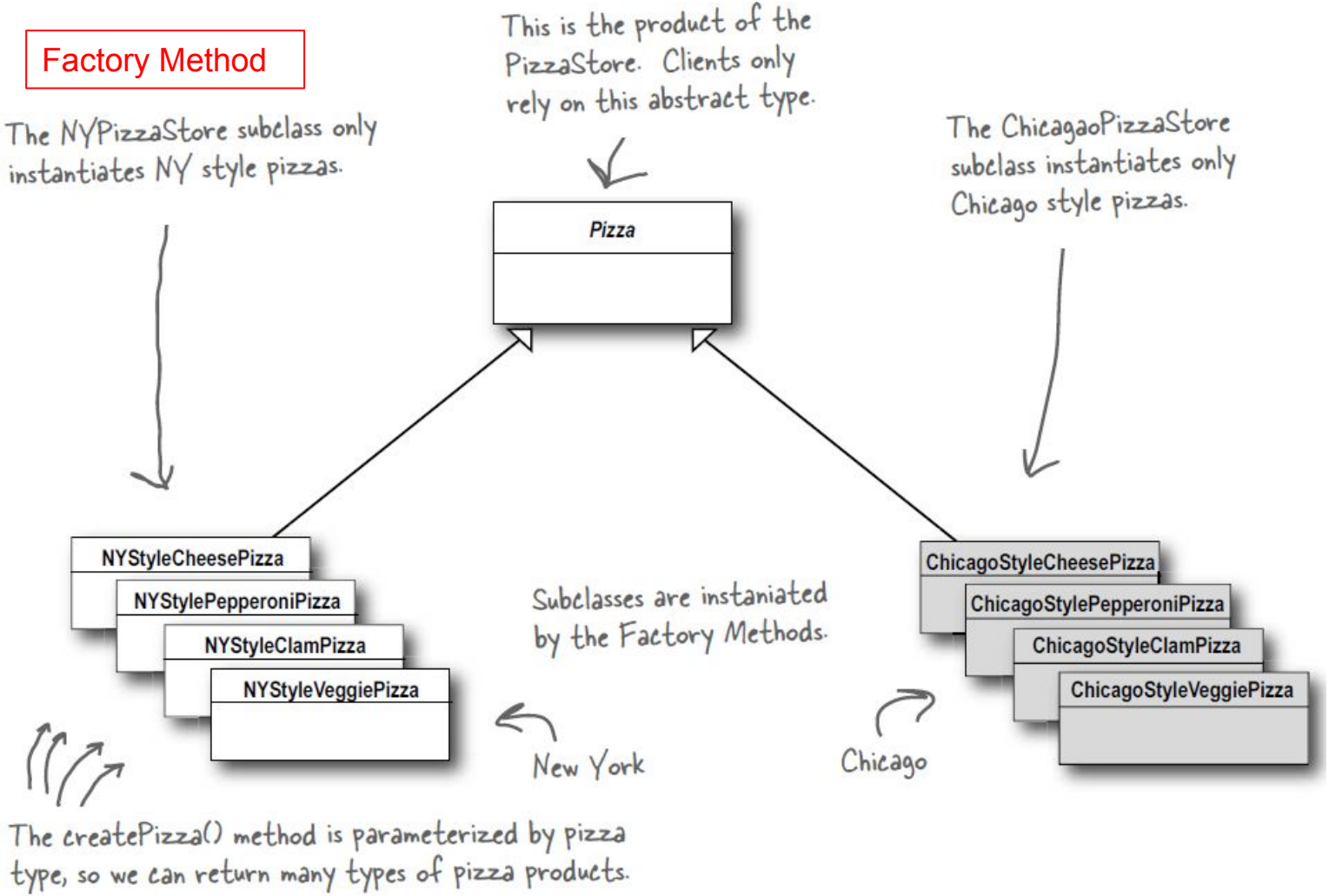
Q: I noticed that each method in the Abstract Factory actually looks like a Factory Method (createDough(), createSauce(), etc.). Each method is declared abstract and the subclasses override it to create some object. Isn't that Factory Method?

- Yes, often the methods of an Abstract Factory are implemented as factory methods.
- The job of an Abstract Factory is to define an interface for creating a set of products. Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations.
- So, factory methods are a natural way to implement your product methods in your abstract factories.

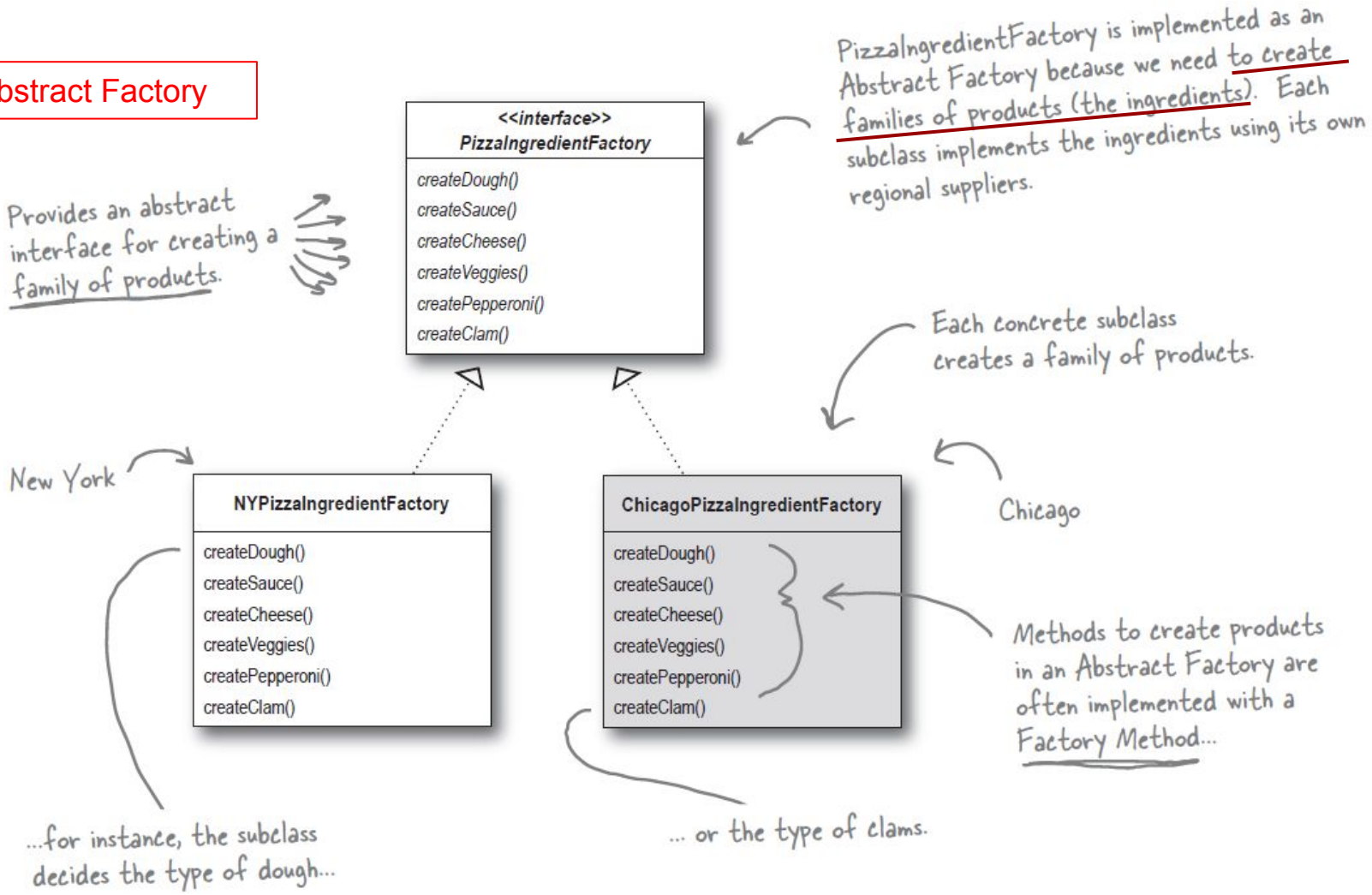
Factory Method



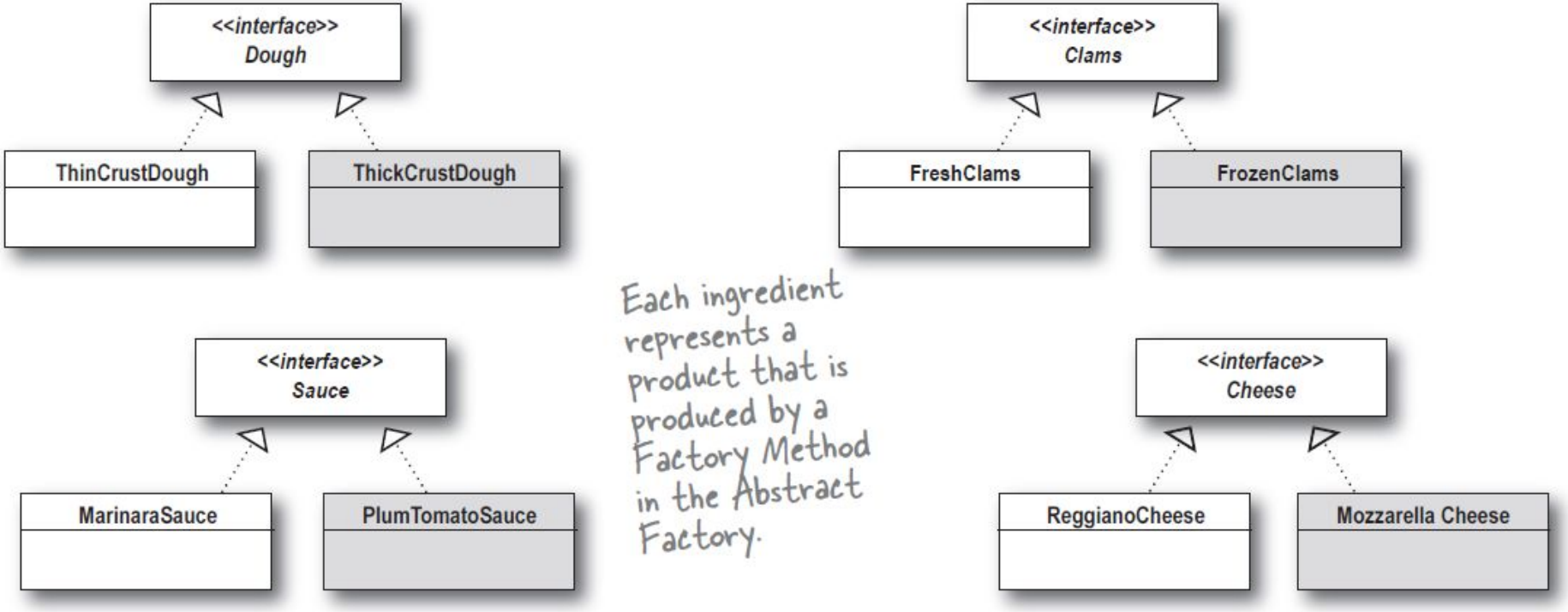
Factory Method



Abstract Factory



Abstract Factory



The product subclasses create parallel sets of product families. Here we have a New York ingredient family and a Chicago family.

Some Bullet Points

- The whole point of the Factory Method Pattern is that we're using a subclass to do our creation for us.
  - In that way, clients only need to know the abstract type they are using, the subclass worries about the concrete type.
  - In other words, Factory Method Pattern keeps clients decoupled from the concrete types.
  - It's only creating one product.

Some Bullet Points (cont.)

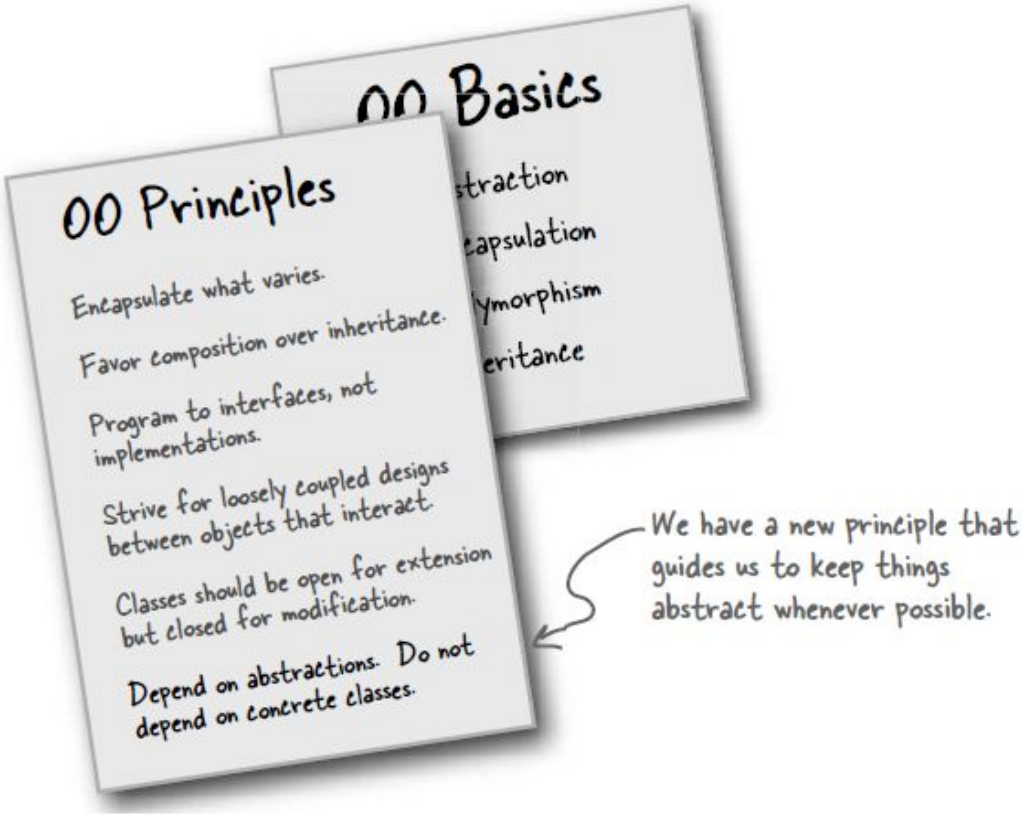
- Abstract Factory provides an abstract type for creating a family of products.
  - Subclasses of this type define how those products are produced.
  - So, like Factory Method, the clients are decoupled from the actual concrete products they use.
  - Another advantage is that it groups together a set of related products.
  - It is used to create entire families of products.
- Both Factory Method and Abstract Factory encapsulate object creation to keep applications loosely coupled and less dependent on implementations.

Some Bullet Points (cont.)

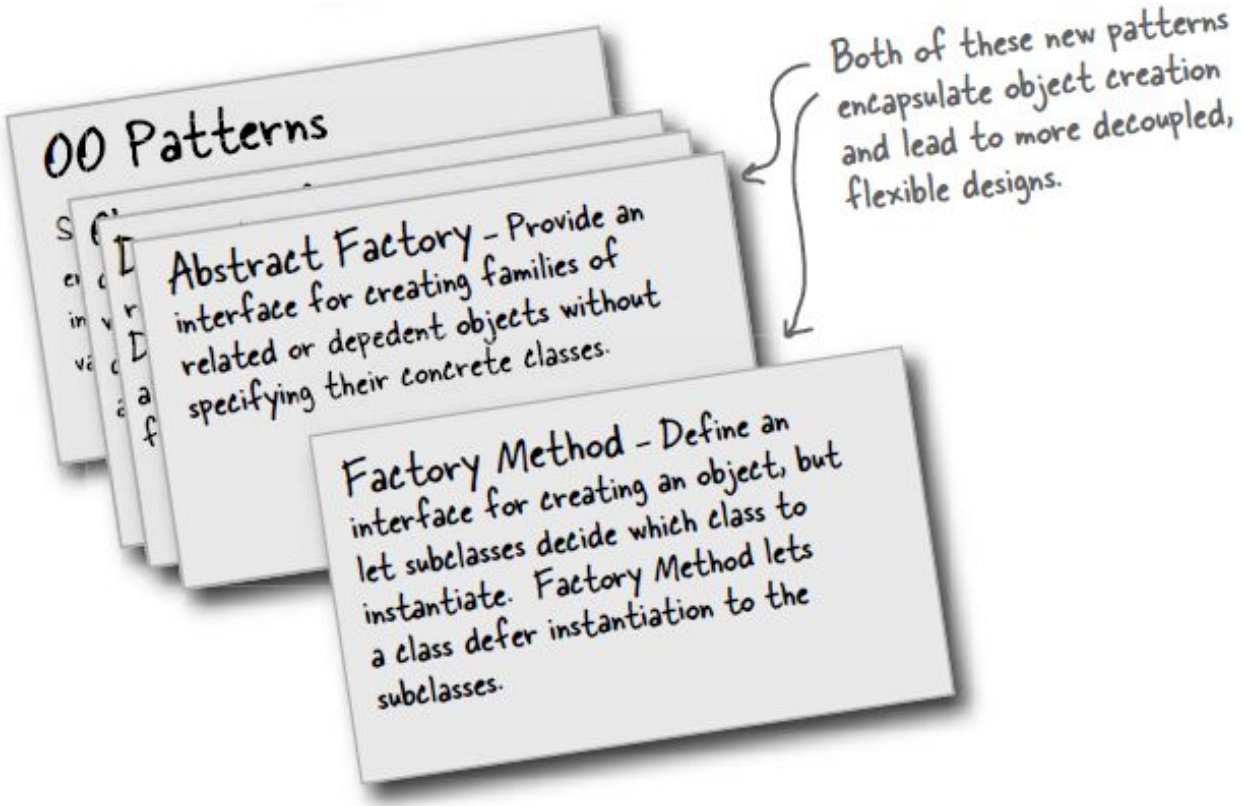
- **Factory Method** relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects.
- The intent of **Factory Method** is to allow a class to defer instantiation to its subclasses.

- **Abstract Factory** relies on object composition: object creation is implemented in methods exposed in the factory interface.
- The intent of **Abstract Factory** is to create families of related objects without having to depend on their concrete classes.

Tools for your Design Toolbox



Tools for your Design Toolbox (cont.)



References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2004.