SOFTWARE ENGINEERING

Software Design

Design Patterns (Creational)

Course ID 06016410, 06016321

Nont Kanungsukkasem, B.Eng., M.Sc., Ph.D.
nont@it.kmitl.ac.th

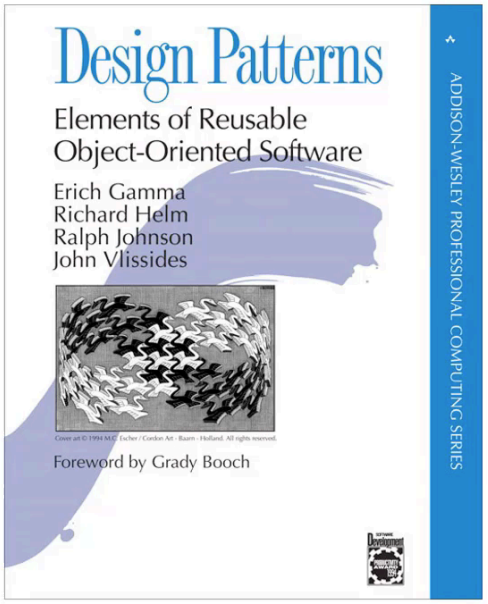
Patterns

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Ale77]
- Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience [Hill]
- Patterns are a way of reusing the knowledge and experience of other designers.

[Ale77] Christopher Alexander, A Pattern Language, 1977
[Hill] The HIllside Group: hillside.net/patterns

Design Patterns

- A design pattern can be characterized as “a three-part rule which expresses a relation between a certain **context**, a **problem**, and a **solution**” [Ale79]
- They are reusable solutions to common programming problems.
- Popularized by the Gang of Four book (1995)
  - Smalltalk & C++
- Translated to many OOP languages
- Universally relevant
  - Internalized in some programming languages
  - Libraries
  - Your own code!



[Ale79] Christopher Alexander,, The Timeless Way of Building, Oxford University Press, 1979.

Elements of design pattern

- Four essential elements of design pattern:
  - **Name**: a handle we can use to describe a design problem. It should be meaningful
  - **Problem**: a description of the problem the pattern addresses
  - **Solution**: description of the elements that make up the design, their relationships, responsibilities, and collaborations
  - **Consequences**: results and trade-offs of applying the pattern

Describing Design Patterns

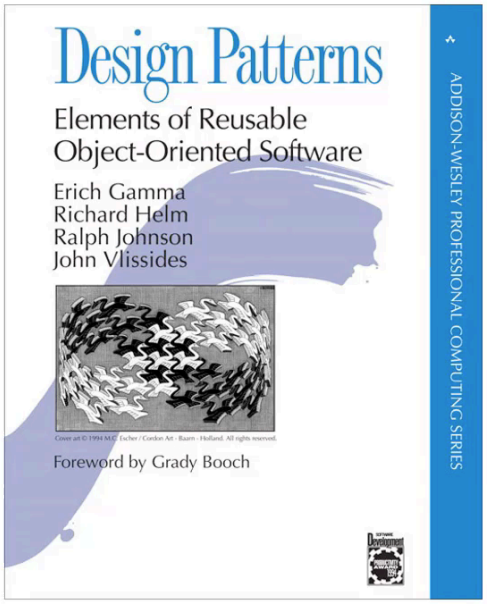
- Pattern Name and Classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample code
- Known Uses
- Related patterns
- Plus
  - **Context**

Design Patterns

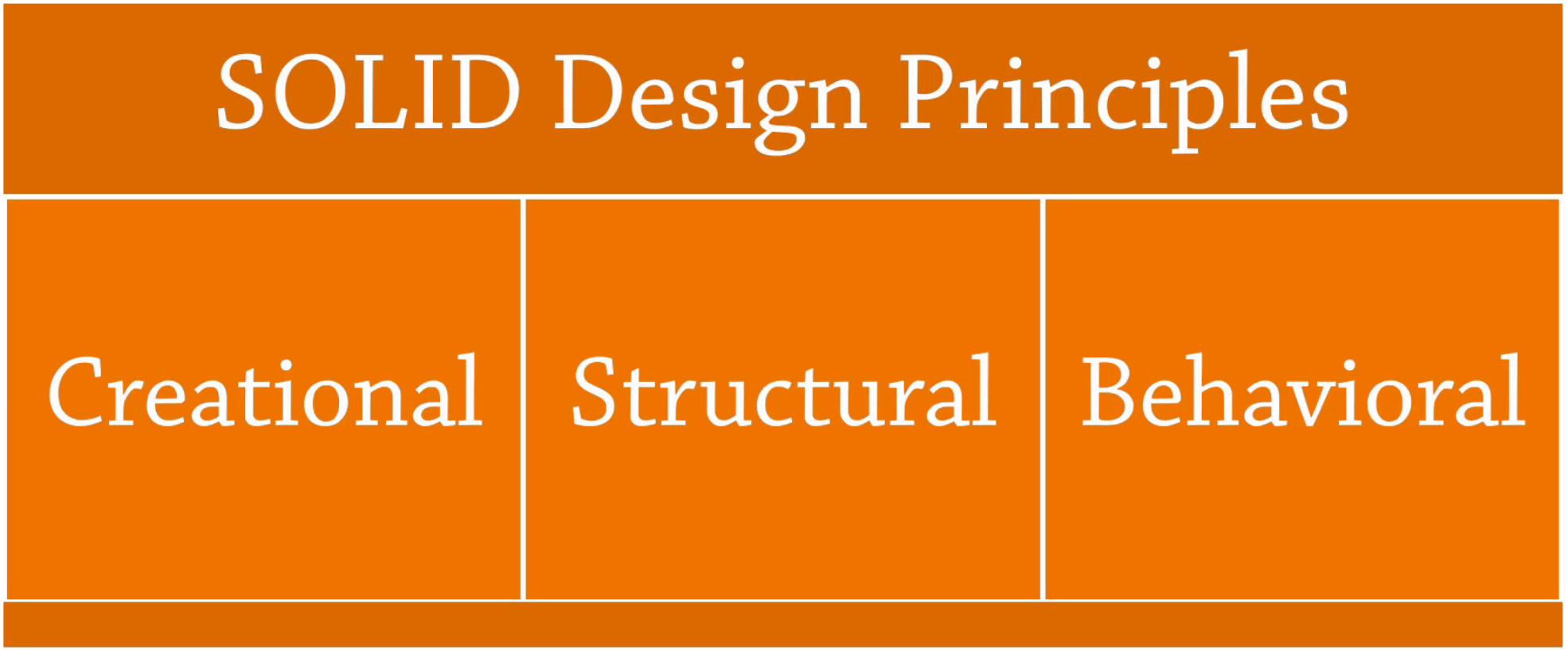
- The original foundational patterns (known as the GoF patterns)
- Sun's J2EE patterns
- JSP patterns
- Architectural patterns
- Game design patterns
- and a lot more
  - Buschmann et al. 1996. Pattern-Oriented Software Architecture Volume 1: A System of Patterns.
  - Schmidt et al. 2000. Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects.
  - Kircher and Jain. 2004. Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management.
  - Buschmann et al. 2007a. Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing.
  - Buschmann et al. 2007b. Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages.
  - Proceedings of PLoP, EuroPLoP, ChiliPLoP, …
  - anti-patterns

Design Patterns
Gamma, Helm, Johnson & Vlissides, 1995

- Aka. “The Gang of Four”, hence the GoF acronym
- Influential software design pattern catalogue defining and classifying 23 recurring design patterns in OO systems
  1. Creational design patterns
  2. Structural design patterns
  3. Behavioral design patterns



Design Patterns



The Patterns

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Creational Design Patterns

1) Creational Design Patterns
Rationale

- Concern the process of class/object creation
- Separate the operations of an application from the creation of classes/objects
- Added flexibility in configuring all aspects of creation
  - Static (compile-time)
  - Dynamic (run-time)

1) Creational Design Patterns
Catalogue (1)

- **Singleton pattern**: Ensure that a class has only one instance, and provide a global point of access to it.
- **Factory Method pattern**: Define an abstract class/ interface (*i*. abstract product) for creating different types of objects of subclasses (*ii*. concrete products.) The Factory method lets a class defer instantiation to subclasses.
- **Abstract Factory pattern**: Provide an abstract class/ interface for creating families of related or dependent objects without specifying their concrete classes.

1) Creational Design Patterns
Catalogue (2)

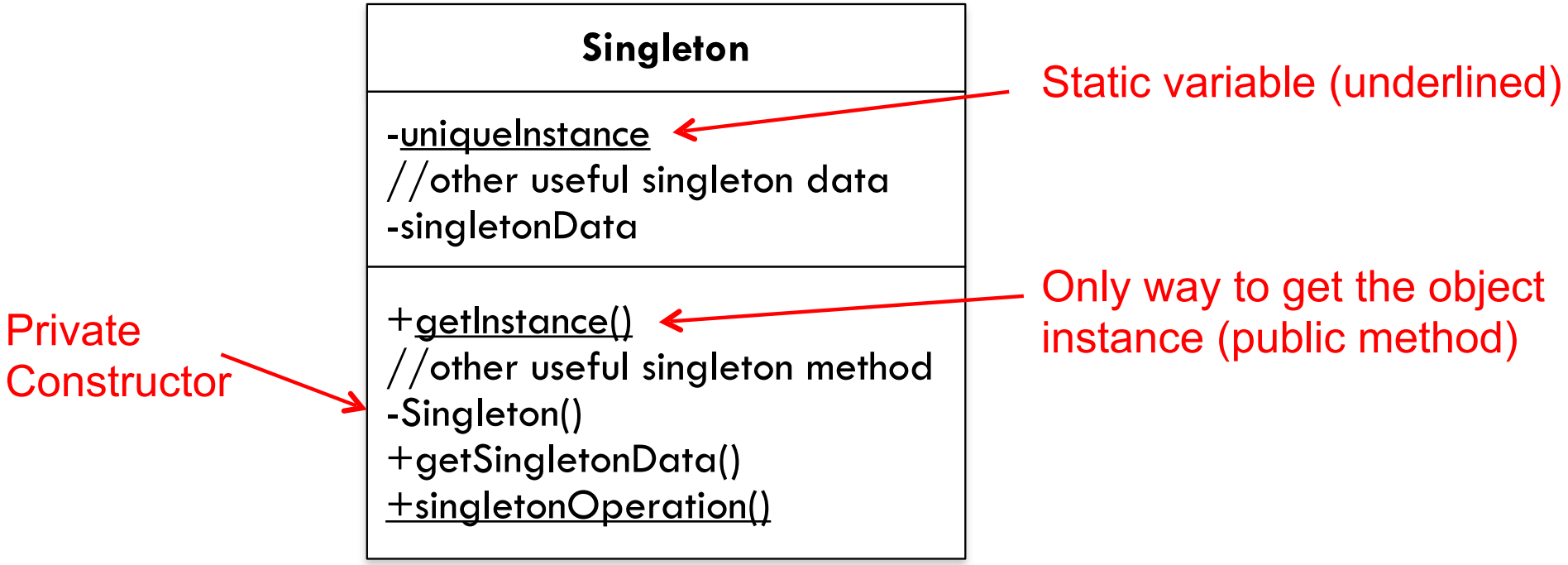
- **Builder pattern:** Separate construction of a complex object from its representation so that the same construction process can create different representations.
- **Prototype pattern:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton Pattern – Description (1)

- **Name:** Singleton
- **Problem:** How can one design a class that should have only one instance and that can be accessed globally within the application?
- **Context:** In some applications, it is important for a class to have exactly one instance.
  A sales order processing application may be dealing with sales for one company. It is necessary to have a Company object that holds details of the company. Clearly there should only be one such object.

Singleton Pattern – Description (2)

- **Solution:** Create a static method getInstance(). When accessed the first time, it creates the object instance and returns a reference to the created object. On subsequent accesses, no additional instance is created, and the reference to the existing object is returned.



Singleton Pattern: Example of Use

- Company details stored in a class
  - The constructor is defined private to the class
  - The constructor is called within a public static method of the class that ensures only a single object exists

```
class Singleton {
    // Private static instance of the class.
    3 usages
    private static Singleton instance;

    // Private constructor to prevent instantiation from outside the class.
    1 usage
    private Singleton() {}

    // Public static method to get the instance of the class.
    2 usages
    public static Singleton getInstance() {
        if (instance == null) {
            // If an instance doesn't exist, create one.
            instance = new Singleton();
        }
        return instance;
    }
}
```

  - Clients call the public method to access the object

```
Singleton firstInstance = Singleton.getInstance();
Singleton secondInstance = Singleton.getInstance();
//Singleton thirdInstance = new Singleton();
```

```
Singleton.getInstance().showMessage();
```

Singleton Pattern: Consequences

- Controlled access to sole instance
- Reduced name space
- Permits refinement of operations and representation
- Permits a variable number of instances
- More flexible than class operations

Simple Factory – Description

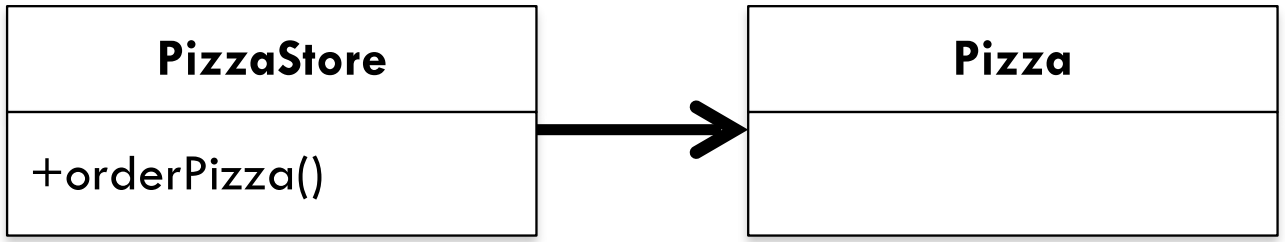
- **Name:** Factory
- **Problem:** How can one design a class that could create more than one type of objects (concrete products)? Furthermore, this class is flexible enough to be extended for newly-developed object types.
- **Context:** In some applications, it is important to assign one class as factory/creator so that it can produce a variety of object types.
  A pizza store application may be making different types of pizza according to customer orders via a pizza store. It is necessary to have a Factory object that produce a pizza based on the type ordered.

Example: Pizza Store (1)

- For example, let's say we have a pizza shop that takes orders from customer to make a nice Italian pizza.

```
public class PizzaStore {
    public Pizza orderPizza() {
        Pizza p = new Pizza();

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

For flexibility, we actually wanted this Pizza class to be an abstract class or interface so that we can make different types of pizzas.

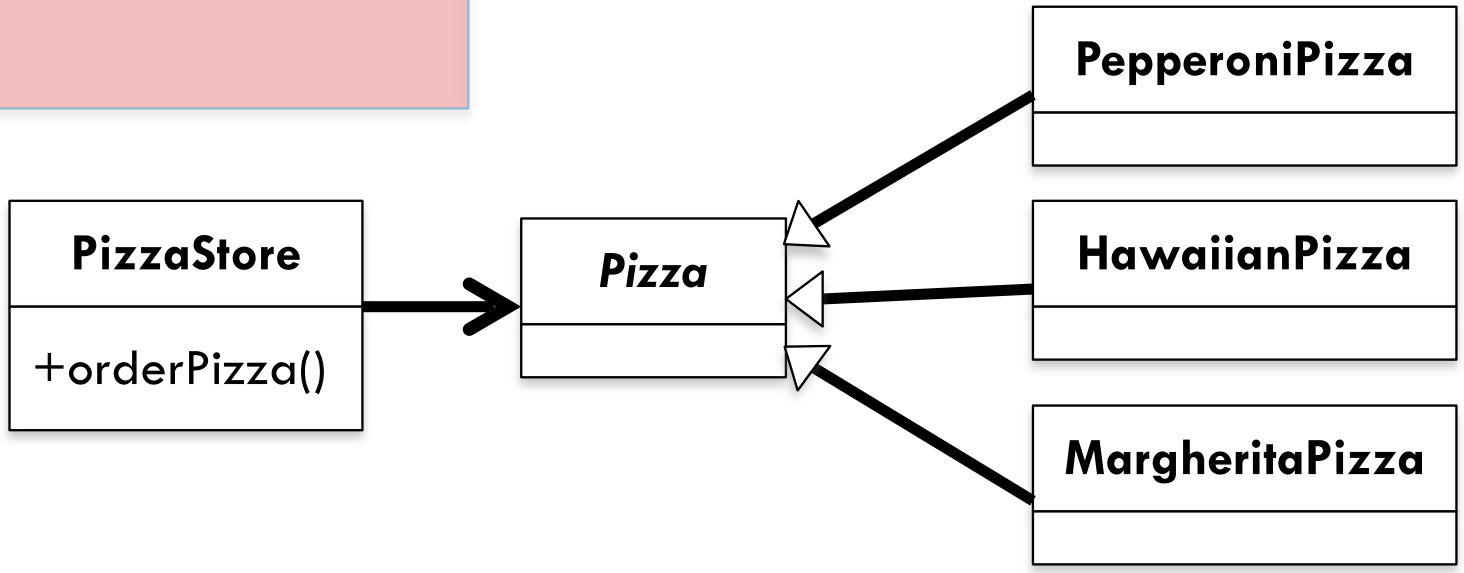


Example: Pizza Store (2)


```
public Pizza orderPizza(String type) {
        Pizza p;
        if (type.equals("pepperoni")) {
            p = new PepperoniPizza();
        } else if (type.equals("hawaiian")) {
            p = new HawaiianPizza()
        } else if (type.equals("cheese")) {
            p = new MargheritaPizza();
        }

        p.prepare();
        ...
        p.box();
        return p;
}
```

- We want our store to accept orders for more types of pizza and then goes about making the pizza.

Based on the order, we instantiate the correct concrete class and assign it to the pizza *p* reference variable.



Example: Pizza Store (3)


```
public Pizza orderPizza(String type) {
        Pizza p;
        if (type.equals("pepperoni")) {
            p = new PepperoniPizza();
        } else if (type.equals("hawaiian")) {
            p = new HawaiianPizza()
        } else if (type.equals("cheese")) {
            p = new MargheritaPizza();
        } else if (type.equals("tomyum")) {
            p = new TomyumPizza();
        }

        p.prepare();
        ...
        p.box();
        return p;
}
```

- Our competitors have added a new taste of pizza to their menus. Obviously, we need keep up with the competition.

This is what varies as the pizza types changes over time, we will have to modify this code over and over.

Example: Pizza Store (4)

Our Factory class for create all different types of objects.

```
public class PizzaFactory {
    public Pizza createPizza(String type) {
        Pizza p;

        if (type.equals("pepperoni")) {
            p = new PepperoniPizza();
        } else if (type.equals("cheese")) {
            p = new MargheritaPizza();
        } else if (type.equals("tomyum")) {
            p = new TomyumPizza();
        }
        return p;
}
```

- **Solution:** Define a Factory class that encapsulates the object creation for all pizzas. It has only one job in life: creating pizzas for its store to serve every client.

We define a createPizza() method in the factory. This is the method the store will use to instantiate new pizza objects.

Example: Pizza Store (5)


```
public class PizzaStore {
    PizzaFacotry pFactory;
    public PizzaStore(PizzaFacotry pFactory) {
        this.pFactory = pFactory;
    }
    public Pizza orderPizza(String type) {
        Pizza p;
        p = pFactory.createPizza(type);
        p.prepare();
        …
        return p;
    }
}
```

PizzaStore gets the factory passed to it in the constructor.

```
public Pizza orderPizza(String type) {
    Pizza p;
    if (type.equals("pepperoni")) {
        p = new PepperoniPizza();
    } else if (type.equals("hawaiian")) {
        p = new HawaiianPizza()
    } else if (type.equals("cheese")) {
        p = new MargheritaPizza();
    }
    p.prepare();
    …
    p.box();
    return p;
}
```

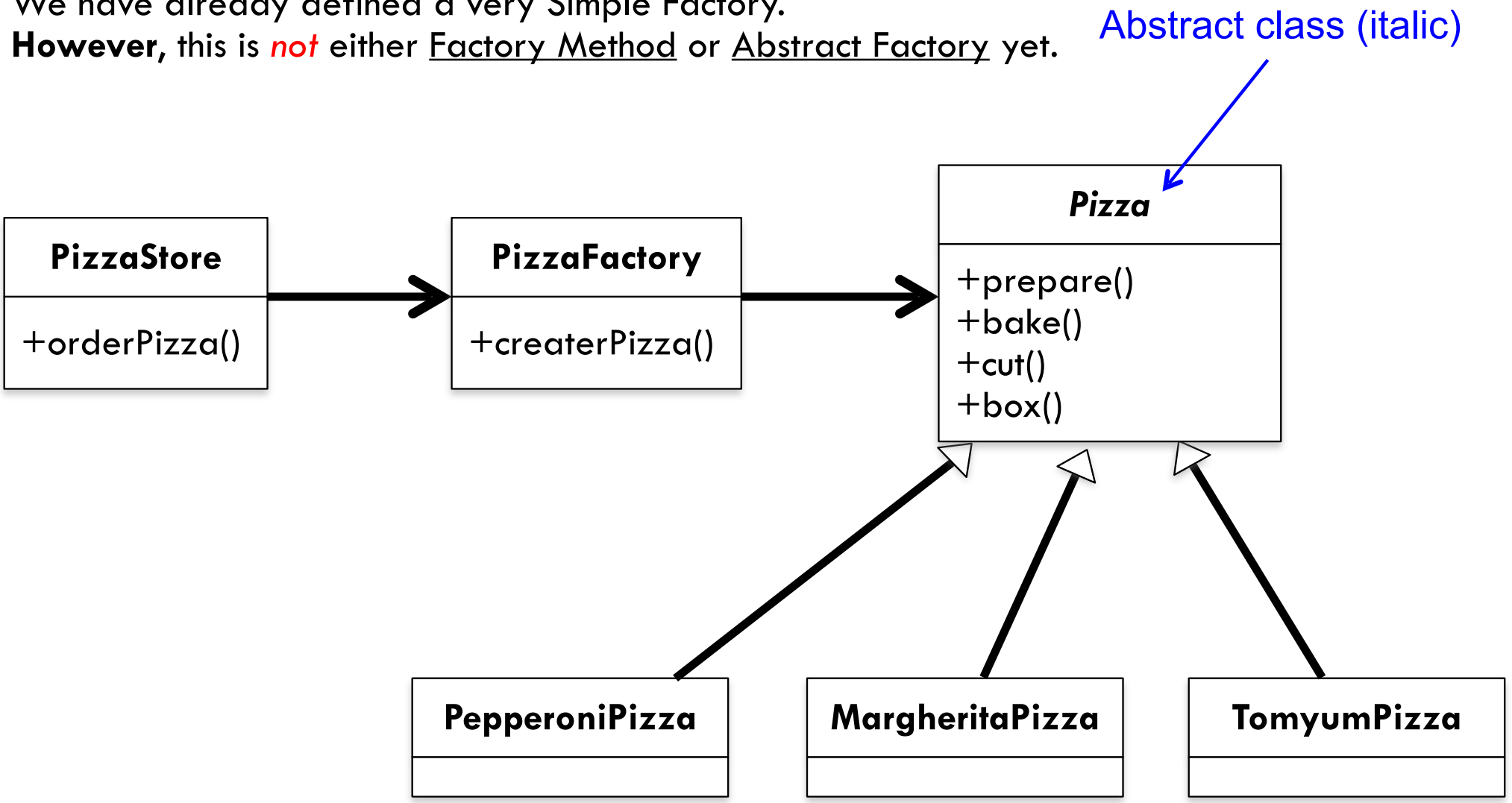
We use the createPizza() method on the PizzaFactory object to create pizza objects according to types.

Example: Pizza Store (6)

Simple Factory

We have already defined a very Simple Factory.

However, this is *not* either Factory Method or Abstract Factory yet.

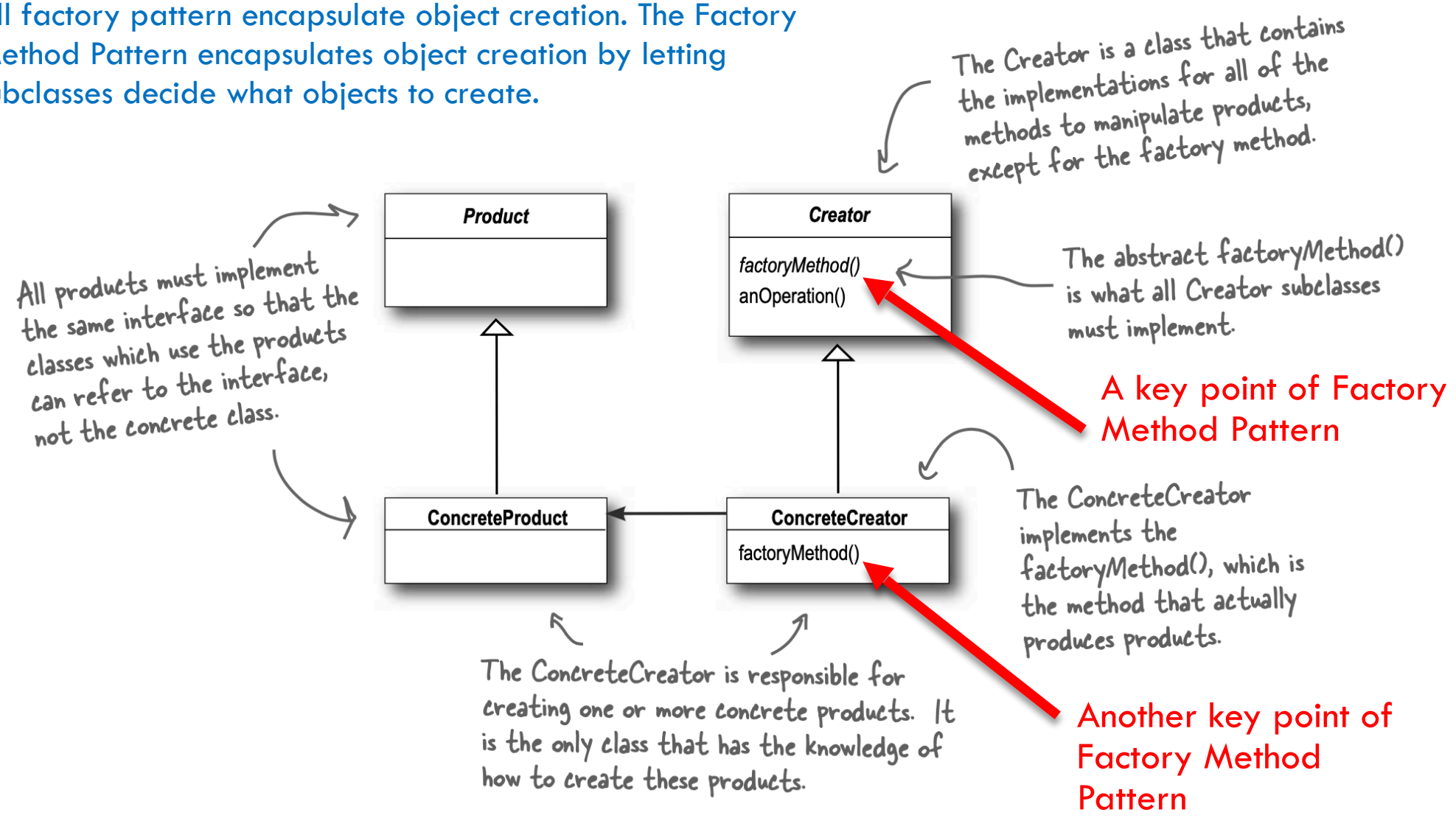


Factory Method Pattern

- **Problem:** How can one define an interface for creating an object, but let subclasses decide which class to instantiate.
- **Context:** The particular Document subclass to instantiate is application-specific, the Application class can't predict the subclass of Document to instantiate—the Application class only knows when a new document should be created, not what kind of Document to create. This creates a dilemma: The framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.
- **Solution:** It lets a class defer instantiation to subclasses. It encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework.

Factory Method Pattern - *Template*

All factory pattern encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create.



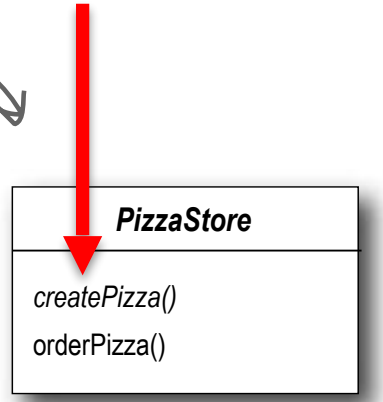
Creator Classes

Abstract Factory Method

```
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

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

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

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.




```
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;
    }
}
```

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

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

Classes that produce products are called concrete creators

Concrete Factory Method

Product Classes


```
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;
    }
}
```

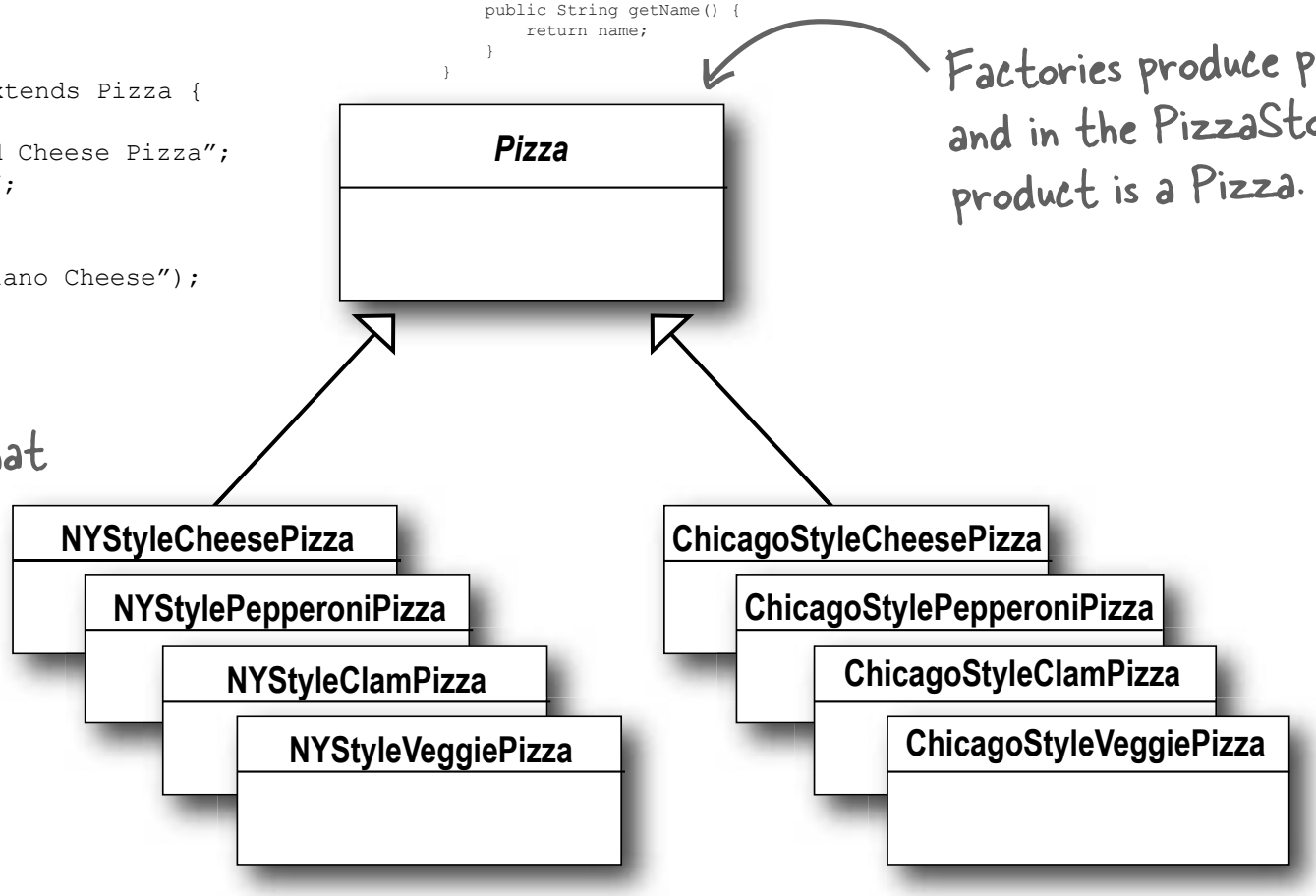
Each Pizza has a name, a type of dough, a type of sauce, and a set of toppings.

The abstract class provides some basic defaults for baking, cutting and boxing.

Preparation follows a number of steps in a particular sequence.

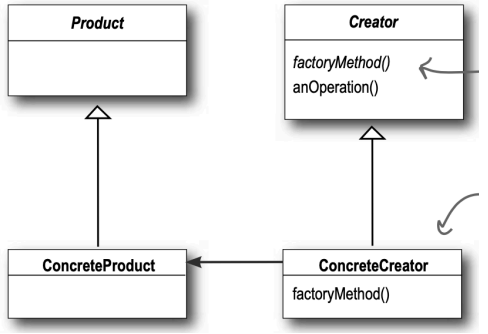
```
public class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
    }
}
```

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



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

Two parallel class hierarchies

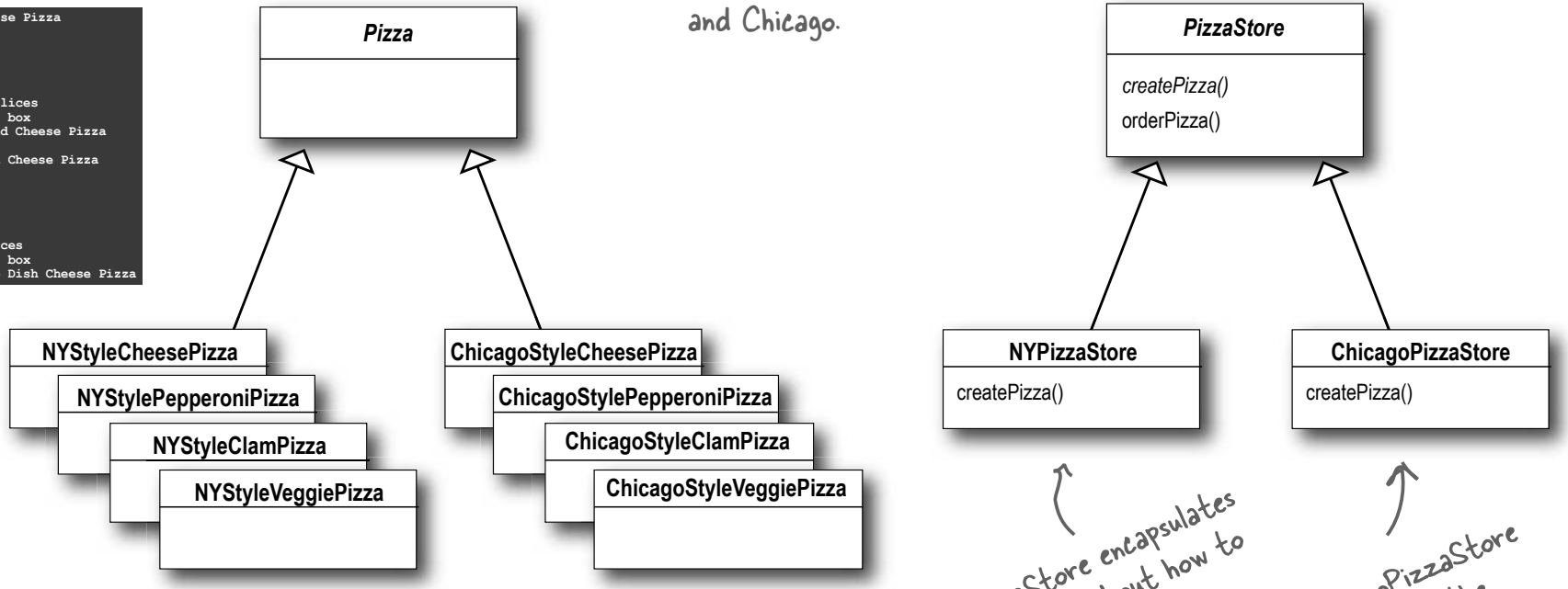


Notice how these class hierarchies are parallel: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago.

The Product classes

The Creator classes

```
Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Grated Regiano cheese
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza

Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```




```
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.

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

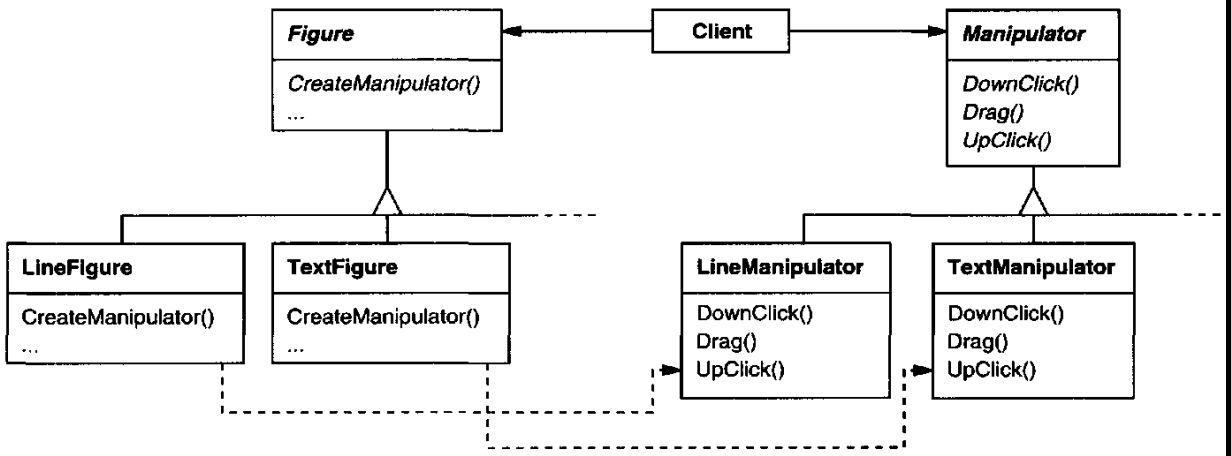
The ChicagoPizzaStore encapsulates all the knowledge about how to make Chicago style pizzas.

The factory method is the key to encapsulating this knowledge.

Factory Method Pattern: Consequences

- Eliminate the need to bind application-specific classes
- Provides hooks for subclasses
- Connects parallel class hierarchies

```java
// Abstract Figure class
abstract class Figure {
    // Factory method to create Manipulator
    abstract Manipulator createManipulator();
}

// Concrete Figure subclass representing a Line
class LineFigure extends Figure {
    @Override
    Manipulator createManipulator() {
        return new LineManipulator(); // Returns specific
    }
}
```

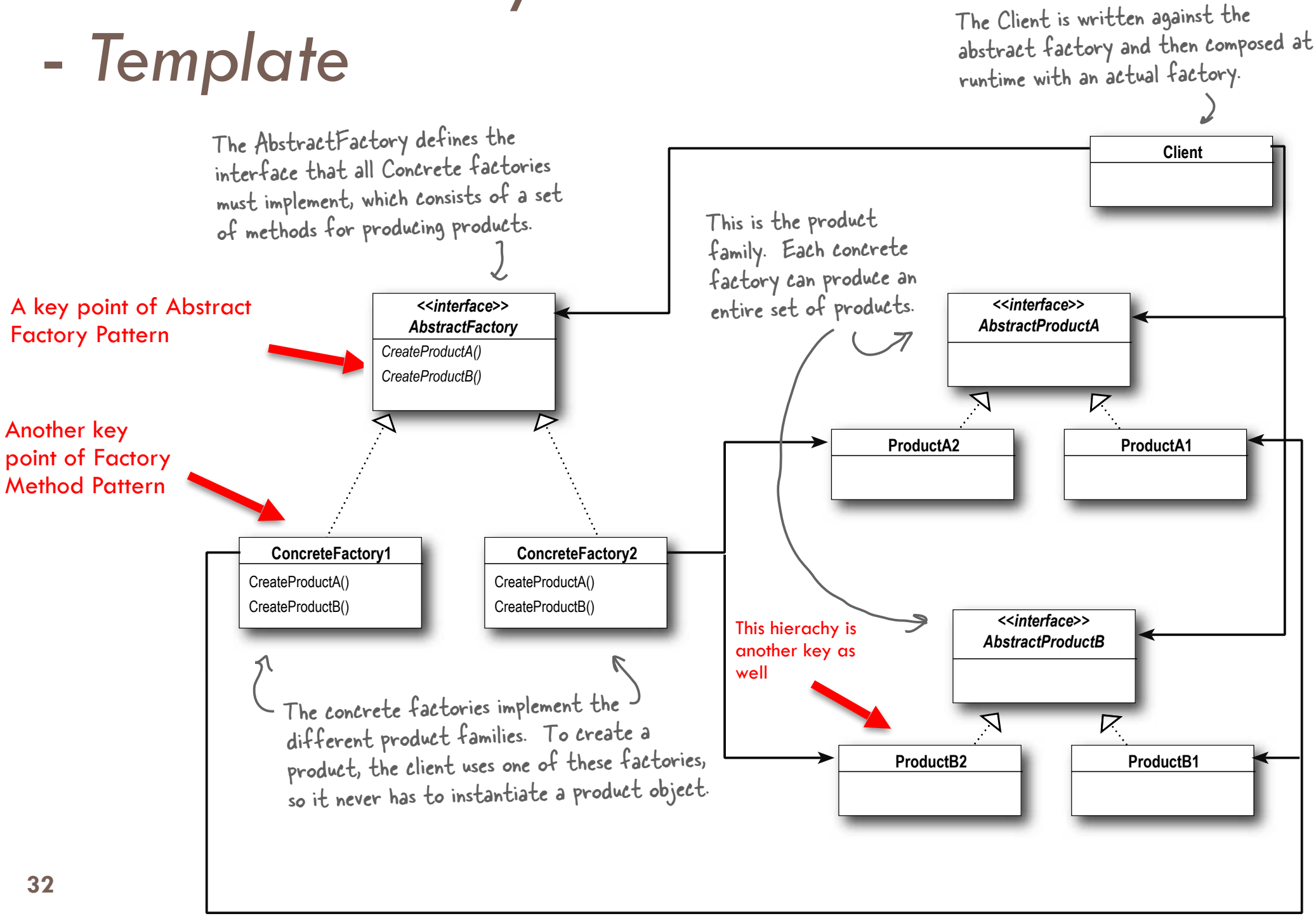



```java
// Client code
public class Main {
    public static void main(String[] args) {
        Figure lineFigure = new LineFigure();
        Figure textFigure = new TextFigure();

        // Draw figures
        lineFigure.draw();
        textFigure.draw();

        // Create manipulators and perform manipulations
        Manipulator lineManipulator = lineFigure.createManipulator();
        Manipulator textManipulator = textFigure.createManipulator();

        lineManipulator.manipulate();
        textManipulator.manipulate();
    }
}
```

Abstract Factory Pattern

- **Problem:** How to provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Context:** Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons. To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel later.

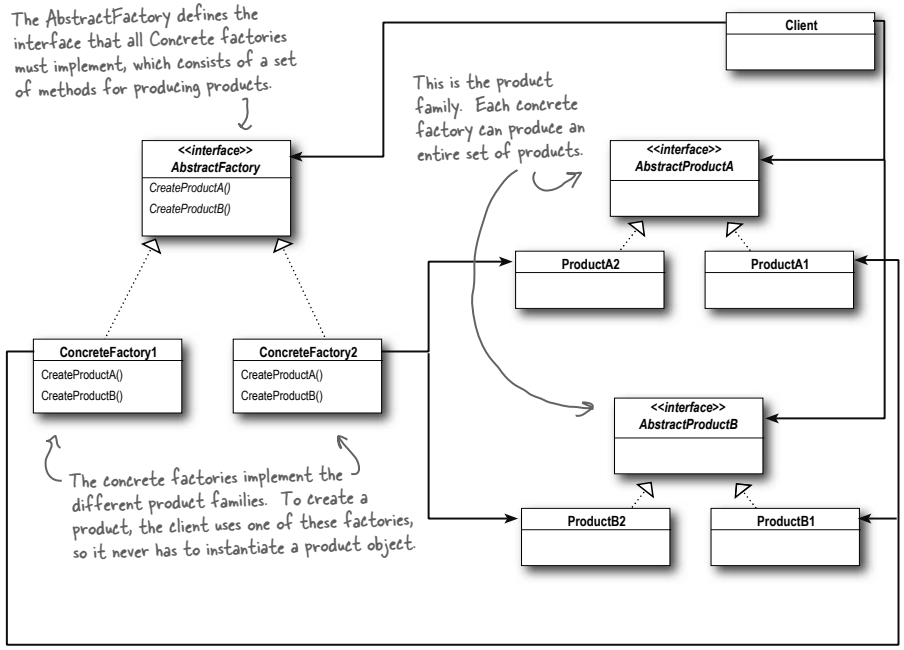
Abstract Factory Pattern - *Template*



Another problem about our PizzaStore

We are now franchising our Pizza Stores, but they are located in different regions. What is red sauce in New York is not red sauce in Chicago. So, you have one set of ingredients that need to be shipped to New York and a *different* set that needs to shipped to Chicago.

Ingredient Factory: Abstract Factory




```
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.

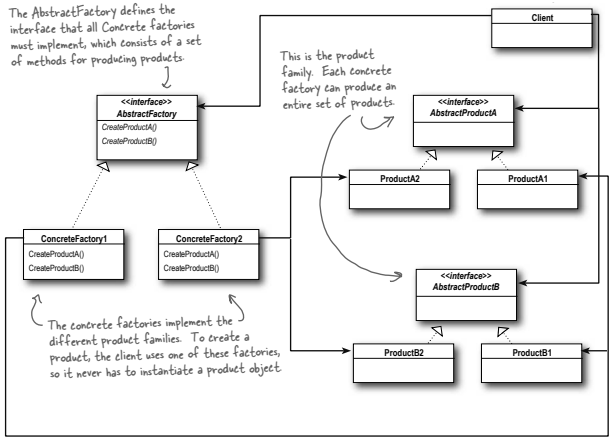
If we'd had some common "machinery" to implement in each instance of factory, we could have made this an abstract class instead...

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();
    }
}
```

The NY ingredient factory implements the interface for all ingredient factories

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. Chicago has to settle for frozen.

The best sliced pepperoni. This is shared between New York and Chicago. Make sure you use it on the next page when you get to implement the Chicago factory yourself

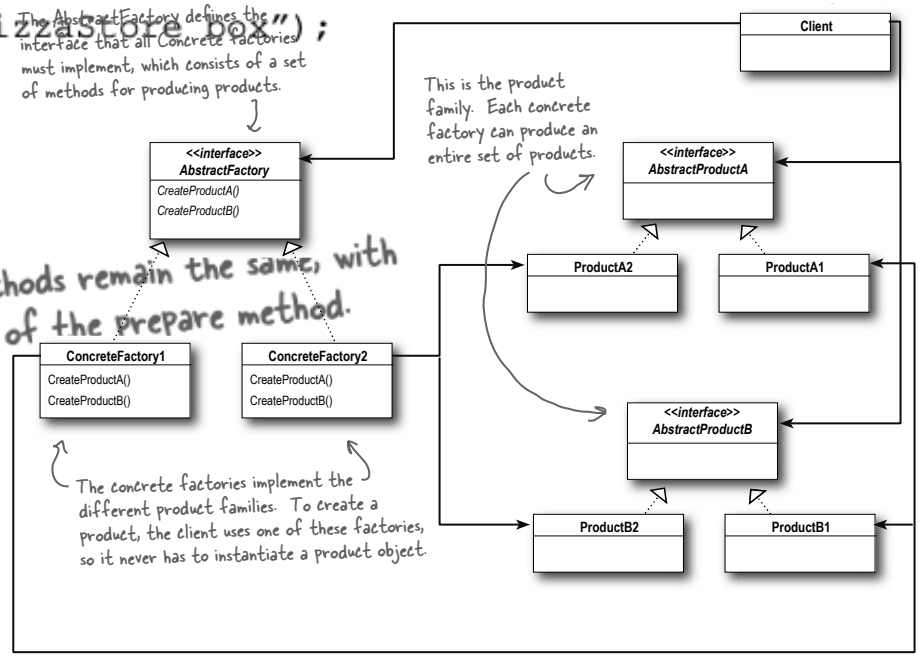
An abstract *Pizza* class

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

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

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
        // code to print pizza here
    }
}
```

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

We've now made the prepare method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory.

Our other methods remain the same, with the exception of the prepare method.



A concrete *Pizza* class

An instance of concrete factory of a specific city needs to be passed here.

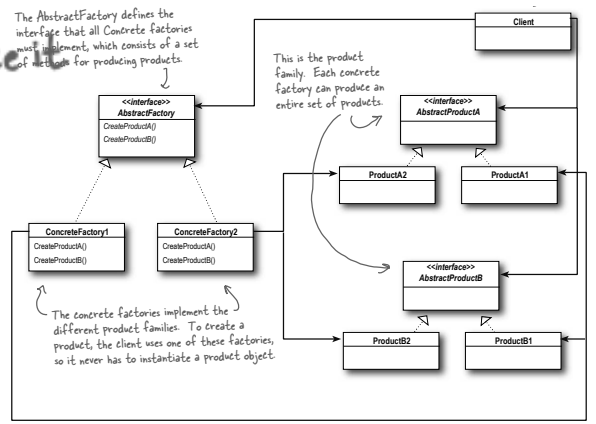
```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    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.

The low-level class does not need to know which factory (of which city) they are using to create ingredients.



Another concrete *Pizza* class

Just as that of the cheese pizza, an instance of concrete factory of a specific city needs to be passed here for a clam pizza.

```
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.

Again, the low-level class does not need to know which factory (of which city) they are using to create ingredients.

New York Pizza Store: a concrete PizzaStore class

```
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.

```
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

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

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

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

Look back one page and make sure you understand how the pizza and the factory work together!

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

That's a fairly complicated class diagram; let's look at it all in terms of our PizzaStore:

Complete an *Abstract* Factory Pattern with our Pizza Problem.

1 First we need a NY PizzaStore:

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

2 Now that we have a store, we can take an order:

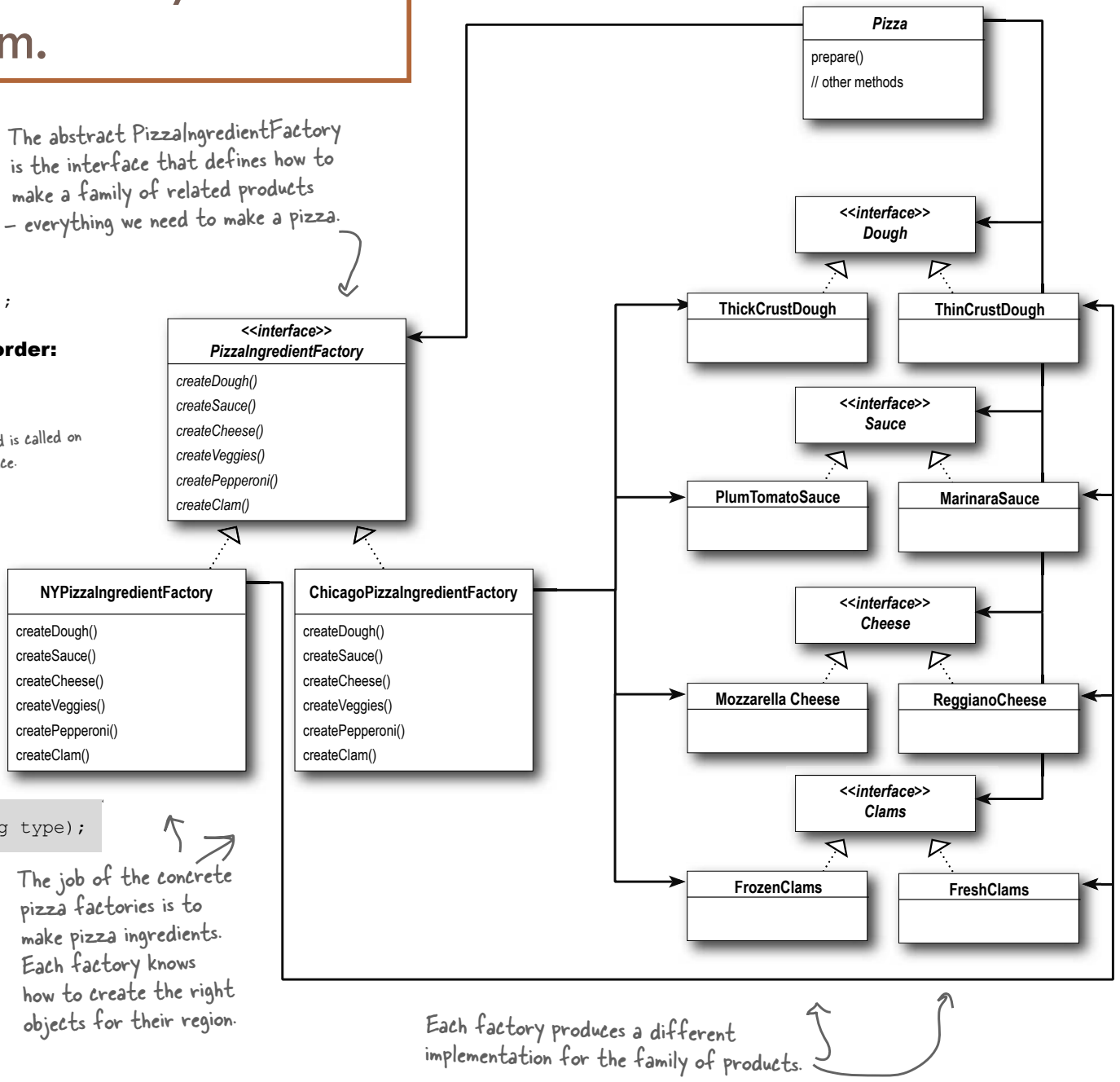
```
nyPizzaStore.orderPizza("cheese");
```

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

```
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

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

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

The clients of the Abstract Factory are the concrete instances of the Pizza abstract class.

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

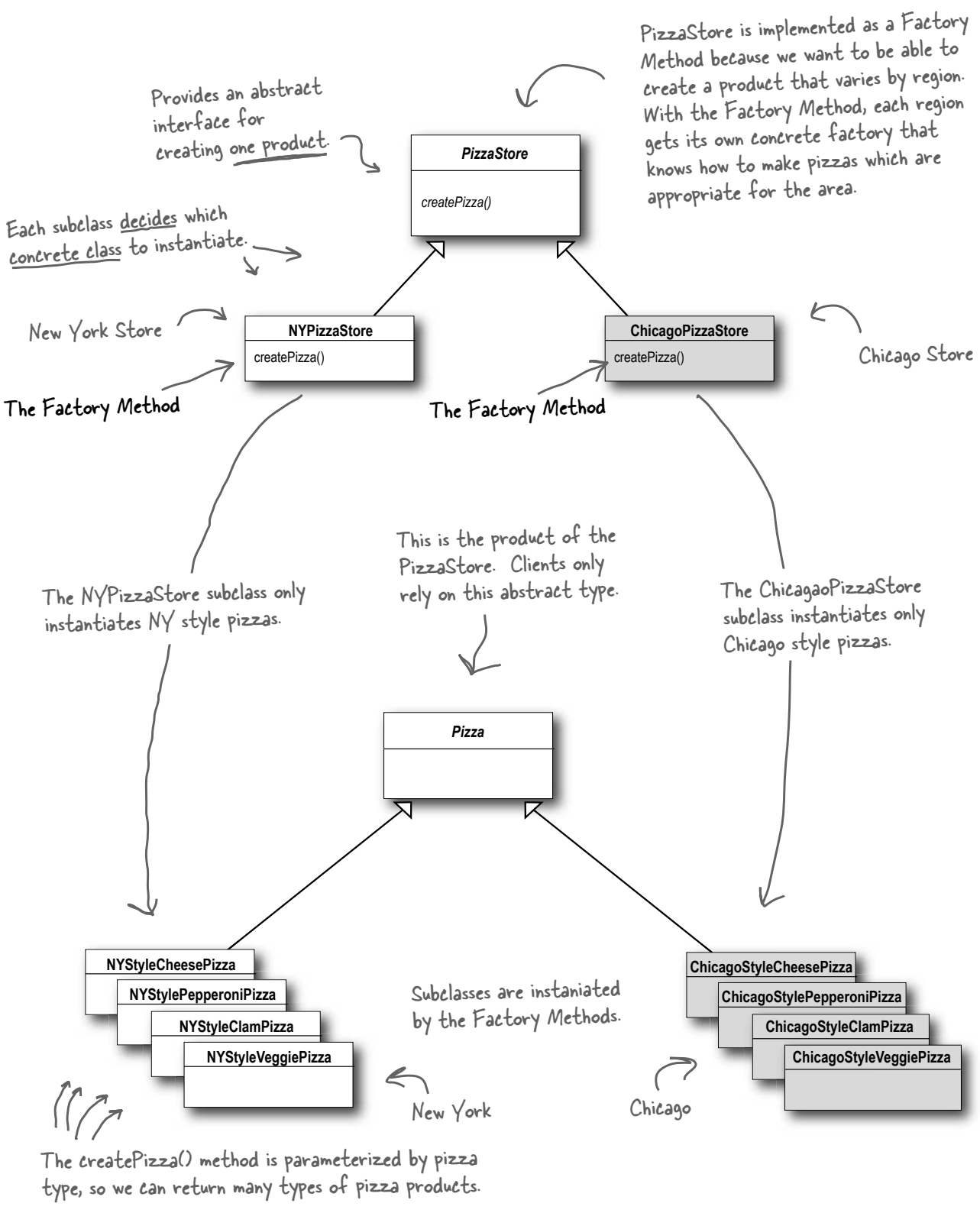


The job of the concrete pizza factories is to make pizza ingredients. Each factory knows how to create the right objects for their region.

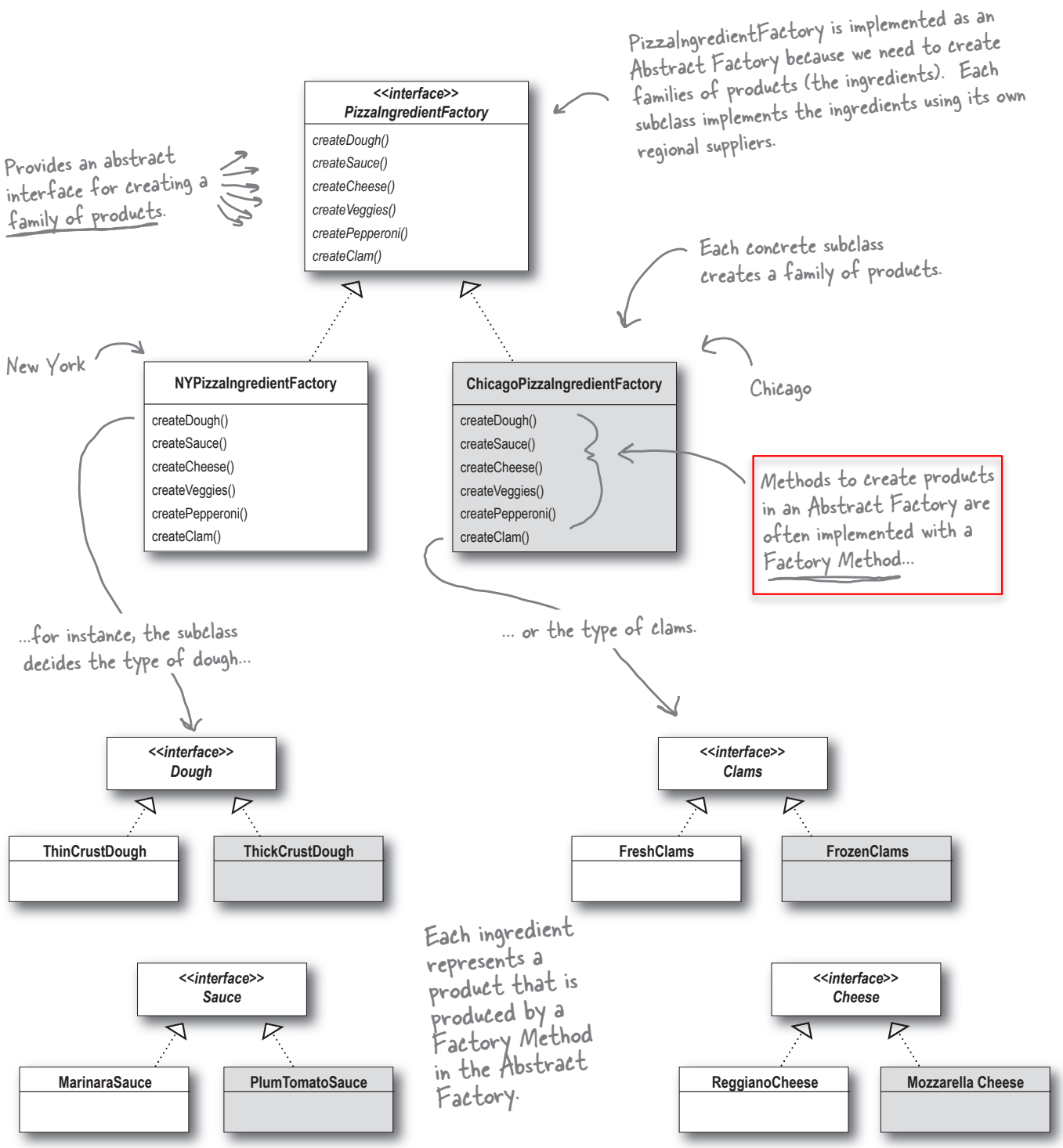
Each factory produces a different implementation for the family of products.

Compare Factory Method with Abstract Factory

Here is the Factory Method.

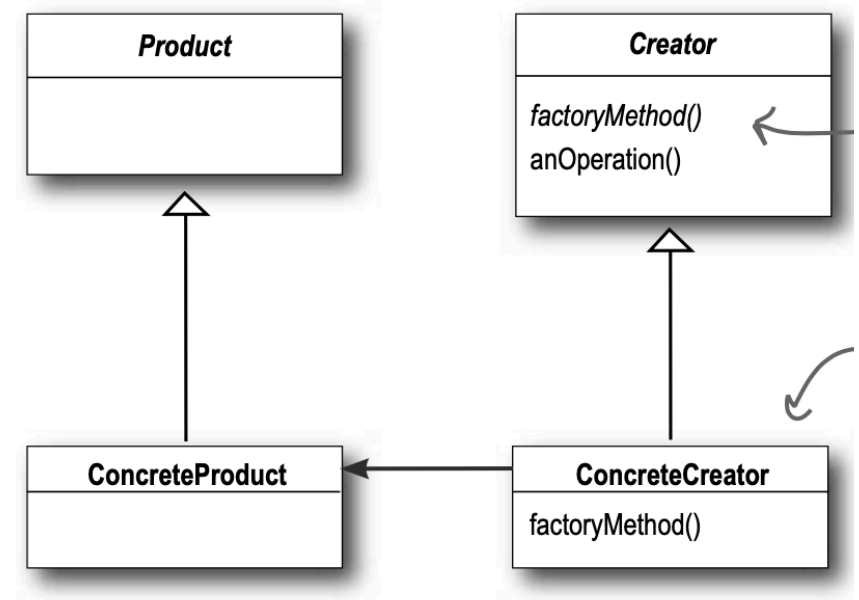


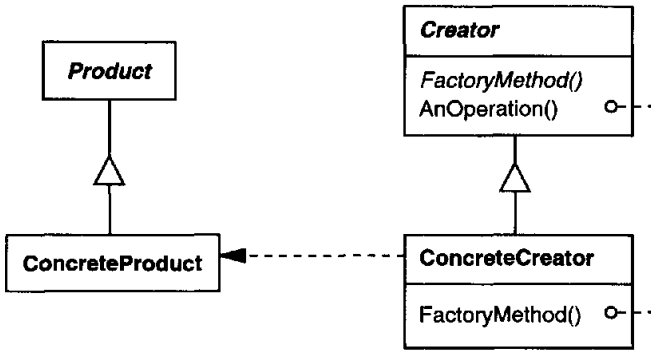
And, here is the Abstract Factory

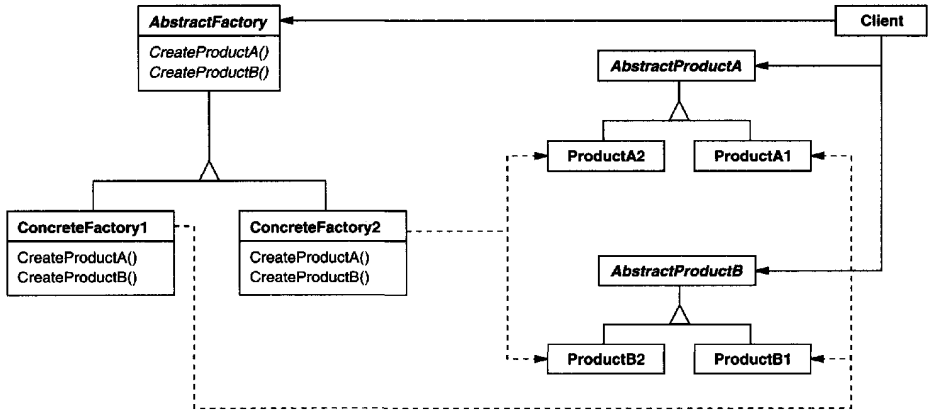


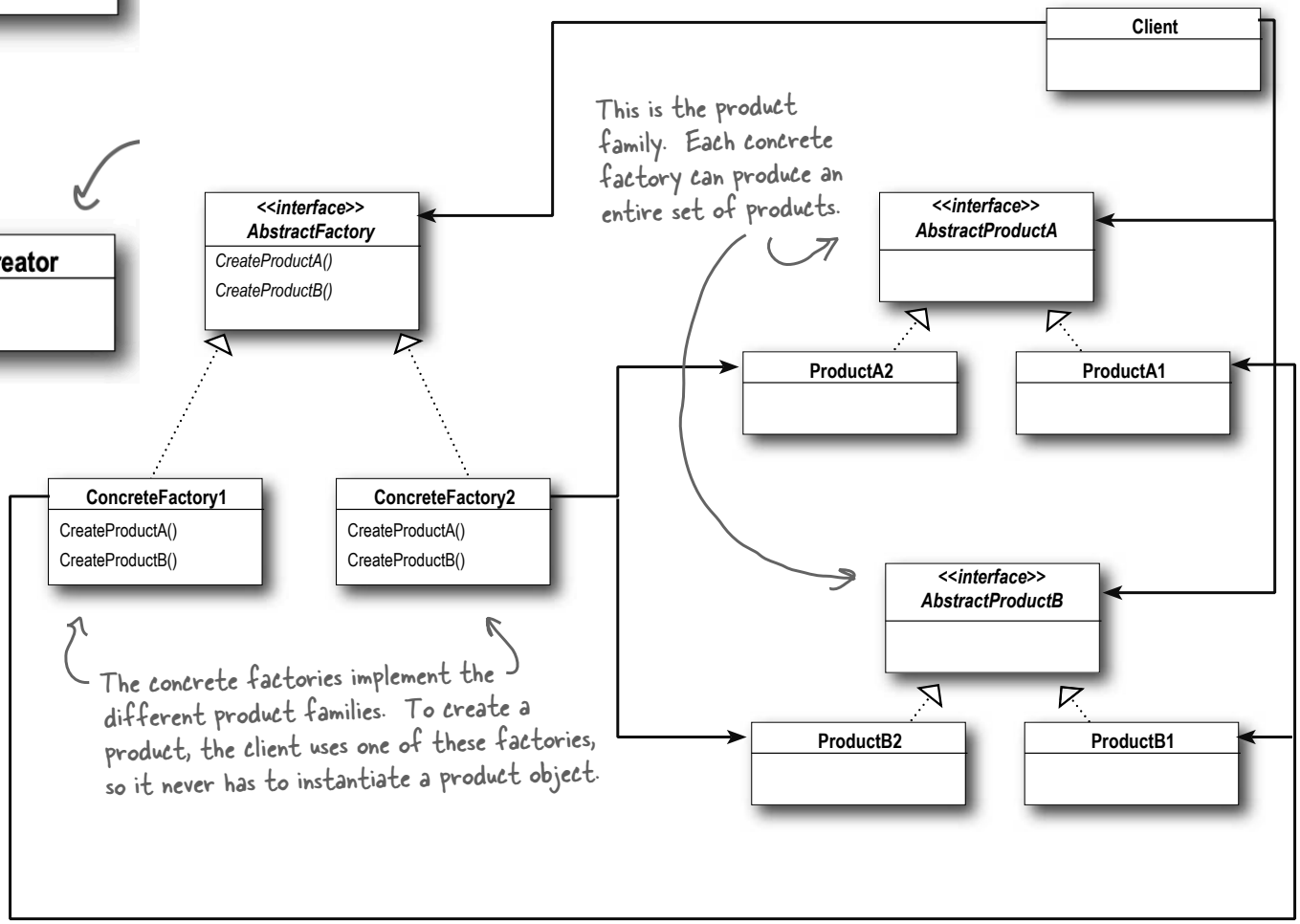
The product subclasses create parallel sets of product families. Here we have a New York ingredient family and a Chicago family.

Factory Method vs. Abstract Factory









Abstract Factory Pattern: Consequences

- It isolates concrete classes
- It makes exchanging product families easy
- It promotes consistency among products
- Supporting new kinds of products is difficult