

# Factory Pattern

# Factory Patterns

- Simple Factory
- Factory Method
- Abstract Factory

# New is an implementation

- Calling “new” is certainly coding to an implementation
- In fact, it’s always related to a concrete class
- (Open for Modification)
- That’s fine when things are simple, but . . .

# Look Out For Change

- When you have several related classes, that's probably a good sign that they might change in the future

```
Duck duck;  
if (picnic) {  
    duck = new MallardDuck();  
} else if (hunting) {  
    duck = new DecoyDuck();  
} else if (inBathTub) {  
    duck = new RubberDucky();  
}
```

This type of code will often lead to problems when new types have to be added.

# Instantiating Concrete Classes

- Using new instantiates a concrete class.
- This is programming to implementation instead of interface.
- Concrete classes are often instantiated in more than one place.
- Thus, when changes or extensions are made all the instantiations will have to be changed.
- Such extensions can result in updates being more difficult and error-prone.

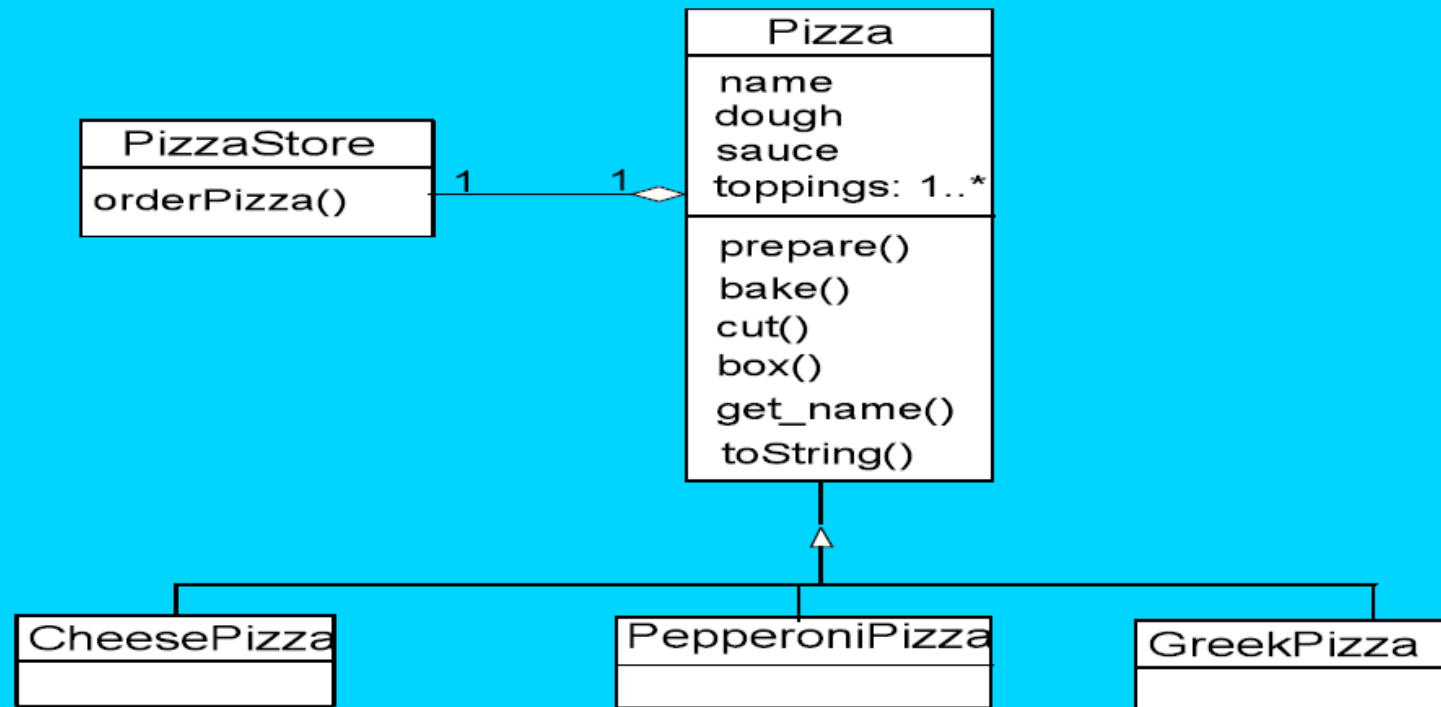
# PizzaStore in Objectville

- Suppose that you are required to develop a system that accepts orders for pizzas.
- There are three types of pizzas, namely,
  - cheese,
  - Greek, and
  - pepperoni.
- The pizzas differ according to the dough used, the sauce used and the toppings.

# PizzaStore in Objectville

- During the ordering of a Pizza, we need to perform certain actions on it:
  - Prepare
  - Bake
  - Cut
  - Box
- We know that all Pizzas *must* perform these behaviors. In addition, we know that these behaviors will not change during runtime. (i.e. the Baking time for a Cheese Pizza will never change!)
- **Question:** Should these behaviors (prepare, bake, etc) be represented using Inheritance or Composition?

# Example: Class Diagram





# PizzaStore in Objectville (pg 112 top)

```
public Pizza orderPizza(String type) {  
  
    Pizza pizza = new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
  
    return pizza;  
}
```

This method is responsible for creating the pizza.

It calls methods to prepare, bake, etc.

Pizza is returned to caller.

- Creating an instance of Pizza() doesn't make sense here because we know there are different types of Pizza.

# PizzaStore in Objectville (pg 112 middle)

A parameter  
indicating type

```
public Pizza orderPizza(String type) {
```

```
    if (type.equals("cheese")) {
```

```
        pizza = new CheesePizza();
```

```
    } else if (type.equals("pepperoni")) {
```

```
        pizza = new PepperoniPizza();
```

Code that varies

```
    pizza.prepare();
```

```
    pizza.bake();
```

```
    pizza.cut();
```

```
    pizza.box();
```

```
    return pizza;
```

```
}
```

Code that stays  
the same

# Identifying the aspects that vary

- If the pizza shop decides to change the types of pizza it offers, the orderPizza method has to be changed.

# Identifying the aspects that vary

```
Pizza orderPizza(String type) {  
    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()  
}
```

Part  
that  
varies.

Part that  
remains  
constant

# Revised System

- Suppose that the Greek pizza is not popular and must be removed and two new pizzas, Clam and Veggie, must be added to the menu.
- Programming to implementation makes such changes difficult.
- Creating a SimpleFactory to encapsulate the code that changes will make the design more flexible.
- Remove the code that creates a pizza – forms a factory.

# Building a simple pizza factory

```
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;  
    }  
}
```

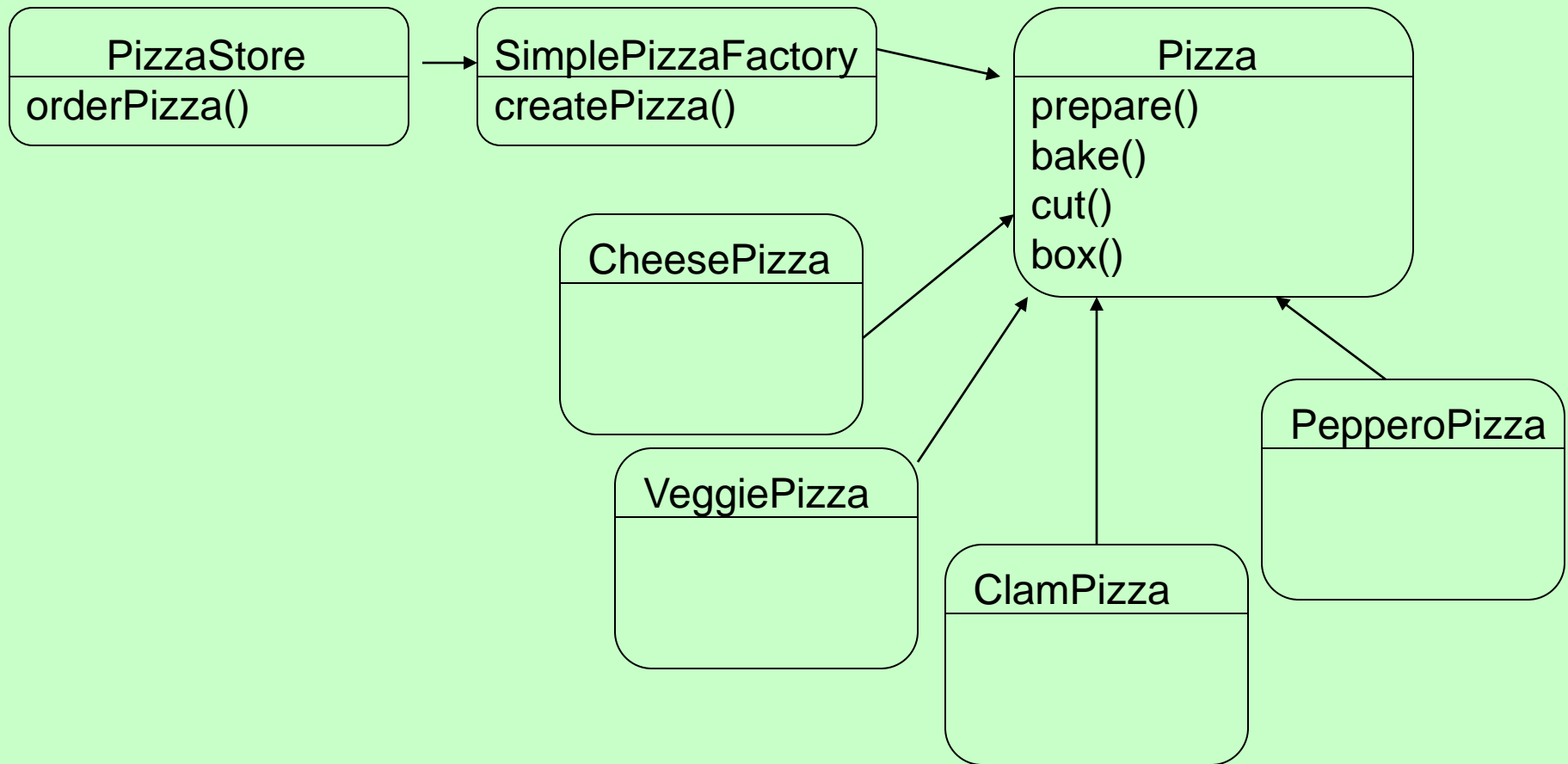
# Rework of PizzaStore (pg 118)

```
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;  
    }  
}
```

Store is *composed*  
of a factory.

Creation of pizza is  
delegated to factory.

# Simple Factory Defined





# Complete example for Simple Factory

```
SimplePizzaFactory factory = new SimplePizzaFactory();  
PizzaStore store = new PizzaStore(factory);  
Pizza pizza = store.orderPizza("cheese");
```

# Simple Factory

- Pull the code that builds the instances out and put it into a separate class
  - Identify the aspects of your application that vary and separate them from what stays the same

# SimplePizzaFactory

- Advantage: We have one place to go to add a new pizza.
  - Multiple clients needing same types of object
  - Ensure consistent object initialization

# Simple Factory Defined (pg 119)

- This is not an official pattern, but it is commonly used.
- Not a bad place to start.
- When people think of “Factory”, they may actually be thinking of this.
  - Important: In a team discussion, you may need to define your vocabulary.

# Change Now Occurs...

- The PizzaStore is very popular and it needs to be franchised.
  - New York is interested
  - Chicago is interested
  - And perhaps one day Fairfax...
- Since PizzaStore is already composed of a Simple Factory, then this should be easy! Let's just create PizzaStore with a different SimpleFactory.

# Example: System Revision Again

Franchises in different parts of the country are now adding their own special touches to the pizza. For example,

- customers at the franchise in New York like a thin base, with tasty sauce and little cheese.
- However, customers in Chicago prefer a thick base, rich sauce and a lot of cheese.
- Some franchises also cut the pizza slices differently (e.g. square)

You need to extend the system.

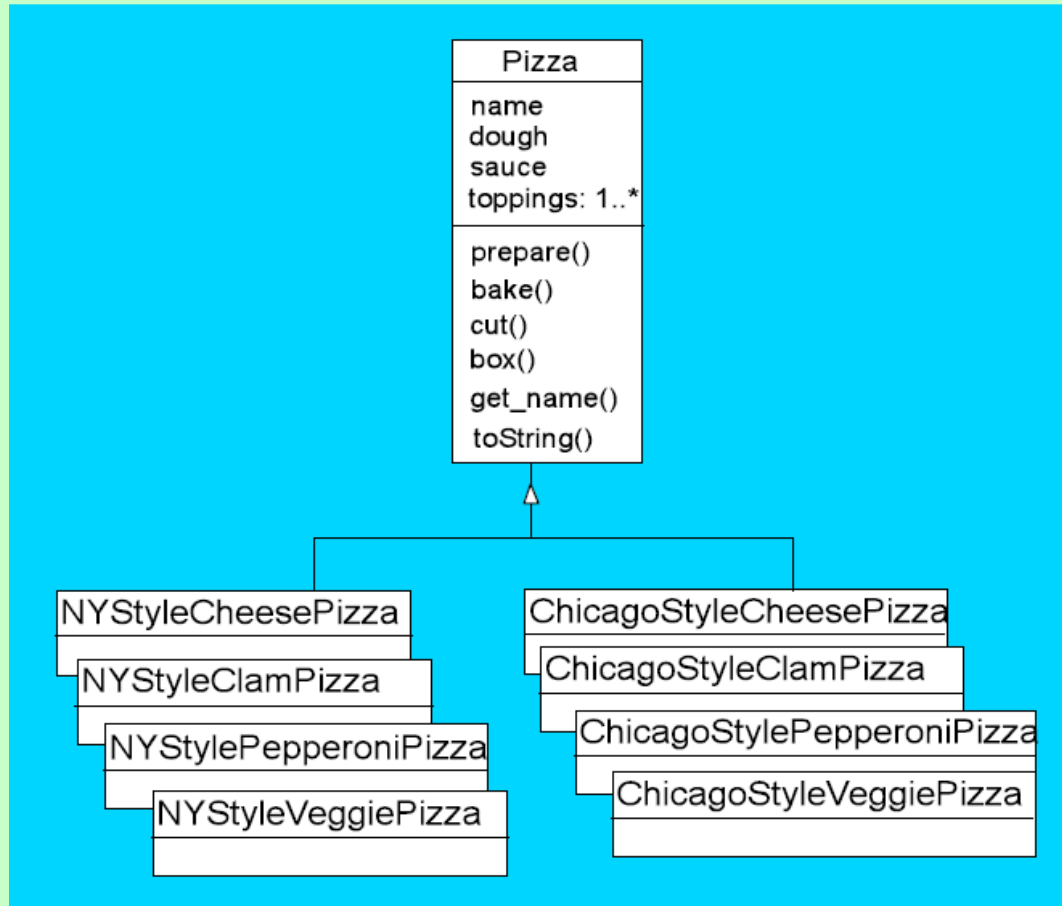
# Example (pg 119)

```
//NY Pizza Factory has a different if/else logic
NYPizzaFactory nyFactory = new NYPizzaFactory();

//Create the Pizza Store, but use this Simple Factory
//instead
PizzaStore nyStore = new PizzaStore(nyFactory);

//Order pizza
nyStore.order("Veggie");
```

# Pizza Class





# More change happens...

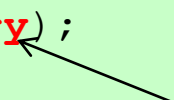
New York likes the PizzaStore, but they want to add more functionality, such as baking things a little differently or schedule a delivery.

New York attempts to extend PizzaStore...

```
public class NYPizzaStore extends PizzaStore {  
  
    public void ScheduleDelivery() {  
        ...  
    }  
}
```

# More change happens... (cont)

```
NYPizzaFactory nyFactory = new NYPizzaFactory();  
NYPizzaStore nyStore = new NYPizzaStore(nyFactory);  
  
//Order pizza  
nyStore.order("Veggie");  
nyStore.ScheduleDelivery();
```



## New York says the following:

- We only have one way to create pizzas; therefore, we don't *need* to use composition for the pizza creation.
- We are not happy that we have to create our extended Pizza store and create a unique factory for creating pizzas. These two classes have a one-to-one relationship with each other. Can't they be combined??

# What New York wants

A framework so that NY can do the following:

- Create pizzas in a NY style
- Add additional functionality that is applicable to NY only.

# A Framework for Pizza Store

## (pg 120)

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

NOTE: We are using *inheritance* here to create the pizzas, not composition. Also, the constructor for PizzaStore has been removed.

# NYPizzaStore (pg 123)

```
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;  
    }  
  
    void ScheduleDelivery();  
}
```

The subclass is defining how the pizza is created....and it is also providing unique functionality that is applicable to New York.

# Factory Method

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

Factory Method simply sets up an interface for creating a Product (in this case, a type of Pizza). Subclasses decide which specific Product to create.

# Test Drive (pg 127)

- 1) `PizzaStore nyStore = new NYPizzaStore();`
- 2) `nyStore.orderPizza("cheese");`
- 3) Calls `createPizza("cheese")`. The `NYPizzaStore` version of `createPizza` is called. It returns a `NYStyleCheesePizza()`;
- 4) `orderPizza` continues to call the "parts that do not vary", such as `prepare`, `bake`, `cut`, `box`.

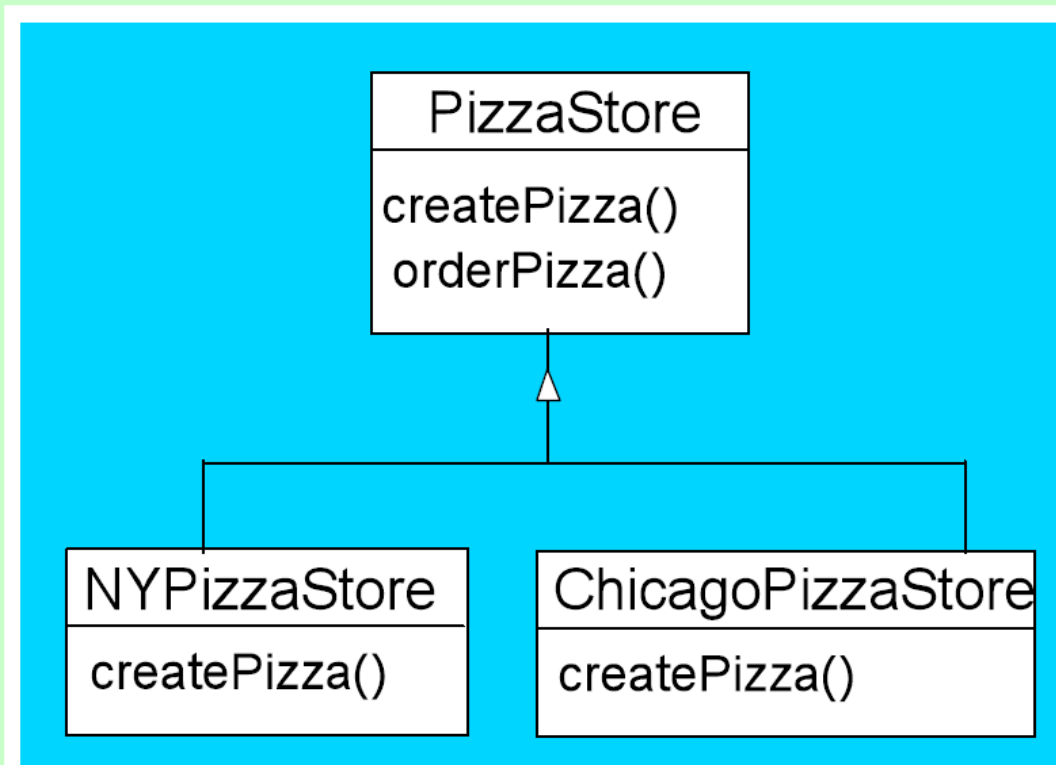
# The Factory Method

Definition:

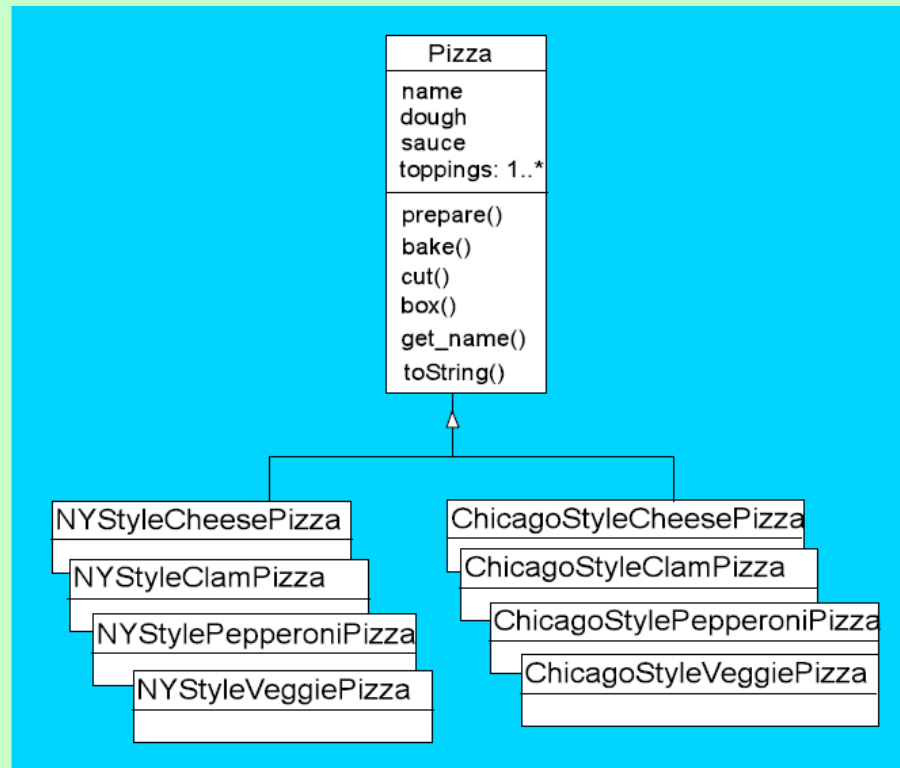
- The factory Method Pattern defines an interface for creating an object, but lets the subclasses decide which class to instantiate.



# Creator Class

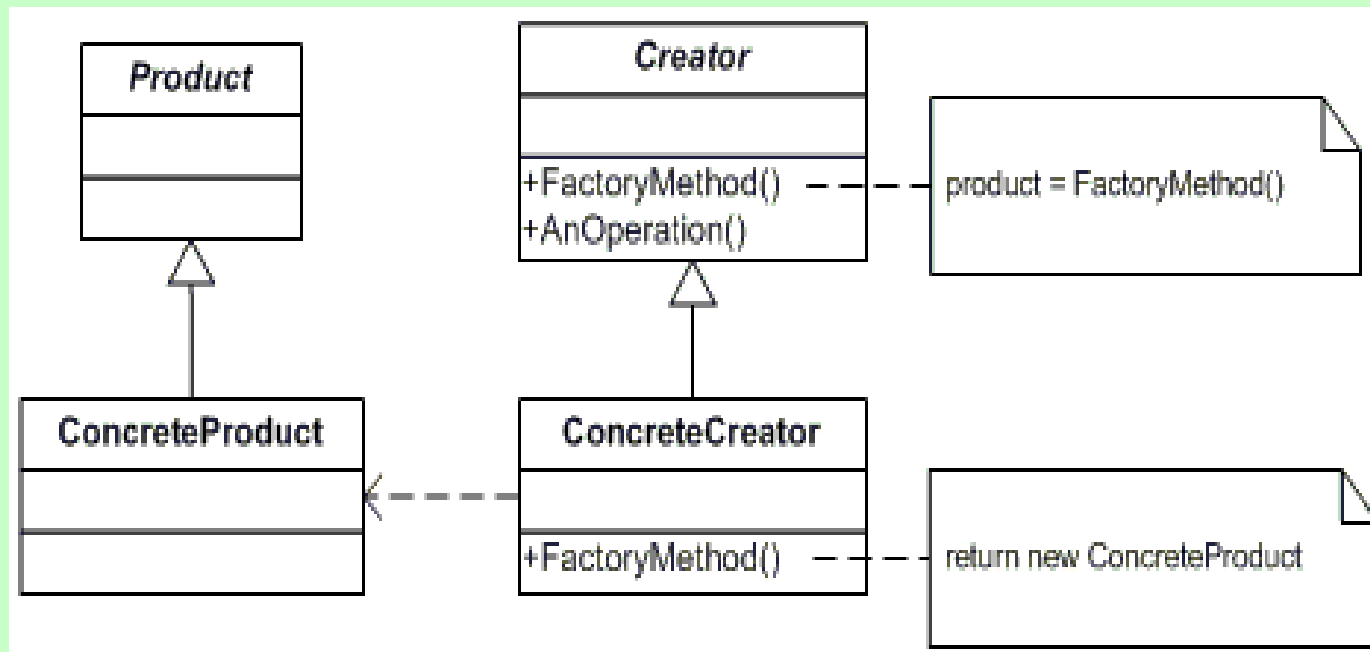


# Product Class



# Factory Method Defined

GoF Intent: “Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.”



# Definitions (pg 134)

- Product - abstraction of the object that we need to create (Pizza)
- Concrete Product – implementations of a specific Product (NYStyleCheesePizza)
- Creator – abstraction of the object that will create a Product (PizzaStore). Contains factory method.
- Concrete Creator – implementation of a Creator that creates a specific ConcreteProduct (NYPizzaStore)

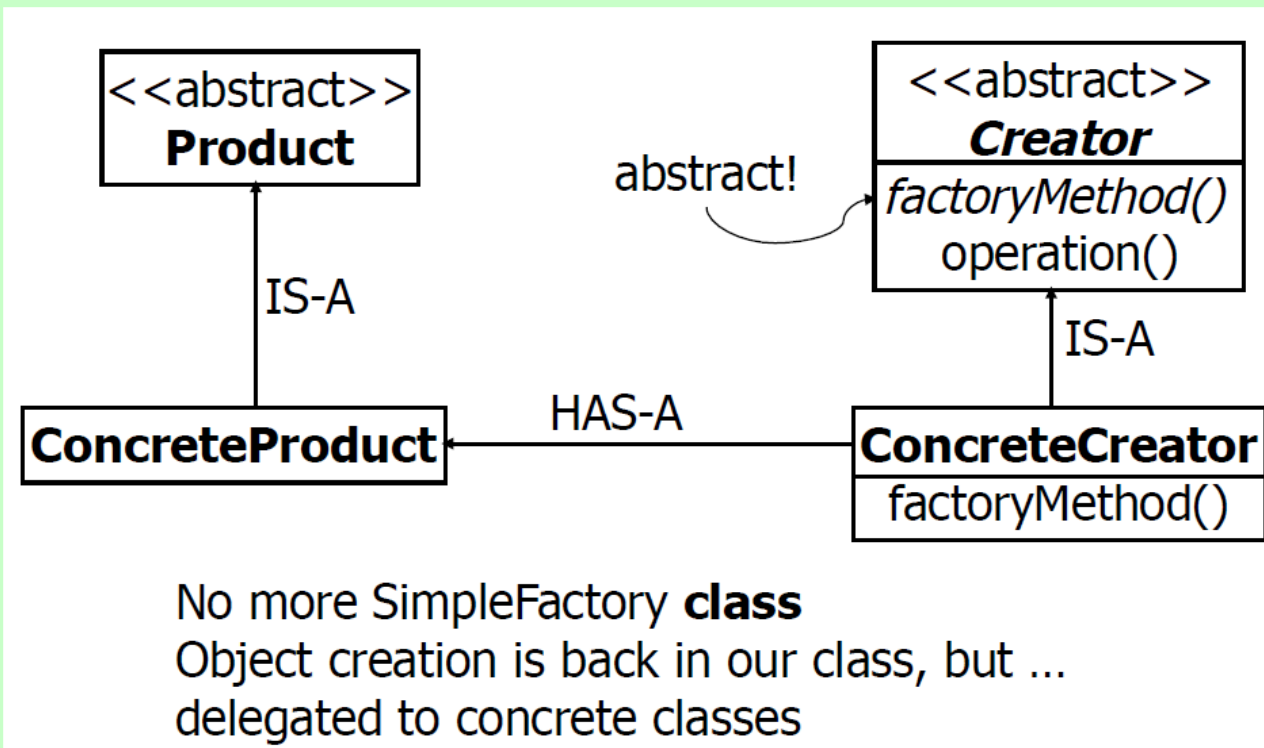
# IMPORTANT

The Factory Method should just return a ConcreteProduct.

# NYPizzaStore

```
public class NYPizzaStore extends
    PizzaStore{
    Pizza createPizza(String item) {
        if (type.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (type.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else if (type.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else return null;
    }
}
```

# NYPizzaStore



# Summary

- ***Pattern Name*** – Factory Method
- ***Problem*** – Do not want our framework to be tied to a specific implementation of an object.
- ***Solution***
  - Let subclass decide which implementation to use (via use of an abstract method)
  - Tie the Creator with a specific Product Implementation



```
public class DependentPizzaStore {
```

```
    public Pizza createPizza(String style, String type) {
```

```
        Pizza pizza = null;
```

```
        if (style.equals("NY")) {
```

```
            if (type.equals("cheese")) {
```

```
                pizza = new NYStyleCheesePizza();
```

```
            } else if (type.equals("veggie")) {
```

```
                pizza = new NYStyleVeggiePizza();
```

```
            } else if (type.equals("clam")) {
```

```
                pizza = new NYStyleClamPizza();
```

```
            } else if (type.equals("pepperoni")) {
```

```
                pizza = new NYStylePepperoniPizza();
```

```
            }
```

```
        } else if (style.equals("Chicago")) {
```

```
            if (type.equals("cheese")) {
```

```
                pizza = new ChicagoStyleCheesePizza();
```

```
            } else if (type.equals("veggie")) {
```

```
                pizza = new ChicagoStyleVeggiePizza();
```

```
            } else if (type.equals("clam")) {
```

```
                pizza = new ChicagoStyleClamPizza();
```

```
            } else if (type.equals("pepperoni")) {
```

```
                pizza = new ChicagoStylePepperoniPizza();
```

```
            }
```

```
        } else {
```

```
            System.out.println("Error: invalid type of pizza");
```

```
            return null;
```

```
        }
```

```
        pizza.prepare();
```

```
        pizza.bake();
```

```
        pizza.cut();
```

```
        pizza.box();
```

```
        return pizza;
```

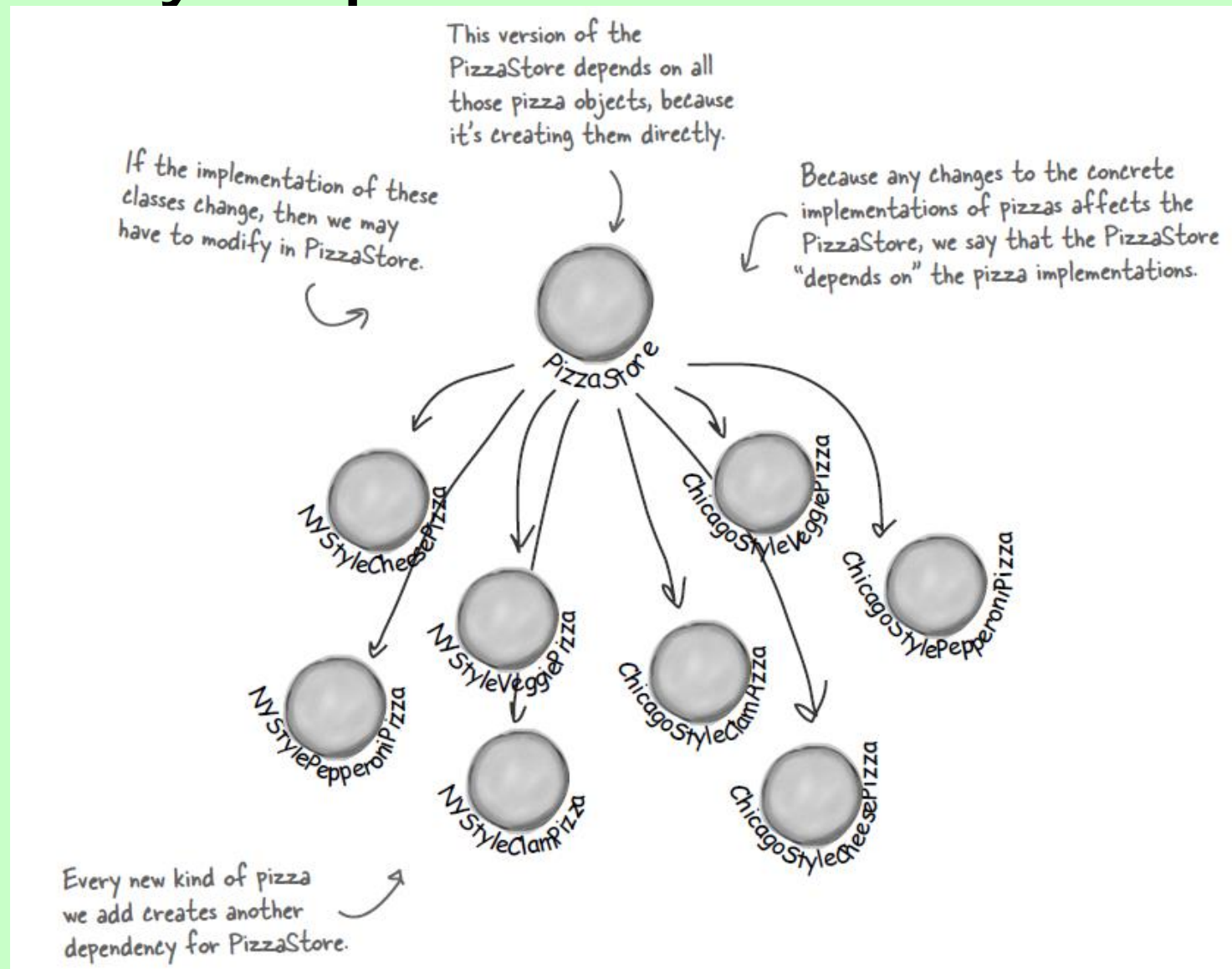
```
    }
```

```
}
```

Handles all the NY  
style pizzas

Handles all the  
Chicago style  
pizzas

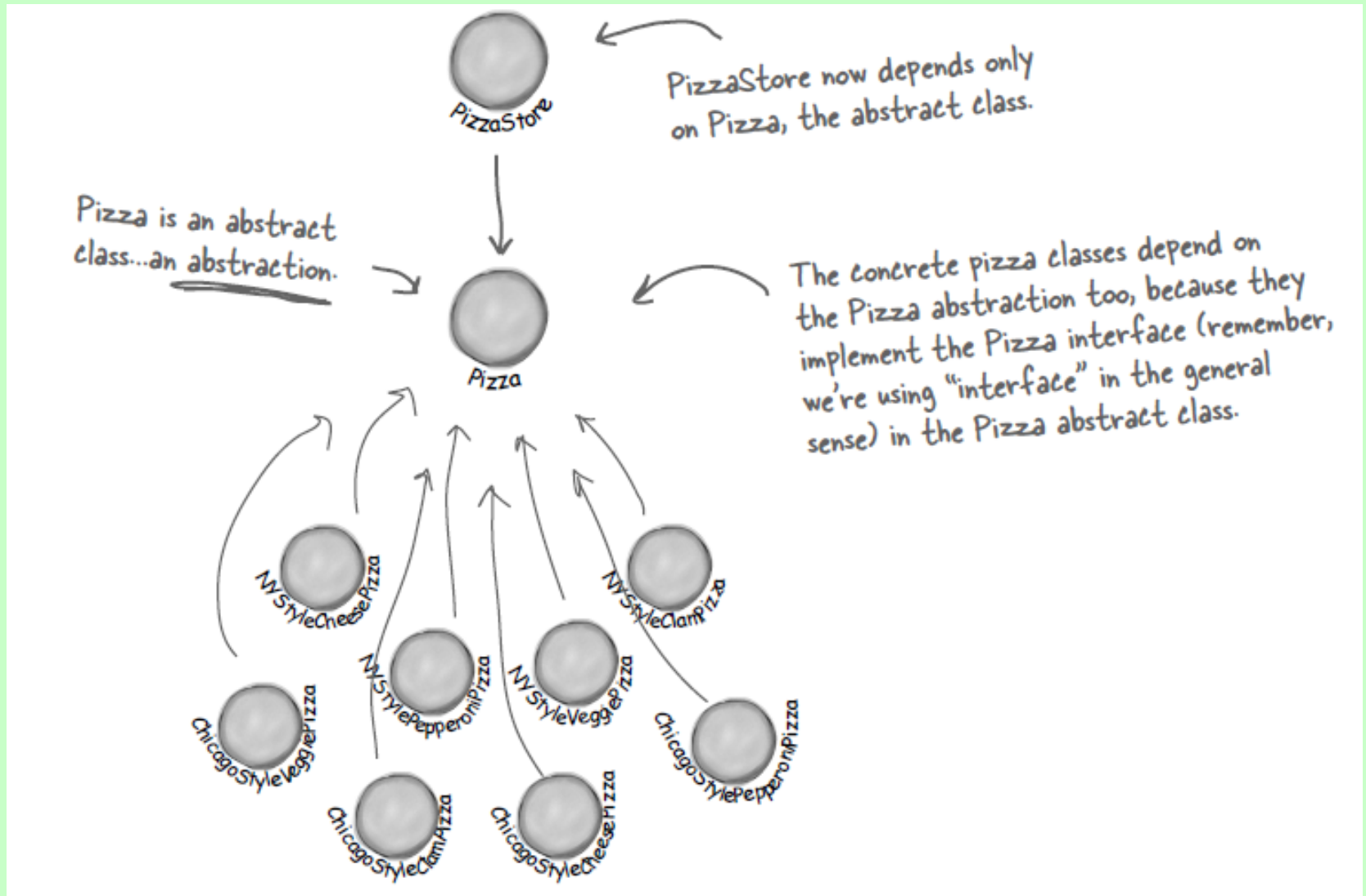
# A very dependent PizzaStore



# Dependency Inversion Principle

- Depend upon abstractions. Do not depend upon concrete classes.
  - Our high-level components should not depend on our low-level components; rather, they should both depend on abstractions.

# Dependency Inversion Principle



# Changes..

- Objectville Pizza HQ gets complaints that the different franchises are not using quality ingredients. They decide that all Pizzas must be composed of the following ingredients:
  - Dough, Sauce, Cheese, Veggies, Pepperoni, and Clams.
- New York says those ingredients are fine, but they want to use their own regional ingredients for all types of pizza, for example
  - Dough = Thin Crust Dough
  - Sauce = Marinara
  - Cheese = Reggiano
  - Veggies = Garlic, Onion

New York uses one set of ingredients and Chicago another. Given the popularity of Objectville Pizza it won't be long before you also need to ship another set of regional ingredients to California, and what's next? Seattle?

For this to work, you are going to have to figure out how to handle families of ingredients.

## Chicago

FrozenClams

PlumTomatoSauce

ThickCrustDough

MozzarellaCheese

## New York

FreshClams

MarinaraSauce

ThinCrustDough

ReggianoCheese

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

## California

Calamari

BruschettaSauce

VeryThinCrust

GoatCheese

Each family consists of a type of dough, a type of sauce, a type of cheese, and a seafood topping (along with a few more we haven't shown, like veggies and spices).

In total, these three regions make up ingredient families, with each region implementing a complete family of ingredients.

# Families of Ingredients



## 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

We've got the  
same product  
families (dough,  
sauce, cheese,  
veggies, meats)  
but different  
implementations  
based on region.



## 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

# Pizza

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



Pizza becomes composed of different ingredients.

Question: How do we get these ingredients associated with a Pizza??



# One idea...Constructor w/many parameters

```
public class Pizza {  
  
    public Pizza(Dough d, Sauce s, Cheese c, Veggies v,  
                Pepperoni p, Clams c) {  
  
        this.dough = d;  
        this.sauce = s;  
        this.veggies = v;  
        this.cheese = c;  
        this.pepperoni = p;  
        this.clam = c;  
    }  
}
```

Makes sense...whenever you create a pizza, you must specify these ingredients...

# Rework NYPizzaStore

```
public class NYPizzaStore extends PizzaStore {  
  
    Pizza createPizza(String item) {  
  
        if (item.equals("cheese")) {  
            return new NYStyleCheesePizza(new ThinCrustDough(),  
                new Marinara(), new Reggiano(), null, null );  
  
        } else if (item.equals("veggie")) {  
            return new NYStyleVeggiePizza(new ThinCrustDough(),  
                new Marinara(), new Reggiano(), new Garlic() , null)  
        }  
    }  
}
```

This will cause a lot of maintenance headaches!!! Imagine what happens when we create a new pizza!

We know that we have a certain set of ingredients that are used for New York..yet we have to keep repeating that set with each constructor. Can we define this unique set just once??

# PizzaIngredientFactory (pg 146)

Create an interface for creating the different ingredients of Pizza.

```
public interface PizzaIngredientFactory {  
  
    public Dough createDough();  
    public Sauce createSauce();  
    public Cheese createCheese();  
    public Veggies[] createVeggies();  
    public Pepperoni createPepperoni();  
    public Clams createClam();  
  
}
```

Note that each ingredient of Pizza has a corresponding method defined in the interface.

# NYPizzaIngredientFactory (pg 147)

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {  
  
    public Dough createDough() {  
        return new ThinCrustDough();  
    }  
  
    public Sauce createSauce() {  
        return new MarinaraSauce();  
    }  
  
    public Cheese createCheese() {  
        return new ReggianoCheese();  
    }  
  
    . . .  
}
```

Here we see the set of ingredients that are specific to the NY region.

Note that each time a createXXX() method is called, a new instance is returned. This will be important later on.

# CheesePizza (pg 150)

```
public class CheesePizza extends Pizza {
```

```
    PizzaIngredientFactory ingredientFactory;
```

```
    public CheesePizza(PizzaIngredientFactory ingredientFactory) {  
        this.ingredientFactory = ingredientFactory;  
    }
```

```
    void prepare() {
```

```
        dough = ingredientFactory.createDough();
```

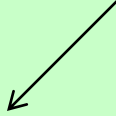
```
        sauce = ingredientFactory.createSauce();
```

```
        cheese = ingredientFactory.createCheese();
```

```
    }
```

```
}
```

Instead of many  
ingredient parameters,  
we just pass one.



← Creation  
delegated

# NYPizzaStore (pg 152)

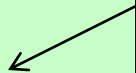
```
public class NYPizzaStore extends PizzaStore {
```

```
    protected Pizza createPizza(String item) {  
        Pizza pizza = null;
```

```
        PizzaIngredientFactory ingredientFactory =  
            new NYPizzaIngredientFactory();
```

```
        if (item.equals("cheese")) {  
            pizza = new CheesePizza(ingredientFactory);  
        } else if (item.equals("veggie")) {  
            pizza = new VeggiePizza(ingredientFactory);  
        } else if (item.equals("clam")) {  
            pizza = new ClamPizza(ingredientFactory);  
        } else if (item.equals("pepperoni")) {  
            pizza = new PepperoniPizza(ingredientFactory);  
        }  
        return pizza;  
    }
```

The specific ingredient factory is defined once for the NY region.



# Reworking the Pizza

```
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;
```

```
    }
```


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.



# Reworking the Pizza

```
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.



# Revisiting Pizza Store

```
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.

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.

# Abstract Factory Defined

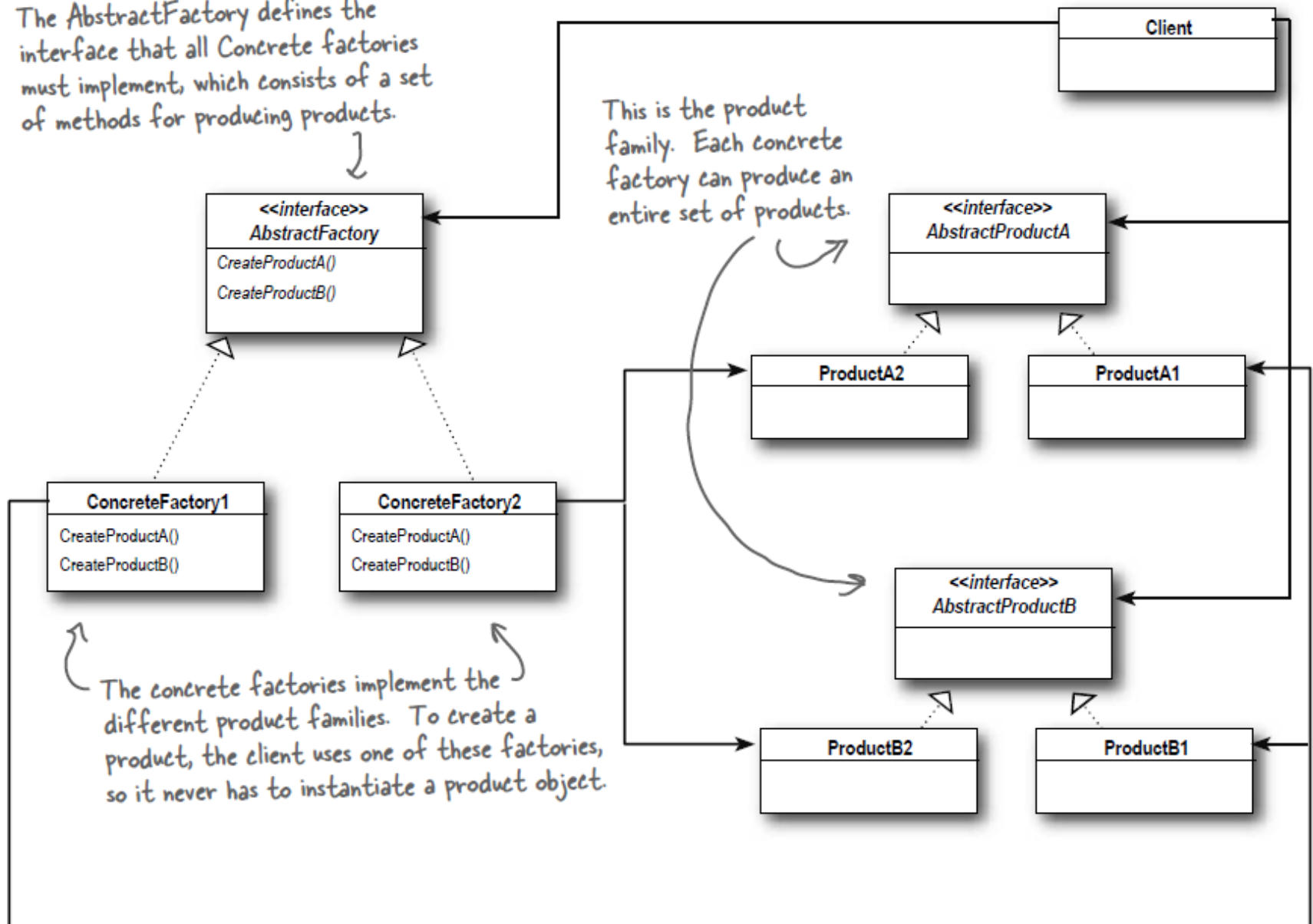
GoF Intent: “Provides an interface for creating families of related or dependent objects without specifying their concrete classes.”

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

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

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

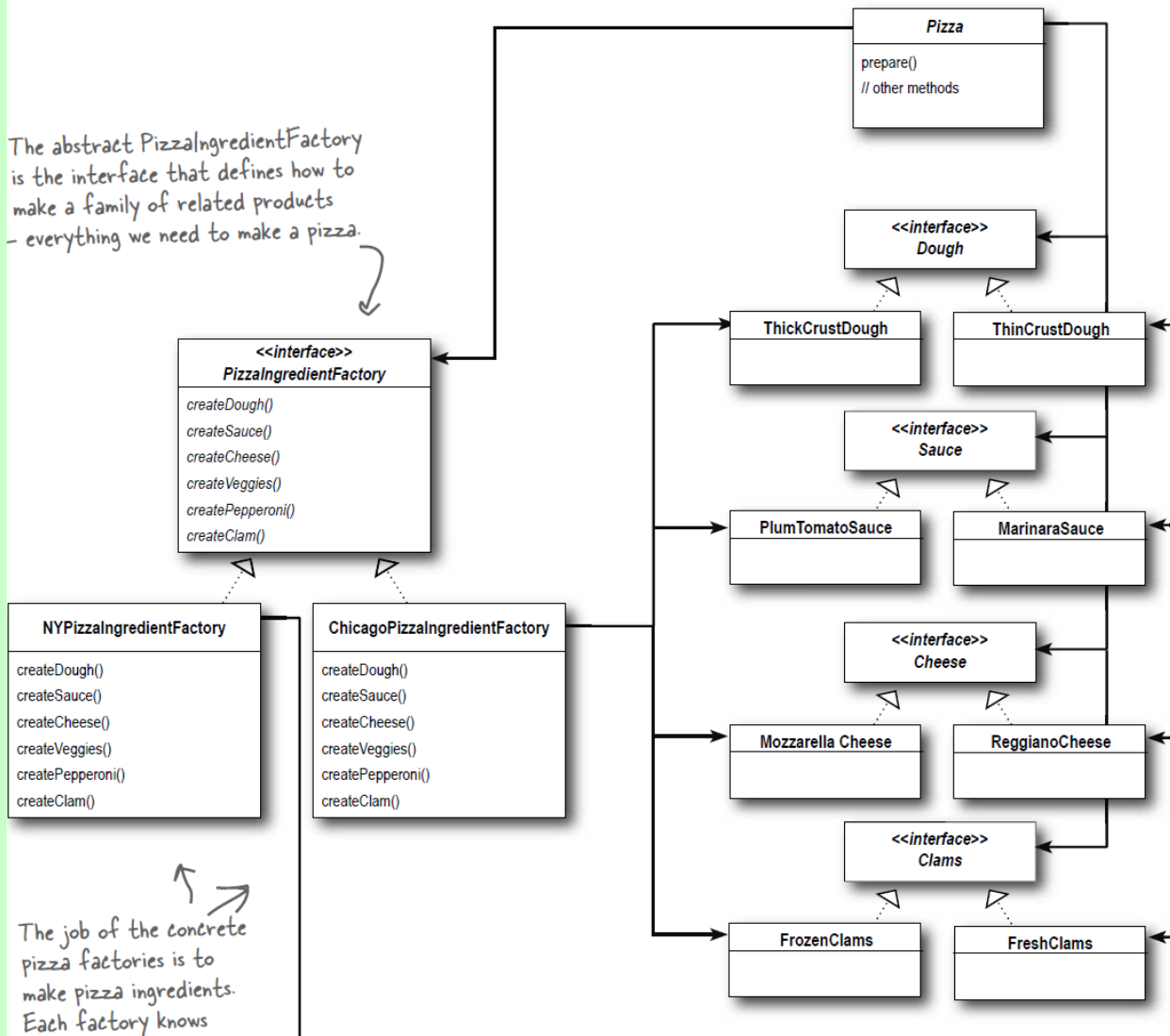
The concrete factories implement the different product families. To create a product, the client uses one of these factories, so it never has to instantiate a product object.



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

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.

# Factory Method vs. Abstract Factory

- Factory Method
  - Uses inheritance to create a Concrete Product
  - Sub classes decide which Concrete Product to use
- Abstract Factory
  - Uses composition to create objects
  - The objects created were a part of a family of objects. For example, NY region had a specific set of ingredients.
  - An abstract factory actually contains one or more Factory Methods!

# Database Example

- Imagine you wanted to create a database manager object as an enterprise component. This object must not be tied to a specific database implementation.
- Each specific database uses different objects:
  - Connection
  - Command
  - Adapter
  - Data Reader
- Oracle, SQL Server, Informix, etc will have their own version of these Sub Products.

# DatabaseFactory

```
public abstract class DatabaseFactory
{
    public abstract IDbCommand CreateCommand();
    public abstract IDbCommand CreateCommand(string cmdText);
    public abstract IDbCommand CreateCommand(string cmdText, IDbConnection cn);
    public abstract IDbConnection CreateConnection();
    public abstract IDbConnection CreateConnection(string cnString);
    public abstract IDbDataAdapter CreateDataAdapter();
    public abstract IDbDataAdapter CreateDataAdapter(IDbCommand selectCmd);
    public abstract IDataReader CreateDataReader(IDbCommand dbCmd);
}
```

Each database implementation (Oracle, SQL Server, etc) will need to create their own version of this DatabaseFactory.

The specific DatabaseFactory must be sent to the constructor of the framework class DatabaseManager.

# Abstract Factory: Last Thoughts

- This may be a hard pattern to “see” during the initial design process.
- You may encounter this pattern during refactoring.



# Summary

- ***Pattern Name*** – Abstract Factory
- ***Problem*** – Need to way to create a family of products
- ***Solution***
  - Create an interface for creating products

# References

- Design Patterns: Elements of Reusable Object-Oriented Software By Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Addison-Wesley. ISBN 0-201-63361-2.
- **Head First Design Patterns** By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates  
First Edition October 2004  
ISBN 10: 0-596-00712-4
- <https://web.cs.dal.ca/~jin/3132/lectures/dp-07.pdf>
- <http://www.slideshare.net/jonsimon2/factory-and-abstract-factory>