

CS 351
Design of Large Programs
Simple Factories, Factory Methods,
and Abstract Factories

Brooke Chenoweth

University of New Mexico

Spring 2024

Creating Objects

- ... it's more than just `new`
- Some key tenets
 - Instantiation shouldn't always be done in public
 - Constructor usages often lead to unintended coupling
- Remember the note in the Strategy pattern?
 - When we say “new” to create a new object by calling a constructor, we're directly programming to an implementation

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

Creating Objects

- ... it's more than just `new`
- Some key tenets
 - Instantiation shouldn't always be done in public
 - Constructor usages often lead to unintended coupling
- Remember the note in the Strategy pattern?
 - When we say "new" to create a new object by calling a constructor, we're directly programming to an implementation

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

We want to use the interface...

Creating Objects

- ... it's more than just `new`
- Some key tenets
 - Instantiation shouldn't always be done in public
 - Constructor usages often lead to unintended coupling
- Remember the note in the Strategy pattern?
 - When we say "new" to create a new object by calling a constructor, we're directly programming to an implementation

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

We want to use the interface...

But we're forced to create an instance of a concrete class!

It's More Complicated than That

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

- ... especially when you think about the fact that things might change
- E.g., you add a new type of duck and have to figure out when/how to instantiate it
- And you make new kinds of Ducks in all different parts of your code

“Open for Extension, Closed for Modification”

A key design goal:

- Allow classes to be easily extended to incorporate new behavior
- Without modifying existing code
 - Because every time you modify it, you risk introducing new bugs


This results in designs that are resilient to change but also flexible enough to accept new functionality to meet changing requirements

Back to Identifying Things that Change

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

Back to Identifying Things that Change

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



It'd really be nice to use an abstract class here, but, alas, one cannot instantiate an abstract class

Back to Identifying Things that Change

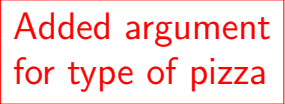
```
public Pizza orderPizza(String type) {
    Pizza pizza;

    if(type.equals("cheese")) {
        pizza = new CheezePizza();
    } 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;
}
```

Back to Identifying Things that Change

```
public Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    if(type.equals("cheese")) {  
        pizza = new CheezePizza();  
    } 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;  
}
```



Added argument
for type of pizza

Back to Identifying Things that Change

```
public Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    if(type.equals("cheese")) {  
        pizza = new CheezePizza();  
    } 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;  
}
```

Added argument
for type of pizza

Make concrete pizza
based on type

Back to Identifying Things that Change

```
public Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    if(type.equals("cheese")) {  
        pizza = new CheezePizza();  
    } 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;  
}
```

Added argument
for type of pizza

Make concrete pizza
based on type

Each pizza type knows
how to prepare itself

Change... it's Coming

- The pizza business is a trendy one

Change... it's Coming

- The pizza business is a trendy one
 - Greek pizza is so yesterday...

Change... it's Coming

- The pizza business is a trendy one
 - Greek pizza is so yesterday...
 - But with all of the people moving in from CA, we've got increasing demands for veggie pizza. And some weirdos who want clam pizza

Change... it's Coming

- The pizza business is a trendy one
 - Greek pizza is so yesterday...
 - But with all of the people moving in from CA, we've got increasing demands for veggie pizza. And some weirdos who want clam pizza
- What to do?

Change... it's Coming

- The pizza business is a trendy one
 - Greek pizza is so yesterday...
 - But with all of the people moving in from CA, we've got increasing demands for veggie pizza. And some weirdos who want clam pizza
- What to do?
- The `orderPizza` method is not *closed for modification*

Change... it's Coming

- The pizza business is a trendy one
 - Greek pizza is so yesterday...
 - But with all of the people moving in from CA, we've got increasing demands for veggie pizza. And some weirdos who want clam pizza
- What to do?
- The `orderPizza` method is not *closed for modification*
- What varies? What stays the same?

Change... it's Coming

- The pizza business is a trendy one
 - Greek pizza is so yesterday...
 - But with all of the people moving in from CA, we've got increasing demands for veggie pizza. And some weirdos who want clam pizza
- What to do?
- The `orderPizza` method is not *closed for modification*
- What varies? What stays the same?
 - The choices of pizza types change over time

Change... it's Coming

- The pizza business is a trendy one
 - Greek pizza is so yesterday...
 - But with all of the people moving in from CA, we've got increasing demands for veggie pizza. And some weirdos who want clam pizza
- What to do?
- The `orderPizza` method is not *closed for modification*
- What varies? What stays the same?
 - The choices of pizza types change over time
 - The process (algorithm) for filling an order stays the same

Information Hiding?

- What is it we're supposed to do with the stuff that changes?
- Encapsulate it!
- Practically, since the thing that's changing is *object creation*, we need an object that encapsulates object creation
- This object is called a *factory*
 - Then the `orderPizza` method is a *client* of the factory
 - Anytime it needs a pizza, it goes to the factory to request that one is created

The 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;
    }
}
```

Wait, what?

- Well, that seems silly. All we did is copy the code out of the `orderPizza` method, but it still has all of the same problems. . .

Wait, what?

- Well, that seems silly. All we did is copy the code out of the `orderPizza` method, but it still has all of the same problems. . .
- Does it?

Wait, what?

- Well, that seems silly. All we did is copy the code out of the `orderPizza` method, but it still has all of the same problems. . .
- Does it?
- The `SimplePizzaFactory` might have lots of clients (not just the `orderPizza` method)

Wait, what?

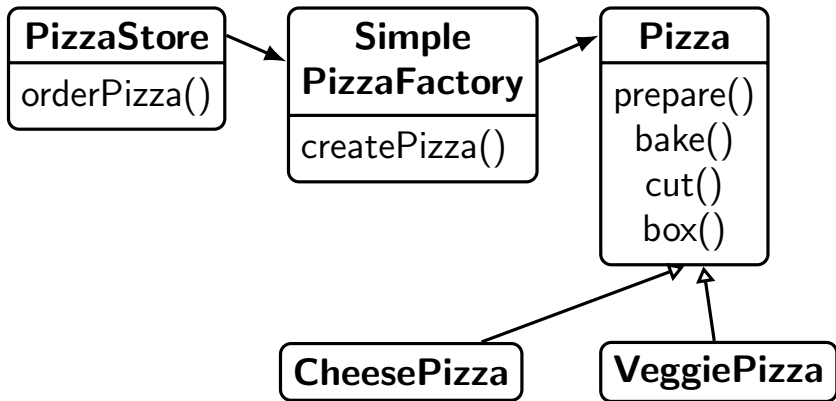
- Well, that seems silly. All we did is copy the code out of the `orderPizza` method, but it still has all of the same problems. . .
- Does it?
- The `SimplePizzaFactory` might have lots of clients (not just the `orderPizza` method)
- That was why we want to encapsulate the thing that changes!

Wait, what?

- Well, that seems silly. All we did is copy the code out of the `orderPizza` method, but it still has all of the same problems. . .
- Does it?
- The `SimplePizzaFactory` might have lots of clients (not just the `orderPizza` method)
- That was why we want to encapsulate the thing that changes!
- Also, the `orderPizza` method no longer needs to know anything at all about concrete Pizzas!

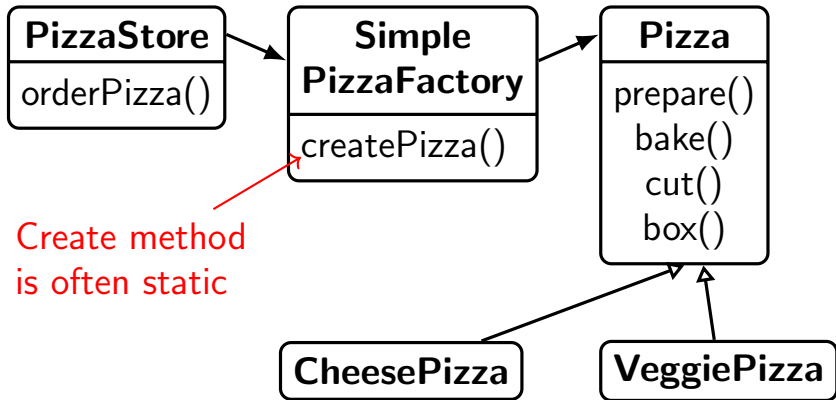
Simple Factory: Not Quite a Pattern

But it is a commonly used *programming idiom*



Simple Factory: Not Quite a Pattern

But it is a commonly used *programming idiom*



Using the Simple Factory

```
public class PizzaStore {
    private SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {

        Pizza pizza = factory.createPizza(type);

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

Using the Simple Factory

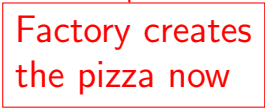
```
public class PizzaStore {
    private SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {

        Pizza pizza = factory.createPizza(type);

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



Franchising the Pizza Store

- Now you want to spread your successful business
 - We want to localize the pizza making activities to the `PizzaStore` class (For quality control)
 - But we want to give regional franchises the liberty to have their own pizza styles
- General framework:
 - Make the `PizzaStore` abstract
 - Put the `createPizza` method back in `PizzaStore`, but make it abstract
 - Create a `PizzaStore` subclass for every regional type of pizza


The Abstract Method

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

The Abstract Method

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

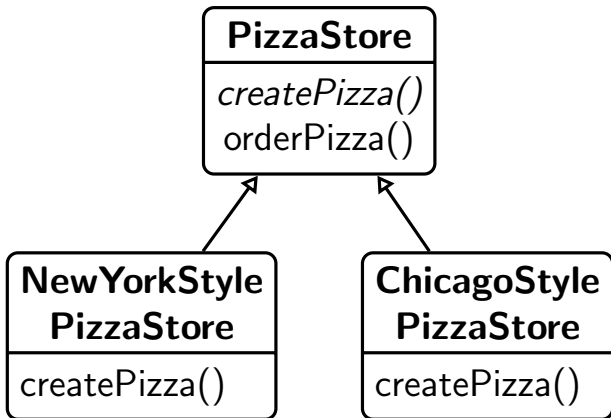
This is the
"factory method"



Delegating to the Subclasses

- We've perfected the pizza ordering method, and it stays the same across all of the subclasses
- But now the regional franchises can differ in the style of pizza they make
 - E.g., thin crust in New York, thick crust in Chicago
- While the `orderPizza` method looks like it's defined in the `PizzaStore` class, this class is abstract
 - It can't *actually* do anything
 - So when it is executed, it is actually executing in the context of a concrete subclass
 - This context gets determined when the (abstract) method `createPizza` gets called.

Delegating to the Subclasses



What's a Franchise Look Like?

- Bonus. The franchises get all of the benefits of the perfected PizzaStore ordering process
- All they have to do is define how to create pizzas!

```
public class NewYorkPizzaStore extends PizzaStore {
    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new NewYorkStyleCheesePizza();
        } else if (type.equals("pepperoni")) {
            return new NewYorkStylePepperoniPizza();
        } else if (type.equals("clam")) {
            return new NewYorkStyleClamPizza();
        } else if (type.equals("veggie")) {
            return new NewYorkStyleVeggiePizza();
        } else return null;
    }
}
```

A General Factory Method

abstract `Product` `factoryMethod(String type)`

- A factory method is abstract so the subclasses are counted on to handle object creation
- The factory method isolates the client (the code in the superclass) from knowing what kind of concrete `Product` is created
- A factory method returns a `Product` that is typically used within methods defined in the superclass
- A factory method may be parameterized (or not) to select among several variations of a `Product`

Ordering a Pizza

1. First, the customer needs to get a NY PizzaStore:

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

2. Now the pizza store can accept our order

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

3. The orderPizza method calls the createPizza method

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

- Remember the createPizza method is implemented in the subclass, so we're automatically getting a NY style cheese pizza here

4. The orderPizza method finishes preparing our pizza

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

- These methods are defined in the abstract PizzaStore class, which doesn't need to know which kind of pizza it is in order to follow the steps

We need pizzas for whole solution

```
public abstract class Pizza {
    protected String name;
    protected String dough;
    protected String sauce;
    protected List<String> toppings = new ArrayList<>();

    public void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough... ");
        System.out.println("Adding sauce... ");
        System.out.println("Adding toppings: ");
        for (String topping : toppings) {
            System.out.println("    " + topping);
        }
    }

    public void bake() { System.out.println("Baking pizza"); }

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

    public void box() { System.out.println("Putting pizza in box"); }
}
```

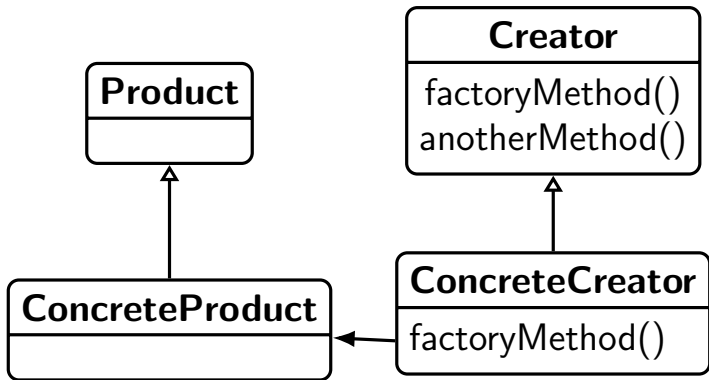

We need pizzas for whole solution

```
public class NewYorkStyleCheesePizza extends Pizza {  
  
    public NewYorkStyleCheesePizza() {  
        name = "NY Style Sauce and Cheese Pizza";  
        dough = "Thin Crust Dough";  
        sauce = "Marinara Sauce";  
  
        toppings.add("Grated Reggiano Cheese");  
    }  
}
```

We need pizzas for whole solution

```
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");  
    }  
  
    public void cut() {  
        System.out.println("Cutting pizza"  
                            + " into square slices");  
    }  
}
```

The Entire Solution



Parallel Class Hierarchies: creators and products

The Factory Method Pattern

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

Our “Dumb” Pizza Store Revisited

- Imagine going back to the beginning and creating a `PizzaStore` that amassed all of the decision making
- Inside the `createPizza` method of this pizza store, I would just have a huge, nested set of if statements to determine which style of pizza and then which type of pizza to create

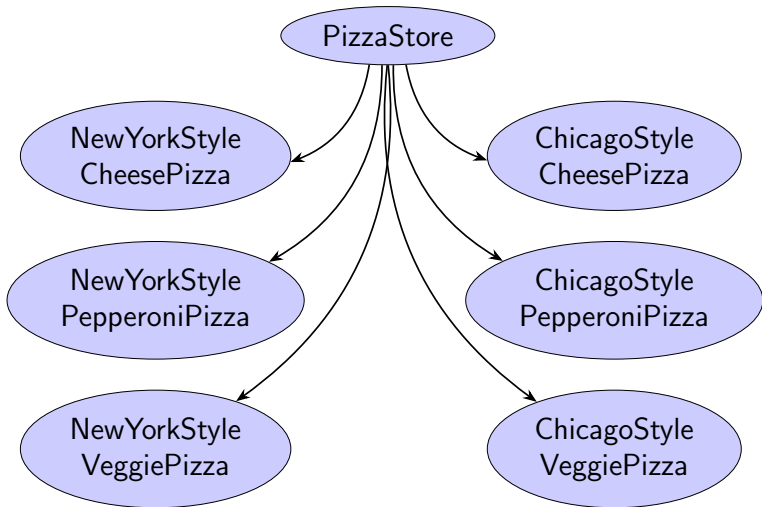
What Does This Look Like?

```
public class PainfulPizzaStore extends PizzaStore {
    public Pizza createPizza(String style, String type) {
        if (style.equals("NewYork") {
            if (type.equals("cheese")) {
                return new NewYorkStyleCheesePizza();
            } else if (type.equals("pepperoni") {
                return new NewYorkStylePepperoniPizza();
            } else if (type.equals("clam") {
                return new NewYorkStyleClamPizza();
            } else if (type.equals("veggie") {
                return new NewYorkStyleVeggiePizza();
            } else return null;
        } else if (style.equals("Chicago") {
            if (type.equals("cheese")) {
                return new ChicagoStyleCheesePizza();
            } else if (type.equals("pepperoni") {
                return new ChicagoStylePepperoniPizza();
            } else if (type.equals("clam") {
```

What Does This Look Like?

- This PizzaStore depends on all the concrete pizza types, since we create them directly.
- If the implementation of the pizza classes change, we may have to modify PizzaStore.
- Every new kind of pizza creates another dependency for PizzaStore

What Does This Look Like?



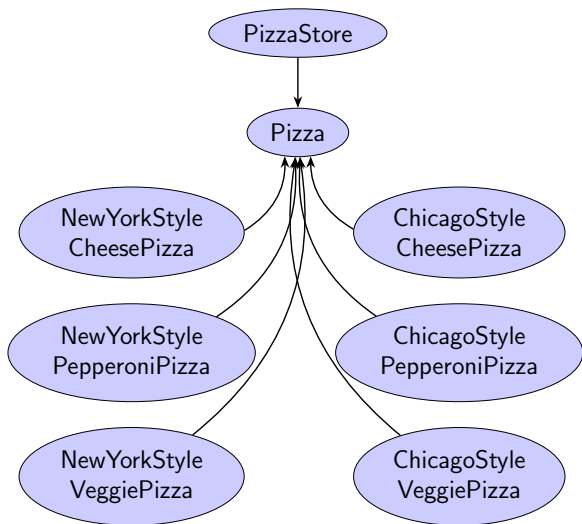
Another Design Principle

- This seems like a bad idea. We're definitely not encapsulating for change.
- If we change any of the concrete pizza classes, we have to change the PizzaStore because it *depends* on them
- Instead we should depend upon abstractions. *Do not depend upon concrete classes*
- High level components should not depend on low-level components; instead, both should depend on abstractions

Another Design Principle

- For example, in the previous pizza store, the store depended on all of the pizza types
- Instead, the pizza store should depend on the abstract notion of Pizza, and the concrete pizza types should too
- This is exactly what the Factory Method pattern we applied did!

Dependency Inversion



Guidelines that Help

- Only guidelines, **not rules to follow**
- No variable should hold a reference to a concrete class
 - If you use new, you'll be holding a reference to a concrete class
 - Use a factory to get around that!
- No class should derive from a concrete class
 - If you do, you're depending on the concrete class
 - Instead, derive from an abstraction (like an interface or an abstract class)
- No method should override an implemented method of any of its base classes
 - If you do, then your base class wasn't really an abstraction
 - The methods implemented in the base class are meant to be shared by the derived classes

Controlling Pizza Quality

- Some of your franchises have gone rogue and are substituting inferior ingredients to increase their per-pizza profit
- Time to enter the pizza ingredient business
- You'll make all the ingredients yourself and ship them to your franchises
- But this is not so easy. . .
- You have the same product families (e.g., dough, sauce, cheese, veggies, meats, etc.) but different implementations (e.g., thin vs. thick or mozzarella vs. reggiano) based on region

The Ingredient Factory Interface

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

Then What?

1. For each region, create a subclass of the `PizzaIngredientFactory` that implements the concrete methods
2. Implement a set of ingredients to be used with the factory (e.g., `ReggianoCheese`, `RedPeppers`, `ThickCrustDough`)
These can be shared among regions if appropriate
3. Integrate these new ingredient factories into the `PizzaStore` code

The New York Ingredient Factory

```
public class NewYorkPizzaIngredientFactory
    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();
    }
}
```


Connecting to the Pizzas

- Now, we need to force our franchise owners to only use factory produced ingredients
- Before, the abstract Pizza class just had Strings to name its ingredients
 - It implemented the `prepare()` method (and `bake()`, `cut()`, and `box()`)
 - The concrete Pizza classes just defined the constructor which, in some cases, specialized the ingredients (and sometimes cut corners) and maybe overrode other methods
- Now, the abstract Pizza class has actual ingredient objects
 - And the `prepare()` method is abstract
 - The concrete pizza classes will collect the ingredients from the factories to prepare the pizza

Concrete Pizzas

- Now, we only need one CheesePizza class (before we had a ChicagoCheesePizza and a NewYorkCheesePizza)
- When we create a CheesePizza, we pass it an IngredientFactory, which will provide the (regional) ingredients

An Example 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();
    }
}
```

An Example 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();
    }
}
```

Which cheese is created is determined at run time by the factory passed at object creation time

Fixing the Pizza Stores

```
public class NewYorkPizzaStore extends PizzaStore {
    protected Pizza createPizza(String type) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NewYorkPizzaIngredientFactory();
        if (type.equals("cheese")) {
            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("New York Style Cheese Pizza");
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("New York Style Veggie Pizza");
        } // more of the same...
        return pizza;
    }
}
```

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

Whew. Recap.

- We provided a means of creating a family of ingredients for pizzas by introducing a new type of factory: the *abstract factory*
- An abstract factory provides an interface for creating a family of products
 - Decouples code from the actual factory that creates the products
 - Makes it easy to implement a variety of factories that produce products for different contexts (we used regions, but it could just as easily be different operating systems, or different “look and feels”)
- We can substitute different factories to get different behaviors

The Abstract Factory Pattern

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

Factory Method vs. Abstract Factory

- Decouples applications from specific implementations
 - Creates objects through inheritance
 - Create objects by extending a class and overriding a factory method
 - Useful if you don't know ahead of time what concrete classes will be needed
- Decouples applications from specific implementations
 - Creates objects through object composition
 - Create objects by providing an abstract type for a family of products
 - Subclasses define how products are produced
 - Interface must change if new products are added