

CS 230 SOFTWARE ENGINEERING

# DESIGN PATTERNS

FACTORY, ABSTRACT FACTORY

Professor Miryung Kim

UCLA Computer Science

Based on Materials from CS 130

# AGENDA

- ▶ Factory Method
- ▶ Abstract Factory

# THE FACTORY METHOD PATTERN

# CALLING THE CONSTRUCTOR

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

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

# ANOTHER EXAMPLE IN SIMILAR VEIN

```
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()  
}
```

# 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.

# ANOTHER EXAMPLE IN SIMILAR VEIN

```
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

# ENCAPSULATING OBJECT CREATION

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

**SimplePizzaFactory**



# 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;  
    }  
}
```

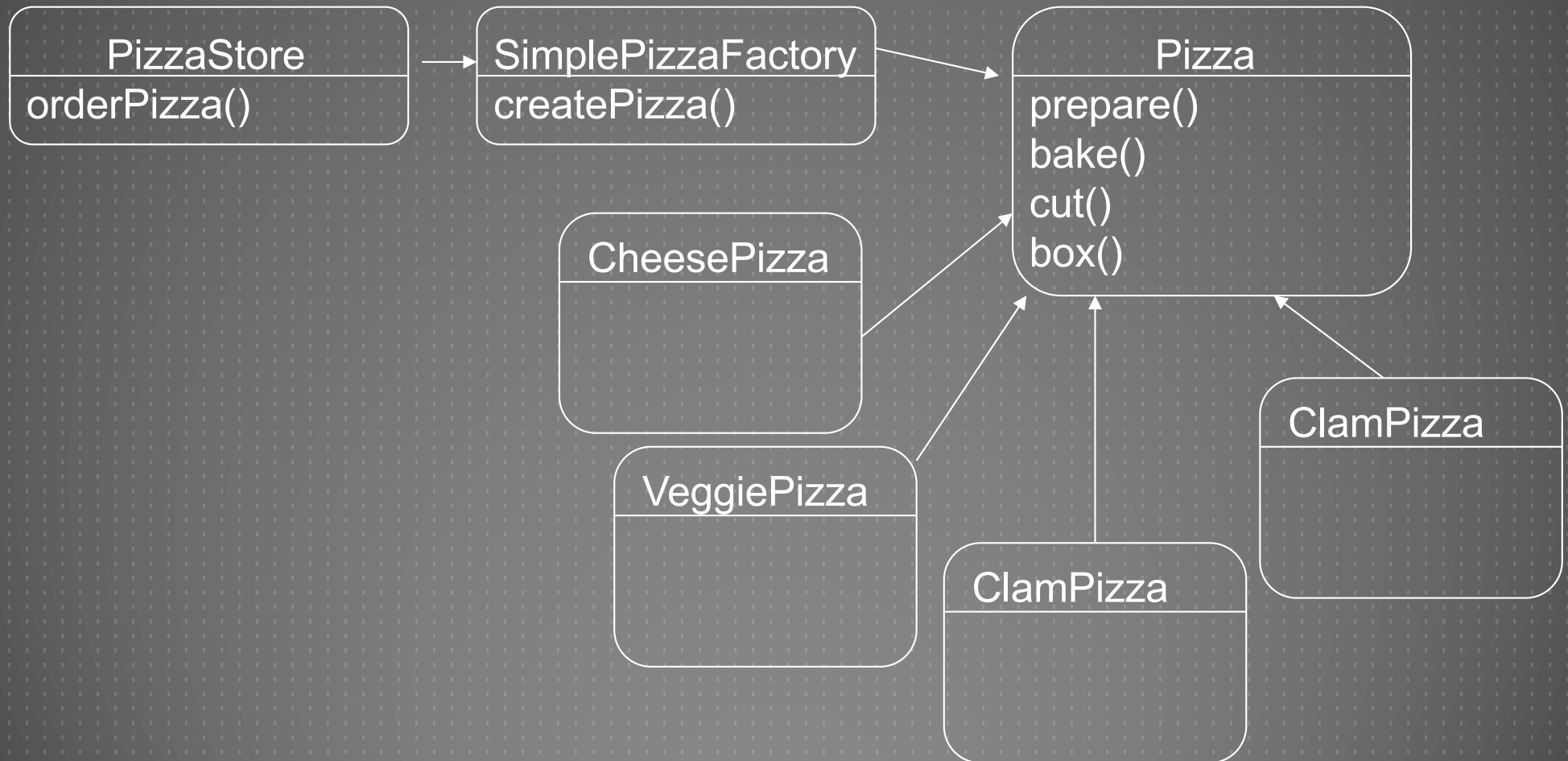
# REWORKING THE PIZZASTORE CLASS

```
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;  
    }  
}
```

# COMPLETE EXAMPLE FOR SIMPLE FACTORY

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

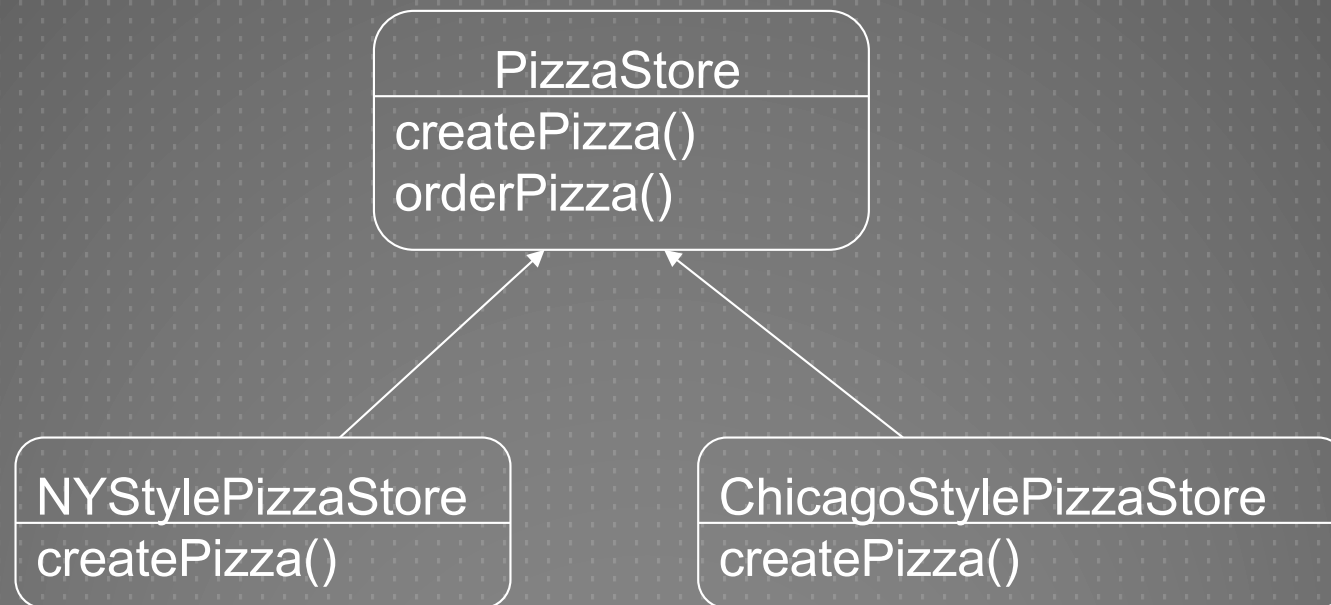
# SIMPLE FACTORY DEFINED



# ABSTRACT METHOD – A FRAMEWORK FOR THE PIZZA STORE

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

# ALLOWING THE SUBCLASSES TO DECIDE



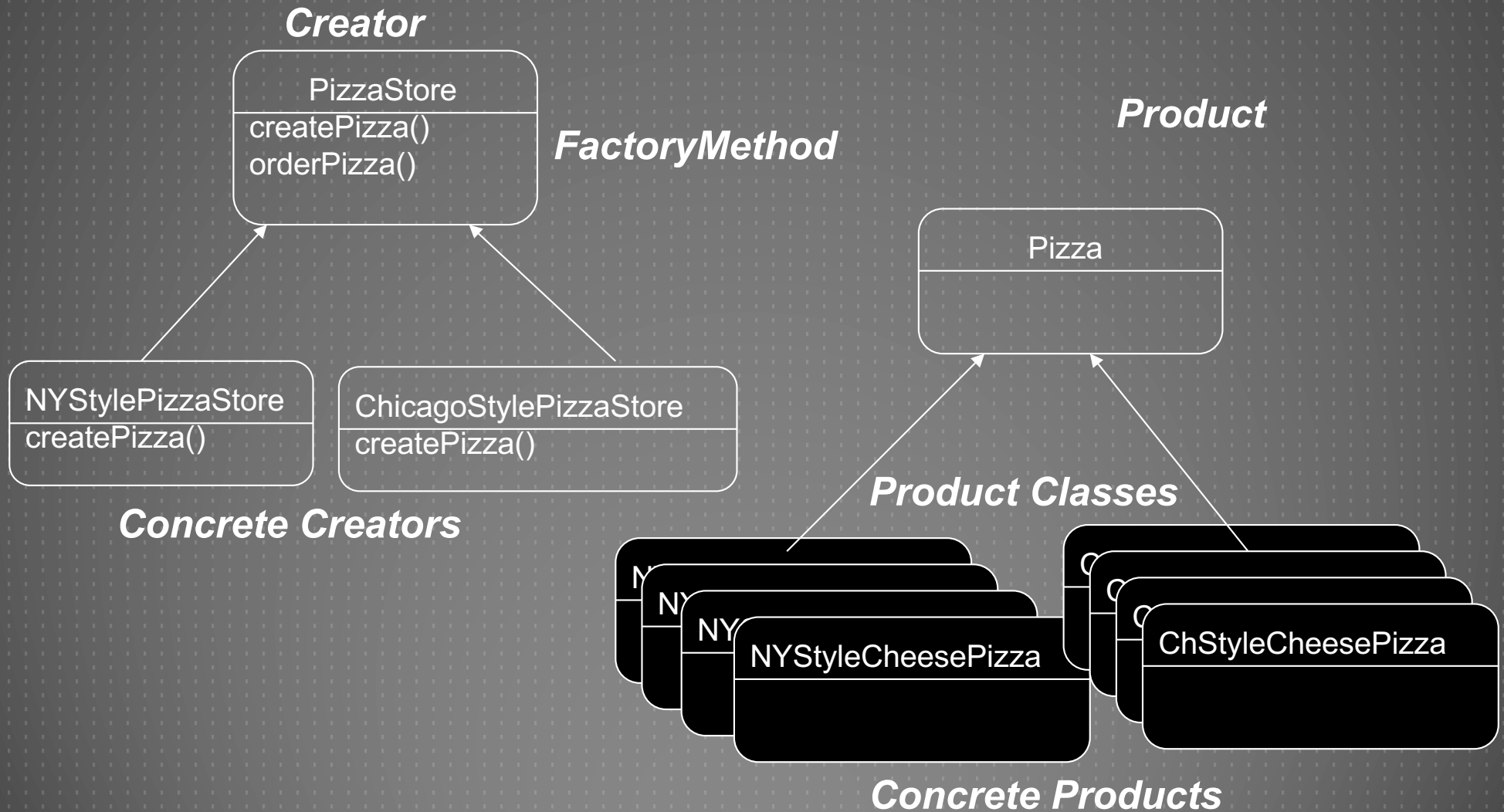
**A factory method handles object creation and encapsulates it in the subclass. This decouples the client code in the super class from the object creation that happens in the subclass.**

# CREATING MULTIPLE INSTANCES

```
NYPizzaFactory nyFactory = new NYPizzaFactory();  
PizzaStore nyStore = new PizzaStore(nyFactory);  
Pizza pizza = nyStore.orderPizza("cheese");
```

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();  
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);  
Pizza pizza = chicagoStore.orderPizza("cheese");
```

# FACTORY METHOD PATTERN





# PROS AND CONS

- ▶ Pros: makes it easy to add a new product type
  - ▶ Makes it easy to add a new creator
  - ▶ Makes it easy to change the conditions under which different objects are created.
  - ▶ Client applications do not know individual concrete types of objects being created.
- ▶ Cons:
  - ▶ A little bit complicated to understand two different parallel class hierarchies
  - ▶ If you want to know which specific type of an object is created, it requires some extra effort for program understanding.

# 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

# THE ABSTRACT FACTORY

# 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 createSuace();
    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 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()};
    }
}
```

# 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 overwrote 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 NYCheesePizza)
- ▶ 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();  
    }  
}
```

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

# FIXING THE PIZZA STORES

```
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");  
        } // 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

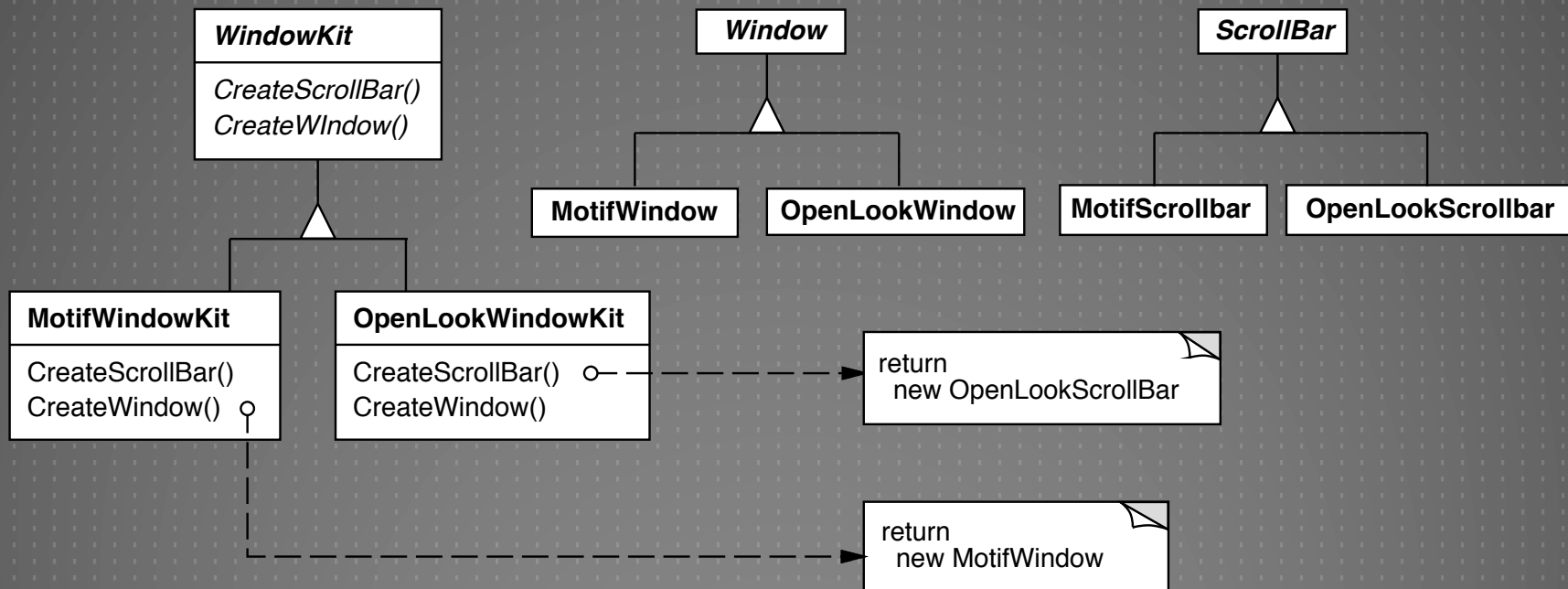
# WHEW. RECAP.

- ▶ 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.

# THINK-PAIR-SHARE



**Which class is an abstract factory?**  
**Which classes are concrete factories?**

# THINK-PAIR-SHARE

- ▶ Which classes must be modified if you want to add a different look and feel called MacWindowKit?
- ▶ Which classes must be modified if you want to add a new type of object such as a button as a part of WindowKit?

# THINK-PAIR-SHARE

- ▶ What happens if we have multiple types of windows?
- ▶ What happens if we need different types of windows that take different arguments?
- ▶ What happens if we want to define a window as combination of window, scroll bar and button?



QUESTIONS?