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

```
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();
  }
}
```

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 `String`s 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

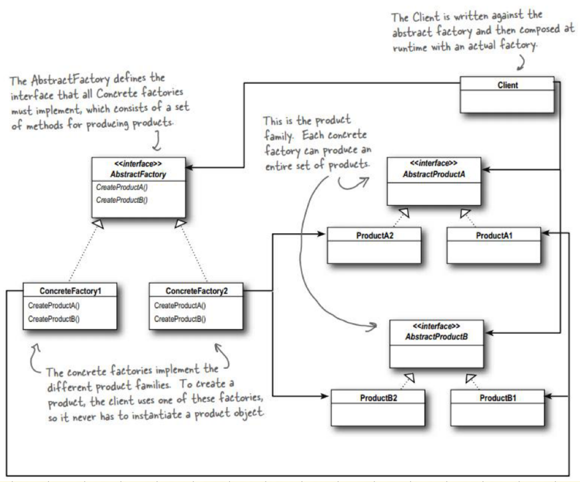
Fixing the Pizza Stores

```
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;
  }
}
```

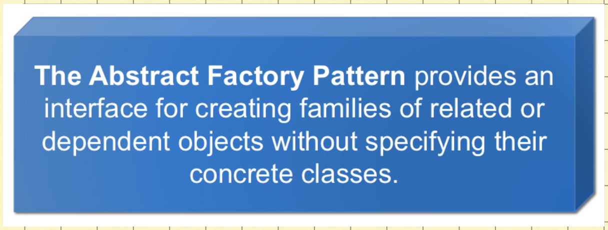
For each type of pizza, we instantiate a new pizza and give it the factory it needs to get its ingredients

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

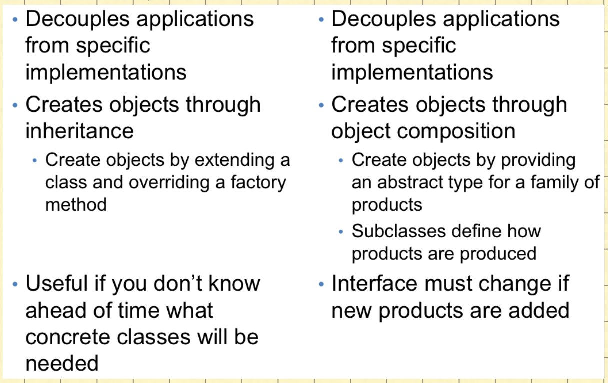

The Client is written against the abstract factory and then composed at runtime with an actual factory.
Client
The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.
<<interface>> AbstractFactory
CreateProductA()
CreateProductB()
This is the product family. Each concrete factory can produce an entire set of products.
<<interface>> AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
<<interface>> AbstractProductB
ProductB2
ProductB1
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.

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