The Factory Pattern

Updated: 3/26/2013

Based on Head-First Design Patterns, Chapter 4
Overview

- Motivation
- The Pizza Store application
- Simple Factory approach
- Ideas for Improvement
- New PizzaStore class
- Defining the Factory Method Pattern
Motivation

- Allowing client programs to create concrete classes prevents flexibility and information hiding:
  - if the class types should change, all client programs will also have to change
  - clients should not have too much information about a library's internal object types
  - *Principle*: avoid coding to implementations—always code to interfaces
orderPizza method

This code is hard to maintain!

Pizza orderPizza(String type) {
    Pizza pizza;
    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza();
    } etc.
Simple Factory

- Use a factory class to determine the specific types of objects to be created.
- Store an instance of the factory class in the class being instantiated.
- Example code from the book:
  - PizzaStore contains an instance of PizzaFactory
  - example code:
    - src/headfirst/factory/pizzas
Drawbacks

• The simple factory approach lacks flexibility.
• What if different types of factories are needed, and they must be selected at runtime?
• In the pizza store application, how can we localize the pizza making activities in the PizzaStore class, yet give franchises freedom to have their own regional styles?
  – constraint: we don't want them to invent completely new pizza names
What is wrong with this approach?

```java
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();
            }
        }
        etc.
    }
}
```

How many objects would be affected if you added California Style Pizzas?

This class is dependent on the different pizza objects.
Dependency Inversion Principle

• Depend upon abstractions. Do not depend upon concrete classes.

• Or, to put it another way…
  – high-level components should never depend upon lower-level components.

• For the pizza store app:
  – let the abstract PizzaStore class depend on the abstract Pizza class
Coding Guidelines

Relating to the Dependency Inversion Principle:

• No variable should hold a reference to a concrete class (don't use the new operator)
• No class should derive from a concrete class
• No method should override an implemented method of any of its base classes

Impossible to follow these guidelines all the time!
Ideas

• Make PizzaStore abstract, and have it define the orderPizza and createPizza methods

• create a subclass for each regional style, such as:
  – NYPizzaStore, ChicagoPizzaStore

• Each pizza store will decide what ingredients go into standard types of pizzas.
Start with an abstract factory class:

```java
public abstract class PizzaStore {

    abstract Pizza createPizza(String item);

    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        System.out.println("--- Making a " +
                        pizza.getName() + " ---");
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```
Extend the abstract factory with a concrete factory class that decides which pizza instances to create:

```java
public class ChicagoPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new ChicagoStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new ChicagoStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new ChicagoStylePepperoniPizza();
        } else return null;
    }
}
```
Here's one of the objects created by the factory, for cheese pizzas in Chicago

```java
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");
    }

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

        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a " + pizza.getName());

        pizza = nyStore.orderPizza("clam");
        System.out.println("Ethan ordered a " + pizza.getName());
    }
}
Next: Ingredient Factories

• Each store in the franchise wants to build a custom type of pizza
  – regional differences (styles) mean different ingredients
  – But ingredient quality must be maintained, so an ingredient factory will produce and ship the ingredients to the stores
interface PizzaIngredientFactory

class NYPizzaIngredientFactory implements PizzaIngredientFactory

class ChicagoPizzaIngredientFactory implements PizzaIngredientFactory
abstract class Pizza {
    // collect the ingredients, using the factory
    abstract void prepare();
}

// sample concrete class:
class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    // collect the ingredients, using the factory
    abstract void prepare() {
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
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");
        }
    }
}
Summary

To summarize:

• 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.