Factory Pattern

The **Factory Pattern**, also called the **Virtual Constructor Pattern**, is a creational pattern that allows us to create objects without exposing the creation logic to the client, which could cause **coupling problems**. The newly created objects are referred to by a common interface and the factory handles the details of object creation. In doing so, we follow the design principle of programming to an interface and not an implementation.

Suppose we have a method, orderPizza, which creates a different object based on the type of pizza.

Pizza orderPizza(String type) {  
 Pizza pizza = null;  
 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();  
 return pizza;  
}

JAVA

This is bad code for two reasons. Firstly, the orderPizza method should be doing just that, ordering the pizza. It shouldn’t be performing additional duties like type checking for the specific type of pizza. This breaks the **Single Responsibility Principle**. Secondly, if there are changes to the types of pizza available, the code has to be re-written. This breaks the **Open Closed Principle**.

An easy way to get around these issues is to move the creation logic to its own class. This hides the creation logic from the client and separates the creation logic from the operation. This type of class is called a **factory**. The factory is a separate class as opposed to a method because the creation logic can become quite complicated, too much for a method to contain.

public class PizzaFactory() {  
 public Pizza createPizza(String type) {  
 Pizza pizza = null;  
 if (type.equals("cheese")) {  
 pizza = new CheesePizza();  
 } else if (type.equals("greek")) {  
 pizza = new GreekPizza();  
 } else if (type.equals("pepperoni")) {  
 pizza = new PepperoniPizza();  
 }  
 }  
}

JAVA

## Abstract Factory

An **abstract factory** is a factory of factories. This can be useful in cases where there are several factories that have common behaviour.

public abstract class AbstractPizzaFactory() {  
 public abstract Pizza createPizza(String type);  
}  
  
public class PizzaFactory() extends AbstractPizzaFactory {  
 public Pizza createPizza(String type) {  
 // code  
 }  
}

public class NewYorkPizzaFactory() extends AbstractPizzaFactory {  
 public Pizza createPizza(String type) {  
 // code  
 }  
}

JAVA

The use of an abstract factory provides some justification for using a factory class instead of a factory method, since only classes can create the inheritance structure.

The actual factory to be used must be created by yet another class which closely represents the factory class. It is a class which creates factories, thus being a factory of factories.

public class PizzaFactoryProducer() {  
 public AbstractPizzaFactory getFactory(String type) {  
 if (type.equals("new\_york")) {  
 return new NewYorkPizzaFactory();  
 }  
 return new PizzaFactory();  
 }  
}

JAVA

The only benefit of using an abstract factory is to reduce the possibility of errors due to repeated logic. However, we must be careful to not overengineer our code and only use this pattern when it is required.