

Factory Pattern

Factory Design Pattern

Motivation

The Factory Design Pattern is maybe the most used pattern in modern languages like Java and C#. Most probably if you are searching for factory design pattern on the internet most of the results are about Factory Method and Abstract Factory.

NEW Forum Discussions: NEW

What Design Pattern Should I Choose?

Design Principles and Patterns Factory Design Pattern

Motivation

The Factory Design Pattern is maybe the most used pattern in modern languages like Java and C#. Most probably if you are searching for factory design pattern on the internet most of the results are about Factory Method and Abstract Factory.

Both of the above mentioned are factories but are different or maybe are just some extensions of the factory design pattern. The Factory pattern is not included in the GoF (Gang of Four).

Intent

- creates objects without exposing the instantiation logic to the client.
- refers to the newly created object through a common interface

Implementation

The implementation is really simple

- The client need a product, but instead of creating it directly using the new operator it asks for a new product to the factory providing some information about the type of object it needs.
- The factory instantiates a new concrete product and then return to the client the created product casted as abstract product.
- The client uses the products only as abstract products without being aware about their concrete implementation.

Applicability & Examples

Probably the factory pattern is one of the most used patterns.

For example a graphical application works with shapes. In our implementation the drawing framework is the client and the shapes are the products. All the shapes are derived from an abstract shape class (or interface). The Shape class defines drawing and moving operations that all the concrete shapes needs to implement. A command is selected from the menu the to create a new Circle. The framework receive the shape type as a string asks the factory to create a new shape sending the parameter received from menu. The factory create a new circle and return it to the framework casted as abstract shape. Then the framework uses only the object as abstract without knowing what type of object it is.

The advantage is very clear. New shapes can be added without changing the framework. There are certain factory implementations that allow adding new products without changing the factory itself.

Specific problems and implementation

Procedural Solution - switch/case noob instantiation.

Those are also known as parameterized Factories. The generating method can be written so that it can generate more types of Product objects, using a condition (entered as a method parameter or read from some global configuration parameters - see abstract factory) to identify the type of the object that should be created, as below:

```
public abstract class ProductFactory{
public Product createProduct(String ProductID){
    if (id==ID1)
        return new OneProduct();
    if (id==ID2) return
        return new AnotherProduct();
    ... // so on for the other Ids

    return null; //if the id doesn't have any of the expected values
}
```

```
...
}
```

This implementation is the most simple and intuitive (Let's call it noob implementation). The problem here is that once we add a new concrete product call we should modify the Factory class. It is not very flexible and it violates open close principle. Of course we can subclass the Creator class, but let's not forget that the factory class is usually used as a singleton. Subclassing it means replacing all the factory class references everywhere through the code.

Another implementation based on the same procedural implementation can be implemented as a Static Factory. See <> for details.

Class Registration - using reflection

In order to be able to create objects inside the factory class without knowing the object type we keep a map between the productID and the class type of the product. In this case when a new product is added to the application it has to be registered to the factory. This operation doesn't require any change in the factory class code.

```
class ProductFactory
{
    private HashMap m_RegisteredProducts = new HashMap();

    public void registerProduct (String productID, Class productClass)
    {
        m_RegisteredProducts.put(productID, productClass);
    }

    public Product createProduct(String productID)
    {
        Class productClass = (Class)m_RegisteredProducts.get(productID);
        Constructor productConstructor = cClass.getDeclaredConstructor(new Class[] { String.class });
        return (Product)productConstructor.newInstance(new Object[] { });
    }
}
```

We can put the code responsible with the registration in the main function, or we can put it in the main method (or everywhere outside of the product classes and factory class), or we can put it inside the product class:

```
1. Registration done outside of product classes: public static void main(String args[]){
    Creator.instance().registerProduct("ID1", OneProduct.class);
}
2. Registration done inside the product classes: class OneProduct extends Product
{
    static {
        Creator.instance().registerProduct("ID1",OneProduct.class);
    }
    ...
}
```

We have to be sure that the product concrete classes are loaded before they are created by the factory (if they are not loaded they will not be registered in the factory and createProduct will return null):

```
class Main
{
    static
    {
        try
        {
            Class.forName("OneProduct");
            Class.forName("AnotherProduct");
        }
        catch (ClassNotFoundException any)
        {
            any.printStackTrace();
        }
    }
    public static void main(String args[]) throws PhoneCallNotRegisteredException
    {
        ...
    }
}
```

This reflection implementation has its own drawbacks. The main one is performance. When the reflection is used the performance can be slowed to 10%. Another issue is that not all the programming languages provide reflection mechanism.

Class Registration - avoiding reflection

As we saw in the previous paragraph the factory object uses internally a HashMap to keep the mapping between parameters (in our case Strings) and concrete products class. The registration is made from outside of the factory and because the objects are created using reflection the factory is not aware of the objects types.

We don't want to use reflection but in the same time we want to have a reduced coupling between the factory and concrete products. Since the factory should be unaware of products we have to move the creation of objects outside of the factory to an object aware of the concrete products classes. That would be the concrete class itself.

We add a new abstract method in the product abstract class. Each concrete class will implement this method to create a new object of the same type as itself. We also have to change the registration method such that we'll register concrete product objects instead of Class objects.

```
abstract class Product
{
    public abstract Product createProduct();
    ...
}

class OneProduct extends Product
{
    ...
    static
    {
        ProductFactory.instance().registerPhoneCall("ID1", new OneProduct());
    }
    public OneProduct createProduct()
    {
        return new OneProduct();
    }
    ...
}

class ProductFactory
{
    public void registerPhoneCall(String productID, Product p) {
        m_PhoneCallClasses.put(productID, p);
    }

    public Product createProduct(String productID){
        ((Product)m_RegisteredProducts.get(productID)).createProduct();
    }
}
```

A more advanced solution - Factory design pattern with abstractions(Factory Method)

This implementation represents an alternative for the class registration implementation. Let's assume we need to add a new product to the application. For the procedural (noob) implementation we need to change the Factory class, while in the class registration implementation all we need is to register the class to the factory without actually modifying the factory class. For sure this is a flexible solution.

The procedural implementation is the classical bad example for the Open-Close Principle. As we can see there the most intuitive solution to avoid modifying the Factory class is to extend it.

This is the classic implementation of the factory method pattern. There are some drawbacks over the class registration implementation but no so many advantages:

- + The derived factory method can be changed to perform additional operations when the objects are created (maybe some initialization based on some global parameters ...).
- - The factory can not be used as a singleton.
- - Each factory has to be initialized before using it.

- - More difficult to implement.
- - If a new object has to be added a new factory has to be created.

Anyway, this classic implementation has the advantage that it will help us understanding the Abstract Factory design pattern.

Conclusion:

When you design an application just think if you really need it a factory to create objects. Maybe using it will bring unnecessary complexity in your application. Anyway if you have many object of the same base type and you manipulate them mostly as abstract objects, then you need a factory. If you're code should have a lot of code like the following, reconsider it.

If you're code should have a lot of code like the following, reconsider it: (if (ConcreteProduct)genericProduct typeof)
((ConcreteProduct)genericProduct).doSomeConcreteOperation().

If you decided I would recommend using one of class registration implementations, not the Factory Method (Factory design pattern with abstractions). Please note the procedural (noob) implementation is the simplest but violates the OCP principle.