

Factory Method Pattern

Also known as Virtual Constructor, the Factory Method is related to the idea on which libraries work: a library uses abstract classes for defining and maintaining relations between objects. One type of responsibility is creating such objects. The library knows when an object needs to be created, but not what kind of object it should create, this being specific to the application using the library.

Intent

- Defines an interface for creating objects, but let subclasses to decide which class to instantiate
- Refers to the newly created object through a common interface

Implementation

The pattern basically works as shown below, in the UML diagram:

Factory Method Pattern

Motivation

Also known as Virtual Constructor, the Factory Method is related to the idea on which libraries work: a library uses abstract classes for defining and maintaining relations between objects. One type of responsibility is creating such objects. The library knows when an object needs to be created, but not what kind of object it should create, this being specific to the application using the library.

The Factory method works just the same way: it defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time. A simple real life example of the Factory Method is the hotel. When staying in a hotel you first have to check in. The person working at the front desk will give you a key to your room after you've paid for the room you want and this way he can be looked at as a "room" factory. While staying at the hotel, you might need to make a phone call, so you call the front desk and the person there will connect you with the number you need, becoming a "phone-call" factory, because he controls the access to calls, too.

Intent

- Defines an interface for creating objects, but let subclasses to decide which class to instantiate
- Refers to the newly created object through a common interface

Implementation

The pattern basically works as shown below, in the UML diagram:

The participants classes in this pattern are:

- Product defines the interface for objects the factory method creates.
- ConcreteProduct implements the Product interface.
- Creator(also referred as Factory because it creates the Product objects) declares the method FactoryMethod, which returns a Product object. May call the generating method for creating Product objects
- ConcreteCreator overrides the generating method for creating ConcreteProduct objects

All concrete products are subclasses of the Product class, so all of them have the same basic implementation, at some extent. The Creator class specifies all standard and generic behavior of the products and when a new product is needed, it sends the creation details that are supplied by the client to the ConcreteCreator. Having this diagram in mind, it is easy for us now to produce the code related to it. Here is how the implementation of the classic Factory method should look:

```
public interface Product { ... }

public abstract class Creator
{
    public void anOperation()
    {
        Product product = factoryMethod();
    }

    protected abstract Product factoryMethod();
}

public class ConcreteProduct implements Product { ... }

public class ConcreteCreator extends Creator
{
    protected Product factoryMethod()
    {
        return new ConcreteProduct();
    }
}

public class Client
{
    public static void main( String arg[] )
    {
        Creator creator = new ConcreteCreator();
        creator.anOperation();
    }
}
```

Applicability & Examples

The need for implementing the Factory Method is very frequent. The cases are the ones below:

- when a class can't anticipate the type of the objects it is supposed to create
- when a class wants its subclasses to be the ones to specify the type of a newly created object

Example 1 - Documents Application.

Take into consideration a framework for desktop applications. Such applications are meant to work with documents. A framework for desktop applications contains definitions for operations such as opening, creating and saving a document. The basic classes are abstract ones, named Application and Document, their clients having to create subclasses from them in order to define their own applications. For generating a drawing application, for example, they need to define the DrawingApplication and DrawingDocument classes. The Application class has the task of managing the documents, taking action at the request of the client (for example, when the user selects the open or save command from the menu).

Because the Document class that needs to be instantiated is specific to the application, the Application class does not know it in advance, so it doesn't know what to instantiate, but it does know when to instantiate it. The framework needs to instantiate a certain class, but it only knows abstract classes that can't be instantiated.

The Factory Method design pattern solves the problem by putting all the information related to the class that needs to be instantiated into an object and using them outside the framework, as you can see below

In the Application class the CreateDocument method either has a default implementation or it doesn't have any implementation at all, this operation being redefined in the MyApplication subclass so that it creates a MyDocument object and returns a reference to it.

```
public Document CreateDocument(String type){
    if (type.isEqual("html"))
        return new HtmlDocument();
    if (type.isEqual("proprietary"))
        return new MyDocument();
    if (type.isEqual("pdf"))
        return new PdfDocument ();
}
```

Assuming that the Application class has a member called docs that represents a list of documents being handled by the application, then the NewDocument method should look like this:

```
public void NewDocument(String type){
    Document doc=CreateDocument(type);
    Docs.add(doc);
    Doc.Open();
}
```

This method will be inherited by the MyApplication class and, so, through the CreateDocument method, it will actually

instantiate MyDocument objects. We will call the CreateDocument method a Factory Method because it is responsible with 'making' an object. Through this method, redefined in Application's subclasses, we can actually shape the situation in which the Application class creates objects without knowing their type. From this point of view the factory method is pattern which provides us a way to achieve the DIP principle.

Specific problems and implementation

When implementing the Factory Method design pattern some issues may appear:

Definition of Creator class

If we apply the pattern to an already written code there may be problems with the way we have the Creator class already defined. There are two cases:

- Creator class is abstract and generating method does not have any implementation. In this case the ConcreteCreator classes must define their own generation method and this situation usually appears in the cases where the Creator class can't foresee what ConcreteProduct it will instantiate.
- Creator class is a concrete class, the generating method having a default implementation. If this happens, the ConcreteCreator classes will use the generating method for flexibility rather than for generation. The programmer will always be able to modify the class of the objects that the Creator class implicitly creates, redefining the generation method.

Factory method is just a particular case of the factory design pattern. In the same time it is the most known factory pattern, maybe because it was published in the GoF. In modern programming languages the factory with registration is more used.

Drawbacks and Benefits

Here are the benefits and drawbacks of factory method pattern:

- + The main reason for which the factory pattern is used is that it introduces a separation between the application and a family of classes (it introduces weak coupling instead of tight coupling hiding concrete classes from the application). It provides a simple way of extending the family of products with minor changes in application code.
- + It provides customization hooks. When the objects are created directly inside the class it's hard to replace them by objects which extend their functionality. If a factory is used instead to create a family of objects the customized objects can easily replace the original objects, configuring the factory to create them.
- - The factory has to be used for a family of objects. If the classes doesn't extend common base class or interface they can not be used in a factory design template.

Hot Points:

The factory method is one of the most used and one of the more robust design patterns. There are only few points which have to be considered when you implement a factory method.

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. I you're code should have a lot of code like the following, reconsider it.

```
if (genericProduct instanceof ConcreteProduct)
    ((ConcreteProduct)genericProduct).doSomeConcreteOperation();
```