Design Patterns | Set 2 (Factory Method)

Difficulty Level : Easy • Last Updated : 07 Sep, 2018

Factory method is a creational design pattern, i.e., related to object creation. In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object.
The idea is to use a static member-function (static factory method) which creates & returns instances, hiding the details of class modules from user.

A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library(explained below) in a way such that it doesn't have tight coupling with the class hierarchy of the library.

What is meant when we talk about library and clients?
A library is something which is provided by some third party which exposes some public APIs and clients make calls to those public APIs to complete its task. A very simple example can be different kinds of Views provided by Android OS.

Why factory pattern?
Let us understand it with an example:

```cpp
// A design without factory pattern
#include <iostream>
using namespace std;

// Library classes
class Vehicle {
public:
    virtual void printVehicle() = 0;
};
class TwoWheeler : public Vehicle {
public:
    void printVehicle()  {
        cout << "I am two wheeler" << endl;
    }
};
class FourWheeler : public Vehicle {
    public:
    void printVehicle()  {
        cout << "I am four wheeler" << endl;
    }
};

// Client (or user) class
class Client {
public:
    Client(int type)  {

        // Client explicitly creates classes according to type
        if (type == 1)
            pVehicle = new TwoWheeler();
        else if (type == 2)
            pVehicle = new FourWheeler();
        else
            pVehicle = NULL;
    }

    ~Client()   {
        if (pVehicle)
        {
            delete[] pVehicle;
            pVehicle = NULL;
        }
    }

    Vehicle* getVehicle() {
        return pVehicle;
    }
private:
    Vehicle *pVehicle;
};

// Driver program
int main() {
    Client *pClient = new Client(1);
    Vehicle * pVehicle = pClient->getVehicle();
    pVehicle->printVehicle();
    return 0;
}
```

Output:

```
 I am two wheeler
```

What is the problems with above design?

As you must have observed in the above example, Client creates objects of either TwoWheeler or FourWheeler based on some input during constructing its object. Say, library introduces a new class ThreeWheeler to incorporate three wheeler vehicles also. What would happen? Client will end up chaining a new else if in the conditional ladder to create objects of ThreeWheeler. Which in turn will need Client to be recompiled. So, each time a new change is made at the library side, Client would need to make some corresponding changes at its end and recompile the code. Sounds bad? This is a very bad practice of design.

How to avoid the problem?

The answer is, create a static (or factory) method. Let us see below code.

```
// C++ program to demonstrate factory method design pattern
#include <iostream>
using namespace std;

enum VehicleType {
    VT_TwoWheeler,    VT_ThreeWheeler,    VT_FourWheeler
};

// Library classes
class Vehicle {
public:
    virtual void printVehicle() = 0;
    static Vehicle* Create(VehicleType type);
};
class TwoWheeler : public Vehicle {
public:
    void printVehicle() {
        cout << "I am two wheeler" << endl;
```

```
    }
```

```
};
class FourWheeler : public Vehicle {
    public:
    void printVehicle() {
        cout << "I am four wheeler" << endl;
    }
};

// Factory method to create objects of different types.
// Change is required only in this function to create a new object type
Vehicle* Vehicle::Create(VehicleType type) {
    if (type == VT_TwoWheeler)
        return new TwoWheeler();
    else if (type == VT_ThreeWheeler)
        return new ThreeWheeler();
    else if (type == VT_FourWheeler)
        return new FourWheeler();
    else return NULL;
}

// Client class
class Client {
public:

    // Client doesn't explicitly create objects
    // but passes type to factory method "Create()"
    Client()
    {
        VehicleType type = VT_ThreeWheeler;
        pVehicle = Vehicle::Create(type);
    }
    ~Client() {
        if (pVehicle) {
            delete[] pVehicle;
            pVehicle = NULL;
        }
    }
    Vehicle* getVehicle()  {
        return pVehicle;
    }

private:
    Vehicle *pVehicle;
};

// Driver program
int main() {
    Client *pClient = new Client();
    Vehicle * pVehicle = pClient->getVehicle();
    pVehicle->printVehicle();
    return 0;
}
```

Output:

```
I am three wheeler
```

In the above example, we have totally decoupled the selection of type for object creation from Client. The library is now responsible to decide which object type to create based on an input. Client just needs to make call to library's factory Create method and pass the type it wants without worrying about the actual implementation of creation of objects.

Thanks to Rumplestiltskin for providing above explanation.

Other examples of Factory Method:

1. Say, in a 'Drawing' system, depending on user's input, different pictures like square, rectangle, circle can be drawn. Here we can use factory method to create instances depending on user's input. For adding new type of shape, no need to change client's code.
2. Another example: In travel site, we can book train ticket as well bus tickets and flight ticket. In this case user can give his travel type as 'bus', 'train' or 'flight'. Here we have an abstract class 'AnyTravel' with a static member function 'GetObject' which depending on user's travel type, will create & return object of 'BusTravel' or ' TrainTravel'. 'BusTravel' or ' TrainTravel' have common functions like passenger name, Origin, destinationparameters.

Thanks to Abhijit Saha providing the first 2 examples.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

RECOMMENDED ARTICLES

Page : 1 2 3