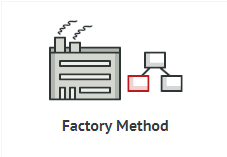
**FACTORY METHOD**

****

**1. Problem:**

Allowing clients to create objects vehicle with different numbers of wheels.

**2. Solution without using pattern:**

// 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;

}

**3. Disadvantages of not using pattern:**

If the library introduces a new class ThreeWheeler, client will end up chaining a new else if in the conditional ladder to create objects of ThreeWheeler. So, client will need to be recompiled.

That means 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.

**4. What is factory method:**

Factory method is a creational design pattern that define an interface for creating an object in a superclass, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

Factory method make a design more customizable and a little more complicated. Other design patterns require new classes, whereas factory method only requires a new operation. Factory method is similar to abstract factory but without the emphasis on families.

**5. Solution using pattern:**

// 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 ThreeWheeler : public Vehicle {

public:

void printVehicle() {

cout << "I am three 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() {

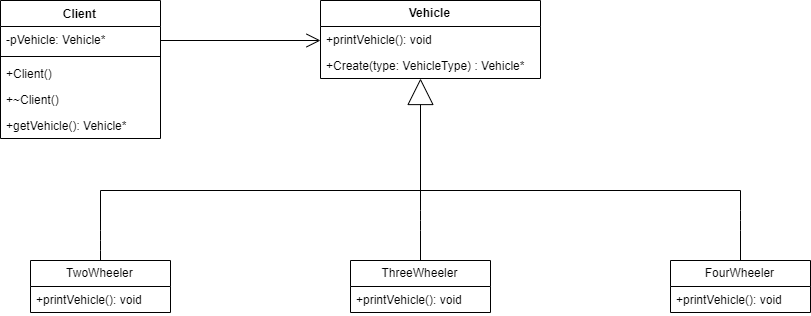
Client \*pClient = new Client();

Vehicle \* pVehicle = pClient->getVehicle();

pVehicle->printVehicle();

return 0;

}



**6. Extra problems using factory method:**

+ In a “drawing” system, different pictures like square, rectangle, circle can be drawm depending on user’s input. We can use factory method to create instances depending on user’s input. There is no need to change client’s code when adding new type of shape.

+ In travelling, we can book train or bus or flight. We can 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.

**7. Pros and cons of factory method:**

Pros:

+ You avoid tight coupling between the creator and the concrete pruduct.

+ You can move the product creation code into one place in the program, making the code easier to support

+ You can introduce new types of products into the program without breaking existing client code.

Cons:

+The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you are introducing the pattern into an existing hierarchy of creator classes.

**References:**

+ <https://sourcemaking.com/design_patterns/factory_method/cpp/1>

+ <https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/>