Problem No: 02
Problem Name: Implementation of Abstract Factory Method Design Pattern

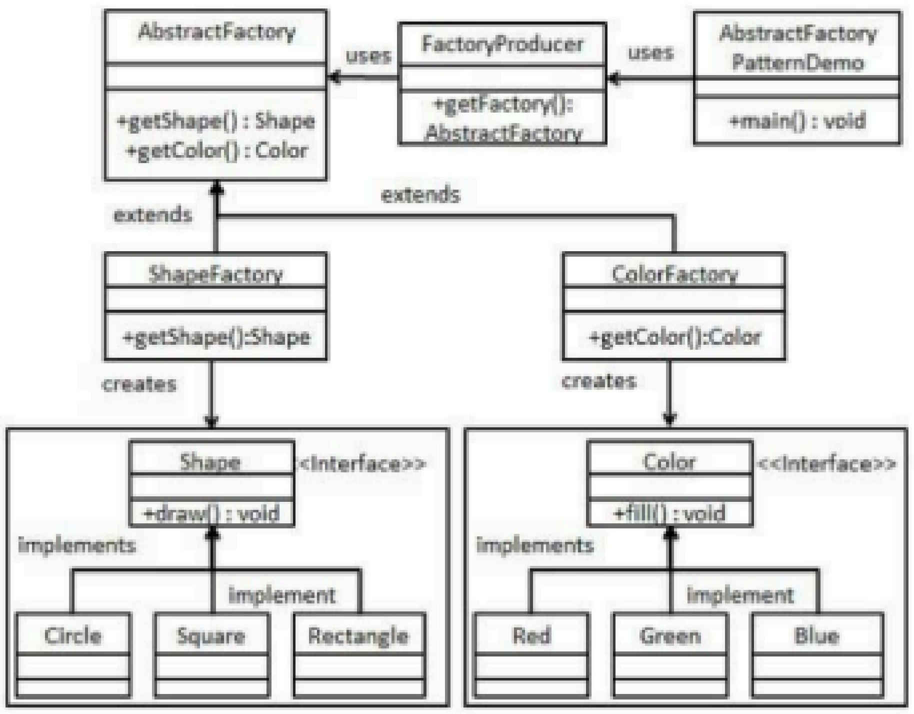
Objectives:

- To understand the Abstract Factory design pattern.
- To implement the Abstract Factory pattern using C++/Java.
- To design a class structure that allows the creation of families of related objects without specifying their concrete classes.

Theory:
The Abstract Factory Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It helps in ensuring loose coupling and enhances scalability.

Consider a graphic design application where users can create shapes (like circles, squares, rectangles) and assign them colors (red, green, blue). Instead of hardcoding object creation, an Abstract Factory can dynamically generate shapes and colors based on the user's selection.

UML Diagram:



Program Implementation (C++)

```
#include <iostream>
#include <memory>
using namespace std;

// Step 1: Abstract Product Interfaces
```

```
class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() {}
};

class Color {
public:
    virtual void fill() = 0;
    virtual ~Color() {}
};

// Step 2: Concrete Products
class Circle : public Shape {
public:
    void draw() override { cout << "Drawing a Circle!" << endl; }
};

class Square : public Shape {
public:
    void draw() override { cout << "Drawing a Square!" << endl; }
};

class Rectangle : public Shape {
public:
    void draw() override { cout << "Drawing a Rectangle!" << endl; }
};

class Red : public Color {
public:
    void fill() override { cout << "Filling with Red color!" << endl; }
};

class Green : public Color {
public:
    void fill() override { cout << "Filling with Green color!" << endl; }
};

class Blue : public Color {
public:
    void fill() override { cout << "Filling with Blue color!" << endl; }
};

// Step 3: Abstract Factory
class AbstractFactory {
public:
    virtual unique_ptr<Shape> getShape(const string& type) { return nullptr; }
    virtual unique_ptr<Color> getColor(const string& type) { return nullptr; }
    virtual ~AbstractFactory() {}
};

// Step 4: Concrete Factories
class ShapeFactory : public AbstractFactory {
public:
    unique_ptr<Shape> getShape(const string& type) override {
        if (type == "Circle") return make_unique<Circle>();
        else if (type == "Square") return make_unique<Square>();
        else if (type == "Rectangle") return make_unique<Rectangle>();
        else return nullptr;
    }
};

class ColorFactory : public AbstractFactory {
public:
    unique_ptr<Color> getColor(const string& type) override {
        if (type == "Red") return make_unique<Red>();
        else if (type == "Green") return make_unique<Green>();
        else if (type == "Blue") return make_unique<Blue>();
        else return nullptr;
    }
};

// Step 5: Factory Producer
class FactoryProducer {
public:
    static unique_ptr<AbstractFactory> getFactory(const string& choice) {
        if (choice == "Shape") return make_unique<ShapeFactory>();
        else if (choice == "Color") return make_unique<ColorFactory>();
        else return nullptr;
    }
};

// Step 6: Client Code
int main() {
    unique_ptr<AbstractFactory> shapeFactory =
    FactoryProducer::getFactory("Shape"); unique_ptr<Shape> shape1 =
    shapeFactory->getShape("Circle");
    shape1->draw();

    unique_ptr<AbstractFactory> colorFactory =
    FactoryProducer::getFactory("Color"); unique_ptr<Color> color1 =
    colorFactory->getColor("Red");
    color1->fill();

    return 0;
}
```

Result and Discussion:

```
Drawing a Circle!
Filling with Red color!
```

- The Abstract Factory Method successfully creates different objects dynamically based on the selected factory.
- This implementation follows the Open-Closed Principle, allowing easy addition of new shapes and colors without modifying existing code.
- It promotes code reusability and maintainability by separating object creation logic from the client code.

Applications:

- GUI Toolkits: Used to create platform-independent UI components like buttons and themes.
- Game Development: Helps in dynamically generating characters, backgrounds, and skins.
- Database Drivers: Allows applications to switch between different database engines without changing core logic.
- Cloud Services: Helps in selecting appropriate services (AWS, Azure, Google Cloud) dynamically.

Conclusion:

The Abstract Factory Pattern is useful when dealing with multiple families of related objects. It enhances code modularity, ensures scalability, and maintains flexibility in software development.