Design Patterns

Creational Patterns - Factory Method

By: Alpesh Chaudhari

This document covers the Factory Method, a key concept in Creational Design Patterns. These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Factory Method

The Factory Method is a design pattern that lets a class create objects, but it allows its subclasses to decide which specific type of object to create.

Problem

Imagine you're building a food delivery app. Initially, your app only supports restaurant orders, so all your code revolves around the RestaurantOrder class.
As your app grows in popularity, you start receiving requests from grocery stores and pharmacies, asking to support grocery and medicine deliveries in your app.

Great news, right? But what about the code? Currently, most of your code is tightly coupled to the RestaurantOrder class. Adding support for grocery or pharmacy orders would require changes throughout the entire codebase.

Moreover, if you decide to add another type of delivery in the future—like flower or electronics deliveries—you'd likely have to repeat these changes all over again.

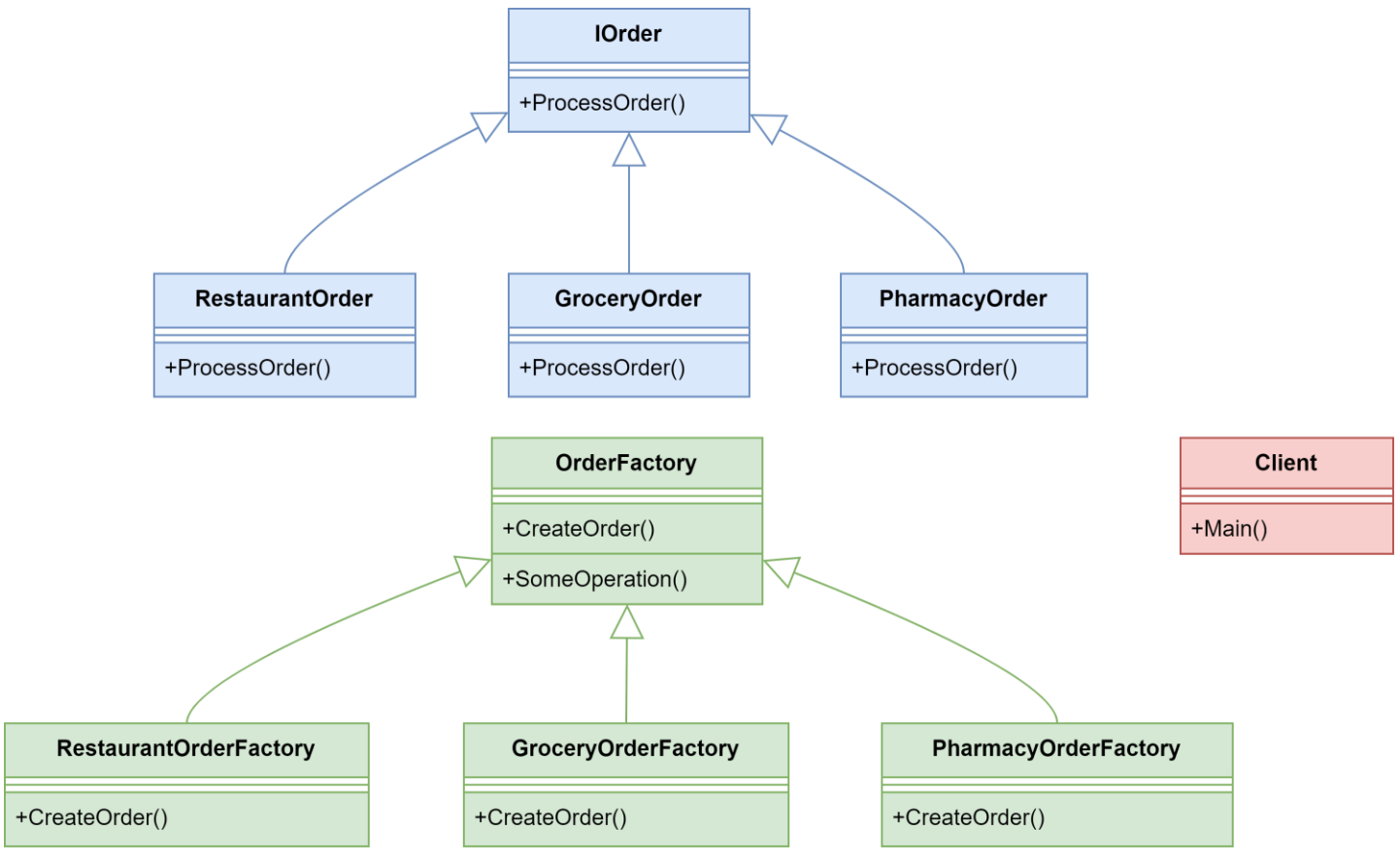
As a result, your code would become messy, filled with conditionals that adjust the app's behavior depending on the type of order, making it harder to maintain and extend.

Solution Using Factory Method Design Pattern

To address the challenges of adding new types of orders in your food delivery app while minimizing code changes and maintaining clean architecture, you can implement the **Factory Method** design pattern. This will help you create a flexible and extensible code structure.

Class Diagram

Here's a class diagram for the Factory Method pattern based on the code structure we discussed for the food delivery application. The diagram illustrates the relationships between the various classes and interfaces.



Here's how you can structure your code:

1.Define an Order Interface

Create an interface (IOrder) that declares the common operation (e.g., ProcessOrder()) that all order types must implement. This ensures that every order class has a standard method for processing orders.

```
public interface IOrder
{
    string ProcessOrder();
}
```

2. Create Concrete Order Classes

Implement specific order classes for each type of order (e.g., RestaurantOrder, GroceryOrder, PharmacyOrder). Each class will implement the ProcessOrder method according to its specific requirements.

```
public class RestaurantOrder : IOrder
{
    public string ProcessOrder()
    {
        return "Processing restaurant order.";
    }
}

public class GroceryOrder : IOrder
{
    public string ProcessOrder()
    {
        return "Processing grocery order.";
    }
}

public class PharmacyOrder : IOrder
{
    public string ProcessOrder()
    {
        return "Processing pharmacy order.";
    }
}
```

3. Create an Order Factory

Define an abstract factory class (OrderFactory) that declares the factory method (CreateOrder()). Concrete factory classes will implement this method to create instances of specific order types.

```
public abstract class OrderFactory
{
    public abstract IOrder CreateOrder();

    public string SomeOperation()
    {
        var order = CreateOrder();
        return "OrderFactory: The factory has created an order: " + order.ProcessOrder();
    }
}
```

4. Implement Concrete Factory Classes

Create specific factories for each order type that will return instances of their respective orders.

```
public class RestaurantOrderFactory : OrderFactory
{
    public override IOrder CreateOrder()
    {
        return new RestaurantOrder();
    }
}

public class GroceryOrderFactory : OrderFactory
{
    public override IOrder CreateOrder()
    {
        return new GroceryOrder();
    }
}

public class PharmacyOrderFactory : OrderFactory
{
    public override IOrder CreateOrder()
    {
        return new PharmacyOrder();
    }
}
```

5. Client Code

The client code uses the factory classes to create orders without needing to know about the specific classes being instantiated. This keeps the client code clean and flexible.

```
public class Client
{
    public void Main()
    {
        OrderFactory factory;

        // Assume we get the type of order from user input or configuration.
        factory = new RestaurantOrderFactory();
        Console.WriteLine(factory.SomeOperation());

        factory = new GroceryOrderFactory();
        Console.WriteLine(factory.SomeOperation());

        factory = new PharmacyOrderFactory();
        Console.WriteLine(factory.SomeOperation());
    }
}
```

Advantages:

1. Reduced Coupling: The order factories and order types are less dependent on each other, making the code easier to manage.
2. Single Responsibility Principle: All order creation code is centralized, which simplifies maintenance and updates.
3. Open/Closed Principle: You can easily add new order types without changing existing code, allowing for flexibility and growth.

Disadvantages:

1. Increased Complexity: Implementing the pattern can lead to more classes, which may make the code harder to understand.

2. Best for Existing Hierarchies: **:** It works best if there's already a good structure in place, which means you might need to plan carefully before using it.

Thank You

I would like to express my gratitude to the following individuals and resources that helped me in understanding and implementing the Factory Method design pattern:

References

1. **Dive into Design patterns** by Alexander Shvets. Refactoring.Guru, 2021.
2. **Desing Patterns**. Refactoring Guru, https://refactoring.guru/