

Factory Pattern

LCSCI5202: Object Oriented Design Week 8

Design Patterns

- Reusable solutions to common software design problems.
- Improve code reusability, scalability, and maintainability
- Creational, Structural, and Behavioral patterns.

Factory Pattern

- A creational pattern that uses a method to create instances of different classes.
- Encapsulate object creation to keep client code independent of specific classes.
 - Simplifies object creation.
 - Reduces dependencies between classes.

When to Use

- Ideal Scenarios
 - When exact types of objects aren't known until runtime.
 - When object creation logic is complex or requires additional steps.
 - When you want a flexible and reusable way to create related objects.
- Example Use Case: Creating a character in a game where the type depends on user choice (e.g., Mage, Warrior).

Scenario: Creating Different Types of Vehicles

```
public Vehicle CreateVehicle(string type)
{
    if (type == "car") {
        return new Car();
    }
    else if (type == "truck")
    { return new Truck();
    }
    else if (type == "motorcycle") {
        return new Motorcycle();
    }
    return null; // What happens when we add new types? }
}
```

Problems with This Approach

- Violates Open/Closed Principle
- Hard to maintain and extend
- Tight coupling between client and concrete classes
- Code duplication when multiple creation points exist

Types of Factory Patterns

- Simple Factory
 - Static method approach
- Factory Method
 - Uses inheritance
 - Subclasses decide instantiation
- Abstract Factory
 - Factory of factories
 - Creates families of related objects
 - Most complex variation

Basic Structure

- Product (interface/abstract class)
- ConcreteProduct (e.g., ProductA, ProductB)
- Creator (Factory with CreateProduct method)
- ConcreteCreator (specific implementation of CreateProduct)

Abstract creator

```
public abstract class PizzaStore {  
    public Pizza OrderPizza(string type) {  
        Pizza pizza = CreatePizza(type);  
  
        pizza.Prepare();  
        pizza.Bake();  
        pizza.Cut();  
        pizza.Box();  
  
        return pizza;  
    }  
  
    protected abstract Pizza CreatePizza(string type);  
}
```

Concrete Creator

```
public class NYPizzaStore : PizzaStore {
    protected override Pizza CreatePizza(string type) {
        if (type == "cheese") return new NYStyleCheesePizza();
        return null;
    }
}

public class LondonPizzaStore : PizzaStore {
    protected override Pizza CreatePizza(string type) {
        if (type == "cheese") return new LondonStyleCheesePizza();
        return null;
    }
}
```

Abstract Product

```
public abstract class Pizza {  
    public abstract string Name { get; }  
    public virtual void Prepare() => Console.WriteLine("Preparing " + Name);  
    public virtual void Bake() => Console.WriteLine("Bake for 25 minutes at 350");  
    public virtual void Cut() => Console.WriteLine("Cutting the pizza into diagonal slices");  
    public virtual void Box() => Console.WriteLine("Place pizza in official PizzaStore box");  
}
```

Concrete Product

```
public class NYStyleCheesePizza : Pizza {  
    public override string Name => "NY Style Sauce and Cheese Pizza";  
}  
  
public class LondonStyleCheesePizza : Pizza {  
    public override string Name => "London Style Deep Dish Cheese Pizza";  
  
    public override void Cut() => Console.WriteLine("Cutting the pizza into square slices");  
}
```

Client Code

```
using System;
namespace FactoryPatternDemo
{
    class Program
    {
        private static void Main()
        {
            // 1) Ask which store (creator) to use
            Console.WriteLine("Choose store (ny / london): ");
            string? storeInput = Console.ReadLine()?.Trim().ToLower();

            PizzaStore store;
            if (storeInput == "ny")
            {
                store = new NYPizzaStore();
            }
            else if (storeInput == "london")
            {
                store = new LondonPizzaStore();
            }
        }
    }
}
```

```
else
{
    Console.WriteLine("Unknown store. Try 'ny' or 'london'.");
    return;
}

Console.WriteLine("Choose pizza type (e.g., cheese): ");
string? type = Console.ReadLine()?.Trim().ToLower();
if (string.IsNullOrEmpty(type))
{
    Console.WriteLine("Please enter a pizza type.");
    return;
}

Pizza pizza = store.OrderPizza(type);

if (pizza == null)
{
    Console.WriteLine($"Sorry, '{type}' is not available at this store.");
    return;
}

Console.WriteLine($"Ordered: {pizza.Name}");
}}}
```

Summary of Methods

- **Abstract Creator** defines the factory method.
- **Concrete Creator** subclasses implement specific product creation.
- **Abstract Product** defines the interface for all products.
- **Concrete Product** subclasses provide specialized implementations for specific types of products.

Gaming Character Factory

You're building a multiplayer RPG game.

- Players can create different character classes, each with unique abilities, stats, and equipment.
- What kind of Creators and Products would be there?
- Do we need Simple Factory, Factory Methods Pattern, or Abstract Factory

Summary

- Factory is a creational pattern: encapsulates object creation so clients depend on abstractions, not concrete classes.
- Flavours: Simple Factory (static helper), Factory Method (subclasses decide), Abstract Factory (families of related objects).
- Core roles: Product (interface/abstract), ConcreteProduct(s), Creator (defines factory method), ConcreteCreator(s).
- Benefits: centralised construction, easier testing/substitution, consistent lifecycles.
Costs: extra indirection/boilerplate.

Activity

Apply Factory Pattern to you Pet Store. Update your UML class diagram.