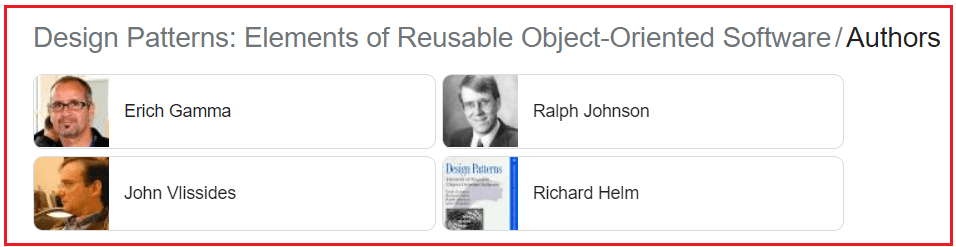
**Design Patterns in C# With Real-time Examples**

**History and Evolution of Design Patterns**

The four authors of the book, famously known as the Gang of Four, introduced the concepts of design patterns in their book Elements of Reusable Object-Oriented Software. Gang of Four (GOF) divided the book into two parts: the first explains the Pros and Cons of Object-Oriented Programming, and the second describes the Evolution Of 23 Classic Software Design Patterns.



Design Patterns are nothing but, documented and tested solutions for recurring problems in a given context. So, in simple words, Design Patterns are reusable solutions to the problems that developers encounter in our day-to-day programming.

Design Patterns are used to solve the problems of Object Generation and Integration. So, Design patterns are reusable solutions to common problems that occur in software design. They represent best practices and have evolved over time through trial and error by experienced software developers.

Design patterns can be thought of as templates for solving specific design problems rather than finished designs that can be transformed directly into code.

**Design Pattern is not a Silver Bullet.** means that **design patterns are helpful tools but not universal solutions** to all software development problems. They are not guaranteed fixes for poor design or architecture. They don’t replace good problem analysis, requirements gathering, or software design skills. Applying patterns blindly can lead to overengineering or unnecessary complexity.

**Do not Overdo Design Patterns**. is a caution against **excessive or inappropriate use** of design patterns in software development. Overdoing Design pattern causes Unnecessary Complexity, Overengineering, Performance Overhead, Loss of Clarity.

When to Use a Design Pattern:

* When a **clear, recurring problem** matches a pattern.
* When it **simplifies** communication or future maintenance.
* When it **adds flexibility or scalability** in a justified way.

**Note:** Sometimes, we may need to use more than one design pattern to solve the problem for a given context. Every design pattern has pros and cons, so only use the design pattern when you are getting more pros than cons.

**Types of Design Patterns**

Gang of Four (GOF) categorized the Design Pattern into three main categories based on the three problem areas (**Object Creation and Initialization, Structural Changes of Classes and Interfaces, and the Relationship Between Classes and communication Between Objects**) of software architecture. They are as follows.

1. [**Creational Design Pattern**](https://dotnettutorials.net/lesson/creational-design-pattern/) (**Object Creation and Initialization**)
2. [**Structural Design Pattern**](https://dotnettutorials.net/lesson/structural-design-pattern/) (**Structural Changes of Classes, and Interfaces, and the Relationship Between Classes**)
3. [**Behavioural Design Pattern**](https://dotnettutorials.net/lesson/behavioral-design-pattern/) (**Communication Between Objects**)

**Creational Design Patterns:**

The **Creational Design Pattern** deals with **Object Creation and Initialization**. The Creational Design Pattern gives the programmer more flexibility in deciding which objects need to be created for a given case. For example, **Creational design patterns** focus on how objects are created. They help make a system independent of how its objects are instantiated, composed, and represented.

🔑 **Purpose:**

To abstract the instantiation process and make it more flexible and reusable.

**Examples of Creational Design Patterns**

|  |  |  |
| --- | --- | --- |
| **Pattern** | **Focus** | **Example Use Case** |
| Singleton | One instance | Config manager, Logger |
| Factory Method | Subclass decides object to create | Document processing |
| Abstract Factory | Create related objects | UI toolkit (OS-specific widgets) |
| Builder | Step-by-step object construction | Building complex forms, cars |
| Prototype | Copy existing object | Game object cloning, templates |

**Structural Design Patterns:**

The **Structural Design Pattern** is used to Manage the Structure of Classes and Interfaces and**the Relationship Between the Classes and Interfaces**. **i.e. these patterns** are concerned with how classes and objects are composed to form larger structures. They help ensure that **components are connected in a flexible and efficient way**.

**Purpose:**

To simplify relationships between entities and make code more modular and reusable by structuring objects and classes effectively.

|  |  |  |
| --- | --- | --- |
| **Pattern** | **Purpose** | **Use Case Example** |
| Adapter | Convert interface | Legacy system integration |
| Bridge | Separate abstraction from implementation | Shapes with different renderers |
| Composite | Tree structure | Graphics or file systems |
| Decorator | Add behavior dynamically | Logging, notifications |
| Facade | Simplify interface | Simplified API to complex subsystem |
| Flyweight | Share common data | Repeated characters, UI elements |
| Proxy | Control access | Lazy loading, logging, access control |

**Behavioural Design Patterns:**

**Behavioural Design Patterns** deal with the **Communication Between Classes and Objects**. That means if you want to change the behavior of a class again, you want it to affect other classes of the project as well. For example, you have an Invoice class that currently applies taxes as 18%. Tomorrow, if you have to add another extra tax. That means you are changing the behavior of a class. To solve such Behavioural issues, Behavioural Design patterns come into the picture.

**Behavioural Design Patterns include**

|  |  |  |
| --- | --- | --- |
| **Pattern Name** | **Definition (Short)** | **Use Case Example** |
| **Chain of Responsibility** | Pass request along a chain until handled. | Logging, middleware pipelines |
| **Command** | Encapsulate a request as an object. | UI buttons, remote controls |
| **Observer** | Notify multiple objects when a subject change. | Event listeners, notification system |
| **Iterator** | Access elements of a collection sequentially. | Custom data collections |
| **State** | Change object behavior based on internal state. | Media player modes, traffic lights |
| **Template Method** | Define algorithm skeleton, let subclasses define steps. | Data import/export, workflow engines |
| **Visitor** | Add operations to objects without changing them. | AST traversal, report generation |
| **Strategy** | Define a family of interchangeable algorithms. | Sorting, payment methods |
| **Mediator** | Central object controls interaction between components. | Chat rooms, UI controls interaction |
| **Memento** | Capture and restore object state. | Undo/Redo, save games |
| **Interpreter** | Interpret sentences in a custom language or grammar. | Expression parsing, calculators |

So, these patterns are focused on communication between objects: how they interact and fulfil their intended purpose. They define clear patterns of communication among objects.

Along with GoF 23 Design Patterns, following Dot Net Design Patterns, which are used frequently in most Real-Time .NET Applications.

* [**Dependency Injection Design Pattern.**](https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/)
* [**Dependency Injection using Unity Container.**](https://dotnettutorials.net/lesson/unity-container-asp-net-mvc/)
* [**Repository Design Pattern using C#.**](https://dotnettutorials.net/lesson/repository-design-pattern-csharp/)
* [**Repository Design Pattern using Unity of Work.**](https://dotnettutorials.net/lesson/unit-of-work-csharp-mvc/)
* [**Inversion of Control in C#.**](https://dotnettutorials.net/lesson/introduction-to-inversion-of-control/)

**Singleton Design Pattern:**

The **Singleton Pattern** ensures that **only one instance** of a class is created and provides a **global point of access** to that instance.

**“Singleton is creational design pattern that lets you ensure that a class has only one instance, while providing a global point of access to that instance.”**

**Problem: (** <https://www.youtube.com/watch?v=oROIXOLO4TU> **)**

1. **Ensure that a class has just a single instance.**
2. **Provide a global access point to that instance.**

A black background with white rectangles

Description automatically generated

**Solution to above:**

* **Make the default constructor private, to prevent other objects from using the new operator with the singleton class.**
* **Create a static creation method that acts as a constructor.**

A screenshot of a computer screen

Description automatically generated

**Code Example:**

**Singletone.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SingletonDP

{

internal class Singleton

{

private Singleton() { }

private static Singleton instance;

// if used = new Singleton(); then always initializes the instance even

// though not called i.e. not lazy

private static object instanceLock = new object(); //To Create lock

public static Singleton getInstance()

{

//Double Check Locking implemented

if (instance == null) //First check to avoid multi thread waiting for lock release

{

lock (instanceLock)

{

if (instance == null) //if null only then create

{

instance = new Singleton();

}

}

}

return instance;

}

}

}

**Program.cs**

namespace SingletonDP

{

internal class Program

{

static void Main(string[] args)

{

//Singleton abc = new Singleton();//Error inaccessible due to protection level

Singleton inst = Singleton.getInstance();//Direct Access because of static

Console.WriteLine("Hello, World!");

}

}

}

**Use Case Examples:**

* Logging service
* Configuration settings
* Database connections
* Caching systems

working and clean implementation of the **Singleton Design Pattern** for a **Logger service** in **.NET (C#)** — extended to include **file logging** and made **Dependency Injection (DI)**-friendly (i.e., avoids static access).

✅ **Key Features:**

* Singleton ensures **one shared instance** of the logger.
* Writes logs to both **console and file**.
* Compatible with **.NET Core/ASP.NET Core Dependency Injection** (no static or new inside services).

🔧 **Step-by-step Implementation**

**1.** **ILoggerService Interface**

public interface ILoggerService

{

void LogInfo(string message);

void LogError(string message);

}

**2. FileLoggerService Implementation (Singleton + File Logging)**

using System;

using System.IO;

public class FileLoggerService : ILoggerService

{

private static readonly object \_lock = new();

private readonly string \_logFilePath;

public FileLoggerService()

{

\_logFilePath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "log.txt");

}

public void LogInfo(string message)

{

Log($"INFO: {message}");

}

public void LogError(string message)

{

Log($"ERROR: {message}");

}

private void Log(string message)

{

var formatted = $"{DateTime.Now:u} - {message}";

lock (\_lock)

{

// Log to Console

Console.WriteLine(formatted);

// Log to File

File.AppendAllText(\_logFilePath, formatted + Environment.NewLine);

}

}

}

**3. Registering in .NET Core DI Container**

In Startup.cs or Program.cs (for .NET 6+):

builder.Services.AddSingleton<ILoggerService, FileLoggerService>();

AddSingleton ensures the **same instance** is used throughout the app – which fulfills the Singleton pattern behavior **in a DI-friendly way**.

**4.** **Using the Logger in a Service or Controller**

public class OrderService

{

private readonly ILoggerService \_logger;

public OrderService(ILoggerService logger)

{

\_logger = logger;

}

public void PlaceOrder()

{

try

{

// Order processing logic

\_logger.LogInfo("Order placed successfully.");

}

catch (Exception ex)

{

\_logger.LogError("Order failed: " + ex.Message);

}

}

}

✅ Summary: Why This is Better Than a Static Singleton

|  |  |
| --- | --- |
| **Traditional Singleton** | **DI-Friendly Singleton (AddSingleton)** |
| Uses static or private constructor | Uses DI container to manage instance |
| Hard to test/mock | Easy to unit test and replace |
| Hidden dependencies | Explicit dependencies via constructor injection |

Adapter Design pattern:

It is a structural design pattern that allows objects with incompatible interface to collaborate.

The **Adapter Design Pattern** is a structural pattern used to allow two incompatible interfaces to work together. It acts as a bridge between the two interfaces.

Real life example:

A black and orange usb adapter

Description automatically generatedA usb adapter with different connectors

Description automatically generated

🔧 **Purpose**

To convert the interface of a class into another interface that clients expect. It lets classes work together that couldn’t otherwise because of incompatible interfaces.

🧱 **Structure**

**Target**: The interface the client expects.

**Adoptee**: The existing interface that needs adapting.

**Adapter**: Bridges the gap between the **Target** and **Adoptee**.

Imagine that you are creating a stock market monitoring application. The application downloads the stock data from multiple sources in JSON format and then displays nice-looking charts and diagrams for the users.

**Problem:**

* But there’s a catch: the analytics library to display the graphs only with library specific objects.
* Changing the library to work with JSON might break some existing code that relies on the library.
* You might not have access to the library’s source code in the first place, making this approach impossible.

**Solution:**

* You can create an adapter that converts the interface of one object so that another object can understand it.
* The adapter gets an interface, compatible with one of the existing objects.
* Using this interface, the existing object can safely call the adapter’s methods.
* Upon receiving a call, the adapter passes the request to the second object, but in an format and order that the second object expects.

**Structure:**

A black and white rectangular object with text

Description automatically generated

📌 **When to Use**

* You want to use an existing class, but its interface does not match what you need.
* You want to reuse legacy code without modifying it.

Code Example:

**AnalyticsLibrary.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace AdapterDP

{

// Adaptee - incompatible interface

public class AnalyticsLibrary

{

public void DisplayGraph(CustomLibraryObject data)

{

Console.WriteLine($"Graph Title: {data.Title}");

Console.WriteLine("Data Points: " + string.Join(", ", data.DataPoints));

}

}

// Adaptee's expected input object

public class CustomLibraryObject

{

public string Title { get; set; }

public List<int> DataPoints { get; set; }

}

}

**IDataVisualizer.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Newtonsoft.Json;

namespace AdapterDP

{

// Target interface

internal interface IDataVisualizer

{

void DisplayGraph(string JSONData);

}

// Adapter

public class DataFormatAdapter : IDataVisualizer

{

private readonly AnalyticsLibrary \_adaptee;

public DataFormatAdapter(AnalyticsLibrary adaptee)

{

\_adaptee = adaptee;

}

public void DisplayGraph(string JSONData)

{

CustomLibraryObject obj = GetObjectFromJSON(JSONData);

\_adaptee.DisplayGraph(obj);

}

private CustomLibraryObject GetObjectFromJSON(string jsonData)

{

// Convert JSON Data to Object

return JsonConvert.DeserializeObject<CustomLibraryObject>(jsonData);

}

}

}

**Program.cs**

namespace AdapterDP

{

//Client

internal class Program

{

static void Main(string[] args)

{

string json = @"{

'Title': 'Monthly Sales',

'DataPoints': [100, 150, 200, 250]

}";

AnalyticsLibrary analytics = new AnalyticsLibrary();

IDataVisualizer visualizer = new DataFormatAdapter(analytics);

visualizer.DisplayGraph(json); // Adapter in action

}

}

}

**Output:**

A black and white text

Description automatically generated

**Builder Design Pattern:**

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

**Problem:**

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.

Basic solution: either 1 or 2

1. Define constructor with multiple parameters but need to maintain correct data format with exact sequence of data.
2. After creating object set these fields and which is very worst thing in terms of complex class fields.

Consider below code snippet in which we have to pass all the parameters in a sequence with their type no matter user is interested in sending the details or not.

**Ex.** If customer knows everything and mentions every parameters required, then this is useful, but what If customer is not interested in sharing the processor details, we cant do that with the same.

A screen shot of a computer program

Description automatically generated

**Solution:**

* Extract the object construction code out of its own class and move it to separate objects called builders.
* To create an object, you execute a series of steps on a builder object.
* The important part is that you don't need to call all of the steps.
* You can call only those steps that are necessary for producing a particular configuration of an object.

**Structure:**

A black background with white rectangles

Description automatically generated

**Example:**

While buying a cell phone need not to specify every property of cell phone!

**ICellPhoneBuilder.cs**

namespace BuilderDP

{

internal interface ICellPhoneBuilder

{

CellPhone GetCellPhone();

ICellPhoneBuilder setBattery(int battery);

ICellPhoneBuilder setCamera(int camera);

ICellPhoneBuilder setOS(string os);

ICellPhoneBuilder setProcessor(string processor);

ICellPhoneBuilder setScreenSize(double screenSize);

}

}

**CellPhone.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BuilderDP

{

internal class CellPhone

{

private string brand;

private string os;

private string processor;

private double screenSize;

private int battery;

private int camera;

public CellPhone(string brand, string os, string processor, double screenSize, int battery, int camera)

{

this.brand = brand;

this.os = os;

this.processor = processor;

this.screenSize = screenSize;

this.battery = battery;

this.camera = camera;

}

public override string ToString()

{

return

$"Brand:{brand}, " +

$"OS: {os}, " +

$"Processor: {processor}, " +

$"ScreenSize: {screenSize}, " +

$"Battery: {battery}, " +

$"Camera: {camera}";

}

}

}

**ApplePhoneBuilder.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BuilderDP

{

internal class ApplePhoneBuilder : ICellPhoneBuilder

{

private readonly string brand = "Apple";

private string os;

private string processor;

private double screenSize;

private int battery;

private int camera;

public CellPhone GetCellPhone()

{

if (processor == null)

processor = "A18 Pro 6 Core" ;

return new CellPhone(brand, os, processor, screenSize, battery, camera);

}

public ICellPhoneBuilder setBattery(int battery)

{

this.battery = battery;

return this;

}

public ICellPhoneBuilder setCamera(int camera)

{

this.camera = camera;

return this;

}

public ICellPhoneBuilder setOS(string os)

{

this.os = os;

return this;

}

public ICellPhoneBuilder setProcessor(string processor)

{

this.processor = processor;

return this;

}

public ICellPhoneBuilder setScreenSize(double screenSize)

{

this.screenSize = screenSize;

return this;

}

}

}

**SamsungPhoneBuilder.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BuilderDP

{

internal class SamsungPhoneBuilder : ICellPhoneBuilder

{

private readonly string brand = "Samsung";

private string os;

private string processor;

private double screenSize;

private int battery;

private int camera;

//Create Setter Methods for the above fields

public ICellPhoneBuilder setOS(string os)

{

this.os = os;

return this;

}

public ICellPhoneBuilder setProcessor(string processor)

{

this.processor = processor;

return this;

}

public ICellPhoneBuilder setScreenSize(double screenSize)

{

this.screenSize = screenSize;

return this;

}

public ICellPhoneBuilder setBattery(int battery)

{

this.battery = battery;

return this;

}

public ICellPhoneBuilder setCamera(int camera)

{

this.camera = camera;

return this;

}

//Create Method which will create new object by setting above values

//and will return the same

public CellPhone GetCellPhone()

{

return new CellPhone(brand, os, processor, screenSize, battery, camera);

}

}

}

**Program.cs**

namespace BuilderDP

{

internal class Program

{

static void Main(string[] args)

{

SamsungPhoneBuilder builder1 = new SamsungPhoneBuilder();

builder1.setCamera(33);

builder1.setOS("Android");

builder1.setBattery(33000);

CellPhone phone= builder1.GetCellPhone();

Console.WriteLine(phone);

//Single Line Code

CellPhone phone1 = new ApplePhoneBuilder()

.setOS("IOS")

.setBattery(3210)

.setCamera(51)

.GetCellPhone();

Console.WriteLine(phone1);

// Example: Direct instantiation (not recommended if using builder)

//CellPhone phone = new CellPhone("Android","Qualcom",15,3300,33);

//Console.WriteLine(phone);

}

}

}

**Prototype Design Pattern:**

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

In other words

The **Prototype Design Pattern** is a **creational pattern** used to create **duplicate objects** while keeping performance in mind. Instead of creating new instances from scratch (which can be expensive), you **clone existing objects**.

✅ When to Use Prototype Pattern

* When object creation is costly (e.g., involving database or network operations).
* When objects are similar and only small changes are needed.
* When creating an object requires a lot of configuration or setup.

**Ex: Problem:** Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their   
values over to the new object.

But, not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.

Since you have to know the object's class to create a duplicate, your code becomes dependent on that class.

**Solution:**

The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning which contains a clone method. The implementation of the clone method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one.

**Structure:**

A black background with white rectangles

Description automatically generated

* **C#**

Supports cloning through the **ICloneable** Interface. This interface provides a way to create a copy of an object.

* **Java**

Also supports cloning via the Cloneable Interface and the clone() method inherited from the Object class.

However, it's important to handle **CloneNotSupportedException** if a class doesn't support cloning.

* **Python**

Uses the copy module for object cloning. This module offers both shallow and deep copying functionalities.

Cloning objects involves creating a copy of an existing object. This can be useful in various scenarios, such as creating backups or working with copies of data without modifying the original.

Code Example:

**Address.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PrototypeDP

{

//Complex Object

public class Address

{

public string Street { get; set; }

public string City { get; set; }

public Address(string street, string city)

{

Street = street;

City = city;

}

public Address DeepCopy()

{

return new Address(Street, City);

}

}

//Prototype Class

public class Person : ICloneable

{

public int Age { get; set; }

public string Name { get; set; }

public Address Address { get; set; }

public Person(String Name, Address address)

{

this.Name = Name;

this.Address = address;

}

public object Clone() // Method of ICloneable

{

return this.MemberwiseClone();

}

//Shallow Copy (default)

public Person ShallowClone() //Object can also be returned

{

return new Person(this.Name, this.Address) { Age = this.Age };

}

//Deep Copy (Custom)

public Person DeepClone()

{

Person clone = (Person)this.MemberwiseClone();

clone.Address = this.Address.DeepCopy();// Important: deep clone Address separately

return clone; //new Person(Name, Address);

}

public void Display()

{

Console.WriteLine($"Name: {Name}, Street: {Address.Street},City: {Address.City}");

}

}

}

**Program.cs**

using System.Net;

using System;

namespace PrototypeDP

{

internal class Program

{

static void Main(string[] args)

{

// Original object

Person original = new Person("John Doe", new Address("123 Main St", "New York"));

// Shallow copy

Person shallowCopy = (Person)original.ShallowClone();

// Deep copy

Person deepCopy = original.DeepClone();

Console.WriteLine("Before Modification:");

Console.WriteLine("Original");

original.Display();

Console.WriteLine("Shallow");

shallowCopy.Display();

Console.WriteLine("Deep");

deepCopy.Display();

// Modify original object's Address

original.Address.Street = "999 Changed St";

Console.WriteLine("\nAfter Modification:");

Console.WriteLine("Original");

original.Display(); // Address changed

Console.WriteLine("Shallow");

shallowCopy.Display(); // Address also changed (because of shallow copy)

Console.WriteLine("Deep");

deepCopy.Display(); // Address NOT changed (because of deep copy)

Console.ReadKey();

}

}

}

**🧠 Key Takeaways:**

* **Shallow Copy:** Shares reference types like Address → changes affect all.
* **Deep Copy:** Creates a new object with independently cloned fields.

**Factor Design Pattern:**

The **Factory Design Pattern** is a **creational pattern** that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

✅ Use Case:

When you have a superclass with multiple sub-classes, and based on input, you need to return one class instance.

Example of implementing the **Factory Design Pattern using Dependency Injection in ASP.NET Core Web API**, with a clean and real-world use case.

**✅ Scenario: Notification System via Web API**

We'll expose an endpoint:  
**POST** /api/notification/send?type=email  
This will send the appropriate notification based on the type parameter (email, sms, push, etc.)

**📁 Project Structure**

/Controllers

NotificationController.cs

/Interfaces

INotification.cs

INotificationFactory.cs

/Services

EmailNotification.cs

SMSNotification.cs

PushNotification.cs

NotificationFactory.cs

Program.cs

1. **Create Interface and Notification Services**

**/Interfaces/INotification.cs**

public interface INotification

{

string NotifyUser();

}

**/Services/EmailNotification.cs**

public class EmailNotification : INotification

{

public string NotifyUser() => "Email Notification Sent!";

}

**/Services/SMSNotification.cs**

public class SMSNotification : INotification

{

public string NotifyUser() => "SMS Notification Sent!";

}

**/Services/PushNotification.cs**

public class PushNotification : INotification

{

public string NotifyUser() => "Push Notification Sent!";

}

1. **Factory Interface and Implementation**

**/Interfaces/INotificationFactory.cs**

public interface INotificationFactory

{

INotification GetNotification(string type);

}

**/Services/NotificationFactory.cs**

public class NotificationFactory : INotificationFactory

{

private readonly IServiceProvider \_serviceProvider;

public NotificationFactory(IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

public INotification GetNotification(string type)

{

return type.ToLower() switch

{

"email" => \_serviceProvider.GetRequiredService<EmailNotification>(),

"sms" => \_serviceProvider.GetRequiredService<SMSNotification>(),

"push" => \_serviceProvider.GetRequiredService<PushNotification>(),

\_ => throw new ArgumentException("Invalid notification type")

};

}

}

1. **Web API Controller**

### **/Controllers/NotificationController.cs**

using Microsoft.AspNetCore.Mvc;

using PrototypeDP.Interfaces;

namespace PrototypeDP.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class NotificationController : ControllerBase

{

private readonly INotificationFactory \_factory;

public NotificationController(INotificationFactory factory)

{

\_factory = factory;

}

[HttpPost("send")]

public IActionResult SendNotification([FromQuery] string type)

{

try

{

var notification = \_factory.GetNotification(type);

var result = notification.NotifyUser();

return Ok(new { Message = result });

}

catch (ArgumentException ex)

{

return BadRequest(new { Error = ex.Message });

}

}

}

}

1. **Register Services in Program.cs**

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

// Register services for DI

builder.Services.AddTransient<EmailNotification>();

builder.Services.AddTransient<SMSNotification>();

builder.Services.AddTransient<PushNotification>();

builder.Services.AddTransient<INotificationFactory, NotificationFactory>();

var app = builder.Build();

app.UseAuthorization();

app.MapControllers();

app.Run();

**✅ Sample Request**

POST /api/notification/send?type=email

**Response:**

{

"message": "Email Notification Sent!"

}

**🧪 Want to add Unit Tests or Swagger Docs next**

Want to add Unit Tests and Swagger Docs next

Great! Let’s handle both Swagger and Unit Tests step-by-step.

✅ Part 1: **Add Swagger Documentation**

🔧 In Program.cs (Add Swagger Services)

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

**🚀 In Program.cs (Enable Swagger UI)**

var app = builder.Build();

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseAuthorization();

app.MapControllers();

app.Run();

**📦 (Optional) Add NuGet package if not already present:**

dotnet add package Swashbuckle.AspNetCore

Now run the app and open:

https://localhost:<port>/swagger

**✅ Part 2: Add Unit Tests Using xUnit and Moq**

**📦 Add test project**

Run this in your terminal:

dotnet new xunit -n NotificationService.Tests

dotnet add NotificationService.Tests reference YourApiProjectName

dotnet add NotificationService.Tests package Moq

**🧪 Sample Unit Test for Controller**

using Xunit;

using Moq;

using Microsoft.AspNetCore.Mvc;

using YourApi.Controllers;

using YourApi.Interfaces;

using YourApi.Services;

namespace NotificationService.Tests

{

public class NotificationControllerTests

{

**[Fact]**

public void SendNotification\_ReturnsOk\_ForValidType()

{

**// Arrange**

var mockFactory = new Mock<INotificationFactory>();

mockFactory.Setup(f => f.GetNotification("email")).Returns(new EmailNotification());

var controller = new NotificationController(mockFactory.Object);

**// Act**

var result = controller.SendNotification("email") as OkObjectResult;

**// Assert**

Assert.NotNull(result);

Assert.Equal(200, result.StatusCode);

Assert.Contains("Email Notification Sent", result.Value.ToString());

}

**[Fact]**

public void SendNotification\_ReturnsBadRequest\_ForInvalidType()

{

**// Arrange**

var mockFactory = new Mock<INotificationFactory>();

mockFactory.Setup(f => f.GetNotification("invalid")).Throws(new ArgumentException("Invalid notification type"));

var controller = new NotificationController(mockFactory.Object);

**// Act**

var result = controller.SendNotification("invalid") as BadRequestObjectResult;

**// Assert**

Assert.NotNull(result);

Assert.Equal(400, result.StatusCode);

}

}

}

**Simple Example implemented Factory Design Pattern:**

**Problem:**

Imagine that you are at a pizza restaurant and want to order a pizza.

You just want to order a pizza and don't care about the type of pizza as long it's veg.

**Solution:**

* The waiter acts as a pizza provider and the chefs act as pizza factories.
* They create the pizza and provides it to you without you having to worry about
* the type of pizza.   
  In future if the pizza restaurant changes pizza toppings on the pizza you don't need to change your order.

**Structure:**

A black background with white text

Description automatically generated

**Code Example:**

**FactoryPattern.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace FactoryDP

{

public interface IPizza //Iproduct

{

string Eat();

}

//Concrete Implementation VegPizza

class VegPizza : IPizza

{

public string Eat()

{

return "Eating Veg Pizza !!!!";

}

}

//Concrete Implementation NonVegPizza

class NonVegPizza : IPizza

{

public string Eat()

{

return "Eating NonVeg Pizza ): ";

}

}

public interface IPizzaChef //Factory

{

IPizza PreparePizza();

}

class VegPizzaChef : IPizzaChef

{

public IPizza PreparePizza()

{

return new VegPizza();

}

}

class NonVegPizzaChef : IPizzaChef

{

public IPizza PreparePizza()

{

return new NonVegPizza();

}

}

class Waiter //Client Class

{

public IPizza GetPizza(string type)

{

IPizzaChef chef;

switch (type)

{

case "Veg":

chef = new VegPizzaChef();

break;

case "NonVeg":

chef = new NonVegPizzaChef();

break;

default:

throw new ArgumentException("Invalid pizza type");

}

return chef.PreparePizza();

}

}

}

**Program.cs**

namespace FactoryDP

{

internal class Program

{

static void Main(string[] args)

{

Waiter waiter = new Waiter();

IPizza pizza = waiter.GetPizza("Veg");

Console.WriteLine(pizza.Eat());

pizza = waiter.GetPizza("NonVeg");

Console.WriteLine(pizza.Eat());

//pizza = waiter.GetPizza("MashroomVeg"); //Will Throw Exception

//Console.WriteLine(pizza.Eat());

//Or

Console.WriteLine("Enter the type of pizza (Veg/NonVeg):");

Console.WriteLine(waiter.GetPizza(Console.ReadLine()).Eat());

}

}

}

**Output:**

A black and white text

Description automatically generated

Use Cases:

* Usually used for third party library integration.
* Ex. Stock market feed provider, Cricket live score provider
* To prevent changes to core logic when adding or removing library.