Design Patterns গুলো Software Engineering-এ খুব গুরুত্বপূর্ণ একটি বিষয়, এবং C# দিয়ে Object-Oriented Design করতে গেলে এগুলো জানা অত্যন্ত প্রয়োজন। নিচে Design Patterns-এর কিছু গুরুত্বপূর্ণ ক্যাটাগরি এবং বাংলা ব্যাখ্যা দেওয়া হলো।

**🔶 Design Pattern কী?**

Design Pattern হলো কিছু সাধারণ সমস্যার জন্য বারবার ব্যবহারযোগ্য সমাধান। এগুলো হলো Best Practice যেগুলো পূর্বে অনেক সফটওয়্যার ইঞ্জিনিয়ারদের দ্বারা ব্যবহার করা হয়েছে এবং পরীক্ষিত।

Design Patterns সাধারণত ৩টি ভাগে ভাগ করা হয়:

**1. Creational Patterns (উৎপাদন সম্পর্কিত প্যাটার্ন)**

**উদ্দেশ্য:** Object তৈরি করার পদ্ধতি সহজ এবং নিয়ন্ত্রিত করা।

**✅ Singleton Pattern**

একটা ক্লাসের শুধুমাত্র একটি instance থাকবে এবং global অ্যাক্সেস থাকবে।

**উদাহরণ:**

public class Singleton

{

private static Singleton instance = null;

private Singleton() { }

public static Singleton Instance

{

get

{

if (instance == null)

instance = new Singleton();

return instance;

}

}

}

➡️ **ব্যবহার:** Logger, Configuration Manager

**✅ Factory Method Pattern**

ক্লায়েন্ট কোন ক্লাস instantiate করবে সেটা না জানিয়ে, অবজেক্ট তৈরির কাজ একটা Factory ক্লাস করবে।

public interface IShape

{

void Draw();

}

public class Circle : IShape

{

public void Draw() => Console.WriteLine("Circle Drawing");

}

public class ShapeFactory

{

public IShape GetShape(string shapeType)

{

if (shapeType == "Circle")

return new Circle();

// অন্যান্য shape

return null;

}

}

➡️ **ব্যবহার:** যখন subclass কোন object তৈরি করবে তা নির্ধারণ করবে।

**2. Structural Patterns (গঠনমূলক প্যাটার্ন)**

**উদ্দেশ্য:** ক্লাস ও অবজেক্ট গুলোর মধ্যে বড় আকারের কাঠামো তৈরি করা।

**✅ Adapter Pattern**

দুইটি incompatible interface-কে compatible করে তোলে।

**উদাহরণ:**

public interface ITarget

{

void Request();

}

public class Adaptee

{

public void SpecificRequest() => Console.WriteLine("Specific Request");

}

public class Adapter : ITarget

{

private Adaptee \_adaptee = new Adaptee();

public void Request()

{

\_adaptee.SpecificRequest(); // Adapting

}

}

➡️ **ব্যবহার:** যখন পুরনো কোড নতুন সিস্টেমে ব্যবহার করতে হয়।

**3. Behavioral Patterns (আচরণমূলক প্যাটার্ন)**

**উদ্দেশ্য:** অবজেক্টগুলোর মধ্যে যোগাযোগ এবং আচরণ নিয়ন্ত্রণ করা।

**✅ Observer Pattern**

একটি অবজেক্টে পরিবর্তন হলে অন্য অবজেক্টগুলোকে স্বয়ংক্রিয়ভাবে জানিয়ে দেওয়া হয়।

**উদাহরণ:**

public interface IObserver

{

void Update();

}

public class ConcreteObserver : IObserver

{

public void Update() => Console.WriteLine("Updated");

}

public class Subject

{

private List<IObserver> observers = new();

public void Attach(IObserver observer) => observers.Add(observer);

public void Notify()

{

foreach (var observer in observers)

observer.Update();

}

}

➡️ **ব্যবহার:** Notification system, Event handling

অসাধারণ! এবার আমরা **Creational Design Patterns** নিয়ে বিস্তারিতভাবে বাংলায় আলোচনা করবো। Creational Patterns গুলো মূলত **অবজেক্ট তৈরি করার প্রক্রিয়া**কে নিয়ন্ত্রণ করে, যাতে সফটওয়্যারের **ফ্লেক্সিবিলিটি এবং পুনঃব্যবহারযোগ্যতা (reusability)** বাড়ে।

**🔷 Creational Design Patterns — বিস্তারিত তালিকা (C# + বাংলা ব্যাখ্যা)**

**১️⃣ Singleton Pattern (সিঙ্গলটন প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

একটি ক্লাসের শুধুমাত্র একটি অবজেক্ট (instance) তৈরি হবে, এবং সেই অবজেক্টটি global ভাবে অ্যাক্সেসযোগ্য হবে।

📌 বিস্তারিত ব্যাখ্যা আগেই দেওয়া হয়েছে।  
[✅ উপরে স্ক্রল করে দেখে নিন বা আমি চাইলে আবার সংক্ষেপে দিতে পারি।]

**২️⃣ Factory Method Pattern (ফ্যাক্টরি মেথড প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

Subclasses-এর উপর নির্ভর করে object creation-এর দায়িত্ব দেয়া।

**🧠 কেন দরকার?**

যখন আপনি জানেন না ঠিক কোন ক্লাস instantiate করতে হবে, তখন Factory Method ব্যবহৃত হয়।

**📦 বাস্তব উদাহরণ:**

আপনার একটি সফটওয়্যার আছে যেখানে বিভিন্ন ধরনের Notification পাঠাতে হয় (Email, SMS, Push)। কোন টাইপের Notification লাগবে তা Runtime-এ নির্ধারণ হবে।

**🧪 C# কোড:**

// Step 1: Interface তৈরি

public interface INotification

{

void Notify();

}

// Step 2: বিভিন্ন ধরনের Notification ক্লাস

public class EmailNotification : INotification

{

public void Notify() => Console.WriteLine("ইমেইল পাঠানো হয়েছে");

}

public class SMSNotification : INotification

{

public void Notify() => Console.WriteLine("এসএমএস পাঠানো হয়েছে");

}

// Step 3: Factory ক্লাস

public class NotificationFactory

{

public INotification CreateNotification(string type)

{

if (type == "EMAIL")

return new EmailNotification();

else if (type == "SMS")

return new SMSNotification();

else

throw new Exception("Invalid Notification Type");

}

}

**✅ ব্যবহার:**

class Program

{

static void Main()

{

var factory = new NotificationFactory();

INotification notification = factory.CreateNotification("EMAIL");

notification.Notify();

}

}

**৩️⃣ Abstract Factory Pattern (অ্যাবস্ট্রাক্ট ফ্যাক্টরি)**

**🎯 উদ্দেশ্য:**

একসাথে Related Object Family তৈরি করার জন্য।

**📦 বাস্তব উদাহরণ:**

আপনার সফটওয়্যার দুইটি UI Theme সাপোর্ট করে (Light, Dark)। প্রতিটি থিমে Button ও Checkbox আলাদা। একসাথে একটা Theme লোড করার জন্য Abstract Factory ব্যবহৃত হয়।

**🧪 C# উদাহরণ:**

// Abstract Products

public interface IButton { void Render(); }

public interface ICheckbox { void Render(); }

// Concrete Products

public class LightButton : IButton { public void Render() => Console.WriteLine("Light Button"); }

public class DarkButton : IButton { public void Render() => Console.WriteLine("Dark Button"); }

public class LightCheckbox : ICheckbox { public void Render() => Console.WriteLine("Light Checkbox"); }

public class DarkCheckbox : ICheckbox { public void Render() => Console.WriteLine("Dark Checkbox"); }

// Abstract Factory

public interface IThemeFactory

{

IButton CreateButton();

ICheckbox CreateCheckbox();

}

// Concrete Factories

public class LightThemeFactory : IThemeFactory

{

public IButton CreateButton() => new LightButton();

public ICheckbox CreateCheckbox() => new LightCheckbox();

}

public class DarkThemeFactory : IThemeFactory

{

public IButton CreateButton() => new DarkButton();

public ICheckbox CreateCheckbox() => new DarkCheckbox();

}

**✅ ব্যবহার:**

class Program

{

static void Main()

{

IThemeFactory factory = new DarkThemeFactory();

IButton button = factory.CreateButton();

ICheckbox checkbox = factory.CreateCheckbox();

button.Render();

checkbox.Render();

}

}

**৪️⃣ Builder Pattern (বিল্ডার প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

কোনো জটিল অবজেক্ট ধাপে ধাপে তৈরি করা।

**📦 বাস্তব উদাহরণ:**

ধরুন আপনি একটি Pizza বানাতে চান — আলাদা করে Bread, Sauce, Topping ইত্যাদি নির্ধারণ করবেন। এভাবে আলাদা স্টেপে তৈরি করার জন্য Builder ব্যবহৃত হয়।

**🧪 C# কোড:**

// Product

public class Pizza

{

public string Bread { get; set; }

public string Sauce { get; set; }

public string Topping { get; set; }

public void Show()

{

Console.WriteLine($"Bread: {Bread}, Sauce: {Sauce}, Topping: {Topping}");

}

}

// Builder Interface

public interface IPizzaBuilder

{

void SetBread();

void SetSauce();

void SetTopping();

Pizza GetPizza();

}

// Concrete Builder

public class VegPizzaBuilder : IPizzaBuilder

{

private Pizza \_pizza = new Pizza();

public void SetBread() => \_pizza.Bread = "Thin Crust";

public void SetSauce() => \_pizza.Sauce = "Tomato";

public void SetTopping() => \_pizza.Topping = "Vegetables";

public Pizza GetPizza() => \_pizza;

}

// Director

public class PizzaDirector

{

public void MakePizza(IPizzaBuilder builder)

{

builder.SetBread();

builder.SetSauce();

builder.SetTopping();

}

}

**✅ ব্যবহার:**

class Program

{

static void Main()

{

IPizzaBuilder builder = new VegPizzaBuilder();

PizzaDirector director = new PizzaDirector();

director.MakePizza(builder);

Pizza pizza = builder.GetPizza();

pizza.Show();

}

}

**৫️⃣ Prototype Pattern (প্রোটোটাইপ প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

Object তৈরি করার জন্য তার **Copy (Clone)** ব্যবহার করা।

📌 যখন নতুন করে অবজেক্ট বানানো ব্যয়বহুল হয়, তখন বিদ্যমান Object-এর Clone কপি করে নতুন অবজেক্ট তৈরি করা হয়।

**🧪 C# উদাহরণ:**

public abstract class Shape

{

public string Color { get; set; }

public abstract Shape Clone();

}

public class Circle : Shape

{

public int Radius { get; set; }

public override Shape Clone()

{

return new Circle { Radius = this.Radius, Color = this.Color };

}

}

**✅ ব্যবহার:**

class Program

{

static void Main()

{

Circle original = new Circle { Radius = 5, Color = "Red" };

Circle clone = (Circle)original.Clone();

Console.WriteLine(original.Color); // Red

Console.WriteLine(clone.Color); // Red

}

}

**🔚 উপসংহার: কখন কোন Creational Pattern ব্যবহার করবেন?**

| **Pattern** | **উদ্দেশ্য** | **কবে ব্যবহার করবেন** |
| --- | --- | --- |
| Singleton | একটি অবজেক্টই যথেষ্ট | Configuration, Logging |
| Factory Method | সাবক্লাস নির্ধারণ করবে কী তৈরি হবে | Notification, Shape |
| Abstract Factory | একাধিক ক্লাস একসাথে তৈরি হবে | Theme, GUI Toolkit |
| Builder | ধাপে ধাপে জটিল অবজেক্ট তৈরি | Pizza, Report Generator |
| Prototype | অবজেক্ট কপি করে নতুন তৈরি | Expensive Object Creation |

চমৎকার! এবার আমরা আলোচনা করবো **Structural Design Patterns** নিয়ে — বাংলায় বিস্তারিত ব্যাখ্যা সহ, C# কোড উদাহরণ দিয়ে।

**🔷 Structural Design Patterns (গঠনমূলক ডিজাইন প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

অবজেক্ট ও ক্লাসগুলোর মধ্যে সম্পর্ক তৈরি করে বড় আকারের **স্ট্রাকচার (structure)** গঠন করা, যাতে কোডের **readability**, **flexibility** এবং **maintenance** বাড়ে।

**✅ Structural Patterns তালিকা:**

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

**১️⃣ Adapter Pattern (অ্যাডাপ্টার প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

দুইটি অসামঞ্জস্যপূর্ণ (incompatible) ইন্টারফেসকে একসাথে কাজ করাতে।

**📦 বাস্তব উদাহরণ:**

আপনার নতুন সফটওয়্যার একটি নতুন ইন্টারফেস ব্যবহার করে, কিন্তু আপনাকে পুরানো লাইব্রেরির ক্লাসও ব্যবহার করতে হবে। অ্যাডাপ্টার প্যাটার্ন দিয়ে পুরানো ক্লাসটিকে নতুন ইন্টারফেসে রূপান্তর করা যায়।

**🧪 C# উদাহরণ:**

// পুরাতন ক্লাস

public class OldPrinter

{

public void PrintOldWay()

{

Console.WriteLine("Printing using old printer...");

}

}

// নতুন ইন্টারফেস

public interface IPrinter

{

void Print();

}

// Adapter

public class PrinterAdapter : IPrinter

{

private OldPrinter \_oldPrinter = new OldPrinter();

public void Print()

{

\_oldPrinter.PrintOldWay(); // পুরাতন মেথড ব্যবহার করে নতুন ইন্টারফেস সাপোর্ট করা

}

}

**✅ ব্যবহার:**

class Program

{

static void Main()

{

IPrinter printer = new PrinterAdapter();

printer.Print();

}

}

**২️⃣ Bridge Pattern (ব্রিজ প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

Abstraction এবং Implementation-কে আলাদা করে রাখে যাতে তারা আলাদাভাবে পরিবর্তনযোগ্য হয়।

**📦 উদাহরণ:**

ধরুন আপনি Remote Control ক্লাস বানাচ্ছেন, যা TV, Radio বা অন্য ডিভাইস কন্ট্রোল করতে পারে। Remote (abstraction) ও Device (implementation) আলাদা রাখা দরকার।

**🧪 C# উদাহরণ:**

// Implementation

public interface IDevice

{

void TurnOn();

void TurnOff();

}

// Concrete Implementations

public class TV : IDevice

{

public void TurnOn() => Console.WriteLine("TV On");

public void TurnOff() => Console.WriteLine("TV Off");

}

public class Radio : IDevice

{

public void TurnOn() => Console.WriteLine("Radio On");

public void TurnOff() => Console.WriteLine("Radio Off");

}

// Abstraction

public class RemoteControl

{

protected IDevice \_device;

public RemoteControl(IDevice device)

{

\_device = device;

}

public void TogglePower()

{

\_device.TurnOn(); // উদাহরণ হিসেবে শুধু TurnOn

}

}

**৩️⃣ Composite Pattern (কম্পোজিট প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

একক (Individual) ও যৌগিক (Composite) অবজেক্টকে একইভাবে ট্রিট করতে দেওয়া।

**📦 উদাহরণ:**

ফাইল সিস্টেম — File এবং Folder উভয়ই IFileComponent হতে পারে।

**🧪 C# উদাহরণ:**

public interface IComponent

{

void Display();

}

public class File : IComponent

{

private string \_name;

public File(string name) => \_name = name;

public void Display() => Console.WriteLine("File: " + \_name);

}

public class Folder : IComponent

{

private string \_name;

private List<IComponent> \_children = new();

public Folder(string name) => \_name = name;

public void Add(IComponent component) => \_children.Add(component);

public void Display()

{

Console.WriteLine("Folder: " + \_name);

foreach (var child in \_children)

child.Display();

}

}

**৪️⃣ Decorator Pattern (ডেকোরেটর প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

বিদ্যমান অবজেক্টে ডাইনামিকভাবে নতুন ফিচার যোগ করা।

**📦 উদাহরণ:**

Text Box এর উপর Scrollbar, Border ইত্যাদি যুক্ত করা।

**🧪 C# উদাহরণ:**

// Component

public interface IText

{

string GetText();

}

// Concrete Component

public class PlainText : IText

{

public string GetText() => "Hello";

}

// Decorator Base

public class TextDecorator : IText

{

protected IText \_text;

public TextDecorator(IText text) => \_text = text;

public virtual string GetText() => \_text.GetText();

}

// Concrete Decorators

public class BoldDecorator : TextDecorator

{

public BoldDecorator(IText text) : base(text) { }

public override string GetText() => "<b>" + base.GetText() + "</b>";

}

**✅ ব্যবহার:**

class Program

{

static void Main()

{

IText text = new PlainText();

text = new BoldDecorator(text);

Console.WriteLine(text.GetText()); // <b>Hello</b>

}

}

**৫️⃣ Facade Pattern (ফ্যাসেড প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

কোনো জটিল সিস্টেমকে সহজ ইন্টারফেসের মাধ্যমে ব্যবহার করা।

**📦 উদাহরণ:**

আপনি যদি একটি ভিডিও প্লেয়ার বানান, সেখানে Codecs, FileLoader, Decoder ইত্যাদি অনেক ক্লাস থাকে। Facade ক্লাস দিয়ে ইউজারকে শুধু Play("file.mp4") দিলে সব কাজ হয়ে যাবে।

**🧪 C# উদাহরণ:**

public class VideoLoader { public void Load(string file) => Console.WriteLine("Loading " + file); }

public class Decoder { public void Decode() => Console.WriteLine("Decoding video"); }

public class Player { public void Play() => Console.WriteLine("Playing video"); }

public class VideoFacade

{

private VideoLoader \_loader = new();

private Decoder \_decoder = new();

private Player \_player = new();

public void PlayVideo(string file)

{

\_loader.Load(file);

\_decoder.Decode();

\_player.Play();

}

}

**৬️⃣ Flyweight Pattern (ফ্লাইওয়েট প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

অসংখ্য ছোট ছোট অবজেক্ট তৈরি করার সময় **মেমোরি খরচ কমানো**।

📦 উদাহরণ: হাজারো গাছ একসাথে রেন্ডার করতে হলে একই টাইপের গাছের জন্য Shape, Texture শেয়ার করা হয়।

**🧪 C# (সংক্ষিপ্ত) উদাহরণ:**

public class TreeType

{

public string Name;

public string Color;

public void Draw(int x, int y)

{

Console.WriteLine($"Drawing {Name} at ({x},{y})");

}

}

TreeFactory থেকে একই TreeType instance বারবার ব্যবহার করা হয়।

**৭️⃣ Proxy Pattern (প্রক্সি প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

আসল অবজেক্টের **কন্ট্রোল, নিরাপত্তা বা লগিং** এর জন্য তার একটি রিপ্লেসমেন্ট/প্রক্সি তৈরি করা।

📦 উদাহরণ: Virtual Proxy (lazy loading), Logging Proxy, Authentication Proxy

**🧪 C# উদাহরণ:**

public interface IService

{

void Request();

}

public class RealService : IService

{

public void Request() => Console.WriteLine("Real service called");

}

public class ProxyService : IService

{

private RealService \_realService = new();

public void Request()

{

Console.WriteLine("Logging request...");

\_realService.Request();

}

}

**🔚 উপসংহার:**

| **প্যাটার্ন** | **উদ্দেশ্য** | **উদাহরণ** |
| --- | --- | --- |
| Adapter | দুই incompatible ক্লাসকে একসাথে কাজ করানো | পুরাতন ক্লাসকে নতুন ইন্টারফেসে রূপান্তর |
| Bridge | Abstraction ও Implementation আলাদা | রিমোট ও ডিভাইস |
| Composite | Individual ও Group কে একরকম ট্রিট করা | ফাইল-ফোল্ডার |
| Decorator | নতুন ফিচার যোগ করা | Scrollbar, Border |
| Facade | জটিল সিস্টেমের সরল ইন্টারফেস | Video player |
| Flyweight | মেমোরি অপ্টিমাইজ করা | গাছ, ক্যারেক্টার |
| Proxy | অবজেক্টে অ্যাক্সেস নিয়ন্ত্রণ করা | Logging, Cache |

**অসাধারণ! এবার আমরা আলোচনা করবো Behavioral Design Patterns নিয়ে — বাংলায় বিস্তারিতভাবে এবং C# কোড উদাহরণসহ।**

**🔷 Behavioral Design Patterns (চরিত্রগত ডিজাইন প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

**অবজেক্টগুলোর মধ্যে যোগাযোগের ধরন ও কন্ট্রোল পরিচালনা করা, যাতে তারা মিলেমিশে কাজ করতে পারে এবং ব্যবহারযোগ্য আচরণ (behavior) আলাদা করা যায়।**

**✅ Behavioral Patterns তালিকা:**

1. **Strategy Pattern**
2. **Observer Pattern**
3. **Command Pattern**
4. **State Pattern**
5. **Template Method Pattern**
6. **Iterator Pattern**
7. **Mediator Pattern**
8. **Chain of Responsibility Pattern**
9. **Memento Pattern**
10. **Visitor Pattern**
11. **Interpreter Pattern**

**১️⃣ Strategy Pattern (স্ট্র্যাটেজি প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

**একটি কাজ করার একাধিক উপায় থাকতে পারে — এই প্যাটার্ন ব্যবহার করে আপনি runtime-এ কোন উপায়টি ব্যবহার করবেন তা নির্ধারণ করতে পারেন।**

**📦 উদাহরণ:**

**একটি সফটওয়্যার আছে যেখানে ডেটা কম্প্রেস করা যায় Zip বা Rar দিয়ে — ইউজার যেটা পছন্দ করবে সেটা runtime-এ প্রয়োগ হবে।**

**🧪 C# উদাহরণ:**

**// Strategy Interface**

**public interface ICompressionStrategy**

**{**

**void Compress(string fileName);**

**}**

**// Concrete Strategies**

**public class ZipCompression : ICompressionStrategy**

**{**

**public void Compress(string fileName) =>**

**Console.WriteLine($"{fileName} -> ZIP এ কম্প্রেস করা হলো");**

**}**

**public class RarCompression : ICompressionStrategy**

**{**

**public void Compress(string fileName) =>**

**Console.WriteLine($"{fileName} -> RAR এ কম্প্রেস করা হলো");**

**}**

**// Context**

**public class Compressor**

**{**

**private ICompressionStrategy \_strategy;**

**public Compressor(ICompressionStrategy strategy)**

**{**

**\_strategy = strategy;**

**}**

**public void CompressFile(string fileName)**

**{**

**\_strategy.Compress(fileName);**

**}**

**}**

**✅ ব্যবহার:**

**var compressor = new Compressor(new ZipCompression());**

**compressor.CompressFile("document.txt"); // document.txt -> ZIP এ কম্প্রেস করা হলো**

**২️⃣ Observer Pattern (অবজারভার প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

**একটি অবজেক্টে পরিবর্তন হলে সেই পরিবর্তনের পরিবর্তনের সাথে সম্পর্কিত সব অবজেক্টকে জানানো।**

**📦 উদাহরণ:**

**YouTube Channel — সাবস্ক্রাইবারদেরকে নতুন ভিডিও আপলোডের সাথে সাথে জানানো হয়।**

**🧪 C# উদাহরণ:**

**// Observer**

**public interface ISubscriber**

**{**

**void Update(string message);**

**}**

**// Concrete Observer**

**public class User : ISubscriber**

**{**

**private string \_name;**

**public User(string name) => \_name = name;**

**public void Update(string message)**

**{**

**Console.WriteLine($"{\_name}, নতুন ভিডিও: {message}");**

**}**

**}**

**// Subject**

**public class Channel**

**{**

**private List<ISubscriber> \_subscribers = new();**

**public void Subscribe(ISubscriber sub) => \_subscribers.Add(sub);**

**public void Notify(string message)**

**{**

**foreach (var sub in \_subscribers)**

**sub.Update(message);**

**}**

**}**

**✅ ব্যবহার:**

**var channel = new Channel();**

**channel.Subscribe(new User("রাহাত"));**

**channel.Subscribe(new User("নুসরাত"));**

**channel.Notify("C# Design Patterns Explained!");**

**৩️⃣ Command Pattern (কম্যান্ড প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

**অপারেশনকে একটি অবজেক্টের মধ্যে প্যাক করা, যাতে undo, redo, logging, queue ইত্যাদি করা যায়।**

**📦 উদাহরণ:**

**UI Button → CopyCommand, PasteCommand আলাদা ক্লাসে রাখা যায়।**

**🧪 C# উদাহরণ:**

**// Command Interface**

**public interface ICommand**

**{**

**void Execute();**

**}**

**// Receiver**

**public class Light**

**{**

**public void TurnOn() => Console.WriteLine("Light ON");**

**public void TurnOff() => Console.WriteLine("Light OFF");**

**}**

**// Concrete Commands**

**public class LightOnCommand : ICommand**

**{**

**private Light \_light;**

**public LightOnCommand(Light light) => \_light = light;**

**public void Execute() => \_light.TurnOn();**

**}**

**// Invoker**

**public class Remote**

**{**

**private ICommand \_command;**

**public Remote(ICommand command) => \_command = command;**

**public void PressButton() => \_command.Execute();**

**}**

**✅ ব্যবহার:**

**var light = new Light();**

**var command = new LightOnCommand(light);**

**var remote = new Remote(command);**

**remote.PressButton(); // Light ON**

**৪️⃣ State Pattern (স্টেট প্যাটার্ন)**

**🎯 উদ্দেশ্য:**

**অবজেক্টের state পরিবর্তনের সাথে সাথে তার আচরণ (behavior) পরিবর্তন করে।**

**📦 উদাহরণ:**

**TV Remote — ON / OFF অবস্থায় Remote ভিন্নভাবে কাজ করে।**

**🧪 C# উদাহরণ:**

**// State Interface**

**public interface IState**

**{**

**void Handle();**

**}**

**// Concrete States**

**public class OnState : IState**

**{**

**public void Handle() => Console.WriteLine("TV is ON");**

**}**

**public class OffState : IState**

**{**

**public void Handle() => Console.WriteLine("TV is OFF");**

**}**

**// Context**

**public class TV**

**{**

**private IState \_state;**

**public void SetState(IState state) => \_state = state;**

**public void PressButton() => \_state.Handle();**

**}**

**৫️⃣ Template Method Pattern**

**🎯 উদ্দেশ্য:**

**একটি অ্যালগরিদমের স্টেপগুলো নির্দিষ্ট করে রাখা হয়, কিন্তু কিছু স্টেপ subclass-এ override করা যায়।**

**📦 উদাহরণ:**

**Document generator — common structure থাকবে, তবে Content পার্ট subclass নির্ধারণ করবে।**

**🧪 উদাহরণ:**

**public abstract class DocumentGenerator**

**{**

**public void Generate()**

**{**

**Header();**

**Content();**

**Footer();**

**}**

**protected abstract void Content();**

**private void Header() => Console.WriteLine("=== Header ===");**

**private void Footer() => Console.WriteLine("=== Footer ===");**

**}**

**public class InvoiceGenerator : DocumentGenerator**

**{**

**protected override void Content()**

**{**

**Console.WriteLine("Invoice Content");**

**}**

**}**

**৬️⃣ Iterator Pattern**

**🎯 উদ্দেশ্য:**

**Collection এর উপাদানগুলোকে ধাপে ধাপে access করা, কিন্তু ভেতরের representation লুকিয়ে রাখা।**

**📦 উদাহরণ: List, Tree, Array traversal।**

**৭️⃣ Mediator Pattern**

**🎯 উদ্দেশ্য:**

**অবজেক্টগুলো সরাসরি একে অপরের সাথে কথা না বলে একটি Mediator Object এর মাধ্যমে কমিউনিকেট করে।**

**📦 উদাহরণ: Chat Room — ব্যবহারকারীরা সরাসরি একে অপরের সাথে না, বরং Mediator (chat room) এর মাধ্যমে কথা বলে।**

**৮️⃣ Chain of Responsibility**

**🎯 উদ্দেশ্য:**

**একটি অনুরোধ অনেকগুলো হ্যান্ডলার এর মধ্যে চেইন করে পাঠানো — যার প্রযোজ্য, সে প্রসেস করবে।**

**📦 উদাহরণ: Support Ticket — Junior → Senior → Manager**

**৯️⃣ Memento Pattern**

**🎯 উদ্দেশ্য:**

**একটি অবজেক্টের পুরানো অবস্থা (state) সংরক্ষণ করা যাতে পরে আবার Restore করা যায়।**

**📦 উদাহরণ: Undo / Redo ফিচার**

**🔚 উপসংহার:**

| **প্যাটার্ন** | **উদ্দেশ্য** | **উদাহরণ** |
| --- | --- | --- |
| **Strategy** | **বিভিন্ন behavior আলাদা class-এ** | **Compression, Sorting** |
| **Observer** | **Event system, update notification** | **YouTube, Stock Price** |
| **Command** | **Action encapsulation** | **Remote Control, Undo** |
| **State** | **ভিন্ন state অনুযায়ী ভিন্ন আচরণ** | **ON/OFF, Traffic light** |
| **Template** | **ফিক্সড অ্যালগরিদম, কাস্টমizable step** | **Report Generation** |
| **Iterator** | **Collection traversal** | **Custom List** |
| **Mediator** | **Object communication centralize** | **Chat App** |
| **Chain of Responsibility** | **Request process করার জন্য চেইন** | **Request Filtering** |
| **Memento** | **State save/restore** | **Undo Feature** |
| **Visitor** | **নতুন behavior, ক্লাস না বদলে** | **AST Walkers** |
| **Interpreter** | **Expression evaluator** | **Calculator Grammar** |