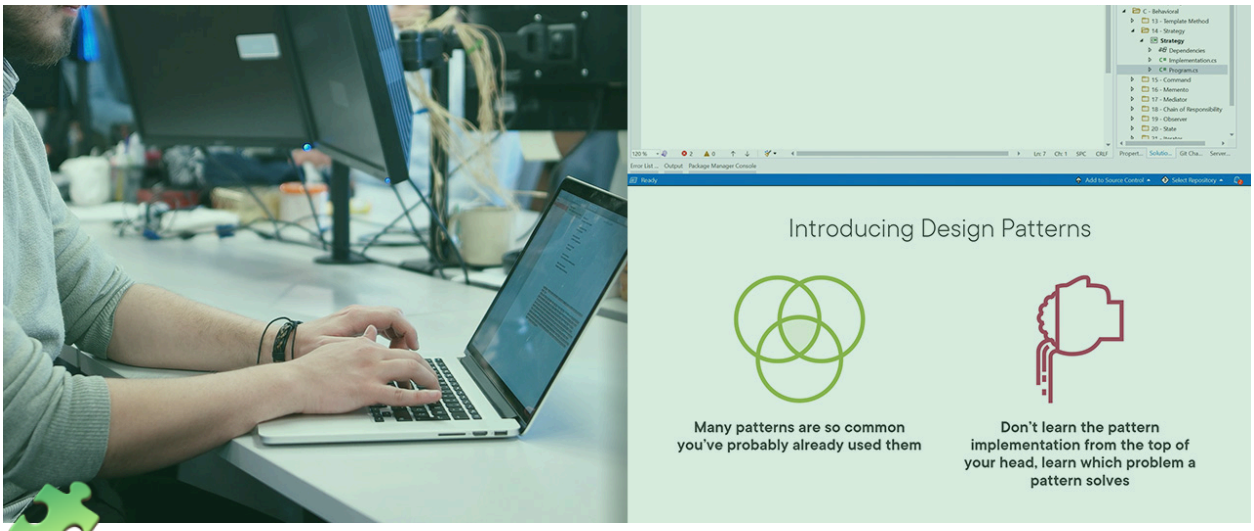


Design Patterns

👨🏻‍💻 **Hello ,I'm EN/Ahmed Hany**

In This File ,I will Explain some Design Patterns

That can Convert You to a Professional Developer

📌 My personal accounts links

🔗LinkedIn	https://www.linkedin.com/in/ahmed-hany-899a9a321?utm_source=share&utm_campaign=share_via&utm_content=profile&utm_medium=android_app
🟢WhatsApp	https://wa.me/qr/7KNUQ7ZI3KO2N1
🟦Facebook	https://www.facebook.com/share/1NFM1PfSjc/

Let's Start.

📌

What is a Design Patterns ?

A **Design Pattern** 🧩 is a reusable, proven solution to a common problem 🛠️ that occurs within a given context in software design. It provides a structured approach or best practice 📚 for solving recurring design issues efficiently 💡. Design patterns are not language-specific but can be applied to improve code maintainability 🔧, scalability 📈, and reusability ♻️. They help developers create more organized and adaptable software systems, making the code easier to understand 🤓, modify 🔄, and extend 🚀.

⚙️

What is a Types of Design Patterns ?

1. **Creational Patterns** 🏗️
2. **Structural Patterns** 🧱
3. **Behavioral Patterns** 🧠

💡

What is a Types of Creational🏗️ Patterns ?

1. **Singleton** 🔐
2. **Factory Method** 🏭
3. **Abstract Factory** 🏗️
4. **Builder** 🏗️
5. **Prototype** 👯

1️⃣ What is a Singleton🔐 Design Pattern?

✍🏻 تخيل أنك تعمل في مطار دولي ✈️، ويوجد برج مراقبة جوية 🏢 مسؤول عن إدارة حركة الطائرات. هذا البرج يجب أن يكون هناك منه نسخة واحدة فقط في المطار حتى لا يحدث تضارب في الأوامر بين الطائرات.

🔷 Singletonلماذا يعتبر برج المراقبة

1. لا يمكن أن يكون هناك أكثر من برج مراقبة واحد للمطار وإلا ستتلقى الطائرات أوامر متضاربة ❌.
2. جميع الطائرات تحتاج إلى الوصول إلى نفس البرج للحصول على التعليمات 🛬.
3. يتم توفير نقطة وصول واحدة فقط للطيارين للتواصل مع البرج 🎤.

طب هنطبق الكلام دا ازاي في الكود؟

1. هنخلي ال default constructor private وبكدا مش هينفع نعمل منو instance direct
2. نعمل private static refence of class
3. نعمل read-only lock object عشان نعمل sencrounization
4. نعمل static property ونعمل للassign علي حسب قمتوinstance

The Singleton 🔐 design pattern

ensures that a class has only **one instance** 🧩 throughout the entire application and provides a **global point of access** 🌍 to that instance. It is used to control access to shared resources like configuration settings ⚙️, database connections 🗃️, or logging mechanisms 📜.

Use Cases 🛠️:

- When you need **only one instance** of a class, such as for a **configuration manager** or **logging system**.
- When you want to **control access** to shared resources like a **database connection** or **network connection**.
- To avoid **duplicated objects** 🗑️, ensuring consistency across the application.
- In scenarios where **global state** is required, like managing **application settings** 📝 or **user sessions** 👤.

Advantages ✅:

- Saves memory by ensuring only one instance.

- Centralizes control of shared resources, making management easier and more efficient.

Example Code :

```
namespace DesignPatterns
{


    class Program
    {
        static void Main(string[] args)
        {

            AirTrafficControlTower tower1 = AirTrafficControlTower.CreateTower;
            tower1.Id = 1;
            AirTrafficControlTower tower2 = AirTrafficControlTower.CreateTower;

            Console.WriteLine(tower2.Id);//1

            Console.WriteLine($"is tower1 == tower2 ? {tower1==tower2}");//true

            AirTrafficControlTower.RebuildTower();

            AirTrafficControlTower tower3 = AirTrafficControlTower.CreateTower;

            Console.WriteLine(tower3.Id);//0

            Console.WriteLine($"is tower1 == tower3 ? {tower1 == tower2}");//false here new instance of
in tower3


        }
    }
    class AirTrafficControlTower
    {
        public int Id { get; set; }
        private static AirTrafficControlTower? _instance = null;
        private static readonly object myloker = new object();
        private AirTrafficControlTower() { } //here can not create an instance

        public static AirTrafficControlTower CreateTower //only can create instanse by this
        {
            get
            {
                lock (myloker)
                {
                    if (_instance == null)
                        _instance = new AirTrafficControlTower();

                    return _instance;
                }
            }
        }

        public static void RebuildTower()
        {
            lock (myloker)
            {
             _instance = null;

            }

        }


        public void GiveLandingPermission(string flightNumber)
        {
            Console.WriteLine($"✈️  Flight {flightNumber}  can landing..");
        }


    }
}
```

2️⃣

Factory Method 🏭 Design Pattern

✍🏻 **📌 تخيل إنك صاحب مطعم بيتزا وعاوز تنظم الشغل**

إنت عندك 3 أنواع بيتزا:

1. مارجريتا 🍕 .
2. بيبروني 🍕🔥.
3. خضار 🥦 .

بدل ما تخلي كل مرة تطلب بيتزا لازم تكريت كائن جديد من الكلاس بنفسك، إحنا هنستخدم Factory Method

💡 علشان يخلي الموضوع مرتب أكتر وسهل التطوير

وبالتالي لما تعوز بيتزا بنوع معين هتروح تنادي علي المصنع وتبعتلو النوع الي عوزو وهو يتولي المهمه ويرجعلك بطلبك والباترن دا بنطبقو بشكل كبير لو بنعملAPI

Factory Method 🏭 Design Pattern

🔷 **Definition:**

Factory Method is a **creational design pattern** that provides an **interface** for creating objects but allows subclasses to **alter the type of objects** that will be created. Instead of calling a constructor directly, the object creation logic is delegated to a method.

🛠️ Why Use Factory Method?

✅ **Encapsulation** – The creation logic is hidden from the client.

✅ **Flexibility** – Allows subclasses to decide which class to instantiate.

✅ **Scalability** – Adding new object types requires minimal changes to existing code.

📌 When to Use Factory Method?

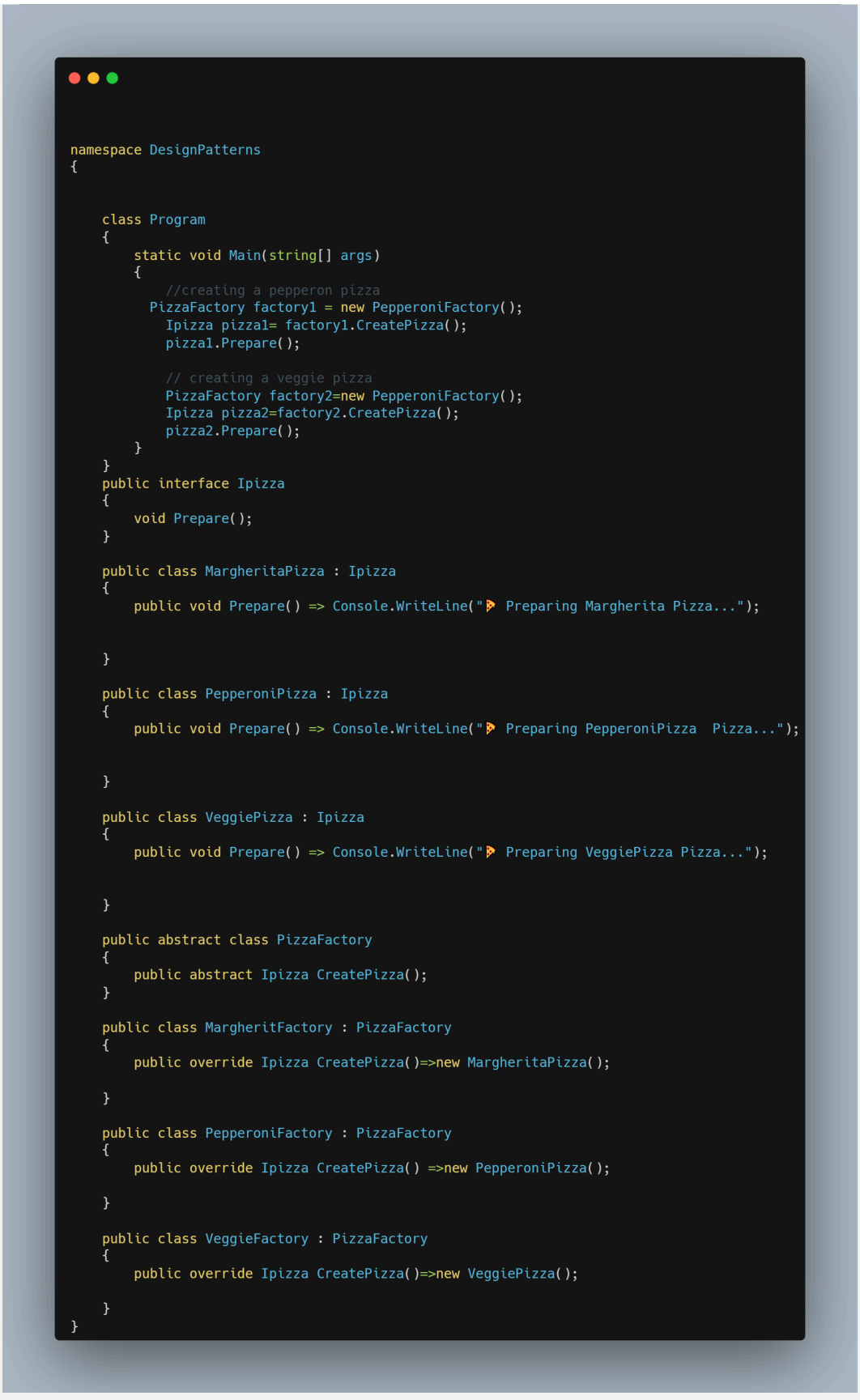
✔️ When you **don't want clients** to depend on specific class implementations.

✔️ When a class **can't anticipate** the type of objects it needs to create.

✔️ When you **want to enforce object creation rules** in a centralized place.

✅ **The Factory Method pattern simplifies object creation, making code cleaner, more maintainable, and extensible!** 🚀

Example :



```
namespace DesignPatterns
{


    class Program
    {
        static void Main(string[] args)
        {
            //creating a pepperon pizza
          PizzaFactory factory1 = new PepperoniFactory();
            Ipizza pizza1= factory1.CreatePizza();
            pizza1.Prepare();

            // creating a veggie pizza
            PizzaFactory factory2=new PepperoniFactory();
            Ipizza pizza2=factory2.CreatePizza();
            pizza2.Prepare();
        }
    }
    public interface Ipizza
    {
        void Prepare();
    }

    public class MargheritaPizza : Ipizza
    {
        public void Prepare() => Console.WriteLine("🍕 Preparing Margherita Pizza...");


    }

    public class PepperoniPizza : Ipizza
    {
        public void Prepare() => Console.WriteLine("🍕 Preparing PepperoniPizza  Pizza...");


    }

    public class VeggiePizza : Ipizza
    {
        public void Prepare() => Console.WriteLine("🍕 Preparing VeggiePizza Pizza...");


    }

    public abstract class PizzaFactory
    {
        public abstract Ipizza CreatePizza();
    }

    public class MargheritFactory : PizzaFactory
    {
        public override Ipizza CreatePizza()=>new MargheritaPizza();

    }

    public class PepperoniFactory : PizzaFactory
    {
        public override Ipizza CreatePizza() =>new PepperoniPizza();

    }

    public class VeggieFactory : PizzaFactory
    {
        public override Ipizza CreatePizza()=>new VeggiePizza();

    }
}
```

3️⃣

Abstract Factory 🏗️

✍🏻 هنا هنطور الموضوع شوية بدل ميبقا عندنا مصنع واحد زي ما شفنا فيFactory Method

هنا هيبقا عندنا مصنع رئيسي يشرف علي مجموعة مصانع فرعية وتعالي ناخد المثال

تخيل معايا إنك فاتح مطعم بيتزا كبير، وعندك نوعين مشهورين جدًا:

1. بيتزا إيطالي 🇮🇹 → بيستخدموا عجينة تخينة وصوص أبيض.
2. بيتزا أمريكي 🇺🇸 → بيستخدموا عجينة رفيعة وصوص طماطم.

كل نوع بيتزا له عجينة وصوص مختلفين، ولو عاوز تضيف نوع جديد (زي البيتزا الشرقية 🥙) فهتحتاج تغيير في كودك كل مرة! 🤯

👨‍🍳 المشكلة اللي بنواجهها:

- لو جيت تكتب كود عادي، هتلاقي نفسك كل مرة بتحدد نوع العجينة والصوص يدويًا.
- لو حبيت تضيف بيتزا جديدة هتضطر تعدل كودك الأساسي، وده خطر وممكن يبوظ حاجات تانية.
- نستخدم 🤔 الحل؟ Abstract Factory علشان نخلي الإنشاء تلقائي ومنظم!

🔧 الحل باستخدام Abstract Factory

بدل ما نحدد العجينة والصوص جوه كود البيتزا، هنقسم المكونات لمصانع مستقلة:

1. **كل نوع بيتزا له مصنع خاص بيه:**
   - **مصنع البيتزا الإيطالي مسؤول عن إنتاج عجينة تخينة وصوص أبيض.**
   - **مصنع البيتزا الأمريكي مسؤول عن إنتاج عجينة رفيعة وصوص طماطم.**
2. **كل مصنع مسؤول عن تجهيز العجينة والصوص المناسبين.**
3. **لما تيجي تطلب بيتزا، المطعم هيعرف يجيب العجينة والصوص الصح بدون ما تضيف كود زيادة.**

Abstract Factory Pattern 🏗️

📜 Definition:

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 maintaining consistency across related objects.

🎯 Use Cases:

- When an application needs to create objects **without knowing their exact class**.
- When multiple objects should be **used together** and belong to the same family.
- When the system needs to be **independent of how objects are created, composed, and represented**.

🛠️ What It Provides?

✅ **Encapsulation of object creation** 🏭

✅ **Ensures object compatibility** 🤝

✅ **Supports different object families** 🎭

✅ **Improves code scalability** 🚀

Code:

```
namespace DesignPatterns
{


    class Program
    {
        static void Main(string[] args)
        {
            //pizza from NewYork factory
            PizzaStore store1 = new PizzaStore(new NewYorkPizzaFactory());
            store1.OrderPizza();
            //Making pizza with Thik Crust and Marinara Sauce

            //pizza from Italic factory
            PizzaStore store2 = new PizzaStore(new ItalianPizzaFactory());
            store2.OrderPizza();
            //Making pizza with Thik Crust and Alfredo Sauce

        }




    }

    public interface IPizzaFactor
    {
        public IDough CreateDough();
        public ISouce CreateSouce();

    }

    public interface IDough { string GetDoughType(); }
    public interface ISouce { string GetSauceType(); }


    public class NewYorkPizzaFactory : IPizzaFactor
    {
        public IDough CreateDough()=>new ThikCrustDough();


        public ISouce CreateSouce() => new MarinaraSauce();

    }

    public class ItalianPizzaFactory : IPizzaFactor
    {
        public IDough CreateDough() => new ThikCrustDough();


        public ISouce CreateSouce()=>new AlfredoSauce();

    }

    public class ThinCrustDough : IDough {public string GetDoughType() => "Thin Crust"; }
    public class ThikCrustDough : IDough { public string GetDoughType() => "Thik Crust"; }
    public class MarinaraSauce : ISouce { public string GetSauceType() => "Marinara Sauce "; }
    public class AlfredoSauce : ISouce { public string GetSauceType() => "Alfredo Sauce "; }

    class PizzaStore
    {
        private readonly IPizzaFactor _factor;

        public PizzaStore(IPizzaFactor factor)
        {
            _factor = factor;
        }

        public void OrderPizza()
        {
            var dough= _factor.CreateDough();
            var sauce=_factor.CreateSouce();
            Console.WriteLine($"Making pizza with {dough.GetDoughType()} and {sauce.GetSauceType()}");
        }
    }

}
```

4️⃣

Builder 🏗️

✍🏻 هنا بقي الموضوع اتطور اكتر في الي فات كنت بتبقا عارف اي الي الزبون هيطلبو لكن لو كان الموضوع اكثر مرونة وفي عدد احتمالات كتير فمش ممنطقي نعملlogicالكل احتمال

📜 تخيل السيناريو ده:

دخلت مطعم برجر، والكاشير سألك:

"تحب البرجر بتاعك عامل إزاي؟"

وهنا عندك اختيارات كتير جدًا:

- نوع العيش؟ (سيمسم، بريوش، عادي) 🥯
- نوع اللحمة؟ (لحم بقرى، فراخ، نباتي) 🍖
- تحب تضيف جبنة؟ 🧀
- الخضار؟ (خس، طماطم، بصل) 🥬🍅🧅
- الصوص؟ (مايونيز، كاتشب، مستردة) 🥫

بدل ما المطعم يعمل برجر ثابت لكل الناس، بيخليك تبني البرجر بتاعك خطوة خطوة حسب رغبتك، وده بالظبط فكرة Builder Pattern! ✅

Builder Pattern 🏗️

📜 Definition:

The **Builder Pattern** is a **creational design pattern** that is used to **construct complex objects step by step**. Instead of using a long constructor with multiple parameters, the builder pattern allows for the creation of an object **piece by piece in a controlled manner**.

🎯 Use Cases:

- When an object has **many optional parameters**.
- When creating an object **requires multiple steps**.
- When you want to **improve code readability and maintainability**.

🛠️ What It Provides?

✅ **Step-by-step object creation** 🛠️

✅ **Avoids complex constructors** 🔪

✅ **Better code readability** 👀

✅ **Ensures immutability** 🔒

Code:

```
namespace DesignPatterns
{


    class Program
    {
        static void Main(string[] args)
        {
         var BurgerShop=new BugerBuilder();
        Burgur mybyrger=     BurgerShop.SetBun("Brioche")
                        .SetPatty("Chicken")
                        .SetSauce("mayo")
                        .AddCheese()
                        .AddLettuce()
                        .Build();
            mybyrger.ShowBurger();
            //burger with Brioche bun , Chicken patty cheese Lettuce   sauce : mayo

            //make Direct Burger
            var shop = new BuilderDirect();
            Burgur burger = shop.MakeCheeseBurger(new BugerBuilder());
            burger.ShowBurger();
            //burger with Sesame bun , Beef patty cheese    sauce : Ketchup

        }




    }

    public class Burgur
    {
        public string Bun { get; set; }
        public string Patty { get; set; }
        public bool Cheese { get; set; }
        public bool Lettuce { get; set; }
        public bool Tomato { get; set; }
        public bool Onion { get; set; }
        public string Sauce { get; set; }

        public void ShowBurger()
        {
            Console.WriteLine($"burger with {Bun} bun , {Patty} patty " +
                $"{(Cheese?"cheese":"")} " +
                $"{(Lettuce?"Lettuce":"")} " +
                $"{(Tomato?"Tomato":"")} " +
                $"{(Onion?"onion":"")} " +
                $"sauce : {Sauce}");
        }
    }

    public interface IBuilderBurger
    {
        IBuilderBurger SetBun(string bun);
        IBuilderBurger SetPatty(string patty);
        IBuilderBurger SetSauce(string sauce);
        IBuilderBurger AddOnion();
        IBuilderBurger AddTomato();
        IBuilderBurger AddLettuce();
        IBuilderBurger AddCheese();

        Burgur Build();

    }

    public class BugerBuilder : IBuilderBurger
    {
        private   Burgur _burger=new Burgur();
        public IBuilderBurger AddCheese() //method chaining
        {
            _burger.Cheese = true;
            return this;
        }

        public IBuilderBurger AddLettuce()
        {
            _burger.Lettuce = true;
            return this;
        }

        public IBuilderBurger AddOnion()
        {
            _burger.Onion = true;
            return this;
        }

        public IBuilderBurger AddTomato()
        {
            _burger.Tomato = true;
            return this;
        }

        public Burgur Build()
        {
            return _burger;
        }

        public IBuilderBurger SetBun(string bun)
        {
            _burger.Bun = bun;
            return this;
        }

        public IBuilderBurger SetPatty(string patty)
        {
            _burger.Patty = patty;
            return this;
        }

        public IBuilderBurger SetSauce(string sauce)
        {
            _burger.Sauce = sauce;
            return this;
        }
    }

    public class BuilderDirect
    {
        public Burgur MakeCheeseBurger(IBuilderBurger builder)
        {
            return builder.SetBun("Sesame")
                   .SetPatty("Beef")
                   .AddCheese()
                   .SetSauce("Ketchup")
                   .Build();
        }
    }



}
```

5️⃣ Prototype pattern 👯

✍🏻 وفكرتو ببساطة ان تقدر تنسخ البيانات من اوبجكت وتخزنها في اخر لتضيف عليها اضافات

نفترض إنك شغال في مطعم بيتزا. في المطعم ده، فيه بيتزا ثابتة زي بيتزا المارجريتا (🍕)، وبعدين لو جالك عميل وعايز يضيف لها مكونات تانية زي الفطر أو الزيتون، بدلاً من إنك تعيد تصنيع البيتزا من الأول، ممكن تنسخ بيتزا المارجريتا وتضيف لها المكونات الجديدة.

مقارنة بسيطة:

- الطريقة التقليدية: كل مرة العميل يطلب بيتزا، تعملها من الأول. حتى لو العميل بيطلب نفس المكونات في كل مرة، بتضطر تعيد العمل من الصفر.
- الطريقة باستخدام Prototype:

العميل يطلب بيتزا، وانت تنسخ بيتزا مارجريتا موجودة وتضيف لها المكونات اللي هو عايزها.

👯 Prototype Pattern

📜 Definition:

The **Prototype Pattern** is a **creational design pattern** that allows objects to be copied (cloned) without depending on their concrete classes. Instead of creating new objects from scratch, we duplicate existing ones, improving performance and flexibility.

🎯 Use Cases:

✅ When object creation is **expensive** (e.g., deep object initialization).

✅ When we need to **clone objects dynamically** without knowing their exact types.

✅ When an object's structure is **complex** and **repetitive creation is inefficient**.

✅ When we need a **registry of predefined objects** to be cloned.

🛠️ What It Provides?

🔄 **Efficient object creation** (avoids repetitive instantiation).

🧩 **Maintains object structure** while allowing modifications.

🎭 **Encapsulates object creation logic** outside the client code.

🚀 **Improves performance** when creating multiple similar objects.

Code:

```
namespace DesignPatterns
{


    class Program
    {
        static void Main(string[] args)
        {
            Pizza mybasepizza = new Pizza()
            {
                Dough = "Thin Crust",
                Sauce = "Tomato",
                Cheese = "Mozzarella",
                Toppings = new List<string> { "Basil" }
            };

            Console.WriteLine(mybasepizza);
            //?? Pizza with Thin Crust dough, Tomato sauce, Mozzarella cheese, Toppings: Basil

            var pepperoniPizza =(Pizza) mybasepizza.Clone();
            pepperoniPizza.Toppings.Add("Pepperoni");
            Console.WriteLine(pepperoniPizza);
               //?? Pizza with Thin Crust dough, Tomato sauce, Mozzarella cheese, Toppings: Basil,
Pepperoni

            Pizza veggiePizza = (Pizza)mybasepizza.Clone();
            veggiePizza.Toppings.AddRange(new List<string> { "Mushrooms", "Olives", "Peppers" });
            Console.WriteLine(veggiePizza);
           //?? Pizza with Thin Crust dough, Tomato sauce, Mozzarella cheese, Toppings: Basil, Mushrooms,
Olives, Peppers

        }




    }


    public class Pizza:IPizzaProtoType
    {
        public string Dough { get; set; }
        public string Sauce { get; set; }
        public string Cheese { get; set; }
        public List<string> Toppings { get; set; } = new List<string>();

        public IPizzaProtoType Clone()
        {
            return new Pizza()
            {
                Dough = this.Dough,
                Sauce = this.Sauce,
                Cheese = this.Cheese,
                Toppings =new List<string>( this.Toppings)

            };
        }

        public override string ToString()
        {
            return $"🍕 Pizza with {Dough} dough, {Sauce} sauce, {Cheese} cheese, " +
                         $"Toppings: {string.Join(", ", Toppings)}";

        }
    }

    public interface IPizzaProtoType
    {
        IPizzaProtoType Clone();
    }

}
```

💡

What is a Types of Structural Patterns 🧱 ?

1. **Adapter Pattern** 🔌
2. **Bridge Pattern** 🌉
3. **Composite Pattern** 🌿
4. **Decorator Pattern** 🎨
5. **Façade Pattern** 🏠
6. **Flyweight Pattern** 🦋
7. **Proxy Pattern** 🛡️

1️⃣ Adapter Pattern 🔌

✍🏻 فكرتو ان لو عندك كود قديم وعملت كود جديد وعاوزهم يشتغلو مع بعض وتعمل دمج ما بنهم بتعمل بنهم وسيط وهو ال Adapter

افترض عندك جهازين بيشحنوا عن طريقUSBولكن لهم أنواع مختلفة:

1. شاحن لاب توب بيستخدم مدخلType-C
2. شاحن موبايل قديم بيستخدم مدخلMicro-USB

دلوقتي عندك جهاز جديد بيقبل مدخلType-C

بس، لكن عندك شاحن قديم بيستخدم.Micro-USB

علشان تقدر تشحن الجهاز القديم، هتستخدم محولAdapter

- ده بيحول مدخل Micro-USBلمدخل Type-C
- علشان تقدر تشحن جهازك من خلال الشاحن القديم.

🔌 Adapter Pattern

📜 Definition:

The **Adapter Pattern** is a **structural design pattern** that allows incompatible interfaces to work together. It acts like a bridge, **converting** the interface of a class into another interface that a client expects. This allows **existing code to work** with other code or classes without changing their code.

🎯 Use Cases:

✅ When you need to **integrate old legacy code** with a new system.

✅ When you have incompatible interfaces that need to **work together**.

✅ When you want to **reuse existing code** that doesn't fit the current interface structure.

🛠️ What It Provides?

✅ **Interface Compatibility** - Converts one interface to another.

✅ **Flexibility** - Allows the integration of different classes that may not be compatible.

✅ **Decouples Systems** - Avoids direct modifications to existing systems.

Code:

```
namespace DesignPatterns
{


    class Program
    {
        static void Main(string[] args)
        {
             MicroCharge oldcharger=new MicroCharge();

            ITypeCCharger typeCCharger=new ChargeAdapter(oldcharger);
            typeCCharger.ChargeWithTypeC();
            //Using Adapter:
            //charging with mico - USB


        }

    }

    public class MicroCharge//old
    {
        public void ChargeWithMicroUSB()
        {
            Console.WriteLine("charging with mico-USB");
        }
    }

    public interface ITypeCCharger//new
    {
        void ChargeWithTypeC();
    }

    public class ChargeAdapter:ITypeCCharger
    {
        private readonly MicroCharge microCharge;
        public ChargeAdapter(MicroCharge microCharge)
        {
            this.microCharge = microCharge;
        }


        public void ChargeWithTypeC()//here we use old with new
        {
            Console.WriteLine("Using Adapter: ");
            microCharge.ChargeWithMicroUSB();
        }
    }
}
```

2️⃣

Bridge Pattern 🌉

✍🏻 وانت شغال مش احسن حاجة ييقي كل حاجة في كلاس واحد دا هيبقي عائق ليك لما تيجي تعدل او تضيف حاجة جديدة وعشان كدا لما تلاقي الشغل بتاعك قبل للتقسيم وبين كل جزء والاخر حاجة مشتركة ليه متقسمش عشان يسبقي عندك مرونة في التعديلات والاضافة

تخيل إن عندك شركات توصيل أكل زي طلبات وهنجرستيشن، وكل شركة بتتعامل مع طرق دفع مختلفة زي:

- كاش 💵
- فيزا 💳
- فودافون كاش 📱

بدل ما تعمل كلاس لكل شركة مع كل طريقة دفع (وده هيعمل عدد كبير جدًا من الكلاسات)، بنستخدم Bridge Pattern:

- عندنا كلاس رئيسي لكل شركة توصيل (Abstraction).
- عندنا كلاس لكل طريقة دفع (Implementation).
- كل شركة بتستخدم أي طريقة دفع بشكل مستقل، وممكن نضيف شركات جديدة أو طرق دفع جديدة بسهولة من غير ما نغيّر في الكود القديم.

يعني الكود بتاعك هيكون أكثر مرونة، وبدل ما كل حاجة تبقى مربوطة ببعضها، تقدر تغير أي جزء بسهولة! 💡

Bridge Pattern 🌉

📜 Definition:

The **Bridge Pattern** is a **structural design pattern** that separates an object's **abstraction** from its **implementation**, allowing both to evolve independently. It helps reduce code complexity by **decoupling** different layers of a system.

🎯 Use Cases:

✔️ When you want to **avoid a large inheritance hierarchy** (e.g., multiple subclasses with similar functionality).

✔️ When you need to **extend a class in multiple ways independently**.

✔️ When you want to **switch implementations dynamically** at runtime.

🛠️ What It Provides?

✅ **Decouples** abstraction from implementation ⚡

✅ **Reduces inheritance complexity** 📉

✅ **Enhances flexibility & scalability** 🚀

✅ **Supports multiple independent extensions** 🔄

Code:

```
namespace DesignPatterns
{


    class Program
    {
        static void Main(string[] args)
        {
            CashPayment cash=new CashPayment();
            CreditCardPayment creditCard=new CreditCardPayment();
            VodafoneCashPayment VodafoneCash=new VodafoneCashPayment();

            FoodDelivery order1 = new Talabat(creditCard);
            order1.OrderFood(250m);

            //Order placed through Talabat.
            //Payment of 250 EGP done using CreditCard

            FoodDelivery order2=new HungerStation(VodafoneCash);
            order2.OrderFood(400m);
            //Order placed through HungerStation.
            //Payment of 400 EGP done using VodafoneCash



            FoodDelivery order3= new Talabat(cash);
            order3.OrderFood(500m);
            //Order placed through Talabat.
            //Payment of 500 EGP done using Cash


        }

    }

    interface IPaymentMethod
    {
        void PaymentProcess(decimal amount);
    }

    class CashPayment : IPaymentMethod
    {
        public void PaymentProcess(decimal amount)
        {
            Console.WriteLine($"Payment of {amount} EGP done using Cash");
        }
    }

    class CreditCardPayment : IPaymentMethod
    {
        public void PaymentProcess(decimal amount)
        {
            Console.WriteLine($"Payment of {amount} EGP done using CreditCard");

        }
    }

    class VodafoneCashPayment : IPaymentMethod
    {
        public void PaymentProcess(decimal amount)
        {
            Console.WriteLine($"Payment of {amount} EGP done using VodafoneCash");
        }
    }

    abstract class FoodDelivery
    {
        protected IPaymentMethod paymentMethod;

        protected FoodDelivery(IPaymentMethod paymentMethod)
        {
            this.paymentMethod = paymentMethod;
        }

        public abstract void OrderFood(decimal amount);
    }

     class Talabat : FoodDelivery
     {
        public Talabat(IPaymentMethod payment):base(payment) { }

        public override void OrderFood(decimal amount)
        {
            Console.WriteLine("Order placed through Talabat.");
            paymentMethod.PaymentProcess(amount);
        }
     }

     class HungerStation : FoodDelivery
     {

        public HungerStation(IPaymentMethod payment) : base(payment) { }

        public override void OrderFood(decimal amount)
        {
            Console.WriteLine("Order placed through HungerStation.");
            paymentMethod.PaymentProcess(amount);

        }
    }


}
```

3️⃣

Composite Pattern 🌿

✍🏻 ودا الي مبني علية انظمة الملفات المختلفة زيFirebase , sqllite

بحيث انك تقدر تعمل حاجة زي متقسمة زي الشجرة

🎯 مثال في الكمبيوتر:

لما تفتح "This PC"

في Windows،

هتلاقي كل حاجة متقسمة بنفس الطريقة:

📂 C:/ Drive فيه مجلدات زي Program Files, Users, Windows، وجواهم ملفات ومجلدات تانية، وده نفس فكرة

Composite Pattern.

🌿 Composite Pattern

Definition:

is a **structural design pattern** that allows you to **treat individual objects and groups of objects uniformly**. It is commonly used to represent **hierarchical structures** like trees 🌳.

📌 Simple Explanation:

Imagine you have a **tree-like structure** consisting of **simple objects (Leaf)** and **composite objects that contain other objects (Composite)**.

📌 With **Composite Pattern**, you can treat both simple and composite objects **in the same way** without writing separate code for each!

📌 When to Use Composite Pattern?

✅ When dealing with a **hierarchical structure** like **menus, file systems, or nested categories**.

✅ When you need to **treat individual objects and groups of objects uniformly**.

✅ When you want to **apply operations on all elements, regardless of whether they are single or composite**.

✅ When you need **flexibility in adding or removing objects within a hierarchy without affecting existing code**.

Code:

```
namespace DesignPatterns
{


    class Program
    {
        static void Main(string[] args)
        {

            FileItem file1 = new FileItem("myfile1.txt");
            FileItem file2 = new FileItem("myfile2.pdf");
            FileItem file3 = new FileItem("myfile3.dox");

            Folder folder1 = new Folder("MyFolder");
            folder1.AddFile(file1);
            folder1.AddFile(file2);
            folder1.AddFile(file3);
            folder1.AddFile(file2);//not added

            folder1.Display();
            // Foler:MyFolder
               //-File: myfile1.txt
               //- File: myfile2.pdf
               //- File: myfile3.dox
        }

    }

  public  interface IFileItemSystem
  {
        void Display(string indent = "");
  }

    public  class FileItem : IFileItemSystem
    {
        private string _name;
        public FileItem(string name )
        {
            this._name = name;
        }

        public void Display(string indent = "")
        {
            Console.WriteLine($"{indent}-  File: {_name}");
        }
    }

    public class Folder:IFileItemSystem
    {
        private string _nmae;

        private HashSet<IFileItemSystem> uniquefiles= new HashSet<IFileItemSystem>(); //استخدمتها هنا عشان استفاد
من عدم التكرار
        private List<IFileItemSystem> files= new List<IFileItemSystem>();

        public Folder(string name)
        {
            this._nmae = name;
        }

        public void AddFile( IFileItemSystem file)
        {
            if(uniquefiles.Add(file))
            {
                files.Add(file);
            }
        }

        public void Display(string indent = "")
        {
            Console.WriteLine($"{indent} Foler:{_nmae}");
            foreach (var item in files)
            {
                item.Display(indent + "    ");
            }
        }
    }



}
```

4️⃣

Decorator Pattern 🎨

✍🏻 فكرتو ببساطة زي مثلا عندك شقة مفروشة وجيت اشتريت وحدة تنيه بنفس الديكور والعفش وجيت يابرنس خطبت وخطبتك قالتلكNo

استيل الاوضة دي مش عاجبني لون الحيطان ومش عارف اي عاوزة اغيرو

تيجي حضرتك مغلوب علي امرك مانت دفعت بقي😂

واشتريت شقة وخلاص قربت تتجوز

تقوم مغيرلها في ديكور اوضة نقف هنا بقي هل انت غيرت كل حاجة ولي جزء؟

طبعا جزء وكمان ممكن تجيب اثاث جديد وتضيفو علي الموجود

هنا بردو نفس الفكرة عندك الكود وحبيت تعدل عليه وتحسنو بدون تغيرو مباشرة

ك عندك محل قهوة ☕، والمحل بيبيع القهوة الأساسية، لكن العميل يقدر يضيف عليها إضافات زي اللبن، الكراميل، والشوكولاتة. بدل ما تعمل كلاس منفصل لكل نوع قهوة ممكن تتخيله، هتستخدم Decorator Pattern علشان تضيف الإضافات دون تغيير الكود الأصلي للقهوة!

🚀 باختصار: عندك قهوة عادية، وبعد كده بتضيف عليها الإضافات اللي يختارها العميل بشكل ديناميكي.

🎨 Decorator Pattern

📜 Definition:

The **Decorator Pattern** is a **structural design pattern** that allows adding new functionalities to an object **dynamically** without modifying its original structure or code. It uses **composition** instead of inheritance, making it more flexible.

🎯 Use Cases:

✅ When you need to extend the behavior of a class **without modifying** its source code.

✅ When multiple features can be **combined flexibly** at runtime.

✅ When subclassing (inheritance) leads to too many unnecessary **subclasses**.

🛠️ What It Provides?

✅ **Flexibility** 🤹 – You can add new behaviors at runtime.

✅ **Open/Closed Principle** 🔄 – Modify behavior without modifying existing code.

✅ **Better than inheritance** 📌 – No need for multiple subclasses.

Code:

```
namespace DesignPatterns
{


    class Program
    {
        static void Main(string[] args)
        {
            Icoffee coffee = new BasicCoffee();
            Console.WriteLine($"{coffee.GetDescription()} - {coffee.GetPrice()} EGP");

            coffee=new MilkDecorator(coffee);
            Console.WriteLine($"{coffee.GetDescription()} - {coffee.GetPrice()} EGP");


            coffee = new CaramelDecorator(coffee);
            Console.WriteLine($"{coffee.GetDescription()} - {coffee.GetPrice()} EGP");
            //output
            //basic coffee - 10.0 EGP
            //basic coffee Milk - 13.0 EGP
            //basic coffee Milk Caramel - 18.0 EGP
        }

    }

    public interface Icoffee
    {
        string GetDescription();
        decimal GetPrice();
    }

    public class BasicCoffee : Icoffee
    {
        public string GetDescription() => "basic coffee";

        public decimal GetPrice() => 10.0m;

    }

    public abstract class DecorationCoffee : Icoffee
    {
        protected Icoffee Icoffee;

        protected DecorationCoffee(Icoffee icoffee)=> Icoffee = icoffee;

        public virtual string GetDescription()=>Icoffee.GetDescription();

        public virtual decimal GetPrice()=>Icoffee.GetPrice();

    }

    public class MilkDecorator:DecorationCoffee
    {
        public MilkDecorator(Icoffee icoffee) : base(icoffee) { }

        public override string GetDescription() => Icoffee.GetDescription()+" Milk";

        public override decimal GetPrice() => Icoffee.GetPrice()+3.0m;



    }


    public class CaramelDecorator : DecorationCoffee
    {
        public CaramelDecorator(Icoffee icoffee) : base(icoffee) { }

        public override string GetDescription() => Icoffee.GetDescription() + " Caramel";

        public override decimal GetPrice() => Icoffee.GetPrice() + 5.0m;



    }

}
```

5️⃣ Façade Pattern 🏠

✍🏻 فكرتو انك بتبسط الكود المعقد وتعمل واجهة بسيطة للمستخدمين بدل ميشغلو دمغهم بالتفاصيل الكتير

افترض إنك بتصمم نظام تشغيل أفلام سينما، والفيلم علشان يشتغل لازم تعمل:

1. تحميل الفيلم من السيرفر.
2. تفعيل الصوت والصورة.
3. إعداد الترجمة.
4. بدء التشغيل.

كل خطوة لها كود معقد، والمستخدم مش لازم يعرف كل ده!

هنبسط الدنيا بواجهة واحدة بس (Façade**).**

💡 Façade Pattern vs Adapter Pattern

الميزة	**Façade Pattern 🏠**	**Adapter Pattern 🔌**
الهدف الأساسي	تبسيط الوصول لنظام معقد	توحيد واجهة مختلفة لتعمل مع نظام آخر
طريقة العمل	يقدم **واجهة موحدة** لنظام يحتوي على أجزاء كثيرة	يغير **الشكل أو الواجهة** لجعل كود قديم يشتغل مع جديد
المثال الواقعي	مكتب استعلامات في مول يبسط الوصول لكل المحلات	محول شاحن بيخلي Type-C يشتغل مع Micro-USB

✅ لو عندكنظام معقد وعاوز تسهل التعامل معاه

استخدم→ Façade

.

✅ لو عندككودين غير متوافقين وعاوزهم يشتغلوا مع بعض

استخدم→ Adapter

.

🏠 Façade Pattern

📜 Definition:

The **Façade Pattern** is a **structural design pattern** that provides a **simplified and unified interface** to a complex system of classes, libraries, or subsystems. It hides the complexity

and exposes only the necessary functionalities to the client.

🎯 Use Cases:

✅ When a system has **multiple complex subsystems**, and you want to provide a **simplified interface** for clients.

✅ When you want to **decouple** clients from a system's complex internals.

✅ When you need to **organize and structure** a large codebase by grouping related functionalities behind a single entry point.

🛠️ What It Provides?

✅ **Simplicity** 🏡 – Reduces dependencies and hides complexities.

✅ **Better Code Organization** 📦 – Helps structure large projects.

✅ **Loose Coupling** 🔌 – Clients interact with a simplified API instead of directly dealing with multiple components.

Code:

```
namespace DesignPatterns
{
    class Program
    {
        static void Main(string[] args)
        {

            CinemaFacade cinema = new CinemaFacade();
            cinema.PlayMovie($"Preson Break Session 2 ", "English");
            //output
            //Loading : Preson Break Session 2
            //setting up Audio
            //Loading English subtitles...
            //Playing Video
            //Playing Audio
           //Enjoy Your Move.......
        }

    }

    public  class VideoPlayer
    {
        public void LoadVideo(string Movie) => Console.WriteLine($"Loading : {Movie}");
        public void PlayVieo() => Console.WriteLine("Playing Video");
    }

    public  class AudioPlayer
    {
        public void SetAudio() => Console.WriteLine($"setting up Audio");
        public void PlayAudio() => Console.WriteLine("Playing Audio");
    }

    public  class SubtitleManager
    {
        public void LoadSubtitles(string language) => Console.WriteLine($"Loading {language}
subtitles...");
    }

    public class CinemaFacade
    {
        private readonly  VideoPlayer _videoPlayer;
        private readonly AudioPlayer _audioPlayer;
        private readonly SubtitleManager _subtitleManager;

        public CinemaFacade()
        {
            _videoPlayer = new VideoPlayer();
            _audioPlayer = new AudioPlayer();
            _subtitleManager = new SubtitleManager();
        }

        public void PlayMovie(string movie,string Language)
        {
            _videoPlayer.LoadVideo(movie);
            Thread.Sleep(2000);
            _audioPlayer.SetAudio();
            Thread.Sleep(1500);
            _subtitleManager.LoadSubtitles(Language);
            Thread.Sleep(1000);
            _videoPlayer.PlayVieo();
            Thread.Sleep(1000);
            _audioPlayer.PlayAudio();
            Console.WriteLine("Enjoy Your Move .......");
        }

    }
}
```

6️⃣

Flyweight Pattern 🦋

✍🏻 هنا بقي تعالي نشوف ازاي البيانات بتتكيش ونرجع نعيد استخدمها

فكرتو نفس فكرت الكاش بس الفرق ان الكاش كهارد وير بتخزن بيانات

لكن هنا هنكيشObjests ازاي الكلام دا وهل هو ممكن ؟

تخيل إنك شغال في مطبعة جرنان، والجرنان فيه آلاف الكلمات، وكل حرف بيتكرر مئات أو آلاف المرات. لو كل حرف بيتخزن ككائن منفصل، ده هيستهلك ذاكرة رهيبة!

الحل؟ 🤔

بدل ما نخزن كل حرف ككائن مستقل، هنستخدم بحيث نخزن كل حرف مرة واحدة بس ونرجعله كل ما نحتاجه، وبكده نوفر في استهلاك الذاكرة

👨‍🏭 الجورنالجي بيكتب المقالات، لكن بدل ما يجيب خط جديد لكل حرف بيستخدم الحروف اللي عنده في المطبعة ويرتبها صح عشان يطلع المقال. الحروف المشتركة بتكون موجودة عنده وبيعيد استخدامها بدل ما يطبع حرف جديد كل مرة (Shared Objects).

🦋 Flyweight Pattern

📜 Definition:

The **Flyweight Pattern** is a **structural design pattern** that reduces memory usage by **sharing common data** between multiple objects instead of storing duplicate data. It is useful when an application creates a large number of similar objects.

🎯 Use Cases:

✅ When an application needs to create **a large number of similar objects**.

✅ When storing duplicate data **causes high memory consumption**.

✅ When objects have **shared (intrinsic) and unique (extrinsic) properties**.

🛠️ What It Provides?

✅ **Memory optimization** 💾 – Reduces memory footprint by sharing common object parts.

✅ **Better performance** 🚀 – Fewer objects mean faster execution.

✅ **Efficient resource usage** 🎯 – Avoids unnecessary object duplication.

Code:

```
namespace DesignPatterns
{
    class Program
    {
        static void Main(string[] args)
        {
            MyCache myCache = new MyCache();

            string word = "Hello Iam BackEnd";

            foreach (char letter in word)
            {
                Iletter letterobj = myCache.GetIletter(letter);
                letterobj.Display("Arial");
            }

        }

    }

    public interface Iletter
    {
        void Display(string font);
        void SetCacheStatus(string v);
    }

    public class Letter : Iletter
    {
        private readonly char _symbol;
        private  string cache_value;
        public Letter(char symbol,string cachval)
        {
            this._symbol = symbol;
            this.cache_value = cachval;
        }

        public void Display(string font)
        {
            Console.WriteLine($"{cache_value} =>  Displaying Letter '{_symbol}' with Font: {font}");
        }

        public void SetCacheStatus(string status)  // 🛠️ Update cache status to "Hit"
        {
            this.cache_value = status;
        }
    }

    public class MyCache
    {
        private readonly Dictionary<char,Iletter> _Letters = new Dictionary<char,Iletter>();

        public Iletter GetIletter(char letter)
        {
            if (!_Letters.ContainsKey(letter))
            {

                _Letters[letter] = new Letter(letter,"Miss");
            }
            else
            {
                _Letters[letter].SetCacheStatus("Hit");
            }

            return _Letters[letter];

        }
    }
}
```

Output

```
Miss ⇒  Displaying Letter 'H' with Font: Arial
Miss ⇒  Displaying Letter 'e' with Font: Arial
Miss ⇒  Displaying Letter 'l' with Font: Arial
Hit ⇒  Displaying Letter 'l' with Font: Arial
Miss ⇒  Displaying Letter 'o' with Font: Arial
Miss ⇒  Displaying Letter ' ' with Font: Arial
Miss ⇒  Displaying Letter 'l' with Font: Arial
Miss ⇒  Displaying Letter 'a' with Font: Arial
Miss ⇒  Displaying Letter 'm' with Font: Arial
Hit ⇒  Displaying Letter ' ' with Font: Arial
Miss ⇒  Displaying Letter 'B' with Font: Arial
Hit ⇒  Displaying Letter 'a' with Font: Arial
Miss ⇒  Displaying Letter 'c' with Font: Arial
Miss ⇒  Displaying Letter 'k' with Font: Arial
Miss ⇒  Displaying Letter 'E' with Font: Arial
Miss ⇒  Displaying Letter 'n' with Font: Arial
Miss ⇒  Displaying Letter 'd' with Font: Arial
```

7️⃣

Proxy Pattern 🛡️

✍🏻 من اسمو انت عاوز تطبق حماية ومش اي حد يتعامل مع الاوبجكت بتاعك لازم يكون في وسيط يتأكدلي من هويتك الاول وانك ليك صلاحية

تخيل إنك رايح البنك علشان تسحب فلوس، بس مش أي حد ينفع يسحب على طول! لازم الأول تعدي على الموظف اللي بيتأكد إن معاك بطاقة الرقم القومي أو كارت البنك والباسورد قبل ما يسمحلك تسحب الفلوس.

في المثال ده:

- الحساب البنكي (Bank Account) = الموضوع الحقيقي (Real Subject)
- الموظف اللي بيطلب البطاقة أو الباسورد (Proxy) = الوكيل اللي بيحمي الوصول للحساب
- العميل اللي عاوز يسحب الفلوس = الكود اللي بيستخدم الحساب البنكي

🛡️ Proxy Pattern

📜 Definition:

The **Proxy Pattern** is a **structural design pattern** that provides a **substitute or placeholder** for another object to **control access** to it. This allows for additional functionalities such as **lazy initialization, access control, logging, caching, or security** without modifying the original object's code.

🎯 When to Use It?

✅ **Security & Access Control** 🛑 – Restrict access to sensitive operations (e.g., authentication).

✅ **Lazy Initialization** ⏳ – Load heavy objects only when needed (e.g., database connections, images).

✅ **Logging & Monitoring** 📊 – Log calls to the real object without modifying it.

✅ **Caching & Performance** 🚀 – Store previous results to avoid redundant computations.

Code:

```
namespace DesignPatterns
{
    class Program
    {
        static void Main(string[] args)
        {
            IBankAcount bankAcount = new BankAccountProxy(8000,"493u");
            bankAcount.Cridit(3500);
            //Enter Your Password : 493U
            //Welcome Sir
            //Withdrawal successful!New balance: 4500 EGP
        }

    }

   public interface IBankAcount
    {
        void Cridit(double amount);
    }

    public class BankAcount(double balance) : IBankAcount
    {
        private  double _balance =balance;


        public void Cridit(double amount)
        {
            if(amount<=_balance)
            {
                _balance-=amount;
                Console.WriteLine($" Withdrawal successful! New balance: {_balance} EGP");

            }
            else
            {
                 Console.WriteLine($" Insufficient funds! Current balance: {_balance} EGP");
            }
        }
    }

    public class BankAccountProxy:IBankAcount
    {
        private string? _pin;
        private BankAcount _realAcount;

        public BankAccountProxy(double intialbalance,string pin)
        {
            _realAcount=new BankAcount(intialbalance);
            _pin = pin.ToLower();
        }

        public void Cridit(double amount)
        {
            Console.Write("Enter Your Password : ");
            string? _enterypin = Console.ReadLine();

            if(_enterypin?.ToLower() ==_pin)
            {
                Console.WriteLine("Welcome Sir");
                _realAcount.Cridit(amount);
            }
            else
            {
                Console.WriteLine(" Incorrect PIN! Access Denied.");
            }
        }
    }
}
```

💡

What is a Types of Behavioral Patterns 🧠?

1. **Observer Pattern** 👀
2. **Strategy Pattern** 🎯
3. **Command Pattern** 🎮
4. **State Pattern** 🏁
5. **Chain of Responsibility Pattern** 🔗
6. **Mediator Pattern** 🗣️
7. **Iterator Pattern** 🔁
8. **Template Method Pattern** 📝
9. **Visitor Pattern** 👤
10. **Memento Pattern** ⏳

1️⃣ Observer Pattern 👀

✍🏻 هنا بقا نيجي للتركات زي لو عاوز تراقب حدث معين انو لو حصل تنفذ حاجة معينه

لو انت فاهم استخدام الeventsمش هيبقي في جديد

📌 أمثلة من الحياة الواقعية:

1️⃣ 🔔 إشعارات السوشيال ميديا

- لما تعمل Follow لشخص، أي بوست جديد بينزله بتوصلك عليه إشعارات.

2️⃣ 📡 تحديثات البورصة أو الطقس

- علشان Observer Pattern التطبيقات اللي بتتابع سعر الدولار أو حالة الطقس بتستخدم أي تغيير يحصل يتم إبلاغ المشتركين فورًا.

3️⃣ 📧 إشعارات البريد الإلكتروني

- لما تشترك في Newsletter، أي خبر جديد بينزله الموقع بتوصلك عليه رسالة تلقائيًا.

4️⃣ 📞 إشعارات تطبيقات الدردشة

- لما حد يكتبلك رسالة على واتساب أو تليجرام، الإشعار بيجيلك فورًا بدون ما التطبيق يفضل يسأل السيرفر كل ثانية.

👀 Observer Pattern

📜 Definition:

The **Observer Pattern** is a **behavioral design pattern** that allows an object (**Subject**) to notify a list of other objects (**Observers**) whenever a change occurs. It helps establish a **one-to-many** dependency without creating strong coupling.

🛠️ When to Use It?

✅ When you need to **notify multiple objects** about changes in another object.

✅ When you want to **decouple** the object that sends updates from the objects that receive them.

✅ When you want **automatic notifications** instead of manually calling methods on each observer.

هنعمل مثال بسيط عن شخص عنده قناة يوتيوب، ولما ينزل فيديو جديد، كل المشتركين في القناة بيجي لهم إشعار تلقائي.

Code:

```
namespace DesignPatterns
{
    class Program
    {
        static void Main(string[] args)
        {
            YouTubeChannel youTubeChannel = new YouTubeChannel("Dragons");

            User user1 = new User("ahmed");
            User user2 = new User("omar");
            User user3 = new User("said");

            youTubeChannel.VideoUploaded += user1.OnVideoUploaded;
            youTubeChannel.VideoUploaded += user2.OnVideoUploaded;


            youTubeChannel.UploadVideo("Observer Pattern");
            //Output
            // Dragons uploaded a new video: Observer Pattern
            //ahmed, new video 'Observer Pattern' uploaded by Dragons!
            // omar, new video 'Observer Pattern' uploaded by Dragons!

            youTubeChannel.VideoUploaded -= user1.OnVideoUploaded;
            youTubeChannel.VideoUploaded += user3.OnVideoUploaded;

            youTubeChannel.UploadVideo("Events");
            //output
            // Dragons uploaded a new video: Events
            //omar, new video 'Events' uploaded by Dragons!
            //said, new video 'Events' uploaded by Dragons!

        }

    }

    public class YouTubeChannel(string channelname)
    {
        public string ChannelName { get; } = channelname;

        public event Action<string, string> VideoUploaded;

        public void UploadVideo(string videoTitle)
        {
            Console.WriteLine($" {ChannelName} uploaded a new video: {videoTitle}");
            VideoUploaded?.Invoke(ChannelName, videoTitle);
        }

    }

    public class User(string name)
    {
        private string username=name;

        public void OnVideoUploaded(string channelname,string videoTitle)
        {
            Console.WriteLine($" {username}, new video '{videoTitle}' uploaded by {channelname}!");
        }
    }

}
```

2️⃣

Strategy Pattern 🎯

✍🏻 من اسمو بتحدد الواجهة الي عاوزها فبدل متستخدمif-else كتير خلي الموضوع اكثر مرونة ويشتغل في الRuntime

تخيل إنك بتروح مشوار كل يوم، بس كل يوم بتروح بطريقة مختلفة حسب الظروف:

- لو معاك عربية، هتسوق بسرعة بس البنزين غالي ⛽🚗.
- لو راكب ميكروباص، هتوفر فلوس بس الطريق زحمة 🚐🛑.
- لو راكب عجلة، هتوصل بلياقة بس هتتعب 🚴💨.

فبدل ما يبقى عندك كود متلغبط جوه الفانكشن بتاعتك، انت بتغير الطريقة اللي بتمشي بيها المشوار من غير ما تعدل الكود الأساسي، وده بالظبط اللي بيعمله الـ Strategy Pattern.

🚀 Strategy Pattern

📜 Definition:

The **Strategy Pattern** is a **behavioral design pattern** that allows an object to **select a behavior (algorithm) at runtime**. Instead of using multiple `if-else` or `switch` statements, different algorithms are encapsulated in separate **strategy classes**, making it easy to switch between them dynamically.

🎯 Use Cases:

✅ When an application needs to **support multiple ways** of performing an action (e.g., different payment methods, sorting algorithms).

✅ When you want to **eliminate long** `if-else` **or** `switch-case` **statements** by organizing logic into separate classes.

✅ When the behavior of an object **needs to change dynamically** at runtime.

🛠️ What It Provides?

✅ **Flexibility** 🤹 – Easily switch between different behaviors without modifying the main code.

✅ **Encapsulation** 📦 – Keeps different algorithms separate and easy to maintain.

✅ **Open/Closed Principle** 🔄 – Allows adding new strategies **without modifying existing code**.

✅ **Improves testability** 🧪 – Each strategy can be tested independently.

Code:

```
namespace DesignPatterns
{

    class Program
    {
        static void Main(string[] args)
        {
            TravelContext context = new TravelContext();


            context.SetTravelStrategy(new CarTravel());
            context.GoToWork("Univeristy");

            context.SetTravelStrategy(new BusTravel());
            context.GoToWork("Office");

            context.SetTravelStrategy(new BicycleTravel());
            context.GoToWork("Football pitch");

            // Traveling to Univeristy by Car. Fast but costly!
            //Traveling to Office by Bus.Slow but cheap!
            //Traveling to Football pitch by Bicycle. Healthy but tiring!
        }
    }

    public interface ITravelStrategy
    {
        void Travel(string destination);
    }


    public class CarTravel : ITravelStrategy
    {
        public void Travel(string destination)
        {
            Console.WriteLine($" Traveling to {destination} by Car. Fast but costly!");
        }
    }

    public class BusTravel : ITravelStrategy
    {
        public void Travel(string destination)
        {
            Console.WriteLine($" Traveling to {destination} by Bus. Slow but cheap!");
        }
    }

    public class BicycleTravel : ITravelStrategy
    {
        public void Travel(string destination)
        {
            Console.WriteLine($" Traveling to {destination} by Bicycle. Healthy but tiring!");
        }
    }


    public class TravelContext
    {
        private ITravelStrategy _strategy;


        public void SetTravelStrategy(ITravelStrategy strategy)
        {
            _strategy = strategy;
        }


        public void GoToWork(string destination)
        {
            if (_strategy == null)
            {
                Console.WriteLine("⚠ No travel strategy selected!");
                return;
            }

            _strategy.Travel(destination);
        }
    }




}
```

3️⃣

Command Pattern 🎮

✍️ هنا بقي انت بتحول الكود بتاعك زي الاوامر الي انت متحكم فيها زي دراع البلايستيشن لما تعوز امر بتبدأ تحطو الاول ثم تضغط علي الزر عشان يتنفذ وكمان ممكن ترجع فيه

تخيل عندك ريموت كنترول في البيت 🎮، وكل زرار فيه بيعمل أمر معين، زي:

- زرار ON يشغل النور 💡.
- زرار OFF يطفي النور 🌑.
- زرار UNDO يرجّع آخر حاجة عملتها.

هنا بيخلّي كل زرار عبارة عن كود منفصل، والريموت نفسه مش محتاج الـ Command Pattern يعرف التفاصيل، هو بس بيدّي الأمر، والنور هو اللي بينفّذه.

🎮 Command Pattern

📜 Definition:

The **Command Pattern** is a **behavioral design pattern** that turns a **request (command)** into an object. This allows us to **parameterize, queue, and log requests**, and even support **undo/redo** functionality.

🎯 Use Cases:

✅ When you want to **decouple** the sender (who requests an action) from the receiver (who executes it).

✅ When you need to **queue requests** or execute them **at different times**.

✅ When implementing **undo/redo functionality** (like in text editors 🎨).

✅ When dealing with **remote control systems** (like smart home automation 🏠).

🛠️ What It Provides?

✅ **Loose Coupling** 🔗 – The sender doesn't need to know who will execute the command.

✅ **Flexibility** 🔄 – You can create different commands dynamically.

✅ **History & Undo Support** ⏪ – Since commands are objects, they can be stored and reversed easily.

Code:

```
namespace DesignPatterns
{

    class Program
    {
        static void Main(string[] args)
        {
           Light Rome=new Light();

            ICommand Turnon=new LightOnCommand(Rome);

            ICommand Turnoff = new LightOffCommand(Rome);

            Remote remote = new Remote();
            remote.SetCommand(Turnon);
            remote.PressButton();//Light On

            remote.SetCommand(Turnoff);
            remote.PressButton();//Light Off

        }
    }


    public interface ICommand
    {
        void Execute();
        void undo();
    }

    class Light
    {
        public void TurnOn() => Console.WriteLine("Light On ");
        public void TurnOff() => Console.WriteLine("Light Off ");

    }

    class LightOnCommand:ICommand
    {
        private readonly Light _light;

        public LightOnCommand(Light light)
        {
            _light = light;
        }

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


        public void undo()=>_light.TurnOff();

    }

    class LightOffCommand:ICommand
    {
        private readonly Light _light;

        public LightOffCommand(Light light)
        {
            _light = light;
        }

        public void Execute() => _light.TurnOff();


        public void undo() => _light.TurnOn();
    }

    class Remote
    {
        private ICommand _command;

        public void SetCommand(ICommand command) =>_command = command;

        public void PressButton()=>_command.Execute();
        public void PressUndo()=>_command.undo();
    }
}
```

4️⃣

State Pattern 🏁

✍🏻 ودا بقي الباترن الدوامة واكتر واحد ديناميكي لانك حرفيا بتمشي الموضوع في دوامة وبل متعملif&else بتخلي الموضوع يتم داخليا وقت التشغيل

تخيل إنك بتشغل مروحة سقف، المروحة ليها 3 سرعات وحالة إيقاف. كل مرة تضغط على زرار التشغيل، المروحة تغيّر سرعتها، ولو ضغطت تاني بعد أعلى سرعة ترجع توقف تاني.

الـ State Pattern بيخلّي كل حالة ليها كود منفصل، والمروحة نفسها متعرفش التفاصيل، هي بس بتغيّر الحالة وتسيبها تنفّذ اللي مطلوب.

🔄 State Pattern

📜 Definition:

The **State Pattern** is a **behavioral design pattern** that allows an object to change its behavior dynamically based on its internal state. Instead of using complex `if-else` or `switch` statements, each state is encapsulated in a separate class.

🎯 Use Cases:

✅ When an object's **behavior changes** based on its **state** (e.g., vending machines, ATMs, traffic lights 🚦).

✅ When using **too many if-else statements** to manage state transitions.

✅ When you want to **add new states easily** without modifying existing code.

✅ When designing **finite state machines** (FSMs) for workflows, games, or UI navigation.

🛠️ What It Provides?

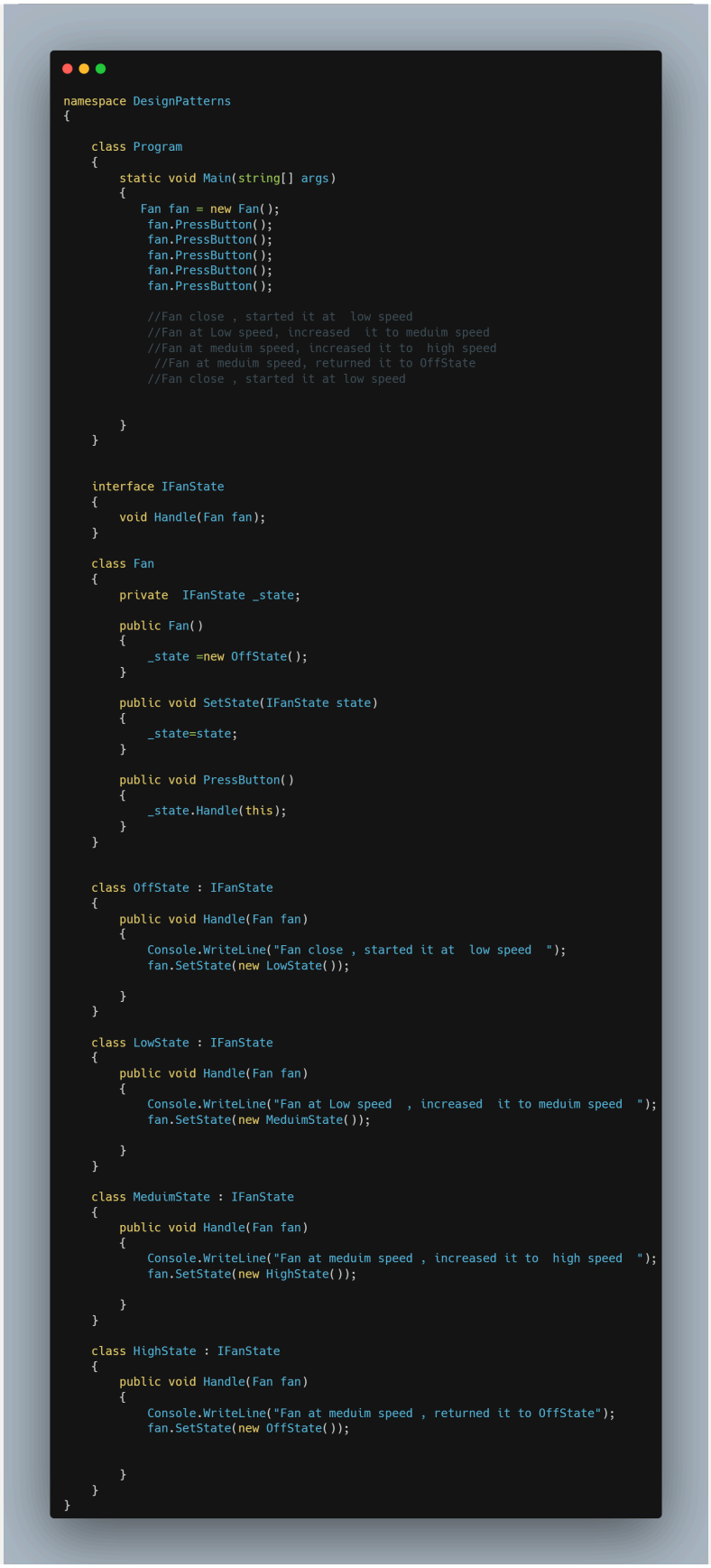
✅ **Encapsulation** 💃 – Each state is a separate class, making the code clean and modular.

✅ **Flexibility** 🔄 – You can add or modify states without affecting existing logic.

✅ **Better Maintainability** 🛠️ – Avoids messy `if-else` blocks, improving readability.

✅ **Dynamic Behavior** ⚡ – The object can change its behavior at runtime based on its state.

Code:



```
namespace DesignPatterns
{

    class Program
    {
        static void Main(string[] args)
        {
           Fan fan = new Fan();
            fan.PressButton();
            fan.PressButton();
            fan.PressButton();
            fan.PressButton();
            fan.PressButton();

            //Fan close , started it at  low speed
            //Fan at Low speed, increased  it to meduim speed
            //Fan at meduim speed, increased it to  high speed
             //Fan at meduim speed, returned it to OffState
            //Fan close , started it at low speed


        }
    }


    interface IFanState
    {
        void Handle(Fan fan);
    }

    class Fan
    {
        private  IFanState _state;

        public Fan()
        {
            _state =new OffState();
        }

        public void SetState(IFanState state)
        {
            _state=state;
        }

        public void PressButton()
        {
            _state.Handle(this);
        }
    }


    class OffState : IFanState
    {
        public void Handle(Fan fan)
        {
            Console.WriteLine("Fan close , started it at  low speed  ");
            fan.SetState(new LowState());

        }
    }

    class LowState : IFanState
    {
        public void Handle(Fan fan)
        {
            Console.WriteLine("Fan at Low speed  , increased  it to meduim speed  ");
            fan.SetState(new MeduimState());

        }
    }

    class MeduimState : IFanState
    {
        public void Handle(Fan fan)
        {
            Console.WriteLine("Fan at meduim speed , increased it to  high speed  ");
            fan.SetState(new HighState());

        }
    }

    class HighState : IFanState
    {
        public void Handle(Fan fan)
        {
            Console.WriteLine("Fan at meduim speed , returned it to OffState");
            fan.SetState(new OffState());


        }
    }
}
```

5️⃣

Chain of Responsibility Pattern 🔗

✍🏻 نفس فكرة الي فات بس هنا بنمشي تدريجيا ومش بنلف في دوامة يعني بنختبر شرط الاول لو اتحقق تمام متحققش نروح للمرحلة الي بعدو لحد منوصل لاخر مرحلة وبنقف مش بنرجع للاول زي الباترن الي قبلو

تخيل إنك بتشتغل في شركة وعندك نظام لاستقبال الشكاوي من العملاء. مش أي حد في الشركة يقدر يحل أي مشكلة، فالموضوع بيمشي بالتسلسل:

1️⃣ خدمة العملاء (Customer Support) تحاول تحل المشكلة الأول.

2️⃣ لو المشكلة كبيرة، تتحول لـ المدير (Manager).

3️⃣ لو المشكلة أخطر، تروح لـ المدير العام (Director).

💡 كل مرحلة بتقرر هل تقدر تحل المشكلة ولا تحولها للي بعدها. وده بالظبط فكرة Chain of Responsibility Pattern! 🔗

🔗 Chain of Responsibility Pattern

📜 Definition:

The **Chain of Responsibility Pattern** is a **behavioral design pattern** that allows multiple handlers to process a request **sequentially**. Each handler decides either to handle the request or **pass it to the next handler** in the chain.

🎯 Use Cases:

✅ When multiple objects can handle a request, but you **don't know which one** will handle it.

✅ When you want to **avoid tightly coupling** senders and receivers.

✅ When requests need to be **processed in a specific order** (e.g., logging, authentication, validation).

✅ When implementing **event bubbling** in UI frameworks.

🛠️ What It Provides?

✅ **Decoupling** 🔗 – The sender doesn't need to know who will handle the request.

✅ **Flexible Processing** 🔄 – New handlers can be added without modifying existing ones.

✅ **Clear Flow** 📜 – Requests are handled in a structured sequence.

✅ **Better Maintainability** 🛠️ – Each handler focuses only on its responsibility.

Code:

```
namespace DesignPatterns
{

    public interface IComplaintHandler
    {
        void SetNextHandler(IComplaintHandler nextHandler);
        void HandleComplaint(string complaintLevel);
    }


    public class CustomerSupport : IComplaintHandler
    {
        private IComplaintHandler _nextHandler;

        public void SetNextHandler(IComplaintHandler nextHandler)
        {
            _nextHandler = nextHandler;
        }

        public void HandleComplaint(string complaintLevel)
        {
            if (complaintLevel == "Low")
            {
                Console.WriteLine(" Customer Support: Issue resolved successfully!");
            }
            else
            {
                Console.WriteLine(" Customer Support: Escalating to the Manager...");
                _nextHandler?.HandleComplaint(complaintLevel);
            }
        }
    }


    public class Manager : IComplaintHandler
    {
        private IComplaintHandler _nextHandler;

        public void SetNextHandler(IComplaintHandler nextHandler)
        {
            _nextHandler = nextHandler;
        }

        public void HandleComplaint(string complaintLevel)
        {
            if (complaintLevel == "Medium")
            {
                Console.WriteLine(" Manager: I have resolved the issue!");
            }
            else
            {
                Console.WriteLine(" Manager: Escalating to the Director...");
                _nextHandler?.HandleComplaint(complaintLevel);
            }
        }
    }


    public class Director : IComplaintHandler
    {
        public void SetNextHandler(IComplaintHandler nextHandler)
        {

        }

        public void HandleComplaint(string complaintLevel)
        {
            Console.WriteLine(" Director: This is a critical issue! I will handle it personally!");
        }
    }


    class Program
    {
        static void Main(string[] args)
        {

            CustomerSupport support = new CustomerSupport();
            Manager manager = new Manager();
            Director director = new Director();

            support.SetNextHandler(manager);
            manager.SetNextHandler(director);


            Console.WriteLine("\n Customer complains about a Low-level issue:");
            support.HandleComplaint("Low");

            Console.WriteLine("\n Customer complains about a Medium-level issue:");
            support.HandleComplaint("Medium");

            Console.WriteLine("\n Customer complains about a High-level issue:");
            support.HandleComplaint("High");

            /*
             output
              Customer complains about a Low-level issue:
              Customer Support: Issue resolved successfully!

              Customer complains about a Medium-level issue:
              Customer Support: Escalating to the Manager...
              Manager: I have resolved the issue!

              Customer complains about a High-level issue:
              Customer Support: Escalating to the Manager...
              Manager: Escalating to the Director...
              Director: This is a critical issue! I will handle it personally!

             */
        }
    }
}
```

6️⃣

Mediator Pattern 🗣️

✍️ ودا فكرتو ان زاي الobjectsيقدرو يكلمو بعض بشكل غير مباشر عن طريق وسيط
هنفترض إن عندنا غرفة شات فيها أكتر من مستخدم، كل مستخدم مش بيبعت رسائل للناس التانيين بشكل مباشر، لكنه بيبعتها للوسيط (اللي هو غرفة الشات) وهي اللي بتوصل الرسالة لكل الناس.

🕹️ Mediator Pattern

📜 Definition:

The **Mediator Pattern** is a **behavioral design pattern** that helps reduce direct communication between multiple objects by introducing a **mediator**. Instead of objects **talking to each other directly**, they **communicate via a central mediator**.

🔷 This **decouples** objects and makes the system **more maintainable and scalable**.

🎯 Use Cases:

✅ When multiple objects need to communicate **without being tightly coupled**.

✅ When you want to **centralize complex interactions** among objects.

✅ When you need to reduce **dependency between components** (e.g., in UI elements, chat systems, or air traffic control 🛫).

🛠️ What It Provides?

✅ **Loose Coupling** 🔗 – Objects don't need direct references to each other.

✅ **Simplified Communication** 📢 – Centralizes interactions.

✅ **Better Maintainability** 🛠️ – Easier to extend and modify.

Code to send to all like Group WhatsApp:

```
namespace DesignPatterns
{



    class Program
    {
        static void Main(string[] args)
        {
          IChatMediator chatroom = new ChatMediator();

            User ahmed =new ChatUser(nameof(ahmed),chatroom);
            User ali = new ChatUser(nameof(ali), chatroom);
            User mena = new ChatUser(nameof(mena), chatroom);


            chatroom.AddUser(ahmed);
            chatroom.AddUser(ali);
            chatroom.AddUser(mena);

            ahmed.SendMessage("Hello Every One");
            ali.SendMessage("welcome every one Iam a Back End");
            mena.SendMessage("hello my friends, Iam a Front End");

            /*
             output
             ahmed send Hello Every One
             ali recieved message from ahmed : Message => Hello Every One
             mena recieved message from ahmed : Message => Hello Every One
             ali send welcome every one Iam a Back End
             ahmed recieved message from ali : Message => welcome every one Iam a Back End
             mena recieved message from ali : Message => welcome every one Iam a Back End
             mena send hello my friends, Iam a Front End
             ahmed recieved message from mena : Message => hello my friends, Iam a Front End
             ali recieved message from mena : Message => hello my friends, Iam a Front End
             */
        }
    }

    public interface IChatMediator
    {
        void SendMessage(string message,User sender);
        void AddUser(User user);
    }

    public class ChatMediator : IChatMediator
    {
        private List<User> _users = new List<User>();

        public void AddUser(User user)
        {
            _users.Add(user);
        }

        public void SendMessage(string message, User sender)
        {
            foreach (var user in _users)
            {
                if (user != sender)
                {
                    user.RecieveMessage(message, sender);
                }
            }
        }
    }

    public abstract class User
    {
        public string Name { get; }
        protected IChatMediator chatMediator;


        protected User(string name, IChatMediator chatMediator)
        {
            Name = name;
            this.chatMediator = chatMediator;
        }

        public abstract void RecieveMessage(string message,User sender);
        public abstract void SendMessage(string message);

    }

    public class ChatUser:User
    {
        public ChatUser(string name,IChatMediator chatMediator) : base(name, chatMediator) { }

        public override void RecieveMessage(string message, User sender)
        {
            Console.WriteLine($"{Name} recieved message from {sender.Name} : Message => {message}");
        }

        public override void SendMessage(string message)
        {
            Console.WriteLine($"{Name} send {message}");
            chatMediator.SendMessage(message,this);
        }
    }
}
```

Code To send to one Person:

```
namespace DesignPatterns
{



    class Program
    {
        static void Main(string[] args)
        {
          IChatMediator chatroom = new ChatMediator();

            User ahmed =new ChatUser(nameof(ahmed),chatroom);
            User ali = new ChatUser(nameof(ali), chatroom);
            User mena = new ChatUser(nameof(mena), chatroom);


            chatroom.AddUser(ahmed);
            chatroom.AddUser(ali);
            chatroom.AddUser(mena);

            ahmed.SendMessage("Hello mena ",nameof(mena));
            ali.SendMessage("welcome ahmed iam a Back End",nameof(ahmed));
            mena.SendMessage("hello ahmed ",nameof(ahmed));

            /*
             output
            ahmed send Hello mena
            mena recieved message from ahmed : Message => Hello mena
            ali send welcome ahmed iam a Back End
            ahmed recieved message from ali : Message => welcome ahmed iam a Back End
            mena send hello ahmed
            ahmed recieved message from mena : Message => hello ahmed
             */
        }
    }

    public interface IChatMediator
    {
        void SendMessage(string message,User sender, string receiverName);
        void AddUser(User user);
    }

    public class ChatMediator : IChatMediator
    {
        private List<User> _users = new List<User>();

        public void AddUser(User user)
        {
            _users.Add(user);
        }

        public void SendMessage(string message, User sender, string receiverName)
        {
            User? reciever=_users.Find(user=>user.Name==receiverName);

            if(reciever!=null)
            {
                reciever.RecieveMessage(message, sender);
            }
        }
    }

    public abstract class User
    {
        public string Name { get; }
        protected IChatMediator chatMediator;


        protected User(string name, IChatMediator chatMediator)
        {
            Name = name;
            this.chatMediator = chatMediator;
        }

        public abstract void RecieveMessage(string message,User sender);
        public abstract void SendMessage(string message,string recievername);

    }

    public class ChatUser:User
    {
        public ChatUser(string name,IChatMediator chatMediator) : base(name, chatMediator) { }

        public override void RecieveMessage(string message, User sender)
        {
            Console.WriteLine($"{Name} recieved message from {sender.Name} : Message => {message}");
        }

        public override void SendMessage(string message, string recievername)
        {
            Console.WriteLine($"{Name} send {message}");
            chatMediator.SendMessage(message,this,recievername);
        }
    }
}
```

7️⃣

Iterator Pattern 🔄

✍🏻 فكرتو ان ازاي تقدر تعملloopingبشكل ديناميكي

📚 مثال

تخيل إنك فاتح مكتبة كتب وفيها رف مليان كتب. بدل ما تمسك الكتب عشوائيًا، عندك أمين مكتبة بيقف معاك ويمر على الكتب واحد واحد، يقولك:

- "تحب تبص عليه؟ GoF لمؤلفه "Design Patterns" معايا كتاب"
- "عاوز تقراه؟ Robert C. Martin لمؤلفه "Clean Code" معايا كتاب"
- "تحب تاخده؟ Andrew Hunt لمؤلفه "The Pragmatic Programmer" معايا كتاب"

يعني بيخليك تعدي على الكتب من غير ما تفتح الرف، الأمين ده هو اللي بيعمل Iterator بنفسك أو تعرف الكتب متخزنة إزاي جوه المكتبة!

تقدر كمان تطلب منه يعدي بترتيب معين، زي إنه يجيبلك الكتب من الآخر للأول أو يجيبلك كتب مؤلف معين بس.

الفكرة ببساطة: بدل ما تتعامل مباشرة مع الكتب جوه المكتبة، بتستخدم حد مسؤول عن 🔄 تمرير الكتب لك بطريقة منظمة، وده اللي بيعمله Iterator Pattern! 🚀

🔄 Iterator Pattern

📜 Definition:

The **Iterator Pattern** is a **behavioral design pattern** that provides a way to **access elements of a collection** sequentially **without exposing its underlying structure**. It allows iteration over a collection in a **consistent and controlled way**.

🎯 Use Cases:

✅ When you need to **traverse a collection** without exposing its internal structure.

✅ When you want to provide **multiple ways to iterate** over a collection.

✅ When different types of collections need a **uniform way of iteration** (e.g., lists, trees, graphs).

✅ When working with **complex data structures** that require custom iteration logic.

🛠️ What It Provides?

✅ **Encapsulation** 🔒 – The collection's internal structure remains hidden.

✅ **Flexibility** 🔄 – Can provide multiple iteration strategies (e.g., reverse order).

✅ **Consistent Traversal** 🔁 – Ensures all collections can be traversed the same way.

✅ **Simplifies Code** 👨‍💻 – No need to manually handle looping logic in different places.

Code:

```
namespace DesignPatterns
{



    class Program
    {
        static void Main(string[] args)
        {
          Libarary library = new Libarary();
            library.AddBook(new Book("Design Patterns", "GoF"));
            library.AddBook(new Book("Clean Code", "Robert C. Martin"));
            library.AddBook(new Book("The Pragmatic Programmer", "Andrew Hunt"));


            Iiterator<Book> iterator = library.CreateInterator();

            while (iterator.HasNext())
            {
                Console.WriteLine(iterator.next());
            }

            /*
             output
            Design Patterns - GoF
            Clean Code - Robert C. Martin
            The Pragmatic Programmer - Andrew Hunt
             */
        }
    }
    public interface Iiterator<T>
    {
        bool HasNext();
        T next();
    }

    public interface IbookCollection
    {
        Iiterator<Book> CreateInterator();
    }

    public class Book
    {
        public string Title { get; }
        public string Author { get; }

        public Book(string title, string author)
        {
            Title = title;
            Author = author;
        }

        public override string ToString()
        {
            return $"{Title} - {Author}";
        }
    }

    public class BookIterator:Iiterator<Book>
    {
        private readonly List<Book> _books;
        private int position = 0;

        public BookIterator(List<Book> books)
        {
            this._books = books;
        }

        public bool HasNext()
        {
            return position < _books.Count;
        }

        public Book next()
        {
            return _books[position++];
        }
    }

    public class Libarary : IbookCollection
    {
        private  List<Book> _books=new List<Book>();

        public void AddBook(Book book)
        {
            _books.Add(book);
        }
        public Iiterator<Book> CreateInterator()
        {
            return new BookIterator(this._books);
        }


    }
}
```

8️⃣ Template Method Pattern 📝

✍️ وهنا انت بتقسم الكود بتاعك طبقات وفي كل طبقة في حاجة ثابتة بتحصل بس بتيجي في طبقة معينه هتنفذ حاجه مختلفة علي حسب لوجيك معين

تخيل إنك شيف محترف وعندك وصفة ثابتة لعمل البيتزا، بس كل مرة ممكن تغيّر مكونات معينة على حسب النوع اللي بتعمله. الوصفة بتاعتك ماشية كده:

1️⃣ تحضير العجينة 🥖

2️⃣ فرد العجينة في الصينية 🍕

3️⃣ إضافة الصوص والمكونات الأساسية 🍅🧀

4️⃣ إضافة المكونات الخاصة بكل نوع بيتزا (هنا كل نوع بيختلف عن التاني)

5️⃣ خبز البيتزا في الفرن 🔥

6️⃣ التقديم والأكل بالهنا والشفا! 😋

كل الخطوات ثابتة، بس الفرق بيكون في الخطوة الرابعة، يعني:

- لو هتعمل مارجريتا، هتحط جبنة موتزاريلا بس.
- لو هتعمل بيتزا سي فود، هتحط جمبري وكاليماري.
- لو هتعمل بيتزا ببروني، هتحط ببروني وشرائح زيتون.

هنا إحنا عاملين نظام ثابت للوصفة، لكن مع ترك مساحة للتغيير في جزء معين حسب النوع، وده هو Template Method Pattern! 🍕🔥

📜 Definition:

The **Template Method Pattern** is a **behavioral design pattern** that defines the **skeleton of an algorithm** in a **base class** but **lets subclasses override specific steps** without changing the algorithm's structure.

🎯 Use Cases:

✅ When multiple classes share a similar workflow but have **some variations in certain steps**.

✅ When you want to **enforce a sequence of steps** while allowing some flexibility.

✅ When avoiding **code duplication** across multiple subclasses.

✅ When defining **frameworks or libraries** where users can customize parts of the behavior.

🛠️ What It Provides?

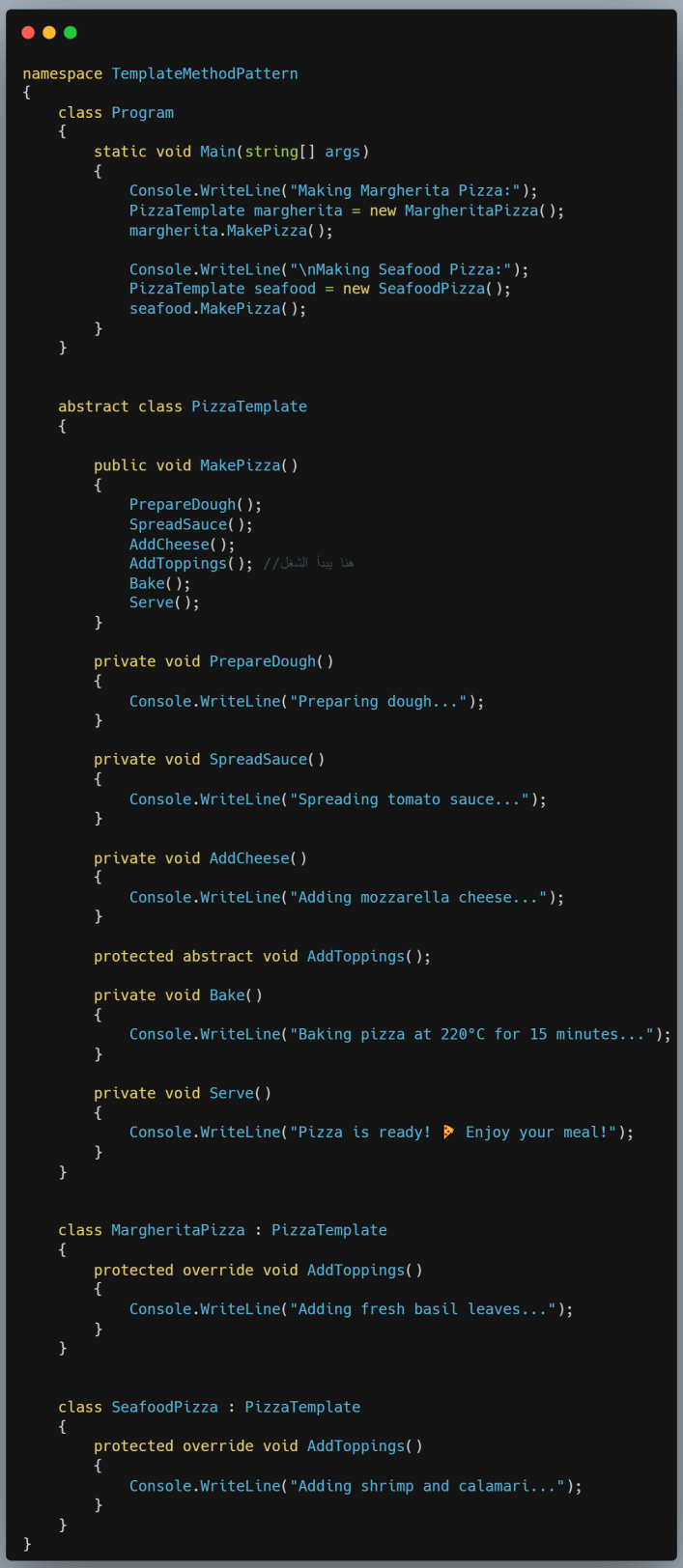
✅ **Code Reusability** 🔁 – Common logic is defined in one place (base class).

✅ **Flexibility** 🔄 – Allows subclasses to define only the varying parts.

✅ **Encapsulation** 🔒 – The overall algorithm remains protected from modifications.

✅ **Enforced Workflow** 📌 – Ensures that steps execute in a fixed sequence.

Code:



```
namespace TemplateMethodPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Making Margherita Pizza:");
            PizzaTemplate margherita = new MargheritaPizza();
            margherita.MakePizza();

            Console.WriteLine("\nMaking Seafood Pizza:");
            PizzaTemplate seafood = new SeafoodPizza();
            seafood.MakePizza();
        }
    }


    abstract class PizzaTemplate
    {

        public void MakePizza()
        {
            PrepareDough();
            SpreadSauce();
            AddCheese();
            AddToppings(); //هنا يبدأ الشغل
            Bake();
            Serve();
        }

        private void PrepareDough()
        {
            Console.WriteLine("Preparing dough...");
        }

        private void SpreadSauce()
        {
            Console.WriteLine("Spreading tomato sauce...");
        }

        private void AddCheese()
        {
            Console.WriteLine("Adding mozzarella cheese...");
        }

        protected abstract void AddToppings();

        private void Bake()
        {
            Console.WriteLine("Baking pizza at 220°C for 15 minutes...");
        }

        private void Serve()
        {
            Console.WriteLine("Pizza is ready! 🍕 Enjoy your meal!");
        }
    }


    class MargheritaPizza : PizzaTemplate
    {
        protected override void AddToppings()
        {
            Console.WriteLine("Adding fresh basil leaves...");
        }
    }


    class SeafoodPizza : PizzaTemplate
    {
        protected override void AddToppings()
        {
            Console.WriteLine("Adding shrimp and calamari...");
        }
    }
}
```

9️⃣ Visitor Pattern 👤

✍️ فكرتو انط لو عندكobjects مختلفة وعاوز تطبق عليها عمليات

من غير متعدل في الكود الاساسي او عاوز تفصل العميات عن الكائنات فبالتالي كودك هيكون سهل التعديل واضافة عمليه جديدة مش هيبقي عائق عن انا عامل فصل لدا عن دا

تخيل إنك شغال في شركة كبيرة، وعندك أنواع مختلفة من الموظفين:

- المهندسين
- المديرين
- المحاسبين

الشركة قررت إنها محتاجة تعمل تحليل للرواتب وتقييم الأداء لكل الموظفين. بدل ما نعدل علشان نفصل العمليات دي في Visitor Pattern في الكود بتاع كل نوع موظف، هنستخدم كلاس منفصل، ونقدر نضيف أي عمليات جديدة بعد كده بسهولة.

📜 Definition:

The **Visitor Pattern** is a **behavioral design pattern** that allows you to **add new behaviors to existing classes without modifying them**. It achieves this by defining a **visitor** class that contains the new operations, which can be applied to objects of different types.

🎯 Use Cases:

✅ When you need to **add multiple operations** to an existing object structure **without modifying its classes**.

✅ When working with **complex object structures**, like trees or composite objects.

✅ When different operations need to be performed on the same objects (e.g., **exporting data, validation, logging**).

✅ When following the **Open/Closed Principle**, allowing new behaviors without modifying existing code.

🛠️ What It Provides?

✅ **Extensibility** 📌 – New operations can be added without modifying existing classes.

✅ **Separation of Concerns** 🔄 – Keeps operations separate from object structure.

✅ **Scalability** 🚀 – Supports adding more visitor classes for different operations.

✅ **Flexibility** 🤹 – Different visitors can define different behaviors for the same object structure.

Code:

```
namespace TemplateMethodPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            List<IEmployee> employees = new List<IEmployee>()
            {
                new Engineer("ahmed",10000),
                new Manager("omar",20000)

            };

            IVistor salarycalculating = new SalaryCalculator();
            IVistor performanceEvaluator = new PerformanceEvaluator();

            Console.WriteLine("salarys");
            foreach (var item in employees)
            {
                item.Accept(salarycalculating);
            }
            Console.WriteLine("Performance");

            foreach (var emp in employees)
            {
                emp.Accept(performanceEvaluator);
            }

            /*
             * Output
             salarys
             engineer=> ahmed has salary = 10000
             manager=>  omar has salary = 20000
             Performance
             Performance of   engineer=> ahmed Excellent
             Performance of  manager=>  omar good

             */
        }
    }

    public interface IEmployee
    {
        void Accept(IVistor vistor);
    }

    public interface IVistor
    {
        void Visit(Engineer engineer);
        void Visit(Manager manager);


    }

    public class Engineer : IEmployee
    {
        public string Name { get; set; }
        public int Salary { get; set; }

        public Engineer(string name, int salary)
        {
            Name = name;
            Salary = salary;
        }

        public void Accept(IVistor vistor)
        {
            vistor.Visit(this);
        }
    }

    public class Manager : IEmployee
    {
        public string Name { get; set; }
        public int Salary { get; set; }

        public Manager(string name, int salary)
        {
            Name = name;
            Salary = salary;
        }

        public void Accept(IVistor vistor)
        {
            vistor.Visit(this);
        }
    }

    public class SalaryCalculator : IVistor
    {
        public void Visit(Engineer engineer)
        {
            Console.WriteLine($"engineer=> { engineer.Name} has salary = {engineer.Salary}");
        }

        public void Visit(Manager manager)
        {
            Console.WriteLine($"manager=>  {manager.Name} has salary = {manager.Salary}");

        }
    }

    public class PerformanceEvaluator : IVistor
    {
        public void Visit(Engineer engineer)
        {
            Console.WriteLine($"Performance of   engineer=> {engineer.Name} Excellent");
        }

        public void Visit(Manager manager)
        {
            Console.WriteLine($"Performance of  manager=>  {manager.Name} good");

        }
    }

}
```

🔟 Memento Pattern ⏳

✍️ فكرتو انك ازاي تعمل حفظ لحالات الobjectالمختلفة وتقدر ترجع لحالة منهم

زي فكرة ال Ctrl + Z😂 الي بتنقذنا في حالات كتيير

تخيل إنك بتلعب لعبة بلايستيشن 🎮، ووصلت للزعيم الأخير بعد ما تعبت جدًا. قبل ما تبدأ القتال، بتعمل Save علشان لو خسرت، تقدر ترجع للوضع اللي كنت فيه بدل ما تعيد المرحلة من الأول. لو كسبت، تمام، ولو خسرت، ترجع لـ Save Point التاني وتحاول.

🧠 Memento Pattern

📜 Definition:

The **Memento Pattern** is a **behavioral design pattern** that allows an object to save and restore its previous state **without violating encapsulation**. It is commonly used for **undo/redo functionality** in applications.

🎯 Use Cases:

✅ **Undo/Redo functionality** – Like in text editors, image editors, or game save states.

✅ **State recovery** – Restoring an object to a previous state after an error or crash.

✅ **Encapsulation of state** – Keeping the object's internal details hidden from the external world while still allowing restoration.

🛠️ What It Provides?

✅ **Encapsulation** 🔒 – The object's internal state remains private.

✅ **Time travel** ⏳ – Allows rolling back to a previous state easily.

✅ **History tracking** 📜 – Can store multiple states for flexible restoration.

✅ **Improved flexibility** 🔄 – Useful for applications requiring frequent state changes.

Code:

```
namespace DesignPatterns
{
    class Program
    {
        static void Main(string[] args)
        {
            TextEditor editor = new TextEditor();
            History history = new History();

            editor.Type("Hello, ");
            editor.ShowContent();
            Console.WriteLine("--------------------");
            history.Save(editor.Save());

            editor.Type("World!");
            editor.ShowContent();
            Console.WriteLine("--------------------");

            history.Save(editor.Save());

            editor.Type(" Let's learn C#!");
            editor.ShowContent();
            Console.WriteLine("--------------------");


            editor.Restore(history.Undo());
            editor.ShowContent();
            Console.WriteLine("--------------------");


            editor.Restore(history.Undo());
            editor.ShowContent();


            /*
             output
             Current Content: Hello,
             --------------------
              Current Content: Hello, World!
             --------------------
              Current Content: Hello, World! Let's learn C#!
             --------------------
              Current Content: Hello, World!
             --------------------
              Current Content: Hello,
             */
        }
    }
    class TextMemento
    {
        public string Content { get; }

        public TextMemento(string content)
        {
            Content = content;
        }
    }


    class TextEditor
    {
        private string _content = "";

        public void Type(string text)
        {
            _content += text;
        }

        public void ShowContent()
        {
            Console.WriteLine($" Current Content: {_content}");
        }

        public TextMemento Save()
        {
            return new TextMemento(_content);
        }

        public void Restore(TextMemento memento)
        {
            _content = memento.Content;
        }
    }


    class History
    {
        private Stack<TextMemento> _history = new Stack<TextMemento>();

        public void Save(TextMemento memento)
        {
            _history.Push(memento);
        }

        public TextMemento Undo()
        {
            return _history.Count > 0 ? _history.Pop() : null;
        }
    }


}
```