**Design patterns**

کمک می کند که ساختار کد بهتری داشته باشیم.

3 مبحث دارد

* Structural
* Behavioral
* Creational

**Creational**

هدف : نحوه ساختن یک instance از یک شی به چه صورت باشد و سرویس ما به چه شکلی instance را به ما بدهد.

**Structural**

آشنایی با ساختار سرویس کامپوننت و .. به شکل باشند. هدف چگونه یک ویژگی را به یک مفهوم اضافه کنیم و مفاهیم چگونه به هم ترجمه می شوند.

**Behavioral**

برای بررسی رفتار یک سرویس مثلا یا اطلاع رسانی کی رفتار کی انجام شده است. یا زمانی که بخواهیم زنجیره ای از کار ها انجام شود chain of responsibility. هدف این است که شی های یا object یا instance ها به هم دستور دهند یا اطلاع رسانی کنند

Abstract Factory

از نوع creational است.

هدف:

ارائه دهنده یک interface برای خانواده ای از object ها که به هم مرتبط یا وابسته هستند. برای ایجاد مجموعه ای از object ها که به هم مرتبط هستند و اطمینان از این که object ها با هم سازگار هستند.

بدون این که مشخص کند concrete class مشخص کند یعنی کلاس پیاده سازی آن معلوم نباشد.

**Problem**

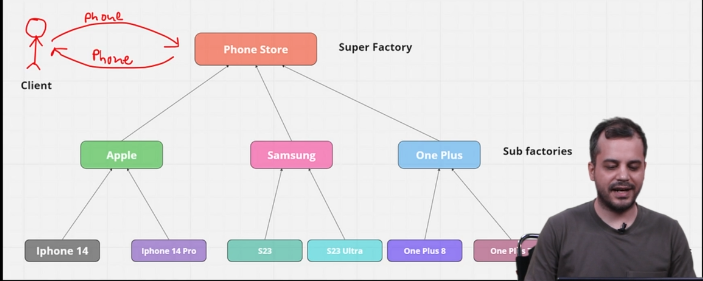
فرض کنید که یک Uiدارید که و در حال توسعه آن هستید و نیازمندی این است که برای ظاهر کار نیازمند تم های برای حالت های light dark , .. هستید و در ui خود نیز دارای ایتم های مانند button checkbox هستند و آن ها را در کنار هم ایجاد کرده اید اما نمی خواهید برای هر حالت یک instace از dark , light ایجاد کنید زیر را تعویض تم و افزودن تم را برای شما سخت می کند.

**Solution**

با ایجاد کردن interface برای object های که به هم ارتباط دارند و در یک خانواده هستند این کار را برای ما راحت می کند.

همانند کارخونه بسته به نیازمندی که دارید مبل رو بهت می دهد اما بسته به کارخونه ظاهر مبل فرق می کند. اما ماهیت همون مبله به شکل دیگری بسته به آن factory

در مثال زیر super factory ما ساخت موبایل است. تعاریف concret در لایه های پایین مثل اپل و سامسونگ که هر کدام موبایل می دهند به سبک خود مثلا S23 و iphone 14

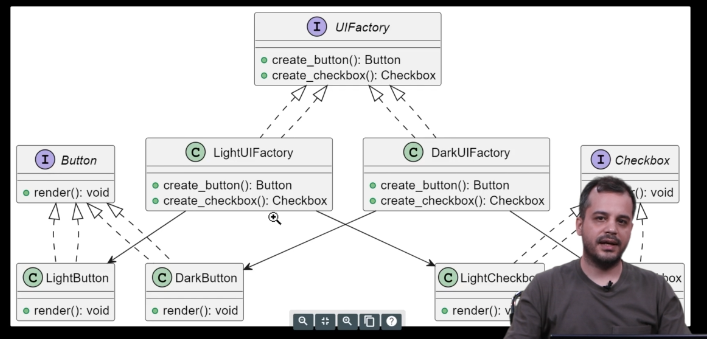


**When to use**

زمانی که سیستم می خواهد مستقل باشد از نحوه ایجاد product ها است در تعاریف بالایی.

می خواهیم یک خانواده مشترک و مرتبط از محصولات داشته باشیم در نهایت.

صرفا به interface نیاز داریم و نه implementation



**Gpt**

واسط (interface) برای ایجاد خانواده‌ای از اشیاء مرتبط فراهم می‌کند، بدون اینکه کد کلاینت (مصرف‌کننده) بداند این اشیاء از کدام کلاس خاص ایجاد می‌شوند.

* وقتی سیستم باید از خانواده‌ای از اشیاء استفاده کند و نیازی نیست بداند این اشیاء دقیقا از چه کلاس‌هایی ساخته شده‌اند.
* وقتی می‌خواهیم پلتفرم‌های مختلف (مثلاً UI ویندوز و مک) را بدون تغییر در کد کلاینت پشتیبانی کنیم.
* وقتی می‌خواهیم مطمئن شویم اشیائی که با هم استفاده می‌شوند، با هم سازگارند.

**ساختار کلی**

1. **AbstractFactory**: رابط یا کلاس پایه برای کارخانه‌ها
2. **ConcreteFactory**: پیاده‌سازی کارخانه‌ها برای ساخت نسخه خاص از اشیاء
3. **AbstractProduct**: رابط یا کلاس پایه برای اشیاء
4. **ConcreteProduct**: پیاده‌سازی اشیاء واقعی
5. **Client**: از Factory استفاده می‌کند تا اشیاء را بسازد و استفاده کند

**مزایای Abstract Factory**

* جداسازی کد ساخت از کد استفاده
* امکان توسعه آسان (افزودن یک تم جدید فقط با ساخت یک Factory جدید)
* اعمال محدودیت برای سازگاری اجزای مختلف (دکمه و چک‌باکس با هم هماهنگ می‌مانند)

**معایب**

* افزایش تعداد کلاس‌ها
* اضافه کردن نوع جدید محصول ممکن است نیاز به تغییر همه کارخانه‌ها داشته باشد
* Abstract Factory Pattern in C#

// Abstract Product A  
public interface IButton {  
 void Paint();  
}  
  
// Abstract Product B  
public interface ICheckbox {  
 void Paint();  
}  
  
// Concrete Product A1  
public class WinButton : IButton {  
 public void Paint() {  
 Console.WriteLine("Render a button in Windows style.");  
 }  
}  
  
// Concrete Product B1  
public class WinCheckbox : ICheckbox {  
 public void Paint() {  
 Console.WriteLine("Render a checkbox in Windows style.");  
 }  
}  
  
// Concrete Product A2  
public class MacButton : IButton {  
 public void Paint() {  
 Console.WriteLine("Render a button in macOS style.");  
 }  
}  
  
// Concrete Product B2  
public class MacCheckbox : ICheckbox {  
 public void Paint() {  
 Console.WriteLine("Render a checkbox in macOS style.");  
 }  
}  
  
// Abstract Factory  
public interface IGUIFactory {  
 IButton CreateButton();  
 ICheckbox CreateCheckbox();  
}  
  
// Concrete Factory 1  
public class WinFactory : IGUIFactory {  
 public IButton CreateButton() {  
 return new WinButton();  
 }  
 public ICheckbox CreateCheckbox() {  
 return new WinCheckbox();  
 }  
}  
  
// Concrete Factory 2  
public class MacFactory : IGUIFactory {  
 public IButton CreateButton() {  
 return new MacButton();  
 }  
 public ICheckbox CreateCheckbox() {  
 return new MacCheckbox();  
 }  
}  
  
// Client  
public class Application {  
 private IButton button;  
 private ICheckbox checkbox;  
  
 public Application(IGUIFactory factory) {  
 button = factory.CreateButton();  
 checkbox = factory.CreateCheckbox();  
 }  
  
 public void Paint() {  
 button.Paint();  
 checkbox.Paint();  
 }  
}  
  
// Usage  
class Program {  
 static void Main(string[] args) {  
 IGUIFactory factory;  
 string os = "Windows"; // Could be set dynamically  
  
 if (os == "Windows")  
 factory = new WinFactory();  
 else  
 factory = new MacFactory();  
  
 Application app = new Application(factory);  
 app.Paint();  
 }

**Builder**

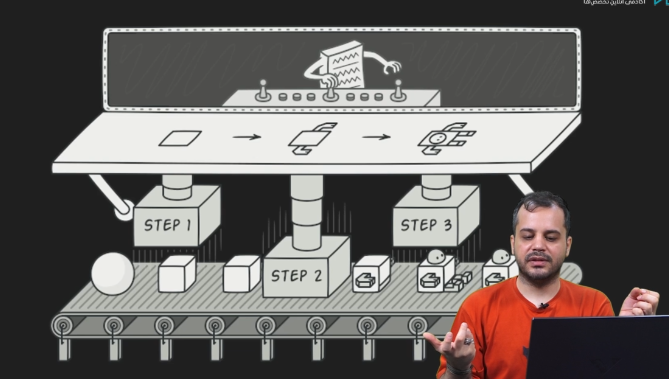
از نوع creational است و هدف مدیریت ایجاد یک complex object را مدیریت کنیم. هدف ایجاد object از یک کلاس است.

**Problem**

می خواهیم یک ساختمان بسازیم و در فرآیند ساخت قصد داریم که بخش های به آن اضافه کنیم مثل استخر حیاط و فضای و سبز و ... با استفاده از روش builder این امر میسر است یعنی متد های مخصوص اضافه کردن این feature ها داریم که کار را برای ما انجام می دهند.

نکته : مهم انجام مرحله به مرحله افزودن ویژگی ها است.

یک مفهوم داریم قصد ساخت object از آن داریم و مرحله به مرحله به آن می افزاییم.



می توانیم قسمت های را حذف کنیم از فرآیند و حتما لازم نیست همه مراحل را انجام بدهیم.

شکستن فرآیند و تقسیم به مراحل کوچک تر که با در کنار هم قرار گرفتن تکمیل می شود.

فرآیند ایجاد کردن باید اجازه بدهد که نمایده های مختلفی از object که ایجاد می شود ساخت.

دارای 2 حالت پیاده سازی است fluent , non fluent

nonfluent همان حالت عادی است

Fluent به گونه ی نوشته می شود که تمامی متد های ما مقدار بازگشتی از نوع همان کلاس را دارند که سبب می شود بتوانیم همانند یک دنباله تمامی متد هارو پشت سر هم صدا کنیم.

Gpt

برای ساخت اشیاء پیچیده استفاده می‌شود. این الگو به شما اجازه می‌دهد تا فرآیند ساخت یک شیء را از نمایش نهایی آن جدا کنید، به طوری که همان فرآیند ساخت می‌تواند نمایش‌های مختلفی از شیء را ایجاد کند.

**کاربردهای Builder Pattern**

* زمانی که ساخت یک شیء شامل مراحل متعدد است.
* وقتی که اشیاء با ترکیب‌های مختلف از ویژگی‌ها باید ساخته شوند.
* زمانی که می‌خواهیم از **constructor های پرپارامتر** اجتناب کنیم (مثلاً بیش از ۴ یا ۵ پارامتر).
* زمانی که می‌خواهید **خوانایی کد را افزایش دهید**.

**ساختار کلی Builder Pattern**

الگوی Builder معمولاً شامل ۴ بخش است:

1. **Product**: شیء پیچیده‌ای که ساخته می‌شود.
2. **Builder**: رابط (interface) یا کلاس انتزاعی برای تعریف مراحل ساخت.
3. **ConcreteBuilder**: پیاده‌سازی مراحل ساخت برای تولید یک نوع خاص از Product.
4. **Director**: کلاس مسئول اجرای ترتیب ساخت توسط Builder.

**مزایا**

* کد تمیزتر و قابل خواندن‌تر.
* جداسازی منطق ساخت از منطق استفاده از شیء.
* مناسب برای ایجاد اشیاء immutable.
* انعطاف‌پذیری در تعریف نحوه ساخت (build logic).

**معایب**

* پیچیدگی بیشتر نسبت به استفاده ساده از constructor.
* ممکن است برای پروژه‌های کوچک اضافه‌کاری محسوب شود.

# Builder Design Pattern in C# - Example

**Product - Computer class**

public class Computer  
{  
 public string CPU { get; set; }  
 public string RAM { get; set; }  
 public string Storage { get; set; }  
 public string GPU { get; set; }  
  
 public override string ToString()  
 {  
 return $"CPU: {CPU}, RAM: {RAM}, Storage: {Storage}, GPU: {GPU}";   
 }  
}

**Builder Interface - IComputerBuilder**

public interface IComputerBuilder  
{  
 void SetCPU();  
 void SetRAM();  
 void SetStorage();  
 void SetGPU();  
 Computer GetComputer();  
}

**Concrete Builder - GamingComputerBuilder**

public class GamingComputerBuilder : IComputerBuilder  
{  
 private Computer \_computer = new Computer();  
  
 public void SetCPU() => \_computer.CPU = "Intel Core i9";  
 public void SetRAM() => \_computer.RAM = "32GB DDR5";  
 public void SetStorage() => \_computer.Storage = "1TB NVMe SSD";  
 public void SetGPU() => \_computer.GPU = "NVIDIA RTX 4090";  
 public Computer GetComputer() => \_computer;  
}

**Director - ComputerDirector**

public class ComputerDirector  
{  
 private IComputerBuilder \_builder;  
  
 public ComputerDirector(IComputerBuilder builder)  
 {  
 \_builder = builder;  
 }  
  
 public void BuildComputer()  
 {  
 \_builder.SetCPU();  
 \_builder.SetRAM();  
 \_builder.SetStorage();  
 \_builder.SetGPU();  
 }  
  
 public Computer GetComputer() => \_builder.GetComputer();  
}

**Usage - Program class**

class Program  
{  
 static void Main()  
 {  
 IComputerBuilder builder = new GamingComputerBuilder();  
 ComputerDirector director = new ComputerDirector(builder);  
 director.BuildComputer();  
 Computer computer = director.GetComputer();  
  
 Console.WriteLine(computer);  
 }  
}

**Factory Method**

از نوع creational است برای ایجاد object ها کاربرد دارد.

متدی است که به ما اجازه می دهد که override شود در factory های مختلف از آن استفاده کرد و instance های مختلفی را دریافت میکند.

**Problem**

**Common = مشترک**

در یک رستوران غدا های مختلفی طبخ می شود غذای ایرانی ایتالیای و چینی اما موضوع مشترک بین همه این ها غدا بودن است پس ما یک interface خواهیم داشت که به ما غذا می دهد و سبک های مختلف می توانند instance از آن بسازند.

**UseCase**

زمانی که می خواهید برای کلاس های helper اطلاعاتی را localize کنید و آن ها delegate یا نماینده باشند.

رابط IProduct را داریم و از آن productA , productB را می سازیم.

**Gpt**

**تعریف:**

الگوی **Factory Method** به ما اجازه می‌دهد که **ایجاد اشیاء را به زیرکلاس‌ها واگذار کنیم**، بدون آن‌که کلاس استفاده‌کننده از شی، وابسته به کلاس اطلاع دقیق باشد. این الگو زمانی مفید است که تصمیم‌گیری درباره نوع دقیق شیء در زمان اجرا مشخص می‌شود، نه در زمان کامپایل.

**کاربردها:**

1. زمانی که نمی‌دانید در زمان طراحی دقیقاً چه کلاس‌هایی باید ساخته شوند.
2. زمانی که کلاس‌ها با واسط یا کلاس پایه‌ای تعریف شده‌اند، ولی پیاده‌سازی خاص در زیرکلاس‌ها قرار دارد.
3. برای جداسازی منطق ساخت از استفاده از اشیاء (Separation of Concerns).
4. برای ایجاد سیستم‌های انعطاف‌پذیر و قابل گسترش با کلاس‌های جدید.

**مزایا:**

* اصل باز-بسته بودن (Open/Closed Principle): کد باز برای توسعه، بسته برای تغییر.
* جدا شدن مسئولیت ساخت از استفاده.
* افزایش انعطاف و تست‌پذیری.

**معایب:**

* افزایش پیچیدگی با افزودن کلاس‌های زیاد.
* ممکن است در پروژه‌های ساده بیش از حد باشد.

# Factory Method Design Pattern in C#

## 1. Interface پیام

public interface IMessage  
{  
 void Send(string to, string message);  
}

## 2. کلاس‌های مختلف پیام

public class EmailMessage : IMessage  
{  
 public void Send(string to, string message)  
 {  
 Console.WriteLine($"Sending Email to {to}: {message}");  
 }  
}  
  
public class SmsMessage : IMessage  
{  
 public void Send(string to, string message)  
 {  
 Console.WriteLine($"Sending SMS to {to}: {message}");  
 }  
}

## 3. سازنده پیام (Factory)

public abstract class MessageFactory  
{  
 public abstract IMessage CreateMessage();  
}  
  
public class EmailMessageFactory : MessageFactory  
{  
 public override IMessage CreateMessage()  
 {  
 return new EmailMessage();  
 }  
}  
  
public class SmsMessageFactory : MessageFactory  
{  
 public override IMessage CreateMessage()  
 {  
 return new SmsMessage();  
 }  
}

## 4. استفاده در برنامه اصلی

class Program  
{  
 static void Main(string[] args)  
 {  
 MessageFactory factory;  
  
 factory = new EmailMessageFactory();  
 var email = factory.CreateMessage();  
 email.Send("test@example.com", "Hello via Email!");  
  
 factory = new SmsMessageFactory();  
 var sms = factory.CreateMessage();  
 sms.Send("123456789", "Hello via SMS!");  
 }  
}

**Prototype**

از نوع creational است و هدف ایجاد یک object جدید از یک instance است یعنی clone کردن یا copy کردن از instance موجود بدون استفاده از کلمه کلیدی new .

دارای روش های مختلف:

Manual

Constractor

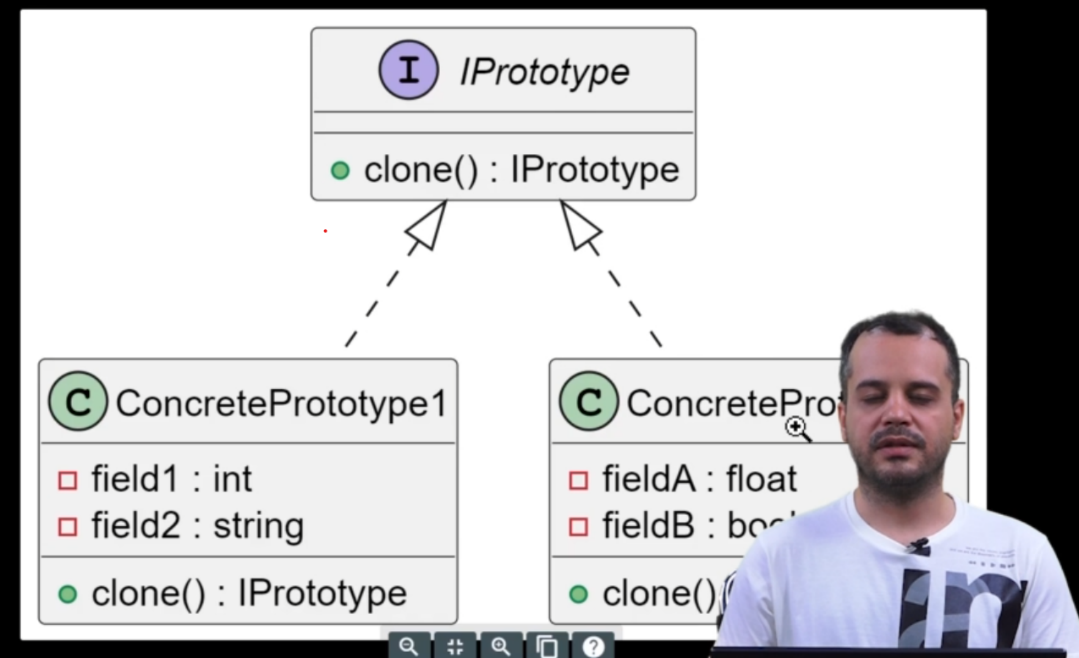
Shallow copy or dep copy دیپ حالتی است که instance ما تو در تو است داخلش کلاس دیگری داره و تو در تو است میخواهیم اون ها هم clone شود اما حالت shallow داخل نمی رود.

Sterilization با استفاده از json

امکانات زبان مثل IClone , Clone

نکته »

میتوانیم موقع clone کردن تصمیم بگیریم به چه صورت انجام شود دقیقا مشابه instance موجود باشد یا تفاوت داشته باشد.



Gpt

**تعریف:**

الگوی **Prototype** به ما اجازه می‌دهد تا با استفاده از یک **شیء نمونه (prototype)**، **اشیاء جدیدی را با کپی کردن آن نمونه ایجاد کنیم**، به جای اینکه از new برای ساختن اشیاء استفاده کنیم. این روش مخصوصاً زمانی مفید است که ساختن شیء بسیار **هزینه‌بر یا پیچیده** باشد.

**کاربردها:**

* زمانی که ایجاد اشیاء جدید هزینه‌بر است (مثلاً محاسباتی یا از نظر منابع).
* زمانی که می‌خواهید ساختار یک شیء را در حالت فعلی آن کپی کرده و مستقل استفاده کنید.
* زمانی که سیستم باید اشیاء را مستقل از کلاس‌های خاص آن‌ها ایجاد کند.
* زمانی که می‌خواهید نسخه‌های مختلفی از یک شیء را با تنظیمات متفاوت داشته باشید.

**مزایا:**

* افزایش کارایی با جلوگیری از ساخت اشیاء پیچیده از صفر.
* امکان سفارشی‌سازی سریع اشیاء با کپی گرفتن.
* انعطاف‌پذیری بالا در سیستم‌های نیازمند به تولید سریع اشیاء مشابه.

**معایب:**

* نیاز به پیاده‌سازی رابط Clone برای همه کلاس‌ها.
* امکان بروز خطا در کپی‌های عمیق (deep copy) اگر به درستی انجام نشود.

**تفاوت بین Shallow Copy و Deep Copy:**

* **Shallow Copy (کپی سطحی)**: فقط آدرس‌ها کپی می‌شود؛ اگر شیء دارای ارجاع به اشیاء دیگر باشد، کپی هم به همان اشیاء اشاره می‌کند.
* **Deep Copy (کپی عمیق)**: تمام داده‌های شیء اصلی به طور کامل کپی می‌شوند؛ کپی جدید به اشیاء جدید اشاره دارد.

# Prototype Design Pattern in C#

**1. کلاس پایه Shape**

public abstract class Shape  
{  
 public string Color { get; set; }  
  
 public abstract Shape Clone();  
}

**2. کلاس‌های مشتق شده: Circle و Rectangle**

public class Circle : Shape  
{  
 public int Radius { get; set; }  
  
 public override Shape Clone()  
 {  
 return new Circle  
 {  
 Radius = this.Radius,  
 Color = this.Color  
 };  
 }  
}  
  
public class Rectangle : Shape  
{  
 public int Width { get; set; }  
 public int Height { get; set; }  
  
 public override Shape Clone()  
 {  
 return new Rectangle  
 {  
 Width = this.Width,  
 Height = this.Height,  
 Color = this.Color  
 };  
 }  
}

**3. استفاده در برنامه اصلی**

class Program  
{  
 static void Main(string[] args)  
 {  
 Circle originalCircle = new Circle { Radius = 10, Color = "Red" };  
 Circle clonedCircle = (Circle)originalCircle.Clone();  
  
 Console.WriteLine($"Original: Radius={originalCircle.Radius}, Color={originalCircle.Color}");  
 Console.WriteLine($"Clone: Radius={clonedCircle.Radius}, Color={clonedCircle.Color}");  
 }  
}

Singleton

از نوع creational است و هدف آن اطمینان از این که یک object از کلاس فقط یکی داشته باشیم. به طور مثال فقط یک filesystem یا windowManager داشته باشیم. فقط یک instance داشته باشیم.

باید نحوه دسترسی مشخص باشد.

Problem

برای جلوگیری از ساخت instance های مختلف کاربرد دارد فقط یک object داریم.

کاربرد در:

Dependency injection

Inversion of control

Ambient context

Service locators

در مواقعی که نیاز داریم فقط یک instance از کلاس باشد استفاده می کنیم باید life cycle و point of access آن مشخص باشد.

**نکته : باید constructor آن از نوع private باشد**

**نکته :**

**در حالتی که یک کلاس که از الگوی singleton استفاده می کند ایجاد می کنید باید مراقب thredSafe بودن آن باشید. اگر چند thread داخل شود دیگر حالت singleton ندارد. باید از lock استفاده کرد یک ترد وارد شود.**

**نوع Lazy<T> تا زمانی که دات نزنیم یک instance نمی سازد کاربرد در performance**

**Gpt**

تعریف Singleton Pattern:

الگوی Singleton تضمین می‌کند که فقط یک نمونه از یک کلاس در کل برنامه وجود داشته باشد و راهی جهانی برای دسترسی به آن فراهم می‌کند.

**مزایا:**

* کنترل بر تعداد نمونه‌های کلاس.
* اشتراک‌گذاری وضعیت بین بخش‌های مختلف برنامه.
* مناسب برای کلاس‌هایی که ایجادشان هزینه‌بر است.

**معایب:**

* ایجاد وابستگی شدید (tight coupling) به کلاس Singleton.
* مشکل در تست واحد (Unit Test) به دلیل وضعیت سراسری (global state).
* در برنامه‌های چندریسمانی (multi-threaded) نیاز به پیاده‌سازی ایمن‌تر دارد.

**نکات مهم**

* از **sealed** برای جلوگیری از ارث‌بری استفاده می‌شود.
* سازنده باید **private**  باشد تا شیء جدید نتوان ساخت.
* در محیط‌های multi-thread حتماً باید از نسخه‌هایی با Lazy یا lock استفاده کرد.
* در ASP.NET یا Web API به خاطر ماهیت multi-threading بودن محیط، نسخه Lazy مناسب‌تر است.

**Singleton Design Pattern in C#**

**پیاده‌سازی ساده Singleton در C#**

کد پیاده‌سازی Singleton:

public sealed class Singleton  
{  
 private static readonly Singleton instance = new Singleton();  
  
 private Singleton()  
 {  
 // Initialization code  
 }  
  
 public static Singleton Instance  
 {  
 get  
 {  
 return instance;  
 }  
 }  
  
 public void DoSomething()  
 {  
 Console.WriteLine("Singleton instance is working.");  
 }  
}

**استفاده از Singleton**

کد استفاده از Singleton:

class Program  
{  
 static void Main(string[] args)  
 {  
 Singleton s1 = Singleton.Instance;  
 Singleton s2 = Singleton.Instance;  
  
 s1.DoSomething();  
  
 Console.WriteLine(object.ReferenceEquals(s1, s2)); // خروجی: True  
 }  
}

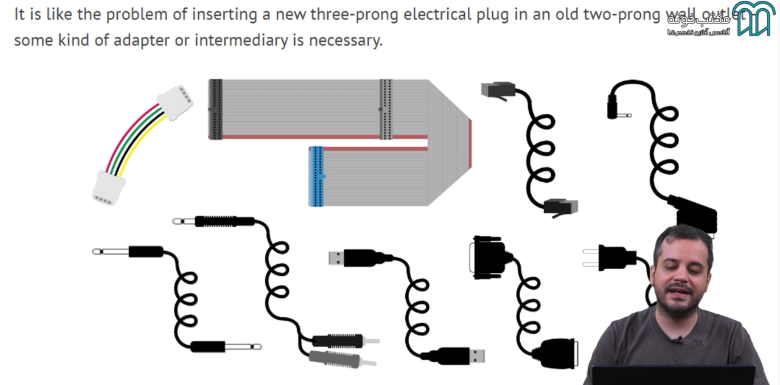
**نسخه Lazy و Thread-Safe**

کد نسخه Lazy و Thread-Safe:

public sealed class LazySingleton  
{  
 private static readonly Lazy<LazySingleton> lazyInstance =  
 new Lazy<LazySingleton>(() => new LazySingleton());  
  
 private LazySingleton()  
 {  
 // Initialization code  
 }  
  
 public static LazySingleton Instance  
 {  
 get  
 {  
 return lazyInstance.Value;  
 }  
 }  
  
 public void DoWork()  
 {  
 Console.WriteLine("Lazy Singleton is working.");  
 }}

**Adapter Design Pattern**

زمانی که یک سری interface داریم که incompatible به هم دیگر نمی خورد کمک می کند که این ها در کنار هم دیگر قرار گیرند آن های که Incompatible هستند را Compatible می کند.

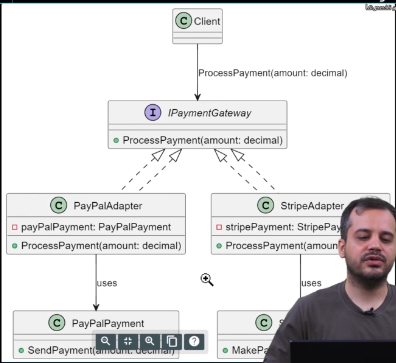


Problem

فرض کنید که یک فروشگاه دارید که با روش های مختلفی می توان اقدام به پرداخت کرد مثل بانک ملت و ملی و پاسارگارد هر کدام درگاه خود را دارند

با استفاده از adapter می توان Interface ایجاد کرد که با استفاده از آن integrate یا یکپارچه کرد.

یک روش برای ترجمه کردن است به متد دلخواه ما در واقع سازگار می کند compatible سازی انجام می دهد.



Gpt

هدف این الگو این است که **دو کلاس ناسازگار را طوری به هم وصل کند که بتوانند با هم کار کنند**، بدون اینکه نیازی به تغییر در کد کلاس‌های اصلی باشد.

**مفهوم کلی الگوی Adapter**

Adapter مانند یک **مبدل** عمل می‌کند. فرض کن یک شارژر موبایل داری که فقط USB-C می‌خورد اما کابلت USB معمولی است. شما یک **آداپتور** می‌گیری تا این دو به هم بخورند.

کاربردهای Adapter

1. استفاده مجدد از کدهای موجود بدون تغییر آنها
2. یکپارچه‌سازی کلاس‌هایی با رابط‌های متفاوت
3. اتصال سیستم‌های قدیمی (Legacy) با سیستم‌های جدید
4. استفاده از کلاس‌هایی که رابط مورد نظر ما را ندارند

## پیاده‌سازی الگوی Adapter در C#

### سناریو:

فرض کنید شما یک سیستم پرداخت دارید که از رابط INewPayment استفاده می‌کند، اما کتابخانه‌ای قدیمی به نام OldPaymentSystem دارید که متد متفاوتی دارد.

Adapter Design Pattern in C# - Example

public interface IUserInfo  
{  
 void DisplayUser();  
}

**کلاس ناسازگار (LegacyUser)**

public class LegacyUser  
{  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
  
 public void ShowFullName()  
 {  
 Console.WriteLine($"User: {FirstName} {LastName}");  
 }  
}

**کلاس Adapter**

public class UserAdapter : IUserInfo  
{  
 private LegacyUser \_legacyUser;  
  
 public UserAdapter(LegacyUser legacyUser)  
 {  
 \_legacyUser = legacyUser;  
 }  
  
 public void DisplayUser()  
 {  
 \_legacyUser.ShowFullName();  
 }  
}

**استفاده از Adapter در برنامه**

class Program  
{  
 static void Main(string[] args)  
 {  
 LegacyUser legacy = new LegacyUser  
 {  
 FirstName = "Sara",  
 LastName = "Mohammadi"  
 };  
  
 IUserInfo user = new UserAdapter(legacy);  
 user.DisplayUser();  
 }  
}

**خروجی**

User: Sara Mohammadi

Bridge

از نوع stractural است و کاری که انجام می دهد این است که abstraction را از implementation جدا سازی می کند که بتوانند جداگانه از هم کارکنند.

در واقع پلی بین 2 سرویس بزرگ یا 2 ماژول است.

مثال:

یک بازی داریم که کلاس abstract است و در پلتفرم های مختلف قصد داریم بازی را اجرا کنیم با استفاده از bridge می توان برای هر پلتفرم برنامه را توسعه دهید.

Bridge vs adapter

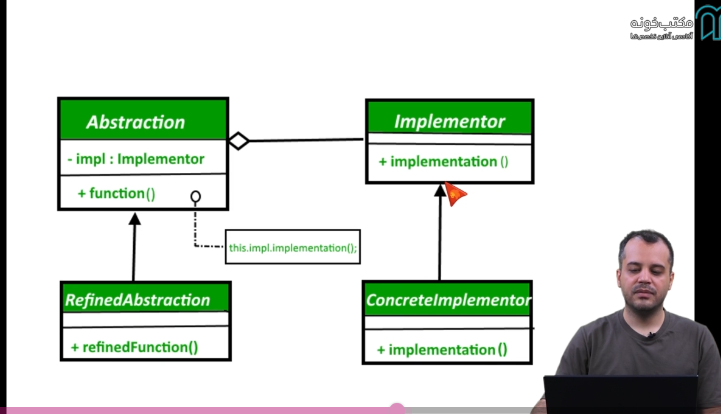
در bridge قسمت های از app که مستقل هستند این را develop می کنیم

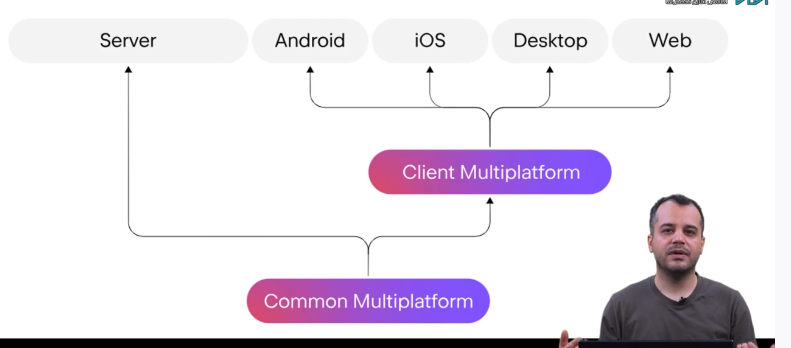
اما adapter در خود app هست و نه قسمت های مختلف داخل سرویس خودم یا App خودم.

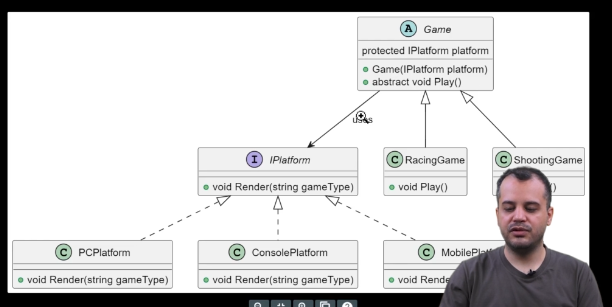
useCase

موقعی که بین پیاده سازی های مختلف در زمان runtime می خواهیم switch کنیم و می خواهیم پیاده سازی را از client خود مخفی کنیم.

همجنین برای share کردن کد خود در پلتفرم های مختلف.







Gpt

تعریف کلی Bridge Pattern

الگوی Bridge کمک می‌کند که وابستگی‌های بین abstraction (مثلاً کلاس‌های بالا‌دستی) و implementation (مثلاً کلاس‌های وابسته یا پایین‌دستی) کاهش یابد. به زبان ساده‌تر: کلاس‌هایی که “چه کاری انجام می‌دهند” را از کلاس‌هایی که “چگونه انجام می‌دهند” جدا می‌کنیم.

**کاربردهای الگوی Bridge**

1. زمانی که می‌خواهید یک کلاس در **دو یا چند بعد (dimension)** توسعه یابد.
2. وقتی که می‌خواهید **کدتان را از تغییرات implementation مستقل نگه دارید**.
3. وقتی که نیاز دارید پیاده‌سازی‌های مختلف را **در زمان اجرا تغییر دهید**.
4. مناسب برای سیستم‌هایی که از **ترکیب چندین کلاس مختلف** با رفتار متفاوت تشکیل شده‌اند (ترکیب ماتریسی).

### سناریو:

فرض کنید ما چند **شکل (Shape)** داریم (مثل دایره، مربع) و هر کدام می‌توانند **با روش‌های مختلفی رسم شوند** (مثلاً با OpenGL، DirectX یا با رنگ متفاوت). ما نمی‌خواهیم در هر کلاس Shape، منطق مربوط به رسم را پیاده‌سازی کنیم.

**مزایای الگوی Bridge**

* جداسازی abstraction و implementation.
* کاهش پیچیدگی کلاس‌ها.
* افزایش **قابلیت توسعه (extensibility)** بدون تغییر کدهای موجود.
* استفاده در **سیستم‌های چندبعدی** که ترکیب چند کلاس لازم است.

Bridge Design Pattern in C#

**1. رابط IMessageSender**

public interface IMessageSender  
{  
 void SendMessage(string subject, string body);  
}

**2. پیاده‌سازی‌های مختلف از IMessageSender**

public class EmailSender : IMessageSender  
{  
 public void SendMessage(string subject, string body)  
 {  
 Console.WriteLine($"ارسال ایمیل: {subject} - {body}");  
 }  
}

public class SMSSender : IMessageSender  
{  
 public void SendMessage(string subject, string body)  
 {  
 Console.WriteLine($"ارسال پیامک: {subject} - {body}");  
 }  
}

**3. کلاس Message (Abstraction)**

public abstract class Message  
{  
 protected IMessageSender sender;  
  
 public Message(IMessageSender sender)  
 {  
 this.sender = sender;  
 }  
  
 public abstract void Send(string subject, string body);  
}

**4. کلاس‌های پیاده‌سازی‌شده**

public class NormalMessage : Message  
{  
 public NormalMessage(IMessageSender sender) : base(sender) { }  
  
 public override void Send(string subject, string body)  
 {  
 sender.SendMessage(subject, body);  
 }  
}  
  
public class UrgentMessage : Message  
{  
 public UrgentMessage(IMessageSender sender) : base(sender) { }  
  
 public override void Send(string subject, string body)  
 {  
 sender.SendMessage("URGENT: " + subject, body.ToUpper());  
{{

**5. استفاده از Bridge در برنامه**

class Program  
{  
 static void Main(string[] args)  
 {  
 IMessageSender email = new EmailSender();  
 IMessageSender sms = new SMSSender();  
  
 Message normalEmail = new NormalMessage(email);  
 Message urgentSMS = new UrgentMessage(sms);  
  
 normalEmail.Send("جلسه", "جلسه ساعت 10 برگزار می‌شود.");  
 urgentSMS.Send("خطا", "سیستم دچار خطا شده است!");  
 }  
}

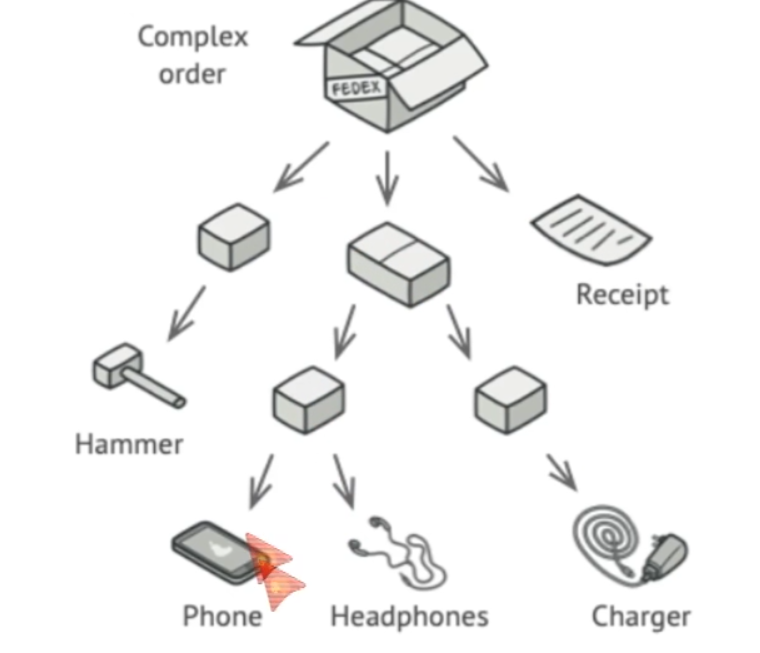
Composite

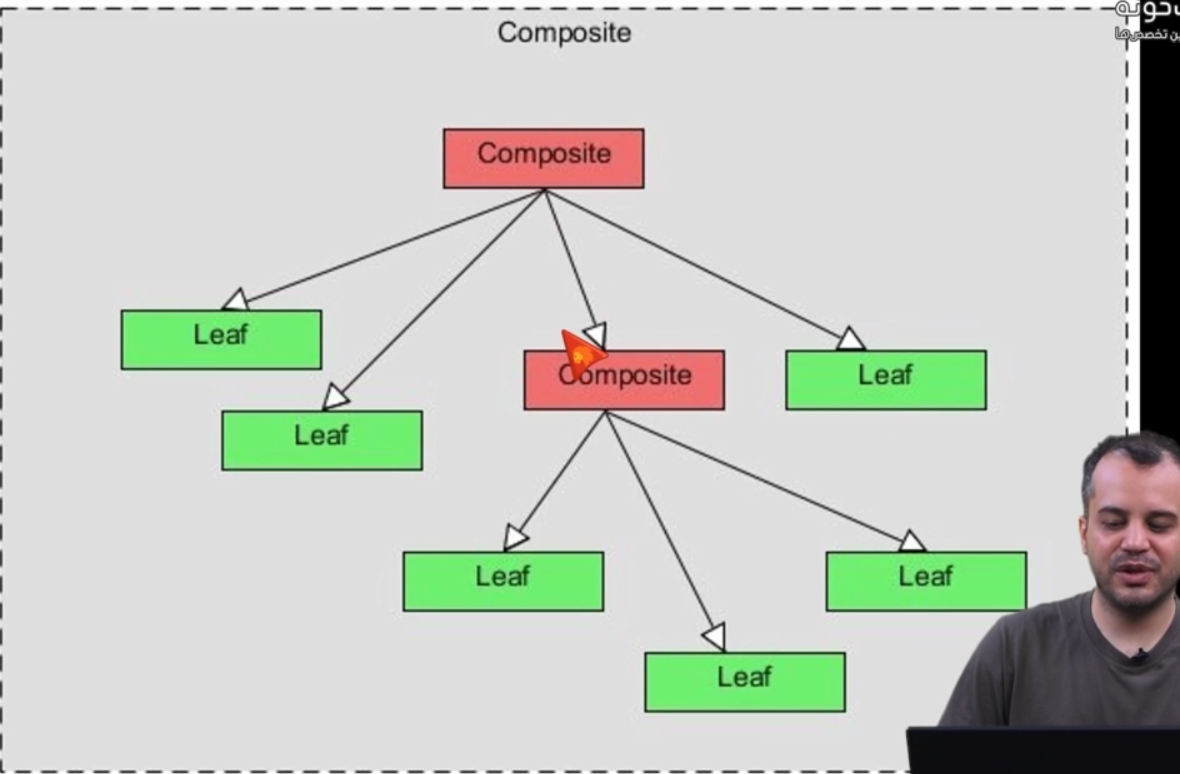
از نوع structural است و هدف آن این است که یک نمودار درختی را در نظر بگیرید این الگوی طراحی می تواند مشکلاتی که در این ساختار وجود دارد را حل کند.

Problem

به طور مثال یک فروشگاه اینترنتی داریم که موجودیتی به نام Order در آن وجود دارد و این موجودیت داخل خود می تواند OrderItem نیز دارشته باشد.

حل مسئله به کمک Compostie برای برای جدا کردن این 2 از یک دیگر کاربرد دارد.





اجزا:

Component

مشخص می کند که تعریف ما چیست؟ یک کلاس abstract یا interface است در واقع یک تعریف کلی داریم از مسئله خود از composite خود. تعریف واحد برای composite یک Component است.

Leaf

مقدار نهایی است که دیگر فرزند ندارد

Composite

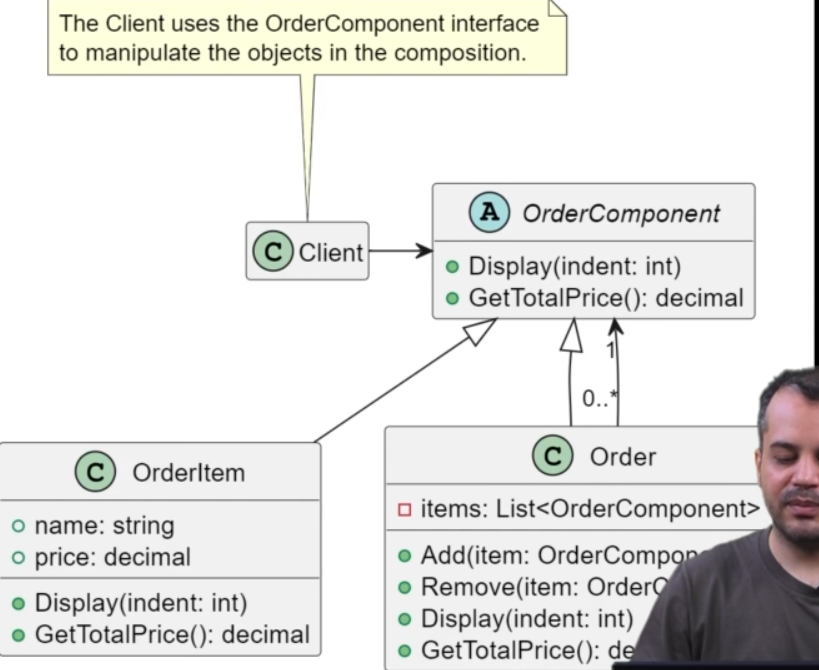
می تواند شامل یک composite دیگر یا leaf باشد.

Client

کسی است که در نهایت از Component ما استفاده می کند.

UseCase

زمانی که یک سلسله مراتبی از object ها را می خواهیم یکی در نظر بگیریم و می خواهیم که client ما فرق بین Leaf , Compositeرا متوجه نشود و در نظر نگیرد ignore کند



### مفهوم الگوی Composite:

الگوی Composite اجازه می‌دهد که اشیاء را به صورت درختی سازماندهی کنیم، به طوری که بتوان با یکسان‌سازی رفتار، به راحتی با اشیاء مجرد (Leaf) و اشیاء مرکب (Composite) به یک شکل رفتار کرد.  
یعنی اشیاء ساده و ترکیبی هر دو از یک اینترفیس یا کلاس پایه مشترک ارث‌بری می‌کنند، و می‌توان آن‌ها را به صورت یکپارچه و بدون توجه به تفاوت‌های ساختاری‌شان مدیریت کرد.

### کاربرد:

* وقتی می‌خواهید یک ساختار درختی مثل فایل‌ها و پوشه‌ها، بخش‌های گرافیکی (widgets)، منوها و زیرمنوها، یا هر نوع ساختار تودرتو داشته باشید.
* وقتی بخواهید عملیات مشابهی را روی اشیاء ساده و ترکیبی به یک صورت انجام دهید.
* وقتی بخواهید ساختار داده‌ای پیچیده را به صورت ساده و قابل مدیریت درآورید.

### اجزای الگو:

1. **Component** (کامپوننت): یک اینترفیس یا کلاس پایه که عملیات مشترک را تعریف می‌کند.
2. **Leaf** (برگ): اشیاء ساده‌ای که زیرمجموعه‌ای ندارند.
3. **Composite** (مرکب): اشیائی که ممکن است شامل چندین کامپوننت (Leaf یا Composite) باشند و عملیات روی آن‌ها را مدیریت می‌کنند.

Decorator

از نوع Structural است. در framework ها و library ها بیشتر کاربرد دارد.

هدف : responsibility ها یا مسئولیت های مختلفی را به یک object به صورت Dynamicly اضافه می کند به یک object و functionality آن را extend می کند.

افزودن یک مسئولیت یا رفتار خاصی به یک object را انجام می دهد.

مثلا یک اتاق که به شکل های مختلف دکور می کنیم.

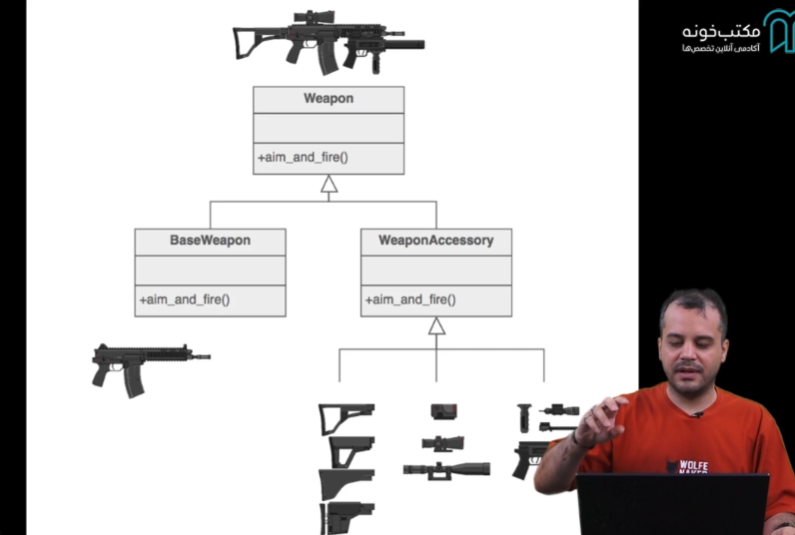
عملیات دکوریت کردن را انجام می دهیم.

Problem

نیاز داریم که یک سری feature را به یک object اضافه کنیم بدون تغییر در ساختار آن.

با استفاده از open/close می توانیم برای extension کار کنیم نه برای modify پس بنا براین می توانیم با استفاده از decorator رفتار جدید را به صورت dynamically اضافه کنیم بدون تغییر ساختار.

به آن wrapper نیز میگویند.



مفاهیم:

Component interface

یک interface است که مفهوم را بیان می کند.

Concrete component

پیاده سازی را در اینجا انجام می دهیم.

Decorator

مفهوم های جدید را اینجا اضافه می کنیم که یک interface است

Concrete decorator

پیاده سازی کلاس decorator را در این انجام می دهیم.

**نکته :**

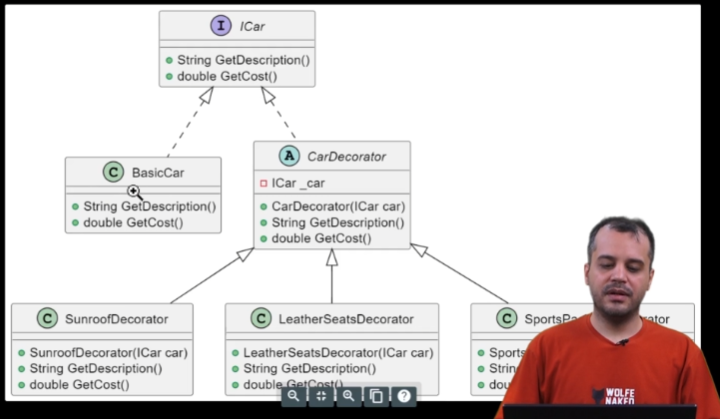
**مشابه chain of responsibility است که هر دور به صورت recursive پیاده سازی می شوند.**

**این امکان وجود دارد که چندین Decorator را به یکدیگر متصل کرد.**

**مثالا jango nodejs , asp هر کدام مجموعه ای از pipeline ها دارند که مراحل را به یک دیپر pass می دهند.**

**موارد استفاده :**

زمانی که نیاز است در runtime یک feature را به یک object اضافه کنیم بدون شکستن کد آن توانی گرفتن آن ویژگی وجود داشته باشد.



Gpt

## Decorator تزئینگر چیست؟

این الگو جزء **الگوهای ساختاری (Structural)** می‌باشد.  
کار این الگو اینه که بتوانیم **بدون ایجاد subclasses جدید**، قابلیت‌ها و ویژگی‌های جدیدی به اشیاء اضافه کرده و آنها را **پویا** ( در هنگام اجرا) گسترش بدهیم.

## کاربردها:

➥ هنگامی که نمی‌خواهید از طریق **ارث‌بری** ویژگی اضافه کنید چون:

* ایجاد subclasses زیاد می‌شود.
* نمی‌خواهید در کد پایه تغییری دهید.

➥ زمانی که می‌خواهید بتوانید در **زمان اجرا** قابلیت‌ها را اضافه کرده یا بردارید.

➥ هنگامی که می‌خواهید از طریق **ترکیب** (composition) عمل کنید نه از طریق وراثت (inheritance).

## مثالی کاربردی:

سناریو:  
ما **یه اینترفیس از اطلاعیه (Notifier)** داریم.  
یه اطلاعیه پایه داریم (مثلاً از طریق **ایمیل** اطلاع می‌فرستد)،  
سپس می‌خواهیم بدون این‌که این اطلاعیه پایه عوض بشه، بتوانیم **SMS** هم کنار آن بفرستیم.

Decorator Design Pattern in C#

# 1️⃣ اینترفیس پایه (INotifier)

public interface INotifier  
{  
 void Send(string message);  
}

# 2️⃣ اطلاعیه پایه (EmailNotifier)

public class EmailNotifier : INotifier  
{  
 public void Send(string message)  
 {  
 Console.WriteLine($"ایمیل ارسال شد: {message}");  
  
 }  
}

# 3️⃣ Decorator پایه (NotifierDecorator)

public abstract class NotifierDecorator : INotifier  
{  
 protected INotifier \_notifier;  
  
 public NotifierDecorator(INotifier notifier)  
 {  
 \_notifier = notifier;  
 }  
 public abstract void Send(string message);  
}

# 4️⃣ اطلاعیه SMS (SMSNotifier)

public class SMSNotifier : NotifierDecorator  
{  
 public SMSNotifier(INotifier notifier) : base(notifier) { }  
 public override void Send(string message)  
 {  
 \_notifier.Send(message);  
 Console.WriteLine($"SMS ارسال شد: {message}");  
  
 }  
}

# 5️⃣ استفاده در برنامه

class Program  
{  
 static void Main(string[] args)  
 {  
 INotifier email = new EmailNotifier();  
  
 INotifier emailWithSMS = new SMSNotifier(email);  
  
 emailWithSMS.Send("سفارش شما ثبت شد.");  
 }  
}

# 6️⃣ نتیجه

اینکار اجازه می‌دهد بدون ایجاد subclasses جدید، قابلیت‌ها در هنگام اجرا اضافه شوند.  
این امر انعطاف‌پذیری بالاتری ایجاد کرده چون می‌شود از طریق ترکیب عمل کرده.  
  
خروجی:  
- ایمیل ارسال شد: سفارش شما ثبت شد.  
- SMS ارسال شد: سفارش شما ثبت شد.

Facade

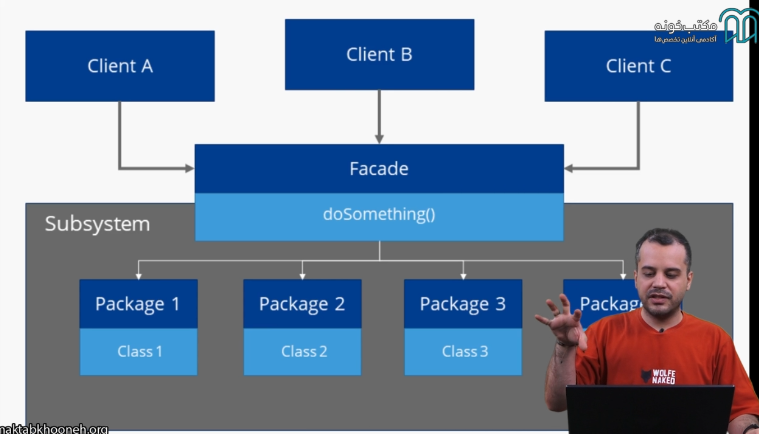
از نوع structural است. فساد.

هدف : به طور مثال سرویسی داریم که برای کنترل کل یک مجموعه کاربرد دارد مثلا ساختمان های مختلفی داریم که یک کارخانه را تشکیل می دهند و ارتباط بین ساختمان ها از طریق facade انجام می شود.

کنترل و مانیتور و صدا کردن را انجام می دهیم.

یک interface داریم در لایه بالا که زیرسیستم های من ار simplefie کرده است یا ساده سازی.

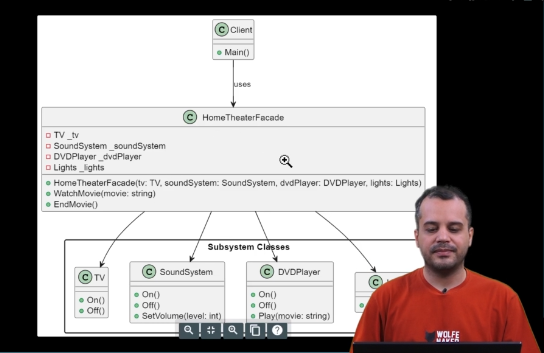
یک سرویس که روندی را برای انجام کاری تعریف می کند. مثلا سرویسی که فرآیند مربوط به خرید را از اول تا آخر کنترل می کند انبار سفارش پرداخت و تحویل.



مشکل : یک کتابخانه پیچیده داریم که می خواهیم کسی از آن استفاده کند در آن شی های ایجاد می شود وابتسگی های هست که برای کنترل این فرآیند از facade استفاده می کنیم.

کارکرد : simplify کردن Interface ساده کردن یک سیستم پیچیده زیر سیستم ها را یکپارچه می کند و مدیریت می کند زمانی که وابستگی ها زیاد است کنترل می کند.

می توان به صورت لایه بندی چندین facade نیز داشته باشیم برای کار های مختلف



Gpt

## Facade (نمای بیرونی) چیست؟

این الگو جزء **الگوهای ساختاری (Structural)** می‌باشد.  
هدف از این الگو اینه که **یک رابط ساده بر روی مجموعه‌ای از رابط‌ها و کلاس‌ها ایجاد کرده**.

➥ اینکار استفاده از **سیستم داخلی پیچیده** رو آسان می‌کند چون نیازی نیست مستقیم با همه اجزاء داخلی ارتباط برقرار کنید.  
➥ در واقع مثل **یک پیشخوان** عمل می‌کند که از طریق آن می‌شود از عملکردهای داخلی استفاده کرد.

## کاربردها:

➥ هنگامی که:

* **سیستم داخلی خیلی پیچیده باشد.**
* **نخواهید مشتری از همه این پیچیدگی آگاه بشه.**
* **یک رابط ساده می‌خواهید بر روی زیربخش‌ها ایجاد کنید.**
* می‌خواهید استفاده از API داخلی آسان‌تر بشه.

## مثالی کاربردی:

➥ مثلاً شما سه جزء جدا دارید:

* Inventory: چک می‌کند آیا کالا در انبار هست.
* Payment: مبلغ از حساب مشتری کسر می‌کند.
* Shipping: سفارش را برای مشتری می‌فرستد

## نتیجه:

➥ با استفاده از **Facade**، مشتری فقط با **یک رابط** ارتباط دارد.  
➥ اینکار استفاده از اجزاء داخلی را آسان کرده و آنها را از مشتری **پنهان** می‌کند.

Facade Design Pattern in C#

# 1️ اجزاء داخلی

public class Inventory  
{  
 public bool CheckStock(string productId)  
 {  
 Console.WriteLine($"بررسی موجودی {productId}.");  
 return true;  
 }  
}

public class Payment  
{  
 public bool DeductFunds(string accountId, decimal amount)  
 {  
 Console.WriteLine($"برداشت {amount} از حساب {accountId}.");  
 return true;  
 }  
}  
  
public class Shipping  
{  
 public void ShipProduct(string productId, string address)  
 {  
 Console.WriteLine($"ارسال {productId} به آدرس {address}.");  
 }  
}

# 2️ ایجاد Facade

public class OrderFacade  
{  
 private Inventory \_inventory = new Inventory();  
 private Payment \_payment = new Payment();  
 private Shipping \_shipping = new Shipping();  
  
 public void PlaceOrder(string accountId, string productId, decimal amount, string address)  
 {  
 if (\_inventory.CheckStock(productId) &&  
 \_payment.DeductFunds(accountId, amount))  
 {  
 \_shipping.ShipProduct(productId, address);  
 Console.WriteLine("سفارش با موفقیت ثبت شد.");  
 }  
 else  
 {  
 Console.WriteLine("سفارش ناموفق.");  
 }  
 }  
}

# 3️ استفاده از Facade

class Program  
{  
 static void Main(string[] args)  
 {  
 OrderFacade order = new OrderFacade();  
  
 order.PlaceOrder("acc123", "prod01", 100m, "تهران");  
  
 // نتیجه:  
 // بررسی موجودی prod01.  
 // برداشت 100 از حساب acc123.  
 // ارسال prod01 به آدرس تهران.  
 // سفارش با موفقیت ثبت شد.  
 }  
}

Proxy

از نوع structural است. به معنی نیابت نیز هست.

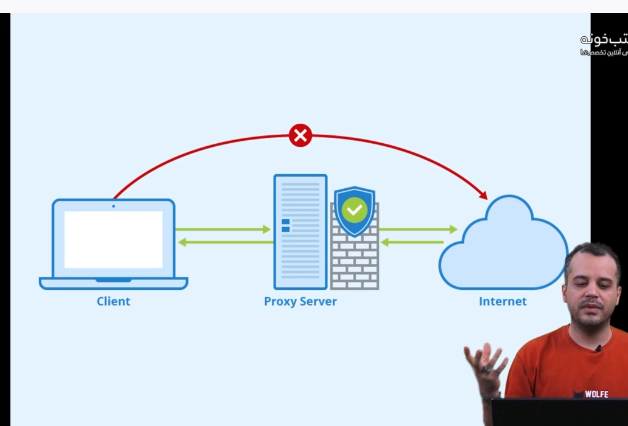
هدف : فرض کنید که یک سرویسی وجود دارد Proxy به ما کمک می کند که جایگزینی برای رسیدن به سرویس باشد.

فرض کنید دربی داریم که با استفاده از سنسور کارت خوان برای باز کردن آن اقدام می کنیم کاری که سنسور کارت خوان می کند به نیابت از در است برای رسیدن به آن.

کنترل دسترسی به در proxy کارت است.

یک proxy برای کنترل دسترسی به object اصلی کاربرد دارد و کنترل می کند.

دسترسی به internet از طریق proxy server ممکن است.



یک نوع بروکراسی است برای رسیدن به هدف است یعنی لازم به طی کردن مراحلی برای رسیدن به هدف.

UseCase

Lazy initialization (virtual proxy)

فرض کنید یک object بزرگ دارید که نیاز نیست هر موقع ایجاد شود با استفاده از این روش می توان تاخیر در ایجاد اضافه کرد.

AcessControl

محدود کردن دسترسی به یک object بنا به شرایط ما

Local execution of remote service(proxy remote)

مدیریت ارتباط با یک سرویس از طریق proxy مدیریت شود

Logging request(logging proxy)

پراکسی لاگ زدن سیتم درخواست های وارده را لاگ بزند و ثبت کند

Cashing request result(caching proxy)

پراکسی که دیتای که لود می شود را cash می کند و در درخواست های وارد شده بعدی دیتای کش را به ما می دهد.

Smart Reference

برای object های که حجیم هستند مدیریت resource ها و optimize کردن آن.

Proxy vs Façade

این دو بسیار شبیه به یک دیگر هستند و هر دو مدیریت یک compelex entity را مدیریت می کنند اما برعکس facade که خودش یک interface است proxy دارای یک interface که بسیار به object اصلی شبیه است.

Decorator , proxy نیز دارای یک ساختار بسیار شبیه به هم است اما هر کدام دارای هدف متفاوتی است.

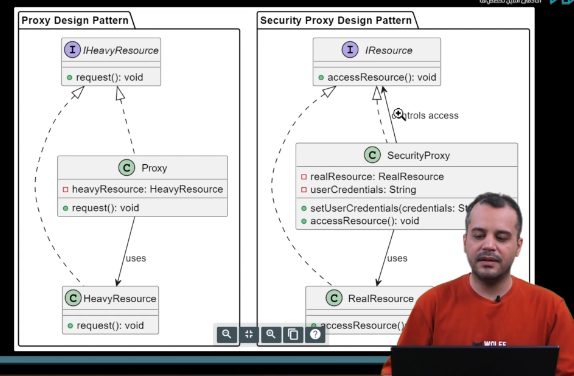
Proxy کار مدیریت life cycle سرویس و object خود را انجام می دهد اما Decorator توسط کلاینت کنترل می شود.

When use

زمانی که یک جایگزین برای دسترسی به یک object داشته باشیم

زمانی که یک extra step می خواهیم برای یک دسترسی داشته باشیم – قدم اضافه برای رسیدن به یک object برای دسترسی به آن.

یک روش برای کاهش پیچیدگی و محافظت از کامپوننت اصلی است یا object ما.



Behavioral – Command

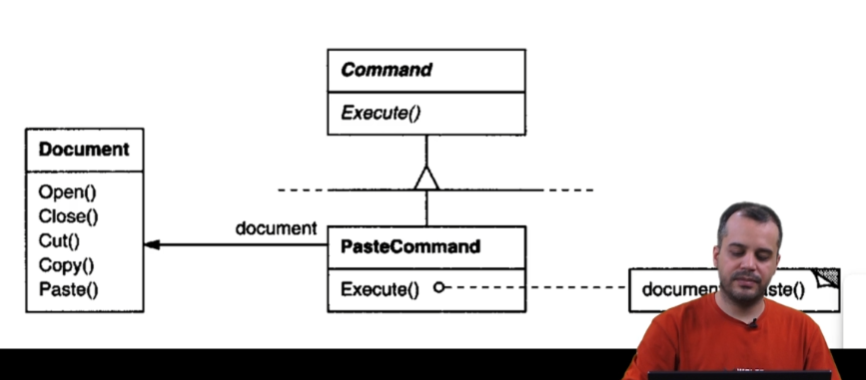
مهم و کاربردی است.

زمانی که یک موجودیتی داریم که به مرور زمان compelex می شود و می خواهیم به طور مثال به ان ویژگی اضافه کنیم بهتر است در این حالت از command استفاده کنیم.

Problem

مثلا زمانی که بخواهیم از یک وضعیت به وضعیت دیگر یا وضعیت قبلی برویم یا مثلا کاری را می خواهیم انجام دهیم یا از آن برگردیم.

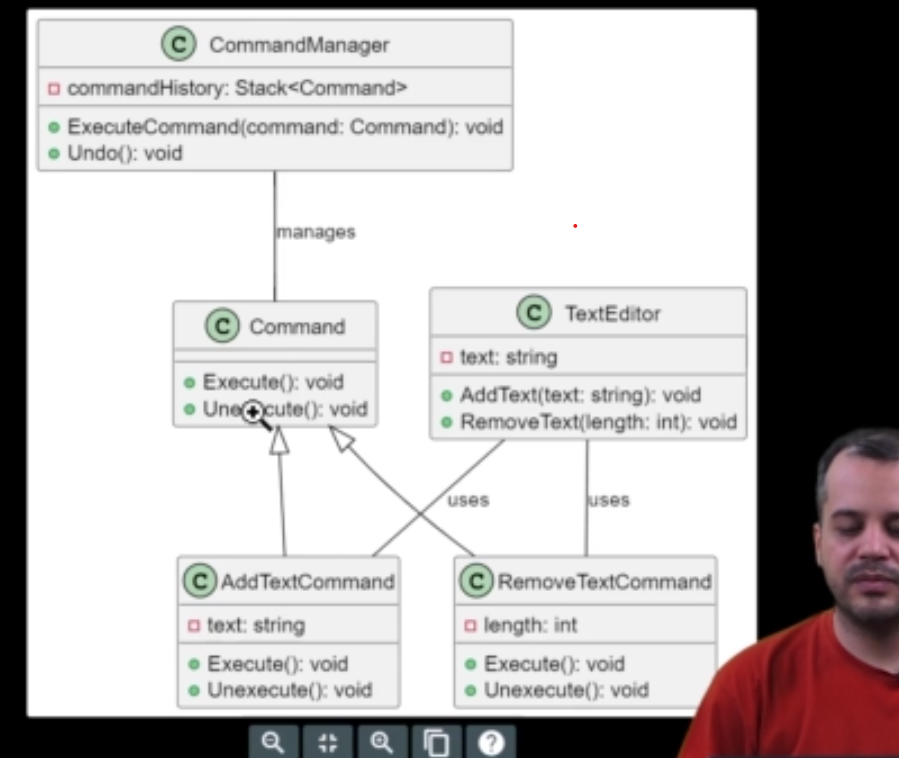
مثلا paste کردن و undo کردن.



When

زمانی که میخواهیم Parameterize کینم یک Action را که بتواند کار های مختلفی انجام دهد مستقل از درخواست اصلی و کار اصلی مدیریت State ها در سرویس است با استفاده از تاریخچه فعالیت.

مثلا سرویسی داریم برای ویرایش متن که کار های مثل افزودن و حذف متن را انجام می دهد حالا با استفاده از command می خواهیم بگوییم که کدام کار را انجام بدهد. و با استفاده از commandManager کار ها مدیریت می شوند.



Gpt

## Proxy (پروکسی) چیست؟

این الگو جزء **الگوهای ساختاری (Structural)** می‌باشد.  
پروکسی در واقع **یک واسط** بین **کلاینت** و **شیء واقعی** ایجاد می‌کند.

➥ اینکار اجازه می‌دهد بتوان بر **دسترسی**، **اعمال محدودیت**، **مدیریت منابع**، **اعمال امنیت** و … کنترل داشته باشیم **بدون این‌که در کد اصلی تغییری ایجاد کرده باشیم**.

## کاربردها:

➥ هنگامی که می‌خواهید:

* بر **دسترسی** به شئ واقعی نظارت داشته باشید.
* در صورت **نیاز ایجاد شئ واقعی را به تاخیر بیندازید** (Lazy Loading).
* **اعمال امنیت** بر روی استفاده از شئ انجام دهید.
* ثبت تاریخ استفاده، cache کردن نتیجه و …

## مثالی کاربردی:

➥ مثلاً ما اینترفیس ISubject داریم که یه عمل Request() دارد.  
➥ RealSubject اجرا کننده واقعی می‌باشد.  
➥ Proxy بر روی RealSubject می‌نشیند و می‌تواند چک کرده که آیا اجازه اجرا دارد یا نه.

## نتیجه:

➥ اینکار اجازه می‌دهد که بدون تغیر در RealSubject بتوانیم:

* محدودیت‌ها
* ثبت تاریخ استفاده
* کنترل امنیت
* Lazy Loading  
  را اعمال کرده و در نتیجه **نرم‌افزار منعطف‌تری** داشته باشیم.

Proxy Design Pattern in c#

# 1️ اینترفیس پایه

public interface ISubject  
{  
 void Request();  
}

# 2️ شئ واقعی (RealSubject)

public class RealSubject : ISubject  
{  
 public void Request()  
 {  
 Console.WriteLine("انجام درخواست در شئ واقعی.");  
 }  
}

# 3️ Proxy

public class Proxy : ISubject  
{  
 private RealSubject \_real = new RealSubject();  
  
 public void Request()  
 {  
 Console.WriteLine("بررسی امنیت قبل از اجرا.");  
 \_real.Request();  
 Console.WriteLine("اعمال بعد از اجرا.");  
 }  
}

# 4️ استفاده در برنامه

class Program  
{  
 static void Main(string[] args)  
 {  
 ISubject subject = new Proxy();  
  
 subject.Request();  
  
 // نتیجه:  
 // بررسی امنیت قبل از اجرا.  
 // انجام درخواست در شئ واقعی.  
 // اعمال بعد از اجرا.  
 }  
}

# 5️ نتیجه

اینکار اجازه می‌دهد بدون ایجاد تغیر در RealSubject بتوان نظارت، امنیت و محدودیت اعمال کرده و نرم‌افزار منعطف‌تری داشت.

Iterator

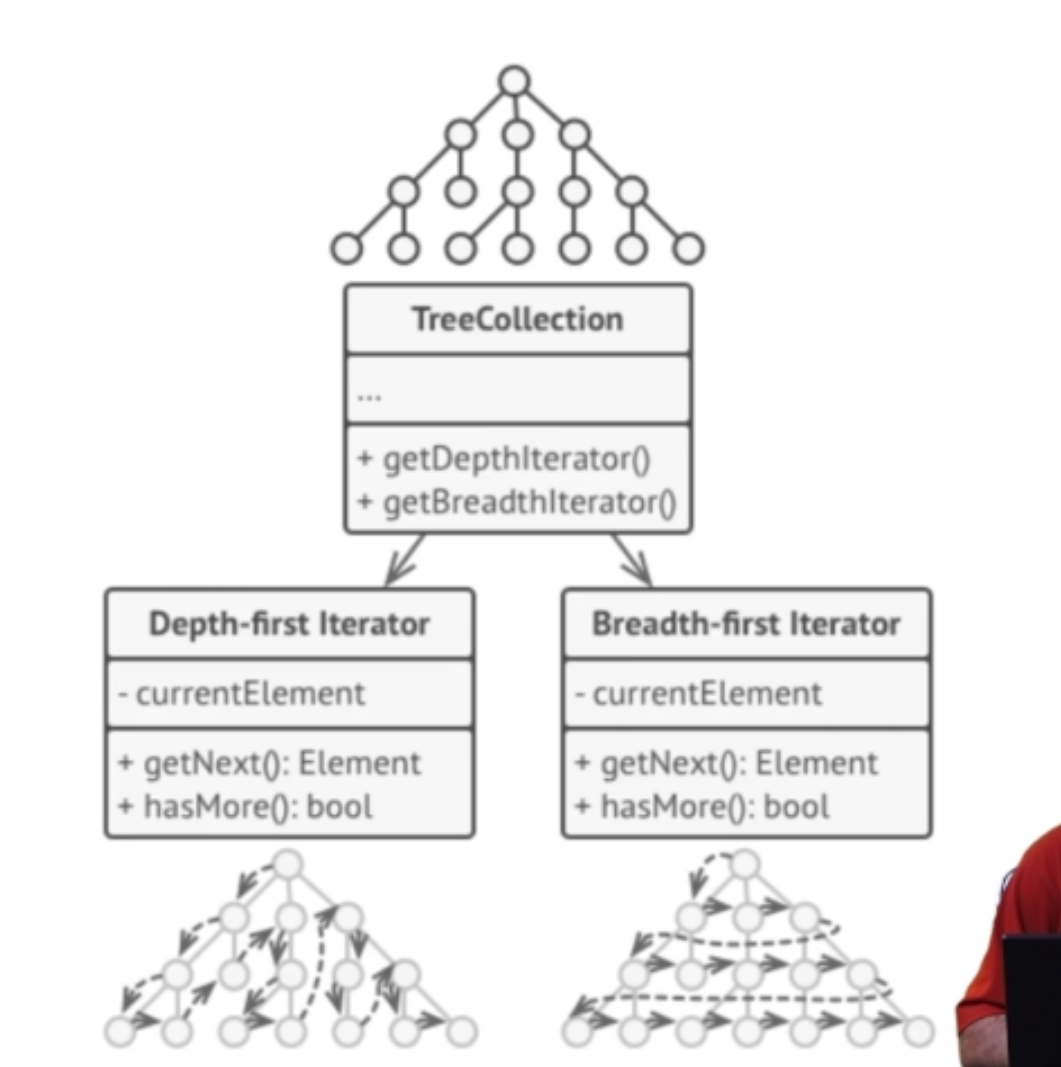
از نوع behavioral است. روشی برای رسیدن و پیمایش element های موجود در یک collection

این عملیات به ترتیب انجام شود در کالکشن.

در لیست و دیکشنری و ... با استفاده از یک designPattern و الگوریتم پیمایش ما الان مهم نیست.

Problem

زمانی که با collection پیچیده ای مواجه هستیم لازم است برای پیمایش از روش خودمان استفاده کنیم برای maintenance و استفاده.



Gpt

## Iterator (پیمایشگر) چیست؟

این الگو جزء **الگوهای رفتاری (Behavioral)** می‌باشد.  
Iterator اجازه می‌دهد بتوانیم **عناصر داخلی مجموعه** مثل لیست، درخت و... را **بدون اطلاع از ساختار داخلی آنها** پیمایش کرده و آنها را بخوانیم.

## کاربردها:

➥ هنگامی که می‌خواهید:

* **نحوه ذخیره داخلی** لیست، درخت، مجموعه و… از استفاده کننده **پنهان بماند**.
* بتوانید از طریق **یک رابط واحد** بر روی مجموعات مختلف **پیمایش کنید**.
* از **تکراری بودن کد در پیمایش** استفاده کرده و از این طریق کدتان خواناتر و تمیزتر بشه.

Iterator Design Pattern in C#

# 1️ اینترفیس IIterator

public interface IIterator<T>  
{  
 bool MoveNext();  
 T Current { get; }  
 void Reset();  
}

# 2️ پیاده‌سازی BookIterator

public class BookIterator : IIterator<string>  
{  
 private List<string> \_books;  
 private int \_current = -1;  
  
 public BookIterator(List<string> books)  
 {  
 \_books = books;  
 }  
 public bool MoveNext()  
 {  
 \_current++;  
 return \_current < \_books.Count;  
 }  
 public string Current => \_books[\_current];  
 public void Reset()  
 {  
 \_current = -1;  
 }  
}

# 3️ BookCollection

public class BookCollection  
{  
 private List<string> \_books = new List<string>();  
  
 public void AddBook(string book)  
 {  
 \_books.Add(book);  
 }  
 public IIterator<string> GetIterator()  
 {  
 return new BookIterator(\_books);  
 }  
}

# 4️ استفاده در برنامه

class Program  
{  
 static void Main(string[] args)  
 {  
 BookCollection collection = new BookCollection();  
  
 collection.AddBook("هری پاتر 1.");  
 collection.AddBook("هری پاتر 2.");  
 collection.AddBook("هری پاتر 3.");  
  
 IIterator<string> iterator = collection.GetIterator();  
  
 while (iterator.MoveNext())   
 {  
 Console.WriteLine(iterator.Current);  
 }  
 }  
}

5️ نتیجه

اینکار اجازه می‌دهد که استفاده کننده از طریق رابط واحد بر روی عناصر داخلی مجموعه پیمایش کرده و نیازی نداشته باشد از ساختار داخلی اطلاع داشته باشد.

Mediator

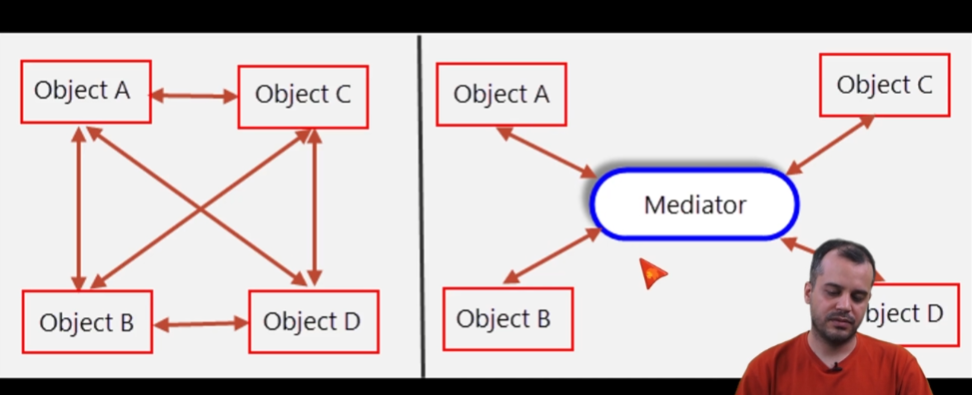
از نوع behavioral است و به معنای میانجی بوده یعنی بین 2 طرف قرار می گیرد برای ارتباط بین 2 object از طریق یک واسطه.

یادآور MediatR , RabbitMq , kafka

کاهش دهنده پیچیدگی ارتباط بین 2 شی و یا 2 سرویس یا سرویس های مختلف.

یک ارتباط برقرار کننده مرکزی است.

نکته : ارتباط بین mediator و شی 2 طرفه است.



Problem

در ترافیک های هوایی تمامی هواپیما ها نیاز به مسیر دهی برای اجرای عملیات فرود هستند که این مورد را با استفاده از برج مراقبت برای کنترل و هماهنگی فرود هواپیما ها چک می کنند که در عمل برج مراقبت نقش mediator را بازی می کند.

**مفاهیم**:

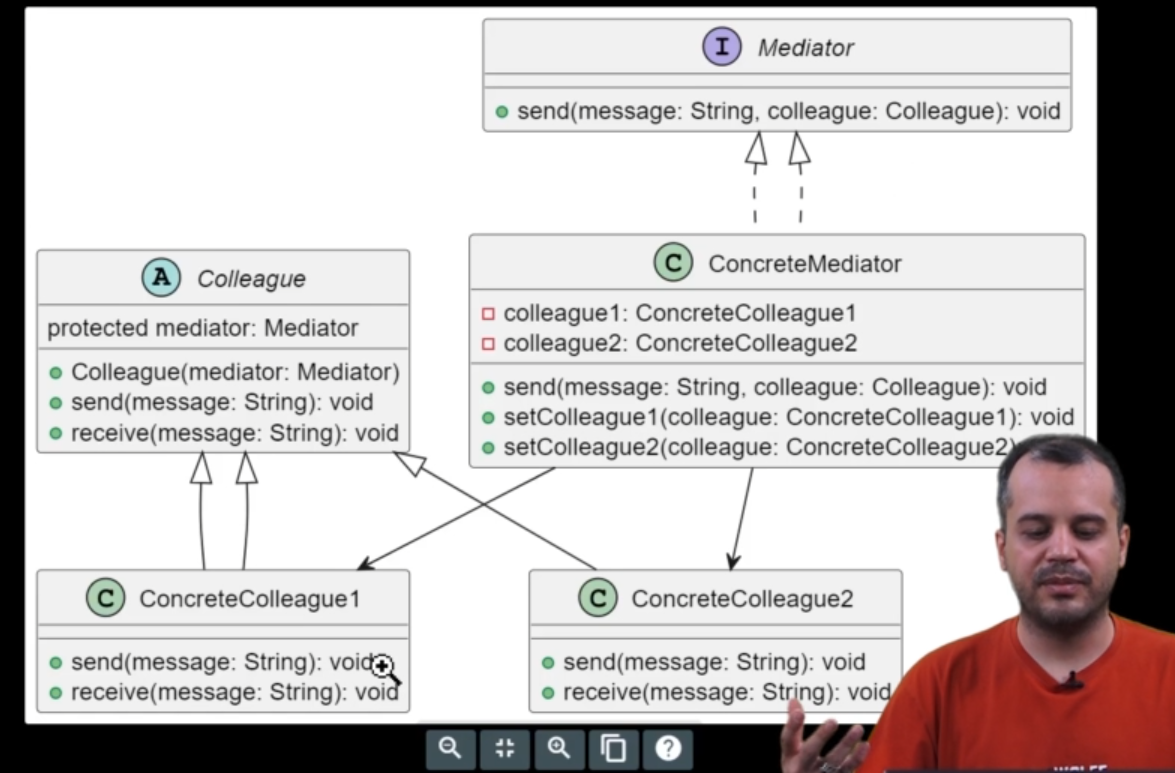
Mediator یک interface برای برقراری ارتباط است بین اشیا.

Concrete mediator در واقع کلاسی است که interface را پیاده سازی کرده است.

Colleagues در واقع موارد یا object های هستند که به mediator ما وصل هستند و ارتباط بین این ها از طریق mediator ایجاد می شود

**When to use**

زمانی که ارتباط برقرار کردن بین سرویس های ما پیچده است و dependency بین آن ها پیچیدگی ایجاد می کند



Gpt

الگوی طراحی **Mediator** یکی از الگوهای رفتاری (Behavioral Design Patterns) است که هدف آن کاهش وابستگی مستقیم بین اشیاء و تسهیل ارتباط آن‌ها از طریق یک شیء میانی (Mediator) است.

**تعریف الگوی Mediator:**

الگوی **Mediator** یک شیء را معرفی می‌کند که تعامل بین مجموعه‌ای از اشیاء را هماهنگ می‌کند. این الگو باعث می‌شود که اشیاء مستقیماً با یکدیگر صحبت نکنند، بلکه از طریق **یک واسطه (Mediator)** ارتباط برقرار کنند.

**کاربردهای Mediator:**

* هماهنگ کردن اجزای رابط کاربری (مانند فرم‌ها، دکمه‌ها، و لیست‌ها).
* طراحی سیستم‌هایی با تعامل پیچیده بین اشیاء زیاد.
* سیستم‌هایی با چرخه حیات طولانی اشیاء و نیاز به کاهش وابستگی بین آن‌ها.
* در **سیستم‌های چت**، **کنترل ترافیک هواپیما**، **موتورهای بازی‌سازی**، و **مدیریت رویدادها**.

**مزایا:**

* کاهش وابستگی مستقیم بین کلاس‌ها (loosely coupled).
* بهبود خوانایی و نگهداری کد.
* متمرکز کردن منطق تعامل در یک کلاس.

**معایب:**

* اگر Mediator خیلی بزرگ شود، می‌تواند به یک **God Object** تبدیل شود که تمام مسئولیت‌ها را برعهده دارد.

**1. رابط IMediator**

public interface IMediator  
{  
 void Notify(object sender, string ev);  
}

**2. کلاس پایه Component**

public class Component  
{  
 protected IMediator \_mediator;  
  
 public Component(IMediator mediator = null)  
 {  
 this.\_mediator = mediator;  
 }  
  
 public void SetMediator(IMediator mediator)  
 {  
 this.\_mediator = mediator;  
 }  
}

**3. اجزای فرم: TextBox و Button**

public class TextBox : Component  
{  
 public string Text { get; set; }  
  
 public void Input(string value)  
 {  
 Text = value;  
 \_mediator?.Notify(this, "TextChanged");  
 }  
}  
  
public class Button : Component  
{  
 public bool Enabled { get; set; }  
  
 public void Click()  
 {  
 if (Enabled)  
 Console.WriteLine("Button clicked!");  
 else  
 Console.WriteLine("Button is disabled.");  
 }  
}

**4. پیاده‌سازی Mediator: FormMediator**

public class FormMediator : IMediator  
{  
 public TextBox NameField { get; set; }  
 public Button SubmitButton { get; set; }  
  
 public FormMediator(TextBox nameField, Button submitButton)  
 {  
 NameField = nameField;  
 SubmitButton = submitButton;  
  
 NameField.SetMediator(this);  
 SubmitButton.SetMediator(this);  
 }  
  
 public void Notify(object sender, string ev)  
 {  
 if (sender == NameField && ev == "TextChanged")  
 {  
 SubmitButton.Enabled = !string.IsNullOrWhiteSpace(NameField.Text);  
 }  
 }  
}

**5. استفاده در برنامه اصلی**

class Program  
{  
 static void Main(string[] args)  
 {  
 var nameBox = new TextBox();  
 var submitBtn = new Button();  
  
 var formMediator = new FormMediator(nameBox, submitBtn);  
  
 nameBox.Input(""); // دکمه غیرفعال  
 submitBtn.Click();  
  
 nameBox.Input("Ali"); // دکمه فعال  
 submitBtn.Click();  
 }  
}

Memento

از نوع behavioral و بسیار مهم و کاربردی.

کارکرد برای نگه داری state یک object و جا به جایی بین state کاربرد دارد. شبیه command , state , observer است اما با تفاوت های.

کارش نگه داری state ها است و شما می توانید بین state ها جا به جا شوید.

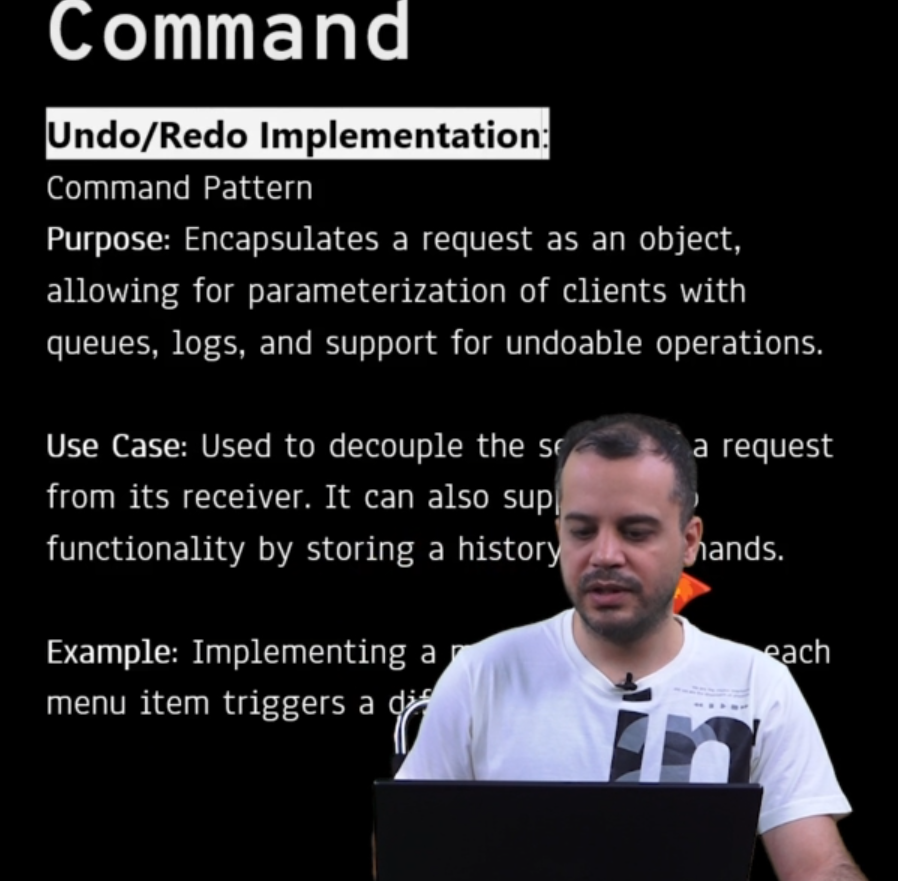
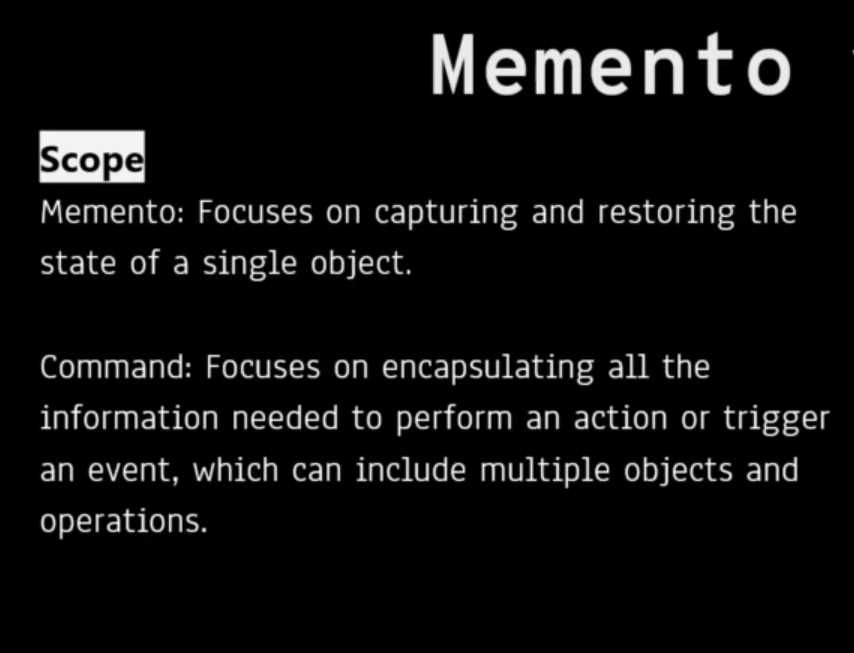
Originator در واقع چیزی است که می خواهیم state آن را نگه داری کنیم. یک سرویس یا کلاس است.

Memento مفهومی است که می خواهیم برای آن state نگه داریم.

Catetake کار مدیریت memento را انجام می دهد.

**Command vs memento**

****

****

**Memento کامند ارسال نمی کند بلکه state مربوط به object را ارسال می کند و capture و restore کردن state یک object تمرکز دارد.**

**Command تمرکزش بر روی ارسال دستور است.**

**Memento شبیه redux است که کتابخونه js است و هدف نگه داری State است.**

می توانیم memento و command را با هم دیگه استفاده کرد کار restore برای memento و undo کردن برای command مناسب است.

یکی برای نگه داری state سرویس دیگری برای نگه داری command history مناسب است.

Gpt

الگوی طراحی Memento یکی از الگوهای رفتاری (Behavioral Design Patterns) است که به شما اجازه می‌دهد وضعیت داخلی یک شیء را ذخیره و بعداً بازگردانی کنید بدون اینکه ساختار داخلی آن شیء افشا شود.

**تعریف الگوی Memento:**

الگوی **Memento** به شما این امکان را می‌دهد که وضعیت (state) یک شیء را در یک شیء جداگانه به نام **Memento** ذخیره کرده و بعداً آن را بازگردانید. این الگو معمولاً در کنار یک **Caretaker (نگهدارنده)** استفاده می‌شود که وظیفه نگهداری نسخه‌های ذخیره‌شده را بر عهده دارد.

**کاربردهای رایج:**

* سیستم Undo/Redo در نرم‌افزارها (مثلاً در ویرایشگر متن).
* ذخیره وضعیت قبل از انجام یک عملیات بحرانی.
* بازیابی تنظیمات یا وضعیت گذشته در اپلیکیشن‌ها.
* بازی‌ها برای ذخیره وضعیت بازیکن یا مرحله.

**مزایا:**

* بازیابی آسان وضعیت قبلی یک شیء.
* عدم نیاز به افشای ساختار داخلی شیء اصلی.
* جداسازی کامل مسئولیت‌ها بین Originator و Caretaker.

**معایب:**

* مصرف زیاد حافظه در صورت ذخیره‌سازی مکرر یا بزرگ بودن وضعیت.
* مدیریت پیچیده نسخه‌ها و بازگردانی در سیستم‌های بزرگ.

**اجزای الگو:**

1. **Originator**: شیء اصلی که وضعیت دارد.
2. **Memento**: وضعیت ذخیره‌شده را نگه می‌دارد.
3. **Caretaker**: مسئول ذخیره و بازیابی Mementoها.

## 1. کلاس Memento

public class TextEditorMemento  
{  
 public string Text { get; }  
  
 public TextEditorMemento(string text)  
 {  
 Text = text;  
 }  
}

## 2. کلاس Originator

public class TextEditor  
{  
 public string Content { get; set; }  
  
 public TextEditorMemento Save()  
 {  
 return new TextEditorMemento(Content);  
 }  
  
 public void Restore(TextEditorMemento memento)  
 {  
 Content = memento.Text;  
 }  
  
 public void Show()  
 {  
 Console.WriteLine("Current Content: " + Content);  
 }  
}

## 3. کلاس Caretaker

public class EditorHistory  
{  
 private Stack<TextEditorMemento> \_history = new Stack<TextEditorMemento>();  
  
 public void Backup(TextEditorMemento memento)  
 {  
 \_history.Push(memento);  
 }  
  
 public TextEditorMemento Undo()  
 {  
 return \_history.Count > 0 ? \_history.Pop() : null;  
 }  
}

## 4. استفاده در برنامه اصلی

class Program  
{  
 static void Main(string[] args)  
 {  
 var editor = new TextEditor();  
 var history = new EditorHistory();  
  
 editor.Content = "Hello";  
 history.Backup(editor.Save());  
  
 editor.Content = "Hello, World!";  
 history.Backup(editor.Save());  
  
 editor.Content = "Hello, World! Version 2";  
 editor.Show();  
  
 // Undo 1  
 editor.Restore(history.Undo());  
 editor.Show();  
  
 // Undo 2  
 editor.Restore(history.Undo());  
 editor.Show();  
 }  
}

**Observer**

از نوع Behavioral است و کاری که می کند این است که به کسانی که به آن گوش می کنند پیامی را اطلاع رسانی می کند.

از نوع one to many مثل دکل مخابراتی که به تعداد زیادی پیام را می رساند که آن ها subscribe کرده اند.

Problem

در مارکت اگر قیمت تغییر کرد به همه کسانی که subscribe کرده اند broadcast می کند و خبر می دهد.

**مفاهیم :**

Subject

ارسال کننده notification به کسانی که خواستار اطلاع از تغییرات هستند یعنی آن های که subscribe کرده اند.

Observer

تمامی اشیایی که درخواست اطلاع از تغییرات را داده اند.

مرور :

Chain of responsibility

مجموعه ای از خدمات که به صورت مرحله به مرحله اجرا می شود

Command

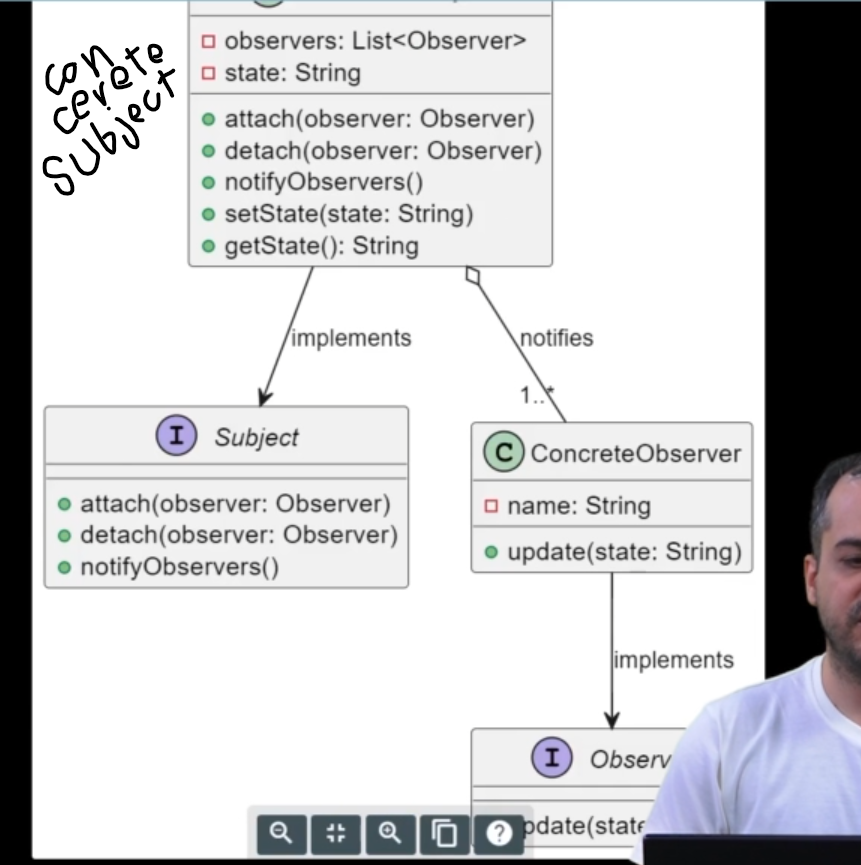
یک دستور ارسال می کند به receiver به صورت یک طرفه

Mediator

به صورت غیر مستقیم ارتباط بین 2 شی را برقرار می کند.

When

زمانی که تغییراتی به صورت داینامیک رخ می دهد و میخواهیم عده ای از آن مطلع شوند.



Gpt

الگوی طراحی **Observer** یکی از پرکاربردترین الگوهای رفتاری (Behavioral Design Patterns) است که به اشیاء اجازه می‌دهد از تغییرات در شیء دیگری آگاه شوند **بدون وابستگی شدید بین آن‌ها**.

**تعریف الگوی Observer:**

الگوی **Observer** یک مکانیزم اشتراک/اطلاع‌رسانی ارائه می‌دهد که در آن یک شیء (**Subject**) لیستی از اشیاء ناظر (**Observers**) را نگه می‌دارد و هر زمان که تغییراتی در وضعیت آن رخ داد، تمام ناظرها را باخبر می‌کند.

**کاربردهای Observer:**

* طراحی رابط کاربری (UI) با قابلیت واکنش به تغییر داده‌ها (مانند **MVVM** در WPF).
* پیاده‌سازی سیستم‌های **Event-driven**.
* **سیستم‌های هشدار** یا **مانیتورینگ**.
* **سرویس‌های نوتیفیکیشن** و **خبرخوان‌ها**.
* **پترن Publish-Subscribe** در معماری‌های مدرن.

**✅ مزایا:**

* حذف وابستگی مستقیم بین Subject و Observerها.
* امکان افزودن ناظرهای جدید بدون تغییر در Subject.
* تسهیل در پیاده‌سازی سیستم‌های پویا و واکنشی.

**❌ معایب:**

* مدیریت پیچیده در سیستم‌هایی با تعداد زیاد ناظر.
* احتمال ایجاد مشکلات عملکردی (Performance) در صورت نوتیفیکیشن‌های زیاد.

**ساختار کلاس‌ها:**

1. **ISubject** – رابطی برای اضافه/حذف ناظر و اطلاع‌رسانی.
2. **IObserver** – رابطی برای ناظرهایی که باید بروزرسانی شوند.
3. **ConcreteSubject** – نگهدارنده وضعیت و مسئول اطلاع‌رسانی.
4. **ConcreteObserver** – دریافت‌کننده بروزرسانی‌ها.

## 1. رابط ناظر (IObserver)

public interface IObserver  
{  
 void Update(float temperature);  
}

## 2. رابط موضوع (ISubject)

public interface ISubject  
{  
 void RegisterObserver(IObserver observer);  
 void RemoveObserver(IObserver observer);  
 void NotifyObservers();  
}

## 3. کلاس ConcreteSubject (WeatherStation)

public class WeatherStation : ISubject  
{  
 private List<IObserver> observers = new List<IObserver>();  
 private float temperature;  
  
 public void RegisterObserver(IObserver observer)  
 {  
 observers.Add(observer);  
 }

public void RemoveObserver(IObserver observer)  
 {  
 observers.Remove(observer);  
 }  
  
 public void SetTemperature(float temp)  
 {  
 Console.WriteLine($"New temperature set: {temp}°C");  
 temperature = temp;  
 NotifyObservers();  
 }  
  
 public void NotifyObservers()  
 {  
 foreach (var observer in observers)  
 {  
 observer.Update(temperature);  
 }  
 }  
}

## 4. کلاس ConcreteObserver (PhoneDisplay)

public class PhoneDisplay : IObserver  
{  
 private string \_name;  
  
 public PhoneDisplay(string name)  
 {  
 \_name = name;  
 }  
  
 public void Update(float temperature)  
 {  
 Console.WriteLine($"{\_name} Display: Temperature is now {temperature}°C");  
 }  
}

## 5. استفاده در برنامه اصلی

class Program  
{  
 static void Main(string[] args)  
 {  
 WeatherStation station = new WeatherStation();  
  
 PhoneDisplay display1 = new PhoneDisplay("Main");  
 PhoneDisplay display2 = new PhoneDisplay("Backup");  
  
 station.RegisterObserver(display1);  
 station.RegisterObserver(display2);  
  
 station.SetTemperature(25.5f);  
 station.SetTemperature(30.0f);  
  
 station.RemoveObserver(display2);  
 station.SetTemperature(22.3f);  
 }  
}

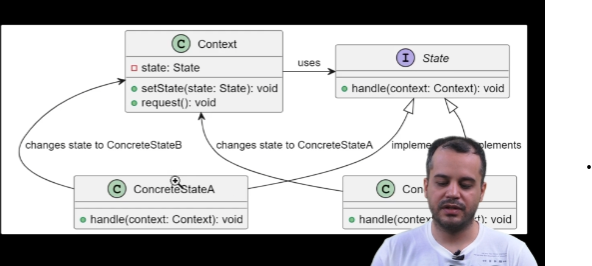
State

از نوع Behavioral است و کاربرد آن در اگر یک وضعیتی در یک object تغییر کرد رفتار آن یا behave ان تغییر کند با توجه به state

Problem

مثل دستگاه دریافت نوشابه در وضعیت های مختلف مثل پرداخت یا هیچ یا ناموفق رفتاری متناسب با وضعیت از خود بروز دهد.

مثال زمانی که گوشی lock یا unlock است در وضعیت های مختلف که مدیریت شده اند رفتار متناسب را از خود ارائه می دهد.



Gpt

الگوی طراحی **State** یکی از **الگوهای رفتاری (Behavioral Design Patterns)** است که به شیء اجازه می‌دهد **رفتار خود را بر اساس وضعیت داخلی‌اش تغییر دهد**، طوری که انگار شیء کلاس خود را تغییر داده است.

**تعریف الگوی State**

الگوی **State** به ما این امکان را می‌دهد که منطق رفتارهای یک شیء را بر اساس وضعیت‌های مختلف به **کلاس‌های مجزا** تقسیم کنیم و در زمان اجرا وضعیت را تغییر داده و رفتار جدید اعمال شود.

**کاربردها**

* ماشین‌های حالت (State Machines)، مانند دستگاه خودپرداز، چراغ راهنمایی.
* سیستم‌های گیمینگ برای تغییر حالت کاراکتر یا بازی.
* طراحی نرم‌افزارهایی با مراحل/فرآیندهای متوالی (مثل فرم‌های چندمرحله‌ای).
* سیستم‌هایی که باید در حالات مختلف رفتار متفاوتی داشته باشند.

**مزایا**

* جدا کردن کد وضعیت‌ها در کلاس‌های مختلف => افزایش خوانایی و نگهداری آسان‌تر.
* حذف شرط‌های if/else یا switch پیچیده.
* افزودن آسان حالت‌های جدید بدون تغییر در کلاس اصلی.

**معایب**

* افزایش تعداد کلاس‌ها.
* گاهی پیاده‌سازی ساده‌تری می‌تواند جایگزین شود اگر فقط ۲–۳ حالت وجود داشته باشد.

**اجزای الگو**

1. **Context**: شیء اصلی که وضعیت دارد و با State تعامل می‌کند.
2. **State Interface**: تعریف رفتارهای قابل تغییر.
3. **Concrete States**: پیاده‌سازی رفتارها در وضعیت‌های مختلف.

## 1. رابط وضعیت (IOrderState)

public interface IOrderState  
{  
 void Proceed(OrderContext context);  
 void Cancel(OrderContext context);  
}

## 2. کلاس NewOrderState

public class NewOrderState : IOrderState  
{  
 public void Proceed(OrderContext context)  
 {  
 Console.WriteLine("Order is now being shipped.");  
 context.SetState(new ShippedOrderState());  
 }  
  
 public void Cancel(OrderContext context)  
 {  
 Console.WriteLine("Order has been canceled.");  
 context.SetState(null);  
 }  
}

## 3. کلاس ShippedOrderState

public class ShippedOrderState : IOrderState  
{  
 public void Proceed(OrderContext context)  
 {  
 Console.WriteLine("Order has been delivered.");  
 context.SetState(new DeliveredOrderState());  
 }  
  
 public void Cancel(OrderContext context)  
 {  
 Console.WriteLine("Cannot cancel. Order is already shipped.");  
 }  
}

## 4. کلاس DeliveredOrderState

public class DeliveredOrderState : IOrderState  
{  
 public void Proceed(OrderContext context)  
 {  
 Console.WriteLine("Order already delivered. No further action.");  
 }  
  
 public void Cancel(OrderContext context)  
 {  
 Console.WriteLine("Cannot cancel. Order is already delivered.");  
 }  
}

## 5. کلاس Context (OrderContext)

public class OrderContext  
{  
 private IOrderState \_state;  
  
 public OrderContext()  
 {  
 \_state = new NewOrderState(); // حالت اولیه  
 }  
  
 public void SetState(IOrderState state)  
 {  
 \_state = state;  
 }  
  
 public void Proceed()  
 {  
 if (\_state != null)  
 \_state.Proceed(this);  
 else  
 Console.WriteLine("No further processing. Order is canceled or completed.");  
 }

public void Cancel()  
 {  
 if (\_state != null)  
 \_state.Cancel(this);  
 else  
 Console.WriteLine("Order already canceled or finalized.");  
 }  
}

## 6. استفاده در برنامه اصلی

class Program  
{  
 static void Main(string[] args)  
 {  
 var order = new OrderContext();  
  
 order.Proceed(); // Shipped  
 order.Proceed(); // Delivered  
 order.Cancel(); // Too late to cancel  
 }  
}

Strategy

از نوع Behavioral

یعنی یک کاری می خواهیم انجام بدهیم از طریق strategy های مختلف یا approach های مختلفی که داریم استفاده می کنیم مانند روش های مختلف برای رفتن به یک سفر با هواپیما یا قطار و ...

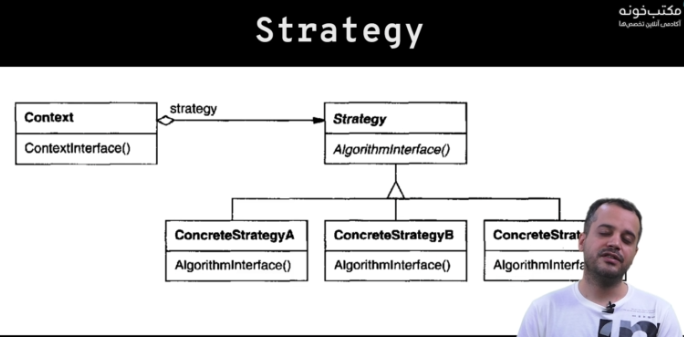
Problem

می خواهیم یک لیست را sort کنیم و برای انجام آن strategy های مختلفی که در زمان runtime می خواهیم استراتژی های مختلف را انتخاب کنیم.

مفاهیم

Context برای تنظیم کردن concrete strategy  
strategy interface یک interface مشترک برای تمام concrete strategy ها

Concrete strategy برای پیاده سازی strategy interface کاربرد دارد



علت جدا سازی بخش ها برای maintain بهتر است.

نکته : state می تواند یک extension از strategy محسوب شود.

در strategy می توان استراتژی های مختلف را به صورت مستقل یا independent تعریف کرد اما در State با یک دیگر در ارتباط هستند

نکته در state در هر موقع که اراده کردیم می توانیم State را تغییر دهیم.

When

Behavior configuration

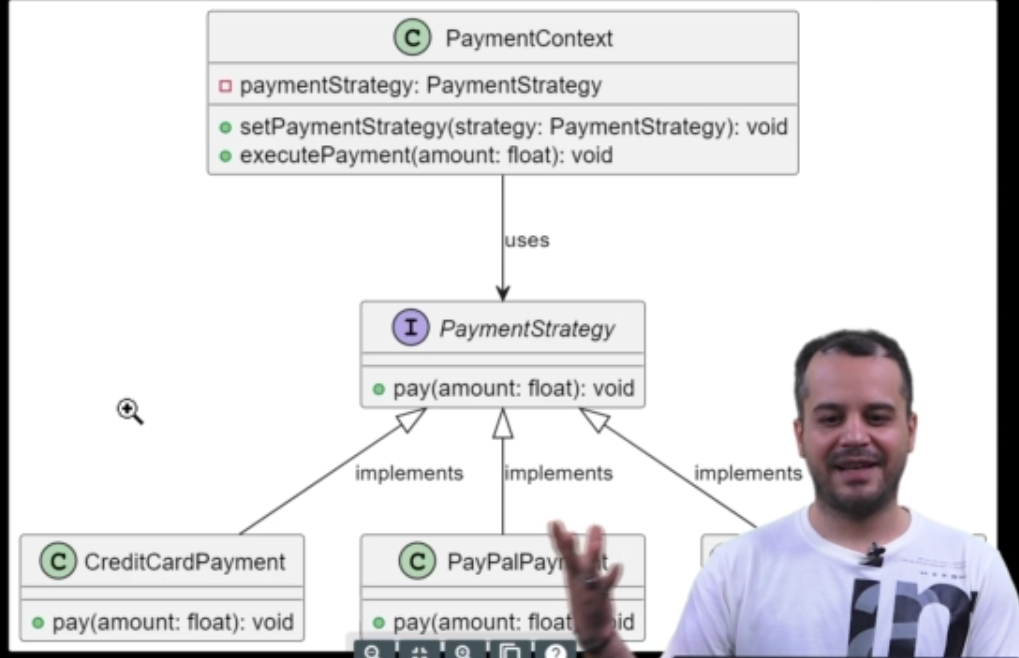
زمانی که استراتژی های ما دارای رفتار های خاصی باشند

Algorithm variants

زمانی که روش های مختلفی داریم برای انجام یک کار انواع sort

Data encapsulation

اجازه نمی دهیم که پیچیدگی های الگوریتم ما expose شود به بیرون درز پیدا کند



Gpt

### تعریف:

الگوی **Strategy** یکی از **الگوهای رفتاری (Behavioral Design Patterns)** است که به ما این امکان را می‌دهد که **الگوریتم‌های مختلف را جدا از کلاس اصلی پیاده‌سازی کنیم** و **در زمان اجرا، یکی از آن‌ها را انتخاب کنیم**.

یعنی بدون تغییر در کد اصلی، رفتار (الگوریتم) را تغییر می‌دهیم.

### کاربردهای Strategy Pattern

استفاده از این الگو زمانی مناسب است که:

* چندین **رفتار مشابه با جزئیات متفاوت** دارید.
* می‌خواهید **وابستگی به شرط‌ها (if/else/switch)** را کاهش دهید.
* الگوریتم‌ها و رفتارهایی دارید که ممکن است در آینده **تغییر کنند یا توسعه یابند**.
* نیاز دارید که در **زمان اجرا، بین چندین رفتار مختلف جابه‌جا شوید**.

## مزایای استفاده از Strategy Pattern:

* کد تمیزتر و قابل نگهداری‌تر.
* اجتناب از شرط‌های زیاد.
* انعطاف‌پذیری بالا در تغییر رفتار.
* استفاده مجدد از کدها و الگوریتم‌ها.

## مثال واقعی: محاسبه تخفیف برای کاربران مختلف

کاربر ممکن است یکی از سه نوع باشد:

* معمولی (بدون تخفیف)
* VIP (20٪ تخفیف)
* ویژه (50٪ تخفیف)

# 1️ اینترفیس IDiscountStrategy

public interface IDiscountStrategy  
{  
 decimal GetDiscount(decimal totalAmount);  
}

# 2️ پیاده‌سازی‌ استراتژی‌های مختلف

public class NoDiscount : IDiscountStrategy  
{  
 public decimal GetDiscount(decimal totalAmount)  
 {  
 return totalAmount;  
 }  
}  
  
public class VipDiscount : IDiscountStrategy  
{  
 public decimal GetDiscount(decimal totalAmount)  
 {  
 return totalAmount \* 0.8m;  
 }  
}  
  
public class SpecialDiscount : IDiscountStrategy  
{  
 public decimal GetDiscount(decimal totalAmount)  
 {  
 return totalAmount \* 0.5m;  
 }  
}

# 3️ کلاس Context (Order)

public class Order  
{  
 private IDiscountStrategy \_discountStrategy;  
  
 public Order(IDiscountStrategy discountStrategy)  
 {  
 \_discountStrategy = discountStrategy;  
 }  
  
 public decimal CalculateFinalAmount(decimal totalAmount)  
 {  
 return \_discountStrategy.GetDiscount(totalAmount);  
 }  
}

# 4️ استفاده در برنامه اصلی

class Program  
{  
 static void Main(string[] args)  
 {  
 decimal totalAmount = 1000;  
  
 Order vipOrder = new Order(new VipDiscount());  
 Console.WriteLine("مبلغ قابل پرداخت (VIP): " + vipOrder.CalculateFinalAmount(totalAmount));  
  
 Order specialOrder = new Order(new SpecialDiscount());  
 Console.WriteLine("مبلغ قابل پرداخت (ویژه): " + specialOrder.CalculateFinalAmount(totalAmount));  
  
 Order normalOrder = new Order(new NoDiscount());  
 Console.WriteLine("مبلغ قابل پرداخت (معمولی): " + normalOrder.CalculateFinalAmount(totalAmount));  
 }  
}

TemplateMethod

از نوع Behavioral است.

به طور مثال سند های مختلفی داریم که کار های مختلفی روی آن ها انجام می شود. مثل این که از اکسل خود یک pdf خروجی بگیریم که این step های مختلف را جداگانه پیاده سازی می کنیم.

سیستمی که نوع سند های مختلفی با workflow های مختلفی را پیاده سازی می کند که هر کدام روند خود را دارد و این ها را در یک subclass خود پیاده سازی می کنیم همچنین یک BaseClass نیز داریم.

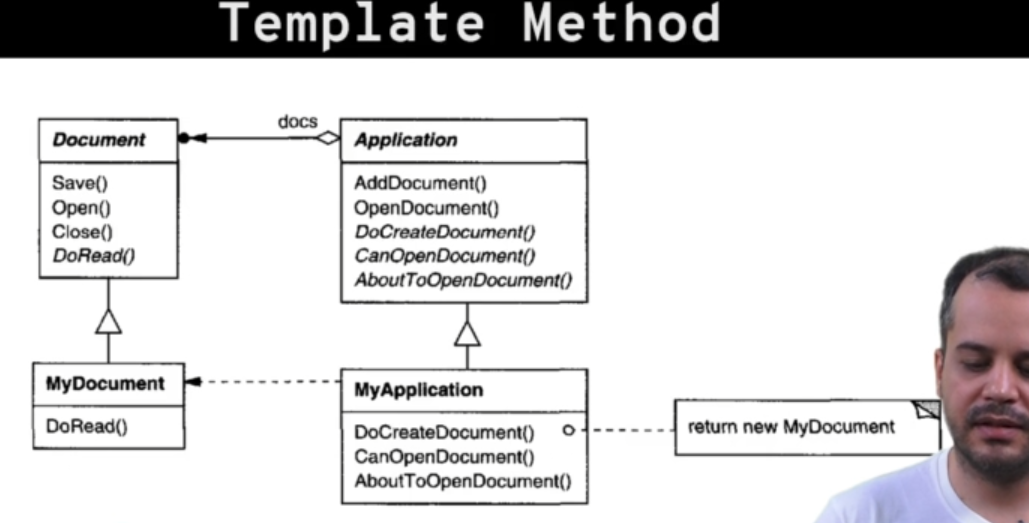
مفاهیم

BaseClass مثل DocumentProcessor

SubClass مثل LetterProcessor , ReportProcessor

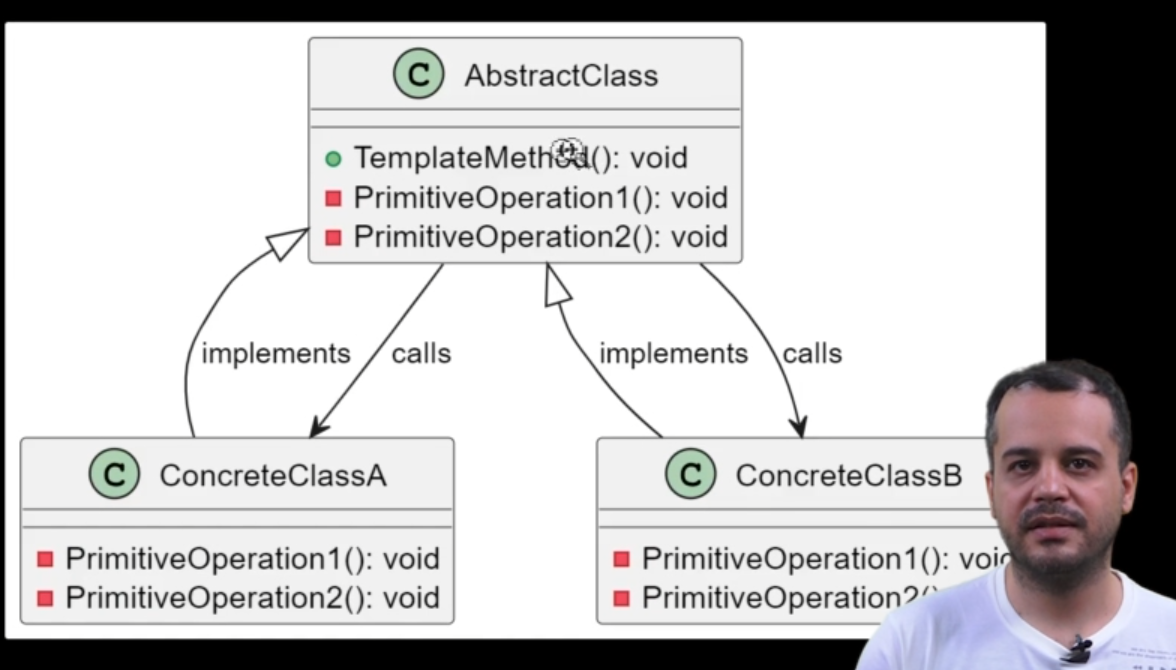
Problem

یک سری الگوریتم های داریم که یک ساختار مشترک دارد اما در Step های مختلفی اجرا می شوند و این ها را برای maintain کردن از subclass ها استفاده می کنیم.



When

برای پیاده سازی مدل های مختلفی از الگوریتم که به subclass ها سپرده می شوند تا آن ها را پیاده سازی کنند.



Gpt

الگوی طراحی **Template Method** یکی از الگوهای رفتاری (**Behavioral Design Patterns**) است که به ما اجازه می‌دهد **ساختار کلی یک الگوریتم را در یک کلاس پایه تعریف کنیم، اما مراحل خاص آن را به کلاس‌های فرزند واگذار کنیم.**

**تعریف ساده**

در این الگو، یک کلاس پایه وجود دارد که الگوریتم اصلی را با استفاده از چند مرحله مشخص تعریف می‌کند. هر مرحله (یا متد) می‌تواند در کلاس‌های فرزند بازنویسی (Override) شود تا رفتار خاص خودش را داشته باشد.

**کاربردها**

* زمانی که الگوریتمی ساختار ثابتی دارد اما در برخی بخش‌ها باید قابل تغییر باشد.
* جلوگیری از تکرار کد در کلاس‌های مختلف با ساختار مشابه.
* وقتی چندین کلاس فرزند داریم که الگوریتم مشابهی دارند با تفاوت‌های جزئی.

**مزایا**

* کد قابل استفاده مجدد و قابل توسعه می‌شود.
* پیاده‌سازی گام‌های الگوریتم در کلاس‌های مختلف ساده است.
* منطق الگوریتم به‌صورت متمرکز در کلاس پایه تعریف می‌شود.

**معایب**

* ایجاد وابستگی بین کلاس پایه و فرزند.
* ممکن است بیش از حد به ارث‌بری متکی شود (ترکیب شاید بهتر باشد در برخی موارد).

**ساختار کلی**

1. **Abstract Class (کلاس پایه):** الگوریتم را با مراحل مشخص پیاده‌سازی می‌کند.
2. **Concrete Class (کلاس فرزند):** برخی مراحل الگوریتم را به صورت خاص پیاده‌سازی می‌کند.

## مثال در C#

فرض کنیم می‌خواهیم الگوریتم آماده‌سازی نوشیدنی را تعریف کنیم: مراحل کلی مثل جوشاندن آب و ریختن در فنجان ثابت هستند، ولی مراحل خاص مانند افزودن مواد (چای یا قهوه) باید قابل تغییر باشد.

## 1. کلاس پایه CaffeineBeverage

public abstract class CaffeineBeverage  
{  
 public void PrepareRecipe()  
 {  
 BoilWater();  
 Brew();  
 PourInCup();  
 AddCondiments();  
 }  
  
 public void BoilWater()  
 {  
 Console.WriteLine("Boiling water");  
 }  
  
 public void PourInCup()  
 {  
 Console.WriteLine("Pouring into cup");  
 }  
  
 public abstract void Brew();  
 public abstract void AddCondiments();  
}

## 2. کلاس Tea

public class Tea : CaffeineBeverage  
{  
 public override void Brew()  
 {  
 Console.WriteLine("Steeping the tea");  
 }  
  
 public override void AddCondiments()  
 {  
 Console.WriteLine("Adding lemon");  
 }  
}

## 3. کلاس Coffee

public class Coffee : CaffeineBeverage  
{  
 public override void Brew()  
 {  
 Console.WriteLine("Dripping coffee through filter");  
 }  
  
 public override void AddCondiments()  
 {  
 Console.WriteLine("Adding sugar and milk");  
 }  
}

## 4. استفاده در برنامه اصلی

class Program  
{  
 static void Main(string[] args)  
 {  
 CaffeineBeverage tea = new Tea();  
 tea.PrepareRecipe();  
  
 Console.WriteLine();  
  
 CaffeineBeverage coffee = new Coffee();  
 coffee.PrepareRecipe();  
 }  
}

Visitor

از نوع behavioral است.

موجودیت های مختلفی داریم که توسط یک شخص مثل پزشک ویزیت می شوند.

می توانیم رفتار های مختلفی روی object انجام دهیم بدون تغییر آن.

Problem

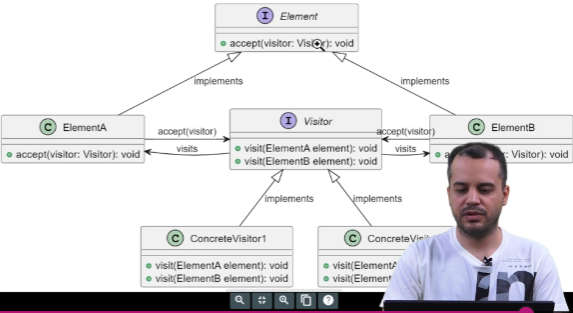
انواع شکل ها را داریم و آن ها را پیاده سازی کرده ایم حالا می خواهیم عملیات های مختلفی بر روی آن ها انجام دهیم ابتدا این شکل را visit می کنیم و سپس operation های مختلفی انجام می دهیم مثل aria calculator visitor و .... پس باید visitor توسط object ما accept شود.

یک interface داریم که ساختار را چیده و یک concrete داریم که پیاده سازی را انچام داده برای شکل های مختلف مثل دایره مثلث و ... که element ما هستند.

مزایا :

اجتناب از آلودگی کلاس object اصلی ما بلکه صرفا ویژگی ها را می گذاریم و operation ها توسط کلاس جداگانه visitor ما انجام می شود.

نکته » ساختار تمیزی می دهد اما scale کردن آن ساده نیست چون باید برای هر object ما یک متد و پیاده سازی در نظر بگیریم.



Gpt

الگوی طراحی **Visitor** یکی از **الگوهای رفتاری (Behavioral Design Patterns)** است که به ما اجازه می‌دهد **عملیات جدیدی روی مجموعه‌ای از اشیاء بدون تغییر ساختار کلاس‌های آن‌ها** اضافه کنیم.

**تعریف ساده**

الگوی Visitor اجازه می‌دهد که **یک عملیات مشخص** را به **کلاس‌های مختلفی که ساختار مشابه دارند ولی رفتار متفاوتی نشان می‌دهند** اضافه کنیم، بدون اینکه نیاز به تغییر کد آن کلاس‌ها باشد.

**کاربردها**

* وقتی می‌خواهید **رفتارهای مختلفی را به کلاس‌های موجود اضافه کنید**، اما نمی‌خواهید یا نمی‌توانید آن‌ها را تغییر دهید.
* مناسب برای کار با **ساختارهایی مانند درخت، گراف یا ساختارهای ترکیبی**.
* زمانی که چند عملیات روی یک ساختار داده‌ای باید پیاده‌سازی شود و نمی‌خواهیم عملیات در خود ساختار داده گسترش یابد.

**مزایا**

* جداسازی عملیات از ساختار داده.
* افزودن آسان عملیات جدید بدون دست زدن به کلاس‌های موجود.
* مناسب برای سیستم‌های پیچیده‌ای که نیاز به چندین نوع عملیات روی اشیاء مختلف دارند.

**معایب**

* سخت‌تر شدن نگهداری در صورت نیاز به افزودن کلاس‌های جدید به سلسله‌مراتب عناصر.
* نیاز به دانش بیشتر از ساختار سیستم برای پیاده‌سازی بازدیدکننده‌ها.

**اجزای الگو**

1. **Visitor Interface**: تعریف متدهای بازدید از هر نوع شیء.
2. **Concrete Visitor**: پیاده‌سازی عملیات واقعی برای هر نوع شیء.
3. **Element Interface**: رابطی که دارای متد Accept(Visitor) است.
4. **Concrete Elements**: اشیایی که از Element ارث می‌برند و متد Accept را پیاده‌سازی می‌کنند.

## 1. رابط Visitor

public interface IVisitor  
{  
 void Visit(Book book);  
 void Visit(CD cd);  
}

## 2. رابط Element

public interface IItem  
{  
 void Accept(IVisitor visitor);  
}

## 3. کلاس Book

public class Book : IItem  
{  
 public string Title { get; set; }  
 public double Price { get; set; }  
  
 public void Accept(IVisitor visitor)  
 {  
 visitor.Visit(this);  
 }  
}

## 4. کلاس CD

public class CD : IItem  
{  
 public string Artist { get; set; }  
 public double Price { get; set; }  
  
 public void Accept(IVisitor visitor)  
 {  
 visitor.Visit(this);  
 }  
}

## 5. کلاس DiscountVisitor

public class DiscountVisitor : IVisitor  
{  
 public void Visit(Book book)  
 {  
 Console.WriteLine($"Book: {book.Title}, Price after 10% discount: {book.Price \* 0.9}");  
 }  
  
 public void Visit(CD cd)  
 {  
 Console.WriteLine($"CD: {cd.Artist}, Price after 5% discount: {cd.Price \* 0.95}");  
 }  
}

## 6. استفاده در برنامه اصلی

class Program  
{  
 static void Main(string[] args)  
 {  
 List<IItem> items = new List<IItem>  
 {  
 new Book { Title = "Design Patterns", Price = 100 },  
 new CD { Artist = "Beethoven", Price = 50 }  
 };  
  
 IVisitor discountVisitor = new DiscountVisitor();  
  
 foreach (var item in items)  
 {  
 item.Accept(discountVisitor);  
 }  
 }  
}

Chain of responsibility

الگوی طراحی **Chain of Responsibility** (زنجیره مسئولیت) یکی از **الگوهای رفتاری** (Behavioral Design Patterns) است. این الگو اجازه می‌دهد تا **درخواست‌ها از میان زنجیره‌ای از گیرنده‌ها عبور کنند** تا یکی از آن‌ها آن را پردازش کند. در این الگو، فرستنده‌ی درخواست نیازی به دانستن این ندارد که کدام شیء آن را پردازش خواهد کرد.

**تعریف ساده**

در این الگو، چند شیء در یک زنجیره (Chain) قرار می‌گیرند. هر شیء، **یا درخواست را پردازش می‌کند** یا آن را به **شیء بعدی در زنجیره منتقل می‌کند**.

**کاربردهای معمول**

* فیلتر کردن یا مدیریت درخواست‌ها مانند **سیستم‌های لاگ‌گیری، احراز هویت، بررسی مجوز**.
* در **فرم‌های بررسی خطا (Validation Forms)** که باید چندین بررسی پشت سر هم انجام شود.
* سیستم‌های پشتیبانی (Help Desk) که هر سطح مسئول پاسخگویی به نوع خاصی از مشکل است.

**مزایا**

* جداسازی فرستنده و گیرنده.
* اضافه کردن/حذف گیرنده‌ها بدون تغییر کد دیگر اجزا.
* پشتیبانی از **اصل Open/Closed**.

**معایب**

* اشکال‌زدایی سخت‌تر به دلیل عبور درخواست از چند شیء.
* ممکن است به درخواست پاسخی داده نشود اگر هیچ گیرنده‌ای آن را قبول نکند.

ساختار کلاس

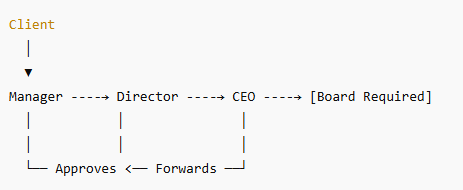


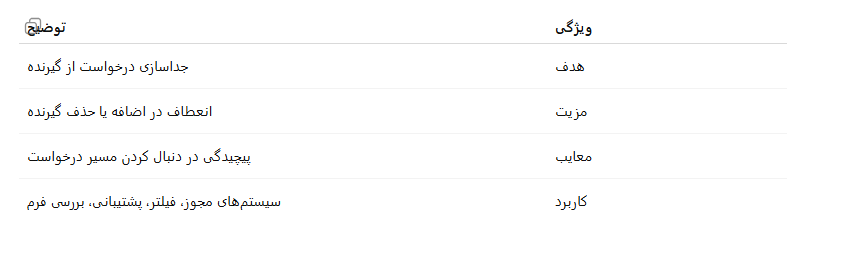
اجزا

1. **Handler رابط یا کلاس پایه:** تعریف متد Handle() و نگهدارنده‌ی شیء بعدی در زنجیره.
2. **ConcreteHandlers** کلاس‌هایی که درخواست را پردازش می‌کنند یا به بعدی می‌دهند.
3. **Client** ایجاد زنجیره و ارسال درخواست.

**مثال واقعی در C#**

فرض کنید یک سیستم بررسی سطح دسترسی داریم (Manager، Director، CEO) که بسته به میزان هزینه، درخواست را بررسی می‌کند.





## 1. کلاس پایه Approver

public abstract class Approver  
{  
 protected Approver NextApprover;  
  
 public void SetNext(Approver next)  
 {  
 this.NextApprover = next;  
 }  
  
 public abstract void HandleRequest(int amount);  
}

## 2. کلاس Manager

public class Manager : Approver  
{  
 public override void HandleRequest(int amount)  
 {  
 if (amount <= 1000)  
 Console.WriteLine("Manager approved request of $" + amount);  
 else if (NextApprover != null)  
 NextApprover.HandleRequest(amount);  
 }  
}

## 3. کلاس Director

public class Director : Approver  
{  
 public override void HandleRequest(int amount)  
 {  
 if (amount <= 5000)  
 Console.WriteLine("Director approved request of $" + amount);  
 else if (NextApprover != null)  
 NextApprover.HandleRequest(amount);  
 }  
}

## 4. کلاس CEO

public class CEO : Approver  
{  
 public override void HandleRequest(int amount)  
 {  
 if (amount <= 20000)  
 Console.WriteLine("CEO approved request of $" + amount);  
 else  
 Console.WriteLine("Request of $" + amount + " needs a board meeting!");  
 }  
}

## 5. استفاده در Main

class Program  
{  
 static void Main(string[] args)  
 {  
 Approver manager = new Manager();  
 Approver director = new Director();  
 Approver ceo = new CEO();  
  
 manager.SetNext(director);  
 director.SetNext(ceo);  
  
 int[] requests = { 500, 3000, 15000, 25000 };  
  
 foreach (var request in requests)  
 {  
 manager.HandleRequest(request);  
 }  
 }  
}

## 6. خروجی برنامه

Manager approved request of $500  
Director approved request of $3000  
CEO approved request of $15000  
Request of $25000 needs a board meeting!