**DESIGN PRINCIPLES**

Design principles in C# refer to guidelines and best practices that developers follow to create well-structured, maintainable, and efficient code. Adhering to these principles helps produce software that is scalable, robust, and easy to understand. Here are some key design principles in C#:

**1.SOLID Principles (Object-Oriented Design):**

* + **Single Responsibility Principle (SRP):** A class should have only one reason to change, meaning it should only have one responsibility.
  + **Open-Closed Principle (OCP):** Software entities should be open for extension but closed for modification. You can add new functionality without altering existing code.
  + **Liskov Substitution Principle (LSP):** Subtypes must be substitutable for their base types without altering the correctness of the program.
  + **Interface Segregation Principle (ISP):** Clients should not be forced to implement interfaces it does not use. Instead of having large, monolithic interfaces, it’s better to have smaller, focused ones.
  + **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details; details should depend on abstarctions.

1. **DRY (Don't Repeat Yourself):**
   * Avoid duplicating code by creating reusable abstractions. This principle promotes code readability, maintainability, and reduces the risk of errors.
2. **KISS (Keep It Simple, Stupid):**
   * Simplicity is key. Keep designs and systems as simple as possible, avoiding unnecessary complexity. Simple solutions are often easier to understand, maintain, and troubleshoot.
3. **YAGNI (You Aren't Gonna Need It):**
4. Only implement features that are needed for the current requirements. Avoid adding functionality based on speculation about future needs.

**5.Separation of Concerns:**

* + Divide your system into distinct sections, each addressing a separate concern or aspect of functionality. This makes the system more modular, easier to understand, and maintain.

**6.Scalability:**

* Design systems to handle growth and increased demand. Scalable systems can adapt to larger workloads without significant redesign.

7. **Testing:**

Design with testability in mind. Write unit tests to ensure that individual components of your code work as expected. This contributes to the reliability and maintainability of the codebase.

**8. High Cohesion, Low Coupling:**

* **High Cohesion:** Elements within a module (class, component) should be closely related and work together for a common goal.
* **Low Coupling:** Minimize the dependencies between modules. Changes in one module should have minimal impact on other modules.

These design principles provide a framework for creating robust, maintainable, and adaptable systems. Depending on the context and specific goals, these principles can be adapted and combined to guide the design process effectively.

Top of Form

**DESIGN PATTERNS**

The Gang of Four (GoF) design patterns mentioned in their book "Design Patterns: Elements of Reusable Object-Oriented Software." These 23 patterns are categorized into three groups: Creational, Structural, and Behavioral.

**Creational Patterns:**

1. **Singleton Pattern:**
   * Ensures a class has only one instance and provides a global point of access to it.
   * Ex: public sealed class Singleton
   * {
   * // The single instance of the class
   * private static Singleton \_instance;
   * // Private constructor to prevent creating multiple instances
   * private Singleton()
   * {
   * // Initialization code, if needed
   * }
   * // Public method to get the single instance
   * public static Singleton Instance
   * {
   * get
   * {
   * if (\_instance == null)
   * {
   * // If no instance exists, create one
   * \_instance = new Singleton();
   * }
   * return \_instance;
   * }
   * }
   * // Other methods and properties of the Singleton class can be added here
   * public void SomeMethod()
   * {
   * Console.WriteLine("Singleton method called");
   * }
   * }
   * class Program
   * {
   * static void Main()
   * {
   * // Access the singleton instance
   * Singleton singleton1 = Singleton.Instance;
   * singleton1.SomeMethod();
   * // Attempting to create another instance will return the same instance
   * Singleton singleton2 = Singleton.Instance;
   * singleton2.SomeMethod();
   * // Both instances point to the same object
   * Console.WriteLine("Is singleton1 the same instance as singleton2? " + (singleton1 == singleton2));
   * /\*
   * Output:
   * Singleton method called
   * Singleton method called
   * Is singleton1 the same instance as singleton2? True
   * \*/
   * }
   * }
2. **Factory Method Pattern:**
   * Defines an interface for creating an object but lets subclasses alter the type of objects that will be created.
   * Ex: // Product interface
   * public interface IProduct
   * {
   * void SomeOperation();
   * }
   * // Concrete product
   * public class ConcreteProductA : IProduct
   * {
   * public void SomeOperation()
   * {
   * Console.WriteLine("ConcreteProductA operation");
   * }
   * }
   * // Concrete product
   * public class ConcreteProductB : IProduct
   * {
   * public void SomeOperation()
   * {
   * Console.WriteLine("ConcreteProductB operation");
   * }
   * }
   * // Creator interface
   * public interface ICreator
   * {
   * IProduct FactoryMethod();
   * }
   * // Concrete creator
   * public class ConcreteCreatorA : ICreator
   * {
   * public IProduct FactoryMethod()
   * {
   * return new ConcreteProductA();
   * }
   * }
   * // Concrete creator
   * public class ConcreteCreatorB : ICreator
   * {
   * public IProduct FactoryMethod()
   * {
   * return new ConcreteProductB();
   * }
   * }
   * // Client class
   * public class Client
   * {
   * public void Main()
   * {
   * // Client uses a creator to get a product
   * ICreator creator = new ConcreteCreatorA();
   * IProduct product = creator.FactoryMethod();
   * product.SomeOperation();
   * }
   * }
3. **Abstract Factory Pattern:**

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Ex: using System;

// Abstract Product A

interface IChair

{

void SitOn();

}

// Concrete Product A1

class ModernChair : IChair

{

public void SitOn()

{

Console.WriteLine("Sitting on a modern chair");

}

}

// Concrete Product A2

class VictorianChair : IChair

{

public void SitOn()

{

Console.WriteLine("Sitting on a Victorian chair");

}

}

// Abstract Product B

interface ITable

{

void PutOn();

}

// Concrete Product B1

class ModernTable : ITable

{

public void PutOn()

{

Console.WriteLine("Putting something on a modern table");

}

}

// Concrete Product B2

class VictorianTable : ITable

{

public void PutOn()

{

Console.WriteLine("Putting something on a Victorian table");

}

}

// Abstract Factory

interface IFurnitureFactory

{

IChair CreateChair();

ITable CreateTable();

}

// Concrete Factory 1

class ModernFurnitureFactory : IFurnitureFactory

{

public IChair CreateChair()

{

return new ModernChair();

}

public ITable CreateTable()

{

return new ModernTable();

}

}

// Concrete Factory 2

class VictorianFurnitureFactory : IFurnitureFactory

{

public IChair CreateChair()

{

return new VictorianChair();

}

public ITable CreateTable()

{

return new VictorianTable();

}

}

// Client code

class Client

{

static void Main()

{

// Client chooses a specific furniture factory (Modern or Victorian)

IFurnitureFactory modernFactory = new ModernFurnitureFactory();

IFurnitureFactory victorianFactory = new VictorianFurnitureFactory();

// Client uses the chosen factory to create chair and table

IChair modernChair = modernFactory.CreateChair();

ITable modernTable = modernFactory.CreateTable();

IChair victorianChair = victorianFactory.CreateChair();

ITable victorianTable = victorianFactory.CreateTable();

// Client uses the created products

modernChair.SitOn();

modernTable.PutOn();

victorianChair.SitOn();

victorianTable.PutOn();

/\*

Output:

Sitting on a modern chair

Putting something on a modern table

Sitting on a Victorian chair

Putting something on a Victorian table

\*/

}

}

1. **Builder Pattern:**
   * Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
   * Ex: using System;
   * // Product
   * class Pizza
   * {
   * public string Dough { get; set; }
   * public string Sauce { get; set; }
   * public string Topping { get; set; }
   * public void Display()
   * {
   * Console.WriteLine($"Pizza with {Dough} dough, {Sauce} sauce, and {Topping} topping");
   * }
   * }
   * // Builder interface
   * interface IPizzaBuilder
   * {
   * void BuildDough();
   * void BuildSauce();
   * void BuildTopping();
   * Pizza GetPizza();
   * }
   * // Concrete Builder 1
   * class MargheritaPizzaBuilder : IPizzaBuilder
   * {
   * private Pizza \_pizza = new Pizza();
   * public void BuildDough()
   * {
   * \_pizza.Dough = "Thin crust";
   * }
   * public void BuildSauce()
   * {
   * \_pizza.Sauce = "Tomato sauce";
   * }
   * public void BuildTopping()
   * {
   * \_pizza.Topping = "Mozzarella cheese";
   * }
   * public Pizza GetPizza()
   * {
   * return \_pizza;
   * }
   * }
   * // Concrete Builder 2
   * class PepperoniPizzaBuilder : IPizzaBuilder
   * {
   * private Pizza \_pizza = new Pizza();
   * public void BuildDough()
   * {
   * \_pizza.Dough = "Thick crust";
   * }
   * public void BuildSauce()
   * {
   * \_pizza.Sauce = "Spicy tomato sauce";
   * }
   * public void BuildTopping()
   * {
   * \_pizza.Topping = "Pepperoni and olives";
   * }
   * public Pizza GetPizza()
   * {
   * return \_pizza;
   * }
   * }
   * // Director
   * class PizzaDirector
   * {
   * private IPizzaBuilder \_builder;
   * public PizzaDirector(IPizzaBuilder builder)
   * {
   * \_builder = builder;
   * }
   * public void ConstructPizza()
   * {
   * \_builder.BuildDough();
   * \_builder.BuildSauce();
   * \_builder.BuildTopping();
   * }
   * }
   * // Client code
   * class Client
   * {
   * static void Main()
   * {
   * // Create a Margherita pizza
   * IPizzaBuilder margheritaBuilder = new MargheritaPizzaBuilder();
   * PizzaDirector margheritaDirector = new PizzaDirector(margheritaBuilder);
   * margheritaDirector.ConstructPizza();
   * Pizza margheritaPizza = margheritaBuilder.GetPizza();
   * margheritaPizza.Display();
   * Console.WriteLine();
   * // Create a Pepperoni pizza
   * IPizzaBuilder pepperoniBuilder = new PepperoniPizzaBuilder();
   * PizzaDirector pepperoniDirector = new PizzaDirector(pepperoniBuilder);
   * pepperoniDirector.ConstructPizza();
   * Pizza pepperoniPizza = pepperoniBuilder.GetPizza();
   * pepperoniPizza.Display();
   * /\*
   * Output:
   * Pizza with Thin crust dough, Tomato sauce sauce, and Mozzarella cheese topping
   * Pizza with Thick crust dough, Spicy tomato sauce sauce, and Pepperoni and olives topping
   * \*/
   * }
   * }

**5.Prototype Pattern:**

* + Creates new objects by copying an existing object, known as the prototype.
  + Ex : using System;
  + // Prototype interface
  + interface ICloneablePrototype
  + {
  + ICloneablePrototype Clone();
  + }
  + // Concrete Prototype 1
  + class ConcretePrototype1 : ICloneablePrototype
  + {
  + public int Number { get; set; }
  + public ICloneablePrototype Clone()
  + {
  + return new ConcretePrototype1 { Number = this.Number };
  + }
  + }
  + // Concrete Prototype 2
  + class ConcretePrototype2 : ICloneablePrototype
  + {
  + public string Text { get; set; }
  + public ICloneablePrototype Clone()
  + {
  + return new ConcretePrototype2 { Text = this.Text };
  + }
  + }
  + // Client code
  + class Client
  + {
  + static void Main()
  + {
  + // Create prototypes
  + ICloneablePrototype prototype1 = new ConcretePrototype1 { Number = 42 };
  + ICloneablePrototype prototype2 = new ConcretePrototype2 { Text = "Hello" };
  + // Clone prototypes
  + ICloneablePrototype clonedPrototype1 = prototype1.Clone();
  + ICloneablePrototype clonedPrototype2 = prototype2.Clone();
  + // Display original and cloned objects
  + Console.WriteLine("Original Prototype 1: " + prototype1.GetHashCode() + " - Number: " + ((ConcretePrototype1)prototype1).Number);
  + Console.WriteLine("Cloned Prototype 1: " + clonedPrototype1.GetHashCode() + " - Number: " + ((ConcretePrototype1)clonedPrototype1).Number);
  + Console.WriteLine();
  + Console.WriteLine("Original Prototype 2: " + prototype2.GetHashCode() + " - Text: " + ((ConcretePrototype2)prototype2).Text);
  + Console.WriteLine("Cloned Prototype 2: " + clonedPrototype2.GetHashCode() + " - Text: " + ((ConcretePrototype2)clonedPrototype2).Text);
  + /\*
  + Output:
  + Original Prototype 1: 1910487 - Number: 42
  + Cloned Prototype 1: 31829242 - Number: 42
  + Original Prototype 2: 1453266 - Text: Hello
  + Cloned Prototype 2: 21590442 - Text: Hello
  + \*/
  + }
  + }

**Structural Patterns:**

1. **Adapter Pattern:**
   * Allows the interface of an existing class to be used as another interface.
   * Ex: using System;
   * // Target interface expected by the client
   * interface ITarget
   * {
   * void Request();
   * }
   * // Adaptee (the class with an incompatible interface)
   * class Adaptee
   * {
   * public void SpecificRequest()
   * {
   * Console.WriteLine("Adaptee's specific request");
   * }
   * }
   * // Adapter class that adapts Adaptee to ITarget
   * class Adapter : ITarget
   * {
   * private Adaptee \_adaptee;
   * public Adapter(Adaptee adaptee)
   * {
   * \_adaptee = adaptee;
   * }
   * public void Request()
   * {
   * // Use Adaptee's specific method in a way that matches the ITarget interface
   * \_adaptee.SpecificRequest();
   * }
   * }
   * // Client code that uses the ITarget interface
   * class Client
   * {
   * public void MakeRequest(ITarget target)
   * {
   * target.Request();
   * }
   * }
   * class Program
   * {
   * static void Main()
   * {
   * // Create an instance of Adaptee
   * Adaptee adaptee = new Adaptee();
   * // Create an Adapter, passing the Adaptee instance
   * ITarget adapter = new Adapter(adaptee);
   * // Client code can now use the ITarget interface
   * Client client = new Client();
   * client.MakeRequest(adapter);
   * /\*
   * Output:
   * Adaptee's specific request
   * \*/
   * }
   * }

1. **Bridge Pattern:**
   * Separates abstraction from implementation so that both can vary independently.
   * Ex: using System;
   * // Abstraction
   * abstract class Shape
   * {
   * protected IColor color;
   * public Shape(IColor color)
   * {
   * this.color = color;
   * }
   * public abstract void Draw();
   * }
   * // Concrete Abstraction: Circle
   * class Circle : Shape
   * {
   * private int radius;
   * public Circle(IColor color, int radius) : base(color)
   * {
   * this.radius = radius;
   * }
   * public override void Draw()
   * {
   * Console.WriteLine($"Drawing a Circle with radius {radius} and color {color.FillColor()}");
   * }
   * }
   * // Concrete Abstraction: Square
   * class Square : Shape
   * {
   * private int sideLength;
   * public Square(IColor color, int sideLength) : base(color)
   * {
   * this.sideLength = sideLength;
   * }
   * public override void Draw()
   * {
   * Console.WriteLine($"Drawing a Square with side length {sideLength} and color {color.FillColor()}");
   * }
   * }
   * // Implementation
   * interface IColor
   * {
   * string FillColor();
   * }
   * // Concrete Implementation: RedColor
   * class RedColor : IColor
   * {
   * public string FillColor()
   * {
   * return "Red";
   * }
   * }
   * // Concrete Implementation: BlueColor
   * class BlueColor : IColor
   * {
   * public string FillColor()
   * {
   * return "Blue";
   * }
   * }
   * // Client code
   * class Client
   * {
   * static void Main()
   * {
   * // Create instances of concrete implementations (RedColor and BlueColor)
   * IColor redColor = new RedColor();
   * IColor blueColor = new BlueColor();
   * // Create instances of concrete abstractions (Circle and Square) with different colors
   * Shape redCircle = new Circle(redColor, 5);
   * Shape blueSquare = new Square(blueColor, 4);
   * // Draw the shapes
   * redCircle.Draw();
   * blueSquare.Draw();
   * /\*
   * Output:
   * Drawing a Circle with radius 5 and color Red
   * Drawing a Square with side length 4 and color Blue
   * \*/
   * }
   * }
2. **Composite Pattern:**
   * Composes objects into tree structures to represent part-whole hierarchies.
   * Ex: using System;
   * using System.Collections.Generic;
   * // Component interface
   * interface IComponent
   * {
   * void Display();
   * }
   * // Leaf class (individual object)
   * class Leaf : IComponent
   * {
   * private string name;
   * public Leaf(string name)
   * {
   * this.name = name;
   * }
   * public void Display()
   * {
   * Console.WriteLine($"Leaf: {name}");
   * }
   * }
   * // Composite class (composite object)
   * class Composite : IComponent
   * {
   * private List<IComponent> children = new List<IComponent>();
   * public void Add(IComponent component)
   * {
   * children.Add(component);
   * }
   * public void Remove(IComponent component)
   * {
   * children.Remove(component);
   * }
   * public void Display()
   * {
   * Console.WriteLine("Composite:");
   * foreach (var child in children)
   * {
   * child.Display();
   * }
   * }
   * }
   * // Client code
   * class Client
   * {
   * static void Main()
   * {
   * // Create leaf objects
   * Leaf leaf1 = new Leaf("Leaf 1");
   * Leaf leaf2 = new Leaf("Leaf 2");
   * Leaf leaf3 = new Leaf("Leaf 3");
   * // Create a composite object and add leaf objects to it
   * Composite composite = new Composite();
   * composite.Add(leaf1);
   * composite.Add(leaf2);
   * // Create another composite object and add a leaf and the previous composite object to it
   * Composite composite2 = new Composite();
   * composite2.Add(leaf3);
   * composite2.Add(composite);
   * // Display the structure
   * composite2.Display();
   * /\*
   * Output:
   * Composite:
   * Leaf: Leaf 3
   * Composite:
   * Leaf: Leaf 1
   * Leaf: Leaf 2
   * \*/
   * }
   * }

1. **Decorator Pattern:**
   * Attaches additional responsibilities to an object dynamically.
   * Ex: //using System;
   * Component
   * interface Coffee {
   * double cost();
   * String description();
   * }
   * // Concrete Component
   * class SimpleCoffee implements Coffee {
   * public double cost() {
   * return 2.0; // Base cost of simple coffee
   * }
   * public String description() {
   * return "Simple Coffee";
   * }
   * }
   * // Decorator
   * abstract class CoffeeDecorator implements Coffee {
   * private Coffee decoratedCoffee;
   * public CoffeeDecorator(Coffee decoratedCoffee) {
   * this.decoratedCoffee = decoratedCoffee;
   * }
   * public double cost() {
   * return decoratedCoffee.cost();
   * }
   * public String description() {
   * return decoratedCoffee.description();
   * }
   * }
   * // Concrete Decorator A
   * class MilkDecorator extends CoffeeDecorator {
   * public MilkDecorator(Coffee decoratedCoffee) {
   * super(decoratedCoffee);
   * }
   * public double cost() {
   * return super.cost() + 0.5; // Additional cost for milk
   * }
   * public String description() {
   * return super.description() + " with Milk";
   * }
   * }
   * // Concrete Decorator B
   * class SugarDecorator extends CoffeeDecorator {
   * public SugarDecorator(Coffee decoratedCoffee) {
   * super(decoratedCoffee);
   * }
   * public double cost() {
   * return super.cost() + 0.2; // Additional cost for sugar
   * }
   * public String description() {
   * return super.description() + " with Sugar";
   * }
   * }
   * // Client code
   * public class DecoratorPatternExample {
   * public static void main(String[] args) {
   * // Create a simple coffee
   * Coffee simpleCoffee = new SimpleCoffee();
   * System.out.println("Cost: " + simpleCoffee.cost() + ", Description: " + simpleCoffee.description());
   * // Add milk to the coffee
   * Coffee milkCoffee = new MilkDecorator(simpleCoffee);
   * System.out.println("Cost: " + milkCoffee.cost() + ", Description: " + milkCoffee.description());
   * // Add sugar to the coffee with milk
   * Coffee sugarMilkCoffee = new SugarDecorator(milkCoffee);
   * Console.WriteLine("Cost: " + sugarMilkCoffee.cost() + ", Description: " + sugarMilkCoffee.description());
   * }
   * }
2. **Facade Pattern:**
   * Provides a simplified interface to a set of interfaces in a subsystem.
   * Ex: using System;
   * // Subsystem components
   * class SubsystemA
   * {
   * public void OperationA()
   * {
   * Console.WriteLine("Subsystem A - Operation A");
   * }
   * }
   * class SubsystemB
   * {
   * public void OperationB()
   * {
   * Console.WriteLine("Subsystem B - Operation B");
   * }
   * }
   * class SubsystemC
   * {
   * public void OperationC()
   * {
   * Console.WriteLine("Subsystem C - Operation C");
   * }
   * }
   * // Facade
   * class Facade
   * {
   * private SubsystemA subsystemA;
   * private SubsystemB subsystemB;
   * private SubsystemC subsystemC;
   * public Facade()
   * {
   * subsystemA = new SubsystemA();
   * subsystemB = new SubsystemB();
   * subsystemC = new SubsystemC();
   * }
   * // High-level methods that hide the complexity of the subsystem
   * public void PerformOperation()
   * {
   * Console.WriteLine("Facade is performing operation:");
   * subsystemA.OperationA();
   * subsystemB.OperationB();
   * subsystemC.OperationC();
   * }
   * }
   * // Client code
   * class Program
   * {
   * static void Main()
   * {
   * // Using the facade to perform operations on the subsystem
   * Facade facade = new Facade();
   * facade.PerformOperation();
   * }
   * }
3. **Flyweight Pattern:**
   * Reduces the number of objects created by sharing objects to be used in multiple contexts.
   * Ex: using System;
   * using System.Collections.Generic;
   * // Flyweight interface
   * interface IFlyweight
   * {
   * void Operation(int extrinsicState);
   * }
   * // ConcreteFlyweight
   * class ConcreteFlyweight : IFlyweight
   * {
   * private int intrinsicState;
   * public ConcreteFlyweight(int intrinsicState)
   * {
   * this.intrinsicState = intrinsicState;
   * }
   * public void Operation(int extrinsicState)
   * {
   * Console.WriteLine($"Concrete Flyweight: Intrinsic State - {intrinsicState}, Extrinsic State - {extrinsicState}");
   * }
   * }
   * // FlyweightFactory
   * class FlyweightFactory
   * {
   * private Dictionary<int, IFlyweight> flyweights = new Dictionary<int, IFlyweight>();
   * public IFlyweight GetFlyweight(int key)
   * {
   * if (!flyweights.ContainsKey(key))
   * {
   * flyweights[key] = new ConcreteFlyweight(key);
   * }
   * return flyweights[key];
   * }
   * }
   * // Client
   * class Client
   * {
   * static void Main()
   * {
   * FlyweightFactory factory = new FlyweightFactory();
   * // Client 1: Reuses existing flyweights
   * IFlyweight flyweight1 = factory.GetFlyweight(1);
   * flyweight1.Operation(100);
   * // Client 2: Reuses existing flyweights
   * IFlyweight flyweight2 = factory.GetFlyweight(2);
   * flyweight2.Operation(200);
   * // Client 3: Reuses existing flyweights
   * IFlyweight flyweight3 = factory.GetFlyweight(1); // Reusing the flyweight with key 1
   * flyweight3.Operation(300);
   * }
   * }
4. **Proxy Pattern:**
   * Provides a surrogate or placeholder for another object to control access to it.

Ex: using System;

// Subject interface

interface IImage

{

void Display();

}

// RealSubject (RealImage) - The real object

class RealImage : IImage

{

private string filename;

public RealImage(string filename)

{

this.filename = filename;

LoadImage();

}

private void LoadImage()

{

Console.WriteLine($"Loading image: {filename}");

// Simulate loading an image from a file

}

public void Display()

{

Console.WriteLine($"Displaying image: {filename}");

}

}

// Proxy - Virtual Proxy

class ImageProxy : IImage

{

private RealImage realImage;

private string filename;

public ImageProxy(string filename)

{

this.filename = filename;

}

public void Display()

{

// Create the RealImage only when it is requested

if (realImage == null)

{

realImage = new RealImage(filename);

}

// Delegate the display operation to the RealImage

realImage.Display();

}

}

// Client code

class Program

{

static void Main()

{

// Using the ImageProxy to control access to the RealImage

// RealImage is not loaded until Display is called on the proxy

IImage imageProxy = new ImageProxy("sample.jpg");

Console.WriteLine("ImageProxy created.");

// RealImage is loaded and displayed only when needed

imageProxy.Display();

}

}

**Behavioral Patterns:**

1. **Chain of Responsibility Pattern:**
   * Passes requests along a chain of handlers.
   * Ex: using System;
   * // Handler interface
   * interface IApprover
   * {
   * void ProcessRequest(PurchaseRequest request);
   * }
   * // Concrete Handler A
   * class Manager : IApprover
   * {
   * private readonly decimal approvalLimit = 10000;
   * public void ProcessRequest(PurchaseRequest request)
   * {
   * if (request.Amount <= approvalLimit)
   * {
   * Console.WriteLine($"Manager approves purchase request #{request.RequestNumber}");
   * }
   * else
   * {
   * Console.WriteLine($"Manager cannot approve. Escalating to Director.");
   * // Pass the request to the next approver in the chain
   * // (In a real scenario, you might have a reference to the next handler, or use a list of handlers.)
   * }
   * }
   * }
   * // Concrete Handler B
   * class Director : IApprover
   * {
   * private readonly decimal approvalLimit = 50000;
   * public void ProcessRequest(PurchaseRequest request)
   * {
   * if (request.Amount <= approvalLimit)
   * {
   * Console.WriteLine($"Director approves purchase request #{request.RequestNumber}");
   * }
   * else
   * {
   * Console.WriteLine($"Director cannot approve. Escalating to Vice President.");
   * // Pass the request to the next approver in the chain
   * }
   * }
   * }
   * // Concrete Handler C
   * class VicePresident : IApprover
   * {
   * private readonly decimal approvalLimit = 100000;
   * public void ProcessRequest(PurchaseRequest request)
   * {
   * if (request.Amount <= approvalLimit)
   * {
   * Console.WriteLine($"Vice President approves purchase request #{request.RequestNumber}");
   * }
   * else
   * {
   * Console.WriteLine($"Vice President cannot approve. Request denied.");
   * // End of the chain - no next approver
   * }
   * }
   * }
   * // Request class
   * class PurchaseRequest
   * {
   * public int RequestNumber { get; }
   * public string Purpose { get; }
   * public decimal Amount { get; }
   * public PurchaseRequest(int requestNumber, string purpose, decimal amount)
   * {
   * RequestNumber = requestNumber;
   * Purpose = purpose;
   * Amount = amount;
   * }
   * }
   * // Client code
   * class Program
   * {
   * static void Main()
   * {
   * // Creating a chain of handlers
   * IApprover manager = new Manager();
   * IApprover director = new Director();
   * IApprover vicePresident = new VicePresident();
   * // Setting up the chain of responsibility
   * manager.ProcessRequest(new PurchaseRequest(1, "Office Supplies", 8000));
   * manager.ProcessRequest(new PurchaseRequest(2, "Conference Room Furniture", 30000));
   * manager.ProcessRequest(new PurchaseRequest(3, "New Server", 120000));
   * }
   * }
2. **Command Pattern:**
   * Encapsulates a request as an object, allowing for parameterization, queuing, and logging of requests.
   * Ex: using System;
   * // Command interface
   * interface ICommand
   * {
   * void Execute();
   * }
   * // Concrete Command A
   * class LightOnCommand : ICommand
   * {
   * private readonly Light light;
   * public LightOnCommand(Light light)
   * {
   * this.light = light;
   * }
   * public void Execute()
   * {
   * light.TurnOn();
   * }
   * }
   * // Concrete Command B
   * class LightOffCommand : ICommand
   * {
   * private readonly Light light;
   * public LightOffCommand(Light light)
   * {
   * this.light = light;
   * }
   * public void Execute()
   * {
   * light.TurnOff();
   * }
   * }
   * // Receiver
   * class Light
   * {
   * public void TurnOn()
   * {
   * Console.WriteLine("Light is ON");
   * }
   * public void TurnOff()
   * {
   * Console.WriteLine("Light is OFF");
   * }
   * }
   * // Invoker
   * class RemoteControl
   * {
   * private ICommand command;
   * public void SetCommand(ICommand command)
   * {
   * this.command = command;
   * }
   * public void PressButton()
   * {
   * command.Execute();
   * }
   * }
   * // Client code
   * class Program
   * {
   * static void Main()
   * {
   * // Creating the receiver
   * Light light = new Light();
   * // Creating the concrete commands
   * ICommand lightOnCommand = new LightOnCommand(light);
   * ICommand lightOffCommand = new LightOffCommand(light);
   * // Creating the invoker
   * RemoteControl remote = new RemoteControl();
   * // Associating commands with the invoker
   * remote.SetCommand(lightOnCommand);
   * // Pressing the button to execute the command
   * remote.PressButton(); // Output: Light is ON
   * // Changing the command
   * remote.SetCommand(lightOffCommand);
   * // Pressing the button again to execute the new command
   * remote.PressButton(); // Output: Light is OFF
   * }
   * }
3. **Interpreter Pattern:**
   * Defines a grammar for interpreting the sentences in a language.
   * Ex: using System;
   * using System.Collections.Generic;
   * // Abstract expression class
   * abstract class Expression
   * {
   * public abstract int Interpret();
   * }
   * // Terminal expression for representing numbers
   * class NumberExpression : Expression
   * {
   * private int \_number;
   * public NumberExpression(int number)
   * {
   * \_number = number;
   * }
   * public override int Interpret()
   * {
   * return \_number;
   * }
   * }
   * // Non-terminal expression for addition
   * class AddExpression : Expression
   * {
   * private Expression \_left;
   * private Expression \_right;
   * public AddExpression(Expression left, Expression right)
   * {
   * \_left = left;
   * \_right = right;
   * }
   * public override int Interpret()
   * {
   * return \_left.Interpret() + \_right.Interpret();
   * }
   * }
   * // Non-terminal expression for subtraction
   * class SubtractExpression : Expression
   * {
   * private Expression \_left;
   * private Expression \_right;
   * public SubtractExpression(Expression left, Expression right)
   * {
   * \_left = left;
   * \_right = right;
   * }
   * public override int Interpret()
   * {
   * return \_left.Interpret() - \_right.Interpret();
   * }
   * }
   * // Client code
   * class Client
   * {
   * static void Main()
   * {
   * // Create a simple expression: 5 + 3 - 2
   * Expression expression = new SubtractExpression(
   * new AddExpression(new NumberExpression(5), new NumberExpression(3)),
   * new NumberExpression(2));
   * // Interpret the expression
   * int result = expression.Interpret();
   * Console.WriteLine("Result: " + result); // Output: Result: 6
   * }
   * }
4. **Iterator Pattern:**
   * Provides a way to access elements of an aggregate object sequentially without exposing its underlying representation.
   * Ex: using System;
   * using System.Collections;
   * // Iterator interface
   * interface IIterator
   * {
   * bool HasNext();
   * object Next();
   * }
   * // Concrete iterator class
   * class IntegerIterator : IIterator
   * {
   * private int[] \_collection;
   * private int \_index;
   * public IntegerIterator(int[] collection)
   * {
   * \_collection = collection;
   * \_index = 0;
   * }
   * public bool HasNext()
   * {
   * return \_index < \_collection.Length;
   * }
   * public object Next()
   * {
   * if (HasNext())
   * {
   * int element = \_collection[\_index];
   * \_index++;
   * return element;
   * }
   * else
   * {
   * throw new InvalidOperationException("No more elements to iterate");
   * }
   * }
   * }
   * // Aggregate interface
   * interface IAggregate
   * {
   * IIterator CreateIterator();
   * }
   * // Concrete aggregate class
   * class IntegerCollection : IAggregate
   * {
   * private int[] \_collection;
   * public IntegerCollection(int[] collection)
   * {
   * \_collection = collection;
   * }
   * public IIterator CreateIterator()
   * {
   * return new IntegerIterator(\_collection);
   * }
   * }
   * // Client code
   * class Client
   * {
   * static void Main()
   * {
   * int[] numbers = { 1, 2, 3, 4, 5 };
   * // Create a concrete aggregate
   * IAggregate aggregate = new IntegerCollection(numbers);
   * // Create an iterator
   * IIterator iterator = aggregate.CreateIterator();
   * // Iterate over the elements
   * while (iterator.HasNext())
   * {
   * int element = (int)iterator.Next();
   * Console.WriteLine("Element: " + element);
   * }
   * }
   * }
5. **Mediator Pattern:**
   * Defines an object that centralizes communication between objects in a system.
   * Ex: using System;
   * using System.Collections.Generic;
   * // Mediator interface
   * interface IChatMediator
   * {
   * void AddUser(IUser user);
   * void SendMessage(IUser sender, string message);
   * }
   * // Colleague interface
   * interface IUser
   * {
   * string Name { get; }
   * void ReceiveMessage(string message);
   * }
   * // Concrete Mediator class
   * class ChatMediator : IChatMediator
   * {
   * private List<IUser> \_users;
   * public ChatMediator()
   * {
   * \_users = new List<IUser>();
   * }
   * public void AddUser(IUser user)
   * {
   * \_users.Add(user);
   * }
   * public void SendMessage(IUser sender, string message)
   * {
   * foreach (var user in \_users)
   * {
   * // Exclude the sender from receiving the message
   * if (user
6. **Memento Pattern:**
   * Captures and externalizes an object's internal state so that it can be restored to this state later.
   * Ex: using System;
   * using System.Collections.Generic;
   * // Memento: Represents the state of the text editor
   * class EditorMemento
   * {
   * public string Text { get; }
   * public EditorMemento(string text)
   * {
   * Text = text;
   * }
   * }
   * // Originator: Represents the text editor
   * class TextEditor
   * {
   * private string \_text;
   * public string Text
   * {
   * get { return \_text; }
   * set
   * {
   * \_text = value;
   * Console.WriteLine("Text set to: " + \_text);
   * }
   * }
   * // Creates a memento representing the current state
   * public EditorMemento CreateMemento()
   * {
   * return new EditorMemento(\_text);
   * }
   * // Restores the state from a memento
   * public void RestoreMemento(EditorMemento memento)
   * {
   * \_text = memento.Text;
   * Console.WriteLine("Text restored to: " + \_text);
   * }
   * }
   * // Caretaker: Manages the history of the text editor's state
   * class History
7. **Observer Pattern:**
   * Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
   * Ex: using System;
   * using System.Collections.Generic;
   * // Subject interface
   * interface INewsPublisher
   * {
   * void AddSubscriber(INewsSubscriber subscriber);
   * void RemoveSubscriber(INewsSubscriber subscriber);
   * void NotifySubscribers(string news);
   * }
   * // Concrete Subject
   * class NewsAgency : INewsPublisher
   * {
   * private List<INewsSubscriber> \_subscribers = new List<INewsSubscriber>();
   * public void AddSubscriber(INewsSubscriber subscriber)
   * {
   * \_subscribers.Add(subscriber);
   * }
   * public void RemoveSubscriber(INewsSubscriber subscriber)
   * {
   * \_subscribers.Remove(subscriber);
   * }
   * public void NotifySubscribers(string news)
   * {
   * foreach (var subscriber in \_subscribers)
   * {
   * subscriber.Update(news);
   * }
   * }
   * public void PublishNews(string news)
   * {
   * Console.WriteLine("News Agency: " + news);
   * NotifySubscribers(news);
   * }
   * }
   * // Observer interface
   * interface INewsSubscriber
   * {
   * void Update(string news);
   * }
   * // Concrete Observer
   * class NewsSubscriber : INewsSubscriber
   * {
   * private string \_name;
   * public NewsSubscriber(string name)
   * {
   * \_name = name;
   * }
   * public void Update(string news)
   * {
   * Console.WriteLine($"{\_name} received news: {news}");
   * }
   * }
   * // Client code
   * class Client
   * {
   * static void Main()
   * {
   * // Create a news agency
   * NewsAgency newsAgency = new NewsAgency();
   * // Create news subscribers
   * INewsSubscriber subscriber1 = new NewsSubscriber("Subscriber 1");
   * INewsSubscriber subscriber2 = new NewsSubscriber("Subscriber 2");
   * INewsSubscriber subscriber3 = new NewsSubscribe
8. **State Pattern:**
   * Allows an object to alter its behavior when its internal state changes.
   * Ex: using System;
   * // Context
   * class VideoPlayer
   * {
   * private IState \_state;
   * public VideoPlayer()
   * {
   * // Initial state is stopped
   * \_state = new StoppedState(this);
   * }
   * public void SetState(IState state)
   * {
   * \_state = state;
   * }
   * public void Play()
   * {
   * \_state.Play();
   * }
   * public void Pause()
   * {
   * \_state.Pause();
   * }
   * public void Stop()
   * {
   * \_state.Stop();
   * }
   * }
   * // State interface
   * interface IState
   * {
   * void Play();
   * void Pause();
   * void Stop();
   * }
   * // Concrete State: Playing
   * class PlayingState : IState
   * {
   * private readonly VideoPlayer \_player;
   * public PlayingState(VideoPlayer player)
   * {
   * \_player = player;
   * }
   * public void Play()
   * {
   * Console.WriteLine("Already playing");
   * }
   * public void Pause()
   * {
   * Console.WriteLine("Pausing video");
   * \_player.SetState(new PausedState(\_player));
   * }
   * public void Stop()
   * {
   * Console.WriteLine("Stopping video");
   * \_player.SetState(new StoppedState(\_player));
   * }
   * }
   * // Concrete State: Paused
   * class PausedState : IState
   * {
   * private readonly VideoPlayer \_player;
   * public PausedState(VideoPlayer player)
   * {
   * \_player = player;
   * }
   * public void Play()
   * {
   * Console.WriteLine("Resuming video");
   * \_player.SetState(new PlayingState(\_player));
   * }
   * public void Pause()
   * {
   * Console.WriteLine("Already paused");
   * }
   * public void Stop()
   * {
   * Console.WriteLine("Stopping video");
   * \_player.SetState(new StoppedState(\_player));
   * }
   * }
   * // Concrete State: Stopped
   * class StoppedState : IState
   * {
   * private readonly VideoPlayer \_player;
   * public StoppedState(VideoPlayer player)
   * {
   * \_player = player;
   * }
   * public void Play()
   * {
   * Console.WriteLine("Playing video");
   * \_player.SetState(new PlayingState(\_player));
   * }
   * public void Pause()
   * {
   * Console.WriteLine("Cannot pause, video is stopped");
   * }
   * public void Stop()
   * {
   * Console.WriteLine("Already stopped");
   * }
   * }
   * // Client code
   * class Client
   * {
   * static void Main()
   * {
   * VideoPlayer player = new VideoPlayer();
   * player.Play();
   * player.Pause();
   * player.Stop();
   * player.Stop();
   * /\*
   * Output:
   * Playing video
   * Pausing video
   * Stopping video
   * Already stopped
   * \*/
   * }
   * }
9. **Strategy Pattern:**
   * Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
   * Ex: using System;
   * // Strategy interface
   * interface IDiscountStrategy
   * {
   * double ApplyDiscount(double total);
   * }
   * // Concrete Strategy: NoDiscount
   * class NoDiscountStrategy : IDiscountStrategy
   * {
   * public double ApplyDiscount(double total)
   * {
   * return total; // No discount applied
   * }
   * }
   * // Concrete Strategy: FixedDiscount
   * class FixedDiscountStrategy : IDiscountStrategy
   * {
   * private double \_discountAmount;
   * public FixedDiscountStrategy(double discountAmount)
   * {
   * \_discountAmount = discountAmount;
   * }
   * public double ApplyDiscount(double total)
   * {
   * return Math.Max(0, total - \_discountAmount); // Apply fixed discount
   * }
   * }
   * // Concrete Strategy: PercentageDiscount
   * class PercentageDiscountStrategy : IDiscountStrategy
   * {
   * private double \_percentage;
   * public PercentageDiscountStrategy(double percentage)
   * {
   * \_percentage = percentage;
   * }
   * public double ApplyDiscount(double total)
   * {
   * return total - (total \* \_percentage / 100); // Apply percentage discount
   * }
   * }
   * // Context: BillingSystem
   * class BillingSystem
   * {
   * private IDiscountStrategy \_discountStrategy;
   * public BillingSystem(IDiscountStrategy discountStrategy)
   * {
   * \_discountStrategy = discountStrategy;
   * }
   * public void SetDiscountStrategy(IDiscountStrategy discountStrategy)
   * {
   * \_discountStrategy = discountStrategy;
   * }
   * public double CalculateTotal(double orderTotal)
   * {
   * // Delegate the discount calculation to the current strategy
   * return \_discountStrategy.ApplyDiscount(orderTotal);
   * }
   * }
   * // Client code
   * class Client
   * {
   * static void Main()
   * {
   * // Create a billing system with no discount strategy
   * BillingSystem billingSystem = new BillingSystem(new NoDiscountStrategy());
   * // Calculate total without discount
   * double totalWithoutDiscount = billingSystem.CalculateTotal(100);
   * Console.WriteLine("Total without discount: $" + totalWithoutDiscount);
   * // Switch to a fixed discount strategy
   * billingSystem.SetDiscountStrategy(new FixedDiscountStrategy(10));
   * // Calculate total with fixed discount
   * double totalWithFixedDiscount = billingSystem.CalculateTotal(100);
   * Console.WriteLine("Total with fixed discount: $" + totalWithFixedDiscount);
   * // Switch to a percentage discount strategy
   * billingSystem.SetDiscountStrategy(new PercentageDiscountStrategy(20));
   * // Calculate total with percentage discount
   * double totalWithPercentageDiscount = billingSystem.CalculateTotal(100);
   * Console.WriteLine("Total with percentage discount: $" + totalWithPercentageDiscount);
   * /\*
   * Output:
   * Total without discount: $100
   * Total with fixed discount: $90
   * Total with percentage discount: $80
   * \*/
   * }
   * }
10. **Template Method Pattern:**
    * Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
    * Ex: using System;
    * // Abstract class defining the template method
    * abstract class HotBeverageTemplate
    * {
    * // The template method defines the algorithm structure
    * public void PrepareBeverage()
    * {
    * BoilWater();
    * Brew();
    * PourInCup();
    * AddCondiments();
    * }
    * // Abstract methods to be implemented by concrete subclasses
    * protected abstract void Brew();
    * protected abstract void AddCondiments();
    * // Common steps implemented in the base class
    * private void BoilWater()
    * {
    * Console.WriteLine("Boiling water");
    * }
    * private void PourInCup()
    * {
    * Console.WriteLine("Pouring into cup");
    * }
    * }
    * // Concrete class implementing specific steps for tea
    * class Tea : HotBeverageTemplate
    * {
    * protected override void Brew()
    * {
    * Console.WriteLine("Steeping the tea");
    * }
    * protected override void AddCondiments()
    * {
    * Console.WriteLine("Adding lemon");
    * }
    * }
    * // Concrete class implementing specific steps for coffee
    * class Coffee : HotBeverageTemplate
    * {
    * protected override void Brew()
    * {
    * Console.WriteLine("Dripping coffee through filter");
    * }
    * protected override void AddCondiments()
    * {
    * Console.WriteLine("Adding sugar and milk");
    * }
    * }
    * // Client code
    * class Client
    * {
    * static void Main()
    * {
    * Console.WriteLine("Making tea:");
    * HotBeverageTemplate tea = new Tea();
    * tea.PrepareBeverage();
    * Console.WriteLine("\nMaking coffee:");
    * HotBeverageTemplate coffee = new Coffee();
    * coffee.PrepareBeverage();
    * /\*
    * Output:
    * Making tea:
    * Boiling water
    * Steeping the tea
    * Pouring into cup
    * Adding lemon
    * Making coffee:
    * Boiling water
    * Dripping coffee through filter
    * Pouring into cup
    * Adding sugar and milk
    * \*/
    * }
    * }
11. **Visitor Pattern:**
    * Represents an operation to be performed on the elements of an object structure.
    * Ex: using System;
    * using System.Collections.Generic;
    * // Element interface
    * interface IShape
    * {
    * void Accept(IVisitor visitor);
    * }
    * // Concrete Element: Circle
    * class Circle : IShape
    * {
    * public double Radius { get; }
    * public Circle(double radius)
    * {
    * Radius = radius;
    * }
    * public void Accept(IVisitor visitor)
    * {
    * visitor.VisitCircle(this);
    * }
    * }
    * // Concrete Element: Rectangle
    * class Rectangle : IShape
    * {
    * public double Width { get; }
    * public double Height { get; }
    * public Rectangle(double width, double height)
    * {
    * Width = width;
    * Height = height;
    * }
    * public void Accept(IVisitor visitor)
    * {
    * visitor.VisitRectangle(this);
    * }
    * }
    * // Visitor interface
    * interface IVisitor
    * {
    * void VisitCircle(Circle circle);
    * void VisitRectangle(Rectangle rectangle);
    * }
    * // Concrete Visitor: AreaCalculator
    * class AreaCalculator : IVisitor
    * {
    * public void VisitCircle(Circle circle)
    * {
    * double area = Math.PI \* circle.Radius \* circle.Radius;
    * Console.WriteLine($"Area of circle with radius {circle.Radius}: {area}");
    * }
    * public void VisitRectangle(Rectangle rectangle)
    * {
    * double area = rectangle.Width \* rectangle.Height;
    * Console.WriteLine($"Area of rectangle with width {rectangle.Width} and height {rectangle.Height}: {area}");
    * }
    * }
    * // Concrete Visitor: PerimeterCalculator
    * class PerimeterCalculator : IVisitor
    * {
    * public void VisitCircle(Circle circle)
    * {
    * double perimeter = 2 \* Math.PI \* circle.Radius;
    * Console.WriteLine($"Perimeter of circle with radius {circle.Radius}: {perimeter}");
    * }
    * public void VisitRectangle(Rectangle rectangle)
    * {
    * double perimeter = 2 \* (rectangle.Width + rectangle.Height);
    * Console.WriteLine($"Perimeter of rectangle with width {rectangle.Width} and height {rectangle.Height}: {perimeter}");
    * }
    * }
    * // Client code
    * class Client
    * {
    * static void Main()
    * {
    * List<IShape> shapes = new List<IShape>
    * {
    * new Circle(5),
    * new Rectangle(3, 4),
    * new Circle(8)
    * };
    * // Use AreaCalculator to calculate areas
    * AreaCalculator areaCalculator = new AreaCalculator();
    * foreach (var shape in shapes)
    * {
    * shape.Accept(areaCalculator);
    * }
    * Console.WriteLine();
    * // Use PerimeterCalculator to calculate perimeters
    * PerimeterCalculator perimeterCalculator = new PerimeterCalculator();
    * foreach (var shape in shapes)
    * {
    * shape.Accept(perimeterCalculator);
    * }
    * /\*
    * Output:
    * Area of circle with radius 5: 78.53981633974483
    * Area of rectangle with width 3 and height 4: 12
    * Area of circle with radius 8: 201.06192982974676
    * Perimeter of circle with radius 5: 31.41592653589793
    * Perimeter of rectangle with width 3 and height 4: 14
    * Perimeter of circle with radius 8: 50.26548245743669
    * \*/
    * }
    * }

These design patterns provide solutions to recurring problems in software design and serve as a guide for designing reusable and maintainable code. For more in-depth understanding, it's recommended to refer to the original book "Design Patterns: Elements of Reusable Object-Oriented Software" by the Gang of Four.

Top of Form