Gang of Four (GoF) design patterns, detailed in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, include 23 design patterns that are categorized into three main types: **Creational**, **Structural**, and **Behavioral** patterns. Below, I've outlined each of the GoF patterns with advanced real-time examples in C#.

Creational Patterns

1. Abstract Factory Pattern

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

Use Case:
Creating UI components (buttons, checkboxes) that vary by theme (e.g., Windows, Mac).

Example:

```
// Abstract Factory
public interface IUIFactory
{
    IButton CreateButton();
    ICheckbox CreateCheckbox();
}

// Concrete Factory for Windows
public class WindowsUIFactory : IUIFactory
{
    public IButton CreateButton() => new WindowsButton();
    public ICheckbox CreateCheckbox() => new WindowsCheckbox();
}

// Concrete Factory for Mac
public class MacUIFactory : IUIFactory
{
    public IButton CreateButton() => new MacButton();
    public ICheckbox CreateCheckbox() => new MacCheckbox();
}

// Abstract Products
public interface IButton { void Render(); }
public interface ICheckbox { void Render(); }
```

```
// Concrete Products
public class WindowsButton : IButton
{
    public void Render() => Console.WriteLine("Rendering Windows
Button.");
}

public class MacButton : IButton
{
    public void Render() => Console.WriteLine("Rendering Mac
Button.");
}

public class WindowsCheckbox : ICheckbox
{
    public void Render() => Console.WriteLine("Rendering Windows
Checkbox.");
}

public class MacCheckbox : ICheckbox
{
    public void Render() => Console.WriteLine("Rendering Mac
Checkbox.");
}

// Client Code
public class Application
{
    private readonly IButton _button;
    private readonly ICheckbox _checkbox;

    public Application(IUIFactory factory)
    {
        _button = factory.CreateButton();
        _checkbox = factory.CreateCheckbox();
    }

    public void Render()
    {
        _button.Render();
        _checkbox.Render();
    }
}

// Usage
class Program
{
    static void Main()
    {
        IUIFactory factory = new WindowsUIFactory();
        Application app = new Application(factory);
        app.Render();
    }
}
```

---

2. Builder Pattern

Definition:
Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Use Case:
Building complex objects like a pizza with various toppings.

Example:

```
// Product
public class Pizza
{
    public string Dough { get; set; }
    public string Sauce { get; set; }
    public List<string> Toppings { get; } = new List<string>();

    public override string ToString() =>
        $"Pizza with {Dough} dough, {Sauce} sauce and toppings:
{string.Join(", ", Toppings)}";
}

// Builder Interface
public interface IPizzaBuilder
```

```
{
    void SetDough(string dough);
    void SetSauce(string sauce);
    void AddTopping(string topping);
    Pizza Build();
}

// Concrete Builder
public class MargheritaPizzaBuilder : IPizzaBuilder
{
    private Pizza _pizza = new Pizza();

    public void SetDough(string dough) => _pizza.Dough = dough;
    public void SetSauce(string sauce) => _pizza.Sauce = sauce;
    public void AddTopping(string topping) =>
_pizza.Toppings.Add(topping);
    public Pizza Build() => _pizza;
}

// Director
public class PizzaDirector
{
    private readonly IPizzaBuilder _builder;

    public PizzaDirector(IPizzaBuilder builder) => _builder =
builder;

    public Pizza ConstructMargheritaPizza()
    {
        _builder.SetDough("Thin Crust");
        _builder.SetSauce("Tomato");
        _builder.AddTopping("Mozzarella");
        return _builder.Build();
    }
}

// Usage
class Program
{
    static void Main()
    {
        IPizzaBuilder builder = new MargheritaPizzaBuilder();
        PizzaDirector director = new PizzaDirector(builder);
        Pizza pizza = director.ConstructMargheritaPizza();
        Console.WriteLine(pizza);
    }
}
```

---

3. Factory Method Pattern

Definition:
Defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.

Use Case:
Logging frameworks that can produce different types of logs (e.g., file, console).

Example:

csharp
Copy code

```
// Product Interface
public interface ILogger
{
    void Log(string message);
}

// Concrete Products
public class FileLogger : ILogger
{
    public void Log(string message) => Console.WriteLine($"Logging
to file: {message}");
}

public class ConsoleLogger : ILogger
{
    public void Log(string message) => Console.WriteLine($"Logging
to console: {message}");
}

// Creator
public abstract class LoggerFactory
{
    public abstract ILogger CreateLogger();
}

// Concrete Creators
public class FileLoggerFactory : LoggerFactory
{
    public override ILogger CreateLogger() => new FileLogger();
}

public class ConsoleLoggerFactory : LoggerFactory
{
    public override ILogger CreateLogger() => new ConsoleLogger();
}

// Client Code
class Program
{
    static void Main()
    {
        LoggerFactory factory = new ConsoleLoggerFactory();
        ILogger logger = factory.CreateLogger();
        logger.Log("This is a log message.");
    }
}
```

---

4. Prototype Pattern

Definition:
Creates new objects by copying an existing object, known as the prototype.

Use Case:
Cloning complex objects, like game characters with various attributes.

Example:

csharp
Copy code

```
// Prototype Interface
public interface ICloneable
{
    ICloneable Clone();
}
```

```
// Concrete Prototype
public class GameCharacter : ICloneable
{
    public string Name { get; set; }
    public int Health { get; set; }

    public ICloneable Clone() => new GameCharacter { Name =
this.Name, Health = this.Health };
}

// Usage
class Program
{
    static void Main()
    {
        GameCharacter original = new GameCharacter { Name = "Hero",
Health = 100 };
        GameCharacter clone = (GameCharacter)original.Clone();

        Console.WriteLine($"Original: {original.Name}, Health:
{original.Health}");
        Console.WriteLine($"Clone: {clone.Name}, Health:
{clone.Health}");
    }
}
```

---

5. Singleton Pattern

Definition:
Ensures that a class has only one instance and provides a global point of access to it.

Use Case:
Configuration settings that should be shared across the application.

Example:

csharp
Copy code

```
// Singleton
public class ConfigurationManager
{
    private static ConfigurationManager _instance;
    private static readonly object _lock = new object();

    private ConfigurationManager() { } // Private constructor

    public static ConfigurationManager Instance
    {
        get
        {
            lock (_lock)
            {
                return _instance ??= new ConfigurationManager();
            }
        }
    }

    public string GetSetting(string key) => "some value"; // Example method
}

// Usage
class Program
{
    static void Main()
    {
        var config = ConfigurationManager.Instance;
        Console.WriteLine(config.GetSetting("SomeKey"));
    }
}
```

Structural Patterns

6. Adapter Pattern

Definition:
Allows incompatible interfaces to work together by wrapping one interface with another.

Use Case:
Integrating third-party libraries into an existing codebase.

Example:

csharp
Copy code

```csharp
// Target Interface
public interface ITarget
{
    void Request();
}

// Adaptee
public class Adaptee
{
    public void SpecificRequest() => Console.WriteLine("Specific
request from Adaptee.");
}

// Adapter
public class Adapter : ITarget
{
    private readonly Adaptee _adaptee;

    public Adapter(Adaptee adaptee) => _adaptee = adaptee;

    public void Request() => _adaptee.SpecificRequest();
}

// Usage
class Program
{
    static void Main()
    {
        Adaptee adaptee = new Adaptee();
        ITarget adapter = new Adapter(adaptee);
        adapter.Request(); // Outputs: Specific request from
Adaptee.
    }
}
```

---

7. Bridge Pattern

Definition:
Decouples an abstraction from its implementation, allowing them to vary independently.

Use Case:
Developing a drawing application where shapes can be drawn in different styles.

Example:

csharp
Copy code

```
// Abstraction
public abstract class Shape
{
    protected IDrawingAPI _drawingAPI;

    protected Shape(IDrawingAPI drawingAPI) => _drawingAPI =
drawingAPI;

    public abstract void Draw();
}

// Implementation Interface
public interface IDrawingAPI
{
    void DrawCircle(double x, double y, double radius);
}

// Concrete Implementations
public class DrawingAPI1 : IDrawingAPI
{
    public void DrawCircle(double x, double y, double radius) =>
        Console.WriteLine($"Drawing Circle at ({x}, {y}) with radius
{radius} using API 1.");
}

public class DrawingAPI2 : IDrawingAPI
{
    public void DrawCircle(double x, double y, double radius) =>
        Console.WriteLine($"Drawing Circle at ({x}, {y}) with radius
{radius} using API 2.");
}

// Refined Abstraction
public class Circle : Shape
{
    private double _x, _y, _radius;
    public Circle(double x, double y, double radius, IDrawingAPI
drawingAPI)
        : base(drawingAPI)
    {
        _x = x;
        _y = y;
        _radius = radius;
    }

    public override void Draw() => _drawingAPI.DrawCircle(_x, _y,
_radius);
}

// Usage
class Program
{
    static void Main()
    {
        Shape circle1 = new Circle(5, 10, 2, new DrawingAPI1());
        Shape circle2 = new Circle(5, 10, 2, new DrawingAPI2());

        circle1.Draw();
        circle2.Draw();
    }
}
```

8. Composite Pattern

Definition:
Composes objects into tree structures to represent part-whole hierarchies.

Use Case:
Building a file system where directories can contain files and subdirectories.

Example:

csharp
Copy code

```
// Component
public interface IFileSystemComponent
{
    void ShowInfo();
}

// Leaf
public class File : IFileSystemComponent
{
    private string _name;

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

    public void ShowInfo() => Console.WriteLine($"File: {_name}");
}

// Composite
public class Directory : IFileSystemComponent
{
    private string _name;
    private List<IFileSystemComponent> _children = new
List<IFileSystemComponent>();

    public Directory(string name) => _name = name;

    public void Add(IFileSystemComponent component) =>
_children.Add(component);
    public void ShowInfo()
    {
        Console.WriteLine($"Directory: {_name}");
        foreach (var child in _children)
        {
            child.ShowInfo();
        }
    }
}

// Usage
class Program
{
    static void Main()
    {
        var root = new Directory("Root");
        var file1 = new File("File1.txt");
        var file2 = new File("File2.txt");

        var subDir = new Directory("SubDirectory");
        subDir.Add(new File("SubFile1.txt"));

        root.Add(file1);
        root.Add(file2);
        root.Add(subDir);

        root.ShowInfo();
    }
}
```

---

9. Decorator Pattern

Definition:
Allows behavior to be added to individual objects dynamically without affecting the behavior of other objects.

Use Case:
Adding features to a coffee order (like milk, sugar) without changing the core Coffee class.

Example:

csharp
Copy code

```
// Component
public interface ICoffee
{
    double Cost();
    string Description();
}

// Concrete Component
public class SimpleCoffee : ICoffee
{
    public double Cost() => 1.00;
    public string Description() => "Simple Coffee";
}

// Decorator
public abstract class CoffeeDecorator : ICoffee
```

```
{
    protected ICoffee _coffee;

    protected CoffeeDecorator(ICoffee coffee) => _coffee = coffee;

    public virtual double Cost() => _coffee.Cost();
    public virtual string Description() => _coffee.Description();
}

// Concrete Decorators
public class MilkDecorator : CoffeeDecorator
{
    public MilkDecorator(ICoffee coffee) : base(coffee) { }

    public override double Cost() => base.Cost() + 0.50;
    public override string Description() => base.Description() + ",
Milk";
}

public class SugarDecorator : CoffeeDecorator
{
    public SugarDecorator(ICoffee coffee) : base(coffee) { }

    public override double Cost() => base.Cost() + 0.25;
    public override string Description() => base.Description() + ",
Sugar";
}

// Usage
class Program
{
    static void Main()
    {
        ICoffee coffee = new SimpleCoffee();
        Console.WriteLine($"{coffee.Description()} costs
{coffee.Cost()}");

        coffee = new MilkDecorator(coffee);
        Console.WriteLine($"{coffee.Description()} costs
{coffee.Cost()}");
```

```
        coffee = new SugarDecorator(coffee);
        Console.WriteLine($"{coffee.Description()} costs
{coffee.Cost()}");
    }
}
```

---

10. Facade Pattern

Definition:
Provides a simplified interface to a complex subsystem.

Use Case:
Providing a unified interface to a set of interfaces in a subsystem, like a home theater system.

Example:

csharp
Copy code

```
// Subsystem Classes
public class Amplifier
{
    public void On() => Console.WriteLine("Amplifier is on.");
    public void Off() => Console.WriteLine("Amplifier is off.");
}

public class DVDPlayer
{
    public void Play(string movie) => Console.WriteLine($"Playing
{movie}.");
}

// Facade
public class HomeTheaterFacade
{
    private readonly Amplifier _amplifier;
    private readonly DVDPlayer _dvdPlayer;

    public HomeTheaterFacade(Amplifier amplifier, DVDPlayer
dvdPlayer)
    {
        _amplifier = amplifier;
        _dvdPlayer = dvdPlayer;
    }

    public void WatchMovie(string movie)
    {
        _amplifier.On();
        _dvdPlayer.Play(movie);
    }

    public void EndMovie()
    {
        _amplifier.Off();
    }
}

// Usage
class Program
{
    static void Main()
    {
        var amplifier = new Amplifier();
        var dvdPlayer = new DVDPlayer();
        var homeTheater = new HomeTheaterFacade(amplifier,
dvdPlayer);

        homeTheater.WatchMovie("Inception");
        homeTheater.EndMovie();
    }
}
```

---

11. Flyweight Pattern

Definition:
Reduces the cost of creating and manipulating a large number of similar objects.

Use Case:
Managing a large number of graphic objects like characters in a game.

Example:

csharp
Copy code

```
// Flyweight
public class Character
{
    private readonly char _symbol;

    public Character(char symbol) => _symbol = symbol;

    public void Display(int x, int y) =>
Console.WriteLine($"Character: {_symbol} at ({x}, {y})");
}

// Flyweight Factory
public class CharacterFactory
{
    private readonly Dictionary<char, Character> _characters = new
Dictionary<char, Character>();

    public Character GetCharacter(char symbol)
    {
        if (!_characters.ContainsKey(symbol))
        {
            _characters[symbol] = new Character(symbol);
        }
        return _characters[symbol];
    }
}

// Usage
class Program
{
    static void Main()
    {
        var factory = new CharacterFactory();
        string text = "Hello World";

        foreach (var c in text)
        {
            var character = factory.GetCharacter(c);
            character.Display(0, 0); // Simplified position for this
example
        }
    }
}
```

---

12. Proxy Pattern

Definition:
Provides a surrogate or placeholder for another object to control access to it.

Use Case:
Controlling access to a resource that is expensive to create, such as a large image or database connection.

Example:

csharp
Copy code

```csharp
// Subject Interface
public interface IImage
{
    void Display();
}

// Real Subject
public class RealImage : IImage
{
    private readonly string _filename;

    public RealImage(string filename) => _filename = filename;

    public void Display() => Console.WriteLine($"Displaying
{_filename}");
}

// Proxy
public class ProxyImage : IImage
{
    private readonly string _filename;
    private RealImage _realImage;

    public ProxyImage(string filename) => _filename = filename;

    public void Display()
    {
        if (_realImage == null)
        {
            _realImage = new RealImage(_filename);
        }
        _realImage.Display();
    }
}

// Usage
class Program
{
    static void Main()
    {
        IImage image = new ProxyImage("my_picture.jpg");
        image.Display(); // Loads and displays the image
        image.Display(); // Displays the image without reloading
    }
}
```

Behavioral Patterns

13. Chain of Responsibility Pattern

Definition:
Allows passing requests along a chain of handlers. Each handler can either process the request or pass it to the next handler in the chain.

Use Case:
Handling multiple levels of logging (info, warning, error) where each level can choose to log the message or pass it along.

Example:

csharp
Copy code

```
// Handler Interface
public abstract class Logger
{
    protected Logger NextLogger;

    public void SetNext(Logger nextLogger) => NextLogger =
nextLogger;
```

```
    public void LogMessage(string message, LogLevel level)
    {
        if (CanHandle(level))
        {
            Handle(message);
        }
        else
        {
            NextLogger?.LogMessage(message, level);
        }
    }

    protected abstract bool CanHandle(LogLevel level);
    protected abstract void Handle(string message);
}

// Concrete Handlers
public class InfoLogger : Logger
{
    protected override bool CanHandle(LogLevel level) => level ==
LogLevel.Info;

    protected override void Handle(string message) =>
Console.WriteLine($"Info: {message}");
}

public class ErrorLogger : Logger
{
    protected override bool CanHandle(LogLevel level) => level ==
LogLevel.Error;

    protected override void Handle(string message) =>
Console.WriteLine($"Error: {message}");
}

// Log Level Enum
public enum LogLevel
{
    Info,
    Error
```

```
}

// Usage
class Program
{
    static void Main()
    {
        var infoLogger = new InfoLogger();
        var errorLogger = new ErrorLogger();

        infoLogger.SetNext(errorLogger);

        infoLogger.LogMessage("This is an information message.",
LogLevel.Info);
        infoLogger.LogMessage("This is an error message.",
LogLevel.Error);
    }
}
```

---

14. Command Pattern

Definition:
Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.

Use Case:
Implementing an undo functionality in a text editor.

Example:

csharp
Copy code

```
// Command Interface
public interface ICommand
{
    void Execute();
    void Undo();
}

// Concrete Command
public class AddTextCommand : ICommand
{
    private readonly Document _document;
    private readonly string _text;

    public AddTextCommand(Document document, string text)
    {
        _document = document;
        _text = text;
    }

    public void Execute() => _document.AddText(_text);
    public void Undo() => _document.RemoveText(_text);
}

// Receiver
public class Document
{
    private readonly StringBuilder _content = new StringBuilder();

    public void AddText(string text) => _content.Append(text);
    public void RemoveText(string text) =>
_content.Remove(_content.Length - text.Length, text.Length);

    public override string ToString() => _content.ToString();
}

// Invoker
public class TextEditor
{
    private readonly Stack<ICommand> _commandHistory = new
Stack<ICommand>();

    public void ExecuteCommand(ICommand command)
    {
        command.Execute();
        _commandHistory.Push(command);
    }

    public void Undo()
    {
        if (_commandHistory.Count > 0)
        {
            var command = _commandHistory.Pop();
            command.Undo();
        }
    }
}

// Usage
class Program
{
    static void Main()
    {
        var document = new Document();
        var textEditor = new TextEditor();

        var command = new AddTextCommand(document, "Hello, World!");
        textEditor.ExecuteCommand(command);

        Console.WriteLine(document); // Outputs: Hello, World!

        textEditor.Undo();
        Console.WriteLine(document); // Outputs: (empty)
    }
}
```

---

15. Interpreter Pattern

Definition:
Defines a representation for a grammar along with an interpreter to use that grammar.

Use Case:
Parsing and evaluating mathematical expressions.

Example:

csharp
Copy code

```
// Abstract Expression
public interface IExpression
{
    int Interpret();
}
```

```
// Terminal Expression
public class Number : IExpression
{
    private readonly int _number;

    public Number(int number) => _number = number;

    public int Interpret() => _number;
}

// Non-terminal Expression
public class Add : IExpression
{
    private readonly IExpression _leftExpression;
    private readonly IExpression _rightExpression;

    public Add(IExpression left, IExpression right)
    {
        _leftExpression = left;
        _rightExpression = right;
    }

    public int Interpret() => _leftExpression.Interpret() +
_rightExpression.Interpret();
}

// Usage
class Program
{
    static void Main()
    {
        IExpression number1 = new Number(5);
        IExpression number2 = new Number(10);
        IExpression addExpression = new Add(number1, number2);

        Console.WriteLine($"Result: {addExpression.Interpret()}");
// Outputs: Result: 15
    }
}
```

16. Iterator Pattern

Definition:
Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Use Case:
Iterating over a collection of items, like a list of products.

Example:

csharp
Copy code

```
// Iterator Interface
public interface IIterator<T>
{
    bool HasNext();
    T Next();
}

// Aggregate Interface
public interface IAggregate<T>
{
    IIterator<T> CreateIterator();
}

// Concrete Iterator
public class ProductIterator : IIterator<Product>
{
    private readonly ProductCollection _collection;
    private int _current = 0;

    public ProductIterator(ProductCollection collection) =>
_collection = collection;

    public bool HasNext() => _current < _collection.Count;

    public Product Next() => _collection[_current++];
}

// Product Class
public class Product
{
    public string Name { get; }
    public Product(string name) => Name = name;
}

// Concrete Aggregate
public class ProductCollection : IAggregate<Product>
{
    private readonly List<Product> _products = new List<Product>();

    public void Add(Product product) => _products.Add(product);
    public int Count => _products.Count;
    public Product this[int index] => _products[index];

    public IIterator<Product> CreateIterator() => new
ProductIterator(this);
}

// Usage
class Program
{
    static void Main()
    {
        var collection = new ProductCollection();
        collection.Add(new Product("Product 1"));
        collection.Add(new Product("Product 2"));
        collection.Add(new Product("Product 3"));

        var iterator = collection.CreateIterator();
        while (iterator.HasNext())
        {
            Console.WriteLine(iterator.Next().Name);
        }
    }
}
```

---

17. Mediator Pattern

Definition:
Defines an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly.

Use Case:
Managing interactions between different components in a chat application.

Example:

csharp
Copy code

```
// Mediator Interface
public interface IChatMediator
{
    void SendMessage(string message, User user);
    void RegisterUser(User user);
}

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

    public void RegisterUser(User user) => _users.Add(user);

    public void SendMessage(string message, User user)
    {
        foreach (var u in _users)
        {
            // Message should not be sent to the user who sent it
            if (u != user)
            {
                u.Receive(message);
            }
        }
    }
}

// Colleague
public class User
{
    private readonly string _name;
    private readonly IChatMediator _mediator;

    public User(string name, IChatMediator mediator)
    {
        _name = name;
        _mediator = mediator;
        _mediator.RegisterUser(this);
    }

    public void Send(string message) =>
_mediator.SendMessage(message, this);

    public void Receive(string message) =>
Console.WriteLine($"{_name} received: {message}");
}

// Usage
class Program
{
    static void Main()
    {
        var mediator = new ChatMediator();
        var user1 = new User("Alice", mediator);
        var user2 = new User("Bob", mediator);

        user1.Send("Hello Bob!");
        user2.Send("Hi Alice!");
    }
}
```

---

18. Memento Pattern

Definition:
Captures and externalizes an object's internal state without violating encapsulation, allowing the object to be restored to this state later.

Use Case:
Implementing an undo feature in a text editor.

Example:

csharp
Copy code

```
// Memento
public class TextMemento
{
    public string Text { get; }
    public TextMemento(string text) => Text = text;
}

// Originator
public class TextEditor
{
    private string _text;

    public void Write(string text) => _text = text;

    public TextMemento Save() => new TextMemento(_text);

    public void Restore(TextMemento memento) => _text =
memento.Text;

    public override string ToString() => _text;
}

// Caretaker
public class Caretaker
{
    private readonly Stack<TextMemento> _mementos = new
Stack<TextMemento>();

    public void Save(TextEditor editor) =>
_mementos.Push(editor.Save());
    public void Undo(TextEditor editor)
    {
        if (_mementos.Count > 0)
        {
            editor.Restore(_mementos.Pop());
        }
    }
}

// Usage
class Program
{
    static void Main()
    {
        var editor = new TextEditor();
        var caretaker = new Caretaker();

        editor.Write("Version 1");
        caretaker.Save(editor);

        editor.Write("Version 2");
        Console.WriteLine(editor); // Outputs: Version 2

        caretaker.Undo(editor);
        Console.WriteLine(editor); // Outputs: Version 1
    }
}
```

---

19. Observer Pattern

Definition:
Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Use Case:
Implementing a news feed where users are notified of new articles.

Example:

csharp
Copy code

```
// Subject Interface
public interface INewsPublisher
{
    void Subscribe(IObserver observer);
    void Unsubscribe(IObserver observer);
    void Notify(string news);
}

// Observer Interface
public interface IObserver
{
    void Update(string news);
}

// Concrete Subject
```

```
public class NewsPublisher : INewsPublisher
{
    private readonly List<IObserver> _observers = new
List<IObserver>();

    public void Subscribe(IObserver observer) =>
_observers.Add(observer);
    public void Unsubscribe(IObserver observer) =>
_observers.Remove(observer);

    public void Notify(string news)
    {
        foreach (var observer in _observers)
        {
            observer.Update(news);
        }
    }
}

// Concrete Observer
public class NewsSubscriber : IObserver
{
    private readonly string _name;

    public NewsSubscriber(string name) => _name = name;

    public void Update(string news) => Console.WriteLine($"{_name}
received news: {news}");
}

// Usage
class Program
{
    static void Main()
    {
        var publisher = new NewsPublisher();
        var subscriber1 = new NewsSubscriber("Alice");
        var subscriber2 = new NewsSubscriber("Bob");

        publisher.Subscribe(subscriber1);
        publisher.Subscribe(subscriber2);
        publisher.Notify("Breaking News: Observer Pattern in C#!");
    }
}
```

---

20. State Pattern

Definition:
Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Use Case:
Implementing a simple traffic light system where the behavior changes based on the light color.

Example:

csharp
Copy code

```
// State Interface
public interface ITrafficLightState
{
    void Change(TrafficLight light);
}

// Concrete States
public class RedState : ITrafficLightState
{
    public void Change(TrafficLight light)
    {
        Console.WriteLine("Red light - stop.");
        light.SetState(new GreenState());
    }
}

public class GreenState : ITrafficLightState
{
    public void Change(TrafficLight light)
    {
        Console.WriteLine("Green light - go.");
        light.SetState(new YellowState());
    }
}

public class YellowState : ITrafficLightState
{
    public void Change(TrafficLight light)
    {
        Console.WriteLine("Yellow light - caution.");
        light.SetState(new RedState());
    }
}

// Context
public class TrafficLight
{
    private ITrafficLightState _state;

    public TrafficLight()
    {
        _state = new RedState(); // Initial state
    }

    public void SetState(ITrafficLightState state) => _state =
state;

    public void Change() => _state.Change(this);
}

// Usage
class Program
{
    static void Main()
    {
        var light = new TrafficLight();

        for (int i = 0; i < 6; i++)
        {
            light.Change();
        }
    }
}
```

---

21. Strategy Pattern

Definition:
Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Use Case:
Sorting a collection of items using different sorting strategies.

Example:

csharp
Copy code

```
// Strategy Interface
public interface ISortStrategy
{
    void Sort(List<int> list);
}

// Concrete Strategies
public class BubbleSort : ISortStrategy
{
    public void Sort(List<int> list)
    {
        Console.WriteLine("Sorting using Bubble Sort");
        // Bubble sort implementation
    }
}

public class QuickSort : ISortStrategy
{
    public void Sort(List<int> list)
    {
        Console.WriteLine("Sorting using Quick Sort");
        // Quick sort implementation
    }
}

// Context
public class Sorter
{
    private ISortStrategy _strategy;
    public void SetStrategy(ISortStrategy strategy) => _strategy =
strategy;

    public void Sort(List<int> list) => _strategy.Sort(list);
}

// Usage
class Program
{
    static void Main()
    {
        var sorter = new Sorter();
        var numbers = new List<int> { 5, 2, 8, 3, 1 };

        sorter.SetStrategy(new BubbleSort());
        sorter.Sort(numbers);

        sorter.SetStrategy(new QuickSort());
        sorter.Sort(numbers);
    }
}
```

---

22. Template Method Pattern

Definition:
Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Use Case:
Creating a framework for a cooking recipe where each recipe has steps, but specific details can be implemented in subclasses.

Example:

csharp
Copy code

```
// Abstract Class
public abstract class Recipe
{
    public void Cook()
    {
        GatherIngredients();
        Prepare();
        CookMethod();
        Serve();
    }

    protected abstract void GatherIngredients();
    protected abstract void Prepare();
    protected abstract void CookMethod();

    private void Serve() => Console.WriteLine("Serving the dish.");
}

// Concrete Class
public class PastaRecipe : Recipe
{
    protected override void GatherIngredients()
    {
        Console.WriteLine("Gathering pasta, sauce, and cheese.");
    }

    protected override void Prepare()
    {
        Console.WriteLine("Boiling pasta and preparing sauce.");
    }

    protected override void CookMethod()
    {
        Console.WriteLine("Cooking pasta with sauce.");
    }
}

// Usage
class Program
{
    static void Main()
    {
        var pastaRecipe = new PastaRecipe();
        pastaRecipe.Cook();
    }
}
```

---

23. Visitor Pattern

Definition:
Separates an algorithm from the object structure on which it operates. Allows adding new operations to existing object structures without modifying them.

Use Case:
Calculating taxes for different product types in a shopping cart.

Example:

csharp
Copy code

```
// Visitor Interface
public interface IShoppingCartVisitor
{
    void Visit(Book book);
    void Visit(Fruit fruit);
}

// Element Interface
public interface IShoppingCartElement
{
    void Accept(IShoppingCartVisitor visitor);
}

// Concrete Elements
public class Book : IShoppingCartElement
{
    public double Price { get; }

    public Book(double price) => Price = price;

    public void Accept(IShoppingCartVisitor visitor) =>
visitor.Visit(this);
}

public class Fruit : IShoppingCartElement
{
    public double Price { get; }

    public Fruit(double price) => Price = price;
    public void Accept(IShoppingCartVisitor visitor) =>
visitor.Visit(this);
}

// Concrete Visitor
public class ShoppingCart : IShoppingCartVisitor
{
    private double _total;

    public void Visit(Book book) => _total += book.Price * 0.9; //
10% discount on books
    public void Visit(Fruit fruit) => _total += fruit.Price; // No
discount

    public double GetTotal() => _total;
}

// Usage
class Program
{
    static void Main()
    {
        var items = new List<IShoppingCartElement>
        {
            new Book(20),
            new Fruit(5)
        };

        var cart = new ShoppingCart();
        foreach (var item in items)
        {
            item.Accept(cart);
        }

        Console.WriteLine($"Total: {cart.GetTotal()}");
    }
}
```