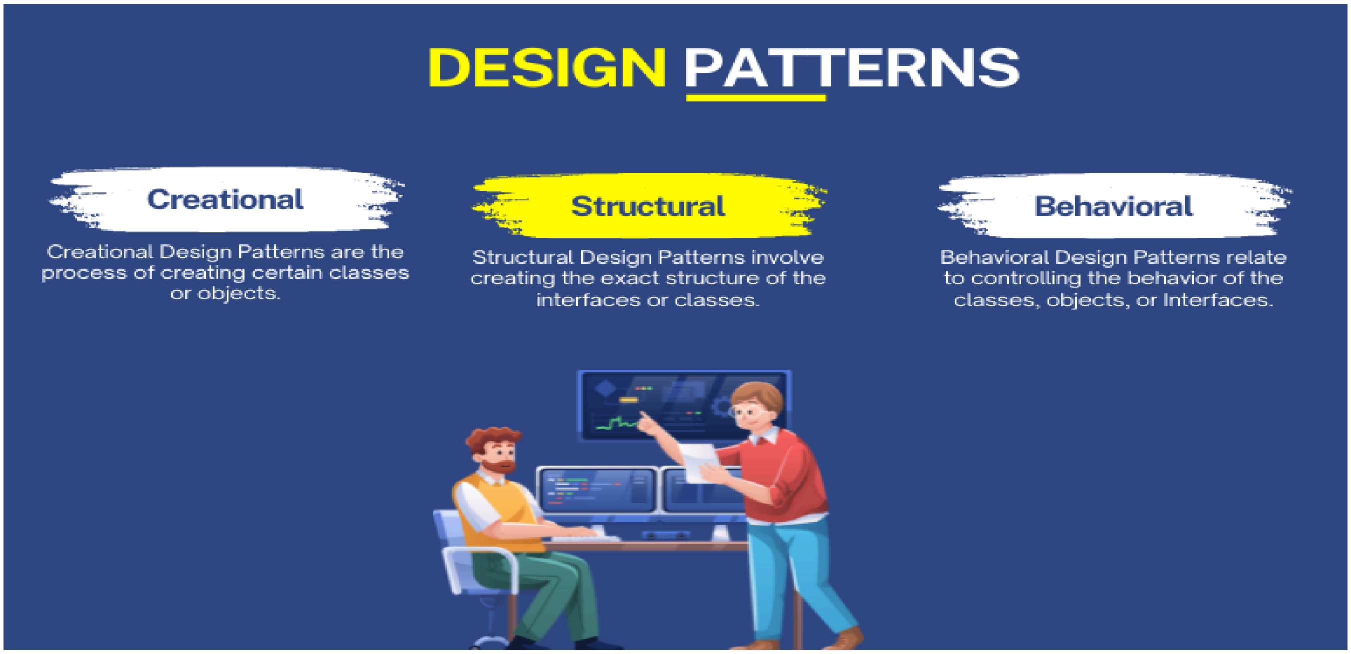
A Comprehensive Guide for Developers

Mastering Design Patterns in C#



Senior Software Engineer
Martin Kleinbooi

13-06-2024

Introduction

Importance of Design Patterns

Design patterns are proven solutions to common problems in software design. They provide a template for writing code that is easier to understand, maintain, and extend. Understanding and implementing design patterns can significantly improve your skills as a developer and enhance the quality of your software.

Overview of Design Patterns in C#

In this ebook, we will explore the three main categories of design patterns: Creational, Structural, and Behavioral. Each pattern will be explained in detail, with examples in C# to illustrate how they can be implemented in real-world scenarios.

Creational Patterns

- Singleton
- Factory
- Builder
- Prototype

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Conclusion

- Best Practices
- Further Reading

Creational Patterns

Singleton Pattern

What it does

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. This is particularly useful for managing shared resources like a logging service or a configuration manager.

Real-World Example: Configuration Manager

Imagine an application that requires a global configuration manager to access settings.

What the Code is Doing

- **Private Constructor:** Prevents external instantiation.
- **Static Instance Property:** Ensures only one instance is created, with thread safety ensured by a lock.
- **Configuration Settings:** Holds application-wide configuration settings.

```
public class ConfigurationManager
{
    private static ConfigurationManager instance = null;
    private static readonly object padlock = new object();

    public string DatabaseConnectionString { get; set; }

    private ConfigurationManager()
    {
        // Initialize configuration settings
        DatabaseConnectionString = "Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;";
    }

    public static ConfigurationManager Instance
    {
        get
        {
            lock (padlock)
            {
                if (instance == null)
                {
                    instance = new ConfigurationManager();
                }
                return instance;
            }
        }
    }
}
```

Factory Method Pattern

What It Does

The Factory Method pattern defines an interface for creating an object but lets subclasses alter the type of objects that will be created. This promotes loose coupling and scalability.

What the Code is Doing

1. **Interface Definition (`IDocument`):**
    - Defines a contract that all document types must adhere to by declaring a `Print()` method.
2. **Concrete Document Classes (`PdfDocument` and `WordDocument`):**
    - Implements the `IDocument` interface, each providing its own implementation of the `Print()` method specific to PDF and Word documents, respectively.
3. **DocumentFactory Class:**
    - Acts as a Factory class responsible for creating instances of `IDocument` based on a provided `DocumentType` enum.

Insights

- **Interface Segregation:** `IDocument` interface segregates the behavior of documents by enforcing a common method signature (`Print()`), allowing diverse document types to be managed uniformly through a single interface.
- **Polymorphic Behavior:** `PdfDocument` and `WordDocument` exhibit polymorphism, where different implementations of the same interface (`IDocument`) are invoked through a unified interface without client code being aware of the specific document type.
- **Factory Method Pattern:** `DocumentFactory` encapsulates the object creation process (`CreateDocument()` method) based on a discriminating parameter (`DocumentType`), abstracting the instantiation logic from client code and promoting flexibility in adding new document types.

This structure ensures that the code remains modular, extensible, and easy to maintain by adhering to object-oriented design principles like abstraction, encapsulation, and polymorphism.

```
3 references
public interface IDocument
{
    2 references
    void Print();
}
1 reference
public class PdfDocument : IDocument
{
    1 reference
    public void Print()
    {
        Console.WriteLine("Printing PDF document...");
    }
}

1 reference
public class WordDocument : IDocument
{
    1 reference
    public void Print()
    {
        Console.WriteLine("Printing Word document...");
    }
}
0 references
public static class DocumentFactory
{
    0 references
    public static IDocument CreateDocument(DocumentType type)
    {
        return type switch
        {
            DocumentType.Word => new WordDocument(),
            DocumentType.Pdf => new PdfDocument(),
            _ => throw new ArgumentException("Invalid document type"),
        };
    }
}
```

Builder Pattern

What It Does

The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

What the Code is Doing

1. **Computer Class:**
    - Represents a `Computer` object with properties such as `CPU`, `GPU`, `RAM`, `Storage`, and `OS`.
    - Overrides the `ToString()` method to provide a formatted string representation of the computer's specifications.
2. **ComputerBuilder Class:**
    - Implements the Builder pattern to construct `Computer` objects step-by-step with customizable configurations.
    - Initializes a `Computer` instance in its constructor and allows setting each component (`CPU`, `GPU`, `RAM`, `Storage`, `OS`) through fluent methods (`SetCPU()`, `SetGPU()`, `SetRAM()`, `SetStorage()`, `SetOS()`).
    - Returns the fully configured `Computer` object using the `Build()` method.

Insights

- **Separation of Concerns:** The `Computer` class focuses on representing a computer's state, while the `ComputerBuilder` class encapsulates the construction process, ensuring that client code remains decoupled from the object creation details.
- **Fluent Interface:** Using fluent methods (`SetCPU().SetGPU()...`) enhances readability and provides a straightforward way to chain method calls for configuring a `Computer` object.
- **Immutable Object Construction:** The `Computer` object is effectively immutable during construction due to the absence of direct public setters for its properties. This promotes thread safety and ensures that once constructed, a `Computer` instance's state remains consistent.

This approach facilitates flexible and scalable object creation, allowing for easy modification and extension of the `Computer` class's construction process without altering its core implementation.

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

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

```
7 references
public class PizzaBuilder
{
    private Pizza _pizza;

    2 references
    public PizzaBuilder()
    {
        _pizza = new Pizza();
    }

    2 references
    public PizzaBuilder SetDough(string dough)
    {
        _pizza.Dough = dough;
        return this;
    }

    2 references
    public PizzaBuilder SetSauce(string sauce)
    {
        _pizza.Sauce = sauce;
        return this;
    }

    2 references
    public PizzaBuilder SetCheese(string cheese)
    {
        _pizza.Cheese = cheese;
        return this;
    }

    6 references
    public PizzaBuilder AddTopping(string topping)
    {
        _pizza.Toppings.Add(topping);
        return this;
    }

    2 references
    public Pizza Build()
    {
        return _pizza;
    }
}
}
```

Prototype Pattern

What it does

The Prototype pattern allows for the creation of new objects by copying an existing object, known as the prototype. This is useful for creating new instances that are identical or similar to an existing instance.

Real-World Example: Document Cloning

Consider a document management system where users can duplicate documents to create new ones.

What the Code is Doing

- **DocumentPrototype Class:** Defines the `Clone` method for creating a copy.
- **Concrete Prototypes:** Implement the `Clone` method to duplicate specific document types.

```
7 references
public abstract class DocumentPrototype
{
    2 references
    public abstract DocumentPrototype Clone();
}

0 references
public class Report : DocumentPrototype
{
    0 references
    public string Title { get; set; }
    0 references
    public string Content { get; set; }

    1 reference
    public override DocumentPrototype Clone()
    {
        return (DocumentPrototype)this.MemberwiseClone();
    }
}

0 references
public class Invoice : DocumentPrototype
{
    0 references
    public string InvoiceNumber { get; set; }
    0 references
    public decimal Amount { get; set; }

    1 reference
    public override DocumentPrototype Clone()
    {
        return (DocumentPrototype)this.MemberwiseClone();
    }
}
```

Structural Patterns

Adapter Pattern

What it does

The Adapter pattern allows incompatible interfaces to work together. It acts as a bridge between two interfaces, enabling classes to collaborate that otherwise couldn't.

Real-World Example: Payment Gateway Adapter

Imagine an e-commerce application that needs to integrate multiple payment gateways with different interfaces.

What the Code is Doing

- **IPaymentGateway Interface:** Standardizes the payment process method.
- **PayPal Class:** Implements a different interface for payment.
- **PayPalAdapter:** Adapts the `PayPal` interface to match `IPaymentGateway`.

```
1 reference
public interface IPaymentGateway
{
    1 reference
    void ProcessPayment(decimal amount);
}

2 references
public class PayPal
{
    1 reference
    public void SendPayment(decimal amount)
    {
        Console.WriteLine($"Processing PayPal payment of {amount}");
    }
}

1 reference
public class PayPalAdapter : IPaymentGateway
{
    2 references
    private readonly PayPal _payPal;

    0 references
    public PayPalAdapter(PayPal payPal)
    {
        _payPal = payPal;
    }

    1 reference
    public void ProcessPayment(decimal amount)
    {
        _payPal.SendPayment(amount);
    }
}

```

Bridge Pattern

What it does

The Bridge pattern separates an object's abstraction from its implementation, allowing the two to vary independently. This is useful for handling complexity and enabling flexibility.

Real-World Example: Device Remote Control

Consider a remote control that can operate different types of devices (e.g., TV, radio) with various brands.

What the Code is Doing

- **IDevice Interface:** Defines device operations.
- **Concrete Devices:** Implement the device operations.
- **RemoteControl Class:** Abstracts the device operations and controls device power.
- **BasicRemoteControl:** Extends `RemoteControl` to toggle device power.

```csharp
5 references
public interface IDevice
{
    3 references
    void On();
    3 references
    void Off();
}

0 references
public class TV : IDevice
{
    2 references
    public void On() => Console.WriteLine("TV is On");
    2 references
    public void Off() => Console.WriteLine("TV is Off");
}

0 references
public class Radio : IDevice
{
    2 references
    public void On() => Console.WriteLine("Radio is On");
    2 references
    public void Off() => Console.WriteLine("Radio is Off");
}
```

```
3 references
public abstract class RemoteControl
{
    3 references
    protected IDevice _device;

    1 reference
    public RemoteControl(IDevice device)
    {
        _device = device;
    }

    1 reference
    public abstract void TogglePower();
}

1 reference
public class BasicRemoteControl : RemoteControl
{
    3 references
    private bool _isOn = false;

    0 references
    public BasicRemoteControl(IDevice device) : base(device) { }

    1 reference
    public override void TogglePower()
    {
        if (_isOn)
        {
            _device.Off();
            _isOn = false;
        }
        else
        {
            _device.On();
            _isOn = true;
        }
    }
}
```

Composite Pattern

What it does

The Composite pattern allows you to compose objects into tree structures to represent part-whole hierarchies. This lets clients treat individual objects and compositions uniformly.

Real-World Example: File System

Consider a file system where files and directories are both treated as nodes.

What the Code is Doing

- **FileSystemNode Class:** Abstracts a file system node with a `Display` method.
- **File Class:** Represents a file and implements the `Display` method.
- **Directory Class:** Represents a directory and can contain child nodes, implementing the `Display` method recursively.

```
6 references
public abstract class FileSystemNode
{
    4 references
    public string Name { get; set; }
    2 references
    public abstract void Display(int depth);
}

1 reference
public class File : FileSystemNode
{
    0 references
    public File(string name) => Name = name;

    1 reference
    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + Name);
    }
}

1 reference
public class Directory : FileSystemNode
{
    3 references
    private List<FileSystemNode> _children = new List<FileSystemNode>();

    0 references
    public Directory(string name) => Name = name;

    0 references
    public void Add(FileSystemNode node) => _children.Add(node);
    0 references
    public void Remove(FileSystemNode node) => _children.Remove(node);

    1 reference
    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + Name);
        foreach (var child in _children)
        {
            child.Display(depth + 2);
        }
    }
}
```

Decorator Pattern

What it does

The Decorator pattern allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class.

Real-World Example: Notification System

Imagine a notification system where notifications can be sent via email, SMS, or both.

What the Code is Doing

- **INotifier Interface:** Defines the `Send` method.
- **EmailNotifier:** Implements the `Send` method for email.
- **NotifierDecorator:** Abstract decorator that extends `INotifier`.
- **SMSNotifier:** Concrete decorator that adds SMS functionality.

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

0 references
public class EmailNotifier : INotifier
{
    2 references
    public void Send(string message)
    {
        Console.WriteLine($"Sending Email: {message}");
    }
}

3 references
public abstract class NotifierDecorator : INotifier
{
    2 references
    protected INotifier _notifier;

    1 reference
    public NotifierDecorator(INotifier notifier)
    {
        _notifier = notifier;
    }

    4 references
    public virtual void Send(string message)
    {
        _notifier.Send(message);
    }
}
```

```
public class SMSNotifier : NotifierDecorator
{
    public SMSNotifier(INotifier notifier) : base(notifier) { }

    public override void Send(string message)
    {
        base.Send(message);
        Console.WriteLine($"Sending SMS: {message}");
    }
}
```

Facade Pattern

What it does

The Facade pattern provides a simplified interface to a complex subsystem. This makes the subsystem easier to use and reduces dependencies on the inner workings.

Real-World Example: Home Theater System

Consider a home theater system with multiple components like DVD player, sound system, and projector.

What the Code is Doing

- **Component Classes:** Represent parts of the home theater system.
- **HomeTheaterFacade:** Simplifies the interaction with the subsystem components by providing a `WatchMovie` method.

```csharp
2 references
public class DVDPlayer
{
    1 reference
    public void On() => Console.WriteLine("DVD Player On");
    1 reference
    public void Play() => Console.WriteLine("DVD Player Play");
}

2 references
public class SoundSystem
{
    1 reference
    public void On() => Console.WriteLine("Sound System On");
    1 reference
    public void SetVolume(int level) => Console.WriteLine($"Setting volume to {level}");
}

2 references
public class Projector
{
    1 reference
    public void On() => Console.WriteLine("Projector On");
    1 reference
    public void SetInput(string input) => Console.WriteLine($"Setting input to {input}");
}

```

```csharp
1 reference
public class HomeTheaterFacade
{
    3 references
    private DVDPlayer _dvdPlayer;
    3 references
    private SoundSystem _soundSystem;
    3 references
    private Projector _projector;

    0 references
    public HomeTheaterFacade(DVDPlayer dvdPlayer, SoundSystem soundSystem, Projector projector)
    {
        _dvdPlayer = dvdPlayer;
        _soundSystem = soundSystem;
        _projector = projector;
    }

    0 references
    public void WatchMovie()
    {
        _dvdPlayer.On();
        _soundSystem.On();
        _soundSystem.SetVolume(10);
        _projector.On();
        _projector.SetInput("DVD");
        _dvdPlayer.Play();
    }
}
```

Flyweight Pattern

What it does

The Flyweight pattern reduces the memory footprint by sharing as much data as possible with other similar objects. It is useful when a large number of objects need to be created and they share some common state.

Real-World Example: Text Formatting

Consider a text editor that needs to format large amounts of text, with many characters sharing the same formatting.

What the Code is Doing

- **TextFormat Class:** Represents shared formatting information.
- **Character Class:** Represents a character with a symbol and text format.
- **TextFormatFactory:** Manages the shared text formats, ensuring formats are reused.

```csharp
public class TextFormat
{
    public string Font { get; set; }
    public int Size { get; set; }
    public string Color { get; set; }
}

public class Character
{
    public char Symbol { get; set; }
    public TextFormat Format { get; set; }

    public Character(char symbol, TextFormat format)
    {
        Symbol = symbol;
        Format = format;
    }
}

public class TextFormatFactory
{
    private Dictionary<string, TextFormat> _formats = new Dictionary<string, TextFormat>();

    public TextFormat GetFormat(string font, int size, string color)
    {
        string key = $"{font}-{size}-{color}";
        if (!_formats.ContainsKey(key))
        {
            _formats[key] = new TextFormat { Font = font, Size = size, Color = color };
        }
        return _formats[key];
    }
}
```

Proxy Pattern

What it does

The Proxy pattern provides a surrogate or placeholder for another object to control access to it. This can be used for lazy initialization, access control, logging, and more.

Real-World Example: Image Viewer

Imagine an image viewer application where loading images is resource-intensive, and you want to load them only when they are actually needed.

What the Code is Doing

- **IImage Interface:** Defines the `Display` method.
- **RealImage Class:** Implements the `Display` method and loads the image.
- **ProxyImage Class:** Controls access to `RealImage`, deferring image loading until necessary.

```
2 references
public interface IImage
{
    3 references
    void Display();
}

3 references
public class RealImage : IImage
{
    3 references
    private string _filename;

    1 reference
    public RealImage(string filename)
    {
        _filename = filename;
        LoadImage();
    }

    1 reference
    private void LoadImage()
    {
        Console.WriteLine($"Loading image {_filename}");
    }

    2 references
    public void Display()
    {
        Console.WriteLine($"Displaying image {_filename}");
    }
}
```

```
public class ProxyImage : IImage
{
    private RealImage _realImage;
    private string _filename;

    public ProxyImage(string filename)
    {
        _filename = filename;
    }

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

Behavioral Patterns

Chain of Responsibility Pattern

What it does

The Chain of Responsibility pattern allows an object to pass a request along a chain of potential handlers until the request is handled. This is useful for scenarios where multiple objects might handle a request.

Real-World Example: Support Ticket System

Consider a support ticket system where tickets can be handled by different support levels.

What the Code is Doing

- **SupportHandler Class:** Abstract handler defining the `HandleRequest` method and a link to the next handler.
- **LevelOneSupport and LevelTwoSupport:** Concrete handlers that handle specific types of issues and pass others along the chain.

```
4 references
public abstract class SupportHandler
{
    5 references
    protected SupportHandler _nextHandler;

    0 references
    public void SetNextHandler(SupportHandler nextHandler)
    {
        _nextHandler = nextHandler;
    }

    4 references
    public abstract void HandleRequest(string issue);
}

0 references
public class LevelOneSupport : SupportHandler
{
    3 references
    public override void HandleRequest(string issue)
    {
        if (issue == "Simple Issue")
        {
            Console.WriteLine("LevelOneSupport handled the issue.");
        }
        else if (_nextHandler != null)
        {
            _nextHandler.HandleRequest(issue);
        }
    }
}

0 references
public class LevelTwoSupport : SupportHandler
{
    3 references
    public override void HandleRequest(string issue)
    {
        if (issue == "Complex Issue")
        {
            Console.WriteLine("LevelTwoSupport handled the issue.");
        }
        else if (_nextHandler != null)
        {
            _nextHandler.HandleRequest(issue);
        }
    }
}
```

Command Pattern

What it does

The Command pattern encapsulates a request as an object, allowing you to parameterize clients with queues, requests, and operations. This is useful for implementing undo/redo functionality and task queues.

Real-World Example: Smart Home System

Imagine a smart home system where you can control various devices with commands.

What the Code is Doing

- **ICommand Interface:** Defines the `Execute` method.
- **Light Class:** Represents a light with `On` and `Off` methods.
- **LightOnCommand and LightOffCommand:** Implement `ICommand` to encapsulate the light operations.
- **RemoteControl Class:** Uses commands to control the light.

```
4 references
public interface ICommand
{
    3 references
    void Execute();
}

4 references
public class Light
{
    1 reference
    public void On() => Console.WriteLine("Light is On");
    1 reference
    public void Off() => Console.WriteLine("Light is Off");
}

1 reference
public class LightOnCommand : ICommand
{
    2 references
    private Light _light;

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

    2 references
    public void Execute()
    {
        _light.On();
    }
}

1 reference
public class LightOffCommand : ICommand
{
    2 references
    private Light _light;

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

    2 references
    public void Execute()
    {
        _light.Off();
    }
}

0 references
```

```
public class RemoteControl
{
    private ICommand _command;

    public void SetCommand(ICommand command)
    {
        _command = command;
    }

    public void PressButton()
    {
        _command.Execute();
    }
}
```

Iterator Pattern

What it does

The Iterator pattern provides a way to access elements of a collection sequentially without exposing the underlying representation. It is useful for iterating over various data structures in a uniform way.

Real-World Example: Social Network Iterator

Imagine a social network where you need to iterate over a user's friends list.

What the Code is Doing

- **IIterator Interface:** Defines methods for iterating over a collection.
- **FriendIterator Class:** Implements `IIterator` to iterate over a user's friends list.
- **User Class:** Manages a list of friends and creates an iterator.

```
2 references
public interface IIterator<T>
{
    1 reference
    bool HasNext();
    1 reference
    T Next();
}

2 references
public class FriendIterator : IIterator<string>
{
    3 references
    private List<string> _friends;
    3 references
    private int _position;

    1 reference
    public FriendIterator(List<string> friends)
    {
        _friends = friends;
        _position = 0;
    }

    1 reference
    public bool HasNext()
    {
        return _position < _friends.Count;
    }

    1 reference
    public string Next()
    {
        return _friends[_position++];
    }
}

0 references
public class User
{
    2 references
    public List<string> Friends { get; private set; } = new List<string>();

    0 references
    public void AddFriend(string friend)
    {
        Friends.Add(friend);
    }

    0 references
    public IIterator<string> CreateIterator()
    {
        return new FriendIterator(Friends);
    }
}
```

Mediator Pattern

What it does

The Mediator pattern defines an object that encapsulates how a set of objects interact. It promotes loose coupling by preventing objects from referring to each other explicitly.

Real-World Example: Chat Room

Consider a chat room where users can send messages to each other via a mediator.

What the Code is Doing

- **User Class:** Represents a user with a name and a method to send messages.
- **IChatRoom Interface:** Defines the method for showing messages.
- **ChatRoom Class:** Implements `IChatRoom` to display messages.

```
4 references
public class User
{
    2 references
    public string Name { get; set; }
    2 references
    private IChatRoom _chatRoom;

    0 references
    public User(string name, IChatRoom chatRoom)
    {
        Name = name;
        _chatRoom = chatRoom;
    }

    0 references
    public void Send(string message)
    {
        _chatRoom.ShowMessage(this, message);
    }
}

3 references
public interface IChatRoom
{
    2 references
    void ShowMessage(User user, string message);
}

0 references
public class ChatRoom : IChatRoom
{
    2 references
    public void ShowMessage(User user, string message)
    {
        Console.WriteLine($"{DateTime.Now} [{user.Name}]: {message}");
    }
}

```

Memento Pattern

What it does

The Memento pattern provides a way to capture and restore an object's state. It is useful for implementing undo mechanisms.

Real-World Example: Text Editor

Consider a text editor that can save and restore the state of a document.

What the Code is Doing

- **Document Class:** Represents a document with content and methods to save and restore its state.
- **DocumentMemento Class:** Captures the state of the document.

```csharp
public class Document
{
    public string Content { get; set; }

    public DocumentMemento Save()
    {
        return new DocumentMemento(Content);
    }

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

public class DocumentMemento
{
    public string Content { get; private set; }

    public DocumentMemento(string content)
    {
        Content = content;
    }
}
```

Observer Pattern

What it does

The 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. This is useful for implementing event handling systems.

Real-World Example: Stock Price Monitoring

Imagine a stock price monitoring system where multiple clients need to be updated when a stock price changes.

What the Code is Doing

- **Stock Class:** Represents a stock with a symbol and price, and manages observers.
- **IObserver Interface:** Defines the `Update` method for observers.
- **StockObserver Class:** Implements `IObserver` to react to stock price changes.

```
1 reference
public class Stock
{
    2 references
    private string _symbol;
    3 references
    private decimal _price;
    3 references
    private List<IObserver> _observers = new List<IObserver>();

    0 references
    public Stock(string symbol, decimal price)
    {
        _symbol = symbol;
        _price = price;
    }

    0 references
    public void AddObserver(IObserver observer)
    {
        _observers.Add(observer);
    }

    0 references
    public void RemoveObserver(IObserver observer)
    {
        _observers.Remove(observer);
    }

    0 references
    public void SetPrice(decimal price)
    {
        _price = price;
        NotifyObservers();
    }

    1 reference
    private void NotifyObservers()
    {
        foreach (var observer in _observers)
        {
            observer.Update(_symbol, _price);
        }
    }
}
```

```
5 references
public interface IObserver
{
    1 reference
    void Update(string symbol, decimal price);
}

1 reference
public class StockObserver : IObserver
{
    2 references
    private string _name;

    0 references
    public StockObserver(string name)
    {
        _name = name;
    }

    1 reference
    public void Update(string symbol, decimal price)
    {
        Console.WriteLine($"Observer {_name}: {symbol} price changed to {price}");
    }
}
```

State Pattern

What it does

The State pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class. This is useful for objects that need to change behavior based on state.

Real-World Example: Traffic Light

Consider a traffic light system where the light changes its behavior based on its current state (red, yellow, green).

What the Code is Doing

- **ITrafficLightState Interface:** Defines the `Handle` method.
- **RedLightState, YellowLightState, GreenLightState:** Implement `ITrafficLightState` for different light states.
- **TrafficLight Class:** Manages the current state and changes behavior based on state.

```
1 reference
public class RedLightState : ITrafficLightState
{
    2 references
    public void Handle(TrafficLight trafficLight)
    {
        Console.WriteLine("Red Light - Stop");
        trafficLight.SetState(new GreenLightState());
    }
}

1 reference
public class YellowLightState : ITrafficLightState
{
    2 references
    public void Handle(TrafficLight trafficLight)
    {
        Console.WriteLine("Yellow Light - Prepare to Stop");
        trafficLight.SetState(new RedLightState());
    }
}

1 reference
public class GreenLightState : ITrafficLightState
{
    2 references
    public void Handle(TrafficLight trafficLight)
    {
        Console.WriteLine("Green Light - Go");
        trafficLight.SetState(new YellowLightState());
    }
}
```

```
public class TrafficLight
{
    private ITrafficLightState _state;

    public TrafficLight(ITrafficLightState state)
    {
        _state = state;
    }

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

    public void Change()
    {
        _state.Handle(this);
    }
}
```

Strategy Pattern

What it does

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows the algorithm to vary independently from clients that use it.

Real-World Example: Payment Processing

Imagine a payment processing system where you can choose different payment methods (credit card, PayPal, etc.).

What the Code is Doing

- **IPaymentStrategy Interface:** Defines the `Pay` method.
- **CreditCardPayment and PayPalPayment:** Implement `IPaymentStrategy` for different payment methods.
- **PaymentContext Class:** Uses a strategy to process payment.

```
4 references
public interface IPaymentStrategy
{
    3 references
    void Pay(decimal amount);
}

0 references
public class CreditCardPayment : IPaymentStrategy
{
    2 references
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount} using Credit Card.");
    }
}

0 references
public class PayPalPayment : IPaymentStrategy
{
    2 references
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount} using PayPal.");
    }
}

0 references
public class PaymentContext
{
    2 references
    private IPaymentStrategy _paymentStrategy;

    0 references
    public void SetPaymentStrategy(IPaymentStrategy paymentStrategy)
    {
        _paymentStrategy = paymentStrategy;
    }

    0 references
    public void Pay(decimal amount)
    {
        _paymentStrategy.Pay(amount);
    }
}
```

Template Method Pattern

What it does

The Template Method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. This allows subclasses to redefine certain steps of an algorithm without changing its structure.

Real-World Example: Data Processing

Consider a data processing system where different types of data are processed in a similar way, with some specific steps differing.

What the Code is Doing

- **DataProcessor Class:** Defines the template method `ProcessData` and abstract methods for specific steps.
- **CsvDataProcessor and JsonDataProcessor:** Implement specific steps for different data types.

```
public abstract class DataProcessor
{
    public void ProcessData()
    {
        ReadData();
        Process();
        SaveData();
    }

    protected abstract void ReadData();
    protected abstract void Process();
    protected abstract void SaveData();
}

public class CsvDataProcessor : DataProcessor
{
    protected override void ReadData()
    {
        Console.WriteLine("Reading CSV data");
    }

    protected override void Process()
    {
        Console.WriteLine("Processing CSV data");
    }

    protected override void SaveData()
    {
        Console.WriteLine("Saving CSV data");
    }
}
```

```
0 references
public class JsonDataProcessor : DataProcessor
{
    2 references
    protected override void ReadData()
    {
        Console.WriteLine("Reading JSON data");
    }

    2 references
    protected override void Process()
    {
        Console.WriteLine("Processing JSON data");
    }

    2 references
    protected override void SaveData()
    {
        Console.WriteLine("Saving JSON data");
    }
}
```

Visitor Pattern

What it does

The Visitor pattern allows you to add further operations to objects without having to modify them. This is useful for performing operations across a heterogeneous collection of objects.

Real-World Example: Employee Structure

Imagine a company with a hierarchy of employees, and you want to perform operations like calculating salaries or generating reports.

What the Code is Doing

- **IEmployee Interface:** Defines the `Accept` method for visitors.
- **Manager and Developer Classes:** Implement `IEmployee` and accept visitors.
- **IVisitor Interface:** Defines methods for visiting different types of employees.
- **SalaryCalculator Class:** Implements `IVisitor` to calculate salaries.

```
2 references
public interface IEmployee
{
    2 references
    void Accept(IVisitor visitor);
}

2 references
public class Manager : IEmployee
{
    1 reference
    public void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

2 references
public class Developer : IEmployee
{
    1 reference
    public void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

4 references
public interface IVisitor
{
    2 references
    void Visit(Manager manager);
    2 references
    void Visit(Developer developer);
}

0 references
public class SalaryCalculator : IVisitor
{
    2 references
    public void Visit(Manager manager)
    {
        Console.WriteLine("Calculating salary for manager");
    }

    2 references
    public void Visit(Developer developer)
    {
        Console.WriteLine("Calculating salary for developer");
    }
}
```

Conclusion

In conclusion, design patterns are essential tools for software developers to solve recurring design problems effectively. By leveraging these patterns, developers can improve code maintainability, scalability, and flexibility. Creational patterns like Singleton, Factory Method, Abstract Factory, Builder, and Prototype help manage object creation processes efficiently. Structural patterns such as Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy facilitate the composition of objects and subsystems, enhancing code organization and reusability. Behavioral patterns like Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor offer solutions for managing communication between objects, defining algorithms, and handling state changes.

These patterns are not just theoretical constructs but proven solutions derived from years of software engineering experience. They embody best practices in software design, promoting separation of concerns, encapsulation, and code reusability. By adopting design patterns, developers can write cleaner, more maintainable code that is easier to extend and refactor.

Best Practices

When applying design patterns, it's crucial to adhere to best practices to maximize their benefits:

1. **Understand the Problem Context:** Before applying a pattern, thoroughly understand the problem context and ensure the pattern addresses the specific design issue.
2. **Follow Naming Conventions:** Use standard naming conventions for classes, methods, and variables to make your code more readable and understandable.
3. **Document Usage:** Document where and why you are applying a pattern in your codebase to help future developers understand the design decisions.
4. **Test Thoroughly:** Design patterns alter the structure of your code; ensure comprehensive testing to validate that patterns are correctly implemented and do not introduce unintended side effects.
5. **Keep Patterns Simple:** Avoid over-engineering by applying patterns only when they are appropriate and necessary. Simplicity should be prioritized over complexity.
6. **Refactor Regularly:** Patterns can evolve over time as requirements change. Be prepared to refactor code to ensure patterns continue to align with evolving needs.

Further Reading

To deepen your understanding and application of design patterns, consider exploring these additional resources:

1. **Books:**
   - "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (the Gang of Four).
   - "Head First Design Patterns" by Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra.
2. **Online Resources:**
   - Refactoring Guru (https://refactoring.guru/design-patterns): Provides comprehensive explanations and examples of various design patterns.
   - Dofactory (https://www.dofactory.com/net/design-patterns): Offers tutorials, examples, and a design pattern framework for .NET developers.
3. **Courses:**
   - Coursera and edX offer courses on software design patterns taught by university professors and industry experts.
   - Udemy has practical courses that teach design patterns with real-world examples and hands-on exercises. - https://www.udemy.com/course/complete-csharp-masterclass-go-from-zero-to-coding-hero/?couponCode=CODEHUB
4. **Community and Forums:**
   - Join online forums like StackOverflow and Reddit where developers discuss design patterns, share experiences, and provide insights.
   - Participate in local meetups or conferences focused on software engineering and design principles.

By continually learning and applying design patterns in your projects, you can enhance your software development skills and contribute to building more robust and scalable applications.