# DESIGN PATTERNS IN USE

Learn the most important Design patterns with real-life examples in C#

**Dr Milan Milanovic**
Microsoft MVP

TechWorld WithMilan
simplifying complex topics

# Table Of Contents

## What are Design Patterns?

The concept of design patterns in software engineering was popularized in the early 1990s by the famous book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, collectively known as the "Gang of Four" (GoF). However, the roots of design patterns go back further, drawing inspiration from the field of architecture.

Architect Christopher Alexander introduced the idea of patterns in architecture in his 1977 book "A Pattern Language," where he described solutions to common design problems in urban planning and building architecture. Alexander's work emphasized that each pattern solves a specific problem and is part of a more extensive design system. This approach resonated with software developers facing similar challenges in software construction.

Seeing the potential of Alexander's concepts in software development, the GoF adapted and expanded these ideas to object-oriented programming. Their book **introduced 23 design patterns categorized into Creational, Structural, and Behavioral patterns**, providing a standardized approach to solving common software design issues.

In software development, design patterns serve a similar purpose—**they provide templated solutions to recurring problems**, ensuring that you don't have to reinvent the wheel each time you encounter a familiar issue.

> *This book is accompanied by a live **GitHub repo** with all examples*

## Design Pattern Types

Design patterns can be separated into three main categories:

### Creational Patterns - Object creation

- **Singleton Pattern**: Ensures a class has only one instance and provides a global point of access to it. This is particularly useful for managing resources like database connections.

- **Factory Method Pattern**: Defines an interface for creating an object but lets subclasses alter the type of objects that will be created. It's ideal when you have a superclass with multiple subclasses, and must create an instance of one of these subclasses based on some initialization parameters.

- **Abstract Factory Pattern**: Offers an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern shines when you must ensure that the created objects can work together without knowing their exact types.

- **Builder Pattern**: Separates the construction of a complex object from its representation, allowing the same construction process to create various representations. This pattern is excellent for when you need to create an object with many optional or required components.

- **Prototype Pattern**: Creates new objects by copying an existing object, known as the prototype. This is particularly useful in scenarios where the cost of creating an object is heavier than copying an existing one.

## Structural Patterns - Object assembly

- **Adapter Pattern**: Allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, enabling them to communicate without changing their existing code. This pattern is perfect when integrating new features or libraries without disrupting existing code.

- **Composite Pattern**: Enables you to treat individual objects and compositions of objects uniformly. It's ideal for representing part-whole hierarchies where you want to ignore the difference between compositions of objects and individual objects.

- **Proxy Pattern**: Provides a placeholder for another object to control access to it. This is useful for lazy loading, controlling access, or logging, acting as an intermediary between the client and the actual object to add a processing layer.

- **Flyweight Pattern**: Minimizes memory use by sharing as much data as possible with similar objects; it's a boon for efficiency when working with many objects with some shared state.

- **Facade Pattern**: Offers a simplified interface to a complex system of classes, library, or framework. Providing a higher-level interface makes the subsystem easier to use, reducing complexity and promoting decoupling.

- **Bridge Pattern**: Decouples an abstraction from its implementation so that the two can vary independently. It's particularly useful when extending a class in several orthogonal (independent) dimensions.

- **Decorator Pattern**: Allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. This pattern provides a flexible alternative to subclassing for extending functionality.

## Behavioral Patterns - Object interactions

- **Strategy Pattern**: Allows you to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. This pattern is perfect when you have multiple ways to accomplish a task and you want to select the method at runtime.

- **Observer Pattern**: Defines a dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically. It's ideal for implementing distributed event handling systems, where changing one object's state needs to be reflected in another.

- **Command Pattern**: Turns a request into a stand-alone object containing all the request information. This transformation allows you to parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.

- **Iterator Pattern**: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. This pattern is useful for collections of objects to provide a standard way to traverse them and potentially access a selection of elements without exposing the internal structure.

- **State Pattern**: An object can alter its behavior when its internal state changes. The object will appear to change its class. This is beneficial when an object's behavior depends on its state and must be able to change its behavior at runtime depending on that state.

- **Memento Pattern**: Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later. This pattern is useful for implementing undo mechanisms or saving and restoring an object's state.

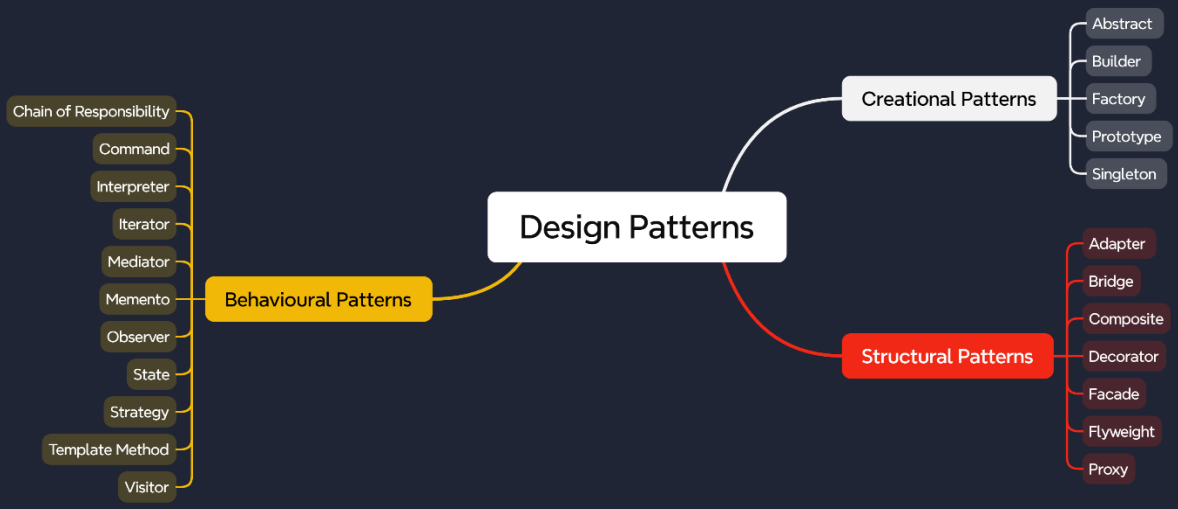
- **Mediator Pattern**: Reduces chaos between interacting classes by encapsulating how disparate sets of objects interact and communicate. Doing so helps prevent the "spaghetti code" scenario where multiple classes communicate directly and in a complex way.

- **Chain of Responsibility Pattern**: Passes the request along a chain of handlers. Upon receiving a request, each handler decides to process the request or pass it to the next handler in the chain. It's particularly useful for processing multiple requests in a decentralized manner.

- **Visitor Pattern**: Let you define a new operation without changing the classes of the elements on which it operates. Ideal for scenarios where you need to perform operations across a group of objects with different classes.

- **Interpreter Pattern**: Provides a way to evaluate language grammar or expression. This is useful in developing tools and compilers for new programming or scripting languages.

- **Template Method Pattern**: Defines the skeleton of an algorithm in the superclass but lets subclasses override specific algorithm steps without changing its structure. It's beneficial when a multi-step process requires flexibility while maintaining the overall structure.




*These Design patterns are not the only kind of design patterns we have. To learn more about other types of design patterns, check here.*

## Don’t fall into the Design Patterns trap

You should be warned that you will probably fall into the **Design Patterns trap** when you first learn design patterns. This means you will try to squeeze a pattern in every solution, and your codebase will become over-engineering and unusable very soon.

But we want to make our codebase as simple as possible, so Design patterns are not the silver bullet for all problems. We should not try to put them into every problem we have because they are solutions to problems, not tools that should be used everywhere. **If you can implement a simple solution without using a design pattern, do it!**

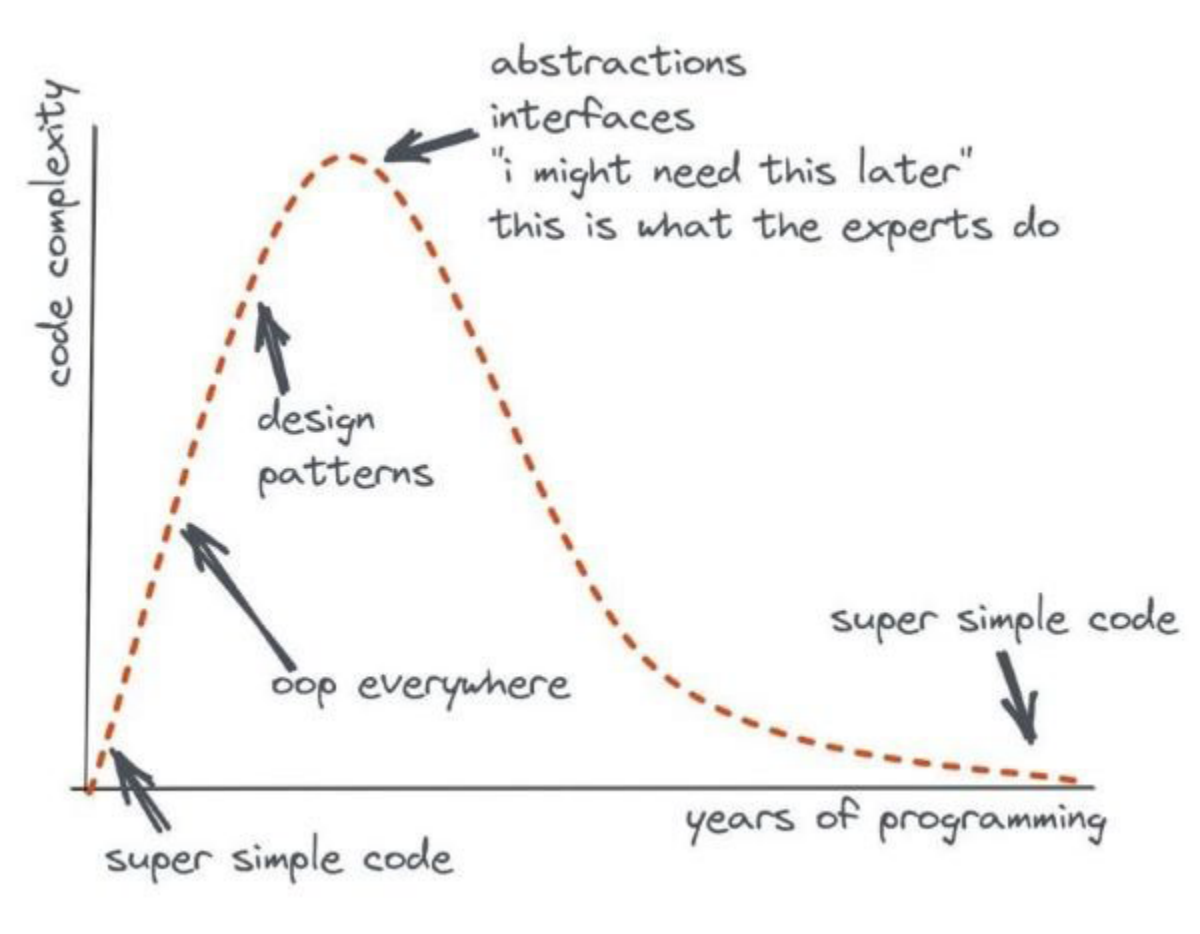



Image credits (@flaviocopes on Twitter)

We find something even more problematic in the latest O'Reilly Technology Trends for 2024. Managers are asking developers how many patterns they use. **This is an open door to over-engineering.**

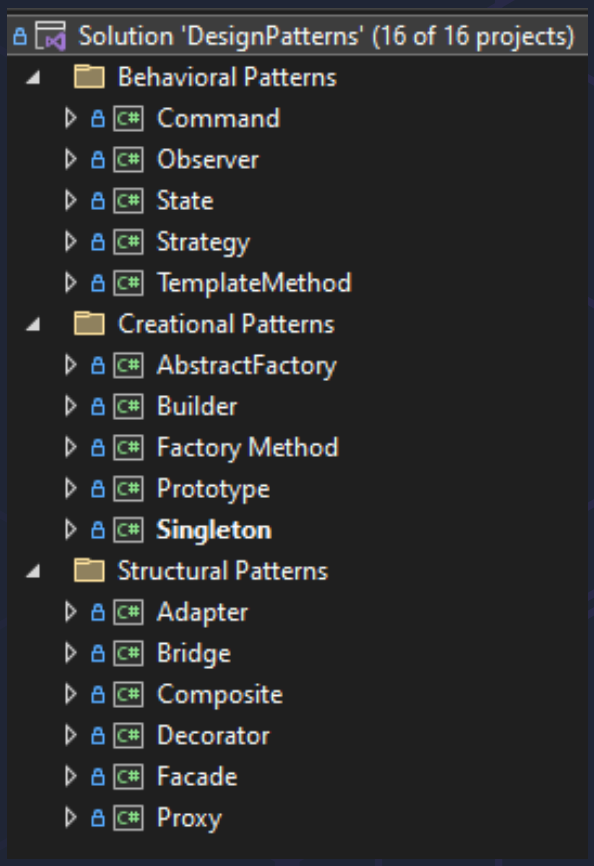
> At the same time, whenever there's a surge of interest in design patterns, there's a corresponding surge in pattern abuse: managers asking developers how many patterns they used (as if pattern count were a metric for good code), developers implementing FactoryFactoryFactory Factories, and the like. What goes around comes around, and the abuse of design patterns is part of a feedback loop that regulates the use of design patterns.

But that's not all, we also saw examples of codebases that organize their code by folder names of design patterns. It cannot be more wrong than this!

Yet, understanding design patterns keeps you from reinventing wheels.

In the rest of the article, we will go through all the important Design Patterns. During many years of using these patterns, I noticed that some are used often, some rarely, and some are not. Here, I will present only those patterns you need daily.

Every pattern has an **implementation in C# language**. The solution can be run using .NET 8.



## Creational Design Patterns

Creational patterns focus on instantiating an object or group of related objects.

Let’s go deep into the most critical creational patterns.

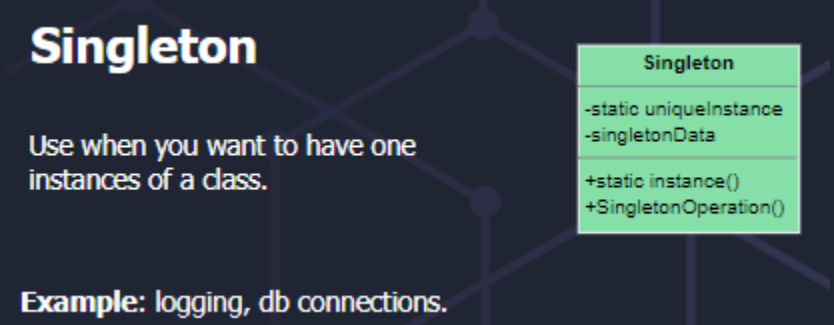

### Singleton

**Usage:** Use when a single instance of a class is needed. Some examples are logging and database connections.

**Real-world example:** Only one CEO leads the company, making decisions and representing the entire organization, like a singleton providing global access and control.

**The remark about usage:** The Singleton pattern is regarded as **an anti-pattern**; hence, using them excessively is advised. Why? Using it, we tend to make procedural code with global variables.

**UML diagram of Singleton pattern:**




**An example in C#:**

```
public class Logger
{
    private static Logger instance;
    private static readonly object lockObject = new object();

    private Logger() {}

    public static Logger Instance
    {
        get
        {
            lock (lockObject)
            {
                if (instance == null)
                {
                    instance = new Logger();
                }
                return instance;
            }
        }
    }

    public void Log(string message)
    {
        Console.WriteLine($"Log: {message}");
    }
}
```

The Singleton pattern in C#

An important note about the Singleton pattern is that it **poses a significant issue for multithreading**, especially in a scenario where multiple threads may simultaneously attempt to create an instance of the Singleton class. This can lead to multiple instances if proper synchronization is not enforced, thus violating the core principle of the Singleton pattern that only one instance of the class should ever exist.

There are multiple solutions to this problem, such as eager initialization, double-checked locking, or lazy initialization.

```
public class Singleton
{
    private static readonly Lazy<Singleton> lazy =
        new Lazy<Singleton>(() => new Singleton());

    public static Singleton Instance => lazy.Value;

    private Singleton() {}
}
```

The Singleton pattern with thread-safe lazy initialization by default

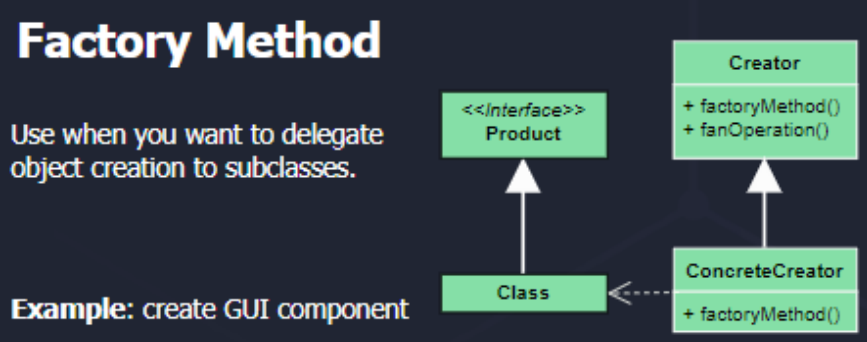
## Factory method

**Usage:** Decouple object creation from usage. For example, you create different types of database connections based on configuration.

**Real-world example:** Think of a pizza joint with a "Pizza Factory" instead of chefs. Customers order "cheese" or "pepperoni," not knowing how it's made. Based on the order, this factory tells specialized "CheesePizza" or "PepperoniPizza" builders to get cookin'. Each builder adds signature toppings, keeping the creation logic separate but the ordering process smooth.

**The remark about usage:** It can lead to increased classes, potentially making the codebase more complex.

**UML diagram of Factory Method pattern:**




**An example in C#**:

```
public interface IDatabaseConnectionFactory
{
    IDatabaseConnection CreateConnection();
}

public class SqlDatabaseConnectionFactory : IDatabaseConnectionFactory
{
    public IDatabaseConnection CreateConnection()
    {
        return new SqlConnection();
    }
}

public class OracleDatabaseConnectionFactory : IDatabaseConnectionFactory
{
    public IDatabaseConnection CreateConnection()
    {
        return new OracleConnection();
    }
}
```

An example of a Factory method in C#

And now the usage of the factory:

```
public class DatabaseClient
{
    private readonly IDatabaseConnection connection;

    public DatabaseClient(IDatabaseConnectionFactory factory)
    {
        connection = factory.CreateConnection();
    }

    public void UseDatabase()
    {
        connection.Connect();
        // Perform database operations...
        connection.Disconnect();
    }
}

// Example usage
public class Program
{
    public static void Main(string[] args)
    {
        IDatabaseConnectionFactory factory;

        // The choice of factory can be dynamically decided based on configuration or environment settings
        factory = new SqlDatabaseConnectionFactory();
        // For Oracle: factory = new OracleDatabaseConnectionFactory();

        DatabaseClient client = new DatabaseClient(factory);
        client.UseDatabase();
    }
}
```

Usage of the Factory method

## Builder

**Usage:** Constructing complex objects step by step. For example, if you need to create a complex domain object.

**Real-world example:** If we hire an architect to design our dream home, we don't need to know every construction detail. We need to tell the architect our preferences (number of rooms, style, materials), and they create a blueprint with those specifications. The architect acts as the "builder pattern," guiding us through the creation process with clear steps (foundation, walls, roof), ensuring correct order, and handling complex details. You make choices (fireplace or no fireplace?), and the builder incorporates them, constructing the house piece by piece until it's complete.

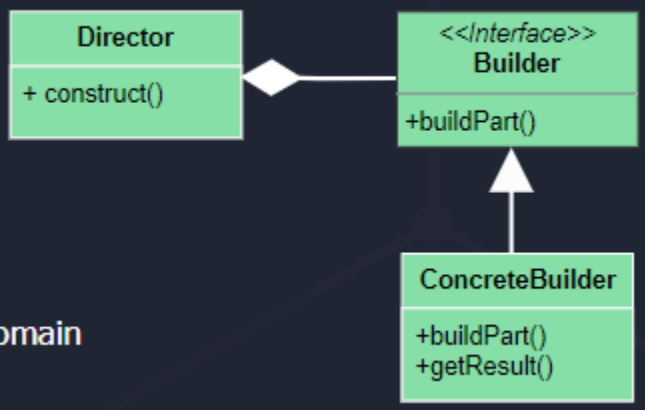
**The remark about usage:** It could lead to increased complexity due to the introduction of multiple new classes.

**UML diagram of Builder pattern:**

# Builder

Constructing complex objects, step by step

**Example**: create complex domain object



**An example in C#** (creating a system to assemble customized compute)

```
public class Computer
{
    public string CPU { get; set; }
    public string RAM { get; set; }

    public override string ToString()
    {
        return $"CPU: {CPU}, RAM: {RAM}";

    }
}

public interface IComputerBuilder
{
    void BuildCPU();
    void BuildRAM();
    Computer GetComputer();
}

public class GamingComputerBuilder : IComputerBuilder
{
    private Computer _computer = new Computer();

    public void BuildCPU()
    {
        _computer.CPU = "High-End CPU";
    }

    public void BuildRAM()
    {
        _computer.RAM = "32GB";
    }

    public Computer GetComputer()
    {
        return _computer;
    }
}

public class OfficeComputerBuilder : IComputerBuilder
{
    private Computer _computer = new Computer();

    public void BuildCPU()
    {
        _computer.CPU = "Mid-Range CPU";
    }

    public void BuildRAM()
    {
        _computer.RAM = "16GB";
    }

    public Computer GetComputer()
    {
        return _computer;
    }
}

public class ComputerDirector
{
    private IComputerBuilder _builder;

    public ComputerDirector(IComputerBuilder builder)
    {
        _builder = builder;
    }

    public void ConstructComputer()
    {
        _builder.BuildCPU();
        _builder.BuildRAM();
    }

    public Computer GetComputer()
    {
        return _builder.GetComputer();
    }
}
```

Builder pattern in C#

And now the usage of the builder:

```
class Program
{
    static void Main(string[] args)
    {
        IComputerBuilder gamingBuilder = new GamingComputerBuilder();
        ComputerDirector director = new ComputerDirector(gamingBuilder);

        director.ConstructComputer();
        Computer gamingComputer = director.GetComputer();
        Console.WriteLine("Gaming Computer: " + gamingComputer);

        IComputerBuilder officeBuilder = new OfficeComputerBuilder();
        director = new ComputerDirector(officeBuilder);

        director.ConstructComputer();
        Computer officeComputer = director.GetComputer();
        Console.WriteLine("Office Computer: " + officeComputer);
    }
}
```

Usage of the Builder pattern

Other exciting design patterns from this group are:

- **Abstratct Factory:** Create families of related objects. For example, I build parsers for different file formats (e.g., JSON, XML, CSV). Check the implementation in C#.
- **Prototype:** Creating duplicate objects and reusing cached objects to reduce database calls. Check the implementation in C#.

## Structural Patterns

Structural patterns are primarily concerned with object composition or, in other words, how the entities can use each other.

Here are the most critical structural patterns we should know about.

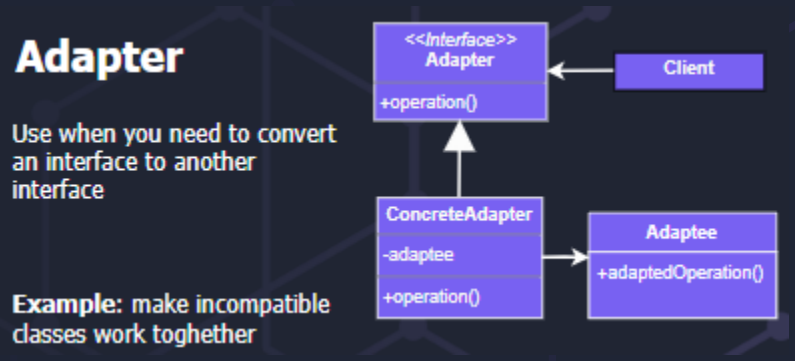

### Adapter

**Usage:** Make incompatible interfaces compatible. For example, it integrates a new logging library into an existing system that expects a different interface.

**Real-world example:** Allows you to use your devices in different countries by adapting to the local power outlet (adapter mediates communication between incompatible systems).

**The remark about usage:** Can lead to an increase in the number of adapters, making the system more complicated. Modifying the service class to align with the rest of the codebase could be simpler sometimes.

**UML diagram of Adapter pattern:**




**An example in C#:**

```
public interface IPaymentGateway
{
    void ProcessPayment(string merchantId, decimal amount);
}

public class PayPal
{
    public void SendPayment(decimal amount)
    {
        Console.WriteLine($"Paying via PayPal: {amount}");
    }
}

public class PayPalAdapter : IPaymentGateway
{
    private readonly PayPal _payPal;

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

    public void ProcessPayment(string merchantId, decimal amount)
    {
        _payPal.SendPayment(amount); // Translate and forward the call
    }
}

// Simularly wer would create the adapter for Stripe
```

The Adapter pattern in C#

A concrete usage of this pattern would be:

```
public class PaymentProcessor
{
    public void ProcessPayment(IPaymentGateway paymentGateway, string merchantId, decimal amount)
    {
        paymentGateway.ProcessPayment(merchantId, amount);
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        IPaymentGateway paypalGateway = new PayPalAdapter(new PayPal());
        //IPaymentGateway stripeGateway = new StripeAdapter(new Stripe());

        PaymentProcessor processor = new PaymentProcessor();

        // Process payment through PayPal
        processor.ProcessPayment(paypalGateway, "merchant123", 100.00m);

        // Process payment through Stripe
        //processor.ProcessPayment(stripeGateway, "merchant456", 200.00m);
    }
}
```

Usage of the Adapter pattern

## Composite

**Usage**: Represent part-whole hierarchies. For example, graphic objects in a drawing application can be grouped and treated uniformly.

**Real-world example:** In the library, books are organized on shelves, but each shelf can further hold categories (fiction, history). These categories might even contain subcategories (romance, mystery). Each shelf acts as a composite, keeping both individual books (leaf nodes) and other categories (composite nodes).

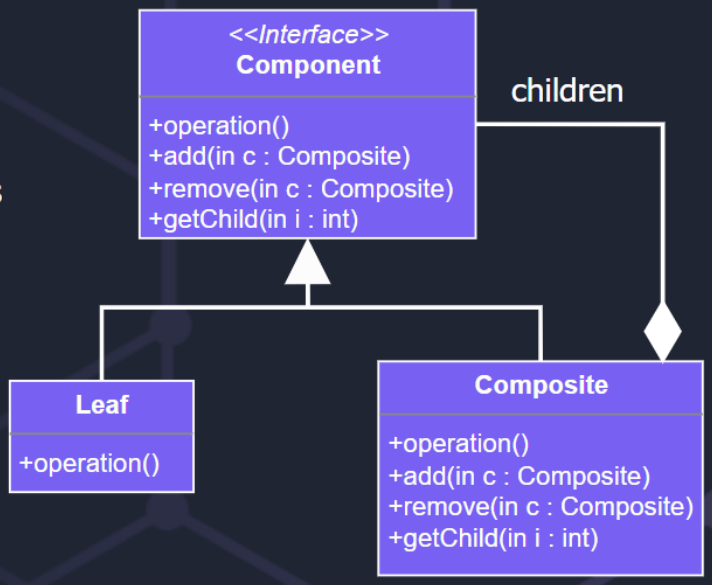
**The remark about usage:** Restricting operations for certain components or leaves in the hierarchy could be challenging.

**UML diagram of Composite pattern:**

# Composite

Represent part-whole hierarchies

**Example**: Graphic object in a drawing can be grouped



**An example in C#**:

```
// 1. Define the component base class
public abstract class GUIComponent
{
    protected string name;

    public GUIComponent(string name)
    {
        this.name = name;
    }

    public abstract void Display(int depth);
}

// 2. Create leaf objects
public class Button : GUIComponent
{
    public Button(string name) : base(name) { }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + " Button: " + name);
    }
}

public class TextBox : GUIComponent
{
    public TextBox(string name) : base(name) { }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + " TextBox: " + name);
    }
}

// 3. Create the composite object
public class Panel : GUIComponent
{
    private List<GUIComponent> children = new List<GUIComponent>();

    public Panel(string name) : base(name) { }

    public void Add(GUIComponent component)
    {
        children.Add(component);
    }

    public void Remove(GUIComponent component)
    {
        children.Remove(component);
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + " Panel: " + name);
        // Recursively display child nodes
        foreach (var component in children)
        {
            component.Display(depth + 2);
        }
    }
}
```

The Composite pattern in C#

The usage of the Composite would look like this:

```
class Program
{
    static void Main(string[] args)
    {
        // Create leaf components
        GUIComponent button1 = new Button("Login");
        GUIComponent textBox1 = new TextBox("Username");

        // Create a composite component
        Panel mainPanel = new Panel("MainPanel");
        mainPanel.Add(button1);
        mainPanel.Add(textBox1);

        // Create another panel and add it to the main panel
        Panel subPanel = new Panel("SubPanel");
        subPanel.Add(new TextBox("Password"));
        subPanel.Add(new Button("Cancel"));
        mainPanel.Add(subPanel);

        // Display the tree structure
        mainPanel.Display(1);
    }
}
```

Usage of the Composite pattern

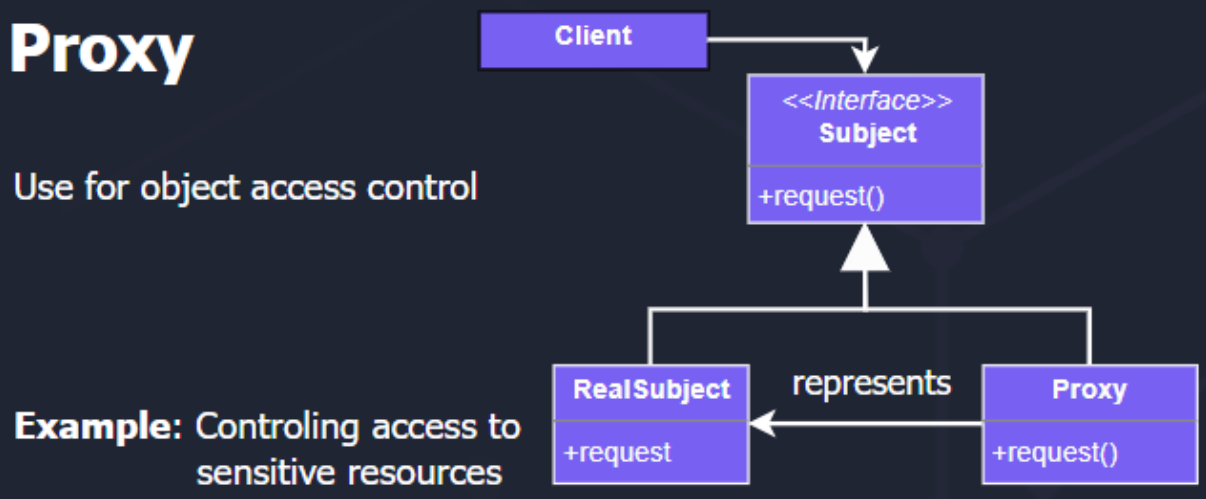
## Proxy

**Usage**: Control access to objects. For example, lazy loading of a high-resolution image in a web application.

**Real-world example:** Let's assume you're a CEO with a personal assistant who acts as a "proxy," handling requests and shielding you from unnecessary distractions. The assistant assesses each request, prioritizing the important ones, filtering out spam, and preparing relevant info. Only the filtered essentials reach the CEO, who focuses on big decisions, while the assistant handles the rest.

**The remark about usage:** Overusing proxies can add unnecessary complexity and impact performance.

**UML diagram of Proxy pattern:**



**An example in C#** (a Proxy pattern is used to control access to sensitive documents):

```
// 1. This interface declares operations for accessing documents.
public interface IDocumentAccessor
{
    string FetchDocument(string documentId);
}

// 2. This class represents the actual document access mechanism
public class DocumentAccessor : IDocumentAccessor
{
    public string FetchDocument(string documentId)
    {
        // In a real scenario, this would involve database access or API call
        return $"Document content for {documentId}.";
    }
}

// 3. The Proxy class implements the same interface as the real subject.
public class SecureDocumentAccessorProxy : IDocumentAccessor
{
    private DocumentAccessor _documentAccessor;
    private readonly string _userId;

    public SecureDocumentAccessorProxy(string userId)
    {
        _userId = userId;
        _documentAccessor = new DocumentAccessor();
    }

    public string FetchDocument(string documentId)
    {
        // Check if the user has access to the document
        if (!HasAccess(_userId, documentId))
        {
            Console.WriteLine($"Access denied for user {_userId} to document {documentId}.");
            return null;
        }

        // Log the access for audit
        LogAccess(_userId, documentId);

        // Delegate the call to the real document accessor
        Console.WriteLine($"User {_userId} accessed document {documentId}.");
        return _documentAccessor.FetchDocument(documentId);
    }

    private bool HasAccess(string userId, string documentId)
    {
        // In a real system, this would check user permissions against the document's access control list.
        return true; // Assume all users have access for this example
    }

    private void LogAccess(string userId, string documentId)
    {
        // In a real system, this would write to a secure log for audit purposes.
        Console.WriteLine($"Access log: User {userId} accessed {documentId} on {DateTime.UtcNow}.");
    }
}
```

The Proxy pattern in C#

The usage of this pattern would look like this:

```
class Program
{
    static void Main(string[] args)
    {
        string userId = "user123"; // Simulate a user ID
        string documentId = "doc456"; // Simulate a document ID

        IDocumentAccessor documentAccessor = new SecureDocumentAccessorProxy(userId);
        string documentContent = documentAccessor.FetchDocument(documentId);

        if (documentContent != null)
        {
            Console.WriteLine($"Fetched document content: {documentContent}");
        }
    }
}
```

Usage of the Proxy pattern

## Decorator

**Usage**: Dynamically add/remove behavior. For example, we are implementing compression or encryption on top of file streams.

**Real-world example:** If we want to make a coffee, we would start with plain coffee (the core object). Then, "decorate" it with cream (adds richness), sugar (sweetness), and cinnamon (extra flavor), each a "decorator" enhancing the base coffee without altering it. You can even combine them (multiple decorators) for unique creations like a creamy, sweet cinnamon latte!

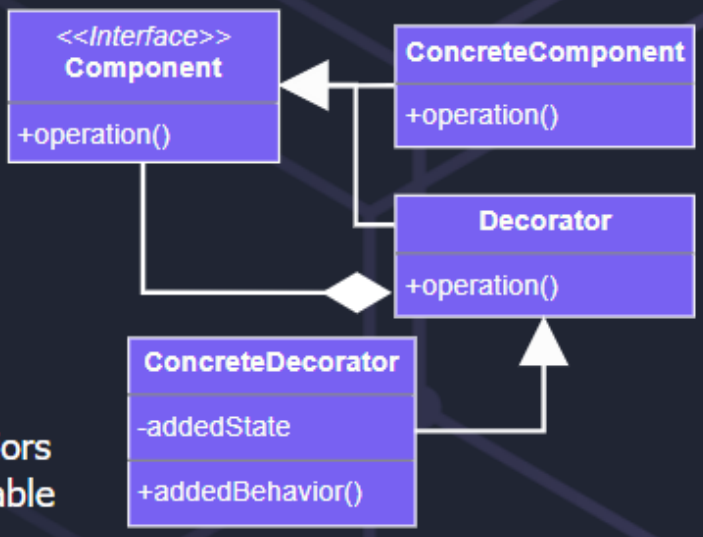
**The remark about usage:** Overuse of decorators can lead to a complex hierarchy of objects.

**UML diagram of Decorator pattern:**

# Decorator

Use when you need to wrap objects to modify their behaviors

**Example:** make object behaviors dynamically modifiable



**An example in C#:**

```
using System.IO;
using System.IO.Compression;

public interface IFileOperation
{
    byte[] Read(string filePath);
    void Write(string filePath, byte[] data);
}

public class FileOperation : IFileOperation
{
    public byte[] Read(string filePath)
    {
        return File.ReadAllBytes(filePath);
    }

    public void Write(string filePath, byte[] data)
    {
        File.WriteAllBytes(filePath, data);
    }
}

public class CompressionDecorator : IFileOperation
{
    private readonly IFileOperation _fileOperation;

    public CompressionDecorator(IFileOperation fileOperation)
    {
        _fileOperation = fileOperation;
    }

    public byte[] Read(string filePath)
    {
        var compressedData = _fileOperation.Read(filePath);
        using (var input = new MemoryStream(compressedData))
        using (var gzipStream = new GZipStream(input, CompressionMode.Decompress))
        using (var output = new MemoryStream())
        {
            gzipStream.CopyTo(output);
            var decompressedData = output.ToArray();
            return decompressedData;
        }
    }

    public void Write(string filePath, byte[] data)
    {
        using (var output = new MemoryStream())
        {
            using (var gzipStream = new GZipStream(output, CompressionMode.Compress))
            {
                gzipStream.Write(data, 0, data.Length);
            }
            var compressedData = output.ToArray();
            _fileOperation.Write(filePath, compressedData);
        }
    }
}
```

The Decorator pattern in C#

The usage of the Decorator pattern would look like this.

```
class Program
{
    static void Main(string[] args)
    {
        // The file path for the compressed data
        string compressedFilePath = "compressed.gz";

        // Original data to compress and write
        string originalData = "This is a test string to compress and decompress.";
        byte[] dataToCompress = System.Text.Encoding.UTF8.GetBytes(originalData);

        // Initialize the file operation with compression decorator
        IFileOperation compressedFileOp = new CompressionDecorator(new FileOperation());

        // Compress and write data to file
        compressedFileOp.Write(compressedFilePath, dataToCompress);
        Console.WriteLine($"Data compressed and written to {compressedFilePath}");

        // Read and decompress data from file
        byte[] decompressedData = compressedFileOp.Read(compressedFilePath);

        // Convert decompressed data back to string
        string decompressedString = System.Text.Encoding.UTF8.GetString(decompressedData);
        Console.WriteLine($"Decompressed data: {decompressedString}");
    }
}
```

Usage of the Decorator pattern

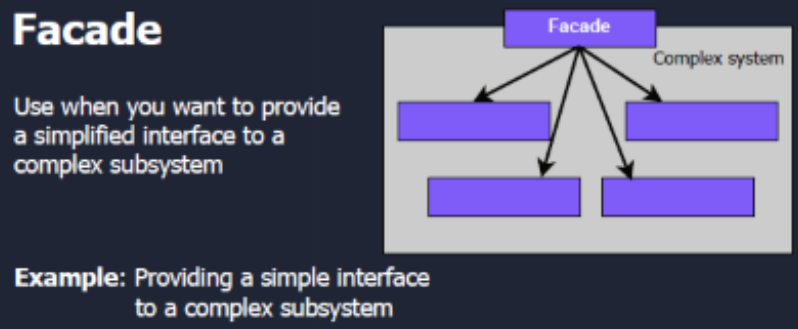
## Facade Pattern

**Usage**: It provides a simplified interface to a complex subsystem.

**Real-world example**: Let's say we are visiting a hotel. We need various services - ordering room service, booking spa appointments, requesting housekeeping. Instead of contacting each department individually, you call the front desk. The front desk acts as a facade, hiding the complexity of the underlying systems. They take your request, communicate with the relevant department (kitchen, spa, housekeeping), and deliver the service, making everything seem effortless. You don't need to know how each department works; the facade provides a simplified interface to access them all.

**The remark about usage**: Facades have the potential to transform into god objects connected to every application class.

**UML diagram of Facade pattern:**



An example in C#

```
public interface ICloudStorageFacade
{
    void UploadFile(string fileName, byte[] fileContent);
    byte[] DownloadFile(string fileName);
}   result += value;
    }
    return result;
}

public class AzureStorageFacade : ICloudStorageFacade
{
    // Assume these are initialized with proper configurations
    //private BlobServiceClient blobServiceClient;

    public AzureStorageFacade()
    {
        // Initialize BlobServiceClient with credentials and config
    }

    public void UploadFile(string fileName, byte[] fileContent)
    {
        Console.WriteLine($"Uploading {fileName} to Azure Blob Storage");
        // BlobClient blobClient = blobServiceClient.GetBlobClient(fileName);
        // blobClient.Upload(fileContent);
    }

    public byte[] DownloadFile(string fileName)
    {
        Console.WriteLine($"Downloading {fileName} from Azure Blob Storage");
        // byte[] fileContent = blobClient.Download(fileName);
        string simulatedContent = "Simulated Azure content for " + fileName;
        return System.Text.Encoding.UTF8.GetBytes(simulatedContent);
    }
}


public class AwsStorageFacade : ICloudStorageFacade
{
    // Assume these are initialized with proper configurations
    //private AmazonS3Client s3Client;

    public AwsStorageFacade()
    {
        // Initialize AmazonS3Client with credentials and config
    }

    public void UploadFile(string fileName, byte[] fileContent)
    {
        // Simplified upload logic
        Console.WriteLine($"Uploading {fileName} to AWS S3");
        // s3Client.PutObject(bucketName, fileName, fileContent);
    }

    public byte[] DownloadFile(string fileName)
    {
        Console.WriteLine($"Downloading {fileName} from AWS S3");
        // byte[] fileContent = s3Client.GetObject(bucketName, fileName);
        string simulatedContent = "Simulated AWS S3 content for " + fileName;
        return System.Text.Encoding.UTF8.GetBytes(simulatedContent);
    }
}
```

The Façade pattern in C#

The usage of the Facade pattern would look like this.

```
// Choose the desired cloud storage facade implementation
ICloudStorageFacade cloudStorage = new AwsStorageFacade(); // or new AzureStorageFacade() for Azure

string fileName = "example.txt";

// Simulated file content as a byte array
// For demonstration purposes, let's create a simple string and convert it to a byte array
string textContent = "Hello, Cloud Storage!";
byte[] fileContent = System.Text.Encoding.UTF8.GetBytes(textContent);

// Uploading simulated content to cloud storage
cloudStorage.UploadFile(fileName, fileContent);

// Assuming the download method returns the content we just uploaded
byte[] downloadedContent = cloudStorage.DownloadFile(fileName);

// Optionally, converting the downloaded byte array back to a string to verify the content
string downloadedText = System.Text.Encoding.UTF8.GetString(downloadedContent);
Console.WriteLine($"Downloaded content: {downloadedText}");
```

Usage of the Facade pattern

Another interesting design pattern from this group is the **Bridge pattern**. It is used to decouple abstraction from implementation. For example, I am separating platform-specific code from core logic. Check the implementation in C#.

## Behavioral Patterns

They focus on distributing responsibility among the objects. They differ from structural patterns in that they define the patterns for message conveyance and communication between them in addition to the structure.

Here are the most critical behavioral patterns we should know about.

### Facade Pattern

**Usage**: Define a family of algorithms. For example, they allow users to choose different sorting or compression algorithms.

**Real-world example:** Let’s plan a travel from city A to city B. You can choose a "transportation strategy" based on your needs: take a fast train (speed focus), a comfortable bus (comfort focus), or a budget-friendly carpool (low-cost focus).

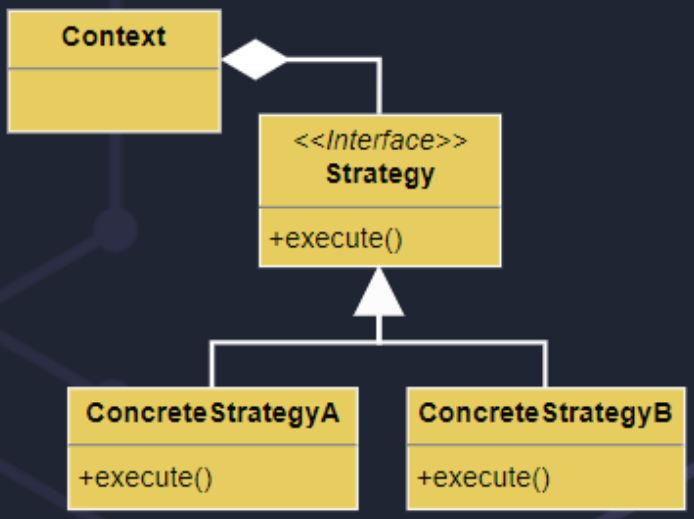
**The remark about usage:** This could lead to an increased number of classes and complexity when dealing with many strategies. There is no need to add new classes and interfaces if you have only a few algorithms that change rarely.

**UML diagram of Strategy pattern:**

# Strategy

Use for interchangeable algorithms that can be swaped at runtime

**Example:** Implement different sorting algorithms



**An example in C#** (rendering data in different formats based on user preference):

```
public interface IRenderStrategy
{
    string Render(string data);
}

public class JsonRenderStrategy : IRenderStrategy
{
    public string Render(string data)
    {
        // Simulate converting data to JSON format
        return $"{{\"data\": \"{data}\"}}";
    }
}

public class CsvRenderStrategy : IRenderStrategy
{
    public string Render(string data)
    {
        // Simulate converting data to CSV format
        return $"data,{data}";
    }
}

public class DataRenderer
{
    private IRenderStrategy _renderStrategy;

    // The render strategy can be set via the constructor or a setter method
    public DataRenderer(IRenderStrategy renderStrategy)
    {
        _renderStrategy = renderStrategy;
    }

    public void SetRenderStrategy(IRenderStrategy renderStrategy)
    {
        _renderStrategy = renderStrategy;
    }

    public string RenderData(string data)
    {
        return _renderStrategy.Render(data);
    }
}
```

An example of the Strategy pattern in C#

The usage of the Strategy pattern would look like this:

```
class Program
{
    static void Main(string[] args)
    {
        string sampleData = "Hello, Strategy Pattern";

        // Renders data in JSON format
        DataRenderer renderer = new DataRenderer(new JsonRenderStrategy());
        Console.WriteLine(renderer.RenderData(sampleData));

        // Renders data in CSV format
        renderer.SetRenderStrategy(new CsvRenderStrategy());
        Console.WriteLine(renderer.RenderData(sampleData));
    }
}
```

Usage of the Strategy pattern

## Observer

**Usage**: Maintain a consistent state by being notified of changes and, for example, notifying subscribers of events in a messaging system.

**Real-world example:** We can think of a breaking news app. Users subscribe to specific topics (sports, politics, etc.), acting as "observers." When news breaks in a subscribed topic, the "observer pattern" notifies all relevant users with personalized alerts. Sports fans get their scores, and political enthusiasts receive election updates without them needing to check actively.

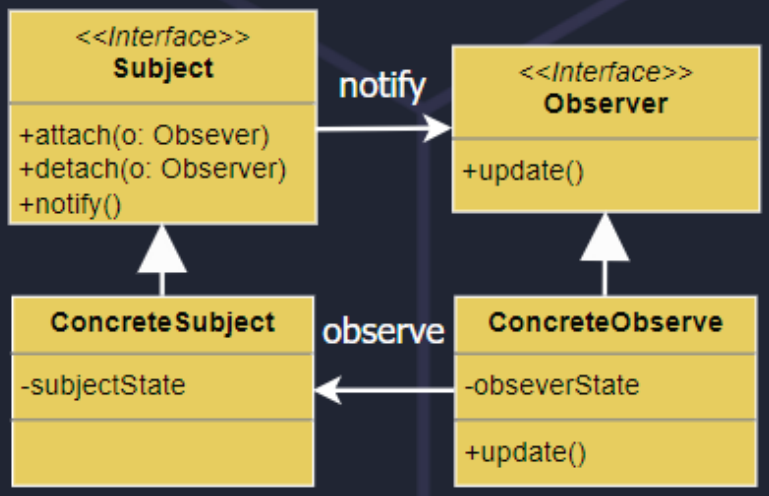
**The remark about usage:** This can result in performance issues when there are numerous observers or the update logic is complex. Subscribers are notified in an unpredictable sequence.

**UML diagram of Observer pattern:**

# Observer

Use for automatic updates of dependand objects

**Example:** Implement subscribers



**An example in C#:**

```
public interface IMessagePublisher
{
    void Subscribe(IObserver observer);
    void Unsubscribe(IObserver observer);
    void NotifySubscribers();
}

public interface IObserver
{
    void Update(string message);
}

public class MessagePublisher : IMessagePublisher
{
    private List<IObserver> _observers = new List<IObserver>();
    private string _latestMessage;

    public void Subscribe(IObserver observer)
    {
        if (!_observers.Contains(observer))
        {
            _observers.Add(observer);
        }
    }

    public void Unsubscribe(IObserver observer)
    {
        if (_observers.Contains(observer))
        {
            _observers.Remove(observer);
        }
    }

    public void NotifySubscribers()
    {
        foreach (IObserver observer in _observers)
        {
            observer.Update(_latestMessage);
        }
    }

    public void PostMessage(string message)
    {
        _latestMessage = message;
        Console.WriteLine($"MessagePublisher: Posting message - {message}");
        NotifySubscribers();
    }
}

public class UserSubscriber : IObserver
{
    private string _name;

    public UserSubscriber(string name)
    {
        _name = name;
    }

    public void Update(string message)
    {
        Console.WriteLine($"{_name} received message: {message}");
    }
}
```

The Observer Pattern in C#

The usage of the Observer pattern would look like follows:

```
class Program
{
    static void Main(string[] args)
    {
        MessagePublisher publisher = new MessagePublisher();

        // Create subscribers
        UserSubscriber alice = new UserSubscriber("Alice");
        UserSubscriber bob = new UserSubscriber("Bob");

        // Subscribe users to receive updates
        publisher.Subscribe(alice);
        publisher.Subscribe(bob);

        // Post a message - all subscribers get notified
        publisher.PostMessage("Hello, World!");

        // Unsubscribe a user
        publisher.Unsubscribe(bob);

        // Post another message - only subscribed users get notified
        publisher.PostMessage("Second message");
    }
}
```

The use of the Observer pattern

## Command

**Usage**: Encapsulate a request as an object. For example, I implement undo/redo functionality in text or image editor.

**Real-world example:** Picture ordering food at a restaurant. You tell the waiter your wishes (pizza, extra cheese), creating an "order command." The waiter then acts as a messenger, carrying your "command" to the Chef in the kitchen (receiver). The Chef, receiving the "command," makes your pizza precisely as specified. This separation of ordering (command) from making (execution) lets you change or cancel easily.

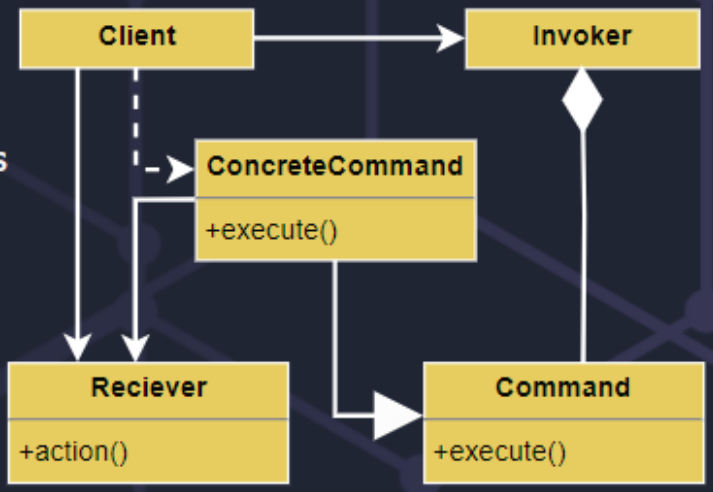
**The remark about usage:** It has the potential to introduce complexity, as it requires creating additional classes for each action or request, complicating the architecture for simple operations.

**UML diagram of Command pattern:**

# Command

Use for encapsulating requests with parameters

**Example:** Implementing operations



**An example in C#:**

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

public class InsertTextCommand : ICommand
{
    private string _textToInsert;
    private Document _document;
    private int _position; // Position at which text is inserted

    public InsertTextCommand(Document document, string text, int position)
    {
        _document = document;
        _textToInsert = text;
        _position = position;
    }

    public void Execute()
    {
        _document.Insert(_position, _textToInsert);
    }

    public void Undo()
    {
        _document.Delete(_position, _textToInsert.Length);
    }
}

public class Document
{
    private StringBuilder _text = new StringBuilder();

    public void Insert(int position, string text)
    {
        _text.Insert(position, text);
    }

    public void Delete(int position, int length)
    {
        _text.Remove(position, length);
    }

    public string GetText(int position, int length)
    {
        return _text.ToString().Substring(position, Math.Min(length, _text.Length - position));
    }

    public override string ToString()
    {
        return _text.ToString();
    }
}
public class CommandManager
{
    private Stack<ICommand> _undoStack = new Stack<ICommand>();
    private Stack<ICommand> _redoStack = new Stack<ICommand>();

    public void ExecuteCommand(ICommand command)
    {
        command.Execute();
        _undoStack.Push(command);
        _redoStack.Clear();
    }

    public void Undo()
    {
        if (_undoStack.Count > 0)
        {
            ICommand command = _undoStack.Pop();
            command.Undo();
            _redoStack.Push(command);
        }
    }

    public void Redo()
    {
        if (_redoStack.Count > 0)
        {
            ICommand command = _redoStack.Pop();
            command.Execute();
            _undoStack.Push(command);
        }
    }
}
```

The Command pattern in C#

The usage of the Command pattern would look like follows:

```
class Program
{
    static void Main(string[] args)
    {
        Document document = new Document();
        CommandManager commandManager = new CommandManager();

        // Simulate text editing
        commandManager.ExecuteCommand(new InsertTextCommand(document, "Hello, ", 0));
        commandManager.ExecuteCommand(new InsertTextCommand(document, "World!", 7));
        Console.WriteLine(document); // Output: Hello, World!

        // Undo last insert
        commandManager.Undo();
        Console.WriteLine(document); // Output: Hello,

        // Redo last insert
        commandManager.Redo();
        Console.WriteLine(document); // Output: Hello, World!
    }
}
```

Usage of the Command pattern

## State

**Usage:** Encapsulate state-specific behavior. For example, we are handling different states of a user interface element (e.g., enabled, disabled, selected).

**Real-world example:** The smartphone effortlessly transitions between states (on, off, silent, airplane mode) based on your actions. Each state (the "concrete state") has unique behavior: on allows calls and notifications, silent mutes them, and airplane mode blocks signals. The phone (the "context") doesn't manage these behaviors directly; it delegates to the current state object. When you press a button or toggle a setting, the phone transitions to a new state, seamlessly changing its behavior without requiring intricate logic.

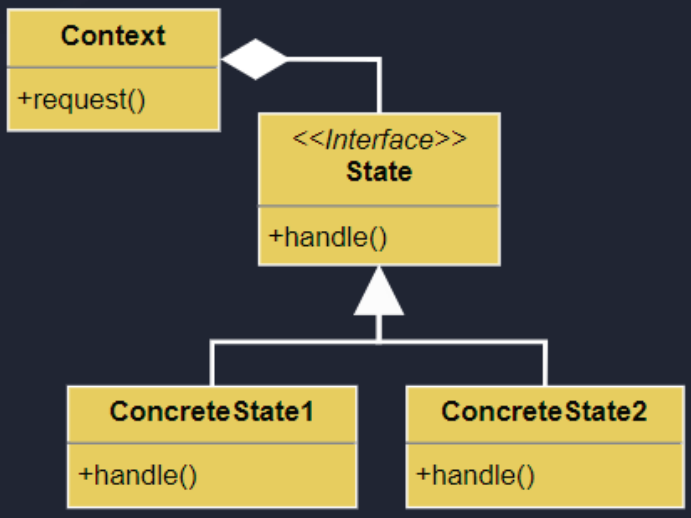
**The remark about usage:** It can lead to a proliferation of classes, as each state is typically represented by its class. This not only increases the complexity of the system but can also make it harder to manage and understand.

**UML diagram of State pattern:**

# State

Encapsulate state-specific behavior

**Example**: Handling different states of a user interface



**An example in C#:**

```
public interface ITextState
{
    string HandleInput(string text);
}

public class NormalTextState : ITextState
{
    public string HandleInput(string text)
    {
        return text; // Returns the text as is
    }
}

public class BoldTextState : ITextState
{
    public string HandleInput(string text)
    {
        return "**" + text + "**"; // Markdown-style bold
    }
}

public class TextEditor
{
    private ITextState _currentState;

    public TextEditor()
    {
        _currentState = new NormalTextState(); // Default state
    }

    public void SetState(ITextState newState)
    {
        _currentState = newState;
    }

    public string TypeText(string text)
    {
        return _currentState.HandleInput(text);
    }
}
```

The State pattern in C#

The usage of the State pattern would look as follows:

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

        // Typing in normal state
        Console.WriteLine(editor.TypeText("This is normal text."));

        // Change state to bold
        editor.SetState(new BoldTextState());
        Console.WriteLine(editor.TypeText("This is bold text."));
    }
}
```

Usage of the State pattern

## Template method

**Usage:** Define the skeleton of an algorithm in operation, deferring some steps to subclasses and implementing a base class for unit testing with customizable setup and teardown steps.

**Real-world example:** Let's say we have a factory where every car (subclasses like Sedan, SUV, and Truck) follows the same basic steps: weld the frame, add the engine, install electrical components, and paint. This is the overall structure defined by the template method. However, each car type has specific variations: Sedans have smaller frames and engines, SUVs have higher clearance and different interiors, and trucks have reinforced frames and more significant engines.

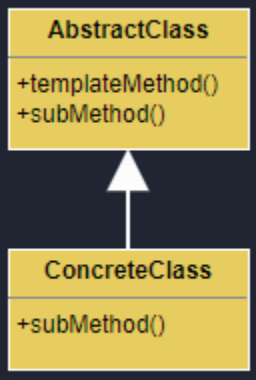
**The remark about usage:** It can lead to overly complex hierarchies when multiple algorithm variations are required.

**UML diagram of Template Method pattern:**

## Template Method

Use when you want to break down an algorithm into a series of steps

**Example**: Common behavior should be located in one class



**An example in C#:**

```
public abstract class DataProcessor
{
    // The template method
    public void ProcessData()
    {
        var data = ReadData();
        var processedData = ProcessDataCore(data);
        SaveData(processedData);
    }

    protected virtual string ReadData()
    {
        // Generic implementation, can be overridden
        Console.WriteLine("Reading generic data...");
        return "Sample data";
    }

    protected abstract string ProcessDataCore(string data); // Requires impl.

    protected virtual void SaveData(string data)
    {
        // Default implementation for saving data
        Console.WriteLine($"Saving data: {data}");
    }
}

public class CsvDataProcessor : DataProcessor
{
    protected override string ReadData()
    {
        Console.WriteLine("Reading CSV data...");
        return "Name, Age, City\nJohn Doe, 29, New York";
    }

    protected override string ProcessDataCore(string data)
    {
        Console.WriteLine("Processing CSV data...");
        // Simulate processing CSV data
        return data.ToUpper(); // Simple transformation for demonstration
    }
}
```

The Template Method in C#

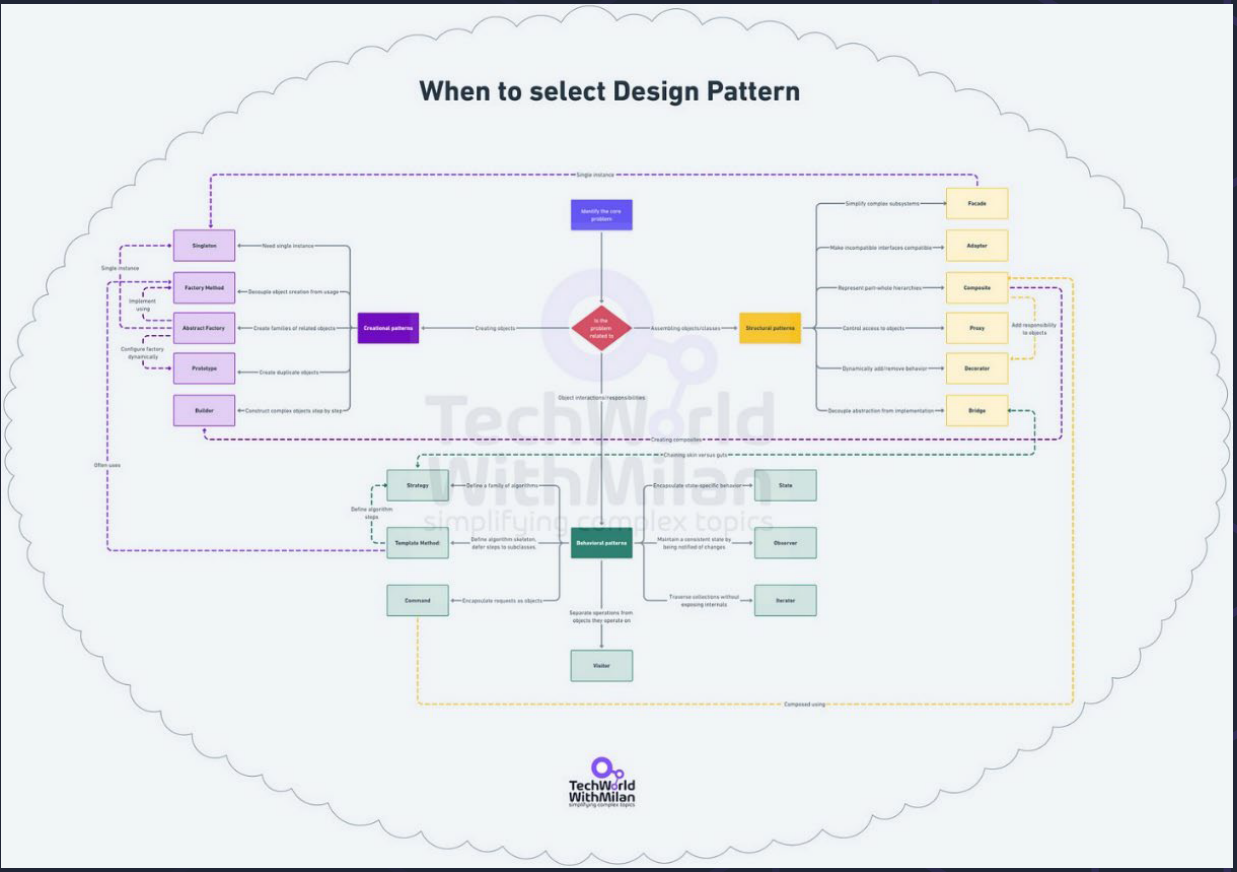
The usage of the Template Method pattern would look as follows:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("CSV Processor:");
        DataProcessor csvProcessor = new CsvDataProcessor();
        csvProcessor.ProcessData();
    }
}
```

Usage of the Template Method pattern

## How To Select The Correct Design Pattern

Now, when we have analyzed all vital design patterns, we came up with the pattern we needed for our problem:




When selecting a design pattern

Here is the summary of the pattern selection:

## HOW TO SELECT DESIGN PATTERN

| | |
|---|---|
| **1. Creational Patterns** → object creation | • **Singleton:** Use when a single instance of a class is needed. Some examples are logging and database connections.<br>• **Factory Method:** Decouple object creation from usage. For example, you create different types of database connections based on configuration.<br>• **Abstract Factory:** Create families of related objects. For example, I build parsers for different file formats (e.g., JSON, XML, CSV).<br>• **Builder:** Constructing complex objects step by step. For example, if you need to create a complex domain object.<br>• **Prototype:** Creating duplicate objects and reusing cached objects to reduce database calls. |
| **2. Structural Patterns** → object assembly | • **Adapter:** Make incompatible interfaces compatible. For example, it integrates a new logging library into an existing system that expects a different interface.<br>• **Composite:** Represent part-whole hierarchies. For example, graphic objects in a drawing application can be grouped and treated uniformly<br>• **Proxy:** Control access to objects. For example, lazy loading of a high-resolution image in a web application.<br>• **Decorator:** Dynamically add/remove behavior. For example, we are implementing compression or encryption on top of file streams.<br>• **Bridge:** Decouple abstraction from implementation. For example, I am separating platform-specific code from core logic |
| **3. Behavioral Patterns** → object interactions | • **Strategy:** Define a family of algorithms. For example, they allow users to choose different sorting or compression algorithms.<br>• **Observer:** Maintain a consistent state by being notified of changes and, for example, notifying subscribers of events in a messaging system.<br>• **Command:** Encapsulate a request as an object. For example, I implement undo/redo functionality in text or image editor.<br>• **State:** Encapsulate state-specific behavior. For example, we are handling different states of a user interface element (e.g., enabled, disabled, selected).<br>• **Template Method:** Define the skeleton of an algorithm in operation, deferring some steps to subclasses and implementing a base class for unit testing with customizable setup and teardown steps. |

*Design patterns are not the only kind of patterns we have. To learn more about other types of patterns, **check here**.*

# Most Used Design Patterns Cheat Sheet

**Creational Patterns**
Used to construct objects

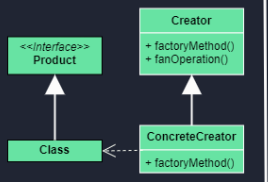
**Structural Patterns**
Used to form large object structures

**Behavioral Patterns**
Used to manage algorithms and relationships

## Factory Method

Use when you want to delegate object creation to subclasses.

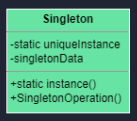
**Example**: create GUI component




## Singleton

Use when you want to have one instances of a class.

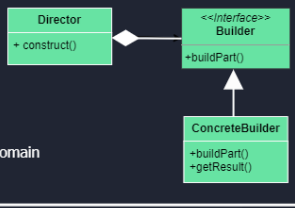
**Example**: logging, db connections.




## Builder

Constructing complex objects, step by step

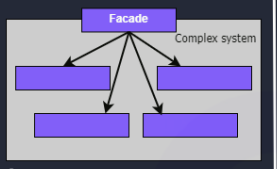
**Example**: create complex domain object




## Facade

Use when you want to provide a simplified interface to a complex subsystem

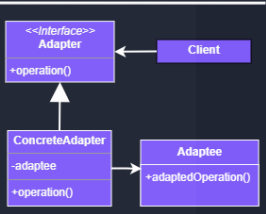
**Example**: Providing a simple interface to a complex subsystem




## Adapter

Use when you need to convert an interface to another interface

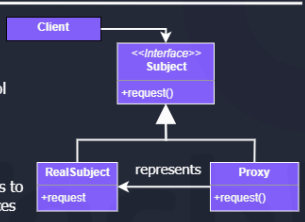
**Example**: make incompatible classes work toghether




## Proxy

Use for object access control

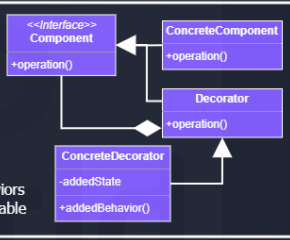
**Example**: Controling access to sensitive resources




## Decorator

Use when you need to wrap objects to modify their behaviors

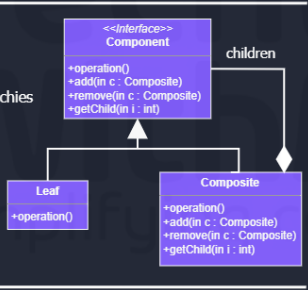
**Example**: make object behaviors dynamically modifiable




## Composite

Represent part-whole hierarchies

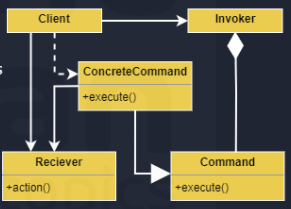
**Example**: Graphic object in a drawing can be grouped




## Command

Use for encapsulating requests with parameters

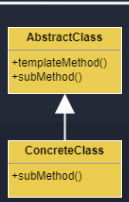
**Example**: Implementing operations




## Template Method

Use when you want to break down an algorithm into a series of steps

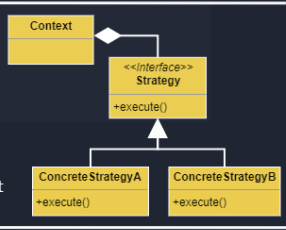
**Example**: Common behavior should be located in one class




## Strategy

Use for interchangeable algorithms that can be swaped at runtime

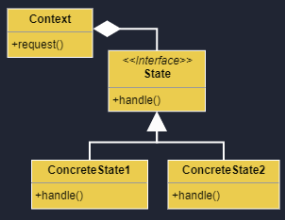
**Example**: Implement different sorting algorithms




## State

Encapsulate state-specific behavior

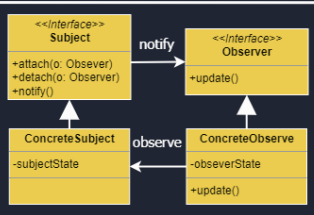
**Example**: Handling different states of a user interface




## Observer

Use for automatic updates of dependand objects

**Example**: Implement subscribers




TechWorld WithMilan
simplifying complex topics

## Resources to learn more

Here are some more resources if you want to dig deeper into the world of design patterns, you can check the following resources:

- **Head First Design Patterns: A Brain-friendly Guide** book.
- **Design Patterns: Elements of Reusable Object-Oriented Software** book by GoF.
- **Awesome Software and Architectural Design Patterns:** A curated list of software and architecture-related design patterns.
- **Design Patterns Library** Pluralsight course.
- **Refactoring.Guru** website.
- **Source Making** website.

# TechWorld WithMilan

simplifying complex topics