Complete Design Patterns Guide

A comprehensive guide combining decision flow charts, pattern descriptions, and practical C# code examples. I've compiled this based on years of experience and common mistakes I've seen developers make.

Table of Contents

Decision Flow Charts

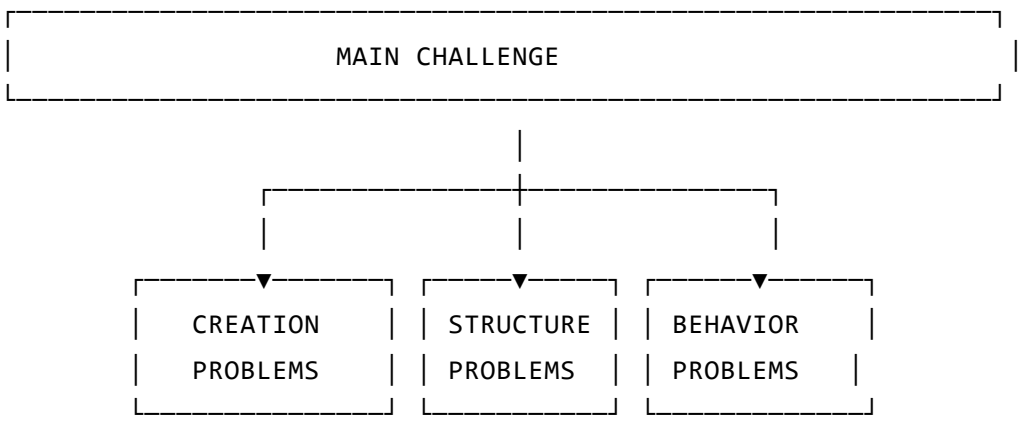
Pattern Categories

Advanced Topics

Decision Tools

Quick Start Decision Tree

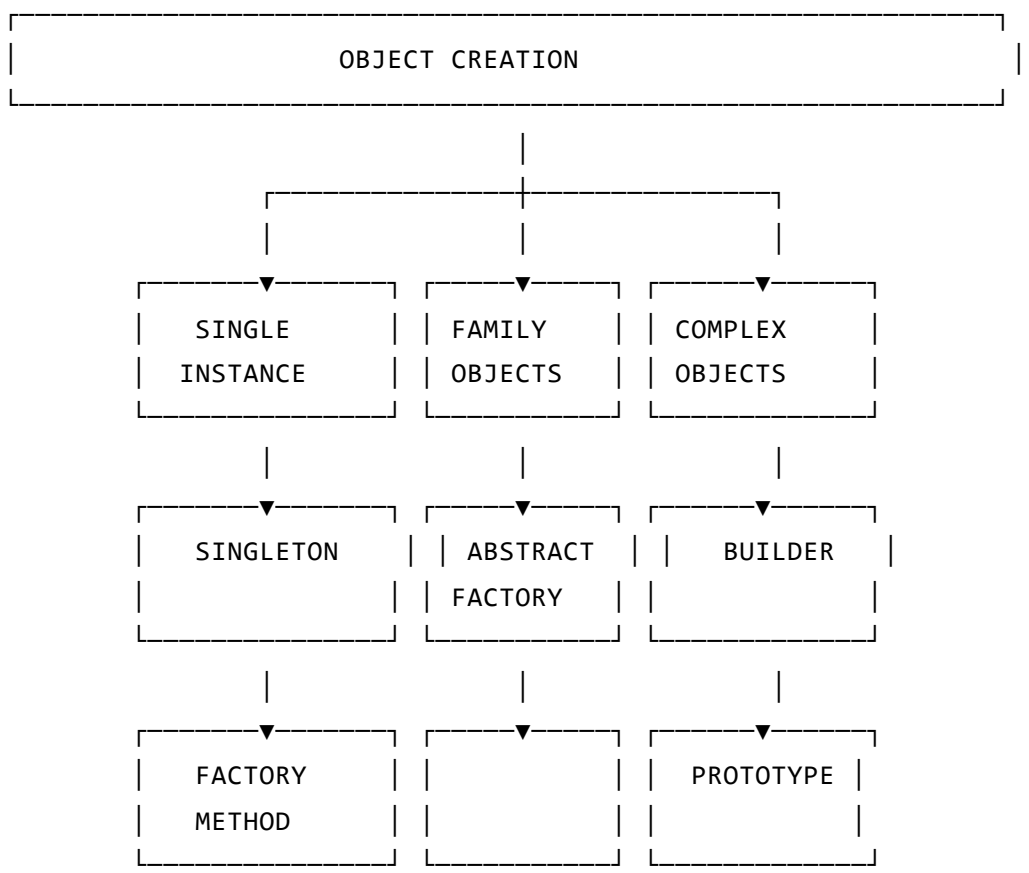
1. What's Your Main Challenge?



CREATIONAL PATTERNS Decision Tree

When you need to create objects...

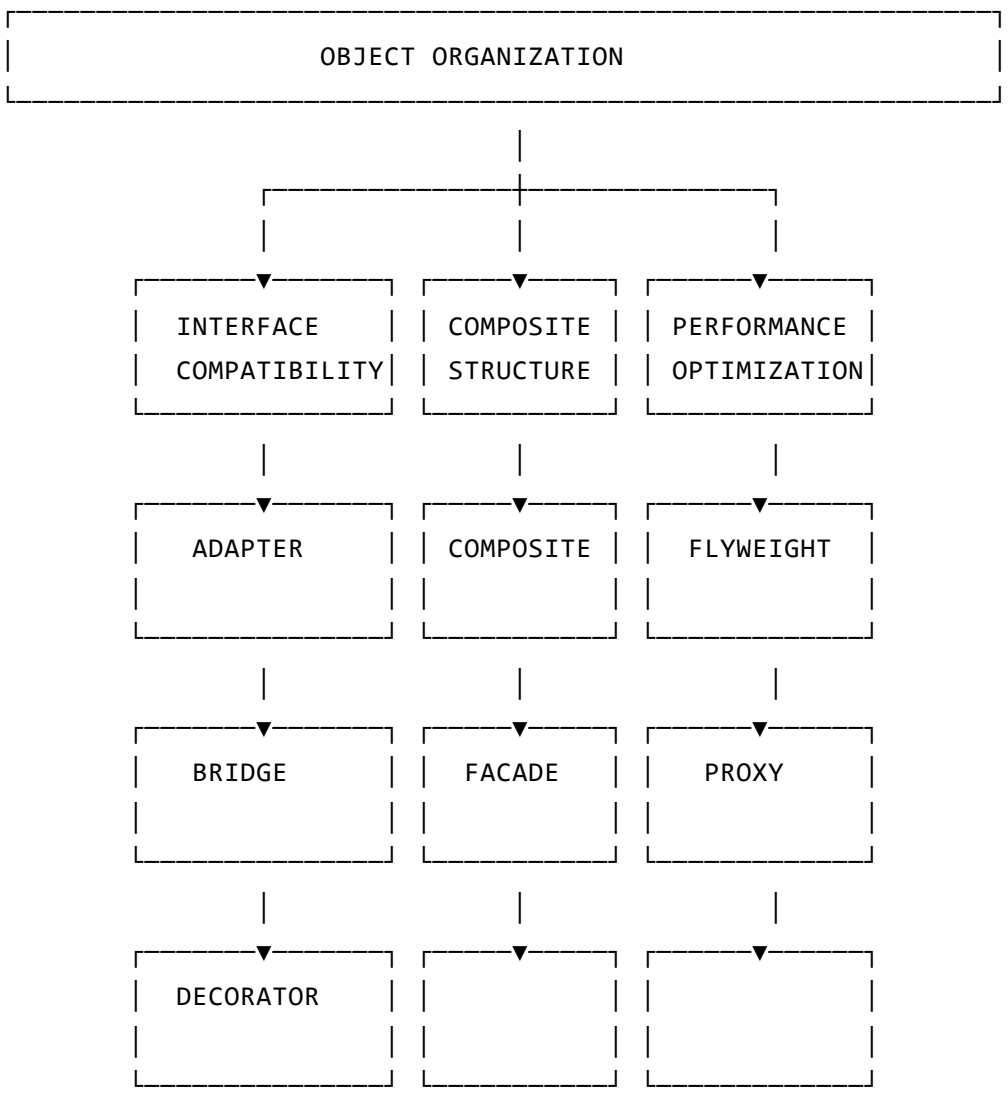
This is probably the most common scenario. Most of the time, you'll want to start with Factory Method or Builder. Singleton should be your last resort - I've seen too many projects ruined by overusing it.



STRUCTURAL PATTERNS Decision Tree

When you need to organize objects...

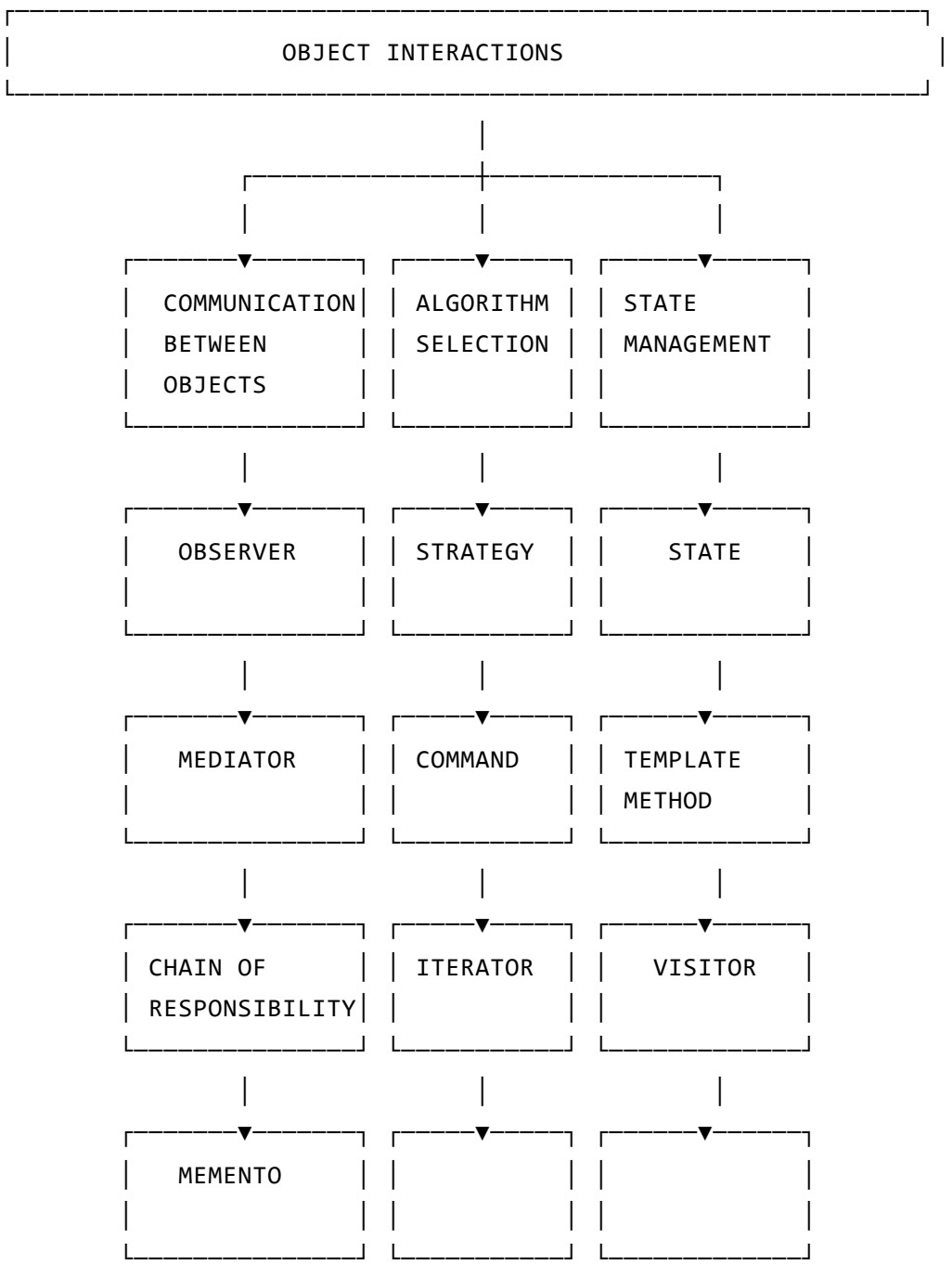
These patterns are about making incompatible things work together or optimizing how objects are structured. Adapter is probably the most useful one here - you'll use it constantly when integrating with third-party libraries.



BEHAVIORAL PATTERNS Decision Tree

When you need to manage object interactions...

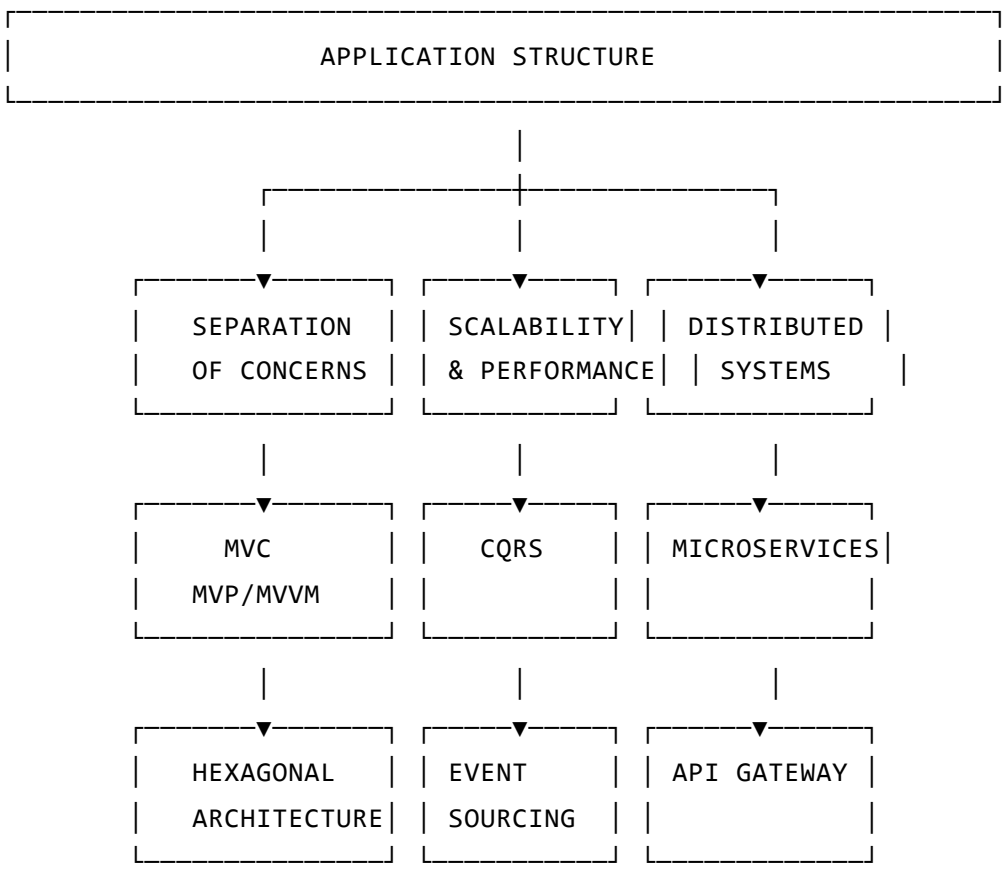
This is where things get interesting. Observer is probably the most important pattern here - it's the foundation of event-driven programming. Strategy is also incredibly useful for making your code more flexible.



ARCHITECTURAL PATTERNS Decision Tree

When you need to structure entire applications...

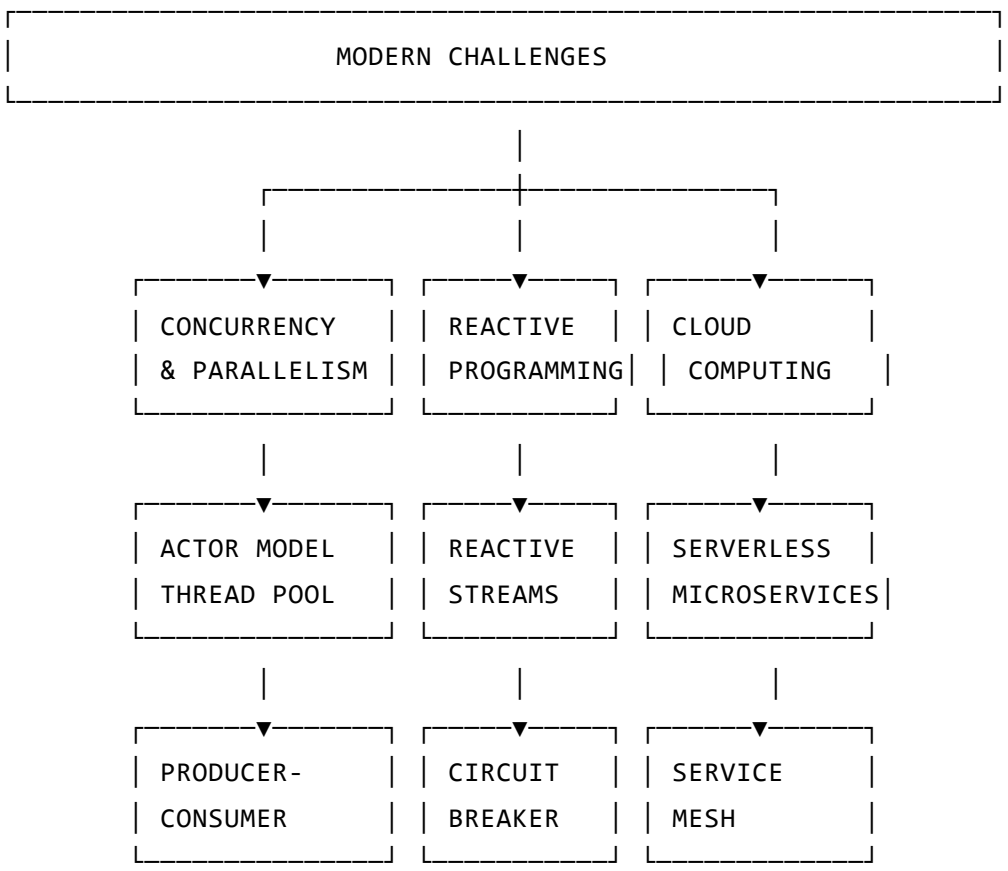
These are the big picture patterns. MVC is probably what you'll use most often, but don't be afraid to mix and match. I've seen some great applications that combine MVC with Repository and Unit of Work patterns.



MODERN PATTERNS Decision Tree

When you need modern solutions...

These are the patterns that have become essential in modern development. Circuit Breaker is a lifesaver when dealing with external services, and Actor Model is great for concurrent programming. Don't try to implement these from scratch - use existing libraries.



1. Creational Patterns (GoF)

Singleton

Description: Ensures a class has only one instance; provides global access. Useful for configuration, logging, or caches. Use sparingly - it can make testing difficult and create hidden dependencies.

```
class Singleton {
    private static Singleton _instance;
    private Singleton() { }
    public static Singleton Instance => _instance ??= new Singleton();
}
```

Factory Method

Description: Defines an interface for creating objects, letting subclasses decide which to instantiate. Decouples object creation from usage. Great for when you don't know the exact type at compile time.

```
// Product interface
interface IProduct { void DoSomething(); }

// Concrete products
class ConcreteProductA : IProduct { public void DoSomething() => Console.WriteLine("Produ
class ConcreteProductB : IProduct { public void DoSomething() => Console.WriteLine("Produ

// Creator abstract class
abstract class Creator {
    public abstract IProduct FactoryMethod();
    public void SomeOperation() {
        var product = FactoryMethod();
        product.DoSomething();
    }
}

// Concrete creators
class ConcreteCreatorA : Creator {
    public override IProduct FactoryMethod() => new ConcreteProductA();
}
class ConcreteCreatorB : Creator {
    public override IProduct FactoryMethod() => new ConcreteProductB();
}
```

Abstract Factory

Description: Creates families of related objects without specifying concrete classes. Helps ensure compatible product families. Perfect for UI frameworks where you need consistent themes.

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

// Concrete products for Windows
class WinButton : IButton { public void Render() => Console.WriteLine("Windows Button");
class WinCheckbox : ICheckbox { public void Render() => Console.WriteLine("Windows Checkb

// Concrete products for Mac
class MacButton : IButton { public void Render() => Console.WriteLine("Mac Button"); }
class MacCheckbox : ICheckbox { public void Render() => Console.WriteLine("Mac Checkbox")

// Abstract factory
interface IUIFactory {
    IButton CreateButton();
    ICheckbox CreateCheckbox();
}

// Concrete factories
class WinFactory : IUIFactory {
    public IButton CreateButton() => new WinButton();
    public ICheckbox CreateCheckbox() => new WinCheckbox();
}

class MacFactory : IUIFactory {
    public IButton CreateButton() => new MacButton();
    public ICheckbox CreateCheckbox() => new MacCheckbox();
}
```

Builder

Description: Separates construction of complex objects from representation. Allows step-by-step creation of objects with optional parts. Excellent for objects with many optional parameters.

```
// Product to be built
class Car {
    public string Engine { get; set; }
    public string Wheels { get; set; }
    public string Color { get; set; }
    public bool HasGPS { get; set; }
    public bool HasSunroof { get; set; }
}

// Builder interface
interface ICarBuilder {
    ICarBuilder SetEngine(string engine);
    ICarBuilder SetWheels(string wheels);
    ICarBuilder SetColor(string color);
    ICarBuilder SetGPS(bool hasGPS);
    ICarBuilder SetSunroof(bool hasSunroof);
    Car Build();
}

// Concrete builder
class CarBuilder : ICarBuilder {
    private Car _car = new Car();

    public ICarBuilder SetEngine(string engine) { _car.Engine = engine; return this; }
    public ICarBuilder SetWheels(string wheels) { _car.Wheels = wheels; return this; }
    public ICarBuilder SetColor(string color) { _car.Color = color; return this; }
    public ICarBuilder SetGPS(bool hasGPS) { _car.HasGPS = hasGPS; return this; }
    public ICarBuilder SetSunroof(bool hasSunroof) { _car.HasSunroof = hasSunroof; return
    public Car Build() => _car;
}

// Director (optional)
class CarDirector {
    public Car BuildSportsCar(ICarBuilder builder) {
        return builder
            .SetEngine("V8")
            .SetWheels("Sports")
            .SetColor("Red")
            .SetGPS(true)
            .SetSunroof(false)
            .Build();
```

```
    }
}
```

Prototype

Description: Create new objects by cloning existing ones. Useful when object creation is costly. Great for game development where you need many similar objects.

```
abstract class Shape { public abstract Shape Clone(); }
class Circle : Shape {
    public int Radius;
    public override Shape Clone() => (Shape)this.MemberwiseClone();
}
```

2. Structural Patterns (GoF)

Adapter

Description: Converts one interface to another expected by clients. Allows incompatible interfaces to work together. You'll use this constantly when integrating with third-party libraries.

```
// Target interface (what client expects)
interface ITarget { void Request(); }

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

// Adapter (makes Adaptee compatible with ITarget)
class Adapter : ITarget {
    private readonly Adaptee _adaptee = new();
    public void Request() => _adaptee.SpecificRequest();
}

// Client code
class Client {
    public void UseTarget(ITarget target) {
        target.Request();
    }
}
```

Bridge

Description: Decouples abstraction from implementation so both can vary independently. Useful for UI/platform independence. Helps avoid the "explosion of classes" problem.

```
// Implementation interface
interface IDevice {
    void TurnOn();
    void TurnOff();
}

// Concrete implementations
class TV : IDevice {
    public void TurnOn() => Console.WriteLine("TV is ON");
    public void TurnOff() => Console.WriteLine("TV is OFF");
}

class Radio : IDevice {
    public void TurnOn() => Console.WriteLine("Radio is ON");
    public void TurnOff() => Console.WriteLine("Radio is OFF");
}

// Abstraction
abstract class Remote {
    protected IDevice device;
    public Remote(IDevice d) => device = d;
    public abstract void Toggle();
}

// Refined abstraction
class AdvancedRemote : Remote {
    public AdvancedRemote(IDevice d) : base(d) { }
    public override void Toggle() => device.TurnOn();
    public void Mute() => Console.WriteLine("Device muted");
}
```

Composite

Description: Treat individual objects and compositions uniformly. Ideal for tree-like structures (folders, UI elements). Makes complex hierarchies easy to work with.

```
interface IComponent { void Display(); }
class Leaf : IComponent { public void Display() => Console.WriteLine("Leaf"); }
class Composite : IComponent {
    List<IComponent> children = new();
    public void Add(IComponent c) => children.Add(c);
    public void Display() { foreach(var c in children) c.Display(); }
}
```

Decorator

Description: Adds responsibilities to objects dynamically. Supports flexible extension without modifying original class. Great for adding features like logging, caching, or validation.

```
interface INotifier { void Send(string msg); }
class EmailNotifier : INotifier { public void Send(string msg) => Console.WriteLine("Emai
class SMSDecorator : INotifier {
    private readonly INotifier _wrappee;
    public SMSDecorator(INotifier n) => _wrappee = n;
    public void Send(string msg) {
        _wrappee.Send(msg);
        Console.WriteLine("SMS: " + msg);
    }
}
```

Facade

Description: Provides a unified, simple interface to a set of interfaces in a subsystem. Simplifies complex subsystems for clients. Perfect for wrapping complex APIs.

```
class ComputerFacade {
    CPU cpu = new();
    Memory mem = new();
    public void Start() { cpu.Freeze(); mem.Load(); cpu.Execute(); }
}
```

Flyweight

Description: Uses shared objects to reduce memory usage for large numbers of similar objects. Great for text editors, games, or any scenario with many similar objects.

```
class FlyweightFactory {
    private Dictionary<string, Flyweight> _flyweights = new();
    public Flyweight Get(string key) => _flyweights.TryGetValue(key, out var f) ? f : _fl
}
```

Proxy

Description: Provides a placeholder for another object to control access, add lazy loading, caching, or security. Essential for performance optimization and access control.

```
interface IImage { void Display(); }
class RealImage : IImage {
    private string filename;
    public RealImage(string f) { filename = f; LoadFromDisk(); }
    void LoadFromDisk() => Console.WriteLine("Loading " + filename);
    public void Display() => Console.WriteLine("Displaying " + filename);
}
class ProxyImage : IImage {
    private RealImage _real;
    private string _filename;
    public ProxyImage(string f) => _filename = f;
    public void Display() {
        _real ??= new RealImage(_filename);
        _real.Display();
    }
}
```

3. Behavioral Patterns (GoF)

Chain of Responsibility

Description: Pass requests along a chain until one handler processes it. Useful for event processing. Great for middleware pipelines and request handling.

```
abstract class Handler {
    protected Handler next;
    public void SetNext(Handler n) => next = n;
    public abstract void Handle(int r);
}
class ConcreteHandler : Handler {
    public override void Handle(int r) {
        if(r < 10) Console.WriteLine("Handled");
        else next?.Handle(r);
    }
}
```

Command

Description: Encapsulates requests as objects. Enables queuing, logging, and undo/redo operations. Perfect for implementing undo functionality and macro commands.

```
// Receiver
class Light {
    public void On() => Console.WriteLine("Light is ON");
    public void Off() => Console.WriteLine("Light is OFF");
}

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

// Concrete commands
class LightOnCommand : ICommand {
    private Light _light;
    public LightOnCommand(Light l) => _light = l;
    public void Execute() => _light.On();
    public void Undo() => _light.Off();
}

class LightOffCommand : ICommand {
    private Light _light;
    public LightOffCommand(Light l) => _light = l;
    public void Execute() => _light.Off();
    public void Undo() => _light.On();
}

// Invoker
class RemoteControl {
    private ICommand _command;
    public void SetCommand(ICommand command) => _command = command;
    public void PressButton() => _command?.Execute();
    public void PressUndo() => _command?.Undo();
}
```

Interpreter

Description: Defines a grammar and interprets sentences in that language. Useful for DSLs and expression evaluation. Great for rule engines and configuration languages.

```
interface IExpression { int Interpret(Dictionary<string,int> ctx); }
class Number : IExpression {
    int value;
    public Number(int v) => value = v;
    public int Interpret(Dictionary<string,int> ctx) => value;
}
```

Iterator

Description: Provides sequential access to elements without exposing underlying structure. Essential for collections and data structures. Makes traversal algorithms reusable.

```
class MyCollection : IEnumerable<int> {
    List<int> _items = new() {1,2,3};
    public IEnumerator<int> GetEnumerator() => _items.GetEnumerator();
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Mediator

Description: Centralizes communication between objects, reducing direct dependencies. Great for complex UI interactions and chat systems. Reduces coupling between components.

```
// Mediator interface
interface IChatMediator {
    void Register(User user);
    void Send(string message, User sender);
}

// Concrete mediator
class ChatMediator : IChatMediator {
    private List<User> users = new();

    public void Register(User user) => users.Add(user);

    public void Send(string message, User sender) {
        foreach (var user in users) {
            if (user != sender) {
                user.Receive(message);
            }
        }
    }
}

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

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

    public void Send(string message) {
        Console.WriteLine($"{_name} sends: {message}");
        _mediator.Send(message, this);
    }

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

Memento

Description: Captures and restores object state without violating encapsulation. Supports undo functionality. Perfect for implementing save/restore features.

```
class Memento {
    public string State { get; }
    public Memento(string s) => State = s;
}
class Originator {
    public string State;
    public Memento Save() => new(State);
    public void Restore(Memento m) => State = m.State;
}
```

Observer

Description: Defines a one-to-many dependency: when one object changes state, all dependents are notified. Foundation of event-driven programming. Essential for MVC and reactive systems.

```
class Subject {
    List<IObserver> observers = new();
    public void Attach(IObserver o) => observers.Add(o);
    public void Notify() { foreach(var o in observers) o.Update(); }
}
```

State

Description: Allows an object to alter behavior when its internal state changes. Perfect for state machines and complex conditional logic. Makes state transitions explicit.

```
interface IState { void Handle(Context c); }
class Context {
    public IState State;
    public void Request() => State.Handle(this);
}
```

Strategy

Description: Encapsulates interchangeable algorithms. Clients can switch strategies at runtime. Great for sorting algorithms, payment methods, or any interchangeable behavior.

```
interface IStrategy { void Execute(); }
class QuickSort : IStrategy { public void Execute() => Console.WriteLine("QuickSort"); }
class Sorter {
    IStrategy s;
    public Sorter(IStrategy s) => this.s = s;
    public void Sort() => s.Execute();
}
```

Template Method

Description: Defines skeleton of algorithm, deferring certain steps to subclasses. Great for frameworks and libraries. Ensures consistent algorithm structure.

```
abstract class Game {
    public void Play() { Init(); Start(); End(); }
    protected abstract void Init();
    protected abstract void Start();
    protected abstract void End();
}
```

Visitor

Description: Represents an operation to be performed on object structures without modifying the objects themselves. Great for adding operations to existing class hierarchies.

```
interface IVisitor { void Visit(Element e); }
abstract class Element { public abstract void Accept(IVisitor v); }
class ConcreteElement : Element {
    public override void Accept(IVisitor v) => v.Visit(this);
}
```

4. Modern Extensions

Null Object

Description: Provides a do-nothing implementation instead of null, avoiding null checks. Eliminates null pointer exceptions and makes code more robust.

```
interface ILogger { void Log(string msg); }
class ConsoleLogger : ILogger { public void Log(string msg) => Console.WriteLine(msg); }
class NullLogger : ILogger { public void Log(string msg) { } }
```

MVC (Model-View-Controller)

Description: Separates application logic into Model (data), View (UI), Controller (input). Promotes separation of concerns. The foundation of most web frameworks.

```
class User { public string Name { get; set; } }
class UserView { public void Render(User u) => Console.WriteLine($"Hello {u.Name}"); }
class UserController {
    User model;
    UserView view;
    public UserController(User m, UserView v) { model = m; view = v; }
    public void UpdateView() => view.Render(model);
}
```

Dependency Injection / Service Locator

Description: Externalizes object creation and dependency resolution. Improves flexibility and testability. Essential for modern application architecture.

```
class ServiceLocator {
    private static Dictionary<Type, object> _services = new();
    public static void Register<T>(T service) => _services[typeof(T)] = service;
    public static T Get<T>() => (T)_services[typeof(T)];
}
```

Repository / DAO

Description: Abstracts data layer, providing a clean interface for CRUD operations. Makes data access testable and allows easy switching between data sources.

```
// Domain entity
class User {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}

// Repository interface
interface IUserRepository {
    User GetById(int id);
    IEnumerable<User> GetAll();
    void Save(User user);
    void Delete(int id);
}

// Concrete repository
class UserRepository : IUserRepository {
    private readonly List<User> _users = new();

    public User GetById(int id) {
        return _users.FirstOrDefault(u => u.Id == id);
    }

    public IEnumerable<User> GetAll() {
        return _users.ToList();
    }

    public void Save(User user) {
        var existing = GetById(user.Id);
        if (existing != null) {
            _users.Remove(existing);
        }
        _users.Add(user);
    }

    public void Delete(int id) {
        var user = GetById(id);
        if (user != null) {
            _users.Remove(user);
        }
    }
}
```

MVP (Model-View-Presenter)

Description: Similar to MVC but with tighter coupling between View and Presenter. The Presenter handles all UI logic. Great for Windows Forms and desktop applications.

```
// Model
class UserModel {
    public string Name { get; set; }
    public string Email { get; set; }
}

// View interface
interface IUserView {
    string UserName { get; set; }
    string UserEmail { get; set; }
    void ShowMessage(string message);
}

// Presenter
class UserPresenter {
    private readonly IUserView _view;
    private readonly UserModel _model;

    public UserPresenter(IUserView view, UserModel model) {
        _view = view;
        _model = model;
    }

    public void LoadUser() {
        _view.UserName = _model.Name;
        _view.UserEmail = _model.Email;
    }

    public void SaveUser() {
        _model.Name = _view.UserName;
        _model.Email = _view.UserEmail;
        _view.ShowMessage("User saved successfully!");
    }
}
```

MVVM (Model-View-ViewModel)

Description: Designed for WPF and modern UI frameworks. Uses data binding and commands. The ViewModel acts as a data converter and command handler.

```
// Model
class User {
    public string Name { get; set; }
    public string Email { get; set; }
}

// ViewModel
class UserViewModel : INotifyPropertyChanged {
    private User _user;
    private string _name;
    private string _email;

    public string Name {
        get => _name;
        set {
            _name = value;
            OnPropertyChanged();
        }
    }

    public string Email {
        get => _email;
        set {
            _email = value;
            OnPropertyChanged();
        }
    }

    public ICommand SaveCommand { get; }

    public UserViewModel(User user) {
        _user = user;
        _name = user.Name;
        _email = user.Email;
        SaveCommand = new RelayCommand(Save);
    }

    private void Save() {
        _user.Name = Name;
        _user.Email = Email;
        // Save logic here
    }
```

```
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = nul
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

CQRS (Command Query Responsibility Segregation)

Description: Separates read and write operations. Uses different models for commands and queries. Great for high-performance systems with complex business logic.

```
// Command side
interface ICommand { }

class CreateUserCommand : ICommand {
    public string Name { get; set; }
    public string Email { get; set; }
}

interface ICommandHandler<T> where T : ICommand {
    Task Handle(T command);
}

class CreateUserCommandHandler : ICommandHandler<CreateUserCommand> {
    public async Task Handle(CreateUserCommand command) {
        // Write to database
        Console.WriteLine($"Creating user: {command.Name}");
        await Task.Delay(100);
    }
}

// Query side
interface IQuery<TResult> { }

class GetUserQuery : IQuery<UserDto> {
    public int UserId { get; set; }
}

interface IQueryHandler<TQuery, TResult> where TQuery : IQuery<TResult> {
    Task<TResult> Handle(TQuery query);
}

class GetUserQueryHandler : IQueryHandler<GetUserQuery, UserDto> {
    public async Task<UserDto> Handle(GetUserQuery query) {
        // Read from database
        await Task.Delay(50);
        return new UserDto { Id = query.UserId, Name = "User Name" };
    }
}

class UserDto {
    public int Id { get; set; }
```

```
    public string Name { get; set; }
}
```

Event Sourcing

Description: Stores changes as a sequence of events rather than current state. Enables audit trails and time travel. Perfect for financial systems and audit requirements.

```
// Domain event
abstract class DomainEvent {
    public DateTime OccurredAt { get; } = DateTime.UtcNow;
    public Guid Id { get; } = Guid.NewGuid();
}

class UserCreatedEvent : DomainEvent {
    public string Name { get; set; }
    public string Email { get; set; }
}

class UserEmailChangedEvent : DomainEvent {
    public string OldEmail { get; set; }
    public string NewEmail { get; set; }
}

// Aggregate root
class User {
    private readonly List<DomainEvent> _events = new();

    public Guid Id { get; private set; }
    public string Name { get; private set; }
    public string Email { get; private set; }

    public static User Create(string name, string email) {
        var user = new User();
        user.Apply(new UserCreatedEvent { Name = name, Email = email });
        return user;
    }

    public void ChangeEmail(string newEmail) {
        var oldEmail = Email;
        Apply(new UserEmailChangedEvent { OldEmail = oldEmail, NewEmail = newEmail });
    }

    private void Apply(DomainEvent evt) {
        _events.Add(evt);
        // Apply event to current state
        switch (evt) {
            case UserCreatedEvent e:
                Id = e.Id;
                Name = e.Name;
                Email = e.Email;
```

```
                break;
            case UserEmailChangedEvent e:
                Email = e.NewEmail;
                break;
        }
    }

    public IEnumerable<DomainEvent> GetUncommittedEvents() => _events;
    public void MarkEventsAsCommitted() => _events.Clear();
}
```

5. Additional Patterns

Object Pool

Description: Reuses objects that are expensive to create (e.g., database connections). Reduces creation overhead. Essential for high-performance applications.

```
class ObjectPool<T> where T : new() {
    Stack<T> items = new();
    public T Get() => items.Count > 0 ? items.Pop() : new T();
    public void Return(T obj) => items.Push(obj);
}
```

Lazy Initialization

Description: Delays object creation until actually needed. Saves memory and computation. Great for expensive resources that might not be used.

```
Lazy<Expensive> obj = new(() => new Expensive());
```

Multiton

Description: Like Singleton, but allows a limited number of instances keyed by a value. Useful for resource management with multiple instances.

```
class Multiton {
    private static Dictionary<string, Multiton> _instances = new();
    private Multiton() {}
    public static Multiton Get(string key) => _instances.TryGetValue(key,out var m) ? m :
}
```

Pipeline

Description: Processes data sequentially through stages. Often used in streaming or functional pipelines. Great for data transformation workflows.

```
var pipeline = new List<Func<int,int>> { x => x+1, x => x*2 };
int result = pipeline.Aggregate(1, (acc,f) => f(acc));
```

Event Bus / Event Aggregator

Description: Decouples publishers and subscribers. Enables scalable event-driven architecture. Perfect for microservices communication.

```
class EventBus {
    public event Action<string> OnMessage;
    public void Publish(string msg) => OnMessage?.Invoke(msg);
}
```

Specification

Description: Encapsulates business rules into reusable predicates. Allows combining rules using logical operators. Great for complex business logic.

```
interface ISpec<T> { bool IsSatisfied(T item); }
class EvenSpec : ISpec<int> { public bool IsSatisfied(int i) => i%2==0; }
```

6. Functional Programming Patterns

Monad

Description: Encapsulates computations with side effects in a composable way. Essential for functional programming and error handling. Makes error handling explicit and composable.

```
// Maybe/Option monad
class Maybe<T> {
    private readonly T _value;
    private readonly bool _hasValue;
    private Maybe(T value, bool hasValue) { _value = value; _hasValue = hasValue; }
    public static Maybe<T> Some(T value) => new(value, true);
    public static Maybe<T> None() => new(default, false);
    public Maybe<U> Bind<U>(Func<T, Maybe<U>> f) => _hasValue ? f(_value) : Maybe<U>.None
    public Maybe<U> Map<U>(Func<T, U> f) => _hasValue ? Maybe<U>.Some(f(_value)) : Maybe<
}
```

Functor

Description: Provides a way to apply functions to values wrapped in a context (like Option, List). Enables functional composition. Foundation of functional programming.

```
interface IFunctor<T> {
    IFunctor<U> Map<U>(Func<T, U> f);
}
class ListFunctor<T> : IFunctor<T> {
    private readonly List<T> _items;
    public ListFunctor(List<T> items) => _items = items;
    public IFunctor<U> Map<U>(Func<T, U> f) => new ListFunctor<U>(_items.Select(f).ToList
}
```

Fold/Reduce

Description: Aggregates collections into single values. Fundamental for functional data processing. Essential for data analysis and transformation.

```
static class FunctionalExtensions {
    public static U Fold<T, U>(this IEnumerable<T> source, U seed, Func<U, T, U> func) {
        return source.Aggregate(seed, func);
    }
}
// Usage: var sum = numbers.Fold(0, (acc, x) => acc + x);
```

7. Concurrency & Parallelism Patterns

Actor Model

Description: Objects communicate through message passing. Excellent for concurrent systems and distributed computing. Eliminates shared state issues.

```
interface IActor { void Receive(object message); }
class SimpleActor : IActor {
    public void Receive(object message) {
        switch (message) {
            case string msg: Console.WriteLine($"Received: {msg}"); break;
            case int num: Console.WriteLine($"Number: {num}"); break;
        }
    }
}
class ActorSystem {
    private readonly Dictionary<string, IActor> _actors = new();
    public void Register(string name, IActor actor) => _actors[name] = actor;
    public void Send(string actorName, object message) => _actors[actorName]?.Receive(mes
}
```

Producer-Consumer

Description: Decouples data production from consumption using queues. Essential for asynchronous processing. Great for background processing and data pipelines.

```
class ProducerConsumer<T> {
    private readonly Queue<T> _queue = new();
    private readonly object _lock = new();
    public void Produce(T item) {
        lock (_lock) {
            _queue.Enqueue(item);
            Monitor.Pulse(_lock);
        }
    }
    public T Consume() {
        lock (_lock) {
            while (_queue.Count == 0) Monitor.Wait(_lock);
            return _queue.Dequeue();
        }
    }
}
```

Thread Pool

Description: Manages a pool of worker threads to execute tasks. Improves performance and resource management. Essential for scalable server applications.

```
class ThreadPool {
    private readonly Queue<Action> _tasks = new();
    private readonly List<Thread> _workers = new();
    private volatile bool _shutdown = false;

    public ThreadPool(int workerCount) {
        for (int i = 0; i < workerCount; i++) {
            var worker = new Thread(WorkerLoop) { IsBackground = true };
            worker.Start();
            _workers.Add(worker);
        }
    }

    public void QueueTask(Action task) {
        lock (_tasks) {
            _tasks.Enqueue(task);
            Monitor.Pulse(_tasks);
        }
    }

    private void WorkerLoop() {
        while (!_shutdown) {
            Action task = null;
            lock (_tasks) {
                while (_tasks.Count == 0 && !_shutdown) Monitor.Wait(_tasks);
                if (_tasks.Count > 0) task = _tasks.Dequeue();
            }
            task?.Invoke();
        }
    }
}
```

8. Reactive Programming Patterns

Circuit Breaker

Description: Prevents cascading failures by temporarily stopping calls to failing services. Critical for microservices. Essential for resilient distributed systems.

```
enum CircuitState { Closed, Open, HalfOpen }
class CircuitBreaker {
    private CircuitState _state = CircuitState.Closed;
    private int _failureCount = 0;
    private DateTime _lastFailureTime;
    private readonly int _threshold;
    private readonly TimeSpan _timeout;

    public CircuitBreaker(int threshold, TimeSpan timeout) {
        _threshold = threshold;
        _timeout = timeout;
    }

    public T Execute<T>(Func<T> operation) {
        if (_state == CircuitState.Open) {
            if (DateTime.Now - _lastFailureTime > _timeout) {
                _state = CircuitState.HalfOpen;
            } else {
                throw new InvalidOperationException("Circuit breaker is open");
            }
        }

        try {
            var result = operation();
            OnSuccess();
            return result;
        } catch (Exception ex) {
            OnFailure();
            throw;
        }
    }

    private void OnSuccess() {
        _failureCount = 0;
        _state = CircuitState.Closed;
    }

    private void OnFailure() {
        _failureCount++;
        _lastFailureTime = DateTime.Now;
        if (_failureCount >= _threshold) {
            _state = CircuitState.Open;
        }
```

```
    }
}
```

Saga

Description: Manages distributed transactions across multiple services. Essential for microservices architectures. Provides eventual consistency in distributed systems.

```
interface ISagaStep {
    Task Execute();
    Task Compensate();
}
class OrderSaga {
    private readonly List<ISagaStep> _steps = new();
    private readonly List<ISagaStep> _completedSteps = new();

    public void AddStep(ISagaStep step) => _steps.Add(step);

    public async Task Execute() {
        try {
            foreach (var step in _steps) {
                await step.Execute();
                _completedSteps.Add(step);
            }
        } catch (Exception) {
            await Compensate();
            throw;
        }
    }

    private async Task Compensate() {
        foreach (var step in _completedSteps.AsEnumerable().Reverse()) {
            await step.Compensate();
        }
    }
}
```

9. Cloud & Distributed Patterns

API Gateway

Description: Single entry point for client requests to microservices. Simplifies client-server communication. Essential for microservices architecture.

```
interface IMicroservice {
    Task<string> ProcessRequest(string request);
}
class APIGateway {
    private readonly Dictionary<string, IMicroservice> _services = new();

    public void RegisterService(string path, IMicroservice service) {
        _services[path] = service;
    }

    public async Task<string> RouteRequest(string path, string request) {
        if (_services.TryGetValue(path, out var service)) {
            return await service.ProcessRequest(request);
        }
        throw new ArgumentException($"No service found for path: {path}");
    }
}
```

Service Mesh

Description: Handles service-to-service communication in microservices. Provides observability and security. Essential for production microservices deployments.

```
interface IServiceMesh {
    Task<T> CallService<T>(string serviceName, string endpoint, object data);
}
class ServiceMesh : IServiceMesh {
    private readonly Dictionary<string, string> _serviceUrls = new();

    public void RegisterService(string name, string url) {
        _serviceUrls[name] = url;
    }

    public async Task<T> CallService<T>(string serviceName, string endpoint, object data)
        if (!_serviceUrls.TryGetValue(serviceName, out var url)) {
            throw new ArgumentException($"Service {serviceName} not found");
        }

        // Simulate HTTP call with retry logic, circuit breaker, etc.
        var fullUrl = $"{url}/{endpoint}";
        Console.WriteLine($"Calling {fullUrl} with data: {data}");

        // Return mock response
        return default(T);
    }
}
```

10. Data Access Patterns

Unit of Work

Description: Maintains a list of objects affected by a business transaction. Ensures data consistency. Essential for maintaining transactional integrity.

```
interface IUnitOfWork {
    void RegisterNew<T>(T entity);
    void RegisterDirty<T>(T entity);
    void RegisterDeleted<T>(T entity);
    Task Commit();
    void Rollback();
}
class UnitOfWork : IUnitOfWork {
    private readonly List<object> _newEntities = new();
    private readonly List<object> _dirtyEntities = new();
    private readonly List<object> _deletedEntities = new();

    public void RegisterNew<T>(T entity) => _newEntities.Add(entity);
    public void RegisterDirty<T>(T entity) => _dirtyEntities.Add(entity);
    public void RegisterDeleted<T>(T entity) => _deletedEntities.Add(entity);

    public async Task Commit() {
        // Save all changes atomically
        Console.WriteLine("Committing all changes...");
        await Task.Delay(100); // Simulate database operations
    }

    public void Rollback() {
        _newEntities.Clear();
        _dirtyEntities.Clear();
        _deletedEntities.Clear();
        Console.WriteLine("Rolled back all changes");
    }
}
```

Identity Map

Description: Ensures each object is loaded only once per session. Prevents duplicate objects in memory. Essential for ORM frameworks and data consistency.

```
class IdentityMap<T> where T : class {
    private readonly Dictionary<object, T> _map = new();

    public T Get(object key) {
        return _map.TryGetValue(key, out var entity) ? entity : null;
    }

    public void Put(object key, T entity) {
        _map[key] = entity;
    }

    public bool Contains(object key) => _map.ContainsKey(key);
    public void Remove(object key) => _map.Remove(key);
    public void Clear() => _map.Clear();
}
```

11. Security Patterns

Authentication

Description: Verifies user identity. Fundamental security requirement. Essential for any system that needs to identify users.

```
interface IAuthenticationService {
    Task<bool> AuthenticateAsync(string username, string password);
    Task<string> GenerateTokenAsync(string username);
}
class AuthenticationService : IAuthenticationService {
    public async Task<bool> AuthenticateAsync(string username, string password) {
        // Simulate authentication logic
        await Task.Delay(100);
        return username == "admin" && password == "password";
    }

    public async Task<string> GenerateTokenAsync(string username) {
        await Task.Delay(50);
        return $"token_{username}_{DateTime.Now.Ticks}";
    }
}
```

Authorization

Description: Controls access to resources based on user permissions. Essential for secure systems. Implements the principle of least privilege.

```
interface IAuthorizationService {
    Task<bool> IsAuthorizedAsync(string user, string resource, string action);
}
class AuthorizationService : IAuthorizationService {
    private readonly Dictionary<string, List<string>> _permissions = new();

    public AuthorizationService() {
        _permissions["admin"] = new List<string> { "read", "write", "delete" };
        _permissions["user"] = new List<string> { "read" };
    }

    public async Task<bool> IsAuthorizedAsync(string user, string resource, string action
        await Task.Delay(10);
        return _permissions.TryGetValue(user, out var userPermissions) &&
               userPermissions.Contains(action);
    }
}
```

12. Performance Patterns

Caching

Description: Stores frequently accessed data in fast storage. Improves application performance. Essential for high-performance applications.

```
interface ICache<T> {
    T Get(string key);
    void Set(string key, T value, TimeSpan? expiry = null);
    void Remove(string key);
    void Clear();
}
class MemoryCache<T> : ICache<T> {
    private readonly Dictionary<string, CacheItem<T>> _cache = new();

    private class CacheItem<U> {
        public U Value { get; set; }
        public DateTime Expiry { get; set; }
        public bool IsExpired => DateTime.Now > Expiry;
    }

    public T Get(string key) {
        if (_cache.TryGetValue(key, out var item) && !item.IsExpired) {
            return item.Value;
        }
        return default(T);
    }

    public void Set(string key, T value, TimeSpan? expiry = null) {
        _cache[key] = new CacheItem<T> {
            Value = value,
            Expiry = DateTime.Now.Add(expiry ?? TimeSpan.FromMinutes(5))
        };
    }

    public void Remove(string key) => _cache.Remove(key);
    public void Clear() => _cache.Clear();
}
```

Connection Pooling

Description: Reuses database connections. Improves performance and resource utilization. Essential for database-intensive applications.

```
class ConnectionPool {
    private readonly Queue<IDbConnection> _availableConnections = new();
    private readonly List<IDbConnection> _allConnections = new();
    private readonly int _maxConnections;

    public ConnectionPool(int maxConnections) {
        _maxConnections = maxConnections;
    }

    public IDbConnection GetConnection() {
        lock (_availableConnections) {
            if (_availableConnections.Count > 0) {
                return _availableConnections.Dequeue();
            }

            if (_allConnections.Count < _maxConnections) {
                var connection = CreateConnection();
                _allConnections.Add(connection);
                return connection;
            }

            throw new InvalidOperationException("No connections available");
        }
    }

    public void ReturnConnection(IDbConnection connection) {
        lock (_availableConnections) {
            _availableConnections.Enqueue(connection);
        }
    }

    private IDbConnection CreateConnection() {
        // Create actual database connection
        return null; // Placeholder
    }
}
```

13. Testing Patterns

Test Double

Description: Replaces real objects with test-specific objects. Essential for unit testing. Enables isolated testing of components.

```
interface IUserService {
    Task<User> GetUserAsync(int id);
}
class UserService : IUserService {
    public async Task<User> GetUserAsync(int id) {
        // Real implementation
        await Task.Delay(100);
        return new User { Id = id, Name = "Real User" };
    }
}
class TestUserService : IUserService {
    private readonly Dictionary<int, User> _users = new();

    public void AddUser(User user) => _users[user.Id] = user;

    public Task<User> GetUserAsync(int id) {
        return Task.FromResult(_users.TryGetValue(id, out var user) ? user : null);
    }
}
```

Mock Object

Description: Simulates behavior of real objects for testing. Enables isolated testing. Essential for testing interactions between objects.

```
class MockEmailService : IEmailService {
    public List<string> SentEmails { get; } = new();
    public bool ShouldThrowException { get; set; } = false;

    public Task SendEmailAsync(string to, string subject, string body) {
        if (ShouldThrowException) {
            throw new InvalidOperationException("Email service unavailable");
        }

        SentEmails.Add($"To: {to}, Subject: {subject}");
        return Task.CompletedTask;
    }
}
```

14. Modern Architectural Patterns

Clean Architecture

Description: Separates concerns into layers with dependency inversion. Promotes maintainable code. Essential for large, complex applications.

```
// Domain Layer (Core)
public class User {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}

public interface IUserRepository {
    Task<User> GetByIdAsync(int id);
    Task SaveAsync(User user);
}

// Application Layer
public class UserService {
    private readonly IUserRepository _userRepository;

    public UserService(IUserRepository userRepository) {
        _userRepository = userRepository;
    }

    public async Task<User> GetUserAsync(int id) {
        return await _userRepository.GetByIdAsync(id);
    }
}

// Infrastructure Layer
public class SqlUserRepository : IUserRepository {
    public async Task<User> GetByIdAsync(int id) {
        // Database implementation
        await Task.Delay(100);
        return new User { Id = id, Name = "User from DB" };
    }

    public async Task SaveAsync(User user) {
        // Database save implementation
        await Task.Delay(50);
    }
}
```

Domain-Driven Design (DDD)

Description: Focuses on business domain modeling. Improves software design. Essential for complex business applications.

```
// Domain Entity
public class Order {
    private readonly List<OrderItem> _items = new();

    public int Id { get; private set; }
    public Customer Customer { get; private set; }
    public OrderStatus Status { get; private set; }
    public decimal TotalAmount => _items.Sum(item => item.Price * item.Quantity);

    public void AddItem(Product product, int quantity) {
        if (Status != OrderStatus.Draft) {
            throw new InvalidOperationException("Cannot modify confirmed order");
        }

        _items.Add(new OrderItem(product, quantity));
    }

    public void Confirm() {
        if (_items.Count == 0) {
            throw new InvalidOperationException("Cannot confirm empty order");
        }

        Status = OrderStatus.Confirmed;
    }
}

// Domain Service
public class OrderPricingService {
    public decimal CalculateDiscount(Order order) {
        if (order.TotalAmount > 1000) {
            return order.TotalAmount * 0.1m; // 10% discount
        }
        return 0;
    }
}
```

Serverless / Function-Based Patterns

Description: Event-driven, stateless functions that scale automatically. Perfect for microservices and cloud-native applications. Essential for modern cloud architectures.

```
// Function-based pattern using Azure Functions
public static class OrderProcessingFunction {
    [FunctionName("ProcessOrder")]
    public static async Task<IActionResult> ProcessOrder(
        [HttpTrigger(AuthorizationLevel.Function, "post", Route = "orders")] HttpRequest
        [ServiceBus("order-queue", Connection = "ServiceBusConnection")] IAsyncCollector<
        ILogger log) {

        try {
            string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
            var order = JsonSerializer.Deserialize<Order>(requestBody);

            // Process order
            await ProcessOrderLogic(order);

            // Send to queue for further processing
            await queueCollector.AddAsync(JsonSerializer.Serialize(order));

            return new OkObjectResult(new { OrderId = order.Id, Status = "Processed" });
        } catch (Exception ex) {
            log.LogError(ex, "Error processing order");
            return new BadRequestObjectResult(new { Error = ex.Message });
        }
    }

    private static async Task ProcessOrderLogic(Order order) {
        // Stateless processing logic
        await Task.Delay(100); // Simulate processing
    }
}

// Event-driven function
[FunctionName("OrderEventHandler")]
public static async Task OrderEventHandler(
    [ServiceBusTrigger("order-queue", Connection = "ServiceBusConnection")] string orderJ
    [CosmosDB("orders", "processed", ConnectionStringSetting = "CosmosDBConnection")] IAs
    ILogger log) {

    var order = JsonSerializer.Deserialize<Order>(orderJson);

    // Process the order event
    var processedOrder = new ProcessedOrder {
        OrderId = order.Id,
```

```
        ProcessedAt = DateTime.UtcNow,
        Status = "Completed"
    };

    await collector.AddAsync(processedOrder);
    log.LogInformation($"Order {order.Id} processed successfully");
}
```

Hexagonal Architecture (Ports and Adapters)

Description: Separates business logic from external concerns using ports and adapters. Makes applications testable and technology-agnostic. Essential for maintainable enterprise applications.

```
// Domain (Core) - Business Logic
public class Order {
    public int Id { get; private set; }
    public decimal Amount { get; private set; }
    public OrderStatus Status { get; private set; }

    public void Process() {
        if (Amount <= 0) throw new InvalidOperationException("Invalid amount");
        Status = OrderStatus.Processing;
    }

    public void Complete() {
        Status = OrderStatus.Completed;
    }
}

// Ports (Interfaces) - Define contracts
public interface IOrderRepository {
    Task<Order> GetByIdAsync(int id);
    Task SaveAsync(Order order);
}

public interface IPaymentService {
    Task<bool> ProcessPaymentAsync(decimal amount, string cardNumber);
}

public interface INotificationService {
    Task SendNotificationAsync(string message, string recipient);
}

// Application Services - Orchestrate business logic
public class OrderService {
    private readonly IOrderRepository _orderRepository;
    private readonly IPaymentService _paymentService;
    private readonly INotificationService _notificationService;

    public OrderService(
        IOrderRepository orderRepository,
        IPaymentService paymentService,
        INotificationService notificationService) {
        _orderRepository = orderRepository;
        _paymentService = paymentService;
        _notificationService = notificationService;
```

```
    }

    public async Task ProcessOrderAsync(int orderId, string cardNumber) {
        var order = await _orderRepository.GetByIdAsync(orderId);

        order.Process();

        var paymentSuccess = await _paymentService.ProcessPaymentAsync(order.Amount, cardN

        if (paymentSuccess) {
            order.Complete();
            await _notificationService.SendNotificationAsync("Order completed", "customer@
        }

        await _orderRepository.SaveAsync(order);
    }
}

// Adapters (Infrastructure) - Implement ports
public class SqlOrderRepository : IOrderRepository {
    public async Task<Order> GetByIdAsync(int id) {
        // Database implementation
        await Task.Delay(100);
        return new Order { Id = id, Amount = 100 };
    }

    public async Task SaveAsync(Order order) {
        // Database save implementation
        await Task.Delay(50);
    }
}

public class StripePaymentService : IPaymentService {
    public async Task<bool> ProcessPaymentAsync(decimal amount, string cardNumber) {
        // Stripe API implementation
        await Task.Delay(200);
        return true; // Simulate successful payment
    }
}

public class EmailNotificationService : INotificationService {
    public async Task SendNotificationAsync(string message, string recipient) {
        // Email service implementation
```

```
        await Task.Delay(100);
        Console.WriteLine($"Email sent to {recipient}: {message}");
    }
}

// Primary Adapters (Controllers, APIs)
[ApiController]
[Route("api/[controller]")]
public class OrdersController : ControllerBase {
    private readonly OrderService _orderService;

    public OrdersController(OrderService orderService) {
        _orderService = orderService;
    }

    [HttpPost("{id}/process")]
    public async Task<IActionResult> ProcessOrder(int id, [FromBody] ProcessOrderRequest
        try {
            await _orderService.ProcessOrderAsync(id, request.CardNumber);
            return Ok(new { Message = "Order processed successfully" });
        } catch (Exception ex) {
            return BadRequest(new { Error = ex.Message });
        }
    }
}
```

Microkernel Architecture

Description: Core system with pluggable modules. Allows dynamic loading and unloading of functionality. Great for extensible applications and plugin systems.

```
// Core system
public interface IPlugin {
    string Name { get; }
    string Version { get; }
    void Initialize();
    void Execute();
    void Shutdown();
}

public class PluginManager {
    private readonly Dictionary<string, IPlugin> _plugins = new();
    private readonly List<IPlugin> _loadedPlugins = new();

    public void LoadPlugin(IPlugin plugin) {
        if (_plugins.ContainsKey(plugin.Name)) {
            throw new InvalidOperationException($"Plugin {plugin.Name} already loaded");
        }

        plugin.Initialize();
        _plugins[plugin.Name] = plugin;
        _loadedPlugins.Add(plugin);
        Console.WriteLine($"Plugin {plugin.Name} v{plugin.Version} loaded");
    }

    public void UnloadPlugin(string pluginName) {
        if (_plugins.TryGetValue(pluginName, out var plugin)) {
            plugin.Shutdown();
            _plugins.Remove(pluginName);
            _loadedPlugins.Remove(plugin);
            Console.WriteLine($"Plugin {pluginName} unloaded");
        }
    }

    public void ExecutePlugin(string pluginName) {
        if (_plugins.TryGetValue(pluginName, out var plugin)) {
            plugin.Execute();
        }
    }

    public void ExecuteAllPlugins() {
        foreach (var plugin in _loadedPlugins) {
            plugin.Execute();
        }
```

```
    }

    public IEnumerable<string> GetLoadedPlugins() {
        return _plugins.Keys;
    }
}

// Plugin implementations
public class LoggingPlugin : IPlugin {
    public string Name => "Logging Plugin";
    public string Version => "1.0.0";

    public void Initialize() {
        Console.WriteLine("Logging plugin initialized");
    }

    public void Execute() {
        Console.WriteLine("Logging: Application event recorded");
    }

    public void Shutdown() {
        Console.WriteLine("Logging plugin shutdown");
    }
}

public class SecurityPlugin : IPlugin {
    public string Name => "Security Plugin";
    public string Version => "2.1.0";

    public void Initialize() {
        Console.WriteLine("Security plugin initialized");
    }

    public void Execute() {
        Console.WriteLine("Security: Access control check performed");
    }

    public void Shutdown() {
        Console.WriteLine("Security plugin shutdown");
    }
}

public class AnalyticsPlugin : IPlugin {
```

```
    public string Name => "Analytics Plugin";
    public string Version => "1.5.0";

    public void Initialize() {
        Console.WriteLine("Analytics plugin initialized");
    }

    public void Execute() {
        Console.WriteLine("Analytics: User behavior tracked");
    }

    public void Shutdown() {
        Console.WriteLine("Analytics plugin shutdown");
    }
}

// Core application
public class MicrokernelApplication {
    private readonly PluginManager _pluginManager;

    public MicrokernelApplication() {
        _pluginManager = new PluginManager();
    }

    public void Start() {
        Console.WriteLine("Microkernel Application Starting...");

        // Load core plugins
        _pluginManager.LoadPlugin(new LoggingPlugin());
        _pluginManager.LoadPlugin(new SecurityPlugin());

        Console.WriteLine("Core plugins loaded");
    }

    public void LoadOptionalPlugin(IPlugin plugin) {
        _pluginManager.LoadPlugin(plugin);
    }

    public void ProcessRequest() {
        _pluginManager.ExecuteAllPlugins();
    }

    public void Shutdown() {
```

```
        var loadedPlugins = _pluginManager.GetLoadedPlugins().ToList();
        foreach (var pluginName in loadedPlugins) {
            _pluginManager.UnloadPlugin(pluginName);
        }
        Console.WriteLine("Application shutdown complete");
    }
}
```

Quick Decision Guide

By Problem Type:

Object Creation Issues

- **Need single instance?** → Singleton
- **Complex object building?** → Builder
- **Family of related objects?** → Abstract Factory
- **Expensive object creation?** → Object Pool / Prototype

Structure & Organization

- **Incompatible interfaces?** → Adapter
- **Tree-like structures?** → Composite
- **Add features dynamically?** → Decorator
- **Simplify complex system?** → Facade
- **Memory optimization?** → Flyweight
- **Control access?** → Proxy

Behavior & Communication

- **Loose coupling needed?** → Observer
- **Multiple algorithms?** → Strategy
- **Undo/Redo functionality?** → Command
- **State-dependent behavior?** → State
- **Request processing chain?** → Chain of Responsibility
- **Sequential data access?** → Iterator
- **External operations?** → Visitor

Architecture & Scale

- **Web application?** → MVC/MVP/MVVM
- **High performance reads?** → CQRS
- **Audit requirements?** → Event Sourcing
- **Testability?** → Hexagonal Architecture
- **Scalability?** → Microservices
- **API management?** → API Gateway

Modern Challenges

- **Concurrency?** → Actor Model / Thread Pool
- **Async processing?** → Producer-Consumer
- **Fault tolerance?** → Circuit Breaker
- **Reactive data?** → Reactive Streams
- **Service communication?** → Service Mesh
- **Serverless?** → Function-based patterns

Pattern Selection Matrix

Scenario	Primary Pattern	Secondary Pattern	When to Use
Web App	MVC	Repository	Standard web applications
API Service	API Gateway	Circuit Breaker	Microservices architecture
Data Processing	Pipeline	Producer-Consumer	Stream processing
Game Development	State	Observer	Game state management
E-commerce	CQRS	Event Sourcing	High-performance systems
Mobile App	MVVM	Repository	Cross-platform development
Legacy Integration	Adapter	Facade	System integration

Scenario	Primary Pattern	Secondary Pattern	When to Use
Real-time System	Observer	Reactive Streams	Live data updates

Final Decision Checklist

Before Choosing a Pattern:

1. **Identify the core problem**
   - Creation, Structure, or Behavior?
2. **Consider the context**
   - Web app, API, Desktop, Mobile?
3. **Evaluate complexity**
   - Simple solution vs. Over-engineering?
4. **Think about future changes**
   - Will this pattern help or hinder?
5. **Consider team familiarity**
   - Can the team maintain this?
6. **Check performance implications**
   - Does it add overhead?

Red Flags:

- **Over-engineering** simple problems
- **Using patterns** just because they exist
- **Ignoring** simpler solutions
- **Not considering** maintenance costs
- **Forgetting** about team expertise

15. Pattern Anti-Patterns

God Object

Problem: When Singleton becomes a monster that knows and does everything.
Symptoms: Single class with 1000+ lines, handles multiple responsibilities.
Solution: Break into smaller, focused classes using Single Responsibility Principle.

```
// BAD: God Object
class ApplicationManager {
    public void HandleUserLogin() { }
    public void ProcessPayment() { }
    public void SendEmail() { }
    public void LogActivity() { }
    public void GenerateReport() { }
    // ... 50 more methods
}

// GOOD: Separated concerns
class UserService { public void Login() { } }
class PaymentService { public void Process() { } }
class EmailService { public void Send() { } }
class LoggingService { public void Log() { } }
class ReportService { public void Generate() { } }
```

Anemic Domain Model

Problem: When Repository pattern goes wrong - entities become just data containers.
Symptoms: Business logic scattered in services, entities with only getters/setters.
Solution: Move business logic back into domain entities.

```
// BAD: Anemic model
class Order {
    public int Id { get; set; }
    public decimal Amount { get; set; }
    public string Status { get; set; }
}

class OrderService {
    public void ConfirmOrder(Order order) {
        if (order.Amount > 0) order.Status = "Confirmed";
    }
}

// GOOD: Rich domain model
class Order {
    public int Id { get; private set; }
    public decimal Amount { get; private set; }
    public OrderStatus Status { get; private set; }

    public void Confirm() {
        if (Amount <= 0) throw new InvalidOperationException("Invalid amount");
        Status = OrderStatus.Confirmed;
    }
}
```

Call Super

Problem: Template Method anti-pattern where subclasses must call parent methods.

Symptoms: Forgot to call base method, inconsistent behavior.

Solution: Use composition over inheritance, or make the pattern more explicit.

```
// BAD: Call Super anti-pattern
abstract class BaseProcessor {
    public void Process() {
        Validate();
        DoWork();
        Cleanup(); // Subclasses must remember to call this
    }
    protected abstract void DoWork();
}

// GOOD: Explicit template method
abstract class BaseProcessor {
    public void Process() {
        Validate();
        DoWork();
        Cleanup();
    }
    protected abstract void DoWork();
    protected virtual void Cleanup() { } // Default implementation
}
```

16. Pattern Combinations & Recipes

MVC + Repository + Unit of Work

Use Case: Classic web application stack

Benefits: Clean separation, testable, maintainable

```
// Model
public class User { public int Id; public string Name; }

// Repository
public interface IUserRepository {
    User GetById(int id);
    void Save(User user);
}

// Unit of Work
public interface IUnitOfWork {
    IUserRepository Users { get; }
    void Commit();
}

// Controller
public class UserController {
    private readonly IUnitOfWork _unitOfWork;

    public UserController(IUnitOfWork unitOfWork) {
        _unitOfWork = unitOfWork;
    }

    public ActionResult GetUser(int id) {
        var user = _unitOfWork.Users.GetById(id);
        return View(user);
    }
}
```

Observer + Command + Memento

Use Case: Undo/Redo system with notifications

Benefits: Flexible, extensible, maintains history

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

// Memento
public class EditorMemento {
    public string Content { get; }
    public EditorMemento(string content) => Content = content;
}

// Observer
public class EditorHistory {
    private readonly List<EditorMemento> _history = new();
    public event Action<EditorMemento> OnStateChanged;

    public void SaveState(EditorMemento memento) {
        _history.Add(memento);
        OnStateChanged?.Invoke(memento);
    }
}
```

Factory + Strategy + Decorator

Use Case: Flexible object creation with runtime behavior

Benefits: Highly configurable, extensible

```
// Strategy
public interface IPricingStrategy {
    decimal CalculatePrice(decimal basePrice);
}

// Decorator
public abstract class PricingDecorator : IPricingStrategy {
    protected IPricingStrategy _strategy;
    public PricingDecorator(IPricingStrategy strategy) => _strategy = strategy;
    public abstract decimal CalculatePrice(decimal basePrice);
}

// Factory
public class PricingFactory {
    public static IPricingStrategy CreateStrategy(string type) {
        return type switch {
            "discount" => new DiscountPricingStrategy(),
            "premium" => new PremiumPricingStrategy(),
            _ => new StandardPricingStrategy()
        };
    }
}
```

17. Performance Impact Analysis

Memory Usage Comparison

Pattern	Memory Overhead	When to Avoid
Singleton	Low	When you need multiple instances
Factory	Low	For simple object creation
Observer	Medium	With many subscribers
Decorator	Medium	Deep decoration chains
Proxy	Low-Medium	For lightweight objects
Flyweight	Very Low	When objects aren't similar

Pattern	Memory Overhead	When to Avoid
Command	Medium	For simple operations
Strategy	Low	When algorithms are simple

Thread Safety Considerations

```
// Thread-safe Singleton
public class ThreadSafeSingleton {
    private static readonly Lazy<ThreadSafeSingleton> _instance =
        new(() => new ThreadSafeSingleton());

    public static ThreadSafeSingleton Instance => _instance.Value;

    private ThreadSafeSingleton() { }
}

// Thread-safe Observer
public class ThreadSafeSubject {
    private readonly List<IObserver> _observers = new();
    private readonly object _lock = new();

    public void Attach(IObserver observer) {
        lock (_lock) {
            _observers.Add(observer);
        }
    }

    public void Notify() {
        List<IObserver> observersCopy;
        lock (_lock) {
            observersCopy = new List<IObserver>(_observers);
        }

        foreach (var observer in observersCopy) {
            observer.Update();
        }
    }
}
```

18. Common Implementation Mistakes

Singleton Pitfalls

Mistake: Not thread-safe, hard to test

```
// BAD
class BadSingleton {
    private static BadSingleton _instance;
    public static BadSingleton Instance {
        get {
            if (_instance == null) _instance = new BadSingleton(); // Race condition!
            return _instance;
        }
    }
}

// GOOD
class GoodSingleton {
    private static readonly Lazy<GoodSingleton> _instance = new();
    public static GoodSingleton Instance => _instance.Value;
    private GoodSingleton() { }
}
```

Observer Memory Leaks

Mistake: Forgetting to unsubscribe

```
// BAD: Memory leak
public class BadSubscriber {
    public BadSubscriber(Subject subject) {
        subject.OnEvent += HandleEvent; // Never unsubscribes!
    }
}

// GOOD: Proper cleanup
public class GoodSubscriber : IDisposable {
    private readonly Subject _subject;

    public GoodSubscriber(Subject subject) {
        _subject = subject;
        _subject.OnEvent += HandleEvent;
    }

    public void Dispose() {
        _subject.OnEvent -= HandleEvent;
    }
}
```

Factory Overuse

Mistake: Creating unnecessary abstractions

```
// BAD: Over-engineered
public interface IStringFactory {
    string CreateString(string value);
}

// GOOD: Simple and direct
public class StringHelper {
    public static string CreateString(string value) => value;
}
```

19. Refactoring Guide

Code Smells That Indicate Pattern Need

Creation Smells

Long Parameter Lists

Problem: Methods with too many parameters are hard to use and maintain.
Symptoms: 5+ parameters, optional parameters, complex constructors.
Solution: Builder pattern for complex object creation.

```
// BAD: Long parameter list
class Person {
    public Person(string firstName, string lastName, int age, string email,
                  string phone, string address, string city, string country,
                  bool isVip, DateTime birthDate, string occupation) {
        // Constructor with 11 parameters!
    }
}

// GOOD: Builder pattern
class PersonBuilder {
    private Person _person = new Person();

    public PersonBuilder SetName(string firstName, string lastName) {
        _person.FirstName = firstName;
        _person.LastName = lastName;
        return this;
    }

    public PersonBuilder SetContact(string email, string phone) {
        _person.Email = email;
        _person.Phone = phone;
        return this;
    }

    public PersonBuilder SetAddress(string address, string city, string country) {
        _person.Address = address;
        _person.City = city;
        _person.Country = country;
        return this;
    }

    public PersonBuilder SetVip(bool isVip) {
        _person.IsVip = isVip;
        return this;
    }

    public Person Build() => _person;
}

// Usage
var person = new PersonBuilder()
    .SetName("John", "Doe")
```

```
    .SetContact("john@email.com", "123-456-7890")
    .SetAddress("123 Main St", "New York", "USA")
    .SetVip(true)
    .Build();
```

Complex Object Construction

Problem: Object creation logic is scattered and complex.
Symptoms: Multiple constructors, complex initialization, conditional creation.
Solution: Factory pattern to centralize creation logic.

```
// BAD: Complex construction scattered
class DocumentProcessor {
    public void ProcessDocument(string filePath) {
        Document doc;
        if (filePath.EndsWith(".pdf")) {
            doc = new PdfDocument(filePath);
            doc.Load();
            doc.Validate();
        } else if (filePath.EndsWith(".docx")) {
            doc = new WordDocument(filePath);
            doc.Load();
            doc.Validate();
        } else if (filePath.EndsWith(".txt")) {
            doc = new TextDocument(filePath);
            doc.Load();
        }
        // Complex creation logic everywhere
    }
}

// GOOD: Factory pattern
interface IDocumentFactory {
    Document CreateDocument(string filePath);
}

class DocumentFactory : IDocumentFactory {
    public Document CreateDocument(string filePath) {
        return Path.GetExtension(filePath).ToLower() switch {
            ".pdf" => new PdfDocument(filePath),
            ".docx" => new WordDocument(filePath),
            ".txt" => new TextDocument(filePath),
            _ => throw new NotSupportedException($"File type not supported: {filePath}")
        };
    }
}

class DocumentProcessor {
    private readonly IDocumentFactory _factory;

    public DocumentProcessor(IDocumentFactory factory) {
        _factory = factory;
    }
```

```
    public void ProcessDocument(string filePath) {
        var doc = _factory.CreateDocument(filePath);
        doc.Load();
        doc.Validate();
    }
}
```

Multiple Similar Constructors

Problem: Many constructors with slight variations (constructor overloading).
Symptoms: 3+ constructors, similar parameter sets, optional parameters.
Solution: Factory Method pattern for different creation scenarios.

```
// BAD: Multiple similar constructors
class DatabaseConnection {
    public DatabaseConnection(string connectionString) { }
    public DatabaseConnection(string server, string database) { }
    public DatabaseConnection(string server, string database, string username) { }
    public DatabaseConnection(string server, string database, string username, string pas
    public DatabaseConnection(string server, int port, string database) { }
    // 5+ constructors for different scenarios
}

// GOOD: Factory Method pattern
abstract class DatabaseConnectionFactory {
    public abstract DatabaseConnection CreateConnection();
}

class SqlServerConnectionFactory : DatabaseConnectionFactory {
    public override DatabaseConnection CreateConnection() {
        return new DatabaseConnection("Server=localhost;Database=MyDB;Trusted_Connection=
    }
}

class MySqlConnectionFactory : DatabaseConnectionFactory {
    public override DatabaseConnection CreateConnection() {
        return new DatabaseConnection("Server=localhost;Database=MyDB;Uid=root;Pwd=passwo
    }
}

class OracleConnectionFactory : DatabaseConnectionFactory {
    public override DatabaseConnection CreateConnection() {
        return new DatabaseConnection("Data Source=localhost:1521/ORCL;User Id=scott;Pass
    }
}
```

Structure Smells

Large Classes with Multiple Responsibilities

Problem: Classes that do too many things (violates Single Responsibility Principle).
Symptoms: 500+ lines, multiple reasons to change, complex dependencies.
Solution: Facade pattern to simplify complex subsystems.

```
// BAD: Large class with multiple responsibilities
class OrderProcessor {
    public void ProcessOrder(Order order) {
        // Validation logic
        ValidateOrder(order);

        // Payment processing
        ProcessPayment(order);

        // Inventory management
        UpdateInventory(order);

        // Shipping calculation
        CalculateShipping(order);

        // Email notifications
        SendConfirmationEmail(order);
        SendNotificationToWarehouse(order);

        // Logging
        LogOrderProcessing(order);

        // Analytics
        UpdateAnalytics(order);

        // Audit trail
        CreateAuditRecord(order);

        // 200+ lines of mixed responsibilities
    }
}

// GOOD: Facade pattern
class OrderProcessingFacade {
    private readonly IOrderValidator _validator;
    private readonly IPaymentProcessor _paymentProcessor;
    private readonly IInventoryManager _inventoryManager;
    private readonly IShippingCalculator _shippingCalculator;
    private readonly INotificationService _notificationService;
    private readonly ILogger _logger;
    private readonly IAnalyticsService _analytics;
    private readonly IAuditService _audit;
```

```
    public OrderProcessingFacade(
        IOrderValidator validator,
        IPaymentProcessor paymentProcessor,
        IInventoryManager inventoryManager,
        IShippingCalculator shippingCalculator,
        INotificationService notificationService,
        ILogger logger,
        IAnalyticsService analytics,
        IAuditService audit) {
        _validator = validator;
        _paymentProcessor = paymentProcessor;
        _inventoryManager = inventoryManager;
        _shippingCalculator = shippingCalculator;
        _notificationService = notificationService;
        _logger = logger;
        _analytics = analytics;
        _audit = audit;
    }

    public void ProcessOrder(Order order) {
        _validator.Validate(order);
        _paymentProcessor.Process(order);
        _inventoryManager.Update(order);
        _shippingCalculator.Calculate(order);
        _notificationService.SendConfirmation(order);
        _logger.LogOrder(order);
        _analytics.Update(order);
        _audit.Record(order);
    }
}
```

Incompatible Interfaces

Problem: Two systems that need to work together but have different interfaces.

Symptoms: Type mismatches, interface conflicts, integration difficulties.

Solution: Adapter pattern to make incompatible interfaces work together.

```
// BAD: Incompatible interfaces
class LegacyPaymentSystem {
    public void ProcessPayment(string accountNumber, decimal amount) {
        Console.WriteLine($"Legacy: Processing ${amount} to account {accountNumber}");
    }
}

class ModernPaymentService {
    public void Charge(PaymentInfo paymentInfo) {
        Console.WriteLine($"Modern: Charging {paymentInfo.Amount} to {paymentInfo.CardNum
    }
}

class PaymentInfo {
    public string CardNumber { get; set; }
    public decimal Amount { get; set; }
}

// Client expects modern interface but needs to use legacy system
class PaymentProcessor {
    public void ProcessPayment(PaymentInfo paymentInfo) {
        // How do we use LegacyPaymentSystem with PaymentInfo?
        // We need to convert between incompatible interfaces
    }
}

// GOOD: Adapter pattern
interface IPaymentProcessor {
    void ProcessPayment(PaymentInfo paymentInfo);
}

class LegacyPaymentAdapter : IPaymentProcessor {
    private readonly LegacyPaymentSystem _legacySystem;

    public LegacyPaymentAdapter(LegacyPaymentSystem legacySystem) {
        _legacySystem = legacySystem;
    }

    public void ProcessPayment(PaymentInfo paymentInfo) {
        // Convert modern interface to legacy interface
        string accountNumber = ExtractAccountNumber(paymentInfo.CardNumber);
        _legacySystem.ProcessPayment(accountNumber, paymentInfo.Amount);
    }
```

```
    private string ExtractAccountNumber(string cardNumber) {
        // Extract account number from card number
        return cardNumber.Substring(cardNumber.Length - 4);
    }
}

class ModernPaymentAdapter : IPaymentProcessor {
    private readonly ModernPaymentService _modernService;

    public ModernPaymentAdapter(ModernPaymentService modernService) {
        _modernService = modernService;
    }

    public void ProcessPayment(PaymentInfo paymentInfo) {
        _modernService.Charge(paymentInfo);
    }
}
```

Tightly Coupled Classes

Problem: Classes that depend too much on each other's implementation details.
Symptoms: Hard to test, hard to change, circular dependencies.
Solution: Bridge pattern to decouple abstraction from implementation.

```
// BAD: Tightly coupled classes
class WindowsButton {
    public void Render() {
        Console.WriteLine("Windows Button rendered");
    }
}

class MacButton {
    public void Render() {
        Console.WriteLine("Mac Button rendered");
    }
}

class WindowsTextBox {
    public void Render() {
        Console.WriteLine("Windows TextBox rendered");
    }
}

class MacTextBox {
    public void Render() {
        Console.WriteLine("Mac TextBox rendered");
    }
}

// If we add Linux support, we need LinuxButton, LinuxTextBox, etc.
// If we add new controls, we need WindowsControl, MacControl, LinuxControl
// This leads to explosion of classes!

// GOOD: Bridge pattern
interface IRenderer {
    void RenderButton(string text);
    void RenderTextBox(string placeholder);
}

class WindowsRenderer : IRenderer {
    public void RenderButton(string text) => Console.WriteLine($"Windows Button: {text}")
    public void RenderTextBox(string placeholder) => Console.WriteLine($"Windows TextBox:
}

class MacRenderer : IRenderer {
    public void RenderButton(string text) => Console.WriteLine($"Mac Button: {text}");
    public void RenderTextBox(string placeholder) => Console.WriteLine($"Mac TextBox: {pl
```

```
}

abstract class UIControl {
    protected IRenderer renderer;

    public UIControl(IRenderer renderer) {
        this.renderer = renderer;
    }

    public abstract void Render();
}

class Button : UIControl {
    private string text;

    public Button(IRenderer renderer, string text) : base(renderer) {
        this.text = text;
    }

    public override void Render() {
        renderer.RenderButton(text);
    }
}

class TextBox : UIControl {
    private string placeholder;

    public TextBox(IRenderer renderer, string placeholder) : base(renderer) {
        this.placeholder = placeholder;
    }

    public override void Render() {
        renderer.RenderTextBox(placeholder);
    }
}
```

Behavior Smells

Long If-Else Chains

Problem: Complex conditional logic that's hard to maintain and extend.
Symptoms: 5+ if-else statements, switch statements with many cases, complex conditions.
Solution: Strategy pattern to encapsulate different behaviors.

```
// BAD: Long if-else chain
class DiscountCalculator {
    public decimal CalculateDiscount(Customer customer, decimal amount) {
        if (customer.Type == "VIP" && customer.MembershipYears >= 5) {
            return amount * 0.20m; // 20% discount
        } else if (customer.Type == "VIP" && customer.MembershipYears >= 2) {
            return amount * 0.15m; // 15% discount
        } else if (customer.Type == "VIP") {
            return amount * 0.10m; // 10% discount
        } else if (customer.Type == "Premium" && customer.MembershipYears >= 3) {
            return amount * 0.12m; // 12% discount
        } else if (customer.Type == "Premium") {
            return amount * 0.08m; // 8% discount
        } else if (customer.Type == "Regular" && customer.MembershipYears >= 5) {
            return amount * 0.05m; // 5% discount
        } else if (customer.Type == "Regular" && customer.MembershipYears >= 2) {
            return amount * 0.03m; // 3% discount
        } else if (customer.Type == "Regular") {
            return amount * 0.01m; // 1% discount
        } else {
            return 0; // No discount
        }
    }
}

// GOOD: Strategy pattern
interface IDiscountStrategy {
    decimal CalculateDiscount(Customer customer, decimal amount);
}

class VipDiscountStrategy : IDiscountStrategy {
    public decimal CalculateDiscount(Customer customer, decimal amount) {
        return customer.MembershipYears switch {
            >= 5 => amount * 0.20m,
            >= 2 => amount * 0.15m,
            _ => amount * 0.10m
        };
    }
}

class PremiumDiscountStrategy : IDiscountStrategy {
    public decimal CalculateDiscount(Customer customer, decimal amount) {
        return customer.MembershipYears >= 3 ? amount * 0.12m : amount * 0.08m;
```

```
    }
}

class RegularDiscountStrategy : IDiscountStrategy {
    public decimal CalculateDiscount(Customer customer, decimal amount) {
        return customer.MembershipYears switch {
            >= 5 => amount * 0.05m,
            >= 2 => amount * 0.03m,
            _ => amount * 0.01m
        };
    }
}

class DiscountCalculator {
    private readonly Dictionary<string, IDiscountStrategy> _strategies;

    public DiscountCalculator() {
        _strategies = new Dictionary<string, IDiscountStrategy> {
            ["VIP"] = new VipDiscountStrategy(),
            ["Premium"] = new PremiumDiscountStrategy(),
            ["Regular"] = new RegularDiscountStrategy()
        };
    }

    public decimal CalculateDiscount(Customer customer, decimal amount) {
        if (_strategies.TryGetValue(customer.Type, out var strategy)) {
            return strategy.CalculateDiscount(customer, amount);
        }
        return 0;
    }
}
```

Duplicate Code in Subclasses

Problem: Similar code repeated across multiple subclasses.

Symptoms: Copy-paste programming, similar methods in subclasses, hard to maintain.

Solution: Template Method pattern to define common algorithm structure.

```
// BAD: Duplicate code in subclasses
class PdfReportGenerator {
    public void GenerateReport(ReportData data) {
        // Common setup
        InitializeReport();
        LoadData(data);

        // PDF-specific implementation
        CreatePdfDocument();
        AddPdfContent(data);
        FormatPdfLayout();

        // Common cleanup
        ValidateReport();
        SaveReport();
    }
}

class ExcelReportGenerator {
    public void GenerateReport(ReportData data) {
        // Common setup (duplicated!)
        InitializeReport();
        LoadData(data);

        // Excel-specific implementation
        CreateExcelWorkbook();
        AddExcelContent(data);
        FormatExcelLayout();

        // Common cleanup (duplicated!)
        ValidateReport();
        SaveReport();
    }
}

class WordReportGenerator {
    public void GenerateReport(ReportData data) {
        // Common setup (duplicated again!)
        InitializeReport();
        LoadData(data);

        // Word-specific implementation
        CreateWordDocument();
```

```
        AddWordContent(data);
        FormatWordLayout();

        // Common cleanup (duplicated again!)
        ValidateReport();
        SaveReport();
    }
}

// GOOD: Template Method pattern
abstract class ReportGenerator {
    // Template method - defines the algorithm structure
    public void GenerateReport(ReportData data) {
        InitializeReport();
        LoadData(data);
        CreateDocument();
        AddContent(data);
        FormatLayout();
        ValidateReport();
        SaveReport();
    }

    // Common implementations
    protected void InitializeReport() => Console.WriteLine("Initializing report...");
    protected void LoadData(ReportData data) => Console.WriteLine("Loading data...");
    protected void ValidateReport() => Console.WriteLine("Validating report...");
    protected void SaveReport() => Console.WriteLine("Saving report...");

    // Abstract methods - must be implemented by subclasses
    protected abstract void CreateDocument();
    protected abstract void AddContent(ReportData data);
    protected abstract void FormatLayout();
}

class PdfReportGenerator : ReportGenerator {
    protected override void CreateDocument() => Console.WriteLine("Creating PDF document"
    protected override void AddContent(ReportData data) => Console.WriteLine("Adding PDF
    protected override void FormatLayout() => Console.WriteLine("Formatting PDF layout");
}

class ExcelReportGenerator : ReportGenerator {
    protected override void CreateDocument() => Console.WriteLine("Creating Excel workboo
    protected override void AddContent(ReportData data) => Console.WriteLine("Adding Exce
```

```
    protected override void FormatLayout() => Console.WriteLine("Formatting Excel layout"
}

class WordReportGenerator : ReportGenerator {
    protected override void CreateDocument() => Console.WriteLine("Creating Word document
    protected override void AddContent(ReportData data) => Console.WriteLine("Adding Word
    protected override void FormatLayout() => Console.WriteLine("Formatting Word layout")
}
```

Hard-Coded Algorithms

Problem: Algorithms embedded directly in code, making them hard to change.

Symptoms: Algorithm logic in business methods, hard to test algorithms, inflexible code.

Solution: Strategy pattern to make algorithms interchangeable.

```
// BAD: Hard-coded algorithms
class DataProcessor {
    public List<int> ProcessData(List<int> data, string algorithm) {
        if (algorithm == "bubble") {
            // Bubble sort implementation
            for (int i = 0; i < data.Count - 1; i++) {
                for (int j = 0; j < data.Count - i - 1; j++) {
                    if (data[j] > data[j + 1]) {
                        int temp = data[j];
                        data[j] = data[j + 1];
                        data[j + 1] = temp;
                    }
                }
            }
        } else if (algorithm == "quick") {
            // Quick sort implementation
            QuickSort(data, 0, data.Count - 1);
        } else if (algorithm == "merge") {
            // Merge sort implementation
            data = MergeSort(data);
        }
        return data;
    }

    private void QuickSort(List<int> arr, int low, int high) {
        // Quick sort implementation
    }

    private List<int> MergeSort(List<int> arr) {
        // Merge sort implementation
    }
}

// GOOD: Strategy pattern
interface ISortStrategy {
    List<int> Sort(List<int> data);
}

class BubbleSortStrategy : ISortStrategy {
    public List<int> Sort(List<int> data) {
        var result = new List<int>(data);
        for (int i = 0; i < result.Count - 1; i++) {
            for (int j = 0; j < result.Count - i - 1; j++) {
```

```
                if (result[j] > result[j + 1]) {
                    int temp = result[j];
                    result[j] = result[j + 1];
                    result[j + 1] = temp;
                }
            }
        }
        return result;
    }
}

class QuickSortStrategy : ISortStrategy {
    public List<int> Sort(List<int> data) {
        var result = new List<int>(data);
        QuickSort(result, 0, result.Count - 1);
        return result;
    }

    private void QuickSort(List<int> arr, int low, int high) {
        if (low < high) {
            int pi = Partition(arr, low, high);
            QuickSort(arr, low, pi - 1);
            QuickSort(arr, pi + 1, high);
        }
    }

    private int Partition(List<int> arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        int temp2 = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp2;
```

```
        return i + 1;
    }
}

class MergeSortStrategy : ISortStrategy {
    public List<int> Sort(List<int> data) {
        if (data.Count <= 1) return new List<int>(data);

        int mid = data.Count / 2;
        var left = data.Take(mid).ToList();
        var right = data.Skip(mid).ToList();

        left = Sort(left);
        right = Sort(right);

        return Merge(left, right);
    }

    private List<int> Merge(List<int> left, List<int> right) {
        var result = new List<int>();
        int i = 0, j = 0;

        while (i < left.Count && j < right.Count) {
            if (left[i] <= right[j]) {
                result.Add(left[i]);
                i++;
            } else {
                result.Add(right[j]);
                j++;
            }
        }

        while (i < left.Count) {
            result.Add(left[i]);
            i++;
        }

        while (j < right.Count) {
            result.Add(right[j]);
            j++;
        }

        return result;
```

```
    }
}

class DataProcessor {
    private ISortStrategy _sortStrategy;

    public DataProcessor(ISortStrategy sortStrategy) {
        _sortStrategy = sortStrategy;
    }

    public void SetSortStrategy(ISortStrategy sortStrategy) {
        _sortStrategy = sortStrategy;
    }

    public List<int> ProcessData(List<int> data) {
        return _sortStrategy.Sort(data);
    }
}
```

Step-by-Step Refactoring Example

Before: Monolithic class with multiple responsibilities

```
class OrderProcessor {
    public void ProcessOrder(Order order) {
        // Validation
        if (order.Amount <= 0) throw new Exception("Invalid amount");

        // Payment processing
        if (order.PaymentMethod == "CreditCard") {
            // Credit card logic
        } else if (order.PaymentMethod == "PayPal") {
            // PayPal logic
        }

        // Inventory management
        foreach (var item in order.Items) {
            // Reduce inventory
        }

        // Email notification
        // Send confirmation email
    }
}
```

After: Separated using multiple patterns

```
// Strategy for payment processing
public interface IPaymentProcessor {
    void ProcessPayment(decimal amount);
}

// Factory for creating processors
public class PaymentProcessorFactory {
    public static IPaymentProcessor Create(string method) {
        return method switch {
            "CreditCard" => new CreditCardProcessor(),
            "PayPal" => new PayPalProcessor(),
            _ => throw new ArgumentException("Unknown payment method")
        };
    }
}

// Command for order processing
public class ProcessOrderCommand {
    private readonly Order _order;
    private readonly IPaymentProcessor _paymentProcessor;

    public ProcessOrderCommand(Order order, IPaymentProcessor processor) {
        _order = order;
        _paymentProcessor = processor;
    }

    public void Execute() {
        ValidateOrder();
        _paymentProcessor.ProcessPayment(_order.Amount);
        UpdateInventory();
        SendNotification();
    }
}
```

20. Language-Specific Considerations

C# Specific Implementations

Async/Await Patterns

```
// Async Factory
public class AsyncFactory {
    public static async Task<T> CreateAsync<T>(Func<Task<T>> factory) {
        return await factory();
    }
}

// Async Observer
public class AsyncSubject {
    private readonly List<Func<Task>> _observers = new();

    public void Attach(Func<Task> observer) => _observers.Add(observer);

    public async Task NotifyAsync() {
        var tasks = _observers.Select(observer => observer());
        await Task.WhenAll(tasks);
    }
}
```

LINQ Functional Patterns

```
// Functional composition with LINQ
public static class FunctionalExtensions {
    public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
        Func<T, bool> predicate) => source.Where(predicate);

    public static IEnumerable<U> Select<T, U>(this IEnumerable<T> source,
        Func<T, U> selector) => source.Select(selector);

    public static T Aggregate<T>(this IEnumerable<T> source,
        Func<T, T, T> func) => source.Aggregate(func);
}
```

Dependency Injection Integration

```
// Service registration
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<ILogger, ConsoleLogger>();
    services.AddScoped<IUserRepository, UserRepository>();
    services.AddTransient<IUserService, UserService>();
}

// Pattern with DI
public class UserController {
    private readonly IUserService _userService;
    private readonly ILogger _logger;

    public UserController(IUserService userService, ILogger logger) {
        _userService = userService;
        _logger = logger;
    }
}
```

Conclusion

This comprehensive guide provides a systematic approach to choosing design patterns. Remember:

- **Start simple** - Don't use patterns unless you have a real problem
- **Consider context** - The same problem might need different solutions in different contexts
- **Think ahead** - Choose patterns that will help with future changes
- **Keep learning** - Patterns are tools, not rules
- **Avoid anti-patterns** - Learn from common mistakes
- **Combine wisely** - Use pattern combinations for complex scenarios
- **Consider performance** - Understand the trade-offs
- **Refactor gradually** - Don't try to fix everything at once

The biggest mistake I see developers make is trying to use every pattern they know. Pick the simplest solution that solves your problem, and refactor later if needed. Most of the time, a simple Factory Method or Strategy pattern will do the job just fine.

Good luck with your pattern matching!