In Control

Object-Oriented Programming in C#

Andrea Pierini

Object-Oriented Programming in C#

In Control

Andrea Pierini

This book is available at
https://leanpub.com/Object-Oriented_Programming_CSharp

This version was published on 2025-04-25

To Sara, the love of my life, and to my family.

Contents

Object-Oriented Programming in C#

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Book introduction

This book provides a thorough introduction to Object-Oriented Programming (OOP) concepts using C#.

OOP is crucial in modern software development because it provides a powerful way to organize and structure code. By using objects that bundle data together with the code that operates on that data, OOP allows developers to create more modular, flexible, and reusable code. It enables easier maintenance, promotes code reuse through inheritance, and helps manage complexity in large software systems by breaking down complex problems into smaller, more manageable pieces. Key concepts like encapsulation, inheritance, and polymorphism make code more intuitive, allowing developers to model real-world entities and relationships more naturally in their software designs.

'In Control' Philosopy

This book is part of the **'In Control' series**, a collection founded on a key principle for modern software development: In an era where artificial intelligence significantly assists in writing code, the emphasis shifts from mastering every syntactical detail to deeply understanding fundamental concepts and learning how to effectively collaborate with AI. The name 'In Control' reflects this philosophy – empowering you to guide the tools and the development process.

Therefore, across this series, we concentrate on mastering core programming concepts. This specific volume provides essential prompts designed to help you practice and apply these ideas. A key focus is learning how to leverage AI assistants effectively when needed. Specifically, you will find prompts formatted like this throughout the book:

Ask your Chat BOT: [A suggested question related to the current topic]

These are designed to encourage you to dive deeper, explore alternatives, or clarify concepts using your preferred AI assistant. Engaging with these prompts actively reinforces the material and helps you practice the crucial skill of asking the right questions to leverage AI effectively in your learning.

'Core' vs 'In-depth'

The first part of this text, titled 'Core OOP Concepts,' will serve as a quick reference, providing a fast and easy way to grasp the main concepts–ideal for a general overview and quick consultations. The second part, titled 'OOP In-Depth' will explore each concept in much greater detail, offering thorough explanations and additional perspectives.

Target Audience

This guide is not intended for complete programming beginners but rather for developers who want to deepen their understanding of Object-Oriented Programming (OOP) and fully leverage its potential once mastered. A solid grasp of C# is recommended. Moreover, the concepts presented here are not only valuable in C# but also applicable to any OOP language.

Introduction to Object-Oriented Programming

Object-Oriented Programming represents a paradigm shift in how we approach software development. Unlike procedural programming, which focuses on functions and procedures, OOP centers around objects and their interactions. This approach allows developers to create more maintainable, reusable, and scalable code.

The core idea behind OOP is modeling software components after real-world entities. For instance, if you're building a car dealership application, you might create classes for Car, Customer, and Sale, each with its own properties and behaviors. This mapping between real-world concepts and programming constructs makes code more intuitive and easier to reason about.

C# stands as an ideal language for learning OOP principles. As a modern, strongly-typed language developed by Microsoft, C# provides robust support for all key OOP concepts while offering a clean syntax and powerful development environment through the .NET framework.

Ask your Chat BOT: Can you provide 3 more examples of real-world scenarios (like the car dealership) and suggest the primary classes, properties, and behaviors that would model them in OOP?

What Makes OOP Different?

Object-Oriented Programming differs from other paradigms through its emphasis on:

1. Modeling real-world objects and their relationships
2. Encapsulating related data and functionality
3. Creating reusable components through inheritance and interfaces
4. Promoting code organization through clear separation of concerns
5. Enabling extensibility through polymorphic behavior

Before diving deeper, it's essential to understand that OOP isn't just about syntax or language features—it's a way of thinking about software design that emphasizes structure, relationships, and behavior. Let's begin exploring these core ideas, starting with the fundamental building blocks.

Core OOP Concepts

Classes and Objects

In OOP, a class serves as a blueprint that defines the structure and behavior for a type of object. An object, by contrast, is a concrete instance of a class. Think of a class as a template, while objects are the actual entities created from that template.

For example, consider a simple `Car` class:

```csharp
class Car
{
    // Properties (data)
    public string Brand { get; set; }
    public string Color { get; set; }
    public int Year { get; set; }

    // Methods (behavior)
    public void StartEngine()
    {
        Console.WriteLine("Engine started!");
    }

    public void Drive()
    {
        Console.WriteLine(
            $"Driving the {Color} {Brand}");
    }
}
```

Creating objects (instances) from this class:

```csharp
// Creating two different Car objects
Car myCar = new Car();
myCar.Brand = "Mustang";
myCar.Color = "Black";
myCar.Year = 2023;

Car friendsCar = new Car();
friendsCar.Brand = "Ferrari";
friendsCar.Color = "Red";
friendsCar.Year = 2022;

// Using the objects
myCar.StartEngine();
myCar.Drive();
```

This example demonstrates how multiple objects can be created from the same class blueprint, each with its own state (property values) but sharing the same behavior (methods).

Encapsulation

Encapsulation represents the practice of bundling data (properties) and the methods that operate on that data within a single unit (class). More importantly, it involves controlling access to that data through access modifiers.

The primary benefit of encapsulation is information hiding–protecting the internal state of an object from inappropriate external access. This creates a clear boundary between the internal implementation and how the class is interacted with externally.

```csharp
class BankAccount
{
    // Private field:
    // hidden from outside the class
    private decimal balance;

    // Public property:
    // controlled access to the private field
    public decimal Balance
    {
        get { return balance; }

        // Can only be set within the class
        private set { balance = value; }
    }

    // Public methods to interact with the balance
    public void Deposit(decimal amount)
    {
        if (amount <= 0)
            throw new ArgumentException(
                "Deposit amount must be positive");

        balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= 0)
            throw new ArgumentException(
                "Withdrawal amount must be positive");

        if (amount > balance)
            throw new InvalidOperationException(
                "Insufficient funds");

        balance -= amount;
    }
}
```

In this example, the `balance` field is completely hidden from external code. Users of the `BankAccount` class must use the provided methods to deposit or withdraw money, ensuring that the account balance can never reach an invalid state.

Ask your Chat BOT: Why is information hiding, achieved through encapsulation using private fields and controlled access via properties/methods (like in the BankAccount example), considered a fundamental principle for building robust and maintainable software? What problems does it help prevent?

Abstraction

Abstraction involves simplifying complex reality by modeling classes based on the essential properties and behaviors while ignoring non-essential details. It focuses on what an object does rather than how it does it.

In practice, abstraction means exposing only the necessary features of an object while hiding the complexity of how those features are implemented.

Abstraction by mean of Interfaces

While classes define a blueprint including both data (state) and behavior (methods), sometimes we only want to define a **contract** – a specific set of capabilities or behaviors that a class must provide, without dictating how it should implement them or what data it should hold. This is the role of an **interface** in C#.

An interface defines a collection of public method signatures, properties, events, or indexers. Any class or struct that implements an interface guarantees that it will provide concrete implementations for all members defined in that interface.

Interfaces are fundamental for:

1. **Achieving Polymorphism:** Allowing different classes to be treated uniformly based on the capabilities they offer.
2. **Enabling Loose Coupling:** Designing components that depend on contracts (interfaces) rather than specific concrete classes, making systems more flexible and testable (crucial for Dependency Injection, as we well see later in this book).

3. **Simulating Multiple Inheritance (of type):** A class can inherit from only one base class, but it can implement multiple interfaces.

   Syntax:

```csharp
// Defining an interface
// (conventionally starts with 'I')
public interface ILogger
{
    // Method signature - no implementation
    void Log(string message);
}

// A class implementing the interface
// Use ':' for implementation
public class ConsoleLogger : ILogger
{
    // Must provide implementation
    // for all interface members
    public void Log(string message)
    {
        Console.WriteLine($"LOG: {message}");
    }
}

// Another implementation
public class FileLogger : ILogger
{
    public void Log(string message)
    {
        // Logic to write the message to a file...
        Console.WriteLine(
            $"Writing to file: {message}");
    }
}
```

   Using an interface:

```csharp
// We can treat both ConsoleLogger and
// FileLogger as ILogger
ILogger logger1 = new ConsoleLogger();
ILogger logger2 = new FileLogger();

// Calls ConsoleLogger's implementation
logger1.Log("System startup.");

// Calls FileLogger's implementation
logger2.Log("Data processed.");
```

Think of an interface as specifying the “What” (what methods must be available), leaving the “How” (the actual implementation) to the classes that implement it. We will explore interfaces in much greater detail in the ‘OOP In-Depth’ section.

Ask your Chat BOT: What is the key difference between inheriting from a base class (like Animal in the Inheritance example) and implementing an interface (like ILogger here)? Why can a C# class implement multiple interfaces but only inherit from one base class?

Abstraction by mean of Abstract Classes

```csharp
// Abstract class
public abstract class Vehicle
{
    public string RegistrationNumber { get; set; }

    // Abstract method:
    // must be implemented by derived classes
    public abstract void Move();

    // Concrete method - shared implementation
    public void RegisterVehicle(string regNumber)
    {
        RegistrationNumber = regNumber;
        Console.WriteLine(
            $"Vehicle registered with number: {regNumber}");
    }
}

// Concrete implementations
public class Car : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Car is driving on the road");
    }
}

public class Airplane : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Airplane is flying in the sky");
    }
}
```

This example shows how abstraction allows us to define common characteristics (registration) and behaviors (moving) for vehicles, while letting specific vehicle types implement their movement behavior differently.

In the context of AI-assisted development, abstractions like interfaces and abstract classes become powerful tools for maintaining control of your software. By defining clear contracts and partial blueprints, you establish the non-negotiable structure and required behaviors – the **“what”** a component must do or be. You can then leverage AI to generate the specific concrete implementations – the **“how”** – ensuring adherence to the framework you’ve mandated. This perfectly aligns with the **“In Control”** philosophy: you define the essential architecture through abstraction, guiding AI to fill in the details according to your specifications.

Inheritance

Inheritance enables a class to inherit properties and methods from another class. This creates a parent-child relationship between classes (also called a “base class” and a “derived class” relationship).

Inheritance promotes code reuse by allowing common functionality to be defined in a base class and specialized behavior to be implemented in derived classes.

```csharp
// Base class
public class Animal
{
    public string Name
    {
        get; set;
    }

    public void Eat()
    {
        Console.WriteLine($"{Name} is eating...");
    }

    public void Sleep()
    {
        Console.WriteLine($"{Name} is sleeping...");
    }
}

// Derived class
public class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine($"{Name} says: Woof!");
    }
}

// Derived class
public class Cat : Animal
{
    public void Meow()
    {
        Console.WriteLine($"{Name} says: Meow!");
    }
}
```

Using inheritance:

```
Dog dog = new Dog();
dog.Name = "Rex";
dog.Eat();    // Inherited from Animal
dog.Bark();   // Specific to Dog

Cat cat = new Cat();
cat.Name = "Whiskers";
cat.Sleep(); // Inherited from Animal
cat.Meow();   // Specific to Cat
```

We get as output:

```
Rex is eating...
Rex says: Woof!
Whiskers is sleeping...
Whiskers says: Meow!
```

This demonstrates how derived classes inherit general behaviors while adding their specialized behaviors.

Ask your Chat BOT: Explain the “is-a” relationship concept in inheritance using the Animal, Dog, and Cat example. Can you also mention one potential drawback or complexity that might arise from using inheritance extensively (e.g., deep inheritance hierarchies)?

Inheritance creates these ‘is-a’ relationships, and polymorphism leverages them, allowing objects of different derived classes to be treated uniformly through their common base class or interface.

Polymorphism

Polymorphism, which literally means “many forms,” is the principle that allows objects of different classes to respond to the same method call in their own specific way, allowing for flexible and extensible code

There are two main types of polymorphism in C#:

1. **Compile-time polymorphism** (Method overloading): Multiple methods with the same name but different parameters.
2. **Runtime polymorphism** (Method overriding): Base class methods being overridden in derived classes.

```csharp
public class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing a shape");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}
```

Demonstrating polymorphism:

```csharp
// Array of different shapes
Shape[] shapes = new Shape[3];
shapes[0] = new Shape();      // Base class
shapes[1] = new Circle();     // Derived class
shapes[2] = new Rectangle();  // Derived class

// Polymorphic behavior
foreach (Shape shape in shapes)
{
    // Each shape draws differently
    shape.Draw();
}
```

We get as output:

```
Drawing a shape
Drawing a circle
Drawing a rectangle
```

Ask your Chat BOT: Using the Shape example, explain why polymorphism is useful here. What benefit does treating Circle and Rectangle objects simply as Shape objects provide in the foreach loop? Could you achieve the same result without polymorphism, and if so, how would the code differ?

Visualizing OOP: An Introduction to UML Class Diagrams

While writing C# code allows us to define the precise implementation of our object-oriented designs, sometimes we need a higher-level view. We need a way to visualize the structure of our classes, their attributes, their methods, and especially the relationships between them, without getting lost in the specific lines of code. This is where the **Unified Modeling Language (UML)** comes in, specifically **UML Class Diagrams**.

UML is a standardized graphical language used in software engineering to visualize, specify, construct, and document the artifacts of a software system. For OOP developers, class diagrams are arguably the most important part of UML. They provide a static, structural view of a system, perfectly complementing the OOP concepts we've discussed.

In line with our 'In Control' philosophy, you don't need to become a UML guru mastering every intricate detail. The goal is to understand the core elements of class diagrams so you can read them, sketch your own simple diagrams to clarify designs, and effectively communicate structures to others (including AI assistants, which are often capable of generating or interpreting UML).

Ask your Chat BOT: "What are the main benefits of using UML class diagrams in the software design process, especially for object-oriented systems?"

Why UML Class Diagrams for OOP?

- **Visualization:** They turn abstract concepts like classes, inheritance, and composition into clear visual diagrams.
- **Communication:** Provide a standard, language-agnostic way to discuss system design with team members, stakeholders, **or AI**.

- **Design Aid:** Sketching diagrams helps in thinking through relationships and responsibilities before coding, potentially catching design flaws early.
- **Documentation:** Serve as valuable documentation for understanding the structure of existing code.

Core Elements of a Class Diagram

A class diagram primarily consists of classes and the relationships between them.

Each class is represented by a rectangle, usually divided into three compartments:

1. **Top:** Class Name (often bold)
2. **Middle:** Attributes (fields, properties)
3. **Bottom:** Operations (methods)

```
classDiagram
    class BankAccount {
        -decimal balance
        +decimal Balance
        +Deposit(decimal amount)
        +Withdraw(decimal amount)
    }
```

UML uses symbols to indicate the visibility (access modifiers) of attributes and operations:

- + : public (Accessible from anywhere)
- - : private (Accessible only within the class)
- # : protected (Accessible within the class and derived classes)
- ~ : internal (Package/assembly visibility - sometimes omitted or noted separately)

Look again at the BankAccount diagram above: balance is private (-), while Balance, BankAccount, Deposit, and Withdraw are public (+).

Lines connecting class boxes represent relationships. The most important ones for OOP are:

1. **Association:** A general relationship indicating that two classes know about each other. Represented by a solid line. You can specify multiplicity (how many instances are related, e.g., 1, * for many, 0..1 for zero or one).

```
classDiagram
    class Order {
        +int OrderId
    }
    class Customer {
        +string Name
    }
    Order "1" -- "*" Customer : places
    // An Order is placed by 1 Customer,
    // a Customer can place many (*) Orders
```

2. **Aggregation:** A special type of association representing a "has-a" relationship, where one class contains or is composed of other classes, but the contained classes can exist independently. Represented by a solid line with an open diamond on the owner's side.

```
classDiagram
    class Department {
        +string Name
    }
    class Employee {
        +string EmployeeId
    }
    Department o-- "*" Employee : contains
    // Department 'has' Employees,
    // but Employees can exist without a Department
```

3. **Composition:** A stronger form of aggregation ("owns-a"). The contained class's lifetime is tied to the container class. If the container is destroyed, the contained parts usually are too. Represented by a solid line with a filled diamond on the owner's side.

```
classDiagram
    class Building {
    }
    class Room {
    }
    Building *-- "1..*" Room : consists of
    // Building 'owns' Rooms; destroy Building, destroy its Rooms
```

4. **Inheritance (Generalization):** Represents an "is-a" relationship between a base class (superclass) and a derived class (subclass). Represented by a solid line with a closed, unfilled arrowhead pointing from the derived class to the base class.

```
classDiagram
    class Animal {
        +string Name
        +Eat()
        +Sleep()
    }
    class Dog {
        +Bark()
    }
    class Cat {
        +Meow()
    }
    Animal <|-- Dog
    Animal <|-- Cat
```

5. **Interface Implementation (Realization):** Shows that a class implements the operations specified by an interface. Represented by a dashed line with a closed, unfilled arrowhead pointing from the implementing class to the interface. Interfaces themselves are often marked with <>.

```
classDiagram
    class IShape <<interface>> {
        +CalculateArea()
        +CalculatePerimeter()
    }
    class Circle {
        +double Radius
        +CalculateArea()
        +CalculatePerimeter()
    }
    class Rectangle {
        +double Width
        +double Height
        +CalculateArea()
        +CalculatePerimeter()
    }
    IShape <|.. Circle
    IShape <|.. Rectangle
```

- **Abstract Classes/Methods:** Often shown with their names in italics. (Mermaid syntax: class Animal { abstract Eat() } or mark class <>)
- **Static Attributes/Methods:** Often shown with their names underlined. (Mermaid syntax: class MathHelper { {static} PI : double })

Mapping Our C# Examples

Let's visualize some classes we've already discussed:

- **Simple Car Class:**

```
classDiagram
    class Car {
        +string Brand
        +string Color
        +int Year
        +StartEngine()
        +Drive()
    }
```

- **BankAccount with Encapsulation:**

```
classDiagram
    class BankAccount {
        -decimal balance
        +decimal Balance { get; private set; }
        +Deposit(decimal amount)
        +Withdraw(decimal amount)
    }
```

  (Note: Representing property accessors like private set precisely can vary in UML tools, but showing the intent via + for the property and - for the field is common. Mermaid doesn't have a standard way to show getter/setter visibility separately).
- **Animal Inheritance Hierarchy:** (Diagram shown in the Inheritance relationship example above)
- **IShape Interface Implementation:** (Diagram shown in the Interface Implementation relationship example above)

Using UML in Your Workflow

You don't need complex tools to start. Sketching simple class diagrams on paper or a whiteboard during design discussions can be incredibly effective. For documentation or sharing, tools that support UML (including some Markdown flavors with Mermaid support, like shown here, or dedicated software) are useful.

Consider using UML when:

- Designing a new system or feature involving multiple interacting classes.
- Trying to understand a complex part of an existing codebase.
- Communicating a design structure to your team or an AI assistant.

Ask your Chat BOT: “Can you generate a UML class diagram (using Mermaid syntax) for the ReportGenerator and ReportSaver classes shown in the Single Responsibility Principle example?”

Conclusions

UML Class Diagrams are a valuable visual companion to object-oriented programming in C#. They provide a standardized way to represent the static structure of your system, focusing on classes, their members, and their relationships. Understanding the basics allows you to better design, communicate, and document your OOP applications, bridging the gap between abstract concepts and concrete code structure.

OOP In-Depth

Encapsulation In Depth: Protecting State and Controlling Access

The Class: Blueprint for Objects

Abstraction In Depth: Simplifying Complexity and Defining Contracts

Interfaces: Defining Pure Contracts

Abstract Classes: Providing Partial Blueprints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Inheritance In Depth: Establishing Hierarchies and Reusing Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Polymorphism In Depth: One Interface, Many Behaviors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Benefits of Runtime Polymorphism:

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Understanding Composition: Building with Blocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

The Principle: Favor Composition Over Inheritance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

SOLID Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

1. Single Responsibility Principle (SRP)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

2. Open/Closed Principle (OCP)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

3. Liskov Substitution Principle (LSP)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

4. Interface Segregation Principle (ISP)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

5. Dependency Inversion Principle (DIP)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Design Patterns, SOLID Principles, and Dependency Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Foundation: Dependency Inversion, Inversion of Control (IoC), and Dependency Injection (DI)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Simple Factory Pattern (or Static Factory)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Simple Factory and SOLID Principles:

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Benefits & Drawbacks:

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Factory Method Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Singleton Design Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Singleton and SOLID Principles:

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Benefits & Drawbacks:

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Alternative: Dependency Injection with Singleton Lifetime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Conclusions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

OOP Principles in Practice: Structuring .NET Applications

Having grasped the fundamentals of Object-Oriented Programming (OOP), the SOLID principles, and foundational design patterns, it's time to see how these concepts come alive in the real world of .NET development. Theory provides the blueprint, but frameworks and architectural patterns provide the structure where these blueprints are actively used to build functional, maintainable software.

This chapter explores how core OOP principles directly influence the design and usage of major .NET frameworks and common architectural patterns like **Model-View-Controller (MVC)** and **Model-View-ViewModel (MVVM)**. We'll examine:

1. How architectural patterns like MVC and MVVM leverage OOP for **separation of concerns**.
2. How ASP.NET Core deeply integrates **Dependency Injection (DI)**, a direct application of the Dependency Inversion Principle (DIP).

By understanding these practical applications, you'll not only reinforce your grasp of OOP theory but also become more proficient in using these essential .NET tools. You'll see why certain framework conventions exist and how they guide you towards better object-oriented design. As always, leverage the "In Control" prompts to explore further with your AI assistant.

Ask your Chat BOT: "Beyond MVC and MVVM, what are one or two other architectural patterns commonly used in software development, and how do they relate to OOP principles?"

1. Architectural Patterns: Organizing Code with OOP

Large applications require structure. Architectural patterns provide proven templates for organizing code into distinct parts with specific responsibilities, leveraging OOP principles extensively.

a) Model-View-Controller (MVC)

MVC is a classic pattern, particularly prevalent in web application frameworks like ASP.NET Core MVC. It divides an application into three main roles:

- **Model:** Represents the application's data, business logic, and validation rules. It's the core of the application's domain. It knows nothing about the View or Controller.
  - OOP Connection: Encapsulates data and behavior related to the domain. Often uses classes for entities and services. Abstraction is used to hide data storage details (e.g., via Repositories).
- **View:** Responsible for presenting the Model's data to the user and capturing user input. In web applications, this is typically the HTML rendered in the browser. It should contain minimal logic.
  - OOP Connection: Focuses on the presentation aspect, abstracting the underlying data structure provided by the Controller (often via a specific View Model object).
- **Controller:** Acts as the intermediary. It receives input (e.g., HTTP requests), interacts with the Model to process data or trigger actions, and selects the appropriate View to display, passing necessary data to it.
  - OOP Connection: Encapsulates request handling and workflow logic. Crucially, it depends on abstractions of the Model (interfaces like IUserService, IProductRepository) rather than concrete implementations, adhering to DIP.

How MVC Uses OOP: The core benefit of MVC is **Separation of Concerns**, a direct outcome of applying principles like SRP and Encapsulation. Each part handles its specific area, making the application easier to understand, test, and maintain. DIP is vital for decoupling the Controller from the concrete Model implementation.

b) Model-View-ViewModel (MVVM)

MVVM is commonly used in modern UI frameworks (like WPF, MAUI, and sometimes in frontend JavaScript frameworks). It also separates concerns but with slightly different roles, particularly suited for stateful UIs with rich data binding capabilities:

- **Model:** Same role as in MVC – data, business logic, domain entities, and services.
- **View:** Represents the UI elements. It binds directly to properties and commands exposed by the ViewModel. It ideally contains no code-behind logic.
- **ViewModel:** The intermediary that prepares data from the Model for display in the View and handles user actions via Commands. It exposes state (data) via properties (often implementing INotifyPropertyChanged) and behavior via Command objects (often implementing ICommand). It knows about the Model but not the View.

  – OOP Connection: Encapsulates UI state and presentation logic. Uses Commands (an application of the Command pattern, relying on interfaces/polymorphism). Exposes data via properties (Encapsulation). Depends on Model abstractions (DIP).

How MVVM Uses OOP: Like MVC, MVVM achieves **Separation of Concerns**. It heavily relies on **Data Binding** (an abstraction mechanism) and the **Command Pattern** (Abstraction, Polymorphism) to decouple the View from the ViewModel. DIP is key for injecting Model dependencies into the ViewModel.

Common Goal: Both MVC and MVVM use OOP principles (Encapsulation, Abstraction, DIP, SRP) to break down complex applications into manageable, loosely coupled, and more testable parts.

Ask your Chat BOT: “What are the key differences in responsibility between a Controller in MVC and a ViewModel in MVVM? How does this affect testability?”

2. Dependency Injection in ASP.NET Core: DIP in Practice

Regardless of whether you use MVC, Razor Pages, or Minimal APIs, ASP.NET Core is built from the ground up around Dependency Injection (DI). This isn't just a feature; it's a core design philosophy deeply rooted in OOP.

- **The Mechanism:** ASP.NET Core provides a built-in Inversion of Control (IoC) container.

  1. **Registration:** You tell the container which concrete classes implement which interfaces (e.g., "When someone asks for IUserRepository, give them an instance of SqlUserRepository"). You also specify the lifetime (how long the instance should live: Singleton, Scoped, Transient).
  2. **Resolution/Injection:** When a class needs a dependency (e.g., a Controller needing IUserRepository), it declares it as an interface parameter in its constructor. The DI container automatically creates (or retrieves) an instance of the registered concrete type and passes ("injects") it into the constructor.

- **Direct Links to OOP/SOLID:**

  - **Dependency Inversion Principle (DIP):** This is the primary principle DI implements. High-level modules (Controllers, Services) depend on abstractions (interfaces), not low-level concrete implementations. The IoC container manages the "inversion" of dependency creation.
  - **Single Responsibility Principle (SRP):** DI encourages creating small, focused services that do one thing well (e.g., EmailSender, OrderProcessor, UserDataValidator). These can then be easily injected wherever needed.
  - **Open/Closed Principle (OCP):** Applications become open for extension. Need to change how users are stored? Implement a new MongoUserRepository and change only the registration line in your Program.cs. Classes using IUserRepository don't need modification.
  - **Liskov Substitution Principle (LSP):** The container assumes that any registered implementation of an interface correctly fulfills the interface's contract, making implementations substitutable.

- Example (Conceptual Program.cs / Startup):

```
// Program.cs in .NET 6+
var builder = WebApplication.CreateBuilder(args);

// Register services with the container
// Map interface to implementation
builder
    .Services
    .AddScoped<IUserRepository, SqlUserRepository>();
builder
    .Services
    .AddTransient<INotificationService, EmailNotificationService>();

// ... other registrations

// Framework services often use DI too

// For MVC
builder.Services.AddControllersWithViews();

var app = builder.Build();
// ... configure middleware ...

// MVC routing
app.MapControllerRoute(name: "default",
  pattern: "{controller=Home}/{action=Index}/{id?}");
app.Run();
```

Example (Constructor Injection in an MVC Controller):

```
public class UsersController : Controller
{
    private readonly IUserRepository _userRepository;
    private readonly INotificationService
        _notificationService;

    // Request dependencies via
    // constructor interfaces
    public UsersController(
        IUserRepository userRepository,
        INotificationService notificationService)
    {
        // DI container provides concrete instances
        _userRepository = userRepository;
        _notificationService = notificationService;
    }

    public IActionResult Index()
    {
        var users = _userRepository.GetAll();
        _notificationService.NotifyAdmin(
            "User list viewed");
        // Pass data to the View
        return View(users);
    }
    // ... other actions
}
```

Ask your Chat BOT: “Explain the practical differences between Singleton, Scoped, and Transient service lifetimes in ASP.NET Core DI. Give an example scenario where choosing the wrong lifetime could cause problems related to object state (Encapsulation).”

Conclusions

The principles of OOP are not just theoretical constructs; they are the practical building blocks of modern software development in the .NET world. Indeed, the emphasis on abstractions and separation of concerns within these frameworks directly enables better testability. Verifying that our carefully designed classes and components function correctly in isolation is the crucial next step, leading us to the practice of unit testing.

Unit Testing and OOP: Verifying Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Why Unit Test?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Testing in Isolation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

The Arrange-Act-Assert (AAA) Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

.NET Testing Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Test-Driven Development (TDD): Guiding Design with Tests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Example: Testing with Dependency Injection and a Manual Stub

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Conclusions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Error Handling in C# and .NET Framework: OOP Aspects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Exception Handling Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Object-Oriented Aspects of Exception Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Inheritance Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Encapsulation of Error Information

Exception Type Specialization

Conclusions

Leveraging Modern C# Features for Enhanced OOP

1. Records (C# 9 and later)

2. Init-Only Setters (C# 9)

3. Pattern Matching Enhancements (C# 8 onwards)

4. Default Interface Methods (C# 8)

5. Required Members (C# 11)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Practical Applications and Examples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Order Processing System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

How this Example Demonstrates Key Concepts:

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/Object-Oriented_Programming_CSharp.