C#10 Design Patterns

Introduction to Design Patterns

Kevin Dockx
Architect

@KevinDockx https://www.kevindockx.com

Version Check

This version was created by using:

- .NET 6.0
- C# 10
- Visual Studio 2022

Version Check

This course is 100% applicable to:

- .NET 6.0
- C# 10

Relevant Notes

New course versions are regularly released:

- https://app.pluralsight.com/profile/author/kevin-dockx

Coming Up

Prerequisites, frameworks and tooling

Course structure

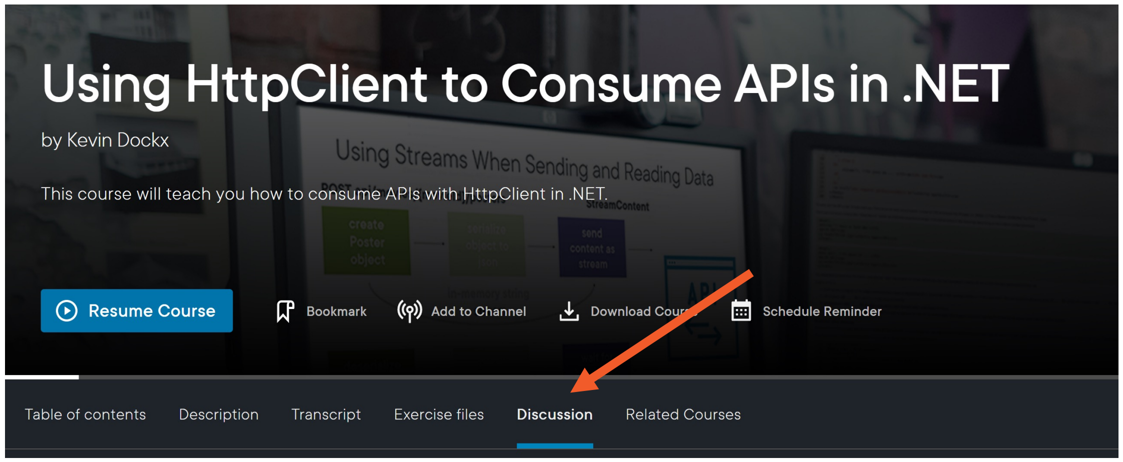
Design patterns

- Introduction
- Gang of Four
- Pattern types

Object-oriented principles refresher

Discussion tab on the course page

Twitter: @KevinDockx


Using HttpClient to Consume APIs in .NET

by Kevin Dockx

This course will teach you how to consume APIs with HttpClient in .NET.

(course shown is one of my other courses, not this one)

Course Prerequisites and Frameworks

C# 10

.NET 6

Course Structure

All 23 Gang of Four patterns are covered

- Each module covers one pattern

Course Structure: Module Structure

Intent of the pattern

Structure of the pattern

Real-life pattern implementation

(Depending on the pattern: variations, extensions, …)

Use cases, consequences & related patterns

A pattern use case tells you for which cases the pattern might be a good match

Pattern consequences can be positive and/or negative: consider implementing a pattern when the advantages outweigh the disadvantages for your use case

Patterns sometimes have comparable templates and implementations: learn how they compare, differ and can be combined.

Course Structure: Following Along

Follow the course from start to finish…

… or simply jump in wherever you want

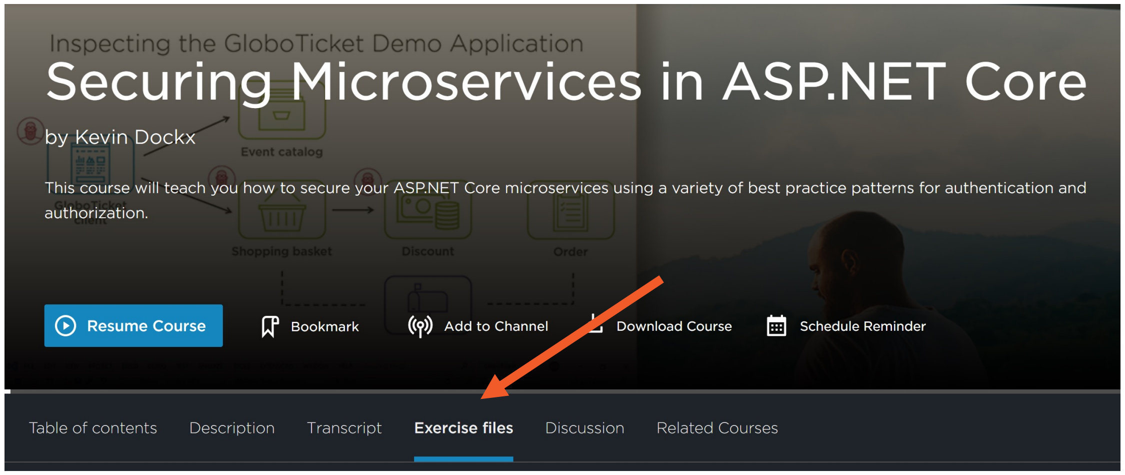
Course Structure: Pattern Types

Creational

Structural

Behavioral

Exercise files tab on the course page


Inspecting the GloboTicket Demo Application

Securing Microservices in ASP.NET Core

by Kevin Dockx

This course will teach you how to secure your ASP.NET Core microservices using a variety of best practice patterns for authentication and authorization.

(course shown is one of my other courses, not this one)

Design Pattern

A general, reusable solution to a commonly occurring problem within a given context in software design

Introducing Design Patterns

View design patterns as a template to start from

- Multiple implementations are possible

Each pattern has an intent

- The intent should remain the same, no matter how you implement the pattern

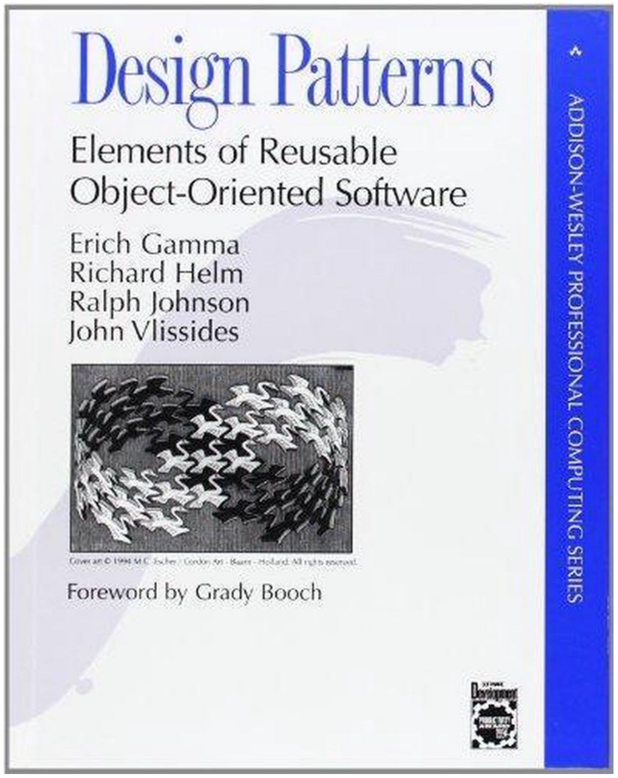
Introducing Design Patterns

Many patterns are so common you've probably already used them

Don't learn the pattern implementation from the top of your head, learn which problem a pattern solves

The Gang of Four

Erich Gamma **Richard Helm** **Ralph Johnson** **John Vlissides**


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns - Elements of Reusable Object-Oriented Software

- 23 design patterns
- Published in 1994
- Still commonly used today

Examples of Problems Design Patterns Solve

How do I ensure only a single instance of a class exists?

How do I make two objects with a different interface work together?

How can I extend an object's interface without changing the underlying object?

How do I enable support for undo functionality?

And many more…

The Gang of Four

We'll use modern-day language features to implement these patterns

- But the intent will remain the same

This is very much a *current* course, not a 1994 course

Gang of Four Pattern Types

Creational

Structural

Behavioral

Creational Patterns

Five patterns

- Abstract Factory, Builder, Factory Method, Prototype, Singleton

These patterns deal with object creation

- Abstract the object instantiation process
- Help with making your system independent of how its objects are created, composed and represented

Gang of Four Pattern Types

Creational

Structural

Behavioral

Structural Patterns

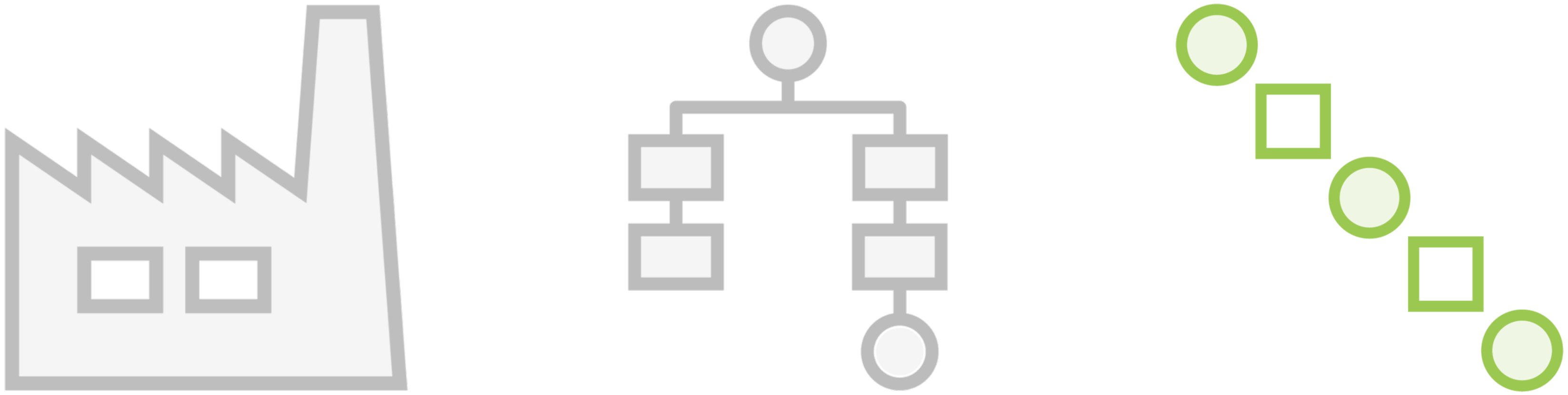
Seven patterns

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

These patterns deal with ways to define relations between classes or objects

- Concerned with how classes and objects are composed to form larger structures

Gang of Four Pattern Types



Creational

Structural

Behavioral

Behavioral Patterns

Eleven patterns

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

These patterns deal with ways to communicate between classes or objects

- Characterize complex control flow that's difficult to follow at runtime
- Let you concentrate on the way objects are interconnected

Object Oriented Principles Refresher

Program to an interface, not an implementation

Favor object composition over class inheritance

Program to an Interface, Not an Implementation

Clients remain unaware of the specific types of objects they use (as long as the objects adhere to the interface that clients expect)

Clients remain unaware of the classes that implement these objects. Clients only know about the interface.

```
public interface IDiscountService {
    int DiscountPercentage { get; } }

public class BelgiumDiscountService : IDiscountService {
    public int DiscountPercentage => 20; }

public class FranceDiscountService : IDiscountService {
    public int DiscountPercentage => 30; }

public class Client
{
    public Client(IDiscountService discountService)
    { // do something with discountService }
}
```

Program to an Interface, Not an Implementation

```
public interface IDiscountService {
    int DiscountPercentage { get; } }

public class BelgiumDiscountService : IDiscountService {
    public int DiscountPercentage => 20; }

public class FranceDiscountService : IDiscountService {
    public int DiscountPercentage => 30; }

public class Client
{
    public Client(IDiscountService discountService)
    { // do something with discountService }
}
```

Program to an Interface, Not an Implementation

```
public interface IDiscountService {
    int DiscountPercentage { get; } }

public class BelgiumDiscountService : IDiscountService {
    public int DiscountPercentage => 20; }

public class FranceDiscountService : IDiscountService {
    public int DiscountPercentage => 30; }

public class Client
{
    public Client(IDiscountService discountService)
    { // do something with discountService }
}
```

Program to an Interface, Not an Implementation

```
public interface IDiscountService {
    int DiscountPercentage { get; } }

public class BelgiumDiscountService : IDiscountService {
    public int DiscountPercentage => 20; }

public class FranceDiscountService : IDiscountService {
    public int DiscountPercentage => 30; }

public class Client
{
    public Client(IDiscountService discountService)
    { // do something with discountService }
}
```

Program to an Interface, Not an Implementation

This allows loose coupling

Program to an Interface, Not an Implementation

When the Gang of Four talks about the "interface", they're talking about the object's type: the set of requests an object can respond to

- Can be implemented with the interface language feature
- Can also be implemented with the abstract class language feature

```
public interface IDiscountService {
    int DiscountPercentage { get; } }

public class BelgiumDiscountService : IDiscountService {
    public int DiscountPercentage => 20; }

public class FranceDiscountService : IDiscountService {
    public int DiscountPercentage => 30; }

public class Client
{
    public Client(IDiscountService discountService)
    { // do something with discountService }
}
```

Program to an Interface, Not an Implementation

```
public abstract class DiscountServiceBase {
    public abstract int DiscountPercentage { get; } }

public class BelgiumDiscountService : DiscountServiceBase {
        public override int DiscountPercentage => 20; }

public class FranceDiscountService : DiscountServiceBase {
        public override int DiscountPercentage => 30; }

public class Client
{
    public Client(DiscountServiceBase discountService)
    { // do something with discountService }
}
```

Program to an Interface, Not an Implementation

Example with an abstract class

Use an abstract base class when you need to provide some basic functionality that can potentially be overridden

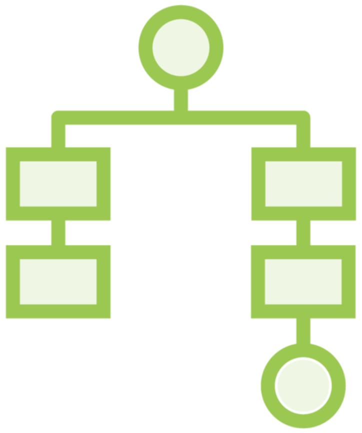
Use an interface when you only need to specify the expected functionality of a class

Program to an Interface, Not an Implementation

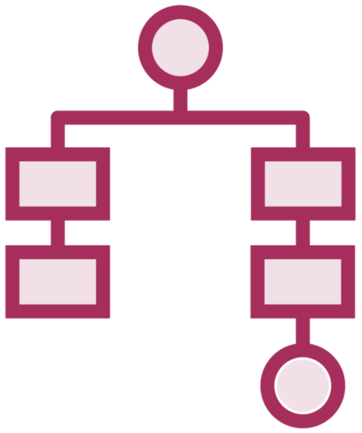
Commonly correlates to adhering to the open/closed principle (SOLID)

- Software entities (classes, modules, functions, …) should be open for extension, but closed for modification

Favor Object Composition Over Class Inheritance



Class inheritance



Object composition

Favor Object Composition Over Class Inheritance

Class inheritance

- Lets you define the implementation of one class in terms of another’s
- White-box reuse

Favor Object Composition Over Class Inheritance

Object composition

- New functionality is obtained by assembling or composing objects to get more complex functionality
- Black-box reuse

Favor Object Composition Over Class Inheritance

In most systems, both reuse techniques are commonly used

The Gang of Four will favor object composition over class inheritance where possible

- Inheritance tends to be overused
- Often simplifies designs and makes them more reusable

Favor Object Composition Over Class Inheritance

Commonly correlates to adhering to the single responsibility principle (SOLID)

- A class should have one, and only one, reason to change

Summary

A design pattern is general, reusable solution to a commonly occurring problem within a given context in software design

- Look at them as a template

Summary

Three GoF pattern types:

- **Creational patterns** help with making your system independent of how its objects are created, composed and represented
- **Structural patterns** are concerned with how classes and objects are composed to form larger structures

Summary

Three GoF pattern types:

- **Behavioral patterns** characterize complex control flow that's difficult to follow at runtime, and let you concentrate on the way objects are interconnected

Summary

Object-oriented principles used by the Gang of Four

- Program to an interface, not an implementation
- Favor object composition over class inheritance

Up Next:
Creational Pattern: Singleton