***Design Patterns Using C# and .NET Core***

Design patterns are reusable solutions to common programming problems. They speed up the development process by providing tested, proven development paradigms. This course starts at the nuts-and-bolts level and shows you everything through to advanced patterns and features, going in-depth to give you the knowledge you need.

You will begin this course with an introduction to SOLID principles, which will introduce you to clean-code concepts and will elevate your skills. With this knowledge in your toolbox, you will be ready to move on to studying creational design patterns, patterns related to the creation of objects, such as Singleton, Factory, and Dependency Injection. You will then acquire more in-depth knowledge of one specific creational pattern, the Dependency Inversion pattern, which will teach you how to write highly extensible, maintainable, and testable code. Moving on, you will get your hands dirty with structural design patterns and you will complete this course by learning the last group of patterns: behavioral design patterns.

By the end of this course, you will be very confident in implementing a new feature in C# and .NET Core because, every time you have a problem, the correct design pattern will spontaneously come to mind.

This course uses .NET Core 2.0, and VS2017 community edition+, while not the latest version available, it provides relevant and informative content for legacy users of .NET Core, and Visual Studio.

**SOLID Principles**

SOLID principles can upgrade the code someone writes. The problem is, they are complex.

* Describe what SOLID means
* Brief summary of all principles
* Introduction to SOLID
* Single Responsibility
* Open/closed principle
* Liscov substitution
* Interface segregation
* Dependency inversion

SOLID:

* S = Single Responsibility Principle – a class/module should be responsible for managing a single part of the app’s functionality
* O = Open/Closed Principle – a class should be open to extension but closed to modification. Whenever you need to add more functionality to a specific class, the ideal course of action would be to extend the class through inheritance or composition, instead of modifying it to add more methods or functions
* L = Liskov Substitution Principle – also known as “substitutability”. You should write your code in a way, that anywhere you need a parent class from an inheritance tree, you should be able to use one of its subclasses or children
* I = Interface Substitution Principle – forces you to split interfaces into smaller ones, so that no class is forced to implement a method that it either does not need or cannot implement properly. (A class should not depend on methods that it does not need to implement)
* D = Dependency Inversion Principle – aims to decouple your dependencies so that your classes and modules should not depend on concrete implementations. Rather, they should depend on decoupled abstractions. (Your classes and modules should depend on abstractions instead of concrete implementations)

It’s fairly common for developers to write classes that do more than one thing.

* Demonstrate an example of such a class
* Refactor the class to adhere to the single responsibility principle

Single Responsibility Principle

* When is this principle violated?
  + You have error handling code in your class
  + You have presentation logic
  + File/Database Read/Write

Example of Error Handling in Class:

public async Task doThing()

{

try

{

... <---*Class responsibility*

}

catch(InvalidArgumentException e)

{

var messegeHandler = MessageHandlerFactory.CreateInstance();

messageHandler.show(“Oops, something went wrong”);

var loggingEndpoint = “https://myapi.com/v1/logging”;

var http = new CustomHttpClient();

http.post(loggingEndpoint, e.Message);

}

catch(Exception e)

{

var messegeHandler = MessageHandlerFactory.CreateInstance();

messageHandler.show(“Critical Error”);

var loggingEndpoint = “https://myapi.com/v1/logging”;

var http = new CustomHttpClient();

http.post(loggingEndpoint, e.Message);

}

}

public async Task FetchClientInfo()

{

var http = new MyHttpClient();

ClientInfo = await http.getClientInfo();

StringBuilder html = new StringBuilder();

builder.Append(“<div class=’clientInfo’>”);

builder.Append(“<label>First Name: </label>”);

builder.Append($”<span>{clientInfo.FirstName}</span>”);

builder.Append(“<label>Last Name: </label>”);

builder.Append($”<span>{clientInfo.LastName}</span>”);

builder.Append(“<label>Orders: </label>”);

builder.Append($”<span>{clientInfo.Orders)</span>”;

builder.Append(“</div>”);

}

public async Task raiseSalaries(Employee[] employees, double percentage)

{

for(var employee in employees)

{

employee.Salary = employee.Salary + employee.Salary \* percentage

}

using(var f = new StreamWriter(“myfile.txt”))

{

for(var employee in employees)

{

f.WriteLine(employee.ToString());

}

}

}

Open/Closed Principle – Open to Extension, Closed to Modification

The open/closed principle is rarely used as developers keep adding stuff to their classes.

* Describe what closed to modification means
* Explain what open to extension means
* Demonstrate a comprehensive example

You should not add stuff to completed classes

Try to keep code as dry as possible, so if there is any new functionality that you want to add, always add it to new class using inheritance.

To allow a function to be overridden in .NET, the function needs to have virtual keyword, and then the override function should have override keyword.

public static class CountriesFormatter

{

public virtual string FormatForConsole(Country country)

{

return $"{country.Name}, Capital: {country.Capital} - Region: {country.Region}";

}

}

public static class CountriesHTMLFormatter

{

public override string FormatForConsole(Country country)

{

return $"<div class=’country’> <h1>{country.name}</h1> <h2>{country.Capital}</h2> </div>";

}

}

The Liskov Substitution Principle / Substitutability

The Liskov substitution principle is perhaps the hardest of the five. We need to provide a high quality explanation without making the video long and boring.

* Theoretical examination
* Use a quiz app example

A subclass should always be able to be used wherever its parent can be used without altering the functionality in a blocking way. The program shouldn’t hang or crash or have any type of undefined behavior just because we swapped a parent class for a sub/child class.

Example: quiz questions that have four answers that are passed to the program, but a subclass of true-false questions that inherit from the quiz question quiz question class only have two answers. For the format function, adding a blank or a “-“ for the last two questions would fix the error issue.

The Interface Segregation Principle

According to the interface segregation principle, a class shouldn’t be forced to implement a method that it does not need or can’t implement.

* Go straight to an example of an interface that is wrongfully used
* Split that interface into two interfaces
* Implement them separately

Interface segregation is NOT a problem, it is actually a ***desirable*** outcome.

How to recognize that you need to segregate your interfaces:

* You don’t know how to implement an interface method
* You get the feeling that your interface method does not belong in the class
* You are forced to leave the method empty
* You are forced to throw a generic exception

Example given is with birds – all birds eat and walk, but not all birds fly. Interfaces created to house walking method, eating method, and flying method, then individual birds can inherit the interfaces, rather from the bird class.

The Dependency Inversion Principle – *classes and modules should depend on abstractions instead of concrete implementations*

Dependency Inversion is so vital that a part of it, Dependency Injection will have its own section. But for now we need to demonstrate how important it is.

* Explain Dependency Inversion and loose coupling in theory
* Demonstrate tight coupling’s downsizes
* Implement a service the correct way by connecting to an abstraction

Abstractions = Interfaces

Example: Computer Mice

* Wired
* Wireless
* Touchpad
* Laser
* Optical
* Trackball

If your computer relied on an actual implementation, you would need the exact type of mouse for it to work, otherwise your computer would need six different ports in order to be able to use those six different types of mice.

Instead, the computer doesn’t even depend on something as simple as a USB as a common interface. It relies on something even more abstract – an entity with two buttons and a pointer.

Implementation does not matter.

This example explains the idea of loose coupling.

***Introduction to Design Patterns***

In this video, we will work almost exclusively with patterns, so introductions are in order.

* Describe design patterns in general
* Briefly explain each creational pattern that we are going to study
* Introduction to Design Patterns
* The Singleton Pattern
* The Factory Pattern
* Loose Coupling
* The Object Pool

Design Patterns – *generalized, reusable solutions to common design issues in software engineering. The have been proven to be the best solutions to a common problem in software engineering.*

“Don’t reinvent the wheel”

Everytime you’re struggling to come up with an architectural breakthrough, chances are there is a design patter than will provide a clean solution for you.

Design patterns are language agnostic – once you learn them in one, you can apply them to others (C#, Java, etc.)

Design pattern groups:

* Creational – all about instantiating objects. How they are instantiated and how many instances of them we are allowed to create.
* Structural – the app’s structure. Sometimes there are complex classes or objects that are composed from smaller classes or objects. Structural design patterns offer the most efficient way to work in situations like this and solve any challenges that may arise.
* Behavioral – determine ways of application flow and how objects communicate with each other and how they interact with each other.

Course Project: A console application simulation of a card game called “Guardians of the Code”

***The Singleton Design Pattern***

Sometimes we need to only have a single instance of a class.

* Describe singleton theoretically
* Implement singleton with a PrimaryPlayer class
* Note limitations and downsides

Singleton – *you can have only a single instance of a specific class throughout the entire application*.

Try as much as you can to not use the work ‘new’ in the code.

Singleton Pattern Benefits:

* Shared State – singleton pattern can be used to store the application’s state. Common in mobile or console applications. Not as common in web apps or API’s.
* Avoid long initializations – if a class has complicated logic in its constructor, you don’t want to have to instantiate it multiple times.
* Cross-class communication
* Perfectly represents unique items

Key Implementation Points

* Constructor must absolutely be private (empty constructor)
* Instance variable must be read only – we don’t want other classes to be able to change where the instance reference points to
* Static Instance
* Static Accessor
* Class should be sealed to prevent inheritance (not mandatory)
* Instance should be initialized inside a static constructor (NOT the private one from the first bullet up top)

***The Factory Pattern***

Following the single responsibility principle, our consumer classes shouldn’t be the ones responsible for knowing how to instantiate classes that we need.

* Describe the factory pattern in theory
* Create an EnemyFactory

Factory Pattern – *a combination of single responsibility and open/closed principles*.

Single responsibility is expressed at the gameboard level. The gameboard shouldn’t have to know how to instantiate the enemies or what enemies to create.

In the example, the enemy factory is essentially a class with three methods that instantiate the different types of enemies, whose classes are all created based on the enemy interface.

***Loose Coupling***

We need to take care of a few stuff in the course project. This presents a nice opportunity to involve the concept of Loose Coupling.

* Create an Iweapon interface
* Create different weapon implementations
* Add them to the player and attack an enemy

Loose Coupling – *software parts that communicate with each other but have little to no knowledge of each other’s actual implementation*

Software components that are loosely coupled are often called Black Boxes.

Loose Coupling Benefits:

* Makes it easier to work with large projects. Conventions always help with large or enterprise level applications and loose coupling is no exception.
* Loose coupling also allows you to swap implementations when you need them, and even on the fly with a strategy pattern.
* With Loose Coupling, code becomes much more testable. When your dependencies are decoupled, you don’t need to worry about using a real database for testing.
* Allows components to grow independently. Other developers can work on them, many lines of code can be added to them, but as long as they follow the obstructions, they will communicate quite well with each other.

***The Object Pool***

The Object Pool is a great pattern to avoid having to be constantly initializing our enemy classes every time we need them.

* Modify the EnemyFactory to work with object pools
* Create a pool for the Zombie class
* Add a reclaim mechanism

Object Pooling – *a pool/collection of pre-initialized objects whose initialization is heavyweight. Every time we need such an object, we take one from the pool and return it back to the pool. Normally, objects that take longer to initialize are stored here. Around 20 or so are created, then are used when needed, and when they are finished being use, are returned to the pool instead of being destroyed.*

There is debate in the software development community of whether or not the Object Pool pattern should be deprecated. However, there are still use cases where it makes sense to still use it. One major one is game development.

***Dependency Injection***

Not all C# developers are familiar with Dependency Injection and since we are dedicating a whole section on it we better spend a video trying to explain the most interesting topics.

* Explain Dependency Injection thoroughly
* Show a very simplistic visual example
* Introduction to Dependency Injection (DI)
* Creating a Custom Container
* The built-in ASP.NET Core IoC (Inversion of Control) Container
* Singleton vs Transient vs Scoped

With Dependency Injection, the goal is to use the word ‘new’ as little as possible. However, an OOP app cannot exist without using it. If you have a value object or a service data object, or generally a short-lived object that does not depend on other entities of your app, you are free to instantiate it with the term ‘new’. Factories are also welcome to use the term.

You are not allowed to use ‘new’ with dependencies.

***Creating a Custom Container***

There’s great value in creating a Custom IoC Container. The viewer will get to understand the inner workings of Dependency Injection.

* Describe the vital operations that an IoC Container needs to be doing
* Implement the registration method
* Implement the Dependency resolution method

***The Built-In ASP.NET Core IoC Container***

The Custom Container is good for educational purposes but we need something more robust. So we have to move on to the built-in ASP.NET Core IoC Container.

* Create an ICardsService interface
* Create a CardsService concrete implementation
* Register the implementation as a Singleton and inject it to a controller
* How to register dependencies with the ASP.NET Core Container
* Hot to resolve dependencies
* How to create a simple API for our card game

***Singleton vs Transient vs Scoped***

The ASP.NET Core Container offers three different registration strategies. We need to make the distinctions clear.

* Describe the three strategies and decide on the best one for the task at hand
* Create a method that tests the API connection
* Consume the API project from the console application
* What is the difference between Singleton, Transient, and Scoped?
* How to call the API we created from our console app

Singleton – *The service is instantiated once and used throughout the application. It is instantiated the first time it is requested.*

Transient – *A new instance is created every time the service is requested.*

Scoped – *A new instance is created for every Http request NOT per service request.*

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

<http://localhost:5053>

**Structural Design Patterns**

* Decorator
* Adapter
* Façade
* Composite
* Proxy

***The Decorator Pattern***

Sometimes we want to combine objects while maintaining their types. To do that we use the decorator pattern.

* Use figures to explain what the decorator pattern does
* Use the decorator pattern to decorate cards with items

Current problem we are trying to solve: Combining Cards

Requirements:

* Multiple types of cards exist
* We want to be able to combine those cards
* We want all cards to have a common interface

A screenshot of a diagram

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Item Card is described as a decorator, and it holds a reference to Card. The question is why would something that is seemingly a Card hold a reference to its base class?

A diagram of a decorator

Description automatically generated

The goal of the decorator is to combine a single concrete class with one or more decorators.

A screenshot of a pattern

Description automatically generated

***The Adapter Pattern***

We must make use of a foreign interface. We need to make it work with what we already have. That’s what the adapter pattern is for.

* Introduce a 3rd party library to the project
* Work with the adapter pattern to circumvent it
* We will take a look at what the Adapter Pattern is
* How we can use it to connect similar objects with different interfaces

This pattern is rarely used by developers, but it is a great solution to common struggles that are faced during development, especially when using third-party libraries.

Adapter Pattern – *converts an interface of a class to one expected by the consumer. We use the adapter pattern when we want to consume a class that implements an interface that is different or varies slightly from the one our client class(es) expect.*

A screenshot of a computer

Description automatically generated

Currently the code is written so the player class expects the IWeapon interface, but if we want them to be able to use space weapons, we will need to build a WeaponAdapter class that implements IWeapon.

***The Façade Pattern***

The Gameboard class is getting more and more complicated.

* Theoretical introduction to the facade pattern
* Use the facade pattern to simplify the Gameboard
* What the Façade Pattern is
* How to build a Gameboard façade to simplify our code

The Façade Pattern – *The goal of the façade pattern is to simplify a complex interface*

Steps to play a level:

* Configure a player’s weapon(s)
* Add player cards
* Initialize enemy factory
* Load Zombies
* Load Werewolves
* Load Giants
* Play

The game is played in turns where the player is asked if they want to use their weapon to attack one of the enemies or use one of their cards. This whole process is fairly complex and we wouldn’t want our caller class to have to repeat all of these steps every time the level is changed, which is where the façade pattern comes in.

The façade pattern simplifies this interface of seven methods into a class into one single method that only takes two arguments: Play(PrimaryPlayer player, int areaLevel)

**Principle of Least Interest (Law of Demeter)** – *Every component should have little knowledge of how other components work and only communicate with a few specific close “friends”*. *Every component in our application should need limited to no knowledge of how other components work. It should also only communicate to a few “friends”. Components should only exchange information with its “neighbors”. This is also known as the “don’t talk to strangers” principle.*

***The Composite Pattern***

We want to provide a way to iterate through decks of cards as well as decks of decks.

* Describe the composite pattern
* Create a common IComposite interface
* How to use tree-like structures to describe part-whole collections

The Composite Pattern – *used to create part-whole collections in the form of tree-like structures that can contain both individual items and collections as well. A part-whole collection is represented below:*

A diagram of a deck card

Description automatically generated

We need a unified, recursive way to have access to all of our decks. The solution proposed by the composite pattern is to have an interface called ICardComponent. Both Cards and CardDecks will implement this common interface. We store a list of ICardComponents in the CardDeck.

\*\*\*Visual Studio Note – F2 allows you to replace all references of a variable if you want to rename it\*\*\*

***The Proxy Pattern***

We want to quickly and easily fetch data from the API in a loosely coupled way.

* Decouple the service from the implementation
* Use a proxy class to fetch the data
* What the proxy pattern is and what it means
* How to decouple our services using the proxy pattern

Current design:

A screenshot of a computer

Description automatically generated

Desired/Suggested design:

A screenshot of a phone

Description automatically generated

The proxy pattern specializes in calling a remote method. In our case, that method is one that returns the cards’ injection format and of course runs in our API project. The pattern’s purpose is to make this communication flawless in a way that the caller can’t tell whether the data came from the same application or from another application running on the same system, or perhaps from an API running anywhere in the world. The proxy as a middleman between the two. In a way, it acts as a service, but it’s main purpose is to make it seem like the call is not coming from across the network. The proxy pattern was used a lot in TestOps C# and mainly Java applications, but it its use is failing in a world of RESTful applications.

Behavioral Design Patterns:

* Strategy Pattern
* Observer Pattern
* Command Pattern
* Template Method
* State Pattern

***The Strategy Pattern***

Sometimes we need to choose an algorithm at runtime. The strategy pattern is perfect at doing that.

* Describe the need for the strategy pattern
* Create two strategies for showing messages
* What the Strategy Pattern is, how it works, and why we need it
* How to implement a damage notification system using strategy

The primary advantage that the strategy pattern has is the ability to choose the most appropriate algorithm from a group of similar algorithms at runtime.

***The Observer Pattern***

Watching the primary player’s health is better than just using strategy.

* Create a HealthChanged event
* Register observers
* Combine them with the strategy pattern
* What the observer pattern is by example
* We will use it to register an event listener so that every time a player is taking damage we are able to log it somewhere (How to use the observer patter to better log the damage)

The concept is pretty simple – you have an entity, a subject that you might want to watch for changes. It could be a property, or it could be an event that’s usually triggered. Then, you have a list of observers that register to that property, and when that property changes, they are notified back with the change, and each and every one of them may behave differently depending on their specific task. But all of them will be notified at the same time as a group for the change of the property.

Implementing the observer pattern in C# is interesting because it is built-in to the language and the framework themselves in the form of events.

So, instead of building the whole functionality, the whole observer communication functionality from the beginning, shortcuts are available with events to simplify the amount of code that needs to be written.

***The Command Pattern – slightly more difficult***

We need a way to encapsulate the battles between players and enemies as well as cards and enemies.

* Create an ICommand interface
* Create two command implementations
* Group them into a list and execute
* What the command pattern is
* How to use the command pattern to unify the battling system and bridge the gaps between players, cards, enemies, and achieve the cleanest code possible

We want to take advantage of the command pattern to create a unified battling system so that we do not care whether it is the player’s turn to attack or if it’s a card that’s attacking the enemy and not the player – this will help us avoid a pyramid of if-else statements and focus on what really matters.

To achieve this, we will encapsulate the battle between the two entities in a command.

Command vs Strategy

* The strategy pattern handles the objects(just reads them and does something with them), the command also changes their state
* When using the strategy pattern, usually there is a set of interchangeable algorithms of similar functionality and you swap them depending on the situation
* The command pattern allows the execution of different encapsulated commands. That way you can group them together in a list and execute them as a group, therefore avoiding the constant use of conditionals.
* The commands can be stored in a command pattern, so you can store a copy of a command you perform. This allows you to revert to a previous state and undo the command, as well as reuse the command.

***The Template Method***

The template method defines an abstract base class that forces sub-classes to only implement part of the functionality.

* Create a template for rendering the battlefield
* Create different battlefields that will be randomly generated
* In the video, we will take a look at what the Template Method is and
* Demonstrate the Template Method by an example of creating multiple battlefields in our application

***The State Pattern***

The state pattern offers a convenience class to store data about your sessions.

* Create a state class
* Store statistics there
* In this video, we will look at: What the State Pattern is
* How we can use it to simulate the game’s multiplayer mode subscription system
* How to use the state pattern to save statistics for the game

A screenshot of a computer

Description automatically generated

The State Pattern is great at imitating states in your application. The above diagram is an example of a Finite State Machine Diagram. We begin with three states for our multiplayer mode subscription.

The first is the trial mode in which the user is given a limited amount of time to test the app for free. If the time expires, we have the Trial Expired state. If the user pays, we move to the paid state.

A screenshot of a diagram

Description automatically generated

In order for the diagram to be complete, we need actions to change the state. Those actions include the Pay action that moves a player from the Trial subscription to the Paid subscription.

That same method can move a player from an Expired Trial state to a Paid Subscription state.

Finally, we have the Expire method, which stops the user’s Trial subscription when the trial period ends.

Summary:

* Began with the Strategy Pattern that allowed us to swap similar algorithms at runtime depending on our state
* We then used that in combination with the Observer Pattern to observe objects for changes
* We then encapsulated the logic in commands
* Moved on to using the template method pattern to build battlefields
* Finally, the state pattern was used to simulate a subscription management system