

# ESSENTIAL SOFTWARE DESIGN PATTERNS



LOUISVILLE  
.NET MEETUP

# Thank You to our Sponsors





# Upcoming LouDotNet Meetups



June 20, 2024



July 18, 2024



August 22, 2024

# Azure Meetups

- June 13
  - Unleashing Extreme Scalability with Azure Functions



LOUISVILLE  
AZURE MEETUP

# Upcoming Events



Scenic City Summit  
Chattanooga, TN  
June 28, 2024  
\$99  
[sceniccitysummit.com](https://sceniccitysummit.com)



Cincy Deliver  
Mason, OH  
July 26, 2024  
\$65 through May; \$75  
[cincydeliver.org](https://cincydeliver.org)

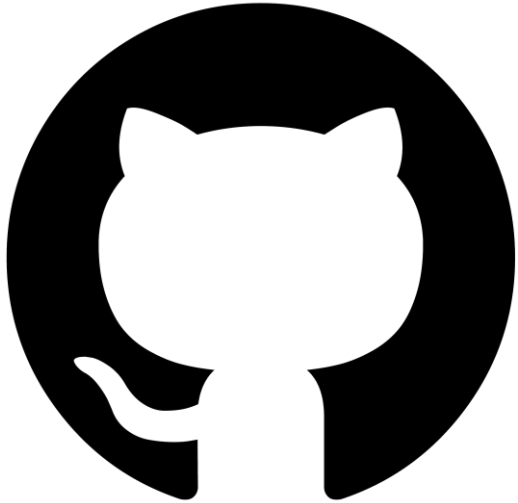


dev up  
St. Charles, MO  
August 14 – 16, 2024  
\$450 GA; \$225 Workshop  
[devupconf.org](https://devupconf.org)

# After Session Social



# New Meetup Repository



[github.com/TaleLearnCode/LouDotNet](https://github.com/TaleLearnCode/LouDotNet)

- Meetup Schedule
- Slides and Demos from Past Meetups
- Upcoming Events (coming soon)
- Resources



# ESSENTIAL SOFTWARE DESIGN PATTERNS





# Who is Chad Green?

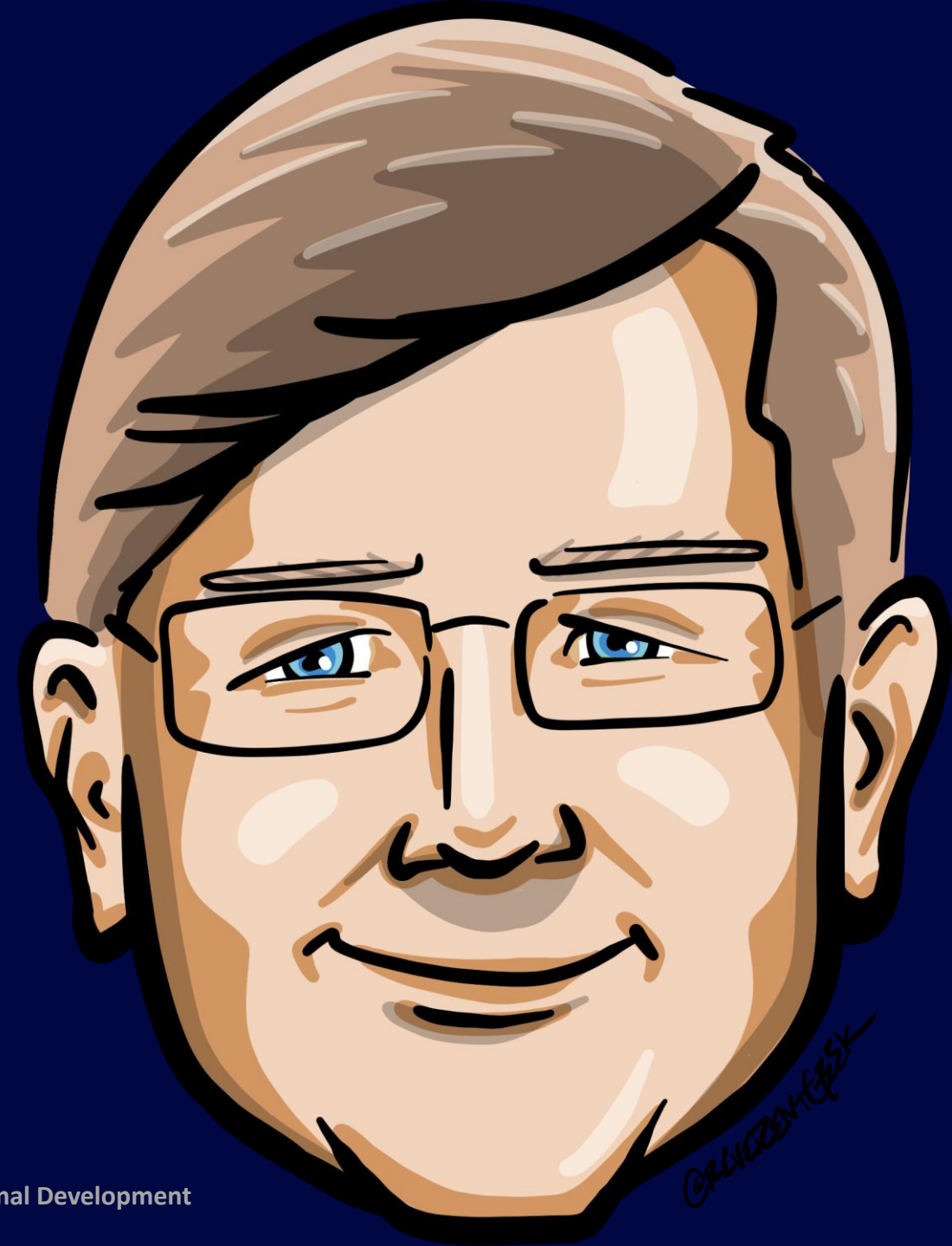
✉ chadgreen@chadgreen.com

💬 TaleLearnCode

🌐 ChadGreen.com

🐦 ChadGreen & TaleLearnCode

📌 ChadwickEGreen



# What Are Design Patterns

Essential Software Design Patterns for Optimal Development

# What Are Design Patterns

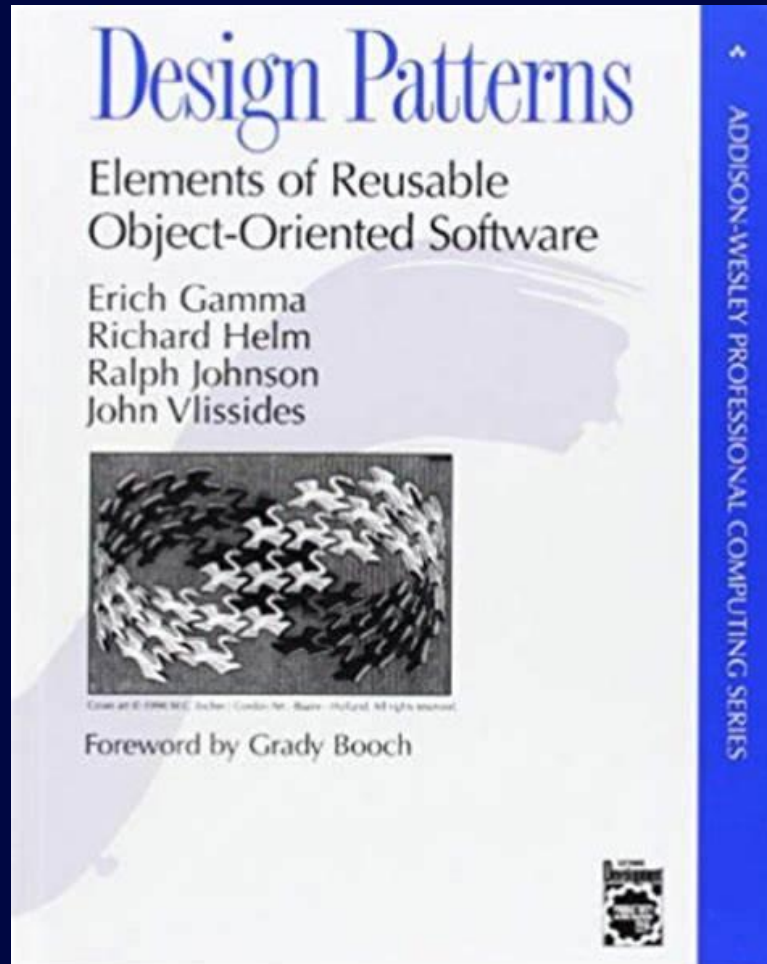
- Reusable solutions to common problems
- Best practices and proven solutions
- Building blocks for maintainable, scalable, and robust software



# Why Design Patterns Matter

- Address complexity
- Encourage best practices and standardization
- Enhance code readability and maintainability
- Facilitate collaboration

# Gang of Four



Essential Software Design Patterns for Optimal Development

# Types of Design Patterns

**Creational**

**Structural**

**Behavioral**



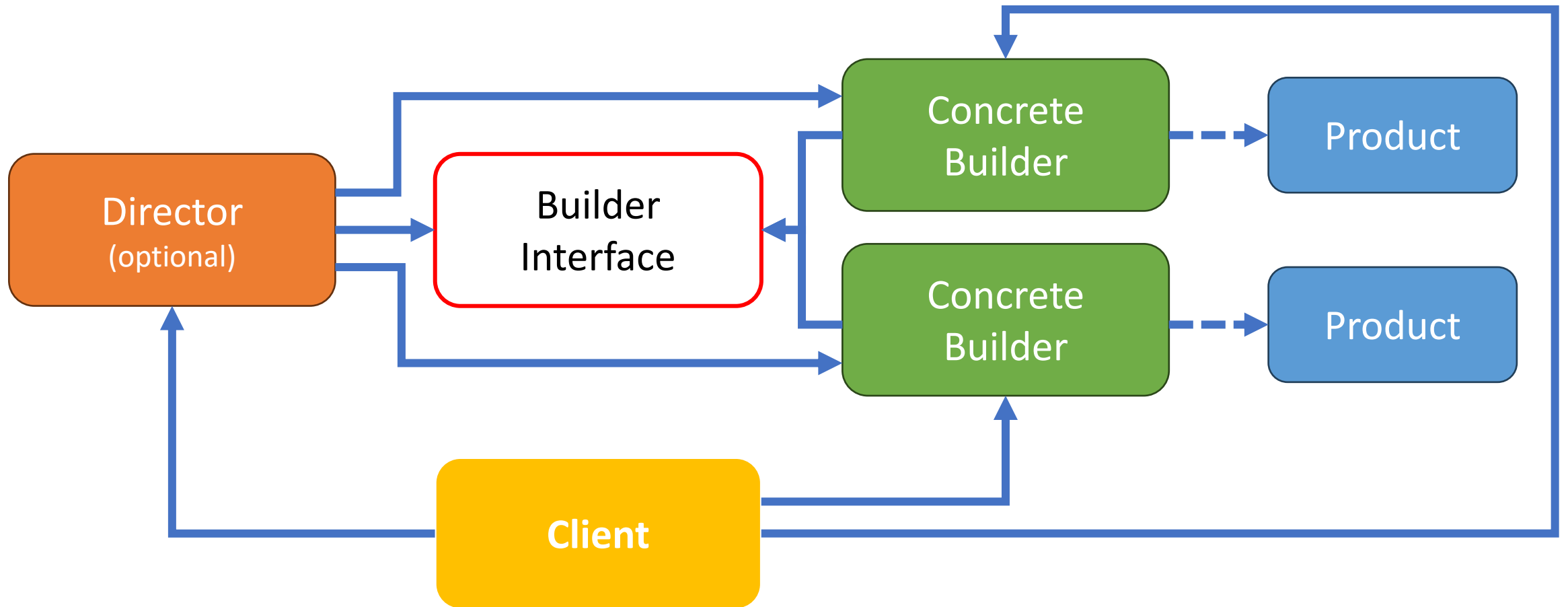
# Creational Design Patterns

Essential Software Design Patterns for Optimal Development

# Builder Pattern

Creational Design Patterns

# Builder Pattern





# Builder – Product

```
public class Pizza
{
    public string Crust { get; set; }
    public string Sauce { get; set; }
    public List<string> Toppings { get; set; } = [];
}
```

# Builder – Builder Interface

```
public interface IPizzaBuilder
{
    void BuildCrust();
    void BuildSauce();
    void BuildTopping();
    Pizza GetPizza();
}
```

# Builder – Concrete Builder

```
public class HawaiianPizzaBuilder : IPizzaBuilder
{
    private readonly Pizza _pizza = new();

    public void BuildDough() => _pizza.Crust = "Original";

    public void BuildSauce() => _pizza.Sauce = "Classic Marinara";

    public void BuildTopping() => _pizza.Topping = ["Ham", "Pineapple"];

    public Pizza GetPizza() => _pizza;
}
```



# Builder – Director

```
public class Waiter(IPizzaBuilder pizzaBuilder)
{
    private readonly IPizzaBuilder _pizzaBuilder = pizzaBuilder;

    public void ConstructPizza()
    {
        _pizzaBuilder.BuildDough();
        _pizzaBuilder.BuildSauce();
        _pizzaBuilder.BuildTopping();
    }

    public Pizza GetPizza() => _pizzaBuilder.GetPizza();
}
```

# Builder - Client

```
IPizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();  
Waiter waiter = new(hawaiianPizzaBuilder);  
  
waiter.ConstructPizza();  
Pizza pizza = waiter.GetPizza();  
  
Console.WriteLine($"Crust: {pizza.Crust}\nSauce: {pizza.Sauce}\nToppings:  
    {string.Join(", ", pizza.Topping)}");
```

# Builder Pattern in .NET

## StringBuilder

```
StringBuilder builder = new();  
builder.Append("Hello");  
builder.Append(" ");  
builder.Append("World");  
string result = builder.ToString();
```

# Builder Pattern in .NET

## Fluent Interfaces

```
// Entity Framework Core example
var query = dbContext.Users
    .Where(u => u.Age > 18)
    .OrderBy(u => u.LastName)
    .Select(u => u.FullName);

// NUnit assertion example
Assert.That(result, Is.EqualTo(expected).IgnoreCase.And.StartWith("Hello"));
```



# Builder Pattern in .NET

## ASP.NET Core Options

```
services.Configure<MyOptions>(options =>  
{  
    options.Option1 = "value1";  
    options.Option2 = "value2";  
});
```

# Builder Pattern in .NET

## JsonSerializerOptions

```
var options = new JsonSerializerOptions
{
    PropertyNameCaseInsensitive = true,
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    WriteIndented = true
};
```

# Builder Pattern

## Benefits

- Separation of Concerns
- Encapsulation
- Reusability
- Complex Object Construction
- Control Over Construction Process
- Immutability

## Times When Not to Use

- Increased Complexity
- Boilerplate Code
- Potential Overhead
- Duplication of Code
- Limited Applicability
- Potential for Inconsistency

# Builder Pattern

## Times to Use

- Complex Object Construction
- Variability in Object Representation
- Immutability and Thread Safety
- Telescoping Constructors  
Antipattern
- Creation of Composite Objects
- Configurable Object Creation
- Testing

## Times When Not to Use

- Simple Object Construction
- Static Configuration
- Limited Variability
- Performance-Sensitive Applications
- Tight Deadlines/Rapid Prototyping
- Highly Coupled Objects

# Factory Pattern

Creational Design Patterns



# Abstract Product

```
public abstract class Animal
{
    public abstract void Speak();
}
```

# Concrete Product

```
public class Dog : Animal
{
    public override void Speak() => Console.WriteLine("Dog says: Bow-Wow.");
}

public class Cat : Animal
{
    public override void Speak() => Console.WriteLine("Cat says: Meow.");
}
```

# Concrete Factory

```
public static class AnimalFactory
{
    public static Animal CreateAnimal(AnimalType animalType)
        => animalType switch
        {
            AnimalType.Dog => new Dog(),
            AnimalType.Cat => new Cat(),
            _ => throw new ArgumentException("Invalid animal type."),
        };
}
```

# Client Code

```
Animal dog = AnimalFactory.CreateAnimal(AnimalType.Dog);  
dog.Speak();
```

```
Animal cat = AnimalFactory.CreateAnimal(AnimalType.Cat);  
cat.Speak();
```

# Factory Pattern in .NET

## DbProviderFactory

```
DbProviderFactory factory =  
DbProviderFactories.GetFactory("System.Data.SqlClient");  
using (DbConnection connection = factory.CreateConnection())  
{  
    connection.ConnectionString = connectionString;  
    connection.Open();  
    // Use connection...  
}
```



# Factory Pattern in .NET

## System.IO.Stream

```
using (Stream stream = File.Open("example.txt", FileMode.Open))  
{  
    // Use stream...  
}
```

# Factory Pattern in .NET

## System.Text.Encoding

```
Encoding encoding = Encoding.GetEncoding("UTF-8");  
byte[] bytes = encoding.GetBytes("Hello, World!");
```

# Factory Pattern

## Benefits

- Encapsulation
- Loose Coupling
- Enhanced Code Maintainability
- Scalability and Flexibility
- Improved Testability
- Consistency in Object Creation

## Drawbacks

- Increased Complexity
- Performance Overhead
- Overuse and Misuse
- Hidden Dependencies

### Times to Use

- Database Connection Management
- Logging Framework
- Parsing Different File Formats
- Payment Processing Systems
- Shape Creation
- Manufacturing

### Times to Avoid

- Simple Object Creation
- Performance-Critical Apps
- Infrequent Changes to Object Creation Logic
- Static Configurations

# Structural Design Patterns

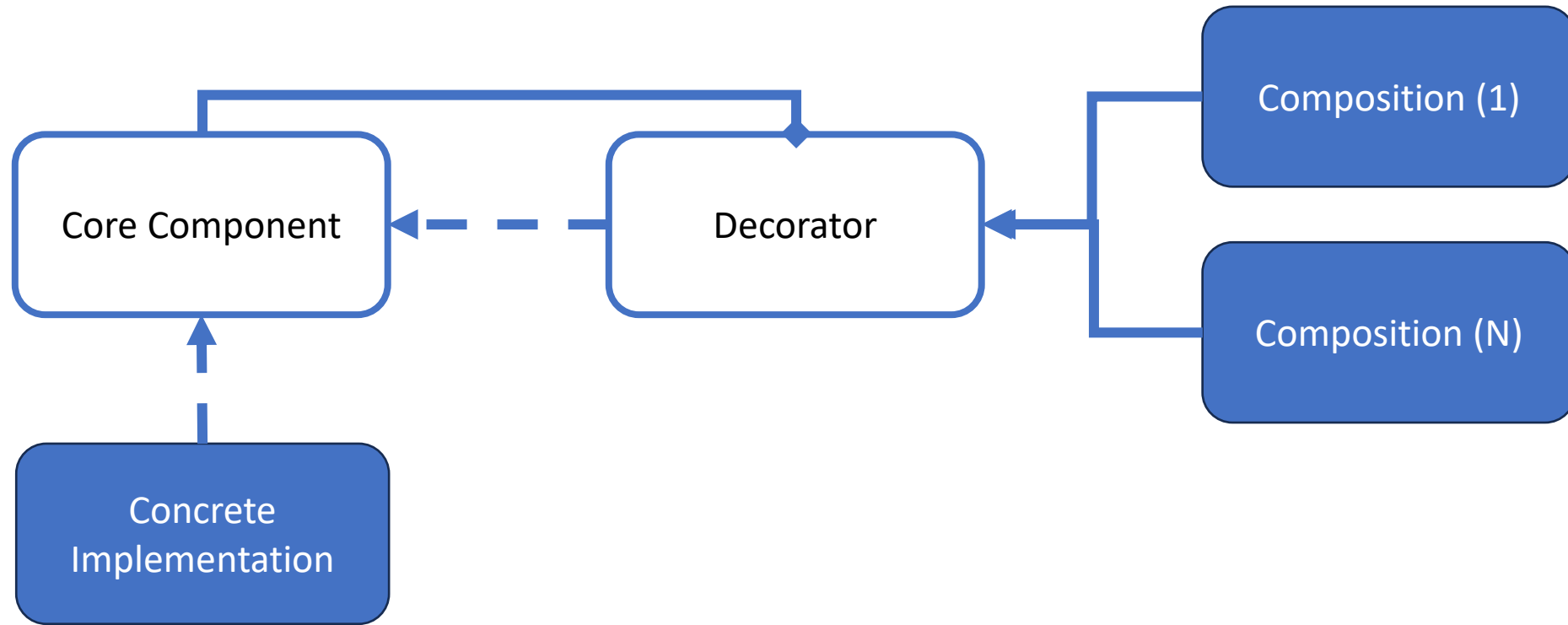
Essential Software Design Patterns for Optimal Development

# Decorator Pattern

Structural Design Patterns



# Decorator Pattern



# Core Component

```
public interface ICar  
{  
    void Assemble();  
}
```

# Concrete Implementation

```
public interface ICar  
{  
    void Assemble();  
}
```

```
public class BasicCar : ICar  
{  
    public void Assemble() => Console.WriteLine("Basic Car is assembled.");  
}
```

# Decorator

```
public interface ICar
{
    void Assemble();
}
```

```
public class CarDecorator(ICar car) : ICar
{
    protected ICar _car = car;

    public virtual void Assemble() => _car.Assemble();
}
```

# Compositions

```
public class SportsCar(ICar car) : CarDecorator(car)
{
    public override void Assemble()
    {
        base.Assemble();
        Console.WriteLine("Adding features of Sports Car.");
    }
}
```

```
public class LuxuryCar(ICar car) : CarDecorator(car)
{
    public override void Assemble()
    {
        base.Assemble();
        Console.WriteLine("Adding features of Luxury Car.");
    }
}
```

# Client

```
// Creating a basic car
ICar basicCar = new BasicCar();
basicCar.Assemble();

// Decorating basic car with sports car features
ICar sportsCar = new SportsCar(basicCar);
sportsCar.Assemble();

// Decorating basic car with luxury car features
ICar luxuryCar = new LuxuryCar(basicCar);
luxuryCar.Assemble();

// Decorating basic car with both sports and luxury car features
ICar sportsLuxuryCar = new LuxuryCar(new SportsCar(basicCar));
```

# Client (More Performant)

```
// Creating a basic car
BasicCar basicCar = new();
basicCar.Assemble();
```

```
// Decorating basic car with sports car features
SportsCar sportsCar = new(basicCar);
sportsCar.Assemble();
```

```
// Decorating basic car with luxury car features
LuxuryCar luxuryCar = new(basicCar);
luxuryCar.Assemble();
```

```
// Decorating basic car with sports and luxury car features
LuxuryCar sportsLuxuryCar = new(new SportsCar(basicCar));
sportsLuxuryCar.Assemble();
```



# Decorator Pattern in .NET

Stream Classes

ASP.NET Core  
Middleware

Attribute Decorators

Logging Frameworks

Encryption and  
Compression  
Libraries

# Decorator Pattern

## Benefits

- Enhanced Flexibility
- Open-Closed Principle
- Single Responsibility Principle
- Modular and Reusable Code
- Fine-Grained Control
- Transparent to Clients

## Drawbacks

- Complexity
- Potential Performance Overhead
- Maintainability
- Potential of Object Proliferation
- Ordering Dependencies

# Decorator Pattern

## Good Times to Use

- Adding Functionality Dynamically
- Extending Functionality without Subclassing
- Open-Closed Principle Compliance
- Combining Multiple Responsibilities
- Dynamic Configuration or Feature Selection

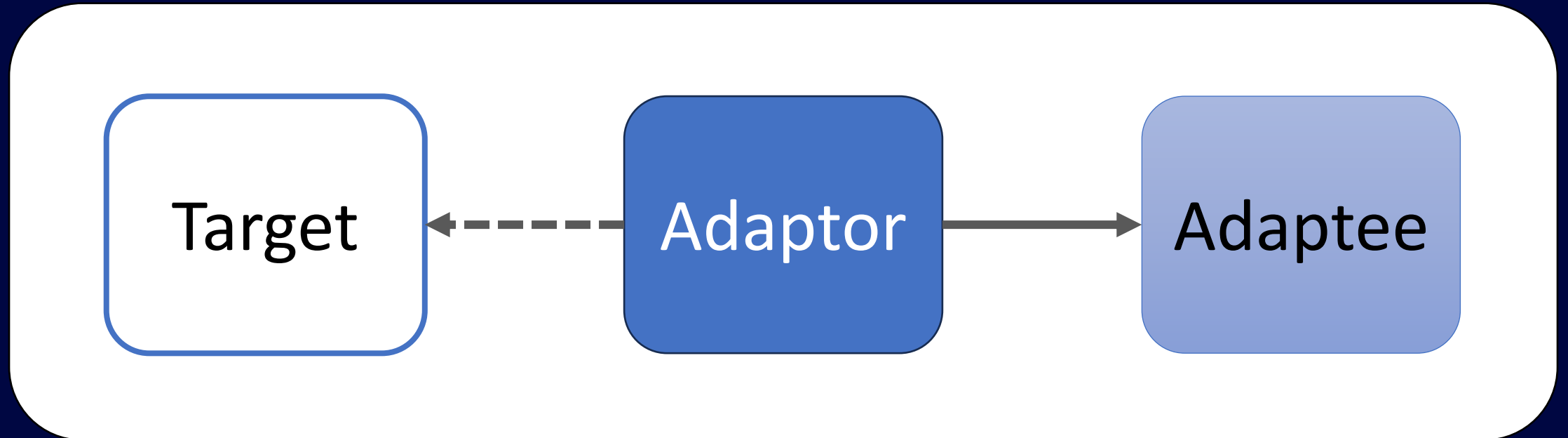
## Bad Times to Use

- Simple Functionality Addition
- Deeply Nested Decorated Chains
- Performance-Critical Systems
- Tightly Coupled Decorators
- Complex Ordering Dependencies

# Adapter Pattern

Structural Design Patterns

# Adapter Pattern Key Concepts



# Adapter Pattern Types

**Class Adapter**

**Object Adapter**

# Target Interface

```
public interface IMediaPlayer
{
    void Play(String audioType, String fileName);
}
```

# Adaptee

```
public class LegacyAudioPlayer
{
    public void PlayMp3(String fileName)
        => Console.WriteLine("Playing mp3 file. Name: " + fileName);

    public void PlayWAV(String fileName)
        => Console.WriteLine("Playing WAV file. Name: " + fileName);
}
```



# Adapter

```
public class MediaAdapter(LegacyAudioPlayer legacyAudioPlayer) : IMediaPlayer
{
    private readonly LegacyAudioPlayer _legacyAudioPlayer = legacyAudioPlayer;

    public void Play(string audioType, string fileName)
    {
        if (audioType.Equals("mp3", StringComparison.OrdinalIgnoreCase))
            _legacyAudioPlayer.PlayMp3(fileName);
        else if (audioType.Equals("wav", StringComparison.OrdinalIgnoreCase))
            _legacyAudioPlayer.PlayWAV(fileName);
        else
            Console.WriteLine("Invalid media. " + audioType + " format not supported");
    }
}
```

# Client

```
IMediaPlayer player = new MediaAdapter(new LegacyAudioPlayer());  
player.Play("mp3", "Thunderstuck.mp3");  
player.Play("wav", "Back-In-Black.wav");  
player.Play("flac", "Hells-Highway.flac"); // Unsupported format
```

# Adapter Pattern in .NET

- Stream Adapters

```
using (FileStream fs = new FileStream("example.txt", FileMode.Open))
{
    using (StreamReader sr = new StreamReader(fs))
    {
        string line = sr.ReadLine();
        Console.WriteLine(line);
    }
}
```

# Adapter Pattern in .NET

- Stream Adapters
- ICollection and IList Interfaces

```
public void PrintCollection(ICollection collection)
{
    foreach (var item in collection)
    {
        Console.WriteLine(item);
    }
}
```

```
ArrayList arrayList = new ArrayList { 1, 2, 3 };
PrintCollection(arrayList);
```

```
List<int> list = new List<int> { 4, 5, 6 };
PrintCollection(list);
```

# Adapter Pattern

## Benefits

- Interface Compatibility
- Reusability
- Flexibility
- Decoupling
- Ease of Refactoring

## Drawbacks

- Increased Complexity
- Performance Overhead
- Maintenance Burden
- Tight Coupling to the Adapter

# Adapter Pattern

## Good Times to Use

- Integrating Legacy Systems
- Using Third-Party Libraries
- Facilitating API Changes
- Bridging Different Technologies
- Abstracting Vendor-Specific Implementations

## Times to Avoid

- Overcomplicating Simple Interfaces
- Adapters for Temporary Fixes
- Avoiding Proper Refactoring
- Performance-Critical Applications

# Behavioral Design Patterns

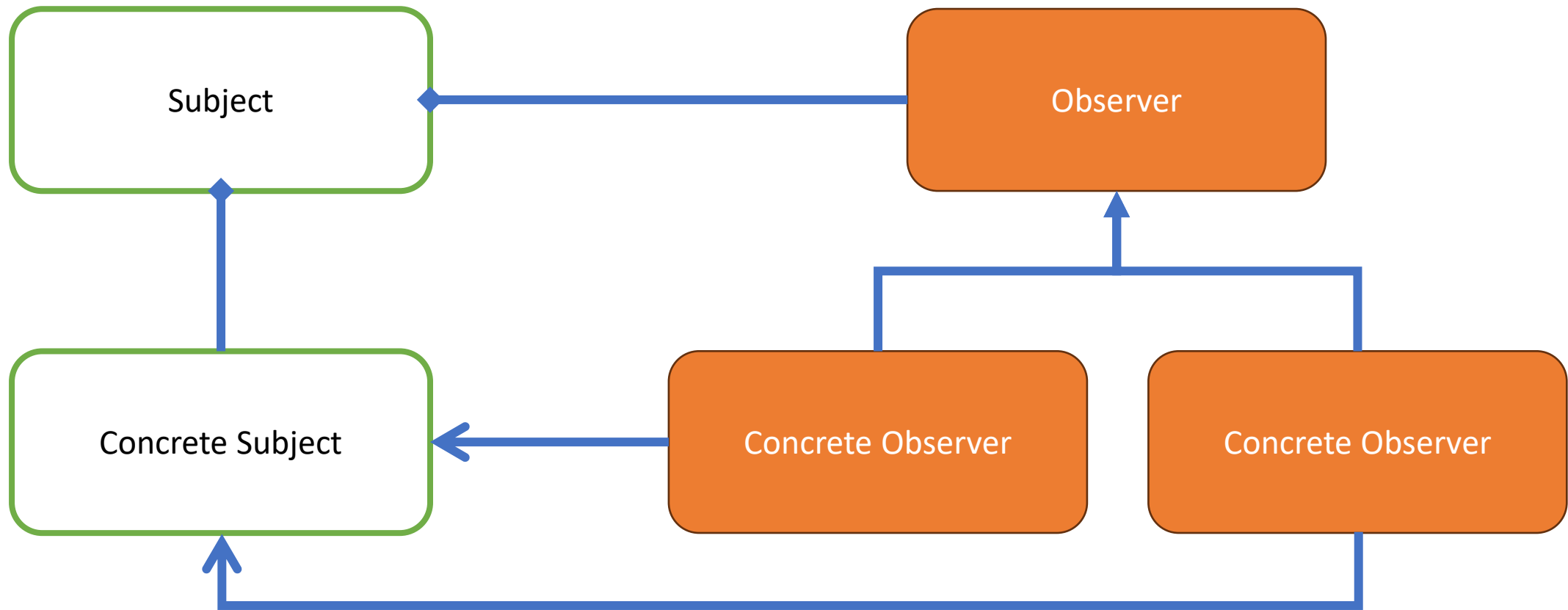
Essential Software Design Patterns for Optimal Development

# Observer Pattern

Behavioral Design Patterns



# Observer Pattern



# Subject

```
public interface ISubject
{
    void Attach(IObserver observer);
    void Detach(IObserver observer);
    void Notify();
}
```

# Observer

```
public interface IObserver
{
    void Update(ISubject subject);
}
```

# Concrete Subject

```
public class ConcreteSubject : ISubject
{
    public int State { get; set; } = 0;

    private readonly List<IObserver> _observers = [];

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

    public void Detach(IObserver observer)
    {
        _observers.Remove(observer);
    }

    public void Notify()
    {
        foreach (var observer in _observers)
            observer.Update(this);
    }
}
```

# Concrete Observers

```
public class ConcreteObserverA : IObserver
{
    public void Update(ISubject subject)
    {
        if (subject is ConcreteSubject { State: < 3 })
        {
            Console.WriteLine("ConcreteObserverA: Reacted to the event.");
        }
    }
}
```

```
public class ConcreteObserverB : IObserver
{
    public void Update(ISubject subject)
    {
        if (subject is ConcreteSubject { State: 0 or >= 2 })
        {
            Console.WriteLine("ConcreteObserverB: Reacted to the event.");
        }
    }
}
```

# Client

```
var subject = new ConcreteSubject();  
var observerA = new ConcreteObserverA();  
subject.Attach(observerA);
```

```
var observerB = new ConcreteObserverB();  
subject.Attach(observerB);
```

```
subject.State = 0;  
subject.Notify();
```

```
subject.State = 2;  
subject.Notify();
```

```
subject.Detach(observerB);
```

```
subject.State = 3;  
subject.Notify();
```

# Observer Pattern in .NET

## Event Handling in WinForms

```
button1.Click += Button_Click;
```

# Observer Pattern in .NET

## INotifyPropertyChanged

```
public class PersonViewModel : INotifyPropertyChanged
{
    private string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (name != value)
            {
                name = value;
                OnPropertyChanged(nameof(Name));
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```



# Observer Pattern in .NET

## Observables in Reactive Extensions

```
IObservable<int> observable = Observable.Range(1, 5);  
IDisposable subscription = observable.Subscribe(  
    value => Console.WriteLine($"Received: {value}"),  
    error => Console.WriteLine($"Error: {error}"),  
    () => Console.WriteLine("Completed")  
);
```

# Observer Pattern in .NET

Event Handling in  
WinForms

INotifyPropertyChanged

Observables in  
Reactive Extensions

ASP.NET Core  
Middleware

# Observer Pattern

## Benefits

- Loose Coupling
- Modular Design
- Event-Driven Architecture
- Support for Broadcast Communication
- Encapsulation
- Flexibility

## Drawbacks

- Potential Performance Overhead
- Memory Management Concerns
- Complexity
- Potential for Circular Dependencies
- Ordering of Notifications
- Difficulty in Debugging

# Observer Pattern

## Good Times to Use

- User Interface Updates
- Event Handling
- Publish-Subscribe Systems
- MVC and MVVM Architectures
- Monitoring Systems
- Distributed Systems
- Logging and Auditing

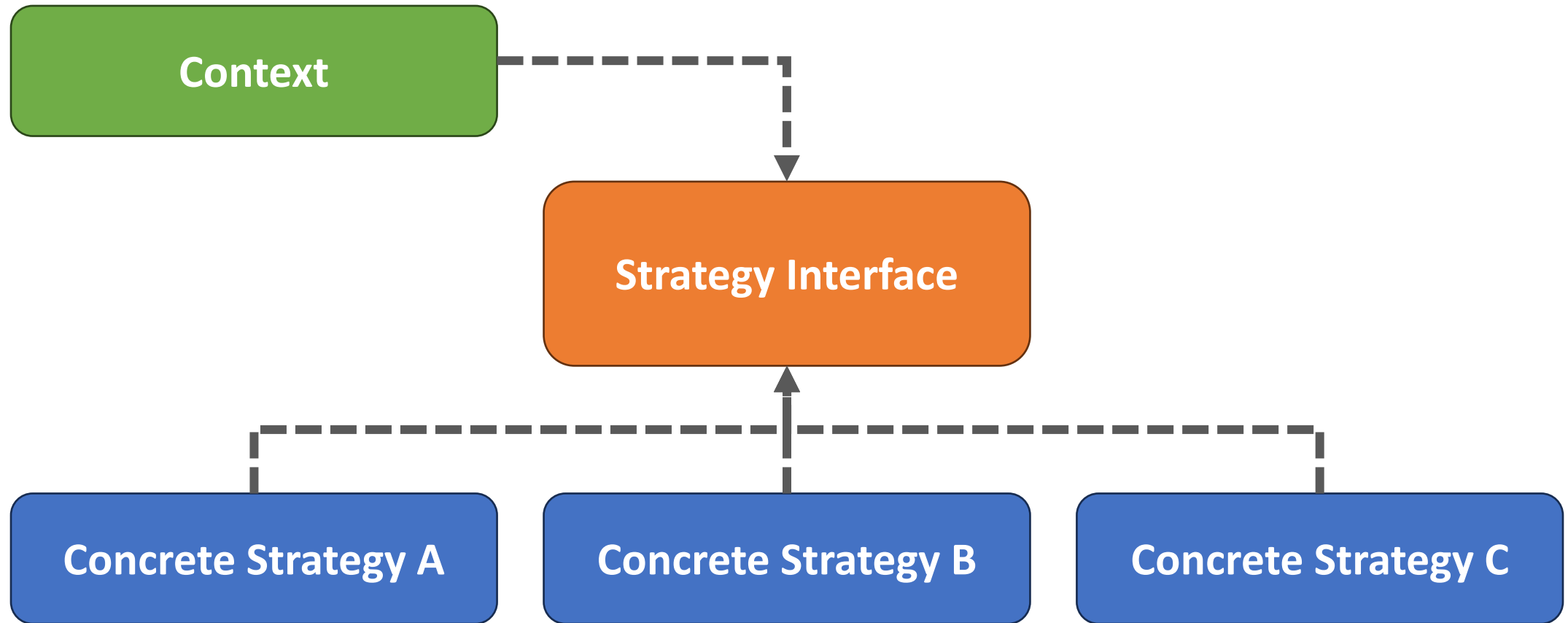
## Bad Times to Use

- Simple Event Handling
- Tight Coupling Between Subject and Observers
- Performance-Critical Systems
- Static Configuration
- Complex Dependency Management
- Asynchronous Communication
- Inflexible Configuration

# Strategy Pattern

Behavioral Design Patterns

# Strategy Pattern



# Strategy Interface

```
public interface IDiscountStrategy
{
    decimal ApplyDiscount(decimal price);
}
```

# Concrete Strategies

```
public class NoDiscount : IDiscountStrategy
{
    public decimal ApplyDiscount(decimal price) => price; // No discount applied
}

public class SeasonalDiscount : IDiscountStrategy
{
    public decimal ApplyDiscount(decimal price) => price * 0.9m; // 10% discount
}

public class LoyaltyDiscount : IDiscountStrategy
{
    public decimal ApplyDiscount(decimal price) => price * 0.85m; // 15% discount
}
```



# Context

```
public class PriceCalculator(IDiscountStrategy discountStrategy)
{
    private IDiscountStrategy _discountStrategy = discountStrategy;

    public void SetDiscountStrategy(IDiscountStrategy discountStrategy)
        => _discountStrategy = discountStrategy;

    public decimal CalculatePrice(decimal price)
        => _discountStrategy.ApplyDiscount(price);
}
```

# Implementation

```
decimal originalPrice = 100.0m;

PriceCalculator calculator = new PriceCalculator(new NoDiscount());
decimal noDiscountPrice = calculator.CalculatePrice(originalPrice);
Console.WriteLine($"Original Price: {originalPrice}, Price with No Discount: {noDiscountPrice}");

calculator.SetDiscountStrategy(new SeasonalDiscount());
decimal seasonalDiscountPrice = calculator.CalculatePrice(originalPrice);
Console.WriteLine($"Original Price: {originalPrice}, Price with Seasonal Discount: {seasonalDiscountPrice}");

calculator.SetDiscountStrategy(new LoyaltyDiscount());
decimal loyaltyDiscountPrice = calculator.CalculatePrice(originalPrice);
Console.WriteLine($"Original Price: {originalPrice}, Price with Loyalty Discount: {loyaltyDiscountPrice}");
```

# Strategy Pattern in .NET

- Dependency Injection and Service Lifetimes

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IMyService, MyTransientService>(); // Transient strategy
    services.AddScoped<IMyService, MyScopedService>(); // Scoped strategy
    services.AddSingleton<IMyService, MySingletonService>(); // Singleton strategy
}
```

# Strategy Pattern in .NET

- Dependency Injection and Service Lifetimes
- Logging

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddLogging(builder =>
    {
        builder.AddConsole(); // Console logging strategy
        builder.AddDebug(); // Debug logging strategy
        builder.AddEventLog(); // Event Log logging strategy
    });
}
```

# Strategy Pattern in .NET

- Dependency Injection and Service Lifetimes
- Logging
- Caching

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDistributedMemoryCache(); // In-memory caching strategy
    services.AddStackExchangeRedisCache(options =>
    {
        options.Configuration = "localhost:6379";
        options.InstanceName = "SampleInstance";
    }); // Redis caching strategy
}
```

# Strategy Pattern in .NET

- Dependency Injection and Service Lifetimes
- Logging
- Caching

# Strategy Pattern

## Benefits

- Flexibility and Reusability
- Maintainability
- Ease of Extension
- Simplified Testing and Debugging
- Runtime Flexibility

## Drawbacks

- Class Proliferation
- Code Complexity
- Context-Strategy Coupling
- Client Awareness
- Performance Overhead

# Strategy Pattern

## Good Times to Use

- Sorting Algorithms
- Payment Processing Systems
- File Compression
- Authentication Mechanisms
- Log Formatting

## Times to Avoid

- Simple or Static Behavior
- Need for Simplicity
- Single Use Case
- Frequent Changes in Strategy Logic



# Careful Consideration Needed

Essential Software Design Patterns for Optimal Developer

# Pattern Considerations

- Should be applied judiciously

# Pattern Considerations

- Should be applied judiciously
- Appropriateness influenced by nature of software being developed

# Pattern Considerations

- Should be applied judiciously
- Appropriateness influenced by nature of software being developed
- Essential to carefully evaluate trade-offs

# Think Before Using



# Other Categories of Design Patterns

Essential Software Development Patterns for Optimal Development

# Design Pattern Categories

**Creational**

**Structural**

**Behavioral**

# Design Pattern Categories

Creational

Structural

Behavioral

Concurrency

- Thread Pool
- Producer-Consumer
- Reader-Writers



# Types of Design Patterns

**Creational**

**Structural**

**Behavioral**

**Concurrency**

**Architectural**

- Event-Driven Architecture
- Layered Architecture
- Microservices

- Model-View-Controller (MVC)
- Service-Oriented Architecture

# Types of Design Patterns

**Creational**

**Structural**

**Behavioral**

**Concurrency**

**Architectural**

**Cloud**

- Simple Web Service
- Robust API
- Decoupled Messaging
- Publish/Subscribe

- Aggregation
- Strangler
- Queue-Based Load Leveling
- Pipes and Filters

- Fan-Out/Fan-In
- Materialized Views

# (Serverless) Cloud Design Patterns



# Summary

Essential Software Design Patterns

# Summary

- Overview of Design Patterns

# Summary

- Overview of Design Patterns
- Builder Pattern (Creational)

# Summary

- Overview of Design Patterns
- Builder Pattern (Creational)
- Factory Pattern (Creational)

# Summary

- Overview of Design Patterns
- Builder Pattern (Creational)
- Factory Pattern (Creational)
- Decorator Pattern (Structural)



# Summary

- Overview of Design Patterns
- Builder Pattern (Creational)
- Factory Pattern (Creational)
- Decorator Pattern (Structural)
- Adapter Pattern (Structural)
- Observer Pattern (Behavioral)

# Summary

- Overview of Design Patterns
- Builder Pattern (Creational)
- Factory Pattern (Creational)
- Decorator Pattern (Structural)
- Adapter Pattern (Structural)
- Observer Pattern (Behavioral)
- Strategy Pattern (Behavioral)
- Considerations

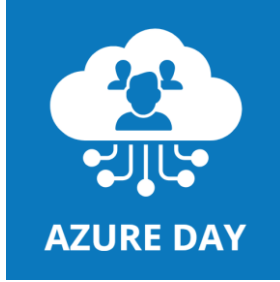
# Summary

- Overview of Design Patterns
- Builder Pattern (Creational)
- Factory Pattern (Creational)
- Decorator Pattern (Structural)
- Adapter Pattern (Structural)
- Observer Pattern (Behavioral)
- Strategy Pattern (Behavioral)
- Considerations
- Other Categories of Design Patterns

# Join Me



LOUISVILLE  
AZURE MEETUP



AZURE DAY

meetup



Nebraska.Code()

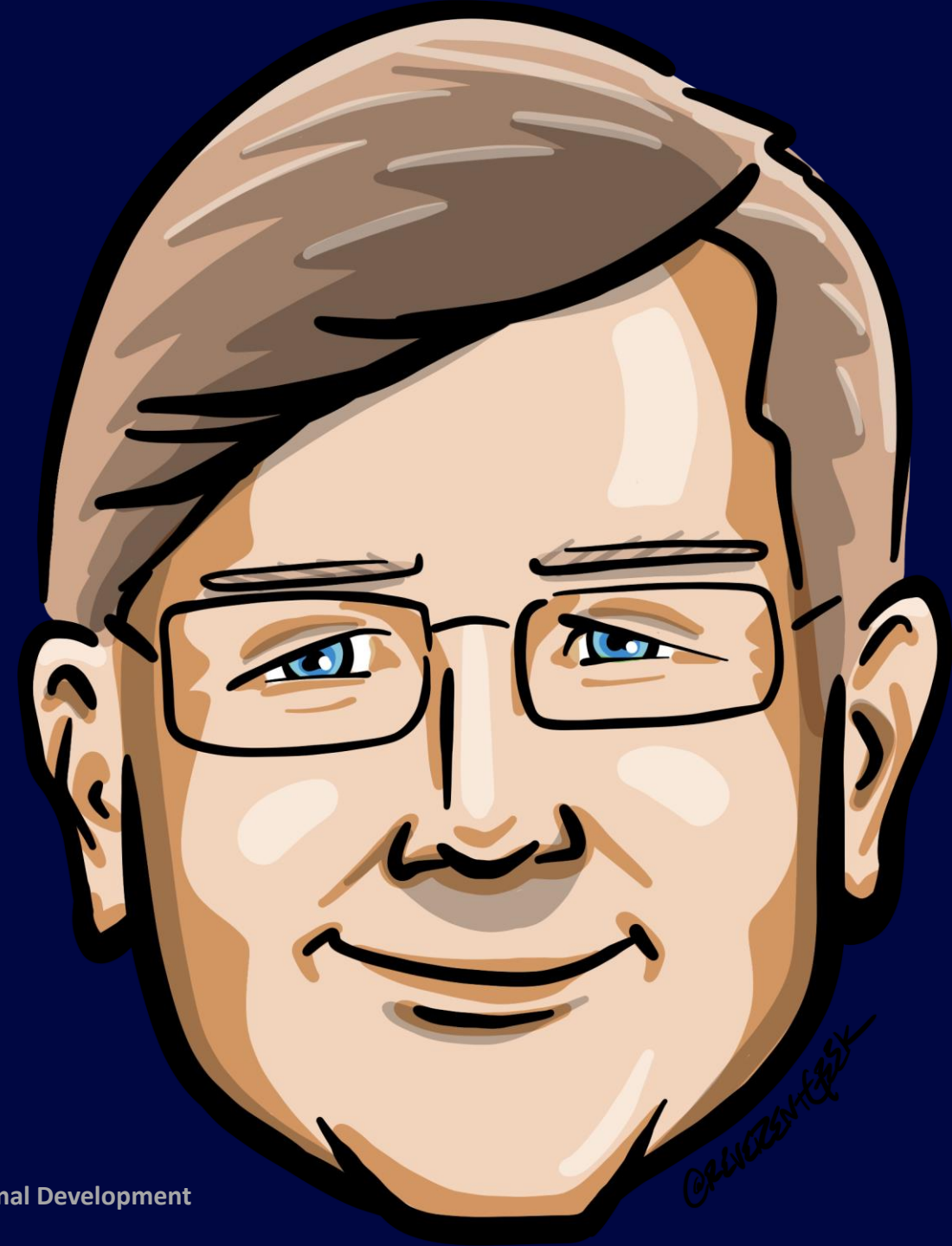


dev up



LOUISVILLE  
.NET MEETUP

API  
CONFERENCE



@m00z0r

# Thank You

✉ chadgreen@chadgreen.com

💬 TaleLearnCode

🌐 ChadGreen.com

🐦 ChadGreen & TaleLearnCode

🌐 ChadwickEGreen

