CS6004NT
Application Development

WEEK - 02

Programming Principles

SOLID Principles

SOLID principles are a set of **five design principles** that are used to make software more **maintainable**, **flexible**, and **scalable**. They were introduced by **Robert C. Martin (Uncle Bob)** in his paper "Design Principles and Design Patterns".

The five principles are:

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

1. Single Responsibility Principle (SRP)

- The SRP states that **a class should have only one reason to change**.
- In other words, **a class should have only one responsibility or job**.
- This makes the class more maintainable and easier to understand.
- Example:
  - A class that handles user authentication should only be responsible for authentication and not also for handling user data or sending emails.

1. Single Responsibility Principle (SRP)

Example of SRP violation: A class that handles both user authentication and sending email

```
1. public class UserService
2. {
3.     public bool AuthenticateUser(string username, string password)
4.     {
5.         // Code to authenticate user
6.         return true;
7.     }

8.     public void SendWelcomeEmail(User user)
9.     {
10.        // Code to send a welcome email to a user
11.    }
12.}
```

1. Single Responsibility Principle (SRP)

Example of SRP compliance: Separate classes for user authentication and sending email

```
1. public class UserService
2. {
3.     public bool AuthenticateUser(string username, string password)
4.     {
5.         // Code to authenticate user
6.         return true;
7.     }
8. }
```

```
1. public class EmailService
2. {
3.     public void SendWelcomeEmail(User user)
4.     {
5.         // Code to send a welcome email to a user
6.     }
7. }
```

2. Open/Closed Principle (OCP)

- The OCP states that **a class should be open for extension but closed for modification**.
- In other words, **you should be able to add new functionality to a class without changing its existing code**.
- This makes the class more flexible and easier to maintain.
- Example:
  - A payment processing module is following OCP if the class can be customized to handle new payment methods without modifying the existing code.

2. Open/Closed Principle (OCP)

Example of OCP violation: A payment processing class that handles only credit card payments

```
1. public class PaymentProcessor
2. {
3.     public void ProcessCreditCardPayment(CreditCardPayment payment)
4.     {
5.         // Code to process credit card payment
6.     }
7. }
```

2. Open/Closed Principle (OCP)

Example of OCP compliance: A payment processing class that uses a payment gateway interface for processing payments

```
1. public class PaymentProcessor
2. {
3.     private IPaymentGateway _paymentGateway;
4.     public PaymentProcessor(IPaymentGateway paymentGateway)
5.     {
6.         _paymentGateway = paymentGateway;
7.     }

8.     public void ProcessPayment(Payment payment)
9.     {
10.        // Code to process payment using IPaymentGateway
11.        _paymentGateway.ProcessPayment(payment);
12.    }
13.}
```

```
1. public interface IPaymentGateway
2. {
3.     void ProcessPayment(Payment payment);
4. }
```

```
1. public class CreditCardPaymentGateway : IPaymentGateway
2. {
3.     public void ProcessPayment(Payment payment)
4.     {
5.         // Code to process credit card payment
6.     }
7. }
```

```
1. public class PayPalPaymentGateway : IPaymentGateway
2. {
3.     public void ProcessPayment(Payment payment)
4.     {
5.         // Code to process PayPal payment
6.     }
7. }
```

3. Liskov Substitution Principle (LSP)

- The LSP states that **objects of a superclass should be able to be replaced with objects of a subclass without affecting the correctness of the program**.
- In other words, **a subclass should be able to replace its superclass without any unexpected behavior.**
- This makes the code more scalable and easier to understand.
- Example:
  - If you have a superclass called Shape, you should be able to replace an object of type Shape with an object of a subclass like Rectangle or Square.

3. Liskov Substitution Principle (LSP)

Example of LSP violation: A subclass that modifies the behavior of the parent class

```
1.  public class Rectangle
2.  {
3.      public virtual int Width { get; set; }
4.      public virtual int Height { get; set; }

5.      public int Area => Width * Height;
6.  }
```

```
1.  public class Square : Rectangle
2.  {
3.      public override int Width
4.      {
5.          get => base.Width;
6.          set
7.          {
8.              base.Width = base.Height = value;
9.          }
10.     }
11.     public override int Height
12.     {
13.         get => base.Height;
14.         set
15.         {
16.             base.Width = base.Height = value;
17.         }
18.     }
19. }
```

```
1. Rectangle rect = new Rectangle();
2. rect.Width = 5;
3. rect.Height = 4;
4. Console.WriteLine(rect.Area); // 20 as expected
5. rect = new Square();
6. rect.Width = 5;
7. rect.Height = 4;
8. Console.WriteLine(rect.Area); // 16 expected 20
```

3. Liskov Substitution Principle (LSP)

Example of LSP compliance: A class hierarchy that follows the LSP

```
1.  public abstract class Shape
2.  {
3.      public abstract double Area();
4.  }
```

```
1.  public class Rectangle : Shape
2.  {
3.      private double _width;
4.      private double _height;
5.      public Rectangle(double Width, double Height)
6.      {
7.        _width = Width;
8.        _height = Height;
9.      }
10.     public override double Area() => _width * _height;
11. }
```

```
1.  public class Square : Shape
2.  {
3.      private double _sideLength;
4.      public Square(double SideLength) {
5.        _sideLength = SideLength;
6.      }
7.      public override double Area() => _sideLength * _sideLength;
8.  }
```

```
1. Shape rect = new Rectangle(Width:5, Height:4);
2. Console.WriteLine(rect.Area()); // 20
3. Shape sq = new Square(SideLength: 4);
4. Console.WriteLine(sq.Area()); // 16
```

4. Interface Segregation Principle (ISP)

- The ISP states that **clients should not be forced to depend on methods they do not use**.
- In other words, **a class should have separate interfaces for each of its responsibilities** so that clients only need to depend on the interfaces they use.
- This makes the code more maintainable and easier to understand.
- Example:
  - A database access interface, is following ISP by having only the required methods.

4. Interface Segregation Principle (ISP)

Example of ISP violation: An interface that contains methods that are not required by its clients

```
1. public interface IDataAccess
2. {
3.     void Save(object data);
4.     void Update(object data);
5.     void Delete(object data);
6.     void GetById(int id);
7.     void GetAll();
8. }
```

Example of ISP compliance: Separate interfaces for different types of data access

```
1. public interface ISaveDataAccess
2. {
3.     void Save(object data);
4. }
```

5. Dependency Inversion Principle (DIP)

- The DIP states that **high-level modules should not depend on low-level modules. Both should depend on abstractions.**
- In other words, **classes should depend on abstractions (interfaces or abstract classes) rather than concrete implementations**.
- This makes the code more flexible and easier to test.
- Example:
  - Instead of a class depending directly on a database connection, it should depend on an interface for a database connection so that it can be easily replaced with a mock implementation for testing.

5. Dependency Inversion Principle (DIP)

Example of DIP violation: A class that depends on a concrete implementation of a dependency

```
1. public class BusinessLogic
2. {
3.     private DataAccess _dataAccess = new DataAccess();

4.     public void DoSomething()
5.     {
6.         // Code that uses _dataAccess to perform some operation
7.     }
8. }
```

```
1. public class DataAccess
2. {
3.     // Code to access data
4. }
```

5. Dependency Inversion Principle (DIP)

Example of DIP compliance: A class that depends on an abstraction of its dependency

```
1.  public class BusinessLogic
2.  {
3.      private IDataAccess _dataAccess;

4.      public BusinessLogic(IDataAccess dataAccess)
5.      {
6.          _dataAccess = dataAccess;
7.      }

8.      public void DoSomething()
9.      {
10.         // Code that uses _dataAccess to perform some operation
11.     }
12. }
```

```
1.  public interface IDataAccess
2.  {
3.      // Methods to access data
4.  }
```

```
1. public class SqlDataAccess : IDataAccess
2. {
3.     // Code to access data using SQL
4. }
```

```
1. public class FileDataAccess : IDataAccess
2. {
3.     // Code to access data using File
4. }
```

Other Programming Principles

In addition to SOLID principles, there are many other programming principles that every programmer should know.

1. Don't Repeat Yourself (DRY) Principle
2. Keep It Simple, Stupid (KISS) Principle
3. You Aren't Gonna Need It (YAGNI) Principle
4. Don't Reinvent the Wheel (DRW) Principle
5. Separation of Concerns (SoC) Principle

1. Don't Repeat Yourself (DRY) Principle

- The DRY principle states that **you should not repeat yourself in code**.
- In other words, **you should not have duplicate code in different parts of your program**.
- This makes the code more maintainable and reduces the chance of introducing bugs.
- Example:
  - Instead of copying and pasting the same code in multiple places, you should extract it into a reusable function or class.

1. Don't Repeat Yourself (DRY) Principle

Example of DRY violation: It repeats the same logic for printing a message in multiple places

```
1.  public void Main()
2.  {
3.      Console.WriteLine("Hello, World!");
4.      Console.WriteLine("Welcome to C#!");
5.  }
```

Example of DRY compliance: Using a single method to perform a common operation instead of duplicating code

```
1.  public void PrintMessage(string message)
2.  {
3.      Console.WriteLine(message);
4.  }

5.  public void Main()
6.  {
7.      PrintMessage("Hello, World!");
8.      PrintMessage("Welcome to C#!");
9.  }
```

2. Keep It Simple, Stupid (KISS) Principle

- The KISS principle states that **you should keep your code simple and easy to understand**.
- In other words, **you should avoid overcomplicating your code with unnecessary features or abstractions**.
- This makes the code more maintainable and easier to debug.
- Example:
  - Instead of creating a complex system to handle user authentication, you could use a simple username and password system.

2. Keep It Simple, Stupid (KISS) Principle

Example of KISS violation: Perform some complex calculations to add the numbers

```
1. public int AddNumbers(int x, int y)
2. {
3.     var result = x;
4.     for (int i = 1; i <= y; i++)
5.     {
6.       result++;
7.     }
8.     return result;
9. }
```

Example of KISS compliance: Writing simple code to perform a common operation

```
1. public int AddNumbers(int x, int y)
2. {
3.     return x + y;
4. }
```

3. You Aren't Gonna Need It (YAGNI) Principle

- The YAGNI principle states that **you should not add features to your code unless you need them**.
- In other words, **you should not try to anticipate every possible use case for your code**.
- This makes the code more efficient and easier to maintain.
- Example:
  - Instead of adding every possible feature to a class, you should only add the features that are needed for its current use case.

3. You Aren't Gonna Need It (YAGNI) Principle

Example of YAGNI violation: Has unnecessary code or functionality

```
1. public class Product
2. {
3.     public string Name { get; set; }
4.     public decimal Price { get; set; }

5.     public decimal GetSalePrice(decimal discount) => Price - (Price * discount);
6. }
```

Example of YAGNI compliance: Only includes necessary code

```
1. public class Product
2. {
3.     public string Name { get; set; }
4.     public decimal Price { get; set; }
5. }
```

4. Don't Reinvent the Wheel (DRW) Principle

- The DRW principle states that **you should not try to create something that already exists** in the programming world.
- In other words, **you should use existing libraries, frameworks, and tools whenever possible**.
- This makes the code more efficient and reduces the chance of introducing bugs.
- Example:
  - Instead of creating your own encryption algorithm, you could use a well-established algorithm like AES.

4. Don't Reinvent the Wheel (DRW) Principle

Example of DRW violation: Writing a custom sorting algorithm

```
1. public class CustomSort
2. {
3.     public static int[] Sort(int[] arr)
4.     {
5.         // implementation of a custom sorting algorithm
6.         return arr;
7.     }
8. }

9. // usage of the custom sorting algorithm
10.int[] arr = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 };
11.int[] sortedArr = CustomSort.Sort(arr);
```

Example of DRW compliance: Using built-in sorting algorithm provided by the language

```
1. int[] arr = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 };
2. Array.Sort(arr);
```

5. Separation of Concerns (SoC) Principle

- The SoC principle states that **different parts of a program should be designed to handle different concerns**.
- In other words, **each part of the program should focus on doing one thing and doing it well**.
- This makes the code more modular, easier to understand, and easier to maintain.
- Example:
  - Instead of having all the code related to user authentication in one class, you could split it into separate classes for handling user input, user authentication, and user data storage.

5. Separation of Concerns (SoC) Principle

Example of SoC violation: Combining presentation and data access logic in a single class

```
1. public class UserController
2. {
3.     public ActionResult Index()
4.     {
5.     // Code to retrieve users from a database
6.     var users = Database.GetAllUsers();
7.     return View(users);
8.     }
9. }
```

```
1. public static class Database
2. {
3.     public static List<User> GetAllUsers()
4.     {
5.     // Code to retrieve users from a database
6.     }
7. }
```

5. Separation of Concerns (SoC) Principle

Example of SoC compliance: Separating concerns into different layers or components

```
1.  // Presentation layer
2.  public class UserController
3.  {
4.      private UserService _userService = new UserService();

5.      public ActionResult Index()
6.      {
7.          var users = _userService.GetAllUsers();
8.          return View(users);
9.      }
10. }
```

```
1.  // Business logic layer
2.  public class UserService
3.  {
4.      private UserRepository _userRepository = new UserRepository();

5.      public List<User> GetAllUsers()
6.      {
7.          return _userRepository.GetAllUsers();
8.      }
9.  }
```

```
1. // Data access layer
2. public class UserRepository
3. {
4.     public List<User> GetAllUsers()
5.     {
6.         // Code to retrieve users from a
   database
7.     }
8. }
```

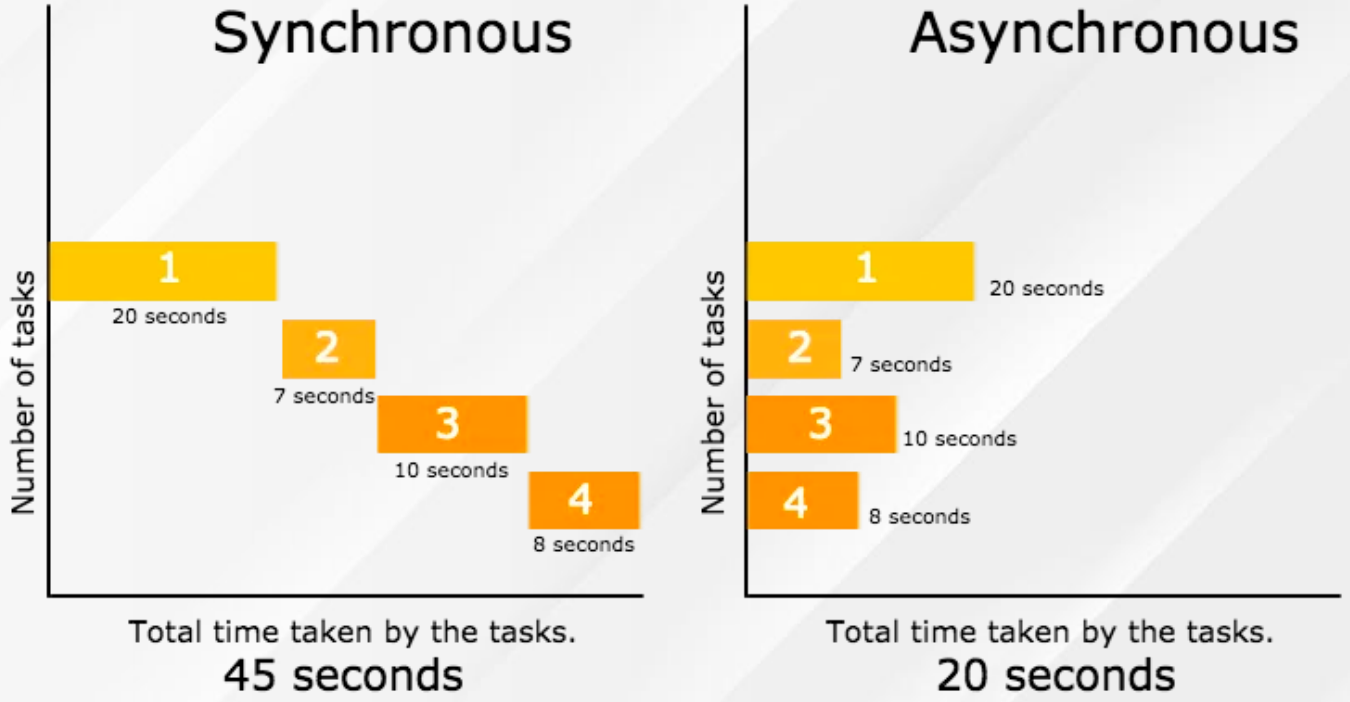
C# Advanced Topics

Async, Await, and Task

Async is a feature in C# that allows you to **run multiple operations simultaneously**.

Await is a keyword that can be used inside an async method to **wait for the completion of another task**.

Task is a class in C# that represents **an asynchronous operation.**

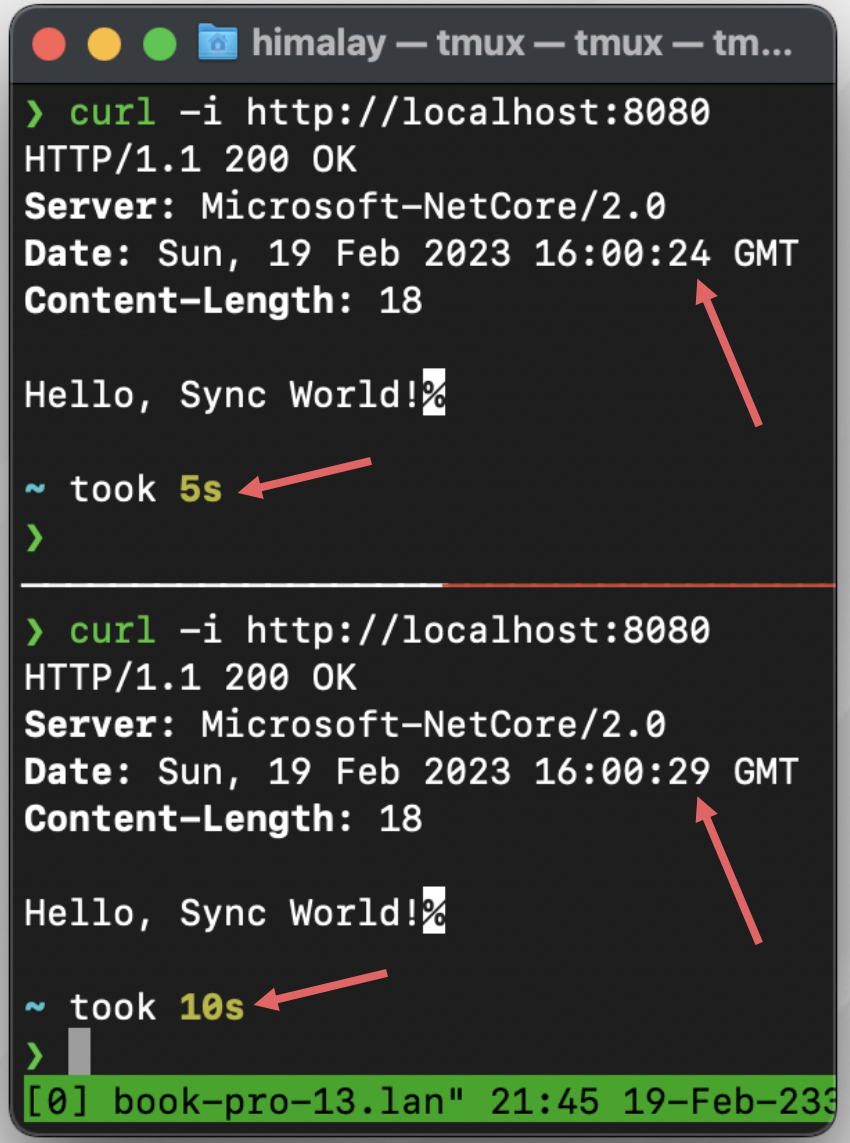


Example

```
14. void SendResponse(HttpListenerContext context, string message)
15. {
16.     // Write a response to the client
17.     byte[] buffer = System.Text.Encoding.UTF8.GetBytes(message);
18.     context.Response.ContentLength64 = buffer.Length;
19.     context.Response.OutputStream.Write(buffer, 0, buffer.Length);
20.     context.Response.OutputStream.Close();
21. }

22. void HandleGetRequest(HttpListenerContext context)
23. {
24.     Thread.Sleep(5000); // Simulate a long-running operation
25.     SendResponse(context, "Hello, Sync World!");
26. }

27. async Task HandleGetRequestAsync(HttpListenerContext context)
28. {
29.     await Task.Delay(5000); // Simulate a long-running operation asynchronously
30.     SendResponse(context, "Hello, Async World!");
31. }
```

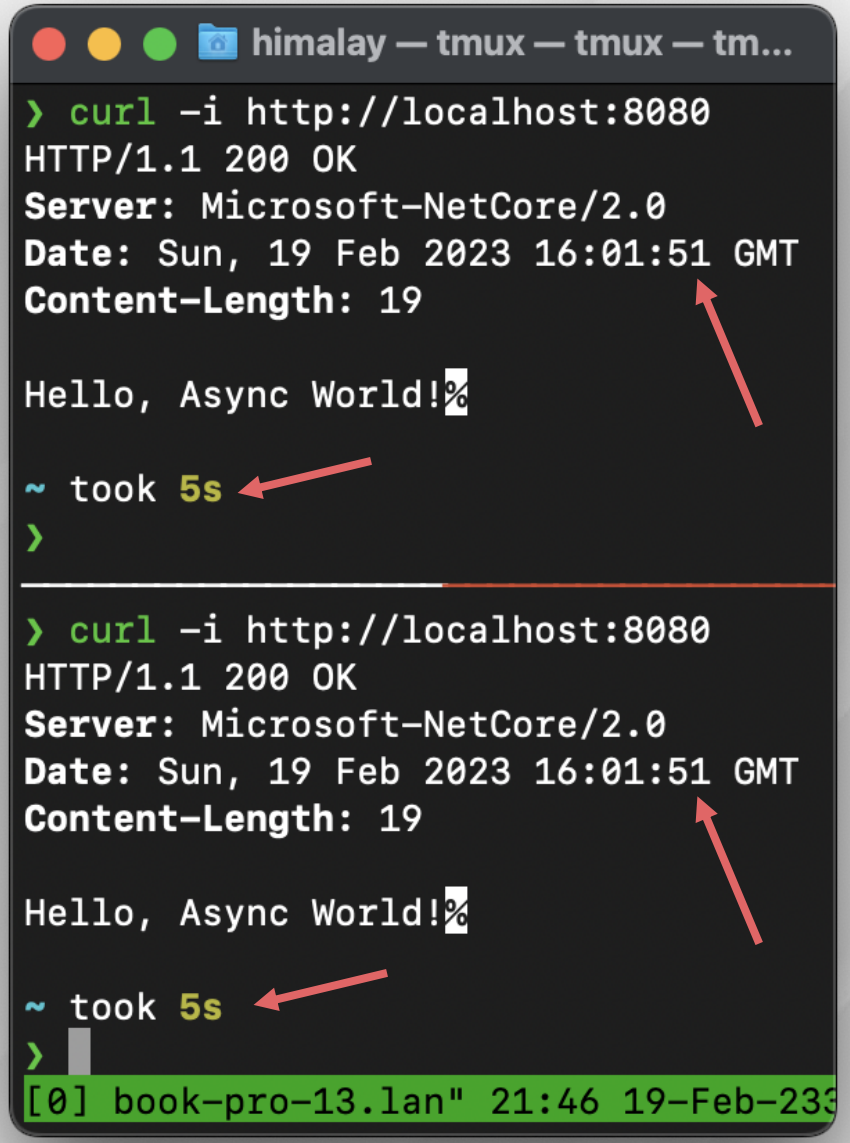
Example: Synchronous

```
1. using System.Net;

2. HttpListener listener = new();
3. listener.Prefixes.Add("http://localhost:8080/");
4. listener.Start();

5. while (true)
6. {
7.     var context = listener.GetContext();
8.     HandleGetRequest(context);
9.     // HandleGetRequestAsync(context);
10.}
```



Example: Asynchronous

```
1. using System.Net;

2. HttpListener listener = new();
3. listener.Prefixes.Add("http://localhost:8080/");
4. listener.Start();

5. while (true)
6. {
7.     var context = listener.GetContext();
8.     // HandleGetRequest(context);
9.     HandleGetRequestAsync(context);
10.}
```



Thank You