UNIVERSITY OF WESTMINSTER

Enterprise Application Development

Lecture 7 – SOLID principles (in C#)

OO programing components

- There are four core components of object-oriented programming
  - Polymorphism
    - An ability of an object to take on different forms
  - Abstraction
    - Actual implementation is hidden from the user and only required functionality will be accessible or available to the user
  - Inheritance
    - It provides code re-usability
  - Encapsulation
    - Grouping data and methods within one unit (class)

What is SOLID principles?

- In OO (object-oriented) programming, SOLID is an acronym for five design principles:
  - **S** - Single-responsibility principle
  - **O** - Open-closed principle
  - **L** - Liskov substitution principle
  - **I** - Interface segregation principle
  - **D** - Dependency inversion principle

Why do we use SOLID?

- SOLID design principles make software designs more understandable, flexible and maintainable.

- The SOLID principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected.

- They are conceptualized by Robert C. Martin (“Uncle Bob”)

Single-responsibility principle

- Each class, module, or function in the program should only do one job
- In other words, each (class, module or function) should have full responsibility for a single functionality of the program
- The class should contain only variables and methods relevant to its functionality
- We could change all but one class in the program without breaking the original class

UNIVERSITY OF WESTMINSTER

Single-responsibility principle

```
// does not follow SRP
public class RegisterService
{
    public void RegisterUser(string username)
    {
        if (username == "admin")
            throw new InvalidOperationException();

        SqlConnection connection = new SqlConnection();
        connection.Open();
        SqlCommand command = new SqlCommand("INSERT INTO [...]"); //Insert user into database.

        SmtpClient client = new SmtpClient("smtp.myhost.com");
        client.Send(new MailMessage()); //Send a welcome email.
    }
}
```

UNIVERSITY OF WESTMINSTER

Single-responsibility principle

```
public void RegisterUser(string username)
{
    if (username == "admin")
        throw new InvalidOperationException();

    _userRepository.Insert(...);

    _emailService.Send(...);
}
```

Open-closed principle

- Software components should be open to extension and closed for modifications
- Allow the behavior of the system to be changed by adding new code, rather than changing existing code
- Through polymorphism, we can extend our parent entity to suit the needs of the child entity while leaving the parent intact

UNIVERSITY OF WESTMINSTER

Open-closed principle

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class Circle
{
    public double Radius { get; set; }
}
```

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

UNIVERSITY OF WESTMINSTER

Open-closed principle

```
public abstract class Shape
{
    public abstract double Area();
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

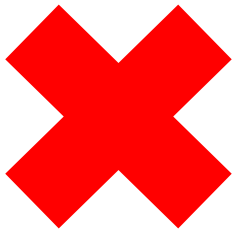
```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

Liskov substitution principle

- To build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another
- Any class can be directly replaceable by any of its subclasses without any errors
- It speeds up the development of new subclasses as all subclasses of the same type share a consistent behavior
- You can trust that all newly created subclasses will work with the existing code

UNIVERSITY OF WESTMINSTER

Liskov substitution principle



```
public class SumCalculator
{
   protected readonly int[] _numbers;
   public SumCalculator(int[] numbers)
   {
      _numbers = numbers;
   }
   public int Calculate()
   {
         return numbers.Sum();
   };
}

public class EvenNumbersSumCalculator : SumCalculator
{
   public EvenNumbersSumCalculator(int[] numbers) : base(numbers)  {}
   public new int Calculate()
   {
         return _numbers.Where(x => x % 2 == 0).Sum();
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      var numbers = new int[] { 5, 7, 9, 8, 1, 6, 4 };
      SumCalculator sum = new SumCalculator(numbers);
      Console.WriteLine($"The sum of all the numbers: {sum.Calculate()}");
      EvenNumbersSumCalculator evenSum =
                              new EvenNumbersSumCalculator(numbers);
      Console.WriteLine($"The sum of all the even numbers: {evenSum.Calculate()}");
   }
}
```

```
//EvenNumbersSumCalculator evenSum = new EvenNumbersSumCalculator(numbers);
 SumCalculator evenSum = new EvenNumbersSumCalculator(numbers);
```

UNIVERSITY OF WESTMINSTER

Liskov substitution principle

```
public class SumCalculator
{
   protected readonly int[] _numbers;
   public SumCalculator(int[] numbers)
   {
      _numbers = numbers;
   }
   public virtual int Calculate()
   {
         return numbers.Sum();
   };
}

public class EvenNumbersSumCalculator : SumCalculator
{
   public EvenNumbersSumCalculator(int[] numbers) : base(numbers)  {}
   public override int Calculate()
   {
         return _numbers.Where(x => x % 2 == 0).Sum();
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      var numbers = new int[] { 5, 7, 9, 8, 1, 6, 4 };
      SumCalculator sum = new SumCalculator(numbers);
      Console.WriteLine($"The sum of all the numbers: {sum.Calculate()}");
      SumCalculator  evenSum =
                        new EvenNumbersSumCalculator(numbers);
      Console.WriteLine($"The sum of all the even numbers: {evenSum.Calculate()}");
   }
}
```

Interface segregation principle

- This principle says many smaller specific interfaces are better than one general-purpose interface
- Requires that classes only be able to perform behaviors that are useful to achieve its end functionality
- In other words, classes do not include behaviors they do not use.

UNIVERSITY OF WESTMINSTER

Interface segregation principle

```
public interface IWorker
 {
    string ID { get; set; }
    string Name { get; set; }
    string Email { get; set; }
    float MonthlySalary { get; set; }
    float OtherBenefits { get; set; }
    float HourlyRate { get; set; }
    float HoursInMonth { get; set; }
    float CalculateNetSalary();
    float CalculateWorkedSalary();
 }

 public class FullTimeEmployee : IWorker
 {
    public string ID { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public float MonthlySalary { get; set; }
    public float OtherBenefits { get; set; }
    public float HourlyRate { get; set; }
    public float HoursInMonth { get; set; }
    public float CalculateNetSalary() => MonthlySalary + OtherBenefits;
    public float CalculateWorkedSalary() => throw new NotImplementedException();
 }
```

```
public class ContractEmployee : IWorker
 {
    public string ID { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public float MonthlySalary { get; set; }
    public float OtherBenefits { get; set; }
    public float HourlyRate { get; set; }
    public float HoursInMonth { get; set; }
    public float CalculateNetSalary() => throw new NotImplementedException();
    public float CalculateWorkedSalary() => HourlyRate * HoursInMonth;
 }
```

UNIVERSITY OF WESTMINSTER

Interface segregation principle

```
public interface IBaseWorker
  {
    string ID { get; set; }
    string Name { get; set; }
    string Email { get; set; }
  }

  public interface IFullTimeWorkerSalary : IBaseWorker
  {
    float MonthlySalary { get; set; }
    float OtherBenefits { get; set; }
    float CalculateNetSalary();
  }

  public interface IContractWorkerSalary : IBaseWorker
  {
    float HourlyRate { get; set; }
    float HoursInMonth { get; set; }
    float CalculateWorkedSalary();
  }
```

```
public class FullTimeEmployeeFixed : IFullTimeWorkerSalary
  {
    public string ID { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public float MonthlySalary { get; set; }
    public float OtherBenefits { get; set; }
    public float CalculateNetSalary() => MonthlySalary + OtherBenefits;
  }

  public class ContractEmployeeFixed : IContractWorkerSalary
  {
    public string ID { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public float HourlyRate { get; set; }
    public float HoursInMonth { get; set; }
    public float CalculateWorkedSalary() => HourlyRate * HoursInMonth;
  }
```

Dependency inversion principle

- This principle tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions

- High-level modules should not depend on low-level modules, and they should both depend on abstractions (interfaces)

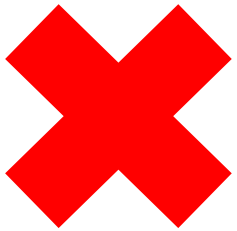
- Abstractions should not depend on details (concrete implementations)

UNIVERSITY OF WESTMINSTER

Dependency inversion principle

```
public class Email
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendEmail()    {
        //Send email
    }
}

public class SMS
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendSMS()    {
        //Send sms
    }
}
```

```
public class Notification
{
    private Email _email;
    private SMS _sms;
    public Notification()
    {
        _email = new Email();
        _sms = new SMS();
    }

    public void Send()
    {
        _email.SendEmail();
        _sms.SendSMS();
    }
}
```



UNIVERSITY OF WESTMINSTER

Dependency inversion principle

```
public interface IMessage
{
    void SendMessage();
}

public class Email : IMessage
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendMessage()
    {
        //Send email
    }
}
```

```
public class SMS : IMessage
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendMessage()
    {
        //Send sms
    }
}
```

```
public class Notification
{
    private ICollection<IMessage> _messages;

    public Notification(ICollection<IMessage> messages)
    {
        this._messages = messages;
    }
    public void Send()
    {
        foreach(var message in _messages)
        {
            message.SendMessage();
        }
    }
}
```

Resources

- Useful links
  - https://www.educative.io/blog/solid-principles-oop-c-sharp?aid=5082902844932096
  - https://exceptionnotfound.net/tag/solidprinciples/
  - https://code-maze.com/single-responsibility-principle/

- Books
  - Clean Architecture: A Craftsman’s Guide to Software Structure and Design, Robert C. Martin