Design Principles

Design principles are fundamental guidelines that help software developers create robust, maintainable, and scalable software systems

Robust : ability of a software system to handle unexpected conditions, errors, and inputs without crashing or producing incorrect results.

Maintainable : ease with which a software system can be modified, extended, and updated over time.

Scalable : refers to the ability of a software system to handle increasing amounts of load or data without a proportional increase in resources.

Why do we need good design?

- To deliver faster
- To manage change
- To deal with complexity

Charactersitics of Good design?

High Cohesion : keeping parts of a code base that are related to each other in a single place.

Low coupling : separating unrelated parts of the code base as much as possible.

Why use design principles?

To avoid rigidity(hard to change because every change affects too many

other parts of the system.), fragility(when a change is made, unexpected parts of the system break), Immobility (hard to reuse in another system)

Design principles provide guidelines and best practices to create effective and well structured software

Improved code quality, making it easier for developers to understand, modify, and collaborate on the codebase.

Modularity and reusability (to use existing code in new concepts as a result reduce the redundancy)

Scalable and adaptable to changing requirements

By having error handling and fault tolerance mechanisms, it helps build systems resilient to unexpected conditions.

If a development team uses same set of principles it would be easier to collaborate and do team work

A well designed software is easier to test, as forming test suits become easier.

Makes it easier to integrate with other systems and tools.

Long-term maintainability of a system by ensuring that it can evolve over time, accommodate new features, and adapt to changing technological landscapes.

The design principles are:

- SOLID
- DRY
- KISS
- YAGNI

SOLID

- **SRP (single responsibilty principle) - A class should have one and only one reason to change.**

Should have only one responsibilty. Primary purpose is to have maintainability, readability and flexibility of code.

Why we have SRP?

1. When a class have only one reason , it will be easy to maintain as changes to one aspect wont affect the other system as a result it wont create any bugs.
2. When each class focuses on one task, it helps the developers to understand the purpose and functionality. Makes it readable and flexible.
3. Classes with only single responsibility can be reused in other functionality in various contexts without any unnecessary baggage. Example : If we create a logging class for logging console errors then the same errors can be used in different context for logging the database errors.
4. Easier to test. If class has only one responsibilty it will be easier to write the test cases. So, can verify the correctness of the code
5. When a system evolves or changes, classes with single responsibilty can be extended or modified without affecting other parts of the system.
6. Have lower complexity as a result it is easy to understand, debug and enhance the code.

Example:

```
// Before applying SRP

        class Employee

        {

           public void CalculateSalary() { /* logic for calculating salary */ }

           public void SaveToDatabase() { /* logic for saving to database */ }

        }
```

```
// After applying SRP

class Employee

{

   public void CalculateSalary() { /* logic for calculating salary */ }

}


class EmployeeRepository

{

   public void SaveToDatabase(Employee employee) { /* logic for saving to database */ }

}
```

- **OCP (Open closed principle) - Open for extension but closed for modification.**

  We should be able to extend the behaviour of a module without altering its existing source code.

  Can be achieved by using abstract classes, interfaces we can have a place that allows for future extensions without modifying existing code. Also by using inheritance.

  Why we need OCP principle?

  - Imagine we have a large codebase with many interconnected modules. If we need to add new features, then there is a high risk of bugs and disturbs the existing functionality. Now, with this principle, we can close the modification to the existing code and only add new features through extensions, like that we wont disturb the functionality of the existing code.
  - In a content management system, where we add multiple modules then we can create interfaces and the new functionality and features can be implemented by implementing the interfaces.
  - Developers can extend the framework's behavior by creating new classes or overriding methods without modifying the existing framework code. So, here OCP is must.
  - When integrating third party libraries it is important to not modify the existing code.
  - Multiple people can work on different modules without interfering with each other work as long as they follow the present interfaces and base classes.

```
// without the OCP principle

public class AreaCalculator

{
    public double CalculateArea(object shape)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle)shape;
            return rectangle.Width * rectangle.Height;
        }
        else if (shape is Circle)
        {
            Circle circle = (Circle)shape;
            return Math.PI * Math.Pow(circle.Radius, 2);
        } return 0;
        // Additional checks for other shapes...
     }
}

// with the OCP Principle
public abstract class Shape
{
    public abstract double CalculateArea();
}
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public override double CalculateArea()
    {
        return Width * Height;
    }
}
public class Circle : Shape
{
    public double Radius { get; set; }

    public override double CalculateArea()
    {
        return Math.PI * Math.Pow(Radius, 2);
    }
}
// Adding a new shape without modifying existing code
public class Triangle : Shape
{
    public double Base { get; set; }
    public double Height { get; set; }
    public override double CalculateArea()
    {
        return 0.5 * Base * Height;
    }
}
```

- **LSP (Liskov Substitution Principle) - Sub types must be substitutable by the base class (Super class must be replaceable by sub class then only to use inheritance)**

Objects of the super class should be replaced by the objects of the sub class without affecting the correctness of the program

LSP is crucial to building robust and maintainable software systems, preventing unexpected behavior when substituting objects of a derived class for objects of a base class. By adhering to LSP, developers can confidently extend and enhance their codebase, making it more adaptable to changes and scalable over time.

```
//without LSP
public class Rectangle
        {
          public virtual void SetWidth(int width)
          {
            // Set the width of the rectangle
          }
          public virtual void SetHeight(int height)
          {
            // Set the height of the rectangle
          }
          public int CalculateArea()
          {
            return Width * Height;
          }
          public int Width { get; set; }
          public int Height { get; set; }
        }
        public class Square : Rectangle
        {
          public override void SetWidth(int width)
          {
            base.SetWidth(width);
            base.SetHeight(width);
          }

          public override void SetHeight(int height)
          {
            base.SetHeight(height);
            base.SetWidth(height);
          }
        }
        class Program
        {
        // Here the Rectangle is expected to behave like the rectangle method where the height and
        width are different but as Square extends rectangle and the methods are overridden so the
        height and width will be same, as a result there will be inconsistencies.
        public static void ResizeRectangle(Rectangle rectangle, int newWidth, int newHeight)
        {
          rectangle.SetWidth(newWidth);
          rectangle.SetHeight(newHeight);
        }
        // Somewhere else in the code
        Rectangle myRectangle = new Square();
        ResizeRectangle(myRectangle, 5, 10);
        }


        // To avoid the above issue using LSP
        public abstract class Shape
        {
          public abstract void SetDimensions(int width, int height);
          public abstract int CalculateArea();
        }
        public class Rectangle : Shape
        {
          private int Width { get; set; }
          private int Height { get; set; }
          public override void SetDimensions(int width, int height)
          {
            Width = width;
            Height = height;
          }
          public override int CalculateArea()
          {
            return Width * Height;
          }
        }
        public class Square : Shape
        {
          private int Side { get; set; }
          public override void SetDimensions(int width, int height)
          {
            // Ensure that the Square's sides are equal
            int side = Math.Min(width, height);
            Side = side;
          }
          public override int CalculateArea()
          {
            return Side * Side;
          }
        }
        class Program
        {
        public static void ResizeShape(Shape shape, int newWidth, int newHeight)
        {
          shape.SetDimensions(newWidth, newHeight);
        }
```

```
// Somewhere else in the code

Shape myShape = new Square();

ResizeShape(myShape, 5, 5);  // No issues with LSP adherence

}
```

- **ISP (Interface Segregation principle) - the dependency of one class to another must be based on smallest possible interface. Instead of one big interface, it is better to have small interfaces based on groups of methods, each one serving one submodule.**

Example : User Authentication system has logging and authentication, some dont have logging   so they can be created separately in two interfaces instead of one interface.

```
// without ISP

// Interface combining printing and scanning

public interface IMachine

{

   void Print();

   void Scan();

}

// Printer implementing IMachine

//printer dont scan but it is forced to implement it

public class Printer : IMachine

{

   public void Print()

   {

      Console.WriteLine("Printing...");

   }

   public void Scan()

   {
}
        // Not relevant for printers, but forced to implement
        throw new NotSupportedException("Printers cannot scan.");
    }
}
// Scanner implementing IMachine
//scanner dont print, but it is forced to implement the method
public class Scanner : IMachine
{
    public void Print()
    {
        // Not relevant for scanners, but forced to implement
        throw new NotSupportedException("Scanners cannot print.");
    }
    public void Scan()
    {
        Console.WriteLine("Scanning...");
    }
}

// to use ISP
// Separate interfaces for printing and scanning
public interface IPrinter
{
    void Print();
}
public interface IScanner
{
    void Scan();
}
```

```
// Printer implementing IPrinter
public class Printer : IPrinter
{
   public void Print()
   {
      Console.WriteLine("Printing...");
   }
}
// Scanner implementing IScanner
public class Scanner : IScanner
{
   public void Scan()
   {
      Console.WriteLine("Scanning...");
   }
}
```

- DIP (Dependency Injection principle) - Depend on interfaces instead of concrete classes.

  High level modules contain business logic and major principles so they should not depend on low level modules which implement the details and specifics, instead both should depend on the abstractions. Also the abstractions should not depend on the details but the details should be depended on abstractions. Abstractions represent a contract or a set of common behaviors that both high-level and low-level modules can adhere to.

```
// without DIP
// High-level module directly depends on low-level module
public class Logger
{
   private FileLogger fileLogger;   // low module instance
   public Logger()
```

```
    {
        fileLogger = new FileLogger();
    }
    public void LogMessage(string message)
    {
        fileLogger.Log(message);
    }
}
// Low-level module
public class FileLogger
{
    public void Log(string message)
    {
        // Log message to a file
    }
}

//with DIP
// Abstraction (interface)
public interface ILogger
{
    void Log(string message);
}

// High-level module depends on abstraction
public class Logger
{
    private ILogger logger;
    public Logger(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogMessage(string message)
    {
        logger.Log(message);
    }
}
// Low-level modules depend on abstraction
public class FileLogger : ILogger
{
public void Log(string message)
    {
        // Log message to a file
    }
}
public class ConsoleLogger : ILogger   // low level module.
{
    public void Log(string message)
    {
        // Log message to the console
    }
}
```

<u>**DRY**</u>

Dont Repeat Yourself

Every piece of knowledge must have single, unambiguos and unauthoritative representation within the system.

Each small pieces of code comes exactly once in the entire system. This helps to write scalable, consistent, maintainable and reusable code.

Consider a scenario where you're developing a web application that involves handling user authentication. In this application, you have different pages where users can sign up, log in, and update their profile information. Without adhering to the DRY principle, you might end up duplicating code related to user authentication across these different pages.

KISS

Keep It Simple Stupid

Simplicity should be the key goal in any software system.

Should avoid complexities

The methods should be small, not more than 50 lines.

Each method should define only one problem but solve many usecases.

If we have more conditions in a method break these out into smaller methods, which helps us to understand the code better and debugging will be easier.

Imagine you are tasked with designing a user interface for a calendar application. The application needs to allow users to view their events, add new events, and delete existing events. Without adhering to the KISS principle, you might be tempted to create a complex and feature-rich interface that includes numerous buttons, options, and advanced settings.

YAGNI

You Aint Gonna Need It

Always implement things when needed them dont implement before you need them.

Suppose you are developing a blogging platform, and your initial requirements include creating, editing, and deleting blog posts. In the planning phase, you might consider implementing a feature for allowing users to categorize their blog posts with tags, even though the current requirements don't explicitly mention this functionality.