SOLID Principles

LCSCI5202: Object Oriented Design Week 6

Waleed Iqbal – 2025/26

The SOLID Principles Overview

- **S**: Single Responsibility Principle
- **O**: Open/Closed Principle
- **L**: Liskov Substitution Principle
- **I**: Interface Segregation Principle
- **D**: Dependency Inversion Principle

Core Concepts:

- Each principle guides us towards specific design choices that enhance code quality
- Together, they promote loose coupling, high cohesion, and flexibility.

Single Responsibility Principle (SRP)

SRP: **One Job, Done Well**
Definition: A class should have only one reason to change.

- **Benefits:** Easier to understand, test, and modify individual components.
- **Challenges:** Identifying the "**right**" level of granularity for responsibilities.
- **Analogy:** A Swiss Army Knife is versatile, but a dedicated tool is often better for a specific task.

C# Example Code – Violating SRP

```
public class Employee {
            public string Name { get; set; }
            public decimal Salary { get; set; }
            public decimal CalculatePay() { return Salary * 1.1m; }
            public string GenerateReport()
                        { return $"Employee: {Name}, Salary: {Salary}"; }
            public void SaveToDatabase() { // Database logic
}
}
```

Changes to reporting, database, or payroll logic all affect this class

C# Example – Following SRP

```
public class Employee {
        public string Name { get; set; }
        public decimal Salary { get; set; }
}
public class PayrollCalculator {
        public decimal CalculatePay(Employee employee) {
                return employee.Salary * 1.1m; }
}
public class EmployeeReportGenerator {
        public string GenerateReport(Employee employee) {
                return $"Employee: {employee.Name}"; }
}
```

Open/Closed Principle (OCP)

OCP: **Open for Extension, Closed for Modification**

Definition: Software entities (classes, modules, functions) should be open for extension, but closed for modification.

- **Benefits**: Reduces the risk of introducing bugs when adding new features.
- **Challenges**: Requires careful upfront design and abstraction.
- **Analogy**: A well-designed house has room for additions without needing to tear down existing walls.
- **Tools to Achieve:** Inheritance, Polymorphism, and Interfaces

C# Example Code - Violating OCP

```
public class AreaCalculator {
	public double CalculateArea(object shape) {
		if (shape is Rectangle rectangle) {
			return rectangle.Width * rectangle.Height;
		}
		else
			if (shape is Circle circle) {
				return Math.PI * circle.Radius * circle.Radius;
			}
		throw new ArgumentException("Unknown shape");
	}
}
```

C# Example Code - Following OCP

```
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 AreaCalculator {
		public double CalculateArea(Shape shape) {
				return shape.CalculateArea();
		}
}
```

Liskov Substitution Principle (LSP)

LSP: Substitutability without Surprises

- **Definition**: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

- **Benefits:** Ensures code behaves predictably when using inheritance.
- **Challenges:** Requires careful consideration of class hierarchies and contracts.
- **Analogy:** A child class should be able to fill its parent's shoes without causing chaos.

C# Example Code - Violating LSP

```
public class Bird {
        public virtual void Fly() {
                Console.WriteLine("Bird is flying");
        }
}
public class Penguin : Bird {
        public override void Fly() { // Violates LSP - breaks substitution!
                throw new NotSupportedException("Penguins can't fly!");
        }
} // Client code breaks with Penguin

public void MakeBirdFly(Bird bird) {
        bird.Fly(); // Exception if bird is Penguin!
}
```

C# Example Code - Following LSP

```
public abstract class Bird {
        public abstract void MakeSound();
        public virtual void Eat() {
        /* common behavior */
        }
}
public interface IFlyable {
        void Fly();
}
public class Sparrow : Bird, IFlyable {
        public override void MakeSound(){
                Console.WriteLine("Chirp");
        }
        public void Fly() {
        Console.WriteLine("Flying high!");
        }
}
public class Penguin : Bird { // No IFlyable!
        public override void MakeSound() {
                Console.WriteLine("Squawk");
        }}
```

Interface Segregation Principle (ISP)

ISP: Don't Force Clients to Depend on Interfaces They Don't Use

Definition: Many client-specific interfaces are better than one general-purpose interface

- **Benefits:** Improves code flexibility and reduces unnecessary dependencies
- **Challenges:** Can lead to a proliferation of interfaces if not managed carefully
- **Analogy:** A restaurant menu with separate sections for appetizers, main courses, and desserts is easier to navigate than one giant list.

C# Example Code - Violating ISP

```
public interface IWorker {
        void Work();
        void Eat();
        void Sleep();
        void AttendMeeting();
}
public class RobotWorker : IWorker {
        public void Work() {
                Console.WriteLine("Working...");
        }
// Forced to implement methods robots don't need!
        public void Eat(){
                throw new NotSupportedException();
        }
        public void Sleep(){
                throw new NotSupportedException();
        }
        public void AttendMeeting(){
                throw new NotSupportedException();
        }}
```

C# Example Code - Following ISP

```
public interface IWorkable {
            void Work();}
public interface IFeedable {
            void Eat(); }
public interface IMeetingAttendee {
            void AttendMeeting();}

public class RobotWorker : IWorkable {
            public void Work(){
                        Console.WriteLine("Robot working...");}
}

public class HumanWorker : IWorkable, IFeedable, IMeetingAttendee {
            public void Work() {
                        Console.WriteLine("Human working...");}
            public void Eat(){
                        Console.WriteLine("Eating lunch...");}
            public void AttendMeeting(){
            Console.WriteLine("In meeting...");}
}
```

Dependency Inversion Principle (DIP)

DIP: Depend on Abstractions, Not Concretions

Definition:

- Abstractions should not depend on details. Details should depend on abstractions.
- High-level modules should not depend on low-level modules. Both should depend on abstractions.

- **Benefits:** Makes code more testable, maintainable, and adaptable to change
- **Challenges:** Adds a layer of abstraction, which can increase complexity initially
- **Analogy:** A car's engine shouldn't be directly welded to the chassis; they should connect via standardized interfaces.

C# Example Code – Violating DIP

```
public class EmailService
{
        public void SendEmail(string message){
                Console.WriteLine("Sending Email: " + message);
        }
}

public class Notification
{
        private EmailService _emailService;

        public Notification(){
                _emailService = new EmailService();     }


        public void Send(string message){
                _emailService.SendEmail(message);
        }
}
```

C# Example Code – Following DIP

```
public interface IMessageService
{
		void SendMessage(string message);
}

public class EmailService : IMessageService
{
		public void SendMessage(string message){
				Console.WriteLine("Sending Email: " + message);}
}

public class SMSService : IMessageService
{
		public void SendMessage(string message){
				Console.WriteLine("Sending SMS: " + message);}
}
```

C# Example Code – Following DIP

```
public class Notification
{
        private IMessageService _messageService;

        public Notification(IMessageService messageService){
                _messageService = messageService;
        }

        public void Send(string message){
                _messageService.SendMessage(message);
        }
}
IMessageService emailService = new EmailService();

IMessageService smsService = new SMSService();
```

Common Pitfalls and Misconceptions

Over-Engineering:

Don't force SOLID principles into every corner of your codebase.

Start by applying them to core components or areas prone to change.

Strive for a balance between flexibility and simplicity.

Premature Optimization:

Avoid over-abstracting code that is stable or rarely modified.

Focus on SOLID when designing new features or refactoring problematic areas.

SOLID is not just for OOP:

SOLID principles are rooted in sound software design principles that apply to any programming paradigm.

The core concepts of loose coupling, high cohesion, and flexibility are universally beneficial.

Summary

- SOLID principles are not just theoretical concepts; they are practical tools for crafting robust, adaptable software.
- By embracing SOLID, we can create codebases that are easier to understand, modify, and extend, leading to increased productivity and reduced technical debt.
- Make code open for extension but closed for modification to support growth without breaking existing logic.
- Ensure subclasses can fully substitute their base classes without unexpected behavior.
- Depend on abstractions (interfaces) rather than concrete implementations for flexibility.
- Apply dependency injection to decouple modules and improve testability.