Title: Mastering SOLID Principles in C#

Introduction

Welcome to this comprehensive guide to mastering the SOLID principles in C#. These principles are foundational for writing clean, maintainable, and scalable object-oriented code. By the end of this lecture, you'll be able to recognize violations of SOLID and refactor your code accordingly.

What is SOLID?

SOLID is an acronym that represents five principles:

- S: Single Responsibility Principle (SRP)

- O: Open/Closed Principle (OCP)

- L: Liskov Substitution Principle (LSP)

- I: Interface Segregation Principle (ISP)

- D: Dependency Inversion Principle (DIP)

These were introduced by Robert C. Martin (Uncle Bob) and help in building robust software architectures.

1. Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change.

Why It Matters: Combining multiple responsibilities in a single class makes your code fragile and harder to maintain.

Bad Example:

public class Report

{

public string Title { get; set; }

public void SaveToFile(string filePath) {

// Save logic

}

public void Print() {

// Print logic

}

}

Improved Example:

public class Report

{

public string Title { get; set; }

}

public class ReportPrinter

{

public void Print(Report report) {

// Print logic

}

}

public class ReportSaver

{

public void SaveToFile(Report report, string filePath) {

// Save logic

}

}

2. Open/Closed Principle (OCP)

Definition: Software entities should be open for extension but closed for modification.

Why It Matters: When requirements change, you want to extend behavior without breaking existing code.

Bad Example:

public class InvoicePrinter

{

public void PrintInvoice(string type) {

if (type == "PDF") {

// PDF logic

} else if (type == "Excel") {

// Excel logic

}

}

}

Improved Example:

public interface IInvoicePrinter {

void Print();

}

public class PdfPrinter : IInvoicePrinter {

public void Print() {

// PDF logic

}

}

public class ExcelPrinter : IInvoicePrinter {

public void Print() {

// Excel logic

}

}

public class InvoicePrintService {

public void PrintInvoice(IInvoicePrinter printer) {

printer.Print();

}

}

3. Liskov Substitution Principle (LSP)

Definition: Subtypes must be substitutable for their base types.

Why It Matters: If a subclass cannot stand in for its base class, inheritance breaks your design.

Bad Example:

public class Bird {

public virtual void Fly() {}

}

public class Ostrich : Bird {

public override void Fly() {

throw new NotImplementedException("Ostriches can't fly!");

}

}

Improved Example:

public abstract class Bird {}

public interface IFlyingBird {

void Fly();

}

public class Sparrow : Bird, IFlyingBird {

public void Fly() {

// Flying logic

}

}

public class Ostrich : Bird {

// No flying behavior

}

4. Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on methods they do not use.

Why It Matters: Fat interfaces create unnecessary implementation overhead and coupling.

Bad Example:

public interface IMachine {

void Print();

void Scan();

void Fax();

}

public class OldPrinter : IMachine {

public void Print() {}

public void Scan() { throw new NotImplementedException(); }

public void Fax() { throw new NotImplementedException(); }

}

Improved Example:

public interface IPrinter {

void Print();

}

public interface IScanner {

void Scan();

}

public class SimplePrinter : IPrinter {

public void Print() {}

}

public class MultifunctionPrinter : IPrinter, IScanner {

public void Print() {}

public void Scan() {}

}

5. Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions.

Why It Matters: Tightly coupled systems are hard to test and extend.

Bad Example:

public class FileLogger {

public void Log(string message) {

// Log to file

}

}

public class AuthService {

private FileLogger logger = new FileLogger();

public void Login(string user) {

logger.Log("User logged in");

}

}

Improved Example:

public interface ILogger {

void Log(string message);

}

public class FileLogger : ILogger {

public void Log(string message) {

// Log to file

}

}

public class AuthService {

private readonly ILogger \_logger;

public AuthService(ILogger logger) {

\_logger = logger;

}

public void Login(string user) {

\_logger.Log("User logged in");

}

}

Conclusion

- SRP: Keep responsibilities focused.

- OCP: Extend, don't modify.

- LSP: Use inheritance properly.

- ISP: Avoid fat interfaces.

- DIP: Depend on abstractions.

By following these principles, your code becomes easier to test, scale, and maintain. Practice by identifying violations in real-world projects and refactoring them with SOLID in mind.

Thank you for joining this session on mastering SOLID principles in C#!