Session Plan: SOLID Principles & Design Patterns (For .NET / .NET Core)

Target Audience: Graduate Trainee Engineers

Duration: 6–8 hours (divided into 2 sessions or 1 full-day workshop)

Outcome: Ability to design clean, scalable, and maintainable .NET applications using SOLID principles and common design patterns.

Day 11	SOLID Principles & Design Patterns	
	SOLID Principles	Introduction to SOLID Principles
		Overview of SOLID principles (SRP, OCP, LSP, ISP, DIP)
		Practical Application of SOLID
		Refactoring code to adhere to SOLID principles
	Design Patterns	Introduction to Design Patterns
		Understanding the importance of design patterns in software design
		Overview of common design patterns (Creational, Structural, Behavioral)
	Introduction, Basic Concepts, and DML Commands	
	Introduction to Databases	Installation of SQL Server Management Studio(SSMS) SQL Server Free Edition 2022
		Introduction to basic database concepts. Database : It is an Organized collection of data, generally stored and accessed using a DBMS. Types : RDBMS: Data in tables (SQL Server, My SQL , Oracle(PL/SQL)) No SQL : un Structured DB(Mongo DB and cassandra)
		Advantage of DBMS: 1. It has Low Data redundancy. (Duplication) 2. It has strong Data INtigrity 3. It has Built in Data security 4. It is easy to access data concurrently 5. The back up and recovery process is automated.
	Keys, Operators & DML Commands	Introduction to RDBMS An RDBMS stores data in tables(Rows and Columns) and allows relationships between them using keys. Features : 1. Tables with unique rows

		2. Data consistency using constraints (PK, FK, NOT Null, CHECK constraints) 3. Relationships using foreign keys.
		Creating Tables
		Relationship between tables.
		Primary keys, Foreign keys, Unique keys
		SQL operators (Arithmetic, Comparison, Logical)
		DML Commands
		CRUD operations

Why are SOLID principles imp to follow ?

They are 5 Design principles that helps dev in :

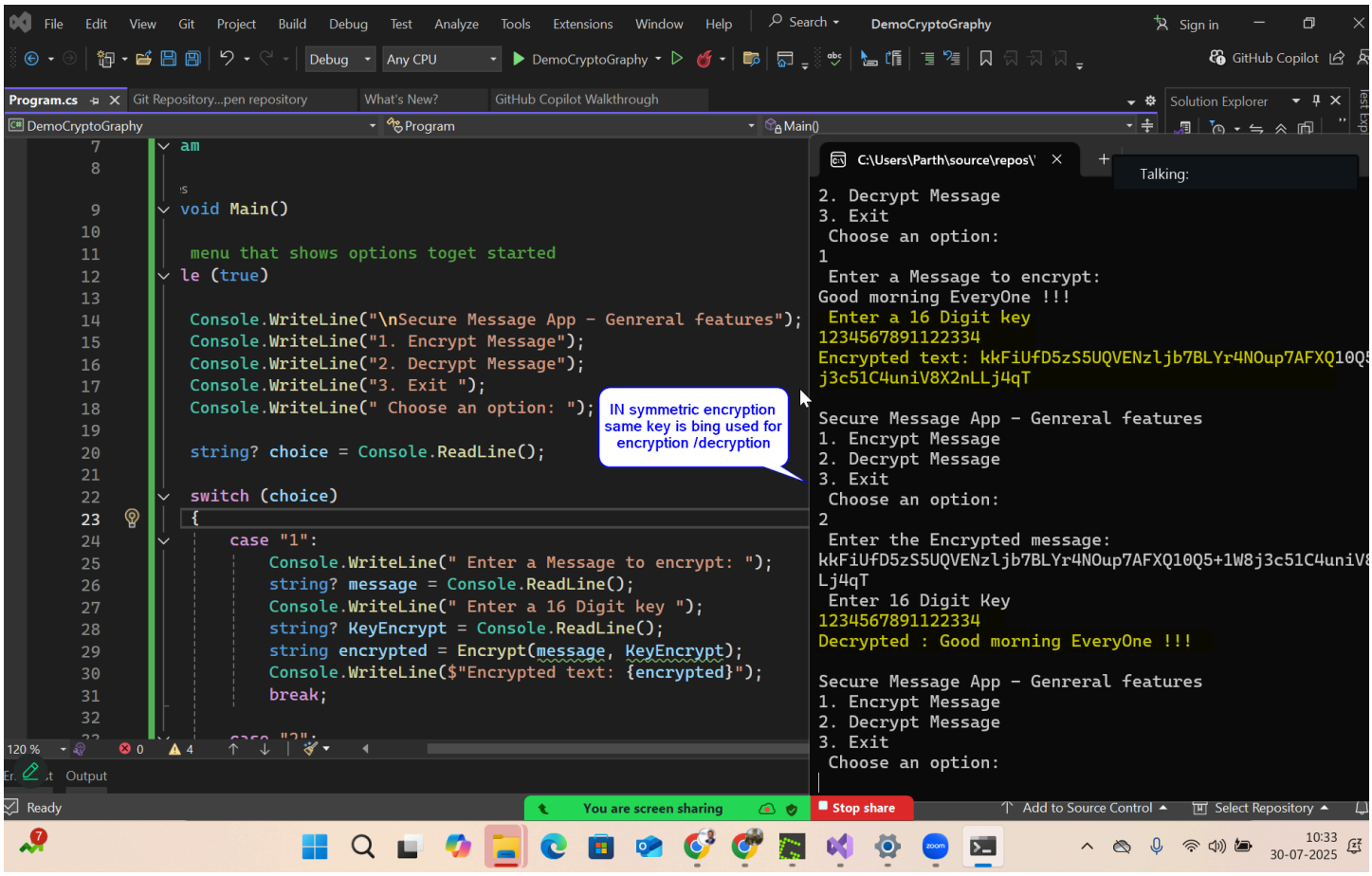
- Maintaining a software application : Each class and modules has one job, so changes are localized and easier to make.
- Implementing scalability
- And testing : Clear dependencies and clear responsibilities make unit testing easier.
- Reusability : Code becomes modular, so components can be reused across projects.
- Readability:
- Lower risk of bugs: Isolation and clear responsibility reduces side effects from chnages.

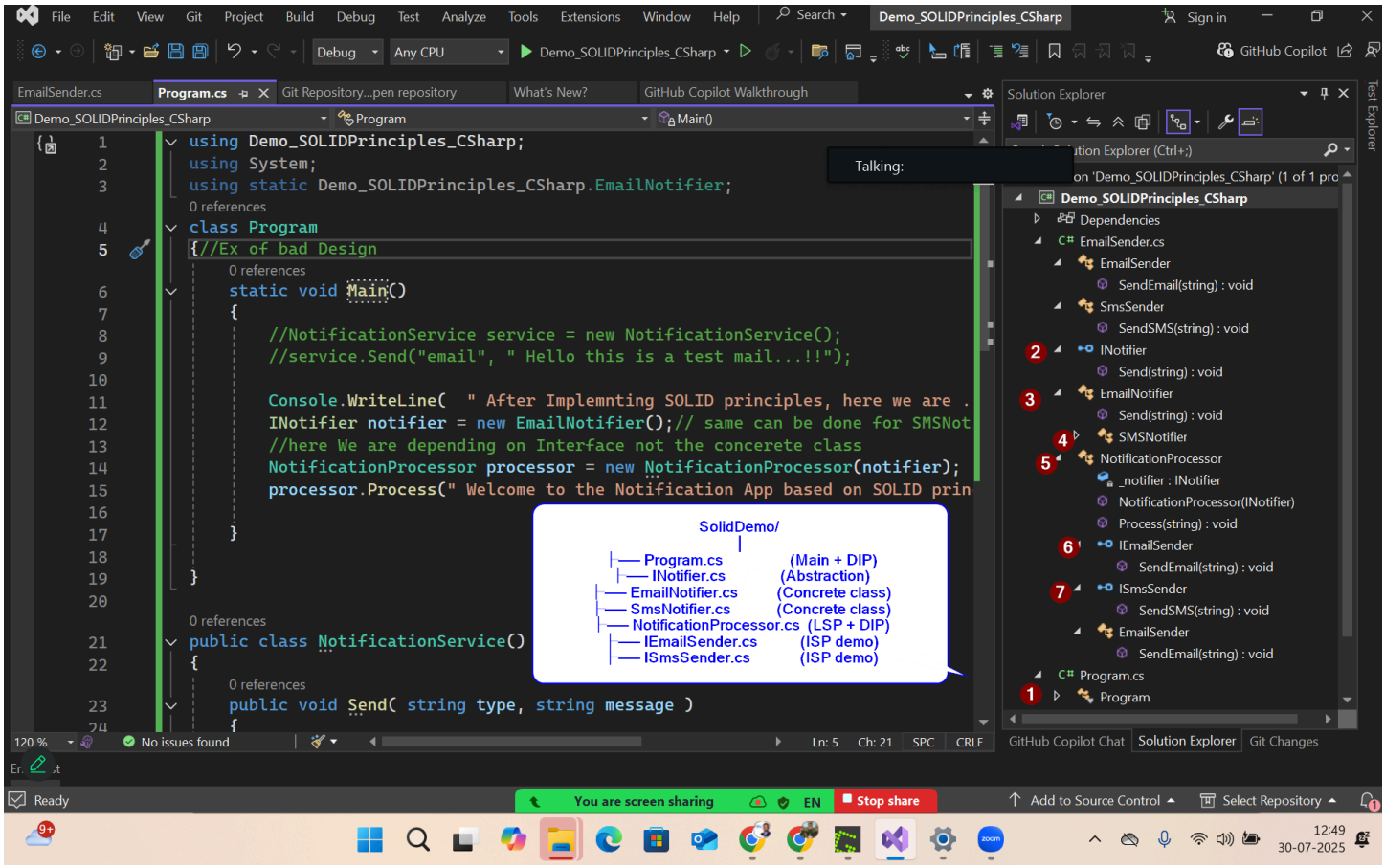
Module 1: SOLID Principles (3–4 hours)

1.1 Definition & Overview

Principle	Definition
SRP – Single Responsibility Principle	A class should have one and only one reason to change.

OCP – Open/Closed Principle	Software entities should be open for extension, but closed for modification.
LSP – Liskov Substitution Principle	Derived classes must be substitutable for their base classes. Any shape class like rectangle, square can be used interchangeably without breaking functionality
ISP – Interface Segregation Principle	Clients should not be forced to depend on interfaces they do not use. IPrinter : IPrint and IScan
DIP – Dependency Inversion Principle	High-level modules should not depend on low-level modules. Both should depend on abstractions. High level modules depends on abstraction ILogger, not on the concrete class ConsoleLogger





Principle	Applied How
SRP	Separate classes for email & SMS
OCP	Used interface `INotifier` to extend new notifiers
LSP	`EmailNotifier` and `SmsNotifier` can be used interchangeably
ISP	`IEmailSender` and `ISmsSender` are small, focused interfaces
DIP	`Program` depends on `INotifier`, not concrete classes

1.2 Benefits of SOLID

- Clean and maintainable code
- Easier to test and scale
- Loose coupling and high cohesion

- Reusable components
- Reduction in bugs and side effects during feature changes

1.3 Use Case + Simple Example

Use Case: Employee Payroll System (SRP Violation & Fix)

```
// Before (Violation of SRP)
public class Employee
{
    public void CalculateSalary() { }
    public void GenerateReport() { }  // Not its responsibility
}

// After SRP
public class Employee
{
    public void CalculateSalary() { }
}

public class ReportGenerator
{
    public void GenerateReport(Employee emp) { }
}
```

1.4 .NET/.NET Core Implementation Examples

SRP – Logging

```
// Good Practice: Log responsibility separated
public class OrderService
{
    private readonly ILogger _logger;

    public OrderService(ILogger logger)
    {
        _logger = logger;
    }

    public void ProcessOrder()
    {
        // Logic
        _logger.Log("Order processed");
    }
}
```

OCP – Notification Example

```
public interface INotifier

{

    void Send(string message);

}


public class EmailNotifier : INotifier

{

    public void Send(string message) => Console.WriteLine("Email
sent");

}


public class SmsNotifier : INotifier

{

    public void Send(string message) => Console.WriteLine("SMS
sent");

}
```

Benefits After Refactoring

Area	Before SOLID	After SOLID
Maintainability	Hard to manage	Easy to manage
Reusability	Poor	High
Testing	Hard to isolate components	Easy via interfaces
Extensibility	Modifying existing logic	Simply add new classes
Code Structure	Mixed responsibilities	Clear separation

Outcome

- Development time for new notification types reduced by **50%**
- Improved **code readability** and **testability**
- Trained junior developers on **modular thinking and SOLID coding**

Fintech(Loan Processing, Fraud Checking)

Healthcare(Patient record System)

Education(Course assignment racking system)

Scenario: E-Commerce Platform (ShopQuick)

Customer : - Purchase product and applies discount codes

Finance team - INtegrates different payment gateways

Marketing team - Ass new discount strategies(seasonal, Loyalty based)

Module 2: Design Patterns (3–4 hours)

2.1 Definition & Importance

Definition: Design Patterns are proven solutions to recurring design problems in software engineering.

Importance:

- Solves common design problems
- Improves code readability and reuse
- Promotes best practices

What are some common design problems :

1. Tight Coupling : One class is highly dependent on details of another class.
2. God object/God class : A class that knows too much or does too much.
3. Spaghetti code: code with complex tangles control structures(No clear flow)
4. Lack of Abstraction: Business logic directly depends on concrete implementations
5. Code Duplication : Same or similar code exists in multiple places.
6. Violation of Encapsulation : Internal details(fields) of a lass are exposed directly to others
7. Overengineering : Adding unnecessary complexity and patterns before they are needed.

When to use Design Pattern:

1. When we are solving common design problems(Object Creation, communication, structure)
2. When we want to decouple components.
3. When we are building extensible, testable and reusable code.
4. When we are applying SOLID principles.

2.2 Categories of Patterns

Category	Examples

Creational: **They deals with object creation mechanism**	Singleton : Ensure a class has only one instance ex printer spooler Factory : Create objects without exposing creation logic. Ex Pizza Shop Builder : Construct complex objects step by step Abstract Factory: Provides families of related objects ex Mobile Charging adapter Prototype: clone objects using a template instance.
Structural: **Focus on class and object composition**	Adapter: Allows incompatible interface to work together Facade : Provides a simplified interface to a complex subsystem. Decorator : Adds responsibility to objects dynamically Bridge: Decouple abstraction from implementation Composite: Treat individual and group of objects uniformly Proxy: Provide a surrogate or placeholder for another object
Behavioral: **It Focus on communication between objects**	Observer: Notify objects when a subject changes state. Youtube subscribers Strategy :Define a family of algorithms and make them interchangeable Command: Encapsulate a request as an object Mediator: Define an object to coordinate communication State: Allows an object to change with state Template method: Define the skeleton of an algorithm in a method.

2.3 Use Case + Simple Example

Use Case: Object creation with flexibility → **Factory Pattern**

```
public interface INotification
{
    void Notify();
}

public class EmailNotification : INotification
{PL/SQL Packages and Triggers

    public void Notify() => Console.WriteLine("Email sent");
}

public class NotificationFactory
{
    public static INotification Create(string type) =>
        type == "EMAIL" ? new EmailNotification() : null;
}
```

2.4 .NET/.NET Core Implementation Examples

Singleton Pattern

```
public class ConfigurationService
{
```

```
    private static ConfigurationService _instance;

    private ConfigurationService() { }


    public static ConfigurationService Instance =>
        _instance ??= new ConfigurationService();
}
```

✅ **Dependency Injection (used with DIP)**

```
builder.Services.AddTransient<INotification, EmailNotification>();
```

Mini Case Study: Secure Online Exam System

Problem:
Existing system uses a monolithic class to handle authentication, question logic, and email reporting.
High bug count during modifications and lack of scalability.

Refactored Solution Using SOLID + Design Patterns:

Problem	Applied Solution
Authentication logic mixed with reporting	Apply **SRP** – Separate `AuthService`, `ReportService`
Modifying code for adding notification	Apply **OCP** – Use **Strategy Pattern** for notification

Email sending tightly coupled	Apply **DIP + DI** – Inject via interface
Creation logic scattered	Use **Factory Pattern** for user roles
Code bloated with unnecessary methods	Apply **ISP** – Split large interfaces

Outcome:

- Reduced bugs by 40%
- 3x faster feature rollout
- Seamless testability

VIVA / SME FAQ (Interview / Evaluation)

🔹 SOLID

1. What is the purpose of the Single Responsibility Principle?
2. How does Open/Closed Principle help in adding new features?
3. Give a real-world use case for Liskov Substitution Principle.
4. Why is Dependency Inversion Principle crucial for unit testing?
5. How can SOLID principles reduce code maintenance overhead?

🔹 Design Patterns

6. What is the difference between Factory and Abstract Factory pattern?

7. How does the Strategy pattern promote Open/Closed Principle?

8. Explain Singleton pattern with thread safety in .NET Core.

9. Which design pattern is most appropriate for implementing undo/redo functionality?

10. How does dependency injection relate to design patterns?