**SOLID Principles**

**What are SOLID Principles:**

They are five design guidelines intended to make software design more understandable, flexible, and maintainable. These principles were introduced by Robert C. Martin and are foundational in Object-Oriented design (OOD), they help in avoiding software design issues that can lead to rigid, fragile, and hard-to-maintain code.

**The SOLID Stands for:**

1. **S** – Single Responsibility Principle (SRP).
2. **O** – Open/Closed Principle (OCP).
3. **L** – Liskov Substitution Principle (LSP).
4. **I** – Interface Segregation Principle (ISP).
5. **D** – Dependency Inversion Principle (DIP).

**Single Responsibility Principle**

A class should have only one reason to change, meaning it should have only one job or responsibility.

This principle helps to ensure that each class or module in you system does one thing and does it well. If class has more than one responsibility, changes to one responsibility may affect the others, making the class more difficult to maintain.

**Open Closed Principle (OCP)**

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

This means that you should be able to extend the behavior of a class without modifying its source code. It’s a way of ensuring that changes don’t break existing code, allowing for greater flexibility and less risk when introducing new functionality.

In software OCP encourages you to extent the functionality of your class without altering their original structure, much like extending your house without knocking it down.

And ensures that the core class (the app) doesn’t need to be modified for every new feature (plugin).

**Liskov Substitution Principle (LSP)**

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

In simpler terms, if a class B is a subclass of class A, then B should be able to replace A without breaking the behavior of the system.

**Any subclass should only extend the behavior of the parent class, not narrow it down.**

Which means, derived classes should be substitutable for their base classes without altering the behavior of the system. This principle is used to ensure that inheritance is used properly.

* LSP ensures that subclass behave consistently when replacing the base class.
* It’s about ensuring that subclasses can replace their base class without breaking the socket. If a class is derived from another, it should still work as expected when substituted.

**Interface Segregation (ISP)**

Clients should not be forced to depend on interfaces they don’t use.

Instead of one large interface, you should break down the interface into smaller, more specific ones. This allows clients to depend only on the methods they need, rather than being force to implement unused methods.

ISP works by ensuring that the clients only interact with the specific functionality they need, without being forced to deal with unnecessary details.

It follows this logic by creating smaller, more specific interfaces so that the clients use only the methods they need.

It is like having those interfaces each interface should offer only what the client needs, not a bloated of unrelated methods. It’s like breaking big-bulky interfaces into smaller, focused ones so that client can use just what they need.

**Dependency Inversion Principle (DIP)**

The High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces or abstract classes).

DIP encourages the decoupling of software modules by ensuring that both high-level and low-level modules depend on abstractions, making your code more flexible and easier to maintain.

DIP promotes that kind of flexibility, where classes rely on abstract concepts (interfaces) instead of concrete details, making your code adaptable to different implementations.

The high0-level doesn’t depend on specific appliance (low-level), and both rely on the abstraction of a standard interface.

Instead of tightly coupling your high-level code to specific implementations (like only supporting one thing), you depend on abstraction, which makes the system more flexible and adaptable to changes.

**Dependency Injection Design Pattern (DI)**

It is the design pattern that implements the DIP by providing dependencies from an external source rather than having classes creates them internally. It promotes modular and decoupled architecture.

**Benefits**

* **Loose Coupling**: By injecting dependencies, classes no longer need to create instances of their dependencies, reducing direct coupling between components.
* **Testability**: Makes it easier to mock dependencies for unit testing, allowing isolated testing of each component.
* **Maintainability**: Changes in dependencies (e.g., swapping service implementation) can be managed easily without altering dependent classes.
* **Flexibility**: Enables easy configuration changes and extension of functionality without modifying the existing code structure.

**Dependency Inversion Principle (DIP) vs Dependency Injection Design Pattern.**

In software development, DI and DIP are related concepts but serve different purposes.

1. **Dependency Inversion Principle (DIP)**

It is one of the five SOLID Principles of Object-Oriented design, introduced by Robert C. Martin. The principle suggests that.

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

In simpler terms, DIP encourages decoupling the components of a system. Instead of having concrete implementations depend on each other, they both rely on abstractions (interfaces or abstract classes). This makes code more flexible and easier to maintain or change.

**Example of DIP violation**

public class ReportGenerator

{

private PdfReport \_pdfReport;

public ReportGenerator()

{

\_pdfReport = new PdfReport(); // Direct dependency on a low-level module

}

public void Generate()

{

\_pdfReport.Generate();

}

}

In the above example, the ReportGenerator class is tightly coupled to the PDFReport class. If you wanted to generate a report in another format. Like HTML, you would need to change the ReportGenerator class, violating DIP.

**DIP applied**public interface IReport

{

void Generate();

}

public class PdfReport : IReport

{

public void Generate()

{

// Generate PDF report

}

}

public class ReportGenerator

{

private IReport \_report;

public ReportGenerator(IReport report)

{

\_report = report; // Depend on abstraction (IReport), not implementation

}

public void Generate()

{

\_report.Generate();

}

}

Now, ReportGenerator can work with any kind of report (PDF, HTML, etc.) as long as it implements IReport interface.

1. **Dependency Injection (DI)**

It is a technique or design pattern used to implement the Dependency Inversion Principle. DI is about passing dependencies (usually as interfaces or abstract classes) into a class, rather than having the class instantiate those dependencies directly. There are three common types of dependency injection.

* **Constructor Injection**: Dependencies are provided through the constructor.
* **Property Injection**: Dependencies are provided through properties.
* **Method Injection**: Dependencies are provided through method parameters.

DI allows for greater flexibility and easier unit testing by decoupling classes from their dependencies.

**Example of DI constructor injection**

public class ReportGenerator

{

private IReport \_report;

// Dependency Injection via constructor

public ReportGenerator(IReport report)

{

\_report = report;

}

public void Generate()

{

\_report.Generate();

}

}

// Usage

ReportGenerator generator = new ReportGenerator(new PdfReport());

generator.Generate();

here, ReportGenerator does not instantiate a PDFReport directly. Instead, the PDFReport (or any other implementation of IReport) is passed in via the constructor. This decouples the ReportGenerator from specific implementations and makes it easier to swap out implementations, making testing or future changes simpler.

**Key Differences**

* **DIP**  is a design principle that encourages reliance on abstractions rather than concrete implementations.
* **DI** is a technique or design pattern used to implement the DIP. It injects dependencies into a class, rather than having the class instantiate them itself.

In summary, DIP is about design philosophy (decoupling), while Di is the practical method for achieving that decoupling.

**Solid Principles Summary**

SOLID is an acronym for five design principles that help developers create more maintainable, understandable, and flexible software. These principles promote better software design and improve scalability and ease of testing.

1. **Single Responsibility Principle (SRP)**: A class should have only one reason to change, meaning it should have one responsibility.

**Benefit**: Enhances code readability and maintainability by reducing complexity. It also makes code easier to refactor.

1. **Open/Closed Principle (OCP)**: Software entities should be open for extension but closed for modification.

**Benefit**: Makes system more flexible and adaptable to change without altering existing code, which reduces the risk of introducing bugs.

1. **Liskov Substitution Principle (LSP)**: Objects of superclass should be replaced with objects of a subclass without affecting program correctness.

**Benefits**: Encourages the use of polymorphism, making the code more reusable and reducing the need for duplicated logic.

1. **Interface Segregation Principle (ISP)**: Clients should not be forced to depend on interfaces they do not use.

**Benefits**: Prevents “fat” interfaces and promotes a decoupled, modular design. It improves code maintainability and allows for flexible changes.

1. **Dependency Inversion Principle (DIP)**: High level modules should not depend on low level modules. Both should depend on abstractions.

**Benefits**: Promotes loose coupling between components, making the system easier to change and maintain. It also enhances testability by allowing easier mock or stub replacement for dependencies.