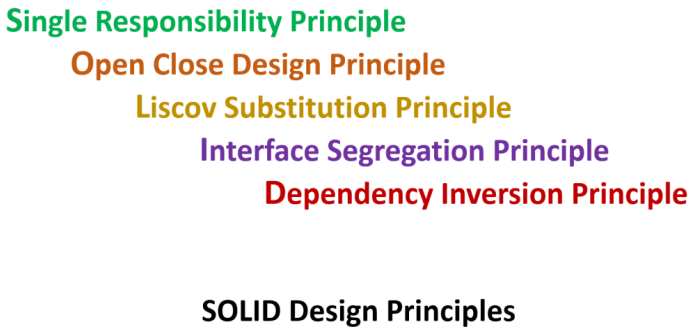
Why Software Design Principles?

Software Design principles are a set of guidelines to handle complexity and reduce the effort needed to develop a good system design. Design principles ease problems of future development, enhancement, maintenance and also reduce the scope of error during design.



What are Software Design Principles?

These are around 12 Software Design Key Principles but most important is **SOLID Software Design principles**. which use in all software designing.

1. **Single Responsibility Principle:** One class should do one thing and do it well.
2. **Open Close Design Principle:** Open for extension, close for modification.
3. **Liscov Substitution Principle:** Subtype must be a substitute for supertype.
4. **Interface Segregation Principle:** avoid monolithic interface, reduce pain on the client-side.
5. **Dependency Inversion Principle:** Don't ask lets framework give it to you.

1: Single Responsibility Principle (SRP)

A class should have only one reason to change. If there is any other reason to create another class.

This principle is completely based on **Coupling** and **Cohesion**. This principle states that your software design classes should in such a way that each class should have a single purpose/responsibility/functionality.

While designing software if you put more than one functionality in a single class then increase **coupling** between functionalities. If change required in one functionality there are chances to broke other functionality and required more testing to avoid such surprises in the production environment.

Responsibility Examples

Suppose you are having JPA classes as SavingAccountRepository.java and CurrentAccountRepository.java then SavingAccountRepository.java class should have only methods and queries related with Saving Accounts. It means your class should specialize in a single purpose.

Others most common example of responsibilities:

- Logging
- Formatting
- Validation
- Notification
- Error Handling
- Parsing
- Caching
- Mapping
- Class Section/ Instantiation etc.

Benefits

- This principle makes your software easier to implement and prevent unexpected side-effects of future changes.
- Your class will change only if anything will change in respected responsibility.
- Need to update dependencies and compile when some respected dependencies change.
- Reduce coupling between software and components.

The **Single Responsibility Principle (SRP)** also provides other benefits with classes, components, and microservices with single responsibility to make your code easier to explain, understand, implement. It also improves development speed and easier to track bugs.

2: Open-Closed Principle (OCP)

Software entities like classes, modules, and functions should be open for extension (new functionality) and closed for modification.

This principle is based on **inheritance** or **composition** design patterns like **Strategy Design pattern**. As per this principle:

- **"Open**" means, Your code should be able to extend existing code in order to introduce new functionality.

- **"Close"** means, Once your module has been developed and tested, the code will change only when correct bugs.

For Example, A BankAccount.java base class contains all basic transaction-related properties and methods. The same class can be extended by SavingAccount.java and CurrentAccount.java to handle saving and current account functionalities. If new functionalities need to add then only modification required in BankAccount.java class. Hence this class is open for extension and close for modification.

Benefits

The main benefit of the **Open/Close Design principle** is that already developed and tested code will not be modified and don't break.

3: Liscov Substitution Principle (LSP)

The driven type must be completely substituted for their base type.

Liscov Substitution Principle is closely related to the **Single Responsibility Principle** and **Interface Segregation Principle**.

This principle states that Subclasses or derived classes should be completely substituted by the superclass. The Subclass should enhance the functionality but not reduce it.

In other words, functions that use pointers and reference of base classes must be able to use objects derived classes without knowing it.

Example

Suppose Employee class extends Person Class by inheritance. In this way wherever you are using person class should also be able to use Employee class because Employee is a subclass of Person class.

Benefits

- If this principle violates then so much extra conditional code of type checking and duplicate code need to write throughout the application that can cause bugs when the application grows.
- In methods or functions which use the superclass type must work with the object of subclass without any issue.

4: Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods in interfaces that they don't use.

This principle states that a client should not implement an interface if it doesn't use that. An interface should belong to clients, not to the library or hierarchy and keep only those methods as required for the client.

Example

In an application service interface exposed to the client should have only those methods that are related to the client.

Benefits

If you violate this principle and add some more methods in an interface that are not used with the client then if you change anything in interface definitely some functionality of the client will break.

5: Dependency Inversion Principle (DIP)

The high-level module should not depend on low-level module, both should depend on abstractions. Abstraction should not depend on detail. detail should depend on abstraction.

This principle provides **loose coupling** between the dependencies of modules and classes.

The Dependency Inversion Principle states that:

High-level modules should not depend on low-level modules directly, both should depend on abstractions. This abstraction should not depend on details, details should depend on abstractions.

Examples

The best example of the **Dependency Inversion Principle** is the Spring framework where dependencies are injected through XML or Annotation which provide abstractions and can change without modification on implementation.