# **Day 21 - 26 July 2025**

**Document Name:**Day 21 - hmuvvala@ - Hari Gopal Muvvala

### SOLID Principles – Summary Notes

#### 1. S — Single Responsibility Principle (SRP)

A class should have **only one reason to change**.

* Each class should focus on **one task or functionality**.
* Helps in better **modularity and maintainability**.

✅ Split classes based on **separation of concerns**.

#### 2. O — Open/Closed Principle (OCP)

Software entities should be **open for extension, but closed for modification**.

* New features should be added by **extending**, not changing existing code.
* Promotes use of **abstraction, inheritance, and polymorphism**.

✅ Prefer **interfaces and abstract classes** for future extension.

#### 3. L — Liskov Substitution Principle (LSP)

Subtypes must be **substitutable for their base types**.

* Derived classes should not **break the behavior** of the base class.
* Ensures **polymorphism works safely**.

✅ Avoid designing parent classes with methods that don’t apply to all children.

#### 4. I — Interface Segregation Principle (ISP)

No class should be **forced to implement** methods it does not use.

* Create **smaller, role-specific interfaces**.
* Promotes **cleaner contracts** between interfaces and implementers.

✅ Split large interfaces into **cohesive, meaningful parts**.

#### 5. D — Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules.

Both should depend on **abstractions**.

* Avoid tight coupling between classes.
* Use **interfaces or abstract classes** to inject dependencies.

✅ Use **constructor injection or interfaces** to manage dependencies.

### 1. Liskov Substitution Principle (LSP)

**Definition:**

If class B is a subclass of class A, then objects of type A in a program should be replaceable with objects of type B without altering the correctness or behavior of the program.

**Why it matters:**

LSP ensures that inheritance is used properly. A subclass should never contradict or weaken the promises made by the superclass.

**Violation Example:**

If we have a class Bird with a method fly(), and subclasses Eagle and Ostrich, the Ostrich can’t fly. Forcing Ostrich to implement fly() breaks the LSP. Anyone calling fly() on a Bird object expects the bird to fly, but Ostrich violates that expectation.

**Fix:**

Separate behavior into more meaningful abstractions like:

* FlyingBird (with method fly())
* NonFlyingBird (with method walk())

Now only the appropriate subclass implements the relevant interface.

### 2. Interface Segregation Principle (ISP)

**Definition:**

A class should not be forced to implement interfaces it does not use. Instead of one fat interface, many small interfaces are preferred.

**Why it matters:**

Large interfaces become burdensome when implemented by classes that only need part of the behavior.

**Violation Example:**

An interface Shape has two methods: calculateArea() and calculateVolume(). A Circle has no volume but is forced to implement calculateVolume() — which it might implement using a dummy body or by throwing an exception.

**Fix:**

Break the interface:

* ICalcArea – for shapes that calculate area
* ICalcVolume – for 3D shapes that calculate volume

Now each class implements only what it needs. No unnecessary code or fake logic is required.

### 3. Dependency Inversion Principle (DIP)

**Definition:**

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

Also, abstractions should not depend on details; details should depend on abstractions.

**Why it matters:**

Tightly coupled systems are hard to maintain, extend, and test. DIP allows flexible, testable, and scalable systems by introducing abstraction (like interfaces).

**Violation Example:**

Class Cupboard directly depends on a Clothes class. Later, if we want to add Books to the cupboard, we must modify the Cupboard class. That’s tight coupling and violates OCP too.

**Fix:**

Create an interface IProduct with common methods like getSample() and seeReview().

Now, both Books and Clothes implement IProduct, and Cupboard can store any IProduct.

This makes Cupboard extensible without needing changes.

### 4. Real-World ISP Example – Payment System

**Structure:**

* Common interface: Payment (methods like status(), getListOfPayment())
* Bank-specific interface: Bank (methods like initiatePayment(), stopPayment())
* Loan-specific interface: Loan (methods like initiateRepayment(), initiateFinalSettlement())

**Implementation:**

* BankPayment class implements Payment and Bank
* LoanPayment class implements Payment and Loan

**Benefits:**

* Each class only implements what it needs.
* Promotes interface segregation and avoids forcing irrelevant methods on unrelated classes.
* Clean design that scales well in real systems.

### 5. Java Generics – Key Concepts

#### **Why Generics?**

* Helps write type-safe, reusable code.
* Removes need for casting.
* Improves code clarity and reduces runtime errors.

#### **Invariance in Generics:**

* List<Cat> is **not** a subtype of List<Animal>, even if Cat extends Animal.
* This is to avoid potential runtime type safety issues.

### 6. Wildcard Types in Java

#### **<?>**

##### – Unbounded Wildcard

* Accepts list of **any** type.
* Use when the operation is independent of the actual type.
* Mostly used for **read-only** operations (e.g., printing a list).

#### **<? extends T>**

##### – Upper Bounded Wildcard

* Accepts T and all its **subtypes**.
* You can **read safely**, but **can’t add** elements.
* Best for **reading/consuming** from a collection.
* Example: List<? extends Animal> accepts List<Dog>, List<Cat>, etc.

#### **<? super T>**

##### – Lower Bounded Wildcard

* Accepts T and all its **supertypes**.
* You can **add** T to the list, but when reading, it’s treated as Object.
* Best for **writing** to a collection.
* Example: List<? super Cat> allows adding Cat or subclasses but reading requires casting.

#### Wildcard Summary Table

| **Wildcard** | **Accepts** | **Can Add** | **Can Read** | **Use Case** |
| --- | --- | --- | --- | --- |
| <?> | Any type | No | Yes (Object) | Read anything |
| <? extends T> | T and all subclasses of T | No | Yes (as T) | Read-only lists |
| <? super T> | T and all superclasses of T | Yes (T) | Yes (Object) | Write into list |

### 7. Encapsulation and Loose Coupling

#### **Encapsulation:**

* Keeping data (fields) private and exposing access via getters and setters.
* Helps enforce validation rules and prevents misuse of object state.
* Promotes object safety and internal control.

#### **Loose Coupling:**

* Objects interact via interfaces or methods, not by accessing internal variables directly.
* Makes code modular and testable.
* For example, a LooseCoupling class accepts a Student object and prints the roll number using getRollNo(), instead of accessing rollNo directly.

### Key Takeaways from Day 21

* SOLID principles guide clean, maintainable object-oriented design.
* LSP helps ensure safe inheritance.
* ISP promotes small, purpose-driven interfaces.
* DIP decouples high-level logic from low-level implementation using abstractions.
* Java Generics adds type safety and flexibility.
* Wildcards allow flexible, constrained access to generic types.
* Encapsulation and loose coupling improve structure and maintainability in any project.

### Class Relationships in UML

UML helps visualize how classes are connected in a software system. Below are three key types of relationships:

#### 1. Dependency

* **Definition**: A class *uses* another class temporarily.
* **Example**: Car uses Engine in a method.
* **Effect**: No long-term association; the object is created inside a method.

#### 2. Aggregation (Has-A)

* **Definition**: A class has a reference to another class.
* **Example**: Car has a Driver.
* **Effect**: Associated object (Driver) can exist even if Car is destroyed.
* **Symbol in UML**: Empty diamond (◊)

#### 3. Composition

* **Definition**: A class *owns* another class strongly.
* **Example**: Car creates and owns its Wheels.
* **Effect**: If Car is destroyed, Wheels are also destroyed.
* **Symbol in UML**: Filled diamond (◆)

#### Why Use These Relationships?

* To represent **real-world associations** in software.
* To guide proper use of **abstraction** and **design principles**.
* Helps enforce concepts like **reusability**, **modularity**, and **encapsulation**.

### UML Structural vs Behavioral Diagrams

| **Aspect** | **Structural Diagrams** | **Behavioral Diagrams** |
| --- | --- | --- |
| Focus | What the system *is* | What the system *does* |
| Represents | Static structure of classes/objects | Dynamic behavior/interactions over time |
| Examples | Class Diagram, Object Diagram, Component | Use Case, Sequence, Activity, State diagrams |

#### Types of Behavioral Diagrams

| **Diagram Name** | **Use / Description** |
| --- | --- |
| **Use Case Diagram** | Shows system features from user perspective |
| **Sequence Diagram** | Object interactions in time sequence |
| **Activity Diagram** | Workflow like flowcharts |
| **State Diagram** | Object states and transitions |
| **Communication Diagram** | Similar to sequence but focuses on object links |
| **Interaction Overview** | Combines activities and sequence flows |
| **Timing Diagram** | Emphasizes time constraints on object behaviors |

#### Benefits of UML

* Visualize system before coding
* Enhances communication across teams
* Supports clean requirement gathering
* Aids maintenance and scaling
* Promotes reusable, modular code
* Useful documentation for onboarding or review
* Tool support: IntelliJ plugin, draw.io, StarUML, PlantUML, etc.