Lecture 6 : SOLID Principles Part-2

1. Recap of SOLID Principles

Before diving into the remaining two principles (Interface Segregation and Dependency Inversion), a quick recap:

1. **Single Responsibility Principle (SRP)**
   • A class should have only one reason to change—i.e., one responsibility.
2. **Open/Closed Principle (OCP)**
   • Software entities (classes, modules, functions) should be open for extension but closed for modification.
3. **Liskov Substitution Principle (LSP)**
   • Subtypes must be substitutable for their base types without altering the correctness of the program.

We have already covered SRP, OCP, and LSP conceptually. What follows is a **detailed breakdown of LSP guidelines**, then full explanations of **Interface Segregation Principle (ISP)** and **Dependency Inversion Principle (DIP)** with illustrative examples.

2. Deep Dive: Liskov Substitution Principle (LSP)

> **Definition:** *"Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program."*

2.1 Why LSP "Breaks" Often

- Inheritance ensures that subclasses have the same methods, but not necessarily the same behavior or contractual guarantees.
- Without clear rules, a subclass may override a method incorrectly (e.g., throwing unexpected exceptions, changing return values or method signatures), causing client code to fail.

2.2 Three Categories of LSP Rules

LSP compliance hinges on three broad categories of rules, each with sub-rules:

1. **Signature Rules**
2. **Property Rules**
3. **Method Rules**

2.3 Signature Rules

Ensure that method overrides preserve the *contractual interface* of the parent:

1. **Method Argument Rule**
   - The overridden method in the subclass must accept the same argument types as the parent, or *wider* (a “broader” type up the inheritance chain).
   - *Example:* If the parent method takes a `String`, the child override must also take `String` (or a supertype, e.g., `Object`), never an unrelated type like `Integer`.
2. **Return Type Rule**
   - The subclass’s return type must be the same as the parent’s, or *narrower* (a subtype).
   - *Covariant returns* are allowed (e.g., parent returns `Animal`; child can return `Dog`), but not contravariant (e.g., child cannot return `Object` if the parent returns `Animal`).
3. **Exception Rule**
   - The subclass may throw fewer or more specific exceptions than the parent, but never broader exceptions that the client is not expecting.
   - *Example:* If the parent method declares it throws `RuntimeError`, the child can throw `IndexOutOfBoundsException` (a subtype) but not a totally unrelated exception like `OutOfMemoryError` if it isn’t within that hierarchy.

2.4 Property Rules

Ensure that the subclass preserves key “properties” of the parent class:

1. **Class Invariant**
   - Any invariant (a condition that must always hold true) specified on the parent must not be violated by the subclass.
   - *Example:* A `BankAccount` class may mandate that `balance >= 0`. A subclass `CheatAccount` that allows negative balances breaks this invariant and thus violates LSP.
2. **History Constraint**
   - The subclass must preserve the “history” or lifecycle behavior of the parent. It cannot remove or disable operations that clients expect to always work.
   - *Example:* A `FixedDepositAccount` (subclass) that throws an exception on every withdrawal violates the parent’s guarantee that withdrawal is always allowed.

2.5 Method Rules

Ensure that method-specific preconditions and postconditions remain consistent:

1. **Precondition (Method Rule – Before Execution)**
   - Preconditions specify what must be true *before* a method executes.
   - A subclass may *weaken* (make less strict) the precondition (accept a broader range of inputs), but must not *strengthen* it (require more than the parent).
   - *Example:* Parent requires `0 ≤ x ≤ 5`; child can accept `0 ≤ x ≤ 10` (weaker), but not `0 ≤ x ≤ 3` (stronger), or clients that supply `x = 7` would fail.
2. **Postcondition (Method Rule – After Execution)**
   - Postconditions specify what must be true *after* a method completes.
   - A subclass may *strengthen* the postcondition (guarantee more), but must not *weaken* it (guarantee less).
   - *Example:* Parent `brake()` method guarantees "speed decreases"; a subclass `HybridCar` may also increase battery charge (strengthening), but must never leave speed unchanged or increased (weakening).

2.6 Key Takeaways for LSP

- Always check whether a subclass truly *behaves* like its parent, not just whether it *compiles*.
- Remember: **Signature**, **Property**, and **Method** rules each have clearly defined sub-rules—use these as a checklist when designing hierarchies.
- Violations often manifest as unexpected exceptions, incorrect return values, or broken invariants.

---

3. Interface Segregation Principle (ISP)

Definition: *"Clients should not be forced to depend on interfaces they do not use."*
Key Idea: It's better to have many small, client-specific interfaces than one large, general-purpose interface.

3.1 The Problem with "Fat" Interfaces

- A single interface/class that includes every conceivable method (e.g., both 2D and 3D shape operations) forces some implementers to override methods they don't need.
- Unneeded methods often either throw exceptions or remain unimplemented, hurting maintainability and violating SRP.

3.2 Illustrative Example: Shapes

“Fat” Interface Approach

```
// See Code for example
```

1. **Problem:** `Square` and `Rectangle` are forced to implement `volume()`, leading to stubs or exceptions.

3.3 ISP Solution: Segregate into Two Interfaces

2DShape

```
class TwoDShape {
  double area();
}
class Square :public TwoDShape { ... }
class Rectangle : public TwoDShape { ... }
```

3DShape

```
class ThreeDShape {
public:
    virtual double area() = 0;
    virtual double volume() = 0;
};

class Cube : public ThreeDShape {
    // ...
};
```

Benefits:

- Each implementer only deals with methods it actually uses.
- Code is cleaner, adheres to SRP, and avoids unnecessary stubs or exceptions.

---

4. Dependency Inversion Principle (DIP)

Definition:

1. High-level modules should not depend on low-level modules; both should depend on abstractions.
2. Abstractions should not depend on details; details should depend on abstractions.

4.1 The Problem with Direct Coupling

- A high-level class (e.g., `UserService`) that directly calls concrete low-level classes (`SqlDatabase`, `MongoDatabase`) becomes tightly coupled.
- Changing the low-level implementation (e.g., swapping MongoDB for Cassandra) forces modifications in the high-level class—violating OCP.

4.2 DIP Solution: Introduce an Abstraction Layer

Define an Abstraction

```
class Persistence {
public:
  virtual void save(const User& u) = 0;
};
```

Make Low-Level Classes Depend on the Abstraction

```
class SqlDatabase : public Persistence { ... override save(...) ... }
class MongoDatabase : public Persistence { ... override save(...) ... }
```

High-Level Module Depends Only on the Abstraction

```
class UserService {
private:
  Persistence* db;   // injected dependency
public:
  UserService(Persistence* p) : db(p) { }
  void storeUser(const User& u) { db->save(u); }
};
```

Dependency Injection

  - At runtime, instantiate `UserService` with either `new SqlDatabase(…)` or `new MongoDatabase(…)` (or a future `CassandraDatabase`), without changing `UserService` itself.

4.3 Real-World Analogy

- A company CEO (high-level) doesn't instruct individual developers (low-level) directly. Instead, a manager (abstraction) relays requirements.
- The CEO depends only on the manager's interface; developers depend on the manager for directives. Swapping out developers doesn't affect the CEO's workflow.

---

5. Final Thoughts & Trade-Offs

- **SOLID principles are guidelines, not hard laws.** In practice, business requirements and performance constraints may necessitate trade-offs.
- Adhering to these principles generally leads to more **maintainable**, **scalable**, and **extensible** code—but balance is key.
- Whenever you find yourself violating one principle, check whether it's in service of a higher-priority need (e.g., performance) and document your reasoning.

By following these LSP guidelines and applying ISP and DIP judiciously, you'll write cleaner, more robust object-oriented code that stands the test of evolving requirements.