SOLID Principles 2

Chapter: Deep Dive into SOLID Design Principles — Liskov Substitution, Interface Segregation, and Dependency Inversion

Introduction

- This chapter continues the exploration of **SOLID design principles**, focusing on the last two principles: the **Interface Segregation Principle (ISP)** and the **Dependency Inversion Principle (DIP)**, while concluding a detailed discussion on the **Liskov Substitution Principle (LSP)**.
- These principles are essential for writing **maintainable**, **extensible**, and **robust** code, especially relevant in **object-oriented programming (OOP)** and **software design**.
- The chapter revisits important vocabulary such as **method signature**, **class invariant**, **precondition**, and **postcondition**, emphasizing their role in enforcing LSP and ensuring clean inheritance hierarchies.
- Proper adherence to these principles prevents common pitfalls such as broken inheritance substitutability, tightly coupled modules, and bloated interfaces.
- Understanding these concepts is crucial both for real-world software development and technical interviews.

---

1. The Liskov Substitution Principle (LSP) — A Refresher and Advanced Guidelines

- Previously, we covered the basics of LSP: A **child class** should be fully substitutable for its **parent class** without breaking client code.
- The principle states that if a client expects an object of type A, it should be able to accept an object of type B (a subclass of A) without any issues.
- However, LSP is often **violated unintentionally** because the child class might override methods incorrectly or change behavior in a way that breaks expectations.

- Three fundamental rules ensure LSP compliance:
  - **Signature Rule**: Method signatures (method name, argument types, and return type) in the subclass must be the same or more general (broader) than in the parent class.
  - **Property Rule (Class Invariant and History Constraint)**: The subclass must maintain all invariants of the parent class and respect its behavioral history; it cannot weaken or violate these established rules.
  - **Method Rule (Precondition and Postcondition)**: Preconditions in subclasses can be weakened (less strict), but never strengthened; postconditions can be strengthened but not weakened.
- These rules act as **guardrails** to avoid subtle bugs in inheritance that cause client code to fail unexpectedly.

Signature Rule

- The method's signature includes its name, parameters, and return type.
- When overriding methods, the subclass must accept the **same or broader parameter types** (e.g., accepting a parent class type instead of a more specific child type) and return the **same or narrower type** (covariance allowed).
- Example: If a parent method accepts a `String`, the overridden method cannot accept an `Integer` because the client expects a `String`.
- This rule is **language-independent** and enforced by most modern OOP languages like C++ and Java via compiler checks.
- Violating this rule causes compilation errors or runtime failures.

Return Type Rule

- The return type in the subclass method must be the same or a subtype (narrower) of the parent method's return type to maintain substitutability.
- Example: A method returning an `Animal` in the parent class can be overridden to return a `Dog` (a subtype), but not a broader type like `Organism`.
- This is known as **covariance** in return types and ensures that clients can safely use subclass objects wherever parent class objects are expected.

Exception Rule

- Exceptions thrown by overridden methods in subclasses must be the same or narrower in scope compared to the parent.
- If the parent method throws a `RuntimeError`, the subclass method can throw `OutOfRangeError` (a subtype) but not a broader or unrelated exception.
- Clients depend on the contract specifying which exceptions to handle, so violating this causes unhandled exceptions and program crashes.
- Example from code: Throwing a `RuntimeError` exception in the subclass when the parent throws a more specific `OutOfRangeError` leads to unhandled exceptions in client code.

---

2. Property Rule — Class Invariants and History Constraints

- **Class Invariant** is a condition that always holds true for an object of a class during its lifetime.
- Subclasses must maintain or strengthen this invariant but must never weaken it.
- Example: A `BankAccount` class has an invariant that **balance must never be negative**.
- A `CheatAccount` subclass violates this by allowing the balance to go negative, breaking LSP and causing client code to malfunction.
- **History Constraint** ensures that the subclass respects the behavioral history of the parent class.
- It means if the parent class allows certain operations (e.g., withdrawal), the subclass should not disallow or throw exceptions for those operations.
- Example: A `FixedDepositAccount` subclass that disallows withdrawal by throwing exceptions violates this constraint.
- This breaks substitutability and leads to failures when clients expect withdrawal to be possible.

Immutable Classes and Methods

- An immutable class or method cannot be changed or overridden to behave differently.
- If the parent declares a method as immutable (e.g., via C++ `final`), subclasses must not override it with mutable versions as this breaks history constraints.

---

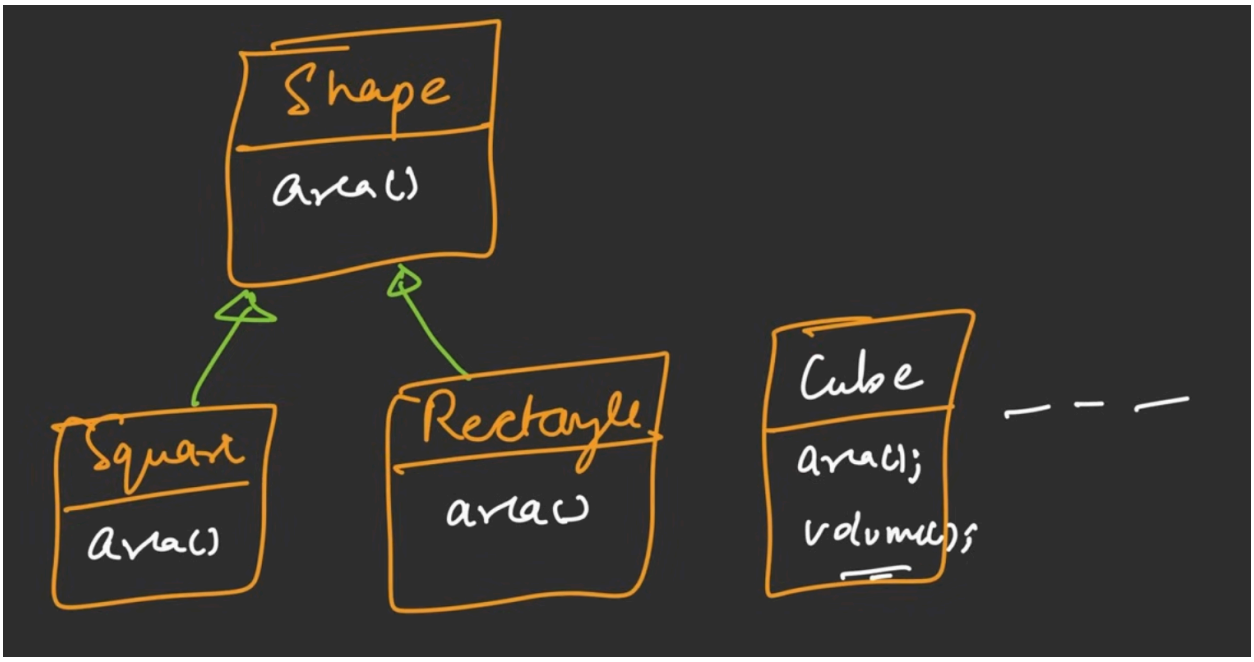
3. Method Rule — Preconditions and Postconditions

- **Preconditions** are conditions that must be true before a method executes.
- Subclasses may **weaken** preconditions (allow more inputs) but cannot strengthen them (become more restrictive).
- Example: A method in the parent class accepts numbers between 0 and 5; the subclass can accept 0 to 10 but not restrict it further (e.g., only 0 to 3).
- This flexibility ensures the subclass can handle all inputs that the parent can, guaranteeing substitutability.
- **Postconditions** are conditions that must be true after the method executes.
- Subclasses may **strengthen** postconditions (guarantee more) but cannot weaken them.
- Example: A parent `Car` class method `brake()` guarantees the car slows down; an electric car subclass can guarantee the car slows down **and** the battery charges (strengthening postconditions), but cannot fail to slow down.
- Violating this rule leads to client code behaving incorrectly or dangerously.

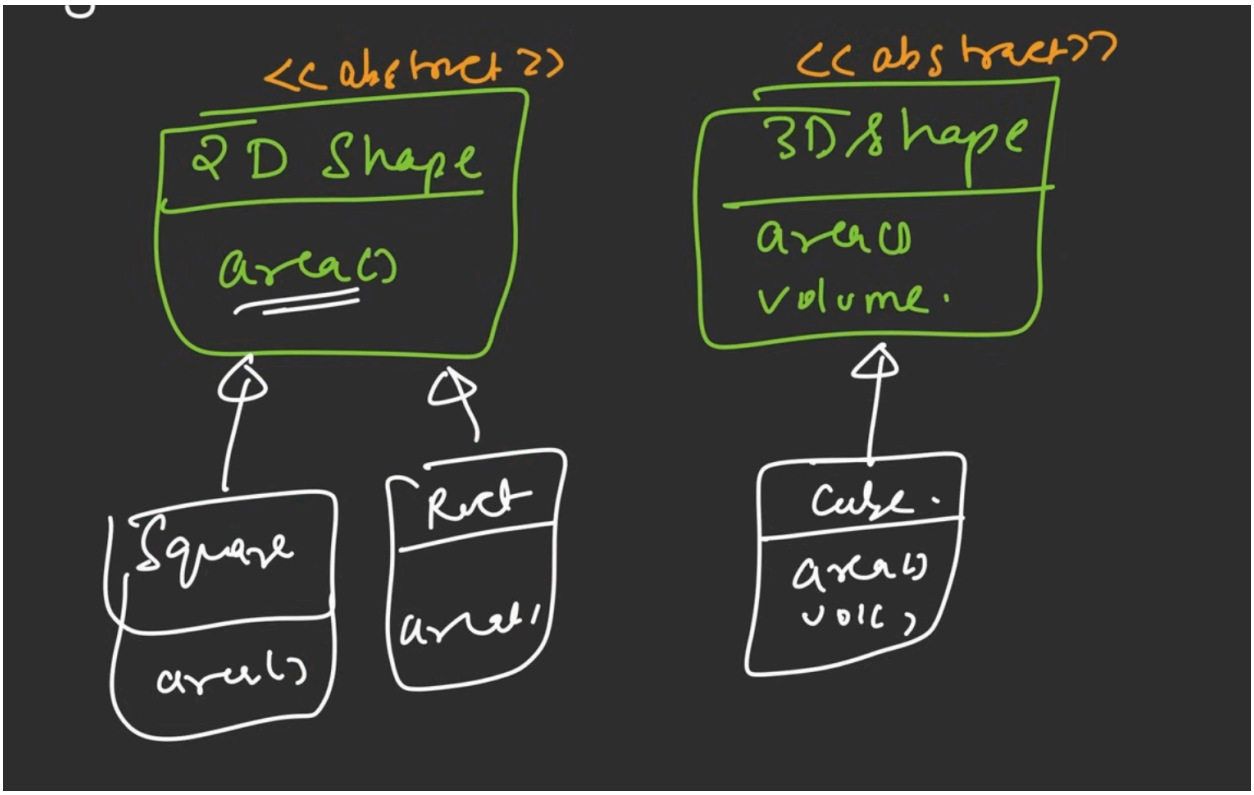
---

4. Interface Segregation Principle (ISP)

- ISP states that **many client-specific interfaces are better than one general-purpose interface**.
- Forcing subclasses to implement methods they do not use leads to unnecessary complexity and broken LSP.

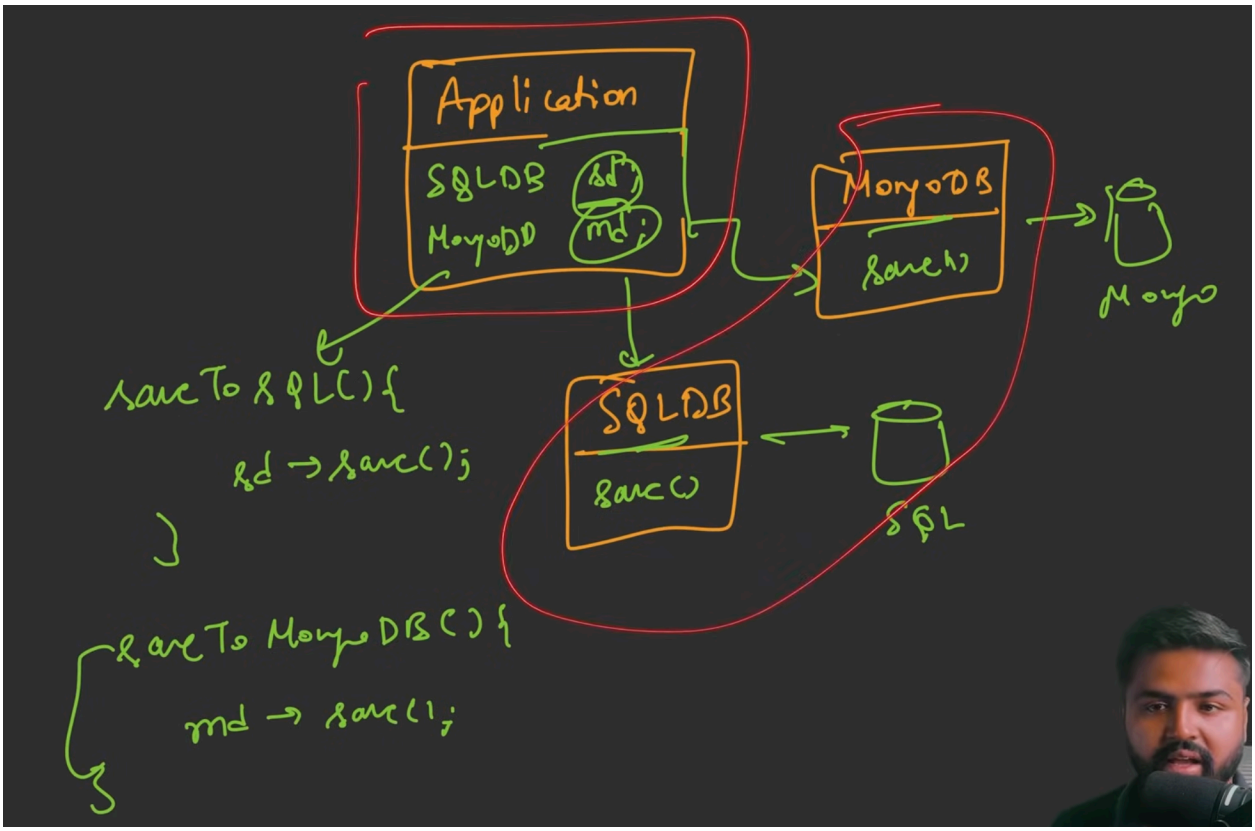


- Example: A `Shape` interface with both `area()` and `volume()` methods forces 2D shapes like `Square` and `Rectangle` to implement irrelevant `volume()` methods, often throwing exceptions or dummy implementations.
- The solution is to **split interfaces** into more specific ones:
  - A `TwoDShape` interface with `area()`
  - A `ThreeDShape` interface with `area()` and `volume()`
- This segregation allows each subclass to implement only the methods it requires, keeping interfaces clean and focused.

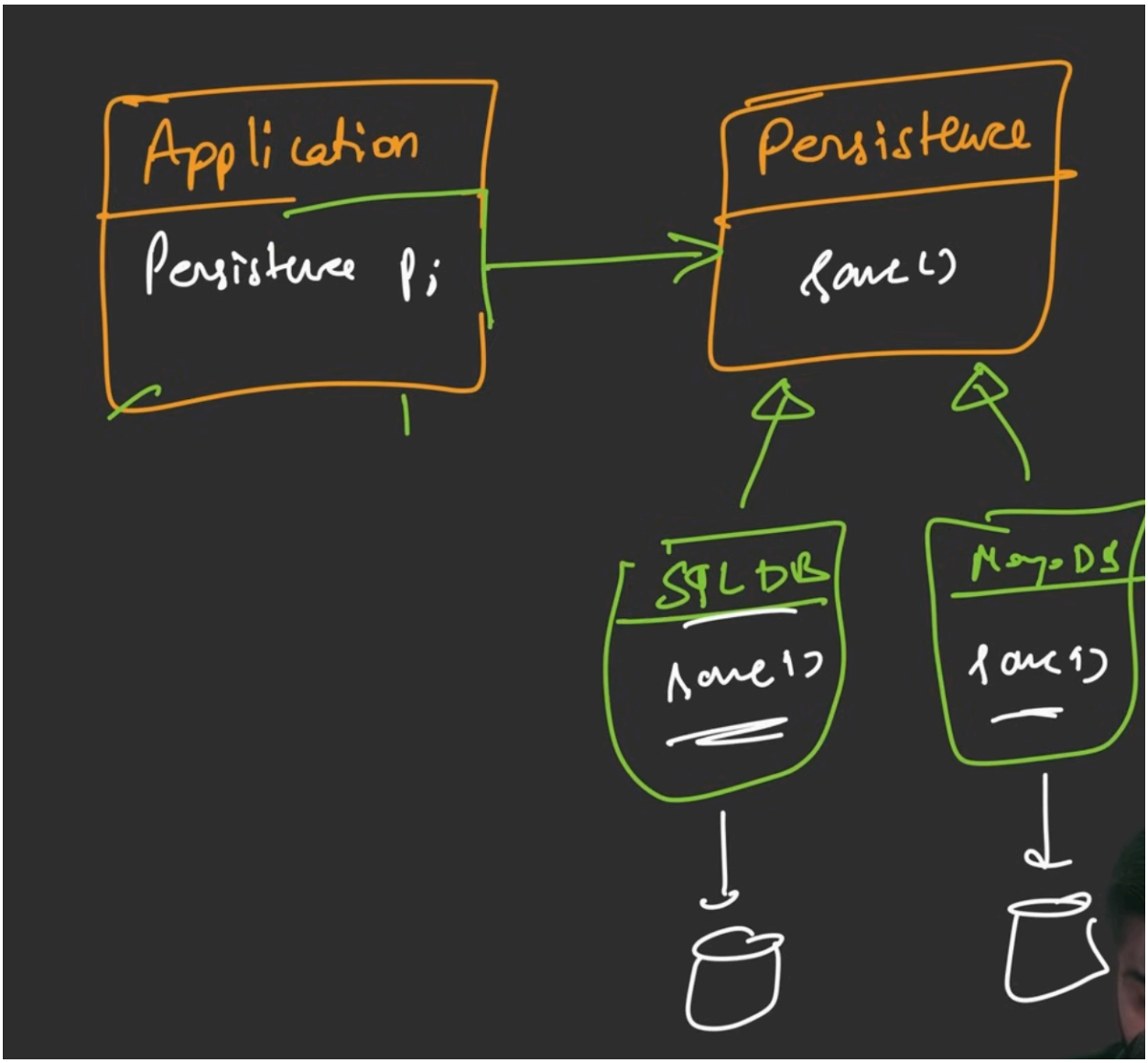


5. Dependency Inversion Principle (DIP)

- DIP dictates that **high-level modules should not depend on low-level modules directly**; both should depend on abstractions (interfaces or abstract classes).
- This principle promotes loose coupling and flexibility in code architecture.
- Example: An application (high-level module) interacting directly with **MongoDB** and **SQL Database** (low-level modules) leads to tight coupling.
- If a new database (e.g., Cassandra) is introduced, changes are needed in the application, violating the **Open-Closed Principle**.



- Solution: Introduce an **abstraction layer** (e.g., `Persistence` interface) between the application and database classes.
- The application depends only on this interface and not on concrete database classes.
- Concrete database classes implement this interface and override methods like `save()`.
- This allows swapping databases without modifying the application, enabling **dependency injection** and maintaining SOLID principles.



Real-World Analogy

- A CEO (high-level) does not communicate directly with developers (low-level) but through managers (abstraction layer).
- If developers change, the CEO remains unaffected, similar to how abstractions decouple high-level and low-level modules.

---

6. Understanding Real-World Complexity and SOLID Principles in Practice

- Real-world objects (e.g., humans/users) are complex and perform many behaviors.
- SOLID principles encourage modeling only the **relevant subset of behaviors** for a specific application scenario.
- For instance, a `User` object in a ride-booking app only exposes ride-related behaviors, whereas in a food delivery app, it exposes order-related behaviors.
- This shows that **single responsibility** and other SOLID principles apply contextually, based on application needs and scenarios.
- The chapter emphasizes that these principles are **guidelines, not rigid laws**.
- Practical codebases often require trade-offs, balancing ideal principles with business logic demands and complexity.
- Recognizing when and how to relax these principles is a key skill in software design.

Conclusion

- This chapter thoroughly examines the **Liskov Substitution Principle**, detailing its three main rules: signature, property, and method rules, with clear examples and code demonstrations.
- Following LSP strictly ensures subclasses are true substitutes for parent classes, promoting code robustness and avoiding runtime failures.
- The **Interface Segregation Principle** encourages creating focused interfaces tailored to specific client needs, preventing bloated interfaces and unnecessary implementations.
- The **Dependency Inversion Principle** fosters decoupled architectures by having high- and low-level modules depend on abstractions, facilitating easier maintenance and scalability.
- Together, these SOLID principles guide developers toward writing cleaner, more modular, and maintainable code.
- However, these principles are not absolute laws but flexible guidelines that must be balanced with real-world business requirements and complexities.

- Mastery of these concepts enables confident design decisions, improved code quality, and readiness for professional software development challenges.

Bullet-Point Summary

Liskov Substitution Principle (LSP):

- Child classes must be substitutable for parent classes without affecting client code.
- Signature Rule: Method signatures must be identical or compatible (parameters broader or equal, return types same or narrower).
- Property Rule: Class invariants and history constraints must be maintained or strengthened, never weakened.
- Method Rule: Preconditions can be weakened; postconditions can be strengthened, but not vice versa.
- Violations lead to runtime errors, unhandled exceptions, and broken substitutability.

Interface Segregation Principle (ISP):

- Prefer multiple client-specific interfaces over one general interface.
- Avoid forcing classes to implement unused methods.
- Split interfaces logically (e.g., 2D vs 3D shapes) to simplify implementation and maintain clarity.

Dependency Inversion Principle (DIP):

- High-level modules should depend on abstractions, not concrete low-level modules.
- Use interfaces/abstract classes to decouple application from implementation details (e.g., database types).
- Introduce abstraction layers and employ dependency injection to improve flexibility and maintainability.

Practical Insights:

- Real-world objects are complex and multifaceted, but software models focus on relevant behaviors per scenario.
- SOLID principles are guidelines, not strict laws — trade-offs may be necessary.
- Following these principles leads to cleaner, scalable, and extensible designs.

Real-World Examples:

- Bank account example illustrating class invariants and history constraints (balance never negative, withdrawal always allowed).
- Shape hierarchy demonstrating interface segregation (2D shapes don’t implement volume).
- Database access example illustrating dependency inversion (application uses persistence interface, not concrete database classes).
- Car braking example to explain preconditions and postconditions (braking always decreases speed).

This chapter provides a comprehensive understanding of advanced OOP design principles vital for creating scalable and maintainable software systems.