**LOW LEVEL DESIGN PROGRAMMING PRINCIPLE**

Low-Level Design (LLD) programming principles focus on how to structure and implement individual components, classes, and methods in a way that ensures **maintainability**, **scalability**, **testability**, and **robustness** of software. These principles are particularly useful for class design and object-oriented programming.

The core **programming principles** of LLD are often best explained using the **SOLID principles**, along with a few supporting design principles.

**🔥 Core Programming Principles in LLD (with Real-World Use Cases)**

**1. Single Responsibility Principle (SRP)**

**🧠 Definition:**  
A class should have only **one reason to change** — it should only have **one job or responsibility**.

**✅ Real-World Example:**  
A class that handles both user registration and email notification violates SRP. These are two separate concerns.

**🧑‍💻 Java Code:**

// BAD: Violation of SRP

class UserService {

void registerUser(String name) { /\* logic \*/ }

void sendWelcomeEmail(String name) { /\* logic \*/ }

}

// GOOD: SRP Applied

class UserService {

void registerUser(String name) { /\* logic \*/ }

}

class EmailService {

void sendWelcomeEmail(String name) { /\* logic \*/ }

}

**2. Open/Closed Principle (OCP)**

**🧠 Definition:**  
Software entities (classes, modules, functions) should be **open for extension**, but **closed for modification**.

**✅ Real-World Example:**  
Imagine a payment system where new payment types (like UPI, credit card, crypto) should be added without changing the core logic.

**🧑‍💻 Java Code:**

interface Payment {

void pay(double amount);

}

class CreditCardPayment implements Payment {

public void pay(double amount) { /\* logic \*/ }

}

class UpiPayment implements Payment {

public void pay(double amount) { /\* logic \*/ }

}

class PaymentProcessor {

void processPayment(Payment payment, double amount) {

payment.pay(amount); // Closed for modification, open for new payment types

}

}

**3. Liskov Substitution Principle (LSP)**

**🧠 Definition:**  
Objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program.

**✅ Real-World Example:**  
If you have a class Bird, and Penguin extends it, calling fly() on a Penguin would break the logic since penguins can't fly.

**🧑‍💻 Java Code:**

abstract class Bird {

abstract void eat();

}

class Sparrow extends Bird {

void eat() { System.out.println("Sparrow eats"); }

}

class Penguin extends Bird {

void eat() { System.out.println("Penguin eats"); }

}

**4. Interface Segregation Principle (ISP)**

**🧠 Definition:**  
Clients should not be forced to depend on interfaces they do not use.

**✅ Real-World Example:**  
If a printer interface includes methods for scanning and faxing, a basic printer shouldn't be forced to implement those.

**🧑‍💻 Java Code:**

// BAD: One fat interface

interface MultiFunctionPrinter {

void print();

void scan();

void fax();

}

// GOOD: Segregated interfaces

interface Printer {

void print();

}

interface Scanner {

void scan();

}

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

**🧠 Definition:**  
High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces).

**✅ Real-World Example:**  
A high-level OrderService shouldn’t depend directly on MySQLDatabase, but on a Database interface.

**🧑‍💻 Java Code:**

interface Database {

void save(String data);

}

class MySQLDatabase implements Database {

public void save(String data) { /\* Save to MySQL \*/ }

}

class OrderService {

Database db;

OrderService(Database db) {

this.db = db;

}

void placeOrder(String order) {

db.save(order);

}

}

**🛠 Additional Principles in LLD**

**6. DRY (Don’t Repeat Yourself)**

**🧠 Definition:**  
Avoid duplicating code. Reuse logic through functions or classes.

**✅ Use Case:** Utility functions like date formatting should be in a single utility class.

**7. KISS (Keep It Simple, Stupid)**

**🧠 Definition:**  
Prefer simple, readable solutions over complex ones unless complexity is required.

**✅ Use Case:** Avoid unnecessary design patterns unless they solve a specific problem.

**8. YAGNI (You Aren’t Gonna Need It)**

**🧠 Definition:**  
Don't implement something unless it is actually needed.

**✅ Use Case:** Avoid designing systems for hypothetical future requirements.

**9. Law of Demeter (LoD)**

**🧠 Definition:**  
A method should only interact with immediate friends (its own fields, method parameters, or locally created objects).

**✅ Use Case:** Avoid chaining too many method calls like order.getCustomer().getAddress().getCity().

**✅ Summary Table**

| **Principle** | **Focus** | **Real-World Example** |
| --- | --- | --- |
| SRP | One job per class | Separate email & registration logic |
| OCP | Extend, don't modify | Add new payments without changing code |
| LSP | Valid subclass usage | Penguin not extending flying bird |
| ISP | Split fat interfaces | BasicPrinter not needing scan/fax |
| DIP | Depend on interfaces | Inject Database interface into services |
| DRY | Avoid duplication | Use utility functions |
| KISS | Simplicity | Avoid unnecessary abstraction |
| YAGNI | Build only what's needed | Don't write unused features |
| Law of Demeter (LoD) | Minimize coupling | Avoid deep object chain calls |

Let me know if you want a mind map, UML diagram, or full project structure demonstrating all of these!

**💡 What is the DRY Principle?**

**🔹 DRY stands for "Don't Repeat Yourself"**

The **DRY Principle** states that **every piece of knowledge (logic, functionality, data)** should have **a single, unambiguous, authoritative representation** in the system.

It was introduced in the book **"The Pragmatic Programmer"** by **Andy Hunt and Dave Thomas**.

**✅ Why DRY is Important in Software Design**

| **Without DRY** | **With DRY** |
| --- | --- |
| Code duplication | Single source of truth |
| Difficult to update | Easy maintenance |
| Higher chance of bugs | Consistent behavior |
| Harder to test | Easier to test & refactor |

**🧠 Real Meaning**

Whenever you **copy-paste code**, or write the **same logic in multiple places**, you're **violating DRY**.

Instead, **abstract** repeated logic into:

* Functions
* Classes
* Modules
* Constants
* Configs
* Database tables

**🔧 Real-World Examples of DRY Principle**

**📌 Example 1: Billing Logic in E-Commerce App**

**❌ Violation of DRY**

class Checkout {

public double calculateTotal(double price, int quantity, double taxRate) {

return (price \* quantity) + (price \* quantity \* taxRate);

}

}

class AdminPanel {

public double calculateTotal(double price, int quantity, double taxRate) {

return (price \* quantity) + (price \* quantity \* taxRate);

}

}

✅ Logic is **duplicated**. If tax logic changes, you must update **both places**!

**✅ DRY Version**

class BillingUtil {

public static double calculateTotal(double price, int quantity, double taxRate) {

return (price \* quantity) + (price \* quantity \* taxRate);

}

}

// Used everywhere

BillingUtil.calculateTotal(p, q, t);

**📌 Example 2: Repeating SQL Queries**

**❌ Violation of DRY**

-- In one service

SELECT \* FROM users WHERE email = 'ajay@example.com';

-- In another module

SELECT \* FROM users WHERE email = 'ajay@example.com';

**✅ DRY Version**

Store the query logic in a **view** or **stored procedure**, or abstract in code:

UserRepository.findByEmail("ajay@example.com");

**📌 Example 3: Repeating Validation Logic**

**❌ Bad**

if (user.getEmail() == null || !user.getEmail().contains("@")) {

// email invalid

}

if (employee.getEmail() == null || !employee.getEmail().contains("@")) {

// email invalid

}

**✅ Good**

public class Validator {

public static boolean isValidEmail(String email) {

return email != null && email.contains("@");

}

}

// Usage

Validator.isValidEmail(user.getEmail());

Validator.isValidEmail(employee.getEmail());

**📌 Example 4: Hardcoded Strings/Constants**

**❌ Bad**

System.out.println("Welcome to BookStore");

System.out.println("Welcome to BookStore");

**✅ Good**

public static final String APP\_TITLE = "Welcome to BookStore";

System.out.println(APP\_TITLE);

**📌 Example 5: Repeating UI Components (e.g., Login Form)**

You have the **same login form** in Admin, User, and Guest dashboards.

**❌ Copy-pasted login HTML/JS three times**

**✅ Create a reusable component:**

* loginForm.js
* LoginFormComponent.java

Then reuse wherever needed.

**🚨 Common Symptoms of DRY Violation**

| **Symptom** | **Example** |
| --- | --- |
| Copy-paste code blocks | Same if-else in 3 places |
| Logic implemented in multiple layers | Tax logic in Checkout, Admin, Reporting |
| Business rules scattered | Discount rule duplicated across microservices |
| Hardcoded config values everywhere | “15%” discount in 5 files |

**✅ How to Apply DRY in Your Code**

| **Strategy** | **How it Helps** |
| --- | --- |
| Use helper functions | Abstracts repetitive logic |
| Use constants or config files | Avoid hardcoded values |
| Create reusable components | Frontend or backend modularity |
| Apply inheritance/composition | Avoid subclass duplication |
| Centralize business rules | Avoid logic spread in multiple places |

**🛠 Real-World System Example**

**Use Case: Parking Lot App**

**Repeated Logic:**

* Fee Calculation Logic used in: Entry Module, Exit Module, Report Module

**❌ Violation:**

Each module copies fee logic separately

**✅ Fix:**

Create FeeCalculatorService and inject wherever needed.

public class FeeCalculator {

public double calculateFee(int hours) {

return 50 + (hours - 1) \* 20;

}

}

**🚫 DRY ≠ Over-Abstraction**

Be careful of "over-abstraction":

Don’t DRY too early — **duplication is cheaper than the wrong abstraction.**

Wait until duplication is painful or logic stabilizes.

**📘 Summary: DRY Principle in One Line**

**"Every piece of knowledge must have a single, unambiguous, authoritative representation."**

**✅ Checklist to Follow DRY**

* Am I copy-pasting code blocks?
* Is the same logic written in multiple files?
* Are my constants hardcoded everywhere?
* Can I move repeated logic into a utility function or service?
* Is business logic scattered?

Would you like a **mini Java project** (like e-commerce or parking lot) demonstrating **DRY vs non-DRY version side-by-side**? I can prepare one for you.

**💡 KISS Principle: Keep It Simple, Stupid**

**✅ Definition:**

The **KISS Principle** states that *most systems work best if they are kept simple rather than made complex.* Therefore, **simplicity should be a key goal** in design, and unnecessary complexity should be avoided.

**💬 Origin:**

Coined by the U.S. Navy in the 1960s. The idea is that systems perform better when they are simple and straightforward.

**✅ Goal of KISS:**

* Improve **readability**
* Make code **easy to maintain**
* Reduce **bugs**
* Encourage **clarity** over cleverness

**🔍 Real-World Analogy:**

Imagine two designs for a **coffee machine**:

1. One with a single button: "Make Coffee" ☕
2. Another with 10 buttons for temperature, strength, steam pressure, timer, etc.

**KISS favors design #1** — unless the complexity is absolutely necessary.

**🧑‍💻 Code Example: Without KISS**

public int findMax(int[] arr) {

int max = Integer.MIN\_VALUE;

for (int i = 0; i < arr.length; i++) {

if (arr[i] > max) max = arr[i];

}

return max;

}

Now someone "clever" rewrites it:

public int findMax(int[] arr) {

return Arrays.stream(arr).boxed().sorted(Collections.reverseOrder()).findFirst().orElseThrow();

}

📉 This violates KISS. It’s complex for a simple problem.

**✅ KISS-friendly Version:**

public int findMax(int[] arr) {

int max = arr[0];

for (int i = 1; i < arr.length; i++)

if (arr[i] > max) max = arr[i];

return max;

}

Simple. Clear. Efficient. ✅

**🛠️ Real-World Problems for KISS**

**1. Login Authentication System**

❌ Complex: Splitting logic into 10 files, abstracting each validation step.  
✅ KISS: One class with clear responsibilities — validate username, password, and session in 2-3 functions.

**2. Student Grading System**

❌ Overengineering:

Map<String, List<Double>> subjectScores = new HashMap<>();

Map<String, Double> weightages = new HashMap<>();

✅ KISS:

class Student {

double math, science, english;

double average() {

return (math + science + english) / 3;

}

}

**3. Chat Application**

❌ Adds Redis cache, Kafka, microservices for 2-user messaging in an assignment.  
✅ KISS: Build with Socket.IO or simple HTTP-based poll for MVP.

**🔁 DRY Principle: Don't Repeat Yourself**

**✅ Definition:**

Every piece of knowledge must have a **single, unambiguous, authoritative representation** within a system.

Put simply: **Don’t write duplicate code.**

**✅ Goal of DRY:**

* Avoid code duplication
* Centralize business logic
* Simplify maintenance
* Reduce bugs when requirements change

**🧑‍💻 Without DRY**

public double getAreaOfCircle(double radius) {

return 3.14 \* radius \* radius;

}

public double getCircumferenceOfCircle(double radius) {

return 2 \* 3.14 \* radius;

}

If 3.14 changes (e.g., use Math.PI), we have to change in **multiple places**.

**✅ With DRY**

public class CircleUtils {

private static final double PI = Math.PI;

public static double getArea(double r) {

return PI \* r \* r;

}

public static double getCircumference(double r) {

return 2 \* PI \* r;

}

}

**🛠️ Real-World Problems for DRY**

**1. User Input Validation**

❌ Copying validation logic (email format, mobile number) in multiple places (Login, Signup, Update).

✅ DRY: Central ValidationUtil class to reuse.

**2. Database CRUD Logic**

❌ Rewriting same SQL queries or ORM methods in multiple services.

✅ DRY: Create a base repository class:

public abstract class BaseRepository<T> {

public void save(T entity) { /\* generic save \*/ }

public void delete(T entity) { /\* generic delete \*/ }

}

**3. Error Logging**

❌ Copying error logging logic everywhere:

System.out.println("ERROR: " + e.getMessage());

✅ DRY:

class Logger {

public static void logError(Exception e) {

System.out.println("ERROR: " + e.getMessage());

}

}

**🔁 DRY vs KISS in Practice**

| **Scenario** | **KISS** | **DRY** |
| --- | --- | --- |
| Remove overengineering | ✅ | ❌ |
| Reuse duplicate code | ❌ | ✅ |
| Use simple constructs | ✅ | ❌ |
| Write single source of truth | ❌ | ✅ |

❗ **Note:** These are complementary, not competing principles. Use **KISS to simplify**, **DRY to avoid duplication**.

**📚 Summary Table**

| **Principle** | **Stands For** | **Key Idea** | **Avoid** |
| --- | --- | --- | --- |
| KISS | Keep It Simple, Stupid | Simpler code is better | Overengineering, unnecessary abstraction |
| DRY | Don’t Repeat Yourself | Eliminate duplication | Copy-paste code, logic replication |

**🚀 Want to Practice?**

Would you like:

* A set of **LLD coding problems** to apply KISS/DRY?
* A **mini project** (like a Library App, To-Do App) where we apply these principles step-by-step?

Let me know, and I’ll tailor examples just for you.

**🧠 What is the YAGNI Principle?**

**YAGNI** stands for:

**"You Aren’t Gonna Need It"**

**✅ Definition:**

**YAGNI is a principle of extreme programming (XP)** that states:

*“You should not add functionality until it is necessary.”*

**🎯 Core Idea:**

Don’t write code for future scenarios that **may never happen**. Focus only on what is required **right now**.

**🚨 Why Is YAGNI Important?**

| **Without YAGNI** | **With YAGNI** |
| --- | --- |
| Overengineering | Simplicity |
| More bugs (in unused code) | Fewer bugs |
| Wasted time on "what-ifs" | Focus on real features |
| Harder maintenance | Easier to manage |

**🔧 Real-World Example 1: E-commerce Site - Coupon System**

**❌ Violating YAGNI:**

While building the cart, a junior dev thinks:  
*"One day we might need a crypto-based discount or time-bound coupon feature."*

They write:

class Coupon {

String type; // "flat", "percentage", "crypto", "seasonal", "future\_something"

// extra unused logic for each type

}

But right now, the business only uses:

type: "flat" or "percentage"

**✅ Applying YAGNI:**

Just handle **flat** and **percentage** coupons:

enum CouponType { FLAT, PERCENTAGE }

class Coupon {

CouponType type;

double value;

}

Add new types **when** there's a clear requirement.

**🚗 Real-World Example 2: Ride-Hailing App - Multi-City Support**

**❌ Violating YAGNI:**

You build a ride-booking service for **Bangalore only**.

But you pre-design:

* Multi-country time zones
* Currency conversion
* Localized language support

You waste 2 weeks building features that:

* May never be used
* Are hard to test
* Introduce complexity

**✅ YAGNI-Compliant Approach:**

Just build for Bangalore. Abstract logic like:

class City {

String name;

double baseFare;

}

Later, when needed, you can extend the system easily.

**📚 Real-World Example 3: Library Management System – Support for eBooks**

**❌ Over-Engineering:**

*"Let me add eBook streaming and DRM support in case they want it later..."*

But current requirement: Only track **physical books** in inventory.

**✅ YAGNI says:**

Build only for physical books.

class Book {

String title;

String author;

boolean isAvailable;

}

Only **add eBook support** when there is a clear, business-driven need.

**🔍 Signs You’re Violating YAGNI**

1. **“We might need this later...”**
2. **“Let’s handle every possible case upfront.”**
3. **Extra abstraction layers with no current use**
4. **Feature toggles for features not requested**

**🛠️ Where YAGNI Applies Most in LLD**

| **Area** | **YAGNI Advice** |
| --- | --- |
| Class design | Don’t add unnecessary properties or methods |
| Inheritance | Don’t generalize until reuse is proven |
| APIs | Don’t expose parameters for unused features |
| Exception handling | Don’t catch exceptions you don’t plan to handle |
| Configurations | Avoid unused flags and toggles |

**✅ When *Not* to Apply YAGNI Blindly**

* Security concerns (e.g., input validation, encryption) – always worth preparing.
* Public APIs – harder to change later, so think ahead.
* Performance bottlenecks – if measurable, you may want to pre-optimize.

**YAGNI ≠ being shortsighted.** It means **prioritizing real needs over speculation**.

**🔄 YAGNI in Agile & Clean Code**

* Part of **Agile Manifesto values**: "Responding to change over following a plan."
* Promotes **iteration**, **refactoring**, and **just-in-time design**.
* Keeps code clean, small, testable, and maintainable.

**🧘 Summary**

| **Principle** | **You Aren’t Gonna Need It** |
| --- | --- |
| Focus | Build only what's needed **now** |
| Prevents | Overengineering, complexity, waste |
| Supports | Simplicity, agility, clean code |
| Use with | SOLID, KISS (Keep it simple), Agile |

**💡 Final Thought**

**“Every line of code is a liability until proven otherwise.”**

If you’d like, I can give you:

* A checklist for applying YAGNI in your next project
* A side-by-side code comparison for YAGNI vs Non-YAGNI approaches

Would you like that?

**🧠 What is the Law of Demeter?**

**🔹 Definition:**

**"A method M of an object O should only call methods of the following:"**

* O itself
* Objects passed as arguments to M
* Objects that are created inside M
* Objects that are O’s direct fields (i.e., instance variables)

This principle is also known as:

* **"Don't talk to strangers"**
* **"Only talk to your immediate friends"**

**✅ Core Idea**

The Law of Demeter tells you to **avoid chaining method calls** to deep levels, like:

a.getB().getC().doSomething();

Instead, the logic should be contained and you should **delegate** or **encapsulate** the behavior:

a.doSomething();

**❌ Violating the Law (Tight Coupling)**

**Example 1: Car Engine Access (Bad)**

class Engine {

void start() {}

}

class Car {

Engine engine = new Engine();

public Engine getEngine() {

return engine;

}

}

class Driver {

public void drive(Car car) {

car.getEngine().start(); // ❌ Violates Law of Demeter

}

}

**✅ Applying Law of Demeter (Good)**

class Car {

Engine engine = new Engine();

public void start() {

engine.start(); // Car delegates the task

}

}

class Driver {

public void drive(Car car) {

car.start(); // ✅ Driver only interacts with Car

}

}

**🤖 Why Is Law of Demeter Important?**

| **Benefit** | **Explanation** |
| --- | --- |
| ✅ Low coupling | Your classes only depend on direct friends |
| ✅ Easier to maintain | If internal structures change, clients aren’t affected |
| ✅ Improves encapsulation | Objects control their own state and behavior |
| ✅ Easier to test | You don’t need deep mocking chains in unit tests |

**📦 Real-World Problem Scenarios**

**🎓 1. University Management System**

**❌ Without LoD:**

class Student {

Transcript transcript;

public Transcript getTranscript() { return transcript; }

}

class Transcript {

public double getGPA() { return 8.5; }

}

class Admin {

void printGPA(Student student) {

System.out.println(student.getTranscript().getGPA()); // ❌ Deep call chain

}

}

**✅ With LoD:**

class Student {

Transcript transcript;

public double getGPA() {

return transcript.getGPA(); // Encapsulation

}

}

class Admin {

void printGPA(Student student) {

System.out.println(student.getGPA()); // ✅ Clean interaction

}

}

**🛍️ 2. E-Commerce Order Processing**

**❌ Bad:**

order.getCustomer().getAddress().getCity(); // ❌ Too much knowledge of internal structure

**✅ Good:**

order.getDeliveryCity(); // ✅ Order internally handles address logic

**🏥 3. Hospital Patient Data Access**

**❌ Deep Object Graph:**

patient.getMedicalRecord().getHistory().getAllergies();

**✅ Clean API:**

patient.getAllergies();

**⚠️ Signs You Are Violating LoD**

* You write chains like a.getB().getC().doX()
* Unit tests require **multiple mock objects** just to test one class
* Changes in one class **break many others**

**📘 Guideline to Follow LoD**

| **✅ Do** | **❌ Avoid** |
| --- | --- |
| Tell objects what to do | Ask objects for internals |
| Delegate responsibility | Chain multiple method calls |
| Work with direct collaborators | Traverse through object graphs |

**🧪 Quick Mnemonic**

**"Only talk to your immediate friends, not their friends’ friends."**

**🛠️ Summary Table**

| **Concept** | **Explanation** |
| --- | --- |
| Principle Name | Law of Demeter |
| Alternative Name | Don't talk to strangers |
| Goal | Reduce tight coupling, improve encapsulation |
| Target | Method call structure in LLD |
| Impact | Clean, testable, maintainable design |

**📦 Bonus Tip: Combine LoD with SOLID**

* Use **SRP** to break classes with multiple responsibilities
* Use **OCP** to extend behavior without modifying internals
* Use **DIP** to work through abstractions, not deep graphs
* Use **LoD** to clean up method interactions

If you’d like, I can take a **real-world app idea** (e.g., ride-hailing app, food delivery system, etc.) and show where **Law of Demeter is applied or violated** in its LLD code. Want that next?