Lecture 1: Introduction to Low-Level Design (LLD)

---

1. What is Low-Level Design (LLD)?

Definition:
Low-Level Design refers to the process of defining the internal structure and behavior of software components within an application. It involves identifying the classes and objects that make up the system, understanding their attributes and responsibilities, and designing the relationships and interactions between them. LLD serves as the **"skeleton"** of your application — a blueprint that outlines how different components of the software interact internally at a code level.

While **Data Structures and Algorithms (DSA)** focus on solving specific computational problems (such as finding the shortest path, sorting data, or implementing stacks/queues), LLD focuses on organizing these solutions within the right object-oriented structures to form a complete, scalable, and maintainable application.

Key differences between DSA and LLD:

- **DSA** is concerned with solving isolated problems using logic and efficient computation (e.g., Binary Search, QuickSort, Dijkstra's Algorithm, Min/Max Heaps, etc.).
- **LLD** involves constructing object models, defining interfaces, implementing design patterns, and deciding where and how to embed DSA solutions within the architecture of a real-world software system.

---

2. An Illustrative Example: Designing a Ride Booking App – "QuickRide"

Let's consider how two different developers, **Anurag** and **Maurya**, approach designing a ride-hailing application like Uber or Ola.

Anurag's DSA-First Approach:

Anurag focuses on solving the computational challenges first. His steps include:

1. **Breaking down the core problem:**
    - Map all the city intersections to graph nodes.
    - Map roads between intersections as edges.
    - Use **Dijkstra's Algorithm** to compute the shortest path between two points.

   - Use a **Min-Heap** or priority queue to find the nearest driver to a rider.

2. **Limitations of this approach:**
   - Fails to identify real-world **entities** like User, Rider, Driver, Location, or Payment.
   - Doesn't include security mechanisms such as **data masking** for phone numbers.
   - Lacks integration points like **notification systems** and **payment gateways**.
   - Ignores the need to **scale** the application to serve millions of users effectively.
   - Overall, this approach is algorithmically sound but not practical for real-world deployment.

Maurya's LLD-First Approach:

Maurya, on the other hand, begins by thinking about the **real-world objects and behaviors** involved in a ride-booking application.

1. **Identifying key entities and classes:**
   - Defines core objects such as: User, Rider, Driver, Location, Trip, NotificationService, PaymentGateway, etc.
2. **Defining relationships and interactions:**
   - How a User books a ride with a Driver based on their Location.
   - How NotificationService sends alerts and confirmations.
   - How PaymentGateway processes transactions securely.
3. **Addressing non-functional requirements:**
   - Ensures **data privacy and security** by encrypting sensitive information.
   - Designs for **scalability**, so the system can serve millions of users simultaneously.
   - Builds code with maintainability and modularity in mind, supporting new features and services.
4. **Finally applying DSA:**
   - Integrates shortest path algorithms and driver-matching logic **within** the class structure he has built.

- Embeds algorithmic solutions in a real-world object-oriented framework, making the code production-ready.

---

3. Core Principles and Goals of LLD

LLD aims to build software that is not just functional, but also **scalable, maintainable, and reusable**. Here are the primary principles:

1. **Scalability:**
    - The design should easily handle increased user load without performance bottlenecks.
    - The system should allow effortless addition of new features, services, or servers with minimal changes.
2. **Maintainability:**
    - Code should be modular and easy to debug or test.
    - New features or updates should not break existing functionality.
3. **Reusability:**
    - Components should be loosely coupled so they can be reused in different applications.
    - For example, a well-designed NotificationService can be reused across apps like Swiggy, Zomato, or even in logistics apps like Delhivery or Dunzo.

---

4. What LLD Is NOT (Compared to HLD)

It's important to distinguish between **Low-Level Design (LLD)** and **High-Level Design (HLD)**.

- **LLD** focuses on code-level design: class diagrams, object relationships, method-level details, and data flow within components.
- **HLD** deals with system-wide architecture and infrastructure decisions:
    - **Technology stack** (e.g., Java Spring Boot, Node.js, Python Django).
    - **Database design** (SQL vs. NoSQL, RDBMS vs. document stores).
    - **Deployment strategies** (cloud platforms like AWS/GCP, autoscaling, load balancers).
    - **Cost optimization** and server provisioning strategies.

In essence:

- **HLD** is about designing *how systems interact at the macro level*.
- **LLD** is about *how each system internally functions at the micro/code level*.

---

5. Summary and Takeaways

- **DSA** represents the **brain** of your application — it helps solve critical algorithmic problems.
- **LLD** forms the **skeleton** of your application — shaping the object models, relationships, and code structure that hold everything together.
- **HLD** defines the **architecture** — the overall system design, including databases, servers, APIs, cloud infrastructure, and more.

Together, these three layers are essential for building robust and production-ready software systems.

Key Line to Remember:

“If DSA is the brain, then LLD is the skeleton of your application.”