What is LLD?

Low-level design (LLD) is about detailing how a software system will be built and how it will work in detail. It focuses on the specific implementation details, forming the basis for the high-level design, which describes the overall structure and architecture of the system. This phase involves designing the detailed logic and structures of a system, including the implementation specifics of individual components.

High-Level Design (HLD) Focus: System architecture and overall framework. Overview: Provides a bird's-eye view of the solution, outlining the major components and their interactions. Aspects Covered: Technology choices, overall system architecture, databases, platform services, and other external dependencies. Objective: Addresses how the system will meet the functional requirements, laying out the blueprint for development. Details: Identifies software modules and their high-level functionalities, data flow, infrastructure, and communication with third-party services. Output: Diagrams and documentation offering a general understanding of the system, such as block diagrams representing various system components and their interrelationships, typically without intricate details.

Low-Level Design (LLD) Focus: Implementation specifics. Overview: Delves into the internal design of individual modules identified during the HLD phase. Aspects Covered: Detailed specifications of the code level, including algorithms, interface designs, procedural details, and complete data models. Objective: Provide detailed guidance for developers to follow during coding, ensuring the precise behavior and structure of components. Details: Includes detailed class diagrams with methods and attributes, database tables with key constraints, detailed sequence diagrams, and state diagrams. Output: Detailed documentation that guides the developer through the coding phase without ambiguity.

Example: Online Shopping Cart High-Level Design (HLD):

Focus: The HLD for an online shopping cart outlines the main components like user interface, product catalog, shopping cart, payment gateway, and order management. Overview: It provides a bird's-eye view of the system, showing how these components interact with each other. Output: High-level diagrams and documentation that explain the major components and their interactions without going into implementation details.

Low-Level Design (LLD): Focus: The LLD for the online shopping cart dives

into the specifics of each component identified in the HLD. Details: It details how each component will be implemented, including: Data Structures: Define the data structures for products, user accounts, shopping cart items, orders, and payment details. Algorithms: Describe the algorithms for adding items to the cart, calculating the total price, applying discounts, and processing payments. Component Interactions: Specify how different components like the shopping cart and payment gateway will communicate. For example, detailing the API calls and data exchange formats. Interfaces: Define the methods and protocols used for communication between components. For instance, how the shopping cart component interfaces with the product catalog to update item quantities.

Standard Coding Guidelines Standard coding guidelines are a set of rules and conventions that developers follow when writing code. These guidelines help ensure that the code is consistent, readable, and maintainable across the entire codebase. Rules Consistency: Using the same style and practices throughout the codebase. Readability: Making the code easy to read and understand for any developer. Maintainability: Ensuring the code can be easily maintained and updated.

Examples Naming Conventions: Variables and Functions: Use meaningful names like totalSum or calculateTotal. Classes: Use CamelCase, like CustomerDetails. Constants: Use all uppercase letters with underscores, like MAX_SIZE. Indentation Style: Use spaces or tabs consistently for indentation (e.g., 4 spaces per indent). Align code blocks properly to enhance readability. Commenting Practices: Write comments to explain complex logic. Use comments to provide an overview of the code's purpose. Keep comments up to date with the code changes. Ideologies Standard coding guidelines are often based on core programming principles or ideologies. These principles promote best practices in coding to enhance code quality.

”Don't Repeat Yourself” (DRY): Avoid duplicating code by creating reusable functions or modules. Example: Instead of writing the same code in multiple places, create a function and call it wherever needed.

”Keep It Simple, Stupid” (KISS): Aim for simplicity in your code. Complex solutions should only be used when necessary. Example: Write clear and straightforward code instead of using overly complicated logic. Benefits Code Reusability: By following these guidelines, developers can write reusable code that can be easily used in different parts of the application. Simplicity: Simplifies the code, making it easier for developers to understand and work with. Maintainability: Ensures that the codebase is easy to maintain, update, and debug, which is crucial for long-term project success.

Standard coding guidelines are essential for any software development project. They ensure that the code is consistent, readable, and maintainable, making it easier for developers to collaborate and maintain the code over time. By

adhering to principles like DRY and KISS, developers can write high-quality code that is both efficient and easy to manage.

Building Blocks of Low-Level Design (LLD):

Object-Oriented Programming (OOP) Low-Level Design (LLD) often leverages Object-Oriented Programming (OOP) to structure and organize code. OOP is a programming paradigm that centers around objects, which are instances of classes that represent real-world entities. It helps in breaking down complex systems into manageable components by using principles and concepts like encapsulation, inheritance, and polymorphism.

Core Concepts: Classes and Objects Class: A class is a blueprint or template for creating objects. It defines a set of attributes (data) and methods (functions) that the created objects will have. Purpose: Classes provide a way to bundle data and functionality together. They allow for the creation of multiple objects with the same structure and behavior. Example:

// Definition of a class public class BankAccount { // Attributes (data) private String accountNumber; private double balance;

// Constructor public BankAccount(String accountNumber, double initialBalance) { this.accountNumber = accountNumber; this.balance = initialBalance; }

// Methods (functions) public void deposit(double amount) { if (amount > 0) { balance += amount; } }

public void withdraw(double amount) { if (amount > 0 && amount <= balance) { balance -= amount; } }

public double getBalance() { return balance; }

public String getAccountNumber() { return accountNumber; } }

Object: An object is an instance of a class. It represents a specific entity with concrete values for the attributes defined in the class. Objects can interact with one another through their methods. Purpose: Objects encapsulate state and behavior, making it easier to model real-world entities and manage their interactions.

Example:

public class Main { public static void main(String[] args) { // Creating objects (instances) of the BankAccount class BankAccount myAccount = new BankAccount("123456", 500.00); BankAccount anotherAccount = new BankAccount("654321", 300.00);

// Using methods on the objects myAccount.deposit(150.00); myAccount.withdraw(50.00); System.out.println("Balance in myAccount: " + myAccount.getBalance()); // Output: 600.00

anotherAccount.withdraw(100.00); System.out.println("Balance in anotherAccount: " + anotherAccount.getBalance()); // Output: 200.00 } }

Major Elements of OOP

Abstraction Definition: Abstraction involves highlighting essential features of an object while hiding unnecessary details. It helps simplify complex systems by focusing on relevant aspects and ignoring implementation complexities. Example: In a banking application, users interact with an ATM to perform transactions without needing to know the internal workings of the ATM. Importance: Abstraction simplifies complex systems by focusing on what an object does rather than how it does it. This helps in managing complexity and improving code readability. Implementation: Abstract classes and interfaces define common methods and properties but leave the specifics to derived classes. abstract class Shape { abstract void draw(); // Abstract method, to be implemented by subclasses }

class Circle extends Shape { void draw() { System.out.println("Drawing Circle"); } }

class Rectangle extends Shape { void draw() { System.out.println("Drawing Rectangle"); } }

Encapsulation Definition: Encapsulation is the process of bundling data (attributes) and methods (functions) into a single unit called a class. It restricts direct access to some of an object's components, which helps protect data integrity and prevent unintended modifications. Example: A BankAccount class encapsulates the account balance and methods for depositing and withdrawing money, ensuring that these operations are performed through controlled interfaces. Importance: Encapsulation provides a clear separation between an object's internal state and the outside world. It ensures that data is only accessed and modified through defined methods, maintaining integrity and security. Implementation: Use private access modifiers for attributes and public access modifiers for methods that interact with these attributes.

class BankAccount { private double balance; // Private attribute

public void deposit(double amount) { // Public method to modify balance if (amount > 0) { balance += amount; } }

public double getBalance() { // Public method to access balance return balance; } }

Inheritance Definition: Inheritance allows one class (child or derived class) to inherit attributes and methods from another class (parent or base class). This promotes code reuse and modularity. Example: A SavingsAccount class and a CheckingAccount class can both inherit from a BankAccount class, reusing common features while adding their specific attributes and methods. Importance: Inheritance allows for code reuse and the creation of a hierarchical relationship between classes. It helps in building complex systems with shared functionalities. Implementation: Define a base class and derive new classes from it using inheritance keywords.

class BankAccount { String accountNumber; double balance;

public void deposit(double amount) { balance += amount; }

public void withdraw(double amount) { balance -= amount; } }

class SavingsAccount extends BankAccount { double interestRate;

public void addInterest() { balance += balance * interestRate; } }

Polymorphism Definition: Polymorphism enables objects of different classes to be treated as objects of a common superclass. It allows methods to be used interchangeably, and the specific method that gets called depends on the object's actual class. Example: The calculateInterest() method can be overridden in both SavingsAccount and CheckingAccount classes, allowing different implementations of interest calculation while using the same method name. Importance: Polymorphism enables objects of different classes to be treated through a common interface, which enhances flexibility and code maintainability. Implementation: Achieved through method overriding in derived classes and method overloading in the same class.

class BankAccount { public void calculateInterest() { System.out.println("Calculating interest for bank account"); } }

class SavingsAccount extends BankAccount { @Override public void calculateInterest() { System.out.println("Calculating interest for savings account"); } }

class CheckingAccount extends BankAccount { @Override public void calculateInterest() { System.out.println("No interest for checking account"); } }

public class Main { public static void main(String[] args) { BankAccount account1 = new SavingsAccount(); BankAccount account2 = new CheckingAccount();

account1.calculateInterest(); // Output: Calculating interest for savings account account2.calculateInterest(); // Output: No interest for checking account } }

Design Principles Design principles are fundamental guidelines that aid developers in crafting maintainable, scalable, and robust software systems. They facilitate the creation of software with desirable qualities, such as flexibility, modularity, and reusability.

Importance of Design Principles Maintainability: Code is easier to understand and modify, reducing future development costs. Adaptability: Changes can be made without extensive rework, ensuring longevity. Collaboration: Shared understanding among team members leads to improved teamwork.

Design Patterns Design patterns represent proven solutions to recurring design problems in software development. They embody best practices that have evolved and are applied to address specific design challenges. By using design patterns, developers can implement reusable and standardized solutions, enhancing software robustness and maintainability.

Difference Between Design Patterns and Principles Design Principles: Serve as general guidelines that inform design decisions and shape a software system's architecture. They provide high-level concepts and rules that govern overall design. Design Patterns: Offer specific solutions to common design problems that emerge from the application of design principles. Design patterns focus more on implementation-level details, providing detailed solutions for specific scenarios.

Single Responsibility Principle (SRP) Definition: A class should have only one reason to change, meaning it should have a single responsibility. Benefits: Reduces complexity. Makes code more understandable. Enhances maintainability. To illustrate SRP, let's consider an application that calculates the total area of a collection of shapes (circles and squares). We'll start by creating the shape classes and then develop a calculator to sum their areas.

Bad Example:

class Employee { private int empId; private String name; private double salary;

public Employee(int empId, String name, double salary) { this.empId = empId; this.name = name; this.salary = salary; }

// Business logic public double calculateSalary() { return salary * 1.1; // add 10% bonus }

// Persistence public void saveToDatabase() { System.out.println("Saving " + name + " to database..."); }

// Reporting public void generateReport() { System.out.println("Employee Report: " + name + ", Salary: " + salary); } }

Problem: * Class has 3 responsibilities (salary calculation, database, report). * If report format changes → this class changes. * If database changes → this class changes. * Hard to test and maintain.

Good Example:

class Employee { private int empId; private String name; private double salary;

public Employee(int empId, String name, double salary) { this.empId = empId; this.name = name; this.salary = salary; }

public int getEmpId() { return empId; } public String getName() { return name; } public double getSalary() { return salary; } }

// Salary calculation responsibility class SalaryCalculator { public double calculateSalary(Employee employee) { return employee.getSalary() * 1.1; } }

// Database responsibility class EmployeeRepository { public void saveToDatabase(Employee employee) { System.out.println("Saving " + employee.getName() + " to database..."); } }

// Reporting responsibility class EmployeeReport { public void generateReport(Employee employee) { System.out.println("Employee Report: " + employee.getName() + ", Salary: " + employee.getSalary()); } }

Benefits: * Each class has exactly one reason to change. * Easier to maintain and test. * Respects Single Responsibility Principle.

Open-Closed Principle (OCP) Definition: Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. Benefits: Encourages the addition of new features without altering existing code. Enhances stability and reduces the risk of introducing bugs.

class Rectangle { double length; double width;

Rectangle(double length, double width) { this.length = length; this.width = width; } }

class Circle { double radius;

Circle(double radius) { this.radius = radius; } }

class AreaCalculator { public double calculateArea(Object shape) { if (shape instanceof Rectangle) { Rectangle r = (Rectangle) shape; return r.length * r.width; } else if (shape instanceof Circle) { Circle c = (Circle) shape; return Math.PI * c.radius * c.radius; } return 0; } }

Problem: * If we add a new shape (like Triangle), we must edit AreaCalculator. * Violates OCP → not closed for modification.

Good Example (follows OCP) We introduce an abstraction (Shape interface). Each shape defines its own area. Now we can add new shapes without modifying existing classes.

interface Shape { double area(); }

class Rectangle implements Shape { private double length; private double width;

Rectangle(double length, double width) { this.length = length; this.width = width; }

@Override public double area() { return length * width; } }

class Circle implements Shape { private double radius;

Circle(double radius) { this.radius = radius; }

@Override public double area() { return Math.PI * radius * radius; } }

class AreaCalculator { public double calculateArea(Shape shape) { return shape.area(); } }

Benefits: * If we add a new Triangle class, we don't touch AreaCalculator. * AreaCalculator is closed for modification, but open for extension. * Clean, maintainable, and scalable.

Liskov Substitution Principle (LSP)

Subclasses should be substitutable for their base classes without affecting the correctness of the program. The Liskov Substitution Principle (LSP) states: Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T. This means that every subclass or derived class should be substitutable for their base or parent class.

Bad Example (violates LSP) Here, Square inherits from Rectangle, but it breaks expected behavior.

class Rectangle { protected int width; protected int height;

public void setWidth(int width) { this.width = width; }

public void setHeight(int height) { this.height = height; }

public int getArea() { return width * height; } }

class Square extends Rectangle { @Override public void setWidth(int width) { this.width = width; this.height = width; // forces height = width }

@Override public void setHeight(int height) { this.height = height; this.width = height; // forces width = height } }

Problem: Square is not really a subtype of Rectangle. If a method expects a Rectangle and sets width and height separately, it will misbehave when given a Square.

Example:

public class Main { public static void main(String[] args) { Rectangle rect = new Square(); rect.setWidth(4); rect.setHeight(5); System.out.println(rect.getArea()); // Expected 20, but gets 25 } } This violates LSP because substituting Square for Rectangle breaks correctness.

Good Example (follows LSP) We separate the abstractions: Shape instead of forcing Square to be a Rectangle.

interface Shape { int getArea(); }

class Rectangle implements Shape { private int width; private int height;

public Rectangle(int width, int height) { this.width = width; this.height = height; }

@Override public int getArea() { return width * height; } }

class Square implements Shape { private int side;

public Square(int side) { this.side = side; }

@Override public int getArea() { return side * side; } }

Benefits: Rectangle and Square both extend Shape. No broken assumptions: they can be substituted wherever a Shape is expected. Clean, follows Liskov Substitution Principle.
