Low Level System Design

LLD focuses on how individual components of a system are designed and implemented. It deals with

- classes
- objects
- Interfaces
- relation b/w classes
- Design patterns.

HLD defines what components exists, LLD defines how those components are implemented.

A good LLD should produce code that is

- Readable
- Reusable
- Testable.
- Extensible.
- Easy to debug.

- **Design Principles :-**

1. SOLID

- **Single Responsibility Principle** - This means a class should handle only one responsibility or concern.
- Change is inevitable, multiple responsibilities causes cascading changes.

Tight coupling make systems fragile.

- SRP minimizes the impact of change by isolating responsibilities.

wrong interperation. ✗

Class has only one public method.

correct interperation. ✓

Will this class change for more than one reason?

Wrong Design :

```
class Order {
    void calculateTotal() {}
    void saveToDatabase() {}
    void sendConfirmationMail() {}
}
```

correct Design :

```
class Order {
    void calculateTotal() {}
}
class OrderRepository {
    void save(Order order) {}
}
class NotificationService {
    void sendConfirmation(Order order) {}
}
```

- Each class has one responsibility.
- changes are localized.
- Testing becomes simpler.

Manager → decision making

Accountant → finances.

HR → employment management.

One person doing everything leads to inefficiency - same with class.

Note :-
- Avoid creating too many tiny classes.
- Splitting logic prematurely.
- Sacrificing readability.

If responsibilities always change together, they can stay together.

SRP ensures that a class has only one responsibility and one reason to change, leading to cleaner and more maintainable systems.

- Open close principle.

Software entities (classes, modules, functions) should be open for extention but closed for modification.

This means:
- you should be able to add new behavior
- without changing existing, tested code.

Why OCP:-
In real systems code is already deployed and tested.
Multiple teams depend on it.
Modifying existing code, introduces bugs, breaks backward compatibility and increases regression risk.
OCP help us extend behavior without risking existing functionality.

OCP usually implemented using:
- Abstraction (Interface/ abstract classes).
- Polymorphism.
- Composition.
- Strategy Pattern.

Wrong Design:

```
class DiscountCalculator {
    double calculate(String type, double amt) {
        if (type.equals("STUDENT")) return amt * 0.9;
        else if (type.equals("SENIOR")) return amt * 0.8;
        return amt;
    }
}
```

```
class DiscountCalculator {
    double calculate(DiscountStrategy strategy,
                     double amt) {
        return strategy.apply(amt);
    }
}
```

Correct Design:

```
interface DiscountStrategy {
    double apply(double amt);
}
class StudentDiscount implements DiscountStrategy {
    public double apply(double amt) {
        return 0.9 * amt;
    }
}
class SeniorDiscount implements DiscountStrategy {
    public double apply(double amt) {
        return 0.8 * amt;
    }
}
```

Eg. Plug Socket analogy.

- Socket is fixed. (Closed for modification)
- Different devices plug in (open for extension).

OCP allows software behaviour to be extended without modifying existing, stable code.

- **Liskov Substitution Principle (LSP)**
- Objects of super class should be replaceable with object's of its subclass without breaking the correctness of the program.

LSP ensures - polymorphism works correctly.
- Inheritance does not introduce bugs.
- Code using base classes remains reliable.

LSP is not about method signatures. It is about behavioral compatibility.

Wrong Design:-

```
Class Rectangle {
int width, height;
void setWidth (int w) { width = w; }
void setHeight (int h) { height = h; }
int area () {
    return width * height;
}}
Class Square extends Rectangle {
void setWidth (int w) {
    width = height = w;
}
void setHeight (int h) {
    width = height = h;
}}
```

```
Rectangle r = new Square ();
r.setWidth (5);
r.setHeight (10);
System.out.println (r.area ());
```

Expects 50, gets 100.

Correct Design

```
interface Shape{
  int area();
}
class Rectange implements Shape{
  int width, height;
  int area(){return width*height;}
}
class Square implements Shape{
  int side;
  int area(){return side*side;}
}
```

- polymorphism works safely.
- clean Abstraction.

Ex:- . Birds can fly.

. Penguin is a bird but cannot fly. If code assumes

bird.fly() → breaks LSP.

better seperate FlyingBird and NonFlying Bird.

- **Interface Segregation Principle (ISP)**

  Clients should not be forced to depend on interfaces they do not use,

- many small, specific interfaces are better than one-large general interface.

Why ISP:-

Large Interfaces forces classes to implement unnecessary methods, increases coupling and make changes risky.

ISP ensures cleaner abstractions, reduces dependency impact and easier mantance.

Wrong Design

```
interface Workerd
  void work();
  void eat();
}
class Robotworker implements Workerd
  public void work(){ }
  public void eat(){ } //not applicable.
}
```

Robot forced to implement eat(). leads to dummy implementation.

Correct Design

```
interface Workable{
  void work();
}
interface Eatable{
  void eat();
}
class Humanworker implements Workable, Eatable{
  public void work(){}
  public void eat(){}
}
class Robotworker implements Workable{
  public void work(){
}
```

Eg: Remote control analogy.

- TV remote has TV buttons.
- AC remote has AC buttons.

one remote for everything → confusing and unnecessary.

ISP ensures that classes depend only on the methods they actually use, leading to cleaner and more flexible designs

• Dependency Inversion Principle.

High level modules should not depend on low-level modules. Both should depend on abstractions.

Why DIP?

- High-level business logic is tightly coupled to low-level implementation,
- Changing databases, API's or framework becomes hard.
- Testing requires complex setup.

DIP enables:

- Loose coupling, easy testing (mocking), swappable implementations.

High level module → Business logic.

Low level module → Implementation details (DB, network, file system).

Wrong Design

```
class MySQLDatabase {
  void save(String data){}
}
class UserService {
  MySQLDatabase db = new
                MySQLDatabase();
  void saveUser(String user){
    db.save(user);
  }
}
```

- Changing DB requires modifying UserService.
- Hard to test.
- Violates DIP.

Correct Design

```
interface Database {
  void save(String data);
}
class MySQLDatabase implements Database{
  Public void save(String data){ }
}
class UserService {
  private Database db;
  UserService(Database db){
    this.db=db;
  }
  void saveUser(String user){
    db.save(user);
  }
}
```

- DB can be swapped.
- User service remains unchanged
- Easy to mock in test.