Low Level Design (LLD) :-

⇒ Breaking down or designing the real logic into extensible, testable, readable code.

SOLID PRINCIPLES :-

↳ Single Responsibility Principle (SRP) :-

⇒ A class should only have one responsibility

OR

A class should have only one reason to change.

Example :-

Task Manager :-

① Manages tasks ② Sends Notifications

③ Logs notification events.

PROBLEM ??

⇒ Let's say the logging logic or notification logic changes, in that case we would have to change the task manager class. Which is a bad design.

* Solution :-

Break the class into 3 different components

$$\begin{tikzcd}
 & \text{Break the class into 3 different components} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{TaskService} & \text{NotificationService} & \text{LoggingService}
\end{tikzcd}$$

* SOLID :-

↳ Open/Close Principle (OCP)

⇒ A class should be open for extension but closed for modification.

Example :- Payment Service :-

```
class PaymentService {
    void pay(String type) {
        if(type.equals("UPI")) { .... }
        else if(type.equals("Card")) { ... }
    }
}
```

*PROBLEM?? ⇒ On adding a new type the OCP is violated.

*Solution :- Create PaymentMethod interface & extend from it.

$$\begin{tikzcd}
 & \text{PaymentMethod} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Card Payment} & \text{UPI Payment} & \text{Wallet Payment}
\end{tikzcd}$$

*Why? Adding new types doesn't affect the previous types code. Less Bugs, Modular.

* SOLID:-
↳ Liskov Substitution Principle (LSP)

⇒ Subtypes must be suitable for their base type.

```
class Rectangle {
    void setWidth (w);
    void set height (w);
}
 class Square extends Rectangle {
     void setWidth (int w) {
          setHeight(w);
     }
     void setHeight (intw) {
          set width(w);
      }
}
   public void resize(Rectangle r) {
        r.setWidth ( 5 );
        r.set Height ( 10 );
        print ( r. area() );
   }
```

Output:- For rectangle → 50
For square → 100 (X)

⇒ Solution:- Keep Rectangle And Square Separately.

* SOLID :-

↳ Interface Segregation Principle (ISP)

⇒ Clients should not be forced to depend on interfaces that they do not use.

Example:-

```
interface MediaPlayer{
   void playVideo();
   void playAudio();
}
```

```
class AudioPlayer implements
                    VideoPlayer{
   void playVideo(){
   }
   void playAudio(){
      - - - -
      - - - -
      -   - -
   }
}
```

Problem:- Bad design as we would either make play video in the above example as empty or throw an exception.

Solution:- Create 2 separate interfaces

```
interface AudioPlayer{
    void play();
}
```

```
interface VideoPlayer{
    void play();
}
```

* SOLID

↳ Dependency Inversion Principle (DIP)

⇒ High level modules should not depend on low level modules. Both should depend on abstractions (like interfaces)

* Example :-

```
class UserService {
    FileLogger logger = new FileLogger();
    void createUser() {
        logger.log("user created");
    }
}
```

Problem :- If you want to test the class & test the logger it would be difficult to do.

Also changing logger to something like database logger would require code changes in UserService.

* Solution :-

```
                                  → class FileLogger implements Logger {
                                      ...}
interface Logger {
    void log(String message);
}                                 → class DBLogger implements Logger {
                                      ...}

class UserService {
    Logger logger;
    UserService(Logger logger) {
        this.logger = logger;
    }
    void createUser() {
        logger.log("user created");
    }
}
```