**Low Level Design Pattern**

**GitHub link:** <https://github.com/geek-shivamraj/design-patterns-2025>

**SOLID Principle**

S – Single Responsibility Principle

O – Open/Closed Principle

L – Liskov Substitution Principle

I – Interface Segregation Principle

D – Dependency Inversion Principle

**Advantages of following these principles:**

Help us to write better code:

* Avoid Duplicate code
* Easy to maintain
* Easy to understand
* Flexible software
* Reduce complexity

**S – Single Responsibility Principle**

* A Class should have only 1 reason to change i.e., a class should have only 1 responsibility.
* Example:

|  |
| --- |
| **Scenario:**  Suppose we have an Invoice class with multiple functionality:   * Feature 1 – CalculateInvoice * Feature 2 – PrintInvoice * Feature 3 – SaveInvoice   **Question:** Is Invoice class following Single Responsibility Principle i.e., does it have only 1 reason to change?  **Answer:** NO  **Problem**:  As this class has multiple reasons to change   * Reason 1 – Suppose we want to change the calculation logic & add GST or discount then definitely calculation logic will change. * Reason 2 – Change in Printing logic * Reason 3 – Change in saving Invoice logic (In DB, File, Caching)   **Solution**:  We can split Invoice class to **CalculateInvoice**, **PrintInvoice**, **SaveInvoice**. |

**O – Open/Closed Principle**

* Open for Extension but closed for Modification i.e., Already existing tested & deployed classes should be extended to add new functionality rather than any Modification to existing one.

|  |
| --- |
| **Scenario:**  Suppose the SaveInvoice class having **saveToDB** () method is already tested & live & now we have a new requirement to SaveInvoice to File as well.  **Brute Approach:**  Add a new method as **saveToFile** () in the same tested & deployed class SaveInvoice.  **Problem:**  With Brute approach, there is a chance that we will introduce bugs to the existing working class & that’s not recommended.  **Solution**:  We can split Invoice class to CalculateInvoice, PrintInvoice, SaveInvoice. |

**L – Liskov Substitution Principle**

* If class B is a subtype of class A, then we should be able to replace object of A with B without breaking the behavior of the program.
* Subclass should extend the capability of parent class, does not narrow it down.

|  |
| --- |
| **Scenario:**  Suppose we’ve a class “Vehicle” having methods **getNumberOfWheels** (), **hasEngine** (),  **accelerate** () & we’ve main client class checking whether all the classes extending Vehicle supports or extends Vehicle class functionality.  Extension to Vehicle class:   1. Car extends Vehicle – Supports all the functionality 2. Motorcycle extends Vehicle – Supports all the functionality 3. Bicycle extends Vehicle – Doesn’t support hasEngine or accelerate functionality     **Problem:**  Vehicle class has some methods which are not relevant to all of its extension that can result in breaking the existing functionality.  E.g., I want to print Boolean whether all the Vehicle extensions hasEngine () or not   * Bicycle object can return null value that can result in **NullPointerException**   **Solution:**  Keep only generic methods/functionality in the super class & let the extensions have their specific functionality. |

**I – Interface Segregation Principle**

* Interfaces should be such, that client should not implement unnecessary functions they don’t need.

|  |
| --- |
| **Scenario:**  Suppose we have an interface **“RestaurantEmployee”** with below methods:   * washDishes () * serveCustomers () * cookFood ()   Now, the classes like Chef or Waiter implementing RestaurantEmployee interface have to implement all the methods, some of which are not relevant to Chef class or Waiter class.  **Solution:**  Segregate interface “RestaurantEmployee” into smaller interfaces such that the client doesn’t have to implement methods not relevant to them.  i.e., RestaurantEmployee can be segregated to   * **WaiterInterface** with waiter specific methods like serverCustomers (), takeOrder () * **ChefInterface** with chef specific methods like cookFood (), decideMenu () |

**D – Dependency Inversion Principle**

* Class should depend on interfaces rather than concrete classes.

|  |
| --- |
| **Scenario:**  Suppose we have a class “MacBook” with keyboard & mouse dependency to it.   * Using concrete class WiredKeyboard & WiredMouse as dependency.     **Problem with this Approach:**   * By using concrete classes instead of using interfaces like Keyboard, Mouse; we’ve constrained further MacBook upgradation like BluetoothKeyboard, BluetoothKeyboard.   **Solution:**   * By using interface as dependency, MacBook class is capable of handling both Wired & Bluetooth keyboard & Mouse. |