|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Definition** | **Example** | **Theory** |
| **Single Responsibility Principle** | a class should do **one thing** and therefore it should have only a **single reason** to **change** | Email Sending System | In an email sending system, you can apply SRP by having separate classes for composing emails, managing email connections, and sending emails. This ensures that each class has a single responsibility, making it easier to modify and maintain. |
| **Open-Closed Principle** | classes should be **open** for **extension** and **closed** to **modification**. | Payment Gateway Integration | Suppose you have a payment processing system with various payment methods (credit card, PayPal, etc.). Applying OCP, you can create an abstract payment method class and extend it for each payment method. This way, you can add new payment methods without modifying existing code, keeping it open for extension but closed for modification. |
| **Liskov Substitution Principle** | subclasses should be **substitutable** for their base classes. | Shapes Hierarchy | If you have a class hierarchy for shapes (e.g., **Circle**, **Rectangle**, **Triangle**), each subclass should be substitutable for its base class. For example, you should be able to use any shape interchangeably in a function that expects a base **Shape** class instance without causing unexpected behavior. |
| **Interface Segregation Principle** | Many **client- specific interfaces** are **better** than one **general-purpose** interface. | Printing Device Interface | Consider a printing system with different types of printers. Instead of having a monolithic **IPrinter** interface with methods for printing, scanning, and faxing, you can segregate the interfaces into **IPrinter**, **IScanner**, and **IFax** interfaces. This allows classes to implement only the interfaces they need, preventing them from being forced to provide unnecessary methods. |
| **Dependency Inversion Principle** | **classes** should **depend** upon **interfaces** or **abstract classes** instead of **concrete classes** and **functions**. | Vehicle Control System | In a vehicle control system, high-level modules (e.g., **VehicleController**) should not depend on low-level modules (e.g., **Engine** or **Transmission**). Instead, both should depend on abstractions (e.g., **EngineInterface** and **TransmissionInterface**). This allows you to change engine or transmission implementations without modifying the high-level vehicle control code. |