5 design principles intended to make understandable , flexible , robust and maintainable .

To avoid code smelling

S - The single responsibility principle

O- Open close Principle

L – Liskov substitution principle

I – Interface segregation principle

D – Dependency Inversion principle

Graphical user interface, text, application

Description automatically generated

Open Closed Principle

Software Entities (classes ) should open for extension and closed for modification.

Graphical user interface, text, application, email

Description automatically generated

Liskov Substitution Principle

* Derived classes must be able to substitute for their base classes without interrupting the current behaviour of the program
* Text

  Description automatically generated

Interface segregation :

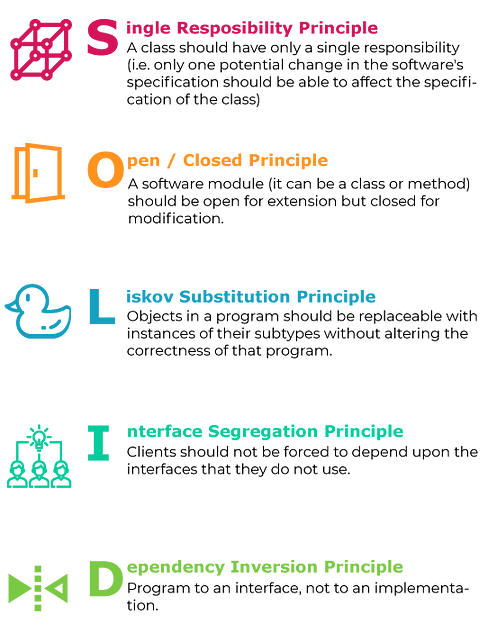
* Client should never be forced to implement an interface that doesn’t use or client shouldn’t be forced to depnednet on methods they do not use
* Text

  Description automatically generated with medium confidence

Dependeny Inversion :

* Entities must depend on abstractions and not on concrete implementation(classes) , also the high level module must no depend on the low-level but both should depend on abstractions

**SOLID Principles in Object Oriented Programming**



The image has been taken from google images

***SOLID*** is an acronym for five design principles that help developers create maintainable and scalable software applications. Here is a brief explanation of each principle with an example in Java:

1. ***Single Responsibility Principle (SRP) :*** A class should have only one reason to change.

*Example:*

public class Employee {  
 private String name;  
 private int salary;  
   
 public void setName(String name) {  
 this.name = name;  
 }  
   
 public void setSalary(int salary) {  
 this.salary = salary;  
 }  
   
 public void save() {  
 // code to save employee data  
 }  
}

In the above example, the Employee class has a single responsibility of managing the employee data. It has methods to set the name and salary of an employee and also to save the employee data to a database.

***2. Open-Closed Principle (OCP) :*** A class should be open for extension but closed for modification.

*Example:*

public interface Shape {  
 public double area();  
}  
  
public class Circle implements Shape {  
 private double radius;  
   
 public Circle(double radius) {  
 this.radius = radius;  
 }  
   
 public double area() {  
 return Math.PI \* radius \* radius;  
 }  
}  
  
public class Rectangle implements Shape {  
 private double length;  
 private double width;  
   
 public Rectangle(double length, double width) {  
 this.length = length;  
 this.width = width;  
 }  
   
 public double area() {  
 return length \* width;  
 }  
}

In the above example, the Shape interface defines the contract for calculating the area of a shape. The Circle and Rectangle classes implement this interface and provide their own implementation of the area() method. If a new shape needs to be added, a new class can be created that implements the Shape interface without modifying the existing Shape, Circle, or Rectangle classes.

*3.****Liskov Substitution Principle (LSP) :*** Subtypes must be substitutable for their base types.

*Example:*

public class Bird {  
 public void fly() {  
 System.out.println("Flying...");  
 }  
}  
  
public class Duck extends Bird {  
 public void quack() {  
 System.out.println("Quacking...");  
 }  
}  
  
public class Ostrich extends Bird {  
 public void run() {  
 System.out.println("Running...");  
 }  
}

In the above example, the Duck and Ostrich classes are subtypes of the Bird class. The Duck class can be substituted for the Bird class in any context without affecting the correctness of the program. However, the Ostrich class violates the LSP as it does not have a fly() method which is expected from a Bird.

***4. Interface Segregation Principle (ISP) :*** Clients should not be forced to depend on methods they do not use.

*Example:*

public interface Printer {  
 public void print();  
}  
  
public interface Scanner {  
 public void scan();  
}  
  
public class MultifunctionPrinter implements Printer, Scanner {  
 public void print() {  
 System.out.println("Printing...");  
 }  
   
 public void scan() {  
 System.out.println("Scanning...");  
 }  
}  
  
public class LaserPrinter implements Printer {  
 public void print() {  
 System.out.println("Printing...");  
 }  
}

In the above example, the MultifunctionPrinter class implements both the Printer and Scanner interfaces while the LaserPrinter class only implements the Printer interface. This allows clients to depend only on the methods they need. For example, if a client only needs to print a document, it can use theLaserPrinterclass which only implements theprint()` method.

***5. Dependency Inversion Principle (DIP) :*** High-level modules should not depend on low-level modules. Both should depend on abstractions.

*Example:*

public interface Notification {  
 public void send(String message);  
}  
  
public class EmailNotification implements Notification {  
 public void send(String message) {  
 // code to send an email  
 }  
}  
  
public class SMSNotification implements Notification {  
 public void send(String message) {  
 // code to send an SMS  
 }  
}  
  
public class NotificationService {  
 private Notification notification;  
   
 public NotificationService(Notification notification) {  
 this.notification = notification;  
 }  
   
 public void sendNotification(String message) {  
 notification.send(message);  
 }  
}

In the above example, the NotificationService class depends on the Notification interface which is an abstraction. This allows different types of notifications (e.g. email, SMS) to be easily added without modifying the NotificationService class. The EmailNotification and SMSNotification classes are low-level modules that implement the Notification interface and can be used interchangeably in the NotificationService class.

In conclusion, the *SOLID* principles provide guidelines for writing clean, maintainable, and scalable code. By following these principles, developers can create software that is easy to understand, modify, and extend. Although implementing all of the SOLID principles may not always be feasible, striving to adhere to them can lead to better software design and development😊.