Introduction

These are the design principles that enable us to manage most of the design problems, intended to design the software design more understandable, flexible, clean, reusable and maintainable.

SOLID is an acronym for

S : Single Responsibility Principle

O : Open and Close Principle

L : Liskov Substitution Principle

I : Interface Segregation Principle

D: Dependency Inversion Principle

# Single Responsibility Principle

A class should do one thing and therefore it should have only a single reason to change.

Let us take the following class Invoice, intended for calculating the invoice.

public class Invoice**{**

public double calculateInvoice**()**

public void printInvoice**()**

public void saveInvoiceToFile**()**

**}**

It is handling 3 responsibilities

calculateInvoice method, which calculates the total price,

printInvoice method, that should print the invoice to console, and

saveToFile method, responsible for writing the invoice to a file.

The first violation is the printInvoice method, which contains our printing logic. The SRP states that our class should only have a single reason to change, and that reason should be a change in the invoice calculation for our class.

There is another method that violates the SRP in our class: the saveToFile method. It is also an extremely common mistake to mix persistence logic with business logic.

We can create new classes for our printing and persistence logic so we will no longer need to modify the invoice class for those purposes.

public class InvoicePrinter **{**

public void printInvoice**()**

**}**

public class InvoicePersistence **{**

public void saveInvoiceToFile**()**

**}**

# Open-Closed Principle

Classes should be open for extension and closed to modification.

Modification means changing the code of an existing class, and extension means adding new functionality.

We should be able to add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs.

But how are we going to add new functionality without touching the class, you may ask. It is usually done with the help of interfaces and abstract classes.

Let's say requirement is invoices to be saved to a database. We might just add saveToDatabase functinality.

But in that case, we are modifying the existing class.

public class InvoicePersistence {

public void saveToFile() {

// Creates a file with given name and writes the invoice

}

public void saveToDatabase() {

// Saves the invoice to database

}

}

Instead, we should need to redesign/refactor the code such that We change the type of InvoicePersistence to Interface and add a save method. Each persistence class will implement this save method.

interface InvoicePersistence {

public void save(Invoice invoice);

}

public class DatabasePersistence implements InvoicePersistence {

@Override

public void save(Invoice invoice) {

// Save to DB

}

}

public class FilePersistence implements InvoicePersistence {

@Override

public void save(Invoice invoice) {

// Save to file

}

}

Now our persistence logic is easily extendable. If our boss asks us to add another database and have 2 different types of databases like MySQL and MongoDB, we can easily do that.

# Liskov Substitution Principle

The Liskov Substitution Principle states that subclasses should be substitutable for their base classes.

This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case.

This is the expected behavior, because when we use inheritance we assume that the child class inherits everything that the superclass has. The child class extends the behavior but never narrows it down.

Liskov's principle is easy to understand but hard to detect in code. So let's look at an example.

class Rectangle {

protected int width, height;

public Rectangle() {

}

public Rectangle(int width, int height) {

this.width = width;

this.height = height;

}

public int getWidth() {

return width;

}

public void setWidth(int width) {

this.width = width;

}

public int getHeight() {

return height;

}

public void setHeight(int height) {

this.height = height;

}

public int getArea() {

return width \* height;

}

}

class Square extends Rectangle {

public Square() {}

public Square(int size) {

width = height = size;

}

@Override

public void setWidth(int width) {

super.setWidth(width);

super.setHeight(width);

}

@Override

public void setHeight(int height) {

super.setHeight(height);

super.setWidth(height);

}

}

Now the following piece of code does not pass the test when we pass Square instance,as it overrides the setHeight(4) to setWidth(5)in the second line.

void test**(**Rectangle r**){**

r**.**setHeight**(**4**);**

r**.**setWidth**(**5**);**

assertEquals**(**4**\***5 **,** r**.**getHeight**()\***r**.**getWidth**())** **;**

**}**

We can handle the Square and Rectangle example using the Liskov Substitution Principle (LSP) by changing the design of the classes. Instead of modeling a square as a subclass of a rectangle, we can model both shapes as separate classes that implement a common interface, such as Shape. This way, we can ensure that the setWidth() and setHeight() methods behave correctly for both shapes, without violating the LSP.

interface Shape {

int getArea();

}

class Rectangle implements Shape {

protected int width;

protected int height;

public void setWidth(int width) {

this.width = width;

}

public void setHeight(int height) {

this.height = height;

}

@Override

public int getArea() {

return width \* height;

}

}

class Square implements Shape {

protected int side;

public void setSide(int side) {

this.side = side;

}

@Override

public int getArea() {

return side \* side;

}

}

public class Main {

public static void main(String[] args) {

Shape r = new Rectangle();

r.setWidth(5);

r.setHeight(10);

System.out.println(r.getArea()); // Output: 50

Shape s = new Square();

s.setSide(5);

System.out.println(s.getArea()); // Output: 25

}

}

# Interface Segregation Principle

The principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do not need.

Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving a specific purpose.

Let us say we have the following

Animal

void feed**();** //abstract

Dog **implements** Animal

void feed**()** //implementation

Tiger **implements** Animal

void feed**()** //implementation

If we want to add one more functionality for animal groom(),it will be suitable for domestic animals like Dog but the wild animals like Tiger have to provide the dummy implementation which is a bad design.

Animal

void feed**();** //abstract

void groom**();** //abstract

Dog **implements** Animal

void feed**()** //implementation

void groom**();** //implementation

Tiger **implements** Animal

void feed**()** //implementation

void groom**();** //DUMMY implementation - to keep compiler happy

Rather than this it's better to create a new interface Pet extending Animal and adding groom method

from where the Dog can provide implementation to it.

Animal

void feed**();** //abstract

Pet **extends** Animal

void groom**();** //abstract

Dog **extends** Pet

void feed**()** //implementation

void groom**();** //implementation

Tiger **extends** Animal

void feed**()** //implementation

# Dependency Inversion Principle

This principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions.

The Open-Closed Principle and Dependency Inversion Principle are interrelated.

We want our classes to be open to extension, so we have reorganized our dependencies to depend on interfaces instead of concrete classes.

(Same example as above Open-Closed Principle)

This principle will enforce the loose coupling/ decoupling of the software.