# Design Principles

Software Design Principles are a set of guidelines that helps developers to make a good system design.

1. SOILD
2. DRY (Don’t Repeat Yourself)
3. KISS (Keep it simple, Stupid!)
4. YAGNI (You ain't gonna need it)

# SOILD

The SOLID Principles are a set of Object-Oriented design principles that have revolutionized how we write software. They are a collection of guidelines and best practices for writing clean, flexible, reusable, scalable, and maintainable code.

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

## Single Responsibility Principle (SRP):

“A class should have one and only one reason to change, meaning that a class should have only one job.”

Example:

class userInfo{

public function getUserName() {

return 'name';

}

public function sendMailToUser() {

return 'success';

}

}

Here userInfo class have two function which do two works, get user name and send email. So it is violate the concept of SRP. So we divided the function into two separate class. Here is also a problem. If we divided each class with each function then, there will be a lot of class which not suitable for maintainable. SRP do not says that each class only one function. There will be a lot of function. But each function do exact same work.

class userInfo{

public function getUserName() {

return 'name';

}

public function getUserAddress() {

return 'address';

}

public function getUserRole(){

return 'role';

}

}

Here userInfo class have 3 method but they all supply user information. So it is maintains SRP states.

## Open-Closed Principle (OCP):

**“classes, modules, and functions should be open for extension but closed for modification.”**

It means that these entities should be implemented in such a way that their functionalities can be extended to other entities without modifying the code of the original entity.

Example:

class Animal {

constructor (type, age,) {

this. type = type;

this. age = age;

}

getSpeed() {

switch (this.type) {

case 'cheetah':

console.log ('Cheetah 130mph ');

break;

case 'lion':

console.log ('Lion 80mph');

break;

default:

throw new Error ();

}

}

}

const animal1 = new Animal('Lion', 4);

animal1.getSpeed();

The code above violates the open-closed principle because if you want to add a new animal type, you have to modify the existing code by adding another case to the switch statement. if you’re using a switch statement, then it’s very likely you will violate the open-closed principle.

Fix the problem:

class Animal {

constructor(name, speedRate) {

this.name = name;

this.speedRate = speedRate;

}

getSpeed() {

return this.speedRate.getSpeed();

}

}

interface A {

getSpeed();

}

class CheetahSpeed extends A {

getSpeed() {

return 130;

}

}

class LionSpeed extends A {

getSpeed() {

return 80;

}

}

const ob1 = new Animal('Cheetah',120 );

ob1.speedrate();

const ob2 = new Animal('Lion', 140);

ob2.speedrate();

If you want to add a new animal type, you can create a new class that extends SpeedRate and pass it to the Animal constructor without modifying the existing code

The Liskov Substitution Principle

*Subtypes must be substitutable for their base types.*