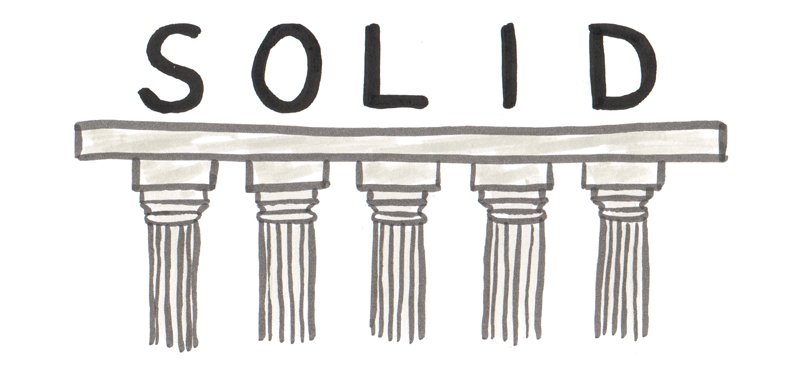
Code and Blog to showcase the understanding of the SOLID principles



# **Introduction**

* The SOLID principles were introduced by **Robert C. Martin** in his 2000 paper “**Design Principles and Design Patterns**” These concepts were later built upon by **Michael Feathers**, who introduced us to the SOLID principles. And in the last two decades, SOLID principles have revolutionized the world of object-oriented programming. This concept changed the way that we write software.
* **Martin** and **Feathers**' design principles motivate us to create more maintainable, understandable, and flexible software. When our applications grow in size, we can reduce their complexity and save ourselves a lot of headaches.
* When we use all the S.O.L.I.D principles in a Software Development , it becomes more simple to develop software that can be managed easily. The other features of using S.O.L.I.D are:
* It avoids code smells.
* Quickly refractor code.
* Can do adaptive or agile software development.
* These principles will help us to reduce tight coupling between classes. Meaning a group of classes that are highly dependent on each other that should be avoided.

# **Benefits of SOLID principle**

The following are the five advantages of the SOLID principles Which play an important role in our software design process.

### 

### **1.Accessibility**

The SOLID Principle ensures easy access to objects, eliminating the risks of unintended inheritance. and control to object entities.

### 

### **2.Ease of refactoring**

As we know Software changes with respect to time. Therefore developers need to build applications with keeping in mind the possibility of future changes in application. If developers will not think about the future possibility then software applications make it difficult to refactor, but it is quite simple and effortless to refactor our code with the SOLID principle.

### **3.Extensibility**

Software goes through phases of upgrades, including extra features. If extending the features of an application is not done, this could affect existing functionalities.But the application of SOLID principles makes the extensibility process easier.

### **4. Debugging**

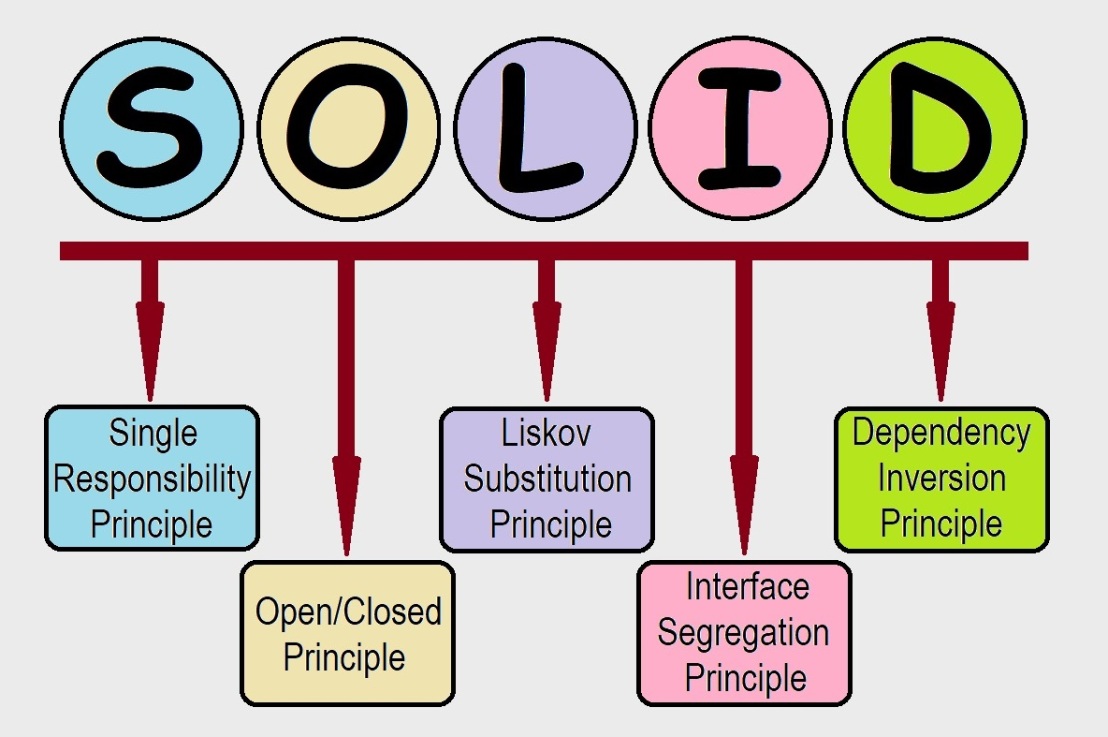
Debugging is an important part of the software development process. When software applications are not well designed, it is hard to debug applications. The SOLID principle makes the debugging process of software much more comfortable.

### **5.Readability**

well-designed code can be easy to understand and read. Readability is also an essential part in software development because it makes the debugging and refactoring operations easier, SOLID principles makes it easy in terms of readability and understanding.

# **SOLID Principles -**

SOLID is a popular set of design principles that are used in object-oriented software development.The SOLID principles were developed by **Robert C. Martin** in a 2000 essay, “**Design Principles and Design Patterns**,” although the acronym was coined later by **Michael Feathers**.Martin acknowledged that successful software will change and develop. It becomes increasingly complex without good design principles, Martin warns that software becomes rigid, fragile, immobile, and viscous.



The goal of the SOLID principles is to reduce dependencies so that engineers change one area of software without impacting others. Additionally, they’re intended to make designs easier to understand, maintain, and extend. While the principles come with many benefits, it makes software so much easier to maintain, test, and extend.

The SOLID principle also helps to save time and effort in both development and maintenance. SOLID principle prevents your code from becoming complex and dependencies, which helps you build long-lasting software.

Solid represents five principles of java which are:

* S: Single responsibility principle
* O: Open-closed principle
* L: Liskov substitution principle
* I: Interface segregation principle
* D: Dependency inversion principle

# **Single Responsibility Principle**

### **What does it say?**

Robert C. Martin describes it as one class should have only one and only responsibility.

According to the single responsibility principle, there should be only one reason due to which a class has to be changed. It means that a class should have one task to do. This principle is often termed as subjective.

The principle can be well understood with an example. Imagine there is a class which performs the following operations.

* Connected to a database
* Read some data from database tables
* Finally, write it to a file.

Have you imagined the scenario? Here the class has multiple reasons to change, and few of them are the modification of file output, new database adoption. When we are talking about single principle responsibility, we would say, there are too many reasons for the class to change; hence, it doesn’t fit properly in the single responsibility principle.

### **Why is that this Principle is Required?**

When the Single Responsibility Principle is followed, testing is easier. With a single responsibility, the class will have fewer test cases. Less functionality also means fewer dependencies to other classes. It leads to better code organisation since smaller and well-purposed classes are easier to search.

### **Here is the practical example -**

package Single\_Responsibility\_Principle;

public class Intern

{

String name;

String company;

public Intern()

{

name="MuZakkkir Saifi";

company="Knoldus Inc";

}

}

package Single\_Responsibility\_Principle;

public class Employee\_ID {

public String name;

int emp;

String salary;

public Employee\_ID()

{

emp=1621;

salary="in Rupees";

}

}

package Single\_Responsibility\_Principle;

public class Joined

{

public String name;

String date="19th July 2021";

public void onboarding(String name, String n)

{

String company = "Knoldus Inc";

int emp=1621;

System.*out*.println("I am from Single Responsibility and here to let you that");

System.*out*.println(name+" has joined "+company+" on "+date+" And his Employee ID is "+emp);

}

public static void main(String[] args) {

Intern I=new Intern();

Employee\_ID E=new Employee\_ID();

Joined J=new Joined();

J.onboarding(I.name,J.name);

}

}

# **Open Closed Principle**

in JavaRobert C. Martin describes it as Software components should be open for extension, but closed for modification.

To be precise, according to this principle, a class should be written in such a manner that it performs its job flawlessly without the assumption that people in the future will simply come and change it. Hence, the class should remain closed for modification, but it should have the option to get extended. Ways of extending the class include:

* Inheriting from the class
* Overwriting the required behaviors from the class
* Extending certain behaviors of the class

An excellent example of open-closed principle can be understood with the help of browsers. Do you remember installing extensions in your chrome browser?

Basic function of the chrome browser is to surf different sites. Do you want to check grammar when you are writing an email using chrome browser?If yes, you can simply use Grammarly extension, it provides you grammar check on the content.

This mechanism where you are adding things for increasing the functionality of the browser is an extension. Hence, the browser is a perfect example of functionality that is open for extension but is closed for modification. In simple words, you can enhance the functionality by adding/installing plugins on your browser, but cannot build anything new.

### **Why is this principle required?**

OCP is important since classes may come to us through third-party libraries. We should be able to extend those classes without worrying if those base classes can support our extensions. But inheritance may lead to sub classes which depend on base class implementation. To avoid this, use of interfaces is recommended. This additional abstraction leads to loose coupling.

### **Here is the Practical example -**

package Open\_Closed\_Principle;

public class Intern

{

String name;

String company;

public Intern()

{

name="Mohd MuZakkir Saifi";

company="Knoldus Inc";

}

}

package Open\_Closed\_Principle;

public class Emp\_ID extends Intern {

int emp;

String salary;

public Emp\_ID()

{

emp=1621;

salary="is in rupees";

}

}

package Open\_Closed\_Principle;

public class Joined extends Emp\_ID {

String date = "19th July 2021";

void onboarding(String name, String n) {

String company = "Knoldus Inc";

int emp = 1621;

System.*out*.println("I am from Open Closed Principle.");

System.*out*.println("Hi all Knolders, "+name+ " has joined " + company + " on " + date);

System.*out*.println("And his Employee ID is " + emp+ " and salary "+salary);

}

public static void main(String[] args) {

Intern I = new Intern();

Emp\_ID E = new Emp\_ID();

Joined J = new Joined();

J.onboarding(J.name, E.name);

}

}

# **Liskov Substitution Principle**

Robert C. Martin describes it as Derived types must be completely substitutable for their base types.

Liskov substitution principle assumes q(x) to be a property, provable about entities of x which belong to type T. Now, according to this principle, the q (y) should be now provable for objects y that belong to type S, and the S is actually a subtype of T. Are you now confused and don’t know what Liskov substitution principle actually means? The definition of it might be a bit complex, but in fact, it is quite easy. The only thing is that every subclass or derived class should be substitutable for their parent or base class.

You can say that it is a unique object-oriented principle. The principle can further be simplified by ; a child type of a particular parent type without making any complication or blowing things up should have the ability to stand in for that parent.This principle is closely related to Liskov Substitution principle.

### **Why is this principle required?**

This avoids misusing inheritance. It helps us conform to the “is-a” relationship.We can also say that subclasses must fulfill a contract defined by the base class. In this sense, it’s related to Design by Contract that was first described by Bertrand Meyer. For example, it’s tempting to say that a circle is a type of ellipse but circles don’t have two foci or major/minor axis.

### **Here is the Practical example -**

package Liskov\_Substitution\_Principle;

interface Animal{

static void makeNoise() {

System.*out*.println("Dog can making noise when someone try to enter in the house");

}

}

class Cat implements Animal {

public static void makeNoise() {

System.*out*.println("Cat can eat Non-veg");

}

}

class Dog implements Animal {

public static void makeNoise()

{

System.*out*.println("Dog can eat Veg as well as Non-Veg");

}

}

class DumbDog implements Animal{

public static void main(String[] args) {

{

System.*out*.println("I am from Liskov\_Substitution Principle");

Dog.*makeNoise*();

Animal.*makeNoise*();

Cat.*makeNoise*();

}

}

}

.