|  |  |
| --- | --- |
| **Principle** | **Description** |
| **Single Responsibility Principle** | Each class should be responsible for a single part or functionality of the system |
|  |  |
| **Open-Closed Principle** | Software components should be open for extension, but not for modification. |
| **Liskov Substitution Principle** | Objects of a superclass should be replaceable with objects of its subclasses without breaking the system. |
|  |  |
| **Interface Segregation Principle** | No client should be forced to depend on methods that it does not use. |
|  |  |
| **Dependency Inversion Principle** | High-level modules should not depend on low-level modules, both should depend on abstractions. |

SOLID is a structured desig

You are here

Home > [Design Principles](https://javatechonline.com/design-principles/) >

**SOLID Principles-The Single Responsibility Principle**

* [Design Principles](https://javatechonline.com/design-principles/)
* [Core Java](https://javatechonline.com/java/core-java/)
* [java](https://javatechonline.com/java/)

*by*[*devs5003*](https://javatechonline.com/author/erdsingh24/)*- May 12, 2023*3

Last Updated on November 17th, 2023

Almost everywhere when we talk about delivery of a product, the first step comes in mind is it’s design. The more focus we put into the design, the better the product will look. Every design has some design principles that need to be followed while designing a product.  Hence, design principles have a crucial role in any product delivery. Design Principles help teams with decision making. In this article, We will discuss about ‘SOLID Principles-The Single Responsibility Principle’.

A few simple principles or valuable questions can guide our team towards taking relevant decisions. SOLID Principles are the set of five principles used to design a software. In fact, the word ‘SOLID’ is the acronym for the set of five principles that contains the first letter of each principle. Robert Martin (Uncle Bob) has introduced the SOLID Design Principles.

Let’s discuss SOLID Principles-The Single Responsibility Principle in detail and related concepts.

Table of Contents

* [Design Principles vs Design Patterns](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Design_Principles_vs_Design_Patterns)
* [What is Code Rot?](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#What_is_Code_Rot)
* [How can we identify future Code Rot?](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#How_can_we_identify_future_Code_Rot)
  + [Rigidity](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Rigidity)
  + [Fragility](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Fragility)
  + [Immobility](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Immobility)
  + [Viscosity](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Viscosity)
* [What are SOLID Principles?](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#What_are_SOLID_Principles)
* [What are the benefits of using SOLID Design Principles?](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#What_are_the_benefits_of_using_SOLID_Design_Principles)
* [SOLID Principles-The Single Responsibility Principle](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#SOLID_Principles-The_Single_Responsibility_Principle)
  + [Example#1: Code that violates Single Responsibility Principle](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Example1_Code_that_violates_Single_Responsibility_Principle)
  + [Example#1: Code that follows the Single Responsibility Principle](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Example1_Code_that_follows_the_Single_Responsibility_Principle)
  + [Class Diagram of Code](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Class_Diagram_of_Code)
  + [Example#2: Code that violates the Single Responsibility Principle](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Example2_Code_that_violates_the_Single_Responsibility_Principle)
  + [Example#2: Code that follows the Single Responsibility Principle](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Example2_Code_that_follows_the_Single_Responsibility_Principle)
* [Other Examples](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Other_Examples)
  + [EntityManager](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#EntityManager)
* [What is the benefit of the Single Responsibility Principle?](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#What_is_the_benefit_of_the_Single_Responsibility_Principle)
  + [Easier to Test](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Easier_to_Test)
  + [Lower coupling](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Lower_coupling)
  + [Better Organization](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Better_Organization)
  + [Easier to implement](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Easier_to_implement)
  + [Easier to maintain](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Easier_to_maintain)
* [Links to Other Design Principles](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Links_to_Other_Design_Principles)
  + [Open Closed Principle(OCP)](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Open_Closed_PrincipleOCP)
  + [Liskov’s Substitution Principle(LSP)](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Liskovs_Substitution_PrincipleLSP)
  + [Interface Segregation Principle(ISP)](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Interface_Segregation_PrincipleISP)
  + [Dependency Inversion Principle(DIP)](https://javatechonline.com/solid-principles-the-single-responsibility-principle/#Dependency_Inversion_PrincipleDIP)

ADVERTISEMENT

[Robopet Make Your Robot Follow You](https://javatechonline.com/solid-principles-the-single-responsibility-principle/)00:00/03:14

[](https://www.cloudways.com/en/home.php?id=1101896&a_bid=fc03401b)

**Design Principles vs Design Patterns**

Design principles are a set of generalized pieces of advice or proven good coding practices that are used as rules of thumb when making design choices. They’re a bit similar concept to design patterns. The main difference is that Design Principles are more abstract and generalized. They are kind of high-level pieces of advice, often applicable to many different programming languages or even different paradigms.

Design Patterns are also abstractions or generalized good practices, but they provide much more concrete and practical low-level advice. However, they are related to entire classes of problems rather than just generalized coding practices.

Design Principles encourage us to create more maintainable, understandable, and flexible software. Consequently, as our applications grow in size, we can reduce their complexity and save ourselves a lot of headaches further down the road!

***Software Design is not perfect if it is executed in a single attempt. There is always a possibility to change it and make it even better. Furthermore, your software must be able to keep up with the changing needs of your customer.***

**What is Code Rot?**

**When developers work on design without using structured design principles, they can create long-lasting problems for future developers working on the project, and limit the potential success for the application they are developing. These issues are commonly referred to as “code rot.”**

If you want to change your programs during any upgradation of the software, you should avoid the factors that can cause the code rot to happen in the future. Hence, you can make your software up to date and the people can continue to use it for years to come.

**How can we identify future Code Rot?**

Below signs probably indicate code rot to come in the future:

**Rigidity**

Small change causes the entire system to rebuild.

**Fragility**

Changes to one module causes other unrelated modules to misbehave.

**Immobility**

A module’s internal components cannot be extracted and reused in new environments. For example, if an application’s login module cannot be used in entirely different system then this module is immobile, caused by couplings and dependencies between different modules. The strategy is to decouple central abstractions from low-level details, like a particular database schema or UI implementation (web, desktop) or specific frameworks.

**Viscosity**

When building and testing are difficult to perform and take a long time to execute. When even a simple change is costly to make, and requires you to make changes in multiple places/levels.

**What are SOLID Principles?**

In the world of object-oriented programming (OOP), there are many design guidelines, patterns or principles. Five of these principles are usually grouped together and are known by the acronym SOLID. While each of these five principles describes something specific, they overlap as well, such that adopting one of them implies or leads to adopting another.

**S** **⇒ stands for Single Responsibility Principle(SRP)**

**O** **⇒ stands for Open Closed Principle(OCP)**

**L ⇒ stands for Liskov’s Substitution Principle(LSP)**

**I** **⇒ stands for Interface Segregation Principle(ISP)**

**D** **⇒ stands for Dependency Inversion Principle(DIP)**

The SOLID principles were first conceptualized 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 acronym. And in the last 20 years, these 5 principles have revolutionized the world of object-oriented programming, changing the way that we write software.

Besides monolithic apps, you can also apply SOLID design principles to microservices where you can treat each microservice as a standalone code module.

When you use all the principles of S.O.L.I.D in a combined manner, it becomes easier for you to develop software that can be managed easily. The other features of using S.O.L.I.D are:

* It avoids code smells.
* Quickly refactor code.
* Can do adaptive or agile software development.

Now it’s time to understand SOLID Principles in much detail with the help of various examples.

**What are the benefits of using SOLID Design Principles?**

Here are some of the most important purposes of following SOLID Design Principles in our code.

1) To make it easier to quickly extend the system with new functionality without breaking the existing ones. If we don’t do implementation of a new feature in the application sensibly, It may even affect the existing functionality and cause unintentional issues.

2) To make the code easier to read and understand. Thus, spend less time figuring out what it does. If our code doesn’t follow the SOLID design principles, we generally take more time to understand what it does. The SOLID principle approach ensures that your code is relatively easier to read and understand.

3) To make the code more maintainable.

4) Every software needs upgradation time to time. Therefore, we need to build applications while keeping in mind the possibility of future changes. It is quite effortless to refactor your code with the SOLID principles.

D :

We have a Car class that depends on the concrete Engine class; therefore, it is not obeying DIP.

public class Engine {  
    public Engine() {  
    }  
    public void start() {  
        System.out.println(“Engine Started”);  
    }  
}

public class Car {  
    private Engine engine;  
    public Car(Engine e) {  
        engine = e;  
    }  
    public void startCar() {  
        engine.start();  
    }  
}

public class DriverApp

{

    public static void main(String[] args)

{

        Engine e1 = new Engine();

        Car c1 = new Car(e1);

        c1.startCar();

}

}

 but what if we wanted to add another engine type, let’s say a diesel engine? This will require refactoring the Car class.

Solution:

solve this by introducing a layer of abstraction. Instead of Car depending directly on Engine, let’s add an interface:

public interface Engine {  
    public void start();  
}

*// Class for PetrolEngine*

class PetrolEngine implements Engine {

    public void start() {

        System.out.println("Petrol engine started.");

    }

}

*// Class for PetrolEngine*

class DieselEngine implements Engine {

    public void start() {

        System.out.println("Diesel engine started.");

    }

}

class Car {

    private Engine engine;

    public Car(Engine e) {

        this.engine = e;

    }

    public void startCar() {

        engine.start();

    }

}

public class Main {

    public static void main(String[] args) {

        Engine petrolEngine = new PetrolEngine();

        Engine dieselEngine = new DieselEngine();

        Car petrolCar = new Car(petrolEngine);

        Car dieselCar = new Car(dieselEngine);

        petrolCar.startCar();

        dieselCar.startCar();

    }

}