**SOLID Principle in Programming: Understand With Real Life Examples**

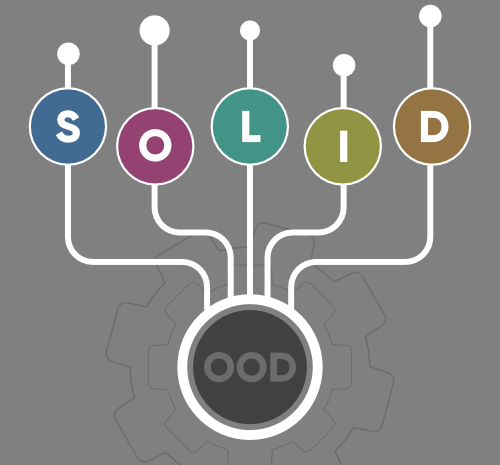
In software development, **Object-Oriented Design** plays a crucial role when it comes to writing flexible, scalable, maintainable, and reusable code. There are so many benefits of using OOD but every developer should also have the knowledge of the SOLID principle for good object-oriented design in programming. The SOLID principle was introduced by ***Robert C. Martin***, also known as *Uncle Bob* and it is a coding standard in programming. This principle is an acronym of the five principles which is given below…

In object-oriented programming languages, the classes are the building blocks of any application. If these blocks are not strong, the building (i.e. the application) is going to face a tough time in the future.

Poorly designed applications can lead the team to very difficult situations when the application scope goes up, or the implementation faces certain design issues either in production or maintenance.

On the other hand, a set of well designed and written classes can speed up the coding process, while reducing the tech debt and the number of bugs in comparison.

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



The SOLID principle helps in reducing tight coupling. Tight coupling means a group of classes are highly dependent on one another which you should avoid in your code. Opposite of tight coupling is loose coupling and your code is considered as a good code when it has loosely-coupled classes. Loosely coupled classes minimize changes in your code, helps in making code more reusable, maintainable, flexible and stable. Now let’s discuss one by one these principles…

1. **Single Responsibility Principle:**This principle states that “*a class should have only one reason to change*” which means every class should have a single responsibility or single job or single purpose. Take the example of developing software. The task is divided into different members doing different things as front-end designers do design, the tester does testing and backend developer takes care of backend development part then we can say that everyone has a single job or responsibility.  
   Most of the time it happens that when programmers have to add features or new behavior they implement everything into the existing class which is completely wrong. It makes their code lengthy, complex and consumes time when later something needs to be modified. Use *layers* in your application and break God classes into smaller classes or modules.

## Single Responsibility Principle

We may come across one of the principles of object-oriented design, **Separation of Concerns (SoC)**, that conveys a similar idea. The name of the SRP says it all:

“One class should have one and only one responsibility”

In other words, we should write, change, and maintain a class only for one purpose. A class is like a container. We can add any amount of data, fields, and methods into it. However, if we try to achieve too much through a single class, soon that class will become bulky. If we follow SRP, the classes will become compact and neat where each class is responsible for a single problem, task, or concern.

For example, if a given class is a model class then it should strictly represent only one actor/entity in the application. This kind of design decision will give us the flexibility to make changes in the class, in future without worrying the impacts of changes in other classes.

Similarly, If we are writing service/manager class then the class should contain only that part of methods and nothing else. The service class should not contain even utility global functions related to the module.

Better to separate the global functions in another globally accessible class. This will help in maintaining the class for that particular purpose, and we can decide the visibility of class to a specific module only.

#### Example

We can find plenty of classes in all popular Java libraries which follow single responsibility principle. For example, in [Log4j2](https://howtodoinjava.com/log4j2/log4j2-json-configuration-example/), we have different classes with logging methods, different classes are logging levels and so on.

In our application level code, we define model classes to represent real time entities such as Person, Employee, Account etc. Most of these classes are examples of SRP principle because when we need to change the state of a person, only then we will modify the Person class, and so on.

In the given example, we have two classes Person and Account. Both have single responsibility to store their specific information. If we want to change the state of Person then we do not need to modify the class Account and vice-versa.

|  |
| --- |
| Person.java |
| public class Person  {    private Long personId;    private String firstName;    private String lastName;    private String age;    private List<Account> accounts;  } |
| Account.java |
| public class Account  {    private Long guid;    private String accountNumber;    private String accountName;    private String status;    private String type;  } |

1. **Open/Closed Principle:**This principle states that “*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*” which means you should be able to extend a class behavior, without modifying it.  
   Suppose developer A needs to release an update for a library or framework and developer B wants some modification or add some feature on that then developer B is allowed to extend the existing class created by developer A but developer B is not supposed to modify the class directly. Using this principle separates the existing code from the modified code so it provides better stability, maintainability and minimizes changes as in your code.

OCP is the second principle which we should keep in mind while designing our application. It states:

“Software components should be open for extension, but closed for modification”

It means that the application classes should be designed in such a way that whenever fellow developers want to change the flow of control in specific conditions in application, all they need to extend the class and override some functions and that’s it.

If other developers are not able to write the desired behavior due to constraints put by the class, then we should reconsider refactoring the class.

I do not mean here that anybody can change the whole logic of the class, but one should be able to override the options provided by software in a non harmful way permitted by the software.

#### Example

If we take a look into any good framework like struts or spring, we will see that we can not change their core logic and request processing, but we modify the desired application flow just by extending some classes and plugin them in configuration files.

For example, spring framework has class [DispatcherServlet](https://howtodoinjava.com/spring5/webmvc/spring-dispatcherservlet-tutorial/" \t "_blank). This class acts as a **front controller** for String based web applications. To use this class, we are not required to modify this class. All we need is to pass initialization parameters and we can extend its functionality the way we want.

Please note that apart from passing initialization parameters during application startup, we can override methods as well to modify the behavior of target class by extending the classes. For example, struts Action classes are extended to override the request processing logic.

|  |
| --- |
| Extending Struts Action |
| public class HelloWorldAction extends Action  {  @Override    public ActionForward execute(ActionMapping mapping,        ActionForm form,        HttpServletRequest request,        HttpServletResponse response)        throws Exception    {      //Process the request      }  } |

1. **Liskov’s Substitution Principle:**The principle was introduced by Barbara Liskov in 1987 and according to this principle “*Derived or child classes must be substitutable for their base or parent classes*“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

“Derived types must be completely substitutable for their base types”

LSP means that the classes, fellow developers created by extending our class, should be able to fit in application without failure. This is important when we resort to [polymorphic behavior](https://howtodoinjava.com/java/oops/what-is-polymorphism-in-java/) through [inheritance](https://howtodoinjava.com/java/oops/java-inheritance/).

This requires the objects of the subclasses to behave in the same way as the objects of the superclass. This is mostly seen in places where we do runtime type identification and then cast it to appropriate reference type.

#### Example

An example of LSP can be [custom property editors](https://howtodoinjava.com/spring-boot/custom-property-editor-example/) in Spring framework. Spring provides property editors to represent properties in a different way than the object itself e.g. parsing human readable inputs from HTTP request parameters or displaying human readable values of pure java objects in view layer e.g. Currency or URL.

Spring can register one property editor for one data type and it is required to follow the constraint mandated by **base class** PropertyEditorSupport. So if any class extends PropertyEditorSupport class, then it can be substituted by everywhere the base class is required.

For example, every book has an ISBN number which is always in a fixed display format. You can have separate representations of ISBN in database and UI. For this requirement, we may write property editor in such a way –

|  |
| --- |
| IsbnEditor.java |
| import java.beans.PropertyEditorSupport;  import org.springframework.util.StringUtils;  import com.howtodoinjava.app.model.Isbn;    public class IsbnEditor extends PropertyEditorSupport {      @Override      public void setAsText(String text) throws IllegalArgumentException {          if (StringUtils.hasText(text)) {              setValue(new Isbn(text.trim()));          } else {              setValue(null);          }      }        @Override      public String getAsText() {          Isbn isbn = (Isbn) getValue();          if (isbn != null) {              return isbn.getIsbn();          } else {              return "";          }      }  } |

You can understand it in a way that a farmer’s son should inherit farming skills from his father and should be able to replace his father if needed. If the son wants to become a farmer then he can replace his father but if he wants to become a cricketer then definitely the son can’t replace his father even though they both belong to the same family hierarchy.  
One of the classic examples of this principle is a rectangle having four sides. A rectangle’s height can be any value and width can be any value. A square is a rectangle with equal width and height. So we can say that we can extend the properties of the rectangle class into square class. In order to do that you need to swap the child (square) class with parent (rectangle) class to fit the definition of a square having four equal sides but a derived class does not affect the behavior of the parent class so if you will do that it will violate the Liskov Substitution Principle. Check the link [Liskov Substitution Principle](https://www.youtube.com/watch?v=Jecou7B3nhc" \t "_blank) for better understanding.

1. **Interface Segregation Principle:**This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that “*do not force any client to implement an interface which is irrelevant to them*“. Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

This principle is my favorite one. ISP is applicable to interfaces as a single responsibility principle holds to classes. ISP says:

“Clients should not be forced to implement unnecessary methods which they will not use”

Take an example. Developer Alex created an interface Reportable and added two methods generateExcel() and generatedPdf(). Now client ‘A’ wants to use this interface but he intends to use reports only in PDF format and not in excel. Will he be able to use the functionality easily?

NO. He will have to implement both the methods, out of which one is an extra burden put on him by the designer of the software. Either he will implement another method or leave it blank. This is not a good design.

So what is the solution? Solution is to create two interfaces by breaking the existing one. They should be like PdfReportable and ExcelReportable. This will give the flexibility to users to use only the required functionality only.

#### Example

The best place to look for IPS examples is Java AWT event handlers for handling GUI events fired from keyboard and mouse. It has different listener classes for each kind of event. We only need to write handlers for events, we wish to handle. Nothing is mandatory.

Some of the listeners are –

* FocusListener
* KeyListener
* MouseMotionListener
* MouseWheelListener
* TextListener
* WindowFocusListener

Anytime, we wish to handle any event, just find out a corresponding listener and implement it.

|  |
| --- |
| MouseMotionListenerImpl.java |
| public class MouseMotionListenerImpl implements MouseMotionListener  {  @Override    public void mouseDragged(MouseEvent e) {      //handler code    }     @Override    public void mouseMoved(MouseEvent e) {      //handler code    }  } |

## Dependen

Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets. In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don’t eat in your food. Here the menu should be different for different types of customers. The common or general menu card for everyone can be divided into multiple cards instead of just one. Using this principle helps in reducing the side effects and frequency of required changes.

**5. Dependency Inversion Principle:**Before we discuss this topic keep in mind that Dependency Inversion and [Dependency Injection](https://en.wikipedia.org/wiki/Dependency_injection) both are different concepts. Most of the people get confused about it and consider both are the same. Now two key points are here to keep in mind about this principle

* High-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions.
* Abstractions should not depend upon details. Details should depend upon abstractions.

The above lines simply state that if a high module or class will be dependent more on low-level modules or class then your code would have tight coupling and if you will try to make a change in one class it can break another class which is risky at the production level. So always try to make classes loosely coupled as much as you can and you can achieve this through *abstraction*. The main motive of this principle is decoupling the dependencies so if class A changes the class B doesn’t need to care or know about the changes.

Most of us are already familiar with the words used in principle’s name. [DI principle](https://howtodoinjava.com/spring-core/spring-ioc-vs-di/) says:

“Depend on abstractions, not on concretions”

In other words. we should design our software in such a way that various modules can be separated from each other using an abstract layer to bind them together.

#### Example

The classical use of this principle of bean configuration in Spring framework.

In the spring framework, all modules are provided as separate components which can work together by simply injecting dependencies in other modules. This dependency is managed externally in XML files.

These separate components are so well closed in their boundaries that we can use them in other software modules apart from spring with the same ease. This has been achieved by dependency inversion and open closed principles. All modules expose only abstraction which is useful in extending the functionality or plug-in in another module.

You can consider the real-life example of a TV remote battery. Your remote needs a battery but it’s not dependent on the battery brand. You can use any XYZ brand that you want and it will work. So we can say that the TV remote is loosely coupled with the brand name. Dependency Inversion makes your code more reusable.