# [SOLID:](#_bpsrngj7dzjl) [Eg: Design a Pen](https://medium.com/@midhinmurali3/s-o-l-i-d-the-five-principles-of-guidelines-for-software-engineers-1939fcc96b40)

SOLID is an acronym for a set of design principles in object-oriented programming. The five principles of SOLID are

1. Single Responsibility Principle (SRP): Every code unit (class/method) should have only one reason to change, meaning it should have only one responsibility or job. This principle helps keep code modular and easier to maintain.
2. Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that new functionality should be added by creating new classes, rather than changing existing code.
3. Liskov Substitution Principle (LSP): Subtypes should be substitutable for their base types. This means that a subclass should be able to replace its parent class without affecting the correctness of the program.
4. Interface Segregation Principle (ISP): A client should not be forced to depend on methods it does not use. This principle helps keep interfaces focused and specific to their purpose.
5. Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions. This principle helps decouple modules and make them more modular and flexible.

## Single Responsibility Principle (SRP):

The Single Responsibility Principle (SRP) states that every class or module in a software system should have only one responsibility, i.e., only one reason to change. This means that a class or module should have a well-defined and narrow purpose, and it should not be responsible for multiple unrelated tasks.

A good example of SRP is the separation of concerns in a web application. The frontend code is responsible for handling user interface and user interactions, while the backend code is responsible for handling data storage, retrieval, and business logic. By separating these responsibilities into different modules or classes, it becomes easier to maintain and modify the code in the future. For instance, if you want to change the design of the user interface, you don't have to worry about breaking the data storage and business logic in the backend.

Another example of SRP is the division of labor in a company. Each employee has a specific role or responsibility, and they are expected to perform that role to the best of their abilities. For instance, the marketing team is responsible for promoting the company's products or services, while the sales team is responsible for closing deals with customers. If employees were expected to perform multiple roles or responsibilities, it would lead to confusion and decreased productivity.

In summary, SRP helps to improve the maintainability, flexibility, and modularity of software systems by ensuring that each module or class has a clear and distinct responsibility.

### Example 1:

Suppose you are building a banking application, and you need to create a class to handle customer transactions. The class could be named *TransactionHandler*.

A good way to apply SRP to this class would be to give it only one responsibility: handling transactions. Here's an example implementation:

| public class TransactionHandler {  public void handleTransaction(Transaction transaction) {  // handle the transaction here  } } |
| --- |

In this example, the *TransactionHandler* class has a single responsibility, which is to handle transactions. This means that if you need to modify the transaction handling logic, you only need to change the *handleTransaction* method, and not any other parts of the class.

However, suppose you also need to send an email confirmation to the customer after a transaction is processed. You could be tempted to add this functionality to the *TransactionHandler* class, but that would violate SRP. Instead, you should create a separate class to handle email confirmation, like this:

| public class EmailSender {  public void sendEmail(String recipient, String subject, String body) {  // send the email here  } } |
| --- |

Then, you could modify the *TransactionHandler* class to use the *EmailSender* class to send the email confirmation:

| public class TransactionHandler {  private EmailSender emailSender;   public TransactionHandler(EmailSender emailSender) {  this.emailSender = emailSender;  }   public void handleTransaction(Transaction transaction) {  // handle the transaction here   // send email confirmation to the customer  emailSender.sendEmail(transaction.getCustomerEmail(), "Transaction confirmation", "Your transaction has been processed successfully.");  } } |
| --- |

In this updated implementation, the *TransactionHandler* class still has a single responsibility, which is to handle transactions. The email confirmation logic has been delegated to the *EmailSender* class, which has its own responsibility for sending emails. This adheres to the Single Responsibility Principle and makes the code easier to understand, maintain, and test.

### Example 2:

Suppose you are developing a payroll application that calculates and pays employee salaries. You could have a *PayrollService* class that handles all the payroll processing, including calculating salaries, deducting taxes, and generating pay stubs. However, this class would have multiple responsibilities and violate the Single Responsibility Principle.

To apply SRP to this scenario, you could break down the responsibilities into separate classes. For example, you could have a *SalaryCalculator* class that calculates the gross pay and net pay for each employee based on their hourly rate and number of hours worked. Here's an example implementation:

| public class SalaryCalculator {  public double calculateGrossPay(double hourlyRate, int hoursWorked) {  return hourlyRate \* hoursWorked;  }   public double calculateNetPay(double grossPay, double taxRate) {  return grossPay \* (1 - taxRate);  } |
| --- |

}

In this example, the *SalaryCalculator* class has a single responsibility, which is to calculate the gross and net pay for an employee based on their hourly rate and hours worked. This means that if you need to modify the salary calculation logic, you only need to change the *calculateGrossPay* and *calculateNetPay* methods, and not any other parts of the class.

Next, you could have a *PayStubGenerator* class that generates pay stubs for each employee. Here's an example implementation:

| public class PayStubGenerator {  public String generatePayStub(String employeeName, double grossPay, double netPay) {  StringBuilder sb = new StringBuilder();  sb.append("Pay Stub for " + employeeName + "\n");  sb.append("Gross Pay: $" + grossPay + "\n");  sb.append("Net Pay: $" + netPay + "\n");  return sb.toString();  } } |
| --- |

In this example, the *PayStubGenerator* class has a single responsibility, which is to generate pay stubs for each employee based on their gross pay and net pay. This means that if you need to modify the pay stub generation logic, you only need to change the *generatePayStub* method, and not any other parts of the class.

Finally, you could have a *PayrollService* class that coordinates the different responsibilities of the payroll processing. Here's an example implementation:

| public class PayrollService {  private SalaryCalculator salaryCalculator;  private PayStubGenerator payStubGenerator;   public PayrollService(SalaryCalculator salaryCalculator, PayStubGenerator payStubGenerator) {  this.salaryCalculator = salaryCalculator;  this.payStubGenerator = payStubGenerator;  }   public String processPayroll(Employee employee) {  double grossPay = salaryCalculator.calculateGrossPay(employee.getHourlyRate(), employee.getHoursWorked());  double netPay = salaryCalculator.calculateNetPay(grossPay, employee.getTaxRate());  String payStub = payStubGenerator.generatePayStub(employee.getName(), grossPay, netPay);  return payStub;  } } |
| --- |

In this example, the *PayrollService* class has a single responsibility, which is to coordinate the salary calculation and pay stub generation for each employee. The actual calculations and generation are delegated to the *SalaryCalculator* and *PayStubGenerator* classes, respectively.

## Open-Close Principle:

The open-Closed Principle is a design principle in object-oriented programming that states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to add new functionality to a class or module without changing its existing code.

### Implementation Guidelines:

* The simplest way to apply OCP is to implement the new functionality on new derived classes
* Allow clients to access the original class with an abstract interface

### Example 1:

| public interface Shape {  double area(); }  public class Rectangle implements Shape {  private double width;  private double height;    public Rectangle(double width, double height) {  this.width = width;  this.height = height;  }    public double area() {  return width \* height;  } }  public class Circle implements Shape {  private double radius;    public Circle(double radius) {  this.radius = radius;  }    public double area() {  return Math.PI \* radius \* radius;  } }  public class AreaCalculator {  public double totalArea(Shape[] shapes) {  double area = 0;  for (Shape shape : shapes) {  area += shape.area();  }  return area;  } } |
| --- |

In this example, we have two classes that implement the Shape interface: Rectangle and Circle. Each class has its own implementation of the area() method.

The AreaCalculator class takes an array of Shape objects and calculates the total area by iterating over each object and calling its area() method. Notice that the AreaCalculator class doesn't care about the specific implementation of the area() method, it just knows that it can call it on any object that implements the Shape interface.

Now suppose we want to add a new shape to our program, such as a Triangle. We can simply create a new class that implements the Shape interface and provides its own implementation of the area() method. We don't have to modify the existing code in Rectangle, Circle, or AreaCalculator.

This is an example of the Open-Closed Principle in action. Our code is open for extension because we can add new shapes without modifying the existing code. At the same time, our code is closed for modification, because we don't have to change the existing code to add new functionality.

### Example 2:

Suppose we have a class called Car which has a method start() that starts the engine of the car. Initially, the start() method directly interacts with the car's engine to start it:

| public class Car {  private Engine engine;    public Car(Engine engine) {  this.engine = engine;  }    public void start() {  engine.start();  } }  public class Engine {  public void start() {  System.out.println("Engine started");  } } |
| --- |

Now suppose we want to add a new feature where we can start the car remotely using a smartphone app. One way to add this feature would be to modify the start() method in the Car class to add remote start functionality:

| public interface Starter {  void start(); }  public class Car implements Starter {  private Engine engine;    public Car(Engine engine) {  this.engine = engine;  }    public void start() {  engine.start();  } }  public class Engine implements Starter {  public void start() {  System.out.println("Engine started");  } }  public class RemoteStarter implements Starter {  private Starter starter;    public RemoteStarter(Starter starter) {  this.starter = starter;  }    public void start() {  System.out.println("Starting car remotely");  starter.start();  } } |
| --- |

This modification violates the Open-Closed Principle because we had to modify the existing code in the Car class to add new functionality. If we want to add more ways to start the car in the future, we will have to keep modifying the start() method.

A better way to implement the remote start feature while adhering to the Open-Closed Principle is to introduce a new interface called Starter that defines the start() method, and create a new class called RemoteStarter that implements this interface and provides its own implementation of the start() method:

| public interface Starter {  void start(); }  public class Car implements Starter {  private Engine engine;    public Car(Engine engine) {  this.engine = engine;  }    public void start() {  engine.start();  } }  public class Engine implements Starter {  public void start() {  System.out.println("Engine started");  } }  public class RemoteStarter implements Starter {  private Starter starter;    public RemoteStarter(Starter starter) {  this.starter = starter;  }    public void start() {  System.out.println("Starting car remotely");  starter.start();  } } |
| --- |

Now we can create a RemoteStarter object and pass it a Car or Engine object as its constructor argument, and the RemoteStarter can start the car remotely without modifying the existing code in the Car or Engine classes:

| Car myCar = new Car(new Engine()); Starter remoteStarter = new RemoteStarter(myCar); remoteStarter.start(); // outputs "Starting car remotely" and "Engine started" |
| --- |

This way, we have achieved the Open-Closed Principle by allowing the Car and Engine classes to be closed for modification and open for extension through the Starter interface and the RemoteStarter class.

## Liskov’s Substitution Principle:

Liskov Substitution Principle (LSP) is a fundamental principle in object-oriented programming that states that objects of a superclass should be replaceable with objects of its subclass without affecting the correctness of the program.

In other words, if we have a class A that is a superclass of class B, and class B is a subclass of A, then any instance of class A should be able to be replaced with an instance of class B without breaking the functionality of the program.

"S is a subtype of T, then objects of type T may be replaced with objects of type S❞

Derived types must be completely substitutable for their base types

Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called (strong) behavioral subtyping

### Implementation Guidelines:

* No new exceptions can be thrown by the subtype
* Clients should not know which specific subtype they are calling
* New derived classes just extend without replacing the functionality of old classes

### Example 1:

Here is an example of how breaking the LSP can lead to incorrect program behavior:

| class Rectangle {  constructor(width, height) {  this.width = width;  this.height = height;  }   setWidth(width) {  this.width = width;  }   setHeight(height) {  this.height = height;  }   area() {  return this.width \* this.height;  } }  class Square extends Rectangle {  constructor(size) {  super(size, size);  }   setWidth(width) {  this.width = width;  this.height = width;  }   setHeight(height) {  this.width = height;  this.height = height;  } }  function printArea(rectangle) {  rectangle.setWidth(4);  rectangle.setHeight(5);  console.log(rectangle.area()); }  const rectangle = new Rectangle(2, 3); const square = new Square(2);  printArea(rectangle); // Output: 20 printArea(square); // Output: 25, but we expected 16! |
| --- |

In this example, we have a Rectangle class that has a setWidth and setHeight method to set the width and height of the rectangle, and an area method to calculate the area of the rectangle. We also have a Square class that extends Rectangle and overrides the setWidth and setHeight methods to ensure that both the width and height are always the same.

However, when we pass a Square object to the printArea function, which is designed to work with a Rectangle object, we get the wrong output. This is because the setWidth and setHeight methods of Square violate the LSP by changing the behavior of the class compared to its superclass. In other words, a Square object is not a proper substitute for a Rectangle object in this case.

To fix this issue, we need to ensure that the Square class conforms to the LSP. One way to do this is to remove the setWidth and setHeight methods from the Square class and instead override the constructor to ensure that both the width and height are always the same:

| class Square extends Rectangle {  constructor(size) {  super(size, size);  } } |
| --- |

Now, when we pass a Square object to the printArea function, we get the expected output of 16:

| const rectangle = new Rectangle(2, 3); const square = new Square(2); printArea(rectangle); // Output: 20 printArea(square); // Output: 16 |
| --- |

This is because a Square object is now a proper substitute for a Rectangle object, and we can use it in the printArea function without breaking the correctness of the program.

## Interface Segregation Principle:

* "No client should be forced to depend on methods it does not use"
* One fat interface needs to be split into many smaller and relevant interfaces so that clients can know about the interfaces that are relevant to them

The Interface Segregation Principle (ISP) states that clients should not be forced to depend on interfaces they do not use. In simpler terms, it means that a client should only be exposed to the methods it needs and not to methods it doesn't need.

### Example 1:

Let's say we have an interface Printer that has the following methods:

| public interface Printer {  void print();  void scan();  void fax(); } |
| --- |

Let's say we have a class Document that needs to use the print() method only. So we implement the Printer interface in the Document class as follows:

| public class Document implements Printer {  @Override  public void print() {  System.out.println("Printing the document...");  }   @Override  public void scan() {  // Nothing to do here  }   @Override  public void fax() {  // Nothing to do here  } } |
| --- |

Here, the Document class is forced to implement the scan() and fax() methods even though it doesn't need them. This violates the ISP, as the Document class is forced to depend on methods it doesn't use.

To fix this, we can create smaller interfaces that are more focused on specific tasks. For example, we can create three interfaces: Printable, Scannable, and Faxable. Each interface will have only one method that corresponds to its name:

| public interface Printable {  void print(); }  public interface Scannable {  void scan(); }  public interface Faxable {  void fax(); } |
| --- |

Now, we can make the Printer interface extend these smaller interfaces:

| public interface Printer extends Printable, Scannable, Faxable { } |
| --- |

This way, clients can choose which interface to implement based on their needs. The Document class can now implement the Printable interface only:

| public class Document implements Printable {  @Override  public void print() {  System.out.println("Printing the document...");  } } |
| --- |

By doing this, we have followed the ISP, and the Document class is now only dependent on the methods it needs.

## Dependency Inversion Principle:

* High-level modules should not depend on low-level modules.
* Both should depend on abstractions
* Abstractions should not depend on implementation.
* Implementation should depend on abstractions

The interaction between high-level and low-level modules should be thought of as an abstract interaction between them. The goal of the DIP is to reduce coupling between components, which leads to more flexible and maintainable code. By using abstractions, we can create a layer of indirection between components, which makes it easier to change the implementation of a component without affecting other components.

A common violation of the DIP occurs when a high-level module depends directly on a low-level module, which can make it difficult to change the implementation of the low-level module without affecting the high-level module.

### Example 1:

| public class UserService {  private UserRepository userRepository;   public UserService() {  this.userRepository = new UserRepository();  }   public void createUser(User user) {  userRepository.save(user);  } } |
| --- |

In this example, the UserService class directly depends on the UserRepository class, violating the DIP. This tight coupling makes it difficult to change the implementation of the UserRepository class without affecting the UserService class.

To fix this violation of the DIP, we can introduce an interface for the repository and have both the UserRepository and UserService classes depend on this interface:

| public interface UserRepository {  void save(User user); }  public class InMemoryUserRepository implements UserRepository {  @Override  public void save(User user) {  // implementation details  } }  public class UserService {  private UserRepository userRepository;   public UserService(UserRepository userRepository) {  this.userRepository = userRepository;  }   public void createUser(User user) {  userRepository.save(user);  } } |
| --- |

In this updated example, the UserService class depends on the UserRepository interface instead of the concrete InMemoryUserRepository class. This makes it easy to add new repository implementations without modifying the UserService class. We can now create a new DatabaseUserRepository class that implements the UserRepository interface and pass it to the UserService class without changing any other code:

| public class DatabaseUserRepository implements UserRepository {  @Override  public void save(User user) {  // implementation details  } }  UserRepository databaseUserRepository = new DatabaseUserRepository(); UserService userService = new UserService(databaseUserRepository); |
| --- |

### Example 2:

| public class PaymentService {  public void processPayment(String paymentMethod, double amount) {  if (paymentMethod.equals("credit card")) {  CreditCardPaymentProcessor processor = new CreditCardPaymentProcessor();  processor.processPayment(amount);  } else if (paymentMethod.equals("paypal")) {  PayPalPaymentProcessor processor = new PayPalPaymentProcessor();  processor.processPayment(amount);  } else {  throw new IllegalArgumentException("Invalid payment method");  }  } } |
| --- |

In this example, the PaymentService class directly depends on the concrete implementations of the payment processors, violating the DIP. This makes it difficult to add new payment processors or change the existing ones without modifying the PaymentService class.

To fix this violation of the DIP, we can introduce an abstraction for the payment processors and have the PaymentService class depend on this abstraction:

| public interface PaymentProcessor {  void processPayment(double amount); }  public class CreditCardPaymentProcessor implements PaymentProcessor {  @Override  public void processPayment(double amount) {  // implementation details  } }  public class PayPalPaymentProcessor implements PaymentProcessor {  @Override  public void processPayment(double amount) {  // implementation details  } }  public class PaymentService {  private PaymentProcessor paymentProcessor;   public PaymentService(PaymentProcessor paymentProcessor) {  this.paymentProcessor = paymentProcessor;  }   public void processPayment(double amount) {  paymentProcessor.processPayment(amount);  } } |
| --- |

In this updated example, the PaymentService class depends on the PaymentProcessor abstraction instead of the concrete implementations of the payment processors. This makes it easy to add new payment processors without modifying the PaymentService class. We can now create a new BitcoinPaymentProcessor class that implements the PaymentProcessor interface and pass it to the PaymentService class without changing any other code:

| public class BitcoinPaymentProcessor implements PaymentProcessor {  @Override  public void processPayment(double amount) {  // implementation details  } }  PaymentProcessor bitcoinPaymentProcessor = new BitcoinPaymentProcessor(); PaymentService paymentService = new PaymentService(bitcoinPaymentProcessor); |
| --- |

## Dependency Injection:

Dependency Injection is a design pattern that helps to achieve the Dependency Inversion Principle by inverting the flow of control in the application. Instead of a class creating its dependencies, those dependencies are provided or injected into the class by an external entity.

| public class OrderService {  private OrderRepository orderRepository;   public OrderService(OrderRepository orderRepository) {  this.orderRepository = orderRepository;  }   public void createOrder(Order order) {  orderRepository.save(order);  } } |
| --- |

In this example, the OrderService class depends on the OrderRepository interface. Instead of creating an instance of the OrderRepository class directly, the OrderRepository instance is injected into the OrderService class through its constructor. The external entity that provides this dependency could be a Dependency Injection container, a factory method, or any other means of creating and managing objects.

This approach allows for easy substitution of different implementations of the OrderRepository interface without modifying the OrderService class. We can create different implementations of the OrderRepository interface, such as an in-memory repository or a database repository, and inject them into the OrderService class:

| public class InMemoryOrderRepository implements OrderRepository {  // implementation details }  public class DatabaseOrderRepository implements OrderRepository {  // implementation details }  OrderRepository inMemoryRepository = new InMemoryOrderRepository(); OrderRepository databaseRepository = new DatabaseOrderRepository();  OrderService inMemoryOrderService = new OrderService(inMemoryRepository); OrderService databaseOrderService = new OrderService(databaseRepository); |
| --- |

In this example, we can create two different instances of the OrderService class, one that uses the InMemoryOrderRepository and another that uses the DatabaseOrderRepository. This demonstrates the flexibility and maintainability that Dependency Injection provides, making it easier to manage complex systems with many dependencies.