



Code and Blog

SOLID principles

(A blog about the chosen principles and explain your work on the Knoldus blogs)

ISSUED BY

Knoldus Inc.

REPRESENTATIVE

Shubham Shrivastava

shubham.shrivastava@knoldus.com

+91-9450067694

Introduction About S.O.L.I.D. Principles

SOLID is a **mnemonic device for 5 design principles** of object-oriented programs (OOP) that result in readable, adaptable, and scalable code. SOLID can be applied to any OOP program.

There are 5 principles of the SOLID principle.

1. **S**ingle-responsibility principle
2. **O**pen-closed principle
3. **L**iskov substitution principle
4. **I**nterface segregation principle
5. **D**ependency inversion principle

The SOLID word comes from all 5 principles first word's first letter.

SOLID principles were Introduced and developed by computer science instructor and author Robert C. Martin in 2000 and quickly became a staple of modern object-oriented design (OOD). The SOLID acronym became commonplace when these principles gained widespread popularity in the programming world.



1. Single-responsibility principle

The **single-responsibility principle** (SRP) states that each class, module, or function in your program should only do one job. In other words, each should have full responsibility for a single functionality of the program. The class should contain only variables and methods relevant to its functionality.

Many classes can work together to complete the task but every class can execute the function from starting to end and after executing the function class passes the output to another function.

“A class should only have a single responsibility, that is, only changes to one part of the software’s specification should be able to affect the specification of the class.”



Implementation

```
code and blog – BillCalculation.java
Factor Build Run Tools Git Window SBTCommands Help
BillCalculation
BillCalculation.java Customer.java DeliveryApp.java GFG.java Order.java
1 package SRP;
2
3 import java.util.Random;
4
5 class BillCalculation {
6
7     private final Order order;
8     public BillCalculation(Order order)
9     {
10         this.order = order;
11     }
12
13     public void calculateBill()
14     {
15
16         Random rand = new Random();
17         int totalAmt
18             = rand.nextInt(1000) * this.order.getQuantity();
19
20         this.order.setTotalBillAmt(totalAmt);
21         System.out.println("Order with order id "
22             + this.order.getOrderid()
23             + " has a total bill amount of "
24             + this.order.getTotalBillAmt());
25     }
}
```

```
code and blog – Customer.java
Factor Build Run Tools Git Window SBTCommands Help
Customer
BillCalculation.java Customer.java DeliveryApp.java GFG.java
1 package SRP;
2
3 class Customer {
4     private String name;
5     private String address;
6     public String getName() { return name; }
7     public void setName(String name) { this.name = name; }
8     public String getAddress() { return address; }
9     public void setAddress(String address)
10    {
11        this.address = address;
12    }
13 }
14
15
```



ity Edition ▾ Sep 15 12:50

code and blog – DeliveryApp.java

Factor Build Run Tools Git Window SBT Commands Help

DeliveryApp > delivery

BillCalculation.java × Customer.java × DeliveryApp.java × GFG.java × Order.java ×

```

1 package SRP;
2
3 class DeliveryApp {
4
5     private Order order;
6     public DeliveryApp(Order order) { this.order = order; }
7
8     public void delivery()
9     {
10         // Here, we would want to interface with another
11         // system which actually assigns the task of
12         // delivery to different persons
13         // based on location, etc.
14         System.out.println("Delivering the order");
15         System.out.println(
16             "Order with order id as "
17             + this.order.getOrderid()
18             + " being delivered to "
19             + this.order.getCustomer().getName());
20         System.out.println(
21             "Order is to be delivered to: "
22             + this.order.getCustomer().getAddress());
23     }
24 }
25

```


BillCalculation.java × Customer.java × DeliveryApp.java × GFG.java × Order.java ×

```

1 package SRP;
2
3 import java.io.*;
4 import java.util.*;
5
6 class GFG {
7     public static void main(String[] args)
8     {
9         Customer customer1 = new Customer();
10        customer1.setName("John");
11        customer1.setAddress("Pune");
12        Order order1 = new Order();
13        order1.setItemName("Pizza");
14        order1.setQuantity(2);
15        order1.setCustomer(customer1);
16
17        order1.prepareOrder();
18
19        BillCalculation billCalculation
20            = new BillCalculation(order1);
21        billCalculation.calculateBill();
22
23        DeliveryApp deliveryApp = new DeliveryApp(order1);
24        deliveryApp.delivery();
25    }
26 }

```

Problems Python Packages SpotBugs CheckStyle




code and blog – Order.java

factor Build Run Tools Git Window SBT Commands Help

customer

BillCalculation.java × Customer.java × DeliveryApp.java × GFG.java × Order.java

```

1 package SRP;
2
3 import java.util.Random;
4
5 class Order {
6
7     private Customer customer;
8     private String orderId;
9     private String itemName;
10    private int quantity;
11    private int totalBillAmt;
12
13    public Customer getCustomer() { return customer; }
14    public void setCustomer(Customer customer) { this.customer = customer; }
15
16    public String getOrderId() { return orderId; }
17    public void setOrderId(String orderId)
18    {
19        Random random = new Random();
20
21        this.orderId = orderId + "-" + random.nextInt(10000);
22    }
23
24    public String getItemName() { return itemName; }
25    public void setItemName(String itemName)
26    {
27        this.itemName = itemName;
28        setOrderId(itemName);
29    }
30
31    public int getQuantity() { return quantity; }
32    public void setQuantity(int quantity) { this.quantity = quantity; }
33
34    public int getTotalBillAmt() { return totalBillAmt; }
35    public void setTotalBillAmt(int totalBillAmt) { this.totalBillAmt = totalBillAmt; }
36
37    public void prepareOrder()
38    {
39        System.out.println("Preparing order for customer -"
40            + this.getCustomer().getName()
41            + " who has ordered "
42            + this.getItemName());
43    }
44
45 }

```

Problems Python Packages SpotBugs CheckStyle

customer

BillCalculation.java × Customer.java × DeliveryApp.java × GFG.java × Order.java ×

```

20
21 {
22     Random random = new Random();
23
24     this.orderId = orderId + "-" + random.nextInt(10000);
25 }
26
27 public String getItemName() { return itemName; }
28 public void setItemName(String itemName)
29 {
30     this.itemName = itemName;
31     setOrderId(itemName);
32 }
33
34 public int getQuantity() { return quantity; }
35 public void setQuantity(int quantity) { this.quantity = quantity; }
36
37 public int getTotalBillAmt() { return totalBillAmt; }
38 public void setTotalBillAmt(int totalBillAmt) { this.totalBillAmt = totalBillAmt; }
39
40 public void prepareOrder()
41 {
42     System.out.println("Preparing order for customer -"
43         + this.getCustomer().getName()
44         + " who has ordered "
45         + this.getItemName());
46 }
47
48 }
49

```

Problems Python Packages SpotBugs CheckStyle



2. Open-closed principle

The idea of the open-closed principle is that existing, well-tested classes will need to be modified when something needs to be added. Yet, changing classes can lead to problems or bugs. Instead of changing the class, you simply want to extend it. With that goal in mind, Martin summarizes this principle, “You should be able to extend a class’s behaviour without modifying it.”

Following this principle is essential for writing code that is easy to maintain and revise. Your class complies with this principle if it is:

1. Open for extension, meaning that the class’s behaviour can be extended; and
2. Closed for modification, meaning that the source code is set and cannot be changed.

At first glance, these two criteria seem to be inherently contradictory, but when you become more comfortable with it, you’ll see that it’s not as complicated as it seems. The way to comply with these principles and to make sure that your class is easily extendable without having to modify the code is through the use of abstractions. Using inheritance or interfaces that allow polymorphic substitutions is a common way to comply with this principle. Regardless of the method used, it’s important to follow this principle in order to write code that is maintainable and revisable.

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



Implementation

code and blog – Context.java

factor Build Run Tools Git Window SBT Commands Help

Context

Context.java × open_close.java × Strategy.java × Strategy1.java × Strategy2.java ×

```

1 package open_close;
2
3 public class Context {
4     private Strategy strategy;
5     // we set the strategy in the constructor
6     public Context(Strategy strategy) { this.strategy = strategy; }
7
8     public void executeTheStrategy() { this.strategy.doSomething(); }
9 }

```

factor Build Run Tools Git Window SBT Commands Help

open_close

Context.java × open_close.java × Strategy.java × Strategy1.java × Strategy2.java ×

```

1 package open_close;
2
3 public class open_close {
4     public static void main(String[] args) {
5         Context context = new Context(new Strategy1()); // we inject the Strategy1
6         context.executeTheStrategy(); // it will print "Execute strategy 1";
7
8         context = new Context(new Strategy2()); // we inject the Strategy2
9         context.executeTheStrategy(); // it will print "Execute strategy 2"
10    }
11 }

```

Activities IntelliJ IDEA Community Edition

File Edit View Navigate Code Refactor Build Run Tools Git Window SBT Commands

code and blog > src > open_close > Strategy

Project

code and blog [code and blo

src

lsp

Father

Context.java × open_close.java × Strategy.java ×

```

1 package open_close;
2
3 public interface Strategy {
4     public void doSomething();
5 }
6

```



Community Edition ▾ Sep 15 11:11 AM

code and blog - Strategy

Factor Build Run Tools Git Window SBTCommands Help

Strategy1 > doSomething

Context.java × open_close.java × Strategy.java × Strategy1.java

```
1 package open_close;
2
3 public class Strategy1 implements Strategy {
4     public void doSomething() {
5         System.out.println("Execute strategy 1");
6     }
7 }
```

Community Edition ▾ Sep 15 11:11 AM

code and blog - Strategy

Factor Build Run Tools Git Window SBTCommands Help

Strategy2 > doSomething

Context.java × open_close.java × Strategy.java × Strategy1.java

```
1 package open_close;
2
3 public class Strategy2 implements Strategy {
4     public void doSomething() {
5         System.out.println("Execute strategy 2");
6     }
7 }
8
```



3. Liskov substitution principle

The **Liskov substitution principle** (LSP) is a specific definition of a subtyping relation created by Barbara Liskov and Jeannette Wing. The principle says that any class must be directly replaceable by any of its subclasses without error.

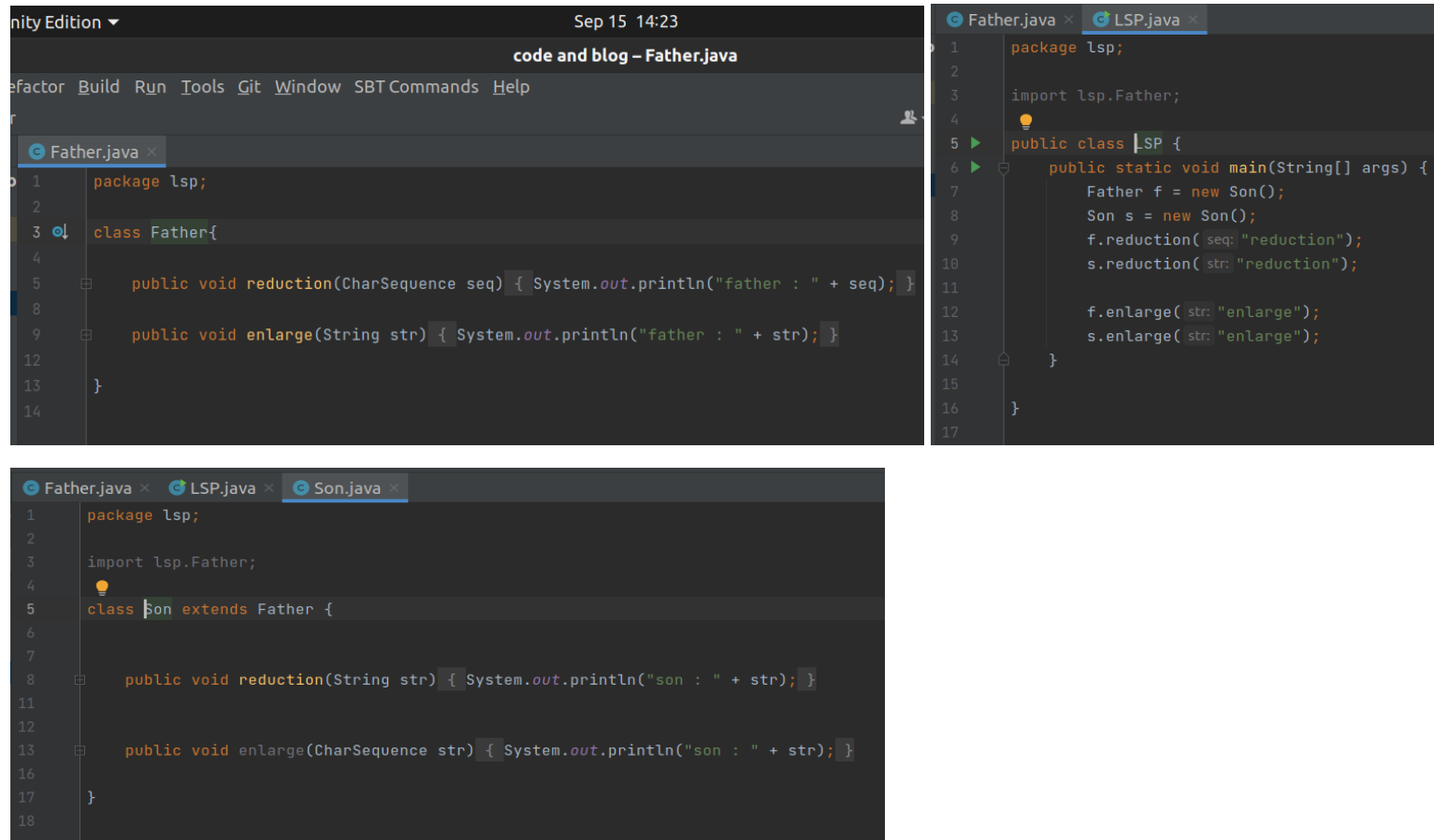
In other words, each subclass must maintain all behaviour from the base class along with any new behaviours unique to the subclass. The child class must be able to process all the same requests and complete all the same tasks as its parent class.

In practice, programmers tend to develop classes based on behaviour and grow behavioural capabilities as the class becomes more specific. The advantage of LSP is that it speeds up the development of new subclasses as all subclasses of the same type share a consistent use.

You can trust that all newly created subclasses will work with the existing code. If you decide that you need a new subclass, you can create it without reworking the existing code.



Implementation



The image displays three screenshots of an IDE showing Java code implementation. The top-left screenshot shows the `Father.java` file with the following code:

```
1 package lsp;
2
3 class Father {
4
5     public void reduction(CharSequence seq) { System.out.println("father : " + seq); }
6
7     public void enlarge(String str) { System.out.println("father : " + str); }
8
9 }
10
```

The top-right screenshot shows the `LSP.java` file with the following code:

```
1 package lsp;
2
3 import lsp.Father;
4
5 public class LSP {
6     public static void main(String[] args) {
7         Father f = new Son();
8         Son s = new Son();
9         f.reduction(seq: "reduction");
10        s.reduction(str: "reduction");
11
12        f.enlarge(str: "enlarge");
13        s.enlarge(str: "enlarge");
14    }
15 }
16
```

The bottom screenshot shows the `Son.java` file with the following code:

```
1 package lsp;
2
3 import lsp.Father;
4
5 class Son extends Father {
6
7     public void reduction(String str) { System.out.println("son : " + str); }
8
9     public void enlarge(CharSequence str) { System.out.println("son : " + str); }
10
11 }
12
```

