SOLID Principles Java

In Java, SOLID principles are an object-oriented approach that are applied to software structure design. It is conceptualized by Robert C. Martin (also known as Uncle Bob). These five principles have changed the world of object-oriented programming, and also changed the way of writing software. It also ensures that the software is modular, easy to understand, debug, and refactor. In this section, we will discuss SOLID principles in Java with proper example.

The word SOLID acronym for:

* Single Responsibility Principle (SRP)
* Open-Closed Principle (OCP)
* Liskov Substitution Principle (LSP)
* Interface Segregation Principle (ISP)
* Dependency Inversion Principle (DIP)

SRP:

Single Responsibility Principle

The single responsibility principle states that every Java class must perform a single functionality. Implementation of multiple functionalities in a single class mashup the code and if any modification is required may affect the whole class. It precise the code and the code can be easily maintained.

Suppose, Student is a class having three methods namely printDetails(), calculatePercentage(), and addStudent(). Hence, the Student class has three responsibilities to print the details of students, calculate percentages, and database. By using the single responsibility principle, we can separate these functionalities into three separate classes to fulfill the goal of the principle.

EXAMPLE :

public class Student

{

public void printDetails();

{

//functionality of the method

}

pubic void calculatePercentage();

{

//functionality of the method

}

public void addStudent();

{

//functionality of the method

}

}

......................................................................................................

STUDENT.JAVA:

public class Student

{

public void addStudent();

{

//functionality of the method

}

}

.......................................................................................................

PRINTSUDENTDETAILS:

public class PrintStudentDetails

{

public void printDetails();

{

//functionality of the method

}

}

........................................................................

PERSENTAGE CLASS:

public class Percentage

{

public void calculatePercentage();

{

//functionality of the method

}

}

.............................................................................

Open-Closed Principle

The application or module entities the methods, functions, variables, etc. The open-closed principle states that according to new requirements the module should be open for extension but closed for modification. The extension allows us to implement new functionality to the module

Suppose, VehicleInfo is a class and it has the method vehicleNumber() that returns the vehicle number.

EXAMPLE:

public class VehicleInfo

{

public double vehicleNumber(Vehicle vcl)

{

if (vcl instanceof Car)

{

return vcl.getNumber();

if (vcl instanceof Bike)

{

return vcl.getNumber();

}

}

.........................................................................

public class VehicleInfo

{

public double vehicleNumber()

{

//functionality

}

}

public class Car extends VehicleInfo

{

public double vehicleNumber()

{

return this.getValue();

}

public class Car extends Truck

{

public double vehicleNumber()

{

return this.getValue();

}

............................................................................................................................

Liskov Substitution Principle

The Liskov Substitution Principle (LSP) was introduced by Barbara Liskov. It applies to inheritance in such a way that the derived classes must be completely substitutable for their base classes. In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.

It extends the open-close principle and also focuses on the behavior of a superclass and its subtypes. We should design the classes to preserve the property unless we have a strong reason to do otherwise. Let's understand the principle through an example.

EXAMPLE:

public class Student

{

private double height;

private double weight;

public void setHeight(double h)

{

height = h;

}

public void setWeight(double w)

{

weight= w;

}

...

}

public class StudentBMI extends Student

{

public void setHeight(double h)

{

super.setHeight(h);

super.setWeight(w);

}

public void setWeight(double h)

{

super.setHeight(h);

super.setWeight(w);

}

}

.................................................................................................................

INTERFACE SEGREGATION PRINCPLE:



The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use.

The goal of the interface segregation principle is similar to the single responsibility principle. Let's understand the principle through an example.

Suppose, we have created an interface named Conversion having three methods intToDouble(), intToChar(), and charToString().

EXAMPLE:

public interface Conversion

{

public void intToDouble();

public void intToChar();

public void charToString();

}

...............................................

public interface ConvertIntToDouble

{

public void intToDouble();

}

public interface ConvertIntToChar

{

public void intToChar();

}

public interface ConvertCharToString

{

public void charToString();

}

..................................

Now we can use only the method that is required. Suppose, we want to convert the integer to double and character to string then, we will use only the methods intToDouble() and charToString()

EXAMPLE:

public class DataTypeConversion implements ConvertIntToDouble, ConvertCharToString

{

public void intToDouble()

{

//conversion logic

}

public void charToString()

{

//conversion logic

}

}

............................................................................................................

DEPENDENCY INVERSION PRINCPLE:



The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module but both should depend on the abstraction. Because the abstraction does not depend on detail but the detail depends on abstraction. It decouples the software. Let's understand the principle through an example.

public class WindowsMachine

{

//functionality

}

EXAMPLE:

public class WindowsMachine

{

public final keyboard;

public final monitor;

public WindowsMachine()

{

monitor = new monitor(); //instance of monitor class

keyboard = new keyboard(); //instance of keyboard class

}

}

..........................................................

Now we can work on the Windows machine with the help of a keyboard and mouse. But we still face the problem. Because we have tightly coupled the three classes together by using the new keyword. It is hard o test the class windows machine.

To make the code loosely coupled, we decouple the WindowsMachine from the keyboard by using the Keyboard interface and this keyword

public interface Keyboard

{

//functionality

}

.......................................

public class WindowsMachine

{

private final Keyboard keyboard;

private final Monitor monitor;

public WindowsMachine(Keyboard keyboard, Monitor monitor)

{

this.keyboard = keyboard;

this.monitor = monitor;

}

}

..............................................................................................

Why should we use SOLID principles?

* It reduces the dependencies so that a block of code can be changed without affecting the other code blocks.
* The principles intended to make design easier, understandable.
* By using the principles, the system is maintainable, testable, scalable, and reusable.
* It avoids the bad design of the software.

..............................................................................................................

Design Patterns in Java

A design patterns are well-proved solution for solving the specific problem/task.

Problem Given:

Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

Solution:

Singleton design pattern is the best solution of above specific problem. So, every design pattern has some specification or set of rules for solving the problems. What are those specifications, you will see later in the types of design patterns.

But remember one-thing, design patterns are programming language independent strategies for solving the common object-oriented design problems. That means, a design pattern represents an idea, not a particular implementation.

By using the design patterns you can make your code more flexible, reusable and maintainable. It is the most important part because java internally follows design patterns.

To become a professional software developer, you must know at least some popular solutions (i.e. design patterns) to the coding problems.

..............................................................................

When should we use the design patterns?

We must use the design patterns during the analysis and requirement phase of SDLC(Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

Categorization of design patterns:

Basically, design patterns are categorized into two parts:

* Core Java (or JSE) Design Patterns.
* JEE Design Patterns.

.......................................................................................................

* Core Java Design Patterns

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

1.Creational Design Pattern

* Factory Pattern
* Abstract Factory Pattern
* Singleton Pattern
* Prototype Pattern
* Builder Pattern.

2. Structural Design Pattern

* Adapter Pattern
* Bridge Pattern
* Composite Pattern
* Decorator Pattern
* Facade Pattern
* Flyweight Pattern
* Proxy Pattern

3. Behavioral Design Pattern

* Chain Of Responsibility Pattern
* Command Pattern
* Interpreter Pattern
* Iterator Pattern
* Mediator Pattern
* Memento Pattern
* Observer Pattern
* State Pattern
* Strategy Pattern
* Template Pattern
* Visitor Pattern

...............................................................................................

Creational design patterns

Creational design patterns are concerned with the way of creating objects. These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

by using new keyword:

StudentRecord s1=new StudentRecord();

Factory Method Pattern:

A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as Virtual Constructor.

Advantage of Factory Design Pattern

Factory Method Pattern allows the sub-classes to choose the type of objects to create.

It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.



We are going to create a Plan abstract class and concrete classes that extends the Plan abstract class. A factory class GetPlanFactory is defined as a next step.

GenerateBill class will use GetPlanFactory to get a Plan object. It will pass information (DOMESTICPLAN / COMMERCIALPLAN / INSTITUTIONALPLAN) to GetPalnFactory to get the type of object it needs.

...............................................................................................

2.ABSTRACT FACTORY PATTREN:

Abstract Factory Pattern

Abstract Factory Pattern says that just define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

An Abstract Factory Pattern is also known as Kit.

EXAMPLE :



Step 1: Create a Bank interface

Step 2: Create concrete classes that implement the Bank interface.

Step 3: Create the Loan abstract class.

Step 4: Create concrete classes that extend the Loan abstract class..

Step 5: Create an abstract class (i.e AbstractFactory) to get the factories for Bank and Loan Objects

Step 6: Create the factory classes that inherit AbstractFactory class to generate the object of concrete class based on given information.

Step 7: Create a FactoryCreator class to get the factories by passing an information such as Bank or Loan.

Step 8: Use the FactoryCreator to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.