Object-Oriented Analysis and Design using JAVA

B.Tech (CSE/IT) 5th SEM
2020-2021

Lecture-32 SOLID Principles

Introduction

- 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.

- In this tutorial, We will learn the *SOLID principles* which are 5 most recommended design principles, that we should keep in mind while writing our classes.

Reference: https://howtodoinjava.com/best-practices/solid-principles/

Introduction to SOLID principles

SOLID is the acronym for a set of practices that, when implemented together, makes the code more adaptive to change.

Bob Martin and Micah Martin introduced these concepts in their book *'Agile Principles, Patterns, and Practices'*.

The acronym was meant to help us remember these principles easily. These principles also form a vocabulary we can use while discussing with other team members or as a part of technical documentation shared in the community.

SOLID principles form the fundamental guidelines for building object-oriented applications that are robust, extensible, and maintainable.

Reference: https://howtodoinjava.com/best-practices/solid-principles/

S.O.L.I.D. Class Design Principles

Principle Name	What it says? howtodoinjava.com
Single Responsibility Principle	One class should have one and only one reasonability
Open Closed Principle	Software components should be open for extension, but closed for modification
Liskov's Substitution Principle	Derived types must be completely substitutable for their base types
Interface Segregation Principle	Clients should not be forced to implement unnecessary methods which they will not use
Dependency Inversion Principle	Depend on abstractions, not on concretions

Reference: https://howtodoinjava.com/best-practices/solid-principles/

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.

Reference: https://howtodoinjava.com/best-practices/solid-principles/

Open Closed Principle

- 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.

Reference: https://howtodoinjava.com/best-practices/solid-principles/

Example

```
public class Animal {
   public void makeNoise() {
      System.out.println("I am making noise");
   }
}
```

Now let’s consider the Cat and Dog classes which extends Animal.

```
public class Dog extends Animal {
   @Override
   public void makeNoise() {
      System.out.println("bow wow");
   }
}
```

```
public class Cat extends Animal {
   @Override
   public void makeNoise() {
      System.out.println("meow meow");
   }
}
```

```
class DumbDog extends Animal {
   @Override
   public void makeNoise() {
      throw new RuntimeException("I can't make
noise");
   }
}
```

Liskov’s Substitution Principle

- LSP is a variation of previously discussed open closed principle. It says:
- “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 through 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.

Reference: https://howtodoinjava.com/best-practices/solid-principles/

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.
- 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.

Reference: https: https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/

Dependency Inversion Principle:

- Before we discuss this topic keep in mind that Dependency Inversion and 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.
- 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.

Reference: https: https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/

Key references

https://howtodoinjava.com/best-practices/solid-principles/

https: https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/

Thank You