# 5 Class Design Principles [S.O.L.I.D.] in Java

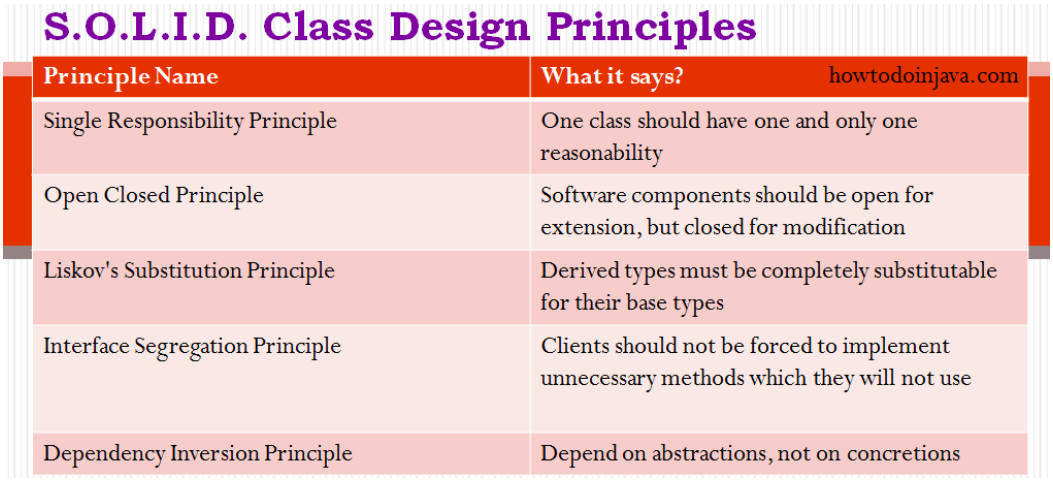
June 7, 2013 by Lokesh Gupta

Classes are the building blocks of your java application. If these blocks are not strong, your building (i.e. application) is going to face the tough time in future. This essentially means that not so well-written can lead to very difficult situations when the application scope goes up or application faces certain design issues either in production or maintenance.

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

In this post, I will list down 5 most recommended design principles, you should keep in mind, while writing your classes. These design principles are called SOLID, in short. They also form the best practices to be followed for designing your application classes.

Table Of Contents



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 |

5 java class design principles

Lets drill down all of them one by one.

## Single Responsibility Principle

The name of the principle says it all:

> "One class should have one and only one responsibility"

In other words, you should write, change and maintain a class for only one purpose. If it is model class then it should strictly represent only one actor/ entity. This will give you the flexibility to make changes in future without worrying the impacts of changes for another entity.

Similarly, If you are writing service/manager class then it should contain only that part of method calls and nothing else. Not even utility global functions related to module. Better separate them in another globally accessible class file. This will help in maintaining the class for that particular purpose, and you can decide the visibility of class to specific module only.

## Open Closed Principle

This is second important rule which you should keep in mind while designing your application. It says:

> "Software components should be open for extension, but closed for modification"

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

If other developers are not able to design desired behavior due to constraints put by your class, then you should reconsider changing your class. I do not mean here that anybody can change the whole logic of your class, but he/she should be able to override the options provided by software in unharmful way permitted by software.

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

## Liskov's Substitution Principle

This principle is a variation of previously discussed open closed principle. It says:

> "Derived types must be completely substitutable for their base types"

It means that the classes fellow developer created by extending your class should be able to fit in application without failure. I.e. if a fellow developer poorly extended some part of your class and injected into framework/ application then it should not break the application or should not throw fatal exceptions.

This can be insured by using strictly following first rule. If your base class is doing one thing strictly, the fellow developer will override only one feature incorrectly in worst case. This can cause some errors in one area, but whole application will not do down.

## Interface Segregation Principle

This principle is my favorite one. It is applicable to interfaces as single responsibility principle holds to classes. It 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 intend to use reports only in PDF format and not in excel. Will he achieve the functionality easily.

NO. He will have to implement two methods, out of which one is extra burden put on him by designer of software. Either he will implement another method or leave it blank. So are not desired cases, right??

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 user to use only required functionality only.

## Dependency Inversion Principle

Most of us are already familiar with the words used in principle's name. It says:

> "Depend on abstractions, not on concretions"

In other words. you should design your software in such a way that various modules can be separated from each other using an abstract layer to bind them together. The classical use of this principle of BeanFactory in spring framework. In spring framework, all modules are provided as separate components which can work together by simply injected dependencies in other module. They are so well closed in their boundaries that you can use them in other software modules apart from spring with 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 plugin in another module.

These were five class design principles which makes the best practices to be followed to design your application classes. Let me know of your thoughts.

Happy Learning !!


## Readers also found them useful:

Top 8 Signs of Bad Unit Testcases
Create Eclipse Templates for Faster Java Coding
Java 8 List All Files In Directory Example
Java web-application performance improvement tips
How you should unit test DAO layer
Google Gson Tutorial : Convert Java Object to / from JSON