

[Crypto](#)[Coding](#)[Future](#)[Startups](#)[About](#)[Community](#)[Sponsor](#)[Recommends](#)[HN](#)
[Shi](#)[.tech](#) Your .TECH domain is available. Check now!

10 OOP Design Principles Every Programmer Should Know

[in](#)
[l](#)

May 5th 2019

[TWEET THIS](#)

The Object-Oriented Design Principles are the core of OOP programming, but I have seen most of the Java programmers chasing design patterns like Singleton pattern, Decorator pattern, or Observer pattern, and not putting enough attention on learning *Object-oriented analysis and design*.

[://javarevisited.blogspot.com](https://javarevisited.blogspot.com)
[://java67.com](https://java67.com)

It's important to learn the basics of Object-oriented programming like Abstraction, Encapsulation, Polymorphism, and Inheritance. But, at the same time, it's equally important to know object-oriented design principles.

They will help you to create a clean and modular design, which would be easy to test, debug, and maintain in the future.

I have regularly seen Java programmers and developers of various experience level, who have either never heard about these OOP and SOLID design principle, or simply doesn't know what benefits a particular design principle offers and how to apply these design principle in coding.

To do my part, I have jotted down all important object-oriented design principles and putting it here for quick reference. These will at least give you some idea about what they are and what benefit they offer.

I have not put examples, just to keep the article short but you can find a lot of examples of these design principles on the internet and even on my **Java blog**, just use the search bar at the top of the page.

If you are not able to understand a design principle, you should try to do more than one example because sometimes we connect to another example or author better but you must understand these design principles and learn how to use it in your code.

Another thing you can do is to join a comprehensive object-oriented design course like **SOLID Principles of Object-Oriented Design** by Steve Smith on Pluralsight. It has helped me a lot in my understanding and application of these principles.

Btw, I have shared some relevant and useful courses and books here and there, both free and paid, and I will earn some money if you buy something which is not free.

They are also some of the resources I have used to learn SOLID design principles and Programming in general and nice for learning some of these principles in depth.

10 Object Oriented and SOLID Desing Principles for Programmers

Though the best way of learning any design principle or pattern is a real-world example and understanding the consequences of violating that design


principle, subject of this article is Introducing *Object-oriented design principles* for Java Programmers, who are either not exposed to it or in the learning phase.

I personally think each of these OOP and SOLID design principles needs an article to explain them clearly, and I will definitely try to do that here, but for now, just get yourself ready for a quick bike ride on design principle town :)

1. DRY (Don't repeat yourself)

Our first object-oriented design principle is DRY, as the name suggests **DRY (don't repeat yourself)** means don't write duplicate code, instead use Abstraction to abstract common things in one place.

If you have a block of code in more than two places consider making it a separate method, or if you use a hard-coded value more than one time make them public final constant. The benefit of this Object oriented design principle is in **maintenance**.



It's important not to abuse it, duplication is not for code, but for functionality.

It means if you have used common code to validate OrderId and SSN it doesn't mean they are the same or they will remain the same in future.

By using common code for two different functionality or thing you closely couple them forever and when your OrderId changes its format, your SSN validation code will break.

So beware of such coupling and just don't combine anything which uses the similar code but are not related. You can further check out the **Basics of Software Architecture & Design Patterns** in Java course on Udemy to learn more about writing good code and best practices to follow while designing a system.

2. Encapsulate What Changes

There is only one thing which is constant in the software field and that is "Change", So, encapsulate the code you expect or suspect to be changed in future.

The benefit of this OOP Design principle is that It's easy to test and maintain proper encapsulated code.

If you are coding in Java then follow the principle of making variable and methods private by default and increasing access step by step like from a private to protected and not public.

Several of the **design patterns in Java** uses Encapsulation, the Factory design pattern is one example of Encapsulation which encapsulates object creation code and provides flexibility to introduce a new product later with no impact on existing code.

Btw, if you are interested in learning more about design patterns in Java and Object Oriented Programming then you must check this **Design Pattern Library** course Pluralsight. It's one of the best collection of design patterns and advice on how to use them in the real world.

3. Open Closed Design Principle

According to this OOP design principle, “Classes, methods or functions should be Open for extension (new functionality) and Closed for modification”.

This is another beautiful SOLID design principle, coined by Uncle Bob on his classic **Clean Code** book, which prevents someone from changing already tried and tested code.

The key benefit of this design principle is that already tried and tested code is not touched which means they won't break.

Here is a Java code example which *violates* the Open Closed Design Principle of Programming:

In this code `GraphicEditor` is tightly coupled with `Shape`, If you need a new `Shape` then you need to modify already tried and tested code inside `drawShape(Shape s)` method, which is both error-prone and not desirable.

Ideally, if you are adding new functionality only then your code should be tested and that's the goal of Open Closed Design principle.

By the way, the Open-Closed principle is “O” from the SOLID acronym. If you want to learn more about this principle, the **SOLID Principles of Object-Oriented Design and Architecture** course on Udemy is one of the best resources to consult.

4. Single Responsibility Principle (SRP)

Single Responsibility Principle is another SOLID design principle, and represent “S” on the SOLID acronym. As per SRP, there should not be more than one reason for a class to change, or a class should always handle single functionality.

The key benefit of this principle is that it reduces coupling between the individual component of the software and Code.

For example, If you put more than one functionality in one Class in Java it introduces **coupling** between two functionality and even if you change one functionality there is a chance you broke coupled functionality, which requires another round of testing to avoid any surprise on the production environment.

You can further see **From 0 to 1: Design Patterns—24 That Matter** course on Udemy to learn about patterns which are based on this principle.

5. Dependency Injection or Inversion principle

Don't ask for dependency it will be provided to you by the framework. This has been very well implemented in Spring framework, one of the most popular Java framework for writing real-worth applications.

The beauty of this **design principle** is that any class which is injected by DI framework is **easy to test** with the mock object and easier to maintain because object creation code is centralized in the framework and client code is not littered with that.

There are multiple ways to implemented **Dependency injection** like using bytecode instrumentation which some AOP (Aspect Oriented programming) framework like AspectJ does or by using proxies just like used in Spring.

You can further see the **SOLID Principles of Object-Oriented Design and Architecture** course on Udemy to learn more about this useful principle. It also represents “D” on the SOLID acronym.

Here is an example of the code which violates Dependency Inversion Principle or DIP in Java:

You can see that AppManager depends upon EventLogWriter which is tightly coupled with the AppManager. If you need to use another way to notify your client like by sending push notifications, SMS, or E-mail, you need to change the AppManager class.

This problem can be solved by using the Dependency Inversion Principle where instead of AppManager asking for EventLogWriter, it will be injected or provided to AppManager by the framework.

You can further see Using SOLID Principles to Write Better Code—A Crash Course on Udemy to learn more about the Dependency Inversion Principle and how to solve such problems.

6. Favor Composition over Inheritance

There are two general ways to reuse the code you have already written, Inheritance and Composition, both have their own advantage and disadvantages, but, in general, you should always favor composition over

inheritance, if possible.

Some of you may argue this, but I found that Composition is the lot more flexible than Inheritance.

Composition allows changing the behavior of a class at run-time by setting property during run-time and by using Interfaces to compose a class we use polymorphism which provides flexibility to replace with better implementation any time.

Even Joshua Bloch's **Effective Java** advise favoring composition over inheritance. If you are still not convinced then you can also read [here](#) to learn more about why your Composition is better than Inheritance for reusing code and functionality.

And, if you keep forgetting this rule, here is a nice cartoon to put in your desk :-)

If you are interested in learning more about Object Oriented Programming Concepts like Composition, Inheritance, Association, Aggregation, etc, you can also take a look at the Object-Oriented Programming in Java course on Coursera.

It's free to explore and learn but you will be charged if you also want to participate in exercises, assignment, evaluation and need Certification to show in your LinkedIn profile.

7. Liskov Substitution Principle (LSP)

According to the Liskov Substitution Principle, Subtypes must be substitutable for supertype I mean methods or functions which uses superclass type must be able to work with the object of subclass without any issue”.

LSP is closely related **to the Single responsibility principle** and **Interface Segregation Principle**.

If a class has more functionality than subclass might not support some of the functionality and does violate LSP.

In order to follow LSP SOLID design principle, derived class or subclass must enhance functionality, but not reduce them. LSP represents “L” on the SOLID acronym.

Here is a code example which violates the Liskov Substitution Principle in Java:

Liskov Substitution Principle in Java

If you have a method `area(Rectangle r)` which calculates the area of Rectangle then that code will break when you pass the Square because Square is not really a Rectangle.

If you are interested in a more real-world example, then the **SOLID Principles of Object-Oriented Design** course on Pluarlsight is a good course to start with.

Btw, you would need a Pluralsight membership to get access this course, which cost around \$29 per month or \$299 annually (14% discount). Even if you don't have a membership, you can still access this course for FREE by

taking advantage of their **10-day free trial** without any commitment, which is a great way to not just access this course for free but also to check the quality of courses before joining Pluralsight

8. Interface Segregation Principle (ISP)

Interface Segregation Principle stats that, a client should not implement an interface if it doesn't use that.

This happens mostly when one interface contains more than one functionality, and the client only needs one functionality and no other.

There is no doubt that Interface design is a tricky job because once you release your interface you can not change it without breaking all implementation. Well, Java 8's default or defender method feature does provide a way for interface evolution but not all Programming language support that features.

Another benefit of this design principle in Java is, the interface has the disadvantage of implementing all method before any class can use it so having single functionality means less method to implement.

If you don't the get the benefit of the interface in coding then I suggest you read my blog post, the real usage of an interface in Java to learn more.

9. Programming for Interface not implementation

A programmer should always *program for the interface and not for implementation* this will lead to flexible code which can work with any new

implementation of the interface.

In concrete words, you should use interface type on variables, return types of a method or argument type of methods in Java like using SuperClass type to store object rather using SubClass.

I mean

```
List numbers= getNumbers();
```

instead of

```
ArrayList numbers = getNumbers();
```

This has also been advised in many Java books including in **Effective Java** and **Head First design pattern** book.

Here is an example of Coding for the interface in Java:

If you are interested in improving code quality of your program, I also suggest you take a look at the **Refactoring to Design Patterns** course on Udemy which will help you to improve the internal design with refactoring techniques and design patterns in C#

10. Delegation principles

Don't do all stuff by yourself, delegate it to the respective class. Classical example of delegation design principle is equals() and hashCode() method in Java.

In order to compare two objects for equality, we ask the class itself to do comparison instead of Client class doing that check.

The key benefit of this design principle is **no duplication of code** and pretty easy to modify behavior. Event delegation is another example of this principle, where an event is delegated to handlers for handling.

Summary

All these **object-oriented design principles** help you write flexible and better code by striving high cohesion and low coupling.

The theory is the first step, but what is most important is to *develop the ability to find out when to apply these design principles*.

Once you get hold of that, the next step is to learn Design patterns in Java, which uses these design patterns to solve common problems of application development and software engineering.

If you are looking for a nice course to start with, I suggest you join the **From 0 to 1: Design Patterns—24 That Matter—In Java** course on Udemy. It's very comprehensive and you can get it in just \$11 on their several flash sales.

Anyway, here is a nice summary of all these OOP design principles.

Find out, whether we are violating any design principle and compromising the flexibility of code, but again as nothing is perfect in this world, don't always try to solve the problem with **design patterns and design principle**

they are mostly for large enterprise project which has longer maintenance cycle.

Bottom line is, professionals programmers should always strive for a highly cohesive and loosely couple solution, code or design. Looking open source code from Apache and Google are some good ways of learning Java and OOP design principles.

They will show you, how design principles should be used in coding and Java programs. Java Development Kit follows many design principles like Factory Pattern in BorderFactory class, Singleton pattern in java.lang.Runtime class, Decorator pattern on various java.io classes.

If you are interested in learning object-oriented principles and patterns, then you can look at my another personal favorite **Head First Object-Oriented Analysis and Design**, an excellent book and probably the best material available in object-oriented analysis and design

Not many programmers know this book because it is often shadowed by its more popular cousin Head First Design Pattern by Eric Freeman, which is more about how these principles come together to create a pattern you can use directly to solve known problems.

These books help a lot to write better code, taking full advantage of various Object-oriented and SOLID design principles. Btw, if you really interested more in Java coding practices then read **Effective Java 3rd Edition** by Joshua Bloch, a gem by the guy who wrote Java Collection API.

If you want to learn more about SOLID design Principles, here are some useful resources you can take a look:

1. Clean Code By Robert Martin
2. SOLID Principles of Object-Oriented Design
3. SOLID Principles of Object-Oriented Design and Architecture
4. Refactoring by Martin Fowler

And, if you like this article, you may like these **Java and Programming articles as well:**

10 Things Java Programmer should learn in 2019

10 Books Every Programmer Must Read

10 Tips to Improve Your Programming skill

10 Tools Every Software Developer should know

5 Courses to Learn Software Architecture in Depth

20 Libraries and APIS Java Programmer Should Know

Top 10 Programming languages to Learn in 2019

10 Framework and Library Java and Web Developer Should Learn

Thanks for reading this article. If you find these object-oriented design principles useful then please share with your friends and colleagues. If you have any questions or feedback then please drop a note.

Btw, if you buy any of my recommended book or course, I'll be also be paid.

If you like this article then please consider following me on medium (javinpaul). if you'd like to be notified for every new post and don't forget to follow **javarevisited** on Twitter!

P. S.—If you are really passionate about your Coding and want to improve your coding skill there are no better books then **Clean Code** by Robert Martin and Refactoring by Martin P. Fowler. Just go and read them.

Programming

Javascript

Java

Coding

Software Development

Continue the discussion 

More by Javin Paul

10 Awesome Gift Ideas for Programmers and Geeks

javinpaul
Dec 24

Programming

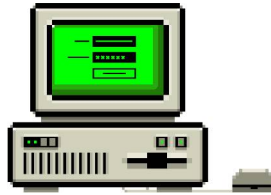
10 Basic Tips on Working Fast in UNIX or Linux Terminal

javinpaul
Oct 16

Linux

10 Best Programming Languages to Learn in 2019

javinpaul



Hackernoon Newsletter curates great stories by real tech professionals

Get solid gold sent to your inbox. Every week!

TOPICS OF INTEREST

☒ Software Development

☒ Blockchain Crypto

☒ General Tech

☒ Best of Hacker Noon

Get great stories by email

More Related Stories

0-100 in Django: Starting an app the right way

Jeremy Spencer
Sep 04

Python

The 7 Pro Tips To Get Productive With Angular CLI & Schematics

Tomas Trajan
Jan 15

Angular

Java bits: 0xFF and 0xFFL

ericniebler



Help

About

**Start
Writing**

Sponsor: *Brand-
as-
Author*

*Sitewide
Billboard*



[Contact Us](#)

[Privacy](#)

[Terms](#)