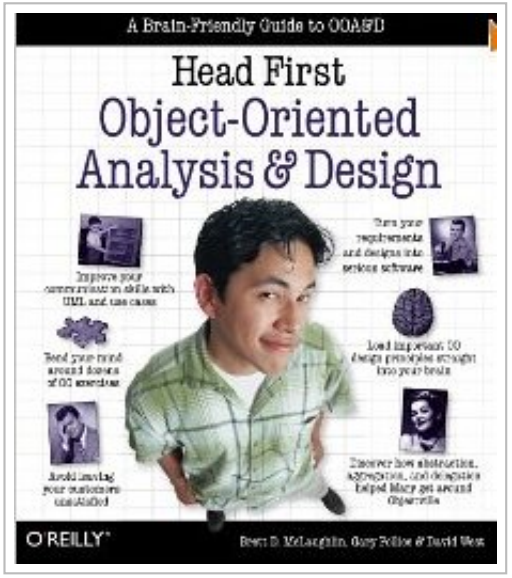
10 Object Oriented Design Principles Java Programmer should know

Object Oriented Design Principles are core of OOP programming, but I have seen most of the Java programmers chasing design patterns likeSingleton pattern, Decorator pattern or Observer pattern, and not putting enough attention on learning *Object oriented analysis and design*. It's important to learn 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, to create clean and modular design. I have regularly seen Java programmers and developers of various experience level, who either doesn't heard about these **OOP** and **SOLID design principle,**or simply doesn't know what **benefits a particular design principle** offers, or how to applythese design principle in coding.

Bottom line is, always strive for highly cohesive and loosely couple solution, code or design. Looking open source code from Apache and Sun are good examples of learning Java and OOPS design principles. They show us, how design principles should be used in coding and Java programs. Java Development Kit follows several design principle like Factory Pattern in `BorderFactory` class, Singleton pattern in `Runtime` class, Decorator pattern on various `java.io` classes. By the way if you really interested more on Java coding practices then read Effective Java by Joshua Bloch , a gem by the guy who wrote Java Collection API.

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. This an excellent book and probably the best material available in object oriented analysis and design, but it often shadowed by its more popular cousin Head First Design Pattern by Kathy Sierra. Later is more about how these principle comes together to create pattern you can use directly to solve known problems. These books helps a lot to write better code, taking full advantage of various Object oriented and SOLID design principles.



Though best way of learning any design principle or pattern is 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 learning phase. I personally think each of these OOPS and SOLID design principle need an article to explain them clearly, and I will definitely try to do that here, but for now just get yourself ready for quick bike ride on design principle town :)

DRY (Don't repeat yourself)

Our first object oriented design principle is DRY, as name suggest **DRY (don't repeat yourself)** means don't write duplicate code, instead use Abstraction to abstract common things in one place. If you have block of code in more than two place consider making it a separate method, or if you use a hard-coded value more than one time make them public final constant. 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 used common code to validate `OrderID` and `SSN` it doesn't mean they are same or they will remain 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 similar code but are not related.

Encapsulate What Changes

Only one thing is constant in software field and that is "Change", So encapsulate the code you expect or suspect to be changed in future. Benefit of this OOPS Design principle is that Its easy to test and maintain proper encapsulated code. If you are coding in Java then follow principle of making variable and methods private by default and increasing accessstep by step e.g. from private to protected and not public. Several of **design pattern in Java** uses Encapsulation, Factory design pattern is one example of Encapsulation which encapsulate object creation code and provides flexibility to introduce new product later with no impact on existing code.

Open Closed Design Principle

Classes, methods or functions should be Open for extension (new functionality) and Closed for modification. This is another beautiful SOLID design principle, which prevents some-one from changing already tried and tested code. Ideally if you are adding new functionality only than your code should be tested and that's the goal of Open Closed Design principle. By the way, Open Closed principle is "O" from SOLID acronym.

Single Responsibility Principle (SRP)

Single Responsibility Principle is another SOLID design principle, and represent "S" on 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. If you put more than one functionality in one Class in Java it introduce **coupling** between two functionality and even if you change one functionality there is chance you broke coupled functionality, which require another round of testing to avoid any surprise on production environment.

Dependency Injection or Inversion principle

Don't ask for dependency it will be provided to you by framework. This has been very well implemented in Spring framework, beauty of this **design principle** is that any class which is injected by DI framework is easy to test with mock object and easier to maintain because object creation code is centralized in framework and client code is not littered with that.There are multiple ways to implemented **Dependency injection** like using byte code instrumentation which some AOP (Aspect Oriented programming) framework like AspectJ does or by using proxies just like used in Spring. See this example of IOC and DI design pattern to learn more about this SOLID design principle. It represent "D" on SOLID acronym.

Favor Composition over Inheritance

Always *favor composition over inheritance* ,if possible. Some of you may argue this, but I found that Composition is lot more flexible than Inheritance. Composition allows to change 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 of to replace with better implementation any time. Even Effective Java advise to favor composition over inheritance.
See here to learn more about why you Composition is better than Inheritance for reusing code and functionality.

Liskov Substitution Principle (LSP)

According to `Liskov Substitution Principle`, Subtypes must be substitutable for super type i.e. methods or functions which uses super class type must be able to work withobject of sub class without any issue". LSP is closely related **to Single responsibility principle** and **Interface Segregation Principle**. If a class has more functionality than subclass might not support some of the functionality ,and does violated LSP. In order to follow **LSP SOLID design principle**, derived class or sub class must enhance functionality, but not reduce them. LSP represent "L" on SOLID acronym.

Interface Segregation principle (ISP)

`Interface Segregation Principle` stats that, a client should not implement aninterface, if it doesn't use that. This happens mostly when one interface contains more than one functionality, and client only need one functionality and not other.Interface design is tricky job because once you release your interface you can not change it without breaking all implementation. Another benefit of this design principle in Java is, interface has disadvantage to implement all method before any class can use it so having single functionality means less method to implement.

Programming for Interface not implementation

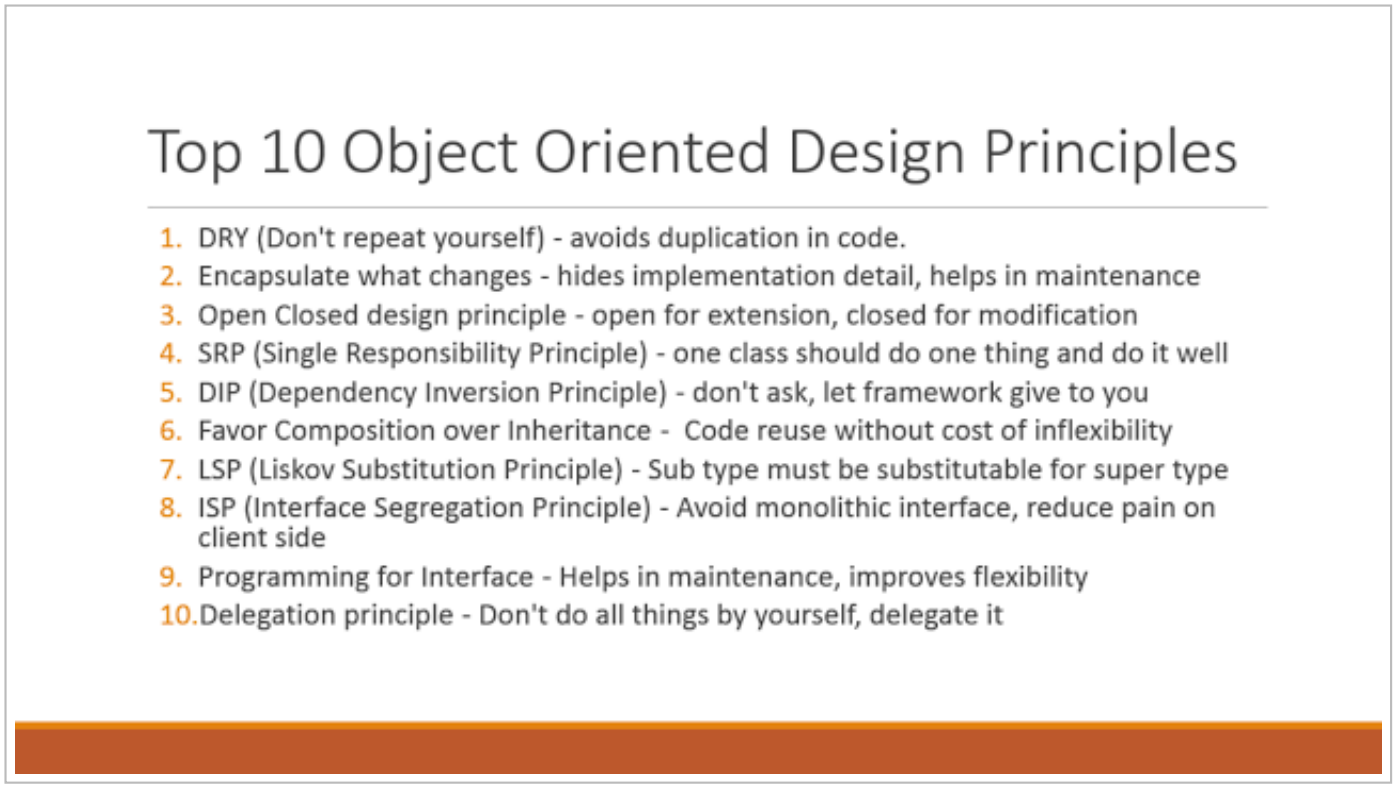
Always *program for interface and not for implementation* this will lead to flexible code which can work with any new implementation of interface. So use interface type on variables, return types of method or argument type of methods in Java. This has been advised by many Java programmer including in Effective Java and Head First design pattern

book.

Delegation principle

Don't do all stuff by yourself, delegate it to respective class. Classical example of delegation design principle is equals() and hashCode() method in Java. In order to compare two object for equality we ask class itself to do comparison instead of Client class doing that check. Benefit of this design principle is no duplication of code and pretty easy to modify behavior.

Here is nice summary of all these OOP design principles :



All these **object oriented design principle** helps you write flexible and better code by striving high cohesion and low coupling. Theory is first step, but what is most important is to *develop ability to find out when to apply these design principle*. Find out, whether we are violating any design principle and compromising flexibility of code, but again as nothing is perfect in this world, don't always try to solve problem with **design patterns and design principle** they are mostly for large enterprise project which has longer maintenance cycle.

Recommended books to learn Object Oriented analysis, design and patterns :

- Head First Design Pattern by Kathy Sierra [see here]
- Head First Object Oriented Analysis and Design by O'Rielly [see here]