**Why do we need a good design?**

To avoid Rigid (high level is dependent to low level) and fragile design (one module impacts other module).

**How to evaluate good design?**

Only constant thing in software is change. Good design is when the cost to change a software is minimum.

**The motivation:**

It’s almost impossible to get it right at first place. We need to be courageous enough to delete our own code and iterate multiple times to get the perfect design. It also required to leave the ego and doubt our own design. Every artist prototype step by step to get the best design.

**How to create a good design?**

Following are some of the many ways to create a good design:

* Have design discussion.
* Take time to code review.
* Good design needs more time.

**Three steps to create great software:**

Make your software what the customer wants it to do -> Apply basic oo principles to add flexibility -> Strive for a maintainable, reusable design.

<https://www.javacodegeeks.com/2012/08/10-object-oriented-design-principles.html>

**Keep it simple! (Kiss – keep it simple and stupid)**

* Simple shows only real problem
* Simple fails less
* Is easier to understand.

Lot of times when we talk to developer they say I’m very busy, but when you ask what you are doing. They will say I’ll tell you once I get to know.

We should not start coding until we don’t know the real problem.

Below are some of the principles helps in achieving this:

**YAGNI – You’re not going to need it.**

How much you know?

Cost of implementation:

Sometime it may happen that the code we have written now is not in use, and by the time we really need it the library itself could provide it.

**DRY:**

While duplication: we actually duplicate the code as well as the effort.

DRY is about having each piece of information and behavior in a **single, sensible place.**

Dry is not just about avoiding duplicate code but also about making sure that each piece of information and behavior in your system has a single, clear place where it exists.

DRY can be also considered while defining use cases and gathering requirements.

**Encapsulate What Varies:** In the code example.

**Delegation Principle:** In the code example.

**Code to interface:** Coding to an interface rather than an implementation makes your software easier to extend. Example: Player

**SOLID principles:**

**SRP:** single responsible principle is similar as a class should not change for more than one reason.

To verify this one should not be able to say that a class or method is doing this and ……

A class or method should be doing only one thing at a time.

Long methods are bad and mostly compensated with inline code comments.

Long methods can’t be defined by no of lines.

SRP can be practiced in the below way:

The \_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_ itself.

Fill in the first blank with class name and second with method name, it would give some idea whether the class follows SRP or not.

**OCP (Open-close principle) -** OCP is achieved usually by extending the super class. Hence you don’t allow base class to be modified but at the same time sub class can define their specific changes.

It’s a combination of encapsulation and abstraction. The behavior that gets change is abstracted in the base class. But when you need changes your subclasses handles the changes by extending the base classes. That’s where encapsulation comes in: encapsulates what varies (behavior in the subclasses) away from what stays the same (the common behavior in base class).

So, is it the only way to use the OCP is by extending the base class?

A: Other example: If you had several private methods in a class, those are closed for modification - no other code can mess with that. But, when you could add public methods that invoked those private methods in different ways. You’re extending the behavior of private methods without changing them. That’s another example of OCP.

Example: in code demo.

**LSP Liksov’s substitution principle:**

**Composition over inheritance:**

* Java doesn’t support multiple inheritance.
* LSP
* The implementation of classes is decoupled.
* Lazy loading.
* Composition support better test ability: If one class is composed of another class, you can easily create [Mock Object](http://javarevisited.blogspot.sg/2014/04/difference-between-stub-and-mock-object-java-junit.html) representing composed class for sake of testing. Inheritance doesn't provide this luxury. In order to test derived class, you must need its super class.

Read more: <http://javarevisited.blogspot.com/2013/06/why-favor-composition-over-inheritance-java-oops-design.html#ixzz466nxa7VF>

**Interface segregation Principle:** When we design an application we should take care how we are going to make abstract a module which contains several submodules. Considering the module implemented by a class, we can have an abstraction of the system done in an interface. But if we want to extend our application adding another module that contains only some of the submodules of the original system, we are forced to implement the full interface and to write some dummy methods. Such an interface is named fat interface or polluted interface. Having an interface pollution is not a good solution and might induce inappropriate behavior in the system.

The **Interface Segregation Principle** states that clients should not be forced to implement interfaces they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

Ex : Clock -> time, alarm, radio : move it from clock interface to timepiece, alarm and radio interfaces.

**Coupling**: worst form of coupling is inheritance.

Two things to get deal with coupling:

* Get rid of it.
* Make it loose instead of tight.

**Dependency Inversion Principle:**

**<http://stackoverflow.com/questions/62539/what-is-the-dependency-inversion-principle-and-why-is-it-important>**

**<http://martinfowler.com/articles/injection.html>**

**Interface as a contract:** The sole purpose of an interface is abstraction. It is used as either a contract to implement multiple methods across multiple classes. Or it can be used as reference to a client on how the implementation should work.

If one doesn’t want the implementation to be instantiated rather want the interface to be used. He can write a factory class to return a private reference of an interface having the implementation object. May be spring already supports it:

public class Lookup {

private static final Foo foo = new FooImpl();

public static Foo getFoo() {

return foo;

}

}

Foo is an interface, FooImpl an implementation class (which can be package private if you want to enforce that it can't be instantiated by clients)

**Program to contract:** When you are programming by contract, you’re working with client code to agree on how you’ll handle problem situation. For ex: you are returning null if something is not found and your client want it to be changed throw an exception.

**Defensive programming:** When programming defensively, you’re making sure the client gets a safe response, no matter what the client wants to have. For ex: Your client wants you to throw runtime exception if something is not found. But, you don’t trust your client and throwing a checked exception to make sure the client handles the exception.

***Additional***

**Encapsulation (private variables with getters and setters):**

* When you realize you need to do more than just set and get the value, you don't have to change every file in the codebase.
* You can perform validation here.
* You can change the value being set.
* You can hide the internal representation. getAddress() could actually be getting several fields for you.
* You've insulated your public interface from changes under the sheets.
* Some libraries expect this. Reflection, serialization, mock objects.
* Inheriting this class, you can override default functionality.
* You can have different access levels for getter and setter.
* Lazy loading.

**Encapsulation (hiding data for protection and separation of concern) vs abstraction (hiding information not needed):**

Encapsulation hiding data advantage:

Encapsulation in simple words

Wrapping data and methods within classes in combination with implementation hiding (through access control) is often called encapsulation. The result is a data type with characteristics and behaviors. Encapsulation essentially has both i.e. information hiding and implementation hiding.

Going deep down in concept

I read it somewhere: “Whatever changes, encapsulate it“. It has been quoted as a famous design principle. For that matter in any class, changes can happen in data in runtime and changes in implementation can happen in future releases. So, encapsulation applies to both i.e. data as well as implementation.

Access control or implementation hiding puts boundaries within a data type or class for two important reasons. The first is to establish what the client programmers can and can’t use. This feeds directly into the second reason, which is to separate the interface from the implementation.

If you are sure that client programmers can’t do anything but send messages to the public interface, then you are free to change anything that’s not public (e.g., package access, protected, or private) without breaking client code. Encapsulation helps you in achieving this surety.

Relation with abstraction with example

If you have gone through [my last post](http://howtodoinjava.com/object-oriented/understanding-abstraction-in-java/), you will see that abstraction is essentially an idea, which helps in setting the guidelines. Encapsulation is the mechanism by which we achieve the desired abstraction.

In short, from OOAD perspective:   
– Abstraction is more about ‘What’s class can do. [Idea]  
– Encapsulation is more about ‘How ‘to achieve that functionality. [Implementation]

I have seen many contradictions to this theory over many blogs. So, if you also don’t agree with this, please bear with me. Also, i will request you to put a comment your idea related to topic. I will happily try to relate or negate.

Going forward, I will take example of our well known class HashMap [Read: [How HashMap works internally?](http://howtodoinjava.com/core-java/interviews-questions/how-hashmap-works-in-java/)]. This class is responsible for storing key-value pair, searching based on key and do more things. From outside, client code only knows the method names and their behavior. It calls these methods and live happily. This is actually what abstraction guidelines are. Abstraction says that client code should call a method to add key-value pair, a method to retrieve value based on key and so on. How it should be done? is not business of abstraction.

And here comes encapsulation, when you start writing actual code. You write HashMap.Entry class and create variable tableof type Entry[]. Then you declare all such things private and give public access to only put() and get() methods etc. This is actually encapsulation. A realization of your desired abstraction.

**Why a single class need to implement an interface?**

Answer: Personally, I feel it over engineered. Having single class implementing interface for dao layers is a good idea as it could help in easily migrating to some other orm tool. Similarly, having it in platform\_common makes it loosely coupled. But when it comes to gid service, it’s not needed for the below reasons:

It forces the method to be public unnecessary.

YAGNI principle – don’t do until not needed.

Java has enough boilerplate code as it is.

There are mocking frameworks nowadays which can easily mock concrete classes.

Not needed when the when the class is not a public api.

You can add it at a later point of time if you really need it.

Writing comments for public behavior of both interface and class is again redundant.

**Dynamic binding, polymorphism, and change**

When you establish an inheritance relationship between two classes, you get to take advantage of *dynamic binding* and *polymorphism.*Dynamic binding means the JVM will decide at runtime which method implementation to invoke based on the class of the object. Polymorphism means you can use a variable of a superclass type to hold a reference to an object whose class is the superclass or any of its subclasses.