10. Clean Code: Coupling, Cohesion, Modularity, DRY, KISS, YAGNI

Maintainable Code

Writing maintainable code helps increase productivity for developers. Having highly maintainable code makes it easier to design new features and write code. Modular, component-based, and layered code increases productivity and reduces risk when making changes.

The key to creating maintainable code is adhering to “low coupling, high cohesion”.
But what exactly does this mean? At what point is your code loosely coupled and highly cohesive?

Modularity

I refer to “modules” which represent almost any sort of language construct you have. In Object Oriented languages, this may represent classes. In JavaScript, it may represent actual packages or domains.

Some Benchmarks for modularity:

1. How many times are you rewriting some code for doing a particular task?
2. How much do you have to refactor your code when you change some part of your program?
3. Are the files **small and easy** to navigate through?
4. Are the application modules performing adequately and independently as and when required?
5. Is your code **minimally disastrous**? Do you get 10 error by making a small change in the code? Do you get 20-odd errors upon re-naming a class?
6. How near is the code to **natural language usage?** (i.e. modules and their subcomponents represent more real world objects without giving much concern to net source file size).

The **basic philosophy** is to break down your application into as small of code fragments as possible, arranged neatly across a multitude of easily understandable and accessible directory layouts.

Each method in your application **must do no more than the minimum quanta of processing needed**. Combining these methods into more and more macro level methods should lead you back to your application.

Low Coupling

How much do your different modules depend on each other?

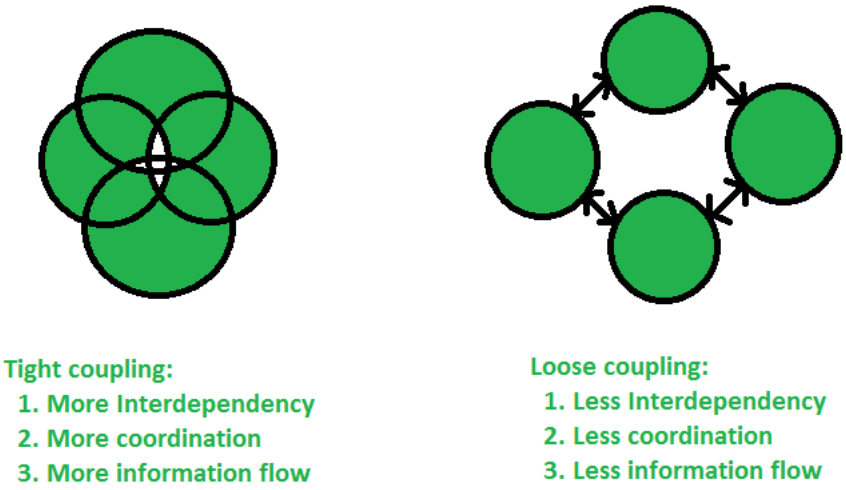
Modules should be as independent as possible from other modules, so that changes to a module don’t heavily impact other modules.

High coupling would mean that your module knows the way too much about the inner workings of other modules. Modules that know too much about other modules make changes hard to coordinate and make modules brittle. If Module A knows too much about Module B, changes to the internals of Module B may break functionality in Module A.

Think of User module and Notification module. User module shouldn’t know or care about internal working of Notification module.

By aiming for low coupling, you can easily make changes to the internals of modules without worrying about their impact on other modules in the system. Low coupling also makes it easier to design, write, and test code since our modules are not interdependent

on each other. We also get the benefit of easy to reuse and compose-able modules. Problems are also isolated to small, self-contained units of code.



Difference between tight coupling and loose coupling

- Tight coupling is not good at the test-ability. But loose coupling improves the test ability.
- Tight coupling does not provide the concept of interface. But loose coupling helps us follow the GOF principle of program to interfaces, not implementations.
- In Tight coupling, it is not easy to swap the codes between two classes. But it's much easier to swap other pieces of code/modules/objects/components in loose coupling.
- Tight coupling does not have the changing capability. But loose coupling is highly changeable.

There are two types of coupling:

1. **Tight coupling :** In general, Tight coupling means the two classes often change together. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.**Example :** If you want to change the skin, you would also have to change the design of your body as well because the two are joined together – they are tightly coupled. The best example of tight coupling is RMI(Remote Method Invocation).

```
// Java program to illustrate
// tight coupling concept
class Subject {
  Topic t = new Topic();
  public void startReading()
  {
    t.understand();
  }
}
class Topic {
  public void understand()
  {
    System.out.println("Tight coupling concept");
  }
  public void read(){
Subject s = new Subject();
s.startReading();
}
}
```

Explanation: In the above program the Subject class is dependents on Topic class. In the above program Subject class is tightly coupled with Topic class it means if any change in the Topic class requires Subject class to change. For example, if Topic class understand() method change to gotit() method then you have to change the startReading() method will call gotit() method instead of calling understand() method.

```
// Java program to illustrate
// tight coupling concept
class Volume
{
  public static void main(String args[])
  {
    Box b = new Box(5,5,5);
    System.out.println(b.volume);
  }
}
class Box
{
  public int volume;
  Box(int length, int width, int height)
  {
    this.volume = length * width * height;
  }
}
```

Output:

```
125
```

Explanation: In the above example, there is a strong inter-dependency between both the classes. If there is any change in Box class then they reflects in the result of Class Volume.

2. **Loose coupling :** In simple words, loose coupling means they are mostly independent. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled. In order to over come from the problems of tight coupling between objects, spring framework uses dependency injection mechanism with the help of POJO/POJI model and through dependency injection its possible to achieve loose coupling.**Example :** If you change your shirt, then you are not forced to change your body – when you can do that, then you have loose coupling. When you can't do that, then you have tight coupling. The examples of Loose coupling are Interface, JMS.In general, Tight Coupling is bad in but most of the time, because it reduces flexibility and re-usability of code, it makes changes much more difficult, it impedes test ability etc. loose coupling is a better choice because A loosely coupled will help you when your application need to change or grow. If you design with loosely coupled architecture, only a few parts of the application should be affected when requirements change.

```
// Java program to illustrate
// loose coupling concept
public interface Topic
{
  void understand();
}

class Topic1 implements Topic {
public void understand()
  {
    System.out.println("Got it");
  }
}

class Topic2 implements Topic {
public void understand()
  {
    System.out.println("understand");
  }
}
public class Subject {
public static void main(String[] args)
  {
```

```
    Topic t = new Topic1();
    t.understand();
  }
}
```

Explanation : In the above example, Topic1 and Topic2 objects are loosely coupled. It means Topic is an interface and we can inject any of the implemented classes at run time and we can provide service to the end user.

```
// Java program to illustrate
// loose coupling concept
class Volume
{
  public static void main(String args[])
  {
    Box b = new Box(5,5,5);
    System.out.println(b.getVolume());
  }
}
final class Box
{
  private int volume;
  Box(int length, int width, int height)
  {
    this.volume = length * width * height;
  }
  public int getVolume()
  {
    return volume;

  }
}
```

Output:

```
125
```

Explanation : In the above program, there is no dependency between both the classes. If we change anything in the Box classes then we dont have to change anything in Volume class.

High Cohesion

We want to design components that are self-contained: independent, and with a single, well-defined purpose—

> Cohesion often refers to how the elements of a module belong together. Related code should be close to each other to make it highly cohesive.

Easy to maintain code usually has high cohesion. The elements within the module are directly related to the functionality that module is meant to provide. By keeping high cohesion within our code, we end up trying DRY code and reduce duplication of knowledge in our modules. We can easily design, write, and test our code since the code for a module is all located together and works together.

Low cohesion would mean that the code that makes up some functionality is spread out all over your code-base. Not only is it hard to discover what code is related to your module, it is difficult to jump between different modules and keep track of all the code in your head.

By keeping code loosely coupled, we can write code within one module without impacting other modules. And by keeping code cohesive, we make it easier to write DRY code that is easy to work with.

A good way to determine how cohesive and coupled your code is, is illistrated by this quote from *The Pragmatic Programmer*:

> When you come across a problem, assess how localized the fix is. Do you change just one module, or are the changes scattered throughout the entire system? When you make a change, does it fix everything, or do other problems mysteriously arise?

While you are writing and working with your code base, ask yourself:

1. How many modules am I touching to fix this or create this functionality?
2. How many different places does this change need to take place?
3. How hard is it to test my code?
4. Can we improve this by making code more loosely coupled? Can this be improved by making our code more cohesive?

Difference between high cohesion and low cohesion:

- High cohesion is when you have a class that does a well-defined job. Low cohesion is when a class does a lot of jobs that don't have much in common.
- High cohesion gives us better-maintaining facility and Low cohesion results in monolithic classes that are difficult to maintain, understand and reduce re-usability

Pictorial view of high cohesion and low cohesion:

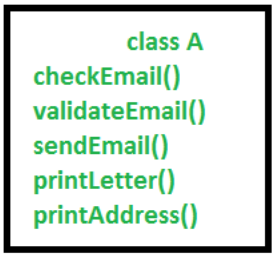


Fig: Low cohesion

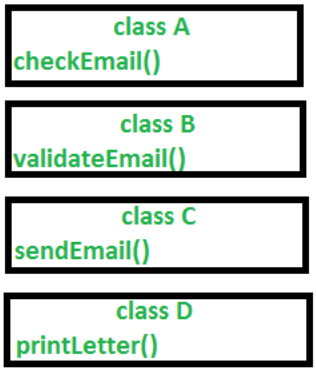


Fig: High cohesion

In the above image, we can see that in low cohesion only one class is responsible to execute lots of jobs that are not in common which reduces the chance of reusability and maintenance. But in high cohesion, there is a separate class for all the jobs to execute a specific job, which results in better usability and maintenance.

Example:
Suppose we have a class that multiplies two numbers, but the same class creates a pop-up window displaying the result. This is an example of a low cohesive class because the window and the multiplication operation don't have much in common. To make it high cohesive, we would have to create a class Display and a class Multiply. The Display will call Multiply's method to get the result and display it. This way to develop a high cohesive solution.

```
// Java program to illustrate
// high cohesive behavior

class Multiply {

  int a = 5;
  int b = 5;
```

```
  public int mul(int a, int b)
  {
    this.a = a;
    this.b = b;
    return a * b;
  }
}

class Display {
  public static void main(String[] args)
  {
    Multiply m = new Multiply();
    System.out.println(m.mul(5, 5));
  }
}
```

The DRY Principle: Don't Repeat Yourself

DRY stand for "Don't Repeat Yourself," a basic principle of software development aimed at reducing repetition of information. The DRY principle "Every piece of knowledge or logic must have a single, unambiguous representation within a system."

//some operation

common();

//continue with the operation

//some other operation

common();

//continue with the other operation

private method common(){

//open a connection

//pass a url (change the port). //

}

Violations of DRY

"We enjoy typing" (or, "Wasting everyone's time."): "We enjoy typing," means writing the same code or logic again and again. It will be difficult to manage the code and if the logic changes, then we have to make changes in all the places where we have written the code, thereby wasting everyone's time.

How to Achieve DRY

To avoid violating the DRY principle, divide your system into pieces. Divide your code and logic into smaller reusable units and use that code by calling it where you want. Don't write lengthy methods, but divide logic and try to use the existing piece in your method.

DRY Benefits

Less code is good: It saves time and effort, is easy to maintain, and also reduces the chances of bugs.

One good example of the DRY principle is the helper class in enterprise libraries, in which every piece of code is unique in the libraries and helper classes.

KISS: Keep It Simple, Stupid

The KISS principle is descriptive to keep the code simple and clear, making it easy to understand. After all, programming languages are for humans to understand — computers can only understand 0 and 1 — so keep coding simple and straightforward. Keep your methods small. Each method should never be more than 40-50 lines.

Each method should only solve one small problem, not many use cases. If you have a lot of conditions in the method, break these out into smaller methods. It will not only be easier to read and maintain, but it can help find bugs a lot faster.

Violations of KISS

We have all likely experienced the situation where we get work to do in a project and found some messy code written. That leads us to ask why they have written these unnecessary lines. Just have a look at below two code snippets shown below. Both methods are doing the same thing. Now you have to decide which one to use:

```
public String weekday1(int day) {
    switch (day) {
        case 1:
            return "Monday";
        case 2:
            return "Tuesday";
        case 3:
            return "Wednesday";
        case 4:
            return "Thursday";
        case 5:
            return "Friday";
        case 6:
            return "Saturday";
        case 7:
            return "Sunday";
        default:
            throw new InvalidOperationException("day must be in range 1 to 7");
    }
}

public String weekday2(int day) {
    if ((day < 1) || (day > 7)) throw new InvalidOperationException("day must be in range 1 to 7");

    string[] days = {
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    };

    return days[day - 1];
}
```

How to Achieve KISS

To avoid violating the KISS principle, try to write simple code. Think of many solutions for your problem, then choose the best, simplest one and transform that into your code. Whenever you find lengthy code, divide that into multiple methods — right-click and refactor in the editor. Try to write small blocks of code that do a single task.

Benefit of KISS

If we have some functionality written by one developer and it was written with messy code, and if we ask for another developer to make modifications in that code, then first, they have to understand the code. Obviously, if the code is written simply, then there will not be any difficulty in understanding that code, and also will be easy to modify.

YAGNI Principle

YAGNI stands for *You Ain't Gonna Need It*. It's a principle from software development methodology of Extreme Programming (XP). This principle says that you should not create features that it's not really necessary.

This principle it's similar to the KISS principle, once that both of them aim for a simpler solution. The difference between them it's that YAGNI focus on removing unnecessary functionality and logic, and KISS focus on the complexity.

> Always implement things when you actually need them, never when you just foresee that you need them.

It means that you should not implement functionality just because you think that you may need it someday, but implement it just when you really need it. Doing that you will avoid spending time with implementations that were not even necessary, and maybe will never be used.

References:

https://medium.com/clarityhub/low-coupling-high-cohesion-3610e35ac4a6
http://geeksforgeeks.org

https://javarevealed.wordpress.com/tag/singleton-and-serialization/

https://stackoverflow.com/questions/26285520/implementing-singleton-with-an-enum-in-java

https://www.javatpoint.com/volatile-keyword-in-java

https://dzone.com/articles/software-design-principles-dry-and-kiss

https://www.educative.io/blog/solid-principles-oop-c-sharp?aid=5082902844932096&utm_source=google&utm_medium=cpc&utm_campaign=blog-dynamic&utm_term=&utm_campaign=Dynamic+-+Blog&utm_source=adwords&utm_medium=ppc&hsa_acc=5451446008&hsa_cam=8090938743&hsa_grp=82569843726&hsa_ad=3837938538428&hsa_kw=&hsa_mt=&hsa_net=adwords&hsa_ver=3&gclid=Cj0KCQiAhf2MBhDNARIsAKXU5GRo9Nhfu8YQ2CkZThj