https://www.interviewbit.com/blog/solid-principles-java/

The word SOLID acronym for:

* Single Responsibility Principle (SRP)
* Open-Closed Principle (OCP)
* Liskov Substitution Principle (LSP)
* Interface Segregation Principle (ISP)
* Dependency Inversion Principle (DIP)

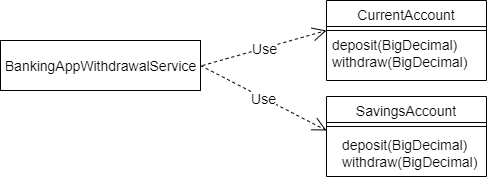
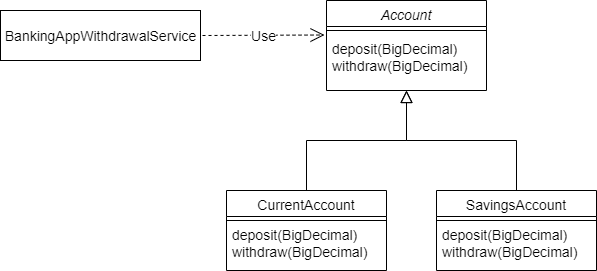
**Single-responsibility Principle (SRP) states:**

**A class should have one and only one reason to change, meaning that a class should have only one job.**

**Example:**

Suppose we have a class FinalExam that has 3 methods that perform 3 operations AddQuestion(), ExpectedAnswer(), Marksdistribution(). Now all these 3 methods perform different actions. By using the single responsibility principle, we can separate these functionalities into three separate classes to fulfil the goal of the principle.

**Open-closed Principle (OCP) states:**

**Objects should be open for extension but closed for modification (**alteration**).**

NOT OCP

OCP

**Liskov Substitution Principle states:**

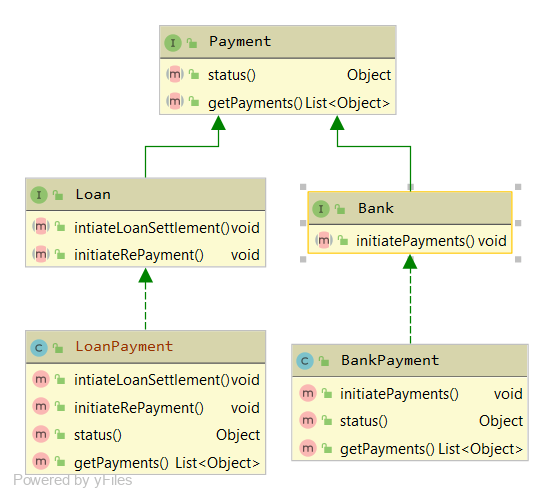
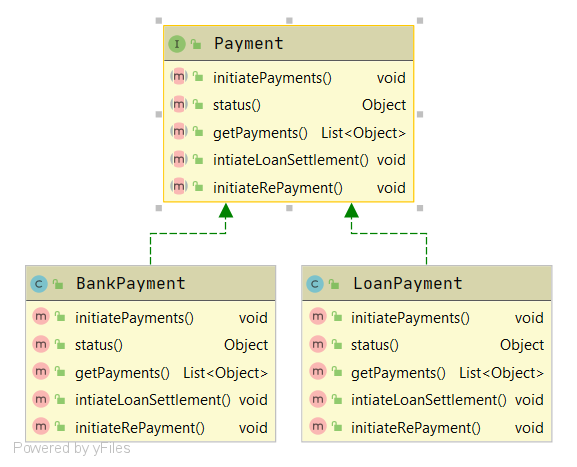
**This means that every child class should be substitutable (exchangeable) for their parent class.**

For example, if MySubclass is a subclass of MyClass, you should be able to replace MyClass with MySubclass without bunging up the program

Let's refactor the code to make "good" design using LSP?

* Refer "MediaPlayer.java" parent class and its child class "AudioMediaPlayer.java" having play audio ability
* Refer "VideoMediaPlayer.java" extends "MediaPlayer.java" and adds play video ability
* Refer "DivMediaPlayer.java", "VlcMediaPlayer.java". Both extends "VideoMediaPlayer.java" for play audio and video ability.
* Refer "WinampMediaPlayer.java" which extends "AudioMediaPlayer.java" for play audio ability.
* So client program can substitute "VideoMediaPlayer.java" super type with "DivMediaPlayer.java" or "VlcMediaPlayer.java", but not with "WinampMediaPlayer.java"

Interface segregation principle states:

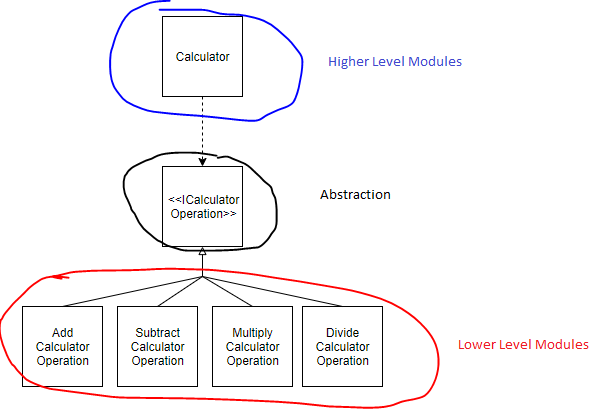
A client should never be forced to implement an interface that it doesn’t use, or clients shouldn’t be forced to depend on methods they do not use.

ok

x

Dependency inversion principle states:

1. High-level modules should not depend on low-level modules. Both should depend on the abstraction.
2. Abstractions should not depend on details. Details should depend on abstractions.



**Ex. Abstractions don't depend on details.**

For me, it doesn't matter whether my engine has changed or not, I still should be able to drive my car the same way.

STUPID is a mnemonic acronym for:

* [Singleton](https://it.badykov.com/blog/2020/03/08/stupid-principles/#singleton)
* [Tight Coupling](https://it.badykov.com/blog/2020/03/08/stupid-principles/#tight-coupling)
* [Untestability](https://it.badykov.com/blog/2020/03/08/stupid-principles/#untestability)
* [Premature Optimization](https://it.badykov.com/blog/2020/03/08/stupid-principles/#premature-optimization)
* [Indescriptive Naming](https://it.badykov.com/blog/2020/03/08/stupid-principles/#indescriptive-naming)
* [Duplication](https://it.badykov.com/blog/2020/03/08/stupid-principles/#duplication)

**Singleton**

* Programs that use global state are very difficult to test;
* Programs that depend on global status hide their dependencies.
* Often you can replace using a singleton with something better.
* Avoiding everything static is very important to prevent strong coupling.

**Tight Coupling**

* Tight coupling is a generalization of the singleton problem.
* Coupling is a measure of how related routines or modules are.
* If making a change in one module in your application requires you to change another module, then coupling exists. For example, you instantiate objects in the class of your constructor instead of passing instances as parameters. This is bad because it does not allow further changes, such as replacing an instance with an instance of a subclass, a mock object, or whatever.
* Strongly coupled modules are difficult to reuse, and also difficult to test.

**Untestability**

In most cases, the impossibility of testing is caused by a tight coupling.

**Premature Optimization**

Donald Ervin Knuth said:

*Premature optimization is the root of all evil. Only costs alone, and no good.*

Optimized systems are much more complex than just writing a loop or using pre-increment instead of post-increment. You will end up with unreadable code. That is why system optimization is much harder.

There are two rules for optimizing an application: Do not do this; (only for professionals!) do not optimize this yet.

**In descriptive Naming**

* Name your classes, methods, attributes, and variables appropriately.
* Do not abbreviate them!
* Write code for people, not for machines.
* Computers only understand 0 and 1.

**Duplication**

Duplicated code is inefficient.

**Summary**

In this post, I tried to briefly and clearly explain what principles you should avoid when writing code.