S = Single Responsibility Principle

O = Opened Closed Principle

L = Liscov Substitution Principle

I = Interface Segregation Principle

D = Dependency Inversion Principle

Topic: Single Responsibility Principle



***Single Responsibility Principle poster***

It says, "just because you can implement all the features in a single device, you shouldn't". Why? Because it adds a lot of manageability problems for you in the long run.

Let me tell you the principle in Object Oriented terms.

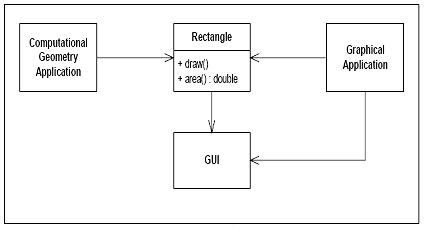
"**There should never be more than one reason for a class to change.**"

Or, differently said: "A class should have one and only one responsibility".

this principle says that if you have a class that has more than one reason to change (or has more than one overall responsibility), you need to split the class into multiple classes based upon their responsibility.

It's important because:

* Each responsibility is an axis of change.
* Code becomes coupled if classes have more than one responsibility.



Here, the Rectangle class does the following:

* It calculates the value of the rectangular area.
* It renders the rectangle in the UI.

And, two applications are using this Rectangle class:

* A computational geometry application uses this class to calculate the area
* A graphical application uses this class to draw a rectangle in the UI

This is violating the SRP (Single Responsibility Principle)!

: You see, the Rectangle class is actually performing two different things. It is calculating the area in one method, and it is returning a GUI representation of the rectangle in another method. This leads to some interesting problems:

* We must include the GUI in the computational geometry application. While deploying the geometry application, we must include the GUI library.
* A change to the Rectangle class for the graphical application may lead to a change, build, and test for the computational geometry application, and vice-versa.

Separate the responsibilities into two different classes, such as:

* Rectangle: This class should define the area() method.
* RectangleUI: This class should inherit the Rectangle class and define the Draw() method.

Topic: Open-Closed Principle

**Shubho**: Here goes the poster for the Open-Closed Principle:



**Figure**: Open Closed Principle poster

If I need to explain it in design oriented terms, it would be as follows:

"**Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.**"

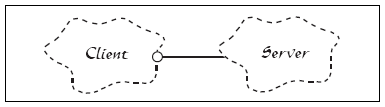
At the most basic level, that means, you should be able to extend a class's behavior without modifying it. It's just like I should be able to put on a dress without doing any change to my body, ha ha.

**Farhana**: Interesting. You can change your look by putting any dress you want, and you don't have to change your body for that. So you are open for extension, right?

**Shubho**: Yes. In OOD, open for extensions means that the behavior of the module/class can be extended and we can make the module behave in new and different ways as the requirements change, or to meet the needs of new applications.

**Farhana**: And your body is closed for modification. I like this example. So, the source code for a core class or module should not be modified when it needs an extension. Can you explain with some examples?

**Shubho**: Sure, take a look at the following example. This doesn't support the "Open-Closed" principle:

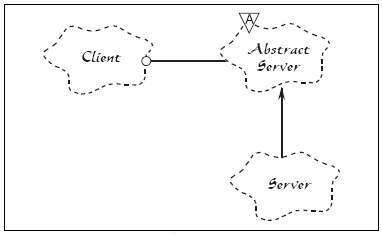


***Class hierarchy showing violation of Open Closed principle***

You see, the Client and Server classes are both concrete. So, if for any reason, the server implementation is changed, the client also needs a change.

**Farhana**: Makes sense. If a browser is implemented as tightly coupled to a specific server (such as IIS), then if the server is replaced for some reason with another one (say, Apache) the browser also needs to be changed or replaced. That would be really horrible!

**Shubho**: Correct. So following would be the correct design:



***Class hierarchy showing Open Closed Principle***

In this example, there is an Abstract Server class added, and the client holds a reference to the abstract class, and the Concrete Server class implements the Abstract Server class. So, if for any reason the Server implementation is changed, the Client is not likely to require any change.

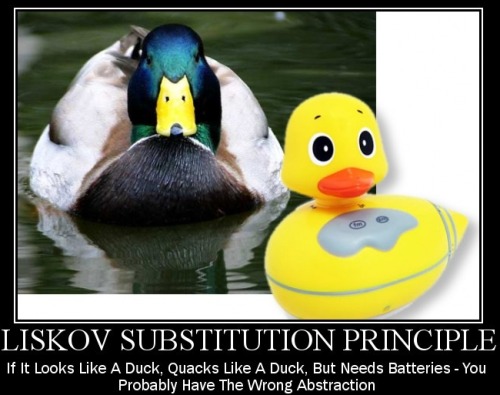
The Abstract Server class here is closed for modification, and the concrete class implementations here are **open**for extension.

**Farhana**: As I understand, abstraction is the key, right?

**Shubho**: Yes, basically, you abstract the things that are the core concepts of your system, and if you do the abstraction well, most likely, it would require no change when the functionality is to be extended (such as the server is an abstract concept). And, you define the abstract things in the implementations (say, IISServer implements the Server) and code against abstractions (Server) as much as possible. This would allow you to extend the abstract thing and define a new implementation (say, ApacheServer) without doing any change in the client code.

Topic: Liskov's Substitution Principle

**Shubho**: The name "Liskov's Substitution Principle" sounds very heavy, but the idea is pretty basic. Take a look at this interesting poster:



***Liskov Substitution Principle poster***

The principle says:

"**Subtypes must be substitutable for their base types.**"

Or, if said differently:

"Functions that use references to base classes must be able to use objects of derived classes without knowing it."

**Farhana**: Sorry, sounds confusing to me. I thought this is the basic rule of OOP. This is polymorphism, right? Why was an Object Oriented Principle required on this issue?

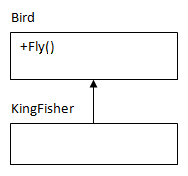
**Shubho**: Good question. Here is your answer:

In basic Object Oriented Principles, "Inheritance" is usually described as an "is a" relationship. If a "Developer" is a "SoftwareProfessional", then the "Developer" class should inherit the "SoftwareProfessional" class. Such "Is a" relationships are very important in class designs, but it's easy to get carried away and end up in a wrong design with a bad inheritance.

The "Liskov's Substitution Principle" is just a way of ensuring that inheritance is used correctly.

**Farhana**: I see. Interesting.

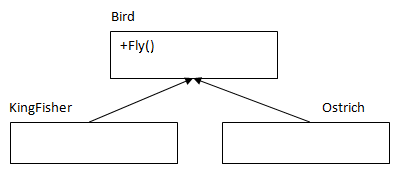
**Shubho**: Yes darling, indeed. Let's take a look at an example:



***Class hierarchy showing an example of Liskov Substitution Principle***

Here, the KingFisher class extends the Bird base class and hence inherits the Fly() method, which is pretty good.

Now take a look at the following example:



***Corrected class hierarchy of Liskov Substitution Principle***

Ostrich is a Bird (definitely it is!) and hence it inherits the Bird class. Now, can it fly? No! Here, the design violates the LSP.

So, even if in real world this seems natural, in the class design, Ostrich should not inherit the Bird class, and there should be a separate class for birds that can't really fly and Ostrich should inherit that.

**Farhana**: OK, understood. So, let me try to point out why the LSP is so important:

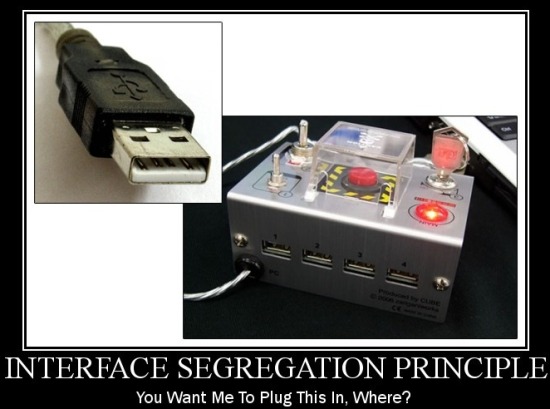
* If LSP is not maintained, class hierarchies would be a mess, and if a subclass instance was passed as parameter to methods, strange behavior might occur.
* If LSP is not maintained, unit tests for the base classes would never succeed for the subclass.

Am I right?

**Shubho**: You are absolutely right. You can design objects and apply LSP as a verification tool to test the hierarchy whether inheritance is properly done.

Topic: The Interface Segregation Principle

**Shubho**: Today, we will learn the "Interface Segregation Principle". Here is the poster:



***Interface Segregation Principle poster***

**Farhana**: What does it mean?

**Shubho**: It means the following:

"**Clients should not be forced to depend upon interfaces that they do not use.**"

**Farhana**: Explain please.

**Shubho**: Sure, here is your explanation:

Suppose you want to purchase a television and you have two to choose from. One has many switches and buttons, and most of those seem confusing and doesn't seem necessary to you. Another has a few switches and buttons, which seems familiar and logical to you. Given that both televisions offer roughly the same functionality, which one would you choose?

**Farhana**: Obviously the second one with the fewer switches and buttons.

**Shubho**: Yes, but why?

**Farhana**: Because I don't need the switches and buttons that seem confusing and unnecessary to me.

**Shubho**: Correct. Similarly, suppose you have some classes and you expose the functionality of the classes using interfaces so that the outside world can know the available functionality of the classes and the client code can be done against interfaces. Now, if the interfaces are too big and have too many exposed methods, it would seem confusing to the outside world. Also, interfaces with too many methods are less re-usable, and such "fat interfaces" with additional useless methods lead to increased coupling between classes.

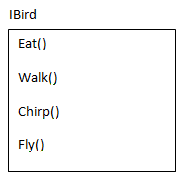
This also leads to another problem. If a class wants to implement the interface, it has to implement all of the methods, some of which may not be needed by that class at all. So, doing this also introduces unnecessary complexity, and reduces maintainability or robustness in the system.

The Interface Segregation principle ensures that Interfaces are developed so that each of them have their own responsibility and thus they are specific, easily understandable, and re-usable.

**Farhana**: I see. You mean interfaces should contain only the necessary methods and not anything else?

**Shubho**: Exactly. Let's see an example.

The following interface is a "Fat interface" which violates the Interface Segregation principle:

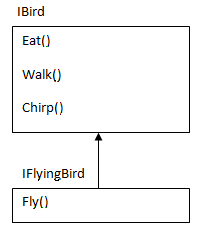


***Example of an interface violating the Interface Segregation principle***

Note that the IBird interface has many bird behaviours defined along with the Fly() behaviour. Now, if a Bird class (say, Ostrich) implements this interface, it has to implement the Fly() behaviour unnecessarily (Ostrich doesn't fly).

**Farhana**: That's correct. So, this interface must be split?

**Shubho**: Yes. The "Fat Interface" should be broken down into two different interfaces, IBird and IFlyingBird, where the IFlyingBird inherits IBird.



***Correct version of the interface in the Interface Segregation principle example***

If there is a bird that can't fly (say, Ostrich), it would implement the IBird interface. And if there is a bird that can fly (say, KingFisher), it would implement the IFlyingBird interface.

**Farhana**: So, if I go back to the example with the television with many switches and buttons, the manufacturer of that television must have a blueprint of that television where the switches and buttons are included in the plan. Whenever they want to create a new model of the television, if they need to re-use this blueprint, they would need to create as many switches and buttons as included in the plan. That wouldn't allow them to re-use the plan, right?

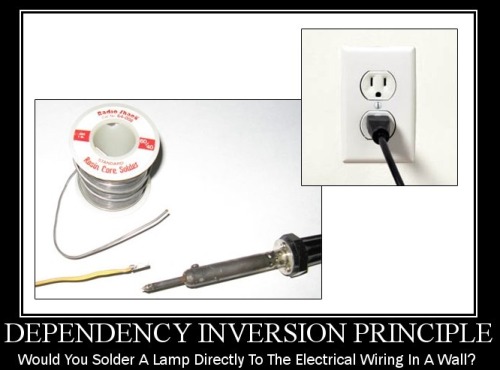
**Shubho**: Right.

**Farhana**: And, if they really want to re-use their plans, they should divide their television's blueprint into smaller pieces so that they can re-use the plan whenever any new kind of television is to be created.

**Shubho**: You got the idea.

Topic: The Dependency Inversion Principle

**Shubho**: This is the last principle among the SOLID principles. Here is the poster:



***Dependency Inversion principle poster***

It says..

"**High level modules should not depend upon low level modules. Rather, both should depend upon abstractions.**"

**Shubho**: Let's consider a real world example to understand it. Your car is composed of lots of objects like the engine, the wheels, the air conditioner, and other things, right?

**Farhana**: Yes, of course.

**Shubho**: OK, none of these things are rigidly built within a single unit; rather, each of these are "pluggable" so that when the engine or the wheel has problem, you can repair it (without repairing the other things) and you can even replace it.

While replacement, you just have to ensure that the engine/wheel conforms to the car's design (say, the car would accept any 1500 CC engine and will run on any 18 inch wheel).

Also, the car might allow you to put a 2000 CC engine in place of the 1500 CC, given the fact that the manufacturer (say, Toyota) is the same.

Now, what if the different parts of your car were not built in such a "pluggable nature"?

**Farhana**: That would be horrible! Because, in that case, if your car's engine is out of order, you will have to fix the whole car or purchase a new one!

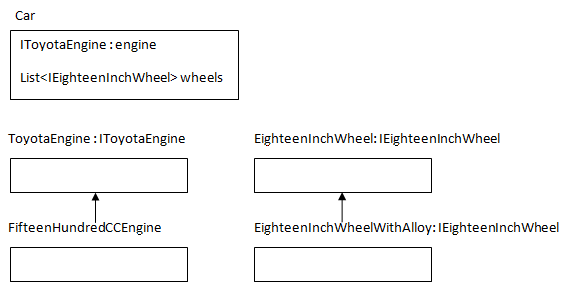
**Shubho**: Yes. Now, how can the "pluggable nature" be achieved?

**Farhana**: "Abstraction" is the key, right?

**Shubho**: Yes. In real world, Car is the higher level module/entity, and it depends on the lower level modules/entities like the Engines or Wheels.

Rather than directly depending on the Engines or Wheels, the car depends on the abstraction of some specification of Engine or Wheels so that if any the Engine or Wheel conforms to the abstraction, these could be assembled with the car and the car would run.

Let's take a look at the following class diagram:



***Dependency Inversion principle class hierarchy***

**Shubho**: In the above Car class, notice that there are two properties, and both of these are of abstract type (Interface) rather than concrete type.

The Engine and Wheels are pluggable because the car would accept any object implementing the declared interfaces, and that will not require any change in the Car class.

**Farhana**: So, if Dependency Inversion is not implemented in the code, we run the risk of:

* Damaging the higher level code that uses the lower level classes.
* Requiring too much time and effort to change the higher level code when a change occurs in the lower level classes.
* Producing less-reusable code.

**Shubho**: You got it perfect darling!

Summary

**Shubho**: There are many other Object Oriented principles other than the SOLID principles. Some are:

* "Composition over Inheritance": This says about favoring composition over inheritance.
* "Principle of least knowledge": This says that "the less your class knows, the better".
* "The Common Closure principle" : This says that "related classes should be packaged together".
* "The Stable Abstractions principle": This says that "the more stable a class is, the more it must consist of abstract classes."