4.96 / 5, 19 votes

First Posted 12 Jul 2010
Views 3,545
Bookmarked 25 times
Licence CPOL
Architect, Dev, Design, Beginner, Intermediate, Advanced, All-Topics, General

How I explained OOD to my wife

By **M.M.Al-Farooque Shubho** | 13 Jul 2010 | Unedited contribution

Learning Object Oriented Design through some interesting conversations

Introduction

My wife Farhana wants to resume her career as a software developer (She started her career as a Software developer, but, couldn't proceed much because of our first child's birth), and these days I am is trying to help her learning Object Oriented Designs as I've got a little experience in software designs and development.

Since my early days in Software development, I have always observed that no matter how hard a technical issue seems, it becomes always easier if explained from real life perspective and discussed in a conversational manner. As we had some fruitful conversations on Object Oriented Designs, I thought, I could share it because someone might find it as an interesting way of learning OOD.

Following is how our OOD conversation took place:

Topic : Introducing OOD

Shubho : Darling, let's start learning the Object Oriented Designs. You know Object Oriented Principles right?

Farhana : You mean, Encapsulation, Inheritance and polymorphism right? Yes.

Shubho : Yes. So, I hope you already know how to use classes and objects. Now, today we will learn Object Oriented Designs.

Farhana : Hold on a second. Isn't Object Oriented Principles enough to do Object Oriented programming? I mean, I can define classes and encapsulate the properties and methods. I can also define class hierarchy based upon their relationship. So, what's left?

Shubho : Good question. Object Oriented Principles and Object Oriented Design, these two are actually two different things. Let me give you a real life example to help you understand.

When you were child you learned alphabets first, right?

Farhana : Yes.

Shubho : OK. You also learned words and how to assemble the alphabets to make those meaningful words. Along with, you learned some basic grammars to assemble the words into sentences. For example, you had to maintain tenses and you had to use prepositions, conjunctions and others properly to create grammatically correct sentences. Say, a sentence like the following:

"I"(Pronoun) "want"(Verb) "to" (Preposition) "learn" (Verb) "OOD"(Noun)

You see, you are assembling the words in some order and you also chose the correct words to end up with a sentence that has some meaning.

Farhana : OK, so, what does it mean?

Shubho : This is analogous to the Object Oriented Principles. The OOP says about the basic principles, the core ideas to do Object Oriented programming. Here, OOP can be compared to the basic English grammars where the basic grammars teaches you how to construct a meaningful and correct sentence using the words and the OOP teaches you to construct the classes , encapsulate properties and methods inside them and also to develop hierarchy between them and use them in your code.

Farhana : Hm..I got the point. So, where OOD fits here?

Shubho : You'll get your answer soon. Now, suppose you need to write articles and essay on some topics. You may also wish to write books on different subjects that you have expertise on. So, knowing how to construct sentences is not enough to write good essays/articles or books, right? You need to write a lot and need to learn explaining things in good way so that, readers can understand easily what you are trying to mean.

Farhana : Seems interesting…carry on

Shubho : Now, if you want to write a book on a particular subject (Say, Learning Object Oriented Design), you got to know how to divide the overall subject into smaller topics. You also need to write chapters on those topics and you need to write prefaces, introductions, explanations, examples and many other paragraphs in the chapters. You need to design the overall book and learn some best practices of writing techniques so that, the reader can understand easily what you are trying to explain. So, this is about the bigger picture.

In Software development, OOD addresses the bigger picture. You need to design your software in a way where your classes and codes are modularized, re-usable and flexible and there are some good guidelines to do that so that you don't have to re-invent the wheels. There are some design principles that you can apply to design your classes and objects. Makes sense?

Farhana : Hm..got the initial idea, but, need to learn more.

Shubho : Don't worry, you will learn soon. Just let our discussion ongoing.

Topic : Why OOD?

Shubho : This is a very important question. Why should we care about Object Oriented Design where we can create some classes quick and finish development and deliver? Isn't that enough?

Farhana : Yes, I didn't know about OOD earlier and still I was able to develop and deliver the project. So, what is the big deal?

Shubho : OK, let me say a classic quote for you

"Walking on water and developing software from a specification are easy if both are frozen." - Edward V. Berard

Farhana : Hm..you mean software development specification keeps changing constantly?

Shubho : Exactly! The universal truth of Software is "Your software is bound to change",

Why?

Because, your software solves some real life business problems, and, the real life business process evolves and changes - always.

Your software does what it is supposed to do today and does it good enough. But, is your software smart enough to support "Changes"? If not, you don't have a smartly designed software.

Farhana : OK, so, please explain the "Smartly designed software" Sir!

Shubho : "A smartly designed software can adjust change easily, it can be extended and it is re-usable."

And, applying a good "Object Oriented Design" is the key to achieve such smart design.

Now, when can you claim that you have applied good OOD in your codes?

Farhana : That's my question too.

Shubho: You are doing Object Oriented Design if your codes are:

- Of course, object oriented.
- Re-usable
- Can be changed with minimal effort.
- Can be extended without changing existing codes.

Farhana : And..?

Shubho : We are not alone. Many people have strived already to achieve good Object Oriented Design and they already have identified some basic principles for doing Object Oriented Designs (Some basic inspirations which you can use to lay out your object oriented design). They also have identified some common design patterns that are applicable to some common scenario (Based upon the basic principles).

Farhana : Can you name some?

Shubho : Sure. There are many design principles out there, but, at the basic level, there are 5 principles, which are abbreviated as the SOLID principles (Thanks to Uncle Bob, the great OOD Mentor).

S = Single Responsibility Principle

O = Opened Closed Principle

L = Liscov Substitution Principle

I = Interface Segregation Principle

D = Dependency Inversion Principle

In the next discussions we will try to understand each of these in details

Topic : Single Responsibility Principle

Shubho : Let me show you the Poster first. These posters are interesting.

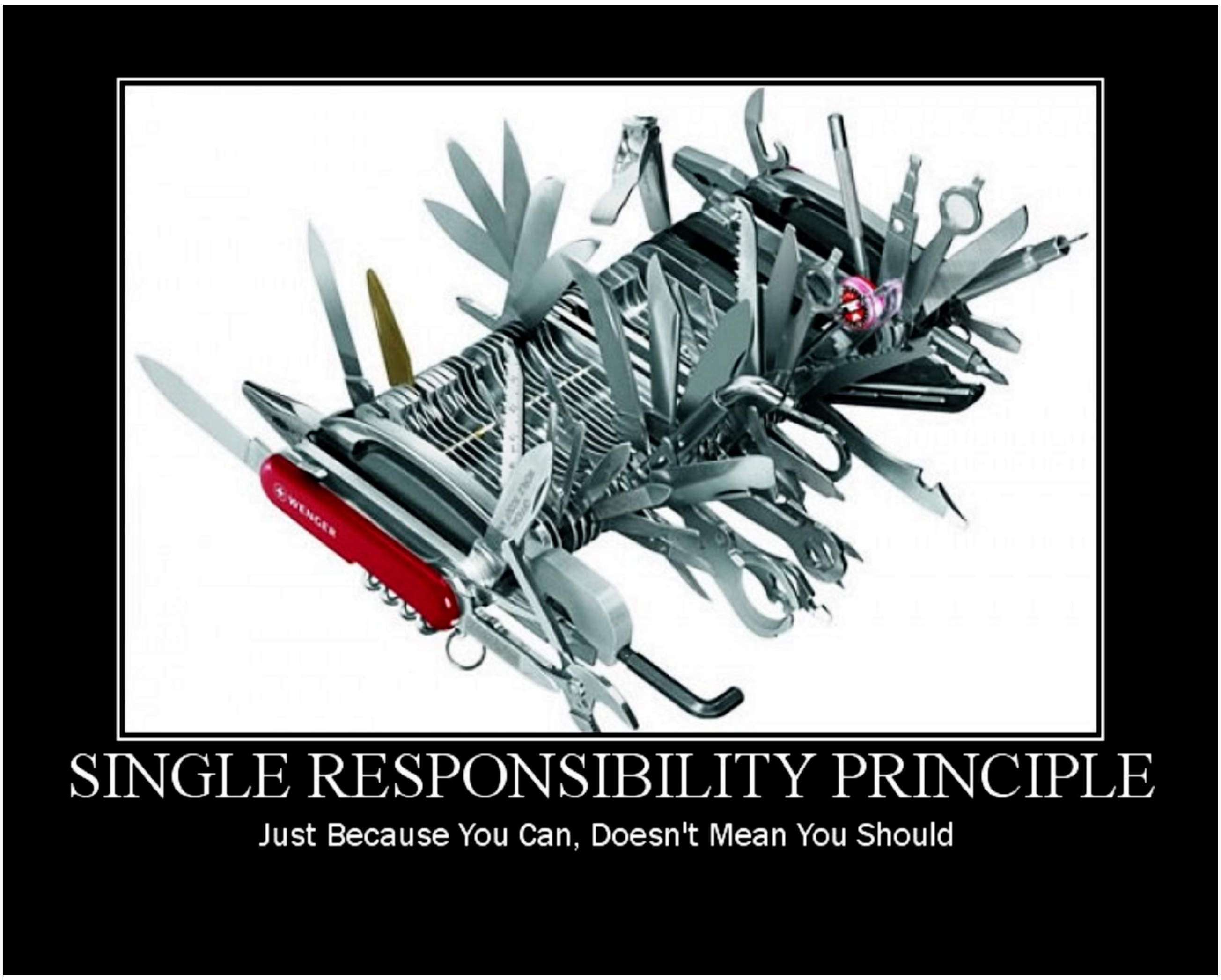
SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

Figure : Single Responsibility Principle

It says, "Just because you can implement all the features in a single device, you shouldn't". Why? Because, it adds 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.

Farhana : Can you please explain?

Shubho : Sure, 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.

Farhana : Hm..does that mean that I can't have multiple methods in a single class?

Shubho : Not at all. Rather, you surely can have multiple methods in a single class, issue is, they have to met a single purpose. Now, why splitting is important?

It's important because:

- Each responsibility is an axis of change.
- Codes become coupled if classes have more than one responsibility.

Farhana : Could you please give me an example?

Shubho : Sure, take a look at the following class hierarchy. Actually, this example is taken from Uncle Bob, so thanks to him again.

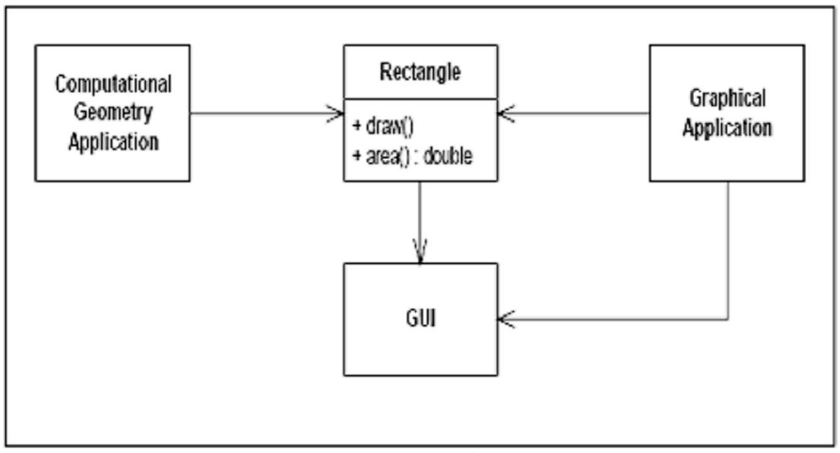

Figure : Class hierarchy showing violation of SRP principle

Here, the Rectangle class does the followings:

- It calculates the value of the Rectangular area.
- It renders the Rectangle in the UI.

And, two applications are using this Rectangle class:

- Computational Geometry Application uses this class to calculate the Area
- Graphical Application uses this class to draw a Rectangle in the UI

This is violating the SRP (Single Responsibility Principle) here!

Farhana : How?

Shubho : 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 following interesting problems:

- We must include the GUI in the computational geometry application. While deployment of the geometry application, we must include GUI library.

- A change to the Rectangle class for Graphical application may lead to change, build and test the Computational geometry application, and, vice-verse.

Farhana : Hm..its getting interesting. So, I guess we should break up the class based upon its responsibility right?

Shubho : Exactly. Can you predict what we should do?

Farhana : Sure, let me try. Following is what we might need to do:

Separate the responsibilities into two different classes, such as:

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

Shubho : Perfect. In this case, the Rectangle class will be used by the Computational geometry aplication and the RectangleUI class will be used by the Graphical application. We could even separate the classes into two separate dlls and that will allow us not to touch the other in case a change is needed to implement in one.

Farhana : Thanks, I think I understood the SRP. One thing, the SRP seems to be an idea of breaking things into molecular parts so that, it becomes reusable and it can be managed centrally. So, shouldn't we apply SRP in the method level also? I mean, we might have written methods that have many lines of codes for doing multiple things. These methods must be violating the SRP, no?

Shubho : You got it right. You should also break down your methods so that each method does a particular work. That will allow you to re-use the methods and in case any change is required, you are able to change by modifying a minimal amount of code.

Topic : Open-Closed Principle

Shubho : Here goes the poster for Open-Closed Principle

Figure : Open Closed Principle

If I need to explain it in design oriented terms, that 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 classes behavior, without modifying it. Its just like I should be able to put on a dress without doing any change in my body, ha ha.

Farhana : Hm..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 changes, 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:

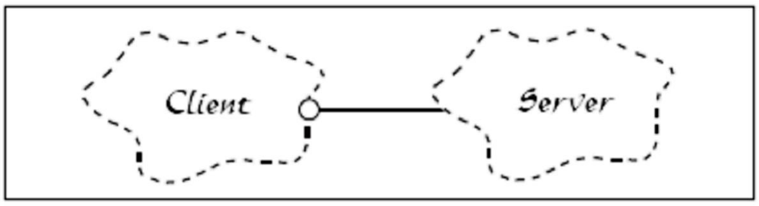

Figure : Class hierarchy showing violation of Open Closed principle

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

Farhana : Hm..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 reasons 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:

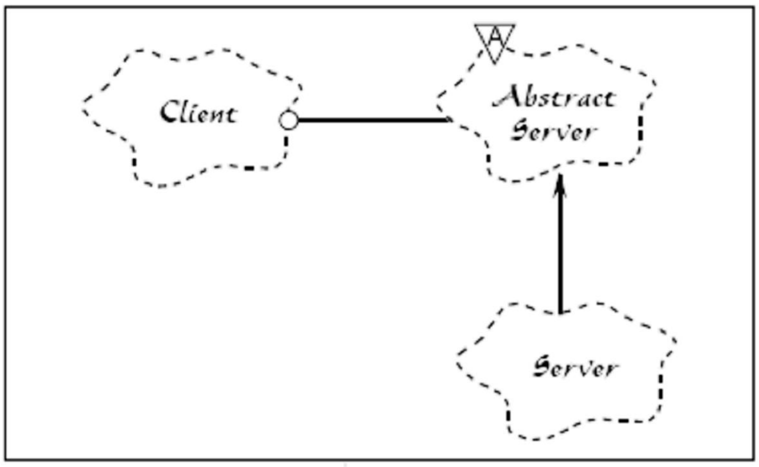

Figure : Class hierarchy showing Open Closed principle

In this example, there is an Abstract Server class added and client holds 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 likely not to require any change.

The Abstract Server class here is closed for modification and the Concrete class implementations here are Open for extension.

Farhana : So, as I understand, Abstraction is the key, right?

Shubho : Yes, basically, you abstract the things that are the core concept of your system and if you do the abstraction well, most likely that 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, the ApacheServer) without doing any change in the client code.

Topic : Liskov's Substitution Principle

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

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Figure : Liskov Substitution Principle

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, no? Why a Object Oriented Principle was required on this issue?

Shubho : Good question. Here is your answer:

In basic Object Oriented Principle, “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” relationship is very important in class designs, but, its easy to get carried away and end up in wrong design with 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. Lets take a look at an example:

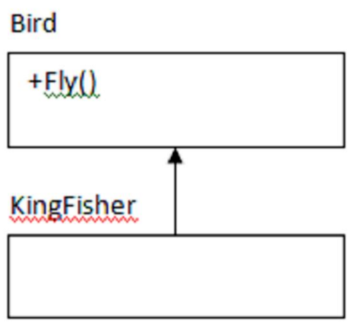

Figure : Class hierarchy showing example of Liskov Substitution principle

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:

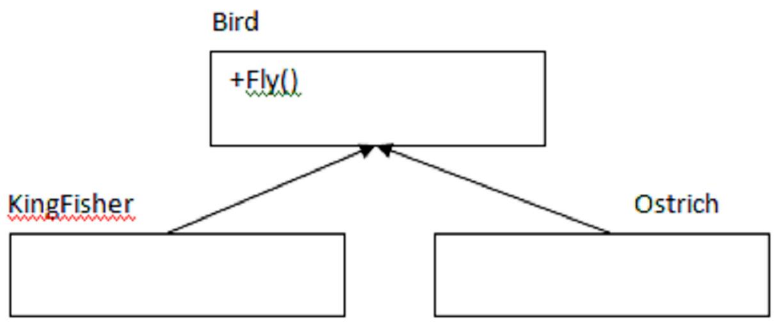

Figure : Corrected class hierarchy of Liskov Substitution principle

Ostrich is also 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 subclass instance was passed as parameter to methods method, 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 the objects and apply LSP as a verification tool to test the hierarchy whether the inheritance is properly done.

Topic : The Interface segregation principle

Shubho : Today we will learn the "Interface segregation principle". Here is the poster:

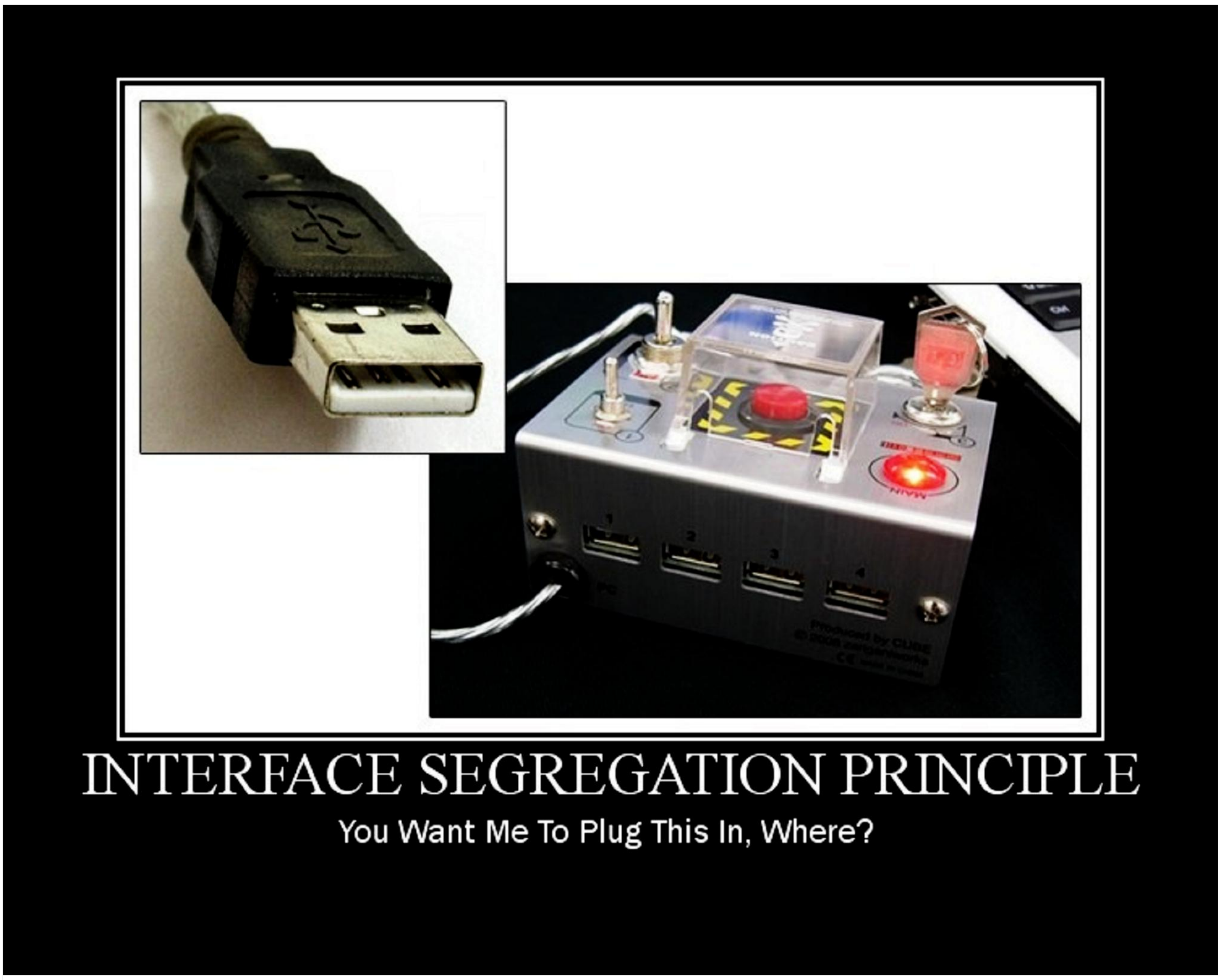
INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Figure : Interface Sugregation Principle

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 choice from. One has many switches and buttons and most of those seems confusing 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 2nd one with fewer switches and buttons.

Shubho : Yes, but, why?

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

Shubho : Correct. Similarly, suppose you have some classes and you exposed the functionality of the classes using interfaces so that, the outside world can know the available functionality of the classes and client code can be done against the interfaces. Now, if the interfaces are too big and have too many exposed methods, this 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 increase 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 introduce 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 necessary methods and not anything else?

Shubho : Exactly. Lets see an example.

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

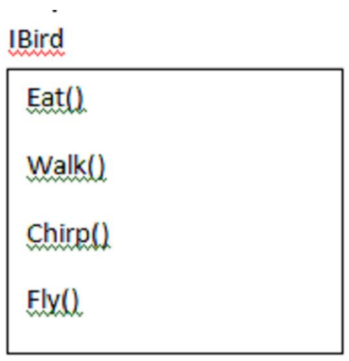

Figure : Example of an interface violating Interface Segregation principle

Note that, the IBird Interface have many bird behaviour defined along with the Fly() behaviour.

Now, if a Bird class (Say, Ostrich) implements this interface, it has to implement the Fly() behaviour unnecessarily (Because, Ostrich doesn't fly)

Farhana : Hm..correct. So, this interface must be splitted.

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

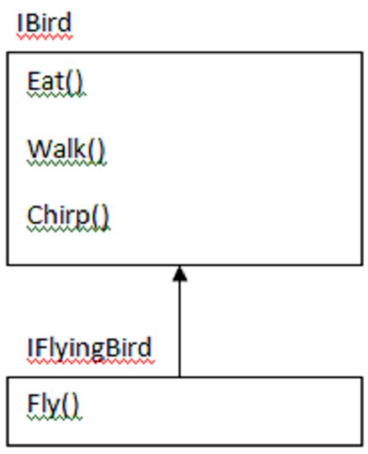

Figure : Correct version of interface Interface Segregation principle example

If there is a bird that can't fly, (Say, Ostrich), they would implement the IBird interface.

And, if there is a bird that can fly (Say, KingFisher), they 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. So, whenever they want to create a new model of television, if they need to re-use this blueprint, they would need to create as many switches and buttons as included in the plan. So, that wouldn't allow them to re-use the plan also, right?

Shubho : Right.

Farhana : And, if they really want to re-use their plans, they should divide their Television 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:

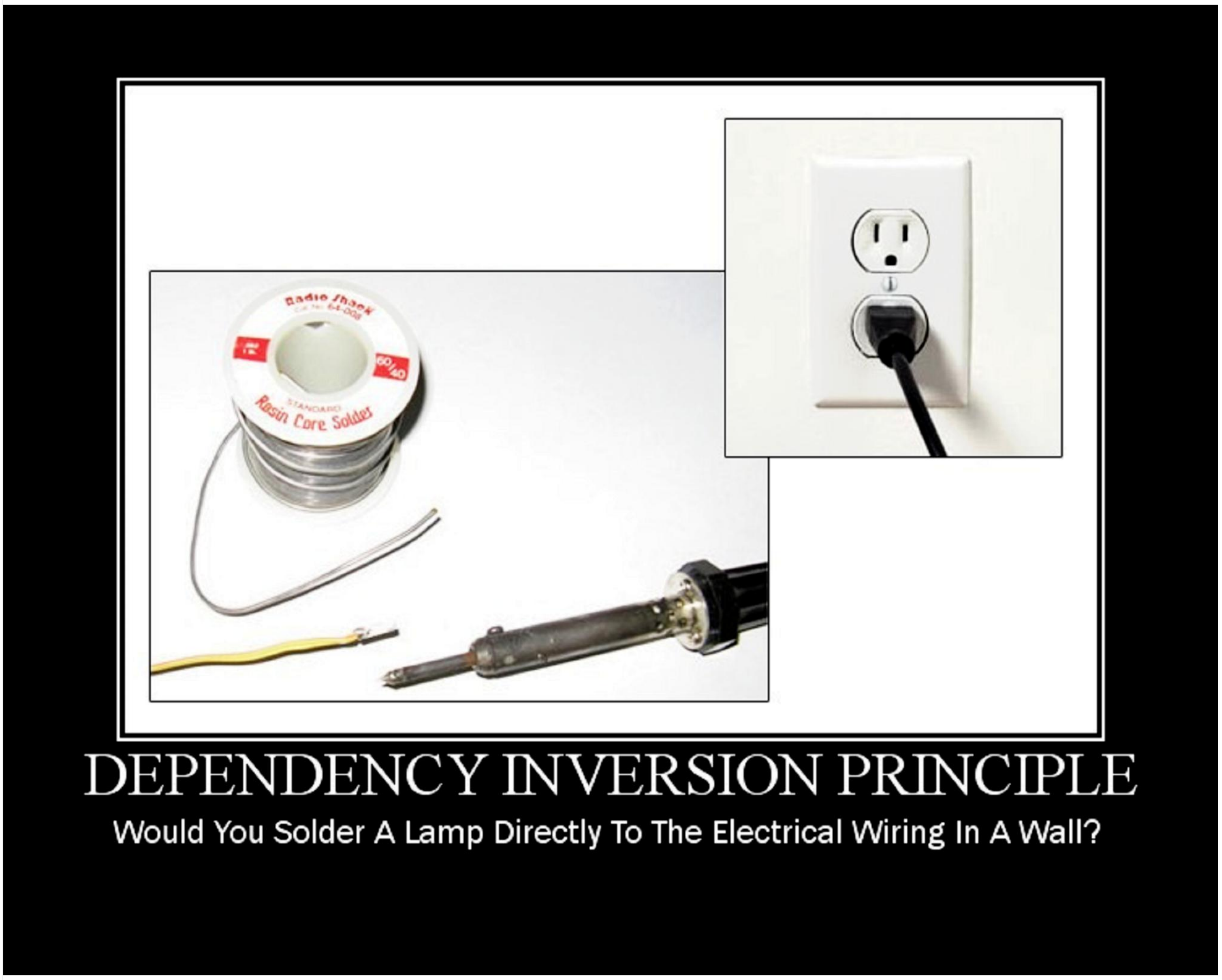
Radio Shack
Rosin Core Solder
60/40
DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Figure : Dependency Inversion princple

It says..

```
"HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. RATHER, BOTH SHOULD DEPEND
UPON ABSTRACTIONS."
```

Shubho : Lets 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 others things, right?

Farhana : Yes, ofcourse.

Shubho : OK, none of these things are rigidly built within a single thing, rather, each of these things are "Pluggable" so that, when the Engine, or the wheel has problem, you can repair it (Without repairing other things) and you can replace it.

While replacement, you just have to ensure that, the Engine/Wheel conforms to the car's design (Say, the card 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.

Shubho : Now, what if the different parts of your car were not built in such "Pluggable nature"?

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

Shubho : Yes. now, how the "Pluggable nature" is to be achieved?

Farhana : "Abstraction" is the key, right?

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

But, 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 Engine or Wheel conforms to the abstraction, these could be assembled with the car and the card would run.

Lets take a look at the following class diagram:

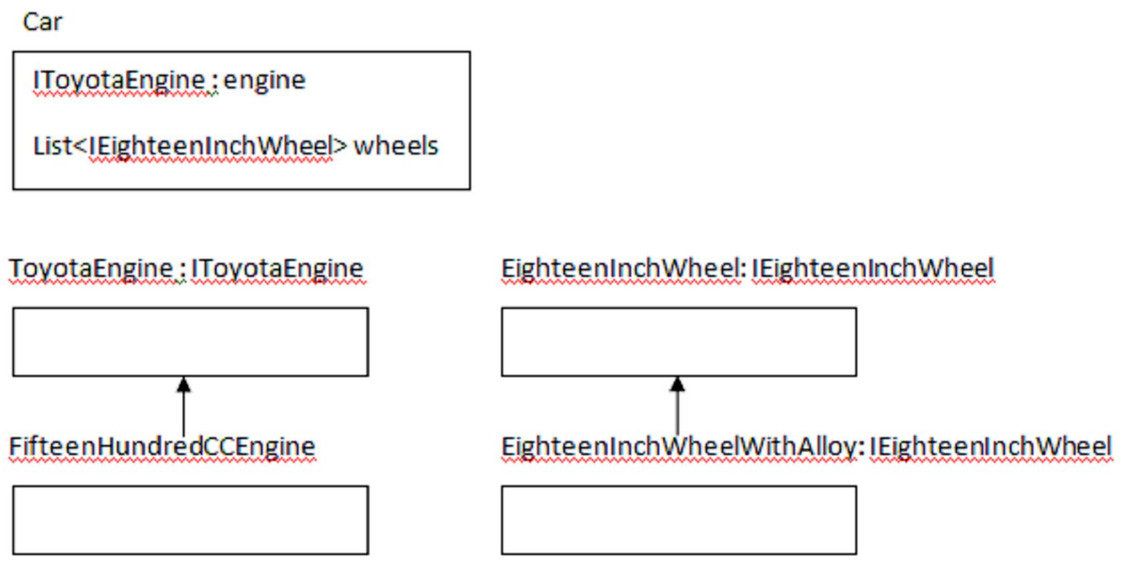
Car
IToyotaEngine : engine
List<IEighteenInchWheel> wheels
ToyotaEngine : IToyotaEngine
EighteenInchWheel: IEighteenInchWheel
FifteenHundredCCEngine
EighteenInchWheelWithAlloy: IEighteenInchWheel

Figure : 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.

So, 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 : Hmm..so if Dependency inversion is not implemented in the code, we run the risk of:

- Damaging the Higher level codes that uses the lower level classes.
- Requiring too much time and effort to change and higher level codes when a change is occurred in the lower level classes.
- Producing less-reusable codes.

Shubho : You got it perfect darling!

Summary

Shubho : There are many other Object Oriented Principles, other than the SOLID principle. Some are:

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

Farhana : Shouldn't I learn those principles also?

Shubho : Yes, of course. You have the whole world wide web to learn from. Just google on those principles and try to understand. And of course, don't hesitate me to ask me if you need.

Farhana : I've heard there are many design patterns built upon those design principles.

Shubho : You are right. The design patterns are nothing but some commonly suggested design suggestions in some common recurring situations. These are mainly inspired by the Object Oriented Design principles. You can think the design patterns as the “Frameworks” and the OOD principles can as the specifications “Specifications”.

Farhana : So, am I going to learn Design Pattern next?

Shubho : Yes darling.

Farhana : That would be exiting, no?

Shubho : Yes, that would be really exiting.

License

About the Author

M.M.Al-Farooque Shubho

Software Developer

Bangladesh

Member

A passionate software developer who loves to think, learn and observe the world around. Working in the .NET based software application development for quite a few years.

Add me on LinkedIn to know more about me.

Have a look at My other CodeProject articles.
Learn about my visions, thoughts and findings at My Blog.
Follow me at twitter @rainynovember12

Awards:

CodeProject MVP 2010
Prize winner in Competition "Best Asp.net article of May 2009"
Prize winner in Competition "Best overall article of May 2009"
Prize winner in Competition "Best overall article of April 2009"

Comments and Discussions

8 messages have been posted for this article Visit **http://www.codeproject.com/KB/tips/SOLIDPrinciplesInOOD.aspx** to post and view comments on this article, or click **here** to get a print view with messages.