* S – Single-responsibility class principle
  + A class should have one and only one reason to change, meaning that a class should have only one job.
  + Classes that are hard to unit test are often breaking SRP.
* – Open-closed principle,
  + Objects or entities should be open for extension, but closed for modification.

|  |  |
| --- | --- |
|  | * + You should be able to extend a classes behavior, without modifying it. |

* L – Liskov substitution principle
  + Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.
  + Derived classes must be substitutable for their base classes.
* I – Interface segregation principle
  + 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.
  + Make fine grained interfaces that are client specific.
* D – Dependency Inversion Principle
  + Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

Sample:

<http://blog.gauffin.org/2012/05/solid-principles-with-real-world-examples/>

<https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>

https://zeroturnaround.com/rebellabs/object-oriented-design-principles-and-the-5-ways-of-creating-solid-applications/

# Single Responsibility Principle

Single responsibility states that every class should only have one reason to change. A typical example is an user management class. When you for instance create a new user you’ll most likely send an welcome email. That’s two reasons to change: To do something with the account management and to change the emailing procedure. A better way would be to generate some kind of event from the account management class which is subscribed by a UserEmailService that does the actual email handling.



The most effective way to break applications it to create [GOD](http://en.wikipedia.org/wiki/God_object) classes. That is classes that keeps track of a lot of information and have several responsibilities. One code change will most likely affect other parts of the class and therefore indirectly all other classes that uses it. That in turn leads to an even bigger maintenance mess since no one dares to do any changes other than adding new functionality to it.

Making sure that a class has a single responsibility makes it per default also easier to see what it does and how you can extend/improve it.

Classes that are hard to unit test are often breaking SRP.

**Summary**

Try to follow the following rules to get a hang of SRP:

1. Document your code. Using AND’s should make you check your code again.
2. You should be able to associate all method names with the class/interface name. (A method called ValidateEmail cannot be associated with a class called UserService)
3. A method should only contain logic that can be associated with the method name.

Good read:

<http://blog.gauffin.org/2011/07/single-responsibility-prinicple/>

# Open/Closed principle

Open/Closed principle says that a class should be open for extension but closed for modification. Which means that you can add new features through inheritance but should not change the existing classes (other than bug fixes).

The reason is that if you modify a class, you’ll likely break the API/Contract of the class which means that the classes that depend on it might fail when you do so. If you instead inherit the class to add new features, the base contract is untouched and it’s unlikely that dependent classes will fail.

Refactor the code if needed to achieve this.

# Liskovs Substitution Principle

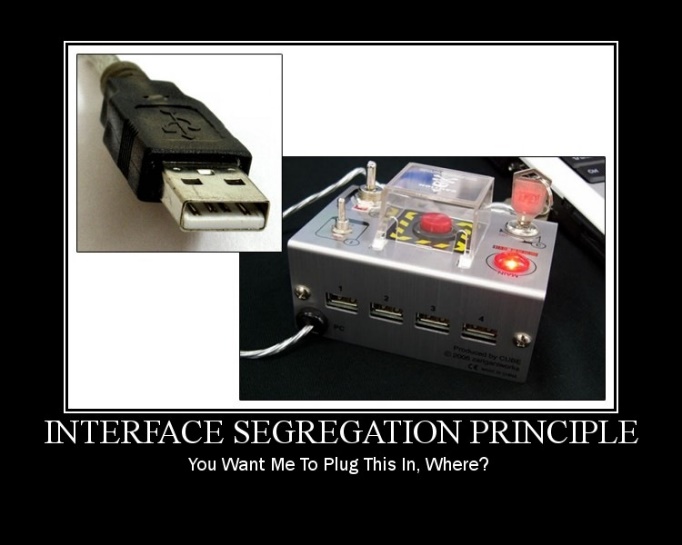
Liskovs Substitution Principle states that any method that takes class X as a parameter must be able to work with any subclasses of X.

The principle makes sure that every class follows the contract defined by its parent class. If the class Car has a method called Break it’s vital that all subclasses breaks when the Break method is invoked. Imagine the suprise if Break() in a Ferrari only works if the switch ChickenMode is activated.



# Interface Segregation Principle

ISP states that interfaces that have become “fat” (like god classes) should be split into several interfaces. Large interfaces makes it harder to extend smaller parts of the system.



There is nothing that says that there should be a one-to-one mapping between classes and interfaces. It’s in fact much better if you can create several smaller interfaces instead (depends on the class though).

# Dependency Inversion Principle

**Dependency Injection** is when you inject dependencies into the constructors instead of creating them in the class.

**Inversion Of Control** is that the container controls your objects and their lifetime.

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON **ABSTRACTIONS**.  
B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS

The original principle targets modules while I also like to apply it at class level too. It makes the principle easier to apply (so the text below is how I apply the principle).

Depend on abstractions simply means that you **should depend on as generic class/interface as possible**. For instance IUserRepository instead of DbUserRepository or HttpRequestBase instead of HttpRequest. The purpose is that your code should be as flexible as possible. The more abstract the dependencies are, the easier it is to refactor them or create new implementations.

Depending on abstractions also make your code less likely to change if the dependency change.

let the caller create the dependencies instead of letting the class itself create the dependencies. Hence inverting the dependency control (from letting the class control them to letting the caller control them).

Before

public class Volvo

{

B20 \_engine;

public Volvo()

{

\_engine = new B20();

}

}

After

public class Volvo

{

IEngine \_engine;

public Volvo(IEngine engine)

{

if (engine == null) throw new ArgumentNullException("engine");

\_engine = engine;

}

}

Which makes it a lot more fun since we now can do the following:

var myVolvo = new Volvo(new BigBadV12());