Computer Science Object Oriented Programming Principles for Self-Taught Programmers.

Writing GOOD code is not only about getting an application/functionality to do what it’s supposed to. It also has to be simple, clear and maintainable. During the development process, any code written will most probably have errors, and will be read by other developers. How that code is written will have a huge impact on how easy it will be to correct errors and make other adaptions or changes to it. That impact will be felt as the amount of time required to do the correction/change.

So these principles are tools to help make the code more developer friendly.

# DRY

DRY stands for “**Don’t Repeat Yourself**”. As developers many times we do not see how many duplicates of code we write. It is very easy to copy and paste and normally you would think that “if it’s working over there it should probably work over here with some minor changes”. But when you repeat a code, you multiply the number of potential errors and change requirements by the number of repetitions. Even worse, it leads quickly to spaghetti code. Consider the code below…

public string listByInput(string input, int[] values) {

var result = "";

switch (input) {

case "sample1":

foreach (var value in values) {

result += "," + value.ToString();

}

break;

case "sample2":

foreach (var value in values) {

result += "|" + value.ToString();

}

break;

case "sample3":

foreach (var value in values) {

result += ";" + value.ToString();

}

break;

case "sample4":

foreach (var value in values) {

result += "." + value.ToString();

}

break;

}

return result;

}

Imagine that the client reports that the result list is “,1,2,3,4” and the correct would be “1,2,3,4”. Ok it is a quick fix but how many other places does that correction need to be applied to? The client just detected the bug on the path that the application passed at that moment. What about the other possible paths in that part of the code. That sample is a very short sample. Imagine it is in a file with hundreds and maybe thousands of lines, or even a code repeated in different files throughout a project.

Now consider that other code:

private string joinValues(string separator, int[] values) {

var result = "";

foreach (var value in values) {

result += separator + value.ToString();

}

return result;

}

public string listByInput(string input, int[] values) {

switch (input) {

case "sample1":

return joinValues(",", values);

case "sample2":

return joinValues("|", values);

case "sample3":

return joinValues(";", values);

case "sample4":

return joinValues(".", values);

}

}

Both codes do exactly the same thing. Both codes would have the same problem reported by the client. Which one would require less points of attention to apply any correction?

Code repetition always leads to multiple points of maintenance, making corrections or changes more difficult, and more prone to be incomplete.

Avoiding repetition also improves the readability of the code. A small simple function is easier to read and understand than a huge complex one.

Consider these steps to reduce the complexity and code repetition:

1. Break huge functions as a sequence of higher level steps;
2. Write every step as a separate function;
3. Repeat the process with the new functions until there is a set of meaningful atomic functions
4. Reduce the repetition by eliminating identical functions and merging very similar functions into a single one.
5. If it is possible and makes sense, make the atomic functions available to be reused by other parts of the application.

# SOLID

**SOLID** is an acronym of acronyms. Shown in the table below:

|  |  |  |
| --- | --- | --- |
| **S** | SRP | Single Responsibility Principle |
| **O** | OCP | Open/Close Principle |
| **L** | LSP | Liskov Substitution Principle |
| **I** | ISP | Interface Segregation Principle |
| **D** | DIP | Dependency Inversion Principle |

These principles are themes that fill many books on coding. They represent studies of programing’s best practices in the OO (object oriented) world. It is a recommended an excellent framework to improve coding.

## The Single Responsibility Principle

“Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.”  
That principle was introduced by Robert C. Martin in an article by the same name as part of his Principles of Object Oriented Design. It is based on the principles of “cohesion” and “separation of concerns”.

**It basically says to put things that are related together and keep things that are not related to them elsewhere.**

This is very important because every point of the application is a potential source of errors. The chances that a change to the code will need to be made at some point are high. Also, chances are that a correction or change will affect some of the functionalities that are more closely related to it. These changes become easier to manage when they have defined boundaries and a set of functions that can be reused in other parts of the software or even in other future developments.

Imagine a process to retrieve a file

1. open it
2. read only the lines that have values
3. sum the values
4. display the result
5. add a line with the total to the file
6. saves the file

All of this can be done in a single method, in a single class. But if examined closely there are **3 basic operations**:

1. *Open*, *read* and *save* the file (Date Read and Persistence)
2. *Sum* the values (Business logic)
3. *Display* the value (Interface)

If those groups of functions are each in a separate class with atomic methods the code will be more maintainable with greater efficiency. For instance, the file manipulation code can be reused in other parts of the code. Or if the business logic changes (adding a percentage to the total for example), it will be easier to locate the point of change. Also, when dealing with bugs or corrections, it is easier to identify what is causing the problem.

In the sample above, imagine that the value is correct but is not being displayed in the correct format. The display class could easily be targeted to make the correction. And there is no reason to touch the other two classes that are working fine.

This is just a very simple example. Algorithms normally are far more complex than that. But this principle helps make complex application code easier to see read, identify reuse blocks, focus points of attention, and transform code into services or external libraries etc.

## Open/Close Principle

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

Bertrand Meyer is generally credited as having originated the term which appeared in his 1988 book Object Oriented Software Construction. Later in 1990 the concept were redefined to refer to the use of abstract interfaces.

Meyer said that once completed, the implementation of a class could only be modified to correct errors and that new or changed features would require that a different all together.

The redefined concept stated that those classes (the original and the new ones) should derive from an abstract interface that defines the functionality to allow it to polymorphically substituted for each other. In other words, once the implementation of a class is done, changing the behavior of a method in the class would often leads to breaking in the code and usually it is not easy to determine every possible break point.

Protect the implementation of that functionality in that class helps to avoid problems. To change a behavior, another class that implements differently the same method and to guarantee that the method signature is the same an abstract interface is used.

An abstract interface is like a contract that says what are the formats of the input and the output of a method but has no implementation.

Consider the sample above. Examine the calculation class as below:

public class Sum {

public double Calculate(IEnumerable values) {

return values.Aggregate(0.0, (s, v) => s + v);

}

}

That class is correct. If the code is changed, all the code were this is called will have to be examined for possible errors. But if the code is written like this:

public interface IBusinessLogic {

double Calculate(IEnumerable values);

}

public class Sum : IBusinessLogic {

public double Calculate(IEnumerable values) {

return values.Aggregate(0.0, (s, v) => s + v);

}

}

Now a new class could be written to change the behavior of the method, such as this new class:

public class SumAndPercent : IBusinessLogic {

public double Calculate(IEnumerable values) {

return values.Aggregate(0.0, (s, v) => s + v) \* 1.1;

}

}

The interface guarantees that the code is interchangeable between the two classes now only the reference to an instance new class where the change is required will need changing. Again this is very simplistic used to illustrate the concept. Obviously the process is more complicated than this but even with more complicated scenarios that principle reduces the potential errors generated by a change.

## The Liskov Substitution Principle

*“In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.).”*

This principle was initially introduced by **Barbara Liskov** in a 1987 conference keynote address entitled “*Data Abstraction and Hierarchy*”. It is a semantic principle rather than merely syntactic relation because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular.

**It says that if an object X of type T has a property (or method) P that behaves in a defined way, the object Y of type S, where S is a subtype of T, should have the same property (or method) P behaving the same way.**

This principle is well aligned with the open/closed principle because it clearly states that a subtype cannot change the behavior of the original type (closed for changes) but subtypes can be used to extend the original type with new methods or properties (open for extensions).

Liskov’s principle imposes some standard requirements on signatures that have been adopted in newer object-oriented programming languages, like **Contravariance**, **Covariance** and **Invariance** (those concepts are out of the scope of this paper but is recommend reading). Liskov’s principle also is an excellent tool for the design of class hierarchies in a system.

## The Interface Segregation Principle

“Many purpose specific interfaces are better than a general purpose one.”

This principle was first used and formulated by **Robert C. Martin** when doing some consulting for Xerox. The fact is mentioned in 2002 Robert’s book *Agile Software Development: Principles, Patterns and Practices*. Here is a short version of the history:

Xerox had created a new printer system that could perform a variety of tasks like stapling a set of printed papers and faxing. The software for this system was created from the ground up and performed its tasks successfully. As the software grew, making modification became more and more difficult so that even the smallest change would take hours to be applied.

The design problem was that one main Job class was used by almost all of the tasks. Anytime a print job or a stapling job had to be done, a call was made to some method in the Job class. This resulted in a huge or 'fat' class with multitudes of methods specific to a variety of different clients. Because of this design, a staple job would know about all the methods of the print job, even though there was no use for them.

The solution suggested by Martin is what is called the **Interface Segregation Principle** today. Applied to the Xerox software, a layer of interfaces between the Job class and all of its clients was added. Instead of having one large Job class, a Staple Job interface or a Print Job interface was created having only the methods relevant to that particular class. Therefore, one interface was created for each job, which were all implemented by the Job class.

That principle is well aligned with the **Single Responsibility Principle** and helps on the implementation of the **Dependency Inversion Principle** mentioned later.

A class can implement multiple interfaces so that capacity should be utilized to use the interfaces to help shape the behavior by adding small interfaces with specific purposes.

A good sample is the **IDisposable** interface. It is an interface that defines classes that have to dispose of some resources and indicates the system that it can use the **Dispose** method to do so. It is very simple, has only one method and can be added to any class that has to have that behavior.

But adding that method on a higher level interface would require that classes that does not have to dispose of resources to also implement that method unnecessarily. That would obviously lead to new potential bugs and more points of attention in the code.

So keep the interfaces lean and with a specific behavior in mind.

## The Dependency Inversion Principle

The principle states:

*A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*

*B. Abstractions should not depend upon details. Details should depend upon abstractions.*

The dependency inversion principle was also postulated by Robert C. Martin and described in several publications including the paper *Object Oriented Design Quality Metrics: an analysis of dependencies*, an article appearing in the C++ Report in May 1996 entitled *The Dependency Inversion Principle*, and the books *Agile Software Development, Principles, Patterns, and Practices*, and *Agile Principles, Patterns, and Practices in C#*.

This principle refers to a specific form of decoupling where conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are inverted (i.e. reversed) for the purpose of rendering high-level modules independent of the low-level module implementation details.

In conventional application architecture there are low level classes (LLCs) which implement basic and primary operations and high level classes (HLCs) which encapsulate complex logic and rely on the LLCs. A natural way of implementing such structures would be to write LLCs and once we have them to write the complex HLCs that directly depend on the behavior of those classes.

It makes it extremely hard to change or replace the LLCs since with the growth of the complexity of the system it becomes more and more dependent of them.

In order to avoid such problems, an abstraction layer (an interface) can be implemented between the HLCs and LLCs. Therefore the HLCs use the interfaces to define its behavior and the LLCs are now responsible to implement that interface but are no longer called directly from the HLCs.

That principle is even more powerful if combined with other ones:

Using it with the Liskov’s principle can guarantee that replacement of the LLCs by any other of their subclasses can be accopmplished without breaking the behavior of the system.

Also, using it with the Interface Segregation Principle can give the flexibility to break the implementation of the interface (and therefore the behavior of the HLC) into smaller replaceable classes.

## More Flexible and Less Dependencies

The DRY and SOLID principles are tools and practices designed to make your coding more flexible and with less dependencies. That leads to a code less prone to errors, with fewer points of potential bugs and easier to maintain.

They are advanced concepts and their use will depend on the experience in coding, and the knowledge of the business requirements a developer is trying to solve. These are concepts taught in computer science course work that many “self-taught” coders may not have had exposure too, by studying this principles code will be higher quality and much easier for a community of developers to maintain.