- **Software engineering principles and best practices you should follow as a good developer**

Modern SE Priniciples

SOLID principles are five key **object-oriented design** concepts relevant to software development.
They were developed by Robert C. Martin, but the acronym was introduced later by Michael Feathers.

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

This means a class should have only one responsibility. You may ask what can we do when there are several responsibilities. The answer is we can use separate classes for each responsibility.

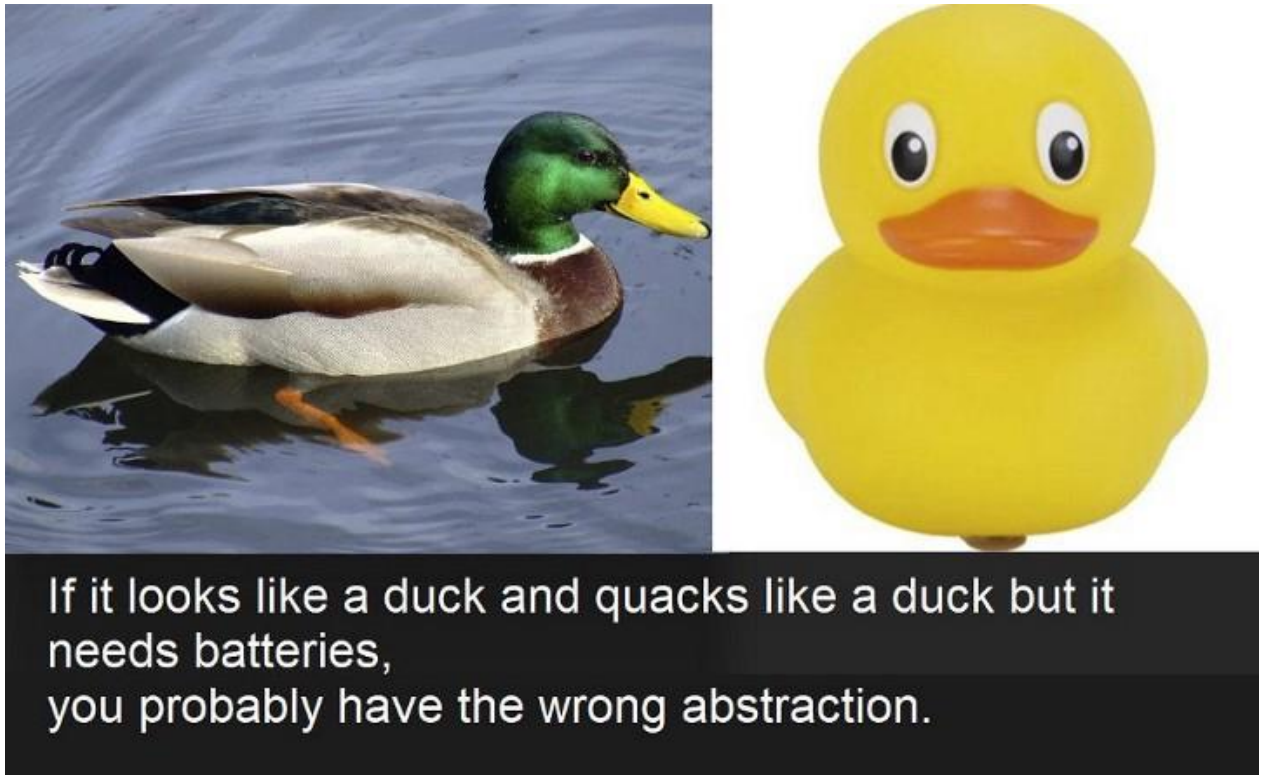
"Objects or entities should be open for extension, but closed for modification"

We need to modify the existing classes when something needs to be added.

But we cannot change the existing classes as we want, because it can lead to problems.

So, objects or entities should be able to add additional features without changing the existing classes.

For that, we can use abstraction.

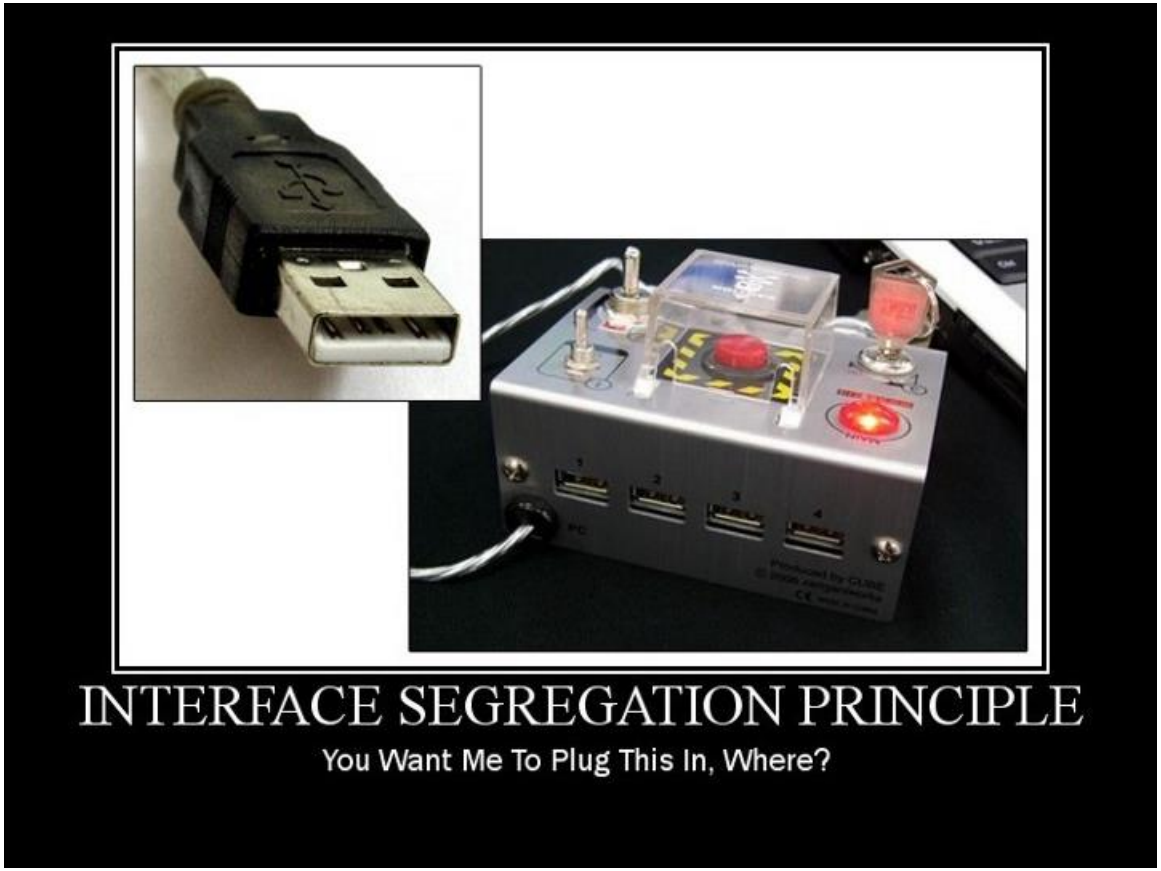


Liskov substitution:

"Every subclass/derived class should be able to substitute their parent/base class"

This means a child class should not give any different meaning to the methods existing in the base class after overriding them.

So we have to ensure that derived classes extend the base class without changing the behavior.

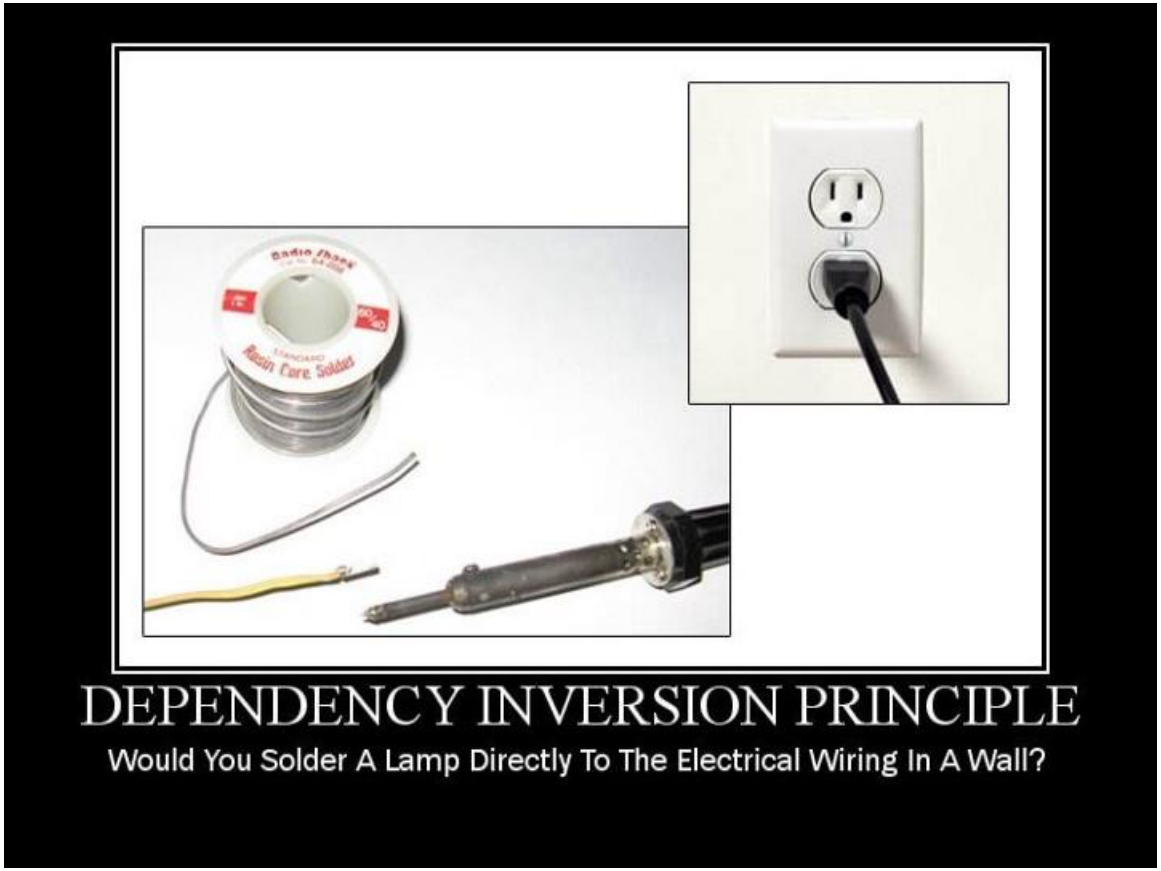


"Clients should not be forced to implement methods they do not use"

This means that no code should be forced to depend on methods it does not use.

To achieve that, we can have several small interfaces as needed, rather than using one large interface.

So, when we need it, we can let our class implement several interfaces according to the need.



"Higher-level modules should not depend on lower-level modules, but they should depend on abstractions"

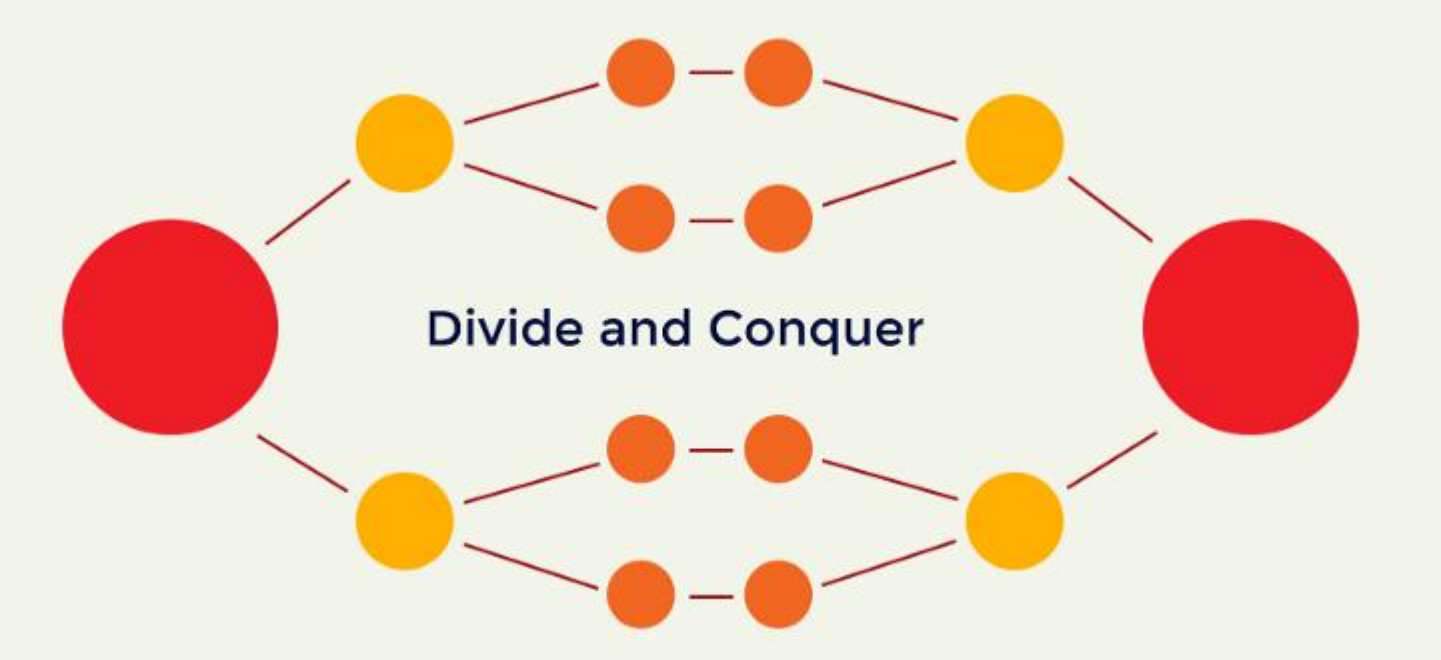
This is about the coupling between different classes or modules. That means developers have to depend on abstract classes and interfaces instead of concrete classes. The abstractions should not depend on details. Details should depend on abstractions.

Think throughout the problem:

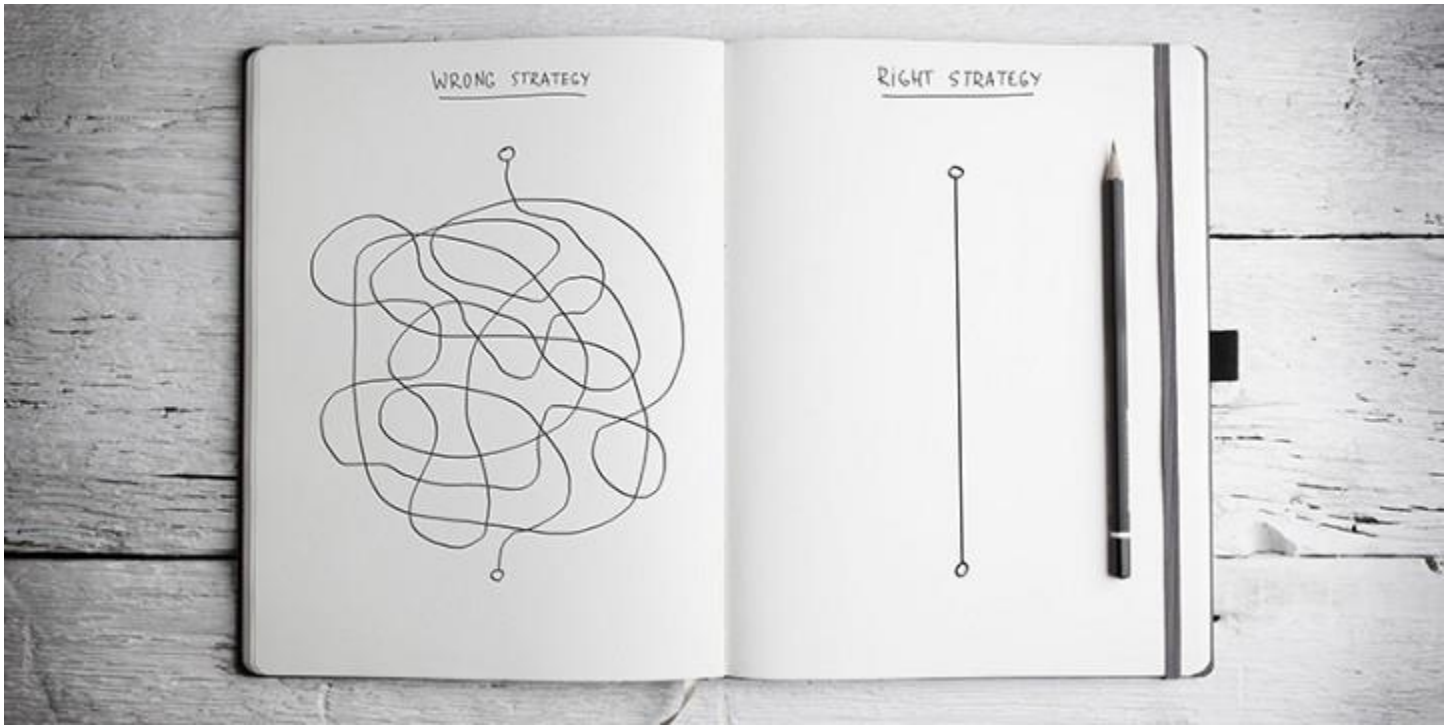
Understanding the problem clearly and completely is important.

It is better to ask questions when you cannot understand something, rather than keep quiet.

This is a key point to find the most suitable solution.



Rather than solving a large problem, we have to understand the sub-problems. Approaching the solution by solving sub-problems makes the work more manageable, efficient, and understandable.



KISS (Keep It Simple and Stupid):

Sometimes we overthink the problem and come up with a complex solution.

But we should avoid over-thinking as well as over-engineering. It will only make it unclear and difficult to manage.

Anticipate the changes as much as possible.

Learn from mistakes and prevent from doing the same mistake again.

When you are trying, we can make mistakes.

But, making mistakes is the best way to learn.

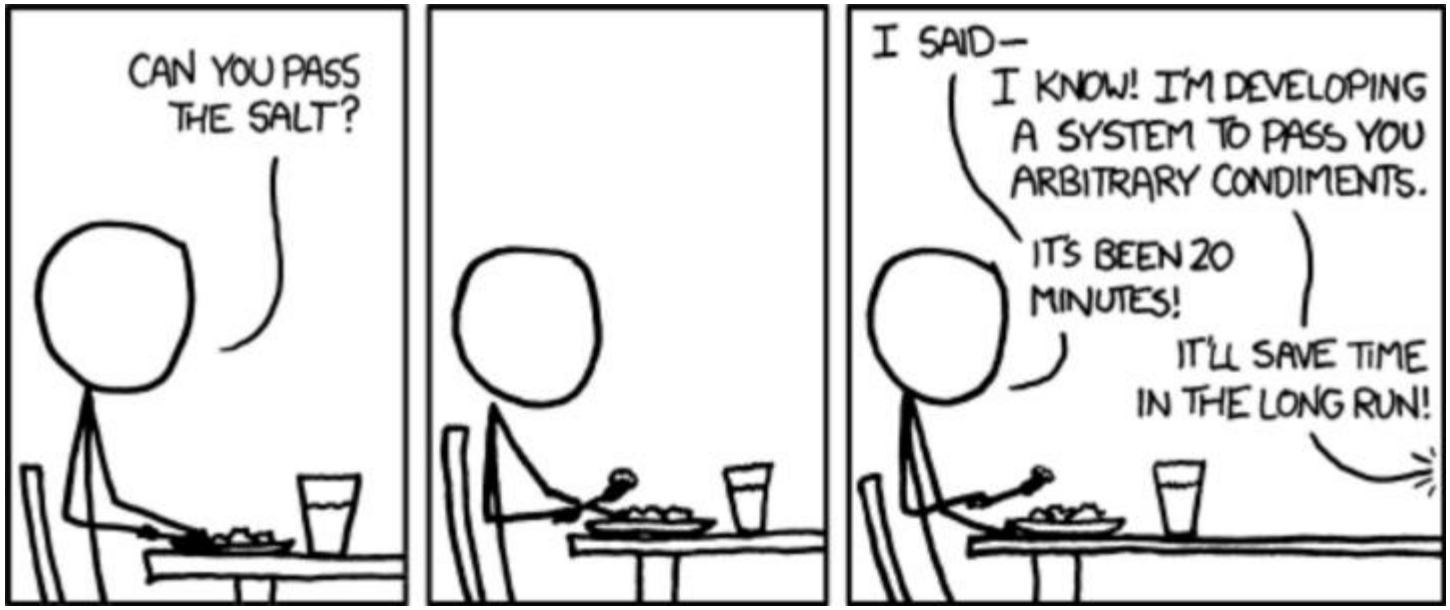
Always remember why software exists:

Always keep in mind the actual reason for the developing software to exist.

It will prevent you from following the wrong path

Remember that you are not the user:

Developers are not the real users of the product.
We have to remember it always. The end-users may not be capable as the developer.
So the user-friendliness and user experience are important.



YAGNI (You Ain't Gonna Need It):

It is not a good practice to write complex codes when we can approach the solution by writing a simple code.

There's no need for adding additional codes just because we think that it will be useful in the future. It may be useful or not. But since we cannot predict the future it is better not to waste time. So, it is better to write codes that are needed for the moment, not for the future.

DRY (Don’t Repeat Yourself):

Reuse codes when it is possible rather than repeating the same code again and again.

It will make your work more efficient.

Embrace Abstraction

System functions should work without knowing the implementation details of every part.

There's no need for users to know how the system works. But the system should be able to provide the service they expected without any problem.

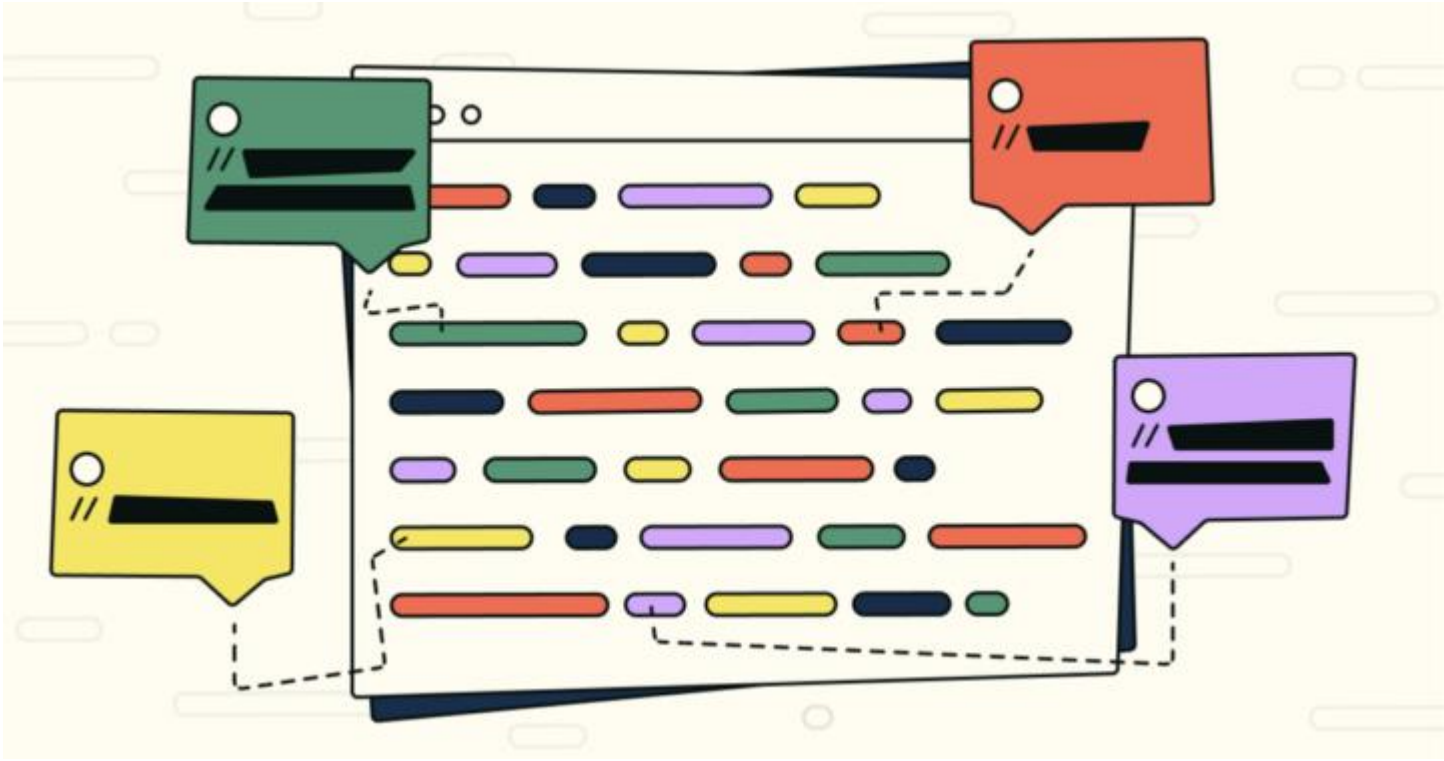
Don't Reinvent The Wheel !

Maybe the problem you are currently solving has already been solved by another person.

When there is a solution for the same problem exists, use it.

Because those are tested solutions and there's no need for doing it again. But we can make the already existing ones better.

Write code that does one thing well:



Debugging is harder than writing code:

Make the code readable as much as possible. It will help you to handle the errors easily.

It should be understandable for others. It will be important for maintenance as well.

Kaizen (Leave it better than when you found it):

When we found a bug we want it to be fixed as soon as possible.

But we should not fix a bug just to make the code runnable.

We have to leave the code better by fixing the bug correctly and completely. It will help to avoid extra problems.

Avoid temporary Fixes