coding-principles

Sadržaj

Bad code comes in many forms. Messy code, massive if-else chains, programs that break with one adjustment, variables that don't make sense. The program might work once but will never hold up to any scrutiny.

If you want to be a programmer, don't settle for shortcuts. Aim to write code that is easy to maintain. Easy for you to maintain, and easy for any other developer on your team to maintain. How do you write effective code? You write good code by being disciplined with programming principles. So Today we're going see some programming principles that can help you to be a Good Developer

1. Keep It Simple, Stupid (KISS) image.png It sounds a little harsh, but it's a coding principle to live by. What does this mean?

It means you should be writing code as simple as possible. Don't get caught up in trying to be overly clever or showing off with a paragraph of advanced code. If you can write a script in one line, write it in one line.

Here's a simple function: function addNumbers(num1,num2){ return num1 + num2; } Pretty simple. It's easy to read and you know exactly what is going on.

Use clear variable names. Take advantage of coding libraries to use existing tools. Make it easy to come back after six months and get right back to work. Keeping it simple will save you the headache.

2. DRY (Don’t repeat yourself) image.png DRY (don’t repeat yourself) means don’t write duplicate code, instead use Abstraction to abstract everyday things in one place.

If you have a block of code in more than two places, consider making it a separate method, or if you use a hard-coded value more than one time, make them public final constant. The benefit of this Object-oriented design principle is in maintenance.

It’s important not to abuse it, duplication is not for code, but for functionality.

It means if you have used standard code to validate OrderId and SSN, it doesn’t mean they are the same, or they will remain the same in the future.

By using standard code for two different functionality or thing, you tightly couple them forever, and when your OrderId changes its format, your SSN validation code will break.

So beware of such coupling and don’t combine anything which uses similar code but is not related. You can further check out the Basics of Software Architecture & Design Patterns in Java course on

Udemy to learn more about writing the right code and best practices to follow while designing a system.

DRY code is easy to maintain. It's easier to debug one loop that handles 50 repetitions than 50 blocks of code that handle one repetition.

To learn more about DRY, You should read this article be thankful to @Chris Bongers

3. Open/Closed image.png This principle means you should aim to make your code open to extension but closed to modification. This is an important principle when releasing a library or framework that others will use.

For example, suppose you're maintaining a GUI framework. You could release for coders to directly modify and integrate your released code. But what happens when you release a major update four months later?

Their code will break. This will make engineers unhappy. They won't want to use your library for much longer, no matter how helpful it may be.

Instead, release code that prevents direct modification and encourages extension. This separates core behavior from modified behavior. The code is more stable and easier to maintain.

4. Composition Over Inheritance If you write code using object-oriented programming you're going to find this useful. The composition over inheritance principle states: objects with complex behaviors should contain instances of objects with individual behaviors. They should not inherit a class and add new behaviors.

Relying on inheritance causes two major issues. First, the inheritance hierarchy can get messy in a hurry. You also have less flexibility for defining special-case behaviors. Let's say you want to implement behaviors to share:

Composition programming is a lot cleaner to write, easier to maintain and allows flexibility defining behaviors. Each individual behavior is its own class. You can create complex behaviors by combining individual behaviors.

5. You Aren't Going to Need It (YAGNI) image.png This principle means you should never code for functionality on the chance that you may need in the future. Don't try and solve a problem that doesn't exist.

In an effort to write DRY code, programmers can violate this principle. Often inexperienced programmers try to write the most abstract and generic code they can. Too much abstraction causes bloated code that is impossible to maintain.

Only apply the DRY principle only when you need to. If you notice chunks of code written over and over, then abstract them. Don't think too far out at the expense of your current code batch.

There are two main reasons to practice YAGNI,

You save time because you avoid writing code that you turn out not to need. Your code is better because you avoid polluting it with 'guesses' that turn out to be more or less wrong but stick around anyway. 6. Single Responsibility image.png As you start writing code, over a long period of time, your code would become clumsy. You will have classes/modules that perform several functionalities. This will end up with classes that are hundreds and thousands of lines of code. This principle says that every class or module in a program should only have specific functionality. In other words, a class or module in a program should only be responsible for tasks regarding one particular function. This helps you keep your modules minimal and clean.

Both Open/Closed and Single Responsibility principles are in under the SOLID Principle. In this article, @Francesco Ciulla has wonderfully explained about each principle in SOLID.

7. Document Your Code Any senior developer will stress the importance of documenting your code with proper comments. All languages offer them and you should make it a habit to write them. Leave comments to explain objects, enhance variable definitions, and make functions easier to understand.

Here's a Python function with comments guiding you through the code: def find_odd(): number = int(input(Enter a number..)) # Get a number if number % 2 == 0: # Divide number by 2. If no remains, print("This is a even number") # Print it is not a odd number else : print("Here is the odd number!") # If 1 remains, print it is a odd number

Leaving comments is a little more work while you're coding, and you understand your code pretty well right?

Leave comments anyway!

Try writing a program, leaving it alone for six months, and come back to modify it. You'll be glad you documented your program instead of having to pour over every function to remember how it works. Work on a coding team? Don't frustrate your fellow developers by forcing them to decipher your syntax.

8. Separation of Concerns The separation of concerns principle is an abstract version of the single responsibility principle. This idea states that a program should be designed with different containers, and these containers should not have access to each other.

A well-known example of this is the model-view-controller (MVC) design. MVC separates a program into three distinct areas: the data (model), the logic (controller), and what the page displays (view). Variations of MVC are common in today's most popular web frameworks.

For example, the code that handles the database doesn't need to know how to render the data in the browser. The rendering code takes input from the user, but the logic code handles the processing. Each piece of code is completely independent.

The result is code that is easy to debug. If you ever need to rewrite the rendering code, you can do so without worrying about how the data gets saved or the logic gets processed.

9. Refactor image.png It's hard to accept, but your code isn't going to be perfect the first time. Refactoring code means reviewing your code and looking for ways to optimize it. Make it more efficient while keeping the results exactly the same.

Codebases are constantly evolving. It's completely normal to revisit, rewrite, or even redesign entire chunks of code. It doesn't mean you didn't succeed the first time you wrote your program. You're going to get more familiar with a project over time. Use that knowledge to adjust your existing code to be DRY, or following the KISS principle.

10. Clean Code At All Costs image.png Leave your ego at the door and forget about writing clever code. The kind of code that looks more like a riddle than a solution. You're not coding to impress strangers.

Don't try to pack a ton of logic into one line. Leave clear instructions in comments and documentation. If your code is easy to read it will be easy to maintain.

Good programmers and readable code go hand-in-hand. Leave comments when necessary. Adhere to style guides, whether dictated by a language or your company. @Mario Cervera has beutifully explained about Clean Code in this article

Happy Coding!