# **15 Software Engineering Principles**

# **https://medium.com/pythoneers/16-software-engineering-principles-i-ignored-for-too-long-a69d32f1a52e**

Software engineering isn’t just about writing code — it’s about writing good, scalable, maintainable, and efficient code. When I started, I focused more on making things work rather than making them right. Over time, I learned that ignoring core software engineering principles leads to technical debt, unmanageable code, and endless debugging nightmares. In this article I will share 16 kept principle that I ignored for too long.

**“Good code is its own best documentation.”** — *Steve McConnell*

# **1. DRY (Don’t Repeat Yourself)**

I used to copy-paste chunks of code across multiple files, thinking it was a quick way to get things done. But whenever a bug showed up, I had to fix it in ten different places, which turned into a nightmare.

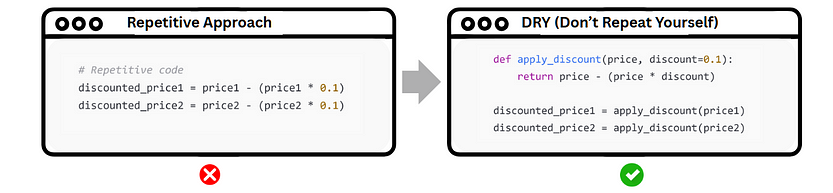
DRY is all about avoiding duplication by using functions, classes, and organizing your code better. It saves time, reduces errors, and makes your project easier to maintain.

## **How to Apply It**

Write reusable functions instead of repeating the same logic.

Use object-oriented programming techniques like inheritance or composition.

Move constants and configuration values into separate files.



# **2. KISS (Keep It Simple, Stupid)**

Code doesn’t earn extra points for being complicated. Layering it with abstractions, nesting loops like a puzzle box, or scattering classes everywhere only sets a trap for your future self.

Clear, simple code is a gift — it speaks for itself, stays flexible, and rarely bites back when things go wrong.

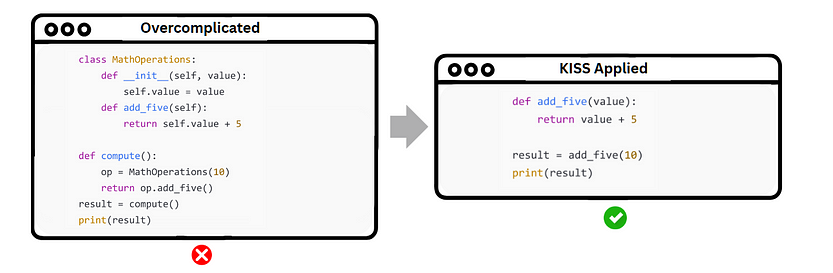
## **How to Apply It**

Don’t force complex design patterns unless you truly need them.

Write code that explains itself instead of relying on too many comments.

Always prefer a straightforward solution over a clever hack that’s hard to understand later.

Tip: Remember the Zen of Python — “Simple is better than complex.”



# **3. YAGNI (You Ain’t Gonna Need It)**

It’s tempting to build for “someday” — to sneak in features or extra code just in case. But most of that “just in case” never comes.

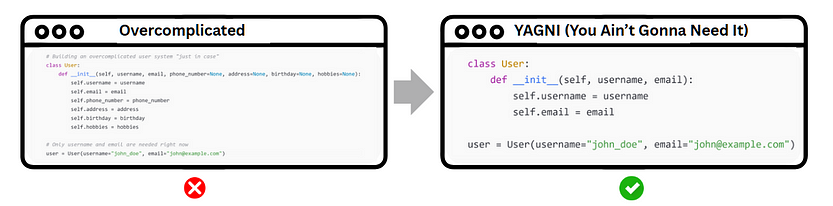
Writing code you don’t yet need adds clutter, invites bugs, and makes the whole system harder to manage. Solve today’s problems — tomorrow can wait its turn.

## **How to Apply It**

Build only what you need right now.

Don’t add extra options, classes, or features until they’re actually needed.

Premature optimization is usually a waste of time early on.



# **4. SOLID Principles (OOP Best Practices)**

In the early days, I built classes that did it all — handling logic, data, and even side quests. They looked impressive, but changing one part felt like pulling a thread from a sweater: everything unraveled.

Discovering the SOLID principles was a turning point. They showed me how to design smaller, focused pieces that play well together — code that grows without breaking its own bones.

## **How I Apply It Now**

S — Single Responsibility Principle: Every class should have only one job.

O — Open/Closed Principle: Your code should be open for extension but closed for modification.

L — Liskov Substitution Principle: Subclasses should be able to replace their parent classes without any issues.

I — Interface Segregation Principle: Avoid forcing classes to implement methods they don’t use.

D — Dependency Inversion Principle: Depend on abstractions, not concrete implementations.

## **Quick Tip**

If a class name sounds like it’s doing multiple things — like UserAndOrderManager — it's a red flag that it's violating the Single Responsibility Principle.

# **5. Composition Over Inheritance**

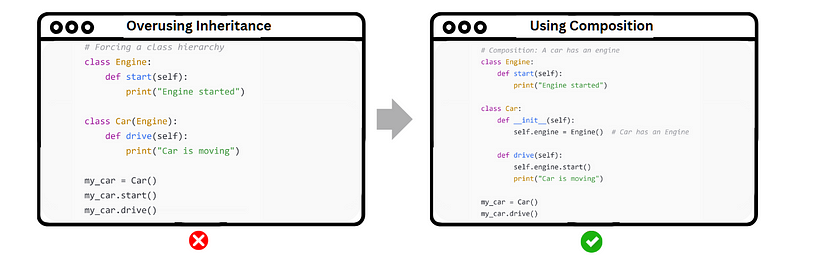
It’s easy to fall into the trap of building towering inheritance trees — everything nested, everything connected. But what starts as organized quickly turns into a mess that’s tough to update or grow.

Composition offers a smarter path: combine simple, focused pieces to get the behavior you need — without locking yourself into a rigid structure.

## **How I Apply It Now**

Use composition when different classes need to share behavior without being tightly coupled.

Reserve inheritance for cases where a clear “is-a” relationship truly exists.



# **6. The Law of Demeter (LoD)**

I often wrote code that directly accessed internal objects and their sub-objects, chaining multiple dots together. It looked fine at first, but over time, it made the code fragile and tightly coupled.

The more you reach into an object’s internals, the more your code depends on how that object is structured. If the internal structure changes, you end up breaking multiple parts of your code.

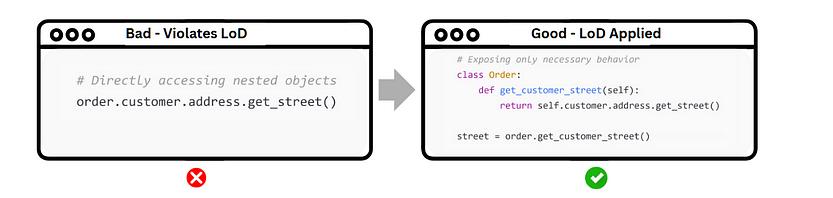
The Law of Demeter taught me to limit my reach, focusing only on the object at hand and its closest collaborators. The less you rely on the inner structure, the more stable your code stays when things evolve.

# **How to Apply It**

Avoid chaining multiple objects together.

Expose what’s needed through methods, rather than revealing internals.

Think in terms of “Tell, Don’t Ask” — tell objects what you want them to do, don’t ask them for their internals.



# **7. Test-Driven Development (TDD)**

I often wrote tests only after I finished coding. By then, the code was already complicated, and finding the root of bugs became messy and time-consuming.

Writing tests *before* coding helps you design better, simpler, and more reliable code. It also forces you to think about edge cases early and dramatically reduces regressions later.

The cycle is simple: Red → Green → Refactor

Red: Write a failing test.

Green: Write just enough code to pass the test.

Refactor: Clean up the code while keeping all tests passing.

## **How to Apply It**

Start by writing a small, focused test for the behavior you want.

Only then, write the minimal code needed to make it pass.

Continuously refactor and keep tests green.

## **Example (TDD in action)**

Step 1: Write a Failing Test (Red)

import unittest  
  
class BankAccount:  
 def \_\_init\_\_(self, balance):  
 self.balance = balance  
  
 def withdraw(self, amount):  
 pass # Not implemented yet  
  
class TestBankAccount(unittest.TestCase):  
 def test\_withdraw\_reduces\_balance(self):  
 account = BankAccount(100)  
 account.withdraw(30)  
 self.assertEqual(account.balance, 70)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

Test will fail because withdraw is not implemented yet.

Step 2: Make It Pass (Green)

def get\_even\_numbers(numbers):  
 return [2, 4, 6]

The test passes!  
But the code is hardcoded — it only works for this specific input. It’s not scalable.

Step 3: Refactor (if Needed)

def withdraw(self, amount):  
 if amount > self.balance:  
 raise ValueError("Insufficient funds")  
 self.balance -= amount

Now it’s dynamic, clean, and still passes the test!

# **8. Don’t Make Me Think**

I created interfaces that required users to think too much about how to use them, causing frustration and confusion.

An interface should be self-explanatory — users should not need to figure out how to use it. When designing, prioritize simplicity and clarity.

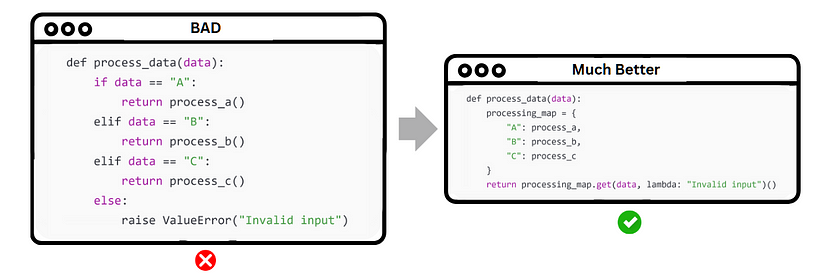
## **How to Apply It**

Keep user actions as simple and logical as possible.

Reduce cognitive load by making processes flow naturally.

Avoid unnecessary features and keep the essential ones easy to find and use.

Provide feedback to users for their actions, such as confirming a successful action or guiding them through a process.



# **9. Fail Fast, Fail Often**

In the past, I’d rush through development, assuming everything would work out and only testing the big picture. But that often meant discovering problems too late — usually after pouring in significant time or resources. By the time issues were uncovered, they were harder to fix and debug.

Now, I’ve learned the value of failing early and often. Catching bugs during development and testing, before they get too big, saves time in the long run. Embracing these small failures isn’t a setback — it’s a chance to refine and improve before things spiral out of control.

## **How to Apply It**

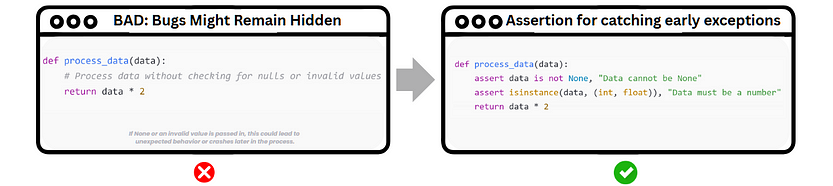
Use assertions to validate conditions during development (e.g., checking input values, conditions, and assumptions).

Implement comprehensive error handling to manage failures gracefully and provide feedback on issues as soon as they arise.

Write tests that cover edge cases, not just the happy path, to ensure that the application handles unexpected inputs correctly.

Adopt a mindset of continuous improvement: view each failure as an opportunity to learn and refine your code, rather than something to fear.

Embrace early debugging, use logs and diagnostics to track potential issues as the code is running.



# **10. SELF (Self-Descriptive, Easily-Legible Function/Variable Names)**

Clear, descriptive names help make the code self-explanatory, reducing the need for excessive comments. Good naming practices provide immediate context, improving readability and maintainability of the codebase. By following naming conventions and being specific, I can make my code more intuitive for anyone reading it, including myself in the future.

## **How to Apply It**

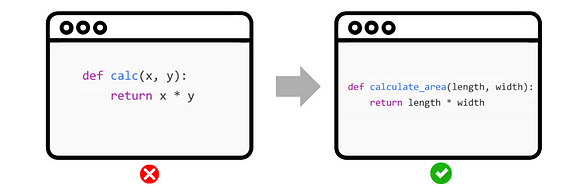
Use meaningful names that convey the purpose of the variable or function. For example, instead of data, use user\_data or product\_list if applicable.

Function names should clearly describe what the function does (e.g., calculate\_total instead of process).

Avoid generic names like temp, foo, bar, or x. If you're using numbers, make sure they have context (e.g., counter vs. counter1).

Use consistent naming conventions across your codebase (camelCase, snake\_case, etc.).

Ensure that variables and functions are named in a way that reflects their roles in the program. If it’s a counter, call it counter, not i or j.



# **11. PRECISION**

I’ve seen many developers — myself included — fall into the trap of using try/except to catch everything. It feels like a quick fix, but it often makes debugging a nightmare. Catching all exceptions can hide the real issues, leaving you with vague error messages and no clear path forward.

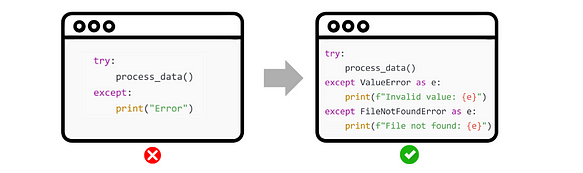
The better approach is to catch specific exceptions and handle them thoughtfully. Clear error messages not only make it easier to debug but also make your code more readable and maintainable.

## **How to Apply It**

Catch specific exceptions (e.g., ValueError, FileNotFoundError).

Avoid a blanket except: unless absolutely necessary.

Use logging to capture errors instead of printing them.



# **12. MSE (Minimized Side Effects)**

Modifying global variables inside functions can lead to unpredictable behavior and make debugging a challenge. When multiple parts of the code change the same global state, tracking down bugs becomes much harder.

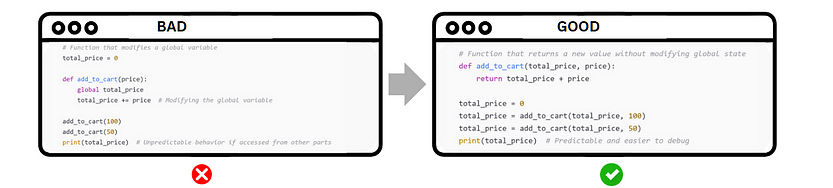
To avoid this, aim to minimize side effects. Functions should focus on returning values and avoid changing global variables or any external state. This makes the code more predictable, easier to test, and simpler to maintain.

## **How to Apply It**

Use pure functions that don’t modify global variables or other shared states.

Return values instead of modifying variables outside the function’s scope.

Avoid relying on mutable global data whenever possible.



# **13. POS (Principle of Least Surprise)**

I sometimes wrote code that threw users or other developers off guard, often by straying from common conventions or patterns. This inconsistency led to confusion and unnecessary misunderstandings.

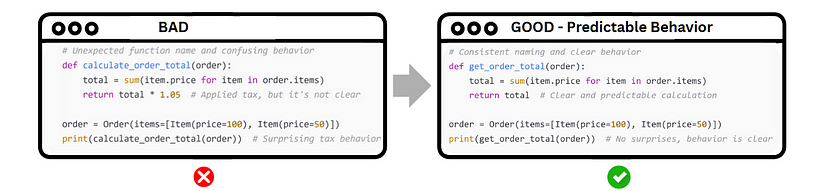
To avoid this, it’s crucial to follow predictable, intuitive patterns. Sticking to established conventions and naming standards makes your code easier for others to understand and work with, reducing friction and improving collaboration.

## **How to Apply It**

Follow conventions for naming variables, functions, and classes.

Adhere to common patterns and frameworks that developers are familiar with.

Keep behavior consistent with expectations — avoid doing things in unexpected ways.



# **14. FROG (Favor Readability Over Cleverness)**

Writing overly complex or “clever” code can make it harder for others — and yourself — to understand and maintain. Shortcuts and advanced techniques might look impressive, but they often lead to confusion when changes or debugging are needed.

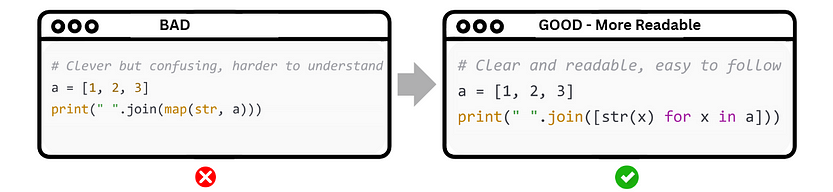
Prioritize readability over cleverness. Readable code is easier to modify, debug, and maintain, ensuring that everyone can understand it and work with it long-term.

## **How to Apply It**

Write code for humans first, computers second.

Avoid overly clever tricks that might save a few lines but make the code harder to follow.

Prefer clarity over brevity or tricks.



# **15. Separation of Concerns (SoC)**

Mixing business logic with presentation logic within the same code can complicate updates and maintenance. When business rules change, the UI code may also need modifications, creating bugs and inefficiencies.

Separating concerns leads to more modular code. By keeping business logic and UI logic separate, changes to one part can be made without affecting the other, making the code easier to maintain and scale.

## **How to Apply It**

Separate business logic from UI logic and data access.

Use MVC (Model-View-Controller) or MVVM (Model-View-ViewModel) architectures to create clear boundaries between different concerns.

Ensure each module has a single responsibility and does not overlap with others.

