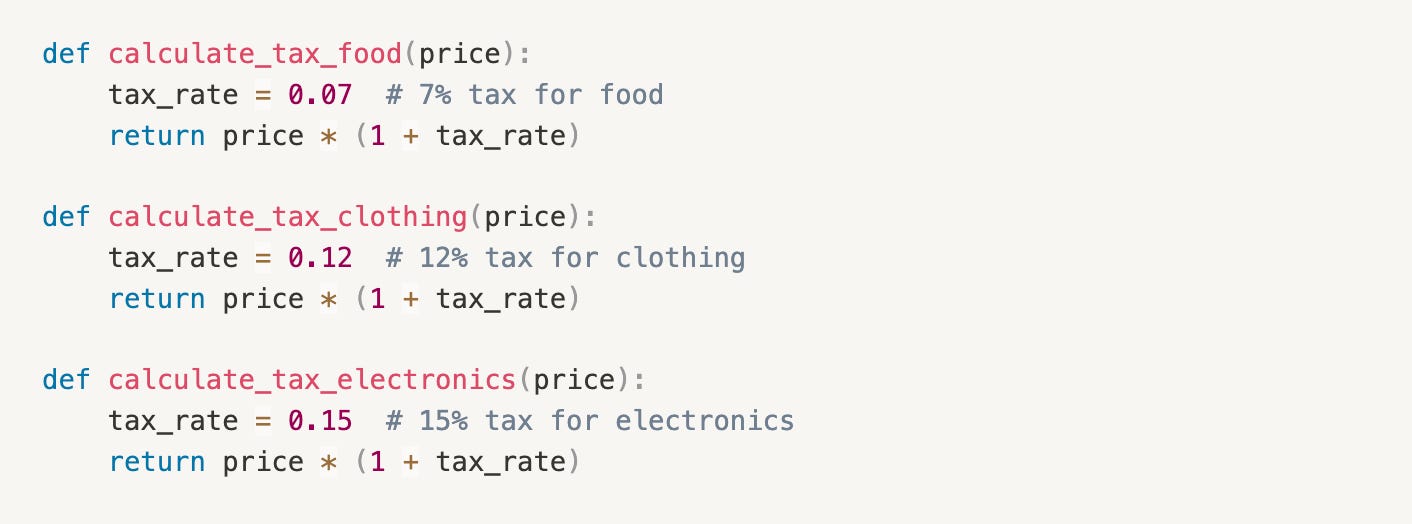
**The DRY Principle**

**What is the DRY Principle?**

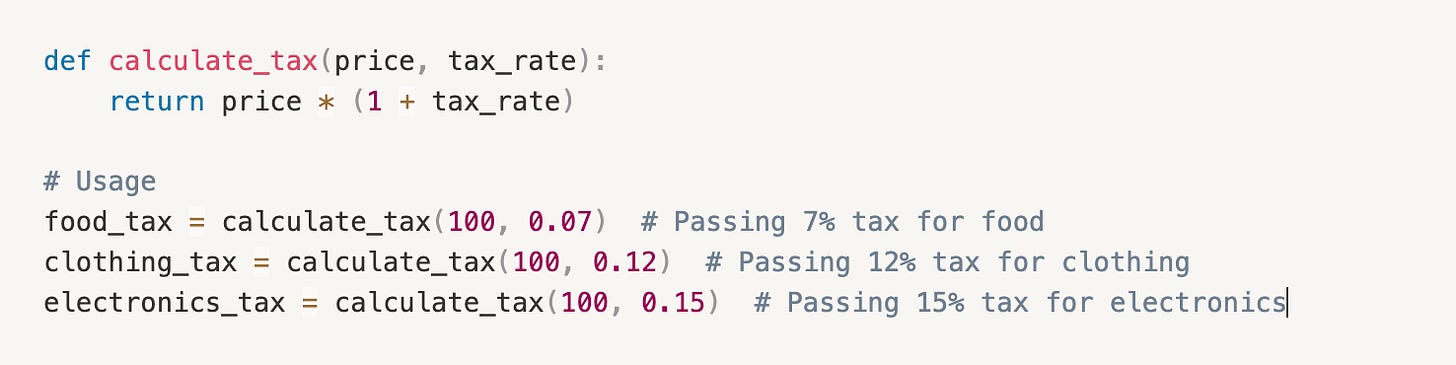
DRY principle encourages developers to write modular, reusable code and avoid duplicating the same functionality in multiple places.

**Example 1: Avoiding Code Duplication**

non-DRY approach



Let's DRY it up:

[[](https://substackcdn.com/image/fetch/$s_!MVus!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F5f1469e7-7cba-4511-83d1-65ec7b02166a_1476x370.png)](https://substackcdn.com/image/fetch/$s_!MVus!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F5f1469e7-7cba-4511-83d1-65ec7b02166a_1476x370.png" \t "_blank)

**Example 2: Using Decorators for Cross-cutting Concerns**

To log function calls in multiple functions. Instead of writing the logging code in each function, use a decorator:

**Why is DRY Important?**

* Reduced Code Duplication
* Improved Code Reusability

**How to Apply the DRY Principle**

* Identify Repetitive Code
* Extract Common Functionality
* Use Inheritance and Composition

**When not to use the DRY Principle**

1. **Premature Abstraction**: Trying to apply DRY too early in the development process might lead to over-engineering. If requirements are likely to change, you might abstract code that ends up getting discarded or significantly reworked.
2. **Performance-critical code**: In some cases, duplicating code can be faster than calling a reusable function, especially if the function has a high overhead or is not inlined.
3. **Sacrificing Readability**: If the duplicated code is very simple and easy to understand, it might be better to leave it as is, rather than creating a complex abstraction.
4. **One-time usage**: If a piece of code is only used once, it might not be worth extracting into a reusable function.
5. **Legacy code or technical debt**: In cases where you're working with legacy code or technical debt, it might be more practical to duplicate code temporarily, rather than trying to refactor the entire system.
6. **Debugging and testing**: In some cases, duplicating code can make it easier to debug and test, as it allows for more isolation and control.

**The KISS Principle**

**What is the KISS Principle?**

The KISS principle, which stands for **"Keep it Simple, Stupid"**

It suggests that software should be designed to be easy to understand, modify, and extend, rather than complex and convoluted.

Eg: Using built-in math.factorial function rather than using recursive code

**Why is the KISS Principle Important?**

Improved Readability , Reduced Complexity , Faster Development , Enhanced Reliability , Increased Flexibility

**The YAGNI Principle**

**What is YAGNI?**

**You Aren't Gonna Need It** : Always implement things when you actually need them, never when you just foresee that you might need them.

Avoiding over-engineering and focusing strictly on the present requirements.

**Over-engineered:**

[[](https://substackcdn.com/image/fetch/$s_!Q7Gk!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F1026e117-5d09-46ab-afbb-67d046f98985_1302x370.png)](https://substackcdn.com/image/fetch/$s_!Q7Gk!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F1026e117-5d09-46ab-afbb-67d046f98985_1302x370.png" \t "_blank)

**YAGNI-aligned:**

[[](https://substackcdn.com/image/fetch/$s_!T_QQ!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fb2a83e62-b99c-4760-b929-567ca0c20544_1110x204.png)](https://substackcdn.com/image/fetch/$s_!T_QQ!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fb2a83e62-b99c-4760-b929-567ca0c20544_1110x204.png" \t "_blank)

Start by supporting only the payment methods you *currently* need. Add support for PayPal or Bitcoin later in the development cycle if the demand arises.

**Benefits of YAGNI**

1. **Reduced waste**
2. **Simplified codebase**:

**When YAGNI Might Be Inappropriate**

Like any principle, YAGNI shouldn't be rigidly applied in every situation. There are times when anticipating near-future needs makes sense:

* **Well-Known Requirements:** If you know with high certainty a feature is coming soon, building some basic support upfront might be wise.
* **Performance-Critical Areas:** Sometimes, a less-than-optimal but more general solution is necessary initially to ensure performance targets are met.

**Law of Demeter**

**The Problem**

Imagine you're building a simple e-commerce system.

You have:

* A Customer who owns a ShoppingCart
* The cart contains a list of CartItems
* Each CartItem refers to a Product
* And every Product has a Price

To display the price of the **first product** in a customer’s shopping cart.

A ***common*** (but flawed) way to write this would be:

Money price = customer.get ShoppingCart().get Items().get (0).get Product().get Price();

This approach works, but it **smells bad**.  
It’s what we call a **“train wreck”** or **“dot-chaining”**: one object reaching through several others to get what it wants.

**What’s Wrong With This?**

**1. High Coupling**

The OrderService method is now tightly coupled to the **entire internal structure** of the customer and their cart.

**2. Encapsulation Violation**

You're reaching deep into object internals—**violating encapsulation at multiple levels**.

**3. Maintenance Nightmare**

Imagine this change: You switch from using a Money wrapper to a BigDecimal for price representation in Product.

Now, every part of your codebase that dot-chased its way to product.getPrice() must be updated.

Your implementation detail leaked—and now you’re paying for it.

**4. Testability Issues**

Testing displayFirstItemPrice() becomes a *mocking marathon*.

To test it in isolation, you'd need to mock:

A Customer - That returns a ShoppingCart - That returns a List - That returns a CartItem - That returns a Product - That returns a Price

One function. Six mocks. Exhausting.

**Enter: The Law of Demeter (LoD)**

“Only talk to your immediate friends.”

The Law of Demeter says an object should only call methods on:

* Itself , Its own fields , Its method parameters , Objects it creates

In plain terms: **don't reach through one object to get to another.**

**Refactoring with LoD in Mind**

**Step 1: Add a method to ShoppingCart**

public class ShoppingCart {

// ... existing code

public Money getFirstItemPrice() {

if (items.isEmpty()) return Money.ZERO;

return items.get(0).getProduct().getPrice();

}

}

**Step 2: Add a method to Customer**

public class Customer {

// ... existing code

public Money getFirstCartItemPrice() {

return shoppingCart.getFirstItemPrice();

}

}

**Step 3: Update the OrderService**

public void displayFirstItemPrice(Customer customer) {

Money price = customer.getFirstCartItemPrice();

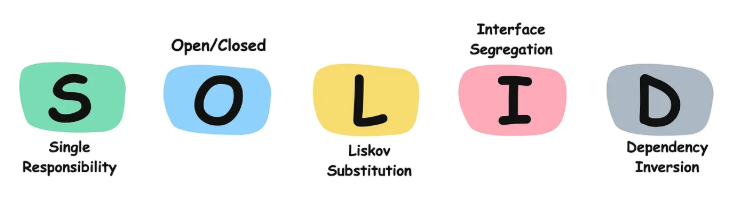
System.out.println("Price of the first item: " + price.getAmount());

}

**Benefits of Following the Law of Demeter**

* **Low Coupling:**Code changes in one place don’t ripple across your codebase.
* **Better Encapsulation:**Each class handles its own logic—no more peeking into internals.

**S.O.L.I.D Principles**



* 1. **Single Responsibility Principle**

Have you ever changed one part of your code... and suddenly, five unrelated things broke?

Eg:

public class Employee {

private String name;

private String email;

private double salary;

public void calculateSalary() { }

public void saveToDatabase() { }

public void generatePayslip() { }

public void sendPayslipEmail() { }

}

* If salary calculation logic changes, this class changes.
* If the payslip format changes, this class changes.
* If the DB schema changes, this class changes.
* If the email service API is replaced, this class changes again.

This class is tightly coupled to **four different reasons to change**. That’s a red flag.

**Enter: The Single Responsibility Principle**

A class should have one, and only one, reason to change.

Think of a **restaurant**.

* Would you hire one person to do all of these?

Cook the food and Take orders and Clean the tables and Do the accounts

* Of course not. You’d hire:

A **chef** and A **waiter** and A **cleaner** and An **accountant**

Each with a **single responsibility**.

**Applying SRP**

public class PayrollCalculator

public class EmployeeRepository

public class PayslipGenerator

public class EmailService

**Common Pitfalls While Applying SRP**

**1. Over-Splitting Responsibilities**

**The mistake:**Breaking a class into *too many* tiny classes that don't add real value.

**2. Confusing Methods with Responsibilities**

**The mistake:**Assuming each method must be its own class. Both methods deal with the same responsibility: sending emails.

**Example:**

public class EmailService {

public void sendWelcomeEmail() {}

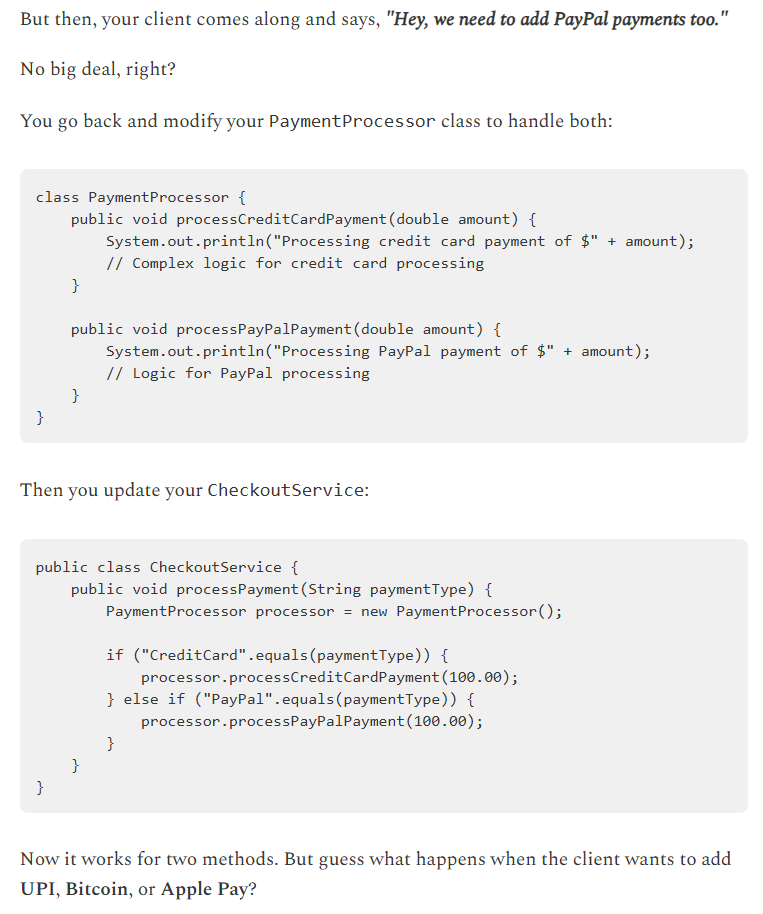
public void sendPayslipEmail() {}

}

* 1. **Open-Closed Principle**

Been afraid to change something because… well, it *might* break something else?





Each time, you're cracking open the PaymentProcessor class.Each modification carries the risk .

**Introducing the Open-Closed Principle (OCP)**

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

Look at that! Now, if the client wants to add "Bitcoin Payments" or "Apple Pay," what do we do?

1. Create a new class BitcoinPayment that implements PaymentMethod.
2. Implement its processPayment method.



1. **Liskov Substitution Principle**

Imagine you're building a system to manage different types of documents.

You start with a simple base class:

class Document:

def \_\_init\_\_(self, data: str):

self.data = data

def open(self) -> None:

print(f"Document opened. Data: {self.data[:20]}...")

def save(self, new\_data: str) -> None:

self.data = new\_data

print("Document saved.")

def get\_data(self) -> str:

return self.data

Now, a new requirement comes in:

“We need a **read-only** document type—for sensitive content like government reports or signed contracts.”

You think: *A ReadOnlyDocument is still a kind of Document*, so **inheritance** makes sense.

class ReadOnlyDocument(Document):

def \_\_init\_\_(self, data: str):

super().\_\_init\_\_(data)

def save(self, new\_data: str) -> None:

raise NotImplementedError("Cannot save a read-only document!")

**But** The client code expected *any* Document to be savable. But when it received a ReadOnlyDocument, that assumption exploded into a runtime exception.

**Introducing the Liskov Substitution Principle (LSP)**

 If a class S extends or implements class T, then you should be able to use S anywhere T is expected—without breaking the program’s behavior or logic.

**Step 1: Define Behavior Interfaces**  
Instead of having one base class with assumptions about mutability, let’s break responsibilities apart:

class Document(ABC):

@abstractmethod

def open(self) -> None:

pass

@abstractmethod

def get\_data(self) -> str:

pass

class Editable(ABC):

@abstractmethod

def save(self, new\_data: str) -> None:

pass

* Document: represents the ability to open and view data
* Editable: represents the capability to modify data

This clearly defines what each object can do—and prevents clients from assuming editability unless explicitly promised.

**Step 2: Implement EditableDocument and ReadOnlyDocument**

Now we implement our two concrete types:

**EditableDocument**

class EditableDocument implements Document, Editable { }

**ReadOnlyDocument**

class ReadOnlyDocument implements Document { }

Now:

* Both EditableDocument and ReadOnlyDocument are valid Document objects
* Only EditableDocument implements the Editable interface

1. **Interface Segregation Principle**

Imagine you're building a **media player app** that supports different types of media:

* **Audio files** (MP3, WAV)
* **Video files** (MP4, AVI)

Start with what feels like a convenient design: a single, unified interface that handles everything.

from abc import ABC, abstractmethod

class MediaPlayer(ABC):

@abstractmethod

def play\_audio(self, audio\_file: str) -> None:

pass

@abstractmethod

def stop\_audio(self) -> None:

pass

@abstractmethod

def adjust\_audio\_volume(self, volume: int) -> None:

pass

@abstractmethod

def play\_video(self, video\_file: str) -> None:

pass

@abstractmethod

def stop\_video(self) -> None:

pass

@abstractmethod

def adjust\_video\_brightness(self, brightness: int) -> None:

pass

@abstractmethod

def display\_subtitles(self, subtitle\_file: str) -> None:

pass

class AudioOnlyPlayer(MediaPlayer):

def play\_audio(self, audio\_file: str) -> None:

print(f"Playing audio file: {audio\_file}")

def stop\_audio(self) -> None:

print("Audio stopped.")

def adjust\_audio\_volume(self, volume: int) -> None:

print(f"Audio volume set to: {volume}")

def play\_video(self, video\_file: str) -> None:

raise NotImplementedError("Video playback not supported")

def stop\_video(self) -> None:

pass

def adjust\_video\_brightness(self, brightness: int) -> None:

raise NotImplementedError("Video operations not supported")

def display\_subtitles(self, subtitle\_file: str) -> None:

raise NotImplementedError("Subtitles not supported")

Even though AudioOnlyPlayer only needs audio methods, it’s forced to **implement unrelated video functionality**.

Now, imagine you add a new method to the interface, like enablePictureInPicture(). Suddenly, **all existing implementations—audio-only, video-only, or otherwise—must update**.

**Enter: The Interface Segregation Principle (ISP)**

Clients should not be forced to depend on methods they do not use.

from abc import ABC, abstractmethod

*# Audio-only capabilities*

class AudioPlayerControls(ABC):

@abstractmethod

def play\_audio(self, audio\_file: str) -> None:

pass

@abstractmethod

def stop\_audio(self) -> None:

pass

@abstractmethod

def adjust\_audio\_volume(self, volume: int) -> None:

pass

*# Video-only capabilities*

class VideoPlayerControls(ABC):

@abstractmethod

def play\_video(self, video\_file: str) -> None:

pass

@abstractmethod

def stop\_video(self) -> None:

pass

@abstractmethod

def adjust\_video\_brightness(self, brightness: int) -> None:

pass

@abstractmethod

def display\_subtitles(self, subtitle\_file: str) -> None:

pass

*# ModernAudioPlayer (Audio-only)*

class ModernAudioPlayer(AudioPlayerControls):

def play\_audio(self, audio\_file: str) -> None:

print(f"ModernAudioPlayer: Playing audio - {audio\_file}")

def stop\_audio(self) -> None:

print("ModernAudioPlayer: Audio stopped.")

def adjust\_audio\_volume(self, volume: int) -> None:

print(f"ModernAudioPlayer: Volume set to {volume}")

*# SilentVideoPlayer (Video-only)*

class SilentVideoPlayer(VideoPlayerControls):

def play\_video(self, video\_file: str) -> None:

print(f"SilentVideoPlayer: Playing video - {video\_file}")

def stop\_video(self) -> None:

print("SilentVideoPlayer: Video stopped.")

def adjust\_video\_brightness(self, brightness: int) -> None:

print(f"SilentVideoPlayer: Brightness set to {brightness}")

def display\_subtitles(self, subtitle\_file: str) -> None:

print(f"SilentVideoPlayer: Subtitles from {subtitle\_file}")

*# ComprehensiveMediaPlayer (Both audio + video)*

class ComprehensiveMediaPlayer(AudioPlayerControls, VideoPlayerControls):

def play\_audio(self, audio\_file: str) -> None:

print(f"ComprehensiveMediaPlayer: Playing audio - {audio\_file}")

def stop\_audio(self) -> None:

print("ComprehensiveMediaPlayer: Audio stopped.")

def adjust\_audio\_volume(self, volume: int) -> None:

print(f"ComprehensiveMediaPlayer: Audio volume set to {volume}")

def play\_video(self, video\_file: str) -> None:

print(f"ComprehensiveMediaPlayer: Playing video - {video\_file}")

def stop\_video(self) -> None:

print("ComprehensiveMediaPlayer: Video stopped.")

def adjust\_video\_brightness(self, brightness: int) -> None:

print(f"ComprehensiveMediaPlayer: Brightness set to {brightness}")

def display\_subtitles(self, subtitle\_file: str) -> None:

print(f"ComprehensiveMediaPlayer: Subtitles from {subtitle\_file}")

1. **Dependency Inversion Principle**

**Low-Level Module – Gmail Integration**

class GmailClient:

def send\_gmail(self, to\_address, subject\_line, email\_body):

print("Connecting to Gmail SMTP server...")

print("Sending email via Gmail to: " + to\_address)

print("Subject: " + subject\_line)

print("Body: " + email\_body)

*# ... actual Gmail API interaction logic ...*

print("Gmail email sent successfully!")

**High-Level Module – The Application's Email Service**

class EmailService:

def \_\_init\_\_(self):

self.gmail\_client = GmailClient()

def send\_welcome\_email(self, user\_email, user\_name):

subject = "Welcome, " + user\_name + "!"

body ="Thanks for signing up to our awesome platform.We're glad to haveyou"

self.gmail\_client.send\_gmail(user\_email, subject, body)

def send\_password\_reset\_email(self, user\_email):

subject = "Reset Your Password"

body = "Please click the link below to reset your password..."

self.gmail\_client.send\_gmail(user\_email, subject, body)

Then one day, a product manager asks:

“Can we switch from Gmail to Outlook for sending emails?”

Now imagine needing to:

* Support **multiple email providers** (Gmail, Outlook, SES, etc.)
* Dynamically select a provider based on configuration

**The Dependency Inversion Principle**

1. High-level modules should not depend on low-level modules. Both should depend on

abstractions (e.g., interfaces).

1. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

**Step 1: Define the Abstraction (The Contract)**

We need an interface that defines what any email sending mechanism should be able to do.

**Step 2: Concrete Implementations**

Now, our specific email clients (the "details") will implement the above interface.

* Gmail implementation
* Outlook implementation