DRY Principle

Have you ever copied the same validation logic into multiple classes?

Or written the same loop, query, or helper method across several files?

Or worse, updated a piece of business logic in one place but forgot it existed in two others?

If so, you have likely violated one of the most fundamental principles in software engineering: the DRY Principle, which stands for "Don’t Repeat Yourself."

This chapter explains the DRY principle through real-world examples, explores the problems caused by repetition, and offers practical advice to help you write cleaner and more maintainable code.

What Is the DRY Principle?
“Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.” — The Pragmatic Programmer

The DRY principle encourages you to avoid duplicating logic, behavior, or knowledge.

It applies not only to code, but also to:

Business rules
Configuration
Data models
Documentation
Tests
Whenever the same concept appears in more than one place, you introduce redundancy. Redundancy makes your system harder to maintain and more prone to bugs.

A Real-World Example
Imagine you are building a system to manage users across three modules: authentication, payments, and messaging.

Each module contains the same logic to validate email addresses:

Java

```java
public boolean isValidEmail(String email) {
    return email != null && email.contains("@") && email.contains(".");
}
```
Now suppose the business changes the rule. Email addresses must now end with ".com" or ".org".

If this logic is duplicated across modules, you face a major problem. You need to update every location where the validation code exists. If you miss even one, the system becomes inconsistent, and bugs start to appear. You have violated the DRY principle and created technical debt.

Why Repetition Is a Problem

1. Harder to Maintain
When a rule or piece of logic changes, you must find and update every occurrence. Missing even one leads to inconsistent behavior.

2. Higher Risk of Bugs
More copies mean more chances to introduce errors. A typo or mismatch in one version of the logic can cause unexpected failures.

3. Bloated Codebase
Redundant logic adds noise to your codebase, making it harder to understand what is unique versus what is shared.

4. Poor Test Coverage
When logic is repeated in many places, each version needs its own set of tests. This increases the testing effort and complexity.

Copy-Paste Is a Red Flag
Copying and pasting code might seem convenient, but it often leads to long-term problems.

Ask yourself:

If I need to change this logic in the future, will I remember all the places where it exists?

If the answer is no or even uncertain, you are creating risk. Following the DRY principle reduces that risk.

How to Apply DRY
Let’s refactor the email validation example by extracting the common logic.

Step 1: Create a Utility Class

Java

```java
public class EmailValidator {
    public static boolean isValid(String email) {
        return email != null &&
               email.contains("@") &&
               email.contains(".") &&
               (email.endsWith(".com") || email.endsWith(".org"));
    }
}
```

Step 2: Use It Across Modules

Java

```java
if (EmailValidator.isValid(user.getEmail())) {
    // Proceed with business logic
}
```

Now the email validation logic lives in one place. You have a single source of truth. Any future updates only need to be made once, and all modules remain consistent.

DRY Before and After
Without DRY

Java

```java
public void registerUser(User user) {
    if (user.getEmail() == null || !user.getEmail().contains("@")) {
        throw new IllegalArgumentException("Invalid email");
    }
    // Additional logic
}

public void sendNewsletter(User user) {
    if (user.getEmail() == null || !user.getEmail().contains("@")) {
        return;
    }
    // Additional logic
}
```

With DRY Applied

Java

```
public class EmailValidator {
    public static boolean isValid(String email) {
        return email != null && email.contains("@");
    }
}

public void registerUser(User user) {
    if (!EmailValidator.isValid(user.getEmail())) {
        throw new IllegalArgumentException("Invalid email");
    }
    // Additional logic
}

public void sendNewsletter(User user) {
    if (!EmailValidator.isValid(user.getEmail())) {
        return;
    }
    // Additional logic
}
```

By centralizing the validation logic, your code becomes more consistent and easier to maintain.

When It Is Okay to Repeat

The DRY principle is a guideline, not a strict rule. There are times when repetition is acceptable.

1. Avoid Premature Abstractions

Do not extract shared code too early. Let duplication reveal itself first. Abstractions created too soon can be misleading or hard to maintain.

“Duplication is far cheaper than the wrong abstraction.” — Sandi Metz

2. Keep Tests Readable

In some cases, repeating a bit of test code improves clarity. Tests should be easy to read and understand.

3. Keep It Simple

If a line of code is extremely simple and unlikely to change, it may be better to repeat it rather than create a new layer of indirection.

Final Thoughts
The DRY principle is more than just a tip for cleaner code. It is a mindset that helps you reduce risk, improve consistency, and write software that can evolve gracefully.

The next time you find yourself copying code, stop and ask: Can I extract this instead?

Following the DRY principle might take a bit more time up front, but it saves you much more time and effort in the long run.