Single Responsibility Principle (SRP)

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

Or added a small feature... and ended up editing dozens of lines across a single class?

If yes, you've probably encountered a violation of one of the most important design principles in software engineering: **The Single Responsibility Principle (SRP).**

Let's understand it with a problem and why it violates SRP.

The Problem: The God Class

Meet the classic **God Class**. You've probably seen it before. Maybe even written it.

Java

```
class Employee {
    private String name;
    private String email;
    private double salary;

    // Constructor, getters, setters...

    public void calculateSalary() {
        // Complex salary calculation logic
        // Includes tax calculations
    }

    public void saveToDatabase() {
        // Connect to database
        // Prepare SQL
        // Execute query
    }

    public void generatePayslip() {
        // Format payslip
        // Add company logo
        // Convert to PDF
```

```
    }

    public void sendPayslipEmail() {
        // Connect to email server
        // Create email with attachment
        // Send email
    }
}
```

Python

```
class Employee:
    def __init__(self, name: str, email: str, salary: float):
        self._name = name
        self._email = email
        self._salary = salary

    def calculate_salary(self):
        # Complex salary calculation logic
        # Includes tax calculations
        pass

    def save_to_database(self):
        # Connect to database
        # Prepare SQL
        # Execute query
        pass

    def generate_payslip(self):
        # Format payslip
        # Add company logo
        # Convert to PDF
        pass

    def send_payslip_email(self):
        # Connect to email server
        # Create email with attachment
        # Send email
        pass
```

At first glance, this may seem convenient — everything about an employee in one place.

But let's pause and examine what this class is actually doing:

- Calculating salary

- Saving data to the database
- Generating a payslip
- Sending an email

That's **four distinct responsibilities** rolled into one class.

- 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. — Robert C. Martin (Uncle Bob)

In simple words: **A class should do one thing and do it well.**

The Single Responsibility Principle (SRP) is the 'S' in the famous **SOLID** principles of object-oriented design.

But what exactly is a "responsibility"?

It's not a method. It's not a function. It's **a reason for the class to change.**

Ask yourself this:

How many reasons might someone need to update this class in the future?

If the answer is more than one — it's likely breaking SRP.

Real-World Analogy

Think of a **restaurant**.

Would you hire one person to do all of these?

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

Of course not. You'd hire:

- A **chef**
- A **waiter**
- A **cleaner**
- An **accountant**

Each with a **single responsibility**.

Why should your code be any different?

Why Does SRP Matter?

- **Easier to read:** You immediately understand what the class is supposed to do. No surprises.
- **Easier to test:** Smaller responsibilities mean smaller test cases and fewer dependencies.
- **Less brittle:** Changes in one responsibility don't ripple across unrelated parts of the code.
- **Easier to reuse:** Small, focused classes are more flexible and can be reused in different contexts.
- **Scales better:** Teams can own different parts of the system without stepping on each other's toes.

Applying SRP

Time to fix our original `Employee` God Class using SRP.

We'll take each distinct responsibility from the original `Employee` class and extract it into its own focused, well-named class.

- **Calculating Salary** is one responsibility.
- **Saving to database** is another.
- **Generating Payslip** is another
- **Sending Payslip Email** is yet another.

They all deserve their own classes.

The Core: `Employee` Class

Let's start by slimming down `Employee` into a simple data class:

Python

```
class Employee:
    def __init__(self, name: str, email: str, base_salary: float):
        self._name = name
        self._email = email
        self._base_salary = base_salary

    def get_name(self) -> str:
        return self._name

    def get_email(self) -> str:
        return self._email

    def get_base_salary(self) -> float:
        return self._base_salary
```

Java

```
class Employee {
    private String name;
    private String email;
    private double baseSalary;

    public Employee(String name, String email, double baseSalary) {
        this.name = name;
        this.email = email;
        this.baseSalary = baseSalary;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }
    public double getBaseSalary() { return baseSalary; }
}
```

This class now does one thing: **represent an employee**.It doesn't calculate salary, store itself, or send emails. That's the job of others.

Responsibility 1: Salary Calculation

Java

```
class PayrollCalculator {
```

```
    public double calculateNetPay(Employee employee) {
        double base = employee.getBaseSalary();
        double tax = base * 0.2;  // Sample tax logic
        double benefits = 1000;   // Fixed benefit deduction
        return base - tax + benefits;
    }
}
```

Python

```
class PayrollCalculator:
    def calculate_net_pay(self, employee: Employee) -> float:
        base = employee.get_base_salary()
        tax = base * 0.2  # Sample tax logic
        benefits = 1000   # Fixed benefit deduction
        return base - tax + benefits
```

This class handles **just the logic** of calculating an employee's net pay.If payroll policy changes, we only update this class.

Responsibility 2: Persistence to Database

Java

```
class EmployeeRepository {
    public void save(Employee employee) {
        // Example: JDBC code or ORM logic
        System.out.println("Saving employee " + employee.getName() + " to database...");
    }
}
```

Python

```
class EmployeeRepository:
    def save(self, employee: Employee):
        print(f"Saving employee {employee.get_name()} to database...")
```

The responsibility for talking to the database belongs here. You can swap out JDBC for JPA or another data layer without touching the rest of the system.

Responsibility 3: Payslip Generation

Java

```
class PayslipGenerator {
    public String generatePayslip(Employee employee, double netPay) {
```

```
        return "Payslip for: " + employee.getName() + "\n" +
               "Email: " + employee.getEmail() + "\n" +
               "Net Pay: ₹" + netPay + "\n" +
               "---------------------------\n";
    }
}
```

Python

```
class PayslipGenerator:
    def generate_payslip(self, employee: Employee, net_pay: float) -> str:
        return (
            f"Payslip for: {employee.get_name()}\n"
            f"Email: {employee.get_email()}\n"
            f"Net Pay: ₹{net_pay}\n"
            "---------------------------\n"
        )
```

This class handles the formatting and creation of a textual payslip. You can replace the output format later (PDF, HTML, JSON) without affecting the rest of your codebase.

Responsibility 4: Emailing the Payslip

Java

```
class EmailService {
    public void sendPayslip(Employee employee, String payslip) {
        System.out.println("Sending payslip to: " + employee.getEmail());
        // Simulate email with a print
        System.out.println(payslip);
    }
}
```

Python

```
class EmailService:
    def send_payslip(self, employee: Employee, payslip: str):
        print(f"Sending payslip to: {employee.get_email()}")
        print(payslip)
```

This class is responsible only for **sending emails**. It doesn't calculate anything. It doesn't generate the report. It just sends it.

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.

Example:

Creating separate classes for `TaxCalculator`, `BonusCalculator`, `BenefitsCalculator`, and `SalaryAggregator` — when all of these could be grouped into a cohesive `PayrollCalculator.`

Why it's a problem:

- Leads to **unnecessary complexity**
- Makes the system **harder to understand**
- Increases overhead in navigating and wiring too many classes

Focus on **cohesion**, not fragmentation. Group logic that changes together or belongs to the same business concern.

2. Confusing Methods with Responsibilities

The mistake: Assuming each method must be its own class.

Example:

```
class EmailService {

   public void sendWelcomeEmail() {}

   public void sendPayslipEmail() {}

}
```

Python

```
class EmailService:

   def send_welcome_email(self):

      pass
```

```
    def send_payslip_email(self):

        Pass
```

Some developers might try to split this into:

- `WelcomeEmailSender`
- `PayslipEmailSender`

Why it's a problem:

- Both methods deal with the same **responsibility**: sending emails
- Splitting them adds more boilerplate without clear benefits

Don't confuse the *number of methods* with *number of responsibilities*. If the methods serve the same purpose (sending emails), it's fine to keep them together.

3. Ignoring SRP in Small or Utility Classes

The mistake: Thinking, "This class is small and works fine — no need to split it."

Example: A utility class that starts off simple but quietly grows:

Java

```
class ReportUtils {

    public void generateCSV() {}

    public void sendReportEmail() {}

    public void archiveReport() {}

}
```

Python

```
class ReportUtils:
```

```
    def generate_csv(self):
        pass
    def send_report_email(self):
        pass
    def archive_report(self):
        Pass
```

Why it's a problem:

- These responsibilities often evolve independently
- Small changes to one feature might introduce bugs in others

Watch for **creeping responsibilities** even in utility classes. Apply SRP **early** before small classes become unmanageable.

4. Misunderstanding "Reason to Change"

The mistake: Taking the "reason to change" definition too literally or too vaguely.

Bad interpretation: "I only ever change this class when a stakeholder asks for a change, so it has one reason to change."

Why it's a problem: SRP is **not** about who asks for the change, but **what kind of change** is being made.

Clarify the responsibility in terms of **business logic or technical behavior**. Ask: *Is this logic cohesive, or are unrelated concerns bundled together?*

Common Questions About SRP

"Doesn't this create too many small classes?"

Yes, **you'll likely end up with more classes** — but that's not a bad thing.

Instead of having one massive class doing everything poorly, you have smaller, focused classes doing one thing well. These classes are:

- Easier to **read**
- Easier to **test**
- Easier to **maintain**
- Easier to **reuse**

Think of it as **managing complexity through separation**, not increasing it. When responsibilities are clearly separated, your system becomes easier to reason about — even if the file count grows.

SRP helps reduce cognitive load, even if it increases the class count.

"How small should a responsibility be?"

There's no hard-and-fast rule. It depends on your domain and use case. But here's a simple **heuristic**:

If you need to use the word "and" or "or" to describe what your class does, it probably has more than one responsibility.

Example:

- "This class generates reports *and* sends emails." → Two responsibilities

Another tip: If the **reasons for change** are unrelated — say, a tax policy update vs. a new email template, your class is likely doing too much.

"Does SRP apply beyond classes?"

Absolutely. SRP can and should be applied across multiple levels:

- **Class:** A class should have one reason to change.
- **Method:** A method should do one thing.
- **Module:** A module should encapsulate one area of functionality.
- **Service:** A service (or microservice) should serve a single domain.
- **System:** Even large systems can be organized around single responsibilities.

SRP is a mindset: **separate concerns to improve clarity and adaptability**, no matter the scale.

"Does SRP make testing harder or easier?"

When a class does only one thing, testing becomes straightforward.

You don't have to:

- Mock half the world
- Stub unrelated services
- Worry about hidden side effects

You can focus on the specific input/output of a class without worrying about unrelated functionality baked into it.

"What if the responsibilities are related?"

Sometimes it's okay to group closely related behaviors into one class.

For example, a `EmailService` class that:

- Sends welcome emails
- Sends password reset emails
- Sends payslip emails

That's fine — they all fall under the same responsibility: **sending emails**.

But if that class also starts doing PDF generation or user authentication, it's time to split it up.

"Is SRP just another rule I *have* to follow?"

Think of SRP less as a strict rule and more as a **guiding principle**.

It won't always be obvious where to draw the line, and that's okay.

Use SRP to:

- Make your code easier to evolve

- Isolate reasons for change
- Reduce the blast radius of bugs

When used wisely, SRP becomes a **tool to manage change and complexity**, not a burden.