✅ Case Study: Single Responsibility Principle (SRP) — "Invoice Processing System"

🎯 Business Context:

Gunika is designing an `Invoice` module for a retail app. It needs to:

1. Store invoice data (like items, amount)
2. Save the invoice to a database
3. Print the invoice
4. Send the invoice via email

She puts all logic in one class, which becomes a **God class** — hard to maintain and modify.

❌ Problematic Code (Violates SRP)

```
public class Invoice {

    private int invoiceId;
    private String customer;
    private double amount;

    public Invoice(int invoiceId, String customer, double amount) {
        this.invoiceId = invoiceId;
        this.customer = customer;
        this.amount = amount;
    }

    // 1. Data logic
    public double calculateTotal() {
        return amount + (amount * 0.18); // tax 18%
    }

    // 2. DB logic
    public void saveToDB() {
        System.out.println("Saving invoice to database...");
    }

    // 3. Printing logic
    public void printInvoice() {
        System.out.println("Printing invoice...");
    }

    // 4. Email logic
    public void emailInvoice() {
        System.out.println("Emailing invoice...");
    }
}
```

❗ What's Wrong?

- It **violates SRP**: This class has **4 responsibilities**:

    - Holding data
    - Database operations
    - Printing
    - Emailing
- Even a small change (like email format or tax rule) requires **modifying this class**.
- Testing becomes harder, changes risk breaking unrelated logic.

✅ Refactored Code (Applies SRP)

---

1. Invoice.java – Only Data & Basic Logic

```
public class Invoice {
    private int invoiceId;
    private String customer;
    private double amount;

    public Invoice(int invoiceId, String customer, double amount) {
        this.invoiceId = invoiceId;
        this.customer = customer;
        this.amount = amount;
    }

    public double calculateTotal() {
        return amount + (amount * 0.18); // tax
    }

    public String getDetails() {
        return "Invoice ID: " + invoiceId + ", Customer: " + customer + ",
Total: " + calculateTotal();
    }

    // Getters (if needed)
}
```

---

2. InvoiceRepository.java – DB Responsibility

```
public class InvoiceRepository {
    public void save(Invoice invoice) {
        System.out.println("Saving invoice to database: " +
invoice.getDetails());
    }
}
```

---

3. InvoicePrinter.java – Printing Responsibility

```
public class InvoicePrinter {
    public void print(Invoice invoice) {
        System.out.println("Printing invoice: " + invoice.getDetails());
    }
}
```

---

4. InvoiceEmailer.java – Email Responsibility

```
public class InvoiceEmailer {
    public void sendEmail(Invoice invoice) {
        System.out.println("Sending invoice email: " + invoice.getDetails());
    }
}
```

✅ Main Usage

```
public class Main {
    public static void main(String[] args) {
        Invoice invoice = new Invoice(101, "Gunika", 5000);

        InvoiceRepository repo = new InvoiceRepository();
        InvoicePrinter printer = new InvoicePrinter();
        InvoiceEmailer emailer = new InvoiceEmailer();

        repo.save(invoice);
        printer.print(invoice);
        emailer.sendEmail(invoice);
    }
}
```

🧠 Real-World Analogy

Bad Design: You ask a single employee to do billing, print receipts, email customers, and do data entry.

Good Design (SRP): Assign billing to one person, printing to another, emailing to another — each role is specialized and easier to replace or upgrade.

✅ Benefits of SRP:

Aspect	Benefit
Readability	Easier to understand each class's purpose
Maintainability	Change in one area doesn't affect others
Testability	Smaller units are easier to test
Reusability	Can reuse `InvoicePrinter`, `InvoiceEmailer` elsewhere

🧠 Discussion Points for Students

- What happens if we change the tax rule?
- What if we want to log printed invoices?
- How would we test just the print functionality?