

Great! Let's go through each **SOLID** principle one by one with **easy-to-understand real-world examples** and **code illustrations** (in Java) where useful. These principles help design software that's **maintainable, extensible, and scalable**.

1. S - Single Responsibility Principle (SRP)

✅ Definition:

A class should have **only one reason to change**, meaning it should do **only one job**.

🏠 Real-world Example:

Imagine a **Housekeeper** in a hotel. If one person is doing **cleaning, cooking, and billing**, that's a bad idea. Each task should be done by a separate person.

- **Cleaning** → Housekeeper
- **Cooking** → Chef
- **Billing** → Accountant

If cooking rules change, we shouldn't have to modify the Housekeeper or Accountant.

👨‍💻 Code Example (bad):

```
java
CopyEdit
class EmployeeService {
    public void saveEmployee(Employee emp) { /* save to DB */ }
    public void generatePayslip(Employee emp) { /* generate PDF */ }
    public void sendEmail(Employee emp) { /* email logic */ }
}
```

🔧 Refactored (good):

```
java
CopyEdit
class EmployeeRepository {
    public void save(Employee emp) { /* save logic */ }
```

```
}

class PayslipGenerator {
    public void generate(Employee emp) { /* generate logic */ }
}

class EmailService {
    public void send(Employee emp) { /* email logic */ }
}
```

2. O - Open/Closed Principle (OCP)

✅ Definition:

A class should be **open for extension** but **closed for modification**.

🏠 Real-world Example:

Think about a **Netflix recommendation engine**. When they add a new type of content (e.g. podcasts), they don't rewrite the whole system. They **extend** it.

👤 Code Example (bad):

```
java
CopyEdit
class DiscountCalculator {
    public double calculate(String customerType) {
        if (customerType.equals("Regular")) return 0.1;
        else if (customerType.equals("Premium")) return 0.2;
        else return 0;
    }
}
```

🔧 Refactored using OCP:

```
java
CopyEdit
interface Discount {
    double calculate();
}
```

```

}

class RegularDiscount implements Discount {
    public double calculate() { return 0.1; }
}

class PremiumDiscount implements Discount {
    public double calculate() { return 0.2; }
}

class DiscountService {
    public double getDiscount(Discount discount) {
        return discount.calculate();
    }
}

```

You can now **add new discount types** without touching existing code.

3. L - Liskov Substitution Principle (LSP)

✓ Definition:

Subtypes must be **replaceable** for their parent types without breaking the program.

Or if parent have two methods fly and run sub class must have fly and run if one is not present in sub then dont inherit that

🏠 Real-world Example:

A **bird** can fly. So can a **sparrow**. But an **ostrich** can't fly. So Ostrich **should not extend** Bird if Bird has `fly()` method.

👤 Code Example (bad):

```

class Bird {
    public void fly() { System.out.println("Flying"); }
}

```

```
class Ostrich extends Bird {  
    @Override  
    public void fly() { throw new UnsupportedOperationException(); }  
}
```

This breaks LSP.



Refactored:

```
interface Bird { }  
  
interface FlyingBird extends Bird {  
    void fly();  
}  
  
class Sparrow implements FlyingBird {  
    public void fly() { System.out.println("Flying"); }  
}  
  
class Ostrich implements Bird {  
    // No fly method  
}
```

Now we can substitute `Sparrow` for `FlyingBird` safely.

4. I - Interface Segregation Principle (ISP)



Definition:

Clients should **not be forced** to depend on interfaces they **don't use**.



Real-world Example:

In a restaurant, a **Waiter** shouldn't be required to cook just because he implements a general `Staff` interface.



Code Example (bad):

```
interface RestaurantWorker {
    void cook();
    void serve();
}

class Waiter implements RestaurantWorker {
    public void cook() { /* ??? */ }
    public void serve() { System.out.println("Serving"); }
}
```

Refactored:

```
interface Cook {
    void cook();
}

interface Server {
    void serve();
}

class Chef implements Cook {
    public void cook() { System.out.println("Cooking"); }
}

class Waiter implements Server {
    public void serve() { System.out.println("Serving"); }
}
```

Each class implements only what it **needs**.

5. D - Dependency Inversion Principle (DIP)

Definition:

High-level modules should **not depend on low-level modules**. Both should depend on **abstractions**.

Real-world Example:

A **remote control** depends on a **TV interface**, not a specific brand. It can work with Samsung, LG, or Sony if they implement that interface.

Code Example (bad):

```
class LightBulb {
    public void turnOn() {}
}

class Switch {
    private LightBulb bulb = new LightBulb();
    public void operate() {
        bulb.turnOn();
    }
}
```

`Switch` is tightly coupled with `LightBulb`.

Refactored:

```
java
CopyEdit
interface Switchable {
    void turnOn();
}

class LightBulb implements Switchable {
    public void turnOn() { System.out.println("Bulb On"); }
}

class Fan implements Switchable {
    public void turnOn() { System.out.println("Fan On"); }
}

class Switch {
    private Switchable device;

    public Switch(Switchable device) {
```

```
        this.device = device;
    }




    public void operate() {
        device.turnOn();
    }
}
```

Now `Switch` can operate **any device** that implements `Switchable`.

Let me know if you'd like a **printable PDF** or **interactive code samples** for these! Want to go deeper into any one of them?

40

Perfect — let's now go principle by principle, showing:

-  Bad Practice (Violating SOLID)
-  Good Practice (Following SOLID)
-  Explanation for both

I'll continue with full Java examples and real-world context for each.

1. Single Responsibility Principle (SRP)

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

Bad Practice

java

CopyEdit

```
public class InvoiceManager {
```

```
public void saveToDatabase(Invoice invoice) {
    System.out.println("Saving invoice to DB...");
}

public void printInvoice(Invoice invoice) {
    System.out.println("Printing invoice...");
}

public void sendEmail(Invoice invoice) {
    System.out.println("Sending invoice email...");
}
}
```

Why it's bad:

- This class does 3 unrelated things: persistence, printing, and emailing.
- A change in printing logic (e.g. format) will require touching this class — affecting other unrelated behaviors.

Good Practice

java

CopyEdit

// Handles only persistence

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

// Handles only printing

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



```
// Handles only email
public class InvoiceEmailService {
    public void sendEmail(Invoice invoice) {
        System.out.println("Sending invoice email...");
    }
}
```

Why it's good:

- Each class has one reason to change.
 - Independent development & testing for email, printing, DB logic.
-

2. Open/Closed Principle (OCP)

Software should be open for extension but closed for modification.

Bad Practice

java

CopyEdit

```
public class AreaCalculator {
    public double calculateArea(Object shape) {
        if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return Math.PI * c.radius * c.radius;
        } else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.length * r.width;
        }
        return 0;
    }
}
```

❌ Why it's bad:

- Every time a new shape is added (Triangle, Hexagon), you have to modify this class.
 - Violates OCP.
-

✅ Good Practice

java

CopyEdit

```
// Shape.java
```

```
public interface Shape {  
    double area();  
}
```

```
public class Circle implements Shape {  
    double radius;  
    public Circle(double r) { this.radius = r; }  
    public double area() { return Math.PI * radius * radius; }  
}
```

```
public class Rectangle implements Shape {  
    double length, width;  
    public Rectangle(double l, double w) { this.length = l; this.width  
= w; }  
    public double area() { return length * width; }  
}
```

java

CopyEdit

```
public class AreaCalculator {  
    public double calculateArea(Shape shape) {  
        return shape.area();  
    }  
}
```

❌ Why it's good:

- You can add new shapes (Triangle, Square) without touching **AreaCalculator**.
 - Open for extension, closed for modification.
-

3. Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

Bad Practice

java

CopyEdit

```
public class Bird {
    public void fly() {
        System.out.println("Flying");
    }
}

public class Ostrich extends Bird {
    public void fly() {
        throw new UnsupportedOperationException("Ostriches can't fly");
    }
}
```

Why it's bad:

- **Ostrich** breaks the contract of **Bird**. You expect **Bird** to fly, but **Ostrich** can't.
 - Will cause runtime exceptions when passed as a **Bird**.
-

Good Practice

java

CopyEdit

```
public interface Bird { }

public interface FlyingBird extends Bird {
    void fly();
}

public class Sparrow implements FlyingBird {
    public void fly() {
        System.out.println("Flying...");
    }
}

public class Ostrich implements Bird {
    // Doesn't implement fly()
}
```

■ Why it's good:

- We now only expect flight from FlyingBirds, not all birds.
- **Sparrow** can be safely used wherever **FlyingBird** is expected.

4. Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they don't use.

✗ Bad Practice

java

CopyEdit

```
public interface MultiFunctionDevice {
    void print();
    void scan();
    void fax();
}
```

```
public class BasicPrinter implements MultifunctionDevice {
    public void print() { System.out.println("Printing..."); }
    public void scan() { throw new UnsupportedOperationException(); }
    public void fax() { throw new UnsupportedOperationException(); }
}
```

Why it's bad:

- **BasicPrinter** is forced to implement methods it doesn't support.
 - Violates ISP.
-

Good Practice

java

CopyEdit

```
public interface Printer {
    void print();
}

public interface Scanner {
    void scan();
}

public interface Fax {
    void fax();
}

public class BasicPrinter implements Printer {
    public void print() {
        System.out.println("Printing...");
    }
}
```

Why it's good:

- **BasicPrinter** implements only what it needs.
 - Each interface is small and specific to client needs.
-

5. Dependency Inversion Principle (DIP)

Depend on abstractions, not concrete implementations.

Bad Practice

java

CopyEdit

```
public class EmailService {
    public void send(String message) {
        System.out.println("Email sent: " + message);
    }
}

public class Notification {
    private EmailService emailService = new EmailService();

    public void notifyUser(String message) {
        emailService.send(message);
    }
}
```

Why it's bad:

- **Notification** class is tightly coupled to **EmailService**.
 - Cannot swap for SMS or PushNotification easily.
-

Good Practice

java

CopyEdit

```
public interface MessageService {
    void send(String message);
}

public class EmailService implements MessageService {
    public void send(String message) {
        System.out.println("Email: " + message);
    }
}

public class SMSService implements MessageService {
    public void send(String message) {
        System.out.println("SMS: " + message);
    }
}

public class Notification {
    private final MessageService service;

    public Notification(MessageService service) {
        this.service = service;
    }

    public void notifyUser(String message) {
        service.send(message);
    }
}

java
CopyEdit
public class Main {
    public static void main(String[] args) {
        Notification emailNotification = new Notification(new
EmailService());
        Notification smsNotification = new Notification(new
SMSService());
    }
}
```

```
        emailNotification.notifyUser("Invoice Ready");
        smsNotification.notifyUser("Your OTP is 123456");
    }
}
```

Why it's good:

- **Notification** depends on an abstraction (**MessageService**), not a concrete class.
- We can switch implementations without changing **Notification**.

Output:

```
vbnet
CopyEdit
Email: Invoice Ready
SMS: Your OTP is 123456
```

Would you like me to bundle these into a downloadable Maven/Gradle project with unit tests as well?

4o

Search

Reason

Deep research

Create image

ChatGPT can make mistakes. Check important info. See Cookie Preferences.