SOLID

Writing Better Code with Java

By Marco A Vincenzi

🚀 If you've ever struggled with messy, hard-to-maintain code, it was probably missing SOLID principles! These five principles help make software modular, flexible, and testable. Let’s break them down with practical Java examples.

📌 Single Responsibility Principle (SRP)
"A class should have only one reason to change."

Bad Example: One class handles both order processing and invoice generation.

```
class Order {
    void processOrder() { /* order logic */ }
    void generateInvoice() { /* invoice logic */ }
}

```

Good Example: Separate responsibilities. Each class now has a single responsibility!

```
class OrderProcessor {
    void processOrder() { /* order logic */ }
}
class InvoiceGenerator {
    void generateInvoice() { /* invoice logic */ }
}

```

📌 Open/Closed Principle (OCP)

"Open for extension, closed for modification."

Bad Example: Every new payment method requires modifying the class.

```
class PaymentProcessor {
    void processPayment(String type) {
        if (type.equals("CREDIT_CARD")) { /* logic */ }
        else if (type.equals("PAYPAL")) { /* logic */ }
    }
}
```

Good Example: Use abstractions. Now, we can add new payment types without modifying *PaymentProcessor*!

```
interface Payment {
    void pay();
}
class CreditCardPayment implements Payment {
    public void pay() { /* logic */ }
}
class PayPalPayment implements Payment {
    public void pay() { /* logic */ }
}
class PaymentProcessor {
    void processPayment(Payment payment) {
        payment.pay();
    }
}
```

📌 Liskov Substitution Principle (LSP)
"Subclasses should be replaceable for their base class without breaking functionality."

Bad Example: *Penguin* inherits a *fly()* method it cannot use.

```
class Bird {
    void fly() { /* flight logic */ }
}
class Penguin extends Bird {
    void fly() { throw new UnsupportedOperationException(); }
}

```

Good Example: Create proper abstractions. Now, only birds that actually fly implement *Flyable*.

```
abstract class Bird { }
interface Flyable {
    void fly();
}
class Sparrow extends Bird implements Flyable {
    public void fly() { /* flight logic */ }
}
class Penguin extends Bird { /* no fly() method */ }

```

📌 Interface Segregation Principle (ISP)

"Don't force clients to depend on methods they don't use."

Bad Example: One interface forces *Robot* to implement *eat()*.

```
interface Worker {
    void work();
    void eat();
}
class Robot implements Worker {
    public void work() { /* logic */ }
    public void eat() { throw new UnsupportedOperationException(); }
}

```

Good Example: Split interfaces. Now, *Robot* doesn't need an unnecessary *eat()* method.

```
interface Workable {
    void work();
}
interface Eatable {
    void eat();
}
class Robot implements Workable {
    public void work() { /* logic */ }
}
class Human implements Workable, Eatable {
    public void work() { /* logic */ }
    public void eat() { /* logic */ }
}

```

📌 Dependency Inversion Principle (DIP)

"Depend on abstractions, not concrete implementations."

Bad Example: *DataManager* is tightly coupled to *MySQLDatabase*.

```
class MySQLDatabase {
    void connect() { /* connection logic */ }
}
class DataManager {
    private MySQLDatabase db = new MySQLDatabase();
    void fetchData() { db.connect(); }
}

```

Good Example: Use an interface. Now, we can swap *MySQLDatabase* with *PostgreSQLDatabase* without modifying *DataManager*.

```
interface Database {
    void connect();
}
class MySQLDatabase implements Database {
    public void connect() { /* connection logic */ }
}
class DataManager {
    private Database db;
    DataManager(Database db) { this.db = db; }
    void fetchData() { db.connect(); }
}

```