**SOLID** is a collection of five key principles that help us write better, more maintainable object-oriented software. These principles make your codebase more readable, flexible, and scalable.

**✅ 1. Single Responsibility Principle (SRP)**

**Every class should focus on a single task.**  
If a class handles multiple jobs, it's harder to maintain and reuse.

**🚫 Problematic Design**

java

CopyEdit

public class Invoice {

public void calculateTotal() {}

public void printInvoice() {}

public void saveToDatabase() {}

}

This class is doing too much — calculations, printing, and saving data.

**✅ Improved Design**

java

CopyEdit

public class Invoice {

public void calculateTotal() {}

}

public class InvoicePrinter {

public void print(Invoice invoice) {}

}

public class InvoiceRepository {

public void save(Invoice invoice) {}

}

Now each class has a **single** job — easier to test, extend, or reuse.

**✅ 2. Open/Closed Principle (OCP)**

**Software should be open to extension, but closed to modification.**  
You should be able to add features without changing existing code.

**🚫 Problematic Design**

java

CopyEdit

public class DiscountService {

public int getDiscount(String type) {

if (type.equals("student")) return 10;

else if (type.equals("senior")) return 20;

return 0;

}

}

Adding a new user type forces you to modify the function.

**✅ Improved Design**

java

CopyEdit

interface Discount {

int getDiscount();

}

class StudentDiscount implements Discount {

public int getDiscount() {

return 10;

}

}

class SeniorDiscount implements Discount {

public int getDiscount() {

return 20;

}

}

Just add a new class — no need to touch existing code.

**✅ 3. Liskov Substitution Principle (LSP)**

**Subclasses must be substitutable for their base class without issues.**

**🚫 Problematic Design**

java

CopyEdit

class Bird {

public void fly() {}

}

class Ostrich extends Bird {

@Override

public void fly() {

throw new UnsupportedOperationException("Ostrich can't fly!");

}

}

An Ostrich *is* a bird, but this behavior breaks expectations.

**✅ Improved Design**

java

CopyEdit

class Bird {}

class FlyingBird extends Bird {

public void fly() {}

}

class Sparrow extends FlyingBird {

@Override

public void fly() {

System.out.println("Sparrow is flying!");

}

}

class Ostrich extends Bird {

// No fly() here, as Ostrich doesn't fly

}

Now, only birds that can fly implement flying behavior.

**✅ 4. Interface Segregation Principle (ISP)**

**Prefer many small, specific interfaces over a large, all-in-one interface.**

**🚫 Problematic Design**

java

CopyEdit

interface Worker {

void work();

void eat();

}

class Robot implements Worker {

public void work() {}

public void eat() {

throw new UnsupportedOperationException("Robots don't eat!");

}

}

**✅ Improved Design**

java

CopyEdit

interface Workable {

void work();

}

interface Eatable {

void eat();

}

class Human implements Workable, Eatable {

public void work() {}

public void eat() {}

}

class Robot implements Workable {

public void work() {}

}

Each class only implements what it needs.

**✅ 5. Dependency Inversion Principle (DIP)**

**High-level modules should depend on abstractions, not concrete classes.**

**🚫 Problematic Design**

java

CopyEdit

class MySQLDatabase {

public void save(String data) {}

}

class App {

private MySQLDatabase db = new MySQLDatabase();

}

App is tightly coupled to a specific database.

**✅ Improved Design**

java

CopyEdit

interface Database {

void save(String data);

}

class MySQLDatabase implements Database {

public void save(String data) {

System.out.println("Saved to MySQL");

}

}

class App {

private Database db;

public App(Database db) {

this.db = db;

}

}

Now the app works with any database that follows the Database interface.

**🧩 Summary of SOLID Principles**

| **Principle** | **Idea** | **Essence** |
| --- | --- | --- |
| SRP | One class = One job | Keep code modular and focused |
| OCP | Extend, don’t rewrite | New behavior ≠ broken old behavior |
| LSP | Substitutable subclasses | Don’t change expectations |
| ISP | Use only what you need | Break big interfaces into small ones |
| DIP | Depend on interfaces | Decouple high and low-level logic |