Mastering SOLID Principles in Java: A Practical, Real-World Guide

SOLID is a five-part blueprint for writing maintainable, testable, and scalable object-oriented code. This guide demystifies each principle for Java developers, pairing crystal-clear definitions with everyday analogies, step-by-step refactorings, interview traps, and production-grade patterns. By the final page you will know when—and when **not**—to enforce each rule, how to spot violations in code reviews, and how to leverage popular Java frameworks (Spring, Jakarta EE, Micronaut) to stay SOLID without ceremony.

Overview: Why SOLID Still Matters in 2025

Software longevity hinges on **flexibility** (adapting to change) and **robustness** (not breaking under change)[^1]. Rigid classes ripple bugs across the codebase, while over-abstracted designs paralyze delivery. SOLID offers a balanced middle path, shrinking defect rates by up to 72% in longitudinal studies of enterprise codebases[^3].

The Principles at a Glance

Letter	Principle	One-Line Motto	Typical Java Artifact	Real-Life Analogy
S	Single Responsibility	“One reason to change.”	Class/method	Cashier vs Chef in a restaurant[^5]
O	Open–Closed	“Extend, don’t hack.”	Abstract class/interface	Smartphone apps added via store updates[^5]
L	Liskov Substitution	“Subtype must fit the socket.”	Inheritance hierarchy	Rental car upgrade—any car still drives with the same pedals[^8]
I	Interface Segregation	“Clients shouldn’t swallow super-sized menus.”	Role-focused interfaces	Separate remotes for TV and A/C instead of one chaotic mega-remote[^10]
D	Dependency Inversion	“Depend on contracts, not concretes.”	Dependency injection	Universal power outlet accepting any brand of charger[^12]

Single Responsibility Principle (SRP)

1. Essence

A class **must have exactly one reason to change**—it should encapsulate a single axis of variation[^1].

2. Real-World Analogy

Think of a café where the cashier also roasts beans and repairs the espresso machine. When the roaster breaks, coffee sales halt. Splitting duties—cashier, roaster, technician—isolates failures and streamlines training[^5].

3. SRP Violation in Java

```
class OrderService {
    void placeOrder(Order o) { /* domain logic */ }
    void generateInvoice(Order o) { /* finance */ }
    void sendEmail(Order o) { /* marketing */ }
}
```

One class juggles sales, finance, and marketing. Any policy tweak in one area risks regression in another[^15].

4. Refactoring to SRP

```
class OrderService { void placeOrder(Order o) { /*...*/ } }

class InvoiceService { void generateInvoice(Order o) { /*...*/ } }

class NotificationService { void sendEmail(Order o) { /*...*/ } }
```

Now each service evolves independently—tax rules change? Only `InvoiceService` changes[^16].

5. Implementation Tips

- Group SRP classes in cohesive packages (e.g., `com.store.billing`).
- Name classes on nouns reflecting their sole job: `TaxCalculator`, `PdfExporter`[^6].
- Use *package-private* access to expose only the single responsibility to callers.

6. Unit-Testing Payoff

Smaller classes slash test fixture overhead: mocking the email gateway is irrelevant when testing `InvoiceService`[^14].

7. Interview Cue

If asked to defend SRP, cite **change amplification**: with SRP, a spec change touches one class instead of a web of tangles[^6].

Open–Closed Principle (OCP)

1. Essence

Software entities should be **open for extension, closed for modification**[^7]. Add new behavior via polymorphism instead of editing stable code.

2. Real-World Analogy

Smartphones add features by downloading apps; the OS remains untouched[^5]. Similarly, a payroll engine adds a new bonus type by plugging in a strategy class, not rewriting `PayrollCalculator`.

3. OCP Violation: Type Switch Hell

```
class AreaCalculator {
   double area(Shape s) {
      if (s instanceof Square) { /*...*/ }
      else if (s instanceof Circle) { /*...*/ }
      // every new shape mutates this method
   }
}
```

Maintenance snowballs as shapes grow[^17].

4. OCP Refactor with Strategy

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

class Square implements Shape { /*...*/ }
class Circle implements Shape { /*...*/ }

class AreaCalculator {
   double area(Shape s) { return s.area(); }
}
```

Adding `Triangle` extends the system without touching `AreaCalculator`[^18].

5. Best Practices

- Favor composition over inheritance. Policies vary? Inject them.
- Template Method or Strategy patterns formalize OCP.
- Spring's `@Component` scanning lets you add new beans without altering consumer code.

6. Performance Watch

Over-polymorphism can sprinkle tiny objects and extra indirections. Profile hot paths before abstracting everything.

Liskov Substitution Principle (LSP)

1. Essence

Subtypes must be substitutable for their base types without altering observable behavior[^8].

2. Real-World Analogy

Car rental upgrades you from compact to SUV. Pedals, steering, indicators still behave predictably. If the SUV required joystick steering, it would violate LSP.

3. Classic Violation: Rectangle vs Square

```
class Rectangle { void setWidth(int w); void setHeight(int h); }
class Square extends Rectangle {
    void setWidth(int w) { super.setWidth(w); super.setHeight(w); }
    void setHeight(int h) { setWidth(h); } // unexpected side-effect
}
```

Client code relying on independent width/height breaks when handed a `Square`[^9].

4. Fix via Composition

```
interface Shape { int area(); }

class Rectangle implements Shape { /* width & height */ }
class Square implements Shape { /* side */ }
```

No inheritance, no surprise contracts, LSP preserved[^8].

5. Behavioral Checklist

For each overridden method:

- **Pre-conditions** cannot be strengthened.
- **Post-conditions** cannot be weakened.
- **Exceptions** must be same or narrower.

6. Detecting LSP Smells

- Subclass methods throwing `UnsupportedOperationException`[^8].
- Extensive `instanceof` checks hint at wrong abstraction.

Interface Segregation Principle (ISP)

1. Essence

No client should be forced to depend on methods it does not use[^21]. Split fat interfaces into role-specific ones.

2. Real-World Analogy

A universal remote randomly triggers coffee machines and vacuum cleaners—Alex's gadget nightmare[^10]. Users prefer one slim remote per appliance.

3. ISP Violation in Java

```
interface Machine { void print(); void scan(); void fax(); }

class OldPrinter implements Machine {
```

```
    public void print() { /* ok */ }
    public void scan() { throw new UnsupportedOperationException(); }
    public void fax()  { throw new UnsupportedOperationException(); }
}
```

Clients depending on `Machine` must handle useless fax/scan methods[^22].

4. ISP Refactor

```
interface Printer { void print(); }
interface Scanner { void scan(); }
interface Fax     { void fax(); }

class OldPrinter implements Printer { public void print() { /*...*/ } }
class ModernMFP implements Printer, Scanner, Fax { /*...*/ }
```

Each device implements only relevant contracts, boosting clarity[^23].

5. Implicit ISP with Java 8 Default Methods

Default interface methods allow gentle evolution without bloating original contracts, but resist piling on unrelated defaults[^17].

Dependency Inversion Principle (DIP)

1. Essence

1. High-level modules should not depend on low-level modules; **both depend on abstractions**.
2. Abstractions should not depend on details; **details depend on abstractions**[^13].

2. Real-World Analogy

Power sockets expose a standard interface; whether copper or aluminum wiring lies behind the wall is implementation detail[^25].

3. DIP Violation

```
class ShoppingCart {
    private CreditCardProcessor processor = new CreditCardProcessor(); // hard
dependency
    void checkout(Order o) { processor.pay(o); }
}
```

Switching to PayPal rewires `ShoppingCart`, breaking OCP and testing isolation[^26].

4. DIP-Compliant Design

```
interface PaymentProcessor { void pay(Order o); }

class CreditCardProcessor implements PaymentProcessor { /*...*/ }
class PayPalProcessor    implements PaymentProcessor { /*...*/ }

class ShoppingCart {
   private final PaymentProcessor processor;
   ShoppingCart(PaymentProcessor p) { this.processor = p; }
   void checkout(Order o) { processor.pay(o); }
}
```

`ShoppingCart` depends only on the abstraction; Spring injects the desired processor at runtime[^27].

5. Dependency Injection Modes

Mode	Java Example	Notes
Constructor	`new Service(dao)`	Immutable, test-friendly
Setter	`setDao(dao)`	Allows late wiring; watch for nulls
Interface	`implements Aware` in Spring	Framework-specific hooks

6. Testing Upside

Mocks/stubs plug into constructor, enabling fast unit tests without databases or gateways[^24].

Cross-Cutting Strategies to Stay SOLID

Design Patterns Cheat Sheet

Principle	Go-To Patterns	Framework Features
SRP	Facade, Singleton (stateless)	Spring `@Service`, `@Component`
OCP	Strategy, Decorator, Template Method	Spring Boot auto-configuration
LSP	Specification, State	Jakarta Bean Validation to enforce invariants
ISP	Adapter, Proxy	Java 17 sealed interfaces to constrain roles
DIP	Service Locator, Dependency Injection	CDI, Micronaut DI, Guice

Tooling for Detection

- **SonarQube** flags God classes (>20 methods) and circular dependencies.
- **ArchUnit** asserts layering rules (“domain must not access persistence”).
- **Jacoco + Mutation Testing** reveals tight coupling that hampers isolation.

Putting It Together: Case Study—Ride-Hailing App

1. Domain Slice

Component	SOLID Focus	Outcome
`RideService`	SRP	Only matches riders & drivers; billing delegated[^15]
`FarePolicy` hierarchy	OCP	Surge algorithm added without editing `RideService`[^29]
`Vehicle` vs `ElectricVehicle`	LSP	Charging status obeys vehicle contract; no `UnsupportedOperationException`[^20]
`Navigation` interface family	ISP	Separate `RoutePlanner`, `TrafficEstimator`—bikes skip highway logic[^21]
`PaymentGateway` abstraction	DIP	Switch from Stripe to local wallet via new bean—zero code change in `RideService`[^26]

2. Deployment Wins

- Feature toggles activate **festival surge** by merely adding a new `SurgePolicy` bean—zero redeploy downtime, thanks to OCP.
- Unit tests simulate wallet failures with Mockito, enabled by DIP.
- Micro-frontends consume lean interfaces; front-end team never touches fax methods because ISP trimmed them.

Common Pitfalls & Defensive Moves

Mistake	Violated Principle	Prevention
God Object with 40+ public methods	SRP & ISP	Refactor via Extract Class, Extract Interface[^30]
Subclass hides base pre-condition errors	LSP	Add JUnit behavioral contract tests[^8]
Over-engineering dozens of interfaces for trivial logic	OCP & DIP	Apply YAGNI—abstract only after proven volatility[^31]
Framework leakage (`EntityManager` injected everywhere)	SRP & DIP	Wrap ORM in repository abstraction[^24]
Parallel Stream on stateful lambda	LSP (behavioral)	Keep lambdas stateless; rely on Collector combiners[^17]

Interview & Certification Prep

Rapid-Fire Questions

1. *How does OCP interplay with microservices?*
    - Each microservice interface is closed to breaking changes but open for additive endpoints (via versioning)[^2].
2. *Why might you violate SRP intentionally?*
    - Premature splitting inflates classes; in prototype spikes, cohesion > purity[^14].

3. *Name a Java feature that eased ISP adoption in 2015.*
    - Default methods eliminated “binary breakage” fear when carving small interfaces[^17].
4. *How do Spring profiles relate to DIP?*
    - They swap concrete beans behind an interface for different environments without source edits[^27].

Practice coding the Rectangle-Square dilemma and payment processors; they recur in interviews[^18].

Advanced Topics

1. SOLID in Functional Java

While lambdas sidestep classes, SRP maps to single-purpose pure functions; DIP surfaces as higher-order functions accepting strategies.

2. JPMS (Java Platform Module System)

JPMS enforces package encapsulation, preventing low-level modules from leaking details, thus complementing DIP[^24].

3. Reactive Programming

Back-pressure flows (`Flux`, `Mono`) respect SRP by isolating *what* to emit from *how* to schedule. LSP applies to operator contracts—`flatMap` must not reorder elements unless documented.

Implementation Roadmap for Teams

1. **Audit** current code with SonarQube SRP/LSP rules.
2. **Refactor hotspots** (high churn + low coverage) first.
3. **Introduce DI container** (Spring Boot) to enforce DIP.
4. **Define coding standards**: class <300 lines, interface methods <6.
5. **Automate arch tests** with ArchUnit in CI.
6. **Run brown-bag sessions** replaying this guide’s examples.

Final Thoughts

Mastering SOLID is less about memorizing acronyms and more about **practicing mindful separation of concerns**. Treat each principle as a stress-test question: *“What will break if this requirement changes?”* When the honest answer is *“Only one class, and clients remain unaffected,”* your design is on stable ground. Embrace incremental refactoring, lean on Java’s powerful interface features, and let real-world change requests guide when to abstract. Your future self—and your teammates—will thank you for writing code that ages like fine wine rather than stale milk.

⁂

What is SOLID and Why Should You Care?

SOLID is an acronym for five design principles intended to make software designs more **understandable, flexible, and maintainable**. Think of them not as strict rules, but as proven guidelines that help you avoid building a "house of cards" that collapses the moment you try to change something.

Following SOLID leads to code that is:

- **Easier to read and understand.**
- **Easier to test.**
- **Easier to extend with new features.**
- **Less prone to bugs when you make changes.**

Let's dive into each principle.

---

1. S - Single Responsibility Principle (SRP)

The Core Idea

A class should have only one reason to change. This means a class should have only one primary job or responsibility.

Real-Life Analogy: The Restaurant Chef

Imagine a chef in a high-end restaurant.

- **Good Design (Follows SRP):** The chef's single responsibility is to **cook food**. He doesn't handle customer payments, he doesn't clean the tables, and he doesn't manage the restaurant's marketing. There are other specialists for those jobs (a cashier, a busboy, a marketing manager). If the restaurant's payment system changes (e.g., they get a new credit card machine), the chef's work is completely unaffected. He has only **one reason to change**: if the menu or a recipe changes.
- **Bad Design (Violates SRP):** The chef is also responsible for processing credit card payments. Now, this single "chef" entity has two responsibilities: cooking and payment processing. If the tax laws change, the `calculateTax()` part of his payment responsibility needs to be updated. If the credit card API changes, the `processPayment()` part needs to be updated. Now, you're modifying the "Chef" class for reasons that have nothing to do with cooking. This makes the class complex, fragile, and hard to test.

The Problem it Solves

Without SRP, your classes become "God Objects" that do everything. A change in one minor feature can cause a cascade of unexpected bugs in other unrelated features because they are all tangled together in the same class.

Java Code Example

Bad Code (Violating SRP):

```
// This class has TWO responsibilities: managing employee data AND calculating
pay.
```

```
// Reason to change #1: The employee data structure changes (e.g., add middle
name).
// Reason to change #2: The pay calculation logic changes (e.g., new tax laws).
class Employee {
    private String name;
    private double hourlyRate;
    private int hoursWorked;

    public Employee(String name, double hourlyRate, int hoursWorked) {
        this.name = name;
        this.hourlyRate = hourlyRate;
        this.hoursWorked = hoursWorked;
    }

    // Responsibility 1: Business Logic
    public double calculatePay() {
        return this.hourlyRate * this.hoursWorked;
    }

    // Responsibility 2: Data Persistence
    public void saveToDatabase() {
        // Complex logic to connect to a MySQL database and save this employee...
        System.out.println("Saving " + this.name + " to the database.");
    }
}
```

Good Code (Following SRP): We split the responsibilities into different classes.

```
// Class 1: Only responsible for holding employee data. A simple POJO.
// Its only reason to change is if the data fields change.
class EmployeeData {
    private String name;
    private double hourlyRate;
    // Getters and Setters...
}

// Class 2: Only responsible for calculating pay.
// Its only reason to change is if pay calculation rules change.
class PayCalculator {
    public double calculatePay(EmployeeData employee, int hoursWorked) {
        return employee.getHourlyRate() * hoursWorked;
    }
}

// Class 3: Only responsible for database interactions.
// Its only reason to change is if the database technology changes (e.g., MySQL ->
MongoDB).
class EmployeeRepository {
    public void save(EmployeeData employee) {
        // Logic to save the EmployeeData object to the database.
        System.out.println("Saving " + employee.getName() + " to the database.");
```

```
    }
}
```

---

2. O - Open/Closed Principle (OCP)

The Core Idea

Software entities (classes, modules, functions) should be **open for extension**, but **closed for modification**.

Real-Life Analogy: The Smartphone and Apps

Think of your smartphone.

- **Closed for Modification:** You cannot (and should not) open your phone's case and start soldering new components onto its main circuit board. The core operating system and hardware are a sealed, "closed" unit. Modifying them is risky and voids the warranty.
- **Open for Extension:** You can add new functionality to your phone at any time by installing new apps from the App Store. You are **extending** the phone's capabilities without changing its core nature.

The App Store is the interface that allows for this extension. The principle states that you should design your classes in a way that new functionality can be added without changing the existing, tested, and proven code.

The Problem it Solves

Without OCP, adding a new feature requires you to go back and change existing, working code. This is risky and can introduce bugs into features that were previously stable.

Java Code Example

Let's say we have a service that calculates the total area of different shapes.

Bad Code (Violating OCP): Every time we add a new shape, we have to **modify** the `AreaCalculator` class.

```
class Rectangle {
    public double width;
    public double height;
}

class Circle {
    public double radius;
}

// This class is NOT closed for modification.
class AreaCalculator {
    public double calculateTotalArea(Object[] shapes) {
        double totalArea = 0;
        for (Object shape : shapes) {
            if (shape instanceof Rectangle) {
                Rectangle rect = (Rectangle) shape;
```

```
                totalArea += rect.width * rect.height;
            }
            if (shape instanceof Circle) {
                Circle circle = (Circle) shape;
                totalArea += circle.radius * circle.radius * Math.PI;
            }
            // What if we add a Triangle? We have to come back here and add a new
"if" block!
            // This is modification, not extension.
        }
        return totalArea;
    }
}
```

Good Code (Following OCP): We use an interface (our "App Store") and polymorphism.

```
// Our "extension point" interface.
interface Shape {
    double getArea();
}

class Rectangle implements Shape {
    public double width;
    public double height;
    @Override
    public double getArea() { return width * height; }
}

class Circle implements Shape {
    public double radius;
    @Override
    public double getArea() { return radius * radius * Math.PI; }
}

// Now, if we want to add a new shape, we just create a new class.
// We NEVER have to touch the AreaCalculator.
class Triangle implements Shape {
    public double base;
    public double height;
    @Override
    public double getArea() { return 0.5 * base * height; }
}


// This class is now CLOSED for modification but OPEN for extension.
class AreaCalculator {
    public double calculateTotalArea(Shape[] shapes) {
        double totalArea = 0;
        for (Shape shape : shapes) {
            totalArea += shape.getArea(); // No more "if" or "instanceof"!
        }
        return totalArea;
```

```
    }
}
```

---

3. L - Liskov Substitution Principle (LSP)

The Core Idea

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In simple terms: if `S` is a subclass of `T`, then an object of type `T` should be replaceable by an object of type `S` without breaking the program.

Real-Life Analogy: The Remote Control

Imagine you have a standard TV remote control (`T`). It has `volumeUp()`, `volumeDown()`, and `changeChannel()` buttons. Now, you buy a fancy universal remote control (`S`) that is a *subclass* of the standard remote.

- **Follows LSP:** The universal remote's `volumeUp()` button increases the volume. Its `changeChannel()` button changes the channel. You can substitute the old remote with the new one, and the program (you, the user) still works correctly. The new remote might have *extra* buttons (like controlling the DVD player), but it correctly implements all the *original* behaviors.
- **Violates LSP:** The universal remote's `volumeUp()` button *mutes* the TV instead of increasing the volume. When you substitute the old remote with this new one, the program breaks. You expect the volume to go up, but something different and unexpected happens. The subclass does not behave like its parent, violating the "contract" of the superclass.

The Problem it Solves

Without LSP, you cannot reliably use polymorphism. You end up with `if (object instanceof SubClass)` checks everywhere to handle the "special" broken behavior of your subclasses, which completely defeats the purpose of inheritance and violates the Open/Closed Principle.

Java Code Example

The classic example is `Rectangle` and `Square`.

Bad Code (Violating LSP): A square *is a* rectangle, right? Let's try to model that with inheritance.

```
class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) { this.width = width; }
    public void setHeight(int height) { this.height = height; }

    public int getArea() { return this.width * this.height; }
}
```

```
// A Square must have equal width and height.
// To enforce this rule, we override the setters.
class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width); // A square's height must equal its width
    }

    @Override
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height); // A square's width must equal its height
    }
}

// A method that uses a Rectangle
class AreaVerifier {
    public static void verifyArea(Rectangle r) {
        r.setWidth(5);
        r.setHeight(4);

        // According to the Rectangle class contract, the area should be 5 * 4 =
20.
        if (r.getArea() != 20) {
            // This line will be triggered for a Square! Its area will be 16.
            System.out.println("Houston, we have a problem! Area is " +
r.getArea());
        }
    }
}

// Main program
public static void main(String[] args) {
    Rectangle rect = new Rectangle();
    Square sq = new Square();

    AreaVerifier.verifyArea(rect); // Works fine.
    AreaVerifier.verifyArea(sq);   // Breaks! The Square subclass is not
substitutable.
}
```

Conclusion: A `Square` in this model is not a valid substitute for a `Rectangle`, so this inheritance hierarchy is flawed, even though it seems logical at first. The solution is often to rethink the inheritance structure, perhaps by having a more generic `Shape` interface.

---

4. I - Interface Segregation Principle (ISP)

The Core Idea

Clients should not be forced to depend on interfaces they do not use. In essence, it's better to have many small, specific interfaces than one large, general-purpose one.

Real-Life Analogy: The Restaurant Menu

Imagine going to a restaurant.

- **Bad Design (Violates ISP):** You are handed a single, gigantic 50-page menu (a "fat interface") that contains breakfast, lunch, dinner, drinks, desserts, and the catering menu. If you're just there for a coffee, you are forced to deal with a massive interface that is 99% irrelevant to you.
- **Good Design (Follows ISP):** The restaurant has several smaller, specific menus (segregated interfaces): a Breakfast Menu, a Lunch Menu, a Dinner Menu, and a Drink Menu. When you arrive for coffee, you are handed only the Drink Menu. You are not forced to see or depend on the parts of the "restaurant interface" that you don't use.

The Problem it Solves

"Fat interfaces" lead to classes having to implement methods they don't need, often with empty or exception-throwing implementations. It also means that a change to an interface method that one client uses will force a re-compilation of *all* other clients, even those that don't use that method.

Java Code Example

Bad Code (Violating ISP): We have one "fat" interface for all types of workers.

```
// A "fat" worker interface
interface IWorker {
    void work();
    void eat(); // All workers must eat, right?
}

class HumanWorker implements IWorker {
    @Override
    public void work() { System.out.println("Human working..."); }
    @Override
    public void eat() { System.out.println("Human eating..."); }
}

// The problem: Robots don't eat!
// We are forced to implement a method that has no meaning for this class.
class RobotWorker implements IWorker {
    @Override
    public void work() { System.out.println("Robot working..."); }

    @Override
    public void eat() {
        // This makes no sense. We are forced to implement it.
        // We might leave it empty, or throw an exception. Both are bad.
        throw new UnsupportedOperationException("Robots don't eat!");
```

```
    }
}
```

Good Code (Following ISP): We segregate the interface into smaller, more logical roles.

```
// A small interface for just working
interface Workable {
    void work();
}

// Another small interface for things that eat
interface Feedable {
    void eat();
}

// Our Human implements both, because it makes sense for a human.
class HumanWorker implements Workable, Feedable {
    @Override
    public void work() { System.out.println("Human working..."); }
    @Override
    public void eat() { System.out.println("Human eating..."); }
}

// Our Robot only implements what it can do. It is not forced to depend on eat().
class RobotWorker implements Workable {
    @Override
    public void work() { System.out.println("Robot working..."); }
}
```

---

5. D - Dependency Inversion Principle (DIP)

The Core Idea

1. High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details. Details should depend on abstractions.

This sounds complex, but it's incredibly powerful.

Real-Life Analogy: The Wall Socket and the Lamp

Think about a lamp in your house.

- **High-Level Module:** The lamp (its job is to provide light).
- **Low-Level Module:** The electrical wiring inside your wall (it provides power).
- **Bad Design (Violates DIP):** Imagine if you had to directly solder the lamp's wires to the wires inside the wall. The high-level lamp would be directly and tightly coupled to the low-level wiring. If you

wanted to move the lamp, or if the internal wiring changed, you'd have to rewire everything.

- **Good Design (Follows DIP):** We have an **abstraction**: the **wall socket (an interface)**.
    - The lamp (high-level) does not depend on the specific wiring in the wall (low-level). It depends on the wall socket abstraction.
    - The wiring in the wall (low-level) does not depend on the specific lamp. It conforms to the wall socket abstraction.

This "inverts" the dependency. Instead of `Lamp -> Wiring`, we have `Lamp -> Socket <- Wiring`. The lamp can now be plugged into any socket, and any compatible appliance can be plugged into the wall. They are decoupled.

The Problem it Solves

Without DIP, your code becomes a rigid, tangled mess. Your high-level business logic becomes dependent on specific implementation details (like a specific database or a specific payment gateway). When those details need to change, you have to change your high-level logic, which is the most valuable and complex part of your system.

Java Code Example

Bad Code (Violating DIP): A high-level notification service directly depends on a low-level email client.

```
// Low-level module
class EmailClient {
    public void sendEmail(String message) {
        System.out.println("Sending email: " + message);
    }
}

// High-level module
// The NotificationService DIRECTLY depends on the concrete EmailClient class.
class NotificationService {
    private EmailClient emailClient; // Direct dependency!

    public NotificationService() {
        this.emailClient = new EmailClient(); // Tight coupling!
    }

    public void sendNotification(String message) {
        this.emailClient.sendEmail(message);
    }
}
// What if we want to send an SMS instead? We have to change the
NotificationService class!
```

Good Code (Following DIP): Both modules depend on an abstraction (an interface).

```java
// The Abstraction (our "wall socket")
interface MessageClient {
    void sendMessage(String message);
}

// Low-level module 1
class EmailClient implements MessageClient {
    @Override
    public void sendMessage(String message) {
        System.out.println("Sending email: " + message);
    }
}

// Low-level module 2 (easy to add!)
class SmsClient implements MessageClient {
    @Override
    public void sendMessage(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

// High-level module
// Now it depends on the ABSTRACTION, not a concrete detail.
class NotificationService {
    private MessageClient messageClient; // Dependency is on the interface!

    // The dependency is "injected" from the outside.
    public NotificationService(MessageClient messageClient) {
        this.messageClient = messageClient;
    }

    public void sendNotification(String message) {
        this.messageClient.sendMessage(message);
    }
}

// In our main application, we choose the implementation.
public static void main(String[] args) {
    // We can easily switch the dependency without touching NotificationService.
    MessageClient email = new EmailClient();
    MessageClient sms = new SmsClient();

    NotificationService emailNotifier = new NotificationService(email);
    emailNotifier.sendNotification("Hello via Email!"); // Sends an email

    NotificationService smsNotifier = new NotificationService(sms);
    smsNotifier.sendNotification("Hello via SMS!"); // Sends an SMS
}
```

Summary Table

Principle	Acronym	Core Idea	Real-Life Analogy
Single Responsibility	SRP	A class should have only one reason to change.	A Chef's job is to cook, not to process payments.
Open/Closed	OCP	Open for extension, closed for modification.	A Smartphone's core OS is closed, but you can extend it with Apps.
Liskov Substitution	LSP	Subclasses must be substitutable for their base classes.	A universal remote must correctly perform all functions of the original TV remote.
Interface Segregation	ISP	Don't force clients to implement interfaces they don't use.	A restaurant should have separate Drink, Breakfast, and Dinner menus.
Dependency Inversion	DIP	Depend on abstractions, not on concretions.	A Lamp and Wall Wiring both depend on the Wall Socket interface.