Interface Segregation Principle (ISP).

Clients should not be forced to depend on interfaces they do not use. Instead of one fat interface with many unrelated methods, we split it into smaller, specific interfaces. This keeps classes clean and focused.

Bad Example:

Here we have a big interface that forces all classes to implement methods they don't need.

interface Worker { void work(); void eat(); }

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

@Override public void eat() { System.out.println("Human is eating..."); } }

class RobotWorker implements Worker { @Override public void work() { System.out.println("Robot is working..."); }

// Robot doesn't eat, but still forced to implement this! @Override public void eat() { throw new UnsupportedOperationException("Robots don't eat!"); } }

Problems: RobotWorker is forced to depend on eat(), which doesn't make sense. This makes the design fragile and misleading.

Good example:

We split the big interface into smaller, role-specific interfaces.

interface Workable { void work(); }

interface Eatable { void eat(); }

class HumanWorker implements Workable, Eatable { @Override public void work() { System.out.println("Human is working..."); }

@Override public void eat() { System.out.println("Human is eating..."); } }

class RobotWorker implements Workable { @Override public void work() { System.out.println("Robot is working..."); } }

Benefits: HumanWorker implements both Workable and Eatable. RobotWorker implements only what it needs (Workable). Classes depend only on relevant behavior.

Key Benefit of ISP: Avoids fat interfaces. Prevents classes from having dummy/unsupported methods. Makes system easier to extend and maintain.

Another Example:

Let's explore how the Interface Segregation Principle can be implemented using an example. We'll use a shape-related scenario where we have both two-dimensional and three-dimensional shapes.

Step 1: Define Specific Interfaces Instead of having a single interface that includes both area and volume calculations, we define two separate interfaces:

// Interface for two-dimensional shapes interface ShapeInterface { double area(); }

// Interface for three-dimensional shapes interface ThreeDimensionalShapeInterface { double volume(); }

By creating these distinct interfaces, we ensure that each interface serves a specific purpose. This means that classes do not have to implement methods that are irrelevant to their intended functionality, thus adhering to the core tenet of ISP.

Step 2: Implementing Interfaces in Classes Now we can create classes that implement these interfaces based on their functionality:

// Class for a square (2D shape) class Square implements ShapeInterface { private double side;

public Square(double side) { this.side = side; }

@Override public double area() { return side * side; // Calculate area of the square } }

// Class for a cuboid (3D shape) class Cuboid implements ShapeInterface, ThreeDimensionalShapeInterface { private double length; private double width; private double height;

public Cuboid(double length, double width, double height) { this.length = length; this.width = width; this.height = height; }

@Override public double area() { return 2 * (length * width + length * height + width * height); // Calculate surface area }

@Override public double volume() { return length * width * height; // Calculate volume } }

Square Class: Implements ShapeInterface only. The Square class focuses on calculating the area and does not implement any volume method because it does not need it. This design aligns perfectly with ISP, as it avoids forcing the Square class to adopt unnecessary methods that do not apply to it. Cuboid Class: Implements both ShapeInterface and ThreeDimensionalShapeInterface. The Cuboid class is designed to handle both area and volume calculations. By implementing both interfaces, it demonstrates the ability to adopt methods relevant to its functionality without being forced to implement any unrelated methods.

Step 3: Managing Shapes To manage these shapes efficiently, we can create a separate interface for shape management: // Interface for managing shapes interface ManageShapeInterface { double calculate(); // Common method for calculations }

// Square class implementing the ManageShapeInterface class Square implements ShapeInterface, ManageShapeInterface { // (Constructor and area method remain unchanged)

@Override public double calculate() { return this.area(); // Return area calculation } }

// Cuboid class implementing the ManageShapeInterface class Cuboid implements ShapeInterface, ThreeDimensionalShapeInterface, ManageShapeInterface { // (Constructor, area and volume methods remain unchanged)

@Override public double calculate() { return this.area(); // Return area calculation } }

ManageShapeInterface: This interface introduces a common method (calculate()) that both the Square and Cuboid classes implement. By unifying the method under ManageShapeInterface, the design promotes flexibility and consistency in how shapes are managed, while still adhering to ISP. Implementation in Classes: Both Square and Cuboid implement the ManageShapeInterface, which means they can be treated interchangeably when calculating areas. This allows for polymorphism and encapsulation of functionality.

Step 4: Using the Interfaces Now we can create a method that uses the ManageShapeInterface to handle any shape, regardless of whether it's two-dimensional or three-dimensional: public class AreaCalculator { public static void main(String[] args) { ManageShapeInterface square = new Square(4); ManageShapeInterface cuboid = new Cuboid(3, 4, 5);

System.out.println("Square Area: " + square.calculate()); System.out.println("Cuboid Area: " + cuboid.calculate()); } }

Managing Instances: In the AreaCalculator class, we instantiate both Square and Cuboid as ManageShapeInterface types. This demonstrates how ISP enables us to work with different shapes through a unified interface while ensuring that each shape class only implements the relevant methods.

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

In simple terms: Don't hardcode dependencies. Depend on interfaces/abstractions, not concrete implementations. Makes code flexible, testable, and extensible.

Bad Example:

Initial Implementation (Violating DIP) In the initial implementation, the PasswordReminder class directly depends on the MySQLConnection class, which is a low-level module. class MySQLConnection { public String connect() { // handle the database connection return "Database connection"; } }

class PasswordReminder { private MySQLConnection dbConnection;

public PasswordReminder(MySQLConnection dbConnection) { this.dbConnection = dbConnection; } }

Problems: In this setup: The PasswordReminder class is tightly coupled to the MySQLConnection class, making it difficult to change the database engine without modifying the PasswordReminder class itself.

Good Example (follows DIP)

To adhere to the Dependency Inversion Principle, we introduce an interface that both high-level and low-level modules will depend on. // Define an interface for database connection interface DBConnectionInterface { String connect(); }

// Implement the MySQLConnection class class MySQLConnection implements DBConnectionInterface { @Override public String connect() { // handle the database connection return "Database connection"; } }

class PostgreSQLConnection implements DBConnectionInterface { @Override public String connect() { // handle PostgreSQL database connection return "PostgreSQL connection"; } }

// Implement the PasswordReminder class class PasswordReminder { private DBConnectionInterface dbConnection;

public PasswordReminder(DBConnectionInterface dbConnection) { this.dbConnection = dbConnection; } }

Benefits of DIP

High-level modules are independent of low-level details. Easy to swap implementations (e.g., MySQL → MongoDB). Improves testability (you can inject a mock database). Encourages clean layered architecture.

DRY (Don't Repeat Yourself) The DRY (Don't Repeat Yourself) principle is a fundamental concept in software development that emphasizes the importance of reducing code duplication. By abstracting common functionalities into reusable components, DRY promotes code reusability and maintainability, and reduces the likelihood of inconsistencies and bugs caused by redundant code.

Example for DRY (Don't Repeat Yourself) Imagine you are developing a web application where multiple forms require similar data validation. Instead of writ-

ing separate validation logic for each form, you can create a reusable validation function or class.

Let's say you have two forms: a registration form and a login form. Both forms need to validate the email and password fields. Without DRY: class RegistrationForm { public boolean validate(String email, String password) { if (email == null || email.isEmpty() || !email.contains("@")) { return false; } if (password == null || password.length() < 6) { return false; } return true; } }

class LoginForm { public boolean validate(String email, String password) { if (email == null || email.isEmpty() || !email.contains("@")) { return false; } if (password == null || password.length() < 6) { return false; } return true; } }

With DRY: class Validator { public boolean validateEmail(String email) { return email != null && !email.isEmpty() && email.contains("@"); }

public boolean validatePassword(String password) { return password != null && password.length() >= 6; } }

class RegistrationForm { private Validator validator = new Validator();

public boolean validate(String email, String password) { return validator.validateEmail(email) && validator.validatePassword(password); } }

class LoginForm { private Validator validator = new Validator();

public boolean validate(String email, String password) { return validator.validateEmail(email) && validator.validatePassword(password); } }

In the DRY version, we created a Validator class to handle the common validation logic. This ensures that any changes to the validation logic only need to be made in one place

---

KISS (Keep It Simple, Stupid) The KISS (Keep It Simple, Stupid) principle advocates for simplicity in software design. Simple solutions are easier to understand, maintain, and extend than complex ones. KISS encourages avoiding unnecessary complexity, convoluted code structures, or over-engineering. Example for KISS (Keep It Simple, Stupid) Consider a scenario where you need to implement a function to calculate the average of a list of numbers. Complex Approach: class ComplexAverageCalculator { public double calculateAverage(List<Integer> numbers) { int sum = 0; for (int number : numbers) { sum += number; } return sum / (double) numbers.size(); } }

Simple Approach (KISS): class SimpleAverageCalculator { public double calculateAverage(List<Integer> numbers) { return numbers.stream().mapToInt(Integer::intValue).average().orElse(
} }

In the KISS version, we use Java's Stream API to simplify the calculation of the average. This approach is more concise and easier to understand compared

to the loop-based method.