4. I - Interface Segregation Principle (ISP)

Resources

- Video → 1. SOLID Principles with Easy Examples (Hindi) | OOPs SOLID Principles - Low Level Design

What is the Interface Segregation Principle (ISP)?

> *Interface Segregation Principle (ISP) states that **“clients should not be forced to depend on interfaces they don't use. Instead of having one large interface with many methods, it's better to have multiple smaller, focused interfaces”.***

That means, Interfaces should be such that the client should NOT implement unnecessary functions they do not need.

Code Example: Violating ISP

```java
// BAD: This class violates ISP
// This is a fat interface
// One large interface forcing all implementers to define unused methods
public interface RestaurantEmployee {

    void prepareFood();

    void decideMenu();

    void serveFoodAndDrinks();

    void takeOrder();

    void cleanTheKitchen();
}


// BAD: This class violates ISP(clients shouldn't depend on unused interfaces)
// Bloated class with empty or error-throwing methods
// This Waiter is forced to implement methods it doesn't need
public class Waiter implements RestaurantEmployee {
    @Override
    public void takeOrder() {
        System.out.println("Taking order...");
    }

    @Override
    public void serveFoodAndDrinks() {
        System.out.println("Serving food and drinks...");
    }

    @Override
    public void cleanTheKitchen() {
        // Forced to implement but doesn't make sense for a waiter
        throw new AssertionError("Detail Message: Waiter cannot clean the kitchen!");
    }

    @Override
    public void prepareFood() {
        // Forced to implement but doesn't make sense for a waiter
        throw new AssertionError("Detail Message: Waiter cannot prepare food!");
    }

    @Override
    public void decideMenu() {
        // Forced to implement but doesn't make sense for a waiter
        throw new AssertionError("Detail Message: Waiter cannot decide the menu!");
    }

}


// Usage example - showing the problem
public class ViolationDemo {
    public static void main(String[] args) {
        Waiter waiter = new Waiter();
        // Works fine
        waiter.takeOrder();
        waiter.serveFoodAndDrinks();

        // These will throw exceptions
        waiter.prepareFood(); // forced implementation
        waiter.decideMenu(); // forced implementation
        waiter.cleanTheKitchen(); // forced implementation
    }
}
```

Problems with the Above Code

- Classes are forced to implement methods they don't support.
- Results in `AssertionError` being thrown.
- Code becomes bloated with empty or error-throwing methods.
- Violates the principle that clients shouldn't depend on unused interfaces.

Code Example: Follows ISP

```java
// GOOD: This follows ISP - Multiple focused interfaces following ISP
public interface ChefTasks {
    void prepareFood();

    void decideMenu();
}

// GOOD: This follows ISP - Multiple focused interfaces following ISP
public interface WaiterTasks {
    void serveFoodAndDrinks();

    void takeOrder();
}

// GOOD: This follows ISP - Multiple focused interfaces following ISP
public interface MaintenanceTasks {
    void cleanTheKitchen();

    void reStockGroceries();
}

// GOOD: This class follows ISP
// Now classes only implement what they actually need - Clean implementations
public class Chef implements ChefTasks {

    @Override
    public void prepareFood() {
        System.out.println("Preparing food...");
    }

    @Override
    public void decideMenu() {
        System.out.println("Deciding menu...");
    }
}

// GOOD: This class follows ISP
// Now classes only implement what they actually need - Clean implementations
public class Waiter implements WaiterTasks {
    @Override
    public void serveFoodAndDrinks() {
        System.out.println("Serving food and drinks...");
    }

    @Override
    public void takeOrder() {
        System.out.println("Taking order...");
    }
}

// Usage example - Following ISP
public class SolutionDemo {
    public static void main(String[] args) {
        // Create the objects
        // Now classes only implement what they actually support
        Chef chef = new Chef();
        Waiter waiter = new Waiter();

        // Use the objects
        // These work perfectly - no forced implementations
        chef.prepareFood();
        chef.decideMenu();
        // These work perfectly - no forced implementations
        waiter.takeOrder();
        waiter.serveFoodAndDrinks();
    }
}
```

Key Benefits of the Refactored Code

- Prevents bloated classes by having each class implement interfaces that it uses.
- No forced dependencies on irrelevant functionality.
- Cleaner, more maintainable design/code.
- Different Restaurant Employee entities can implement multiple interfaces only when they support those features.

Summary

The **Interface Segregation Principle** (ISP) helps maintain focused classes by ensuring they only implement the methods they need. To identify potential violations of the ISP, look for indicators such as low cohesion, large or “fat” interfaces, empty methods, and challenging testing scenarios. By recognizing these signs early, you can refactor your code to uphold a modular and flexible design.