**Interface Segregation Principle**

The [Interface Segregation Principle](https://www.oodesign.com/interface-segregation-principle.html) is the fourth SOLID design principle represented by the letter “I” in the acronym. It was Robert C Martin who first defined the principle by stating that “clients should not be forced to depend on methods they don’t use.” By clients, he means classes that implement interfaces. In other words, interfaces shouldn’t include too many functionalities.

The violation of Interface Segregation Principle harms code readability and forces programmers to write dummy methods that do nothing. In a well-designed application, you should avoid interface pollution (also called fat interfaces). The solution is to create smaller interfaces that you can implement more flexibly.

**Example of the Interface Segregation Principle**

Let’s add some user actions to our online bookstore so that customers can interact with the content before making a purchase. To do so, we create an interface called BookAction with three methods: seeReviews(), searchSecondHand(), and listenSample().

public interface BookAction {

void seeReviews();

void searchSecondhand();

void listenSample();

}

Then, we create two classes: HardcoverUI and an AudiobookUI that implement the BookAction interface with their own functionalities:

class HardcoverUI implements BookAction {

@Override

public void seeReviews() {...}

@Override

public void searchSecondhand() {...}

@Override

public void listenSample() {...}

}

class AudiobookUI implements BookAction {

@Override

public void seeReviews() {...}

@Override

public void searchSecondhand() {...}

@Override

public void listenSample() {...}

}

Both classes depend on methods they don’t use, so we have broken the Interface Segregation Principle. Hardcover books can’t be listened to, so the HardcoverUI class doesn’t need the listenSample() method. Similarly, audiobooks don’t have second-hand copies, so the AudiobookUI class doesn’t need it, either.

However, as the BookAction interface include these methods, all of its dependent classes have to implement them. In other words, BookAction is a polluted interface that we need to segregate. Let’s extend it with two more specific sub-interfaces: HardcoverAction and AudioAction.

public interface BookAction {

void seeReviews();

}

public interface HardcoverAction extends BookAction {

void searchSecondhand();

}

public interface AudioAction extends BookAction {

void listenSample();

}

Now, the HardcoverUI class can implement the HardcoverAction interface and the AudiobookUI class can implement the AudioAction interface.

This way, both classes can implement the seeReviews() method of the BookAction super-interface. However, HardcoverUI doesn’t have to implement the irrelevant listenSample() method and AudioUI doesn’t have to implement searchSecondhand(), either.

class HardcoverUI implements HardcoverAction {

@Override

public void seeReviews() {...}

@Override

public void searchSecondhand() {...}

}

class AudiobookUI implements AudioAction {

@Override

public void seeReviews() {...}

@Override

public void listenSample() {...}

}

The refactored code follows the Interface Segregation Principle, as neither classes depend on methods they don’t use. The UML diagram below excellently shows that the segregated interfaces lead to simpler classes that only implement the methods they really need:

