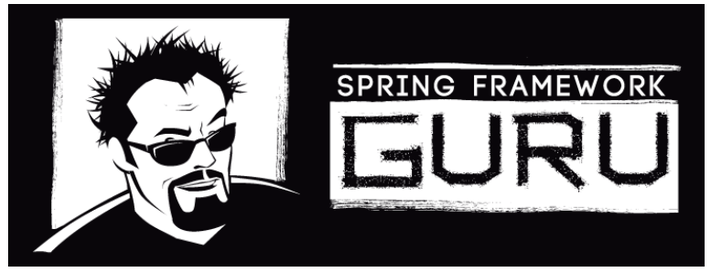


Interface Segregation Principle

Interfaces form a core part of the Java programming language and they are extensively used in enterprise applications to achieve abstraction and to support multiple inheritance of type- the ability of a class to implement more than one interfaces. From coding perspective, writing an interface is simple. You use the **interface** keyword to create an interface and declare methods in it. Other classes can use that interface with the **implements** keyword, and then provide implementations of the declared methods. As a Java programmer, you must have written a large number of interfaces, but the critical question is- have you written them while keeping design principles in mind? A design principle to follow while writing interfaces is the Interface Segregation Principle.

The Interface Segregation Principle represents the “I” of the five SOLID principles of object-oriented programming to write well-designed code that are more readable, maintainable, and easier to upgrade and modify. This principle was first used by Robert C. Martin while consulting for Xerox, which he mentioned in his 2002 book, Agile Software Development: Principles, Patterns and Practices. This principle states that “Clients should not be forced to depend on methods that they do not use”. Here, the term “Clients” refers to the implementing classes of an interface.

What the Interface Segregation Principle says is that your interface should not be bloated with methods that implementing classes don’t require. For such interfaces, also called “fat interfaces”, implementing classes are unnecessarily forced to provide implementations (dummy/empty) even for those methods that they don’t need. In addition, the implementing classes are subject to change when the interface changes. An addition of a method or change to a method signature requires modifying all the implementation classes even if some of them don’t use the method.

The Interface Segregation Principle advocates segregating a “fat interface” into smaller and highly cohesive interfaces, known as “role interfaces”. Each “role interface” declares one or more methods for a specific behavior. Thus clients, instead of implementing a “fat interface”, can implement only those “role interfaces” whose methods are relevant to them.

Interface Segregation Principle Violation (Bad Example)

Consider the requirements of an application that builds different types of toys. Each toy will have a price and color. Some toys, such as a toy car or toy train can additionally move, while some toys, such as a toy plane can both move and fly. An interface to define the behaviors of toys is this.

Toy.java

```
public interface Toy {
    void setPrice(double price);
    void setColor(String color);
    void move();
    void fly();
}
```

A class that represents a toy plane can implement the `Toy` interface and provide implementations of all the interface methods. But, imagine a class that represents a toy house. This is how the `ToyHouse` class will look.

ToyHouse.java

```
public class ToyHouse implements Toy {
    double price;
    String color;
    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
```

```
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){}
    @Override
    public void fly(){}
}
```

As you can see in the code, `ToyHouse` needs to provide implementation of the `move()` and `fly()` methods, even though it does not require them. This is a violation of the Interface Segregation Principle. Such violations affect code readability and confuse programmers. Imagine that you are writing the `ToyHouse` class and the intellisense feature of your IDE pops up the `fly()` method for auto complete. Not exactly the behavior you want for a toy house, is it?

Violation of the Interface Segregation Principle also leads to violation of the complementary Open Closed Principle. As an example, consider that the `Toy` interface is modified to include a `walk()` method to accommodate toy robots. As a result, you now need to modify all existing Toy implementation classes to include a `walk()` method even if the toys don't walk. In fact, the Toy implementation classes will never be closed for modifications, which will lead to a fragile application that is difficult and expensive to maintain.

Following the Interface Segregation Principle

By following the Interface Segregation Principle, you can address the main problem of the toy building application- *The `Toy` interface forces clients (implementation classes) to depend on methods that they do not use*.

The solution is- *Segregate the `Toy` interface into multiple role interfaces each for a specific behavior*. Let's segregate the `Toy` interface, so that our application now have three interfaces: `Toy`, `Movable`, and `Flyable`.

Toy.java

```java
package guru.springframework.blog.interfacesegregationprinciple;

public interface Toy {
     void setPrice(double price);
     void setColor(String color);
}
```

Movable.java

```java
package guru.springframework.blog.interfacesegregationprinciple;

public interface Movable {
    void move();
}
```

Flyable.java

```java
package guru.springframework.blog.interfacesegregationprinciple;

public interface Flyable {
    void fly();
}
```

In the examples above, we first wrote the `Toy` interface with the `setPrice()` and `setColor()` methods. As all toys will have a price and color, all Toy implementation classes can implement this interface. Then, we wrote the `Movable` and `Flyable` interfaces to represent moving and flying behaviors in toys. Let's write the implementation classes.

ToyHouse.java

```java
package guru.springframework.blog.interfacesegregationprinciple;

public class ToyHouse implements Toy {
    double price;
    String color;

    @Override
    public void setPrice(double price) {

        this.price = price;
    }
    @Override
    public void setColor(String color) {

        this.color=color;
    }
    @Override
    public String toString(){
        return "ToyHouse: Toy house- Price: "+price+" Color: "+color;
    }
}
```

ToyCar.java

```java
package guru.springframework.blog.interfacesegregationprinciple;

public class ToyCar implements Toy, Movable {
    double price;
    String color;

    @Override
    public void setPrice(double price) {

        this.price = price;
    }

    @Override
```

```
    public void setColor(String color) {
     this.color=color;
    }
    @Override
    public void move(){
        System.out.println("ToyCar: Start moving car.");
    }
    @Override
    public String toString(){
        return "ToyCar: Moveable Toy car- Price: "+price+" Color: "+color;
    }
}
```

ToyPlane.java

```
package guru.springframework.blog.interfacesegregationprinciple;

public class ToyPlane implements Toy, Movable, Flyable {
    double price;
    String color;

    @Override
    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){

        System.out.println("ToyPlane: Start moving plane.");
    }
    @Override
    public void fly(){

        System.out.println("ToyPlane: Start flying plane.");
```

```
    }
    @Override
    public String toString(){
        return "ToyPlane: Moveable and flyable toy plane- Price: "+price+" Color: "+color;
    }
}
```

As you can see, the implementation classes now implement only those interfaces they are interested in. Our classes do not have unnecessary code clutters, are more readable, and lesser prone to modifications due to changes in interface methods.

Next, let's write a class to create objects of the implementation classes.

ToyBuilder.java

```
package guru.springframework.blog.interfacesegregationprinciple;

public class ToyBuilder {
    public static ToyHouse buildToyHouse(){
        ToyHouse toyHouse=new ToyHouse();
        toyHouse.setPrice(15.00);
        toyHouse.setColor("green");
        return toyHouse;
        }
    public static ToyCar buildToyCar(){
        ToyCar toyCar=new ToyCar();
        toyCar.setPrice(25.00);
        toyCar.setColor("red");
        toyCar.move();
        return toyCar;
    }
    public static ToyPlane buildToyPlane(){
        ToyPlane toyPlane=new ToyPlane();
        toyPlane.setPrice(125.00);
        toyPlane.setColor("white");
        toyPlane.move();
        toyPlane.fly();
        return toyPlane;
    }
}
```

In the code example above, we wrote the `ToyBuilder` class with three static methods to create objects of the `ToyHouse`, `ToyCar`, and `ToyPlane` classes. Finally, let's write this unit test to test our example.

ToyBuilderTest.java

```
1  package guru.springframework.blog.interfacesegregationprinciple;
2
3  import org.junit.Test;
4
5  public class ToyBuilderTest {
6
7      @Test
8      public void testBuildToyHouse() throws Exception {
9      ToyHouse toyHouse=ToyBuilder.buildToyHouse();
10     System.out.println(toyHouse);
11     }
12
13     @Test
14     public void testBuildToyCar() throws Exception {
15     ToyCar toyCar=ToyBuilder.buildToyCar();;
16         System.out.println(toyCar);
17     }
18
19     @Test
20     public void testBuildToyPlane() throws Exception {
21     ToyPlane toyPlane=ToyBuilder.buildToyPlane();
22         System.out.println(toyPlane);
23     }
24 }
```

The output is:

```
1    .   ____          _            __ _ _
2   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
3  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
4   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
5    '  |____| .__|_| |_|_| |_\__, | / / / /
6   =========|_|==============|___/=/_/_/_/
7   :: Spring Boot ::        (v1.2.3.RELEASE)
```

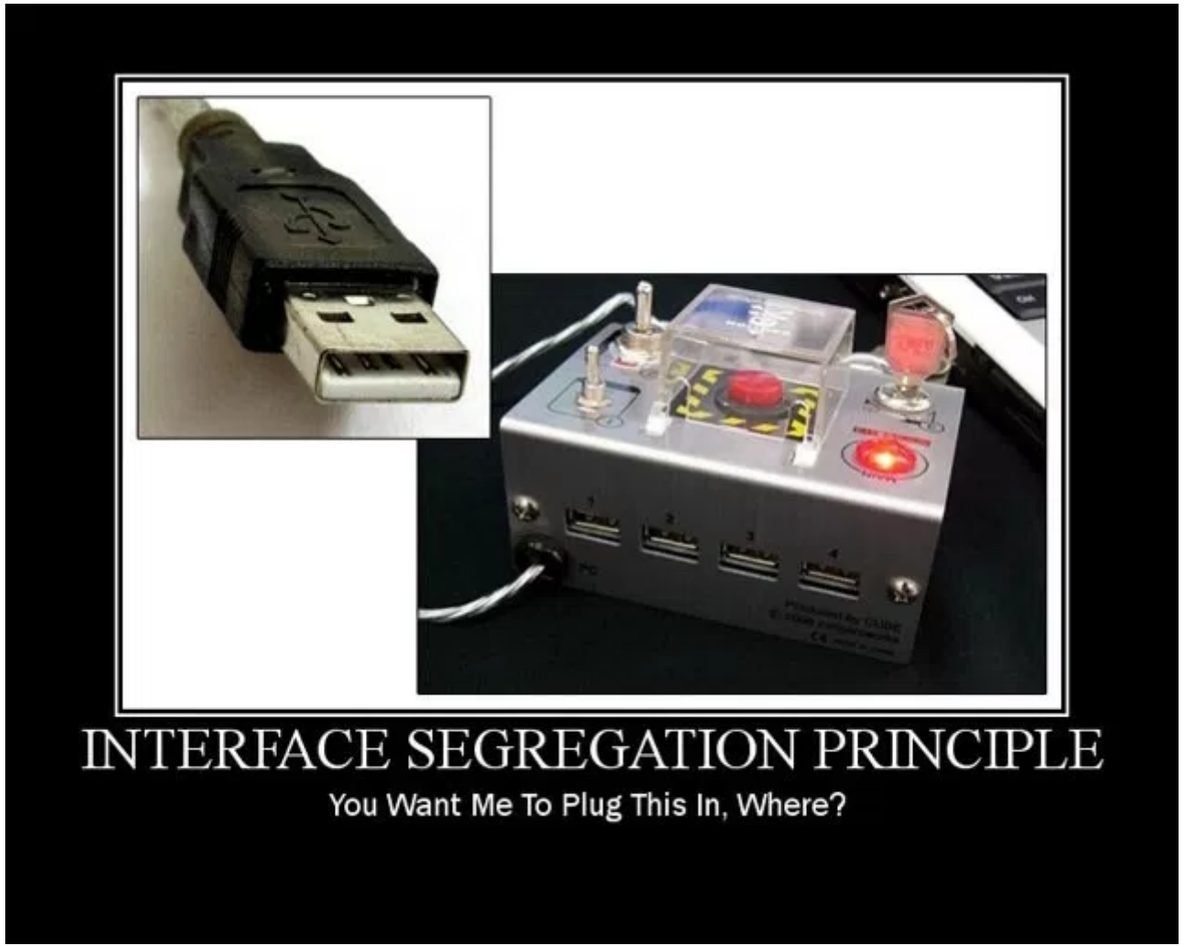
```
Running guru.springframework.blog.interfacesegregationprinciple.ToyBuilderTest
ToyHouse: Toy house- Price: 15.0 Color: green
ToyPlane: Start moving plane.
ToyPlane: Start flying plane.
ToyPlane: Moveable and flyable toy plane- Price: 125.0 Color: white
ToyCar: Start moving car.
ToyCar: Moveable Toy car- Price: 25.0 Color: red
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec - in guru.springframework.blog.interfacesegregationprinciple.ToyBuilderTest
```

Summary of Interface Segregation Principle

Both the Interface Segregation Principle and Single Responsibility Principle have the same goal: ensuring small, focused, and highly cohesive software components. The difference is that Single Responsibility Principle is concerned with classes, while Interface Segregation Principle is concerned with interfaces.
Interface Segregation Principle is easy to understand and simple to follow. But, identifying the distinct interfaces can sometimes be a challenge as careful considerations are required to avoid proliferation of interfaces. Therefore, while writing an interface, consider the possibility of implementation classes having different sets of behaviors, and if so, segregate the interface into multiple interfaces, each having a specific role.

Interface Segregation Principle in the Spring Framework

A number of times on this blog I've written about programming for Dependency Injection when programming using the Spring Framework. The Interface Segregation Principle becomes especially important when doing Enterprise Application Development with the Spring Framework.



As the size and scope of the application you're building grows, you are going to need pluggable components. Even when just for unit testing your classes, the Interface Segregation Principle has a role. If you're testing a class which you've written for dependency injection, as I've written before, it is ideal that you write to an interface. By designing your classes to use dependency injection against an interface, any class implementing the specified interface can be injected into your class. In testing your classes, you may wish to inject a mock object to fulfill the needs of your unit test. But when the class you wrote is running in production, the Spring Framework would inject the real full featured implementation of the interface into your class.

The Interface Segregation Principle and Dependency Injection are two very powerful concepts to master when developing enterprise class applications using the Spring Framework.

---

4 comments on “Interface Segregation Principle”

Mikiyas

March 30, 2017 at 1:52 am # Reply

Thank you so much!!

Ivan

May 27, 2017 at 3:08 pm # Reply

Thanks for great and simple explanation!!!

Sundar Rajagopal

August 4, 2017 at 7:21 am # Reply

Thanks much for the simple and very clear eXplanation on ‘Interface Segregation’

ZE QIN

October 22, 2017 at 12:03 pm # Reply

Thank you
