

# **SEMINAR PAPER**

In Computer Science

## **Interface segregation principle**

By: Peter Oettl

Student Number: 1910257146

Supervisor: DI Wolfgang Berger

Wien, May 25, 2021

# Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>SOLID principles</b>                            | <b>1</b> |
| 1.1      | S - Single Responsibility Principle . . . . .      | 1        |
| 1.2      | O - Open Close Principle . . . . .                 | 1        |
| 1.3      | L - Liskov Substitution Principle . . . . .        | 1        |
| 1.4      | I - Interface Segregation Principle . . . . .      | 1        |
| 1.5      | D - Dependency Inversion Principle . . . . .       | 2        |
| <b>2</b> | <b>Interface segregation principle</b>             | <b>2</b> |
| 2.1      | Interface . . . . .                                | 2        |
| 2.2      | ISP . . . . .                                      | 2        |
| 2.3      | Problem with will be solved with the ISP . . . . . | 3        |
| <b>3</b> | <b>Practical Example</b>                           | <b>4</b> |
|          | <b>Bibliography</b>                                | <b>6</b> |
|          | <b>List of Figures</b>                             | <b>7</b> |
|          | <b>List of source codes</b>                        | <b>8</b> |

# 1 SOLID principles

These principles describe practices for a software development with considerations for maintaining and extending as the project grows. Adopting these practices can also contribute to avoiding code smells, refactoring code, and Agile or Adaptive software development. [1]

## 1.1 S - Single Responsibility Principle

The single responsibility principle states that a class should have only a single reason to change throughout its lifetime. This principle ensures that a class exists only for a single reason but can have multiple methods to carry out distinct functions.[2]

## 1.2 O - Open Close Principle

This principle states that objects should be open for extension but closed for modification. The open and close principle ensures our code is easily extensible without editing or rewriting the existing codebase. Designing software applications using this principle ensures extensibility and the reusability of objects.[2]

## 1.3 L - Liskov Substitution Principle

This principle illustrates that if a section of your code is extending a superclass, then all subclasses of the superclass should be able to replace the superclass in your code. A section of your code does not have to know which class it is as far as it is a superclass subclass. The application of this principle ensures the easy integration of classes.[2]

## 1.4 I - Interface Segregation Principle

The I in SOLID stands for interface segregation, and it simply means that larger interfaces should be split into smaller ones. This ensure that implementing classes only need to be concerned about the methods that are of interest and necessary. [3]

## 1.5 D - Dependency Inversion Principle

The dependency inversion principle ensures that classes should not depend on solid classes but should only depend on abstraction. By following this approach, it makes the interchanging of modules and classes or services simple. The dependency inversion principle makes changing dependencies easier.[2]

## 2 Interface segregation principle

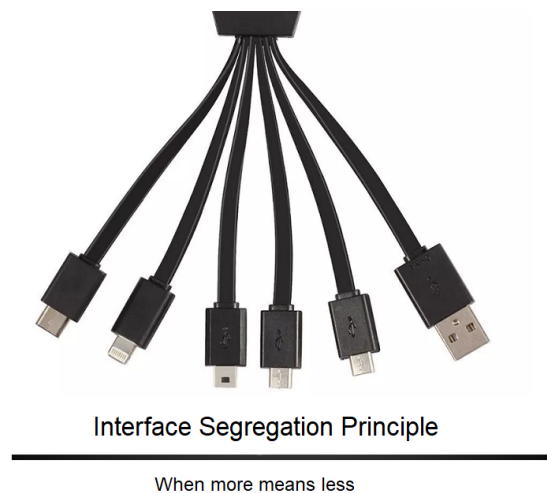


Figure 1: Interface segregation principle

### 2.1 Interface

An interface is a blueprint for an object and what it is capable of. In Object Oriented Programming, an Interface is a description of all functions that an object must have in order to be an "X". Again, as an example, anything that "ACTS LIKE" a light, should have a `turnOn()` method and a `turnOff()` method. The purpose of interfaces is to allow the computer to enforce these properties and to know that an object of TYPE T must have functions called X,Y,Z, etc. [4]

### 2.2 ISP

"Clients should not be forced to depend upon interfaces that they do not use."[3]

This principle was developed by Robert C. Martin while working at Xerox as a consultant. Martin was in charge of developing a new printing management system that could carry out tasks concurrently, which was new by the time.[5]

While developing this new printing system, Martin noticed that small changes in the design caused large-scale deployments. In some same cases caused deployments on modules out of the scope of the change. The problem was mainly caused by a large fat interface used throughout all the modules of the system.[5]

This principle is simple and clear, interfaces should not be forced to implement methods that they do not use. It has the main goal of limiting the scope of future changes. However, it is very difficult, in the first iterations, to identify the critical parts of your solution that will violate the principle. So, my advice is to wait for your code to evolve, so you can identify where you can apply it. Don't try to guess future requirements because there is a chance that you make your code more complicated.[5]

## 2.3 Problem with will be solved with the ISP

With ISP you can prevent overloaded interfaces that forces the client to implement a method that is not required. The client ends up implementing a usefulness method, in other words a method that has no meaning to the client. This decreases the readability of the code and also confuses the developer using the client code. The client interface ends up violating SRP sometime since it might perform some action that is not related to it.[6]

## 3 Practical Example

In the first example the Ostrich class is forced to use the fly method. It is a clear violation of the Interface Segregation Principle. As you can see in the second example, the fly and walk methods were separated into an extra interface. With this implementation it is possible to use the fly and walk method at the Duck class and the walk method at the Ostrich class. Now no class is forced to implement an unused method.

```
interface Bird {
    fly(): void;
    walk(): void;
}

class Duck extends Bird{
    fly(){
        // Duck can fly
    }
    walk(){
        // Duck can walk
    }
}

class Ostrich extends Bird{
    fly(){
        // Ostrich cant fly... throw some error
    }
    walk(){
        // Ostrich can walk
    }
}
```

Listing 1: initial scenario

```
interface BirdFly{
    fly(): void;
}
interface BirdWalk{
    walk(): void;
}
class Duck extends BirdFly, BirdWalk{
    fly(){
        // Duck can fly
    }
    walk(){
        // Duck can walk
    }
}
class Ostrich extends BirdWalk{
    walk(){
        // Ostrich can walk
    }
}
```

Listing 2: ISP Implementation

# Bibliography

- [1] Digitalocean, "Solid: The first 5 principles of object oriented design," 2021. [Online]. Available: [https://www.digitalocean.com/community/conceptual\\_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design](https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design)
- [2] A. Ridwan, "Introduction to the solid principle," 2021. [Online]. Available: <https://www.section.io/engineering-education/introduction-to-solid-principle/>
- [3] T. JANSSEN, "Solid design principles explained: Interface segregation with code examples," 2018. [Online]. Available: <https://www.section.io/engineering-education/introduction-to-solid-principle/>
- [4] "Interfaces in object oriented programming languages." [Online]. Available: <https://www.cs.utah.edu/~germain/PPS/Topics/interfaces.html>
- [5] V. M. Pinzon, "Solid: Interface segregation principle." [Online]. Available: <https://dev.to/victorpinzon1988eng/solid-interface-segregation-principle-31c2>
- [6] "Interface segregation principle." [Online]. Available: <https://www.c-sharpcorner.com/UploadFile/pranayamr/overview-of-interface-segregation-principle/>



# List of Figures

|  |   |
|--|---|
| Figure 1 Interface segregation principle . . . . . | 2 |
|--|---|

## List of source codes

|                                |   |
|--------------------------------|---|
| 1 initial scenario . . . . .   | 4 |
| 2 ISP Implementation . . . . . | 5 |