SEMINAR PAPER

In Computer Science

Interface segregation principle

By: Peter Oettl

Student Number: 1910257146

Supervisor: DI Wolfgang Berger

Wien, May 24, 2021

Contents

1 SOLID principles

These principles establish practices that lend to developing software 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 Responsiblity 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.

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.

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.

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. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

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.

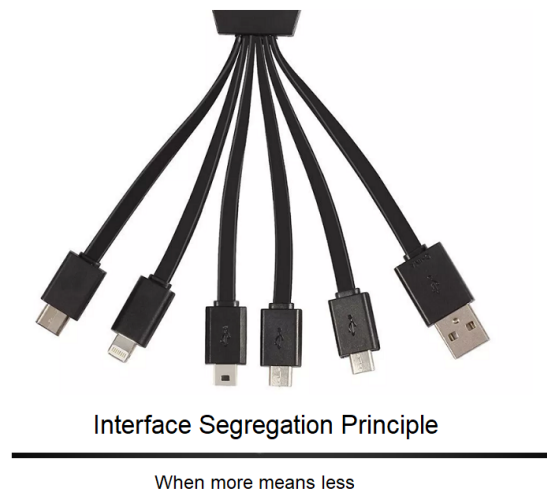# 2 Interface segregation principle

Figure 1: Interface segregation principle

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

Im Prinzip kann man das ISP so zusammenfassen: Verwende immer nur die kleinstmgliche Schnittstelle zu deinen Abhigkeiten. Je mehr Funktionen eine Komponente an ihren Abhigkeiten aufrufen kann, desto abhiger wird sie von ihr. Wenn sich diese Funktionen nich ern (z.B. die Signatur einer Methode), muss die nutzende Komponente neu kompiliert werden. Aurdem knnen Funktionen aufgerufen werden, die die nutzende Komponente weder bentigt, noch anwenden soll, z.B. die clear()-Methode einer Liste, die sie eigentlich nur durchlaufen soll. Zuletzt mssen auch implementierende Klassen von zu gron Interfaces fr sie unntige Methoden implementieren, nur um der Schnittstelle zu entsprechen.

3 Practical Example

```
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

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

List of Figures

List of Abbreviations

aaa   allaboutapps