

Interface Segregation Principle

When we design an application we should take care how we are going to abstractize a module which contains several submodules. Considering the module implemented by a class, we can have an abstraction of the system done in an interface...

[NEW Forum Discussions: NEW](#)

[What Design Pattern Should I Choose?](#)

[Design Principles and Patterns](#) [Interface Segregation Principle \(ISP\)](#)

[Motivation](#)

When we design an application we should take care how we are going to make abstract a module which contains several submodules. Considering the module implemented by a class, we can have an abstraction of the system done in an interface. But if we want to extend our application adding another module that contains only some of the submodules of the original system, we are forced to implement the full interface and to write some dummy methods. Such an interface is named fat interface or polluted interface. Having an interface pollution is not a good solution and might induce inappropriate behavior in the system.

The Interface Segregation Principle states that clients should not be forced to implement interfaces they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule. Intent

Clients should not be forced to depend upon interfaces that they don't use. Example

Below is an example which violates the Interface Segregation Principle. We have a Manager class which represent the person which manages the workers. And we have 2 types of workers some average and some very efficient workers. Both types of workers works and they need a daily launch break to eat. But now some robots came in the company they work as well , but they don't eat so they don't need a launch break. One on side the new Robot class need to implement the IWorker interface because robots works. On the other side, the don't have to implement it because they don't eat.

This is why in this case the IWorker is considered a polluted interface.

If we keep the present design, the new Robot class is forced to implement the eat method. We can write a dummy class which does nothing(let's say a launch break of 1 second daily), and can have undesired effects in the application(For example the reports seen by managers will report more lunches taken than the number of people).

According to the Interface Segregation Principle, a flexible design will not have polluted interfaces. In our case the IWorker interface should be split in 2 different interfaces. // interface segregation principle - bad example

```
interface IWorker {
    public void work();
    public void eat();
}
```

```
class Worker implements IWorker{
    public void work() {
        // ....working
    }
    public void eat() {
        // ..... eating in launch break
    }
}
```

```
class SuperWorker implements IWorker{
    public void work() {
        //.... working much more
    }
}
```

```
public void eat() {
    //.... eating in launch break
}
}
```

```
class Manager {
    IWorker worker;
```

```
public void setWorker(IWorker w) {
```

```
worker=w;
}

public void manage() {
worker.work();
}
}
```

Following it's the code supporting the Interface Segregation Principle. By splitting the IWorker interface in 2 different interfaces the new Robot class is no longer forced to implement the eat method. Also if we need another functionality for the robot like recharging we create another interface IRechargeable with a method recharge. // interface segregation principle - good example

```
interface IWorker extends Feedable, Workable {
}
```

```
interface IWorkable {
public void work();
}
```

```
interface IFeedable{
public void eat();
}
```

```
class Worker implements IWorkable, IFeedable{
public void work() {
// ....working
}
}
```

```
public void eat() {
//.... eating in launch break
}
}
```

```
class Robot implements IWorkable{
public void work() {
// ....working
}
}
```

```
class SuperWorker implements IWorkable, IFeedable{
public void work() {
//.... working much more
}
}
```

```
public void eat() {
//.... eating in launch break
}
}
```

```
class Manager {
Workable worker;

public void setWorker(Workable w) {
worker=w;
}

public void manage() {
worker.work();
}
}
```

Conclusion

If the design is already done fat interfaces can be segregated using the Adapter pattern.

Like every principle Interface Segregation Principle is one principle which require additional time and effort spent to apply it during the design time and increase the complexity of code. But it produce a flexible design. If we are going to apply it more than is necessary it will result a code containing a lot of interfaces with single methods, so applying should be done based on experience and common sense in identifying the areas where extension of code are more likely to happens in the future.