# Python Interface Segregation Principle

**If this Python Tutorial saves you hours of work, please whitelist it in your ad blocker 😭 and**

Donate Now

(https://www.pythontutorial.net/donation/)

**to help us ❤️ pay for the web hosting fee and CDN to keep the website running.**

**Summary**: in this tutorial, you'll learn about the interface segregation principle and how to apply it in Python.

## Introduction to the interface segregation principle

The interface segregation principle is one of five SOLID principles in object-oriented programming:

- **S** – Single responsibility Principle (https://www.pythontutorial.net/python-oop/python-single-responsibility-principle/)
- **O** – Open-closed Principle (https://www.pythontutorial.net/python-oop/python-open-closed-principle/)
- **L** – Liskov Substitution Principle (https://www.pythontutorial.net/python-oop/python-liskov-substitution-principle/)
- **I** – Interface Segregation Principle
- **D** – Dependency Inversion Principle (https://www.pythontutorial.net/python-oop/python-dependency-inversion-principle/)

An interface is a description of behaviors that an object can do. For example, when you press the power button on the TV remote control, it turns the TV on, and you don't need to care how.

In object-oriented programming, an interface is a set of methods (https://www.pythontutorial.net/python-oop/python-methods/) an object must-have. The purpose of interfaces is to allow clients to request the correct methods of an object via its interface.

Python uses abstract classes (https://www.pythontutorial.net/python-oop/python-abstract-class/) as interfaces because it follows the so-called duck typing principle. The duck typing principle states that "if it walks like a duck and quacks like a duck, it must be a duck." In other words, the methods of a class determine what its objects will be, not the type of the class.

The interface segregation principle states that an interface should be as small a possible in terms of cohesion. In other words, it should do ONE thing.

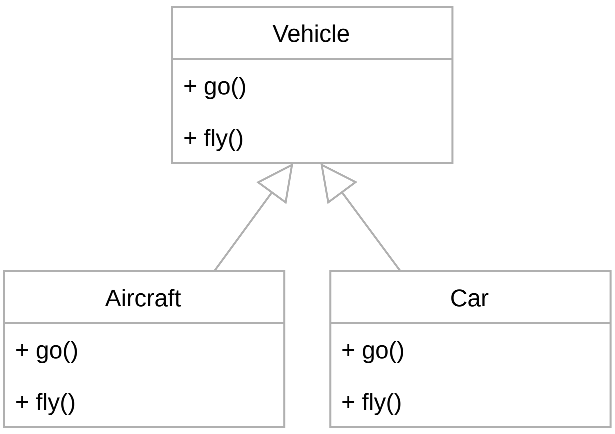
It doesn't mean that the interface should have one method. An interface can have multiple cohesive methods.

For example, the `Database` interface can have the `connect()` and `disconnect()` methods because they must go together. If the `Database` interface doesn't use both methods, it'll be incomplete.

Uncle Bob (https://en.wikipedia.org/wiki/Robert_C._Martin) , who is the originator of the SOLID term, explains the interface segregation principle by advising that "Make fine-grained interfaces that are client-specific. Clients should not be forced to implement interfaces they do not use."

## Interface segregation principle example

Consider the following example:



First, define a `Vehicle` abstract class that has two abstract methods, `go()` and `fly()` :

```
from abc import ABC, abstractmethod


class Vehicle(ABC):
    @abstractmethod
    def go(self):
        pass

    @abstractmethod
    def fly(self):
        pass
```

Second, define the `Aircraft` class that inherits from the `Vehicle` class and implement both `go()` and `fly()` methods:

```
class Aircraft(Vehicle):
    def go(self):
        print("Taxiing")

    def fly(self):
        print("Flying")
```

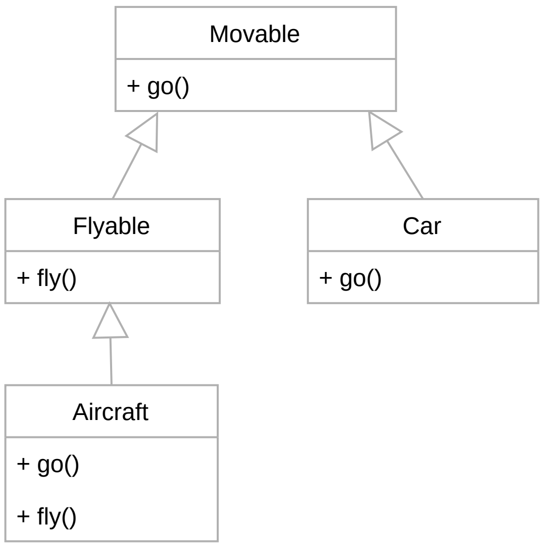
Third, define the `Car` class that inherits from the `Vehicle` class. Since a car cannot fly, we raise an exception in the `fly()` method:

```
class Car(Vehicle):
    def go(self):
        print("Going")

    def fly(self):
        raise Exception('The car cannot fly')
```

In this design the `Car` class must implement the `fly()` method from the `Vehicle` class that the `Car` class doesn't use. Therefore, this design violates the interface segregation principle.

To fix this, you need to split the `Vehicle` class into small ones and inherits from these classes from the `Aircraft` and `Car` classes:



First, split the `Vehicle` interface into two smaller interfaces: `Movable` and `Flyable`, and inherits the `Movable` class from the `Flyable` class:

```python
class Movable(ABC):
    @abstractmethod
    def go(self):
        pass


class Flyable(Movable):
    @abstractmethod
    def fly(self):
        pass
```

Second, inherits from the `Flyable` class from the `Aircraft` class:

```python
class Aircraft(Flyable):
    def go(self):
        print("Taxiing")

    def fly(self):
        print("Flying")
```

Third, inherit the `Movable` class from the `Car` class:

```
class Car(Movable):
    def go(self):
        print("Going")
```

In this design, the `Car` only need to implement the `go()` method that it needs. It doesn't need to implement the `fly()` method that it doesn't use.

## Summary

- The interface segregation principle advises that the interfaces should be small in terms of cohesions.
- Make fine grained interfaces that are client-specific. Clients should not be forced to implement interfaces they do not use.