course: COMP6210 week: 03 lesson: "02b" topic: solid-principle-4-interface-segregation-principle

DOWNLOAD PDF CLICK HERE

SOLID Principles

Each week in this section of the course we will look at 1 or 2 of the SOLID principles.

The principles stand for:

- Single Repsonsibility Principle
- Open Closed Principle
- Liskov Substitution Principle
- **Interface Segregation Principle**
- Dependency inversion principle

The idea behind using this principles is that you look at how you strucutre and organise your code so that it is easier to maintain. Although it will look like you need to do more work (and it is) for things that are relatively simple, it will help you to read and update your code better once it is in production.

The other takeaway from these principles is that you are going to go through a process of decoupling your code from the other code in your application.

PART 2: - SOLID Principle 4 - Interface Segregation Principle

> “Clients should not be forced to depend upon interfaces that they do not use.”

> Similar to the Single Responsibility Principle, the goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.

The idea behind this principle is that you think about what is in your interface.

The tricky part is that if you are designing your program and you think you have the right interfaces etc. there is the possibility that they may change.

So a tip would be that if something sounds remotely like it could be change or not required for every instance of a class then put it in a seperate interface.

Let's look at this use case:

Say you are creating some software for a coffee machine that can also grind the beans, it is safe to assume that all coffee machines need to grind the coffee beans.

What may not be the same is how the coffee is made.

So you may need to brew the coffee for a long black coffee and for this you just need the coffee beans. However for a flatwhite you also need milk. The machine is also capable of making hot chocolates. So how do we create a program who can take care of all these things?

You could create an interface like this:

```
public interface ICoffeeMachine
{
  void gridBeans();
  void brewLongBlack(bool needsBeans);
  void brewFlatWhite(bool needsBeans, bool needsMilk);
}

class LongBlack: ICoffeeMachine
{
  void grindBeans()
  {
    Console.WriteLine("Grind The Beans");
  }

  void brewLongBlack(bool needsBeans)
  {
    Console.WriteLine("Make a Long Black Coffee");
  }

  void brewFlatWhite(bool needsBeans, bool needsMilk)
  {
    Console.WriteLine("Make a Flat White Coffee");
  }
}
```

In the above class you would argue that simply changing the class name would be enough to make it more generic. But we now know that after looking at OCP that we should just keep changing that class when different types of coffee get added.

It is not uncommon and possibly better to create multiple and smaller interfaces so that we can create different classes that match the combinations.

The idea of an interface is that we have a contract and that any new features (classes) can be made up of any combination of interfaces.

So let's refactor the code above into a better setup.

```
public interface ICoffeeMachine
{
  void gridBeans();
}

public interface ILongBlack
{
```

```
    void brewLongBlack(bool needsBeans);
  }

  public interface IFlatWhite
  {
    void brewFlatWhite(bool needsBeans, bool needsMilk);
  }

  class LongBlack: ICoffeeMachine, ILongBlack
  {
    void grindBeans()
    {
      Console.WriteLine("Grind The Beans");
    }

    void brewLongBlack(bool needsBeans)
    {
      Console.WriteLine("Make a Long Black Coffee");
    }
  }

  class FlatWhite: ICoffeeMachine, IFlatWhite
  {
    void grindBeans()
    {
      Console.WriteLine("Grind The Beans");
    }

    void brewFlatWhite(bool needsBeans, bool needsMilk)
    {
      Console.WriteLine("Make a Long Black Coffee");
    }
  }
```

Now it is possible to create a coffee machine that only makes long blacks or only flatwhites, but in addition we can add the other option at a later stage without breaking the code that is already in place.

If we want to add another drink option we can add a new interface and a class that implements one or more interfaces, but it won't need to implement what we do not need.