course: COMP6215 week: 04 lesson: 02 topic: solid-5-and-assignment-prep

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 5 - Dependency Inversion Principle

The Dependency Principle set the following guidelines to a project:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Which then leads to:

1. the high-level module depends on the abstraction, and
2. the low-level depends on the same abstraction.

Now if you look at the above descriptions - we have actually already covered them when we implemented previous principles.

The idea behind the DIP is that all the classes are modeled off the same set of abstraction classes or interfaces and that a "sub" class is not inheriting from a base class, but that it implements the same abstractions as what the base class does.

Last week when we looked at the ISP (Interface segregation principle) we did exactly that to solve that principle.

The difference between the 2 principles is as follows:

1. The ISP pricinple simply states that you shouldn't overload the interfaces so that you force things on the classes that implement them.
2. The DIP principle states more or less the solution for that, but you could can also use abstract classes to comply with this principle

Abstract classes VS Interfaces

So we can use an abstract class or an interface to set a template for other classes, so what are the differences and why use one over the other.

Abstract Classes

Let's have a look at the following abstract class and break it down line by line:

```
01.  abstract class Person
02.  {
03.      public string Name { get; set; }
04.      public string Email { get; set; }
05.
06.      public Person(string name, string email)
07.      {
08.          Name = name;
09.          Email = email;
10.      }
11.
12.      public string ShowNameAndEmail()
13.      {
14.          return $"Hi, I am {Name} and my email is: {Email}";
15.      }
16.
17.      public abstract string Greeting();
18.  }
```

- **Line 01** : We define that we want this class to be an abstract class
- **Line 03 - 04** : We define our properties
- **Line 06 - 10** : We set a constrcutor
- **Line 12 - 15** : We have a method that is implemented
- **Line 17** : We define an abstract method that needs to be implemented in the subclass

What are the limitations of an abstract class?

- An abstract cannot be instantiated and therefore needs to be inherited - that is the purpose of an abstract class.
- You can only inherit from a single class, but that class could inherit from another class

Note: that the second point above creates a tight coupliing between classes and dependencies which is not what you want.

Interface

An interface for a similar setup looks like this:

```
interface IPerson
{
    string Name { get; set; }
    string Email { get; set; }
    string ShowNameAndEmail();

    string Greeting();
}
```

The above code is much simpler, but it comes with a few draw backs of its own:

1. An interface cannot implement anything - it is purely a contract that requires the classes that are linked to the interface to implement everything
2. An interface cannot contain constructors, destructors or fields - only properties or methods

Combining both

You can link an interface to a abstract class so that the abstract class is bound to the contract, the abstract class can then decide which methods it implements and which it sets to abstract to be implemented later.

This can get messy, but with good program design it is doable.