course: COMP6210 week: 03 lesson: 02 topic: solid-principle-3-liskov-subsitution-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:

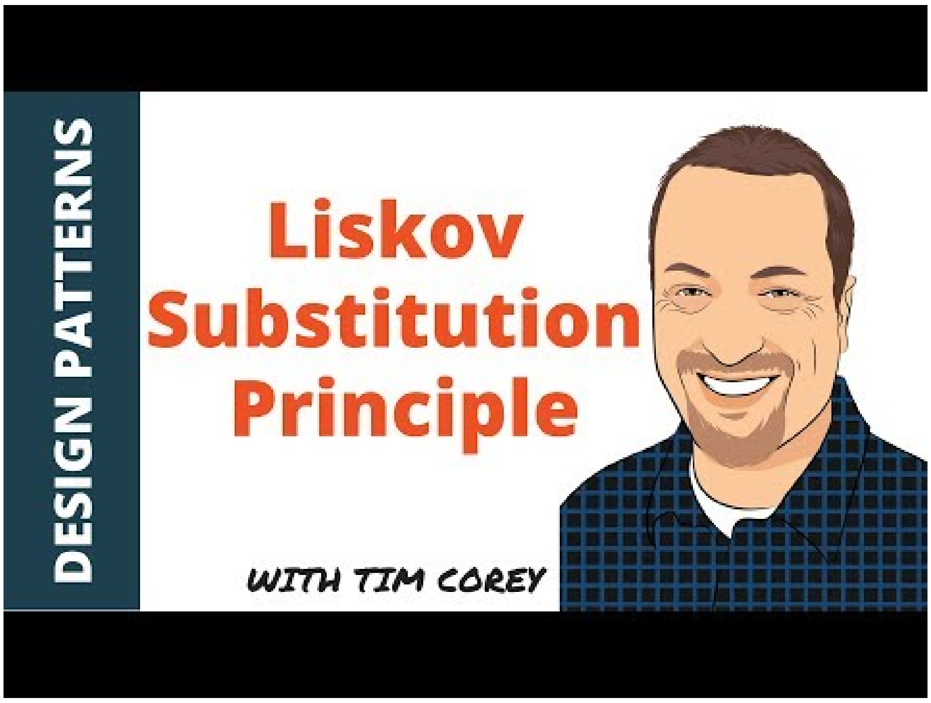
- Single Repsonsibility Principle
- Open Closed Principle
- **Liskov Substitution Principle**
- Interface Segregation Principle
- Dependency inversion principle

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 3 - Liskov Substitution Principle

Watch a video about it here



The Liskov Substitution Principle (LSP from here on in) stands for "That any class that is a subclass of another class should be able to placed in base class' position instead."

Wikipedia's definition is:

> Substitutability is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.).
>
> It is a semantic rather than merely syntactic relation, because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular.

So far whenever we have looked at inheritance we have been limited by the code that is either `public`, `protected` or `private`. These limitations are syntactical as they are written and defined in code.

The LSP is more semantical as it looks at wether something "breaks" logically.

An example given by Uncle Bob (Bob Martin who developed these principles) is the use of squares and rectangles. We could extend the exact same idea to circles and elipses.

When we looked at the OCP principle - we created the dependency of the `Shape` class and created a `Rectangle` class and an `Elipse` class. Because both of those classes inherit from the `Shape` class, we should be able to replace the instance of the class without breaking the code with the parent class.

This is how would instantiate the class normally.

```
Shape shape1 = new Shape();
shape1.Width = 24.00;
shape1.Height = 8.00;
```

But the code should also be able to run when you do this:

```
Shape shape1 = new Rectangle();
shape1.Width = 24.00;
shape1.Height = 8.00;
```

The rectangle may have some different calculation, but it is still a shape and therefore it can be defined as the type. This way you can see at the implementation of the `Rectangle` that it inherits from the `Shape` class.

If the code where to break then `Rectangle` should be evaluated to see if it is needs to be its own class - and not inheriting from `Shape`.

Just beaware that when dealing with abstract classes because you cannot instantiate them, they can never be truly conforming to LSP. You will also probably find that you will use interfaces a lot more than abstract classes, althought there are use cases for both.

The other parts

The Liskov principle also talks about the following:

- Contravariance
- Covariance
- Preconditions
- Postconditions

Contravariance

This refers to the arguments that are set for a contructor. When confirming to LSP these cannot change. For example the following code breaks LSP, becuase the argument changes:

```
class Something
{
  public Something (char letter) {}
  ...
}

class SomethingElse : Something
{
  public SomethingElse (string letter) : base (letter) {}
  ...
}
```

Covariance

Just like how parameters need to stay the same, the return type also needs to be same. The following example therefore breaks LSP since it changes the return type from an int to a double:

```
// In the base class
public int calculateSomething()
{
  return 0;
}

//In the sub class
public double calculateSomething()
{
  return 0.00;
}
```

Preconditions

Preconditions are parts of code that force something to be true before other code is executed.

So if a base class does not have this in their method, a subclass cannot have this either.

The following example will therefore break LSP because it affects the flow of the method:

```
// base class
public double calculateWage(int rank)
{
  double basewage = 150;
  return basewage * rank;
}

// sub class
public double calculateWage(int rank)
{
  if(rank < 0 || rank > 5)
  {
    return 0;
  }

  double basewage = 150;
  return basewage * rank;
}
```

Postconditions

Postconditions are parts of code that force something to be true after other code is executed.

So if a base class does not have this in their method, a subclass cannot have this either.

The following example will therefore break LSP because it affects the flow of the method:

```
// base class
public double GetSquareRoot(int x)
{
  double output = 0;

  if(x >= 0)
  {
    output = //calculate square root.....
  }

  double total = (output * output == x) ? output : 0;

  return total;
}

// sub class
public double GetSquareRoot(int x)
{
  double output = 0;

  if(x >= 0)
  {
    output = //calculate square root.....
  }

  double total = (output * output != x) ? output : 0;  // changing the equation is not allowed.

  return total;
}
```