SOLID Principles

ZUNERA ZAHID

Main Concept

- SOLID is the most popular sets of design principles in object-oriented software.
- make software designs more understandable, flexible and maintainable.
- Generally, software should be written as simply as possible in order to produce the desired result. However, once updating the software becomes painful, the software's design should be adjusted to eliminate the pain.

Concepts

- Single Responsibility Principle
- Open / Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion

Single Responsibility Principle

- Every Module, Class, or function should have responsibility over a single part of the functionality provided by the software.
- That responsibility should be entirely encapsulated by the class.

Single Responsibility Principle

Let's take a simpler example, where we have a list of number L = [n1, n2, ..., nx] and we compute some mathematical functions to this list. For example, compute the mean, median, etc.

A bad approach would be to have a single function doing all the work:

```python
import numpy as np

def math_operations(list_):
    # Compute Average
    print(f"the mean is {np.mean(list_)}")
    # Compute Max
    print(f"the max is {np.max(list_)}")

math_operations(list_ = [1,2,3,4,5])
# the mean is 3.0
# the max is 5
```

Single Responsibility Principle

The first thing we should do, to make this more SRP compliant, is to split the function math_operations into **atomic functions**! Thus, when a function's responsibility cannot be divided into more subparts.

The second step is to make a single function (or class), generically named, "main". This will call all the other functions one-by-one in a step-to-step process.

```
1   def get_mean(list_):
2       '''Compute Mean'''
3       print(f"the mean is {np.mean(list_)}")
4
5   def get_max(list_):
6       '''Compute Max'''
7       print(f"the max is {np.max(list_)}")
8
9   def main(list_):
10      # Compute Average
11      get_mean(list_)
12      # Compute Max
13      get_max(list_)
14
15  main([1,2,3,4,5])
16  # the mean is 3.0
17  # the max is 5
```

Single Responsibility Principle

Now, you would only have one single reason to change each function connected with “main”.

The result of this simple action is that now:

1. It is easier to localize errors. Any error in execution will point out to a smaller section of your code, accelerating your debug phase.
2. Any part of the code is reusable in other section of your code.
3. Moreover and, often overlooked, is that it is easier to create testing for each function of your code. Side note on testing: You should write tests before you actually write the script. But, this is often ignored in favour of creating some nice result to be shown to the stakeholders instead.

This is already a much bigger improvement with respect to the first code example. But, having created a “main” and calling functions with single responsibility is not the full fulfilment of the SR principle.

Indeed, our “main” has many reasons to be changed. The class is actually fragile and hard to maintain.

To solve that, let’s introduce the next principle:

Open – closed Principle

To clarify this point let's refer to the example we saw earlier. If we wanted to add new functionality, for example, compute the median, we should have created a new method function and add its invocation to "main". That would have added an *extension* but also *modified* the main.

. . .

We can solve this by turning all the functions we wrote into subclasses of a class. In this case, I have created an abstract class called "Operations" with an abstract method "get_operation". (Abstract classes are generally an advanced topic. If you don't know what an abstract class is, you can run the following code even without).

Now, all the old functions, now classes are called by the __subclasses__() method. That will find all classes inheriting from Operations and operate the function "operations" that is present in all sub-classes.

```python
import numpy as np
from abc import ABC, abstractmethod

class Operations(ABC):
    '''Operations'''
    @abstractmethod
    def operation():
        pass

class Mean(Operations):
    '''Compute Max'''
    def operation(list_):
        print(f"The mean is {np.mean(list_)}")

class Max(Operations):
    '''Compute Max'''
    def operation(list_):
        print(f"The max is {np.max(list_)}")

class Main:
    '''Main'''
    @abstractmethod
    def get_operations(list_):
        # __subclasses__ will found all classes inheriting from Operations
        for operation in Operations.__subclasses__():
            operation.operation(list_)


if __name__ == "__main__":
    Main.get_operations([1,2,3,4,5])
# The mean is 3.0
# The max is 5
```

Open – closed Principle

If now we want to add a new operation e.g.: median, we will only need to add a class “Median” inheriting from the class “Operations”. The newly formed sub-class will be immediately picked up by __subclasses__() and no modification in any other part of the code needs to happen.

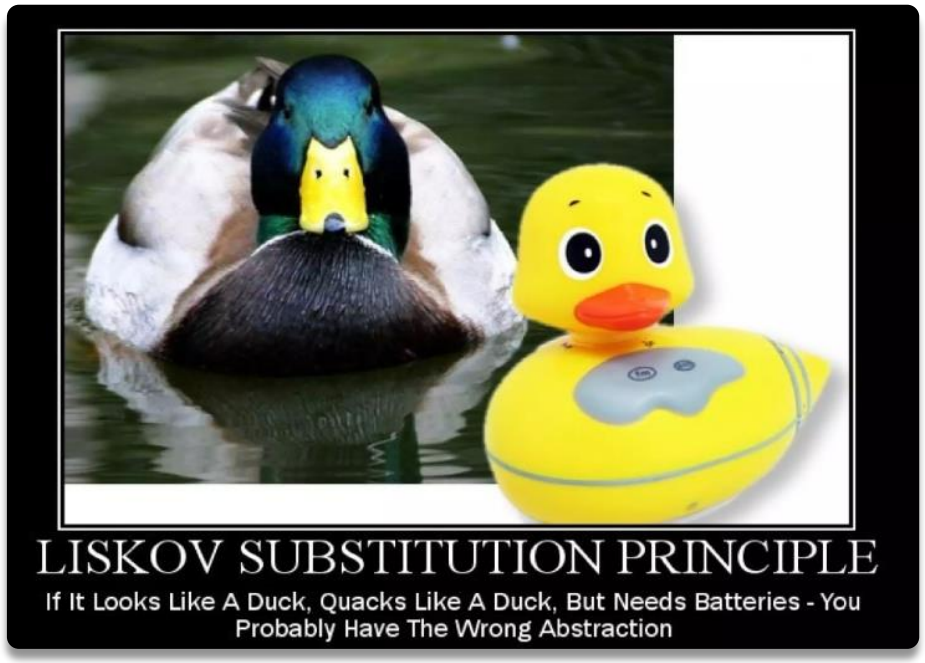
The result is a very flexible class, that requires minimum time to be maintained.

Open – closed Principle

Open – closed Principle

- Every Module, Class, or function should be open for extension, but closed for modification.
- An entity can allow its behavior to be extended without modifying its source code.

Liskov Substitution principle



- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

- The **Liskov substitution principle** (**LSP**) is a particular definition of a subtyping relation, called (**strong**) **behavioral subtyping**.

Liskov Substitution principle

> "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it"

Alternatively, this can be expressed as "*Derived classes must be substitutable for their base classes*".

In (maybe) simpler words, if a subclass redefines a function also present in the parent class, a client-user should not be noticing any difference in **behaviour,** and it is a **substitute** for the base class.
For example, if you are using a function and your colleague change the base class, you should not notice any difference in the function that you are using.

Among all the SOLID principle, this is the most abstruse to understand and to explain. For this principle, there is no standard "template-like" solution where it must be applied, and it is hard to offer a "standard example" to showcase.

Liskov Substitution principle

In the most simplistic way, I can put it, this principle can be summarised by saying:
If in a *subclass*, you redefine a *function* that is also present in the *base class*, the two functions ought to have the same behaviour. This, though, does not mean that they must be *mandatorily* equal, but that the user, should expect that the same *type* of result, given the same input.
In the example ocp.py, the "operation" method is present in the subclasses and in the base class, and an end-user should expect the same behaviour from the two.

The result of this principle is that we'd write our code in a consistent manner and, the end-user will need to learn how our code works, only one.

Interface Segregation Principle

- No client should be forced to depend on methods it does not use.
- Many client-specific interfaces are better than one general-purpose interface
- Splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.

Interface Segregation Principle

> "Many client-specific interfaces are better than one general-purpose interface"

In the contest of classes, an interface is considered, *all the methods and properties* "**exposed**", thus, everything that a user can interact with that belongs to that class.

In this sense, the IS principles tell us that a class should only have the interface needed (SRP) and avoid methods that won't work or that have no reason to be part of that class.

This problem arises, primarily, when, a subclass inherits methods from a base class that it does not need.

Interface Segregation Principle

```python
import numpy as np
from abc import ABC, abstractmethod

class Mammals(ABC):
    @abstractmethod
    def swim() -> bool:
        print("Can Swim")

    @abstractmethod
    def walk() -> bool:
        print("Can Walk")

class Human(Mammals):
    def swim():
        return print("Humans can swim")

    def walk():
        return print("Humans can walk")

class Whale(Mammals):
    def swim():
        return print("Whales can swim")
```

Interface Segregation Principle

For this example, we have got the abstract class "Mammals" that has two abstract methods: "walk" and "swim". These two elements will belong to the sub-class "Human", whereas only "swim" will belong to the subclass "Whale".

And indeed, if we run this code we could have:

```
1  Human.swim()
2  Human.walk()
3
4  Whale.swim()
5  Whale.walk()
6
7  # Humans can swim
8  # Humans can walk
9  # Whales can swim
10 # Can Walk
```

The sub-class whale can still invoke the method "walk" but **it shouldn't**, and we must avoid it.

The way suggested by ISP is to create more ***client-specific interfaces*** rather than ***one general-purpose interface***. So, our code example becomes:

Interface Segregation Principle

```python
from abc import ABC, abstractmethod

class Walker(ABC):
  @abstractmethod
  def walk() -> bool:
    return print("Can Walk")

class Swimmer(ABC):
  @abstractmethod
  def swim() -> bool:
    return print("Can Swim")

class Human(Walker, Swimmer):
  def walk():
    return print("Humans can walk")
  def swim():
    return print("Humans can swim")

class Whale(Swimmer):
  def swim():
    return print("Whales can swim")

if __name__ == "__main__":
  Human.walk()
  Human.swim()

  Whale.swim()
  Whale.walk()

# Humans can walk
# Humans can swim
# Whales can swim
```

Now, every sub-class inherits only what it needs, avoiding invoking an out-of-context (wrong) sub-method. That might create an error hard to catch.

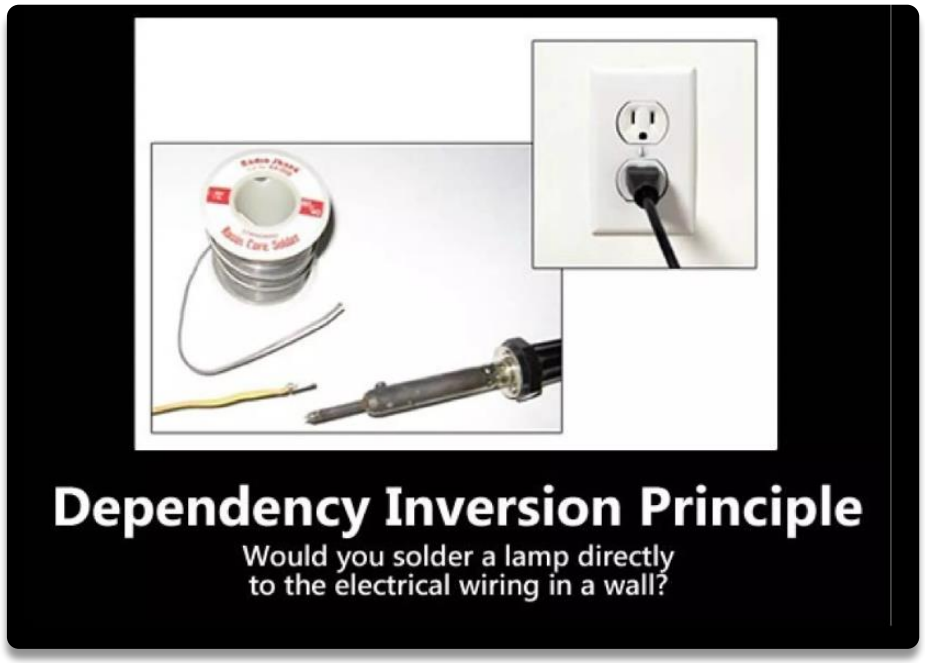
This principle is closely connected with the other ones and specifically, it tells us to *keep the content of a subclass clean from elements of no use to that subclass.* This has the final aim to keep our classes clean and minimise mistakes.

Interface Segregation Principle

Dependency inversion principle

"Abstractions should not depend on details. Details should depend on abstraction. High-level modules should not depend on low-level modules. Both should depend on abstractions"

Dependency inversion principle



- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

SOLID Design Principles

Software inevitably changes/evolves over time (maintenance, upgrade)

- **S**ingle responsibility principle (SRP)
  - Every class should have only one reason to be changed
  - If class "A" has two responsibilities, create new classes "B" and "C" to handle each responsibility in isolation, and then compose "A" out of "B" and "C"
- **O**pen/closed principle (OCP)
  - Every class should be *open for extension* (derivative classes), but *closed for modification* (fixed interfaces)
  - Put the system parts that are likely to change into implementations (i.e. *concrete classes*) and define *interfaces* around the parts that are unlikely to change (e.g. *abstract base classes*)
- **L**iskov substitution principle (LSP)
  - Every implementation of an interface needs to fully comply with the requirements of this interface (requirements determined by its clients!)
  - Any algorithm that works on the interface, should continue to work for any substitute implementation
- **I**nterface segregation principle (ISP)
  - Keep interfaces as small as possible, to avoid unnecessary dependencies
  - Ideally, it should be possible to understand any part of the code in isolation, without needing to look up the rest of the system code
- **D**ependency inversion principle (DIP)
  - Instead of having concrete implementations communicate directly (and depend on each other), *decouple* them by formalizing their communication interface as an *abstract interface* based on the needs of the higher-level class