SOLID PRINCIPLES

David Corzo

Single-responsibility principle

```
class incorrect:
    def __init__(self,text):
        self.text = text
    def manipulate(self):
        self.text.upper()
    def printt(self):
        print(self.text)
```

```
class manipulate():
    def __init__(self,text):
        self.text = text
    def manipulate(self):
        self.text.upper()


class printt():
    def __init__(self,text):
        self.text = text
    def printt(self):
        print(self.text)
```

Open–closed principle

```
# original class:
class Employe:
    def __init__(self,_id,name,salary)
:
        self.id = _id
        self.name = name
        self.salary = salary
    def bonus(self):
        return self.salary*0.1
print(Employe(25,"Someone"))
```

```
# we add another functionality to the same class
class Employe:
    def __init__(self,_id,salary,name,t):
        self.id = _id
        self.name = name
        self.salary = salary
        self.type = t
    def bonus(self):
        if (self.type == "Permanent"):
            return self.salary*0.1
        else:
            return self.salary*0.05
print(Employe(25,"Someone",25000,"Permanent"))
```

“A new functionality should be implemented by adding new classes, attributes and methods, instead of changing the current ones or existing ones”

Better...

```
class Employe:
    def __init__(self,_id,name):
        self.id = _id
        self.name = name
        self.salary = None
    def __str__(self):
        return f"id: {self.id}, name: {self.name}, salary bonus:{self.salary}"
```

```
class PermanentEmploye(Employe):
    def __init__(self,_id,name,salary):
        self.id = _id
        self.name = name
        self.salary = salary*0.05
class TemporaryEmploye(Employe):
    def __init__(self,_id,name,salary):
        self.id = _id
        self.name = name
        self.salary = salary*0.05
```

```
em_1 = PermanentEmploye(25,"Someone_perm",90_000)
em_2 = TemporaryEmploye(25,"Someone_temp",90_000)
print(em_1)
print(em_2)
```

Liskov substitution principle

```
# Liskov principle
class T:
    def do(self,a,b):
        return a*b
class S(T):
    def do(self,a,b):
        return a/b
# New exception:
t_ = T().do(78,0)
s_ = S().do(78,0) # ZeroDivisionError
```

```
class Line(Shape):
    def calculate_surface_area(self):
        return -
1 # a line does not have a surface area
```

- “S is a subtype of T, then objects of type T may be replaced with objects of type S.”
- No new exceptions for subtype. Derived classes just extend without replacing the functionality of old classes.

Dependency inversion principle

```
# dependency inversion principle
class CEO:
    def instruct(self,instructions):
        return instructions
    def do_ceo_stuff(self,do):
        for i in do:
            print(i)

class truck_driver(CEO):
    def carry_out_those_instructions(self,instr
uctions):
        exec(instructions)
```

- High level modules should not depend on low level modules.

Interface segregation principle

```
class Shape:
    def draw_circle(self): # circle
        pass
    def draw_square(self): # square
        pass

class Circle(Shape):
    def draw_circle(self): # circle
        pass
    def draw_square(self): # square?
        pass
```

```
# solution
class Shape:
    def draw(self): # draw
        pass
class Circle(Shape): # inherits shape
    def draw(self):
        pass
class Square(Shape): # inherits shape
    def draw(self):
        pass
```