Single object responsibility - nigdy nie powino być więcej niż jednego powodu do istnienia klasy bądź metody”
Open closed responsibility – zasada otwartego zamknięcia -eliminuje ifologię. Klasa (lub metoda) powinna być otwarta na rozszerzenia, ale zamknięta na modyfikacje.
Liskov substitution – zasada substytucji Liskova - klasa dziedzicząca nie powinna nigdy wykonywać mniej operacji niż klasa, po której dziedziczy. Klasa dziedzicząca, nigdy nie powinna zmieniać sposobu działania odziedziczonych metod.
Interface segregation – rozdzielenie interfejsów - możliwie często wprowadzać do programu interfejsy (klasy abstrakcyjne)
Dependency inversion injection – odwrócenie zależności - bazować na kompozycji

1. Single object responsibility

```python
from dataclasses import dataclass

@dataclass
class Raport:
 content: str
 date: str

class Generator:
 def run(self):
   print("Generating a raport...")
```

```python
class Printer:
 def run(self):
   print("Printing a raport...")

class Manager:
 def __init__(self):
   self.generator = Generator()
   self.printer = Printer()
   self.raport = None

  def execute_generating(self):
   self.raport = self.generator.run()

 def execute_printing(self):
   if not self.raport:
     self.printer.run(self)
   else:
     print("Error! You have to generate a raport.")
```

“nigdy nie powino być więcej niż jednego powodu do istnienia klasy bądź metody” – możnaby to wszystko wsadzić w jedną klasę. Ale wtedy taka klasa miałaby 2 powody do istnienia – generowanie raportu i drukowanie raportu.

2. Open-closed responibility:

Eliminuje ifologię.

NOT OK:

```
class Executor:
 def do_sth_one(self):
   print("Sth one!")

 def do_sth_two(self):
   print("Sth two!")

 def do_sth_three(self):
   print("Sth three!")

class Menu:
 def __init__(self):
   self.executor = Executor()

 def show_menu(self):
   choice = int(input("Choose any option:\n1. Do Sth One \n2. Do Sth
Three \n3. Do Sth Three\n"))
   self.execute(choice)

 def show_error(self):
   print("Error!")

 def execute(self, choice: int):
   if choice == 1:
     self.executor.do_sth_one()
   elif choice == 2:
     self.executor.do_sth_two()
   elif choice == 3:
     self.executor.do_sth_three()
   else:
     self.show_error()

menu = Menu()
menu.show_menu()
```

OK:

```
class Executor:
 def do_sth_one(self):
   print("Sth one!")

 def do_sth_two(self):
   print("Sth two!")

 def do_sth_three(self):
   print("Sth three!")


class Menu:
 def __init__(self):
   self.executor = Executor()
   self.options = {1:self.executor.do_sth_one,
2:self.executor.do_sth_two, 3:self.executor.do_sth_three}

 def show_menu(self):
   choice = int(input("Choose any option:\n1. Do Sth One \n2. Do Sth
Three \n3. Do Sth Three\n"))
```

```
   self.execute(choice)

 def show_error(self):
   print("Error!")

 def execute(self, choice: int):
   self.options.get(choice, self.show_error)()


menu = Menu()
menu.show_menu()
```

“Klasa (lub metoda) powinna być otwarta na rozszerzenia, ale zamknięta na modyfikacje”. Powinniśmy dążyć do pisania takiego kodu, który umożliwi Nam dodanie nowych funkcjonalności bez naruszenia już istniejących.

```python
class Discount:
  def __init__(self, customer, price):
      self.customer = customer
      self.price = price
  def give_discount(self):
      if self.customer == 'fav':
          return self.price * 0.2
      if self.customer == 'vip':
          return self.price * 0.4
```

Open-Closed.py hosted with ❤ by GitHub view raw

Nie, to nie spełnia zasady OCP. OCP tego zabrania. Jeśli chcemy, być może, udzielić nowego rabatu procentowego, dla innego typu klientów, zobaczysz, że zostanie dodana nowa logika. Aby było zgodne z zasadą OCP, dodamy nową klasę, która wydłuży Rabat. W tej nowej klasie zaimplementowalibyśmy jej nowe zachowanie:

```python
class Discount:
    def __init__(self, customer, price):
      self.customer = customer
      self.price = price
    def get_discount(self):
      return self.price * 0.2
class VIPDiscount(Discount):
    def get_discount(self):
      return super().get_discount() * 2
```

Opened-Closed Updated.py hosted with ❤ by GitHub view raw

Jeśli zdecydujesz się na 80% zniżki dla klientów super VIP, powinno wyglądać tak:

Rozszerzenie bez modyfikacji.

```python
class SuperVIPDiscount(VIPDiscount):
    def get_discount(self):
        return super().get_discount() * 2
```

3. Liskov substitution

Zasada Liskov opiera się bowiem na idei, że klasa dziedzicząca nie powinna nigdy wykonywać **mniej operacji niż klasa, po której dziedziczy**.
Zawsze powinna ona rozszerzać działanie rodzica. To samo odnosi się do modyfikowania działania odziedziczonych metod. Dobrze zaimplementowana klasa dziedzicząca, **nigdy nie powinna zmieniać sposobu działania odziedziczonych metod**, przez np. zdefiniowanie własnej metody o tej samej nazwie, ale różnej funkcjonalności, co przysłonięta metoda rodzica (zakładając, że rodzic nie jest interfejsem lub klasą abstrakcyjną).

```
from dataclasses import dataclass

@dataclass
class Shape:
 @property
 def area(self):
   raise NotImplementedError

 def increase_width(self):
   raise NotImplementedError

@dataclass
class Rectangle(Shape):
 width: float
 height: float

 @property
 def area(self):
   return self.width * self.height

 def increase_width(self, to_add: float):
   self.width += to_add

@dataclass
class Square(Shape):
 side: float

 @property
 def area(self):
   return self.side * self.side

 def increase_width(self, to_add: float):
   self.side += to_add

rect = Rectangle(10, 5)
print(rect.area) # returns 50, OK
square = Square(10)
print(square.area) # returns 100, OK
square.increase_width(10)
print(square.area) # returns 400
```

4. Interface segregation
   Zasada ta mówi, aby możliwie często wprowadzać do programu interfejsy (lub szerzej patrząc ogólnie klasy abstrakcyjne/klasy rodzicielskie). Służyć one by miały segregacji różnych klas dziedziczących po tych samych rodzicach i jasne zdefiniowanie grup klas o wspólnych cechach.
5. Dependency inversion injection

Realizacja założeń powyższej zasady sprowadza się do tego, by często bazować na tzw. kompozycji modułów.

```python
from abc import ABC, abstractmethod

class Application(ABC):
 @abstractmethod
 def run(self) -> None:
   '''Method called when app is running'''

class GameApplication(Application):
 def run(self):
  print("Starting the game...")

class TaskManager:
 def __init__(self, app: Application):
  self.app = app
 def process(self):
  self.app.run()
```

Abstrakcję Application może przecież implementować zarówno klasa GameApplication, jak i ExcelApplication i wiele innych, a TaskManager nie odnosi się tylko i wyłącznie do jednej z nich (możemy w jego konstruktorze, przesyłać dowolny obiekt dziedziczący po Application, który ma przetwarzać).