Zasady Clean Code i SOLID

W szczegółach tkwi siła! Myślę, że w ten sposób mogę najlepiej zacząć dzisiaj omawiany temat. Jego celem będzie bowiem przedstawienie Ci najważniejszych zasad pisania czystego oprogramowania. Co znaczy czystego? Czyli takiego, które jest przede wszystkim **łatwo rozszerzalne**, **łatwe do czytania** - zrozumiałe dla innych programistów i dostosowane do najważniejszych konwencji kodowania.

Mówiąc o konwencjach, mam na myśli przede wszystkim sztywne reguły zawarte w **PEP** (Python Enhancement Proposals), do których powinniśmy się stosować. PEP obejmuje takie zasady pisania kodu jak: ilość linii w wierszu, przerwy między klasami, sposób nazywania metod itd. Są to reguły, których możemy przestrzegać, mając choćby pod ręką dokumentację.

Dlatego celem tej lekcji nie będzie na pewno przytaczanie wszystkich ścisłych zasad z PEP, ale skupienie się na innych obszarach pisania czystego i efektywnego kodu.

Umiejętność ta bowiem opiera się głównie na intuicji i wypracowanych nawykach. Dlatego uczenie się tych zasad na pamięć będzie nieskuteczne. Nauka pisania czystego kodu jest ciężką pracą. Wymaga czegoś więcej niż tylko wiedzy na temat zasad i wzorców. Musisz się przy tym spocić. Musisz to sam praktykować i przeżywać porażki.

Czym jest czysty kod?

Prawda jest brutalna - ilu na świecie jest programistów, tyle istnieje różnych przekonań i przyzwyczajeń do pisania kodu. Niektóre są lepsze, niektóre gorsze, ale wszystkie tworzą bazę wyznań i przekonań koderów.
Przyznaj, ile razy spotkałeś się choćby z takimSposobem (**camel case**) nazywania zmiennych, a ile_z_takim (**snake case**).
Dopóki nie mieszamy tych stylów nazewnictwa w jednym projekcie, nie jest powiedziane, że któryś z powyższych sposobów jest lepszy, a inny gorszy.
To wszystko wynika z ugruntowanych już nawyków.

Dlatego, definiując, czym tak naprawdę jest pojęcie czystego kodu, odwołajmy się do książki autorstwa **Martina R. "Clean Code"** (którą Ci serdecznie polecam!).

> *"Lubię, gdy mój kod jest elegancki i efektywny. Logika kodu powinna być prosta, aby nie mogły się w niej kryć błędy, zależności minimalne dla uproszczenia utrzymania, obsługa błędów kompletna zgodnie ze zdefiniowaną strategią, a wydajność zbliżona do optymalnej, aby nikogo nie kusiło psucie kodu w celu wprowadzenia niepotrzebnych optymalizacji. Czysty kod wykonuje dobrze jedną operację."*
>
> - **Bjarne Stroustrup, twórca C++**

> *Czysty kod jest prosty i bezpośredni. Czysty kod czyta się jak dobrze napisaną prozę. Czysty kod nigdy nie zaciemnia zamiarów projektanta; jest pełen trafnych abstrakcji i prostych ścieżek sterowania.*

- **Grady Booch, autor Object Oriented Analysis and Design with Applications**

Czysty kod to taki, którego tworzenie spędza sen z powiek
moim uczniom programowania i najbardziej Nas poróżnia.
Założę się, że nie ma nic bardziej denerwującego na świecie niż
mentor robiący code review i potrafiący przyczepić się do największego
szczegółu w kodzie! Źle nazwana zmienna, kod zamknięty
na nowe funkcjonalności i moje zbyt częste wykrzyknienia "WTF!? DAFQ!?" to
oznaki, że w swoim projekcie łamiesz zasady efektywnej architektury
programowania.

- **Mentor Devs-Mentoring.pl**

Czyste Zmienne

Celem tego szkolenia jest przede wszystkim przekazanie Ci praktycznej wiedzy z zakresu efektywnego wytwarzania oprogramowania. Dlatego nie przedłużając, chciałbym na przykładach zebrać najważniejsze zasady.

W tej sekcji skupimy się na prawidłowym nazywaniu i tworzeniu zmiennych.

Zasada 0.

Nie używaj komentarzy w kodzie! Zaczynam od przedstawienia Ci tej złotej zasady, ponieważ zauważyłem, że młodzi programiści zapominają o niej najczęściej. Co więcej, są oni również błędnie ugruntowani w przekonaniu, że komentarze reprezentują kod wysokiej jakości!

Zapamiętaj raz na zawsze! Nigdy nie komentuj swojego kodu. Komentarze mogą Ci posłużyć tylko do tworzenia, tzw. **docstringów**.
Docstringi to po prostu część dokumentacji projektu. Czasami przy implementacji rozbudowanych zadań, opisanie pokrótce, do czego służy dana klasa czy metoda, może być zbawieniem dla osób z zewnątrz, chcących na szybko zapoznać się z Twoim kodem.

Jako kontrargument, możesz powiedzieć, że komentarze pozwalają lepiej zrozumieć kod. Jasne! Ale czy nie lepiej od razu tak trafnie nazywać zmienne, metody czy klasy, aby kod nie wymagał żadnych wyjaśnień w komentarzach?

Przykład OK (docstring):

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root

    ...
```

NOK:

```
p = ‘123456’ # user’s password
```

OK:

```
password = ‘123456’
```

Zasada 1.

Używaj krótkich, treściwych i łatwych do wymówienia nazw zmiennych.

Wyjaśnienie:
Czytanie kodu ma być niczym czytanie książki. Zasada ta niejako wiążę się z **Zasadą 0**, ponieważ kod powinien być "**samokomentujący się**".

To znaczy tyle, że nazwy tworzonych zmiennych, funkcji i innych struktury powinny wyraźnie mówić za co odpowiadają i do czego służą.

I jeszcze jedno. Wiem, że to przykład dość skrajny, ale gdy rozmawiasz z innym programistą o swoim kodzie, to łatwiej Ci będzie omówić kod, w którym zamieściłeś zmienną: *lieutenant* czy po prostu *soldier*? :)
Miej to na uwadze!

OK:

```
import datetime
year = datetime.date.today().year
```

NOK:

```
import datetime
y = datetime.date.today().year
```

Zasada 2.

Bądź konsekwentny.

Wyjaśnienie:

Jeżeli decydujesz się w projekcie nazywać zmienne zgodnie z camelCase lub snake_case, to trzymaj się tylko jednego sposobu nazewnictwa. Nigdy nie mieszaj (w programowaniu też :)) różnych wariantów nazywania zmiennych.

To samo tyczy się języka, w którym nazywasz dane struktury. Niezbyt trafne będzie mieszanie języka angielskiego i polskiego w kodzie (no chyba, że studiowałeś polishing). Bądź konsekwentny i od początku do końca projektu trzymaj się ustalonych zasad nazewnictwa.

Ja osobiście, programując w Pythonie, wolę nazywać zmienne zgodnie z konwencją snake_case i tylko w języku angielskim (głównie dlatego, że łatwiej jest nazywać zmienne po angielsku niż w Naszym ojczystym języku).

NOK:

```
tekst  = "Ala ma kota"
lenOfTekst = len(tekst)
new_txt = tekst + " i dwa psy"
len_of_new_txt = len(new_txt)
```

OK:

```
txt = "Ala ma kota"
len_of_txt = len(txt)
new_txt = txt + " i dwa psy"
len_of_new_txt = len(new_txt)
```

Zasada 3.

Zmienne constowe (stałe), które nigdy nie zmieniają swojej wartości, umieszczaj globalnie i zapisuj WIELKIMI LITERAMI.

Wyjaśnienie:

Chodzi o to, że w momencie, gdy tworzysz zmienną, której wartość będzie taka sama przez cały cykl życia programu, np. MINUTES_IN_HOUR = 60,

WINDOW_WIDTH = 1280, powinieneś umieszczać ją globalnie (może nawet w oddzielnym pliku?) oraz nazywać ją, wykorzystując tylko i wyłącznie wielkie litery.

Przykład:

```
SECONDS_IN_A_DAY = 60 * 60 * 12

def main():
    days = input("Ile dni uczyłeś się programowania?")
    print(f"To aż: {days * SECONDS_IN_A_DAY} sekund!" )

if __name__ == "__main__":
    main()
```

Zasada 4.

Używaj zmiennych, które jasno wyrażają intencje.

Wyjaśnienie:

Dąż do tego, żeby maksymalnie często wypakowywać wartości do odpowiednio nazwanych zmiennych. Dzięki temu zwiększysz łatwość czytania kodu oraz możliwość szybkiego wyszukiwania zależności w kodzie!

Przykład 1

NOK:

```
def main():
    days = input("Ile dni uczyłeś się programowania?")
    print(f"To aż: {days * 60 * 60 * 12} sekund!" )

if __name__ == "__main__":
    main()
```

OK:

```
SECONDS_IN_A_DAY = 60 * 60 * 12

def main():
    days = input("Ile dni uczyłeś się programowania?")
    print(f"To aż: {days * SECONDS_IN_A_DAY} sekund!" )

if __name__ == "__main__":
    main()
```

Przykład 2

NOK:

```
import re

address = "Wrocław 53-225"
city_zip_regex = r"([A-z]+\s?)+\s(\d{2}-\d{3}?)"

matches = re.match(city_zip_regex, address)
if matches:
    print(f"{matches[1]}, {matches[2]}")
```

OK:

```
import re

address = "Wrocław 53-225"
city_zip_regex = r"([A-z]+\s?)+\s(\d{2}-\d{3}?)"

matches = re.match(city_zip_regex, address)
if matches:
    city, zip_code = matches[1], matches[2]
    print(f"{city}, {zip_code}")
```

Zasada 5

Dbaj o poprawne nazywanie zmiennych nawet podczas iteracji w pętli!

NOK:

```
names = ['Kacper', 'Joanna', 'Kinga', 'Stanisław']

print("Moje imiona: ")
for elem in names:
    print(elem)
```

OK:

```
names = ['Kacper', 'Joanna', 'Kinga', 'Stanisław']

print("Moje imiona: ")
for name in names:
    print(names)
```

Zasada 6.

Zrezygnuj z notacji węgierskiej.

Wyjaśnienie:

Notacja węgierska to w programowaniu sposób zapisu nazw zmiennych oraz obiektów, polegający na poprzedzeniu właściwej nazwy małą literą (literami) określającą rodzaj tej zmiennej (obiektu).

W obecnej dobie programowania i łatwości przeskakiwaniu między użyciem a definicją zmiennej, nie jest konieczne używanie opisywanej notacji.

Tak więc poniższy przykładowy zapis:

```
class Human:
    def __init__(self, imie: str, wiek: int):
        self.human_imie = imie
        self.human_wiek = wiek
```

warto zastąpić poniższym:

```
class Human:
    def __init__(self, imie: str, wiek: int):
        self.imie = imie
        self.wiek = wiek
```

Zasada 7.
Używaj adnotacji typów zawsze tam, gdzie to możliwe.

Wyjaśnienie:
Z racji tego, że Python jest językiem dynamicznie typowanym, nie musimy ręcznie określać typów tworzonych zmiennych (np. int, float, str), ponieważ są one **automatycznie dedukowane**.

I to jest jak najbardziej dużą zaletą Pythona, ale z drugiej strony może to prowadzić do ograniczenia czytelności kodu w większych projektach (nie zawsze na pierwszy rzut oka widać, obiektem jakiej klasy jest pewne zmienna).

Dlatego warto wykorzystać możliwość adnotacji typów w Pythonie! Sprawdza się ona szczególnie wtedy, gdy pracujesz w większym zespole i chcesz zadbać o dobrą specyfikację kodu.

I choć, to czy w kodzie korzystamy z adnotowanych typów czy też nie, nie ma wpływu na działanie programu, to dzięki nim zmniejszasz szansę na przesłanie wartości float do, np. funkcji, która oczekuje zmiennej typu int.
Dobre środowisko programistyczne powinno Cię ostrzec przed próbą dokonania takiej czynności (po określeniu odpowiedniej adnotacji, rzecz jasna).

Powiem Ci więcej - z własnego doświadczenia komercyjnego wiem, że w kodzie produkcyjnym prawie zawsze korzysta się z omawianego rozwiązania.

Nie zawsze, aczkolwiek często, aby móc efektywnie dodawać adnotacje do typów, konieczne będzie zaimportowanie biblioteki **typing**! (standard zapewnia adnotację tylko podstawowych typów)

Przykład:

```
def greeting(name: str) -> str:
    return f'Hello {name}'
```

`def greeting(name: str) -> str` -- to sygnatura funkcji przyjmującej zmienną o etykiecie name i oczekiwanym typie tekstowym.
Zwraca ona również zmienną typu str, o czym mówi użyty zapis `... -> str`.

Wykorzystanie bardziej złożonych typów:

```python
from typing import List

def sum_nums(nums: List[int]) -> int:
   return sum(nums)

print(sum_nums([1, 2,3]))
```

Dowolny typ argumentu i brak zwracanych wartości:

```python
def foo(item: Any) -> None:
    ...
```

Czyste funkcje, metody i klasy

Zasada 8.
Ograniczaj ilość argumentów funkcji (metody).

Wyjaśnienie:
Idealna sytuacja dla tworzonej funkcji to taka, w której nie przyjmuje ona żadnego argumentu lub jej lista ogranicza się maksymalnie do dwóch parametrów.
Każda większa ilość powinna zapalać w Twojej głowie lampkę ostrzegawczą i konieczne jest wtedy rozważenie lekkiej modyfikacji programu (zazwyczaj

problem ten rozwiązuje podzielenie funkcjonalności na kilka oddzielnych funkcji lub nową klasę).

Duża ilość argumentów danej funkcji utrudnia przede wszystkim testowanie i może świadczyć o tym, że stworzona funkcja łamie pierwszą zasadę konceptu **S**OLID (Single-responsibility) i jest “a God’s Function” odpowiadającą za realizację kilku zadań.

NOK:

```
def render_window(title, width, height, body, frequency):
  ...
```

OK:

```
class WindowConfig:
    ''' Represents a window config:
        Contains:
        title: The title of the window.
        width, height: The size of the window.
        wody: The displayed content of the window.
        frequency: The frequency that the window refreshes itself with.
    '''

    def __init__(self, config: dict):
        self.title = config["title"]
        self.width, self.height = config["width"], config["height"]
        self.body = config["body"]
        self.frequency = config["frequency"]

def render_window(config: WindowConfig):
   ...
```

Zasada 9.

Preferuj keyword arguments podczas wywoływania funkcji.

Wyjaśnienie:
Czasami jednak może okazać się, że rozbicie danej funkcji z dużą ilością argumentów na kilka mniejszych części lub nową klasę jest niemożliwe i musimy zostać przy stworzonym rozwiązaniu.
Wówczas, wywołując funkcję, używaj **keyword arguments** w celu jasnego definiowania, do którego argumentu w danym momencie się odwołujesz.

Dzięki temu nie będziesz musiał pamiętać, w jakiej kolejności zdefiniowane zostały parametry funkcji oraz nieznacznie poprawisz czytelność jej wywoływania.

NOK:

```
def do_sth(param1, param2, param3):
    ...

do_sth(1, 10, "example")
```

OK:

```
def do_sth(param1: int, param2: int, param3: str) -> None:
    ...

do_sth(param1=1, param2=10, param3="example")
```

Zasada 10.

Funkcje powinny realizować jedno zadanie.

Wyjaśnienie:
O tej zasadzie wspomnieliśmy już przy wcześniejszych przykładach, ale podkreślmy to jeszcze raz - **funkcje powinny być odpowiedzialne za realizację tylko jednego zadania**. Jakiego? Najlepiej takiego, które jasno określa nazwa funkcji.

NOK:

```
import re

users = ["jan.grab@gmail.com", "ania.kowalska@onet.pl",
"excited.girl@yahoo.com", "dg.soldier@o2.pl"]

def email(user: str):
  """Sends an email to the single user"""
  pass

def email_users():
   """Sends an email to the users with the gmail.com domain"""
   for u in users:
       if re.match(r"\S*?@gmail\.com", u):
           email(user)
```

W powyższym przykładzie widzisz, jak funkcja email_users odpowiada za realizację **dwóch funkcjonalności**. Odpowiada ona za przekierowanie procesu wysyłania emaila do poszczególnych użytkowników - `email(user)`, ale również wybierania użytkowników, do których ma zostać wysłana wiadomość: `if re.match(r”\S*?@gmail\.com”, u).`

Śmiało możesz zmodyfikować ją do postaci:

OK:

```
import re
from typing import List

users = ["jan.grab@gmail.com", "ania.kowalska@onet.pl",
"excited.girl@yahoo.com", "dg.soldier@o2.pl"]

def email(user: str):
  """Sends an email to the single user"""
  pass

def get_active_users() -> List[str]:
 """Select users with the gmail.com domain """
 gmail_regex = r"\S*?@gmail\.com"
 return [user for user in users if re.match(gmail_regex, user)]

def email_users():
   """Sends an email to the users with the gmail.com domain"""
   for user in get_active_users():
       email(user)

email_users()
```

Zasada 11.

Stosuj generatory przy dużych zbiorach danych.

Wyjaśnienie:

Zasada ta została omówiona w temacie **Listy Składane. Generatory**, ale dla przypomnienia - w momencie gdy analizujesz duże zbiory danych, w celu zoptymalizowania działania programu - ograniczenia zasobożerności - lepiej będzie wykorzystać generatory niż listy składane.

Zadanie:

Zmodyfikuj przykład z Zasady 10. tak, aby był przetwarzany generator użytkowników, których adres mailowy jest zarejestrowany na domenie gmail.com.

Zasada 12.

Używaj dekoratora **@datalcass** dla klas typowo przechowujących dane.

Wyjaśnienie:

W momencie, gdy Nasza klasa ma być prostą reprezentacją jakiegoś kontenera na dane (np. klasa zawierająca kilka podstawowych funkcjonalności operujących na przechowywanych danych, nieposiadająca metod statycznych), lepszym rozwiązaniem będzie stworzenie, tzw. klasy na dane, aniżeli jej standardowej wersji.

NOK:

```
class Book:
   '''Object for tracking physical books in a collection.'''
   def __init__(self, id:int, name: str, weight: float = 150.0):
       self.id = id
       self.name = name
       self.weight = weight

   def __repr__(self):
       return(f"Book: id={self.id}, name={self.name}, weight={self.weight}")

print(Book(0, "Pan Tadeusz", 350.5))
```

OK:

```python
from dataclasses import dataclass

@dataclass
class Book:
   '''Object for tracking physical books in a collection.'''
   id: int
   name: str
   weight: float = 0

print(Book(0, "Pan Tadeusz", 350.5))
```

Uwaga:

Zauważ ponadto, że klasa ozdobiona dekoratorem @dataclass automatycznie generuje metodę __repr__! (możliwe jest wywołanie metody print() bezpośrednio na obiekcie).

Przykład 2:

Dataklasy przydają się również w momencie, gdy chcesz stworzyć słownik posiadający wartości w postaci zagnieżdżonych struktur (np. słownik w słowniku, lista w słowniku itp.). Wówczas o wiele lepszym i czytelniejszym rozwiązaniem będzie stworzenie klasy przechowującej pola będącymi wypakowanymi z, np. wnętrza słownika, strukturami.

Przykład: Słownik odzwierciedlający zbiór uczniów, do której przyporządkowany jest przedmiot o dowolnej nazwie, a do danego przedmiotu lista ocen, gdzie pojedyncza składa się z wartości i wagi.

NOK:

```
students = {"Kacper" : {"Matematyka" : [(1, 4), (1, 1), (5, 4)]}}
```

OK:

```
from typing import Dict, List
from dataclasses import dataclass

@dataclass
class Grade:
 weight: int
 score: int

@dataclass
class Subject:
 grades: List[Grade]

 def add_grade(self, grade: Grade):
   self.grades.append(grade)

@dataclass
class Student:
 subjects: Dict[str, Subject]

 def add_subject(self, name: str, subject: Subject):
   self.subjects.update({name, subject})
 def average_grade(self):
   pass

best_grade, worst_grade = Grade(4, 6), Grade(4, 1)
math = Subject([best_grade, worst_grade])
student1 = Student({"math" : math})

print(student1)
```

Zasada 13.

Używaj @property zamiast odwoływać się do metod ściśle związanych z polami obiektu.

Wyjaśnienie:
Dodanie dekoratora @property do metody pozwala traktować ją **jak atrybut danego obiektu**. Daje to taką przewagę, że chcąc wyliczać nową zmienną obiektu, bazującą na znanych już wartościach, nie musimy martwić się o wywoływanie odpowiedniej metody określającej rezultat, czy niepotrzebnie wydłużać ciało konstruktora klasy.

NOK:

```
class Celsius:
    def __init__(self, temperature: int = 0):
        self.temperature = temperature

    def to_kelvin(self):
        return self.temperature + 273.0

    def get_temperature(self):
        print("Getting value...")
        return self.temperature

    def set_temperature(self, new_temp: int):
        print("Setting value...")
        if new_temp < -273.0:
            raise ValueError("ERROR")

        self.temperature = new_temp

temperature = Celsius(230)
temperature.set_temperature(-250)
print(temperature.get_temperature())
```

OK:

```
class Celsius:
    def __init__(self, temperature: int = 0):
        self._temperature = temperature

    def to_kelvin(self):
        return self._temperature + 273.0

    @property
    def temperature(self):
        print("Getting value...")
        return self._temperature

    @temperature.setter
    def temperature(self, new_temp: int):
        print("Setting value...")
        if new_temp < -273.0:
            raise ValueError("ERROR")

        self._temperature = new_temp


human = Celsius(37)
human.temperature = 37
print(human.temperature)
```

UWAGA:

1. Zauważ, że aby móc stworzyć metodę w postaci gettera o nazwie temperature(), konieczna była modyfikacja nazwy pola klasy z temperature na _temperature.

2. Zwróć uwagę również na sposób odwoływania się do metod z dekoratorem `@property (human.temperature = 37; print(human.temperature))`

SOLID i efektywne wytwarzanie oprogramowania

Zły design kodu.

Zanim przejdziemy do omówienia pięciu zasad, które znać powinien każdy dobry programista, warto zastanowić się, co zbudowało potrzebę wymyślenia reguł, których to właśnie dotyczy **SOLID**.

Przyznaj się, że nigdy nie chciałbyś doświadczyć sytuacji, w której, pracując z kimś nad wspólnym projektem, zmuszony byłbyś naprawiać błędy wynikające z czyichś nieprzemyślanych kroków.
Lub, co gorsza, napotkać się na tak sztywny kod, że wprowadzenie jakichkolwiek rozszerzeń czy modyfikacji, powodowałoby wysyp licznych błędów i nieoczekiwanych zachowań programu.

Rozwijając myśl, źle napisany kod możemy powiązać z następującymi symptomami:

- **Rigidity (Sztywność)**
- **Fragility (Kruchość)**
- **Immobility (Nieruchomość)**

Rigidity (Sztywność) - projekt jest sztywny, wtedy gdy nie można go modyfikować. Wynika to z faktu, że pojedyncza zmiana w kodzie powoduje konieczność modyfikacji X innych zależnych miejsc/modułów.

Fragility (Kruchość) - Cecha ta jest niejako związana ze sztywnością kodu. Wtedy gdy uda nam się jakimś cudem wprowadzić zmiany, pojawia się duże ryzyko, że kod się rozsypie i jego naprawa będzie wymagała wielu zmian, których nie dało się przewidzieć. Obniża to znacznie zaufanie względem projektu, gdyż każdy “fix” może powodować kolejną lawinę błędów z nieoczekiwanej strony.

Immobility (Nieruchomość) - Pisząc aplikacje, powinniśmy dążyć do tego, by tworzyć jak najbardziej uniwersalne klasy i funkcjonalności.
Takie podejście umożliwi Nam łatwe przenoszenie modułów z jednego projektu do drugiego (gdy nastąpi taka konieczność).
Dzięki temu możliwa jest łatwa reimplementacja stworzonej już funkcjonalności i przemieszczenie jej do nowo tworzonego projektu. Znacząco zyskujemy wtedy na czasie i kosztach wytwarzania.

SOLID jest niejako remedium na wszystkie bolączki programistów i wyżej wymienione problemy. Przestrzeganie zasad, których opisywany mnemonik dotyczy, umożliwi tworzenie efektywnego, łatwo rozwijanego i działającego oprogramowania!

Czym są zasady SOLID?

Zasady SOLID zostały opracowane przez wcześniej wspomnianego przeze mnie **Roberta C. Martin** znanego jako **“uncle Bob”**. W swojej publikacji **Design Principles and Design Patterns** zawarł on najważniejsze zasady **OOD** (object-oriented design), którymi powinien kierować się dobry programista.

Jednym słowem, SOLID buduje jasne reguły, które umożliwiają unikanie, tzw. śmierdzącego i niestabilnego kodu, idą w myśl **manifestu programowania zwinnego Agile** oraz **Adaptacyjnego tworzenia oprogramowania**.
Umożliwiają sprawne wytwarzanie aplikacji działających na najwyższym poziomie wymagań klienta.

SOLID to w rozwinięciu:
S - **Single Responsibility** principle (Zasada jednej odpowiedzialności)
O - **Open/Closed principle** (Zasada otwarte/zamknięte)
L - **Liskov Substitution principle** (Zasada podstawienia Liskov)
I - **Interface Segregation principle** (Zasada segregacji interfejsów)
D - **Dependency Inversion Principle** (Zasada odwrócenia zależności)

- **Single Responsibility Principle:**

Zasada mówi: ***“nigdy nie powino być więcej niż jednego powodu do istnienia klasy bądź metody”***

Wyjaśnienie:
Jeżeli uważnie prześledziłeś wcześniej wymienione zasady czystego kodu, na pewno pamiętasz o tym, jak wspomniałem, że funkcja powinna realizować tylko jedno zadanie. Reguła ta ma swoje uzasadnienie właśnie w pierwszej zasadzie SOLID.

Zgodnie z tym, co podaje Martin - aby lepiej zrozumieć omawianą zasadę, możemy rozważyć **jeden moduł generujący** i **drukujący raport**.
Odpowiada on za dwie funkcjonalności - dwa procesy. Zmiana dowolnego z nich prowadzi do konieczności modyfikacji całego modułu. Rozwiązaniem jest wówczas wydzielenie dwóch niezależnych modułów komunikujących się ze sobą **za pomocą wspólnej klasy**.

Przykład:

```
from dataclasses import dataclass

@dataclass
class Raport:
 content: str
 date: str

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

```
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.")
```

- **Open/Closed principle (zasada otwarte/zamknięte):**

Zasada ta mówi: ***"Klasa (lub metoda) powinna być otwarta na rozszerzenia, ale zamknięta na modyfikacje"***

Wyjaśnienie:

Oznacza to, że powinniśmy dążyć do pisania takiego kodu, który umożliwi Nam dodanie nowych funkcjonalności bez naruszenia już istniejących.
Dzięki temu kod jest niczym stacja, do której możemy dodawać kolejne rozszerzenia, nie naruszając już istniejących.

Nie ukrywam, że tworzenie kodu O/C wymagać będzie od Ciebie nowego podejścia do pisania programów - słowem wstępu, zapomnij o ifologii kierującej logiką programu!

NOK:

```
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()
```

Zauważ, że po wprowadzeniu drugiego sposobu, możliwe jest łatwe rozszerzanie menu wyboru o kolejne opcje.
Nie musimy bowiem ingerować w istniejącą logikę i modyfikować żadnych instrukcji warunkowych, tylko wystarczy dodać odpowiednie **referencje do metod z klasy Executor w słowniku options**. Zmiana wówczas ogranicza się tylko do kilku słów, np.

`self.options = {1:self.executor.do_sth_one, 2:self.executor.do_sth_two, 3:self.executor.do_sth_three, 4:self.executor.do_sth_new}`

Zamiast

```
if choice == 1:
      self.executor.do_sth_one()
    elif choice == 2:
      self.executor.do_sth_two()
    elif choice == 3:
      self.executor.do_sth_three()
    elif choice == 4:
      self.executor.do_sth_new()
    else:
```

```
    self.show_error()
```

- **L** - Liskov Substitution Principle

Zasada mówi:

"Jeżeli S jest klasą dziedziczącą z klasy T, to obiekty klasy T mogą zostać zastąpione obiektami klasy S bez zaburzania poprawności działania programu."

Wyjaśnienie:
Utworzenie właściwej hierarchii klas powinno się objawiać tym, że jeżeli w funkcjonalności programu wszystkie obiekty dowolnego rodzica podmienilibyśmy obiektami jego dziecka, to **nie powinniśmy zauważyć żadnej różnicy** w działaniu programu.

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).

Chyba najczęściej przytaczanym przykładem klas łamiących zasadą substytucji Liskov jest przykład **Prostokąta i dziedziczącego po nim Kwadratu**.

Z matematycznego punktu widzenia taka zależność jest jak najbardziej prawidłowa (każdy kwadrat jest przecież prostokątem).
Pytanie jednak, czy faktycznie w rozumieniu programistycznym stworzenie takiej zależności będzie prawidłowe?

OK

```
from dataclasses import dataclass

@dataclass
class Rectangle:
 width: float
 height: float

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

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

class Square(Rectangle):
 def __init__(self, size: float):
   super().__init__(size, size)



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 200, should return 400
```

Problemem w powyższym kodzie jest linia **square.increase_width(10)**. Otóż, w momencie, gdy próbujemy zwiększyć szerokość obiektu będącego kwadratem, zgodnie z zasadami, powinien on równocześnie zwiększać swoją wysokość.

Z racji jednak, że dziedziczy on metodę increase_width po klasie Rectangle, aby zapewnić realizację tego założenia, konieczne by było przeładowanie metody increase_width w klasie Square tak, aby równocześnie zwiększała wysokość jak i szerokość figury.

Zgodnie z tym, co zostało wspomniane w wyjaśnieniu trzeciej zasady SOLID, byłoby to niepożądane działanie i łamałoby podstawowe zasady efektywnego kodu (zmodyfikowanie działania odziedziczonej metody).

Powyższy przykład można zamienić na:

```
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
```

- **I - 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.

Czym jest interfejs?

Interfejs to typ podobny do klasy. W zasadzie w Pythonie różnica między klasą a interfejsem jest dość cienka, ale ogólnie wszystko sprowadza się do tego, że interfejs mówi Nam tylko o tym, **jakie metody muszą znaleźć się w klasie** po nim dziedziczącej. To, jak dana metoda ma się zachowywać, określane jest już z poziomu klasy dziedziczącej.

I choć o tym nie wspomniałem, to właśnie klasa Shape z wcześniejszego przykładu jest niejako interfejsem:

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

 def increase_width(self):
   raise NotImplementedError
```

Zauważ, że definiuje ona jedynie metody, jakie będą dziedziczone przez potomków (def area).

Natomiast to, co mają one realizować, zależy już od klas dziedziczących. Jeżeli nie spełnilibyśmy tego założenia, wówczas po wywołaniu dowolnej metody, zostałby rzucony **wyjątek NotImplementedError**.

Inny sposób na tworzenie interfejsu w Pythonie:

Z racji, że w standardzie Pythona nie jest jasno zdefiniowane, jak tworzyć interfejs (można to zrobić tak, jak we wcześniejszym przykładzie) lub można posłużyć się dodatkowymi modułami jak, np. **zope.**

Spójrz na przykład:

```
from zope.interface import Interface, Attribute, implementer

class IEmployee(Interface):

   name = Attribute("Name of employee")

   def do(work):
       """Do some work"""

@implementer(IEmployee)
class Employee:
 name = 'Anonymous'

 def do(self):
   print("I am a common employee and doing just some stuff...")

@implementer(IEmployee)
class Manager():
 def do(self):
```

```
    print("...")

employee = Employee()
employee.do()

manager = Employee()
manager.do()
```

- **D - Dependency Inversion Injection**

Ostatnia, ale równie ważna, co pozostałe, reguła wytwarzania efektywnego programu. Mówi:

1. *Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Wszystkie powinny zależeć od abstrakcji.*
2. *Abstrakcje nie powinny zależeć od szczegółów. To szczegóły powinny zależeć od abstrakcji.*

Wyjaśnienie:
Wspominając o abstrakcjach, mamy przede wszystkim na myśli interfejsy oraz klasy abstrakcyjne dla danego dziecka (**formalnie różnica między interfejsem a klasą abstrakcyjną** jest taka, że interfejs dostarcza tylko sygnatury danych metod, a klasa abstrakcyjna oprócz sygnatur może również implementować funkcjonalności metod, które będą wspólne dla wszystkich klas dziedziczących; w Pythonie aczkolwiek interfejs/klasa abstrakcyjna tak naprawdę **oznaczają to samo**).

Tak więc realizacja założeń powyższej zasady sprowadza się do tego, by często bazować na tzw. **kompozycji modułów**.

Przykład:

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

class TaskManager:
 def process(self):
   app = GameApplication()
   app.run()
```

Poprawiony przykład:

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


class TaskManager:
 def __init__(self, app: GameApplication):
   self.app = app

 def process(self):
   self.app.run()
```

Wzorcowy przykład:

```
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()
```

UWAGA:

Akurat w tym przykładzie w celu stworzenia klasy abstrakcyjnej, skorzystaliśmy z **modułu abc**. Równie dobrze moglibyśmy w tym przypadku stworzyć interfejs, tak jak robiliśmy to we wcześniejszych sekcjach przy użyciu biblioteki zope.

Jak widzisz, sposobów na realizację tego samego celu jest wiele, a to głównie z powodu tego, że nie jest jasno zdefiniowane w standardzie Pythona, jak tworzyć klasy abstrakcyjne/interfejsy. Pełna dowolność :)
Dlatego też staram się przedstawić Ci jak najwięcej wariantów na rozwiązanie jednego problemu.

Zauważ, że dzięki rozwiązaniu problemu przez kompozycję (zawieranie obiektu jednej klasy w drugiej), nie uzależniamy k**lasy Task od obiektu jednej konkretnej klasy**.
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).