



# Wzorce Projektowe I Dobre Praktyki



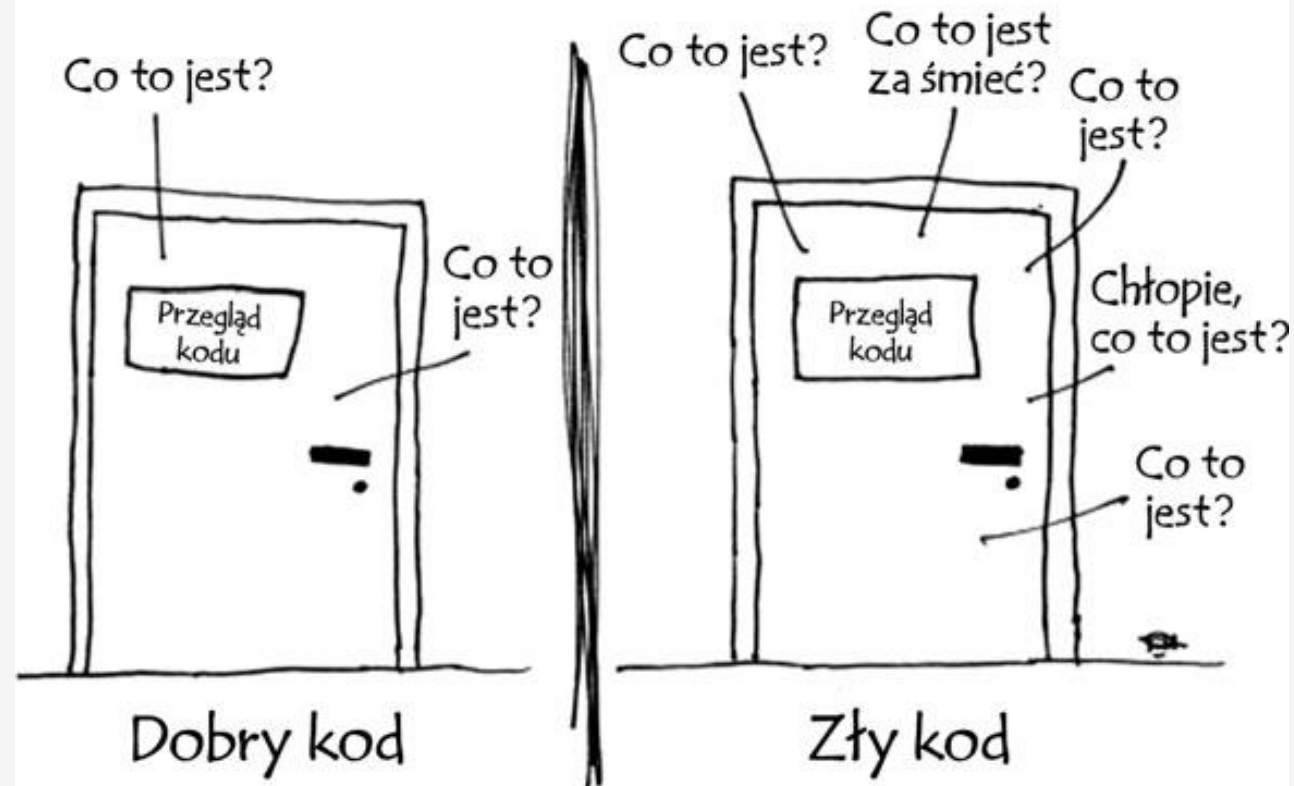
- Czysty kod
- PEP
- Projektowanie obiektowe
  - Wstęp
  - SOLID
  - KISS, DRY, YAGNI
  - GRASP



# CZYSTY KOD



Jedyna prawidłowa miara  
jakości kodu: Co to jest/minutę



Źródło: Czysty kod. Podręcznik dobrego programisty. R.C.Martina



Definicja czystego kodu wg najlepszych programistów:

**Bjarne Stroustrup – twórca języka C++:**

*„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 kusilo psucie kodu w celu wprowadzenia niepotrzebnych optymalizacji. Czysty kod wykonuje dobrze jedną operację.”*

**Grady Booch:**

*„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.”*

**Ron Jeffries:**

*„W podanej kolejności prosty kod:*

- przechodzi wszystkie testy;*
- nie zawiera powtórzeń;*
- wyraża wszystkie idee projektowe zastosowane w systemie;*
- minimalizuje liczbę encji, takich jak klasy, metody, funkcje i podobne.”*



Zasada skautów:

*„Pozostaw obóz czyściejszym, niż go zastałeś”*

Co to oznacza?

- Nie zawsze opłaca się poświęcać kilku dni pracy na samo oczyszczanie kodu,
- Najlepiej jest robić to stopniowo,
- Jeżeli dopisujesz kod do pewnego pliku, spraw, by pozostał on czystszy również w części, która nie została przez Ciebie napisana.
- Wystarczy zmienić niepoprawną nazwę zmiennej, funkcji, uprościć algorytm, usunąć zbędną instrukcję if, zagnieżdżone niepotrzebnie pętle
- W przypadku, gdy każdy będzie coś poprawiał po trochu, kod będzie utrzymywany na właściwym poziomie
- Powyższe sugestie to podręcznikowy ideał



Co robią te dwie funkcje?

```
def x(a):  
    return (a - 32) * 5 / 9
```

```
def y(b):  
    return (b * 9 / 5) + 32
```



## Brak:

- Poprawnych nazw jasno określających, do czego powinny być użyte funkcje,
- Dokumentacji (docstringów)
- Type annotation (\* to już jako miły dodatek)

```
def x(a):  
    return (a - 32) * 5 / 9
```

```
def y(b):  
    return (b * 9 / 5) + 32
```





To są te same funkcje, co na poprzednich slajdach.

Czy nie jest teraz wszystko jasne?

```
def convert_fahrenheit_to_celsius(fahrenheit_degrees):  
    return (fahrenheit_degrees - 32) * 5 / 9
```

```
def convert_celsius_to_fahrenheit(celsius_degrees):  
    return (celsius_degrees * 9 / 5) + 32
```



Co jest nie tak z tymi nazwami?

1. `d = 10`

2. `x = person_data[3]`

3. `get_active_account()`

`get_active_accounts()`

`get_active_account_info()`

4. `imnawiwapr = ("Andrzej", "Kowalski", 50, 95, "bezrobotny")`

5. `for element in lista:`

`if element == 5:`

`...`

# Czysty kod – znaczące nazwy



Co jest nie tak z tymi nazwami?

```
1. d = 10

2. x = person_data[3]

3. get_active_account()
   get_active_accounts()
   get_active_account_info()

4. imnawiwapr = ("Andrzej", "Kowalski", 50, 95, "bezrobotny")

5. for element in lista:
    if element == 5:
        ...
```

1. Zmienna `d` nic nie mówi o przechowywanej wartości
2. Zmienna `x` nic nie mówi o przechowywanej wartości plus nie wiadomo, co wyciągamy ze zmiennej `person_data`; liczba 3 powinna być stałą nazwaną w odpowiedni sposób, np.  
`WIEK = 3`
3. Zbyt podobne nazwy funkcji, ciężko się domyślić, czym się różnią nie patrząc w ich kod
4. Ciężka do wymówienia i zapamiętania nazwa zmiennej (im od imię, na od nazwisko, wi od wiek, wa od waga, pr od praca)
5. Nie wiadomo czym jest 5, dlaczego akurat przyrównujemy element do niej, nazwy `element` i `lista` też nie są najlepsze



Co to znaczy napisać dobrą funkcję?


- funkcja powinna działać i spełniać postawione wymagania
- dobra funkcja to prosta funkcja, czyli taka, która robi tylko jedną rzecz i nic więcej (rozwiązuje jeden problem)
- prosta funkcja to również taka, która ma możliwie jak najmniejszą liczbę argumentów (najlepiej: zero)
- taka funkcja nie powinna posiadać argumentów wyjściowych (tylko wejściowe)
- powinna być w miarę krótka: celujemy w kilka-kilkanaście linii
- funkcja powinna być prosta do przetestowania
- funkcja z właściwą nazwą swoją i argumentów

# Czysty kod – funkcje



Najlepsza funkcja jest bezargumentowa lub posiada jeden argument. W wielu książkach uznaje się trzyargumentową funkcję za skrajny przypadek. W przypadku jeszcze większej liczby argumentów do funkcji, najprawdopodobniej można to rozwiązać inaczej.

```
def multiply_fractions(first_nominator, first_denominator,
                       second_nominator, second_denominator):
    new_nominator = first_nominator * second_nominator
    new_denominator = first_denominator * second_denominator
    return new_nominator, new_denominator
```



```
class Fraction:
    def __init__(self, nominator, denominator):
        self.nominator = nominator
        self.denominator = denominator
    def multiply(self, fraction):
        nominator = self.nominator * fraction.nominator
        denominator = self.denominator * fraction.denominator
        return Fraction(nominator, denominator)
```



Zgodnie z zasadą, nazwy funkcji powinny być czasownikami lub wyrażeniami posiadającymi czasownik. Najlepsze są funkcje, które wraz ze swoim(i) argumentem(ami) tworzą czytelną parę czasownik-rzeczownik.

## OK.:

```
def write(file):  
    ...  
  
def write_to(file):  
    ...  
  
def play(game):  
    ...
```

## NIE OK.:

```
def file(path):  
    ...  
  
def computer_started():  
    ...  
  
def doctor_heals(patient):  
    ...
```



Funkcje powinny wykonywać tylko jedną czynność. Dzięki temu funkcje są krótkie, czytelne i łatwiej nimi zarządzać. Jest to związane poniekąd również z tym, że najczęściej nazwa funkcji jest powiązana tylko z jedną akcją wykonywaną przez funkcję – każda kolejna będzie „ukryta” w jej ciele.

Przykład złej funkcji:

```
def print_each_element(elements):  
    for element in elements:  
        print(element)  
        elements.remove(element)
```

Funkcja `print_each_element` oprócz wypisania na terminalu każdego elementu z listy, dokonuje również cichego usunięcia połowy z nich. To niedopuszczalne!



Stanowczo złą praktyką jest używanie w funkcjach parametrów typu bool. Funkcje typu:

```
def kiss_or_ride(should_kiss):  
    if should_kiss:  
        ...  
    else:  
        ...  
  
kiss_or_ride(True)
```

są złe, ponieważ parametr boolowski od razu sugeruje, że funkcja może wykonywać dwie czynności (dwie ścieżki) po jednej na flagę ustawioną na `True` i `False`.

W tym przypadku lepiej rozdzielić funkcję `kiss_or_ride()` na dwie: `kiss()` i `ride()`.





# BURZA MÓZGÓW: PO CO NAM KOMENTARZE?



Według R.C. Martina używanie komentarzy jest próbą zniwelowania błędów popełnionych przez programistę w trakcie pisania kodu.

Zanim napiszemy komentarz, należy się zastanowić, czy faktycznie jest on nam koniecznie potrzebny, czy nie dałoby się tego samego wyrazić za pomocą kodu.

Problemem/niepotrzebnością w kodzie są:

- niepotrzebne informacje (takie jak historia zmian, autor, data ostatniej modyfikacji – to nie jest miejsce na takie rzeczy, po to jest GIT!)
- przestarzałe komentarze
- nadmiarowe komentarze
- zakomentowany kod



## Niepotrzebne informacje w kodzie

```
1  """
2      -----
3      Date of origin: 26-01-2017
4      Author: Jan Kovalsky
5      -----
6      Name of file: RequestHandler.py
7      First commit: 27-01-2017
8
9      History:
10     22-02-2017 - added respond_alert method
11     04-04-2017 - removed bug, now objects are always created
12     ...
13  """
14
15
16  class RequestHandler:
```



## Przestarzałe komentarze

```
def add_new_member(self, member):  
  
    db = self.get_db()  
    db.add_member(member)  
    db.approve()  
  
    # TODO: VERIFY ADDITION SUCCEEDED  
    #     ONCE METHOD WILL BE IMPLEMENTED
```

1

```
def add_new_member(self, member):  
  
    db = self.get_db()  
    db.add_member(member)  
    db.approve()  
    db.verify_addition_succeeded()  
    # TODO: VERIFY ADDITION SUCCEEDED  
    #     ONCE METHOD WILL BE IMPLEMENTED
```

2



## Nadmiarowe komentarze

```
def add(a, b):  
    return a + b # same as b + a
```



## Zakomentowany kod

```
def add_new_member(self, member):  
    db = self.get_db()  
    db.add_member(member)  
    # Shouldn't be done this way anymore  
    # db.approve()  
    # db.verify_addition_succeeded()  
    # TODO: VERIFY ADDITION SUCCEEDED  
    # ONCE METHOD WILL BE IMPLEMENTED
```



Jeżeli po dłuższym kodzeniu okaże się, że nasz kod jest trudny do zrozumienia i najdzie nas ochota skomentowania go, to jest to tylko pozorne naprawienie problemu. Należy ten kod poprawić, a nie komentować!

```
// Sprawdzenie, czy człowiek ma prawo jeździć autem  
if human.driving_license and NO_ALCOHOL and human.age > 18:  
    ...
```

Lepiej powyższą instrukcję warunkową zamienić na niewymagającą już komentarza:

```
if human.is_able_to_drive_a_car():  
    ...
```



Przykłady dobrych komentarzy:

- komentarze TODO (do usunięcia od razu po dostarczeniu implementacji!)
- komentarze prawne (tzw. copyrighty)
- komentarze wyjaśniające DLACZEGO coś się dzieje w kodzie, a nie CO się dzieje w kodzie
- komentarze ostrzegające o pewnych nieoczywistych konsekwencjach





Jedna z najważniejszych sentencji dotyczących się komentowania:

„Prawda zawsze jest po stronie kodu!”

Nieistotne jak bardzo trafny i słuszny byłby komentarz, jeżeli kod będzie zły, to program nie zadziała jak powinien. Pielęgnować powinno się listing, nie nasze zakomentowane „opowiadania”.



**Python jest językiem wymuszającym poprawne formatowanie poprzez wprowadzenie wymagania stosowania wcięć w przypadku tworzenia bloków kodu i zakresów.**

Formatowanie jest sprawą dosyć istotną (zresztą nie tylko ono). Takie rzeczy jak nazewnictwo zmiennych, odstępy między blokami kodu w funkcjach, długość linii itd., powinno zostać ustalone przez programistów danego projektu.

Raz ustalony standard i zasady respektować powinien każdy pracujący nad danym projektem



Dobra klasa to taka, która:

- realizuje tylko jedno zadanie
- jest otwarta na dziedziczenie, zamknięta na modyfikacje
- jest spójna
- charakteryzuje się niskim powinowactwem z innymi klasami
- zawiera funkcje napisane zgodnie z zasadami
- w jej skład wchodzi „skończona” liczba metod (maksymalnie kilkanaście)
- posiada tylko pola, z których rzeczywiście korzysta



Zgodnie z podstawowymi paradygmatami programowania obiektowego, obiekty powinny ukrywać swoje atrybuty, a udostępniać metody do zarządzania polami w obiekcie. Struktura obiektu nie może być przezroczysta i udostępniana każdemu innemu obiektowi, ponieważ spowoduje to złamanie paradygmatu hermetyzacji.

*Prawo Demeter* (nie rozmawiaj z nieznajomymi!) – metoda *m* w klasie *A* może wywołać tylko metody z:

- swojej klasy (tutaj: z *A*)
- obiektu, którego ta metoda utworzyła
- obiektu, który został dostarczony do metody *m* jako parametr
- obiektu będącego jednym z atrybutów obiektu klasy *A*



## Przykład złamania prawa Demeter:

```
person = census.get_country("Poland") \
           .get_city("Cracow") \
           .get_district("Stare Miasto") \
           .get_place("Rynek Glowny") \
           .get_restaurant("XYZ") \
           .get_manager()
```



Wróć do swoich poprzednich programów i zastosuj w nich dobre praktyki omówione dotychczas. Zasady dotyczące: nazewnictwa, funkcji, komentarzy, klas.



# **PEP – Python Enhancement Proposals**



- Wcięcia (4 spacje) i puste linie
- Długość linii
- Operator na początku linii przy wcześniejszym jej złamaniu
- Odstęp między klasami i funkcjami
- Łańcuchy znaków
- Formatowanie komentarzy
- Importowanie modułów





- UpperCamelCase
- lowerCamelCase
- snake\_case
- alllowercase
- ALLCAPS
- ALLCAPS\_WITH\_UNDERSCORES



- UpperCamelCase – nazwy klas, nazwy typów, wyjątki
- lowerCamelCase
- snake\_case – funkcje, zmienne globalne, metody, metody statyczne, argumenty funkcji i metod
- alllowercase – nazwy modułów
- ALLCAPS
- ALLCAPS\_WITH\_UNDERSCORES – stałe w kodzie



- Argumenty self, cls w metodach
- Formatowanie wyrażeń matematycznych o złożonej strukturze
- Underscore prefix, Underscore suffix
- Małe L, litera O jako zmienne o pojedynczych nazwach
- Białe znaki



- PEP to tylko rekomendacja
- Wyjątki: wsteczna kompatybilność, bezpośredni port bibliotek niskopoziomowych
- Kompatybilność przede wszystkim z obecnym projektem
- Z reguły PEP to punkt wyjściowy dla wewnętrznych firmowych regulacji

<https://www.python.org/dev/peps/pep-0008/>

# Okiem praktyka – zbiór reguł własnych



- Funkcje i metody nie dłuższe niż 20 linii
- Nazwa każdej zmiennej ma znaczenie
- Samotłumaczące nazwy zmiennych /metod / klas są lepsze niż komentarz
- Jeśli wchodzisz w piąty stopień wcięcia, prawdopodobnie robisz coś źle
- Jeśli Twoja klasa ma \*-dziesiąt metod, prawdopodobnie robisz coś źle
- W rozbudowanych programach zdecydowanie lepiej postawić na komunikację „międzyobiekтовую” niż „międzyfunkcyjną”

# Okiem praktyka – zbiór wybranych reguł w Pythonie



- Nie porównujemy wartości zmiennych logicznych do True i False: **if valid:** a nie **if valid == True:**
- Używamy kontekstu logicznego (np. pusta lista ma wartość False): **if lista\_prac:** a nie **if len(lista\_prac) > 0:**
- Używamy **is not** zamiast **not ... is** w instrukcji if: **if x is not None:** a nie **if not x is None**
- Nie używamy **if x:** jeśli mamy na myśli **if x is not None**
- Używamy **.startswith()** i **.endswith()** zamiast slicing'u



Otwórz plik `dobre_praktyki.py` z repozytorium. Popraw kod tak, żeby był on czytelny i spełniał wymogi PEP.



Wróć do swoich poprzednich programów i zastosuj się w nich do zaleceń PEP.





- Istnieje olbrzymia ilość formatterów i automatycznych upiększaczy kodu
- pylint – biblioteka analizująca kod, pomaga go ulepszyć zgodnie z przyjętymi zasadami (można jest zmieniać wedle swoich zasad/upodobań)
- spora ilość oczyszczaczy kodu online (wystarczy wpisać „python clean code online” w Google)
- black – biblioteka formatująca automatycznie kod z pliku w miejscu (!), nie jest tworzona kopia, plik jest naprawiany i stary format znika bezpowrotnie



Zainstaluj bibliotekę black za pomocą narzędzia pip.  
Uruchom ją na pliku dobre\_praktyki.py, sprawdź jej działanie i oceń przydatność.



# **PROJEKTOWANIE OBIEKTOWE**



- *Klasa, obiekt, moduł, pakiet*
- *Funkcja, metoda, metoda statyczna*
- *Interfejs, klasa bazowa, klasa pochodna*
- *Hermetyzacja, dziedziczenie, polimorfizm*



Napisz klasę `Note`, reprezentującą notatkę z listą tagów, notatkę stringową i datą utworzenia. Metoda `match(filter: str) : bool` powinna przeszukiwać zawartość notatki oraz listę tagów i zwracać `True` lub `False`, w zależności od tego czy znalazło podany string. Notatki powinny mieć unikalny numer zaczynając od 1. Do wyznaczania daty utworzenia użyj modułu `datetime`.

Atrybuty klasy: `id_ : int`, `creation_date : datetime`, `tags : list`,  
`memo : str`



**DALSZA CZĘŚĆ:** napisz klasę Notebook która gromadzi w swojej liście obiekty typu Note i zawiera metody:

`add_new_note(memo: str, tags: list): Note` tworzy nową notatkę o podanej zawartości memo oraz liście tagów tags i dodaje do listy notes,

`modify_memo(note_id: int, memo: str)` nadpisuje zawartość notatki o numerze note\_id podanym stringiem memo,

`modify_tags(note_id: int, tags: list)` analogicznie jak powyżej tylko, że nadpisuje tagi,

`search(filter: str): List[Note]` zwraca listę notatek, dla których wywołanie ich metody `match` z argumentem `filter` zwróciło `True`



Zastanówmy się chwilę, czego klient może wymagać od naszej aplikacji (ogólnie)?



Zastanówmy się chwilę, czego klient może wymagać od naszej aplikacji (ogólnie)?

Aplikacja powinna:

- działać
- działać długo
- być w stanie uaktualniona





Jak to wygląda z punktu widzenia programisty?

Tworzony kod powinien:

- być elastyczny
- być przeznaczony do wielokrotnego użytku (tak, żeby mógł być użyty w innym projekcie)



## Uproszczony schemat tworzenia oprogramowania:

1. Idea
2. Projektowanie
3. Programowanie
4. Testowanie
5. Utrzymywanie

Jedyną pewną rzeczą są zmiany! Dlatego tak ważne jest odpowiednie zaprojektowanie aplikacji!



Czy dziedziczenie to jedyne zjawisko relacji między obiektami?

Dziedziczenie to relacja „jest”, oprócz tego w programowaniu istnieje jeszcze relacja „ma”.

1. Delegacja – przekazywanie wykonania zadania innej klasie/metodzie (np. porównywanie obiektów)
2. Kompozycja – w naszej klasie (i tylko w niej) istnieje dodatkowy obiekt innego typu w postaci atrybutu (relacja całość-część), np. Budynek<-Pomieszczenie
3. Agregacja – podobnie jak kompozycja, ale obiekt może istnieć również na zewnątrz naszej klasy, np. KatalogBiblioteczny<-KartaKsiążki



1. Korzystanie z interfejsów i klas abstrakcyjnych
2. Spójność – cecha która powinna charakteryzować nasze klasy:  
wysoka spójność oznacza silną interakcję wewnątrz napisanej klasy,  
a mniejszą zależność od klas znajdujących się „obok”; dążymy do  
jak największej spójności klas lub modułów
3. Zasady SOLID, GRASP, DRY, KISS, YAGNI



- **S**ingle responsibility principle – zasada pojedynczej odpowiedzialności
- **O**pen/closed principle – zasada otwarte-zamknięte
- **L**iskov substitution principle – zasada podstawienia Liskov
- **I**nterface segregation principle – zasada segregacji interfejsów
- **D**ependency inversion principle – zasada odwrotnej zależności

*By Robert C. Martin*

# SOLID - Single responsibility principle



- *Stan:*  
*Sporej wielkości klasa, posiadająca kilka funkcjonalności.*
- *Problem:*  
*Zbyt rozbudowane, trudne do utrzymania klasy*
- *Rozwiązanie:*  
*Rozbicie monolitu na mniejsze jednostki. Klasa powinna mieć jeden powód do zmiany!*

# SOLID - Single responsibility principle



Które z metod przypisanych do danych obiektów mają sens?

`rower.napraw()` (się)

`piekarnik.podaj_temperature()`

`pies.szczekaj()`

`morze.zwoduj_statek()`

`dziura.zaklej()` (się)

`prysznic.lej_wode()`

`kaczka.prowadz_samochod()`

## SOLID - Single responsibility principle



Przejrzyj plik `srp.py` z katalogu `SOLID/bad` i spraw, by zawarty w nim kod spełniał wymogi zasady single responsibility.



# SOLID - Single responsibility principle



- *Problem praktyczny:*

*Po złożeniu zamówienia przez klienta generowana jest faktura i wysyłana jest wiadomość email potwierdzająca złożenie zamówienia.*

*Jak powinna wyglądać struktura klas/modułów?*

# SOLID - Open/closed principle



- *Stan:*  
*Zespół często dokonuje poprawek w kodzie, ruszając i modyfikując stabilny i działający kod.*
- *Problem:*  
*Częste modyfikacje kluczowych części kodu zagrażają stabilności systemu*
- *Rozwiązanie:*  
*Implementacja modyfikacji poprzez nadpisanie części funkcjonalności bez zmian w oryginalnej części programu . Rozbudowa bez zmiany aktualnego kodu.*

# SOLID - Open/closed principle – zmienne globalne



- *Stosowanie zmiennych globalnych jest sprzeczne z zasadą open/closed!*
- *Klasy/moduły powinny być otwarte na rozszerzanie, ale zamknięte na modyfikacje.*
- *Jeżeli jakakolwiek klasa czy moduł korzystają ze zmiennej globalnej, to nigdy nie będą zamknięte – zmienne globalne mają to do siebie, że każdy może zmienić ich wartość, przez co może zostać zmieniony sposób pracy naszej klasy/modułu.*



Przejrzyj plik `open_closed.py` z katalogu `SOLID/bad` i spraw, by zawarty w nim kod spełniał wymogi zasady open/closed.

# SOLID - Open/closed principle



- *Problem praktyczny:*

*Musimy rozszerzyć naszą funkcjonalność zamówień o płatności online:*

- *PayPal*
- *Stripe*
- *DotPay*
- *BitCoin*

*Jaki jest status płatności, jaki jest status zamówienia? Czy zamówienie zostało opłacone, wysłane, dostarczone? Czy klient był zadowolony?*

# SOLID - Liskov substitution principle



- *Stan:*  
*Częste rozbudowywanie kodu o klasy dziedziczące. Nadmiar dziedziczenia.*
- *Problem:*  
*Po implementacji kilku wariantów, musimy wykrywać z jaką klasą pracujemy*
- *Rozwiązanie:*  
*Klasy pochodne muszą w pełni implementować interfejs klasy bazowej*

# SOLID - Liskov substitution principle



*WNIOSEK: Każda klasa pochodna powinna móc być podstawiona w miejsce klasy bazowej!*

*Wobec tego, czy taki kod jest w porządku?*

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_distance_to_zero_coordinates(self):
        ...

class Point3D(Point2D):
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z

    def get_distance_to_zero_coordinates(self):
        ...
```



Przejrzyj plik `liskov.py` z katalogu `SOLID/bad i spraw`, by zawarty w nim kod spełniał wymogi zasady podstawienia Liskov.



# SOLID - Interface segregation principle



- *Stan:*

*Klasy abstrakcyjne, interfejsy o olbrzymich rozmiarach.*

- *Problem:*

*Implementacja wszystkich metod wymaganych przez interfejs zajmuje dużo czasu, część zaimplementowanych metod jest nieużywana. Rozszerzenie interfejsu wymusza szeroką rozbudowę elementu.*

- *Rozwiązanie:*

*Rozbij interfejs na wiele małych elementów. Istnienie wielu pomniejszych interfejsów jest lepsze niż istnienie jednego potężnego.*

# SOLID - Interface segregation principle



- *Interfejsy są aplikowane do klasy na podstawie kontraktu metod (często abstrakcyjnych), które dana klasa musi zdefiniować na podstawie interfejsu.*
- *Z interfejsami pythonowymi związany jest termin o nazwie duck typing.*
- *Główną ideą kaczego typizowania jest przyjęcie, że obiekt tak naprawdę jest reprezentowany przez metody jakie ten obiekt posiada; to metody są esencją obiektu, nie jego nazwa, docstring czy atrybuty.*
- *„Jeśli chodzi jak kaczka i kwacze jak kaczka to jest to kaczka.”*

# SOLID - Interface segregation principle



- *Problem praktyczny:*

*W naszym zamówieniu mogą się znajdować produkty spożywcze, alkoholowe, cyfrowe, według indywidualnej specyfikacji klienta, inne (standardowe). Jak je zaimplementować?*

# SOLID - Dependency inversion principle



- *Problem:*

*Zmiana w klasie obsługującej jeden z niskopoziomowych komponentów systemu wymusza aktualizację wysoko poziomowych komponentów.*

*Prace nad wysokopoziomowymi komponentami muszą być opóźnione, ponieważ niskopoziomowy komponent nie jest jeszcze gotowy.*

- *Rozwiązanie:*

*Zaimplementuj interfejs, którego będą używać komponenty niskopoziomowe i wysokopoziomowe.*

# SOLID - Dependency inversion principle



- *Problem praktyczny:*

*Nasza klasa do renderowania faktur musi teraz obsługiwać nowy rządowy format XML*

# BIZNES BARDZO LUBI :)



- *KISS – Keep It Stupid Simple*
- *DRY – Don't Repeat Yourself*
- *YAGNI – You 'Aint Gonna Need It*



## *General Responsibility Assignment Software Patterns/Principles*

- *Dlaczego definiujemy wzorce?*
- *Dlaczego “Principles”? Nie zawsze da się dany wzorzec rozpisać jako strukturę klas*
- *„Filozoficzne” podejście do klas i obiektów*
- *OOD (object oriented design) często ma przełożenie na programowanie strukturalne i funkcyjne*
- *Obiekt może jednocześnie implementować wiele wzorców*
  
- *Zbiór dobrych praktyk – zasady nie są bezwzględne! Mają pomóc, a nie utrudnić pracę*
- *Stosując się do tych zasad można zaprojektować obiektowy program w sposób racjonalny i zrozumiały dla innych programistów*



- *Controller (Kontroler)*
- *Creator (Twórca)*
- *Indirection (Pośrednictwo)*
- *Information Expert (Ekspert)*
- *High Cohesion (Wysoka spójność)*
- *Low coupling (Mało powiązań)*
- *Polymorphism (Polimorfizm)*
- *Protected Variation (Chroniona zmienność)*
- *Pure fabrication (Czyste wytwarzanie)*





*Pytanie: Który obiekt powinien obsłużyć daną zewnętrzną operację?*

- *Przyjmuje zdarzenia (eventy)*
- *Pierwszy obiekt przyjmujący interakcje*
- *Powinien delegować wykonywanie zadania do elementów nie będących częścią UI*



*Pytanie: która klasa powinna być odpowiedzialna za tworzenie obiektów klasy X?*

- *Gdy Creator, zawiera obiekt(y) komponentu docelowego*
  - *Gdy Creator zapisuje stan komponentu docelowego*
  - *Gdy Creator inicjalizuje komponent docelowy*
  - *Gdy Creator jest “blisko powiązany” z komponentem docelowym*
- 
- *Każdy program obiektowy obiekty tworzy, istotne jest odpowiednie zorganizowanie procesu tworzenia obiektów, tak by ilość powiązań między klasami była znikoma.*



*Pytanie: jak korzystać z klas bez uprzedniego niepotrzebnego przystosowywania się do nich?*

- *Komponent służy jako pośrednik w komunikacji między dwoma innymi komponentami*
- *Niekompatybilne interfejsy?*
- *Weryfikacja komunikatów*
- *Propagacja komunikatu w wielu obiektach*
  
- *Obniżenie zależności między klasami może zostać osiągnięte dzięki zastosowaniu pośredniej klasy, która będzie służyć jako komunikator między dwoma klasami*



*Pytanie: której klasie przydzielić odpowiedzialność za wykonywanie pewnego zadania?*

- *Znajdź komponent, który jest odpowiedzialny za zadanie*
- *Znajdź wszystkie informacje potrzebne do wykonania zadania*
- *Przekaż informacje potrzebne do wykonania zadania do komponentu odpowiedzialnego za jego wykonanie*
- *Wykonywać pewne zadanie powinna klasa posiadająca najwięcej informacji potrzebnych do jego wykonania*



*Pytanie: jak projektować klasy, aby spełniając jedno zadanie, miały przejrzystą implementację i mogły być ponownie użyte w innym projekcie?*

- *Komponenty powinny być “zwarte”*
- *Ich funkcjonalność powinna być jasno określona*
- *Powinny implementować tylko funkcjonalność związaną z ich bezpośrednim działaniem*
- *Powinny być łatwe w zarządzaniu*
- *Powinny być łatwe w zrozumieniu*
- *Prosty test na high cohesive:*
  - *Czy możesz w jednym zdaniu określić, jaki jest cel komponentu?*
  - *Czy kod jest zrozumiały, jeśli usuniesz z niego wszystkie komentarze?*

# GRASP – Low/Loose coupling



*Pytanie: jak zaprojektować komponenty, żeby nie zależały one znacząco od siebie?*

- *Komponenty powinny w możliwie małym stopniu zależeć od siebie nawzajem*
- *Prosty test na low coupling:*
  - *ile właściwości z jak bardzo złożonych klas jest potrzebnych?*
  - *jak bardzo złożone są argumenty metod/funkcji?*
- *Im więcej zależności, tym ciężiej użyć klasy ponownie, ciężiej jest zrozumieć klasę w izolacji, jedna zmiana może pociągnąć za sobą zmiany w wielu innych klasach*



*Pytanie: jak realizować zadanie, które może być wykonywane na kilka sposobów?*

- *Wspólne interfejsy działań niezależne od klas*
- *Parametryczne - gdy nie znamy typów argumentów, ale mamy wspólny wzorzec*
- *Podtyp - używamy dziedziczenia / ducktyping do wywoływania metod*
- *Row polymorphism - używamy dziedziczenia / ducktyping do dostępu do właściwości*



*Pytanie: jak projektować, by uniezależniać klasy od siebie – by jedna zmiana nie wymusiła zmian w kilku miejscach?*

- *Zmiany w komponencie A nie powinny mieć wpływu na komponent B*
- *Zmiany w komponencie B powinny być realizowane za pomocą interfejsów*

*(Zobacz też **O** w SOLID)*

- *Powinno się korzystać z abstrakcji! Przypisywać odpowiedzialności raczej do interfejsów, niż konkretnych klas.*





*Pytanie: komu przydzielić odpowiedzialność, jeżeli nie pasuje ona do żadnej dostępnej klasy?*

- *Komponent, który nie reprezentuje logicznie części problemu*
- *Jest elementem pomocniczym dla innych klas*
- *Szczególnie pomocne w przypadku high coupling/ low cohesion*
- *Sztuczne wytworzenie nowej klasy, ona powinna się zająć rozwiązaniem problemu*