# Testowanie i Jakość Oprogramowania

**Akademia Tarnowska**

**Katedra Informatyki**

Tomasz Gądek

# Wykład 6: Czysty kod, OOP, dobre praktyki

# Konwencje nazewnicze (zmienne)

Stosuj styl **snake_case** dla zmiennych.

```
user_name = "Jan Kowalski"
account_balance = 1000.50
```

# Konwencje nazewnicze (klasy)

Stosuj styl **PascalCase** dla klas.

```
class UserProfile:
    pass
```

# Konwencje nazewnicze (prywatne pola)

Prywatne pola w klasach oznaczamy prefiksem `_` lub `__`.

```
class User:
    def __init__(self, name):
        self._hidden_value = 42
        self.__private_value = "sekret"
```

# Konwencje nazewnicze (prywatne i publiczne metody)

Metody prywatne poprzedzamy `_`, a publiczne nie wymagają prefiksu.

```
class User:
    def public_method(self):
        return "Dostępna metoda"

    def _private_method(self):
        return "Metoda prywatna"
```

# Konwencje nazewnicze (stałe)

Stałe zapisujemy dużymi literami. Słowa wchodzące w skład stałej rozdzialamy znakiem `_`.

```
PI = 3.14159
MAX_CONNECTIONS = 100
```

# Konwencje nazewnicze (moduły i pakiety)

Nazwy modułów i pakietów tworzymy stosując styl **snake_case**.

Struktura katalogów:

```
my_package/
    data_processing.py
    utilities.py
```

Przykład importu:

```
import my_package.data_processing
```

# PEP (Python Enhancement Proposal)

PEP 8 to dokument zawierający zbiór wytycznych dotyczących stylu pisania kodu w Pythonie, mający na celu poprawę czytelności i spójności kodu.

# Paradygmaty OOP

Programowanie obiektowe (OOP) opiera się na kilku kluczowych paradygmatach, które definiują sposób projektowania i organizacji kodu:

- Hermetyzacja
- Abstrakcja
- Dziedziczenie
- Polimorfizm

# Hermetyzacja

Polega na ukrywaniu szczegółów implementacji i udostępnianiu tylko niezbędnego API

Dlaczego jest ważna?

- Chroni dane przed bezpośrednią modyfikacją.
- Poprawia modularność i bezpieczeństwo kodu.

# Hermetyzacja (przykład)

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def balance(self):
        return self.__balance

account = BankAccount(1000)
print(account.balance()) # 1000
```

# Abstrakcja

Polega na ukrywaniu zbędnych szczegółów i udostępnianiu tylko istotnych funkcji.

Dlaczego jest ważna?

- Upraszcza korzystanie z klas.
- Pomaga skupić się na funkcjonalności, a nie na implementacji.

# Abstrakcja (przykład)

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

class Car(Vehicle):
    def start_engine(self):
        print("Engine start (Car).")

car = Car()
car.start_engine()
```

```
Engine start (Car).
```

# Dziedziczenie

Pozwala na tworzenie nowych klas na podstawie istniejących.

Dlaczego jest ważne?

- Zapewnia ponowne użycie kodu.
- Ułatwia rozszerzanie funkcjonalności.

# Dziedziczenie (przykład)

```
class Animal:
    def speak(self):
        return "???"

class Dog(Animal):
    def speak(self):
        return "Woof woof!"

dog = Dog()
print(dog.speak())
```

```
Woof woof!
```

# Polimorfizm

Polimorfizm oznacza wielopostaciowość. Pozwala on zastosować jedną jednostkę (metodę, operator lub obiekt) do reprezentowania różnych typów w różnych scenariuszach.

Dlaczego jest ważny?

- Umożliwia elastyczność kodu.
- Redukuje konieczność pisania wielu if-else.

# Polimorfizm (problem)

Dlaczego funkcja `send_notification()` jest problematyczna?

```
def send_notification(notification_type, message):
    if notification_type == "email":
        print(f"e-mail notification: {message}")
    elif notification_type == "sms":
        print(f"sms notification: {message}")
    elif notification_type == "push":
        print(f"push notification: {message}")
    else:
        raise ValueError("unknown")

send_notification("sms", "hello Mike!")
```

# Polimorfizm (rozwiązanie)

```
class Notification:
    def send(self, message):
        raise NotImplementedError("Not implemented method!")

class EmailNotification(Notification):
    def send(self, message):
        print(f"e-mail notification:: {message}")

class SMSNotification(Notification):
    def send(self, message):
        print(f"sms notification:: {message}")

class PushNotification(Notification):
    def send(self, message):
        print(f"push notification:: {message}")

# ...
```

# Polimorfizm (rozwiązanie)

```
# ...

def send_notification(notification: Notification, message: str):
    notification.send(message)

send_notification(SMSNotification(), "hello Mike!")
```

# Asocjacja

**Luźne powiązanie** między obiektami. Jeden obiekt używa drugiego, ale oba mogą istnieć niezależnie.

> Relacja koleżeńska: możesz znać wielu ludzi, ale nie jesteście od siebie zależni.

# Asocjacja (przykład)

```
class Teacher:
    def __init__(self, name):
        self.name = name

    def teach(self):
        print(f"{self.name} prowadzi zajęcia.")

class Student:
    def __init__(self, name):
        self.name = name

    def attend_class(self, teacher):
        print(f"{self.name} uczestniczy w zajęciach prowadzonych przez {teacher.name}.")

smith = Teacher("Dr Smith")
alice = Student("Alice")

alice.attend_class(smith)
```

```
Alice uczestniczy w zajęciach prowadzonych przez Dr Smith.
```

# Agregacja

**"Całość" i "część" mogą istnieć niezależnie**. Jeden obiekt przechowuje referencję do drugiego, ale nie zarządza jego cyklem życia.

> Związek partnerski: jesteście razem, ale możecie funkcjonować osobno.

# Agregacja (przykład)

```
class Engine:
    def __init__(self, power):
        self.power = power

    def start(self):
        print(f"Silnik o mocy {self.power}KM został uruchomiony.")

class Car:
    def __init__(self, model, engine):
        self.model = model
        self.engine = engine

    def start(self):
        print(f"Samochód {self.model} uruchamia się...")
        self.engine.start()
# ...
```

# Agregacja (przykład)

```
# ...
engine = Engine(150)
toyota = Car("Toyota", engine)

toyota.start()
```

```
Samochód Toyota uruchamia się...
Silnik o mocy 150KM został uruchomiony.
```

# Kompozycja

**"Całość" i "część"** są silnie związane. Obiekt "część" nie istnieje bez obiektu "całość".

> Małżeństwo: jesteście tak związani, że trudno mówić o jednym bez drugiego.

# Kompozycja (przykład)

```
class CPU:
    def __init__(self, cores):
        self.cores = cores

    def process(self):
        print(f"Procesor działa ({self.cores} rdzeni).")

class Computer:
    def __init__(self, brand, cores):
        self.brand = brand
        self.cpu = CPU(cores)

    def run(self):
        print(f"Komputer {self.brand} uruchamia się...")
        self.cpu.process()

dell = Computer("Dell", 8)
dell.run()
```

# Kompozycja (przykład)

```
Komputer Dell uruchamia się...
Procesor działa (8 rdzeni).
```

# Koniec