Testowanie i Jakość Oprogramowania

Akademia Tarnowska

Katedra Informatyki

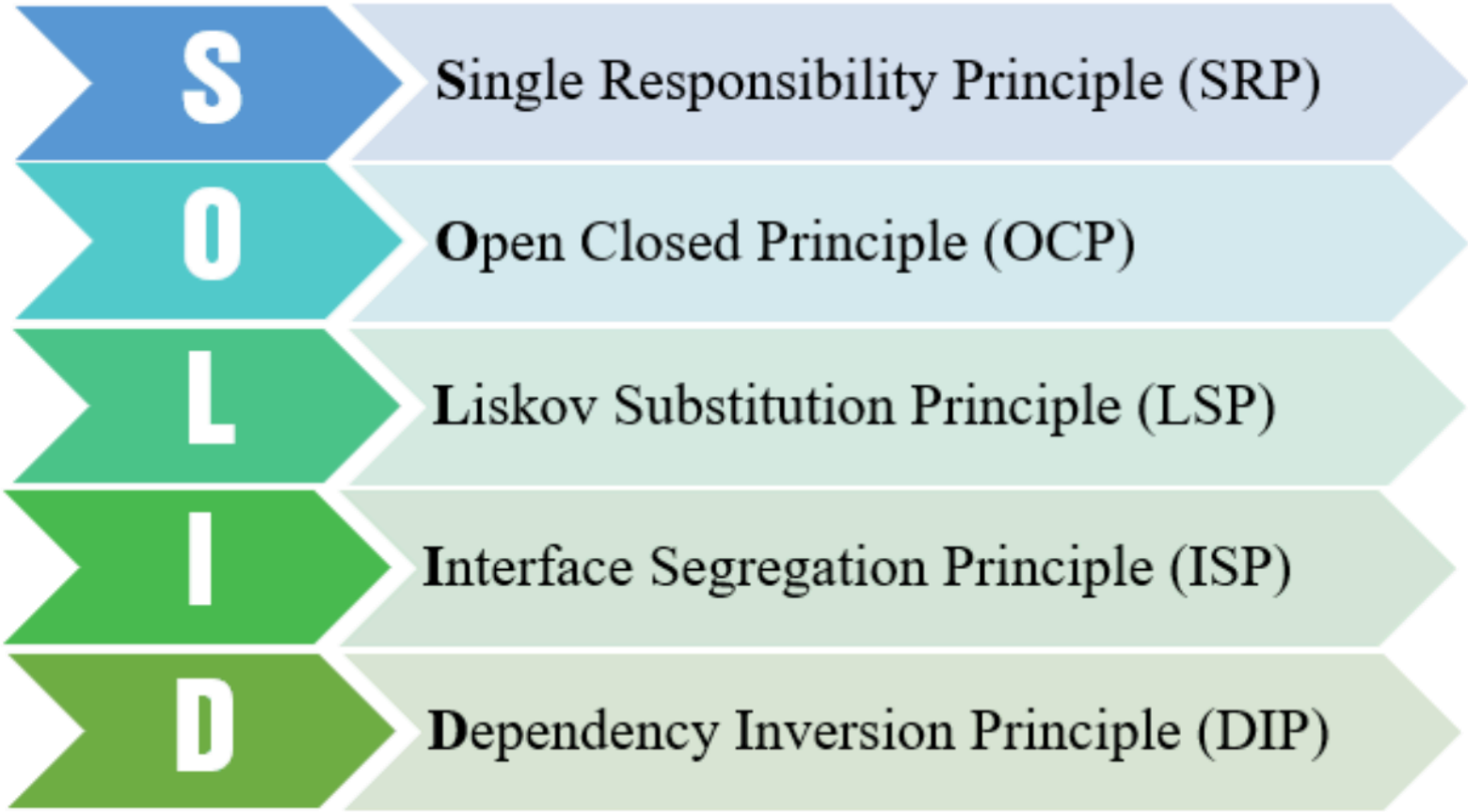
Tomasz Gądek

Wykład 7: SOLID, DRY, KISS, PRAWO DEMETER, CODE SMELL

SOLID

SOLID to akronim zaproponowanu przez Roberta C. Martina (Uncle Bob) przedstawiający zestaw wytycznych. Wytyczne te stosuje się podczas pisania programów w sposób obiektowy. Samo słowo SOLID można przetłumaczyć jako solidny, konkretny, mocny. Ta gra słów pewnie też miała spore znaczenie dla popularności samego akronimu.

SOLID


S
Single Responsibility Principle (SRP)
O
Open Closed Principle (OCP)
L
Liskov Substitution Principle (LSP)
I
Interface Segregation Principle (ISP)
D
Dependency Inversion Principle (DIP)

SRP

Zasada jednej odpowiedzialności. Klasa powinna mieć tylko jedną odpowiedzialność (nigdy nie powinien istnieć więcej niż jeden powód do modyfikacji klasy).

S jak **S**amodzielny

SRP (naruszenie zasady)

Jak to naprawić?

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

    def save_to_database(self):
        print(f"Saving user {self.name} to the database.")

    def send_welcome_email(self):
        print(f"Sending welcome email to {self.email}.")

user = User("John Doe", "john.doe@example.com")
user.save_to_database()
user.send_welcome_email()
```

OCP

Zasada otwarte zamknięte. Klasa powinna być otwarta na rozszerzenia oraz zamknięta na modyfikacje.

O jak **O**twarty

OCP (naruszenie zasady)

Jak to naprawić?

```
class Calculator:
    def calculate(self, operation, a, b):
        if operation == "addition":
            return a + b
        elif operation == "subtraction":
            return a - b
        elif operation == "multiplication":
            return a * b
        elif operation == "division":
            return a / b
        else:
            raise ValueError("Unknown operation")

calculator = Calculator()
print(calculator.calculate("addition", 5, 3))  # Output: 8
print(calculator.calculate("subtraction", 5, 3)) # Output: 2
```

LSP

Zasada podstawienia Liskov. Funkcje, które używają wskaźników lub referencji do klas bazowych muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych bez dokładnej znajomości tych obiektów.

L jak **L**iskov Barbara

LSP (wyjaśnienie)

- Klasy dziedziczące powinny rozszerzać, a nie zmieniać zachowanie klas bazowych.
- Program powinien działać poprawnie, niezależnie od tego, czy używa obiektu klasy bazowej, czy obiektu klasy dziedziczącej.

LSP (naruszenie zasady)

Dlaczego zasada zostanie naruszona po usunięciu wskazanej linii?

```
class CoffeeMachine:
    def make_coffee(self):
        print("przygotowuję kawę...")

class SweetCoffeeMachine(CoffeeMachine):
    def make_coffee(self):
        super().make_coffee()  # Usunięcie tej linii złamie zasadę Liskov
        self.add_sugar()

    def add_sugar(self):
        print("dodaję cukier...")
```

ISP

Zasada segregacji interfejsów. Wiele dedykowanych interfejsów jest lepsze niż jeden ogólny.

I jak **I**nterfejsy

ISP (naruszenie zasady)

Jak to naprawić?

```
class MultifunctionDevice:
    def print_document(self, document):
        raise NotImplementedError

    def scan_document(self, document):
        raise NotImplementedError

class Printer(MultifunctionDevice):
    def print_document(self, document):
        print(f"Printing document: {document}")

    def scan_document(self, document):
        raise Exception("Printer cannot scan!")
```

DIP

Zasada odwrócenia zależności. Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych. Zależności pomiędzy nimi powinny wynikać z abstrakcji.

D jak o**D**wrócenie zależności

DIP (naruszenie zasady)

Jak to naprawić?

```
class EmailService:
    def send_email(self, message):
        print(f"Wysyłam e-mail: {message}")

class NotificationService:
    def __init__(self):
        self.email_service = EmailService()

    def send_notification(self, message):
        self.email_service.send_email(message)

notification_service = NotificationService()
notification_service.send_notification("Witaj!")
```

DRY - Don't Repeat Yourself!

- **Nie powtarzaj się!** Wielokrotnie powielony kod nie tylko pogarsza czytelność naszego programu, lecz także skutecznie utrudnia jego późniejsze modyfikacje i tym samym utrudniamy życie sobie, oraz współpracownikom.
- **Jak stosować zasadę?** Jeśli wpadł Ci do głowy pomysł, żeby skopiować sekwencje if-ów, pętli etc. pomyśl nad stworzeniem abstrakcji, którą będziesz mógł (mogła) wielokrotnie użyć.

KISS - Keep It Simple Stupid!

Zrób to prosto, idto!**

- Kod który piszemy, ma być zrozumiały dla maszyny oraz prosty w czytaniu dla programisty.
- Program działa najlepiej gdy jego kod polega na prostocie, a nie złożoności.
- Staraj się, aby Twój kod wymagał jak najmniej myślenia podczas analizy.
- Jeśli za miesiąc wracasz do swojego kodu i nie wiesz co tam się dzieje, to pomyśl co czuje osoba która widzi ten kod pierwszy raz.

Zasada Skautów

Regułę **KISS** można rozszerzyć o zasadę **SKAUTÓW**:

> *Zawsze zostawiaj obóz czystszy niż go zastałeś.*

Tłumacząc na język programisty:

> *Usprawniaj kod w miejscu, w którym wprowadzasz zmiany.*

Law Of Demeter - Talk Only With Friends!

Prawo Demeter

- Prawo Demeter nakazuje odwoływania się tylko do **bliskich przyjaciół** czyli obiektów, które są bardzo dobrze znane danemu obiektowi.
- Trzymając się prawa Demeter oddzielamy od siebie dalekie części kodu, które nigdy nie powinny się ze sobą komunikować.

Law Of Demeter

Metoda danej klasy może odwoływać się jedynie do...

- metod tej samej klasy.
- pól (i ich metod) tej samej klasy.
- parametrów (i ich metod) przekazanych do niej.
- obiektów (i ich metod), które sama stworzy.

Law Of Demeter (przykład)

```
class Addition:
    def get_price(self):
        return 0.0

class Pizza:
    def get_price(self):
        return 0.0

class Pizzeria:
    def __init(self):
        self.addition = None

    def prepare_pizza(self, pizza):      # Możemy się odwoływać do...
        self.say_hello_to_customer()     # 1. metod tej samej klasy
        self.addition = Addition()       # 2. pól (i ich metod) tej samej klasy
        self.addition.get_price()        # 2. pól (i ich metod) tej samej klasy
        pizza.get_price()                # 3. parametrów (i ich metod), przekazanych do niej
        other_addition = Addition()      # 4. obiektów (i ich metod), które sama stworzy
        other_addition.get_price()       # 4. obiektów (i ich metod), które sama stworzy

    def say_hello_to_customer(self):
        print("Welcome to our restaurant!")
```

Law Of Demeter (naruszenie prawa)

```
class Addition:
    def get_price(self):
        return 0.0

class Pizza:
    def __init__(self):
        self.addition = Addition()

    def get_addition(self):
        return self.addition

class Pizzeria:
    def prepare_pizza(self, pizza):
        # Naruszenie prawa demeter (prawo jednej ".")
        pizza.get_addition().get_price()
```

Code Smell

Nazwa **code smell** odnosi się do pewnych cech programu, mówiących o złym sposobie implementacji oraz będące sygnałem do wykonania refaktoryzacji.

Przykłady **code smell**:

- Bardzo długie metody.
- Duże klasy, posiadające zbyt wiele odpowiedzialności.
- Zazdrość o kod (metoda jednej klasy częściej korzysta z atrybutów i metod innej klasy, niż z własnych).

Code Smell

Przykłady **code smell**:

- Powielanie kodu (niestosowanie zasady DRY).
- Stosowanie skomplikowanych wzorców projektowych, gdzie użycie prostszych byłoby wystarczające oraz zgodne z regułą KISS.
- Nazwy zmiennych i metod, które nic nam nie mówią.

Bad practices

Na czym polega problem?

```
cache = {}

def get_data(key):
    if key in cache:
        print("Data from cache.")
        return cache[key]
    else:
        print("Fetching data from external source.")
        data = "sample data..."
        cache[key] = data
        return data

print(get_data("key123"))
print(get_data("key123"))
```

Bad practices

Na czym polega problem?

```
# Returns the total price by multiplying price and quantity.
def calculate_total_price(price, quantity):
    return price * quantity
```

Bad practices

Na czym polega problem?

```
def process_customer_order(customer_id, product_id, quantity, shipping_address, billing_address, payment_method, discount_code, order_date):
    pass
```

Bad practices

Na czym polega problem?

```
def process_order(product_id: int, quantity: int, is_priority: bool) -> None:
    pass
```

Koniec