

**Szkoła Gospodarstwa Wiejskiego
Wydział Zastosowań Informatyki i Matematyki**

PARADYGMATY PROGRAMOWANIA

Jednostka 12 – Paradygmat aspektowy (AOP – Aspect-Oriented Programming)

1. Cel i zakres zajęć

Celem zajęć jest:

- poznanie **paradygmatu aspektowego** i jego roli w projektowaniu oprogramowania,
- zrozumienie pojęć: **aspekt, punkt zaczepienia (join point), przekroje (cross-cutting concerns)**,
- nauczenie się implementowania prostych aspektów np. logowania i pomiaru czasu w Pythonie,
- zrozumienie, jak AOP umożliwia **separację logiki głównej od przekrojowej**,
- przygotowanie studentów do rozpoznawania miejsc, gdzie AOP przynosi korzyści.

Zakres tematyczny:

- podstawowe pojęcia AOP,
 - problemy przekrojowe i separacja odpowiedzialności,
 - implementacja aspektów w Pythonie za pomocą dekoratorów i proxy,
 - przykłady zastosowań: logowanie, bezpieczeństwo, pomiar czasu, cache.
-

2. Wprowadzenie teoretyczne

2.1 Dlaczego AOP?

W dużych projektach często występują tzw. **zagadnienia przekrojowe** (cross-cutting concerns), które:

- pojawiają się w wielu miejscach kodu,
- nie należą bezpośrednio do logiki biznesowej,
- trudno je utrzymać i modyfikować, jeśli są „rozsiane” po kodzie.

Przykłady:

- logowanie,
- autoryzacja i bezpieczeństwo,
- obsługa wyjątków,
- pomiar czasu i profilowanie,
- cache.

 AOP umożliwia **wydzielenie takich zagadnień do osobnych modułów – aspektów**.

2.2 Podstawowe pojęcia AOP

- **Aspekt (Aspect)** – moduł implementujący funkcjonalność przekrojową.

- **Join Point** – konkretne miejsce w programie, w którym można „wstrzyknąć” kod aspektu (np. początek metody, wywołanie funkcji).
 - **Advice** – kod wykonywany w join point (np. „przed” lub „po” wywołaniu funkcji).
 - **Pointcut** – wzorzec określający, do których join points zastosować aspekt.
 - **Weaving** – proces „wplatania” aspektów do programu, zwykle automatyczny.
-

2.3 AOP vs OOP

- OOP — organizuje kod wokół obiektów i klas.
 - AOP — organizuje kod wokół aspektów i przekrojowych funkcjonalności.
 - AOP **nie zastępuje** OOP, lecz **uzupelnia** je.
-

3. Tutorial 1 – Prosty aspekt logowania (Python)

Najłatwiejszy sposób implementacji aspektów w Pythonie to **dekoratory**.

3.1 Dekorator logujący

```
def logowanie(funkcja):
    def wrapper(*args, **kwargs):
        print(f"[LOG] Wywołanie {funkcja.__name__} z args={args},
        kwargs={kwargs}")
        wynik = funkcja(*args, **kwargs)
        print(f"[LOG] Wynik {funkcja.__name__}: {wynik}")
        return wynik
    return wrapper

@logowanie
def dodaj(a, b):
    return a + b

@logowanie
def podziel(a, b):
    return a / b

print(dodaj(5, 3))
print(podziel(10, 2))
```

 Dekorator działa jako **aspekt** logowania — wstrzykuje kod przed i po wywołaniu każdej dekorowanej funkcji.

4. Tutorial 2 – Pomiar czasu i łączenie aspektów

4.1 Dekorator mierzący czas

```
import time

def mierz_czas(funkcja):
    def wrapper(*args, **kwargs):
        start = time.time()
        wynik = funkcja(*args, **kwargs)
        end = time.time()
        print(f"[TIMER] {funkcja.__name__} zajęła {end - start:.6f} s")
        return wynik
    return wrapper
```

4.2 Łączenie wielu aspektów

```
@logowanie
@mierz_czas
def suma_naturalnych(n):
    return sum(range(n))

suma_naturalnych(1_000_000)
```

👉 Aspekty można **komponować** – każdy dekorator wprowadza własną logikę przekrojową.

5. Tutorial 3 – Proxy obiektowe (zaawansowane)

Dekoratory działają na poziomie funkcji. Można też tworzyć **proxy dla całych obiektów**, które przechwytyują wszystkie wywołania metod:

```
class LoggingProxy:
    def __init__(self, obj):
        self._obj = obj

    def __getattr__(self, name):
        attr = getattr(self._obj, name)
        if callable(attr):
            def wrapper(*args, **kwargs):
                print(f"[LOG] Wywołanie metody {name} args={args},
kwargs={kwargs}")
                result = attr(*args, **kwargs)
                print(f"[LOG] Wynik {name}: {result}")
                return result
            return wrapper
        return attr
```

```
# Przykład
class Kalkulator:
    def dodaj(self, a, b): return a+b
    def pomnoz(self, a, b): return a*b

k = Kalkulator()
proxy = LoggingProxy(k)
print(proxy.dodaj(2, 3))
print(proxy.pomnoz(4, 5))
```

👉 Proxy pozwala stosować aspekt do **całych klas** bez modyfikacji ich kodu.

6. Zadania do samodzielnego wykonania

Zadanie 1 (obowiązkowe)

Napisz dekorator `autoryzacja`, który:

- sprawdza, czy w zmiennej `uzytkownik` jest odpowiednia rola,
 - tylko wtedy pozwala wykonać funkcję,
 - w przeciwnym razie wypisuje komunikat o braku uprawnień.
-

Zadanie 2 (obowiązkowe)

Zaimplementuj dwa aspekty:

- logowanie,
- mierz_czas.

Zastosuj je do kilku funkcji, które wykonują obliczenia lub przetwarzają dane.
Zbadaj, w jakiej kolejności są wykonywane.

Zadanie 3 (dodatkowe)

Napisz **proxy obiektowe**, które dodaje aspekt cache do dowolnego obiektu — zapamiętuje wyniki wywołań metod i zwraca je z pamięci przy powtórnych wywołaniach z tymi samymi argumentami.

7. Pytania kontrolne

1. Czym są aspekty i po co się je stosuje?
 2. Co to jest join point i advice?
 3. Jakie są różnice między AOP a OOP?
 4. Jak w Pythonie można zaimplementować AOP?
 5. Jakie są potencjalne zastosowania AOP w dużych projektach?
-

8. Checklisty

AOP

- Zaimplementowano co najmniej jeden aspekt jako dekorator.
 - Aspekty wstrzykują logikę przed/po wywołaniu funkcji.
 - Kod główny nie zawiera logiki przekrojowej.
 - Aspekty są wielokrotnego użytku i niezależne.
 - (Dla chętnych) Proxy obiektowe działa poprawnie.
-

9. Wzór sprawozdania

Dodatkowo:

- Diagram ilustrujący przepływ wykonania z aspekiem.
 - Fragmenty kodu funkcji i dekoratorów.
 - Krótkie porównanie AOP i wcześniejszych paradymatów.
-



Zakończenie

Po wykonaniu ćwiczenia student powinien:

- rozumieć koncepcję paradymatu aspektowego,
 - umieć identyfikować zagadnienia przekrojowe,
 - potrafić implementować proste aspekty w Pythonie,
 - stosować dekoratory i proxy do separacji logiki przekrojowej,
 - znać podstawowe pojęcia AOP (aspekt, join point, weaving).
-

Chcesz, żebym przygotował w tym samym stylu **Jednostkę 13 – Podsumowanie i projekt końcowy (integracja paradymatów)?**

Powiedzialeś(-aś):

tak

ChatGPT powiedział:

Świetnie 🌟

Poniżej masz pełny, kompleksowy materiał dydaktyczny dla

 **Jednostki 13: Podsumowanie i projekt końcowy — integracja paradymatów programowania**

Zachowuje ten sam rozbudowany układ jak poprzednie jednostki (strona tytułowa, teoria, projekt, wskazówki krok po kroku, checklisty, pytania kontrolne i wzór sprawozdania końcowego).



STRONA TYTUŁOWA

<div style="text-align: center; line-height: 1.6;">

**Szkoła Gospodarstwa Wiejskiego
Wydział Zastosowań Informatyki i Matematyki**

PARADYGMATY PROGRAMOWANIA

Jednostka 13 – Podsumowanie i projekt końcowy: integracja paradymatów

Kierunek: Informatyka, semestr 5
Prowadzący: dr inż. Krzysztof Malczewski

Data zajęć:

Grupa:

Prowadzący (podpis):

</div>

1. Cel i zakres zajęć

Celem zajęć jest:

- podsumowanie wszystkich poznanych paradymatów programowania,

- pokazanie, że w praktycznych projektach programistycznych stosuje się kombinacje różnych paradygmatów,
- zaprojektowanie i zaimplementowanie niewielkiego projektu programistycznego wykorzystującego co najmniej 4 różne paradygmaty,
- utrwalenie umiejętności dokumentowania i prezentowania rozwiązań.

Zakres tematyczny:

- powtórka: imperatywny, modularny, obiektowy, funkcyjny, deklaratywny, logiczny, współbieżny, zdarzeniowy, aspektowy,
 - integracja różnych stylów programowania,
 - praca projektowa w małych zespołach lub indywidualnie,
 - sprawozdanie końcowe i prezentacja rozwiązań.
-

2. Przypomnienie paradygmatów

Paradygmat	Główna idea
Imperatywny	Sekwencja instrukcji zmieniających stan
Modularny	Podział programu na moduły i interfejsy
Obiektowy	Modelowanie danych i zachowań przez obiekty
Funkcyjny	Deklaratywne operacje na danych, czyste funkcje, map/filter/reduce
Deklaratywny / logiczny	Opis relacji i zapytań (SQL, Prolog)
Współbieżny	Równoczesne wykonywanie wielu zadań (wątki/procesy)
Zdarzeniowy / reaktywny	Program reaguje na zdarzenia i dane w czasie rzeczywistym
Aspektowy	Oddzielenie logiki przekrojowej (logowanie, pomiar czasu) od logiki głównej

3. Projekt końcowy – Założenia

Studenci tworzą niewielką aplikację lub system, który:

- Rozwiązuje konkretny, dobrze zdefiniowany problem,
- Integruje minimum 4 różne paradygmaty programowania,
- Jest napisany w jednym języku (np. Python, C, Java) lub mieszaninie (np. Python + SQL + Prolog),
- Posiada czytelny podział logiczny na moduły i warstwy,
- Jest dobrze udokumentowany (kod + sprawozdanie końcowe).

Przykładowe tematy:

- **System monitorowania czujników** – moduły funkcyjne + współprzeźność + zdarzenia + logika reaktywna.
 - **Analizator danych** – modularny + funkcyjny + deklaratywny (SQL) + aspekty (logowanie).
 - **System ekspercki** – logiczny (Prolog) + deklaratywny (fakty) + imperatywny (interfejs) + zdarzeniowy.
 - **Gra tekstowa** – obiektowy (postacie), funkcyjny (przetwarzanie akcji), zdarzeniowy (reakcje), współprzeźny (tło / AI).
 - **System obsługi zadań** – modularny backend, funkcyjna analiza, deklaratywne zapytania SQL, AOP do logowania.
-

4. Etapy realizacji projektu

4.1 Etap 1 – Definicja problemu

- Zidentyfikuj cel projektu, użytkowników i kontekst zastosowania.
 - Określ **jakie paradygmaty zostaną użyte i w jakich częściach systemu**.
 - Przygotuj diagram wysokiego poziomu (np. komponentów lub przepływu danych).
-

4.2 Etap 2 – Projekt systemu

- Podziel system na **moduły / komponenty**, przypisz im role i odpowiednie paradygmaty.
 - Zdefiniuj interfejsy między modułami (np. API, funkcje, zdarzenia, bazy danych).
 - Zaplanuj obsługę zdarzeń, współprzeźność i aspekty przekrojowe.
-

4.3 Etap 3 – Implementacja

- Zaimplementuj moduły zgodnie z projektem, wykorzystując wcześniej poznane wzorce.
 - Testuj częściowo (moduły) i całościowo (system).
 - Dokumentuj zastosowane paradygmaty w kodzie (np. komentarze, nagłówki plików).
-

4.4 Etap 4 – Prezentacja i dokumentacja

- Przygotuj krótką prezentację (np. 5–7 minut) projektu.
- Oddaj kod źródłowy + sprawozdanie końcowe z opisem zastosowanych paradygmatów i ich integracji.

- Pokaż, jak różne style współpracują w praktyce.
-

5. Wymagania minimalne projektu

- Co najmniej **4 różne paradygmaty** użyte sensownie i spójnie,
 - Działająca implementacja i krótka prezentacja działania,
 - Dokumentacja kodu i opis koncepcyjny,
 - Sprawozdanie końcowe,
 - Kod zorganizowany w sposób modularny i czytelny.
-

6. Kryteria oceny projektu

Kryterium	Maks. pkt
Poprawność działania	20
Integracja paradygmatów	20
Jakość architektury i kodu	15
Dokumentacja i sprawozdanie	15
Innowacyjność / pomysłowość	10
Prezentacja	10
Styl, czytelność, współpraca	10
Razem	100

7. Pytania kontrolne

1. Dlaczego w praktyce często łączy się różne paradygmaty programowania?
 2. Jakie paradygmaty najlepiej nadają się do warstw danych, a jakie do logiki biznesowej czy interfejsu?
 3. W jakich miejscach systemu można zastosować aspekty?
 4. Jak można połączyć paradygmat zdarzeniowy z logicznym?
 5. Jak modularność wpływa na możliwość integracji paradygmatów?
-

8. Checklisty projektowe

Architektura

- System ma dobrze zdefiniowane komponenty.

- Każdy komponent używa świadomie wybranego paradygmatu.
- Komunikacja między komponentami jest jasna.

Implementacja

- Kod działa poprawnie i bez błędów krytycznych.
- Zastosowano min. 4 paradygmaty w różnych warstwach.
- Projekt jest spójny i logiczny.

Dokumentacja

- Sprawozdanie opisuje każdy paradygmat i jego rolę.
 - Kod jest opatrzony komentarzami.
 - Dołączono przykłady działania i diagramy.
-

9. Wzór sprawozdania końcowego

Tytuł projektu:

Autorzy:

Cel projektu:

Opis funkcjonalności:

Zastosowane paradygmaty:

- Paradygmat 1: ...
- Paradygmat 2: ...
- Paradygmat 3: ...
- Paradygmat 4: ...

Opis architektury systemu:

- Moduły, interfejsy, diagramy.

Przykłady użycia / wyniki:

- Zrzuty ekranu, przykładowe logi, działania.

Wnioski i refleksje:

- Co się sprawdziło?
 - Jakie były trudności?
 - Co można byłoby rozwinąć?
-



Zakończenie

Po zakończeniu tej jednostki student powinien:

- umieć rozpoznać i świadomie stosować różne paradygmaty w jednym projekcie,
- potrafić dobrać paradygmat do rodzaju problemu,
- rozumieć, jak łączyć różne style w architekturze oprogramowania,
- potrafić udokumentować i zaprezentować złożone rozwiązanie,
- mieć doświadczenie projektowe zbliżone do realnych zespołowych praktyk.