Szkoła Gospodarstwa Wiejskiego
Wydział Zastosowań Informatyki i Matematyki

PARADYGMATY PROGRAMOWANIA

Jednostka 3 – Paradygmat modularny

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

Data zajęć: .....
Grupa: .....
Prowadzący (podpis): .....

---

1. Cel i zakres zajęć

Celem ćwiczeń jest:

- zrozumienie **paradygmatu modularnego** i roli modułów w organizacji kodu,
- umiejętność **dzielenia programu na moduły** i definiowania interfejsów między nimi,
- wykorzystanie nagłówków, bibliotek i plików źródłowych w C oraz modułów w Pythonie,
- przygotowanie modularnej wersji programu z poprzednich jednostek.

Zakres tematyczny:

- definicja i cechy paradygmatu modularnego,
- podział programu na logiczne komponenty (moduły),
- interfejsy i enkapsulacja,
- kompilacja wieloplika (C) i import modułów (Python),
- zalety modularności dla zespołowej pracy i konserwacji kodu.

---

2. Wprowadzenie teoretyczne

2.1 Definicja

Paradygmat modularny polega na dzieleniu programu na **niezależne, logiczne moduły**, które:

- realizują określone zadania,
- mają jasno zdefiniowane **interfejsy publiczne**,
- ukrywają szczegóły implementacyjne (enkapsulacja),
- mogą być niezależnie kompilowane, testowane i ponownie używane.

2.2 Cechy modularności

- **Separacja odpowiedzialności** — każdy moduł odpowiada za ściśle określoną część funkcjonalności.
- **Interfejs / implementacja** — np. nagłówek `.h` i plik `.c` w C lub publiczne API w Pythonie.
- **Niezależność** — zmiana jednego modułu nie powinna wpływać na resztę programu (jeśli interfejs nie ulega zmianie).
- **Ponowne użycie** — moduły mogą być stosowane w wielu projektach.
- **Równoległa praca** — różni członkowie zespołu mogą rozwijać różne moduły.

---

3. Tutorial 1 – Modularny projekt w C (krok po kroku)

Załóżmy, że mamy program do wczytywania danych, analizowania ich (suma, średnia, min, max, sortowanie), podobny do jednostki 2. Teraz podzielimy go na moduły.

3.1 Struktura projektu

```
lab03/
├── main.c
├── io.c
├── io.h
├── stats.c
├── stats.h
├── sort.c
├── sort.h
└── Makefile
```

Podział:

- `io.[ch]` – moduł wejścia/wyjścia: wczytywanie i wypisywanie danych.
- `stats.[ch]` – moduł obliczeń statystycznych (suma, średnia, min, max).
- `sort.[ch]` – moduł sortowania.
- `main.c` – funkcja główna, korzystająca z interfejsów pozostałych modułów.

3.2 Przykład: `io.h`

```
#ifndef IO_H
#define IO_H

int wczytaj_dane(double dane[], int max_n);
void wypisz_dane(const double dane[], int n);

#endif
```

io.c

```
#include <stdio.h>
#include "io.h"

int wczytaj_dane(double dane[], int max_n) {
    int n;
    printf("Podaj liczbę elementów: ");
    if (scanf("%d", &n) != 1 || n <= 0 || n > max_n) {
        printf("Błędna liczba elementów.\n");
        return 0;
    }
    for (int i = 0; i < n; i++) {
        printf("Podaj liczbę %d: ", i+1);
        scanf("%lf", &dane[i]);
    }
    return n;
}

void wypisz_dane(const double dane[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%.2f ", dane[i]);
    }
    printf("\n");
}
```

3.3 Przykład: `stats.h`

```
#ifndef STATS_H
#define STATS_H

double oblicz_sume(const double dane[], int n);
double oblicz_srednia(const double dane[], int n);
void znajdz_min_max(const double dane[], int n, double *min, double *max);

#endif
```

stats.c

```
#include "stats.h"

double oblicz_sume(const double dane[], int n) {
    double suma = 0;
    for (int i = 0; i < n; i++) suma += dane[i];
    return suma;
}

double oblicz_srednia(const double dane[], int n) {
    return (n == 0) ? 0.0 : oblicz_sume(dane, n)/n;
}
```

```
void znajdz_min_max(const double dane[], int n, double *min, double *max) {
    if (n == 0) return;
    *min = *max = dane[0];
    for (int i = 1; i < n; i++) {
        if (dane[i] < *min) *min = dane[i];
        if (dane[i] > *max) *max = dane[i];
    }
}
```

3.4 Przykład: `sort.h` i `sort.c`

```
#ifndef SORT_H
#define SORT_H

void bubble_sort(double dane[], int n);

#endif
#include "sort.h"

void bubble_sort(double dane[], int n) {
    int swapped;
    do {
        swapped = 0;
        for (int i = 0; i < n-1; i++) {
            if (dane[i] > dane[i+1]) {
                double tmp = dane[i];
                dane[i] = dane[i+1];
                dane[i+1] = tmp;
                swapped = 1;
            }
        }
    } while (swapped);
}
```

3.5 `main.c`

```
#include <stdio.h>
#include "io.h"
#include "stats.h"
#include "sort.h"

#define MAX_N 1000

int main(void) {
    double dane[MAX_N];
    int n = wczytaj_dane(dane, MAX_N);
    if (n == 0) return 0;

    double suma = oblicz_sume(dane, n);
    double srednia = oblicz_srednia(dane, n);
    double min, max;
    znajdz_min_max(dane, n, &min, &max);

    bubble_sort(dane, n);
```

```
    printf("Dane posortowane:\n");
    wypisz_dane(dane, n);
    printf("Suma: %.2f, Średnia: %.2f, Min: %.2f, Max: %.2f\n",
           suma, srednia, min, max);
    return 0;
}
```

3.6 Makefile (wieloplikowa kompilacja)

```
CC=gcc
CFLAGS=-std=c11 -Wall -Wextra -O2

OBJS=main.o io.o stats.o sort.o

all: lab03

lab03: $(OBJS)
	$(CC) $(CFLAGS) -o lab03 $(OBJS)

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f *.o lab03
```

---

4. Tutorial 2 – Modularność w Pythonie

Struktura:

```
lab03/
├── main.py
├── io_utils.py
├── stats.py
└── sorting.py
```

io_utils.py

```
def wczytaj_dane():
    n = int(input("Podaj liczbę elementów: "))
    return [float(input(f"Liczba {i+1}: ")) for i in range(n)]

def wypisz_dane(dane):
    print(" ".join(str(x) for x in dane))
```

stats.py

```
def suma(dane):
    return sum(dane)

def srednia(dane):
    return sum(dane)/len(dane) if dane else 0

def min_max(dane):
    return (min(dane), max(dane)) if dane else (None, None)
```

sorting.py

```
def bubble_sort(dane):
    dane = dane[:]
    while True:
        swapped = False
        for i in range(len(dane)-1):
            if dane[i] > dane[i+1]:
                dane[i], dane[i+1] = dane[i+1], dane[i]
                swapped = True
        if not swapped:
            break
    return dane
```

main.py

```
import io_utils
import stats
import sorting

def main():
    dane = io_utils.wczytaj_dane()
    s = stats.suma(dane)
    sr = stats.srednia(dane)
    min_val, max_val = stats.min_max(dane)
    posortowane = sorting.bubble_sort(dane)

    print("Dane posortowane:")
    io_utils.wypisz_dane(posortowane)
    print(f"Suma: {s}, Średnia: {sr}, Min: {min_val}, Max: {max_val}")

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

---

5. Zadania do samodzielnego wykonania

Zadanie 1 (obowiązkowe)

Przepisz swój program z jednostki 2 do wersji **modularnej**, dzieląc na min. 3 moduły (I/O, statystyki, sortowanie). Przygotuj Makefile lub analogiczną strukturę w Pythonie.

Zadanie 2 (obowiązkowe)

Dodaj nowy moduł `analityka` z funkcją `mediana` i `odchylenie_standardowe`. Zmodyfikuj `main` tak, aby korzystał z tego modułu.

Zadanie 3 (dodatkowe)

Zaprojektuj moduł `raport` generujący podsumowanie w pliku tekstowym (`raport.txt`), zawierające wszystkie statystyki.

---

6. Pytania kontrolne

1. Na czym polega modularność?
2. Jakie są różnice między interfejsem a implementacją modułu?
3. Jak modularność wpływa na testowanie i konserwację kodu?
4. Dlaczego kompilacja wieloplika wymaga Makefile?
5. Jak w Pythonie odzwierciedla się modularność?

---

7. Checklisty

Projekt modularny

- Kod podzielony na minimum 3 moduły.
- Każdy moduł ma własny interfejs (nagłówek / publiczne funkcje).
- `main` korzysta wyłącznie z interfejsów.
- Projekt kompiluje się przez Makefile lub działa poprawnie z importami.

---

8. Wzór sprawozdania

Dodatkowo:

- Diagram zależności modułów.
- Opis, co robi każdy moduł.
- Komentarz o potencjale ponownego użycia i pracy zespołowej.

Zakończenie

Po wykonaniu ćwiczenia student powinien:

- rozumieć istotę modularności,
- potrafić podzielić program na moduły i zdefiniować interfejsy,
- znać proces kompilacji wieloplika w C i strukturę modułów w Pythonie,
- umieć rozbudować program o nowe moduły bez modyfikacji istniejących.