Paradygmaty Programowania

Programowanie strukturalne

dr inż. Marcin Bączyk (prowadzący)
dr inż. Marek Niewiński (współautor prezentacji)

Wykład 3

14 marca 2022

Treść dzisiejszego wykładu

- Przegląd sposobów programowania
  - strukturalne
  - obiektowe
  - funkcyjne
- Programowanie strukturalne
  - wprowadzenie
  - programowanie proceduralne
  - instrukcja `goto`
  - programowanie strukturalne
  - przykład

Przegląd sposobów tworzenia oprogramowania

Programowanie strukturalne

Skoki zastąpiono instrukcjami **if**/**then**/**else** oraz **for**/**do**/**while**/**until**.

"Programowanie strukturalne wymusza dyscyplinę bezpośredniego przekazywania sterowania".

Stosowane jest jako algorytmiczna podstawa tworzonych modułów.

Programowanie obiektowe

Stos wywołań funkcji można przenieść na stertę, a zmienne lokalne zadeklarowane w funkcji mogą istnieć po jej zakończeniu.

"Programowanie obiektowe wymusza dyscyplinę pośredniego przekazywania sterowania".

Polimorfizm wykorzystywany jako metoda przekraczania granic poszczególnych modułów.

Przegląd sposobów tworzenia oprogramowania

Programowanie funkcyjne

Niezmienność jako fundamentalna cecha rachunku lambda.

"Programowanie funkcyjne wymusza dyscyplinę podczas przypisywania wartości".

Programowanie funkcyjne wykorzystywane do zaprowadzenia dyscypliny z dostępie do danych.

Każdy ze sposobów programowania nakłada dodatkowe ograniczenia nie dając nic w zamian. Uwaga skupiona jest na tym czego nie robić. Rzadko udaje się uzyskać informację na temat co należy robić. - **doświadczenie**.

Decydując się na któryś ze sposobów programowania zmuszeni jesteśmy do **kompromisu**.

Programowanie strukturalne

Cechy charakterystyczne:

- struktury kontrolne
  - sekwencja
  - wybór
  - iteracja
- podprogramy
- bloki

Konsekwencją stosowania podejścia czysto strukturalnego jest pojedyncze wyjście.
W ramach wykładu jako pojedyncze wyjście oznaczać będzie sytuacja w której funkcja / procedura zawiera pojedyncza instrukcję zwracającą obliczony wcześniej wynik.

Programowanie strukturalne

Przy pomocy struktur kontrolnym oraz podprogramów możliwe jest stworzenie dowolnego oprogramowania.

Jednakże programowanie jest trudne, a duża złożoność systemów informatycznych skutkuje dużą podatnością na pojawianie się błędów.

W związku z tym, aby zmniejszyć lokalnie złożoność systemu program dzieli się na podprogramy.

Aby zapewnić "bezbłędność" oprogramowania dowodzi się poprawności każdego z fragmentów programu. Zmniejszenie ilości kodu każdego podprogramu mniejsza złożoność dowodu.

Programowanie strukturalne

Dekompozycja funkcyjna

- Istota programowania strukturalnego;
- Rekursywna dekompozycja modułów aż do osiągnięcia poziomu dla którego możliwe jest przeprowadzenie dowodu;
- Matematyczne dowodzenie poprawności kodu jest pracochłonne i trudne; obecnie praktycznie nie stosowane.

Testowanie

- Metoda naukowa;
- Sprawdzanie poprawności kodu odbywa się poprzez próbę stwierdzenia ich niepoprawności;
- W przypadku niemożliwości stwierdzenia niepoprawności kodu przyjmuje się, że jest on pozbawionych błędów.

Programowanie strukturalne

Tworzenie oprogramowania nie jest zagadnieniem matematycznym.

W programowaniu strukturalnym:

- Podstawą są instrukcje sterujące przepływem programu i bloki instrukcji często organizowane w podprogramy
- Problem (program) dekomponowany jest na dostatecznie małe podprogramy, dla których można przeprowadzić dowód poprawności lub jeżeli dowiedzenie jest niemożliwe, lub nie da się dowieść ich niepoprawności.
- Główny nacisk kładziony jest na to **w jaki sposób** wykonać dane zadanie.

Cechy charakterystyczne:

- abstrakcja
- hermetyzacja
- dziedziczenie
- polimorfizm

Konsekwencją stosowania podejścia czysto obiektowego jest przypisanie każdej operacji w systemie konkretnemu obiektowy i przekazanie mu odpowiedzialności za tę operację.

Programowanie obiektowe

Programowanie obiektowe jest odpowiedzią na potrzebę modelowania świata rzeczywistego w możliwie wierny sposób (z dokładnością do poziomu abstrakcji). Podstawowe cechy programowania obiektowego ułatwiają tworzenie oprogramowania dla konkretnej dziedziny.

Żadna z podstawowych cech programowania obiektowego nie musi być wspierana przez dany język aby móc napisać w nim program obiektowy.

Systemy projektowane obiektowo składają się z dużej ilości "małych" obiektów współpracujących ze sobą. Interfejs publiczny, oznaczający usługi jakie obiekt może świadczyć swoim klientów informuje jedynie o tym co może być zrobione. Klienci poszczególnych obiektów nie ingerują w to w jaki sposób dana usługa jest świadczona.

Systemy obiektowe projektowane są w taki sposób aby obiekt odpowiedzialny za daną funkcjonalność mógł być zastąpiony innym bez zmiany kodu klienta.

Programowanie obiektowe

W programowaniu obiektowym:

- Podstawą systemu są współpracujące ze sobą obiekty;
- Obiekty mogą być przekazywane pomiędzy modułami bez wprowadzania dodatkowych (zbędnych) zależności;
- Klienci danego obiektu nie wiedzą w jaki sposób realizowana jest jego odpowiedzialność;
- Moduły wysokiego poziomu nie zależą od modułów niskiego poziomu;
- Rozwój (dodawanie nowych) funkcjonalności odbywa się bez zmiany wysokopoziomowych reguł biznesowych;
- Główny nacisk kładziony jest na to **kto ma** wykonać dane zadanie.

Cechy charakterystyczne:

- brak przypisań
- brak pętli
- czyste funkcje

Konsekwencją stosowania podejścia czysto funkcyjnego jest brak jakiegokolwiek modyfikowalnego stanu wewnętrznego programu.

Programowanie funkcyjne

System projektowany funkcyjnie składa się z funkcji, które przyjmują argumenty i zwracają wyniki obliczeń. Funkcje wchodzące w skład systemu składają się z innych funkcji, aż do pojedynczych bloków instrukcji - mechanizm dekompozycji funkcyjnej.

Podobnie do programowania obiektowego istotniejsze jest to co robi dana funkcja niż to w jaki sposób to robi. Często obiekty (niezmienne!) są wykorzystywane jako obiekty funkcyjne - funkcje posiadające pewien niemodyfikowalny stan wewnętrzny.

Programowanie funkcyjne obecnie jest odpowiedzią na problemy pojawiające się w aplikacjach współbieżnych. Niemożliwość zmiany stanu, któregokolwiek z elementów działającego programu usuwa możliwość pojawienia się wyścigów i zakleszczeń.

W programowaniu funkcyjnym:

- Minimalizowana jest liczba elementów które mogą ulegać zmianie, a mechanizmy zmiany podlegają ścisłej kontroli
- Zakłada się, że czas niezbędny na ponowne wykonanie obliczeń jest pomijalnie mały i w związku z tym wartości zmiennych nie są zapamiętywane i modyfikowane
- Zakłada się, że ilość pamięci potrzebnej do przechowywania wszystkich niezbędnych informacji jest nieograniczona
- Główny nacisk kładziony jest na to **co ma być** zrobione.

Wnioski

Poszczególne sposoby programowania nakładają w programie różne ograniczenia:

- strukturalne – na bezpośrednie przekazywanie sterowania,
- obiektowe – na pośrednie przekazywania sterowania,
- funkcyjne – na przypisywanie wartości zmiennym,

przy czym nie oferują niczego dodatkowego.

Wszystkie przedstawione sposoby tworzenia oprogramowania znane były już w latach 60-tych XX wieku. Ze względu na rożne ograniczenia w danej epoce rożne sposoby programowania stawały się dominującymi.

Natomiast od momentu stworzenia pierwszych programów reguły programowania nie uległy zasadniczej zmianie. Programy składają się z sekwencji, selekcji, iteracji i pośrednictwa.

Wnioski

wniosek nadrzędny

Poza pewnymi skrajnymi przypadkami nie istnieje podejście czysto strukturalne, obiektowe czy funkcyjne do wytwarzania oprogramowania.

Duże systemy informatyczne składają się z wielu modułów, które mogą być napisane z wykorzystaniem rożnych sposobów programowania.

Przykład

Problem obliczeniowy

Znaleźć w zbiorze liczb naturalnych liczbę tych, które spełniają zadany warunek.
Obecnie zdefiniowane są dwa warunki. Zliczane są liczby parzyste oraz większe od 3.
W przyszłości mogą pojawić się nowe warunki.

Przykład - rozwiązanie strukturalne

```
from strukturalnie_funkcje import *

if __name__ == "__main__":
    natural_numbers = (n for n in range(1, 5))

    numbers = list(natural_numbers)

    print(numbers)
    print(calcNumbers(numbers, "even"))
    print(calcNumbers(numbers, "greaterThan3"))
```

Przykład - rozwiązanie strukturalne

```
def isEven(number):
    return not(number % 2)

def isGreaterThan3(number):
    return number > 3

def calcNumbers(numbers, isKindOf):
    sum = 0
    for number in numbers:
        if isKindOf == "even":
            sum = sum + isEven(number)
        elif isKindOf == "greaterThan3":
            sum = sum + isGreaterThan3(number)
        else:
            print("Incorrect option")

    return sum
```

Przykład - rozwiązanie obiektowe

```
from obiektowo_obiekty import *

if __name__ == "__main__":
    natural_numbers = (n for n in range(1, 5))

    numbers = list(natural_numbers)

    print(numbers)

    calcEven = NumberCalculator(isEven())
    calcGreateThan3 = NumberCalculator(isGreaterThan(3))

    print(calcEven(numbers))
    print(calcGreateThan3(numbers))
```

Przykład - rozwiązanie obiektowe

```
class isEven:
    def __call__(self, number):
        return not(number % 2)

class isGreaterThan:
    def __init__(self, value):
        self.value = value

    def __call__(self, number):
        return number > self.value

class NumberCalculator:
    def __init__(self, isKindOf):
        self.isKindOf = isKindOf

    def __call__(self,  numbers):
        sum = 0
        for number in numbers:
            if self.isKindOf(number):
                sum = sum + 1
        return sum
```

Przykład - rozwiązanie funkcyjne

```
def calcNumbers(numbers, isKindOf):
    if len(numbers) == 0:
        return 0
    else:
        return isKindOf(numbers[0]) + calcNumbers(numbers[1:], isKindOf)

if __name__ == "__main__":
    natural_numbers = (n for n in range(1, 5))

    numbers = list(natural_numbers)

    print(numbers)
    print(calcNumbers(numbers, lambda x: not(x % 2)))
    print(calcNumbers(numbers, lambda x: x > 3))
```

Język programowania

Przyjęty umowny sposób zapisu symboli, dzięki któremu można komunikować się z maszyną obliczeniową, zlecając jej wykonanie zadania.

W drugiej połowie lat 40. (ubiegłego wieku), kiedy John von Neumann zaproponował nowy sposób budowy maszyn obliczeniowych – charakteryzujący się między innymi tym, że maszyny te posiadają skończoną listę podstawowych rozkazów – operatorzy mogli wprowadzać sekwencje tych rozkazów, by rozwiązywać specyficzne problemy obliczeniowe i stali się pierwszymi **programistami**.

Programowanie – język maszynowy

Pierwsze programy pisane były w językach maszynowych:

- program składał się z sekwencji: **kodów rozkazów** i **adresów** ich argumentów

Programy w kodzie maszynowym

1. absolutnie nieprzenośne między różnymi typami maszyn
2. bardzo trudne w modyfikacji
3. nieczytelne, a więc i wyszukanie błędów często awykonalne

Przykład: x86-64

```
1000:   f3 0f 1e fa
1004:   48 83 ec 08
1008:   48 8b 05 d9 2f 00 00
100f:   48 85 c0
1012:   74 02
1014:   ff d0
```

Programowanie – asembler

Asembler

Program tłumaczący programy zapisane w postaci symbolicznej na kod maszynowy

Idea:

- Przypisać każdemu kodowi instrukcji **mnemonik** w sposób jednoznacznie kojarzący się z wykonywaną czynnością
- Nadać **identyfikator (nazwę symboliczną)** każdej z danych, by uniknąć operowania jawnymi adresami pamięci

Przykład: x86-64

```
push {r7}
add r7, sp, #0
movs r3, #0
mov r0, r3
mov sp, r7
ldr.w r7, [sp], #4
```

Programowanie – pierwsze języki wysokiego poziomu

W drugiej połowie lat 50 powstały języki kompilowalne:

- **FORTRAN** (*FORmula TRANslation*) stworzony przez zespół pracujący w **IBM**
- **COBOL** (*COmmon Business-Oriented Language*) stworzony przez komitet CODASYL (Conference on Data Systems Languages)
- **ALGOL** (*ALGOrithmic Language*) stworzony przez zespół pracujący w Swiss Federal Institute of Technology in Zurich (ETH Zurich)

Wszystkie te języki udostępniały możliwość definiowania **podprogramów**

Podprogram (ang. *subroutine*)

Sekwencja instrukcji wykonująca konkretne zadanie – zdefiniowana w postaci jednostki, którą można wywołać (ang. *callable unit*).
W różnych językach programowania może być nazywany:

- funkcją
- procedurą
- metodą

Programowanie proceduralne

Programowanie proceduralne to metodyka, według której kod programu powinien być podzielony na **podprogramy**.

- Każdy **podprogram** może być wywołany w dowolnym miejscu wykonywanego programu:
  - przez inny podprogram,
  - przez siebie samego.
- **Podprogramy** powinny pobierać wszystkie dane jako argumenty wywołania. Wynikiem ich działania może być:
  - wartość zwracana (ang. *return value*) oraz/lub
  - modyfikacja argumentu wejściowego (tzw. **skutek uboczny**)

Podprogram jest podmiotem wykonywanych działań, a dane ich przedmiotem

Programowanie strukturalne - wprowadzenie historyczne

Pierwszy kryzys oprogramowania

W latach 60 i 70 ubiegłego wieku po raz pierwszy zauważono, że tworzenie coraz to bardziej złożonego oprogramowania, działającego efektywnie i bezbłędnie, jest problemem **trudnym** a w związku z tym coraz bardziej kosztownym.

Kryzys oprogramowania przejawiał się między innymi w tym, że stworzone oprogramowanie było:

- nieefektywne,
- często nie spełniało postawionych wymagań,
- jego kod źródłowy był niskiej jakości, a w związku z tym trudny w "utrzymaniu".

Dlatego wiele projektów kończyło się:

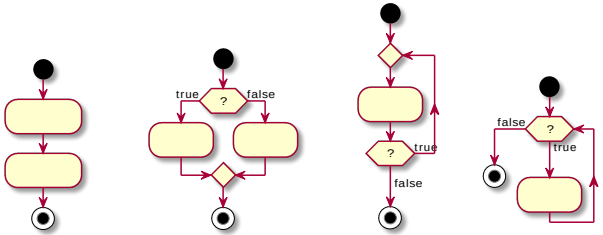
- przekroczeniem przyjętego budżetu,
- przekroczeniem założonych ram czasowych wykonania.

W celu poprawienia "jakości" tworzonego oprogramowania wprowadzono metodykę **programowania strukturalnego**

Twierdzenie Böhma-Jacopiniego

Do opisania dowolnego algorytmu wystarcza **diagram przepływu** (inaczej: **sieć działań**, ang. *flowchart*) złożony tylko z trzech typów struktur kontrolnych:

- sekwencji
- selekcji
- iteracji



sekwencja selekcja iteracja

Edsger Dijkstra – prace

Holenderski naukowiec uważany za jednego z "ojców" **programowania strukturalnego**.

W swoich pracach postulował wprowadzenie **matematycznego dowodzenia** poprawności implementacji algorytmów poprzez:

- ich rekursywną dekompozycję na coraz to mniejsze elementy do poziomu podstawowych jednostek
- matematycznego dowodzenia poprawności działania tych podstawowych jednostek

Podczas swoich prac zauważył, że:

- istnieją pewne sposoby stosowania instrukcji **goto**, które **uniemożliwiają** rekursywną dekompozycję problemu,
- istnieją także takie sposoby stosowania instrukcji **goto**, które nie wprowadzają tego problemu (odpowiadają one operacjom: selekcji lub iteracji).

Edsger Dijkstra – prace cd.

Dijkstra wykazał, że:

Poprawność działania **instrukcji sekwencyjnych** oraz **selekcji** może być dowiedziona przy użyciu enumeracji.

Poprawność działania **instrukcji iteracyjnych** może być dowiedziona przy użyciu techniki indukcji.

Niestety dowodzenie tego typu było **bardzo pracochłonne** i nie przyjęło się w praktycznych zastosowaniach.

Proszę zauważyć:

Struktury kontrolne programów, które pozwalają na matematyczne wyprowadzenie dowodu poprawności działania, są **tymi samymi** strukturami kontrolnymi, dzięki którym można skonstruować implementację dowolnego algorytmu zgodnie z twierdzeniem Böhma-Jacopiniego.

Instrukcja skoku **goto**

Instrukcja występująca w wielu starszych językach programowania, umożliwiająca przekazanie sterowania do dowolnej innej instrukcji w ramach działającego procesu. Miejsce przekazania sterowania oznaczone jest poprzez etykietę (label:)

Przykład w języku C++

```
int main() {
  bool condition = true;
START:
  if (condition)
    goto START;
  else
    goto END;
END:
  return 0;
}
```

Przykład w języku PHP

```
<?php
$condition=True;
START:
if($condition==True) goto START;
if($condition==False) goto END;
END:
echo "End"
?>
```

Instrukcja skoku **goto** – cd.

Go To Statement Considered Harmful

W 1968 r. Edsger Dijkstra napisał list do redakcji czasopisma "Communications of the Association for Computing Machinery", w którym zasugerował, że używanie instrukcji **goto** jest szkodliwe i prowadzi to tworzenia nieczytelnego i trudno-testowalnego kodu (ang. *spaghetti code*[1]).
Wywołał on szeroką dyskusję w środowisku programistów, która trwała ponad 10 lat.

Ostatecznie jego stanowisko zwyciężyło i nowoczesne języki programowania, takie jak: *Python*, *Java*, *Javascript*, *Rust*, nie udostępniają tej instrukcji.
Dodatkowo języki, które mają wbudowaną tą instrukcję zazwyczaj **ograniczają** zakres celu skoku (np: do ciała aktualnej funkcji)

[1] Spaghetti code

Termin oznaczający skomplikowany i trudny do zrozumienia kod źródłowy, powstały np. z powodu nadużywania instrukcji **goto**.

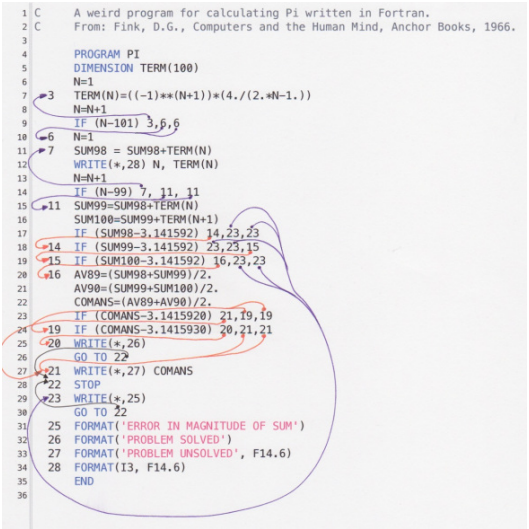
Spaghetti code – przykład

Fortran **IF**

```
IF (value)
    3, 6, 9
```

odpowiada semantycznie

```
IF (value)
    (value<0) THEN GOTO 3
    (value==0) THEN GOTO 6
    (value>0) THEN GOTO 9
```



```
C     A weird program for calculating Pi written in Fortran.
C     From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.

      PROGRAM PI
      DIMENSION TERM(100)
      N=1
    3 TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
      N=N+1
      IF (N-101) 3,6,6
    6 N=1
    7 SUM98 = SUM98+TERM(N)
      WRITE(*,28) N, TERM(N)
      N=N+1
      IF (N-99) 7, 11, 11
   11 SUM99=SUM98+TERM(N)
      SUM100=SUM99+TERM(N+1)
      IF (SUM98-3.141592) 14,23,23
   14 IF (SUM99-3.141592) 23,23,15
   15 IF (SUM100-3.141592) 16,23,23
   16 AV89=(SUM98+SUM99)/2.
      AV90=(SUM99+SUM100)/2.
      COMANS=(AV89+AV90)/2.
      IF (COMANS-3.1415920) 21,19,19
   19 IF (COMANS-3.1415930) 20,21,21
   20 WRITE(*,26)
      GO TO 22
   21 WRITE(*,27) COMANS
   22 STOP
   23 WRITE(*,25)
      GO TO 22
   25 FORMAT('ERROR IN MAGNITUDE OF SUM')
   26 FORMAT('PROBLEM SOLVED')
   27 FORMAT('PROBLEM UNSOLVED', F14.6)
   28 FORMAT(I3, F14.6)
      END
```

Źródło: https://craftofcoding.files.wordpress.com/2013/10/lore_spaghetti.pdf

Programowanie strukturalne

Każda metodyka programowania narzuca programiście pewien zestaw ograniczeń

Szeroka definicja programowania strukturalnego

"Metodyka programowania, która nakłada ograniczenia na **bezpośrednie** przekazywanie sterowania programem" (Robert C. Martin).

Definicja bardziej użyteczna

Metodyka programowania, która narzuca:

- stosowanie wysokopoziomowych struktur kontrolnych: selekcji, iteracji i sekwencji oraz unikanie stosowania niskopoziomowej instrukcji skoku,
- stosowanie rekursywnej dekompozycji problemu obliczeniowego typu *top-down* w celu zdefiniowania tzw. *podstawowych jednostek obliczeniowych*,
- każda *podstawowa jednostka* ma rozwiązywać pojedynczy problem programistyczny.

Programowanie strukturalne – cd.

Sposób rozwiązywania problemu obliczeniowego metodyką *programowania strukturalnego* można podsumować w następujący sposób:

- Pisz kod programu:
  - stosując struktury kontrolne typu: selekcja, iteracja i sekwencja,
  - deklarując i wywołując podprogramy: $P_1, ..., P_n$.
- Definiuj implementację podprogramów $P_1, ..., P_n$ w ten sposób, że wykorzystuje ona wywołania innych podprogramów.
- Dekompozycję podprogramów wykonuj tak długo, aż ich implementacja będzie na tyle prosta, że nie będzie wymagać wywołań innych podprogramów.

Instrukcja selekcji

```
if expression:
    statement(s)
else:
    statement(s)
```

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
else:
    statement(s)
```

Instrukcje iteracji

```
while expression:
    statement(s)
```

```
for iterating_var in sequence:
    statement(s)
```

Definiowanie podprogramu przy użyciu funkcji

```
def function_name(parameters):
    statement(s)
    return [expression]
```

Python – odstępstwa od "idealnych" struktur kontrolnych

Wiele współczesnych języków programowania ma wbudowane instrukcje:

- **break**
- **continue**
- **exit**

których użycie może zmienić sposób działania głównie instrukcji iteracji

Python – break

```
for iterating_var in sequence:
    # loop code before condition
    if condition:
        break
    #loop code after condition
# code outside for-loop
```

```
while expression:
    # loop code before condition
    if condition:
        break
    #loop code after condition
# code outside while-loop
```

Po wykonaniu instrukcji **break** sterowanie wykonaniem programu zostanie przeniesione do wiersza `# code outside ... loop`

Python – continue

```
for iterating_var in sequence:
    # loop code before condition
    if condition:
        continue
    #loop code after condition
# code outside for-loop
```

```
while expression:
    # loop code before condition
    if condition:
        continue
    #loop code after condition
# code outside while-loop
```

Po wykonaniu instrukcji **continue** sterowanie wykonaniem programu zostanie przeniesione do linii `# loop code before condition`

Python sys.exit

```
import sys
sys.exit(0)
```

Wywołanie funkcji **exit** kończy działanie programu i zwraca do systemu operacyjnego kod zakończenia.

Programowanie strukturalne – przykład

Problem obliczeniowy

Znaleźć rozwiązania równania kwadratowego w dziedzinie liczb rzeczywistych i zespolonych

$$a \cdot x^2 + b \cdot x + c = 0$$

Krok 1

Definiujemy dedykowany podprogram, który będzie rozwiązywał problem obliczeniowy

```
#!/usr/bin/env python3
def quadraticEquationsSolver():
    pass

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

Programowanie strukturalne – przykład cd.

W ramach rozwiązywanego zadania można wyróżnić następujące podzadania:

- pobranie współczynników równania od użytkownika,
- właściwe obliczenia znajdujące miejsca zerowe,
- wyświetlenie uzyskanych wyników.

Na potrzeby bieżącego zadania obliczeniowego dodatkowo przyjęto:

- współczynniki równania są wprowadzane jako argumenty wywołania skryptu (3 liczby rzeczywiste w postaci: a b c),
- współczynniki te będą przechowywane w postaci 3-elementowej krotki,
- wyniki będą udostępniane jako 2-elementowa krotka liczb rzeczywistych lub zespolonych.

Programowanie strukturalne – przykład cd.

Krok 2

Dekompozycja na podzadania – wczytanie danych wraz z ich weryfikacją i właściwe wyznaczenie pierwiastków, jeśli dane są poprawne. Na razie żadna z tych funkcji nie jest zaimplementowana.

```
def quadraticEquationsSolver():
    status,coefficients=readCoefficients()
    if status:
        zeroes=findZeroes(coefficients)
        displayZeroes(zeroes)
    else:
        print("Error!! - wrong input format")
```

Zmienna *status* służy do raportowania o poprawności wczytanych współczynników

Krok 3

Implementacja podzadania: wczytanie wartości współczynników równania i ich weryfikacja.

```
import sys
import math

def readCoefficients():
    if len(sys.argv)-1!=3:
        return False, (0,0,0)
    else:
        a = float(sys.argv[1])
        b = float(sys.argv[2])
        c = float(sys.argv[3])
        if abs(a)!=0.0:
            return True, (a,b,c)
        else:
            return False, (0,0,0)
```

Programowanie strukturalne – przykład cd.

W ramach podzadania wyznaczania miejsc zerowych można wyróżnić kolejne podzadania:

- wyznaczenie wartości Δ,
- gdy $\Delta >= 0$, wyznaczenie miejsc zerowych w dziedzinie liczb rzeczywistych,
- gdy $\Delta < 0$, wyznaczenie miejsc zerowych w dziedzinie liczb zespolonych.

Krok 4

Dekompozycja podzadań dla podprogramu `findZeroes`

```
def findZeroes(coefficients):
    a,b,c=coefficients
    delta=calculateDelta(a,b,c)
    if delta>=0:
        zeroes=findRealZeroes(a,b,c,delta)
    else:
        zeroes=findComplexZeroes(a,b,c,delta)
    return zeroes
```

Krok 5

Implementacja podzadania obliczania wartości Δ

```
def calculateDelta(a,b,c):
    return b*b - 4*a*c
```

Krok 6

Implementacja podzadania wyznaczania zer w dziedzinie liczb rzeczywistych

```
def findRealZeroes(a,b,c,delta):
    x1=(-b-math.sqrt(delta))/(2*a)
    x2=(-b+math.sqrt(delta))/(2*a)
    return (x1,x2)
```

Programowanie strukturalne – przykład cd.

Krok 7

Implementacja podzadania wyznaczania zer w dziedzinie liczb zespolonych

```
def findComplexZeroes(a,b,c,delta):
    import cmath
    x1=(-b-cmath.sqrt(delta))/(2*a)
    x2=(-b+cmath.sqrt(delta))/(2*a)
```

Krok 8

Implementacja podzadania wyświetlania wyników

```
def displayZeroes(zeroes):
    print(zeroes)
```

Programowanie strukturalne - przykład - wyniki działania

```
$python3 example_8.py 1 0 0
(-0.0, 0.0)

$python3 example_8.py -2 -8 10
(1.0, -5.0)

$python3 example_8.py 2 2 2
((-0.5-0.8660254037844386j), (-0.5+0.8660254037844386j))
```