

Systemy pozycyjnego zapisu liczb

System liczbowy to zbiór reguł jednolitego zapisu i nazewnictwa liczb¹.

Systemy pozycyjne to metody zapisywania liczb w taki sposób, że w zależności od pozycji danej cyfry w ciągu, oznacza ona wielokrotność potęgi pewnej liczby uznawanej za bazę danego systemu². Pozycje cyfr wyznacza się licząc od 0.

Każda liczba całkowita $N \geq 2$ może być podstawą systemu liczbowego. Mówimy wówczas o systemie o podstawie N .

Do zapisu liczb wykorzystywane są cyfry $\{0-9\}$, a w dalszej kolejności wykorzystane mogą być litery $\{A-Z\}$.

$$W = \sum_i c_i \cdot N^i$$

gdzie:

W- wartość liczby

c_i – cyfra znajdująca się na pozycji i

N – podstawa systemu liczbowego

Najpopularniejsze systemy pozycyjne to:

- dziesiętny
- dwójkowy (binarny)
- ósemkowy
- szesnastkowy

Liczy zapisane w innych systemach niż dziesiętny oznacza się często podając w indeksie dolnym podstawę systemu w nawiasach okrągłych. Podając wartości w różnych systemach liczbowych warto oznaczać w ten sposób również wartości w systemie dziesiętnym.

System dziesiętny

Dziesiętny system liczbowy³ (zwany też systemem decymalnym lub arabskim) to pozycyjny system liczbowy, w którym podstawą jest liczba 10. Do zapisu liczb potrzebne jest w nim 10 cyfr: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Jak w każdym pozycyjnym systemie liczbowym, liczby zapisuje się jako ciąg cyfr. Każda cyfra jest mnożnikiem kolejnej potęgi liczby 10. Część całkowitą i ułamkową oddziela separator dziesiętny. Separatorem dziesiętnym może być przecinek lub kropka.

Przykład:

$$437 = 4 \cdot 100 + 3 \cdot 10 + 7 \cdot 1 = 4 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0$$

$$3,14 = 3 \cdot 1 + 1 \cdot 0,1 + 4 \cdot 0,01 = 3 \cdot 10^0 + 1 \cdot 10^{-1} + 4 \cdot 10^{-2}$$

¹ http://pl.wikipedia.org/wiki/System_liczbowy

² http://pl.wikipedia.org/wiki/Systemy_pozycyjne

³ http://pl.wikipedia.org/wiki/Dziesi%C4%99tny_system_liczbowy

System binarny (dwójkowy)

Dwójkowy system liczbowy⁴ (zwany też systemem binarnym) to system liczbowy, w którym podstawą jest liczba 2. Do zapisu liczb potrzebne są tylko dwie cyfry: 0 i 1.

System binarny znalazł powszechne zastosowanie w informatyce. Wszelkie informacje przechowywane, przetwarzane oraz przesyłane w systemach komputerowych mają postać binarną.

Zamiana liczb całkowitych na system binarny

Aby obliczyć wartość binarną liczby podanej w systemie dziesiętnym należy wyznaczyć kolejne reszty z dzielenia jej przez 2 i ustawić w kolejności od ostatniej do pierwszej.

Przykład:

Liczba $437_{(10)}$ w kodzie binarnym wynosi $110110101_{(2)}$.

437	2	↑
218	1	
109	0	
54	1	
27	0	
13	1	
6	1	
3	0	
1	1	
0	1	

Zamiana liczb rzeczywistych na system binarny

Aby obliczyć wartość binarną liczby rzeczywistej podanej w systemie dziesiętnym należy osobno liczyć część całkowitą (przepis powyżej), a osobno część dziesiętną.

W części dziesiętnej mnożymy wartość po przecinku, aż do uzyskania 0. Jeżeli wartość mnożenia przekroczy 1, to w ciągu kolejnych cyfr stanowiących wartość binarną ułamka podajemy 1. W przeciwnym wypadku wpisujemy 0.

0	14
0	28
0	56
1	12
0	24
0	48
0	96
1	92
...	...

Na końcu sklejamy część całkowitą i dziesiętną oddzielając je separatorem dziesiętnym i otrzymując binarną wartość liczby rzeczywistej podanej w systemie dziesiętnym.

⁴ http://pl.wikipedia.org/wiki/Dw%C3%B3jkowy_system_liczbowy

Często zdarza się, że ułamek dziesiętny o skończonej liczbie cyfr może wymagać ułamka binarnego o nieskończonej liczbie cyfr. Wyznaczanie części dziesiętnej przerywamy wówczas, gdy dojdziemy do wartości powtarzającej się, okresowej.

$$3.14_{(10)} = 11,0010001\dots_{(2)} = 11,001000(1)_{(2)}$$

Zamiana liczb ujemnych na system binarny

Liczby ujemne na system binarny przelicza się tak samo jak dodatnie.

Przykład:

$$-437_{(10)} = -110110101_{(2)}$$

$$-3.14_{(10)} = -11,0010001\dots_{(2)} = -11,001000(1)_{(2)}$$

Zamiana liczb binarnych na system dziesiętny

Aby wyznaczyć wartość dziesiętną liczby binarnej wystarczy posłużyć się wzorem, 1 jako podstawę podając 2.

Poniższa tabela przedstawia kolejne potęgi liczby 2 od -4 do 16.

i	2 ⁱ	i	2 ⁱ	i	2 ⁱ
0	1	1	2	9	512
-1	0,5 (1/2)	2	4	10	1024
-2	0,25 (1/4)	3	8	11	2048
-3	0,125 (1/8)	4	16	12	4096
-4	0,0625 (1/16)	5	32	13	8192
		6	64	14	16384
		7	128	15	32768
		8	256	16	65536

Przykład:

$$110110101_{(2)} = 1 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$= 256 + 128 + 32 + 16 + 4 + 1 = 437_{(10)}$$

$$11,0010001_{(2)} = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7}$$

$$= 1 + 2 + 0,125 + 0,0078125 = 3 + 0,1328125 = 3,1328125 \approx 3,13_{(10)}$$

Jak widać z powyższego przykładu zaokrąglenia konieczne przy przeliczaniu liczb rzeczywistych na system binarny powodują, że po ponownym przeliczeniu wartości do systemu dziesiętnego i zaokrągleniu następuje utrata precyzji przechowywanej wartości.

Reprezentacja liczb w pamięci

Odrębną kwestią od przeliczania wartości liczb pomiędzy systemami pozycyjnymi jest przechowywanie ich w pamięci komputera. Stosowanie różnych typów zmiennych i stałych wyznacza, w jaki sposób interpretowane będą ciągi 0 i 1 przechowywane w pamięci.

Generalnie wartości przechowywane w pamięci komputera mają postać bitów. Bit może przyjąć wartość 1 lub 0. Stąd popularność systemu binarnego w informatyce.

Bity łączą się i tworzą bajty. Jedne bajt to 8 bitów.



Liczby całkowite dodatnie

Jeśli chodzi o liczby całkowite, to ich reprezentacja w pamięci jest prosta. Aby określić ile bitów jest potrzebne do zapamiętania danej wartości należy policzyć ilość cyfr tworzących reprezentację binarną danej liczby. Liczbę tą zawsze zaokrąglamy do pełnych bajtów. Liczbą uzupełniamy zerami po stronie najbardziej znaczących bitów.

Przykładowo liczba $437_{(10)}$ w systemie binarnym to $110110101_{(2)}$. Do jej zapisu potrzebne jest 9 cyfr. Zaokrąglając do pełnych bajtów otrzymamy 2 bajty, czyli 16 bitów. Pierwsze 7 bitów uzupełniamy zerami.

0	0	0	0	0	0	0	1	1	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

W języku C do przechowywania wartości całkowitych dodatnich służą typy: unsigned int, unsigned long oraz unsigned char. Ponieważ przechowywane są tylko wartości dodatnie, wszystkie bity służą do przechowania wartości liczby. Ilość bitów, na których wartość zostanie zapisana ogranicza zakres wartości możliwych do zapisania.

Nazwa typu	Zakres	Zajętość pamięci
unsigned int	<0;65535>	16 bits
unsigned long (unsigned long int)	<0;4294967295>	32 bits
unsigned char	<0;255>	8 bits

Liczby całkowite ujemne

Najprostszą reprezentacją liczb całkowitych ujemnych jest reprezentacja znak – moduł. Najbardziej znaczący bit zostaje przeznaczony na bit znaku. Pozostałe bity służą do zapisu modułu liczby.

Bit znaku może przyjąć wartość 0 dla liczb dodatnich lub 1 dla liczb ujemnych.



Poniżej znajduje się reprezentacja liczby $-437_{(10)}$ w pamięci komputera z wykorzystaniem notacji znak – moduł.

1	0	0	0	0	0	0	1	1	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Taki sposób zapisu liczb ze znakiem ma kilka wad. Najistotniejsze z nich to:

- możliwość zakodowania 0 na dwa sposoby:
 - 0000000000000000
 - 1000000000000000
- błędy w przypadku dokonywania działań na liczbach ze znakiem (przy wykonywaniu działań bit znaku jest traktowany jak każdy inny bit)

Rozwiązaniem tego problemu jest tzw. kod uzupełnieniowy. W kodzie tym liczby dodatnie reprezentowane są jak w przypadku zapisu znak – moduł. Natomiast wartości binarna liczb ujemnych wyznaczane są według następujących zasad:

1. Zapisujemy liczbą wraz ze znakiem w kodzie prostym na odpowiedniej liczbie bitów

1	0	0	0	0	0	0	1	1	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

2. Dokonujemy negacji wszystkich bitów liczby z wyjątkiem bitu znaku

1	1	1	1	1	1	1	0	0	1	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

3. Do wyniku dodajemy jedynekę.

1	1	1	1	1	1	1	0	0	1	0	0	1	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Powyższy przykład przedstawia zapis uzupełnieniowy liczby $-437_{(10)}$.

Aby wyznaczyć kod prosty liczby w kodzie uzupełnieniowym:

1	1	1	1	1	1	1	0	0	1	0	0	1	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1. Dokonujemy negacji wszystkich bitów liczby za wyjątkiem bitu znaku.

1	0	0	0	0	0	0	1	1	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

2. Do wyniku dodajemy jedynekę.

1	0	0	0	0	0	0	1	1	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

W tej notacji $0_{(10)}$ jest kodowane tylko w jeden sposób. Bit znaku w dalszym ciągu mówi nam, czy mamy do czynienia z liczbą dodatnią czy ujemną.

Aby zamienić liczbę dziesiętną (moduł liczby – b) bezpośrednio na notację U_2 (u) korzystamy z następującego wzoru: $u = 2^N - b$ gdzie N oznacza liczbę bitów, na których zapisujemy liczbę.

Przykładowo dla liczby -437:

$$u = 2^{16} - 437 = 65536 - 437 = 65099_{(10)} = 1111111001001011_{(2)}$$

Konwersja liczby zapisanej w kodzie uzupełnieniowym na system dziesiętny odbywa się tak jak konwersja z kodu prostego z tą różnicą, że cyfra oznaczająca bit znaku jest mnożona przez ujemną wartość odpowiedniej potęgi liczby 2.

$$\begin{aligned}
 &1111111001001011_{(U_2)} \\
 &= 1 \cdot -(2^{15}) + 1 \cdot 2^{14} + 1 \cdot 2^{13} + 1 \cdot 2^{12} + 1 \cdot 2^{11} + 1 \cdot 2^{10} + 1 \cdot 2^9 + 0 \cdot 2^8 + 0 \\
 &\quad \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
 &= -32768 + 16384 + 8192 + 4096 + 2048 + 1024 + 512 + 64 + 8 + 2 + 1 \\
 &= -437_{(10)}
 \end{aligned}$$

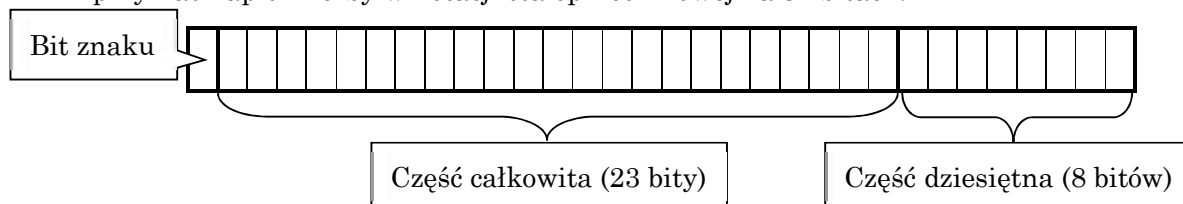
Takie rozdysponowanie bitów powoduje, że możemy zapisać wartości o połowę mniejsze niż miało to miejsce w przypadku liczb całkowitych bez znaku. Możemy jednak zapisać zarówno liczby dodatnie jak i ujemne. W języku C liczby całkowite ze znakiem mogą być przechowywane w zmiennych typów `int`, `long` oraz `char`.

Można zauważyć, że przedziały nie są symetryczne. Wartości ujemne możemy zapisać do modułu o 1 większego niż wartości dodatnie.

Nazwa typu	Zakres	Zajętość pamięci
int (signed int, short int)	<-32768;32767>	16 bits
long (long int)	<-2147483648; 2147483647>	32 bits
char	<-128;127>	8 bits

Liczby rzeczywiste (notacja stałoprzecinkowa)

Jednym ze sposobów przechowywania liczb rzeczywistych jest notacja stałoprzecinkowa. W notacji tej liczba bitów przeznaczonych na część całkowitą i ułamkową jest stała. Poniżej przykład zapisu liczby w notacji stałoprzecinkowej na 32 bitach.



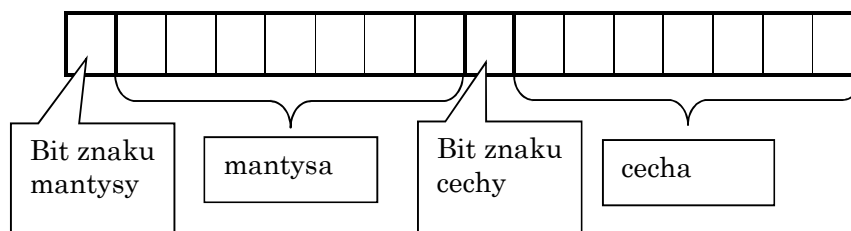
Powoduje to pewne ograniczenia. Przykładowo liczba $10000000000000000000000(2)$ nie będzie mogła zostać zapisana w tej notacji. Z kolei liczba $0,111111111(2)$ zostanie zapisana z dokładności jedynie do 8 cyfr w części ułamkowej.

Liczby rzeczywiste (notacja zmiennoprzecinkowa)

Alternatywą dla notacji stałoprzecinkowej jest notacja zmiennoprzecinkowa zwana również notacją półlogarytmiczną. Każda liczba rzeczywista X może zostać zapisana w postaci

$$X = \text{mantysa} * 2^{\text{cecha}}$$

W pamięci komputera zapisywane są mantysa i cecha wraz z ich znakami.



W notacji zmiennoprzecinkowej wybiera się taki sposób zapisu, w którym mantysa (w zapisie binarnym) zaczyna się od 0,1. Reprezentacja spełniająca ten warunek nazywana jest znormalizowana.

Kod BCD

BCD⁵ (ang. Binary-Coded Decimal – dziesiętny zakodowany dwójkowo) – sposób zapisu liczb polegający na zakodowaniu kolejnych cyfr dziesiętnych liczby dwójkowo przy użyciu czterech bitów stosowany w elektronice i informatyce. Taki zapis pozwala na łatwą konwersję liczby do i z systemu dziesiętnego, jest jednak nadmiarowy (wykorzystuje tylko 10 czterobitowych układów z 16 możliwych).

Przykładowo:

$$437_{(10)} = 0100\ 0011\ 0111_{(BCD)}$$

$$10\ 0000\ 0001\ 0010_{(BCD)} = 2012_{(10)}$$

System ósemkowy

System ósemkowy, czyli system, którego podstawą jest liczba 8. W zapisie wykorzystywane są cyfry 0, 1, 2, 3, 4, 5, 6 oraz 7.

Aby odróżnić liczbę zapisaną w systemie ósemkowym, zamiast dodawania w indeksie podstawy systemu, można poprzedzić ją cyfrą 0.

437₍₈₎ to to samo co 0437

Konwersja z systemu dziesiętnego na system ósemkowy i odwrotnie odbywa się według tych samych zasad co w przypadku systemu binarnego.

437	8
54	5
6	6
0	6

$$437 = 0665$$

$$665_{(8)} = 6 \cdot 8^2 + 6 \cdot 8^1 + 5 \cdot 8^0 = 6 \cdot 64 + 6 \cdot 8 + 5 = 384 + 48 + 5 = 437_{(10)}$$

System szesnastkowy

System szesnastkowy, czyli system, którego podstawą jest liczba 16. W zapisie wykorzystywane są wszystkie cyfry od 0 do 9 oraz litery A, B, C, D, E oraz F.

Aby odróżnić liczbę zapisaną w systemie szesnastkowym, zamiast dodawania w indeksie podstawy systemu, można poprzedzić ją zapisem 0x.

437₍₁₆₎ to to samo co 0x437

Konwersja z systemu dziesiętnego na system szesnastkowy i odwrotnie odbywa się według tych samych zasad co w przypadku systemu binarnego.

437	16
27	5
1	B
0	1

$$437 = 0x1B5$$

$$1B5_{(16)} = 1 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0 = 256 + 11 \cdot 16 + 5 = 256 + 176 + 5 = 437_{(10)}$$

⁵ http://pl.wikipedia.org/wiki/Kod_BCD

Konwersje pomiędzy systemami szesnastkowym i ósemkowym a binarnym

Konwersja pomiędzy systemami ósemkowym i szesnastkowym a binarnym są dużo prostsze. Może w tym celu wykorzystana zostać poniższa tabelka.

10	2	16	8
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20

10	2	16	8
17	10001	11	21
18	10010	12	22
19	10011	13	23
20	10100	14	24
21	10101	15	25
22	10110	16	26
23	10111	17	27
24	11000	18	30
25	11001	19	31
26	11010	1A	32
27	11011	1B	33
28	11100	1C	34
29	11101	1D	35
30	11110	1E	36
31	11111	1F	37
32	100000	20	40

System ósemkowy

Przy konwersji z systemu ósemkowego na system binarny każda z cyfr liczby zapisanej ósemkowo jest konwertowana na trzycyfrową liczbą binarną.

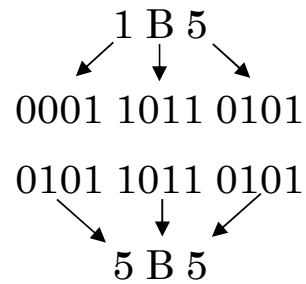
6 6 5
 ↙ ↓ ↘
 110 110 101

Analogicznie w przypadku konwersji z systemu binarnego na szesnastkowy. Liczba jest dzielona na trzycyfrowe grupy zaczynając od najmniej znaczącego bitu. Następnie każda trójka jest konwertowana na cyfrę ósemkową.

10 110 110 101
 ↘ ↙ ↘ ↙
 2 6 6 5
 101 101 101 01 błąd

System szesnastkowy

Analogicznie wygląda konwersja pomiędzy systemami binarnym i szesnastkowym z tą różnicą, że jednej cyfrze szesnastkowej odpowiadają cztery cyfry binarne, a nie trzy.



Działania na liczbach binarnych i szesnastkowych

Arytmetyka binarna

Uwaga! Błędy nadmiaru i niedomiaru

Uwaga! Liczba bitów przy kodzie uzupełnieniowym U2

Dodawanie

$$\begin{array}{r} 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1 \\ +\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1 \\ \hline 1\ 1\ 0\ 0\ 0\ 0\ 1\ 0 \end{array}$$

Dodawanie z nadmiarem na 4 bitach

$$1111\ (15_{(10)}) + 1 = 10000\ (0_{(10)})$$

$$\begin{array}{r} 1\ 1\ 1\ 1 \\ +\ 0\ 0\ 0\ 1 \\ \hline 0\ 0\ 0\ 0 \end{array}$$

Odejmowanie

$$\begin{array}{r} 0\ 1\ 1\ 0\ 1\ 1\ 0\ 0 \\ -\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1 \\ \hline 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1 \end{array}$$

Odejmowanie z niedomiarem na 4 bitach

$$1001\ (9_{(10)}) - 1101\ (13_{(10)}) = 1100\ (12_{(10)})$$

$$\begin{array}{r} 1\ 0\ 0\ 1 \\ +\ 1\ 1\ 0\ 1 \\ \hline 1\ 1\ 0\ 0 \end{array}$$

Mnożenie

$$\begin{array}{r} 1\ 0\ 1\ 1\ 0 \\ * 1\ 0\ 1 \\ \hline 1\ 0\ 1\ 1\ 0 \\ +\ 1\ 0\ 1\ 1\ 0 \\ \hline 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0 \end{array}$$

Dzielenie

$$\begin{array}{r} 1\ 0\ 1 \\ \hline 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0 : 1\ 1\ 0\ 0\ 1\ 0 \\ -\ 1\ 1\ 0\ 0\ 1\ 0 \\ \hline 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \\ -\ 1\ 1\ 0\ 0\ 1\ 0 \\ \hline 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

Operacje logiczne w systemie binarnym

a=10110011

b=01011010

c=00000000

Koniunkcja

a&b=00000001

a&c=00000000

Alternatywa

a | b=00000001

a | c=00000001

Negacja

!a=00000000

!b=00000000

!c=00000001

Iloczyn bitowy

a&b=00010010

	1	0	1	1	0	0	1	1
&	0	1	0	1	1	0	1	0
<hr/>								
	0	0	0	1	0	0	1	0

a&c=00000000

	1	0	1	1	0	0	1	1
&	0	0	0	0	0	0	0	0
<hr/>								
	0	0	0	0	0	0	0	0

Suma bitowa

a | b=11111011

	1	0	1	1	0	0	1	1
	0	1	0	1	1	0	1	0
<hr/>								
	1	1	1	1	1	0	1	1

a | c=10110011

	1	0	1	1	0	0	1	1
	0	0	0	0	0	0	0	0
<hr/>								
	1	0	1	1	0	0	1	1

Bitowa różnica symetryczna (alternatywa wykluczająca, suma modulo dwa)

$$a \oplus b = 11101001$$

$$\begin{array}{r} 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\ \oplus \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \\ \hline 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \end{array}$$

$$a \oplus c = 10110011$$

$$\begin{array}{r} 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\ \oplus \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ \hline 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \end{array}$$

Negacja bitowa

$$\sim 10110011 = 01001100$$

$$\sim 01011010 = 10100101$$

$$\sim 00000000 = 11111111$$

Przesunięcia bitowe

$$a/8 = a \gg 3 = 00010110 \text{ (bo } 2^3=8)$$

$$\begin{array}{r} 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\ \gg \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \mid 0 \ 1 \ 1 \end{array}$$

$$b*2 = b \ll 1 = 10110100 \text{ (bo } 2^1=2)$$

$$\begin{array}{r} 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \\ \ll \ 0 \mid 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \end{array}$$

Ustawianie zadanego bitu

Ustawiamy czwarty bit w słowie binarnym 110001001011 na 1=10110100.

	110001001011	zmieniane słowo
OR()	000000010000	maska bitowa
	110001011011	wynik operacji

Zerowanie zadanego bitu

Zerujemy czwarty bit w słowie binarnym 110001010111.

	110001010111	zmieniane słowo
AND(&)	111111101111	maska bitowa
	110001000111	wynik operacji

Negacja zadanego bitu.

Zmieniamy stan czwartego bitu w słowie binarnym 110001000111.

	110001000111	zmieniane słowo
XOR(^)	000000010000	maska bitowa
	110001011111	wynik operacji

Arytmetyka binarna w kodzie uzupełnieniowym U2**Dodawanie i odejmowanie liczb w kodzie uzupełnieniowym**

Liczbę w kodzie uzupełnieniowym dodajemy i odejmujemy tak samo jak w kodzie prostym. Przeniesienia poza bit znaku ignorujemy.

$$5 + (-3) = 2$$

$$\begin{array}{r} 0 \ 1 \ 0 \ 1 \\ + \ 1 \ 1 \ 0 \ 1 \\ \hline 1 \ 0 \ 0 \ 1 \ 0 \end{array}$$

Mnożenie w kodzie uzupełnieniowym

Przed wykonaniem operacji rozszerzamy znakowo obie mnożone liczby tak, aby ich liczba bitów wzrosła dwukrotnie. Rozszerzenie znakowe polega na powielaniu bitu znaku na wszystkie dodane bity.

$$(-2) \times 3 = (-6)$$

$$\begin{array}{r} 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \\ * \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \\ \hline 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \\ + \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \\ \hline 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \end{array}$$

Wynik mnożenia musi być liczbą o długości równej sumie długości mnożonych przez siebie liczb (tutaj $4+4=8$).

Dzielenie w kodzie uzupełnieniowym.

Zapamiętujemy znaki dzielonych liczb. Zamieniamy liczby na dodatnie. Dokonujemy dzielenia binarnego. Zmieniamy liczbę na liczbę przeciwną jeśli znaki dzielnej i dzielnika były różne. Jeśli w wyniku dzielenia otrzymamy resztę, to musi ona mieć ten sam znak, co dzielna.

$$6 : (-3) = (-2)$$

$$6 = 0110_{(2)}$$

$$-3 = 1101_{(U2)} - \text{zmieniamy na } 3 = 0011_{(2)}$$

$$0100 : 1101 = 0010 = 2_{(10)}$$

liczby miały różne znaki, więc zamieniamy na $-2 = 1110_{(U2)}$

Arytmetyka szesnastkowa

Działania wykonuje się analogicznie jak przy arytmetyce dziesiętnej i binarnej. Poniżej pomocna w obliczeniach szesnastkowa tabliczka mnożenia. Wszystkie wartości w niej są w systemie szesnastkowym.

Strona 5

Maszynowa reprezentacja informacji nienumerycznych

Kodowanie danych

Kodowaniem nazywamy przyporządkowanie wiadomościom (wg reguły zw. kodem) sygnałów w postaci ciągów sygnałów elementarnych¹. W informatyce oznacza ono przekształcenie informacji do postaci cyfrowej (ciągów zer i jedynek) w celu wygodnego jej przesyłania lub w celu uniemożliwienia dostępu do przekazywanych informacji osobom niepowołanym.

Dwoma kluczowymi zagadnieniami związanymi z kodowaniem danych są:

- kompresja – pozwalająca na zmniejszenie ilości przesyłanej informacji,
- bezpieczeństwo danych – zapewniające ochronę przed niepowołanym dostępem.

Podstawowe grupy kodów:

- równomierne – o stałej długości słowa kodowego,
- nierównomierne – o zmiennej długości słowa kodowego.

W obydwu przypadkach można do zakodowanej wiadomości dołączyć pewne dodatkowe bity kontrolne, ułatwiające odtworzenie informacji, nawet w przypadku częściowego uszkodzenia przesyłanego komunikatu. Uzyskujemy wówczas **kody nadmiarowe**.

Reprezentacja znaków

Dane wprowadzane za pomocą klawiatury, czyli znaki alfabetu, cyfry i inne znaki nienumeryczne (np.: „A”, „5”, „#”) nazywane są **znakami alfanumerycznymi**. Mają one postać znaków pisarskich – liter, cyfr, znaków przestankowych i innych symboli.

Obecne konstrukcje komputerów wymuszają, aby informacje przetwarzane przez maszyny miały postać zero-jedynkową. Chcąc zatem przedstawić w komputerze znaki alfanumeryczne, należy przyjąć umowę ich oznaczania za pomocą ciągów bitów.

Taką procedurę nazywamy **kodowaniem lub mapowaniem znaków**. Jest to przyporządkowanie znakom alfanumerycznym odpowiedników liczbowych, ciągów sygnałów binarnych.

Zasada działania kodowania znaków polega na tabelarycznym zestawieniu:

- zbioru znaków (charset),
- i ich liczbowych odpowiedników.

¹ Słownik Języka Polskiego PWN

W systemach komputerowych znaki reprezentowane są więc przez wartości numeryczne (kody znaków), określające pozycję danego symbolu w tablicy kodowej. Kody znaków są liczbami całkowitymi i przechowywane są w postaci binarnej.

Kod ASCII

Jednym z najczęściej stosowanych standardów kodowania znaków jest kod ASCII (ang. *American Standard Code for Information Interchange*), posiadający 128 znaków. Kod ten jest 7-bitowy, co oznacza, że poszczególnym znakom przypisano liczby z zakresu 0-127 (2^7). W zestawie znaków ASCII znajdują się litery alfabetu łacińskiego, cyfry, znaki interpunkcyjne i polecenia sterujące sposobem wyświetlania znaków na drukarce lub terminalu znakowym (np. LF (ang. *Line Feed*) – znak nowej linii, HT (ang. *Horizontal Tab*) – tabulator poziomy).

Tabela 1. Kod ASCII – kody sterujące

Dec	Hex	Znak	Skrót	Dec	Hex	Znak	Skrót
0	00	Null	NU L	18	12	Device Control 2	DC2
1	01	Start Of Heading	SOH	19	13	Device Control 3 (XOFF)	DC3
2	02	Start of Text	STX	20	14	Device Control 4	DC4
3	03	End of Text	ETX	21	15	Negative Acknowledge	NA K
4	04	End of Transmission	EOT	22	16	Synchronous Idle	SYN
5	05	Enquiry	EN Q	23	17	End of Transmission Bl ock	ETB
6	06	Acknowledge	ACK	24	18	Cancel	CAN
7	07	Bell	BEL	25	19	End of Medium	EM
8	08	Backspace	BS	26	1A	Substitute	SUB
9	09	Horizontal Tab	HT	27	1B	Escape	ESC
10	0A	Line Feed	LF	28	1C	File Separator	FS
11	0B	Vertical Tab	VT	29	1D	Group Separator	GS
12	0C	Form Feed	FF	30	1E	Record Separator	RS
13	0D	Carriage Return	CR	31	1F	Unit Separator	US
14	0E	Shift Out	SO	32	20	Spacja	
15	0F	Shift In	SI	12	7F	Delete	DEL
16	10	Data Link Escape	DLE	7			
17	11	Device Control 1 (XON)	DC1				

Tabela 2. Kod ASCII – cyfry, litery alfabetu, znaki interpunkcyjne

Dec	Hex	Znak	Dec	Hex	Znak	Dec	Hex	Znak	Dec	Hex	Znak
33	21	!	57	39	9	81	51	Q	105	69	i
34	22	"	58	3A	:	82	52	R	106	6A	j
35	23	#	59	3B	;	83	53	S	107	6B	k
36	24	\$	60	3C	<	84	54	T	108	6C	l
37	25	%	61	3D	=	85	55	U	109	6D	m
38	26	&	62	3E	>	86	56	V	110	6E	n
39	27	'	63	3F	?	87	57	W	111	6F	o
40	28	(64	40	@	88	58	X	112	70	p
41	29)	65	41	A	89	59	Y	113	71	q
42	2A	*	66	42	B	90	5A	Z	114	72	r
43	2B	+	67	43	C	91	5B	[115	73	s
44	2C	,	68	44	D	92	5C	\	116	74	t
45	2D	-	69	45	E	93	5D]	117	75	u
46	2E	.	70	46	F	94	5E	^	118	76	v
47	2F	/	71	47	G	95	5F	_	119	77	w
48	30	0	72	48	H	96	60	`	120	78	x
49	31	1	73	49	I	97	61	a	121	79	y
50	32	2	74	4A	J	98	62	b	122	7A	z
51	33	3	75	4B	K	99	63	c	123	7B	{
52	34	4	76	4C	L	100	64	d	124	7C	
53	35	5	77	4D	M	101	65	e	125	7D	}
54	36	6	78	4E	N	102	66	f	126	7E	~
55	37	7	79	4F	O	103	67	g			
56	38	8	80	50	P	104	68	h			

Litery, cyfry oraz inne znaki drukowane tworzą zbiór **znaków drukowalnych** – jest to 95 znaków o kodach 32-126. Pozostałe 33 kody (0-31 i 127) to tzw. **znaki niedrukowalne** (kody sterujące), służące do sterowania urządzeniem odbierającym komunikat, np. drukarką czy terminalem.

Do reprezentacji kodów początkowo używano 7 bitów. Ósmy bit służył do kontroli poprawności zapisu (tzw. bit parzystości). W momencie, gdy kontrolę poprawności zapisu zaczęto realizować innymi metodami, rozbudowano tablicę kodów do 256 znaków – powstał 8-bitowy **rozszerzony kod ASCII**.

Pierwsze 128 pozycji rozszerzonego kodu ASCII jest identyczna jak w wariantcie podstawowym. Kolejne pozycje zawierają jednak znaki dodatkowe, np. znaki semigraficzne (do tworzenia obramowań np. ¶, ¶¶, ||), litery akcentowane, rozszerzony zestaw symboli matematycznych, litery alfabetów narodowych.

Strony kodowe

Na 8 bitach można zakodować tylko 256 różnych znaków. Wielkość ta jest niewystarczająca do zmieszczenia w jednym zestawie znaków alfabetów wszystkich krajów świata. W rezultacie powstało wiele różnych rozszerzeń ASCII wykorzystujących ósmy bit, nazywanych stronami kodowymi (np. norma ISO 8859, rozszerzenia firm IBM lub Microsoft). **Strony kodowe** to wersje kodu ASCII różniące się interpretacją symboli o numerach od 128 do 255.

Duża liczba dostępnych stron kodowych niesie ze sobą następujące konsekwencje:

- różne strony kodowe mogą przyjąć odmienne znaki dla tego samego kodu,
- różne strony kodowe mogą różnić się wyborem znaków.

Najpopularniejszym zestawem standardów służących do kodowania znaków za pomocą 8 bitów jest ISO 8859. Standardy te zostały utworzone przez ECMA (ang. *European Computer Manufacturers' Association*) w połowie lat osiemdziesiątych, a następnie zostały zaakceptowane przez organizację normalizacyjną ISO (ang. *International Organization for Standardization*).

Lista standardów ISO 8859

- ISO 8859-1 (Latin-1) - alfabet łaciński dla Europy zachodniej
- ISO 8859-2 (Latin-2) - łaciński dla Europy środkowej i wschodniej, również odpowiednia Polska Norma
- ISO 8859-3 (Latin-3) - łaciński dla Europy południowej
- ISO 8859-4 (Latin-4) - łaciński dla Europy północnej
- ISO 8859-5 (Cyrillic) - dla cyrylicy
- ISO 8859-6 (Arabic) - dla alfabetu arabskiego
- ISO 8859-7 (Greek) - dla alfabetu greckiego
- ISO 8859-8 (Hebrew) - dla alfabetu hebrajskiego
- ISO 8859-9 (Latin-5)
- ISO 8859-10 (Latin-6)
- ISO 8859-11 (Thai) - dla alfabetu tajskiego
- ISO 8859-12 - brak
- ISO 8859-13 (Latin-7)
- ISO 8859-14 (Latin-8)
- ISO 8859-15 (Latin-9) - z ISO 8859-1 usunięto kilka rzadko używanych znaków i wprowadzono znak euro oraz litery Š, š, Ž, ž, Œ, œ oraz Ÿ
- ISO 8859-16 (Latin-10) - łaciński dla Europy środkowej - zmodyfikowany ISO 8859-2 ze znakiem euro i dodatkowymi literami dla kilku języków.

Kod ASCII – problem polskich znaków

Do znaków specyficznych dla języka polskiego (ĄĆĘŁŃÓŚŻąćęłńóśż) przypisane są kody większe od 128. Oznacza to, że zbiór tych znaków znajduje się w rozszerzonym kodzie ASCII. Ponadto, istnieje kilka stron kodowych, zawierających polskie znaki, co utrudnia wymianę dokumentów tekstowych pomiędzy różnymi systemami. Do najbardziej popularnych stron kodowych, zawierających polskie znaki należą:

- ISO 8859-2 (ISO Latin-2, Polska Norma PN-93/T-42118) – sposób kodowania określony przez ISO oraz zatwierdzony przez Polski Komitet Organizacyjny.
- Windows CP-1250 (Windows 1250, CP-1250) – sposób kodowania wprowadzony przez firmę Microsoft wraz z systemem Windows 3.11 PL. Strona ta jest bardzo podobna do ISO 8859-2 (większość znaków posiada identyczne kody), jednak biorąc pod uwagę polskie znaki diakrytyczne - różnice obejmują litery Ą, ś, ź, ż.

Tabela 3. Polskie znaki – standardy kodowania ASCII

Znak	ISO 8859-2		Windows CP-1250	
	DEC	HEX	DEC	HEX
Ą	161	#A1	165	#A5
Ć	198	#C6	198	#C6
Ę	202	#CA	202	#CA
Ł	163	#A3	163	#A3
Ń	209	#D1	209	#D1
Ó	211	#D3	211	#D3
Ś	166	#A6	140	#8C
Ż	172	#AC	143	#8F
Ż	175	#AF	175	#AF
ą	177	#B1	185	#B9
ć	230	#E6	230	#E6
ę	234	#EA	234	#EA
ł	179	#B3	179	#B3
ń	241	#F1	241	#F1
ó	243	#F3	243	#F3
ś	182	#B6	156	#9C
ź	188	#BC	159	#9F
ż	191	#BF	191	#BF

Różne kody polskich znaków diakrytycznych w powyższych systemach prowadzą często do problemów podczas wyświetlania dokumentów tekstowych. Przykład tekstu zapisanego w standardzie ISO 8859-2:

ĄĆĘŁŃÓŚŻąćęłńóśż

może zostać wyświetlony w edytorze tekstowym Windows w sposób następujący:

~ĆĘŁŃÓ!-Ż±ćęłńóŹLż

Unicode

Niejednoznaczność kodów ASCII powyżej kodu 127 oraz brak obsługi wielu języków na raz stanowią wadę tego standardu. W celu ujednolicenia sposobu interpretacji kodów liczbowych jako znaków oraz zapewnienia możliwości reprezentowania za ich pomocą zarówno tekstów międzynarodowych jak i pewnych dodatkowych symboli, zaproponowano nowy sposób kodowania znaków znany pod nazwą **Unicode** (Unikod).

Unicode jest uniwersalnym kodem znakowym, umożliwiającym reprezentację wszystkich znaków pisarskich zapisu fonetycznego używanych na całym świecie (np. znaki polskie, greckie, znaki hebrajskie, arabskie, chińskie, cyrylica), symbole muzyczne, techniczne i inne. W odróżnieniu od kodu ASCII kody te jednoznacznie identyfikują symbol, więc nie zaistnieje sytuacja, w której dany kod może oznaczać różne symbole w zależności od przyjętej strony kodowej. W rezultacie istnieje możliwość swobodnego mieszania znaków różnych krajów bez obawy o niejednoznaczność.

Znaki w Unikodzie podzielone zostały na:

- podstawowy zestaw znaków (określany jako Basic Multilingual Plane – BMP lub Plane 0) – dla tych znaków stosowane są kody 16 bitowe.
- dodatkowy zestaw znaków – stosowane są kody 32 bitowe.

Najpowszechniejszą metodą przechowywania unikodów w pamięci komputera jest **UTF** (ang. *Unicode Transformation Format*).

Rodzaje implementacji UTF:

- UTF-8 – kody znaków wchodzących w skład podstawowego zestawu ASCII zapisywane są jako wartości jednobajtowe; pozostałe kody zapisywane są na dwóch, trzech, czterech, pięciu lub sześciu bajtach (znaki o kodach zapisywanych na trzech i większej liczbie bajtów spotykane są we współczesnych językach bardzo rzadko).
- UTF-16 – kody znaków zapisywane są na dwóch, trzech lub czterech bajtach (najczęściej wykorzystywane są znaki o kodach dwubajtowych).
- UTF-32 – kody znaków zapisywane są na czterech bajtach (kodowanie czterobajtowe rozwiązuje problem języków ideograficznych; np. język chiński liczący tysiące znaków; cztery bajty dają $2^{32} = 4$ miliardy kombinacji).

Znaki końca wiersza

W systemie DOS/Windows każdy wiersz pliku zakończony jest parą znaków:

- CR, kod ASCII - 13(10) = 0D(16) – (powrót karetki; ang. *carriage return*) powrót na początek wiersza
- LF, kod ASCII - 10(10) = 0A(16) – (ang. *line feed*) przejście do nowej linii

W systemie Linux znakiem końca wiersza jest tylko:

- LF, kod ASCII - 10₍₁₀₎ = 0A₍₁₆₎ – (ang. *line feed*) przejście do nowej linii

Podczas przesyłania pliku tekstowego z systemu Linux do systemu DOS/Windows (np. poprzez FTP) pojedynczy znak LF zamieniany jest automatycznie na parę znaków CR LF. Błędne przesłanie pliku tekstowego (w trybie binarnym zamiast ASCII) powoduje nieprawidłowe jego wyświetlanie.

Przykładowy plik tekstowy:

```
First line
Second line
Third line
```

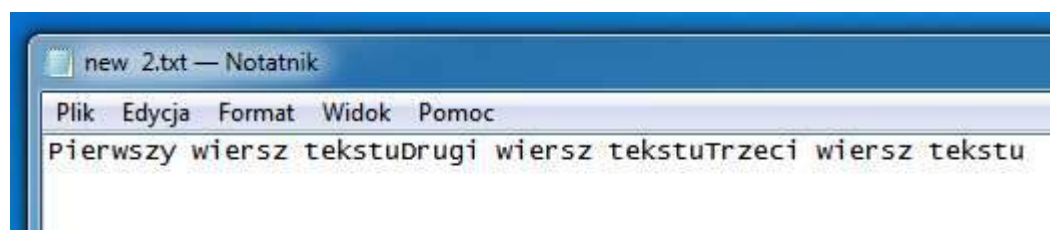
Znaki końca wiersza w systemie Windows:

00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
46	69	72	73	74	20	6c	69	6e	65	0d	0a	53	65	63	6f	First line..Seco
6e	64	20	6c	69	6e	65	0d	0a	54	68	69	72	64	20	6c	nd line..Third l
69	6e	65	ine.....

Znaki końca wiersza w systemie Linux:

00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
46	69	72	73	74	20	6c	69	6e	65	0a	53	65	63	6f	6e	First line..Secon
64	20	6c	69	6e	65	0a	54	68	69	72	64	20	6c	69	6e	d line..Third lin
65	e.....

Próba wyświetlenia pliku ze znakami końca wiersza LF w systemie Windows



Odwrotna Notacja Polska

Wprowadzenie teoretyczne do tematu

Odwrotna notacja polska¹ (ONP, ang. Reverse Polish Notation, RPN) – jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach (zapis postfiksowy), a nie pomiędzy nimi jak w konwencjonalnym zapisie algebraicznym (zapis infiksowy) lub przed operandami jak w zwykłej notacji polskiej (zapis prefiksowy). Zapis ten pozwala na całkowitą rezygnację z użycia nawiasów w wyrażeniach, jako że jednoznacznie określa kolejność wykonywanych działań.

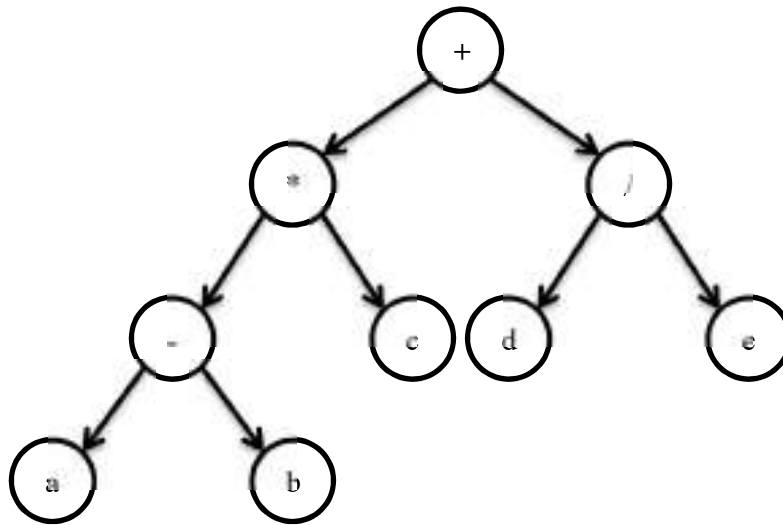
Poniższa tabelka przedstawia kilka wyrażeń arytmetycznych w zapisie tradycyjnym oraz w Odwróconej Notacji Polskiej. Przy zapisie tradycyjnym korzystanie z nawiasów jest konieczne w celu określenia kolejności wykonywania działań. W ONP nie używa się nawiasów.

Zapis infiksowy (tradycyjny)	Zapis postfiksowy (ONP)
$a + b$	$a b +$
$a + b + c$	$a b + c +$ (ab+ stanowi pierwszy argument drugiego dodawania)
$(a + b) * c$	$a b + c *$
$c * (a + b)$	$c a b + *$
$(a + b) * c + d$	$a b + c * d +$
$(a + b) * c + d * a$	$a b + c * d a * +$
$(a + b) * c + d * (a + c)$	$a b + c * d a c + * +$
$(a + b) * c + (a + c) * d$	$a b + c * a c + d * +$

Drzewa wyrażeń arytmetycznych

Jedno z zastosowań drzew binarnych to reprezentacja wyrażeń arytmetycznych. Wyrażenia arytmetyczne można reprezentować jako drzewa, gdzie w liściach pamiętane są liczby, a w węzłach symbole operacji arytmetycznych. W zależności od wybranego sposobu poruszania się po takim drzewie można uzyskać tradycyjny zapis (inorder), Notację Polską (preorder) lub Odwróconą Notację Polską (postorder).

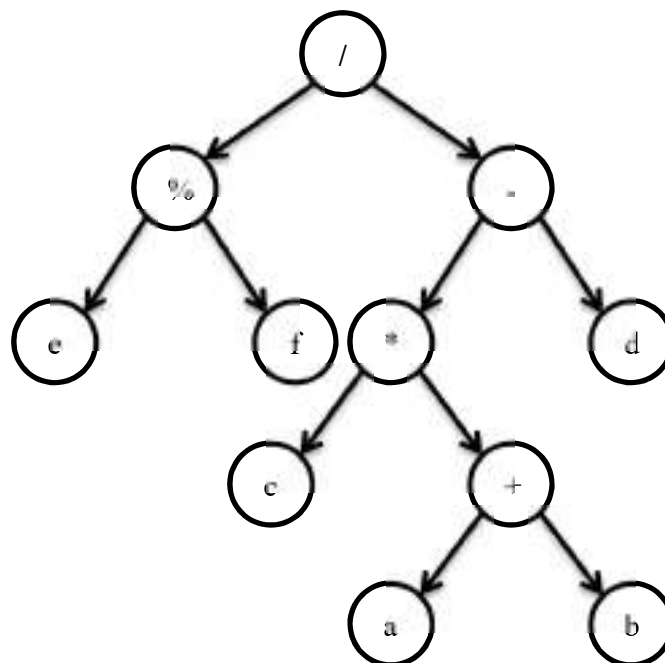
¹ http://pl.wikipedia.org/wiki/Odwrotna_notacja_polska



Wyrażenie zapisane na powyższym drzewie można zapisać zatem na trzy sposoby:

1. Zapis tradycyjny (infiksowy): $((a-b)*c)+(d/e)$
2. Notacja Polska (prefiksowa): $+*-abc/de$
3. Odwrotna Notacja Polska (postfiksowa): $ab-c*de/+$

Możliwe jest również tworzenie drzew wyrażeń arytmetycznych na podstawie zapisu wyrażenia w dowolnej notacji. Przykładowo dla zapisu w tradycyjnej notacji drzewo wyrażenia będzie wyglądało następująco: $(e\%f)/((c*(a+b))-d)$.

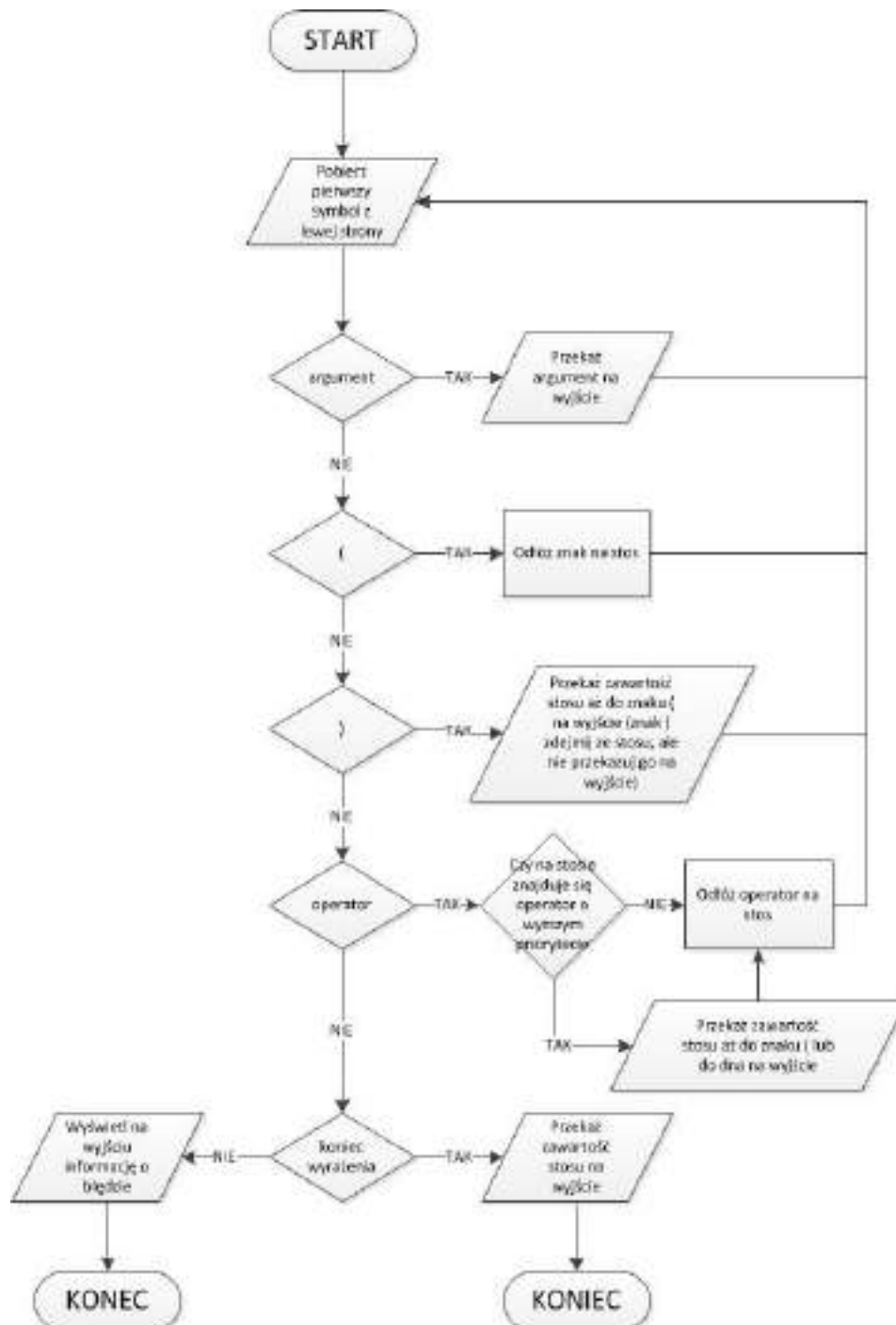


To samo wyrażenie w notacji postfiksowej $(ef\%cab+*d-/)$ i prefiksowej $(/\%ef-*c+abd)$.

Przy tworzeniu drzewa wyrażenia arytmetycznego ma znaczenie, po której stronie znajdzie się węzeł. Nie jest to dowolne.

Tworzenie drzewa dla notacji postfiksowej jest nieco trudniejsza gdyż zaczynamy od liści i kończymy na korzeniu. Z kolei dla notacji prefiksowej jest to łatwiejsze z uwagi na to, iż rozpoczynamy od korzenia i zmierzamy do liści.

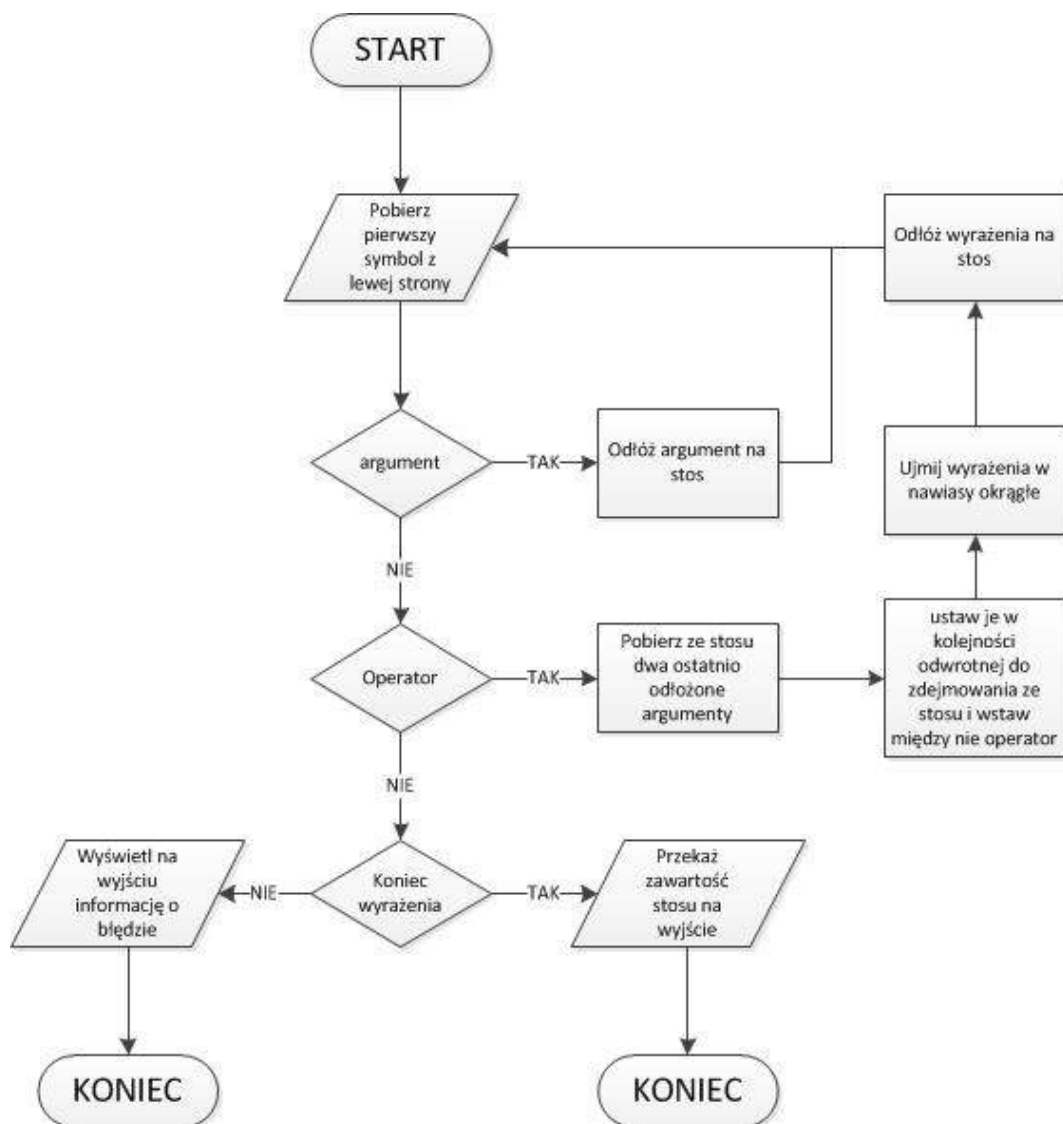
Konwersja z notacji algebraicznej do ONP



Powyższy schemat blokowy przedstawia algorytm konwersji wyrażenia z notacji algebraicznej do notacji ONP. Poniżej znajduje się operacja konwersji przykładowego wyrażenia krok po kroku według tegoż algorytmu.

Wyrażenie: $(e-f)+(a*b+c/d)$

Numer kroku	Wejście	Stos	Wyjście
1	((
2	e	(e
3	-	(-	e
4	f	-	ef
5)		ef-
6	+	+	ef-
7	(+(ef-
8	a	+(ef-a
9	*	+(*	ef-a
10	b	+(*	ef-ab
11	+	+(+	ef-ab*
12	c	+(+	ef-ab*c
13	/	+(+/ /	ef-ab*c
14	d	+(+/ /	ef-ab*cd
15)	+(+	ef-ab*cd/+
16	Koniec wyrażenia		ef-ab*cd/++

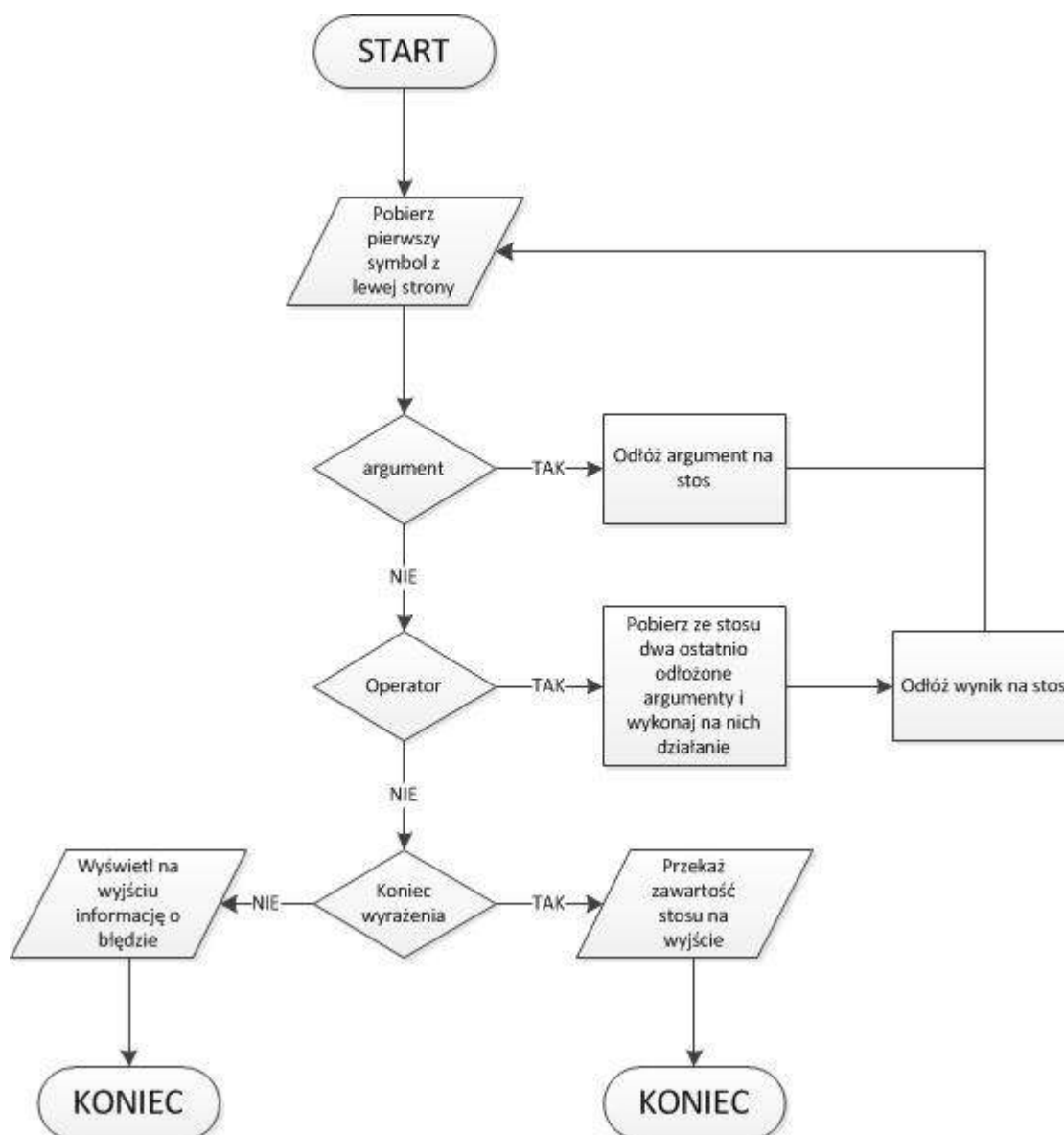
Konwersja z notacji ONP do algebraicznej

Powyższy schemat blokowy przedstawia algorytm konwersji wyrażenia z notacji ONP do notacji algebraicznej. Poniżej znajduje się operacja konwersji przykładowego wyrażenia krok po kroku według tegoż algorytmu.

Wyrażenie: $ef \cdot ab * cd / ++$

Numer kroku	Wejście	Stos	Wyjście
1	e	e	
2	f	ef	
3	-	(e-f)	
4	a	(e-f)a	
5	b	(e-f)ab	
6	*	(e-f)(a*b)	
7	c	(e-f)(a*b)c	
8	d	(e-f)(a*b)cd	
9	/	(e-f)(a*b)(c/d)	
10	+	(e-f)((a*b)+(c/d))	
11	+	(e-f)+((a*b)+(c/d))	
12	Koniec wyrażenia		(e-f)+((a*b)+(c/d))

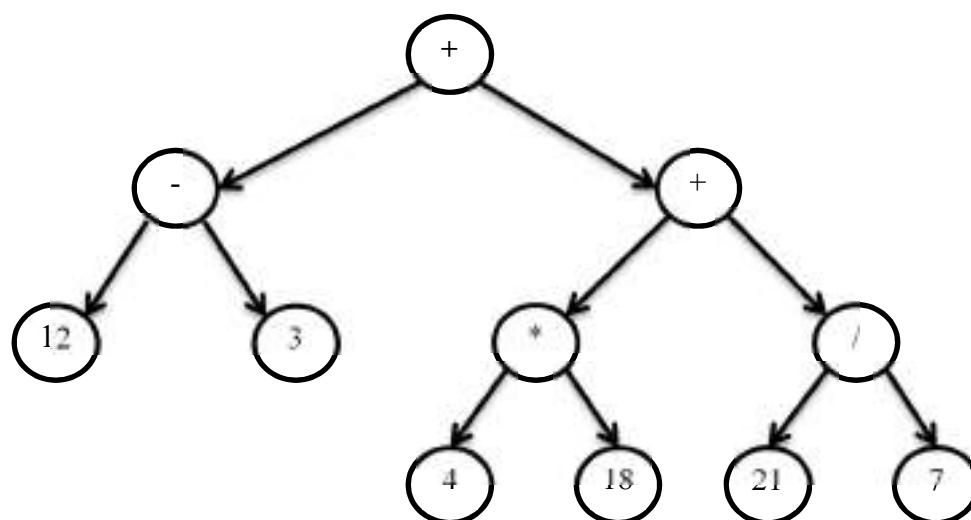
Można zauważyć, iż pomimo, że po ponownej konwersji zostały dołożone nawiasy, kolejność działań nie uległa zmianie.

Obliczanie wartości wyrażeń zapisanych przy pomocy ONP

Powyższy schemat blokowy przedstawia obliczanie wartości wyrażenia zapisanego w ONP. Poniżej znajduje się operacja obliczania wartości przykładowego wyrażenia krok po kroku według tegoż algorytmu.

Wyrażenie: 12 3 - 4 18 * 21 7 / + +

Numer kroku	Wejście	Stos	Wyjście
1	12	12	
2	3	12 3	
3	-	9	
4	4	9 4	
5	18	9 4 18	
6	*	9 72	
7	21	9 72 21	
8	7	9 72 21 7	
9	/	9 72 3	
10	+	9 75	
11	+	84	
12	Koniec wyrażenia		84



Powyższe drzewo obrazuje przykładowe wyrażenia, którego wartość została obliczona.

Wyrażenia regularne

Język regularny

Wśród języków formalnych można wyróżnić klasę języków regularnych. Językiem regularnym nazywamy język, dla którego możemy zbudować deterministyczny automat skończony potrafiący zdecydować, czy dane słowo należy do języka.

Wyrażenia regularne

Wyrażenia regularne to ciągi symboli (znaków konkretnego alfabetu) opisujące pewien język formalny. Wszystkie języki definiowane przez wyrażenia regularne są językami regularnymi.

W praktyce wyrażenia regularne traktuje się jako wzorce utworzone z literałów i metaznaków (znaków o specjalnym znaczeniu), pełniących w wyrażeniu ściśle określone funkcje. Do ich zapisu stosuje się bogatszą i łatwiejszą w użyciu składnię niż tę, stosowaną w rozważaniach teoretycznych.

Wyrażenia regularne wykorzystuje się często przy przeszukiwaniu łańcuchów znakowych i porównywaniu ich z określonym wzorcem oraz przy przekształcaniu ciągów znaków zgodnie z regułami zapisanymi w danym wyrażeniu. Jest to potężne, elastyczne i efektywne narzędzie do manipulowania danymi tekstowymi.

Mechanizmy wyrażeń regularnych są zaimplementowane w większości popularnych języków programowania np.: Perl, Java, JavaScript, Python, Ruby, PHP, C#.

Przykładem wyrażenia regularnego może być wzorzec poprawności zapisu adresu MAC:

```
^([0-9a-fA-F][0-9a-fA-F]){5}([0-9a-fA-F][0-9a-fA-F])$
```

Metaznaki

Wyrażenia regularne składają się z dwóch typów symboli:

- Literałów – zwykłych znaków tekstowych,
- Metaznaków – znaków o specjalnym znaczeniu, interpretowanych przez parser w określony sposób.

Literały we wzorcu reprezentują same siebie. Wzorzec **Modrzew to drzewo** pasuje do części tekstu identycznej z tym wzorcem. Innymi słowy, jeżeli w danej linii znajdzie się fraza **Modrzew to drzewo**, to wzorzec zostanie dopasowany.

Początek i koniec linii

Najbardziej podstawowymi z metaznaków są `^` (kareta/daszek) oraz `$` (znak dolara) oznaczające odpowiednio początek i koniec linii. Wyrażenie regularne `rak` będzie pasowało do czterech liter następujących po sobie `rak` gdziekolwiek w danej linii. Jednak wyrażenie `^rak` będzie pasowało do tej sekwencji znaków wyłącznie na początku linii. Podobnie wyrażenie `rak$` będzie pasowało do sekwencji `rak` tylko na końcu linii. Oprócz oczywistego dopasowania, wyrażenie będzie także dopasowane do ciągów znaków zawierających dane wyrażenie, np.: w ostatnim przypadku wyrazu `wiatrak` znajdującego się na końcu linii.

Najłatwiej interpretować wyrażenia regularne dosłownie: znak po znaku, np. zamiast myśleć:

- `^rak` – pasuje do wyrazu `rak` znajdującego się na początku linii, można zinterpretować wyrażenie w następujący sposób:
- `^rak` – pasuje do linii, która ma początek, potem literę `r`, potem literę `a`, a potem literę `k`.

Obydwie interpretacje znaczą to samo, jednak wersja dosłowna ułatwi analizę nowo poznanych wyrażeń regularnych.

Przykład 1

What do the following regular expressions mean: `^age$`, `^$`, `^`?

Rozwiązanie

- `^rak$`
znaczenie dosłowne: pasuje do linii, która posiada początek (wszystkie linie posiadają początek), po której występują kolejno litery `rak`, po których następuje koniec linii.
znaczenie potoczne: linia zawiera wyłącznie słowo `rak`, bez żadnych innych wyrazów, bez żadnych spacji, bez żadnych znaków interpunkcyjnych; wyłącznie słowo `rak`.
- `^$`
znaczenie dosłowne: pasuje do linii, która posiada początek, po którym następuje koniec linii.
znaczenie potoczne: pusta linia (brak jakichkolwiek znaków).
- `^`
znaczenie dosłowne: pasuje do linii, która posiada początek.
znaczenie potoczne: bezsensowne; wyrażenie pasuje do każdej linii, bez względu na jej zawartość (nawet do linii pustej), gdyż każda linia posiada początek.

Klasy znaków

Klasy znaków oznaczane [...] są konstrukcją umożliwiającą podanie dozwolonych znaków w konkretnym miejscu wyrażenia, które spowodują dopasowanie. Wyrażenie [ay] spowoduje dopasowanie do znaku a lub znaku y.

Analizując wyrażenie m[ay]sz, pasuje ono do litery m, po której występuje litera a lub litera y, po której występuje litera s, po której występuje litera z. Skutkuje to dopasowaniem wyrażenia zarówno do np.: słowa **mysz**, jak i słowa **masz**.

Zakresy

W klasie znaków może wystąpić metaznak klasy znaków¹ - (myślnik). Oznacza on zakres znaków np.: [0-9] jest popularnym oznaczeniem znaku, który jest cyfrą, a [a-z] oznacza małą literę alfabetu łacińskiego. Zakresy można łączyć. Jeśli występuje potrzeba dopasowania do znaku, który jest cyfrą bądź dużą literą, można użyć wyrażenia: [0-9A-Z].

Zauważmy, że w klasie znaków, znak - posiada specjalne znaczenie i oznacza zakres. Występując w wyrażeniu, ale poza klasą oznacza zwykły literał. Ponadto, będąc w klasie, ale na samym jej początku, również oznacza literał (jest to sposób na wprowadzenie myślnika do klasy w celu jego przyszłego dopasowania).

Jeśli chcemy dopasować znak, który może być cyfrą, małą literą, myślnikiem lub znakiem zapytania, możemy skonstruować następujące wyrażenie: [-0-9a-z?]. Pierwszy myślnik interpretowany jest dosłownie, natomiast każdy kolejny oznacza zakres. Występuje tu także znak zapytania, który w wyrażeniach regularnych **poza klasą** jest metaznakiem. W klasie znaków traktowany jest jednak jak literał (podobnie inne metaznaki).

¹ Znak o specjalnym znaczeniu w klasie.

Negacja klasy znaków

Jeśli oznaczymy klasę znaków za pomocą `[^...]` będzie pasować do pojedynczego znaku, który **nie jest** zawarty w klasie, np.: `[^4-7]` pasuje do każdego znaku, który nie jest cyfrą od 4 do 7. Znak `^` neguje listę znaków klasy. Zwróćmy ponownie uwagę, że znak `^` w klasie oznacza coś innego (negację) niż poza nią (początek linii). W wyrażeniu regularnym jest metaznakiem, a w klasie jest metaznakiem klasy znaków.

Przykład 2

Tekst, składający się z listy słów:

```
Mama
komar
Mirosław
komiks
derma
modrzew
imadło
kamera
dom
```

Przeszukiwany jest wyrażeniem `m[^o]`. Poniższa lista przedstawia dopasowane słowa:

```
Mama
komar
komiks
derma
imadło
kamera
```

Trzy słowa występujące na liście: **Mirosław**, **modrzew**, **dom** nie zostały dopasowane. Dlaczego?

Rozwiązanie

- Mirosław** – ciąg nie został dopasowany, gdyż wyrażenie „poszukiwało” małej litery `m`, podczas, gdy **Mirosław** zaczyna się od majuskuły `M`. Do wylistowania tego słowa można użyć wyrażenia `M[^o]` lub `[Mm][^o]`, które dopasowane byłoby zarówno do małej jak i wielkiej litery `m`.
- modrzew** – ciąg nie został dopasowany, gdyż wyrażenie „poszukiwało” małej litery `m`, po której występował znak, niebędący literą `o`. W tym przypadku tak właśnie było.
- dom** – ciąg nie został dopasowany, gdyż wyrażenie „poszukiwało” małej litery `m`, po której występował znak, niebędący literą `o`, co wyklucza dopasowanie do litery `m`, będącej na końcu linii. Zwykle na końcu linii występuje znak końca linii, jednak duża część programów wspierająca wyrażenia regularne usuwa te znaki przed dokonaniem dopasowania, więc po literze `m`, nie było żadnego innego znaku.

Kropka

Metaznak `.` (kropka) jest skrótem dla klasy znaków oznaczającej dowolny znak. Jeśli chcemy w wyrażeniu użyć dopasowania do dowolnego znaku, możemy ją wykorzystać, np. wyrażenie `k.r.` zostanie dopasowane do słów: kora, kura, kort, kara, kurz, kurs, itd.

Przykład 3

Skonstruuj wyrażenie regularne znajdujące datę składającą się z trzech części: dwucyfrowego dnia, dwucyfrowego miesiąca i czterocyfrowego roku. Przykłady zapisu dat: `12/03/2012`, `12-03-2012`, `12.03.2012`.

Rozwiązanie

Zacznijmy od konkretnej daty i zapiszmy wyrażenie w postaci np.: `12/03/2012`

Aby zezwolić na stosowanie wyłącznie znaków `/`, `-` i `.` pomiędzy poszczególnymi częściami, możemy przekształcić wyrażenie do postaci: `12[-/.]03[-/.]2012`. Zauważmy, że wszystkie znaki w naszej klasie są literałami.

Jeśli chcemy być bardziej liberalni i pozwolić na dowolne znaki pomiędzy poszczególnymi częściami możemy przekształcić wyrażenie do postaci: `12.03.2012`, gdzie kropki są już metaznakami i oznaczają dowolny znak. Trzeba pamiętać o tym, że oprócz konkretnej daty można również przypadkowo dopasować np. numer konta `312403 201243 345673`... W zależności od zawartości tekstu, na którym będziemy pracować, powinniśmy wybrać rozwiązanie najbardziej efektywne i najmniej pracochłonne zarazem.

Aby zezwolić na różne daty, wyrażenie może przyjąć ostateczną postać:

```
[0-3][0-9].[0-1][0-9].[12][0-9][0-9][0-9]
```

Zauważmy, że wyrażenie zapisane w ten sposób zostanie dopasowane również np.: do daty `38.18.2999`, dlatego można je jeszcze doprecyzować.

Alternatywa

Metaznak `|` jest przydatny do łączenia pojedynczych podwyrażeń w jedno wyrażenie regularne, które będzie pasowało do każdego z nich. Przykładowo: `kura` i `jajko`, to dwa osobne wyrażenia, ale `kura|jajko`, to jedno wyrażenie, które pasuje zarówno do pierwszego, jak i drugiego słowa.

Nawiązując do podrozdziału „Klasy znaków” wyrażenie `m[ay]sz` jest równoważne wyrażeniu `mysz|masz` albo krócej `m(a|y)sz`.

Różnica pomiędzy klasą znaków a alternatywą polega na tym, że klasa znaków zawsze pasuje do pojedynczego znaku, podczas gdy alternatywa rozdziela poszczególne wyrażenia regularne, które mogą pasować do dowolnej liczby znaków.

Na przykład wyrażenia:

1. `^koza|owca|baran$`
2. `^(koza|owca|baran)$`

wydają się być podobne, jednak znaczą zupełnie co innego. Pierwsze wyrażenie złożone jest z alternatywy trzech wyrażeń: `koza`, `owca`, `baran$`, które nie są zbyt użyteczne. Drugie wyrażenie natomiast pasuje do pojedynczych wyrazów `koza`, `owca` lub `baran` znajdujących się pojedynczo w liniach.

Dosłowna interpretacja to:

- pasuje do linii, która posiada początek,
 - po którym występują znaki `k o z a`,
 - LUB po którym występują znaki `o w c a`,
 - LUB po którym występują znaki `b a r a n`,
- po którym występuje koniec linii.

Nawiasy mają funkcję grupującą jedno wyrażenie, oddzielając je również od innych wyrażeń.

Przykład 4

Usprawniając „Przykład 3” możemy uzupełnić wyrażenie za pomocą alternatyw:

```
(0[1-9]|[12][0-9]|3[01]).(0[1-9]|1[0-2]).[12][0-9][0-9][0-9]
```

Kwantyfikatory

Kwantyfikatory określają liczbę wystąpień poprzedzającego je bezpośrednio podwyrażenia. Jednym z nich jest `?` (znak zapytania) i oznacza wystąpienie opcjonalne. Podwyrażenie, po którym występuje `?` może wystąpić maksymalnie jeden raz (czyli zero lub jeden raz), np.: wyrażenie `s?er`, pozwala na wystąpienie lub brak litery `t`. Oznacza to, że będzie pasowało zarówno do słowa `ster` jak i `ser`.

Kolejnym kwantyfikatorem jest `+` (plus). Metaznak oznacza jedno lub więcej wystąpień bezpośrednio poprzedzającego go wyrażenia. Wyrażenie `go+1` będzie pasowało do słowa `gol`, jak i `goool`, ale też `goo1`. Nie będzie natomiast pasować do słowa `gl`, gdyż `o` musi wystąpić przynajmniej jeden raz.

Istnieje także kwantyfikator `*` (gwiazdka) oznaczająca dowolną liczbę wystąpień wyrażenia bezpośrednio go poprzedzającego. Innymi słowy, takie wyrażenie może wystąpić zero lub więcej razy. Wyrażenie `go*1` pozwoliłoby również na dopasowanie słowa `gl` z poprzedniego przykładu.

Ponadto można określić przedział liczby wystąpień za pomocą kwantyfikatora `{min,max}`. Określa on ile razy minimalnie i maksymalnie może pojawić się wyrażenie bezpośrednio go poprzedzające, np.: `{5,23}` będzie dopasowane, jeśli liczba wystąpień danego wyrażenia nie przekroczy 23 a jednocześnie nie będzie mniejsza niż 5. Innym przykładem może być wyrażenie `[4-7]{6}` oznaczające wystąpienie dokładnie sześciu cyfr od 4 do 7. Jeśli znamy wyłącznie minimalną liczbę wystąpień możemy użyć wzorca: `[a-zA-Z]{4,}`, który oznacza co najmniej 4 litery.

Przykład 5

Za pomocą kwantyfikatorów „Przykład 4” może zostać usprawniony jeszcze bardziej:

```
(0[1-9]|[12][0-9]|3[01]).(0[1-9]|1[0-2]).[12][0-9]{3}
```

Znaki ucieczki

Jeżeli chcemy, aby jakikolwiek metaznak został zinterpretowany dosłownie, musimy użyć znaku ucieczki `\`. Pozwala on na zmianę interpretacji z metaznaków na literały.

Jeżeli przykładowo szukamy strony internetowej o adresie `licze.nie.com` i użyjemy wyrażenia regularnego `licze.ie.com`, możemy otrzymać dopasowanie typu: `rozliczenie.comiesięczne`. Kropka oznacza dowolny znak (w tym spację).

Aby kropki były interpretowane dosłownie (jako kropki w adresie www) trzeba poprzedzić je znakami ucieczki. Wtedy tracą one swoje znaczenie i stają się literałami: `licze\.ie\.com`

Podobnie, jeśli chcemy użyć nawiasów `()` w sposób dosłowny, musimy poprzedzić je znakami ucieczki. Jeżeli wyrażenie ma być dopasowane do słowa w nawiasach np.: `(hipopotam)`, to w wyrażeniu należy użyć znaków ucieczki: `\([a-zA-Z]\)`.

Zestawienie poznanych metaznaków

Metaznak	Opis
Dopasowanie do pojedynczych znaków	
<code>.</code>	dowolny znak
<code>[...]</code>	dowolny znak zawarty w klasie
<code>[^...]</code>	dowolny znak, oprócz znaków zawartych w klasie
<code>\znak</code>	znak ucieczki
Kwantyfikatory	
<code>?</code>	wzorzec powtórzony 1 raz lub wcale
<code>*</code>	wzorzec powtórzony dowolną liczbę razy (z zerem włącznie)
<code>+</code>	wzorzec powtórzony minimum 1 raz
<code>{min,max}</code>	wzorzec powtórzony co najmniej <i>min</i> i nie więcej niż <i>max</i> razy
<code>{min,}</code>	wzorzec powtórzony co najmniej <i>min</i> razy
<code>{x}</code>	wzorzec powtórzony dokładnie <i>x</i> razy

Metaznak	Opis
Metaznaki określające pozycję	
<code>^</code>	początek linii
<code>\$</code>	koniec linii
Inne	
<code> </code>	alternatywa; pasuje do każdego z podwyrażeń, które rozdziela
<code>(...)</code>	ograniczają zakres alternatywy oraz pełnią funkcję grupującą dla kwantyfikatorów

Przykład 6

Wyrażenie pasujące do wyrazów w cudzysłowach np.: „kot”

```
"[\"^\"]*"
```

Cudzysłowy na początku i końcu wyrażenia służą do oznaczenia cudzysłowów, w których znajduje się wyraz. Pomiedzy nimi może znaleźć się wszystko oprócz cudzysłowu.

Wyrażenie pasujące do ceny w złotych z opcjonalnymi groszami np.: 25.98zł lub 31zł.

```
[0-9]+(\\.[0-9]{2})?zł
```

Pierwsza klasa określa nam dowolną cyfrę, która musi wystąpić co najmniej jeden raz. Następne jest opcjonalne podwyrażenie (nawiasy i znak zapytania), w którym powinna znaleźć się kropka oraz dokładnie dwie cyfry. Koniec wyrażenia to znaki **z** i **ł**.

Wyrażenie pasujące do godziny wraz z minutami np.: 23:46 lub 7:02

```
([0-9]|1[0-9]|2[0-3]):[0-5][0-9]
```

Pierwszy nawias grupuje trzy możliwości: godzinę od 0 do 9, godzinę od 10 do 19 oraz godzinę od 20 do 23. Następnie występuje dwukropek, po czym muszą wystąpić dwie cyfry. Pierwsza z nich z przedziału od 0 do 5, druga od 0 do 9.

Przeszukiwanie plików tekstowych

Wyrażenia regularne są także obsługiwane w prostszych programach narzędziowych wywoływanych z linii komend np.: `grep`, `sed`, `awk`.

`grep` jest akronimem „*Global Regular Expressions Print*”. Program odczytuje dane wejściowe linia po linii i zwraca linie pasujące do zadanego wzorca (wyrażenia regularnego). Program dostępny jest dla różnych systemów operacyjnych, m.in. Windows, MacOS, Unix.

Przykład użycia:

```
$ grep -E '^cat' file
```

`grep` z przełącznikiem `-E` oznacza, że argument, następujący po przełączniku, traktowany jest jako rozszerzone wyrażenie regularne, a kolejne argumenty jako pliki, w których `grep` ma dokonać wyszukiwania. Pojedyncze cudzysłowy nie są częścią wyrażenia regularnego, lecz znakami wymaganymi przez powłokę.

Przykład 7

Chcemy przeanalizować korespondencję mailową ze znaną nam osobą. Jej adres email to: *lama@zoo.org*. Wszystkie wiadomości znajdują się w skrzynce pocztowej i przechowywane są w formie tekstowej w jednym pliku o nazwie *mailbox*. Należy wyświetlić jedynie linijki zawierające nadawcę oraz temat wiadomości.

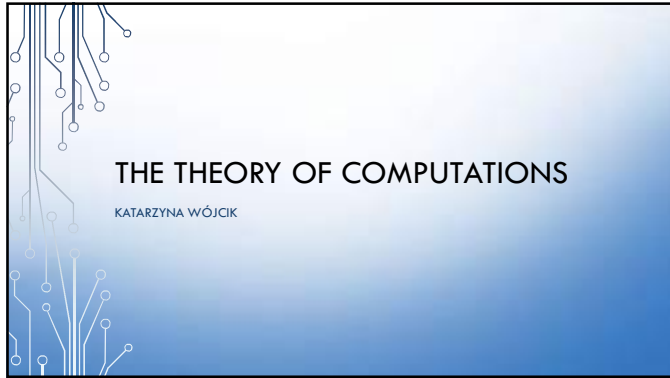
Poniżej znajduje się pojedyncza przykładowa wiadomość email pochodząca z pliku *mailbox*:

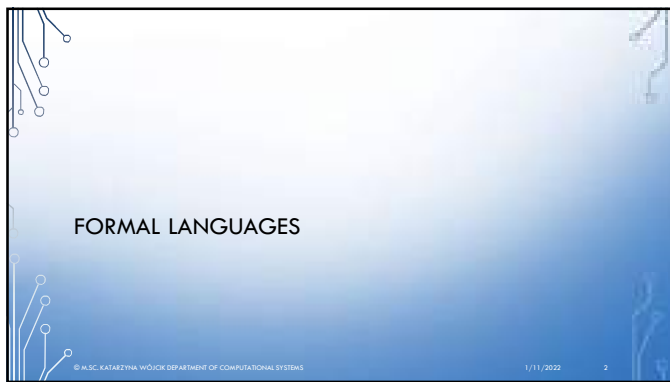
```
Received: from lama@localhost by zoo.org (6.13.1) id JD9EBU
Received: from zoo.org by garden.net (8.12.5/2) id G73HS
To: zebra@garden.info (Zebra Pasiasta)
From: lama@zoo.org (Lama Futrzasta)
Date: Wed, Sep 14 2012 15:38
Message-Id: <2012091456832.JD9EBU@zoo.org>
Subject: Zaproszenie na kawę
Reply-To: lama@zoo.org
X-Mailer: Lamas Mailer [version 4.5]
Czesc Zebro!
Serdecznie zapraszam na kawę do mojego wybiegu. Tak dawno sie nie
widzialysmy, ze postanowiłam zorganizowac male spotkanie. Odbędzie się
ono w najbliższa sobotę o godzinie 16:00. Dołącza do nas także
Krokodyl, Jaszczurka i Zyrafa.
Trzymaj sie ciepło,
Lama
```

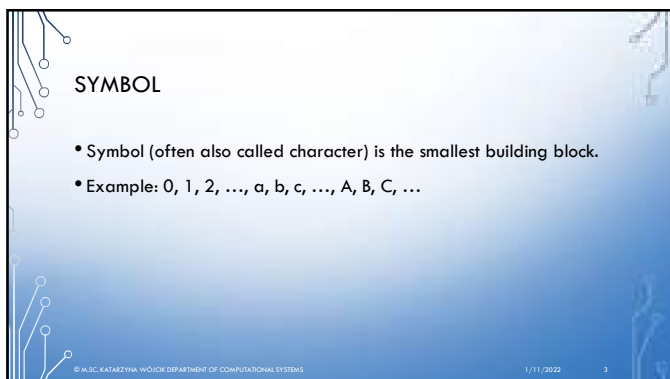
Rozwiązanie

Należy użyć komendy w postaci:

```
$ grep -E '^(From|Subject):( llama@zoo.org)?' mailbox
```







ALPHABET

- Alphabet is a finite, nonempty set of symbols.
- Very often symbol Σ is used for alphabet representation.
- Examples:
 - $\Sigma = \{0, 1\}$, the binary alphabet
 - $\Sigma = \{a, b, \dots, z\}$, the set of all lower-case letters

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS

1/11/2022

4

STRING

- String is a finite sequence of symbols chosen from an alphabet.
- Symbol ϵ represents empty string. Empty string may be chosen from any alphabet.
- Example:
 - Alphabet $\Sigma = \{0, 1\}$
 - Strings from Σ alphabet: 1100110, 111001, 1, 0, 10

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS

1/11/2022

5

STRING

- Length of a string is a number of position for symbols in this string. The length of the string s is denoted $|s|$.
- Examples:
 - $|011| = 3$
 - $|\epsilon| = 0$

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS

1/11/2022

6

STRINGS

- Σ^k is a set of all strings of length k from the alphabet Σ .

- Example:

- $\Sigma = \{0, 1\}$ alphabet
- $\Sigma^1 = \{0, 1\}$ set of strings of length 1
- $\Sigma^2 = \{00, 01, 10, 11\}$ set of strings of length 2
- $\Sigma^3 = \{000, 001, 010, 100, 011, 101, 110, 111\}$...
- $\Sigma^0 = \{\epsilon\}$...

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS

1/11/2022

7

STRINGS

- Σ^* is a set of all strings over an alphabet Σ .

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots \cup \Sigma^n$$

- Σ^+ is a set of all nonempty strings over an alphabet Σ .

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \dots \cup \Sigma^n$$

$$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$$

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS

1/11/2022

8

SET-BUILDER NOTATION

- Set-builder notation is a mathematical notation for describing a set by enumerating its elements ($\{0, 1, 2, \dots\}$) or stating the properties that its members must satisfy ($\{\text{natural even numbers}\}$)

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS

1/11/2022

9

SET-BUILDER NOTATION

- Set-builder notation is often used with a predicate characterizing the elements of the set being defined ($\{n \in \mathbb{N} \mid (\exists k)[k \in \mathbb{N} \wedge n=2k]\}$ – set of all natural even numbers).
- The vertical bar (\mid) is a separator that can be read as "such that", "for which", or "with the property that".

© MISC. KATARZYŃA WÓJCIE DEPARTMENT OF COMPUTATIONAL SYSTEMS

1/11/2022

10

FORMAL LANGUAGE

- Formal language L is a set of strings all of which are chosen from some Σ^* .
- We can say that a language is a subset of Σ^* .
- If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a language over Σ .

© MISC. KATARZYŃA WÓJCIE DEPARTMENT OF COMPUTATIONAL SYSTEMS

1/11/2022

11

FORMAL GRAMMAR

- A formalism for formal language definition.
- A formal grammar can be defined as $G = (N, \Sigma, P, S)$, where:
 - N is a set of nonterminal symbols (temporary),
 - Σ is a set of terminal symbols (alphabet),
 - P is a set of production (transforming) rules,
 - S is a start symbol.

© MISC. KATARZYŃA WÓJCIE DEPARTMENT OF COMPUTATIONAL SYSTEMS

1/11/2022

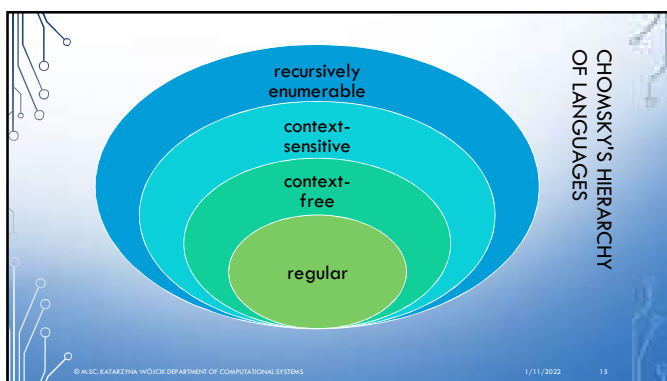
12

• A grammar is a set of production rules which are used to generate strings of a language.

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS 1/11/2022 13

CLASSIFICATION OF FORMAL LANGUAGES

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS 1/11/2022 14



Grammar	Languages	Automaton	Production rules
Type-0	Recursively enumerable	Turing machine	no constraints
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \alpha$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ $A \rightarrow aB$

Symbol	Meaning
a	Terminal symbol
A, B	Non-terminal symbols
$\alpha \beta$	Strings of terminal and/or non-terminal symbols
γ	Not empty string of terminal and/or non-terminal symbols

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS 1/11/2022 16

AUTOMATA THEORY

- Automata theory (also known as Theory Of Computation) is a theoretical branch of Computer Science and Mathematics, which mainly deals with the logic of computation with respect to simple machines, referred to as automata.
- Automata enables scientists to understand how machines compute the functions and solve problems. The main motivation behind developing Automata Theory was to develop methods to describe and analyze the dynamic behavior of discrete systems.


© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS 1/11/2022 17

FINITE AUTOMATA

- Finite Automata (FA) is the simplest machine to recognize patterns. The finite automata or finite state machine is an abstract machine that has five elements or tuples. It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Basically, it is an abstract model of a digital computer.

© MISC. KATARZYNA WOJCIK DEPARTMENT OF COMPUTATIONAL SYSTEMS 1/11/2022 18

FINITE AUTOMATA



The diagram illustrates a finite automaton with three main components: an input signal block labeled x_1, x_2, \dots, x_n , a state transition block labeled Q_1, Q_2, \dots, Q_n , and an output signal block labeled y_1, y_2, \dots, y_n . Arrows indicate the flow of information from input to state and from state to output.

- Input signals
- Output signals
- States of automata
- State relations
- Output relation

© M.SC. KATARZYŃA WOJCIECH DEPARTMENT OF COMPUTATIONAL SYSTEMS 1/11/2022 19

SOURCES

- <https://www.geeksforgeeks.org/theory-of-computation-automata-tutorials/>
- https://en.wikipedia.org/wiki/Formal_grammar
- Lectures of prof. Paweł Lula

© M.SC. KATARZYŃA WOJCIECH DEPARTMENT OF COMPUTATIONAL SYSTEMS 1/11/2022 20

Algorytmy

Etapy tworzenia oprogramowania

	Opis etapu	Kto realizuje
1	Dostrzeżenie problemu i możliwości jego rozwiązania za pomocą komputera	Człowiek
2	Zaprojektowanie sposobu rozwiązania problemu	Człowiek
3	Zapis sposobu rozwiązania problemu w sposób zrozumiały dla człowieka	Człowiek
4	Przekształcenie opisu sposobu rozwiązania problemu do postaci zrozumiałej dla komputera	Komputer

Definicja algorytmu

Algorytm – określony sposób postępowania prowadzący do rozwiązania postawionego problemu; zestaw instrukcji. Algorytm może być realizowany przez człowieka lub przez komputer.

Własności algorytmów

- **Dyskretność** – algorytm składa się z elementarnych kroków niezbędnych do rozwiązania problemu. Czynności bardziej złożone powinny być rozbite na oddzielne algorytmy, złożone wyłącznie z elementarnych kroków.
- **Uniwersalność** – algorytm powinien rozwiązywać pewną klasę problemów z danej dziedziny, a nie tylko jeden szczególny przypadek.
- **Jednoznaczność** – dla takich samych danych wejściowych algorytm zawsze zwróci te same dane wyjściowe.
- **Kompletność** – algorytm musi uwzględniać wszystkie możliwe przypadki, które pojawiają się podczas jego realizacji.
- **Skończoność (finistyczność)** – algorytm powinien gwarantować osiągnięcie rozwiązania w skończonej liczbie kroków, a więc także w skończonym czasie. Niedopuszczalne jest, aby algorytm wykonywał się w nieskończoność. Warunki zakończenia lub przerywania pracy algorytmu powinny być tak sformułowane, aby w przypadku braku rozwiązania lub gdy jest ono nieosiągalne w dostępnym czasie, algorytm kończył swoje działanie.

Paradygmaty tworzenia algorytmów

- **Dziel i zwyciężaj** – dzielimy problem na kilka mniejszych, a te dzielimy ponownie, aż ich rozwiązania staną się oczywiste.
- **Programowanie dynamiczne** – problem dzielimy na kilka mniejszych. Ważność każdego z nich jest oceniana i po pewnym wnioskowaniu wyniki analizy niektórych prostszych zagadnień wykorzystuje się do rozwiązania głównego problemu.
- **Metoda zachłanna** – nie analizujemy podproblemów dokładnie, tylko wybieramy najbardziej obiecującą w danym momencie drogę rozwiązania.
- **Programowanie liniowe** – oceniamy rozwiązanie problemu za pomocą pewnej funkcji celu. Wprowadzamy warunki ograniczające i szukamy jej ekstremum.
- **Poszukiwanie i wyliczanie (metoda brutalnej siły)** – nie skupiamy się na analizie problemu, lecz przeszukujemy zbiór wszystkich możliwych rozwiązań aż do odnalezienia rozwiązania poprawnego.
- **Heurystyki** – człowiek na podstawie swojego doświadczenia tworzy algorytm, który działa w najbardziej prawdopodobnych warunkach. Rozwiązanie zawsze jest przybliżone. Heurystyk używa się, gdy realizacja pełnego algorytmu jest z przyczyn technicznych zbyt kosztowna lub gdy pełny algorytm jest nieznany (np. przy przewidywaniu pogody lub przy wykrywaniu niektórych zagrożeń komputerowych, takich jak wirusy lub robaki).

Sposoby zapisu algorytmów

1. Zapis w języku naturalnym
2. Zapis za pomocą notacji matematycznej
3. Zapis za pomocą schematów blokowych
4. Zapis za pomocą języków programowania
5. Zapis za pomocą pseudokodu

Złożoność obliczeniowa algorytmów

Złożoność obliczeniowa algorytmu to ilość zasobów komputerowych potrzebnych do jego realizacji.


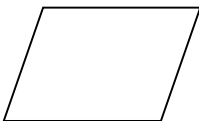

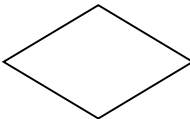
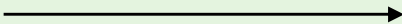
Typy złożoności

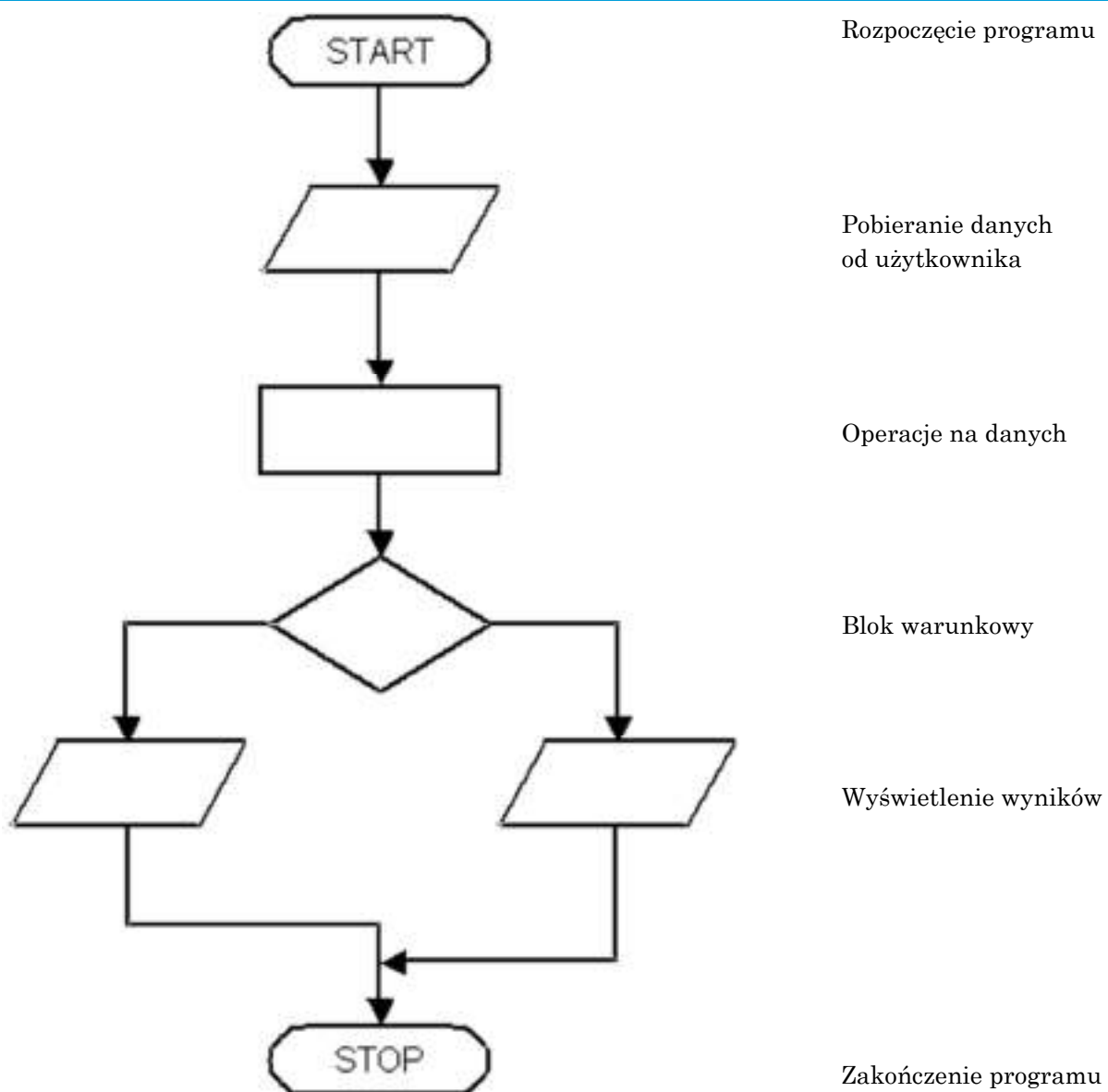
- **Czasowa** (czas pracy procesora).
- **Pamięciowa** (pamięć operacyjna).

Schemat blokowy

Schemat blokowy jest graficznym sposobem przedstawienia algorytmu. Ułatwia on przemyślenie rozwiązania problemu, którego dotyczy sam algorytm.

Elementy schematu blokowego

Symbol	Opis
	Blok graniczny, początek lub koniec algorytmu
	Blok wprowadzenia i wyprowadzania danych
	Blok obliczeniowy
	Blok decyzyjny, warunkowy
	Droga przepływu danych

Pusty schemat blokowy

Rekurencja

Definicja

Rekurencja (inaczej rekursja - ang. recursion) oznacza odwoływanie się funkcji lub definicji do samej siebie

Cechy algorytmów rekurencyjnych

- Algorytm wywołuje sam siebie.
- Zakończenie algorytmu jest jasno określone.
- Złożony problem zostaje rozłożony na problemy elementarne o mniejszym stopniu skomplikowania.

Przykład 1

Dana jest tablica n liczb całkowitych. Sprawdź, czy w tej tablicy znajduje się liczba całkowita x .

Dane:

- Tablica n liczb całkowitych; indeksy numerowane od zera ($tab[0], tab[1], \dots, tab[n-1]$).
- Liczba x .

Problem:

- Czy w tablicy występuje liczba x ?

Rozwiązanie rekurencyjne (zapis w języku naturalnym):

1. Jeśli została przebadana cała tablica, ogłoś niepowodzenie i zakończ.
2. Weź pierwszy niezbadany element z tablicy n -elementowej.
3. Jeśli aktualnie analizowany element jest równy x , to ogłoś sukces i zakończ przeszukiwanie.
4. W przeciwnym przypadku zbadaj pozostałą część tablicy.

Kod:

- tab – tablica elementów,
- $left$ – indeks elementu leżącego „najbardziej na lewo” w tablicy – lewa granica obszaru poszukiwań,
- $right$ – indeks elementu leżącego „najbardziej na prawo” w tablicy – prawa granica obszaru poszukiwań,
- x – wartość poszukiwanego elementu.

```
szukaj(tab, left, right, x) {  
  if (left > right)  
    wypisz „Element nie został znaleziony”  
  else  
    if (tab[left] == x)  
      wypisz „Element został znaleziony”  
    else  
      szukaj(tab, left+1, right, x)  
}
```

Czy powyższy przykład posiada cechy algorytmu rekurencyjnego?

- Funkcja wywołuje samą siebie.
- Warunek zakończenia jest jasno określony:
 - element zostaje odnaleziony,
 - następuje przekroczenie zakresu tablicy.
- Duży problem zostaje podzielony na problemy elementarne:
 - z tablicy o rozmiarze n schodzimy do tablicy o rozmiarze $n-1$.

Przykład 2

Oblicz silnię liczby całkowitej n .

Data:

- Liczba n .

Problem:

- Obliczenie silni dowolnej liczby całkowitej n .

Rozwiązanie rekurencyjne (zapis matematyczny):

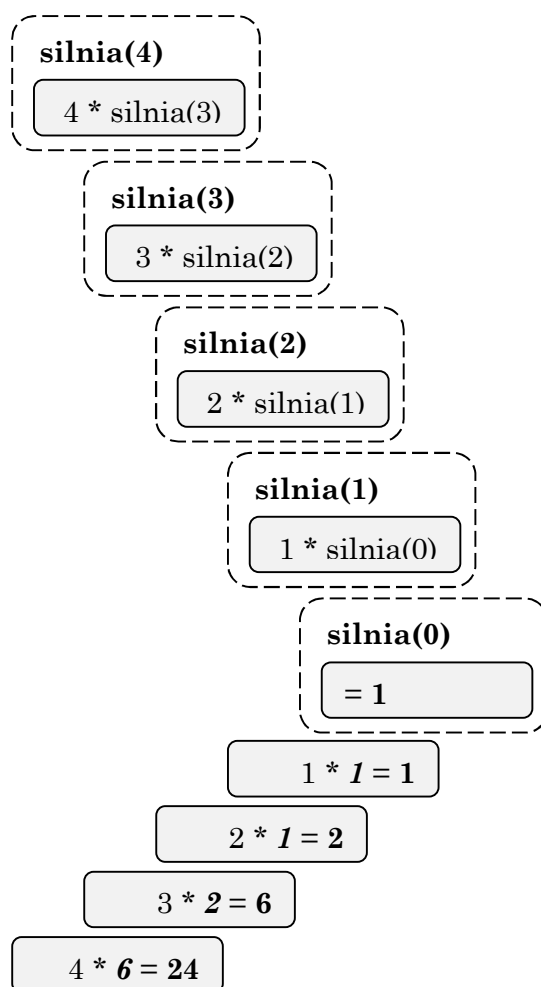
- $$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \end{cases} \text{ gdzie } n \in \mathbb{N}, n \geq 1$$

Kod:

```
silnia(n) {  
  if (n == 0)  
    zwróć 1  
  else  
    zwróć n*silnia(n-1)  
}
```

Czy powyższy przykład posiada cechy algorytmu rekurencyjnego?

- Funkcja wywołuje samą siebie.
- Warunek zakończenia jest jasno określony:
 - dla elementu równego zero, algorytm zwraca liczbę 1
- Duży problem zostaje podzielony na problemy elementarne:
 - przy kolejnych wywołaniach rekurencyjnych liczba n jest dekrementowana o 1.

Przykład wywołania funkcji silnia:**Problemy rozwiązań rekurencyjnych**

1. Pamięciożerność (stack overflow).
2. Nieskończona liczba wywołań funkcji.
3. Część obliczeń może być wykonywana więcej niż jeden raz.
4. Zbieżność procesu rekurencyjnego (problem kompilatora).

3. Materiały.

Temat 1: Proces wytwarzania oprogramowania

Oprogramowanie

Oprogramowanie jest zbiorem instrukcji kontrolujących zachowanie programowalnej maszyny obliczeniowej. Oprogramowanie jest niematerialne i nie podlega żadnym prawom fizycznym (np. prawom natury). Zwykle jest wykonywane od podstaw na zamówienie konkretnego klienta (w przeciwieństwie do sprzętu, który często jest składany z gotowych modułów). W konsekwencji proces wytwarzania oprogramowania jest jednym z najtrudniejszych procesów produkcji dóbr.

W latach 1960 na całym świecie pojawiało się wiele problemów związanych z wytwarzaniem oprogramowania:

- przekraczano budżety i terminy,
- końcowy produkt był nieefektywny,
- oprogramowanie było niskiej jakości,
- oprogramowanie nie spełniało wymagań funkcjonalnych,
- kod źródłowy był trudny w utrzymaniu,
- praca w zespołach nie była odpowiednio zorganizowana (nadgodziny).

Skala problemu była tak duża, że fenomen ten został nazwany **kryzysem oprogramowania** (termin pierwszy raz pojawił się na konferencji NATO Software Engineering Conference w Niemczech w 1960r.)

Edsger Dijkstra na temat kryzysu oprogramowania:

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

— Edsger Dijkstra, 1972

Jako remedium na problem chaotycznego procesu wytwarzania oprogramowania zaproponowano kilka nowych rozwiązań. Usystematyzowane podejścia do rozwiązania pewnych zagadanie nazywamy **metodykami**. Metodyka to zbiór zasad dotyczących sposobów wykonywania jakiejś pracy lub trybu postępowania prowadzącego do określonego celu.

Metodyka wytwarzania oprogramowania to zbiór reguł używanych do planowania, organizowania, prowadzenia i kontrolowania procesu wytwarzania systemu informacyjnego – narzuca pewną dyscyplinę i pomaga utrzymać porządek w prowadzonych pracach.

- Przed kryzysem oprogramowania wykorzystywano metodę *ad-hoc*, która w rzeczywistości w ogóle nie była metodyką. Proces tworzenia oprogramowania odbywał się bez planowania oraz dokumentacji.
- Po okresie kryzysu skonstruowano i zaproponowano wiele metodyk porządkujących proces wytwórczy oprogramowania. Każda z nich posiadała zarówno zalety jak i wady. Metodyki użyteczne w jednych projektach, mogły w ogóle nie pasować do innych projektów.

- Każda z dostępnych metodyk jest dopasowana do pewnych typów projektów w zależności od wielu czynników technicznych i organizacyjnych.

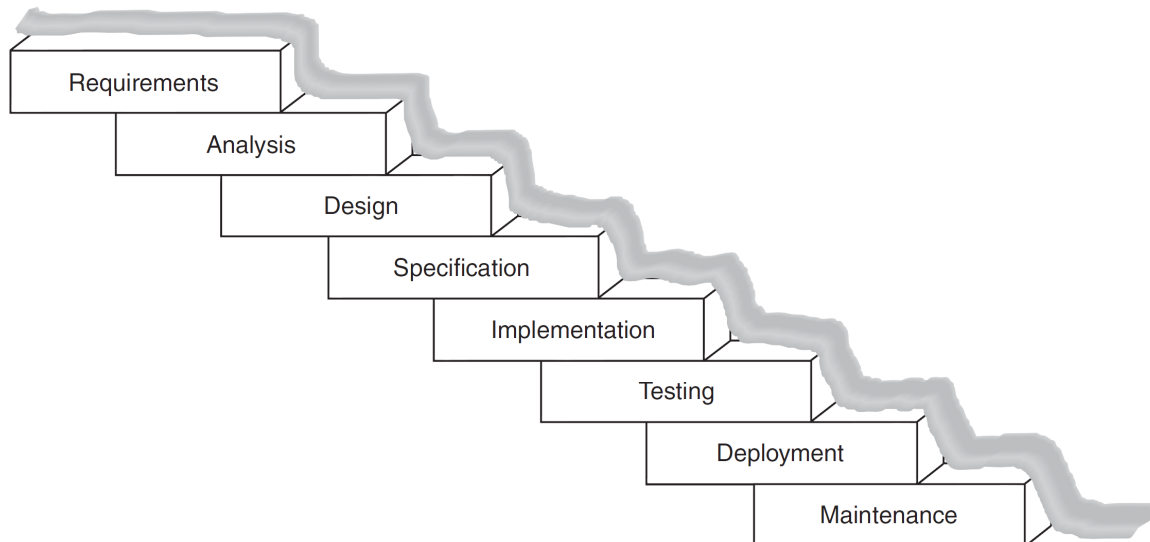
Klasyczne fazy wytwarzania oprogramowania

- Planowanie (Requirements)
 - Definiowanie wymagań funkcjonalnych.
 - Definiowanie potrzeb użytkownika końcowego.
 - Wynikiem modelowania wymagań jest specyfikacja wymagań systemu.
- Analiza (Analysis)
 - Analiza zajmuje się badaniem obszaru funkcjonowania organizacji/instytucji/przedsiębiorstwa, dla którego ma powstać nowe oprogramowanie.
 - Analiza obejmuje:
 - podmioty, ich właściwości i związki między nimi,
 - procesy zachodzące w organizacji.
- Projektowanie (Design)
 - Projektowanie jest procesem definiowania architektury, komponentów, modułów, interfejsów i danych systemu w oparciu o specyfikację wymagań.
 - Wynikiem fazy projektowania jest specyfikacja systemu.
 - Specyfikacja systemu powinna istnieć w formie pisemnej. To ona stanowi podstawę do późniejszego pisania kodu.
- Implementacja (Implementation)
 - Implementacja polega na przełożeniu specyfikacji wymagań na kod źródłowy oprogramowania. Zadanie to jest realizowane przez programistów.
- Testowanie (Testing)
 - Testowanie to sprawdzenie, czy wytworzone oprogramowanie:
 - spełnia wymagania określone w specyfikacji systemu,
 - działa prawidłowo i nie posiada błędów.
- Wdrożenie (Deployment)
 - Wdrożenie to wszystkie działania składające się na dostarczeniu, udostępnieniu i umożliwieniu użytkownikom końcowym korzystania z systemu (włącznie ze sprzętem, dokumentacją i szkoleniami).
- Pielęgnacja (Maintenance)
 - Pielęgnacja lub utrzymanie jest poprawianiem systemu po jego dostarczeniu w celu eliminacji występujących błędów lub zmiany parametrów systemu (np.: zwiększenia szybkości jego działania).

Klasyczne modele wytwarzania oprogramowania

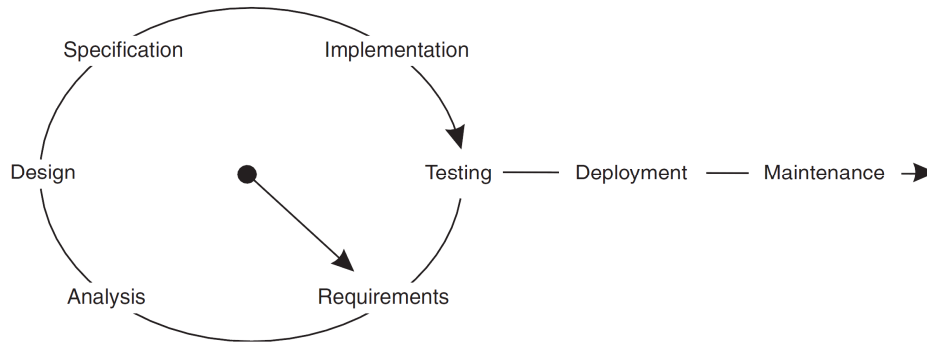
- Model kaskadowy (Waterfall),
- Model spiralny (Spiral),
- Model iteracyjny (Iterative),
- Model przyrostowy (Incremental).

Model kaskadowy



- Model kaskadowy jest sekwencyjnym procesem tworzenia oprogramowania, w którym progres postrzegany jest, jako przechodzenie przez kolejne klasyczne fazy tworzenia oprogramowania.
- Pierwszym formalnym opisem tego modelu był artykuł z 1970r. napisany przez Winstona W. Royce.
- Model kaskadowy dzieli proces wytwórczy na części, z których każda stanowi odrębną czynność.
- Przed rozpoczęciem kolejnego etap, etap poprzedni musi być zakończony.
- Planowanie i kontrolowanie jest stosunkowo łatwe.
- Stosowanie klasycznego modelu kaskadowego jest nierealistyczne!
 - Korygowanie błędnych decyzji podjętych we wcześniejszych etapach nie jest możliwe.
 - Osoby zaangażowane w projekt boją się przechodzić do kolejnych etapów z uwagi na ryzyko podejmowania nieodwracalnych błędnych decyzji.

Model spiralny

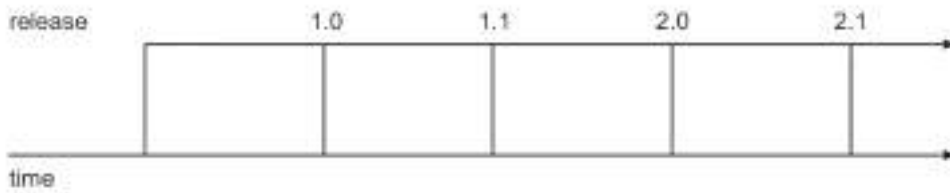


- W odróżnieniu od modelu kaskadowego, w modelu spiralnym powtarzane są następujące fazy wytwórcze:
 - planowanie,
 - analiza,
 - projektowanie,
 - implementacja,
 - testowanie.
- Każdy z etapów nie być zakończony przed przejściem do kolejnego. W trakcie trwania cyklu zdobywana jest nowa wiedza, a problem staje bardziej zrozumiały. Po powrocie do tego samego etapu przy kolejnej iteracji, prawdopodobieństwo poprawnego rozwiązania problemu rośnie.
- Po kilku cyklach (3-4) następują fazy:
 - wdrożenia,
 - pielęgnacji.
- Pierwsza iteracja koncentruje się na celach głównych. Szczegóły są brane pod uwagę w kolejnych iteracjach.
- Wynikiem każdej pełnej iteracji jest działający prototyp tworzonego systemu.
- Błędne decyzje mogą zostać poprawione w kolejnych iteracjach.
- Model nadaje się do prowadzenia dużych i skomplikowanych projektów.
- Zastosowanie modelu spiralnego daje większą elastyczność, ale jest droższe niż zastosowanie modelu kaskadowego.

Model iteracyjny

- Model iteracyjny pozwala na przechodzenie pomiędzy fazami do przodu i do tyłu, a także zapętlanie fazy, jeśli jest taka potrzeba.
- W razie potrzeby można powrócić do dowolnej fazy procesu.
- Model iteracyjny może być postrzegany, jako ulepszona wersja modelu kaskadowego i spiralnego.
- Wykorzystanie klasycznych faz procesu tworzenia oprogramowania zapobiega powstaniu chaosu.
- Efekty poszczególnych faz (diagramy, dokumentacje, kod, etc.) są stopniowo ulepszone.

Model przyrostowy



- Model przyrostowy dzieli cały proces na części realizujące poszczególne funkcjonalności systemu.
- Model próbuje dostarczyć działający system "kawałek po kawałku".
- Rezultatem pierwszej wersji powinien być działający system z podstawową funkcjonalnością.
- Następnie dodatkowa funkcjonalność jest dodawana wraz z nowymi wersjami systemu.
- Przykładowy proces w modelu przyrostowym wygląda następująco:

Planowanie → Analiza → Projektowanie → Implementacja → Testowanie → Wdrożenie
→ Pielęgnacja => wersja 1.0 (podstawowa funkcjonalność)

Planowanie → Analiza → Projektowanie → Implementacja → Testowanie → Wdrożenie
→ Pielęgnacja => wersja 2.0 (nowa funkcjonalność, zmiany)

Planowanie → Analiza → Projektowanie → Implementacja → Testowanie → Wdrożenie
→ Pielęgnacja => wersja 3.0 (nowa funkcjonalność, zmiany)

Planowanie → Analiza → Projektowanie → Implementacja → Testowanie → Wdrożenie
→ Pielęgnacja => wersja 4.0 (nowa funkcjonalność, zmiany)

....

- Model bardzo łatwo przystosowuje się do rzeczywistych i zmiennych w czasie wymagań.
- Model zakłada częste dokonywanie zmian idei działania istniejącego kodu w celu usprawnienia działania systemu (refactoring), co jest dużo lepszym rozwiązaniem niż "łatanie" słabo napisanego kodu.
- Model nadaje się do projektów o dużym stopniu skomplikowania.
- Podstawowa funkcjonalność dostarczana jest bardzo szybko.
- Wadą jest wysoki koszt i długi czas wytworzenia oprogramowania.

Metodyki hybrydowe

Wszystkie opisane metodyki posiadają zalety, jednak bardzo często, efektywność każdej z nich osobno, nie jest wysoka. Aby rozwiązać ten problem zaproponowano metodyki hybrydowe, łączące cechy kilku metodyk podstawowych. Przykłady kilku z nich to:

- RUP – Rational Unified Process
 - nacisk na szczegółową dokumentację,
 - zarządzanie ryzykiem,
 - dalekie planowanie,
 - ciągła analiza jakości systemu,
 - wymaga ekspertów,
 - złożona.
- Agile methodologies (metodyki zwinne)
 - zorientowana na ludzi,
 - adaptacyjna,
 - ciągły współpraca z klientem,
 - wykorzystuje krótkie, ograniczone w czasie iteracje (miesięczne lub krótsze),
 - mały nacisk na projektowanie i dokumentację,
 - podatna na złe ukierunkowanie procesu w przypadku niejasnych informacji przekazywanych przez klienta.
- RAD – Rapid Application Development
 - klient aktywnie uczestniczy w procesie tworzenia systemu,
 - szybka,
 - niskie koszty stosowania,
 - mało użyteczna przy skomplikowanych projektach,
 - bardzo uzależniona od umiejętności technicznych programistów.

Temat 2: Ewolucja metod programowania

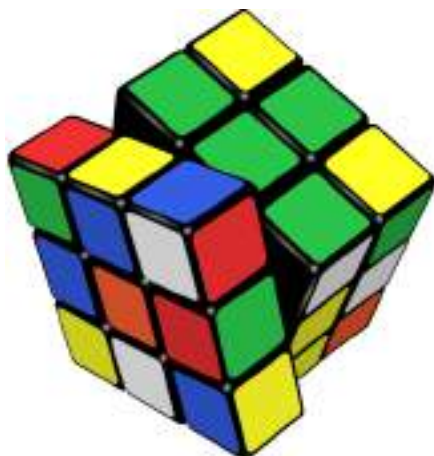
Jak piszemy programy komputerowe?

- obliczenia,
- przetwarzanie danych,
- strony internetowe,
- rozrywka,
- tworzenie dokumentów,
- przetwarzanie obrazów,
- ...



**ROZWIĄZYWANIE
PROBLEMÓW**

Problem



- Problem nie jest często związany z dziedziną informatyki.
- Terminologia z dziedziny problemu.
- Problem niezrozumiały dla informatyków.

duża przepaść

Rozwiązanie problemu za pomocą komputera



- Instrukcje procesora
 - odczyt/zapis do komórek pamięci
 - przesuwanie bloków danych,
 - transfer danych pomiędzy rejestrami,
 - operacje logiczne i arytmetyczne na wartościach binarnych.
- Rozwiązanie niezrozumiałe dla ekspertów z dziedziny samego problemu.
- Żargon informatyczny jest często niezwiązany z dziedziną samego

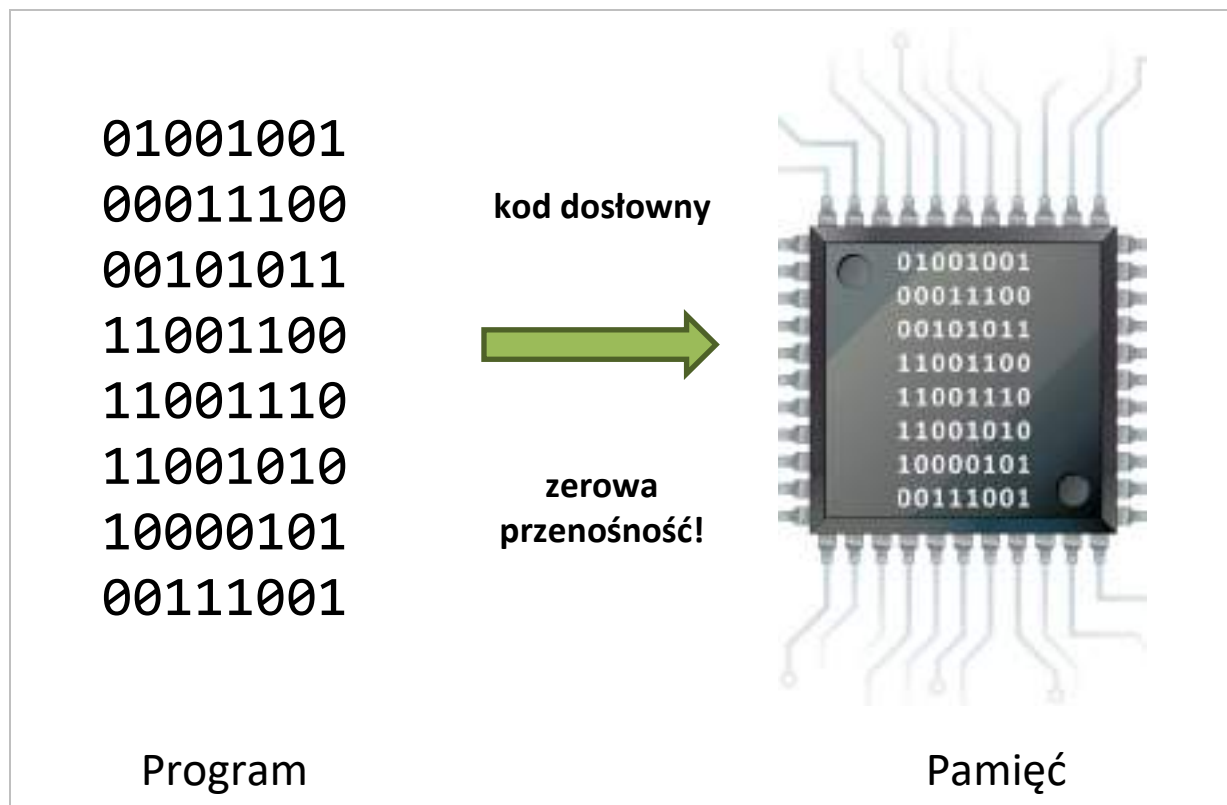
problemu.

Generacje metod programowania

- Kod maszynowy
- Języki assemblerowe
- Języki wysokopoziomowe
- Programowanie strukturalne
- Programowanie obiektowe

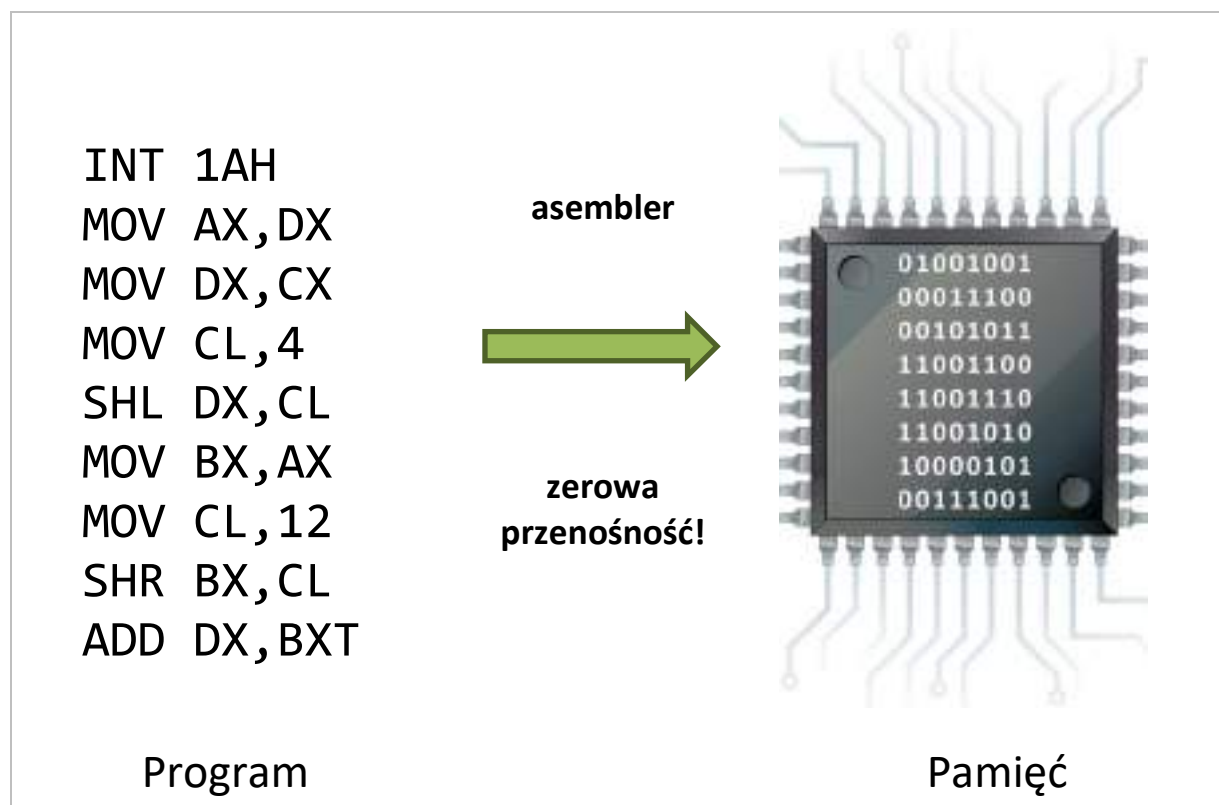
Kod maszynowy

- Program komputerowy składa się ze zbioru instrukcji realizowanych przez procesor:
 - odczyt/zapis do komórek pamięci
 - przesuwanie bloków danych,
 - transfer danych pomiędzy rejestrami,
 - operacje logiczne i arytmetyczne na wartościach binarnych.
- Instrukcje reprezentowane są za pomocą odpowiednich kodów binarnych.



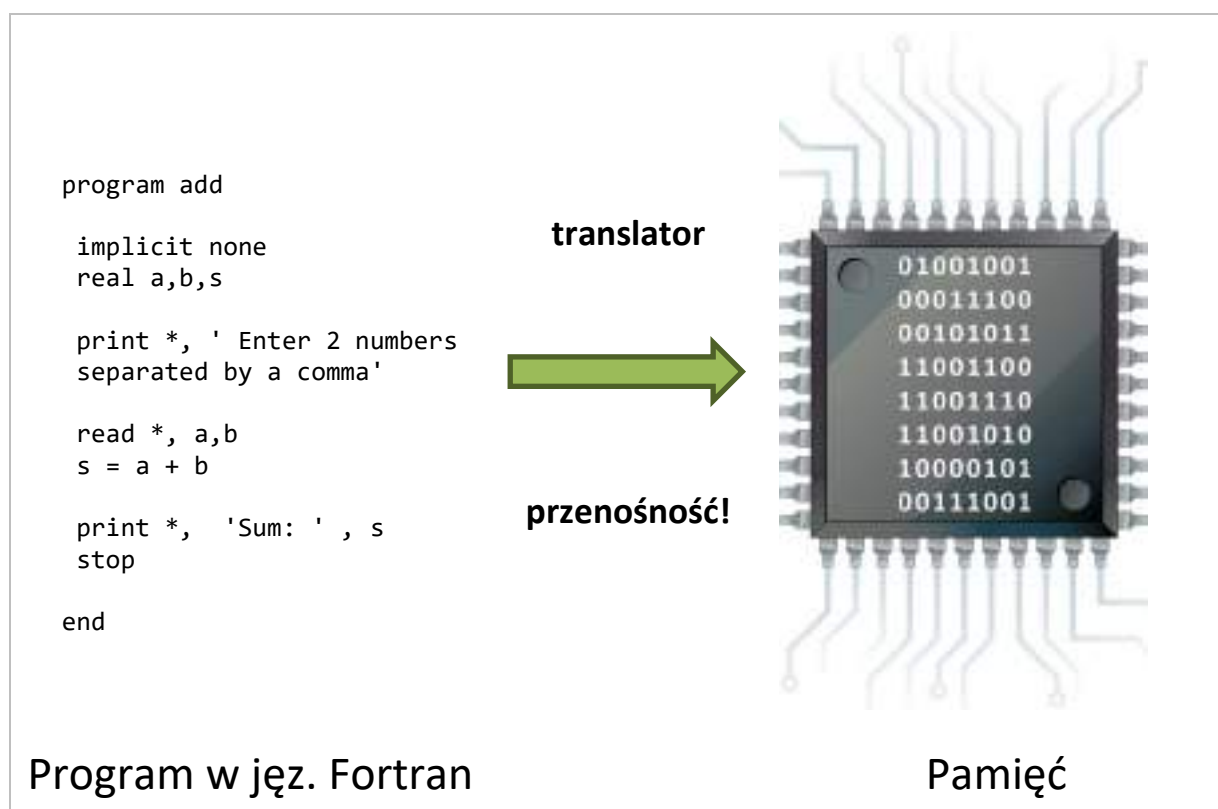
Języki asemblerowe

- Instrukcje reprezentowane są przez symbole alfanumeryczne lub mnemoników.
- Mnemoniki to skróty słowne oznaczające konkretne czynności procesora.
- Mnemoniki różnią się pomiędzy różnymi architekturami.
- Możliwe jest nazywanie komórek oraz bloków pamięci.
- Kod w języku asemblerowym tłumaczony jest do kodu maszynowego za pomocą asemblerów.



Języki wysokiego poziomu

- Instrukcje reprezentowane są za pomocą struktur o wyższym poziomie abstrakcji.
- Kod źródłowy składa się z kilku podstawowych instrukcji:
 - podstawienia (np. $a = 5$),
 - warunków (np. instrukcja warunkowa if),
 - pętli (np. pętla for),
 - podprogramów tworzących nowe instrukcje.
- Wykorzystanie struktur danych wysokiego poziomu:
 - wektorów,
 - macierzy,
 - rekordów,
 - plików.
- Kod źródłowy języków wysokiego poziomu jest tłumaczony do kodu maszynowego na pomocą translatorów
- Przykłady: Fortran, COBOL.



Języki proceduralne (programowanie strukturalne)

- Ulepszona wersja języków wysokiego poziomu.
- Większa czytelność i jakość kodu.
- Wysokie wykorzystywanie podprogramów, bloków kodu oraz pętli.
- Dynamiczne struktury danych: listy, grafy, drzewa.
- Możliwość tworzenia własnych struktur danych.
- Dostępność danych w zakresie całego projektu (zakres globalny zmiennych).
- Przykłady: C, Pascal, Basic, PHP, ALGOL, Modula, Ada.

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

```
main() {  
    int a, b, c;  
  
    printf("Enter 2 numbers to  
    add\n");  
  
    scanf("%d%d",&a,&b);  
    c = a + b;  
  
    printf("Sum: %d\n",c);  
  
    return 0;  
}
```

translator



przenośność!

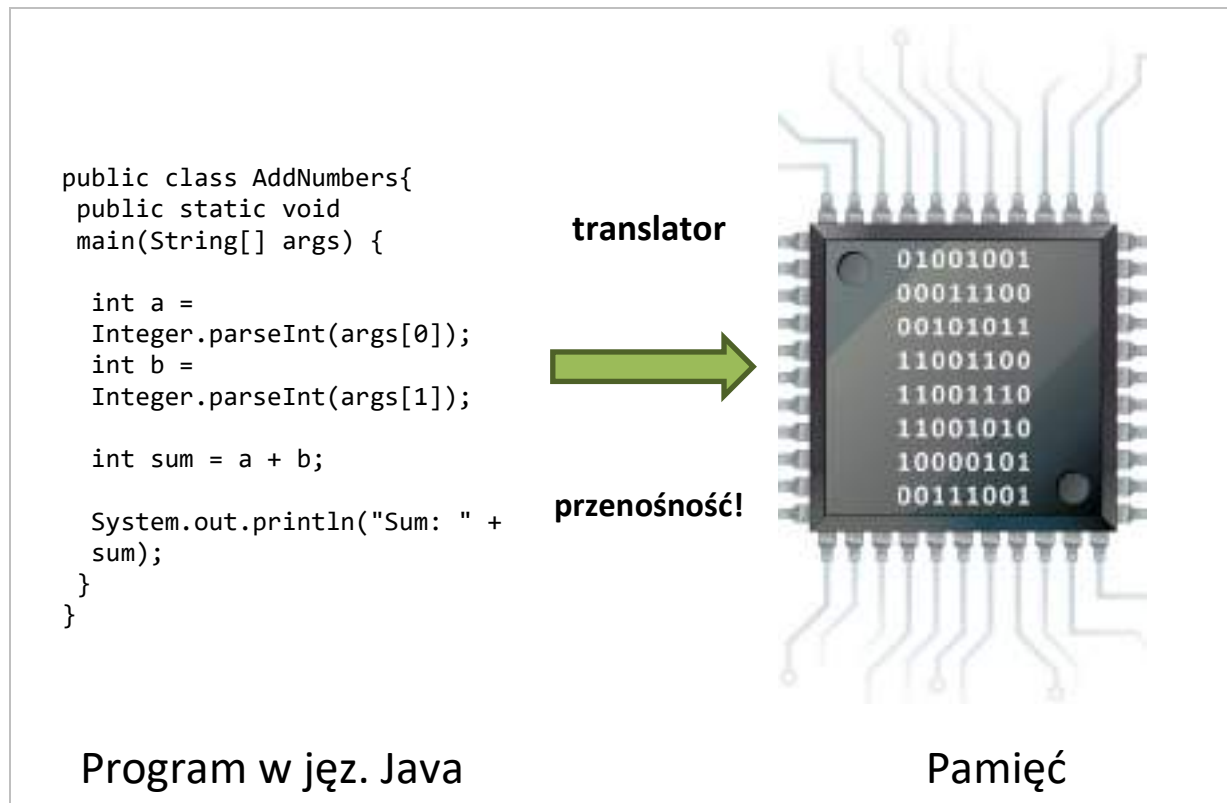


Program w C



Pamięć

Języki zorientowane obiektowo (programowanie obiektowe)

- Struktury danych oparte na świecie rzeczywistym.
- Wykorzystywanie niezależnych modułów (klas) odpowiadających konceptom z zakresu dziedziny problemu.
- Każdy moduł składa się z danych oraz funkcji.
- Możliwość tworzenia relacji pomiędzy modułami.
- Przykłady: C++, Java, Eiffel, Ruby, Smalltalk, C#.



Języki programowania – poziomy abstrakcji

<p>Poziom problemu</p> 	<p>Języki zorientowane obiektowo</p>	<ul style="list-style-type: none"> • znajomość sprzętu nie jest wymagana, • skupienie się na modelowaniu problemu z danej dziedziny, • moduły programu: <ul style="list-style-type: none"> ○ reprezentują rzeczywiste obiekty – ich atrybuty oraz zachowanie, ○ potrafią się komunikować, • przenośność kodu źródłowego.
	<p>Języki proceduralne</p>	<ul style="list-style-type: none"> • znajomość sprzętu nie jest wymagana, • skupienie się na rozwiązaniu problemu, • opis danych i opis algorytmów są niezależne, • wymagana translacja kodu, • przenośność kodu źródłowego.
	<p>Języki wysokiego poziomu</p>	<ul style="list-style-type: none"> • znajomość sprzętu nie jest wymagana, • program składa się z kilku podstawowych instrukcji, • wymagana translacja kodu, • kod źródłowy zrozumiały dla ludzi, • przenośność kodu źródłowego, • możliwość wykorzystywania skomplikowanych struktur danych.
	<p>Języki assemblerowe</p>	<ul style="list-style-type: none"> • znajomość sprzętu jest wymagana, • podatność na błędy, • czasochłonność, • zerowa przenośność kodu źródłowego.
<p>Poziom sprzętowy</p> 	<p>Kod maszynowy</p>	<ul style="list-style-type: none"> • znajomość sprzętu jest wymagana, • podatność na błędy, • czasochłonność, • zerowa przenośność kodu źródłowego.

Temat 3: Analiza i projektowanie obiektowe

Dekompozycja

- W przypadku projektowania skomplikowanego systemu informacyjnego, niezbędna jest dekompozycja problemu na mniejsze części.
- W wyniku takiej operacji, do zrozumienia działania systemu na pewnym poziomie, wystarczy zrozumienie kilku części systemu (nie trzeba zrozumieć całości za jednym podejściem).

Dekompozycja algorytmiczna

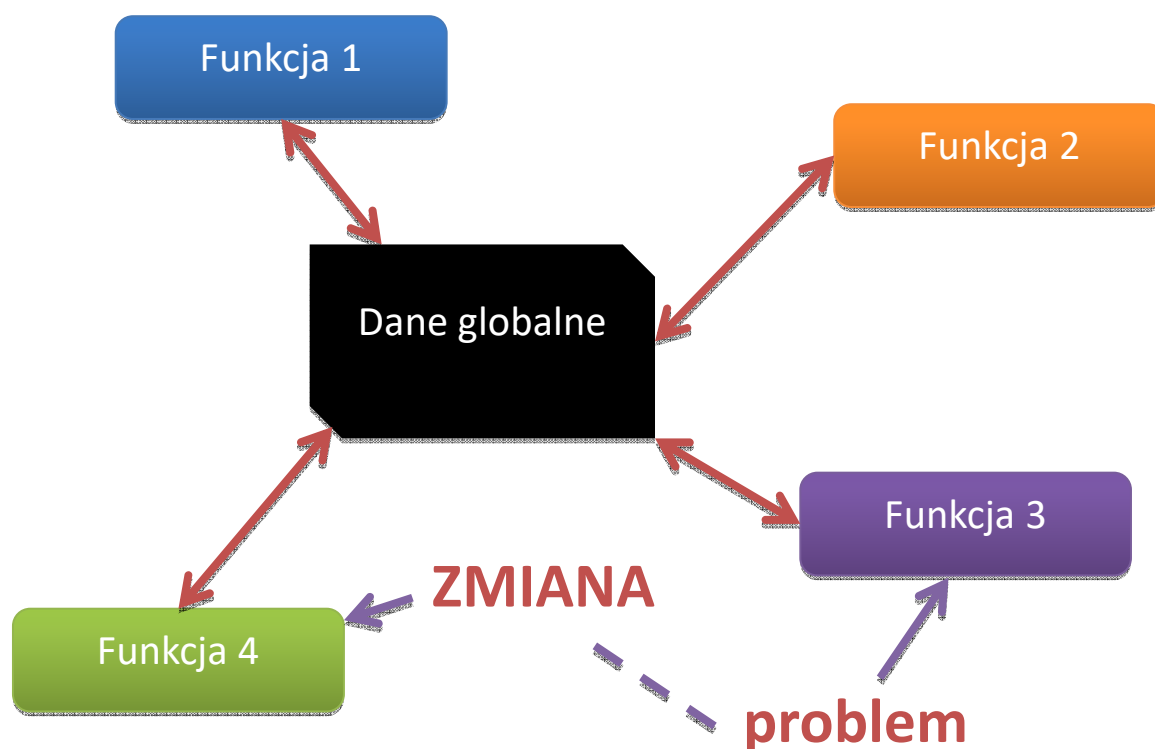
- Problem dzielony jest na części logiczne (abstrakcje algorytmiczne, elementy funkcjonalne).
- Każda część stanowi ważny element całości.

Dekompozycja obiektowa

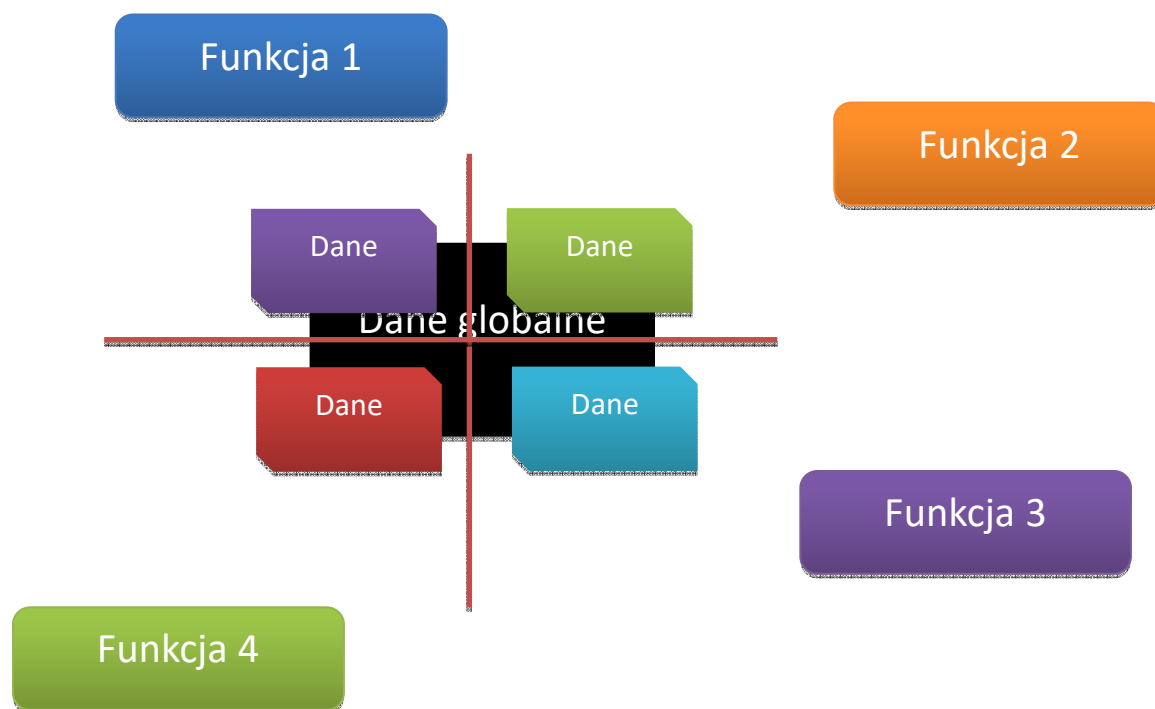
- Polega na wskazaniu modułów (abstrakcji obiektowych) współpracujących ze sobą na poziomie zachowań o wysokim poziomie abstrakcji.
- Każdy moduł odpowiada obiektowi w świecie rzeczywistym.

Przejście z programowania strukturalnego do programowania obiektowego

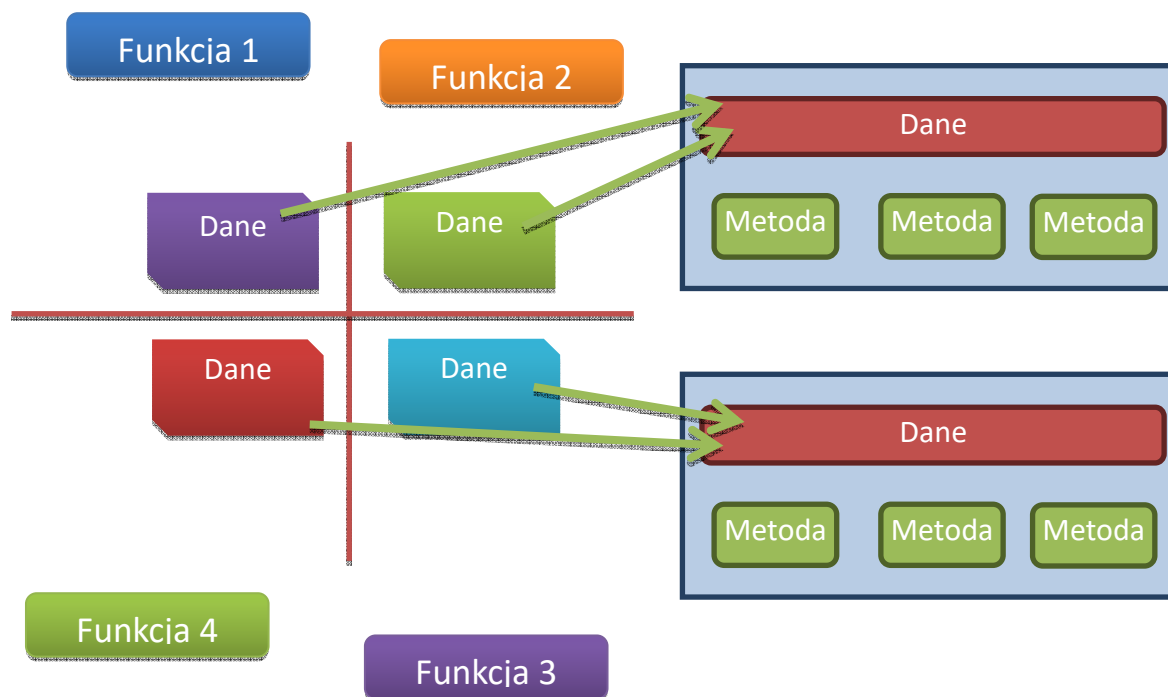
- W programowaniu strukturalnym dane są zwykle odseparowane od podprogramów i funkcjonują na zasadzie zmiennych globalnych dostępnych jednocześnie dla wszystkich podprogramów.
- Dostęp do danych nie jest ograniczony do zakresu konkretnego podprogramu, ani w żaden sposób kontrolowany. W rezultacie jedna funkcja może manipulować danymi, na których bazuje inna funkcja.
- Testowanie, debugowanie, pielęgnacja kodu oraz utrzymanie jego spójności jest bardzo trudne.



- W podejściu obiektowym dane globalne podzielone są na grupy.



- Grupy danych (wraz z metodami, wykonującymi operacje na tych danych) są schowane w logicznie wydzielonych **kontenerach**.
- Metody kontrolują dostęp do danych w zakresie danego kontenera i pozwalają na komunikację pomiędzy kontenerami.
- Te kontenery nazywane są **obiektami**.
- Obiekty łączą dane oraz metody w pojedynczym zestawie.



Czym jest obiekt?

- Czymś namacalnym i/lub widzialnym.
- Czymś, co może być pojęte za pomocą rozumu.
- Czymś, ku czemu możemy kierować myśli lub działania.

Obiekt w procesie tworzenia oprogramowania

- Reprezentuje indywidualny, identyfikowalny, rzeczywisty lub abstrakcyjny przedmiot, jednostkę lub element, o gruntownie zdefiniowanej roli w zakresie dziedziny problemu.
- Obiekt jest jednostką abstrakcyjną, utworzoną na podstawie świata rzeczywistego, posiadającą:
 - Stan
 - Reprezentowany za pomocą danych i wyrażany przy pomocy zbioru atrybutów.
 - Atrybutem nazywamy charakterystyczną dla danego obiektu cechę.
 - Stan obiektu reprezentowany jest przez skumulowany rezultat jego zachowań.
 - Zachowanie
 - Zachowaniem obiektu jest zestaw operacji określający sposób jego działania oraz reakcja na działania innych obiektów prowadzące się do zmiany stanu obiektu oraz wymiany informacji pomiędzy obiektami.
 - Operacją nazywamy akcję, którą obiekt jest w stanie wykonać.
 - Operacje nazywane są również metodami lub funkcjami składowymi.

- Funkcja składowa może być wykonana na obiekcie.
- Tożsamość
 - Tożsamość obiektu jest to jego właściwość, odróżniająca go od innych obiektów.
 - W praktyce jest to zwykle adres w pamięci, gdzie dany obiekt jest przechowywany.

Przykład obiektu

- Rzeczywiste obiekty posiadają cechy i podejmują działania.

Cechy

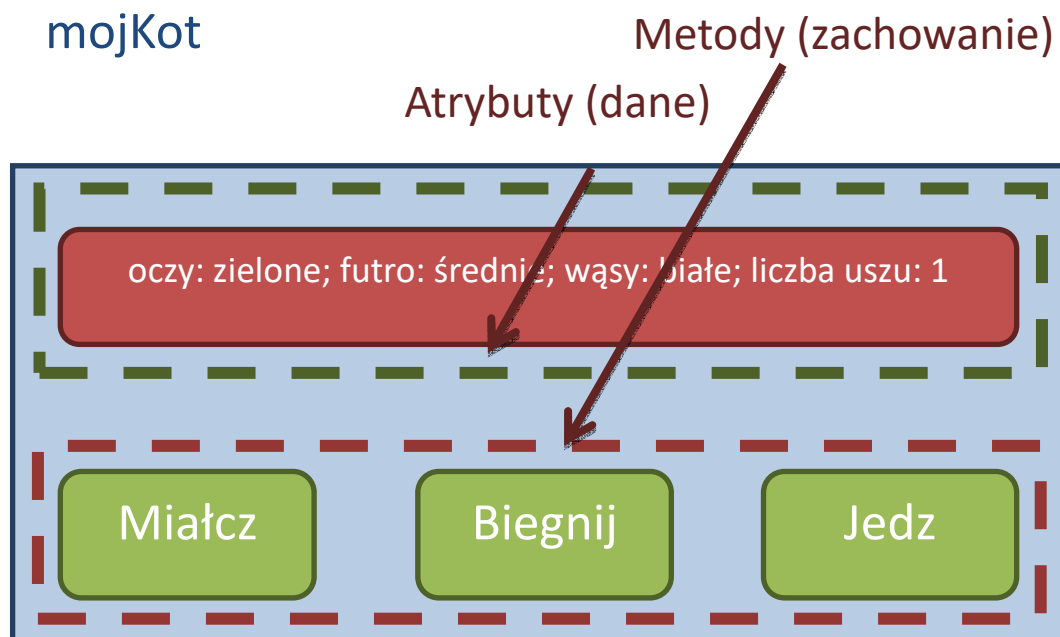
- Oczy: zielone
- Futro: średniej długości
- Wąsy: białe
- Liczba uszu: 1

Zachowanie

- Miałczy
- Biega
- Je



- Powyższe cechy i zachowania są reprezentowane w abstrakcyjnym obiekcie, jako atrybuty i funkcje składowe.



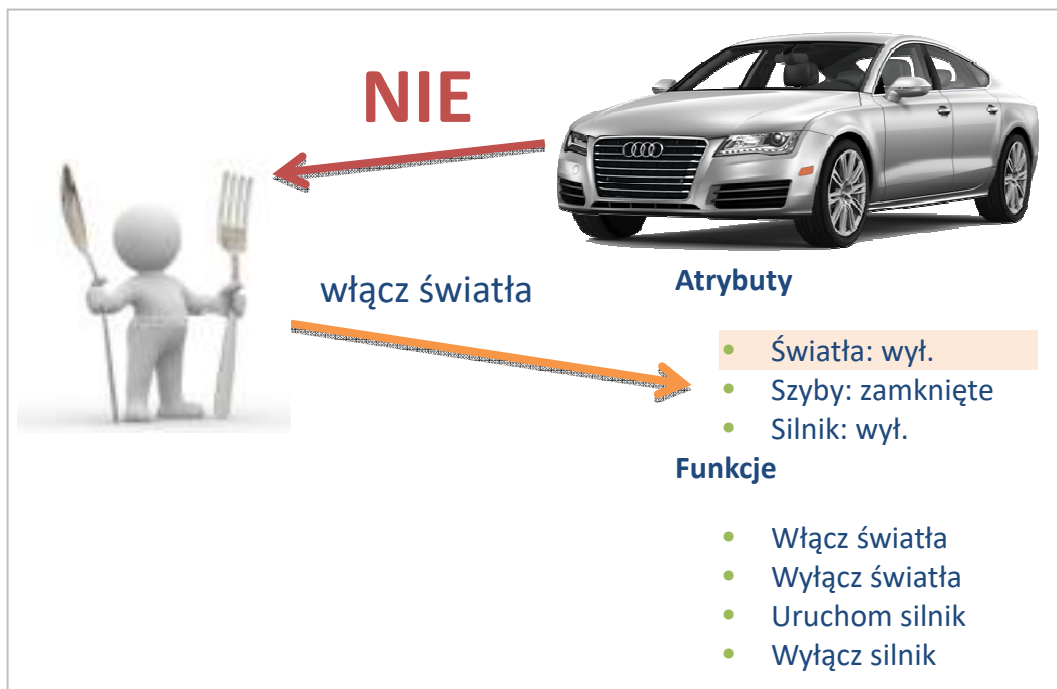
Komunikacja między obiektami

- Obiekty są podstawowym elementem budowania programów zorientowanych obiektowo.
- Obiekty komunikują się ze sobą za pomocą **wiadomości**.
- Wysłanie wiadomości = wywołaniu funkcji na obiekcie.
- Np.: jeżeli obiekt A wywołuje metodę na obiekcie B, oznacza to, że obiekt A przesyła wiadomość do obiektu B. Odpowiedzią obiektu B jest wartość zwracana przez wywołaną funkcję.

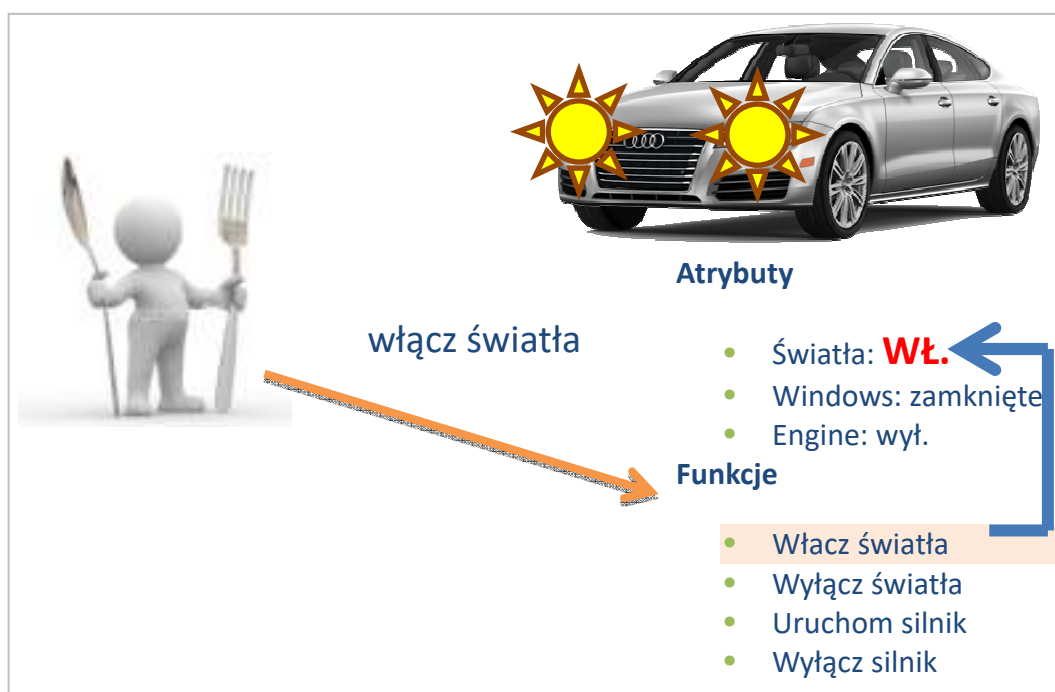


Integralność obiektu

- Obiekty posiadają ustalone atrybuty, które je charakteryzują składając się na stan obiektu.
- Obiekty posiadają również pewną **integralność**, która nie powinna być naruszana. Oznacza to, że obiekt może zmienić swój stan, podjęć jakieś działania, lub być manipulowanym wyłącznie w sposób właściwy dla danego obiektu.
- Dostęp do atrybutów powinien być kontrolowany za pomocą **specjalnych metod** (getterów i setterów).
- Bezpośrednia zmiana stanu obiektu nie powinna być możliwa (np. zmiana wartości atrybutu "Światła").



- Jednak powinna istnieć możliwość zmiany wartości atrybutu w sposób pośredni (np. poprzez funkcję "Włącz światła").



Czym jest klasa?

- Klasa jest szablonem dla obiektów o takiej samej strukturze i posiadających wspólny zbiór zachowań.
- Pojedynczy obiekt jest egzemplarzem klasy.
- OBIEKT != KLASA

Przykład klasy

- Różne gitary są różnymi obiektami w świecie rzeczywistym.
- Projekt (np. rysunek techniczny) gitary, na podstawie którego gitary zostały zbudowane jest odpowiednikiem klasy.



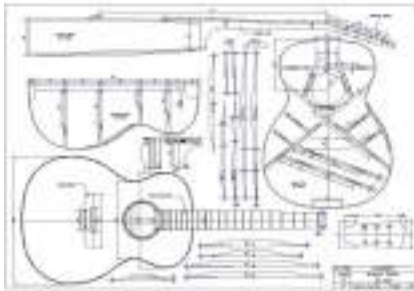
- Klasa jest wzorcem dla obiektów i definiuje:
 - cechy, jakie posiadają obiekty (atrybuty),
 - zachowania, które obiekty mogą podejmować (funkcje),
 - sposób realizacji tych zachowań (definicje funkcji).
- Obiekt posiada zbiór atrybutów zdefiniowanych przez klasę. Atrybutom w obiekcie przypisywane są wartości charakterystyczne dla danego obiektu.

Klasa

- materiał
- liczbaStrun
- kolor

Obiekt

- materiał: **drewno**
- liczbaStrun: **6**
- kolor: **Czarno-Biały**



Dekompozycja algorytmiczna czy dekompozycja obiektowa?

- **Podejście strukturalne** – kładzie nacisk na kolejność zdarzeń; atrybuty i zachowania są zwykle odseparowane.
- **Podejście obiektowe** - kładzie nacisk na jednostki, które podejmują akcje bądź podlegają akcjom; atrybuty i zachowania są zgromadzone w pojedynczym obiekcie.
- Nie możemy stworzyć zaawansowanego systemu używając dwóch podejść jednocześnie, ponieważ są one niezależne.

Którego podejścia używać?

- Najpierw użyj podejścia obiektowego.
- Lepiej sprawdza się podczas projektowania i pielęgnacji złożonego systemu.
- Zmniejsza rozmiary kodu z uwagi na łatwiejszą możliwość użycia raz napisanego kodu.
- Ma większą podatność na zmiany, dlatego z czasem łatwiej ewoluuje.

Programowanie obiektowe (OOP – Object-Oriented Programming)

- Metoda implementacji, w której elementami podstawowymi są współdziałające zbiory obiektów.
- Jako podstawowy budulec wykorzystuje obiekty, a nie algorytmy.
- Każdy obiekt jest egzemplarzem jakiejś **klasy**.
- Klasy mogą być powiązane mechanizmem dziedziczenia.
- Program może wydawać się napisany w języku obiektowym, jeśli jednak brakuje któregoś z powyższych elementów, najprawdopodobniej nie jest to program obiektowy.

Projektowanie obiektowe (OOD – Object-Oriented Design)

- Metoda projektowania bazująca na procesie dekompozycji obiektowej i reprezentacji zarówno logicznych i fizycznych, jak i statycznych i dynamicznych aspektów projektowanego systemu.
- Za pomocą modeli i mechanizmów próbuje sprostać wymaganiom powstałym w wyniku analizy.
- Niezależna od języka programowania.
- Wynikiem projektowania jest gotowy do implementacji model systemu.
- Wykorzystuje dekompozycję obiektową.
- Wykorzystuje różne notacje do opisu różnych aspektów systemu (logiczne, fizyczne, statyczne, dynamiczne).
- Jeżeli używamy metody, która prowadzi do dekompozycji obiektowej = używamy projektowania obiektowego.

Analiza obiektowa (OOA – Object-Oriented Analysis)

- Metoda analizy postrzegająca wymagania systemu i pojęć z nim związanych w kategoriach klas i obiektów występujących w dziedzinie problemu.
- Objaśnia i dokumentuje wymagania systemu.
- Kładzie nacisk na zrozumienie dziedziny problemu.
- Powstałe obiekty odzwierciedlają jednostki rzeczywiste powiązane z danym problemem.

OOA, OOD i OOP - współzależności

- Rezultat **analizy obiektowej** służy jako model, na podstawie którego dokonujemy **projektowania obiektowego**.
- Rezultat **projektowania obiektowego** służy jako projekt, na podstawie którego dokonujemy implementacji systemu używając metod **programowania obiektowego**.

Elementy podejścia obiektowo-zorientowanego

- Abstrakcja
- Enkapsulacja
- Modularność
- Hierarchia

Model nie zawierający przynajmniej jednego z powyższych elementów nie jest modelem obiektowym.

Abstrakcja

- Uproszczony opis obiektu, uwypatniający pożądane cechy obiektu, jednocześnie ignorujący cechy nieistotne.
- Koncentruje się na możliwych do obserwowalnych cechach i zachowaniach obiektu.
- Ułatwia analizę problemu.
- Przykład
 - Prawdziwa gitara istnieje w rzeczywistym świecie.
 - Model gitary zawiera wyłącznie tyle informacji, ile jest potrzebnych do jej prawidłowej reprezentacji w systemie.
 - Model abstrahuje (ignoruje) szczegóły, które nie są istotne z punktu widzenia systemu.

Enkapsulacja

- Proces ukrywania elementów abstrakcji stanowiących strukturę i zachowanie danego obiektu.
- Pozwala klasie na ukrycie szczegółów jej implementacji przed światem zewnętrznym, eksponując jednocześnie wybrane funkcje i dane, które są dostępne dla programisty (interfejs).
- Implementacja
 - informacja niezbędna do prawidłowego działania obiektu,
 - kombinacja funkcji i zasobów mająca na celu realizację istoty danej akcji,
 - zapewnia integralność informacji, na podstawie których bazuje dany obiekt.
- Interfejs
 - minimum niezbędnych informacji, aby móc używać dany obiekt,
 - pozwala programistom na dostęp do "wiedzy" obiektu,
 - musi być dostępny,
 - jest pośrednikiem w dostępie do wewnętrznej struktury obiektu.
- Przykład: Gitara elektryczna ukrywa przed użytkownikiem skomplikowane obwody elektryczne, na których nikt nie powinien bezpośrednio operować. Udostępnia za to struny, jako środek, za pomocą którego możemy osiągnąć nasz cel, czyli wydobyć dźwięk z instrumentu.

Modularność

- Proces dekompozycji do spójnych, luźno sprzężonych modułów.
- Jest sposobem porządkowania logicznie powiązanych abstrakcji.
- Każdy moduł powinien stanowić logiczną całość, która może być niezależnie skompilowana.
- Przykład: grupowanie różnych typów zwierząt (różnych klas) w jeden moduł o nazwie ZOO.

Hierarchia

- Jest zaszeregowaniem lub ustaleniem kolejności abstrakcji.
- Przykład: sedan jest typem samochodu, który z kolei jest typem pojazdu.

Temat 4: Wprowadzenie do języka UML

Czym jest modelowanie?

- Model jest uproszczoną reprezentacją rzeczywistości.
- Modelowanie jest procesem abstrahowania nieistotnych szczegółów świata rzeczywistego w celu stworzenia modeli.
- Wybór modeli zależy od podejścia do danego problemu.
- Każdy model może być wyrażony na różnych poziomach abstrakcji.
- Dobry model powinien być powiązany ze światem rzeczywistym.
- Żaden model w pojedynkę nie jest w stanie opisać złożonego systemu.

UML

Unified Modeling Language jest zbiorem graficznych notacji (diagramów), pomagających w opisywaniu i projektowaniu systemów informacyjnych. Jest wykorzystywany do określania, wizualizacji, modyfikowania, konstruowania i dokumentowania elementów systemu zorientowanego obiektowo.

Język UML został stworzony przez Gradyego Boocha i Jima Rumbaugh'a i publicznie pokazany w 1995 r. jako *Unified Method* ver.0.8. Duża część rynku zawładnięta przez firmę Rational (obecnie część firmy IBM), gdzie pracowali autorzy języka, przestraszyła konkurencję, która obawiała się dominacji na rynku narzędzi typu CASE tej właśnie firmy. W konsekwencji międzynarodowe konsorcjum OMG (Object Management Group) zostało poproszone o standaryzację metody w celu umożliwienia przenośności modeli pomiędzy różnymi narzędziami typu CASE. Rezultatem działań była pierwsza oficjalna wersja 1.1 standardu *Unified Modeling Language*. Ostatnią zatwierdzoną specyfikacją jest rodzina UML 2.x.

Sposoby wykorzystywania języka UML

Istnieją trzy różne podejścia do tworzenia diagramów w języku UML:

- **Szkic** – nieformalne, dynamiczne i najbardziej popularne podejście. Jego głównym założeniem jest wybiórczość. Koncentruje się bardziej na komunikacji i współdziałaniu, niż na kompletności projektu. Na diagramie ukazane są wyłącznie najistotniejsze elementy, dlatego często są one postrzegane, jako niedokończone. Najpopularniejszym nośnikiem diagramów jest tablica lub flipchart. Restrykcyjne przestrzeganie standardów UMLa nie jest praktykowane.
 -
 - Szkic progresywny (forward sketching). Jego celem jest przekazanie idei do stworzenia kodu źródłowego. Szkic progresywny skupia się na uwydatnieniu najważniejszych kwestii projektu i ich wizualizacji przed implementacją systemu.
 - Szkic odwrotny (reverse sketching) jest używany do objaśnienia działania funkcjonujących już części systemu (np. nowemu programiście, aby szybciej i łatwiej zapoznał się z mechanizmem działania systemu). Prezentowane są wyłącznie ogólne i najistotniejsze informacje.

- **Projekt** – podejście, w którym najważniejsze są kompletność i szczegółowość. Diagramy powinny być wyczerpujące i precyzyjne, aby sprowadzić programowanie do czynności mechanicznej. Taki stopień zaawansowania wymaga specjalistycznych narzędzi zwanych CASE (computer-aided software engineering). Ułatwiają generowanie kodu źródłowego na podstawie utworzonych diagramów (inżynieria progresywna), jak i interpretację kodu źródłowego w celu utworzenia graficznych diagramów UML (inżynieria odwrotna). W zależności od ustalonego stopnia szczegółowości, podejście to może przyczynić się do spowolnienia procesu twórczego oprogramowania.
 - Inżynieria progresywna ma miejsce, gdy projektant utworzy szczegółowy projekt systemu, na podstawie którego programista jest w stanie wiernie system zaimplementować. Projekt powinien być tak zaawansowany, aby programowanie wymagało jak najmniej myślenia. Takie podejście może jednak negatywnie odbić się na procesie produkcji systemu.
 - Inżynieria odwrotna sprowadza się do tworzenia szczegółowych diagramów systemu na podstawie istniejącego kodu. Diagramy stanowią pomoc dla nowych deweloperów, zapoznających się z kodem źródłowym aplikacji.
- **Język programowania.** Gdy czynność programowania staje się coraz bardziej mechaniczna, w którymś momencie powinna być ona zautomatyzowana. Punkt, w którym modele UML opisują dany system w stopniu tak, szczegółowym, że mogą być one skompilowane do kodu wykonywalnego, jest punktem, w którym UML staje się językiem programowania. Takie podejście sprawia, że terminy "inżynieria progresywna" i "inżynieria odwrotna" przestają mieć znaczenie, gdyż UML i kod źródłowy są jednym i tym samym. Takie postępowanie wymaga skomplikowanych narzędzi i ogromnego stopnia szczegółowości modeli, które poważnie mogą zagrozić efektywności procesu produkcji. Debata na temat wyższości form graficznych nad formami tekstowymi nadal trwa. Ponadto, specyfikacja UMLa nie wspomina o żadnych powiązaniach z jakimkolwiek językiem programowania.

Notacje i meta-modele

Język UML bazuje na notacjach oraz meta-modelach.

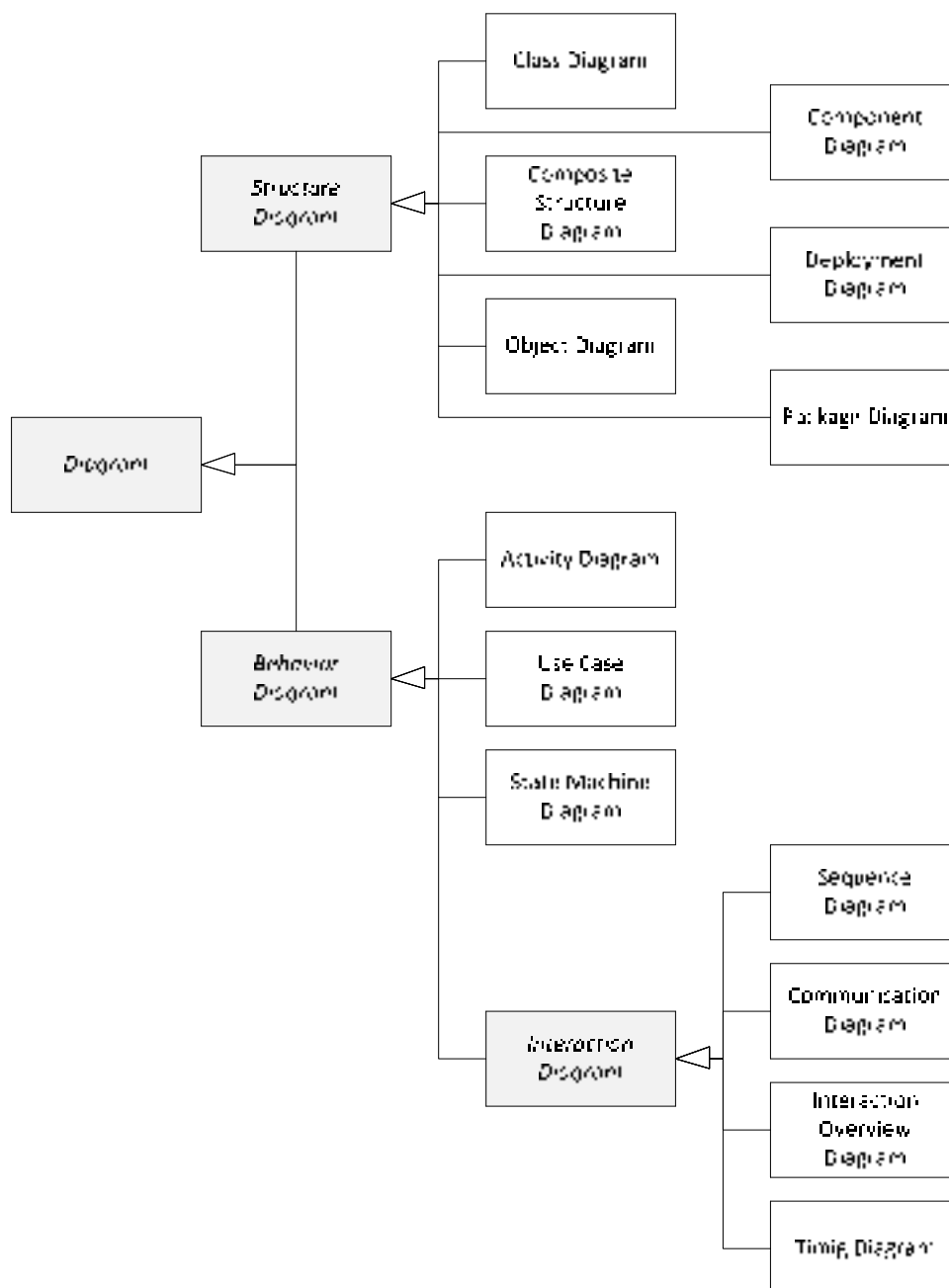
Notacja UML to składnia języka, definiująca graficzny sposób reprezentacji elementów. Semantyka elementów języka UML jest opisana za pomocą meta-modeli. Rygor restrykcyjnego podporządkowania się tym regułom jest różny dla różnych podejść.

- Szkicownicy – tak długo jak idea skutecznej komunikacji jest spełniona, nie muszą być zbyt restrykcyjni.
- Projektanci – z uwagi na bliskie relacje pomiędzy diagramami UML a kodem źródłowym, muszą być bardziej restrykcyjni niż Szkicownicy.
- Ludzie traktujący język UML, jako język programowania – muszą restrykcyjnie stosować się do wszystkich zasad języka, gdyż meta-modele definiują abstrakcyjną składnię tego języka jako języka programowania.

Diagramy UML

Podstawową notację UML są logicznie powiązane diagramy. Diagramy pozwalają na opis systemu na pożądanym stopniu szczegółowości: od ogólnych do bardzo złożonych i szczegółowych modeli. UML 2 wprowadza 13 oficjalnych typów diagramów. Każdy służy do innego celu.

Poniżej przedstawiona została klasyfikacja diagramów. Strzałki wskazują od diagramów bardziej szczegółowych do bardziej ogólnych.



Istnieją dwie główne kategorie diagramów: abstrakcyjne i konkretne. Diagramy abstrakcyjne (czcionka pochyła, szare tło) to wyłącznie nazwy do zgrupowania diagramów konkretnych.

W rezultacie diagramy abstrakcyjne to:

- Diagram – kontener dla wszystkich diagramów.
- Diagram struktury (Structure Diagram) – pozwala na modelowanie struktury systemu, pokazanie jego elementów składowych i atrybutów niezmiennych w czasie.
- Diagram dynamiki (Behavior Diagram) – jest wykorzystywany w celu pokazania dynamiki systemu. Pokazuje jak system reaguje na żądania lub zmiany w czasie.
- Diagram interakcji (Interaction Diagram) – jest typem diagramu dynamiki i wizualizuje interakcję pomiędzy współdziałającymi obiektami. Pokazuje jak obiekty przekazują informacje i kooperują.

Diagramy konkretne stanowią pełną specyfikację standardu UML 2. Ich opisy znajdują się poniżej.

Kategoria	Diagram	Opis
Diagram struktury	klas (Class)	Pokazuje klasy, cechy, interfejsy i współzależności pomiędzy nimi.
Diagram struktury	komponentów (Component)	Pokazuje istotne komponenty wewnętrzne systemu i interfejsów służących do komunikacji z innymi interfejsami.
Diagram struktury	struktur połączonych (Composite Structure)	Pokazuje wewnątrz klasy oraz relacje pomiędzy jej elementami.
Diagram struktury	rozlokowania (Deployment)	Pokazuje sprzęt i architekturę systemu oraz sposób ich wdrożenia w świecie rzeczywistym.
Diagram struktury	obiektów (Object)	Pokazuje przykładowe egzemplarze klas oraz relacje między nimi w określonym punkcie czasu.
Diagram struktury	pakietów (Package)	Pokazuje hierarchiczną organizację klas wraz z ich zależnościami.
Diagram dynamiki	czynności (Activity)	Pokazuje przepływ danych w ramach sekwencyjnych oraz równoległych czynności w systemie.
Diagram dynamiki	przypadków użycia (Use Case)	Pokazuje usługi dostępne dla aktorów systemu.
Diagram dynamiki	maszyny stanowej (State Machine)	Pokazuje różne stany (oraz zdarzenia mogące zmienić te stany) danego obiektu w perspektywie cyklu jego życia.
Diagram interakcji	sekwencji (Sequence)	Pokazuje interakcje pomiędzy obiektami śledząc kolejność wymiany informacji pomiędzy nimi.
Diagram interakcji	komunikacji (Communication)	Pokazuje interakcje grup obiektów oraz zależności wspomagające te interakcje.
Diagram interakcji	sterowania interakcją (Interaction Overview)	Pokazuje diagramy sekwencji, komunikacji i harmonogramowania razem dla uchwycenia istotnych zależności w systemie.
Diagram interakcji	harmonogramowania (Timing)	Pokazuje na osi czasu interakcje pomiędzy obiektami oraz zmiany ich stanów.

Bardzo rzadko w jednym projekcie wykorzystywane są wszystkie typy diagramów. Niektórych z nich używa się wyłącznie w specyficznych sytuacjach. W praktyce wystarczy jedynie niewielki podzbiór diagramów. Diagramy klas, sekwencji i przypadków użycia są najbardziej popularnymi diagramami.

Temat 5: Diagram przypadków użycia

Przypadek użycia

Przypadek użycia (ang. use case) to lista kroków, które system musi wykonać, aby osiągnięty został konkretny cel. Przypadki użycia są najczęściej wykorzystywane w inżynierii oprogramowania do opisu wymagań tworzonego system informacyjnego.

Przypadek użycia definiuje interakcje pomiędzy aktorem a systemem. Pod pojęciem aktora kryje się zarówno człowiek (pełniąc różne role) jak i zewnętrzny system.

Każdy przypadek użycia ma jedną ścieżkę główną oraz co najmniej jedną ścieżkę alternatywną. Wszystkie możliwe sytuacje powinny zostać wzięte pod uwagę podczas opracowywania przypadku użycia.

Reasumując przypadek użycia opisuje jaką funkcjonalność musi posiadać system aby zaspokoić konkretne wymaganie użytkownika. Jednakże przypadek użycia nie powinien zawierać szczegółów implementacji ani dotyczących interfejsu użytkownika.

Jak wynika to z poprzednich akapitów każdy przypadek użycia musi mieć oczywiste znaczenie. Jest to konsekwencja tego, iż opisuje on realizację przez system konkretnego celu. Ponadto każdy przypadek użycia musi mieć ściśle zdefiniowany początek i koniec. Musi być wiadome jaki jest pierwszy krok w danym przypadku użycia. Konieczne jest również precyzyjne określenie warunku stopu, czyli momentu który go kończy. Ponadto dla każdego takiego przypadku użycia musimy zidentyfikować zewnętrzny czynnik inicjujący, który wywoła rozpoczęcie pierwszego kroku z listy.

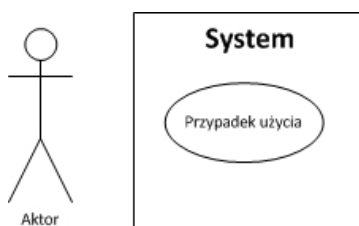
Kompletna ścieżka (czy to główna czy też alternatywna) prowadząca od pierwszego do ostatniego kroku przypadku użycia nazywana jest scenariuszem. Im więcej ścieżek alternatywnych tym więcej scenariuszy. Ważne jest jednak, że każdy z tych scenariuszy (ścieżek) realizuje jeden określony cel systemu.

Diagram przypadków użycia

Diagram przypadków użycia to jeden z behawioralnych diagramów UML. Oznacza to, że jest sposobem opisu zachowania tworzonego lub badanego systemu. Jest on graficzną reprezentacją przypadku użycia przy czym jeden diagram przypadków użycia może prezentować kilka przypadków użycia na raz. Diagram przypadków użycia może opisywać kilku aktorów (reprezentujących różne role), którzy wchodzi w różne interakcje z systemem. Diagramowi takiemu towarzyszy najczęściej tekstowy opis przypadku użycia lub też diagramy innych typów.

Poniższy rysunek przedstawia podstawowe elementy diagramu przypadków użycia:

- Aktora
- Przypadek użycia
- Granice systemu

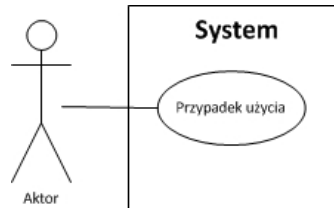


Aktorzy umieszczani są na diagramie poza systemem. Z kolei przypadki użycia jako cele/funkcjonalności systemu na diagramie znajdują się wewnątrz systemu.

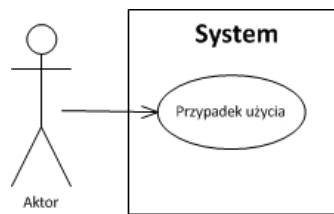
Pomiędzy aktorami i przypadkami użycia występują związki. Każdy aktor musi być bezpośrednio powiązany z co najmniej jednym przypadkiem użycia. Z kolei każdy przypadek użycia musi być powiązany z co najmniej jednym aktorem, ale nie musi to być powiązanie bezpośrednie. W diagramie przypadków użycia nie umieszcza się nazwy związku.

Według standardu UML można wyróżnić następujące rodzaje związków:

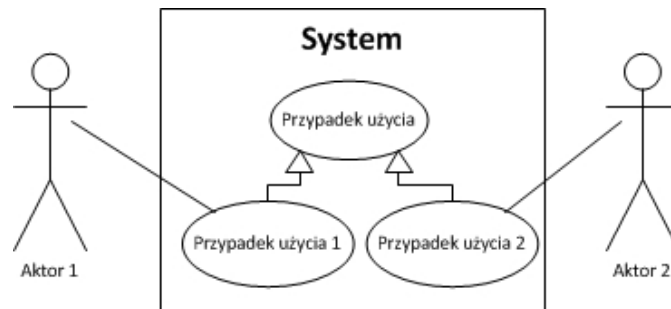
- Asocjacja (ang. association) – relacja oznaczająca komunikację dwukierunkową pomiędzy przypadkiem użycia a aktorem.



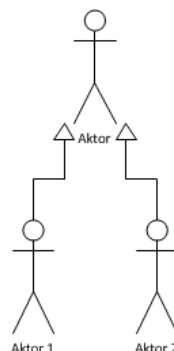
- Asocjacja skierowana (ang. directed association) – asocjacja skierowana to asocjacja, która wskazuje kierunek relacji; używana kiedy chcemy ukazać inicjatora interakcji



- Uogólnienie (ang. generalization) – oznacza, że pewien przypadek użycia może być szczególną odmianą innego, już istniejącego, przypadku użycia

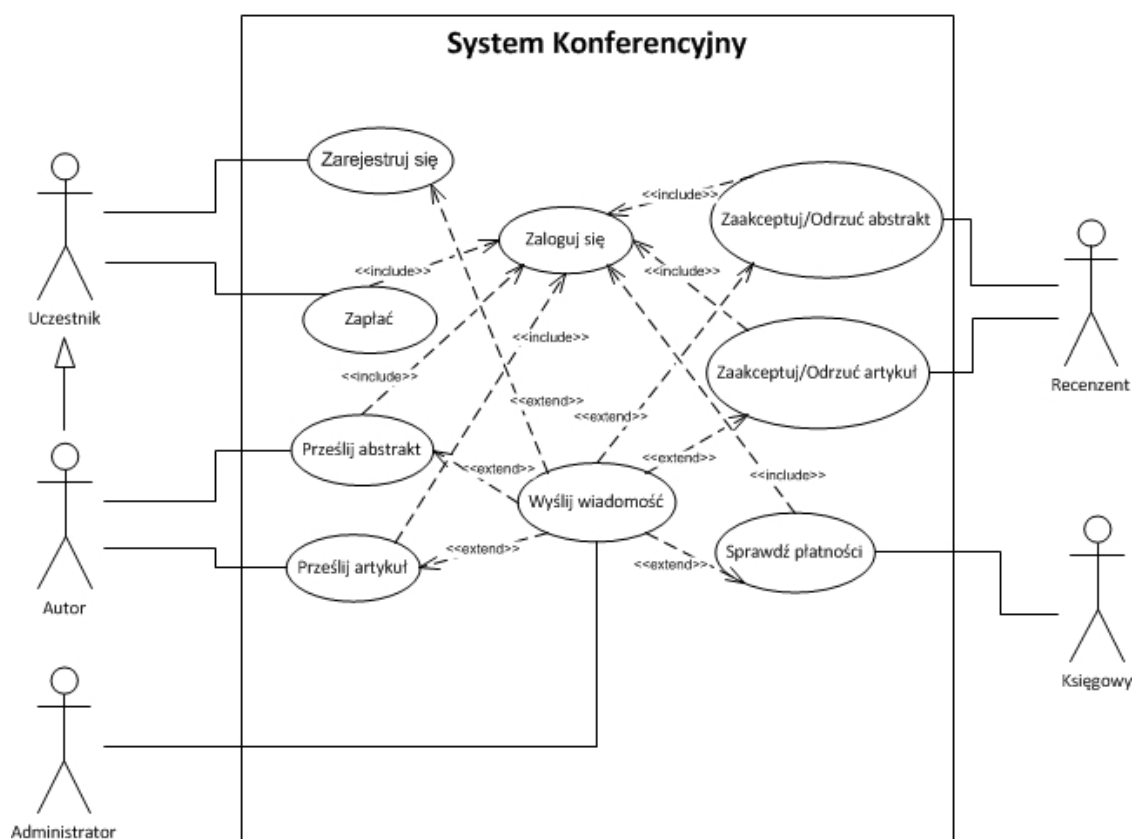


Uogólnienie może dotyczyć również aktorów. W praktyce oznacza to, że jeden lub kilku aktorów może być szczególną odmianą innego, już istniejącego aktora (pełnić pochodną rolę).



- Zależność (ang. dependancy) – w diagramie przypadków użycia mogą występować pomiędzy poszczególnymi przypadkami użycia zależności różnego rodzaju. Najczęstsze z nich to:
 - `<<include>>` -----> oznacza, że jeden z przypadków użycia co najmniej raz korzysta z innego przypadku użycia.
 - `<<extends>>` -----> oznacza, że jeden przypadek użycia wywołuje kolejny (jeden lub wiele) przypadek użycia
- Realizację (ang. realization) – to specjalny rodzaj relacji pomiędzy dwoma przypadkami użycia z których jeden jest modelem definiującym zachowanie systemu, a drugi realizacją tego modelu. Symbol tej relacji to

----->



Przykład powyżej przedstawia podstawowe funkcjonalności systemu konferencyjnego. Poniżej zaprezentowano pojedynczy przypadek użycia.

Nazwa przypadku użycia: Prześlij abstrakt:

Kroki::

1. Zaloguj się do systemu konferencyjnego
2. Napisz tytuł pracy
3. Załaduj plik zawierający abstrakt
4. Naciśnij przycisk „Wyślij”

Wyzwalacz: Wybranie opcji „Prześlij abstrakt”

Alternatywna ścieżka:

1. Zaloguj się do systemu konferencyjnego
 - 1a. Podano błędne dane. Wróć do kroku 1
2. Napisz tytuł pracy
 - 2a. Wpisz współautorów artykułu (ich imiona i nazwiska, tytuły oraz afiliację)
3. Załaduj plik zawierający abstrakt
4. Naciśnij przycisk „Wyślij”
 - 4a. Brakuje danych. Wróć do kroku 2
 - 4b. Nie załączono pliku. Wróć do kroku 3

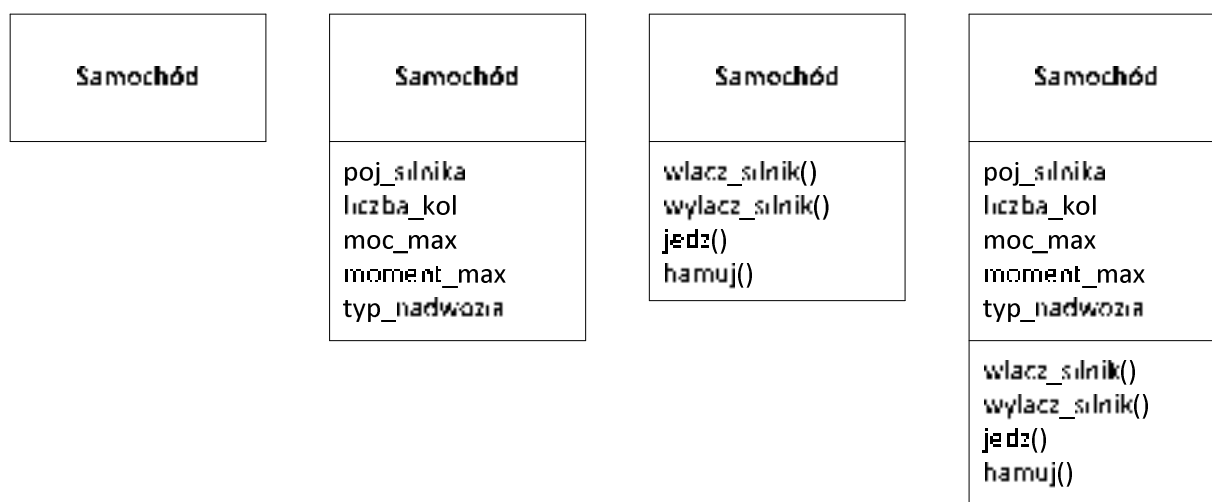
Temat 6: Diagramy klas i obiektów

Diagramy klas

- Najbardziej popularny typ diagramów.
- Pomagają wymodelować logiczną strukturę systemu.
- Opisują typy obiektów w systemie oraz ich relacje.

Klasy w języku UML

- Każda klasa jest reprezentowana jako prostokąt podzielony na trzy sekcje:
 - górna zawiera nazwę klasy,
 - środkowa zawiera atrybuty,
 - dolna zawiera funkcje.
- Istnieje kilka różnych możliwości graficznej reprezentacji klasy:
 - wyłącznie nazwa,
 - nazwa klasy z atrybutami,
 - nazwa klasy z funkcjami,
 - nazwa klasy z atrybutami i funkcjami.



- Atrybuty i funkcje są opcjonalne. Jeśli nie są pokazane, niekoniecznie znaczy to, że nie istnieją. Może to oznaczać, że zostały ukryte w celu zwiększenia czytelności diagramu.

- Cechy dobrej nazwy klasy

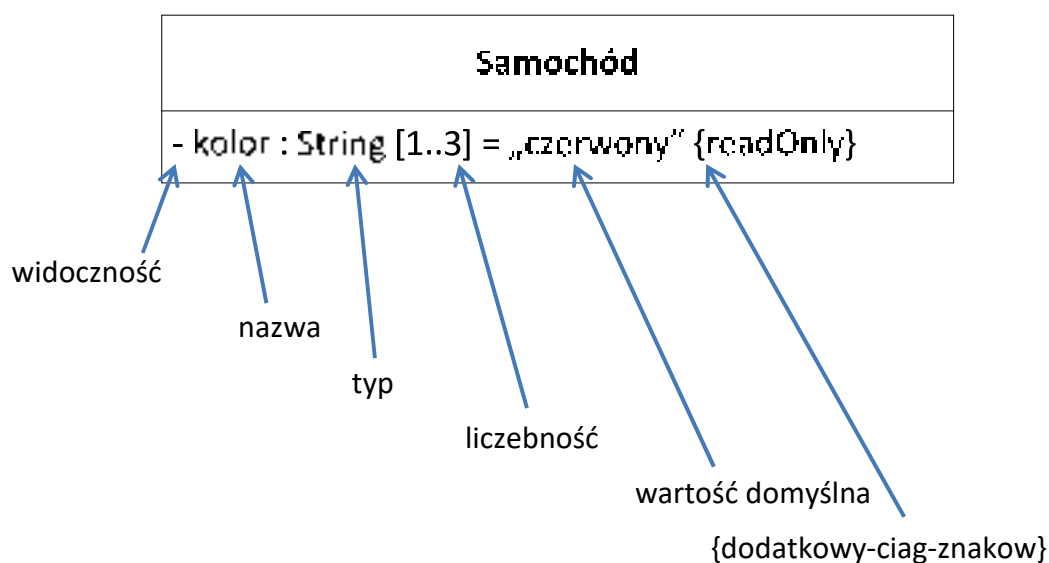
Używa rzeczowników	moje te wspaniałe baseny
Jest w liczbie pojedynczej	mój ten wspaniały basen
Unika zaborności	ten wspaniały basen
Nie posiada zbędnych przymiotników	ten basen
Jest pogrubiona i wyśrodkowana	ten basen
Używa majuskuły	Ten Basen
Nie posiada spacji pomiędzy słowami	TenBasen
Nie zawiera przedimków i zaimków	Basen

Atrybuty

- Diagram klas reprezentuje atrybut w postaci linii tekstu w odpowiedniej części prostokąta klasy.
- Pełna forma atrybutu:

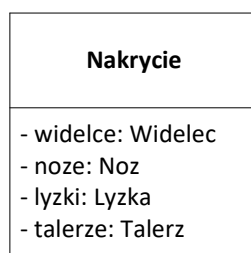
```
widocznosc nazwa : typ liczebosc = wartosc_domyslna {dodatkowy-  
ciag-znakow}
```

- **widoczność** wskazuje, czy atrybut ma widoczność typu public, private, protected czy package,
- **nazwa** atrybutu odpowiada nazwie pola klasy w kodzie źródłowym,
- **typ** atrybutu ogranicza jego wartość do konkretnego rodzaju obiektu,
- **liczebność** wskazuje jak wiele egzemplarzy klas może być ze sobą powiązanych,
- **wartosc_domyslna** jest wartością atrybutu dla nowo utworzonego obiektu, jeśli w trakcie tworzenia nie została podana żadna inna wartość,
- **{dodatkowy-ciag-znakow}** pozwala na wskazanie dodatkowych cech atrybutu.

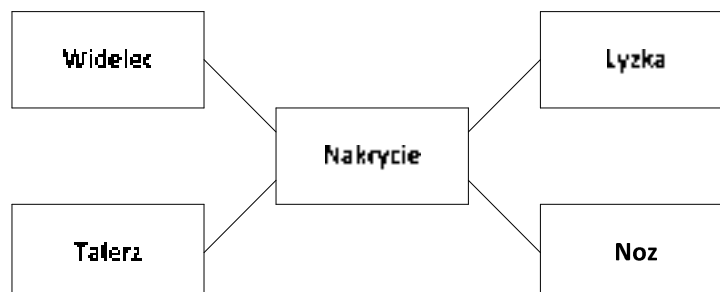


Atrybuty typu inline vs. atrybuty za pomocą asocjacji

- Atrybuty mogą się pojawiać w dwóch różnych notacjach:
 - inline



- za pomocą asocjacji



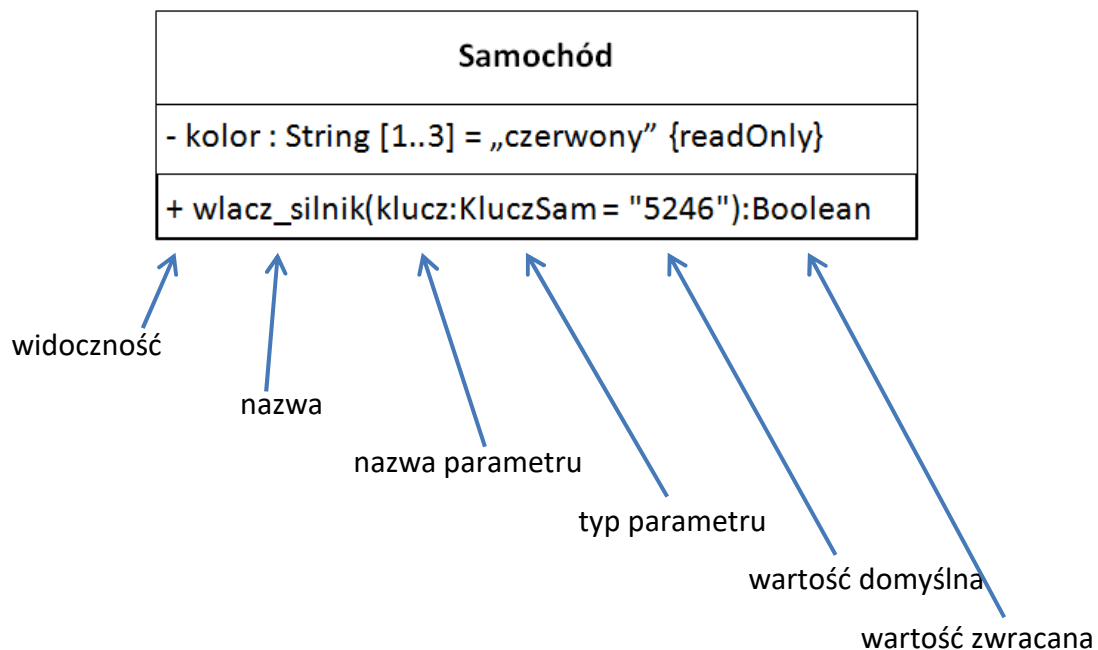
- Obie notacje mają to samo znaczenie i mogą być stosowane zamiennie.
- Z atrybutami typu inline diagramy zwykle są bardziej przejrzyste i czytelne, lecz czasem podejście drugie może być pożądane ze względu na przekazanie konkretnej idei.

Funkcje

- Funkcje są działaniami, które obiekt może wykonać.
- Na diagramach nie umieszcza się informacji na temat sposobu realizacji działań klasy.
- Pełna forma funkcji:

`widoczność nazwa (lista-argumentow) : typ-zwracany {dodatkowy-ciag-znakow}`

- **widoczność** wskazuje, czy funkcja ma widoczność typu public, private, protected czy package,
- **nazwa** jest ciągiem znaków,
- **lista-argumentow** jest listą parametrów, które funkcja może przyjmować; każdy parametr składa się z **nazwy**, **typu** i **wartosci-domyslnej**, których znaczenie jest takie samo jak dla atrybutów,
- **typ-zwracany** jest typem wartości zwracanej przez funkcję, jeśli funkcja cokolwiek zwraca,
- **{dodatkowy-ciag-znakow}** pozwala na wskazanie dodatkowych cech funkcji.



Widoczność

- Widoczność jest cechą kontrolującą dostęp do atrybutów i funkcji danej klasy.
- Każdy atrybut i funkcja w klasie powinna mieć określoną widoczność.

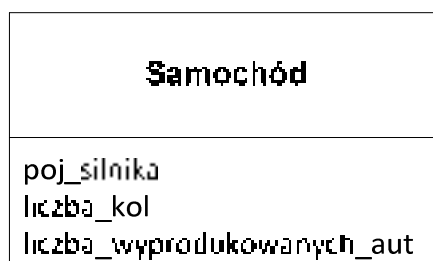
- W języku UML istnieją cztery typy widoczności

Widoczność	Symbol	Opis
Public	+	dostępne bezpośrednio przez obiekty innych klas
Protected	#	dostępne wyłącznie przed obiekty danej klasy i obiekty jej potomków
Packet	~	dostępne przez obiekty znajdujące się w tym samym pakiecie, co dana klasa
Private	-	dostępne wyłącznie przez obiekty danej klasy

- Zwykle unika się atrybutów o widoczności typu public, aby zminimalizować ryzyko utraty kontroli nad danym atrybutem lub funkcją.
- Dostęp do atrybutów powinien być realizowany za pomocą funkcji publicznych.

Atrybuty i funkcje statyczne

- Atrybuty instancyjne – każdy obiekt przechowuje własny zbiór wartości atrybutów.
- Atrybuty statyczne – wszystkie obiekty danej klasy współdzielą niektóre atrybuty, które są charakterystyczne jednocześnie dla nich wszystkich.
- Atrybuty statyczne odnoszą się do klasy, a nie do obiektów.
- Cechy statyczne są podkreślone na diagramach klas.



Diagramy obiektów

- Diagram obiektów jest migawką obiektów systemu w danym punkcie czasu.
- Użyteczny przy próbie pokazania przykładowej konfiguracji obiektów.
- Nazwy obiektów są podkreślone.
- Każda nazwa przyjmuje formę **nazwa obiektu : nazwa klasy**.
- Nazwa klasy (z dwukropkiem) może zostać pominięta, jeśli jasno wynika z kontekstu.

- Alternatywnie nazwa obiektu może zostać pominięta (dwukropek i nazwa klasy pozostaje) w celu zasygnalizowania, że można utworzyć obiekt anonimowy.

brak klasy

FordFocus

wraz z klasą

FordFocus : Samochod

obiekt anonimowy
danej klasy

: Samochod

- Przykład klasy i jej egzemplarza (obiektu)

Klasa

Obiekt

Samochod

poj_silnika
liczba_kol
moc_max
moment_max
typ_nadwozia

FordFocus : Samochod

poj_silnika = 1.6
liczba_kol = 4
moc_max = 120
moment_max = 330
typ_nadwozia = "hatchback"

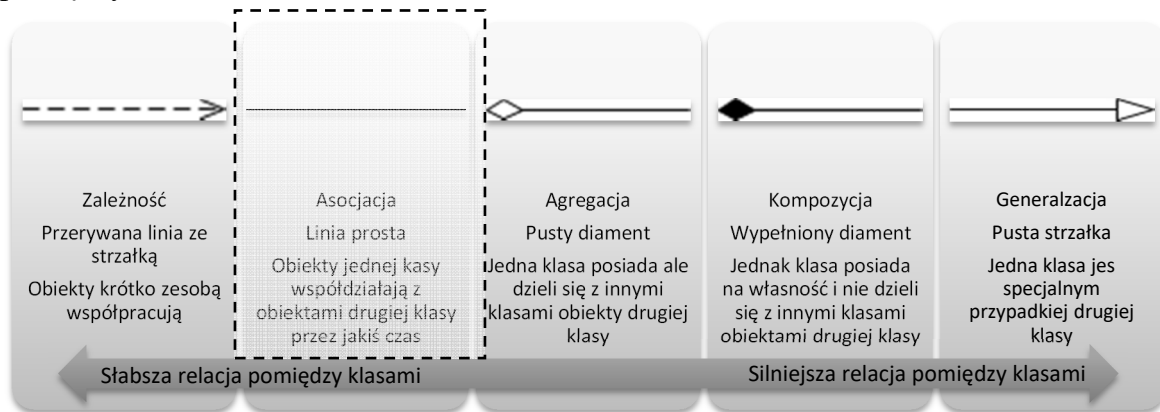
Temat 7: Związki pomiędzy klasami i obiektami

Związki

Na diagramach klas i obiektów związki pomiędzy klasami czy obiektami mogą być przedstawione. Rysunek poniżej przedstawia różne typy relacji. Możemy wyróżnić:

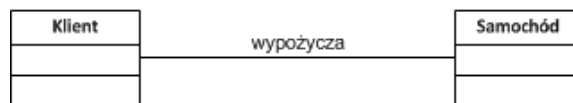
- Zależność
- Asocjacje
- Agregację (częściową)
- Kompozycję (agregację całościową)
- Generalizację

Każdy z tych rodzajów relacji posiada inny symbol reprezentujący go na diagramie. Jednakże każda relacja zawsze wiąże dwie klasy (jedynym wyjątkiem jest relacja zwrotna oraz klasa asocjacji). Temat ten jest skoncentrowany na opisie asocjacji jako związku pomiędzy klasami lub obiektami.

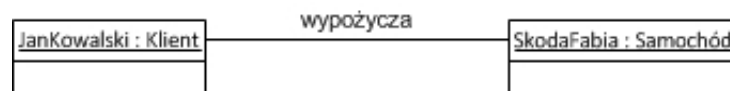


Asocjacja

Asocjacja jest jednym z najsłabszych typów relacji pomiędzy klasami. Najprościej rzecz ujmując oznacza ona, że pomiędzy dwoma klasami jest jakiś rodzaj powiązania. Symbolem asocjacji jest prosta linia. Rysunek poniżej przedstawia przykład asocjacji:



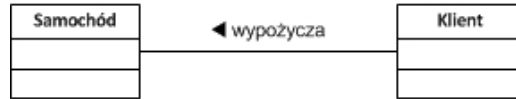
Relację pomiędzy obiektami reprezentującymi powiązane klasy nazywa się łącznikiem (ang. link). Rysunek poniżej przedstawia przykład dwóch połączonych obiektów.



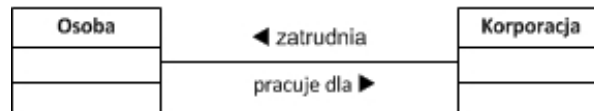
Podsumowując asocjacja wiąże klasy, a łącznik łączy obiekty.

Nazwy asocjacji

Asocjacje mogą/powinny posiadać nazwy (jak na rysunkach powyżej). Nazwa asocjacji powinna być czytana od lewej do prawej lub z góry na dół. Jeśli jednak chcemy by asocjacja była czytana z prawej do lewej lub z dołu do góry, powinniśmy to zaznaczyć wypełnioną strzałką wskazującą kierunek odczytywania relacji.

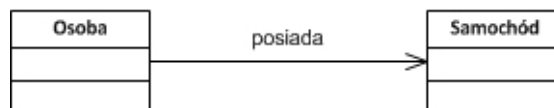


Nazwa asocjacji jest z reguły ustalana z perspektywy jednej z klas. Zdarza się jednak, że asocjacja może mieć dwie nazwy – każdą z perspektywy innej klasy.



Asocjacja skierowana

Prosta linia użyta na diagramie oznacza asocjację dwukierunkową. Jeśli na jednym końcu pojawia się otwarta strzałka, to oznacza to, że jest to asocjacja skierowana, a strzałka wskazuje w którym kierunku ona przebiega. W tym wypadku asocjacja jest odczytywana w kierunku wskazanym przez strzałkę i tak też powinna być nazywana.



Liczebność

Liczebność określa jak dużo instancji (obiektów reprezentujących klasę) danej klasy może być połączonych z obiektami innej klasy. Pozwala to zaznaczyć, że pojedyncza klasa w rzeczywistości reprezentuje zbiór obiektów. Rysunek poniżej prezentuje zastosowanie liczebności w praktyce.



Ważne jest, żeby wiedzieć w jaki sposób czytać symbole liczebności. Liczebności zawsze należy analizować z perspektywy klas powiązanych asocjacją osobno chyba, że mamy do czynienia z relacją skierowaną. Wówczas liczebność jest analizowana tylko w kierunku wskazanym przez strzałkę nawigacyjną.

Symbole użyte na diagramie powyżej należy odczytać w następujący sposób:

1. Instancja klasy Klient musi być połączona z co najmniej jedną instancją klasy Samochód. Innymi słowy jeden klient może teoretycznie wypożyczyć nieskończenie wiele samochodów, ale nie mniej niż jeden.
2. Instancja klasy Samochód może być połączona z dowolną ilością instancji klasy Klient. Innymi słowy samochód może, ale nie musi zostać wypożyczony. Jednej samochód może wypożyczyć wielu klientów.

Jak można to zauważyć na przykładzie powyżej liczebność jest zawsze odczytywana z perspektywy pojedynczej instancji reprezentującej daną klasę.

Jest kilka symboli liczebności, których można używać. Tabela poniżej przedstawia przykładowe symbole. Wartości w nich występujące mogą zostać zastąpione innymi, zgodnymi z wymaganiami konkretnej aplikacji.

Symbol liczebności	Znaczenia
1	Pojedyncza instancja klasy Klient musi być połączona z dokładnie jedną instancją klasy Samochód
*	Pojedyncza instancja może być połączona z dowolną ilością instancji klasy Samochód. Może również nie być połączona z żadną z nich.
0..*	Pojedyncza instancja może być połączona z dowolną ilością instancji klasy Samochód. Może również nie być połączona z żadną z nich. Znaczący to samo co *
0..1	Pojedyncza instancja klasy Klient może być połączona z co najwyżej jedną instancją klasy Samochód, ale może nie być połączona z żadną. Taka sytuacja określana jest mianem liczebności opcjonalnej.
1..*	Pojedyncza instancja klasy Klient musi być połączona z co najmniej jedną instancją klasy Samochód. Nie ma określonej ilości maksymalnej instancji klasy Samochód.
5..9	Pojedyncza instancja klasy Klient może być połączona z 5 do 9 instancji klasy Samochód
3,5,7	Pojedyncza instancja klasy Klient może być połączona z 3, 5 lub 7 instancjami klasy Samochód

W celu ustalenia liczebności kilka kroków musi zostać przeprowadzonych:

1. Utwórz klasy, które chcesz powiązać asocjacją
2. Przeanalizuj charakter związku z perspektywy jednej z klas
3. Umieść symbol liczebności reprezentujący krok 2 we właściwym miejscu na diagramie UML (bliżej drugiej z klas)
4. Powtórz kroki 2 i 3 z perspektywy drugiej klasy (tylko jeśli relacja jest dwukierunkowa)

Procedura ta odzwierciedla sposób w jaki liczebność jest odczytywana. Podsumowując podczas ustalania liczebności należy przeczytać relację z perspektywy obydwu klas stanowiących ją i określić ile instancji może być połączonych w ramach danej relacji.

Często liczebności zależą od wymagań systemu. Różne aplikacje przechowują te same dane w różnym celu. Określa to nie tylko jakie atrybuty klas zostaną wykorzystane, ale również ile instancji może zostać połączonych z pojedynczą instancją danej klasy (wyrażone liczebnością).

Przykładowo prosty system do wypożyczania samochodów może wymagać, aby klient wypożyczył co najmniej jedno samochód. Z kolei aplikacja, która przechowuje informacje o wszystkich klientach musi mieć możliwość pokazania, że dany klient nie ma w danej chwili wypożyczonego żadnego samochodu.

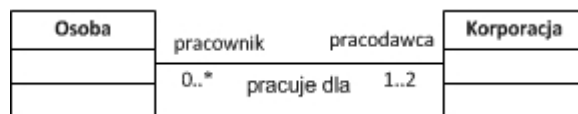
Aby uczynić diagram łatwiejszym do analizy dobrze jest:

1. Umieszczać symbole liczebności pod lub nad linią asocjacyjną blisko klasy
2. Umieszczać symbole liczebności na obu końcach asocjacji
3. Używać liczebności do pokazania jak wiele obiektów na każdym z końców asocjacji może potencjalnie zostać ze sobą powiązanych

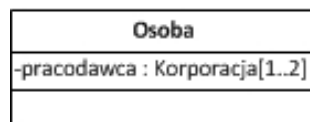
Sprawi to również, że system będzie bardziej zrozumiały i może ograniczyć nieporozumienia związane ze złym odczytaniem diagramu.

Role

Role opisują jak dana klasa zachowuje się w powiązaniu z inną klasą (wyjaśnia naturę związku). Innymi słowy musimy określić jaką rolę instancja danej klasy pełni w konkretnym związku.

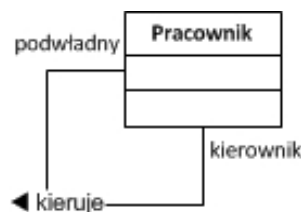


Role mogą być traktowane jako nazwy atrybutów należących do klasy znajdującej się po przeciwnej stronie relacji. Na przykład



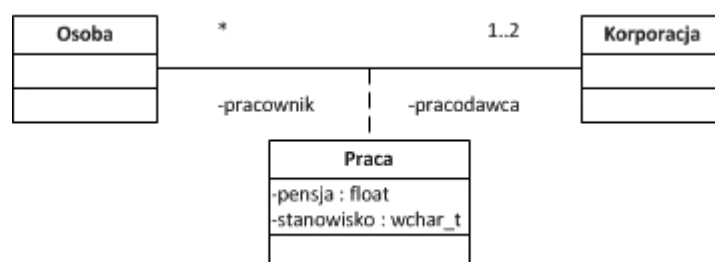
Asocjacja zwrotna

Asocjacja zwrotna to taka asocjacja, która wiąże klasę samą ze sobą. Z tego powodu role są bardzo istotne przy wykorzystywaniu relacji zwrotnych. Pozwalają one na określenie w jakim charakterze występują instancje danej klasy na obu końcach relacji. Przy relacji zwrotnej role powinny być określone na obu jej końcach.



Klasa asocjacji

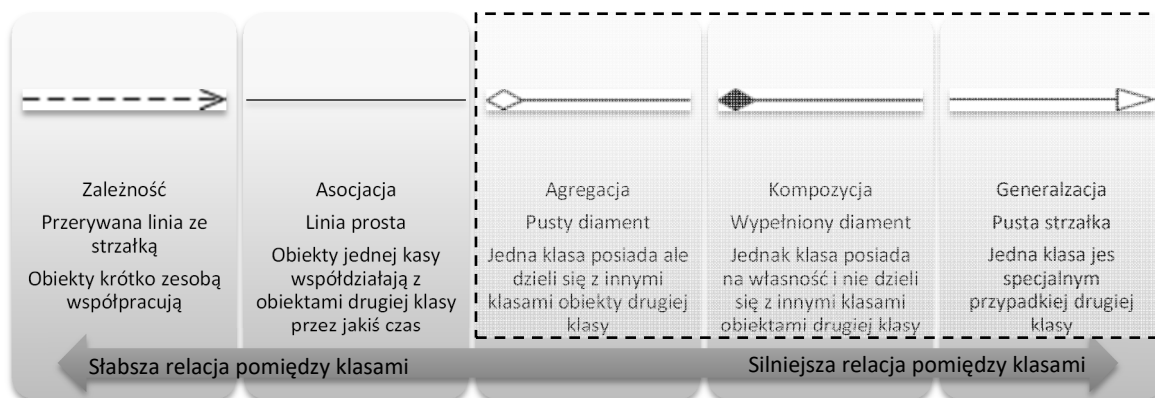
Klasa asocjacji wykorzystywana jest w sytuacji gdy relacja pomiędzy dwoma klasami jest złożona. Wówczas połączenie jest reprezentowane przez dodatkową klasę zwaną klasą asocjacji. Zachowuje się ona jak normalna klasa (ma nazwę oraz atrybuty). Musi jednak być połączona z relacją którą reprezentuje za pomocą przerywanej linii.



Temat 8: Agregacja, Kompozycja i Generalizacja

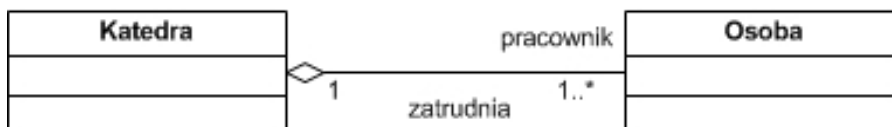
Silniejsze związki pomiędzy klasami.

Poprzedni temat poświęcony był jednej z najsłabszych relacji pomiędzy klasami- asocjacji. Ten temat skoncentrowany jest na agregacji, kompozycji oraz generalizacji. Są to silniejsze powiązania pomiędzy klasami. Oznacza to, że silniej wiążą klasy ze sobą. Pomiędzy klasami zachodzi większa zależność.



Agregacja

Agregacja częściowa jest silniejszą wersją asocjacji. Często jest odczytywana jako relacja typu „zawiera” (ang. has a) co oznacza, że jedna z klas posiada, ale dzieli się z innymi klasą z którą tworzy daną relację. W praktyce oznacza to, że posiadany obiekt może istnieć bez posiadacza, może być połączony relacją z obiektem innej klasy. Jeden obiekt może być współdzielony przez obiekty reprezentujące różne klasy. Implikuje to również, że jeśli posiadacz zostanie zniszczony (usunięty), obiekt posiadany może przetrwać. Symbolem tej relacji jest pusty diament.

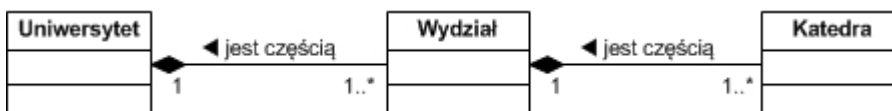


W przykładzie powyżej osoba jako pracownik jest zatrudniona w katedrze. Jeden pracownik pracuje w jednej katedrze, ale jedna katedra zatrudnia wielu pracowników. Osoba może istnieć bez katedry (pracować w innym miejscu, na innym stanowisku).

Kompozycja

Kompozycja (agregacja całkowita) jest jeszcze silniejszą relacją niż agregacja. Może być odczytywana jako relacja typu „posiada” (ang. owns a) co oznacza, że jedna klasa posiada na własność (nie dzieli z innymi klasami) obiekt klasy z którą tworzy daną relację. Jest wykorzystywana do zaprezentowanie relacji typu całość -część. W praktyce oznacza to, że posiadany obiekt (część) nie może istnieć bez posiadacza (całość). Kiedy posiadacz jest niszczone lub usuwany, giną również wszystkie posiadane przez niego części.

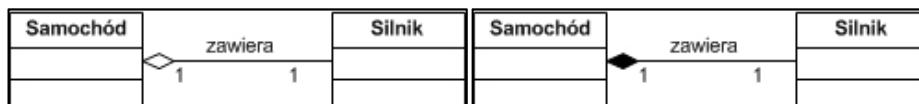
Symbolem tej relacji jest wypełniony diament. Liczebność po stronie całości jest zawsze równa 1 (część należy do jednej całości).



W przykładzie powyżej Katedra jest częścią Wydziału (tylko jednego wydziału), który z kolei jest częścią Uniwersytetu (tylko jednego Uniwersytetu). Na Uniwersytecie może być wiele Wydziałów, a każdy z nich może mieć wiele Katedr. Ani Wydział, ani Katedra nie mogą istnieć bez Uniwersytetu (jeśli zamknięty zostanie Uniwersytet, Wydziały i Katedry również zostaną rozwiązane).

Agregacja a kompozycja

Agregacja i kompozycja są bardzo podobne do siebie. Często trudno określić, którą z tych relacji zastosować w konkretnym przypadku. Najczęściej zależy to od sytuacji i zastosowania diagramu oraz celów realizowanych przez daną aplikację na potrzeby której diagram powstaje



Powyższy przykład przedstawia te same klasy połączone za pomocą agregacji (po lewej) oraz za pomocą kompozycji (po prawej). W pewnych okolicznościach oba diagramy są poprawne.

Jeśli myślimy o konkretnym egzemplarzu samochodu, to ma on jeden silnik zamontowany w nim. Wówczas używamy kompozycji. Kiedy samochód rozbije się, silnik również nie będzie działał zgodnie z jego przeznaczeniem. Jeśli jednak myślimy o projekcie samochodu, to kilka projektów samochodów może dzielić ten sam projekt silnika (np. niektóre modele Skody mają takie same silniki jak niektóre modele Volkswagena). Wówczas użyjemy agregacji. Jeśli zaniechana zostanie produkcja jednego z samochodów, dany model silnika może nadal być wykorzystywany do produkcji innego samochodu.

Generalizacja

Generalizacja to najsilniejszy typ relacji pomiędzy klasami. Jest to relacja typu „jest” (ang. is a). W praktyce oznacza to, że jedna klasa (podklasa) jest traktowana jako specjalny rodzaj innej klasy (nadklasy). Nadklasa jest również nazywana rodzicem lub klasą nadrzędną, a podklasa dzieckiem, potomkiem lub klasą podrzędną. W wielu językach programowania generalizacja nazywana jest dziedziczeniem.

W większości przypadków (w większości języków programowania) jedna klasa może dziedziczyć od tylko jednej innej klasy. Czasami jedna zdarza się, że klasa może dziedziczyć od wielu klas (np. C++). Klasa nadrzędna zawsze może mieć wiele klas podrzędnych.

Na diagramie klas UML generalizacja jest przedstawiana jako pusta w środku zamknięta strzałka. Przy generalizacji nie oznacza się liczebności.

