

Programowanie Obiektowe i C++

Marcin Benke

16 stycznia 2005

Kompilacja i łączenie

- Każdy plik z implementacją kompilujemy do pliku obiektowego z rozszerzeniem `.o` (pod Windows `.obj`), np.

```
g++ -c dtest.cpp
```

```
g++ -c drawing.cpp
```

- Na zakończenie łączymy uzyskane pliki obiektowe w plik wykonywalny, np:

```
g++ -o dtest dtest.o drawing.o
```

- Gotowe biblioteki możemy dołączyć używając opcji `-l`, np

```
g++ -o dtest -lX11 dtest.o drawing.o
```

dołącza bibliotekę zawierającą podstawowe funkcje systemu Xwindow.

- opcja `-L` mówi, gdzie szukać bibliotek

Makefile

Zwykle wygodnie jest użyć programu `make`, opisując proces kompilacji w `Makefile`:

```
all: dtest
```

```
drawing.o: drawing.cpp drawing.h  
    g++ -c drawing.cpp
```

```
dtest.o: dtest.cpp drawing.h  
    g++ -c dtest.cpp
```

```
dtest: dtest.o drawing.o  
    g++ -o dtest -lX11 dtest.o drawing.o
```

Zawartość Makefile

- cele (np `all`)
- zależności między nimi (cel `all` zależy od celu `dtest`)
- czynności, które trzeba wykonać dla osiągnięcia celu

Celem jest zwykle stworzenie pewnego pliku; cel jest osiągnięty jeśli plik ten istnieje i jest nowszy od swoich zależności. Jeśli wywołamy `make` bez argumentów, dążymy do pierwszego celu.

Zmienne

```
LIBS=-lX11
```

```
all: dtest
```

```
drawing.o: drawing.cpp drawing.h
```

```
g++ -c drawing.cpp
```

```
dtest.o: dtest.cpp drawing.h
```

```
g++ -c dtest.cpp
```

```
dtest: dtest.o drawing.o
```

```
g++ -o dtest $(LIBS) dtest.o drawing.o
```

Zmienne specjalne

- `$@` — bieżący cel
- `$<` — pierwsza zależność
- `$^` — wszystkie zależności

np

```
dtest.o: dtest.cpp drawing.h
```

```
    g++ -c $<
```

```
dtest: dtest.o drawing.o
```

```
    g++ -o $@ $(LIBS) $(LDFLAGS) $^
```

Makefile

```
LIBS=-lX11
LDFLAGS=-L/usr/X11R6/lib -g

dtest: dtest.o drawing.o
    g++ -o $@ $(LIBS) $(LDFLAGS) $^
drawing.o: drawing.cpp drawing.h
    g++ -c $<
dtest.o: dtest.cpp drawing.h
    g++ -c $<
clean:
    -rm -f dtest
    -rm -f *.o
```

Makefile

Plik `Makefile` składa się z definicji zmiennych (`CFLAGS`, `LDFLAGS`) oraz reguł.

Reguły mają składnię następującą:

`<cel> : <zależności>`

`<TAB> <polecenia>`

`make` kompiluje tylko te pliki które są nieaktualne, tj. starsze od plików od których zależą.

Korzysta w tym celu z reguł wyspecyfikowanych w pliku, oraz z tzw. reguł domyślnych.

Smalltalk

- Stworzony w latach 70-tych w firmie Xerox.
- Język czysto obiektowy: "wszystko jest obiektem".
- Sterowanie przez wysyłanie komunikatów.
- Pierwszy język dla którego stworzono IDE; prekursor dzisiejszych języków "wizualnych"
- Programy są tworzone przez dodawanie nowych klas do środowiska.
- Stan środowiska razem z kodem programu są zapisywane do pliku obrazu (*image file*)

Smalltalk

- Wszystko jest obiektem
- Przykłady:
 - ▶ 3
 - ▶ 'ala ma kota'
 - ▶ licznik
- Klasa - wzorzec dla obiektów
- Przykłady:
 - ▶ Integer
 - ▶ String
 - ▶ Count
- Każda klasa jest podklasą klasy `Object`

Komunikaty i metody

- Klasa też jest obiektem.
- Nowy obiekt powstaje przez wysłanie komunikatu `new` do klasy, lub jest literałem (3, 'ala ma kota')

```
IntegerCount new
```

- Obiekt odbiera komunikaty i reaguje na nie wykonując akcje zdefiniowane w odpowiednich metodach.

```
2 + 3  
licznik increment
```

Licznik

```
Object subclass: #Count
  instanceVariableNames: 'value resetValue'
  classVariableNames: ''

" Count methodsFor: 'initialize-release'"
initialize
    self reset

" Count methodsFor: 'accessing' "
resetValue
    ^resetValue
resetValue: aValue
    resetValue := aValue.
```

value

^value

value: aValue

value := aValue.

" Count methodsFor: 'counting' "

decrement

" Musi być zaimplementowane w podklasie"

increment

" Musi być zaimplementowane w podklasie"

reset

value := resetValue

Metody klasowe

```
" Count class methodsFor: 'instance-creation' "  
new  
    ^super new initialize
```

Metody

Metoda: funkcja opisująca akcje obiektu

- metoda zeroargumentowa:

`nazwa`

`treść`

`treść` - zmienne tymczasowe i ciąg wyrażeń oddzielonych kropkami

- metoda jednoargumentowa (operator dwuaargumentowy)

`znak_specjalny argument`

`treść`

- metoda wieloargumentowa

`nazwa1: argument1 nazwa2: argument2 ...`

`treść`

Wyrażenia

- **wyrażenie = komunikat**

odbiorca nazwa_komunikatu_z_argumentami
np.

`x + 3`

`x wypisz: ekran`

`Count new`

`kolekcja add: x after: y`

- **wyrażenie przekazujące wynik z metody:**

`^wyrażenie`


```
Count subclass: #IntegerCount
    instanceVariableNames: ''
    classVariableNames: ''

initialize
    resetValue := 0.
    super initialize

decrement
    value := value - 1

increment
    value := value + 1
```

Charakterystyczne cechy Smalltalku

- wszystko jest obiektem (chyba, że jest metodą)
- także klasy są obiektami
- program = biblioteka klas + obiekty
zadeklarowanie nowej klasy oznacza dodanie jej do biblioteki klas — nie ma rozróżnienia pomiędzy klasami “standardowymi” a klasami użytkownika, wszystko można modyfikować
- brak typów zmiennych
wartością zmiennej może być dowolny obiekt. Nie ma żadnej kontroli w czasie kompilacji. Jeśli obiekt nie rozumie komunikatu, jest błąd w czasie wykonania programu

Komunikaty

- unarne (zero parametrów)

```
odbiorca nazwa_komunikatu  
licznik reset  
3 negated  
'abcde' size  
5 factorial
```

- binarne (1 parametr)

```
odbiorca znak_specjalny parametr  
x == 2  
1 = 4  
2 + 3  
'abc' , 'de'
```

Komunikaty

- złożone (keyword, wiele parametrów)

```
odbiorca nazwa1: par1 nazwa2: par2 ...  
tab at: 1 put: 3  
licznik asBase: 8
```

- kaskada (wysłanie wielu komunikatów do tego samego odbiorcy)

```
tab at: 1 put: 'ala';  
      at: 2 put: 'ma';  
      at: 3 put: 'asa'
```

Deklaracja klasy

```
NazwaNadklasy subclass: #NazwaKlasy  
instanceVariableNames: `zmienna1 zmienna2...'  
classVariableNames: `Zmienna1 Zmienna2 ... '  
poolDictionaries: `Pool1 Pool2 ... '
```

- Zmienne egzemplarzowe (indywidualne, obiektowe)
- Zmienne klasowe
 - atrybuty wspólne dla wszystkich obiektów danej klasy
- pule (słowniki)
 - pule zmiennych globalnych, wspólne dla obiektów klas, w których są wymienione
- zmienne globalne (słownik System) wspólne dla wszystkich obiektów

Nie ma innych zmiennych globalnych

Rodzaje metod

- klasowe
 - wykonywane są w klasie, mogą działać na zmiennych globalnych i na zmiennych klasowych
- kategorie
 - ▶ inicjalizacja zmiennych klasowych
 - ▶ tworzenie nowych obiektów
- egzemplarzowe (obektowe)
 - wykonywane w obiektach, mogą działać na zmiennych egzemplarzowych, klasowych i globalnych
- kategorie
 - ▶ dostęp do atrybutów
 - ▶ porównywanie obiektów
 - ▶ kopiowanie
 - ▶ wyświetlanie
 - ▶ inicjalizacja obiektu
 - ▶ pomocnicze (prywatne)

Licznik ASCII

```
Count subclass: #ASCIICount
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

" ASCIICount methodsFor: 'initialize-release' "
initialize
    resetValue := $a.
    super initialize
```

```
" ASCIICount methodsFor: 'counting' "  
decrement
```

```
    | newAsciiValue |  
    newAsciiValue :=  
        value asInteger <= 32  
        ifTrue: [126]  
        ifFalse: [value asInteger - 1].  
    ^self value: (Character value: newAsciiValue)
```

```
increment
```

```
    | newAsciiValue |  
    newAsciiValue :=  
        value asInteger >= 126  
        ifTrue: [32]  
        ifFalse: [value asInteger + 1].  
    ^self value: (Character value: newAsciiValue)
```


Bloki

- Bloki są anonimowymi funkcjami
`[ :arg1 :arg2 | S1. S2. ... Sn ]`
- bloki są obiektami klasy `BlockContext`.
- mogą być przypisywane na zmienne, przekazywane jako argumenty i dawane w wyniku
- w miejscu wystąpienia blok nie jest wykonywany; tworzony jest tylko egzemplarz klasy `BlockContext`.
- Blok może być wykonany przez wysłanie doń komunikatu `value`. Daje to w wyniku wartość ostatniego wyrażenia wykonanego w bloku.

```
[ i:=2+2 ] value
```

```
[ :x :y | x+y*3 ] value: 2 value: 5
```

- Blok może mieć zmienne lokalne:
`[ :arg | | locals | stmts ]`
- Zmienne nielokalne są wiązane statycznie.

“Instrukcje” warunkowe

```
cond ifTrue: aBlock  
cond ifFalse: aBlock  
cond ifTrue: aBlock1 ifFalse: aBlock2  
cond ifFalse: aBlock ifTrue: aBlock2  
cond musi być obiektem klasy True lub False  
self = 0  
  ifTrue: [^self error: 'zero has no reciprocal']  
  ifFalse: [^1 / self]
```

Formalnie `ifTrue:` etc. są zwykłymi komunikatami; zachowują się, jakby klasa `True` miała metody

```
ifTrue: trueBlock ifFalse: falseBlock  
  ^trueBlock value  
ifFalse: aBlock  
  ^nil
```