

**Język do tworzenia diagramów UML**

Kamil Barszczak

Małgorzata Duda

Hanna Jarlaczyńska

**Opis języka**

Celem naszego projektu jest stworzenie prostego języka do opisywania podstawowych diagramów UML. Naszym językiem będzie się dało opisać diagramy klasowe, sekwencji oraz diagramy use-casów. Użytkownik po wpisaniu kodu dostanie wygenerowany diagram UML w postaci grafiki. Planujemy stworzyć język tak, aby:

* dało się stworzyć diagram o dowolnym stopniu skomplikowania.
* było możliwe definiowanie własnych funkcji
* dało się używać pętli oraz list

**Narzędzia**

Do stworzenia tego języka planujemy wykorzystać następujące narzędzia oraz biblioteki:

* ANTLR4
* plantUML
* Python

**Składnia**

1. **Tworzenie diagramów**

diagclass diag\_name

# deklaracja diagramu klasowego

!diagclass diag\_name

diagseq diag\_name

# deklaracja diagramu sekwencji

!diagseq diag\_name

diagusecase diag\_name

# deklaracja diagramu use-casów

!diagusecase diag\_name

1. **Tworzenie obiektów**

*Tworzenie wyglądu który można przypisać do bloków:*

theme theme\_name:

attribute 1

attribute 2

attribute 3

…

*Tworzenie notatki*

note theme\_name note\_name:

note content

…

*Tworzenie pakietu/grupy:*

package theme\_name package\_name:

object 1

object 2

…

Dostęp do poszczególnych pól grupy odbywa się przez operator ‘.’. Przykład: package\_name.attribute1

*Tworzenie połączeń pomiędzy obiektami:*

*Połączenie podstawowe:*

source\_block\_name -> destiny\_block\_name

*Połączenie niestandardowe (w [] nawiasach są możliwe typy strzałek):*

source\_block\_name -> ([x<, <, <<, \, //, \\, o<, \\o ], [., -, \_], [>x, >, >>, \, //, \\, >o, o\\]) destiny\_block\_name labeled “label\_name”

*Połączenie relacji*

source\_class\_name [inherit, implement, associate, depend, aggregate, compose] destiny\_class\_name

* 1. Obiekty specyficzne dla diagramów klasowych

class theme\_name class\_name:

public int value

protected int calc(int x)

private string name

abstract theme\_name class\_name:

public int value

abstract protected int calc(int x)

private string getName()

interface theme\_name interface\_name:

int calc(string source)

* 1. Obiekty specyficzne dla diagramów sekwencji

*Tworzenie/deklaracja bloku:*

block theme\_name block\_name labeled “label\_name”

*Aktywowanie bloku:*

activate block\_name

*Zniszczenie bloku:*

destroy block\_name

*Połączenie bloków:*

Łączenie bloków tak jak innych obiektów. Można dodać opóźnienie w następujący sposób:

source\_block\_name -> … destiny\_block\_name

* 1. Obiekty specyficzne dla diagramów use-casów

*Tworzenie aktora:*

actor theme\_name actor\_name labeled “label\_name”

*Tworzenie use-casów:*

usecase theme\_name usecase\_name:

usecase\_text

…

1. **Pojemnik na obiekty - lista**

*Tworzenie listy:*

list\_name [object1, object2, …]

*Indeksowanie listy:*

list\_name[index] # 0 <= index < list\_size

1. **Pętle**

*Deklaracja pętli:*

for item\_name in list\_name:

function\_name(item\_name)

item\_name -> other\_class labeled “Brrt”

1. **Funkcje**

*Deklarowanie funkcji:*

def function\_name(...):

# do something

return [...]

1. **Komentarze w kodzie**

# <- tak zaczyna się komentarz w kodzie. Tekst komentarza.

1. **Zapisywanie stanu diagramu**

*Obecny stan**diagramu zapisuje się po wywołaniu poniższej komendy*

exec diag\_name [brief, all] [ [list of objects] ] [“filename.png”]

Komenda exec udostępnia kilka opcjonalnych atrybutów tj.:

- rodzaj wypisania diagramu. all - wszystkie informacje. brief - wypisuje tylko konieczne informacje dotyczące diagramu

- lista obiektów która zostanie wypisana

- nazwa pliku do którego zostanie zapisany diagram

**Przykład użycia**

diagclass helloworld

# creating the class object

class hello:

private string hello

public void print()

class world:

private string world

public void print()

# creating the connection

hello -> world labeled “This is the connection”

# now I want to save my diagram

save helloworld

!diagclass helloworld

**Oczekiwany wynik działania programu**