Inne podejścia obiektowe

Referat na seminarium magisterskie Zagadnienia Programowania Obiektowego

Dymitr Pszenicyn

Wprowadzenie

- Obiektowość oparta na prototypowaniu w językach programowania obiektowego
- Są mniej popularne od języków z „typową” obiektowością (klasami i instancjonowaniem)

Języki z prototypowaniem

- ActionScript (oparty na ECMAScript)
- Agora (porzucony projekt ?)
- Amulet (rozszerzenie Garnet'a i przeniesienie do C++)
- Brain (składnia podobna do Smalltalk'a, zapożyczenia z Scheme, Self i JavaScript)
- Cecil/Vortex (bazuje na Self'ie)
- Cel (porzucony projekt ?)

Języki z prototypowaniem (cd.)

- Dialect (podobny do NewtonScript, opracowany dla urządzeń przenośnych – Windows CE)
- ECMAScript (standard dla JavaScript i JScript)
- Garnet (dodanie obiektowości do Common Lisp'a, wspomaga tworzenie GUI)
- GlyphicScript
- Io (oparty na Smalltalku)
- JavaScript (oparty na ECMAScript)
- JScript (oparty na ECMAScript)
- Kevo

Języki z prototypowaniem (cd.)

- Lua (skryptowy język opracowany jako biblioteka w C)
- Merlin (porzucony projekt ?)
- MLud (rozszerzenie Standard ML)
- Moostrap (oparty na Scheme)
- MSmalltalk (mój Smalltalk z prototypowaniem)
- NewtonScript (obiekty posiadają dwa sloty na dziedziczenie: ‘is-a’ i ‘is-in’ po których idzie przeszukiwanie w głąb)

Języki z prototypowaniem (cd.)

- Obliq (dla zorientowanych obiektowo obliczen rozproszonych)
- Omega (oparty na Smalltalk'u)
- OScheme (rozszerzenie Scheme)
- Self (oparty na Smalltalku)
- SK8 (porzucony projekt ?)

ECMAScript (JavaScript/JScript oraz ActionScript)

Historia

- JavaScript (na początku nazywał się LiveScript) został zaprojektowany przez Netscape jako interpretowany język skryptowy do zastosowania na stronach WWW i interpretowany przez przeglądarkę.
- JavaScript pojawił się w przeglądarkach Netscape Navigator 2.0 (JavaScript) oraz Microsoft Internet Explorer 3.0 (JScript)

O języku

- ECMAScript jest standardem powstałym na podstawie języków JavaScript i JScript.
- Najnowsza wersja standardu to wersja 3 z 24 marca 2000 roku.
- ECMAScript używany jest również przez Macromedia Flash (nazywa się ActionScript).

Cechy języka

- Jest to język skryptowy, czyli mający wspomagać, rozszerzać i automatyzować już istniejące środowisko (np. wbudowany do przeglądarki WWW).
- Języka można używać do pisania programów proceduralnych oraz obiektowych.
- Obiektowość oparta na prototypowaniu, brak klas.
- Brak silnego systemu typów.

Cechy języka (cd.)

- Język jest w całości dynamiczny, np. atrybuty mogą być dynamicznie dodawane do obiektów.
- Każdy obiekt ma konstruktor (funkcję za pomocą której został stworzony).
- W funkcjach obiektu mamy dostęp do *this*.
- Możemy dynamicznie przypisywać funkcje do obiektów.

Cechy języka (cd.)

- Możemy otrzymać polimorfizm po prostu tworząc funkcje o identycznych nazwach w różnych obiektach.
- Każdy obiekt ma wskaźnik na prototyp (*prototype*).
- Jeśli metoda albo atrybut nie jest znaleziony w obiekcie, to sprawdzamy w prototypie.
- W ten sposób mamy hierarchie ze względu na prototypy.
- Object jest korzeniem hierarchii.

Wbudowane typy

- Undefined
- Null
- Boolean
- Number
- String

Przykłady wbudowanych obiektów

- Boolean
- Number
- String
- Array
- Image
- Date
- Math

Różnica pomiędzy typami wbudowanymi a obiektami

```
var pimitiveString1 = "This is a primitive string";
var pimitiveString2 = String("This is a primitive string");
var stringObject = new String("This is a String object");

primitiveString1.prop = "This is a property";
primitiveString2.prop = "This is a property";
stringObject.prop = "This is a property";

alert(primitiveString1.prop) // displays "undefined"
alert(primitiveString2.prop) // displays "undefined"
alert(stringObject.prop) // displays "This is a property"

alert(typeof primitiveString1); // displays "string"
alert(typeof primitiveString2); // displays "string"
alert(typeof stringObject) // displays "object"
```

Funkcje

- Funkcje są jednocześnie konstruktorami obiektów.
- Wewnątrz funkcji możemy odwołać się do atrybutu *arguments*:

```
function testArg(){
 for(i=0;i<arguments.length;i++){
   alert("Argument "+i+" is "+arguments[i]);
 }
}
```

Przykładowy fragment programu

```
function Circle(radius){
 this.radius = radius;
 this.getArea = function(){
   return (this.radius*this.radius*3.14);
 }
 this.getCircumference = function(){
   var diameter = this.radius*2;
   var circumference = diameter*3.14;
   return circumference;
 }
}

var bigCircle = new Circle(100);
var smallCircle = new Circle(2);

alert(bigCircle.getArea()); // displays 31400
alert(smallCircle.getCircumference()); // displays 12.56
```

Funkcje (cd.)

- Atrybut *constructor* zwraca funkcję za pomocą której stworzono ten obiekt.
- Można łatwo zmienić zachowanie obiektów (nawet wbudowanych) np.

```
String.prototype.addDot=function (){
   return this+”.”;
 }
var s = ”Test string”;
alert(s.addDot());
```

- Ustawiając atrybut *prototype* możemy łatwo odziedziczyć atrybuty i metody.

Przykład dziedziczenia z użyciem prototypu

```
function Shape(color){
 this.x = 4;
 this.y = 4;
 this.color = color;
 this.move = function(dx, dy){
   this.x+=dx;
   this.y+=dy;
 }
}
function Circle(color){
 this.r = 2;
 this.color = color;
}
Circle.prototype = new Shape();
function Square(color){
 this.a = 2;
 this.color = color;
}
Square.prototype = new Shape();
var myCircle = new Circle("black");
var mySquare = new Square("red");
```

Podsumowanie

- Mechanizm obiektowości w ECMAScript dobrze pasuje do języka skryptowego, który zarządza różnymi obiektami graficznymi (np. na stronie WWW), gdyż łatwo jest dodać lub zmienić atrybuty obiektów (również tworzonych na podstawie tego obiektu).

MSmalltalk

Język z prototypowaniem oparty na Smallalku

Cechy języka

- MSmalltalk jest czysto obiektowym językiem programowania w którym nowe obiekty są tworzone na podstawie już istniejących (prototypowanie).
- Klasy istnieją w tym sensie, że obiekty należące do tej samej „klasy” mają te same metody i takie same zmienne obiektowe. W ten sposób można mowić o dziedziczeniu po pewnej klasie.
- Hierarchia klas jest drzewem z wyróżnionym korzeniem, więc każda klasa może dziedziczyć co najwyżej po jednej klasie.
- Z powodu prototypowania, metody (i zmienne) klasowe są metodami (i zmiennymi) w obiekcie klasy – czyli są jednocześnie metodami (i zmiennymi) we wszystkich obiektach danej klasy.

Cechy języka (cd.)

- Metody *(#def_method* i *#def_class*) do tworzenia metod i klas są częścią języka.
- W języku występuje zmienna *#self*, która oznacza ten właśnie obiekt.
- W języku występuje konstrukcja *#super*, która oznacza wywołanie metody z nadklasy (w klasie *Object* to jest *Object*).
- Zmienne obiektowe są deklarowane (np. | *a b c* |) w momencie definiowania nowej klasy.
- Deklaracja zmiennych lokalnych (np. | *x y z* |) może wystąpić tylko na początku definicji metody.

Cechy języka (cd.)

- Zmienne globalne muszą być zapisywane z dużej litery. Używa się ich bez deklarowania.
- Są trzy rodzaje metod (w kolejności od najmocniej wiążących): metody bezargumentowe, operatory dwuargumentowe (+, -, , /, =, <, itd.), metody wieloargumentowe.

Cechy języka (cd.)

- W przypadku metod wieloargumentowych, w odróżnieniu od Smalltalk'a, argumenty nie są „nazywane" i nazwa metody nie powstaje z wielu części oddzielonych dwukropkami.
- W momencie deklaracji metody, przed każdym argumentem trzeba dodać dwukropek.
- W momencie wywołania metody, po nazwie metody należy umieścić dwukropek.
- W języku jest konstrukcja (średnik) pozwalająca na wysłanie kilku komunikatów (wywołania metod) do jednego obiektu (sekwencyjnie).
- Argumenty w definicji bloku powinny być poprzedzone dwukropkiem.

Cechy języka (cd.)

- Bloki instrukcji mogą mieć do trzech argumentów (włącznie).
- W programie można używać zagnieżdżonych komentarzy rozpoczynających się od „(*”, a kończących się „*)”.
- Istnieje klasa *Object* po to by hierarchia klas miała korzeń i żeby mieć metody takie jak *new.*

Przykład programu

- Program ten nie wykorzystuje żadnych cech specyficznych dla prototypowania.
- W programie tworzona jest klasa *Example* której obiekty mają zmienną *liczba*. Następnie jest tworzona podklasa *MyExample* która ma dodatkowo zmienną *inna_liczba*. Niektóre metody podklasy korzystają z metod nadklasy. Na końcu programu tworzony jest obiekt klasy *MyExample* i jest przypisywany na zmienną globalną *MExamp*. W obiekcie tym są wołane metody zmieniające zmienne tego obiektu, a następnie metoda wypisująca wartości zmiennych.

Przykład programu (cd.)

```
Object #def_class Example | liczba |.
Example #def_method new {
        ^(#super new) set_liczba: 123 ; yourself
}.
Example #def_method set_liczba :licz {
        liczba := licz
}.
Example #def_method get_liczba {
        ^ liczba
}.
Example #def_method add_liczba :licz {
        liczba := liczba + licz
}.
Example #def_method show {
        ("Liczba = " , (liczba toString) , "\n") print
}.
```

Przykład programu (cd.)

```
Example #def_class MyExample | inna_liczba |.
MyExample #def_method new {
        ^(#super new) initialize ; yourself
}.
MyExample #def_method initialize {
        inna_liczba := 456
}.
MyExample #def_method zamien_liczby | tmp | {
        tmp := inna_liczba.
        inna_liczba := liczba.
        liczba := tmp
}.
MyExample #def_method show {
        #super show.
        ("Inna_Liczba = " , (inna_liczba toString) , "\n") print.
        (liczba < inna_liczba) ifTrue:
        [ ((liczba toString) , " < " ,
                        (inna_liczba toString) , "\n") print
        ]
}.
```

Przykład programu (cd.)

```
MyExample #def_method show {
        #super show.
        ("Inna_Liczba = " , (inna_liczba toString) , "\n") print.
        (liczba < inna_liczba) ifTrue:
        [ ((liczba toString) , " < " ,
                (inna_liczba toString) , "\n") print
        ]
}.
MExamp := MyExample new.
MExamp zamien_liczby; add_liczba: 111; zamien_liczby; show
```

- Po uruchomieniu, ten program wypisze:

```
Liczba = 123
Inna_Liczba = 567
123 < 567
```

Przykład programu

- Program ten wykorzystuje i prezentuje wlasności języka związane z prototypowaniem.
- W programie tworzona jest klasa (obiekt) *Test1* ze zmienna *zm_obj1* oraz metodami dostępu do tej zmiennej *get_zm1* oraz *set_zm1*. Następnie zmienna jest ustawiana na *„Test1”*. Tworzona jest także klasa (obiekt) *Test2* z dodatkową zmienną *zm_obj2*. Następnie w obiekcie *Test1* zmienna *zm_obj1* jest ustawiana na *„Test1_2”*. Tworzone są również nowe obiekty na podstawie obiektów *Test1*, *Test2* oraz innych obiektów. Wartości zmiennych sa wypisywane.

Przykład programu (cd.)

```
Object #def_class Test1 | zm_obj1 |.
Test1 #def_method get_zm1 {
        ^ zm_obj1
}.
Test1 #def_method set_zm1 :x {
        ^ zm_obj1 := x
}.
Test1 set_zm1: "Test1".

Test1 #def_class Test2 | zm_obj2 |.
Test2 #def_method get_zm2 {
        ^ zm_obj2
}.
```

Przykład programu (cd.)

```
Test2 #def_method set_zm2 :x {
        ^ zm_obj2 := x
}.
Test1 set_zm1: "Test1_2".

"Test1 get_zm1 :: " print.
(Test1 get_zm1) print. "\n" print.

"Test2 get_zm1 :: " print.
(Test2 get_zm1) print. "\n" print.

"Glob1 := Test1 new\n" print.
Glob1 := Test1 new.
```

Przykład programu (cd.)

```
"Glob1 get_zm1 :: " print.
(Glob1 get_zm1) print. "\n" print.

"Glob1 set_zm1: TTTTT1\n" print.
Glob1 set_zm1: "TTTTT1".
"Glob1 get_zm1 :: " print.
(Glob1 get_zm1) print. "\n" print.

"Glob1_2 := Glob1 new\n" print.
Glob1_2 := Glob1 new.
"Glob1_2 get_zm1 :: " print.
(Glob1_2 get_zm1) print. "\n" print.
```

Przykład programu (cd.)

```
"Glob2 := Test2 new\n" print.
Glob2 := Test2 new.
"Glob2_2 := Glob2 new\n" print.
Glob2_2 := Glob2 new.
"Glob2_2 get_zm1 :: " print.
(Glob2_2 get_zm1) print. "\n" print
```

Przykład programu (cd.)

- Po uruchomieniu, ten program wypisze:
  Test1 get_zm1 :: Test1_2
  Test2 get_zm1 :: Test1
  Glob1 := Test1 new
  Glob1 get_zm1 :: Test1_2
  Glob1 set_zm1: TTTTT1
  Glob1 get_zm1 :: TTTTT1
  Glob1_2 := Glob1 new
  Glob1_2 get_zm1 :: TTTTT1
  Glob2 := Test2 new
  Glob2_2 := Glob2 new
  Glob2_2 get_zm1 :: Test1

Podsumowanie

- Języka można używać nawet nie wiedząc że jest oparty na prototypowaniu.
- Jeśli wiemy że jest on oparty na prototypowaniu to możemy tworzyć obiekty na podstawie dowolnych innych oraz przypisać wartości na zmienne obiektowe w obiekcie który używamy do stworzenia innego żeby uzyskać wartości początkowe w nowym obiekcie.

Różnice w obiektowości pomiędzy ECMAScript a MSmalltalk

- W ECMAScript można dynamicznie dodawać atrybuty do obiektów.
- W ECMAScript można zmienić prototyp już istniejącego obiektu.
- MSmalltalk bardziej przypomina zwyczajną obiektowość z klasami.
- W MSmalltalk'u zmienne obiektowe nie są widziane na zewnątrz obiektów.

Bibliografia

- Strona poświęcona językom programowania obiektowego z prototypowaniem: http://www.dekorte.com/Proto/Chart.html
- Inna strona o tej samej tematyce:

  http://www.pasteur.fr/~letondal/object-based.html

Bibliografia (cd.)

- Strona w Google Directory poświęcona językom z prototypowaniem

  http://directory.google.com/Top/Computers/Programming/Languages/Object-Oriented/Prototype-based/
- Strona domowa języka Brain

  http://brain.sourceforge.net/

Bibliografia (cd.)

- Specyfikacja języka ECMAScript ver. 3
  http://www.mozilla.org/js/language/E262-3.pdf
- Implementacje języka JavaScript
  http://www.mozilla.org/js/

Bibliografia (cd.)

- Strona z wprowadzeniem do obiektowości w JavaScripcie

  http://www.webmasterbase.com/article/470
- Druga część tego artykułu

  http://www.sitepoint.com/article/oriented-programming-2/1

Bibliografia (cd.)

- Specyfikacja języka MSmalltalk (z formalną kontynuacyjną semantyką denotacyjną):

  Dymitr Pszenicyn „Opis języka MSmalltalk ver. 1.6.5”

  http://rainbow.mimuw.edu.pl/~dp189434/sml/msmallt.pdf
- Kod interpretera języka MSmalltalk napisanego w Moscow ML’u.