Zadanie na pracownię nr 14

Uwaga: Termin oddania rozwiązań tego zadania to wtorek, 12 czerwca 2018, godz 6:00.

Programowanie obiektowe

Na wykładzie numer 14 zobaczyliśmy, jak można zrealizować proste modyfikowalne obiekty przy użyciu mutowalnego stanu w Rackecie. Podejście to można rozszerzyć, aby wprowadzić możliwość dziedziczenia – definiowania nowych rodzajów obiektów, które rozszerzają istniejące rodzaje obiektów o nowe zachowania.

Obiektem będziemy nazywać procedurę która, gdy otrzyma symbol jako argument, zwróci inną procedurę, nazywaną *metodą*.

```
(define (get-method object message)
  (object message))
```

Przykładowo, poniższa procedura tworzy prosty obiekt – mówcę, który może powiedzieć (say) coś (jakąś listę), co robi przez wypisanie tej listy:

```
(define (make-speaker)
  (define (self message)
    (cond [(eq? message 'say)
           (lambda (stuff) (display stuff))]
          [else (no-method "SPEAKER")]))
  self)
```

Nazwy `self` będziemy używać, aby w obiekcie móc odwołać się do samego siebie. Wprowadzamy konwencję, aby obiekt, w przypadku nie rozpoznania wiadomości, zakomunikował to przy użyciu `no-method`:

```
(define (no-method name)
  (list 'no-method name))

(define (no-method? x)
  (if (pair? x)
      (eq? (car x) 'no-method)
      false))

(define (method? x)
  (not (no-method? x)))
```

Aby poprosić obiekt, aby uruchomić jedną z jego metod z pewnymi argumentami, wysyłamy mu komunikat, po czym aplikujemy otrzymaną metodę do argumentów.

```
(define (ask object message . args)
  (let [(method (get-method object message))]
    (if (method? method)
        (apply method args)
        (error "Brak metody" message (cadr method)))))
```

Teraz obiekt utworzony przez `make-speaker` może wreszcie coś powiedzieć:

```
(define george (make-speaker))
(ask george 'say '(racket jest ok))
(racket jest ok)
```

Możemy chcieć stworzyć rodzaj obiektów, który *rozszerza* inny rodzaj obiektów. Na przykład, możemy uznać, że wykładowca jest rodzajem mówcy, ale który potrafi nie tylko mówić, ale też wykładać:

```
(define (make-lecturer)
  (let [(speaker (make-speaker))]
    (define (self message)
      (cond [(eq? message 'lecture)
             (lambda (stuff)
               (ask self 'say stuff)
               (ask self 'say '(powinniscie notowac)))]
            [else (get-method speaker message)]))
    self))
```

Zwróć uwagę, że obiekt wykładowcy posiada wewnątrz obiekt mówcy, który realizuje metody, których obiekt wykładowcy sam nie posiada.

```
(define mpi (make-lecturer))
(ask mpi 'say '(racket jest ok))
(racket jest ok)

(ask mpi 'lecture '(racket jest ok))
(racket jest ok)
(powinniscie notowac)
```

Następny przykład pokaże mały problem występujący w naszej prostej implementacji.

Zdefiniujmy aroganckiego wykadowcę, który będzie rodzajem wykładowcy. Arogancki wykładowca, cokolwiek nie powie, poprzedzi to sformułowaniem „To oczywiste że...”:

```
(define (make-arrogant-lecturer)
  (let [(lecturer (make-lecturer))]
    (define (self message)
```

```
      (cond [(eq? message 'say)
             (lambda (stuff)
               (ask lecturer 'say (append '(to oczywiste ze) stuff)))]
            [else (get-method lecturer message)]))
    self))

(define mma (make-arrogant-lecturer))

(ask mma 'say '(racket jest ok))
(to oczywiste ze racket jest ok)
```

Nowo zdefiniowany wykładowca przestaje być arogancki, gdy poprosimy go, aby wykładał:

```
(ask mma 'lecture '(racket jest ok))
(racket jest ok)
(powinniscie notowac)
```

Problem polega na tym, że gdy obiekt aroganckiego wykładowcy mówi wewnętrznie zdefiniowanemu obiektowi wykładowcy, aby wykładał, informacja o tym, że robi to arogancki wykładowca, jest tracona. Obiekt wykładowcy wywoła metodę say mówcy, a nie aroganckiego wykładowcy. (Podobne zachowanie występuje w języku C++ dla metod, które nie są wirtualne.)

Aby poprawić ten problem, będziemy od teraz tak pisać metody, żeby otrzymywały obiekt, na którym operują, jako parametr (a nie, jak dotychczasowo, pobierały go z kontekstu leksykalnego). Przykładowo, definicja konstruktora obiektu-mówcy będzie teraz wyglądać tak:

```
(define (make-speaker)
  (define (self message)
    (cond [(eq? message 'say)
           (lambda (self stuff) (display stuff))]
          [else (no-method message)]))
  self)
```

Musimy jeszcze zmodyfikować definicję procedury ask:

```
(define (ask object message . args)
  (let [(method (get-method object message))]
    (if (method? method)
        (apply method (cons object args))
        (error "Brak metody" message (cadr method)))))
```

Po poprawieniu pozostałych definicji konstruktorów problem jest poprawiony:

```
(ask mma 'lecture '(racket jest ok))
(to oczywiste ze racket jest ok)
(to oczywiste ze powinniscie notowac)
```

Zadanie: gra przygodowa

W pliku `game.rkt` znajduje się (niekompletny) kod prostej gry przygodowej, zrealizowany przy użyciu systemu obiektów opisanego powyżej. Dodaj nowe rodzaje przedmiotów lub postaci oraz rozszerz świat gry według własnego uznania. Rozbuduj też zestaw poleceń dla gracza (na przykład o polecenia związane z obsługą inwentarza).