MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.037—Structure and Interpretation of Computer Programs
IAP 2019

Object-Oriented Programming

Solutions

Problems

1. Write a `food` class

   - Input state is the `name`,`nutrition` value, and `good-until` time.
   - Additional state is the `age` of the food, initially 0.
   - Methods are:
     - `NAME` - returns the name of the food
     - `AGE` - returns the age of the food
     - `SIT-THERE` - takes an amount of time, and increases the age of the food by the amount.
     - `EAT` - return the nutrition if the food is still good; 0 otherwise.

```
(define food
  (make-class
   'food
   '(name nutrition good-until age)
   root-object
   (make-methods
      'CONSTRUCTOR
      (lambda (name nutrition good-until)
        (write-state! 'name name)
        (write-state! 'nutrition nutrition)
        (write-state! 'good-until good-until)
        (write-state! 'age 0))
      'NAME
      (lambda () (read-state 'name))
      'AGE
      (lambda () (read-state 'age))
      'SIT-THERE
      (lambda (time)
        (write-state! 'age (+ time (read-state 'age))))
      'EAT
      (lambda ()
        (if (< (read-state 'age) (read-state 'good-until))
            (read-state 'nutrition)
            0)))))
```

2. Write an `aged-food` class

- Input state is the same as the `food` class, with an additional parameter, which is the `good-after` time.
- Should inherit from the `food` class.
- Methods are:
  - `SNIFF` - returns `#t` if it has aged enough to be good.
  - `EAT` - returns 0 if the food is not good yet; otherwise behaves like normal food.

```
(define aged-food
  (make-class
   'aged-food
   '(good-after)
   food
   (make-methods
      'CONSTRUCTOR
      (lambda (name nutrition good-until good-after)
        (super 'CONSTRUCTOR name nutrition good-until)
        (write-state! 'good-after good-after))
      'SNIFF
      (lambda ()
        (> (invoke self 'AGE) (read-state 'good-after)))
      'EAT
      (lambda ()
        (if (invoke self 'SNIFF)
            (super 'EAT)
            0)))))
```

3. Extend the object system to support dynamic mixin classes. A "mixin" is when one class, after being defined, can be modified to include methods definitions from some other class[1]. This effectively allows a class to inherit from multiple classes, and is also sometimes called a role or an abstract base class. [2]

```
(define (mixin! from to)
  (cond ((not (class? from))
         (error "first arg to mixin! must be a class"))
        ((class? to)
         (set-car! (cddddr to)
                   (append (class-methods to)
                           (class-methods from))))
        (error "second arg to mixin! must be a class")))
```

[1] Technically, since this is only adding the methods of the other class, and not its state, this is a "trait" and not a mixin

[2] Mixins actually first appeared in an object system for Lisp Machine Lisp in 1982; the name was inspired by Steve's Ice Cream Parlor in Somerville, which allowed toppings to be mixed into their ice cream.

4. Further extend the system to support mixins on *instances*, in addition to classes. That is, some particular instance of `aged-food` (a `stinky-cheese-wheel`, for instance) might mix in the methods of the `round` trait to get the `ROLL` method.

```
(define (mixin! from to)
  (cond ((not (class? from))
         (error "mixin! takes a class as the first arg"))
        ((class? to)
         (set-car! (cddr to)
                   (append (class-state to)
                           (class-state from)))
         (set-car! (cddddr to)
                   (append (class-methods to)
                           (class-methods from))))
        ((instance? to)                              ;; ADDED
         (set-car! (cddr to)
                   (make-class (gensym)
                               '()
                               (instance-class to)
                               (class-methods from))))
        (else
         (error "unknown second type to mixin!"))))
```

...or:

```
(define (instance-methods inst)                      ;; ADDED
  (fourth inst))

(define (make-instance class . args)
  (let ((inst
         (list 'instance
               (map (lambda (x) (list x #f))
                    (collect-state class))
               class
               '())))                                ;; ADDED
    (if (has-method? inst 'CONSTRUCTOR)
        (apply invoke inst 'CONSTRUCTOR args)
        (void))
    inst))

(define (find-instance-method methodname instance)   ;; ADDED
  (let ((result (assq methodname (instance-methods instance))))
    (if result
        (second result)
        #f)))

(define (has-method? instance method)
  (define (helper class)
    (cond ((not (class? class))
           #f)
```

```
          ((find-class-method method class)
           #t)
          (else (helper (class-parent class)))))
    (or (find-instance-method method instance)         ;; ADDED
        (helper (instance-class instance))))

(define (invoke instance method . args)
  (fluid-let ((self instance))
    (let ((proc (find-instance-method method self))) ;; ADDED
      (if proc
          (fluid-let ((super (lambda (method . args)
                               (method-call
                                (instance-class instance)
                                method args))))
            (apply proc args))
          (method-call (instance-class instance) method args)))))

(define (mixin! from to)
  (cond ((not (class? from))
         (error "first arg to mixin! must be a class"))
        ((class? to)
         (set-car! (cddddr to)
                   (append (class-methods to)
                           (class-methods from))))
        ((instance? to)                                ;; ADDED
         (set-car! (cdddr to)
                   (append (instance-methods to)
                           (class-methods from))))
        (else
         (error "unknown second arg to mixin!"))))
```