MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001 Structure and Interpretation of Computer Programs
Spring, 2007

Recitation 16, April 13

Object-Oriented Adventures Dr. Kimberle Koile

From Project 4 Description: An Object System

Consider the problem of simulating the activity of a few interacting agents wandering around different places in a simple world. Real people are very complicated; we do not know enough to simulate their behavior in any detail. But for some purposes (for example, to make an adventure game) we may simplify and abstract this behavior. In particular, we can use objects to capture common state parameters and behaviors of things, and can then use the message-passing paradigm to control interaction between objects in a simulation.

Let's start with the fundamental ideas first. We can think of our object oriented paradigm as consisting of classes and instances. A class can be thought of as the "template" for how we want a particular kind of object to behave. The way we define the class of an object is with a basic "make handler" procedure; this procedure is used with a "create instance" procedure which builds for us a particular instance. As you will see, when we examine the code, each class is defined by a procedure that when invoked will create some internal state (including instances of other class objects) and a message passing procedure (created by a “make handler”) that returns methods in response to messages.

Our object instances are thus procedures that accept messages. An object will give you a method if you send it a message; you can then invoke that method on the object (and possibly some arguments) to cause some action, state update, or other computation to occur.

The main pieces we will use in our code to capture these ideas are detailed as follows:

Instance of an object – each individual object has its own identity. An instance knows its type, and has a message handler associated with it. One can “ask” an object to do something, which causes object's message handler to look for a method to handle the request and then invoke the method on the arguments to ask.

“Making” an object message handler – each instance needs a new message handler to inherit the state information and methods of the specified class. The message handler is not a full “object instance” in our system; the message handler needs to be part of an instance object (or part of another message handler that is part of an instance object). All procedures that define classes should take a self pointer (a pointer to the enclosing instance) as the first argument.

“Creating” an object – the act of creation does three things: it makes a new instance of the object; it makes and sets the message handler for that instance; and finally it INSTALLs that new object into the world.

“Installing” an object – this is a method in the object, by which the object can initialize itself and insert itself into the world, by connecting itself up with other related objects in the world.

Examples (simplified by deleting code from Project 4 definitions)

create-instance: make an instance, make its message handler, send instance INSTALL message. The message handler for each object type automatically provides methods for 'IS-A, 'TYPE, and 'METHODS.

```
(define (create-instance maker . args)
 (let* ((instance (make-instance))                  ; create the instance object (procedure)
        (handler (apply maker instance args)))      ; make the message handler procedure
  (set-instance-handler! instance handler)
  (if (method? (get-method 'INSTALL instance))
     (ask instance 'INSTALL))                        ; an initialization method
  instance))
```

named-object

```
(define (create-named-object name)     ; symbol -> named-object
 (create-instance named-object name))
```

```
(define (named-object self name)
  (let ((root-part (root-object self)))          ; note superclass
    (make-handler                                 ; note make-handler
     'named-object
     (make-methods                                ; note methods
      'NAME    (lambda () name)
      'INSTALL (lambda () 'installed))
     root-part)))
```

thing: a named-object with a location

```
 (define (create-thing name location)    ; symbol, location -> thing
  (create-instance thing name location))

(define (thing self name location)
  (let ((named-part (named-object self name)))
    (make-handler
     'thing
     (make-methods
      'INSTALL  (lambda ()
                  (ask named-part 'INSTALL)                    ; note  "appending" install methods; note ask
                  (ask (ask self 'LOCATION) 'ADD-THING self))    ; note self
      'LOCATION (lambda () location))
     named-part)))
```

mobile-thing: thing with location that can change

```
(define (mobile-thing self name location)
  (let ((thing-part (thing self name location)))
    (make-handler
     'mobile-thing
     (make-methods
      'LOCATION  (lambda () location)
      'CHANGE-LOCATION
      (lambda (new-location)
            (ask location 'DEL-THING self)
            (ask new-location 'ADD-THING self)
            (set! location new-location))
      'ENTER-ROOM    (lambda () #t)
      'LEAVE-ROOM    (lambda () #t))
     thing-part)))
```

container: holds things. This class is not meant for stand-alone objects, so no create-container procedure. Other classes will inherit from this class.

```
(define (container self)
  (let ((root-part (root-object self))
         (things '()))                    ; note local variable
    (make-handler
     'container
     (make-methods
      'THINGS      (lambda () things)
      'HAVE-THING? (lambda (thing)
                     (not (null? (memq thing things))))
      'ADD-THING   (lambda (thing)
                     (if (not (ask self 'HAVE-THING? thing))
                          (set! things (cons thing things)))
                     'DONE)
      'DEL-THING   (lambda (thing)
                     (set! things (delq thing things))
                     'DONE))
     root-part)))
```

place: a container (so things may be in the place); has exits,
which are passages from one place to another.

```
(define (create-place name)     ; symbol -> place
  (create-instance place name))

(define (place self name)
  (let ((named-part (named-object self name))
        (container-part (container self))           ; note multiple inheritance
        (exits '()))
    (make-handler
     'place
     (make-methods
      'EXITS (lambda () exits)
      'EXIT-TOWARDS
      (lambda (direction)
        (find-exit-in-direction exits direction))
      'ADD-EXIT
      (lambda (exit)
        (let ((direction (ask exit 'DIRECTION)))
          (if (ask self 'EXIT-TOWARDS direction)
              (error (list name "already has exit" direction))
              (set! exits (cons exit exits)))
          'DONE)))
     container-part named-part)))
```

person

```
(define (create-person name birthplace) ; symbol, place -> person
  (create-instance person name birthplace))

(define (person self name birthplace)
  (let ((mobile-thing-part (mobile-thing self name birthplace))
        (container-part    (container self)))
    (make-handler
     'person
     (make-methods
      'SAY
      (lambda (list-of-stuff)
        (ask screen 'TELL-ROOM (ask self 'location)
             (append (list "At" (ask (ask self 'LOCATION) 'NAME)
                           (ask self 'NAME) "says --")
                     list-of-stuff))  'SAID-AND-HEARD)
      'PEOPLE-AROUND        ; other people in room...
      (lambda ()
        (delq self (find-all (ask self 'LOCATION) 'PERSON)))
      'STUFF-AROUND         ; stuff (non people) in room...
      (lambda ()
        (let* ((in-room (ask (ask self 'LOCATION) 'THINGS))
               (stuff (filter (lambda (x) (not (ask x 'IS-A 'PERSON))) in-room)))
          stuff))
      'PEEK-AROUND          ; other people's stuff...
      (lambda ()
        (let ((people (ask self 'PEOPLE-AROUND)))
          (fold-right append '() (map (lambda (p) (ask p 'THINGS)) people))))
      'TAKE
      (lambda (thing)
        (cond ((ask self 'HAVE-THING? thing)  ; already have it
               #t
               #f)
              ((or (ask thing 'IS-A 'PERSON)
                   (not (ask thing 'IS-A 'MOBILE-THING)))
               #f)
               (else
                (let ((owner (ask thing 'LOCATION)))
                       (if (ask owner 'IS-A 'PERSON)                    ; note IS-A
                           (ask owner 'LOSE thing self)
                           (ask thing 'CHANGE-LOCATION self))
                       thing))))
      'LOSE
      (lambda (thing lose-to)
              (ask thing 'CHANGE-LOCATION lose-to))
      'GO-EXIT
      (lambda (exit)
              (ask exit 'USE self))
      'ENTER-ROOM                                          ; note overwriting inherited method
      (lambda ()
              (let ((others (ask self 'PEOPLE-AROUND)))
                (if (not (null? others))
                    (ask self 'SAY (cons "Hi" (names-of others)))))
              #t))
     mobile-thing-part container-part)))
```

autonomous-person: activity determines max movement; miserly determines chance of picking stuff up

```
(define (create-autonomous-person name birthplace activity miserly)
  (create-instance autonomous-person name birthplace activity miserly))

(define (autonomous-person self name birthplace activity miserly)
  (let ((person-part (person self name birthplace)))
    (make-handler
     'autonomous-person
     (make-methods
      'INSTALL
      (lambda ()
              (ask person-part 'INSTALL)
              (ask clock 'ADD-CALLBACK                        ; note 'ADD-CALLBACK
                   (create-clock-callback 'move-and-take-stuff self
                                          'MOVE-AND-TAKE-STUFF)))
      'MOVE-AND-TAKE-STUFF
      (lambda ()
              ;; first move
              (let loop ((moves (random-number activity)))
                (if (= moves 0)
                    'done-moving
                    (begin
                            (ask self 'MOVE-SOMEWHERE)
                            (loop (- moves 1)))))
              ;; then take stuff
              (if (= (random miserly) 0)
                  (ask self 'TAKE-SOMETHING))
              'done-for-this-tick)
      'MOVE-SOMEWHERE
      (lambda ()
              (let ((exit (random-exit (ask self 'LOCATION))))
                (if (not (null? exit)) (ask self 'GO-EXIT exit))))
      'TAKE-SOMETHING
      (lambda ()
              (let* ((stuff-in-room (ask self 'STUFF-AROUND))
                     (other-peoples-stuff (ask self 'PEEK-AROUND))
                     (pick-from (append stuff-in-room other-peoples-stuff)))
                (if (not (null? pick-from))
                    (ask self 'TAKE (pick-random pick-from))
                    #f))))
     person-part)))
```

clock: keeps track of time; used to initiate actions by means of callbacks.
Callbacks are invoked with each tick of the clock.

```
(define (clock self . args)
  (let ((root-part (root-object self))
        (name (if (not (null? args))
                  (car args)  'THE-CLOCK))
        (the-time 0)
        (callbacks '())
        (removed-callbacks '()))
    (make-handler
     'clock
     (make-methods
      'INSTALL
      (lambda ()
           (ask self 'ADD-CALLBACK
               (create-clock-callback 'tick-printer self 'PRINT-TICK)))
      'NAME      (lambda () name)
      'THE-TIME  (lambda () the-time)
'TICK
      (lambda ()
          (set! removed-callbacks '())
          (for-each (lambda (x)
                        (if (not (memq x removed-callbacks))
                               (ask x 'activate)))
                      (reverse callbacks))
          (set! the-time (+ the-time 1)))
      'ADD-CALLBACK
      (lambda (cb)
          (cond ((not (ask cb 'IS-A 'CLOCK-CALLBACK))
                 (error "Non callback provided to ADD-CALLBACK"))
                ((null? (filter (lambda (x) (ask x 'SAME-AS? cb))
                                 callbacks))
                 (set! callbacks (cons cb callbacks))
                 'added)
                (else  'already-present))))
     root-part)))
```

run-clock: advance clock some number of ticks

```
(define (run-clock n)
  (cond ((= n 0) 'DONE)
        (else (ask clock 'tick)
              ;; remember that this activates each item in callback list
              (run-clock (- n 1)))))

(define (create-clock . args)
  (apply create-instance clock args))
```

clock-callback: stores a target object, message, and arguments. When activated, it sends the target object the message. It can be thought of as a button that executes an action at every tick of the clock.

```
(define (clock-callback self name object msg . data)
  (let ((root-part (root-object self)))
    (make-handler
     'clock-callback
     (make-methods
      …
      'ACTIVATE (lambda () (apply ask object msg data))
      …)
     root-part)))
```