MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Lecture Notes, October 27 – Object Oriented Programming

Elements of Object Oriented Programming (OOP)

- Object oriented programs correspond to another perception of the world: entities (*objects*) which appear to be categorized into groups (*classes*) that behave similarly, but with individual differences bases on internal state.
- *Message passing*: Objects are loosely coupled; they communicate by sending messages to one another.
- Object oriented approaches are useful for construction of *systems* of many objects to model or *simulate* behavior of real or imaginary worlds.
- Additional means are needed for managing complexity in OOP systems, such as *inheritance* or *delegation*.

Message Passing Pair Implementation

```
(define (cons x y)
  (lambda (msg)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          ((eq? msg 'SET-CAR!)
           (lambda (new-car) (set! x new-car)))
          ((eq? msg 'SET-CDR!)
           (lambda (new-cdr) (set! y new-cdr)))
          (else (error "Unknown message" msg)))))

(define (car p)
  (p 'CAR))

(define (cdr p)
  (p 'CDR))

(define (pair? p)
  (and (procedure? p) (p 'PAIR?)))

(define (set-car! p new-car)
  ((p 'SET-CAR!) new-car))

(define (set-cdr! p new-cdr)
  ((p 'SET-CDR!) new-cdr))
```

An environment diagram illustrating

```
(define foo (cons 1 2))
(set-car! foo 0)
```

Example class and instance diagram corresponding to the `PAIR` abstraction.

Data-Directed Ship Implementation

```
(define (install-ship-package)
  ;; INTERNAL REPRESENTATION
  (define (make-ship position velocity num-torps)
    (list position velocity num-torps))

  (define (ship-position ship) (car ship))  ;accessor
  (define (ship-velocity ship) (cadr ship)) ;accessor

  (define (ship-move ship)                  ;mutator
    (set! position (add-vect position (scale-vect velocity *time-step*)))
    ship)

  (define (ship-fire-torp ship)             ;action
    (cond ((> num-torps 0) ...)
          (else ’FAIL)))

  ;; EXTERNAL REPRESENTATION - tagged object
  (define (tag x) (attach-tag ’ship x))
  (put ’make ’spaceship
    (lambda (p v t) (tag (make-ship p v t))))
  (put ’position ’spaceship ship-position)
  (put ’velocity ’spaceship ship-velocity)
  (put ’move ’spaceship (lambda (s) (tag (ship-move s))))
  (put ’fire-torpedo ’spaceship ship-fire-torp)
  ’done
  )

(define (move obj)
  (apply-generic ’move obj))
```

An operation table corresponding to the data-directed spaceship implementation:

- *Generic operations* are organized along the *operations* direction.
- *Object oriented* approach: organize along the *type* direction.

Message-Passing Ship Implementation

```
(define (make-spaceship position velocity num-torps)
  (define (move)
    (set! position (add-vect position (scale-vect velocity *time-step*)))
    'DONE)
  (define (fire-torp)
    (cond ((> num-torps 0) ...)
          (else 'FAIL)))
  (define (dispatch message . args)
    (cond ((eq? message 'POSITION) position)
          ((eq? message 'VELOCITY) velocity)
          ((eq? message 'MOVE) (move))
          ((eq? message 'FIRE-TORP) (fire-torp))
          (else (error "No method" message))))
  dispatch)

(define enterprise
  (make-spaceship (make-vect 10 10) (make-vect 5 0) 3))

(enterprise 'MOVE)
==> DONE

(enterprise 'POSITION)
==> (15 . 10)
```

With a corresponding environment diagram:

Some additional ideas to add to our universe:

- Other kinds of object in space: a PLANET
- A `clock` that moves time forward in the universe.
- A `display-handler` that manages a screen for drawing objects.
- A `proximity-sensor` that can determine if a collision occurs between objects, find nearby objects, and determine the range between two objects.
- A TORPEDO that moves toward a target, and explodes when near enough.

Here is a class diagram for our universe with extensions.

Here is an instance diagram for our universe with extensions.

Message-passing code to implement (some of) these extensions.

```
(define (make-planet position)
  (define (dispatch message)
    (cond ((eq? message 'PLANET?) #T)
          ((eq? message 'POSITION) position)
          ((eq? message 'CLOCK-TICK) 'DONE)
          ((eq? message 'DISPLAY) (draw ...))
          (else (error "No method" message))))
  dispatch)

(define (make-spaceship position velocity num-torps)
  (define (move)
    (set! position (add-vect position (scale-vect velocity *time-step*)))
    'DONE)
  (define (fire-torp target)
    (cond ((> num-torps 0)
           (set! num-torps (- num-torps 1))
           (let ((torp (make-torpedo position
                                    (sub-vect (target 'POSITION) position)
                                    target 3)))
             (add-to-universe torp)))
          (else 'FAIL)))
  (define (explode ship)
    (print "Ouch.  That hurt.")
    (remove-from-universe ship))
  (define (dispatch message . args)
    (cond ((eq? message 'SPACESHIP?) #T)
          ((eq? message 'POSITION) position)
          ((eq? message 'VELOCITY) velocity)
          ((eq? message 'MOVE) (move))
          ((eq? message 'FIRE-TORP) (fire-torp (car args)))
          ((eq? message 'EXPLODE) (explode (car args)))
          ((eq? message 'CLOCK-TICK)
           ; some strategy to decide what to do, e.g. find out
           ; if any enemy ships nearby and fire a torpedo.
           ...
           (move))
          ((eq? message 'DISPLAY) (draw ...))
          (else (error "No method" message))))
  dispatch)

(define (make-torpedo position velocity target proximity-fuse)
  (define (move)
    (set! position (add-vect position (scale-vect velocity *time-step*)))
    (let ((proximity (find-distance position (target 'POSITION))))
      (cond ((<= proximity proximity-fuse)
             (target 'EXPLODE target)
             (remove-from-universe torp))
            (else 'NOT-CLOSE-YET))))
  (define (dispatch message . args)
    (cond ((eq? message 'TORPEDO?) #T)
          ((eq? message 'POSITION) position)
```

```
          ((eq? message 'VELOCITY) velocity)
          ((eq? message 'MOVE) (move))
          ((eq? message 'CLOCK-TICK) (move))
          ((eq? message 'DISPLAY) (draw ...))
          (else (error "No method" message))))
  dispatch)

; The universe
;
(define *universe* '())
(define (add-to-universe thing)
  (set! *universe* (cons thing *universe*)))
(define (remove-from-universe thing)
  (set! *universe* (delq thing *universe*)))

; The clock
;
(define *time-step* 1)

(define (clock)
  (for-each (lambda (thing) (thing 'CLOCK-TICK)) *universe*)
  (let ((collisions (find-collisions *universe*)))
    (for-each (lambda (thing)
                (if (SPACESHIP? thing) (thing 'EXPLODE thing)))
              *universe*))
  (for-each (lambda (thing) (thing 'DISPLAY)) *universe*))

(define (run-clock n)
  (cond ((zero? n) 'DONE)
        (else (clock)
              (run-clock (-1+ n)))))

; Proximity Detector
;
(define (find-collisions things)
  ;; return a list of objects (from things) that have collided
  ...)
(define (distance p1 p2)
  (vector-size (sub-vect p2 p1)))

;; Building some things
;;
(define earth (make-planet (make-vect 0 0)))
(define enterprise (make-spaceship (make-vect 10 10) (make-vect 5 0) 3))
(define warbird (make-spaceship (make-vect -10 10) (make-vect 10 0) 10))

(add-to-universe earth)
(add-to-universe enterprise)
(add-to-universe warbird)

(run-clock 100)
```