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 29 – OOP with Inheritance

Inheritance

The notion of a class of objects that is a specialization or "subclass" of another class is a useful tool for organizing complex systems. It enables us to localize shared behavior in the *superclass* and isolate just the new or changed behavior in the *subclass* that *inherits from* the superclass.

When inheritance is used, the object oriented programming system (OOPS) must also define an *inheritance rule* that tells us what method to use. In single inheritance this is relatively easy. With *multiple inheritance* (one class directly inheriting from more than one superclass) this can be very confusing and byzantine.

Here is a simple class diagram showing

- "is-a" (or "inherits-from") relationships between classes
- "composed of" relationships
- other relationships

Object Oriented Design of a Lecturer/Singer System

Sketch the class diagram for a system consisting of speaker, lecturer, arrogant lecturer, and singer classes.

OOP Implementation in Scheme

We will implement the elements of OOP in Scheme as follows:

- No new mechanisms needed: just some *conventions*
- *Objects* will be implemented as:
  - object *identity* is achieved because:
  - *local state* for each object instance is achieved by:
  - *methods* are implemented as:
  - *inheritance* will be achieved by:
- *Classes* will be implemented as:

OOP System - Version 1: Methods

```
(define (make-speaker name)
  (lambda (message)
    (case message
      ((NAME) (lambda () name))
      ((CHANGE-NAME)
       (lambda (new-name) (set! name new-name)))
      ((SAY)
       (lambda (list-of-stuff)
         (display-message list-of-stuff)
         'NUF-SAID))
      (else (no-method)))))
```

Abstract out retrieval of method from the object (given the message)...

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

... and the combined (1) retrieval and (2) application of that method to the arguments:

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method args)
        (error "No method for message" message))))
```

Detection of methods (or missing methods):

```
(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))

(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else (error "Object returned non-message" x))))
```

Example using this approach:

```
(define p (make-speaker 'George))

(ask p 'NAME)
==> george

(ask p 'SAY '(the sky is blue))
the sky is blue
==> nuf-said
```

OOP System - Version 2: Inheritance by Delegation

We need to extend our OOP system in two ways: (1) we need our objects to have access to "themselves" (as a variable that can be used in methods); and (2) we need a way to inherit (or gain use of) the structure and methods of a superclass.

(1) The `self` variable

What if we want a speaker to call its own method? The **problem** with our first implementation is that object methods have no access to the "object" itself! The **solution** is to require that all methods take `self` as their first parameter, and always pass the "object" as the first argument.

Let's reimplement the `speaker` class to do this:

```
(define (make-speaker name)
  (lambda (message)
    (case message
      ((NAME) (lambda (self) name))
      ((CHANGE-NAME)
       (lambda (self new-name)
         (set! name new-name)
         (ask self 'SAY (list 'call 'me name))))
      ((SAY)
       (lambda (self list-of-stuff)
         (display-message list-of-stuff)
         'NUF-SAID))
      (else (no-method)))))
```

with the following extension to `ask` so that the object is always passed:

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method object args)
        (error "No method for message" message))))
```

And an example:

```
(ask p 'CHANGE-NAME 'fred)
call me fred
```

(2) Inheritance by Delegation

We want a lecturer to be a kind of speaker - that inherits the behavior of speakers but adds to that behavior. Our approach to inheritance using **delegation**:

- Inherit speaker behavior by adding an "internal" speaker object

  - Get internal object to act on behalf of object by delegation

- If message is not recognized, pass the buck
- Can change or specialize behavior:
  - Add new methods
  - Change operation of methods

For example, `lecturer` inherits from `speaker`:

```
(define (make-lecturer name)
  (let ((speaker (make-speaker name)))
    (lambda (message)
      (case message
        ((LECTURE)
         (lambda (self stuff)
           (delegate speaker self 'SAY
                     (append '(therefor) stuff))))
        (else (get-method message speaker))))))

(define d (make-lecturer 'Duane))
(ask d 'LECTURE '(the sky is blue))
therefor the sky is blue
```

This requires the following new OOP system mechanism:

```
(define (delegate to from message . args)
  (let ((method (get-method message to)))
    (if (method? method)
        (apply method from args)
        (error "No method" message))))

(define (ask object message . args)
  (apply delegate object object message args))
```

A chain of inheritance is thus possible. For example, consider an “Arrogant Lecturer” that changes the basic way of talking: he/she appends “obviously” to *everything* he/she says...

```
(define (make-arrogant-lecturer name)
  (let ((lecturer (make-lecturer name)))
    (lambda (message)
      (case message
        ((SAY)
         (lambda (self stuff)
           (delegate lecturer self
               'SAY (append stuff '(obviously)))))
        (else (get-method message lecturer))))))

(define b (make-arrogant-lecturer 'Bill))
```

```
(ask b 'SAY '(the sky is blue))
the sky is blue obviously

(ask b 'LECTURE '(the sky is blue))
therefor the sky is blue                 ;; BUG!
```

Fixing the Bug - ask vs delegate

To get the lecturer to change the way he/she says *everything* as desired:

```
(define (make-lecturer name)
  (let ((speaker (make-speaker name)))
    (lambda (message)
      (case message
        ((LECTURE)
         (lambda (self stuff)
  ;;       (delegate speaker self 'SAY
  ;;                   (append '(therefor) stuff))
           (ask self 'SAY
                     (append '(therefor) stuff))))
        (else (get-method message speaker))))))

(define b (make-arrogant-lecturer 'Bill))
(ask b 'SAY '(the sky is blue))
he sky is blue obviously

(ask b 'LECTURE '(the sky is blue))
therefor the sky is blue obviously
```

OOP System - Version 3: Multiple Inheritance

We might want objects that inherit methods from more than one type. Suppose in addition to a named speaker we have an anonymous singer:

```
(define (make-singer)
  (lambda (message)
    (case message
      ((SAY)
       (lambda (self stuff)
         (display-message (append stuff '(tra la la))))
      ((SING)
       (lambda (self)
         (ask self 'SAY '(the hills are alive))))
      (else (no-method)))))
```

Now we'll create a singing arrogant lecturer:

```
(define julie
  (let ((singer (make-singer))
        (lecturer (make-arrogant-lecturer 'Julie)))
    (lambda (message)
      (find-method message singer lecturer))))

(ask julie 'SING)
the hills are alive tra la la

(ask julie 'LECTURE '(the sky is blue))
therefor the sky is blue tra la la
```

This requires an additional `find-method` that gives power over the order of method lookup:

```
(define (find-method message preferred . others)
  (define (loop objs)
    (let ((method (get-method-from-object
                    message (car objs)))
          (rest (cdr objs)))
      (if (or (method? method) (null? rest))
          method
          (loop rest))))
  (loop (cons preferred others)))
```

Alternative Multiple Inheritance

We could give ourselves (build an OOPS with) lots of flexibility - suppose we want to pass the message on to multiple internal objects (not just some "preferred" one)?

```
(define julie
  (let ((singer (make-singer))
        (lecturer (make-arrogant-lecturer 'Julie)))
    (lambda (message)
      (lambda (self . args)
        (apply delegate-to-all
               (list singer lecturer)
               self
               args)))))

(ask julie 'SAY '(the sky is blue))
the sky is blue tra la la
therefore the sky is blue tra la la
therefore therefore the sky is blue tra la la
```

With the following additional "method routing" procedure:

```
(define (delegate-to-all to-list from message . args)
  (foreach
    (lambda (to-whom)
      (apply delegate to-whom from message args))
  to-list)
```