Implementing Objects and Classes in Scheme

Lecturer:
Professor Albert R. Meyer

November 6, 1997

The Simple-place Class

```
(define-class
  (Class-Name Simple-place)
  (Slots (things '()))
  (Methods
   ((HAVE-THING?
     (lambda (thing) (memq thing things)))
    (ADD-THING
       (lambda (new-thing)
         (or (ask self 'HAVE-THING? new-thing)
             (set! things (cons new-thing things)))))
    (DEL-THING
     (lambda (thing)
       (delq thing things))))))
```

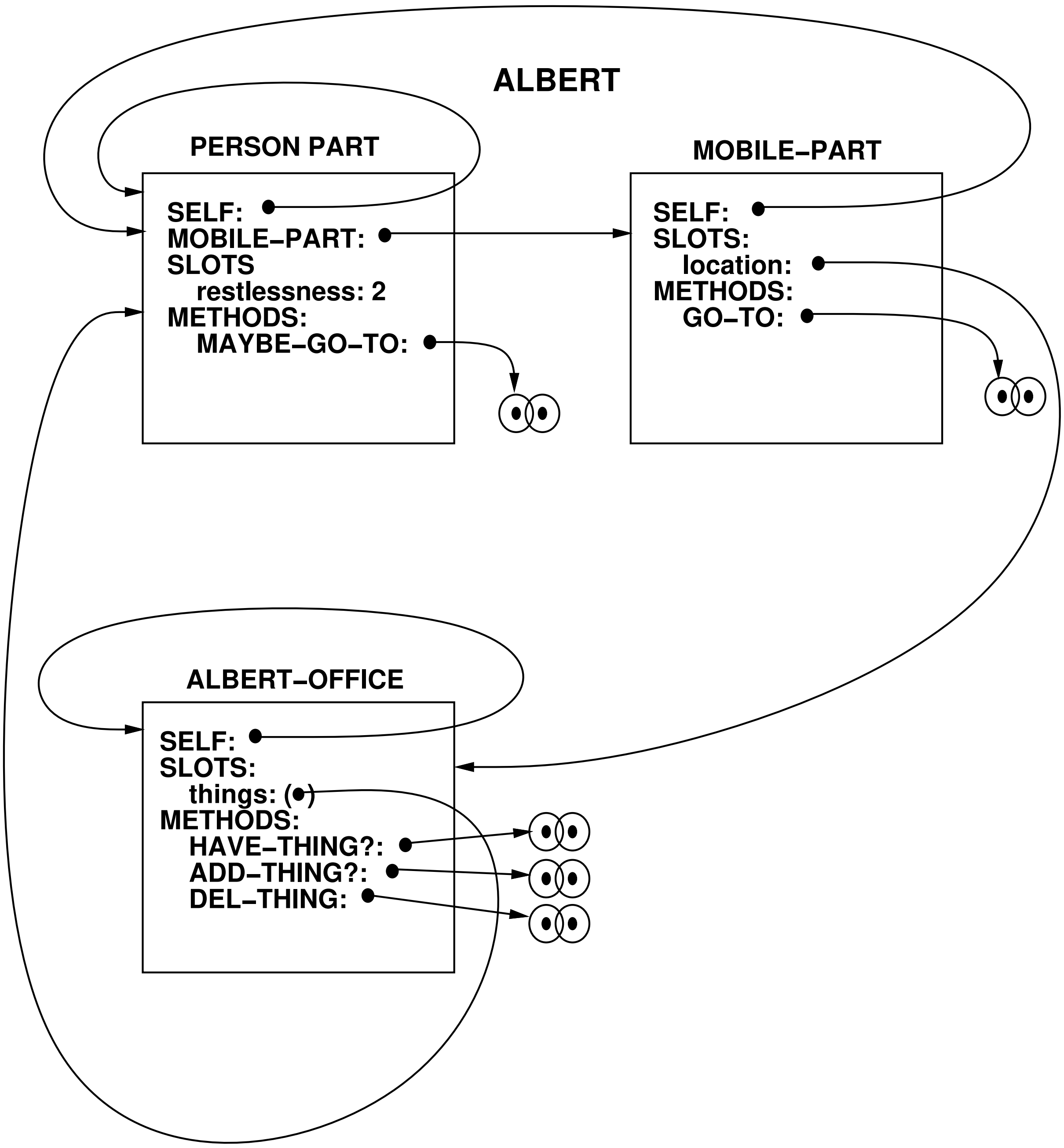
The Simple-Mobile Class

```
(define-class
  (CLASS-NAME simple-mobile)
  (SLOTS (location '?))
  (CONSTRUCTOR
   (lambda (initial-location)
     (set! location initial-location)
     (ask location 'ADD-THING self))
   (METHODS
    ((GO-TO
      (lambda (new-place)
        (begin
          (ask location 'DEL-THING self)
          (set! location new-place)
          (ask new-place 'ADD-THING self))))))))
```

Simple People

```
(define-class
  (CLASS-NAME simple-person)
  (PARTS (mobile-part))        ;INHERITED PART
  (SLOTS (restlessness '?))
  (CONSTRUCTOR
   (lambda (init-place init-restness)
     (begin
       (set! mobile-part
               ;;INHERITANCE FROM SIMPLE-MOBILE:
              (new-part
               simple-mobile self init-place))
       (set! restlessness init-restness))))
  (METHODS
   ((MAYBE-GO-TO
     (lambda (new-place)
       (and (= (random restlessness) 0)
            (ask self 'GO-TO new-place)))))))

(define albert-office (new simple-place))
(define albert
  (new simple-person albert-office 2))
```


ALBERT
PERSON PART
SELF:
MOBILE-PART:
SLOTS
restlessness: 2
METHODS:
MAYBE-GO-TO:
MOBILE-PART
SELF:
SLOTS:
location:
METHODS:
GO-TO:
ALBERT-OFFICE
SELF:
SLOTS:
things: ( )
METHODS:
HAVE-THING?:
ADD-THING?:
DEL-THING:

Why SELF

```
(define corner-office (new simple-place))
(ask albert 'MAYBE-GO-TO corner-office)
```

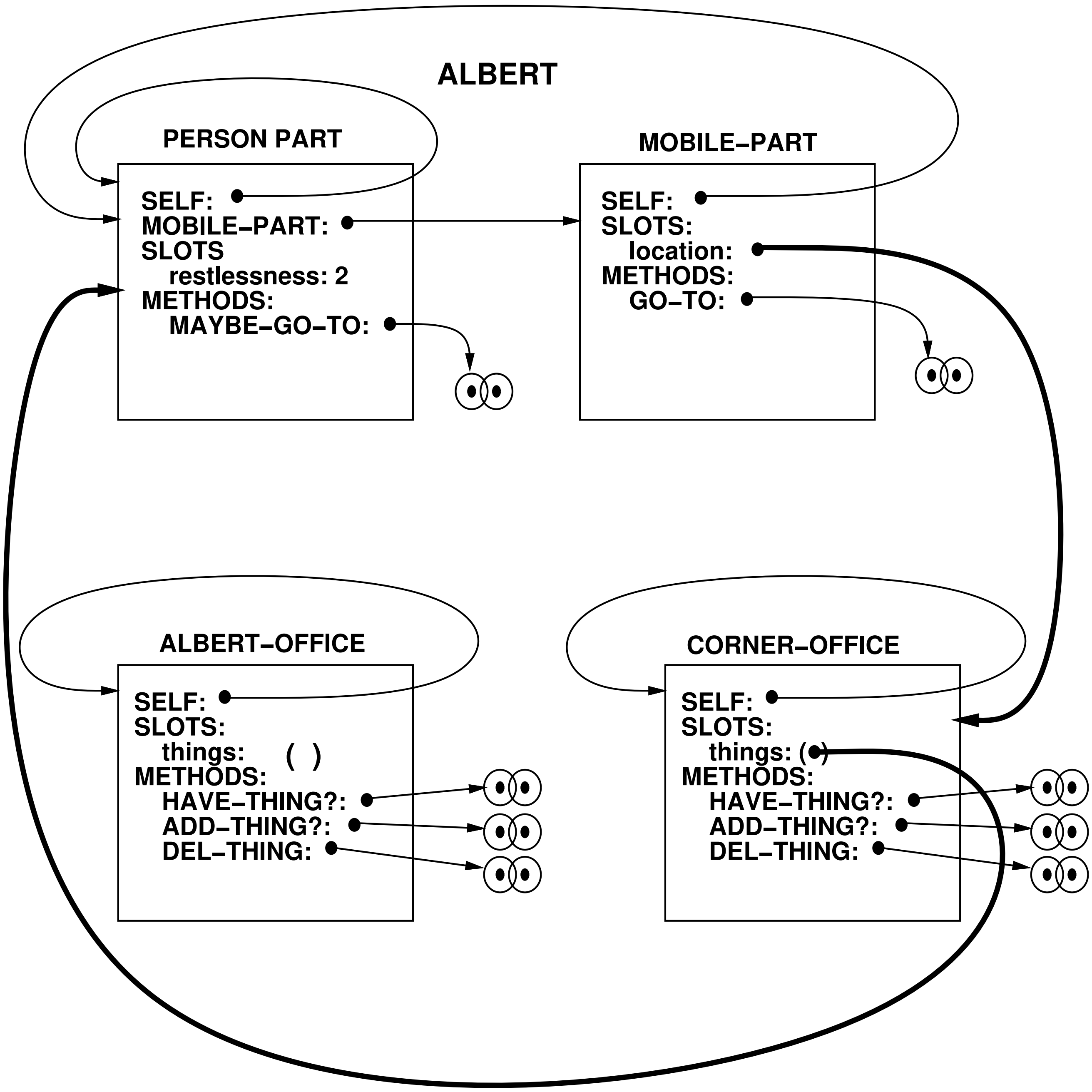
Suppose random returns 0, then Albert would:

```
(ask self 'GO-TO new-place)
```

Albert's PERSON-PART has NO **GO-TO** METHOD. He gets an INHERITED **GO-TO** method FROM his MOBILE-PART:

```
(lambda (new-place)
  (begin
    (ask location 'DEL-THING self)
    (set! location new-place)
    (ask new-place 'ADD-THING self)))
```

and applies it to CORNER-OFFICE. Since the value of SELF is Albert's PERSON-PART the **things** of Albert-office and Corner-Office get updated properly.



ALBERT
PERSON PART
SELF:
MOBILE-PART:
SLOTS
restlessness: 2
METHODS:
MAYBE-GO-TO:
MOBILE-PART
SELF:
SLOTS:
location:
METHODS:
GO-TO:
ALBERT-OFFICE
SELF:
SLOTS:
things: ( )
METHODS:
HAVE-THING?:
ADD-THING?:
DEL-THING:
CORNER-OFFICE
SELF:
SLOTS:
things: ( )
METHODS:
HAVE-THING?:
ADD-THING?:
DEL-THING:

Object Representation as Message Passing Procedures

```
(define albert-office
  (let ((self 'will-be-albert-office)
        (things '()))
    (lambda (mes)
      (cond
       ((eq? mes 'HAVE-THING?
             (lambda (thing)
               (memq thing things))))
       ((eq? mes 'ADD-THING
             (lambda (new-thing)
               (or
                (ask self
                     'HAVE-THING? new-thing)
                (set! things
                      (cons new-thing things)))))
        ((eq? mes 'DEL-THING ...))
        (else (fail)))))))
```

PUZZLE: Want to

```
(set! self albert-office)
```

but **self** is private to **albert-office**.

Wiring Up Albert-office **self**

```
(define albert-office
  (let ((self
          'will-be-albert-office) ;INSTALL IT LATER
         (things '()))
              ;;;FIRST BUILD OFFICE:
    (let ((office-object
            (lambda (mes)
              (cond
               ((eq? mes 'HAVE-THING?)
                (lambda (thing)
                  (memq thing things)))
               ((eq? mes 'ADD-THING)
                (lambda (new-thing)
                  (or
                   (ask self
                        'HAVE-THING? new-thing)
                   (set! things
                         (cons new-thing things)))))
               ((eq? mes 'DEL-THING) ...)
               (else (fail))))))
      (begin
              ;;;THEN INITIALIZE SELF:
        (set! self office-object)
        office-object))))
```

EASY because NO inherited PARTS.

Wiring Up Inherited Parts

```
(define albert
  (let ((self 'will-be-albert)
        (mobile-part
         'will-be-albert-mobile-part)
        (restlessness 2))
    (let ((uninitialized-albert
           (lambda (mes)
             (cond
              ((eq? mes 'MAYBE-GO-TO)
               (lambda (new-place)
                 (and
                  (= (random restlessness) 0)
                  (ask self 'GO-TO new-place))))
              (else
                   ;INHERIT ADDITIONAL METHODS:
               (get-method mes mobile-part))))))
      (begin
        (set! self uninitialized-albert)
        uninitialized-albert))))
```

Installing **`albert`**'s **`self`** as for offices – easy. But can't initialize **`mobile-part`**, because it's not built yet.

OK, BUILD IT: also easy because MOBILE-PART's self is **albert** who is already built:

```
(define albert-mobile-part
  (let ((self albert)    ;ALBERT ALREADY BUILT
        (location
         albert-office)) ;OFFICE ALREADY BUILT
    (lambda (mes)
      (cond
       ((eq? mes 'GO-TO)
        (lambda (new-place)
          (begin
            (ask location 'DEL-THING self)
            (set! location new-place)
            (ask new-place 'ADD-THING self)))
       (else (fail)))))))
```

The only thing left to do is to SET! **albert**'s **mobile-part** variable to **albert-mobile-part**.

PUZZLE: **mobile-part** is private to **albert**. How to set it?

ANSWER: Can be done if **albert** *cooperates* – add an **INSTALL-MOBILE-PART** method to **ALBERT**:

```
(define albert
  (let ((self 'will-be-albert)
        (mobile-part 'will-be-albert-mobile-part)
        (restlessness 2))
    (let ((uninitialized-albert
           (lambda (mes)
             (cond
              ((eq? mes 'MAYBE-GO-TO)
               (lambda (new-place)
                 (and (= (random restlessness) 0)
                      (ask self 'GO-TO new-place))))
     ;SYSTEM METHOD ONLY:
              ((eq? mes 'INSTALL-MOBILE-PART)
               (lambda (new-mobile-part)
                 (set! mobile-part new-mobile-part)))
              (else (get-method mes mobile-part))))))
      (begin
        (set! self uninitialized-albert)
        uninitialized-albert))))
```

Now simply

```
(ask albert
     'INSTALL-MOBILE-PART albert-mobile-part)
```

This works correctly, but burdens **`albert`** with the system's **`INSTALL-MOBILE-PART`** method.

Since this method is *not needed* once **`albert`** has been built, and we don't really want it to stay around.

Here's a way around this:

Define a "make albert and installer" procedure which returns **two** things:

(i) **`albert`** with an uninitialized **`mobile-part`** variable, and

(ii) the **`INSTALL-MOBILE-PART`** method.

```
(define (make-uninitialized-albert-and-installer)
  (let ((self 'will-be-albert)
        (mobile-part 'will-be-albert-mobile-part)
        (restlessness 2))
    (let ((uninitialized-albert
           (lambda (mes)
             (cond
              ((eq? mes 'MAYBE-GO-TO)
               (lambda (new-place)
                 (and (= (random restlessness) 0)
                      (ask self 'GO-TO new-place))))
              (else (get-method mes mobile-part))))))
      (begin
        (set! self uninitialized-albert)
        (list
         uninitialized-albert
            ;INSTALL-MOBILE-PART method:
          (lambda (new-mobile-part)
            (begin
              (set! mobile-part new-mobile-part))))))))
```

Now previous construction goes as follows:

(1) Define **`albert-office`** as above.

(2) Create an uninitialized **`albert`** and installer for it:

```
(define albert-and-installer
  (make-uninitialized-albert-and-installer))

(define albert (car albert-and-installer))

(define (albert-mobile-part-installer
          (cadr albert-and-installer)))
```

(3) Define **`albert-mobile-part`** as above.

This leaves us in exactly the ”puzzling” situation above, with

- **`albert-office`**,
- **`albert`**, and
- **`albert-mobile-part`** fully built,
- **except** that **`albert`**'s *still private* **`mobile part`** variable is not initialized.

But now we have a solution:

(4) Use the installer to plug in **`albert-mobile-part`** where it belongs:

```
(mobile-part-installer albert-mobile-part)
```

CONSTRUCTION DONE!

General Technique for Wiring Up Objects

(1) define a

`make-uninitialized-object-and-installer`

procedure for each object part,

(2) use these **`make-`** procedures to define the uninitialized parts and installer procedures for each part,

(3) apply the installer procedures to the parts until all the parts are installed in the proper slots.

Implementing Classes

A CLASS is implemented as a procedure which uses the general technique above to construct instance objects of the class.

Given initial values for instance variables (like **`location`** or **`restlessness`**), the class procedure calls the **`make-`** procedure for uninitialized instance objects of the class and the **`make-`** procedures of the classes from which it inherits, and then uses the installer procedures to initialize slots and wire up the parts.

The DEFINE-CLASS form is implemented as a procedure which takes a class specification and translates it into a class procedure. The details of DEFINE-CLASS are given in the code for Problem Set 7.

6.001 students are expected to understand how objects are represented as message-passing procedures and how to wire up some simple ones as we did above. They are not expected to master the implementation of Classes and DEFINE-CLASS; that is left as an optional challenge for aspiring Scheme programmers.