Object-Oriented Programming
MIT 6.001 Recitation

Diagrams

Where do you find the list of states and method for a class such as named-object or container?

Draw a class diagram for a thing. Draw a class diagram for a person

Draw the environment diagram for a thing instance (create-thing ‘stuff statacenter).

Use the classes

Create a “toy” mobile-thing instance. Create a person bob.

Have bob take the thing. What happens exactly in terms of inheritance and message handling?

Draw an instance diagram.

How do you get the name of all the things carried by bob? How do you get the name of their location?

Create a new method for the class person that takes all the things of another person.

Inheritance

We are going to derive new classes for the adventure game. As you go, ask yourself these questions:

1. What should class or classes should it inherit from?
2. What new fields does it need?
3. What new methods does it need?
4. What methods from its superclass(es) should it override? Should the behavior of its overrided methods completely replace what the supperclass does, or just augment it? How should superclass methods be called from an overriding method?

Ideas of classes

A **monk** refuses all possessions. (Hint: a person can be asked to TAKE something.)

A **bomb**, when triggered, destroys everything around it. (Hint: a thing has a LOCATION, which is a container with a list of THINGS.)

A **recorder** remembers everything it ever said and can replay it all on command.

Achile, never gets hurt. A **person with a small pocket** can only carry 2 objects.

Derive a **weapons** system (include notions such as maximum of damage, ammunition).

Code

```
;;-------------------------------------------------------------
;; Instance

(define (make-instance) (list 'instance #f))

(define (create-instance maker . args)
  (let* ((instance (make-instance))
         (handler (apply maker instance args)))
    (set-instance-handler! instance handler)
    (if (method? (get-method 'INSTALL instance)) (ask instance 'INSTALL))
    instance))

;;-------------------------------------------------------------
;; Handler

(define (make-handler typename methods . super-parts)
  […]    ;check for possible programmer errors (omitted for legibility
   (lambda (message)
     (case message
       ((TYPE) (lambda () (type-extend typename super-parts)))
       ((METHODS) (lambda ()  (append (method-names methods)
             (append-map (lambda (x) (ask x 'METHODS)) super-parts))))
       (else (let ((entry (method-lookup message methods)))
               (if entry (cadr entry) (find-method-from-handler-list message super-parts))))))))
```

```
;;--------------------
;; named-object

(define (create-named-object name)   (create-instance named-object name))

(define (named-object self name)
  (let ((root-part (root-object self)))
    (make-handler
     'named-object
     (make-methods
      'NAME    (lambda () name)
      'INSTALL (lambda () 'installed)
      'DESTROY (lambda () 'destroyed))
     root-part)))

;;--------------------
;; container

(define (container self)
  (let ( (root-part (root-object self))
         (things '()))
    (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)))


;;--------------------
;; thing

(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) (ask (ask self 'LOCATION) 'ADD-THING self))
      'LOCATION (lambda () location)
      'DESTROY  (lambda () (ask (ask self 'LOCATION) 'DEL-THING self))
      'EMIT     (lambda (text) (ask screen 'TELL-ROOM (ask self 'LOCATION)
                                (append (list "At" (ask (ask self 'LOCATION) 'NAME)) text))))
     named-part)))

;;--------------------
;; mobile-thing

(define (create-mobile-thing name location)
  (create-instance mobile-thing name location))

(define (mobile-thing self name location)
  (let ((thing-part (thing self name location)))
    (make-handler
     'mobile-thing
     (make-methods
      'LOCATION  (lambda () location) ; This shadows message to thing-part!
      '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)
      'CREATION-SITE (lambda () (ask thing-part 'location)))
     thing-part)))

;;--------------------
;; place

(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))
        (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))
        (health            3)
        (strength          1))
    (make-handler
     'person
     (make-methods
      'STRENGTH (lambda () strength)
      'HEALTH (lambda () health)
      '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)
      'HAVE-FIT (lambda () (ask self 'SAY '("Yaaaah! I am upset!")) 'I-feel-better-now)
      '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
             (ask self 'SAY (list "I am already carrying" (ask thing 'NAME))) #f)
           ((or (ask thing 'IS-A 'PERSON) (not (ask thing 'IS-A 'MOBILE-THING)))
             (ask self 'SAY (list "I try but cannot take" (ask thing 'NAME)))  #F)
         (else (let ((owner (ask thing 'LOCATION)))
           (ask self 'SAY (list "I take" (ask thing 'NAME) "from" (ask owner 'NAME)))
           (if (ask owner 'IS-A 'PERSON) (ask owner 'LOSE thing self)
           (ask thing 'CHANGE-LOCATION self))
         thing))))
      'LOSE   (lambda (thing lose-to)
         (ask self 'SAY (list "I lose" (ask thing 'NAME)))
         (ask self 'HAVE-FIT)
         (ask thing 'CHANGE-LOCATION lose-to))
      'DROP (lambda (thing)
         (ask self 'SAY (list "I drop" (ask thing 'NAME) "at" (ask (ask self 'LOCATION) 'NAME)))
         (ask thing 'CHANGE-LOCATION (ask self 'LOCATION)))
      'SUFFER (lambda (hits perp)
         (ask self 'SAY (list "Ouch!" hits "hits is more than I want!"))
         (set! health (- health hits))
         (if (<= health 0) (ask self 'DIE perp))
         health)
      'DIE    (lambda (perp)
         (for-each (lambda (item) (ask self 'LOSE item (ask self 'LOCATION))) (ask self 'THINGS))
         (ask screen 'TELL-WORLD '("An earth-shattering, soul-piercing scream is heard..."))
         (ask self 'DESTROY))
      'HAS-A            ;your method here (exercise 2)
      'HAS-A-THING-NAMED ;your method here (exercise 2)
     mobile-thing-part container-part)))
```

Multiple inheritance

```
;; a THING is an object with a name, a weight, and a volume.
(define (create-thing name weight volume) (create-instance thing name weight volume))

(define (thing self name weight volume)
  (let ((root-part (root-object self)))
    (make-handler
      'thing
      (make-methods
        'NAME (lambda () name)
        'WEIGHT (lambda () weight)
        'VOLUME (lambda () volume)
        'DENSITY (lambda () (/ weight volume))
      root-part)))

;; a CONTAINER is a set of things.
(define (create-container) (create-instance container))

(define (container self)
  (let ((root-part (root-object self))
        (things '()))
    (make-handler
      'container
      (make-methods
        'THINGS) (lambda () things)
        'ADD-THING (lambda (thing)  (set! things (cons thing things))
                                    (map (lambda (thing) (ask thing 'NAME)) things))
        'WEIGHT (lambda() (foldr + 0  (map (lambda (thing) (ask thing 'WEIGHT)) things))))
        'VOLUME (lambda() (foldr + 0 (map (lambda (thing) (ask thing 'VOLUME))      things)))
  root-part)))

;; a CRATE is a hard-sided wooden crate. its weight varies with its
;; contents, but its volume is always the same.

(define (create-crate name weight-when-empty volume)
   (create-instance crate name weight-when-empty volume))

(define (crate self name weight-when-empty volume)
  (let ((thing-part (thing self name weight-when-empty volume))
        (container-part (container self)))
    (make-handler
      'crate
      (make-methods ; YOUR-CODE-HERE
  container-part thing part)))

;; a BAG is a flexible cloth bag. Both its weight and its volume
;; depend on its contents.
(define (create-bag name)
  (create-instance bag name))
(define (bag self name)
  (let ((thing-part (thing self name 0 0))
        (container-part (container self)))
    (make-handler
      'container
      (make-methods ;YOUR-CODE-HERE
  container-part thing part)))
```

Finish the definitions above. You can make other changes to the code if you provide proper justification.

```
(define armoire (create-thing 'old-armoire 200 16))
(define bear (create-thing 'teddy-bear 1 0.5))
(define crate (create-crate 'crate 40 20))
(define bag (create-bag 'bag))
* (ask crate 'WEIGHT)            =>
* (ask crate 'ADD-THING armoire) =>
* (ask crate 'WEIGHT)            =>
* (ask crate 'VOLUME)            =>
* (ask crate 'DENSITY)           =>
* (ask bag 'WEIGHT)              =>
* (ask bag 'ADD-THING bear)      =>
* (ask bag 'WEIGHT)              =>
* (ask bag 'VOLUME)              =>
```