Quiz 7a

1. (3 points) We are given the following class definition:

(define-class (counter x)

(method (incr)

(set! x (+ x 1))))

Ben Bitdiddle wants to extend the counter class to

start at 0 and increment by 2.

He creates the following definition.

(define-class (even-counter-from-0)

(parent (counter 0))

(method (incr)

(set! x (+ x 2))))

Is there anything wrong with the above definition?  **More than one**

**of the following might be true; check all that are true.**

\_\_\_\_\_ Yes, you cannot have a parent that has a different number of

instantiation variables.

\_\_\_X\_\_ Yes, x is only defined in the parent. even-counter-from-0 cannot change the value.

\_\_\_\_\_ Yes, the method incr conflicts with the parent's incr.

\_\_\_\_\_ Yes, you cannot have both a parent and a method in an OOP class.

\_\_\_\_\_ No, there is nothing wrong with the definition.

2. (3 points) We are going to simulate a Compact Disc (CD) player using object-oriented

programming. Use the OOP notation as described in the course reader.

(a) Define a CD object class. Every CD contains the following information:

a number identifying the recording, and a list of numbers, one per song,

indicating where on the CD that song begins. (For our purposes we can think

of positions on the disc in terms of the number of seconds of music that come

before it.) To instantiate a CD we'll provide two arguments, the ID number and

the timing list:

(define With-the-Beatles (instantiate cd 102574 '(0 102 293 542 ...)))

The timing list contains one extra number at the end, which is the position

at which an additional song would begin if there were one. In other words,

this extra number indicates the total time of the CD.

A CD object accepts three messages: id asks for the ID number;

songs asks for the number of songs recorded on the disc; and

index takes a number as its argument and returns the corresponding

number from the timing list. (If the argument is zero it returns the

first element of the list, and so on.)

> (ask With-the-Beatles 'index 2)

293

(define-class (cd id timing)

(method (id) id)

(method (songs) (-1+ (length timing)))

(method (index n) (nth n timing)) )

(b) Now define a CD-player class. A good simulation would be very

complicated, mainly because once a CD is playing, the object continues

to do work even if it gets no more messages. But we'll only simulate

a couple of features. In particular, we won't actually play any songs!

The load message takes a CD object as its argument and ``loads'' that

CD into the player. The returned value is the ID number of the CD. The

effect of loading a CD is that later messages to the player refer implicitly

to the loaded CD.

The length message takes a song number as its argument, and returns

the length of that song (the difference between its position and the one

after it).

The goto message takes a song number as its argument, and returns

the position at which that song begins. (A more realistic simulation would

actually move the laser beam to that position and begin playing, but we'll

just return the position.)

> (define my-player (instantiate CD-player))

> (ask my-player 'load With-the-Beatles)

102574

> (ask my-player 'goto 3)

542

> (ask my-player 'length 1)

191 ;; this is 293-102

(define-class (cd-player)

(instance-vars (loaded-cd #f))

(method (load cd)

(set! loaded-cd cd)

(ask cd 'id))

(method (length n)

(- (ask loaded-cd 'index (1+ n))

(ask loaded-cd 'index n) ))

(method (goto n)

(ask loaded-cd 'index n)) )

1 point for part a and 2 for part b. For part b minus one point if parent was used. Or if a CD player owned a single CD permanently. The other 2 points go to having correct logic/syntax.

3. (4 points) We are going to simulate a mailbox using object-oriented programming.

Use the OOP notation as described in the course reader.

(a) Define a mailbox object class. This is the kind of mailbox you

see by the side of the road, out in the country, with a little red

flag that can be up or down. A mailbox has the following information:

an *address* that is given when the mailbox is created, that

never changes; the *flag*, represented as #t if it's up or

#f if it's down; and the letters, in a list. To create a mailbox

we provide the address, which is a number:

(define this-box (instantiate mailbox 547))

A mailbox object accepts the following messages: address asks for

the address; flag? returns true or false depending on the state of

the flag; flag takes an argument of true or false and sets the flag

accordingly; put takes a letter (which we'll represent with any

word or list) as its argument and adds that letter to the ones already in

the box; get empties the mailbox and returns the list of all the

letters that were in it.

> (ask this-box 'address)

547

> (ask this-box 'flag #t)

> (ask this-box 'flag?)

#t

> (ask this-box 'put 'hello)

> (ask this-box 'put '(another letter))

> (ask this-box 'get)

((another letter) hello)

(define-class (mailbox address)

(instance-vars (flag? #f) (letters ‘()))

(method (flag val)

(set! flag? val))

(method (put letter)

(set! letters (cons letter letters)))

(method (get)

(let ((myletters letters))

(set! Letters ‘())

myletters)))

(b) Now define a movable-mailbox class. These objects work just like

mailboxes, except that there is an additional message new-address

that takes a number as its argument and changes the address to that

number, so that later address messages will get the new value:

> (define a-box (instantiate movable-mailbox 2456))

> (ask a-box 'address)

2456

> (ask a-box 'new-address 4321)

> (ask a-box 'address)

4321

(define-class (movable-mailbox address)

(parent (mailbox address))

(method (new-address newaddr)

(set! address newaddr)))

2 points for each section. 1 point off for each mistake.