Structure and Interpretation of Computer Programs
Second Edition
Programming task 4

Adventure game - Object Oriented Programming

This programming assignment explores two ideas: the simulation of a world in which objects are characterized by collections of state variables, and the use of *object-oriented programming* as a technique for modularizing worlds in which objects interact. These ideas are presented in the context of a simple simulation game like the ones available on many computers. Such games have provided an interesting waste of time for many computer lovers.

This problem set begins by describing the overall structure of the simulation. The theoretical questions in part 2 will help you to master the ideas involved. Part 3 contains the assignment itself.

Part 1: The SICP Adventure Game

The basic idea of simulation games is that the user plays a character in an imaginary world inhabited by other characters. The user plays the game by issuing commands to the computer that have the effect of moving the character about and performing acts in the imaginary world, such as picking up objects. The computer simulates the legal moves and rejects illegal ones. For example, it is illegal to move between places that are not connected (unless you have special powers). If a move is legal, the computer updates its model of the world and allows the next move to be considered.

Our game takes place in a strange, imaginary world called MIT, with imaginary places such as a computer lab, Building 36, and Tech Square. In order to get going, we need to establish the structure of this imaginary world: the objects that exist and the ways in which they relate to each other.

Initially, there are three procedures for creating objects:

```
(make-thing  name)
(make-place  name)
(make-person name birthplace restlessness)
```

In addition, there are procedures that make people and things and procedures that install them in the simulated world. The reason that we need to be able to create people and things separately from installing them will be discussed in one of the exercises later. For now, we note the existence of the procedures

```
(make&install-thing  name birthplace)
(make&install-person name birthplace restlessness)
```

Each time we make or make and install a person or a thing, we give it a name. People and things also are created at some initial place. In addition, a person has a restlessness factor that determines how often the person moves. For example, the procedure `make&install-person` may be used to create the two imaginary characters, `albert` and `gerry`, and put them in their places, as it were.

```
(define albert-office (make-place 'albert-office))
(define gerry-office  (make-place 'gerry-office))

(define albert (make&install-person 'albert albert-office 3))
(define gerry  (make&install-person 'gerry  gerry-office  2))
```

All objects in the system are implemented as message-accepting procedures.

Once you load the system in the laboratory, you will be able to control `albert` and `gerry` by sending them appropriate messages. As you enter each command, the computer reports what happens and where it is happening. For instance, imagine we had interconnected a few places so that the following scenario is feasible:

```
(ask albert 'look-around)
At albert-office : albert says -- I see nothing
;Value: #f

(ask (ask albert 'place) 'exits)
;Value: (west down)

(ask albert 'go 'down)
albert moves from albert-office to tech-square
;Value: #t

(ask albert 'go 'south)
albert moves from tech-square to building-36
;Value: #t

(ask albert 'go 'up)
albert moves from building-36 to computer-lab
;Value: #t

(ask gerry 'look-around)
at gerry-office : gerry says -- I see nothing
;Value: #f

(ask (ask gerry 'place) 'exits)
;Value: (down)

(ask gerry 'go 'down)
gerry moves from gerry-office to albert-office
;Value: #t

(ask gerry 'go 'down)
gerry moves from albert-office to tech-square
;Value: #t

(ask gerry 'go 'south)
gerry moves from tech-square to building-36
;Value: #t

(ask gerry 'go 'up)
gerry moves from building-36 to computer-lab
At computer-lab : gerry says -- hi albert
;Value: #t
```

In principle, you could run the system by issuing specific commands to each of the creatures in the world, but this defeats the intent of the game since that would give you explicit control over

all the characters. Instead, we will structure our system so that any character can be manipulated automatically in some fashion by the computer. We do this by creating a list of all the characters to be moved by the computer and by simulating the passage of time by a special procedure, `clock`, that sends a `move` message to each creature in the list. A `move` message does not automatically imply that the creature receiving it will perform an action. Rather, like all of us, a creature hangs about idly until he or she (or it) gets bored enough to do something. To account for this, the third argument to `make-person` specifies the average number of clock intervals that the person will wait before doing something (the restlessness factor).

Before we trigger the clock to simulate a game, let's explore the properties of our world a bit more.

First, let's create a `computer-manual` and place it in the `computer-lab` (where `albert` and `gerry` now are).

```
(define computer-manual (make&install-thing 'computer-manual computer-lab))
```

Next, we'll have `albert` look around. He sees the manual and `gerry`. The manual looks useful, so we have `albert` take it and leave.

```
(ask albert 'look-around)
At computer-lab : albert says -- I see computer-manual gerry
;Value: (computer-manual gerry)

(ask albert 'take computer-manual)
At computer-lab : albert says -- I take computer-manual
;Value: #t

(ask albert 'go 'down)
albert moves from computer-lab to building-36
;Value: #t
```

`Gerry` had also noticed the manual; he follows `albert` and snatches the manual away. Angrily, `albert` sulks off to the EGG-Atrium:

```
(ask gerry 'go 'down)
gerry moves from computer-lab to building-36
At building-36 : gerry says -- Hi albert
;Value: #t

(ask gerry 'take computer-manual)
At building-36 : albert says -- I lose computer-manual
At building-36 : albert says -- yaaaah! I am upset!
At building-36 : gerry says -- I take computer-manual
;Value: #t

(ask albert 'go 'west)
albert moves from building-36 to egg-atrium
;Value: #t
```

Unfortunately for `albert`, beneath the EGG-Atrium is an inaccessible dungeon, inhabited by a troll named `grendel`. A troll is a kind of person; it can move around, take things, and so on. When a troll gets a `move` message from the clock, it acts just like an ordinary person—unless someone else is in the room. When `grendel` decides to `act`, it's game over for `albert`:

```
(ask grendel 'move)
grendel moves from dungeon to egg-atrium
At egg-atrium : grendel says -- Hi albert
;Value: #t
```

After a few more moves, `grendel` acts again:

```
(ask grendel 'move)
At egg-atrium : grendel says -- Growl.... I'm going to eat you, albert
At egg-atrium : albert says --
                   Dulce et decorum est
                   pro computatore mori!
albert moves from egg-atrium to heaven
At egg-atrium : grendel says -- Chomp chomp. albert tastes yummy!
;Value: *burp*
```

Implementation The simulator for the world is contained in one file, which is attached. It contains code that creates all procedures and classes, initializes our particular imaginary world and installs `albert`, `gerry`, and `grendel`.

Part 2: Theoretical understanding

This part contains multiple theoretical question to better understand the basic setup of the environment. Although they are not examined by the CMS, answering and understanding them is vital to solve the programming exercise.

Question 1: Draw a simple inheritance diagram showing all the kinds of objects (classes) defined in the adventure game system, the inheritance relations between them, and the methods defined for each class.

Question 2: Draw a simple map showing all the places created by evaluating `tester4.scm`, and how they interconnect. You will probably find this map useful in dealing with the rest of the problem set.

Question 3: Suppose we evaluate the following expressions:

```
(define ice-cream (make-thing 'ice-cream dormitory))
(ask ice-cream 'set-owner gerry)
```

At some point in the evaluation of the second expression, the expression

```
(set! owner new-owner)
```

will be evaluated in some environment. Draw an environment diagram, showing the full structure of `ice-cream` at the point where this expression is evaluated. Don't show the details of `gerry` or `dormitory`—just assume that `gerry` and `dormitory` are names defined in the global environment that point off to some objects that you draw as blobs.

Question 4: Suppose that, in addition to `ice-cream` in question 3, we define

```
(define rum-and-raisin (make-named-object 'ice-cream))
```

Are `ice-cream` and `rum-and-raisin` the same object (i.e., are they `eq?`)? If `gerry` wanders to a place where they both are and looks around, what message will he print?

Exercises

Load `tester4.scm` into RACKET. Since the simulation model works by data mutation, it is possible to get your RACKET-simulated world into an inconsistent state while debugging. To help you avoid this problem, we suggest the following discipline: any procedures you change or define that aren't part of the exercise should be placed in your testing file `tester4.scm`; any new characters, objects or procedures asked by the exercises should be added to `ps4.scm`. This way whenever you change some procedure you can make sure your world reflects these changes by simply re-evaluating the entire `tester4.scm` file. Finally, to save you from retyping the same scenarios repeatedly—for example, when debugging you may want to create a new character, move it to some interesting place, then ask it to act—we suggest you define little test "script" procedures at the end of `tester4.scm` which you can invoke to act out the scenarios when testing your code. See the comments in `tester4.scm` for details.

For submission you only need to change the `'<YOUR-CODE-HERE>` parts of the `ps4.scm`.

After loading the system, make `albert` and `gerry` move around by repeatedly calling `clock` (with no arguments). Check that everything works so far.

Exercise 0: Define a procedure `make-flip` that can be used to generate `flip` procedures. A `flip` procedure (with no parameters) returns 1 the first time it is called, 0 the second time it is called, 1 the third time, 0 the fourth time, and so on. That is, we should be able to write `(define flip (make-flip))`.

Exercise 1: Make and install a new character, yourself, with a high enough threshold (say, 100) so that you have "free will" and are not likely to be moved by the clock. Name your character `myself`. Place yourself initially in the `dormitory`. Also make and install a thing called `late-homework`, so that it starts in the `dormitory`. Pick up the `late-homework`, find out where `gerry` is, go there, and try to get `gerry` to take the homework even though he is notoriously adamant in his stand against accepting tardy problem sets. Can you find a way to do this that does not leave *you* upset? (Watch out for `grendel`!)

Student Disservice Cards

The MIT Office against Student Affairs has asked us for help in expanding the features offered by the MIT "Student Disservice Card" system. Luckily, our object-oriented simulation is just what's needed for trying out new ideas.

To model a student disservice card, we can make a new kind of object, called an `sd-card`, which is a special kind of thing. Besides inheriting the standard properties of a thing, each sd-card has some local state: an `id`, which identifies the person to whom the card was issued. An sd-card supports a message `sd-card?`, indicating that it is an sd-card. The card also accepts message that returns the `id`.

The procedure `make&install-sd-card` (shown below) can be used to make a card and install it. It uses the procedure `make-sd-card` to actually create the card. Note that both procedures take a name, an initial place and an id (which should be a symbol).

```
(define (make&install-sd-card name birthplace id)
  (let ((card (make-sd-card name birthplace id)))
    (ask card 'install)
    card))

(define (make-sd-card name birthplace idnumber)
  (let ((id idnumber)
        (thing (make-thing name birthplace)))
    (lambda (message)
      (cond ((eq? message 'sd-card?) (lambda (self) true))
            ((eq? message 'id) (lambda (self) id))
            (else (get-method thing message))))))
```

Note the presence of the `sd-card?` method, which identifies the object as an sd-card. In general, our system is structured so that a recognizable `foo` must have a `foo?` method that answers `true`. Objects that aren't `foo`s don't have a `foo?` method. We've supplied a procedure called `is-a` that can be used to test whether an object is of some particular type. `Is-a` works like `ask` except that if the message doesn't correspond to a method it returns `false` rather than causing an error. For example, you can test whether an object is an sd-card by evaluating `(is-a object 'sd-card?)`.

Exercise 2: A person may move to a new place only if the place returns `true` in response to the message `accept-person?`, for example

```
(ask gerry-office 'accept-person? gerry)  => #T
```

This is trivially true of ordinary places. Make a new kind of place that will accept a person only if they are carrying an `sd-card` by completing the procedure below:

```
(define (make-card-locked-place name)
  (let ((place (make-place name)))
    (lambda (message)
      (cond ((eq? message 'accept-person?)
             ...)
            (else (get-method place message))))))
```

Exercise 3: The Director of Housing has ordered that student residences can be entered only by students living at that residence. He has decided to enforce his policy by securing each student residence with a card lock that opens only for cards with a registered id.

Create a new class of place, a `student-residence`, which implements this policy. A student-residence should keep a list of card id-numbers (not the cards themselves in case a student loses their card and has to get a replacement). In addition to the `accept-person?` method we need a `register-card` method which adds the card's id to the list. `Register-card` should register the card only if the card is already inside the residence. This way we can create cards with the residence as their 'birth place' and register those cards. Only those cards will let us back in.

Exercise 4: There has been a spate of card thefts on campus recently. Obviously these criminals need to be sorted out. Create a new kind of person called an `ogre`, which is like a troll but only eats people who are carrying a card which has been reported stolen. You can use `grendel` as an example of a being that does special things when asked to act. To make the ogres especially effective they should have a low restlessness factor (equal with 1).

Write a procedure `(report-stolen-card id)` which creates and dispatches a new ogre to hunt down the felon. Naturally, the ogre should start its hunt from the dungeon. The name of the ogre should be ogre followed by the card id (i.e if `(report-stolen-card '123-456)` is called then the ogre name should be `ogre123-456`. The procedure must return the ogre object.

Exercise 5: Ace hackers Ben Bitdiddle and Alyssa P. Hacker find the new card locks on the student residences a nuisance. It is difficult to get together to do problem sets and their friends who used to drop by to chat never do so anymore because they can't get in without a valid card. Luckily, the cards are easy to duplicate and distribute to friends.

To discourage the use of duplicate cards the Director of Housing has decided to monitor the use of cards. If a card is used in the same place at the same time then one of the copies must be forged.

Implement a new object, `big-brother`, which accepts a message `inform` which takes a card-id and a place. `Big-brother` should monitor all the information to detect forged cards and report them by calling `report-stolen-card`. The time is available from the procedure `(current-time)`. Create a new class `surveillance-room` so that they inform `big-brother` when someone gains access with a card. `big-brother` also accepts a message `display-stolen-card` that returns a list with all the reported stolen cards.

Exercise 6: Now you have the elements of a simple game that you play by interspersing your own moves with calls to the clock.

To make the game more interesting, you should also set up a new class `secret` that extends `thing` by overriding `set-owner`, so that when someone takes it, a new passage is opened between dormitory (north) and Tech-Square (east). It should also make the person that picks it up say `A new passage opens north!`. Then install an object called `suspicious-book` of this type in the dormitory.