6.001 Tutorial 9 Notes

TA: Gerald Dalley
11–12 Apr 2005

Announcements

- The tutor's version of the online lectures use an old version of the object system.
- OOP II Lecture slide 30 is incorrect.
- Problem Set 8 due on Tuesday. Note that the object systems used in the problem set are older, simpler, and less powerful versions compared to the one in project 4 (despite what the intro text says in PS.8.1.1).
- Email me (`dalleyg@mit.edu`) your design plan for project 4 by 6PM on Wednesday 13 April 2005 (or earlier!).
- Project 4 is due before 6PM on 15 April. See the clarifications on the web and in your email.
- No class/tutorials/problem sets on 18–20 April. If you'd like a review session for the quiz, email me your availability on the 18th, 19th, and 20th so I can schedule a few of them.
- Quiz 2: Wednesday 20 April, 7:30-9:30PM, in 32-123 and 34-101.

Object-Oriented Programming

The High Level

Another "paradigm" for thinking about programming (like "functional programming" or "imperative programming"). Not necessarily better, just different. The idea is to combine the data and the operations performed on that data into one blob. Then, all the data and operations are combined, which makes extending the system easier later on.

Note to Java and C++ programmers: all member variables are are *private*.

In *our* object-oriented system, the implementation is based on *messages*. To request that an object does something, the `ask` procedure first finds the method, then it evaluates that method based on the rest of its input. For example, `(ask car 'park 'carefully)` searches `car` for a method tagged `park`, then evaluates it with `'carefully` as its only argument.

In our system, we have `foo` and `create-foo` procedures. The `foo` procedure defines how the object works — the methods it has, the state variables that are part of the object. It is sometimes called a "maker." The `create-foo` procedure builds an instance of the object — an actual piece of data that is that type of object.

Objects have the nice property that they can *inherit* from other objects. For example, a `person` is a `mobile-thing`, which is a `thing`, which is a `named-object`, which is a `root-object`. By inheriting, you can define new objects that have different behaviors, but only write a small bit of code.

The Low Level

In a little bit, we'll see what the internal representation is for an object. This year's representation is quite a bit more complex than in previous terms.

- **Instance:** a list containing the tag `instance` and a pointer to the message handler. Example: `(create-place '10-250)` returns a `place` instance.
- **Message handler:** a procedure that works in conjunction with `ask` to find and evaluate methods. The message handler's environment keeps (1) a list of the methods for the instance's class and (2) a list of all the superclasses of the instance's class.
- **Method:** a procedure created by a maker that can be evaluated by the message handler. It has access to parameters of the maker and to any local state for the instance (created by `let` statements in the maker. Example: `ADD-EXIT` is a method of `place`.
- **Maker:** a procedure used to set up a new instance. A maker procedure always takes an argument called `self` that is a pointer to the instance. You should never call the maker directly. Example: `place`.
- **Constructor:** a thin wrapper around the maker that uses `create-instance` to create a new instance, make it into the desired type of object, and install it in the world (if the instance has a method called `INSTALL`). Example: `create-place`.

The point of all this is to have one piece of data that represents the object itself — the `instance` *is* the object.

Warmup Exercise 1

Consider the expression

```
(ask (ask me 'location) 'name)
```

What kind of value does `(ask me 'location)` return here?

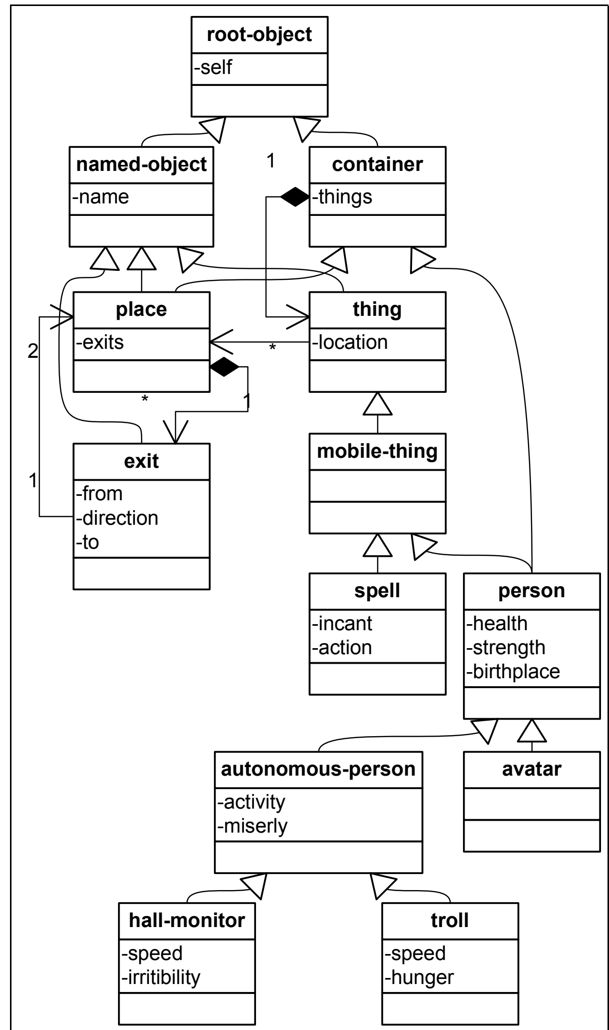
an instance of type `place`.

What other messages, besides `name`, can you send to this value?

`place`: `EXIT, EXIT-TOWARDS, ADD-EXIT`
`container`: `THINGS, HAVE-THING?, ADD-THING, DEL-THING`
`named-object`: `NAME, INSTALL, DESTROY`
`root-object`: `IS-A, TYPE, METHODS`

Warmup Exercise 2

Look through the code in `objtypes.scm` to discover which classes are defined in this system and how the classes are related. For example, place is a subclass of `named-object`. Also look through the code in `setup.scm` to see what the world looks like. Draw a class diagram like the ones presented in lecture. You will find such a diagram helpful (maybe indispensable) in doing the programming assignment.



Warmup Exercise 4

Aside from you, the avatar, what other characters roam this world?

The autonomous people include ben-bitdiddle, alyssa-hacker, course-6-frosh, and lambda-man. dr-evil and mr-bigglesworth are monitors. grendel and registrar are trolls.

What sorts of things are around?

Spells, blackboard, lovely-trees, flag-pole, ...

How is it determined which room each person and thing starts out in?

Spells are in chamber-of-stata, everything else is random

Warmup Exercise 5

Create an environment diagram corresponding to the evaluation of tt (define my-foo (thing 'foo some-location)). Warning: this environment diagram can get out of hand, and we want you to use this exercise to get a sense of how the system works. So, don't worry about the value bound to some-location, just draw it as a blob. Similary, don't worry about showing the object bound to maker. For the bindings associated with methods, just leave the actual value blank. Once you have drawn your environment model, draw boxes around the structures that correspond to each of the superparts of the object created.

> Solutions are posted at:
> `http://people.csail.mit.edu/ dalleyg/6.001/tutorial09warmup5.png`
> `http://people.csail.mit.edu/ dalleyg/6.001/tutorial09warmup5.pdf`
> `http://people.csail.mit.edu/ dalleyg/6.001/tutorial09warmup5.vsd`

Hippogryph

The hippogryph, living far beyond the seas in the Rhipaean Mountains, is the result of the rare breeding of a male gryphon and a filly. It has the head, wings and front legs of a gryphon, and the back and hind legs of a horse. It is a large powerful creature that can move through the air more swiftly than lightning. It figured in several of the legends of Charlemagne as a mount for some of the knights.

(`http://webhome.idirect.com/~donlong/monsters/Html/Hippogry.htm`)

We are going to design a hippogryph creature to add to our world. A hippogryph is a temperamental creature. When being friendly, it likes to pick up people and give them a ride. When being unfriendly, it ejects its current rider, or if there is no rider, it eats a nearby person.

What is a good super-class?

> The two most natural superclasses are `person` and `autonomous-person`. `person` is probably a better fit though since, when we look at the details, we'll find we have to override just about everything new in `autonomous-person`.

How can we make sure that the rider stays on the hippogryph and can't wander off on its own?

> We can give the hippogryph a private variable for its mount that is a place with no exits. Our rider sits in that place.

```scheme
(load "objsys.scm") (load "objtypes.scm") (load "setup.scm")
(setup 'dalleyg)

;; Utility procedure that finds a room based on its name
;; Assumes setup has been run.
(define (find-room room-name)
  (let ((matches (filter (lambda (room) (eq? room-name (ask room 'NAME)))
                         all-rooms)))
     (if matches (car matches) #f)))
;; Utility procedure that finds all people in a room based
;; on its the room's name.
;; Assumes setup has been run
(define (whos-in-room room-name)
  (map (lambda (person) (ask person 'NAME))
       (filter (lambda (thing) (ask thing 'IS-A 'person))
               (ask (find-room room-name) 'THINGS))))

;; HIPPOGRYPH Constructor
;;
;; A hippogryph is a tempermental creature.  When being friendly,
;; it likes to pick up people and give them a ride.  When being
;; unfriendly, it ejects its rider, or if there is no rider, it
;; finds someone to eat.
```

```
;;
;; name          : symbol
;; birthplace    : place
;; activity      : integer (max distance to move in a turn)
;; friendliness : flonum (fraction of time to not eject or eat people)
(define (create-hippogryph name birthplace activity friendliness)
  (create-instance hippogryph name birthplace activity friendliness))

;; Maker for a hippogryph.  See constructor for documentation
(define (hippogryph self name birthplace activity friendliness)
  (let* ((person-part (person self name birthplace))
         (mount-name (string->symbol
                      (string-append (symbol->string name) "-mount")))
         (mount (create-place mount-name)))

    (make-handler
     'hippogryph
     (make-methods

      'INSTALL
      (lambda ()
        (ask person-part 'INSTALL)
        (ask clock 'ADD-CALLBACK
             (create-clock-callback 'move-about self 'MOVE-ABOUT)))

      ; Moves about randomly decides whether to be friendly or not.
      ; If being friendly and is carrying a person on its mount, it
      ; continues to carry the person.  If nobody is mounted, it
      ; randomly selects someone in the room to put on its mount.
      ; If being unfriendly and is carrying someone, it dismounts
      ; that person.  If nobody is mounted, it randomly eats someone.
      'MOVE-ABOUT
      (lambda ()
        (let loop ((moves (random-number activity)))
          ; First move
          (if (= moves 0)
              'done-moving
              (begin
                (ask self 'MOVE-SOMEWHERE)
                (loop (- moves 1)))))
        ; then decide what to do
        (if (> (random 1.) friendliness)
            ; do the mean thing
            (begin
              (ask self 'HAVE-FIT)
              (if (ask mount 'THINGS)
                  (ask self 'EJECT-RIDER)
                  (ask self 'EAT-SOMEONE)))
            (begin
              (ask self 'SAY (list "I feel friendly"))
              (if (not (ask mount 'THINGS))
                  (ask self 'PICKUP-RIDER)))))

      ;; Pick a random exit from the current location and go there.
      'MOVE-SOMEWHERE
      (lambda ()
        (let ((exit (random-exit (ask self 'LOCATION))))
          (if (not (null? exit)) (ask self 'GO-EXIT exit))))

      ;; Drop off a rider.  Must have a rider to work properly.
      'EJECT-RIDER
      (lambda ()
        (let ((rider (car (ask mount 'THINGS))))
          (ask self 'EMIT (list (ask rider 'NAME) " is being ejected"))
          (ask rider 'CHANGE-LOCATION (ask self 'LOCATION)))
        'unburdened)
```

```
        ;; Eat someone random in the same room if there is anyone around.
        'EAT-SOMEONE
        (lambda ()
          (let ((people (ask self 'PEOPLE-AROUND)))
            (if people
                (let ((victim (pick-random people)))
                  (ask self 'EMIT
                       (list (ask self 'NAME) "takes a bite out of"
                             (ask victim 'NAME)))
                  (ask victim 'SUFFER (random-number 3) self)
                  'tasty)
                (begin
                  (ask self 'EMIT
                       (list (ask self 'NAME)
                             "glowers petulantly at the sky"))
                  'left-alone))))

        ;; Randomly pick up someone in the room and make them a rider
        'PICKUP-RIDER
        (lambda ()
          (let ((people (ask self 'PEOPLE-AROUND)))
            (if people
                (let ((rider (pick-random people)))
                  (ask self 'EMIT
                       (list (ask rider 'NAME) "is being picked up"))
                  (ask rider 'CHANGE-LOCATION mount)
                  'rider-added)
                (begin
                  (ask self 'SAY (list "There's nobody around "
                                       "to be picked up."))
                  'lonely))))

        ;; Invert the INSTALL process
        'DIE
        (lambda (perp)
          (ask clock 'REMOVE-CALLBACK self 'move-about)
          (let ((riders (ask mount 'THINGS)))
            (if riders (for-each (lambda (rider) (ask rider 'DIE perp))
                                 riders)))
          (ask person-part 'DIE perp))
        )
      person-part)))

; Test: create a hippogryph and install it in the world
(create-hippogryph 'buck-beak (pick-random all-rooms) 6 .5)

; Create a hippogryph that will always try to hold someone
; so we can make sure the rider is handled properly when
; this hippogryph is eaten.
(create-hippogryph '******friendly-gryff****** (pick-random all-rooms) 1 1.)

; Create a bunch of trolls that will increase the likelihood of
; someone eating the hippogryph
(let loop ((num-to-create 50))
  (if (> num-to-create 0)
      (begin
        (let ((name (string->symbol
                     (string-append "auto-troll-"
                                    (number->string num-to-create)))))
        (create-troll name (pick-random all-rooms) 5 1)
        (loop (dec num-to-create))))))

#|
(create-troll 't1 (pick-random all-rooms) 5 .75)
(create-troll 't2 (pick-random all-rooms) 5 .75)
(create-troll 't3 (pick-random all-rooms) 5 .75)
(create-troll 't4 (pick-random all-rooms) 5 .75)
```

```
(create-troll 't5 (pick-random all-rooms) 5 .75)
(create-troll 't6 (pick-random all-rooms) 5 .75)
(create-troll 't7 (pick-random all-rooms) 5 .75)
(create-troll 't8 (pick-random all-rooms) 5 .75)
(create-troll 't9 (pick-random all-rooms) 5 .75)
|#

; Run the clock for a while
(run-clock 2)
```