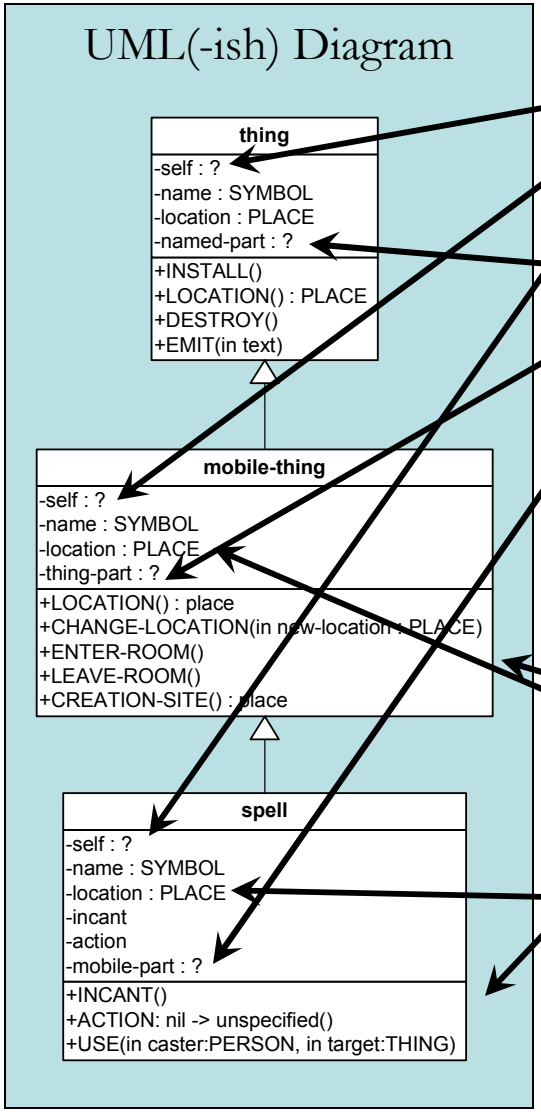
6.001 Recitation: Object-Oriented Systems

13 April 2005

By Gerald Dalley

spell
-self ? -name SYMBOL -location PLACE -incant -action -mobile-part ?
+INCANT() +ACTION nil -> unspecified() +USE(in caster PERSON in target THING)

Abstract View of Objects



Q: What is the type of *self*?

A: an *instance*

Q: What is the type of *named-part*?
thing-part? *mobile-part*?

A: They are all message handler procedures

Q: What happens when we:

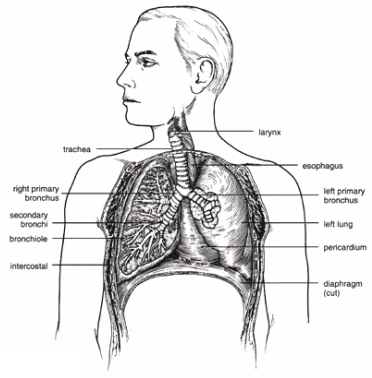
```
(ask my-spell 'CHANGE-LOCATION some-location)
```

A: 1) we look in *spell* for *CHANGE-LOCATION*
2) we look in *mobile-thing* for *CHANGE-LOCATION*
3) We retrieve *mobile-thing*'s *location*

Q: When is *spell*'s *location* variable used?

A: only in *spell*'s maker
(when it's passed to the *mobile-thing* maker)

Anatomy of a Class




```
;;
;; spell
;;
(define (create-spell name location incant action)
  (create-instance spell name location incant action))

(define (spell self name location incant action)
  (let ((mobile-part (mobile-thing self name location)))
    (make-handler
     'spell
     (make-methods
      'INCANT
      (lambda () incant)
      'ACTION
      (lambda () action)
      'USE
      (lambda (caster target) (action caster target)))
     mobile-part)))
```

Constructor

Maker

Make super parts & internal state

Make handler

Method definitions

Link up super parts

Note: chest anatomy image borrowed from http://www.usctransplant.org/lung/aboutourprogram.html

Practice: Diagram Given Code

```
(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)))
```

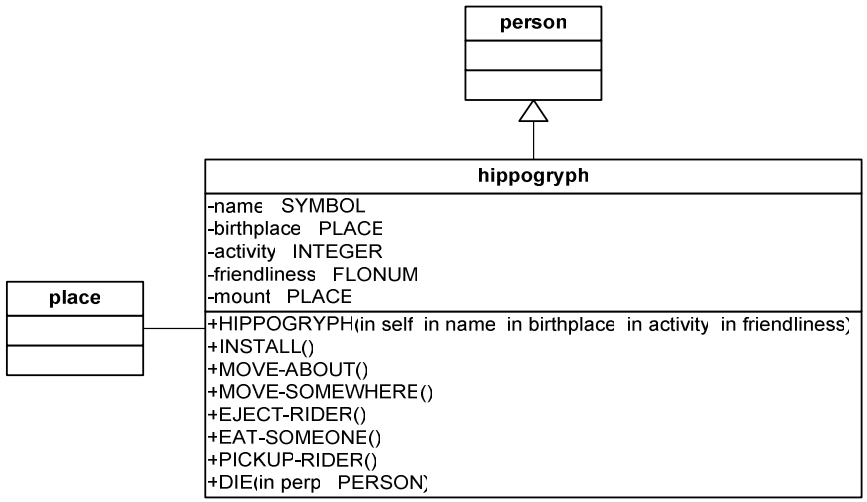
Practice: Code Given Diagram

- Sketch out the constructor and maker code
- For methods, just write the shell of it, *e.g.* for

  `DO-FOO(victim : PERSON)`

  put the following in the make-methods portion:

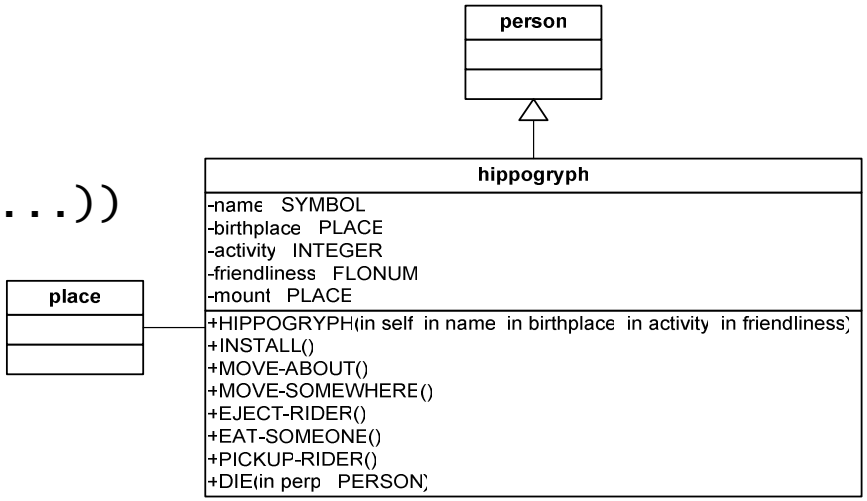
  `'DO-FOO (lambda (victim) ...)`



Practice: Code Given Diagram

```
(define (create-hippogryph name birthplace activity friendliness)
  (create-instance hippogryph name birthplace activity friendliness))

(define (hippogryph self name birthplace activity friendliness)
  (let* ((person-part (person self name birthplace))
         ...
         (mount (create-place ...)))
    (make-handler
     'hippogryph
     (make-methods
      'INSTALL         (lambda () ...)
      'MOVE-ABOUT      (lambda () ...)
      'MOVE-SOMEWHERE  (lambda () ...)
      'EAT-SOMEONE     (lambda () ...)
      'PICKUP-RIDER    (lambda () ...)
      'DIE             (lambda (perp) ...))
     person-part)))
```



Design & Implementation Practice

- We are going to implement a few new classes in the project 4 world. As you go, ask yourself these questions:
  - What should class or classes should it inherit from?
  - What new fields does it need?
  - What new methods does it need?
  - What methods from its superclass(es) should it override? Should the behavior of its overridden methods completely replace what the superclass does, or just augment it? How should superclass methods be called from an overriding method?

  Hints are provided so you don't have to hunt around too much.

- A **grumpy troll** attacks someone whenever it has a fit.
  (Hint: a person can HAVE-FIT, a troll can EAT-PEOPLE)
- 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.
  (Hint: a person can SAY something.)
- An **enchanted person** is a person that wanders around and IS-A spell. Whenever the enchanted person has a fit, it should cast itself on everyone else in the room.
  (Hint: think about multiple-inheritance issues)

Note: grumpy-troll, bomb, and recorder were adapted from Rob Miller's "Inheritance and Delegation" recitation http://people.csail.mit.edu/people/rcm/6001/

grumpy-troll

- A **grumpy troll** eats someone whenever it has a fit.
  (Hint: a person can HAVE-FIT, a troll can EAT-PEOPLE)

```
(define (grumpy-troll self name birthplace speed hunger)
  (let ((troll-part (troll self name birthplace speed hunger)))
    (make-handler
      'grumpy-troll
      (make-methods
        'HAVE-FIT
        (lambda ()
          (ask troll-part 'HAVE-FIT)
          (ask self 'EAT-PEOPLE)))
      troll-part)))
```

- Why did we have to use `(ask troll-part 'HAVE-FIT)`, instead of just `(ask self 'HAVE-FIT)`?

Note: troll image borrowed from http://www.nofrag.com/2003/aou/18/7138/

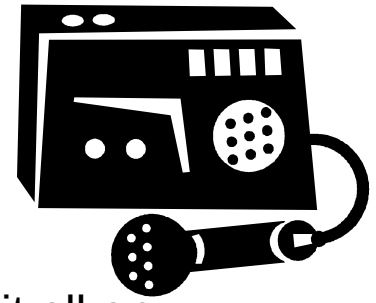
bomb

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

```
(define (bomb self name birthplace)
  (let ((mobile-thing-part (mobile-thing self name birthplace)))
    (make-handler
      'bomb
      (make-methods
        'EXPLODE
        (lambda ()
          (for-each (lambda (thing)
                      (if (ask thing 'IS-A 'person)
                          (ask thing 'DIE self)
                          (ask thing 'DESTROY)))
                    (ask (ask self 'LOCATION) 'THINGS))))
      mobile-thing-part)))
```

- In the `for-each` expression, why didn't we just write
  `(ask self 'THINGS)`
  to find out what things are around us, since we already have a local variable with our location in it?

recorder

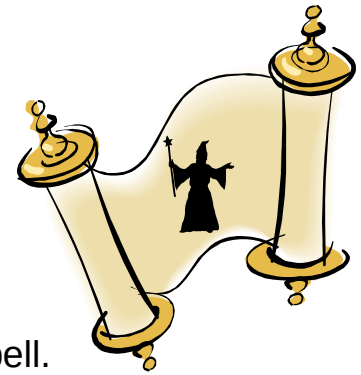


- A **recorder** remembers everything it ever said and can replay it all on command.
  (Hint: a person can SAY something.)

```
(define (recorder self name birthplace)
  (let ((person-part (person self name birthplace))
        (recording '()))
    (make-handler
      'recorder
      (make-methods
        'SAY
        (lambda (list-of-stuff)
          (set! recording (cons list-of-stuff recording))
          (ask person-part 'SAY list-of-stuff))
        'REPLAY
        (lambda ()
          (ask person-part 'SAY (reverse recording))))
      person-part)))
```

- Why did we have to use `(reverse recording)`?
- Why did we have to use `(ask person-part 'SAY ...)`, instead of just `(ask self 'SAY ...)` in `REPLAY`?

enchanted-person



- An **enchanted person** is a person that automatically wanders around and IS-A spell. Whenever the enchanted person has a fit, it should cast itself on everyone else in the room. *(Hint: think about multiple-inheritance issues)*

```
(define (enchanted-person self name birthplace activity miserly
                          incant action)
  (let ((auto-part (autonomous-person self name birthplace
                                      activity miserly))
        (spell-part (spell self name birthplace incant action)))
    (make-handler
     'enchanted-person
     (make-methods
      'HAVE-FIT
      (lambda ()
        (ask auto-part 'HAVE-FIT)
        (for-each (lambda (victim) (ask spell-part 'USE self victim))
                  (ask self 'PEOPLE-AROUND))))
     auto-part spell-part)))
```

- Why don't we have to override `LOCATION` and `CHANGE-LOCATION`?
- What happens if make the `action` be `(lambda (caster target) (set! health (+ health 10)))`?

monk

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

```
(define (monk self name birthplace)
  (let ((person-part (person self name birthplace)))
    (make-handler
      'monk
      (make-methods
        'TAKE
        (lambda (thing)
          (ask self 'SAY (list "Possessions are fleeting. I cannot take"
                               (ask thing 'NAME))))
      person-part)))
```

Note: monk image borrowed from http://www.hasslefreeclipart.com/cart_fantasy/page1.html