

**MASSACHUSETTS INSTITUTE OF TECHNOLOGY**  
**Department of Electrical Engineering and Computer Science**  
**6.001 & Structure and Interpretation of Computer Programs**  
**Fall Semester, 2005**

## **Project 4 - The Object-Oriented Adventure Game**

- Issued: Monday, November 14<sup>th</sup>
- Warm-up Exercises: You may be asked to show this work in tutorial on November 21<sup>st</sup> or 22<sup>nd</sup>
- Design Plan: Your plan for the design exercise should be emailed to your TA by Wednesday, November 23<sup>rd</sup> before 6:00 pm, at the latest (we encourage you to do this earlier!)
- Project Due: Friday, December 2<sup>nd</sup> before 6:00 pm
- Code to load for this project:
  - Link to the system code file `objsys.scm` is provided from the Projects link on the course web page.

You should begin working on the assignment once you receive it. It is to your advantage to get work done early, rather than waiting until the night before it is due. You should also read over and think through each part of the assignment (as well as any project code) before you sit down at the computer. It is generally much more efficient to test, debug, and run a program that you have thought about beforehand, rather than doing the planning "online." Diving into program development without a clear idea of what you plan to do generally causes assignments to take much longer than necessary.

**Word to the wise:** This project is difficult. The trick lies in knowing which code to write, and for that you must understand the project code, which is considerable. You'll need to understand the general ideas of object-oriented programming and the implementation provided of an object-oriented programming system (in the first part of `objsys.scm`). Then you'll need to understand the particular classes (in the second part of `objsys.scm`) and the world (in the last part of `objsys.scm`) that we've constructed for you. In truth, this assignment is much more an exercise in reading and understanding a software system than in writing programs, because reading significant amounts of code is an important skill that you must master. The warmup exercises will require you to do considerable digesting of code before you can start on them. And we **strongly urge** you to study the code before you try the programming exercises themselves. Starting to program without understanding the code is a good way to get lost, and will virtually guarantee that you will spend more time on this assignment than necessary.

In this project we will develop a powerful strategy for building simulations of possible worlds. The strategy will enable us to make modular simulations with enough flexibility to allow us to expand and elaborate the simulation as our conception of the world expands and becomes more detailed.

One way to organize our thoughts about a possible world is to divide it up into discrete objects, where each object will have a behavior by itself, and it will interact with other objects in some lawful way. If it is useful to decompose a problem in this way then we can construct a computational world, analogous to the "real" world, with a computational object for each real object.

Each of our computational objects has some independent local state, and some rules (or code), that determine its behavior. One computational object may influence another by sending it messages and invoking methods in the other. The program associated with an object describes how the object reacts to messages and how its state changes as a consequence.

You may have heard of this idea in the guise of "Object-Oriented Programming systems"(OOPs!). Languages such as C++ and Java are organized around OOP. While OOP has received a lot of attention in recent years, it is only one of several powerful programming styles. What we will try to understand here is the essence of the idea, rather than the incidental details of their expression in particular languages.

## **An Object System**

Consider the problem of simulating the activity of a few interacting agents wandering around different places in a simple world. Real people are very complicated; we do not know enough to simulate their behavior in any detail. But for some purposes (for example, to make an adventure game) we may simplify and abstract this behavior. In particular, we can use objects to capture common state parameters and behaviors of things, and can then use the message-passing paradigm to control interaction between objects in a simulation.

Let's start with the fundamental stuff first. We can think of our object oriented paradigm as consisting of classes and instances. A class can be thought of as the "template" for how we want a particular kind of object to behave. The way we define the class of an object is with a basic "make handler" procedure; this procedure is used with a "create instance" procedure which builds for us a particular instance. As you will see, when we examine the code, each class is defined by a procedure that when invoked will create some internal state (including instances of other class objects) and a message passing procedure (created by a "make handler") that returns methods in response to messages.

Our object instances are thus procedures which accept messages. An object will give you a method if you send it a message; you can then invoke that method (possibly with some arguments) to cause some action, state update, or other computation to occur.

The main pieces we will use in our code to capture these ideas are detailed as follows:

**Instance of an object** – each individual object has its own identity. The instance knows its type, and has a message handler associated with it. One can "ask" an object to do something, which will cause the object to use the message handler to

look for a method to handle the request and then invoke the method on the arguments.

**“Making” an object message handler** – each instance needs a new message handler to inherit the state information and methods of the specified class. The message handler is not a full “object instance” in our system; the message handler needs to be part of an instance object (or part of another message handler that is part of an instance object). All procedures that define classes should take a self pointer (a pointer to the enclosing instance) as the first argument.

**“Creating” an object** – the act of creation does three things: it makes a new instance of the object; it makes and sets the message handler for that instance; and finally it INSTALL’s that new object into the world.

**“Installing” an object** – this is a method in the object, by which the object can initialize itself and insert itself into the world, by connecting itself up with other related objects in the world.

Let’s look at these different elements in a bit more detail.

## Classes and Instances

Here is the template for a class definition in our object system. This is quite similar to the one introduced in lecture, but we have cleaned up the interface a little bit to make things easier to read:

```
(define (type self arg1 arg2 ... argn )
  (let ((super1-part (super1 self args)
        (super2-part (super2 self args)
          other superclasses
          other local state )
        (make-handler
          type
          (make-methods
            message-name-1 method-1
            message-name-2 method-2
            other messages and methods
          )
          super1-part super2-part ...)))
```

That form is a little mystifying (we have put some terms in *italics* to indicate that these would be replaced by specific instances), so let’s look at an example. In our simulation, almost everything is going to have a name, thus let’s create a `named-object` class:

```
(define (named-object self name)
  (let ((root-part (root-object self))))
```

```
(make-handler
  'NAMED-OBJECT ; name of the class
  (make-methods
    'NAME (lambda () name)
    'INSTALL (lambda () 'installed)
    'DESTROY (lambda () 'destroyed))
  root-part)))
```

**One key thing to be careful of – DrScheme is case sensitive (while MIT Scheme is not). This means that you need to be careful about “UPPER CASE” versus “lower case”. For readability, we have tried to use upper case for names of object types, and for names of methods.**

We can see that this class procedure defines a template for a class. It includes some state variables (both parameters required as part of the procedure application, as well as any internal state variables we want to create); and it creates a message handler for controlling instances of the objects. That is performed by invoking `make-handler`, which takes as input the type of the object, a set of message-method pairs, and any inherited superclasses. Note that the message-method pairs are a combination of a symbol and a procedure that will do something. Each such class procedure will be used to create instances (see below).

We have designed some conventions, which will be useful in following the code. We use the type of the object as the name of the procedure that defines a class (e.g. `named-object` in the above example). Note that this is also the first argument passed to the `make-handler` procedure.

The actual code for creating the handler is given by:

```
(define (make-handler typename methods . super-parts)
  (cond ((not (symbol? typename))
        ;check for possible programmer errors
        (error "bad typename" typename))
        ((not (method-list? methods))
         (error "bad method list" methods))
        ((and super-parts (not (filter handler? super-parts)))
         (error "bad part list" super-parts))
        (else
         (let ((handler
                (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)))))))))
      handler))))

```

If you look through this code (you don't need to understand all of it!) you can see that this procedure first checks for some error cases, and then in the general case creates a procedure that takes as argument a message and then checks that symbol against a set of cases. If it is the special case of `TYPE`, then it returns a list of the types of objects inherited by this class. If it is the special case of `METHODS`, it returns a list of method names of this class, followed by the method names inherited from associated superclasses. Otherwise it tries to look up the message in set of message-method pairs that were provided, and return the actual method. If there is no method for this particular class type, it then tries to look up the message in the set inherited from the superclasses.

Note that `make-methods` will build a list of (name, procedure) pairs suitable as input to `make-handler`.

```

(define (make-methods . args)
  (define (helper lst result)
    (cond ((null? lst) result)

          ; error catching
          ((null? (cdr lst))
           (error "unmatched method (name,proc) pair"))
          ((not (symbol? (car lst)))
           (if (procedure? (car lst))
               (pp (car lst))
               (error "invalid method name" (car lst))))
          ((not (procedure? (cadr lst)))
           (error "invalid method procedure" (cadr lst)))

          (else
           (helper (cddr lst)
                   (cons (list (car lst) (cadr lst)) result))))))
  (cons 'methods (reverse (helper args '()))))

```

This set of code is a slightly modified version of what was presented in lecture, but with the same overall behavior. Every *foo* procedure defining a class of type *foo* takes *self* as the first argument. This indicates of which instance the class handler is a part.

Returning to our definition of `named-object` we see that the second argument to `named-object` is `name`, which is part of the state of the `named-object`.

The `let` statement which binds `root-part` to the result of making a root-object, together with the `type-extend` usage inside the `type` method of `make-handler`, and the use of the `super-parts` at the end of the definition, all together tell us that named-objects are a *subclass* of root-object.

```
(define (root-object self)
  (make-handler
   'ROOT
   (make-methods
    'IS-A
    (lambda (type)
      (memq type (ask self 'TYPE))))))
```

The root object provides a basis for providing common behaviors to all classes. Specifically, it provides a convenient method (`IS-A`) to see if a type descriptor is in the `TYPE` list. We will by convention use this class as the root for all other classes.

Named-objects have no other local state than the `name` variable. They do have four methods: `TYPE`, `NAME`, `INSTALL`, and `DESTROY`. The `TYPE` method comes from the `make-handler` procedure and it indicates that named-objects have the type named-object in addition to any type descriptors that the `root-part` has. The `INSTALL` method is not required, but if it exists, it is called when an instance is created. In the case of named-object, there is nothing to do at creation time, but we'll later see examples where this method is non-trivial. The `NAME` is a *selector* in that it returns the name with which the object was created.

However, the `named-object` procedure only builds a *handler* for an object of type named-object. In order to get an *instance*, we need a `create-named-object` procedure:

```
(define (create-named-object name) ; symbol -> named-object
  (create-instance named-object name))
```

Here, an instance is created using the `named-object` procedure. The `create-instance` procedure builds a handler procedure that serves as a container for the real handler for the instance. It also attaches a tag or label to the handler, so that we know we are working with an instance. We need this extra complexity because each *foo* procedure expects `self` as an argument, so we build an instance object, then create the handler, passing the instance object in for `self`. You'll explore more of this system in the questions below.

## Using Instances

Once you have an instance, you can call the *methods* on it using the `ask` procedure:

```
(define book (create-named-object 'sicp))
```

```
(ask book 'NAME)
;Value: sicp
```

```
(ask book 'TYPE)
;Value: (NAMED-OBJECT ROOT)
```

The `ask` procedure retrieves the method of the given name from the instance, and then calls it. Retrieving a method from a handler is done with `get-method`, which ends up calling the handler procedure with the method name as the `message`. The specifics of the `ask` procedure and related procedures can be found in the code file.

## Inheritance and Subclasses

We've already built a class, `named-object`, that *inherited* from its parent class, `root-object`. If the handler for a `named-object` is sent a message that it doesn't recognize, it attempts to get a method from its parent (last line of `make-handler` procedure). Each handler creates a private handler for its parent to pass these messages to (the `let` statement in `named-object`). Because this parent handler is part of the same instance as the overall handler, the `self` value is the same in both.

However, let's move on to a subclass of `named-object` called a `thing`. A `thing` is an object that will have a location in addition to a name. Thus, we may think of a `thing` as a kind of named object except that it also handles the messages that are special to things. This arrangement is described in various ways in object-oriented jargon, e.g., "the `thing` class *inherits* from the `named-object` class," or "`thing` is a *subclass* of `named-object`," or "`named-object` is a *superclass* of `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)))
```

A very interesting (and confusing!) property of object-oriented systems is that subclasses can *specialize* or *override* methods of their superclasses. In lecture, you saw this with professors SAYING things differently than students. A subclass *overrides* a method on the *superclass* by supplying a method of the same name. For example, `thing` *overrides* the `INSTALL` method of `named-object`. When the user of the object tries to get the method named `INSTALL`, it will be found in `thing` and `thing`'s version of the method will be returned (because it never reaches the `else` clause in `make-handler` which checks the parent `named-object-part`). The `thing` class *overrides* two methods on `named-object` explicitly (as well as two implicitly); can you point out which two explicit methods are overridden?

One of the methods which `thing` overrides in an implicit manner is the `TYPE` method. This is one of the methods that every class is supposed to override, as it allows the class to include its *type descriptor* in the list of types that the object has. This allows the class of an instance to be discovered at run time:

```
(define building (create-thing 'stata-center MIT))

(ask building 'TYPE)
;Value: (THING NAMED-OBJECT ROOT)
```

There is a handy method on the root-object called `IS-A` that uses the `TYPE` method to determine if an object has a certain type:

```
(ask building 'IS-A 'THING)
;Value: #t

(ask building 'IS-A 'NAMED-OBJECT)
;Value: #t

(ask book 'IS-A 'THING)
;Value: #f

(ask book 'IS-A 'NAMED-OBJECT)
;Value: #t
```

You'll note that `building` is considered to be both a `thing` and a `named-object`, because even though it was built as a `thing`, things inherit from `named-object`.

### Using superclass methods

In the `thing` code, the `DESTROY` method uses `(ask self 'LOCATION)` in order to figure out where to remove itself from. However, it could have just referenced the `location` variable. It doesn't because one of the tenets of object-oriented programming is "**if there's a method that does what you need, use it.**" The idea is to re-use code as much

as is reasonable. (It turns out just using `location` would be a bug in this case; you'll be able to see why after doing the warm-up exercises!)

Some of the time, when you specialize a method, you want the subclass' method to do something completely different than the superclass. For example, the way massive-stars DIE (supernova!) is very different than the way stars DIE (burn out). However, the rest of the time, you may want to specify some *additional* behavior to the original. This presents a problem: how to call your superclass' method from within the overriding method. Following the usual pattern of `(ask self 'METHOD)` will give rise to an infinite loop! Thus, instead of asking `ourselves`, we ask our superclass-part. Note that we do this with the `INSTALL` method of a thing, where we explicitly ask the superpart to also install, as well as doing some specific actions. ***This is the only situation in which you should be asking your superclass-part!***

## Classes for a Simulated World

When you read the code in `objsys.scm`, you will see definitions of several different classes of objects that define a host of interesting behaviors and capabilities using the OOP style discussed in the previous section. Here we give a brief "tour" of some of the important classes in our simulated world.

### Container Class

Once we have `things`, it is easy to imagine that we might want containers for things. We can define a utility `container` class as shown below:

```
(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)))
```

Note that a container does not inherit from `named-object`, so it does not support messages such as `NAME` or `INSTALL`. Containers are not meant to be stand-alone objects (there's no `create-container` procedure); rather, they are only meant to be used

internally by other objects to gain the capability of adding things, deleting things, and checking if one has something.

## Place class

Our simulated world needs places (e.g. rooms or spaces) where interesting things will occur. The definition of the `place` class is shown below.

```
(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)))
```

If we look at the first and last lines of `place`, we notice that `place` inherits from two different classes: it has both an internal `named-part` and an internal `container-part`. If the place receives a message that doesn't match any of its methods, the get-method procedure will first check the `container-part` for the method, then use the `named-part`. This is generally called "multiple inheritance," which comes with a host of issues as discussed briefly in lecture. You'll note that `named-object` and `container` only share one method of the same name, `TYPE`, and `place` overrides it. The `TYPE` method calls the `type-extend` procedure with *both* parent-parts. Retrieving the type of a place:

```
(define stata (create-place 'stata-center))

(ask stata 'TYPE)
;Value: (PLACE NAMED-OBJECT ROOT CONTAINER)
```

You aren't guaranteed anything about the order of the type-descriptors except that the first descriptor in the list is the class that you instantiated to create the instance. You can also see that our `place` instances will each have their own internal variable `exits`, which will be a list of `exit` instances which lead from one place to another place. In our object-

oriented terminology, we can say the place class establishes a "has-a" relationship with the exit class (as opposed to the "is-a" relationship denoting inheritance). You should examine the `objsys.scm` file to understand the definition for exits.

### Mobile-thing class

Now that we have things that can be contained in some place, we might also want mobile-things that can CHANGE-LOCATION.

```
(define (create-mobile-thing name location)
  ; symbol, location -> mobile-thing
  (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)))
```

When a mobile thing moves from one location to another it has to tell the old location to DEL-THING from its memory, and tell the new location to ADD-THING. You'll note that the CHANGE-LOCATION method adds and removes the `self` from locations, thus the location contains a reference to the *instance* not the *handler*!

### Person class

A person is a kind of mobile thing that is also a container. The objective of the multiple inheritance is that people can "contain things" which they carry around with them when they move.

A person can SAY a list of phrases. A person can TAKE and DROP things. People also have a health meter which is reduced by SUFFERING. If a person's health reaches zero, they DIE. Some of the other messages a person can handle are briefly shown below; you should consult the full definition of the person class in `objsys.scm` to understand the full set of capabilities a person instance has.

```

(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)

      ...

      'TAKE
      (lambda (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)))

      ...
      )
     mobile-thing-part container-part)))

```

## Avatar class

One kind of character you will use in this project is an avatar. The avatar is a kind of person who must be able to do the sorts of things a person can do, such as TAKE things or GO in some direction. However, the avatar must be able to intercept the GO message, to do things that are special to the avatar, as well as do what a person does when it receives a GO message. This is again accomplished by asking the superclass-part.

```

(define (create-avatar name birthplace)
  ; symbol, place -> avatar
  (create-instance avatar name birthplace))

(define (avatar self name birthplace)
  (let ((person-part (person self name birthplace)))
    (make-handler
     'AVATAR
     (make-methods
      'LOOK-AROUND      ; report on world around you
      (lambda ()
        ...)
      'GO
      (lambda (direction) ; Shadows person's GO
        (let ((success? (ask person-part 'GO direction)))
          (if success? (ask our-clock 'TICK))
          success?))
      'DIE
      (lambda (perp)
        (ask self 'SAY (list "I am slain!"))
        (ask person-part 'DIE perp))))
     person-part)))

```

The avatar also implements an additional message, `LOOK-AROUND`, which you will find very useful when running simulations to get a picture of what the world looks like around the avatar.

## Clocks and Callbacks

In order to provide for the passage of time in our system, we have a global clock object, whose implementation may be found in `objsys.scm`. This class has exactly one instance which is created when `objsys.scm` is loaded and bound to the globally accessible variable `our-clock`. Unlike the real world, time passes only when we want it to, by asking the clock to `TICK`. The rest of the system finds out that time has passed because the clock informs them by sending them a message. However, not every object cares about time, so the clock only informs objects that have indicated to the clock that they care.

In order to hear about the passage of time, an object registers a *callback* with the clock. A *callback* is a promise to send a particular message to a particular object when the callback is activated. As with everything else in our system, a callback is an instance, in this case of the class `clock-callback`. Clock-callbacks are created with a name, an object, and a message. When a `clock-callback` is `ACTIVATED`, it sends the object the message (e.g. it does `(ask object message)`).

To register a callback with the clock, use `ADD-CALLBACK` to add your callback to the clock's list of callbacks. When the clock `TICKS`, it `ACTIVATES` every callback on its list. An example of the process, which registers a callback named `do-thingy` to invoke the `THINGY` method on the current object:

```
(ask our-clock 'ADD-CALLBACK
  (create-clock-callback 'do-thingy self 'THINGY))
```

Remember to remove callbacks (with `REMOVE-CALLBACK`) when the object should no longer be responding to time.

### Autonomous-person class

Our world would be a rather lifeless place unless we had objects that could somehow "act" on their own. We achieve this by further specializing the person class. An `autonomous-person` is a person who can move or take actions at regular intervals, as governed by the clock through a callback. As described above, the instance indicates that it wants to know when the clock ticks by registering a callback with the clock. It does this upon creation by placing the code to add the callback in the `INSTALL` method. Once again, the `INSTALL` method wants to specify additional behavior, so it calls the superclass' method by asking the `person-part`. Also note how, when an autonomous person dies, we send a "remove-callback" message to the clock, so that we stop asking this character to act.

```
(define (create-autonomous-person name birthplace activity miserly)
  (create-instance autonomous-person name birthplace activity miserly))

(define (autonomous-person self name birthplace activity miserly)
  (let ((person-part (person self name birthplace)))
    (make-handler
     'AUTONOMOUS-PERSON
     (make-methods
      'INSTALL
      (lambda ()
        (ask person-part 'INSTALL)
        (ask our-clock 'ADD-CALLBACK
         (create-clock-callback 'MOVE-AND-TAKE-STUFF self
                               'MOVE-AND-TAKE-STUFF)))
      'MOVE-AND-TAKE-STUFF
      (lambda ()
        ;; first move
        (let loop ((moves (random-number activity)))
          (if (= moves 0)
              'done-moving
              (begin
               (ask self 'MOVE-SOMEWHERE)
               (loop (- moves 1))))))
        ;; then take stuff
        (if (= (random miserly) 0)
            (ask self 'TAKE-SOMETHING))
        'done-for-this-tick))
```

```

'DIE
(lambda (perp)
  (ask our-clock 'REMOVE-CALLBACK self 'MOVE-AND-TAKE-STUFF)
  (ask self 'SAY '("SHREEEEEEK! I, uh, suddenly feel very
faint..."))
  (ask person-part 'DIE perp))
'MOVE-SOMEWHERE
(lambda ()
  (let ((exit (random-exit (ask self 'LOCATION))))
    (if (not (null? exit)) (ask self 'GO-EXIT exit))))
'TAKE-SOMETHING
(lambda ()
  (let* ((stuff-in-room (ask self 'STUFF-AROUND))
        (other-peoples-stuff (ask self 'PEEK-AROUND))
        (pick-from (append stuff-in-room other-peoples-
stuff))))
    (if (not (null? pick-from))
      (ask self 'TAKE (pick-random pick-from))
      #F))))
person-part)))

```

## Configuring and Running the Game

Our world is built by the `setup` procedure that you will find in the file `objsys.scm`. You are the deity of this world. When you call `setup` with your name, you create the world. It has rooms, objects, and people based on a minor technical college on the banks of the Mighty Chuck River; and it has an avatar (a manifestation of you, the deity, as a person in the world). The avatar is under your control. It goes under your name and is also the value of the globally-accessible variable `me`. Each time the avatar moves, simulated time passes in the world, and the various other creatures in the world age by a time step, possibly with a change in state (where they are, how healthy they are, etc.). This works by using a clock that sends an activate message to all callbacks that have been created. This causes certain objects to perform specific actions. In addition, you can cause time to pass by explicitly calling the clock, e.g. using `(run-clock 20)`.

If you want to see everything that is happening in the world, do

```
(ask screen 'DEITY-MODE #t)
```

which causes the system to let you act as an all-seeing god. To turn this mode off, do

```
(ask screen 'DEITY-MODE #f)
```

in which case you will only see or hear those things that take place in the same place as your avatar is. To check the status of this mode, do

```
(ask screen 'DEITY-MODE?)
```

To make it easier to use the simulation we have included a convenient procedure, `thing-`named for referring to an object *at the location of the avatar*. This procedure is defined in the file `objsys.scm`.

When you start the simulation, you will find yourself (the avatar) in one of the locations of the world. There are various other characters present somewhere in the world. You can explore this world, but the real goal is to survive the onslaught of the denizens of darkness.

Here is a sample run of a variant of the system (we have added a few new objects to this version but it gives you an idea of what will happen). Rather than describing what's happening, we'll leave it to you to examine the code that defines the behavior of this world and interpret what is going on.

```
> (setup 'Randymort)
ready
> (ask (ask me 'LOCATION) 'NAME)
eecs-hq
> (ask me 'LOOK-AROUND)
```

```
You are in eecs-hq
You are not holding anything.
You see stuff in the room: SLUG-SPELL
There are no other people around you.
The exits are in directions: up west down OK
> (ask me 'TAKE (thing-named 'SLUG-SPELL))
```

```
At eecs-hq Randymort says -- I take SLUG-SPELL from eecs-hq (instance
#<procedure:handler>)
> (ask me 'GO 'west)
```

```
Randymort moves from eecs-hq to eecs-ug-office
At eecs-ug-office Randymort says -- Hi alyssa-hacker
--- THE-CLOCK Tick 0 ---
ben-bitdiddle moves from 6001-lab to eecs-hq
alyssa-hacker moves from eecs-ug-office to eecs-hq
At eecs-hq alyssa-hacker says -- Hi ben-bitdiddle
course-6-frosh moves from graduation-stage to great-court
At great-court course-6-frosh says -- I try but cannot take flag-pole
lambda-man moves from building-13 to edgerton-hall
lambda-man moves from edgerton-hall to legal-seafood
dr-evil moves from lobby-7 to lobby-10
At lobby-10 dr-evil says -- I'll let you off this once...
mr-bigglesworth moves from graduation-stage to great-court
At great-court mr-bigglesworth says -- Hi course-6-frosh
mr-bigglesworth moves from great-court to legal-seafood
At legal-seafood mr-bigglesworth says -- Hi lambda-man
mr-bigglesworth moves from legal-seafood to edgerton-hall
At edgerton-hall mr-bigglesworth says -- I'll let you off this once...
grendel moves from baker to bexley
grendel moves from bexley to baker
At baker grendel 's belly rumbles
registrar moves from barker-library to 10-250
registrar moves from 10-250 to lobby-10
At lobby-10 registrar says -- Hi dr-evil
At lobby-10 registrar says -- I take SLUG-SPELL from lobby-10
At lobby-10 registrar takes a bite out of dr-evil
```

At lobby-10 dr-evil says -- Ouch! 2 hits is more than I want! #t  
> (run-clock 2)

--- THE-CLOCK Tick 1 ---

ben-bitdiddle moves from eecs-hq to eecs-ug-office  
At eecs-ug-office ben-bitdiddle says -- Hi Randymort  
At eecs-ug-office ben-bitdiddle says -- I take SLUG-SPELL from  
Randymort  
At eecs-ug-office Randymort says -- I lose SLUG-SPELL  
At eecs-ug-office Randymort says -- Yaaaah! I am upset!  
alyssa-hacker moves from eecs-hq to 6001-lab  
At 6001-lab alyssa-hacker says -- I take BOIL-SPELL from 6001-lab  
course-6-frosh moves from great-court to lobby-10  
At lobby-10 course-6-frosh says -- Hi registrar dr-evil  
At lobby-10 course-6-frosh says -- I take SLUG-SPELL from registrar  
At lobby-10 registrar says -- I lose SLUG-SPELL  
At lobby-10 registrar says -- Yaaaah! I am upset!  
lambda-man moves from legal-seafood to great-court  
lambda-man moves from great-court to lobby-10  
At lobby-10 lambda-man says -- Hi course-6-frosh registrar dr-evil  
dr-evil moves from lobby-10 to great-court  
dr-evil moves from great-court to graduation-stage  
At graduation-stage dr-evil says -- I take BOIL-SPELL from graduation-  
stage  
At graduation-stage dr-evil says -- I'll let you off this once...  
mr-bigglesworth moves from edgerton-hall to legal-seafood  
mr-bigglesworth moves from legal-seafood to edgerton-hall  
mr-bigglesworth moves from edgerton-hall to building-13  
At building-13 mr-bigglesworth says -- I'll let you off this once...  
grendel moves from baker to bexley  
grendel moves from bexley to student-center  
grendel moves from student-center to bexley  
At bexley grendel 's belly rumbles  
registrar moves from lobby-10 to 10-250  
registrar moves from 10-250 to barker-library  
At barker-library registrar 's belly rumbles

--- THE-CLOCK Tick 2 ---

ben-bitdiddle moves from eecs-ug-office to eecs-hq  
alyssa-hacker moves from 6001-lab to eecs-hq  
At eecs-hq alyssa-hacker says -- Hi ben-bitdiddle  
At eecs-hq alyssa-hacker says -- I take SLUG-SPELL from ben-bitdiddle  
At eecs-hq ben-bitdiddle says -- I lose SLUG-SPELL  
At eecs-hq ben-bitdiddle says -- Yaaaah! I am upset!  
course-6-frosh moves from lobby-10 to lobby-7  
At lobby-7 course-6-frosh says -- I take BOIL-SPELL from lobby-7  
lambda-man moves from lobby-10 to building-13  
At building-13 lambda-man says -- Hi mr-bigglesworth  
At building-13 lambda-man says -- I take SLUG-SPELL from building-13  
dr-evil moves from graduation-stage to great-court  
At great-court dr-evil says -- What are you doing still up? Everyone  
back to their rooms!  
mr-bigglesworth moves from building-13 to lobby-10  
mr-bigglesworth moves from lobby-10 to grendels-den  
At grendels-den mr-bigglesworth says -- I'll let you off this once...  
grendel moves from bexley to baker  
At baker grendel 's belly rumbles  
registrar moves from barker-library to 10-250

```

At 10-250 registrar 's belly rumbles DONE
> (ask screen 'DEITY-MODE #f)
> (run-clock 3)

--- THE-CLOCK Tick 3 ---
--- THE-CLOCK Tick 4 ---
--- THE-CLOCK Tick 5 --- DONE
> (run-clock 3)

--- THE-CLOCK Tick 6 ---
--- THE-CLOCK Tick 7 ---
--- THE-CLOCK Tick 8 --- DONE
> (run-clock 5)

--- THE-CLOCK Tick 9 ---
--- THE-CLOCK Tick 10 ---
--- THE-CLOCK Tick 11 ---
--- THE-CLOCK Tick 12 ---
An earth-shattering, soul-piercing scream is heard...
--- THE-CLOCK Tick 13 ---
An earth-shattering, soul-piercing scream is heard... DONE
> (ask me 'LOOK-AROUND)

You are in eecs-ug-office
You are not holding anything.
There is no stuff in the room.
There are no other people around you.
The exits are in directions: east OK
>

```

In parts of this project, you will be asked to elaborate or enhance the world (e.g. add things in `objsys.scm`), as well as add to the behaviors or kinds of objects in the system (e.g. modify `objsys.scm`). If you do make such changes, you must remember to re-evaluate all definitions and re-run `(setup 'your-name)` if you change anything, just to make sure that all your definitions are up to date. An easy way to do this is to reload all the files (be sure to save your files to disk before reloading), and then re-evaluate `(setup 'your-name)`.

## Warmup Exercises

Your TA may ask to see these exercises in tutorial on November 21<sup>st</sup> or 22<sup>nd</sup>, but these do not need to be formally submitted as part of the project. These exercises are intended to help you get a head start on understanding our object-oriented world before you start writing code. Note that for some of these exercises, the result is a diagram, which you can draw by hand.

### Warmup Exercise 1

In the transcript above there is a line: `(ask (ask me 'location) 'name)`. What kind of value does `(ask me 'location)` return here? What other messages, besides `name`, can you send to this value?

## Warmup Exercise 2

Look through the code in `objsys.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 `objsys.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 3

Look at the contents of the file `objsys.scm`. What places are defined? How are they interconnected? Draw a map. You must be able to show the places and the exits that allow one to go from one place to a neighboring place.

## Warmup Exercise 4

Aside from you, the avatar, what other characters roam this world? What sorts of things are around? How is it determined which room each person and thing starts out in?

## Warmup Exercise 5

Create an environment diagram corresponding to the evaluation of `(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. Similarly, 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.

## Warmup Exercise 6

To warm up, load the code file `objsys.scm` and start the simulation by typing `(setup '<your name>)` (where you replace `<your name>` with an actual name, of course!). Walk the avatar to a room that has an unowned object. Have the avatar `take` this object, only to `drop` it somewhere else. Show a transcript of this session.

## Computer Exercises

**What to turn in:** When preparing your answers to the questions below, please just turn in the procedures that you have either written or changed (highlighting the actual portions changed) for each problem, a brief description of your changes, and a brief transcript indicating how you tested the procedure. Put each of your solutions and associated transcript into a file, and submit that through the tutor. *Please do not overwhelm your TA with huge volumes of material!!*

### Some general hints for success:

- When you need to test a new object that you have created, you can just “create” it in front of you after you run setup. For example, to test a wand (see below), you can use

```
(begin (setup 'my-name)
      (create-wand 'MY-WAND (ask me 'LOCATION))
      (ask me 'TAKE (thing-named 'MY-WAND))
      (ask (thing-named 'MY-WAND) 'WAVE)))
```

You should not need to change the setup code to do simple tests.

- Don't forget to re-run setup after you change the code for one of the classes! That is, re-evaluating the class's procedure definition does not change the instances of the class that already exist in the world.
- Never use `thing-named` in object code, only for testing; this is a corollary to never using "me" except in testing.
- Outside of the code for `create-world`, variables such as `lobby-10` are not bound to a value. This means you can't teleport there, or create objects there, or ask it questions without filtering over the variable `all-rooms`.

### Hairy Cdr and the Chamber of Stata

Recently, the Wizard's Institute of Technocracy (WIT) revealed itself as an additional department at MIT. It came full-fledged with a large mysterious building topped by twin towers. Preliminary exploration of this building revealed a very confusing interior (kind of maze-like) and an utter lack of square corners. Deep within the center of the building is the reputed Chamber of Stata; the source for magic on the eastern seaboard. To the non-muggles, the building is chock-full of mystical energy, and students have flocked to the new department. We follow the adventures of Hairy Cdr, a young student at WIT who narrowly escaped destruction at the hands of a notoriously evil wizard Lord Randymort.

### Computer Exercise 1: I know I had one of those things somewhere....

For many of the things that follow, it is going to be handy to be able to tell if a person has in his/her possession either a specific type of thing, or a specific instance of a thing. For example, we might want to know if a person has any objects of type `spell`:

```
(ask me 'HAS-A 'SPELL)
;Value: ((instance #<procedure:handler>))
```

That is, we can ask if some person (me in this case) has any objects of a particular type. This should look through the set of `THINGS` held by the person and return a list of those objects, or the empty list if the person does not possess any objects of this type.

The second behavior we want is similar, but now we are looking for an instance of an object with a particular name:

```
(ask me 'HAS-A-THING-NAMED 'SLUG-SPELL)
;Value: ((instance #<procedure:handler>))
```

Modify your definition of `person` to add these two new methods. Demonstrate them working on some test cases.

## Computer Exercise 2: Scoping the Joint

One of the major tools in Hairy Cdr's arsenal is the Marauder's Map, which indicates the location of everybody in the system. This way he can avoid the hall-monitors and trolls!

Add a method `MARAUDERS-MAP` to the `Avatar` which displays the name and location of everybody. You will find the `all-people` variable useful. And you may find the procedure `for-each` useful. The behavior we want is shown below:

```
(ask me 'MARAUDERS-MAP)
```

```
ben-bitdiddle is at lobby-10  
alyssa-hacker is at grendels-den  
course-6-frosh is at bexley  
lambda-man is at 34-301  
susan-hockfield is at edgerton-hall  
eric-grimson is at lobby-10  
dr-evil is at stata-center  
mr-bigglesworth is at student-center  
grendel is at baker  
registrar is at great-court
```

## Computer Exercise 3: The Art of Not Being Seen

Another key component of Hairy Cdr's success is his Cloak of Obfuscation. This unassuming outwear prevents the person carrying it from being seen! Implement a new type of mobile-thing, an obfuscation-cloak. An obfuscation-cloak is an instance of an object that does not accept any new types of messages, being the simplest extension of a mobile-object. Then change `make-person` so that when they look for the `PEOPLE-AROUND`, people who are carrying obfuscation-cloaks are not returned. Note that the Marauder's map is not fooled by these cloaks.

Modify the code in `objsys.scm` to populate the world with some `obfuscation-cloak`'s.

Turn in the entire `make-obfuscation-cloak` code, but only the methods changed in `make-person`. Demonstrate the effectiveness of your obfuscation-cloak objects with test cases.

## Computer Exercise 4: Wand to cause some trouble?

A spell is a type of mobile-thing which can be `USED` to effect some magical work. However, the only way to use a spell is to wave a wand around. Implement a new type of object, a wand. Since a wand should be something that can be transported from place to place, you should think about the class of objects from which it should inherit. Wands have no interesting properties of their own, other than a name and a location. A wand should automatically figure out its caster by looking at its location, and not work unless a person is carrying it. A wand needs to support two methods:

**ZAP:** takes a target to be zapped as its argument. It should pick a random spell from the caster's `THINGS`, print out a message (using the caster's `EMIT` method) about how the caster is waving the wand and saying the spell's `INCANT`, then ask the spell to `USE` with the caster and the target. If the caster isn't carrying any spells, it should print out a message about how the caster is waving the wand, but nothing is happening.

**WAVE:** takes no arguments. It picks a random target from the set of things at the caster's location, picks a random spell from the caster's `THINGS`, prints out a message (using the caster's `EMIT`) about how the caster is waving the wand and saying the spell's `INCANT`, then asks the spell to `USE` with the caster and the target. You should do this by employing the `ZAP` method. You should decide whether you want this to apply only to people, or to any type of thing. You should probably also ensure that you don't accidentally cast a spell on yourself.

Turn in the entire `wand` code. Demonstrate your additions with test cases (create a wand, pick up a spell, and `ZAP/WAVE` for fun and profit).

### Computer Exercise 5: Spell out the options

Examine the `(instantiate-spells)` code in `objsys.scm`. This should help you understand the kinds of properties that spells possess.

Note that the two spells we provide seem to implicitly assume that the target is a person (unless other things have noses and mouths!) yet the code we wrote in the previous exercise could easily have a `WAVE` action apply to a non-person object or thing. Modify the code in `objsys.scm` so that the two spells we provide only work on targets that are people. Reload your system and demonstrate this.

Now, create a new spell, called `ZEPHYR-FROM-THE-DARK`. This spell applies to every person in the room, and causes, at random, up to 2 units of suffering. For any other object, with one chance in four, it should destroy the target.

Finally, create some completely new spell of your design and add it to the world.

Demonstrate tests using all of your spells.

### **Computer Exercise 6: Bette Midler where are you (i.e. “I put a spell on you and now you’re mine”)?**

The halls of Stata are filled with enterprising WIT students who try out their spells on anybody and everybody. Implement a new type of class, a `wit-student`, which acts like an autonomous-person, but also attempts to cast spells every other tick of the clock. On every tick, the student should try to find a spell in its things, then find another person in the room, and if both are found, apply the spell to the person. If the student has a spell, but there are no people present, then the student should instead take its wand (if it has one) and `WAVE` the wand at a random target. You may wish to look at the `INSTALL` code for `autonomous-person` to see how to add clock callbacks. Remember that the students should stop spelling or waving their wands when they’re dead.

In order to assist in the process, each student should begin their life with at least one spell and carrying a wand. This spell and wand should end up in the usual `THINGS` list so that it can be stolen, dropped, etc.

Alter `objsys.scm` to populate the world with students instead of autonomous-persons. Turn in your code for `wit-student` and show examples of the working.

### **Computer Exercise 7: I profess, these students are incorrigible!**

In addition to students, WIT employs a number of professors whose job is to teach students spells! Implement a `wit-professor` that acts like a `wit-student`, except each tick of the clock it also looks around for a student to whom he or she can teach a spell. Teaching a spell to a student involves cloning a spell out of the Chamber of Stata and into the student’s inventory.

Once you’ve implemented `wit-professor`, uncomment the line in `populate-players` in `objsys.scm`.

Turn in a listing of the `wit-professor` code and a test run that shows it working.

### **Computer Exercise 8: Oh yeah, well take that!**

To make things a fair fight, we want to add something that can counteract spells. So create a new type of object, called a counterspell. A counterspell should have the property that if a character attempts to cast a specific spell on a target, and the target possesses the appropriate counterspell, then the spell itself does not take effect. Thus, in addition to creating counter spells, you will want to modify your spells to obey this new behavior.

### **Computer Exercise 9: See this scar in the shape of a Lambda?**

Lastly, it’s time to implement Hairy Cdr himself. Hairy is a WIT student, but more importantly, he’s the “chosen one”. Thus he cannot die until the series is finished, which

at the current rate will be long after you graduate. Any time Hairy is about to die, his scar flares brightly and the person attempting to kill him dies instead (like Lord Randymort). More specifically, when a `chosen-one` is asked to `SUFFER` enough to kill them, it should print out an appropriate message and then kill the perpetrator, leaving the `chosen-one` unharmed.

Note that we are asking you to have a chosen one override the `SUFFER` method, rather than the `DIE` method. Why? What could go wrong if a chosen one were to implement this behavior using the `DIE` method instead?

As usual, turn in your code that implements the `chosen-one` and test cases that demonstrate it working.

### **Computer Exercise 10: Your turn**

Now that you have had an opportunity to play with our "world" of characters, places, and things, we want you to extend this world in some substantial way. The last part of this project gives you an opportunity to do this. As you haven't had much freedom to design your own code up until now, this exercise gives you the opportunity to demonstrate your knowledge of design principles. **NOTE: this exercise is worth significantly more points than the others; give it the time it deserves!**

This is your opportunity to have fun with the game! There's only one hard requirement: you must add at least one new class to the system. Other than that, you are free to take the simulation in any direction that interests and excites you. You don't have to stick with the Hairy Cdr theme if you don't want to. Here are a few ideas that we've come up with for extensions, but you certainly aren't limited to these:

**Dementors** – They're like trolls, but immune to suffering. They can only be killed with the patronus spell.

**Moving Exits** – Exits that change their destination over time.

**Brooms** – If the person is carrying a broom they can use "flying" exits to move around campus quickly.

**House points** – Each student belongs to a house. Doing certain activities adds points, doing others subtracts points.

**Spell Points** – Every person has an amount of spell points. Casting spells uses up spell points, which return slowly over time.

Remember to select an extension that you have an expectation of finishing. If you're going to attempt something ambitious, remember to have a fall back plan that meets the objectives for the exercise!

#### **Stage One: Plan**

Here, we want you to plan out the design for some extensions to your world. You will submit a brief description of your plan by email to your TA on *Wednesday, November 23<sup>rd</sup>* at the latest (send earlier if possible!).

We want you to design some new elements to our world. The first thing we want you to do is design a new class of objects to incorporate into the world. To do this, you should plan each of the following elements:

**Object class:** First, define the new class you are going to build. What kind of object is it? What are the general behaviors that you want the class to capture?

**Class hierarchy:** How does your new class relate to the class hierarchy of the existing world? Is it a subclass of an existing class? Is it a superclass of one or more existing classes?

**Class state information:** What internal state information does each instance of the class need to know?

**Class methods:** What are the methods of the new class? What methods will it inherit from other classes? What methods will shadow methods of other classes?

**Demonstration plan:** How will you demonstrate the behavior of instances of your new class within the existing simulation world?

## **Stage Two: Implementation**

Attempt to follow your plan. Remember to test your code in a number of ways to ensure that it does what you intended it to do.

## **Stage Three: Submission**

Your submission should have a couple of components:

### **Design of your improvements**

Write up a BRIEF description of your design, addressing each of the issues raised above.

### **Code**

For this part of the project, using the online tutor submit your code and a transcript of your system in action. Do not just submit the entire file of objects, rather submit only those changes (if any) that you have made to the existing system, and the new code that you have written. Be sure to document appropriately!

You will be graded based on the quality and scope of your design, how well you implemented your design, and the quality of your documentation.

**We will award prizes for the most interesting modifications combined with the cleverest technical ideas. Note that it is more impressive to implement a simple, elegant idea than to amass a pile of characters and places.**

## **Collaboration Statement**

Please respond to the following question as part of your answers to the questions in the project set:

We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout) and so long as you observe the rules on collaboration and using "bibles". If you cooperated with other students, LA's, or others, please indicate your consultants' names and how they collaborated. Be sure that your actions are consistent with the posted course policy on collaboration.