MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 2002

Project II

- Issued: Wednesday, Week 7 / Day 3.
- Tutorial Problems Due: Week 8, in tutorial.
- Part I Due: Friday, Week 9 / Day 5, by evening.
- Part II Due: Friday, Week 10 / Day 5, by evening.
- Code: The following code (attached) should be studied as part of this problem set:
  - `objsys_fa02.scm`—support for an elementary object system
  - `objtypes_fa02.scm`—a few nice object classes
  - `setup_fa02.scm`—a bizarre world constructed using these classes

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 ensures that the assignments will take much longer than necessary.

Note that you may find it useful to listen to the lectures on Object Oriented Programming ahead of schedule, in order to get a good start on this project.

You must hand in solutions by submitting them via the online tutor, by 6:00pm on the date listed. Late work will not be accepted.

Word to the wise: This project is difficult. The trick lies in knowing *which* code to write, and for that you must understand the attached 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 `objsys_fa02.scm`). Then you'll need to understand the particular classes (in `objtypes_fa02.scm`) and the world (in `setup_fa02.scm`) that we've constructed for you. In truth, this assignment in 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. 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 recently, 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.

2. 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.

Let's start with the fundamental stuff first. We can think of our object oriented paradigm as consisting of *classes* and *instances*. Classes can be thought of as the "template" for how we want different kinds of objects to behave. The way we define the class of an object is with a basic "make object" procedure; when this procedure is applied, it makes for us a particular instance.

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

2.1 Classes, Instances, and Methods

For example, our simulation world will consist of named objects. We can make a named object using the procedure `make-named-object`. A named object is a procedure that takes a message and returns the method that will do the job you want.[1] For example, if we call the method obtained from a named object by the message `NAME` we will get the object's name.

```
(define (make-named-object name . characteristics)
  (let ((root-part (make-root-object)))
    (lambda (message)
      (case message
        ((NAMED-OBJECT?) (lambda (self) #T))
        ((NAME) (lambda (self) name))
        ((CHARACTERISTICS) (lambda (self) characteristics))
```

[1]We will use the special form `case` to do the dispatch. See the Scheme Reference Manual for details. In essense, this acts much like a `cond`, matching the first argument against the first clause of each subsequent term using `eq?`, when it finds one that matches it evaluates and returns the subsequent part of that expression.

```
        ((INSTALL) (lambda (self) 'INSTALLED))
        ((DESTROY) (lambda (self) 'DESTROYED))
        (else (find-method message root-part))))))

(define foo (make-named-object 'george ``a handsome devil''))

((foo 'NAME) foo) ==> george
```

Note the use of the "dot" (.) notation in the definition. Check this out in the Scheme manual for details. Essentially this allows `make-named-object` to be called with one or more arguments. The parameter `name` will be bound to the value of the first argument. The value `characteristics` will be bound to a **list** of the values of all additional arguments. In this case, we are particularly interested in zero or one additional argument, since some of our objects will have characteristics, and others will not.

The first formal parameter of every method is `self`. The corresponding argument must be the object that needs the job done. This was explained in lecture, and we will see it again below.

Note that a named object inherits from a root object, which we treat as the most fundamental, and simplest, of classes.

```
(define (make-root-object)
  (lambda (message)
    (no-method)))
```

This object simply provides a basis for providing common behaviors to all classes, which for now is simply a way of indicating that no method is available for the desired message. We will by convention use this class as the base for all other classes.

A named object has a method for five different messages: `NAMED-OBJECT?`, `NAME`, `CHARACTERISTICS`, `INSTALL` and `DESTROY`. Depending on the message, a named object will return a method that confirms that it is indeed a `named-object`; it will give a method to return its `name`; it will give a method to return its `characteristics`; it will give a method for installation that does nothing; and it will give a method for destruction that does nothing.

In the above example, we created an instance `foo`, then sent it the message `NAME` to get its name method, and finally applied that method to the object itself to get the name. Our system provides a preferred short-hand way of putting together the method lookup and method application using `ask`. What `ask` does here is get the `NAME` method from `foo` and then call it with `foo` as the argument (so the value of `foo` will be bound to `self` in the method body). The full `ask` procedure is defined in the file `objsys_fa02.scm`, but here is a simplified version that works for messages requiring no arguments:

```
(define (simple-ask object message)
  ((get-method message object) object))

(define (get-method message object)
  (object message))

(simple-ask foo 'NAME) ==> george
```

We see that our system also provides the procedure `get-method` to request a method from an object, which simply sends the message to the object. There is a special way for our objects to say there is no method: `(no-method)`, as shown in the `root-object` class definition above. This returns a special value that can be used later on in our system to detect when there is no method using the `method?` predicate, e.g.

```
(method? (foo 'NAME)) ==> #T
(method? (foo 'SHAPE)) ==> #F
```

2.2 Inheritance and Subclasses

A `thing` is another kind of computational object which will be located somewhere in our world. In the code below we see that a thing is implemented as a message acceptor that intercepts some messages. If it cannot handle a particular message itself, it passes the message along to a private, internal named object (`named-object-part`) that it has made as part of itself to deal with such messages (see the last line in the definition of `make-thing`). 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 (make-thing name location characteristics)
  (let ((named-object-part (make-named-object name characteristics)))
    (lambda (message)
      (case message
        ((THING?) (lambda (self) #T))
        ((LOCATION) (lambda (self) location))
        ((INSTALL)
         (lambda (self)         ; Install: synchronize thing and place
           (ask (ask self 'LOCATION) 'ADD-THING self)
           (delegate named-object-part self 'INSTALL)))
        ((DESTROY)
         (lambda (self)         ; Destroy: remove from place
           (ask (ask self 'LOCATION) 'DEL-THING self)
           (delegate named-object-part self 'DESTROY)))
        ((EMIT)
         (lambda (self text)         ; Output some text
           (ask screen 'TELL-ROOM (ask self 'LOCATION)
                (append (list "At" (ask (ask self 'LOCATION) 'NAME))
                        text))))
        (else (get-method message named-object-part))))))
```

There are several other interesting aspects of the `thing` class definition above. We see that a thing instance will respond to the `THING?` message with a procedure that, when applied to the instance, will return `#T`. But an object that is not a `thing` will not find the `THING?` message and an error will result. To get around this problem, and for improved convenience as well, our system provides a procedure `is-a` that can be used to check the class of an object.

```
(define (is-a object type-pred)
  (if (not (procedure? object))
      #f
      (let ((method (get-method type-pred object)))
        (if (method? method)
            (ask object type-pred)
            #F))))

(define my-book (make-thing 'great-gatsby dark-room))

((get-method 'THING? my-book) my-book) ==> #T
((get-method 'NAMED-OBJECT? my-book) my-book) ==> #T

(is-a my-book 'THING?) ==> #T
(is-a my-book 'NAMED-OBJECT?) ==> #T
(is-a my-book 'EMOTION?) ==> #F
```

This enables us to ask an object if it is an instance of a particular class. For example, we can see that a thing we make is a `thing`, but also is a `named-object` (you can assume that `dark-room` is a location previously made). How does the `is-a` procedure work? If we ask for the `THING?` method from a thing instance (`my-book`, in this case), `my-book` immediately gets and returns the method defined in `make-thing`. However, if we ask for the `NAMED-OBJECT?` method from `my-book`, the `my-book` object passes the message along to its internal `named-object-part`, where the `NAMED-OBJECT?` method is finally found and returned. The `is-a` utility procedure tries to find the appropriate type check method, and if found invokes it on the object, otherwise concluding that the object is not an instance of the requested type.

2.3 Delegation

Another idea shown in the "thing" class (which is specified by the `make-thing` procedure above) is that of *delegation*, which is the explicit use of an "internal" object's method by the object. In the `thing` class, we see that the `INSTALL` method "shadows" or intercepts the `INSTALL` method in the `named-object` class. In `make-thing`, we want to first do some work to integrate the thing object into our simulation world (more on that later), but then we *also* want to invoke the superclass named-object `INSTALL` method in case something important happens there as well. But since the internal `named-object-part` is really not a "stand-alone" object all its own, we don't `ask` it to do something on its own, instead we `delegate` the task to the internal object. To delegate is to have the internal object do the requested work, but on *behalf* of the full `self` object.

The important difference is that if we `ask` an object to do something, then the `self` value passed to the method will be the object itself. Using `delegate`, on the other hand, we can explicitly control what the `self` value will be that is passed to the method, and can thus have a part (inherited superclass) of the object do something to the whole object. This is perhaps the single most subtle and difficult aspect of our system, and you will explore this idea and issue in more detail in the exercises.

3. Classes for a Simulated World

When you read the code in `objtypes_fa02.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.

3.1 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 (make-container)
  (let ((root-part (make-root-object))
        (things '())) ; a list of THING objects in container
    (lambda (message)
      (case message
        ((CONTAINER?) (lambda (self) #T))
        ((THINGS) (lambda (self) things))
        ((HAVE-THING?)
         (lambda (self thing)  ; container, thing -> boolean
           (not (null? (memq thing things)))))
        ((ADD-THING)
         (lambda (self new-thing)
           (if (not (ask self 'HAVE-THING? new-thing))
               (set! things (cons new-thing things)))
           'DONE))
        ((DEL-THING)
         (lambda (self thing)
           (set! things (delq thing things))
           'DONE))
        (else (find-method message root-part))))))
```

Notice 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; 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.

3.1 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 (make-place name characteristics)
  (let ((named-obj-part (make-named-object name characteristics))
        (container-part (make-container))
        (exits '()))    ; a list of exit
    (lambda (message)
      (case message
        ((PLACE?) (lambda (self) #T))
        ((EXITS) (lambda (self) exits))
        ((EXIT-TOWARDS)
         (lambda (self direction)  ; place, symbol -> exit | #F
           (let ((ex (find-exit-in-direction exits direction)))
             (if (and ex (ask ex 'HIDDEN?))
                 #f
                 ex))))
        ((ADD-EXIT)
         (lambda (self exit)
           (let ((direction (ask exit 'DIRECTION)))
             (cond ((ask self 'EXIT-TOWARDS direction)
                    (error (list name "already has exit" direction)))
                   (else
                    (set! exits (cons exit exits))
                    'DONE)))))
        (else
         (find-method message container-part named-obj-part))))))
```

If we look at the first and last lines of `make-place`, we notice that `place` inherits from two different classes: it has both an internal `named-object-part` and an internal `container-part`. Here we use the object oriented system procedure `find-method` (defined in `objsys_fa02.scm`) which will try to find the first matching method by looking (in order) in the provided internal objects. Thus, if we ask for the `NAME` method from a place instance, the method will be found in the internal `named-object-part`, while if we ask for the `HAVE-THING?` method from a place instance, the appropriate method well be found and returned from the internal `container-part` object. This idea is often termed "multiple inheritance".

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. You should examine the `objtypes_fa02.scm` file to understand the definition for `make-exit`.

3.2. Mobile-thing Class

Now that we have things that can be contained in some place, we might also want `mobile-things` (made by `make-mobile-thing`) that can `CHANGE-LOCATION`.

```
(define (make-mobile-thing name location characteristics)
  (let ((thing-part (make-thing name location characteristics)))
    (lambda (message)
      (case message
        ((MOBILE-THING?) (lambda (self) #T))
        ((LOCATION)     ; This shadows message to thing-part!
         (lambda (self) location))
        ((CHANGE-LOCATION)
         (lambda (self new-location)
           (ask location 'DEL-THING self)
           (ask new-location 'ADD-THING self)
           (set! location new-location)))
        ((ENTER-ROOM)
         (lambda (self exit) #t))
        ((LEAVE-ROOM)
         (lambda (self exit) #t))
        ((CREATION-SITE)
         (lambda (self)
           (delegate thing-part self 'location)))
        (else (get-method message 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`. Note that here we use the `ask` procedure, since we are sending a message to the specified location objects that exist external to the `mobile-thing`; it would be inappropriate to `delegate` in this situation.

3.3. Person Class

A person is a kind of mobile thing. When a person is made, an internal mobile thing is also made to handle messages such as `CHANGE-LOCATION`. The mobile thing is bound to a variable that is visible only within the person object – `mobile-thing-part`. When a person moves from one place to another, it does so by using the `CHANGE-LOCATION` method from its internal `mobile-thing-part`. However, it is the person that moves. Thus, it is the person that must be added or removed from the location, not the mobile thing from which the method was obtained. The internal `mobile-thing-part` is not a whole person – it is only a fragment of the person. To implement the desired behavior the `CHANGE-LOCATION` method needs to know the complete or whole moving object (the person), and this is what is passed to the method as `self`. This is crucial for you to understand if your objects are to maintain their integrity!

If we consider the (partial) definition of `make-person`, we also notice that a person is a container as well as a mobile thing. Again, this is an example of multiple inheritance. The idea here is that people can also "contain things" which they carry around with them when they move.

A person can `SAY` a list of phrases. A person can `TAKE` something, as well as `DROP` something. Some of the other messages a person can handle are briefly shown below; you should consult the full definition of `make-person` in `objtypes_fa02.scm` to understand the full set of capabilities a person instance has.

```
(define (make-person name birthplace characteristics)
  (let ((mobile-thing-part (make-mobile-thing name birthplace characteristics))
        (container-part    (make-container))
        (health            3)
        (strength          1))
    (lambda (message)
      (case message
        ((PERSON?) (lambda (self) #T))
        ((STRENGTH) (lambda (self) strength))
        ((HEALTH) (lambda (self) health))
        ((SAY)
         (lambda (self 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 (self)
           (ask self 'SAY '("Yaaaah! I am upset!"))
           'I-feel-better-now))
        ((PEOPLE-AROUND) (lambda (self) ...))
        ...
        ((TAKE) (lambda (self thing) ...))
        ((LOSE)
         (lambda (self thing lose-to)
           (ask self 'SAY (list "I lose" (ask thing 'NAME)))
           (ask self 'HAVE-FIT)
           (ask thing 'CHANGE-LOCATION lose-to)))
        ((DROP)
         (lambda (self thing)
           (ask self 'SAY (list "I drop" (ask thing 'NAME)
                                "at" (ask (ask self 'LOCATION) 'NAME)))
           (ask thing 'CHANGE-LOCATION (ask self 'LOCATION))))
        ((GO) (lambda (self direction) ...))
        ...
        (else (find-method message mobile-thing-part container-part))))))
```

3.4 Avatar Class

One kind of character you will use in this problem set 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 to do what a person does when it receives a `GO` message. This is again accomplished by explicit delegation. The avatar does whatever it has to, and in addition, it delegates to its internal person the processing of the `GO` message, with the avatar as `self`. Notice that we have a fairly fine degree of control over how inheritance and delegation are managed. In the case of the avatar, we first delegate to the internal person to handle the `GO` message, and then do something more after that (in this case, invoke the simulation clock).

```
(define (make-avatar name birthplace murder-details characteristics)
  (let ((person-part (make-person name birthplace characteristics))
        (crime-details murder-details)
        (count 0))
    (lambda (message)
      (case message
        ((AVATAR?) (lambda (self) #T))
        ((LOOK-AROUND)          ; report on world around you
         (lambda (self) ...))
        ((GO)
         (lambda (self direction)  ; Shadows person's GO
           (let ((success? (delegate person-part self 'GO direction)))
             (if success? (ask clock 'TICK))
             success?)))
        ...
        ((TAKE) (lambda (self thing) ...))
        (else (get-method message person-part))))))
```

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

3.5 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-player` is a person who can move or take actions at regular intervals, as governed by the clock through a callback.

Our clock works by using what are known as "callbacks". This means that we create an instruction which we install in the clock, with the property that every time the clock iterates, it executes all the instructions it has stored up. Each of these instructions sends a message to an object, causing it to synchronously execute an action. In the example below, installing an autonomous person causes the clock object to add an instruction that will send this object a "move-and-take-stuff" message, which will then cause this object to select an action. See the discussion on the clock in the `objsys_fa02.scm` file for details on how the clock operates. However, the template used below for sending the clock a "callback" will be valuable to you in creating your own objects and methods. Also note how, when an autonomous player dies, we send a "remove-callback" message to the clock, so that we stop asking this character to act.

```
(define (make-autonomous-player name birthplace activity miserly characteristics)
  (let ((person-part (make-person name birthplace characteristics))
        (alibi-room 'nowhere)
        (alibi-possessios 'nothing)
        (alibi-witnesses 'nobody))
    (lambda (message)
      (case message
        ((AUTONOMOUS-PLAYER?) (lambda (self) #T))
        ((INSTALL) (lambda (self)
                     (ask clock 'ADD-CALLBACK
                          (make-clock-callback 'move-and-take-stuff self
                                               'MOVE-AND-TAKE-STUFF))
                     (delegate person-part self 'INSTALL)))
        ((MOVE-AND-TAKE-STUFF)
         (lambda (self)
           ;; 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 (self)
           (ask clock 'REMOVE-CALLBACK self 'move-and-take-stuff)
           (delegate person-part self 'DIE)))
        ((MOVE-SOMEWHERE)
         (lambda (self)
           (let ((exit (random-exit (ask self 'LOCATION))))
             (if (not (null? exit)) (ask self 'GO-EXIT exit)))))
        ((TAKE-SOMETHING)
         (lambda (self)
           (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))))
        ...
        (else (get-method message person-part))))))
```

3.6 Installation

One final note about our system. If you look in `objtypes_fa02.scm`, you'll see that objects have an `INSTALL` method which does some appropriate initialization for a newly made object. For example, if you *create* a new mobile thing at a place, the object must be added to the place. As you'll see in the code, we define two procedures for each type of object: `make-` and a `create-` procedure. The make procedure (e.g. `make-person`) simply makes a new instance of the object, while the create procedure (e.g. `create-person`) both (1) makes the object *and* (2) installs it. When you create objects in our simulation world, you should do this using the appropriate create procedure. Thus, to create a new person, use `create-person` rather than calling `make-person` directly.

The following distinction should also help you think about `make-object` versus `create-object` procedures. The `make-object` procedure should only be used "inside" our object oriented programming code: e.g., in `objtypes_fa02.scm` you "make" a stand-alone person or part of an person using, for example `make-person` or `make-named-object` or whatever. But this only gives you an object that is not yet connected up with our world. To get a fully functioning object in a particular world, you need to "create" that object. Thus you should use the `create-object` variant when you actually want to make and install an object in a simulation world, as we do in `setup_fa02.scm`.

Our world is built by the `setup` procedure that you will find in the file `setup_fa02.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 the Clue game (by Parker Brothers) 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 take a time step. The way this works is that there is a clock that sends a `clock-tick` message to all autonomous persons. (The avatar is not an autonomous person; it is directly under your control.) 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 convenience procedure, `thing-named` for referring to an object at the location of the avatar. This procedure is defined at the end of the file `setup_fa02.scm`.

When you start the simulation, you will find yourself (the avatar) in one of the rooms of the Clue mansion. The Clue characters are also present somewhere in the mansion, and one of them is about to commit a murder by using one of the Clue weapons.[2] When a murder is committed, the victim screams and it's up to you to figure out who did it, in which room and with what weapon... but look out because you too (the avatar) can be killed. Note however, that once the murderer has found a victim, he becomes filled with remorse, repents and does not commit any more murders for the duration of the current simulation.

Here is a sample run of the system. 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 'eric)
eric moves from heaven to hall
;Value: ready

(ask (ask me 'location) 'name)
;Value: hall

(ask me 'look-around)
You are in hall
You are not holding anything.
You see stuff in the room: knife rembrandt
There are no other people around you.
The exits are in directions: west south
;Value: ok

(ask me 'examine (thing-named 'rembrandt))
It is one of Rembrandt's masterpieces!
;Value: message-displayed

(ask me 'take (thing-named 'knife))
At hall eric says -- I take knife from hall
;Value: #[unspecified-value]

(ask screen 'deity-mode #f)
;Value: #f

(run-clock 3)
---the-clock Tick 0---
At foyer : miss-scarlet says -- I take wrench
---the-clock Tick 1---
mrs-white moves from lounge to dining-room
At billiard-room : professor-plum says -- I lose rope
miss-scarlet moves from foyer to ballroom
```

[2]If you are unfamiliar with the Clue game, just peruse the `setup_fa02.scm` code to become more familiar with this world.

```
---the-clock Tick 2----
professor-plum moves from billiard-room to library
mrs-peacock moves from study to library
At library : mrs-peacock says -- Hi professor-plum
;Value: done
```

3.7 Changing the World

In parts of this project, you will be asked to elaborate or enhance the world (e.g. add things in `setup_fa02.scm`), as well as add to the behaviors or kinds of objects in the system (e.g. modify `objtypes_fa02.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)`.

4. Tutorial Exercises

You should prepare these exercises early, in order to get a sense for the world you will be exploring. **You will be expected to turn in answers to these problems during tutorials the week of October 21.**

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?

Exercise 2: Look through the code in `objtypes_fa02.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_fa02.scm` to see what the world looks like. Draw a class diagram and a skeletal instance diagram like the ones presented in lecture. You will find such a diagram helpful (maybe indispensable) in doing the programming assignment.

Exercise 3: Look at the contents of the file `setup_fa02.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.

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?

Exercise 5: The avatar, as a person, may have possessions. How does the avatar handle the request `(ask me 'things)`? In particular, which method is used to respond to the request and which variable holds the list of possessions? Sketch a skeletal environment diagram to help. Note that we are not asking you to draw a fully detailed environment diagram here—it is huge and more confusing than helpful!

Exercise 6: Draw an environment diagram showing the state of the environment after evaluating:

```
(define foo (make-mobile-object 'eric lounge '()))
```

Assume that `lounge` is bound to some procedure, but don't worry about the details of that procedure.

Further, show the state of the environment after evaluating

```
(ask foo 'location)
```

Don't worry about showing the frames created by calling `ask` or `ask-helper`.

Though it is more work, you may find it useful to think about what happens when other methods, such as `install` or `name` are called.

5. Programming Assignment

To warm up, load the three files `objsys_fa02.scm`, `objtypes_fa02.scm` and `setup_fa02.scm` and start the simulation by typing `(setup '<`*`your name`*`>)`. (If you are using Athena, the `M-x load-problem-set` command, with argument 2, will work.) Play with the world a bit. One simple thing to do is to stay where you are and run the clock for a while with `(run-clock <`*`ticks`*`>)`. Since the characters in our simulated world have a certain amount of restlessness, people should come walking by and say Hi to you. Try running the clock with the screen's `deity-mode` parameter set to both true and false. When it is set to true, you see almost everything that happens everywhere in the simulation. When it is set to false, you see only what happens in the room you are in. You should set `deity-mode` to false when you are ready to "play" the game and attempt to solve the murder.

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. **Please do not overwhelm your TA with huge volumes of material!!**

Computer Exercise 1: Getting Acquainted with the System Walk the avatar to a room that has an unowned weapon. Have the avatar `take` this weapon, only to `drop` it somewhere else. Show a transcript of this session.

Computer Exercise 2: Understanding Installation Note how `install` is implemented as a method defined as part of `thing` and `autonomous-person`. Notice that the `autonomous-person` version puts the person on the clock list (this makes them "animated") then `delegate`s an `install` message from its `self` to its internal `thing`, which contains the `INSTALL` method responsible for adding the `person` to its `birthplace`. The relevant details of this situation are outlined in the code excerpts below:

```
(define (make-autonomous-person name birthplace laziness characteristics)
  ;; Laziness determines how often the person will move.
  (let ((person-part (make-person name birthplace characteristics)))
    ...
      (case message
        ...
        ((INSTALL)
         (lambda (self)
           (ask clock 'ADD-CALLBACK
                (make-clock-callback 'move-and-take-stuff self
                                     'MOVE-AND-TAKE-STUFF))
           (delegate person-part self 'INSTALL)))   ; **
        ...)))


(define (make-thing name location characteristics)
  (let ((named-object-part (make-named-object name characteristics)))
    ...
      (case message
        ...
        ((INSTALL)
         (lambda (self) ; Install: synchronize thing and place
             ...
             (ask (ask self 'LOCATION) 'ADD-THING self)
             (delegate named-object-part self 'INSTALL))
                   ...))))))
```

Louis Reasoner suggests that it would be simpler if we change the last line of the `make-autonomous-person` version of the `install` method (marked `; **`) to read:

```
                    (ask person-part 'INSTALL) ))   ; **
```

Alyssa points out that this would be a bug. "If you did that," she says, "then when you make and install an autonomous person, and this person moves to a new place, he'll be in two places at once!"

What does Alyssa mean? Specifically, what goes wrong? You may need to draw an appropriate environment diagram to help you to explain carefully.

Computer Exercise 3: Who Just Died? Explore the world until "An earth-shattering, soul-piercing scream is heard...", which means that someone (hopefully not you) has just been murdered. Where does the victim go? If you know where the victim goes (and assuming you are not in `deity-mode`), what simple scheme expression can you evaluate to find out who just died?

Computer exercise 4: Having a quick look

Change the behavior of the avatar, to LOOK-AROUND whenever it successfully moves to a new location. Shows the change to your code, and demonstrate it working in an example scenario.

Now, for some real changes! In the next several exercises you will extend the system to add additional behaviors and nuances.

Computer Exercise 5: On Closer Inspection You may have noticed that some rooms contain secret passages. When you first enter a room, these do not show up on the list of exits for the room, because they are `hidden`.

Add a new method to `person` objects, called `SCOUR`. This method should find all the exits (whether hidden or not) in a location, and for those that are hidden, change them to no longer be hidden. It should then print out a list of all exits. For example, in the example given earlier, if we had evaluated (while in the kitchen)

```
(ask me 'scour)
```

our new system might return

```
On closer examination, you see that the exits are (north west secret-passage)
;Value: message-displayed
```

thus indicating that there is a secret passage.

Add this method, then recreate the world and test it out. Show a transcript indicating your code at work.

Computer Exercise 6: Pardon me! Coming through! Even once you have the ability to find hidden passages, you may discover that you can't get through them. This is a nuisance! So we want you to change the system to allow this.

If you look carefully at the code, you will discover that each exit has a variable that indicates if the exit is locked (which is different from being hidden). This prevents the method `USE` from moving through the exit.

Note how this is currently done. Each exit has a procedure it calls to see if it can unlock itself (see `setup_fa02.scm` for examples of these procedures). In our case, we just require that some object be in the person's possession when they try to use the exit. If so, the state of the exit is changed, and subsequently anyone can move freely through the exit. You may find it interesting to play the game to see that this works as described by trying to move through locked exits without and with the necessary "key" object.

In this problem, we want to improve this idea. To do this, you are to create a new kind of object, called a `key-object`. This object has a name, and a location, like other objects. It also has a target location and a key missing piece. The object should support a method in which it checks to see if it is in the target location and if the missing piece is in the same location. If so, it should then change the status of the secret passage, to be unlocked.

Using this new kind of object, create two such key objects – a whistle and a chime. The whistle needs a pea in order to work and should be in the lounge or conservatory to open the passage. When the whistle gets the message `BLOWIT`, it will try to open the passage.

The chime needs a tapper, and should be in the kitchen or study to open the passage. When the chime gets the message `RINGIT`, it will try to open the passage.

You should think carefully about how each of these objects should access methods of internal objects.

You will need to modify the `setup_fa02.scm` file to create and place the whistle, the chime, the pea, and the tapper randomly in the house.

You may also need to modify the definition of the `exit` class so that its `USE` method no longer relies on a procedure to attempt to unlock.

Reinitialize your world, then find each new object, move to the right place, and use the object to demonstrate that you can now move through the secret passages.

Submit your code and detail any design decisions you made. Also include a short transcript (only the relevant parts in your excerpt) that shows that this code is working properly.

Computer Exercise 7: Fingerprinting Weapons Next, extend the system so that when someone picks up a weapon (or any `thing` object), his/her fingerprints are left behind and recorded on the object.

(a) modify `make-thing` so that it can keep track of its previous owners.

(b) extend `make-thing` with a method `FINGERPRINT` that returns a list of all its previous owners.

Submit your code and include a short transcript (only relevant parts) that shows that this code is working properly.

Computer Exercise 8: Reading Fingerprints Carefully Now create a magnifying glass, and modify the system so that the fingerprints on a weapon are visible only to someone who is holding the magnifying glass.

Submit your code and detail any design decisions you made. Also include a short transcript (only the relevant parts in your excerpt) that shows that this code is working properly.

Computer Exercise 9: Solving the Crime Once a murder has been committed, it is your task to solve the mystery. You can figure out where the murder occurred (examine the code and explain how). Armed with the magnifying glass, you can examine the fingerprints on each weapon you find to figure out which people it has come in contact with. Lastly, you can ask each character for his/her `ALIBI`. An alibi is simply a report from a Clue character (now turned suspect) that says (1) where they were when the murder occurred, (2) what was in their possession at that time and (3) who else was in the room. These three pieces of information are returned as a list, and the suspect will interpret them for you as a side effect. For example:

```
(ask (thing-named 'colonel-mustard) 'ALIBI)
colonel-mustard says -- Me?  I was in the conservatory
colonel-mustard says -- I had in my possession: (knife)
colonel-mustard says -- Oh, and  (miss-scarlet) was in the room with me
;Value: (conservatory (knife) (miss-scarlet))
```

If you think you have solved the crime, you can then hazard a `GUESS` as to the room, weapon and murderer (in this order). For example:

```
(ask me 'GUESS 'library 'wrench 'mr-green)
```

Turn in a sample excerpt (not the whole transcript) showing that everything works appropriately and that you can indeed figure out who did it, where and with what. Note that in some cases, you may not be able to deduce the answer uniquely so you may have to make a few guesses.

Collaboration Statement Please respond to the following question as part of you 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). 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.

Part II

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. Part II of this project will give you an opportunity to do this.

In Part II, we want you to plan out the design for some extensions to your world. You will submit a brief description of your plan to your TA. As well, you will implement your ideas, and demonstrate their use.

Designing changes to the world – a new class We want you to design some new elements to the "Clue" 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.

1. **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?
2. **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?
3. **Class state information:** What internal state information does each instance of the class need to know?
4. **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?
5. **Demonstration plan:** How will you demonstrate the behavior of instances of your new class within the existing simulation world?

Here are some examples of a possible new class of objects:

- A video camera. If you place a camera in a room, it should be able to record all the people who pass through the room. It might also be able to record when they were in the room. Note that this already suggests some information about the new class. First, it is likely to be a specialization of an `object`, but it may not need to be a `mobile-object` (in fact, if we were to make this a movable object we might want to consider that as a different class, since it would probably need to keep track of more information than a stationary camera). Second, it is going to need to get information from the clock, in order to know when people enter or leave the room. And it is going to need some internal state information, such as a list or some other data structure, for capturing when people enter and leave.
- A dog. This is a very loyal dog, so it always wants to stay with its owner. Thus, if the owner is in some room together with the dog, the dog should stay in that room. If, however, the owner changes locations, the dog will want to follow. If it doesn't know which direction the owner moved, then it will need to move randomly until it finds its owner. Clearly this needs to be a specialization of a `mobile-object`. You might even consider it as a kind of `person`, though you will then need to think about what methods of a person will need to be shadowed by this kind of object.

What to turn in

Design of your improvements You should work out a design of some new object. Use your imagination, and invent something intriguing (i.e., you don't have to make cameras or dogs!). Write up a **BRIEF** description of your design, addressing each of the issues raiseed above.

Submit your description as the first part of your writeup for Part II of Project 2.

Making it work Now, implement your new class of objects, and test them out.

For this part of the project, 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!

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 you 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). 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.