

6.001 Tutorial 10

David Ziegler

November 15, 2004

1 Administrivia

- Problem set 8 is due tomorrow.
- Project 4 is due Friday. Please remember to send me an email with the details of your extension to the system by 6:00 on Wednesday! Also, it will almost certainly *not* work in other Scheme implementations — use MIT Scheme!

2 Object-Oriented Programming

2.1 The High Level

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

In *our* object-oriented system, the implementation is based on *messages*. To request that an object does something, you send it a message by applying the object to a symbol, e.g. `(car 'park)`. This returns a *method*, which actually does the computation you're interested in. You apply the method to the arguments, e.g. `((car 'park) 'carefully)`.

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

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

2.2 The Low Level

An object is a procedure and the environment associated with it. The procedure takes a message (symbol) and returns a procedure that “does the right thing.” The environment associated with the procedure contains all the state of the object — whatever fields the object contains.

When a procedure is created, an instance is built for it. This object just takes a message and passes it to the handler. The instance's handler is set to be the result of `make-foo`. If the object has an `install` method, it is called.

The point of all this is to have one piece of data that represents the object itself — the instance *is* the object. The handler can be changed later. When the object is created, the `make` procedure is called with the instance as `self`.

3 Carry Cdr

Assuming that `the-floor` and `my-desk` are container objects, which we can draw as clouds, what does the environment diagram look like as a result of this?

```
(define spell-book
  (create-mobile-thing 'spell-book
    the-floor))
```

Next, we evaluate the following. What happens?

```
(ask book 'change-location my-desk)
```

4 Appliances

First, we're going to create an appliance class. Appliances can be either on or off, and they have a fuse that can blow if they draw too much current. As an example:

```
(define washing-machine (create-appliance))
(ask washing-machine 'on?) ; #f
(ask washing-machine 'switch 'on)
(ask washing-machine 'on?) ; #t
(ask washing-machine 'blow-fuse)
Bang! Fuse blew!
(ask washing-machine 'on?) ; #f
(ask washing-machine 'switch 'on)
(ask washing-machine 'on?) ; #f

(define (make-appliance self)
  (let ((switch-state 'off)
        (fuse-blown #f)
        (root-part (make-root-object self)))
    (make-handler 'appliance
                  (make-methods

)
root-part)))
```

Now we add new appliance types — a blender and a tv. A blender is just like an appliance, but it also has a speed. A tv is an appliance that also has a channel.

```
(define cuisinart (create-blender))
(ask cuisinart 'set-speed 5)
(ask cuisinart 'speed) ; 5
(ask cuisinart 'on?) ; #f

(define (make-blender self)
  (let ((speed 0)
        (appliance-part (make-appliance self)))
    (make-handler 'blender
                  (make-methods

                    )
                  appliance-part)))
```

```
(define sony (create-tv))
(ask sony 'set-channel 7)
(ask sony 'channel) ; 7
(ask sony 'on?) ; #f

(define (make-tv self)
  (let ((channel 0)
        (appliance-part (make-appliance self)))
    (make-handler 'tv
                  (make-methods

                    )
                  appliance-part)))
```

We realize that lots of people like to watch TV while they cook, and we begin to offer a combination tv/blender (bleegee) appliance — it chops, it dices, it changes channels! The problem is that our fuses are cheap, and if the blender speed goes above 3 while both the tv and blender are on, both fuses blow.

The bleegee should support the messages `blow-if-overload`, `switch`, and `set-speed`, and inherit methods from the tv and blender as appropriate.

```
(define (make-bleegee self)
  (let ((blender-part (make-blender self))
        (tv-part (make-tv self)))
    (make-handler 'bleegee
                  (make-methods
                   )
                  blender-part
                  tv-part)))
```