COMP 210, Fall 2000
Lecture 33 – From Data Hiding to Data Abstraction

Announcements

1. The next homework will not be available until Friday.
2. Missionaries & Cannibals is done. Long live the cannibals.

Moving on …

Finish the example of the address book.

So, you finish typing in all of the **add-to-address-book** calls to enter all of your friends' names into the address book. Your roommate asks to use your computer, and starts writing a Scheme program. Unfortunately, she begins by typing

```
(define address-book empty)
```

What happened to your work? It's gone. Destroyed. Since **set!** is irrevocable, you've lost the entire address book. We'll assume that your roommate did this non-maliciously. However, you can easily see how someone could use **set!** to deliberately change your data–-from destroying your address book or checkbook records to changing the amount of money you think you have in your account, to ….

The problem arises because **set!** really does destroy the value associated with a name (by replacing it with a new value). This being COMP 210, we should design our programs to avoid such misteaks (or abuses). This suggests the following simple rule:

*Only **set!** variables declared in a **local**.*

This point arose briefly when we looked at building a memo function. We want to hide state variables inside the functions that use them. So far, we've talked about this as an issue of style. It is also an issue of security.

We want to hide our persistent data structures inside a local. But what about our address book example? How can we hide address book in such a way that all three programs, **lookup**, **add-to-address-book**, and **update-address-book**, can access it? What options do we have?

1. *We can put address book in each function.* That cannot work, since they will be different address books. Each will have a private copy, and they won't communicate.

2. *Put the three functions that access address book inside a single wrapper function.* This seems unlikely to work; how can we access the functions if we hide them in a local? Let's explore this approach a little more.

```
(define address-interface
   (local [
        (define address-book empty)
        (define (lookup-number name)
           (local [(define matches  (filter (lambda(an-entry)
                                              (symbol=? name (entry-name an-entry)))
                                             address-book))]
                   (cond [(emtpy? matches)  false]
                         [else  (entry-number (first matches))])))
        (define (add-to-address-book name num)
            (begin
                (set!  address-book
                      (cons (make-entry name num)  address-book))
                true)) ]
        (what should it return?)
    ))
```

We could have it return a list of functions, as in

```
        (list   lookup-number  add-to-address-book)
```

This is a bad solution for three reasons:

1. It doesn't scale. As we go from two programs to sixteen, or twenty, or twenty-four, the return list gets long and picking it apart gets complex. The user needs to remember whether the function they want is fifteenth or sixteenth on the list, and then needs to pick the list apart the right way to find the function.
2. It has a bad interface. To use the address book, you must know about all of the functions and where they are in the list. The customer ought to be able to use the address book without learning everything there is to know about the address book.
3. Why should this function return a list of functions? That seems like an incredibly arbitrary decision. We didn't know what else to do, so we wrote down a list of programs? That sort of design practice is contrary to all the claims that we've made for COMP 210 having a rational way to design programs!

If **address-interface** is going to return programs, it should return one program. What if we make it take a symbol as an argument and return a specific function as its result–-depending on the symbol. Thus, we could make it return **lookup-number** for the symbol **'lookup**, and **add-to-**

address-book for the symbol **‘add**. To do this, we should tack onto the local some code like

```
(lambda(service)
   (cond [(symbol=? service ‘lookup)  lookup-number]
         [symbol=? service  ‘add)   add-to-address-book]))
```

[Notice that we didn’t use **else** in the cond clause for ‘add because we will undoubtedly want to add more services at a later time.]

With this addition, calling **(address-interface ‘add)** returns the function **add-to-address-book**. We can use **address-interface** to access the two programs.

We can define some names to hold these functions

```
(define lookup-an-address (address-interface ‘lookup))
(define add-an-address      (address-interface ‘add))
```

```
(lookup-an-address ‘Keith)
➢ false
(add-an-address  ‘Keith  7133486013)
➢ true
(lookup-an-address ‘Keith)
➢ 7133486013
```

The other way that we can use **address-interface** is to invoke the interface directly each time, as in

```
((address-interface ‘lookup)  ‘Keith)
➢ false
((address-interface ‘add) ‘Keith 7133486013)
➢ true
((address-interface ‘lookup) ‘Keith)
➢ 7133486013
```

This looks awful, but works just fine. There are times when you might want to use this kind of interface. We’ll talk about that later. For now, trust me.

How hard is it to extend this address book package? Simple. You just add the new program to the **local**, and add an appropriate clause to the **cond**. The hardest part is implementing the new function, not adding it to the interface function.

This approach **encapsulates** the data inside a function. All of the data is hidden. None of it is directly accessible to the user; the data can only be seen when accessed using one of the supplied functions. (In this case, **‘lookup** and **‘add**.)

This notion of hiding data, or encapsulating it, or providing an **abstract data type**, is a fundamental idea that computer scientists have played with for years. In particular, disciples of ***software engineering*** have long advocated design strategies that limit access to data, that provide access functions similar to **lookup** and **add**, and that prevent unexpected modification (usually by hiding the name of the data from outsiders). This style of program design goes back a long way in the literature of computer science (if anything that is only forty years old has a “long way” to go!)