EECS is everywhere!

- EECS Freshman Open House
  - If you are thinking about majoring in Course 6 (or even curious), please come to the EECS open house, Friday, 3:30 – 5:00 in 34-401
  - Talk with faculty and students about department, about degree programs, about career opportunities
  - Hear about the new curriculum
  - Get free “swag”

6.001 SICP
Object Oriented Programming

- Data Abstraction using Procedures with State
- Message-Passing
- Object Oriented Modeling
  - Class diagrams
  - Instance diagrams
- Example: spacewar simulation

The role of abstractions

- Procedural abstractions
- Data abstractions

Goal: treat complex things as primitives, and hide details

Questions:
- How easy is it to break system into abstraction modules?
- How easy is it to extend the system?
- Adding new data types?
- Adding new methods?

One View of Data

- Data structures
  - Some complex structure constructed from cons cells
  - point, line, 2dshape, 3dshape
  - Explicit tags to keep track of data types
    - (define (make-point x y) (list 'point x y))
  - Implement a data abstraction as set of procedures that operate on the data

“Generic” operations by looking at types:

(define (scale x factor)
  (cond ((point? x) (point-scale x factor))
        ((line? x) (line-scale x factor))
        ((2dshape? x) (2dshape-scale x factor))
        ((3dshape? x) (3dshape-scale x factor))
        (else (error "unknown type")))))

Generic Operations

- Adding new methods
  - Just create generic operations

<table>
<thead>
<tr>
<th></th>
<th>Point</th>
<th>Line</th>
<th>2-dShape</th>
<th>3-dShape</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>point-scale</td>
<td>line-scale</td>
<td>2dshape-scale</td>
<td>3dshape-scale</td>
</tr>
<tr>
<td>translate</td>
<td>point-translate</td>
<td>line-translate</td>
<td>2dshape-translate</td>
<td>3dshape-translate</td>
</tr>
<tr>
<td>color</td>
<td>point-color</td>
<td>line-color</td>
<td>2dshape-color</td>
<td>3dshape-color</td>
</tr>
<tr>
<td>new-op</td>
<td>point-op</td>
<td>line-op</td>
<td>2dshape-op</td>
<td>3dshape-op</td>
</tr>
</tbody>
</table>
Generic Operations

• Adding new methods
  • Just create generic operations
• Adding new data types
  • Must change every generic operation
  • Must keep names distinct

<table>
<thead>
<tr>
<th></th>
<th>Point</th>
<th>Line</th>
<th>2-dShape</th>
<th>3-dShape</th>
<th>curve</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>point-scale</td>
<td>line-scale</td>
<td>2dShape-scale</td>
<td>3dShape-scale</td>
<td>scale-c</td>
</tr>
<tr>
<td>translate</td>
<td>point-trans</td>
<td>line-trans</td>
<td>2dShape-trans</td>
<td>3dShape-trans</td>
<td>trans-c</td>
</tr>
<tr>
<td>color</td>
<td>point-color</td>
<td>line-color</td>
<td>2dShape-color</td>
<td>3dShape-color</td>
<td>color-c</td>
</tr>
<tr>
<td>new-op</td>
<td>point-op</td>
<td>line-op</td>
<td>2dShape-op</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Views Of The World

Data object

Generic operation

Thinking About Data Objects

• A data type, but....
  • it has operations associated with it
  • we want both the generic concept (a line), and a specific instance (line17)
  • the specific instance can have private data associated with it (e.g., its endpoints)

• AKA: object oriented programming

Scheme OOP: Procedures with State

• A procedure has
  • parameters and body as specified by λ expression
  • environment (which can hold name-value bindings!)

• Can use procedure to encapsulate (and hide) data, and provide controlled access to that data
  • Procedure application creates private environment
  • Need access to that environment
  • constructor, accessors, mutators, predicates, operations
  • mutation: changes in the private state of the procedure

Programming Styles – Procedural vs. Object-Oriented

• Procedural programming:
  • Organize system around procedures that operate on data
    (do-something <data> <arg> ...
    (do-another-thing <data>)

• Object-based programming:
  • Organize system around objects that receive messages
    (<object> `do-something <arg>)
    (<object> `do-another-thing)
  • An object encapsulates data and operations (i.e. specific procedures that apply to that object, and handle local state associated with that object)

Object-Oriented Programming Terminology

• Class:
  • specifies the common behavior of entities
  • in scheme, a <type> procedure

• Instance:
  • a particular object or entity of a given class
  • in scheme, an instance is a message-handling procedure made by a create-<type> procedure
Using classes and instances to design a system

• Suppose we want to build a spacewar game
• I can start by thinking about what kinds of objects do I want (what classes, their state information, and their interfaces)
  • ships
  • planets
  • other objects
• I can then extend to thinking about what particular instances of objects are useful
  • Millenium Falcon
  • Enterprise
  • Earth

A Space-Ship Object

```
(define (ship position velocity num-torps)
  (define (move)
    (set! position (add-vect position ...))
  (define (fire-torp)
    (cond ((> num-torps 0) ...)
           (else 'FAIL))
  (lambda (msg)
    (cond ((eq? msg 'POSITION) position)
           ((eq? msg 'VELOCITY) velocity)
           ((eq? msg 'MOVE) (move))
           ((eq? msg 'ATTACK) (fire-torp))
           (else (error "ship can't" msg))))
```

Note the internal state (passed in as parameters in this case), and the object-specific procedures.

Note value returned is procedure with access to internal state

Creating instances of a class

• The definition of ship specifies the properties of a class
  • Every instance of a ship will have its own version of position, velocity, etc., and will have its own procedures for accessing that state
  • Need a mechanism for creating specific instances of this class
  • For now, we will use a simple instantiation – this will get extended in the next lecture

```
(define (create-ship pos vel torp)
  (create-instance ship pos vel torp))
```

Example – Instance Diagram

```
(define enterprise
  (create-ship (make-vector 10 10) (make-vector 5 0) 3))
(define war-bird
  (create-ship (make-vector -10 10) (make-vector 10 0) 10))
```

SPACEWAR: the original video game
first realized on the MIT PDP-1 in 1962
PDP-1 – 100KHz, 4K RAM, $100,000
Example – Environment Diagram
(define enterprise
  (ship (make-vector 10 10) (make-vector 5 0) 3)) ; skipping step
(enterprise 'MOVE) ==> DONE
(enterprise 'POSITION) ==> ? (vec 15 10)

Filling out our World
-- how do we think about programming in this space?*
• Add a PLANET class to our world
• Add predicate messages so we can check type of objects
• Add display handler to our system
  • Draws objects on a screen
  • Can be implemented as a procedure (e.g. draw)
    -- not everything has to be an object!
• Add display message to classes so objects will display
  themselves upon request (by calling draw procedure)

Space-Ship Class

Planet Implementation
(define (planet position)
  (lambda (msg)
    (cond ((eq? msg 'PLANET?) #T)
          ((eq? msg 'POSITION) position)
          ((eq? msg 'DISPLAY) (draw ...))
          (else (error "planet can't" msg))))

This is like our tags in data structures

Keeping time...
• Animate our World!
  • Add a clock that moves time forward in the universe
  • Keep track of things that can move (the *universe*)
  • Clock sends ‘ACTIVATE message to objects to have
    them update their state
• Add TORPEDO class to system

Class Diagram
Coordinating with a clock

CLOCK
- The-time: callbacks:
- THE-TIME
- TICK
- ADD-CALLBACK

CALLBACK
- Object
- message:
- Data
- ...
- ...
- methods
- ...

ACTIVATE

Sends object message and data

Object:
message:
Data:
...
...
...

ACTIVATE

Torpedo Implementation

(define (torpedo position velocity)
  (define (explode torp)
    (display "torpedo goes off!"
    (remove-from-universe torp))
  (define (move)
    (set! position ...)
  (define (me msg . args)
    (cond ((eq? msg 'TORPEDO?) #T)
       ((eq? msg 'POSITION) position)
       ((eq? msg 'VELOCITY) velocity)
       ((eq? msg 'MOVE) (move))
       ((eq? msg 'EXPLODE) (explode (car args)))
       (else (error "No method" msg)))
  (clock 'ADD-CALLBACK)
  (clock-callback 'moveit me 'MOVE))

Variable number of arguments

A scheme mechanism to be aware of:

• Desire:
  (add 1 2)
  (add 1 2 3 4)

• How do this?
  (define (add x y . rest) ...)
  (add 1 2) => x bound to 1
  y bound to 2
  rest bound to ()
  (add 1) => error: requires 2 or more args
  (add 1 2 3) => rest bound to (3)
  (add 1 2 3 4 5) => rest bound to (3 4 5)

Summary, so far...

• Introduced a new programming style:
  • Object-oriented vs. Procedural
  • Uses – simulations, complex systems, ...

• Object-Oriented Modeling
  • Language independent!
    Class – template for state and behavior
    Instances – specific objects with their own identities

• Next: inheritance and delegation
User View: OO System in Scheme

- **Instance**: created by a `create-<type>` procedure (e.g. `create-named-object`)
  - Each instance has its own identity in sense of `eq?`
  - One can invoke methods on the instance:
    
    ```scheme
    (ask <instance> '<message> <arg1> ... argn>)
    ```
  - Default methods for all instances:
    
    ```scheme
    (ask <instance> 'TYPE)
    ⇒ (<type> <supertype> ...)
    ```
    
    ```scheme
    (ask <instance> 'IS-A <some-type>)
    ⇒ <boolean>
    ```

A sidebar on interacting with objects

```scheme
(define [ask object message . args]
  (let ((method (get-method message object)))
    (cond ((method? method)
         (apply method args))
        (else
         (error "No method for" message 'in
                (safe-ask 'UNNAMED-OBJECT object 'NAME))))))
```

```scheme
(define (get-method message . objects)
  (find-method-from-list message objects))
```

```scheme
(define (find-method-from-list message objects)
  (if (null? objects)
      (no-method)   ;; we are suppressing a few details here
       (let ((method (car objects) message)))
      (if (not (eq? method (no-method))
            method
            (find-method-from-list (cdr objects))))))
```

This is just sending a message to an object together with some arguments
**OO System in Scheme**

- Named-object inherits from our root class
  - Gains a "self" variable: each instance can refer to itself
  - Gains an IS-A method
  - Specializes a TYPE method

**Alternative case syntax for message match:**

- case is more general than this (see Scheme manual), but our convention for message matching will be:

```scheme
(case message
  ((<msg-1>) <method-1>)
  ((<msg-2>) <method-2>)
  ...
  ((<msg-n>) <method-n>)
  (else <expr>))
```

**User View: Using an Instance in Scheme**

```scheme
(define x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGE-NAME 'sicp-2nd-ed)
(ask x 'NAME) => sicp-2nd-ed
(ask x 'TYPE) => (named-object root)
(ask x 'IS-A 'NAMED-OBJECT) => #t
(ask x 'IS-A 'CLOCK) => #f
```

**An Intermediate Step: Message Handlers**

- Object behaviors are specified using message-handlers
- Response to every message is a method
- A method is a procedure that can be applied to actually do the work

```scheme
(define (make-named-object-handler name)
  (lambda (message)
    (case message
      ((NAME)
        (lambda () name))
      ((CHANGE-NAME)
        (lambda (new-name) (set! name new-name)))
      (else (no-method))))
```

**An Intermediate Step: Handler with case syntax**

- Object behaviors are specified using message-handlers
- Response to every message is a method
- A method is a procedure that can be applied to actually do the work

```scheme
(define (make-named-object-handler name)
  (lambda (message)
    (case message
      ((NAME)
        (lambda () name))
      ((CHANGE-NAME)
        (lambda (new-name) (set! name new-name)))
      (else (no-method))))
```
Big Step: User’s View of Class Definition

- A class is defined by a `<type>` procedure
- Inherited classes
- Local state (must have "self" as first argument)
- Message handler with messages and methods for the class
  - must have a `TYPE` method as shown
  - must have `(else (get-method ...))` case to inherit methods

```scheme
(define (<type> self <arg1> <arg2> ... <argn>)
  (let ((<super1>-part (<super1> self <args>))
        (<super2>-part (<super2> self <args>))
        <other superclasses>
        <other local state>)
    (lambda (message)
      (case message
        ((TYPE) (lambda ()
          (type-extend '<type> <super1>-part
                       <super2>-part ...)) )
        <other messages and methods>
        (else (get-method message <super1>-part
                        <super2>-part ...))))))
```

We will eventually replace this with some cleaner code.

User’s View: Instance Creation

- User should provide a `create-<type>` procedure for each class
- Uses the `create-instance` higher order procedure to
  - Generate an instance object
  - Make and add the `message handler` for the object
  - Return the instance object
- An instance is created by applying the `create-<type>` procedure

```scheme
(define (create-<type> <arg1> <arg2> ... <argn>)
  (create-instance <type> <arg1> <arg2> ... <argn>))
```

User’s View: Instance Creation

- User should provide a `create-<type>` procedure for each class
- Uses the `create-instance` higher order procedure to
  - Generate an instance object
  - Make and add the `message handler` for the object
  - Return the instance object
- An instance is created by applying the `create-<type>` procedure

```scheme
(define (create-<type> <arg1> <arg2> ... <argn>)
  (create-instance <type> <arg1> <arg2> ... <argn>))
```

Another Example: NAMED-OBJECT Class

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

```scheme
(define (named-object self name)
  (let ((root-part (root-object self)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'named-object root-part)))
        ((NAME) (lambda () name))
        ((CHANGE-NAME) (lambda (newname) (set! name newname)))
        (else (get-method message root-part))))))
```

• In this example, named-object only inherits from root-object

User’s View: Using an Instance

- Method lookup: `get-method` for `<MESSAGE>` from instance
- Method application: apply that method to method arguments
- Can do both steps at once:
  - ask an instance to do something

```scheme
(define <inst> (create-<type> <arg1> <arg2> ... <argm>))
```

```scheme
(define some-method (get-method <instance> '<MESSAGE>))
```

```scheme
(define <instance> (create-<type> <arg1> <arg2> ... <argm>))
```

User’s View: Type System

- With inheritance, an instance can have multiple types
  - all objects respond to `TYPE` message
  - all objects respond to `IS-A` message

```scheme
(ask a-instance 'IS-A 'C) => #f
(ask a-instance 'IS-A 'B) => #f
(ask a-instance 'IS-A 'A) => #t
(ask a-instance 'IS-A 'root) => #t
(ask c-instance 'IS-A 'C) => #f
(ask c-instance 'IS-A 'B) => #f
(ask c-instance 'IS-A 'A) => #t
(ask c-instance 'IS-A 'root) => #t
```