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

<table>
<thead>
<tr>
<th>Operation</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>
</tbody>
</table>

Generic Operations

<table>
<thead>
<tr>
<th>Operation</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>
</tbody>
</table>
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-trans</td>
<td>line-trans</td>
<td>2dShape-trans</td>
<td>3dShape-trans</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>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Generic Operations

- Adding new methods
  - Just create generic operations

Views of The World

![Data object]

- 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

Thinking About Data Objects

- A procedure has
  - parameters and body as specified by \( \lambda \) 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

Scheme OOP: Procedures with State

- A procedure has
  - parameters and body as specified by \( \lambda \) 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.

Object-Oriented Programming Terminology

- **Class:**
  - Specifies the common behavior of entities
  - In Scheme, a "maker" procedure

- **Instance:**
  - A particular object or entity of a given class
  - In Scheme, an instance is a message-handling procedure made by the maker 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

```Scheme
(define (make-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)))))
```

Space-Ship Class

```
<table>
<thead>
<tr>
<th>class</th>
<th>private state</th>
<th>public messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHIP</td>
<td>position:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>velocity:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>num-torps:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>POSITION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VELOCITY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOVE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ATTACK</td>
<td></td>
</tr>
</tbody>
</table>
```

SPACEWAR: the original video game
first realized on the MIT PDP-1 in 1962
PDP-1 – 100KHz, 4K Ram, $100,000

A Space-Ship Object

```Scheme
(define (make-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))))
```
Example – Instance Diagram

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

Example – Environment Diagram

```
(define enterprise
  (make-ship (make-vect 10 10) (make-vect 5 0) 3))
(enterprise 'MOVE) ==> DONE
(enterprise 'POSITION) ==> (vec 15 10)
```

Filling out our World

- 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)

Planet Implementation

```
; (define (make-planet position)
  (lambda (msg)
    (cond ((eq? msg 'PLANET?) #T)
          ((eq? msg 'POSITION) position)
          ((eq? msg 'DISPLAY) (draw ...))
          (else (error "planet can't" msg))))
```

Space-Ship Class

```
SHIP
  position:
  velocity:
  num-torps:

POSITION
  VELOCITY
  MOVE
  ATTACK

SHIP?

DISPLAY

PLANET
  position:

POSITION

PLANET?

DISPLAY
```

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

SHIP
position:
velocity:
um-torps:

POSITION
VELOCITY
MOVE
SHIP?
DISPLAY
CLOCK-TICK
EXPLODE

TORPEDO
position:
velocity:

TORPEDO?
POSITION
VELOCITY
MOVE
DISPLAY
CLOCK-TICK
EXPLODE

PLANET
position:

POSITION
PLANET?
CLOCK-TICK
DISPLAY

The Universe and Time

(define (make-clock . args)
  (let ((the-time 0)
         (callbacks '()))
    (lambda (message)
      (case message
        ((CLOCK?) (lambda (self) #t))
        ((NAME) (lambda (self) name))
        ((THE-TIME) (lambda (self) the-time))
        ((TICK)
          (lambda (self)
            (map (lambda (x) (ask x 'activate)) callbacks)
            (set! the-time (+ the-time 1))))
        ((ADD-CALLBACK)
          (lambda (self cb)
            (set! callbacks (cons cb callbacks))
            'added))
        (else (error "No method" message))))

Torpedo Implementation

(define (make-torpedo position velocity)
  (define (explode torp)
    (display "torpedo goes off!"
    (remove-from-universe torp))
  (define (move)
    (set! position ...))
  (ask clock 'ADD-CALLBACK
       (make-clock-callback 'moveit me 'MOVE))
  (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)))
          ((eq? msg 'DISPLAY) (draw ...))
          (else (error "No method" msg))))
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
  
  Class – template for state and behavior
  Instances – specific objects with their own identities

- Next: inheritance and delegation

Abstract View – Class/Instance Diagrams

Class Diagram

Instance Diagram

Abstract View – with Inheritance

Class Diagram

Instance Diagram

Abstract View: Multiple Inheritance

- Superclass & Subclass
  
  A is a superclass of C
  
  C is a subclass of both A & B
  
    - C "is-a" B
    - C "is-a" A

- A subclass inherits the state variables and methods of its superclasses
  
  - Class C has methods ACK, BAR, and COUGH

User View: OO System in Scheme

- Class: defined by a make-type procedure
  
  - Defines what is common to all instances of that class
    - Provides local state variables
    - Provides a message handler to implement methods
    - Specifies what superclasses and methods are inherited

- Root class: root-object
  
  - All user defined classes should inherit from either root-object class or from some other superclass

- Types:
  
    - Each class should specialize the TYPE method
User View: OO System in Scheme

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

User View: Using an Instance in Scheme

```
(define x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGE-NAME 'sicp-2nd-ed)
(ask x 'NAME) => sicp-2nd-ed)
```

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

```
define z (create-book 'sicp 1996))
(ask z 'YEAR) => 1996
(ask z 'NAME) => sicp
(ask z 'IS-A 'BOOK) => #t
(ask z 'IS-A 'NAMED-OBJECT) => #t
```

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

```
define (make-named-object-handler name)
  (lambda (message)
    (cond ((eq? message 'NAME)
           (lambda () name))
          ((eq? message 'CHANGE-NAME)
           (lambda (new-name) (set! name new-name))
           (else (no-method)))))
```

Alternative case syntax for message match:

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

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

```
(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 make-<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

```
(define (make-<type> self <arg1> <arg2> … <argn> )
  (let ((<super1>-part (make-<super1> self <args>))
        (<super2>-part (make-<super2> self <args>))
        …)
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend '<type> <super1>-part ...
                              <other superclasses> <other local state>)
                   (lambda (message)
                    (case message
                      ((TYPE) (lambda ()
                           (type-extend 'book named-object))
                        …)
                      (else (get-method message <super1>-part ...
                              <super2>-part ...))))))
```

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

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

User’s View Example: BOOK Class with Inheritance

```
; create-book: symbol, number -> book
(define (create-book name copyright)
  (create-instance make-book name copyright))
```

Another Example: NAMED-OBJECT Class

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

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

```
(define <inst> (create-<type> <arg1> <arg2> ... <argm>))
(define some-method (get-method <inst> '<MESSAGE>))
(some-method <m-arg1> <m-arg2> … <m-argm>)
```

User’s View Example: BOOK Class with Inheritance

```
; create-book: symbol, number -> book
(define (create-book name copyright)
  (create-instance make-book name copyright))
```
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
(define a-instance (create-A))
(define c-instance (create-C))
(ask a-instance 'TYPE) => (A root)
(ask c-instance 'TYPE) => (C A B root)
(ask c-instance 'IS-A 'C) => #t
(ask c-instance 'IS-A 'B) => #t
(ask c-instance 'IS-A 'A) => #t
(ask c-instance 'IS-A 'root) => #t
(ask a-instance 'IS-A 'C) => #f
(ask a-instance 'IS-A 'B) => #f
(ask a-instance 'IS-A 'A) => #t
```

Different Views of Object-Oriented System

• An abstract view
  • class and instance diagrams
  • terminology: messages, methods, inheritance, superclass, subclass, ...

• Scheme OO system user view
  • conventions on how to write Scheme code to:
    – define classes
    – inherit from other classes
    – create instances
    – use instances (invoke methods)

Next lecture:
⇒ Scheme OO system implementer view (under the covers)
  • How implement instances, classes, inheritance, types