6.037 Lecture 6
Implementation of Object Oriented Programming Systems

Mike Phillips mpp@mit.edu

Some slides originally by Prof. Eric Grimson

IAP 2018

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 modules?
• How easy is it to extend the system?
• Adding new data types?
• Adding new methods?

Overview
• Data abstraction, a few ways
• Object-Oriented Programming
  – What it is, and how to implement it:
    • via Procedures with State (Closures)
    • via simpler data structures

One View of Data
• Data structures
  – Some complex structure constructed from cons cells
    • point, line, 2shape, 3shape
  – Explicit tags to keep track of data types
    • (define (make-point x y) (list 'point x y))
    • Implement a data abstraction as a set of procedures that operate on the data
  • "Generic" operations by dispatching on type:

(define (scale x factor)
  (cond ((point? x) (point-scale x factor))
        ((line? x) (line-scale x factor))
        ((2shape? x) (2shape-scale x factor))
        ((3shape? x) (3shape-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>2shape-scale</td>
<td>3shape-scale</td>
</tr>
<tr>
<td>translate</td>
<td>point-trans</td>
<td>line-trans</td>
<td>2shape-trans</td>
<td>3shape-trans</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>
</tbody>
</table>

Two Views of Data

Data Objects

<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</td>
</tr>
<tr>
<td>Translate</td>
<td>point-trans</td>
<td>line-trans</td>
<td>2dshape-trans</td>
<td>3dshape-trans</td>
<td>translate</td>
</tr>
<tr>
<td>Color</td>
<td>point-color</td>
<td>line-color</td>
<td>2dshape-color</td>
<td>3dshape-color</td>
<td>color</td>
</tr>
</tbody>
</table>

Generic Operations

- Adding new methods
- Just create generic operations

Object-Oriented Programming Terminology

- **Class:**
  - Template for state and behavior
  - Internal state (fields), operations (methods), relationships to other classes

- **Instance:**
  - A particular object or entity of a given class
  - The result of “instantiating” a class
  - Has its own identity separate from other instances
Using classes and instances

- Suppose we wanted to build *Spacewar!*
- Start by thinking about what kinds of objects should exist (state and interfaces)
  - Planets
  - Ships
- Think about useful instances of these
  - Centauri Prime
  - Enterprise
Abstract View – with Inheritance

- Name method is overridden
- Might want to call superclass'

A FANCY-OBJECT reports its name with hearts and stars before and after it

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 and methods of its superclasses
  - Class C has methods ACK, BAR, and COUGH

Different Views of an Object-Oriented System

- An abstract view
  - class and instance diagrams
  - terminology: methods, inheritance, superclass, subclass, abstract class, interfaces, traits, mixins...

- 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)

- Scheme OO system implementer view
  - How do instances, classes, inheritance, and types work?

Abstract View – with Inheritance

- Suppose the PARTY method calls the DANCE method
- If we override DANCE, and then ask an instance of DANCER to PARTY, which DANCE method runs?

Abstract View: Multiple Inheritance

- Diamond Inheritance Problem
  - Which BAR do you get from C?
  - Should this be allowed?

Object-Oriented Design & Implementation

- Focus on classes
  - Relationships between classes
  - Kinds of interactions that need to be supported between instances of classes

- Careful attention to behavior desired
  - Inheritance of methods
  - Explicit use of superclass methods
  - Shadowing of methods to override default behaviors
Implementation #1

- A procedure has
  - parameters and body as specified by λ expression
  - environment (which can hold name-value bindings!)
- Encapsulate data, and provide controlled access
- Applying a procedure creates a private environment
- Need access to that environment
  - constructor, accessors, mutators, predicates, operations
  - mutation: changes in the private state of the procedure

A Space-Ship Object

(define (make-ship position velocity num-torps)

(define (move)

(set! position (add-vect position ...)))

(define (fire-torp)

(cond ((> num-torps 0) ...)

(else 'FAIL)))

(lamdba (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)])))

Missing elements

- What about inheritance?
- How do I call another method on myself?
  - Or from my superclass?

Environment Diagram

```
(define enterprise

(make-ship (make-vect 10 10) (make-vect 5 0) 3))

(enterprise 'MOVE)
```

OO System View in Scheme with Inheritance

```
(define enterprise

(make-ship (make-vect 10 10) (make-vect 5 0) 3))

(enterprise 'MOVE)
```

Implementer’s View of this in Environment Model

```
(define enterprise

(make-ship (make-vect 10 10) (make-vect 5 0) 3))

(enterprise 'MOVE)
```
Implementation #1 Summary

- Implemented with procedures doing message dispatch
- All methods are public
- All state is private
- Could support multiple inheritance
- Objects are first class
- Classes are not

User view: Class Definition

- Classes are created by applying

```scheme
define make-class
  (make-class 'NAMED-OBJECT 'name) root-object
  (make-methods
    'CONSTRUCTOR
      (lambda (self super name)
        (write-state! self 'name name))
    'NAME
      (lambda (self super)
        (read-state self 'name)))))
This means classes are first-class objects
```

User View: Object Instantiation

- Apply make-instance to instantiate an object
- Extra arguments are passed to the CONSTRUCTOR method

```
define book
  (make-class 'BOOK '(copyright) named-object
  (make-methods
    'CONSTRUCTOR
      (lambda (self super name year)
        (super 'CONSTRUCTOR name)
        (write-state! self 'name name))
    'NAME
      (lambda (self super)
        (read-state self 'name)))))
```

User View: Method invocation

- Use the invoke procedure with method name and optional parameters

```
(invoke sicp 'YEAR)
=> 1996
```

Implementation #2

- Simple data structure approach
  - Easier for user to use, in some ways
  - Easier for implementer to implement
    - And to play with!
  - May be more/less/differently powerful

User view: Class Definition

- Call methods with "invoke" on "self"
- Shadowed methods accessed via "super"

```
define book
  (make-class 'BOOK 'copyright) named-object
  (make-methods
    'CONSTRUCTOR
      (lambda (self super name year)
        (super 'CONSTRUCTOR name)
        (write-state! self 'copyright year))
    'YEAR
      (lambda (self super)
        (read-state self 'copyright))
    'NAME
      (lambda (self super)
        (list (super 'NAME)
          'Copyright
          (invoke self 'YEAR)))))
```

User View: Object Instantiation

```
define sicp
  (make-instance book 'SICP 1996))
```
Implementer’s view: Classes

• Data abstraction for a Class:

(define (make-class type state parent methods)
  (list 'class type state parent methods))

(define (class? obj)
  (tagged-list? obj 'class))

(define (class-type class)
  (second class))

(define (class-state class)
  (third class))

(define (class-parent class)
  (fourth class))

(define (class-methods class)
  (fifth class))

Aside: Using apply

(define (foo a b c)
  (+ a b c))

(foo 1 2 3)
=> 6

(foo '(1 2 3))
=> error: Too few arguments

(apply foo '(1 2 3))
=> 6

(apply foo 1 2 '(3))
=> 6

Aside: Variable number of arguments

A scheme mechanism to be aware of:

Desire:

(add 1 2)
(add 1 2 3 4)

How do we 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)

Implementer’s View: Methods

• Methods are procedures that take self, super, and optionally other arguments

• Classes store an association-list of method names and procedures

  ((NAME <#procedure>)
   (YEAR <#procedure> ... )

User’s View: Method list

(define book
  (make-class 'BOOK '(copyright) named-object
    (make-methods
      'CONSTRUCTOR
        (lambda (self super name year)
          (super 'CONSTRUCTOR name)
          (write-state! self 'copyright year))
      'YEAR
        (lambda (self)
          (read-state self 'copyright))
      'NAME
        (lambda (self)
          (list (super 'NAME)
            'Copyright (invoke self 'YEAR))))))

Implementer’s View: Method list

• Helper for constructing methods: From easy to type to an association list

  (define (make-methods . args)
    (define (mhelper lst result)
      (cond ((null? lst) result)
            ((null? (cdr lst))
             (error "unmatched method (name,proc) pair")
            ((not (symbol? (car lst)))
             (error "invalid method name" (car lst)))
            ((not (procedure? (cadr lst)))
             (error "invalid method procedure" (cadr lst)))
            (else
             (mhelper (cddr lst)
              (cons (list (car lst) (cadr lst)) result))))
    (mhelper args '()))
Implementer's View: Instances

- Data abstraction for an instance

```scheme
(define (make-instance class . args)
  (let ((inst
         (list 'instance
               (map (lambda (x) (list x #f)) (collect-state class))
               class)))
    (if (has-method? inst 'CONSTRUCTOR)
        (apply invoke inst 'CONSTRUCTOR args)
        inst)))

(define (instance? obj)
  (tagged-list? obj 'instance))

(define (instance-state inst)
  (second inst))

(define (instance-class inst)
  (third inst))
```

Implementer's View: State

```scheme
(define (read-state self varname . default)
  (let ((result
         (assq varname (instance-state self))))
    (if result
        (cadr result)
        (if (not (null? default))
            (car default)
            (error "no state named" varname)))))

(define (write-state! self varname value)
  (let ((result
         (assq varname (instance-state self))))
    (if result
        (set-car! (cdr result) value)
        (error "no state named" varname))))
```

User's View: Method Invocation

```scheme
(define book
  (make-class 'BOOK '(copyright) named-object
              (make-methods
                'CONSTRUCTOR
                (lambda [self super name year]
                  (super 'CONSTRUCTOR name)
                  (write-state! self 'copyright year))
                'YEAR
                (lambda [self super]
                  (read-state self 'copyright))
                'NAME
                (lambda [self super]
                  (list (super 'NAME)
                        'Copyright
                        (invoke self 'YEAR))))))
```

Implementer's View: Method Invocation

```scheme
(define (invoke instance method . args)
  (method-call instance (instance-class instance) method args))
```
Implementer's View: Method Invocation

(define (method-call self class method args)
  (if (class? class)
      (let ((proc (find-class-method method class))
            (super (make-super class self)))
        (if proc
            (apply proc self super args)
            (method-call instance (class-parent class)
                             method args)))
      (error "no such method" method)))

(define (make-super class self)
  (lambda (method . args)
    (method-call self (class-parent class) method args)))

Implementer's view: State

(define (read-state varname . default)
  (let ((result
            (assq varname (instance-state self)))
        (if result
            (cadr result)
            (if default
                (car default)
                (error "no state named" varname))))))

Dynamic scoping

• Want to have dynamic scoping just for self and super
• We want to bind specific values for the duration of the method invocation only
• Could we define self and super in the GE and then change it before a method call and reset it after?

Implementation oddities

• All methods are public
• All state is public
  – Would be easy to violate the abstraction barrier
  – Would be better if read-state/write-state! only worked from within method bodies

Implementation oddities

• Methods require explicit self and super
  – Why can’t self and super just “have the right value” while the method is executing?
  – We want to be able to refer to these free variables in our methods without passing them around
  – Actual value depends on the calling context, not the program text

Dynamic scoping: Actually useful

(define self #f)
(define super #f)

(define (invoke instance method . args)
  (fluid-let ((self instance))
    (method-call (instance-class instance) method args)))
Dynamic scoping: Actually useful

```
(define (method-call class method args)
  (if (class? class)
      (let ((proc (find-class-method method class)))
        (if proc
            (fluid-let ((super (make-super class)))
              (apply proc args))
            (method-call (class-parent class) method args)))
      (error "no such method" method)))
```

User view: Class Definition

- With our dynamic `self` and `super`

```
(define book
  (make-class 'BOOK '(copyright) named-object
    (make-methods
      'CONSTRUCTOR
        (lambda (name year)
          (super 'CONSTRUCTOR name)
          (write-state! 'copyright year))
      'YEAR
        (lambda ()
          (read-state 'copyright))
      'NAME
        (lambda ()
          (list (super 'NAME)
            'Copyright
            (invoke self 'YEAR))))))
```

Where do we go from here?

- Current idea provides a “library” of procedures to give OOP behavior
- What if you wanted it to be part of the language itself, with custom syntax?
  - Macros (`define-syntax ...`)
  - Extend m-eval (Problem Set 4)
    * Do better than `read-state` and `write-state`!

MetaObject Protocol (MOP)

- Gives programmer access to objects and classes
  - Introspection: Look up fields, methods
  - Intercession: Modify the behavior of an object
- Macros
  - Extend m-eval (Problem Set 4)
- Metaobject protocol
  - The “class of a class” (i.e. classes are objects)
  - Can expose how the OOP system works

Where do we go from here?

- What other features might you want?
  - Allow some public state access?
  - Private/protected methods?
  - Metaobject protocol?

Recitation Time!

- Problem Set 4 released after class
  - Implement an OO system in m-eval
  - Text Adventure Game
  - It will take a good deal of time
  - Lots of room for optional exploration