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

- Tagged data:
  - Some complex structure constructed from cons cells
  - Explicit tags to keep track of data types
  - Implement a data abstraction as set of procedures that operate on the data

<table>
<thead>
<tr>
<th>Operation 1</th>
<th>Data type 1</th>
<th>Data type 2</th>
<th>Data type 3</th>
<th>Data type 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some proc</td>
<td>Some proc</td>
<td>Some proc</td>
<td>Some proc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation 2</th>
<th>Data type 1</th>
<th>Data type 2</th>
<th>Data type 3</th>
<th>Data type 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some proc</td>
<td>Some proc</td>
<td>Some proc</td>
<td>Some proc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation 3</th>
<th>Data type 1</th>
<th>Data type 2</th>
<th>Data type 3</th>
<th>Data type 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some proc</td>
<td>Some proc</td>
<td>Some proc</td>
<td>Some proc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation 4</th>
<th>Data type 1</th>
<th>Data type 2</th>
<th>Data type 3</th>
<th>Data type 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some proc</td>
<td>Some proc</td>
<td>Some proc</td>
<td>Some proc</td>
<td></td>
</tr>
</tbody>
</table>

---

Factoring Operation/Type Association

- "Generic" operations by looking at types:

```scheme
(define (scale x factor)
  (cond ((number? x) (* x factor))
        ((line? x)   (line-scale x factor))
        ((shape? x)  (shape-scale x factor))
        (else (error "unknown type")))))
```

---

Dispatch on Type

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

---

An Alternative View of Data: Procedures with State

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

---

An Alternative View of Data: 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
Example: Pair as a Procedure with State

```scheme
(define (cons x y)
  (lambda (msg)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          (else (error "pair cannot" msg))))

(define (car p)   (p 'CAR))
(define (cdr p)   (p 'CDR))
(define (pair? p)
  (and (procedure? p) (p 'PAIR?)))
```

Example: What is our "pair" object?

```scheme
(define foo (cons 1 2))
```

```
```

Example: Pair Mutation as Change in State

```scheme
(define (cons x y)
  (lambda (msg)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          ((eq? msg 'SET-CAR!)
            (lambda (new-car) (set! x new-car)))
          ((eq? msg 'SET-CDR!)
            (lambda (new-cdr) (set! y new-cdr)))
          (else (error "pair cannot" msg))))

(define (set-car! p new-car)
  ((p 'SET-CAR!) new-car))
(define (set-cdr! p new-cdr)
  ((p 'SET-CDR!) new-cdr))
```

Example: Mutating a pair object

```scheme
(define bar (cons 3 4))
```

```
```

Message Passing Style - Refinements

- lexical scoping for private state and private procedures

```scheme
(define (cons x y)
  (lambda (msg)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          ((eq? msg 'SET-CAR!)
            (lambda (new-car) (set! x new-car)))
          ((eq? msg 'SET-CDR!)
            (lambda (new-cdr) (set! y new-cdr)))
          (else (error "pair cannot" msg))))

(define (car p)   (p 'CAR))
(define (set-car! p val)
  (p 'SET-CAR! val))
```

Variable number of arguments

A scheme mechanism to be aware of:

- Desire:
  ```scheme
  (add x y . rest)  ...
  ```
  ```scheme
  (add 1 2)  =>  x bound to 1
  y bound to 2
  (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)
  ```
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-oriented 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 <type> procedure that makes instances
    of this type when called with the initial values of the
    state variables of the instance

• Instance:
  – A particular object or entity of a given class
  – in Scheme, we implement an instance as the message-
    handling procedure made by the maker procedure
  – (Slightly more complex in Project 4.)

Using classes & instances to design a system

• Suppose we want to build a “space wars” simulator
• We can start by thinking about what kinds of objects we
  want (what classes, their state information, and their
  interfaces)
  – ships
  – space-stations
  – other objects
• We can then extend to thinking about what particular
  instances of objects are useful
  – Millenium Falcon
  – Enterprise
  – Babylon3

Space-Ship Class of Objects

(define (ship position velocity num-torps)
  (define (move)
    (set! position (add-vect position velocity)))
  (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

SHIP
position:
velocity:
um-torps:
POSITION
VELOCITY
MOVE
ATTACK
Example – Instance Diagram

(define enterprise
  (ship (make-vect 10 10) (make-vect 5 0) 3))
(define falcon
  (ship (make-vect -10 10) (make-vect 10 0) 8))

SHIP
pos: (vec 10 10)
vel: (vec 5 0)
num-torps: 3

SHIP
pos: (vec -10 10)
vel: (vec 10 0)
num-torps: 8

Example – Environment Diagram

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

Some Extensions to our World

• Add more classes to our world
  – a SPACE-STATION class
  – a TORPEDO class

• 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)

Station Implementation

(define (station position)
  (lambda (msg)
    (cond ((eq? msg 'POSITION) position)
          ((eq? msg 'DISPLAY) (draw ...))
          (else (error "station can't" msg)))))

Add Torpedo Class

(define (torpedo)
  (lambda (msg)
    (cond ((eq? msg 'POSITION) (set! position ...))
          ((eq? msg 'VELOCITY) (set! velocity ...))
          ((eq? msg 'MOVE) (set! position ...))
          ((eq? msg 'ATTACK) (set! position ...))
          ((eq? msg 'DISPLAY) (draw ...))
          (else (error "torpedo can't" msg))))

Add Space-Station Class

(define (station position)
  (lambda (msg)
    (cond ((eq? msg 'POSITION) position)
          ((eq? msg 'DISPLAY) (draw ...))
          (else (error "station can't" msg)))))

Add Torpedo Class

(define (torpedo)
  (lambda (msg)
    (cond ((eq? msg 'POSITION) (set! position ...))
          ((eq? msg 'VELOCITY) (set! velocity ...))
          ((eq? msg 'MOVE) (set! position ...))
          ((eq? msg 'ATTACK) (set! position ...))
          ((eq? msg 'DISPLAY) (draw ...))
          (else (error "torpedo can't" msg))))
Torpedo Implementation

(define (torpedo position velocity)
  (define (explode torp)
    (display "torpedo goes off!")
    (remove-from-universe torp))

(define (move)
  (set! position ...))

(lambda (msg . args)
  (cond ((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)))))

Application: Running a Simulation

;; Build some things
(define babylon3 (station (make-vect 0 0)))
(define enterprise
  (ship (make-vect 10 10) (make-vect 5 0) 3))
(define falcon
  (ship (make-vect -10 10) (make-vect 10 0) 8))

;; Run a simulation
(define "the-universe" (list babylon3 enterprise falcon))
(init-clock "the-universe")
(run-clock 100)

... and magical things happen on a display near you ...

Elements of Object-Oriented Programming

• Object
  – "Smart" data structure
  • Set of state variables
  • Set of methods for manipulating state variables

• Class:
  – Specifies the common structure and behavior of entities

• Instance:
  – A particular object or entity of a given class

Space War Class Diagram

Ships and torpedoes have some behavior that is the same — is there way to capture this commonality?

Space War Class Diagram with Inheritance

• SHIP class is a specialization or sub-class of the MOBILE-THING class
  – SHIP is-a MOBILE-THING
  – SHIP inherits the state and behavior of MOBILE-THING

• MOBILE-THING class is a super-class of the SHIP and TORPEDO classes

Elements of OOP

• Object
  – "Smart" data structure
  • Set of state variables
  • Set of methods for manipulating state variables

• Class:
  – Specifies the common structure and behavior of instances
  – Inheritance to share structure and behaviors between classes

• Instance:
  – A particular object or entity of a given class

• Next time: a full-bodied object-oriented system implementation
  – In particular, how to incorporate inheritance
Re-Interpreting Conventional Programming as OOP

• Smalltalk, 1972:
  – The expression "a+b" means "send a the message + and argument b"

• Types of problems for which OOP seems particularly natural
  – Simulation (Simula, 1962-7)
  – GUI: Windowing systems such as X, MacOS, Windows
  – Web state (e.g., Shopping Cart)
  – Business Process Models