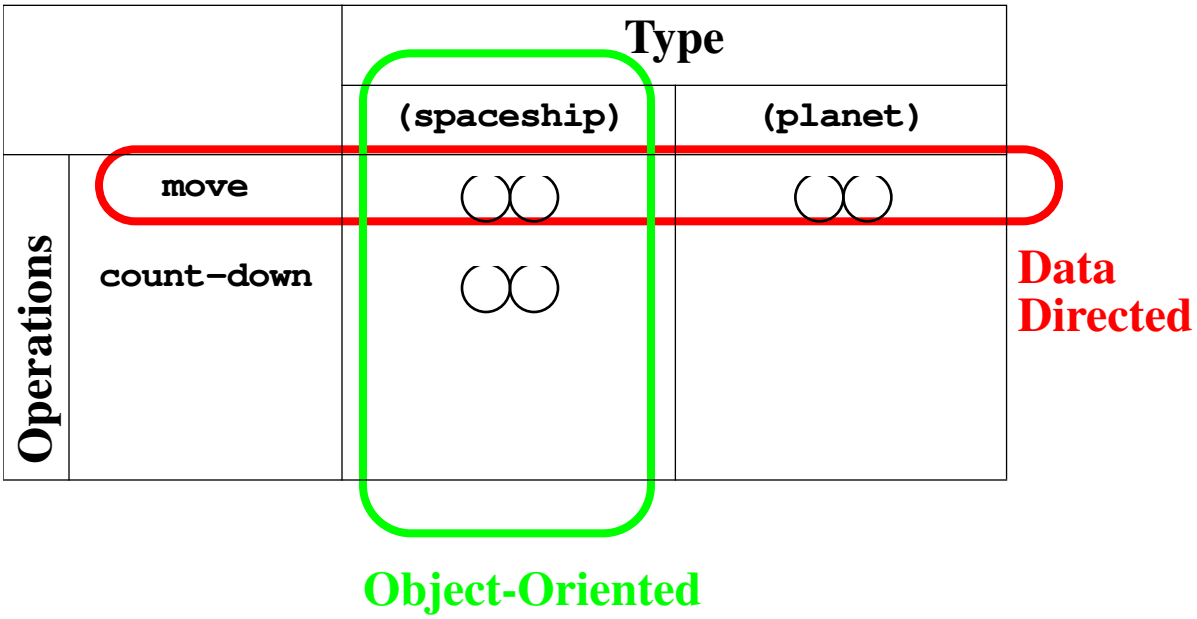
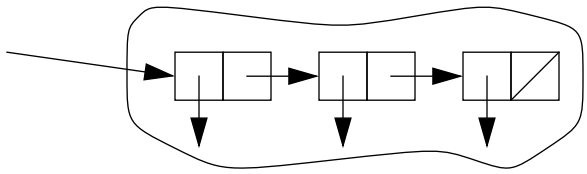
Managing Large Systems



- Operation-Centric:
  - Generic operations
  - Dispatch on type
  - Data-directed programming
- Type/Object-Centric:
  - Message passing
  - Object-oriented programming (today!)

Data-Directed Ship Implementation

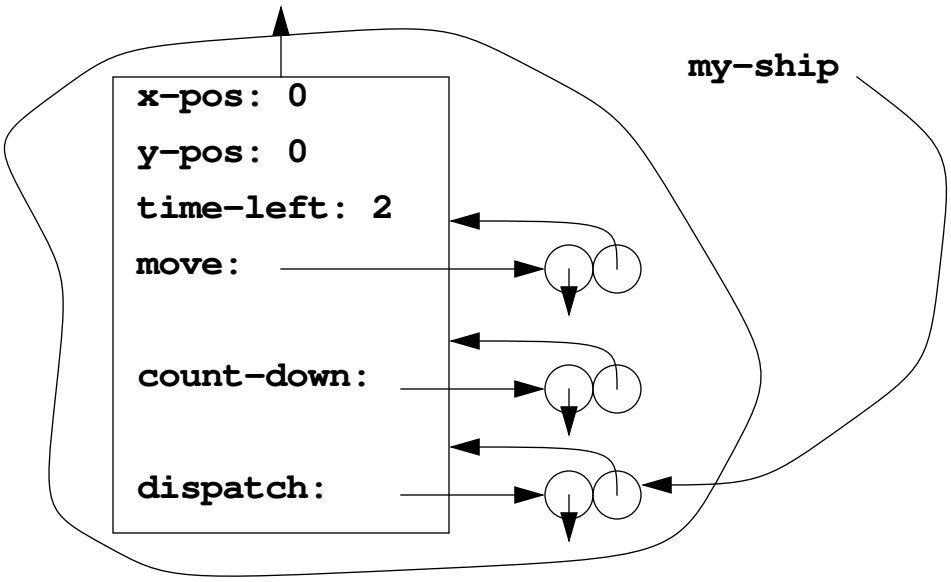



```
(define (install-ship-package)
  ;; Internal representation
  (define (make-ship x y time) (list x y time))
  (define (ship-x ship) (car ship))
  (define (set-ship-x! ship new-x)
    (set-car! ship new-x))
  ...
  (define (move ship dx dy)
    (set-ship-x! (+ (ship-x ship) dx))
    (set-ship-y! (+ (ship-y ship) dy))
    (list (ship-x ship) (ship-y ship)))
  (define (count-down ship)
    (let ((time (ship-time-left ship)))
      (set-time-left! (- time 1))
      (if (<= time 0) 'blast-off time) ;bug!

  ;; External representation - tagged object
  (define (tag x) (attach-tag 'spaceship x))
  (put 'make 'spaceship
    (lambda (x y t) (tag (make-ship x y t))))
  (put 'move 'spaceship
    (lambda (s dx dy) (tag (move s dx dy))))
  (put 'count-down 'spaceship count-down)
  'done
  )

(define (move obj dx dy)
  (apply-generic 'move obj dx dy)
```

Message-Passing Ship Implementation




```
(define (make-spaceship x-pos y-pos time-left)
  (define (move dx dy)
    (set! x-pos (+ x-pos dx))
    (set! y-pos (+ y-pos dy))
    (list x-pos y-pos))
  (define (count-down)
    (set! time-left (- time-left 1))
    (if (<= time-left 0)
        'blast-off
        time-left))  ; no bug
  (define (dispatch message)
    (cond ((eq? message 'move) move)
          ((eq? message 'count-down) count-down)
          (else (error "No method" message))))
  dispatch)

(define enterprise (make-spaceship 0 0 2))
((enterprise 'move) 1 2) ==> (1 2)
```

Object-Oriented Programming

- 1960's: Simula
- 1970's: Smalltalk
- 1980's: C++
- 1990's: Java

Individual entities or **objects** which are categorized into groups or **classes** that behave similarly, but with individual differences based on internal state of each **instance**.

Properties of an Object

1. Instances have **Identity**: in sense of eq?
   - Objects instances as Scheme message-passing procedures
   - Classes as Scheme "make-<object>" procedure
2. **Private State**: gives each object (each instance of a class) the ability to behave differently
   - Local environment
3. **Methods** for responding to messages
   - Scheme procedures (take method-dependent arguments)
4. An **Inheritance Rule** telling what method to use if no specific method is defined for a given message
   - Need to add conventions on messages & methods

Object-Oriented System - Version 1

```
(define (make-speaker name)
  (lambda (message)
    (cond ((eq? message 'NAME) (lambda () name))
          ((eq? message 'CHANGE-NAME)
           (lambda (new-name) (set! name new-name)))
          ((eq? message 'SAY)
           (lambda (list-of-stuff)
             (if (not (null? list-of-stuff))
                 (display-message list-of-stuff))
                 'NUF-SAID)))
          (else (no-method)))))
```

Or, with an alternative case syntax:

```
(define (make-speaker name)
  (lambda (message)
    (case message
      ((NAME) (lambda () name))
      ((CHANGE-NAME)
       (lambda (new-name) (set! name new-name)))
      ((SAY)
       (lambda (list-of-stuff)
         (if (not (null? list-of-stuff))
             (display-message list-of-stuff))
         'NUF-SAID))
      (else (no-method)))))
```

OO System - Version 1 Cont'd

Abstract out retrieval of method from the object (given the message)...

```
(define (get-method message object)
  (object message))
```

... and the combined retrieval and application of that method to the arguments:

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method args)
        (error "No method for message" message))))
```

Detection of methods (or missing methods):

```
(define (no-method) '(NO-METHOD))

(define (method? x)
  (cond ((procedure? x) #t)
        ((eq? x (no-method)) #f)
        (else (error "Object returned non-message" x))))
```

Example

```
(define p (make-speaker 'George))

(ask p 'NAME)
==> george

(ask p 'SAY '(I cannot tell a lie))
I cannot tell a lie
==> nuf-said
```

A Specialized Speaker (Subclass)

Want **lecturers** to be a kind of **speaker** - that **inherit** the behavior of speakers but add to that behavior:

```
(define (make-lecturer name)
  (let ((speaker (make-speaker name)))
    (lambda (message)
      (case message
        ((LECTURE)
         (lambda (self stuff)
           (delegate speaker self
                     'SAY '(Good Morning!))
           (delegate speaker self 'SAY stuff)))
        (else (get-method message speaker))))))

(define d (make-lecturer 'Duane))
(ask d 'LECTURE '(Today we learn more))
Good Morning!
Today we learn more
```

Approach: Inheritance by Delegation

- Inherit behavior by adding an "internal" speaker
  - Get internal object to act on behalf of object by **delegation**
- If message is not recognized, pass the buck
- Can change or specialize behavior:
  - Add new methods
  - Change operation of methods

Object-Oriented System - Version 2

```
(define (make-speaker name)
  (lambda (message)
    (case message
      ((NAME) (lambda (self) name))
      ((CHANGE-NAME)
       (lambda (self new-name)
         (set! name new-name)
         (ask self 'SAY (list 'call 'me name))))
      ((SAY)
       (lambda (self list-of-stuff)
         (if (not (null? list-of-stuff))
             (display-message list-of-stuff))
             'NUF-SAID))
      (else (no-method)))))

(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method object args)
        (error "No method for message" message))))

(ask p 'CHANGE-NAME 'fred)
Call me fred
```

Another Subclass

Want a "Canadian Lecturer" that changes the basic way of talking: append "Eh?" to everything he says...

```
(define (make-canadian-lecturer name)
  (let ((lecturer (make-lecturer name)))
    (lambda (message)
      (case message
        ((SAY)
         (lambda (self stuff)
           (delegate lecturer self
                     'SAY (append stuff '(Eh?)))))
        (else (get-method message lecturer))))))

(define (delegate to from message . args)
  (let ((method (get-method message to)))
    (if (method? method)
        (apply method from args)
        (error "No method" message))))

(define (ask object message . args)
  (apply delegate object object message args))

(define e (make-canadian-lecturer 'Eric))
(ask e 'SAY '(The sky is blue))
The sky is blue Eh?

(ask e 'LECTURE '(The sky is blue))
Good Morning!
The sky is blue
```

Fixing the Bug

```
(define (make-lecturer name)
  (let ((speaker (make-speaker name)))
    (lambda (message)
      (case message
        ((LECTURE)
         (lambda (self stuff)
           (delegate speaker self
                     'SAY '(Good Morning!))
           (delegate speaker self 'SAY stuff)
           (ask self 'SAY '(Good Morning!))
           (ask self 'SAY stuff)))
        (else (get-method message speaker))))))

(define e (make-canadian-lecturer 'Eric))
(ask e 'SAY '(The sky is blue))
The sky is blue Eh?

(ask e 'LECTURE '(The sky is blue))
Good Morning! Eh?
The sky is blue Eh?
```

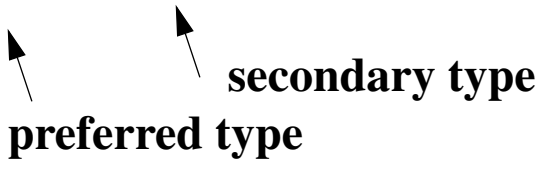
Multiple Inheritance

Can have objects that inherit methods from more than one type. Suppose rather than a named speaker we have an anonymous comic:

```
(define (make-comic)
  (lambda (message)
    (case message
      ((SAY)
       (lambda (self stuff)
         (display-message (append stuff '(ha ha)))))
      ((JOKE)
       (lambda (self)
         (display-message '(A duck walks into a bar))))
      (else (no-method)))))
```

Now we'll create a funny lecturer:

```
(define eric
  (let ((comic (make-comic))
        (lecturer (make-canadian-lecturer 'Eric)))
    (lambda (message)
      (get-method message lecturer comic))))
```




```
(ask eric 'JOKE)
A duck walks into a bar

(ask eric 'SAY '(The sky is blue))
The sky is blue Eh?
```

OO System - Version 3

```
(define (get-method message preferred . others)
  (define (loop objs)
    (let ((method (get-method-from-object
                    message (car objs)))
          (rest (cdr objs)))
      (if (or (method? method) (null? rest))
          method
          (loop rest))))
  (loop (cons preferred others)))

(define (get-method-from-object message object)
  (object message))
```

Alternative Multiple Inheritance

We have lots of flexibility - suppose we want to pass the message on to multiple internal objects (not just some "preferred" one)?

```
(define eric
  (let ((comic (make-comic))
        (lecturer (make-canadian-lecturer 'Eric)))
    (lambda (message)
      (lambda (self . args)
        (apply delegate-to-all
               (list lecturer comic)
               self
               args)))))

(ask eric 'SAY '(The sky is blue))
The sky is blue Eh?
The sky is blue ha ha
```

```
(define (delegate-to-all to-list from message . args)
  (foreach
    (lambda (to-whom)
      (apply delegate to-whom from message args))
  to-list)
```