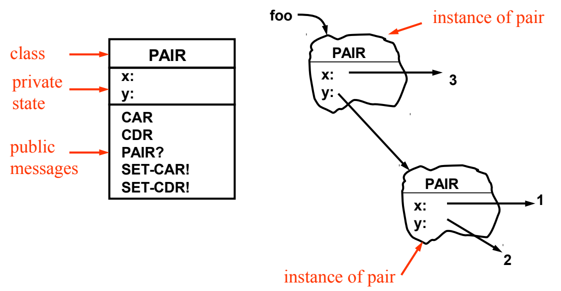
Object-Oriented Programming Terminology

- **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

Class Diagram   Instance Diagram




```
(define (cons x y)
  (λ (msg) ...))
```

```
(define foo
  (cons 3 (cons 1 2)))
```

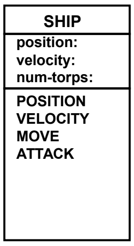

Using classes and instances to design a system

- Suppose we want to build a “star wars” simulator
- 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

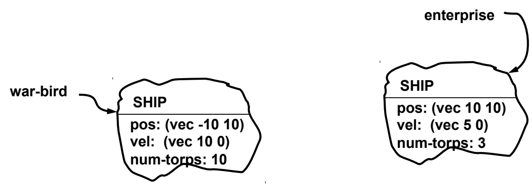
```
(define (make-ship position velocity num-torps)
  (define (move)
    (set! position (add-vect position ...)))
  (define (fire-torp)
    (cond ((> num-torps 0) ...)
          (else 'FAIL)))
  (lambda (msg)
    (cond ((eq? msg 'POSITION) position)
          ((eq? msg 'VELOCITY) velocity)
         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

Space-Ship Class



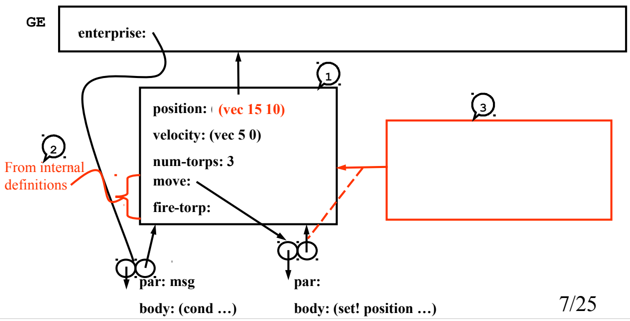
Example – Instance Diagram

```
(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

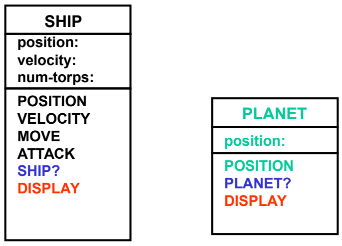
```
(define enterprise
  (make-ship (make-vect 10 10) (make-vect 5 0) 3))
(enterprise 'MOVE) ==> DONE
(enterprise 'POSITION) ==> (vect 15 10)
```



Some Extensions to 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)

Space-Ship Class



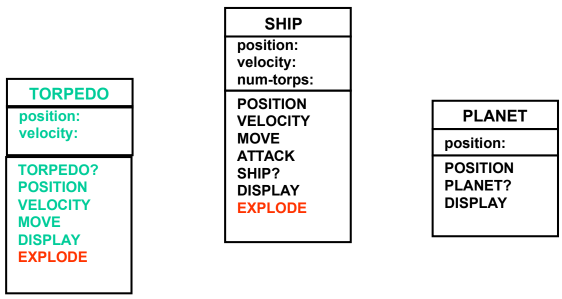
Planet Implementation

```
(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)))))
```

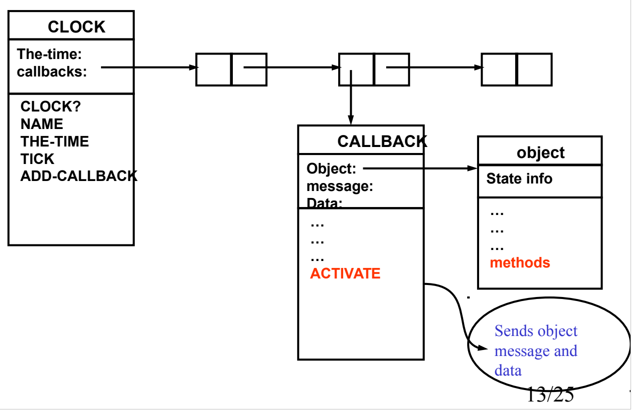

Further Extensions to our World

- 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



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))
```


Controlling the clock

```
;; Clock callbacks
;;
;; A callback is an object that stores a target object,
;; message, and arguments.  When activated, it sends the target
;; object the message.  It can be thought of as a button that
;; executes an action at every tick of the clock.
(define (make-clock-callback name object msg . data)
  (lambda (message)
      (case message
       ((CLOCK-CALLBACK?) (lambda (self) #t))
       ((NAME) (lambda (self) name))
       ((OBJECT) (lambda (self) object))
       ((MESSAGE) (lambda (self) msg))
       ((ACTIVATE) (lambda (self)
             (apply-method object object msg data)))
```


Implementations for our Extended World

```
(define (make-ship position velocity num-torps)
 (define (move) (set! position (add-vect position ...)))
 (define (fire-torp)
   (cond ((> num-torps 0)
          (set! num-torps (- num-torps 1))
          (let ((torp (make-torpedo ...))
            (add-to-universe torp))))
 (define (explode ship)
   (display "Ouch.  That hurt."))
 (ask clock 'ADD-CALLBACK
     (make-clock-callback 'moveit me 'MOVE))
 (define (me msg . args)
    (cond ((eq? msg 'SHIP?) #T)
         ...
         ((eq? msg 'ATTACK) (fire-torp))
         ((eq? msg 'EXPLODE) (explode (car args)))
         (else (error "ship can't" msg))))
  ME)
```

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))))
  ME)
```


Running the Simulation

```
;; Build some things
(define earth (make-planet (make-vect 0 0)))
(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))


;; Start simulation
(run-clock 100)
```


Summary

- 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 time: powerful ideas of *inheritance* and *delegation*

OOPS

- Using objects to structure systems
- Behaviors of object oriented systems
- Designing object oriented systems
  - Focus initially on conceptual plans
  - Eventually show a Scheme implementation

Elements of OOP

Example: bank accounts

Class: Behavior of accounts in general

Instances: My account versus yours

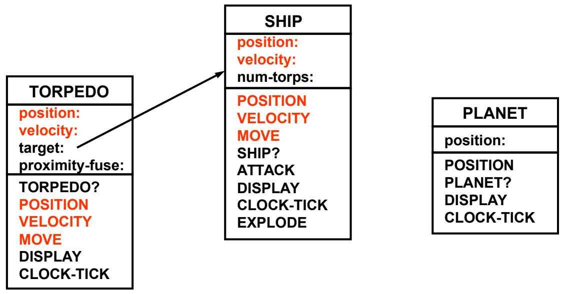
- **Object**
  - “Smart” data structure
    - Set of state variables
    - Set of methods for manipulating state variables
- **Class**:
  - Specifies the common behavior of entities

Focus here during design

- **Instance**:
  - A particular object or entity of a given class

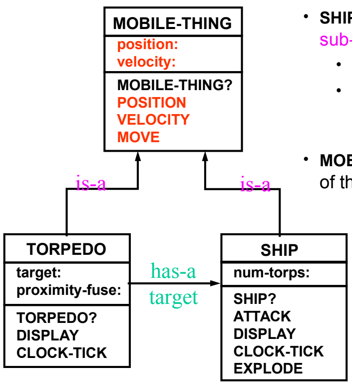
Focus here during simulation

Space War Class Diagram



- Ships and torpedoes have some behavior that is the **same** – is there are 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

How to design interactions between objects

- Focus on classes objects
  - Relationships between classes
  - Kinds of interactions that need to be supported between instances of classes
- For now, assume the following interface to an object:

```
(ask <object> <method> <arguments>)
```

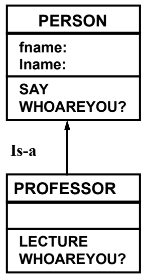
An initial class hierarchy


```
(define person-1 (make-person 'Fred 'Jones))

(ask person-1 'SAY '(hello there))
➔ hello there

(ask person-1 'WHOAREYOU?)
➔ Fred
```

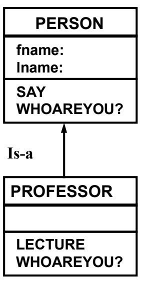

An initial class hierarchy




```
(define professor-1 (make-professor 'sam 'smith))

(ask professor-1 'say '(hello there))
➔ hello there
```


An initial class hierarchy

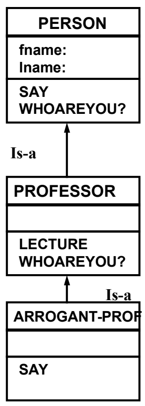



```
(ask professor-1 'WHOAREYOU?)
➔ Professor Smith

(ask professor-1 'LECTURE '(the sky is
blue))
➔ Therefore, the sky is blue
```

A professor should "delegate" part of the **lecture** method to a person's **say** method.

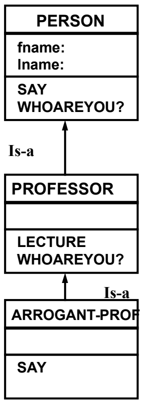
An initial class hierarchy




```
(define ap-1 (make-arrogantprof 'Ned 'Infallible))

(ask ap-1 'SAY '(nice weather we are having))
➔ nice weather we are having, obviously
```

An initial class hierarchy

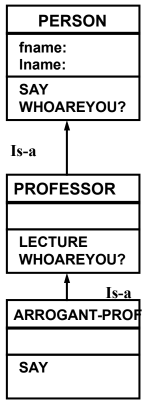



```
(define ap-1 (make-arrogantprof 'Ned 'Infallible))

(ask ap-1 'LECTURE '(nice weather we are having))
➔ Therefore, nice weather we are having
```

29/25

An initial class hierarchy

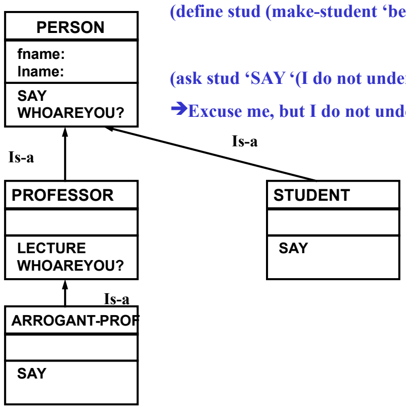



```
(define ap-1 (make-arrogantprof 'Ned 'Infallible))

(ask ap-1 'LECTURE '(nice weather we are having))
➔ Therefore, nice weather we are having, obviously
```

30/25

An initial class hierarchy



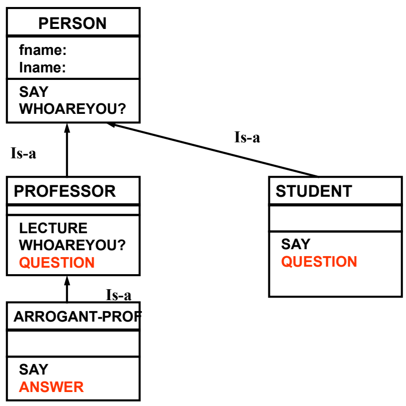
(define stud (make-student 'bert 'sesame))

(ask stud 'SAY '(I do not understand))

➔Excuse me, but I do not understand

31/25

An initial class hierarchy

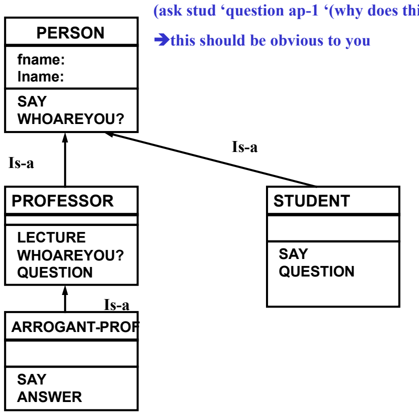


32/25

An initial class hierarchy

(ask stud 'question ap-1 '(why does this code work))

➔this should be obvious to you

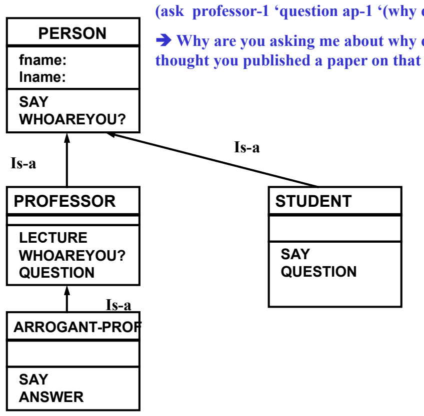


33/25

An initial class hierarchy

(ask professor-1 'question ap-1 '(why does this code work))

➔ Why are you asking me about why does this code work I thought you published a paper on that topic

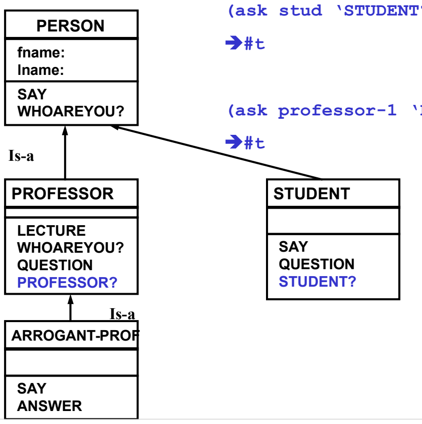


34/25

An initial class hierarchy

```
(ask stud 'STUDENT?)
➔#t

(ask professor-1 'PROFESSOR?)
➔#t
```



35/25

Lessons from our simple class hierarchy

- tagging of instances
- specifying class hierarchies and ensuring that instances creating superclass instances
- inheriting of methods from class hierarchies
- delegation of methods to other instances within a class hierarchy

36/25