Last Time

(define TOTAL 0)
(define WORKING 0)
(define PREV-OP +)
...
(define (add-digit n) ...)
...
(define (change-total n OP) ...)
...

The add-digit and change-total functions "remember" using TOTAL, WORKING, and PREV-OP

Organizational Chart, Effects, Templates

; add-digit : num -> true
; Adds a digit to the number being entered
; Effect: extends number, updates GUI
(define (add-digit n)
  ... n
  ... WORKING ... (set! WORKING ...) ...)

Organizational Chart, Effects, Templates

; change-total : num (num num -> num) -> true
; Combines number and total
; Effect: sets total, resets number, sets op, updates GUI
(define (change-total n OP)
  ... n ... OP
  ... WORKING ... (set! WORKING ...) ...)
  ... PREV-OP ... (set! PREV-OP ...) ...)
  ... TOTAL ... (set! TOTAL ...) ...)

Designing Functions with State

- New design tool: organizational charts
- Contract, Purpose, and Header becomes Contract, Purpose, Effect, and Header
- Examples include starting state and effect
- Template includes potential assignments
Examples

(begin
  (set! WORKING 0)
  (add-digit 5) "should be" true
WORKING "should be" 5)

(begin
  (set! WORKING 10)
  (add-digit 5) "should be" true
WORKING "should be" 105)

Examples

(begin
  (set! TOTAL 3)
  (set! WORKING 5)
  (set! PREV-OP *)
  (change-total 5 +) "should be" true
TOTAL "should be" 15
WORKING "should be" 0
PREV-OP "should be" +)

Simpler Example
Suppose we want a GUI to manage a fish

Run

New rule: keep the view and control separate from the model
- The view and control are in the GUI
- The model is a fish with a weight

Design the model first

Fish Model
- The only operation in the model is feed

; feed : num -> num
; Grows the fish by n, returns new size
; Effect: adjusts the fish’s weight
**Fish Model**

- The only operation in the model is `feed`:

  ```scheme
  (define (feed n)
    ... n ... WEIGHT
    ... (set! WEIGHT ...) ...)
  
  (begin
    (set! WEIGHT 1)
    (feed 10) "should be" 11
    WEIGHT "should be" 11)
  
  ; feed : num -> num
  ; Grows the fish by n, returns new size; Effect: adjusts the fish’s weight
  
  ; The only operation in the model is feed
  ; 
  ; feed : num -> num
  ; Grows the fish by n, returns new size; Effect: adjusts the fish’s weight
  
  (define (feed n)
    (begin
      (set! WEIGHT (+ WEIGHT n))
      WEIGHT))
  
  (begin
    (set! WEIGHT 1)
    (feed 10) "should be" 11
    WEIGHT "should be" 11)
  
  (define (feed n)
    (begin
      (set! WEIGHT (+ n WEIGHT))
      WEIGHT))
  
  (begin
    (set! WEIGHT 1)
    (feed 10) "should be" 11
    WEIGHT "should be" 11)

**Fish Model Implementation**

- The only operation in the model is `feed`:

  ```scheme
  (define (feed n)
    ... n ... WEIGHT
    ... (set! WEIGHT ...) ...)
  
  (begin
    (set! WEIGHT 1)
    (feed 10) "should be" 11
    WEIGHT "should be" 11)
  
  ; feed : num -> num
  ; Grows the fish by n, returns new size; Effect: adjusts the fish’s weight
  
  (define (feed n)
    (begin
      (set! WEIGHT (+ n WEIGHT))
      WEIGHT))
  
  (begin
    (set! WEIGHT 1)
    (feed 10) "should be" 11
    WEIGHT "should be" 11)

**Implementing the View and Controller**

- Use the GUI teachpack to construct view and control
- Message objects implement the view
- Button callbacks implement the control

**Complete Fish Program**

- The model:

  ```scheme
  (define WEIGHT 3)
  ; feed : num -> num
  ; ... test here ...
  (define (feed n)
    (begin
      (set! WEIGHT (+ n WEIGHT))
      WEIGHT))
  
  (define (feed n)
    (begin
      (set! WEIGHT (+ WEIGHT n))
      WEIGHT))
  
  (begin
    (set! WEIGHT 1)
    (feed 10) "should be" 11
    WEIGHT "should be" 11)

  ; The model:
  (define WEIGHT 3)
  ; feed : num -> num
  ; ... test here ...
  (define (feed n)
    (begin
      (set! WEIGHT (+ n WEIGHT))
      WEIGHT))
  
  ; The control:
  (define (feed-button n)
    (make-button (string-append "Feed " (number->string n))
      (lambda (evt)
        (draw-message
          msg
          (number->string (feed n))))))
  
  (create-window
    (list (list msg) (list (feed-button 1) (feed-button 3))))

  ; The model:
  (define WEIGHT 3)
  ; feed : num -> num
  ; ... test here ...
  (define (feed n)
    (begin
      (set! WEIGHT (+ n WEIGHT))
      WEIGHT))
  
  ; The control:
  (define (feed-button n)
    (make-button (string-append "Feed " (number->string n))
      (lambda (evt)
        (draw-message
          msg
          (number->string (feed n))))))
  
  (create-window
    (list (list msg) (list (feed-button 1) (feed-button 3))))
Multiple Fish

As we saw last time, if we want multiple fish, we can use `local`:

```scheme
(define (make-fish init-weight)
  local [(define WEIGHT init-weight)
    (define (feed n)
      (begin
        (set! WEIGHT (+ WEIGHT n))
        WEIGHT))
    ...
  ]
  (create-window ...)))
```

Evaluating `make-fish`

```scheme
(define (make-fish init-weight)
  local [(define WEIGHT init-weight)
    (define (feed n)
      (begin
        (set! WEIGHT (+ WEIGHT n))
        WEIGHT))
    ...
  ]
  (create-window ...)))
```

Every time we call `make-fish` a new `WEIGHT` is created for the new fish.

We can make a whole aquarium....

- How can we get the current total weight of all fish?
- How can we auto-feed all fish?

**Problem:** `make-fish` returns only a window

The renamed `WEIGHT` is completely hidden.
Returning the Weight

Does this help?

; make-fish : num -> num
(define (make-fish init-weight)
  (local [(define WEIGHT init-weight)
    ...
    ]
    (begin
      (create-window ...)
      WEIGHT)))

No:

(make-fish 5)
→ (local [(define WEIGHT 5) ...] ... WEIGHT)
→ (define WEIGHT, 5) ... WEIGHT,
→ → (define WEIGHT, 5) ... 5

Returning the Feeder

Only functions inside make-fish can see WEIGHT

So maybe make-fish should return a function:

; make-fish : num -> (num -> num)
(define (make-fish init-weight)
  (local [(define WEIGHT init-weight)
    (define (feed n) ...
      WEIGHT ...
      )
    ]
    (begin
      (create-window ...)
      feed)))

(make-fish 5)
→ (local [(define WEIGHT 5) (define (feed n) ... WEIGHT ...) ...
  ...
  feed)
→ (define WEIGHT, 5) (define (feed, n) ... WEIGHT, ...) ... feed,

Feeding an Aquarium

; A live-fish is
;   (num -> num)

; make-fish : num -> live-fish
...
(define aquarium (list (make-fish 5)
  (make-fish 3)
  (make-fish 12)))

; aq-weight : list-of-live-fish -> num
(define (aq-weight l)
  (foldr (lambda (f r) (+ (f 0) r)) 0 l))

; feed-all : n list-of-live-fish -> ...
(define (feed-all n l)
  (map (lambda (f) (f n)) l))

for-each

The built-in function for-each is like map, but it returns (void)

; feed-all! : n list-of-live-fish -> (void)
; Feeds n to each live-fish in l
; Effect: each live-fish becomes heavier
(define (feed-all! n l)
  (for-each (lambda (f) (f n)) l))
The built-in function `for-each` is like `map`, but it returns `(void)`

```scheme
(define (feed-all! n l)
  (for-each (lambda (f) (f n)) l))
(begin
  (define l (list (make-fish 1) (make-fish 2)))
  (feed-all! 3 l) "should be" (void)
  l "should be" (list (make-fish 4) (make-fish 5)))
```

- Testing with state is often difficult
- Avoid this difficulty by avoiding state whenever possible
A Tale of Two Fish Representations

; A fish is
; num

; A live-fish is
; (num -> num)

- **A fish** represents a fish of a particular weight
  - Feed the fish \( \Rightarrow \) new value
- **A live-fish** represents a fish with a particular identity
  - Feed the fish \( \Rightarrow \) same value, new state

live-fish is more closely reflects reality
- On the one hand, reflecting reality makes things more intuitive
- On the other hand, reality can be messy

Key question when designing a program: what to represent

Encapsulation

Packaging fish state with its operations is called **encapsulation**

More on encapsulation soon...

Design with State Summary

- Deciding to use state: often motivated by GUIs
  - Split into model and view/controller
- The design recipe for state
  - Charts (no handin artifact)
  - Effects (handin with purpose)
  - Template with assignments (handin optional)
  - Multi-step tests (handin as usual)
- Design for the single-instance case, then encapsulate if necessary