Stateful Objects and Stable Identities

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 10.2
Key Points for Lesson 11.2

• Sometimes objects need to ask questions of each other over time.
• To accomplish this, the object being queried needs to have a stable identity that the querier can rely on.
• In this lesson, we'll show what can happen when this fails.
Sometimes making a new object doesn't do what's needed

• We now begin a sequence of programs illustrating patterns of object communication.
• These programs will involve a ball bouncing on a canvas
• What’s interesting, though, is that the canvas has an draggable wall, so the ball needs to find out about the position of the wall at every tick.
Let's look at some code: 10-2A-ball-and-wall.rkt

;; The World implements the WorldState<%> interface
(define WorldState<%>
 (interface ()
  ; -> WorldState
  ; GIVEN: no arguments
  ; RETURNS: the state of the world at the next tick
  after-tick

  ; Integer Integer MouseEvent-> WorldState
  ; GIVEN: a location
  ; RETURNS: the state of the world that should follow the
  ; given mouse event at the given location.
  after-mouse-event

  ; KeyEvent -> WorldState
  ; GIVEN: a key event
  ; RETURNS: the state of the world that should follow the
  ; given key event
  after-key-event

  ; -> Scene
  ; GIVEN: a scene
  ; RETURNS: a scene that depicts this World
  to-scene
 ))

;; Every object that lives in the world must implement the Widget<%> interface.
(define Widget<%>
 (interface ()
  ; -> Widget
  ; GIVEN: no arguments
  ; RETURNS: the state of this object that should follow the next tick
  after-tick

  ; Integer Integer -> Widget
  ; GIVEN: a location
  ; RETURNS: the state of this object that should follow the
  ; specified mouse event at the given location.
  after-button-down after-button-up after-drag

  ; KeyEvent -> Widget
  ; GIVEN: a key event and a time
  ; RETURNS: the state of this object that should follow the
  ; given key event
  after-key-event

  ; Scene -> Scene
  ; GIVEN: a scene
  ; RETURNS: a scene like the given one, but with this object
  ; painted on it.
  add-to-scene
 )))

WorldState<%> and Widget<%> interfaces as before
(define Wall<%> (interface (Widget<%>))

; -> Int
; RETURNS: the x-position of the wall
get-pos

))

This means that the Wall<%> interface includes all the methods from the Widget<%> interface. This is called "interface inheritance."

The wall will have an extra method that returns the current position of the wall. This information is needed by the ball.
The Ball% class

;;; A Ball is a (new Ball%;
;;; [x Int][y Int][speed Int][w Wall])

(define Ball%
  (class* object% (Widget<%>)
    (init-field w) ;; the Wall ...
    ;; after-tick : -> Ball
    ;; RETURNS: state of this ball
    (define/public (after-tick)
      (if selected? this
        (new Ball%
          [x (next-x-pos)]
          [y y]
          [speed (next-speed)]
          [selected? selected?]
          [saved-mx saved-mx]
          [saved-my saved-my]
          [w w])
      )))

;; -> Integer
;; position of the ball at the next
;; tick.
;; STRATEGY: ask the wall for its
;; position and use that to
;; calculate the upper bound for
;; the ball's x position
(define (next-x-pos)
  (limit-value
    radius
    (+ x speed)
    (- (send w get-pos) radius)))

;; Number^3 -> Number
;; WHERE: lo <= hi
;; RETURNS: val, but limited to the
;; range [lo,hi]
(define (limit-value lo val hi)
  (max lo (min val hi)))

At every tick, the ball asks w about its position
The Wall% class

;;; A Wall is (new Wall% [pos Integer] 
;;; [saved-mx Integer] 
;;; [selected? Boolean]) 
;;; all these fields have default values.

(define Wall% 
  (class* object% (Wall<%>) 
    
    ;; the x position of the wall 
    (init-field [pos INITIAL-WALL-POSITION]) 
    ;; is the wall selected? Default is false. 
    (init-field [selected? false]) 
    
    ;; if the wall is selected, the x position of 
    ;; the last button-down event near the wall, 
    ;; relative to the wall position 
    (init-field [saved-mx 0]) 
  
  (super-new) 

  ;; the extra behavior for Wall<%> 
  (define/public (get-pos) pos) 

  ;; after-button-down : Integer Integer -> Wall 
  ;; GIVEN: the location of a button-down event 
  ;; STRATEGY: Cases on whether the event is near 
  ;; the wall 
  ;; RETURNS: A wall like this one, but selected, and 
  ;; with mouse x location (relative to the wall 
  ;; position) recorded 
  (define/public (after-button-down mx my) 
    (if (near-wall? mx) 
      (new Wall% 
        [pos pos] 
        [selected? true] 
        [saved-mx (- mx pos)]) 
      this)) 

  ;; after-drag : Integer Integer -> Wall 
  ;; GIVEN: the location of a drag event 
  ;; STRATEGY: Cases on whether the wall is selected. 
  ;; If it is selected, returns a wall like this one, 
  ;; except that 
  ;; the vector from its position to 
  ;; the drag event is equal to saved-mx 
  (define/public (after-drag mx my) 
    (if selected? 
      (new Wall% 
        [pos (- mx saved-mx)] 
        [selected? true] 
        [saved-mx saved-mx]) 
      this)) 

The code for the Wall% class is perfectly routine
Here's a demo

If you have difficulty with this video, look at it on YouTube, or just run 10-2A-ball-and-wall.rkt.
What went wrong?

• After a drag, however, the world has a new wall at the new position.
• But the ball still points at the original wall, in the original position.
• So the ball bounces at the position where the wall used to be.
We need to make the wall stateful

• We need to give the wall a stable identity, so balls will know who to ask.
• But the information in the wall must change!
• Solution: we need to make the box MUTABLE.
• In other words, it should have state.
• What does that mean? How do we do this? That is the topic of the next two lessons.
Next Steps

• Study 10-2A-ball-and-wall.rkt in the Examples folder.

• In the next lesson, we'll consider the difference between real state and simulated state in a little more detail.

• Then we'll consider how to program systems with state in our framework.