Generalizing this Design

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 10.5

Problems with our design so far

- It used to be that WorldState% created new widgets by trapping keystrokes in its after-key-event method.
- But what if we want to add new objects by some other means (e.g. pushing a button on the screen)?
- And having widget creation handled by the World means the World has to know both about distributing messages AND about keystrokes
  - that's a violation of one task per function.

Solving this problem

- We'll do this in three steps:
  1. We'll make the world stateful, too
  2. We'll give it methods for adding widgets and stateful widgets.
  3. Then we'll create a ball factory to create balls and add them to the world.
     - the factory will know about the wall, so the balls it creates will be equipped with knowledge about the wall.

StatefulWorld<%>

```
;; The World implements the StatefulWorld<%> interface

(define StatefulWorld<%>
  (interface ()

    ; -> Void
    ; GIVEN: no arguments
    ; EFFECT: updates this world to its state after a tick
    after-tick

    ; Integer Integer MouseEvent-> Void
    ; GIVEN: a location
    ; EFFECT: updates this world to the state it should be in
    ;  following the given mouse event at the given location.
    after-mouse-event

    ; KeyEvent -> Void
    ; GIVEN: a key event
    ; EFFECT: updates this world to the state it should be in
    ;  following the given key event
    after-key-event

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

    ))
```

This is just like what we did before: We change the contracts to return Void, and replace RETURNS with EFFECT in the purpose statements.

World%

```
; ListOfWidget ListOfSWidget -> World
(define (make-world-state objs sobjs)
  (new World% [objs objs][sobjs sobjs]))

(define World%
  (class* object% (StatefulWorld<%>)

    (init-field objs)  ; ListOfWidget
    (init-field sobjs)  ; ListOfSWidget

    (super-new)

    ;; after-tick : -> Void
    ;; Use map on the Widgets in this World; use for-each on the
    ;; stateful widgets

    (define/public (after-tick)
      (begin
        (for-each
          (lambda (obj) (send obj after-tick))
          sobjs)
        (set! objs
          (map
            (lambda (obj) (send obj after-tick))
            objs))))
```

In keeping with Lesson 10.3, I've changed the name of this class of WorldState% to World%, since it models an actual world, not merely the mathematical value that is its state.

We replace each call to make-world-state or new with a suitable set!, just as in the preceding lesson.

We need to modify our call to big-bang

```
; run : PosReal -> World
; GIVEN: a frame rate, in secs/tick
; EFFECT: runs an initial world at the given frame rate
; RETURNS: the world in its final state
(define (run rate)
  (big-bang (initial-world)
    (on-tick
      (lambda (w) (begin (send w after-tick) w))
      rate)
    (on-draw
      (lambda (w) (send w to-scene)))
    (on-key
      (lambda (w kev)
        (begin
          (send w after-key-event kev)
          w)))
    (on-mouse
      (lambda (w mx my mev)
        (begin
          (send w after-mouse-event mx my mev)
          w)))))
```

The methods of the world used to return a new world, but not any more. **Big-bang** still expects its handlers to return a world, so we do this explicitly by writing **(begin (send w ...) w)**

We still initialize the world in the same way

```
;; initial-world : -> World
;; RETURNS: a world with a wall and a ball that knows about
;;  the wall.
(define (initial-world)
  (local
    ((define the-wall (new Wall%))
     (define the-ball (new Ball% [w the-wall])))
    (make-world-state
     (list the-ball)
     (list the-wall))))
```

Now let's add a method to add new widgets to the world

First we add it to the interface **World<%>** :

```
; Widget -> Void
; GIVEN: A widget
; EFFECT: adds the given widget to the world
add-widget

; SWidget -> Void
; GIVEN: A stateful widget
; EFFECT: adds the given widget to the world
add-stateful-widget
```

And the method definitions:

```
(define/public (add-widget w)
  (set! objs (cons w objs)))

(define/public (add-stateful-widget w)
  (set! sobjs (cons w sobjs)))
```

Now we can build a ball factory

```
;; The BallFactory% class

;; accepts "b" key events and adds them to the world.
;; gets the world as an init-field

(define BallFactory%
  (class* object% (SWidget<%>)

    (init-field world)  ; the world to which the factory adds balls
    (init-field wall)   ; the wall that the new balls should bounce
                        ; off of.

    (super-new)

    (define/public (after-key-event kev)
      (cond
        [(key=? kev "b")
         (send world add-widget (new Ball% [w wall]))]))

    ;; the Ball Factory has no other behavior. Return nonsense values for Void,
    ;; to aid in debugging.

    (define/public (after-tick) 15)
    (define/public (after-button-down mx my) 16)
    (define/public (after-button-up mx my) 17)
    (define/public (after-drag mx my) 18)
    (define/public (add-to-scene s) s)

    ))
```

The factory receives key events from the world. On each "b", it creates a new ball, and then passes it to the world as an argument to **add-widget**.

And let's initialize the system

```
;; initial-world : -> WorldState
;; RETURNS: a world with a wall, a ball, and a factory
(define (initial-world)
  (local
    ((define the-wall (new Wall%))
     (define the-ball (new Ball% [w the-wall]))
     (define the-world
       (make-world-state (list the-ball) (list the-wall)))
     (define the-factory
       (new BallFactory% [wall the-wall][world the-world])))
    (begin
      (send the-world add-stateful-widget the-factory)
      the-world)))
```

1. Create a wall
2. Create a ball that knows about the ball
3. Create a world with the ball and the wall
4. Create a factory that knows about the wall and the world
5. Add the factory to the world
6. Return the resulting world

We just created a cyclic structure!

- Notice: the factory needed to know about the world, and the world needed to know about the factory.
- This is a *cyclic* structure.
- You can't build a cyclic structure without state.

Wasn't that fun?

- Go play with 10-4-ball-factory.rkt

Key Points for Lesson 10.5

- We applied the iterative design strategy in an object-oriented system
- At every step, we first designed the interface, so we'd know what our methods were supposed to do.
- Then we designed the methods.
- We needed a cyclic structure, so both the world and the factory needed to be stateful.

Next Steps

- Study 10-4-ball-factory.rkt in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson.