Testing Mutable Objects

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 11.5
Key Points for Lesson 11.5

• State makes testing harder.
• To test a stateful system, create scenarios that create objects, send them sequences of messages, and then checks the observable outputs.
• Good OO designs use as little state as possible.
State makes testing harder

• You have to get things into the state you want
• Then observe the relevant portions of the final state (at just the right time!)
• May want to test a sequence of states
• In real world, may have to do tear-down to prepare for next test.
  – Living in a mostly-functional world makes this unnecessary for us.
Setting up a test scenario

(begin-for-test
  create objects for the test
  check to see that objects are initialized correctly
  (send obj1 method1 arg1 ...)
  check to see that objects have the right properties
  ...continue through sequence of events...
)

Use getter methods if necessary

Here is a skeleton for setting up tests for imperative objects in rackunit and "extras.rkt".
We can use getter methods to pull out the relevant properties of the objects. Remember that these should only be used for testing. Using getter methods on non-observables as part of your computation is considered bad OO design and should be avoided (see Lesson 11.1)
A Simple Test Case

(begin-for-test
  (local
    ((define box1
        (new Box%
            [x 200][y 50][w 100][h 20][selected? true])))
     (check-equal? (send box1 left-edge) 150)
     (check-equal? (send box1 right-edge) 250)
     (send box1 on-mouse 252 50 "drag")
     (check-equal? (send box1 left-edge) 150)
     (check-equal? (send box1 right-edge) 252))
  )
Using Generalization

• Use generalization to avoid repeated code, even in your tests!
• Introduce functions to generalize repeated patterns of code in your tests.
• For these, we won’t require examples, tests, etc.
  – In the real world, these might be as complicated as your real code, and might need to be tested themselves.
Help Functions

;; World% (list Integer Integer) [String] -> Check.
;; RETURNS: a check that checks whether the object
;; and stateful-object lists in the given world have
;; the given lengths
;; str is optional error message, default is empty
;; string
;; Example:
;; (check-lengths init-world (list 0 3)
;;  "initial world has incorrect object lists")
(define (check-lengths w lst [str ""])
  (check-equal?
   (map length (send w for-test:get-all-objects))
   lst
   str))

Here we use a tester method to get the object lists.
Another help function that came in handy

;;; Box<%> -> (list Number Number)
(define (get-edges box)
  (list
   (send box left-edge)
   (send box right-edge))))
A slightly more elaborate test

(local
  ;; create a box and check its edges
  ((define box1
      (new Box% [x 200][y 50][w 100][h 20][selected? false])))
  (check-equal? (get-edges box1)
    (list 150 250)
    "edges should be initialized correctly")
  ;; send the box a button-down
  (send box1 on-mouse 252 50 "button-down")
  ;; send the box a drag message
  (send box1 on-mouse 252 50 "drag")
  ;; now the edges of the box should have changed
  (check-equal? (get-edges box1)
    (list 150 252)
    "left edge should stay the same, right edge should change")
And another...

(local
  ;; first create the objects
  ((define the-box
      (new Box% [x 100][y 45][w 100][h 75][selected? false]))
  ;; right edge of box is at 150. So put the ball close to
  ;; the edge.
  ;; center at edge - speed - radius + 1, so ball should
  ;; bounce on next tick
  ;; x = 150 - 10 - 15 + 1 = 126
  (define the-ball
    (new Ball% [x 126][y 45][box the-box][speed 10]))
  (define ball-after-tick (send the-ball on-tick)))
  ;; check to see that the speed is now -10
  (check-equal?
    (send ball-after-tick for-test:get-speed)
    -10
    "after bounce, ball speed should be -10")
)
; ; "check that balls are added correctly"
(local
; ; first create a box, a ball in that box, and a world containing
; ; the ball and the box
((define the-box (new Box% [x 100][y 45][w 150][h 75][selected? false]))
 (define the-ball (new Ball% [x 100][y 45][box the-box][speed 5]))
 (define the-world
 (new World%
 [objects (list the-ball)]
 [stateful-objects (list the-box)]))
 (check-equal?
 (map length (send the-world for-test:get-all-objects))
 '(1 1)
 "check initial lengths of object lists")
; ; add a ball directly to the world
 (send the-world add-object
 (new Ball% [x 100][y 45][box the-box][speed 5]))
 (check-equal?
 (map length (send the-world for-test:get-all-objects))
 '(2 1)
 "check adding a ball to the world")
.. And on for another 20 or so lines...

This was a complicated test sequence I made up when I was trying to figure out why balls were not being added correctly. I tried adding balls directly to the container, then adding them through more and more layers until I eventually found the layer that wasn’t doing what it was supposed to. This is just like the detective work we talked about in Lesson 2.3
General Design Principle

• Use as little state as you can.
• Pass values whenever you can.
Java Guru on State:

Keep the state space of each object as simple as possible. If an object is immutable, it can be in only one state, and you win big. You never have to worry about what state the object is in, and you can share it freely, with no need for synchronization. If you can't make an object immutable, at least minimize the amount of mutation that is possible. This makes it easier to use the object correctly.

As an extreme example of what not to do, consider the case of java.util.Calendar. Very few people understand its state-space -- I certainly don't -- and it's been a constant source of bugs for years.

-- Joshua Bloch, Chief Java Architect, Google; author, Effective Java

Here's a quotation on state from a famous Java programmer.
Summary

• We've studied the difference between a value (usually data) and a state (usually information)
• State enables objects to share information with objects that it doesn't know about.
• State makes testing and reasoning about your program harder.
• Use as little state as you can.
• Pass values whenever you can.