Testing Objects

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 9.4
Key Points of this lesson

• We can only test observable behavior
  – observables are not fields

• We need to decide what properties are important, and test those.
  – we can’t use equal?
An example

- Let’s consider a really simple interface:
StupidRobot<>

;; A StupidRobot represents a robot moving along a one-dimensional line, ;; starting at position 0.

(define StupidRobot<>
  (interface ()
    ;; a new StupidRobot<> is required to start at position 0

    ;; -> StupidRobot<>
    ;; RETURNS: a Robot just like this one, except moved one
    ;; position to the right
    move-right

    ;; -> Integer
    ;; RETURNS: the current x-position of this robot
    get-pos
  ))

The only observable is the position of the robot.
Scenario and Observation

If we have a correct implementation of `StupidRobot<%>`, the following test should pass:

```scheme
(local
  ((define r0 ..a new StupidRobot<%>..)
   ;; move r0 right twice
   (define r1 (send (send r0 move-right) move-right)))
  ;; get-pos should then return 2
  (check-equal
   (send r1 get-pos)
   2))
```
The "obvious" implementation

(define Robot1%
  (class* object% (StupidRobot<%>))

  (init-field [x 0])
  ;; interp: the position of the robot.

  (super-new)

  (define/public (move-right)
    (new Robot1% [x (+ x 1)]))

  (define/public (get-pos)
    x)

))

Here the observable is the value of a field.
Of course, our choice of x for the field name is arbitrary.
You could name fields anything you want

(define Robot2%
 (class* object% (StupidRobot<%>))

  (init-field [blerch 0])
  ;; interp: the position of the robot.

  (super-new)

  (define/public (move-right)
    (new Robot2% [blerch (+ blerch 1)]))

  (define/public (get-pos)
    blerch)

))
But we could have done it differently

```
(define Robot3%
  (class* object% (StupidRobot<%>)
    (init-field [y 0])
    ;; interp: the negative of the position of the robot.

    (super-new)

    (define/public (move-right)
      (new Robot3% [y (- y 1)]))

    ;; RETURNS: the x-position of the robot
    (define/public (get-pos)
      (- y))
))
```

Here the observable is *not* the value of any field. The observation method translates the field value into the external value of the observable.
Or we could have done it very differently

(define Robot4%
  (class* object% (StupidRobot<%>))

  (init-field [x empty])
  ;; Interp:
  ;; a list whose length is equal to the position of the robot
  (super-new)

  (define/public (move-right)
    (new Robot4% [x (cons 99 x)]))

  ;; RETURNS: the x-position of the robot
  (define/public (get-pos)
    (length x))
)

Puzzle: the other two implementations would work fine if we had a move-left method as well as move-right. How could you modify this implementation to handle move-left?
All three of these implementations have the SAME observable behavior

- no combination of scenarios and observations can tell them apart!
- If these are the only methods and observations we have on these objects, then we don't care which implementation we use—they will behave the same in any program.
- We could even write something like
Choose a random implementation

;; -> StupidRobot<%>
(define (new-robot)
  (local
    ((define i (random 3)))
    (cond
      [(= i 0) (new Robot1%)]
      [(= i 1) (new Robot2%)]
      [(= i 2) (new Robot3%)]
      [(= i 3) (new Robot4%)]))

Returns a random number between 0 and 3
Contracts and Interfaces (again)

;; move-right-by-distance
;; : Robot<%> Nat -> Robot<%>
(define (move-right-by-distance r n)
  (cond
   [(zero? n) r]
   [else (move-right-by-distance
           (send r move-right)
                   (- n 1))]))

This works with ANY class that implements Robot<%>.

Contracts should be in terms of interfaces, not classes.
We need to change the way we write tests

- Our tests should work with any implementation of StupidRobot<%>.
- We construct a scenario, in which we create some objects, send them some messages, and see what the observables are.
- The tests talk only through the interface.
(begin-for-test
  (local
    ((define r0 (new-robot))
     ;; move r0 right twice
     (define r1 (send (send r0 move-right) move-right)))
    ;; get-pos should then return 2
    (check-equal?
      (send r1 get-pos)
      2)))

(begin-for-test
  (local
    ((define r0 (new-robot))
     (define r1 (move-right-by-distance r0 3)))
    (check-equal?
      (send r1 get-pos)
      3)))
Observables in the problem sets

• In our problem sets, we've required you to provide just enough observables so that our automated testing routines can see if you've solved the problem.

• In a test, we create a scenario and then check the observables of the final state.

• The set of observables in the problem set is purposely minimal, in order to give you the maximum freedom in implementing the objects.
You may need to add some observables for debugging

- The set of observer methods in the problem sets is purposely minimal, in order to give you the maximum freedom in implementing the objects.
- You may need to add some observation methods for your own testing and debugging, so you can see what is going on inside your objects.
- That's ok, but give them names like `for-test:whatever` and do NOT use them for any other purpose.
Example of a scenario using a test method

(define w1 (make-world 5))
(define w2 (send w1 on-key "n"))
(define w3 (send w2 on-key "n"))
...
(check-equal?
  (length
   (send w3 for-test:rectangles))
  2
  "After 2 'n's, there should be two rectangles")
You can’t use `equal?` on objects

- In Racket, `equal?` on objects measures whether its arguments are the same object.
- In Racket, you can have two different objects with exactly the same values in the fields.
- We say that objects have *identity*.
- This will make more sense next week.
- In the meantime, here’s an example:
Example of object identity

(let ((r1 (new-robot)))
  (equal? r1 r1)) ➞ true

(let ((r1 (new-robot)))
  (let ((r2 r1))
    (equal? r1 r2))) ➞ true

(let ((r1 (new Robot1%))
  (r2 (new Robot1%)))
  (equal? r1 r2)) ➞ false
Luckily, most of the time we can avoid this.

• Usually, we’re not interested in whether we have the same object.
• We just care that our new object has the right observable properties.
Key Points of this lesson

• We can only test observable behavior
  – observables are not fields

• We need to decide what properties are important, and test those.
  – problem set will specify the interfaces that our tests rely on.
  – we can’t use `equal`?
  – test by creating scenarios and checking whether your objects have the right properties afterwards.
  – ok to add `for-test:` methods, but only for debugging.
Next Steps

• Study example 09-6-testing.rkt in the Examples folder.
• If you have questions about this lesson, ask them on the Discussion Board
• Go on to the next lesson