Patterns of Interaction 2: Publish-Subscribe

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 10.6

Key Points for Lesson 10.6

- Publish-Subscribe is a programming pattern for implementing push-style communication between objects over time.
- In pub-sub, a publisher keeps a list of subscribers.
- When the publisher changes state, it sends a message notifying each of its subscribers about the state change.
- Each subscriber changes its local state to take note of the messages it receives from the publisher.
- Now, the subscriber can consult its local state instead of sending queries to the publisher.
- Good if queries are much more frequent than state changes.

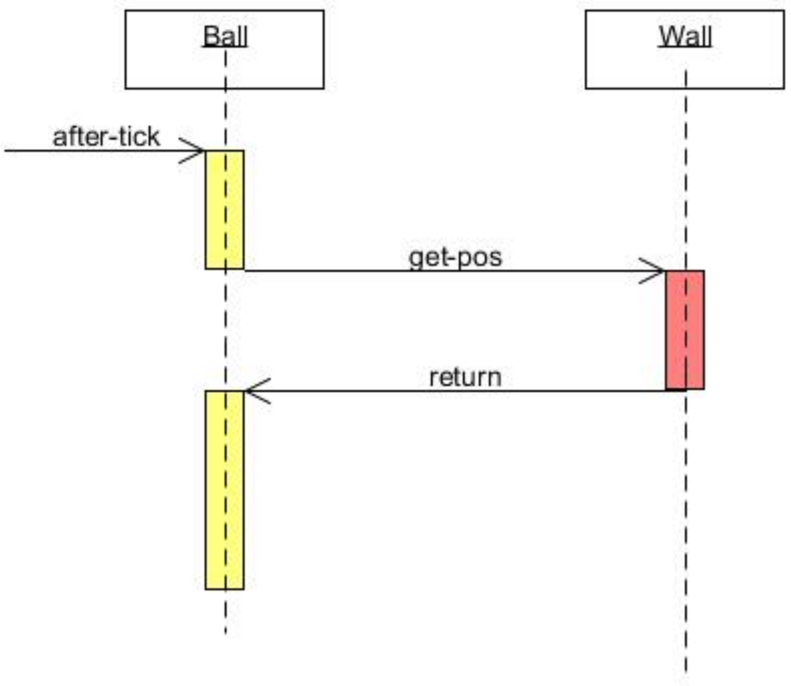
How to organize collaborating objects?

- Principle of Least Knowledge:
- Reveal only what's necessary.

First rule of good OO Design!

- Problem: how does the information get to where it's needed?
- We've already talked about this a little in Lesson 10.1
- What happens in a stateful system?

How does a ball decide when to bounce in ball-factory.rkt?



In ball-factory.rkt, every time the ball receives an **on-tick** message, it asks its wall for the location of its right edge. This is a pull model.

Diagrams like this are called *sequence diagrams* in UML.

Ball *pulls* info from the wall

Can we do better?

- Each ball asks the wall about its position at every tick.
- But this information doesn't change very often.
- Better idea: Have the wall send a "changed-edge" message to the balls only when the edge actually changes.

This is a *push* model

- When information changes, the person who changes it pushes it out to the people who need to know.
- How does the information-changer know who to tell?
  - The information-needer must *register* with the information-changer.

Push model, cont'd

- So each ball must tell the wall that it needs to hear about changes in the edge position.
- This means that the balls will now need to be stateful, too, so the wall can find them.
- This pattern is called *publish/subscribe*
  - also called the *observer* pattern.

Updated interfaces:

```
;; Additional method for Ball:

(define SBall<%>
  (interface (SWidget<%>)

    ; Int -> Void
    ; EFFECT: updates the ball's cached value of the wall's position
    update-wall-pos

    ))

;; Additional method for Wall:

(define SWall<%>
  (interface (SWidget<%>)

    ; SBall<%> -> Int
    ; GIVEN: An SBall<%>
    ; EFFECT: registers the ball to receive position updates from this wall.
    ; RETURNS: the x-position of the wall
    register

    ))
```

We use the prefix "S" for "stateful" or "stable". So SBall<%> is the interface for stateful balls.

We say the ball contains a *cache* of the wall's position. This is analogous to a memory cache. If you are not familiar with the idea of a cache, you should go look it up. It's a neat and widely-used pattern.

Add code to Ball%

```
(define Ball%
  (class* object% (SWidget<%>)

    (init-field w)  ;; the Wall that the ball should bounce off of

    ;; initial values of x, y (center of ball)
    (init-field [x INIT-BALL-X])
    (init-field [y INIT-BALL-Y])
    (init-field [speed INIT-BALL-SPEED])

    ...

    ;; register this ball with the wall, and use the result as the
    ;; initial value of wall-pos
    (field [wall-pos (send w register this)])

    (super-new)

    ;; update-wall-pos : Int -> Void
    ;; EFFECT: updates the ball's idea of the wall's position to the
    ;; given integer.
    (define/public (update-wall-pos n)
      (set! wall-pos n))
```

When the ball is initialized, it registers with the wall. The wall responds with its current position. This means we can add balls even after the wall has been moved.

Add code to Wall%

```
(define Wall%
  (class* object% (SWall<%>)

    (init-field [pos INITIAL-WALL-POSITION]) ; the x position of the wall

    ...

    (field [balls empty])  ;; the list of registered balls

    (super-new)

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

    ;; register : SBall<%> -> Int
    ;; EFFECT: registers the given ball
    ;; RETURNS: the current position of the wall
    (define/public (register b)
      (begin
        (set! balls (cons b balls))
        pos))
```

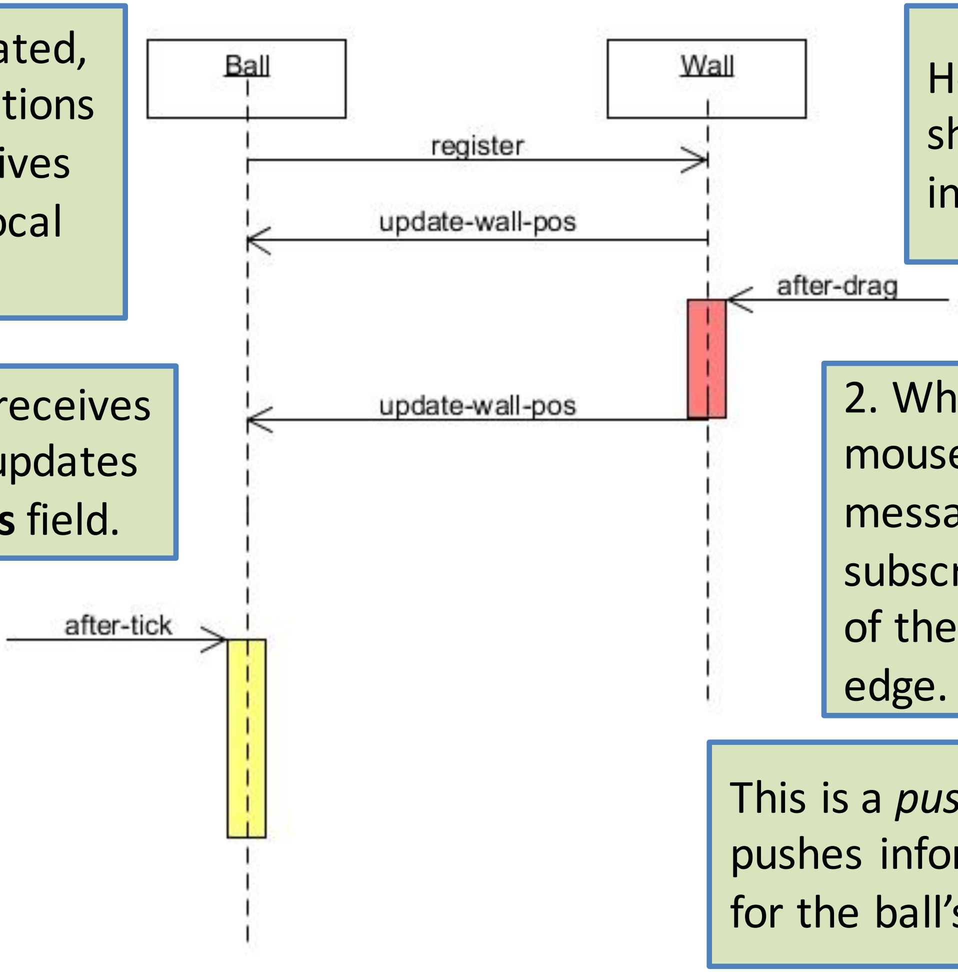
And the wall needs to publish whenever its position changes

```
; after-drag : Integer Integer -> Void
; GIVEN: the location of a drag event
; STRATEGY: Cases on whether the wall is selected.
; If it is selected, move it so that the vector from its position to
; the drag event is equal to saved-mx.  Report the new position to
; the registered balls.
(define/public (after-drag mx my)
  (if selected?
    (begin
      (set! pos (- mx saved-mx))
      (for-each
        (lambda (b) (send b update-wall-pos pos))
        balls))
    this))
```

How does a ball decide when to bounce in publish-subscribe.rkt?

1. When the ball is created, it subscribes to notifications from the wall and receives an initial value for its local **wall-pos** field

Here's a similar diagram showing what happens in publish-subscribe.rkt



2. When the wall receives a mouse drag, it sends out a message to all its subscribers notifying them of the new location of the edge.

3. When the ball receives this message, it updates its local **wall-pos** field.

4. When the ball receives an on-tick message, it consults its local **wall-pos** field to determine the current location of the right edge.

This is a *push* model: the wall pushes information to the ball for the ball's later use.

wall *pushes* information to the ball

Initializing the world

```
;; 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
         empty ; (list the-ball)  -- the ball is now stateful
         (list the-ball the-wall)))
     (define the-factory
       (new BallFactory% [wall the-wall][world the-world])))
    (begin
      ;; put the factory in the world
      (send the-world add-stateful-widget the-factory)
      the-world)))
```

But wait: this doesn't quite work

- If you run this, you'll see that the ball doesn't quite bounce at the right places.
- What happened?
- Hmm, must be time to think harder about testing and debugging stateful systems.

In our next lesson

- We'll see how to test and debug stateful objects
- In particular, we'll see how we found the bug in our system.

Reasons to use publish-subscribe

- Metaphor:
  - "you" are an information-supplier
  - You have many people that depend on your information
- Your information changes rarely, so most of your dependents' questions are redundant
- You don't know who needs your information

Other uses of publish-subscribe

- Use whenever you need to disseminate information to people you don't know.
- They sign up once, and then you promise to update them when something happens to you (eg your information changes)
- Both you and your subscribers must be stateful.

Summary

- Objects may need to know each other's identity:
  - either to *pull* information from that object
  - or to *push* information to that object
- Publish-subscribe enables you to send information to objects you don't know about
  - objects register with you ("subscribe")
  - you send them messages ("publish") when your information changes
  - must agree on protocol for transmission
    - eg: `(send <subscriber> method-name <data>)`
  - it's up to receiver to decide what to do with the data.

Next Steps

- Study 10-5-push-model.rkt in the Examples folder.
- Can you find the bug without looking ahead?
- If you have questions about this lesson, ask them on the Discussion Board