Patterns of Interaction 2: Publish-Subscribe

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 11.6
Key Points for Lesson 11.6

- 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.

• Problem: how does the information get to where it's needed?
• We've already talked about this a little in Lesson 11.1
• What happens in a stateful system?
How does a ball decide when to bounce in ball-factory.rkt?

Ball pulls info from the box

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

Diagrams like this are called sequence diagrams in UML.
Can we do better?

• The ball asks the box about its edge at every tick.
• But this information doesn't change very often.
• Better idea: Have the box 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, con'td

• So each ball must tell the box 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 box can find them.
• This pattern is called *publish/subscribe*—also called the *observer* pattern.
Video Demonstration: publish-subscribe.rkt

• http://www.youtube.com/watch?v=EsAO0QjCZbA
How does a ball decide when to bounce in publish-subscribe.rkt?

1. When the ball is created, it subscribes to notifications from the box.

2. When the box 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 right-edge field.

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

Box *pushes* information to the ball.

Here’s a similar diagram showing what happens in publish-subscribe.rkt.
Extending pub-sub

• What if we wanted to deal with multiple messages?
  – Say we wanted to move both left-edge and right-edge.

• Here are several possible designs:
Design #1: Separate subscription lists

• Each kind of message would have its own subscription list and its own method name
  
  • e.g., change-left-edge, change-right-edge

• Each class signs up for just the messages it wants to see.

• Will need separate interfaces for each kind of message, e.g., left-edge-publisher<%, left-edge-subscriber<%>

• Good choice if different groups of methods want to see different sets of messages.
Design #2: Single subscription list, multiple methods

• Better if most classes want to see most of the same messages.
• Have a different method name for each kind of message, eg `change-left-edge, change-right-edge` (same as design #1)
• All subscribers now see all the messages, so they need to supply methods for all the messages.
• The method can simply ignore the messages it’s not interested in.
• Single interface `subscriber<%>.
• This is just what we did in `World`, with `on-tick, on-mouse`, etc.
Design #3: Single subscription list, single method

• Like #2, but use the message body to represent which message it is.

• Can do this in different ways, too:
Represent messages in arguments

;; A Direction is one of:
;; -- "left"
;; -- "right"

;; Code in receiver:

;; change-edge : Direction NonNegInteger -> Void
;; EFFECT: sets the edge in the given direction to
;; the given value
;; STRATEGY: data decomp on dir : Direction
(define/public (change-edge dir val)
  (cond
    [(string=? dir "left") (set! left-edge val)]
    [(string=? dir "right") (set! right-edge val)])
)
Introduce data type of messages

(define-struct change-left-message (val))
(define-struct change-right-message (val)) ;; A ChangeEdgeMessage is one of
;; -- (make-change-left-message NonNegInteger)
;; -- (make-change-right-message NonNegInteger)

;; change-edge : ChangeEdgeMessage -> Void
;; EFFECT: sets the edge in the given direction to the given value
;; STRATEGY: data decomp on dir : Direction
(define/public (change-edge msg)
  (cond
[(change-left-message? msg)
 (set! left-edge (change-left-message-val msg))]
[(change-right-message? msg)
 (set! right-edge (change-right-message-val msg))])
The agreement between publisher and subscriber

• The publisher and subscriber must agree on a protocol for exchanging messages.

• The protocol consists of:
  – A method for an object to subscribe to the messages
  – A set of subscriber-side methods that the publisher can call to deliver the messages
  – An agreement on what messages mean and how they are represented.
A Shortcoming of publish-subscribe

• All of these designs use at least one of the subscriber’s method names.

• What if some other object wanted to subscribe to notifications from the box, but for some reason it still needed to use the name \texttt{edge-changed} for something else?

• Our current design ties up a method name (in this case \texttt{edge-changed}).
Doing pub-sub without relying on a common method name

• Better solution: instead of registering an object, register a function to be called.
  – \( f : X \to \text{Void} \) where \( X \) is the kind of value being published

• To publish a value, call each of the registered functions
  – It's a callback!

• These functions are called delegates or closures.
Video Demonstration: delegates.rkt

- [http://www.youtube.com/watch?v=01ZbNpkGbho](http://www.youtube.com/watch?v=01ZbNpkGbho)
Publishing through a delegate

(box = box1
  x = 100
  y = 50
  right-edge = ...

(f1 250)  250

(subscribe f1)

on-mouse

right-edge = 250

(publish 250)

(subscribers = (list f1))
Whose right-edge?

• When we write
  
  `(lambda (n) (set! right-edge n))`
  
  we are referring to the right-edge field in this object.

• The next slide shows a similar diagram illustrating what happens when there are two balls in the world.

• Each ball has its own delegate, which refers to its own right-edge field.
Many balls, many delegates
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.
Example: multiple viewers

• Imagine: some temperature is being monitored/modelled/controlled

• Multiple viewers:
  – display in Celsius
  – display in Fahrenheit
  – display on a slider

• May want to change/add viewers dynamically
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: (method-name <data>)
    • eg: call a registered closure with some data
  – it's up to receiver to decide what to do with the data.