Examining Two Pieces of Data

CS 5010 Program Design Paradigms

Lesson 2.3

You can only use one template at a time.

- If you need to do examine more than one value, examine one argument first, using its template, and pass the results on to a suitable help function or functions.

There's one small exception to this; see slides 10-11 below.

Examining multiple values: example #1

- https://www.youtube.com/watch?v=6XYpgHiXkA0

Reminder: “structural decomposition” (in the video) is just a fancier word for what we’re calling “using the template”.

YouTube link

Examining more than one value: example #2

- Let's consider **ball-after-mouse**:
- We are modelling the behavior of a ball in a simulation.
- The ball responds to mouse events. To model this response, we clearly have to look both at the ball and the mouse event.
- Let's look at the data definition and the functions.

Structural Decomposition on more than one value: example #2

- Contract and Purpose Statement:

```
;; ball-after-mouse :
;;    Ball Integer Integer MouseEvent -> Ball
;; GIVEN: a ball, a location and a mouse event
;; RETURNS: the ball after the given mouse event at
;; the given location.
```

- Remember, when we say "a ball", we mean “the state of the ball”: this function takes a ball state and returns another ball state.
- This is sometimes called “the successor-value pattern.”

Data Definition: Ball

```
(define-struct ball (x y radius selected?))

;; A Ball is a (make-ball Integer Integer Real Boolean)
;; x and y are the coordinates of the center of the ball,
;; in pixels, relative to the origin of the scene.
;; radius is the radius of the ball, in pixels
;; selected? is true iff the ball has been selected for dragging.

;; TEMPLATE:
;; (define (ball-fn b)
;;   (...
;;     (ball-x b) (ball-y b) (ball-radius b) (ball-selected? b)))
```

We follow the design recipe: we start with the data definitions.

ball-after-mouse

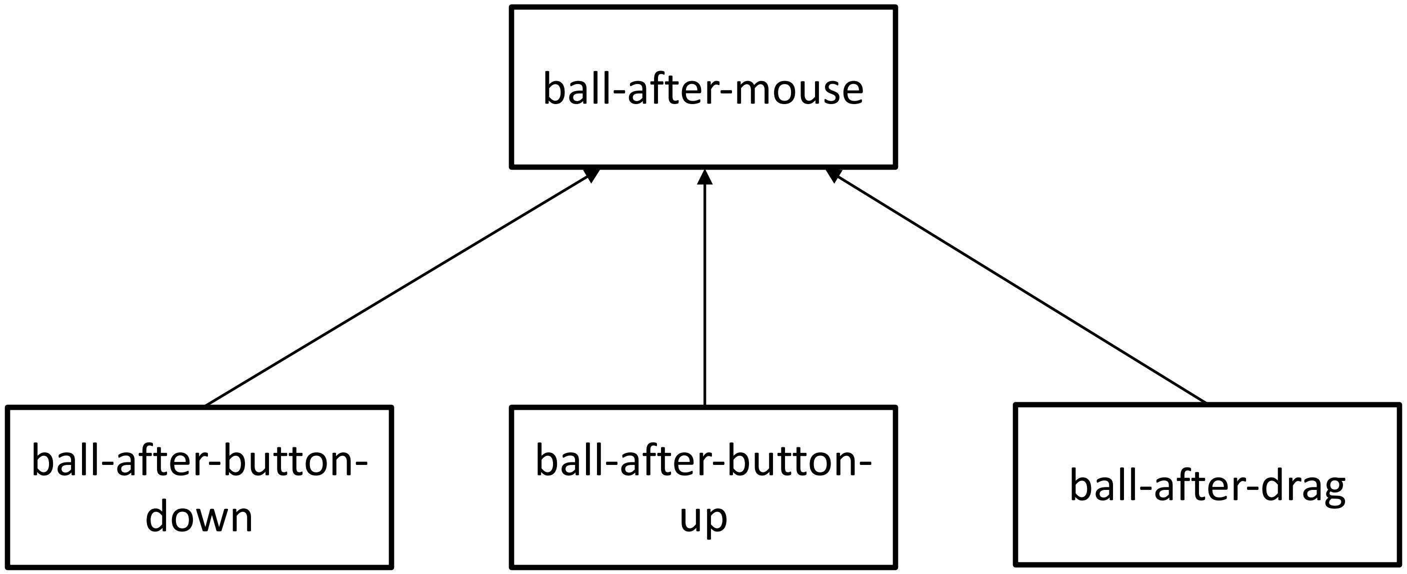
```
;; ball-after-mouse :
;;    Ball Integer Integer MouseEvent -> Ball
;; GIVEN: a ball, a location and a mouse event
;; RETURNS: the ball after the given mouse event at
;; the given location.
;; STRATEGY: Cases on mev
(define (ball-after-mouse b mx my mev)
  (cond
    [(mouse=? mev "button-down")
     (ball-after-button-down b mx my
    [(mouse=? mev "drag")
     (ball-after-drag b mx my)]
    [(mouse=? mev "button-up")
     (ball-after-button-up b mx my)]
    [else b]))
```

We first do cases on the mouse event. The data is handed off to one of several help functions. Each help function will decompose the compound data.

We now have a wishlist of functions to design:

1. **`ball-after-button-down`**
2. **`ball-after-drag`**
3. **`ball-after-button-up`**

Let’s draw a picture



This tree shows the organization of these functions.   The arrows go from the called function to the caller. Let’s explore **ball-after-drag**

ball-after-drag

```
;; ball-after-drag
;;      : Ball Integer Integer -> Ball
;; GIVEN: a ball and a location
;; RETURNS: the ball after a drag event at the
;; given location.
;; STRATEGY: Use template for Ball on b.
(define (ball-after-drag b x y)
  (if (ball-selected? b)
      (ball-moved-to b x y)
      b))
```

This moves the ball so its center is at the mouse point. That's probably not what you want in a real application. You probably want something that we call "smooth drag", which we'll learn about in a problem set coming up soon.

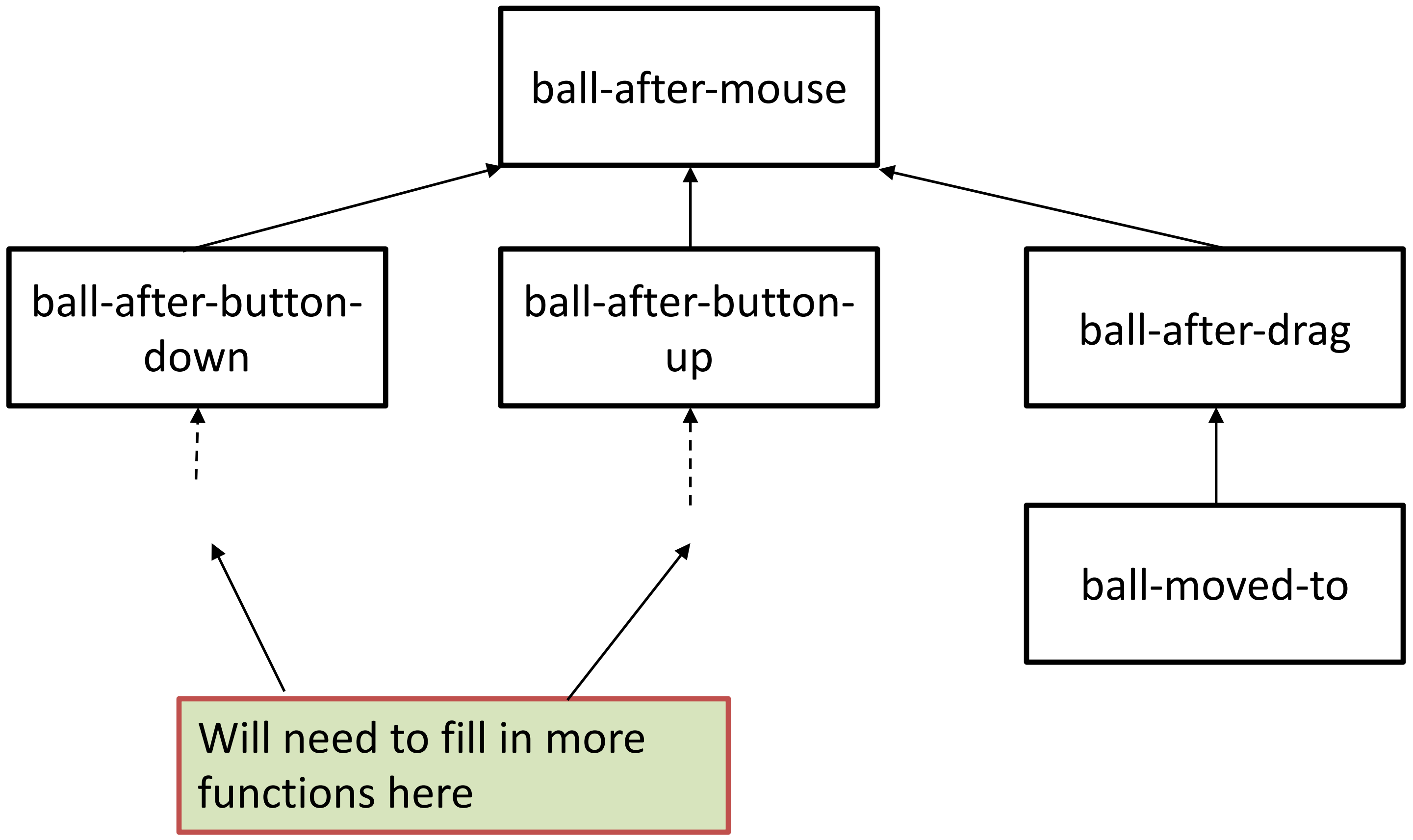
ball-moved-to

```
;; ball-moved-to : Ball Integer Integer -> Ball
;; GIVEN: a ball and a set of coordinates
;; RETURNS: a ball like the given one, except
;; that it has been moved to the given
;; coordinates.
;; STRATEGY: use template for Ball on b

(define (ball-moved-to b x y)
  (make-ball x y
    (ball-radius b)
    (ball-selected? b)))
```

So now we need to write **ball-moved-to**. It's also going to look at the data inside the ball, using the Ball template.

A bigger portion of the call tree



An inferior version of **ball-after-drag**

```
;; ball-after-drag
;;    : Ball Integer Integer -> Ball
;; GIVEN: a ball and a location
;; RETURNS: the ball after a drag event at the
;; given location.
;; STRATEGY: Use template for Ball on b


(define (ball-after-drag b x y)
  (if (ball-selected? b)
      (make-ball x y
                 (ball-radius b)
                 (ball-selected? b)))
      b))
```

This version is not as good as the preceding one, because it does two tasks: it decides WHEN to move the ball, and it also figures out HOW to move the ball.

Exception

- You can use the template for more than one compound if you really need to.

Example: balls-collide.rkt

```
;; balls-intersect? : Ball Ball -> Boolean
;; GIVEN: two balls
;; ANSWERS: do the balls intersect?
;; STRATEGY: Use template for Ball on b1 and b2.

(define (balls-intersect? b1 b2)
  (circles-intersect?
    (ball-x b1) (ball-y b1) (ball-radius b1)
    (ball-x b2) (ball-y b2) (ball-radius b2)))
```

This is OK, because trying to take the balls apart in separate functions just leads to awkward code.

circles-intersect?

```
;; circles-intersect? : Real^3 Real^3 -> Boolean
;; GIVEN: two positions and radii
;; ANSWERS: Would two circles with the given
;;  positions and radii intersect?
;; STRATEGY: Function Composition
(define (circles-intersect? x1 y1 r1 x2 y2 r2)
  (<=
    (+
      (sqr (- x1 x2))
      (sqr (- y1 y2)))
    (sqr (+ r1 r2))))
```

circles-intersect? knows about geometry. It doesn't know about balls: eg it doesn't know the field names of **Ball** or about **ball-selected?** .

If we changed the representation of balls, to add color, text, or to change the names of the fields, **circles-intersect?** wouldn't need to change.

If you didn't break up **balls-intersect?** with a help function like this, you would very likely be penalized for "needs help function"

Writing good definitions

- If your code is ugly, try decomposing things in the other order
- Remember: Keep it short!
  - If you have complicated junk in your function, you must have put it there for a reason. Turn it into a separate function so you can explain it and test it.
  - If your function is long and unruly, it probably means you are trying to do too much in one function. Break up your function into separate pieces and use "Combine Simpler Functions."

Summary

- We've now seen three Design Strategies:
  - Combine Simpler Functions
    - Combine simpler functions in series or pipeline
    - Use with any kind of data
  - Use Template
    - Used for enumeration , compound, or mixed data
    - Template gives sketch of function
    - Our most important tool
  - Cases
    - For when you need to divide data into cases, but the template doesn't fit.

Remember:
The shape of the data determines the shape of the program.

Next Steps

- Study the files
  - 02-3-traffic-light-with-timer.rkt
  - 02-4-ball-after-mouse.rkt
  - 02-5-balls-collide.rkt

  in the Examples folder.
  - Especially look at the tests. Observe how the unused code shows up in orange or black.
- If you have questions or comments about this lesson, post them on the discussion board.
- Go on to the next lesson.