Patterns of Communication Between Objects

CS 5010 Program Design Paradigms

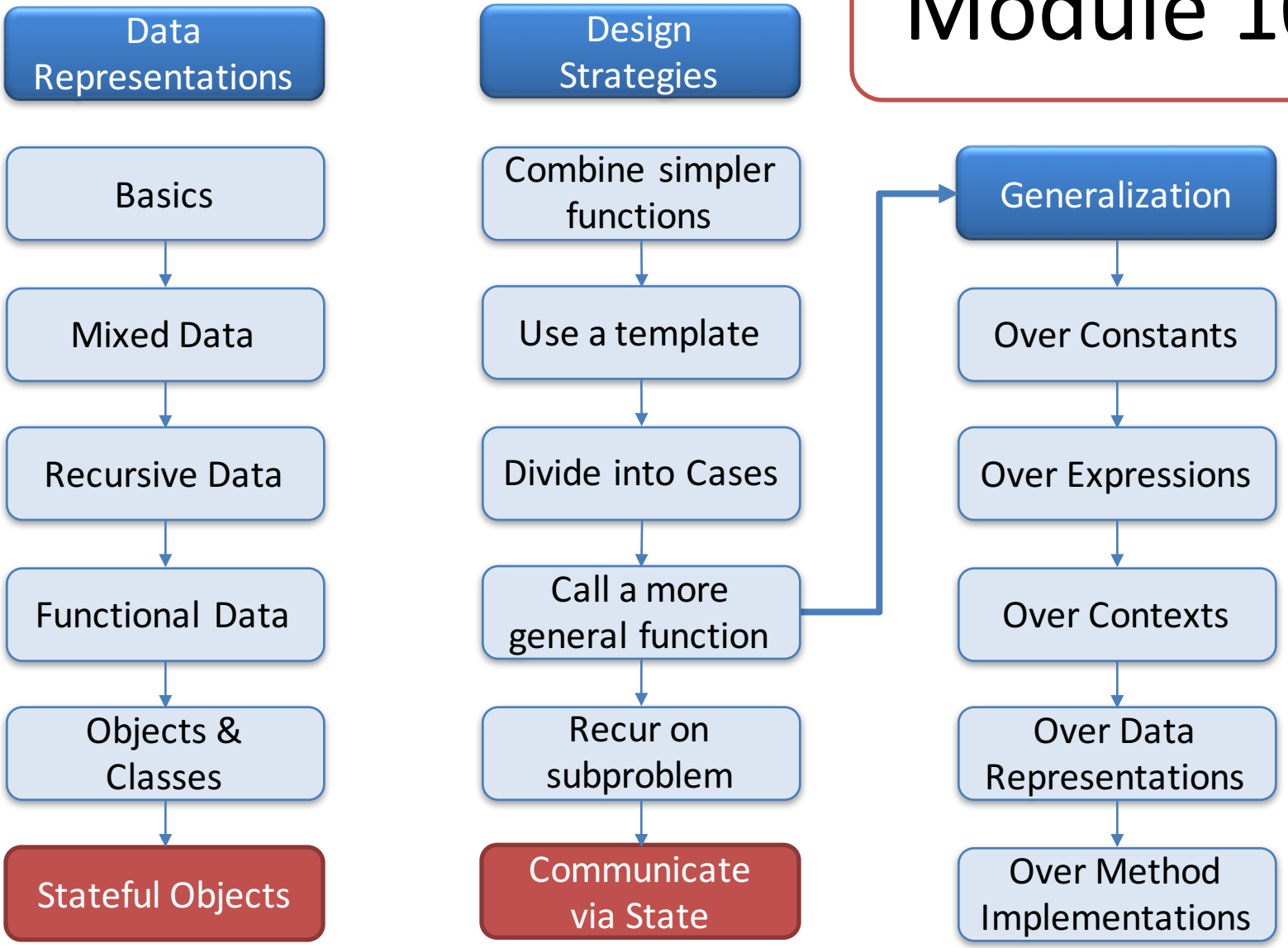
"Bootcamp"

Lesson 10.1

Key Points for this Module

- Objects can communicate in two basic ways: pull and push.
- Objects must have stable identity in order to communicate reliably
- We use stateful objects to implement objects with stable identity.
- Publish-subscribe is a common pattern for implementing push-style communication
- Delegates are a refinement of publish-subscribe.

Module 10


Data Representations
Basics
Mixed Data
Recursive Data
Functional Data
Objects & Classes
Stateful Objects
Design Strategies
Combine simpler functions
Use a template
Divide into Cases
Call a more general function
Recur on subproblem
Communicate via State
Generalization
Over Constants
Over Expressions
Over Contexts
Over Data Representations
Over Method Implementations

Key Points for Lesson 10.1

- Sometimes you need to combine data from two objects.
- The data could be combined in 3 possible places:
  - some external function (typical in functional organization, but generally considered bad OO design)
  - asking the other object to give you its data ("pull" model)
  - sending your information to the other object, and asking it to do the computation ("push" model)

Most methods have an obvious home

- Most of the time, we want to do calculations in the object where the data is.
- If you need to compute the area of a circle, make that a method of the **`Circle%`** class.

Sometimes you need to combine information from two objects

- How do you determine if two balls intersect?
- Let's look at three designs.

Design #1: Balls as just data structures

```
(define Ball0<%>
  (interface ()
    ;; -> Integer
    ;; RETURN: x, y coords of center and radius, all in pixels
    get-x
    get-y
    get-r))

(define Ball0%
  (class* object% (Ball0<%>)
    (init-field x y r)
    ; interpretation omitted...
    (super-new)
    (define/public (get-x) x)
    (define/public (get-y) y)
    (define/public (get-r) r)))
```

Design #1: Just use objects in place of structs. We equip our objects with methods that get each field, and do our computation outside of any object.

Here is the interface and a sample class definition in this style.

Implementation of first design

```
;; Ball0<%> Ball0<%> -> Boolean
(define (intersects? b1 b2)
  (coordinates-intersect?
    (send b1 get-x) (send b1 get-y) (send b1 get-r)
    (send b2 get-x) (send b2 get-y) (send b2 get-r)))

(define (coordinates-intersect? x1 y1 r1 x2 y2 r2)
  (<=
    (+ (sqr (- x1 x2)) (sqr (- y1 y2)))
    (sqr (+ r1 r2))))
```

This is considered poor OO design: we are just using objects as structs! We want to package the computation with the data.

Design #2: Collaborate by pulling information from the other object

```
(define Ball-Pull<%>
  (interface ()
    ;; -> Integer
    ;; RETURN: x, y coords of center and radius,
    ;; all in pixels
    get-x
    get-y
    get-r

    ;; Ball1<%> -> Boolean
    ;; Does the given ball intersect with this one?
    intersects?
    ))
```

In our second design, we add a method **intersects?** to the interface.

Method Definitions for Pull Model

```
(define Ball1%
  (class* object% (Ball-Pull<%>)
    (init-field x y r)  ; interpretation omitted...
    (super-new)
    ;; STRATEGY: Ask the other ball for its data
    (define/public (intersects? other-b)
      (coordinates-intersect?
        (send other-b get-x)
        (send other-b get-y)
        (send other-b get-r)))

    ;; Integer^3 -> Boolean
    ;; GIVEN: the coordinates of some ball
    ;; RETURNS: would that ball intersect this one?
    (define (coordinates-intersect? other-x other-y other-r)
      (<= (+ (sqr (- x other-x)) (sqr (- y other-y)))
        (sqr (+ r other-r))))

    (define/public (get-x) x)
    (define/public (get-y) y)
    (define/public (get-r) r)
))
```

Ask the other ball for its information

Do the computation here

Be prepared to answer if someone asks you the same questions!

Pull model: what happens

1. **`(send b1 intersects? b2)`**
2. **b1** asks **b2** for its data. **b2** gives it.
3. then **b1** does the arithmetic.

OK if **x**, **y**, **r** are already observable. But what if they are not?

b1 is the object that actually does the computation.

Design #3. Push Model: the object pushes its data to the other object

```
(define Ball-Push<%>
  (interface ()

    ;; Ball-Push<%> -> Boolean
    ;; does the given ball intersect this one?
    intersects?

    ;; Integer^3 -> Boolean
    ;; GIVEN: the x,y,r of some ball
    ;; RETURNS: would that ball
    ;;  intersect with this one?
    intersect-responder
    ))
```

In this design, when this ball is asked whether it intersects with some other ball, it sends its information to the other ball, and asks that ball to compute the intersection.

So now we have two methods

- **intersects?** sends this ball's data to the other ball.
- **intersect-responder** responds to the request, computing whether or not there is an intersection between the two balls.

Method Definitions for push model

```
;; A Ball2 is a (new Ball2% [x Integer][y Integer][r Integer])
(define Ball2%
  (class* object% (Ball-Push<%>)
    (init-field x y r) ; interpretation omitted...
    (super-new)

    (define/public (intersects? other-b)
      (send other-b intersect-responder x y r))

    ;; Integer^3 -> Boolean
    ;; GIVEN: the coordinates of some ball
    ;; RETURNS: would that ball intersect this one?
    (define/public
      (intersect-responder other-x other-y other-r)
      (<= (+ (sqr (- x other-x)) (sqr (- y other-y)))
        (sqr (- r other-r))))
    ))
```

Send your data to the other ball and ask him to finish the computation

If someone asks you a question, be prepared to answer it

Push model: what happens

1. **`(send b1 intersects? b2)`**
2. b1 sends its data to b2
3. b2 answers the question.

This is sometimes called “double dispatch”

b2 doesn’t know who’s asking.

In this design, **b2** is the ball that does the geometric calculation.

This pattern is also sometimes called “double dispatch”. It shows up often in object-oriented programming.

A related pattern is called “the visitor pattern.” This is a variation of this design when one of the structures is itemization data. We don’t have time in this course to deal with the visitor pattern, but you should now be equipped to learn about it.

Push or pull: how to choose?

- Most of the time the answer is clear: most operations naturally act on a particular object.
- Operations should happen in the object where the data resides
  - our first attempt was not good design
- Binary operations like intersect? are relatively rare in practice
  - either design 2 or design 3 would be ok for our purposes

Lesson Summary

- Sometimes you need to combine data from two objects.
- The data could be combined in 3 possible places:
  - some external function (typical in functional organization, but generally considered bad OO design)
  - asking the other object to give you its data ("pull" model)
  - sending your information to the other object, and asking it to do the computation ("push" model)

Next Steps

- Study 10-1-communicating-objects.rkt in the Examples folder
- If you have questions about this lesson, ask them on the Discussion Board
- Think about the following question:
  - How would b1 know about b2?
- Go on to the next lesson