The Design Recipe using Classes

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 10.5
Goals of this lesson

• See how the design recipe and its deliverables should appear in an object-oriented system

• Note: this is about OUR coding standards. Your workplace may have different standards.
Let's review the Design Recipe

<table>
<thead>
<tr>
<th>The Function Design Recipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Data Design</td>
</tr>
<tr>
<td>2. Signature and Purpose Statement</td>
</tr>
<tr>
<td>3. Examples and Tests</td>
</tr>
<tr>
<td>4. Design Strategy</td>
</tr>
<tr>
<td>5. Function Definition</td>
</tr>
<tr>
<td>6. Program Review</td>
</tr>
</tbody>
</table>
Step 1: Data Design

• For each kind of information we want to manipulate, we must choose a representation.

• The *data definition* documents our choice of representation. It has 4 components:
  – Whatever **define-structs** are needed.
  – A template that shows how to *construct* a value of the right kind.
  – An interpretation that shows how to *interpret* the data as information
  – For structured data, it includes a **template** that provides an outline for functions that manipulate that data.

• This serves as a reference as we design the rest of our program.
Representing information using classes and interfaces

<table>
<thead>
<tr>
<th>Functional Organization</th>
<th>Object-Oriented Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compound Data</td>
<td>Class with the same fields</td>
</tr>
<tr>
<td>Itemization Data</td>
<td>Interface</td>
</tr>
<tr>
<td>Mixed Data</td>
<td>• Interface specifies functions that work on that information</td>
</tr>
<tr>
<td></td>
<td>• One class for each variant, with the same fields as the variant</td>
</tr>
<tr>
<td></td>
<td>• Each class implements the interface</td>
</tr>
</tbody>
</table>

We've already discussed how information is represented in the object-oriented model.
What about the interpretation?

• For each class or interface write a purpose statement describing what information it represents.

• For each class, give an example of how to build an object of that class
  – just like A World is a (make-world ...)

• Each init-field should have an interpretation, just as every field in a struct has an interpretation.
Example

;; A World is a
;; (new World% [heli Helicopter] [bombs ListOf<Bomb>])
;; INTERPRETATION: represents a world containing a
;; helicopter and some bombs.
(define World%
  (class* object% (WorldObj<%>))
  (init-field heli) ; a Helicopter  --the
                  ; helicopter in the game
  (init-field bombs) ; a ListOf<Bomb> -- the list of
                ; bombs that the UFO has
                ; dropped.
What happened to the template?

• The object system does all the cond's for you.
• All that's left for you to do is to write the right-hand side of each cond-line.
  – You can use fields instead of selectors.
  – So there's no need for a separate template! (Yay!)
Step 2: Contract and Purpose Statement

• Contract and purpose statement go with the interface.
  – Each method in the interface has the same signature and the same purpose in each class.
  – That's the point of using an interface

• Each method definition should have a signature that refines the signature in the interface.

• Interface method signatures should be in terms of interfaces, not classes
Write signatures in terms of Interfaces, not Classes: Example

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

(define StupidRobot<%> 
  (interface () 
    ;; -> 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 
  ))

Here’s an interface for a very stupid robot
Write signatures in terms of interfaces, not classes

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

This works with ANY class that implements **Robot<%>**.
Example: signature in Interface

(define WorldObj<%>
  (interface ()

    ; -> WorldObj<%>
    ; Returns the WorldObj that should follow
    ; this one after a tick
    after-tick

    ...

  ))
In the Heli% class, the signature specifies that the result of after-tick is not just any object that implements the WorldObj<%> interface, but in fact it must be another object of class Heli% . Since Heli% implements the WorldObj<%> interface, this is OK.
Step 3: Examples and Tests

• Put these with the class or with the method, whichever works best.

• Phrase examples in terms of information (not data) whenever possible.

• Use meaningful names, etc., just as before.
Tests

• Checking equality of objects is usually the wrong question.
• Instead, use `check-equal?` on *observable* behavior.
• We’ll talk about this more in the next lesson.
Step 4: Design Strategy

• Design strategy is part of implementation, not interface
• So write down design strategy with each method definition.
• But that's hard, because a method definition generally corresponds to a cond-line rather than a whole function definition.
Step 4: Design Strategy

• So, in the interests of keeping your workload down, we will not require you to write down design strategies for most methods.
  – there are a few exceptions, which we'll illustrate below.
Method definitions that don't need design strategies

(define/public (weight) (* 1 1))

(define/public (volume) (* (send this height) (send this area)))

- functional combination of fields
- calling methods on this
Method definitions that don't need design strategies (2)

(define/public (weight)
  (+ (send front weight)
      (send back weight)))

calling methods on fields

(define/public (volume other-obj)
  (* (send other-obj area)
      (send other-obj height)))

calling methods on arguments
This also doesn't need a design strategy

(define/public (after-mouse-event x y evt)
  (new World%
    [heli (send heli after-mouse-event x y evt)]
    [bombs (map
      (lambda (bomb)
        (send bomb after-mouse-event x y evt))
      bombs)]))
A Method Definition that Needs a Design Strategy

;;; data decomposition on mev : MouseEvent
(define/public (after-mouse mx my mev)
  (cond
   [(mouse=? mev "button-down") ...]
   [(mouse=? mev "drag") ...]
   [(mouse=? mev "button-up") ...]
   [else ...]))

The MouseEvent template!
This is still just structural decomposition on mev.

;; data decomposition on mev : MouseEvent
(define/public (after-mouse mx my mev)
  (cond
    [(mouse=? mev "button-down") ...]
    [(mouse=? mev "drag")
     (send this after-drag mx my)]
    [(mouse=? mev "button-up") ...]
    [else ...]))

OK to do message sends as part of your "..."
Examples of Design Strategies

(define Graph%  
  (class* object% () ...  
  ;; STRATEGY: generative recursion  
  (define/public (path? src tgt)  
    (local  
      ((define (reachable-from? newest nodes)  
        ;; RETURNS: true iff there is a path from src to tgt in this graph  
        ;; INARIANT: newest is a subset of nodes  
        ;; AND:  
        ;;   (there is a path from src to tgt in this graph)  
        ;; iff (there is a path from newest to tgt)  
        ;; STRATEGY: general recursion  
        ;; HALTING MEASURE: the number of graph nodes _not_ in 'nodes'  
        (cond  
          [(member tgt newest) true]  
          [else (local  
            ((define candidates (set-diff  
              (send this all-successors newest)  
              nodes)))  
            ...etc...  
          )  
        ]  
      )  
    )  
  )  
)

Here's `path?` as a method of a `Graph%` class. It still uses generative recursion, so we must document that fact, and also provide all the usual deliverables for generative recursion.

We're talking about "this" graph"

Instead of saying `(all-successors newest graph)`, we made `all-successors` a method of `Graph%`, and we asked it to work on this graph.
Design Strategies turn into Patterns

• In OO world, the important design strategies are at the class level.

• Examples:
  – interpreter pattern (basis for our DD→OO recipe)
  – composite pattern (eg, composite shapes)
  – container pattern (we'll use this shortly)
  – template-and-hook pattern (later)
Step 6: Program Review

- All the same things apply!
Summary

• The Design Recipe is still there, but the deliverables are in different places
• You should now be able to identify where each of the deliverables go in an object-oriented program