The Design Recipe using Classes

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 9.5

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

The Function Design Recipe
1. Data Design
2. Contract and Purpose Statement
3. Examples and Tests
4. Design Strategy
5. Function Definition
6. Program Review

In an OO system, the steps are a little different, but they are all there

The Object-Oriented Design Recipe	
Step	**Description**
1. Interface Design	Identify the kinds of things in your system and the messages they need to respond to. For each method in an interface, write a contract and purpose statement.
2. Class Design	Identify the kinds of things that may be behind each interface. For each class, give a purpose statement. For each field of a class, give an interpretation.
3. Method Design	For each method, copy down the contract and purpose statement from the interface. Specialize the purpose statement to specify how the purpose is fulfilled for this class. Include examples as needed.
4. Unit Tests	For each class, write tests that exercise every method
5. Program Review	Same as before

Step 1: Interface Design

- What kinds of things will exist in your system?
- What messages will they need to respond to?
- List the messages (methods) in each interface
- Write a purpose statement for the interface
- For each method in the interface, write a contract and purpose statement.
- Write the contracts in terms of data types and interfaces (never classes).

Example 1: StupidRobot<%>

```
;; 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

    ))
```

Purpose statement for the interface

Example 2: Widget<%>

```
;; Every object that lives in the world must implement the Widget<%>
;; interface.

(define Widget<%>
  (interface ()

    ; -> Widget
    ; GIVEN: no arguments
    ; RETURNS: the state of this object that should follow at time t+1.
    after-tick

    ; Integer Integer -> Widget
    ; GIVEN: a location
    ; RETURNS: the state of this object that should follow the
    ; specified mouse event at the given location.
    after-button-down
    after-button-up
    after-drag

    ; KeyEvent -> Widget
    ; GIVEN: a key event
    ; RETURNS: the state of this object that should follow the
    ; given key event
    after-key-event

    ; Scene -> Scene
    ; GIVEN: a scene
    ; RETURNS: a scene like the given one, but with this object
    ; painted on it.
    add-to-scene
    ))
```

Another way to write a purpose statement for an interface

Step 2: Class Design

- For each interface, consider the different kinds of objects that will implement this interface. Each kind becomes a class.
- For each class, include a purpose statement that says what information is represented by objects of that class.
- For each class, give a constructor template showing how to build an object of that class.
- Each field should have an interpretation, just as every field in a **struct** has an interpretation.

Example

```
;; A Bomb is a (new Bomb% [x Integer][y Integer])
;; A Bomb represents a bomb.
;; A bomb just falls.  It has no other behavior.

(define Bomb%
  (class* object% (Widget<%>)
    (init-field x y)  ; the bomb's x and y position

    ;; image for displaying the bomb
    (field [BOMB-IMG (circle 10 "solid" "red")])
    ;; the bomb's speed, in pixels/tick
    (field [BOMB-SPEED 8])
```

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!)

Coding Standards

- Every method in the class (defined with **define/public**) MUST be listed in the interface.
- Exception: methods named **for-test:...** These methods may only be used for testing and debugging.
- You may have functions (defined with **define**) in your class. These will be private to the class.

Coding Standards Illustrated

```
;; A Foo is an object of any class that implements Foo<%>

(define Foo<%>
  (interface ()

    ; -> Integer
    ; purpose statement omitted...
    m1

    ; Bar -> Foo<%>
    ; purpose statement omitted...
    add-bar))
```

```
(define Class1%
  (class* object% (Foo<%>)

    (init-field a b c)
    ;; interpretations omitted...

    (field [LOCAL-CONSTANT ...])
    ;; interpretation omitted

    (super-new)

    ; m1 : -> Integer
    ; purpose statement omitted...
    (define/public (m1) ...)

    ; add-bar : Bar -> Foo<%>
    (define/public (add-bar b) ...)

    (define/public (method-not-in-interface ...) ...)

    (define (function1 ...) a b c this ...)
    (define (function2 ...) a b c this ...)

    ;; for-test:... methods don't need to be
    ;; in the interface

    (define/public (for-test:test-fcn1 ...) ...)

    ))
```

Interfaces and Classes should also have purpose statements; these are omitted here.

Constants used only in one class should be fields.

No methods except those listed in the interface

If you think you need a private method, use a function instead. Functions can refer to fields and to **this**.; these functions will not be accessible outside the class

Exception: methods named **for-test:...** need not be in the interface, but they may only be used for testing.

Step 3: Method Design

- Each method definition should have a contract that is the same as the contract in the interface.
- A method may have a purpose statement that specializes the purpose statement in the interface to the current class.
- Each method should have examples if needed to clarify the purpose statement.
- Each method should have associated tests. These will occur later in the file, with the unit tests.
- Document your method with a strategy if needed for explanation.

Remember, a strategy is a tweet-sized description of how your function works

Contract and Purpose Statement in Class

```
(define Bomb%
  (class* object% (Widget<%>)
    ...
    ;; after-tick : -> Widget<%>
    ;; RETURNS: A bomb like this one, but as it should be after a tick
    ;; DETAILS: the bomb moves vertically by BOMB-SPEED
    (define/public (after-tick)
      (new Bomb% [x x][y (+ y BOMB-SPEED)])))
```

Since **Bomb%** implements the **Widget<%>** interface, the value of **(after-tick)** is a **Widget**. So **after-tick** satisfies its contract.

Here's an example of a refined purpose statement

This one is so simple it doesn't need any examples.

Examples and Tests

- Examples and tests will generally be different.
- Put examples with the method.
- Phrase examples in terms of information (not data) whenever possible.
- Use meaningful names, etc., just as before.

Step 4: Unit Tests

- Write tests for a class after each class, or at the end of your file, whichever is clearer.
- Don't use **equal?** on objects. Test observable behavior instead, as we did in the preceding lesson.
- Construct testing scenarios and check to see that your objects have the right observable values afterwards.
- We still want 100% expression coverage, except for calls to **big-bang**.

What happened to the strategy?

- In the interests of keeping your workload down, we will not require you to write down design strategies for most methods.
- Write down strategies when they're helpful.

Simple method definitions don't need design strategies

```
(define/public (weight) (* l l))

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

Method definitions that don't need design strategies (2)

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

You could call this “recur on front and back” if you wanted, but you don’t have to.

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

This also doesn't need a design strategy, but it might help

```
;; STRATEGY: Use HOF map to send after-tick to each of the
;;  widgets

(define/public (after-tick)
  (new World%
    [widgets (map
              (lambda (widget) (send widget after-tick))
              widgets)]))
```

Another method where the design strategy is optional

```
;; STRATEGY: Cases on MouseEvent mev

(define/public (after-mouse-event mx my mev)
  (cond
    [(mouse=? mev "button-down") ...]
    [(mouse=? mev "drag") ...]
    [(mouse=? mev "button-up") ...]
    [else ...]))
```

Complicated things need strategies to document them

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

```
(define Graph%
 (class* object% ()
     ...

 (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
       ;; INVARIANT: 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: recur on successors of newest; halt when tgt is
       ;; found.
       ;; 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...
```

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

- Same as before:

The Program Review Recipe
1. Do all the tests pass?
2. Are the contracts accurate?
3. Are the purpose statements and interpretations clear and accurate?
4. Are there ugly pieces of code that should be broken out into their own functions?
5. Are there pieces of code that are duplicated (or almost duplicated) and should be made into independent functions?

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

Next Steps

- Study the files in the Examples folder. Did we get all the deliverables in the right places?
- If you have questions about this lesson, ask them on the Discussion Board.