The DD→OO Recipe

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 10.4
Introduction

• We’ve seen the connections:
  – objects are like structs
  – classes are like define-structs
  – interfaces are like data definitions

• We’ve seen that an OO program is just like a functional program, but with the pieces arranged in a different way

• If I had a functional program, just what would I need to do to convert it to an OO program?
Goals of this lesson

- Learn how to translate from data definitions to classes and interfaces
- Learn how to go from function definitions to method definitions (in more detail)

If the last lesson was the Big Picture, then this is the Small Picture.
Objects and Classes as Data Designs

• Objects and Classes provide a new representation for information.
• This representation is different from the one we have been using, but the correspondence between the two representations is simple.
• Here it is:
## 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>
Shapes: Data Definition

(define-struct my-circle (x y r color) #:transparent)
(define-struct my-square (x y l color) #:transparent)
(define-struct my-composite (front back) #:transparent)

;;; A Shape is one of
;;; -- (make-my-circle Number Number Number ColorString)
;;; -- (make-my-square Number Number Number ColorString)
;;; -- (make-my-composite Shape Shape)

;;; interp:
;;; ...

Here is the data definition for our shapes example.
And here is the corresponding class diagram. We have an interface for shapes, and three classes corresponding to the three variants. Each class has fields corresponding to the fields of the variant.

Note that we don't know the class of front or back. We only know their interface, but that's enough!
Recipe for converting from data defs to interfaces and classes

<table>
<thead>
<tr>
<th>Converting from Data Definitions to Interfaces and Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Define an interface for each kind of itemization data.</td>
</tr>
<tr>
<td>2. Define a class for each kind of compound data. In the</td>
</tr>
<tr>
<td>class, put in an <strong>init-field</strong> for each field in the</td>
</tr>
<tr>
<td>struct.</td>
</tr>
<tr>
<td>3. Convert <em>(make-whatever ...) to</em></td>
</tr>
<tr>
<td><em>(new whatever% [field1 ...][field2 ...])</em></td>
</tr>
<tr>
<td>4. For each function that follows the template, add a</td>
</tr>
<tr>
<td>method to the interface.</td>
</tr>
<tr>
<td>5. Convert functions to methods.</td>
</tr>
<tr>
<td><em>(see recipe below)</em></td>
</tr>
</tbody>
</table>

Here is a recipe for converting from data definitions and functions to interfaces and classes. The details of this recipe are specific to Racket, but the ideas can be used in any object-oriented language.
(define-struct foo
  (field1 left right))
(define-struct bar (lo hi))

;; Data Definition
;; A Baz is one of
;; -- (make-foo
; Number Baz Baz)
;; -- (make-bar
; Number Number)

Here’s an example of steps 1 and 2 of the recipe. The compound data Baz, with variants foo and bar, turns into an interface Baz<%>, with classes Foo% and Bar% that implement Baz<%>.

(define Baz<%>
  (interface ()
  ...
))

(define Foo%
  (class* object% (Baz<%>)
  (init-field
  field1 left right)
  ...
  (super-new)))

(define Bar%
  (class* object% (Baz<%>)
  (init-field lo hi)
  ...
  (super-new)))
Creating Objects

Replace

(make-bar 12 13)

by

(new Bar% [lo 12][hi 13])

Here’s an example of step 3 of the recipe.
From function calls to method calls

Instead of saying
(baz-fn a-baz n)
say
(send a-baz fn n)

Think:
"Send John a book"
(not "send a book to John").
From Function Definitions to Method Definitions

<table>
<thead>
<tr>
<th>Converting a Function Definition to a Method Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Add function name to interface</td>
</tr>
<tr>
<td>2. Pull out the parts</td>
</tr>
<tr>
<td>3. Change selectors to field references</td>
</tr>
<tr>
<td>4. Change function calls to method calls</td>
</tr>
<tr>
<td>5. Put method definitions into classes</td>
</tr>
</tbody>
</table>

Here is a recipe for converting a function definition to a method definition. This expands step 5 of the recipe above.
Turning a function definition into a set of method definitions

Example:

;; baz-mymin : Baz Number -> Number
(define (baz-mymin b n)
  (cond
   [(foo? b) (min
               (foo-field1 b)
               (baz-mymin (foo-left b) n)
               (baz-mymin (foo-right b) n)
               (other-fn b n))]
   [(bar? b) (min
               n (bar-lo b) (bar-hi b))])))
1. Add function name to the interface

Add to baz interface:

(define baz<%>
  (interface ()
    mymin    ; Number -> Number
    other-fn ; Number -> Number
    ...
  ))

First we add a method for this function to the interface. The name of the method need not be the same as the name of the function, but it should be similar.

The function version had contract **Baz Number -> Number**, so the method has contract **Number -> Number**, since the Baz argument is replaced by "this" object.

Here's another function that we have put in the interface.
Original Function Definition

Example:

;;; baz-mymin : Baz Number -> Number
(define (baz-mymin b n)
  (cond
   [(foo? b) (min
                (foo-field1 b)
                (baz-mymin (foo-left b) n)
                (baz-mymin (foo-right b) n)
                (other-fn b n))]
   [(bar? b) (min
              n (bar-lo b) (bar-hi b))])))
2. Pull Out the Parts

Example:

For a foo: \( \min \)
\[ (\text{foo-field1 } b) \]
\[ (\text{baz-mymin} (\text{foo-left } b) \ n) \]
\[ (\text{baz-mymin} (\text{foo-right } b) \ n) \]
\[ (\text{other-fn } b \ n)) \]

For a bar: \( \min \)
\[ n (\text{bar-lo } b) (\text{bar-hi } b)) \]
3. Change Selectors to field references

Example:

For a foo:  \( (\text{min} \]
\[
\text{field1}
\]
\[
(\text{baz-mymin left n})
\]
\[
(\text{baz-mymin right n})
\]
\[
(\text{other-fn this n})
\)

For a bar:  \( (\text{min} \]
\[
n \text{lo hi})
\)

(fioo-left b) is replaced by the field left.

A b that appears by itself refers to the whole struct, so it is replaced by this.
4. Change Function Calls to Method Calls

Example:

For a foo: (min
    field1
    (send left mymin n)
    (send right mymin n)
    (send this other-fn n))

For a bar: (min
    n lo hi)
5. Put method definitions into classes

Example:

In Foo%: 

(define/public (mymin n)
 (min
     field1
     (send left mymin n)
     (send right mymin n)
     (send this other-fn n)))

In Bar%: 

(define/public (mymin n)
 (min
     n lo hi))
Summary

• We've seen the connections:
  – objects are like structs
  – classes are like define-structs
  – interfaces are like data definitions

• We've seen how to translate from data definitions to classes and interfaces

• We've seen how to go from function definitions to method definitions.
Summary

• You should now be able to convert any set of functions working on mixed data to an OO style.