UMass Boston Computer Science

CS450 High Level Languages (section 2)

Functional Programming vs OOP

Wednesday, December 13, 2023

(last lecture!)
Logistics

• HW 10 out
  • Shapes! Written in OOP-style
  • due: Sun 12/17 11:59 pm EST

(last lecture!)
There’s Nothing Special About OOP!

• A typical (interface and classes) OOP program is just specific data definition / function design choice!
  • imposed by the language!

• Data definition:
  • itemization of compound data ...
  • ... where processing functions are grouped with other data fields!

• Function design:
  • Function to process this itemization data is split into separate “methods” (one for each kind of item in the itemization)
A Simple OO Example: Compare to CS450

```java
interface Shape {
    Image render();
}

class Circle {
    Num radius;
    Color col;

    Image render() {
        // render-circ
        return circ-img (radius, col);
    }
}

class Rectangle {
    Num width;
    Num height;
    Color col;

    Image render() {
        // render-rect
        return rect-img (width, height, col);
    }
}

;; A Shape is one of:
;; - Rectangle
;; - Circle
;; interp: Represents a shape image

;; A Circle is a (circ Num Color)
;; fields are radius and color

;; A Rectangle is a (rect Num Num Color)
;; fields are width, height, color
```
interface Shape
        Image render();

class Circle
        Num radius;
        Color col;
        Image render() { // render-circ
            return circ-img ( radius, col );
        }

class Rectangle
        Num width; Num height;
        Color col;
        Image render() { // render-rect
            return rect-img ( width, height, col );
        }

;; A Shape is one of:
;; - Rectangle
;; - Circle
;; interp: Represents a shape image

In OO langs, this “dispatch” function is implicitly written for you

(one cond clause of a)
Shape-processing function, as a (hidden) field!

Calls item-specific implementations

Last Time

A Simple OO Example: Compare to CS450
A Simple OO Example: as structs!

Last Time

interface Shape
Image render();

(struct Shape [render])

(required) method, as field

class Circle
Num radius;
Color col;

Image render() {
// render-circ
return circ-img (radius, col);
}

;; render-circ : Circle -> Image
(define (render-circ this) ...)

(struct circ Shape [r col])

“implements” interface

class Rectangle
Num width;
Num height;
Color col;

Image render() {
// render-rect
return rect-img (width, height, col);
}

;; render-rect : Rectangle -> Image
(define (render-rect this) ...)

(struct rect Shape [w h col])

“implements” interface

In OO langs, every method implicitly has a class instance arg ("this")!
OO-style Constructors ... with structs!

Last Time

manually write alternate Shape constructors, with explicit method imps

(define (mk-circ r col [circ-render-fn render-circ])
  (circ circ-render-fn r col))

;; render-circ : Circle -> Image
(define (render-circ this) ...)

(define (mk-rect w h col [rect-render-fn render-rect])
  (rect rect-render-fn w h col))

;; render-rect : Rectangle -> Image
(define (render-rect this) ... )
OO-style dispatch ... with structs!

Last Time

450-style “dispatch” function

```scheme
;; render : Shape -> Image
(define (render sh)
  (cond
   [(rect? sh) (render-rect sh)]
   [(circ? sh) (render-circ sh)]))
```

OO-Style “dispatch”

```scheme
(struct Shape [render])
```

```scheme
;; render : Shape -> Image
(define (render sh)
  (define (render-rect sh)
    ((shape-render-sh) sh))
  (define (render-circ sh)
    (match sh
      [define (circ r col)]
      [circle r "solid" col]))
)
```

```scheme
;; render-rect : Rectangle -> Image
(define (render-rect this) ...)
```

```scheme
;; render-circ : Circle -> Image
(define (render-circ this) ...)
```
OO vs CS450 Comparison

**OO Programming**

- interface + class imply specific (Itemization-of-compound) Data Def
- class (compound data) has fields and methods together!
- class constructor implicitly adds method_impls to created object
- data value to process is implicit method arg
- Implicit itemization **dispatch**

**CS 450 Design Recipe**

- Explicitly define any kind of Data Def
- struct (compound data) **fields** typically do not include functions
- data processing function is separate definition
- data value to process is explicit function arg
- Explicit itemization **dispatch** (cond)
OO vs CS450 “OO”-Style Comparison

**OO Programming**
- interface + class **imply** specific (itemization-of-compound) Data Def
- class (compound data) **has fields** and **methods** **together**!
- class constructor **implicitly adds** method impls to created object
- data value to process is **implicit** method arg
- Implicit itemization dispatch

**CS 450 “OO-style” Design Recipe**
- **Explicitly define** (itemization-of-compound) Data Def
- Include methods in struct (compound data) **fields**
- Define additional constructor with explicit method args
- data value to process is explicit function “method” arg
- Define **explicit** OO-style dispatch
How to Design ... OO-Style Programs

• For **Itemization Data Definition**
  1. List Item DataDefs (and other prev data def parts)
  2. Specify required methods
  3. Define "abstract" struct (with # fields = # of methods)
  4. Define explicit dispatch function(s) (one per method)

```plaintext
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; interp: Represents shape to draw on a canvas
;; Required methods:
;; - render : Shape -> Image

(struct Shape [render])

;; render : Shape -> Image
(define (render sh) ((shape-render sh) sh))
```
How to Design ... OO-Style Programs

Data Definition

Defer (and omit) order of defined methods; the "dispatch" struct (with dispatch function(s) (one per method)

• For each item:
  1. Define separate Data def;
  2. Define a struct, as substruct of “abstract” struct;
  3. Define required methods;
  4. Define constructor that includes method impls;

;; A Rectangle is a:
;; (rect width : Num
;;  height : Num
;;  color : Color)

;; A Circle is a:
;; (circ radius : Num
;;  color : Color)

;; render-circ : Circle -> Image
(define (render-circ this) ... )

;; render-rect : Rectangle -> Image
(define (render-rect this) ... )

(struct rect Shape [w h col])
(struct circ Shape [r col])

(define (mk-rect w h col
  [render
    render-rect])
  (rect render w h col))

(define (mk-circ r col
  [render
    render-circ])
  (circ render r col))
A Simple OO Example: Extensions?

Add a Triangle?

Add a rotate method?

Easy: Just define another class

```java
interface Shape
Image render();

class Circle
Num r;    Color col;
Image render() { return circ-img ( r, col ); }

class Rectangle
Num w;    Num h;    Color col;
Image render() { return rect-img ( w, h, col ); }

class Triangle
Num side1; // ...
Image render() { return tri-img ( ... ); }
```
A Simple OO Example: Extensions?

interface Shape
Image render();
Image rotate();

Add a rotate method?
Hard!: must update interface and every existing class
(might not have access!)

class Circle
Num r;    Color col;
Image render() {
    return circ-img ( r, col );
}
Circle rotate() { ... }

class Rectangle
Num w;    Num h;    Color col;
Image render() {
    return rect-img ( w, h, col );
}
Rectangle rotate() { ... }

class Triangle
Num side1; // ...
Image render() {
    return tri-img ( ... );
}
Triangle rotate() { ... }
Shapes, CS450 style

Add a Triangle?

Hard!: must:
- update data def,
- define new struct,
- update every existing “dispatch” function (might not have access!)

;; A Shape is one of:
;; - Rectangle
;; - Circle
;; interp: Represents a shape image

;; A Rectangle is a (rect Num Num Color)
;; fields are width, height, color
(struct rect [w h col])
;; A Circle is a (circ Num Color)
;; fields are radius and color
(struct circ [r col])

;; render: Shape -> Image
(define (render sh)
(cond
    [(rect? sh) (render-rect sh)]
    [(circ? sh) (render-circ sh)]))
Shapes, CS450 style

Add a Triangle?

*Hard!: must:
- update data def,
- define new struct,
- update every existing “dispatch” function (might not have access!)

```lisp
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; - Triangle
;; interp: Represents a shape image

;; A Rectangle is a (rect Num Num Color)
;; fields are width, height, color
(struct rect [w h col])
;; A Circle is a (circ Num Color)
;; fields are radius and color
(struct circ [r col])
;; A Triangle is a (tri ... )
;; fields are ...
(struct tri [ ... ])

;; render: Shape -> Image
(define (render sh)
  (cond
   [(rect? sh) (render-rect sh)]
   [(circ? sh) (render-circ sh)]
   [(tri? sh) (render-tri sh)])))
```
Shapes, CS450 style

Add a rotate function?

Easy!: Just define another function!

;; A Shape is one of:
;; - Rectangle
;; - Circle
;; interp: Represents a shape image

;; A Rectangle is a (rect Num Num Color)
;; fields are width, height, color
(struct rect [w h col])
;; A Circle is a (circ Num Color)
;; fields are radius and color
(struct circ [r col])

;; render: Shape -> Image
(define (render sh)
  (cond
   [(rect? sh) (render-rect sh)]
   [(circ? sh) (render-circ sh)]))

;; rotate: Shape -> Shape
(define (rotate sh)
  (cond
   [(rect? sh) (rotate-rect sh)]
   [(circ? sh) (rotate-circ sh)]))
FP vs OO Comparison

Adding another “item” to itemization, e.g., Triangle
• **OO**: Easy
  • Just define another class
    • Class methods only process that kind of item
    • Implicit “Dispatch” function(s) automatically updated
• **FP**: Hard
  • Must update data def, define another struct
  • Every explicit “dispatch” function must be manually updated with another cond clause

Adding a new operation for itemization data, e.g., rotate
• **OO**: Hard
  • Must update interface, and add new method to every class that implements it
• **FP**: Easy
  • Just define another function
A better way? Mixins demo

- A Mixin is a function, whose input and output is a class!

- Available in many languages:
  - RACKET
  - JAVASCRIPT
  - SCALA

- `(add-rotate-mixin class-without-rotate) => class-with-rotate`
Use Racket to create an OO-Style shape drawing API.

**Shape**

```racket
(struct Shape [place-method])
(struct rect Shape [w h topleft])
(struct circ Shape [r center])
```

**Circle**

```racket
Num r;
Posn center; // center coordinate

// places circle into given canvas
Image place-shape( Image canvas ) {
  return
    place-image( circ-img( r, ... ), ... center, ... canvas);
}
```

**Rectangle**

```racket
Num w; Num h;
Posn offset; // topleft corner

// places rect into given canvas
Image place-shape( Image canvas ) {
  return
    place-image ( rect-img(w, h, ...), ... offset, ... canvas);
}
```

**Recommended (OO-style) struct and data defs**

- `place-method` (that add methods to struct)
- `create` (one per operation)

**create-circ**: Posn Posn -> Circle
**create-rect**: Posn Posn -> Rectangle

---

Thank you for a great semester!
Submit your in-class work to github

Thank you for a great semester!