システム理論演習 II

# (9) 継承

増原英彦 (*masuhara@graco.c.u-tokyo.ac.jp*)

1997 年 6 月 27 日

## 1 クラスの拡張

クラスを定義する場合、どの程度一般的なものを作るのが適当だろうか? 例えば、前回定義した「乗り物」は、(1) 状態として位置と速度があり、(2) 速度を変えることができ、(3) 現在の速度と経過時間を基に位置が更新されるものであった。これは自動車であろうと、自転車であろうと、電車であろうと共通している。

走れば走るほど燃料を消費するような「自動車」を表現するために、この乗り物クラスを使うことを考えると、(4) 内部状態として燃料がある、(5) 移動すると燃料が減る、(6) 燃料が無くなると動けなくなる、という性質を加えることになる。また、「電車」であれば、レールの向きに移動方向が制限されたり、乗客がいるという性質を加える。

このような場合、ありとあらゆる性質を兼ね備えるように「乗り物」クラスを拡張すると、巨大な定義になってしまい、オブジェクトによる部品化の恩恵が損われてしまう。そこで、一般的なクラスには共通する性質だけを定義しておき、そのクラスをもとに、特殊な性質をつけ加えた新しいクラスを作ることを継承 (*inheritance*) と言う。もとになるクラスを親クラス、スーパークラス (*superclass*) あるいは基底クラス (*base class*) と呼び、拡張したクラスのことを子クラス、部分クラス (*subclass*) あるいは派生クラス (*derived class*) と呼ぶ。

子クラスを作る手続として `make-inherited-class` という手続を用意する。これによって作られたクラスは、基本的に ⟨親クラス⟩ と同じ状態変数とメソッドを持ち、さらに独自のものも持つ。

(`make-inherited-class` ⟨親クラス⟩ ⟨状態表作成手続⟩ ⟨メソッドリスト⟩)

「自動車」クラスでは、以下の 3 点についてが「乗り物」クラスからの変更になる。

- 状態変数として `position`, `velocity`, `fuel` を持つ
- 新しいメソッドとして、燃料の残量を知る `fuel`, 燃料が無くなったら、速度を 0 にする `check-fuel` を持つ
- メソッド `move!` は、クラス `vehicle` のメソッドに従って移動したあと、移動距離に応じて燃料を減らし、さらにその後で `check-fuel` を呼び出すものとする。

結局、手続 `make-inherited-class` は以下のように使う。

```
(define automobile
  (make-inherited-class
   vehicle                                ; 親クラス
   (lambda (p v f)                        ; 状態表作成手続
    (list 'position 'velocity v 'fuel f))
   (list 'fuel automobile-fuel            ; 追加 / 変更されるメソッド
         'check-fuel automobile-check-fuel
         'move! automobile-move!)))
```

### 1.1　メソッド表

継承クラスのインスタンスに対して、 send を行った場合、いままでのもと動作が変わってくる。まず所属クラスのメソッド表を探す。そこで指定された名前のメソッドが見つからなかったら、親クラスのメソッド表を探す。

そこで、メソッド表抽象データの定義を以下のように変更する。いままでの make-method-table の引数の整合性を保つために、 make-inherited-method-table という手続きでメソッド表を作ることにする。

- (make-inherited-method-table (list ⟨名前⟩ ⟨手続⟩ ...) ⟨親メソッド表⟩)— メソッド表を作る。

  ⟨親メソッド表⟩ は、親クラスのメソッド表。親クラスが無い場合には、偽 (#f) を与える。

- (make-method-table (list ⟨名前⟩ ⟨手続⟩ ...))— 親クラスの無いメソッド表を作る。

```
(define (make-method-table method-list)
  (make-inherited-method-table method-list #f))
```

- (method-table-lookup ⟨メソッド表⟩ ⟨名前⟩)— ⟨名前⟩ に対応する手続をとり出す。もし見つからなかった場合は偽 (#f) を返す

問題は method-table-lookup であるが、自クラス→親クラス→その親クラス→...... という順でメソッドを探すので、以下のような再帰的な定義になる。

```
(define (method-table-lookup method-table name)
  (let ((proc ⟨自クラスのメソッド表から name を探す⟩)
        (super ⟨親クラスのメソッド表⟩))
    (cond (proc proc)                              ; 自クラスに定義があった
          (super (method-table-lookup super name)) ; 親クラスがある
          (else #f))))                             ; メソッドがない
```

**課題 9-1: (method-table-lookup)**

手続 method-table-lookup を完成させよ。 ∎

### 1.2　クラスデータ

継承クラスを定義する (make-inherited-class ⟨親クラス⟩ ⟨状態表作成手続⟩ ⟨メソッドリスト⟩)) は、次の点を除いて make-class と同じものになる。

- 引数として ⟨親クラス⟩ が増えている
- ⟨親クラス⟩ からメソッド表を取り出して、 make-inherited-method-table を使ってメソッド表を作る

**課題 9-2: (make-inherited-class)**

手続 make-inherited-class を定義せよ。

ただし、クラスデータからメソッド表を取り出す手続 class-method-table が必要となるので、それも定義せよ。

∎

### 1.3　メソッド

新しい (つまり親クラスに定義されていない) メソッドを定義する場合は、いままで同じように定義できる。

```
(define fuel (send 'fuel))                          ; 総称関数の定義
(define check-fuel (send 'check-fuel))

(define (automobile-fuel a)                         ; 燃料の残量
  (with-slots ((fuel velocity) a)))

(define (automobile-check-fuel a)
  (if (< (fuel a) 0)                                ; 燃料が無くなると
      (set-speed! a (scale velocity 0))))           ; 速度を 0 にする
```

また、親クラスに定義されているメソッドを変更する場合、動作のつけ足しになることが多い。例えば、 move! メソッドは、次のような動作をする。

1. まず、「乗り物」メソッドの move! を使って、位置を動かす
2. 次に、移動距離と燃費から消費した燃料を求め、燃料を減らす
3. 最後に、燃料計のチェックをする

```
(define *cost/distance* 10) ; 燃費
(define (automobile-move! a time)
  (define (fuel-consumption dist)
    (* dist *cost/distance*))
  (let ((pos-before (position a)))             ; 移動前の位置
    (vehicle-move! a time)                     ; 親クラスのメソッド
    (let ((pos-after (position a)))            ; 移動後の位置
      (with-slot ((fuel) a)
        (set! fuel (- fuel                     ; 燃料を減らす
                      (fuel-consumption
                       (distance pos-before pos-after)))))
      (check-fuel a))))                        ; 燃料切れのチェック
```

注意: 親クラスの move! メソッドを呼び出すために、手続 vehicl-move! を直接呼び出してしまっている。また、状態変数の変更がある場合、 with-slots が重ならないようにする注意が必要になる。

## 2 抽象クラス

デカルト座標と極座標の 2 つのクラスには、共通する性質も多いのだが、一方が他方を継承していると見なすことは難しい。むしろ、この 2 つのクラスに「点」クラスという共通の祖先がいて、両者はその子クラスになっていると考える方が都合がよい。点クラスは、特定の内部表現を持っていないので、直接インスタンスを作ることはできない。しかし、 2 点を足す add や、点のスカラー倍を求める scale などは、共通の定義であり、点クラスのメソッドとして定義できる。逆に、 x-coord や radius のようなメソッドは、内部表現に依存するので、点クラスではなく、それぞれの子クラスで定義する必要がある。このようなクラスを抽象クラス (*abstract class*) という。

```
;;; 点クラス (抽象クラス) の定義
(define (point-add p1 p2)
  (let ((x1 (x-coord p1)) (y1 (y-coord p1))
        (x2 (x-coord p2)) (y2 (y-coord p2)))
    (new cartesian (+ x1 x2) (+ y1 y2))))
(define (point-mul p1 p2)
  (let ((r1 (radius p1)) (a1 (angle p1))
        (r2 (radius p2)) (a2 (angle p2)))
    (new polar (* r1 r2) (+ a1 a2))))

(define point
  (make-class
    (lambda () '())
    (list 'add point-add
          'mul point-mul
          ...)))
```

クラス cartesian、 polar は、 point の子クラスとして定義することができる。メソッド add、 mul 等はもはや固別に定義する必要がない。

```
(define cartesian
  (make-inherited-class
   point
   (lambda (x y) (list 'x x 'y y))
   (list 'x-coord cartesian-x
         'y-coord cartesian-y
         'radius  cartesian-radius
         'angle   cartesian-angle
         'print   cartesian-print)))

(define ploar
  (make-inherited-class
   point
   (lambda (r a) (list 'r r 'a a))
   (list 'x-coord polar-x
         'y-coord polar-y
         'radius  polar-radius
         'angle   polar-angle
         'print   polar-print)))
```

**課題 9-3:** (抽象クラス)

クラス point に、以下の手続きを定義せよ。

- スカラー倍をする (scale p s) (点 $(s, 0)$ との乗算になる)
- 2 点の引き算をする (sub p1 p2) ($-1$ 倍して足す)
- 2 点間の距離を求める (distance p1 p2) (差の radius になる)

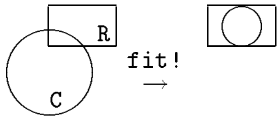
計算例:

```
> (let ((p0 (new cartesian 1 2)))
    (let ((p1 (add p0 (new cartesian (sqrt 12) 0)))
          (p2 (add p0 (new polar 4 (/ pi 6)))))
      (distance p1 p2)))
⟨??⟩
```

∎

**課題 9-4:** (さらに抽象クラス)

課題 8-2, 8-3 では長方形、円、合成図形を定義したが、これらに共通する親クラスとして、 graphic-object を定義せよ。この子クラスは、 center(中心の座標を返す)、 width(図形全体の幅を返す)、 height(高さ)、 move!、 scale!、 draw!(絵を描く) などのメソッドを定義しているものとする。また、子クラス rectangle,circle, composite の定義を修正せよ。




抽象クラス graphic-object のメソッドとして、図形をある長方形の領域におさまるように移動、拡大・縮小する fit! が定義できる。例えば、(fit! C R) とすると、上の図のような変形をしてくれるものとする。このメソッドの定義は、以下のような感じになる。実際に動作を確めよ。

```
(define (graphic-object-fit! obj rect)
  (let ((obj-w  (width obj))  (obj-h  (height obj)) ; 幅・高さを求める
        (rect-w (width rect)) (rect-h (height rect)))
    (let ((scaling-ratio                              ; 縦横の各比率を求め
           (min (/ rect-w obj-w) (/ rect-h obj-h)))); 小さい方をとる
      (scale! obj scaling-ratio)                      ; 図形を拡大・縮小
      (move! obj (center rect)))))                    ; 中心を合わせる
```

∎

## 3　メタクラス (のお話し)

今回の資料では、クラスに継承機能を付加するために、クラスデータやメソッド表といったデータを拡張していった。この先、より強力なオブジェクト指向プログラミングを提れしようと思ったら、さらに拡張が必要になるだろう。

もしクラスデータやメソッド表自身がオブジェクトであったら、こういった拡張も容易にできるかも知れない。そこで、 `automobile` のようなクラスも、あるクラスのインスタンスであるとする。そのようなクラスのことをメタクラス (*metaclass*) と言う。同様に、メソッド表に対しても何らかのメタクラスがあることになる。

また、 `automobile` のようなクラスを、メタクラスのインスタンスだと思う場合、このようなオブジェクトのことをメタオブジェクト (*metaobject*) と呼ぶ。メタオブジェクトが規定するメソッドや状態変数の名前といったものをメタオブジェクトプロトコル (*metaobject protocol*) という。

当然、メタクラスはクラスであるので、上と同じ議論を繰り返せば、何らかのクラスのインスタンス、ということになり、無限にクラスが必要になるが、多くの場合は、どこかに原始クラス c を用意しておいて、 c のメタクラスも c である、という状態を作っておく。

ところで、 `make-class` のような手続は、メタクラスのインスタンスを生成するものになるわけだが、メタクラスを定義するためには `make-class` が無ければいけない。このようなシステムは、一見、実現不可能に見えるが、

1. これまでやったように、クラスやメソッド表を、単な抽象データとして定義して、オブジェクト指向なシステムを作る。
2. その上でメタクラスの定義を作る。
3. メタクラスのメソッド (インスタンス生成・子クラス定義等々) は、直接対応する手続きを実行し、あたかもメソッドが実行されたかのように振る舞う

のようにして作ることができる。

## 4　最終課題 (の予告)

何か規模の大きな or 役に立つ or 凝ったプログラムを作成せよ。特に複数人で共同開発することも認めるつもりなので、この段階から構想を立てておくとスムースにできるかも知れない。詳しいことについては、次回以降に説明する予定である。