情報システム科学実習 II (10)

継承

1999 年 12 月 22 日 増原 英彦

多くのオブジェクト指向言語では、同じ性質を持ったオブジェクトをまとめたクラスを定義し、引数のクラスによって異なるメソッドを実行させることで多相性を実現していた。全く同じではなくても、部分的に性質を持っているクラスがあった場合、継承 (**inheritance**) によって定義を共通化できる。

1 クラス定義の継承

あるクラスの定義から、それを特殊化したクラス定義を作ることをクラス定義の継承 (**inheritance**) と言う。

例として学校の名簿を作ることを考える。学校には教師と生徒がいる。名簿には氏名、住所のほか、教師であれば教えている科目を、生徒であれば学籍番号を印刷することにしよう。

実世界では教師と生徒は異なる種類のモノに対応しているのでそれぞれのクラスを定義するのが自然だろう。従って次のようなクラス定義になる:

```
(define-class <Teacher> ()
  ((subject :init-keyword :subject :getter get-subject)
   (name :init-keyword :name :getter get-name)
   (address :init-keyword :address :getter get-address)))

(define-class <Student> ()
  ((id :init-keyword :id :getter get-id)
   (name :init-keyword :name :getter get-name)
   (address :init-keyword :address :getter get-address)))
```

クラス定義の継承機能を使えば、共通する性質を一つのクラスにまとめることができる。教師と生徒の場合、どちらも名前や住所を持つ「人間」という点では共通であるので、次のような定義にすることができる:

```
(define-class <Person> ()
  ((name :init-keyword :name :getter get-name)
   (address :init-keyword :address :getter get-address)))

(define-class <Teacher> (<Person>)
  ((subject :init-keyword :subject :getter get-subject)
   (classes :init-keyword :classes :getter get-classes)))

(define-class <Student> (<Person>)
  ((class :init-keyword :class :getter get-class)
   (id :init-keyword :id :getter get-id)))
```

文法: クラス定義の文法は(define-class クラス名（クラス名）（スロット定義...))のようになる。1 番目のクラス名は定義されるクラスの名前、2 番目のクラス名[1]では親クラスまたはスーパークラスの名前を指定する。逆に、定義されるクラスは子クラス、派生クラス、サブクラスなどと呼ばれる。

スロット定義の継承: 親クラスが P であるようなクラス C のオブジェクトは、C のために定義されたスロットの他に、クラス P のオブジェクトが持つ全てのスロットを持つ。

2 メソッドの継承

名簿の印刷をするメソッドは、教師と生徒で異なる情報を表示しなければならないので、別々に定義するのが自然だろう。しかし、どちらの定義でも名前と住所を表示する部分は共通である。しかも、将来名簿を拡張して、電話番号を扱えるようにしたくなるかも知れない。そのような場合、共通部分は一つの定義になっていた方が便利である。

クラス定義を継承している場合、メソッド定義も同時に継承される。この場合、<Person>クラスに関してメソッドを定義すれば、<Teacher>クラスのオブジェクトも<Student>クラスのオブジェクトも<Person>クラス用のメソッドが適用できる:

```
STk> (define-method make-string ((x <Person>))
       (string-append (get-name x) ":" (get-address x)))
STk> (define sato
       (make <Teacher> :subject "English" :name "Ichiro Sato"
                       :address "Yokohama, Kanagawa"))
sato
STk> (define tanaka
       (make <Student> :id 1234 :name "Jiro Tanaka"
                       :address "Tsukuba, Ibaraki"))
tanaka
STk> (make-string sato)
"Ichiro Sato:Yokohama, Kanagawa"
STk> (make-string tanaka)
"Jiro Tanaka:Tsukuba, Ibaraki"
```

子クラスにメソッドが定義された場合は、それが優先される:

```
STk> (define-method make-string ((x <Teacher>))
       (string-append (get-name x)
                      ":(teacher's address is hidden)"))
STk> (make-string sato)
"Ichiro Sato:(teacher's address is hidden)"
```

(next-method) と書くと、子クラスのメソッドから親クラスの同名のメソッドを呼び出すことができる。これによって、「大部分は親クラスのメソッドのままだけど、少しだけ追加・変更をしたい」といったことができる:

[1]実は 2 つ以上のクラス名を書くことができる。

```
STk> (define-method make-string ((x <Teacher>))
       (string-append (get-name x) ":" (next-method)))
STk> (define-method make-string ((x <Student>))
       (string-append (number->string (get-id x))
                      ":" (next-method)))
STk> (make-string sato)
"Ichiro Sato:Ichiro Sato:Yokohama, Kanagawa"
STk> (make-string tanaka)
"1234:Jiro Tanaka:Tsukuba, Ibaraki"
```

課題 **10-1:** (描画エディタ)

前回例として出した描画エディタにおける線や円などの図形要素は、(1) 図形が配置されている位置と (2) キャンバス上に描かれたオブジェクトを持つという点で共通している。そこで、

1. 図形要素に共通する親クラスとして<DrawingElement>を定義せよ。スロットとしては上の 2 つを持つとする。

2. クラス<Stroke>, <Box>, <Circle>などを<DrawingElement>の子クラスとして定義し直せ。

3. もとのメソッドdraw では、

   (a) 以前に描いた図形オブジェクトをキャンバスから消す

   (b) キャンバスに新しく図形オブジェクトを描く

   (c) 描かれた図形オブジェクトを図形要素のスロットに書き込む

   の 3 つの動作をしていた。このうち、(b) は図形要素ごとに異なる操作になる。そこで、(b) の部分を別のメソッドdraw-on-canvas によって実現することにして、次のようにdraw を定義し直す:

   ```
   (define-method draw ((e <DrawingElement>) canvas)
     (let ((obj (drawing-object e)))
       (if obj (destroy obj)))             ; erase if drawn before
     (set-drawing-object
      e (draw-on-canvas e (position e) canvas)))
   ```

   図形要素ごとのメソッドdraw-on-canvas を定義せよ。

4. メソッドmove を<DrawingElement> のメソッドとして定義し直せ。

∎

課題 **10-2:** (図形要素の特化)

クラス<Stroke>の子クラスとして水平線を描画するクラス HorizontalStroke を定義せよ。メソッドresize を変更して、変形しても水平線であり続けるようにせよ。 ∎