情報システム科学実習 II (11)

デザイン・パターン

1999 年 1 月 12 日 増原 英彦

1 デザイン・パターン

オブジェクト指向 (言語) でプログラムを書く理由はプログラムの再利用性を高くすることである。しかし、オブジェクト指向 (言語) をどう使えば再利用性が高くなるかは自動ではない。デザイン・パターン (**design patterns**)[GHJV95] は、オブジェクト指向 (言語) によって再利用性の高いプログラムを作るための一つの解である。何をしてくれるかというと:

- 「このような場合には、このようにプログラムすればよい」という経験的なものである (理論的な裏付けがあるわけではない)
- プログラム中の数個のクラスの関係についての指針である (大きなシステムの全体の設計に関する話ではないし、1 つのクラスの設計方法でもない)
- カタログである (人間が見て「これを使おう」と思うもの)

2 パターンの例: Observer パターン

(配布資料参照)

3 例題: 時計プログラム

時計には「時を刻む」機能と「時刻を表示する」機能があるが、これらを切り離すことができると双方の再利用性が高まる。そこで、前者を subject 側、後者を observer 側として作る。

3.1 Observer パターン

まず、Observer パターン部分としてクラス `<Subject>` と`<Observer>`を定義する (以下、web ページの`observer.scm` にて定義を配布):

```
(require "stklos")

(define-class <Subject> ()
  ((observers :initform '()
              :getter observers
              :setter set-observers!)))
(define-class <Observer> ()
  ());; スロット定義なし

(define-method attach ((s <Subject>) (o <Observer>))
  (set-observers! s (cons o (observers s))))
(define-method detach ((s <Subject>) (o <Observer>))
  (set-observers! s (remove o (observers s))))
(define-method notify ((s <Subject>))
  (for-each do-update (observers s)))
```

- 関数(remove e l)はリストlの中でeと同じ要素があったら(高々1つだけ)とり除いたリストを返す。(observer.scmに定義がある。)
- 関数(for-each p l)はリストlの各要素について関数pを適用する。例えば(for-each display (list "A" "B"))は(begin (display "A") (display "B"))とほぼ同じである。
- パターン中の *Observer* は、スロットやメソッドを持たない抽象クラスである。STklosやCLOSのような言語ではこのようなクラスを実際に定義しなくても、updateという名前のメソッドを持ったクラスであれば何でも*Observer* になり得る。

  配布した資料[GHJV95]は、C++やJavaのように静的に型付けされた言語も対象としているため、このようなクラスを定義する必要がある。(Javaではインターフェイスとして定義する方がよかろう。)
- もともとのObserverパターンでは *Observer* クラスのメソッド名はUpdateとなっていたが、ここではSTkの組み込み関数と名前が衝突するのでdo-updateとした。

3.2 時を刻む部分

時計の時を刻む部分としてクラス<ClockTimer>を定義する。このクラスは、次のようなインターフェイスを持っている:

- (make <ClockTimer> :hour h :minute m :second s)によって、h時m分s秒を表わす時計が作られる。
- メソッドget-hour, get-minute, get-secondによって、その時計が表わしている時刻が得られる。
- メソッドtickによって1秒時刻が進む。

さらにこのクラスは<Subject>の子クラスとして定義する。つまり、時刻が変化した場合は登録されたobserversのupdateメソッドが呼び出すためにnotifyメソッドを呼ぶようにする。(以下の定義はwebページclock.scmにある。)

```
(define-class <ClockTimer> (<Subject>)
  ((hour   :init-keyword :hour   :getter get-hour
           :setter set-hour!)
   (minute :init-keyword :minute :getter get-minute
           :setter set-minute!)
   (second :init-keyword :second :getter get-second
           :setter set-second!)))

(define-method tick ((c <ClockTimer>))
  (let* ((next-second (+ (get-second c) 1))
         (next-minute (+ (get-minute c)
                         (quotient next-second 60)))
         (next-hour   (+ (get-hour   c)
                         (quotient next-minute 60))))
    (set-hour!   c next-hour)
    (set-minute! c (remainder next-minute 60))
    (set-second! c (remainder next-second 60)))
  (notify c)) ; observers に変更を伝える
```

ここでは 1 秒ごとにメソッド tick が呼ばれるものとする。このメソッドは内部状態を変化させるので最後に notify を呼ぶ。

3.3 時刻を表示する部分

時刻を表示する部分として、ここでは単純にコンソールに文字列を書くクラス<ConsoleClock>を定義しよう。

- このクラスは、do-update が呼ばれた際に、最新の時刻を得る必要があるため、<ClockTimer>クラスのオブジェクトへの参照を持っている必要がある。そこで、スロットsubject を用意する。
- メソッドdo-update はスロットsubject に入っているオブジェクトの最新の時刻を問い合わせ、表示する。表示は時分秒をコロンでつないでコンソールに書くだけである。
- このクラスのオブジェクトは、<ClockTimer>のオブジェクトにattach される必要がある。ここでは、オブジェクトがmake によって作られた際に必ず呼ばれるメソッドinitialize をオーバーライドすることで実現している。

```
(define-class <ConsoleClock> (<Observer>)
  ((subject :init-keyword :subject
            :initform #f
            :getter get-subject)))

(define (hms->string hour minute second)
  (string-append
   (number->string hour) ":" (number->string minute) ":"
   (number->string second)))

(define-method do-update ((c <ConsoleClock>))
  (let ((subject (get-subject c)))
    (display (hms->string (get-hour subject)
                          (get-minute subject)
                          (get-second subject)))
    (newline)))

(define-method initialize ((c <ConsoleClock>) initargs)
  (next-method) ; 親クラスの initialize を呼び出すことで、
                ; スロットの初期化がなされる
  (let ((subject (get-subject c)))
    (if subject
        (attach subject c))))
```

実行例:

```
STk> (define clock
       (make <ClockTimer> :hour 22 :minute 54 :second 13))
clock
STk> (define cc (make <ConsoleClock> :subject clock))
cc
STk> (tick clock)
22:54:14
STk> (tick clock)
22:54:15
STk> (tick clock)
22:54:16
```

課題 **11-1:** (デジタル時計)

時刻をウィンドウに表示する時計を作れ。

1. クラス<DigitalClock>を定義せよ。スロットとしては、subject の他に、<Label>クラスのオブジェクトを保持するものが必要になるだろう。
2. クラス<DigitalClock>をメソッドdo-update を定義せよ。スロットsubject のオブジェクトから最新の状態をとり出して、<Label>クラスのオブジェクトに文字列を書き込むものになるはずである。
3. クラス<DigitalClock>をメソッドinitialize を定義せよ。
4. クラス<DigitalClock>のオブジェクトを作り、時計を動かしてみよ。
5. 余力があったら、tick メソッドを呼び出すボタンもつけてみよ。

課題 11-2: (アナログ時計)
時刻をアナログで表示する時計を作れ。(ヒント: アナログの針を描くには第 3 回で使ったタートルグラフィクスを使うと便利かも知れない。) ∎

練習 11-1: (複数の observer)
1 つの<ClockTimer>オブジェクトに対して複数の observer オブジェクトを作った場合でも、全ての表示が正しく行われることを確認せよ。 ∎

課題 11-3: (砂時計)
残り時間を測る時計を作れ。残り時間が無くなったらアラームを鳴らすものとする。
<ClockTimer>とほとんど同じだが、tick によって時間を減らし、残り時間が 0 になった時にアラームを鳴らすようにすればよい。
表示方法としては、<ClockTimer>のために作った observer クラスをほとんどそのまま使えるはずである。必要に応じてアラームを鳴らすなどの拡張をする。 ∎

参考文献

[GHJV95] Erich Gamma, Richrad Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.