https://www.laurentbloch.net/BlogLB/Envoi-de-messages-et-encapsulation

Envoi de messages et encapsulation

- Zinformatiques - Cours de bioinformatique au CNAM -

Date de mise en ligne : lundi 4 octobre 2004

Le dernier cours a introduit la quatrième primitive fondamentale : `set!`

Avec `set!` nous pouvons modifier des variables dont la durée de vie doit excéder le temps d'exécution de la procédure qui y accède. Nous avons vu deux exemples très simples : une variable qui contient la valeur du solde d'un compte bancaire, et une variable qui comptabilise le nombre d'exécutions d'une procédure.

Une façon de créer de tels objets serait de créer des variables globales, c'est-à-dire au *Top-level*. Mais cette façon de procéder a de nombreux inconvénients, elle est une source d'erreurs.

Un meilleur procédé consiste à encapsuler la variable en question dans une fermeture. Ainsi la variable aura bien une durée de vie qui excède le temps d'exécution des procédures qui y accèdent, mais sans être visible directement. Une telle variable est appelée *variable rémanente*.

Pour ce faire nous allons utiliser ce que nous savons des fermetures. Une fermeture est constituée d'un environnement et d'une lambda-expression. Si nous voulons ainsi créer un compteur, voici comment nous allons procéder :

– je crée un objet nommé `compteur` :

```
(define compteur
```

– dans cet objet j'inclus une liaison entre la variable qui représentera la valeur du compteur et la valeur, au début bien sûr égale à 0 :

```
(define compteur    (let ((valeur 0))
```

– maintenant il me faut placer ici une lambda-expression (c'est-à-dire le corps d'une procédure) qui saura recevoir des messages ; ces messages lui diront ce qu'elle doit faire :

```
(define compteur    (let ((valeur 0))       (lambda (message)
```

– que doit savoir faire un compteur ? Me donner sa valeur si je la lui demande, et augmenter sa valeur de 1 si l'événement qu'il est chargé de compter survient :

```
(define compteur    (let ((valeur 0))       (lambda (message)          (case message             ((val)
valeur)             ((inc) (set! valeur (+ 1 valeur)))))))
```

– C'est tout. Nous avons créé un objet `compteur` qui contient une variable et deux méthodes qui permettent d'y accéder, l'une pour connaître sa valeur que nous invoquerons ainsi (le message est ici un symbole, voyez la syntaxe de `case` ici) :

```
(compteur 'val)
```

– l'autre pour modifier sa valeur :

```
(compteur 'inc)
```

– si nous voulons utiliser notre nouveau `compteur` pour comptabiliser les appels à notre procédure `carre` nous pourrons procéder ainsi :

```
(define (carre x)    (compteur 'inc)    (* x x))
```

Ce procédé qui consiste à encapsuler une variable et les méthodes qui permettent de la manipuler dans un objet unique est un premier pas (encore très rudimentaire) vers la programmation objet.

Post-scriptum :

Vous pouvez ajouter une troisième méthode que vous pourrez appeler `reset` pour remettre le compteur à zéro.