CLOS-TD2
Méta-programmation et aspects ouverts

Stéphane Ducasse
ESSI3-DESS-DEA

December 19, 1995

Avant toute définition d'une métaclasse, vous ajouterez les lignes suivantes en tête de votre fichier.

```
(defmethod PCL::validate-superclass ((c PCL::class) (new-super PCL::class)) t)
```

Bien qu'une classe abstraite puisse être vue comme une classe ayant un nombre limité d'instances éqale à zéro, durant ce TD nous ferons une différence. Aussi ferez vous chacun des exercices proposés de manière autonome.

1 Mixins

Afin d'implémenter correctement le concept de *mixins* comme il vous a été expliqué en cours, vous allez définir une métaclasse dont les instances ne peuvent pas être instanciées. Cette métaclasse représentera le comportement des classes abstraites et pourra donc être ré-utilisée lors d'autres problèmes.

Définir la classe **Abstract** qui est instance de **standard-class**. Cette classe interdira la création d'instances de ses instances.

Pour tester la puissance des mixins, on réutilisera par exemple la hièrarchie définie lors du premier TD.

```
(defclass Mixin-Dog ()
        ((dog-name :accessor dog-name :initarg :dog :initform "milou"))
        :metaclass Abstract)

(defmethod presentation ((md Mixin-dog))
  (call-next-method)
  (format t "~% avec ~s~%" (dog-name md)))

(defclass thesard-avec-chien (Mixin-dog thesard)
        ()
        (:metaclass standard-class))

;; (setq p1 (make-instance 'mixin-dog))
;; -> erreur


(defclass chercheur-with-dog (mixin-dog chercheur)
  ())

(setq e (make-instance 'chercheur-with-dog :lab 'inria :dog-name "filou"))
(presentation e)

-> ...
   avec filou
```

2 Nombre limité d'instances

Il s'agit simplement de définir des classes dont le nombre d'instances est limité. Si l'utilisateur demande la création d'une instance d'une classe C alors que le nombre d'instances maximum pour cette classe C est atteint, il y a un message d'erreur.

Définir la classe **limited-instance** qui hérite de **standard-class**, elle aura deux slots : nb-max-instances et nb-instances. Attention, pour des problèmes d'initialisation, on stockera une liste contenant le nombre d'instances maximales pouvant être créées. Ceci est du au comportement de CLOS : dans la définition ci-dessous `(:nb-max-instances 3)`, le slot `nb-max-instances` sera affecté avec la valeur `'(3)` et non `3`.

```
(defclass cochon ()
        ((nom :accessor nom :initarg :nom :initform "gnaf")
         (temperament :initarg :initarg :temperament : initform "faineant"))
        (:metaclass limited-instance)
        (:nb-max-instances 3))

;; cette  dernière ligne est équivalente à
;; (setf (nb-max-instances (find-class 'cochon)) '(3))

(setq gnaf (make-instance 'cochon :nom gnaf :temperament "fainéant"))
(setq gnif (make-instance 'cochon :nom gnif :temperament "normal"))
(setq gnouf (make-instance 'cochon :nom gnouf :temperament "travailleur"))
```

3 Quelques statistiques

Comme vous l'avez vu en cours, il est possible de spécialiser le comportement des fonctions génériques : recherche et tri des méthodes applicables, application de la fonction générique invoquée, des méthodes et d'une méthode. Nous allons maintenant exploré cet aspect des choses. Nous voulons faire des statistiques afin de savoir combien de fois sont appelées certaines fonctions génériques dans un soucis par exemple d'amélioration.

Pour cela, nous devons spécialiser la fonction générique responsable de l'application de fonctions génériques : la fonction `apply-generic-function`. Or en CLOS, pour des raisons d'optimisations ce protocole est décomposé en de la manière suivante:

```
        (apply-generic-function gf args) est équivalent à

        (apply (compute-discriminating-function gf) args).
```

Cette décomposition permet ainsi aux implémenteurs de CLOS de ne calculer la fonction générique que lors du premier appel et de stocker ce résultat afin de s'en reservir lors des autres appels. Cette façon de faire est nommée *mémoization*. Ce qu'il faut noter à propos de `compute-discriminating-function` est que cette fonction est une fonction qui rend une fonction. Pour illustrer notre propos voici le code nécesssaire pour compter le appels d'une fonction générique.

```
(defclass Counting-Gen (STANDARD-GENERIC-FUNCTION)
  ((howmany :initform 0 :accessor howmany))
  (:metaclass FUNCALLABLE-STANDARD-CLASS))

(defmethod compute-discriminating-function ((cgf Counting-Gen))
 (let ((normal-dfun (call-next-method)))
   #'(lambda (&rest args)
        (incf (howmany cgf))
        (format t " appel numero ~s~%" (howmany cgf))
```

```
        (apply normal-dfun args)))))

;;; test
(defgeneric yeap (l)  (:generic-function-class counting-gen))

(defmethod yeap (l)  (format t "yeap on Top~%"))

(defclass lulu () ())

(defmethod yeap ((l lulu))
  (call-next-method)
  (format t "yeap it is lulu~%"))

(setq l (make-instance 'lulu))
(yeap l)

(yeap 23)
```

Une petite astuce voici la fonction qui étatnt un symbol permet de récupérer la fonction générique à laquelle le symbole est associé. Vous porrez vous en servir pour remettre à zero les compteurs de vos fonctions.

```
(defun give-gf (name)
       (pcl::prototype-of-generic-function name))

(howmany (give-gf 'yeap))
(setf (howmany (give-gf 'yeap)) 0)
```

4 Des fonctions génériques avec trace

Maintenant c'est à vous de faire ! Nous voudrions des fonctions génériques que l'on puisse tracer à volonté. Pour cela, vous allez définir une classe **Traceable-Gen** ayant un slot **tracing** initiliasé par défaut à nil ayant comme accesseur `tracing?`. Vous définirez la fonction générique `compute-discriminating-function` pour avoir le résultat voulu. Vous définirez deux fonctions ou méthodes qui permettront de basculer l'état de la fonction générique de traçable à normale.

```
(defgeneric new-yeap (l)  (:generic-function-class traceable-gen))

(defmethod new-yeap (l)  (format t "yeap on Top~%"))

(defclass lulu () ())
(defmethod new-yeap ((l lulu))
  (call-next-method)
  (format t "yeap it is lulu~%"))

(setq l (make-instance 'lulu))
(new-yeap l)

(rend-tracable (give-gf 'new-yeap))
(new-yeap l)
;; Entrant dans la fonction new-yeap avec les args ...
  ....
;; Sortant de la fonction new-yeap avec le résultat ...
```

```
(rend-normal (give-gf 'new-yeap))
```