



Protocoles Standards



Méta-Classes



Redéfinitions

Approches Objet de la Programmation

～ CLOS Avancé ～

Didier Verna

EPITA / LRDE

didier@lrde.epita.fr



lrde/~didier



@didierverna



didier.verna



google+



in/didierverna

Plan

Protocoles Standards

- Fonctions Génériques
- Instanciation
- eq1 Specializers

Méta-Classes

- Principe
- Applications

Redéfinitions



Plan

Protocoles Standards

Fonctions Génériques

Instanciation

eq1 Specializers

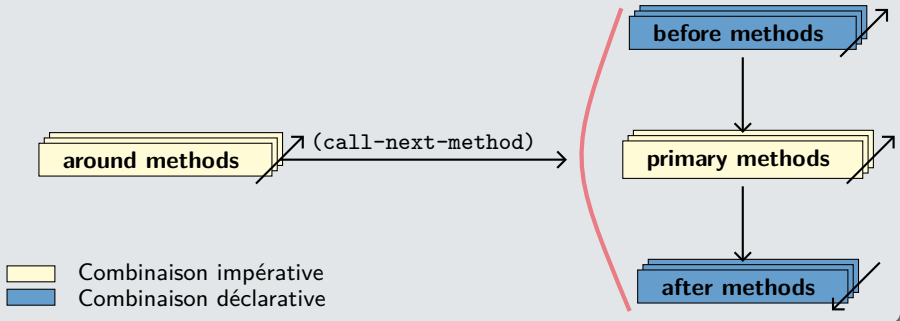
Méta-Classes

Redéfinitions



Fonctions Génériques

Protocole standard



► Rôles des méthodes :

- ▶ primary : travail principal, retour de valeur
- ▶ before / after : effets de bords
- ▶ around : travail auxiliaire

Application : Accumulation Déclarative

```
(defmethod hello ((human human))
  (format t "Hello! I'm ~A, ~A, ~A.~%"
    (name human)
    (gender human)
    (age human))
  (values))

(defmethod hello :after ((employee employee))
  (call-next-method)
  (format t "Working at ~A for ~A euros, started at the age of ~A.~%"
    (company employee)
    (salary employee)
    (hiring-age employee))
  (values))
```



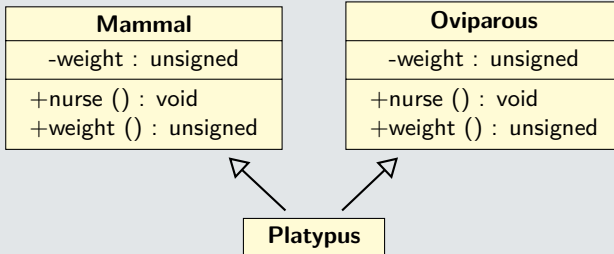
Combinaisons de Méthodes

- ▶ Utilisation d'une ou plusieurs méthodes (primaires) applicables
- ▶ Spécification déclarative plutôt qu'impérative
- ▶ Combinaison standard (par défaut) : méthode la plus spécifique + before, after et around
- ▶ Combinaisons disponibles (built-in)
 - ▶ +, min, max, list, nconc, append, and, or, progn
 - ▶ Ni before, ni after
 - ▶ call-next-method interdit
- ▶ Programmation de nouvelles combinaisons
 - ▶ define-method-combination (macro)
 - ▶ Deux formes
- ▶ Voir aussi [Verna, 2018]



Application

Rappel



- ▶ `weight` : problème traité (fusion des définitions de slot)
- ▶ `nurse` ?

Application

La combinaison progn

```
(defgeneric nurse (animal)
  (:method-combination progn))

(defmethod nurse progn ((mammal mammal))
  (when (next-method-p) (call-next-method))
  (format t "I milk my children.~%")
  (values))

(defmethod nurse progn ((oviparous oviparous))
  (when (next-method-p) (call-next-method))
  (format t "I brood my eggs.~%")
  (values))
```

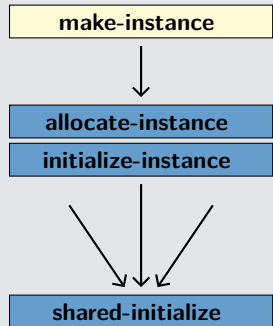
► **Note** : pas de méthode spécifique aux platypus requise




Instanciation

- ▶ `make-instance` :
calcul et validité des options
d'initialisation
- ▶ `allocate-instance` :
allocation d'un objet non initialisé
- ▶ `initialize-instance` :
appel à `shared-initialize`
- ▶ `shared-initialize` :
initialisation des slots
- ▶ **Note** : fonctions génériques
spécialisables

Protocole



 Couche 2
 Couche 3

Application : Vérification Post-Initialisation

```
(defmethod initialize-instance :after ((human human) &key)
  (slot-value human 'name)
  (slot-value human 'gender)
  (incf (slot-value human 'population))) ;; bonus

(defmethod initialize-instance :after ((employee employee) &key)
  (slot-value employee 'company)
  (slot-value employee 'salary)
  (slot-value employee 'hiring-year))
```

► Voir également :

- slot-boundp (fonction)
- slot-unbound (fonction générique)
- unbound-slot (condition)

► Remarque : vérification dynamique (run-time)



eq1 Specializers

- Spécialisation sur des objets particuliers

Example : pattern matching

```
(defgeneric product (x y)
  (:method (x (y (eq1 0))) 0)
  (:method ((x (eq1 0)) y) 0)
  (:method (x y)
    (* x y)))
```



eq1 Specializers

- Les objets peuvent être des classes

Exemple : une classe singleton

```
(defclass singleton (#/.../#)
  (#/.../#))
(let (instance)
  (defmethod make-instance
    ((class (eq1 (find-class 'singleton))) &key)
    (or instance (setf instance (call-next-method))))))
```



eq1 Specializers

- Les objets peuvent être des (noms de) classes

Exemple : un faux slot partagé

```
(defclass human (#/.../#)
  (#/.../#))
(let ((population 0))
  (defmethod initialize-instance :after ((human human) &key)
    (incf population))
  (defgeneric census (obj)
    (:method ((human human)) population)
    (:method ((class (eq1 (find-class 'human)))) population)
    (:method ((symbol (eq1 'human))) population)))
```



Plan

Protocoles Standards

Méta-Classes

Principe

Applications

Redéfinitions



Méta-Classes Utilisateur

► Principe :

- Spécialiser le comportement d'un *ensemble* de classes
- Classe de classes (méta-classe)

► Processus :

1. Créer une méta-classe (sous-classer `standard-class`)
2. Macrologie (couche 1)
3. Déclarer sa validité (couche 3 / MOP)
4. Spécialiser (option `:metaclass` de `defclass`)



Application : Classes Abstraites

```
(defclass abstract-class (standard-class) ())

(defmethod validate-superclass
  ((class abstract-class) (superclass standard-class))
  t)

(defmethod validate-superclass
  ((class standard-class) (superclass abstract-class))
  t)

(defmethod make-instance :before ((class abstract-class) &key)
  (error "~A is an abstract class." (class-name class)))

;; (defclass foobar (#...#)
;;   (#...#)
;;   (:metaclass abstract-class))
```



Application : Classes Finales

```
(defclass final-class (standard-class) ())

(defmethod validate-superclass
  ((class final-class) (superclass standard-class))
  t)

(defmethod validate-superclass
  ((class standard-class) (superclass final-class))
  nil)

(defmethod validate-superclass
  ((class final-class) (superclass final-class))
  nil)

;; (defclass foobar (#...#)
;;   (#...#)
;;   (:metaclass final-class))
```



Application : Classes Singleton

```
(defclass singleton-class (standard-class)
  ((instance :initform nil)))

(defmethod validate-superclass
  ((class singleton-class) (superclass standard-class))
  t)

(defmethod validate-superclass
  ((class standard-class) (superclass singleton-class))
  nil)

(defmethod make-instance ((singleton-class singleton-class) &key)
  (or (slot-value singleton-class 'instance)
      (setf (slot-value singleton-class 'instance)
            (call-next-method))))

;; (defclass foobar (#...#)
;;   (#...#)
;;   (:metaclass singleton-class))
```



Application : Classes Comptables

```
(defclass counting-class (standard-class)
  ((counter :initform 0)))

(defmethod validate-superclass
  ((class counting-class) (superclass standard-class))
  t)

(defmethod validate-superclass
  ((class standard-class) (superclass counting-class))
  t)

(defmethod make-instance :after ((counting-class counting-class) &key)
  (incf (slot-value counting-class 'counter)))

;; (defclass foobar (#...#)
;;   (#...#)
;;   (:metaclass counting-class))
```



Application : un (Autre) Faux Slot Partagé

```
(defclass population-class (standard-class)
  ((population :initform 0 :accessor population)))

(defmethod validate-superclass
  ((class population-class) (superclass standard-class))
  t)

(defmethod validate-superclass
  ((class standard-class) (superclass population-class))
  t)

(defclass human () (#/.../#) (:metaclass population-class))
(defmethod initialize-instance :after ((human human) &key)
  (incf (population (find-class 'human))))

(defgeneric census (object)
  (:method ((class population-class) (population class))
    (:method ((symbol (eql 'human))) (population (find-class 'human)))
    (:method ((human human) (population (find-class 'human)))
```



Plan

Protocoles Standards

Méta-Classes

Redéfinitions



Redéfinitions

► Rappels (Langages statiques / MOB1) :

- Classe \Leftrightarrow Type
Fixe, connue à la compilation
- Objet \Leftrightarrow Valeur
Variables de type fixe, connu à la compilation

► Nouveau contexte :

- Langage dynamique
Les valeurs (\neq variables) portent leur propre information de type
- MOP
Chaque composant du système objet est lui-même un (méta-)objet, donc modifiable

► Nouvelles fonctionnalités :

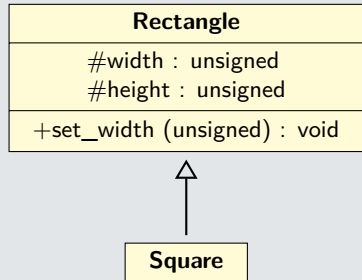
- Ajout / Suppression / Modification de classes, méthodes, fonctions génériques
- Changement de classe d'une instance



Rappel : Héritage par Restriction

- ▶ Le problème cercle/ellipse (carré/rectangle)
- ▶ Un carré est un rectangle...
- ▶ ...mais avec des contraintes statiques...
- ▶ ...et dynamiques
- ▶ Programmation Différentielle :
 - ▶ Hériter de manière additive et non restrictive
 - ▶ Problème surtout lié à la mutation
- ▶ Cf. le principe de substitution de Liskov

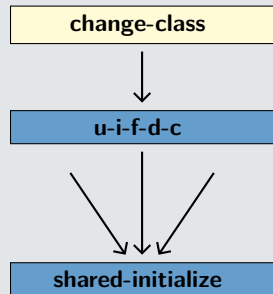
Exemple UML



Changement de Classe

- ▶ `change-class` :
modification destructive d'un objet
(sans changement d'identité)
- ▶ `update-instance-for-different-class` :
validité des options d'initialisation
- ▶ `shared-initialize` :
initialisation des slots
- ▶ **Note** : fonctions génériques
spécialisables
- ▶ **Attention** : bien choisir son
moment !

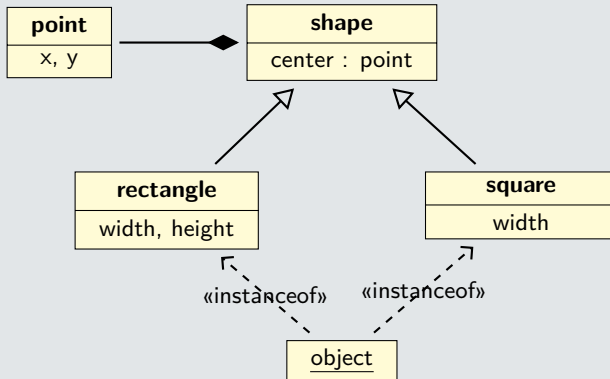
Protocole



 Couche 2
 Couche 3

Application : Carrés / Rectangles

Pseudo-UML



Côté Rectangle

```
(defun make-rectangle (width height)
  (if (= width height)
      (make-instance 'square :width width)
      (make-instance 'rectangle :width width :height height)))

(defmethod (setf width) :after (width (rectangle rectangle))
  (when (= (width rectangle) (height rectangle))
    (change-class rectangle 'square)))

(defmethod (setf height) :after (height (rectangle rectangle))
  (when (= (width rectangle) (height rectangle))
    (change-class rectangle 'square)))
```



Côté Carré

```
(defclass square (shape)
  ((width :initarg :width :reader width :reader height :accessor side)))

(defmethod (setf width) (width (square square))
  (let ((side (side square)))
    (unless (= width side)
      (change-class square 'rectangle :width width :height side)))
  width)

(defmethod (setf height) (height (square square))
  (unless (= height (side square))
    (change-class square 'rectangle :height height))
  height)
```



Plan

Bibliographie

Bibliographie



Didier Verna

Method Combinators

ELS, 2018