Object-Oriented Approaches to Programming

Advanced CLOS

Didier Verna

didier@didierverna.net


didierverna.net


@didierverna


didier.verna


in/didierverna

Build 2026-06-03 11:45:59+02:00

Outline

Plan

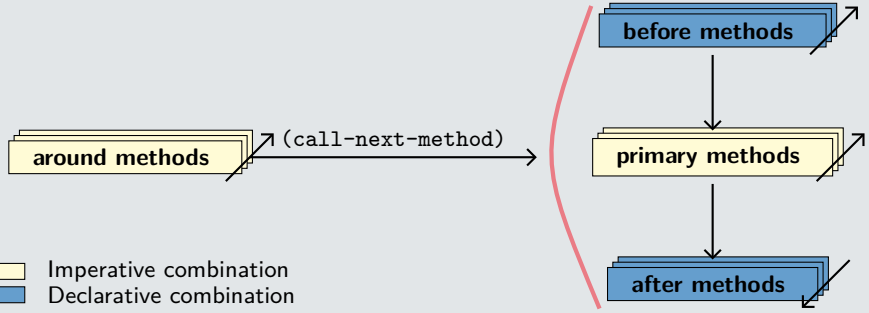
Plan

Method Combinations

- Use of one or several applicable (primary) methods
- Declarative (rather than imperative) specification
- Standard combination (default): most specific method
  + `before`, `after` et `around`
- Available combinations (built-in)
  - `+`, `min`, `max`, `list`, `nconc`, `append`, `and`, `or`, `progn`
  - No `before`, no `after`
  - `call-next-method` prohibited
- Programming new combinations
  - `define-method-combination` (macro)
  - Two forms
- See also [Verna, 2018], [Verna, 2023], [Verna, 2026]

Regular Generic Functions

Standard Protocol



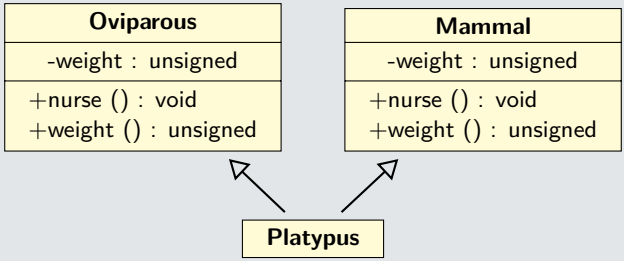
- **Method roles:**
  - `primary`: main work, return value
  - `before` / `after`: side effects
  - `around`: auxiliary work

Application I: Declarative Accumulation

```
(defmethod hello ((human human))
  (format t "Hello! I'm ~A, ~Am, ~Ayo.~%"
    (name human)
    (size 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))
```

Application II

Reminder



- `weight`: problem already solved (slot definitions fusion)
- `nurse` ?

Application II

The progn combination

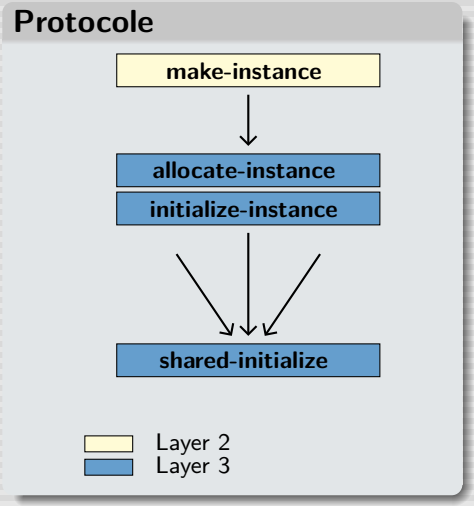
```
(defgeneric nurse (animal)
  (:method-combination progn))

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

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

- **Note:** no platypus-specific method required

Plan

Instantiation

- `make-instance`:
  initialization options computation and validity checks
- `allocate-instance`:
  uninitialized object allocation
- `initialize-instance`:
  call to `shared-initialize`
- `shared-initialize`:
  slots initialization
- **Note:** generic functions *specializable*



Application: Post-Initialization Checks

```
(defmethod initialize-instance :after ((human human) &key)
  (slot-value human 'name)
  (slot-value human 'size)
  (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))
```

- **See also:**
  - `slot-boundp` (function)
  - `slot-unbound` (generic function)
  - `unbound-slot` (condition)
- **Remark:** dynamic (run-time) checks

Plan

`eql` Specializers

- Specialization on specific objects

Exemple: pattern matching

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

`eql` Specializers

- The objects may be classes

Example: a singleton class

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

eql Specializers

- The objects may be classes (names)

Example: a fake shared slot

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

Plan

Plan

User-Level Meta-Classes

- **Principle:**
  - Specialize the behavior of a *set* of classes
  - Class of classes (meta-class)
- **Process:**
  1. Create a meta-class (sub-class `standard-class`)
  2. Macrology (layer 1)
  3. Declare its validity (layer 3 / MOP)
  4. Specialize (`:metaclass` option to `defclass`)

Plan

Application: Abstract Classes

```
(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: Final Classes

```
(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: Singleton Classes

```
(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: Accounting Classes

```
(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: A(nother) Fake Shared Slot

```
(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

Redefinitions

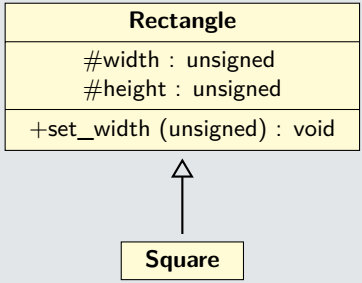
- **Reminder (Static Languages / AOP1):**
  - Class ⇔ Type
    Fixed, known at compile-time
  - Object ⇔ Value
    Variables of a fixed type, known at compile-time
- **New context:**
  - Dynamic Language
    Values (≠ variables) carry their own type information
  - MOP
    Every object system component is itself a (meta-)object, hence modifiable
- **New features:**
  - Addition / Suppression / Modification of classes, methods, generic functions
  - Change of an instance's class

Reminder: Inheritance by Restriction

- **The elliptic circle (square rectangle) problem**

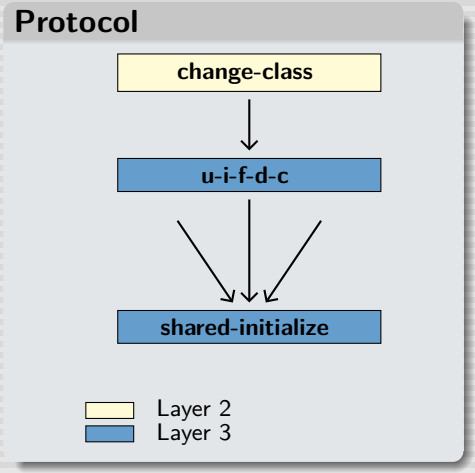
- A square is a rectangle...
- ...with additional constraints, static...
- ...and dynamic
- **Differential programming:**
  - Inherit additively, as opposed to restrictively
  - Problem tightly related to mutation
- Cf. the Liskov Substitution Principle

Example



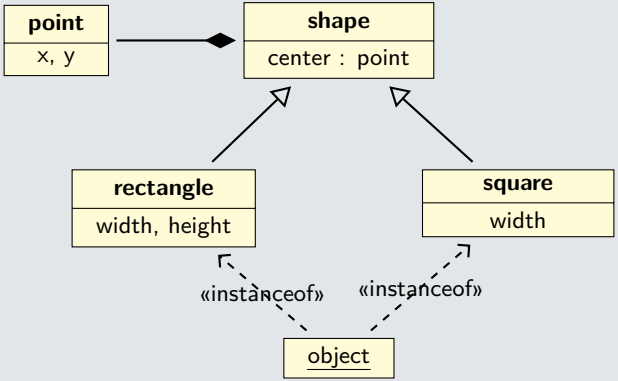
Class Change

- `change-class`:
  destructive modification of an objet (identity preserved)
- `update-instance-for-different-class`:
  initialization options validity checks
- `shared-initialize`:
  slots initialization
- **Note:** generic functions *specializable*
- **Caution:** choose your moment carefully!



Application: Squares / Rectangles

Pseudo-UML



Rectangle-Wise

```
(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)))
```

Square-Wise

```
(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

Bibliography

Bibliography

Didier Verna
Method Combinators
ELS, 2018

Didier Verna
A MOP-Based Implementation for Method Combinations
ELS, 2023

Didier Verna
An Update on the Method Combinations MOP
ELS, 2026