Object-Oriented Approaches to Programming

CLOS: Common Lisp Object System

Didier Verna

EPITA / LRE

didier@lrde.epita.fr

lrde/~didier

@didierverna

didier.verna

in/didierverna

Build 2025-05-21 19:20:01+02:00

Outline

Plan

Origins

- Experimentation since the 70's
  - Smalltalk, original ideas, *etc.*
- 1986: ACM Lisp and Functional Programming Conference
  - Informal group for standardizing an object system
    Strong pressure for standardizing the whole language
- X3J13 committee for Common Lisp standardization
  - ANSI standard X3.226:1994 (R1999)
- Result: a "best-of" from ideas / systems of that time
  - [New] Flavors (Symbolics Inc., MIT Lisp Machines), *Dave Moon & Sonya Keene*
  - [Portable] Common Loops (Xerox PARC, Interlisp-D), *Daniel Bobrow & Gregor Kiczales*
  - Lucid, *Dick Gabriel & Linda DeMichiel*

Characteristics I

- Stratified, flexible
  - API (syntactic) $\Longrightarrow$ API (fonctional) $\Longrightarrow$ Implementation
- Generic functions ($\neq$ message passing)
  - Message passing ill-suited to n-ary operations
  - Multi-methods
  - Natural extension to classical functions
- Multiple inheritance
  - Linearized class precedence system
- Method combinations
  - Multiple / simple invocation
  - Several standard ones, programmable

Characteristics II

- Higher order (MOP)
  - Meta-objects: generic functions, classes, *etc.*
  - 1st class: anonymous manipulation, *etc.*
- Dynamic typing
  - It's Lisp !
- No encapsulation or protection mechanism
  - Orthogonal to object-oriented programming
  - *Cf.* accessors and packages

Plan

Plan

Classes

The `human` class

```
(defclass human () (name size birth-year))
```

- **Remarks**
  - Functional definition (`defclass` is a macro)
  - Dynamic life cycle (both a type and an object)
  - Dynamic typing
  - No member method (*cf.* generic functions)
  - No protection mechanism
  - No abstract / final classes, *etc.*

Instantiation: Allocation

General mechanism

```
(make-instance 'human)
```

- **Remarks**
  - Function instantiation ($\neq$ constructor)
  - Function common to all classes
  - Reminder: garbage-collector $\Longrightarrow$ no destructor
- **Problem:** slot initialization

Instantiation: Initialization I

Initialization arguments

```
(defclass human ()
  ((name :initarg :name)
   (size :initarg :size)
   (birth-year :initarg :birth-year)))

(make-instance 'human
  :name "Alain Térieur" :size 1.80 :birth-year 1970)
```

- **Problem:** initialization by default

Instantiation: Initialization II

Initialization values

```
(defclass human ()
  ((name :initarg :name)
   (size :initarg :size)
   (birth-year :initarg :birth-year :initform 1970)))

(make-instance 'human :name "Alain Térieur" :size 1.80)
```

- **Problem:** mandatory initialization

Instantiation: Initialization III

Instantiation function

```
(defun make-human (name size &rest keys &key birth-year)
  (apply #'make-instance 'human :name name :size size
                                keys))

;; (make-human "Alain Térieur" 1.80)
;; (make-human "Alex Térieur" 1.80 :birth-year 1939)
```

- **Remarks**
  - `make-<class>` is conventional
  - Prefer *keywords* over optional parameters
  - Call semantics $\neq$ Overloading

Plan

Information Scope

Local *vs.* shared slots

```
(defclass human ()
  ((population :allocation :class :initform 0)
   (name :initarg :name)
   (size :initarg :size)
   (birth-year :initarg :birth-year :initform 1970))
```

- **Remarks**
  - By default: `:allocation :instance`
  - No standard access to shared slots through classes
  - No direct equivalent to class (static) methods

Information Accessibility I

General mechanism

```
(slot-value alain 'birth-year)
```

- **Remarks**
  - Functional access ($\neq$ syntactic)
  - Function common to all classes
  - No protection mechanism (little sense)
    Not even packages, no friendship concept, etc.
- **Problem:** lack of abstraction

Information Accessibility II

Accessors

```
(defclass human ()
  ((name :initarg :name :reader name :writer rename)
   (size :initarg :size :accessor size)
   (birth-year :initarg :birth-year :initform 1970 :reader birth-year)))

(name alain) ;; => "Alain Térieur"
(rename "Alain Verse" alain) ;; => "Alain Verse"

(size alain) ;; => 1.80
(setf (size alain) 1.78) ;; => 1.78

(birth-year alain) ;; => 1970
(setf (birth-year alain) 1971) ;; error
```

- **Remark:** Automatic generation of (generic) accessor functions

Outside Class Behavior

Example

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

- **Remarks**
  - Traditional systems: methods
  - Here: simple functions (generic ones later on)
  - Generic behavior nevertheless (dynamic typing)

Plan

Classes, Objects, Types

- Strong type / class integration
  - Single hierarchy: root class `t` (`class-of`)
  - Class / type correspondance (`type-of`, `typep`)
  - Native type / class correspondance
    Specialization possible, instantiation forbidden
- Note: sub-classing $\Longleftrightarrow$ sub-typing (`subtypep`)
  - *Cf.* traditional systems, although more robust
- MOP (total reflexivity: introspection / intercession)
  - Classes are objects (1st class type system)
  - Meta-class: class of a class
  - `standard-class`: class of all user classes (among others)
    Class of aggregative classes, circularity

Plan

Plan

Reminder: Relations Between Classes

- **Copy-Paste:** evil!
- **Aggregation:** “set / parts” relationship
- **Composition:** stronger aggregation
- **Inheritance:** implicit inclusion
  Structural and behavioral inheritance

Inheritance

```
(defclass employee (human)
  ((company :initarg :company :reader company)
   (salary :initarg :salary :accessor salary)
   (hiring-year :initarg hiring-year)))
```

Plan

Specificities

- **Implicit inheritance:**
  - Unique class hierarchy
  - `user-class` $\longrightarrow$ `...` $\longrightarrow$ `standard-object` $\longrightarrow$ `t`
  - No slot, *cf.* `print-object`, *etc.*
- **Slot inheritance:**
  - $\neq$ traditional systems
  - A unique slot (no ambiguity)
- **Slot options inheritance:**
  - initargs, initforms, *etc.*
  - Modalities may depend on the option
- **Method inheritance:**
  - Sub-classing $\Longleftrightarrow$ sub-typing
- **Multiple inheritance**

Instantiation in the Presence of Inheritance

Exemple

```
(defun make-employee (name size company salary hiring-year
                      &rest keys &key birth-year)
  (let ((employee (apply #'make-instance 'employee
                         :name name :size size
                         :company company :salary salary
                         :hiring-year hiring-year
                         keys)))
    (incf (slot-value employee 'population))
    employee))
```

- **Remarks**
  - One single entry point (`make-instance`)
  - No constructor chain

Plan

Problems Related to Inheritance

- **Persisting:**
  - Inheritance *vs.* Instantiation (“is a”)
    Class-based system
  - Inheritance ambivalence (interface / implementation)
    Sub-classing $\Longleftrightarrow$ *sub-typing*
- **Alternatives:**
  - Inheritance by restriction / Differential Programming
    Cf. next chapter: dynamic aspects of CLOS
  - Multiple / diamond inheritance
    - Definition merging / $\neq$ packages
    - *Cf. next chapter: method combinations*

Plan

Plan

Reminder: 3 Kinds of Polymorphism

- **Static Polymorphism**
  1. **Overloading**
     - Types: nonsensical in a dynamic language
     - Cardinality: too ambiguous / complicated
     - Replaced with a richer call semantics
  2. **Masking**
     - Specific to member methods
     - Nonsensical with generic functions
- **Dynamic polymorphism**
  3 **Rewriting**
     - Generic functions
     - Methodes

Plan

Generic Functions

The `translate` function

```
(defgeneric translate (object x &optional y))
```

- **Remarks**
  - Functional definition (`defgeneric` is a macro)
  - Dynamic life cycle (meta-object)
  - Dynamic typing
  - Interface only ($\neq$ regular function)
  - Optional declaration
  - Accessors are generic functions
  - Behavior identical to regular functions
    Same call syntax, function-cell, anonymous generic functions, etc.

Methods

A `translate` method

```
(defmethod translate ((circle circle) x &optional (y 0))
  (translate (center circle) x y))
```

- **Remarks**
  - Functional definition (`defmethod` is a macro)
  - Dynamic life cycle (meta-object)
  - Dynamic typing
  - Method = one specific implementation
  - Specialization on mandatory arguments
  - Multi-methods (specialization on several arguments possible)
  - Default method = not specialized (class `t`)
  - Methods are anonymous and not executable

Methods Chaining

call-next-method

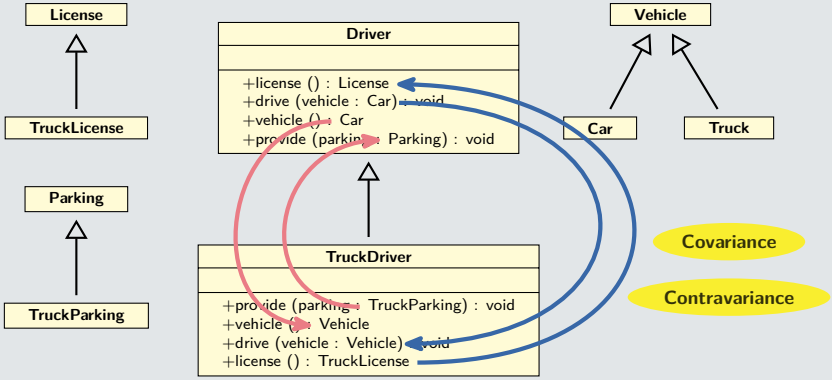
```
(defgeneric hello (object))

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

(defmethod hello ((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))
```

- **Remark:** sorted list of applicable methods

Plan

(Sub-)Class / (Sub-)Type Relationship

Reminder



Covariance / Contravariance

- **Everything is expressible** ($\neq$ AOP1)
  Expressible $\Rightarrow$ *compilable*
  - Contravariance on return values
  - Covariance on arguments
  - Why: dynamic typing / external methods
- **Not everything might make sense** ($=$ AOP1)
  - Contravariance on return values
    - Potential failure
    - Example: `(drive DRIVER (vehicle TRUCK-DRIVER))`
  - Covariance on argument
    - Potentially ignored error
    - Exemple: `(offer TRUCK-DRIVER PARKING)`
- $\Rightarrow$ Dynamic checks thanks to new (multi-)methods

Plan

Bibliography

Bibliography

Linda G. DeMichiel and Richard P. Gabriel
The Common Lisp Object System: An Overview
ECOOP, 1987

Sonja E. Keene
Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS
Addison-Wesley, 1989

Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales and David A. Moon.
Common Lisp Object System specification
ACM SIGPLAN Notices, 1988.