Object-Oriented Approaches to Programming

Case Study: Binary Methods

Didier Verna

didier@didierverna.net


didierverna.net


@didierverna


didier.verna


in/didierverna

Build 2025-06-19 09:00:25+02:00

Plan

Plan

Introduction

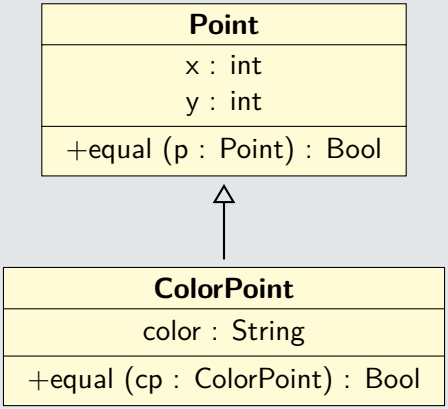
- **Binary Operation:** 2 arguments of the same type
  Arithmetics, ordering relations ($=, +, >$ etc.)
- **OO Programming:** 2 objects of the same class
  Benefit from polymorphism
- Hence the expression “binary method”
- **[Bruce et al., 1995]:**
  - Problematic concept in traditional OO approach
  - Type / class relation in the context of inheritance

Plan

Plan

Context

The `Point` Hierarchy



C++: Attempt #1

```
class Point
{
  int x, y;

  bool equal (Point& p)
  { return x == p.x && y == p.y; }
};

class ColorPoint : public Point
{
  std::string color;

  bool equal (ColorPoint& cp)
  { return color == cp.color && Point::equal (cp); }
};
```

C++: Attempt #1

Fail

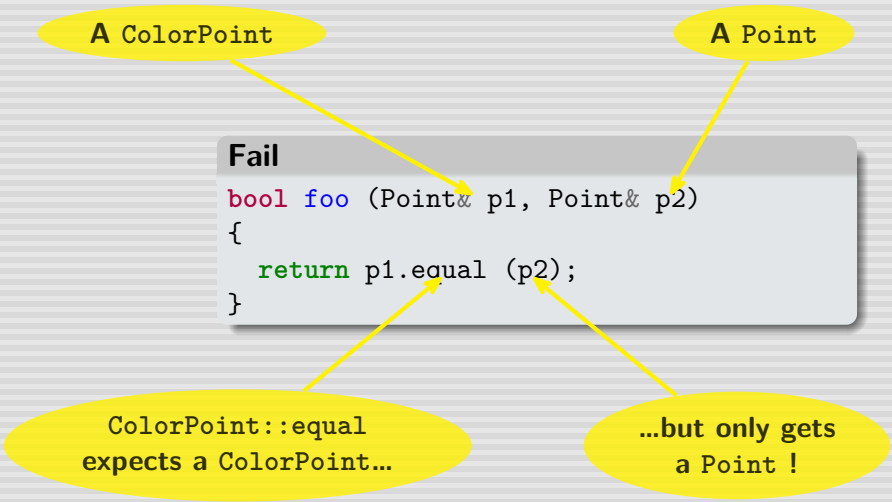
```
int main (int argc, char *argv[])
{
  Point& p1 = * new ColorPoint (1, 2, "red");
  Point& p2 = * new ColorPoint (1, 2, "green");

  std::cout << p1.equal (p2) << std::endl;
  // => True. #### Wrong !
}
```

- `ColorPoint::equal` *masks* `Point::equal` (statically)
- Only `Point::equal` is seen from the base class
- We need the exact version

C++: Attempt #2

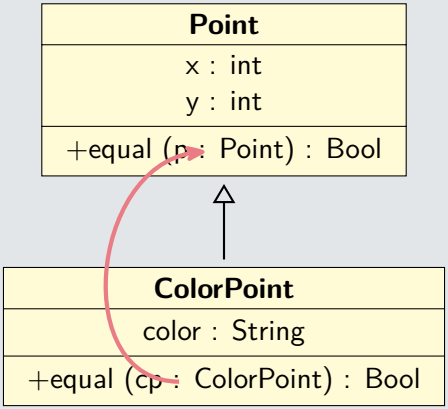
```
class Point
{
  int x, y;

  virtual bool equal (Point& p)
  { return x == p.x && y == p.y; }
};

class ColorPoint : public Point
{
  std::string color;

  virtual bool equal (ColorPoint& cp)
  { return color == cp.color && Point::equal (cp); }
};
```

C++: Attempt #2



Context

The `Point` Hierarchy



Reminder: Covariance / Contravariance

- **Theoretic Rule:**
  - *contravariance* on arguments
  - *covariance* on return values
- **In Practice:**
  - *invariance* on arguments
  - Ambiguities due to overloading
- **Remark:**
  - Eiffel allows covariance on arguments
  - Potential errors at run-time
- **Analyse:** [Castagna, 1995]
  - Lack of expressivity
    sub-typing (by sub-classing) $\neq$ *specialization*
  - Defect of the traditional OO model
    Lack of multi-methods

```
virtual bool equal (Point& p);
virtual bool equal (ColorPoint& cp);
```

CLOS: Attempt #1

```
(defgeneric point= (a b))

(defclass point ()
  ((x :initarg :x :reader point-x)
   (y :initarg :y :reader point-y)))

(defmethod point= ((a point) (b point))
  (and (= (point-x a) (point-x b))
       (= (point-y a) (point-y b))))

(defclass color-point (point)
  ((color :initarg :color :reader point-color)))

(defmethod point= ((a color-point) (b color-point))
  (and (string= (point-color a) (point-color b))
       (call-next-method)))
```


CLOS: Attempt #1

Success

```
(let ((p1 (make-point :x 1 :y 2))
      (p2 (make-point :x 1 :y 2))
      (cp1 (make-color-point :x 1 :y 2 :color "red"))
      (cp2 (make-color-point :x 1 :y 2 :color "green")))
  (values (point= p1 p2)
          (point= cp1 cp2)))
;; => (T NIL)
```

- Selection of the appropriate method based on the 2 arguments *Multiple dispatch*
- Call syntax more pleasing
  `p1.equal(p2)` or `p2.equal(p1)`?
- **“Binary Function”**

Plan

Reminder: Method Combinations

- **Avoid code duplication:**
  - **C++:** `Point::equal()`
  - **CLOS:** `(call-next-method)`
- **Applicable methods:**
  - All methods compatible with the arguments'classes
  - Sorted by decreasing specificity order
  - `call-next-method` executes the next method in that ordering
- **Method comnbinations:**
  - Call several applicable methods
    Not only the most specific
  - Predefined combinations: `and`, `or`, `progn` *etc.*
  - Programmable

The `and` Combination

```
(defgeneric point= (a b)
  (:method-combination and))

(defmethod point= and ((a point) (b point))
  (and (= (point-x a) (point-x b))
       (= (point-y a) (point-y b))))

(defmethod point= and ((a color-point) (b color-point))
  (and (call-next-method)
       (string= (point-color a) (point-color b))))
```

- **Note:** the CLOS dispatch is programmable

Plan

Non-Conforming Usage

Illegitimate Success

```
(let ((p (make-point :x 1 :y 2))
      (cp (make-color-point :x 1 :y 2 :color "red")))
  (point= p cp))
;; => T #### Wrong !
```

- (point= <point> <point>) is applicable
  A color-point is a point
- We want to avoid that situation

Plan

Introspection in CLOS

```
(assert (eq (class-of a) (class-of b)))
```

- **When to Check ?**
  - Every method: duplication
  - Last method: not efficient / not always possible
  - `:before`: unavailable outside the standard combination
  - New combination: not the place
  - `:around`: yes but…
- **Where to Check ?** (better question)
  - Nowhere near the code for `point=`
  - Related to the concept of binary function, not to `point=`
- Express the concept of binary function itself
  - *Binary functions are special kinds of generic functions*

Plan

A Binary Function Meta-Class

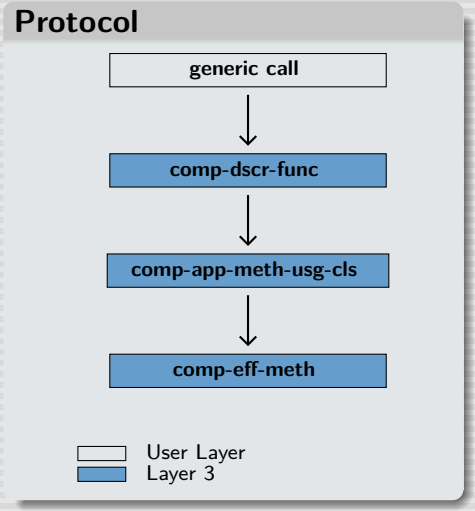
- Generic functions: `standard-generic-function`
- Function-objects: `funcallable-standard-class`

```
(defclass binary-function (standard-generic-function)
  ()
  (:metaclass funcallable-standard-class))

(defmacro defbinary (function-name lambda-list &rest options)
  `(defgeneric ,function-name ,lambda-list
    (:generic-function-class binary-function)
    ,@options))
```

- **Remark:** the concept mandates a specific `lambda-list`…

Generic Calls

- `compute-discriminating-function`: the generic function's actual function
- `compute-applicable-methods-using-classes`: figure out which methods are applicable based on the arguments'classes
- `compute-effective-method`: create the (combined) global method for these arguments
- **Note:** *specializable* generic functions



Application to Binary Functions

```
(defmethod compute-applicable-methods-using-classes
    :before ((binary-function binary-function) classes)
  (assert (= (length classes) 2))
  (assert (apply #'eq classes)))
```

Plan

Plan

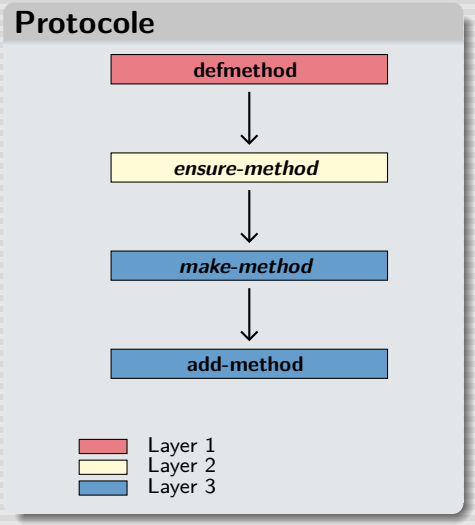
Non-Conforming Implementation

```
;; #### WRONG SPEC!!
(defmethod point= ((p1 point-3d) (p2 point-2d))
  #| ... |#)
```

- Previously: protect against illegal calls
- New question: protect against illegal implementations ?

Methods Definitions

- `ensure-method`:
  creation and addition of a new method
- `make-method`:
  creation of the method meta-object
- `add-method`:
  addition and update to the generic function
- **Note:** *specializable* generic functions



Application to Binary Functions

```
(defmethod add-method :before ((bf binary-function) method)
  (let ((method-specializers (method-specializers method)))
    (assert (= (length method-specializers) 2))
    (assert (apply #'eq method-specializers))))
```

- `method`: meta-object (Cf. `standard-method`)
- `method-specializers`: accessor

Plan

Missing Implementation

- Every sub-class of `point` must specialize `point=`
- Late checking: at generic call time
  preserve high dynamicity
- "Binary-Completeness": there exists a primary method applicable for every class in the hierarchy

```
(defmethod compute-applicable-methods-using-classes
    :around ((bf binary-function) classes)
  (multiple-value-bind (methods ok) (call-next-method)
    ;; 1. Check for binary completeness on METHODS
    ;; 2. If everything goes well, simply return
    ;;    what we got from CALL-NEXT-METHOD
    (values methods ok)))
```

Implementation

```
(loop :for class :in (class-precedence-list (car classes))
      :until (eq class (find-class 'standard-object))
      :do (assert
           (find-if
            (lambda (method)
              (let ((qualifiers (method-qualifiers method)))
                (and (equal (method-specializers method)
                            (list class class))
                     (not (member :around qualifiers))
                     (not (member :before qualifiers))
                     (not (member :after qualifiers)))))
            methods)))
```

- `class-precedence-list`: accessor
- `method-qualifiers`: accessor

Plan

Conclusion

- **Binary methods:** a very simple concept
  - Difficult to implement in traditional OO systems
  - Trivial with multi-methods
- **Thanks to CLOS:**
  - Usage validation
  - Implementation validation
- **Thanks to the MOP:**
  - Reified Concept
  - Extension of the core OO system

Plan

Bibliography

Bibliography

Bruce, K. B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S. F., Trifonov, V., Leavens, G. T., and Pierce, B. C. (1995)
On Binary Methods
Theory and Practice of Object Systems, 1(3):221–242

Castagna, G. (1995)
Covariance and Contravariance: Conflict Without a Cause
ACM Transactions on Programming Languages and Systems, 17(3):431–447

Verna, D. (2008)
Binary Methods Programming: the CLOS Perspective
Journal of Universal Computer Science, Vol. 14.20