CLOS solutions to binary methods

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/˜didier

March 21 2007

Introduction
What are binary methods?

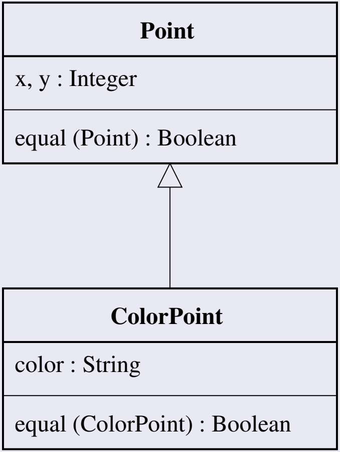
- **Binary Operation:** 2 arguments of the same type
  Examples: arithmetic / ordering relations ($=, +, >$ *etc.*)
- **OO Programming:** 2 *objects* of the same class
  Benefit from polymorphism *etc.*
- $\Rightarrow$ Hence the term **binary method**
- **However:**
  - problematic concept in traditional OO languages
  - type / class relationship in the context of inheritance

Table of contents

The test case

Used throughout this presentation

The `Point` class UML hierarchy



C++ implementation attempt #1

Details omitted

The C++ `Point` class hierarchy

```
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); }
};
```

But this doesn't work !

Overloading is not what we want

Looking through base class references

```
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` only *overloads* `Point::equal` in the derived class
- From the base class, only `Point::equal` is seen
- What we want is to use the definition from the exact class

C++ implementation attempt #2

Details omitted

The C++ `Point` class hierarchy

```
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); }
};
```

But this doesn't work either !

We still get overloading, still not what we want

The forbidden fruit

```
virtual bool equal (Point& p);
virtual bool equal (ColorPoint& cp); // #### Forbidden !
```

- Invariance required on virtual methods argument types
- **Worse:** here, the `virtual` keyword is *silently* ignored
- And we get an overloading behavior, as before
- **Why ?** To preserve type safety

Why the typing would be unsafe

And lead to errors at run-time

Example of run-time typing error

In fact, a ColorPoint

Just a Point

```
bool foo (Point& p1, Point& p2)
{
  return p1.equal (p2);
}
```

The ColorPoint implementation
expects a ColorPoint argument
(ex. accesses the color field)

But gets only a Point !

Constraints for type safety

covariance, contravariance… invariance

- When **subtyping a polymorphic method**, we must
  - **supertype** the arguments (*contravariance*)
  - **subtype** the return value (*covariance*)
- **Note:** Eiffel allows for arguments covariance
  - But this leads to possible run-time errors
- **Note:** C++ is even more constrained
  - The argument types must be *invariant*
- $\Rightarrow$ Implementing binary methods in traditional OO languages is
  - either impossible directly
  - or possible but unsafe

CLOS: the Common Lisp Object System
A different object model

- **C++ methods *vs.* CLOS generic functions**
  - C++ methods belong to classes
  - CLOS generic functions look like ordinary functions (outside classes)
- **C++ single dispatch *vs.* CLOS multi-methods**
  - C++ dispatch based on the first (hidden) argument type (`this`)
  - CLOS dispatch based on the type of *any* number of arguments
- **Note:** a CLOS “method” is a specialized implementation of a generic function

The CLOS `Point` class hierarchy

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

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


(defgeneric point= (a b))

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

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

How to use it ?

Just like ordinary function calls

Using the generic function

```
(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)
```

- Proper *method* selected based on *both* arguments (multiple dispatch)
- Function call syntax, more pleasant aesthetically (`p1.equal(p2)` or `p2.equal(p1)` ?)
- $\Rightarrow$ Hence the term **binary function**

Applicable methods

There are ore than one…

- **To 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` calls the next most specific applicable method
- **Method combinations:**
  - Ways of calling several (all) applicable methods (not just the most specific one)
  - Predefined method combinations: `and`, `or`, `progn` *etc.*
  - User definable

Using the `and` method combination

Comes in handy for the equality concept

The `and` method 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))
       )
  )
```

- ⇒ In CLOS, the generic dispatch is (re-)programmable

Binary methods could be misused

Can we protect against it ?

The `point=` function used incorrectly

```
(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 an applicable method (because a `color-point` *is* a `point`)
- ⇒ The code above is valid
- ⇒ And the error goes unnoticed

Introspection in CLOS

Inquiring the class of an object

Using the function `class-of`

```
(unless (eq (class-of a) (class-of b))
  (error "Objects not of the same class."))
```

- **When to perform the check ?** (w/o code duplication)
  - In the basic method: neither efficient, nor elegant
  - In a `before-method`: not available with the `and` method combination
  - In a user-defined method combination: not elegant
- **Where to perform the check ?** (a better question)
  - Nowhere near the code for `point=` !
  - Part of the binary function concept, not `point=`
- ⇒ We should implement the binary function **concept**
  - A specialized class of generic function?

The CLOS Meta-Object Protocol

aka the CLOS MOP

- **CLOS *itself* is object-oriented**
  - The CLOS MOP: a *de facto* implementation standard
  - The CLOS components (classes *etc.*) are (meta-)objects of some (meta-)classes
  - Generic functions are meta-objects of the `standard-generic-function` meta-class
- ⇒ We can subclass `standard-generic-function`

The `binary-function` meta-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))
```

Back to introspection
Hooking the check

- **Calling a generic function involves:**
  - Computing the list of applicable methods
  - Sorting and combining them
  - Calling the resulting *effective* method
- `compute-applicable-methods-using-classes`
  - Does as its name suggests
  - Based on the classes of the arguments
  - A good place to hook
- We can specialize it !
  - It is a generic function

Specializing the `c-a-m-u-c` generic function

```
(defmethod c-a-m-u-c :before ((bf binary-function) classes)
  (assert (equal (car classes) (cadr classes))))
```

Binary methods could be misimplemented

Can we protect against it ?

- We protected against calling
  `(point= <point> <color-point>)`
- Can we protect against *implementing* it ?
- `add-method`
  - Registers a new method (created with `defmethod`)
  - Is a generic function
  - Can be specialized

Specializing the `add-method` generic function

```
(defmethod add-method :before ((bf binary-function) method)
  (assert (apply #'equal (method-specializers method))))
```

Binary methods could be forgotten

Can we protect against it ?

- **Strong binary functions:**
  - Every subclass of `point` should specialize `point=`
  - Late checking: at generic function call time (preserve interactive development)
- **Binary completeness:**
  1. There is a specialization on the arguments' exact class
  2. There are specializations for all super-classes
- **Introspection:**
  - Binary completeness of the list of applicable methods
  - `c-a-m-u-c` returns this !

Hooking the check

```
(defmethod c-a-m-u-c ((bf binary-function) classes)
  (multiple-value-bind (methods ok) (call-next-method)
    ;; ...
    (values methods ok)))
```

Is there a bottommost specialization ?

Check #1

- classes = '(<exact> <exact>)
- method-specializers returns the arguments classes from the defmethod call
- ⇒ We should compare <exact> with the specialization of the first applicable method

Check #1

```lisp
(let* ((method (car methods))
       (class (car (method-specializers method))))
  (assert (equal (list class class) classes))
  ;; ...
  )
```

Are there specializations for all super-classes ?

Check #2

- `find-method` retrieves a generic function's method given a set of qualifiers / specializers
- `method-qualifiers` does as its name suggests
- `class-direct-superclasses` as well

Check #2

```
(labels ((check-binary-completeness (class)
           (find-method bf (method-qualifiers method)
                          (list class class))
           (dolist
               (cls (remove-if
                     #'(lambda (elt)
                         (eq elt (find-class
                                  'standard-object)))
                     (class-direct-superclasses class)))
             (check-binary-completeness cls))))
  (check-binary-completeness class))
```

Conclusion

- Binary methods problematic in traditional OOP
- Multi-methods as in CLOS remove the problem
- CLOS and the CLOS MOP let you support the concept:
  - make it available
  - ensure a correct usage
  - ensure a correct implementation
- **But the concept is implemented explicitly**
  - CLOS is not just an object system
  - CLOS is not even just a customizable object system

CLOS is an object system designed to let you program new object systems

Quesλions ?

Logo by Manfred Spiller