# Binary Methods Programming: the CLOS Perspective

Didier Verna

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

May 23 – ELS 2008

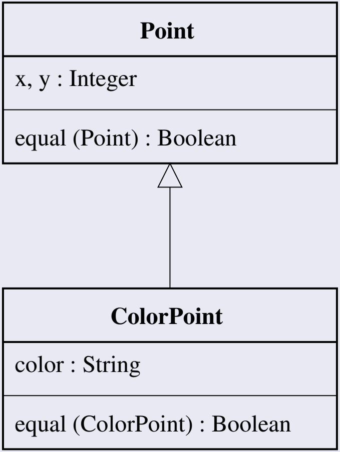
# Introduction

## 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:** [Bruce et al., 1995]
  - problematic concept in traditional OO languages
  - type / class relationship in the context of inheritance

# Table of contents

## The `Point` class 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`
- From the base class, only `Point::equal` is seen
- We want the definition from the *exact* class

# C++ implementation attempt #2

## Details still 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. . .

## Still got 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: `virtual` keyword *silently* ignored (overloading behavior, just as before)
- **Why?** To preserve static 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

## When subtyping a polymorphic method

- **The covariance / contravariance rule**
  - **supertype** the arguments (*contravariance*)
  - **subtype** the return value (*covariance*)
- **Note:** C++ is even more constrained
  - The argument types must be *invariant*
- **Note:** Eiffel allows for arguments covariance
  - But this leads to possible run-time errors
- **Analysis:** [Castagna, 1995].
  - *Lack of expressiveness*
    subtyping (by subclassing) $\neq$ specialization
  - *Object model defect*
    single dispatch (not the record-based model)

# CLOS: the Common Lisp Object System

## A different model

- **Class methods *vs.* Generic functions**
  - C++ methods belong to classes
  - CLOS generic functions look like ordinary functions (outside classes)
- **Single dispatch *vs.* Multi-methods**
  - C++ dispatch: the first (hidden) argument's type (`this`)
  - CLOS dispatch: the type of *any* number of arguments

# CLOS implementation

## No detail omitted

### 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)))

;; optional
(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)))
```

## 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)
```

- Method selection: *both* arguments (multiple dispatch)
- Function call syntax: more pleasant aesthetically (`p1.equal(p2)` or `p2.equal(p1)`?)
- ⇒ Hence the term **binary function**

# Applicable methods

There are more 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

```lisp
(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 the last method, the line `(and (call-next-method)` and the closing parenthesis after `(string= …)` are struck through on the slide.)

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

# Anatomy of a method

## A bit of vocabulary



*Method qualifiers* → `and`

*Method specializers* → `point`, `point`

```
defmethod point= and ((a point) (b point))
```

# 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* applicable (a color-point *is* a point)
- The code above is valid
- The error goes unnoticed

# Introspection in CLOS
## Inquiring the class of an object

### Using the function `class-of`

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

- **When to perform the check?**
  - In all methods: code duplication
  - In the basic method: not efficient
  - In a `before-method`: not available with the `and` method combination
  - In a user-defined method combination: not the place
- **Where to perform the check?** (a better question)
  - Nowhere near the code for `point=`
  - Part of the binary function concept, not `point=`
- $\Rightarrow$ We should implement the binary function **concept**
  - A specialized class of generic function?

- **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
- $\Rightarrow$ 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 (eq (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)
  - We can specialize it!
    - It is a generic function . . .

### Specializing the add-method generic function

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

# Binary methods could be forgotten

## Can we protect against it?



- Every subclass of `point` should specialize `point=`
- **Late checking:** at generic function call time (preserve interactive development)
- **Binary completeness check:**
  1. There is a specialization on the arguments' exact class
  2. There are specializations for all super-classes

**Hooking the check: `c-a-m-u-c` still the best candidate**

```
(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>)`
- `methods = '(appmeth1 appmeth2 ...)`
- ⇒ We should compare `<exact>` with the specializers of `appmeth1`
- `method-specializers` does as its name suggest

### Check #1

```
(let* ((method (car methods))
       (class (car (method-specializers method))))
  (assert (eq class (car 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**

# Articles



- 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.