Revisiting the Visitor: the “Just Do It” Pattern

Didier Verna

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

ACCU 2009 – Friday, April 24th

Introduction

Necessary literature

- **The GOF Book:** Design Patterns, Elements of Reusable Object-Oriented Software. *Gamma, Helm, Johnson, Vlissides.*
- **The POSA Book:** Pattern-Oriented Software Architecture. *Buschmann, Meunier, Rohnert, Sommerlad, Stal.*

- What is a software design pattern ?
  - Context (POSA)
  - Problem
  - Solution
  - Consequences (GOF)

A constatation

Peter Norvig (Object World, 1996)

About the GoF book:

> *16 of 23 patterns are either invisible or simpler [...] in Dylan or Lisp*

- **Peter Norvig is right**, so
  - is the GoF book (70%) wrong ?
  - are patterns (70%) useless ?

Some clues from the GOF book itself

> *Although design patterns describe object-oriented designs, they are based on **practical** solutions that have been implemented in **mainstream** object-oriented programming languages [. . .]*

> *Similarly, some of our patterns are supported directly by the less common object-oriented languages.*

- That's what people usually miss

Patterns descriptions / organizations

- **GoF:** Creational, Structural, Behavioral
  - usage-oriented
- **Posa:** Architectural, Design, Idioms
  - abstraction-oriented

Idioms according to Posa

> *An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language. [...] They address aspects of both design and implementation.*

- GoF's design patterns are closer to Posa's idioms

The risk: blind pattern application

POSA's advice:

> *[. . .] sometimes, an idiom that is useful for one programming language does not make sense into another.*

GOF's Visitor example:

> *Use the Visitor pattern when [. . .] many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.*

- But who said operations belong to classes ?

Table of contents

1 Visiting in C++

2 Visiting in Lisp

- Step 1: plain Lisp
- Step 2: brute force visiting
- Step 3: first class generic functions
- Step 4: mapping
- Step 5: generic mapping

3 Visiting with state

- Step 6: objects
- Step 7: lexical closures
- Step 8: dynamic visitation schemes

Visiting in C++

Problems:

- Original hierarchy R/O
- Abstract the visiting process away

Solution:

1. Equip original hierarchy for visits
   - A `Visitable` abstract class
   - An `accept` method in each visitable component
2. Write independent visitors
   - A `Visitor` abstract class
   - A `visit` method for each visitable component

Step 1: plain LISP

Classes

```
(defclass class (superclass1 superclass2 ...)
  ((slot :initform <form> :initarg :slot :accessor slot)
   ...)
  options...)

(make-instance 'class :slot <value> ...)
```

Generic functions, methods

```
(defgeneric func (arg1 arg2 ...)
  (:method ((arg1 class1) arg2 ...)
    body)
  options...)

(defmethod func ((arg1 class1) arg2 ...)
  body)
```

- Methods are *outside* the classes (ordinary function calls)
- Multiple dispatch (multi-methods)

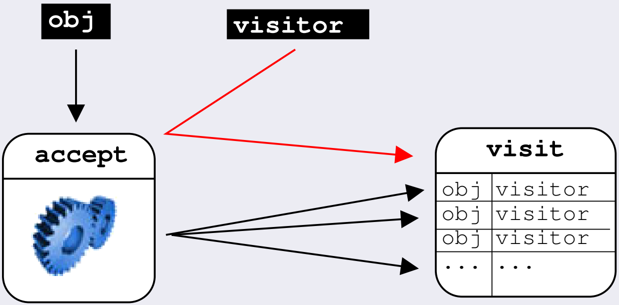
Summary of step 1

1. **Original hierarchy untouched**
   - Generic function model (outside the classes)
2. **Abstract the visiting process away**
   - Still needs to be done

Step 2: brute force visiting

- **Abstract the visiting process away**
  - OK: the `accept` generic function

But what's wrong with this picture ?



▶ One indirection too many

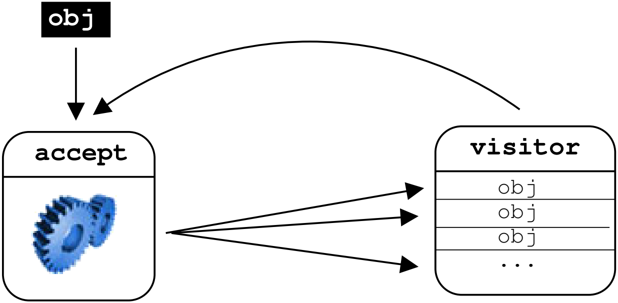
Step 3: first class (generic) functions

Notion of first class / order
(Christopher Strachey, 1916–1975)

- storage (in variables)
- aggregation (in structures)
- argument (to functions)
- return value (from functions)
- anonymous manipulation
- dynamic creation
- ...

▶ Generic functions are first class objects in LISP

The better picture



Retrieving function objects in LISP

```
(function func) ;; => #<FUNCTION FUNC>
#'func          ;; => #<FUNCTION FUNC>
```

Step 4: mapping

- **Prominent concept in functional programming**
  - Along with folding (reduction), filtering *etc.*
- **Thanks to first class functions**
  - Argument passing

Typical mapping example

```
(mapcar #'string-upcase '("foo" "bar" "baz"))
;; => ("FOO" "BAR" "BAZ")
```

▶ “visiting” is a form of *structural mapping*

Step 5: generic mapping

- **Having to specialize `mapobject` is boring**
  - Mapping over lists, vectors, arrays, *even class slots* should be written only once

The CLOS Meta-Object Protocol (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

▶ We have *reflexive* (introspective) access to class slots

Step 6: objects

How about a component counter visitor ?

- **C++:** left as an exercise. . .
- **LISP:** how does that fit with first class functions ?
  - Global state (yuck !)
  - Behavior + state = objects !
- So we're back to visitor *objects* ?

▶ There has *got* to be a better way. . .

Step 7: lexical closures

- **Behavior + State** without the OO machinery

Typical functional example (with anonymous function)

```
(defun make-adder (n)
  (lambda (x) (+ n x)))

(funcall (make-adder 3) 5) ;; => 8
```

Closures with mutation (impure functional programming)

```
(let ((count 0))
  (defun increment ()
    (incf count)))

(increment) ;; => 1
(increment) ;; => 2
;; ...
```

Step 8: dynamic visitation schemes

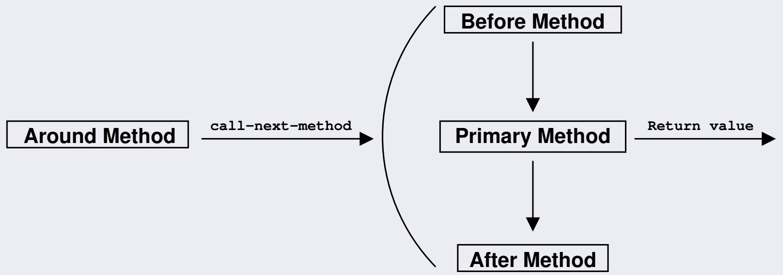
How about a component *nesting* counter visitor ?

- **C++:** left as an exercise. . .
- **LISP:** modification of the visit process required
  1. increment nesting level before visiting an object
  2. actual visit
  3. decrement nesting level afterwards
- Do we need a dedicated `mapobject` for that ?

▶ No ! We have the MOP's generic function protocol

The generic function protocol

Generic function invocation



Around Method

`call-next-method`

Before Method

Primary Method

After Method

`Return value`

- Methods are CLOS meta-objects
- Methods can be added/removed dynamically

Summary

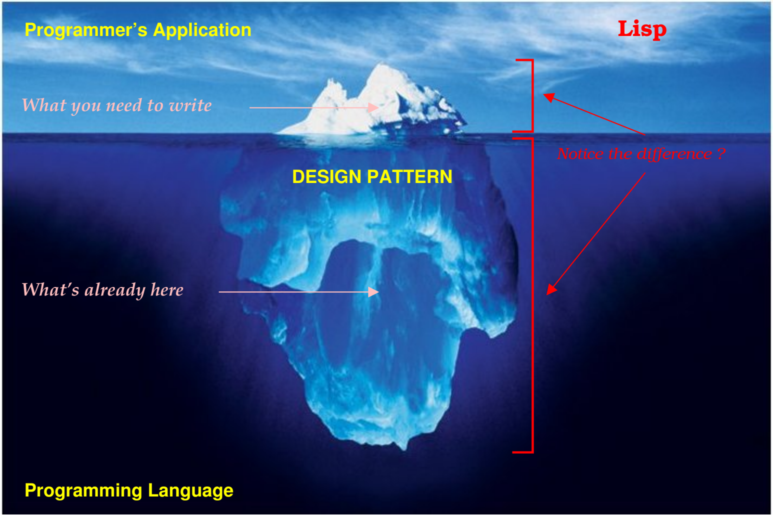
- **Decoupling from original hierarchy:** n/a
  - Generic function model (outside the classes)
- **Visiting infrastructure:**
  - First class generic functions (as argument)
  - CLOS MOP (introspection)

  ▶ Generic machinery in 10 lines of code
- **Visiting with state:**
  - Lexical closures
  - First class functions (anonymous)
  - Generic function protocol (before/after)-methods

  ▶ 5–10 more lines of code (original code untouched)

Conclusion

The “iceberg” metaphor



Programmer’s Application

Lisp

What you need to write

DESIGN PATTERN

Notice the difference ?

What’s already here

Programming Language

Next LISP Events

- **ELS'09: 2nd European LISP Symposium**
  May 27-29 2009, Milan, Italy
  `http://www.european-lisp-symposium.org`
- **ELW'09: 6th European LISP Workshop**
  July 6 2009, Genova, Italy
  co-located with ECOOP.
  `http://elw.bknr.net/2009`