Object-Oriented Approaches to Programming

Case Study: Design Patterns

Didier Verna

didier@didierverna.net


didierverna.net


@didierverna


didier.verna


in/didierverna

Build 2025-06-25 22:04:33+02:00

Outline

Plan

Initial Assessment

- **Design Patterns**
  - Software engineering problems “FAQ”
  - Inspired from [Alexander, 1977]’s work
  - (Context **) / Problem / Solution / (Consequences *)
- **Bibliography**
  - “GoF” (*) [Gamma et al., 1994] (23 patterns)
  - “POSA” (**) [Bushmann et al., 1996, Schmidt et al., 2000, Kirsher et al., 2004, Buschmann et al., 2007, Buschmann et al., 2007]
- **Assessment** [Norvig, 1996]

  “16 of 23 patterns are either invisible or simpler in Dylan or Lisp”

Explanation

GoF (Introduction)

"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."

- Disclaimer too frequently forgotten

Patterns Classification

- **GoF:** creational, structural, behavioral
  - Usage oriented
- **POSA:** architectural, conceptional, idioms
  - Abstraction oriented

Idioms (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."

- Design patterns (GoF) $\Longleftrightarrow$ idioms (POSA)

Design Patterns Usage

- Software engineering “ready-made”
- No substitute to common sense / reflexion

Remark (POSA)

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

“Visitor” (GoF) example

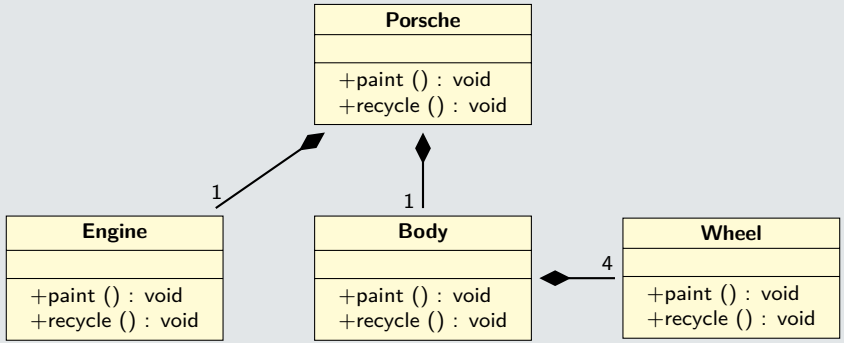
“Use the Visitor pattern when [...] many distinct and unrelated operations need to be performed on objects [...], and you want to avoid ***“polluting” their classes with these operations****.”*

- But who said that “methods belonged to classes”?

Plan

Typical Case

Naive C++ version

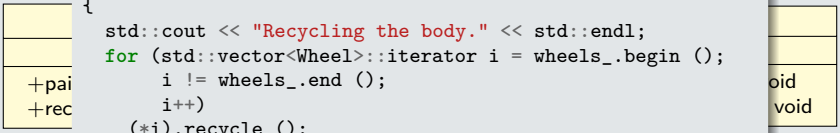


- New actions: modification to the hierarchy necessary
- Code duplication: hierarchy traversal (propagation)

Typical Case

Naive C++ version

```
void paint ()
{
  std::cout << "Painting the body." << std::endl;
  for (std::vector<Wheel>::iterator i = wheels_.begin ();
       i != wheels_.end ();
       i++)
    (*i).paint ();
};
void recycle ()
{
  std::cout << "Recycling the body." << std::endl;
  for (std::vector<Wheel>::iterator i = wheels_.begin ();
       i != wheels_.end ();
       i++)
    (*i).recycle ();
};
```



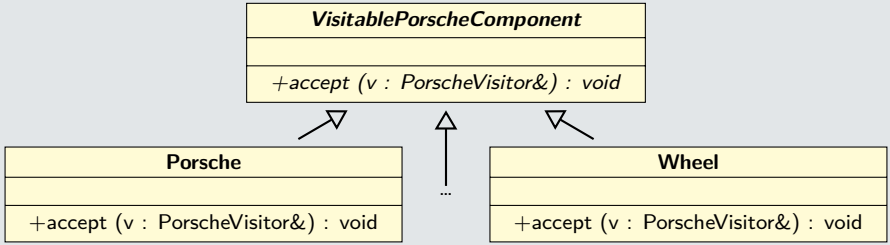
- New actions: modification to the hierarchy necessary
- Code duplication: hierarchy traversal (propagation)

The Visitor Pattern

- **Goals**
  1. Leave the original hierarchy intact
  2. Abstract structure traversal away
- **Mechanism**
  - Original hierarchy
    - *visitable* (abstract class)
    - *accept* visitors (methods)
  - Visitors
    - know how to *visit* (abstract class) ...
    - ... each component (methods)

The Visitor Pattern

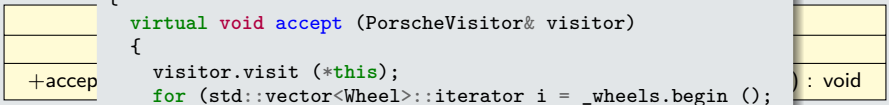
Original hierarchy



The Visitor Pattern

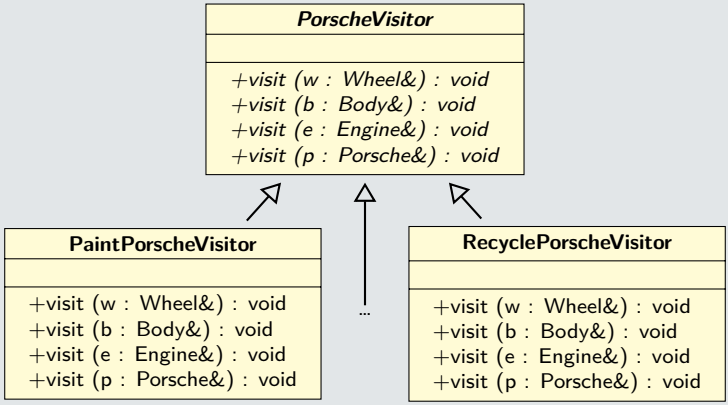
Original hierarchy

```
struct VisitablePorscheComponent
{
  virtual void accept (PorscheVisitor&) = 0;
};

struct Body : public VisitablePorscheComponent
{
  virtual void accept (PorscheVisitor& visitor)
  {
    visitor.visit (*this);
    for (std::vector<Wheel>::iterator i = _wheels.begin ();
         i != _wheels.end ();
         i++)
      i->accept (visitor);
  };
};
```



The Visitor Pattern

Visitors



The Visitor Pattern

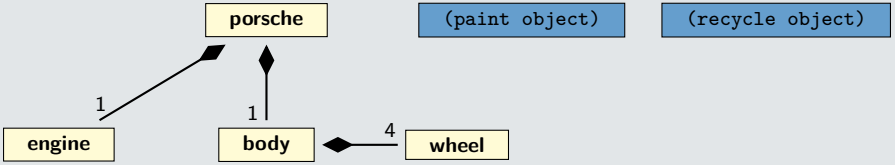
Visitors

```
struct PorscheVisitor
{
  virtual void visit (Wheel&) = 0;
  virtual void visit (Body&) = 0;
  virtual void visit (Engine&) = 0;
  virtual void visit (Porsche&) = 0;
};

struct PaintPorscheVisitor : public PorscheVisitor
{
  virtual void visit (Wheel& wheel)   { ... };
  virtual void visit (Body& body)     { ... };
  virtual void visit (Engine& engine) { ... };
  virtual void visit (Porsche& porsche) { ... };
};
```


Plan

Typical Case

Naive Lisp version



- ✔ Leave the original hierarchy intact
  Methods do not belong to classes
- ✘ Abstract the structure traversal away

Typical Case

Naive Lisp version

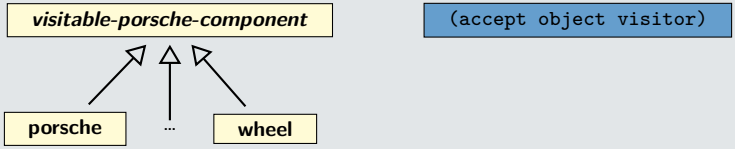
```
(defgeneric paint (object)
  ...
  (:method ((body body))
    #| paint the body |#
    (dolist (wheel (wheels body))
      (paint wheel)))
  (:method ((wheel wheel)) #| paint the wheel |#))

(defgeneric recycle (object)
  ...
  (:method ((body body))
    #| recycle the body |#
    (dolist (wheel (wheels body))
      (recycle wheel)))
  (:method ((wheel wheel)) #| recycle the wheel |#))
```

engine

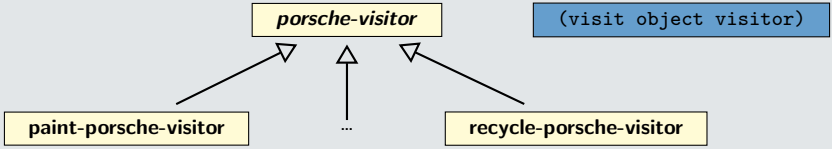
- ✔ Leave the ori
  Methods do
- ✘ Abstract the

The Visitor Pattern

Original hierarchy



Visitors



Static Idioms Eradication

- `visitable-porshe-component` useless, even in C++
  - Not robust (subclassing may be forgotten)
  - Not necessary (missing `accept` method ⇒ error)
  - Note: checking for `accept` completeness *is* possible (MOP)

Back to the original hierarchy...

```
~~(defclass visitable-porsche-component () ())~~

(defclass wheel (~~visitable-porsche-component~~) ())
...
```

Static Idioms Eradication

- `porshe-visitor` useful C++
  - accept prototyping

`(accept object visitor)`

```
(:method ((wheel wheel) (visitor porsche-visitor))
  (visit wheel visitor))
(:method ((body body) (visitor porsche-visitor))
  (visit body visitor)
  (dolist (wheel (wheels body))
    (accept wheel visitor)))
...
```

- Only the first argument is discriminating

Static Idioms Eradication

- porshe-visitor useful C++
  - accept prototyping

```
(accept object visitor)
```

```
(:method ((wheel
  (visit wheel vi
(:method ((body b
  (visit body vis
  (dolist (wheel
    (accept wheel
...
```

```
(defclass porsche-visitor () ())

(defgeneric accept (object visitor)
  (:method ((wheel wheel) (visitor porsche-visitor))
    (visit wheel visitor))
  ...)

(defclass paint-porsche-visitor   (porsche-visitor) ())
(defclass recycle-porsche-visitor (porsche-visitor) ())
```

- Only the first argument is discriminating

Visitor Objects Eradication

Usefulness?

```
(defgeneric accept (object visitor)
  (:method ((wheel wheel) visitor)
    (visit wheel visitor))
  ...)
(defclass paint-porsche-visitor () ())
(defmethod visit ((wheel wheel) (visitor paint-porsche-visitor))
  #| paint the wheel |#)
...
```

- Select the appropriate `visit` method
- Useless in a functional language
  Higher order functions as arguments

Visitor Objects Eradication

Usefulness?

```
(defgeneric accept (object visitor-function)
  ...
  (:method ((body body) visitor-function)
    (funcall visitor-function body)
    (dolist (wheel (wheels body))
      (accept wheel visitor-function)))
  ...)

(defgeneric paint (object)
  (:method ((wheel wheel)) #| paint the wheel |#...)
  ...)

(accept porsche #'paint)
```

- Select the ap
- Useless in a f
  Higher order

Interlude: on Name Pertinence

accept ⇒ visit

```
(let ((porsche (make-instance 'porsche)))
  (~~accept~~visit porsche #'paint)
  (~~accept~~visit porsche #'recycle))
```

visit ⇒ map-object

```
(defgeneric ~~visit~~map-object (func object)
  (:method (func object)
    (funcall func object))
  (:method (func (body body))
    (funcall func body)
    (dolist (wheel (wheels body))
      (~~visit~~map-object func wheel)))
  ...)
```


Automatic Mapping

- Avoid specializing `map-object` by hand
- Map on `object`, then all slots

MOP-based introspection

```
(defgeneric map-object (func object)
  (:method (func (object list)) (mapc func object))
  (:method (func (object standard-object))
    (funcall func object)
    (mapc
        (lambda (slot)
          (map-object func
                      (slot-value object (slot-definition-name slot))))
      (class-direct-slots (class-of object)))))
```

- Many other refinements possible…

Plan

Stateful Visitors

- **C++**
  - Trivial (object = state + behavior)
- **Lisp**
  - Functions + global state (yuck)
  - CLOS Objects (yuck)
  - Lexical closures (obviously)!

Example: a components counter

```
(let ((porsche (make-instance 'porsche))
      (counter 0))
  (map-object (lambda (obj) (incf counter)) porsche))
```

Visitation Mechanism Customization

- **C++**
  - Not trivial (left as an exercise…)
- **Lisp**
  1. Temporary (dynamic) modification of `map-object`
  2. EQL-specialization of the visiting function

Example: a nesting counter (I)

```
(let* ((porsche (make-instance 'porsche))
       (depth 0)
       (before (defmethod map-object :before (func object) (incf depth)))
       (after  (defmethod map-object :after  (func object) (decf depth))))
  (map-object (lambda (obj)
                (format t "~S, current level: ~A~%"
                  (class-name (class-of obj)) depth))
              porsche)
  (remove-method #'map-object before)
  (remove-method #'map-object after))
```


Visitation Mechanism Customization

- **C++**
  - Not trivial (left as an exercise…)
- **Lisp**
  1. Temporary (dynamic) modification of `map-object`
  2. EQL-specialization of the visiting function

Example: a nesting counter (II)

```
(let ((depth 0))
  (defun print-depth (obj)
    (format t "~S, current level: ~A~%" (class-name (class-of obj)) depth))
  (defmethod map-object :before ((func (eql #'print-depth)) object)
    (incf depth))
  (defmethod map-object :after ((func (eql #'print-depth)) object)
    (decf depth)))

(let* ((porsche (make-instance 'porsche)))
  (map-object #'print-depth porsche))
```

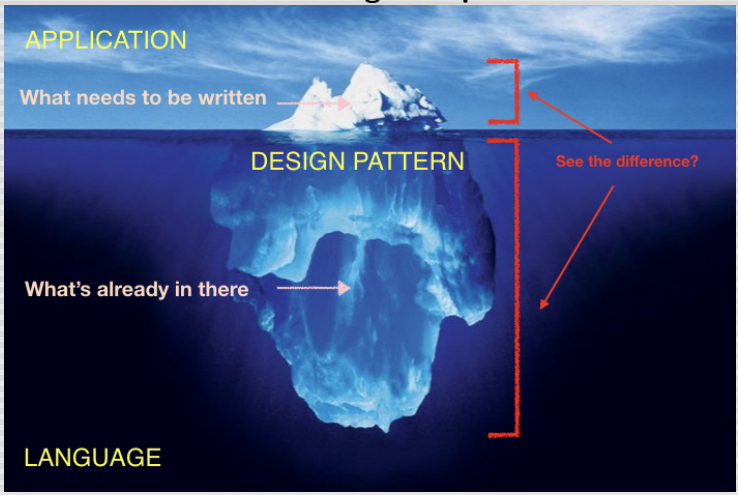

Plan

Summary

- **Leave the original hierarchy intact:** n/a
  - Generic functions outside classes
- **Visitation infrastructure abstraction:** 10 lines
  - Higher order generic functions
  - CLOS MOP (introspection)
- **Stateful visitors:** +5-10 lines
  - Lexical closures
  - Anonymous functions
  - Extended methods (`before`/`after`)

Conclusion

The “Iceberg Metaphor”



Plan

Bibliography

Bibliography I

Alexander, C. (1977)
A Pattern Language: Towns, Buildings, Constructions
Oxford University Press

Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. (1994)
Design Patterns
Addison-Wesley

Buschmann, F.and Meunier, R. and Rohnert, H. and Sommerlad, P. and Stal M. (1996)
Pattern-Oriented Software Architecture
Wiley

Schmidt, D.C. and Stal, M. and Rohnert, H. and Buschmann, F.
Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (2000)
Wiley

Bibliography II

Kircher, M. and Jain, P. (2004)
Pattern-Oriented Software Architecture: Patterns for Resource Management
Wiley

Buschmann, F. and Henney, K. and Schmidt, D.C. (2007)
Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing
Wiley

Buschmann, F. and Henney, K. and Schmidt, D.C. (2007)
Pattern-Oriented Software Architecture: On Patterns and Pattern Languages
Wiley

Norvig, P. (1996)
Design Patterns in Dynamic Programming
Object World Conference

Bibliography III

Verna, D. (2010)
Revisiting the Visitor: the Just Do It pattern
Journal of Universal Computer Science, Vol. 16.2