



Introduction



C++



Lisp



État



Conclusion

# Approches Objet de la Programmation

## ～ Étude de Cas : les Design Patterns ～

Didier Verna  
EPITA / LRDE

[didier@lrde.epita.fr](mailto:didier@lrde.epita.fr)



[lrde/~didier](#)



[@didierverna](#)



[didier.verna](#)



[google+](#)



[in/didierverna](#)

# Plan

Introduction

Visiteurs C++

Visiteurs Lisp

Bonus : Visiteurs à État

Conclusion



# Plan

Introduction

Visiteurs C++

Visiteurs Lisp

Bonus : Visiteurs à État

Conclusion



# Constat Initial

- ▶ **Design Patterns** (« patrons de conception »)
  - ▶ « FAQ » des problèmes de génie logiciel
  - ▶ Inspirés des travaux d'[Alexander, 1977]
  - ▶ (Contexte \*\*) / Problème / Solution / (Conséquences \*)
- ▶ **Bibliographie**
  - ▶ « GoF » (\*) [Gamma et al., 1994] (23 patterns)
  - ▶ « POSA » (\*\*)  
[Bushman et al., 1996, Schmidt et al., 2000, Kirsher et al., 2004,  
Buschmann et al., 2007, Buschmann et al., 2007]
- ▶ **Constat** [Norvig, 1996]  
« 16 of 23 patterns are either invisible or simpler in Dylan or Lisp »

# Explication

## 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. »*

- ▶ Aspect critique trop souvent oublié

# Classification des patterns

- ▶ **GoF** : créationnelles, structurelles, comportementales
  - ▶ Orientée usage
- ▶ **POSA** : architecturales, de conception, idiotismes
  - ▶ Orientée abstraction

## Idiotismes (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)  $\longleftrightarrow$  idiotismes (POSA)



# Utilisation des Design Patterns

- ▶ Prêt-à-Porter du génie logiciel
- ▶ Pour autant, pas un substitut au bon sens / à la réflexion

## Remarque (POSA)

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

## Exemple du « Visiteur » (GoF)

*« 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. »*

- ▶ Qui a dit que les « opérations appartiennent aux classes » ?



# Plan

Introduction

Visiteurs C++

Visiteurs Lisp

Bonus : Visiteurs à État

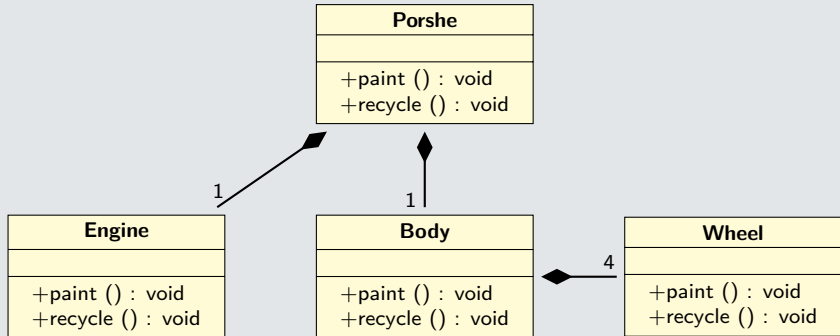
Conclusion





# Cas d'École

## Version C++ naïve



- Nouvelles actions : modification de la hiérarchie nécessaire
- Duplication de code : parcours de la hiérarchie (propagation)

# Cas d'École

## Version C++

```

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 ();
};

```

En

+paint  
+recycle

► Not

► Duplication de code : parcours de la hiérarchie (propagation)



# Le Pattern Visiteur

## ► Objectifs

1. Ne pas toucher à la hiérarchie de départ
2. Abstraire le parcours de la structure

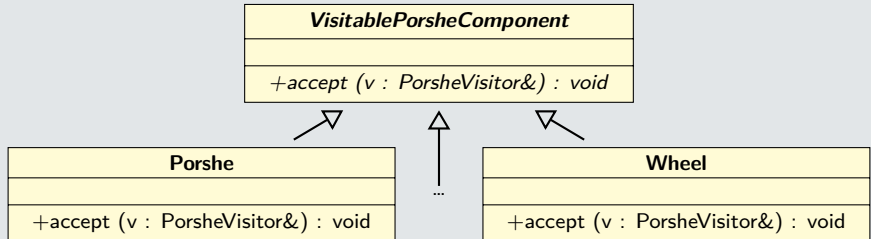
## ► Mécanisme

- Hiérarchie de départ
  - *visitable* (classe abstraite)
  - *accepter* des visiteurs (méthodes)
- Visiteurs
  - savoir *visiter* (classe abstraite) ...
  - ... chaque composant (méthodes)



# Le Pattern Visiteur

## Hiérarchie de départ



# Le Pattern Visiteur

## Hiérarc C++

```

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);
    };
};

```

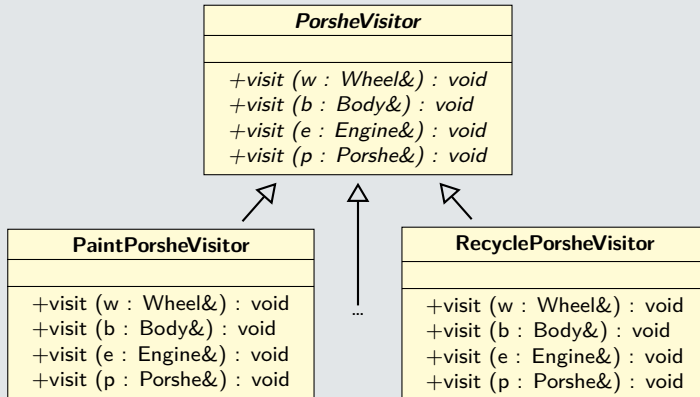
+accept

: void



# Le Pattern Visiteur

## Visiteurs



# Le Pattern Visiteur

## Visiteurs

### C++

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

struct PaintPorsheVisitor : public PorsheVisitor
{
    virtual void visit (Wheel& wheel)  { ... };
    virtual void visit (Body& body)    { ... };
    virtual void visit (Engine& engine) { ... };
    virtual void visit (Porshe& porshe) { ... };
};
```

# Plan

Introduction

Visiteurs C++

Visiteurs Lisp

Bonus : Visiteurs à État

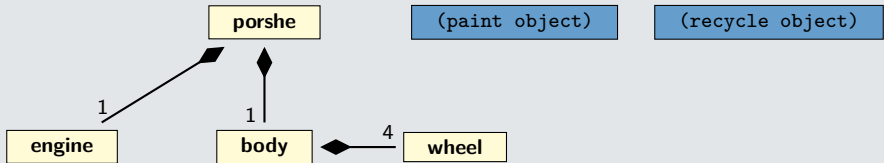
Conclusion





# Cas d'École

## Version Lisp naïve



- ✓ Ne pas toucher à la hiérarchie de départ  
*Les méthodes ne font pas partie des classes*
- ✗ Abstraire le parcours de la structure

# Cas d'École

## Version Lisp naïve Lisp

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

1. engine

✓ Ne
Les (defgeneric recycle (object)
  ...
  (:method ((body body))
    #!/ recycle the body /#
    (dolist (wheel (wheels body))
      (recycle wheel)))
  (:method ((wheel wheel)) #!/ recycle the wheel /#))

X Abs
```

object)

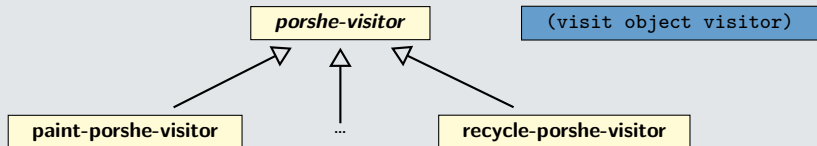


# Le Pattern Visiteur

## Hiérarchie de départ



## Visiteurs



# Élimination des Idiotismes Statiques

- ▶ visitable-porsche-component inutile, même en C++
  - ▶ Pas robuste (oubli toujours possible)
  - ▶ Pas nécessaire (méthode accept manquante ⇒ erreur)
  - ▶ Note : vérification de la complétude de accept possible (MOP)

## Retour à la hiérarchie de départ...

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

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

```
...
```



# Élimination des Idiotismes Statiques

- ▶ porsche-visitor utile en C++
  - ▶ Prototypage de accept

## (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)))  
...
```

- ▶ Seul le premier argument est discriminant



# Élimination des Idiotismes Statiques

- ▶ porsche-visitor utile en C++
  - ▶ Prototypage de accept

## Lisp

(accept

```
(defclass porsche-visitor () ())
(:method
  (visit
    (defgeneric accept (object visitor)
      (:method ((wheel wheel) {visitor porsche-visitor})
        (visit wheel visitor))
      (dolis
        ...))
    (acc
      ...
      (defclass paint-porsche-visitor (porsche-visitor) ())
      (defclass recycle-porsche-visitor (porsche-visitor) ())
```

- ▶ Seul le premier argument est discriminant



# Élimination des Objets Visiteurs

## Utilité des objets visiteurs ?

```
(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 |#)
...
```

- ▶ Sélectionner la méthode `visit` appropriée
- ▶ Inutile dans un langage fonctionnel  
*Fonctions du premier ordre utilisées comme argument*



# Élimination des Objets Visiteurs

## Utilité des objets visiteurs ?

### Lisp

```
(defgeneric accept (object visitor-function)
  (vis ...
    ...) (:method ((body body) visitor-function)
      (funcall visitor-function body))
  (defmeth (dolist (wheel (wheels body))
    #/ paint (accept wheel visitor-function)))
  ...)

(defgeneric paint (object)
  (:method ((wheel wheel) #/ paint the wheel /#...)
    ...)
  (accept porsche #'paint))
```

► Sélectionner

► Input

For





## Interlude : Pertinence du Nommage

accept  $\Rightarrow$  visit

```
(let ((porsche (make-instance 'porsche)))  
  (acceptvisit porsche #'paint)  
  (acceptvisit porsche #'recycle))
```

visit  $\Rightarrow$  map-object

```
(defgeneric visitmap-object (func object)  
  (:method (func object)  
    (funcall func object))  
  (:method (func (body body))  
    (funcall func body)  
    (dolist (wheel (wheels body))  
      (visitmap-object func wheel)))  
  ...)
```

# Mapping Automatique

- ▶ Éviter de spécialiser map-object à la main
- ▶ Mapper sur object puis sur tous ses slots

## Introspection par le MOP

```
(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)))))
```

- ▶ Nombreux autres raffinements possibles...



# Plan

Introduction

Visiteurs C++

Visiteurs Lisp

Bonus : Visiteurs à État

Conclusion



# Visiteurs à État

## ► C++

- Trivial (objet = état + comportement)

## ► Lisp

- Fonctions + état global (beurk)
- Objets CLOS (beurk)
- Fermetures lexicales (tout simplement) !

## Exemple : un compteur de composants

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

# Spécialisation du Mécanisme de Visite

## ► C++

- Non trivial (laissé en exercice...)

## ► Lisp

1. Modification temporaire (dynamique) de map-object
2. EQL-spécialisation de la fonction visiteuse

## Exemple : un compteur d'imbrication (I)

```
(let* ((porshe (make-instance 'porshe))
      (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))
              porshe)
  (remove-method #'map-object before)
  (remove-method #'map-object after))
```

# Spécialisation du Mécanisme de Visite

## ► C++

- Non trivial (laissé en exercice...)

## ► Lisp

1. Modification temporaire (dynamique) de map-object
2. EQL-spécialisation de la fonction visiteuse

## Exemple : un compteur d'imbrication (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

Introduction

Visiteurs C++

Visiteurs Lisp

Bonus : Visiteurs à État

Conclusion



# Résumé

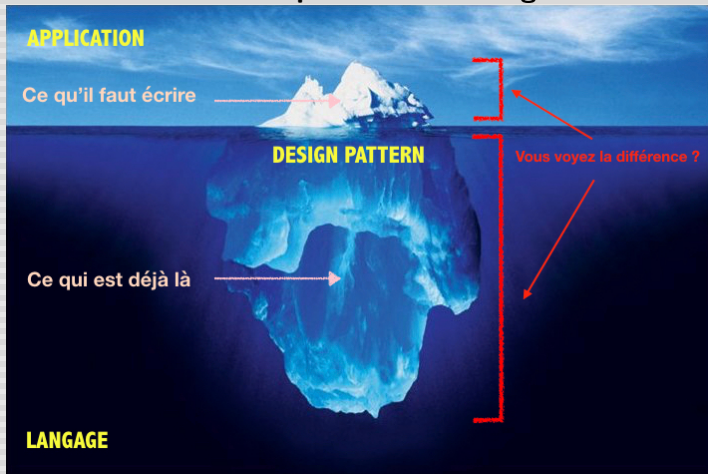
- ▶ **Ne pas toucher à la hiérarchie de départ** : n/a
  - ▶ Fonctions génériques hors-classes
- ▶ **Abstraire l'infrastructure de visite** : 10 lignes
  - ▶ Fonctions génériques de 1<sup>re</sup> classe
  - ▶ CLOS MOP (introspection)
- ▶ **Visiteurs à État** : +5-10 lignes
  - ▶ Fermetures lexicales
  - ▶ Fonctions anonymes
  - ▶ Méthodes étendues (before/after)





# Conclusion




## La « Métaphore de l'Iceberg »






# Plan

Bibliographie




# Bibliographie 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*

## Bibliographie II

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

## Bibliographie III

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