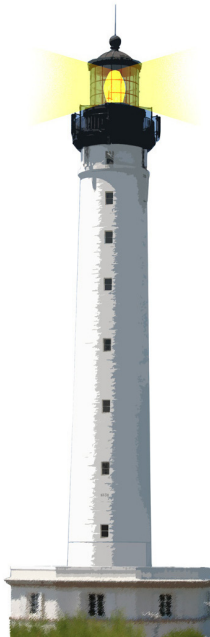
Object-Oriented
Design Lecture

Visitor

Stéphane Ducasse and Luc Fabresse
http://stephane.ducasse.free.fr
http://car.mines-douai.fr/luc



Pharo
http://www.pharo.org

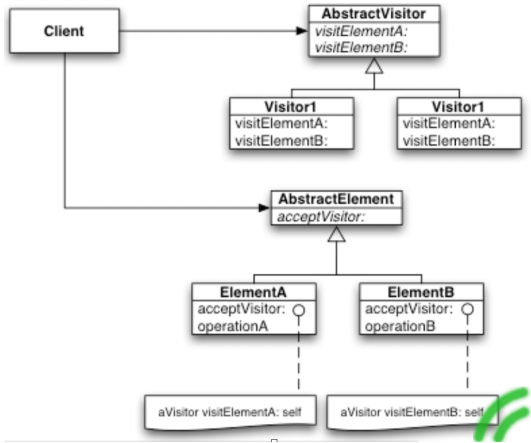
Outline

- Visitor
- Visitor discussions
- Visitor variations

Visitor intent

Represent an operation to be performed on the elements of an object structure in a class separate from the elements themselves.
Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor design



Little arithmetic expressions

- Supports one kind of number, and has +, *, (,)
- We want to evaluate expressions, and print them

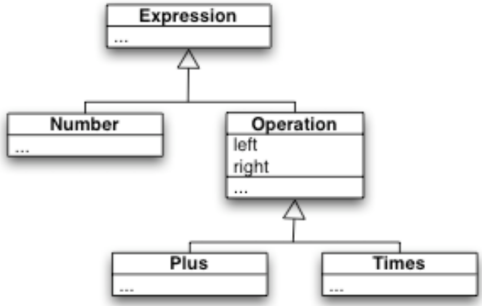
Evaluating

```
1 + (3 * 2)
gives 7
```

Printing

```
+1*32
```

Expression hierarchy



Some expressions

1

```
ENumber value: 1
```

(3 * 2)

```
Times left: (ENumber value: 3) right: (ENumber value: 2)
```

1 + (3 * 2)

```
Plus
  left: (ENumber value: 1)
  right: (Times left: (ENumber value: 3) right: (ENumber value: 2))
```

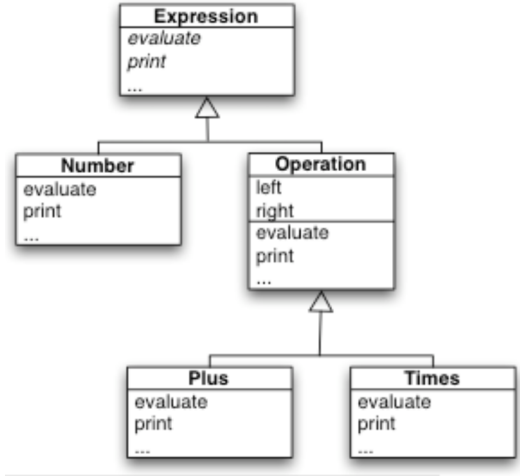
Of course in Pharo we can just extend Number so no need of ENumber value: but this is a detail

Operation implementation

Two solutions:

- add methods for evaluating, printing, ... on Expression and its subclasses
- create a Visitor, add the visit methods on Expression and its subclasses, and implement visitors for evaluation, printing, ...

Expression first design

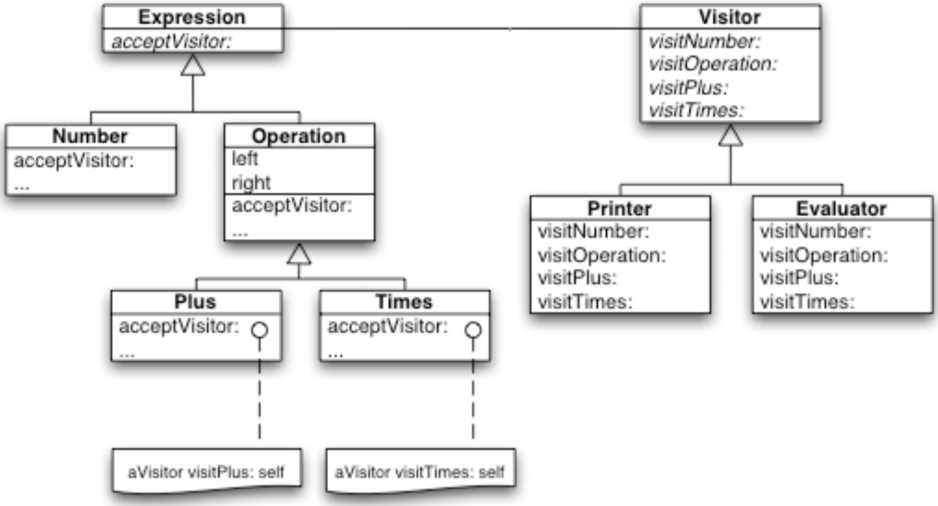


First design analysis

- What if we need a stack to print well the expressions?
  - Should we put it in the expressions even if this is related only to print?
- What if we need a table for mathematical expression only for the LaTeX exporter?

Why should we mix the information about the treatment of items and items themselves?

Expression Visitor



Expression Visitor analysis

Each visitor knows what to do for a number, a plus and times operation

Evaluator Visitor

```
Evaluator >> visitNumber: aNumber
  ^ aNumber value
```

```
Evaluator >> visitPlus: anExpression
  | l r |
  l := anExpression left acceptVisitor: self.
  r := anExpression right acceptVisitor: self.
  ^ l + r
```

```
Evaluator >> visitTimes: anExpression
  | l r |
  l := anExpression left acceptVisitor: self.
  r := anExpression right acceptVisitor: self.
  ^ l * r
```

Invoking the Visitor

```
Evaluator new evaluate:
  (Plus
    left: (ENumber value: 1)
    right: (Times left: (ENumber value: 3) right: (ENumber value: 2)))
> 7
```

```
Evaluator >> evaluate: anExpression
  ^ anExpression acceptVisitor: self
```

Printer

```
Visitor subclass: #Printer
  iv: 'stream level'
```

```
Printer >> visitNumber: aNumber
    stream nextPutAll: aNumber value asString
```

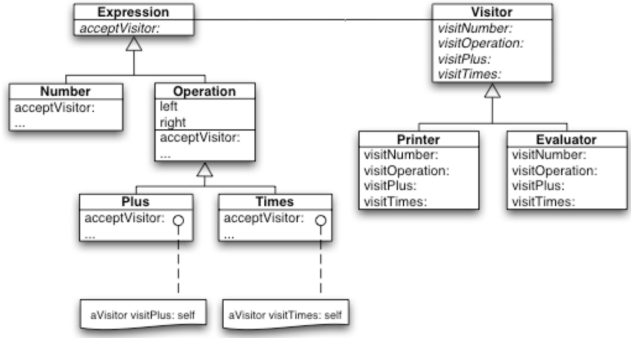
```
Printer >> visitPlus: anExpression
   stream nextPutAll: '+'.
   anExpression left acceptVisitor: self.
   anExpression right acceptVisitor: self.
```

```
Printer >> visitPlus: anExpression
   stream nextPutAll: '*'.
   anExpression left acceptVisitor: self.
   anExpression right acceptVisitor: self.
```

Visitor study

Check the double dispatch

- The Visitor knows the elementary operations
- The items declare how they want to be visited



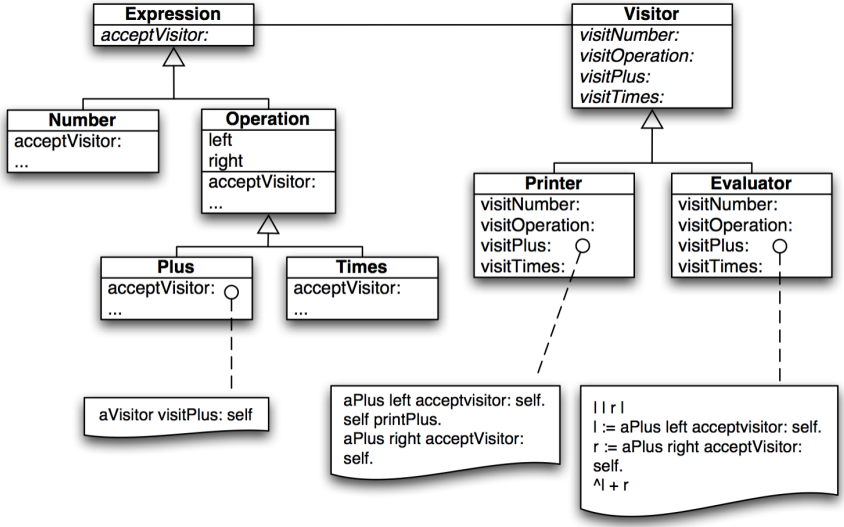
Controlling the traversal

A visitor embeds a structure traversal

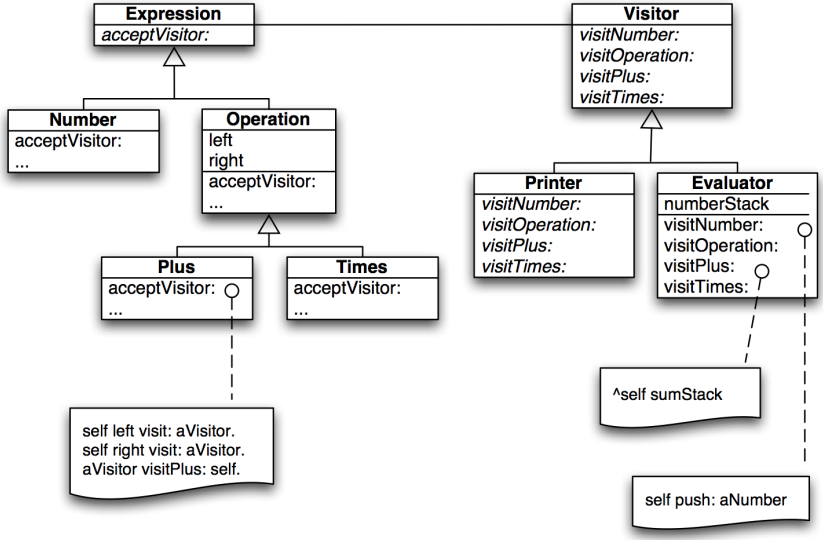
- Be default all the items should be reached
- There are different places where the traversal can be implemented:
  - in the visitors
  - in the items themselves

Usually the visitor is under control but may be the domain logic is more important.

Visitor in control



Items in control



Subtle points

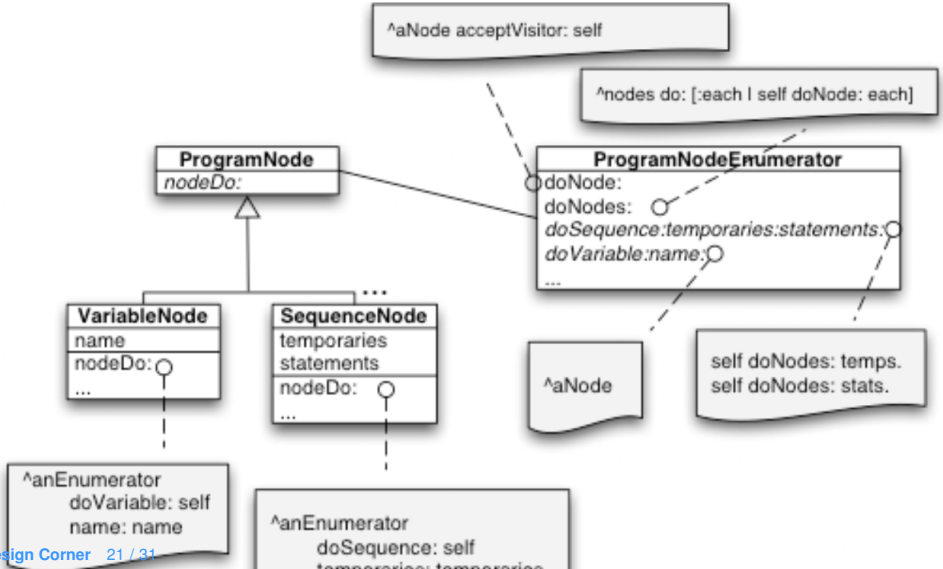
Because a hook is associated with a template

- when subclassing a hook the user can ’know’ more

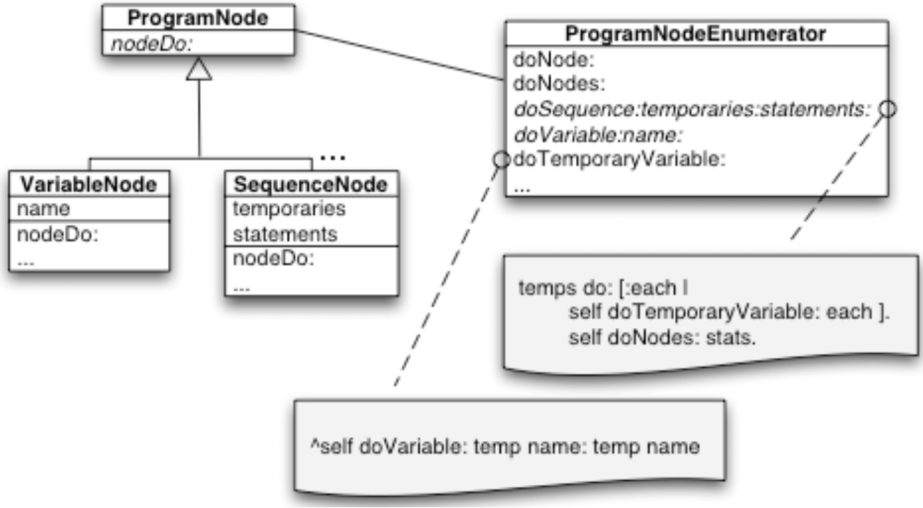
Now the visit method name can also encode context

- visit methods can be used to let user extend functionality at fine granularity
- the granularity of visit methods has an impact on the hooks they offer

Misopportunities



Using messages as cases (again)

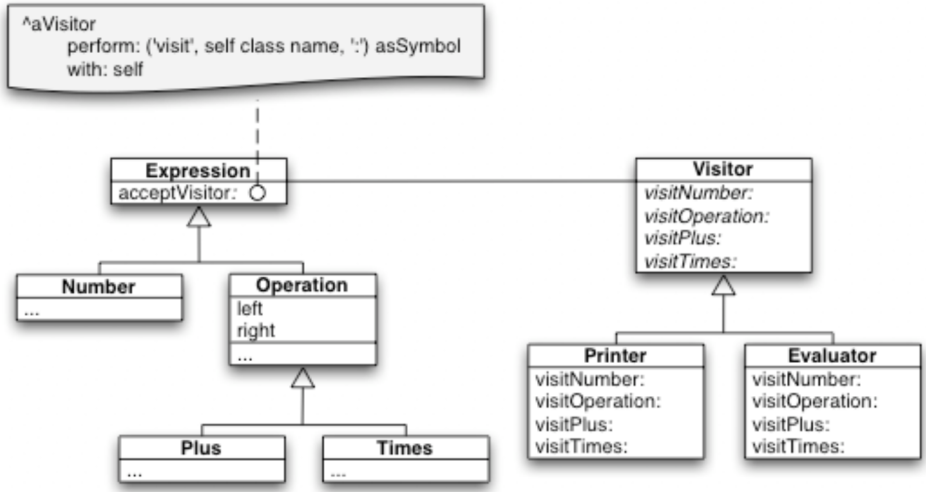


Using messages as cases (again)

In this case using doTemporaryVariable: tell the extender that such method is not for all the variables but only for temporaries.

- No need to test
- No need to get a state
- just subclass doTemporaryVariable:

Visitor with reflection



When to use a Visitor

- Whenever you have a number of items on which you have to perform a number of actions
- When you ‘decouple’ the actions from the items.

Examples:

- Parse tree (ProgramNode) uses a visitor for the compilation (emitting code on CodeStream)
- Rendering documents (Document) in different formats

If your domain is not stable

You may end up having to change multiple visitors each time and it can be tedious.

Visitor is not OOP controversy

Yes operations applied on objects are defined outside the objects. But

- May be the operations require a complex state that has nothing to do with the structure (RTF/PDF stack)
- Each Visitor encapsulates a complex operation

Class extension

Even if a language supports class extension (defining methods on a class from another package than the class package), using a visitor is better because

- Each Visitor encapsulates a complex operation
- Each Visitor has its own state

Do not use overloading for visit method

As a summary, overloading does not really work in Java and you will have to explicitly cast your visitor or use getClass everywhere.

- Better define method visitNumber(), visitPlus(), visitTimes()
- than visit()

Trust an expert :)
Check overloading lecture.

Conclusion

- Visitor can be tricky to master
  - use accept/visit vocabulary to really help you
- Visitor is nice for complex structure operations

A course by

Stéphane Ducasse
http://stephane.ducasse.free.fr

and

Luc Fabresse
http://car.mines-douai.fr/luc