Visitor

Modular and extensible first class actions

S. Ducasse, L. Fabresse, G. Polito, and P. Tesone
Goals

- Studying examples
- Understanding the Visitor design pattern
- Discussions on pros and cons
Example: basic arithmetic expressions

Imagine a simple mathematical system

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

Remarks:

- In this example, we reify everything
- Pharo supports class extension
  - So, no need to wrap numbers with our own `Number`, we could directly extend the Pharo core `Number`

```
Plus
  left: 1
  right: (Times left: 3 right: 2)
```
Basic arithmetic expressions as Composite

An expression is represented by a Composite with numbers and operations (see Lecture on Composite)
Some expressions

1

Number value: 1

(3 * 2)

Times left: (Number value: 3) right: (Number value: 2)

1 + (3 * 2)

Plus
  left: (Number value: 1)
  right: (Times left: (Number value: 3) right: (Number value: 2))
Operations on the expressions

We want two operations on expressions:

- Evaluate

\[
1 + (3 \times 2) > 7
\]

- Print (in Polish notation)

\[
1 + (3 \times 2) > +1*32
\]
First design: behavior defined in the domain
First design: behavior defined in the domain

```
Number >> evaluate
  ^ value

Plus >> evaluate
  ^ left evaluate + right evaluate

Number >> print
  stream nextPutAll: value asString

Plus >> print
  ...
```
First design: analysis

- Some operations require some state
  - e.g. a stack is needed to print expressions in infix notation
- Where should we define such state?
  - in the expression classes?
  - even if this is only related to print?

Should we mix the state of operations on items with the items themselves?
Overview of a real system

The Pillar Pharo library:
- a core hierarchy of 50 classes (document model)
- export to LaTeX (two versions)
- export to HTML
- export to Beamer
- export to ASCIIdoc, Markdown, Microdown
- transform trees for expansion
- code checkers
- ...
First design: conclusion

Putting all the behavior inside domain objects:

- **Blows up** the class API / state / methods
- **Mixes** concerns
- Is **not modular**: we cannot have one operation only
- **Prevents extension**: adding a new behavior requires changing the domain
Essence of the Visitor design pattern

A Visitor:

- **Represents** an operation
- **Decouples** this operation from the domain objects it applies to (separate class)
- Supports **modularity** (separate package)
- Supports **extension**
  - We define **once** a set of messages (e.g., `visitX`) in domain objects
  - Then, new visitors (operations) are easy to define **without changing domain objects** it operates on
Overview of a Visitor-based design

Visitor

ExpressionVisitor

Evaluator

Printer

InfixPrinter

Domain

Expression

Number

Operation

Plus

Times

Evaluator

Printer

InfixPrinter

evaluate(e)

print(e)

stack
Visitor: key points

A Visitor:

- requires a structure to operate on
- performs different actions based on the kind of the elements
  - knows what operation to do for a Number, a Plus, and a Times
- manages its own specific state
- is independent of other ones

Visitor + Composite: a perfect match
Using Visitors

"1+(3*2)"
expr := (Plus
  left: (Number value: 1)
  right: (Times
    left: (Number value: 3)
    right: (Number value: 2)))).

Evaluator new evaluate: expr.
> 7

Printer new print: expr.
> +1*32

InfixPrinter new print: expr.
> 1+(3*2)
**Visitor implementation: Domain instrumentation**

Prepare the domain to accept Visitors:

- **add** `acceptVisitor` on each composite element
- **tells** the visitor passed in parameter how to visit it

**Only once for all Visitors**
Visitor implementation: Domain instrumentation

Number >> acceptVisitor: aVisitor
  ^ aVisitor visitNumber: self

Plus >> acceptVisitor: aVisitor
  ^ aVisitor visitPlus: self

Times >> acceptVisitor: aVisitor
  ^ aVisitor visitTimes: self

- Only once for all Visitors
- Domain objects tell to the Visitor how they want to be visited
  - visitNumber:, visitPlus:, visitTimes:, visitXXX:
Visitor implementation

A Visitor:

- executes the right operation for an element
- propagates recursively on composite elements
  - acceptVisitor:

```
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
```
Visitor: an extensible design

Supporting a new operation is simple:

- Define a new Visitor class
  - e.g., Printer
- Implement the expected API
  - i.e. visitNumber, visitPlus and visitTimes
- Use it

```plaintext
anExpression acceptVisitor: Printer new
Printer new print: anExpression
```
Visitor: step back

Did you really understood the subtle interaction between acceptVisitor and visitXXX methods?
Double dispatch

Visitor

ExpressionVisitor

...  

Printer

print(e)
visitNumber(n)
visitPlus(p)
visitTimes(t)
...

Domain

Expression

acceptVisitor(v)
...

dispatch

Number

acceptVisitor(v)
...

Operation

left
right
...

Plus

acceptVisitor(v)
...

Times

acceptVisitor(v)
...

print: anExpression

^ anExpression acceptVisitor: self
Double dispatch

```
Visitor

ExpressionVisitor

...  

Printer

print(e)
visitNumber(n)
visitPlus(p)
visitTimes(t)
...

Domain

Expression

acceptVisitor(v)
...

Number

acceptVisitor(v)
...

Operation

left
right
...

Plus

acceptVisitor(v)
...

Times

acceptVisitor(v)
...
```

- `print: anExpression` 
- `acceptVisitor: self`
- `dispatch`
- `acceptVisitor: aVisitor`
- `visitPlus: self`

Diagram: Visitor and Domain classes with methods to handle expressions and operations.
Visitor core: Double dispatch

Double dispatch:
- Core mechanism of Visitor
- No conditional checks
- Provides decoupling between:
  - Visitors and domain objects
  - Different visitors
When to use a Visitor

Whenever you have to perform multiple operations on structured object graphs
Examples:

- Parse tree (ProgramNode) uses a Visitor for
  - the compilation (emitting code on CodeStream),
  - pretty printing, syntax highlighting
  - different analysis pass, rotten green test analysis

- Rendering documents (Document) in different formats
  - nodes expansion, HTML, LaTeX, ...

M6S4 24/27
When using a Visitor is challenging

- If the elements of the composite **change**
  - It requires to change **all** Visitors
- Related to the *expression problem* in statically typed languages
Conclusion

Pros:
- Visitor is a very nice pattern
- It provides a modular and extensible design
- Double dispatch makes it plug and play

Cons:
- Can look complex
- Not well adapted to changing structures
Advanced Object-Oriented Design and Development with Pharo

A course by
S. Ducasse, L. Fabresse, G. Polito, and P. Tesone

Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France
https://creativecommons.org/licenses/by-nc-nd/3.0/fr/