The Visitor Design Pattern

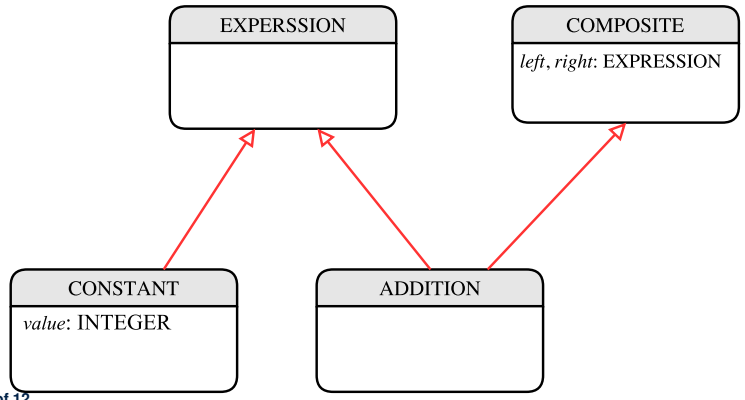
YORK UNIVERSITÉ UNIVERSITY

EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Motivating Problem (1)

Based on the *composite pattern* you learned, design classes to model *structures* of arithmetic expressions (e.g., `341`, `2`, `341 + 2`).



Open/Closed Principle

Software entities (classes, features, etc.) should be ***open*** for *extension*, but ***closed*** for *modification*.

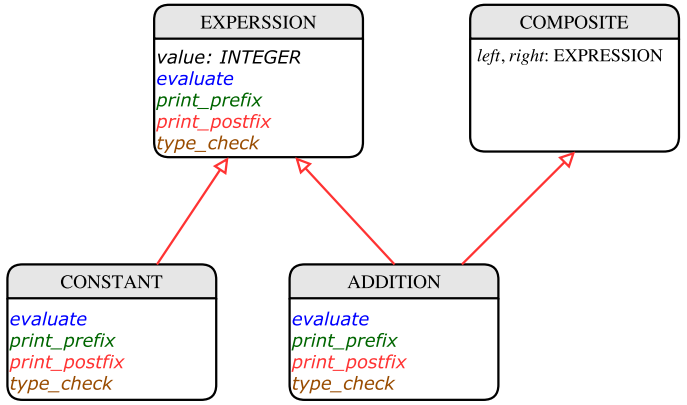
⇒ When *extending* the behaviour of a system, we may *add new code*, but we should *not modify the existing code*.

e.g., In the design for structures of expressions:

- ***Closed***: Syntactic constructs of the language [stable]
- ***Open***: New operations on the language [unstable]

Motivating Problem (2)

Extend the *composite pattern* to support *operations* such as `evaluate`, pretty printing (`print_prefix`, `print_postfix`), and `type_check`.



Problems of Extended Composite Pattern

- Distributing the various **unrelated** ***operations*** across nodes of the ***abstract syntax tree*** violates the *single-choice principle*:

  To add/delete/modify an operation
  ⇒ Change of all descendants of `EXPRESSION`

- Each node class lacks in *cohesion*:

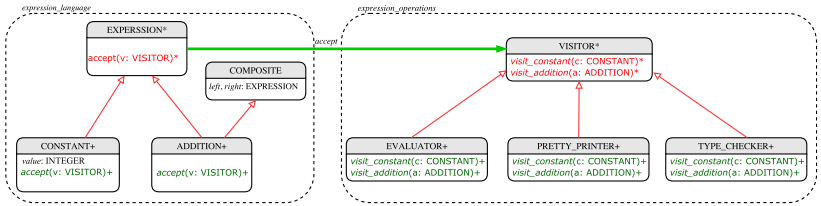
  A *class* is supposed to group *relevant* concepts in a *single* place.
  ⇒ Confusing to mix codes for evaluation, pretty printing, and type checking.
  ⇒ We want to avoid "polluting" the classes with these various unrelated operations.

Visitor Pattern

- *Separation of concerns*:
  - Set of language constructs [***closed***, stable]
  - Set of operations [***open***, unstable]

  ⇒ Classes from these two sets are *decoupled* and organized into two separate clusters.

Visitor Pattern: Architecture



Visitor Pattern Implementation: Structures

Cluster ***expression_language***

- Declare *deferred* feature `accept(v: VISITOR)` in `EXPRSSION`.
- Implement `accept` feature in each of the descendant classes.

```
class CONSTANT
...
  accept(v: VISITOR)
    do
      v.visit_constant(Current)
    end
end
```

```
class ADDITION
...
  accept(v: VISITOR)
    do
      v.visit_addition(Current)
    end
end
```

Visitor Pattern Implementation: Operations

Cluster ***expression_operations***

- For each descendant class `C` of `EXPRESSION`, declare a *deferred* feature `visit_c (e: C)` in the *deferred* class `VISITOR`.

```
class VISITOR
  visit_constant(c: CONSTANT) deferred end
  visit_addition(a: ADDITION) deferred end
end
```

- Each descendant of `VISITOR` denotes a kind of operation.

```
class EVALUATOR
  value: INTEGER
  visit_constant(c: CONSTANT)  do value := c.value end
  visit_addition(a: ADDITION)
    local eval_left, eval_right: EVALUATOR
    do a.left.accept(eval_left)
       a.right.accept(eval_right)
       value := eval_left.value + eval_right.value
    end
end
```


Testing the Visitor Pattern

```
1  test_expression_evaluation: BOOLEAN
2   local add, c1, c2: EXPRESSION ; v: VISITOR
3   do
4     create {CONSTANT} c1.make (1) ; create {CONSTANT} c2.make (2)
5     create {ADDITION} add.make (c1, c2)
6     create {EVALUATOR} v.make
7     add.accept(v)
8     check attached {EVALUATOR} v as eval then
9       Result := eval.value = 3
10    end
11  end
```

Double Dispatch in **Line 7**:

1. ***DT*** of `add` is *ADDITION* ⇒ Call `accept` in *ADDITION*
   `v.visit_addition(add)`
2. ***DT*** of `v` is *EVALUATOR* ⇒ Call `visit_addition` in *EVALUATOR*
   visiting result of `add.left` + visiting result of `add.right`

To Use or Not to Use the Visitor Pattern

- In the architecture of visitor pattern, what kind of *extensions* is easy and hard? Language structure? Language Operation?
  - Adding a new kind of ***operation*** element is easy.
    To introduce a new operation for generating C code, we only need to introduce a new descendant class `C_CODE_GENERATOR` of `VISITOR`, then implement how to handle each language element in that class.
    ⇒ *Single Choice Principle* is *obeyed*.
  - Adding a new kind of ***structure*** element is hard.
    After adding a descendant class `MULTIPLICATION` of `EXPRESSION`, every concrete visitor (i.e., descendant of `VISITOR`) must be amended to provide a new `visit_multiplication` operation.
    ⇒ *Single Choice Principle* is *violated*.
- The applicability of the visitor pattern depends on to what extent the ***structure*** will change.
  ⇒ Use visitor if ***operations*** applied to ***structure*** might change.
  ⇒ Do not use visitor if the ***structure*** might change.

Index (1)