



Ingegneria del Software

12/02/2018

Design Pattern Visitor and Composite

Valutatore di operazioni logiche

Emanuele Vivoli

Intento

Il sistema si propone nel valutare delle operazioni logiche (composizioni di operazioni and, or, not) fra elementi booleani, in un contesto dove le operazioni supportate sull'espressione logica comprendono:

es. $((A \text{ and } !B) \text{ or } (!C \text{ and } D))$

- **stampa dell'espressione** caratterizzando il **nome** delle variabili in gioco
 $((A \ \&\& \ !B) \ || \ (\ !C \ \&\& \ D))$
- **stampa dell'espressione** focalizzata sul **valore** di ogni variabile
 $((\text{true} \ \&\& \ !\text{false}) \ || \ (\ !\text{true} \ \&\& \ \text{false}))$
- **stampa del risultato dell'espressione**
 true

Motivazione

Un'espressione logica è caratterizzata tramite il Pattern Composite come una composizione limitata (poiché il Composite permette un numero variabile di figli, mentre le operazioni logiche and, or e not permettono rispettivamente 2, 2 ed 1 figlio soltanto).

I figli sono rappresentati sia come Leaf che come Composite (attenendosi ai nomi GoF).

Le operazioni che vogliamo siano supportate sono le tre specificate nell' **Intento**, ma potrebbero essere possibili altre operazioni come la valutazione della correttezza dell'espressione (rilevare se vi è un errore e a che livello), e molte altre.

E' dunque necessario implementare ogni operazione in maniera differente sia su Leaf che su Composite, poiché per il Pattern Composite sia Leaf che Composite devono essere trattati alla stessa maniera relativamente all'operazione da implementare. Questo ci fa intendere che le varie operazioni debbano avere un'implementazione diversa anche per tipi diversi di Composite differenziandosi in And, Or e Not.

Inoltre per ogni operazione da specializzare aggiunta alla classe astratta Component è necessario mettere mano ad ogni classe creata ed aggiungere per ognuna un'implementazione della stessa.

Sarebbe, dunque, un buon criterio quello di separare la gerarchia di operazioni dalla gerarchia di dati sulle quali le operazioni possono essere svolte.

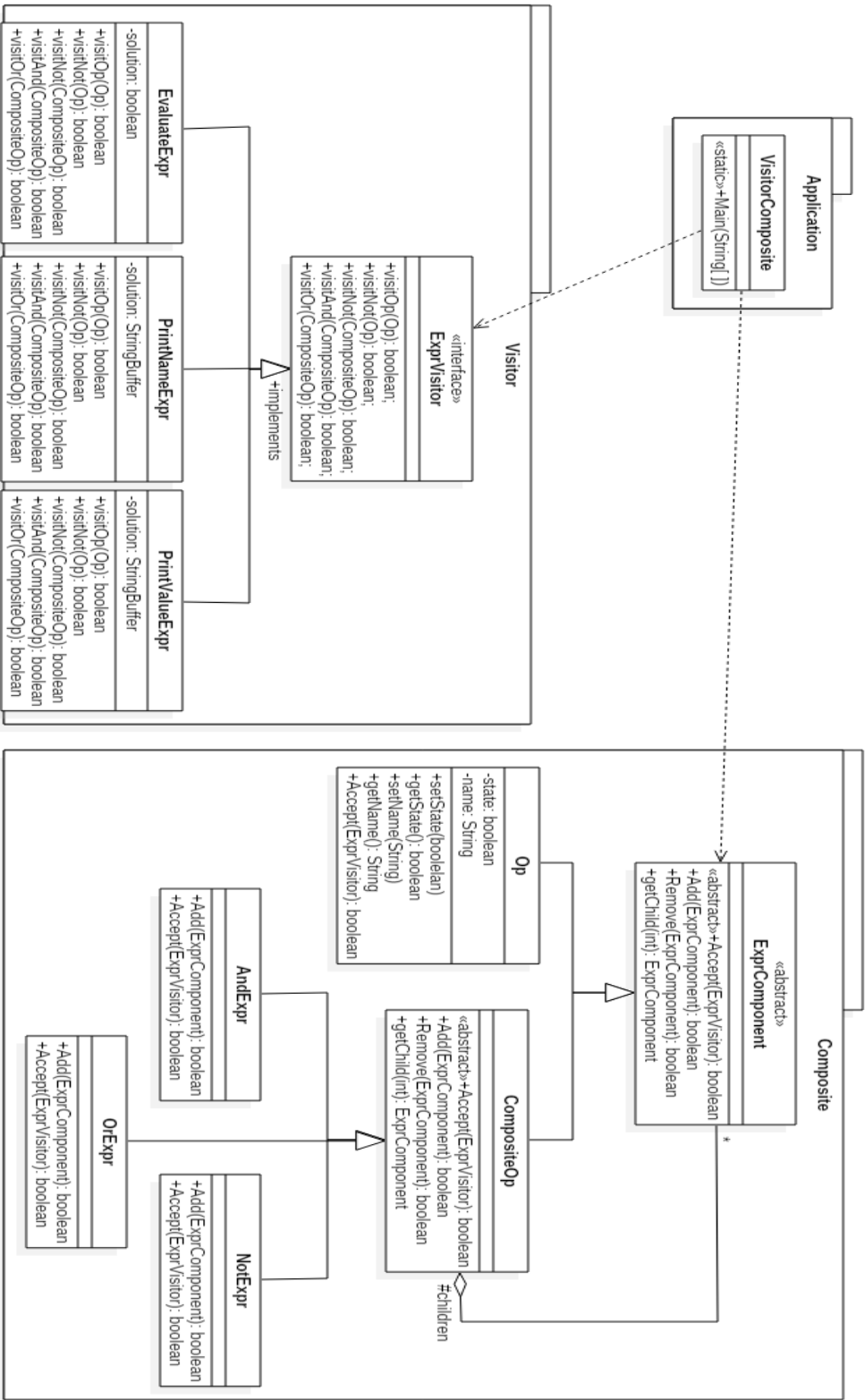
Questa è proprio la responsabilità del Pattern Visitor, cioè quello di rappresentare una gerarchia di operazioni che possono essere applicate ad una gerarchia di dati.

In questo modo si rende possibile l'aggiunta, la cancellazione e la modifica di operazioni sull'insieme di dati rappresentato dal Composite (nel nostro caso un'espressione logica), senza dover modificare lo schema del Pattern Composite ogni qualvolta si necessita di ampliare le operazioni.

Ciò è appunto fatto definendo un solo metodo in Component che accetta un Visitor, e richiama il metodo riferito alla propria classe, nel Visitor, passando un Self-reference.

Per aggiungere una nuova operazione basterà aggiungere una nuova sottoclasse nella gerarchia del Pattern Visitor.

Struttura



Partecipanti

ExprVisitor (package Visitor)

Rappresenta l'interfaccia Visitor (in riferimento al Pattern) ed espone una definizione dei metodi specificando per ognuno il valore di ritorno *boolean* poiché si deve essere capaci di gestire espressioni di tipo *boolean*.

Vedremo poi che, con il fatto di implementare *ExprVisitor*, anche altre operazioni (classi derivate) che non trattano l'espressione dal punto di vista del valore booleano che rappresenta, sono obbligate ad avere un *return true* fasullo per poter essere conformi all'interfaccia.

```
1. package Visitor;
2.
3. public interface ExpressionVisitor {
4.     public boolean visitOp(Op op);
5.     public boolean visitNot(Op op);
6.     public boolean visitNot(CompositeOp Cop);
7.     public boolean visitAnd(CompositeOp Cop);
8.     public boolean visitOr(CompositeOp Cop);
9. }
```

EvaluateExpr (package Visitor)

Classe responsabile del calcolo del valore booleano dell'espressione.

All'interno i metodi sono implementati in modo da operare come specificato dal nome del metodo stesso. Le operazioni sono svolte sulle foglie, che vengono ottenute tramite iterazioni successive di chiamata passando un self-reference alla foglia.

Il self-reference viene inviato al metodo Accept (che vedremo successivamente).

Il metodo visitOp(...) è il "caso base", che conclude il circolo di "call-back" ritornando il valore della variabile a cui ci stiamo riferendo.

```
1. public class EvaluateExpression implements ExpressionVisitor{
2.     boolean solution = true;
3.
4.     @Override
5.     public boolean visitOp(Op op) {
6.         return op.getState();
7.     }
8.
9.     @Override
10.    public boolean visitNot(Op op) {
11.        solution = !(boolean) op.Accept(this);
12.        return solution;
13.    }
14.
15.    @Override
16.    public boolean visitNot(CompositeOp Cop) {
17.        solution = !(boolean) Cop.getChild(0).Accept(this);
18.        return solution;
19.    }
```

```
20.
21.     @Override
22.     public boolean visitAnd(CompositeOp Cop) {
23.         Boolean b1 = ((boolean) Cop.getChild(0).Accept(this));
24.         Boolean b2 = ((boolean) Cop.getChild(1).Accept(this));
25.         solution = new Boolean(b1.booleanValue() && b2.booleanValue());
26.         return solution;
27.     }
28.
29.     // public boolean visitOr(CompositeOp Cop) is similar to And
30. }
```

Print[Name,Value]Expr (package Visitor)

Queste classi (una *PrintName*, l'altra *PrintValue*) hanno la responsabilità di stampare l'espressione booleana.

Questo avviene, rispettivamente, in base al nome ed al valore assegnati alle variabili (al momento della creazione).

In questo caso viene utilizzato uno *StringBuffer* per memorizzare passo passo l'espressione ottenuta. Poiché ad ogni riciclo il valore dello *StringBuffer* viene sovrascritto, al termine delle operazioni si avrà nello *StringBuffer* la sequenza che rappresenta l'espressione.

Si noti che tutti i metodi della classe hanno un valore di ritorno di tipo booleano, *return true*, che non è assolutamente necessario ai fini del ruolo che ricoprono, ma è necessario però per una corretta realizzazione del programma, poiché ogni classe che definisce una nuova operazione deve attenersi alla struttura dell'interfaccia *ExprVisitor*, facendo "override" di tutti i metodi.

```
1. public class PrintNameExpression implements ExpressionVisitor{
2.     private StringBuffer solution = new StringBuffer();
3.
4.     @Override
5.     public boolean visitOp(Op op) {
6.         solution.append(" "+op.getName());
7.         return true;
8.     }
9.
10.    @Override
11.    public boolean visitNot(Op op) {
12.        solution.append(" !(");
13.        op.Accept(this);
14.        solution.append(" )");
15.        return true;
16.    }
17.
18.    // public boolean visitNot(CompositeOp Cop) is similar to the upper
19.
20.    @Override
21.    public boolean visitAnd(CompositeOp Cop) {
22.        solution.append(" (");
23.        Cop.getChild(0).Accept(this);
24.        solution.append(" && ");
25.        Cop.getChild(1).Accept(this);
26.        solution.append(" )");
27.        return true;
28.    }
```

```

29.
30.    // public boolean visitOr(CompositeOp Cop) is similar to And
31.
32.    public void printExpression(){
33.        System.out.println(solution);
34.    }
35. }

```

ExprComponent (package *Composite*)

Questa classe è realizzata come classe astratta per non avere una implementazione dei metodi “*leaf-releated*” nelle foglie, ma implementare una definizione “*fake*” nella classe base.

Questo accade poiché mi sono attenuto al Pattern Composite. Avrei potuto evitare di utilizzare il metodo *Add(...)* per aggiungere un child ad un’oggetto di tipo *CompositeOp*, e piuttosto utilizzare il metodo più intuitivo di passare due elementi come parametri del metodo costruttore. Per completezza ho deciso dunque di permettere entrambe le implementazioni.

```

1.  package Composite;
2.
3.  public abstract class ExpressionComponent {
4.      public abstract boolean Accept(ExpressionVisitor v);
5.
6.      public boolean Add(ExpressionComponent op) throws Exception{
7.          throw new Exception("impossible Add() in leaf");
8.      }
9.
10.     // same implementation for Remove() and getChild()
11. }

```

Op (package *Composite*)

Questa classe rappresenta un oggetto della fattispecie “variabile”, cioè un singolo elemento della espressione logica (o booleana). Ha ovviamente al suo interno il valore “*state*” che lo rappresenta ed un’informazione relativa al nome. Entrambe le variabili con i rispettivi metodi “*getter and setter*”.

Poiché *ExprComponent* è una classe astratta e non un’interfaccia non ho bisogno di implementare in questa classe i metodi *children-releated*, nel caso utilizzo le implementazioni “*fake*” fornite dalla classe base.

Questa classe, poiché classe concreta di *ExprComponent*, espone il metodo *Accept(ExprVisitor v)* che ha il compito di richiamare il metodo “giusto” nell’interfaccia *Visitor*, senza sapere cosa realmente faccia il metodo, o di classe sia l’istanza concreta che le viene passata.

Questo è possibile poiché l’interfaccia *Visitor* ha l’obbligo di fornire un metodo per ogni istanza concreta di Component, in modo che ogni operazione che viene svolta possa avere implementazioni tutte diverse, una per ogni tipo di oggetto.

Questo è il fulcro del disaccoppiamento.

```

1.  public class Op extends ExpressionComponent{
2.      private boolean state;
3.      private String name;

```

```
4.
5.     public Op(String name, boolean state){
6.         this.state = state;
7.         this.name = name;
8.     }
9.
10.    // setter and getter methods
11.
12.    @Override
13.    public boolean Accept(ExpressionVisitor v){
14.        return v.visitOp(this);
15.    }
16. }
```

CompositeOp (package Composite)

CompositeOp è una classe astratta che eredita a sua volta dalla classe astratta ExprComponent. Da questa eredita anche i metodi Add(...), Remove(...) e getChild(..) che però deve re-implementare per le classe figlie.

```
1. public abstract class CompositeOp extends ExpressionComponent{
2.     protected ArrayList<ExpressionComponent> children = new ArrayList<>();
3.     // implementation of Add(), Remove() and getChild()
4.     @Override
5.     public abstract boolean Accept(ExpressionVisitor v);
6. }
```

[And | Or | Not]Expression (package Composite)

Queste classi rappresentano le operazioni che posso esser fatte sulle variabili booleane.

Propongono una propria versione del metodo *Add()* , ognuna secondo le politiche che la caratterizzano.

Riguardo al metodo *Accept(ExprVisitor v)* sono già state spese diverse parole.

Di particolare importanza è il fatto che l'aggiunta di una nuova operazione nella gerarchia dei Component, comporta una modifica di tutta l'infrastruttura del visitor.

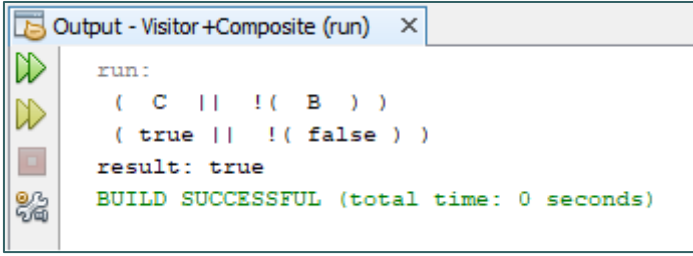
Questo è uno dei pericoli nell'utilizzare questo Pattern comportamentale (visitor).

```
1. public class AndExpression extends CompositeOp{
2.     // constructor and Add() method implementation
3.
4.     @Override
5.     public boolean Accept(ExpressionVisitor v){
6.         return v.visitAnd(this);
7.     }
8. }
```

Codice Esempio

```
1. public class VisitorComposite {
2.     public static void main(String[] args) {
3.         // (C or !B)
4.
5.         ExpressionComponent B = new Op(" B ", false);
6.         ExpressionComponent C = new Op(" C ", true);
7.
8.         ExpressionComponent not1 = new NotExpression(B);
9.         ExpressionComponent or1 = new OrExpression(C, not1);
10.
11.         ExpressionVisitor print1 = new PrintNameExpression();
12.         ExpressionVisitor print2 = new PrintValueExpression();
13.         ExpressionVisitor result = new EvaluateExpression();
14.
15.         boolean val;
16.         val = or1.Accept(result);
17.         or1.Accept(print1);
18.         or1.Accept(print2);
19.
20.         ((PrintNameExpression) print1).printExpression();
21.         ((PrintValueExpression) print2).printExpression();
22.         System.out.println("result: "+val);
23.     }
24. }
```

Output



```
Output - Visitor+Composite (run) X
run:
( C || !( B ) )
( true || !( false ) )
result: true
BUILD SUCCESSFUL (total time: 0 seconds)
```


Sequence Diagram dell'esempio

