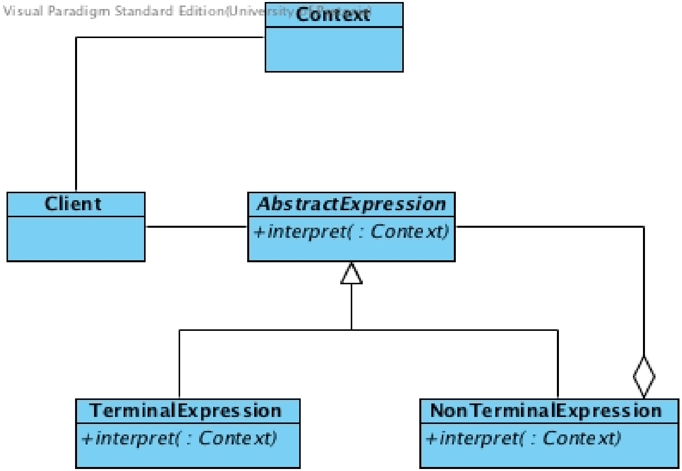
Interpreter

Linda Marshall and Vreda Pieterse

Department of Computer Science
University of Pretoria

17 October 2014

Name and Classification: Interpreter, Class Behavioural
Intent: “Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.” (GoF:243)


Visual Paradigm Standard Edition(University of Pretoria)
Context
Client
AbstractExpression
+interpret(: Context)
TerminalExpression
+interpret(: Context)
NonTerminalExpression
+interpret(: Context)

- Client: manipulates the abstract syntax tree that represents a sentence in a language
- Context: information required by the interpreter

- AbstractExpression: declares an interpret method, for a particular context, that is common to all nodes in the expression tree.

- TerminalExpression: implements the interpret method for terminal symbols in the language. Each terminal symbol in a sentence will have an instance defined
- NonterminalExpression: every rule in the grammar is represented by a non-terminal. A rule comprises of a sequence of terminals and/or non-terminals.

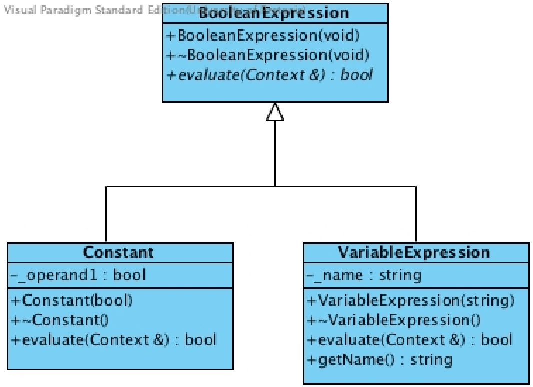
- **Composite** forms the basis for the abstract syntax tree.
- **Iterator** can be used to traverse the abstract syntax tree and visitor is used to maintain behaviour of the nodes in the abstract syntax tree.

```
BooleanExpression ::= Constant |
                      VariableExpression |
                      NotExpression |
                      AndExpression |
                      OrExpression
AndExpression ::= BooleanExpression '&&' BooleanExpression
OrExpression ::= BooleanExpression '||' BooleanExpression
NotExpression ::= '!' BooleanExpression
Constant ::= true | false
VariableExpression ::= 'a' | 'b' | ... { 'a' | 'b' |  ... }*
```

```
#ifndef BOOLEANEXPRESSION_H
#define BOOLEANEXPRESSION_H

 class Context;

 class BooleanExpression
 {
 public:
     explicit BooleanExpression(void);
     virtual ~BooleanExpression(void);

     virtual bool evaluate(Context&) = 0;
 };
#endif

 // explicit keyword prevents compiler from making
 // implicit conversions
```

```
BooleanExpression::BooleanExpression(void)
{
}

BooleanExpression::~BooleanExpression(void)
{
}
```

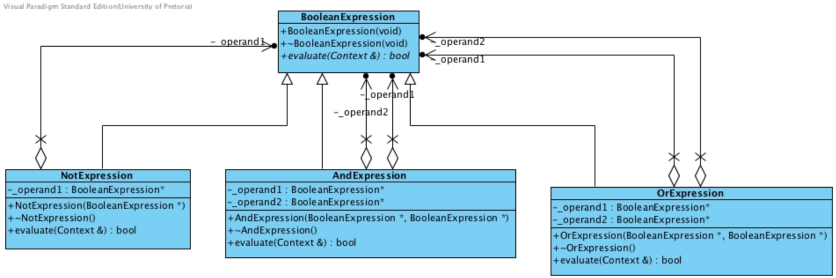
TerminalExpression participants: Constant and VariableExpression


Visual Paradigm Standard Edition(University of Pretoria)
BooleanExpression
+BooleanExpression(void)
+~BooleanExpression(void)
+evaluate(Context &) : bool
Constant
-_operand1 : bool
+Constant(bool)
+~Constant()
+evaluate(Context &) : bool
VariableExpression
-_name : string
+VariableExpression(string)
+~VariableExpression()
+evaluate(Context &) : bool
+getName() : string

```
Constant::Constant(bool op1)
{
    _operand1 = op1;
}

bool Constant::evaluate(Context& aContext)
{
    return (_operand1);
}

Constant::~Constant(void)
{
}
```

```
VariableExpression::VariableExpression(const std::string name)
{
    _name = name;
}

VariableExpression::~VariableExpression(void)
{
}

bool VariableExpression::evaluate(Context& aContext)
{
    return aContext.lookup(_name);
}

std::string VariableExpression::getName()
{
    return _name;
}
```

NonterminalExpression participants: NotExpression, AndExpression, OrExpression

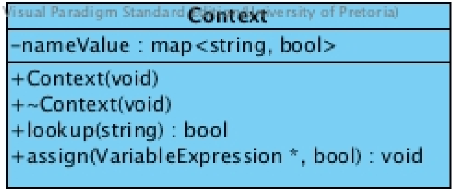

Visual Paradigm Standard Edition(University of Pretoria)
BooleanExpression
+BooleanExpression(void)
+~BooleanExpression(void)
+evaluate(Context &) : bool
- operand1
-_operand2
-_operand1
-_operand1
-_operand2
NotExpression
-_operand1 : BooleanExpression*
+NotExpression(BooleanExpression *)
+~NotExpression()
+evaluate(Context &) : bool
AndExpression
-_operand1 : BooleanExpression*
-_operand2 : BooleanExpression*
+AndExpression(BooleanExpression *, BooleanExpression *)
+~AndExpression()
+evaluate(Context &) : bool
OrExpression
-_operand1 : BooleanExpression*
-_operand2 : BooleanExpression*
+OrExpression(BooleanExpression *, BooleanExpression *)
+~OrExpression()
+evaluate(Context &) : bool

```
NotExpression::NotExpression(BooleanExpression* op1)
{
    _operand1 = op1;
}

bool NotExpression::evaluate(Context& aContext)
{
    bool operEval = !_operand1->evaluate(aContext);

    return operEval;
}

NotExpression::~NotExpression(void)
{
}
```

```
AndExpression::AndExpression(BooleanExpression* op1,
                             BooleanExpression* op2)
{
    _operand1 = op1;
    _operand2 = op2;
}

bool AndExpression::evaluate(Context& aContext)
{
    bool oper1Eval = _operand1->evaluate(aContext);
    bool oper2Eval = _operand2->evaluate(aContext);

    return oper1Eval && oper2Eval;
}

AndExpression::~AndExpression(void)
{
}
```

```
OrExpression::OrExpression(BooleanExpression* op1,
                           BooleanExpression* op2)
{
    _operand1 = op1;
    _operand2 = op2;
}

bool OrExpression::evaluate(Context& aContext)
{
    bool oper1Eval = _operand1->evaluate(aContext);
    bool oper2Eval = _operand2->evaluate(aContext);

    return oper1Eval || oper2Eval;
}

OrExpression::~OrExpression(void)
{
}
```

Context participant


Visual Paradigm Standard Edition(University of Pretoria)
Context
-nameValue : map<string, bool>
+Context(void)
+~Context(void)
+lookup(string) : bool
+assign(VariableExpression *, bool) : void

```
Context::Context(void)
{
}

Context::~Context(void)
{
}

bool Context::lookup(const std::string paramName) const
{
    if (nameValue.find(paramName) != nameValue.end())
        return nameValue.find(paramName)->second;

    return false;
}

void Context::assign(VariableExpression* anExpression,
                     bool xBoolValue)
{
    nameValue[anExpression->getName()] = xBoolValue;
}
```

```
int main() {
    BooleanExpression* expression;
    Context context;
    bool result;

    VariableExpression* x = new VariableExpression("X");
    VariableExpression* y = new VariableExpression("Y");
    VariableExpression* z = new VariableExpression("zValue");

    expression = new OrExpression(
        new AndExpression (new Constant(true), x),
        new AndExpression (y, new NotExpression(z)));

    context.assign(x,false);     context.assign(y,false);
    context.assign(z,false);    // false = 0, true = 1

    result = expression->evaluate(context);
    cout << "Overall result is "
         << (result == 0 ? "false": "true") << endl;
    return 0;
}
```