TP 2 : Design Patterns (C++)

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

Exercise I : Composite

We will re-implement the recursive structure of arithmetic expressions from the previous lab, but this time in C++. To do this, we give ourselves the interface below, which you will copy into an *expression.h* file :

```
#ifndef EXPR_H
#define EXPR_H

#include <string>

class Expression{
 public:
  virtual std::string print()=0;
};

#endif
```

This abstract interface for expressions declares a virtual method `print`, that builds a string representation for an expression. Our different kinds of expression will inherit from this interface. We will limit ourselves to constants and binary operations.

Q1 – Copy the interface below to a file *constant.h* and implement the corresponding class in a file *constant.cc*.

```
#ifndef CST_H
#define CST_H

#include <string.h>
#include "expression.h"

class Constant : public Expression {
 public:
  int value;

  Constant(int value);

  std::string print();
};
#endif
```

Be careful to :

- put the `#include "constant.h"` directive at the start of the file *constant.cc*
- check that your code compiles by doing `g++ -c constant.cc`

Q2 – Define a second child class *Binop* in a file *binop.cc*, with its header *binop.h*, which inherits of *expression*. The class *Binop* will have two operands (which are themselves arithmetic expressions, therefore of type `Expression*`) and an operator represented by a string. Don't forget to overload the method `print` and verify that your code compiles.

Q3 – Add to our expressions a method *size* allowing to measure the size (integer) of an expression. For that it is necessary to :

- add virtual method signature in the interface *expression.h*
- add a method signature in all interfaces that inherit from it.
- implement the corresponding methods in all child classes.

We will assume that the size of a constant is 1 and that the size of a binary operation is 1 + the sum of the sizes of its operands.

Example | size("1+3*2") = 5
size("42") = 1

Note : This notion of size corresponds to the number of nodes of the expression's AST.

Q4 – In the same way, add to our expressions a method *eval* allowing to evaluate a given expression.

Example | eval("1+3*2") = 7
eval("42") = 42

Exercise II : Factory

The factory design pattern makes it possible to instantiate objects whose class is derived from an abstract class. The exact class of the object is therefore unknown to the caller.

Q1 – Create in a file *main.cc* functions *mul add*, each of which takes two `Expression*` as parameter and which returns `Expression*`. Create a function `cst` which takes an `int` as a parameter and which returns a `Expression*`. Don't forget to include the directives :

```
#include "expression.h"
#include "constant.h"
#include "binop.h"
```

Q2 – Create a `main` function and verify that your code compile by doing :
`g++ constant.cc binop.cc main.cc -o eval`

Q3 – To test your implementation, copy the following code into your `main` method and verify that the `./eval` command displays "expression : (2*(3+18)) of size 5 is equal to 42".

```
  Expression *e = mul(cst(2), add(cst(3), cst(18)));
  cout << "expression: " << e→print()
       << " of size "<< e→size() << " is equal to "
       << e→eval() << endl;
```

Exercise III : Visitor

As you noticed in the first exercise, adding a processing that traverses the expression tree requires adding a method for every kind of expression. This has several drawbacks :

— The classes representing the different types of expression grow as treatments are added, which can make harder the maintainability of the code.

— All changes take place at the level of these classes which can give rise to conflicts if several people work together.

— A bug related to a processing can come from different files.

The purpose of the *visitor* design pattern is to solve these problems by separating the data from its processing. Thus, adding a new processing can be done *without modifying the pre-existing code*, but rather by adding new code dedicated to it. Let's re-write our arithmetic evaluator using a visitor. We do not recommend that you modify the code of your first two exercises but rather work on a copy in a new directory.

We give ourselves the following interface, which you will copy in a file *visitor.h* :

```
#ifndef VIS_H
#define VIS_H

class Constant;
class Binop;

/* The Visitor Interface declares a set of visiting methods corresponding to
    component classes. */
class Visitor {
  audience:
   virtual int Visit(Constant&element) = 0;
```

```
    virtual int Visit(Binop&element) = 0;
};

#endif
```

This interface defines the methods necessary to implement a processing that recursively traverses an AST and whose return value of return is an integer.

Q1 – Rather than having one method per use-case, the `Expression` class will now only declare a single general purpose method `virtual int Accept` which will take as parameter a visitor passed by address. Modify the interface *expression.h* Consequently. You will not forget to put the directive `#include "visitor.h"` at the beginning of your file.

Q2 – Modify the *Constant* and *Binop* classes so they implement the `Accept` method. Note that this method simply calls the `Visit` method of the visitor passed as a parameter. You will not forget to check that your code compiles by doing `g++ -c constant.cc binop.cc`

Q3 – Create an *Evaluator* class in a file *evaluator.cc* (as well as its header *evaluator.h*), which will implement the *Visitor* interface. His two `Visit` methods will evaluate the expressions passed in parameter.

Q4 – Do the same for the measuring of the size of an expression.

Q5 – Test your implementation by verifying that the following code produces the display : "expression of size 5 is equal to 42"

```
int main() {
    Expression *e = mul(cst(2), add(cst(3), cst(18)));
    Measure *m = new Measure();
    Evaluator *ev = new Evaluator();
    cout << "expression of size " << e→ Accept(*m) <<" is equal to " << e→ Accept(*ev
    ) << endl;
    return 0;
}
```