TP 2 : Design Patterns (C++)

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

Exercice I : Composite

Nous allons ré-implémenter la structure récursive des expressions arithmétiques du TP précédent, mais cette fois-ci en C++. Pour cela on se donne l'interface ci-dessous, que vous copierez dans un fichier *expression.h* :

```
#ifndef EXPR_H
#define EXPR_H

#include <string>

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

#endif
```

Cette interface abstraite pour les expressions déclare une méthode virtuelle `print`, permettant de convertir une expression en chaine de caracteres. Nos différents types d'expression hériteront donc de cette interface. On se limitera aux constantes et aux opérations binaires.

Q1 – Copiez l'interface ci-dessous dans un fichier *constant.h* et implémentez la classe correspondante dans un fichier *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
```

Vous penserez à :

- mettre la directive `#include "constant.h"` au début du fichier *constant.cc*
- vérifier que votre code compile en faisant `g++ -c constant.cc`

Q2 – Définissez une seconde classe fille *Binop* dans un fichier *binop.cc*, avec son header *binop.h*, qui hérite d'*expression*. La classe *Binop* aura deux opérandes (qui sont elles-memes des expressions arithmétiques, donc de type `Expression*`) et un opérateur représenté par une chaine de caractères. Vous n'oublierez pas de surcharger la méthode `print` et de vérifier que votre code compile.

Q3 – Ajoutez à nos expressions une méthode *size* permettant de mesurer la taille (entière) une expression. Pour cela il faut :

- ajouter une signature de méthode virtuelle dans l'interface *expression.h*
- ajouter une signature de méthode dans toute les interfaces qui en héritent.
- implémenter les méthodes correspondantes dans toute les classes filles.

On supposera que la taille d'une constante est de 1 et que la taille d'une opération binaire est 1 + la somme des tailles de ses opérandes.

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

Note : Cette notion de taille correspond au nombre de noeuds de l'ast de l'expression.

Q4 – De la même manière, ajoutez à nos expressions une méthode *eval* permettant d'évaluer une expression donnée.

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

Exercice II : Fabrique

Le patron de conception fabrique permet d'instancier des objets dont la classe est dérivée d'une classe abstraite. La classe exacte de l'objet n'est donc pas connue par l'appelant.

Q1 – Dans un fichier *main.cc*, créez des fonctions *mul add*, qui prennent chacun deux `Expression*` en paramètre et qui retourne des `Expression*`. Créez un function `cst` qui prend un `int` en paramètre et qui retourne une `Expression*`. Vous n'oublierez pas d'inclure les directives :

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

Q2 – Créez une fonction `main` et vérifiez que votre code compile en faisant :
`g++ constant.cc binop.cc main.cc -o eval`

Q3 – pour tester votre implémentation, copiez le code suivant dans votre méthode `main` et vérifiez que la commande `./eval` produit bien l'affichage "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;
```

Exercice III : Visiteur

Comme vous l'avez remarqué dans le premier exercice, ajouter un traitement qui parcours l'arbre d'expressions nécessite d'ajouter une méthode par type d'expression. Ceci présente plusieurs inconvénients :

- Les classes représentant les différents types d'expression grossissent au fur et à mesure de l'ajout des traitements, ce qui peut compliquer la maintenabilité du code.
- Tous les changements ont lieu au niveau de ces classes ce qui peut donner lieu à des conflits si on travaille à plusieurs.
- Un bug lié à un traitement peut provenir de plusieurs fichiers différents.

Le but du design pattern *visitor* est de résoudre ces problèmes en séparant la donnée de son traitement. Ainsi, l'ajout d'un nouveau traitement pourra se faire *sans toucher au code pré-éxistant*, mais plutôt en ajoutant du code nouveau dédié. Nous allons ré-écrire notre evaluateur arithmétique en utilisant un visiteur. Nous vous déconseillons de modifier directement le code de vos deux premiers exercices mais de travailler sur une copie dans un nouveau repertoire.

On se donne l'interface suivante, que vous copierez dans un fichier *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 {
 public:
  virtual int Visit(Constant& element) = 0;
  virtual int Visit(Binop& element) = 0;
};

#endif
```

Cette interface définit les méthodes nécessaire à implémenter pour un traitement qui parcours récursivement un AST et dont la valeur de retour est entière.

Q1 – Plutot que d'avoir une méthode par traitement, la classe `Expression` ne déclarera à présent qu'une unique méthode générique `virtual int Accept` qui prendra en paramètre un visiteur passé par adresse. Modifiez l'interface *expression.h* en conséquence. Vous n'oublierez pas de mettre la directive `#include "visitor.h"` au début de votre fichier.

Q2 – Modifiez les classes *Constant* et *Binop* pour leur faire implémenter la méthode `Accept`. Notez que celle ci ne fait qu'appeler la méthode `Visit` du visiteur passé en paramètre. Vous n'oublierez pas de vérifier que votre code compile en faisant `g++ -c constant.cc binop.cc`

Q3 – Crééz une classe *Evaluator* dans un fichier *evaluator.cc* (ainsi que son header *evaluator.h*), qui implémentera l'interface *Visitor*. Ses deux méthodes `Visit` procéderont à l'évaluation des expressions passées en paramètre.

Q4 – Faites de même pour la mesure de la taille d'une expression.

Q5 – Testez votre implémentation en vérifiant que le code suivant produit l'affichage : "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;
}
```