TME 10 et 11 : Simplification d'expressions arithmétiques

Objectifs pédagogiques : polymorphisme et surcharge, composite et visiteur

L'objectif de ce TME est d'effectuer des traitements (calculs, simplifications, dérivations) sur des expressions arithmétiques, permettant ainsi d'illustrer les notions de surcharge et de polymorphisme paramétrique. Ce TME est basé sur le partiel de l'année 2012–2013 proposé par Emmanuel Chailloux (épreuve individuelle en 3 heures) et son système de correction automatique.

Nous vous fournissons ici les jeux de test utilisés lors de la correction afin de vous permettre de vous auto-évaluer.

L'énoncé couvre les deux séances ; il n'y a pas d'énoncé séparé pour chaque TME. Vous ferez un rendu à votre responsable comme d'habitude à la fin de chaque TME, où vous préciserez la question atteinte à la fin de la séance.

Un objectif à viser pour la séance de TME 10 est de compléter les questions 1 à 7, qui présentent la migration d'une architecture basée sur Composite à une architecture basée sur Visiteur. Dans ces questions, nous ne traitons pas de façon complète les variables apparaissant dans les expressions.

Les questions 8 à 13 mettent en place un moteur de calcul symbolique plus complet (couvrant aussi les variables), en combinant des visiteurs entre eux. Il permet notamment de calculer la dérivée partielle d'une expression et de simplifier les expressions.

Le squelette de projet `Expr` est disponible sur le serveur GitLab. Comme pour les projets de TME précédents, vous devez commencer par en faire un *fork* par binôme et à l'importer sous Eclipse. Toutes les classes fournies et demandées sont dans le package `pobj.expr`, et les tests unitaires dans `pobj.expr.test`.

10.1 Expressions arithmétiques

Nous souhaitons représenter des expressions arithmétiques, par exemple :

1. $e_1 = (2 + 3) * 4$
2. $e_2 = (x + 3) * (x + 4)$
3. $e_3 = (x + 10) * (y + -8)$

Les constantes (entières) seront représentées par des instances de la classe `Constant` ; les opérateurs $+$ et $*$ correspondront respectivement à des instances des classes `Add` et `Mult` ; une variable est représentée par une instance de la classe `Var` dont un attribut (variable d'instance) contient la chaîne de caractères correspondant à son nom.

Toutes ces classes implémentent l'interface `Expression` et permettent de représenter un arbre de syntaxe de l'expression. L'interface `Expression` est initialement vide (pas de méthode déclarée), mais permet d'offrir un typage commun à tous les nœuds d'un arbre d'expression ; ainsi, les opérandes d'un `Add` sont des `Expression` en terme de typage. L'interface sera enrichie par la suite.

Question 1

⇒ **Écrivez la classe `Constant`** implantant `Expression`, munie d'un attribut entier avec un *getter* `getValue` et des méthodes standard `equals` et `toString`. La classe possède un constructeur ayant un argument entier (sa valeur) et un constructeur sans argument (indiquant une valeur nulle). La méthode `toString` affichera la valeur stockée. Exécutez le jeu de test `TestQ1` fourni pour valider votre implantation.

Question 2

⇒ **Écrivez la classe `Var`** implantant `Expression`, munie d'un attribut stockant son nom (de type `String`) immuable (`final`), d'un *getter* `getName`, d'un constructeur prenant une `String` en argument

et des méthodes standard `equals` et `toString`. La méthode `toString` affichera le nom de la variable. L'égalité des variables est l'égalité de leurs noms. Exécutez le jeu de test `TestQ2` fourni.

Question 3

⇒ **Écrivez les classes `Add` et `Mult`** selon le *Design Pattern* Composite.

Suivant le *Design Pattern* Composite, les classes `Add` et `Mult` implémentent `Expression`, et leurs fils sont eux même des objets implantant `Expression`. Les constantes et les variables sont aussi des expressions, sans sous-expression.

Les classes `Add` et `Mult` présentent de fortes similarités. Pour factoriser le code, nous recommandons de définir une classe abstraite `BinOp` portant les attributs `left` et `right` ainsi que les *getters* associés ; `Add` et `Mult` pourront alors étendre cette classe. Ces deux attributs `left` (pour fils gauche) et `right` (pour fils droit) correspondent aux deux opérandes d'une opération binaire (comme l'addition ou la multiplication). Ces attributs sont immuables (`final`) et sont typés `Expression`.

Pour la méthode `toString`, nous demandons de produire un affichage infixe standard. Pour la classe `Add`, vous placerez des parenthèses autour de l'opération, comme `"( 2 + x )"`. Pour la classe `Mult`, vous vous contenterez d'un signe '∗', comme `"3 * x"`. Ces classes ont un constructeur avec deux arguments (`left` et `right`) et des *getters*. La méthode `equals` n'est pas demandée.

Exécutez le jeu de test `TestQ3` fourni.

Question 4

⇒ **Définissez les expressions** e_1, e_2, e_3 de l'introduction. Pour cela, écrivez une classe `Question4` qui porte des opérations `public static Expression e1()`, `e2()` et `e3()` : une pour chaque expression. Assurez-vous de valider le test `TestQ4` fourni.

Question 5

⇒ **Ajoutez une opération `int eval()`** à l'interface `Expression`. Implantez l'opération dans toutes les classes qui réalisent `Expression`. `Add` et `Mult` suivent la sémantique de l'addition et de la multiplication. Les constantes s'évaluent en leur valeur. Pour les variables, vous signalerez pour l'instant une exception standard `UnsupportedOperationException`. `eval()` signale donc une exception si l'arbre d'expression contient des variables.

Exécutez le jeu de test `TestQ5` fourni qui teste l'évaluation.

10.2 Passage aux Visiteurs

Nous cherchons à présent à nous doter de traitements extensibles sur les expressions arithmétiques en utilisant le modèle de conception « Visiteur ».

Le problème de l'approche suivie jusqu'à maintenant, qui correspond au *Design Pattern* Composite, est que chaque nouveau besoin induit une nouvelle opération dans `Expression` (nous avons traité `toString` et `eval` pour l'instant), et donc la modification de l'interface et de toutes les classes qui l'implantent. Cette approche n'est pas une bonne *réutilisation* puisqu'elle oblige à modifier des classes qui jouaient leur rôle jusque là.

Dans le modèle des visiteurs, chaque classe visiteur correspond à un traitement spécifique sur les expressions arithmétiques (e.g., `toString` ou `eval`). Le visiteur regroupe dans une seule classe le traitement sur l'ensemble des classes d'expression (i.e., implantant `Expression`), avec une méthode par classe d'expression. Le modèle est illustré ici d'abord en adaptant l'exemple du visiteur d'affichage d'expressions arithmétiques (`toString`). Au fil de l'énoncé nous implanterons plusieurs autres visiteurs de calcul sur les expressions : un visiteur d'évaluation des expressions sans variable, un visiteur d'évaluation d'expressions avec variables, un visiteur de test de constante, un visiteur de simplification d'expressions et un visiteur de dérivation symbolique.

Voici l'interface `IVisitor<T>` que devra respecter tout visiteur d'expressions arithmétiques :

IVisitor.java

```
package pobj.expr;

public interface IVisitor<T> {
   T visit(Constant c);
   T visit(Add e);
   T visit(Mult e);
   T visit(Var v);
}
```

Il y a bien une méthode `visit` pour chaque classe concrète d'expression arithmétique. Par ailleurs, différents visiteurs retourneront différents types (chaîne pour le visiteur d'affichage, `Integer` pour un visiteur d'évaluation, etc.). L'interface est donc générique, paramétrée par le type de retour `T`.

Afin d'aiguiller l'exécution vers la bonne méthode d'un visiteur, chaque implantation concrète d'`Expression` possédera une méthode `accept`, qui prend en argument l'instance de visiteur, appelle la méthode `visit` du bon type, et retourne sa valeur. Cette méthode, ajoutée dans l'interface `Expression` et dans toutes les classes concrètes, a pour signature `public<T> T accept(IVisitor<T>)`, indiquant qu'elle est également générique en `T`, prend un visiteur de type `IVisitor<T>` en argument, et renvoie une valeur de type `T`.

⇒ **Ajoutez la méthode `accept` dans `Expression`**. L'interface `IVisitor` est déjà fournie. Vous implanterez l'opération `accept` dans toutes les `Expression` concrètes : elle invoque la méthode `visit` du visiteur en lui passant la référence `this`.

Notons que nous avons modifié les classes `Expression` pour l'implantation du motif visiteur. Néanmoins, nous n'aurons plus à les modifier par la suite : chaque ajout de fonctionnalité sera réalisé par une nouvelle classe visiteur indépendante de nos `Expression` ; nous avons donc atteint notre objectif de réutilisation.

Question 6

Pour vous familiariser avec l'écriture de visiteurs, voici un exemple d'un visiteur de conversion en chaînes de caractères appelé `VisitorToString`. Le résultat de ce visiteur est une chaîne de caractères ; pour cela, la classe `VisitorToString` implante l'interface `IVisitor<String>`. La méthode `visit` est surchargée, permettant d'effectuer un travail spécifique pour chaque classe concrète sous-classe d'`Expression`. Si nécessaire pour le calcul, le visiteur peut parcourir les sous-expressions qui composent l'expression et obtenir leur valeur générée par le visiteur, avant de construire sa valeur de retour (comme pour `Add` et `Mult`). Ceci est réalisé par un appel à `accept` sur chaque sous-expression, en passant le visiteur courant (`this`) en argument.

VisitorToString.java

```
package pobj.expr;

public class VisitorToString implements IVisitor<String> {
   public String visit(Constant c) {
      return Integer.toString(c.getValue());
   }
   public String visit(Var v) {
      return v.getName();
   }
   public String visit(Add a) {
      String s1 = a.getLeft().accept(this);
      String s2 = a.getRight().accept(this);
      return "( "+ s1 + " + " + s2 + " )";
   }
   public String visit(Mult a) {
```

```
        String s1 = a.getLeft().accept(this);
        String s2 = a.getRight().accept(this);
        return s1 + " * " + s2;
    }
}
```

Et voici un exemple de code qui utilise le visiteur :

```
VisitorToString vts = new VisitorToString();
String e1str = Question4.e1().accept(vts);
```

⇒ **Ajoutez ce visiteur concret à votre projet.** Assurez-vous de passer au moins le test `TestQ6`.

Question 7

⇒ **Écrivez une classe `VisitorEval`** qui implante l'interface `IVisitor<Integer>` et qui calcule la valeur d'une expression sans variable.

Ce visiteur concret va permettre de voir comment réimplémenter le mécanisme d'évaluation défini en question 5 dans le cadre du *Design Pattern* Visiteur. Si un tel visiteur rencontre une instance de `Var`, il déclenche l'exception standard `UnsupportedOperationException`.

Remarquons que ce visiteur concret se substitue à `eval()`, que nous pourrions à présent supprimer de nos classes et de l'interface `Expression`, les rendant plus simples.

⇒ Assurez-vous de valider au moins les tests fournis dans `TestQ7`, qui utilisent ce nouveau mécanisme d'évaluation au lieu de la méthode `eval()`.

Rendu intermédiaire

À ce stade, nous avons obtenu un moteur d'évaluation des expressions assez limité, car il ne traite pas les variables. La suite de l'énoncé va développer ce moteur ; selon le temps disponible, les questions suivantes peuvent être traitées au TME 10 ou au TME 11.

N'oubliez pas de faire un **rendu pour le TME 10**, avec un *push* dans le GitLab suivi de la création d'un *tag*.

Vous indiquerez votre état d'avancement dans le champ « Release notes » associé au *tag* (questions traitées, nombre de tests passés).

10.3 Environnements et variables

Question 8

Pour les formules contenant des variables, il est nécessaire de pouvoir les calculer dans un certain environnement. Un environnement est formé de couples liant une variable à une valeur entière. Nous allons utiliser pour cela la classe `Map<String,Integer>` associant au nom des variables leur valeur respective.

Nous cherchons à construire les environnements suivants :

1. env_1 est l'environnement vide ;
2. env_2 est l'environnement qui associe `10` à la variable "x" et `20` à "y" ;
3. env_3 est l'environnement qui associe `9` à la variable "z".

Pour cela, il vous est demandé de fournir une classe `Question8` qui définit des méthodes statiques `env1()`, `env2()` et `env3()` qui retournent respectivement les environnements env_1, env_2 et env_3 décrits ci-dessus.

Assurez-vous de valider au moins le test `TestQ8` fourni.

Question 9

Il vous est demandé d'écrire une classe `VisitorEvalVar`, sous-classe de `VisitorEval`, permettant l'évaluation, sans exception, d'une expression contenant des variables dans un environnement donné. La classe `VisitorEvalVar` possède un constructeur prenant un environnement (`Map<String,Integer>`) comme paramètre. Nous pourrons ainsi construire de tels visiteurs de la manière suivante :

```
VisitorEvalVar vev1 = new VisitorEvalVar(Question8.env2());
Integer resultat = Question4.e2().accept(vev1);
```

N'oubliez pas de tester votre classe au moins sur les tests `TestQ9` fournis.

Question 10

Pour la question suivante, nous voulons vérifier si une expression est constante (i.e., son arbre d'expression ne contient aucune variable). Nous voulons également savoir, dans le cas où l'expression est constante, si elle est toujours égale à 0, ou à 1.

Pour décider si une expression est constante, on pourra écrire une classe `VisitorConstant` qui réalise l'interface `IVisitor<Boolean>` et rend vrai pour les arbres d'expressions dans lesquels il n'y a aucune variable.

Pour évaluer une expression constante, on réutilisera le visiteur `VisitorEval` déjà développé en question 7.

Complétez la classe `Question10` suivante, afin de valider les tests fournis dans `TestQ10` :

```
public class Question10 {
   // rend vrai si e est un arbre d'expression constant
   public static boolean isConstant(Expression e) {
   }
   // rend la valeur d'une expression constante
   // signale une exception si l'expression n'est pas constante
   public static int evalConstantExpression (Expression e) {
   }
}
```

10.4 Manipulation d'expressions

Question 11

Nous cherchons à simplifier les expressions arithmétiques selon les règles suivantes :

- évaluation des opérations sur les constantes :

$$1 + 3 \to 4$$

$$3 * 10 \to 30$$

- élément neutre pour les opérateurs $+$ et $*$:

$$e + 0 \to e$$

$$0 + e \to e$$

$$e * 1 \to e$$

$$1 * e \to e$$

- élément absorbant pour la multiplication :

$$e * 0 \to 0$$

$$0 * e \to 0$$

Si un opérateur binaire ($+$ ou $*$) est une constante (au sens de la question 10), nous pouvons substituer au nœud courant un objet `Constant` qui porte l'évaluation de l'expression (calculée en utilisant la question 10).

Chaque opération (`Add` et `Mult`) possède un élément neutre (0 pour l'addition et 1 pour la multiplication). De plus, la multiplication possède un élément absorbant : 0.

Il vous est demandé d'écrire une classe `VisitorSimplify` qui implante l'interface `IVisitor<Expression>` et permet de calculer une expression simplifiée en fonction des règles de simplification proposées. Un tel visiteur retourne une *nouvelle* expression correspondant à la simplification de l'expression qu'il visite.

```
VisitorSimplify vs = new VisitorSimplify();
Expression simple = Question4.e1().accept(vs);
```

N'oubliez pas de tester votre classe à l'aide des tests fournis dans `TestQ11`.

Question 12

Nous souhaitons maintenant effectuer une dérivée symbolique des expressions par rapport à une variable. Nous rappelons les formules pour une dérivée $(e)'$ d'une expression e par rapport à la variable x :

- constante :

$$(c)' = 0$$

- variable :

$$(x)' = 1$$

$$(y)' = 0$$

- pour les opérateurs binaires, addition et multiplication :

$$(e_1 + e_2)' \;=\; (e_1' + e_2')$$

$$(e_1 * e_2)' \;=\; ((e_1 * e_2') + (e_1' * e_2))$$

Il vous est demandé d'écrire une classe `VisitorDerive` qui implante l'interface `IVisitor<Expression>` permettant le calcul symbolique de la dérivée d'une expression par rapport à une variable. Un tel visiteur retourne une nouvelle expression correspondant à la dérivée de l'expression qu'il visite. Il est nécessaire, à la construction, d'indiquer par rapport à quelle variable s'effectue la dérivée, comme dans l'exemple suivant :

```
VisitorDerive vd = new VisitorDerive(new Var("x"));
Expression resultat = Question4.e3().accept(vd);
```

Dans cette question, on ne cherche pas (encore) à simplifier l'expression ainsi obtenue.

N'oubliez pas de tester votre classe à l'aide de `TestQ12` fourni.

Question 13

Comme le calcul de la dérivée symbolique construit de grosses expressions peu lisibles, il est nécessaire de les simplifier. Pour cela, on demande de définir une fonction `compose` générique qui prend

deux visiteurs f et g, et une expression e, et qui retourne le résultat de la composition des deux visiteurs en appliquant d'abord g puis f sur le résultat de g. Vous pourrez alors combiner les visiteurs des deux questions précédentes pour dériver et simplifier le résultat dérivé.

Pour cela nous vous demandons de compléter la classe `Question13` suivante, de façon à valider les tests dans `TestQ13` :

```
public class Question13 {
   public static <T> T compose(IVisitor<T> f, IVisitor<Expression> g, Expression e) {
      // à compléter...
   }
}
```

Rendu final

Faites un **rendu pour le TME 11**, avec un *push* dans le GitLab suivi de la création d'un *tag*.