Patron de conception : le visiteur
Corrigé

Exercice 1 : Modélisation d'expressions arithmétiques
Soit $\mathcal{G}$ la grammaire des expressions arithmétiques suivante :

1. $\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E}$
2. $\mathrm{E} \rightarrow \mathrm{E} * \mathrm{E}$
3. $\mathrm{E} \rightarrow - \mathrm{E}$
4. $\mathrm{E} \rightarrow$ ident
5. $\mathrm{E} \rightarrow$ constante

1.1. Donner quelques exemples d'expressions.
Solution : Peut-être que l'énoncé n'est pas très clair. Que signifie E ? Que signifient les lignes numérotées ?

E est un non terminal (par convention écrit en majuscule). Ceci signifie qu'il représente un morceaux de texte (appelé E, de type E, etc.). Les textes que représente E s'obtiennent en remplaçant E par la partie droite des règles où E est à gauche. C'est une **dérivation**.

L'application de cette dérivation peut conduire à faire apparaître de nouveau non terminaux (ici des E). C'est le cas des trois premières règles. Ces nouveaux E devront être, à leur tour, dérivés.

On obtient le texte quand il n'y a plus de non terminaux, ici plus de E.

Quels sont les textes que l'on peut donc atteindre ?

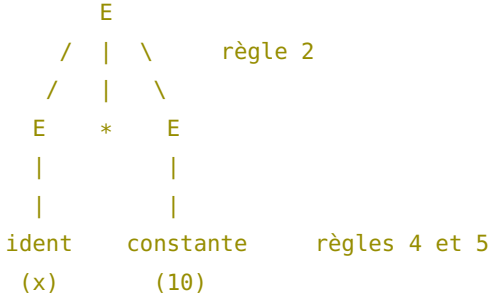
Certains commencent tout de suite par donner des exemples d'expressions compliquées. On peut bien sûr en donner mais il est préférable de commencer par des exemples simples, en commençant par n'utiliser qu'une seule règle.

```
10
x
- 5
x * 10
a + b
3 + x * y
- x + y + z + t * 10
```

Prenons l'exemple de `x * 10` et montrons qu'on peut le dériver à partir de E :

```
E -> E * E                      (règle 2)
    -> ident(x) * E             (règle 4)
    -> ident(x) * constante(10) (règle 5)
```

On peut aussi le représenter sous forme d'un arbre :



Quand on fait un parcours infixe de cet arbre, on retrouve bien le texte de départ : $x * 10$.

Est-ce l'expression $x - 3$ fait partie des expressions possibles ?

Non, car on ne peut pas la dériver à partir des règles proposées.

En fait, on a que le « - » unaire (règle 3) et pas le moins binaire. On sera donc obligé de l'écrire sous la forme $x + -3$.

1.2. Proposer un diagramme de classe permettant de représenter une expression quelconque correspondant à cette grammaire.

Solution :

Ici, on veut utiliser le diagramme de classe pour spécifier ce qu'est une expression.

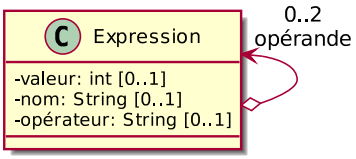
Un indice : on peut faire un diagramme de classe avec une seule classe ou avec beaucoup de classes.

On peut envisager une première solution avec une seule classe, naturellement cette classe s'appelle Expression puisque ce sont les expressions que l'on veut modéliser. Elle doit contenir les caractéristiques de toutes les expressions possibles :

- une valeur (pour constante)
- un nom (pour ident)
- un opérateur unaire - (par exemple un attribut) et une autre expression (relation réflexive)
- un opérateur binaire + ou * (par exemple un attribut de type char ou String) et deux expressions (opérande gauche et opérande droite)

Ces informations sont optionnelles et dépendent de la nature de l'expression. Par exemple n'aura que la valeur de définie et le reste ne le sera pas. Tous les attributs sont donc optionnels (multiplicité 0..1 en UML).

Voici le diagramme de classe correspondant sur lequel un opérateur peut être unaire ou binaire et les opérandes sont sont représentés avec une multiplicité 0..2.



Une autre solution (certainement préférable) consiste à définir plusieurs classes en ayant un objectif clair pour chacune. On a bien sûr une première classe qui est Expression. Cependant, comme ne sait a priori rien sur une expressions, ce sera une notion abstraite (classe abstraite ou interface).

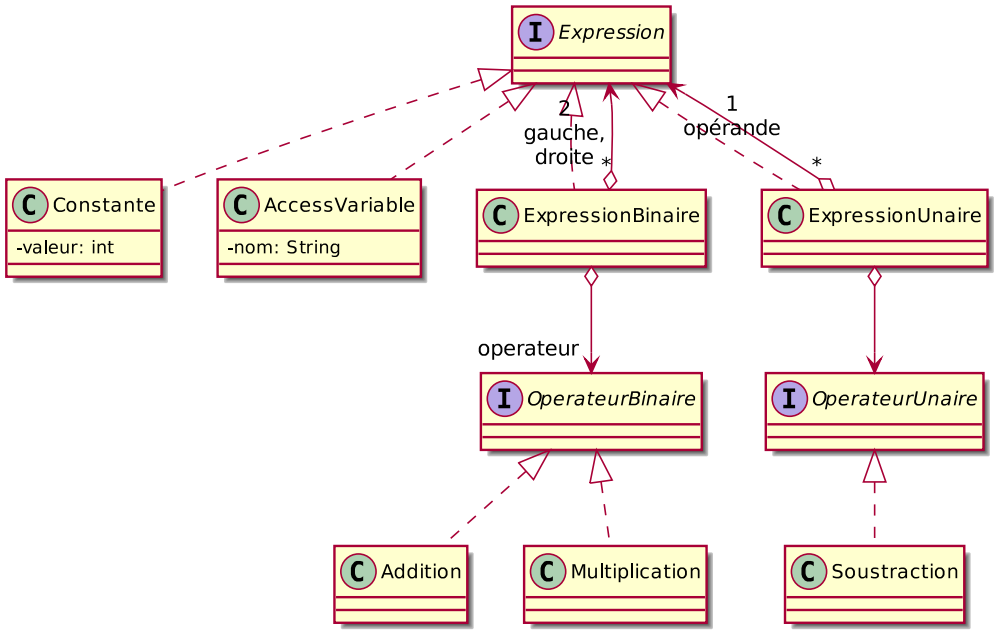
Les différentes règles nous permettent de déduire différents types d'expression :

- une Constante avec une valeur
- un ident (AccesVariable) avec un nom
- la somme de deux expressions gauche et droite
- le produit avec deux expressions gauche et droite
- l'opposé d'une expression.

On remarque que Somme et Produit ont toute deux relations vers Expression pour représenter les opérandes à gauche et à droite. On pourrait les factoriser en faisant une classe abstraite. Et mieux, plutôt que de laisser la sous-classe définir l'opérateur, on pourrait avoir une classe concrète OpérateurBinaire avec trois relations, une sur l'opérateur binaire (OpérateurBinaire) qui pourra être Addition, Multiplication, etc. et un une sur chaque opérande, gauche et droite.

Bien sûr d'autres modélisation seraient possibles...

On retrouve les mêmes informations que dans la version 1 mais chacune sur une classe spécifique. L'ensemble est donc plus facile à comprendre et sera plus facile à manipuler.



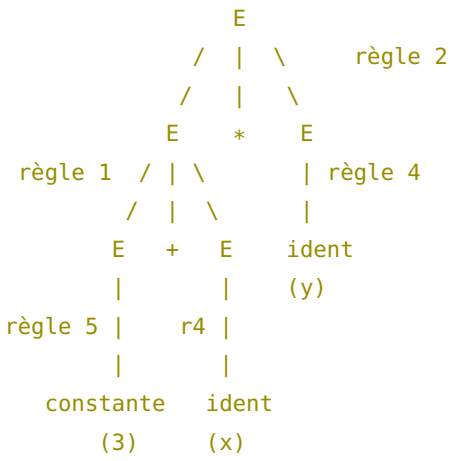
La première solution peut paraître plus simple mais son exploitation est difficile car elle oblige à faire de nombres tests pour savoir ce qu'est réellement l'expression. Par exemple, est ce que les opérateur sont définis ? est-ce que le nom est défini ?

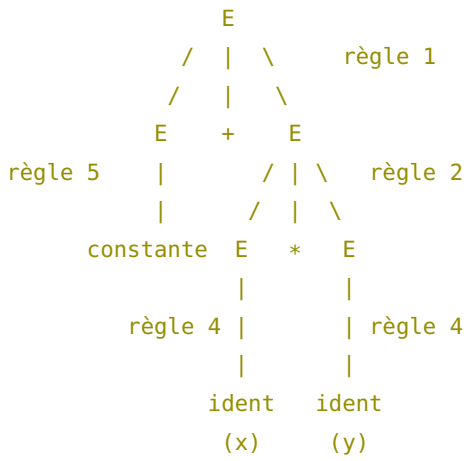
La deuxième solution conduit à un plus grand nombre de classes mais on sait pour chaque classe ce qu'elle représente et donc comment la traiter.

1.3. Dessiner un ou plusieurs diagrammes d'objet qui correspondent à l'expression 3 + x * y.

Solution : Deux solutions. La grammaire est donc ambiguë car plusieurs arbres de dérivation existent. L'une correspond à (3+x)*y et l'autre à 3+(x*y). C'est la deuxième qui est naturelle (qui correspond à la priorité usuelle des opérateurs).

Remarque : Il est intéressant de noter que quelque soit le diagramme de classe proposé, on a le même nombre d'objets (aux opérateurs près). Mais avec le deuxième diagramme de classe, on a des objets plus précis.





On pourrait alors construire les objets correspondant aux feuilles et aux noeuds de ces arbres

de dérivation. Voici la classe Java correspondante (en utilisant les constructeurs naturels des classes identifiées).

```java
/**
  * Produire l'expression donnée en exemple dans le sujet.
  */
public class ExempleSujet {

    // Volontairement, toutes les expressions sont définies comme des attirbuts
    // publics.  ces expressions étant inaltéables, on est sûr qu'elles ne
    // seront pas modifiées (les attributs sont constants).

    static final Multiplication multiplication = new Multiplication();
    static final Addition addition = new Addition();
    static final Constante c3 = new Constante(3);
    static final AccesVariable vx = new AccesVariable("x");
    static final AccesVariable vy = new AccesVariable("y");

    static final ExpressionBinaire xPlusY =
        new ExpressionBinaire(multiplication, vx, vy);
    static final ExpressionBinaire eNaturelle =
        new ExpressionBinaire(addition, c3, xPlusY);

    static final ExpressionBinaire c3plusX =
        new ExpressionBinaire(addition, c3, vx);
    static final ExpressionBinaire eAutre =
        new ExpressionBinaire(multiplication, c3, vy);

}
```

Exercice 2 : Exploitation du modèle objet des expressions

Intéressons nous à quelques exploitations des expressions ainsi modélisées.

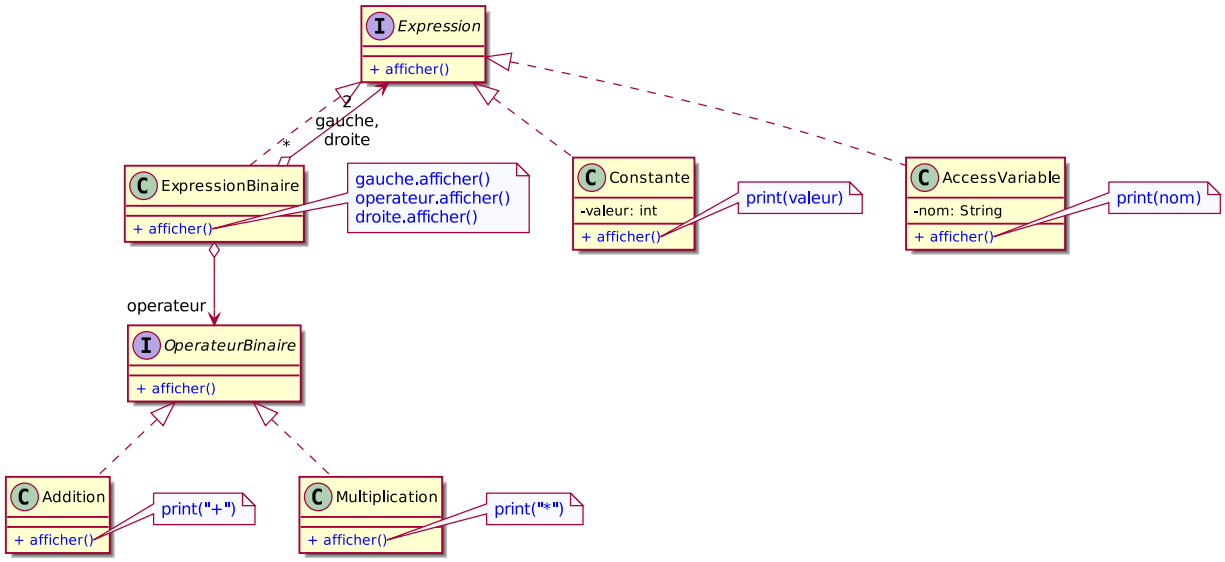
2.1. *Affichage classique (infixe).* Nous souhaitons réaliser une opération d'affichage d'une expression correspondant au diagramme de classe précédent.

2.1.1. Compléter le diagramme de classe.

Solution : On veut pourvoir afficher une expression. Naturellement, en objet, on s'attend donc à avoir une méthode `afficher()` sur `Expression`.

Au niveau de `Expression`, on ne sait pas écrire son code. Elle sera donc abstraite et définie sur les sous-classes ou réalisations de `Expression`. Notons, qu'elle sera aussi abstraite sur `OperateurBinaire` et `OperateurUnaire`.

On a fait disparaître les expressions unaires du diagramme de classe pour réduire sa taille. Elles se ressemblent beaucoup aux expressions binaires...

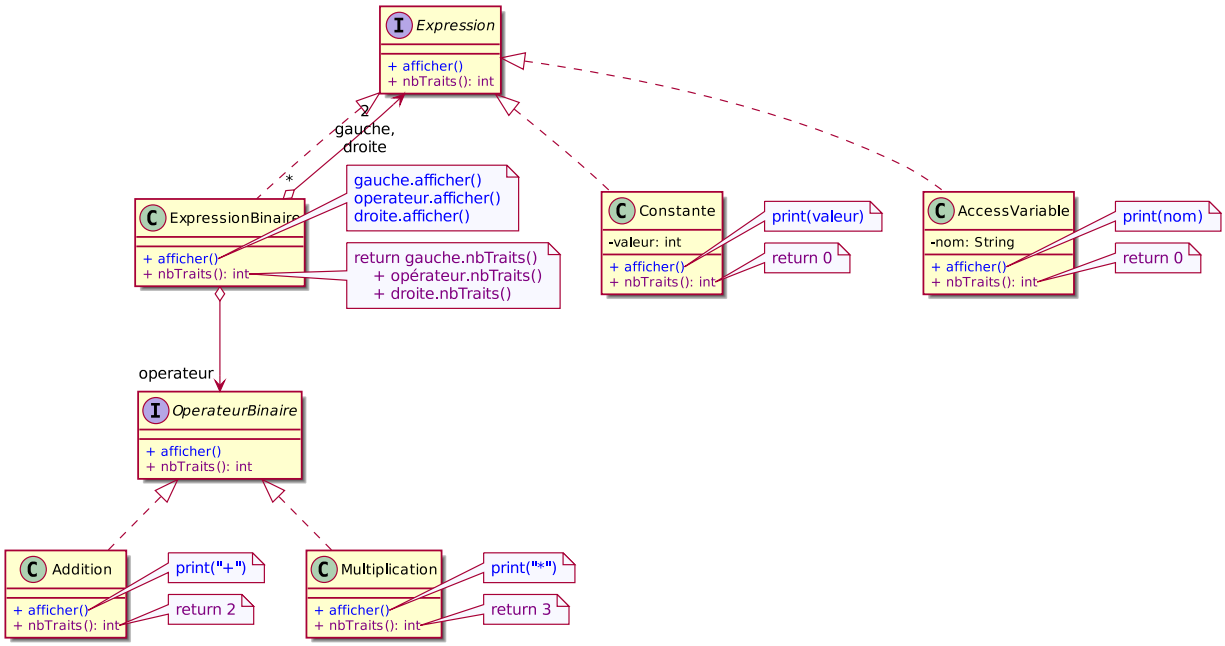


2.1.2. Utiliser un diagramme de séquence pour expliquer comment afficher l'expression de la question 1.3.

2.2. *Nombre de traits des opérateurs d'une expression.* Indiquer, en complétant le diagramme de classe, comment calculer le nombre de traits utilisés par les opérateurs d'une expression. La soustraction compte pour un trait, l'addition pour deux et la multiplication pour 3.

Solution :

Le principe est le même : on définit une méthode `nbTraits()` sur Expression qui sera réalisée sur les sous-types et les autres classes du modèle. On arrive alors au diagramme suivant.

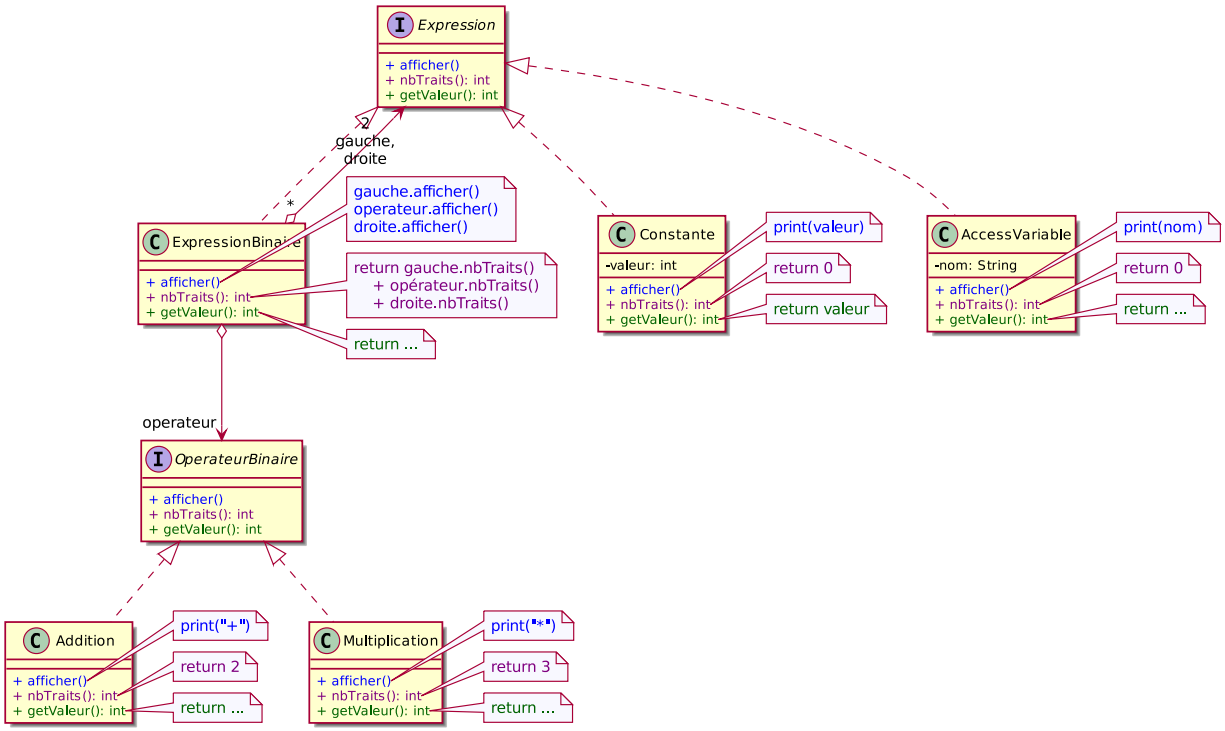


2.3. *Valeur d'une expression.* Indiquer, en complétant le diagramme de classe, comment calculer la valeur d'une expression. On suppose que les valeurs des variables sont disponibles dans un

environnement qui associe à chaque variable sa valeur.

Solution :

On applique le même principe : une méthode `getValeur()` sur Expression et les autres classes du modèle.



2.4. *Critique de l'approche choisie.* Maintenant que nous avons réalisé plusieurs traitements sur les activités, essayons de prendre du recul.

2.4.1. Expliquer comment définir un nouveau traitement sur les expressions. Le traitement pourrait être calculer le nombre d'opérateurs d'un expression, calculer le nombre de variables utilisées, simplifier une expression, etc.

Solution : Définir une méthode polymorphe sur l'ensemble des classes du diagramme de classe.

2.4.2. Lister les critiques que l'on peut faire concernant l'approche suivie.

Solution : Pour chaque nouveau traitement considéré, il faut définir une nouvelle méthode polymorphe définie sur l'ensemble de la structure de classes :

- Un traitement est éclaté, éparpillé, sur l'ensemble du diagramme de classe (problème de compréhension et de maintenance).
- Chaque classe est polluée par chaque traitement.
- Pour ajouter un nouveau traitement, il faut pouvoir modifier toutes les classes (peu extensible).

Exercice 3 : Le patron Visiteur

Proposer une nouvelle manière de définir un traitement. Cette solution doit répondre aux problèmes identifiés dans l'exercice précédent.

Solution :

1. Définir une classe par traitement
2. Y regrouper les opérations (et données) associées à ce traitement.
   On doit mettre explicitement en paramètre la classe des expressions (Constante, AccesVariable, etc.) puisqu'on a déplacé la méthode. Voir le code ci-après.
3. Il faut alors révéler des éléments qui n'étaient jusqu'à présent que privés : la valeur d'une variable, le nom d'un accès à une variable, les opérandes et l'opérateur d'une expression binaire ou unaire...
4. Comment faire pour écrire afficher(ExpressionBinaire) ?
   Il faut faire un test sur le type des expressions correspondant à opérateur gauche et droit !

```java
// Version naïve : exemple à ne pas suivre
public class Afficheur {
    // On regroupe ici les méthodes éparpillées sur les différentes classes

    public void afficher(Constante c) {
        System.out.print(c.getValeur());
    }

    public void afficher(AccesVariable v) {
        System.out.print(v.getNom());
    }

    public void afficher(ExpressionBinaire e) {
        afficher(e.getOperandeGauche());
        System.out.print(" ");
        afficher(e.getOperateur());
        System.out.print(" ");
        afficher(e.getOperandeDroite());
    }

    public void afficher(Addition a) {
        System.out.print('+');
    }

    public void afficher(Multiplication m) {
        System.out.print('*');
    }

    public void afficher(ExpressionUnaire e) {
        afficher(e.getOperateur());
        System.out.print(" ");
```

```
32          afficher(e.getOperande());
33      }
34
35      public void afficher(Negation n) {
36          System.out.print('-');
37      }
38
39      // Les méthodes suivantes sont définies pour que le code ci-dessus fonctionne
40
41      public void afficher(Expression e) {
42          if (e instanceof Constante) {
43              afficher((Constante) e);
44          } else if (e instanceof AccesVariable) {
45              afficher((AccesVariable) e);
46          } else if (e instanceof ExpressionBinaire) {
47              afficher((ExpressionBinaire) e);
48          } else if (e instanceof ExpressionUnaire) {
49              afficher((ExpressionUnaire) e);
50          } else {
51              throw new RuntimeException("Unexpected expression: " + e.getClass());
52          }
53      }
54
55      public void afficher(OperateurBinaire op) {
56          if (op instanceof Addition) {
57              afficher((Addition) op);
58          } else if (op instanceof Multiplication) {
59              afficher((Multiplication) op);
60          } else {
61              throw new RuntimeException("Unexpected operator: " + op.getClass());
62          }
63      }
64
65      public void afficher(OperateurUnaire op) {
66          if (op instanceof Negation) {
67              afficher((Negation) op);
68          } else {
69              throw new RuntimeException("Unexpected operator: " + op.getClass());
70          }
71      }
72
73  }
```

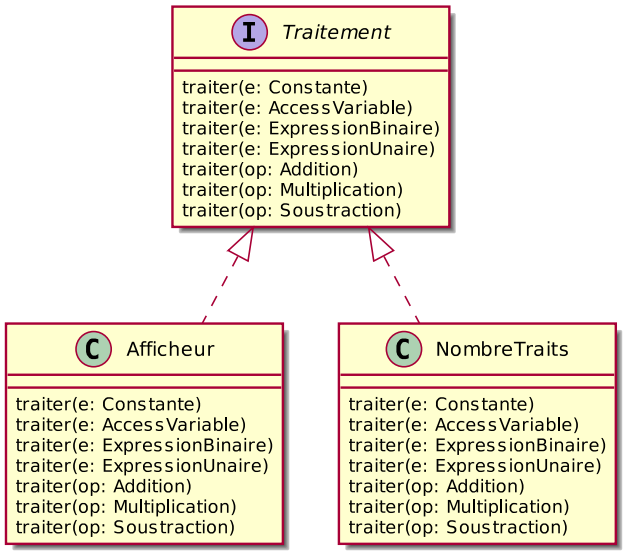
5. On sait résoudre ce problème. Le principe est de définir une méthode polymorphe sur le type le plus général, ici Expression (mais aussi OpérateurBinaire et OpérateurUnaire) et la (re)définir sur les sous-types.

   Mais on retombe alors sur la solution initiale avec les traitements éparpillés sur les différentes classes du modèle d'objet (et chaque traitement modifiant ces classes), avec la nécessité de pouvoir modifier ces classes.

6. C'est pourtant bien la bonne solution. Mais l'idée est de ne pas y avoir recours pour chaque traitement, mais seulement une fois. Il faut que chaque traitement puisse en profiter. L'idée est alors de généraliser les traitements en définissant une interface Traitement.

   On remarque en effet que tous les traitements proposent les mêmes signatures de méthodes (même si leurs noms changent) : une classe concrète du modèle d'objets. Dans la version générique, on unifie donc les noms en prenant un terme générique : traiter.

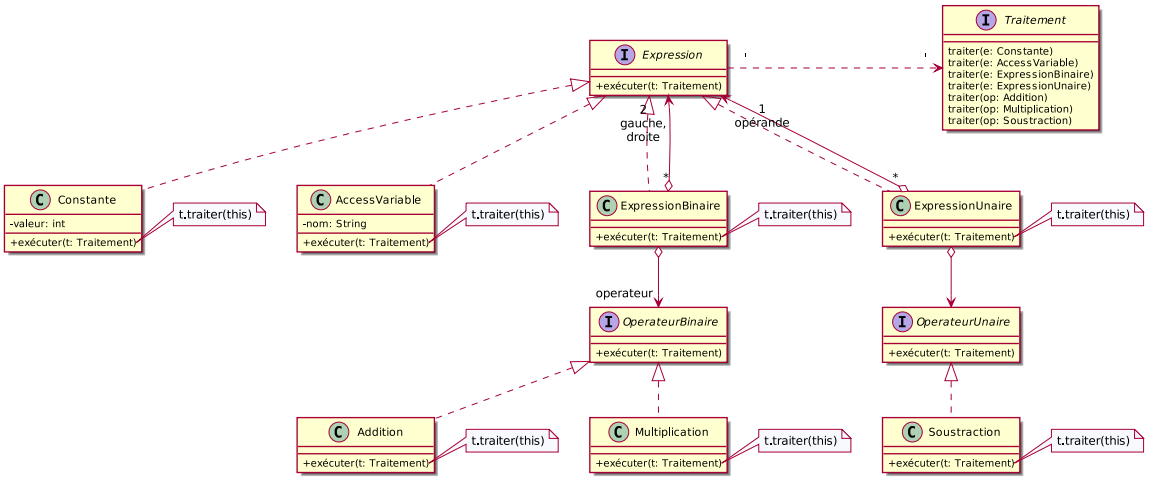
   Il suffit alors dans les classes de Traitement déjà écrites de renommer afficher, nbTraits, getValeur, etc. en traiter. C'est le nom de la classe qui le contient qui donnera le sens du traitement.



   Sur le modèle d'objet, on définit alors une méthode exécuter qui prend en paramètre un traitement. Son but est d'appeler la méthode traiter de ce traitement qui correspond à cet objet. Ceci se fait naturellement en redéfinissant cette méthode dans les sous-classes avec le code suivant :

```
1  public void exécuter(Traitement t) {
2      t.traiter(this);
3  }
```

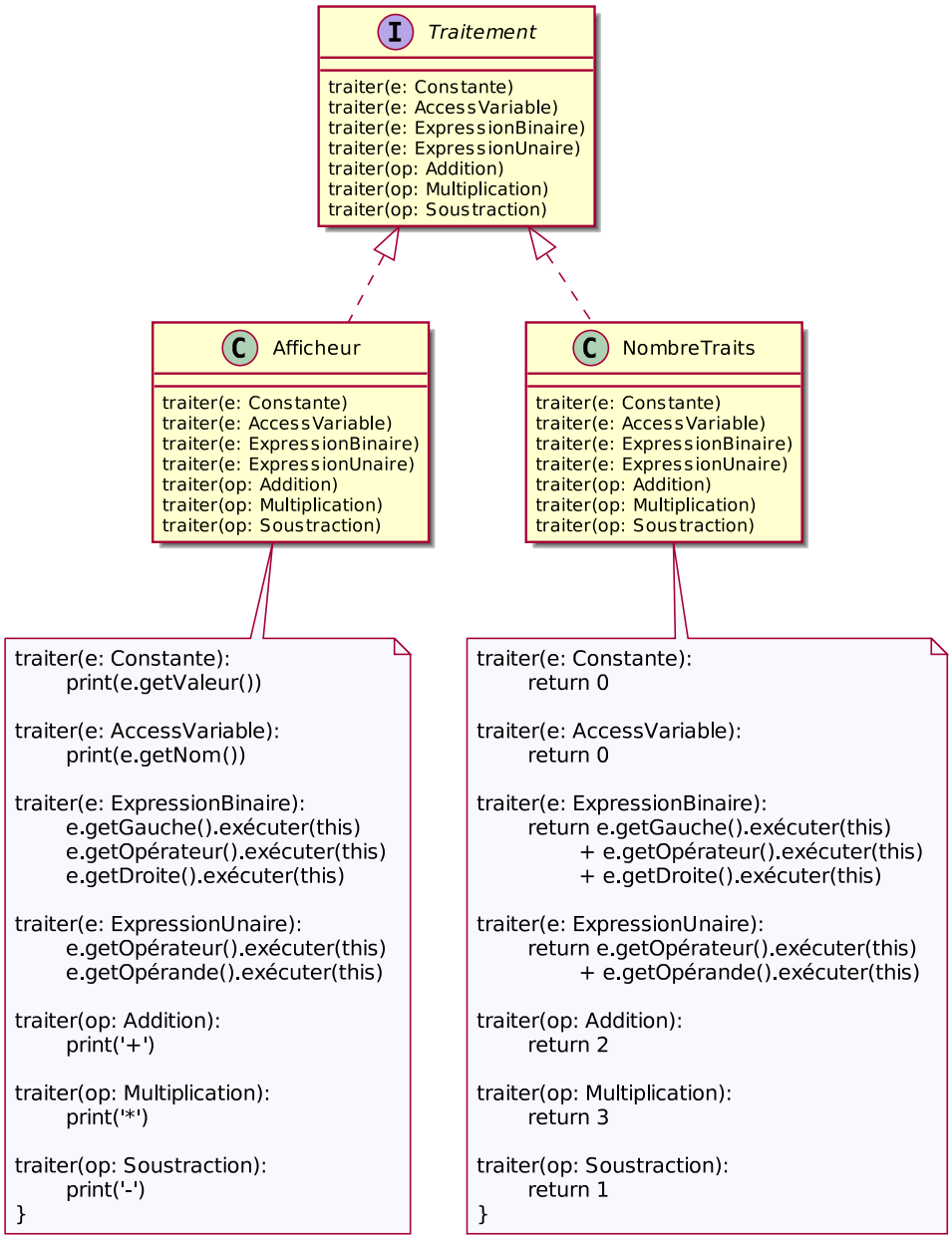
   `this` a pour type celui de la classe. Il permettra donc de choisir la méthode de Traitement qui correspond exactement à ce type (il doit y avoir une méthode traiter pour chaque classe concrète du modèle d'objet).



La solution (et qui constitue le patron de conception Visiteur) est Définir une seule méthode polymorphe pour tous les traitements.

Quand on veut exécuter un traitement sur une expression, il faut alors passer par la méthode exécuter de Expression. C'est elle qui garantit que c'est la bonne méthode traiter du traitement qui sera exécutée, celle qui correspond au type réel de l'expression.

Attention : Utiliser directement la méthode traiter conduirait à l'exécution de la mauvaise méthode traiter si le type apparent de l'expression ne coïncide pas avec son type réel.



Le patron de conception Visiteur ainsi défini permet de définir des opérations sur une structure d'objets sans qu'il soit nécessaire de modifier les classes représentant cette structure. Il est particulièrement adapté au cas où le développeur est amené à définir de nombreuses opérations différentes sur une structure d'objets dont la structure est connue ou si des développeurs différents ont besoin de définir des traitements différents sur une même structure d'objets.

Lister les inconvénients de cette nouvelle approche.

Solution : Le modèle Visiteur possède plusieurs inconvénients :

- assez abstrait (+ difficile à appréhender). En particulier, le visiteur est un paramètre de l'objet ET l'objet est un paramètre du visiteur (double aiguillage) ;
- addition de nouvelles structures d'objet difficile (modification de tous les visiteurs) ;

— remise en cause de l'encapsulation (le visiteur a accès à la structure des objets) ;
— pas de contrôle de la bonne gestion des attributs (c'est là où se trouvent une majorité de bugs dans ce type d'exercice) ;