

R3.04 – Au menu : un composite toulousain-limousin ! avec un zeste de diagramme objets

*Pour se familiariser avec le pattern Composite et,
en profiter pour modéliser/manipuler/lire quelques diagrammes objets,
ce TP s'inspirera d'un énoncé initialement écrit par Hervé Leblanc, un collègue toulousain du département
Informatique de l'IUT Paul Sabatier de Toulouse mais avec quelques adaptations locales pour le
transformer en TP limousin du département Informatique de l'IUT de Limoges 😊*

Retrouver le pattern Composite en TDD

1. Problématique générale de l'exercice

Le design pattern composite a pour **intention** de :

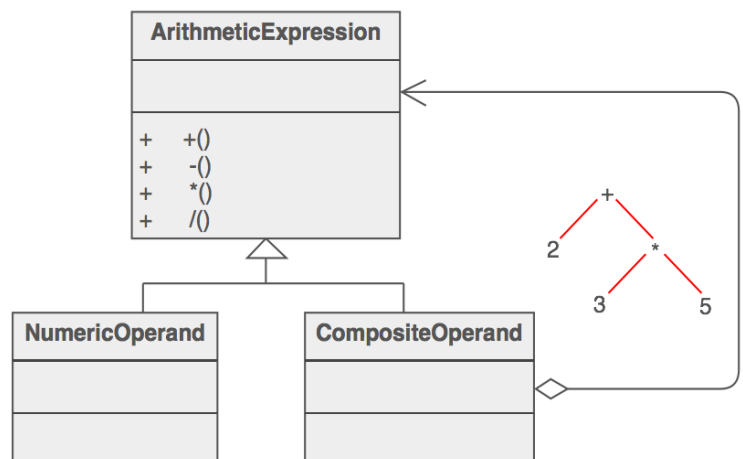
- Composer des objets à l'aide d'arbres ou arborescences (*tree structure*) pour représenter des hiérarchies de composition d'objets entre objets simples et objets composés.
- Rendre les programmes clients indépendants de la façon dont sont structurés ces objets en traitant les objets simples (ou feuilles) et les objets complexes (les nœuds internes de l'arbre) de manière uniforme.
- Composer de façon récursive des objets dans d'autres objets. Par exemple, un répertoire contient des fichiers et d'autres répertoires.
- Modéliser une relation avoir comme composants de multiplicité 1..n dans une hiérarchie.

La **problématique** est alors la suivante :

Une application doit manipuler une collection hiérarchique d'objets «primitifs» et «composites». Le traitement d'un objet primitif se fait d'une manière, le traitement d'un objet composite se fait différemment. Avoir à interroger le type de chaque objet avant de traiter une requête n'est pas souhaitable.

L'exemple que nous allons utiliser pour retrouver la structure et la cinématique de ce pattern est le suivant :

Le design pattern Composite compose des objets dans les structures d'arbres et permet aux clients de traiter des objets, et des compositions uniformément. Bien que l'exemple soit abstrait (pour le moment), les expressions arithmétiques sont des composites. Une expression arithmétique se compose d'un opérande, d'un opérateur (+ - * /), et d'une liste composée au moins d'un autre opérande. L'opérande peut être un nombre ou un autre expression arithmétique. Ainsi, $2 + 3$ et $(2 + 3) + (4 * 6)$ sont toutes deux des expressions arithmétiques valides. La figure suivante illustre ce vers quoi nous allons aller.



Le programme suivant donne un exemple d'application utilisant un Composite.
Vous pourrez exécuter ce code à la fin du TP, si vous le souhaitez 😊

```
public class ClientComposite {  
  
    // calcul de l'expression arithmétique 2 + (3 * (5 -2)) * 6 /6 + 7  
    public static void main(String[] args) {  
  
        Expression expressionComplete;  
        expressionComplete = new Addition();  
  
        expressionComplete.ajouter(new Entier(2));  
  
        Expression divisionExpressionMilieu = new Division();  
        expressionComplete.ajouter(divisionExpressionMilieu);  
  
        expressionComplete.ajouter(new Entier(7));  
  
        Expression numerateurExpressionMilieu = new Multiplication();  
        divisionExpressionMilieu.ajouter(numerateurExpressionMilieu);  
        divisionExpressionMilieu.ajouter(new Entier(6));  
  
        numerateurExpressionMilieu.ajouter(new Entier(3));  
        Expression soustraction = new Soustraction();  
        numerateurExpressionMilieu.ajouter(soustraction);  
        numerateurExpressionMilieu.ajouter(new Entier(6));  
        soustraction.ajouter(new Entier(5));  
        soustraction.ajouter(new Entier(2));  
  
        System.out.println("2 + (3 * (5 -2)) * 6 /6 + 7 = " +  
                           expressionComplete.evaluer());  
    }  
}
```

Le résultat à la console devant-être :

```
valeur feuille : 2  
valeur feuille : 3  
valeur feuille : 5  
valeur feuille : 2  
valeur -: 3  
valeur feuille : 6  
valeur * : 54  
valeur feuille : 6  
valeur / : 9  
valeur feuille : 7  
valeur + : 18  
2 + (3 * (5 -2)) * 6 /6 + 7 = 18
```


2. Retrouver le design pattern Composite en TDD

Pour commencer, vous travaillerez **en mode déconnecté**.
Laissez vos ordinateurs éteints pour le moment 😊

Question 1 (déconnecté) : Diagramme Objets

❑ **Modélisez le diagramme objets** correspondant à l'expression arithmétique de l'application cliente ci-dessus.

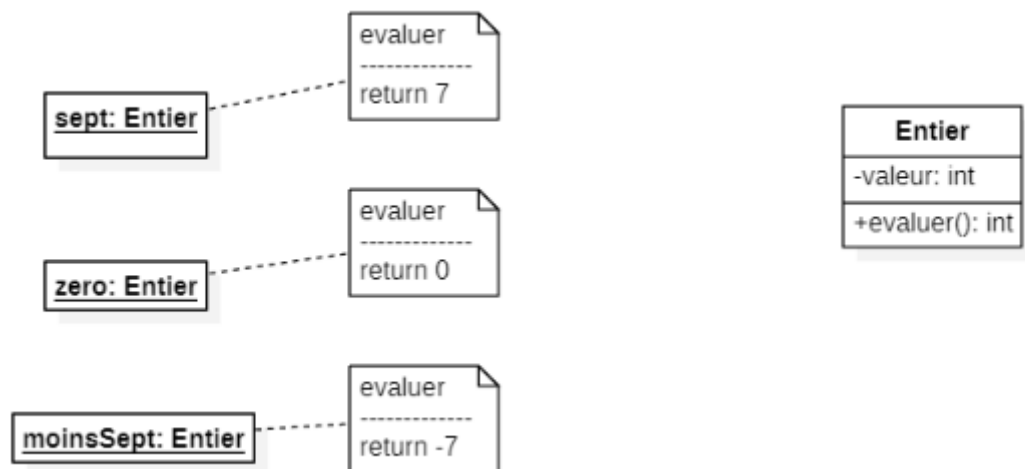
Au besoin, une annexe propose un rappel UML pour modéliser un objet et un diagramme d'objets

❑ Une fois le diagramme objets réalisé, ajoutez sous forme de commentaires UML , au bon endroit sur ce diagramme, les éléments du calcul mentionnés dans le résultat à la console, à savoir : 3, 54, 9, 18.

Remarque : Il est à noter que comme le montre le `main` de la classe `ClientComposite`, on pose comme hypothèse de bonne conception objets (et donc simplificatrice) que chacune des classes utilisées par l'application sait répondre correctement au message **evaluer()**

Question 2 (déconnecté) : Conception du code de test et du code de production de la classe Entier

Ecrire les tests unitaires ainsi que le code métier vérifiant que les expressions arithmétiques suivantes soient justes :



Remarque : Pour bien nommer vos méthodes de tests, interrogez-vous sur ces 3 tests :
Pourquoi des trois cas de tests ont-ils été choisis ?
Quels cas d'usage (plus général) visent-t-ils à vérifier et valider ?

Et maintenant, si on commençait à implémenter tout ça ...

*Vous pouvez maintenant passer en **mode connecté** 😊*

❑ Dans votre IDE préféré, créez un **projet Maven** que vous appellerez **expression** avec votre **pom.xml** habituel c.-à-d. avec au minimum :

- une version java supérieure à 1.8
- et une dépendance vers JUnit 5 😊

❑ **N'oubliez pas de versionner votre projet** (juste en local pour commencer, vous pousserez sur le distant uniquement en fin de séance) **avec un premier « commit initial »**

Question 2 bis (connecté) :

Implémentation du code de test et du code de production de la classe Entier

❑ Dans le projet **expression**, **implémentez** :

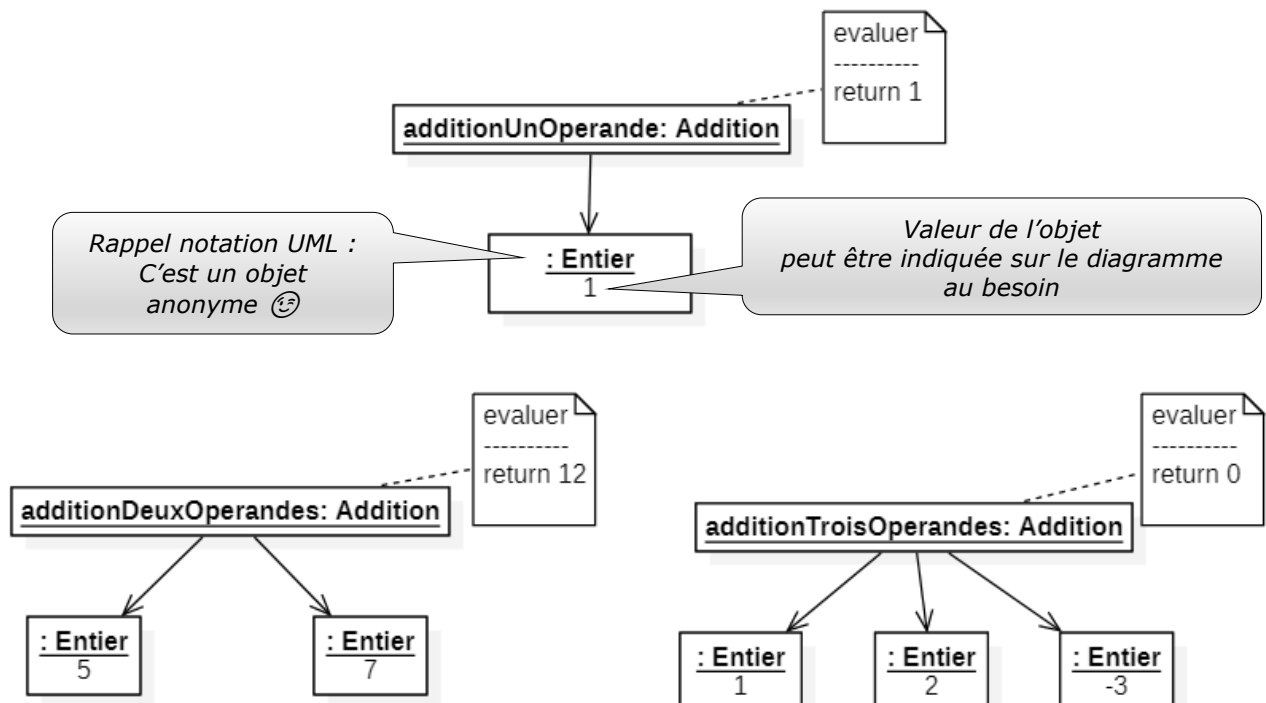
- ⇒ dans le **code de production** (**src/main/java**), la classe métier **Entier**
- ⇒ dans le **code de test** (**src/test/java**), la classe de tests **EntierTest** qui reprend les trois tests unitaires écrits précédemment afin de vérifier et valider la classe **Entier**.

❑ Après vous être assuré que le **code compile** et que **tous les tests passent AU VERT**, n'oubliez pas de **commiter** avec un message du genre : « **ajout Entier** »

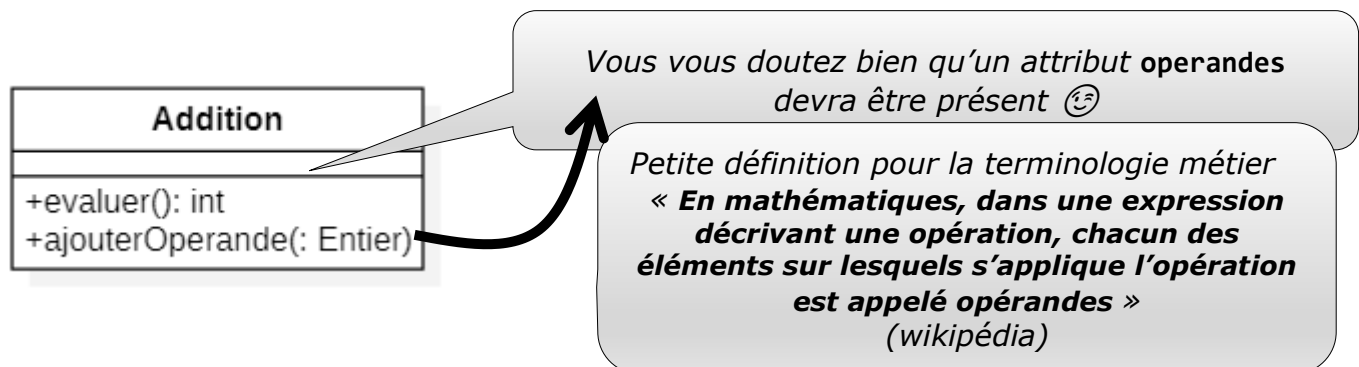
Question 3 : Implémentation et Tests de la classe Addition

❑ **Commencez par écrire le code des 3 tests unitaires** dont le comportement est modélisé dans le diagramme objets suivant.

Vous implémenterez ces trois tests dans **src/test/java** dans une classe nommée **AdditionTest**.



Pour vous aider à bien écrire ces tests, le programme de la classe **ClientComposite** vous donne des indications sur les services offerts (opérations) par la classe **Addition** 😊
 Pour l'instant ces tests ne compilent pas et c'est tout à fait normal puisque la classe métier **Addition** va être écrite juste après 😊

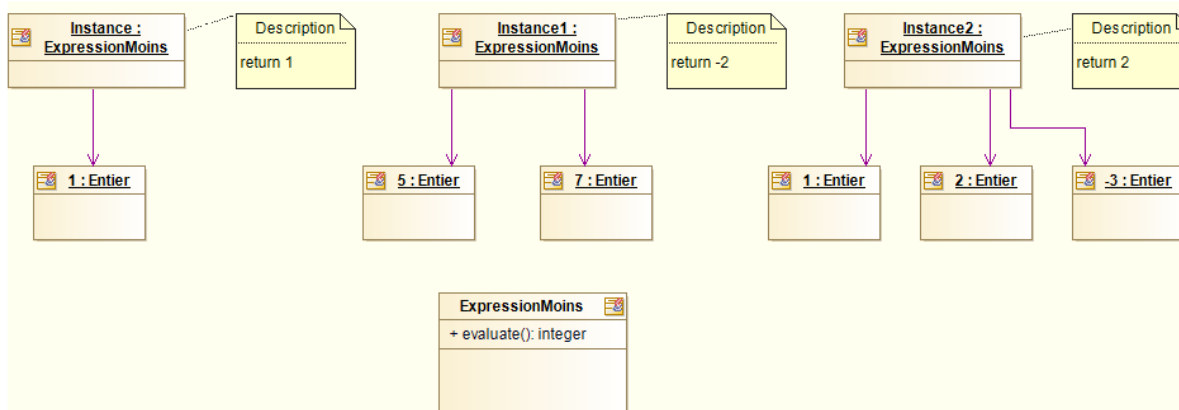


❑ **Implémentez ensuite le code métier de la classe **Addition** (dans `src/main/java`)** qui permet de **faire compiler** et de **faire passer AU VERT les trois tests de **AdditionTest****. Autrement dit, vérifier et valider que les expressions arithmétiques modélisées dans le diagramme objet précédent soit justes (conformes au comportement attendu).

❑ Une fois que le **code compile** et que **tous les tests passent AU VERT**, n'oubliez pas de **commiter** avec un message du genre : « **ajout Addition** »

Question 4 : Implémentation et Tests de la classe Soustraction

❑ **Commencez par écrire le code des 3 tests unitaires** dont le comportement est modélisé dans le diagramme objets suivant (dans `src/test/java` dans la classe **SoustractionTest**)

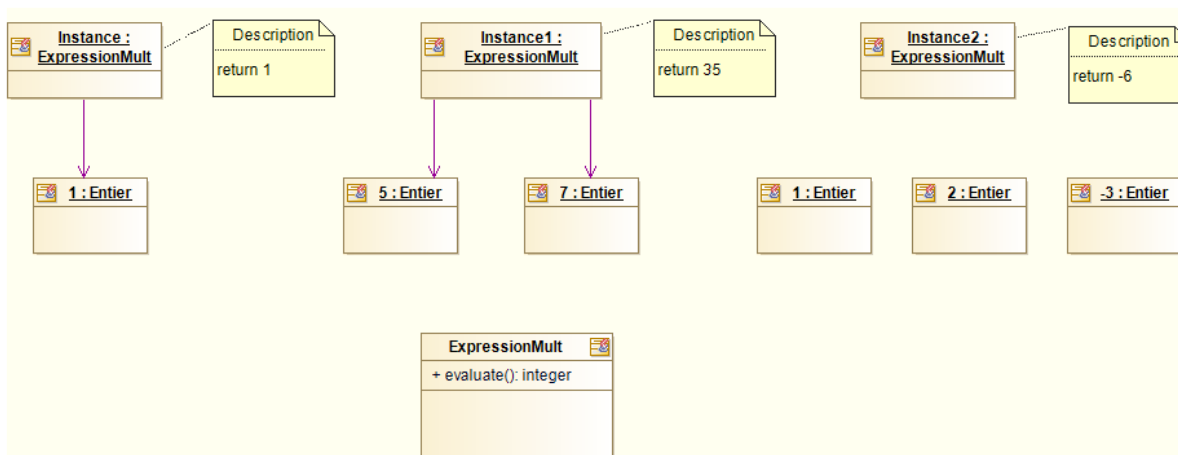


❑ **Implémentez ensuite le code métier de la classe Soustraction (dans `src/main/java`)** qui permet de **faire compiler** et de **faire passer AU VERT les trois tests précédents**. Inspirez-vous de l'implémentation de la classe Addition 😊

❑ Une fois que le **code compile** et que **tous les tests passent AU VERT**, n'oubliez pas de **commiter** avec un message du genre : « **ajout Soustraction** »

Question 5 : Implémentation et Tests de la classe Multiplication

❑ **Commencez par écrire le code des 3 tests unitaires** dont le comportement est modélisé dans le diagramme objets suivant (dans `src/test/java` dans la classe `MultiplicationTest`)

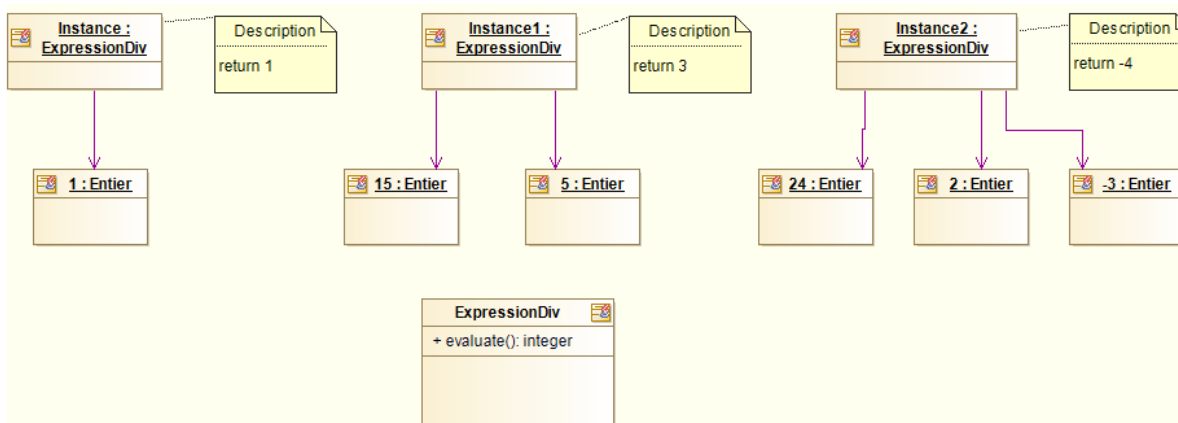


❑ **Implémentez ensuite le code métier de la classe Multiplication** qui permet de **faire compiler** et de **faire passer AU VERT les trois tests précédents**.

❑ Une fois que le **code compile** et que **tous les tests passent AU VERT**, n'oubliez pas de **commiter** avec un message du genre : « **ajout Multiplication** »

Question 6 : Implémentation et Tests de la classe Division

❑ Commencez par écrire le code des 3 tests unitaires dont le comportement est modélisé dans le diagramme objets suivant (dans `src/test/java` dans la classe `DivisionTest`)



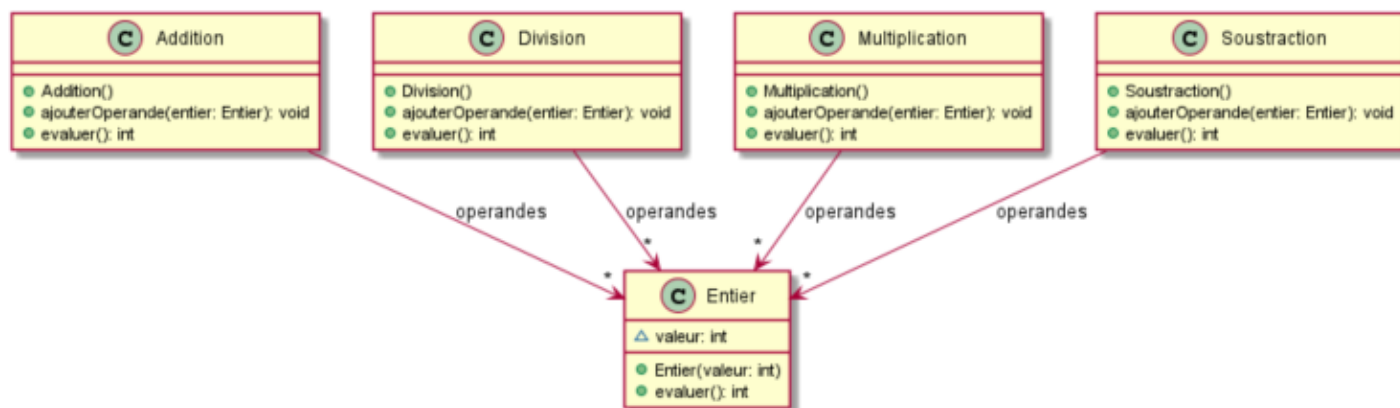
❑ Implémentez ensuite le code métier de la classe `Division` qui permet de **faire compiler** et de **faire passer AU VERT** les trois tests précédents.

❑ Ajoutez un nouveau test unitaire (et codez le comportement induit dans la classe métier) pour empêcher la division par zéro. Le comportement attendu dans ce projet en cas de division par zéro est le suivant : ne pas laisser lever l'exception `java.lang.ArithmeticException` mais l'attraper pour retourner, en cas de division par zéro, la valeur maximale associée à un entier, à savoir `Integer.MAX_VALUE`

❑ Une fois que le **code compile** et que **tous les quatre tests passent AU VERT**, n'oubliez pas de **commiter** avec un message du genre : « **ajout Division** »

Question 7 : Refactoring autour du DRY (Don't Repeat Yourself) Héritage à la rescousse pour éviter la duplication ! 😊

❑ Procédez à **une rétro-conception de votre code actuel**.
et vérifiez **que votre diagramme de classes actuel** est similaire au diagramme suivant :

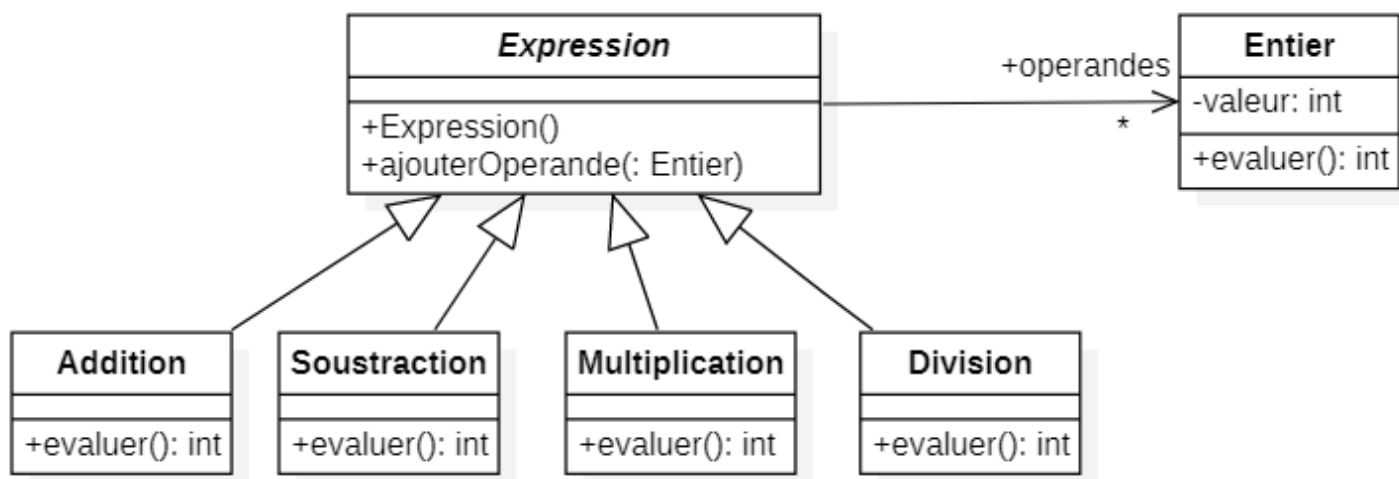


❑ En relisant votre code et/ou vous aidant du diagramme de classes précédent, vous constatez que vous avez un attribut et une méthode dupliquée dans les quatre classes **Addition**, **Division**, **Multiplication** et **Soustraction** :

- L'attribut dupliqué est :
- La méthode dupliquée est :

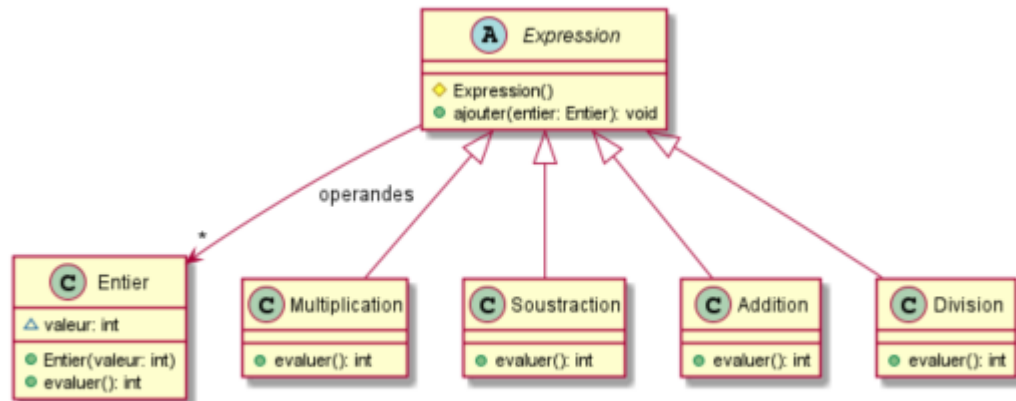
Remarque : Soyez également attentif aux DRY dans les constructeurs des classes filles 😊

❑ Afin de supprimer la duplication de code dans ces quatre classes, refactorisez votre code en **mettant en place un héritage dont la classe mère sera la classe abstraite Expression** comme indiqué dans le diagramme de classes ci-dessous.



❑ Pour vérifier que votre refactoring n'a pas changé le comportement de votre programme et donc tester la non-régression de votre code, une fois que votre **code compile** et vérifiez que **tous les tests passent AU VERT !**

❑ Renommez ensuite, à l'aide de votre IDE (option Rename du menu Refactor), la méthode **ajouterOperande** en **ajouter** de manière à ce que le refactoring se répercute automatiquement dans tout le code de test (**src/test/java**) et qu'une **rétro-conception** sur votre code de production actuel (**src/main/java**) actuel vous donne un diagramme de classes similaire au suivant.



❑ **Règle des boy scout** : Un design encore un peu *plus propre*

⇒ A la lecture de ce diagramme de classes, pour que le polymorphisme apparaisse de manière *plus propre*, ne trouvez-vous pas pertinent de faire apparaître la méthode **évaluer()** comme **méthode abstraite** de la classe abstraite **Expression** ? Procédez à cette modification.

⇒ Faites ensuite passer un *linter* sur votre code (**SonarLint** par exemple) et si ce n'est pas déjà le cas suivez ces indications quant à l'**annotation** à ajouter sur les méthodes **évaluer** des classes filles pour faire le code lié au polymorphisme 😊

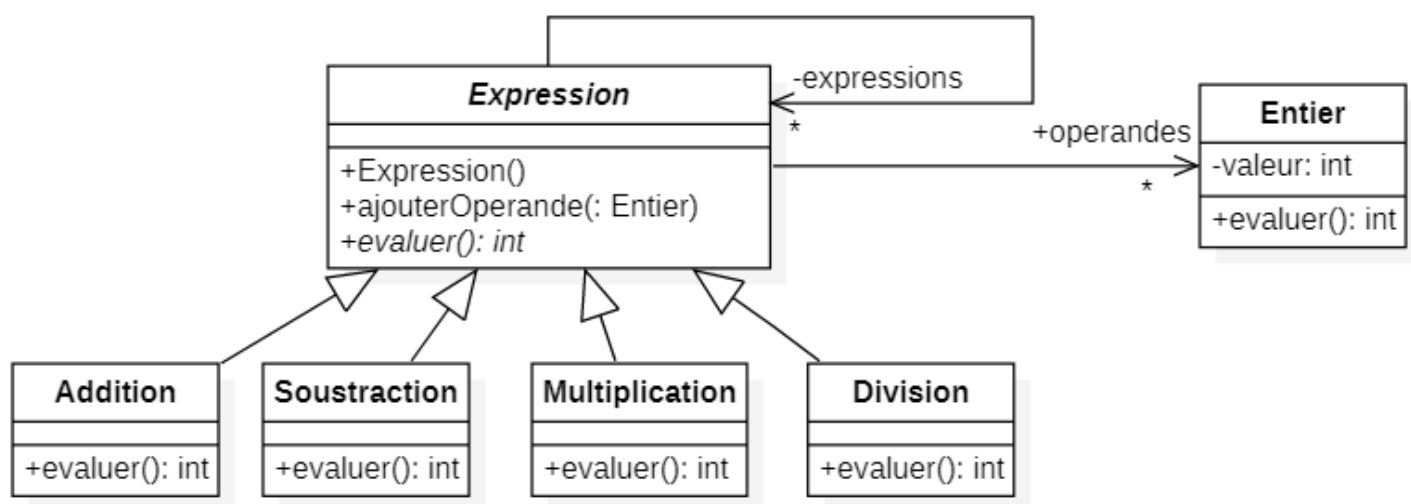
❑ Une fois que votre **code compile** et que **tous les tests passent AU VERT**, n'oubliez pas de **commiter** avec un message du genre : « **ajout héritage avec classe abstraite Expression** »

Question 8 : Mettre en place la réflexivité afin qu'une expression puisse éventuellement être composée d'autres expressions

Pour l'instant, nous n'avons que **des expressions (arithmétiques) simples à un niveau**. En effet, une expression (arithmétique) est uniquement composée d'entiers. En effet, les exemples implémentés comme tests unitaires des classes `Addition`, `Soustraction`, `Division` et `Multiplication` illustrent uniquement ce cas d'usages où seuls des entiers sont passés comme terme de l'expression.

Nous allons maintenant essayé de programmer une version plus complexe de l'expression (arithmétique) en permettant qu'une expression arithmétique **soit composée** d'entiers **et** éventuellement d'autres expressions arithmétiques comme le modéliser le diagramme de classes suivant

(Rappel UML : **est composée** est une Association forte (A-UN).



Prenez le temps de bien comprendre la spécification précédente et avant de modifier votre code pour le faire émerger vers ce nouveau design, vous allez écrire un petit test qui permettra de vérifier & valider le refactoring dans lequel vous allez vous lancer ...

❑ Le comportement à tester pour vérifier et valider cette nouvelle architecture est : une expression est composée d'expressions et d'entiers.

Pour écrire un test qui permet de couvrir ce comportement, vous allez reprendre le scénario proposé dans le « **main** » de la classe **ClientComposite** qui vous a été donné en début de sujet.

Vous trouverez le code de la classe **ClientComposite** programme sur le **gist** suivant :

<https://unil.im/clientcomposite>

(<https://gist.github.com/iblasquez/d04754e22f21f8c14030167b3799e155>)

Reprenez et transformez ce code pour écrire un test (d'acceptation) dans un fichier que vous appellerez **ExpressionArithmetiqueTestAcceptance** dans (`src/test/java`).

Remarque : Bien sûr pas de `System.out.println` dans le code de test ! 😊

... Votre code ne compile pas : c'est normal ! Il reste quelques erreurs de compilation sur les méthodes `ajouter()` et la méthode `evaluer()` qui sont (pas encore) définies et ce sera votre prochaine tâche 😊

❑ Tentez de faire passer le test d'acceptation en implémentant l'architecture du diagramme de classes proposé précédemment.

Remarque : Parfois, il faut volontairement faire apparaître de la duplication pour permettre un meilleur refactoring par la suite 😊

❑ Une fois que votre **code compile** et que **tous les tests (unitaires et acceptance) passent AU VERT**, procédez à une petite **rétro-conception** pour vérifier que le code que vous avez écrit correspond bien à l'architecture demandée dans la spécification, puis n'oubliez pas de **commiter** avec un message du genre : « **Ajout réflexivité sur Expression qui peut être composée d'expressions** »

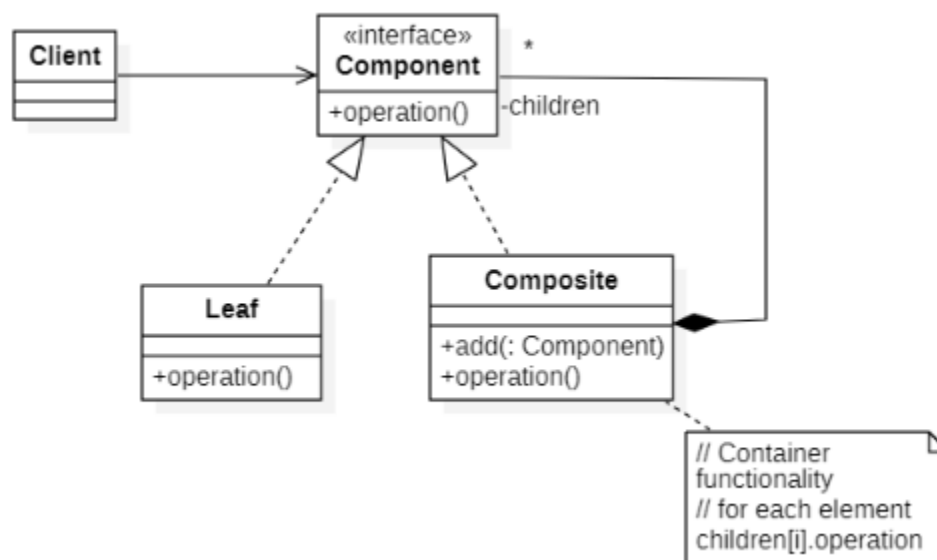
Question 9 : Vers un design plus propre et un pattern Composite 😊

En vous inspirant du premier schéma donné à la page 1, faites-en sorte de faire apparaître une classe abstraite **ExpressionArithmetique** dans votre code pour n'avoir **plus qu'une seule association**, et par conséquent réduire la complexité cyclomatique dans un seul **for** 😊

Si vous y arrivés, vous venez de redécouvrir la structure du **design pattern Composite** !

Consultez la documentation du pattern Composite donné à l'annexe 2.

A l'aide de votre IDE, **procédez à une rétro-conception** du code de votre projet **expression**. Faites apparaître le diagramme de classes rétro-conçu à partir de votre code sur le diagramme de classes génériques du pattern Composite ci-dessous :



... ce qui vous aide à appareiller les classes génériques du pattern Composite aux classes de votre contexte métier 😊 :

- ⇒ **Component** correspond dans notre contexte à
- ⇒ **Leaf** correspond dans notre contexte à la classe
- ⇒ **Component** correspond dans notre contexte à la classe

❑ Une fois que votre **code compile** et que **tous les tests passent AU VERT**, n'oubliez pas de **commiter** avec un message du genre : « **ajout ExpressionArithmetique pour mise en place DP Composite** »

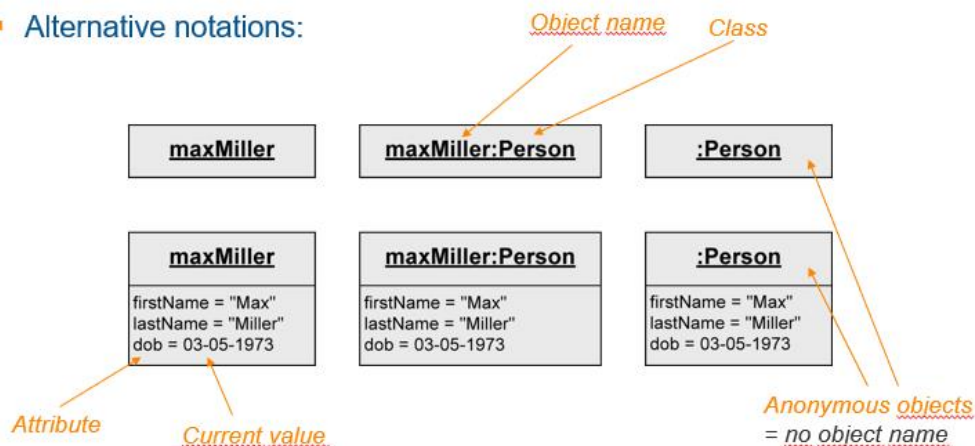
Annexe 1 : Rappel du formalisme UML pour modéliser un objet et un diagramme d'objets

Extrait de UML@Classroom de Martina Seidl, Marion Scholz, Christian Huemer & Gerti Kappel, notamment la [rubrique Material\(diapos additionnelles\)](#) du site uml.ac.at
Accès direct via : <http://www.uml.ac.at/en/lernen> puis choisir **Class Diagram** dans PowerPoint Slides

Object

o:C

- Individuals of a system
- Alternative notations:



BIG © BIG / TU Wien

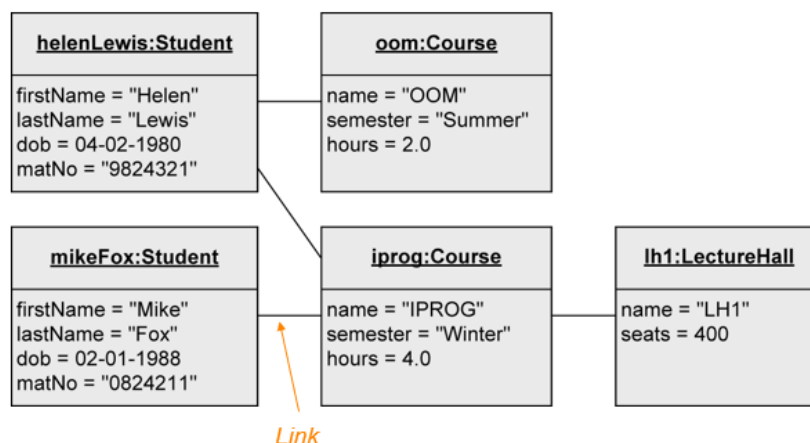
4

Object Diagram

o1

o2

- Objects of a system and their relationships (links)
- Snapshot of objects at a specific moment in time



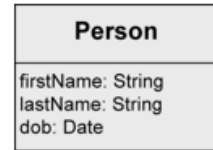
BIG © BIG / TU Wien

5

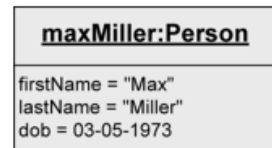
From Object to Class

- Individuals of a system often have identical characteristics and behavior
- A class is a construction plan for a set of similar objects of a system
- Objects are instances of classes
- Attributes:** structural characteristics of a class
 - Different value for each instance (= object)
- Operations:** behavior of a class
 - Identical for all objects of a class
→ not depicted in object diagram

Class



Object of that class



© BIG / TU Wien

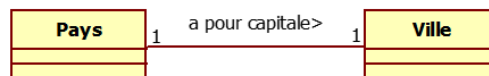


Rappel Cours R2.01 : Annexes du Diagramme de classes

<https://github.com/iblasquez/enseignement-but1-developpement>

Diagramme de classes vs diagramme d'objets (1/2)

Les **diagrammes de classes** permettent de modéliser les **classes** et les **associations entre classes**.



Les **diagrammes d'objets** (ou diagrammes d'instances) permettent de modéliser les **instances** et les **liens entre instances**.



→ un **lien** représente une relation entre deux instances :
c'est une connexion physique ou conceptuelle

→ un **lien** est une **instance d'association**

⇒ Graphiquement, s'il a un nom, il peut être souligné.

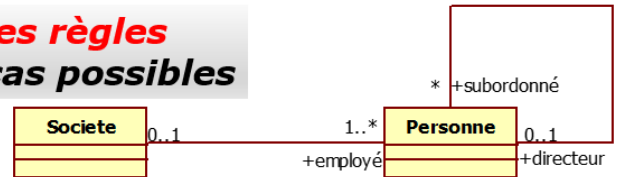
⇒ Les multiplicités ne sont pas représentées sur les diagrammes d'objets
(plusieurs objets ⇒ plusieurs liens...)



Isabelle BLASQUEZ

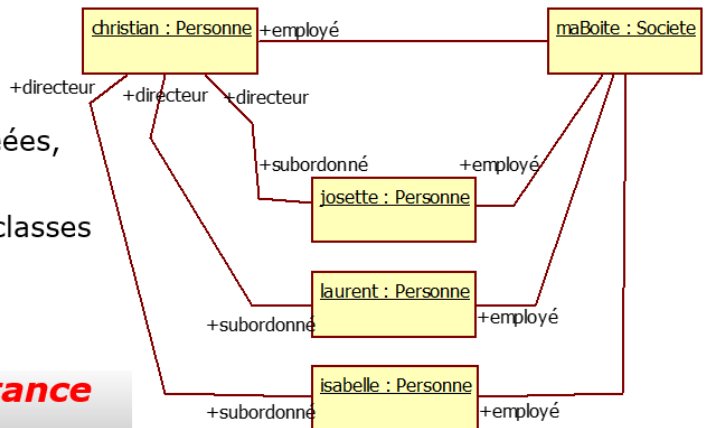
Diagramme de classes vs diagramme d'objets (2/2)

Le diagramme de classes modélise **les règles** en représentant **l'ensemble de tous les cas possibles**



Le diagramme d'objets modélise **des faits**

- Il décrit, **à un instant t**, un état du système en montrant les instances créées, leur état et les liens entre elles.
- Il doit toujours rester conforme au modèle de classes



Un diagramme d'objet est une instance d'un diagramme de classes.

Isabelle BLASQUEZ

Utilisation des diagrammes d'objets

Un **diagramme d'objets** s'utilise pour **montrer un contexte** (avant ou après une interaction entre objets par exemple). Il peut être vu comme un **instantané**, une **photo** d'un sous-ensemble d'objets d'un système à un certain moment.

Il aide à **raisonner sur la base d'exemples** et est utilisé pour :

- expliquer un diagramme de classes (exemple simple ou cas particulier)
- valider un diagramme de classes (le "tester")

Un **diagramme d'objets** représente une **vue statique des instances** qui apparaissent dans les diagrammes de classes

Pour représenter une interaction, d'autres diagrammes à base d'objets seront utilisés lors de la **modélisation dynamique**

- **diagramme de séquence**
- **diagramme de communication (2.0) (collaboration en 1.0)**
- **diagramme d'états**

Isabelle BLASQUEZ

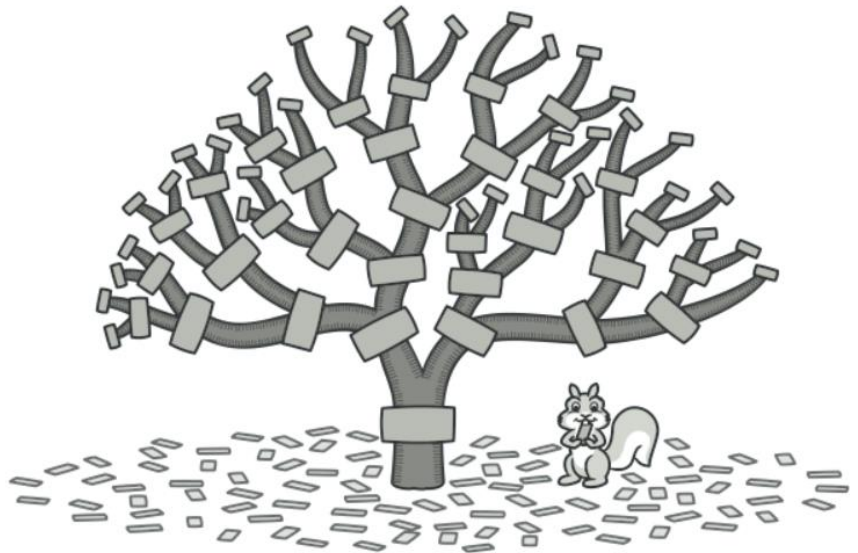
Annexe 2 : Documentation sur le pattern Composite

Commencez par vous familiariser avec le pattern composite en consultant un peu de documentation sur ce pattern extraite de vos sites favoris autour des patterns 😊

Le design pattern **composite** est un **pattern structurel**.

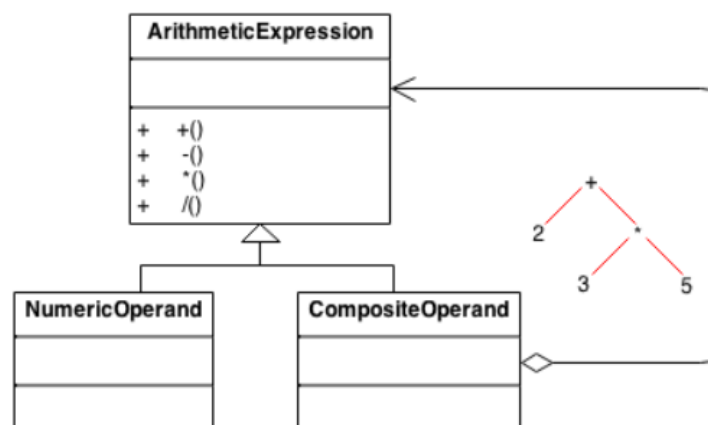
L'intention de ce pattern est de composer des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter d'une unique façon les objets et les combinaisons d'objets.

Image ci-contre extraite de <https://refactoring.guru/design-patterns/composite>

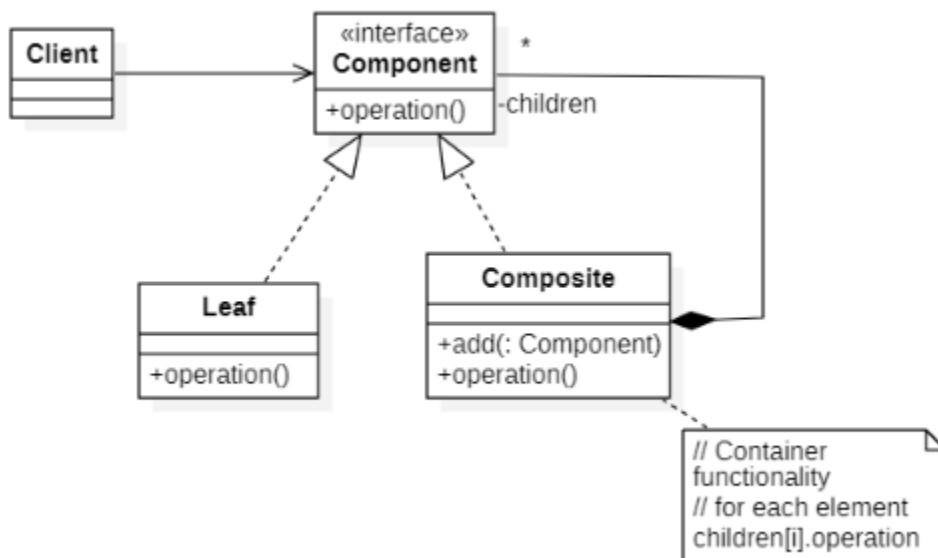


L'exemple le plus courant pour le pattern composite est celui des expressions arithmétiques : comme le montre l'extrait suivant issu de https://sourcemaking.com/design_patterns/composite

The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, $2 + 3$ and $(2 + 3) + (4 * 6)$ are both valid expressions.

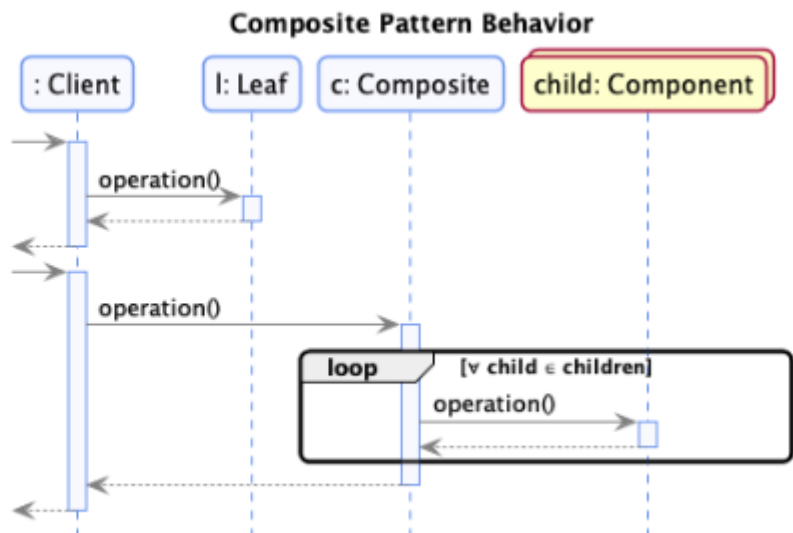


Le diagramme de classes du pattern **Composite** est le suivant



Le diagramme de séquences pour comprendre un peu mieux le comportement et surtout l'implémentation à donner à **operation**

(extrait de <https://github.com/ace-lectures/pattern-repository/tree/master/structure/composite>)



Les classes participantes à ce patron sont :

(extrait de <http://www.goprod.bouhours.net/>)

→ Component

Déclare l'interface des objets entrant dans la composition.

Implémente le comportement par défaut qui convient pour l'interface commune à toutes les classes.

Déclare une interface pour accéder à ses composants enfants et les gérer.

Eventuellement, il définit une interface pour accéder à un parent du composant dans une structure récursive, et l'implémente si besoin est.

→ Leaf

Représente des objets feuille dans la composition. Une feuille n'a pas d'enfants.

Définit le comportement d'objets primitifs dans la composition.

→ Composite

Définit le comportement des composants dotés d'enfants.

Il stocke les composants enfants.

Il implémente les opérations liées aux enfants dans l'interface Composant.

→ Client

Manipule les objets de la composition à l'aide de l'interface Composant.

Applicabilité :

Utilisez le Composite lorsque :

- Vous souhaitez représenter des hiérarchies de l'individu.
- Vous souhaitez que le client n'ait pas à se préoccuper de la différence entre "combinaisons d'objets" et "objets individuels". Les clients pourront traiter de façon uniforme tous les objets de la structure composite.

Points forts :

1. Découplage et extensibilité

1.1 Factorisation maximale de la composition

1.2 L'ajout ou la suppression d'une feuille n'implique pas de modification de code

1.3 L'ajout ou la suppression d'un composite ne n'implique pas de modification de code