Rapport de projet

LO21
Comp'UT, le calculateur des UT

Semestre A20

Dylan Cornelie - GI02
Luca Rougeron - GI02
Cécile Asselin - HU05
Julie Kocianova - GI02
Pauline Breteau - HU07

Sommaire

I. Introduction

Le projet Comp'UT présenté dans ce rapport a été réalisé entre octobre et décembre 2020 dans le cadre de l'UV LO21, Programmation Orientée Objet, à l'Université Technologique de Compiègne, par Cécile Asselin, Pauline Breteau, Dylan Cornelie, Julie Kocianova et Luca Rougeron.

Ce projet a pour but de développer une calculatrice en utilisant la notation postfixe. Cette calculatrice doit être capable de faire des calculs, de stocker et manipuler des variables et des programmes.

Dans le cadre de l'UV LO21, les technologies utilisées sont le langage C++ pour la partie développement, le framework Qt pour l'interface de la calculatrice, Visual Studio Code pour l'éditeur de texte, Qt Creator pour la partie interface et GitLab pour gérer l'intégration continue du projet. Enfin, pour la gestion de projet, nous nous sommes appuyés sur Trello pour suivre le développement du projet.

II. Description des fonctionnalités de la calculatrice

A) Fonctionnement général

L'interface de notre calculatrice comprend plusieurs vues : la vue principale comprend l'état du calculateur avec l'affichage des derniers éléments de la pile, un pavé numérique et les opérateurs les plus utilisés, un clavier avec le nom des variables et des programmes créés par l'utilisateur. La calculatrice comprend également des vues secondaires : une pour la gestion et l'édition des variables stockées dans l'application, une pour les programmes.

L'utilisateur peut ainsi entrer des opérandes, des opérations complètes (opérandes et opérateurs) ou créer des mini-programmes. L'évaluation de l'entrée de l'utilisateur peut survenir à plusieurs moments. L'opérateur EVAL permet l'évaluation d'une littérale expression et programme. Par ailleurs, une ligne d'opérande peut être évaluée lorsque l'utilisateur rentre +, -, *, / ou ENTER. Tant que la ligne n'est pas évaluée, l'utilisateur peut modifier sa saisie.

La calculatrice permet aussi de sauvegarder les programmes et variables lors de sa fermeture s'ils ont été sauvegardés avec l'opérateur STO lors de l'utilisation de la calculatrice. Ainsi l'utilisateur pourra retrouver toutes ses variables et programmes lors de la prochaine utilisation de la calculatrice.

B) Les opérateurs disponibles sur Comp'UT

a) Opérateurs numériques

Notre calculatrice permet d'effectuer des additions par l'opérateur « + », des soustractions « - », des multiplications « * », des divisions « / », des divisions entières « DIV » et des restes de divisions entières « MOD ». Elle permet aussi de changer le signe d'une littérale grâce à « NEG ». Par ailleurs, « NUM » renvoie le numérateur d'une littérale rationnelle, « DEN » renvoie le dénominateur d'une littérale rationnelle, « POW » renvoie la puissance, « SQRT » renvoie la racine carrée, « EXP » renvoie l'exponentielle, « LN » renvoie le logarithme népérien et « SIN », « COS », « TAN », « ARCSIN », « ARCCOS » et « ARCTAN » permet de renvoyer les valeurs trigonométriques associées en radian.

b) *Opérateurs logiques*

Notre calculatrice permet les opérations binaires suivante :

- =, !=, =<, =>, <, >
- AND
- OR
- NOT

Ces opérations utilisent la littérale entière 1 pour « VRAI » et 0 pour « FAUX ». À noter que toute littérale différente de la littérale entière 0, a pour valeur « VRAI ».

c) Opérateurs de manipulation de la pile

Notre calculatrice comprend « DUP » pour dupliquer la littérale du sommet de la pile, « DROP » pour dépiler la littérale du sommet de la pile, « SWAP » pour intervertir les deux derniers éléments empilés, « CLEAR » pour vider la pile. Enfin, « UNDO » et « REDO » permettent de rétablir l'état du calculateur avant la dernière opération ou avant l'application du dernier UNDO.

d) Opérateurs conditionnels et de boucle

Notre calculatrice comprend les opérateurs « IFT », « IFTE » et « WHILE ».

« IFT » prend 2 arguments, une expression et un test logique. L'expression n'est évaluée qu'à condition que le test associé soit vrai.

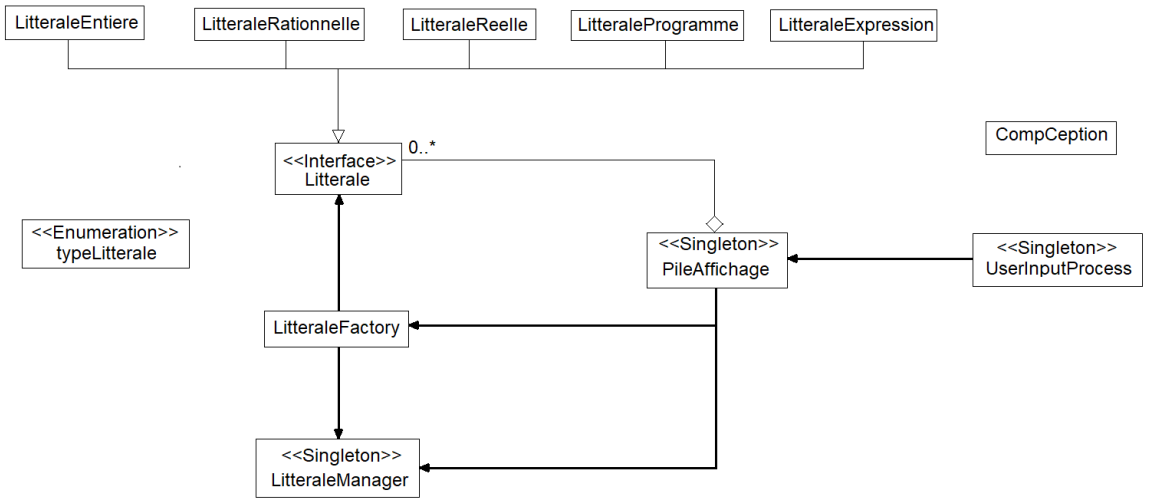
Le fonctionnement est similaire pour « IFTE », avec une subtilité supplémentaire : « IFTE » prends 3 arguments dont 2 expressions et un test logique. Si la valeur de ce test est vraie, le 2e argument est évalué et le 3e argument est abandonné sinon le 3e argument est évalué et le 2e argument est abandonné.

Enfin, pour « WHILE », prends 2 arguments dont 1 test et 1 expression ou programme. Si la valeur de ce test est vraie, alors l'expression ou le programme est évalué, et tant que ce test est vrai, l'expression ou le programme continue d'être évalué.

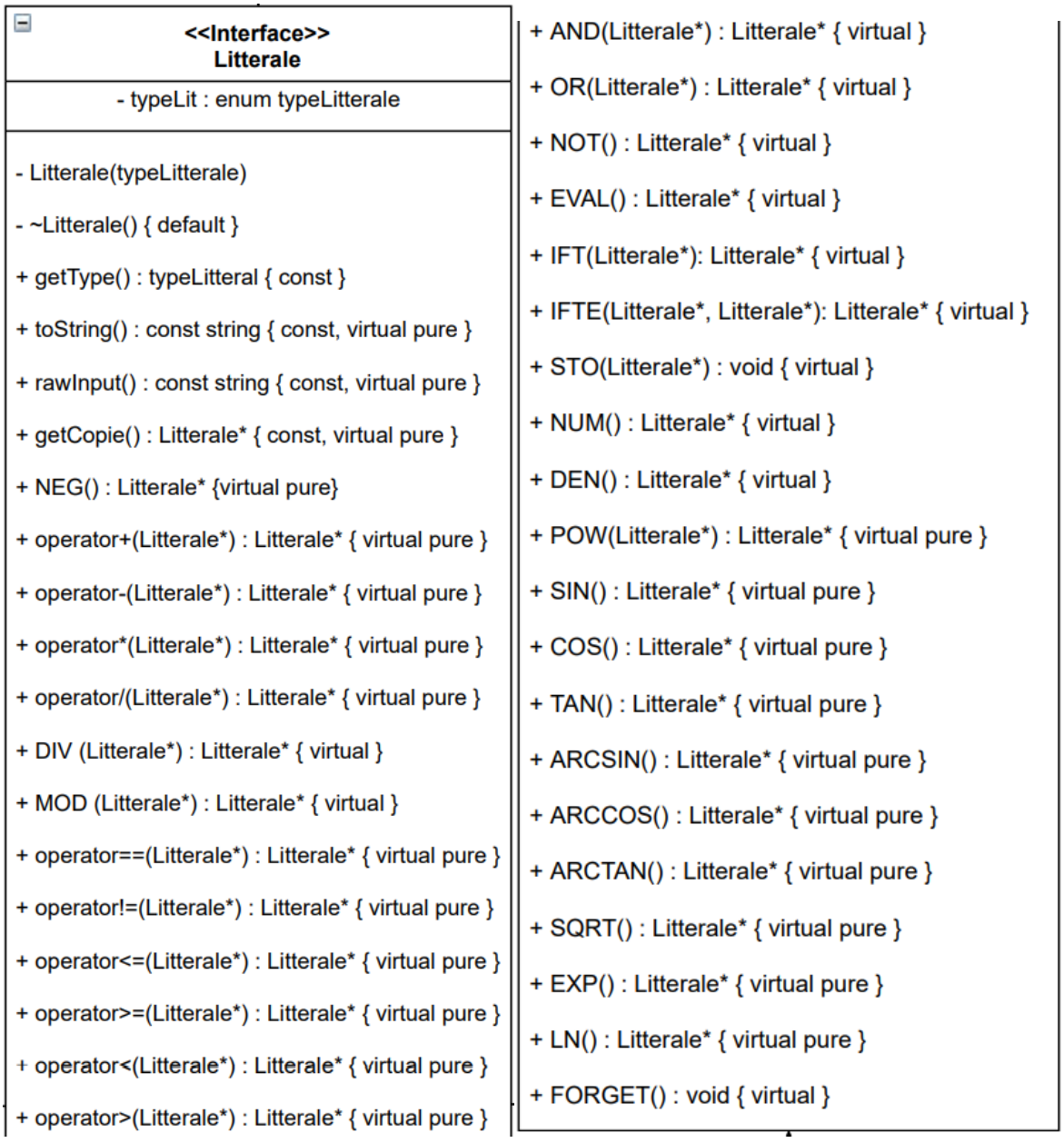
IV. Description de l'architecture

Pour une meilleure visibilité de la modélisation UML, nous allons nous baser sur cette vue globale qui montre l'UML sans les classes et les méthodes. En effet, l'UML est trop important pour être lisible.
L'UML complet est visible en annexe.

Vue d'ensemble :



A) La classe Litterale



`Litterale` est une classe abstraite, elle a le rôle d'interface. Tous les opérateurs de la calculatrice y sont déclarés, la plupart de ces méthodes sont virtuelles pures. En effet, elles devront être implémentées dans chaque classe LitteraleX car elles auront toutes un comportement propre au type de littérale.

À noter que certaines méthodes ont un comportement par défaut comme `DIV(Litterale*)`, la division entière n'est possible qu'avec des `LitteraleEntiere`, par conséquent la méthode `DIV(Litterale*)` a été redéfinie dans la classe `LitteraleEntiere`, par défaut, la méthode renvoie une erreur.

La méthode `getCopie()` renvoie une copie de la littérale en question.

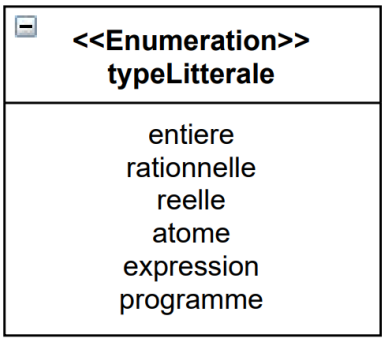
La méthode `rawInput()` renvoie la commande permettant la création de cette même littérale dans l'interpréteur de commande. À l'exception de la `litteraleProgramme` pour qui cette méthode renvoie le contenu du programme.

La méthode `toString()` renvoie une représentation de la littérale sous forme de chaîne de caractère.

La méthode `getType()` renvoie le type de la littérale, son type de retour est l'énumeration `typeLitterale`.

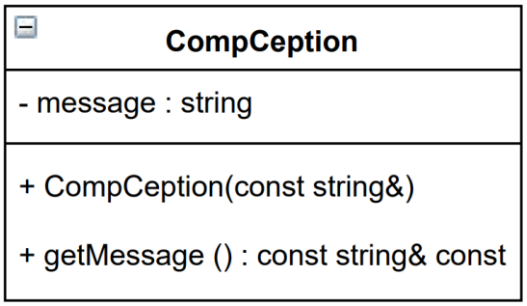
Les classes `LitteraleFactory` et `LitteraleManager` sont déclarées amies de la classe `Litterale` afin de permettre respectivement l'instanciation et la destruction des littérales par ces classes.

B) L'énumération typeLitterale



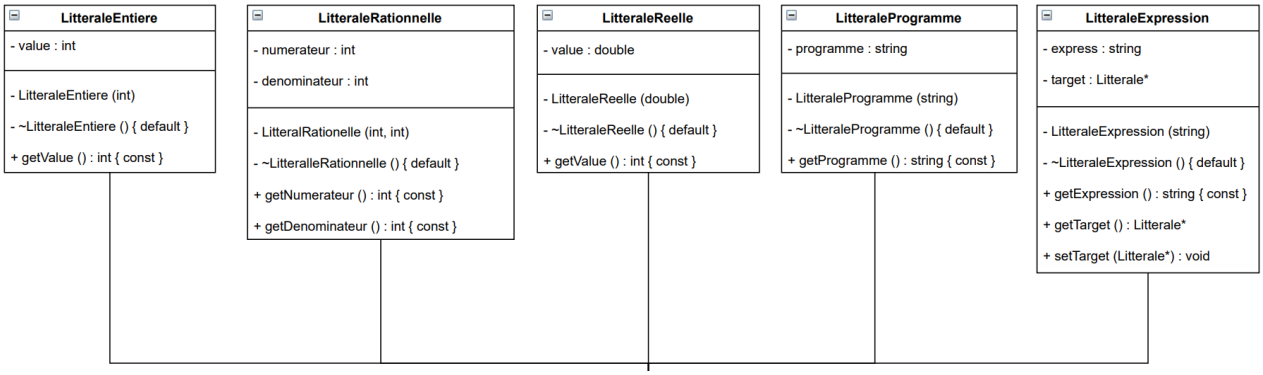
Une énumération a été créée pour répertorier tous les types de littérale auxquels on aura à faire dans l'application, on s'en sert notamment dans la classe `Litterale`, mais aussi dans les classes `LitteraleFactory`, `PileAffichage`, `UserInput`...

C) La classe CompCeption



CompCeption est la classe qui permet la gestion des exceptions. Elle contient un message qui sera affiché pour l'utilisateur en cas d'erreur avec la méthode `getMessage()`.

D) Les classes LittéraleX



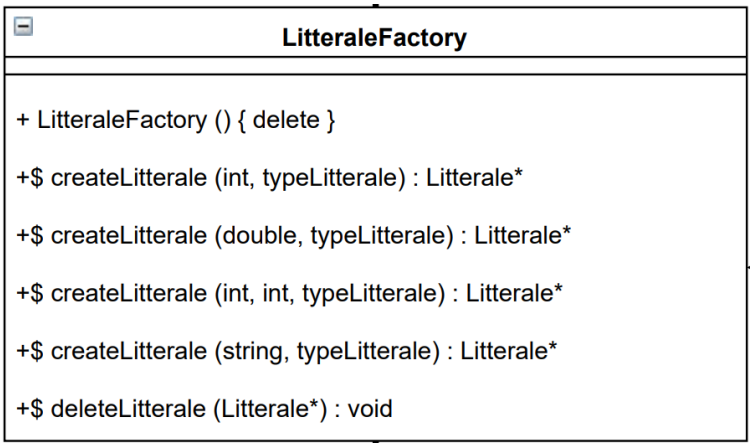
Les classes `LitteraleX` permettent de définir chaque type de littérale, elles héritent toutes de la classe abstraite `Litterale`, assurant ainsi que chaque classe implémente l'interface donnée par la classe `Litterale`.
Le constructeur de chaque `LitteraleX` est privé, nous obligeant ainsi à passer par la classe LitteraleFactory pour créer des littérales.

Chaque littérale a un accesseur permettant d'avoir accès à ce que stocke la littérale, comme `getValue()`, `getNumerateur()`, `getDenominateur()`, `getProgramme()`, `getExpression()`, `getTarget()`.

La méthode `setTarget()` de la classe `LitteraleExpression` permet de redéfinir la cible de la littérale expression, en d'autre mot, redéfinir la littérale qu'elle stocke.

Les classes `LitteraleFactory` et `LitteraleManager` sont déclarées amies des classes `LitteraleX` afin de permettre respectivement l'instanciation et la destruction des littérales par ces classes.

E) La classe LitteraleFactory

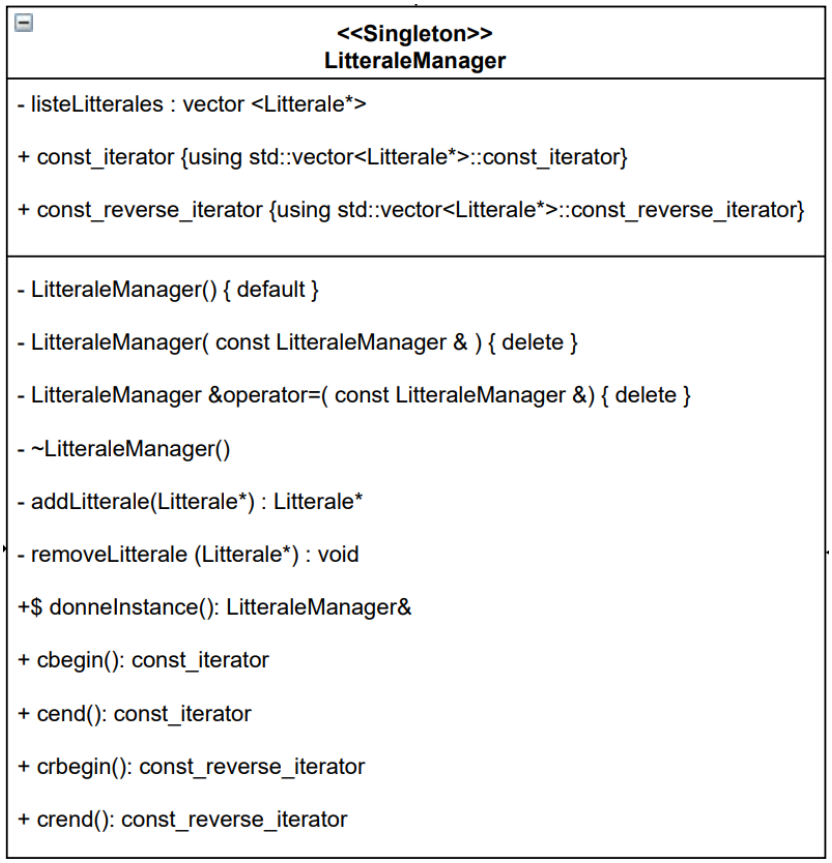


Afin d'instancier les littérales, nous avons utilisé le Design Pattern Factory Method. Ainsi nous sommes obligés de passer par la méthode `createLitterale(...)` de la classe `LitteraleFactory` afin d'instancier des objets. Ainsi, `LitteraleFactory` va permettre de créer des littérales en fonction du type de litterale et des arguments donnés. Si les arguments donnés ne sont pas cohérents, une exception est levée. L'utilisation de ce Design Pattern nous offre plus de flexibilité pour l'ajout d'un nouveau type il nous suffira de surcharger la méthode `createLitterale(...)`.

La méthode `deleteLitterale(Litterale*)`, permet quant à elle de supprimer une littérale, c'est le seul moyen qu'on offre à l'utilisateur pour supprimer une littérale, le destructeur des classes LitteraleX étant privé.

Le constructeur de la classe est désactivé car on ne souhaite pas pouvoir instancier d'objet de cette classe, elle doit uniquement comporter des méthodes static.

F) La classe LitteraleManager



La classe `LitteraleManager` est un Singleton. Cela nous permet d'avoir un seul objet qui gère le cycle de vie des littérales. On a donc une méthode `donneInstance()` qui renvoie le singleton de la classe `LitteraleManager` .

Dès qu'une littérale est créée avec la factory, son adresse est ajoutée au tableau `listeLitterales` de la classe `LitteraleManager` avec la méthode privé `addLitterale(Litterale*)`.

La méthode privée `removeLitterale(Litterale*)` est appelée par la classe `LitteraleFactory`, elle libère la mémoire de la littérale donner en paramètre et retire son adresse du tableau `listeLitterales`.

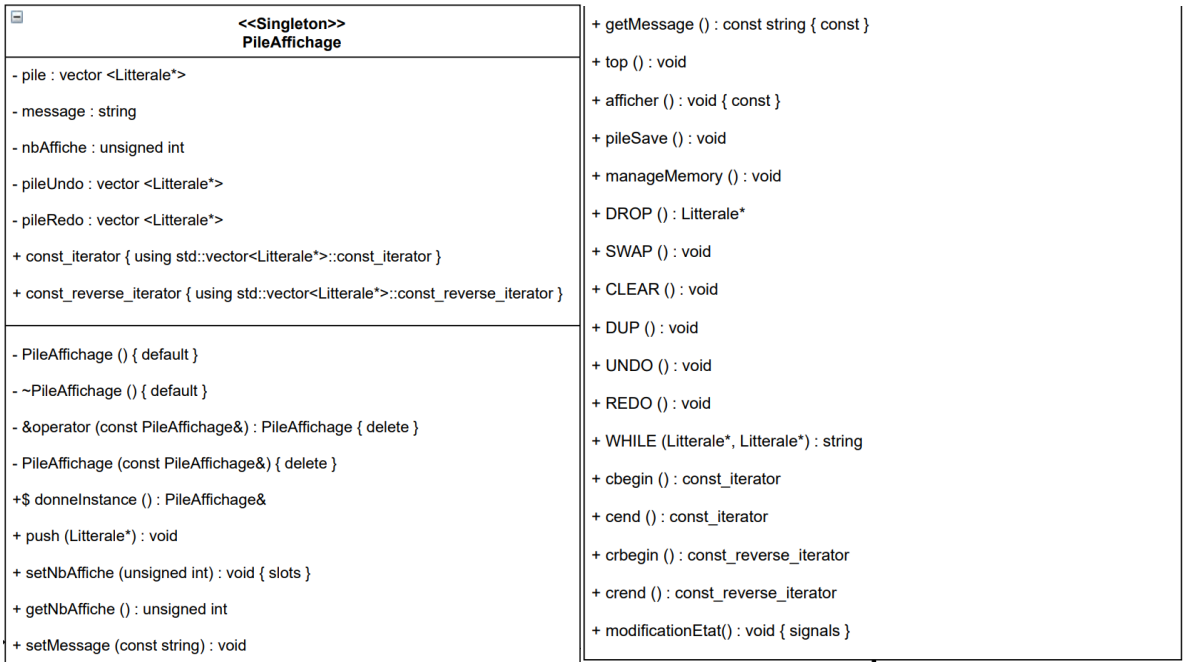
Le destructeur de `LitteraleManager`, qui intervient à la fermeture du programme, se charge de libérer la mémoire de chaque littérale créée grâce au tableau `listeLitterales`.

Afin de pouvoir accéder séquentiellement au contenu de l'attribut `listeLitterales`, un itérateur a été créé pour itérer dessus.

La classe `LitteraleFactory` est déclarée amie de la classe `LitteraleManager` afin de lui donner accès aux méthodes `addLitterale(Litterale*)` et `removeLitterale(Litterale*)`.

Il résulte de toute cette construction qu'une littérale est instanciable uniquement via la méthode `createLitterale(...)` de `LitteraleFactory` et la destruction d'une littérale peut uniquement se faire via la méthode `deleteLitterale(Litterale*)` de `LitteraleFactory` nous assurant la libération de la mémoire lors de la suppression d'une littérale. Cette construction nous assure aussi la libération de la mémoire allouée pour la création de chaque littérale à la fermeture du programme grâce au destructeur de `LitteraleManager`.

G) La classe PileAffichage



Pour la classe `PileAffichage`, nous avons utilisé le Design Pattern Singleton afin d'avoir une unique instance de `PileAffichage`. On a donc une méthode `donneInstance()` qui crée une instance s'il n'en existe pas ou qui renvoie une référence sur celle-ci si elle existe déjà. Ainsi, la classe ne peut être instanciée que par cette méthode et cela garantit son unicité.

Cette classe comprend également tous les opérateurs de pile (DROP, SWAP, CLEAR, DUP ,UNDO, REDO).

La méthode `push(Litterale*)`, permet d'empiler une littérale dans l'attribut `pile`.

La méthode `top()`, dépile le dernier élément de l'attribut `pile` (i.e : le premier élément de la pile), et le renvoie.

L'opérateur WHILE a été implémenté dans la classe `PileAffichage` car durant cette opération nous avons sans cesse besoin de récupérer le sommet de la pile avec la méthode `top()`. L'implémenter dans la classe `Litterale` impliquerait un couplage fort entre la classe `Litterale` et la classe `PileAffichage`.

Afin de pouvoir accéder au contenu de l'attribut `pile` à l'extérieur de la classe `PileAffichage` des itérateurs ont été implémentés. Cela servira notamment à l'affichage de la pile dans l'interface.

Les méthodes `setNbAffiche()` et `getNbAffiche()` permettent respectivement de modifier le nombre d'éléments à afficher sur l'interface et connaître le nombre d'éléments à afficher dans l'interface.

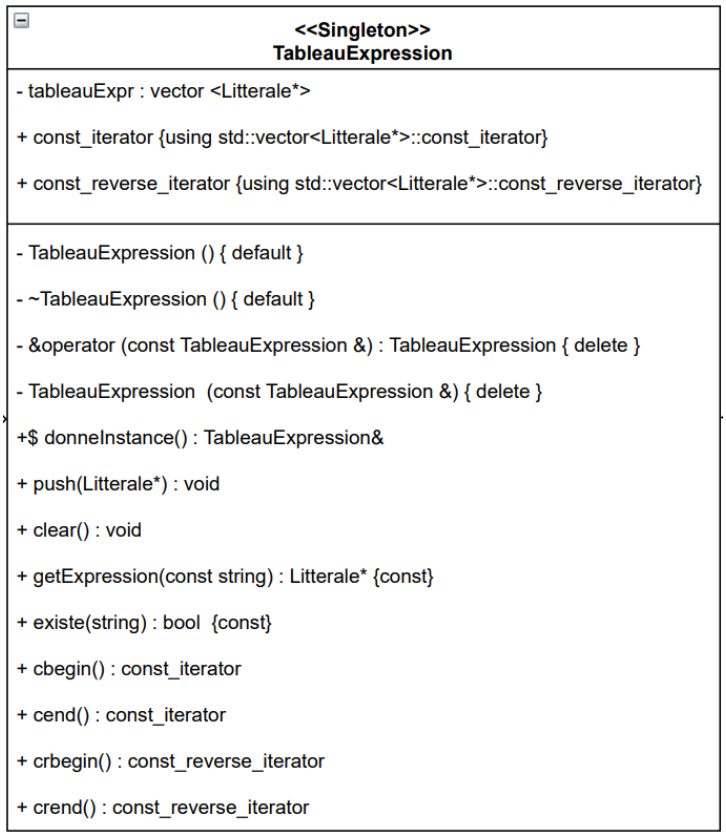
Les méthodes `setMessage()` et `getMessage()` permettent respectivement de modifier le message de la pile à afficher sur l'interface et d'obtenir le message à afficher sur l'interface.

L'attribut `pileUndo` permet de sauvegarder l'état de la pile avant chaque opération et l'attribut `pileRedo` de sauvegarder l'état de la pile avant l'opération UNDO pour l'opération REDO.

La méthode `pileSave()`, permet de sauvegarder l'état de l'attribut pile dans l'attribut `pileUndo`.

La méthode `manageMemory()`, permet de supprimer les littérales qui ne sont plus utilisées, pour cela, la méthode va pour chaque élément de la classe LitteraleManager, en utilisant son itérateur, "scanner" les attributs `pile`, `pileUndo`, `pileRedo` et le tableau d'expression avec leurs itérateurs respectifs, pour savoir si une littérale n'est plus utilisé, auquel cas elle sera supprimée avec la méthode `removeLitterale(Litterale*)` de la classe `LitteraleFactory`.

H) La classe TableauExpression



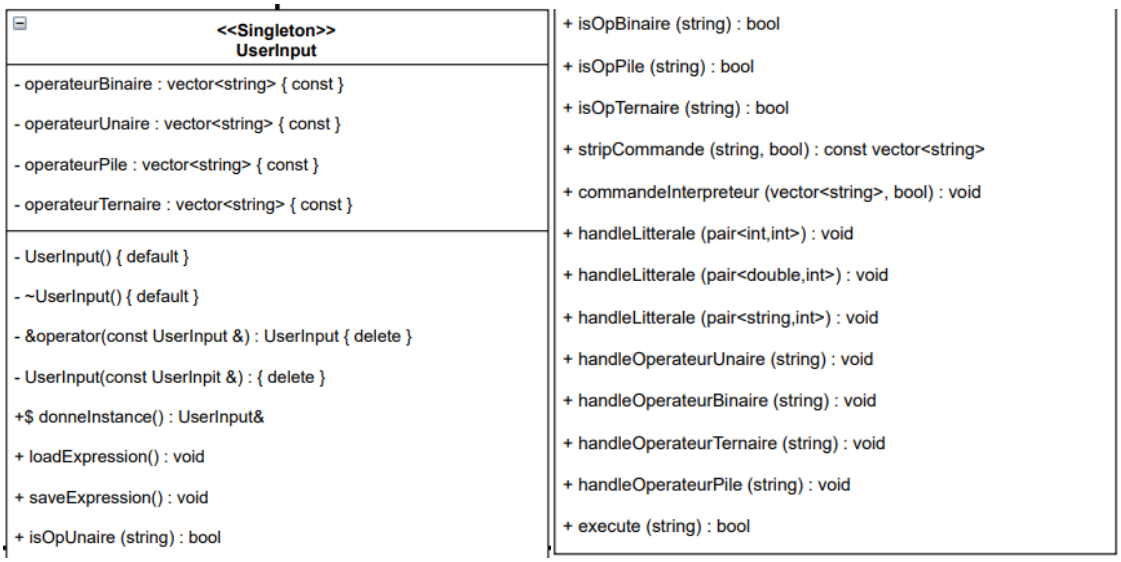
Pour la classe `TableauExpression`, nous avons aussi utilisé le Design Pattern Singleton pour les mêmes raisons que la classe `PileAffichage`. Cette classe permet de stocker toutes les littérales expression créées.

La méthode `push(Litterale*)` permet d'insérer une expression dans le tableau.

La méthode `getExpression(const string)` permet de récupérer une expression dont le nom correspond à la chaîne de caractère transmise en paramètre, la fonction renvoie `nullptr` si aucune expression ne correspond au paramètre transmis.

La méthode `existe(string)` permet de savoir si une expression dont le nom correspond à la chaîne de caractère transmise en paramètre existe ou non en renvoyant un booléen.

I) La classe UserInput



Pour la classe `UserInput`, nous avons aussi utilisé le Design Pattern Singleton pour les mêmes raisons que la classe `PileAffichage` et `TableauExpression`. Cette classe fait office d'interface entre la saisie de l'utilisateur et la partie logique de notre programme. C'est ici que sera filtrée la saisie de l'utilisateur.

La méthode `stripCommande(string,bool)` permet de "casser" la commande de l'utilisateur et de faire une vérification de la saisie de l'utilisateur, en s'assurant que les espaces ont été respectés. La méthode retourne un tableau de string avec dans chaque case une commande élémentaire. Le booléen permet de déterminer s'il faut faire une sauvegarde de la pile ou non dans pileUndo, par défaut il vaut `true`.

Par exemple, si l'utilisateur saisie "2 3 + 10 MOD [MON PROGRAMME]", la méthode nous renverra le tableau suivant : "2" | "3" | "+" | "10" | "MOD" | "[MON PROGRAMME]".

Si l'utilisateur avait rentré "2 3 + 10 MOD ", une erreur aurait été lancée en disant qu'il y a un espace inattendu et l'exécution aurait été arrêtée.

Si l'utilisateur n'avait pas refermé le crochet du programme, une erreur aurait été lancée en disant qu'un crochet n'a pas été refermé et l'exécution aurait été arrêtée.

Si l'utilisateur saisit une commande inconnue, la méthode lancera une exception.

La méthode `commandeInterpreteur(vector<string>,bool)` permet de vérifier le contenu des programmes et déduire à quel type de commande on a à faire. Le booléen permet de savoir si on utilise la méthode pour vérifier le contenu d'un programme ou non, si oui, les méthodes `handleOperateurX(string)` et `handleLitterale(pair<...,int>)` ne sont pas appelées, par défaut le booléen vaut false. Cela permet de ne pas réécrire la même fonction pour vérifier les programmes. La méthode va itérer sur le vector et pour chaque commande élémentaire de l'utilisateur, déduire quel est son type, si aucune expression régulière ne match avec la commande l'exécution est arrêtée une exception est lancé disant que la commande __ est inconnue.

Si la commande correspond à la création d'une littérale la méthode `handleLitterale(pair<...,int>)` est appelée, le second élément de la paire correspond au type de la littérale dans l'énumération `typeLitterale`, il en résulte qu'une nouvelle littérale va être créée et empiler sur la pile. Si la commande correspond à un opérateur unaire `handleOperateurUnaire()` va être appelée et l'opération va s'exécuter, si la commande correspond à un opérateur Binaire alors `handleOperateurBinaire()` va être appelée... Si l'exécution de l'opération n'est pas possible, une exception est levée et l'évaluation de la commande s'interrompt.

Dans cette méthode nous vérifions aussi que le contenu des programmes n'est pas aberrant, si un programme contient un atome qui ne correspond à aucune expression, un message d'avertissement apparaît sur la pile, s'il contient une commande inconnu une exception est lancée.

Les méthodes `isOpUnaire(string)` , `isOpBinaire(string)`, `isOpPile(string)`, `isOpTernaire(string)` permettent de déterminer si la chaîne de caractère qu'a saisi l'utilisateur correspond à une opération respectivement Unaire, Binaire, de Pile ou Ternaire de notre programme.

La méthode `loadExpression()` permet de charger les expressions sauvegardées lors d'une utilisation ultérieure dans le fichier expression.json, si le chargement du fichier .json échoue d'une quelconque manière, une erreur est lancée et l'utilisateur en est informé. Si le fichier expression.json n'existe pas rien ne se passe. Pour la lecture du fichier .json nous nous sommes appuyés sur la bibliothèque *JSON for Modern C++*.

La méthode `saveExpression()` permet quant à elle de sauvegarder dans un fichier .json toutes les expressions qui stock une littérale. Pour l'écriture de ce fichier nous nous sommes aussi appuyés sur la bibliothèque *JSON for Modern C++*.

La structure du fichier expression.json est la suivante :

```
{
      "expression1" : {
            "commande" : "ce que contient l'expression"
            "nom" : "le nom de la variable"
      },
      ...
}
```

V. Evolutions possibles

Notre architecture permet différentes évolutions de manière aisée :

- On peut ajouter des opérateurs facilement en définissant une méthode virtuelle pure dans la classe Litterale et en l'implémentant dans les classes LitteralesEntiere etc…
- On peut ajouter un type de littérale en l'ajoutant à l'énumération et en lui créant une classe avec les méthodes concernant les différents opérateurs.

A) Ajout d'un nouveau type

Pour ajouter un nouveau type comme le type complexe, il faut :

1. Ajouter le type complexe dans l'énumération typeLitterale,
2. Créer une classe LitteraleComplexe, qui héritera de Litterale, et implémenter les différentes méthodes virtuelles de la classe Litterale en fonction du type complexe.
3. Surcharger la méthode createLitterale() de LitteraleFactory
4. Définir son expression régulière dans commandeInterpreteur afin de détecter la commande permettant la création d'une LitteraleComplexe.
5. Surcharger handleLitterale(pair<...,int>) pour créer la littérale.

B) Ajout d’un nouvel opérateur

Pour ajouter nouvel opérateur il faut :

1. Définir une méthode virtuelle dans la classe abstraite Litterale
2. Implémenter cette méthode dans toutes les classes filles
3. Dans UserInput, ajouter l’opérateur dans operateurBinaire, operateurUnaire, operateurTernaire ou operateurPile en fonction de l’arité de l’opérateur
4. Dans HandleOperateurX ajouter un nouveau cas de figure avec un else if. Si cela n’a pas été fait, un message d’erreur en informe le programmeur.

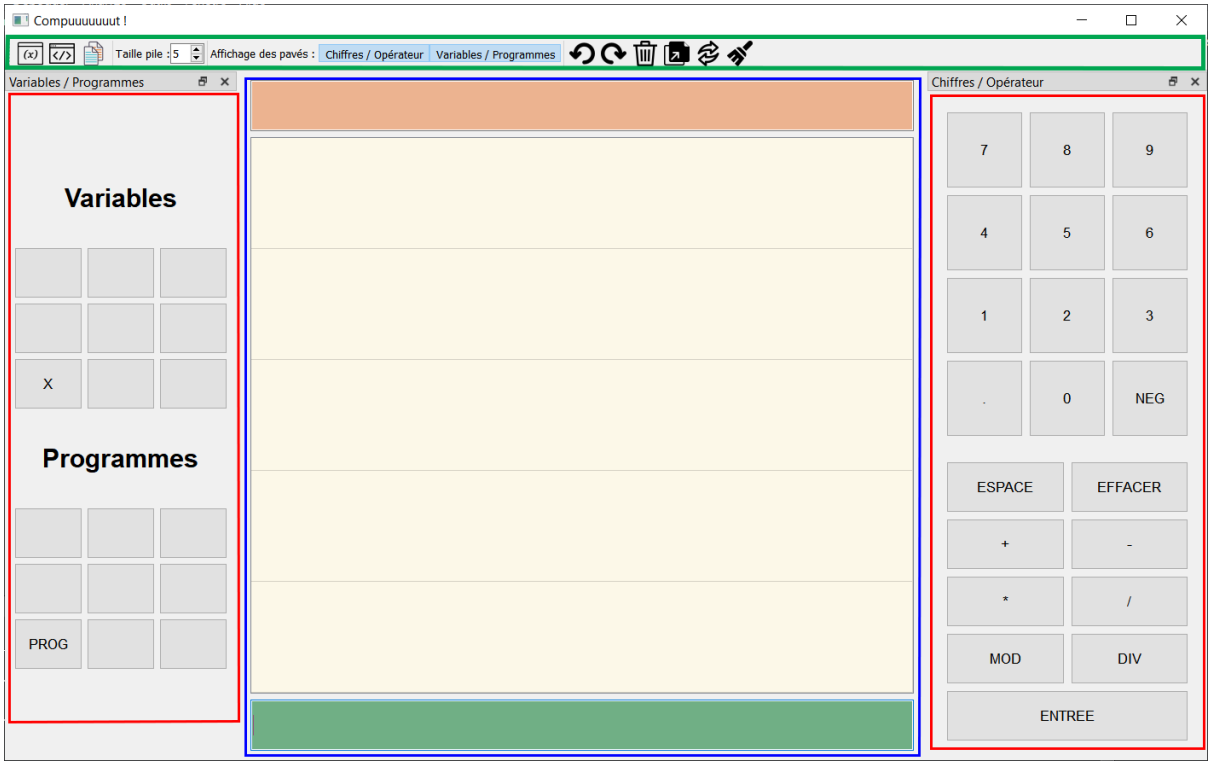
VI. Interface de Comp'UT

Pour l'interface de Comp'UT, nous avons choisi d'utiliser la classe *QMainWindow* pour afficher les éléments principaux du calculateur (pile, ligne de commande, pavés).

A) La MainWindow

Notre *QMainWindow* est composée de 3 parties :

- *Le widget central :* contient la pile (*QTableWidget*), la ligne de commande et l'affichage des exceptions (*QLineEdit*).
- La *ToolBar* : contient les actions d'ouverture des gestionnaires (variables, programmes, réglages et documentation), la modification du nombre d'éléments affichés dans la pile, l'affichage et le désaffichage des pavés et les icônes UNDO/REDO.
- Les 2 *docks :* les pavés chiffres/opérateurs et variables/programmes pouvant être affichés et désaffichés.



Bleu : widget Central, vert : ToolBar, rouge : docks

B) Vues secondaires

Pour les vues secondaires, nous avons décidé d'utiliser des fenêtres héritant de *QDialog*. On lance les vues secondaires en cliquant sur les icônes correspondant sur la *ToolBar*.

a) Gestionnaire de variables

Quand l'utilisateur ouvre une fenêtre, le programme récupère toutes les variables qui ont été créées grâce à un itérateur sur la tableau des variables. On vérifie que l'itérateur pointe bien sur une variable grâce à la condition suivante :

```
static_cast<LitteraleExpression *>(*it)->getTarget() != nullptr
&& static_cast<LitteraleExpression *>(*it)->getTarget()->getType() != programme)
```

On crée un widget pour chaque variable contenant le nom de la variable et sa valeur actuelle (*QLineEdit en setReadOnly*), un *QLineEdit* pour modifier la valeur et un bouton permettant de supprimer la variable.

Chaque widget de variable est créé grâce à la méthode *creerWidget(QString texte).*

b) Gestionnaire de programmes

De même que pour le gestionnaire de variables, la différence vient de la condition sur l'itérateur :

```
static_cast<LitteraleExpression *>(*it)->getTarget() != nullptr &&
static_cast<LitteraleExpression *>(*it)->getTarget()->getType() == programme)
```

VII. Planning du projet

La calculatrice Comp'UT a été développée lors du semestre d'automne, entre septembre et décembre 2020. L'organisation du projet est explicitée dans le tableau ci-dessous :

Tâches	**Novembre**				**Décembre**				
UML									
Code									
Interface									
Tests									
Rapport									
Vidéo									

IX. Contribution de chaque membre au projet

L’organisation générale de notre projet est détaillée dans le tableau ci-dessous, plus un membre a contribué à une partie du projet, plus sa case est foncée.

Tâches	**Dylan**	**Julie**	**Cécile**	**Pauline**	**Luca**
UML					
Code					
Interface					
Tests					
Rapport					
Vidéo					
Début version bilingue espagnole					
% TOT	24%	19%	19%	19%	19%
Nb d’heures	40	35	35	35	35

De manière générale, nous organisions des sessions de travail, en groupe de deux ou trois, grâce à l’extension Live Share de Visual Studio Code qui nous a permis de travailler de manière collaborative.