|  |
| --- |
| Universite technologique de compiegne |
| UTComputer |
| LO21 - Printemps 2016 - Projet |
|  |
| **Thibaut Roy & Romain Huber** |
| **6/13/2016** |

|  |
| --- |
| UTComputer, un calculateur scientifique permettant de faire des calculs, destocker et de manipuler des variables et des programmes, et utilisant la notation RPN |

Contents

[I – Architecture du projet 2](#_Toc453520163)

[A – Diagramme de classe du projet 2](#_Toc453520164)

[B – Description de l’architecture 3](#_Toc453520165)

[1 – La décomposition des littéraux 3](#_Toc453520166)

[2 – La littéral factory 3](#_Toc453520167)

[3 – Les opérateurs 3](#_Toc453520168)

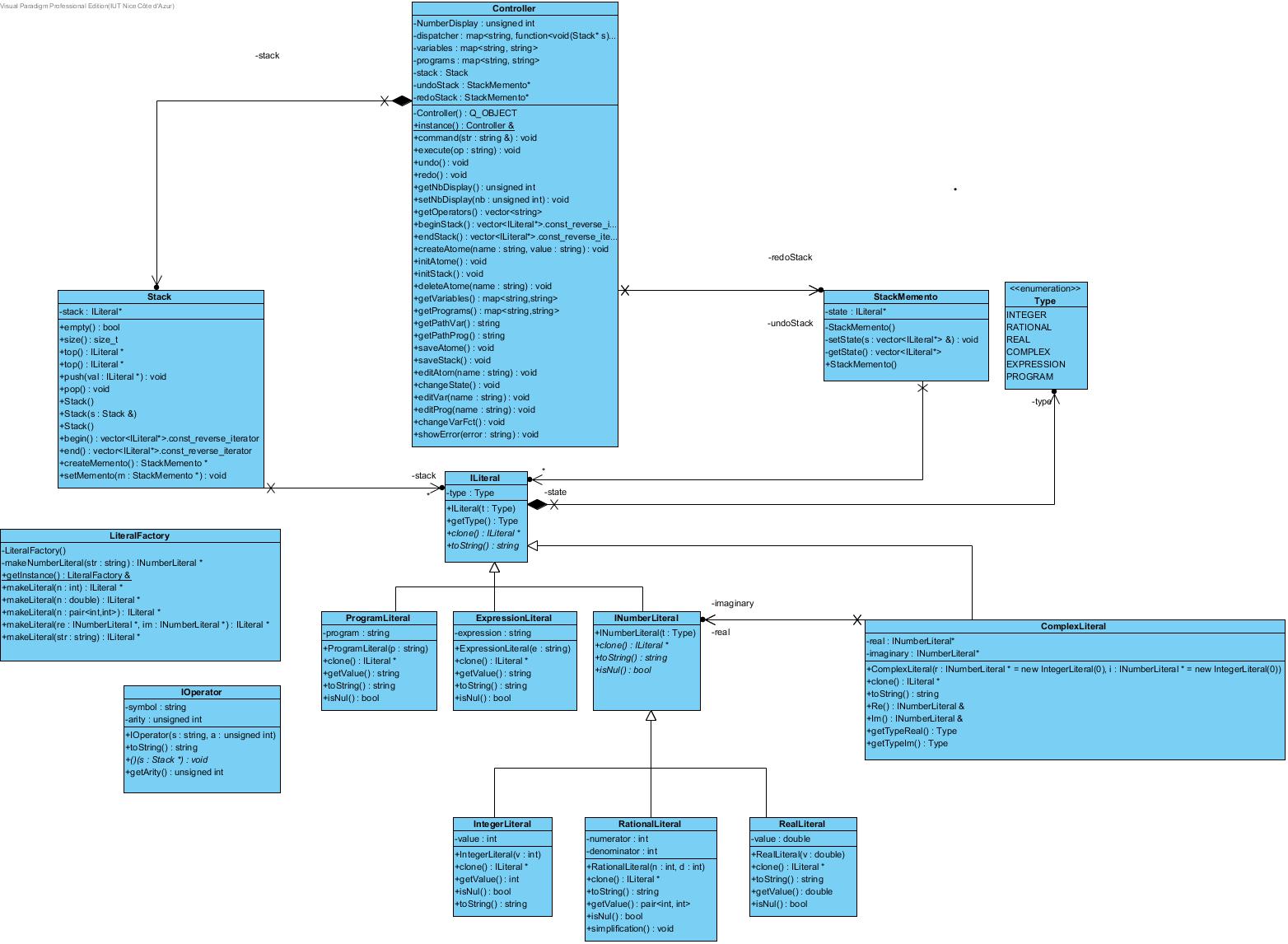
[4 – La pile 4](#_Toc453520169)

[5 – Le contrôleur 4](#_Toc453520170)

[6 – La vue 4](#_Toc453520171)

# I – Architecture du projet

## A – Diagramme de classe du projet



## B – Description de l’architecture

### 1 – La décomposition des littéraux

Comment on peut le voir sur le diagramme de classe, ILiteral est une classe abstrait qui est la base de tous les autres littéraux. Cela nous permet d’avoir une pile ne contenant que des ILiteral, qui seront eux même des ProgramLiteral, ExpressionLiteral, INumberLiteral ou ComplexLiteral. Le type contenu dans ILiteral permettra ensuite de savoir le littéral au quel on a affaire. Les types disponibles sont définis grâce à l’énumération TYPE. ILiteral définit les méthodes de base de tout littéral : getType() permet notamment de récupérer le type du littéral pour savoir comment l’utiliser.

La classe INumberLiteral est elle aussi une classe abstraite, dérivée d’ILiteral, mais qui permet de définir les littéraux-nombres. Ainsi, les entiers, réels et rationnels vont hériter de cette classe, alors que les complexes possèderont deux attributs de cette classe, un pour définir la partie réelle du complexe, l’autre sa partie imaginaire. Les méthodes isNul() permettent de voir si un NombreLiteral est nul. Cela permet notamment d’éviter les divisions par 0, ou lors de test logique, 0 représentant la valeur faux.

### 2 – La littéral factory

La factory est une classe accessible dans tous le projet grâce à sa méthode statique « getInstance ». Elle contient cinq méthodes makeLiteral qui renvoient toutes un ILiteral mais qui prennent des paramètres différents. Cela permet de créer des littéraux sans avoir à se soucier de la valeur que l’on a. La méthode prenant un string en paramètre permet par exemple de créer chacun des types de littéraux selon la chaine de caractère. Une chaine de caractère commençant par un « [ » et finissant par un « ] » créera un ProgramLiteral, alors que celle avec un « $ » créera un complexe.

### 3 – Les opérateurs

Bien qu’on ne le voie pas sur le diagramme, du fait de leur trop grand nombre, chaque opérateur hérite d’IOperator. Les opérateurs ne définissent pas plus de méthode que ce IOperator a déjà, mais chacun d’eux aura un symbole particulier (+, -, MOD, etc…) et une arité (2 pour +, - etc.. par exemple). Chacun d’eux va aussi redéfinir l’opérateur () et exécuter des actions sur la pile selon leur but. Le fait qu’ils redéfinissent tous la même méthode permet d’avoir une liste d’opérateur (dispatcher) dans le contrôleur qui est une map de <string, operator()> et donc d’appeler facilement les opérateurs. L’opérateur prend en paramètre la pile pour permettre d’effectuer des modifications à celle-ci. L’opérateur va aussi contrôler que le nombre d’éléments dans la pile est bien supérieur ou égale à l’arité de l’opérateur, sinon celui-ci déclenchera une exception.

### 4 – La pile

La pile contient un vecteur d’ILiteral qui permet de gérer tous les types de littéraux. Grâce au pattern adaptor, le vecteur est utilisé comme une pile, ainsi push mettra un nouveau littéral au bout du vecteur, mais pop renverra aussi le dernier élément du vecteur. Cela permet une gestion LIFO (Last In First Out) de la pile. La méthode begin qui permet d’itérer sur la pile est lui dans cet optique, en envoyant un reverse itérator. Les méthodes createMemento et setMemento permettent l’utilisation du pattern memento dont on parlera plus tard pour les opérateurs UNDO et REDO. Ils vont créer une copie de la pile à un moment T pour conserver ses états et donc les réutiliser par la suite.

### 5 – Le contrôleur

Le contrôleur est le cœur de la calculette. Il fait la liaison entre la pile, les opérateurs et la vue. Il possède une instance de la pile pour la gérer lorsque la vue envoie des demandes utilisateurs, une liste pour les variables, une pour les opérateurs et une pour les programmes. La méthode commande permet traduire la ligne de commande envoyée par l’utilisateur. A partir de cette ligne, c’est cette méthode qui décidera d’exécuter un opérateur, de créer une variable, un programme, un INumberLiteral etc… Il utilise différents signaux pour communiquer avec la vue, notamment showError pour afficher à l’utilisateur les différentes exceptions générées par sa ligne de commande (comme une division par 0), ou changeState pour mettre à jour la vue de la pile quand la stack est modifiée. C’est aussi lui qui récupèrera dans des fichiers les variables, fonctions et l’état de la pile lors du démarrage de la pile, puis qui les enregistrera à sa fermeture.

### 6 – La vue

Bien qu’elle ne soit pas dans le diagramme de classe, la vue est une part importante du projet. Elle est fortement liée au contrôleur qui s’occupera de faire le lien entre elle et la pile. Elle a de nombreux slots qui permettent de gérer les signaux du contrôleur, en mettant à jour la vue de la pile, l’affichage des variables et des programmes ou les messages d’erreurs.

Elle récupère aussi dans le contrôleur tous les opérateurs, les variables et les programmes pour les afficher dans un menu contextuel afin que l’utilisateur puisse les utiliser facilement sans les connaître par cœur.