Université de Technologie de Compiègne

LO21 - POO

Rapport de projet

Calculatrice à Notation Polonaise Inverse

Printemps 2016

Alexis Dutot - Valentin Montupet

12 juin 2016

Table des matières

1 Introduction

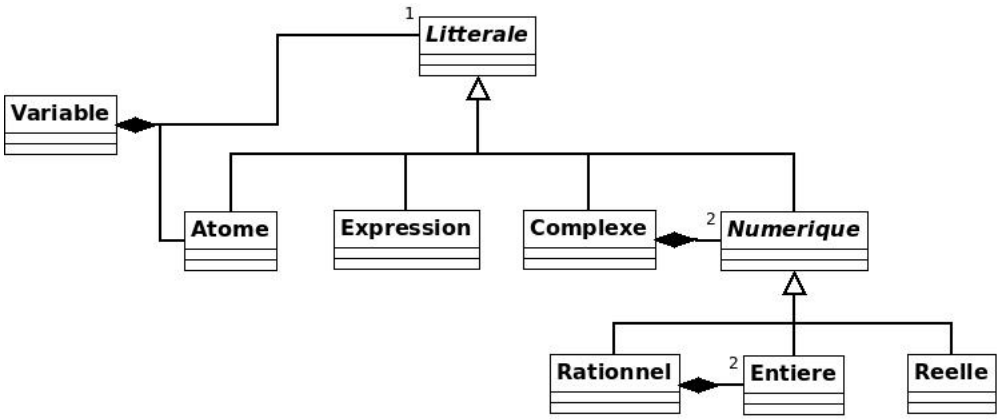
Ce projet a été réalisé dans le cadre de l'UV "LO21 - Programmation Orientée Objet" enseignée à l'Université de Technologie de Compiègne (UTC) par Alexis DUTOT et Valentin MONTUPET, étudiants en Génie Informatique.
Ce projet s'insère dans le cadre de l'enseignement sous forme d'application concrète des concepts de la programmation orientée objet et du langage C++ vus en cours tout au long du semestre. L'objectif fût de développer l'application *UTComputer*, une calculatrice scientifique permettant de faire des calculs, de stocker et manipuler des variables et des programmes, tout en utilisant la Notation Polonaise Inverse (NPI). La notation polonaise inverse est en fait une méthode de notation mathématique permettant d'écrire une formule arithmétique de façon non ambiguë sans utiliser de parenthèses. Pour plus d'informations, se référer à la littérature sur le Web

Pour mener à bien ce projet, plusieurs étapes ont été nécessaires :

- L'analyse complète du sujet, établir les différentes fonctionnalités de l'application
- la conception de l'architecture de l'application, afin de pouvoir produire un code **modulable, réutilisable et extensible**
- l'écriture du code source, suivant l'architecture établie
- la création de l'interface graphique
- la création de la documentation html à l'aide de Doxygen©

Ce rapport sera organisé en deux parties : la première rendra compte de la description de l'architecture établie, la seconde sera une argumentation détaillée montrant que l'architecture permet facilement des évolutions.

2 Description de l'architecture

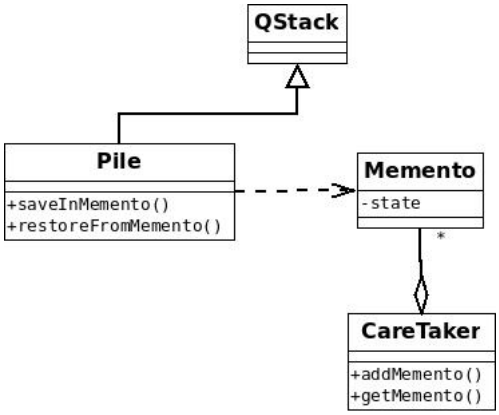


La calculatrice est amenée à manier, à l'aide d'une **Pile**, des **Litterales**. Ces dernières peuvent être de type **Numérique** (**Littérale Entières, Réelles, ou Rationnelles**), **Complexes**, **Programmes** ou **Atomes**.

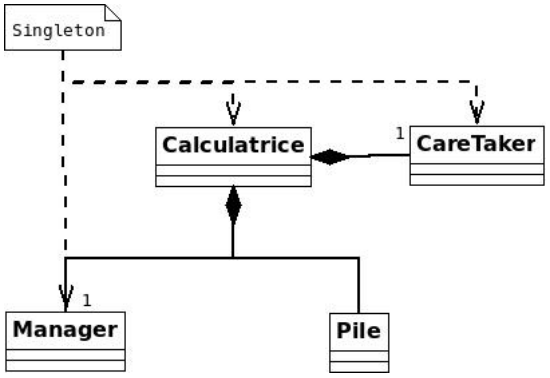
Nous avons donc implémenté la classe mère abstraite **Littérale** dont hérite directement ou indirectement tout type de littérale. Dans cette classe mère sont regroupées les méthodes (déclarées virtuelles pures) dont toutes les littérales héritent. Par exemple, quelque soit le type de littérale, il sera nécessaire d'avoir une méthode *toString*() afin de pouvoir convertir la littérale en une chaine de caractère afin de l'afficher dans l'application par la suite.
Une **Littérale Complexe** est une littérale découpée en deux parties, appelées partie réelle et partie imaginaire. Nous avons choisi que **chaque partie pouvait être une littérale Numérique**, à savoir un entier, un réel ou un rationnel, d'où la composition entre **Numérique** et **Complexe**.

Une **Littérale Atome** n'est autre qu'une chaine de caractère pouvant représenter le nom d'une variable, d'un programme ou d'un opérateur. Une **Variable** est donc composé d'un **Atome** (son identificateur) et d'une valeur, qui peut être une **Littérale** quelconque.

Les variables sont stockées dans un **Manager**. Il s'agit de vecteur de variables. A noter que le **Design Pattern Singleton** est utilisé pour le Manager car l'application n'a besoin que d'une unique instance de cette classe.



La **Pile** est la classe qui stocke les littérales. Nous l'avons fait hériter de la classe **QStack**. Ainsi, nous pouvons utiliser ses méthodes $push(), pop(), top()$ et les itérateurs de la classe associée à Qt. Les littérales sont donc empilées une à une dans la QStack.
Pour réaliser l'opération **undo/redo**, nous avons implémenté le **Design Pattern Memento**. La classe Pile encapsule une classe Memento. Cette classe permet de réaliser une sauvegarde de la pile à un instant donné t. A chaque modification de la Pile, l'application réalise une sauvegarde de la pile dans un Memento. Le CareTaker mémorise une liste de Memento qu'il peut restituer quand on lui demande.
Ainsi quand l'utilisateur demande une opération UNDO, le CareTaker restitue le Memento décrémente son index actuel et retourne le Memento à cette position.
Quand celui-ci demande une opération de REDO, le CareTaker incrémente son index actuel et retourne le Memento à cette position.
La classe Pile n'a plus qu'à restaurer sa pile depuis un Memento donné. A noter que le **Design Pattern Singleton** a été utilisé pour CareTaker afin d'avoir une unique instance gérant les Memento.

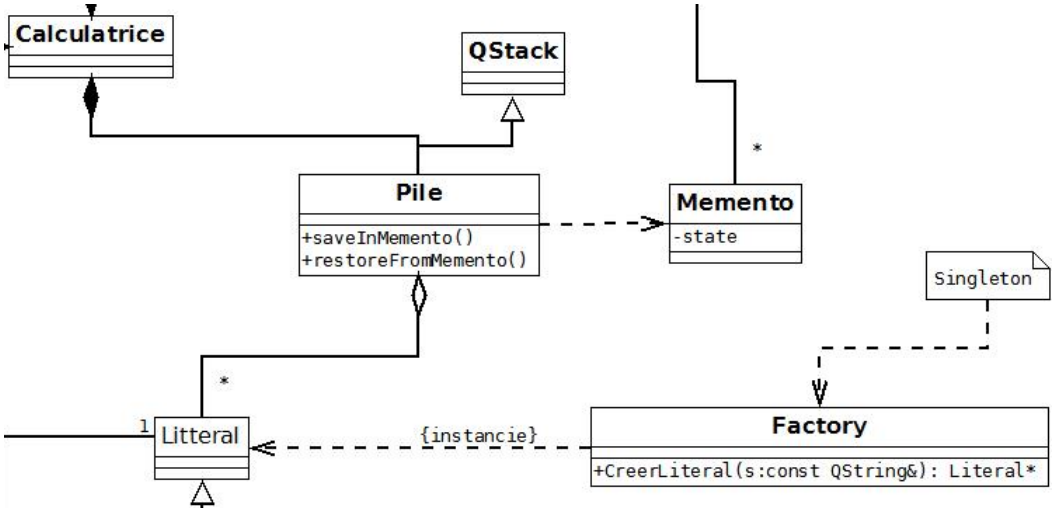


Pour coordiner le tout, nous avons créé une classe **Calculatrice**. Le **Design Pattern Singleton** est utilisé afin de n'avoir qu'une seule instance de Calculatrice.
La calculatrice est composée d'une **Pile**, d'un **Manager** et d'un **CareTaker** pour les opérations de Undo/Redo.
En plus de gérer les différents composants de l'application, la classe Calculatrice permet de garder en mémoire tout attribut en rapport avec ses composants (comme les derniers opérateurs ou le dernier opérateur utilisé).
Evidemment, c'est ici que les opérations seront déterminés : chaque opération aura une méthode, qui déterminera quelle fonction appelée en fonction des types de littérales.
Par exemple, si l'on souhaite faire une addition, la méthode $PLUS()$ de la Calculatrice est appelée. Cette fonction teste les types de chaque littérale est appelle la méthode $l1.operator+(l2)$ qui correspond.
Lorsqu'un utilisateur va rentrer une commande dans la calculatrice, celle-ci va devoir être analysée. Il va donc falloir déterminer s'il s'agit d'un opérateur, d'une littérale entière, réelle ou encore une expression. Il faut aussi savoir s'il s'agit d'un identificateur d'un programme ou d'une variable et si tel est le cas, exécuter le programme ou empiler la valeur de la variable.
Nous avons séparé cette phase de "parsing" en deux étapes :

— **Etape1** : la calculatrice se charge de regarder si la chaine de caractères saisie correspond à un opérateur ou à la "forme" d'une littérale (notamment grâce à des expressions régulières que nous avons définies spécifiquement). Si ce n'est pas le cas, elle renverra une exception.

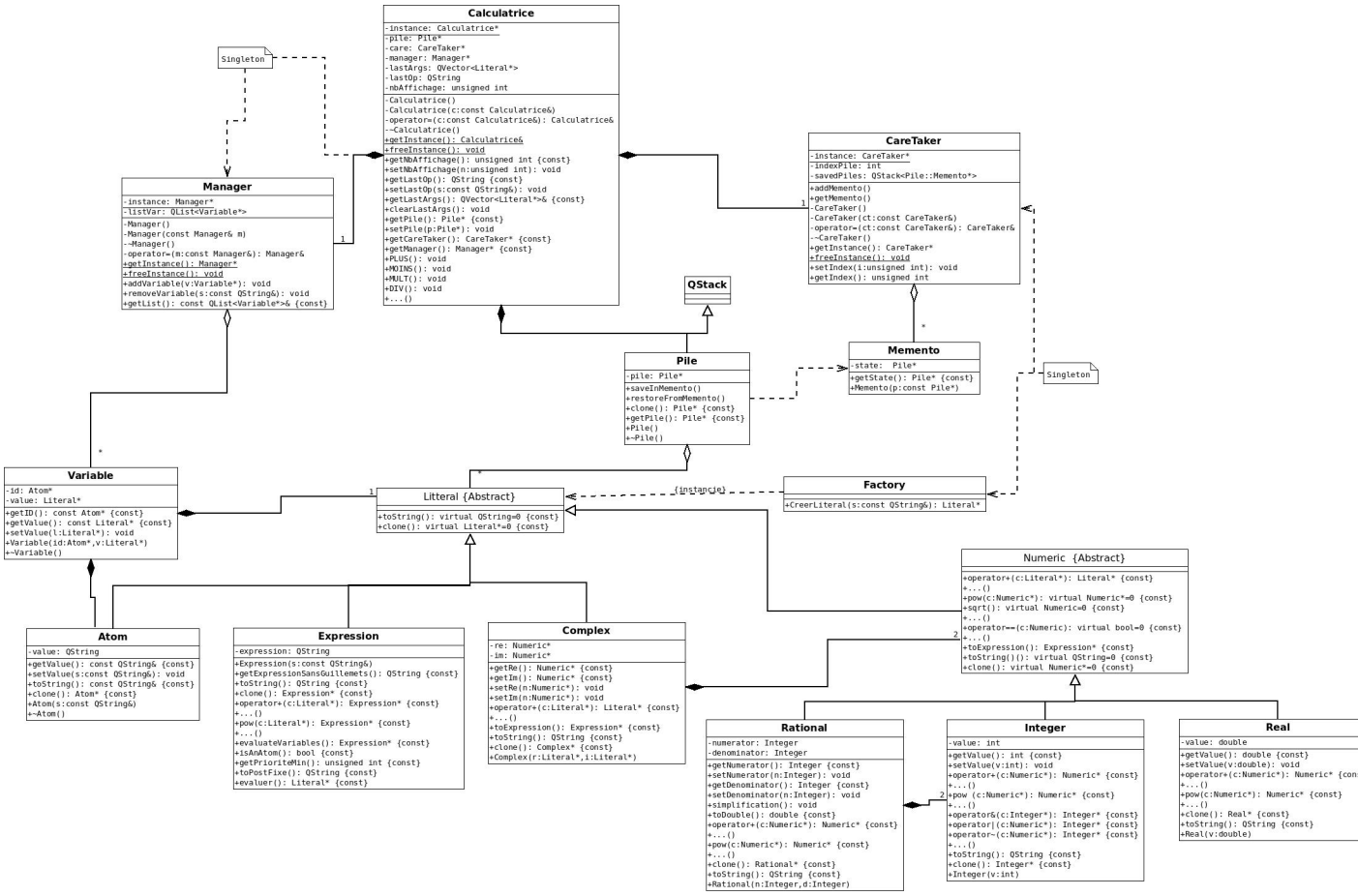
— **Etape 2** : si la chaine entrée correspond à un opérateur alors la fonction de calcul correspondant à cet opérateur est exécutée. Si la calculatrice a determiné que la chaine était une litterale alors elle va déléguer la fabrication de la littérale à la classe **Factory**.

Ce même processus est utilisé dans l'évaluation d'une expression. Nous avons donc implémenté le **design pattern Factory Method** afin de déléguer une partie du travail de la calculatrice à la fabrique. A celle-ci, nous avons implementé le **design pattern Singleton** pour qu'elle soit unique à toute notre application. Il s'agit d'une fabrication paramétrée de littérales. Un objet (tel que la calculatrice ou une expression) va appeler la fabrique par le biais de la méthode *CreerLiteral(const QString s)*. L'objet appelant va transmettre un objet de type QString qui represente la littérale à créer. A l'aide d'expressions regulières, la fabrique va choisir, créer et retourner la bonne littérale à l'objet appelant. Par exemple, s'il existe une variable X1 avec pour valeur 5 et que "X1" est transmis à la fabrique, une littérale entière ayant pour valeur 5 sera retournée.



3 Une architecture évolutive

Voici donc le diagramme UML global de notre application :



L'application a été construite de manière à être flexible, modulable et évolutive. UTComputer étant notre première application conçue, nous avons réalisé à quel point produire une architecture 100% modulable, réutilisable et extensible est extrêmement difficile. De ce fait, nous sommes conscient que la conception de notre application peut être améliorée et n'est pas optimale. Néanmoins, nous avons mis une attention particulière à rendre l'application souple, et chaque choix de conception est justifié par celà. Voici les differents choix :

— **Litterales** : nous avons essayé au maximum de décomposer les différents types de littérales, afin d'atteindre un niveau de granularité suffisant pour que chaque type de littérale soit indépendant, ou bien qu'un type de littérale soit composé d'autres. Ainsi, nous avons créé deux classes abstraites : la première étant la classe **Literal**, classe mère de toutes les littérales, et la seconde étant **Numeric**, classe mère des littérales qui ne font intervenir que des nombres. Ainsi, il est possible d'introduire dans l'application un nouveau type de littérale en le faisant hériter au moins de **Literal** et en redéfinissant les fonctions qui doivent être définies pour tout type de littérales.
A noter que nous n'avons pas eu le temps d'implémenter les **Littérales Programmes** dans notre application, mais qu'il serait aisé de les rajouter en créant une classe **Program** qui hérite de **Literal** et en (re)définissant les fonctions nécessaires.

— **La classe Calculatrice** : Elle agit comme le "cerveau" de l'application. La classe contrôle tout ce qu'il se passe autour d'elle tels que le **CareTaker**, la **Pile** ou le **Manager**. Ces derniers seraient vus comme les membres du corps que le cerveau contrôle. Dès lors, il serait possible et simple de rajouter des fonctionnalités à la calculatrice en créant un nouveau membre que la classe **Calculatrice** dirigerait.

— **La sauvegarde du contexte** : La sauvegarde du contexte est elle aussi très modulable. En effet, elle est gérée par la classe **QSettings** qui écrit et lit les données dans un fichier *.ini*. Les données sont écrites sous forme de groupes, et chaque donnée est un couple clef-valeur. Ainsi pour la sauvegarde des paramètres, il exite dans le fichier un groupe [*Setting*] avec des données telles que $nbAffichage = 9$ ou $showKeyboard = true$. Il existe aussi un groupe [*Pile*] et un groupe [*Variables*]. Ainsi, en ajoutant des nouvelles fonctionnalités à la calculatrice, il serait possible de sauvegarder et restituer les données en rajoutant simplement un groupe lors de l'écriture des données. De plus, il est possible d'écrire TOUT type de données dans le fichier *.ini*, même les structures créées par l'utilisateur. C'est d'ailleurs comme ceci que nous arrivons à écrire des littérales.
C'est pourquoi il n'y a même pas à se préoccuper de comment stocker les nouvelles données ajoutées à l'application. Il suffit juste de rajouter dans la méthode de sauvegarde de contexte les instructions pour les sauvegarder, QSettings se charge du reste.

— **La classe Manager** qui permet de gérer les variables pourrait également être facilement modulable lors de l'ajout de nouvelles fonctionnalités. En effet, si on voulait gérer les programmes alors il suffirait d'ajouter dans notre manager un tableau regroupant les identificateurs de programmes et leur action. Au delà, on pourrait également imaginer de faire une classe abstraite manager dont hériteraient plusieurs managers (de programme, variable, documents par exemple). Lors ainsi lors de la création d'un nouvel identificateur, le bon manageur serait appelé pour stocker le nouvel élément.

— **L'implémentation du design pattern Factory method** permet de rendre l'application plus évolutive. En effet, si l'on voulait pouvoir saisir dans la calculatrice un nouveau type de littérale ayant été implementé, il suffirait uniquement de modifier la methode *CreerLiteral()* de la classe Factory ainsi que la fonction de la calculatrice permettant de savoir si une chaîne de caractères saisie correspond à une littérale ou non.

— A première vue, le choix d'implémentation **d'une classe Atom** peut être discutable.Cependant, nous avons fait ce choix d'implémentation pour favoriser des ajouts à l'application par la suite. Imaginons par exemple que nous voulions pouvoir sauvegarder des documents ou des graphes dans notre calculatrice et que ces éléments soient identifiés par une chaîne de caractère. il sera beaucoup plus simple de gérer l'identification de ces éléments en les regroupant grâce à la classe Atom plutôt que de les ajouter dans le manager directement.

4 Conclusion

Ce projet nous a permis de mettre en pratique toutes les connaissances acquises en programmation orientée-objet au cours de ce semestre. Le developpement d'une application de A à Z telle qu'UTComputer a été très enrichissant en termes de compétences techniques mais aussi conceptuelles. Nous nous sommes notamment rendu compte de la difficultée à concevoir une architecture "optimale" pour un logiciel/application.