RAPPORT DE PROJET LO21

UTComputer

Semestre P16

Pierre-Aymeric GILLES

Marion CHAN-RENOUS-LEGOUBIN

Table des matières

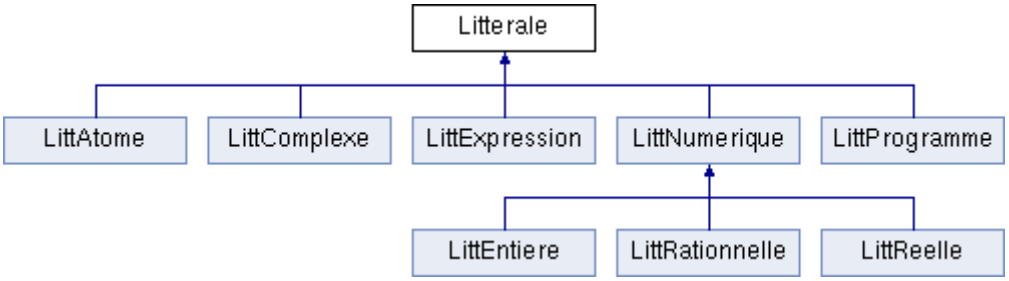
I/ DESCRIPTION DU SUJET

Le but de ce projet est développer un calculateur scientifique permettant de faire des calculs, de stocker et de manipuler des variables et des programmes en utilisant la notation RPN (Reverse Polish Notation). Afin de réaliser ce projet, nous avons utilisé le langage de programmation C++ et le framework Qt.

Ce rapport sera décomposé en plusieurs parties. Dans un premier temps nous allons décrire l'architecture que nous avons choisie pour notre application. Puis, nous allons expliquer en quoi l'architecture permet des évolutions, ce qui est primordial dans le processus de création d'une application. Ensuite, nous allons décrire le fonctionnement global de notre application. Enfin, nous allons expliquer les différentes difficultés au quels nous nous sommes confrontés tout au long de ce projet, et les améliorations possibles.

II/ DESCRIPTION DE L'ARCHITECTURE

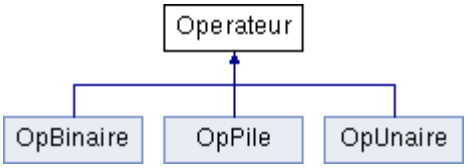
1) Les littérales



Nous avons choisi de définir une classe "mère" Littérale abstraite. Ainsi, toutes les classes définies dans le sujet sont représentées par différentes classes filles. Nous avons également défini une autre classe "mère" LittNumérique, héritant de la classe Littérale, dont héritent les trois classes filles LittEntiere, LittRationnelle et LittReelle.

Les littérales sont créées via une classe appelée FactoryLitterale. Celle-ci permet, à partir d'un string, de créer un pointeur alloué dynamiquement correspondant au type de littérale désiré. Cette classe possède également un attribut de type QVector<Litterale*>, qui permet de désallouer à la fin du programme la totalité des pointeurs alloués dynamiquement.

2) Les opérateurs



Nous avons défini classe "mère" Opérateur. Nous avons décidé de créer trois classes filles, "OpBinaire", "OpUnaire" et "OpPile". Ceci permet de gérer plus simplement les multiplicités de chaque opérateur.

3) La pile

La classe pile est l'élément central de notre architecture. Celle-ci possède un attribut de type QStack<Litterale*> (une stack de pointeurs vers des littérales). Nous avons choisi de stocker des pointeurs de type Litterale* pour que la stack puisse contenir des pointeurs vers n'importe quels objets des classes filles de la classe Litterale. Celle-ci possède également un attribut nbAffiche de type unsigned int, nous permettant de gérer le nombre de littérales à afficher à l'écran.

4) Le contrôleur

Le contrôleur permet de gérer notre application. Celui-ci possède un attribut de type Pile, un attribut de type FactoryLitterale, et un attribut message (string). Ce string permet d'afficher un message à l'écran destiné à l'utilisateur.

5) La gestion des exceptions

Afin de gérer les exceptions dans notre application, nous avons choisi de définir une classe UTComputerException. Celle-ci possède un attribut info de type string. Cette classe permet de récupérer le message de l'exception, et de changer le message du contrôleur pour en informer l'utilisateur.

III/ FONCTIONNEMENT DE L'APPLICATION

1) Traitement des lignes de commandes

Afin de traiter les lignes de commandes entrées par l'utilisateur, le contrôleur de l'application possède une méthode appelée "commande(const QString&s)". Celle-ci prend en argument le string rentré au clavier par l'utilisateur et la traite correctement. Cette méthode permet de rentrer une suite d'opérandes (littérales ou opérateurs) séparés par un espace, et de les traiter une par une. En effet, cette méthode détecte la présence d'espaces dans la ligne de commande, et grâce à plusieurs fonctions prédéfinies de Qt (QStringList::split en particulier), nous pouvons extraire les différents opérandes et les traiter séparément en fonction de leur type.

a) Traitement des littérales

Lorsqu'une littérale est détectée (méthode booléenne Litterale::isLitterale(const QString &s)), la méthode booléenne Litterale::isLitteraleToPush(const QString &s) est appelée. Celle-ci permet de reconnaitre si il s'agit d'une littérale dont le traitement consiste seulement à l'ajouter sur la pile (par exemple dans le cas des littérales numériques). La méthode makeLitt de la classe LitteraleFactory est ainsi appelée, le type de la littérale est détecté grâce aux regex présents dans les méthodes booléennes de chaque classe (par exemple LittEntiere::isLittEntiere(const QString&s)). Le pointeur correspondant au bon type de littérale est alloué dynamiquement, et ajouté à la pile. A chaque allocation dynamique de pointeur, celui-ci est ajouté au tableau de la classe FactoryLitterale.

Si la littérale n'est pas une littérale concernée par la méthode isLitteraleToPush (littAtome ou LittProgramme), des instructions spécifiques sont exécutées. Ainsi, si une littérale atome est détectée, nous allons vérifier si cet atome correspond à l'identificateur de variable (méthode isVariable (const QString& s)) ou de programme (méthode isProgramme(const QString& s)). Si c'est le cas, la variable ou le programme associé à l'identificateur est empilé. Sinon, une nouvelle LittExpression est empilée avec le comme valeur le nom de l'atome entouré de côtes. Si une littérale programme est détectée, celle-ci est ajoutée à la pile sans évaluer son contenu.

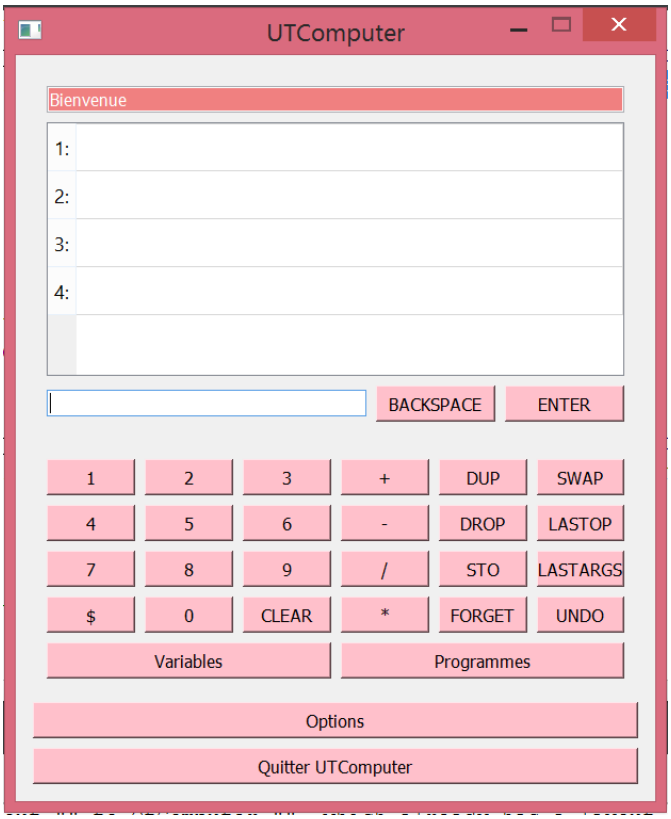
b) Traitement des opérateurs

Afin de gérer les différents opérateurs, nous avons choisi de créer plusieurs QVector afin de les répertorier. Ainsi, il existe un QVector<QString> pour chaque classe d'opérateurs (opPile, opUnaire et opBinaire), initialisé avec les différents opérateurs de l'application.

Ainsi, lorsqu'un opérateur est entré, il est tout d'abord testé via la méthode statique isOperateur(const QString &s). Ensuite, le type de l'opérateur est détecté: chaque classe opérateur possède une méthode

booléenne permettant de dire si le string passé en paramètre est un opérateur de ce type ou non (par exemple OpPile::isOpPile(const QString &s)). Les opérateurs unaires et binaires sont applicables seulement si la taille de la pile le permet, ainsi, si l'opérateur entré est binaire mais que la taille de la pile est seulement de 1, une exception sera lancée. Si la taille de la pile est conforme avec les besoins de l'opérateur, celui-ci sera appliqué à la pile. Chaque classe d'opérateur possède ainsi une fonction qui permet d'appliquer l'opérateur en question. (par exemple applyOpUnaire(Pile &p,FactoryLitterale& fl,const QString& s)). En fonction du string passé en argument, les opérations adéquates seront effectuées. Dans le cas où les types de littérales sur la pile ne correspondent pas à celles nécessaires pour appliquer l'opérateur, une exception sera lancée.

2) L'interface graphique



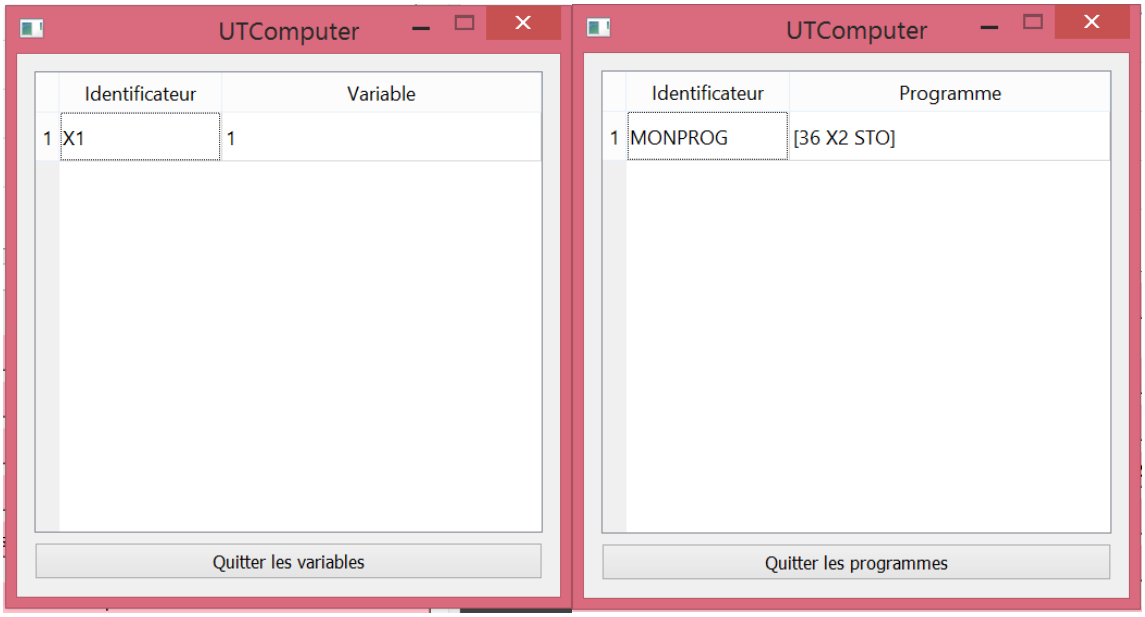
L'interface graphique de notre application est composée d'une fenêtre principale et de trois fenêtres secondaires.

La fenêtre principale est composée d'une zone de texte affichant un message destiné à l'utilisateur. Ce message est mis à jour automatiquement en fonction des différentes commandes entrées par l'utilisateur. La pile de littérales est ensuite affichée. Le nombre de littérales à afficher est initialisé automatiquement à 4.

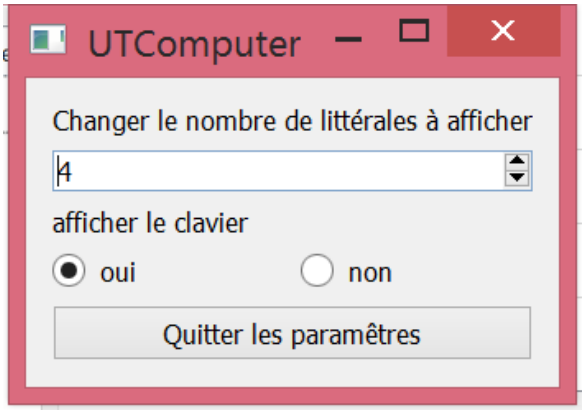
Ensuite se trouve une zone de texte permettant à l'utilisateur de rentrer la ligne de commande souhaitée.

Enfin, le clavier cliquable est automatiquement affiché au lancement de l'application.

Les boutons "Variables", "Programmes" et "Options" permettent d'ouvrir les fenêtres secondaires de l'application.



Les fenêtres "Variables" et "Programmes" permettent d'afficher les identificateurs de variables et de programmes stockés, avec leurs valeurs associés. Ces fenêtres sont mises à jour automatiquement suite au stockage de nouvelles variables ou programmes. Sur ces fenêtres sont présentes un bouton "Quitter" qui permet de revenir à la fenêtre principale.



La fenêtre "Options" permet de gérer les paramètres de l'application. Via cette fenêtre, il est ainsi possible de changer le nombre de littérales à afficher et de choisir si l'on souhaite afficher le clavier cliquable ou non. Un bouton "Quitter' permet de revenir à la fenêtre principale.

IV/ POSSIBILITES EVOLUTIVES DE NOTRE ARCHITECTURE

1) Littérales

Nous avons défini une classe "mère" abstraite Littérale. Celle-ci permet un ajout simplifié de nouvelles littérales. Il n'y aura ainsi qu'un nombre raisonnable d'ajouts à effectuer pour utiliser un nouveau type de littérales :

- ajouter la déclaration de la classe correspondant au nouveau type dans le fichier litterale.h ainsi que la définition de ses méthodes dans le fichier litterale.cpp
- ajouter une méthode « create » dans la FactoryLitterale qui permettra la création de la littérale à partir d'un string fourni en argument, ainsi que l'allocation dynamique du pointeur associé à cette littérale
- ajouter l'interaction de ce nouveau type de littérale avec les différents opérateurs et les autres types de littérales dans les opérateurs concernés dans le fichier operateur.h

2) Operateurs

Nous avons choisi de classer les opérateurs selon leur caractère unaire ou binaire, à l'exception des opérateurs de pile, dont l'application n'a pas d'impact direct sur les littérales, et de l'opérateur EVAL. Cette classification permet de rassembler des opérateurs dont l'application est similaire et permet donc de simplifier le traitement de ces opérateurs.

Par exemple, pour un opérateur binaire, on récuperera toujours les deux dernières littérales de la pile afin de les traiter.

Nous avons par ailleurs rassemblé les désignations ces opérateurs dans des vector d'opérateurs.

Pour ajouter des opérateurs à l'application, il suffit donc d'ajouter la désignation de l'opérateur dans le vector opérateur correspondant et d'ajouter le code correspondant à son application dans la fonction applyOp correspondante.

V/ DIFFICULTES RENCONTREES ET AMELIORATIONS POSSIBLES

Nous avons éprouvé quelques difficultés à débuter le projet. Nous avons passé beaucoup de temps à chercher quelle architecture serait la plus efficace pour répondre au cahier des charges proposé.

Nous nous sommes cependant rendu compte en commençant l'implémentation que ce qui paraissait bien sur le papier n'était pas nécessairement aussi efficace ou aussi pratique à implémenter que nous l'espérions et nous avons dû revoir et ajuster notre architecture en conséquence. Par exemple, nous avions pensé au départ, à regrouper les opérateurs comme indiqué dans le sujet (opérateurs numériques, logiques, conditionnels, boucle…) mais après avoir commencé l'implémentation de ces classes et des méthodes associées, nous nous sommes rendus compte que les regrouper en seulement trois classes était plus adéquat.

Avec du recul, nous avons réalisé que nous aurions dû commencer à tester nos idées plus tôt, passer plus rapidement de la théorie à la pratique. Nous avons malheureusement perdu du temps et n'avons pas pu implémenter toutes les fonctionnalités de manière optimale.

Voici quelques points qu'il aurait été possible d'améliorer :

1) Memento pour UNDO/REDO

Dans l'état actuel du programme, l'opérateur UNDO permet d'annuler l'action du dernier opérateur appliqué, à condition qu'aucune littérale n'ait été ajoutée entre temps.

L'implémentation du design pattern memento aurait permis de rendre l'utilisation de l'opérateur UNDO plus conforme à celle proposée par le sujet (rétablir l'état du calculateur avant la dernière opération) et aurait permis l'implémentation de l'opérateur REDO.

2) Design pattern singleton

L'implémentation du design pattern singleton sur les classes de notre programme aurait permis de s'assurer que nos classes ne sont instanciées qu'une unique fois et d'assurer ainsi la cohérence du programme.

3) Littérales numériques en complexes

Nous aurions pu réaliser une implémentation différente des classes littérales numériques et complexes en considérant toutes littérales numériques comme des complexes avec dénominateur et numérateur pour la partie imaginaire et la partie réelle. Il n'y aurait alors plus eu qu'un seul type LitteraleNumerique à la place des types entiers, rationnels, réels et complexes actuels. L'application des opérateurs, notamment numériques, en aurait grandement facilitée.

Nous n'avons réalisé cette possibilité que trop tard et il n'était plus possible de revenir sur l'implémentation des littérales.

VI/ CONCLUSION

Ce projet a été une expérience enrichissante pour nous. En effet, il nous a permis de mettre en pratique les concepts théoriques appris en cours. Nous nous sentons désormais plus à l'aise avec la programmation en C++, et l'utilisation du framework Qt pour le développement de l'interface graphique du projet. Il nous a également montré la difficulté de gérer un projet informatique en binôme : utilisation de Github, réunions hebdomadaires… mais également la satisfaction d'aboutir à un projet crée par nous-même.