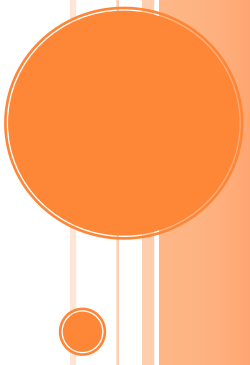
UT COMPUTER

Projet de LO21

UTComputer est une calculette scientifique programmable fonctionnant en notation polonaise inversée. Développée essentiellement en C++ et avec le framework Qt.



Durocher Alexis – Gerbaux Irvin

Semestre P16

UT Computer

Projet de LO21

I] Architecture

L'objectif de ce projet était de comprendre et maîtriser les rouages de la programmation orientée objet. Ainsi, nous avons cherché tout au long de ce projet à suivre le fil rouge de la modularité. En effet, un logiciel dont le code et l'architecture ont été bien pensé est un logiciel qui peut évoluer dans le temps sans impacter les versions antérieures.

L'idée, donc, était de conceptualiser une architecture logicielle la plus modulable possible.

Cette partie sera consacrée à l'exposé de cette architecture et à la justification de nos choix. En premier lieu, nous aborderons une description générale de cette dernière puis nous aborderons les différentes parties qui la composent plus en détail.

Vue générale

Notre architecture se décompose en 5 parties relativement distinctes. Les littéraux, les opérateurs, l'interface homme-machine (front-end), la gestion back-end et le stockage en mémoire.

Littérales

Une littérale est une entité qui sera créée dynamiquement lors de l'utilisation de la calculette. Ainsi, selon des caractéristiques particulières (propre à la syntaxe de l'énoncé), une littérale du type correspondant sera allouée.

Afin de regrouper les littérales, selon des caractéristiques communes, nous avons choisi d'utiliser de nombreuses classes abstraites qui définissent leurs squelettes. Nous retrouvons la classe mère Littérale, mère de LitAtome, LitProgramme, et LitExpression mais aussi de Nombres qui définit le squelette de LitNumerique (Entier, Réelle et Naturelle) et celui de Complexe.

Nous avons séparé Complexe de LitNumerique pour des raisons de non-compatibilité de calculs. En effet, certaines opérations peuvent s'appliquer sur toutes les LitNumerique mais pas sur des Complexe.

Les littérales possèdent toutes une méthode *toString()* qui permet leur affichage dans la pile de l'IHM.

// Autre méthodes communes ?

Opérateurs

Pour effectuer les opérations sur les littérales, nous avions deux possibilités.

La première était de surcharger des méthodes dans les différentes classes littérales. Ce choix paraîtrait le plus naturel d'un point de vue sémantique mais pose des problèmes de modularité. En effet, l'ajout d'une nouvelle opération impliquerait une modification de toutes les classes littérales concernées.

La seconde possibilité était de créer un ensemble de classes opérateur composées d'arguments Litterale et qui exécutent le bon algorithme à l'appel d'une unique méthode commune *executer()*. On retrouvera ici une adaptation des designs patterns **strategy** et **template method**.

La méthode *executer()* oriente les arguments (litterales composantes de l'objet opérateur) vers des fonctions virtuelles pures adaptée. Ce choix sera déterminé //

Nous avons donc choisi la seconde architecture qui apporte l'avantage de la modularité et donc de l'évolution facile dans le temps. En effet, pour ajouter une nouvelle opération à la calculette, il suffira d'implémenter une nouvelle classe opérateur qui héritera du squelette commun des autres et qui ne fera que redéfinir son algorithme propre.

Toutefois, choisir cette solution posait un problème. En effet, il fallait trouver un moyen de créer un objet opérateur adéquat à la commande de l'utilisateur et ce, dynamiquement. Nous avons choisi d'implémenter le design pattern **factory** afin de résoudre ce problème. Ce dernier, nous permet de générer un opérateur selon un nom qui lui est propre depuis le controleur. La **factory** retourne un pointeur de type Operateur (classe abstraite) qui pointe en réalité vers l'opérateur voulu.

Tous les opérateurs numériques/logiques possèdent un attribut **taille** qui dépend de leur cardinalité et qui est définit depuis les constructeurs des classes filles (binaire /unaire). Cette attribut sera récupéré depuis le pointeur Operateur (renvoyé par la **factory**) et permettra de définir le nombre de littérale à récupérer dans la pile avant l'exécution.

On parle de **composition** entre opérateur et littérale. En effet, un opérateur sera responsable de la désallocation mémoire des littéraux qu'il récupère en argument. Cependant la nouvelle littérale résultante de l'execution de l'opération sera indépendante de la durée de vie de l'opérateur exécutant.

Chaque opérateur est desalloué après son execution (le destructeur se chargera de la désallocation des arguments littéraux).

Les opérateurs numériques sont distingués en différentes catégories selon leurs caractéristiques. Cette distinction permettra de les catégoriser et donc de les reconnaitre lors de dynamic cast.

Aussi certaines méthodes virtuelles pures ne sont pas partagées entre les différentes catégories d'opérateurs (cas des opérateurs logiques et opérateurs classiques par exemple). On a donc trouvé plus intéressant de les distinguer afin d'éviter de redéfinir des fonctions inutiles.

// Developper différents opérateurs

IHM : Interface Homme-Machine

C'est un QObject qui permet à l'utilisateur de rentrer des commandes et d'observer l'évaluation d'opérations sur les littérales stockées sur la pile.

Un message d'erreur indique à l'utilisateur si la commande est erronée. De plus l'utilisateur peut observer directement les variables stockées avec l'opérateur STO, grâçe aux vues secondaire de l'application.

//EDIT programme ?

Back-end

Nous appellerons Back-end, toute la partie analyse d'une commande, et travail sur la pile. Cette partie est gérée par un contrôleur, un litteraleManager et une pile.

La pile

La pile est la structure de donnée principale de notre logiciel. C'est elle qui représente « l'état » de la calculette à un instant t.

La pile est en relation avec l'IHM car c'est elle qui contient les données à afficher dans le tableau Qt et c'est elle qui déclenche le **reset** de l'affichage à chaque appel de son slot : *modificationEtat()*. Finalement cette pile met aussi à jour le message affiché dans la ligne Qt réservée à cet effet.

Chaque littérale de la pile est encapsulée dans un objet de type item. Ainsi la pile n'est pas responsable de la destruction ni de la création des littérale mais bien seulement des items qui ne servent qu'à les encapsuler.

Le litteralManager

Le litteralManager est un objet qui appartient au contrôleur directement. Ses méthodes permettent de fabriquer les littérales adéquates et d'effectuer des vérifications de validité.

Le contrôleur

Le contrôleur est l'objet central de notre logiciel. C'est lui qui met en relation chaque partie avec une autre et gère le stockage en mémoire.

Il réagit directement après un input sur la ligne de commande de l'IHM. C'est lui qui va récupérer la chaîne de caractère, la scinder en différents opérandes et appliqués **l'algorithme générale** sur chacun d'entre eux.

Le contrôleur définit dans sa méthode *commande()* **l'algorithme générale** de notre calculette.

Explicitation de l'algorithme générale commande :

Contrôleur :: commande(Qstring& v (*on considèrera qu'elle aura déjà été scindée))

```
Test = true ;
Try
        Crée Opérateur via la fabrique ;
        Test = false ;
        Si taille_pile >= taille_opérateur
                For i : 0 à taille_opérateur
                        arg = dépiler(pile)
                        Ajouter argument arg à Operateur
                Créer Litterale res = Opérateur.execute()
                Empiler res
        Sinon
                Message utilisateur : Impossible, pas assez d'arguments dans la pile

        Destruction Opérateur


Catch
If (test)

        Création Litterale res = LitManager.addLitterale(v)
        If (res <> null)
                Empiler res
        Else If v est un atome
                Recuperer la littérale lit
                Empiler la littérlale lit
        Else If v est une variable de programme
                Recuperer la littérale programme
                Evaluer le programme
                Empiler dans la pile le resultat de l'évaluation
        Else
                Message Utilisateur : Erreur de syntaxe
Else
Message Utilisateur :   Erreur
End.
```