UT Computer

Projet de LO21

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

Durocher Alexis – Gerbaux Irvin

Semestre P16

UT Computer

Projet de LO21

Table des matières

I] ARCHITECTURE

L'objectif de ce projet était de comprendre et maîtriser les rouages de la programmation orientée objet. Ainsi, nous avons cherché 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 de ce dernier.

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

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.

Cette partie sera consacrée à l'exposé de cette architecture et à la justification de nos choix. Nous aborderons à chaque fois une description générale de chaque partie puis nous en détaillerons les composantes.

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 LitNumeriques mais pas sur des objets de type Complexe.

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

Littérales Expressions

Nos littérales expressions sont construites de manière assez particulière. En effet, nous avons trouvé judicieux de les rendre plus riches que de simple chaine de caractères. Une littérale expression est composée d'un vector d'Operande* qui contient la version polonaise inversée de l'expression considérée (qui elle est en notation classique). Nous avons implémenté un algorithme qui vérifie que la syntaxe d'une expression est bonne, et qui construit en parallèle le vector en notation polonaise inversée (depuis le litteraleManager, *verifExpressionValide)*. Finalement, la littérale expression se voit associer ce vector comme attribut composite.

Lors de son évaluation, l'opérateur EVAL récupèrera le vector de l'expression et c'est un autre algorithme récursif qui va dépiler ce dernier (à la manière d'une structure FIFO) en appliquant l'évaluation sur chaque opérande contenu.

Voici la description de cet algorithme :

```
Litterale * Evaluation( QVector<Operande*>& vec)

Operande = vec.Element_front

Dépiler_front vector

        Si Operande = Litterale

                Si Litterale = Expression

                        Créer OpEval

                        Ajouter argument Expression à OpEval

                        Retourner OpEval->executer()

                Sinon retourner Litterale

        Sinon // Operande = Operateur

        Pour i allant de 0 à taille de Operateur

        Ajouter argument : Evaluation(vec) à Operateur// appel recursif

        Fin Pour

        Retourner Operateur -> executer ()

Fin
```

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.

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 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'exécution de l'opération sera indépendante de la durée de vie de l'opérateur exécutant.

Chaque opérateur est desalloué après son exécution (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.

Opérateurs classiques

Ce sont tous les opérateurs numériques (symbole ou caractère) qui fonctionnent selon le même principe.

Les opérateurs classiques possèdent tous une définition différentes des méthodes « *actionNum* » dont ils existent autant de surcharges que de cas différents (Entier/Naturel , Entier/Entier etc…). Ces méthodes seront appelées en prenant en paramètres les bons arguments via la méthode *fonctionNum / fonctionExpression* (elle-même appelé depuis *executer()).*

Opérateurs logiques

Ce sont tous les opérateurs logiques. Ils ne possèdent pas de surcharge de méthodes *actionNum()* comme les classiques. En effet, ils ne font pas de distinction sur leurs arguments puisque ils n'utilise que la méthode *getValue()* commune à toutes les littérales Numériques pour appliqués des opérations logiques. Chaque opérateurs redéfinira la fonction virtuelle pure *actionLogiNumerique()* qui effectuera l'algorithme souhaité lors de son appel via *executer().*

Opérateurs de Pile

Les opérateurs de pile vont redéfinir la méthode virtuelle *addArg()* commune à tous les autres opérateurs car ces derniers opèrent sur la pile directement et non sur des littéraux. Chaque opérateur de Pile se verra donc affecter la pile principale en paramètre avant de lui affecter le bon algorithme.

Nous ferons la distinction entre Operateurs de pile et les autres au niveau du destructeur de la classe mère Operateur afin de ne pas détruire la pile lors de la destruction d'un opérateur de pile.

De la même manière que les opérateurs logiques, ces derniers vont uniquement redéfinir leur algorithme propre qui correspondra à la méthode virtuelle pure de la classe mère OpPile : *exectuerPile()* appelé par la méthode *executer()*

Opérateurs Autres

Nous avons regroupé dans cette catégorie tous les autres opérateurs (STO et DOL) qui avait leur propre particularité.

Finalement le polymorphisme est utilisé sur plusieurs étages afin de garder un modèle générique en surface (travaille sur Litterale ou Operateur*) tout en limitant au maximum la redondance de ligne de code plus en profondeur.*

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, grâce aux vues secondaire de l'application.

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 litteral manager

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 général** sur chacun d'entre eux.

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

Explicitation de l'algorithme général 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 && contenu dans la mémoire
                Recuperer la littérale lit dans le stock mémoire
                Empiler la littérlale lit
        Else If v est une variable de programme && contenu dans la mémoire
                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.
```

Stockage mémoire

Nous avons utilisé une base de données SQL Lite pour stocker les variables en dehors de l'application.

Afin d'y accéder facilement en interne, nous avons une map<QString, Nombres*> qui est initialisée dès l'ouverture de l'appli et qui correspondra à la « BDD » interne.

Le stockage en mémoire de programme aurait suivi le même principe mais nous n'avons pas eu le temps de l'implémenter.

II] EVOLUTION

Dans cette partie nous allons vous montrer un exemple concret qui permettra de démontrer la modularité de notre architecture ainsi que ses capacités évolutives.

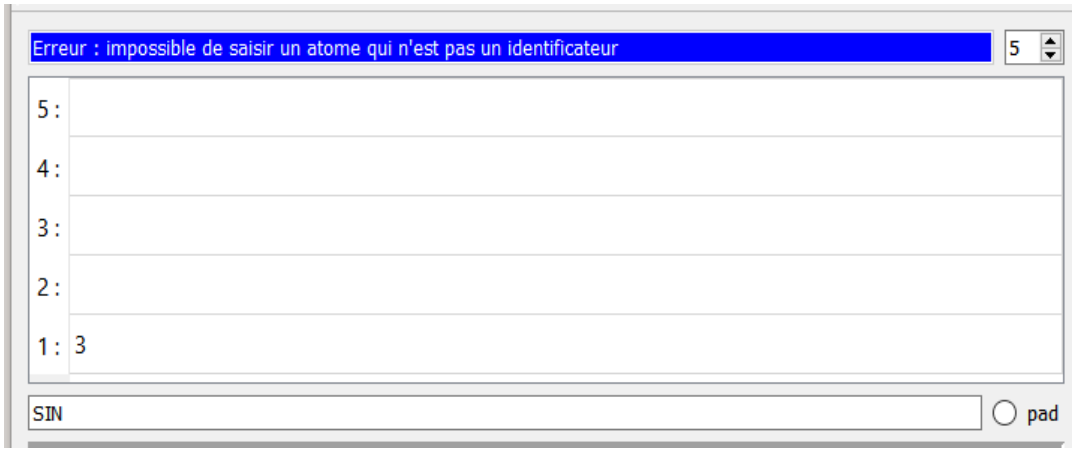
L'évolution de cette calculette est l'ajout de nouvelle fonctionnalité. Et dans notre cas, une nouvelle fonctionnalité se résume à ajouter un nouvel opérateur. L'ajout de ce nouvel opérateur, n'aura aucun impact sur les anciens, et encore moins sur le reste de l'architecture car son modèle générique est déjà définie dans la classe abstraite Operateur.

L'ensemble de l'architecture travaillant avec des pointeurs Operateurs, il n'y aura aucune modification à effectuer.

Exemple de sinus(3)

L'énoncé proposait d'ajouter des opérateurs optionnels que nous n'avons malheureusement pas eu le temps d'implémenter. Toutefois, notre architecture permet un ajout très rapide de nouveaux opérateurs et nous allons vous le montrer en prenant l'exemple de SIN : qui calcule le sinus d'une littérale Numérique en radian.

Avant l'implémentation de la classe OpSin, on peut voir que la calculette considère la commande SIN comme un simple atome.



Avant d'ajouter un nouvel opérateur, il faut se poser deux questions :

- Quelle est sa taille (binaire/unaire) ?
- A quelle catégorie il appartient (symbole/caractère/logique/pile..) ?

En fonction de la réponse à ces deux questions, il suffira de faire hériter le nouvel opérateur de la classe de base qui lui convient et de redéfinir les fonctions virtuelles pures qui définiront l'opération en question.

En prenant l'exemple de SIN, nous avons un opérateur unaire (doit donc hériter d'OPUNAIRE), qui est un opérateur numérique. Tout son fonctionnement (récupération des arguments, moment d'utilisation et choix de la bonne opération) est donc déjà implémenté pour son exécution. Il suffira de définir les méthodes *actionNum()* surchargés pour appliquer la fonction sinus de manière appropriée sur la **valeur** de la littérale passé en argument (récupéré sur la pile). Cette fonction renverra une nouvelle Littérale résultante dont le type sera choisie selon la valeur obtenue.

Une fois cette classe définie, elle sera « linkée » à une partie de l'architecture grâce à l'héritage mais l'opérateur ne pourra toujours pas être ni crée ni reconnu depuis la commande SIN.

En effet, c'est notre factory qui se charge de la création d'un opérateur en réaction à un input QString. Cette factory possède une map qui associe à chaque clé (ici 'SIN') le constructeur d'un operateurFactory (ici OpSinFactory).

Pour que la factory puisse créer un objet de type OpSin, il va donc falloir ajouter cette classe OpSinFactory (qui n'est qu'un intermédiaire entre la map et le OpSin) ainsi que le couple (SIN, (new OpSinFactory)) à la map :

```
factories["SIN"] = new OpSinFactory()
```

La nouvelle classe OpSin est alors entièrement linkée à l'intégralité de l'architecture qui n'aura pas été modifiée (sauf ajout).

Nous pouvons donc maintenant entrer en ligne de commande SIN.

Le contrôleur envoie à la factory cette clé 'SIN'. Un nouvel opérateur OpSin est fabriqué et renvoyé au contrôleur sous la forme d'un Operateur*. Le contrôleur lui passe en argument la littérale entière 3 et finalement, le contrôleur appelle la méthode exécuter () sur ce dernier. La littérale résultante est alors push dans la pile et l'opérateur OpSin détruit (avec l'ancienne littérale qui le compose). (cf. diagramme de séquence)

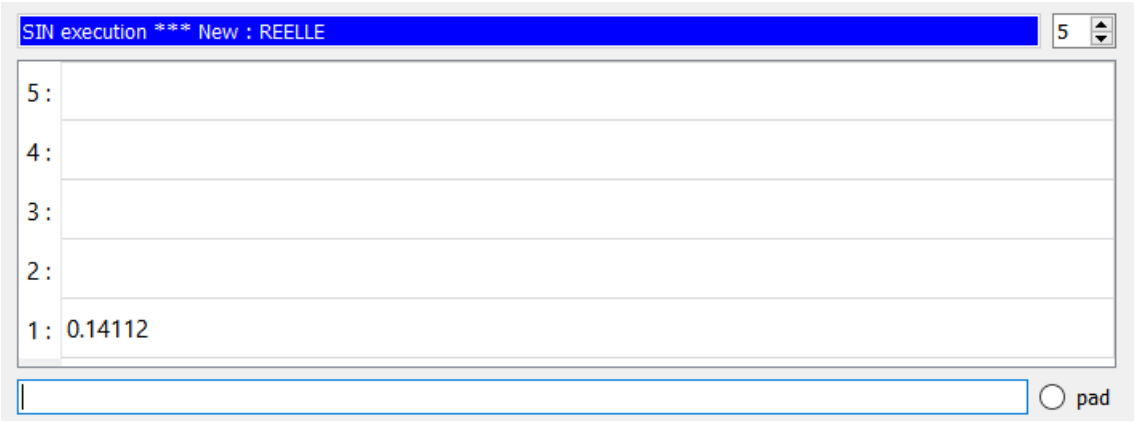
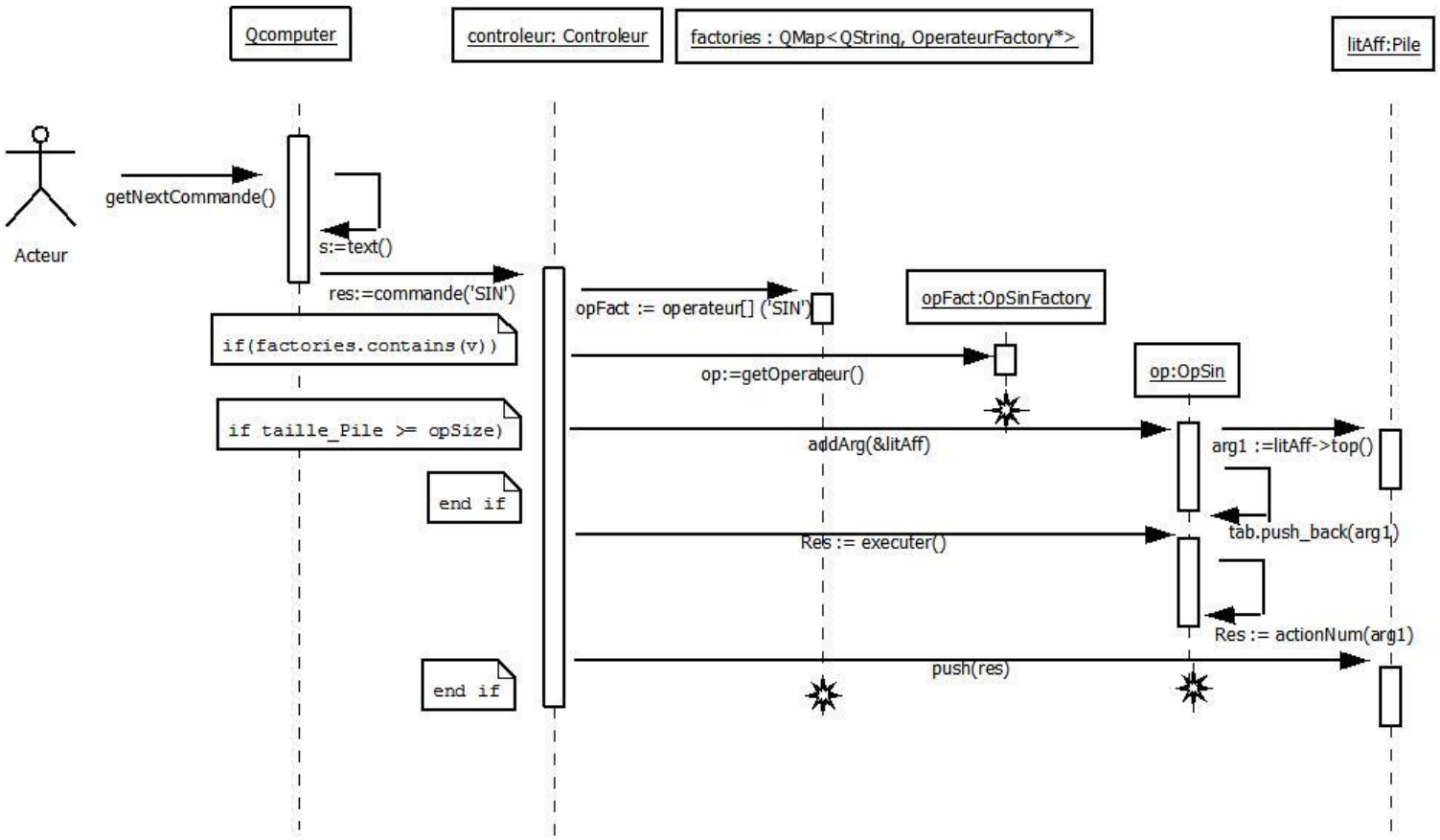


Diagramme de séquence

Ce diagramme de séquence représente le scénario (Sin(3)) décrit ci-dessus. Afin de simplifier la compréhension générale nous n'avons pas intégrer le litterale Manager dans ce scenario. Deplus, nous considérons que la pile (litAff) le contrôleur (controleur) et la factory ont déjà été créées.

L'utilisateur aura déjà entré la commande 3 (créée la littérale entière 3).



Conclusion

Ce projet aura été très riche sur deux niveaux.

Premièrement, nous avons énormément appris lors de la phase de conception de l'architecture. Nous avons beaucoup apprécié chercher les solutions les plus « modulables », et les plus « propres » en appliquant des techniques vues en cours mais aussi d'autres découvertes par nous-même. Ce projet nous aura réellement appris à 'penser Objet'.

Aussi, l'envergure de ce projet, représentait un réel défis et nous avons donc utilisé des plateformes telles que Github pour partager / mettre à jour notre code et travailler en équipe. Ce travail nous aura donc appris en parallèle à collaborer informatiquement de manière efficace et « professionnelle ».

ANNEXE

Notice d'utilisation du logiciel UTComputer

Réglage

L'unique réglage est d'entrer le path (du répertoire contenant les codes sources) en paramètre de la fonction *setDatabaseName* suivi de **/calculette.db**.

(fichier : **storage.cpp**, → constructeur de DbManager)

Attention au sens des / :

Ex :

```
db.setDatabaseName("C:/Users/Alexis/Desktop/You.Thea.sea/Compiègne2015/P16/LO21/UT_Computer/src/calculette.db");
```

Utilisation

La ligne de commande peut contenir une opération complète à effectuer. Toutefois, la commande ne sait interpréter qu'une par une les différents opérandes de l'opération. Il faudra donc appuyer autant de fois sur entrée qu'il y a d'opérandes (l'affichage sera mis à jour de manière naturelle pour vous aider).

La vue des variables est un QFrame qui « pop » et vient en complément de la vue principale. Ainsi, vous pouvez voir directement les nouvelles variables stockés en mémoires lors de l'exécution de STO.

UML préliminaire

Voici l'UML que nous avons fait au tout début du projet, lors de la phase de conception de notre architecture. Cet UML ne représente que l'idée générale de notre logiciel, l'architecture a été modifiée de nombreuses fois depuis.