Blanquet Clément – Martinache Grégoire Groupe du mercredi matin

Projet de LO21

Rendu :

Merci de trouver ci joint :
→ Le projet couvrant toutes les fonctionnalités demandées (sauf les facultatives).
=> Dossier « Code Source »

→ Le rapport sur la facon dont nous avons élaborés l'architecture ainsi que les explications sur nos choix de conceptions
=> Page suivante

→ Une vidéo explicative sur le fonctionnement de notre calculatrice
=> Dossier « Vidéo »

→ Une documentation Doxygen complète
=> Dossier « Documentation »

→ Le fichier de configuration pour la génération de la documentation
=> Fichier « Doxyfile »

→ Les UMLs integrés dans ce rapport sont également disponibles en PDF
=> Dossier « UML »

Note concernant le projet Qt :

Le fichier permettant de charger notre projet se trouve dans 'CodeSource/Qt/Qt.pro'

Blanquet Clément – Martinache Grégoire Groupe du mercredi matin

Tout projet doit commencer par une réflexion sur l'architecture et la façon dont les différents composants interagissent entre eux.

Partie A : L'architecture - Fonctionnement de l'application

1) Architecture globale des litterales (voir UML)

Pour gérer les entiers, rationnels, réels et complexes, nous avons créé une classe pour chaque. Ainsi, les entiers n'ont comme argument qu'un entier, les rationnels deux entiers et les réels un double. En ce qui concerne les complexes, ils sont composés de deux littérales numériques (c'est à dire entière, rationnelle ou réelle) qui représentent la partie réelle et imaginaire.
Tous les opérateurs de base (+,-,*,/) sont définis – par surcharge des opérateurs – comme méthodes de ces classes.

Ensuite, on a les classes Expression et Programme qui contiennent chacune un string en argument. Un objet Expression sera toujours créé entre quotes, et un objet Programme entre crochets.

Enfin, la classe Atome, ayant comme argument un string et un pointeur vers une Litterale, permet notamment de gérer la création de variables avec l'opérateur STO, qui sera détaillé juste après ;

2) Processus général de traitement :

Après discussions, nous avons déterminé le fonctionnement de la calculatrice :

---

1) L'utilisateur entre une commande. Ex : 1 1 + 1 -.

2) Cette commande est divisée selon ses espaces pour produire une commande par opérande. Dans notre exemple, on a donc 5 commandes : 1,1,+,1,-.

3) Chacune des commandes est analysée :
→ Si c'est une littérale, elle est envoyée à FactoryLitterale qui transformera le string en une littérale du bon type.
→ Si c'est un opérateur, il est envoyé à FactoryOperateur qui transformera le string en un opérateur de la bonne arité (0, 1 ou 2).

4) A ce stade, on a donc une suite d'opérandes. On les envoit à l'exécution du contrôleur. Lorsqu'on rencontre un opérateur, on l'exécute et on continue

l’empilement. Dans notre exemple, 1 est empilé, 1 est empilé puis + est empilé. + est détecté comme opérateur et est donc exécuté. A la fin de cette exécution, la pile contient 2. On continue l’empilement avec 1 puis avec -. Puisque – est un opérateur, il est exécuté et la pile contient donc 1.

5) On affiche la pile à l’utilisateur.

Ce choix de fonctionnement présente des avantages et des inconvénients :
→ Un opérateur peut avoir deux arités puisque la séparation se fait grâce aux espaces. Ainsi -2 sera évalué comme étant l’opérateur NEG alors que – 2 empilera d’abord – puis 2.

→ L’avantage précédent implique de bien séparer les opérateurs par des espaces. Ainsi, 1 1+ n’est pas valide.

→ Il a été décidé que la commande n’est envoyé au contrôleur que lorsque l’utilisateur appuie sur ‘Entrée’. Après différents essais, cela à été jugé beaucoup plus simple que d’envoyer la commande à chaque fois qu’un opérateur est cliqué, notamment pour permettre à l’utilisateur de faire des copier coller.

A la suite des ces discussions, nous avons écrit l’UML qui à été amélioré et étoffé tout au long du projet.

Blanquet Clément – Martinache Grégoire Groupe du mercredi matin


Operande
Expression
Complexe
Reel
Entiere
Rationelle
Litterale
ExpressionPart
LitteraleNumerique
Atome
AtomeManager
Programme
Operateur
Binaire
Unaire
Zero
NB : Seules les classes significatives dans l'architecture sont présentées ici. Les classes concrètes implémentant l'interface graphique ne figurent pas.
La classe AtomeManager possède également un itérateur non représenté ici.

Remarques concernant l'UML :

→ Toutes les opérandes possèdent la fonction d'affichage qui appelle à la fonction toString(), redéfinie dans toutes les classes concrètes.

→ Les opérateurs de notre projet sont soit d'arité 0 (Ex : CLEAR), soit d'arité 1 (Ex : NEG), soit d'arité 2 (Ex : +). Lors de la création d'un opérateur, celui-ci est donc contenu dans un objet correspondant à son arité. Un simple dynamic_cast permet ainsi de connaître l'arité de l'opérateur.

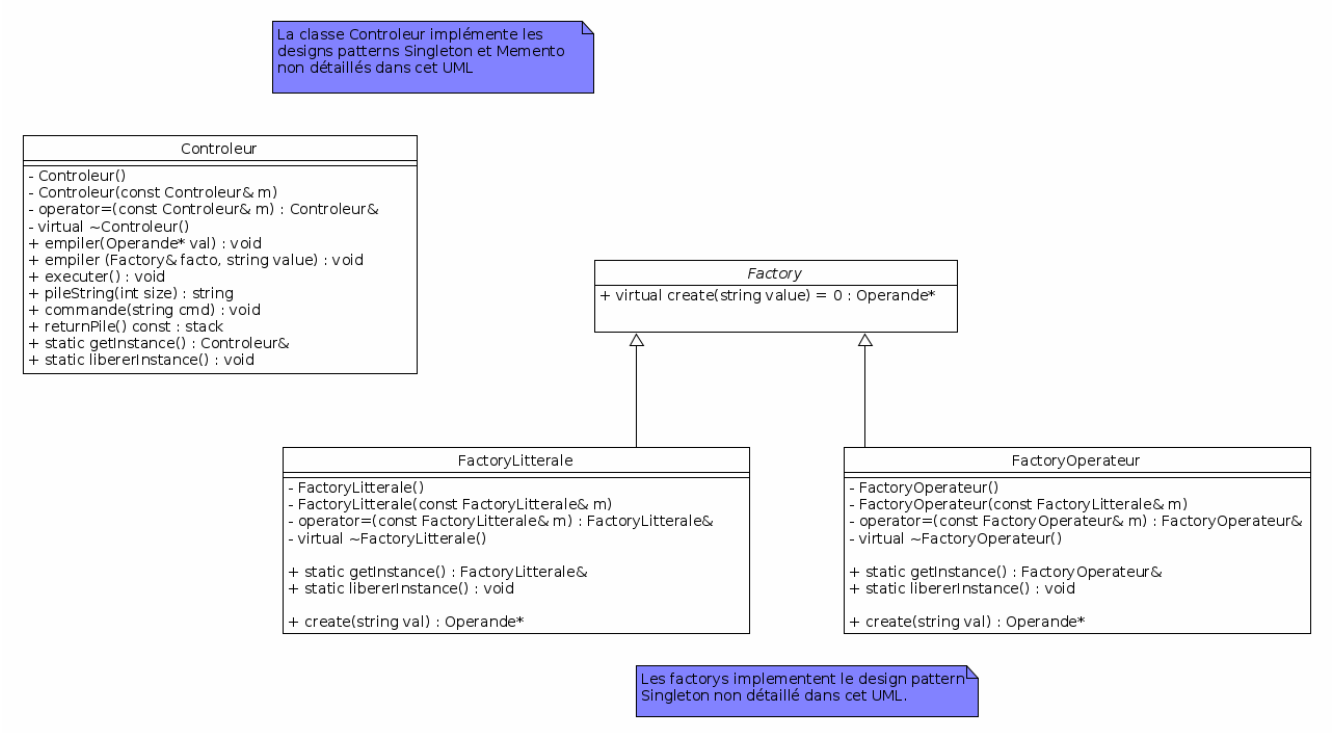
3) Les factorys et le controleur:

Une fois les littérales et leurs opérateurs implémentés, nous avons travaillé sur la création des factorys et du controleur. Les rôles sont repartis ainsi :

→ Les factorys doivent être capable de construire le bon type d'opérande à partir d'un string. Par exemple : 3.1$4/5 devra créer un reel (3.1) et un rationnel(4/5) qui serviront à la creation d'une littérale complexe (3.1$4/5)

→ Le controleur empile les littérales construites dans une factory, et lorsqu'un opérateur est empilé, le controleur fait les actions necessaires. Il s'agit enfaite du « coeur » de la calculatrice. C'est le controleur qui gère la pile.

Voici l'UML qui à était construit à la suite de ces remarques :



4) Explication des fonctions avancées

a) Opérateur STO

L'opérateur STO permet de stocker des entiers, rationnels, réels, complexes, expressions ou programmes dans des variables, ce qui permet de les conserver. Pour cela, l'utilisateur entre une commande du type « 3 X1 STO », ce qui signifie « je veux stocker 3 dans la variable X1 ». A ce moment là, la procédure est la suivante :

- On crée un nouvel atome ayant pour nom X1
- On le lie (par l'intermédiaire du pointeur dans Atome) à la littérale (ici donc un pointeur vers 3), e$
- On ajoute cette atome à la liste de tous les atomes. On gère cela grâce à une classe AtomeManager, dans laquelle nous avons implémenté un iterator pour pouvoir parcourir tous les atomes.

A partir de là, la variable a été créée.
Par la suite, lorsque l'utilisateur entre un atome dans la barre de commande (c'est à dire une suite de majuscules et de chiffres), le traitement est le suivant :

- Si cet atome existe déjà (recherche dans AtomeManager avec l'iterator), alors on empile sa valeur
- Sinon on crée une nouvelle expression avec le nom de cet atome

b) Opérateur EVAL

L'opérateur EVAL, unaire, ne s'applique qu'aux expressions et aux programmes. Pour les programmes, il suffit de retirer les crochets autour de celui ci.
Pour les expressions, la manipulation est d'une complexité bien supérieure. Nous avons développé une méthode qui fonctionne plutôt bien, mais qui est difficile à expliquer. Le principe est le suivant :
- Quand on tombe sur un opérateur dans la chaine de caractères, on va l'inverser avec les chiffres suivant. Pour prendre un exemple, si on a une expression '1+1', on va construire un nouveau string qui aura cette forme : 1 1 +. Il n'y aura plus qu'à envoyer ce string à la commande pour que tout s'empile correctement et s'exécute (voir le point « processus général de traitement). Pour gérer les priorités, par exemple si on a '1+2*3', on a une vérification qui permet de ne pas obtenir 1 2 + 3 * (ce qui est faux), mais 1 2 3 * +. En fait, si on a un * ou un / (qui sont prioritaires sur + et -), on sauvegarde l'opérateur précédent, et on l'ajoute à la fin.
- Quand on tombe sur des parenthèses, on évalue l'intérieur de cette parenthèse (il peut y avoir des parenthèses imbriquées, nous avons géré ce cas), et on la remplace par son évaluation. Exemple : '2*(3+4)' va devenir 2 3 4 + *
- Nous avons aussi géré d'autres choses, comme les opérateurs écrits en toutes lettres (DIV, MOD… etc), qui sont détectés s'ils sont dans l'expression. Il en est de même pour les '-' qui ne sont pas des opérateurs, par exemple : '-2+3' sera bien évalué, car l'évaluation rendra le string : 2 NEG 3 +.

c) Les opérations avec les expressions

Il est possible d'appliquer entre deux expressions tous les opérateurs. On peut aussi entre expressions et autres littérales numériques (et complexes).

5) La sauvegarde du contexte, les fonctions undo/redo

La sauvegarde du contexte est implémenté sous la forme de 2 routines qui sont exécutées à la fermeture de la fenêtre :

- Sauvegarde de la pile : La pile est dépilé dans un fichier dataPile.dat
- Sauvegarde des atomes : L'ensemble des atomes sont parcourus avec l'itérateur fourni par la classe AtomeManager. La valeur et le nom de l'atome est stocké dans un fichier texte. Ces informations sont stockées dans un fichier dataAtome.dat

Ces fichiers, ainsi que le XML de réglages, sont chargés à l'ouverture de fenêtre.

Les fonctions undo/redo ont été implémentées à l'aide du design pattern 'Memento'.
Notre mémento implémente les 3 classes prévues dans ce design pattern

(Memento, Originator (Le controleur) ainsi que le CareTaker). Le mémento est constitué d'un vector de stack. Les piles peuvent dont être parcourues simplement dans le temps. Une sauvegarde est effectuée à chaque fois qu'une commande est envoyée.

6) Les fenêtres supplémentaires

Les fenêtres d'éditions de programmes et d'atomes sont de simples implémentations graphiques des fonctions de atomes manager.

La fenêtre de réglages est plus compliqué puisqu'elle doit sauvegarder l'ensemble des réglages dans un fichier, afin que ceux-ci soit sauvegardés. Nous avons décidé d'utiliser le format XML qui s'adaptait bien à notre utilisation. La gestion du XML se fait via QTXml.

Partie B : L'architecture - Extensibilité

L'architecture a été conçue afin de permettre un grande modulabilité et une extension des fonctionnalités.

Nous pensons que l'objectif est rempli puisque la création et l'implémentation de nouveaux opérateurs ne nécessite la modification que de deux fichiers :

→ operateurs.cpp
Ce fichier recense la liste des opérateurs qui sont testés dans le contrôleur.

```
bool estUnOperateurUnaire(const std::string& c)
{
    bool listOperateurUnaire =
            c=="NEG" ||
            c=="NUM" ||
            c=="DEN" ||
            c=="RE"  ||
            c=="IM"  ||
            c=="NOT" ||
            c=="EVAL" ||
            c=="FORGET";

    if(listOperateurUnaire)
        return true;

    return false;
```

```
}
```

L'ajout donc de l'opérateur LN demande de rajouter la ligne

```
        c=="LN" ||
```

Dans la fonction estUnOperateurUnaire

→ controleur .cpp
Dans la section

```
        if(operateurUnaire)
            {
              // ....
            }
```

Il suffit alors de rajouter

```
            if(operateurUnaire->toString() == "LN")
                    {
                        res = maFonctionNeg(l1) ;
                        this->empiler(res);

                    }
```