LO21 : Rapport de projet
Calculatrice à notation polonaise inversée

Agathe Oddon
Jean-Michel Tozzini

Printemps 2012

Introduction

Dans le cadre de notre UV LO21, nous devions réaliser la conception puis implémenter en C++ une calculatrice à notation polonaire inversée.

Le rendu final du projet est composé du présent rapport, du code du programme, de sa documentation Doxygen, ainsi que l'accès au système de versions Git.

Table des matières

Partie 1 Conception

1.1 Diagramme de classes

1.1.1 Types de données

Nous avons choisi de représenter les données manipulées par la calculatrice comme des objets de plusieurs classes : Entier, Réel, Rationnel, Complexe et Expression.

La classe complexe est composée de deux objets de type Base, classe abstraite de laquelle dérivent Entier, Réel et Rationnel. Cela permet d'obtenir des complexes composés de deux attributs de types différents.

Les classes Base, Complexe et Expression dérivent de la classe abstraite et exclusive Constante.

L'utilisation de la classe mère abstraite Constante nous permet à la fois d'empiler des objets de type pointeur sur Constante, et de définir des méthodes polymorphes pour les opérateurs.

1.1.2 Sauvegarde de l'historique

L'historique est représenté par les classes Gardiens, mementoAff et mementoStock. Sont utilisation est explicitée par la suite.

1.1.3 Piles

La pile est gérée par deux structures, la pile de stockage qui contient des Constantes et la pile d'affichage qui contient l'équivalent de la pile de stockage mais en QString.

1.1.4 Calculatrice

Cette classe est la classe principale de notre UML, elle gère l'interface graphique ainsi que l'affichage des objets. Elle ne manipule pas directement des données : cela se fait soit par le Gardien pour les mementos, soit par les piles pour les constantes et les chaines de caractères associées à ces constantes.

1.1.5 Schéma

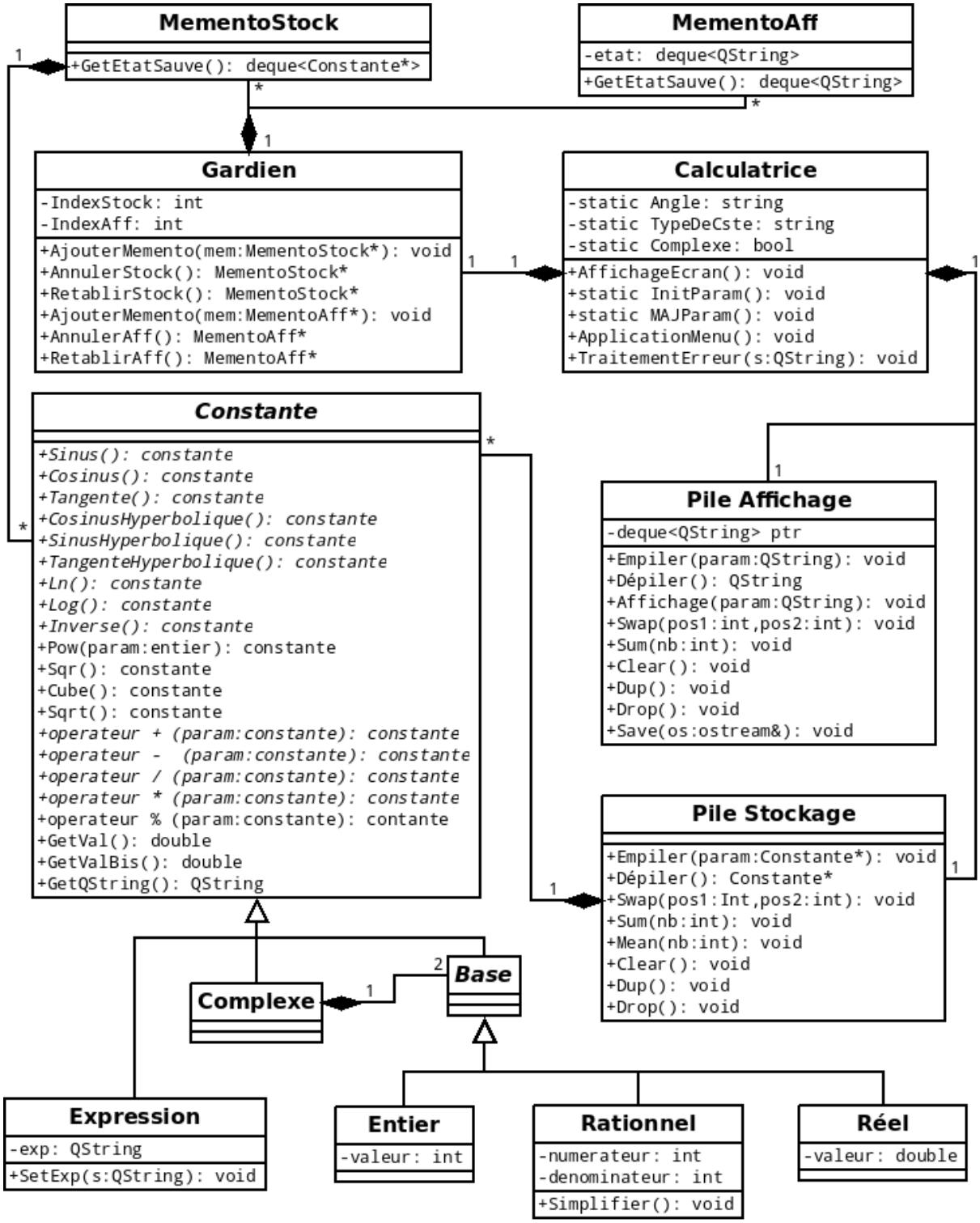

FIGURE 1.1 – Diagramme de classe de la Calculatrice

1.2 Diagrammes de séquences

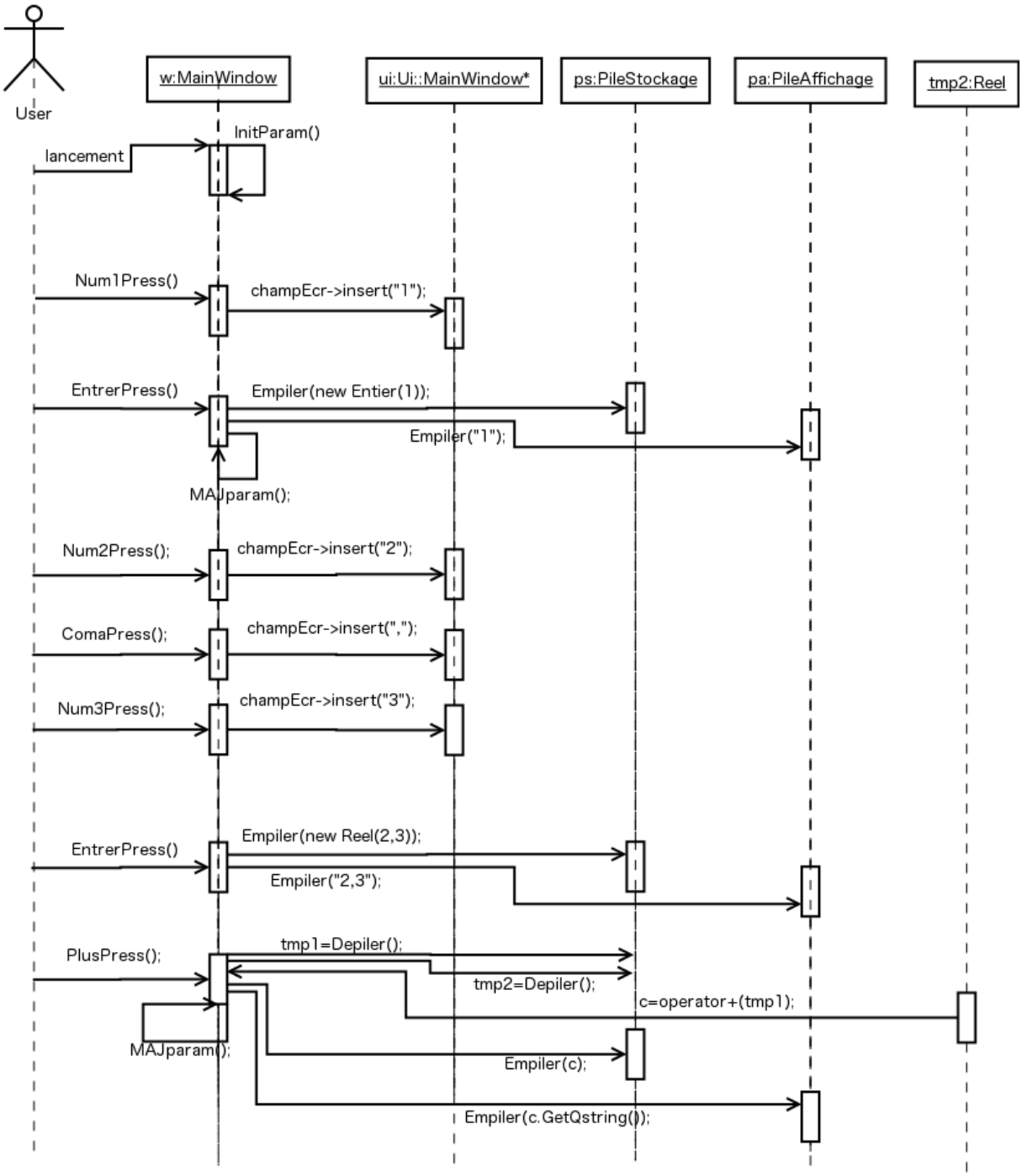

FIGURE 1.2 – Diagramme de séquence 1

Ce diagramme de séquence décrit le lancement de la fenêtre, la saisie de "1", "2,3" , l'appui sur le bouton "+" par l'utilisateur, puis l'addition des deux valeurs.

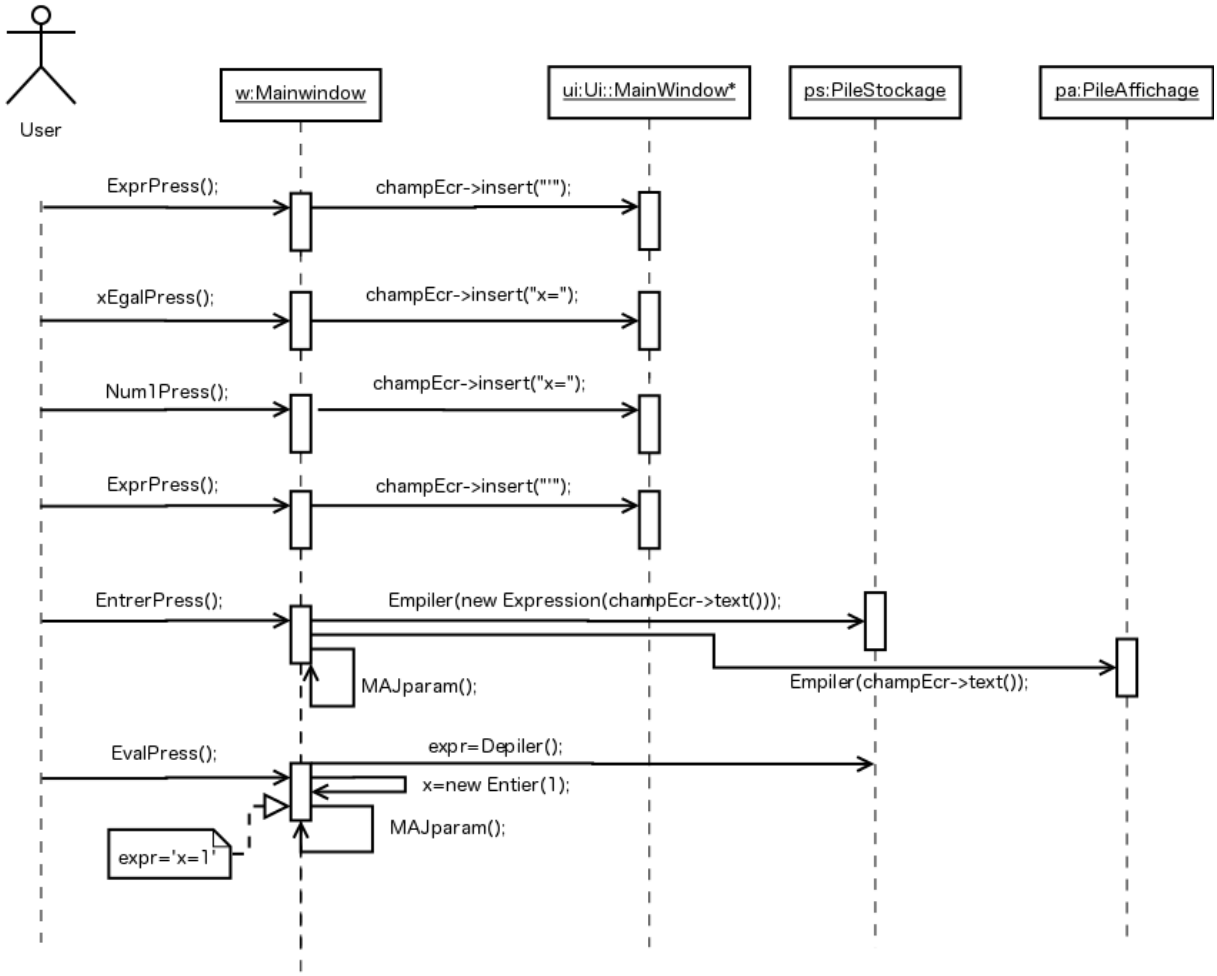

FIGURE 1.3 – Diagramme de séquence 2

Ce diagramme de séquence décrit le lancement de la fenêtre, la saisie de 'x=1', puis l'évaluation de l'expression et la sauvegarde de la valeur 1 dans la variable x.

Partie 2 Implémentation

2.1 Paramètres utilisateur

L'utilisateur dispose d'un menu "Paramètres" lui permettant :
- De changer le type de constante
- D'activer ou désactiver le clavier
- D'activer ou désactiver le mode complexe
- De choisir l'unité d'angle pour les fonctions cos, sin, tan, cosh, sinh et tanh

Le type de constante à été implementé pour une évolution future possible. Ainsi nous avons choisi de garder le type le plus riche en résultat d'une opération et donc de ne pas tenir compte de ce menu.

2.2 Variable utilisateur

L'initialisation de la variable utilisateur x se fait par l'intermédiaire d'une expression de la forme 'x=valeur'. Après évaluation (bouton "Eval"), cette expression initialise la variable x avec la valeur spécifiée, qui peut-être de type Entier, Rationnel, Réel ou Complexe. Ce comportement est décrit dans la figure 1.3.

La variable x sera ensuite accessible en tapant directement x ou par l'intermédiaire du bouton "x".

2.3 Sauvegarde et restauration de contexte

A l'ouverture du programme, une méthode est lancée permettant la restauration du contexte de la Calculatrice : il s'agit des piles, des valeurs des menu et de la valeur de la variable utilisateur si elle a été initialisée. Cette méthode interroge un fichier "param.txt" contenant les données du contexte. Si il s'agit de la première ouverture du programme, le fichier est initialisé avec des valeurs par défaut.

A chaque entrée de l'utilisateur ou modification des paramètres, une méthode est lancée pour mettre à jour le ficher.

2.4 Historique

Notre historique est implémenté avec le design pattern memento. Ainsi il est composé de trois classes, le Memento qui est une version de sauvegarde, le gardien qui est composé de plusieurs mementos et la pile est l'élément à sauvegarder.

Lors d'un changement de la pile, un memento est crée pour chacune des deux piles. Ce memento est une copie de la pile, Il est ensuite empilé dans le gardien. Les fonctions annuler et rétablir naviguent dans les mementos du gardien pour récupérer ces mémentos et utilise une méthode des pile pour lire et récupérer l'information contenu dans le memento.

2.5 Exceptions

Nous avons choisi d'implémenter qu'une seule classe Exception. Cette classe est composé d'une chaine de caratères et d'une méthode renvoyant celle-ci. Lors de l'execution d'un *throw* une instance d'erreur est créée et la chaine de caractère est envoyé par la méthode de la classe Exception. Cette chaine de caractère est ensuite prise en charge par une méthode de la calculatrice pour l'affichage dans le champ texte prévu à cet effet.

2.6 Méthodes de Constante et ses classes filles

Pour les opérateurs utilisables avec plusieurs types de données, par exemple l'opérateur + peut-être utilisés avec les types Entier, Réel, Rationnel, Complexe et Expression, nous avons d'abord défini une méthode de Constante permettant d'appeler la bonne méthode de ses classes filles.

Toutes les méthodes ont ensuite dues être implémentées pour tous les types de données. Nous avons fait le choix de toujours renvoyer le type le plus riche. Par exemple, l'addition d'un entier et d'un réel renverra toujours un réel, de même la soustraction d'un rationnel et d'un complexe renverra un complexe. [1] Les méthodes qui n'acceptait pas le type de données, par exemple l'opérateur mod (modulo) n'accepte que les entiers, renvoient des exceptions.

2.7 Piles

Nous avons fait le choix d'utiliser deux piles. Leur contenu est identique à la différence que la pile de stockage contient des Constantes alors que la pile d'affichage contient cette la valeur de cette constante sous forme de QString.

La pile d'affichage permet un affichage sur l'écran simplifié, ainsi il n'est pas nécessaire de parcourir la pile de stockage en entier et de transformer à chaque itération les constantes en QString. De plus, cette pile d'affichage sert à sauvegarder le contexte de manière plus aisée, en effet il est plus facile d'écrire un QString dans un fichier qu'une Constante.

Enfin, nous avons utilisé la STL pour créer nos piles. La structure deque a été utilisée, ainsi nous avons adapter cette structure à travers la pile pour une utilisation spécifique à nos problématiques.

2.8 Opérateur binaires

Pour la majorité de nos opérateurs binaires, nous avons fait le choix d'utiliser le design pattern Template Method. Ainsi nous traitons chaque opération binaire par la classe Constante qui délègue le calcule dans les surcharges des classes filles. Cela évite d'utiliser des *dynamic_cast* de manière excessive et permet de factoriser le code.

2.9 Types de données

Nous avons représenté nos types en nous inspirant du design pattern Composite, ainsi nous avons une classe abstraite Constante dont hérite tout les types de données. De plus les méthodes sont virtuelles pures et donc quelque soit la classe fille de Constante les méthodes sont identiques et s'appellent de la même façon.

1. Pour les opérations entre les types Rationnel et Réel, nous avons choisi de toujours renvoyer un Réel, méme si il ne s'agit pas du type le plus riche, du fait que nous ne pouvons pas convertir un Réel en Rationnel sans perdre d'information sur celui-ci.

2.10 Utilisation des Singletons

Pour permettre de respecter la constrainte d'unicité de que nous avons décrit dans l'UML, nous avons choisi d'implementer les classes Gardien, Pile de Stockage et Pile d'Affichage avec le design pattern singleton.

Partie 3 Améliorations et difficultés

3.1 Critiques de conception

Bien que la Pile d'Affichage possède des aspect pratique évoqué précédemment, elle contient également des contraintes. Ces dernières résident surtout dans l'obligation de créer un memento dédié et de faire fonctionner la classe Gardien de manière parallèle afin d'assurer la cohérence des données présentes dans les deux piles.

De plus nous n'avons pas utilisé le design pattern MVC, ainsi les contrôles et les affichages sont à la fois diséminées dans la structure et dans le cas de la Classe calculatrice ils sont fusionnés, ce qui engendre un problème de clarification stricte des rôles et des actions réalisées.

Enfin, nous n'avons pas réussi à éviter une surcharge massive des opérateurs binaires en prenant en compte chaque situations de calcul, cela a pour effet d'encombrer le code et engendre un manque d'efficacité dans l'implémentation.

3.2 Difficultés rencontrées

Nous avons vite pris en main Git pour éviter les problèmes de versions qui se sont posés dans les deux premières semaines.

Le projet aurait demander plus de temps pour une implémentation correcte. En effet, peu de design pattern ont été abordé en cours se qui demandait un gros effort de recherche personnel pour réaliser une projet implémenté de façon propre.

Conclusion

Ce projet nous aura permis de prendre conscience des difficultés et des problématiques lors de la conception et l'implémentation logiciel [1] et a commencer à nous familiariser avec celles-ci. Nous avons de plus découvert de nombreuses méthodes de programmation à travers les design pattern, enfin l'apprentissage d'un logiciel de gestion de versions à été très utile pour le travail collaboratif.

1. Qui est bien plus vaste que LO21