CooCoo

Projet LO21 Printemps 2012

Calculatrice Orienté Objet en notation polonaise inversée

LETELLIER Perrine | YASSINE TAREK

21/06/2012

Université de Technologie de Compiègne

Table des matières

INTRODUCTION

Dans le cadre de l'UV L021, nous avons été amenés à réaliser une calculatrice en notation polonaise inversée en C++. Ce document présente le résultat de ce projet et les différentes étapes de sa conception à son implémentation. Le développement se base sur le framework Qt, on obtient ainsi une interface claire et surtout une application multiplateforme.

Pour faciliter le travail à deux, nous avons crée un dépôt github et le logiciel de gestion de version associé Git. Le code ainsi que sa documentation et le présent rapport sont entièrement disponibles aux adresses :

https://github.com/arkzyna/CooCoo

ou

https://github.com/tyassine/CooCoo.

La documentation a été effectuée selon la norme doxygène.

Nous verrons dans ce document le fonctionnement de la calculatrice, le modèle UML des classes et enfin le diagramme de séquence des principaux scénarios.

FONCTIONNALITES

Types de constantes

CooCoo travaille sur des données de trois types : des expressions (implémentées par la classe ConstanteExp), des constantes simples (classe Constante) et des complexes (classe Complexe). Les constantes elles mêmes sont composées de trois types différents : les nombres entiers (classe Entier), les nombres réels (classe Reel), les nombre rationnels (classe Rationnel).

Ainsi les instances de la classe Entier sont composées d'une valeur de type int. Les instances de la classe Reel sont composées d'une valeur de type double. Les instances de la classe Rationnel sont composées de deux valeurs de type int, le numérateur et le dénominateur. Enfin les complexes sont composés de deux Constantes, la partie entière et la partie imaginaire, qui peuvent être soit des réels, soit des rationnels, soit des entiers (il est possible que la partie réelle et imaginaire ne soient pas du même type).

Les expressions sont des chaines de caractère qui sont ajoutées dans la pile de travail dans le but d'être évaluées plus tard. La pile de calcul stocke donc n'importe quel type de donnée.

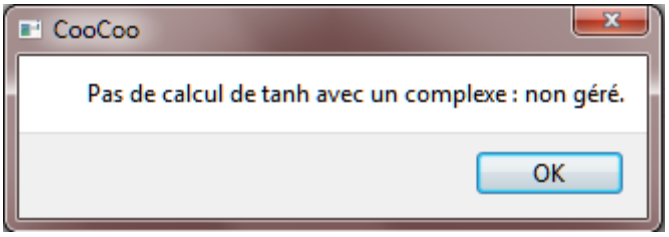
Utilisation et notation des constantes

Chaque donnée doit être notée de façon spécifique pour que son type soit reconnu par CooCoo. Une expression doit être entourée par des quotes, un entier est un nombre simple (seulement des chiffres et rien d'autre), un réel doit posséder une partie décimale et donc être inscrit avec le caractère ' . ', un rationnel doit posséder deux parties séparées par le caractère ' / ', un complexe doit posséder deux parties (de type réel, entier ou rationnel) séparées par le caractère ' $ '. Le contenu de l'expression n'est pas analysé est n'importe quelle expression, même si elle n'est pas en notation polonaise inversée, peut être empilée. Ceci ne pose pas de souci car la vérification sera effectuée lors de l'évaluation de cette expression et c'est alors qu'un message d'erreur de syntaxe pourra être affiché à l'utilisateur.

Nous avons fait le choix de laisser à l'utilisateur de CooCoo la possibilité d'entrer à la saisie des données de n'importe quel type. Mais à l'appel d'opération, il peut choisir le type retour, celui-ci peut être complexe ou non, et réel, rationnel ou entier. Si dans le menu l'utilisateur choisi complexe entier, alors les deux parties du nombre complexe résultat de l'opération seront entières.

Opérations

Certaines opérations ne sont pas compatibles avec certains types. Ainsi, par exemple les spécifications indiquaient que l'opérateur modulo ne devait être disponible que pour les entiers. De la même façon le logarithme népérien ne peut être calculé que sur une donnée non nulle. Lorsqu'une opération est impossible, CooCoo le signale à l'utilisateur avec une fenêtre pop-up indiquant la raison pour laquelle l'opération est impossible, et invite l'utilisateur à changer son entrée.



Quand une opération est demandé sur deux types de donnée différente, CooCoo effectue d'abord une conversion du type « inférieur » vers le type englobant et ceci afin de ne pas perdre d'information lors des calculs. L'opérateur est ensuite appliqué, et enfin si besoin le résultat est converti vers le type spécifié par l'utilisateur pour être ensuite empilé. Ainsi si l'on rentre ' 3 4$2 +', CooCoo traduit l'entier 3 en complexe 3$0 pour ensuite appliquer l'opérateur +.

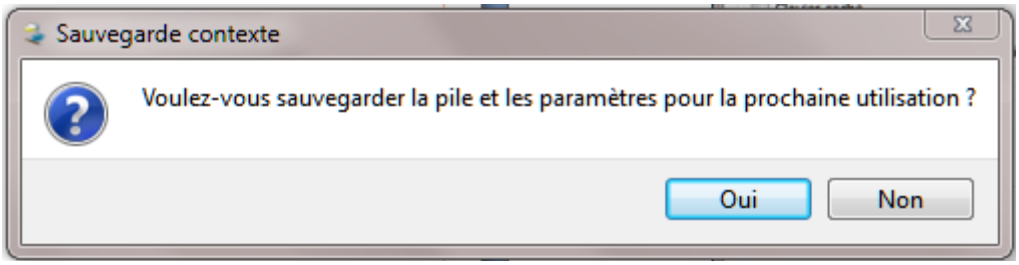
Fonction Annuler / Rétablir

Afin de répondre aux exigences du projet, CooCoo présente également les fonctions Annuler / Rétablir. Accessibles depuis le menu « Fichier », ou via les commandes Ctrl + Z et Ctrl + Y, elles permettent à l'utilisateur d'annuler ou de rétablir les actions qu'il a déjà réalisées sur la pile. Elles peuvent reconstituer n'importe quel état de la pile entre son lancement et l'état actuel. Nous avons également pris soin d'interdire momentanément le rétablissement d'état si l'on venait à faire une autre modification, comme dans tout autre logiciel.

Au niveau du code, nous avons implémenté ce système en nous calquant sur le design pattern « Memento », dont nous détaillerons ultérieurement l'utilisation.

Sauvegarde du contexte

A chaque demande de fermeture de l'application, CooCoo demande à l'utilisateur s'il souhaite enregistrer le contexte de travail actuel pour pouvoir à la prochaine ouverture, recommencer l'utilisation là où il était resté la fois précédente.

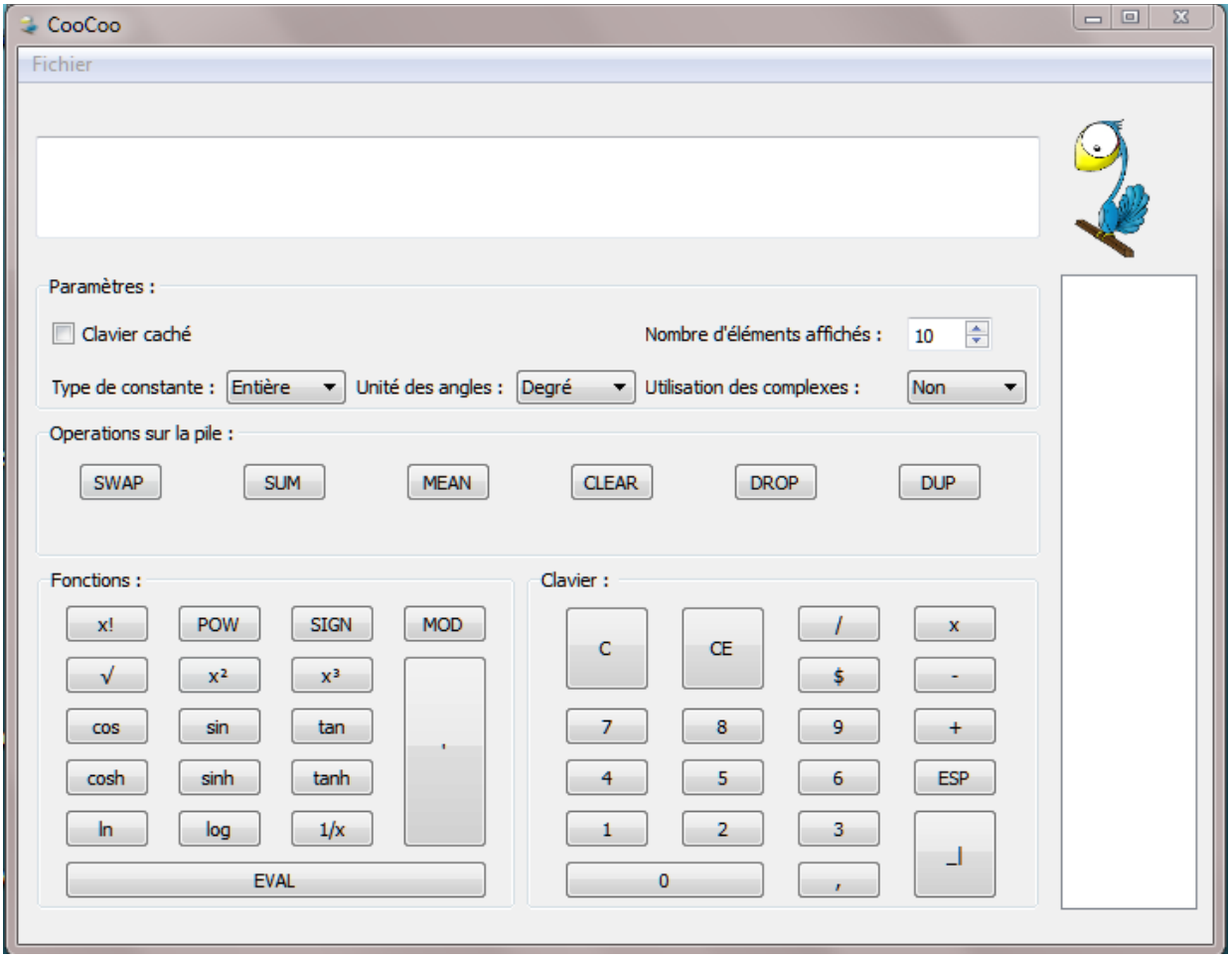


Si l'utilisateur indique son accord pour la sauvegarde, alors un fichier de sauvegarde est crée ou réécrit s'il existait déjà. Y sont sauvegardés tous les éléments contenus dans la partie « Paramètres » de l'interface, et ensuite tous les éléments de la pile.
A l'ouverture du programme, ce fichier est lu, les paramètres sont changés en conséquence et la pile est crée et initialisée avec le nombre d'éléments sauvegardés de la dernière utilisation.
Nous tenons à préciser que si fois que si l'utilisateur démarre le programme en chargeant un contexte sauvegardé préalablement, il ne lui sera pas possible d'utiliser les fonctions annuler/rétablir sur les actions effectuées antérieurement.
Cela vient du simple fait que l'objet Gardien référencé par notre Pile n'est pas sauvegardé dans ce ficher, mais ce n'est pas très problématique puisque le but de ces fonctions est d'annuler ou de rétablir des changements immédiats, et non lointains. La plupart des logiciels que nous connaissons ne permettent pas non plus d'annuler ou de rétablir des états enregistrés avant même le lancement du programme.

Système de log

CooCoo permet à l'utilisateur de garder une trace des actions effectuées sur l'application. Ces traces sont à la fois affichées en console pendant l'utilisation et imprimées dans un fichier « LogFichier ». A chaque action effectuée sur la calculatrice (et donc presque à chaque widget cliqué), de même qu'à chaque erreur récupérée sont attribués un degré d'importance et un message. L'inscription dans un fichier permet de garder une trace plus durable que celle dans la console.

INTERFACE



Affichage

La calculatrice possède deux afficheurs. Un QLineEdit pour les entrées de l'utilisateur. Avant que son contenu soit parsé puis entré dans la pile suite au click sur la touche commit marquée ' _| ', l'utilisateur à la possibilité de corriger son entrée grâce aux touches 'C' et 'CE'. L'affichage de la pile est effectué par un QListView, à chaque opération sur la pile, celui-ci est mis à jour par une fonction de refresh. Cette dernière fonction parcours toute la pile de travail, et affiche chaque élément en utilisant la méthode toQString que nous avons implémenté sur chaque type de donnée.

Paramètres

L'utilisateur peut choisir de cacher le clavier et de le faire réapparaître. Il peut choisir d'afficher plus ou moins d'éléments de la pile dans l'afficheur de droite, sans pour autant supprimer d'éléments de la vraie pile de travail. L'utilisateur peut aussi choisir le type de constante pour le retour des

fonctions et opérations et si celui-ci sera complexe ou non. Et enfin il peut choisir l'unité de retour des fonctions trigonométriques : degré ou en radian.

Clavier

Tous les boutons du clavier de l'interface peuvent être appelés par des raccourcis clavier. On peut ainsi masquer le clavier de l'interface pour ne se servir que du clavier de notre ordinateur. Comme demandé dans les spécifications, si la ligne d'édition ne contient rien et que la touche 'CE' est cliquée alors le dernier élément de la pile est supprimé. Les boutons de numéros et d'opérations peuvent être appelés par leurs équivalents sur le clavier. Backspace est l'équivalent du clic sur 'CE' et C l'équivalent du clic sur 'C'.

Opérations sur la pile

Les opérations MEAN, SUM et SWAP entrainent le dépilement de d'éléments de la pile, elles ne prennent en paramètre que des entiers, si ce n'est pas le cas des données dépilées, celle-ci sont converties en entier. MEAN et SUM renvoient un résultat dans le type de donnée le plus englobant et ne prend pas compte des paramètres de constante choisis par l'utilisateur.

Fonctions

Certaines fonctions, comme dit précédemment, ne sont pas disponibles pour certains types de constantes (voir les spécifications[1]). Le type le plus restrictif est celui des complexes. Ainsi lorsque dans les paramètres les complexes sont choisis, de nombreux boutons de fonctions sont grisés afin de montrer à l'utilisateur qu'ils ne peuvent être utilisés. Si malgré tout l'utilisateur les rentres dans la ligne d'édition en ayant choisi un type non complexe et que juste avant d'appuyer sur le bouton commit pour lancer le parser sur la ligne il choisit les complexes, une erreur s'affichera.

[1] https://www.hds.utc.fr/~ajouglet/LO21/co/Spec.html

MODELISATION

Diagramme de séquence

Le diagramme de séquence si dessous explicite le fonctionnement de la calculette suite à un commit, lorsque l'utilisateur appuie sur le bouton ' _|' du clavier de l'interface ou 'Entrer' du clavier de l'ordinateur. L'exemple utilisé est celui pour lequel l'utilisateur à entré dans la ligne d'édition « 5.0 3$5 + » en ayant choisi dans les paramètres d'utiliser les rationnels en type de constante de retour et pas de complexes.

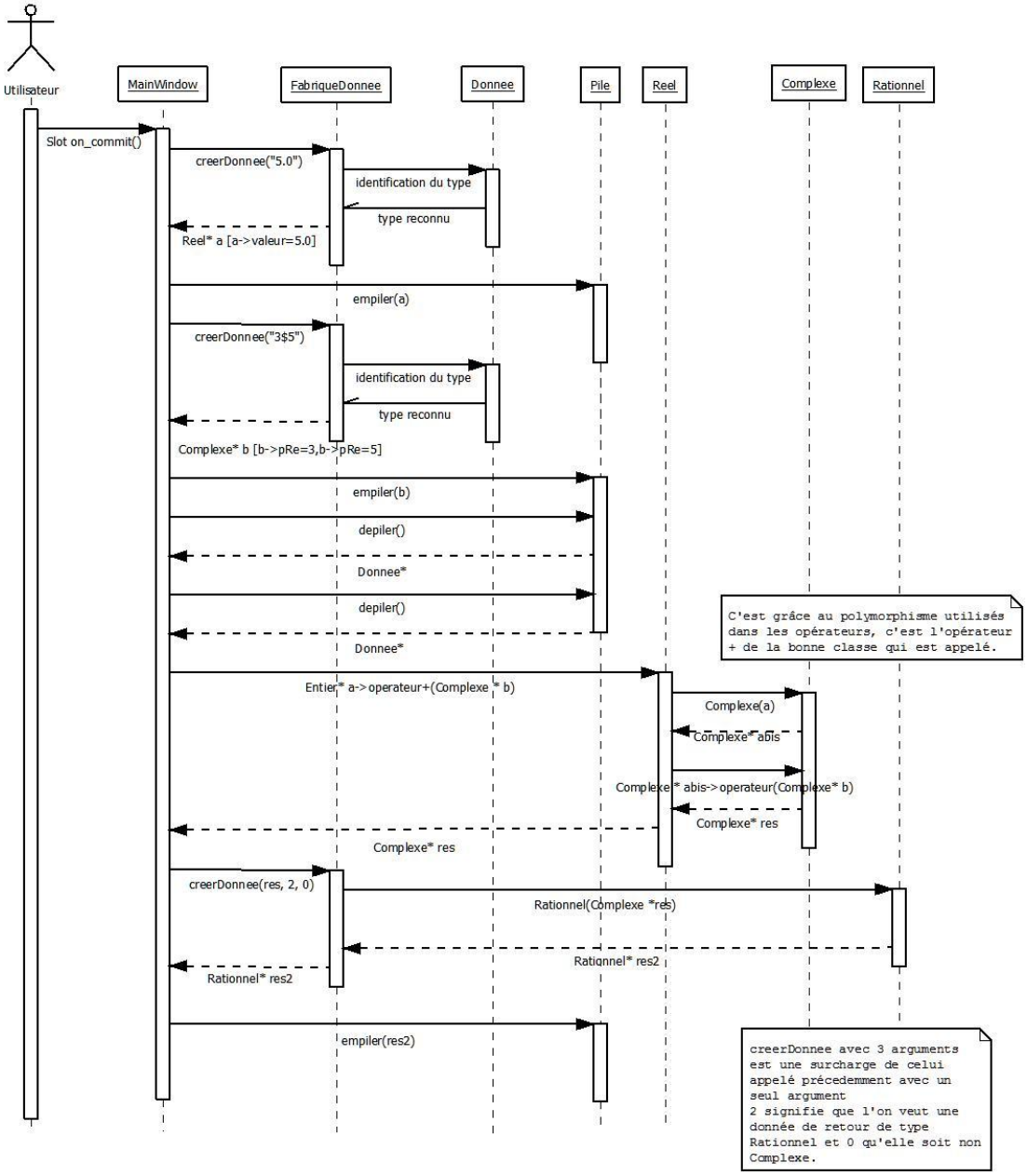


Diagramme de classes

Afin de maximiser l'efficacité de notre programme final, nous avons pensé dès la modélisation à nous calquer sur quelques design patterns.

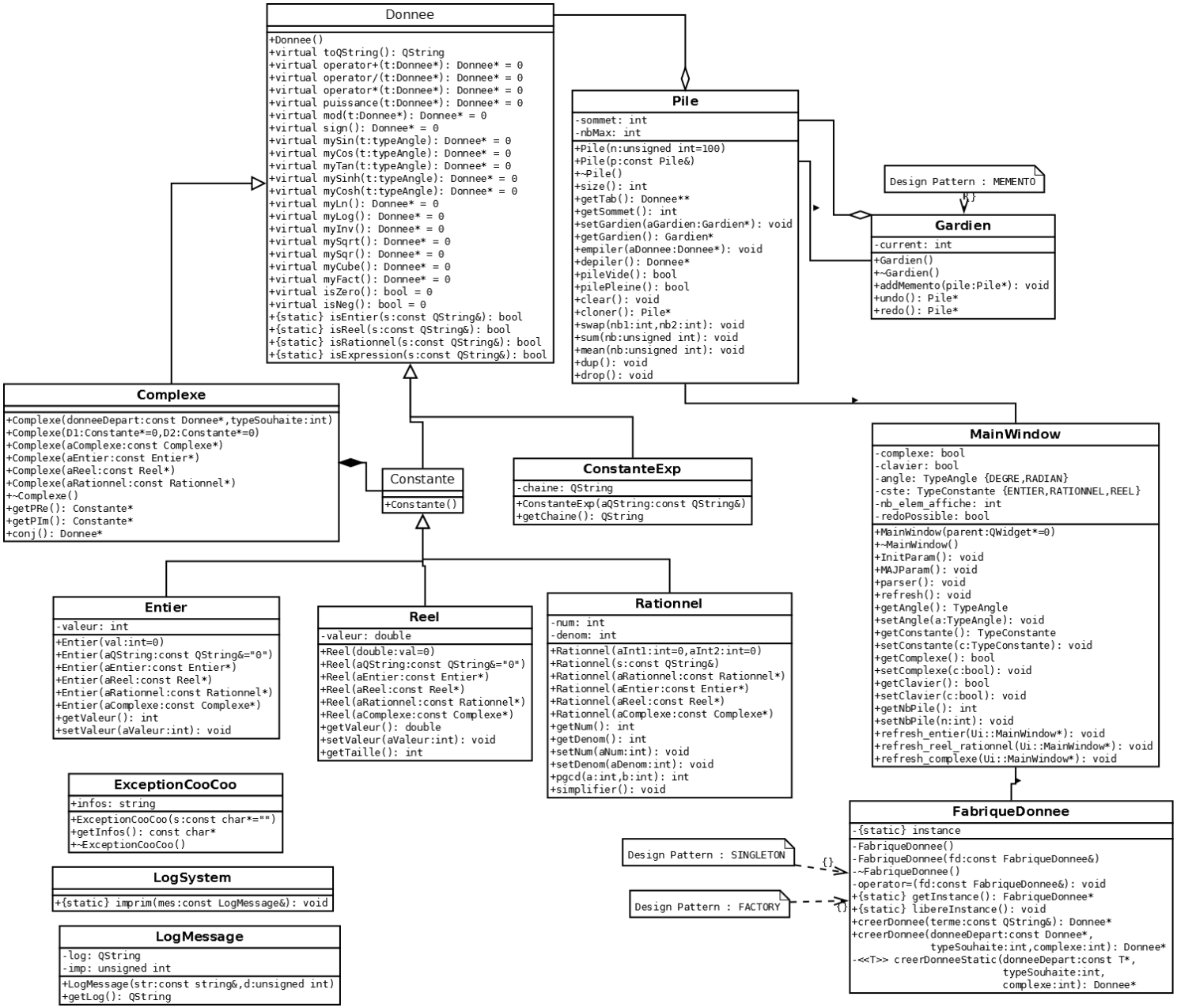
Voyant que la calculatrice aurait très souvent besoin d'instancier de nouveaux objets représentant n'importe quel type héritant de Donnee, et cela à partir de chaînes de caractères (lors du parsing) ou d'autres objets (pendant les opérations), nous avons décidé d'implémenter le design pattern Factory.

Ainsi, notre classe FabriqueDonnee comporte une méthode qui détecte automatiquement le type d'une donnée en lisant la chaîne qui le représente, et l'instancie alors. Une autre de ses méthodes permet de prendre un objet en entrée et créer un autre objet selon ce que décidera n'importe quelle fonction appelante, notamment en précisant le type de l'objet, et en indiquant s'il sera complexe ou non. Dans les deux cas, le bon constructeur du type de sortie souhaité est appelé : cette classe sert de couche d'abstraction et permet d'instancier des objets sans avoir à se soucier des opérations internes.

Comme tout au long du programme nous n'avions besoin que d'une unique instance de cette classe FabriqueDonnee, nous en avons profité pour lui appliquer le design pattern Singleton. Ainsi, en utilisant les techniques abordées en cours, en TP et en TD, nous sommes parvenus à faire en sorte qu'il soit impossible d'avoir, à tout moment du programme, plus d'une instance de FabriqueDonnee. De plus, le seul moyen d'accéder à celle-ci est d'utiliser une fonction bien précise, qui renvoie un simple pointeur vers l'instance.

Lorsqu'il a fallu implémenter le système d'annulation et de rétablissement des modifications, nous nous sommes intéressés à un design pattern que nous n'avions pas vu en cours : Memento. Celui-ci s'est avéré très pratique, puisqu'il a été conçu justement pour ce cas d'utilisation. Il repose en de nouvelles classes qui enregistrent et conservent en mémoire les différents états d'un objet au cours du temps. Ces classes emploient également des méthodes qui permettent de naviguer entre ces états et de restaurer celui que l'on cherche.

Nous avons utilisé une version simplifiée de ce design pattern, qui a tout a fait répondu à nos attentes. Dans le fichier « memento.h », nous avons créé une classe « Gardien » permettant de gérer un vecteur de pointeurs de pile. Ainsi, dans chaque case de ce vecteur, on peut enregistrer l'état de la pile principale à un instant donné, grâce à la fonction addMemento, qui est appelé après chaque modification de la pile. Les méthodes undo et redo ne font que naviguer dans ce vecteur pour renvoyer l'état recherché. Seul un nouvel attribut a été rajouté à la classe Pile : un pointeur vers un objet Gardien.


Donnee
+Donnee()
+virtual toQString(): QString
+virtual operator+(t:Donnee*): Donnee* = 0
+virtual operator/(t:Donnee*): Donnee* = 0
+virtual operator*(t:Donnee*): Donnee* = 0
+virtual puissance(t:Donnee*): Donnee* = 0
+virtual mod(t:Donnee*): Donnee* = 0
+virtual sign(): Donnee* = 0
+virtual mySin(t:typeAngle): Donnee* = 0
+virtual myCos(t:typeAngle): Donnee* = 0
+virtual myTan(t:typeAngle): Donnee* = 0
+virtual mySinh(t:typeAngle): Donnee* = 0
+virtual myCosh(t:typeAngle): Donnee* = 0
+virtual myLn(): Donnee* = 0
+virtual myLog(): Donnee* = 0
+virtual myInv(): Donnee* = 0
+virtual mySqrt(): Donnee* = 0
+virtual mySqr(): Donnee* = 0
+virtual myCube(): Donnee* = 0
+virtual myFact(): Donnee* = 0
+virtual isZero(): bool = 0
+virtual isNeg(): bool = 0
+{static} isEntier(s:const QString&): bool
+{static} isReel(s:const QString&): bool
+{static} isRationnel(s:const QString&): bool
+{static} isExpression(s:const QString&): bool
Pile
-sommet: int
-nbMax: int
+Pile(n:unsigned int=100)
+Pile(p:const Pile&)
+~Pile()
+size(): int
+getTab(): Donnee**
+getSommet(): int
+setGardien(aGardien:Gardien*): void
+getGardien(): Gardien*
+empiler(aDonnee:Donnee*): void
+depiler(): Donnee*
+pileVide(): bool
+pilePleine(): bool
+clear(): void
+cloner(): Pile*
+swap(nb1:int,nb2:int): void
+sum(nb:unsigned int): void
+mean(nb:unsigned int): void
+dup(): void
+drop(): void
Design Pattern : MEMENTO
Gardien
-current: int
+Gardien()
+~Gardien()
+addMemento(pile:Pile*): void
+undo(): Pile*
+redo(): Pile*
Complexe
+Complexe(donneeDepart:const Donnee*,typeSouhaite:int)
+Complexe(D1:Constante*=0,D2:Constante*=0)
+Complexe(aComplexe:const Complexe*)
+Complexe(aEntier:const Entier*)
+Complexe(aReel:const Reel*)
+Complexe(aRationnel:const Rationnel*)
+~Complexe()
+getPRe(): Constante*
+getPIm(): Constante*
+conj(): Donnee*
Constante
+Constante()
ConstanteExp
-chaine: QString
+ConstanteExp(aQString:const QString&)
+getChaine(): QString
MainWindow
-complexe: bool
-clavier: bool
-angle: TypeAngle {DEGRE,RADIAN}
-cste: TypeConstante {ENTIER,RATIONNEL,REEL}
-nb_elem_affiche: int
-redoPossible: bool
+MainWindow(parent:QWidget*=0)
+~MainWindow()
+InitParam(): void
+MAJParam(): void
+parser(): void
+refresh(): void
+getAngle(): TypeAngle
+setAngle(a:TypeAngle): void
+getConstante(): TypeConstante
+setConstante(c:TypeConstante): void
+getComplexe(): bool
+setComplexe(c:bool): void
+getClavier(): bool
+setClavier(c:bool): void
+getNbPile(): int
+setNbPile(n:int): void
+refresh_entier(Ui::MainWindow*): void
+refresh_reel_rationnel(Ui::MainWindow*): void
+refresh_complexe(Ui::MainWindow*): void
Entier
-valeur: int
+Entier(val:int=0)
+Entier(aQString:const QString&="0")
+Entier(aEntier:const Entier*)
+Entier(aReel:const Reel*)
+Entier(aRationnel:const Rationnel*)
+Entier(aComplexe:const Complexe*)
+getValeur(): int
+setValeur(aValeur:int): void
Reel
-valeur: double
+Reel(double:val=0)
+Reel(aQString:const QString&="0")
+Reel(aEntier:const Entier*)
+Reel(aReel:const Reel*)
+Reel(aRationnel:const Rationnel*)
+Reel(aComplexe:const Complexe*)
+getValeur(): double
+setValeur(aValeur:int): void
+getTaille(): int
Rationnel
-num: int
-denom: int
+Rationnel(aInt1:int=0,aInt2:int=0)
+Rationnel(s:const QString&)
+Rationnel(aRationnel:const Rationnel*)
+Rationnel(aEntier:const Entier*)
+Rationnel(aReel:const Reel*)
+Rationnel(aComplexe:const Complexe*)
+getNum(): int
+getDenom(): int
+setNum(aNum:int): void
+setDenom(aDenom:int): void
+pgcd(a:int,b:int): int
+simplifier(): void
ExceptionCooCoo
+infos: string
+ExceptionCooCoo(s:const char*="")
+getInfos(): const char*
+~ExceptionCooCoo()
LogSystem
+{static} imprim(mes:const LogMessage&): void
LogMessage
-log: QString
-imp: unsigned int
+LogMessage(str:const string&,d:unsigned int)
+getLog(): QString
Design Pattern : SINGLETON
Design Pattern : FACTORY
FabriqueDonnee
-{static} instance
-FabriqueDonnee()
-FabriqueDonnee(fd:const FabriqueDonnee&)
-~FabriqueDonnee()
-operator=(fd:const FabriqueDonnee&): void
+{static} getInstance(): FabriqueDonnee*
+{static} libereInstance(): void
+creerDonnee(terme:const QString&): Donnee*
+creerDonnee(donneeDepart:const Donnee*,
typeSouhaite:int,complexe:int): Donnee*
-<<T>> creerDonneeStatic(donneeDepart:const T*,
typeSouhaite:int,
complexe:int): Donnee*

CONCLUSION

Ce projet a d'abord été pour nous une occasion d'appliquer toutes les connaissances et compétences que nous avions acquises, tout au long du semestre, dans l'UV LO21. En effet, nous avons mené ce projet intégralement : de la modélisation jusqu'à la réalisation. Nous avons constamment utilisé des outils propres à chaque étape, notamment les diagrammes UML, les design patterns, et bien évidemment tout le langage de programmation C++.

Cela nous a donc permis d'aller au delà de l'utilisation individuelle d'un chapitre du cours, en conjuguant véritablement tous les aspects de la programmation orientée objet.

Enfin, ce projet nous a également permis de découvrir de nouveaux outils et de nous familiariser à leur utilisation. Nous pensons bien sûr au framework Qt, mais également à Git, gestionnaire de version qui nous a rendu un grand service. Cela s'est avéré parfois difficile de « partir de zéro », mais nous ne regrettons pas le temps passé à apprendre ces nouvelles compétences, qui nous seront certainement utiles plus tard.