LUCHIANCENCO Tudor

NJO EWELE Kevin

LO21 : Rapport

Calculatrice à notation polonaise inversée

Printemps 2012

Introduction

Durant la seconde moitié du semestre, il nous a été demandé de développer une calculatrice à notation polonaise inversée. Ce type de calculatrice a pour principe la notation post-fixée. Nous avons pour habitude d'utiliser des calculatrices à notation infixée.

Version infixée : 2 + 3 égale à 5.

Version post-fixée : 2 3 + toujours égale à 5

La calculatrice gère plusieurs types de **constantes** :

- des nombres
- des expressions

Une **expression** est une chaîne de caractères encapsulant des nombres et/ou des opérateurs.

Quant aux **nombres**, ils peuvent être soit des nombres complexes (partie réelle et partie imaginaire), soit des nombres non-complexes (entiers, réels, ou rationnels).

La calculatrice gère une **pile (de constantes)** qui affiche les constantes empilées et les résultats de calculs. Elle possède également une **« pile d'affichage »** appelée historique qui sert à afficher toute commande entrée dans la ligne de commande par l'utilisateur.

La calculatrice gère la **sauvegarde de contexte**, c'est à dire qu'à sa fermeture, l'état de la pile ainsi que ses paramètres sont sauvegardés sous la forme d'un fichier binaire (illisible), afin qu'à son réouverture on retrouve l'environnement de travail initial.

La calculatrice gère les fonctions **« annuler » et « rétablir »** de la pile, c'est à dire que l'utilisateur peut retrouver à n'importe quel moment l'état de pile avant le dernier empilement de constante et inversement.

Pour finir, la calculatrice gère également un **système de log**. Elle enregistre dans un fichier de log lisible cette fois-ci, tous les messages de log qu'elle a généré au cours de son exécution (des messages d'erreurs, des messages d'informations, etc.). Chacun des messages de log possède une date ainsi qu'un niveau d'importance.

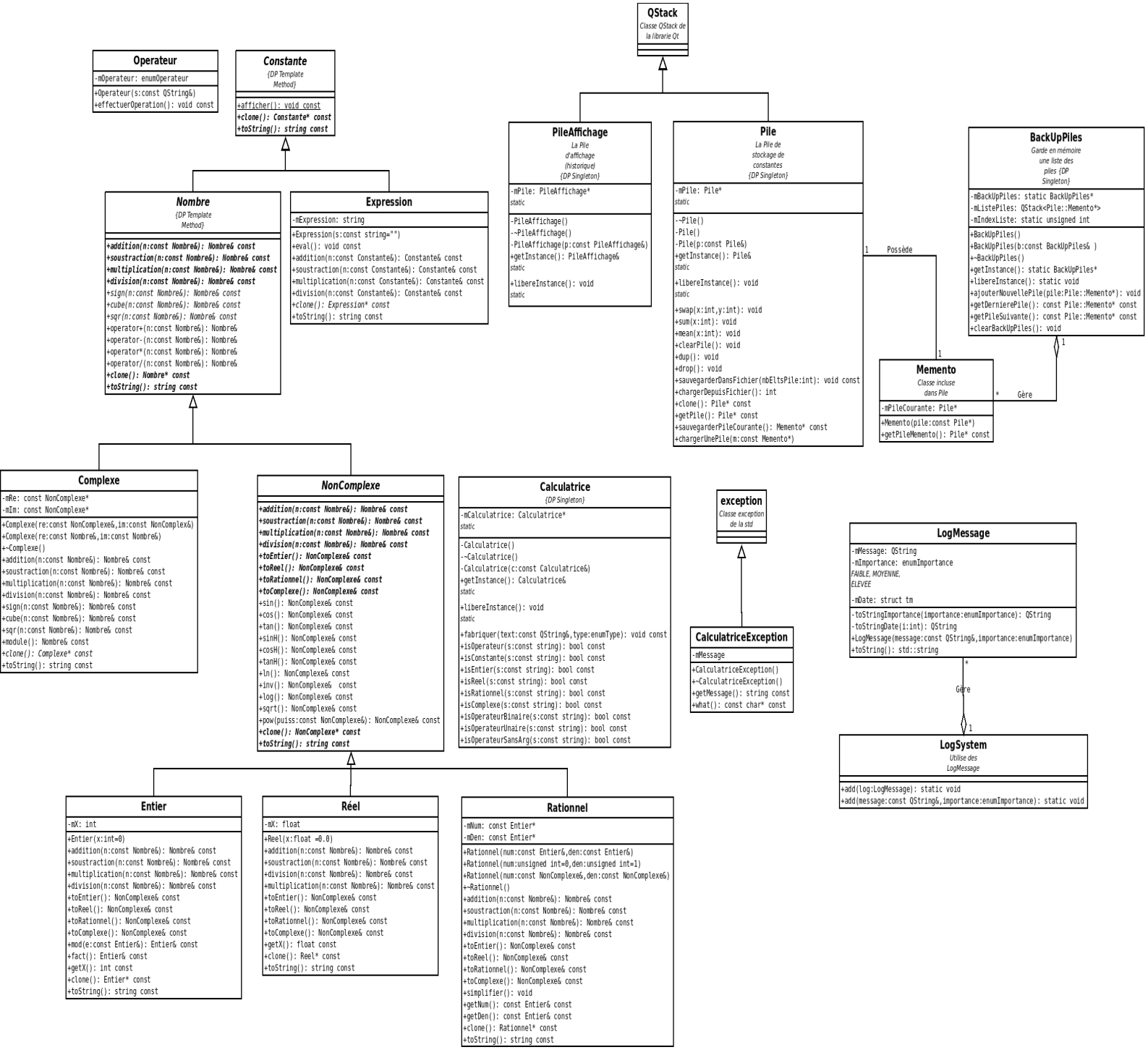
Dans ce rapport, nous détaillerons dans un premier temps les formalismes UML nous ayant permis de réaliser notre calculatrice. Cela nous donnera l'occasion de justifier nos choix de conception. Pour finir, on donnera les diagrammes de séquences caractérisant les actions principales de la calculatrice.

I – Modélisation

Afin de mener à bien notre projet, il était important de se fixer une ligne directrice, qui a été éventuellement retravaillée en cours de projet.

Pour ce faire, une phase de formalisation précédent l'étape du développement à proprement parler était essentielle. C'est grâce au langage UML que nous avons pu clarifier nos idées.

Ci-dessous, voici le diagramme UML final relatif à notre projet.



A – Traitement des données

Comme précisé dans l'énoncé, les données peuvent être de plusieurs types. Nous avons décidé que qu'elle que soit le type de la constante, n'importe laquelle doit au moins pouvoir afficher les informations qu'elle encapsule et pouvoir se cloner. D'où la déclaration des méthodes virtuelles pures ***toString()*** et ***clone()*** présentes dans la classe *Constante.*
Ces méthodes sont déclarées dans chacune des classes représentant des types de données et sont implémentées dans les classes non abstraites à savoir, *Complexe, Entier, Réel, Rationnel et Expression*. Ceci fait de la classe Constante, une classe abstraite, comme les classes *Nombre et NonComplexe* le sont également.

Tout d'abord on récupère une chaîne de caractères à partir de la ligne de commande sur laquelle on applique la méthode *split()* pour enlever les espaces. Ensuite, nous parcourons chaque élément grâce à un itérateur. Nous testons le type de la donnée, et nous demanderons à la classe *Calculatrice* de nous fabriquer la constante respective. Si il s'agit d'un opérateur, l'opération respective va être appliquée sur la pile, sinon il s'agit bien sur d'une constante qui sera donc empilée. Les schémas de séquences détaillerons bien plus précisément ces étapes.

Comme spécifié sur le diagramme ci-dessus, la classe *Nombre* fait office d'interface puisqu'elle factorise les méthodes devant être implémentées par les classes qui en héritent. Ces méthodes sont donc communes au nombres complexes et aux nombres non complexes :

```
virtual Nombre& addition(const Nombre& n) const = 0 ;

virtual Nombre& soustraction(const Nombre& n) const = 0 ;

virtual Nombre& multiplication(const Nombre& n) const = 0 ;

virtual Nombre& division(const Nombre& n) const = 0 ;

virtual Nombre* clone() const = 0 ;

virtual QString toString() const = 0 ;
```

Le Design Pattern Singleton a été utilisé pour les deux piles (classes *Pile* et *PileAffichage*) ainsi que pour la classe *Calculatrice*.

Le Design Pattern Template Method a été utilisé deux fois (classes *Constante* et *Nombre*).

Le Design Pattern Memento a été utilisé dans la classe *Pile* afin de réaliser les fonctions « annuler » et « rétablir » de la pile.

B – Gestion de la pile

Le fonctionnement de la calculatrice que nous avons développée nécessite l'existence d'une pile permettant le traitement des informations. Ce système de pile est à plus d'un égard important pour une calculatrice à notation post-fixée.

Voici le principe d'implémentation de note classe *Pile* :

La pile réalisée hérite de la méta-classe **QStack**. On spécifie le type d'objet que notre pile contiendra grâce à **< Constante *>**. Elle contiendra donc des pointeurs vers des objets de type *Constante*.

```
class Pile: public Qstack<Constante*> { … }
```

Pour la gestion des fonctions « annuler » et « rétablir » de la pile, comme nous l'avons dit, nous avons utilisé Le Design Pattern Memento. Il s'agit de la classe *Memento* que notre classe *Pile* contient. Celui-ci stocke toujours un pointeur vers un clone de la pile courante.

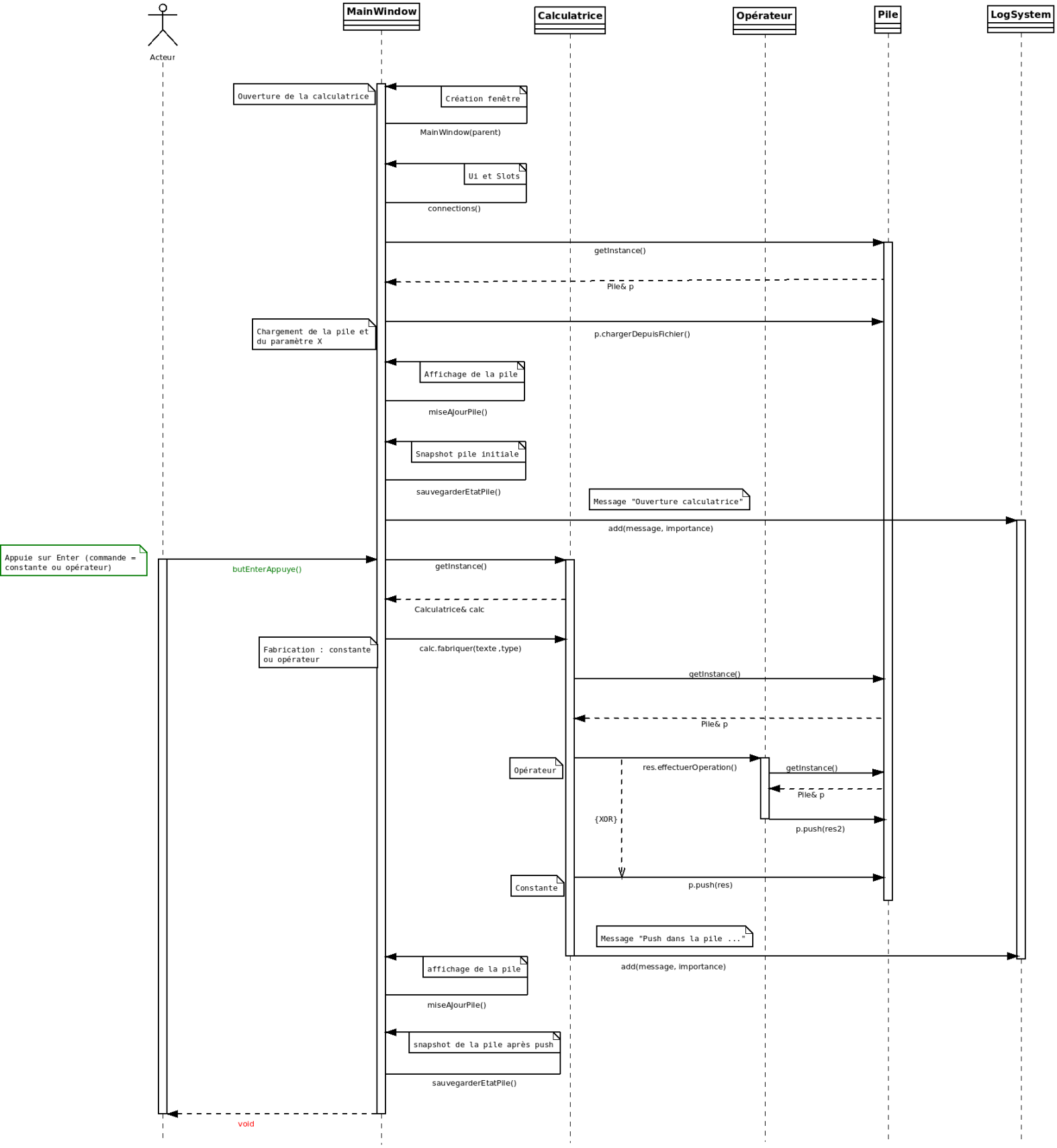
A l'ouverture de la calculatrice on charge la pile et on fait un « snapshot » de cette pile grâce à la fonction ***sauvergarderPile()*** qui utilise la classe *BackUpPiles.* Également, après l'empilement d'une nouvelle constante, la fonction ***sauvergarderPile()*** est appelée.

Une autre classe *BackUpPiles* permet de stocker une liste de pointeurs vers des objets de type *Memento,* donc en d'autres termes, la classe *BackUpPiles* garde la trace de toutes les modifications de la pile, et c'est donc grâce à cette classe et notamment grâce aux fonctions : **getDernierePile()** et **getPileSuivante()** que nous pouvons revenir et avancer dans les modifications de notre pile.

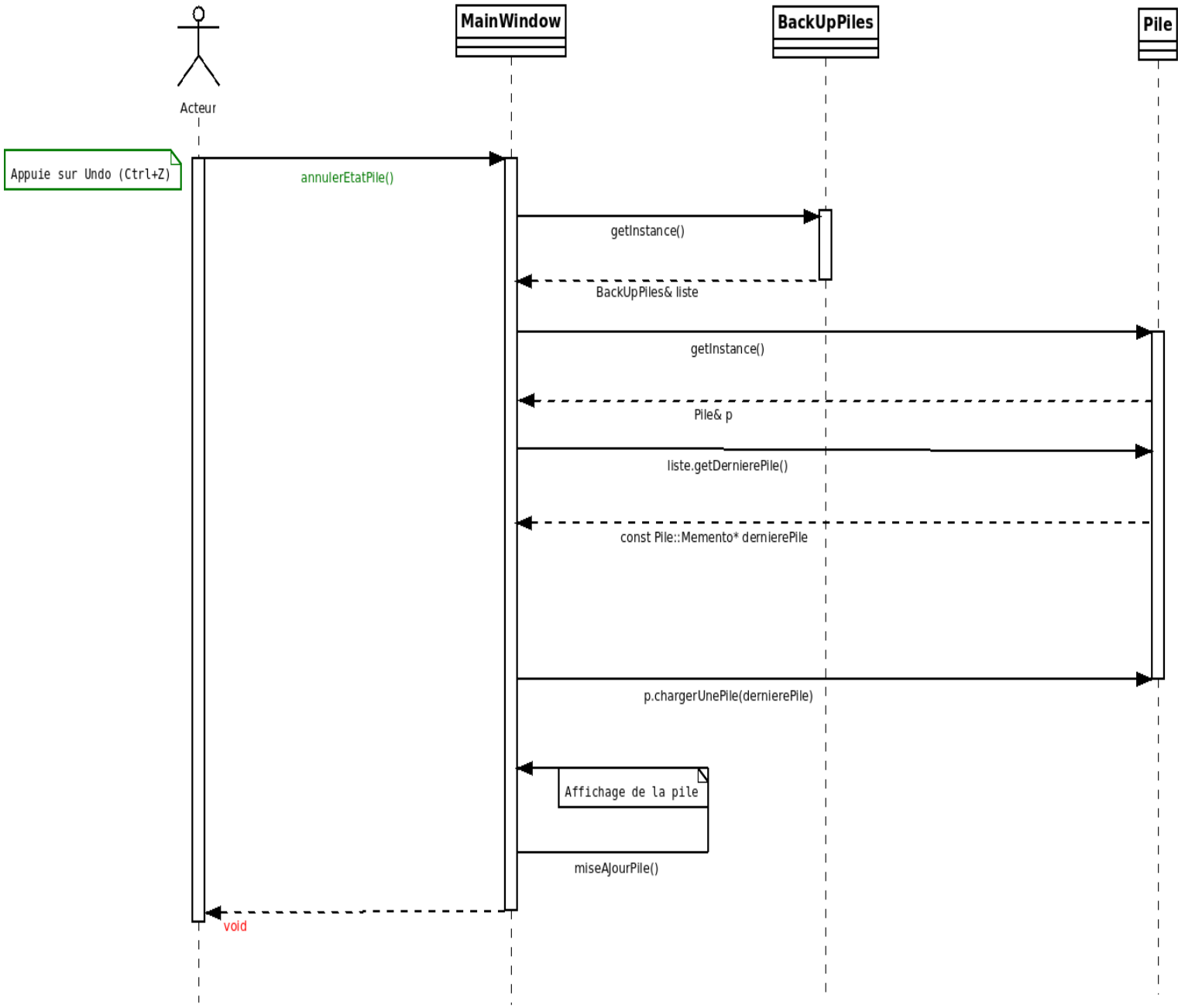
Voici les diagrammes de séquences qui détaillent toutes ces opérations principales :

II – Diagrammes de séquence

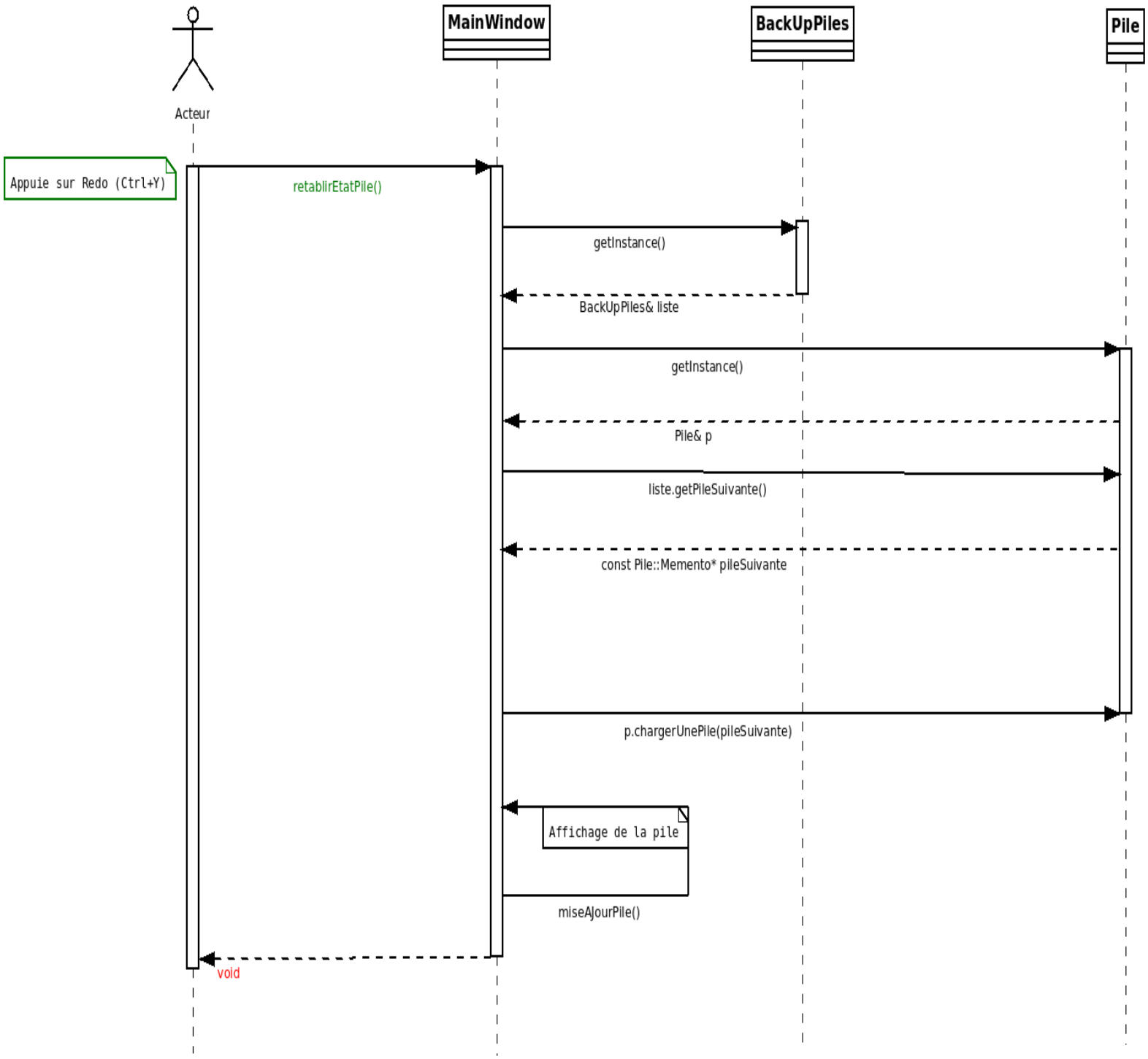
Ouverture calculatrice et exécution Enter



Undo



Redo


Acteur
MainWindow
BackUpPiles
Pile
Appuie sur Redo (Ctrl+Y)
retablirEtatPile()
getInstance()
BackUpPiles& liste
getInstance()
Pile& p
liste.getPileSuivante()
const Pile::Memento* pileSuivante
p.chargerUnePile(pileSuivante)
Affichage de la pile
miseAJourPile()
void

Conclusion

Ce projet a été pour nous l'occasion de mettre en pratique toutes les connaissances emmagasinées au cours du semestre. Il nous a semblé être un projet très complet dans la mesure où il a fait intervenir nombre de concepts de programmation C++ étudiés durant les TD comme les Design Pattern, les classes abstraites, etc.

Ce qui nous a semblé très intéressant c'était surtout toute la phase de conception du projet où il a fallu chercher des solutions déjà existantes (les Design Pattern, la bibliothèque Qt, etc.). La documentation Doxygène nous a également fait une petite introduction au monde qu'est le formalisme projet.

Pour résumer, ce projet nous a permis de consolider nos connaissances sur le langage C++ mais aussi sur les techniques de modélisation de manière plus profonde que les cours de TD habituels pouvaient le faire.