Dufrot Florian 2012

Fradcourt Sébastien

Rapport de fin de projet

Intitulé du projet : Calculatrice à notation polonaise inversée

## Introduction

Au cours de l’UV de LO21, nous avons dû programmer une calculatrice à notation polonaise inversée. En premier temps il nous a fallu réfléchir à l’organisation des classes et ainsi que des différentes fonctions qui étaient utiles au bon fonctionnement, et ainsi établir l’UML de ce projet. (Celui-ci se trouvant en document joint) L’option permettant de compléter le projet qui a été sélectionné est la gestion des logs systèmes.

Cet UML terminé, nous avons commencé le développement.

Durant ce projet, nous avons fait face à plusieurs difficultés, notamment concernant l’élaboration de l’UML, ce n’était pas facile de tout déterminé au début puisque certains éléments se trouvaient être correct lors de l’élaboration mais lors du développement nous nous sommes rendus compte que cela était inadapté et donc il nous a fallu revoir quelques classes.

De plus concernant ce projet, nous devions le gérer grâce à un outil de gestion de versions de notre choix, nous avions choisi d’utiliser GIT (le lien vers le dépôt en mode public : …) qui d’ailleurs nous a posé quelques petits problèmes en rapport avec sa configuration. De plus l’EDI utilisé (QT) est celui utilisé en cours et également cet EDI est tout à fait adapté à ce genre de projet.  
Afin d’obtenir une documentation complète de ce projet, nous avons généré une documentation avec le logiciel Doxygen.

## Les choix de conception Les designs patterns :

#### Le singleton

Nous l’avons utilisé dans plusieurs de nos classes, en effet le singleton était nécessaire afin de s’assurer qu’il n’y avait qu’une unique instance de cette classe.

Pour exemple : dans les classes LogSystem et Context, étant donné que ces classes manipulaient des fichiers (logSystem.text et contexte.xml), il ne fallait pas que plusieurs instances manipulent le fichier en même temps sinon on aurait eu des problèmes de partage de fichier.

Mais également, le singleton nous a permis de pouvoir accéder à l’instance dans n’importe quelle autre classe. Par exemple concernant la classe Settings, nous devions pouvoir accéder à ces attributs depuis n’importe quelle classe afin de pouvoir gérer les bonnes conversions et donc le singleton était nécessaire.

#### Le factory

Nous l’avons utilisé dans le cadre de la gestion des constantes, en effet dans la classe ConstanteFactory, on lui passe une chaine de caractères en paramètres et lui il cherche la constante qui correspond le mieux à la chaine (à l’aide des Regex) et renvoie un objet de type Constante.

#### Le command

Nous l’avons utilisé dans le cadre de la gestion de l’historique, en effet dans la class Pile, on y trouve un attribut de type QStack contenant des objets de type Historique (qui contient les deux fonctions : undo et redo) et après étant donné que plusieurs classes héritent de Historique, nous avons pu les intégrer à cette pile d’historique tout ceci ayant pour but que chaque objet héritant de Historique ait sa propre définition de undo et redo et donc des comportements différents.

#### Le strategy

Nous l’avons utilisé dans le cadre des opérateurs, en effet la fonction call() contenue dans la classe Operateur permet à ce que lorsqu’on a un opérateur et plusieurs constantes de type Constante, on va appeler le bon opérateur en fonction des types de ces deux constantes et ainsi c’est cette fonction qui gère l’appel aux bonnes fonctions.

#### L’iterator

Ce design pattern faisait partie de notre UML de départ, mais après réflexion, nous nous sommes dit qu’il valait mieux réutiliser les classes déjà présentes sous QT (ex : QStack) afin de ne pas s’embêter à les redéfinir, et donc nous avons adapté ces classes pré-existantes dans notre projet.

## Les autres choix de conception

Dans le sujet, il n’était pas signalé par exemple si on devait convertir la constante que l’on insère en fonction des paramètres, ainsi nous avons choisi de ne convertir la constante que lors du cas où la constante était un nombre non complexe et que les paramètres étaient configurés sur le mode complexe, et donc on convertissait la constante en complexe, mais quel que soit le mode concernant le type de constante (Entier, rationnel, réel) nous n’en avons tenu compte que lorsqu’un opérateur était appelé et donc une conversion était effectuée.

Plus particulièrement maintenant concernant les erreurs, étant donné que nous avions sélectionné de faire l’option concernant la gestion des logs, la création d’une classe LogMessage était nécessaire. Ensuite, au lieu de renvoyer une exception lors d’une erreur, nous avons choisi de renvoyer directement un objet de type LogMessage contenant le message et son ordre d’importance, et ainsi nous avions que très peu d’appel à LogSystem (que dans la classe MainWindow) qui lui par sa méthode addMessage, gère la création du message dans le fichier de log mais aussi modifie un élément de l’interface graphique afin d’afficher l’erreur.

De même concernant la gestion des erreurs mais cette fois due aux appels de fonctions, par exemple l’appel de cos sur un nombre complexe qui renvoie une erreur car cela était impossible, nous avons donc décidé d’implémenter cette méthode à l’intérieur de la classe Nombre et donc si la constante appelante est un complexe alors on renvoie directement une erreur. Le but de cette manœuvre est d’éviter de contrôler le type de la constante avant l’appel de la fonction et qui se trouve être une opération relativement couteuse.

La sauvegarde contexte a été gérée en XML, au départ nous nous étions tournés vers la sérialisation mais due à nos faibles connaissances sur ce sujet, l’XML nous a paru une solution beaucoup plus facile à mettre en place.

L’un des choix les plus importants a été de savoir si oui ou non, nous allions mettre les fonctions permettant l’appels aux autres bonnes fonctions (tel que push, l’évaluation, etc) dans une classe à part qu’on aurait appelé « Calculateur » par exemple ou alors de les intégré directement dans la classe MainWindow, et étant donné qu’il était beaucoup plus facile d’accéder aux éléments contenus dans la MainWindow depuis cette classe justement, nous avons donc choisi de mettre ces fonctions dans la MainWindow et qui étaient appelées par le biais des slots .

## Finalisation du projet

Avant de finaliser notre projet, nous avons effectué une batterie de tests afin de voir les potentielles erreurs et effectivement celle-ci en a révélé quelques une, notamment l’oubli d’une fonction (pow) et également quelques cas qui ne fonctionnaient pas mais que nous avons réussi à rectifier.  
Tous les commentaires nécessaires à la bonne compréhension mais également ceux permettant la génération de la doc Doxygen ont été rajoutés durant cette phase.

## Les diagrammes de séquences

## Conclusion

Ce projet nous a permis de gérer un projet depuis la phase de conception à celle du développement, tout en nous faisant réfléchir sur les designs patterns à utiliser. Le fait que le projet fût terminé à temps est relativement motivant, car ce n’était pas gagné dès le début vu le temps qui nous était imparti.