LO21 – Printemps 2016

RAPPORT DE PROJET

Maxime Churin - Nicolas Dejon – Louis Martignoni

Projet LO21

Table des matières

Introduction

Dans le cadre de l'UV LO21, nous avons comme projet de développer l'application UTComputer. Il s'agit d'un calculateur capable de gérer les opérations entre entiers, réels, rationnels, expressions et nombres complexes en utilisant la Reverse Polish Notation. Il est également capable de stocker et utiliser des variables et programmes.

Pour avoir une application efficace et portable, nous avons utilisé plusieurs design pattern à savoir singleton, template method, memento, strategy et iterator.

Ce présent rapport expose nos choix de conception pour répondre au problème, le diagramme UML sur lequel repose notre architecture (disponible en annexe), toutes les possibilités d'évolutivition qu'il offre, ainsi que les améliorations que nous pourrions y apporter.

Choix d'architecture

Pour la réalisation de ce projet, nous avons pensé notre architecture à la fois pour que l'application soit performante mais aussi portable pour facilement la faire évoluer.

Package Littérale

Nous avons créé la classe abstraite Littérale qui sera la classe mère de toutes nos littérales. Nous avons donc déclaré des méthodes virtuelles pures qui seront reprises dans les classes filles.

Nous avons ensuite créé une autre classe abstraite héritant de la classe Littérale, la classe Numérique. Elle regroupe les classes Entiere, Reelle et Rationnelle. Nous avons mis une relation d'agrégation entre Rationnelle et Entiere puisque qu'une littérale rationnelle est composée d'un numérateur et d'un dénominateur étant tous les deux des littérales Entiere. Une composition n'est pas envisageable puisqu'on a choisi que le cycle de vie entre les objets de ces deux classes soit indépendant, ceci ce justifiant par l'existence propre d'un objet de type Rationelle.

Nous avons ensuite créé la classe Complexe qui hérite de la classe Litterale. Un Complexe est composé de deux objets, le premier représentant la partie réelle et l'autre la partie imaginaire. Nous avons donc décidé de créer une relation de composition entre la classe Complexe et la classe Numerique. Ainsi, un objet Complexe est composé de deux objets Numerique et il gère leur destruction. Ici, on ne veut pas que nos objets Numerique perdurent après la destruction du Complexe.

Nous avons également les classes Expression, Programme et Atome qui héritent de la classe Litterale. Les variables sont stockées dans une map, chaque variable est composée de son identificateur qui sera la clé de la map, un identificateur d'atome, et de sa valeur, un pointeur vers une Litterale. L'utilisation d'une map permet une gestion simple de toutes les variables. Après réflexion une classe Variable aurait pu être crée mais nos choix initiaux de conception ne se sont pas orientés dans ce sens.

Pour stocker les objets Litterale, nous utilisons la classe LitteraleManager. Elle gère ses objets par le biais d'un tableau de pointeurs vers des Litterale. Nous avons décidé d'utiliser ici le design pattern Singleton car nous ne voulons pas avoir plusieurs LitteraleManager et ainsi empêcher sa duplication. Ceci pourrait être nocif pour notre application.

Package Computer Manager

La sauvegarde des données se fait au travers d'un document XML. Celle-ci est effectuée par la classe ComputerManager au travers d'un design pattern Strategy, implémenté sur la classe ComputerManagerDataContexte. Cela permet d'assurer le fonctionnement de l'application quelque soit le support utilisé pour stocker les données (document XML, base de données…). Pour le projet nous avons utilisé un document XML mais il aurait été tout à fait possible d'utiliser une base de données tel que SQLite.

La classe ComputerManagerDataXML est chargée de produire le document XML permettant le stockage de l'état de la pile, les paramètrages, les variables et les programmes du calculateur. Pour réaliser cela, la classe ouvre un fichier QFile dans lequel elle insère les données gardées grâce à des balises spécifiques : <pile> pour le stockage des littéraux stockés dans les sous-balises telles que <exp> pour expression, <prog> pour programme, et <renum>, <reden>, <imnum>,<imden> pour les littéraux numériques. Ce document peut ensuite être récupéré lors du démarrage suivant de l'application et interprété en ayant connaissance de ces balises. La lecture et l'analyse des données donnent lieu à leur reconstruction par l'empilage des littéraux dans la pile principale et par le remplissage de la map utilisée pour associer des objets atome et littéraux numérique, expression et programme.

La classe ComputerManager est une super-classe qui gère le stockage et le cycle de vie des deux autres managers principaux : LitteraleManager et Pile. C'est elle qui décide de la destruction des Managers, de leur persistance et de leur reconstruction.

Package Pile

Pour gérer les calculs et les littérales qu'utilise notre application UTComputer, nous avons implémenté une pile. Nous avons choisi de gérer cette pile comme une deque d'objets Litterale.

Ce conteneur comporte de nombreux avantages par rapport à un simple tableau de pointeurs vers des objets Litterale ou à une stack. En effet, la deque gère elle-même son expansion en mémoire. De plus, la deque comporte un iterateur qui nous permet donc de la parcourir séquentiellement, ce qui n'est pas le cas avec une stack. La deque nous permet donc d'avoir une pile efficace et simple d'utilisation. Il existe une relation de composition entre notre classe Pile et la classe Litterale. Nous avons ici aussi utilisé le pattern Singleton puisqu'on souhaite avoir seulement qu'une pile dans notre application.

Pour pouvoir implémenter les opérateurs LASTOP et LASTARGS, notre pile contient un attribut lastOp qui contient le nom du dernier opérateur utilisé et trois attributs lastarg qui contiennent chacun un des derniers arguments utilisés par le dernier opérateur. Ainsi, si on utilise un opérateur unaire, on affectera seulement une valeur à lastarg1, s'il est binaire à lastarg1 et lastarg2 et ainsi de suite. LASTARGS teste donc dans un premier temps l'arité de lastOp en comparant son nom dans des maps contenant les opérateurs unaires, binaires et ternaires et retourne les attributs lastArg en fonction du résultat du parcours des différentes maps. Notre classe Pile contient également tous les opérateurs de pile et une méthode particulière SavePile(). En effet, pour les opérateurs undo() et redo() qui font partie de la classe Controleur, il doit exister un moyen de stocker les états successifs de la pile. Pour faire ceci de manière efficace et portable nous avons utilisé le design pattern Memento. Nous avons donc une classe Memento dont les objets sont des états de la pile, à chaque modification de la pile la méthode SavePile() doit sauvegarder l'état actuel de la pile, ceci est fait en créant l'objet Memento correspondant à cet état.

Package Controleur

Nous avons également implémenté une classe Controleur, en utilisant le design pattern Singleton, qui s'occupe de stocker les états de la pile. Pour ce faire, nous utilisons un vecteur de pointeurs vers des objets Memento. En effet, le vecteur est un conteneur adéquat à notre utilisation des états de la pile, nous voulons seulement ajouter des objets Memento en fin de vecteur et grâce à notre compteur nous pouvons parcourir notre vecteur et donc parcourir nos différents états de la pile.

Le Controleur gère aussi les opérateurs UNDO et REDO, ces derniers utilisent le vecteur de pointeurs vers les objets Memento pour revenir aux états précédent de la pile ou repartir à l'état suivant. Il y a également la méthode ajoutMemento() qui permet de rajouter un pointeur d'objet Memento au vecteur et elle supprime également les états ultérieurs à l'état actuel de la pile pour permettre un bon fonctionnement des opérateurs UNDO et REDO.

L'évaluation des instructions de la calculatrice

La calculatrice a été conçue pour supporter le traitement en notation post-fixée ou Reverse Polish Notation. Cette dernière impose alors un traitement particulier des instructions entrées en ligne de commande puisque l'évaluation n'est pas comprise naturellement avec les règles d'évaluation classique.

Algorithme d'évaluation d'une ligne de commande en notation RPN

Pour évaluer une expression mise sous la forme RPN il est nécessaire d'utiliser une pile, qui sera l'élément central du fonctionnement de la calculatrice. En effet, chaque token composant la chaine de caractères peut être considéré placé dans une file d'attente de traitement. Ils seront évalués un par un avec chacun son propre traitement. Ce traitement apparaît dans notre classe contrôleur dans la méthode traitementRPN. Si jamais cette calculatrice devait évoluer, il y aurait une possibilité d'y inclure une autre forme de traitement de la ligne de commande.

Les littérales Entiere, Reelle et Rationnelle (qui sont des littérales Numerique) et Complexe provoquent un empilement sur la pile. Seul un opérateur +, -, *,/,$, opérateurs prédéfinis etc… peuvent permettre leur dépilement. La calculatrice accepte également des littérales programme qui sont

directement évalués s'ils ne contiennent pas de sous-programmes, ou empilés sinon, ainsi que des expressions écrites en notation infixe.

Algorithme d'évaluation d'une expression en notation infixe

La notation infixe de la littérale Expression ne peut bien évidement pas être évaluées par le fonctionnement de base de la calculatrice comme vu précédemment puisqu'elle est régie par ses propres règles de calcul. Cependant, nos recherches nous ont mené sur l'algorithme de Shunting Yard permettant la conversion d'une notation infixe en post-fixe que nous avons décidé d'implémenter. Cela permet donc de traiter directement l'expression après conversion comme s'il s'agissait d'une ligne de commande.

L'algorithme fait intervenir deux structures de données : une file et une pile. La pile est utilisée pour établir l'ordre de la notation RPN en accueillant les opérateurs, les fonctions et les parenthèses. Dès qu'on lit un opérateur, il est empilé jusqu'à ce que ses conditions d'arité et de sa précédence avec d'autres opérateurs soient remplies. Il est ensuite évalué. Lorsqu'une fonction apparaît elle est empilée avec sa parenthèse qui suit, puis tous les arguments le sont également jusqu'à l'arrivée d'une parenthèse fermante qui passe à l'analyse du token suivant.

A la fin du traitement la pile doit être vide et la file contient tous les tokens placés dans le bon ordre pour une exécution du traitement RPN.

Gestion des opérateurs

L'un des points fondamentaux de cette calculatrice RPN passe par la gestion des opérateurs. En effet, ceux-ci sont appliqués à différents classes assez couramment utilisées comme les Complexes, Reelles … mais aussi à des classes plus générales comme les Expressions.

Après une période de réflexion, nous avons choisi d'implémenter les opérateurs de façon non-membre par rapport à la classe abstraite Littérale : le patron général de tous nos opérateurs est donc Litterale& ...(Litterale& l1, ...). Pour cela nous avons défini dans la classe abstraite Littérale des accesseurs virtuels sur les numérateurs et dénominateurs de la partie réelle et imaginaire de l'objet. Pour les classes qui ne l'implémentent pas, une valeur par défaut de 1 pour les dénominateurs et de 0 pour les numérateurs nous permettent de simuler le comportement d'un complexe composé de rationnelle qui est notre objet le plus complet. Ces méthodes virtuelles sont ensuite redéfinies dans les classes filles pour accéder cette fois-ci à la vraie valeur de chaque objet.

Ces fonctions servent de base à la création du bon type de Littérale que retourne chaque opération. Nous avons alors implémenté la méthode bonType() qui prend donc ces 4 arguments, et qui construit le Littérale le plus approprié selon ces 4 paramètres.

Enfin dans chaque opérateur nous utilisons donc la bibliothèque typeid pour retrouver le type effectif de la Litterale passée en argument et effectuer selon les règles de l'opérateur le bon calcul.

Étant un groupe de trois, nous avons implémenté tous les opérateurs obligatoires et facultatifs numériques mais aussi logiques qui empilent une Entiere de valeur 0 ou 1 selon le résultat de la comparaison.

Cette implémentation nous a permis d'optimiser notre code. Pour comparaison, l'implémentation des opérateurs dans les classes filles de Litterale nécessitait une redondance

importante de code notamment pour gérer un opérateur s'appliquant sur X et Y mais aussi sur Y et X. Nous aurions aussi pu choisir d'implémenter les opérateurs par la méthode de la programmation générique.

Gestion du projet

Pour ce projet nous avons décidé de nous mettre à GitLab. Nous étions tous les trois novices sur cet outil. Au cours du projet, nous avons rencontré beaucoup de problèmes liés au merging et perdu de nombreuses fois de grandes partie de code. Tout le monde a ainsi pu percevoir le potentiel de cet outil qui devient de plus en plus indispensable dans tout projet conséquent.

Au travers de ce projet nous avons aussi découvert l'environnement Qt. Nous étions au départ focalisé sur notre IDE favori (CB, VS et XCode) et avons codé sur celui-ci. Nous avons ensuite intégré ce code dans Qt mais ce n'est qu'avec du recul que nous pouvons regretter de ne pas avoir directement commencé avec Qt qui propose de nombreuses fonctionnalités très intéressantes. On peut notamment citer l'utilisation de la QStack qui nous aurait grandement facilité la tâche. Enfin l'utilisation de Qt Designer est aussi très pratique pour auto générer notre interface graphique et créer les interactions avec ceux-ci.

Possibilités d'évolution

Deux des caractéristiques importantes de notre application sont sa portabilité et sa capacité à évoluer. On peut donc facilement lui ajouter des fonctionnalités.

Ainsi, notre application UTComputer couvre un grand éventail de fonctionnalités mais nous pouvons encore en ajouter pour l'améliorer et répondre aux besoins d'un plus grand nombre d'utilisateurs.

Dans notre version, nous pouvons visualiser les programmes stockés, les éditer ou en ajouter dans le même tableau. Une fenêtre d'édition permettrait d'améliorer la clarté de l'application. De même, lors de la fermeture de l'application, les littérales présentent dans la pile sont sauvegardées et seront toujours présentes à la prochaine ouverture. Cependant, les paramètres comme la taille de la pile à afficher ne sont pas sauvegardés, c'est une amélioration que l'on pourrait ajouter pour parfaire l'application.

Enfin, nous avons décidé de ne pas afficher les boutons "annuler" et "rétablir" puisque notre calculatrice possède déjà des boutons "UNDO", "REDO" et ces méthodes sont également connectées aux raccourcis CTRL-Z et CTRL-Y. Nous avons donc choisi de ne pas surcharger l'interface graphique avec des boutons doublons.

Design Pattern utilisés

- Les classes LitteraleManager, Pile, Controleur sont des singletons.
- La plupart des iterators que nous utilisons sont basés sur les iterators de la STL.
- Pour le stockage des données nous avons utilisé le design pattern Strategy.
- Utilisation du design pattern Memento

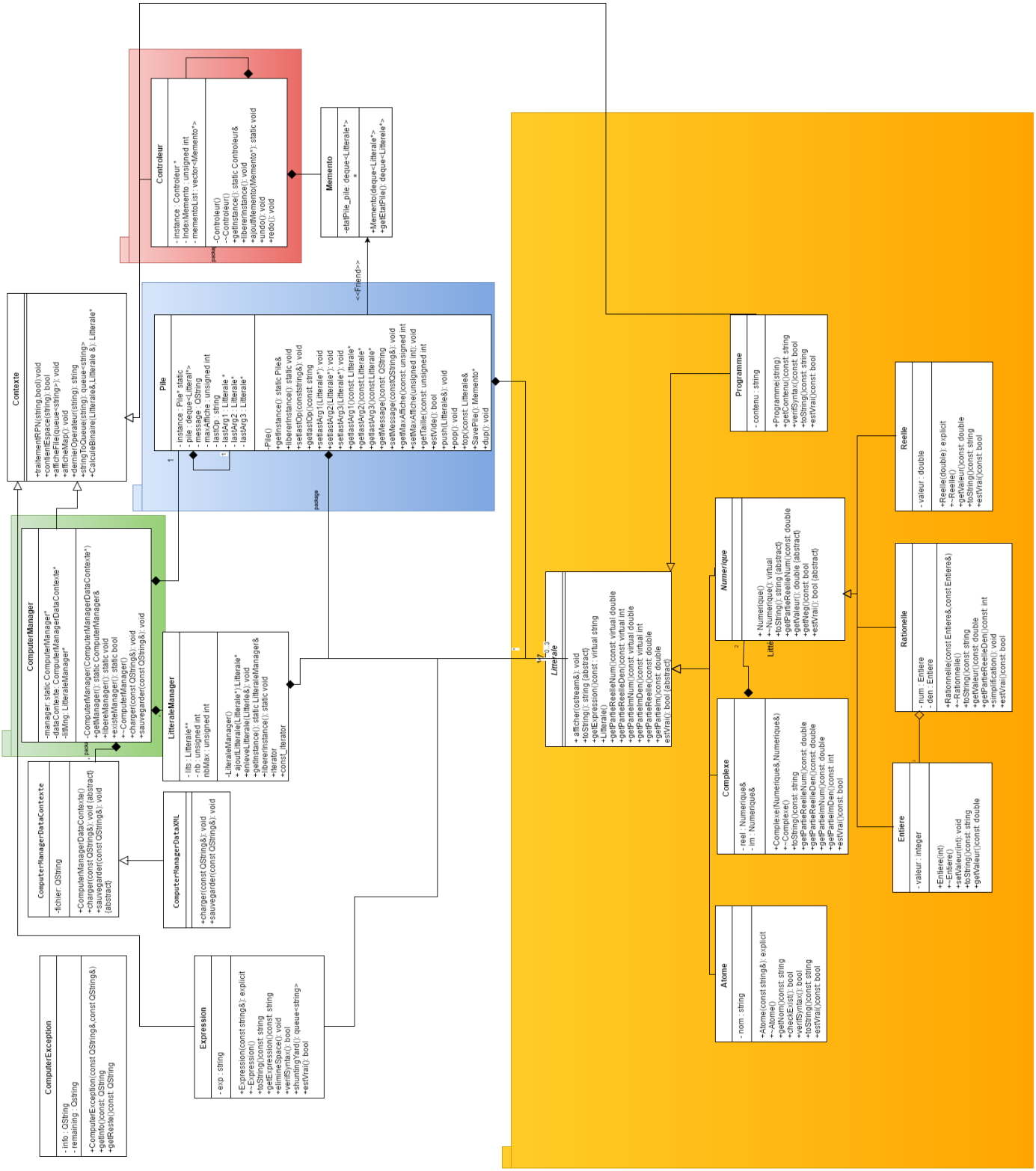
Conclusion

L'Unité de Valeur LO21 nous a permis de mener à bien un projet, depuis sa conception (détermination des objectifs, réflexion théorique sur la méthode à adopter, répartition des tâches) jusqu'à son achèvement, rendu possible grâce aux différentes étapes de sa réalisation(recherche d'information, manipulation de logiciel, élaboration d'algorithmes, code). Ce projet nous a donc donné la possibilité de nous familiariser avec un cadre de travail que nous sera susceptible de connaître plus tard, en tant qu'ingénieurs.

La réalisation de ce projet a été très enrichissante, tant au niveau scolaire que personnel. En effet, ce projet nous a permis d'appliquer dans un cas concret nos connaissances théoriques sur les possibilités offertes par le langage objet et la conception. Plus encore, atteindre un objectif au sein d'une équipe dans un temps imparti a nécessité de notre part une grande rigueur, le port d'une attention particulière à toutes les propositions, et une importante coordination entre les membres du groupe. De plus, ce projet nous a permis de développer notre esprit d'initiative. Enfin, nous avons pris conscience des difficultés liées au passage de la théorie à la pratique, puisque l'implémentation du modèle théorique s'est parfois révélé plus complexe qu'il n'y paraissait.

En outre, ce projet nous a permis de découvrir de nombreux outils collaboratifs et fonctionnalités du C++. Nous sommes satisfaits de notre calculatrice car il s'agissait d'un projet conséquent. Il reste cependant de nombreuses possibilités d'évolution.

ANNEXES


Contexte
Controleur
Memento
Pile
ComputerManager
LitteraleManager
ComputerManagerDataContexte
ComputerManagerDataXML
ComputerException
Expression
Litterale
Programme
Numerique
Reelle
Rationelle
Complexe
Entiere
Atome