Université Technologique de Compiègne

Projet LO21 - UTComputer

Rapport

HORDEQUIN Théo, LERAT Loïc, VIBERT Andréa

Juin 2016

Table des matières

Le projet UTComputer, développé dans le cadre de l'UV LO21, consiste en la création d'une Calculatrice utilisant la notation Polonaise inversée (RPN).

1 Description de l'architecture

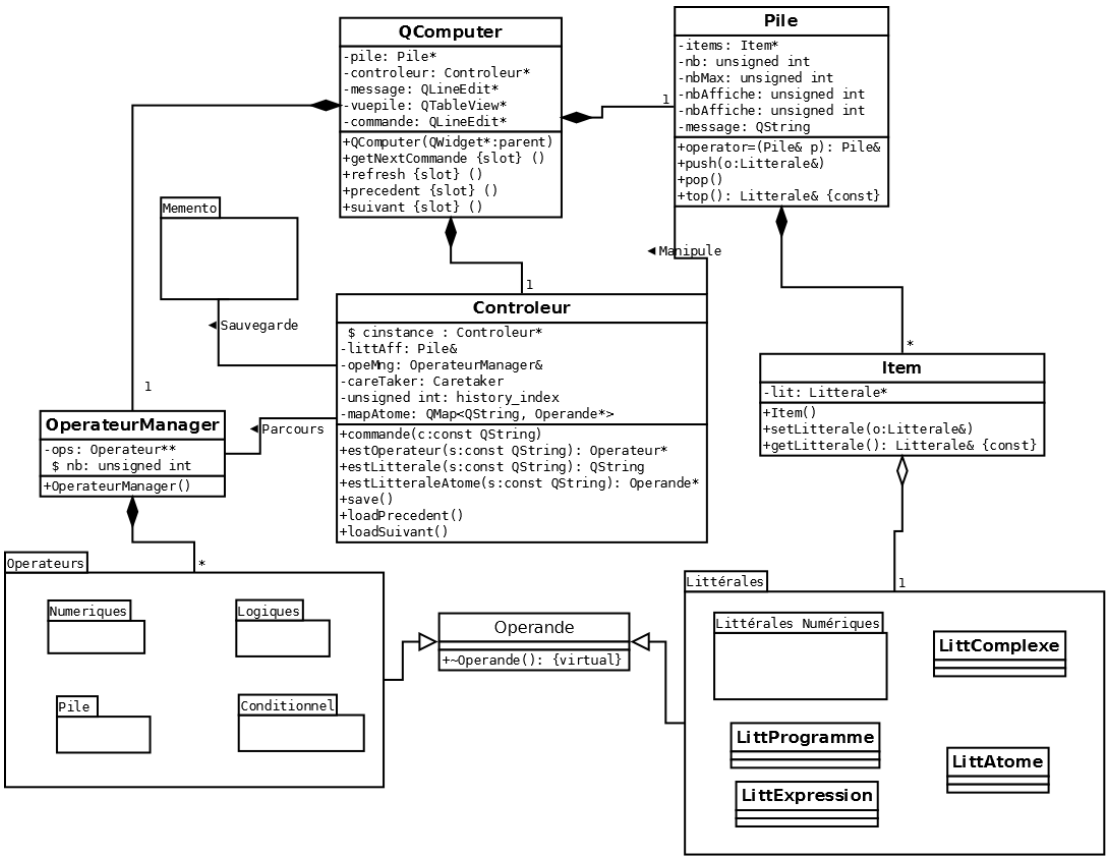

FIGURE 1 – UML simplifié de notre architecture

1.1 QComputer

La classe *QComputer* est le point d'entrée de notre application. Elle définit l'interface homme machine. Plus concrètement, nous intancierons dans cette classe tous les éléments visuels (Widgets) avec lesquels pourra interagir l'utilisateur.

QComputer se compose aussi d'une *Pile* permettant la gestion des *Operandes* traitées par la calculatrice, et d'un *Controleur* pour manipuler cette dernière.

1.2 Controleur (singleton et factory)

Une fois les instructions entrées par l'utilisateur de l'application, le *Controleur* va effectuer une batterie de tests vérifiant le validité des commandes rentrées. Si celles-ci sont valides, elles seront traitées en fonction de leur type. Les littérales seront empilées dans la *Pile* tandis que les opérateurs dépileront les littérales déjà existantes pour effectuer leur fonction de traitement. Son implémentation laisse la possibilité d'entrer une ligne de commandes composée de plusieurs instructions séparées par des espaces.

1.3 Memento

Afin de pouvoir proposer une fonction "Undo/Redo", nous implémentons le design pattern *Memento*. Il s'agit ici de sauvegarder chaque état de la Pile dans un objet *Memento*. Le *CareTaker* se charge de stocker chacune de ces sauvegardes dans un vecteur de *Memento* qui représente l'historique des états de la *Pile*. Le *Controleur* sera ensuite en mesure de restaurer l'un de ses états (le précédent ou le suivant dans l'historique).

1.4 Pile

La *Pile* est chargée de stocker les *Litterales* manipulées lors de nos calculs. Elle seront successivement empilées ou dépilées. Cela est parfaitement adapté à la notation RPN que nous utilisons dans cette calculatrice. La *Pile* est composée d'*Items* qui pointent chacun sur une littérale. Elle contient aussi des messages destinés à l'utilisateur (erreurs, opérations impossibles, ...). Ces messages seront affichés dans le *QComputer*.

1.5 Opérandes

La calculatrice doit être en mesure d'associer une chaîne de caractères composée de lettres majuscules et de chiffres (une littérale atome) à une littérale ou un opérateur. Nous avons donc choisi d'établir une classe Opérande, abstraite et totalement vide, dont héritent les opérateurs et les littérales. Actuellement, notre calculatrice ne permet pas l'association d'atome aux variables, mais cette structure laisse la possibilité de développer cette fonctionnalité plus tard.

1.5.1 Opérateurs

A chaque opérateur implémenté par la calculatrice est associé une classe, spécialisant la classe abstraite *Operateur*. Un opérateur est associé à un symbole, qui permettra sa reconnaissance lors de la saisie de l'utilisateur, une arité, qui détermine le nombre de littérales à dépiler pour effectuer l'action attendue, et une fonction traitement prenant en paramètres les littérales dépilées. C'est cette méthode qui définit les opérations à effectuer et qui retourne la littérale qui en résulte. Grâce au polymorphisme et au caractère virtuel de cette méthodes nous pourrons la redéfinir pour chaque opérateur en fonction de son arité et des actions à effectuer.

OperateurManager (Singleton)

Dès sa création, la classe *OperateurManager* instancie et stocke tous les opérateurs. Un attribut statique *nb* fournit le nombre d'éléments contenus dans le tableau d'opérateurs. Les itérateurs que nous lui définissons permettent un parcours séquentiel de ces opérateurs. Ainsi le *Controleur* sera en mesure de récupérer l'opérateur correspondant à une commande entrée par l'utilisateur et exécuter sa fonction traitement.

1.5.2 Littérales

Les littérales sont les objets manipulés pour effectuer les calculs. Chaque littérale peut être affichée dans l'interface, c'est pourquoi la classe *Litterale* possède une méthode virtuelle *affichage()*. De la même façon, nous définissons une méthode virtuelle *simplifier()* qui permet d'effectuer certaines simplifications propres à chaque type de *Littérale* (simplification de fraction, etc...).

- *LittNumber* : Cette classe regroupe les littérales numérique et la classe de littérales complexes. Toutes les classes qui héritent de celle-ci représentent des nombres et disposent de surcharges pour les opérateurs « + », « - », « / » et « * » afin de définir la manière de réaliser ces calculs selon le type de littérale en jeu.
  - *LittNumerique* : Cette classe abstraite généralise les différentes Littérales représentant des nombres (non complexes). Elle permet de spécifier le type des deux parties qui peut être attribué aux deux éléments d'une littérale complexe. Les *LittRat*, *LittEntiere*, *LittReelle* héritent de la classe LittNumerique et fonctionnent toutes les trois d'une façons assez similaire et intuitive.
  - *LittComplexe* : Chaque objet de cette classe se compose de deux pointeurs vers des littérales numériques pour former les parties réelles et les parties complexes.
- *LittProgramme* : cette catégorie de littérales enregistre une ligne de commande entourée par des crochets dans un attribut *str* dans un format *QString*. Empilée telle quelle, cette ligne ne sera évaluée qu'à l'exécution d'un opérateur *EVAL*.

Le sujet présentait en plus des littérales précédentes les littérales atomes et les littérales expressions. Les premières, se trouvent dans notre contrôleur dans une structure *QMap*, attribut de *Controleur*, qui permet de faire le même travail de référencement vers les opérateurs et potentiellement vers des expressions et des littérales (même si nous n'avons pas permis ces deux cas de figure). Concernant les littérales expressions, nous avons eu des difficultés à les implémenter et avons donc fait sans.

2 Argumentation

2.1 Respect des bonnes pratiques de codage

La déclaration des destructeurs des classes abstraites comme étant *virtual* garantit aussi un respect du principe de substitution. En mettant les attributs de nos classes dans les parties privées ils ne sont accessibles que par l'emploi de méthodes de classe. Ce respect du principe d'encapsulation contribue à la sécurité de notre programme.

2.2 Protection (singletons, private, const, etc...)

Les méthodes getters de nos *l*ittérales sont const, afin qu'une littérale ne puisse être modifié directement.

En l'état actuel, notre architecture permet de garantir une certaine sécurité. Certaines classes se trouvent être des singletons afin de s'assurer quelles ne seront instanciées qu'une seule fois lors de l'exécution de notre programme. Les deux classes qui bénéficient de la protection de ce design pattern sont les classes *operateurManager* et *Controleur*.

2.3 Praticité

La présence d'un attribut de type *Qmap* dans la classe *Controleur* est un choix permettant d'effectuer les fonctionnalités des littérales atomes décrites dans le sujet. On associe au symbole des opérateurs, ayant des capitales et chiffres, un pointeur vers cet opérateur. Dès que cet atome récupéré par le *Controleur* est identifié, il permet le traitement de l'opérateur correspondant. Cette façon de procéder permet d'envisager des associations à des pointeurs vers des littérales ou des programmes ayant un atome pour les référencer sans problème.

Pour sauvegarder les états de notre pile, nous avons aussi employé le design pattern Memento. Une classe *CareTaker*, vector d'objets de classe *memento*, permet de parcourir des sauvegardes effectuées à chaque changement d'état. Nos littérales restent donc présentes tout le long de l'application et ne seront supprimées qu'à l'arrêt du programme.

Dans le but d'accepter les lignes de commandes comportant des espaces, ainsi que les programmes, une factory d'opérandes est créée et permettra d'effectuer un traitement adapté à chacun des éléments.

2.4 Adaptabilité, évolution

En répartissant les tâches à des classes précises (*Controleur* pour la gestion des commandes, l'interface par le *Qcomputer*, la gestion des opérateurs par l'*OperateurManager*, ...) permet une assez grande modularité de notre code et améliore l'adaptabilité de l'application.

Le fait d'avoir une classe par opérateur permet d'envisager des ajouts en aval sans que cela ne vienne perturber l'existant. De plus, l'emploi de la classe *OperateurManager* qui permet de parcourir séquentiellement les opérateurs sans avoir à manipuler la classe *Controleur* dans le cas d'un ajout d'opérateur.

De la même façon, le fait d'avoir la classe mère *litterale* abstraite, et de spécialiser étape par étape nos littérales (*littNumber*, *littNumérique*, ...) permet d'envisager la création de nouvelle littérale. Par exemple, une classe littérale qui permettrait d'encoder des nombres en base 2 pourrait être ajoutée. Ou même, une classe littérale expression que nous n'avons pas faites mais qui pourrait être ajoutée en héritage direct de la classe *litterale*. Néanmoins, nous avons conscience que l'usage des *dynamic cast* dans les surcharges d'opérateur, force une modification des surcharges d'opérateurs des autres littérales afin de traiter tous les cas de figures. Ce qui n'est peut-être pas la solution optimale.

3 Annexes

3.1 Instructions d'utilisation

Pour un bon usage de notre programme il est nécessaire de prendre en compte les consignes suivantes :

— Penser aux espaces lors de la saisie d'une ligne de commande contenant plus d'une opérande à évaluer, de même pour les littérales programmes.

4 Retours et conclusions

Si nous n'avons pas pu développer l'intégralité des fonctionnalités proposées par l'énoncé, ce projet a tout de même été l'occasion d'avoir un aperçu des difficultés liées aux projets assez conséquents et se rapprochant des problématiques réelles.

Les relations d'interdépendances entre les différents membres de notre architectures nous ont contraints, à plusieurs reprises, d'effectuer des changements au cours du développement. Une architecture, même bien pensée au départ, est souvent remaniée dans la phase de codage. D'autant plus lorsque l'on n'a pas l'habitude de programmer ce type d'application. Nous avons donc appris, à nos dépends, qu'il ne faut pas se focaliser uniquement sur le conceptuel en espérant tout pouvoir codé d'une traite. La gestion de groupe et du temps imparti sont aussi des difficultés que nous avons rencontrées dans ce projet qui ne peut pas se faire intégralement dans un court laps de temps.