Université de Technologie de Compiègne

LO21

Rapport de projet

UTComputer

Printemps 2016

Jérome COSTE - Gabriel HURTADO

12 juin 2016

Table des matières

Première partie

Projet et architecture

1 Objectifs

L'objectif de ce projet était de réaliser une calculatrice utilisant la notation RPN.

Dans le cadre du projet de LO21, nous avons été chargé de la programmation de ce logiciel et de son interface.

Cette calculatrice devait être capable de gérer tout types de nombres, mais aussi des programmes simples, et des variables. Elle devait par ailleurs comporter une interface graphique, avec une vue principale comportant la pile, le champ de saisie de commande ainsi que divers boutons, mais aussi des vues secondaires.

1.1 Équipe et organisation

Le projet était constituée de deux étudiants :

— Jérôme Coste, étudiant en GI02

— Gabriel Hurtado, étudiant en GI02

Etant relativement peu expérimentés dans le domaine de la programmation orientée objet, nous nous sommes beaucoup servis des notions aquises lors de l'UV LO21, notamment au niveau des design patterns.

Au niveau des outils, nous avons utilisé :

— **Draw.io** : l'outil en ligne, permettant la mise en ligne d'une version collaborative de notre UML

— **Qt Creator** : l'IDE, environnement de développement, permettant la création facilité d'applications fenêtrées

— **Qt Designer** : cet environnement de création d'interface graphique à simplifier la mise en forme de celle-ci

1.2 Répartition du travail

Au vu de la charge importante de travail, il à été nécessaire de trouver une organisation satisfaisante. Lors d'une étape préliminaire, nous avons travaillés en collaboration sur la modélisation du problème, en étudiant ensemble les différents design patterns et les différentes solutions qui en découlaient. Suite à cette phase de modélisation, nous avons mis en place un repository sur Github, et nous nous somme répartis les différents "blocs" de notre UML. Jérôme s'est principalement occupé de l'interprétation des commandes saisies, de l'interface graphique, de la gestion des littérales, de la pile et des opérateurs Undo et Redo. Gabriel, s'est occupé des différents opérateurs, de la sauvegarde et restauration du contexte lors de l'échec d'opérations, ainsi que de la gestion des variables. Certaines parties ont été faites en collaboration, comme les expressions, et les littérales.

1.3 Réunions hebdomadaires

Pour mener à bien efficacement ce projet, nous avons choisi d'utiliser une méthode Agile : SCRUM

SCRUM repose sur des sprints, qui sont des périodes de travail généralement étalée sur une à plusieurs semaines avec des objectifs précis clairement définis au début de chaque période. Nous avons travaillé d'un façon très similaire, avec des réunions au moins hebdomadaires.

1.4 Git

Notre *workflow* était principalement basé sur l'outil Git, couplé à la plateforme web Github. Cela offrait plusieurs avantages indéniables :

— Reporter les bugs facilement en ouvrant des tickets. La fonctionnalité s'appelle *Issues* sur Github. Ainsi, dès que quelqu'un repérait un bug, dû à du code écrit par cette même personne ou quelqu'un d'autre, nous ouvrions un ticket. Cela permettait de suivre l'évolution de la résolution de bugs et d'en garder une trace.

— Le travail à distance était grandement facilité : il est possible de travailler conjointement sur un même document, grâce aux outils de résolution des conflits.

2 Choix d'architecture

La premier travail à effectuer, étant donné que le cahier des charges exprime clairement les besoins, a été un travail de modélisation de notre architecture. L'utilisation d'une pile pour le stockage, couplée au design pattern Mémento pour la gestion des sauvegardes de la pile, nous a semblé évident.

La hiérarchisation des classes dérivant de la classe *littérale* était clairement exprimée dans le sujet. Nous avons ainsi choisi de modéliser une classe *littérale complexes*, qui exclue les programmes, atomes et expressions. Ces *littérales complexes* ont pour particularité d'exprimer des nombres. Ainsi ces littérales partagent certaines similitudes telles que la possibilité d'une méthode renvoyant leur négation.
Les *littérales numériques* caractérisent toutes les *littérales complexes* n'étant pas des complexes. Même si à ce niveau de hiérarchisation, cette classe n'apporte aucun avantage technique, elle permet de décrire très simplement un complexe comme composé de deux *littérales numériques*. Il en est de même pour la classe *littérale simple* qui n'offre pas non plus d'avantage technique, mais permet de bien exprimer la structure.
Comme un dessin vaut 1000 mots, voici la hiérarchisation complète des littérales

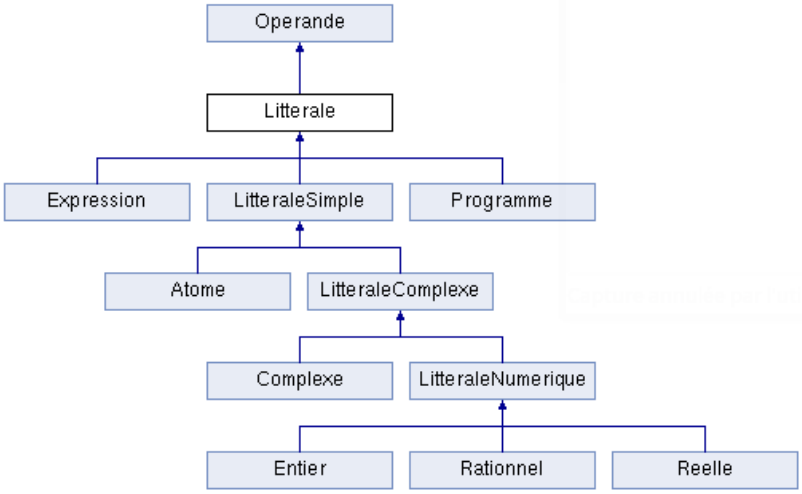

FIGURE 2.1 – Hiérarchisation des littérales

L'un des points de la conception, qui s'est révélé être le plus délicat a été la modélisation des opérateurs. Nous avions tout d'abord opté pour la surcharge d'opérateurs, car il s'agit d'un outil formidable de la programmation Orientée Objet, mais en étudiant le nombre de combinaisons possibles à traiter, et en réfléchissant à la souplesse de cette solution, nous nous sommes rendu compte qu'elle n'était pas viable. Nous avons alors opté pour des opérateurs sous forme d'objets instanciables, qui sont beaucoup plus faciles à maintenir. La hiérarchie des classes d'opérateur se base sur un héritage multiple. Une partie de l'héritage exprime l'arité de l'opérateur (unaire ou binaire), et l'autre partie exprime son type (conditionnel, numérique, pile,...). L'avantage d'une telle conception est que chaque classe mère apporte ce qui est nécessaire. Par exemple, la classe OpérateurBinaire apporte deux attributs littérale* .

La modélisation d'un contrôleur qui coordonne la pile les opérateurs et les littérales, ainsi que la modélisation de l'interface graphique et des variables, ont émergées durant la phase de programmation, car il nous fallait avant tout définir clairement comment seraient construit les littérales et les opérateurs à partir d'une chaine de caractère

Nous avons donc opté pour deux "factory", une pour les opérateurs et une pour les littérales, qui prenaient en entrée une chaîne de caractère, afin de produire le bon opérateur ou la bonne littérale en sortie.

La phase de conception s'est achevée sur cette réflexion.

L'UML présent en annexe témoigne de l'état de la conception avant le début de la programmation.

3 Fonctionnement de la calculatrice

Dans cette partie nous allons voir plus en détail le fonctionnement de routine principale de la calculatrice en supposant l'arrivée d'une opération sur la ligne de commande

3.1 Le contrôleur

La méthode getNextCommande() de l'interface principale s'occupe d'envoyer la chaîne de caractère tapée dans la ligne de commande graphique au contrôleur via sa méthode commande().

Ce dernier doit alors trouver le premier mot de la chaîne envoyée, grâce à la méthode firstWord() du contrôleur (fonctionnement détaillé dans la documentation). Suite à cela, il tente de fabriquer une littérale à partir du premier mot via la LitteraleFactory. Si celle-ci renvoie nullptr, la fabrication à échouée , sinon la littérale obtenue est empilée. Dans tous les cas le contrôleur tente ensuite de fabriquer un opérateur avec ce même mot en l'envoyant dans l'OpérateurFactory. Si celle-ci renvoie nullptr, la fabrication a aussi échouée, sinon il demande à l'opérateur d'effectuer son opération. Ensuite, le contrôleur passe au mot suivant jusqu'à qu'il finisse.

3.2 La LitteraleFactory

La LitteraleFactory est appelée par le contrôleur via la méthode creerRPNLitterale. Il reçois une String, et demande à la méthode getRPNExample() de lui fournir un exemple de littérale construit sur ce modèle. La méthode gerRPNExample() cherche alors parmi tout les symboles enregistrés, par ordre de priorité, un qui existe dans la string donnée. Dès qu'il trouve, il renvoie l'exemple de littérale associé au symbole. La méthode creerRPNLitterale se charge alors d'appeler la méthode getFromString() de la littérale avec la string, pour recevoir une copie de la littérale, chargée avec les valeurs de la string.

3.3 L'OpérateurFactory

Celle ci cherche juste à faire correspondre la string avec un symbole d'opérateur connu. S'il trouve, il prends l'exemple associé à cette littérale et en demande une copie, qu'il renvoie au contrôleur.

3.4 Ce qu'il manque

Par manque de temps nous avons préféré privilégier au maximum la gestion de la mémoire, au détriment des opérateurs optionnels. D'autre part, nous avions un doute sur le but de l'opérateur edit, permettant de modifier un programme référencé par un atome sur la pile. Étant donné qu'un atome ne peut être sur la pile que si il n'est pas lié à une variable ou un programme, nous avons choisi de lui faire ouvrir le menu d'édition des programmes. D'autre fonctionnalités n'ont pu être implémentées, faute de temps. Par exemple, quand un opérateur comme + * ou - est saisi sur la "ligne de commande", il n'est pas directement interprété, il est nécessaire de faire "Entrée". Finalement, nous n'avons pas eu le temps d'afficher les délimitateurs gauche et droits de la littérale expression ou programme dans le cas ou elle "déborde"

Deuxième partie

Adaptabilité

Au niveau de l'adaptabilité, il était nécessaire de faire un choix. Nous avons considéré que l'ajout de nouveaux opérateurs était bien plus fréquent que l'ajout de nouvelles littérales. Nous avons donc mis l'accent sur la flexibilité des opérateurs, parfois au détriment de celle des littérales.

3.5 Adaptabilité des littérales

La création de littérale peut se faire en héritant au minimum de la classe littérale. Pour enregistrer une nouvelle littérale, il faudra donc définir son comportement pour quelques méthodes comme getFromString(), getCopy(),etc et enregistrer la littérale auprès de la litteraleFactory avec la méthode enregistrer() (ou enregistrerInfix(), si l'on veut qu'elle ne soit comprise que dans des expressions.). L'enregistrement requiert au minimum un symbole et une priorité différente de celles définies de base (visibles dans le fichier main.cpp), mais l'on peut aussi définir la façon dont la littérale est détectée dans le contrôleur, en définissant une classe héritant de WordIdentifier, qui redéfinit les méthodes wordGuesser() qui vérifie que le mot est bon et wordPosition() qui permet l'extraction du mot. On peut bien entendu omettre ce paramètre et le comportement par défaut fait que le mot se fini au prochain espace (ou fin de la ligne de commande). Ainsi, ces littérales nouvellement ajoutées seront comprises partout. Le seul problème est la compatibilité de celles-ci avec les opérateurs. Leur comportement n'étant pas défini, il faudra modifier directement les opérateurs en question.

3.6 Adaptabilité des opérateurs

L'adaptabilité est l'avantage principal de notre choix d'architecture, au niveau des opérateurs. En effet, si l'utilisateur souhaite définir un nouvel opérateur, il n'a qu'a le faire hériter des classes qui le constitue, et son comportement sera en très grande partie hérité de ces classes supérieures. Par exemple, pour implémenter un opérateur comme sinus, il suffit de le faire hériter d'opérateur numérique, et d'opérateur préfixe. La seule méthode à définir sera *traitementOperateur()*, définissant ce que l'opérateur doit faire de ses deux littérales. Il faudra par la suite enregistrer cet opérateur auprès de sa factory. Grâce à cette architecture, l'opérateur n'a à s'occuper ni de savoir si elles viennent de la pile, ni à les sauvegarder pour la gestion du contexte. Son seul travail sera de définir le comportement, et éventuellement d'émettre une exception si les littérales reçues sont d'un mauvais type, ou si l'opération demandée est sémantiquement incorrecte.

Cependant ce choix d'architecture présente un désavantage. Lors de l'ajout d'une nouvelle littérale, il est nécessaire de modifier une partie des méthodes *traitementOperateur()* des opérateurs pouvant s'appliquer à ce nouveau type de littérale. Le désavantage de ce choix d'architecture est clair, mais nous avons considéré qu'il sera très rarement utile de rajouter de nouvelles littérales, et donc qu'il était justifié de privilégier l'adaptabilité des opérateurs.

Troisième partie

Annexes

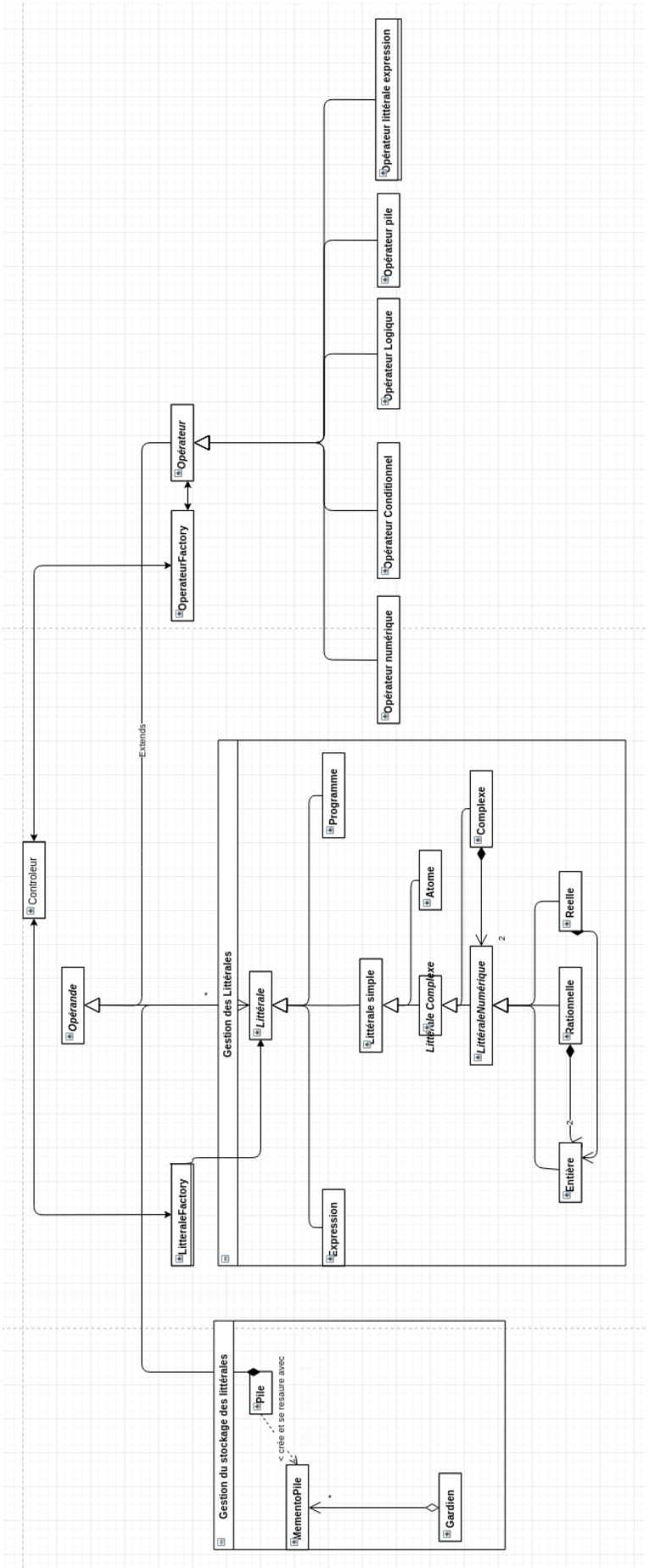

FIGURE 3.1 – Uml initial de notre calculatrice