Université technologique de Compiègne

Projet LO21

Printemps 2016

UTComputer

Groupe :
Quentin Duchemin et
Aurélie Digeon

Responsable LO21 :
Antoine Jouglet

Table des matières

1 Notes **2**
- 1.1 Le sujet . . . 2
- 1.2 Fonctionnalités implémentées . . . 2
- 1.3 Fonctionnalités supplémentaires . . . 2
- 1.4 Utilisation des pointeurs intelligents . . . 2
- 1.5 Vocabulaire . . . 2

2 Description de l'architecture **3**
- 2.1 Objets Métiers . . . 3
  - 2.1.1 Littérales . . . 3
  - 2.1.2 Opérateurs . . . 4
  - 2.1.3 Opérandes et Arguments : les utilitaires . . . 4
  - 2.1.4 Opérations . . . 5
  - 2.1.5 Exceptions . . . 6
  - 2.1.6 Pile . . . 7
- 2.2 Couche Logique . . . 7
  - 2.2.1 Fabrique de Littérales . . . 7
  - 2.2.2 Manager d'Opérateurs . . . 8
  - 2.2.3 Manager de l'application . . . 8
  - 2.2.4 Parser d'expressions . . . 9
- 2.3 Couche Graphique . . . 9
  - 2.3.1 Organisation des élèments . . . 9
  - 2.3.2 Fenêtre des paramètres . . . 10
  - 2.3.3 Evénements sur la ligne de commmande . . . 10
  - 2.3.4 Sauvegarde . . . 10
  - 2.3.5 Calcul et gestion des erreurs . . . 11

3 Evolutivité de l'application **11**
- 3.0.1 Couplage faible . . . 11
- 3.0.2 Extensions de l'application . . . 12

4 Améliorations possibles **13**

A Annexes **14**
- A.0.1 Exemples avancés . . . 14
- A.0.2 Instructions de compilation . . . 14
- A.0.3 Contenu du rendu . . . 15
- A.0.4 Easter-eggs . . . 15

1 Notes

1.1 Le sujet

On souhaite développer l'application `UTComputer`, un calculateur scientifique permettant de faire des calculs, de stocker et de manipuler des variables et des programmes, et utilisant la notation RPN (`Reverse Polish Notation`, i.e. la notation polonaise inversée ou encore notation post-fixée). La notation polonaise inverse est en fait une méthode de notation mathématique permettant d'écrire une formule arithmétique de façon non ambiguë sans utiliser de parenthèses.

1.2 Fonctionnalités implémentées

Toutes les fonctionnalités du sujet ont été implémentées, incluant les optionnelles. Ceci inclut tous les éléments d'interface et tous les opérateurs obligatoires optionnels. Nous n'excluons pas que quelques bugs non-rencontrés jusqu'à maintenant subsistent, mais l'implémentation est entièrement fonctionnelle.

1.3 Fonctionnalités supplémentaires

Nous trouvions pratique de pouvoir se déplacer dans l'historique des commandes (au même titre que dans la plupart des terminaux) à l'aide des flèches verticales du clavier. Nous avons donc implémenté cette fonctionnalité pour la ligne de commandes. Y est associée une fenêtre permettant de visualiser l'historique des commandes dans le menu `Tools`. Enfin, l'historique des commandes est sauvegardé lors de la fermeture de l'application, de sorte que les commandes précédentes soient accessibles lors d'un futur lancement.

1.4 Utilisation des pointeurs intelligents

Nous avons choisi d'utiliser les **pointeurs intelligents** de la librairie standard, en particulier les `shared_ptr`. Ces objets spéciaux encapsulent un pointeur et gèrent son cycle de vie via un compteur de références. Lorsque ce dernier tombe à zéro, la mémoire allouée est libérée. Ceci nous évite d'avoir à manager manuellement le cycle de vie des objets manipulés, et d'être certains d'éviter les fuites de mémoire.

1.5 Vocabulaire

Pour alléger les phrases dans la suite du rapport, nous utiliserons l'expression « *renvoie des littérales* » ou « *renvoie des opérandes* » pour signifier « renvoie une *collection* (*i.e.* un conteneur) d'objets `shared_ptr` pointants sur des objets `Literal` / `Operand` génériques ». Un vocabulaire similaire est utilisé pour les arguments de fonctions. Il n'y a pas d'ambiguité

car nous n'utilisons jamais de collections d'objets concrets, mais uniquement des collections de pointeurs.

2 Description de l'architecture

2.1 Objets Métiers

Les littérales et les opérateurs ont été créés selon cette hiérarchie :

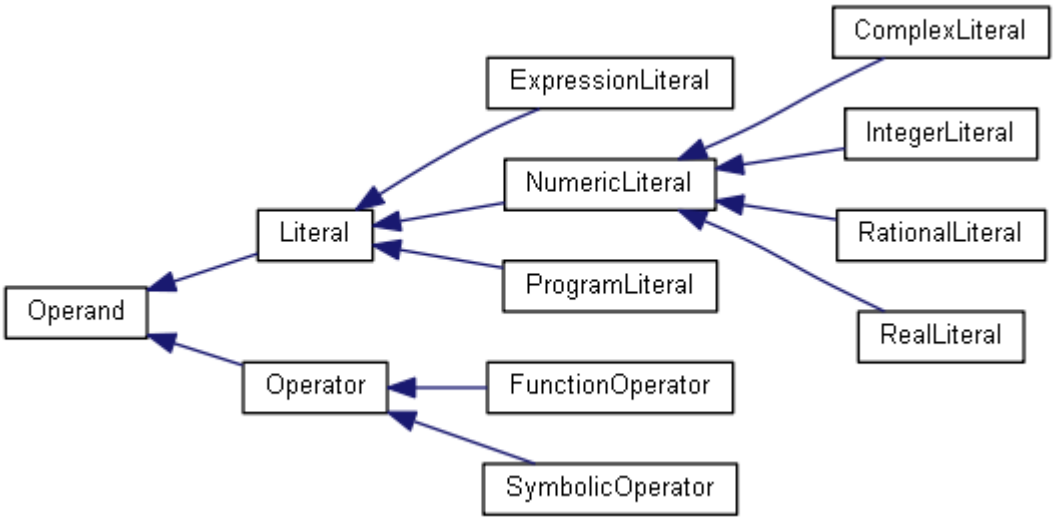

FIGURE 1 – Hiérarchie des objets n'implémentant pas de logique.

Nous détaillons ce schéma d'héritage dans la suite de cette section.

2.1.1 Littérales

Toutes les littérales héritent de la classe abstraite `Literal`, qui déclare et définit des opérateurs de *cast* virtuels vers toutes les littérales concrètes. Ces opérateurs sont particulièrement utiles pour les opérations, que nous détaillons en sous-section 2.1.4. Si la littérale concrète n'a pas ré-implémenté l'opérateur de cast, une exception de type `TypeError`[1] est levée.

Chaque littérale définit également un opérateur de cast (cette fois-ci non générique) vers des types de la librairie standard. Par exemple, un objet `IntegerLiteral` peut être utilisé comme un `int`, et un objet `ComplexLiteral` peut être utilisé comme un `complex`.

1. Le fonctionnement des exceptions personnalisées d'`UTComputer` est détaillé à la sous-section 2.1.5.

Nous avons choisi de ne définir aucun comportement, *ni* de calcul *ni* de simplification au niveau des littérales. Ces objets demeurent de simples abstractions sur lesquelles on peut projeter un comportement qui n'est pas figé. L'évolutivité de l'application est détaillée à la section 3.

Les littérales atomes ne font pas l'objet d'une classe, car nous les traitons comme un simple état temporaire. En effet, un atome est immédiatement transformé en une des littérales existantes après traitement dans la ligne de commande.

Enfin, les littérales programmes sont implémentées dans la classe `ProgramLiteral` en utilisant le design pattern **Composite**, leur permettant de contenir d'autres programmes (*noeud*) ou d'autres littérales (*feuille*).

2.1.2 Opérateurs

Un opérateur est représenté par la classe abstraite `Operator`. Chaque opérateur est principalement composé d'un symbole, d'une arité et d'un *comportement*. C'est via le design pattern **Strategy** que nous définissons ce comportement, avec un pointeur sur un objet `Operation`[2.1.4]. Tout comme les littérales, un opérateur n'est qu'une abstraction dont le comportement est modulable : il n'y a aucune raison de lier indéféctiblement un symbole ou une chaîne à un comportement.

Nous avons séparé deux grands types d'opérateurs : les opérateurs *fonctionnels* et les opérateurs *symboliques*. Un opérateur fonctionnel est vu comme une fonction, avec l'écriture associée, tandis qu'un opérateur symbolique est vu comme un opérateur d'arité 2 ayant une **priorité**, qu'il est nécessaire de définir pour le *parsing* d'expressions ou les opérations symboliques sur des expressions.

Nous différencions aussi, cette fois-ci par des booléens, le fait qu'un opérateur soit *numérique* ou *arithmétique* :

- Un opérateur est **numérique** si sa sémantique effectue un **calcul** et produit un résultat interprétable numériquement (*e.g.* $\geq$, $\sqrt{X}$, $\neq$, $+$, *etc.*) ;
- Un opérateur est **arithmétique** si son symbole n'est composé que d'un caractère et qu'il produit un nombre (*e.g.* $+$, $-$, $/$, \$, *etc.*).

Cette différenciation nous permet de choisir le comportement à adopter dans les situations ambigües : à titre d'exemple, un opérateur symbolique devra être traité avec plus grand soin lors d'opérations sur des littérales expressions afin d'ajouter des parenthèses ou non.

2.1.3 Opérandes et Arguments : les utilitaires

Les classes `Literal` et `Operator` héritent d'une classe `Operand` qui nous facilite l'implémentation. En effet, certaines opérations peuvent produire à la fois des littérales et des

opérateurs, *e.g.* `EVAL`. Ces classes ont aussi en commun le fait d'être gérées **indifféremment** par la pile d'UTComputer. Enfin, la classe `Operand` définit une méthode virtuelle pure `toString`, utilisée pour tous les affichages.

La classe templatée `Arguments` est un *wrapper* (*i.e.* un adapteur de classe public) de `vector`. C'est ce conteneur que l'on manipule dès lors qu'on effectue une opération. Pour les types réels, il n'ajoute aucun comportement. En revanche, pour les pointeurs (`shared_ptr`, ici), il ajoute deux fonctions de cast donc l'utilisation est paramétrée par les *type traits* de C++11. Le comportement de ces opérateurs est de *caster* l'objet pointé, puis de l'encapsuler dans un pointeur sur le nouveau type, pour chaque élément du vecteur. À titre d'exemple, il sera possible d'effectuer la conversion d'un `Arguments<shared_ptr<IntegerLiteral»` en `Arguments<shared_ptr<ComplexLiteral»` directement.

2.1.4 Opérations

Une opération est representée sous la forme d'une classe dérivée de `Operation`. La classe `Operation` définit une méthode d'évaluation générique pour pointeurs de `Literal` ainsi que des méthodes d'évaluation virtuelle pour pointeurs sur type concret de littérale, qui par défaut lèvent une exception `UTException`. Une opération produit des pointeurs sur `Operand`.

La raison de l'existence de méthodes virtuelles d'évaluation à la fois génériques et spécialisée est la suivante : nous voyons un objet `Operation` comme un élément d'une librairie. Chacun pourrait alors coder une opération. Chaque implémenteur d'opération peut décider s'il veut traiter lui-même des `Literal` génériques (*i.e.* vérifier leurs types, opérer sur des types différents, caster, *etc.*) ou récupérer directement des littérales unifiées d'un type particulier. Par exemple, l'implémentation de l'opérateur `STO` prend des littérales génériques car il faut vérifier que l'identificateur est bien une expression, tandis que le type de l'autre littérale ne nous importe pas. Au contraire, les opérateurs numériques ne s'appliquent que sur des littérales concrètes du même type.

Pour facilliter ce travail, `Operation` définit également une méthode `static` appelée `apply`. C'est par elle que passent toute les demandes d'opérations, et elle va se charger du *dispatch* :

- Si l'objet `Operation` passé en argument implémente la méthode d'évaluation générique, c'est elle qui sera utilisée ;
- Sinon, nous utilisons un mécanisme de **promotion** des opérandes afin de choisir la surcharge concrète à utiliser. Pour ce faire, nous essayons de promouvoir[2.1.3] les littérales du type concret le plus spécialisé au type concret le plus général. La priorité de promotion est la suivante :
  1. `IntegerLiteral` ;
  2. `RationalLiteral` ;
  3. `RealLiteral` ;

4. `ComplexLiteral` ;
5. `ProgramLiteral` ;
6. `ExpressionLiteral`.

Au final, la surcharge appelée est celle de la littérale la plus générale. Si cette surcharge n'est pas implémentée, on tente de promouvoir en un type plus général, `etc.` Ce système permet d'implémenter des opérations très rapidement en évitant la **redondance**, ce choix était laissé à l'implémenteur.

À titre d'exemple, l'implémentation de l'opérateur + (`PlusOperation`) n'est définie que pour les `RationalLiteral` et les `ComplexLiteral` ; la première pour préserver la forme (x/y), qui traitera les `IntegerLiteral` promus, la deuxième pour traiter les autres littérales, *i.e.* `RealLiteral` et `ComplexLiteral` via le mécanisme de promotion.

On peut remarquer qu'avec ce système, additionner deux entiers semble produire un `RationalLiteral`. Ce n'est pas le cas, grâce à la **Factory** que nous détaillons en section 2.2.1.

2.1.5 Exceptions

Pour gérer les situations exceptionnelles de l'application, nous avons créé une hierarchie d'exceptions dérivées d`exception`, visible ci-après.

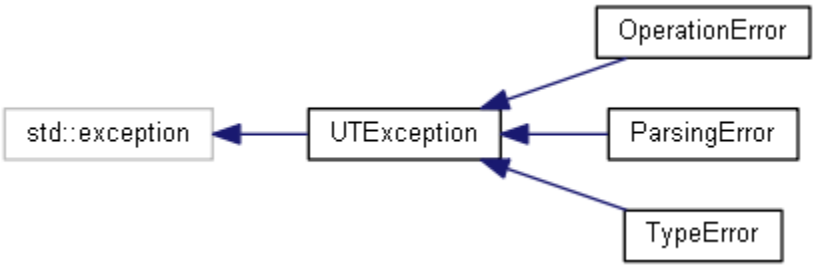

FIGURE 2 – Hiérarchie des exceptions.

— Une exception `UTException` représente une exception simple, avec un texte ;
— Une exception `OperationError` représente une exception lors de l'application d'un opération : elle référence l'objet `Operator` responsable ainsi que les arguments problématiques ;
— Une exception `ParsingError` représente une exception lors du parsing d'une chaîne de caractères quelconque : elle référence un message contextuel ainsi que la chaîne problématique ;
— Une exception `TypeError` représente une exception causée par un *type mismatch*, *i.e.* une erreur de cast ou l'utilisation d'un type d'`Operand` non-autorisé.

Ces classes implémentent une forme de **Factory** sous la forme de méthodes `clone`. En effet, l'instruction `catch` ne récupère qu'une référence : on doit copier l'objet pour pouvoir le conserver.

L'idée est de conserver toutes les exceptions levée jusqu'à l'exception finale, via une méthode `add`. La copie est effectuée lors de l'appel à cette méthode, et les exceptions sont alors stockées en cascade. Chaque exception a donc pour membre un `vector` de pointeurs sur `UTException`. On peut obtenir le détail via des appels successifs à la méthode `what`, implémentée par chaque sous-classe. C'est ce mécanisme qui permet d'obtenir le détail via le bouton **More** de l'interface graphique.

2.1.6 Pile

Pour notre calculateur, nous avons besoin d'une pile pour stocker les litterales.

Nous avons choisi de créer un *adapteur d'objet* pour implémenter la classe `Pile`. En effet, comme nous voulons itérer sur notre pile, il est plus pratique d'utiliser un vecteur de `Literal`, et non une `stack` de la librairie standard, qui ne fournit pas d'itérateur. Nous avons palié ce problème en implémentant un itérateur pour pouvoir parcourir la liste, ainsi que les fonctions standards d'une pile (*push, pop, empty* etc.) qui wrappent les méthodes du `vector` interne.

2.2 Couche Logique

2.2.1 Fabrique de Littérales

Notons, à ce stade, que nous n'avons pas rendu les constructeurs des littérales privés. Cependant, il est beaucoup plus pratique de passer par un **Singleton** `LiteralFactory` qui fédère la création de pointeurs sur `Literal`.

Les deux méthodes principales sont `makeLiteralFromString` et `makeLiteral` :

— La première permet de parser une littérale quelconque, par priorité de la plus spécialisée à la plus générale, depuis une chaîne de caractère. Cette méthode est destinée aux entrées de la ligne de commande et prend en compte les formats imposés pour les expressions, *i.e.* être entourée de guillemets.

— La deuxième est destinée aux résultats d'opérations, lorsque ce dernier est déjà un type standard ou métier. Elle est utilisée par la première méthode. C'est ici que l'on gère les opérations de *simplification* ou de *rétrogradation*. À titre d'exemple, les `RationalLiteral` sont simplifiées au maximum par division des membres par leur $PGCD$. De plus, un `ComplexLiteral` est simplifié dans son type numérique correspondant si sa partie imaginaire est nulle. Une logique similaire s'applique pour passer de `RealLiteral` à `IntegerLiteral`, et de `RationalLiteral` à `IntegerLiteral`.

C'est aussi dans cette classe que se produit le *parsing* d'une littérale programme. Deux fonctions, `makeCompositeProgram` et `makeLeafProgram` viennent gérer respectivement le cas d'un programme contenant des sous-programmes, et le cas d'un programme n'en contenant pas, *i.e.* ne contenant que des *feuilles*. Des appels récursifs permettent de gérer le cas de plusieurs niveaux de programmes imbriqués ou bien de sous-programmes adjacents.

2.2.2 Manager d'Opérateurs

La classe `OperatorManager` est la classe de référence pour les objets `Operator`. C'est cette classe qui va créer toutes les instances d'`Operator` et les associer à des objets `Operation` lors du lancement du programme. Il s'agit, comme la fabrique, d'un **Singleton**.

Les opérateurs ainsi créés sont stockés dans un conteneur, et des méthodes membres permettent de récupérer l'instance d'un opérateur correspondant à un identifieur.

La méthode importante dans cette classe est la méthode `dispatchOperation`, qui permet d'appliquer un `Operator` sur un `Arguments<shared_ptr<Literal»`. Après avoir vérifié que l'arité de l'opérateur est compatible avec le nombre d'arguments, nous passons par les cas particuliers. Ici, le seul cas particulier est celui d'un opérateur **numérique** appliqué sur **au moins** une littérale expression. Dans ce cas, la méthode membre `opExpression` est appelée. En effet, le comportement ne diffère qu'en fonction du symbole de l'opérateur (il s'agit basiquement d'une concaténation), et nous avons besoin de récupérer tous les autres opérateurs existants pour comparer leurs priorités et décider de l'application de parenthèses aux membres gauche et droit. Si aucun cas particulier n'a été traité, nous appelons simplement la méthode d'évaluation statique de la classe `Operation` décrite à la sous-section 2.1.4. La gestion des exceptions permet d'informer *finement* l'utilisateur.

2.2.3 Manager de l'application

La classe `Manager` est la pierre angulaire de notre application : son rôle est de gérer la pile, les identifieurs, l'évaluation des lignes de commande ainsi que les fonctions 'annuler' et 'rétablir' de l'application.

L'évaluation d'une ligne de commande se fait avec la méthode membre `handleOperandLine`, qui parse sommairement une ligne de commande :

- Reconnaissance des expressions ;
- Reconnaissance des programmes pour conserver les espaces ;
- Séparation sur les espaces et fabrication d'une littérale ou récupération d'une instance d'opérateur ;
- Associations des atomes libres (*i.e.* sans guillemets) à leur variable ou à l'évaluation de leur programme ;
- Encapsulation des atomes libres non-identificateurs dans un objet `ExpressionLiteral`.

Le `Manager` est responsable de la gestion de la pile. À ce titre, sa méthode `eval` traite un ensemble d'`Operand` : il empile les littérales et applique les opérateurs via `OperatorManager`.

Il est aussi respondable de la gestion des identificateurs. Nous utilisons une même liste associative (`map`) pour stocker tout les identifieurs, variables et programmes car ce sont tous des literrales. Pour acceder spécifiquement aux variables ou aux programmes, il existe des fonctions qui renvoient exclusivement l'un ou l'autre. Le Manager s'occupe également de la création, mise à jour et suppression des identifieurs, en prenant garde à ne pas utiliser des identifieurs correspondant à des opérateurs.

Le Manager s'occupe également des fonctions *annuler* et *établir* de l'application. Nous avons utilisé le *design pattern* **Memento** pour implémenter ces fonctionnalité. Pour cela nous avons créé une classe `Memento`, qui contient les éléments du Manager que nous voulons sauvegarder : la pile, les identifieurs, les options et l'historique. A chaque évaluation d'une ligne de commande, nous enregistrons un `Memento` correspondant à l'état actuel du Manager dans un vecteur de `Memento`. Une variable conserve l'indice de l'état actuel du Manager dans ce vecteur. Ainsi il suffit, pour restaurer l'état précédent du `Manager`, il suffit de récupérer le `Memento` correspondant et rétablir chaque élément du Manager à partir de celui-ci. De même, pour rétablir un état précédemment annulé. L'utilisation du vecteur nous permet d'effectuer les opérations `UNDO` et `REDO` en chaîne.

2.2.4 Parser d'expressions

Le parser d'expressions est implémenté dans la classe `ExpressionParser`. Nous n'avons rien inventé car nous avons simplement implémenté l'algorithme *Shunting Yard* de **Dijkstra** (disponible à l'adresse https://en.wikipedia.org/wiki/Shunting-yard_algorithm).

2.3 Couche Graphique

2.3.1 Organisation des élèments

Toute la couche graphique à été construite en utilisant `Qt`, une API orientée objet qui offre des composants d'interface graphique (widgets), d'accès aux données, de connexions réseaux, de gestion des fils d'exécution, d'analyse XML, etc.

L'interface graphique est construite autour de la classe principale `UTComputer`, héritant de `QMainWindow`. Elle contient les menus de l'application, ainsi qu'un widget central, le `Calculator`, qui contiendra tout le nécessaire pour entrer les commandes et lancer l'évaluation. `UTComputer` s'occupe également de la sauvegarde des éléments du calculateur et de les restaurer au démarage.

La classe `Calculator` contient la ligne de commande, le message à l'utilisateur en cas d'erreur, l'affichage de la pile et elle contient la clavier clicable sous la forme d'un objet

`MainFrame`.

Le clavier clicable contient les boutons correspondants aux chiffres, le point, entrer, supprimer, ainsi que les opérateurs symbolique. Un bouton permet d'acceder à la liste des opérateurs de fonctions. Notons que ces opérateurs sont affichés dynamiquement à partir du vecteur contenu dans le `Manager`. Ainsi, l'ajout **logique** d'un opérateur quelconque provoquera automatiquement son affichage fonctionnel **graphique**.

2.3.2 Fenêtre des paramètres

L'application posséde une fenêtre de paramètres. Celle-ci donne la possibilité de changer le nombre de ligne affichées de la pile, la présence d'un signal sonore lors de l'apparition d'un message utilisateur et la présence du clavier clickable. Ces options sont connecté, grâce a aux signals des widgets, à la classe `Calculator`, qui s'adapte directement lors de leur modification et modifie le Manager en conséquence.

Cette fenêtre offre également une interface de gestion des identifieurs, divisé en deux onglets, l'un pour les variables, l'autre pour les programmes. Toutes les informations sont récupérées du singleton Manager. La modification et l'ajout d'identificateur se fait par des fenêtres secondaires. Celles-ci ajoutent ou modifient l'identificateur dans le `Manager` puis appellent le slot de la fenêtre de paramètres lors de la validation pour déclencher la mise à jour de l'affichage.

Nous avons créé une classe spéciale `ButtonIdentifier`, héritant de `QButton`, qui contient la clé de l'identificateur à modifier ou supprimer, pour pouvoir utiliser cette identificateur dans la fenêtre ou la fonction appelée par le bouton : c'est un quelque sorte une simulation du *binding* que l'on trouve dans d'autres langages objets.

2.3.3 Evénements sur la ligne de commmande

L'élement principal de l'interface graphique est la ligne de commande. Nous avons rédéfini la fonction virtuelle `eventFilter` pour pouvoir filtrer les évenements de la `QLineEdit`. Ainsi, quand l'utilisateur utilise les touches haut et bas, nous pouvons faire défiler l'historique des commandes. Quand il utilise un opérateur **arithmetique** ou bien la touche `Enter`, on peut lancer l'évaluation de la commande. Nous avons également empéché l'utilisation des touches droite et gauche pour naviguer dans la ligne de commande (seule l'utilisation de `Backspace` est autorisée).

2.3.4 Sauvegarde

Pour que notre application sauvegarde son état à la fermeture et le restaure au lancement, nous avons utilisé les fonctionnalité de `Qt`, notamment avec l'utilisation de `QFile` et de `QDatatStream`. Il suffit donc d'écrire les élements du `Manager` dans un fichier, puis de les

restaurer à l'ouverture de l'application. Pour éviter de devoir reconstruire les objets *manuellement*, et donc d'instaurer une forme de redondance avec les objets fabriques, nous avons défini cette convention :

— Les éléments de la pile sont sauvegardés tels quels, séparés par des espaces, selon leur représentation (`toString`). Cette ligne sera ensuite évaluée directement par `handleOperandLine` afin de reconstruire la pile sans aucune logique supplémentaire.
— Les éléments identificateurs sont sauvegardés selon la représentation de leur variable ou programme, en ajoutant l'opérateur `STO`. Une fois encore, `handleOperandLine` permet de les restaurer sans logique supplémentaire.

2.3.5 Calcul et gestion des erreurs

La fonction `calculate` est celle qui s'occupe d'envoyer la ligne de commande au `Manager` pour son évaluation. Elle met ensuite à jour toute l'interface graphique et, notamment en cas d'erreur, elle affiche le résumé de l'erreur dans le `QEditLine` correspondante.

Nous avons aussi décidé d'offrir la possibilité d'ouvrir une fenêtre présentant les détails de l'erreur. Pour cela nous gardons ceux-ci dans une variable de `Calculator`.

3 Evolutivité de l'application

Nous avons, depuis la conception et pendant tout le développement, accordé une attention toute particulière à la **modularité** et **l'extensionnalité** de l'application. À ce stade, nous avons tout fait pour qu'il soit possible de modifier des composants (ajout / retrait / mise à jour) sans impacter le reste de l'application. Dans cette section, nous démontrons le potentiel évolutif d'UTComputer.

3.0.1 Couplage faible

Une des principales difficultés dans ce type d'application est de conserver un couplage très faible entre différents composants qui interagissent constamment. Nous avons cherché à réduire au minimum l'*interface publique* utilisée par les différentes classes.

Tout d'abord, les littérales sont utilisables en l'état, vierges de tout comportement. Cette abstraction maintenue au niveau métier permet de les déporter vers une autre application totalement différente, sans devoir déporter d'autre composant.

Les opérateurs sont également indépendant de leur opération : en ce moment, il est possible d'instancier de nouveaux opérateurs ou de modifier leur comportement sans avoir à modifier la classe `Operator`, qui peut être distribuée sous forme de bibliothèque sans le moindre problème.

Les objets *Operation* sont **a priori** indépendantes de l'architecture d'UTComputer. Nous entendons par cela que les prototypes et comportements par défaut définis par la classe abstraite `Operation` ne font pas référence à d'autres classes. La classe `Operation` n'est qu'une sorte d'interface que n'importe qui est libre d'étendre sans toucher au reste du code pour définir de nouveaux comportements. À ce titre, il est possible d'effectuer une opération indépendemment de tout autre composant, comme le montre l'exemple ci-après.

```
std::shared_ptr<Literal> l1= std::make_shared<IntegerLiteral>(4);
std::shared_ptr<Literal> l2 = std::make_shared<RealLiteral>(4.5);
std::shared_ptr<Operation> op = std::make_shared<PlusOperation>();
Arguments<std::shared_ptr<Literal>> args{l1, l2};
auto res = Operation::apply(op, args);
for(const auto& l : res) std::cout << l->toString() << std::endl;
```

Cet exemple n'utilise que les classes `Literal`, `Operation` et le *wrapper* `Arguments`. La sortie de l'application est `8.5`, ce qui correspond au résultat attendu.

Précisons que **notre** choix d'implémentation de certains opérateurs fait appel à des classes de la couche logique, *e.g.* `EVAL`, `WHILE`, *etc.*. Cependant, il s'agit d'un choix cantonné à l'opération qui peut ête détachée sans problème. C'est le choix de l'implémenteur dont nous avons parlé précédemment.

Enfin, nous avons accordé une attention particulière à séparer **complètement** `Qt` et la couche graphique associée du reste de l'application. En d'autres termes, il n'existe aucune référence à une classe graphique ailleurs que dans la fonction `main`. Il est tout à fait possible de travailler en ligne de commandes en utilisant la classe `Manager` et les classes avec lesquelles il collabore sans aucune modification à effectuer dans le code (excepté le lancement dans le `main`).

3.0.2 Extensions de l'application

En conséquence du couplage faible que nous avons maintenu entre les différents composants, il est possible de faire évoluer l'application très facilement :

- Il est possible d'implémenter un nouvel opérateur en une ligne dans `OperatorManager`. Il suffit de créer une nouvelle instance, de lui associer un comportement et de l'ajouter au vecteur des opérateurs. Aucune autre modification n'est nécessaire pour que l'application fonctionne en l'utilisant. (affichage dynamique **inclus**).
- Il est possible d'implémenter une nouvelle opération en héritant d'`Operation`. Aucun autre changement n'est nécessaire, et peut se faire dans un fichier séparé.
- Il est possible de modifier le comportement complet d'un opérateur en changeant un mot dans l'instanciation de l'`Operation`, *i.e.* la classe stratégie.
- Il est possible de modifier le comportement de la création de littérales uniquement en changeant le code de la `LiteralFactory`.

— Il est possible, mais *plus complexe*, d'ajouter un nouveau type de littérale. Pour que cet ajout soit fonctionnel, il faut :

1. Hériter de `Literal` ;
2. Si l'on souhaite une compatibilité avec les autres littérales, définir un opérateur de cast depuis ces littérales. Cette étape est optionnelle ;
3. Ajouter une méthode virtuelle d'évaluation dans la classe `Operation`, et une tentative de cast dans la méthode statique `apply` ;
4. Ajouter une méthode dédiée dans la `LiteralFactory`. La littérale étant le centre de l'application, et aussi le point de choix des opérations, cette étape est évident plus délicate, mais faisable assez rapidement. À noter que chacune des étapes, même incomplète, n'empêche pas l'application de compiler, ce qui rend le développement plus souple.

— Enfin, nous avons porté une attention soutenue à la *généricité* des méthodes, nous efforçant de ne revenir aux types concrets des objets métiers que lors de l'exécution d'un comportement dépendant de ce type concret. Cependant, toute l'application gère des pointeurs génériques, qui permettent de gérer l'implémentation de sous-objets métiers et de les faire fonctionner sans modification nécessaire du reste du code.

4 Améliorations possibles

Il y a sans doute beaucoup d'améliorations possibles à notre calculatrice, mais nous n'avons pas le temps de les lister ici. Le principal exemple auquel nous pensons est le suivant : actuellement, il est dommageable que certaines classes dérivées d'`Operation` fassent appel à des classes logiques telles que `Manager` ou `OperatorManager`, parce que ces opérations sont dépendantes de l'implémentation de l'application.

Une solution serait de créér une couche d'abstraction supplémentaire afin de séparer les opérateurs métiers des opérateurs indépendants de l'applications (*e.g.* `EvalOperation` *v.s.* `PlusOperation`).

A Annexes

A.0.1 Exemples avancés

Dans ce paragraphe, nous donnons un exemple de programme avancé créé sur notre calculatrice, qui permet d'effectuer des opérations un peu plus sophistiquées et modulables. À ce titre, vous trouverez trois programmes prédéfinis (`BOUCLE`, `ISPRIME` et `PRIMES`) dans l'application :

- Le programme `BOUCLE` dépile un programme *ou* un identificateur de programme, puis deux nombres i et j. Tant que $i < j$, `BOUCLE` évalue le programme dépilé, puis incrémente i. Vous l'aurez compris, ce prédicat simule une boucle. Exemple d'utilisation trivial : `1 10 [ "I" EVAL ] BOUCLE`, qui empile les entiers de 1 à 10 **exclu**.
- Le programme `ISPRIME` teste si l'entier pointé par l'identificateur `I` est un nombre premier. Si oui, il l'empile. Une logique interne de boucle est utilisée.
- Le programme `PRIMES` dépile deux nombres i et j, et empile tous les nombres premiers p tels que $i \leq p < j$.

Le programme `BOUCLE` est détaillé ci-dessous. Les autres programmes sont consultables dans l'application.

```
[ "PROG" STO "J" STO "I" STO ]
    [
        "I" EVAL "J" EVAL <
    ]
    [
        "PROG" EVAL EVAL
        "I" EVAL
        1 + "I" STO
    ]
    WHILE
]
```

Sommairement, cette fonction dépile les trois arguments nécessaires à la boucle. La condition est encapsulée dans un programme, et se lit $i < j$. En cas de succès, le deuxième programme est évalué. Il évalue deux fois l'identificateur `PROG` : une fois suffit s'il s'agissait d'un programme en dur, deux fois sont nécessaires s'il s'agissait d'un identificateur de programme encapsulé dans une littérale expression. Après exécution du programme, la valeur pointée par `I` est empilée, incrémentée, puis stockée dans `I`. On notera la différence essentielle entre `"I"` et `"I" EVAL`.

A.0.2 Instructions de compilation

Le projet ayant été développé sur *QtCreator*, il est beaucoup plus pratique de l'exécuter dans cet environnement. Il suffira pour cela d'ouvrir le ficher `UTComputer.pro` avec QTCreator, de sélectionner un kit installé sur la machine, puis de lancer successivement `Build All`

et `Run qmake` depuis de menu `Build`. Depuis ce même menu, il suffit ensuite de lancer la commande `Run`.

Des *warnings* pourraient indiquer des comparaisons entre des entiers signés et non-signés, elles sont volontaires et maîtrisées.

A.0.3 Contenu du rendu

L'archive contient :

— Un dossier `src`, qui contient les sources et le fichier projet ;
— Un dossier `html`, dont la page `index.html` mène à l'index de la documentation ;
— Un dossier `ressources`, contenant le fichier son utilisé par l'application ;
— Un dossier `report`, qui contient le présent rapport ainsi que la vidéo de présentation.

A.0.4 Easter-eggs

Si vous avez eu la bravoure de lire ce rapport jusque ici, peut-être apprécierez vous les fonctions suivantes : `PIMPMYCALC` et `ROTATE`.

Nous nous excusons platement si notre signal d'erreur vous est resté dans la tête. Voilà de quoi nous faire pardonner.