Projet LO21 - UTComputer

Introduction

L'objectif de ce projet était de développer l'application UTComputer, un calculateur scientifique permettant de faire des calculs, de stocker et de manipuler des variables et des programmes en utilisant la notation RPN (Reverse Polish Notation). 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.

Ainsi :

$$'(2 + 3) * 4' \text{ pourra s'écrire } 2\ 3 + 4 *$$

La notation post-fixée implique l'utilisation d'une pile pour stocker les résultats intermédiaires lorsque l'on évalue une formule arithmétique. Cette pile sera la zone d'affichage principale de la calculatrice. La pile est gérée de la manière suivante lorsque l'on saisit un opérande :

- Si l'opérande est une littérale, elle est empilée sur la pile
- Si l'opérande est un opérateur d'arité *n*, *n* expressions sont dépilées et le résultat du calcul est empilé.

Un des inconvénients de cette méthode est qu'un opérateur ne possède **qu'une et une seule** arité. Il faudra donc différencier l'opérateur binaire de soustraction '-' de l'opérateur unaire de négation NEG.

Table des matières

Etudes préliminaires

Avant de pouvoir commencer notre conception de l'architecture d'application nous avons dû réaliser des recherches additionnelles afin de poser clairement les problèmes que nous risquions de rencontrer ainsi que de répondre aux différentes interrogations que nous avions.

Nous avons donc fait des recherches supplémentaires sur la notation polonaise inverse ainsi que sur les opérateurs et opérations.

Etude sur les opérateurs

Une des premières étapes a été de trouver les priorités et arités des opérateurs usuels afin de pouvoir les utiliser correctement. Ci-dessous un tableau récapitulatif des opérateurs.

Opérateur	Priorité	Arité
()	9	Pas vraiment un opérateur
$	8	Unaire
NEG	7	Unaire
NOT	7	Unaire
*****	6	Binaire
/	6	Binaire
MOD	6	Binaire
DIV	6	Binaire
+	5	Binaire
-	5	Binaire
<	4	Binaire
<=	4	Binaire
>	4	Binaire
>=	4	Binaire
=	3	Binaire
!=	3	Binaire
AND	2	Binaire
OR	1	Binaire

Tableau 1- Priorité des opérateurs

Etude sur les opérations

Une autre étude que nous avons effectuée a été de, pour chaque opération *(ex : '+', '-', 'MOD', 'EX')* dresser un tableau du type de sortie de l'opération en fonction de celle des opérandes en entrée. Veuillez trouver ci-joint quelques exemples non exhaustifs de ces tableaux récapitulatifs.

Opérande 1	Opérande 2	Sortie
LittéraleEntière	LittéraleEntière	LittéraleEntière
LittéraleRelationnelle	LittéraleRelationnelle	LittéraleRelationnelle ou LittéraleEntière (si dénominateur = 1)
LittéraleEntière ou LittéraleRelationnelle ou LittéraleRéelle	LittéraleRéelle	LittéraleRéelle
LittéraleEntière ou LittéraleRelationnelle ou LittéraleRéelle (=Numérique)	LittéraleComplexe	LittéraleComplexe ou LittéraleNumérique (si imaginaire = 0)
LittéraleExpression	LittéraleExpression	LittéraleExpression
LittéraleExpression	LittéraleComplexe ou LittéraleNumérique	LittéraleExpression

Tableau 2 - Tableau des types pour l'opération '+'

Opérande 1	Opérande 2	Sortie
LittéraleEntière	LittéraleEntière	LittéraleEntière
LittéraleExpression	LittéraleExpression	LittéraleExpression
LittéraleExpression	LittéraleNumérique ou LittéraleComplexe	LittéraleExpression

Tableau 3 - Tableau des types pour l'opération 'DIV'. Provoque une erreur sur LittéraleRelationnelle, LittéraleRéelle et LittéraleComplexe

Opérande	Sortie
LittéraleNumérique ou LittéraleComplexe	LittéraleNumérique ou LittéraleComplexe
LittéraleExpression	LittéraleExpression

Tableau 4 - Tablleaux des types pour l'opération 'NEG'

Architecture de l'application (version light)

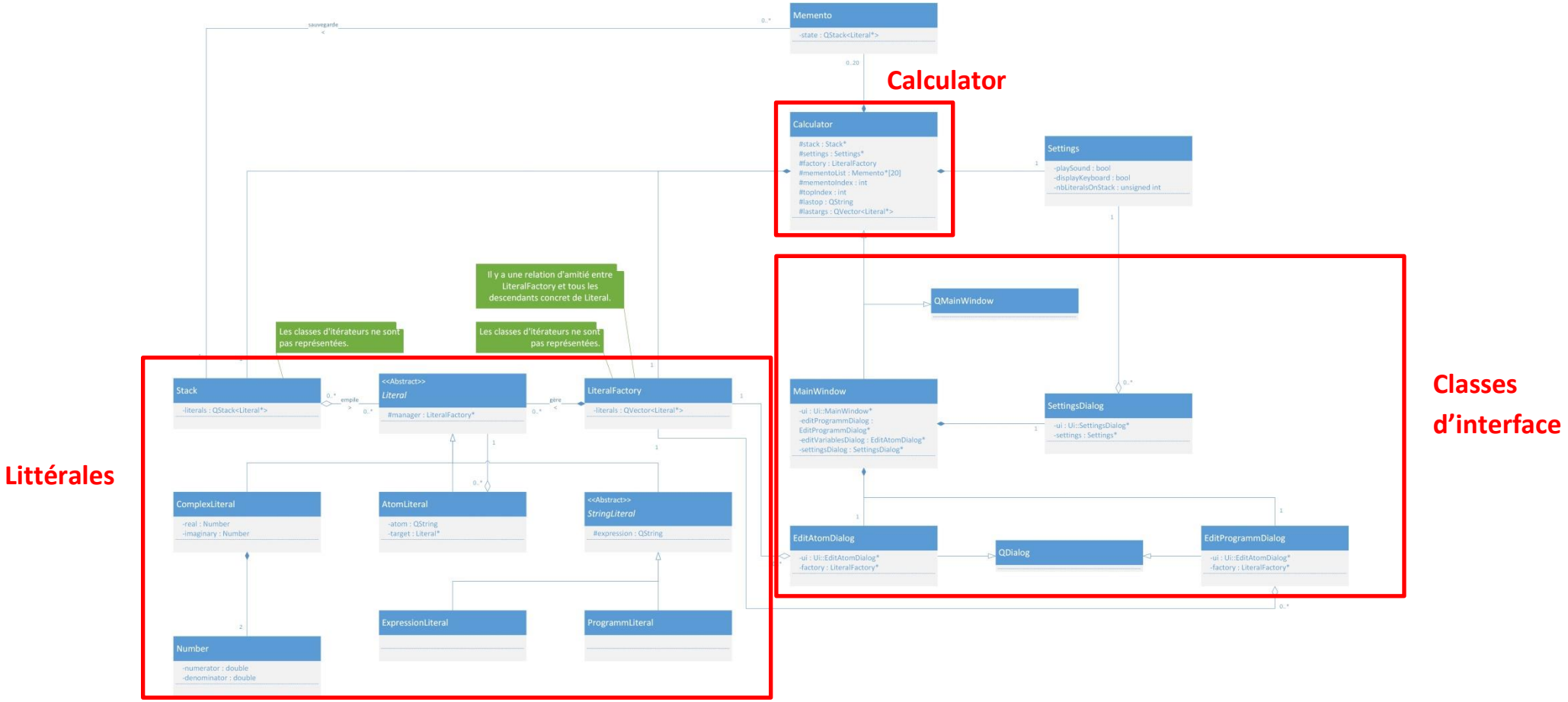
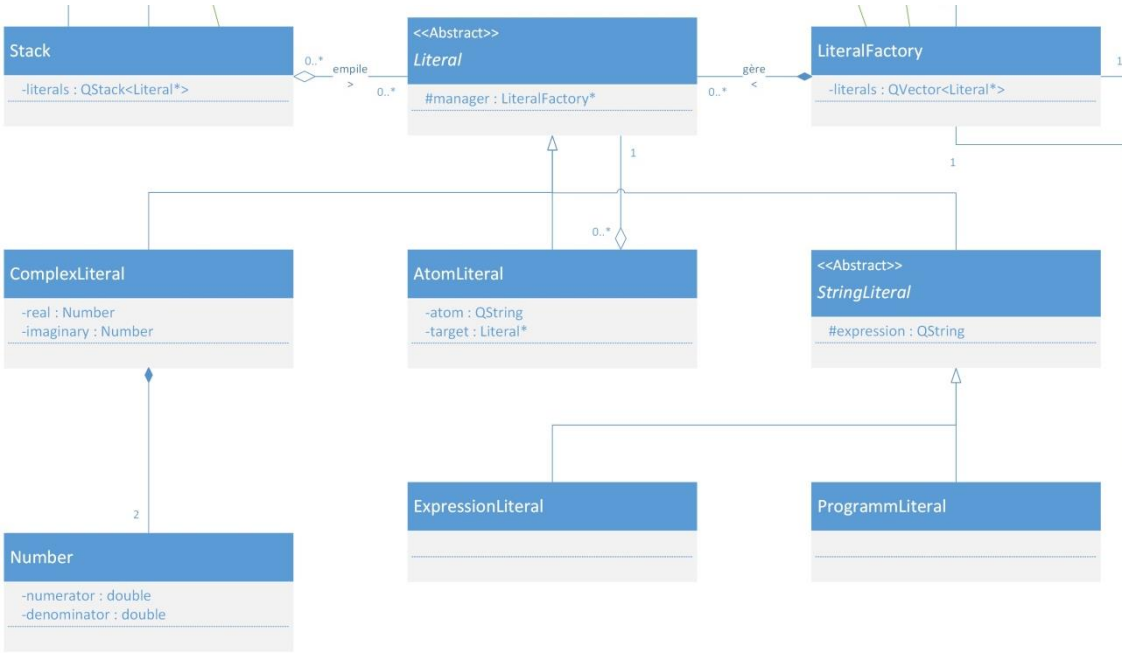


Figure 1 - UML light de UTComputer

Afin de démontrer et justifier notre architecture nous allons traiter point par point de cet UML.

Les littérales



L'application peut manier différents types de littérales.

Pour gérer les nombres (entiers, réels ou rationnels), nous avons décidé d'implémenter une classe **Number**. Le « type » du nombre importe peu car son comportement est le même et les règles de calcul sont identique. En fait, tout nombre peut être considérer comme un rationnel (avec un dénominateur éventuellement égal à 1). Cette classe n'est pas une littérale mais sert de support à l'implémentation des littérales numériques ou complexes.

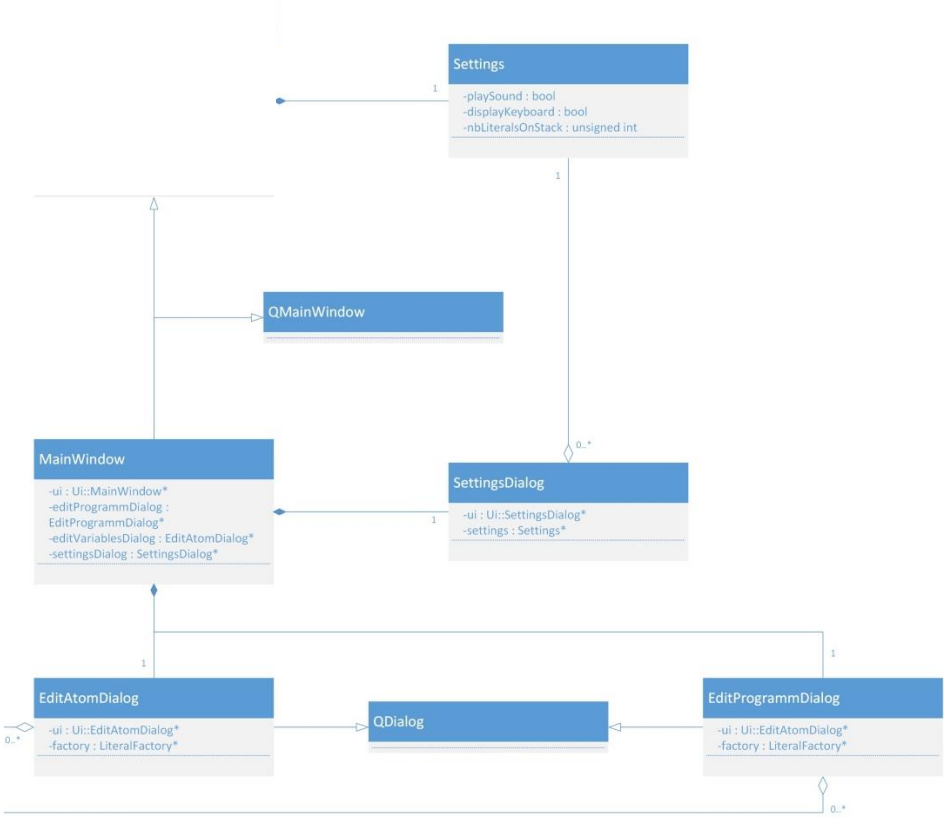
Nous avons décidé de faire hériter tous nos types de littérales : **ComplexLiteral** / **AtomLiteral** / **ExpressionLiteral** / **ProgrammLiteral** d'une classe abstraite ***Literal*** qui nous permet de regrouper les attributs et méthodes communs à toutes les classes. Les nombres réels (au sens de l'ensemble des réels) sont représentés par la classe ComplexLiteral qui permet d'instancier les nombres complexes. Comme pour la classe Number, nous tentons de simplifier au maximum : un nombre réel n'est rien d'autre qu'un nombre complexe dont la partie imaginaire est nulle. La classe AtomLiteral est la classe qui permet d'instancier des variables. Elle a un nom (identifieur) et pointe sur une autre Literal.

Un autre avantage de cette architecture est de pouvoir implémenter le **design pattern Factory**. Ce design pattern nous est utile pour isoler la création des objets de leur utilisation. La création des Literal se fait donc dans la ***LiteralFactory*** qui est constituée d'un QVector de Literal* pour garder la trace des Literal créées (composition) et qui gère la création du bon type de Literal. Plusieurs méthodes de création existent (depuis des Number, depuis un booléen ou depuis une chaine de caractères QString).

L'utilisation des ***Literal*** se fait dans la **Stack**, cette classe est constituée d'une QStack de Literal* (agrégation). L'utilité d'une QStack est d'avoir de nombreux opérateurs de pile déjà définit que nous pouvons surcharger.

Nous avons de plus utilisé pour la gestion des Literals le **design pattern Iterator** sur les classes **Stack** et **LiteralFactory** pour parcourir séquentiellement tous les éléments contenus dans ces classes (qui sont des objets agrégateurs de literals) sans exposer la structure de donnée du conteneur. Ces itérateurs sont des adaptations des itérateurs de QStack ou de QVector. Nous n'avons pas représenté ce design pattern sur l'UML pour que celui-ci reste compréhensible et aéré.

Les classes d'interface



Les différentes classes qui sont nécessaires à la construction de l'interface sont :

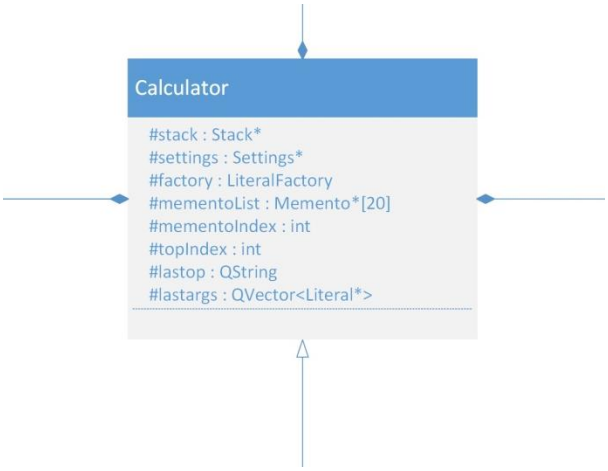
EditAtomDialog et **EditProgrammDialog** qui sont des QDialog pour créer une vue secondaire à l'application. Ces QDialog ont toutes deux un pointeur sur la LiteralFactory pour proposer à l'utilisateur une édition et une consultation des variables ou des programmes.

Settings et **SettingsDialog**, sont deux classes pour gérer les paramètres de l'application. La première regroupe les attributs qui gère l'affichage du clavier, la génération d'un son système pour l'affichage des messages et le nombre de champs que l'on affiche dans la pile. Elle s'occupe également de la sauvegarde et du chargement du contexte de l'application. La seconde classe permet l'utilisation de

Settings et de faire le lien avec MainWindow : l'utilisateur peut y modifier les paramètres de l'application. C'est aussi une QDialog.

MainWindow est la classe principale de l'interface qui hérite de QMainWindow. C'est elle qui gère EditAtom, EditProgramm et SettingsDialog. Elle est responsable de l'affichage. La gestion des calculs et de l'état global de la calculatrice se fait par la classe **Calculator** dont hérite MainWindow.

Calculator



La classe **Calculator** est responsable de la gestion globale de la calculatrice. Elle est composée d'un pointeur sur la **Stack** et d'une **LiteralFactory** pour garder un contrôle sur la création ou non de literal et de leur empilement sur la pile. Cette classe gère l'interaction entre les principaux objets de la calculatrice en assurant une cohérence et en vérifiant qu'aucune erreur ne se produit. En cas d'erreur un message est récupéré et affiché à l'attention de l'utilisateur (par la classe MainWindow).

Note : Les messages d'erreur peuvent se propager d'une classe à l'autre et cela nous permet donc de récupérer des erreurs même si celle si ne sont pas « proches » de Calculator.

Nous n'avons pas implémenté de réel Singleton sur le Calculator car un objet Calculator est construit au démarrage de l'application et ne peut être généré autre part. Nous avons cependant protégé le Calculator d'une construction par recopie et de l'opérateur « = » pour assurer un minimum d'intégrité. Un des derniers design pattern que nous avons implémenté est un **memento**.

Memento

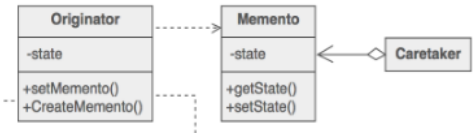


Figure 2 - Structure d'un memento

Dans notre cas, nous avons implémenté un memento pour gérer les fonctions undo/redo qui permettent de naviguer dans les états précédents de la pile. L'originator est donc la **Stack** puisque c'est l'objet dont nous voulons garder une trace. C'est l'objet **Stack** qui créer les memento. La classe

Memento est une classe « simplifiée » de la stack. Le gestionnaire du memento est la classe Calculator qui est composé d'un tableau de Memento afin de garder en mémoire différents états de la pile. L'objet Calculator gère au cours de l'exécution d'opérations l'index du tableau (en l'incrémentant ou le décrémentant en fonction des demandes de l'utilisateur.

Réflexions sur le modèle

Ce modèle ne correspond pas exactement à la première approche que nous souhaitions. Afin de rendre le code plus flexible et maintenable, nous aurions souhaité implémenter des classes d'opérateurs qui s'occupent du calcul. Bien que nous y ayons pensé au tout début, il nous a fallu du temps pour intégrer le projet et cette idée ne s'est concrétisée dans notre esprit que vers la fin du développement, lorsque nous avions le recul nécessaire sur le projet. C'est très probablement l'une des limites de notre conception.

Toutefois, tout n'est pas à jeter ! Notre modèle reste quand même assez souple. En effet, l'implémentation des programmes a été assez tardive dans le développement. Sémantiquement, les programmes sont proches des expressions, à quelques ajustements près. Au final, l'ajout des programmes n'a été l'affaire que de quelques dizaines de minutes, le temps de créer un héritage et d'ajuster quelques lignes de code en conséquence (notamment sur les opérateurs).

Une autre amélioration possible du modèle serait la création d'un héritage pour EditAtomDialog et EditProgrammDialog dont le fonctionnement est très proche. L'interface de ces classes est même identique. Ces classes ayant été implémentées à la toute fin du projet, nous avons manqué de temps pour faire un héritage propre et nous avons préféré le code tel quel, bien qu'il soit redondant.

Enfin, nous aurions aimé utiliser des tests unitaires dans notre développement. Il existe même un outil Qt pour ça. Toutefois, après plusieurs heures perdues à tenter l'utiliser avec nos classes, nous avons préféré mettre cette idée de côté et ne pas perdre plus de temps.

Dans un souci d'efficacité et de gain de temps, nous avons choisi d'utiliser Qt Designer pour réaliser notre interface.

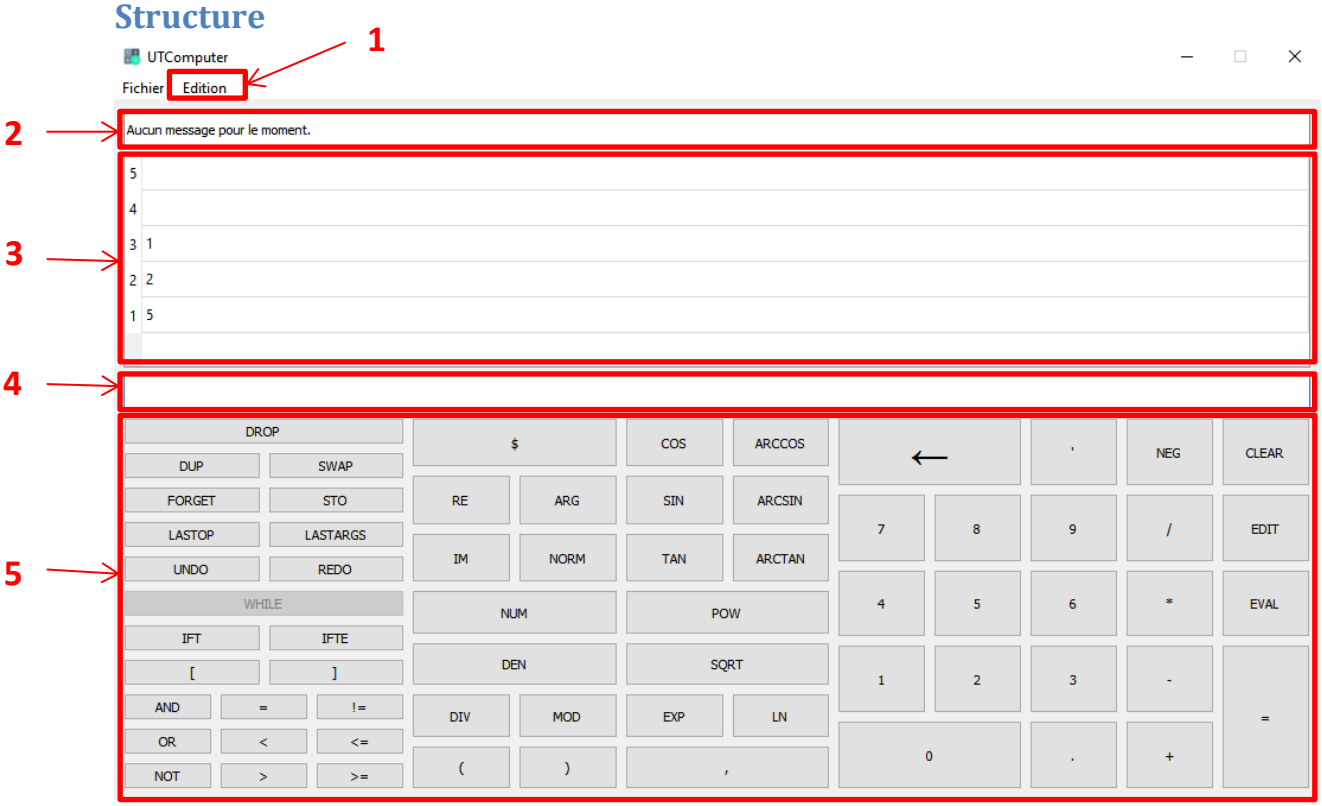
Puissance et limites du modèle

Comme expliqué ci-dessus, nous avons choisi une architecture tournée vers les Littérales pour effectuer le calcul et non pas des classes d'opérateurs. La seconde méthode (classe d'opérateur) est plus souple dans le sens ou pour ajouter un opérateur il suffit de créer une nouvelle classe. Cette méthode est donc la meilleure si on souhaite un code évolutif. Notre architecture possède cependant l'avantage de pouvoir effectuer des opérateurs et opérations complexes en croisant les types *(ex : complexes & expression)* sans erreur car ils héritent tous de la même classe mère.

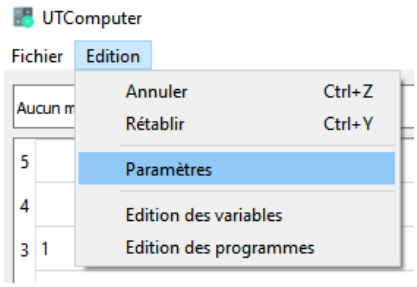
Eléments d'interface

Pour l'interface, nous avons décidé de faire quelque chose de sobre en attribuant à l'application une taille fixe et en offrant peu de possibilités de personnalisation de design à l'utilisateur.

Structure



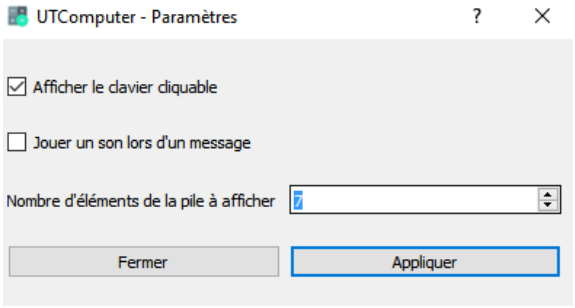
Les principaux éléments d'interface sont les suivants :



1. Menu d'édition dans lequel on peut annuler ou rétablir les actions effectuées, accéder aux paramètres, éditer les variables et les programmes.
2. Zone d'affichage à l'attention de l'utilisateur en cas d'erreur.
3. Affichage de la pile (modifiable dans les paramètres)
4. Zone de saisie
5. Clavier cliquable

Nous avons donc dû réaliser des vue secondaires de l'application qui sont dédiées à la gestion et l'édition des variables stockées dans l'application, des mini-programmes utilisateurs et des paramètres.

Paramètres



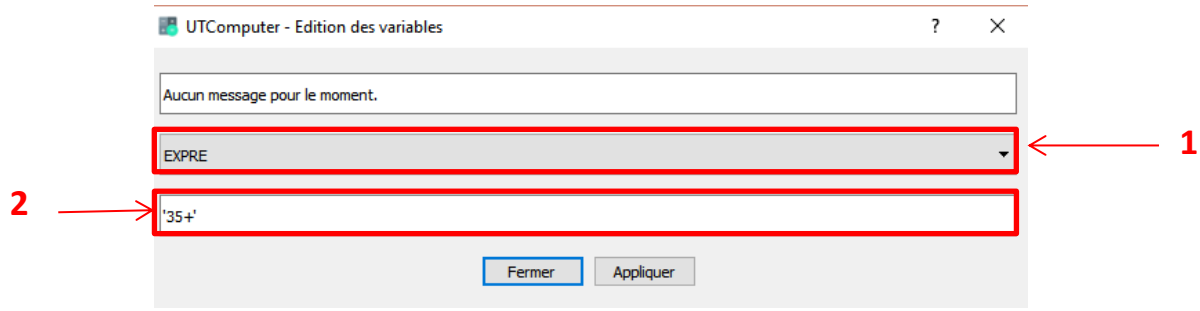
Les fonctionnalités modifiables par le biais des paramètres sont l'affichage ou non du clavier cliquable. Nous avons choisis de pouvoir cacher uniquement le clavier numérique et les opérateurs principaux (+, -, *, / ou =) mais de laisser toutes les fonctions dans un souci de d'utilisabilité. Finalement, nous pouvons aussi modifier la taille d'affichage de la pile de 1 à 10 éléments.

Edition des variables

Une autre vue nécessaire est celle de l'édition des variables. La création d'une variable est rendue possible par la fonction *STO* qui s'utilise de la manière suivante :

$$LITTERALE\ NOM\_DE\_LA\_VARIABLE\ STO$$

La fenêtre d'édition permet donc une gestion de ces variables stockées en mémoire et d'informer sur le contenue de chacune d'elle.



1. Sélection de la variable
2. Contenu de la variable (opérande ou expression)

Il est possible de modifier le contenu de la variable en lui donnant pour valeur la chaine de caractères correspondant à la littérale que l'on souhaite stocker. Par exemple, 1$2 pour stocker le complexe de partie réelle 1 et de partie entière 2.

Edition des programmes

La vue pour l'édition des programmes est une vue qui contient encore une barre de message à l'attention de l'utilisateur, un menu déroulant pour sélectionner le programme ainsi qu'un encart pour voir le contenu de ce programme. De même que pour les variables classiques, cette fenêtre permet d'édition des variables programmes.

Conclusion

Pour conclure, nous pouvons dire que ce projet nous a permis de mettre en application les connaissances acquises en LO21. Nous remarquons que l'expérience est une grande plus-value lors de la conception. En effet, nous avons passé beaucoup de temps à concevoir notre modèle qui n'est, au final, pas exempt de défauts. De plus, beaucoup d'idées de conception nous sont venues lorsque nous étions en train de développer et il est parfois fastidieux de revenir sur un code déjà établi. Dans certains cas, cela permet d'améliorer la conception et faciliter le reste du développement.