Université de Technologie de Compiègne

Projet LO21 - Comp'UT

Janvier 2021

Rapport

Groupe :
DEBUREAUX Anaïs, GRALL Thibaut, MARGERIT Paul-Edouard, MARQUIS Antoine, ZHANG Aline

Responsable LO21 : Antoine Jouglet

Table des matières

1. Indications

1.1. Objectif

L'objectif de notre projet est de développer l'application Comp'UT, 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 » dite aussi « la notation postfixe ». La notation postfixe est une méthode de notation mathématique permettant d'écrire une formule arithmétique sans utiliser de parenthèses.

1.2. Fonctionnalités implémentées

Nous avons fait le choix d'implémenter les fonctions non facultatives. Ces fonctions sont toutes fonctionnelles et visibles sur les deux claviers cliquables. Les fonctions optionnelles ne sont donc pas implémentées.

1.3. Choix d'implémentation

L'utilisation de pointeurs intelligents nous permet de gérer nos différents bloc-mémoires. Nous avons fait le choix d'utiliser cette technique de programmation qui permet de garantir que les ressources devenues inutilisées dans notre programme seront relâchées de manière automatique. Par exemple, au cours d'une opération nous pouvons simplifier une fraction en entier. Nous allons donc créer un nouvel objet de manière dynamique dans une méthode. Les pointeurs intelligents permettent d'allouer dynamiquement le nouvel objet sans avoir à supprimer l'ancien et sans avoir à se préoccuper de la libération du nouveau dès lors qu'il ne sera plus utilisé. Ici, nous l'appliquons à la gestion de la mémoire.

2. Description de l'architecture

2.1. Diagramme simplifié

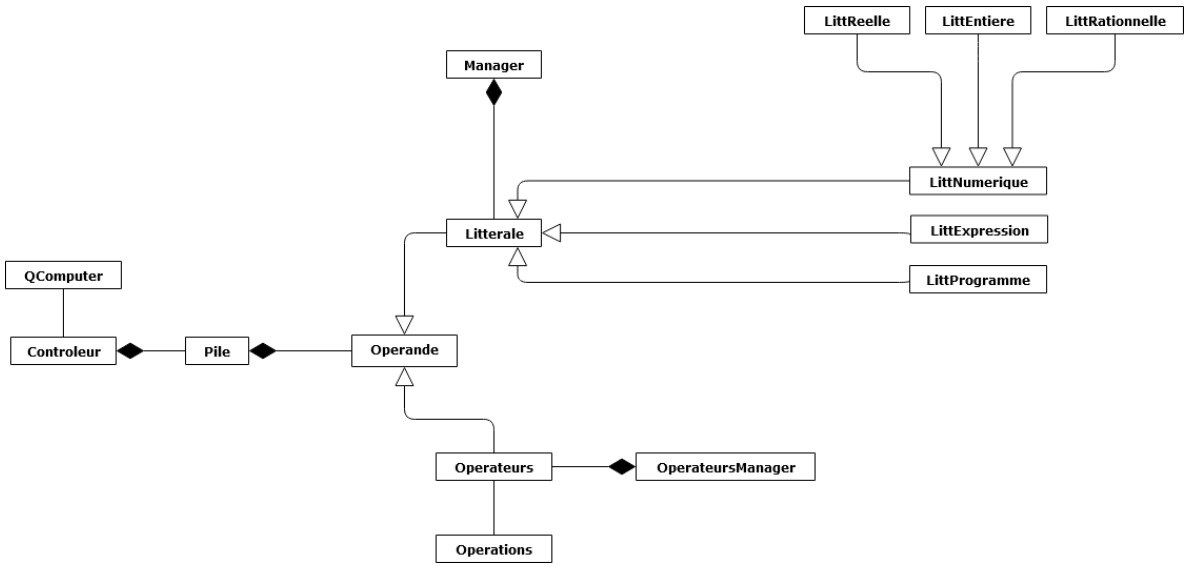

Figure 1 : UML simplifié de l'architecture

2.2. Opérandes

La classe `Operande` permet à la pile de gérer à la fois les littérales et les opérateurs qui héritent de cette classe. Cette classe mère facilite par exemple l'utilisation de l'opérateur `STO` sur les littérales expressions et littérales programmes. La classe `Operande` définit une méthode virtuelle pure `toString` qui est utilisée pour tous les affichages.

2.3. Les littérales

Les littérales désignent tous les objets manipulés par la pile et sur lesquels nous pouvons réaliser des opérations, ce sont tout simplement les objets manipulables par le calculateur. La classe `Litterale` est une classe abstraite qui regroupe les principaux types de littérales: les programmes, les expressions ainsi que les littérales numériques.

Littérales Numériques:

Les littérales numériques regroupent tous les objets désignant des nombres, sur lesquels nous pouvons réaliser des opérations telles que l'addition ou la multiplication. La classe `LittNumerique` est abstraite et est la classe mère de `LittReelle`, `LittEntiere` et `LittRationnelle` qui désignent respectivement les nombres "réels" enregistrés sous forme de double, les nombres entiers et les nombres rationnelles s'écrivant comme des fractions. Les opérateurs sont définis comme des méthodes virtuelles pures dans la classe `LittNumerique`, ces méthodes doivent donc être redéfinies par toutes les classes filles. En revanche, les opérateurs logiques sont définis directement dans cette classe. En effet, un accesseur `getVal()` permet de renvoyer la valeur d'un réel, d'un entier ou d'une fraction sous forme d'un double. Nous pouvons donc comparer les valeurs stockées par chaque littérale et renvoyer un objet de type `LittEntiere.` Ainsi l'attribut `val` vaut 1 si le test est vrai, 0 sinon. La classe `LittNumerique` contient aussi une méthode virtuelle pure `toString()` permettant de renvoyer une littérale sous sa forme affichable, dans une chaîne de caractères.

La classe `LittReelle` est constituée d'un attribut `val` qui contient la valeur du réelle représenté et qui est de type double ainsi que la redéfinition de tous les opérateurs (+-*/). La méthode `toString()` renvoie la littérale sous la forme d'une chaîne de caractères ne contenant aucun zéro. Par exemple, le réel 0.870 est représenté par ".87" et -0.38 par "-.38". Le traitement sur l'affichage se passe sur la chaîne de caractère : nous cherchons s'il y a des zéros en fin de chaîne et s'il y en a un en première ou deuxième position. Nous les supprimons s'il y en a.

La classe `LittEntiere` est constituée d'un attribut `val` qui contient la valeur entière de type int et redéfinie les opérateurs. Cette classe est très proche de celle décrite précédemment à la différence que l'affichage est encore plus simple puisqu'il suffit de convertir l'attribut en chaîne de caractère.

La classe `LittRationnelle` est constituée de deux attributs désignant le dénominateur et le numérateur définis par des variables de types int. Une méthode de simplification permet de simplifier la fraction lors des calculs en s'appuyant sur le calcul du PGCD du numérateur et du dénominateur. Cette classe redéfinit elle aussi les opérateurs numériques de base. La méthode `toString()` permet d'afficher la fraction sous la forme "numerateur/denominateur".

La redéfinition des opérateurs est la partie la plus complexe de cette partie. Les méthodes sont construites de telle manière à prendre une référence de `LittNumerique` en paramètre et donne un pointeur intelligent de `LittNumerique` en sortie. À l'aide de la fonction `dynamic_cast` et de pointeurs, nous testons le type du paramètre. L'opération est ainsi définie quel que soit le type donné en paramètre (qu'il soit `LittReelle`, `LittEntiere` ou `LittRationnelle`). Ensuite, l'opération est réalisée et l'objet à placer en sortie est construit. La fabrique de littérales permet de convertir le résultat en `LittEntiere` si le dénominateur de la littérale rationnelle vaut 1 ou si la partie après la virgule de la littérale réelle vaut 0. Les opérations commutatives comme l'addition et la multiplication sont définies une seule fois entre chaque type. Par exemple, si nous avons définie l'addition entre les `LittEntiere` et les `LittRationnelle` alors nous ne

redéfinissons pas l'addition entre les `LittRationnelle` et les `LittEntiere` mais nous échangeons seulement l'ordre des paramètres pour appeler la méthode déjà définie. Cela permet d'éviter les bugs et les incohérences. Notons aussi que si une opération n'a pas pu être réalisée, celle-ci renvoie un pointeur null.

Comme dit au début, nous avons choisi de ne pas implémenter les fonctions facultatives. En ce qui concerne les opérations facultatives, celles-ci seraient très simples à ajouter. Il suffirait de les définir dans la classe `LittNumerique` en utilisant la méthode `getVal()`. Suivant l'opération, nous aurions pu la redéfinir pour les littérales rationnelles.

Littérales Expressions:

Les littérales expressions sont des chaînes de caractères qui servent à identifier les atomes. La classe `LittExpression` est donc constituée d'un attribut `val` contenant la valeur sous forme d'une chaîne de caractères. Les méthodes associées sont tout d'abord, la méthode `getParam()` permettant de renvoyer la chaîne de caractères et la méthode `toString()` renvoyant cette même chaîne de caractères mais entre quotes cette fois. Ensuite, la méthode `sto` qui permet d'ajouter un atome qui sera identifié par cette expression et dont la valeur sera la littérale en paramètre et la méthode `forget()` qui permet de détruire l'atome identifié par l'expression. Enfin, la fonction `eval()` permet d'évaluer l'atome associé à cette expression via le manager.

Littérales Programmes:

Les littérales programmes sont des chaînes de caractères commençant et terminant par des crochets ("[ ]") représentant un programme. La classe `LittProgramme` est donc constituée d'un attribut `val` contenant le programme sous forme d'une chaîne de caractères. Les méthodes associées sont tout d'abord, la méthode `getParam()` et la méthode `toString()` permettant de renvoyer le programme (toujours sous forme d'une chaîne de caractères). Ensuite, la fonction `eval()` permet d'évaluer le programme.

2.4. Fabrique de Litterales

La classe `FabriqueLitterale` est une implémentation du *Design Pattern Factory Method.* Étant donné que la fabrique est unique, elle constitue également un singleton. Cette classe permet de construire des littérales dont nous ne connaissons pas exactement le type. Les méthodes qu'elle contient prennent un ou plusieurs paramètres et renvoient un pointeur intelligent vers une un objet `Litterale`. Concrètement cette classe permet de convertir des littérales, par exemple si nous créons une littérale à un paramètre et dont le paramètre est un type double, nous souhaitons sans doute créer une littérale réelle, mais si la partie entière de cette valeur est égale à la valeur du réel, alors nous allons automatiquement créer une littérale entière. Cela fonctionne de la même manière avec les littérales rationnelles où nous faisons intervenir la simplification pour voir si l'on peut la convertir. La fabrique est donc utilisée à chaque fois que l'on veut créer une littérale et par conséquent elle est présente dans chaque redéfinition des opérateurs pour générer la littérale résultant du calcul.

2.5. Opérateurs

Un objet `Operateur` est une `Operande`, au même niveau qu'une `Litterale`. Une instance d'un objet `Operateur` peut représenter indifféremment un opérateur numérique, logique, de manipulation de la pile, etc. Nous attribuons à chaque opérateur une chaîne de caractère (son symbole), une arité et un pointeur sur une opération. Nous définissons cette opération comme un comportement de l'opérateur : en considérant les opérations comme plusieurs classes qui spécifient la classe Operation, celles-ci ne diffèrent que d'un point de vue comportemental. C'est la raison pour laquelle nous implémentons le *Design Pattern Strategy* que nous détaillons pour les Opérations et que nous précisons également dans l'évolutivité de l'application.

2.6. Opérations

Nous constituons ce modèle de plusieurs classes (les différentes opérations) héritant de la classe abstraite `Operation` et qui accèdent à une fonctionnalité de cette classe : la méthode `eval()` qui utilise une stratégie abstraite. Pour changer l'algorithme (le comportement sur des opérandes), il suffit à la classe `Operation` de changer son objet. Ici le changement d'objet `Operation` est fonction du changement d'objet `Operateur`. Cette dépendance se fait par la classe `OperateursManager`.

2.7. Manager d'Opérateurs

`OperateurManager` est un singleton qui est chargé de la création des opérateurs, de leurs accès, et de la direction de ces opérateurs vers leur opération associée.

Pour la création, les instances d'`Operateur` sont créées une seule fois et placées dans un conteneur `operateurs` (un vecteur de pointeurs intelligents sur des opérateurs).

Concernant les accès, la méthode `getOperateur` de la classe `OperateurManager` permet la recherche d'un opérateur défini dont le symbole est le même que la chaîne passée en argument. Ce singleton permet également de vérifier si une expression est un opérateur ou non à l'aide de la méthode `isOperateur`, appelée dans le parsage de la chaîne de caractère entrée en ligne de commande.

2.8. Données

Pour la manipulation des données et particulièrement pour la gestion des opérations, nous implémentons le *Design Pattern Adapter*. La classe `Donnees` est un adaptateur de classe public (*wrapper*) de Vector utilisant un Template. Nous manipulons ce conteneur pour

effectuer les opérations entre les différentes littérales. L'utilisation d'un vecteur d'opérandes nous facilite l'application des opérateurs tel que `SWAP` : pour intervertir les deux derniers éléments de la pile, nous renvoyons un vecteur contenant les mêmes opérandes mais en inversant leur emplacement à l'aide des méthodes `back()` et `front()`. Nous utilisons ces deux accesseurs de la classe vector dans chacune des opérations implémentées.

Nous spécialisons cet adaptateur de vector pour les pointeurs intelligents que nous utilisons pour des raisons que nous spécifions dans nos choix d'implémentation. Afin de vérifier le type des pointeurs intelligents nous utilisons la fonction `dynamic_pointer_cast`. Cette fonction appliquée aux shared_ptr remplace le mot clef `dynamic_cast` et s'utilise de manière semblable (sauf que le paramètre template est B, et non pas shared_ptr<B>). Cette fonction retourne un pointeur ne pointant sur rien si la conversion n'a pu avoir lieu. C'est en testant la valeur de retour que nous déterminons si la conversion s'effectue. Nous pouvons, par la suite, décider dans `Controleur` si les opérations sont applicables. Le cas échéant, nous renvoyons une exception de type `ProjetException`.

2.9. Pile

La pile est le conteneur de base de notre calculateur RPN, elle permet de stocker les littérales avant qu'elles soient évaluées par un opérateur. Cette classe contient un conteneur comme attribut majeur. Nous avons fait le choix d'utiliser un vector de la **STL** qui possède notamment la méthode `push_back()` pour ajouter un élément à la pile ainsi qu'un iterator. C'est ce dernier point qui nous a fait choisir un vector plutôt qu'une stack. En effet, il est très important de pouvoir itérer sur les éléments de la pile pour pouvoir afficher celle-ci dans l'interface.
La classe `Pile` implémente ensuite plusieurs méthodes dont le nom est familier aux méthodes traditionnellement utilisées dans ces structures de données notamment `push()`, `pop()` ou encore `drop()`. De plus, une méthode `clear()` permet de supprimer tout le contenu de la pile.

2.10. Contrôleur

La classe `Controleur` est le cerveau du calculateur. Ce contrôleur permet de lier tous les éléments, de gérer la chaîne de caractère entrée en ligne de commande, de gérer la pile ainsi que les interactions entre les opérateurs, les littérales et la pile. Cette classe est implémentée sous la forme de singleton étant donné que l'élément majeur qu'elle manipule (la pile) ne peut exister que dans un seul exemplaire.

La classe `Controleur` contient donc un attribut de type `Pile` qui constitue LA pile du calculateur. Elle contient aussi une méthode `exec()` qui permet d'exécuter la commande placer dans une chaine de caractère dans le clavier de la calculatrice.

Cette méthode `exec()` fait notamment appel à une fonction qui parse la chaîne de caractère. Le parsage s'appuie à la fois sur la présence des programmes dont le début est signalé par le caractère '[' et sur les espaces. Un programme (et les éventuels sous

programmes qu'il contient) est vu comme un élément à part entière. Si un programme n'est pas de fin, cela provoque une erreur. Cette fonction permet donc de séparer les différents éléments placées en paramètre, elle ressort un vecteur de chaîne de caractère.

Ensuite, la méthode `exec()` fait appel à différentes fonctions pour identifier le type de chaque élément. Nous cherchons notamment à savoir s'il s'agit d'une littérale numérique, puis nous cherchons à identifier s'il s'agit d'un opérateur, puis d'une littérale expression et enfin d'une littérale atome. Si une chaîne n'est identifiée à rien alors une exception est provoquée.

Dans le cas où il s'agit d'une littérale, celle-ci est construite puis empilée. S'il s'agit d'une littérale atome, celle-ci est évaluée. Si la chaîne a été identifiée à un opérateur alors, en fonction de l'arité, nous dépilons le bon nombre de littérales et nous les mettons dans un vecteur pour évaluer l'opération. L'opération renvoie à son tour un vecteur dont les composantes sont empilées.

La méthode `exec()` de la classe `Controleur` est aussi appelée lors de l'évaluation d'un programme. En effet, un programme étant stocké comme une chaîne de caractère à part entière, il est évalué comme n'importe quelle autre commande qui aurait été mise en ligne dans la calculatrice

2.11. Manager d'atomes

La classe `Manager` permet de gérer les atomes grâce à son attribut `atomeManager`, la map des atomes, vecteur de littérales chacune identifiée par une chaîne de caractères correspondant à une littérale expression. Cette classe reflète la manière dont nous avons abordé les atomes. En effet, ceux ci ne sont pas des littérales mais ils sont simplement représentés par une classe extérieure, qui associe in identifiant (une chaine de caractère) à une littérale.
Les méthodes de cette classe sont : `AddAtome()` permettant d'ajouter un atome à partir d'une littérale expression (identifiant l'atome) et d'une littérale (contenue de l'atome) en paramètres. `ForgetAtome()` permettant de supprimer un atome grâce à la littérale expression (chaîne de caractères) qui l'identifie. `isAtome()` qui renvoie un booléen vrai si la chaîne de caractères en paramètre identifie bien un atome existant. Et enfin, `GetAtome()` qui permet de renvoyer un pointeur intelligent vers la littérale, c'est à dire le contenue de l'atome correspondant à l'identificateur (chaîne de caractères) en paramètre.

2.12. Design Patterns

Afin d'améliorer l'utilisabilité de notre application, nous avons implémenté les patrons de conception *Strategy* pour les opérations, *Adapter* pour les données et *Factory Method* pour la fabrique de littérales.

3. Evolutivité de l'application

Les choix que nous avons fait sur la conception de l'architecture font que l'application dispose d'une évolutivité importante. Nous avons particulièrement pris en compte la modularité et l'extensibilité. extensionnalité

Tout au long du projet nous avons essayé d'avoir le couplage le plus faible possible. Ainsi la classe la classe `LittNumerique` peut fonctionner toute seule, nous n'avons pas besoin de l'application complète pour manipuler des littérales et les additionner entre elles par exemple. De plus, nous avons développé séparément la partie "backend" et l'interface. Cela fait que notre application peut fonctionner en ligne de commande sans l'interface Qt (bien sûr de manière extrêmement réduite). Il y a cependant certaines classes qui nécessitent l'appel à d'autres classes, notamment des singletons tels que le contrôleur qui est appelé dans la méthode eval pour exécuter un programme.
Ce couplage relativement faible rend les mises à jour et les corrections facilement réalisables.

Ensuite, notre application peut être étendue très facilement en ajoutant de nouveaux opérateurs ou en ajoutant un type de littérale.

Développons le cas où l'on souhaiterait ajouter un type de littéral représentant les nombres complexes. Nous pourrions créer 2 nouvelles classes, une représentant les littérales numériques complexes (à 2 paramètres) et une représentant les autres littérales. La classe des littérales complexes serait alors une composition de 2 littérales numériques simples. Une représentant la partie réelle et une représentant la partie imaginaire. Cette classe pourrait redéfinir ses opérateurs + - / etc en réutilisant simplement ceux déjà redéfinis pour les autres littérales numériques. Aucune modification sur la structure globale ou sur d'autres fichiers ne serait nécessaire.

De plus, l'utilisation du *Design Pattern Strategy* permet facilement l'extension des opérations dans notre application. En effet, intégrer un nouvel algorithme est très simple : il suffit de définir une nouvelle classe concrète qui dérive de `Operation`. L'opération ajoutée, elle devra s'associer à un ou plusieurs opérateurs et dépendra donc d'une arité et d'un symbole. Il suffira ensuite d'ajouter un opérateur dans le conteneur géré par `OperateurManager`. Le contrôleur faisant appel à la stratégie `eval()` après avoir identifié un opérateur, les appels restent les mêmes. Ce seront donc les deux seules modifications à réaliser pour l'ajout d'une opération.

Cela montre bien l'évolutivité de notre application. En revanche, la moindre modification dans la structure globale du projet serait extrêmement compliquée car elle nécessiterait de très nombreuses modifications.

4. Eléments d'interface

La calculatrice est divisée en quatre onglets. Nous avons donc opté pour l'utilisation d'un `QTabWidget` comprenant :

- la vue principale : affichage de l'état du calculateur, d'une barre de commande et de deux claviers cliquables à la souris ;
- trois vues secondaires :

- vue des variables dédiée à la gestion et l'édition des variables stockées dans l'application ;
- vue des programmes dédiée à la gestion et l'édition des (mini-)programmes utilisateurs stockés dans l'application ;
- vue des paramètres dédiée à l'édition des paramètres du calculateur.

4.1. Sauvegarde des données

Nous avons utilisé le système de gestion de base de données SQLite pour sauvegarder les différentes informations du programme. La base de données est créée automatiquement par le programme. Une classe est entièrement dédiée à la persistance des données : la classe `ConnexionBaseDeDonnees`.
Elle sert :

- créer la base de données ou l'ouvrir si elle existe déjà ;
- à créer les différentes tables ;
- à fermer la base de données au moment de la fermeture de la calculatrice.

4.2. Vue principale

Pour la vue principale, nous avons décidé de séparer la calculatrice en 3 classes : `ClavierVariables` qui s'occupe des variables et des programmes créés par l'utilisateur, `ClavierNumeriques` pour le pavé numérique et les opérateurs les plus utilisés, et enfin `FenetreCalculatrice`, un singleton qui regroupe le tout en affichant l'état du calculateur et la barre de commande.

Les deux premières classes ont un constructeur qui construit un clavier contenant des `QPushButton` pour afficher les opérateurs et les variables. Nous avons choisi de les regrouper dans un `QGroupBox` pour avoir l'esthétique d'une calculatrice. Ces classes contiennent également les fonctions `minimize1()` et `minimize2()` qui permettent de réduire les claviers grâce à un `QCheckBox`.
C'est dans la classe `ClavierVariables` que sont gérés l'ajout d'un bouton variable ou programme lorsque l'utilisateur exécute une ligne d'opérandes

La classe `FenetreCalculatrice` créée une fenêtre composée de:

- un `QLineEdit` pour afficher les messages pour l'utilisateur
- un `QTableWidget` qui affiche l'état de la Pile
- un `QLineEdit` correspondant à la barre de commande
- un `ClavierNumerique`
- un `ClavierVariable`

Nous avons réimplémenté la fonction `eventFilter()`, qui filtre ainsi les touches du clavier pressées par l'utilisateur, ce qui permet d'évaluer les lignes d'opérandes avec les touches *, - , / et +.

Cette classe définit les slots connectés aux boutons des claviers numérique et variable, `clickChiffre()`, `clickOperateur()` et `clickClear()`. Les fonctions `getNextCommande()` et `refresh()` servent respectivement à transmettre la ligne de commande au contrôleur, et à mettre à jour l'affichage de la pile.

4.3. Vue des variables

La classe `FenetreVariables` s'occupe d'afficher les variables contenues dans la base de données. Cette classe est un singleton : cela garantit l'unicité d'une instance pour cette classe tout en fournissant un point d'accès global à cette instance.

Cette fenêtre est composée de :

- un `QLineEdit` responsable de la communication avec l'utilisateur ;
- un `QLineEdit` permettant à l'utilisateur de saisir des informations ;
- trois boutons : Modifier, Valider et Supprimer ;
- un `QTableWidget` qui a deux colonnes : **Nom** et **Valeur** de la variables.

Cette fenêtre permet de modifier et de supprimer des variables (l'ajout d'une variable se fait depuis la fenêtre principale).

Étapes pour modifier une variable :

- entrer le nom de la variable dans le champ prévu à cet effet ;
- l'existence de la variable dans la base de données est vérifiée via des requêtes SQL;
- si la variable n'existe pas, un message d'erreur sera affiché ;
- si la variable existe, le bouton Modifier devient inutilisable et le bouton Valider devient utilisable ;
- il suffit ensuite de rentrer la nouvelle valeur de variable dans le champ et de valider ;
- la variable sera modifiée dans la base de données ainsi que dans le Manager.

Étapes pour supprimer une variable :

- entrer le nom de la variable dans le champ prévu à cet effet et appuyer sur le Bouton Supprimer ;
- l'existence de la variable dans la base de données est vérifiée via des requêtes SQL;
- si la variable n'existe pas, un message d'erreur sera affiché ;
- si la variable existe, elle sera bel et bien supprimée de la base de données et du Manager.

Aucune vérification du type de donnée entrée n'est réalisée. Cela pourrait être une amélioration possible de la calculatrice.

4.4. Vue des programmes

Le fonctionnement est similaire à celui de la `FenetreVariables`.

4.5. Vue des paramètres

La classe `FenetreParametres` s'occupe d'afficher les paramètres de l'application. Cette fenêtre est composée d'un champ contenant le nom de l'utilisateur et d'un autre champ permettant de modifier le nombre de lignes de la pile à afficher dans la fenêtre principale.

4.6. Modifications possibles

Au lieu d'implémenter des singletons pour les différentes fenêtres, des fonctions static auraient suffit. Cependant, un singleton assure l'unicité d'une instance pour ces classes, ce qui peut s'avérer utile dans notre cas.

De plus, nous pouvons retrouver de la syntaxe SQL autre part que dans la classe `ConnexionBaseDeDonnees`. Cela signifie que les autres classes "savent" qu'une base de données est utilisée. Il aurait été préférable de ne pas avoir à se soucier du type de sauvegarde de données dans toutes les autres classes et de tout gérer directement via `ConnexionBaseDeDonnees.`

5. Organisation

L'organisation est primordiale dans un projet de groupe, surtout dans un contexte de travail à distance qui rend les échanges très compliqués et multiplie les incompréhensions. Celle-ci n'a donc pas été la partie la plus simple du projet.

Dans un premier temps nous avons pris connaissance du sujet personnellement et nous avons engagé les premières discussions pour imaginer une structure très globale de l'application.

Nous nous sommes ensuite rapidement divisés en deux sous-groupes : le premier chargé de l'interface et le deuxième de la partie backend.

Chaque groupe a ensuite réfléchi longuement à la structure de sa partie en concevant un diagramme UML pour exposer au mieux ses idées.

Dans un troisième temps, chaque sous-groupe s'est réparti le travail de manière individuelle. Nous avons chacun pu approfondir la réflexion sur nos parties et réaliser le code. Tout au long de ses parties nous avons essayé de maintenir les interactions pour comprendre au mieux la vision de chacun sur le projet.

Une date a été fixée pour fusionner les différents éléments de la partie backend, puis quelques jours après nous avons pu l'intégrer à l'interface. Ces parties n'ont pas été simples puisque nous avons rencontré de nombreuses erreurs de compilation dues à quelques incompréhensions mineures mais nécessitant quelques modifications.

De plus, bien que git soit le meilleur outil pour fusionner différentes parties d'un code, nous avons eu de grosses difficultés à l'utiliser car son utilisation ne nous était pas familière.

5.1. Planning

19 - 22 octobre	Prise de connaissance du sujet et réflexion individuelle
22 octobre	Réunion globale pour définir la structure globale et planifier l'organisation globale
mi novembre - début décembre	Conception de l'architecture
7 décembre	Réunion globale
début décembre - 20 décembre	Développement du code de l'application
20 décembre	Fusion de la partie backend
20 - 27 décembre	Corrections des bugs restants
27 - 28 décembre	Fusion de l'interface et de la partie backend
29 décembre	Application fonctionnelle et début de l'écriture du rapport
29 décembre - 4 janvier	Relecture, corrections et ajout de quelques commentaires sur les parties non commentées Finalisation du rapport et confection de la video

5.2. Répartition des tâches

Nous avons détaillé la contribution personnelle de chacun des membres du groupe sur les différents livrables :

Anaïs :

- Création des classes `Donnees`, `Projet Exception`, `Operateur`, `Operande`, `OperateursManager`, `Operation`.
- Liaison de ces classes avec `Litterale` et `Manager`, et aide fusion du code backend et frontend.
- Création d'un diagramme UML (seule la version très simplifiée figure ici).
- Recherches sur les *Design Pattern Factory, Strategy*, *Adapter* puis implémentation de ces deux derniers.

Thibaut :

- Création des classes `LittNumerique`, `LittEntiere`, `LittRationnelle`, `LittReelle` et `FabriqueLitterale`.
- Création de la classe `Contrôleur` et `Pile`, fusion des différentes parties backend et debug ;
- Aide fusion du code backend et frontend.

Paul-Edouard :

- création des classes `FenetreVariables`, `FenetreProgrammes`, `FenetreParametres`, `QComputer` et `ConnexionBaseDeDonnees` ;
- liaison du code backend et frontend.

Antoine :

- Création des classes `LittExpression`, `LittProgramme`, `Manager`.
- Aide à la création de la classe `Contrôleur`, fusion des différentes parties backend et debug ;
- Aide fusion du code backend et frontend.

Aline :

- création des classes `ClavierNumerique`, `ClavierVariables` et `FenetreCalculatrice`
- aide liaison du code frontend et backend

Nous avons reporté individuellement le nombre d'heures de travail consacrées au projet :

- **Anaïs** : entre 40 et 50 ;
- **Thibaut** : entre 40 et 50 ;
- **Paul-Edouard** : entre 40 et 50 ;
- **Antoine** : entre 40 et 50 ;
- **Aline** : 40.

Nous estimons que la répartition des tâches a été équitable. De ce fait, le pourcentage de contribution au projet de chaque membre s'approche des 20%.

Conclusion

Nous avons fait le choix d'une application entièrement fonctionnelle et évolutive avec l'implémentation de plusieurs Design Pattern ce qui nous a amené à ne pas implémenter les fonctionnalités optionnelles. Cependant, toutes les fonctionnalités nécessaires ont été implémentées et sont fonctionnelles. De plus, nous avons une vision très précise de comment nous pouvons étendre les fonctionnalités de l'application.

L'organisation au sein du groupe n'était pas simple. Le contexte sanitaire nous a contraints à gérer ce projet à distance tout le long du semestre.

En raison des contraintes de temps, il était difficile de gérer la séparation des tâches de manière équivalente en parallèle de la compréhension du sujet. Nous avons eu une première expérience de ces problèmes auxquels nous serons confrontés dans le monde

professionnel. Nous avons également été amenés à nous approprier github pour gérer nos codes.

Enfin, ce projet nous a permis de réaliser la complexité d'une architecture numérique tant dans la conception que dans le développement.