Université de Technologie de Compiègne
LO21, A20

Rapport Projet LO21

BRANLY Stephane, CHAHM Saad, DAVIET Paul,
GIRAMELLI Manon, POUGHET Louise

Janvier 2021

Sommaire

Première partie

Introduction

Au cours de ce semestre et dans le cadre de l'UV LO21, nous avons réalisé une application en C++ intitulée COMP'UT. Ce rapport nous permettra de décrire brièvement les fonctionnalités de notre application, d'expliciter le choix de l'architecture, l'implémentation de chacune des classes ainsi que de décrire le planning établi et la contribution personnelle des membres du groupe.

Notre application est une calculatrice utilisant la notation RPN (Reverse Polish Notation). Elle dispose de plusieurs fonctions :

— Une vue principale, non-masquable, qui affiche la saisie actuelle, ainsi que les X derniers éléments de la pile (les éléments les plus récents en bas) et l'état du calculateur (succès de l'opération ou erreur). La barre de saisie peut être remplie à l'aide du clavier, et le bouton EXE à sa droite lance l'évaluation de la saisie, de la même façon que la touche Enter du clavier.

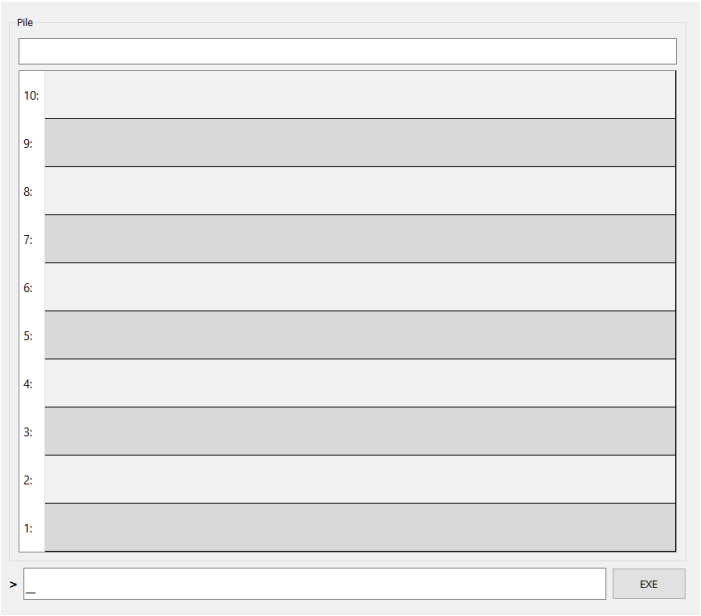

Figure 1 : Vue principale de l'application

— Une vue clavier numérique qui comprend des boutons cliquables correspondants aux 10 chiffres, à la virgule, ainsi qu'aux cinq opérateurs que nous avons considérés comme les plus récurrents et les plus importants : +, -, *, / et EVAL. Chacun de ses opérateurs est fonctionnel.

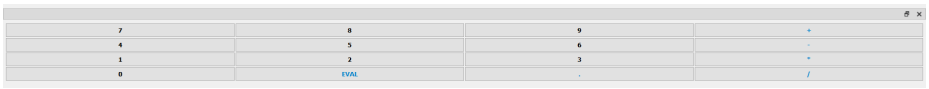

Figure 2 : Vue du clavier numérique

— Une vue clavier des fonctions qui comprend des boutons cliquables pour l'ensemble des opérateurs. Les opérateurs fonctionnels dans notre application sont :

  — Les opérateurs trigonométriques : SIN, COS, TAN, ARCCOS, ARCSIN, ARCTAN

  — Les fonctions mathématiques essentielles : NEG, DIV, MOD, NUM, DEN, SQRT, POW, LN, EXP, ainsi que ‘, [et] pour gérer les atomes et les programmes.

  — Les opérateurs logiques : =, <, >, !=, =<, >=, AND, OR, NOT

  — Les fonctions relatives à la pile : DUP, SWAP, CLEAR et DROP

  — Les opérateurs conditionnels et de boucle : IFT, IFTE, WHILE

  — Les opérateurs de gestions des variables et programme : FORGET et STO

  L'ensemble des opérateurs est pleinement fonctionnel, et seul les opérateurs UNDO et REDO n'ont pas été implémentés.

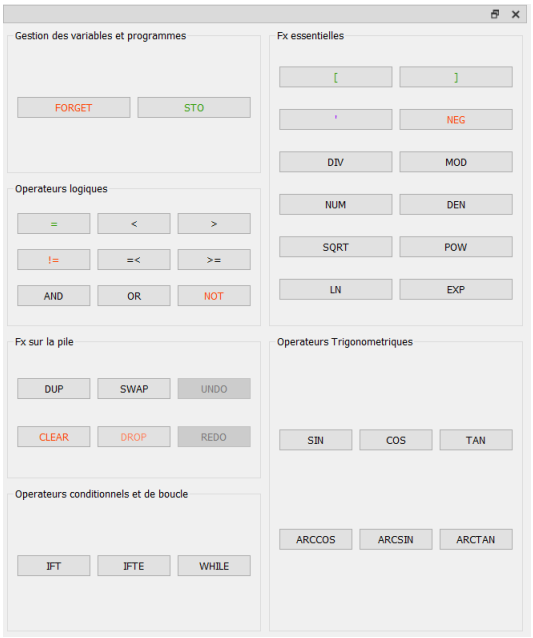

Figure 3 : Vue du clavier des fonctions

— Une vue de gestion des variables qui est initialement vide. Pour chaque variable créée par l'utilisateur, elle affiche un bouton cliquable. Le bouton permet d'empiler automatiquement le contenu de la variable. À droite de chaque bouton variable se trouve l'option “Editer” qui ouvre une fenêtre permettant d'éditer le nom ou la valeur de la variable.

Figure 4 : Vue de gestion des variables

— Une vue de gestion des fonctions qui est initialement vide. Pour chaque fonction créée par l'utilisateur, elle affiche un bouton cliquable permettant d'exécuter automatiquement le programme. À droite de chaque bouton variable se trouve l'option "Editer" qui ouvre une fenêtre permettant d'éditer le nom ou le contenu du programme.

Figure 5 : Vue de gestion des programmes

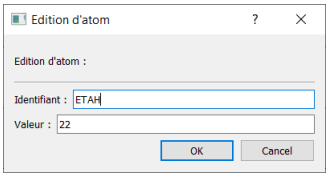

Figure 6 : Fenêtre d'édition des programmes ou variables

— En haut de l'interface se trouvent trois menus déroulants :
  — "Interface" permet de masquer ou de démasquer toutes les vues précédentes, à l'exception de la vue principale. Toutes ces vues sont déplaçables et masquables directement à l'aide des boutons "Réduire" et "Fermer" situés en haut à droite de chaque vue. Une fois réduite, une vue peut être déplacée autant que souhaitée, ou être de nouveau figée avec la vue principale. On peut aussi ouvrir les paramètres, permettant de modifier le nombre d'éléments de la pile que l'application va afficher.

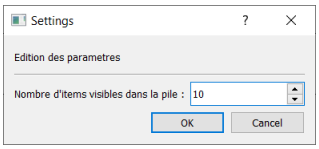

Figure 7 : Fenêtre des paramètres

— “Autres” permet d’ouvrir une fenêtre “A propos” contenant les informations sur le projet et le groupe, ainsi qu’un lien vers le repository git du projet.

Figure 8 : Fenêtre d’informations

— “Fichier” permet de sauvegarder les variables ou les programmes de son choix, ou de les charger à nouveau dans l’application à partir d’un fichier.

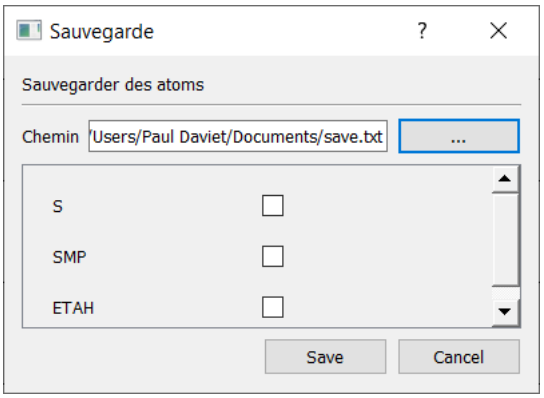

Figure 9 : Fenêtre de sauvegarde des variables et programmes

Deuxième partie

Architecture

Nous avons choisi, lors de l'implémentation, de séparer le code en deux parties : l'Interface, qui gère l'interface utilisateur (UI) à l'aide des différentes classes de Qt, et l'Engine, qui gère le traitement des expressions. Afin de faire la liaison entre ces deux parties, on utilise la classe *Connector*, sous-classe de *Observer*.

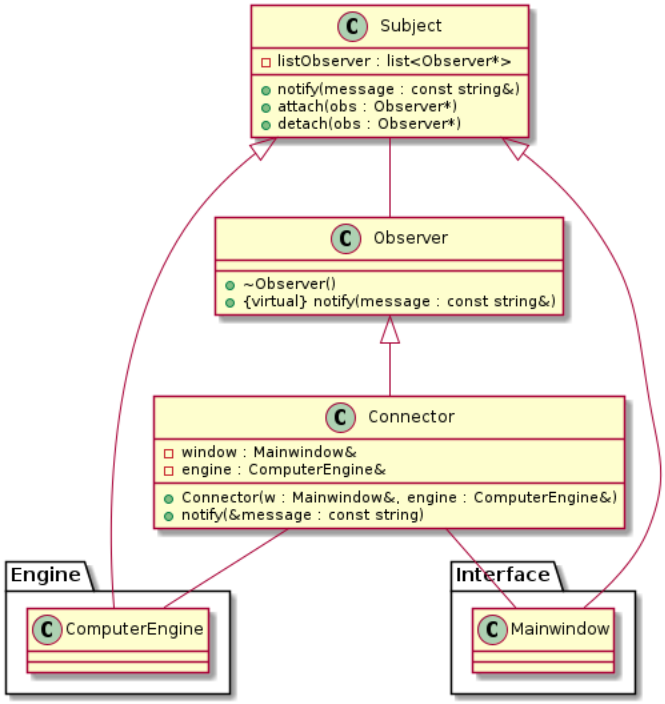

Figure 10 : UML de la structure général

1 Interface

Nous avons décidé de travailler avec QtDesigner et utiliser des fichiers .ui pour l'interface. Cela permet d'alléger le code (nous avons juste besoin de charger les .ui) mais aussi de pouvoir éditer l'interface de manière beaucoup plus rapide grâce au Drag and Drop.

La classe principale gérant l'interface est *Mainwindow*. Elle hérite à la fois de la classe *Subject*, afin de communiquer avec l'Engine, mais aussi de la classe *QMainwindow* de Qt, afin de pouvoir afficher la fenêtre principale. Elle possède en attributs les références de l'ensemble des sous-classes de *QWidget* et *QDialog* que nous avons défini, afin de pouvoir communiquer avec eux.

Pour gérer les différents éléments de la fenêtre principale, plusieurs classes héritent de la classe de Qt *QWidget* :

— *Pile* qui gère l'affichage de la pile

— *KeyboardNumeric* qui gère l'affichage du clavier numérique

— *KeyboardFunctions* qui gère l'affichage du clavier des fonctions

— *Commandline* qui gère l'affichage de la ligne de commande

— *Programmes* qui gère l'affichage de la vue de gestion des fonctions

— *Variables* qui gère l'affichage de la vue de gestion des variables

Les classes *Programmes* et *Variables* devant pouvoir afficher de nouveaux boutons cliquables, ceux-ci sont gérés par une classe spécifique, respectivement *Programme* et *Variable*. Ces deux classes héritent également de *QWidget*. Elles sont agrégées à *Programmes* et *Variables* par une liste en attribut privé, afin de pouvoir recevoir les informations.

Afin de rendre certains textes cliquables, comme dans les vues de gestion des variables et des programmes, on utilise la classe *ClickableLable*, qui hérite de la la classe *QLabel* de Qt et dont la méthode *MoussPressEvent()* permet de détecter le clic de l'utilisateur.

Pour gérer les différentes fenêtres de dialogues, plusieurs classes héritent de la classe de Qt *QDialog* :

— *EditAtom* gère la fenêtre d'éditions des atoms (programme ou variable)

— *About* gère la fenêtre d'informations

— *Settings* gère la fenêtre d'édition des paramètres

— *SaveWindow* gère la fenêtre de sauvegarde des variables et programmes

La classe *SaveWindowItem* génère les différents composants de la fenêtre de sauvegarde, et hérite pour ce faire de la classe *QWidget*.

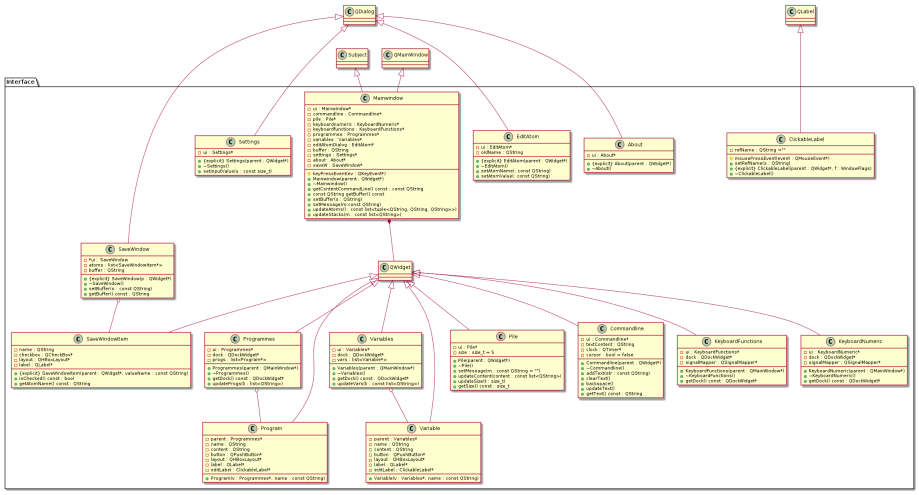

Figure 11 : UML de l'interface

2 Engine

L'ensemble des classes de la partie Engine ont été placées dans un même namespace intitulé *Engine*.

La classe principale est *ComputerEngine*, qui communique avec les différentes classes ainsi qu'avec l'interface via un objet *Connector*.

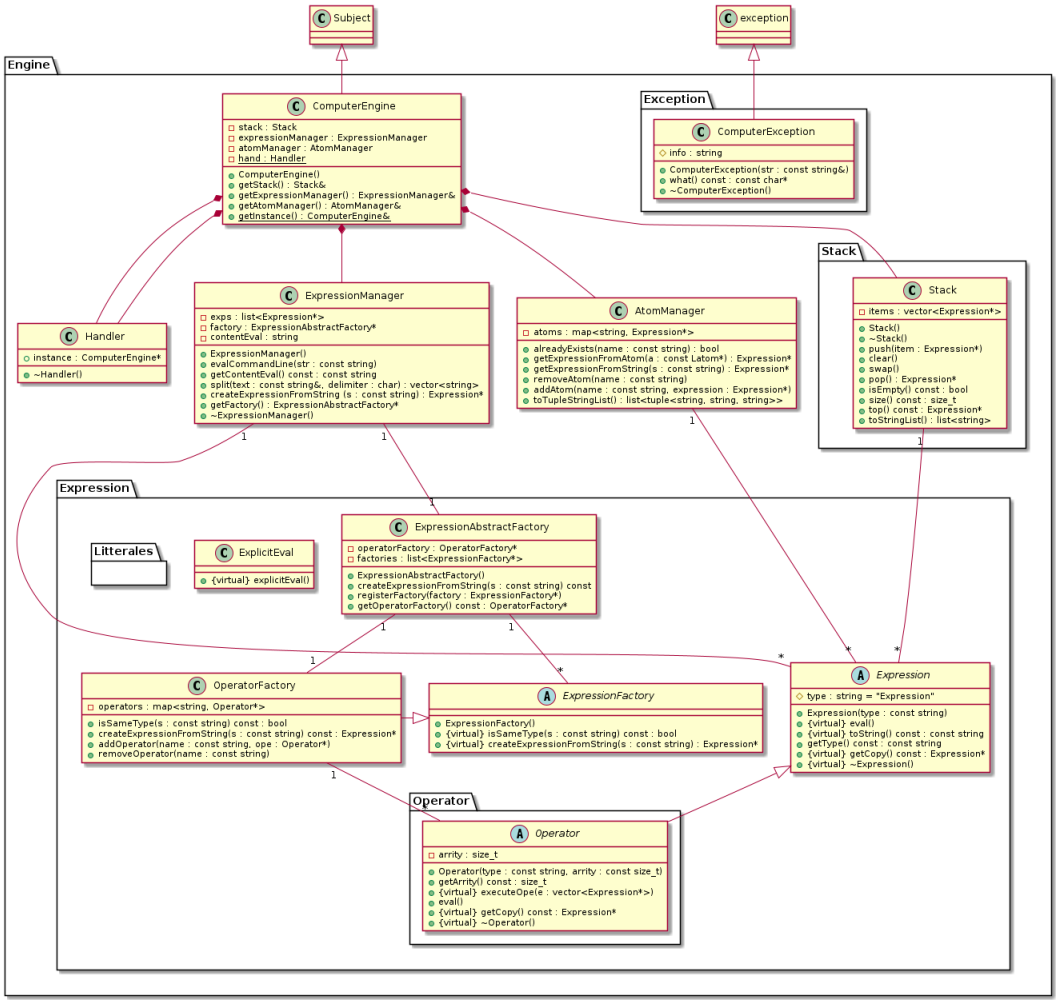

Figure 12 : UML de l'engine

2.1 ComputerEngine

La classe *ComputerEngine* hérite de la classe *Subject*, afin de pouvoir communiquer avec l'interface en notifiant la classe *Observer*. Étant le moteur principal, il contient en attribut privé les classes *Stack*, *ExpressionManager* et *AtomManager*, qui le composent. Devant nous assurer de son unicité, nous avons intégré le design pattern *Singleton* à l'aide d'une classe

Handler passée en attribut privé, et d'un accès à cette classe par un attribut et une méthode statique.

2.2 Expression

2.2.1 Operator

Différents opérateurs sont implémentés dans l'application. Chaque classe d'opérateur hérite de la classe abstraite *Operator*.

La classe *Operator*, qui hérite elle-même de la classe abstraite *Expression*, est composée de :

- un attribut privé représentant l'arrité de l'opérateur, qui détermine le nombre de littérales à dépiler pour effectuer l'opération attendue.
- une méthode *executeOpe()* prenant en paramètre un vecteur de pointeurs sur des objet *Expressions* ; cette méthode définit les opérations à effectuer et appelle les méthodes nécessaires pour retourner la littérale qui en résulte. Grâce au polymorphisme et au caractère virtuel de cette méthode, nous pourrons la redéfinir pour chaque opérateur en fonction de son arrité et des actions à effectuer.
- une méthode *eval()* qui vérifie que le nombre de littérale dans la pile est bien supérieur ou égale à l'arrité de la classe. En cas d'échec, les littérales sont bien rétablis dans la pile. En cas de succès, les littérales sont transmises à la méthode *executeOpe()*.

Les opérateurs arithmétiques unaires et binaires, ainsi que les opérateurs logiques ont chacun une classe abstraite correspondante, *OperatorAritBinary*, *OperatorAritUnary* et *OperatorLogic*. Les opérateurs suivants héritent d'une de ces classes abstraites :

- Les opérateurs *OperatorPLUS*, *OperatorMINUS*, *OperatorMUL*, *OperatorDIV*, *OperatorDIVINT*, *OperatorMOD*, *OperatorPOW* héritent de *OperatorAritBinary*.
- Les opérateurs *OperatorNEG*, *OperatorNUM*, *OperatorDEN*, *OperatorSIN*, *OperatorCOS*, *OperatorTAN*, *OperatorARCCOS*, *OperatorARCSIN*, *OperatorARCTAN*, *OperatorSQRT*, *OperatorEXP*, *OperatorLN* héritent de *OperatorAritUnary*.
- Les opérateurs *OperatorAnd*, *OperatorOr*, *OperatorNot*, *OperatorEq*, *OperatorGeq*, *OperatorLeq*, *OperatorGt*, *OperatorLt*, *OperatorDiff* héritent de *OperatorLogic*.

Afin de gérer les différents comportements en fonction du type des littérales dépilés, et afin d'être flexible à l'ajout de nouvelles littérales, on a définit des classes Actions. Chaque type d'opérateur possède une classe abstraite d'actions correspondante, à savoir *ActionBinary*, *ActionUnary*, *LogicTest*. De multiples classes filles sont définis pour chaque opérateur.

On a aussi implémenté le design pattern *Factory* à l'aide de la classe *operatorfactory* qui permet d'instancier le bon opérateur en fonction du type des littérales et des types valides pour les Actions. Cette dernière hérite de la classe *ExpressionFactory*.

Pour enregistrer un nouvel opérateur il suffit que l'objet de la classe *OperatorFactory* créé dans la classe *ExpressionAbstractFactory* appelle la méthode *addOperator* qui prend en paramètre le nom de l'opérateur et l'instancie au moyen du constructeur correspondant.

Voici un exemple d'enregistrement de l'opérateur *CLEAR* :
operatorFactory->addOperator("CLEAR",new OperatorCLEAR) ;

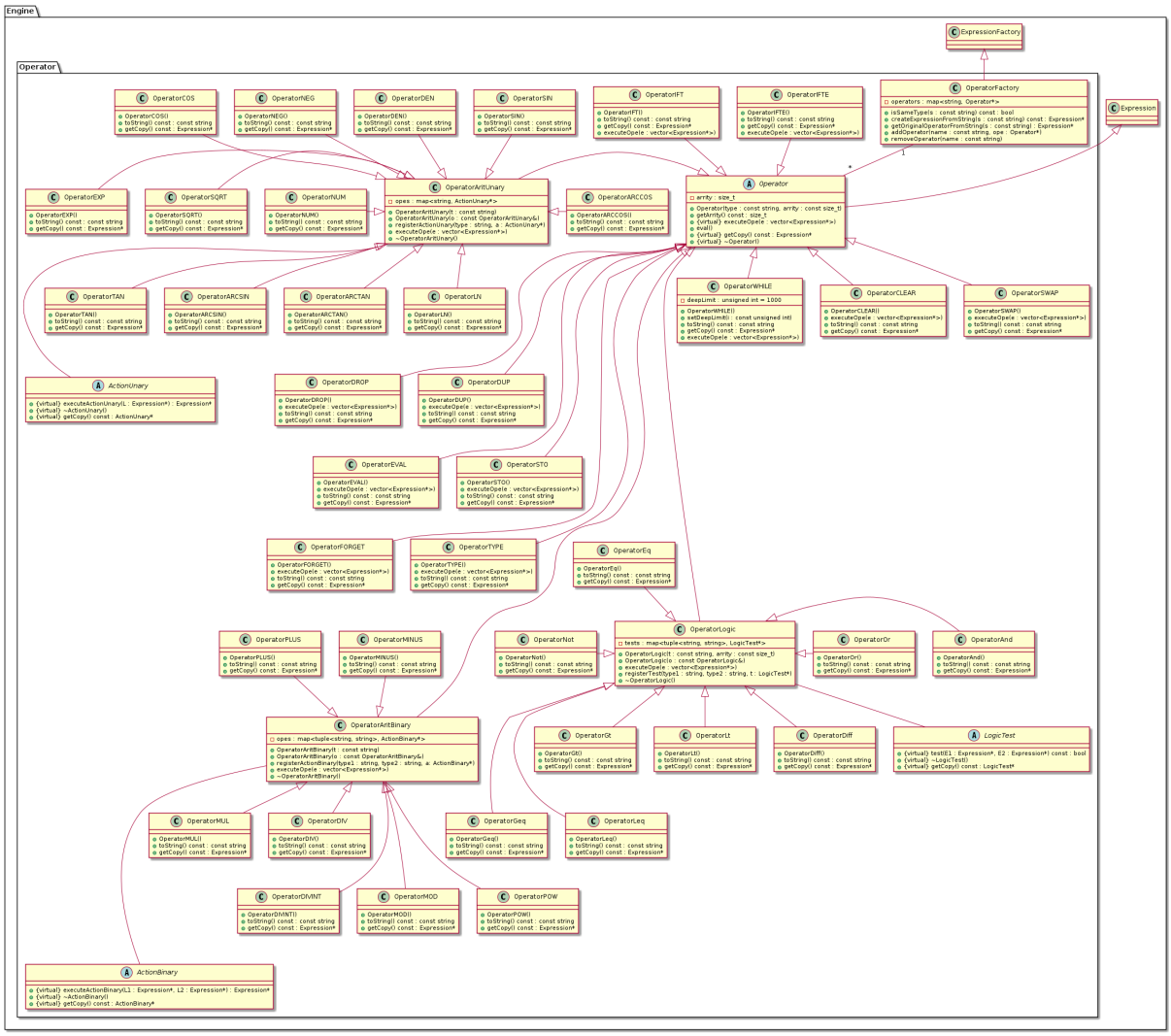

Figure 13 : UML des operators

2.2.2 Literal

Les différentes valeurs saisies par l'utilisateur sont représentées par des classes littérales, héritant toutes de la classe abstraite *Expression*. Elles héritent ainsi notamment de l'attribut *type*, qui permet de différencier chaque littéral, de la même façon que les opérateurs. Chaque classe littérale représente un type de valeur différents. Afin de gérer la création de ces objets littérales, chaque classe utilise le design pattern Factory, chaque Factory héritant de la classe *ExpressionFactory*. Les classes Factory disposent d'une méthode reconnaissant si un string fourni correspond au type de littérale produit par la Factory, sans quoi la littérale n'est pas

créée.

Afin de gérer les littérales numériques, on utilise une classe abstraite *lnumerical*. Cette classe, en plus de ses constructeurs, dispose de deux méthodes. La méthode *eval()* implémente l'évaluation de la littérale, qui est identique pour toutes les classes filles, en empilant juste la valeur. La méthode virtuelle *simplifyType()* permet de simplifier les littérales, en changeant la littérale de classe si nécessaire (un rationnel peut être simplifié en entier par exemple). Les classes filles de *lnumerical* sont :

- *linteger*, qui gère les littérales entières. La valeur est stockée dans un attribut privé de type int. Cette classe dispose de sa propre classe factory, *lintegerFactory*.
- *lreal*, qui gère les littérales réelles. La valeur est stockée dans un attribut privé de type double. Cette classe dispose de sa propre classe factory, *lrealFactory*. Un objet *lreal* peut être simplifié en *linteger* grâce à la méthode *simplifyType()*.
- *lrational*, qui gère les littérales rationnelles. La valeur est stockée dans deux attributs privés de type int, le dénominateur et le numérateur. Cette classe hérite de la classe *editString*, afin de pouvoir afficher une chaîne différente que celle utilisé lors de sa génération. Un objet *lrational* peut être simplifié en objet *linteger* ou en un autre objet *lrational* grâce à la méthode *simplifyType()*.

Afin de définir le domaine des différentes classes filles de *Lnumerical*, on utilise une classe *R1value*, contenant la méthode *getValue()* qui est héritée par les classes *linteger*, *lreal*, *lrational*.

La classe *lprogram* enregistre une liste d'expression entourée par des crochets dans un attribut str. Sa méthode *eval()* héritée d'*Expression* l'empile simplement. Cette classe hérite d'une autre classe, *ExplicitEval*, et ainsi de la méthode *explicitEval()* qui permet une évaluation demandée par l'utilisateur (EVAL) ou par l'Engine (exécution d'un opérateur IFT, IFTE, ...). Tout comme la classe *lrational*, *lprogram* hérite aussi de la classe *editString* qui permet d'afficher une littérale différemment de sa génération.

La classe *lexpression* encapsule des expressions, c'est-à-dire des littérales entre apostrophes. De la même façon que *lprogram*, cette classe est empilée simplement par *eval()*, et hérite donc de la classe *ExplicitEval* pour pouvoir être évaluée différemment par certains opérateurs.

Enfin, la classe *latom* sert d'identifiant pour les variables et les programmes. S'il n'est pas défini, il empile simplement une *lexpression* lors de son évaluation. S'il est défini, et donc liée à un objet *lprogram*, il évalue cet objet. Il faudra noter qu'un objet de la classe *latom* est un paramètre de la méthode *getExpressionFromAtom()* de la classe *AtomManager* qui s'occupe de la gestion de stockage des objets *Lnumerical* et *Lprogram* associés a un nom. La classe *AtomManager* nous sera utile dans l'implementation des operateurs *OperatorSTO* (resp. *OperatorFORGET*) qui stockent (resp. suppriment) des variables ou des programmes. La méthode *executeOpe()* de l'opérateur *OperatorSTO* prend en référence un objet de la classe *AtomManager* et vérifie si le type des deux dernières expressions de la pile permet la formation d'un programme ou d'une variable. Si oui elle vérifie par la suite si ce nom de variable ou de programme existe déjà pour que l'*AtomManager* fasse appel à la méthode *addAtom()*. Cette dernière fait elle-même appel à la méthode *insert()* prédéfini de la classe

map (STL) pour ajouter un nouvel atom dans le conteneur *atoms*, qui est l'unique attribut de la classe *AtomManager*. Par ailleurs l'opérateur *OperatorFORGET* fait appel à la méthode *removeAtom()* pour supprimer un atom du conteneur *atoms*.

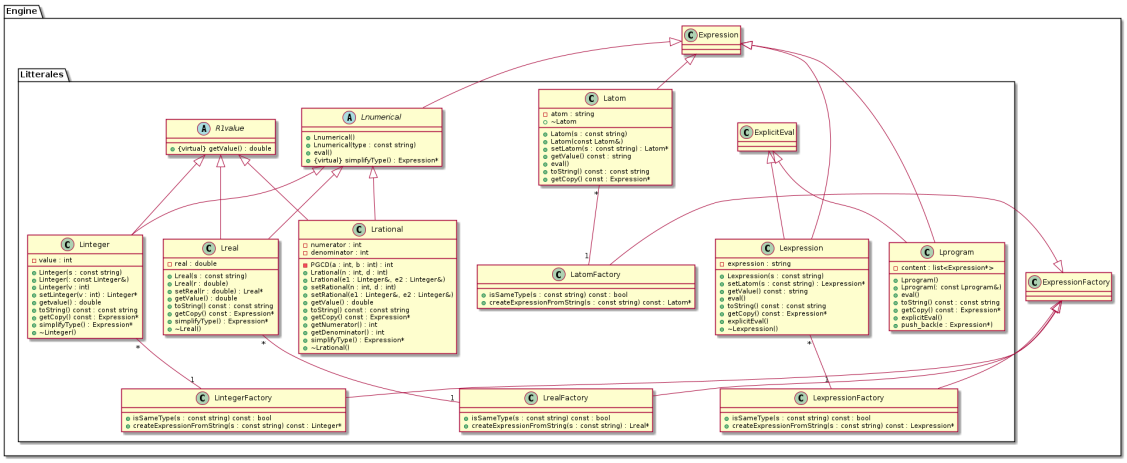

Figure 14 : UML des littérales

2.3 Stack

La classe *Stack* représente la pile de la calculatrice. La calculatrice fonctionnant en Notation Polonaise Inversée (RPN), chaque littérale ou opérateur est empilé sur la pile lors de sa saisie, puis dépilé lors de son évaluation. Un vecteur de pointeurs sur des objets *Expression* en attribut privé permet de stocker le contenu de la pile, tandis que les différentes méthodes de la classe permettent de modifier la pile.

— *push()* permet de rajouter une expression à la pile

— *pop()* dépile une expression

— *clear()* supprime toutes les expressions stockées

— *swap()* inverse la position des deux dernières expressions

— les autres méthodes renvoient des informations sur la pile

Chaque méthode envoie un message au *Connector* qui transmet à l'Interface via *ComputerEngine*. A la réception du message, le *Connector* récupère le contenu de la pile en std : :string, et le convertit en QString, puis demande à l'Interface de le prendre en compte et de se mettre à jour.

2.4 Exception

Afin de gérer les différentes exceptions pouvant avoir lieu, on utilise une classe *ComputerException*, qui hérite de la classe standard *Exception*, afin de garder une cohérence avec la majorité des codes C++ qui utilisent une même interface de gestion des erreurs.

3 Observer et Connector

Afin de permettre la communication entre l'Interface et l'Engine, on passe par la classe *Connector*, en utilisant le design pattern Observer.

La classe *Observer* est une classe abstraite, uniquement désignée à être héritée par la classe *Connector*. Elle permet l'implémentation du design pattern Observer et de respecter le principe d'encapsulation.

La classe *Subject* permet de désigner une classe comme observable. Elle est héritée par les classes *ComputerEngine* et *Mainwindow*. Les méthodes *attach()* et *detach()* permettent respectivement d'associer ou de supprimer un objet *Observer*. La méthode *notify()* permet d'envoyer un message à tous les objets *Observer* associés à l'objet *Subject*.

La classe *Connector* hérite de la classe *Observer*. Son constructeur connecte l'objet avec un objet *ComputerEngine* et *Mainwindow*, afin de pouvoir recevoir et transmettre les signaux via la méthode *notify()*.

Troisième partie

Argumentation

1 Protection de l'architecture

Les attributs des classes sont privés et ne sont accessibles que par des méthodes de classe. De plus les méthodes qui ne sont pas supposées changer la valeur des attributs sont const.

Chaque classe correspond à une seule fonction, et l'architecture est ainsi correctement encapsulée.

Par ailleurs, étant un Singleton, la classe *ComputerEngine* nous permet de nous assurer qu'elle sera instanciée une seule fois lors de l'exécution de notre programme.

2 Évolution et adaptabilité

L'architecture de notre projet est prévue pour être facilement adaptable et étendue au besoin.

Utiliser la classe *Connector* qui hérite d'*Observer* pour faire le lien entre les parties Engine et Interface permet d'être flexible. En effet, cela permettrai de rajouter facilement une nouvelle partie qui pourrait communiquer avec les deux autres. De même, on peut avoir plusieurs Interfaces en multipliant les objets *mainwindow*. Cependant, le design pattern Singleton implémentée avec *ComputerEngine* assure l'unicité de l'Engine et donc la sûreté de l'application.

Au moyen du design pattern Abstract Factory un nouveau type d'opérateur ou de littéral peut facilement être ajouté. Si un nouvel opérateur devait être ajouté, il suffirait de le définir, avec son arrité, son type et sa méthode *executeOpe()*. S'il doit pouvoir interagir avec différents types de littérales, des actions seront aussi à définir. Enfin, pour l'enregistrer dans la factory, il suffit que l'objet de la classe *OperatorFactory* créé dans la classe *ExpressionAbstractFactory* appelle la méthode *addOperator()* qui prend en paramètre le nom de l'opérateur et l'instancie au moyen du constructeur correspondant. Tout ceci se fait sans avoir besoin de modifier le code existant.

De la même façon, si l'on ajoutait une nouvelle littérale, comme une littérale complexe, il suffirait de créer la nouvelle classe *lcomplex*, héritant de *lnumerical* et donc de *Expression* ainsi que sa factory *lcomplexFactory* héritant de *ExpressionFactory*. Les autres classes n'ont pas besoin d'être modifiées. La méthode *isType()* de la factory permettra de définir ce type, en reconnaissant une expression regex que l'on aura défini. La factory peut facilement être ajoutée à *ComputerEngine* grâce à la méthode *registerFactory()*. Afin de rendre les opérateurs existants compatibles avec la nouvelle littérale *lcomplex*, il suffirait de rajouter les classes Actions correspondantes en accédant à la classe *OperatorFactory*, puis à l'opérateur désiré afin d'enregistrer une nouvelle action. Cette méthode permet de respecter le principe

ouvert/fermé qui affirme qu'une classe doit être extensible sans avoir à modifier le code déjà existant .

3 Respect des bonnes pratiques de codage

Nous avons tenu à respecter le principe de substitution pendant l'implémentation de chaque classe et sous classe dans lesquelles les méthodes non virtuelles de la super classe ne sont pas redéfinies.

Le destructeur de toute classe sous-classée est virtuel afin de respecter le principe de substitution, toutes les méthodes qui vont être redéfinies sont déclarées virtuelles pour que le principe d'encapsulation soit respecté. Le but de l'implémentation des méthodes virtuelles pures est que les classes dérivées héritent seulement de l'interface mais pas du comportement qui n'est pas défini dans la classe de base. On utilise bien le mot clé "override" pour indiquer qu'une méthode sera une redéfinition d'une méthode existante dans une super classe.

Durant tout le projet on s'est appuyé sur un logiciel de gestion de versions Git afin de synchroniser les tâches et d'ajouter le code à l'aide du concept de branches. A chaque fois qu'une modification ou une nouvelle fonctionnalité a été ajoutée, l'auteur crée une Pull Request (PR) de manière à ce qu'elle soit vérifié et commentée par les autres membres du groupe. Une fois approuvé l'auteur peut ajouter son bout de code au projet sur la branche principale (main) en faisant un merge. Voici le lien de notre dépôt.
Ci-dessous un schéma de notre flux de travail :

Figure 15 : Schéma de l'organisation

Quatrième partie
Organisation

1 Planning

Notre planification se divise en trois phases :

— la phase de conception
— la phase de développement
— la phase de rendu

S'ajoute à ces trois phases notre formation à Github.

Nous faisons la distinction des phases, cependant nous avons souvent travaillé en parallélisant ces dernières. En effet, nous voulions débuter le code dès qu'une partie de l'architecture était approuvée. Nous avons donc travaillé de manière cyclique durant toute la durée du projet :

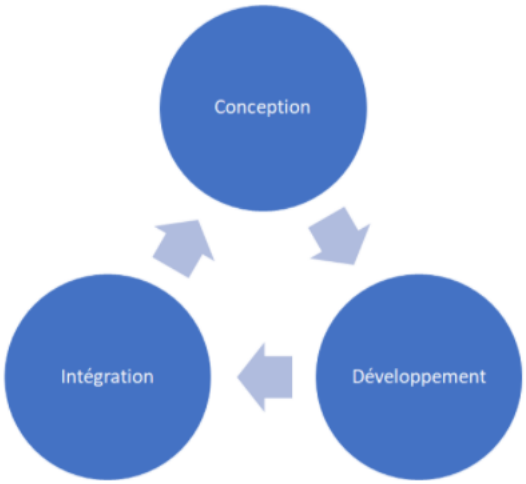

Figure 16 : Schéma de l'organisation

Nous avons travaillé en mode AGILE, c'est-à-dire que nous avons développé notre application morceau par morceau (grâce aux branches de github) et intégré chacun de ces morceaux à la partie principale du code (c'est-à-dire à la branche main du repository github). Cela nous permettait de concevoir, développer et intégrer non pas notre application en une seule et grande partie, mais en plusieurs parties toutes fonctionnelles ; étant donné que pour intégrer le code dans le main, celui-ci devait avoir une review approved.

Par ailleurs, nous pouvons distinguer quatre parties dans le développement de l'application :

— D'abord, le développement de la partie interface. En premier lieu, ce sont les fenêtres où se trouvent les claviers cliquables qui ont été développées, ensuite les fenêtres pouvant

se fermer et s'ouvrir. En fin de projet, nous avons ajouté la fenêtre "à propos" puis la fenêtre "sauvegarder". L'interface ayant été développée en parallèle de la partie moteur, nous avons pu déboguer facilement notre application, et nous avions un logiciel fonctionnel à chaque instant du développement.

— Ensuite, dans le développement de la partie moteur ou Engine : nous avons d'abord codé les parties les plus utiles, et celles que nous avions vues en TD : Expression, ExpressionManager et Pile notamment. Ensuite, nous nous sommes réparties les tâches entre les deux types d'expressions que sont les littérales et les opérateurs. Pour les littérales, nous avons d'abord codé les plus basiques afin de pouvoir tester notre interface. Puis nous avons développé des littérales plus complexes, notamment les littérales atome et programme qui nécessitaient un traitement particulier. En parallèle, les opérateurs ont été codés. Tout ceci a été fait sur différentes branches (github) et représente les morceaux de code dont on parlait au-dessus. Chacune de ces branches ont été développé indépendamment les unes des autres puis intégrées au reste du code qui était fonctionnel.

— De même, la partie reliant interface et moteur a été codée à chaque fois qu'une fonctionnalité était ajoutée. Par exemple, la fonctionnalité de calcul de somme a pu être testée sur l'interface après avoir relié cette fonctionnalité à cette dernière.

— Nous avons pu détecter les bugs de manière séquentielle grâce aux différentes pull request, lors de l'intégration des morceaux de code au reste.

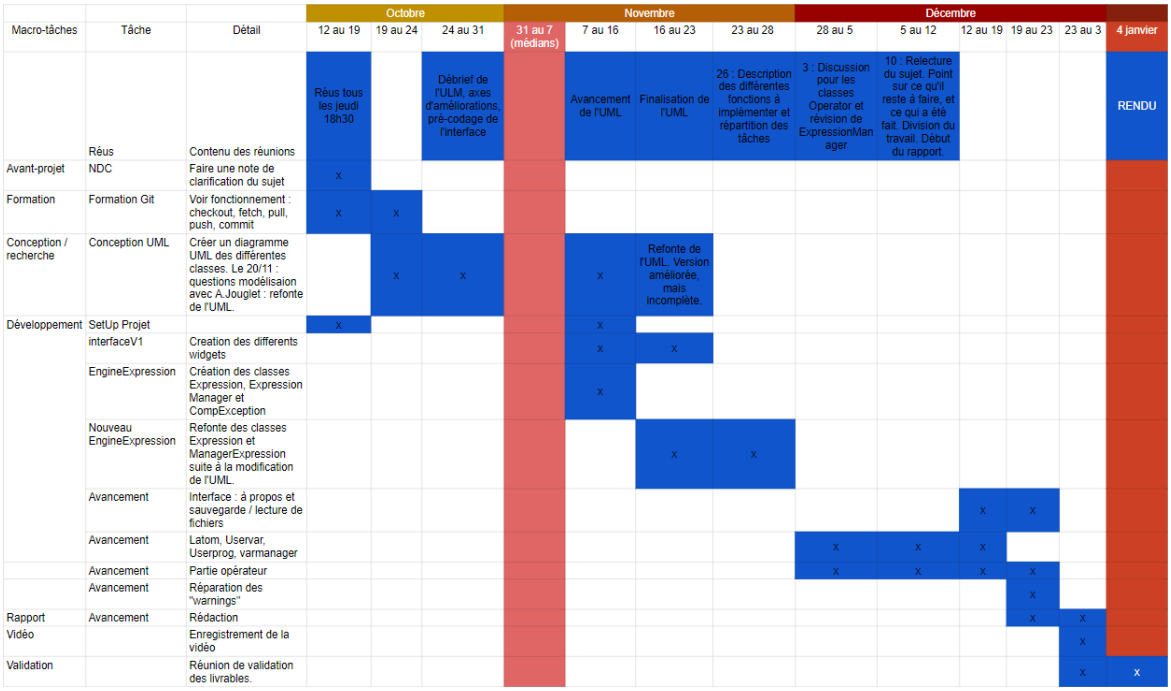

Figure 17 : Planning du projet

2 Contributions personnelles

BRANLY Stéphane

Gestion du projet :

- Répartition des tâches
- Création et gestion du Repository Github
  - ajout de template de PR
  - ajout de sécurité sur la branche main (anciennement 'master') pour empêcher les commits et PRs sans Approve.
- Reviews de PR de mes camarades
- Aide pour mes camarades afin de guider sur l'implémentation de fonctionnalités

Implémentation en C++ :

- Interface utilisateur
  - Création de toute l'interface
    - Classe mainwindow (contenant la pile, la commandLine, les différents onglets)
    - Vues secondaires (les claviers cliquables, le gestionnaire de programmes et variables, la sauvegarde/ouverture de fichiers, les informations, les paramètres de l'application)
  - Connection de l'interface avec la partie Engine
- Moteur de la calculette
  - Implémentation de différentes parties
    - Connecteur UI et Engine
    - Concept factories pour les expressions et pour les opérateurs
    - Quelques opérateurs agissant sur la pile
    - La gestion de sauvegarde et suppressions de Latoms
    - Création des Lexpression et Lprogram
    - Fonctionnement de la création et l'évaluation des expressions (avec gestion de la profondeur des programmes)
  - Corrections et refactor
    - Quelques corrections ont été faites sur différentes parties afin d'avoir le comportement global désiré
    - Refactor global de plusieurs parties du code afin d'éviter les répétitions et avoir un code plus lisible
- Réflexion
  - Réflexion avec mes camarades sur l'architecture à adopter

Rapport :

- Ajout de commentaires pour la rédaction du rapport

Globalement, en comptant les réunions, les Reviews de code, débogage, implémentation de l'engine et interface, réflexion du projet, débat avec un autre camarade d'un autre groupe sur l'architecture, je pense cumuler plus de 75h de travail sur ce projet.

CHAHM Saad

Etude de projet :

- Réflexion sur l'architecture
- Réalisation d'une première version de l'UML de l'interface
- Participation à la répartition des tâches
- Review d'une PR

Implémentation en C++ :

- Opérateurs Logiques :
  - opérateurs binaires pour les tests respectivement égal, différent, inférieur ou égal, supérieur ou égal, strictement inférieur, strictement supérieur.
  - `AND`, opérateur binaire : ET logique.
  - `OR`, opérateur binaire : OU logique.
  - `NOT`, opérateur unaire : NON logique.
- Opérateurs Conditionnels :
  - L'opérateur binaire `IFT`
  - L'opérateur binaire `IFTE`

Rapport :

- Rédaction du rapport
- Mise en page sous LaTeX

De manière générale, en comptant les réunions, l'assimilation de l'architecture, l'initiation a `Git`, l'implémentation des opérateurs précisés ci-dessus, la rédaction du rapport, je pense cumuler plus de 40h de travail sur ce projet.

DAVIET Paul

Conception :

- Réflexion commune sur l'architecture
- Réalisation de l'UML
- Review de PR et report de bug

Rapport :

- Rédaction du rapport
- Mise en page sous LaTeX
- Réalisation des différents UML
- Réalisation de la vidéo de présentation

Globalement, je suis conscient d'avoir moins travaillé et moins participé à ce projet, car j'ai accumulé du retard au début de l'implémentation difficile à rattraper. J'ai tout de même passé une quarantaine d'heure sur le projet, entre les différents rendus, les réunions et le travail sur l'architecture.

GIRAMELLI Manon

Gestion du projet :

- Gestion avec Stéphane de la création du Repository Github
  - Aide initialisation et paramétrage
  - Fix des premières erreurs concernant le fonctionnement sous différents OS
- Reviews de quelques PR de mes camarades
- Aide pour mes camarades sur l'utilisation du Repository Github
- Participation et gestion active des réunions, répartition des tâches hebdomadaires à chacun pour faire avancer le projet
- Réflexions autour de l'UML

Implémentation en C++ :

- Moteur de la calculette
  - Base de la pile refaite complètement par la suite
  - Implémentations de tous les opérateurs arithmétiques :
    - Opérateurs unaires : NEG NUM DEN COS SIN TAN ARCCOS ARCSIN ARCTAN SQRT EXP LN
    - Opérateurs binaires : + - * / DIV MOD POW
    - Gestion de toutes les classes actions liées à ceux-ci
    - Gestion operatorFactory
    - Élaboration d'actions cohérentes, respectant notamment l'ensemble de définitions des fonctions liées à certains opérateurs.
  - Réflexion avec mes camarades sur l'architecture à adopter

Rapport :

- Relecture et commentaires

Globalement, en comptant les réunions, les Reviews de code, débogage, implémentation de l'engine, réflexion du projet, je pense cumuler entre 40h et 50h de travail sur ce projet.

POUGHET Louise

Dans la première phase du projet (la phase de définition), j'ai rédigé une note de clarification pour condenser le cahier des charges, et donc pour clarifier les attentes.

Ensuite, dans la phase de réflexion sur l'architecture logiciel, j'ai étudié en détail le sujet. Avec l'aide des TDs sur le thème de la calculatrice et l'étude du sujet, j'ai pu en déduire une première architecture assez modeste et globale. Très vite, il est apparu pertinent de séparer en deux parties notre projet : d'un côté la partie interface que l'on peut apparenter au front-end et la partie moteur ou engine que l'on peut nommer Back-end. Après une seconde réunion, nous avons mis en commun nos travaux, nous avons avancé dans notre réflexion : l'architecture était déjà plus complexe et nous avions séparé en deux parties notre application. Dans toute cette phase j'ai essayé de tenir à jour un diagramme de classe UML succinct mais qui nous permettait de reprendre nos idées rapidement. Au fur et à mesure, certaines parties de l'architecture se sont révélées adaptées et d'autres ont dû être

repensées. J'ai pris part aux discussions de l'équipe, et à chaque fois nous pesions le pour et le contre.

Lors de la phase de développement, j'ai implémenté les parties suivantes : Expression et ExpressionManager Lnumerical et ses sous classes : Lreal, Linteger, Lrationnal Lexpression Latom la classe CompException Stéphane a créé les factories, que j'ai parfois reprises. Dans notre utilisation de Github, nous avions à "review" des "pull request". C'est à dire qu'il fallait que l'on valide le code d'une autre personne de l'équipe. Il est nécessaire de faire ces validations rapidement pour avancer rapidement le projet. J'ai review de nombreuses pull request. Cela m'a amené à repérer certains "bugs" que je communiquais au reste de l'équipe. En terminaison de la partie développement, je créais des "issues", c'est-à-dire des problèmes de tout ordre sur l'application. De même, les autres membres de l'équipe me communiquait des problèmes dans ma partie du code, ou des améliorations auxquelles ils ont pensé, et je modifiais ces parties.

Pour la rédaction du rapport, je me suis chargée de la partie 4.1 sur le planning.

Date	Nombre d'heures	Tâche
14/10/2020	2	Lecture du sujet
15/10/2020	1	Réunion : division des tâches et création de deadline
21/10/2020	2,5	Relecture du sujet, première formalisation du problème (UML).
10/11/2020	1,5	Plantuml - réflexion sur l'architecture
11/11/2020	2	Plantuml + recherches design pattern
12/11/2020	1	Réunion
18/12/2020	2,5	Expression ExpressionManager CompException apprentissage git add, push etc
19/11/2020	3	Réunion avec sequence diagram
20/11/2020	1,5	Réunion avec A.Jouglet et mise à jour de l'UML
24/11/2020	2	Session code ExpressionManager et Expression mise à jour du Calendrier
01/12/2020	4	Session code ExpressionManager
03/12/2020	2	Réunion et code
05/12/2020	4	Déclaration des litéralles numérique, reelle, entiere, rationnelle
08/12/2020	4	Latom + définition
13/12/2020	3,5	Révision de Simplify de Lnumerail, Lreal, Linteger, Lnumerical
17/12/2020	2	Réunion + modification constructeur Linteger, Lreal simplification
21/12/2020	1	Review Pull request
22/12/2020	1,5	Commentaires + fix warnings
02/01/2021	3	Finitions : rapport, calendrier, rédaction contribution personnelle.

Total heures : 44

Au total, et d'après mon estimation (j'ai parfois oublié de mettre à jour le tableau ci-dessus, notamment pour les Pull request), j'ai passé environ 44 heures sur ce projet.

Pourcentage de la contribution personnelle

BRANLY Stephane	CHAHM Saad	DAVIET Paul	GIRAMELLI Manon	POUGHET Louise
34%	11%	7%	24%	24%

Cinquième partie

Conclusion

Ce projet nous a donné idée sur le déroulement des projets dans le monde professionnel et nous a permis de consolider nos acquis non seulement en termes de programmation orientée objet mais aussi nos compétences d'organisations de planifications et de communications.

Certes il a été difficile de pouvoir communiquer de temps à autre au vu de la situation actuelle, mais au final nous arrivions toujours à trouver un terrain d'entente afin de mener à bien le projet et de réaliser toutes les tâches demandées.

Enfin nous tenons à remercier le professeur de tout les efforts fournis durant ce semestre assez particulier dans le but de nous transmettre toutes les notions de la Programmation Orientée Objet, ainsi que celles du langage C++.