

|  |
| --- |
| **Rapport LO21** |
| **Calculatrice à notation polonaise inversée** |

**P12**

**BLOT Clémence GM04**

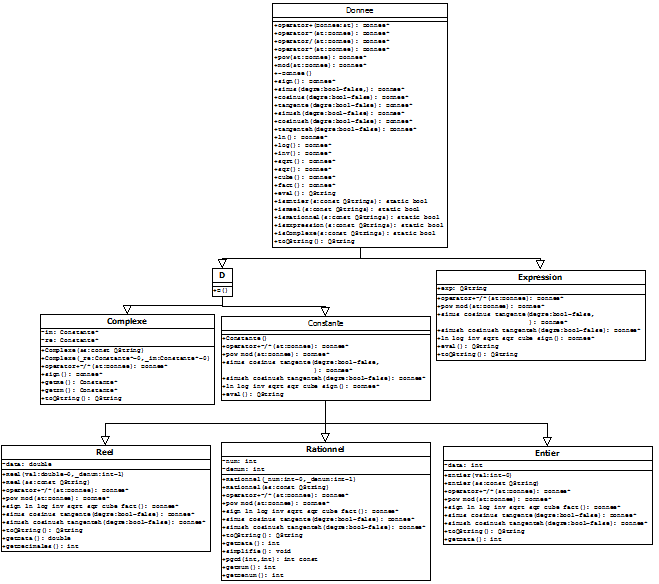
**GAVALDA Benoit TC06**

**Introduction :**

Le projet de LO21 nous a proposé de réaliser une calculatrice à notation polonaise inversée avec des fonctionnalités variées et des modes d’utilisation particuliers. Afin de remplir le cahier des charges imposé dans le cadre de cette Unité de Valeur, il a été nécessaire de faire des choix sur la conception du modèle. Le mémo suivant expose et explique ces différents choix.

**Conception de la calculatrice et diagramme de classe :**

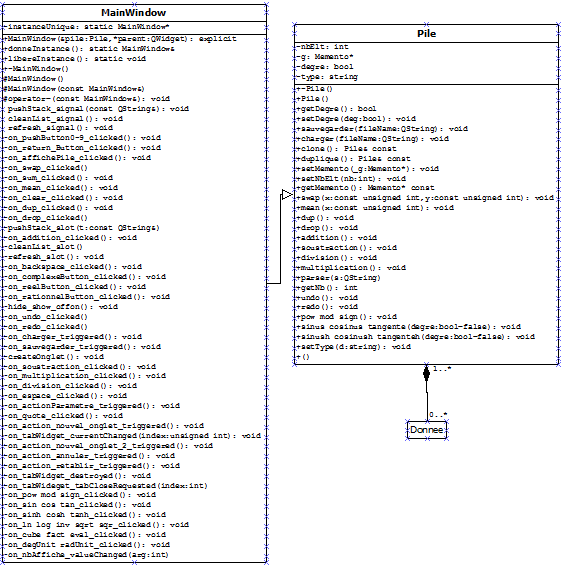
Afin de respecter les principes d’encapsulation du C++ et d’obtenir un résultat global cohérent, une réflexion a été faite pour organiser les différentes classes de ce projet.

1. Diagramme de classe des types de Données

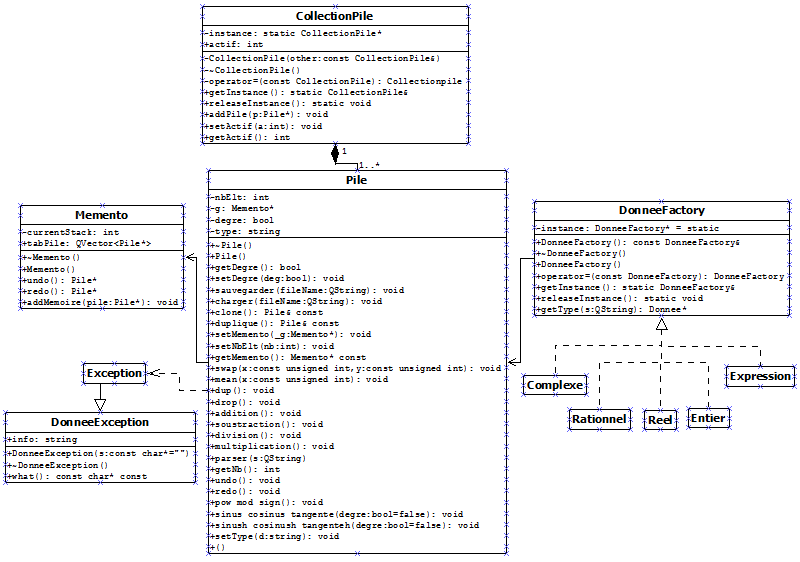
La classe Donnee est une classe virtuelle pure qui regroupe tous les types pouvant être supportés par la calculatrice. Ce sont des objets de type Donnee, c'est-à-dire du type le plus général possible, qui seront empilés.

Le système d’héritage ici présenté est celui qui nous a paru le plus adapté dans la représentation du problème à résoudre. Les classes Reel, Rationnel et Entier héritent de la classe virtuelle pure Constante. En effet, dans chacune de ces classes, les opérateurs de Constante seront redéfinis.

1. Diagramme de classe du fonctionnement principal de la calculatrice

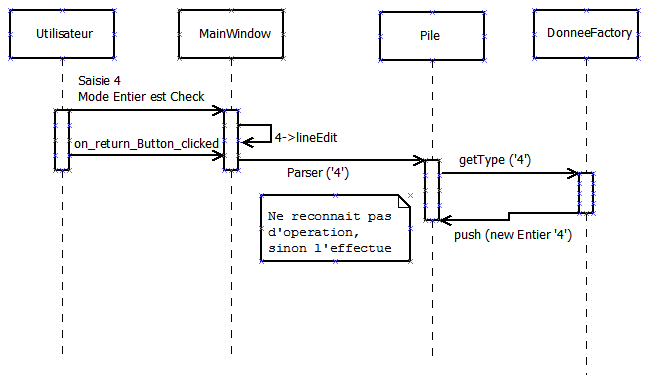


Ce diagramme montre les relations entre l’interface : MainWindow, instance unique grâce au Design Pattern Singleton, la classe Pile et la classe Donnee (celle-ci ayant été réduite pour plus de clarté).

1. Diagramme des classes des fonctionnalités annexes

Le diagramme ci-dessus montre les relations entre les différentes classes qui décrivent les fonctionnalités annexes de la calculatrice :

* **Memento :** Le Memento est un Design Pattern comportemental ici employé afin de réaliser la fonctionnalité Undo/Redo de la calculatrice. Il permet en effet de restaurer l’état précédent d’un objet. L’implémentation telle qu’elle est proposée ici permet de sauvegarder le contexte de la pile à chaque nouvelle opération.
* **CollectionPile :** La collection de piles ici implémentée permet de contrôler la création d’objets Pile. De nouvelles piles vont en effet être créées à la création d’un nouvel onglet et pour les fonctionnalités undo et redo.
* **DonneeFactory :** Le DonneeFactory est un Design Pattern créationnel permettant d’instancier des objets dont le type est dérivé d’une classe abstraite. Pour le projet, il a été utilisé pour instancier les classes : Complexe, Rationnel, Reel, Entier et Expression.
* **Exception :** La classe Exception permet de gérer les erreurs rencontrées lors de l’exécution du programme.
* **Dom :** Cette classe n’est pas représentée sur le diagramme car c’est une classe déjà présente dans Qt. Elle permet de gérer les fichiers. Ici, elle sera implémentée afin de sauvegarder et de charger le contexte de la pile à la fermeture ou à l’ouverture du programme.

**Exemple de scénario :**

Le diagramme de séquence ci-dessus représente la saisie d’un entier ‘4’ par l’utilisateur jusqu’à l’empilement de celui-ci dans la Pile active. Avec un opérande, la calculatrice fonctionne de la même manière sauf qu’au niveau du Parser, l’opération à effectuer est reconnue et la pile l’effectue en dépilant son ou ses derniers éléments.

**Difficultés rencontrées :**

Lors de ce projet, quelques problèmes ont été rencontrés et nous ont conduits à devoir laisser certaines erreurs ou imperfections dans notre code.

Pour la création d’onglet, après avoir étudié différentes solutions, nous avons opté pour l’implémentation d’une collection de piles pouvant ainsi permettre d’avoir plusieurs onglets avec plusieurs piles sans liens entre elles, simultanément. Cependant, il s’avère qu’à la création d’un nouvel onglet, la pile créée est et reste liée à la précédente. Nous n’avons malheureusement pas réussi à trouver la source de notre erreur.

La fonction DUP ne fonctionne pas sur le programme. Elle copie le dernier élément de la pile de façon tout à fait normal, mais lorsque nous essayons d’effectuer une opération ensuite, le programme échoue. Nous pensons que cela est dû au fait que le DUP crée un objet erroné mais n’avons pas réussi à trouver la source exacte du problème. Afin de limiter les éventuelles erreurs, nous avons décidé de laisser l’essai d’implémentation dans le projet mais de supprimer le bouton « DUP » de l’interface graphique.

Une autre erreur a été constatée dans l’utilisation de Undo/Redo. Malgré l’utilisation du Pattern Memento (dérivé du Pattern Command), l’état précédent ou suivant n’est pas toujours celui rafraichit. Pour mieux comprendre le problème, nous pouvons proposer un petit exemple :

**pile 1 : (3)**

**pile 2 : (6 3)**

*On fait '\*'* : **pile 3 : (18)**

*On fait ctrl + Z* : **pile 4 : (6 3)**

*Jusque là tout est normal*

**pile 5 : (2 6 3)**

*On fait '\*' :* **pile 6 : (12 3)**

*On fait ctrl + Z :* **pile 7 : (18)**

Les étapes enregistrées sont les suivantes :

1)**(6 3)**

2)**(18)**

3) **(2 6 3)**

4) **(12 3)**

Il faudrait que la classe supprime de sa mémoire « 18 » au moment où "\*" est effectué au niveau de la ***pile 5,*** et qu'il se replace ensuite sur (2 6 3), c’est à dire sur le dernier ***ctrl + Z.***

**Conclusion :**

La calculatrice à notation polonaise inversée proposée ici tente de répondre au mieux au cahier des charges imposé, tout en considérant et respectant les règles de conception du C++.

Nous avons donc essayé de rendre un projet très proche des exigences du sujet. Cependant certaines améliorations pourraient y être apportées d’une part par la correction des erreurs précédemment évoquées ; d’autre part, par l’ajout de fonctionnalités « confort » pour l’utilisateur (comme par exemple le fait de proposer la sauvegarde avant arrêt de la console).