Rapport laboratoire 6 POO

Tristan Gerber & Edison Sahitaj
28.11.2024

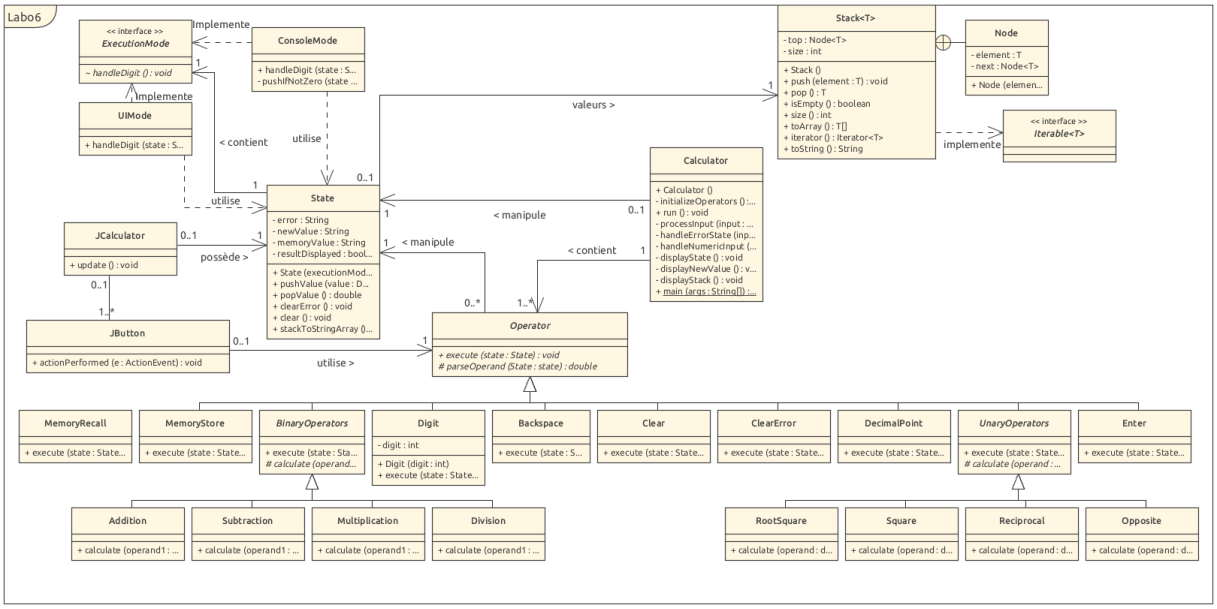
Contenu

1. Introduction

Ce laboratoire consiste à développer une calculatrice fonctionnant en notation polonaise inverse. Le projet inclut déjà une interface graphique, ainsi que la gestion de la logique sous-jacente pour garantir une extensibilité et une réutilisabilité du code.

2. Conception

2.1 Diagramme UML



2.2 Description des composants

2.2.1 JCalculator

Classe principale représentant l'interface graphique de l'application. Elle contient un attribut State pour récupérer le contenu de la pile par exemple et une méthode update pour mettre à jour l'affichage après chaque action.

2.2.2 State

Elle va permettre de gérer l'état interne de la calculatrice y compris la pile, les erreurs qu'il pourrait y avoir, les nombres qui peuvent être stockés en mémoire, la valeur actuelle ainsi que le mode d'exécution (graphique ou console). Il y a aussi un flag resultDisplayed qui permet de savoir si un résultat est affiché dans la vraible newValue permettant de push le résultat si un bouton d'un digit est appuyé.

2.2.3 Operator

Classe abstraite représentant chaque opération de la calculette. Elle a plusieurs sous-classes dont notamment la classe BinaryOperations pour des opérations demandant 2 opérandes comme l'addition et UnaryOperations qui demande quant à eux 1 opérande comme la racine carrée. Elle comporte aussi d'autre sous-classes qui ne font pas partie des 2 sous-classes d'avant comme ClearError ou bien Enter par exemple.

2.2.4 Calculator

Cette classe va gérer la logique pour que la calculette fonctionne en mode console en fonction les entrées de l'utilisateur. Elle réutilise la logique implémentée dans les classes State et Operator.

2.2.5 Stack<T>

Cette entité contient l'implémentation d'une pile générique de type LIFO qui est utilisée pour gérer les résultats intermédiaires. Elle fournit pas mal de méthode qui seront utilisée dans la classe State comme push ou pop par exemple.

2.2.6 BinaryOperations & UnaryOperations

Comme dit au-dessus, ces deux sous-classes de Operator ont été implémenté de sorte à différencier les opérations à 1 opérande et 2 opérandes.

2.2.7 ExecutionMode

Étant donné que le comportement de détails change entre l'application graphique et console, j'ai implémenté une interface qui contenant les différentes méthodes qui changent en fonction de l'exécution. Il y a donc 2 classes qui implémentent cette interface (UIMode pour la logique de l'application graphique et ConsoleMode pour l'application non-graphique)

2.3 Choix techniques

2.3.1 Structure modulaire avec héritage

Nous avons implémenté une sorte de hiérarchie pour les opérateurs (UnaryOperations et BinaryOperations) permettant de factoriser notre code pour chaque opération nécessitant des opérandes.

2.3.2 Gestion mode d'exécution

Au début nous avions une variable de type boolean dans State qui permettait de savoir si on était en mode console ou graphique, mais nous avons constaté qu'implémenter une interface comportant les méthodes qui changeaient entre les deux exécutions était plus propre et plus flexible.

2.3.3 Tests unitaires

Des tests unitaires pour la les différents cas de notre classe Stack<T> ainsi que l'application en mode console ont été implémentés mis à part pour la partie graphique qui est un peu plus difficile de faire des tests unitaires.

2.3.4 Gestion des erreurs

Il y a des différents endroits qui sont imbriqués dans un try catch notamment dans la classe Calculator pour permettre d'ajouter une erreur lorsque l'utilisateur ne rentre pas un nombre. Des erreurs sont aussi ajoutées dans la classe Division par exemple où là on regarde si le deuxième opérande n'est pas égal à 0 dans quel cas on throw une exception de type ArithmeticException où on ajoute une erreur.

2.3.5 Factorisation

On a pas mal factorisé notre code comme dans la classe Operator où on a implémenté une méthode parsOperand pour permettre de récupérer un opérande dans la pile ou dans newValue du fait que les classes UnaryOperators et BinaryOperators en ont besoins ainsi qu'aux autres sous-classes d'Operators.

3. Tests

3.1 StackTest

Ce tableau représente les différents tests qui sont implémenté dans la classe de test « StackTest» avec le résultat pour chaque sujet qui est attendu.

Sujet	Résulat attendu
Ajout de 10 et 20 à la pile via la méthode push	Pile comportant 10 et 20 comme valeurs
Suppression de la valeur en haut de la pile via la méthode pop	Retourne la valeur 20 (retirée)
Suppression d'une valeur sur une stack vide	EmptyStackException levé
Vérifier si la pile est vide via la méthode isEmpty	Retourne true
Récupérer la taille de la stack qui comporte 10 et 20 via size	Retourne 2
Transfomer la stack en un tableau d'objet via la méthode toArray	Retourne un tableau d'objet avec les valeurs
Parcours via les itérateurs sur la stack	Retourne les éléments dans l'ordre LIFO
Affichage en mode LIFO de la stack	Retourne un string des éléments dans l'ordre LIFO séparé par un ,

3.2 CalculatorTest

Ce tableau de test vérifie que le programme en mode console fonctionne correctement pout les différents cas traités

Sujet	Résulat attendu
Addition de deux nombres (5 + 3)	Retourne 8.0
Soustraction de deux nombres (4 - 10)	Retourne -6.0
Multiplication de deux nombres (-5 * 3)	Retourne -15.0
Division de deux nombres (10 / 2)	Retourne 5.0
Division par 0 (2 / 0)	Retourne erreur : Division by zero
Racine carré d'un nombre (9)	Retourne 3.0
Racine carré d'un nombre négatif (-9)	Retourne erreur : Square root of negative number
Nombre au carré (3)	Retourne 9
Inverser un nombre (4)	Retourne 0.25
Inverser zero (0)	Retourne erreur : Division by zero
Effacer l'erreur : Division by zero (CE)	Supprime Division by zero
Rénitialiser complètement l'état après une erreur (C)	Vide la pile et supprime l'erreur
Rénitialiser la pile (C)	Vide la pile
Stocker une valeur en mémoire (MS 5)	Valeur 5 stockée en mémoire et retirée de l'affichage
Récupérer une valeur stocker en mémoire (MR après MS avec 5)	Retourne 5.0 à l'affichage après MR
Empiler un opérateur avec une pile vide (5 +)	Retourne erreur : Stack is empty

3.3 Test application graphique

Comme je n'ai pas de tests unitaires pour cette partie, il y aura donc un tableau comportant les différents sujets testés, les étapes, les résultats attendus et les résultats obtenus.

Sujet	Etape	Résulat attendu	Résulat obtenu
Addition de deux nombres (5 + 3)	Appuie 3, ent, 5, +	Retourne 8.0	Correct
Addition de deux nombres contenus dans la pile (5 + 3)	Appuie 3, ent, 5, ent, +	Retourne 8.0	Correct
Soustraction de deux nombres (4 - 10)	Appuie 10, ent, 4, -	Retourne -6.0	Correct
Multiplication de deux nombres (-5 * 3)	Appuie 3, ent, 5, +/-, *	Retourne -15.0	Correct
Division de deux nombres (10 / 2)	Appuie 2, ent, 10, /	Retourne 5.0	Correct
Division par 0 (2 / 0)	Appuie ent, 2, /	Retourne erreur : Division by zero	Correct
Racine carré d'un nombre (9)	Appuie 9, sqrt	Retourne 3.0	Correct
Racine carré d'un nombre négatif (-9)	Appuie 9, +/-, sqrt	Retourne erreur : Square root of negative number	Correct
Nombre au carré (3)	Appuie 3, x^2	Retourne 9	Correct
Inverser un nombre (4)	Appuie 4, 1/x	Retourne 0.25	Correct
Inverser zero (0)	Appuie 1/x	Retourne erreur : Division by zero	Correct
Effacer l'erreur : Division by zero (CE)	Appuie ent, /, CE	Supprime Division by zero	Correct
Rénitialiser complètement l'état après une erreur (C)	Appuie ent, /, C	Vide la pile et supprime l'erreur	Correct
Rénitialiser la pile (C)	Appuie 2, ent, 1, ent, C	Vide la pile	Correct
Stocker une valeur en mémoire (MS 5)	Appuie 5, MS	Valeur 5 stockée en mémoire et retirée de l'affichage	Correct
Récupérer une valeur stocker en mémoire (MR après MS avec 5)	Appuie 5, MS, MR	Retourne 5.0 à l'affichage après MR	Correct
Empiler un opérateur avec une pile vide (5 +)	Appuie 5, +	Retourne erreur : Stack is empty	Correct
Opposé d'une valeur (5)	Appuie 5, +/-	Retourne -5	Correct
Enter sans valeur	Appuie ent	Retourne 0.0	Correct
Ajouter decimal à une valeur (3.5)	Appuie 3, ., 5, ent	Retourne 3.5	Correct
Effacer une décimal (3.5)	Appuie, 3, ., 5, <=, ent	Retourne 3.0	Correct

4. Conclusion

En conclusion, ce projet a permis de développer une calculatrice fonctionnelle en mettant en œuvre une architecture claire et modulaire.