POO - Laboratoire 7

Calculatrice

Lestiboudois Maxime & Parisod Nathan

27/11/2024

Contents

Introduction 2
Cahier des charges 2
1. Objectif du Projet 2
2. Spécifications Fonctionnelles 2
2.2 Mode Console 3
2.3 Gestion des Données et Opérations 3
3. Spécifications Techniques 3
4. Contraintes 4
5. Tests 4
Schéma UML 5
Listing du code 6
Choix de conception 6
Choix 1 - Les opérateurs 6
Choix 2 - Opérations arithmétiques 6
Choix 3 - Gestion d'écriture et de mémoire 6
Choix 4 - State et Operator 6
Choix 5 - Stack 6
Tests effectués 6
Conclusion 6

Introduction

Dans ce laboratoire, nous avons implémenté la logique d'une calculatrice utilisant la notation polonaise inversée. L'interface nous étant déjà fournie, nous nous sommes concentrés sur l'aspect fonctionnel de la calculatrice.
Nous avons également implémenter une extension de notre calculatrice afin de pouvoir l'utiliser directement dans un terminal.

Cahier des charges

1. Objectif du Projet

Créer une calculatrice fonctionnant en notation polonaise inverse (Reverse Polish Notation - RPN). Elle doit inclure une interface graphique (GUI) et un mode console, réutilisant les mêmes classes et principes. Le projet doit suivre une architecture Modèle-Vue-Contrôleur (MVC).

2. Spécifications Fonctionnelles

2.1 Interface Graphique (GUI)

- Implémenter une interface utilisateur dans la classe JCalculator.
- Boutons requis :
  - Chiffres : 0-9.
  - Opérations unaires : sqrt, 1/x, `x^2`.
  - Opérations binaires : +, -, *, /.
  - Autres boutons :
    - MR : récupérer une valeur en mémoire.
    - MS : stocker une valeur en mémoire.
    - `<=` : backspace pour supprimer le dernier caractère.
    - CE : réinitialiser l'affichage.
    - C : réinitialiser l'affichage et vider la pile.
    - Ent : placer la valeur courante sur la pile.
    - +/- : changer le signe de la valeur courante.
    - . : pour les nombres décimaux.
- Affichage :
  - Zone de texte (JTextField) pour la valeur courante.
  - Liste (JList) pour visualiser la pile.
- Mise à jour de l'affichage après chaque opération via une méthode update().

2.2 Mode Console

- Développer une classe Calculator permettant une interaction textuelle.
- Commandes utilisateur :
  - Saisir un nombre pour l'ajouter à la pile.
  - Entrer une opération (+, sqrt, etc.).
  - exit : quitter la calculatrice.
- Fonctionnalités similaires à la version graphique :
  - Gestion de la pile.
  - Support des opérations unaires, binaires et de mémoire.

2.3 Gestion des Données et Opérations

- Pile personnalisée (Stack) :
  - Empiler une valeur.
  - Désempiler une valeur.
  - Obtenir l'état actuel de la pile sous forme de tableau.
  - Fournir un itérateur pour parcourir la pile.
  - Implémentée avec une liste chaînée sans structures Java préconstruites.
- État interne (State) :
  - Stocker :
    - Valeur courante.
    - Pile des valeurs.
    - État d'erreur.
  - Fournir des méthodes pour gérer et manipuler ces données.

3. Spécifications Techniques

3.1 Architecture MVC

- Modèle (State) :
  - Indépendant de l'interface graphique.
  - Stocke les données et implémente la logique de calcul.
- Vue (JCalculator) :
  - Interface utilisateur graphique basée sur Swing.
  - Réagit aux changements dans le modèle.

- Contrôleurs (Operator) :
  - Boutons dans l'interface graphique agissant comme des contrôleurs.
  - Appel de la méthode execute() d'un objet Operator.

3.2 Hiérarchie des Classes

- Classe générique Stack pour représenter une pile.
- Classe State pour l'état interne.
- Hiérarchie Operator :
  - Classe de base Operator :
    - Méthode execute().
  - Sous-classes spécialisées :
    - NumberOperator, AdditionOperator, SqrtOperator, etc.
- Classe JCalculator pour l'interface graphique.
- Classe Calculator pour le mode console.

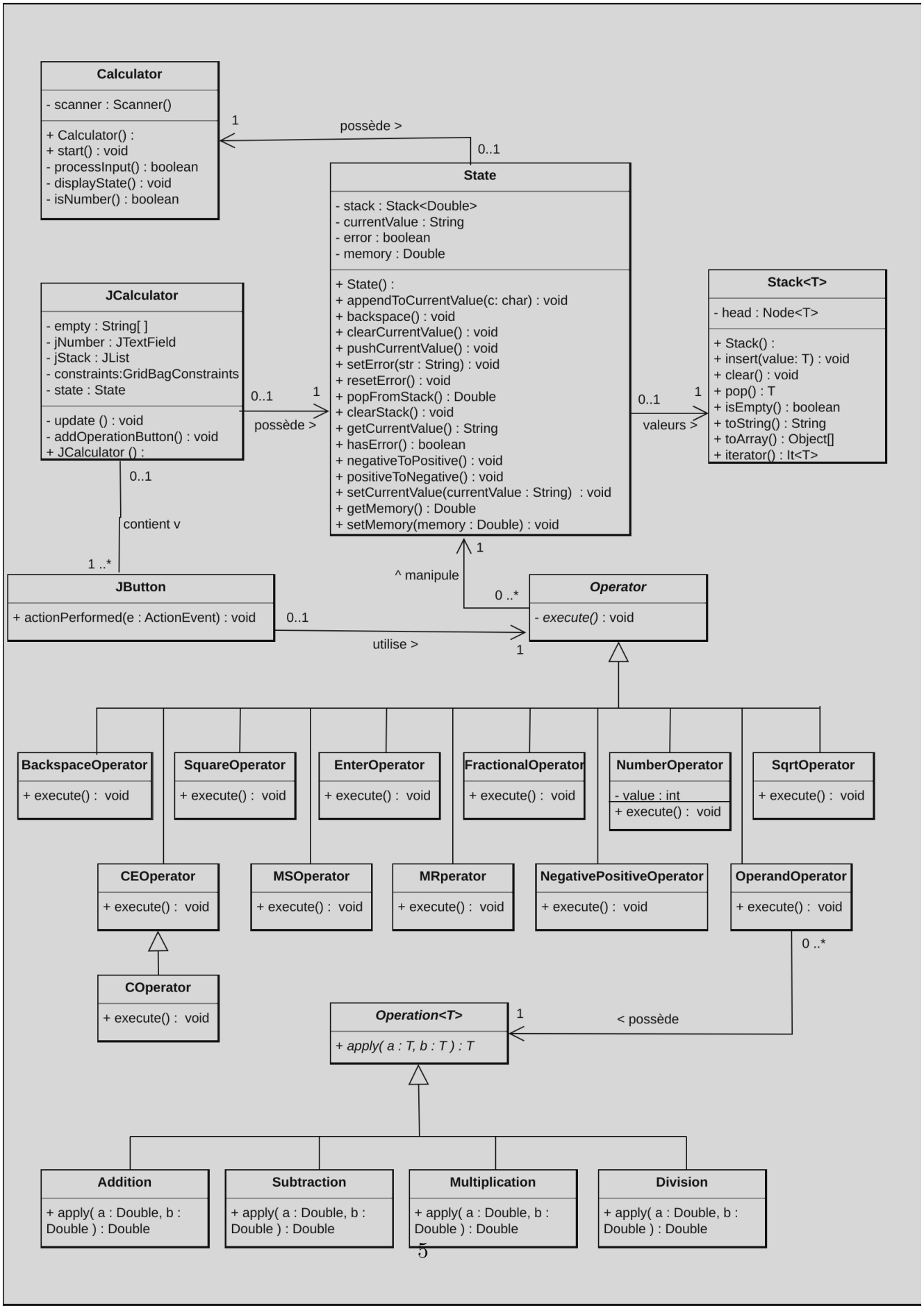
4. Contraintes

- Aucune utilisation de switch ou de if pour sélectionner l'opération dans Operator.
- Pas de propriétés statiques pour le stockage des données.
- Le code doit être modulaire et réutilisable.
- Respect des principes de conception objet.

5. Tests

- Élaborer une grille de tests couvrant :
  - Les opérations unaires et binaires.
  - Les erreurs (ex. : division par zéro, pile vide).
  - Le stockage et rappel de mémoire.
  - La compatibilité entre le mode console et GUI.
- Inclure des cas limites et des tests d'intégration.

Schéma UML


Calculator
- scanner : Scanner()
+ Calculator() :
+ start() : void
- processInput() : boolean
- displayState() : void
- isNumber() : boolean
1
possède >
0..1
State
- stack : Stack<Double>
- currentValue : String
- error : boolean
- memory : Double
+ State() :
+ appendToCurrentValue(c: char) : void
+ backspace() : void
+ clearCurrentValue() : void
+ pushCurrentValue() : void
+ setError(str : String) : void
+ resetError() : void
+ popFromStack() : Double
+ clearStack() : void
+ getCurrentValue() : String
+ hasError() : boolean
+ negativeToPositive() : void
+ positiveToNegative() : void
+ setCurrentValue(currentValue : String) : void
+ getMemory() : Double
+ setMemory(memory : Double) : void
JCalculator
- empty : String[ ]
- jNumber : JTextField
- jStack : JList
- constraints:GridBagConstraints
- state : State
- update () : void
- addOperationButton() : void
+ JCalculator () :
0..1
1
possède >
Stack<T>
- head : Node<T>
+ Stack() :
+ insert(value: T) : void
+ clear() : void
+ pop() : T
+ isEmpty() : boolean
+ toString() : String
+ toArray() : Object[]
+ iterator() : It<T>
0..1
1
valeurs >
0..1
contient v
1 ..*
^ manipule
1
0 ..*
JButton
+ actionPerformed(e : ActionEvent) : void
0..1
utilise >
1
Operator
- execute() : void
BackspaceOperator
+ execute() : void
SquareOperator
+ execute() : void
EnterOperator
+ execute() : void
FractionalOperator
+ execute() : void
NumberOperator
- value : int
+ execute() : void
SqrtOperator
+ execute() : void
CEOperator
+ execute() : void
MSOperator
+ execute() : void
MRperator
+ execute() : void
NegativePositiveOperator
+ execute() : void
OperandOperator
+ execute() : void
0 ..*
COperator
+ execute() : void
Operation<T>
+ apply(a : T, b : T) : T
1
< possède
Addition
+ apply(a : Double, b : Double) : Double
Subtraction
+ apply(a : Double, b : Double) : Double
Multiplication
+ apply(a : Double, b : Double) : Double
Division
+ apply(a : Double, b : Double) : Double

Listing du code

Choix de conception

Choix 1 - Les opérateurs

Nous avons reçu un dossier "starter" fourni par le corps professoral. Dans ce dossier, nous avons trouvé le fichier Operator, contenant la classe abstraite Operator, qui nous a servit de base pour construire tous les opérateurs (fonctionnalités des boutons) de notre calculatrice.
Nos opérateurs sont tous, sans exception, des classes enfants de la classe abstraite Operator, elles redéfinissent donc toutes la méthode abstraite execute() de la super-classe. Nous avons choisi que chaque opérateur, excepté les opérateurs d'opérations arithmétiques, serait un enfant direct de la super-classe Operator.
Par ailleurs, chaque classe-enfant ne comporte aucun attribut, excepté les classe NumberOperator et OperantOperator. NumberOperator possède un attribut privé et final. Ce choix nous permet de ne pas redéfinir une nouvelle classe pour chaque chiffre, ces derniers étant définis lors de l'instanciation de l'objet via le constructeur.
Nous discuterons de la classe Operand Operator dans le point suivant.

Choix 2 - Opérations arithmétiques

Choix 3 - Gestion d'écriture et de mémoire

Choix 4 - State et Operator

Choix 5 - Stack

Tests effectués

Conclusion