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 2
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 6
Choix 2 6
Choix 3 6
Tests effectués 6
Conclusion 6

Introduction

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.

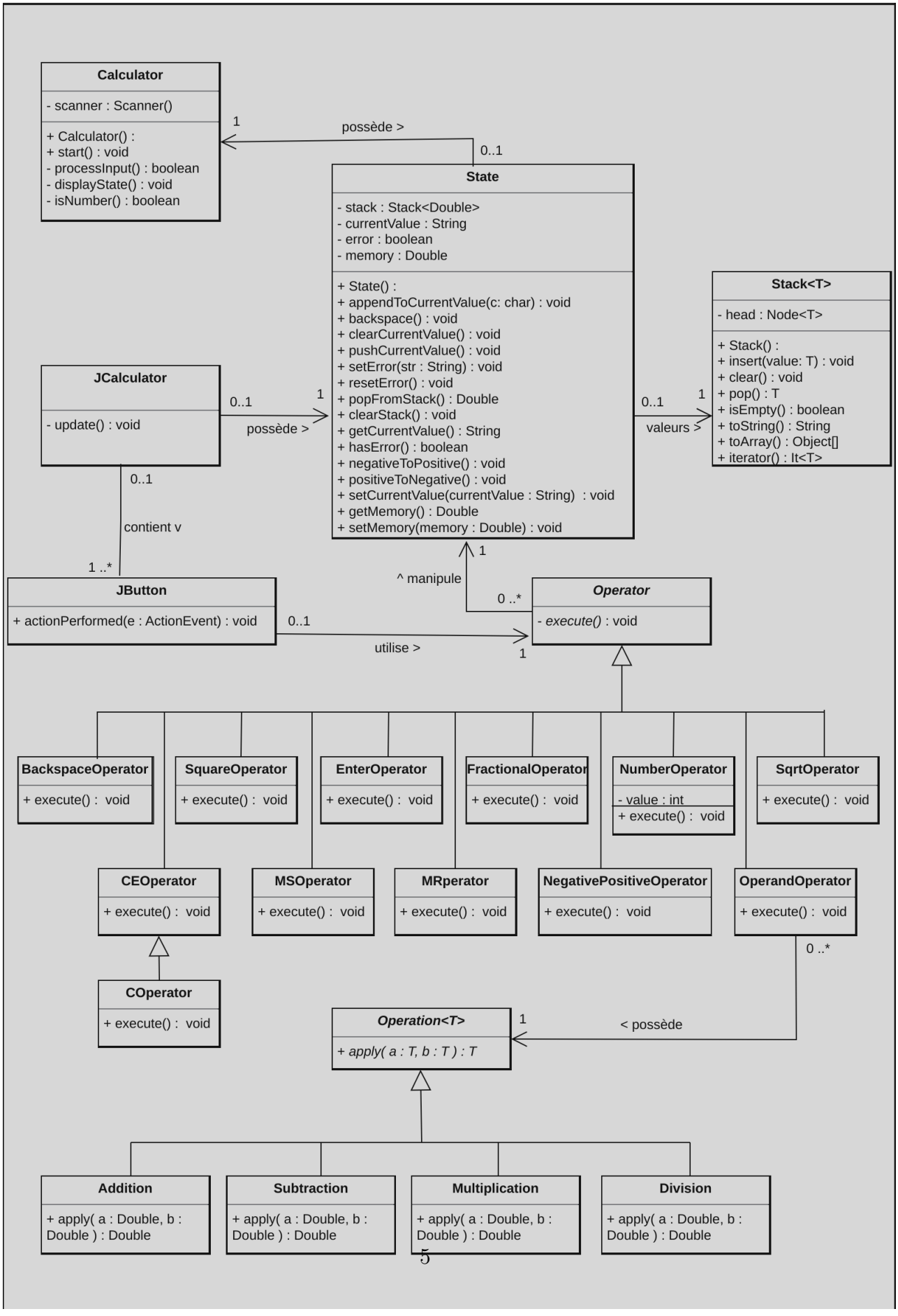
4. Contraintes

- 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
Stack<T>
- head : Node<T>
+ Stack() :
+ insert(value: T) : void
+ clear() : void
+ pop() : T
+ isEmpty() : boolean
+ toString() : String
+ toArray() : Object[]
+ iterator() : It<T>
JCalculator
- update() : void
0..1
1
possède >
0..1
1
valeurs >
0..1
contient v
1 ..*
1
^ manipule
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

Choix 2

Choix 3

Tests effectués

Conclusion