Labo7 - POO Calculatrice

Dani Tiago Faria dos Santos
Antoine Aubry

Groupe L02GrP
HEIG-VD

November 27, 2024

Contents

1 Choix de conception **3**
1.1 Operator 3
1.2 State 3

2 Modélisation UML **4**
2.1 Diagramme des classes 4

3 Tests **5**
3.1 Stack 5
3.2 Addition 5
3.3 Multiplication 5
3.4 Division 5
3.5 Opérations avancées 5
3.6 Réinitialisation 6
3.7 Résumé des résultats 6

4 Annexes **6**

1 Choix de conception

1.1 Operator

Pour la classe `Operator`, nous avons décidé de créer une hiérarchie en concevant des sous-classes abstraites pour chaque type d'opération. Cela permet de structurer et d'optimiser les opérations selon le principe de la séparation des responsabilités. Par exemple, la méthode `Digit()` est dédiée à la création d'un entier correspondant au bouton numérique pressé. Une fois le chiffre saisi, il est directement affiché via la méthode publique `update()`. Cette approche rend le code plus modulaire et facilite l'ajout de nouvelles fonctionnalités ou d'opérations dans le futur.

Chaque sous-classe implémente les comportements spécifiques à une opération donnée. Cela inclut les opérateurs arithmétiques de base (`Add`, `Subtract`, `Multiply`, `Divide`) ainsi que d'autres fonctionnalités comme la gestion des pourcentages, des puissances ou des opérations spécifiques définies par l'utilisateur.

1.2 State

La classe `State` est responsable de la gestion des valeurs et de la pile (`Stack`). Elle joue un rôle central dans la manipulation des données et leur affichage sur l'application Calculatrice. Voici ses principales responsabilités :

- **Gestion de la valeur affichée :**
  La variable publique `value`, de type `String`, représente le contenu actuellement affiché sur l'écran de la calculatrice. Les méthodes publiques de `State` permettent d'accéder directement à cette variable et de la modifier en fonction des actions effectuées par l'utilisateur.
- **Vérifications et gestion des erreurs :**
  Des méthodes spécifiques permettent de vérifier si un objet existe ou si sa valeur est `null`. Cela garantit une manipulation sécurisée des données et évite les erreurs inattendues lors de l'exécution.
- **Interaction avec la pile (`Stack`) :**
  La pile est utilisée pour gérer les opérations complexes nécessitant un stockage temporaire de valeurs intermédiaires. Des méthodes privées sont mises en place pour permettre à la classe `State` de vérifier et de manipuler l'état de la `Stack` sans exposer directement son contenu. Cette encapsulation renforce la sécurité et la robustesse de la logique.
- **Extensions et modularité :**
  La classe `State` est conçue de manière extensible, permettant d'ajouter de nouvelles fonctionnalités ou de modifier le comportement existant sans compromettre la stabilité du système. Par exemple, des méthodes pourraient être ajoutées pour prendre en charge des formats d'entrée plus complexes (nombres décimaux, exposants, etc.).

2 Modélisation UML

2.1 Diagramme des classes

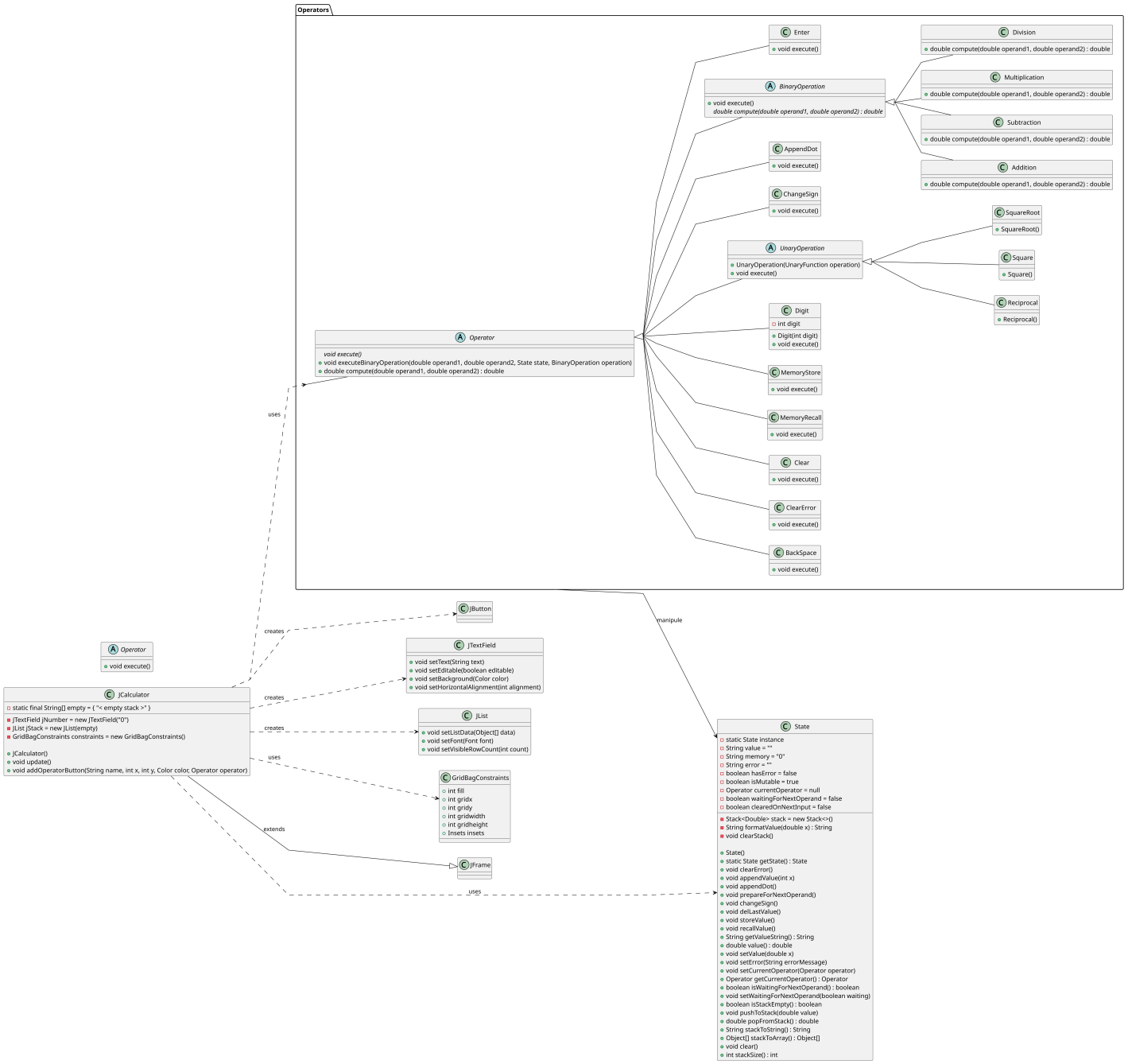

Figure 1: *Implémentation de la modélisation de la Calculatrice*

3 Tests

Les tests suivants ont été réalisés pour valider les fonctionnalités principales de la calculatrice.

3.1 Stack

- **Étape 1 : Empiler des valeurs et tester**
  Valeurs empilées : 10.0 et 5.0.
  Résultat attendu : `[5.0, 10.0]`
  Résultat obtenu : `[5.0, 10.0]`
  Test réussi.

3.2 Addition

- **Étape 2 : Addition (10 + 5)**
  Opération : Addition des deux valeurs empilées.
  Résultat attendu : `[15.0]`
  Résultat obtenu : `[15.0]`
  Test réussi.

3.3 Multiplication

- **Étape 3 : Empiler une nouvelle valeur et multiplier**
  Valeur ajoutée : 3.0.
  Opération : Multiplication (15.0 * 3.0).
  Résultat attendu : `[45.0]`
  Résultat obtenu : `[45.0]`
  Test réussi.

3.4 Division

- **Étape 4 : Diviser par une nouvelle valeur**
  Valeur ajoutée : 9.0.
  Opération : Division (45.0 / 9.0).
  Résultat attendu : `[5.0]`
  Résultat obtenu : `[5.0]`
  Test réussi.

3.5 Opérations avancées

- **Étape 5 : Racine carrée**
  Valeur ajoutée : 4.0.
  Opération : Racine carrée de 4.0.
  Résultat attendu : `[2.0, 5.0]`
  Résultat obtenu : `[2.0, 5.0]`
  Test réussi.

- **Étape 6 : Mise au carré**
  Opération : Mise au carré du sommet de la pile (2.0).
  Résultat attendu : `[4.0, 5.0]`
  Résultat obtenu : `[4.0, 5.0]`
  Test réussi.

- **Étape 7 : Inverse (1/x)**
  Opération : Calcul de l'inverse (1/4.0).
  Résultat attendu : `[0.25, 5.0]`
  Résultat obtenu : `[0.25, 5.0]`
  Test réussi.

- **Étape 8 : Combinaison complexe**
  Valeur ajoutée : 2.0.
  Opérations : Multiplication (0.25 * 2.0), ajout de 5.0, soustraction.
  Résultat attendu : `[-4.5, 5.0]`
  Résultat obtenu : `[-4.5, 5.0]`
  Test réussi.

3.6 Réinitialisation

- **Étape 9 : Réinitialisation de la pile**
  Opération : Effacement complet de la pile.
  Résultat attendu : `[]`
  Résultat obtenu : `[]`
  Test réussi.

3.7 Résumé des résultats

Tous les tests ont été exécutés avec succès, validant ainsi les principales fonctionnalités de la calculatrice, y compris les opérations arithmétiques de base, les opérations avancées, et la gestion correcte de l'état interne.

4 Annexes

- Code au format .pdf
- Code au format .java
- Schéma UML au format .jpeg et .svg