1 Introduction

Ce laboratoire a pour but de mettre en pratique les principes de la programmation orientée objet vus en cours et de les appliquer à la réalisation d'une calculatrice graphique et terminal. On utilise le principe de modèle-vue-contrôleur (MVC) pour implémenter notre calculatrice avec une classe `JCalculator` pour la version graphique et une classe `TCalculator` pour la version terminale. Nous avons aussi les classes `Operator` qui permettent de faire les opérations sur la calculatrice. Et pour finir les classes `State` et `Stack` qui permettent de stocker les données de la calculatrice et de les manipuler par les opérateurs.

2 Diagrammes de classes

2.1 Package `util`

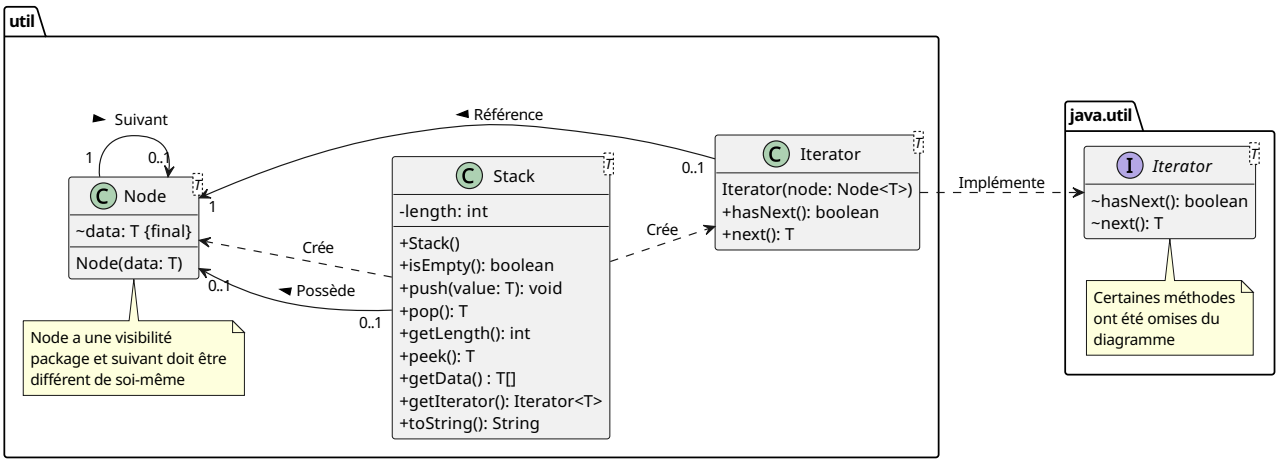

FIGURE 1 – Modélisation d'une pile générique.

2.2 Package `calculator`

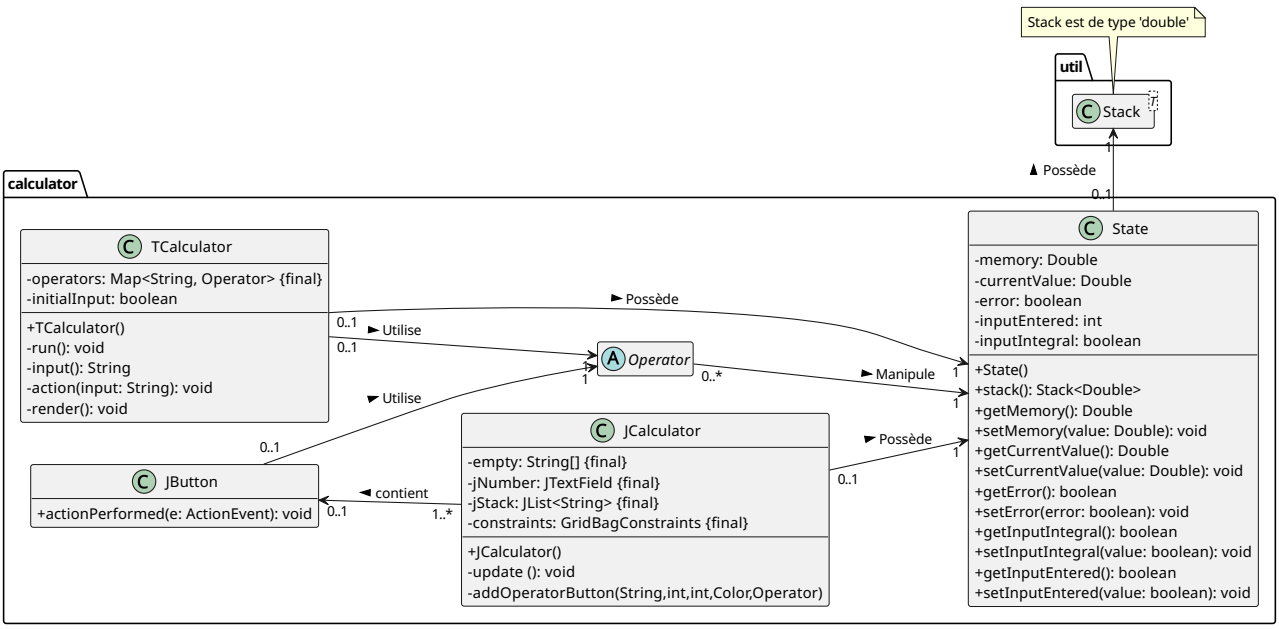

FIGURE 2 – Modélisation d'une calculatrice MVC.

N.B. La hiérarchie de classe de `Operator` est détaillée dans la figure 3.

2.3 Opérateurs

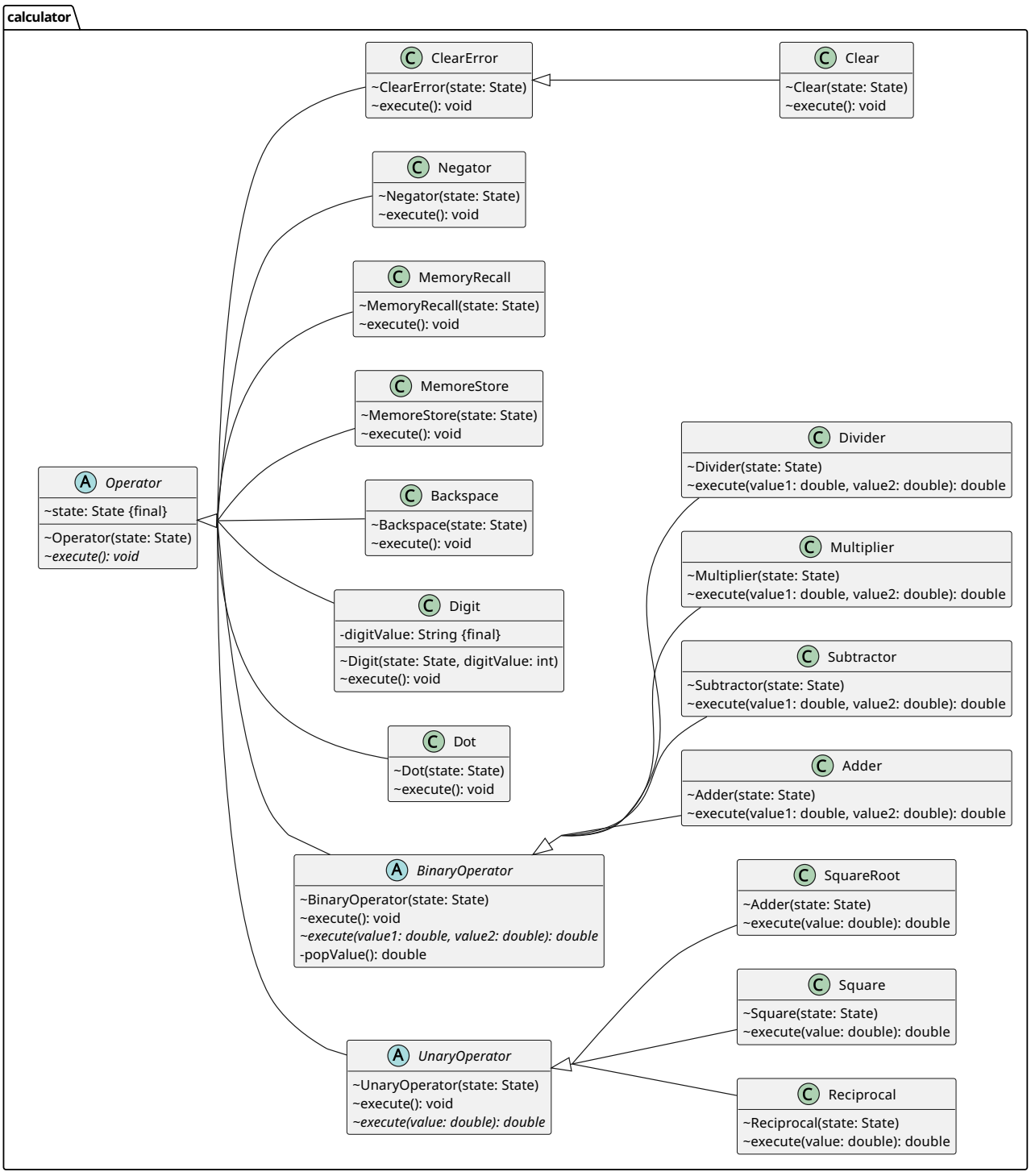

FIGURE 3 – Modélisation des opérateurs du pacakge `calculator`.

3 Implémentation

Le laboratoire a été réalisé avec les hypothèses de travail et choix d'implémentation suivants.

3.1 Modèle

Le modèle contient les données de la calculatrice et les méthodes permettant de les manipuler. Les membres sont volontairement déclarés `private` pour éviter qu'ils soient modifiés directement depuis l'extérieur de la

classe et force l'utilisation des accesseurs. Ce choix est fait afin d'améliorer l'encapsulation des données et la séparation des préoccupations. Accessoirement, cela permet de faciliter l'extension du programme pour, par exemple, rajouter des contrôles lors de l'écriture des valeurs dans le modèle.

Des membres en lien avec la saisie des données ont aussi été ajoutés au modèle afin de faciliter l'implémentation des opérateurs liés à la saisie de nombres (par exemple les classe `Digit`, `Dot` ou encore les classes de la famille `Memory`) sans avoir à recourir à des variables statiques dans le *controller*. Ces membres sont déclarés `private` pour les mêmes raisons que les autres membres du modèle.

3.2 Contrôlleurs

La classe `Operator` est étendue par les classes `UnaryOperator` et `BinaryOperator` qui implémentent respectivement l'exécution d'opérations à un et deux opérandes. Cette factorisation permet de réutiliser le code commun, notamment lors de l'écriture du résultat obtenu dans le modèle, ainsi que de simplifier l'ajout de nouveaux opérateurs à un ou deux opérandes.

Les opérateurs liés à la saisie de nombres (par exemple la classe `Digit` ou encore les classes de la famille `Memory`) héritent quant à elles directement de la classe `Operator` car elles ne font que manipuler des attributs spécifiques du modèle.

Les constructeurs réutilisent autant que possible le constructeur de la classe parente pour éviter la duplication de code.

3.3 Classes `Stack` et `Iterator`

Les classes classes `Stack` et `Iterator` sont fortement inspirées de l'exercice simpleList que nous avons vu durant notre cours de programmation orienté objet où nous avons dû implémenter une liste chaînée. Il y a tout de même plusieurs différences étant donné que dans le context de ce laboratoire, c'est une `stack` que nous devons implémenter et non une liste chaînée. Les classes suivantes sont utilisées pour implémenter la pile :

- La classe `Node` qui encapsule les données et contient un pointeur vers le prochain élément de la stack. Cette classe n'est pas destinée à être instanciée par l'utilisateur final.
- La classe `Iterator` qui implémente `java.util.Iterator` est la classe qui permet de parcourir la stack grâce à un itérateur. Elle implémente les méthodes *hasNext* et *next*. Normalement, un itérateur contient aussi une méthode *remove* qui permet de supprimer un élément de la stack mais nous avons fait le choix de pas implémenter cette fonction, car supprimer un élément de la stack à un emplacement autre que le début est à l'encontre du principe de la stack. Seul la méthode *pop* devrait pouvoir supprimer un élément de la stack et le premier élément de celle-ci.
- La classe `Stack` qui est la classe principale et qui contient les méthodes *push*, *pop*, *peek*, *isEmpty*, *getLength*, *getData* et *getIterator*. Comme choix d'implémentations, nous avons décidé de faire la méthode *peek* qui permet de récupérer la valeur du premier élément de la stack sans le supprimer. Cette fonctionnalité est très utile en général lors de l'implémentation d'une stack afin de connaitre l'élément qui se trouve au sommet de la stack sans le supprimer. Nous l'avons utilisé lors des tests, mais elle n'est pas utilisée dans le code de la calculatrice.

3.4 Accessibilité des classes

Les classes `JCalculator` et `TCalculator` sont déclarées `public` car elles sont destinées à être instanciées par l'utilisateur final.

Le package `util` n'expose que la classe `Stack` et son itérateur `Iterator`. La classe `Iterator` ne peut cependant pas être instanciée hors du package.

Les classes des couches *modèle* (classe `State`) et *controller* (sous-classes de `Operator`) sont déclarées avec une visibilité `package` car elles sont destinées à être utilisées uniquement par les classes de la couche *view* (classes `JCalculator` et `TCalculator`) du modèle MVC.

L'une des contraintes de la donnée du laboratoire est que la méthode `execute` de la classe `Operator` doit avoir une visibilité `package`. Cette visibilité s'étend donc aux autres méthodes, constructeurs et classes du projet puisqu'une visibilité moins restrictive ferait que l'utilisateur désireux d'étendre l'application ne pourrait pas réutiliser l'une des méthodes charnières du projet. Il a donc été décidé que si, par contrainte, les classes du *controller* ne peuvent pas être accessibles hors du package, alors le modèle ne doit pas l'être non-plus.

Par ailleurs, déclarer le modèle public pour qu'il soit réutilisable par d'autres applications n'aurait pas de sens car il est spécifique à cette calculatrice. De plus, il n'est pas destiné à être accédé directement par l'utilisateur final, mais seulement par les classes de la couche *controller*.

Les classes `Operator`, `UnaryOperator` et `BinaryOperator` sont déclarées `abstract` car elles sont seulement héritables et pas instanciables.

Les classes d'opérateurs (a l'excéption des classes abstraites et de la classe `ClearError`) sont déclarées `final` car elles ne sont pas destinées à être héritées puisqu'elles sont fortement couplées à une fonctionnalité spécifique implémentée par la calculatrice. Bien que certaines opérations puissent potentiellement être réutilisées pour faciliter l'ajout de nouvelles fonctionnalités, il a ete decide de ne par les rendre extensibles pour eviler de rendre le code trop complexe et ainsi difficile à maintenir.

3.5 Remarques

Il est à noter qu'il serait souhaitable de réstreindre la visibilité des constructeurs des classes d'opérateurs abstraits aux sous-classes, mais que cela n'est pas possible en Java car la visibilité `protected` rend une variable accessible au package. Il en va de même pour la variable membre de la classe `Operator`.

4 Tests

Des tests unitaires réalisés avec `JUnit` valident le fonctionnement des classes implémentées.

4.1 Classe `Stack`

Les tests de la stack sont les suivants :
- La pile est correctement mise à jour lors de l'ajout d'un élément.
- La pile est correctement mise à jour lors de la suppression d'un élément et que la pile reste inchangée et une exception est lancée lorsque la pile est vide.
- Le sommet de la pile est correctement retourné sans que la pile soit changée et une exception est lancée lorsque la pile est vide.
- La taille de la pile est correctement retournée.
- Les données de la pile sont correctement retournées sous forme d'un tableau et les éléments pile ne sont pas exposés au travers de ce tableau.
- La pile est correctement représentée sous forme de chaîne de caractères.

4.2 Classe `Iterator`

Les tests de la classe `Iterator` sont les suivants :
- La méthode `hasNext` retourne la valeur attendue.
- La méthode `next` retourne la valeur attendue et la pile reste inchangée, ou une exception est lancée lorsque la pile est vide.

4.3 Opérateurs

Les tests réalisés sur les opérateurs servent à valider que les opérations sont correctement effectuées sur le modèle (ici la classe `State`) indépendamment de la variante (graphique ou console) de calculatrice utilisée.

Les tests des opérateurs sont les suivants :
- L'addition est correctement effectuée sur le modèle.
- La division est correctement effectuée sur le modèle et la division par zéro est correctement gérée.
- La soustraction est correctement effectuée sur le modèle.
- Le carré est correctement effectué sur le modèle.
- La multiplication est correctement effectuée sur le modèle.
- L'inverse est correctement effectué sur le modèle et l'inverse de zéro est correctement gérée.
- La racine carrée est correctement effectuée sur le modèle et la racine d'un nombre négatif est correctement gérée.
- Le stockage et la récupération de la mémoire sont correctement effectués sur le modèle.
- Le nettoyage de l'erreur et la réinitialisation des variables d'état sont correctement effectués sur le modèle.
- La réinitialisation de la pile est correctement effectuée sur le modèle.
- La valeur est correctement empilée dans le modèle.
- Le point est correctement ajouté à la valeur courante du modèle et les entrées subséquentes sont correctement ajoutées à la partie décimale de la valeur courante dans le modèle.
- La valeur courante du modèle est correctement inversée, sauf si la valeur est zéro, dans quel cas elle reste inchangée.
- La valeur courante du modèle est correctement mise à jour avec la valeur de l'opérateur.
- La valeur courante du modèle est correctement mise à jour avec la nouvelle valeur obtenue suite à la suppression d'un chiffre avant ou après la virgule.

De plus, le test `sequenceTest` vérifie que l'application ait un comportement identique à celui présenté dans l'énoncé du laboratoire étant donné la même sequence d'opérations.

4.4 Calculatrice graphique

Les tests menés dans l'interface graphique sont les suivants :
— Les boutons sont correctement liés aux opérations.
— L'affichage s'effectue correctement.
— Les opérations sont correctement effectuées.
— Une erreur est correctement affichée.

4.5 Calculatrice console

Les tests menés dans l'interface console sont les suivants :
— Les opérations sont correctement effectuées.
— L'erreur est correctement affichée.
— La saisie d'une opération invalide affiche un message d'erreur.
— La saisie valide d'un nombre décimal en un seul argument est correctement effectuée.