**Rapport laboratoire 4 POO**

Tristan Gerber & Edison Sahitaj

30.10.2024

Contenu

[1. Introduction 2](#_Toc181206519)

[2. Conception 2](#_Toc181206520)

[2.1 Diagramme UML 2](#_Toc181206521)

[2.2 Description des composants 2](#_Toc181206522)

[**2.2.1** **Operation** 2](#_Toc181206523)

[**2.2.2** **Sous-classes (Addition, Subtraction et Multiplication)** 3](#_Toc181206524)

[**2.2.3** **Matrix** 3](#_Toc181206525)

[2.3 Choix techniques 3](#_Toc181206526)

[**2.3.1** **Attributs** 3](#_Toc181206527)

[**2.3.2** **Relation « Matrix »** 3](#_Toc181206528)

[**2.3.3** **Matrice vide** 3](#_Toc181206529)

[**2.3.4** **Méthode checkModulus()** 3](#_Toc181206530)

[**2.3.5** **Tests unitaires** 3](#_Toc181206531)

[**2.3.6** **Gestion d’exceptions** 4](#_Toc181206532)

[3. Tests 4](#_Toc181206533)

[3.1 MatrixTest 4](#_Toc181206534)

[3.2 AppTest 4](#_Toc181206535)

[4. Conclusion 4](#_Toc181206536)

# **Introduction**

Dans ce laboratoire, nous devons gérer des opérations entre matrices en utilisant des concepts vus en cours de POO. Le laboratoire est simple jusqu’à ce qu’on y ajoute des contraintes :

* Factoriser le code commun aux différentes opérations (addition, soustraction et multiplication) de manière qu’il soit possible d’en ajouter de nouvelles ultérieurement.
* Définir des objets représentant l’opération à effectuer sur les éléments des matrices opérandes et ceci sans utiliser de structures de contrôle (ifs, switchs…).
* Tester les fonctionnalités implémentées, en particulier les cas limites.

# **Conception**

Puis qu’on doit factoriser le code commun aux différentes opérations et définir des objets pour les opérations, alors il serait logique de créer une classe pour chaque opération, héritant d’une classe abstraite Opération. Et une classe Matrix qui contiendra toutes les méthodes dont on a besoin pour créer et faire le calcul des matrices.

## **Diagramme UML**

Une image contenant texte, capture d’écran, nombre, Parallèle

Description générée automatiquement

## **Description des composants**

### **Operation**

L’entité « Operation » définit une classe abstraite pour les opérations mathématique entre deux entiers. Elle possède un attribut « symbol » qui permettra d’identifier l’opération qui sera effectué en y insérant « + », « - » ou « x ». Elle contient aussi une méthode abstraite « apply » qui sera redéfini dans les sous-classes pour exécuter une opération d’addition, de soustraction ou de multiplication. La méthode « toString » quant à elle retourne la représentation en chaîne de caractère du symbole de l’opération.

### **Sous-classes (Addition, Subtraction et Multiplication)**

Les sous-classes d’operation implémente chacune une opération mathématique spécifique. Elle comporte un constructeur pour changer l’attribut « symbol » hérité par le parent pour indiquer l’opération. Elle redéfinisse aussi la méthode « apply » pour réaliser respectivement l’addition, la soustraction et la multiplication.

### **Matrix**

La classe « Matrix » va contenir un attribut « rows », « columns » pour les lignes et les colonnes de la matrice et « modulus » pour définir un modulo utilisé dans les opérations. L’attribut « matrix » définit la structure principale où seront enregistrées les valeurs de chaque élément accessibles et manipulables lors des opérations. L’entité contient aussi plusieurs méthodes, comme « applyOperation » qui va permettre d’effectuer une opération sur une matrice à partir des paramètres fournis. « checkModulus » va vérifier la compatibilité des modulos qui sera utilisé dans la méthode cité juste avant. La méthode « toString » va quant à elle afficher le contenu d’une matrice. De plus, la classe contient 2 constructeurs, une qui va permettre de créer une matrice à partir d’un tableau 2D et un modulo fourni et l’autre pour créer une matrice avec des valeurs générées aléatoirement allant de 0 à modulo – 1 à partir des lignes, colonnes et d’un modulo fourni.

## **Choix techniques**

### **Attributs**

Nous avons décidé de rendre nos attributs de type « final » de la classe « Matrix » pour les rendre immuable. Ces attributs sont « private » pour conserver l’encapsulations. L’attribut de la classe « Operation » est « protected » pour permettre seulement aux sous-classes du même package d’y accéder.

### **Relation « Matrix »**

L’entité a une relation de dépendance étant donné que « Matrix » utilise juste la classe « Operation » de façon temporaire pour appliquer une opération dans une méthode sans avoir besoin de stocker un lien permanent avec une quelconque classe.

### **Matrice vide**

Nous avons décidé d’interdire la construction d’une matrice vide étant donné que la consigne ne donne pas d’informations concernant cela. De plus, cela ne fait pas de sens d’utiliser des matrices vides sur l’utilisation de notre programme.

### **Méthode checkModulus()**

L’ajout de cette méthode s’explique juste pour rendre le code un peu plus lisible étant donné que la vérification du modulo se fait sur quasiment toutes nos méthodes. De plus, la méthode est déclarée « private » car elle est seulement utilisée dans la classe elle-même. Celle-ci va seulement regarder si le modulo d’une matrice fournie est plus grand que 0, sinon une exception est levée.

### **Tests unitaires**

Nous avons implémenté des tests unitaires à l’aide de JUnit étant donné que nous avions déjà utilisé cette technologie auparavant comme dans un laboratoire précédemment fait au cours de DAI. Le projet contient 2 fichiers de tests, un pour les différents cas à tester sur les matrices et l’autre pour avoir un visuel sur la console des différents cas d’arguments fournie au programme.

### **Gestion d’exceptions**

Le programme « Main » est englobé par un try catch afin d’avoir un aperçu des différentes erreurs générées pour des arguments donnés non valides. Cela permet donc de faire les tests dans « AppTest ». Sur les différents cas d’erreurs, une exception de type « RuntimeException » est levée comme indiqué dans la consigne.

# **Tests**

## **MatrixTest**

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

Une image contenant texte, reçu, capture d’écran, document

Description générée automatiquement

## **AppTest**

Ce tableau de test vérifie que le programme affiche correctement les matrices avec des arguments valides et qu’il génère les messages d’erreur de « RuntimeException » lorsqu’un argument n’est pas numérique, au format incorrect ou manquant.

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

# **Conclusion**

En conclusion, ce projet nous a permis de mettre en pratique les principes de l’orientation objet en favorisant la modularité et la flexibilité du code.