HEIG-VD — POO

Laboratoire 5 (Matrices) – Compte-rendu

[redacted]

20 octobre 2023

Ce document décrit nos choix et hypothèses de travail effectués pour réaliser le laboratoire n° 5 dans lequel nous avons dû créer une classe permettant la modélisation de matrices de taille quelconque. En outre, cette classe doit permettre de gérer des opérations arithmétiques effectuées composante-par-composante.

1 Modélisation

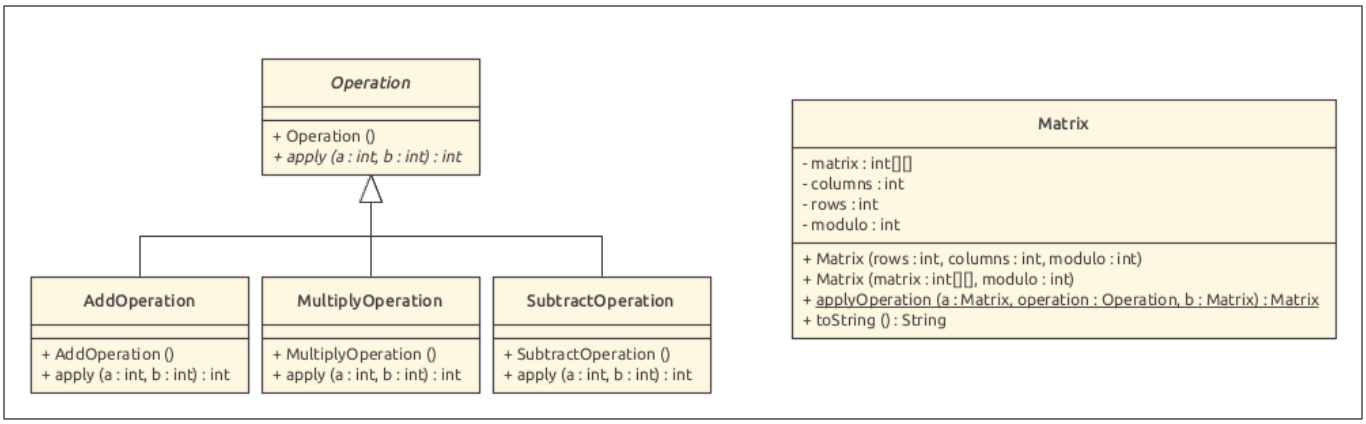

FIGURE 1 – Diagramme UML représentant les classes implémentées

2 Choix et hypothèses de travail

2.1 Matrices (*Matrix*)

La classe *Matrix* représente une matrice $m \times n; \forall m, n \in \mathbb{N}$. Les opérations sont effectuées composante-par-composante dans la classe de congruence du modulo fourni à l'initialisation de la matrice.

2.1.1 Attributs

Tous nos attributs sont marqués *final*, car nous souhaitions implémenter la matrice de manière immuable. Cela dit, si un besoin de modification survient, on retire facilement le *final* pour permettre la mutabilité. Évidemment, les attributs sont privés afin de conserver l'encapsulation. Aucun getter n'existe sur les attributs présents dans la version actuelle, car il n'y en a pas le besoin.

2.1.2 Constructeurs

La classe *Matrix* possède deux constructeurs, le premier permet de générer une matrice selon la taille et le modulo (n) donnés, et va la remplir de valeurs aléatoires réparties uniformément dans l'intervalle $[0, n[$. Ce constructeur vérifie donc que les valeurs pour les lignes et colonnes ne sont pas négatives, et que le modulo soit strictement positif.

Le second constructeur permet de faire une copie depuis un tableau en deux dimensions classique. Le nombre de lignes est la longueur du tableau donné. Le nombre de colonnes est le nombre d'éléments du tableau de la première ligne. Le modulo est aussi donné en paramètre et validé selon les mêmes règles que le premier constructeur.

Certaines validations sont faites sur le tableau donné en paramètre pour le second constructeur. Les deux premières validations confirment que toutes les lignes du tableau soit non-null et que leur taille corresponde à celle de la première ligne. Enfin, la troisième validation lance une exception si le chiffre à la position a_{ij} n'est pas dans l'intervalle $[0, n[$ où, pour rappel, n est le modulo de la matrice.

2.1.3 Opérations

Une seule méthode est fournie dans la classe *Matrix*, elle prend en paramètre deux matrices ainsi qu'une définition de la classe abstraite *Operation*. Pour l'instant, il est uniquement possible d'ajouter, soustraire et multiplier les composantes des deux matrices entre elles.
Le résultat de l'opération est rendu dans une nouvelle matrice, raison pour laquelle notre méthode *applyOperation* est statique.
On notera quand même qu'il aurait probablement été préférable de définir une interface fonctionnelle pour ces opérations avec quelques définitions par défaut pour celles qui sont demandées, mais cela s'éloigne du domaine orienté objet pour se rapprocher du fonctionnel.

2.1.4 *toString()*

Pour implémenter la génération du *string* de retour, nous avons préféré utiliser un *StringBuilder* qui permet de limiter la concaténation d'une multitude de *string*s dans les différentes itérations de la matrice. Cependant, dans le cas d'une matrice vide (0×0) le résultat retourné sera « [] » pour simplifier la lecture.

2.2 Définition des opérations (*Operation*)

Cette classe permet d'abstraire la notion d'opérations composante-par-composante de la matrice afin de factoriser la méthode propre à la classe *Matrix* en une seule définition réutilisable.
Elle ne demande pas de constructeur, si ce n'est que les implémentations doivent pouvoir être instanciées d'une certaine manière (par exemple, au travers du constructeur par défaut que nous avons laissé tel quel dans notre implémentation).
Les implémentations sont classiques et font ce qui est attendu dans la définition de la méthode *apply*. Nous les avons marquées comme *final* pour éviter des extensions qui n'auraient pas de sens.

3 Tests des cas limites

Avec la classe de test demandée, nous avons pu effectuer quelques vérifications des effets de bord. Nous avons également créé une autre classe de test (*TestingMain*) qui nous a permis de tester les scénarios ci-dessous.

Scénario	Résultat attendu
Création d'une matrice aléatoire avec un modulo nul ou négatif	*IllegalArgumentException* levée dans le constructeur
Création d'une matrice aléatoire avec nombre de lignes négatif	*IllegalArgumentException* levée dans le constructeur
Création d'une matrice aléatoire avec nombre de colonnes négatif	*IllegalArgumentException* levée dans le constructeur
Création par copie avec un modulo nul ou négatif	*IllegalArgumentException* levée dans le constructeur
Création par copie avec une ligne étant *null*	*IllegalArgumentException* levée dans le constructeur
Création par copie avec des lignes de longueurs différentes	*IllegalArgumentException* levée dans le constructeur
Création par copie avec des valeurs négatives	*IllegalArgumentException* levée dans le constructeur
Création par copie avec des valeurs supérieures/égales au modulo	*IllegalArgumentException* levée dans le constructeur
Création de matrices de taille 0×0	Aucune erreur lancée
Opération entre deux matrices n'ayant pas le même modulo	*IllegalArgumentException* levée lors de l'appel à la méthode
Opération entre deux matrices de tailles différentes	Les valeurs manquantes sont bien remplacées par 0
Opération entre une matrice de taille nulle et une autre	L'opération se déroule sans erreurs

Table 1 – Tests effectués avec la classe *Matrix*