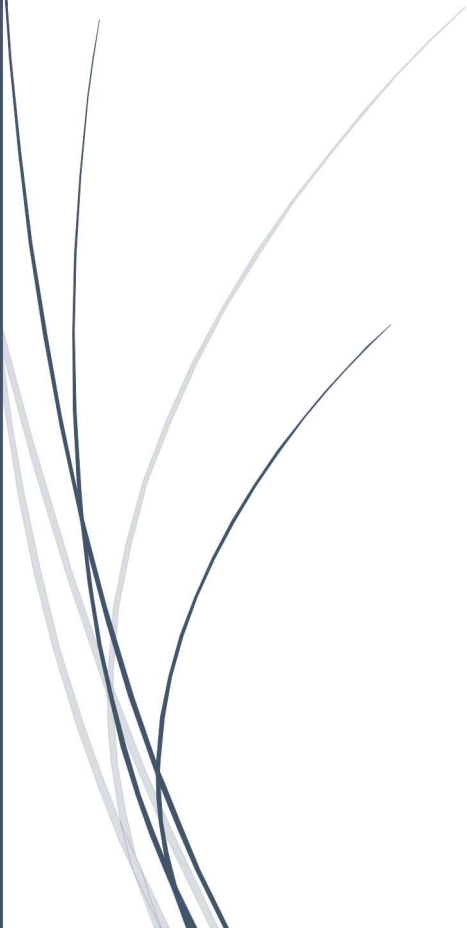
A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

17/03/2022

# Rapport Laboratoire 1 : Matrix Reloaded

Several thin, curved lines in shades of blue and grey originate from the bottom left and sweep upwards and to the right.

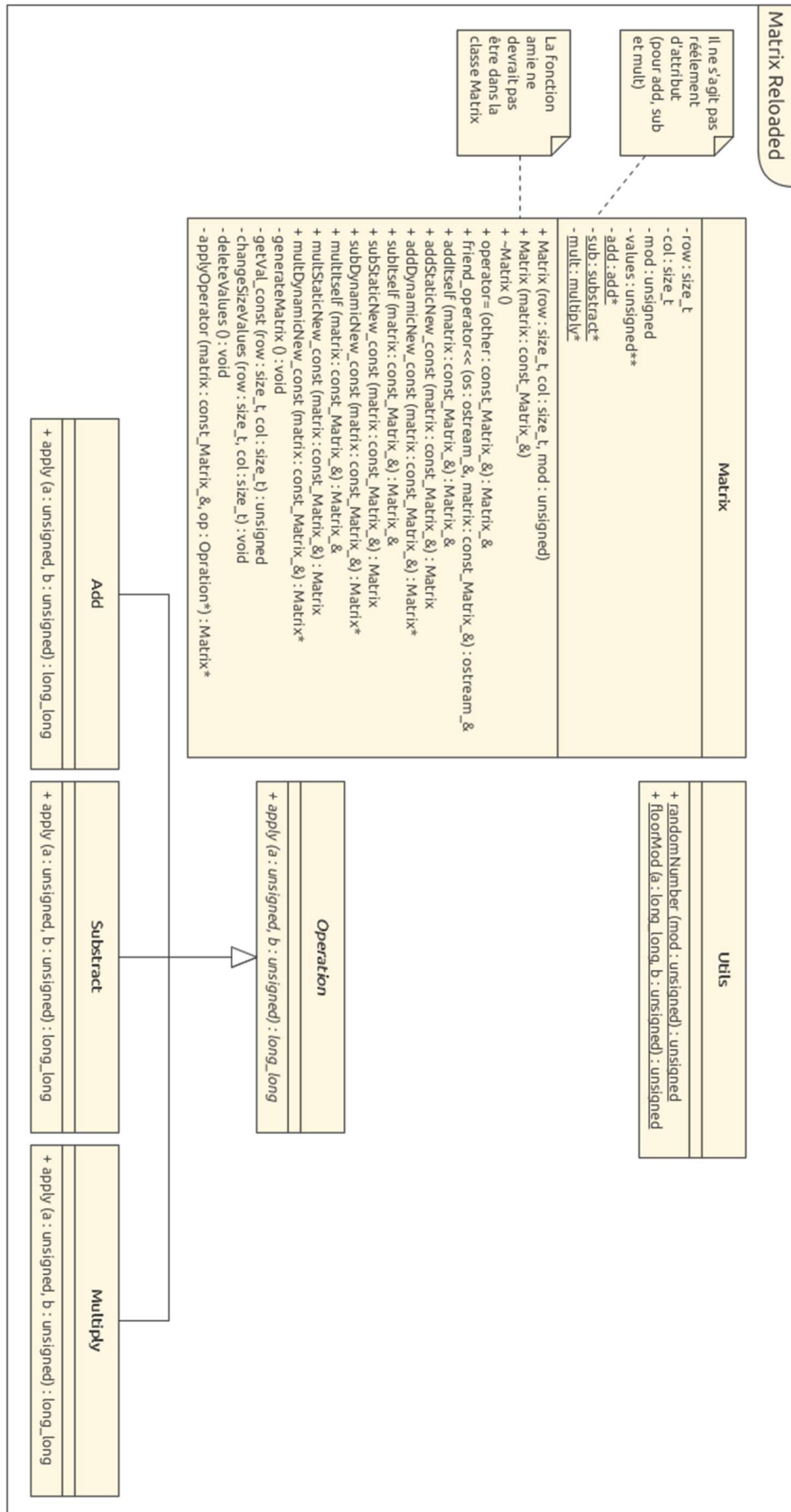
Alexandre Jaquier et Jonathan Friedli



## Introduction

Dans le cadre de ce laboratoire, nous devons implémenter une classe `matrix`, permettant de représenter des matrices de taille quelconque. Chaque matrice aura des valeurs qui seront modulo  $n$ ,  $n$  étant une valeur passée en paramètre à la création de la matrice. Il sera également possible de d'effectuer des opérations sur ces dernières, telles que l'addition, la soustraction et la multiplication par composante. Chaque opération doit pouvoir être fait de 3 manières différentes. Soit en modifiant la première matrice, soit en retournant par valeur une nouvelle matrice résultat, soit en retournant par pointeur une nouvelle matrice résultat allouée dynamiquement.

## Diagramme UML



## Choix de modélisation et d'implémentation

### Matrix

Les attributs de Matrix sont tous évidemment privés afin de préserver l'encapsulation. Le modulo, ainsi que les dimensions d'une matrice ne pouvant pas être négatif, nous avons choisi de leur mettre un type `unsigned`. Pour `row` et `col`, nous avons fait le choix du type `size_t` car c'est généralement le type que l'on donne aux tableaux. L'attribut `mod` quant à lui est simplement du type `unsigned int`.

Les matrices acceptent comme 0x0 comme taille minimale. Cela représente la matrice vide et toutes les opérations fonctionnent correctement sur la matrice vide. La taille maximale est quant à elle, limitée par la taille maximale qu'un tableau simple peut prendre en c++. Comme mentionné précédemment, la taille de la matrice ne peut pas être négative. Si nous tentons de mettre une valeur négative, il y aura un `underflow` et cela tentera de créer une matrice d'une taille de quelques milliards, ce qui résultera en un crash du programme.

Le modulo accepte 1 comme valeur minimale. En effet, appliquer l'opération modulo 0 sur un entier ne fait aucun sens. Nous aurions également pu prendre 2 comme valeur minimale, car si la valeur du modulo est égale à 1. Cela veut dire que toutes les valeurs de la matrice seront de 0 et cela rendrait la classe Matrix peu utile mais nous avons choisis de ne pas faire cela.

Les opérations ne peuvent être effectuées que sur deux matrices ayant la même valeur de modulo. Si les deux matrices n'ont pas la même taille, cela n'est pas grave, nous remplaçons les valeurs manquantes par 0.

Etant donné que chaque opération doit pouvoir être effectuée de 3 manières différentes, nous leur avons donné des noms précis. L'opération modifiant la première matrice, se nomme `« nomOpérationItself »`, cela donne `« addItself »` pour l'addition. La méthode retournant une matrice par valeur se nomme `« addStaticNew »` et celle retournant un pointeur sur une matrice résultat allouée dynamiquement, se nomme `« addDynamicNew »`.

Dans la consigne, il est demandé pourquoi dans le `« addStaticNew »` nous retournons la matrice par valeur et non pas par référence. C'est parce que si nous renvoyons une référence, une fois que nous quittons le corps de la méthode, cette référence pointerait sur une matrice qui n'existe plus.

### Opération

Afin de rendre l'ajout d'une autre opération (ex : la division) rapide et intuitif, nous avons créé une classe abstraite Opération. Chaque opération doit donc hériter de cette classe abstraite et réimplémenter la méthode `apply`. Cette dernière permet de calculer le résultat de l'opération entre deux éléments de matrice.

Les différentes opérations prennent en paramètre le même type que les éléments se trouvant dans les matrices, à savoir des `« unsigned int »` et renvoie un `« long long »` signé. Cela est dû au fait que les opérations posent des problèmes d'`underflow` et d'`overflow`, par la nature non-signée du type. Nous castons donc les paramètres en `« long long »` avant d'effectuer l'opération afin d'éviter ce problème.

## Utils

Nous avons regroupé deux fonctions utilitaires dans cette classe. Il s'agit des méthodes `randomNumber`, permettant de générer un entier aléatoire compris entre 0 et  $n-1$ ,  $n$  étant passé en paramètre. La deuxième méthode est `floorMod`. C'est une implémentation plus stricte de l'opérateur modulo. En effet, en c++, si l'on effectue l'opération suivante : «  $-10 \% 4$  », le résultat sera -2. Alors que mathématiquement le résultat d'un modulo est toujours positif ou nul.

## Tests effectués

Nous avons créé une méthode de test en plus du petit programme demandé dans la consigne.

Test	Résultat attendu
La création d'une matrice de taille 0x0 doit être possible. Si une des composantes de la taille est négative, une exception se lance.	OK
La création d'une matrice avec un modulo valant 0 ne doit pas être possible	OK
Les opérations entre deux matrices sont possibles uniquement si les deux matrices ont le même modulo.	OK
Les opérations entre deux matrices de taille différentes sont possibles si leur modulo est pareil.	OK
Les valeurs de la matrice ne sont jamais négatives. En particulier après une soustractions entre deux matrices.	OK
Il est possible d'écrire le contenu d'une matrice grâce à l'opérateur d'écriture de flux <<	OK
Il est possible d'affecter une matrice à une autre matrice via l'opérateur d'affectation =	OK
Il est possible de créer copie d'une matrice en passant cette dernière comme paramètre.	OK