TD04 – Développement Piloté par les Tests

L'objectif de ce TD est de développer une classe de gestion de matrices en étant sûr qu'elle réponde à ses spécifications.

Contexte

Afin d'effectuer des calculs matriciels dans différents programmes (transformations 2D, changements de repères 3D, etc…) nous souhaitons développer un composant logiciel de gestion de matrices. Ce composant sera ensuite intégré dans les différents programmes. Ce composant sera développé sous la forme d'une classe **`CMatrice`** ayant les caractéristiques définies par le diagramme UML de la figure 1 :

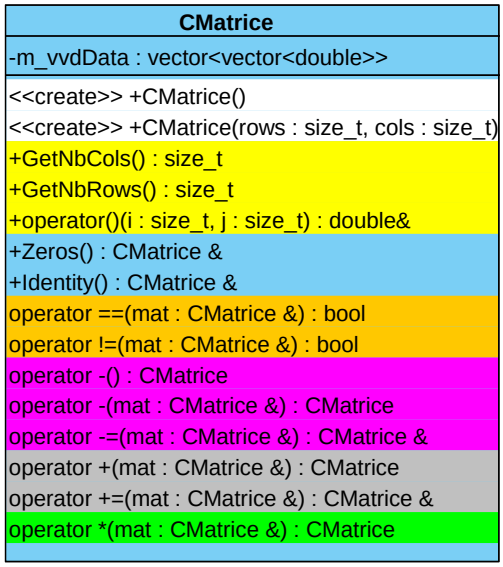

Figure 1 : Diagramme UML de la classe `CMatrice`

- ***Données membres privées***
  - **`m_vvdData`** : tableau à deux dimensions des données de la matrice. La matrice est définie colonne par colonne.
- ***Constructeurs / destructeurs***
  - **`CMatrice()`** : constructeur par défaut. Crée une matrice vide (0 ligne, 0 colonne)
  - **`CMatrice(rows : size_t, cols : size_t)`** : constructeur avec deux paramètres. Crée une matrice de taille **`rows`** × **`cols`**. **Note :** les valeurs de la matrice sont non initialisées.
- ***Accesseurs publics :***
  - **`GetNbCols()`** : retourne le nombre de colonnes de la matrice.
  - **`GetNbRows()`** : retourne le nombre de lignes de la matrice.
  - **`operator () (i : size_t, j : size_t)`** : retourne par référence la valeur à la position **`i`** × **`j`** de la matrice courante. **Note :** les indices sont basés à 0.
- ***Méthodes publiques :***
  - **`Zeros()`** : initialise à zéro toutes les valeurs de la matrice.
  - **`Identity()`** : initialise le première diagonale de la matrice à 1 et toutes les autres valeurs à 0.
- ***Opérateurs de tests :***
  - **`operator == (mat : CMatrice &)`** : vérifie si la matrice **`mat`** est égale à la matrice courante.
  - **`operator != (mat : CMatrice &)`** : vérifie si la matrice **`mat`** est différente de la matrice courante.

- ***Opérateurs mathématiques :***
  - **`operator -()`** : opérateur d'opposition.
  - **`operator -(mat : CMatrice &)`** : opérateur de soustraction.
  - **`operator -=(mat : CMatrice &)`** : opérateur de soustraction – affectation.
  - **`operator +(mat : CMatrice &)`** : opérateur d'addition.
  - **`operator +=(mat : CMatrice &)`** : opérateur d'addition – affectation.
  - **`operator *(mat : CMatrice &)`** : opérateur de multiplication.
- ***Opérateurs externes à la classe `CMatrice` :***
  - **`operator <<(out : ostream &, mat : CMatrice &) : ostream &`** : opérateur de sortie vers flux.

Afin de nous assurer du respect de ces spécifications, nous allons développer cette classe en suivant la méthodologie ***TDD*** (*Test Driven Development – Développement Dirigé par les Tests*)

1 - Cours Présentation de la méthodologie TDD

La technique de développement piloté par les tests préconise d'écrire les tests du programme avant d'écrire le code source du programme. Avant de pouvoir mettre en œuvre cette technique, nous devons savoir ce qu'est un test.

1.1 Tests unitaires

1.1.1 Définitions

Un ***test*** est une vérification du comportement d'un programme à l'exécution. Le programme doit se comporter tel que défini dans ses spécifications.

Une ***unité*** est un petit élément d'un programme complet. Il peut s'agir par exemple d'une méthode d'une classe, d'une classe complète, d'un ensemble de classes…

On appelle ***test unitaire***, le test d'une unité du programme. L'unité étant une notion variable en fonction des contextes, un ensemble de test unitaire peut devenir le test unitaire d'un composant plus gros.

Dans notre contexte, la classe **`CMatrice`** est un composant logiciel que nous voulons tester. Cela consistera à tester les comportements de toutes ses fonctions membres et méthodes. Nous avons donc deux niveaux d'unités : le niveau des méthodes et fonctions membres et le niveau de la classe **`CMatrice`** elle-même comme unité des programmes qui l'utiliseront.

1.1.2 Frameworks

Pour réaliser ces tests, on peut soit écrire un code complet de tests soi-même, soit utiliser des bibliothèques qui encadre le travail de réalisation des tests (des ***frameworks***). L'intérêt d'utiliser des frameworks est de ne pas à avoir à réinventer la roue. La plupart des frameworks de tests unitaires proposent un cadre pour définir les vecteurs de test (les entrées), définir les tests, valider les résultats des unités testées, exécuter les tests et analyser les résultats des tests. Internet regorge de ces frameworks (https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C++ ou https://en.cppreference.com/w/cpp/links/libs#Testing).

Pour rendre les choses très simples, un framework très élémentaire a été codé pour vous et se présente sous la forme d'un unique fichier **`test.h`** à inclure dans le programme de test d'un

composant. Ce framework comporte trois définitions principales : **BEGIN_TESTS**, **END_TESTS** et **TEST** et structure le programme de tests comme l'exemple de code 1.

```
#ifdef _WIN32
#include <Windows.h>
#endif
#include "test.h"

int main(void)
{
#ifdef _WIN32
  SetConsoleOutputCP(CP_UTF8);//Ligne magique pour afficher les accents
#endif
  BEGIN_TESTS
    int a = 2;
    int b = 3;
    TEST("Vérification de la valeur de a", a, 2);
    TEST("Vérification de l'inégalité entre a et b", a == b, false);
    TEST("Ce test doit échouer", a == b, true);
  END_TESTS
}
```

Code 1 : Exemple de code de test

La définition **TEST** prend trois paramètres :

- **text** Texte d'explication du test. Est affiché en début de ligne de test.
- **operation** Opération à tester.
- **resultat** Résultat attendu de l'opération.

Avec ces trois paramètres, la définition **TEST** teste que l'**operation** obtient bien le bon **resultat** pour le test identifié par le texte **text**.

L'exécution du code 1 donne le résultat de la figure 2.

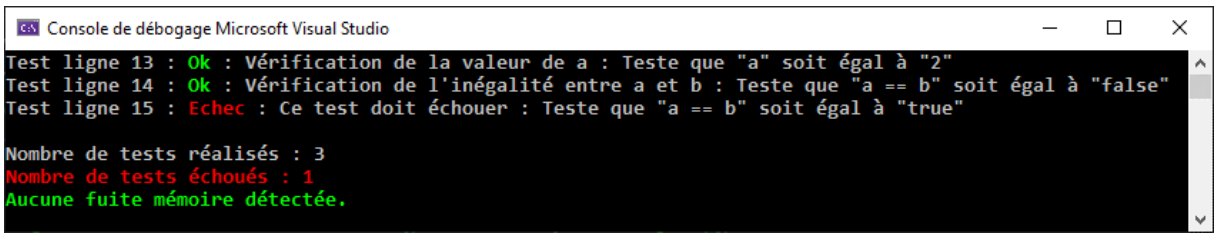

Figure 2 : Résultat de l'exécution du code 1.

1.2 Tests de non régression

En développement logiciel, on parle de ***régression*** lorsque l'ajout d'une nouvelle fonctionnalité ou la correction d'un bug à un logiciel provoque un défaut dans les fonctionnalités déjà présentes dans le logiciel.

Pour s'assurer de la non-régression d'un logiciel, il faut relancer les anciens tests lors de la modification du programme. Ainsi la création des tests unitaires dans un programme de tests conservé

permet de relancer les tests lors de la modification des composants logiciels. Le programme de tests est évidemment augmenté pour tester les nouvelles fonctionnalités.

1.3 Développement piloté par les tests

1.3.1 Procédure

Le principe de cette méthodologie de développement est de commencer à écrire les tests de notre unité. Comme l'unité n'existe pas encore, le test échoue (voire le programme ne compile pas). Il faut alors écrire le code minimal de façon à ce que le test passe puis écrire un nouveau test qui échouera. Ce principe cyclique peut être représenté par la figure 3.

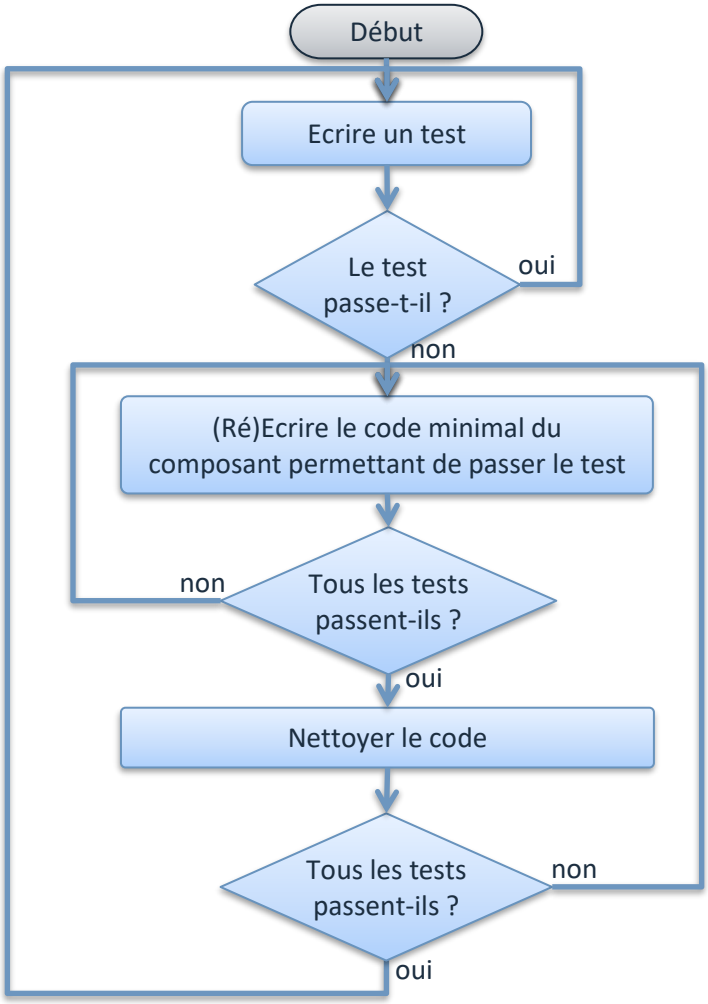

Figure 3 : Cycle de développement par la méthodologie TDD

1.3.2 Avantages

Suivre la technique TDD lors des développements apporte de nombreux avantages :

1.3.2.1 Trouver les problèmes tôt

Les problèmes sont levés au fur et à mesure du développement de par l'essence même de la méthode. En effet, la méthodologie revient à créer un problème (un test qui échoue) puis écrire le code du logiciel pour lever ce problème (faire que le test passe)

1.3.2.2 Faciliter les changements d'implémentation

En cas de refactorisation du code (réécriture pour changer l'implémentation mais pas les fonctionnalités), les tests unitaires permettent de revalider automatiquement le nouveau code. Cela assure du respect des spécifications.

1.3.2.3 Faciliter l'intégration des composants logiciels entre eux

Lorsque plusieurs composant s'intègrent dans un programme, les tests d'intégration (vérification que les composants fonctionnent en harmonie) sont facilités car il suffit de considérer l'unité à un cran supérieur dans les niveaux du logiciel.

2 - Travaux Dirigés Utilisation de la classe `CMatrice`

Pour cet exercice, vous devez vous placer dans le cas d'un développement en TDD. Cependant, en séances de TD, vous n'écrirez que les tests en supposant à chaque fois qu'ils passent. En séances de TP, vous avancerez test par test pour développer le corps de la classe **`CMatrice`**.

- En utilisant le framework présenté en section 1.1.2, écrivez un programme de tests de la classe **`CMatrice`**.
  - Vous devez réfléchir à l'ordre des tests !

3 - Travaux Pratiques Développement de la classe `CMatrice`

- En déroulant un à un les tests déterminés dans la section 2 - Travaux Dirigés et en appliquant la méthodologie TDD abordée dans la section 1 - Cours, écrivez le code de la classe **`CMatrice`**.
  - Dans les méthodes où des conditions de taille peuvent être nécessaires (par exemple pour l'opérateur de multiplication matricielle), vous pouvez utiliser la procédure **`assert`** de la bibliothèque standard **`<cassert>`** pour vous assurer du respect des contraintes.