Cd LRéférence/type[1]	LOO_CPP_AP1		☐ TD ☒ AP
Thématique principale	Premiers pas en C++		
Thématique(s) secondaire(s)	Codage et tests unitaires d'une classe via la méthode TDD (Test Driven Development)		
Durée	2h	***Niveau***[2]	A
Prérequis	✓ Travaux TD1 & TD2 ✓ Notions de TDD		
Compétences opérationnelles et niveau cible[3]	✓ Mettre en place un projet C++ de type « tests unitaires d'une classe » (avec ou sans Framework de test) (***M***) ✓ Développer une classe en utilisant le TDD (***A***) ✓ Déclarer, définir, instancier une classe (***M***) ✓ Implémenter une énumération comme classe imbriquée (***A***) ✓ Identifier et définir les constructeurs nécessaires (***A***) ✓ Mettre en place une méthode « toString() » et la surcharge de l'opérateur << associée (***N***)		
Modalités et critères d'évaluation	Suite au dépôt du code source de la classe Bit (.hpp ET .cpp) et du « main » de test (avec assertions valides activées) sur Teams ✓ Résultats du test (16 assertions) : 16 points ✓ Qualité du code produit (cohérence, efficacité, clarté, etc.) : 4 points ***Attention : la note de 0/16 sera attribuée d'office à tout code ne compilant pas.***		
Matériel(s), équipement(s), composant(s) nécessaires			
Logiciel(s) nécessaire(s)[4]	« Framework » de test doctest.		
Ressources	Projet doctest : https://github.com/doctest/doctest		

Avant la séance…

Se renseigner sur la méthode de développement « TDD ».

Revoir, si nécessaire, les éléments du C++ liés à la déclaration et à définition d'une classe, à l'instanciation et l'utilisation d'un objet ?

Travail encadré en séance

Il s'agit, pour commencer à se familiariser avec le langage C++, de travailler sur une classe ***Bit***. Cette classe a pour but de proposer les services permettant de créer des objets qui seraient à considérer comme des « avatars numériques » de bits (Binary Digits).

Cette classe sera développée de manière itérative en utilisant une technique de développement type TDD (*Test-Driven Development*) basée sur le framework de test « ***doctest*** ».

[1] TD : ½ groupe, apports de cours, mise en pratique guidée – TP : ½ groupe, travail autonome majoritaire.

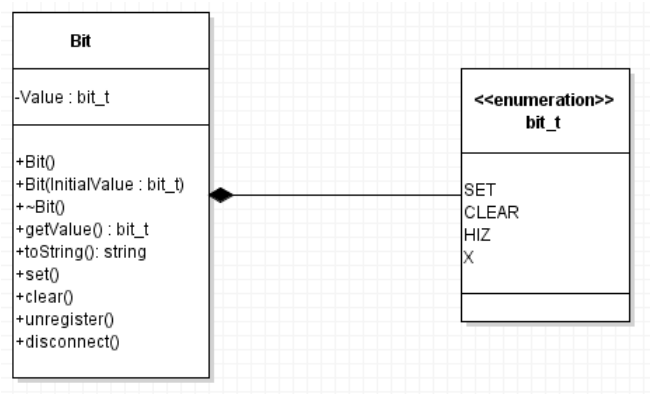
[2] Le niveau se rapporte à la thématique principale. Il peut ici être entendu comme un indicateur de la difficulté du travail (N-Facile, A-Sans grande difficulté, M-Quelques points complexes, MA-Difficile)

[3] Les niveaux cible correspondent pour chaque compétence opérationnelle au niveau d'acquisition de la celle-ci à l'issue du travail dans son intégralité (en incluant les phases préparatoires et de synthèse).

[4] En plus d'un environnement de développement C++

Modèle et Cahier des charges de la classe Bit

Cette classe doit permettre d'instancier des « avatars numériques » de bits.
Le seul attribut d'un objet de type Bit est sa *valeur*, stockée dans le champ ***Value***.
Une énumération (bit_t) permet de spécifier quatre valeurs possibles pour un bit :



- ✓ SET : Le bit est « à 1 »
- ✓ CLEAR : Le bit est « à 0 »
- ✓ HIZ : Le bit est « en haute impédance », ou déconnecté.
- ✓ X : La valeur du bit est inconnue

L'énumération bit_t est une classe imbriquée (réellement composée) dans la classe Bit. Se référer éventuellement à l'annexe 3 pour plus d'informations sur ce point.
Côté interface (méthodes publiques), cette classe doit implémenter les méthodes suivantes :

- ✓ Un constructeur sans paramètre: construit un objet de type Bit initialisé avec la valeur X.
- ✓ Un constructeur prenant comme paramètre un bit_t permettant d'initialiser la valeur du Bit.
- ✓ Un destructeur trivial (vide)
- ✓ Un accesseur en lecture (getter) retournant la valeur du bit
- ✓ Une méthode set() positionnant la valeur du bit à SET
- ✓ Une méthode clear() positionnant la valeur du bit à CLEAR
- ✓ Une méthode disconnect() positionnant la valeur du bit à HIZ
- ✓ Une méthode unregister() positionnant la valeur du bit à X
- ✓ Une méthode toString() retournant une chaîne de caractères C++ (std::string, ou std::string_view). Cette chaîne vaut ''SET'', ''CLEAR'', ''HIZ'' ou ''X'' en fonction de la valeur du bit. L'opérateur << sera surchargé de manière à être un relais vers cette méthode toString().

Au niveau du comportement, cette classe doit supporter l'instanciation de bits « en lecture seule » via le type ***const Bit***. Seules les méthodes getValue() et toString() seront accessibles sur ces objets constants.

Activités

La méthode de développement étant basée sur le TDD et le framework de test doctest, l'organisation suivante est proposée :

1. Mettre en place le répertoire de travail (LOO_AP1 par exemple).
2. Récupérer (sur Teams) le fichier de test « test_Bit.cpp ». Le copier dans le répertoire de travail.
3. Récupérer le « framework » doctest (doctest.h), l'intégrer au projet comme indiqué dans l'annexe 2 et/ou le tutoriel de l'environnement.
4. Démarrer le travail, en se laissant « driver » par les tests et en appliquant la méthode (fail → pass → refactor).

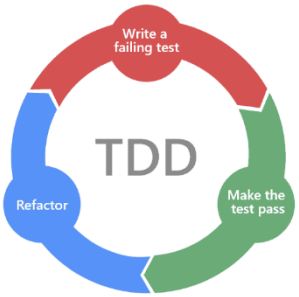
Après la séance…

Procéder à son auto-évaluation par rapport aux compétences opérationnelles listées dans le tableau de présentation de l'activité en se situant par rapport aux niveaux-cibles attendus.

Commencer (poursuivre) la rédaction d'une fiche « aide-mémoire » où vous intègrerez, selon le format qui vous semble approprié des éléments, principalement syntaxiques, de manière à gagner du temps dans le futur. Cette fiche a vocation à être complétée au fur et à mesure de l'avancée dans les TDs et APs.

Annexes

Annexe 1 – Le TDD en 2 mots



La méthode TDD est une méthode permettant de développer une fonction ou une classe, de manière itérative, en commençant par coder les tests. Les tests doivent être mis en place de manière à mettre en évidence le fait que le comportement de l'unité testée répond exactement au comportement attendu. Ce comportement est issu de l'analyse du cahier des charges de la fonction / classe dont il est question.

La méthode est basée sur la reproduction jusqu'à validation complète des trois séquences suivantes :

1. Ecrire un test. Il est évident que ce test va échouer vu que le code à tester n'existe pas encore.
2. Ecrire le code permettant au test de réussir. Il est important ici de n'ajouter ***que*** les éléments permettant au test de réussir (la qualité du code n'est pas un critère à ce moment). L'objectif est d'avoir un code simple (souvent abominable d'un point de vue analyse).
3. Soumettre le code à une analyse puis un refactoring afin de le rendre « correct » (propre, respectant les guidelines, etc.). Ce nouveau code doit, bien entendu, continuer à « passer le test »

Remarque : Il ne s'agit bien là que des principes fondamentaux et minimaux de la méthode. Toute volonté de mettre en œuvre le TDD de manière un peu plus sérieuse devra s'accompagner de recherches complémentaires (il y a une multitude de ressources, souvent très didactiques et claires sur Internet. Elles peuvent concerner le TDD en général, le TDD spécifique au C++, avec ou sans framework de test). Un exemple de ces ressources ici : https://www.youtube.com/watch?v=N2gTxeIHMPo

Annexe 2 – Le minimum pour démarrer avec doctest

1. Récupérer le fichier "doctest.h" (github de doctest), le copier dans le répertoire de travail (ou dans son dossier /include/ ou dans tout autre endroit d'où il pourra être inclus dans le projet).
2. Dans le fichier de test (remplaçant le fichier contenant le main), inclure (en plus des autres inclusions) le fichier « doctest.h ».
3. Indiquer que le mode de fonctionnement sera sans main (à définir ***avant*** l'include) :
   #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
4. Définir des séquences de test « macro » avec TEST_CASE, éventuellement composés de sous-tests (SUBCASE).
5. Mettre en place les assertions avec la macro CHECK
6. S'assurer d'éventuelles conditions initiales avec REQUIRE

Annexe 3 – Une énumération comme classe imbriquée

Si les énumérations sont parfaitement gérées par le C++ à la façon du C, il est fortement recommandé, depuis le C++11 d'utiliser une classe pour encapsuler une énumération. On parle alors de classe d'énumération, ou *strongly typed enums*. De plus, vu qu'il est aussi possible en C++ d'imbriquer (réellement) des classes on peut encapsuler l'énumération au sein d'une classe, en lui donnant la visibilité adéquate (private, protected ou public).

La déclaration d'une classe d'énumération est assez simple en soi. Par exemple :

- ✓ enum class Color {RED, GREEN, BLUE};

Une variable de ce type est déclarée et utilisée de la manière suivante :

```
Color color = Color::GREEN;
if ( Color::RED == color )
{
    // the color is red
}
```

Il est aussi possible de contrôler le type support de ces énumérations (int par défaut), cela permet d'optimiser l'empreinte mémoire des variables de ce type, par exemple :

- ✓ enum class Color : std ::int8 {RED, GREEN, BLUE};

Dans ce cas, un entier 8 bits sera utilisé comme type support, en lieu et place d'un entier « standard » qui serait lui, codé sur 16, 32 voire 64 bits.

Si cette classe d'énumération est déclarée et/ou définie dans une zone publique de la classe devant l'utiliser, elle pourra être utilisée depuis l'extérieur, pour passer des paramètres à des méthodes par exemple.

Annexe 4 – Surcharge de l'opérateur de sortie << et relais vers toString()

Beaucoup de classes implémentent une méthode toString(), ou surchargent l'opérateur de sortie <<. Si les deux éléments sont présents, alors, la surcharge de l'opérateur de sortie sert uniquement de relais (ou réimplémente à l'identique) vers la méthode toString().

Dès lors que ce cas se présente, l'implémentation pour une classe « Classe » en est la suivante :

```
std::ostream& operator<<(std::ostream& os, const Classe& obj) {
      os << obj.toString();
      return os;
      }
```