Référence/type[1]	LOO C++ - TD2		☒ TD ☐ AP
Thématique principale	Premiers pas en C++ : Déclaration, définition et tests unitaires d'une classe		
Thématique(s) secondaire(s)	✓ API Design ✓ C++ Best Practices		
Durée	2h	***Niveau***[2]	A
Prérequis	✓ LOO C++ CM et TDs précédents		
Compétences opérationnelles et niveau cible[3]	✓ Déclarer, définir une classe en C++ (***M***) ✓ Tester unitairement une classe (***M***) ✓ Mettre en place une « bonne » API (***A***) ✓ Appliquer les bonnes pratiques du C++ (***N***)		
Modalités et critères d'évaluation	Auto évaluation		
Matériel(s), équipement(s), composant(s) nécessaires			
Logiciel(s) nécessaire(s)[4]			
Ressources			

Avant la séance…

Mettre en place le projet C++ minimal permettant de commencer le travail :

- ✓ Un main de test : test_de.cpp
- ✓ Les fichiers de déclaration et de définition de la classe : De.hpp et De.cpp

Travail encadré en séance

Apports de cours

- ✓ void 😊

Activités

Il s'agit lors de ce TD de se familiariser avec la déclaration et la définition de classes, l'instanciation et l'utilisation d'un objet de cette classe.

Un point d'attention spécifique sera porté à la phase de construction et d'initialisation de l'objet ainsi que la mise en évidence de problèmes classiques liés à ces phases.

Cahier des charges de la classe à produire

L'objectif est de fournir une version POC d'une classe « Dé ». On parle ici d'une version POC car on tolèrera que certains éléments, même critiques, ne soient pas développés afin de se focaliser sur les principaux éléments du travail. La gestion des erreurs (via exceptions) notamment pourra être mise de côté.

[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++

La matrice MOSCOW ci-dessous définit clairement le périmètre du travail à effectuer lors de cette séance.

MUST have	***SHOULD have***
1. La création d'un dé d'un des types suivants : D4, D6, D8, D10, D12, D20, D30 et D100. 2. La création d'un dé « quelconque » (tirage entre une borne inférieure et une borne supérieure). 3. La création d'un dé à 6 faces par défaut. 4. Le Tirage d'une valeur. 5. Le tirage d'un nombre n de valeurs (n tirages du même dé). 6. Une gestion des tirages basée sur les outils disponibles dans la bibliothèque std::random depuis le C++11 (std::random_device, std::default_random_engine, std::uniform_int_distribution…)	1. Une identification des erreurs possibles de la création à la destruction de l'objet.
COULD have	***WON'T have***
1. Une réflexion sur la gestion moderne des erreurs, au moins au niveau stratégique.	1. La possibilité de changer le type de dé (nombre de faces) une fois l'objet construit. 2. La gestion complète et moderne des erreurs.

Organisation (proposée) du travail

1. Proposer une première version, forcément incomplète, potentiellement erronée, assurément perfectible, de l'API de cette classe (méthodes publiques, d'interface). Identifier les propriétés ainsi que le niveau d'accessibilité (cachée, ro, rw). Produire en même temps la représentation façon diagramme des classes de cette classe.
2. Réfléchir aux différents constructeurs à mettre en place. La règle du zéro semble-t-elle atteignable ?
3. Commencer à analyser une méthode ainsi que les tests unitaires à mettre en place. Démarrer l'implémentation et les tests.
4. Discuter sur les aspects « bonnes pratiques », sécurité, fiabilité, optimisation…
5. Recommencer les phases 3 et 4 jusqu'à obtenir une classe fonctionnelle.

Synthèse

Commencer à rédiger un document listant/recensant toutes les bonnes pratiques et autres informations utiles vues au cours des séances afin de s'appuyer dessus par la suite.

Après la séance…

Après avoir eu les éléments de cours sur les exceptions, ajouter à la classe « De » une gestion d'erreur. La principale (seule ?) erreur identifiée est la tentative de créer un dé à 1 face (limite haute = limite basse).

Annexe 1 : La génération de nombres aléatoires en C++

Ref : https://cplusplus.com/reference/random/

En C++ moderne, une bonne qualité de valeurs aléatoires est obtenue en utilisant trois types d'objets :

- ✓ Un ***std::random_device*** : qui est en soi un générateur de nombres aléatoires, basé sur les capacités matérielles/logicielles disponibles (qui dépendent de la cible). Cet objet sera utilisé comme la graine (seed) d'un générateur (random_engine).
- ✓ Un Générateur (random_engine) : objet qui utilise un algorithme afin de générer des nombres de manière pseudo-aléatoire. Le générateur à besoin d'une graine (seed) pour initialiser sa séquence. Plusieurs générateurs sont disponibles, du plus générique ***std::default_random_engine***, aux implémentations les plus spécifiques (Mersenne twisters, RanLux…)
- ✓ Une distribution (distribution) : objet qui transforme une séquence aléatoire issue d'un générateur en séquences suivant une loi spécifique (uniforme, normale, binomiale…)

L'exemple suivant montre la manière la plus simple (triviale) d'utiliser ces outils :

```
std::default_random_engine generator;
std::uniform_int_distribution<int> distribution(1,6);
int dice_roll = distribution(generator);  // generates number in the range 1..6
```

Annexe 2 : La règle des 3, 5, 7, 0, 4, 6…

Ces concepts font partie des éléments clés du C++ et participent, de par leur complexité (au moins apparente), à l'impopularité du langage.

L'objectif de cette annexe n'est pas d'expliquer dans le détail ce qui se cache là-dessous, mais de donner quelques informations et une bonne pratique vers laquelle tendre à chaque fois que ce sera possible.

Au départ il y a la règle des 3…

La règle des trois nous dit que dans le cadre de l'implémentation d'une classe, si l'on est amené à définir ***par nous-même*** le constructeur par recopie, l'opérateur d'affectation (associé au constructeur par recopie) ou le destructeur alors il faut définir nous même l'ensemble de ces trois fonctions (car le compilateur sera dans l'incapacité de le faire de manière automatique).

Cette règle est obsolète depuis l'apparition du move semantic en C++11.

… puis la règle des 5…

Le C++11 ajoute au standard la notion de move, beaucoup plus efficace que la copie. De nouveaux opérateurs apparaissent et il est possible de construire une clase à partir d'un objet « déplacé ».

En pratique, cela ajoute deux éléments à la règle des trois : le constructeur « move » et l'opérateur d'affectation associé. En résumé, si l'on est amené à définir par nous-même une de ces cinq fonctions, il faudra définir par nous-même l'ensemble des cinq fonctions, ou a minima demander explicitement au compilateur de le faire en ajoutant le mot-clé « = default ». Cependant, rien ne garantit que le compilateur saura implémenter cette fonction. Une erreur de compilation se produira dans ce cas.

… et enfin la règle du zéro.

La règle du zéro, qui n'est pas une règle, mais une bonne pratique vers laquelle tendre nous dit « If you can avoid defining default operations, do. »

(https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c20-if-you-can-avoid-defining-default-operations-do)

L'idée est de faire en sorte de ne pas avoir à définir l'une des cinq fonctions (appelées special functions) par nous-même, et de laisser le compilateur en implémenter des versions par défaut.

Dans la réalité, cette « règle » s'applique souvent sous le nom de « ***règle des 5 défauts*** » : les cinq « special functions » sont explicitement marquées comme devant être générées automatiquement par le compilateur (= default).

```
class base_of_five_defaults
{
public:
    base_of_five_defaults(const base_of_five_defaults&) = default;
    base_of_five_defaults(base_of_five_defaults&&) = default;
    base_of_five_defaults& operator=(const base_of_five_defaults&) = default;
    base_of_five_defaults& operator=(base_of_five_defaults&&) = default;
    virtual ~base_of_five_defaults() = default;
};
```

Qu'en est-il du constructeur par défaut ?

Le constructeur par défaut (ie sans paramètre) est parfois intégré à ces règles, qui deviennent alors la règle des 4, ou des 6 ou des 6 défauts. En réalité, que l'on définisse nous-même le constructeur par défaut n'a pas d'impact automatique sur les autres constructeurs et opérateurs d'affectation. Il faudra étudier chaque cas de manière spécifique. Il y a cependant une règle simple que l'on peut appliquer : on ne doit jamais définir par nous-même un constructeur par défaut vide. Si le constructeur par défaut ne fait rien (situation à rechercher) il faut alors le marquer « = default ».