Référence/type[1]	LOO CPP AP2b		☐ TD ☒ AP
Thématique principale	Spécialisation de classe		
Thématique(s) secondaire(s)	Classes abstraites Templates et classe template Concepts		
Durée	2h	***Niveau***[2]	M
Prérequis	✓ Bases solides sur la spécialisation (héritage) et les classes abstraites ✓ Notion sur les templates		
Compétences opérationnelles et niveau cible[3]	✓ Analyser et définir une classe abstraite (***A***) ✓ Mettre en place une classe template (***N***) ✓ Spécialiser la classe vers une classe concrète (***M***) ✓ Exploiter la notion de concept (C++20) (***N***)		
Modalités et critères d'évaluation	Evaluation en cours de séance (/20)		
Matériel(s), équipement(s), composant(s) nécessaires			
Logiciel(s) nécessaire(s)[4]			
Ressources	https://www.codeproject.com/Articles/5340890/An-Introduction-to-Cplusplus-Concepts-for-Template		

Avant la séance…

Se renseigner, si nécessaire, sur la notion de template et le principe général de la programmation générique. Revoir les points clés de la spécialisation (héritage) et la notion de classe abstraite.

Travail encadré en séance

Apports de cours

- ✓ Précisions sur les templates, les avantages, inconvénients et limites.
- ✓ Les concepts (à partir de C++20) pour faciliter le travail avec les templates.

Activités

L'objectif de cette activité pratique est, à travers un exemple autour d'objets « génériques » de type capteur de mettre en œuvre la spécialisation de classes, la création de classes abstraites et concrètes. Il s'agira aussi, à titre d'ouverture d'utiliser, dans une forme simple, voire simpliste, les templates et concepts.

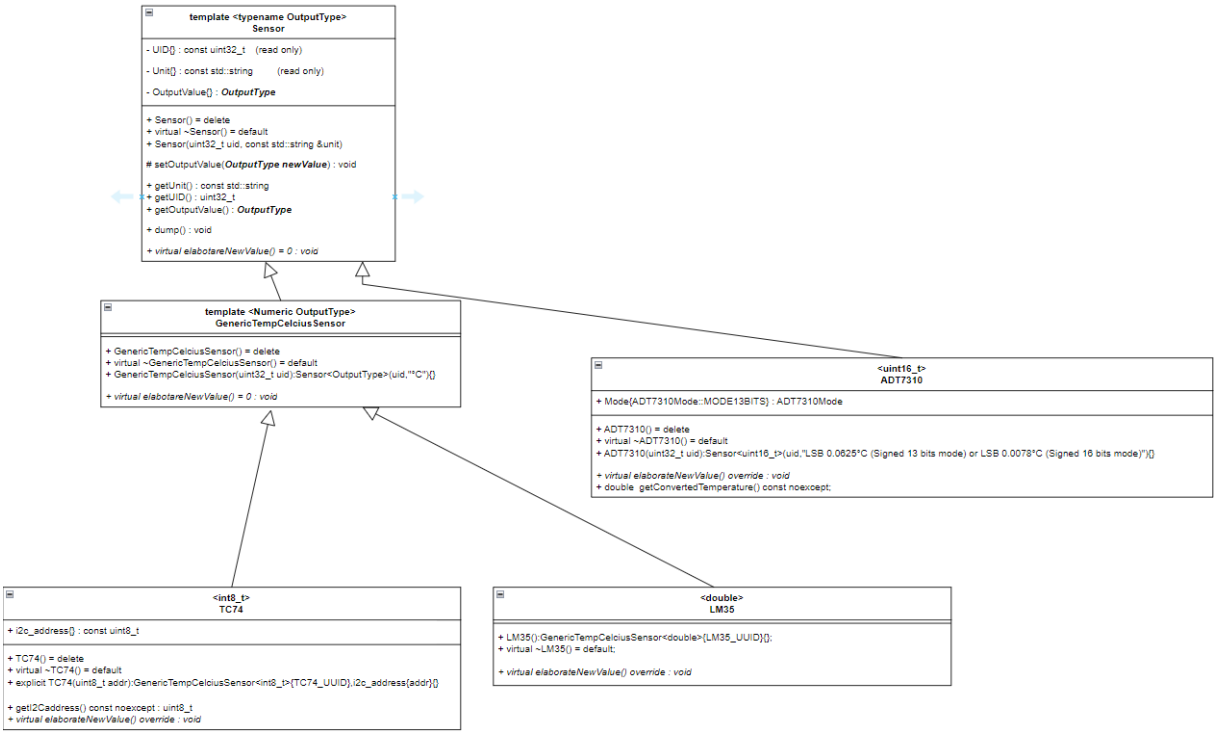
L'ensemble des capteurs codés ici seront des simulations basées sur une génération aléatoire de la valeur de sortie.

[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++

Présentation du travail – Partie 1

Le diagramme des classes cible est donné ci-dessous.

Il n'est pas obligatoire lors de la séance d'implémenter l'ensemble des classes mais seul ce travail permet de se rendre complètement compte des interactions entre objets. Il faut donc se donner cet objectif.



Le point de départ est une classe « Sensor ». Ses deux principales caractéristiques sont :

- ✓ Il s'agit d'une classe ***abstraite*** (impossible à instancier directement). Une seule méthode est responsable de ce fait : la méthode virtuelle pure *elaborateNewValue()*. Cette méthode est appelée pour élaborer une nouvelle valeur de sortie du capteur. Cette méthode est différente selon le capteur réel instancié, néanmoins tous les capteurs concrets doivent implémenter cette méthode.
- ✓ Il s'agit d'une classe ***template*** car on a choisi ici de pouvoir adapter la classe en fonction du type réel de la donnée en sortie (entier, signé ou non, 8, 16, 32 bits, flottant…). Un objet de type sensor devra donc indiquer pour quel type il sera construit. La notion de ***concept*** sera ici abordée afin de limiter les types utilisables à des types numériques (on considère qu'un capteur renvoie forcément une valeur numérique).

Le diagramme des classes de cette classe est « explicite », il montre exactement les méthodes à définir, y compris au niveau des accesseurs.

Parmi les éléments de cahier des charges de cette classe, on peut citer :

- ✓ ***UID*** est un Unique IDentifier propre à chaque capteur concret. Cet UID codé sur 32bits est donné de manière arbitraire et n'est pas modifiable une fois l'objet créé. Un getter permet de récupérer cet UID.
- ✓ ***Unit***, permet d'indiquer sous forme de chaîne de caractères l'unité de la valeur de sortie. Il s'agit là d'une donnée purement informative pour l'utilisateur. Ce champ n'est pas modifiable une fois l'objet créé. Un getter permet de récupérer sa valeur.

- ✓ ***OutputValue*** est le champ stockant la dernière valeur captée. Cette valeur est mise à jour suite à un appel à la méthode « elaborateNewValue() ». Le setter associé est donc protégé.
- ✓ Le constructeur par défaut est explicitement supprimé.
- ✓ Le destructeur à implémenter est celui par défaut.
- ✓ Le seul constructeur utilisable prend deux paramètres : l'UID et l'unité.
- ✓ La méthode « dump() » réalise un affichage console de la valeur de sortie et de l'unité.
- ✓ La méthode « elaborateNewValue() » est une méthode virtuelle pure.

Travail à réaliser – Partie 1

- ➢ Coder l'ensemble des éléments de la classe « Sensor » dans un fichier Sensor.hpp. Vérifier que la compilation se passe correctement (ne pas essayer d'instancier cette classe dans le main de test !).

Présentation du travail – Partie 2

Il s'agit de réaliser un capteur concret dérivant directement de la classe « Sensor ». Il s'agit d'un capteur bâti sur une classe « ADT7310 ». Ce capteur est un capteur de température numérique retournant la température sous forme d'un entier signé 16 bits. Selon le mode de fonctionnement du capteur, la température est construite soit à partir de 13bits utiles soit à partir des 16bits.

Une énumération est à créer pour gérer le mode :

- ✓ enum class ADT7310Mode {MODE13BITS, MODE16BITS};

Le mode en lui-même est une propriété privée de la classe, accessible en lecture/écriture via des accesseurs publics.

L'unité prend ici la forme d'une chaîne de caractères explicative :

- ✓ "LSB 0.0625°C (Signed 13 bits mode) or LSB 0.0078°C (Signed 16 bits mode)"

Tout comme la classe « Sensor » le constructeur par défaut est supprimé et c'est le destructeur par défaut qui est à implémenter.

Le constructeur prend un seul paramètre : l'UID du capteur concret.

La méthode « ***elaborateNewValue()*** » est ici concrète, elle génère une valeur aléatoire entière à placer dans le champ OutputValue de la classe mère. Cette valeur est codée sous forme d'un entier non signé 16 bits, dont les valeurs sont comprises :

- ✓ Entre 0 et 0x1FFF en mode 13 bits
- ✓ Entre 0 et 0xFFFF en mode 16 bits

La méthode « ***getConvertedTemperature()*** » permet de récupérer la température convertie en °C sous la forme d'un flottant. Les formules à appliquer en fonction du mode et du bit de signe (bit de poids fort) de la valeur brute sont données ci-contre.

TEMPERATURE CONVERSION FORMULAS

16-Bit Temperature Data Format

Positive Temperature = *ADC Code*(dec)/128

Negative Temperature = (*ADC Code*(dec) – 65,536)/128

where *ADC Code* uses all 16 bits of the data byte, including the sign bit.

Negative Temperature = (*ADC Code*(dec) – 32,768)/128

where the MSB is removed from the ADC code.

13-Bit Temperature Data Format

Positive Temperature = *ADC Code*(dec)/16

Negative Temperature = (*ADC Code*(dec) – 8192)/16

where *ADC Code* uses all 13 bits of the data byte, including the sign bit.

Negative Temperature = (*ADC Code*(dec) – 4096)/16

where the MSB is removed from the ADC code.

Travail à réaliser – Partie 2

- ➢ Coder le maximum d'éléments de cette classe dans le .hpp (on pourra utiliser, exceptionnellement, le même fichier .hpp pour toutes les classes de l'exercice).

- Coder la méthode elaborateNewValue() dans le fichier .cpp (Sensor.cpp ou autre).
- Maintenant que cette méthode existe, la classe est instanciable. Tester l'ensemble des éléments déjà codés à partir d'un main de test (y compris les éléments de la classe mère).
- Finir par le codage et le test de la méthode getConvertedTemperature()

Présentation du travail – Partie 3

Dans cette troisième partie, il s'agit de travailler sur d'autres spécialisations de la classe Sensor. Tout d'abord en créant une nouvelle classe ***abstraite*** « GenericTempCelsiusSensor », héritant de Sensor et servant de base à d'autres classes, concrètes, représentant des capteurs concrets. Ces capteurs concrets on la caractéristique commune d'être des capteurs de température retournant une température en °C. Pour l'exemple, on se donnera l'objectif d'implémenter deux classes « capteurs concrets » :

- ✓ Classe TC74
- ✓ Classe LM35

Les TC74 sont des capteurs de température I2C, retournant la température sous la forme d'un entier 8 bits signé, avec un LSB de 1°C.

Les LM35 sont des capteurs analogiques. Un objet LM35 fournira la température sous forme de double.

Travail à réaliser – Partie 3

- Coder la classe GenericTempCelsiusSensor. Identifier les éléments qui devront faire l'objet de tests spécifiques une fois cette classe rendue « instanciable ».
- Analyser et coder l'une des deux classes concrètes LM35 ou TC74. Valider le comportement ainsi que les points identifiés précédemment.
- Finir le travail en codant et testant la classe restante.

Synthèse

Réfléchir sur une nouvelle analyse d'une problématique similaire (mise en place de capteurs « génériques ») sous la forme d'interface.

Après la séance…

Finir, le cas échéant, le travail demandé lors de ce TP.

Annexe 1 : Quelques mots sur les templates et classes templates

Voici quelques liens vers de ressources expliquant plus ou moins simplement/complètement les templates en C++. Une vision « macro », globale de ce concept, suffit pour réaliser le travail demandé lors de cette séance. Inutile de trop chercher à rentrer dans le détail et la compréhension fine.

- ✓ https://koor.fr/Cpp/Templates/TemplatedFunction.wp
- ✓ https://cpp.developpez.com/cours/cpp/?page=page_14
- ✓ https://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/templates.html

Annexe 2 : Aller plus loin dans les templates avec les concepts (since C++20)

Ici encore, quelques liens, pour aborder sans entrer dans les détails les concepts C++.

- ✓ https://www.cppstories.com/2021/concepts-intro/
- ✓ https://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Concept-de-concept.html
- ✓ https://en.cppreference.com/w/cpp/language/constraints
- ✓ https://en.cppreference.com/w/cpp/concepts