Référence/type[1]	LOO C++ EC 2024/2025		☐ TD ☒ AP
Thématique principale	Création, test et mise en œuvre de classes C++		
Thématique(s) secondaire(s)	Spécialisation – Réalisation d'interface Application multithread		
Durée	2h	***Niveau***[2]	M
Prérequis	✓ Bases du C++		
Compétences opérationnelles et niveau cible[3]	✓ Coder et valider une classe dont le cahier des charges est fourni ✓ Réaliser une interface en spécialisant une classe mère ✓ Utiliser les outils de base du multithreading C++23 (jthread…)		
Modalités et critères d'évaluation	En cours de séance, conformité au cahier des charges, qualité du code… (/30). Note ramenée sur 20pts avec arrondi au 1/2pt supérieur.		
Matériel(s), équipement(s), composant(s) nécessaires			
Logiciel(s) nécessaire(s)[4]			
Ressources			

Avant la séance…

Vérifier son environnement de développement.

Travail en séance

Partie 1 – La classe CanMessage et sa validation (16 pts)

Il s'agit, dans un premier temps, de coder et de valider la classe CanMessage, dont l'analyse et la description sont fournies en annexe2. Les tests unitaires seront réalisés à partir d'un main de test « classique » ou à l'aide du framework doctest. La procédure à suivre est indiquée ci-dessous. Il est recommandé (pas imposé) de faire valider chaque étape par l'encadrant de TP.

Etape 1 : Mise en place du projet, validation de l'environnement de développement (1pt)

- Créer les trois fichiers du projet C++ :
  - CanMessage.hpp / CanMessage.cpp
  - test_CanMessage.cpp
- Mettre en place les éléments de base (fonction main ou un premier TEST_CASE, code guards dans le .hpp…)
- Construire l'exécutable – Vérifier la construction et l'exécution sans erreur ni warning.

Etape 2 : Infrastructure minimale de la classe CanMessage (2pts)

- Déclarer la classe CanMessage, réaliser la composition des classes énumérées canType et canFormat.
- Définir les propriétés conformément au cahier des charges (cf annexe 2).

[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++

- Définir le constructeur par défaut conformément au cahier des charges.
- Définir le destructeur conformément au cahier des charges.
- Dans le main (ou dans un TEST_CASE) instancier un objet de type CanMessage.
- Construire l'exécutable – Vérifier la construction et l'exécution.

Etape 3 : Définition et tests des accesseurs (4 pts)

- Définir et valider les accesseurs pour les propriétés Id, Type, Format et Dlc

Remarque : la qualité des tests mis en place sera prise en compte dans l'évaluation.

Etape 4 : Définition et tests des constructeurs standard (3 pts)

- Définir et tester chacun des trois constructeurs standard

Etape 5 : Accesseur spécial sur les données (1 pt)

- Mettre en place l'accesseur sur les données (méthode Data())
- Valider l'accès au champ de données en lecture et en écriture

Etape 6 : Définition et tests de la méthode isMessageValid (2 pts)

- Identifier clairement, en fonction des types des propriétés, les situations faisant qu'une trame est valide ou non.
- Définir la méthode isMessageValid.
- Tester le plus complètement possible cette méthode

Etape 7 : Définition et test de la méthode toString (3 pts)

- Définir la méthode toString conformément au cahier des charges
- Valider le plus complètement possible cette méthode.

Partie 2 – L'interface CanManager et une réalisation (6 pts)

Il s'agit maintenant de réaliser une interface. Le code complet de la classe mère « CanManager » est fourni, de même qu'une partie de son analyse (cf annexe 3).
La classe à développer est un objet concret de type CanManager, appelé « ***dummyCanSender*** ».
Le cahier des charges de cette classe est fourni en annexe 4.

- Coder et tester la classe dummyCanSender.

Partie 3 – Application (8pts)

Cette dernière partie consiste en l'écriture d'une application émettant périodiquement deux trames CAN différentes.
Une première trame est à émettre toutes les secondes :

- ✓ Trame standard, data, Id = 0x41, Dlc = 2, data = {0x42,0x66}

Une seconde trame est à émettre toutes les 666ms :

- ✓ Trame étendue, data, Id = 0x42, Dlc = 0.

L'émission en elle-même est à faire prendre en charge par une fonction threadable :

- void txThread(unsigned int nbTx, std::chrono::milliseconds tx_delay, CanManager &sender, const CanMessage &msg)

Les paramètres de cette fonction sont :

- ✓ nbTx : Le nombre de fois que la trame doit être émise.

- ✓ tx_delay : délai entre deux émissions, en ms.
- ✓ sender : Référence sur une interface CAN
- ✓ msg : Référence sur le message à émettre

Etape 1 : Mise en place de la fonction (2 pts)

- ➢ Sans (pour le moment) se soucier de l'aspect « threading », analyser et coder la fonction txThread.
- ➢ Valider à partir d'un nouveau main de test ou d'un TEST_CASE dédié.

Etape 2 : Préparation à l'exécution dans un cadre multithread (2 pts)

- ➢ Identifier les contraintes ajoutées par la prise en compte d'une exécution dans le cadre d'une application multithread.
- ➢ Identifier le(s) outil(s) à utiliser pour répondre à ces contraintes.
- ➢ Modifier la fonction txThread en conséquence. Valider le fonctionnement.

Etape 3 : Application (4 pts)

- ➢ Dans un nouveau main ou un TEST_CASE dédié, mettre en place l'application, à base de std::jthread. On considère que chaque message doit être émis entre 3 et 7 fois, avec un nmbre d'émissions différent (3 fois pour le premier message et 5 fois pour le second par exemple).
- ➢ Valider le fonctionnement

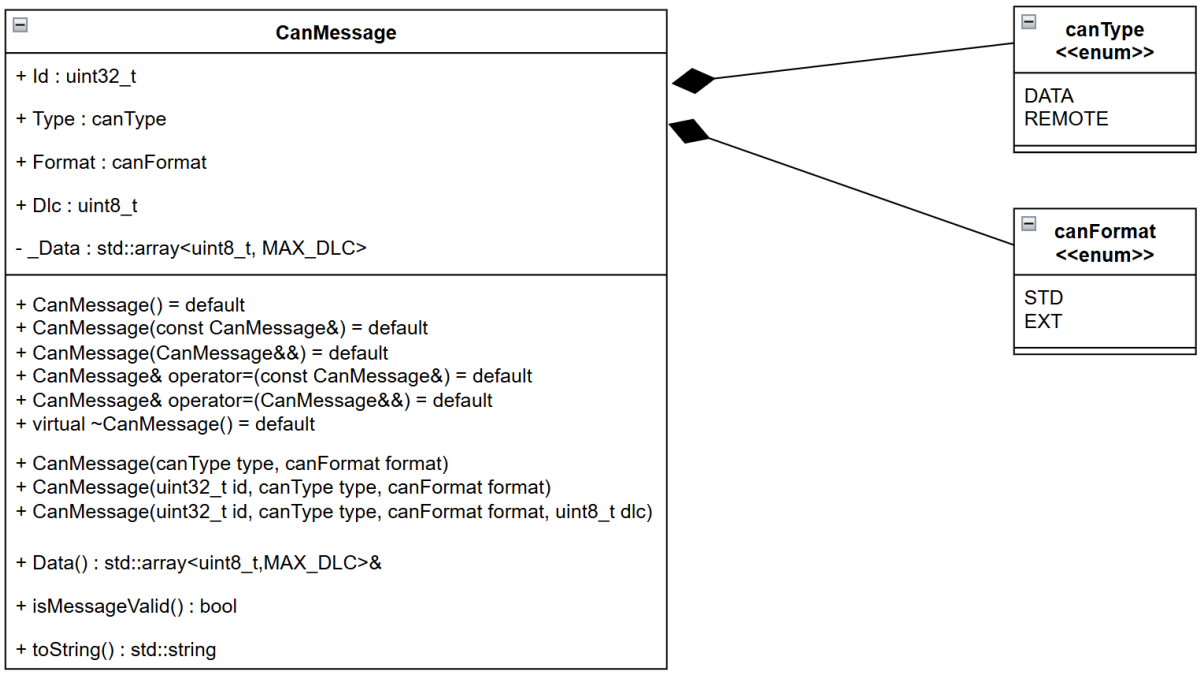
Annexe 1 – Rappels sur une trame CAN

Du point de vue logique, une trame CAN est définie par les éléments suivants :

- ✓ Un ***identifiant*** sur 11 ou 29 bits (selon le format de la trame)
- ✓ Un ***type*** :
  - DATA : La trame est une trame de donnée, pouvant contenir de 0 à 8 octets de donnée
  - REMOTE : La trame est une trame de requête, les données sont vides
- ✓ Un ***format*** :
  - Standard (STD) : L'identifiant de la trame est codé sur 11 bits
  - Etendu (EXT) : L'identifiant de la trame est codé sur 29 bits
- ✓ Une taille de données (***dlc***) : valeur comprise entre 0 et 8 indiquant le nombre d'octets de données d'une trame « DATA » ou le nombre d'octets attendus en retour dans le cas d'une trame « REMOTE »
- ✓ Un champ de données (***data***) : Les données en elles-mêmes (entre 0 et 8 octets)

Annexe 2 – Analyse et CdC de la classe CanMessage

Diagramme des classes



Cahier des charges

Constructeurs

La règle du zéro d'applique ici : les constructeurs par défaut, par recopie et par déplacement sont instanciés par le compilateur. Il en va de même pour les opérateurs de recopie et de déplacement.
Par défaut, une trame CAN est initialisée avec un identifiant égal à 0, un Dlc égal à 0 et une zone de données « vide ». Le type par défaut est « DATA » et le format par défaut est « STD ».
Cette classe est aussi dotée de trois constructeurs « standard », permettant de construire un objet en donnant comme paramètres :

- ✓ Le type et le format de la trame
- ✓ L'identifiant, le type et le format de la trame
- ✓ L'identifiant, le type, le format et le Dlc de la trame

Remarque : aucune vérification de la cohérence/validité des paramètres n'est effectuée par les constructeurs.

Destructeur

En conformité avec la règle du zéro, le destructeur est instancié par le compilateur.

Accesseurs

Comme indiqué par le diagramme des classes les champs Id, Type, Format, et Dlc bénéficient d'accesseurs en lecture et en écriture.

Aucun accesseur n'est à définir pour le champ _Data (l'accès à ce champ sera pris en charge par une méthode spécifique, qualifiée d'accesseur « spécial »).

Remarque : aucune vérification de la cohérence/validité des valeurs n'est effectuée par les accesseurs, ni en écriture, ni en lecture.

L'accesseur « spécial » sur les données

Il s'agit de la méthode Data().

Cette méthode ne prend aucun paramètre et retourne une référence sur le champ _Data de l'objet.

Méthode isMessageValid

Cette méthode est destinée à vérifier la validité de la trame en fonction des valeurs des champs Id, Type, Format et Dlc. Elle ne reçoit aucun paramètre et retourne un booléen. La valeur retournée est « vrai » si la trame est considérée comme valide, « faux » dans le cas contraire.

Méthode toString

Cette méthode a pour rôle de proposer une représentation de la trame sous la forme d'une chaîne de caractères. Cette chaîne est celle retournée par la méthode.

Cette chaine est formatée de la manière suivante :

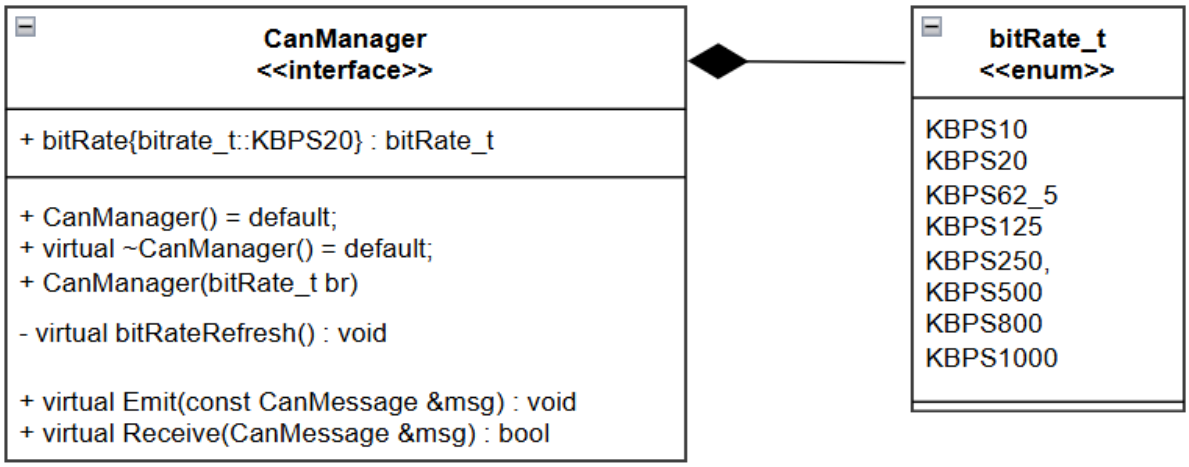
format;type;id;dlc;data;validity

- ➢ format : "STD" or "EXT"
- ➢ type : "DATA" or "RMT"
- ➢ id : 3 (std frame) or 8 (ext frame) hexadecimal digits representing the frame ID
- ➢ dlc : dlc value (decimal)
- ➢ data : each data byte is represented as a 2 digit hex value - No separators between data bytes
  - Rq : if dlc=0 or if the frame is a remote one this field is empty
- ➢ Validity : "TRUE" if the frame is valid, "FALSE" if not

Exemple : une trame valide, standard, d'id 0x0A5, de DLC 2 transportant les données 0x42 et 0x66 sera représentée par la chaîne suivante :

STD;DATA;0A5;2;4266;TRUE

Annexe 3 – Analyse et CdC de la classe CanManager



La classe CanManager est une ***interface***. Son API est constituée :

- ✓ D'un constructeur par défaut, initialisant le débit à 20kbps
- ✓ D'un constructeur standard, prenant en paramètre le débit de l'interface CAN.
- ✓ Deux méthodes permettant l'émission et la réception de trames.

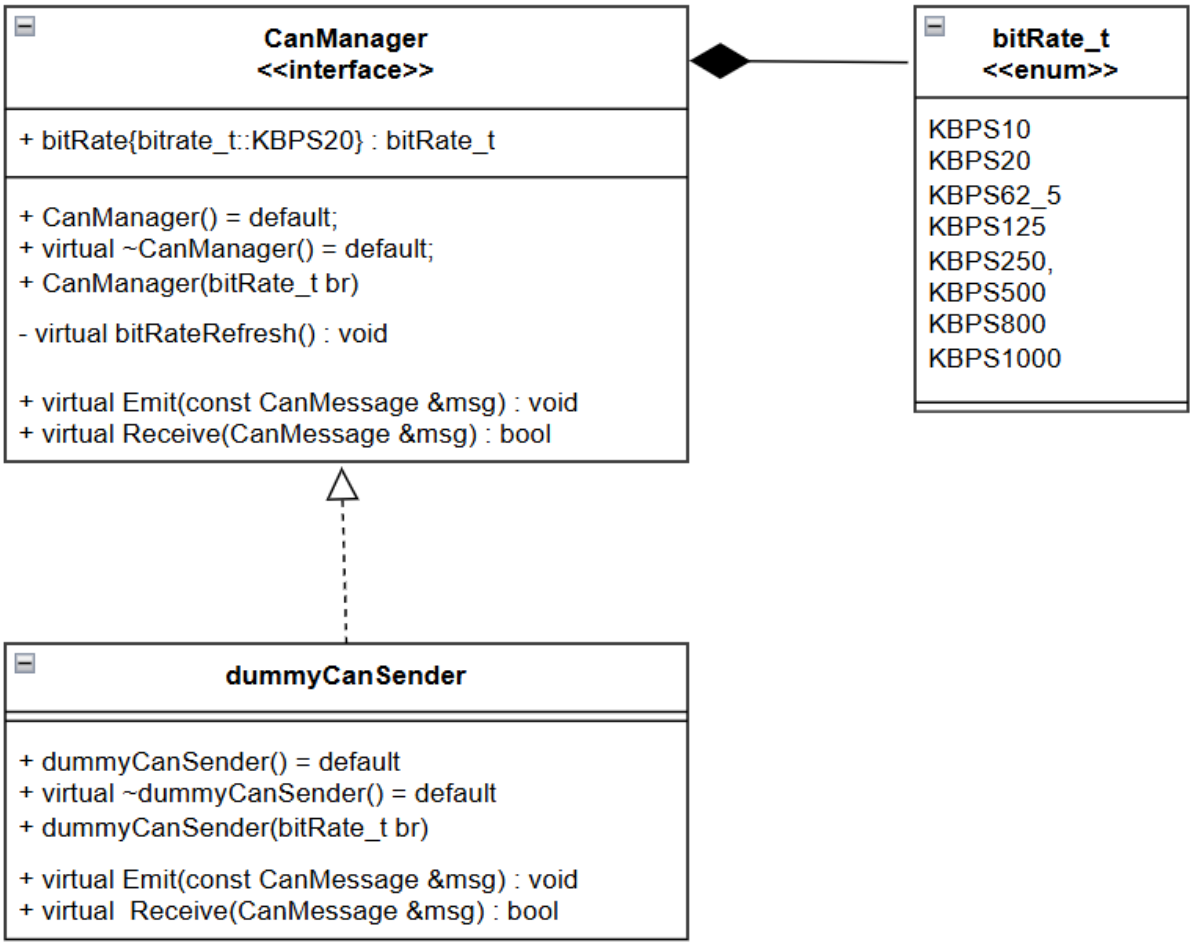
Remarque : Afin de gérer correctement tous les types d'interfaces CAN, une méthode privée virtuelle « bitRateRefresh » est définie. Cette méthode peut être redéfinie et être appelée à chaque mise à jour du débit.

Le code complet de la classe CanManager est fourni :

```
#ifndef __CANMANAGER_HPP__
#define __CANMANAGER_HPP__

#include "CanMessage.hpp"

#include <iostream>
#include <vector>
#include <cstdint>
#include <stdexcept>
#include <algorithm>
#include <print>

/**------------------------------------------------------------------------- */

/**
 * @brief   CanManager Class - Pure virtual class
 */

class CanManager
{
public:
    enum class bitRate_t
    {
        KBPS10,
        KBPS20,
        KBPS62_5,
        KBPS125,
        KBPS250,
        KBPS500,
        KBPS800,
        KBPS1000
    };

private:
    bitRate_t bitRate{bitRate_t::KBPS20}; /** Default bitrate is 20 kbps  */

    /**
     * @ brief Virtual "callback", called after each setbitRate
```

```
     * Empty function if not overrided.
     * */
    virtual void bitRateRefresh() {};

public:
    CanManager() = default;          /** Default Ctor is efaulted */
    virtual ~CanManager() = default; /** Dtor is defaulted    */

    explicit CanManager(bitRate_t br) noexcept : bitRate{br} {}; /** Standard Ctor */

    /**
     * @brief bitRate accessors
     */
    [[nodiscard]] bitRate_t getbitRate() noexcept { return this->bitRate; };
    void setbitRate(bitRate_t br) noexcept
    {
        this->bitRate = br;
        this->bitRateRefresh();
    };

    virtual void Emit(const CanMessage &msg) = 0;     /** Tx function - Pure virtual */
    [[nodiscard]] virtual bool Receive(CanMessage &msg) = 0; /** Rx function-Pure virtual */
};
#endif /*  __CANMANAGER_HPP__ */
```

Annexe 4 – Analyse et Cdc de la classe dummyCanSender



La classe dummyCanSender réalise l'interface CanManager de la manière la plus simple qui soit :

- ✓ La méthode Emit affiche à l'écran la trame à émettre (via toString() de CanMessage).
- ✓ La méthode Receive ne fait rien, et retourne false.