

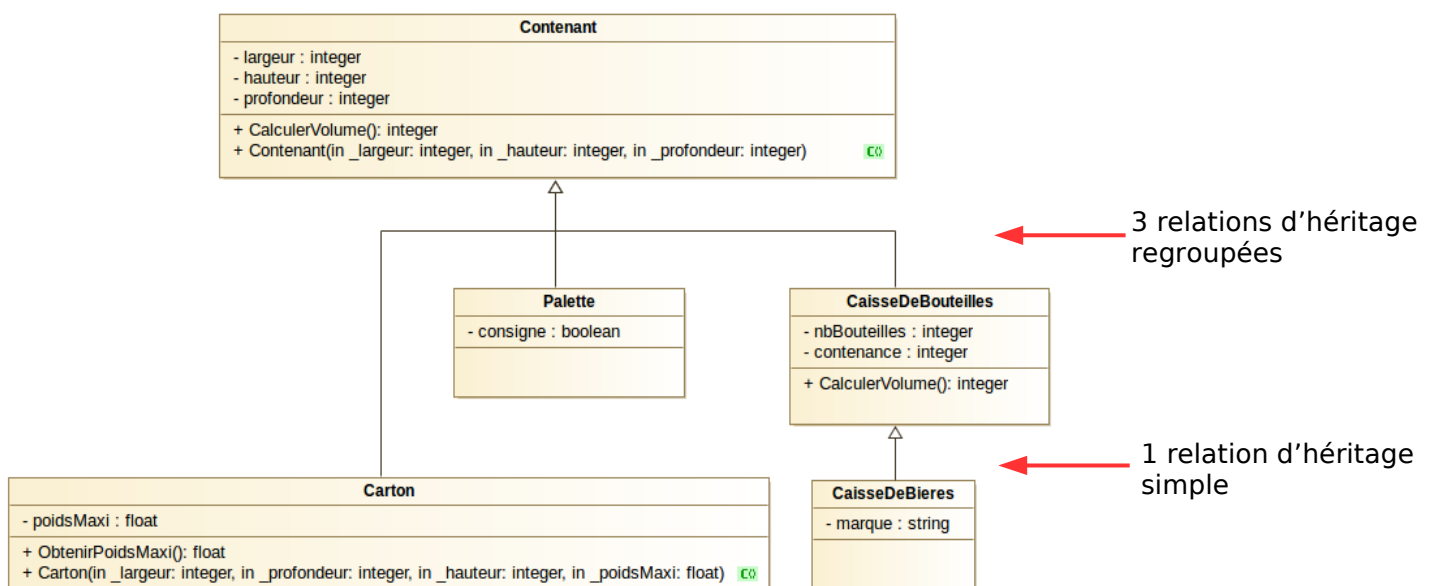
3. Relations entre objets

3.1. Mécanisme d'héritage

3.1.1. Introduction

Le concept d'héritage est l'un des fondements de la programmation orientée objet. Il permet de créer une nouvelle classe nommée **classe dérivée** à partir d'une classe existante, la **classe de base**. La classe dérivée hérite des caractéristiques de sa classe de base, données et fonctions membres dans une certaine mesure. Les seuls membres qui ne sont pas transmis sont les constructeurs, les destructeurs, les membres qui sur définissent l'opérateur d'affectation et les **fonctions amies**.

Cette relation traduit l'idée « **est une sorte de** ». Dans le modèle UML, elle est représentée par un triangle comme la montre la figure ci-après. Ce diagramme de classes est une vue incomplète des différentes classes de la hiérarchie.



L'héritage permet de **réutiliser** le code de la classe de base, tout en gardant la possibilité d'apporter les adaptations nécessaires à chaque cas particulier. Cette opération peut être réalisée avec ou sans les codes sources de la classe de base.

Lorsque le programmeur cherche à regrouper les éléments communs à plusieurs classes dans une même classe, on parle de **généralisation**. Le terme **spécialisation** est retenu lorsque les classes dérivent d'une classe de base.

Carton, **Palette** et **CaisseDeBouteilles** sont des sortes de **Contenant**. Ces classes spécialisent la classe **Contenant** en apportant chacun leurs spécificités. D'un autre point de vue, on peut dire que **Contenant** généralise les classes **Carton**, **Palette** et **CaisseDeBouteilles** en regroupant ce qui est commun aux trois classes.

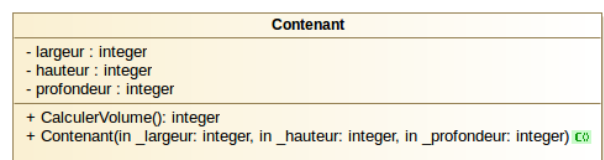
La classe de base se traduit en C++ comme cela est décrit dans le chapitre précédent.

Déclaration de la classe Contenant

contenant.h

```

#ifndef CONTENANT_H
#define CONTENANT_H
class Contenant
{
public:
    Contenant(const int _largeur, const int _hauteur, const int _profondeur);
    int CalculerVolume();
private:
    int largeur;
    int hauteur;
    int profondeur;
};
#endif // CONTENANT_H
  
```

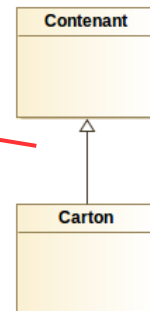


Pour la déclaration de la classe **Carton**, il faut maintenant tenir compte de la relation d'héritage.

Déclaration de la classe Carton

carton.h

```
#ifndef CARTON_H
#define CARTON_H
#include "contenant.h"
class Carton : public Contenant
{
public:
    Carton(const int _largeur, const int _hauteur,
           const int _profondeur, const float _poidsMaxi);
    float ObtenirPoidsMaxi();
private:
    float poidsMaxi;
};
#endif // CARTON_H
```



La déclaration de la classe **Carton** est suivi de « : **public Contenant** » pour indiquer la relation d'héritage entre ces deux classes.

Le constructeur de la classe **Carton** reprend les paramètres de la classe **Contenant** car, il est chargé de les transmettre à celui de la classe de base.

Définition des opérations de la classe Contenant

contenant.cpp

```
#include "contenant.h"
#include <iostream>
using namespace std;

Contenant::Contenant(const int _largeur, const int _hauteur, const int _profondeur):
    largeur(_largeur),
    hauteur(_hauteur),
    profondeur(_profondeur)
{
    cout << "constructeur de la classe Contenant" << endl ;
}

int Contenant::CalculerVolume()
{
    return largeur * hauteur * profondeur ;
}
```

Ici, les attributs sont initialisés en utilisant la liste d'initialisation. Cette méthode est à privilégier, elle est obligatoire pour les constantes et pour initialiser le constructeur d'une classe de base dans le cadre d'un héritage. Dans le cas présent, la définition du constructeur est équivalente au code présenté ci-dessous.

Autre façon de définir le constructeur de la classe Contenant

contenant.cpp

```
Contenant::Contenant(const int _largeur, const int _hauteur, const int _profondeur)
{
    largeur = _largeur;
    hauteur = _hauteur;
    profondeur = _profondeur;
    cout << "constructeur de la classe Contenant" << endl ;
}
```

Exercice d'application

- Après avoir ajouté un destructeur à ces deux classes, dont le rôle est simplement d'afficher de quel destructeur il s'agit, implémentez les deux classes **Contenant** et **Carton** en C++ (Le code source de la classe **Carton** est donné à la page suivante).
- Dans le programme principal, instanciez la classe **Carton** et relevez l'ordre de l'appel des constructeurs et des destructeurs.
- A partir de la hiérarchie de classe présentée en introduction, réalisez la classe **CaisseDeBouteilles**, ajoutez un constructeur et un destructeur. Attention, la méthode **CaisseDeBouteilles::CalculerVolume** retourne le volume contenu dans les bouteilles et non pas le volume brute du contenant.
- Après avoir instancié un objet de type **CaisseDeBouteilles**, comment faire appel à la méthode **Contenant::CalculerVolume()** qui retourne le volume brute ? Testez l'appel dans le programme principal.

L'extrait de code suivant montre le passage de paramètre au constructeur de la classe **Contenant**. C'est toujours la première opération à effectuer lors de la définition du constructeur de la classe dérivée. Il reste ensuite à initialiser le ou les attributs de la classe dérivée.

Définition des opérations de la classe Carton

carton.cpp

```
#include "carton.h"
#include <iostream>

using namespace std;

Carton::Carton(const int _largeur, const int _hauteur, const int _profondeur, const float _poidsMaxi):
    Contenant(_largeur, _hauteur, _profondeur),
    poidsMaxi(_poidsMaxi)
{
    cout << "Constructeur de la classe Carton" << endl;
}

float Carton::ObtenirPoidsMaxi()
{
    return poidsMaxi;
}
```

Remarque

Lors d'un héritage, le constructeur de la classe de base est appelé en premier lieu, ce qui permet au constructeur de la classe dérivée de compléter les initialisations de l'objet. Pour les destructeur, ils sont appelés dans l'ordre inverse, le destructeur de la classe dérivée d'abord puis celui de la classe de base pour finir.

3.1.2. Modalités d'accès aux membres de la classe de base

Dans l'exemple précédent, la classe dérivée n'a pas eu besoin d'accéder aux attributs de la classe de base. Si telle avait été le cas, cela n'aurait pas été possible. Le compilateur refuse systématiquement l'accès aux membres privés d'une classe même dans le cadre d'un héritage, c'est le principe de l'encapsulation.

Outre les statuts **public** et **privé** vus jusqu'à maintenant, il existe un statut **protégé**. Un membre protégé se comporte comme un membre privé pour une utilisation quelconque de la classe ou de la classe dérivée, mais comme un membre public pour la classe dérivée.

Pour la classe **Contenant**, on peut imaginer qu'il s'agit d'une erreur de conception. Dans le cadre d'une classe sujette à devenir une classe de base, le statut **protégé** sauf cas particulier est à privilégier. La Déclaration de la classe devient donc :

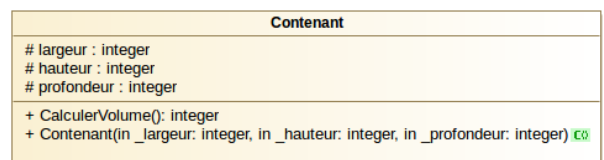
Déclaration de la classe Contenant

contenant2.cpp

```
#ifndef CONTENANT_H
#define CONTENANT_H

class Contenant
{
public:
    Contenant(const int _largeur, const int _hauteur, const int _profondeur);
    int CalculerVolume();
protected:
    int largeur;
    int hauteur;
    int profondeur;
};

#endif // CONTENANT_H
```



On remarque dans la représentation UML de la classe le signe # devant chaque attribut, traduit en C++ par le mot clé **protected**.

L'utilisation du qualificateur **protected** ne fait pas bénéficier d'une encapsulation aussi complète que le qualificateur **private**, mais rend les attributs **largeur**, **hauteur** et **profondeur** accessibles dans les classes dérivées.

3.1.3. Dérivations publique, protégée et privée

Il est possible de contrôler les accès aux membres de la classe dérivée, quels que soient les choix qui avaient été effectués au niveau de la classe de base. En particulier, si celle-ci autorise des accès jugés trop libéraux.

Dérivation publique

Les membres de la classe de base conservent leur statut dans la classe dérivée. C'est la situation la plus usuelle.

Dérivation publique

```
class Carton : public Contenant ;
```

Dérivation protégée

Les membres publics de la classe de base deviennent membres protégés de la classe dérivée. Les autres membres conservent leur statut.

Dérivation publique

```
class Carton : protected Contenant ;
```

Dérivation privée

Tous les membres de la classe de base deviennent privés dans la classe dérivée. Par défaut, la dérivation est privée.

Dérivation publique

```
class Carton : private Contenant ;
```

ou

```
class Carton : Contenant ;
```

Par contre

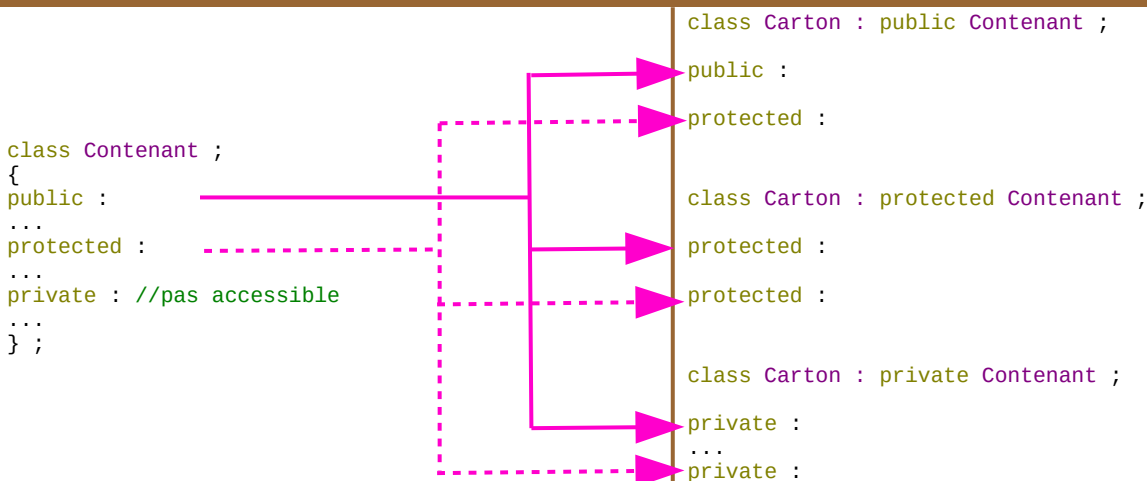
L'utilisation de dérivations protégées ou privées rend plus complexe la détermination du droit d'accès aux membres de chaque classe et donne donc un programme moins lisible.

De plus, les conversions implicites définies précédemment ne s'appliquent que pour l'héritage public.

Il est donc conseillé d'utiliser les dérivations « privée » et « protégée » avec prudence.

Voici un résumé du devenir des éléments de la classe de base dans la classe dérivée en fonction du type d'héritage

Résumé :



Exercice d'application

Les établissements **mLego**, une fonderie à **Boëssé le sec** (72), sont fabricant d'alliages de cuivre depuis 1894. Leur principale production est basée sur des barres de formes et de dimensions diverses en alliage de cuivre, d'aluminium et autres matières.

Chaque barre est caractérisée par une référence, une longueur, la densité et le nom de l'alliage qui la compose.



Étude de la classe *Barre* :

1. Déclarer et définir la classe **Barre** pertinente. Les paramètres pour l'initialisation seront passés au constructeur de la classe. La référence et le nom seront définis en utilisant le type **string** de la librairie standard. L'ensemble des attributs de cette classe sera privé.
2. Déclarer et définir une méthode **AfficherCaractéristiques** » en utilisant le flux de sortie standard de la librairie **iostream**.

Études des différentes formes :

Comme le montre l'extrait du catalogue de la société, les barres sont de différentes formes rondes, hexagonales, rectangulaires, carrées, creuses, pleines...



3. Déterminer pour chacune de ces formes le nombre de paramètres nécessaire au calcul de la section. Répondre sous la forme d'un tableau.
4. Déclarer et définir les classes **BarreRonde**, **BarreRectangle** et **BarreCarree** disposant des attributs nécessaires pour calculer leur section respective. Chaque classe hérite de la classe de base **Barre**. Les paramètres d'initialisations des attributs seront passés par le constructeur et transmis le cas échéant au constructeur de la classe de base. Indiquer l'ordre de l'appel des constructeurs et des destructeurs.

Pour rappel, on donne ici les formules de calcul nécessaire pour les différentes formes de barre.

 Cercle	$S_{\text{Cercle}} = \pi * d^2 / 4$	d représente ici, le diamètre du cercle
 Rectangle	$S_{\text{Rectangle}} = L * l$	L représente ici la longueur et l la largeur du rectangle. Pour un carré, la longueur et la largeur sont identiques.
 Hexagone	$S_{\text{Hexagone}} = 2 \sqrt{(3 * d_i^2 / 4)}$	di représente, ici, le diamètre du cercle inscrit de l'hexagone
 Octogone	$S_{\text{Octogone}} = 2 * d_i^2 * (\sqrt{2} - 1)$	di représente, ici, le diamètre du cercle inscrit de l'octogone

5. Ajouter à chaque sous-classe la méthode **CalculerSection** et son code.
6. Ajouter à chaque sous-classe une méthode **CalculerMasse** et son code. Pour rappel, la masse d'une barre est donnée par la formule :

$$\text{masse} = \text{longueur} * \text{section} * \text{densité}$$

Les attributs nécessaires à ce calcul dans la classe de base sont-ils accessibles dans la classe dérivée ? Réaliser la modification si nécessaire. Réaliser le codage de la méthode pour chaque classe déclarée précédemment.

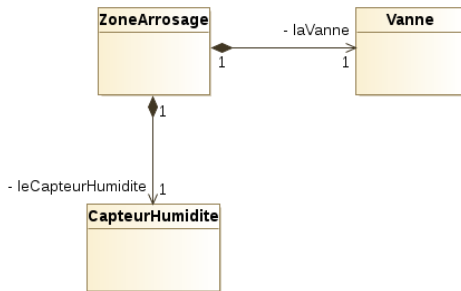
7. Déclarer une instance de chaque type de barre et compléter le code du programme principal pour faire afficher les caractéristiques et la masse de chaque barre.

3.2. Mécanisme de composition

3.2.1. Introduction

La relation de composition implique une notion d'appartenance. Le cycle de vie des éléments composants est lié à celui de l'agrégat, ou élément contenant. Si l'agrégat est détruit ou copié, ses composants le sont aussi. À un même instant une instance composante ne peut être liée qu'à un seul et unique agrégat.

Cette relation traduit l'idée « **est constituée physiquement de** ». Dans le modèle UML, elle est représentée par un losange noir plein, comme le montre la figure ci-après. La relation est généralement orientée, le sens de la flèche indique la navigabilité, de la classe agrégat, celle du côté losange, vers la classe agrégée.



L'exemple ci-contre représente une étude partielle d'un contrôleur de serre chargé de l'arrosage. Ici, une zone d'arrosage est composée d'une vanne et d'un capteur d'humidité.

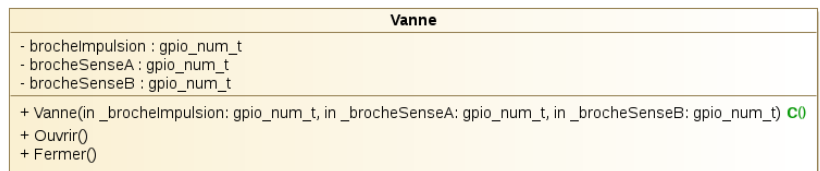
La relation de composition est représentée en UML par le losange plein. Les deux relations étant orientées, c'est bien la classe **ZoneArrosage** qui contient un objet nommé **laVanne** une instance de la classe **Vanne** et un autre objet nommé **leCapteurHumidite** une instance de la classe **CapteurHumidite**. Le nombre d'instances est déterminé par la cardinalité, ici 1 pour chacune des relations.

Les rôles, noms donnés à **laVanne** et **leCapteurHumidite**, représentent deux attributs de la classe **ZoneArrosage**, ils ne sont pas directement représentés dans la classe agrégat, comme le montre la figure.

3.2.2. Implémentation en C++

Comme la durée de vie d'un composant dépend de la classe agrégat, c'est cette dernière qui doit créer l'instance du composant et la détruire. Elle peut être déclarée de manière automatique ou de manière dynamique.

Le détail de la classe **Vanne** est donné sous sa représentation **UML** ci-contre. Le type **gpio_num_t** désigne une broche d'entrée / sortie sur un micro-contrôleur type **ESP32**, il est équivalent à un entier désignant un numéro de broche.



La déclaration de la classe **Vanne** et de la classe **CapteurHumidite** est tout à fait classique, voici celle de la classe **Vanne**.

Déclaration de la classe Vanne

vanne.h

```

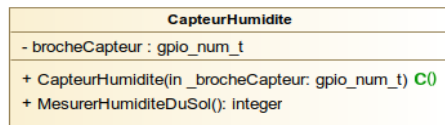
#ifndef VANNE_H
#define VANNE_H
class Vanne
{
public:
    Vanne(const gpio_num_t _brocheImpulsion, const gpio_num_t _sensA, const gpio_num_t _sensB);
    void Ouvrir();
    void Fermer();
private:
    gpio_num_t impulsions;
    gpio_num_t sensA;
    gpio_num_t sensB;
};
#endif // VANNE_H
  
```

Pour la compilation dans un environnement C++ classique, **gpio_num_t** peut être redéfini en **int**

```
#define gpio_num_t int
```

Voici la représentation **UML** de la classe **CapteurHumidite**. La méthode **MesurerHumiditeDuSol** retourne un entier, le pourcentage d'humidité mesuré dans le sol de la serre.

Exercice d'application



1. À titre d'exercice, réalisez la déclaration de cette classe en C++.

Implémentation de la relation sous la forme d'une instance automatique :

Dans ce cas de figure, l'attribut représentant le composant est une simple instance. Pour l'exemple, seule la composition avec la classe **Vanne** est prise en considération.

Déclaration de la classe ZoneArrosage

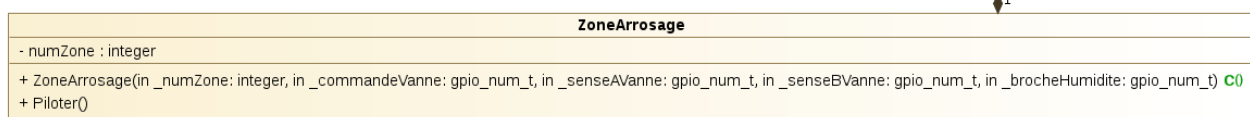
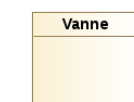
zonearrosage.h

```
#ifndef ZONEARROSAGE_H
#define ZONEARROSAGE_H

#include "vanne.h"

class ZoneArrosage
{
private:
    Vanne laVanne;
    int numZone;
public:
    ZoneArrosage(int _numZone,
                  const gpio_num_t _commandeVanne,
                  const gpio_num_t _senseAVanne,
                  const gpio_num_t _senseBVanne,
                  const gpio_num_t _brocheHumidite);
    void Piloter();
};

#endif // ZONEARROSAGE_H
```



Dans le cas présent, l'instanciation n'est pas suffisante, car le constructeur de la classe `Vanne` possède trois paramètres qu'il faut initialiser. Le passage de paramètres ne pouvant se faire dans le fichier d'en-tête `zonearrosage.h`, il doit être fait lors de l'implémentation du constructeur.

Implémentation du constructeur de la classe `ZoneArrosage`

`zonearrosage.cpp`

```
ZoneArrosage::ZoneArrosage(const int _numZone, const int _commandeVanne, const int _senseAVanne,
                           const int _senseBVanne, const int _brocheHumidite):
    laVanne(_commandeVanne, _senseAVanne, _senseBVanne),
    numZone(_numZone)
{
    // reste du code pour le constructeur
}
```

Le passage de paramètres doit se faire **impérativement** avec la liste d'initialisation comme cela avait été le cas dans la relation d'héritage.

2. Complétez la déclaration de la classe **ZoneArrosage** pour faire apparaître la deuxième relation de composition avec la classe **CapteurHumidite**.
3. Modifiez l'implémentation du constructeur afin de prendre en compte cette deuxième relation.

À retenir

Lorsque le constructeur du composant ne possède pas de paramètre, il peut être instancié simplement dans la classe agrégat. Dans le cas contraire, comme l'a montré l'exemple, il est nécessaire de passer les paramètres au travers de la liste d'initialisation.

Implémentation de la relation sous la forme d'une instance dynamique :

Cette fois-ci, l'attribut ne représente pas directement une instance du composant. La relation va être implémentée sous la forme d'un pointeur que le constructeur **doit initialiser dynamiquement**. Un destructeur est également nécessaire pour libérer la mémoire allouée dans le constructeur. En procédant de la sorte, le comportement reste identique, le constructeur crée le composant et le destructeur le détruit, à la différence que c'est, ici, le programmeur qui prend à sa charge ces opérations.

Dans la déclaration ci-après, **laVanne** est maintenant un pointeur sur la classe **Vanne**, pointeur qu'il est nécessaire d'initialiser dans le constructeur.

Certains peuvent se demander, pourquoi faire aussi compliqué puisque la première version fonctionnait très bien sans intervention du programmeur ?

Parfois, le constructeur de l'agrégat ne possède pas directement les éléments pour le passage de paramètres au constructeur du composant. Ces éléments peuvent être lus dans un fichier ou obtenus sous une autre forme, dans ce cas l'implémentation sous la forme d'un pointeur s'avère très utile.

Déclaration de la classe `ZoneArrosage`

`zonearrosage.h`

```
#ifndef ZONEARROSAGE_H
#define ZONEARROSAGE_H
#include "vanne.h"
class ZoneArrosage
{
private:
    Vanne *laVanne;
    int numZone;
public:
    ZoneArrosage(const string _initialisationZone);
    void Piloter();
};
#endif // ZONEARROSAGE_H
```

Ici, les paramètres sont passés sous la forme d'une chaîne de caractères au lieu d'être transmis sous la forme de plusieurs entiers comme dans la version précédente.

Le programme principal peut alors s'écrire de la manière suivante :

Programme principal

serre.cpp

```
#include "zonearrosage.h"
int main(int argc, char *argv[])
{
    ZoneArrosage zone1("1 25 15 5"); // zone 1, impulsion sur 25, sens A=15 et B=5
}
```

Et, voici le code du constructeur : il est chargé de décomposer la chaîne et instancier la classe **Vanne**.

Implémentation du constructeur

zonearrosage.cpp

```
ZoneArrosage::ZoneArrosage(const string _initialisationZone)
{
    int parametres[4];
    size_t prec = 0;
    size_t pos = 0;
    for (int indice = 0; indice < 4; indice++)
    {
        pos = _initialisationZone.find(' ', prec);
        parametres[indice] = atoi(_initialisationZone.substr(prec, pos).c_str());
        prec = pos + 1; // on incrémente pos pour le tour d'après.
    }
    numZone = parametres[0];
    laVanne = new Vanne(parametres[1], parametres[2], parametres[3]);
}
```

Le destructeur libère simplement la mémoire allouée dynamiquement.

Destructeur de la classe ZoneArrosage

zonearrosage.cpp

```
ZoneArrosage::~ZoneArrosage()
{
    delete laVanne;
}
```

À retenir

Dans cette deuxième version pour l'implémentation de la relation de composition, le programmeur prend la charge de créer l'instance du composant dynamiquement et de restituer la mémoire au sein de la classe agrégat.

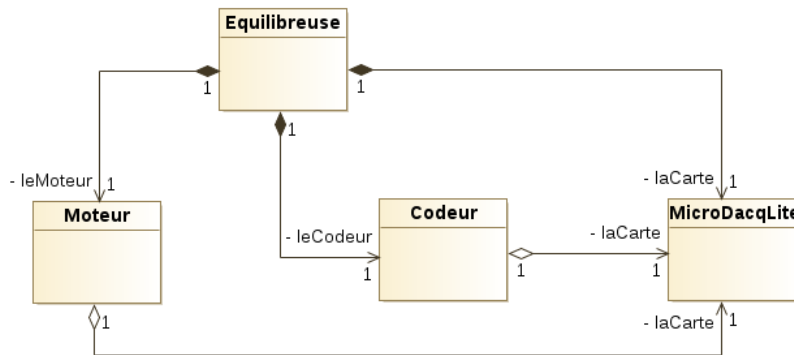
4. Ajoutez les éléments nécessaires pour implémenter la deuxième relation de composition avec la classe **CapteurHumidite** de manière dynamique. Dans le programme principal, ajouter un nouveau paramètre à la chaîne d'initialisation par exemple 12 qui représente la broche utilisée par le capteur d'humidité.
5. Dans chaque constructeur, affichez vers la sortie standard le nom de la classe et la valeur des attributs afin de vérifier le bon fonctionnement de votre implémentation.

3.3. Mécanisme d'agrégation.

3.3.1. Introduction

L'agrégation représente une relation de subordination et exprime un couplage fort. Il est cependant moins important que dans la relation de composition. Dans ce cas de figure, la durée de vie des objets « agrégat » et « agrégé » est indépendante. L'instance d'une classe agrégée peut être liée à d'autres objets à un même moment. L'agrégation se représente en UML par le losange vide. Pour la relation de composition, on pouvait parler d'agrégation forte ou par valeur.

L'exemple suivant présente un extrait de l'étude d'une maquette d'équilibrage. La classe **Equilibreuse** est composée d'une instance **leMoteur** d'une classe **Moteur** et d'une instance **leCodeur** d'une classe **Codeur**. Dans le même temps, chacune des classes **Moteur** et **Codeur** utilise une classe nommée **MicroDacqLite**. Cette dernière implémente des méthodes pour piloter physiquement le moteur et lire les informations en provenance du codeur, mais elle ne doit être instanciée qu'une seule fois. C'est donc la classe **Equilibreuse** qui va réaliser cette implémentation, elle est physiquement composée de cette carte d'acquisition, et les deux autres classes posséderont une relation d'agrégation avec cette classe. La figure suivante représente une vue partielle du diagramme de classes UML.



On remarque, les 3 relations de composition, les losanges noirs pleins et les 2 relations d'agrégation, les losanges vides.

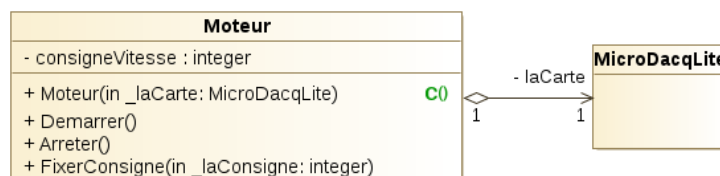
Dans les cas, les relations sont orientées, de la classe agrégat vers une classe agrégée. La cardinalité est également à 1, ce qui signifie : une seule instance.

3.3.2. Implémentation en C++

La relation d'agrégation s'implémente sous la forme d'une référence en C++.

Ainsi, la classe **Moteur** et la classe **Codeur** disposeront chacune d'une référence sur la classe **MicroDacqLite**. Cela reprend bien l'idée d'un alias vers cette classe. La classe **Equilibreuse** crée l'instance **MicroDacqLite** en premier lieu et en transmet une référence au constructeur de ses deux autres composants. La classe **Moteur** et la classe **Codeur** doivent posséder chacune un attribut nommé **laCarte**, comme l'indique le rôle sur la relation d'agrégation, de type référence sur la classe **MicroDacqLite**.

Le détail de la classe **Moteur** est représenté en UML par la figure suivante :



De la même manière que pour la composition, l'attribut **laCarte** n'apparaît pas avec l'attribut **consigneVitesse** puisqu'il est déjà représenté sous la forme d'un rôle au niveau de la relation.

La déclaration en C++ de cette relation peut se traduire de la manière suivante :

Déclaration de la classe Moteur

moteur.h

```
#ifndef MOTEUR_H
#define MOTEUR_H

class MicroDacqLite;

class Moteur
{
private:
    int consigneVitesse;
    const MicroDacqLite & laCarte; // Pour la relation d'agrégation,
                                   // l'attribut est ici constant pour être en accord avec le passage de
                                   // paramètres. Il n'est pas modifié dans la classe Moteur.

public:
    Moteur(const MicroDacqLite & _laCarte);
    void Demarrer();
    void Arrêter();
    void FixerConsigne(const int _laConsigne);
};

#endif // MOTEUR_H
```

À noter, pour éviter tous problèmes d'inclusion circulaire, le fichier *microdacqlite.h* n'a pas été inclus dans le fichier *moteur.h*. On y trouve simplement le fait que **MicroDacqLite** est une classe.

Pour ce qui concerne la classe **Equilibreuse**, il s'agit de compositions, on propose ici d'implémenter de manière automatique la classe **MicrodacqLite** et de manière dynamique les deux autres. On s'intéresse, dans un premier temps, aux constructeurs des classes différentes classes.

Déclaration de la classe Equilibreuse

equilibreuse.h

```
#ifndef EQUILIBREUSE_H
#define EQUILIBREUSE_H

#include "moteur.h"
#include "microdacqlite.h"

class Equilibreuse
{
private:
    MicroDacqLite laCarte;
    Moteur *leMoteur;
public:
    Equilibreuse();
};

#endif // EQUILIBREUSE_H
```

Le constructeur de la classe **Equilibreuse** se traduira simplement de la manière suivante :

Constructeur de la classe Equilibreuse

equilibreuse.cpp

```
#include "equilibreuse.h"
Equilibreuse::Equilibreuse()
{
    leMoteur = new Moteur(laCarte);
}
```

Il ne reste plus qu'à étudier le constructeur de la classe **Moteur** pour voir l'ensemble du processus.

Constructeur de la classe Moteur

moteur.cpp

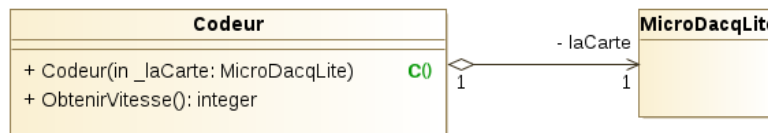
```
#include "moteur.h"
#include "microdacqlite.h"

Moteur::Moteur(const MicroDacqLite &_laCarte) :
    laCarte(_laCarte) // initialisation de la relation d'agrégation
{
    consigneVitesse = 0;
}
```

Le fichier *microdacqlite.h* est inclus ici, pour que la classe **Moteur** puisse faire appel aux différentes méthodes de la classe **MicroDacqLite** et palier au fait qu'il n'a pas été inclus dans le fichier *moteur.h*.

Exercice d'application

1. Réalisez la déclaration de la classe **Codeur** dont la représentation UML est donnée ici.



2. Complétez la déclaration de la classe **Equibreuse** pour réaliser la relation avec la classe **Codeur**.
3. Réalisez le code du constructeur de la classe **Codeur** et complétez celui de la classe **Equibreuse**.
4. Un destructeur explicite est-il nécessaire dans la classe **Equibreuse** ? Si oui, complétez la déclaration de la classe **Equibreuse** et codez ce destructeur.

À retenir

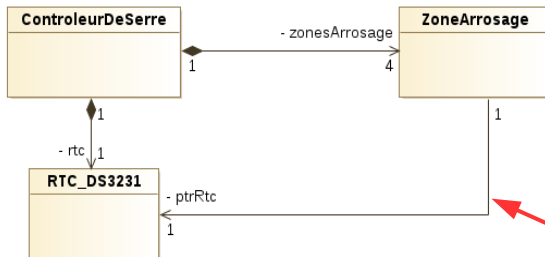
Dans le cas de la relation d'agrégation, l'objet agrégé n'est pas instancié dans le constructeur. Il est simplement initialisé à l'aide d'un paramètre passé au constructeur.

3.4. Mécanisme d'association

3.4.1. introduction

La relation d'association est similaire à la relation d'agrégation vue précédemment dans le sens où la durée de vie des objets n'est pas liée. Elle indique cependant un lien moins fort que l'agrégation et par conséquent que la composition. La symbolique UML pour la représenter est un simple trait entre deux classes. Elle peut être navigable dans les deux sens ou uniquement dans un seul, dans ce cas une flèche marque le sens de la navigabilité.

Pour illustrer ce propos, l'exemple du contrôleur de serre va être approfondi :



La poursuite de l'étude montre qu'un contrôleur assure le contrôle de 4 zones d'arrosage et qu'il dispose d'un dispositif d'horloge temps réel basé sur un composant DS3231. Chaque zone d'arrosage ayant également besoin de faire appel à ce module, une association a été mise en place vers ce composant.

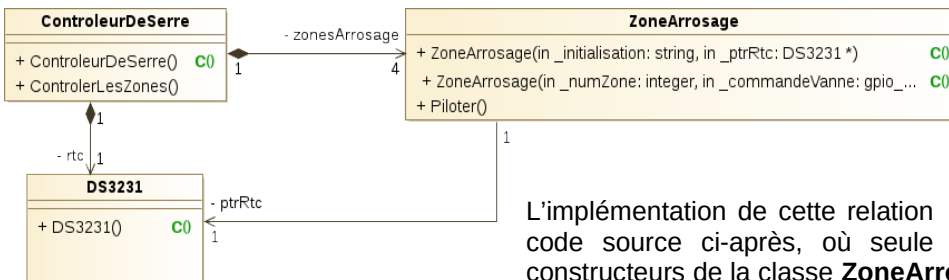
Relation d'association

3.4.2. Implémentation en C++

L'implémentation de la relation d'association est réalisée par un pointeur, comme elle peut l'être avec la relation de composition, mais l'instanciation est faite par ailleurs comme pour la relation d'agrégation.

La classe **ControleurDeSerre** va créer une instance nommée **rtc** de la classe **RTC_DS3231** puis 4 instances implémentées sous la forme d'un tableau nommé **zonesArrosage** de la classe **ZoneArrosage**. En effet, on constate que la cardinalité pour la relation vers la classe **ZoneArrosage** est égale à 4, d'où l'utilisation d'un tableau pour stocker les 4 instances. Le constructeur de la classe **ZoneArrosage** doit également recevoir un pointeur sur la classe **RTC_DS3231** pour initialiser son attribut **ptrRtc**.

La représentation UML des différentes classes est montrée à la figure ci-après :



L'implémentation de cette relation d'association est donnée dans le code source ci-après, où seule la deuxième implémentation du constructeurs de la classe **ZoneArrosage** été codée.

Déclaration de la classe ZoneArrosage

zonearrosage2.h

```

#ifndef ZONEARROSAGE_H
#define ZONEARROSAGE_H

#include "vanne.h"
class DS3231;

class ZoneArrosage
{
private:
    const DS3231 *ptrRTC;
    int numZone;
public:
    ZoneArrosage(const string _initialisationZone, const DS3231 * _ptrRTC);
    void Piloter();
};

#endif // ZONEARROSAGE_H
  
```

La déclaration de la classe **ContrôleurDeSerre** montre l'implémentation des deux compositions, dont une multiple pour la classe **ZoneArrosage**, la cardinalité est de 4 dans le diagramme de classes.

Déclaration de la classe **ContrôleurDeSerre***contrôleurdeserre.h*

```
#ifndef CONTROLEURDESERRE_H
#define CONTROLEURDESERRE_H

#include "zonearrosage.h"
#include "DS3231.h"

class ContrôleurDeSerre
{
private:
    DS3231 rtc;
    ZoneArrosage * zonesArrosage[4];
public:
    ContrôleurDeSerre();
    ~ContrôleurDeSerre();
    void ContrôlerLesZones();
};

#endif // CONTROLEURDESERRE_H
```

Un tableau de pointeur a été utilisé ici afin de réaliser une initialisation dynamique des 4 instances.

Le constructeur est chargé d'instancier 4 fois la classe **ZoneArrosage**. L'initialisation est réalisée à partir des données lues dans un fichier texte.


Implémentation du constructeur et du destructeur de **ContrôleurDeSerre***contrôleurdeserre.cpp*

```
#include <fstream>
#include <iostream>
#include <string>
#include "contrôleurdeserre.h"

using namespace std;

ContrôleurDeSerre::ContrôleurDeSerre()
{
    ifstream fichier("config.txt");
    if (!fichier.is_open())
        cerr << "Erreur lors de l'ouverture du fichier" << endl;
    else
    {
        string init;
        for (int indice = 0; indice < 4; indice++)
        {
            getline(fichier, init);
            if (fichier.good())
                zonesArrosage[indice] = new ZoneArrosage(init, &rtc);
        }
        fichier.close();
    }
}

ContrôleurDeSerre::~ContrôleurDeSerre()
{
    for (int indice = 0; indice < 4; indice++)
    {
        delete zonesArrosage[indice];
    }
}
```



config.txt				
Fichier	Édition			
1	21	15	5	
2	22	15	5	
3	23	15	5	
4	24	15	5	

Transmission de l'adresse de l'instance
rtc pour finaliser l'association

Pour la classe **ZoneArrosage**, le constructeur reste identique à la version présentée précédemment en ajoutant l'initialisation du pointeur pour finaliser la relation d'association avec la classe **DS3231**.

Implémentation du constructeur de la classe ZoneArrosage

zonearrosage2.cpp

```
#include <iostream>
#include "zonearrosage.h"

using namespace std;

ZoneArrosage::ZoneArrosage(const string _initialisation, const DS3231 * _ptrRTC):
    ptrRTC(_ptrRTC)
{
    int parametres[4];
    size_t prec = 0;
    size_t pos = 0;
    for (int indice = 0; indice < 4; indice++)
    {
        pos = _initialisation.find(' ', prec);
        parametres[indice] = atoi(_initialisation.substr(prec, pos).c_str());
        prec = pos + 1; // on incrémente pos pour le tour d'après.
    }
    numZone = parametres[0];
    cout << "Zone " << numZone << " ";
    laVanne = new Vanne(parametres[1], parametres[2], parametres[3]);
}
```

Initialisation du pointeur

Pour les besoins de la compilation, la classe DS3231 peut être définie de la manière suivante :

Compléments pour la compilation

----- Fichier ds3231.h -----

```
#ifndef DS3231_H
#define DS3231_H

class DS3231
{
public:
    DS3231();
};

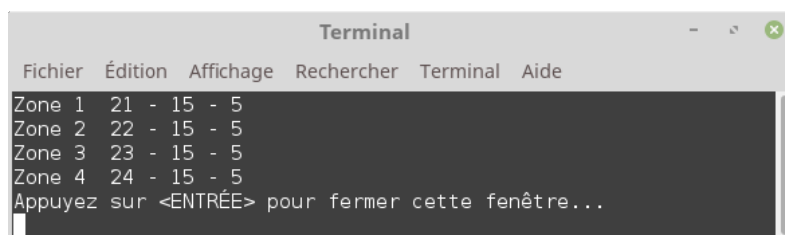
#endif // DS3231_H
```

----- Fichier ds3231.cpp -----

```
#include "ds3231.h"
DS3231::DS3231()
{
}

----- Fichier main.cpp -----
#include "controleurdeserre.h"

int main(int argc, char *argv[])
{
    ControleurDeSerre controleur;
}
```



Remarque

L'utilisation du premier constructeur de la classe **ZoneArrosage** doit également être modifiée, car jusqu'à présent il n'initialise pas le pointeur sur la classe **DS3231**. L'appel d'une de ces méthodes pourraient engendrer un plantage du programme. **Un pointeur, comme toute variable, doit toujours être initialisé avant son utilisation.**

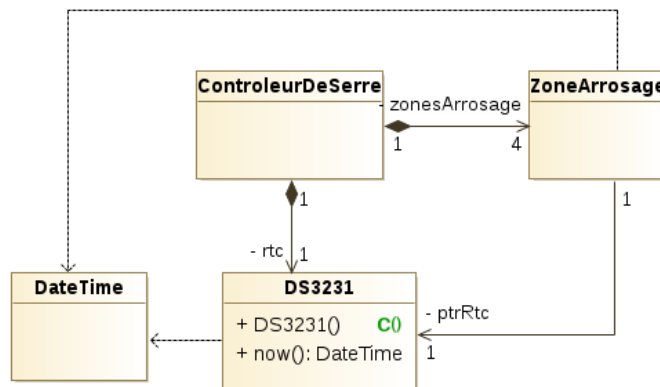
Exercice d'application

1. Modifiez et complétez le code de ce 1^{er} constructeur vu lors de l'implémentation automatique de la relation de composition.

3.5. Notion de dépendance

Une dernière relation peut apparaître dans les diagrammes de classe, c'est la relation de dépendance. Elle exprime le fait que deux classes ne nécessitent pas forcément de lien structuré entre leurs objets. Lorsque cette relation est réalisée, elle est limitée dans le temps. Typiquement, cette relation existe lorsqu'une méthode déclare localement un nouvel objet et n'est donc pas attribut de la classe.

La relation est représentée par une flèche avec un trait en pointillé. La Classe **DS3231** utilise pour fournir la date et l'heure courante une instance de la classe **DateTime** dans la méthode **now()**. De même, chaque zone d'arrosage a besoin de connaître cette heure courante pour déclencher ou arrêter l'arrosage. Une de ces méthodes utilise donc également une instance de la classe **DateTime**.



Cette relation a tendance à surcharger les diagrammes et donc rendre moins lisibles. Chaque fois qu'une méthode utilise une instance d'une autre classe, il y a relation de dépendance. Elle doit apparaître uniquement pour montrer un point important que le concepteur souhaite souligner.

À retenir

La relation de dépendance n'est généralement pas représentée dans un diagramme de classes pour éviter de le surcharger. Elle se traduit en C++ par une instance locale à une méthode, en aucun cas par un attribut.