LANGAGE C++

Les associations

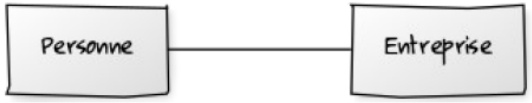
4 types d’ associations :

- standard
- agrégation
- composition
- dépendance

Principe

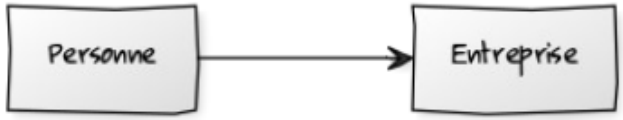
Une association permet d'accéder aux services (méthodes et attributs publics) des instances de la classe associée.

- **Notation UML** :



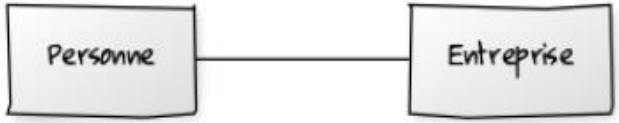
- **Navigabilité des associations**:

Accessibilité a sens unique :

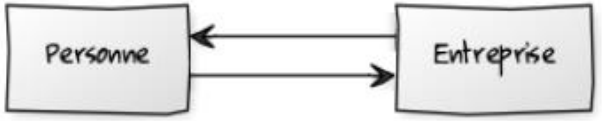


Une instance de Personne accède aux service d'une instance d 'Entreprise

Accessibilité bidirectionnelle:



équivalent à:

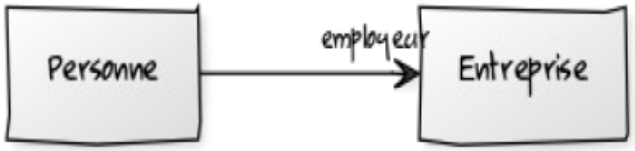


Une instance de Personne accède aux service d'une instance d 'Entreprise et vice-versa

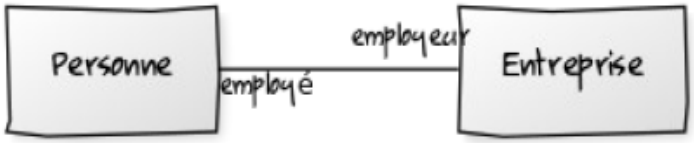
Principe

Chaque association possède un rôle (un nom)

- **Notation UML** :
  - Placer en bout de flèche de l'association
  - Casse : en minuscule
  - En pratique : un substantif



employeur est un attribut de Personne de type Entreprise



employeur est un attribut de Personne de type Entreprise

employé est un attribut d'Entreprise de type Personne

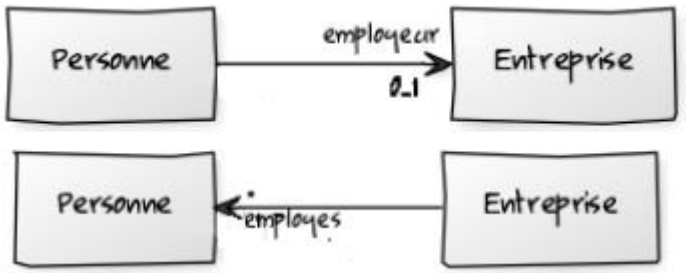
Principe

Multiplicité des associations : elle définit le nombre d'instances de l'association pour une instance de la classe.

- Un nombre entier ou un intervalle de valeurs.

1	Une et une seule instance (par défaut)
0..1	Zéro ou une instance
M..N	De M à N instances
*	De zéro à plusieurs instances
1..*	De 1 à plusieurs instances
N	Exactement N instances

Notation UML :



Une instance de type Personne est en association avec au maximum une instance de type Entreprise

Une instance de type d'entreprise est en association avec plusieurs instances de type Personne

Principe

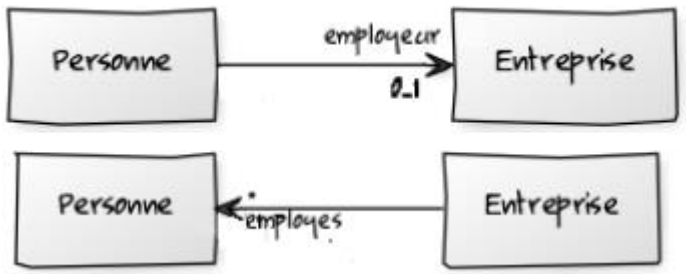
Multiplicité des associations : elle définit le nombre d'instances de l'association pour une instance de la classe.

- Un nombre entier ou un intervalle de valeurs.

1	Une et une seule instance (par défaut)
0..1	Zéro ou une instance
M..N	De M à N instances
*	De zéro à plusieurs instances
1..*	De 1 à plusieurs instances
N	Exactement N instances

Notation UML :

- *Notée avec le rôle.*
- *Par défaut 1 (non notée).*



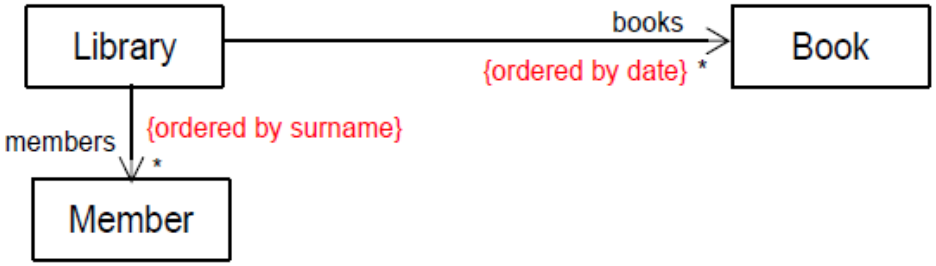
Une instance de type Personne est en association avec au maximum une instance de type Entreprise

Une instance de type d'entreprise est en association avec plusieurs instances de type Personne

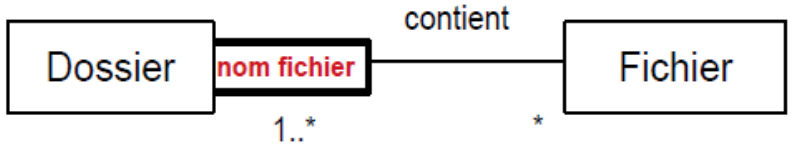
Principe

Contraintes sur les associations:

- Association **ordonnée**.
  - unordered (c'est la valeur par défaut).
  - ordered : les associés sont stockés dans un ordre précis.



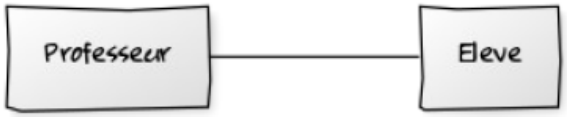
- Association **qualifiée** : tableau associatif, table de hachage, dictionnaire.
  - Les éléments sont accessibles par une clé (p.ex, *le nom du fichier*).



Association standard

- **C'est la relation par défaut.**
  - Relation de type « connaît ».

- **Notation UML**



Association bidirectionnelle, sans représentation de rôle, ni de cardinalité (donc 1)
Une instance de type Professeur connait une instance de type Eleve et vice-versa.

- **En C++**

On utilise
les pointeurs !!!

```
#include <iostream>
#include <string>


using namespace std;

class Eleve;
class Professeur
{
    private:
        string nom;
        Eleve * monEleve;

     public:
        Professeur(string name);
        ~Professeur();
        void setEleve(Eleve * el);
        string getEleve();
        string getNom();
};
```

```
#include "eleve.h"
#include "professeur.h"

Professeur::Professeur(string name):nom(name),monEleve(0)
{

}

void Professeur::setEleve(Eleve * el)
{
            monEleve = el; // association professeur->eleve

}

string Professeur::getEleve()
{
            if(monEleve != 0)
            {
                return monEleve->getNom();
            }
}

string Professeur::getNom(){return nom;}

Professeur::~Professeur()
{
    //dtor
}
```

Association standard

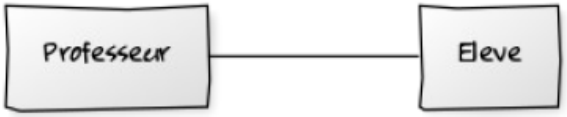
- **En C++**

On utilise
les pointeurs !!!

```
#include <iostream>
#include <string>
#include "eleve.h"
#include "professeur.h"

using namespace std;

int main()
{
    cout << "Association standard bidirectionnelle" << endl;

    Professeur unProf("Tourensol");
    cout << unProf.getNom() << endl;

    Eleve unEleve("Tintin");
    cout << unEleve.getNom() << endl;

    unProf.setEleve(&unEleve); // mise en relation professeur->eleve
    unEleve.setProf(&unProf); // mise en relation eleve->professeur

    cout << "(" << unProf.getNom() << ") : Mon eleve est " << unProf.getEleve()<< endl;
    cout << "("<<  unEleve.getNom() << ") : mon professeur est " << unEleve.getProf() << endl;

    return 0;
}
```

```
#include <iostream>

using namespace std;

class Professeur;

class Eleve
{
    private:
        string nom;
        Professeur* monProf;

    public:
        Eleve(string name);
        ~Eleve();
        void setProf(Professeur * p);
        string getProf();
        string getNom();

};
```

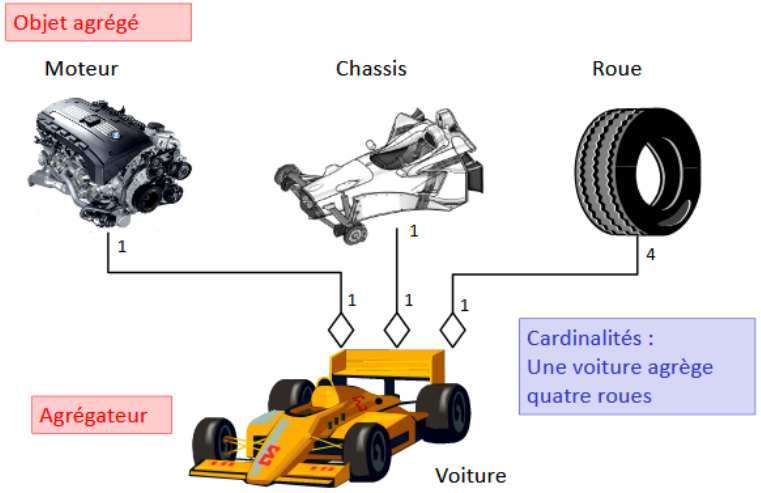
```
#include "eleve.h"
#include "professeur.h"

 Eleve::Eleve(string name):nom(name),monProf(0)
 {

 }

void Eleve::setProf(Professeur * p)
{
    monProf = p; //association eleve->professeur
}

string Eleve::getProf()
{
    if(monProf != 0)
    {
        return monProf->getNom();
    }
}

string Eleve::getNom()
{
    return nom;
}

Eleve::~Eleve()
{
    //dtor
}
```




```
"I:\Utilisateurs\Documents\Cours 2019-2020\SNIR1\TPC++\tpc++\CoursC+
Association standard bidirectionnelle
Tourensol
Tintin
(Tourensol) : Mon eleve est Tintin
(Tintin) : mon professeur est Tourensol

Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.
```

Association de type agrégation

- **Relation ensembliste non symétrique de type « *possède* ».**
  - Une classe joue le rôle d'ensemble et une autre classe le rôle d'élément.
  - Relation à portée essentiellement sémantique.
- **Notation UML**
  - *Un losange vide du côté de l'agrégat.*



- **En C++**

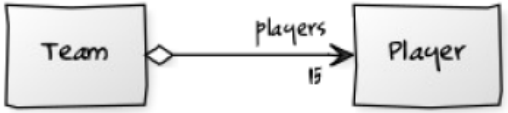
Conséquences visibles sur le code :

- Des méthodes dans l'agrégat pour ajouter ou supprimer des éléments agrégés

On utilise également les pointeurs !!!

Association de type agrégation

- **En C++**



On utilise également les pointeurs !!!

```
#include <iostream>
#include <string>
using namespace std;

class Player
{
    public:
        Player(string name);
        ~Player();
        string getNom();

    private:
        string nom;
};
```

```
#include "player.h"

Player::Player(string name):nom(name)
{
    //ctor
}

Player::~Player()
{
    //dtor
}

string Player::getNom()
{
    return nom;
}
```

```
#include <iostream>
#include <string>

using namespace std;

class Player;
class Team
{
    public:
        Team(string name);
        ~Team();
        string getNom();
        void ajoutPlayer(Player * p);
        bool supprimerPlayer(Player * p);
        void afficheEquipe();

    private:
        string nom;
        Player * players[15];
        int nbrePlayers;
};
```

```
#include "team.h"
#include "player.h"

Team::Team(string name):nom(name),nbrePlayers(0)
{
    //ctor initialisation des case de players à 0
      for(int i=0; i<15;i++)
        {
            players[i] = 0;
        }
}

Team::~Team()
{
    //dtor
}

string Team::getNom()
{
    return nom;
}
```

```
void Team::ajoutPlayer(Player * p)
{
    if(nbrePlayers<15)
    {
        players[nbrePlayers]= p;
        nbrePlayers++;
    }
    else
    {
        cout << "La team est complete !! " << endl;
    }

}
```

Association de type agrégation

- **En C++**


```
bool Team::supprimerPlayer(Player * p)
{
    // Rechercher le player dans le tableau players;
    int pos=-1;
    bool retour = 0;
    for(int i=0; i<nbrePlayers;i++)
    {
        if(players[i] == p)
        {
            pos = i;
        }
    }
    // Supprimer le player
    if(pos >=0)
    {
        players[pos] = 0;  // suppression du player dans le tableau
        // Reorganiser le tableau players sans trou
        for(int i=pos;i<nbrePlayers; i++)
        {
            players[i] = players[i+1];
        }
        nbrePlayers--;
        retour = true;
    }

    return retour;
}


void Team::afficheEquipe()
{
        cout << "Les membres de la Team " << getNom() << " sont :" << endl;
        for(int i=0; i<nbrePlayers;i++)
        {
            cout<< "\t" << players[i]->getNom()<< endl;
        }

}
```

```
#include <iostream>

#include "team.h";
#include "player.h"
using namespace std;

int main()
{
    cout << "Agregation unidirectionnelle Team/Players" << endl;

    Team uneEquipe("PSG");

    Player player1("RantanPlan");
    uneEquipe.ajoutPlayer(&player1);

    Player * player2;

    player2 = new Player("Mickey");
    uneEquipe.ajoutPlayer(player2);

    Player player3("Donald");
    uneEquipe.ajoutPlayer(&player3);

    uneEquipe.afficheEquipe();

    uneEquipe.supprimerPlayer(player2);
    delete player2;
    uneEquipe.afficheEquipe();

    Player player4("ReineDesNeiges");
    uneEquipe.ajoutPlayer(&player4);
    uneEquipe.afficheEquipe();
    return 0;
}
```

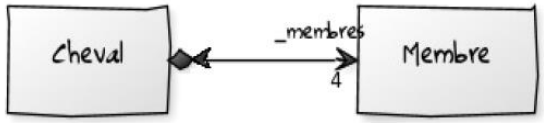
```
"I:\Utilisateurs\Documents\Cours 2019-2020\SNIR1\TPC++\tpc++\Player\bi
Agregation unidirectionnelle Team/Players
Les membres de la Team PSG sont :
        RantanPlan
        Mickey
        Donald
Les membres de la Team PSG sont :
        RantanPlan
        Donald
Les membres de la Team PSG sont :
        RantanPlan
        Donald
        ReineDesNeiges

Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

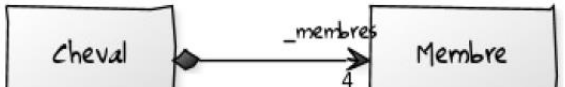
Association de type composition

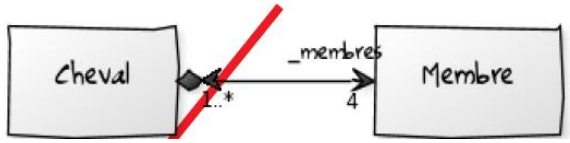
- **Relation de subordination non symétrique de type « *est constitué de* ».**
  - S'interprète comme si la classe composant était une partie intrinsèque et privée de la classe composite.

- **Durée de vie**
  - L'objet composite **a la responsabilité de l'existence et du stockage de l'objet composé**.
  - Conséquence : un objet ne peut être le composant que d'un objet composite.

- **Notation UML**
  - *Un losange plein du côté du composite.*







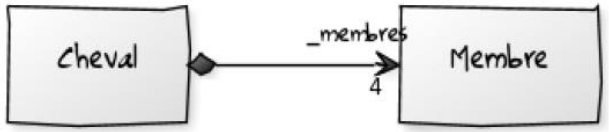
- **En C++**

Conséquences visibles sur le code :

- Le composé **crée et détruit** les instances du composant dans ses méthodes.

Association de type composition

- **En C++**



Un cheval est constitué de 4 membres.

Typiquement, le composé est une instance de classe interne du composite.

Notation C++ : tableau, liste, vector …

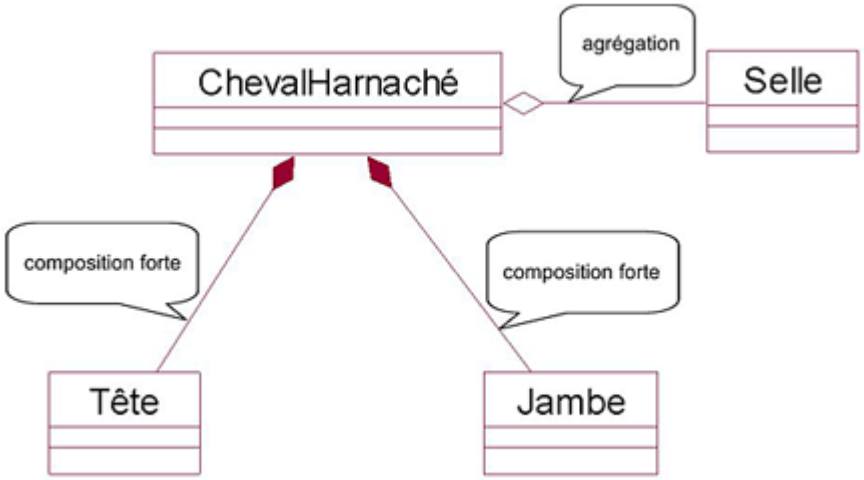
```
class Membre
{
    public:
        Membre(int num);
        ~Membre();

    private:
        int numero;
};
```

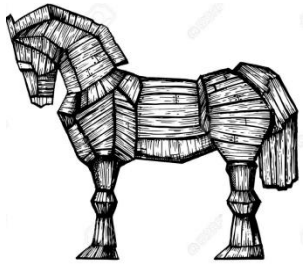
```
#include <iostream>
#include <string>
using namespace std;

class Membre;

class Cheval
{
    public:
        Cheval();
        ~Cheval();

    private:
    Membre * membres[4];
};
```

```
#include "cheval.h"

#include "membre.h"

Cheval::Cheval()
{
    //ctor
    // creation des 4 membres
    for(int i = 0; i<4; i++)
    {
      membres[i] = new membre(i+1);
    }

}

Cheval::~Cheval()
{
    //dtor
    // suppression des 4 membres
     for(int i = 0; i<4; i++)
    {
      delete membres[i];
    }
}
```

Association de type composition

- **Autre exemple :**





```
class Jambe;
class Tete;
class Selle

class ChevalHarnache
{
  public:
      ChevalHarnache();  //Le constructeur créera de maniere dynamique 4 instances de type Jambe qui seront stockées dans le tableau jambes.
      ~ChevalHarnache(); // le destructeur detruira les 4 instances de type Jambe
      void setUneSelle(Selle *); // methode permettant l association avec un objet de type Selle

  private :

      Tete uneTete;  // objet statique : appel du constructeur par defaut de la classe Tete
      Jambe * jambes[4], // comme dans l exemple precedent par exemple.
      Selle * maselle;
};
```

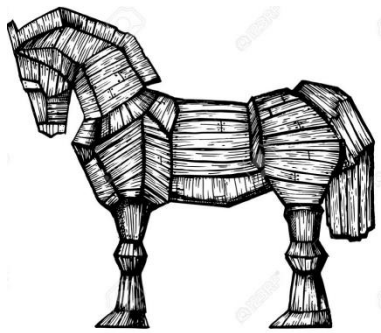
Association de type composition

- **Remarque:**

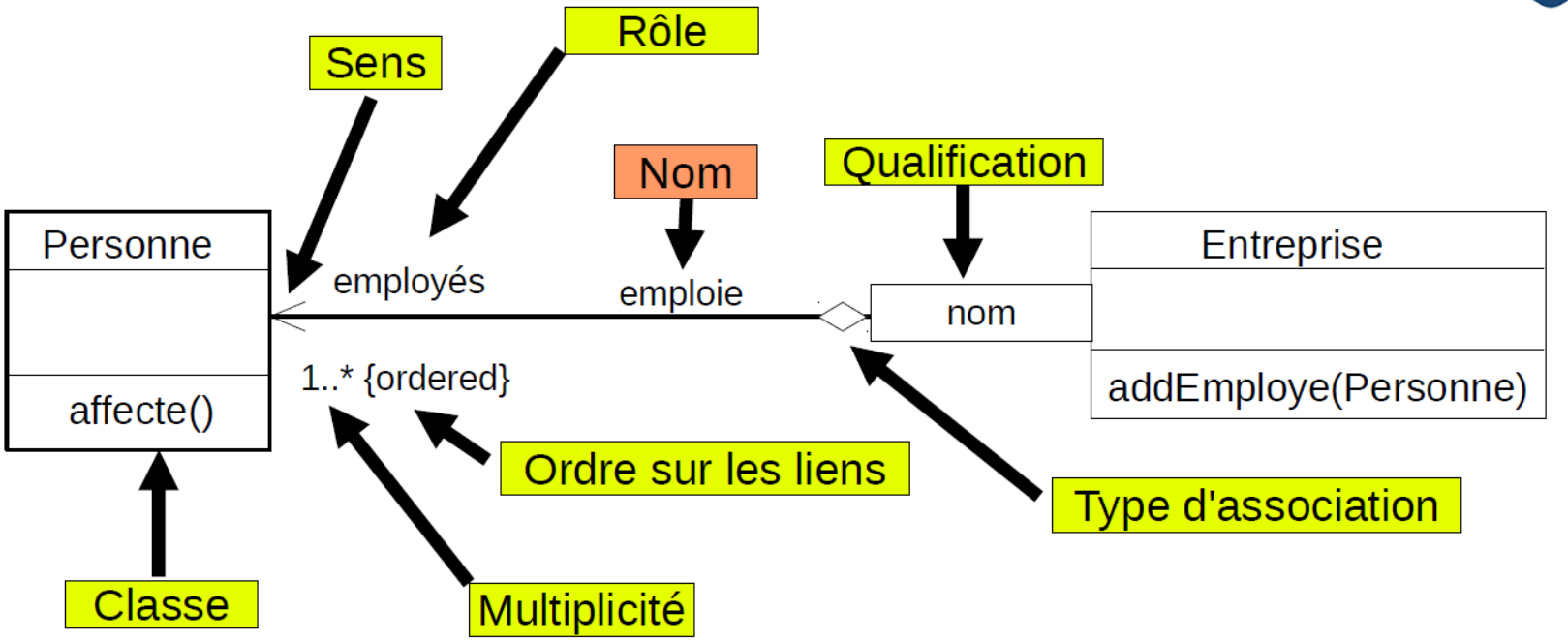
  Si le constructeur d'un composant nécessite un ou plusieurs paramètres.

  Alors le constructeur de la classe composite devra dans son constructeur initialisé les paramètres et les transmettre au constructeur du composant via la liste d'initialisation.

```
class Jambe;
class Tete;
class Selle

class ChevalHarnache
{
  public:
      ChevalHarnache();  //Le constructeur créera de maniere dynamique 4 instances de type Jambe qui seront stockées dans le tableau jambes.
      ~ChevalHarnache(); // le destructeur detruira les 4 instances de type Jambe
      void setUneSelle(Selle *); // methode permettant l association avec un objet de type Selle

  private :

      Tete uneTete;  // objet statique : appel du constructeur par defaut de la classe Tete
      Jambe * jambes[4], // comme dans l exemple precedent par exemple.
      Selle * maselle;
};


ChevalHarnache::ChevalHarnache(int paramTete1, int paramTete2):uneTete(paramTete1,paramTete2)
{
    // sous entendu que le constructeur de Tete necessite 2 paramètres

}
```



Résumé de la notation



- **Importance des décorations d'une association :**

  Forte implication forte dans le code : construction, destruction et choix de la structure de données pour représenter l'association.

Dépendance

- **Relation lâche entre objets qui n'est pas de nature structurelle.**
  - Marquer un lien dynamique.
    p. ex. paramètre ou valeur de retour ou variable locale d'une classe.
- **Notation UML**
  - *Une flèche pointillée.*

- **Remarque**
  - À noter sur le diagramme que si cela est réellement nécessaire
- **En C++**

```
Cartegrise * voiture::obtenirCG()
{
    CarteGrise * cg;
    cg = new CarteGrise();
    cg.setImmatriculation("AB-125-ZZ");
    return cg;
}
```