UML - Le diagramme de classes

Emmanuel Ravrat

Version v0.1.2, 2023-02-24 08:50:04

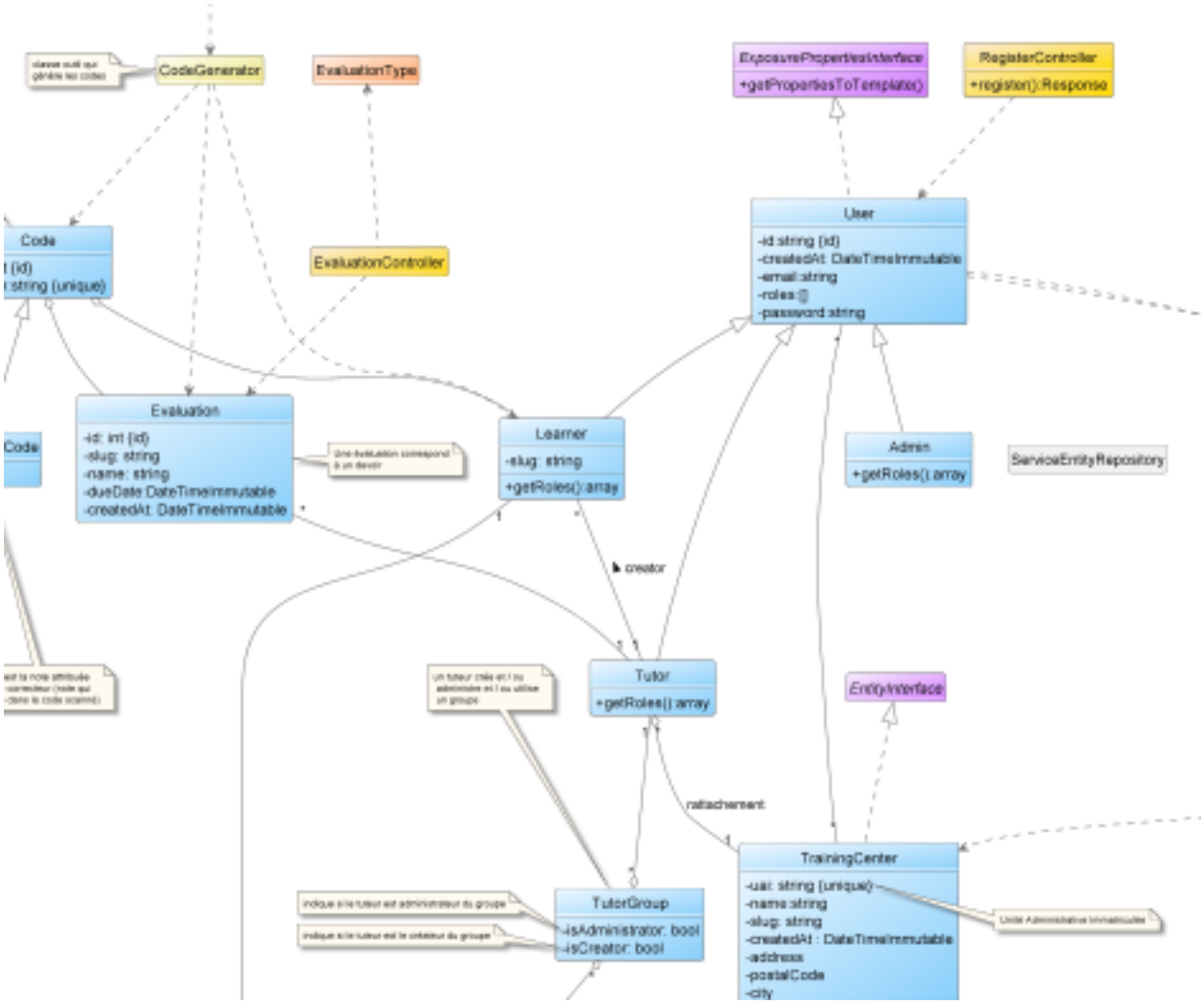


Table des matières

1. A lire avant de commencer

Ce support contient de nombreux extraits de code PHP.

Vous allez voir régulièrement des renvois numérotés dans le code. Ces renvois ne font pas partie du langage. Il ne faut donc pas les copier.

Voici l'illustration d'un renvoi :

```
1 class UneClassePhp {
2
3     public function uneMethode():void{ ①
4         //du code qui fait de la magie ②
5     }
```

① Je suis du texte qui est lié au numéro de renvoi. Je ne fais pas partie de la syntaxe du langage PHP.

② Je suis un autre renvoi.

Ces renvois permettent de cibler des lignes afin d'apporter des explications spécifiques. Il est important de les lire.

Le langage utilisé dans ce support est le langage PHP dans sa version 8.1.

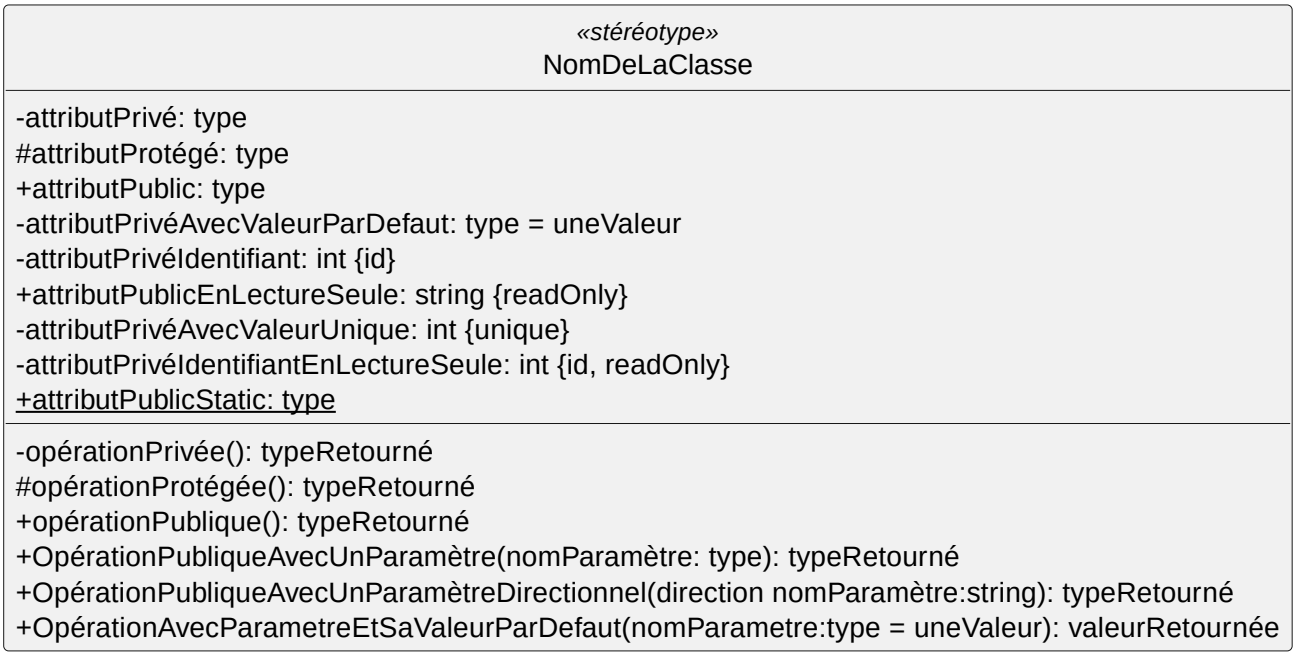
Bonne lecture !

2. Représentation d'une classe avec un diagramme de classes UML

2.1. Représentation générique

La **représentation d'une classe** est formalisée par un rectangle découpé en trois parties du haut vers la bas :

1. le nom de la classe
2. les attributs de la classe (également appelés membres ou propriétés)
3. les opérations de la classe (également appelées membres ou méthodes)



2.2. Précisions sur la visibilité des membres

- Le signe - désigne une visibilité privée. Le membre n'est accessible que depuis l'intérieur de la classe.
- Le signe # désigne une visibilité protégée. Le membre n'est accessible que depuis la classe et ses héritières.
- Le signe + désigne une visilité publique. Le membre est accessible depuis la classe et en dehors.

2.3. Précisions sur la notion de type

- Le type correspond aux types des attributs et des valeurs retournées par les opérations
- Exemples de type : int, bool, string, float, array, List, etc (tout dépend du langage de

programmation utilisé)

- Le type peut être le nom d'une classe puisqu'une classe revient à définir un type.

2.4. Précisions sur la notion d'opération

Une opération est tout simplement une méthode. Son nom doit être réfléchi afin d'exprimer ce qu'elle fait. Par exemple, une méthode qui calcule l'âge d'une personne pourrait s'appeler « calculerAge » ou encore mieux « obtenirAge ». Une méthode qui vérifie qu'une personne est majeur (donc soit c'est vrai, soit c'est faux) pourrait s'appeler « **est**Majeur ».

Il est fortement recommandé d'utiliser l'anglais pour nommer les membres.

2.5. Précisions sur la notion de direction

Cette notion de direction est pertinente dans des langages compilés tels que `C#` ou encore `Java`. En `PHP` et `javascript`, il n'y a pas la possibilité de spécifier une direction à un paramètre.

- `in` : direction par défaut, la variable passée comme argument restera inchangée dans le programme appelant (même si dans l'opération, sa valeur a été modifiée). Elle est utilisée à l'intérieur de l'opération sans être modifiée. Cela prend le nom de passage par valeur.
- `inout` : une modification de la variable passée comme argument dans l'opération se verra également modifiée dans le programme appelant. (c'est ce qui s'apparente à un passage par référence).

Les objets passés en argument le sont automatiquement par référence. Pour les types qui ne le sont pas par défaut (exemples : le type « int » en C#, une chaîne en PHP) et qui doivent l'être dans la méthode, il faut indiquer « `inout` » dans le diagramme.

En PHP, le signe `&` représente ce type de passage alors qu'en C#, c'est le mot clé `ref`.

- `ref` : idem à inout (les logiciels de modélisation proposent généralement `inout`).
- `out` : précise la variable que l'opération doit retourner afin que le programme appelant puisse l'utiliser. Celui-ci doit avoir prévu sa déclaration. Ce terme sera utilisé dans le chapitre sur les procédures stockées.

Si un objet est utilisé comme argument d'une opération, il est TOUJOURS PASSE PAR REFERENCE.
Cela revient à utiliser par défaut « inout » ou « ref »

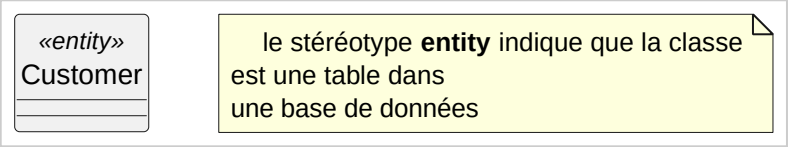
2.6. Précision sur la notion de stéréotype

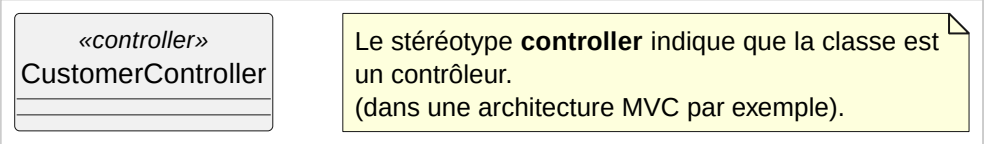
Le **stéréotype** permet d'étendre le vocabulaire de l'UML.

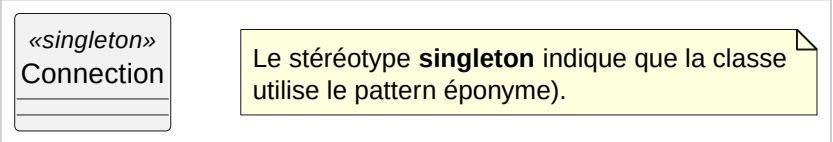
Le **stéréotype d'une classe** permet une meilleure compréhension des éléments qui composent une architecture logicielle. Vous pouvez donc utiliser n'importe quel mot qui permet de faciliter la

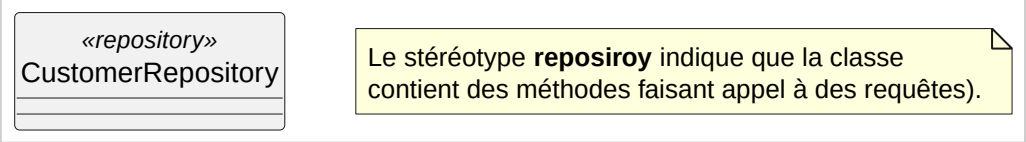
lecture du diagramme.

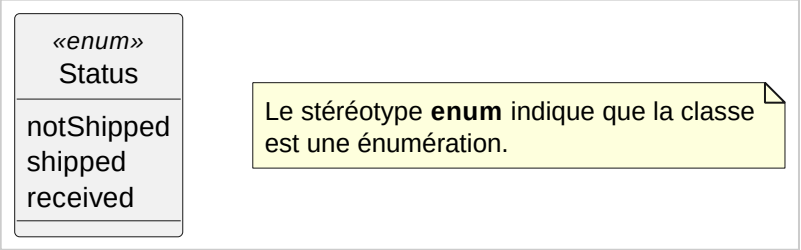
Voici quelques exemples :



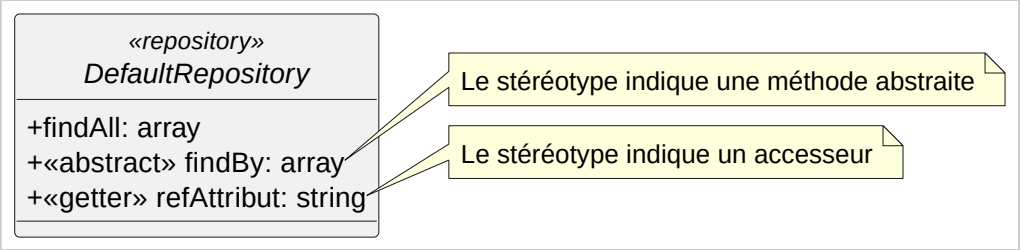








Le **stéréotype d'une opération** (méthode) peut permettre de la classer dans une catégorie de comportement.



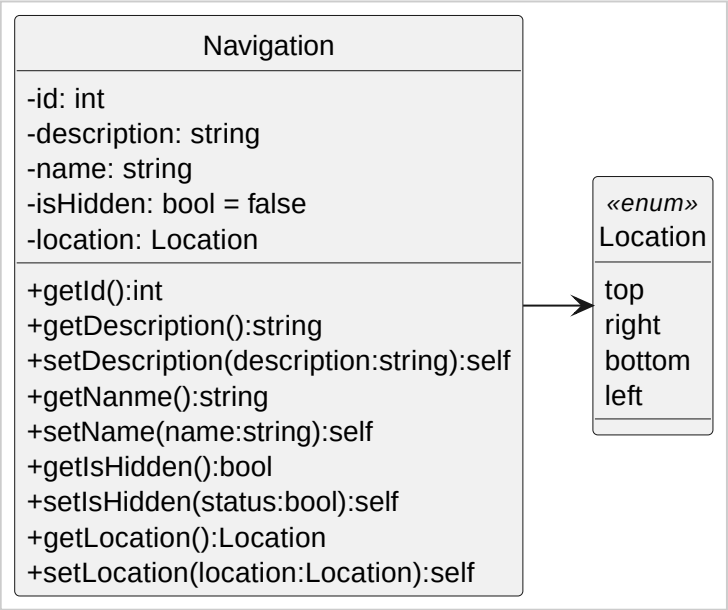
2.7. Avec UML, on affiche que ce qui est essentiel

Un diagramme UML peut rapidement devenir complexe à lire. Certaines classes peuvent avoir de nombreux membres. En fonction du destinataire de l'information, il est possible de ne montrer que l'essentiel.

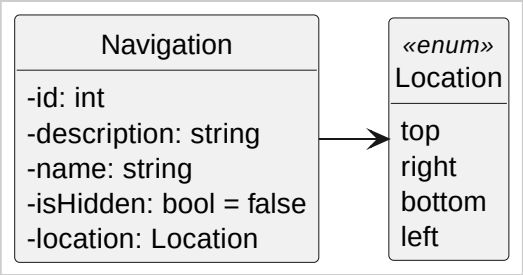
Imaginons une application qui laisse la possibilité à celui qui l'utilise de créer un ou plusieurs

menus de navigation qui pourront être placés à des endroits spécifiques de la fenêtre (en haut, à droite, en bas ou à gauche)

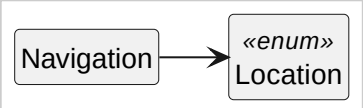
Le diagramme ci-dessous peut être utile au développeur car il sait exactement ce qu'il doit développer :



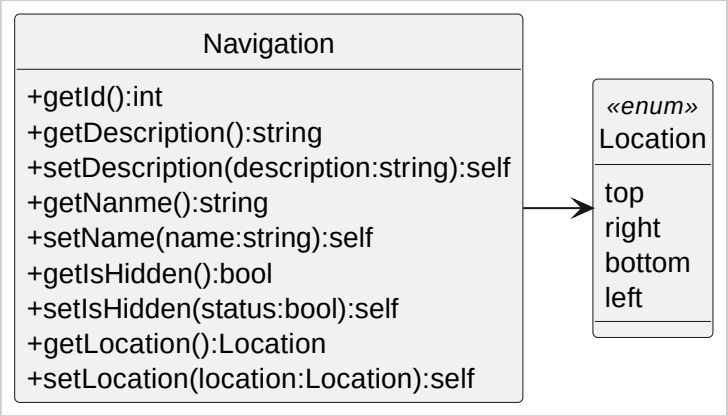
Cependant, il n'y a pas besoin de connaître les méthodes pour créer la table correspondante. Le diagramme ci-après est suffisant :



Lors de la réflexion sur les différentes classes à créer, seule les nom des classes peuvent être affichés



L'utilisateur d'une classe n'a besoin de connaître que ce qu'il peut utiliser (donc les membres publiques) :



Comme vous pouvez le constater, il n'y a pas qu'une seule façon de représenter un même diagramme. Le degré de précision de la conceptualisation dépend de la volonté de transmettre plus ou moins d'informations. Il convient de s'interroger sur le destinataire du diagramme de classes.

3. Quels outils pour réaliser des diagrammes de classes ?

3.1. Quelques outils UML

Il existe de nombreux outils pour réaliser des diagrammes de classe.

- **Dia** (https://dia.fr.softonic.com/) : logiciel gratuit, léger et simple à prendre en main.
- **Argouml** (https://argouml.fr.uptodown.com/windows). Logiciel gratuit assez simple à prendre en main.
- **Staruml** (https://staruml.io/). Logiciel agréable à utiliser et bénéficiant d'une interface assez pratique (mais payant après une période d'essai).
- et de nombreux logiciels payants (windesign,...)
- **PlantUml** (https://plantuml.com/fr/). C'est plus un moteur de rendu qui permet de générer des diagrammes uml à partir de lignes de texte. C'est gratuit.

3.2. PlantUml

PlantUml est l'outil que je préconise car il permet de faire évoluer rapidement les diagrammes. Il permet de générer de nombreux diagrammes (https://plantuml.com/fr/).

Sa particularité tient dans le fait qu'il n'y a aucune interface graphique. Un diagramme est réalisé à partir de lignes de "code" qui décrivent ce que le moteur de PlantUml doit "dessiner".

Comme il s'agit de lignes de "code", il est très facile de versionner ses modélisations.

Q1) Travail à faire

- Chargez la documentation de Plant Uml concernant les diagrammes de classe (https://plantuml.com/fr/class-diagram)
- Chargez la page qui permet de réaliser des diagrammes (http://www.plantuml.com/plantuml/uml/)
- Réalisez directement dans l'éditeur ouvert précédemment le diagramme de classes correspondant à ces besoins :
  - Il faut modéliser deux classes dont les instances vont être persistées en base de données. Il y a deux objets à conceptualiser.
    - Le premier est un employé qui est caractérisé par un numéro unique, un nom, une date de naissance. Il doit être possible de retourner l'âge d'un employé en années entières.
    - Le second objet est une entreprise caractérisée par sa dénomination sociale et les employés qu'elle fait travailler. Une méthode doit retourner le nombre d'employés qu'elle fait travailler.

- La définition des différents attributs doit respecter le principe d'encapsulation.

3.3. Utiliser PlantUml dans son éditeur de code

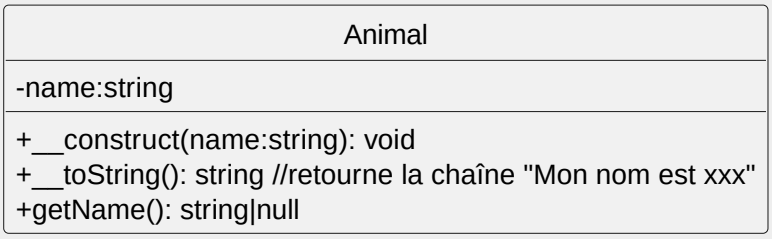
Les IDE tels que PhpStorm et Visual Studio Code sont capables de rendre les diagrammes une fois que le bon plugin est installé.

Q2) Travail à faire

- En fonction de votre éditeur, cherchez et installez le plugin permettant de rendre des diagrammes écrits pour PlantUML.
- Copiez et collez le code du travail précédent de façon à le prévisualiser dans votre éditeur.

Q3) Travail à faire

- A l'aide du langage php, implémentez la classe suivante :



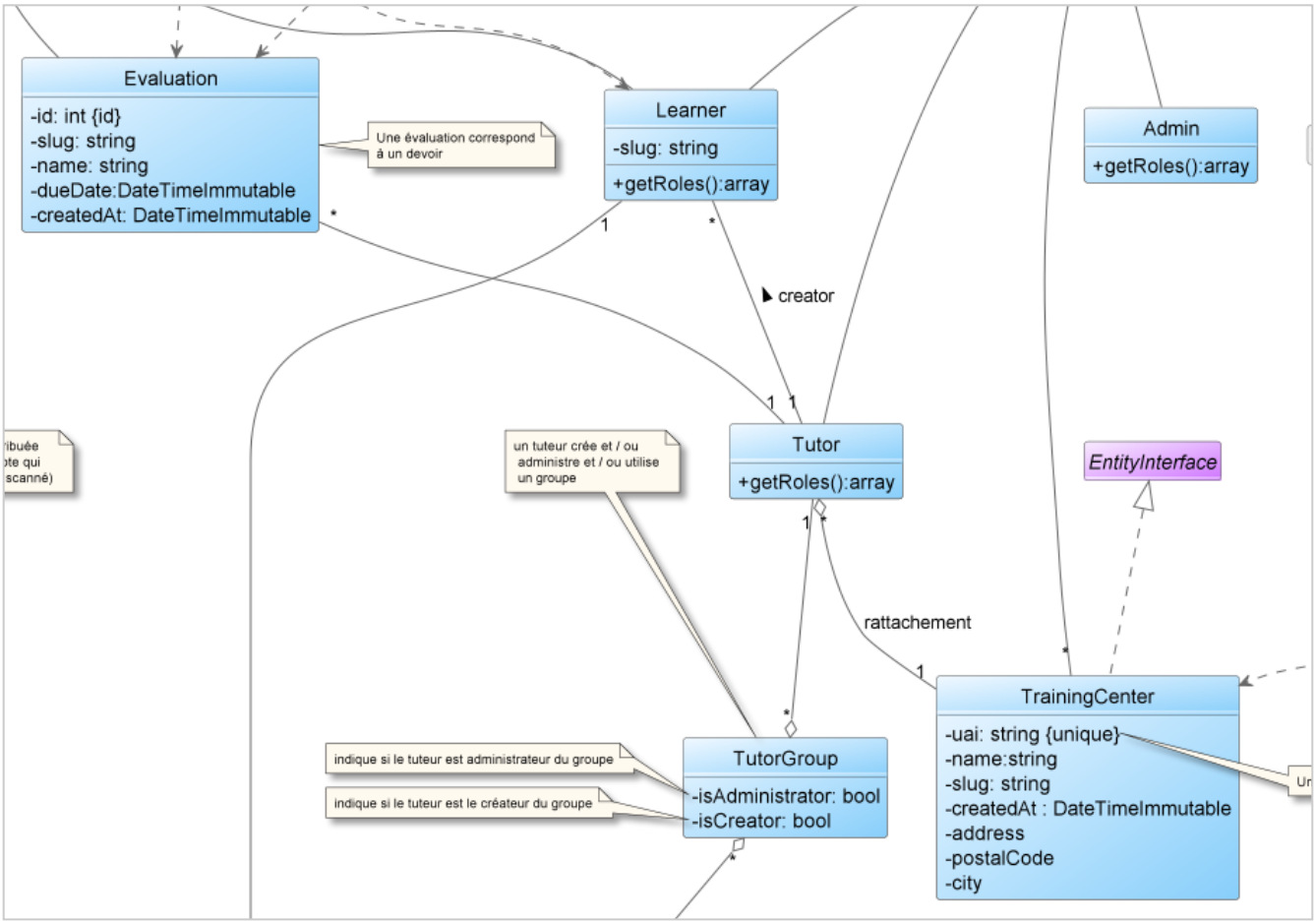
- Une fois la classe implémentée, testez-la de façon à créer deux animaux.
- Afficher le nom du premier objet animal avec un echo directement appliqué sur l'objet afin d'appeler automatiquement la méthode `__toString`
- Afficher le nom du second objet animal en utilisant la méthode `getName()`.

4. Faire des liens entre les classes

Le diagramme de classes **permet de mettre en évidence les liens entre les différentes classes** composant une application.

La **logique sémantique d'architecture de l'application** peut ainsi être lue. C'est-à-dire que l'on peut comprendre le rôle de chaque classe dans l'application et le ou les liens qu'elles ont entre elles. C'est également très utile pour le développeur car il sait exactement ce qu'il doit "coder".

Voici un extrait d'un diagramme de classes :



Les classes sont reliées entre elles avec parfois des traits continus, des traits pointillés, des flèches, etc. Chaque lien a une signification particulière. La suite de ce cours va vous permettre de déterminer quel lien utiliser pour relier des classes entre elles et comment implémenter ces liaisons.

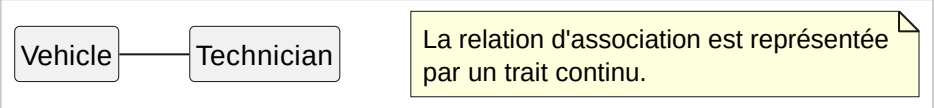
5. Le lien associatif : l'association

L'**association** désigne un lien entre deux objets A et B sachant que A **contient** une ou des instances de B et/ou que B **contient** une ou des instances de A.

Une association exprime une **relation de contenance**. A prévoit un **attribut qui stocke** une ou plusieurs instances de B et / ou vice-versa.

Ce lien est représenté par un trait continu entre les classes dont les objets sont liés.

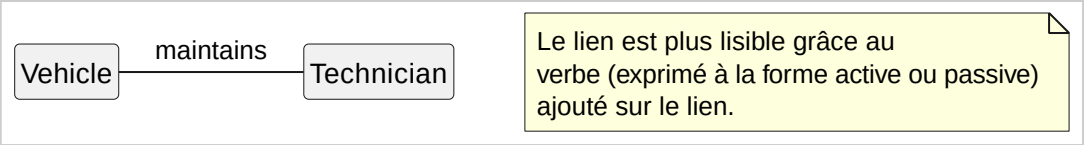
Modélisons le fait qu'un véhicule est entretenu par un technicien :



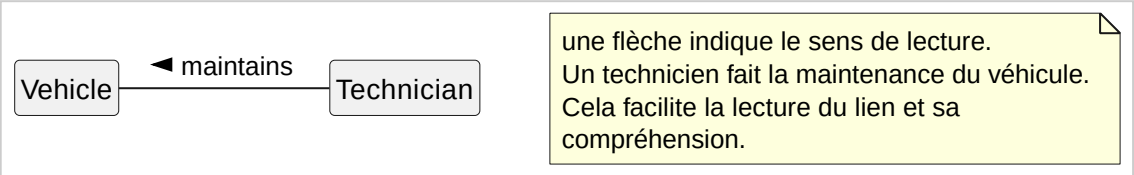
Grâce au lien, nous savons que `Vehicle` est entretenu par `Technician`. S'il n'y avait aucun lien, cela signifierait que `Vehicle` n'utilise pas `Technician` et vice-versa. Ces deux classes n'auraient alors aucune interaction l'une avec l'autre, comme deux personnes qui vivraient à 1000km l'une de l'autre sans même connaître l'existence de l'autre.

Les représentations suivantes sont possibles :

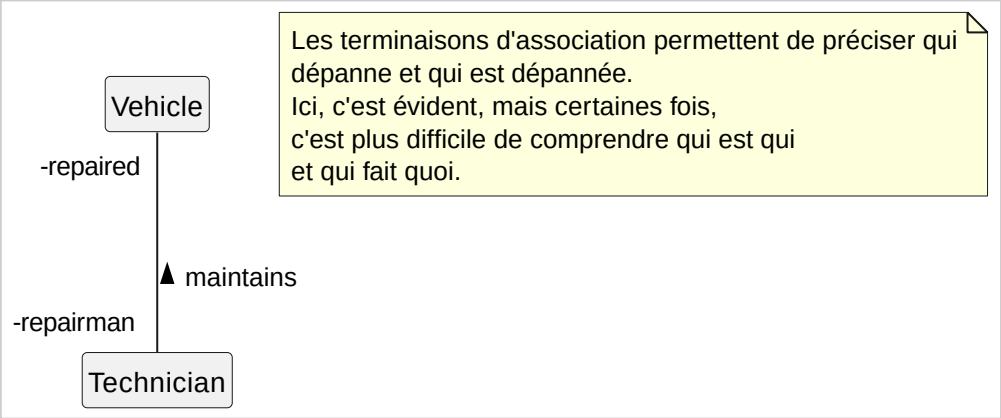
Réprésentation avec un lien et du "texte"



Représentation qui précise le sens de lecture de la relation :

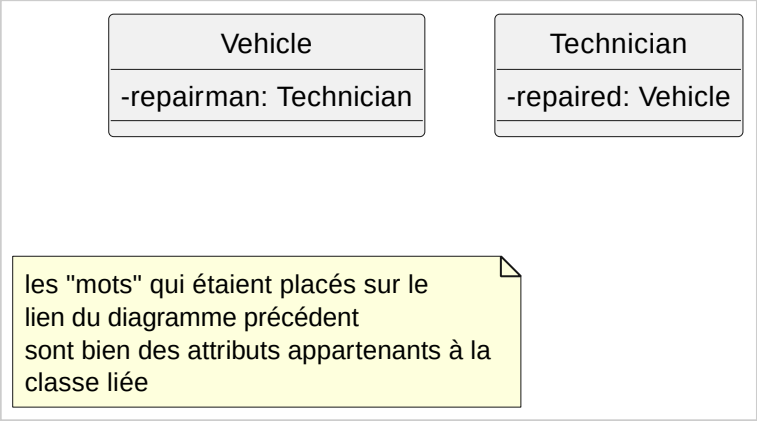


Représentation avec des **terminaisons d'association** :

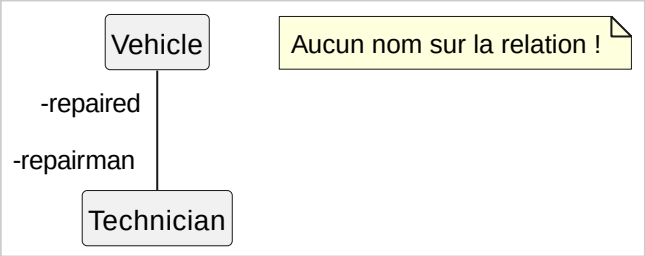


Une **terminaison d'association** est un attribut de la classe liée. Pour mieux comprendre, la

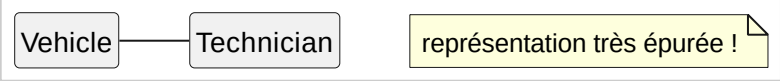
représentation précédente peut être modélisée ainsi :



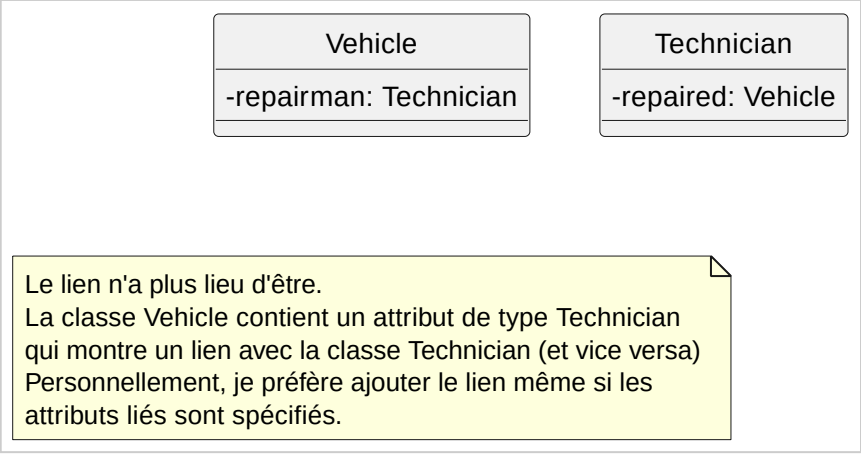
Il n'est pas obligatoire de nommer la relation :



Nous pouvons tout à fait ne laisser que les classes :



L'association peut être représentée sans le lien mais en précisant les attributs pertinents dans chacune des classes liées :

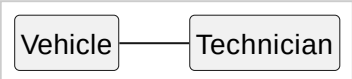


Ce qu'il faut bien comprendre, c'est qu'une association conduit l'entité liée à **contenir** une ou plusieurs instances de l'entité liée. Nous reviendrons sur ce point à plusieurs reprises dans la suite du cours.

6. Les cardinalités d'une association

Des **cardinalités** peuvent être ajoutées afin d'exprimer une **multiplicité du lien associatif entre deux classes**. **C'est utile lorsque l'on souhaite indiquer qu'une instance de classe peut être liée (sémantiquement) à plusieurs instances d'une autre classe.**

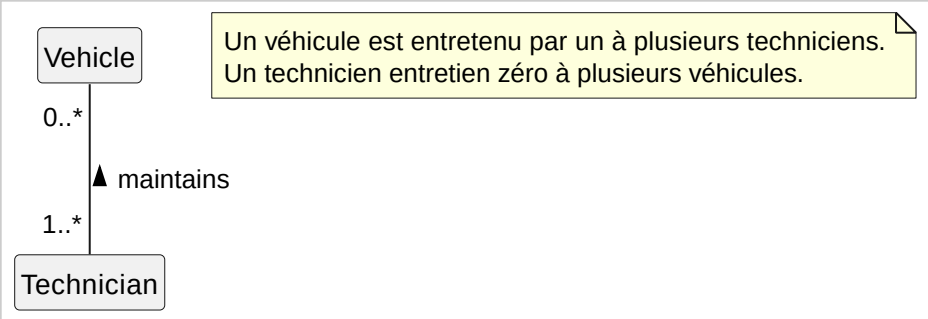
Dans le cas ci-après, il est impossible de savoir si plusieurs techniciens entretiennent un même véhicule ou si un véhicule est entretenu par plusieurs techniciens.



Les règles de gestion à exprimer sur le diagramme sont les suivantes :

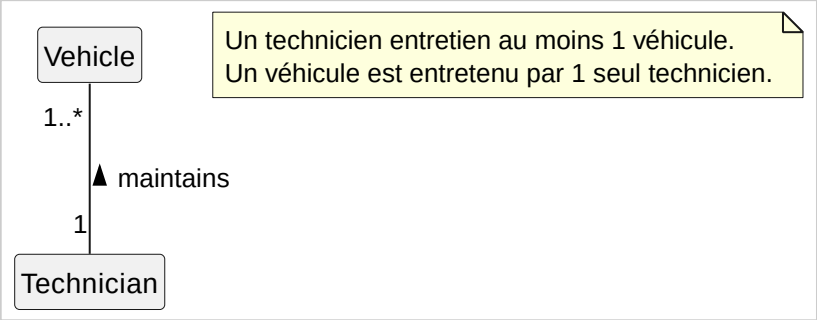
- Un technicien entretien zéro, un ou plusieurs véhicules.
- Un véhicule est entretenu par au moins un technicien

Voici notre diagramme à jour de ces dernières informations :



Afin de bien comprendre le sens de lecture, voici de nouvelles règles de gestion :

- Un technicien entretien au moins 1 véhicule
- Un véhicule est entretenu par 1 seul technicien



Voici quelques exemples de cardinalités :

Exemple de cardinalité	Interprétation
1	Un et un seul. On n'utilise pas la notation 1..1.
1..*	Un à plusieurs
1..5	1 à 5 (maximum)

Exemple de cardinalité	Interprétation
1-5	1 à 5 (maximum)
3..7	3 à 7 (maximum)
3-7	3 à 7 (maximum)
0..1	0 ou 1 seul
1,5	1 **ou** 5
1,5,7	1 **ou** 5 **ou** 7
0..*	0, 1 ou plusieurs
*	0, 1 ou plusieurs

Si vous avez l'habitude de faire de l'analyse selon la méthode Merise, vous aurez remarqué que les cardinalités sont inversées par rapport à celles d'UML.

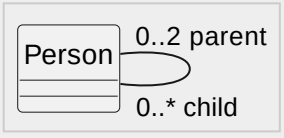
La cardinalité est très utile au développeur pour savoir s'il doit contrôler le nombre d'objets B qu'il est possible d'associer à un objet A.

Q4) Pour chaque diagramme, exprimez la relation en prenant en compte les cardinalités.

a.)

b.)

c.)



Lorsqu'une association exprime un lien vers un maximum de 0 ou une instance de l'objet lié, on parle d'**association simple**.

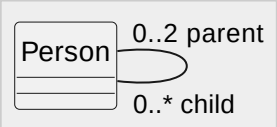
Lorsqu'une association exprime un lien vers un maximum de plusieurs instances de l'objet lié, on parle d'**association multiple**.

Q5) Pour chaque diagramme, indiquer s'il s'agit d'une association simple ou d'une association multiple en fonction du sens de lecture.

a.)

b.)

c.)



Q6)

Réalisez le diagramme de classes correspondant au domaine de gestion décrit ci-après :

Une entreprise gère des hôtels. Des clients peuvent réserver des chambres dans ces hôtels. Une réservation ne peut porter que sur une seule chambre. Des prestations supplémentaires (petit déjeuner, réveil par l'accueil, encas nocturne) peuvent compléter la mise à disposition d'une chambre. Ces prestations peuvent être prévues lors de la réservation ou ultérieurement. Une chambre est équipée ou non de différentes options (lit simple / double, micro-onde, lit enfant, baignoire de type balnéo, etc)

Les associations doivent être nommées et les cardinalités précisées.

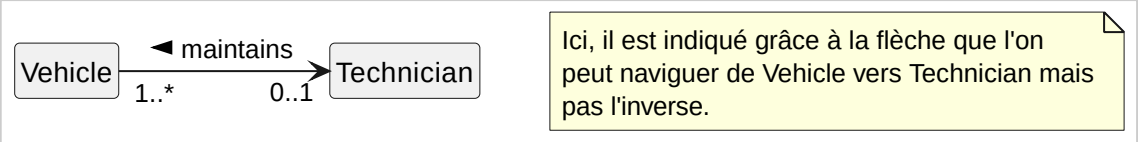
Afin de gagner du temps, les attributs et méthodes ne sont pas attendus.

L'implémentation des cardinalités nécessite de savoir implémenter la navigabilité. Nous reviendrons alors sur ce sujet dans la partie sur l'implémentation des cardinalités.

7. La navigabilité d'une association

La **navigabilité** désigne le fait de connaître à partir d'une instance de classe la ou les instances d'une autre classe. Autrement dit, la navigabilité permet de savoir qu'une instance d'une classe A contient une ou des instances de la classe B.

Illustrons ce concept avec ce diagramme :

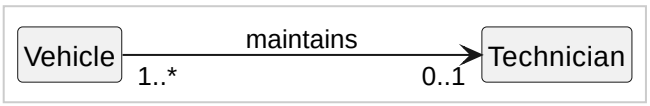


Cette modélisation nous permet d'affirmer qu'une instance de `Vehicle` contient zéro ou une instance de `Technician`. La classe `Vehicle` doit prévoir un attribut capable de contenir une instance de `Technician`. Nous retrouvons la relation de contenance abordée lors de la découverte de la notion d'association.

Lorsqu'un objet de type `Vehicle` a un attribut qui peut contenir un objet de type `Technician`, on dit que l'on peut **naviguer** de `Vehicle` vers `Technician`.

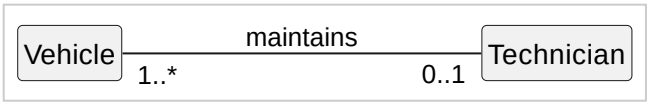
Le **sens de la navigabilité** doit être explicitement précisé sur l'association. Effectivement, la navigabilité peut être exprimée :

- **dans un seul sens** : de `A` vers `B` OU de `B` vers `A`. Dans ce cas, on parle de **navigabilité unidirectionnelle**.

  La **représentation de la navigabilité unidirectionnelle** est modélisée par une flèche qui pointe l'objet vers lequel il est possible de naviguer.

- **dans les deux sens** : de `A` vers `B` ET de `B` vers `A`. Dans ce cas, on parle de **navigabilité bidirectionnelle**.

  La **représentation de la navigabilité bidirectionnelle** est modélisée par **l'absence de flèches sur le lien associatif**.

Sachez qu'il est possible de trouver une représentation avec une flèche de chaque côté de l'association. Mais ce formalisme est peu utilisé :

Comprendre la navigabilité est indispensable car elle se traduit par du code à écrire dans la classe depuis laquelle on navigue vers l'objet lié.

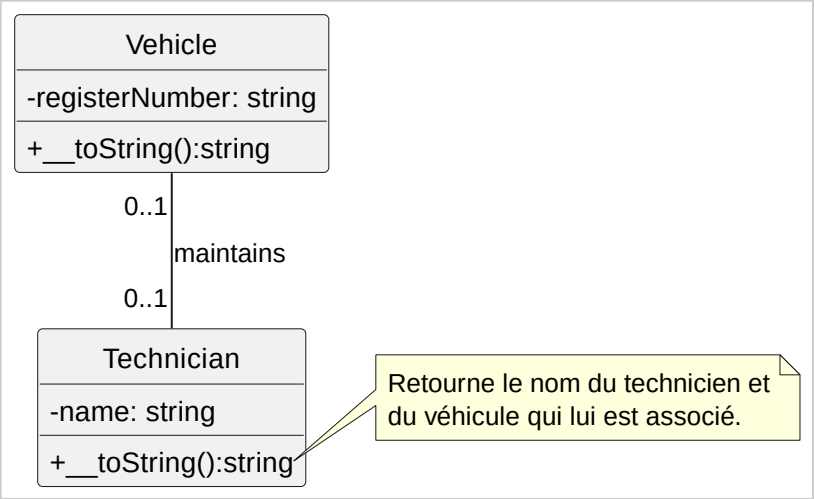
8. Implémentation d'une association unidirectionnelle simple

Rappel 1 : une association entre deux classes `A` et `B` traduit un lien de contenance. Dans ce cas `A` doit prévoir un attribut permettant de stocker une instance de `B` et/ou vice-versa.

Rappel 2 : une association unidirectionnelle n'est navigable que dans un sens (de `A` vers `B` OU de `B` vers `A`).

Rappel 3 : Une association est qualifiée de simple lorsque zéro ou une seule instance de `B` est liée à `A` (ou l'inverse en fonction du sens de navigabilité).

Le diagramme suivant exprime une association unidirectionnelle simple.



Lecture de l'association : Un véhicule est maintenu par 0 ou 1 technicien.

L'association représentée indique clairement au développeur le code qu'il doit écrire. Si deux développeurs doivent implémenter ce diagramme, le code doit être le même !

Nous allons commencer par la classe `Technician` :

```php
<?php
class Technician
{
    private string $name;

    public function __construct(string $name)
    {
        $this->name = $name;
    }

    /**
     * @return string
     */
```

```
14      public function getName(): string
15      {
16          return $this->name;
17      }
18
19      /**
20       * @param string $name
21       *
22       * @return Technician
23       */
24      public function setName(string $name): Technician
25      {
26          $this->name = $name;
27
28          return $this;
29      }
30
31  }
```

Maintenant, implémentons la classe `Vehicle` comme si elle n'était pas liée à `Technician` :

```
1  <?php
2
3  class Vehicle
4  {
5      private string $registerNumber;
6
7
8      public function __construct(
9          string $registerNumber,
10     )
11     {
12         $this->registerNumber = $registerNumber;
13     }
14
15     /**
16      * @return string
17      */
18     public function getRegisterNumber(): string
19     {
20         return $this->registerNumber;
21     }
22
23     /**
24      * @param string $registerNumber
25      *
26      * @return Vehicle
27      */
28     public function setRegisterNumber(string $registerNumber): Vehicle
29     {
```

```
30          $this->registerNumber = $registerNumber;
31
32          return $this;
33      }
34
35
36  }
```

Nous avons nos deux classes mais le lien associatif n'apparaît pas dans le code. C'est maintenant qu'il faut regarder le sens de navigabilité. Il faut exprimer le lien depuis l'objet qui peut naviguer vers l'objet lié soit ici la classe `Vehicle`. Puisqu'une association traduit un lien de contenance, la classe `Vechicle` doit prévoir un attribut qui va contenir zéro ou une instance de `Technician` :

Concrètement, il faut ajouter dans la classe `Vehicle` un attribut `$technician` qui va **contenir** zéro ou une instance de type `Technician`

```
1 //à ajouter dans la classe Vehicle
2 <?php
3
4     //attribut qui va permettre de stocker une instance de Technician
5     private ?Technician $technician = null;
```

Comme l'attribut `technician` est privé, il faut l'encapsuler dans un mutateur et un accesseur (et éventuellement ajouter la possibilité de le passer dans le constructeur si on le désire):

```
 1 //à ajouter dans la classe Vehicle
 2 <?php
 3
 4
 5     /**
 6      * @return Technician|null
 7      */
 8     public function getTechnician(): ?Technician
 9     {
10         return $this->technician;
11     }
12
13     /**
14      * @param Technician|null $technician
15      *
16      * @return Vehicle
17      */
18     public function setTechnician(?Technician $technician): Vehicle
19     {
20         $this->technician = $technician;
21
22         return $this;
23     }
```

Nous venons de mettre en place la notion de navigabilité !

Nous n'avons pas encore implémenté la méthode `__toString()`. Elle va nous permettre d'illustrer le principe de navigabilité car depuis la classe `Vehicle`, nous allons manipuler une instance de `Technician` :

```
1 //à ajouter dans la classe Vehicle
2 <?php
3
4     //cette méthode est une méthode magique qui est automatiquement appelée lorsque
  l'objet est utilisé comme s'il s'agissait d'une chaîne au lieu d'un élément
  complexe
5     public function __toString(): string
6     {
7         $string = "Je suis le véhicule immatriculé {$this->registerNumber}.";
8
9         if ($this->technician === null) {
10            $string .= " Je n'ai pas de technicien.";
11        } else {
12            $string .= " Mon technicien est {$this->technician->getName()}."; ①
13        }
14
15        return $string;
16    }
```

① Le technicien est manipulé à l'intérieur du véhicule courant.

En affichant le véhicule, on obtient bien le nom de son technicien :

```
1 //à ajouter dans la classe Vehicle
2 <?php
3
4 $vehicleAAAA = new Vehicle('AAAA');
5 $paul = new Technician('Paul');
6 $vehicleAAAA->setTechnician($paul);
7 //on affiche l'objet comme si c'était une simple chaîne de caractères (ce n'est
  possible que parce que l'objet prévoit une méthode __toString()
8 echo $vehicleAAAA;
```

Résultat :

```
Je suis le véhicule immatriculé AAAA. Mon technicien est Paul.
```

Cette navigabilité peut être démontrée en récupérant le technicien depuis le véhicule :

```
1 <?php
2
```

```php
$vehicleBBBB = new Vehicle('BBBB');
$sofien = new Technician('Sofien');

$vehicleBBBB->setTechnician($sofien);

//récupération du technicien depuis le véhicule
$technicianOfBBBB = $vehicleBBBB->getTechnician(); ①

echo "{$technicianOfBBBB->getName()} est le technicien du véhicule {$vehicleBBBB->getRegisterNumber()}.";
```

① Depuis une instance de `Vehicle` on navigue vers l'instance de `Technician` associée. C'est le concept de navigabilité.

Résultat :

```
Sofien est le technicien du véhicule BBBB.
```

Depuis PHP 8, il est possible de promouvoir les arguments du constructeur d'une classe comme étant des propriétés d'objets. Cela s'appelle la **promotion de propriété de constructeur** (voir la documentation)

C'est-à-dire qu'un argument de constructeur qui est déclaré avec une visibilité devient automatiquement un attribut d'objet.

La valeur passée au constructeur à l'instanciation de l'objet sera la valeur par défaut de la propriété promue.

Voici la classe `Technician` avec l'utilisation de la promotion des propriétés de son constructeur :

```php
<?php

class Technician
{
    public function __construct(
        private string $name, ①
    )
    {
        //il n'y a plus besoin d'écrire $this->name = $name
    }

    /**
     * @return string
     */
    public function getName(): string
    {
        return $this->name;
    }
```

```
19
20      /**
21       * @param string $name
22       *
23       * @return Technician
24       */
25      public function setName(string $name): Technician
26      {
27          $this->name = $name;
28
29          return $this;
30      }
31
32  }
```

① Le paramètre `$name` est déclaré avec la visibilité `private`. `$name` devient alors automatiquement une propriété d'objet. Lorsqu'un technicien sera instancié et qu'une chaîne sera passée en argument, la propriété d'objet `name` sera initialisée avec cette valeur.

Voici maintenant la classe `Vehicle` réécrite avec cette technique :

```
1  <?php
2
3
4  class Vehicle
5  {
6
7      public function __construct(
8          private string $registerNumber, ①
9          private ?Technician $technician = null,  ①
10     )
11     {
12         //il n'est plus nécessaire d'écrire l'affectation de
   l'immatriculation et du technicien
13         // $this->registerNumber = $registerNumber;
14         // $this->technician = $technician;
15     }
16
17     /**
18      * @return string
19      */
20     public function getRegisterNumber(): string
21     {
22         return $this->registerNumber;
23     }
24
25     /**
26      * @param string $registerNumber
27      *
```

```
28      * @return Vehicle
29      */
30     public function setRegisterNumber(string $registerNumber):
   Vehicle
31     {
32         $this->registerNumber = $registerNumber;
33
34         return $this;
35     }
36
37     /**
38      * @return Technician|null
39      */
40     public function getTechnician(): ?Technician
41     {
42         return $this->technician;
43     }
44
45     /**
46      * @param Technician|null $technician
47      *
48      * @return Vehicle
49      */
50     public function setTechnician(?Technician $technician): Vehicle
51     {
52         $this->technician = $technician;
53
54         return $this;
55     }
56
57     //cette méthode est une méthode magique qui est automatiquement
   appelée lorsque l'objet est utilisé comme s'il s'agissait d'une
   chaîne au lieu d'un élément complexe
58     public function __toString(): string
59     {
60         $string = "Je suis le véhicule immatriculé {$this-
   >registerNumber}.";
61
62         if ($this->technician === null) {
63             $string .= " Je n'ai pas de technicien.";
64         } else {
65             $string .= " Mon technicien est {$this->technician-
   >getName()}."; ①
66         }
67
68         return $string;
69     }
70 }
```

① Les deux arguments du constructeur sont déclarés avec une visibilité. Ils sont automatiquement promus au rang de propriété d'objet.

> L'utilisation des deux classes reste exactement la même.

Vous savez maintenant implémenter une association undirectionnelle simple.

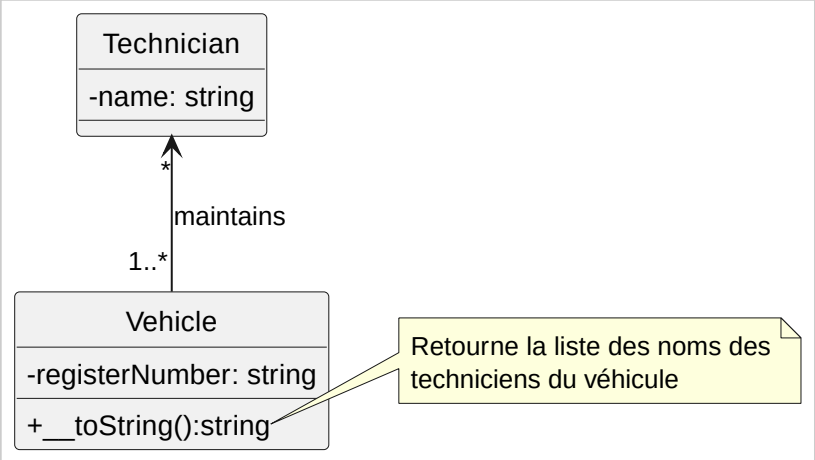
9. Implémentation d'une association unidirectionnelle multiple

Rappel 1 : une association entre deux classes `A` et `B` traduit un lien de contenance. Dans ce cas `A` doit prévoir un attribut permettant de stocker une instance de `B` et/ou vice-versa.

Rappel 2 : une association unidirectionnelle n'est navigable que dans un sens (de `A` vers `B` OU de `B` vers `A`).

Rappel 3 : Une association est qualifiée de multiple lorsque zéro ou une ou plusieurs instances de `B` sont liées à `A` (ou l'inverse en fonction du sens de navigabilité).

Le diagramme suivant exprime une association unidirectionnelle multiple.



Lecture de l'association : Un véhicule est maintenu par 0 à plusieurs techniciens.

L'association représentée indique clairement au développeur le code qu'il doit écrire. Si deux développeurs doivent implémenter ce diagramme, le code doit être le même !

Nous allons commencer par la classe `Technician` :

```php
<?php
class Technician
{
    private string $name;

    public function __construct(string $name)
    {
        $this->name = $name;
    }

    /**
     * @return string
     */
```

```
14      public function getName(): string
15      {
16          return $this->name;
17      }
18
19      /**
20       * @param string $name
21       *
22       * @return Technician
23       */
24      public function setName(string $name): Technician
25      {
26          $this->name = $name;
27
28          return $this;
29      }
30
31  }
```

Maintenant, implémentons la classe `Vehicle` comme si elle n'était pas liée à `Technician` :

```
1  <?php
2
3  class Vehicle
4  {
5
6      public function __construct(
7          private string $registerNumber,
8      )
9      {
10     }
11
12
13     /**
14      * @return string
15      */
16     public function getRegisterNumber(): string
17     {
18         return $this->registerNumber;
19     }
20
21     /**
22      * @param string $registerNumber
23      *
24      * @return Vehicle
25      */
26     public function setRegisterNumber(string $registerNumber): Vehicle
27     {
28         $this->registerNumber = $registerNumber;
29
```

```
30         return $this;
31     }
32
33 }
```

Nous avons nos deux classes mais le lien associatif n'apparaît pas dans le code. C'est maintenant qu'il faut regarder le sens de navigabilité. Il faut exprimer le lien depuis l'objet qui peut naviguer vers l'objet lié soit ici la classe `Vehicle`. Puisqu'une association traduit un lien de contenance, la classe `Vechicle` doit **prévoir un attribut qui va contenir zéro à plusieurs instances** de `Technician` :

Concrètement, il faut ajouter dans la classe `Vehicle` un attribut `$technicians` (au pluriel) qui va **contenir** zéro à plusieurs instances de type `Technician`. Cet attribut est qualifié de **collection**. Une collection regroupe des objets de même type. Ici, il s'agit de stocker une collection d'instances de type `Technician`.

Dans le cas d'une navigabilité vers plusieurs instances liées, l'attribut qui va contenir ces instances doit permettre de stocker une collection.

Ajoutons l'attribut `technicians` dont le pluriel indique bien qu'il s'agit d'une collectionj de techniciens. Par défaut, cet attribut est une collection vide (en PHP, ce sera un tableau vide).

```
1 <?php
2
3     public function __construct(
4         private string $registerNumber,
5         private array $technicians = [],  ①
6     )
7     {
8     }
```

① l'attribut `technicians` au pluriel est un tableau qui va contenir 0 à plusieurs instances de `Technician`. Cet attribut est une collection.

L'attribut qui contient la collection doit toujours être initialisé avant d'être manipulé.

Cet oubli est une erreur courante qu'il faut veiller à ne pas faire !

Il faut prévoir le mutateur et l'accesseur de notre attribut `technicians`. Comme il s'agit d'une collection, les méthodes habituelles `getXXX` et `setXXX` ne conviennent pas.

Quand on manipule une collection, soit on ajoute un élément à la collection, soit on en retire un. Cela signifie qu'il y a deux mutateurs à prévoir :

- un mutateur `addTechnician()` qui comme son nom l'indique doit permettre d'ajouter une instance de `Technician` à la collection.
- un mutateur `removeTechnician` qui comme son nom l'indique doit permettre de retirer une

instance de `Technician` de la collection.

Commençons par la méthode `addTechnician` qui permet d'ajouter un technicien :

```php
//à ajouter à la classe Vehicle
<?php

    /**
     * @param Technician $technician ajoute un item de type Technician à la
     *                               collection
     */
    public function addTechnician(Technician $technician): bool
    {
        if (!in_array($technician, $this->technicians, true)) { ①
            $this->technicians[] = $technician; ②

            return true;
        }

        return false;
    }
```

① On vérifie que le technicien à ajouter à la collection n'y serait pas déjà (par défaut, on considère que l'on ne stocke pas plusieurs fois la même instance dans une collection)

② Le technicien est ajouté à la collection

Ajoutons la possibilité de retirer un technicien de la collection (s'il y figure bien entendu) :

```php
//à ajouter à la classe Vehicle
<?php

    /**
     * @param Technician $technician retire l'item de la collection
     */
    public function removeTechnician(Technician $technician): bool
    {
        $key = array_search($technician, $this->technicians, true); ①

        if ($key !== false) {
            unset($this->technicians[$key]); ②

            return true;
        }

        return false;
    }
```

① On recherche la position du technicien à retirer dans la collection (tableau). Si le technicien n'est pas dans le tableau, la fonction `array_search()` retourne `false`.

② La référence à l'instance de `Technician` stockée à l'index `$key` est effacée.

Maintenant que nous sommes capables de lier des techniciens à un véhicule, nous pouvons implémenter le code de la méthode `__toString()` de façon à ce qu'elle retourne leur nom :

```php
//à ajouter à la classe Vehicle
<?php

    public function __toString(): string
    {
        $string = "Je suis le véhicule immatriculé {$this->registerNumber}.";

        if (count($this->technicians) === 0) {
            $string .= "\nJe ne suis associé à aucun technicien.\n";
        } else {
            $string .= "\nJe suis associé à un ou plusieurs techniciens :";
            foreach ($this->technicians as $technician) {
                $string .= "\n- {$technician->getName()}";
            }
        }

        return $string;
    }
```

Testons cela en affectant 3 techniciens à un véhicule :

```php
<?php

$vehicleAAAA = new Vehicle('AAAA');
$paul = new Technician('Paul');
$sofien = new Technician('Sofien');
$anna = new Technician('Anna');

//affectation de plusieurs techniciens
$vehicleAAAA->addTechnician($paul);
$vehicleAAAA->addTechnician($sofien);
$vehicleAAAA->addTechnician($anna);

echo $vehicleAAAA;
```

Résultat :

```
Je suis le véhicule immatriculé AAAA.
Je suis associé à un ou plusieurs techniciens :
- Paul
- Sofien
- Anna
```

Retirons un technicien :

```
<?php

$vehicleAAAA->removeTechnician($paul);
echo $vehicleAAAA;
```

Résultat :

```
Je suis associé à un ou plusieurs techniciens :
- Sofien
- Anna
```

Si nous avons nos deux mutateurs, nous n'avons pas encore d'accesseur afin d'accéder à la collection de techniciens. Le voici :

```
//à ajouter à la classe Vehicle
<?php

    /**
     * @return Technician[]
     */
    public function getTechnicians(): array
    {
        return $this->technicians;
    }
```

Voici comment utiliser ce mutateur `getTechnians()`:

```
<?php

echo "\nVoici la liste des techniciens du véhicule {$vehicleAAAA->
getRegisterNumber()} :";

foreach ($vehicleAAAA->getTechnicians() as $technician) { ①
    echo "\n* Technicien {$technician->getName()}";
}
```

① La collection fait l'objet d'une itération afin de naviguer vers chaque technicien lié au véhicule.

Parfois, il peut être utile de passer une collection en une fois plutôt que d'ajouter les items un par un. Dans ce cas, une méthode nommée `setTechnicians()` peut être pertinente. C'est en fait un troisième mutateur qui vient compléter `addTechnician()` et `removeTechnician()`.

Voici le code de la méthode `Vehicle::setTechnicians()` :

```php
<?php

    /**
     * Initialise la collection avec la collection passée en argument
     *
     * @param array $technicians collection d'objets de type Technician
     *
     * @return $this
     */
    public function setTechnicians(array $technicians): self
    {
        $this->technicians = $technicians; ①


        return $this;
    }
```

① La collection est entièrement initialisée avec le tableau passé en argument. Si une collection était déjà stockée dans l'attribut `technicians`, elle est écrasée par la nouvelle.

Mettons en oeuvre cette nouvelle méthode :

```php
<?php

$cedric = new Technician('Cédric');
$baptiste = new Technician('Baptiste');

$techniciansCollection = [$cedric, $baptiste];

$vehicleCCCC = new Vehicle('CCCC');

$vehicleCCCC->setTechnicians($techniciansCollection);

echo $vehicleCCCC;
```

Résultat :

```
Je suis associé à un ou plusieurs techniciens :
- Cédric
- Baptiste
```

Notez qu'une collection peut également être directement passée au constructeur de `Vehicle` :

```php
<?php

$cedric = new Technician('Cédric');
$baptiste = new Technician('Baptiste');
```

```
5
6 $techniciansCollection = [$cedric, $baptiste];
7
8 $vehicleCCCC = new Vehicle('CCCC', $techniciansCollection);
```

La méthode `setTechnicians` attend en argument un tableau mais en PHP, il n'est pas possible de "dire" que l'on souhaite un tableau ne contenant que des instances de `Technician`. Ainsi, rien n'empêche d'initialiser un véhicule avec un tableau ne contenant que des entiers :

```
1 <?php
2
3 $arrayInt = [14,84,170];
4
5 $vehicleCCCC = new Vehicle('CCCC', $arrayInt); ①
```

① L'instanciation est réalisée sans problème. Le constructeur attendait un tableau en second argument et c'est bien un tableau qui lui a été passé.

Par contre, ça se gâte si on cherche à afficher le véhicule :

```
1 <?php
2
3 $arrayInt = [14,84,170];
4
5 $vehicleCCCC = new Vehicle('CCCC', $arrayInt); ①
6
7 echo $vehicleCCCC; ①
```

① Le fait d'afficher le véhicule va appeler la méthode `__toString()` qui va tenter d'itérer sur la collection de techniciens qui n'est autre qu'un tableau de chaînes de caractères.

Résultat :

```
Fatal error: Uncaught Error: Call to a member function getName() on int in ...
```

Le message est explicite. Un entier n'a pas de méthode `getName()`. C'est une simple **valeur scalaire** (des chiffres ou une chaîne de caractères.)

Pour éviter ce problème, il faut contrôler chaque item de la collection. C'est en fait très simple à faire. Il suffit de boucler sur la collection passée en argument et d'ajouter à la collection de technicien chacun des items parcourus :

```
1 <?php
2
3     /**
4      * Initialise la collection avec la collection passée en argument
5      *
```

```
6       * @param array $technicians collection d'objets de type Technician
7       *
8       * @return $this
9       */
10      public function setTechnicians(array $technicians): self
11      {
12
13          foreach($technicians as $technician){ ①
14              $this->addTechnician($technician); ②
15          }
16
17          return $this;
18      }
```

① La tableau passé en argument est parcouru afin d'accéder à chacun de ses items

② Chaque item du tableau est passé à la méthode `addTechnician`. Cette méthode attend une instance de `Technician`. Si ce n'est pas le cas, un erreur fatale sera générée. Il n'est alors plus possible d'avoir une collection qui ne contiendrait pas que des techniciens.

Nous allons vérifier cela en affectant un tableau d'entiers en guise de collection de techniciens :

```
1 <?php
2
3 $arrayInt = [14,84,170];
4
5 $vehicleDDDD = new Vehicle('DDDD');
6
7 $vehicleDDDD->setTechnicians($arrayInt); ①
```

① Une erreur doit indiquer qu'une instance de `Technicien` est attendue.

Résultat :

```
Fatal error: Uncaught TypeError: Vehicle::addTechnician(): Argument #1 ($technician)
must be of type Technician, int given
```

Nous avons contrôlé qu'une collection d'objets de type `Technician` étaient passée en argument de `setTechnicians`. Cependant, il est toujours possible de passer un tableau d'entiers à l'instanciation d'un véhicule ! Heureusement, nous pouvons faire appel au travail que l'on vient de faire en appelant la méthode `setTechnicians()` depuis le constructeur :

```
1 <?php
2
3     public function __construct(
4         private string $registerNumber,
5     )
6     {
7         $this->setTechnicians($technicians); ①
```

```
8     }
```

① En faisant appel à la méthode `setTechnicians()`, on s'assure de contrôler chaque élément du tableau passé en argument.

Ce qu'il faut retenir

- Une association avec cardinalité multiple nécessite d'utiliser un attribut de type **collection**.
- L'attribut stockant la collection doit être initialisé avec un tableau vide en PHP.
- Un attribut qui stocke une "collection" doit être encapsulé avec 4 méthodes :
  - une méthode `addXXX` qui permet d'ajouter une instance de `XXX` dans la collection
  - une méthode `removeXXX` qui permet de retirer une instance de `XXX` dans la collection
  - une méthode `getXXXs` **(avec `XXX` au pluriel)** qui retourne la collection complète
  - une méthode `setXXXs` **(avec `XXX` au pluriel)** qui initialise la collection en une fois.
- En PHP, il faut contrôler que chaque item ajouté à la collection est bien du type attendu.

10. Implémentation d'une association bidirectionnelle simple

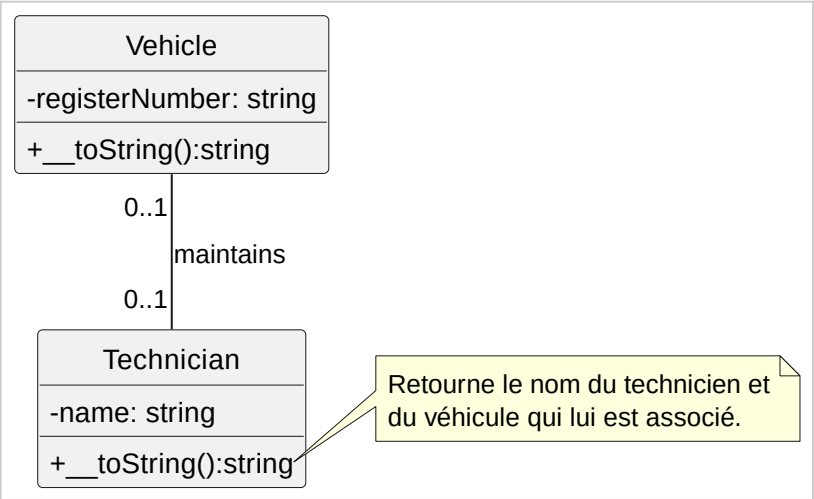
Rappel 1 : une association entre deux classes `A` et `B` traduit un lien de contenance. Dans ce cas `A` doit prévoir un attribut permettant de stocker une instance de `B` et/ou vice-versa.

Rappel 2 : une association bidirectionnelle est navigable dans les deux sens (de `A` vers `B` **ET** de `B` vers `A`).

Rappel 3 : Une association est qualifiée de simple lorsque zéro ou une seule instance de `B` est liée à `A` (ou l'inverse).

10.1. Mise en place de la navigation bidirectionnelle

Le diagramme suivant exprime une association bidirectionnelle simple quel que soit le sens de navigation.



Lecture de l'association : Un véhicule est maintenu par 0 ou 1 technicien et un technicien maintient 0 ou un véhicule.

L'association représentée indique clairement au développeur le code qu'il doit écrire. Si deux développeurs doivent implémenter ce diagramme, le code doit être le même !

Commençons par la classe `Technician` sans nous soucier de la navigabilité :

```php
<?php

class Technician
{
    public function __construct(
        private string $name,
    )
    {
```

```
    }


    /**
     * @return string
     */
    public function getName(): string
    {
        return $this->name;
    }

    /**
     * @param string $name
     *
     * @return Technician
     */
    public function setName(string $name): Technician
    {
        $this->name = $name;

        return $this;
    }


}
```

Faisons de même pour la classe `Vehicle` :

```
<?php


class Vehicle
{
    public function __construct(
        private string $registerNumber,
    )
    {
    }

    /**
     * @return string
     */
    public function getRegisterNumber(): string
    {
        return $this->registerNumber;
    }

    /**
     * @param string $registerNumber
     *
```

```
23      * @return Vehicle
24      */
25     public function setRegisterNumber(string $registerNumber): Vehicle
26     {
27         $this->registerNumber = $registerNumber;
28 
29         return $this;
30     }
31 
32 
33 }
```

Maintenant, nous pouvons nous intéresser à la navigabilité entre les deux classes.

Tout d'abord, il y a une navigabilité simple de `Vehicule` vers `Technician` (cardinalité maximale à 1). Nous savons déjà implémenté cette situation dans la partie sur la navigabilité unidirectionnelle simple. Il suffit donc de refaire la même chose.

Nous ajoutons un attribut `technician` qui va permettre de stocker une instance de `Technician` dans la classe `Vehicle` :

```
1 <?php
2 
3 
4     public function __construct(
5         private string $registerNumber,
6         private ?Technician $technician = null,
7     )
8     {
9     }
```

Puis le mutateur et l'accesseur de l'attribut `technician` :

```
1 //classe Vehicle
2 <?php
3 
4 
5 
6     /**
7      * @return Technician|null
8      */
9     public function getTechnician(): ?Technician
10    {
11        return $this->technician;
12    }
```

Ajoutons la méthode `__toString` qui retourne le technicien associé au véhicule :

```php
//classe Vehicle
<?php


    public function __toString(): string
    {
        $string = "Je suis le véhicule immatriculé {$this->registerNumber}.";

        if ($this->technician === null) {
            $string .= " Je n'ai pas de technicien.";
        } else {
            $string .= " Mon technicien est {$this->technician->getName()}."; ①
        }

        return $string;
    }
```

Nous avons mis en place la navigabilité dans le sens `Vehicle` vers `Technician`. Il nous faut mettre en place la navigabilité de `Technician` vers `Vehicle`. Cela nécessite un attribut `vehicle` qui va stocker l'instance de véhicule associé au technicien :

```php
//classe Technician
<?php

    public function __construct(
        private string $name,
        private ?Vehicle $vehicle = null,
    )
    {
    }
```

Il faut le mutateur et l'accesseur de cet attribut d'objet :

```php
//classe Technician
<?php

    /**
     * @return Vehicle|null
     */
    public function getVehicle(): ?Vehicle
    {
        return $this->vehicle;
    }

    /**
     * @param Vehicle|null $vehicle
     *
     * @return Technician
```

```
16      */
17     public function setVehicle(?Vehicle $vehicle): Technician
18     {
19         $this->vehicle = $vehicle;
20 
21         return $this;
22     }
```

Implémentons la méthode `__toString` de la classe `Technician` :

```
1 //classe Technician
2 <?php
3 
4     public function __toString(): string
5     {
6         $string = "Je suis le technicien nommé {$this->name}.";
7 
8         if ($this->vehicle === null) {
9             $string .= " Je n'ai pas de voiture en charge.";
10        } else {
11            $string .= " La voiture dont j'ai la charge a pour immatriculation
  {$this->vehicle->getRegisterNumber()}."; ①
12        }
13 
14        return $string;
15    }
```

Comme vous pouvez le remarquer, l'aspect "bidirectionnel" ne change rien à l'implémentation (pour l'instant). Nous avons géré la navigabilité dans les deux sens de lecture de l'association.

10.2. La problématique de l'association bidirectionnelle

Observons la mise en oeuvre de cette navigabilité bidirectionnelle.

Nous commençons par associer un technicien à un véhicule :

```
1 //classe Technician
2 <?php
3 
4 
5 $vehicleAAAA = new Vehicle('AAAA');
6 $paul = new Technician('Paul');
7 //association d'un véhicule à son technicien
8 $vehicleAAAA->setTechnician($paul);
```

Puis en affichant le véhicule, on mobilise la navigabilité vers l'instance de `Technician` liée :

```
1 //classe Technician
2 <?php
3
4
5 //le véhicule AAAA connait son technicien :
6 echo $vehicleAAAA;
```

Résultat :

```
Je suis le véhicule immatriculé AAAA. Mon technicien est Paul.
```

La navigation est bien fonctionnelle. Depuis le véhicule, nous avons accès à son technicien.

Puisque nous sommes dans une navigation bidirectionnelle, la navigation doit être possible dans l'autre sens (du technicien vers son véhicule) :

```
1 //classe Technician
2 <?php
3
4
5 var_dump($paul->getVehicle()); ①
```

① Nous nous attendons logiquement à voir une instance de `Vehicle` puisque nous sommes censés pouvoir naviguer d'un technicien vers son véhicle.

Résultat :

```
NULL
```

Nous venons de mettre en avant la **problématique de la mise à jour d'une association bidirectionnelle**.

La navigation d'un objet vers l'autre n'est pas automatiquement réciproque. Ce n'est pas parce que vous liez un objet `B` à un objet `A` que `A` sera lié à `B`.

Si la navigabilité est bidirectionnelle, cela signifie que si l'on navigue d'un objet à l'autre, il faut pouvoir le faire dans les deux sens ! Cela nécessite donc de mettre à jour la navigabilité dans l'autre sens.

La mise à jour de l'objet lié peut être faite de deux façons, soit manuellement, soit automatiquement.

10.3. Mise à jour manuelle de l'association bidirectionnelle

Ce qu'il faut bien comprendre, c'est que lorsque nous associons un technicien à un véhicule, ce technicien ne sait pas qu'il est associé à ce véhicule. Nous pouvons le faire manuellement :

```php
<?php


$vehicleBBBB = new Vehicle('BBBB');
$anna = new Technician('Anna');

//nous associons le technicien au véhicule (navigabilité de Vehicle vers Technician)
$vehicleBBBB->setTechnician($anna);

//nous associons également le véhicule au technicien (navigabilité de Technician vers Vehicle)
$anna->setVehicle($vehicleBBBB);

//Nous pouvons naviguer de Vehicle vers Technician :
echo $vehicleBBBB;

//Nous pouvons naviguer de Technician vers Vehicle :
echo $anna;
```

Résultat :

```
Je suis le véhicule immatriculé BBBB. Mon technicien est Anna.

Je suis le technicien nommé Anna. La voiture dont j'ai la charge a pour immatriculation BBBB.
```

Nous pouvons être satisfaits du résultat. Les deux objets associés se connaissent réciproquement. La navigabilité est bien bidirectionnelle.

Il faut toujours garder en tête que l'association des objets liés dans une association bidirectionnelle n'est pas réciproque.

La solution de mettre à jour manuellement les liens entre les objets souffre d'une limite importante : **le développeur ne doit pas oublier de faire l'association dans les deux sens !**

Heureusement, il est possible d'éviter cette situation en prévoyant la mise à jour automatique de l'objet lié.

10.4. Mise à jour automatique de l'association bidirectionnelle

Rappelons le processus de mise à jour d'une association bidirectionnelle entre un objet de type `A` et un objet de type `B` :

1. L'objet `B` est lié à l'objet `A` en faisant `$a->setB($b)`
2. L'objet `A` est ensuite lié à l'objet `B` en faisant `$b->setA($a)`

Pour automatiser ces deux étapes, il suffit lors de l'étape 1 de déclencher l'étape 2.

Si l'on prend la méthode `Vehicle::setTechnician()` suivante :

```
1  <?php
2
3
4
5      /**
6       * @param Technician|null $technician
7       *
8       * @return Vehicle
9       */
10     public function setTechnician(?Technician $technician): Vehicle
11     {
12
13         //on associe le nouveau technicien au véhicule
14         $this->technician = $technician;
15
16         return $this;
17     }
```

Nous avons stocké le technicien dans l'attribut `technician` de la classe `Vehicle`. Pour cela nous avons mobilisé la méthode `setTechnician()`. Cela correspond à la première étape de la mise à jour de l'association bidirectionnelle. Nous allons imbriquer dans cette étape l'étape 2 (l'association d'une voiture à un technicien) :

```
1  <?php
2
3
4
5      /**
6       * @param Technician|null $technician
7       *
8       * @return Vehicle
9       */
10     public function setTechnician(?Technician $technician): Vehicle
11     {
12         //mise à jour de l'objet lié (ici le technicien à qui l'on affecte la
```

```
    voiture courante $this)
13          if (null !== $technician) {
14              $technician->setVehicle($this);
15          }
16
17          //on associe le nouveau technicien au véhicule
18          $this->technician = $technician;
19
20          return $this;
21      }
```

Désormais, lorsqu'un technicien est affecté à une voiture, l'association inverse est également réalisée :

```
1  <?php
2
3
4  $vehicleIIII = new Vehicle('IIII');
5  $malo = new Technician('Malo');
6
7  //On associe un véhicule au technicien
8  $vehicleIIII->setTechnician($malo);
9
10 //la navigabilité est maintenant possible depuis l'objet lié
11 var_dump($malo->getVehicle()); // IIII
```

Nous avons bien depuis le technicien accès au véhicule.

Si nous sommes dans le cas où le véhicule est associé une première fois à un technicien puis une seconde fois à un autre technicien, le premier technicien sera encore lié au véhicule alors qu'il ne le devrait plus (la navigabilité bidirectionnelle est rompue). Il faut alors indiquer à cet ancien technicien qu'il n'est plus lié au véhicule courant :

```
1  <?php
2
3
4
5      /**
6       * @param Technician|null $technician
7       *
8       * @return Vehicle
9       */
10     public function setTechnician(?Technician $technician): Vehicle
11     {
12         //mise à jour de l'objet lié (ici le technicien à qui l'on
   affecte la voiture courante $this)
13         if (null !== $technician) {
```

```
14              $technician->setVehicle($this);
15          }
16          //l'ancien technicien affecté au véhicule courant ne doit
    plus l'être
17          if (null !== $this->technician) {
18              $this->technician->setVehicle(null); ①
19          }
20
21          //on associe le nouveau technicien au véhicule
22          $this->technician = $technician;
23
24          return $this;
25      }
```

① L'ancien technicien n'est plus lié au véhicule courant, d'où la valeur null passée en argument (et parce que le diagramme nous indique que l'attribut est nullable du fait de la cardinalité minimale à 0)

Désormais, lorsqu'un nouveau technicien sera associé à la voiture, l'ancien ne le sera plus :

```
1  <?php
2
3
4  $vehicleEEEE = new Vehicle('EEEE');
5  $cedric = new Technician('Cédric');
6
7  //une seule affectation depuis la voiture
8  $vehicleEEEE->setTechnician($cedric);
9
10 //la navigabilité est possible dans les deux sens
11 var_dump($vehicleEEEE->getTechnician()); //Cédric
12 var_dump($cedric->getVehicle()); // EEEE
13
14 //le véhicule est associé à un nouveau technicien
15 $karl = new Technician('Karl');
16 $vehicleEEEE->setTechnician($karl);
17
18 //le nouveau technicien est bien lié au véhicule (bidirectionnelle
    ok)
19 var_dump($karl->getVehicle()); // EEEE
20
21 //l'ancien technicien n'est plus lié au véhicule EEEE
22 var_dump($cedric->getVehicle()); // null
```

Tout fonctionne comme attendu. En faisant une seule association, les deux objets sont liés réciproquement (et l'ancien lien est correctement "défait").

C'est super mais que se passe-t-il si le lien est initié depuis une instance de `Technician` ?

```php
<?php


$vehicleHHHH = new Vehicle('HHHH');
$julien = new Technician('Julien');

//cette fois, on associe un véhicule au technicien
$julien->setVehicle($vehicleHHHH);

//la navigabilité doit être possible dans les deux sens
var_dump($vehicleHHHH->getTechnician()); //NULL ①
var_dump($julien->getVehicle()); // HHHH
```

① Il n'est pas possible de naviguer de l'instance de `Vehicle` vers l'instance de `Technician`.

Que s'est-il passé ?

L'association d'un technicien et d'un véhicule a été initiée via la méthode `Technician::setVehicle()`. Cette méthode ne fait qu'associer un véhicule à un technicien. Elle ne s'occupe donc pas d'associer au véhicule ce technicien.

Dans le cas d'une association bidirectionnelle, seul un des deux objets liés est responsable de la mise à jour de l'autre.

Par conséquent, dans notre cas, l'association doit obligatoirement être réalisée depuis une instance de `Vehicle` via sa méthode `setTechnician` puisque cette dernière contient l'appel à l'association inverse. La classe `Vehicle` est responsable de la mise à jour de l'association dans les deux sens.

Attention : il faut savoir déterminer la classe responsable de cette mise à jour ! C'est l'objetif du point suivant.

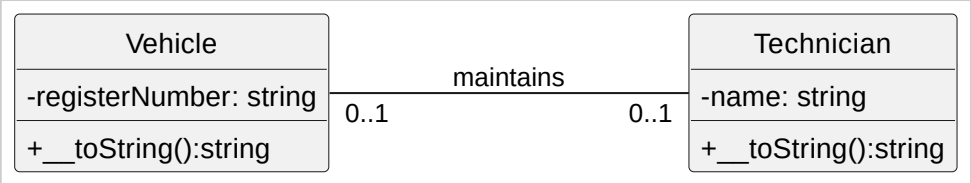
10.5. Choisir l'objet qui sera responsable de la mise à jour de l'objet lié

Nous venons de voir que dans le cadre d'une association bidirectionnelle, il faut associer deux objets depuis l'objet qui est responsable de la mise à jour de l'objet lié (ou objet inverse).

La classe qui est responsable de la mise à jour de l'objet lié est appelée **classe propriétaire** (on peut trouver le terme de **classe possédante** ou de **classe dominante**. L'objet lié est appelé **objet inverse** (ou **classe inverse** si on parle de classe.)

En parallèle, si l'on parle des instances de ces classes, on pourra utiliser les termes d'**objet possédant** ou d'**objet propriétaire** ou d'**objet dominant** ou tout simplement d'**objet responsable de la mise à jour** de l'objet associé.

Précédemment, nous avons travaillé avec ce diagramme :



Nous avons placé dans la méthode `Vehicle::setTechnician()` l'appel à `Technician::setVehicle` afin de mettre à jour l'association dans le sens inverse.

J'avais arbitrairement choisi la classe `Vehicle` pour être la classe possédante, c'est-à-dire la classe qui est responsable de la mise à jour de l'objet lié.

Le choix de la classe possédante ne doit pas être fait au hasard.

La classe propriétaire doit être celle qui est à l'opposée de la cardinalité maximale à 1.

Appliquer cette règle se révèlera très utile si jamais les instances des classes liées doivent être persistées en base de données.

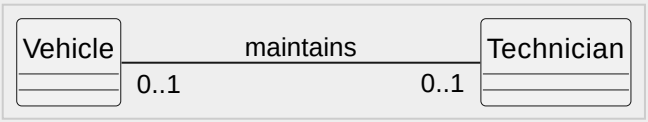
S'il y a une cardinalité maximale à 1 de chaque côté de l'association, alors ce peut être l'une ou l'autre classe.

S'il y a une cardinalité maximale à plusieurs de chaque côté de l'association, alors là aussi, ce peut être l'une ou l'autre classe.

Lorsque la classe possédante peut être l'une ou l'autre des classes associées, il faut retenir la classe depuis laquelle il est le plus logique de faire l'association. Par exemple, s'il paraît plus naturel de partir d'un véhicule pour lui associer un technicien, alors c'est que la classe `Vehicle` domine la classe `Technician`. Ce sera donc elle qui sera responsable de la mise à jour de l'objet inverse.

Q7) Voici différents diagrammes. Pour chacun d'eux, précisez quelle sera la classe propriétaire et justifiez votre choix.

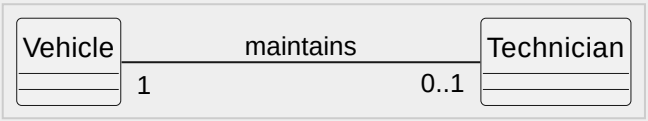
a. Diagramme :



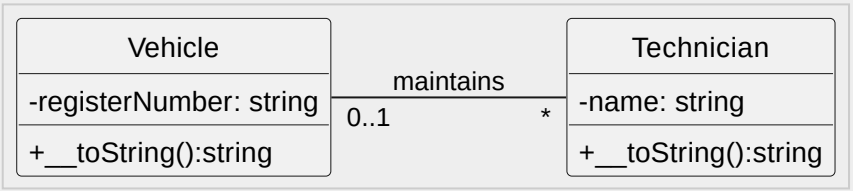
b. Diagramme :

c. Diagramme :

d. Diagramme :



e. Diagramme :



Je le répète avant de terminer cette partie car c'est vraiment très important :

La classe propriétaire doit être celle qui est à l'opposée de la cardinalité maximale à 1 lorsque c'est possible. A défaut ce sera la classe qui domine l'autre du fait de son utilisation naturelle.

11. Implémentation d'une association bidirectionnelle multiple

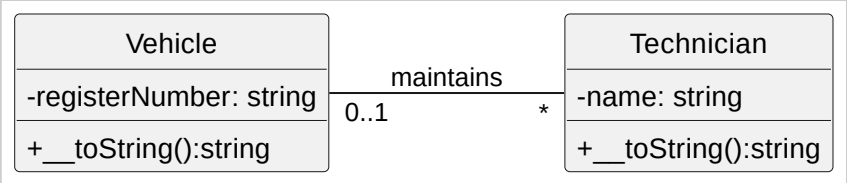
Rappel 1 : une association entre deux classes `A` et `B` traduit un lien de contenance. Dans ce cas `A` doit prévoir un attribut permettant de stocker une instance de `B` et/ou vice-versa.

Rappel 2 : une association bidirectionnelle est navigable dans les deux sens (de `A` vers `B` **ET** de `B` vers `A`).

Rappel 3 : Une association est qualifiée de multiple lorsque zéro, une ou plusieurs instances de `B` sont liées à `A` (ou l'inverse).

Rappel 4 : Dans une association bidirectionnelle, il faut choisir la classe propriétaire qui va être responsable de la mise à jour de l'objet inverse.

Le diagramme suivant exprime une association bidirectionnelle multiple car au moins une des cardinalités maximales de l'association est à plusieurs.



Lecture de l'association : Un véhicule est maintenu par 0, 1 ou plusieurs techniciens et un technicien maintient 0 ou un véhicule.

Nous avons appris dans le cadre de l'implémentation d'une association bidirectionnelle simple qu'il fallait choisir la classe propriétaire.

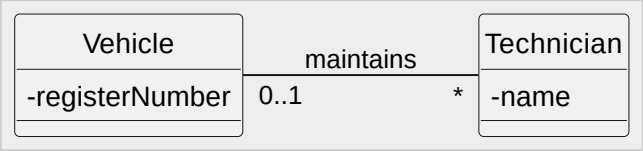
Compte tenu de notre diagramme, la classe propriétaire sera `Technician` car c'est celle qui est à l'opposée de la cardinalité maximale à 1.

A ce niveau, il n'y a plus rien de nouveau pour nous. L'implémentation se fera en plusieurs étapes :

1. Mettre en place la navigabilité de `Vehicule` vers `Technician` (en utilisant un attribut permettant de contenir une collection)
2. Mettre en place la navigabilité de `Technician` vers `Vehicule` (en utilisant un attribut permettant de contenir zéro ou une instance de `Vehicle`)
3. Mettre en place dans la classe propriétaire `Technician` la mise à jour de l'objet inverse.

Q8) Travail à faire

- Implémentez scrupuleusement le diagramme suivant (les propriétés `registerNumber` et `name` seront toutes les deux initialisées dans les constructeurs. Inutile de prévoir des mutateur et accesseur les concernant) :



- Testez votre implémentation en répondant aux questions suivantes :

a. Créez deux instances de véhicules respectivement référencées par les variables `$vA` et `$vB`. Faire un `var_dump()` de chaque variable et noter l'identifiant propre à chaque objet (un identifiant est un `#` suivi d'un chiffre tel que `#1`)

b. Créez trois instances de technicien respectivement référencées par les variables `$paul`, `$juliette` et `$jalila` (leur affecter le prénom correspondant aux noms des variables). Faire un `var_dump()` de chaque technicien et noter l'identifiant d'objet qui leur est propre.

c. Associez le véhicule A aux techniciens Paul et Juliette et le véhicule B au technicien Jalila (il ne faut pas oublier que l'objet responsable de la mise à jour de l'objet lié est la technicien). Faire un var_dump de chaque véhicule afin de constater que la mise à jour de l'objet inverse (ici Vehicle) a été réalisée à partir de la classe Technician. Noter pour chaque voiture le ou les techniciens associés.

d. Associez le véhicule B au technicien Paul (sans oublier qui est la classe propriétaire). Constatez que le fait d'avoir affecté le véhicule B à Paul à produit trois effets :

   - Paul n'est plus associé au véhicule A
   - Le véhicule A n'est plus associé au technicien Paul mais encore à Juliette.
   - Le véhicule B est associé au technicien Paul

Q9) Travail à faire

- Implémentez scrupuleusement le diagramme ci-dessous (les propriétés `registerNumber` et `name` seront toutes les deux initialisées dans les constructeurs. Inutile de prévoir des mutateur et accesseur les concernant). (testez votre implémentation en ajoutant / retirant des techniciens aux voitures et en consultant le contenu des objets avec des `var_dump()`)

Nous partirons du principe que la classe propriétaire est `Technician` afin que la correction corresponde à votre travail.

12. Implémentation des cardinalités

Nous avons appris lors de l'étude de la notion de cardinalité qu'elle permet d'exprimer une contrainte

Dans la partie du cours sur les cardinalités, nous avons vu que les cardinalités exprimait une contrainte sur le nombre d'objets B associés à un objet A.

Le développeur doit tenir compte de celles-ci dans l'implémentation de la classe.

Pour prendre en compte la cardinalité à l'extrémité d'une association navigable, le développeur doit compter le nombre d'instances liées et s'assurer que ce nombre respecte cette cardinalité. En PHP, la fonction `count` retourne le nombre d'éléments dans un tableau (utile pour dénombrer une collection).

Les cardinalités minimale et maximale doivent être vérifiées par le développeur.

Il n'y a aucune difficulté dans le contrôle des cardinalités. Ainsi, vous pouvez attaquer les exercices qui suivent.

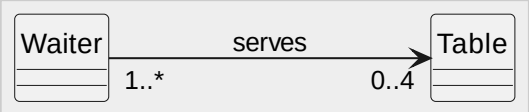
Q10) Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ? Si oui, indiquez pour chacune d'elle si le contrôle doit être fait dès l'instanciation de l'objet depuis lequel commence la navigabilité ou après (dans ce cas préciser depuis quelle méthode).

Q11) Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?

Q12) Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?

Q13) Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?

Q14) Implémentez le diagramme suivant :



Lorsqu'une cardinalité n'est plus respectée, une exception doit être levée avec un message explicatif.

En PHP, une exception se lance de la façon suivante :

```php
throw new Exception('ici un message');
```

Il n'est pas demandé de gérer l'exception. Elle sera seulement levée ce qui mettra fin au programme.

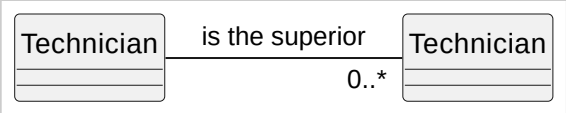
Q15) Implémentez le diagramme suivant et levez une exception lorsque la cardinalité n'est pas respectée :

Q16) Implémentez le diagramme suivant et levez une exception lorsque la cardinalité n'est pas respectée :

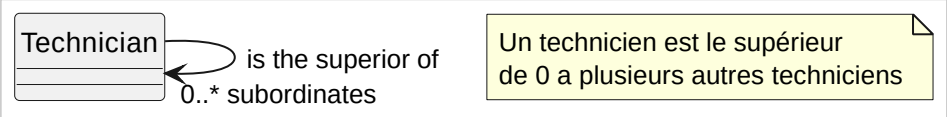
13. L'association réflexive

Une association réflexive est un lien entre deux objets de même type.

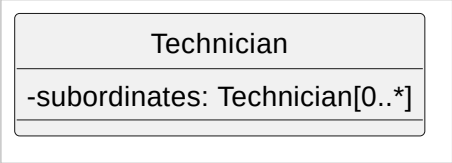
Imaginons un technicien qui peut être le supérieur hiérarchique d'autres techniciens. Le diagramme suivant illustre cette relation :



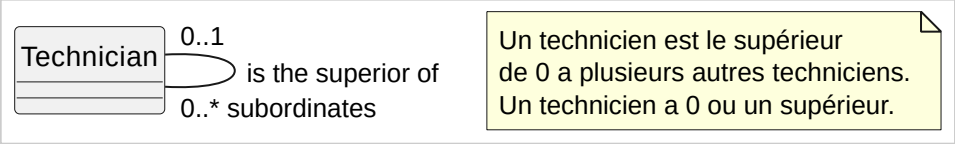
Comme les deux classes mobilisées sont identiques, il ne faut en utiliser qu'une seule et donc faire un lien qui point sur elle-même :



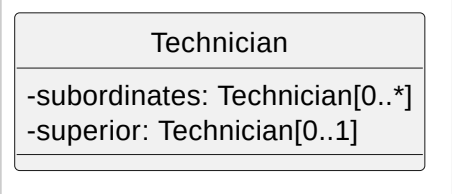
C'est équivalent à cette représentation :



Voici la même modélisation mais avec une bidirectionnalité :

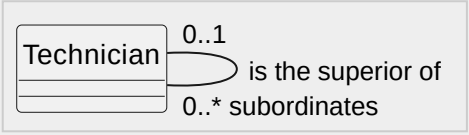


Ce qui est équivalent à :



Q17) Travail à faire

a. Implémentez le diagramme suivant (il n'y a rien de nouveau, cela reste une association bidirectionnelle comme nous savons les implémenter) :



b. Vous veillerez à ce qu'un technicien ne puisse pas être son propre subordonné ou supérieur.

14. L'agrégation

14.1. Qu'est-ce qu'une agrégation ?

Rappel : l'association traduit un lien entre deux objets.

Le terme d'agrégation signifie l'action d'agréger, d'unir en un tout.

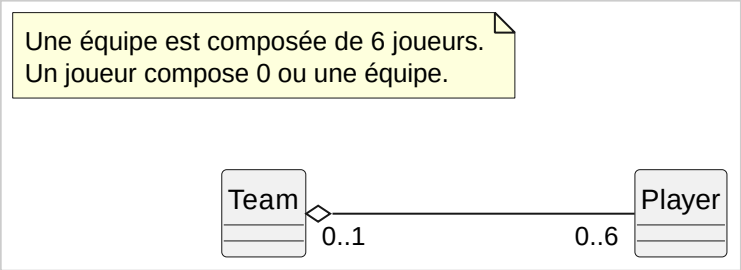
L'**agrégation** exprime la construction d'un objet à partir d'autres objets. Cela se distingue de la notion d'association que nous avons abordée jusqu'à maintenant. Effectivement, l'association traduit un lien entre deux objets alors que l'agrégation traduit le "regroupement" ou "l'assemblage" de plusieurs objets. Le lien exprimé est donc plus fort que pour une association.

Imaginez des pièces de Légo que vous utilisez pour construire une maison. Chaque pièce est un objet qui une fois agrégée avec les autres pièces permettent d'obtenir un autre objet (la maison). Il est tout à fait possible d'utiliser chaque pièce pour faire une autre construction. Détruire la maison ne détruit pas les pièces.

Implicitement, l'agrégation signifie « contient », « est composé de ». C'est pour cela qu'on ne la nomme pas sur le diagramme UML.

Autrement dit, une agrégation est le regroupement d'un ou plusieurs objets afin de construire un objet « complet » nommé **agrégat**.

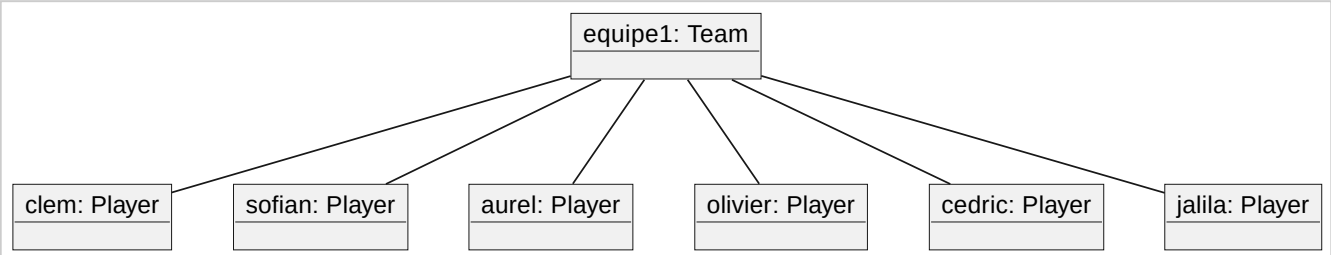
Représentons le lien entre une équipe et les joueurs qui composent celle-ci (ici une équipe de volley) :



Le losange vide est le symbole qui caractérise une agrégation. Il est placé du côté de l'agrégat (l'objet qui est composé / assemblé).

La classe `Team` est l'**agrégat** (le composé de) alors que la classe `Player` est le **composant**.

Ce diagramme objet représente les joueurs qui composent une équipe de volley



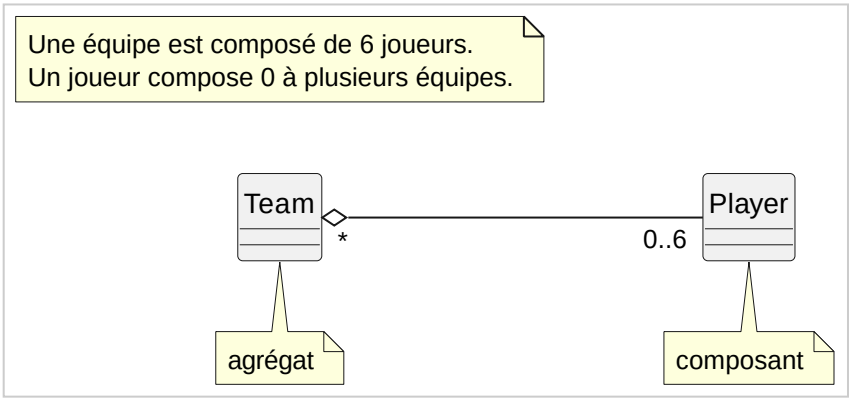
Une agrégation présente les caractéristiques suivantes :

- L'agrégation est composée d'"éléments".
- Ce type d'association est non symétrique. Il n'est pas possible de dire "Une équipe est composée de joueurs et un Joueur est composé d'une équipe"
- les composants de l'agrégation sont **partageables**
- l'agrégat et les composants ont leur **propre cycle de vie**)

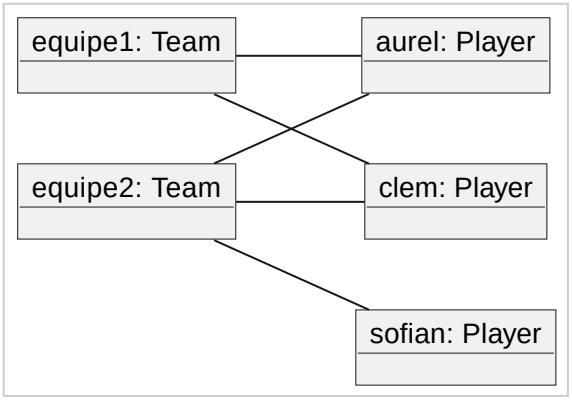
Les composants sont partageables :

La particularité d'une agrégation est que **le composant peut être partagé**.

Par exemple, un joueur d'une équipe peut jouer (se partager) dans d'autres équipes :



Voici un diagramme objet pour illustrer ce partage :



Les instances de `Player` "aurel" et "clem" sont **partagées** dans deux équipes. Celle représentant "sofian" n'est pas partagée mais peut l'être à un moment donné.

Voici un autre exemple :

L'entreprise est la réunion en un tout de personnes et de locaux. Les personnes qui composent une entreprise peuvent travailler dans d'autres et un local peut servir à plusieurs entreprises. Nous

retrouvons la notion de partage des composants.

Comme les composants sont partageables, la multiplicité du côté de l'agrégat peut être supérieur à 1.

L'agrégat et les composants ont leur propre cycle de vie

Le cycle de vie d'un objet désigne sa création, ses changements d'état jusqu'à sa destruction.

- Un agrégat **peut** exister sans ses composants. (en programmation, il doit être possible d'instancier un agrégat sans ses composants)
  Une équipe peut exister même s'il elle n'a aucun joueur.
- Un composant **peut** exister sans être utilisé par l'agrégat. (en programmation, il doit être possible d'instancier un composant sans que l'agrégat l'utilise ou existe)
- la destruction de l'agrégat ne détruit pas ses composants (et vice versa) ce qui va dans le sens des deux points précédents

14.2. Navigabilité et agrégation

Tout ce que nous avons vu dans la partie sur la navigabilité s'applique dans le cadre d'une agrégation.

Il y a navigabilité bidirectionnelle entre `Person` et `Enterprise` et navigabilité unidirectionnelle de `Enterprise` vers `Local`.

14.3. Implémentation d'une agrégation

L'implémentation d'une agrégation est exactement la même qu'une association classique.

L'agrégation permet seulement d'exprimer conceptuellement le fait que les instances d'une classe sont des "assemblages" d'autres instances de classe. Une conceptualisation est une représentation de la réalité. Cela peut aider à mieux cerner la logique métier de l'application.

15. La composition

15.1. Qu'est-ce qu'une composition ?

Si l'agrégation désigne un assemblage d'objets, la composition exprime la même chose à la différence près que ce qui

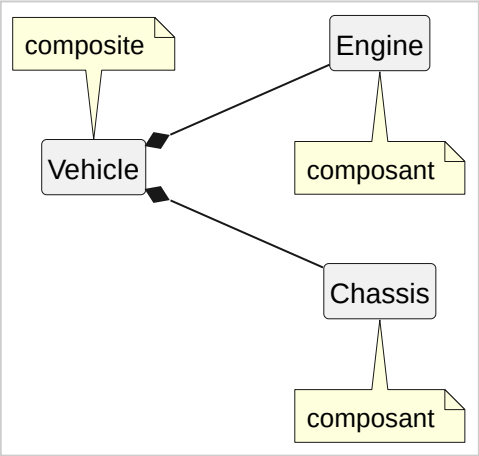
La composition reprend l'idée de l'agrégation. La composition exprime un assemblage d'objets. Cet assemblage est tellement "fort" que les objets assemblés ne peuvent pas servir dans un autre assemblage.

Imaginons des parpaings et le mortier qui permettent de construire un mur. Les parpaings et le mortier sont les composants et le mur est le composé ou la composition. Une fois le mur réalisé, les parpaings ne peuvent pas être utilisés pour construire un autre mur. De plus, si le mur est détruit, les parpaings le sont également.

Une **composition** est donc une association qui traduit un assemblage d'objets tellement fort que ceux-ci ne peuvent faire partie d'un autre assemblage. Les éléments assemblés sont appelés des **composants** et le résultat de leur assemblage est appelé une composition ou un **objet composite**.

Imaginons un concessionnaire de véhicules. Pour ce dernier, une voiture est composée d'un moteur et d'un chassis. La voiture est donc composée de deux éléments.
La voiture ne devient une voiture qu'à l'assemblage du chassis et du moteur. Sans l'un ou l'autre, ou sans les deux, l'objet voiture n'existe pas.
Par ailleurs, si la voiture est détruite, le moteur et son chassis le sont également.

Voici le diagramme de classes qui représente cette situation :



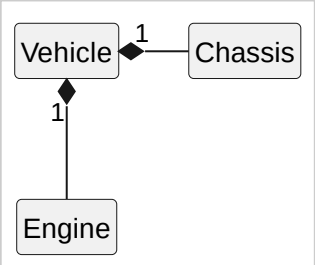
- L'objet composé d'éléments est appelé **composite**.
- L'objet qui compose le composite est un **composant**.
- Le losange **plein** est placé du côté du composite.

La composition présente donc les caractéristiques suivantes :

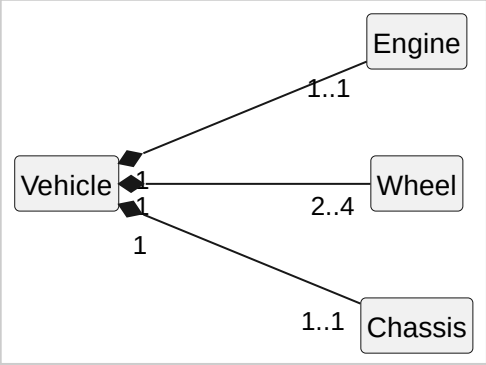
- **la destruction du composite détruit également ses composants**.
- **le composant ne peut pas être partagé** (c'est logique puisque si le composite qui le contient est

détruit, le composant est aussi détruit. Il ne peut donc être utilisé par un autre objet.)

- puisque le composant ne peut pas être partagé, **la cardinalité du côté du composite est forcément 1** (c'est pourquoi elle n'est en fait jamais précisée). Un même moteur ne peut être utilisé par deux véhicules.

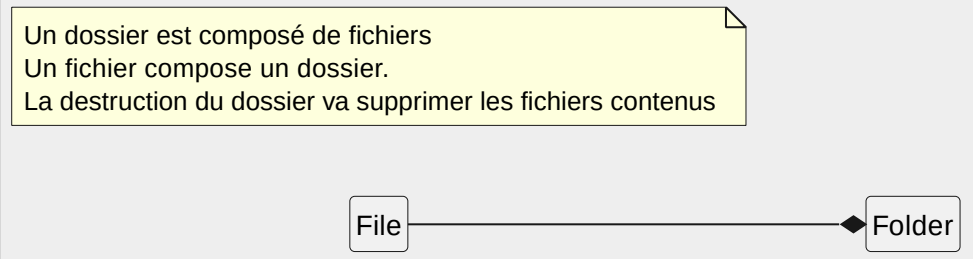


- La composition implique que **le composite contient ses composants dès sa création** d'où l'absence de cardinalités minimales à 0 (en voici une illustration avec en plus des roues)



Q18) La modélisation précédente serait-elle la même pour une casse automobile ?

Q19) A votre avis, peut-on modéliser le diagramme suivant ?



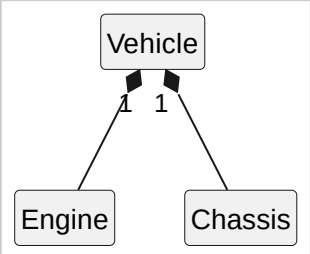
15.2. Navigabilité et composition

Tout ce que nous avons vu dans la partie sur la navigabilité s'applique dans le cadre d'une composition à une exception près : **Il est toujours possible de naviguer de la composition vers ses composants.** C'est logique puisque le composite "connait" ses composants dès sa création.

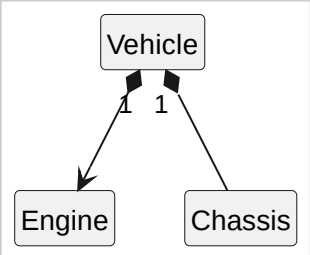
Il est toujours possible de naviguer de la composition vers ses composants.

Le diagramme suivant nous indique qu'il est possible de naviguer de `Vehicle` vers `Engine` et de

`Engine` vers `Vehicle`. Il en va de même avec le composant `Chassis`.



Il est possible de restreintre la navigabilité d'une relation de composition mais seulement de la composition vers le composant :



Le diagramme nous indique qu'il y a navigabilité bidirectionnelle entre `Vehicle` et `Chassis` et navigabilité unidirectionnelle de `Vehicle` vers `Engine`.

15.3. Implémentation d'une composition

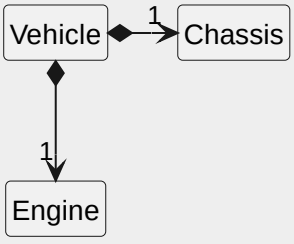
La composition nécessite une implémentation qui prend en compte ces caractéristiques :

- la création du composite nécessite les composants
- composants non partageables
- destruction du composite = destruction des composants

Q20) Implémentez le diagramme de classes suivant compte tenu des notions qui ont été abordées dans cette partie.

Dans cette implémentation, il sera considéré que les composants peuvent exister avant d'être utilisés dans une composition. **Dans la solution proposée, les composants seront donc instanciés avant d'instancier la composition.**

Ce travail est loin d'être trivial, c'est pourquoi votre travail consiste à étudier attentivement la correction proposée. Veillez à réellement comprendre le code et ses explications !



Pour rappel, une composition est responsable du cycle de vie de ses composants. Ainsi, si le composite est détruit, les composants doivent l'être également.

Cela m'amène à vous rappeler deux aspects techniques propres à PHP :

- En php, pour réellement détruire un objet, il faut détruire **toutes** ses références.
- En php, toutes les références existantes sont détruites à la fin du script.

Correction de Q20 (affichée volontairement du fait de l'approche qui n'est pas toujours simple à "deviner")

La correction de cette question va être faite en plusieurs parties :

- les classes `Vehicle`, `Chassis` et `Engine` vont être implémentées dans un premier temps
- 3 scénarios d'utilisation de ces classes vont être expliqués pour respecter le principe suivant : "si le composite est détruit, les composants le sont également"

En PHP, comme le souligne la note dans la question, il faut détruire toutes les références pour détruire l'objet référencé.

- les 3 classes à implémenter :

```
class Vehicle
{

    //dans une composition, les composants sont indispensables à la création du
composite.
    //ils ne peuvent donc pas être null
    public function __construct(
        private Chassis $chassis,
        private Engine $engine
    ) {


    }

    public function __destruct()
    {
        echo "voiture détruite\n";
    }

    //ici les mutateurs et accesseurs des attributs d'objet
}
```

```php
class Chassis
{

    public function __destruct()
    {
        echo "chassis détruit\n";
    }
}

class Engine
{
    public function __destruct()
    {
        echo "moteur détruit\n";
    }
}
```

- **[solution 1]** Création d'un véhicule avec ses composants (observez bien la destruction du composite et des composants)

```php
$c = new Chassis();
$e = new Engine();

//création du composite
$v = new Vehicle($c, $e);

//destruction du véhicule
unset($v); ①
// à noter que détruire $v détruit également la référence stockée dans $this->chassis
de $v (idem pour la référence au moteur ($this->engine)

//si une référence au chassis a été détruite avec l'objet véhicule, il reste la
référence stockée dans la variable $c. Il faut donc la détruire également.
unset($c); ②
//même remarque pour le moteur
unset($e); ②
echo "\n---FIN DU SCRIPT---\n";

//A ce stade, le composite et ses composants sont détruits. Il n'est plus possible de
les manipuler.
```

① Le destruction de l'objet véhicule détruit également les deux références qu'il contenait vers le chassis et le moteur.

② Il ne faut pas oublier de supprimer toutes les autres références vers les composants du véhicule détruit.

Cela affiche la sortie suivante qui nous indique bien que les éléments sont détruits :

```
voiture détruite
chassis détruit
moteur détruit

---FIN DU SCRIPT---
```

Pour bien comprendre ce point, je vous invite à regarder cette vidéo de 7min sur le destructeur.

- **[solution 2]** Création d'un véhicule en passant les instances directement en argument sans les référencer avant son instanciation :

```php
//Les objets chassis et moteur qui composent la voiture ne sont pas référencés par
d'autres variables que celles qui sont utilisées dans l'objet véhicule instancié
$v2 = new Vehicle(new Chassis(), new Engine());

//la destruction est alors très simple : il suffit de détruire l'objet véhicule
unset($v2);

echo "\n---FIN DU SCRIPT---\n";
```

Les éléments sont également tous détruits avec cette solution qui évite d'oublier de supprimer toutes les références aux composants :

```
voiture détruite
chassis détruit
moteur détruit

---FIN DU SCRIPT---
```

- [solution 3] : Les objets composants sont directement instanciés dans le constructeur ce qui nous assure qu'il n'y a pas de référence extérieure (enfin, tant qu'aucune méthode ne retourne un des composants à un programme appelant).

```php
class Vehicle2
{

    //déclaration des composants
    private Chassis $chassis;
    private Engine $engine;

    public function __construct()
    {
        //création des composants à la création du véhicule
        $this->chassis = new Chassis();
        $this->engine = new Engine();
    }
```

```
①
    //!!!ATTENTION!!!
    //IL NE DOIT PAS Y AVOIR DE MUTATEURS OU D'ACCESSEURS pour les variables qui
référencent les composants car il ne doit pas être possible de manipuler ces
composants depuis l'extérieur de la classe

    public function __destruct()
    {
        echo "voiture détruite\n";
    }

}


$v3 = new Vehicle2();

unset($v3);

echo "\n---FIN DU SCRIPT---\n";
```

① pas de mutateurs et d'accesseurs pour les composants car ils ne doivent pas être accessibles depuis l'extérieur de la classe. Ainsi, aucune référence extérieure ne pourra être créée.

Nous arrivons au même résultat :

```
voiture détruite
chassis détruit
moteur détruit

---FIN DU SCRIPT---
```

Nous avons 3 solutions possibles, alors laquelle choisir ?

La solution qui consiste à prévoir l'instanciation des composants dans le constructeur de la classe composite est la plus proche du concept de composition (solution 3).

Dans la réalité applicative, les composants sont souvent instanciés en dehors de l'objet composite et passés en argument du constructeur (solution 1).

16. L'association n-aire

16.1. Qu'est-ce qu'une association n-aire ?

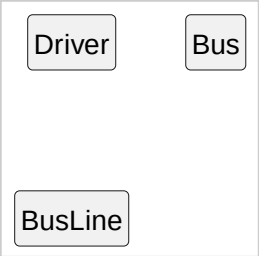
Jusqu'à maintenant, nous n'avons abordé que des relations entre deux classes. Cela couvre la très grande majorité des cas mais parfois, 3 classes ou plus peuvent être liées entre elles.

Une **relation n-aire** est une relation entre au moins 3 classes (jusqu'à maintenant, il s'agissait de relations binaires). Les cas d'utilisation sont rares, ou plutôt rarement pertinents car il n'est pas rare de trouver des relations n-aires qui n'en sont pas. Chaque instance de l'association est un tuple de valeurs provenant chacune de leur classe respective. Ce n'est pas facile à comprendre, j'en conviens.

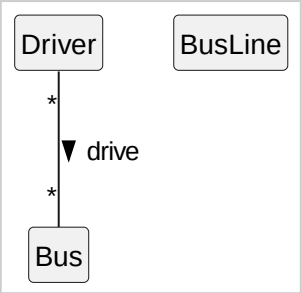
Je vais expliquer ce concept en m'appuyant sur un cas qui va évoluer progressivement.

Imaginons que nous devions modéliser des chauffeurs qui conduisent des bus de ville sur des lignes de bus.

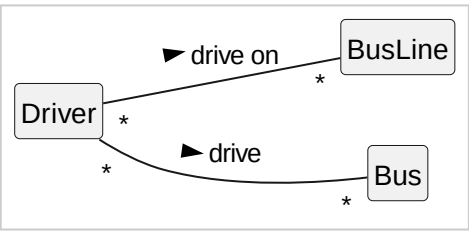
Nous avons 3 classes à modéliser :



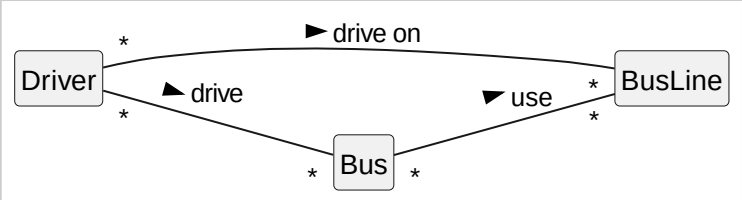
Nous savons qu'un chauffeur ne conduit pas toujours le même bus et qu'un bus peut être conduit par plusieurs chauffeurs. Cela donne la modélisation suivante :



Un chauffeur conduit sur des lignes de bus. Nous pourrions être tentés de modéliser la solution suivante :



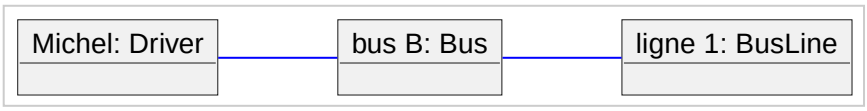
Et de continuer en indiquant qu'un bus circule sur une ligne de bus qui n'est pas toujours la même :



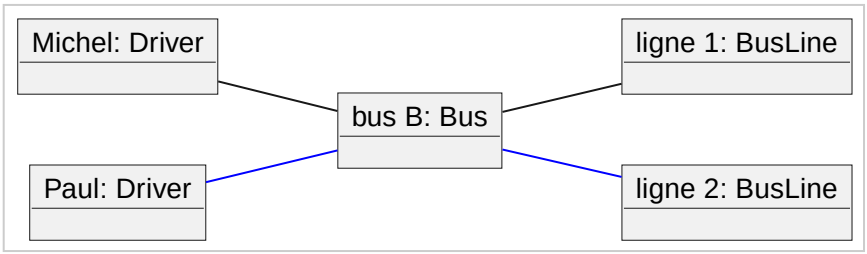
Cependant, cette modélisation conduit à ne connaître que les couples Driver/Bus, Driver/BusLine, Bus/BusLine.

Nous allons utiliser notre diagramme de classes pour mettre en avant la limite de ce qu'il conceptualise. Pour cela, nous allons faire évoluer un diagramme d'objets à partir de phrases simples qui s'appuient sur notre dernier diagramme de classes.

- Michel conduit le bus B sur la ligne 1 :

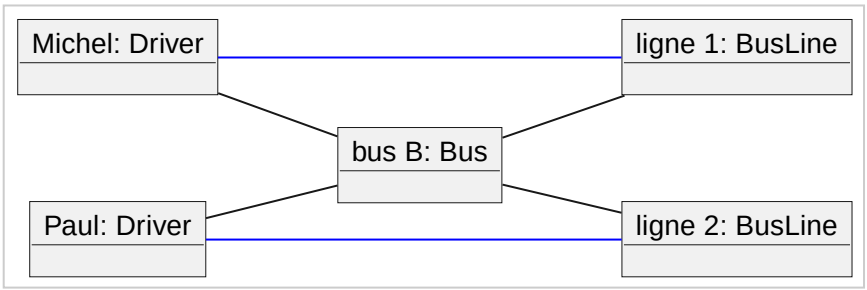


- Paul conduit le bus B sur la ligne 2



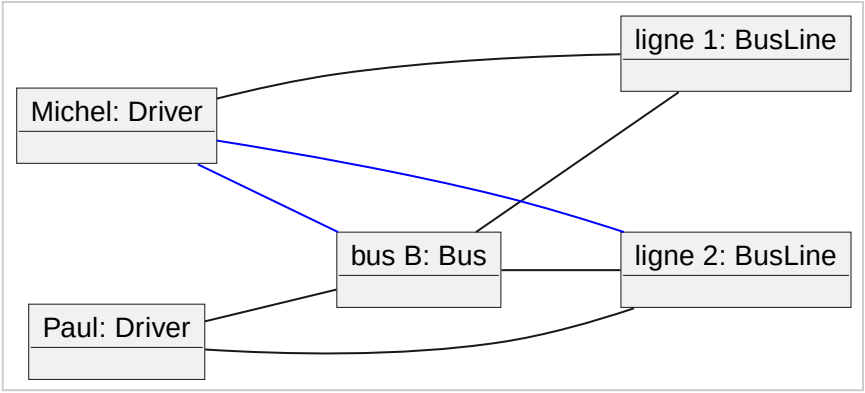
A ce stade, il n'est plus possible de savoir sur quelle ligne de bus Paul a conduit le bus B car ce bus est lié à deux lignes. Mais, notre diagramme de classes de tout à l'heure nous montre qu'il y a une association entre `Driver` et `BusLine`. Nous pouvons donc associer la ligne de bus au chauffeur :

- Michel a conduit sur la ligne 1 et Paul sur la 2



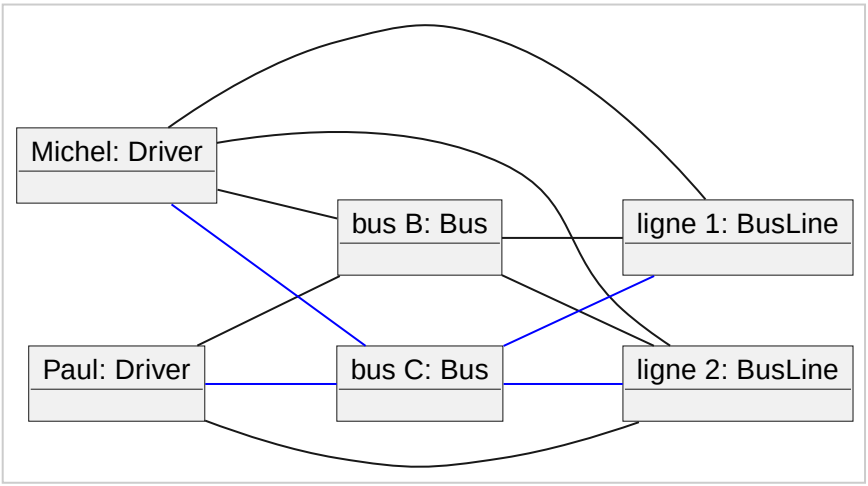
On peut maintenant affirmer que Michel a conduit le bus B et qu'il l'a fait sur la ligne 1. Mais que se passe-t-il s'il doit conduire le bus B sur la ligne 2 ?

- Michel conduit également le bus B sur la ligne 2



Il reste encore possible de dire que Michel a conduit le bus B sur les lignes 1 et 2.

- Maintenant, prenons en compte le fait que Michel et Paul conduisent un bus C sur les lignes 1 et 2 :



Nous sommes coincés maintenant ! Il n'est plus possible de savoir sur quelles lignes ont été conduit chaque bus.

Lorsque l'on navigue d'un objet à l'autre, voici que l'on peut avancer :

- On peut dire que Michel conduit les bus B et C. (association Driver/Bus)
- On peut dire que Michel conduit sur les lignes 1 et 2 (association Driver/LineBus)
- On peut dire que le bus B circule sur les lignes 1 et 2 (association Bus/BusLine)

Voici ce que l'on ne peut pas affirmer :

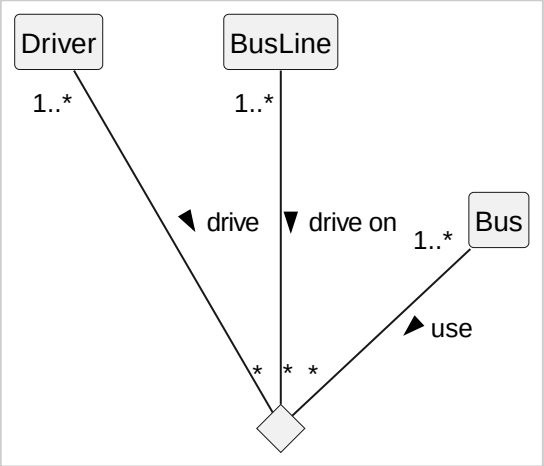
- Michel conduit le bus B sur la ligne 1 (effectivement, le bus B est lié à 2 lignes, cela ne veut pas dire que Michel à conduit sur la ligne 1. Ce peut être Paul)
- La ligne 2 est utilisée par Paul avec le bus B (effectivement, la ligne 2 est liée au bus B mais c'est peut être Paul qui conduisait).

S'il n'est pas possible de déterminer qui à conduit tel bus sur telle ligne, alors la modélisation proposée n'est pas bonne. Nous avons besoin de connaître "en même temps" le chauffeur, le bus qu'il utilise et la ligne sur laquelle il roule. Pour cela il faut "associer" les 3 classes ensembles.

La solution est **d'associer** `Driver` / `BusLine` et `Bus`, de les lier ensemble. Lorsque l'on associe / relie

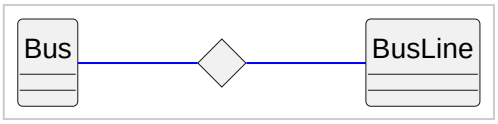
trois classes, on parle d'**association ternaire**.

Voici la modélisation qui permet de savoir que Michel a conduit le bus B sur la ligne 1 :

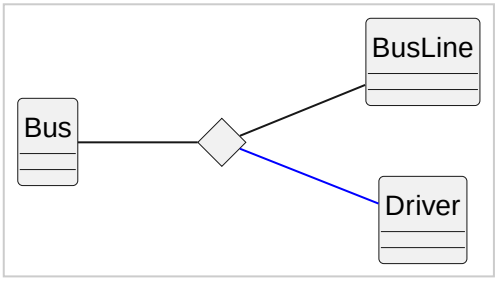


Vous remarquerez les cardinalités minimales à 1.
Il ne peut y avoir une association alors qu'il manquerait un chauffeur ou la ligne de bus ou encore le bus.

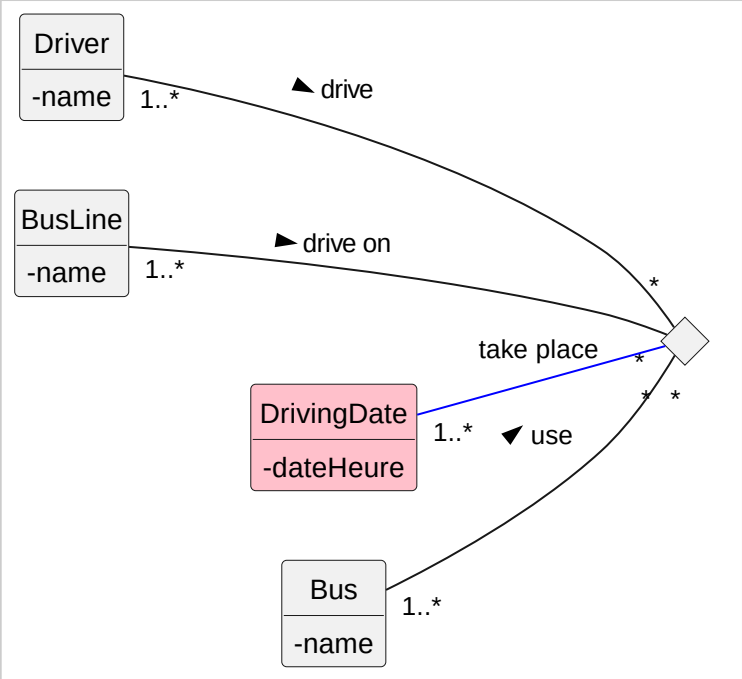
Si vous avez des difficultés à comprendre la représentation avec le losange carré, imaginez qu'une association binaire soit représentée comme ceci :



Lorsqu'une association concerne une troisième classe, il faut relier cette classe au symbol de l'association :

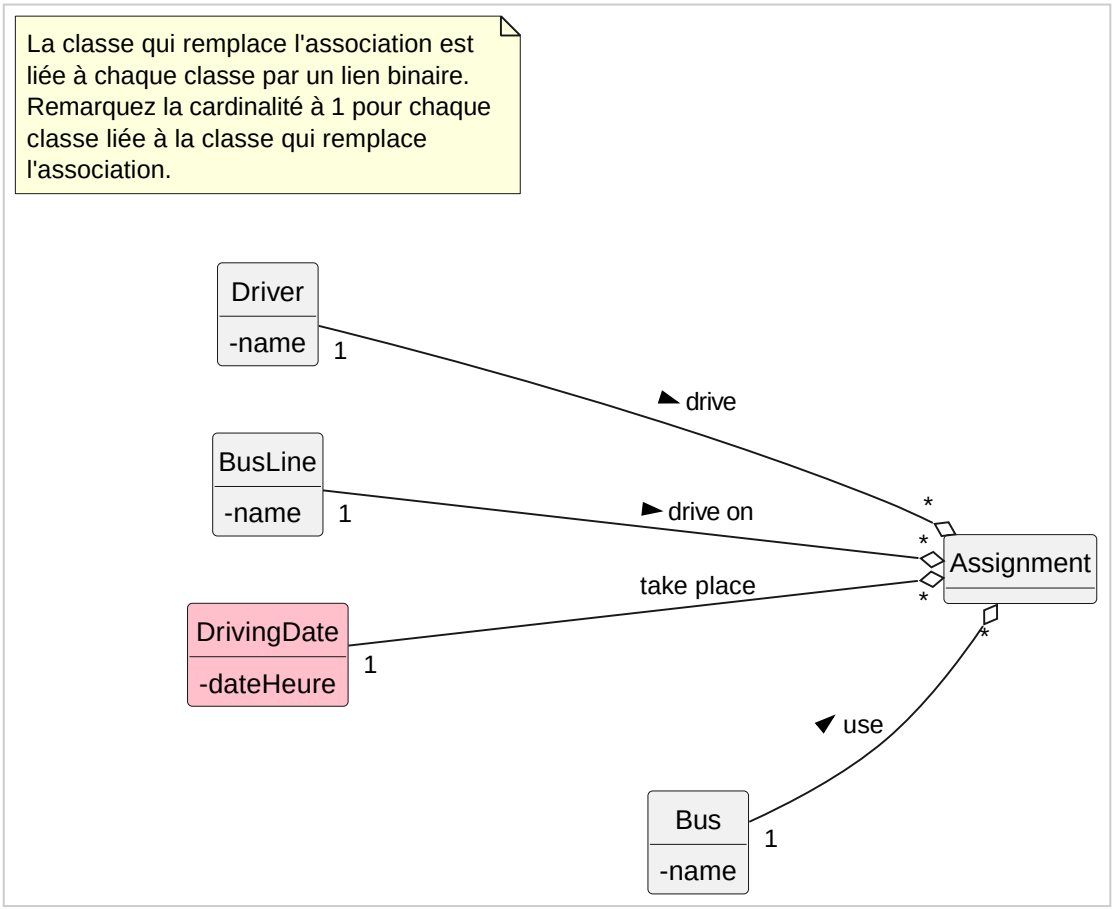


Il subsiste encore un manque. Michel sait qu'il doit conduire le bus B sur la ligne 3, mais c'est aussi le cas pour d'autres chauffeurs. Il faut donc pouvoir préciser "quand" chacun d'eux va conduire le bus B sur la ligne 3. La solution consiste à ajouter un objet date à l'association ternaire :



Grâce à cette association, il est possible de savoir qui à conduit quoi, où et quand !

L'association n-aire est difficile à interpréter et régulièrement source d'erreurs. Une fois la conceptualisation réalisée, il est préférable de remplacer ce type d'association par des associations binaires. Ainsi, l'association est remplacée par une agrégation :

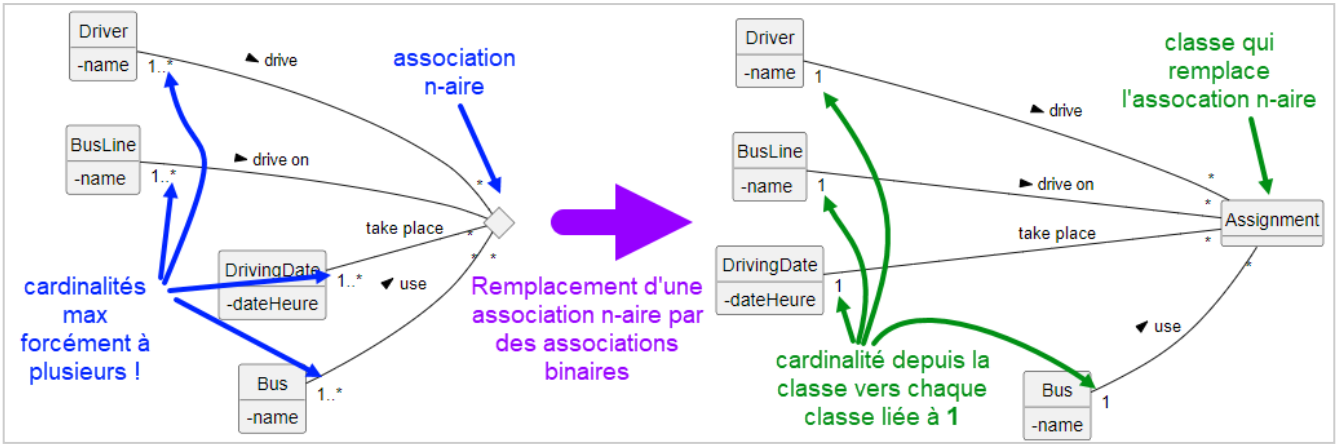


La conséquence est qu'il ne reste plus que des associations binaires que nous

savons traiter.

16.2. Implémentation d'une association n-aire

Je conseille plus que vivement de remplacer une association n-aire par une classe qui va être liée aux autres par des associations binaires (vous remarquerez que je n'ai pas utilisé le losange indiquant l'agrégation, c'est rarement utilisé dans la pratique même si c'est plus pertinent):



Les ORM tels que Doctrine nécessite d'ailleurs de passer par des associations binaires.

Une fois que vous n'avez plus que des associations binaires, il n'y a qu'à réaliser les implémentations comme nous l'avons appris jusqu'à maintenant. Il n'y a rien de nouveau à aborder sur ce point.

16.3. Exercice

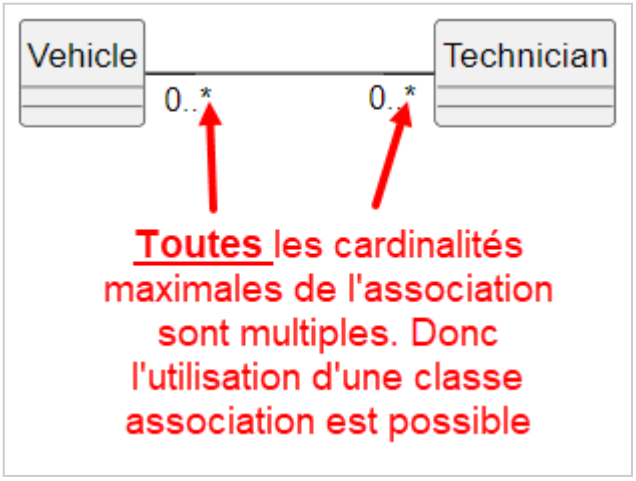
Q21) Un cinéma vous commande la conceptualisation de ses projections afin de faire développer un logiciel de gestion des séances à planifier.

Réalisez le diagramme de classes qui permet de savoir pour chaque séance, le film et la salle concernée.

17. L'association porteuse (ou classe association)

17.1. Qu'est-ce qu'une association porteuse ou classe association ?

Avant de commencer à expliquer ce qu'est une association porteuse, sachez que cela ne peut exister que si les cardinalités situées de chaque côté de l'association sont multiples.

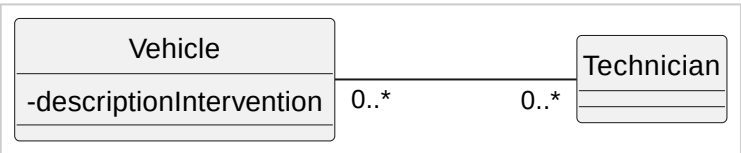


Une **classe association** permet de prévoir des attributs et/ou des méthodes qui ne concernent que le "couple" d'objets liés. C'est-à-dire qu'il n'est pas possible de rattacher un tel attribut ou une telle méthode à une des deux classes en particulier.

A partir du diagramme précédent, nous savons qu'un véhicule est réparé par 0 à plusieurs techniciens et qu'un technicien répare 0 à plusieurs véhicules.

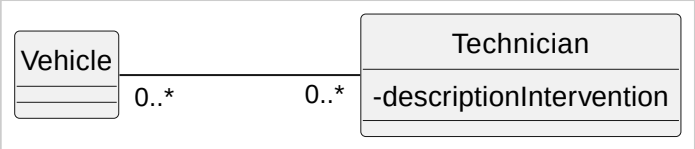
Si l'on souhaite conserver une trace de l'intervention, il faut se poser la question de la place de l'attribut à ajouter sur le diagramme.

Plaçons l'attribut `descriptionIntervention` dans la classe `Vehicle` :



Avec cette solution, on peut connaître la description d'une intervention mais on ne saura pas quel est le technicien qui l'a écrite. Si jamais, il faut par la suite lui poser des questions, on ne saura pas à qui s'adresser.

Adoptons une autre solution : l'attribut `descriptionIntervention` est placé dans la classe `Technician` :

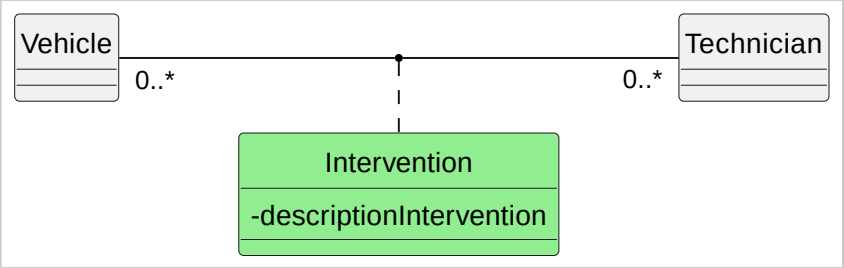


Avec cette nouvelle solution, on sait par qui est écrite une description d'intervention mais on ne sait pas quel véhicule cela concerne.

Conclusion : l'attribut ne peut être rattaché à aucune des deux classes de l'association. L'attribut est rattaché à l'association ce qui en fait une **association porteuse** (ou classe association).

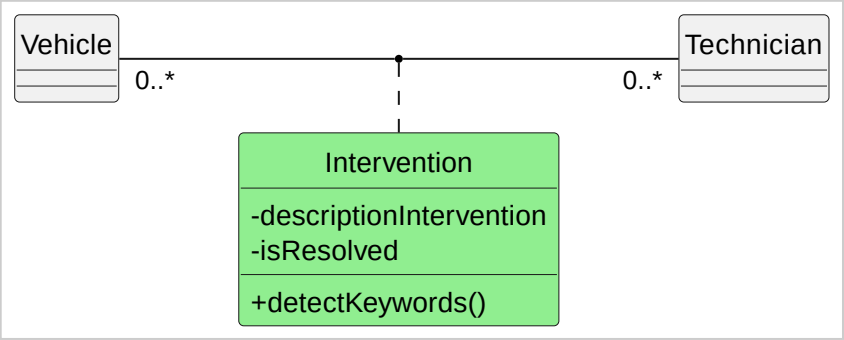
Lorsque vous vous trouvez dans une telle situation, c'est forcément que les cardinalités maximales de l'association sont multiples (sinon, c'est que vous vous êtes trompés sur les cardinalités ou que vous rattachez mal votre attribut).

La solution, c'est de créer une classe qui va contenir cet attribut et lier cette classe à l'association :



Si nous souhaitons savoir si l'intervention a résolu le problème, un nouvel attribut `isResolved` peut être ajouté. Cet attribut est à placer dans la classe association car il n'est pas logique de le lier au véhicule (quelle signification aurait cet attribut placé dans `Vehicle`. On peut faire la même remarque si on le place dans `Technician`). Si nous souhaitons connaître la présence de certains mots clés dans la description de l'intervention, l'attribut à créer ne peut être placé que dans la classe association pour les mêmes raisons.

Voici le diagramme compte tenu de ces évolutions :



17.2. Implémentation d'une classe associative

Nous savons déjà implémenter la navigabilité entre `Vehicle` et `Technician`.

Commençons par la classe `Vehicle` :

```
1 <?php
```

```php
class Vehicle
{
    /**
     * @param array|Technician[] $technicians collection d'objets de type
     *                                        Technician
     */
    public function __construct(
        private array $technicians = []
    ) {
        $this->setTechnicians($this->technicians);

    }

    //mutateurs et accesseur de la collection de techniciens

    /**
     * @param Technician $technician ajoute un item de type Technician à la
     *                               collection
     */
    public function addTechnician(Technician $technician): bool
    {
        if (!in_array($technician, $this->technicians, true)) {
            $this->technicians[] = $technician;

            return true;
        }

        return false;
    }

    /**
     * @param Technician $technician retire l'item de la collection
     */
    public function removeTechnician(Technician $technician): bool
    {
        $key = array_search($technician, $this->technicians, true);

        if ($key !== false) {
            unset($this->technicians[$key]);

            return true;
        }

        return false;
    }

    /**
     * Initialise la collection avec la collection passée en argument
     *
     * @param array $technicians collection d'objets de type Technician
     *
```

```
     * @return $this
     */
    public function setTechnicians(array $technicians): self
    {

        //mise à jour de la collection de techniciens
        foreach ($technicians as $technician) {
            $this->addTechnician($technician);
        }

        return $this;
    }
    /**
     * @return Technician[]
     */
    public function getTechnicians(): array
    {
        return $this->technicians;
    }
}
```

Poursuivons avec la classe `Technician` :

```
<?php
class Technician
{
    /**
     * @param array $vehicles tableau d'objets de type Vehicle
     */
    public function __construct(
        private array $vehicles = []
    ) {

        $this->setVehicles($vehicles);

    }

    //mutateurs et accesseurs pour la collection de Vehicle

    /**
     * @param Vehicle $vehicle ajoute un item de type Vehicle à la collection
     */
    public function addVehicle(Vehicle $vehicle): bool
    {
        if (!in_array($vehicle, $this->vehicles, true)) {
            $this->vehicles[] = $vehicle;

            return true;
        }

```

```
28          return false;
29      }
30
31      /**
32       * @param Vehicle $vehicle retire l'item de la collection
33       */
34      public function removeVehicle(Vehicle $vehicle): bool
35      {
36          $key = array_search($vehicle, $this->vehicles, true);
37
38          if ($key !== false) {
39              unset($this->vehicles[$key]);
40
41              return true;
42          }
43
44          return false;
45      }
46
47      /**
48       * Initialise la collection avec la collection passée en argument
49       *
50       * @param array $vehicles collection d'objets de type Vehicle
51       *
52       * @return $this
53       */
54      public function setVehicles(array $vehicles): self
55      {
56          foreach ($vehicles as $vehicle) {
57              $this->addVehicle($vehicle);
58          }
59
60          return $this;
61      }
62
63
64      /**
65       * @return Vehicle[]
66       */
67      public function getVehicles(): array
68      {
69          return $this->vehicles;
70      }
71
72  }
```

Maintenant, il faut désigner la classe possédante, c'est-à-dire celle qui va être responsable de la mise à jour des objets liés. Je choisis la classe `Vehicle`. C'est un choix purement arbitraire, j'aurais pu retenir la classe `Technician`.

Puisque l'on connait la classe possédante (`Vehicle`) , il faut prévoir dans celle-ci la mise à jour de l'objet lié (`Technician`).

Cela n'impacte que les méthodes `Vehicle::AddTechnician`, `Vehicle::removeTechnician` et `Vehicle::setTechnicians` :

```
1     /**
2      * @param Technician $technician ajoute un item de type Technician à la
3      *                               collection
4      */
5     public function addTechnician(Technician $technician): bool
6     {
7         if (!in_array($technician, $this->technicians, true)) {
8             $this->technicians[] = $technician;
9
10            //mise à jour de l'objet lié ①
11            $technician->addVehicle($this);
12
13
14            return true;
15        }
16
17        return false;
18    }
19    /**
20     * @param Technician $technician retire l'item de la collection
21     */
22    public function removeTechnician(Technician $technician): bool
23    {
24        $key = array_search($technician, $this->technicians, true);
25
26        if ($key !== false) {
27
28            //mise à jour de l'objet lié (on indique au technicien qu'il n'est plus
  lié à la voiture courante
29            // cette mise à jour est à faire AVANT la suppression du technicien
  sans quoi il ne sera pas possible de l'utiliser pour retirer le véhicule ②
30            $this->technicians[$key]->removeVehicle($this);
31            //suppression du technicien (à faire après avoir retiré le véhicule qui
  lui était associé
32            unset($this->technicians[$key]);
33
34            return true;
35        }
36
37        return false;
38    }
39    /**
40     * Initialise la collection avec la collection passée en argument
41     *
```

```
42      * @param array $technicians collection d'objets de type Technician
43      *
44      * @return $this
45      */
46     public function setTechnicians(array $technicians): self
47     {
48         //mise à jour des objets de la collection courante (avant son
   actualisation)
49         foreach($this->technicians as $technician){
50             $technician->removeVehicle($this); ②
51         }
52
53         //mise à jour de la collection de techniciens
54         foreach ($technicians as $technician) {
55             $this->addTechnician($technician);
56         }
57
58         return $this;
59     }
```

① Lorsque l'on associe un technicien, il faut que ce dernier soit associé au véhicule courant.

② Lorsqu'on retire un technicien du véhicule, il faut retirer le véhicule du technicien. Cela doit être fait AVANT de supprimer le technicien sans quoi nous n'aurons plus accès à ce dernier pour appeler la méthode `Technician::removeVehicle`

L'implémentation de l'association bidirectionnelle est terminée. Il faut maintenant gérer la classe association.

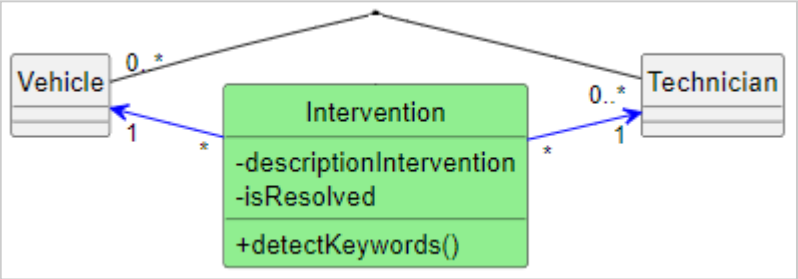
La classe association `Intervention` est avant tout une classe comme une autre, simple à implémenter :

```
1  <?php
2  class Intervention
3  {
4      /**
5       * @param string $descriptionIntervention
6       * @param bool   $isResolved
7       */
8      public function __construct(
9          private string $descriptionIntervention,
10         private bool $isResolved
11
12     )
13     {
14     }
15
16
17     //mutateurs et accesseurs des attributs Intervention::descriptionIntervention
   et Intervention::isResolved
18
```

```
19 }
```

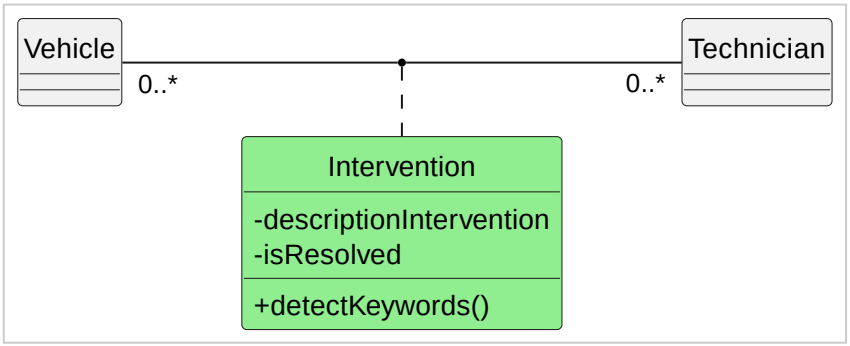
Maintenant, il faut prendre en compte le fait qu'une classe association doit connaître chaque instance des extrémités de l'association sur laquelle elle porte. Cela signifie que `Intervention` peut naviguer vers `Vehicle` et vers `Technician`. De plus, comme une intervention ne concerne qu'un véhicule et un seul technicien simultanément, les cardinalités de `Intervention` vers `Vehicle` et `Technician` sont à 1.

Cette interprétation peut être modélisée ainsi :

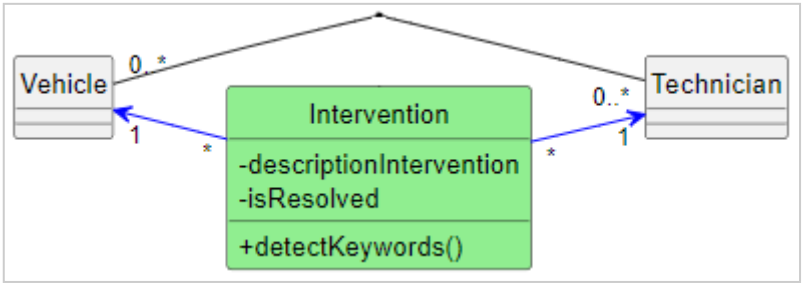


Cette représentation n'est pas valide, elle représente seulement la réflexion que l'on vient de mener. Une classe association, c'est considérer qu'il y a un lien entre cette classe et chaque classe de l'association avec une cardinalité à 1.

Pour faire simple, quand vous voyez cela :



Vous devez implémenter cela :



Riche de ces informations, il est facile de compléter le code de la classe en implémentant le lien entre `Intervention` et `Vehicle` :

```
1 <?php
2 class Intervention
3 {
```

```
4       /**
5        * @param string $descriptionIntervention
6        * @param bool   $isResolved
7        */
8       public function __construct(
9           private string $descriptionIntervention,
10          private bool $isResolved
11          ,
12          private Vehicle $vehicle
13
14      )
15      {
16      }
17
18
19      //accesseur pour le véhicule
20      /**
21       * @return Vehicle
22       */
23      public function getVehicle(): Vehicle
24      {
25          return $this->vehicle;
26      }
27
28      //mutateur pour le véhicule (on pourrait ne pas avoir de mutateur car le
    véhicule est affecté à l'intervention au moment de l'instanciation de cette
    dernière). Laisse le mutateur permet de modifier le véhicule associé (suite à une
    erreur de saisi par exemple)
29      /**
30       * @param Vehicle $vehicle
31       */
32      public function setVehicle(Vehicle $vehicle): void
33      {
34          $this->vehicle = $vehicle;
35
36      }
37
38      //mutateurs et accesseurs des attributs Intervention::descriptionIntervention
    et Intervention::isResolved
39
40  }
```

Il reste encore à implémenter le lien entre `Intervention` et `Technician` :

```
1  <?php
2  class Intervention
3  {
4      /**
5       * @param string $descriptionIntervention
6       * @param bool   $isResolved
```

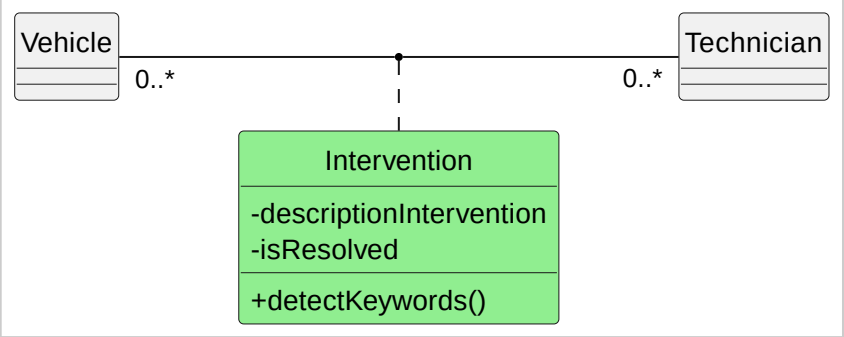
```php
     */
    public function __construct(
        private string $descriptionIntervention,
        private bool $isResolved
        ,
        private Vehicle $vehicle

        ,
        private Technician $technician
    )
    {
    }


    //accesseur pour le véhicule
    /**
     * @return Vehicle
     */
    public function getVehicle(): Vehicle
    {
        return $this->vehicle;
    }

    //mutateur pour le véhicule (on pourrait ne pas avoir de mutateur car le véhicule est affecté à l'intervention au moment de l'instanciation de cette dernière). Laisse le mutateur permet de modifier le véhicule associé (suite à une erreur de saisi par exemple)
    /**
     * @param Vehicle $vehicle
     */
    public function setVehicle(Vehicle $vehicle): void
    {
        $this->vehicle = $vehicle;

    }

    /**
     * @return Technician
     */
    public function getTechnician(): Technician
    {
        return $this->technician;
    }

    /**
     * @param Technician $technician
     */
    public function setTechnician(Technician $technician): void
    {
        $this->technician = $technician;
    }
```

```
55     //mutateurs et accesseurs des attributs Intervention::descriptionIntervention
   et Intervention::isResolved
56
57 }
```

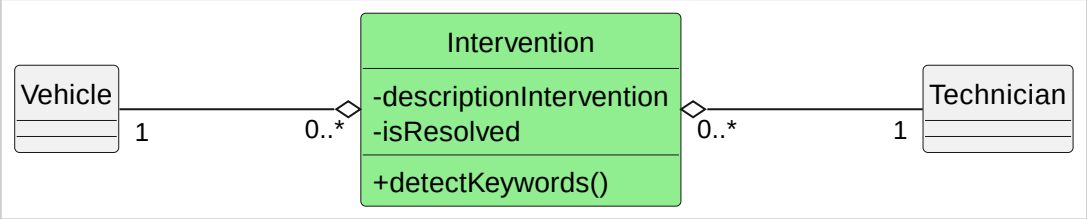
Désormais, vous savez implémenter une classe association. Cependant, cette implémentation n'est généralement pas la solution retenue.

17.3. Dans la pratique, on simplifie les choses

Dans la pratique, la modélisation suivante :



est simplifiée comme ceci :



La classe association est transformée en une agrégation (et encore, la plupart du temps, l'agrégation n'est même pas représentée !). Il n'y a plus d'association directe entre `Vehicle` et `Technician` .

Cette solution offre plusieurs avantages :

- Il est possible de naviguer de `Vehicule` vers `Intervention` et vice-versa.
- Il est possible de naviguer de `Technician` vers `Intervention` et vice-versa.
- Il n'y a qu'une classe possédante, c'est `Intervention`. C'est donc elle qui va mettre à jour les liens bidirectionnels.
- Les ORM tel que Doctrine nécessitent de travailler avec des associations binaires. Cette solution est donc compatible avec leur usage.

Voici l'implémentation qui en découle (en partant du code que nous avons écrit jusque là):

- La classe `Vehicle` n'a plus d'attribut `$technicians`. Par conséquent, les mutateurs et l'accesseur associés sont à retirer. Il n'y a plus les méthodes `addTechnician`, `removeTechnician`, `setTechnicians` et `getTechnicians` :

```php
<?php
class Vehicle
{
    /**
     * @param array $intervention |Intervention[] tableau d'objets de type
     *                            Intervention
     */
    public function __construct(
        private array $interventions = []

    ) {
    }

    /**
     * @param Intervention $intervention ajoute un item de type Intervention à
     *                                   la collection
     */
    public function addIntervention(Intervention $intervention): bool
    {
        if (!in_array($intervention, $this->interventions, true)) {
            $this->interventions[] = $intervention;

            return true;
        }

        return false;
    }

    /**
     * @param Intervention $intervention retire l'item de la collection
     */
    public function removeIntervention(Intervention $intervention): bool
    {
        $key = array_search($intervention, $this->interventions, true);

        if ($key !== false) {
            unset($this->interventions[$key]);

            return true;
        }

        return false;
    }

    /**
     * @return Intervention[]
     */
    public function getInterventions(): array
    {
        return $this->interventions;
```

```
51     }
52
53 }
```

- La classe `Technician` n'a plus d'attribut `$vehicles` ainsi que les méthodes associées à celui-ci (`addVehicle`, `removeVehicle`, `setVehicles` et `getVehicles`) :

```php
class Technician
{
    /**
     * @param array $intervention |Intervention[] tableau d'objets de type
     *                            Intervention
     */
    public function __construct(
        private array $interventions = []
    ) {

    }

    /**
     * @param Intervention $intervention ajoute un item de type Intervention à
     *                                   la collection
     */
    public function addIntervention(Intervention $intervention): bool
    {
        if (!in_array($intervention, $this->interventions, true)) {
            $this->interventions[] = $intervention;

            return true;
        }

        return false;
    }

    /**
     * @param Intervention $intervention retire l'item de la collection
     */
    public function removeIntervention(Intervention $intervention): bool
    {
        $key = array_search($intervention, $this->interventions, true);

        if ($key !== false) {
            unset($this->interventions[$key]);

            return true;
        }

        return false;
    }

```

```
44      /**
45       * @return Intervention[]
46       */
47      public function getInterventions(): array
48      {
49          return $this->interventions;
50      }
51
52      //méthode à ajouter dans la classe Technician
53      public function getVehicles()
54      {
55          $vehicles = [];
56          /** @var Intervention $intervention */
57          foreach ($this->interventions as $intervention) {
58              $vehicles[] = $intervention->getVehicle();
59          }
60
61          return $vehicles;
62
63      }
64  }
```

- La classe `Intervention` est responsable de la mise à jour des objets qui la compose (puisqu'il s'agit d'une "association")

```
1  class Intervention
2  {
3      /**
4       * @param string     $descriptionIntervention
5       * @param bool       $isResolved
6       * @param Vehicle    $vehicle
7       * @param Technician $technician
8       */
9      public function __construct(
10         private string $descriptionIntervention,
11         private bool $isResolved,
12         private Vehicle $vehicle,
13         private Technician $technician
14     ) {
15     }
16
17
18     //accesseur pour le véhicule
19
20     /**
21      * @return Vehicle
22      */
23     public function getVehicle(): Vehicle
24     {
25         return $this->vehicle;
```

```
26      }
27
28      //mutateur pour le véhicule (on pourrait ne pas avoir de mutateur car le
    véhicule est affecté à l'intervention au moment de l'instanciation de cette
    dernière). Laisse le mutateur permet de modifier le véhicule associé (suite à une
    erreur de saisi par exemple)
29
30      /**
31       * @param Vehicle $vehicle
32       */
33      public function setVehicle(Vehicle $vehicle): void
34      {
35          $this->vehicle = $vehicle;
36
37          //mise à jour de l'objet lié pour la navigabilité bidirectionnelle
38          $vehicle->addIntervention($this);
39      }
40
41      /**
42       * @return Technician
43       */
44      public function getTechnician(): Technician
45      {
46          return $this->technician;
47      }
48
49      /**
50       * @param Technician $technician
51       */
52      public function setTechnician(Technician $technician): void
53      {
54          $this->technician = $technician;
55
56          //mise à jour de l'objet lié pour la navigabilité bidirectionnelle
57          $technician->addIntervention($this);
58      }
59
60      //mutateurs et accesseurs des autres attributs
61      //...
62
63  }
```

Q22) Faites évoluer le code précédent de façon à ce qu'à partir d'une instance de `Vehicle`, il soit possible de récupérer les techniciens qui sont intervenus dessus.

L'objectif est de pouvoir faire la chose suivante :

```
1 $liste = $vehicleA->getTechnicians(); //tableau contenant tous les techniciens
```

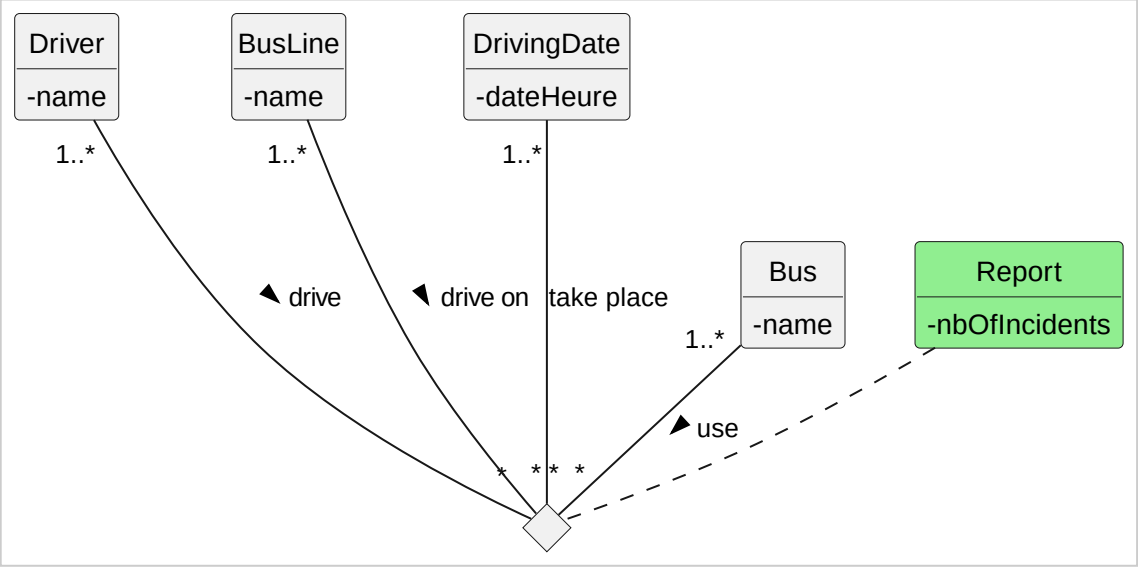
```
qui sont intervenus sur le véhicule A
```

Q23) Faites évoluer le code de façon à ce qu'à partir d'une instance de `Technician`, il soit possible de récupérer les véhicules sur lesquels est intervenu un technicien

17.4. Que faire si une classe association porte sur une association n-aire ?

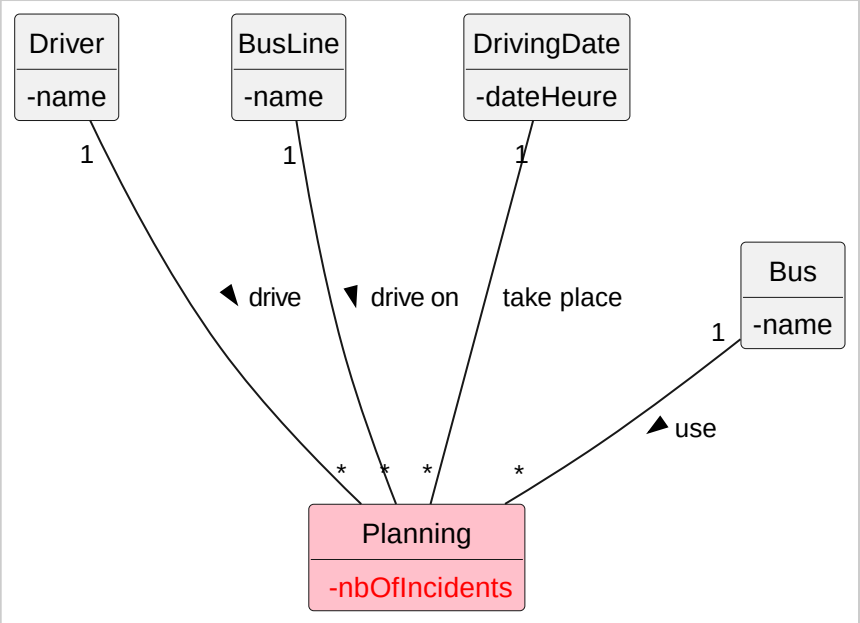
Lorsque votre analyse aboutit à une associations n-aire avec une classe association, cela peut être déroutant.

Prenons cette modélisation en exemple :



Comme je l'ai indiqué dans la partie qui aborde la simplification d'une classe association, la classe association doit être liée à chaque classe de l'association par une cardinalité à 1 et l'association doit être supprimée.

Cela donne le résultat suivant :



Vous n'avez maintenant que des liens bidirectionnels. L'implémentation d'une telle modélisation n'a plus de secret pour vous dorénavant !

18. La relation de dépendance

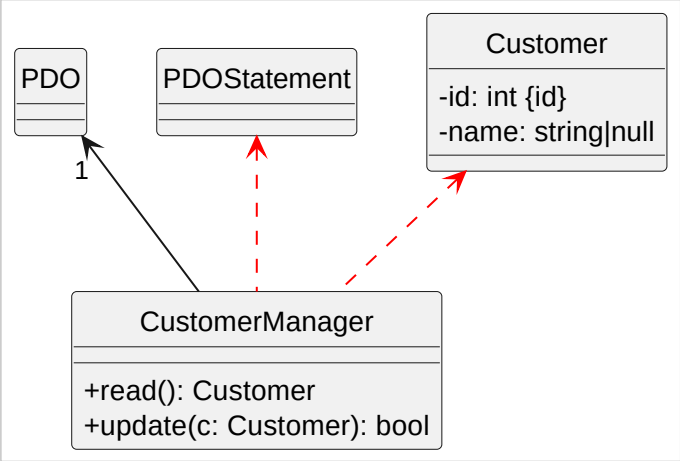
18.1. Qu'est-ce qu'une dépendance ?

Il y a **dépendance** entre deux objets lorsqu'un objet A utilise un objet B sans le « stocker » dans un de ses attributs*. Il n'y a pas de navigabilité vers cet objet B.

Dans le cas d'une dépendance, l'instance utilisée ne l'est que temporairement (contrairement à une association classique). Il n'y a donc pas besoin de stocker celle-ci dans un attribut. Le lien aux objets utilisés ne sont pas permanents.

Nous allons illustrer ces concepts au travers d'un gestionnaire d'entité client `CustomManager`. Ce gestionnaire est responsable de la récupération d'un objet en bdd et de sa mise à jour.

Voici sa modélisation :



La classe `CustomerManager` manipule trois objets :

- **un objet** `PDO` qui correspond à la connexion à la base de données.

  Comme cet objet va être utilisé dans différentes méthodes, il est nécessaire de le stocker durablement dans la classe utilisatrice. Le lien est permanent, nous sommes dans le cadre d'une association (d'où l'association unidirectionnelle).

  (Pour rappel, un objet `PDO` permet d'accéder aux méthodes `prepare`, `query` qui retourne une objet de type `PDOStatement`.)

- **un objet** `PDOStatement`

  (Pour rappel, un objet `PDOStatement` permet d'utiliser des méthodes telles que `bindParam`, `execute`, `fetch`, `rowCount`,...)

  Cet objet est différent en fonction de la requête exécutée. Il n'est donc pas pertinent de le stocker dans un attribut car il ne sera pas utilisé en dehors de la méthode dans laquelle il a été créé. Le lien entre `PDOStatement` et `CustomerManager` n'est pas durable.

- **un objet** `Customer`. Cet objet correspond au client sur lequel porte la requête. Un client récupéré

via la méthode `read()` n'est pas forcément le client qui sera mis à jour. Il n'est pas forcément le client manipulé par les autres méthodes (update, delete, etc). Il n'y a donc pas d'intérêt à stocker le client dans un attribut spécifique. Le lien entre `CustomerManager` et `Customer` est éphémère.

Compte tenu du caractère non durable de leur lien avec la classe `CustomerManager`, les classes `PDOStatement` et `Customer` sont appelées des **dépendances**.

Sur le diagramme de classes, une dépendance est pointée par une flèche en pointillé.

18.2. Implémentation d'une dépendance

Je vais implémenter le diagramme précédent.

Les classes `PDO` et `PDOStatement` sont des classes déjà incluses dans PHP. Par contre, il faut implémenter la classe `Customer` :

```
class Customer
{
    private ?string $name;
    private int $id;

    public function getId(): int
    {
        return $this->id;
    }

    public function getName(): ?string
    {
        return $this->name;
    }

    public function setName(?string $name): void
    {
        $this->name = $name;
    }


}
```

Puis la classe `CustomerManager` :

```
class CustomerManager
{

    //l'objet PDO utilisé dans les méthodes read et update est le même. Le lien est
"fort" et durable dans le temps. C'est pouquoi il est stocké dans un attribut. (c'est
donc une association)
    private PDO $pdo;
```

```
    public function __construct()
    {
        $this->pdo = new PDO(
            'mysql:host:localhost;dbname=cinema',
            'gandahlf',
            'l3precieuX#'
        );
    }


    //récupération d'un client (ou rien s'il n'existe pas en bdd)
    public function read(int $id): ?Customer
    {
        $pdoStatement = $this->pdo->prepare(
            'SELECT * FROM customer WHERE id = :id'
        ); ①

        //liaison du paramètre nommé
        $pdoStatement->bindValue('id', $id, PDO::PARAM_INT);

        //exécution de la requête (il faudrait tester le retour dans la réalité pour
voir si tout est ok)
        $pdoStatement->execute(); ②

        //on récupère le client recherché (il faudrait tester le retour pour voir si
tout est ok
        return $pdoStatement->fetchObject('Customer');
    }

    //mise à jour d'une client
    public function update(Customer $customer): bool
    {

        $pdoStatement = $this->pdo->prepare('UPDATE contact set name = :name WHERE id
= :id'); ①

        $pdoStatement->bindValue('name', $customer->getName(), PDO::PARAM_STR); ③
        $pdoStatement->bindValue('id', $customer->getId(), PDO::PARAM_INT); ③

        return $pdoStatement->execute();

    }

}
```

① l'objet de type `PDOStatement` est stocké dans une variable locale à la méthode mais pas dans un attribut d'objet (cela marque le lien non durable entre l'objet utilisateur et l'objet utilisé)

② L'objet `PDOStatement` n'a d'intérêt que dans le contexte de la méthode `read()`. La requête qu'il contient ne concerne que le client à retourner. Cette requête n'aurait pas d'utilité dans une autre méthode (update, delete, create par exemple).

③ L'objet `Customer` n'a d'intérêt que pour lier les paramètres nommés utilisés dans la requête préparée. Le client utilisé n'est pas forcément utilisé dans les autres méthodes.

Pour faire simple : une dépendance traduit un lien non durable entre deux objets.

19. La relation d'héritage (classe mère, classe fille)

19.1. Comprendre et modéliser la notion d'héritage

La **relation d'héritage** n'exprime plus l'idée qu'un objet A est lié à un objet B mais qu'**un objet A est un objet B**.

Imaginons que nous devions gérer des fichiers dans une application de gestion électronique des documents. La première chose est de créer la classe `File` :

Les types de fichiers à gérer sont les suivants :

- fichier texte avec un nom et une taille en Mo.
- fichier image avec un nom, une taille en Mo et des dimensions en pixels. Une image peut être redimensionnée.
- fichier vidéo avec un nom, une taille, une durée. Une vidéo peut être compressée.

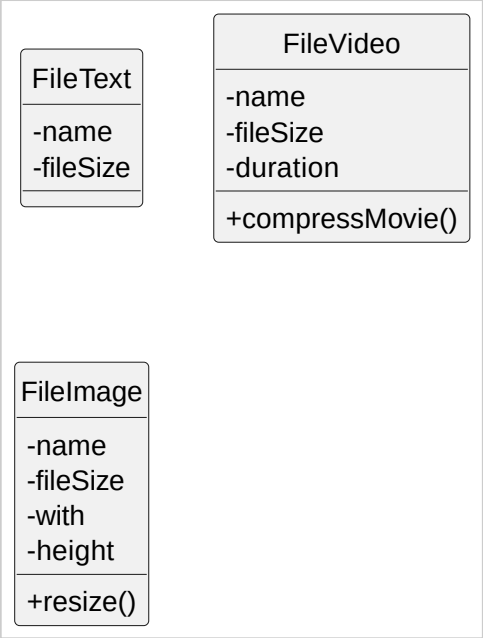
Nous pouvons prendre en compte ces informations dans le diagramme suivant :

Cette classe soulève plusieurs remarques :

- Certains attributs et ou certaines méthodes sont inutiles en fonction du type de fichier. Un fichier texte ne peut pas être redimensionné ou subir une compression vidéo, une image n'a pas de durée, etc.
- Dès qu'il faut ajouter une nouvelle caractéristique (par exemple, le format d'affichage d'une image : portrait ou paysage), il faut faire évoluer la classe `File`.
  Elle va devenir de plus en plus "grosse" et poser un problème de maintenance évolutive (et peut être devenir un God object).

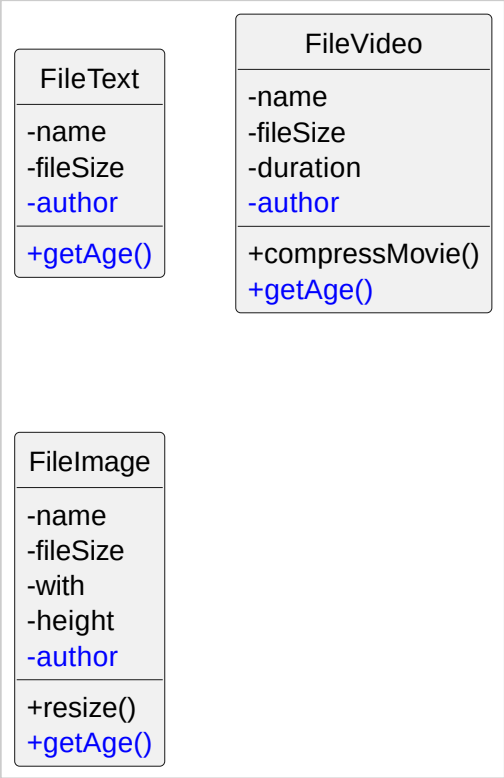
Le principe général en programmation est de "diviser pour régner". Dans notre cas, cela signifie qu'il faut que chaque type de fichier ait sa propre classe :

Voici comment illustrer ce principe :



Maintenant, s'il faut ajouter le type d'affichage (portrait ou paysage) d'une image, il est facile de cibler la classe à faire évoluer sans "polluer" les autres avec des attributs ou méthodes qui leur seraient inutiles.

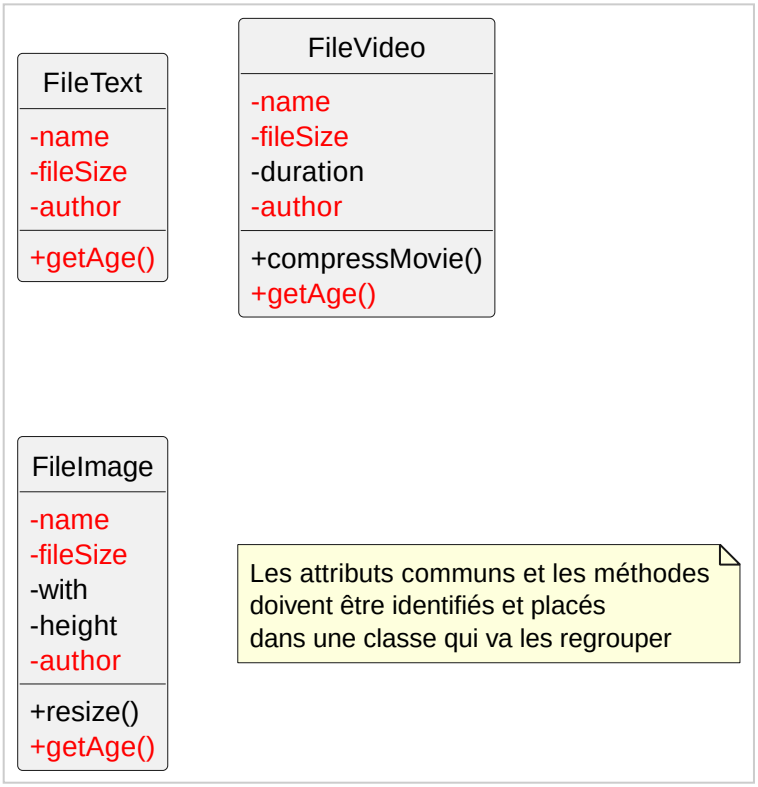
Par contre, si l'on souhaite ajouter l'auteur du fichier, il faut le faire dans les 3 classes. S'il faut ajouter une méthode qui retourne l'âge du fichier, il faut le faire dans chaque classe. Si l'implémentation de cette méthode évolue, il faudra faire la mise à jour dans les 3 classes. Tout cela va vite devenir difficile à maintenir. Toutefois, si l'on passe outre ces remarques, cela donne le diagramme suivant :



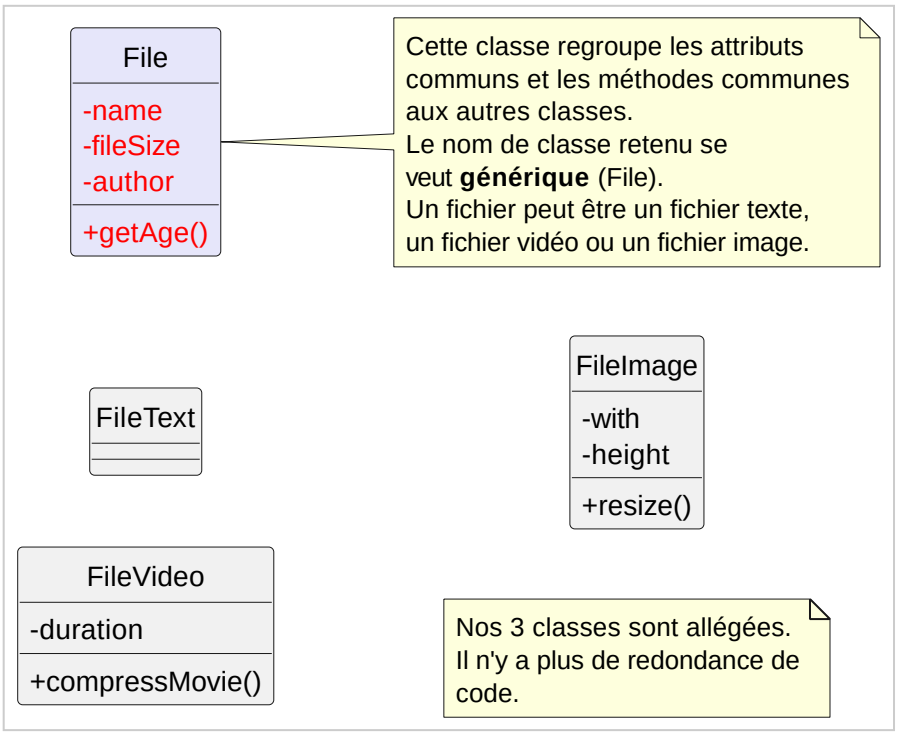
Le problème de cette approche est que nous nous répétons. Il y a un autre grand principe en programmation, c'est le principe DRY (don't repeat yourself ⇒ ne vous répétez pas). Dans le cas présent, ce principe n'est clairement pas respecté.

L'idéal serait de mettre dans une classe les attributs et les méthodes communes aux 3 classes et de laisser ce qui est spécifique dans les classes précédemment créées.

Pour cela il faut repérer tous les attributs communs et les méthodes communes :



Après regroupement :



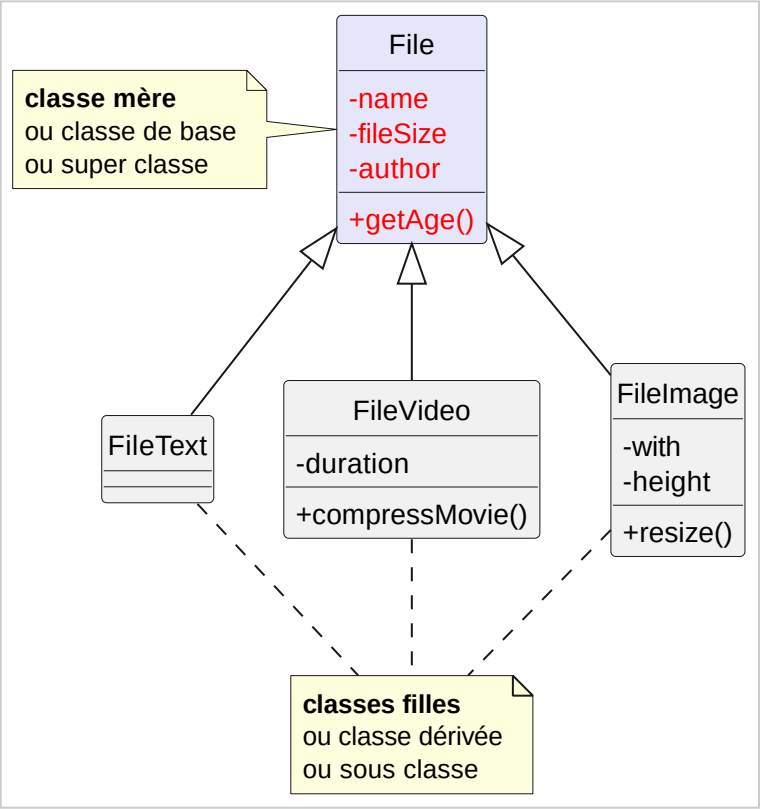
A ce stade, comment savoir que nos 3 classes possèdent les attributs et méthodes de la classe `File` ?

C'est là qu'intervient la **relation d'héritage**.

UML nous permet de dire qu'un fichier texte **est un** fichier, qu'un fichier vidéo **est un** fichier et

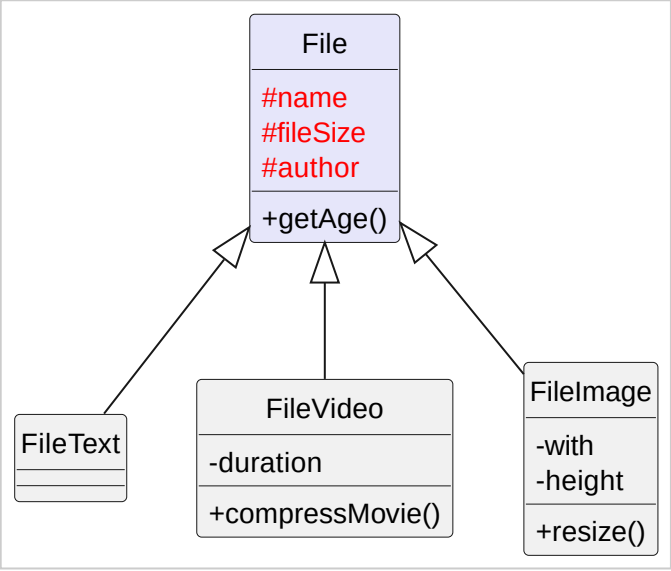
enfin qu'un fichier image **est un** fichier.

Sur le diagramme de classes, cela se traduit par une **flèche du côté de la classe qui regroupe les attributs communs et méthodes communes** :



Il y a un problème majeur dans cette modélisation. Les attributs sont déclarés comme privés. Ils ne peuvent être utilisés à l'extérieur de la classe dans laquelle ils sont déclarés. Pour permettre leur utilisation à l'extérieur, il faut les déclarer comme protégés avec le caractère #. Attention, il ne faut pas les déclarer avec une visibilité publique sinon le principe d'encapsulation n'est plus respecté.

Voici le diagramme corrigé :



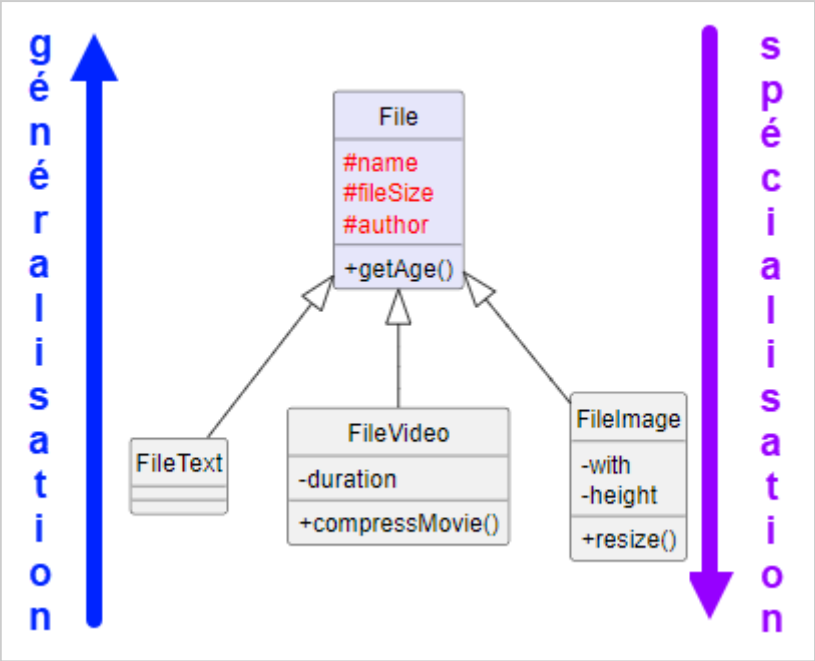
La classe qui regroupe les attributs communs et méthodes communes est appelée la **classe mère** (ou **super classe** ou **classe de base**). La ou les classes qui

contiennent des attributs spécifiques / spécialités sont des **classes filles** (ou **classes dérivées** ou **sous classes**)

Il est dit que **les classes filles héritent de la classe mère**. C'est-à-dire que les classes filles peuvent utiliser les attributs et méthodes de la classe mère si leur visibilité est au moins protégée.

Lorsque l'on remonte d'une classe fille vers la classe mère (donc que l'on factorise les attributs et les méthodes en commun), on parle de **généralisation**.

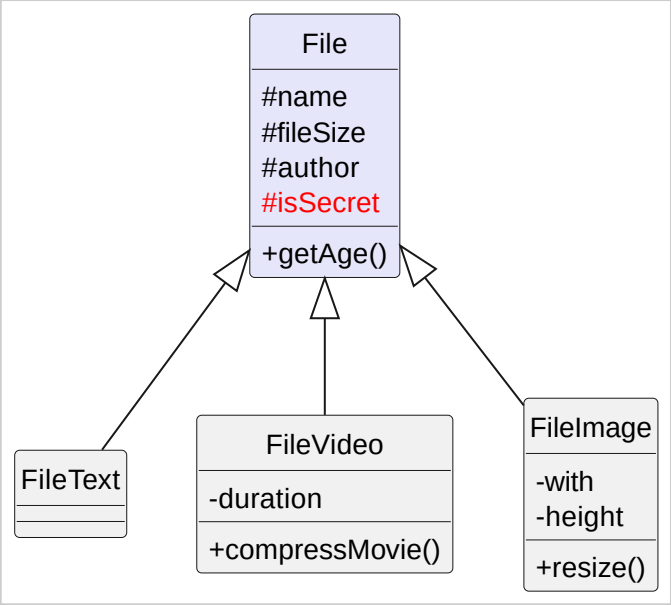
Lorsque l'on descend de la classe mère vers les classes filles (donc lorsque l'on spécialise certains attributs ou méthodes qui ne concernent qu'une classe), on parle de **spécialisation**.



Le gros avantage de l'héritage est de pouvoir faire évoluer rapidement une application.

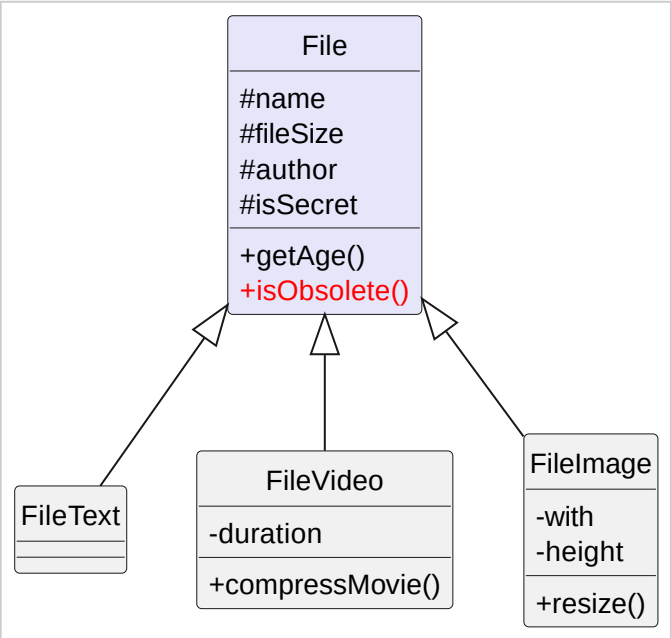
Imaginons que nous ayons besoin de savoir si un fichier est top secret ou pas. Cette caractéristique concerne tous les fichiers. Il n'y a qu'à ajouter cet attribut dans la classe mère et le travail est terminé !

Comme ceci :



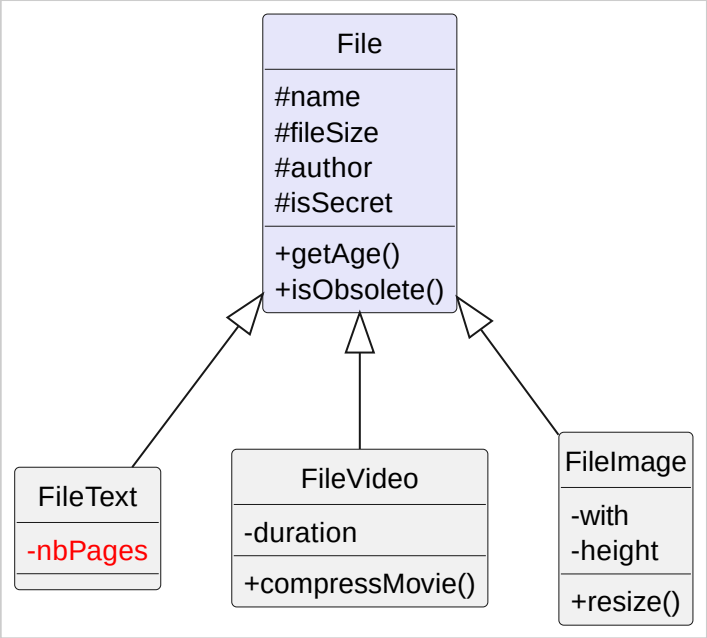
Puisque l'attribut ajoutée l'est dans la classe mère, cela signifie que l'on a fait de la **généralisation**.

Si nous devons ajouter une méthode qui indique que le fichier est peut être obsolète car il a plus de 12 mois, l'ajout d'une méthode `isObsolete()` dans la classe mère suffit pour qu'elle soit utilisable dans les classes filles :



Puisque la méthode ajoutée l'est dans la classe mère, cela signifie que l'on a fait de la **généralisation**.

Si l'on doit ajouter la possibilité qu'un fichier retourne un nombre de pages, cela ne concerne que les fichiers texte. La classe mère n'est pas impactée, seule la classe `FileText` est concernée :



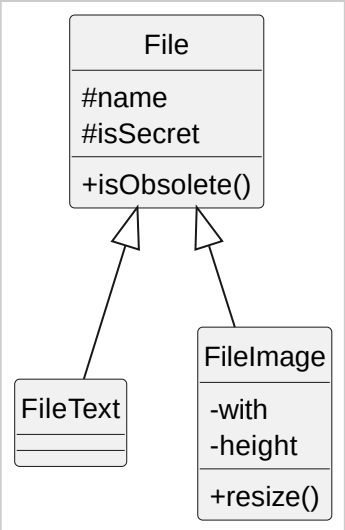
Puisque l'attribut ajouté l'est dans la classe fille, cela signifie que l'on a fait de la **spécialisation**.

19.2. Implémentation de la relation d'héritage

L'implémentation de l'héritage est très simple :

- Il faut implémenter les attributs communs et les méthodes communes aux classes filles (avec une visibilité protégée) dans la classe mère.
- Il faut implémenter chaque classe fille avec leurs attributs spécialisés et méthodes spécialisées.
- Il faut indiquer pour chaque classe fille le nom de la classe mère dont elle hérite

Je simplifie le diagramme pour avoir moins de code à implémenter :



Commençons par la classe mère `File`:

```
class File
{
```

```
    //déclaration des attributs communs
    public function __construct(
        protected string $name,
        protected bool $isSecret = false
    ) {
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function setName(string $name): void
    {
        $this->name = $name;
    }

    public function isSecret(): bool
    {
        return $this->isSecret;
    }

    public function setIsSecret(bool $isSecret): void
    {
        $this->isSecret = $isSecret;
    }


    // méthodes commune
    public function isObsolete(): bool
    {
        //code qui renvoie true si l'age du document est > à 12 mois
        return false; //on décide de renvoyer false de façon arbitraire
    }

}
```

Voici le code de la sous classe `FileText` :

```
class FileText extends File ①
{

    //la classe FileText n'a aucun attribut spécifique ni aucune méthode spécifiques

    //dans certains langages (en C# par exemple), il faut appeler le constructeur
parent depuis la classe fille. En PHP, c'est implicite. ②

    //elle hérite du code de la classe mère (c'est comme si le code de la classe mère
était copié à l'intérieur de la classe fille sauf qu'il n'est pas nécessaire de le
copier !)
```

```
}
```

① Bien indiqué que la classe hérite de `File` via le mot clé `extends` (pour PHP)

② Bien lire l'information à propos du constructeur. Il est inutile de le prévoir car celui de la classe mère sera appelé automatiquement.

Et enfin le code de la sous classe `FileImage` :

```
class FileImage extends File ①
{

    //on n'ajoute que les attributs spécialisés
    private int $width;
    private int $height;

    public function getWidth(): int
    {
        return $this->width;
    }

    public function setWidth(int $width): void
    {
        $this->width = $width;
    }

    public function getHeight(): int
    {
        return $this->height;
    }

    public function setHeight(int $height): void
    {
        $this->height = $height;
    }

    //ajout de la méthode spécialisée
    public function resize(): bool
    {
        //ici du code qui redimensionne l'image
        return true; //on considère que tout est ok
    }
}
```

① Bien indiqué que la classe hérite de `File` via le mot clé `extends` (pour PHP)

Pour bien, comprendre, nous allons créer un fichier texte et un fichier image. Nous allons ensuite appeler soit des méthodes issues de la classe mère depuis une classe fille, soit appeler des méthodes

spécialisées.

```
//création d'un fichier texte
$fileText = new FileText("monFichierDeDonnees", false);
//nom du fichier (utilisation d'une méthode de la classe mère)
$fileName = $fileText->getName();
//le fichier texte est-il obsolète ?  (idem)
$fileTextIsObsolete = $fileText->isObsolete();

//création d'un fichier image
$fileImage = new FileImage("imageDuMarcheurBlanc", false);

//affectation des dimensions de l'image (utilisation des méthodes de la classe fille)
$fileImage->setHeight(800);
$fileImage->setWidth(600);

//l'image est-elle secrète ? (utilisation d'une méthode de la classe mère)
$imageIsSecret = $fileImage->isSecret();
//on redimensionne l'image (utilisation d'une méthode de la classe mère)
$resizeIsOk = $fileImage->resize();
```

L'autre avantage de l'héritage, c'est que si un nouveau type de fichier qui n'est pas géré par l'application doit être manipulé, on peut le faire grâce à la classe `File` car elle est générique (n'importe quel type de fichier est … un fichier).

Si nous devons gérer un fichier de type exécutable, aucune des sous classes ne correspond. Soit on crée un nouveau sous-type (mais ce doit être une volonté clairement établie), soit on utilise le type `File` qui convient à tous les types de fichiers :

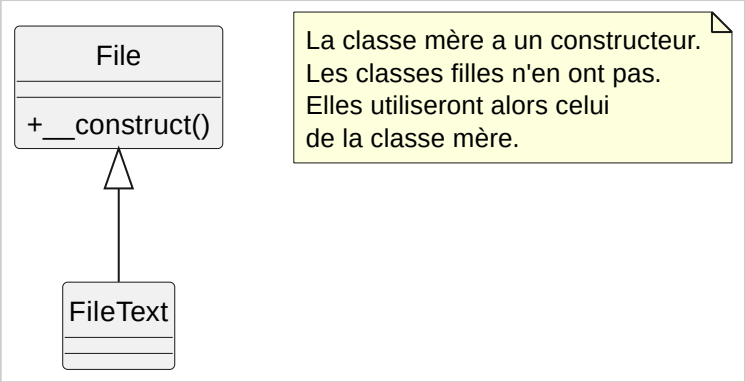
```
//Création d'un fichier exécutable (qui est avant tout un fichier)
$executableFile = new File("unVirus", true);

$fileExeAge = $executableFile->getAge();
$fileExeIsObsolete = $executableFile->isObsolete();
```

Nous voyons tout l'intérêt de pouvoir utiliser la classe mère pour gérer des sous types qui ne sont pas pris en charge par l'application.

19.3. Point technique (en PHP)

Si vous avez compris que depuis une instance d'une classe fille, il est possible d'utiliser un membre de la classe mère (attribut ou méthode), il en va de même pour le constructeur (puisque c'est aussi une méthode).

Prenons le diagramme simplifié suivant :



L'implémentation du diagramme donne le code suivant :

```
1 class File
2 {
3     public function __construct()
4     {
5         echo "Je suis du texte dans le constructeur de la classe File.\n";
6     }
7 }
8
9 class FileText extends File
10 {
11
12 }
```

Et voici son utilisation et son rendu :

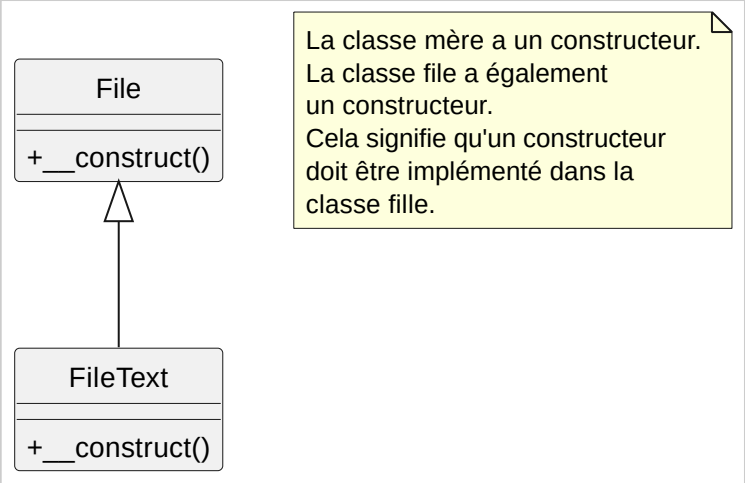
```
1 $file = new File();
2 $fileText = new FileText();
```

La sortie :

```
Je suis du texte dans le constructeur de la classe File.
Je suis du texte dans le constructeur de la classe File.
```

Il apparaît clairement que le constructeur parent (celui de la classe mère) a été utilisé par la classe fille. Elle en a hérité.

Maintenant, ajoutons un constructeur dans la classe fille :




```php
class File
{
    public function __construct()
    {
        echo "Je suis du texte dans le constructeur de la classe File.\n";
    }
}

class FileText extends File
{

    public function __construct()
    {
        echo "Appel du constructeur de la classe FileText.\n";
    }
}
```

Procédons au même appel que tout à l'heure :

```php
$file = new File();
$fileText = new FileText();
```

La sortie :

```
Je suis du texte dans le constructeur de la classe File.
Appel du constructeur de la classe FileText.
```

Ce comportement est utile lorsque la classe fille doit prévoir un comportement différent de celui de la classe mère tout en utilisant le même nom de méthode.

Il est également possible de profiter du comportement de la classe mère et d'ajouter celui de la classe fille :

```php
class File
```

```php
{
    public function __construct()
    {
        echo "Je suis du texte dans le constructeur de la classe File.\n";
    }
}

class FileText extends File
{

    public function __construct()
    {
        echo "Appel du constructeur de la classe FileText.\n";
        //appel du constructeur parent
        parent::__construct(); ①
    }
}
```

① le constructeur parent est appelé depuis le constructeur de la classe fille.

Notre exemple utilise le constructeur mais les remarques sont les mêmes pour n'importe quelle méthode. Il est possible de surcharger une méthode dans la classe fille pour remplacer celle de la classe mère. Il est possible de faire appel à la méthode de la classe mère (via `parent::nomMethode`) et de la "compléter" depuis la méthode fille.

Lorsqu'une méthode "fille" surcharge une méthode mère, il est indispensable de respecter le principe de substitution de Liskov. Pour faire simple, ce principe préconise que la signature de la méthode "fille" qui surcharge la méthode "mère" soit compatible. Vous pouvez lire la documentation au sujet des règles de compatibilité de signature. La signature d'une méthode est défini par le type de ses paramètres, leur nombre et le type de son retour.

Une classe (en PHP) ne peut hériter que d'une seule classe. L'héritage multiple n'est donc pas possible.

L'objectif n'est pas de faire un cours sur la programmation orientée objet. Effectivement, il reste bien des aspects à aborder concernant l'héritage et les aspects techniques qui en découlent.

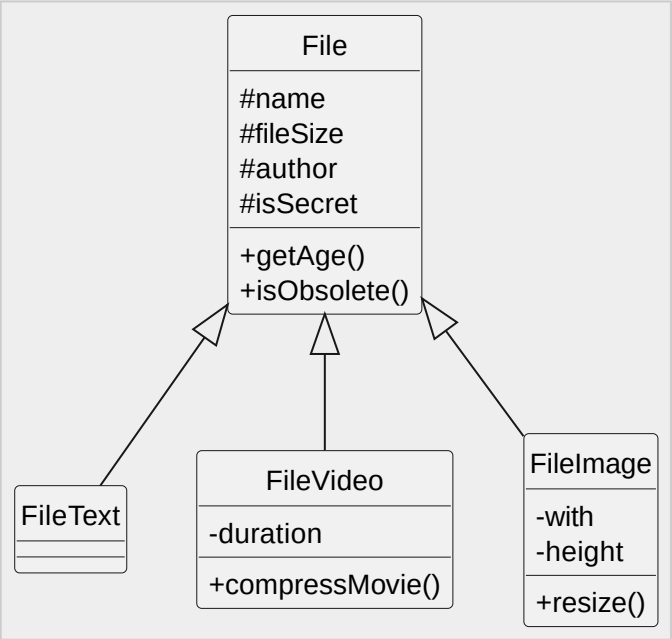
19.4. Quelques exercices

Q24) Faites évoluer le diagramme ci-dessous de façon à prendre compte les évolutions suivantes (la documnentation plantuml) :

- l'application doit prendre en compte un nouveau sous-type pour les fichiers exécutables. Un fichier exécutable doit disposer d'une méthode `isValidate` qui indique qu'il ne s'agit pas d'un virus.

- l'application doit permettre de gérer des fichiers de type "média". Ces fichiers sont des fichiers vidéos, des images, des fichiers audio, etc (il n'est pas attendu de gérer un type "audio"). Un fichier media doit pouvoir être noté avec une note chiffrée.

Diagramme de départ :



Code "plantuml" du diagramme :

```
class File {
    #name
    #fileSize
    #author
    #isSecret
    +getAge()
    +isObsolete()
}
class FileText {
}
class FileVideo {
    -duration
    +compressMovie()
}
class FileImage {
    -with
    -height
    +resize()
}
FileText -up-|> File
FileImage -up-|> File
FileVideo -up-|> File
```

Q25) Implémentez le code des classes qui apparaissent en couleur dans la correction du point précédent sans oublier les éventuelles modifications à apporter aux autres classes.

Q26) Répondre aux questions qui suivent à partir du diagramme qui a été implémenté à la question précédente

a. Quelle classe sera instanciée pour manipuler un fichier csv ?

b. Quelle classe sera instanciée pour manipuler un fichier audio ?

c. Quelle classe sera instanciée pour manipuler un fichier de type "archive" (.zip, .rar, .jar, ...) ?

20. La relation abstraite (classe abstraite)

20.1. Comprendre le sens de l'adjectif "abstrait"

La relation abstraite reprend les principes de la relation d'héritage :

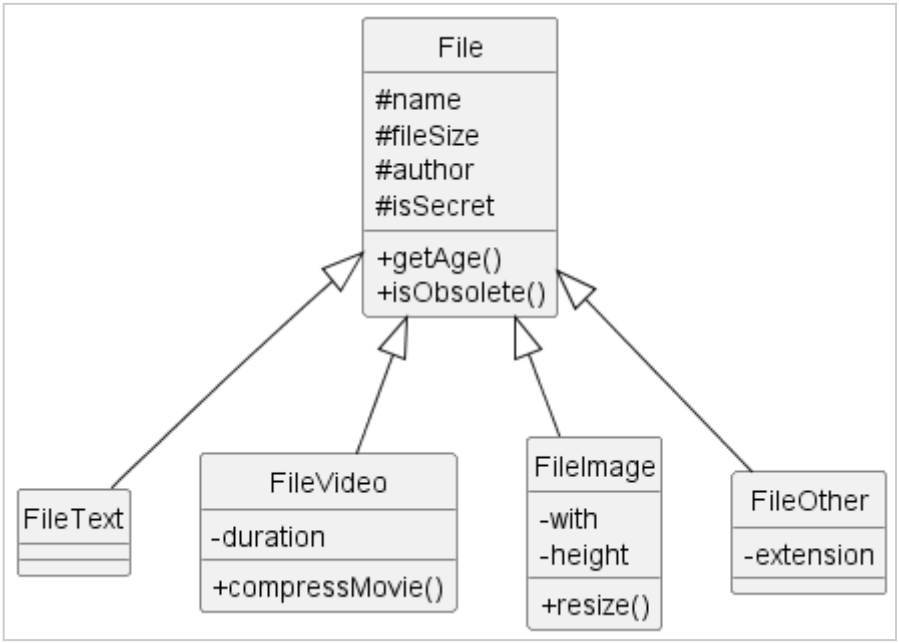
- Les membres communs (attributs et méthodes) sont généralisés dans une classe mère.
- Les membres spécifiques (attributs et méthodes) aux sous classes sont spécifiés dans ces dernières.

La difficulté majeure pour comprendre cette partie de cours est la connaissance du mot "abstrait". Comprendre ce terme aide clairement à comprendre ce que cela implique.

Une "chose" abstraite n'a pas d'existence dans la réalité.

Une classe abstraite est une classe dont les instances n'ont pas de sens dans la réalité. Cela signifie qu'il n'y a aucun intérêt à instancier la classe car les "objets" instanciés ne seront jamais utilisés. **Une classe abstraite est donc une classe qui ne peut pas être instanciée. Elle est obligatoirement héritée.**

Pour bien comprendre ce concept, nous allons partir du diagramme suivant :



La classe mère est la classe `File`. Les sous-classes héritent des membres de la classe mère (s'ils ne sont pas privés bien sûr).

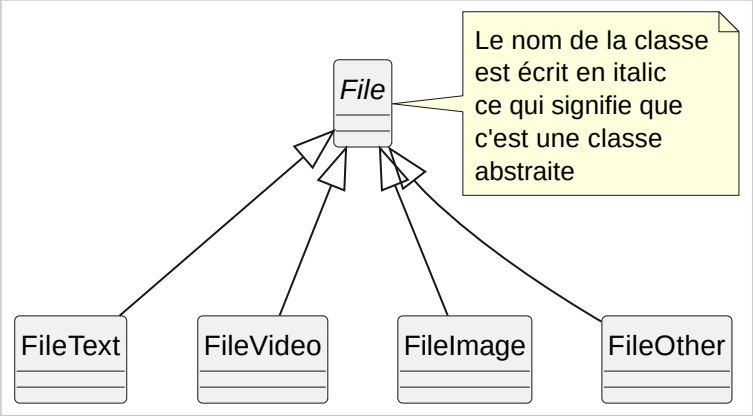
Nous avons 4 sous-types (un pour le texte, un pour la vidéo, un pour les images et un pour tous les autres types de fichiers). Cela signifie que si un fichier n'est pas du texte, une vidéo ou une image, ce sera forcément un fichier de type "autre" `FileOther`. Le type `File` est donc totalement inutile. Cette classe ne sera jamais instanciée ! Son existence n'est là que pour généraliser des membres communs aux sous-types. Puisqu'il n'y aura jamais d'instance de `File`, on peut affirmer que le type `File` ne correspond à rien de réel, c'est un type abstrait.

Lorsqu'une classe n'est pas abstraite, on parle de classe concrète.

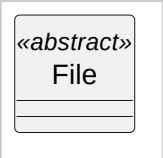
Une classe concrète est une classe qui peut être instanciée. Les instances représentent un objet du réel.

20.2. Représentation UML d'une classe abstraite

Le nom d'une classe abstraite est écrit en italique :



Il est également possible de spécifier que la classe est abstraite grâce à son stéréotype :



La flèche qui représente le lien d'héritage reste inchangée.

20.3. Implémentation d'une classe abstraite

L'implémentation d'une classe abstraite est très simple, il suffit (en PHP) de préfixer la déclaration de celle-ci avec le mot `abstract` :

```
1 <?php
2
3 abstract class File { ①
4     //ici les membres (attributs et méthodes)
5 }
6
7 class FileText extends File { ②
8     //membres spécifiques au sous-type (attributs et méthodes)
9 }
10 class FileVideo extends File { ②
11     //membres spécifiques au sous-type (attributs et méthodes)
12 }
13 class FileImage extends File { ②
14     //membres spécifiques au sous-type (attributs et méthodes)
15 }
```

```
16 class FileOther extends File { ②
17     //membres spécifiques au sous-type (attributs et méthodes)
18 }
```

① Le mot `abstract` permet de rendre la classe abstraite.

② Que l'héritage provienne d'une classe concrète ou abstraite, le mot `extends` est à utiliser dans les deux cas dans les classes filles.

Comme dit précédemment, une classe abstraite ne peut pas être instanciée mais si c'est tout de même la cas ...

```
1 $file = new File();
```

... une erreur fatale sera générée :

```
Fatal error: Uncaught Error: Cannot instantiate abstract class File in ...
```

20.4. Les méthodes aussi peuvent être abstraites

L'héritage à pour objectif de factoriser des membres au sein d'une classe afin que d'autres puissent les utiliser.

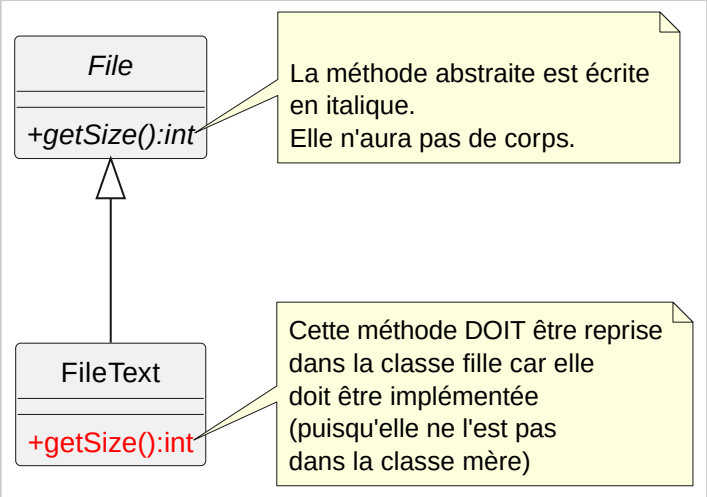
Puisqu'une classe abstraite ne peut pas être instanciée, il est tout à fait possible de prévoir une méthode qui n'a aucun corps (même pas les accolades qui marquent le début et la fin de la méthode !). Cette méthode est une méthode abstraite . Cela ne pose aucun problème puisqu'il ne sera pas possible d'appeler cette méthode puisque la classe qui la contient est abstraite.

Mais quelle utilité peut avoir une méthode sans corps me direz-vous !

N'oubliez pas que nous sommes dans un contexte d'héritage.

Une classe abstraite est forcément prévue pour être héritée (sans quoi elle n'a absolument aucune utilité). Si une classe hérite d'une classe abstraite qui contient une méthode abstraite, cela signifie que la classe fille hérite de cette méthode sans corps. Mais comme la classe fille est concrète, celle-ci doit prévoir le corps de la méthode abstraite de la mère puisqu'il sera possible de l'appeler depuis la classe fille.

Nous allons illustrer ce concept avec ce diagramme :



Voyons ce que cela donne au niveau du code :

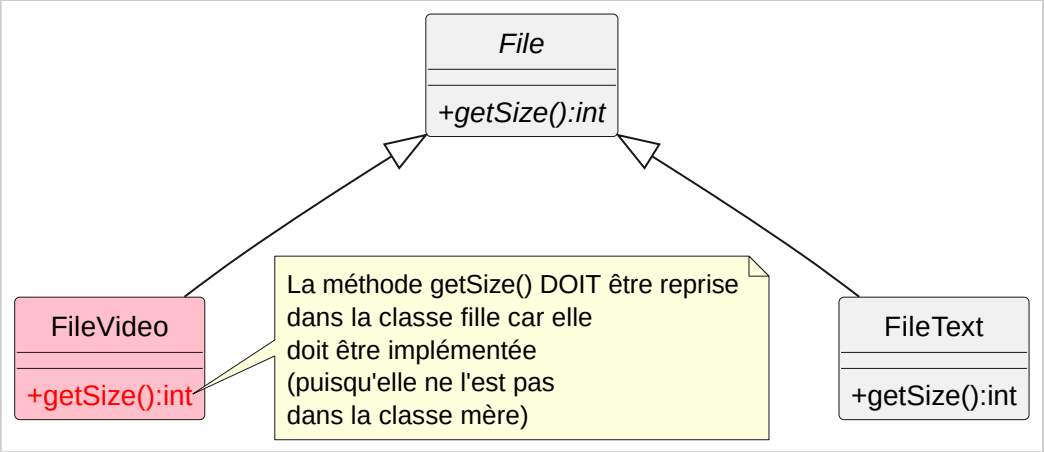
```php
abstract class File {
    //ici les membres "factorisés" de la classe

    //méthode abstraite
    abstract public function getSize():int; ①
}

class FileText extends File {
    //ici les membres spécifiques de FileText

    //l'implémentation de la méthode abstraite (qui est obligatoire)
    public function getSize():int { ②
        //ici le code qui retourne un entier.
    }
}
```

① La méthode abstraite est déclaré avec le mot `abstract` comme c'est le cas pour la classe. La méthode n'a pas de corps, donc pas d'accolade.

② La méthode abstraite est obligatoirement implémentée dans la classe fille. Elle n'a pas le choix !

Une méthode abstraite est très pratique lorsque l'on souhaite "forcer" les classes filles à implémenter une méthode particulière avec des arguments et des retours dont le type est imposé. Celle-ci permet de prévoir les contraintes à respecter par les classes filles, c'est-à-dire la signature de la méthode.

Imaginons que notre diagramme évolue et qu'une nouvelle classe hérite de la classe `File` :



Dès lors, la classe `FileVideo` doit obligatoirement prévoir la méthode `getSize()`

```
1 class FileVideo extends File {
2     //ici les membres spécifique à FileVideo
3
4     //obligation d'implémenter les méthodes abstraites de la classe mère
5     public function getSize():int {
6         //ici l'implémentation
7     }
8 }
```

Petite précision avant de clore cette partie :

J'ai précisé tout à l'heure qu'il était possible de prévoir une méthode abstraite dans une classe abstraite.

Il faut également faire le raisonnement inverse :
Dès lors qu'il y a une méthode abstraite dans une classe, cette dernière doit également être abstraite (sans quoi, on pourrait tenter d'utiliser la méthode abstraite ce qui n'est pas possible !)

21. L'interface

Point d'arrêt du cours au jeu. 16/02 à 11:09

21.1. Notion d'interface et modélisation UML

Il faut avoir parfaitement compris ce qu'est une relation abstraite pour comprendre la relation d'interface.

Une **interface** est une **classe dont toutes les méthodes sont publiques et abstraites**.

Cela ressemble à une classe abstraite sauf que contrairement à elle, une interface n'a que des méthodes sans corps. Il n'y a donc que la signature d'une ou plusieurs méthodes.

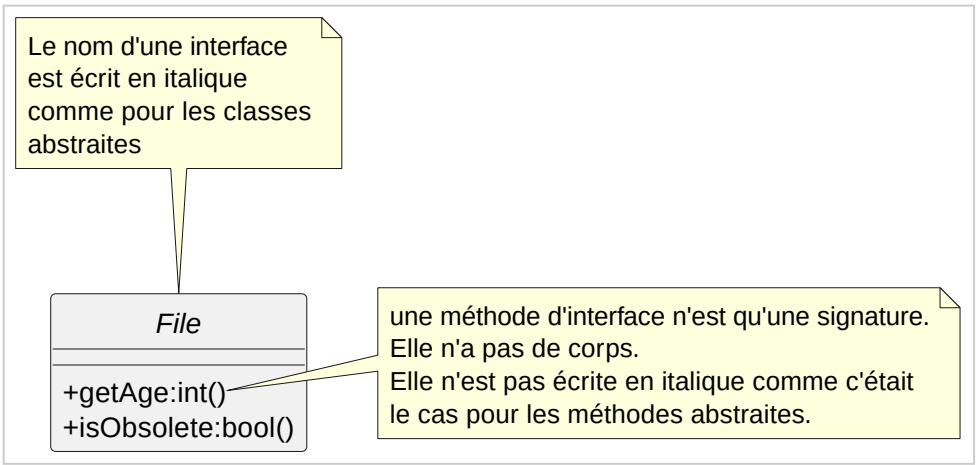
Vous devez vous demander à quoi peut servir une "classe" qui n'a que des méthodes abstraites.

Dans la partie sur la relation abstraite, nous avons vu qu'une classe qui hérite d'une classe abstraite qui contient des méthodes abstraites doit obligatoirement implémenter celles-ci. La classe qui hérite est contrainte, elle n'a pas le choix, elle DOIT implémenter ces méthodes.

Donc, si une classe *hérite* d'une interface, elle devra obligatoirement implémenter les méthodes de celle-ci. C'est très utile pour contraindre les classes à être manipulées avec des méthodes qui sont prévisibles car définies par l'interface.

En programmation, on ne dit pas qu'une classe *hérite* d'une interface, on dit qu'une classe **implémente une interface**.

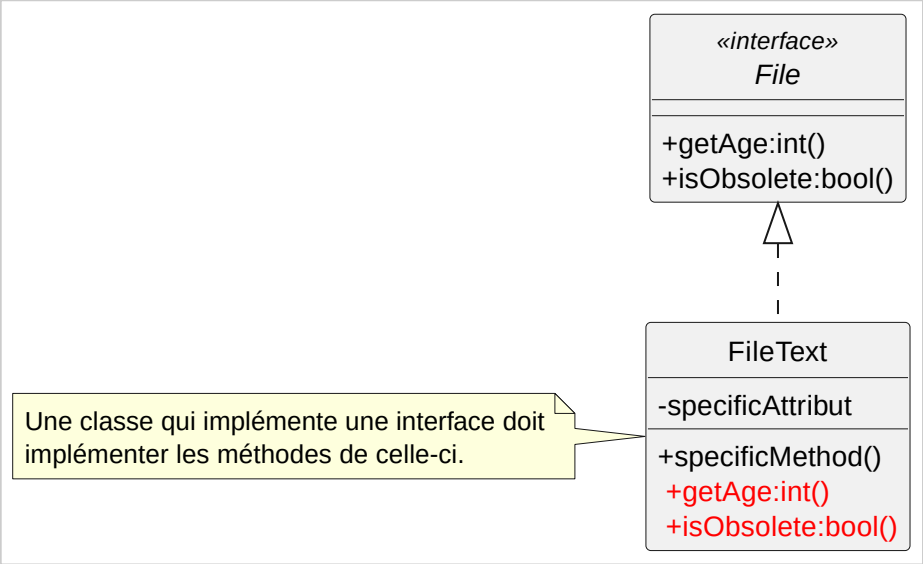
Le mieux est d'illustrer cette logique par un diagramme :



Il est difficile de distinguer une interface d'une classe abstraite (dans les deux cas, le nom est écrit en italique). Il est possible d'utiliser le stéréotype pour une meilleure lisibilité :

Nous avons dans ce diagramme une interface (il ne faut pas utiliser le terme de "classe" dans le cadre d'une interface) qui prévoit deux méthodes. Cela signifie que toute classe implémentant cette interface devra prévoir l'implémentation de ces 2 méthodes.

Ajoutons une classe `FileText` qui **implémente** l'interface `File` :



21.2. Implémentation d'une relation avec une interface

Une interface est déclarée (en PHP) avec le mot `interface` :

```
1 interface File ①
2 {
3     /**
4      * @return int retourne la taille du fichier en Mo
5      */
6     public function getSize():int; ②
7
8     /**
9      * @return int retourne l'âge du fichier en mois
10     */
11    public function getAge():int; ②
12
13 }
```

① Déclaration d'une interface

② Signature des méthodes qui devront être implémentées par les classes qui vont implémenter l'interface

Maintenant que notre interface est en place, nous pouvons indiquer à la classe `FileText` de l'implémenter :

```
1 class FileText implements File{ ①
2
3     //ici les membres spécifiques de FileText
4
5 }
```

① La classe **implémente** l'interface `File` .

Enfin, il ne reste plus qu'à implémenter les méthodes de l'interface :

```
1 class FileText implements File{ ①
2
3     //ici les membres spécifiques de FileText
4
5
6     public function getAge(): int
7     {
8         // ici du code qui retourne un nombre de mois
9     }
10
11    public function getSize(): int
12    {
13        // ici du code qui retourne un nombre de Mo
14    }
15 }
```

① Notre classe implémente toutes les méthodes de l'interface

Si une classe ne peut pas hériter de plusieurs classes mères, elle peut implémenter plusieurs interfaces :

```
1 class classA implements classB, classC, classD {
2     //ici les membres spécifiques de classA
3
4     //ici toutes les méthodes déclarées dans les interfaces classB,
  classC et classD
5 }
```

Index

T

V