UML

Baptiste Bauer

Version v0.0.8, 2022-11-23 09:31:43

Table des matières

1. L’association n-aire 1
1.1. Qu’est-ce qu’une association n-aire ? 1
1.2. Implémentation d’une association n-aire 6
1.3. Exercice 6
2. L’association porteuse (ou classe association) 8
2.1. Qu’est-ce qu’une association porteuse ou classe association ? 8
2.2. Implémentation d’une classe associative 9
2.3. Dans la pratique, on simplifie les choses 18
2.4. Que faire si une classe association porte sur une association n-aire ? 24
3. La relation de dépendance 26
3.1. Qu’est-ce qu’une dépendance ? 26
3.2. Implémentation d’une dépendance 27
Index 30

1. L'association n-aire

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

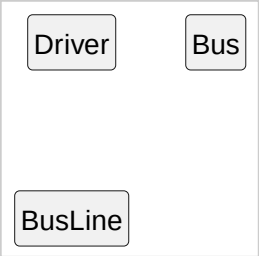
Jusqu'à maintenant, nous n'avons abordé que des relations entre deux classes. Cela couvre la très grande majorité des cas mais parfois, 3 classes ou plus peuvent être liées entre elles.

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.

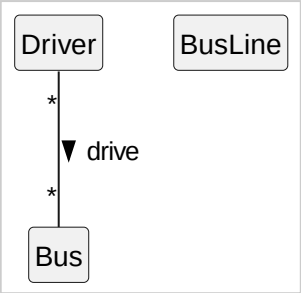
Je vais expliquer ce concept en m'appuyant sur un cas qui va évoluer progressivement.

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

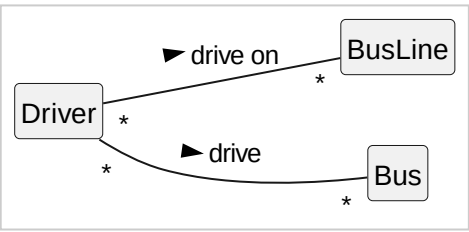
Nous avons 3 classes à modéliser :



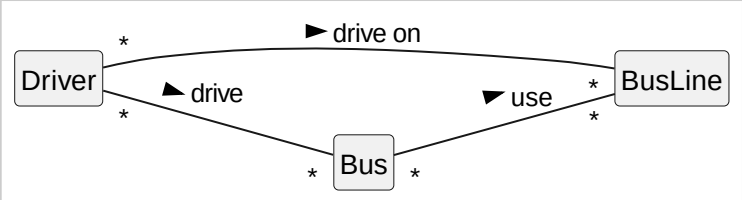
Nous savons qu'un chauffeur ne conduit pas toujours le même bus et qu'un bus peut être conduit par plusieurs chauffeurs. Cela donne la modélisation suivante :



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



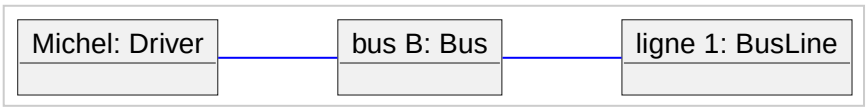
Et de continuer en indiquant qu'un bus circule sur une ligne de bus qui n'est pas toujours la même :



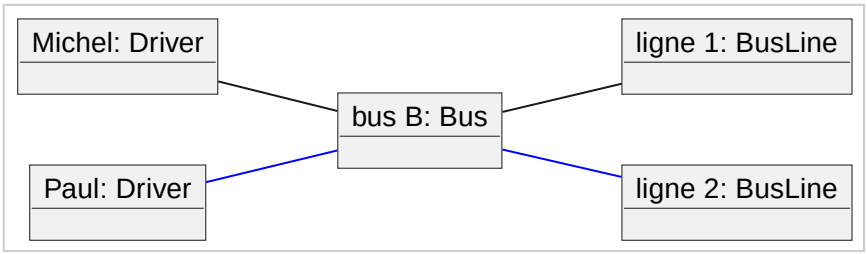
Cependant, cette modélisation conduit à ne connaître que les couples Driver/Bus, Driver/BusLine, Bus/BusLine.

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 :

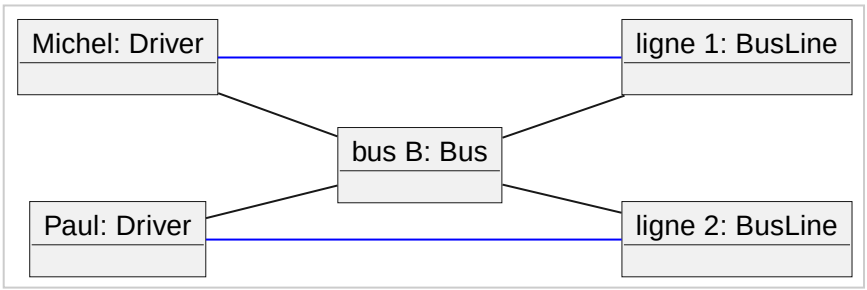


- Paul conduit le bus B sur la ligne 2



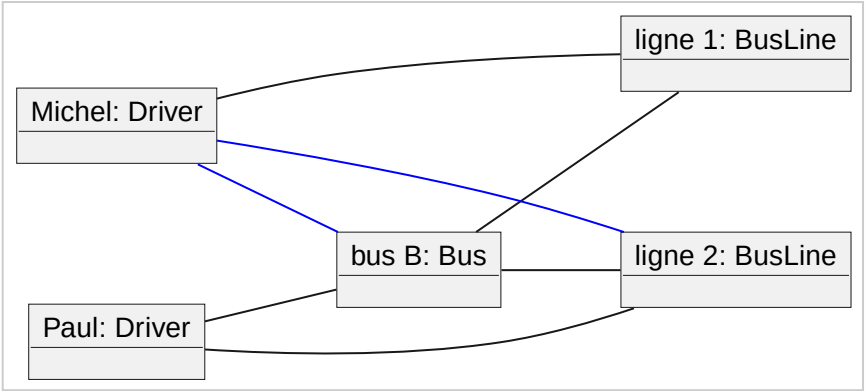
A ce stade, il n'est plus possible de savoir sur quelle ligne de bus Paul a conduit le bus B car ce bus est lié à deux lignes. Mais, notre diagramme de classes de tout à l'heure nous montre qu'il y a une association entre `Driver` et `BusLine`. Nous pouvons donc associer la ligne de bus au chauffeur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

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



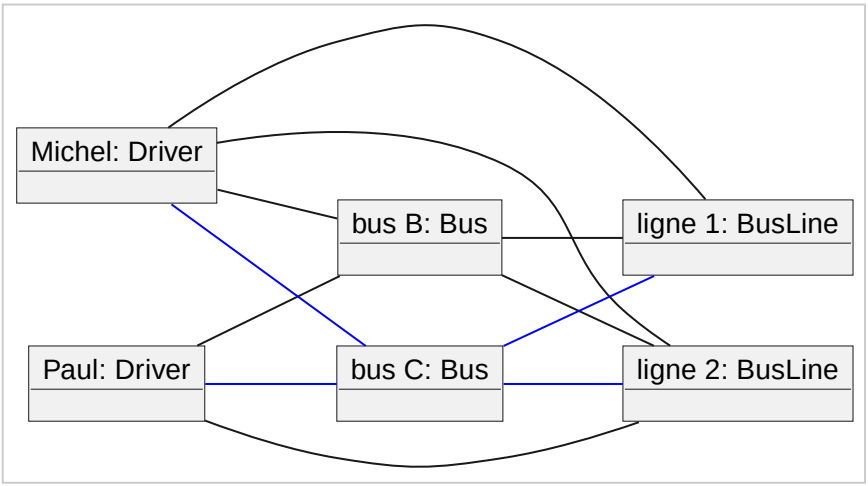
On peut maintenant affirmer que Michel a conduit le bus B et qu'il l'a fait sur la ligne 1. Mais que se passe-t-il s'il doit conduire le bus B sur la ligne 2 ?

- 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 :

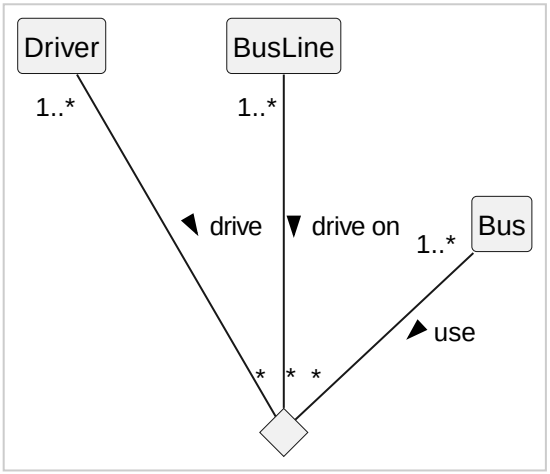
- Michel conduit le bus B sur la ligne 1 (effectivement, le bus B est lié à 2 lignes, cela ne veut pas dire que Michel à conduit sur la ligne 1. Ce peut être Paul)
- La ligne 2 est utilisée par Paul avec le bus B (effectivement, la ligne 2 est liée au bus B mais c'est peut être Paul qui conduisait).

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

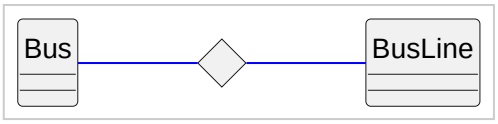
trois classes, on parle d'**association ternaire**.

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

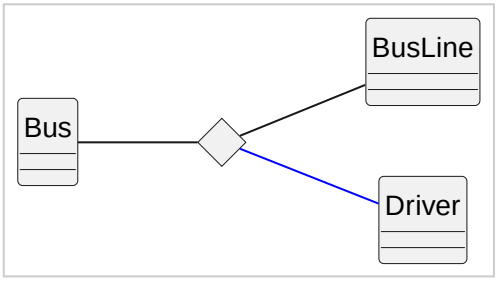


Vous remarquerez les cardinalités minimales à 1.
Il ne peut y avoir une association alors qu'il manquerait un chauffeur ou la ligne de bus ou encore le bus.

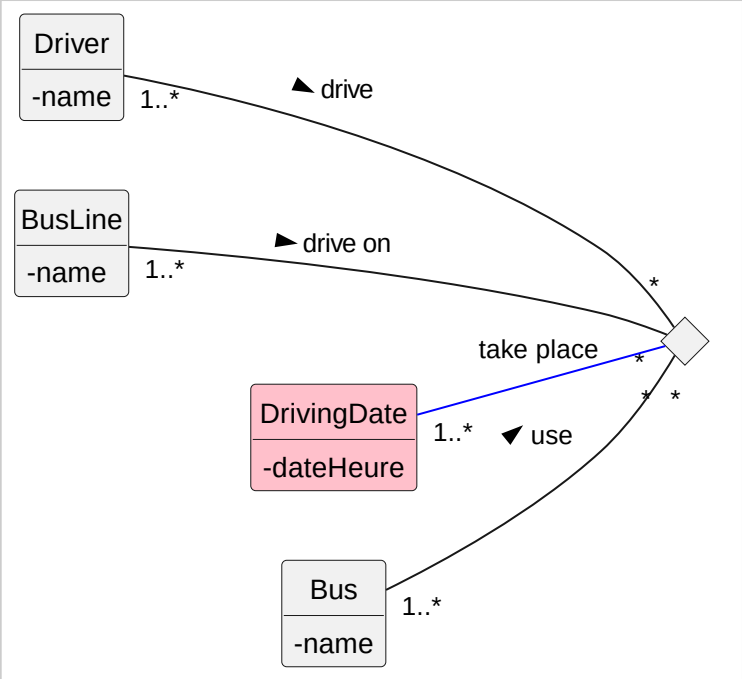
Si vous avez des difficultés à comprendre la représentation avec le losange carré, imaginez qu'une association binaire soit représentée comme ceci :



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

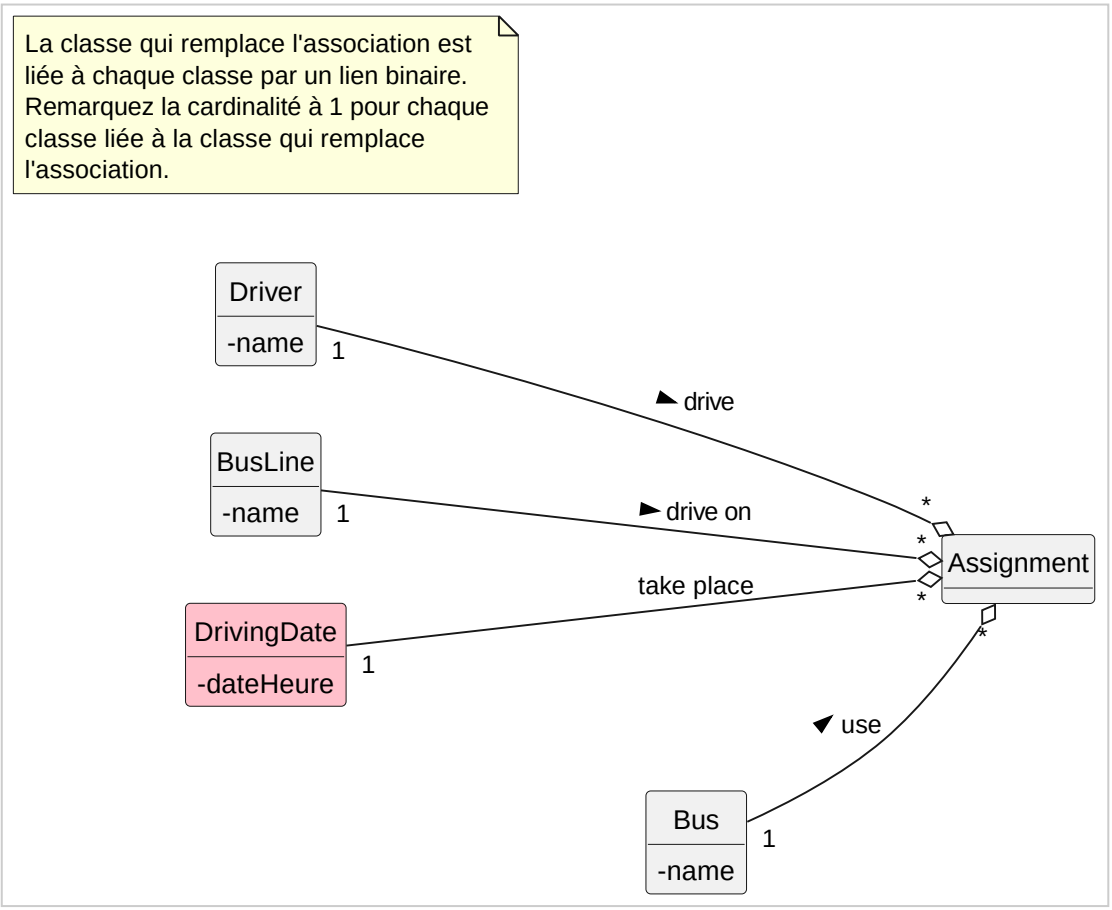


Il subsiste encore un manque. Michel sait qu'il doit conduire le bus B sur la ligne 3, mais c'est aussi le cas pour d'autres chauffeurs. Il faut donc pouvoir préciser "quand" chacun d'eux va conduire le bus B sur la ligne 3. La solution consiste à ajouter un objet date à l'association ternaire :



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 :

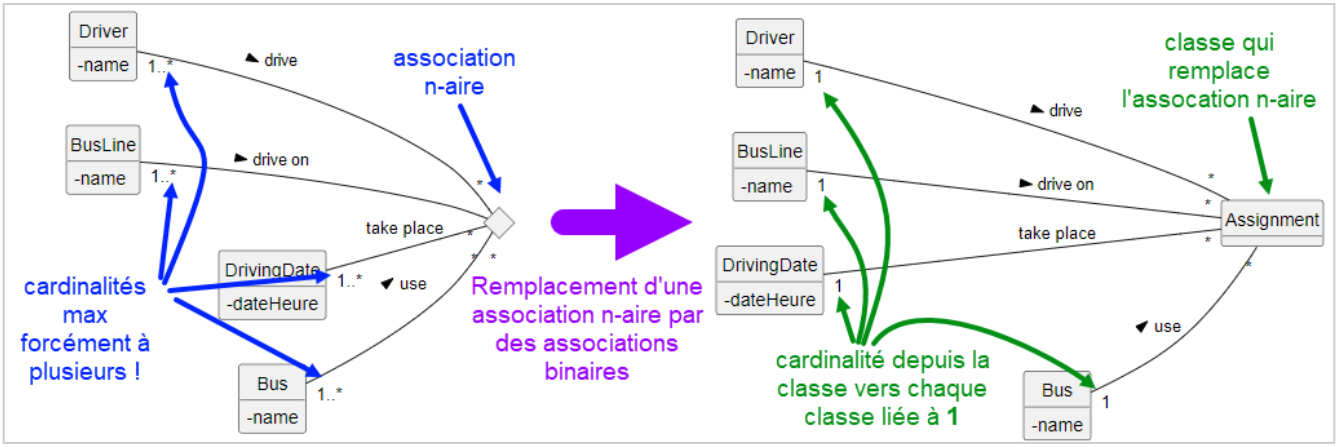


La conséquence est qu'il ne reste plus que des associations binaires que nous

savons traiter.

1.2. 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.

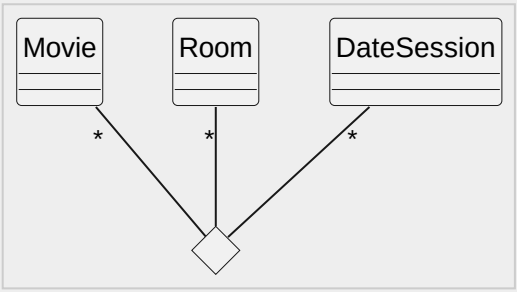
Une fois que vous n'avez plus que des associations binaires, il n'y a qu'à réaliser les implémentations comme nous l'avons appris jusqu'à maintenant. Il n'y a rien de nouveau à aborder sur ce point.

1.3. Exercice

Q1) 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.

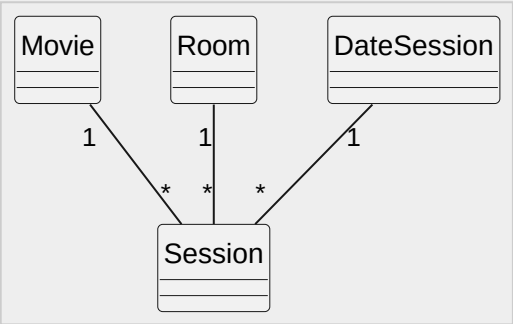
Correction de Q1



Un film est projeté plusieurs fois dans une même salle. Il faut pouvoir distinguer les

projections entre elles. Cela est rendu possible avec `DateSession`.

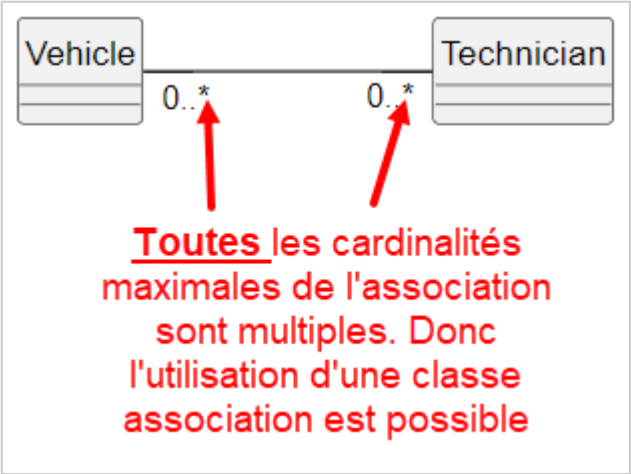
On remplace l'association ternaire par des associations binaires :



2. L'association porteuse (ou classe association)

2.1. 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.

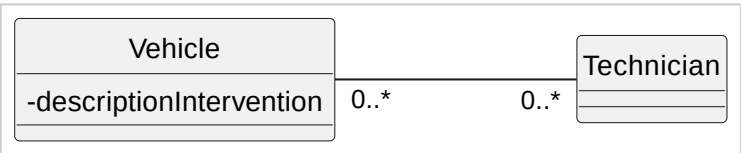


Une **classe association** permet de prévoir des attributs et/ou des méthodes qui ne concernent que le "couple" d'objets liés. C'est-à-dire qu'il n'est pas possible de rattacher un tel attribut ou une telle méthode à une des deux classes en particulier.

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.

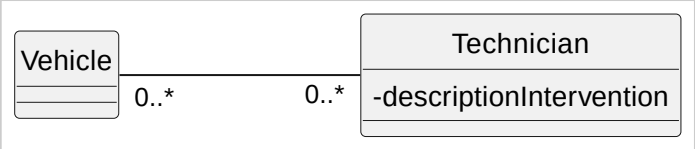
Si l'on souhaite conserver une trace de l'intervention, il faut se poser la question de la place de l'attribut à ajouter sur le diagramme.

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` :

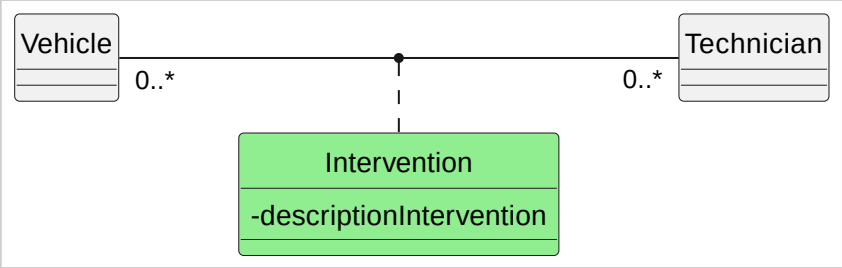


Avec cette nouvelle solution, on sait par qui est écrite une description d'intervention mais on ne sait pas quel véhicule cela concerne.

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).

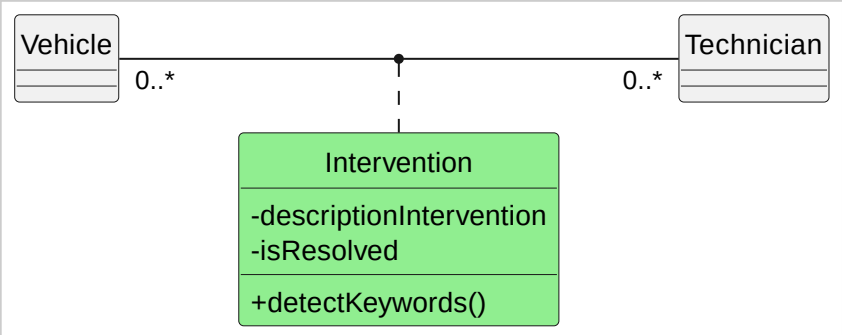
Lorsque vous vous trouvez dans une telle situation, c'est forcément que les cardinalités maximales de l'association sont multiples (sinon, c'est que vous vous êtes trompés sur les cardinalités ou que vous rattachez mal votre attribut).

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 :



2.2. 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
     *
```

```
53      * @param array $technicians collection d'objets de type Technician
54      *
55      * @return $this
56      */
57     public function setTechnicians(array $technicians): self
58     {
59
60         //mise à jour de la collection de techniciens
61         foreach ($technicians as $technician) {
62             $this->addTechnician($technician);
63         }
64
65         return $this;
66     }
67
68
69
70     /**
71      * @return Technician[]
72      */
73     public function getTechnicians(): array
74     {
75         return $this->technicians;
76     }
77
78 }
```

Poursuivons avec la classe `Technician` :

```
1  <?php
2
3
4
5  class Technician
6  {
7      /**
8       * @param array $vehicles tableau d'objets de type Vehicle
9       */
10     public function __construct(
11         private array $vehicles = []
12     ) {
13
14         $this->setVehicles($vehicles);
15
16     }
17
18     //mutateurs et accesseurs pour la collection de Vehicle
19
20     /**
21      * @param Vehicle $vehicle ajoute un item de type Vehicle à la collection
```

```php
     */
    public function addVehicle(Vehicle $vehicle): bool
    {
        if (!in_array($vehicle, $this->vehicles, true)) {
            $this->vehicles[] = $vehicle;

            return true;
        }

        return false;
    }

    /**
     * @param Vehicle $vehicle retire l'item de la collection
     */
    public function removeVehicle(Vehicle $vehicle): bool
    {
        $key = array_search($vehicle, $this->vehicles, true);

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

            return true;
        }

        return false;
    }

    /**
     * Initialise la collection avec la collection passée en argument
     *
     * @param array $vehicles collection d'objets de type Vehicle
     *
     * @return $this
     */
    public function setVehicles(array $vehicles): self
    {
        foreach ($vehicles as $vehicle) {
            $this->addVehicle($vehicle);
        }

        return $this;
    }


    /**
     * @return Vehicle[]
     */
    public function getVehicles(): array
    {
        return $this->vehicles;
```

```
73     }
74
75 }
```

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.

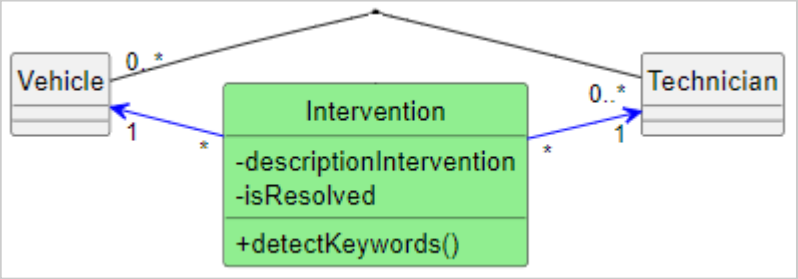
La classe association `Intervention` est avant tout une classe comme une autre, simple à implémenter :

```
1  <?php
2
3
4
5
6
7  class Intervention
8  {
9      /**
10      * @param string $descriptionIntervention
11      * @param bool   $isResolved
```

```
12      */
13     public function __construct(
14         private string $descriptionIntervention,
15         private bool $isResolved
16 
17     )
18     {
19     }
20 
21 
22     //mutateurs et accesseurs des attributs Intervention::descriptionIntervention
   et Intervention::isResolved
23 
24 }
```

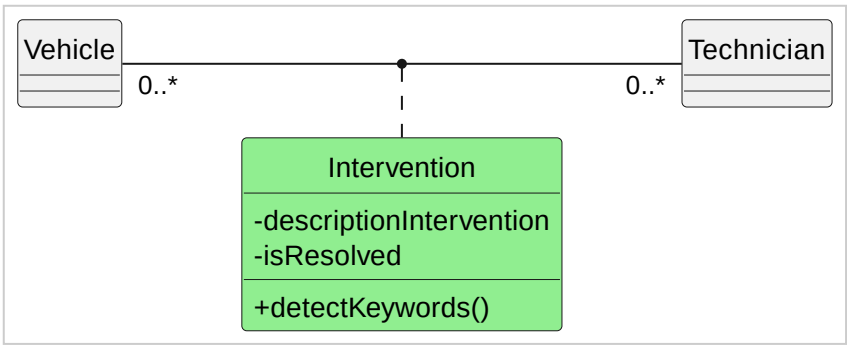
Maintenant, il faut prendre en compte le fait qu'une classe association doit connaître chaque instance des extrémités de l'association sur laquelle elle porte. Cela signifie que `Intervention` peut naviguer vers `Vehicle` et vers `Technician`. De plus, comme une intervention ne concerne qu'un véhicule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

Cette interprétation peut être modélisée ainsi :

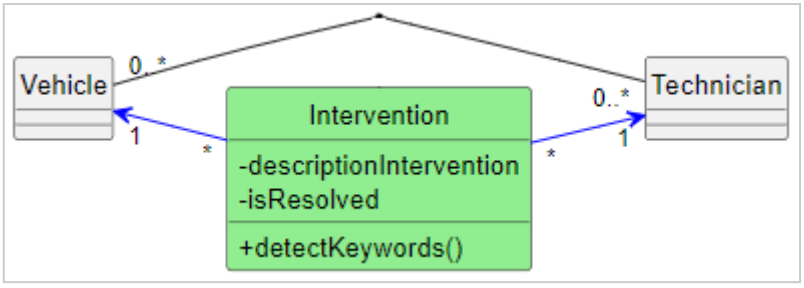


Cette représentation n'est pas valide, elle représente seulement la réflexion que l'on vient de mener. Une classe association, c'est considérer qu'il y a un lien entre cette classe et chaque classe de l'association avec une cardinalité à 1.

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` :

```php
<?php




class Intervention
{
    /**
     * @param string $descriptionIntervention
     * @param bool   $isResolved
     */
    public function __construct(
        private string $descriptionIntervention,
        private bool $isResolved
        ,
        private Vehicle $vehicle

    )
    {
    }


    //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
     */
```

```
37     public function setVehicle(Vehicle $vehicle): void
38     {
39         $this->vehicle = $vehicle;
40 
41     }
42 
43     //mutateurs et accesseurs des attributs Intervention::descriptionIntervention
   et Intervention::isResolved
44 
45 }
```

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

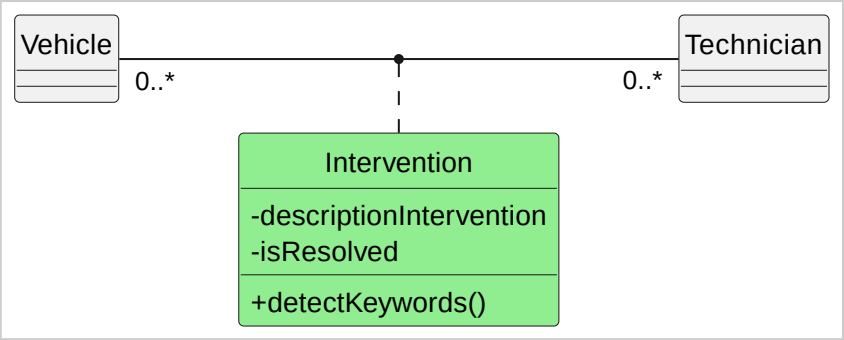
```
1  <?php
2  
3  
4  
5  
6  
7  class Intervention
8  {
9      /**
10      * @param string $descriptionIntervention
11      * @param bool   $isResolved
12      */
13     public function __construct(
14         private string $descriptionIntervention,
15         private bool $isResolved
16         ,
17         private Vehicle $vehicle
18 
19         ,
20         private Technician $technician
21     )
22     {
23     }
24 
25 
26     //accesseur pour le véhicule
27     /**
28      * @return Vehicle
29      */
30     public function getVehicle(): Vehicle
31     {
32         return $this->vehicle;
33     }
34 
35     //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)
36      /**
37       * @param Vehicle $vehicle
38       */
39      public function setVehicle(Vehicle $vehicle): void
40      {
41          $this->vehicle = $vehicle;
42
43      }
44
45      /**
46       * @return Technician
47       */
48      public function getTechnician(): Technician
49      {
50          return $this->technician;
51      }
52
53      /**
54       * @param Technician $technician
55       */
56      public function setTechnician(Technician $technician): void
57      {
58          $this->technician = $technician;
59      }
60      //mutateurs et accesseurs des attributs Intervention::descriptionIntervention
    et Intervention::isResolved
61
62  }
```

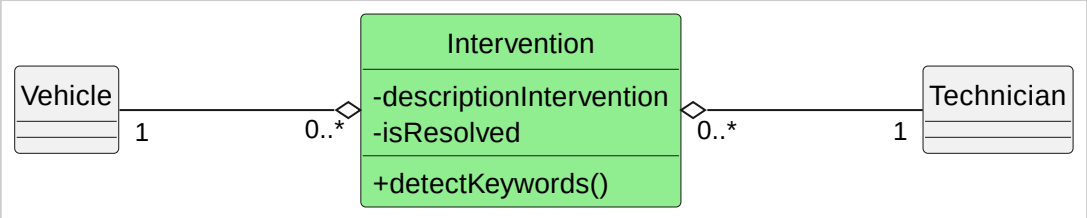
Désormais, vous savez implémenter une classe association. Cependant, cette implémentation n'est généralement pas la solution retenue.

2.3. 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` et donc plus de méthode `addTechnician` et `removeTechnician` :

```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;
```

```php
    }

    /**
     * @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;
    }

    //méthode à ajouter dans la classe Vehicle
    public function getTechnicians(){
        $technicians = [];
        /** @var Intervention $intervention */
        foreach($this->interventions as $intervention){
            $technicians[]= $intervention->getTechnician();
        }
        return $technicians;

    }
}
```

- La classe `Technician` n'a plus d'attribut `$vehicles` et donc plus de méthode `addVehicle` et `removeVehicle` :

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

```php

    }

    /**
     * @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;
    }

    //méthode à ajouter dans la classe Technician
    public function getVehicles(){
        $vehicles = [];
        /** @var Intervention $intervention */
        foreach($this->interventions as $intervention){
            $vehicles[]= $intervention->getVehicle();
        }
        return $vehicles;

```

```
59     }
60 }
```

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

```php
class Intervention
{
    /**
     * @param string     $descriptionIntervention
     * @param bool       $isResolved
     * @param Vehicle    $vehicle
     * @param Technician $technician
     */
    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;

        //mise à jour de l'objet lié pour la navigabilité bidirectionnelle
        $vehicle->addIntervention($this);
    }

    /**
     * @return Technician
```

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

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

        //mise à jour de l'objet lié pour la navigabilité bidirectionnelle
        $technician->addIntervention($this);
    }

    //mutateurs et accesseurs des autres attributs
    //...

}
```

Q2) 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 :

```php
$liste = $vehicleA->getTechnicians(); //tableau contenant tous les techniciens qui sont intervenus sur le véhicule A
```

Correction de Q2

```php
//class Vehicle
    //méthode à ajouter dans la classe Vehicle
    public function getTechnicians(){
        $technicians = [];
        /** @var Intervention $intervention */
        foreach($this->interventions as $intervention){
            $technicians[]= $intervention->getTechnician();
        }
        return $technicians;

    }
```

Q3) 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

Correction de Q3

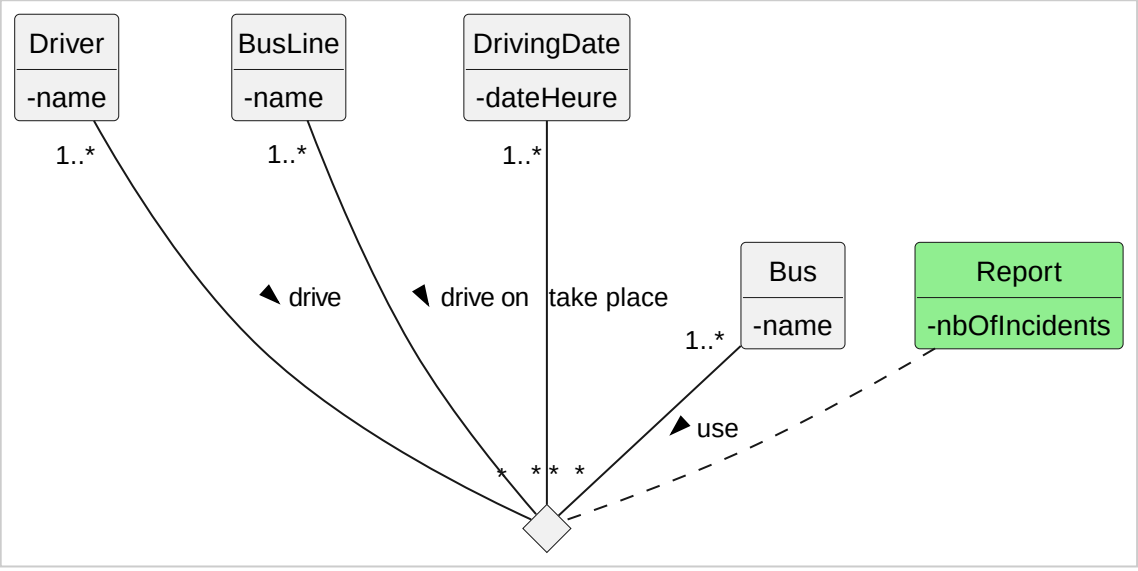
```
//class Technician
    //méthode à ajouter dans la classe Technician
    public function getVehicles(){
        $vehicles = [];
        /** @var Intervention $intervention */
        foreach($this->interventions as $intervention){
            $vehicles[]= $intervention->getVehicle();
        }
        return $vehicles;

    }
```

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

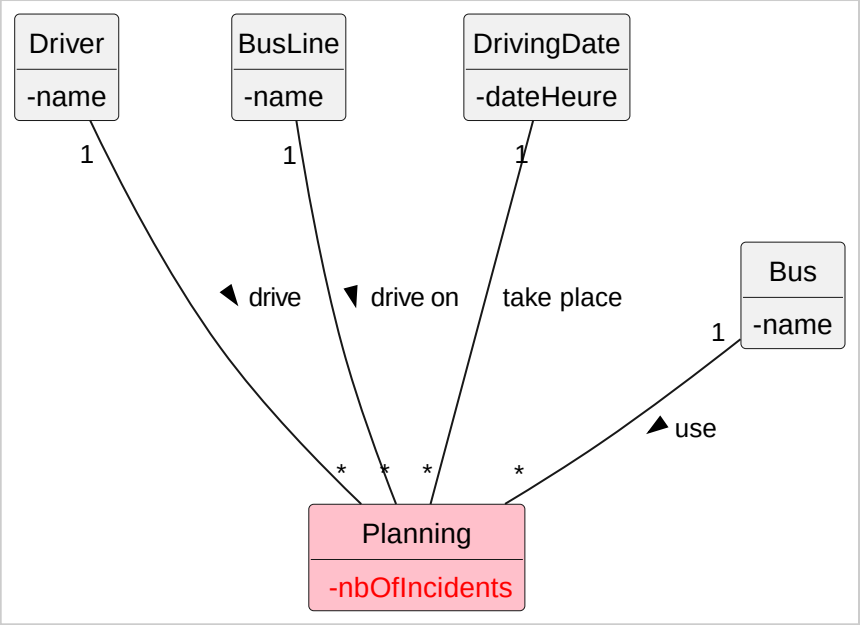
Lorsque votre analyse aboutit à une associations n-aire avec une classe association, cela peut être déroutant.

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 !

3. La relation de dépendance

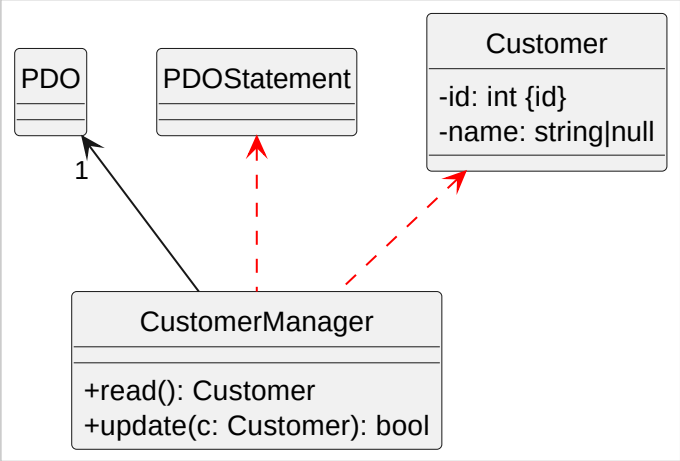
3.1. Qu'est-ce qu'une dépendance ?

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.

3.2. 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.

Index

A

association porteuse, 9
association ternaire, 4

C

classe association, 8

D

dépendance, 26

R

relation n-aire, 1