UML

Baptiste Bauer

Version v0.0.2, 2022-11-21 19:05:25

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.

Correction de Q1

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.

Correction de Q2

Pour Phpstorm :

File › Settings › Plugins

[installation plugin plantuml phpstorm] | *../images/installation-plugin-plantuml-phpstorm.png*

Ensuite, il suffit de créer un nouveau fichier `PlantUML File` (ou un fichier avec l'extension `puml`)

Un fichier `puml` doit commencer par `@startuml` et se terminer par `@enduml`.

Pour Visual Studio Code :

Il faut installer l'extension plantuml pour VSCode

L'extension par défaut d'un fichier utilisant PlantUml est `plantuml`. Mais on peut tout à fait utiliser l'extension `puml`.

Code PlantUml à écrire dans le fichier :

```
@startuml ①
```

```
class Person {
    - id: int {id} //pas besoin d'écrire {unique} pour un id
    - name: string
    - dateOfBirth: Datetime //ou tout équivalent

    +setId() //en principe, l'id est récupéré via la bdd
    +getId()
    +setName(name: string):self
    +getName(): string
    +setDateOfBirth(Datetime):self
    +getDateOfBirth(): Datetime
    + getAge(): int
}

class Enterprise {
    -corporateName: string
    -employees: Person[ ] //collection d'objets de type Person

    +setCorporateName(name: string):self
    +getCorporateName():string
    +addEmployee(e: Person):self
    +getEmployees(): array
    +removeEmployee(e: Person)
    +countEmployees():int
}

@enduml ②
```

① Un fichier plantuml commence par `@startuml`

② Un fichier plantuml commence par `@enduml`

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()`.

Correction de Q3

```
<?php

//avant PHP 8
```

```php
class AnimalOld{
    private string $name;

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

    public function __toString(): string
    {
        return "Mon nom est $this->name";
    }

    public function getName(): string
    {

        return $this->name;

    }
}

//version 8 de php
//les propriétés déclarées avec une visibilité  dans une méthode sont automatiquement des propriétés d'objet.
// depuis php 8.0, il est conseillé d'implémenter l'interface Stringable explicitement lors de l'appel à la méthode __toString. Si ce n'est pas fait, elle sera implémentée implicitement.
class Animal implements Stringable {

    public function __construct(private string $name)
    { }

    public function __toString(): string
    {
        return "Mon nom est $this->name";
    }

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

//affichage
$dragon1 = new Animal('Viserion');
echo $dragon1;
echo '<br/>';
$dragon2 = new Animal('Drogon');
echo "Le nom de cet animal est {$dragon2->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 :

[extrait diagramme de classes] | *../images/extrait-diagramme-de-classes.png*

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)
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.)

Correction de Q4

a.)
   - Une personne utilise 0 à plusieurs animaux
   - un animal est utilisé par une seule personne (donc toujours la même)

b.)
   - Un serveur sert 0 à plusieurs tables
   - une table est servie par au moins un serveur

c.)
   - Une personne est le parent de 0, 1 ou plusieurs enfants
   - Un enfant a 0, 1 ou 2 parents

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.)

Correction de Q5

a. L'association entre `Person` et `Animal` est une association multiple. De `Animal` à `Person`, l'association est simple.

b. L'association entre `Waiter` et `Table` est une association multiple. De `Table` à `Waiter`, l'association est multiple.

c. L'association entre `Person` (parent) et `Person` (enfant) est une association multiple. De `Person` (enfant) à `Person` (parent), l'association est multiple.

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 équipé 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.

Correction de Q6

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

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

```
27
28         return $this;
29     }
30
31 }
```

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

```php
<?php

class Vehicle
{
    private string $registerNumber;


    public function __construct(string $registerNumber, ?Technician $technician =
null)
    {
        $this->registerNumber = $registerNumber;
        $this->technician = $technician;
    }

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

    /**
     * @param string $registerNumber
     *
     * @return Vehicle
     */
    public function setRegisterNumber(string $registerNumber): Vehicle
    {
        $this->registerNumber = $registerNumber;

        return $this;
    }


}
```

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 :

```
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
3 $vehicleBBBB = new Vehicle('BBBB');
4 $sofien = new Technician('Sofien');
5
6 $vehicleBBBB->setTechnician($sofien);
7
8 //récupération du technicien depuis le véhicule
9 $technicianOfBBBB = $vehicleBBBB->getTechnician(); ①
10
11 echo "{$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;
    }

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

        return $this;
    }

```

```
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      */
```

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

    /**
     * @param Technician|null $technician
     *
     * @return Vehicle
     */
    public function setTechnician(?Technician $technician): Vehicle
    {
        $this->technician = $technician;

        return $this;
    }

    //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
    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;
    }
}
```

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

    /**
     * @param string $name
     *
     * @return Technician
     */
    public function setName(string $name): Technician
    {
```

```
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
4  class Vehicle
5  {
6
7
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
4     public function __construct(
5         private string $registerNumber,
6         private array $technicians = [],  ①
7     )
8     {
9     }
```

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 :

```
1 //à ajouter à la classe Vehicle
2 <?php
3
4
5     /**
6      * @param Technician $technician ajoute un item de type Technician à la
```

```
7       *                               collection
8       */
9      public function addTechnician(Technician $technician): bool
10     {
11         if (!in_array($technician, $this->technicians, true)) { ①
12             $this->technicians[] = $technician; ②
13
14             return true;
15         }
16
17         return false;
18     }
```

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) :

```
1  //à ajouter à la classe Vehicle
2  <?php
3
4
5      /**
6       * @param Technician $technician retire l'item de la collection
7       */
8      public function removeTechnician(Technician $technician): bool
9      {
10         $key = array_search($technician, $this->technicians, true); ①
11
12         if ($key !== false) {
13             unset($this->technicians[$key]); ②
14
15             return true;
16         }
17
18         return false;
19     }
```

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 :

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

```
3
4
5       public function __toString(): string
6       {
7           $string = "Je suis le véhicule immatriculé {$this->registerNumber}.";
8
9           if (count($this->technicians) === 0) {
10              $string .= "\nJe ne suis associé à aucun technicien.\n";
11          } else {
12              $string .= "\nJe suis associé à un ou plusieurs techniciens :";
13              foreach ($this->technicians as $technician) {
14                  $string .= "\n- {$technician->getName()}";
15              }
16          }
17
18          return $string;
19      }
```

Testons cela en affectant 3 techniciens à un véhicule :

```
1  <?php
2
3
4  $vehicleAAAA = new Vehicle('AAAA');
5  $paul = new Technician('Paul');
6  $sofien = new Technician('Sofien');
7  $anna = new Technician('Anna');
8
9  //affectation de plusieurs techniciens
10 $vehicleAAAA->addTechnician($paul);
11 $vehicleAAAA->addTechnician($sofien);
12 $vehicleAAAA->addTechnician($anna);
13
14 echo $vehicleAAAA;
```

Résultat :

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

Retirons un technicien :

```
1  <?php
2
```

```
3
4 $vehicleAAAA->removeTechnician($paul);
5 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 :

```
1 //à ajouter à la classe Vehicle
2 <?php
3
4
5     /**
6      * @return Technician[]
7      */
8     public function getTechnicians(): array
9     {
10         return $this->technicians;
11     }
```

Voici comment utiliser ce mutateur `getTechnians()`:

```
1 <?php
2
3
4 echo "\nVoici la liste des techniciens du véhicule {$vehicleAAAA->
  getRegisterNumber()} :";
5
6 foreach ($vehicleAAAA->getTechnicians() as $technician) { ①
7     echo "\n* Technicien {$technician->getName()}";
8 }
```

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()` :

```
1 <?php
2
```

```
3
4       /**
5        * Initialise la collection avec la collection passée en argument
6        *
7        * @param array $technicians collection d'objets de type Technician
8        *
9        * @return $this
10       */
11      public function setTechnicians(array $technicians): self
12      {
13          $this->technicians = $technicians; ①
14
15
16          return $this;
17      }
```

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 :

```
1 <?php
2
3
4
5 $cedric = new Technician('Cédric');
6 $baptiste = new Technician('Baptiste');
7
8 $techniciansCollection = [$cedric, $baptiste];
9
10 $vehicleCCCC = new Vehicle('CCCC');
11
12 $vehicleCCCC->setTechnicians($techniciansCollection);
13
14 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` :

```
1 <?php
2
3
4
```

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

$techniciansCollection = [$cedric, $baptiste];

$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 :

```
<?php



$arrayInt = [14,84,170];

$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 :

```
<?php



$arrayInt = [14,84,170];

$vehicleCCCC = new Vehicle('CCCC', $arrayInt); ①

//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 :

```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
    {

        foreach($technicians as $technician){ ①
            $this->addTechnician($technician); ②
        }

        return $this;
    }
```

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 :

```php
<?php



$arrayInt = [14,84,170];

$vehicleDDDD = new Vehicle('DDDD');

$vehicleDDDD->setTechnicians($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
4     public function __construct(
5         private string $registerNumber,
6     )
7     {
8         $this->setTechnicians($technicians); ①
9     }
```

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
{


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

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

```
23
24          return $this;
25      }
26
27
28  }
```

Faisons de même pour la classe `Vehicle` :

```php
<?php


class Vehicle
{

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

    /**
     * @param string $registerNumber
     *
     * @return Vehicle
     */
    public function setRegisterNumber(string $registerNumber): Vehicle
    {
        $this->registerNumber = $registerNumber;

        return $this;
    }


}
```

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` :

```php
<?php

```

```
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 :

```
1 //classe Vehicle
2 <?php
3
4
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    }
```

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 :

```
1 //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 :

```
//classe Technician
<?php

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

    /**
     * @param Vehicle|null $vehicle
     *
     * @return Technician
     */
    public function setVehicle(?Vehicle $vehicle): Technician
    {
        $this->vehicle = $vehicle;

        return $this;
    }
```

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

```
//classe Technician
<?php

    public function __toString(): string
    {
        $string = "Je suis le technicien nommé {$this->name}.";

        if ($this->vehicle === null) {
            $string .= " Je n'ai pas de voiture en charge.";
        } else {
            $string .= " La voiture dont j'ai la charge a pour immatriculation
{$this->vehicle->getRegisterNumber()}."; ①
        }
```

```
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 véhicule.

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 fait 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 :

```
1 <?php
2
3
4 $vehicleBBBB = new Vehicle('BBBB');
5 $anna = new Technician('Anna');
6
7 //nous associons le technicien au véhicule (navigabilité de Vehicle vers
  Technician)
8 $vehicleBBBB->setTechnician($anna);
9
10 //nous associons également le véhicule au technicien (navigabilité de Technician
  vers Vehicle)
11 $anna->setVehicle($vehicleBBBB);
12
13 //Nous pouvons naviguer de Vehicle vers Technician :
14 echo $vehicleBBBB;
15
16 //Nous pouvons naviguer de Technician vers Vehicle :
```

```php
17 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 :

```php
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 :

```php
<?php




    /**
     * @param Technician|null $technician
     *
     * @return Vehicle
     */
    public function setTechnician(?Technician $technician): Vehicle
    {
        //mise à jour de l'objet lié (ici le technicien à qui l'on affecte la voiture courante $this)
        if (null !== $technician) {
            $technician->setVehicle($this);
        }
        //l'ancien technicien affecté au véhicule courant ne doit plus l'être
        if (null !== $this->technician) {
            $this->technician->setVehicle(null); ①
        }

        //on associe le nouveau technicien au véhicule
        $this->technician = $technician;

        return $this;
    }
```

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 :

```php
<?php


$vehicleEEEE = new Vehicle('EEEE');
$cedric = new Technician('Cédric');

```

```php
//une seule affectation depuis la voiture
$vehicleEEEE->setTechnician($cedric);

//la navigabilité est possible dans les deux sens
var_dump($vehicleEEEE->getTechnician()); //Cédric
var_dump($cedric->getVehicle()); // EEEE

//le véhicule est associé à un nouveau technicien
$karl = new Technician('Karl');
$vehicleEEEE->setTechnician($karl);

//le nouveau technicien est bien lié au véhicule (bidirectionnelle ok)
var_dump($karl->getVehicle()); // EEEE

//l'ancien technicien n'est plus lié au véhicule EEEE
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 :

Correction de Q7

a. La classe propriétaire peut être `Vehicle` ou `Technician`. Les cardinalités maximales de chaque association sont à 1. Si l'on "travaille" à partir des véhicules, alors il est préférable d'utiliser `Vehicle` comme classe possédante. S'il est plus logique de partir du technicien pour lui associer un véhicule, alors la classe possédante sera `Technician`

b. La classe propriétaire est `vechicle` car un véhicule est lié à 1 technicien maximum.

c. La classe propriétaire peut être l'une ou l'autre des deux classes. Il faut voir s'il est plus logique de partir du technicien pour lui associer un véhicule, ou de partir d'un véhicule pour lui associer un technicien.

d. Les cardinalités maximales sont à 1 de chaque côté de l'association. On peut penser que l'une ou l'autre classe peut être la classe propriétaire. Cependant, comme un objet de type `Technician` sera toujours lié à un véhicule, c'est la classe `Technician` qui sera la classe propriétaire, tout simplement parce que le lien est plus fort dans ce sens puisqu'il est obligatoire.

e. La classe `Technician` est la classe propriétaire car elle est liée à une seule instance de `Vehicule` au maximum.

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 navigabilitié de `Vehicule` vers `Technician` (en utilisant un attribut permettant de contenir une collection)
2. Mettre en place la navigabilitié 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 :
- Testez votre implémentation en répondant aux questions suivantes :

a. Créez deux instances de véhicules respectivement référencées par les variables `$vA` et `$vB`.

b. Créez trois instances de technicien respectivement référencées par les variables `$t1`, `$t2` et `$t3`.

c. Associez le véhicule A aux techniciens t1 et t2 et le véhicule B au technicien t3.

d. Faire un `var_dump()` de chaque véhicule et repérer l'identifiant de ressource associé à chaque instance (un identifiant de ressource est noté `#` suivi d'un numéro. Cet identifiant est propre à chaque instance)

e. Pour chaque identifiant véhicule, listez les identifiants des techniciens associés.

f. Associez le véhicule B aux techniciens t1 et t2.

g. A ce stade, à combien de technicien doit être associé le véhicule A ? Et le véhicule B ?

h. Faire un `var_dump()` de chaque véhicule et repérer l'identifiant de ressource associé à chaque instance

i. Pour chaque identifiant véhicule, listez les identifiants des techniciens associés afin de valider la réponse apportée à la question "g".

Correction de Q8

- Nous sommes dans le cas d'une navigabilité bidirectionnelle. Il faut choisir la classe propriétaire. Ce sera la classe `Technician` car elle est associée à `Vehicle` avec une cardinalité maximale à 1 alors que le lien dans l'autre sens est multiple.
- Voici le code de la classe `Vehicule` (navigable vers `Technician` avec son attribut `technicians` permettant de stocker une collection d'objets `Technician`).

Cette correction reprend tous les concepts que nous avons abordés dans les parties précédentes.

```php
<?php


class Vehicle
{
    public function __construct(
        private array $technicians = [],
    ) {

        $this->setTechnicians($this->technicians);
    }

    /**
     * @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;
```

```php
    }

    /**
     * @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
    {

        foreach ($technicians as $technician) {
            $this->addTechnician($technician);
        }

        return $this;
    }

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


}
```

- Classe `Technician` (navigable vers `Vehicule`)

```php
<?php

```

```php
class Technician
{
    public function __construct(
        private ?Vehicle $vehicle = null,
    ) {
    }

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

    /**
     * @param Vehicle|null $vehicle
     *
     * @return Technician
     */
    public function setVehicle(?Vehicle $vehicle): Technician
    {

        //on met à jour le nouveau véhicule associé au technicien courant $this
        $this->vehicle = $vehicle;


        return $this;
    }
}
```

- Mise à jour depuis la classe propriétaire `Technician` de l'objet inverse de type `Vehicle`.

```php
//class Technician
<?php

    public function setVehicle(?Vehicle $vehicle): Technician
    {
        //maj de l'objet inverse véhicule
        if (null !== $vehicle) {
            $vehicle->addTechnician($this);
        }

        //le véhicule précédemment associé au technician courant $this ne doit plus être associé à lui
        if (null !== $this->vehicle) {
            $this->vehicle->removeTechnician($this);
        }

```

```
16         //on met à jour le nouveau véhicule associé au technicien courant
   $this
17         $this->vehicle = $vehicle;
18
19
20         return $this;
21     }
```

- Réponses aux questions a) à i) :

```
1  <?php
2
3
4
5  //a) Créez deux instances de véhicules respectivement référencées par les
   variables `$vA` et `$vB`.
6  $vA = new Vehicle();
7  $vB = new Vehicle();
8
9  //b) Créez trois instances de technicien respectivement référencées par les
   variables `$t1`, `$t2` et `$t3`.
10 $t1 = new Technician();
11 $t2 = new Technician();
12 $t3 = new Technician();
13
14 //c) Associez le véhicule A aux techniciens t1 et t2 et le véhicule B au
   technicien t3.
15
16 //la classe propriétaire est Technician, les affectations doivent avoir lieu
   depuis les instances de celle-ci
17 $t1->setVehicle($vA);
18 $t2->setVehicle($vA);
19 $t3->setVehicle($vB);
20
21 //d) Faire un `var_dump()` de chaque véhicule et repérer l'identifiant de
   ressource associé à chaque instance (un identifiant de ressource est noté `#`
   suivi d'un numéro. Cet identifiant est propre à chaque instance)
22 var_dump($vA); //#1 pour le véhicule A
23 var_dump($vB); //#2 pour le véhicule B
24
25 //e) Pour chaque identifiant véhicule, listez les identifiants des techniciens
   associés.
26 //véhicule #1 -- #3, #4
27 //véhicule #2 -- #5
28
29 //f) Associez le véhicule B aux techniciens t1 et t2.
30 $t1->setVehicle($vB);
31 $t2->setVehicle($vB);
32
33 //g) A ce stade, à combien de technicien doit être associé le véhicule A ? Et
```

```
   le véhicule B ?
34 //A est associé aux trois techniciens et B à aucun.
35
36 //h) Faire un `var_dump()` de chaque véhicule et repérer l'identifiant de
   ressource associé à chaque instance
37 var_dump($vA); //#1 pour le véhicule A
38 var_dump($vB); //#2 pour le véhicule B
39
40 //i) Pour chaque identifiant véhicule, listez les identifiants des techniciens
   associés afin de valider la réponse apportée à la question "g".
41 //véhicule #1 -- #5, #3, #4
42 //véhicule #2 -- aucun
```

Q9) Travail à faire

- Implémentez scrupuleusement le diagramme suivant :