# UML

Baptiste Bauer

Version v0.0.7, 2022-11-23 09:29:39

# Table des matières

# 1. Implémentation des cardinalités

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

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

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

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

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

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

**Q1)** Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ? Si oui, indiquez pour chacune d'elle si le contrôle doit être fait dès l'instanciation de l'objet depuis lequel commence la navigabilité ou après (dans ce cas préciser depuis quelle méthode).


*Correction de Q1*

- Il n'y a qu'une seule cardinalité à contrôler. Il s'agit de la cardinalité minimale à 1 (une table est servie par un à plusieurs serveurs).

Cette cardinalité doit être contrôlée dès l'instanciation de `Table` car une table doit toujours être associée au minimum à un serveur. Il n'est pas possible de créer une table puis de lui affecter un serveur. Cela irait à l'encontre du diagramme.

**Q2)** Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?


*Correction de Q2*

Il n'y a aucune cardinalité à contrôler car l'association n'est pas navigable de `Table` vers `Waiter`. Donc il n'y a pas à se soucier du nombre de serveurs associés à une table.

**Q3)** Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?


*Correction de Q3*

Il y a une cardinalité à contrôler. Il s'agit de la cardinalité qui fixe à 4 le nombre maximum de tables qu'un serveur peut servir. La cardinalité à 1 concerne l'association non navigable. Elle n'est donc pas à contrôler.

La cardinalité à contrôler doit l'être à chaque fois que l'on ajoute un élément dans la collection de tables associée au serveur.

**Q4)** Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?


*Correction de Q4*

Il y a deux cardinalités à contrôler :

- la cardinalité qui fixe à 4 le nombre maximum de tables qu'un serveur peut servir
- la cardinalité minimale qui indique qu'une table doit être servie par au moins un serveur (et cela dès son instanciation)

La cardinalité minimale 1 doit être contrôlée dès l'instanciation d'une table. Une table doit dès son instanciation être associée à un serveur. Il n'est pas possible de créer une table puis de lui associer un serveur. Cela irait à l'encontre de ce que précise le diagramme.

**Q5)** Implémentez le diagramme suivant :

*Correction de Q5*

- Code de la classe `Table`

```php
<?php

class Table
{

}
```

- Code de la classe `Waiter` avec la navigabilité vers `Table` sans contrôle de la limite de 4 tables maximum :

```php
<?php


class Waiter
{


    public function __construct(
        private array $tables = []
    )
    {
    }


    /**
     * @param Table $table retire l'item de la collection
     */
    public function removeTable(Table $table): bool
    {
        $key = array_search($table, $this->tables, true);

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

            return true;
        }

        return false;
    }

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

```
35      *
36      * @return $this
37      */
38     public function setTables(array $tables): self
39     {
40
41         foreach ($tables as $table) {
42             $this->addTable($table);
43         }
44
45         return $this;
46     }
47
48     /**
49      * @return Table[]
50      */
51     public function getTables(): array
52     {
53         return $this->tables;
54     }
55
56 }
```

- On définit une propriété qui permet de fixer une limite de tables pouvant être affectées à un serveur. Comme cette limite ne dépend pas des instances de `Waiter`, c'est une propriété de classe. Elle n'est pas destinée à évoluer au cours du script ce qui nous oriente vers l'utilisation d'une constante :

```
1 //A ajouter dans la classe Waiter
2 <?php
3
4
5     //nombre maximum de tables pouvant être affectées à un serveur
6     const MAX_TABLES = 4;
```

- Cette constante peut être utilisée dans la méthode `addTable` afin de vérifier que la limite n'est pas dépassée :

```
1 //A ajouter dans la classe Waiter
2 <?php
3
4
5     /**
6      * @param Table $table ajoute un item de type Table à la collection
7      */
8     public function addTable(Table $table): bool
9     {
10        //on vérifie que le nombre maximum de tables n'est pas déjà atteint
```

```php
        if (count($this->tables) === self::MAX_TABLES) {
            return false;
        }

        if (!in_array($table, $this->tables, true)) {
            $this->tables[] = $table;

            return true;
        }

        return false;
    }
```

Avec ce code, il ne sera pas possible d'affecter plus de 4 tables.

**Q6)** Implémentez le diagramme suivant :


*Correction de Q6*

Le contrôle porte sur le nombre de serveurs minimum devant être affectés à une table.

- Code de la classe `Waiter` (il est fait le choix de lever une exception lorsque la contrainte n'est pas respectée):

```php
<?php


class Waiter
{
}
```

- Code de la classe `Table` :

```php
<?php

class Table
{
//on définit un nombre de serveurs minimum à affecter à une table
    const MIN_WAITERS_BY_TABLE = 1;

    public function __construct(
        //il faut forcer le passage d'un  tableau à l'instanciation afin de
```

```
contrôler s'il contient au moins une instance de Waiter
        private array $waiters ①
    )
    {
        $this->setWaiters($waiters);
    }

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

            return true;
        }

        return false;
    }

    /**
     * @param Waiter $waiter retire l'item de la collection
     */
    public function removeWaiter(Waiter $waiter): bool
    {
        $key = array_search($waiter, $this->waiters, true);

        if ($key !== false) {

            //on ne tente pas de supprimer le serveur s'il n'y en a qu'un seul
dans la collection
            if(count($this->waiters) === self::MIN_WAITERS_BY_TABLE){ ②
                throw new Exception("Une table doit être associée à au moins
{self::MIN_WAITERS_BY_TABLE} serveur(s)");
            }

            unset($this->waiters[$key]);

            return true;
        }

        return false;
    }

    /**
     * Initialise la collection avec la collection passée en argument
     *
     * @param array $waiters collection d'objets de type Waiter
     *
     * @return $this
```

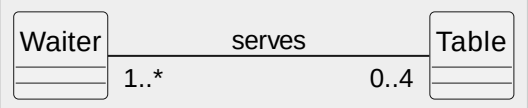
```php
     */
    public function setWaiters(array $waiters): self
    {

        foreach ($waiters as $waiter) {
            $this->addWaiter($waiter);
        }

        //on contrôle qu'il y a au moins un serveur dans la collection
        if(count($this->waiters) < self::MIN_WAITERS_BY_TABLE){ ③
            throw new Exception("Une table doit être associée à au moins {self::MIN_WAITERS_BY_TABLE} serveur(s)");
        }

        return $this;
    }

    /**
     * @return Waiter[]
     */
    public function getWaiters(): array
    {
        return $this->waiters;
    }

}
```

① Aucune valeur par défaut n'est utilisée car il faut forcer l'utilisateur de la classe `Table` à passer une collection. Celle-ci est ensuite passée à la méthode `setWaiters()` de façon à contrôler son contenu.

② On vérifie le nombre d'items dans la collection avant de procéder à la suppression. Cela pourrait être fait avant de rechercher si l'item est dans la collection mais il est plus pertinent de lever l'exception si le serveur à supprimer est effectivement affecté à la table courante.

③ Une fois la collection mise à jour, on compte le nombre d'items afin de s'assurer que la contrainte est respectée.

**Q7)** Implémentez le diagramme suivant :




*Correction de Q7*

Nous sommes dans le cas d'une navigation bidirectionnelle. Il est donc nécessaire de choisir la classe propriétaire. Les cardinalités maximales sont multiples de chaque côté de

l'association. Ainsi, nous devons choisir la classe depuis laquelle il est plus naturel de faire l'association. Ce sera la classe `Table` car elle est forcément liée à un serveur (cardinalité minimale à 1).

Dans les exercices précédents, nous avons mis en place les navigabilités pour chaque sens de lecture de l'association. Il suffit dans cet exercice de reprendre le code que nous avons écrit. Ce code contient déjà les contrôles des cardinalités. Je ne reviens pas dessus.

Voici le code de la classe `Waiter` :

```php
<?php

class Waiter
{

    //nombre maximum de tables pouvant être affectées à un serveur
    const MAX_TABLES = 4;

    public function __construct(
        private array $tables = []
    )
    {
    }

    /**
     * @param Table $table ajoute un item de type Table à la collection
     */
    public function addTable(Table $table): bool
    {
        //on vérifie que le nombre maximum de tables n'est pas déjà atteint
        if (count($this->tables) === self::MAX_TABLES) {
            return false;
        }

        if (!in_array($table, $this->tables, true)) {
            $this->tables[] = $table;

            return true;
        }

        return false;
    }

    /**
     * @param Table $table retire l'item de la collection
     */
    public function removeTable(Table $table): bool
    {
        $key = array_search($table, $this->tables, true);

```

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

            return true;
        }

        return false;
    }

    /**
     * Initialise la collection avec la collection passée en argument
     *
     * @param array $tables collection d'objets de type Table
     *
     * @return $this
     */
    public function setTables(array $tables): self
    {

        foreach ($tables as $table) {
            $this->addTable($table);
        }

        return $this;
    }

    /**
     * @return Table[]
     */
    public function getTables(): array
    {
        return $this->tables;
    }

}
```

Voici le code de la classe `Table` qui est celui de l'exercice précédent auquel est ajouté la mise à jour des objets inverses :

```php
<?php

class Table
{
//on définit un nombre de serveurs minimum à affecter à une table
    const MIN_WAITERS_BY_TABLE = 1;

    public function __construct(
        //il faut forcer le passage d'un  tableau à l'instanciation afin de contrôler s'il contient au moins une instance de Waiter
        private array $waiters
```

```
11      )
12      {
13          $this->setWaiters($waiters);
14      }
15
16      /**
17       * @param Waiter $waiter ajoute un item de type Waiter à la collection
18       */
19      public function addWaiter(Waiter $waiter): bool
20      {
21
22          if (!in_array($waiter, $this->waiters, true)) {
23              $this->waiters[] = $waiter;
24
25              //maj de l'objet inverse
26              $waiter->addTable($this); ①
27
28              return true;
29          }
30
31          return false;
32      }
33
34      /**
35       * @param Waiter $waiter retire l'item de la collection
36       */
37      public function removeWaiter(Waiter $waiter): bool
38      {
39          $key = array_search($waiter, $this->waiters, true);
40
41          if ($key !== false) {
42
43              //on ne tente pas de supprimer le serveur s'il n'y en a qu'un seul
    dans la collection
44              if(count($this->waiters) === self::MIN_WAITERS_BY_TABLE){
45                  throw new Exception("Une table doit être associée à au moins
    {self::MIN_WAITERS_BY_TABLE} serveur(s)");
46              }
47
48              unset($this->waiters[$key]);
49
50              //maj de l'objet inverse
51              $waiter->removeTable($this); ①
52
53
54              return true;
55          }
56
57          return false;
58      }
59
```

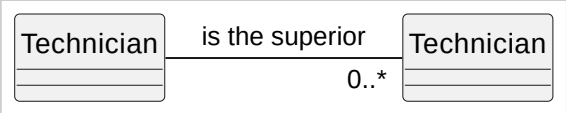
```php
    /**
     * Initialise la collection avec la collection passée en argument
     *
     * @param array $waiters collection d'objets de type Waiter
     *
     * @return $this
     */
    public function setWaiters(array $waiters): self
    {
        //maj des objets inverses de la collection AVANT son actualisation
        foreach($this->waiters as $waiter){
            $waiter->removeTable($this); ①
        }

        foreach ($waiters as $waiter) {
            $this->addWaiter($waiter);
        }

        //on contrôle qu'il y a au moins un serveur dans la collection
        if(count($this->waiters) < self::MIN_WAITERS_BY_TABLE){
            throw new Exception("Une table doit être associée à au moins {self::MIN_WAITERS_BY_TABLE} serveur(s)");
        }

        return $this;
    }

    /**
     * @return Waiter[]
     */
    public function getWaiters(): array
    {
        return $this->waiters;
    }

}
```

① L'objet inverse est mis à jour car nous sommes dans une navigabilité bidirectionnelle.

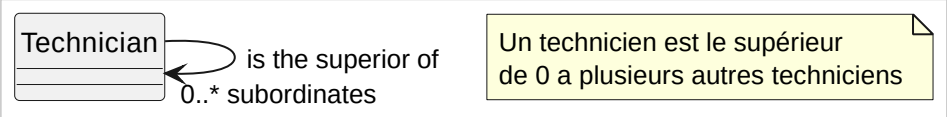
# 2. L'association réflexive

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

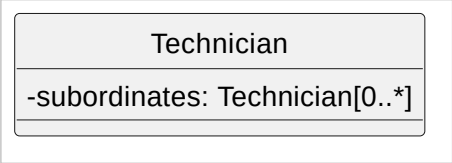
Imaginons un technicien qui peut être le supérieur hiérarchique d'autres techniciens. Le diagramme suivant illustre cette relation :



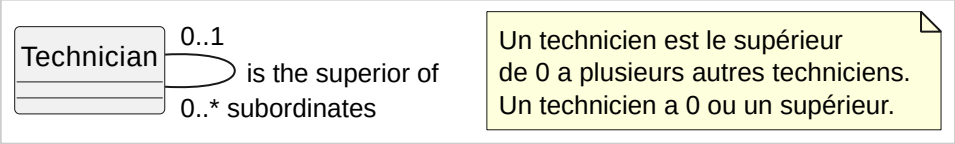

Comme les deux classes mobilisées sont identiques, il ne faut en utiliser qu'une seule et donc faire un lien qui point sur elle-même :



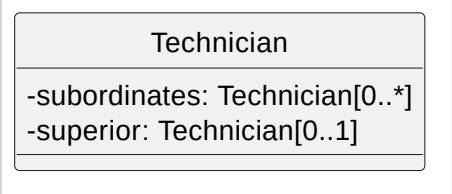

C'est équivalent à cette représentation :




Voici la même modélisation mais avec une bidirectionnalité :

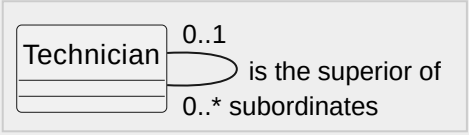



Ce qui est équivalent à :




**Q8)** Travail à faire

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




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

*Correction de Q8*

Nous allons implémenter le diagramme en plusieurs étapes :

1. Création de la classe `Tehchnician` avec navigabilité depuis un supérieur vers ses subordonnées (propriété de type collection nommée `subordinates`)
2. Ajout de la navigabilité du subordonné vers son supérieur
3. Ajout du contrôle sur l'impossibilité d'un technicien à être son propre subordonné ou supérieur

- Création de la classe `Technician` (navigabilité vers une collection de subordonnés) :

```php
<?php

class Technician
{
    public function __construct(
        private array $surbordinates = [],
    ) {
    }

    /**
     * Ajoute un subordonné au technicien courant
     * @param Technician $subordinate  ajoute un item de type Technician à la
     *                                 collection
     */
    public function addSubordinate(Technician $subordinate): bool
    {

        if (!in_array($subordinate, $this->subordinates, true)) {
            $this->subordinates[] = $subordinate;

            return true;
        }

        return false;
    }

    /**
     * Retire un subordonné au technicien courant
     * @param Technician $subordinate retire l'item de la collection
     */
    public function removeSubordinate(Technician $subordinate): bool
    {
        $key = array_search($subordinate, $this->subordinates, true);

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

```

```php
            return true;
        }

        return false;
    }

    /**
     * Initialise la collection de subordonnés du technicien courant
     *
     * @param array $subordinates collection d'objets de type Technician
     *
     * @return $this
     */
    public function setSubordinates(array $subordinates): self
    {

        foreach ($subordinates as $subordinate) {
            $this->addSubordinate($subordinate);
        }

        return $this;
    }

    /**
     * Retourne la collection de subordonnés du technicien courant
     * @return Technician[]
     */
    public function getSubordinates(): array
    {
        return $this->subordinates;
    }


}
```

- Ajout de la navigabilité vers un supérieur (ajout d'un attribut d'objet nommé `superior` pouvant stocker une instance de `Technician`:

```php
//Seule les mutateurs et accesseurs concernés par la navigabilité vers le technicien supérieur sont repris ici. Ce code est à ajouter au code précédent.
<?php

class Technician
{
    public function __construct(
        private array $surbordinates = [],
        private ?Technician $superior = null,
    ) {
    }

```

```
12      /**
13       * Retourne le supérieur du technicien courant
14       * @return Technician|null
15       */
16      public function getSuperior(): ?Technician
17      {
18          return $this->superior;
19      }
20
21      /**
22       * Affecte un supérieur au technicien courant
23       * @param Technician|null $superior
24       *
25       * @return Technician
26       */
27      public function setSuperior(?Technician $superior): Technician
28      {
29
30
31
32          $this->superior = $superior;
33
34          return $this;
35      }
36
37
38  }
```

- Il ne reste que la gestion de la mise à jour de la classe inverse à faire. Il faut déterminer la classe propriétaire. Nous savons que c'est la classe qui est à l'opposé de la cardinalité maximale à 1, soit ici la classe `Technician` (mais du côté du subordonné). Lorsqu'un subordonné sa voit lié ou "délié" d'un supérieur, il faut indiquer à ce supérieur qu'il a un nouveau subordonné ou qu'il en a un en moins.

```
1  <?php
2
3  class Technician
4  {
5      public function __construct(
6          private array $surbordinates = [],
7          private ?Technician $superior = null,
8      ) {
9      }
10
11     /**
12      * Retourne le supérieur du technicien courant
13      * @return Technician|null
14      */
15     public function getSuperior(): ?Technician
16     {
```

```
17          return $this->superior;
18      }
19
20      /**
21       * Affecte un supérieur au technicien courant
22       * @param Technician|null $superior
23       *
24       * @return Technician
25       */
26      public function setSuperior(?Technician $superior): Technician
27      {
28
29
30          //mise à jour de l'ancien supérieur ①
31          if (null !== $this->superior) {
32              $this->superior->removeSubordinate($this);
33          }
34
35          //mise à jour du nouveau supérieur ②
36          if (null !== $superior) {
37              $superior->addSubordinate($this);
38          }
39
40          $this->superior = $superior;
41
42          return $this;
43      }
44
45
46  }
```

① on met à jour l'ancien supérieur qui n'a plus le technicien courant comme subordonné

② on met à jour le nouveau supérieur en lui indiquant qu'il a un nouveau subordonné

- Il faut encore contrôler qu'un technician ne soit pas son propre supérieur :

```
1 //code à ajouter au début de la méthode "setSuperior"
2 <?php
3
4         //contrôle que le supérieur du technicien courant n'est pas lui-même
5         if ($superior === $this) {
6             throw new Exception(
7                 "Un technicien ne peut pas être son propre supérieur"
8             );
9         }
```

- Et pour terminer, il faut contrôler que le technicien ne soit pas son propre subordonné :

```
1 //code à ajouter au début de la méthode "addSubordinate"
```

```php
<?php

        //contrôle que le subordonné du technicien courant n'est pas lui-même
        if ($subordinate === $this) {
            throw new Exception(
                "Un technicien ne peut pas être son propre subordonné"
            );
        }
```

- Nous pouvons tester le code en affectant à un technicien un supérieur qui est lui-même et un subordonné qui est également lui-même :

```php
<?php

//Mise en oeuvre du contrôle qu'un technicien ne peut être ni son propre supérieur, ni son propre subordonné
$t = new Technician();
$t->setSuperior($t); //lève une exception : Un technicien ne peut pas être son propre supérieur !
$t->addSubordinate($t); //lève une exception : Un technicien ne peut pas être son propre subordonné !
```

# 3. L'agrégation

## 3.1. Qu'est-ce qu'une agrégation ?

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

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

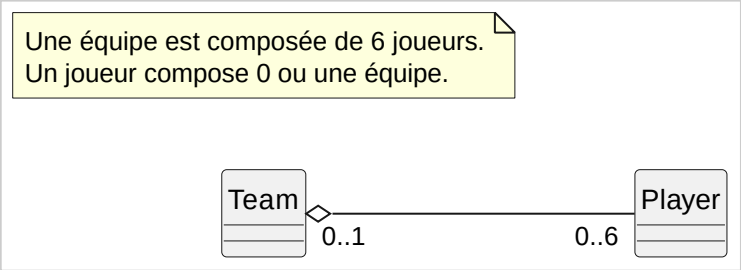
L'**agrégation** exprime la construction d'un objet à partir d'autres objets. Cela se distingue de la notion d'association que nous avons abordée jusqu'à maintenant. Effectivement, l'association traduit un lien entre deux objets alors que l'agrégation traduit le "regroupement" ou "l'assemblage" de plusieurs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

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

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

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

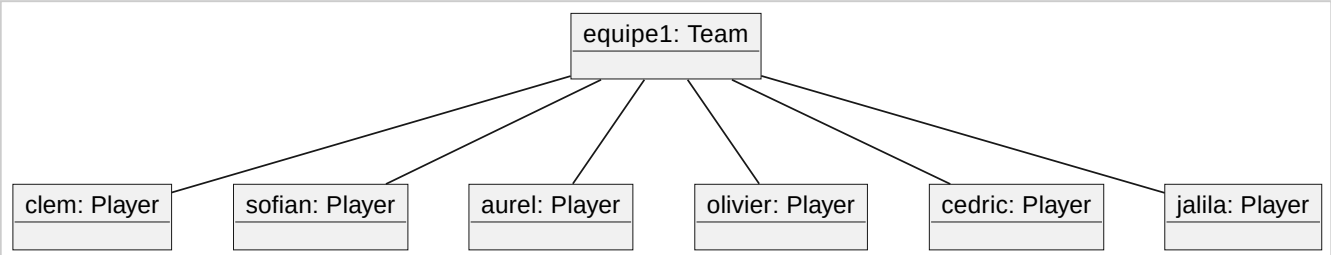
Représentons le lien entre une équipe et les joueurs qui composent celle-ci (ici une équipe de volley) :




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

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

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



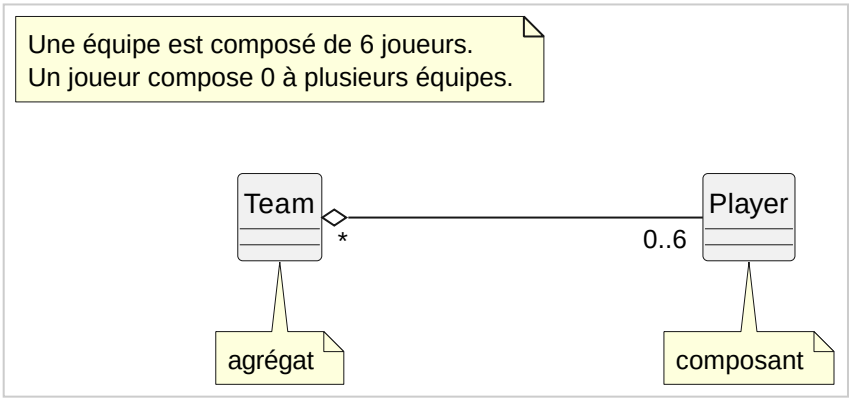
Une agrégation présente les caractéristiques suivantes :

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

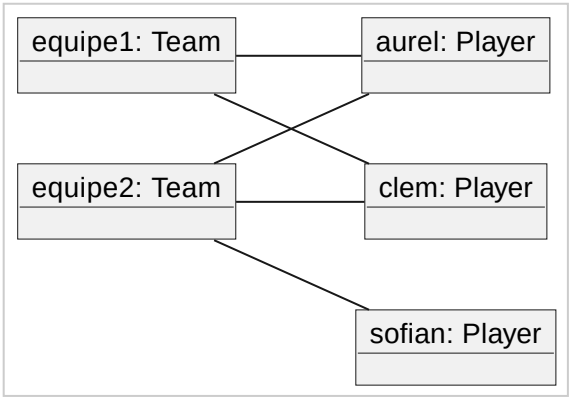
**Les composants sont partageables :**

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

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




Voici un diagramme objet pour illustrer ce partage :




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

Voici un autre exemple :


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

retrouvons la notion de partage des composants.

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


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

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

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

## 3.2. Navigabilité et agrégation

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


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

## 3.3. Implémentation d'une agrégation

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

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

# 4. La composition

## 4.1. Qu'est-ce qu'une composition ?

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

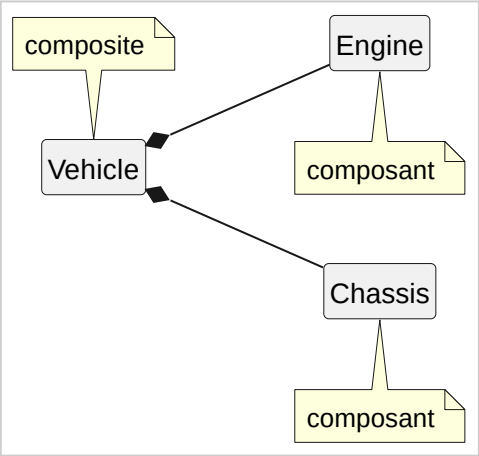
La composition reprend l'idée de l'agrégation. La composition exprime un assemblage d'objets. Cet assemblage est tellement "fort" que les objets assemblés ne peuvent pas servir dans un autre assemblage.

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

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

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

Voici le diagramme de classes qui représente cette situation :



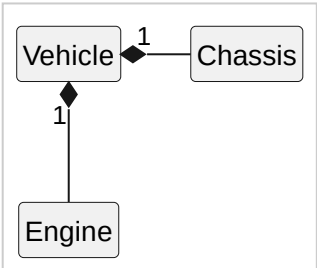

- L'objet composé d'éléments est appelé **composite**.
- L'objet qui compose le composite est un **composant**.
- Le losange **plein** est placé du côté du composite.

La composition présente donc les caractéristiques suivantes :

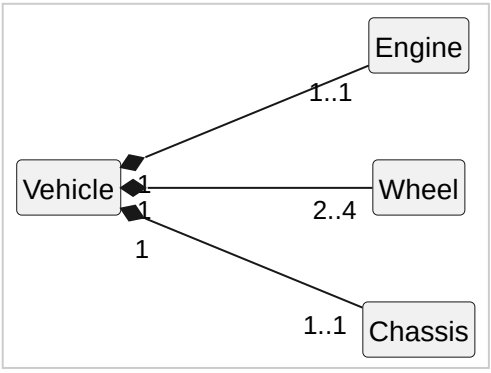
- **la destruction du composite détruit également ses composants**.
- **le composant ne peut pas être partagé** (c'est logique puisque si le composite qui le contient est

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

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




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

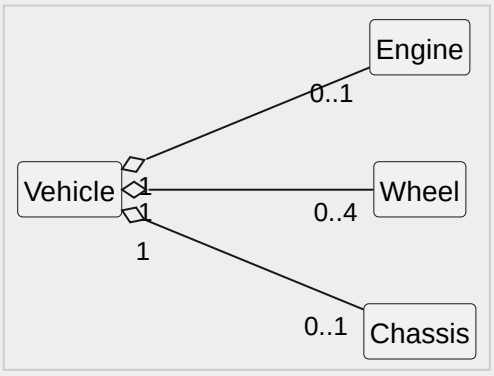



**Q9)** La modélisation précédente serait-elle la même pour une casse automobile ?

*Correction de Q9*

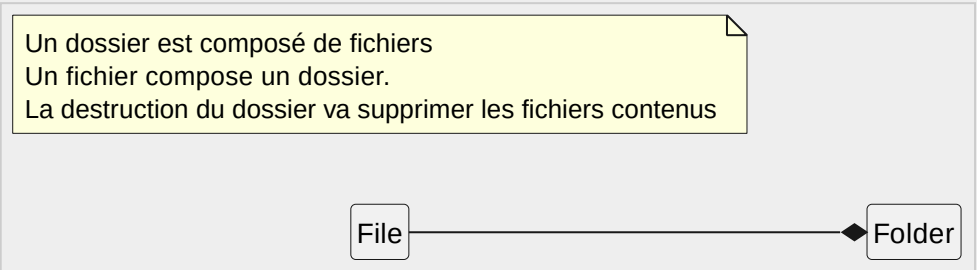
Dans une casse automobile, les éléments d'une voiture sont vus comme des éléments partageables. Effectivement, chaque partie peut être vendue sous forme d'élément ayant sa propre vie (un moteur est démonté pour être remonté dans une autre voiture, etc).

Une casse auto verrait plutôt cette modélisation :




Dès lors, on voit les cardinalités minimales qui peuvent être à zéro. Dans une casse, une voiture peut ne plus avoir de moteur.

**Q10)** A votre avis, peut-on modéliser le diagramme suivant ?




*Correction de Q10*

Il faut valider les caractéristiques de la composition :

- ☑ les composants (fichiers) ne sont pas partageables
- ☑ la destruction du composite (dossier) détruit ses composants (fichiers)
- ☐ le composite ne peut pas exister sans ses composants

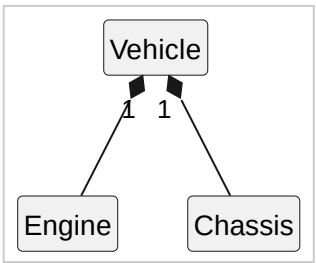
La dernière caractéristique n'est pas cochée. La bonne modélisation est donc une agrégation :


## 4.2. Navigabilité et composition

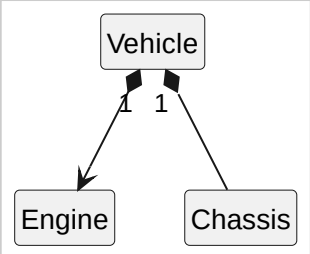
Tout ce que nous avons vu dans la partie sur la navigabilité s'applique dans le cadre d'une composition à une exception près : **Il est toujours possible de naviguer de la composition vers ses composants.** C'est logique puisque le composite "connait" ses composants dès sa création.

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

Le diagramme suivant nous indique qu'il est possible de naviguer de `Vehicle` vers `Engine` et de `Engine` vers `Vehicle`. Il en va de même avec le composant `Chassis`.




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




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

## 4.3. Implémentation d'une composition

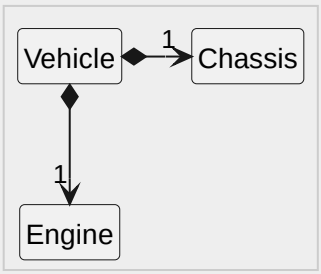
La composition nécessite une implémentation qui prend en compte ces caractéristiques :

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

**Q11)** Implémentez le diagramme de classes suivant compte tenu des notions qui ont été abordées dans cette partie.

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

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




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

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

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

*Correction de Q11* (affichée volontairement du fait de l'approche qui n'est pas toujours simple à "deviner")

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

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

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

- les 3 classes à implémenter :

```php
class Vehicle
{

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


    }

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

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


class Chassis
{

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

class Engine
{
    public function __destruct()
```

```
    {
        echo "moteur détruit\n";
    }
}
```

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

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

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

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

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

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

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

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

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

```
voiture détruite
chassis détruit
moteur détruit

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

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

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

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

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

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

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

```
voiture détruite
chassis détruit
moteur détruit

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

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

```
class Vehicle2
{

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

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

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

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

}
```

```php

$v3 = new Vehicle2();

unset($v3);

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

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

Nous arrivons au même résultat :

```
voiture détruite
chassis détruit
moteur détruit

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

Nous avons 3 solutions possibles, alors laquelle choisir ?

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

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

# Index