

UML

Baptiste Bauer

Version v0.0.6, 2022-11-21 20:07:38

Table des matières

1. Implémentation des cardinalités	1
2. L'association réflexive	12
3. L'agrégation	18
3.1. Qu'est-ce qu'une agrégation ?	18
3.2. Navigabilité et agrégation	20
3.3. Implémentation d'une agrégation	20
Index	21

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 expriment 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.

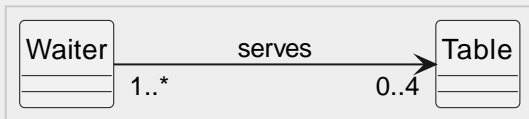
Q2) Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?



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**

```
1 <?php
2
3 class Table
4 {
5
6 }
```

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

```
1 <?php
2
3
4 class Waiter
5 {
6
7
8     public function __construct(
9         private array $tables = []
10    )
11    {
12    }
13
14
15    /**
16     * @param Table $table retire l'item de la collection
17     */
18    public function removeTable(Table $table): bool
19    {
20        $key = array_search($table, $this->tables, true);
21
22        if ($key !== false) {
23            unset($this->tables[$key]);
24
25            return true;
26        }
27
28        return false;
29    }
30
31    /**
32     * Initialise la collection avec la collection passée en argument
33     *
34     * @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

```

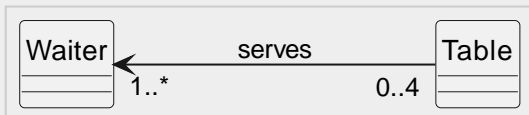
```

11     if (count($this->tables) === self::MAX_TABLES) {
12         return false;
13     }
14
15     if (!in_array($table, $this->tables, true)) {
16         $this->tables[] = $table;
17
18         return true;
19     }
20
21     return false;
22 }

```

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):

```

1 <?php
2
3
4 class Waiter
5 {
6 }

```

- Code de la classe **Table** :

```

1 <?php
2
3 class Table
4 {
5 //on définit un nombre de serveurs minimum à affecter à une table
6     const MIN_WAITERS_BY_TABLE = 1;
7
8     public function __construct(
9         //il faut forcer le passage d'un tableau à l'instanciation afin de

```

```
    contrôler s'il contient au moins une instance de Waiter
10     private array $waiters ①
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 $waiter): bool
20 {
21     if (!in_array($waiter, $this->waiters, true)) {
22         $this->waiters[] = $waiter;
23
24         return true;
25     }
26
27     return false;
28 }
29
30 /**
31  * @param Waiter $waiter retire l'item de la collection
32  */
33 public function removeWaiter(Waiter $waiter): bool
34 {
35     $key = array_search($waiter, $this->waiters, true);
36
37     if ($key !== false) {
38
39         //on ne tente pas de supprimer le serveur s'il n'y en a qu'un seul
        dans la collection
40         if(count($this->waiters) === self::MIN_WAITERS_BY_TABLE){ ②
41             throw new Exception("Une table doit être associée à au moins
        {self::MIN_WAITERS_BY_TABLE} serveur(s)");
42         }
43
44         unset($this->waiters[$key]);
45
46         return true;
47     }
48
49     return false;
50 }
51
52 /**
53  * Initialise la collection avec la collection passée en argument
54  *
55  * @param array $waiters collection d'objets de type Waiter
56  *
57  * @return $this
```



```

58     */
59     public function setWaiters(array $waiters): self
60     {
61
62         foreach ($waiters as $waiter) {
63             $this->addWaiter($waiter);
64         }
65
66         //on contrôle qu'il y a au moins un serveur dans la collection
67         if(count($this->waiters) < self::MIN_WAITERS_BY_TABLE){ ③
68             throw new Exception("Une table doit être associée à au moins
        {self::MIN_WAITERS_BY_TABLE} serveur(s)");
69         }
70
71         return $this;
72     }
73
74     /**
75     * @return Waiter[]
76     */
77     public function getWaiters(): array
78     {
79         return $this->waiters;
80     }
81
82 }

```

- 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** :

```
1 <?php
2
3 class Waiter
4 {
5
6     //nombre maximum de tables pouvant être affectées à un serveur
7     const MAX_TABLES = 4;
8
9     public function __construct(
10         private array $tables = []
11     )
12     {
13     }
14
15     /**
16      * @param Table $table ajoute un item de type Table à la collection
17      */
18     public function addTable(Table $table): bool
19     {
20         //on vérifie que le nombre maximum de tables n'est pas déjà atteint
21         if (count($this->tables) === self::MAX_TABLES) {
22             return false;
23         }
24
25         if (!in_array($table, $this->tables, true)) {
26             $this->tables[] = $table;
27
28             return true;
29         }
30
31         return false;
32     }
33
34     /**
35      * @param Table $table retire l'item de la collection
36      */
37     public function removeTable(Table $table): bool
38     {
39         $key = array_search($table, $this->tables, true);
40     }
```

```

41     if ($key !== false) {
42         unset($this->tables[$key]);
43
44         return true;
45     }
46
47     return false;
48 }
49
50 /**
51  * Initialise la collection avec la collection passée en argument
52  *
53  * @param array $tables collection d'objets de type Table
54  *
55  * @return $this
56  */
57 public function setTables(array $tables): self
58 {
59
60     foreach ($tables as $table) {
61         $this->addTable($table);
62     }
63
64     return $this;
65 }
66
67 /**
68  * @return Table[]
69  */
70 public function getTables(): array
71 {
72     return $this->tables;
73 }
74
75 }

```

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

```

1 <?php
2
3 class Table
4 {
5     //on définit un nombre de serveurs minimum à affecter à une table
6     const MIN_WAITERS_BY_TABLE = 1;
7
8     public function __construct(
9         //il faut forcer le passage d'un tableau à l'instanciation afin de
        contrôler s'il contient au moins une instance de Waiter
10         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
44             //dans la collection
45             if(count($this->waiters) === self::MIN_WAITERS_BY_TABLE){
46                 throw new Exception("Une table doit être associée à au moins
47                 {self::MIN_WAITERS_BY_TABLE} serveur(s)");
48             }
49
50             unset($this->waiters[$key]);
51
52             //maj de l'objet inverse
53             $waiter->removeTable($this); ①
54
55             return true;
56         }
57
58         return false;
59     }
```

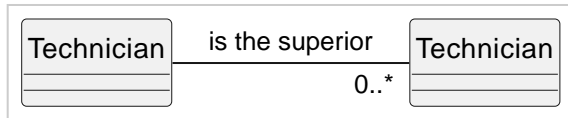
```
60  /**
61   * Initialise la collection avec la collection passée en argument
62   *
63   * @param array $waiters collection d'objets de type Waiter
64   *
65   * @return $this
66   */
67  public function setWaiters(array $waiters): self
68  {
69      //maj des objets inverses de la collection AVANT son actualisation
70      foreach($this->waiters as $waiter){
71          $waiter->removeTable($this); ❶
72      }
73
74      foreach ($waiters as $waiter) {
75          $this->addWaiter($waiter);
76      }
77
78      //on contrôle qu'il y a au moins un serveur dans la collection
79      if(count($this->waiters) < self::MIN_WAITERS_BY_TABLE){
80          throw new Exception("Une table doit être associée à au moins
81      {self::MIN_WAITERS_BY_TABLE} serveur(s)");
82      }
83
84      return $this;
85  }
86  /**
87   * @return Waiter[]
88   */
89  public function getWaiters(): array
90  {
91      return $this->waiters;
92  }
93
94 }
```

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

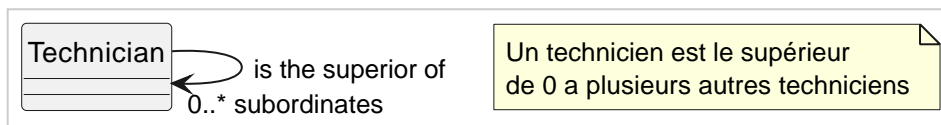
2. L'association réflexive

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

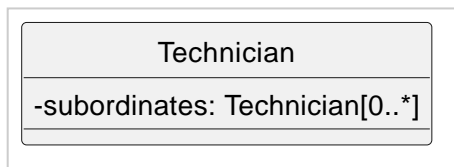
Imaginons un technicien qui peut être le supérieur hiérarchique d'autres techniciens. Le diagramme suivant illustre cette relation :



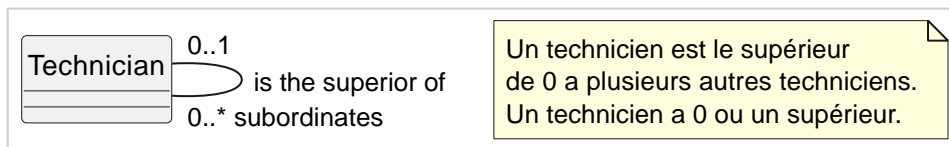
Comme les deux classes mobilisées sont identiques, il ne faut en utiliser qu'une seule et donc faire un lien qui point sur elle-même :



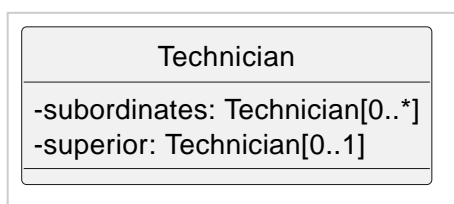
C'est équivalent à cette représentation :



Voici la même modélisation mais avec une bidirectionnalité :

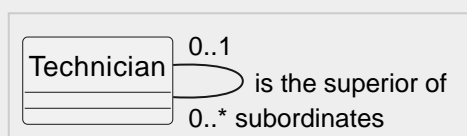


Ce qui est équivalent à :



Q8) Travail à faire

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



- 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 subordonnés (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) :

```
1 <?php
2
3 class Technician
4 {
5     public function __construct(
6         private array $subordinates = [],
7     ) {
8     }
9
10    /**
11     * Ajoute un subordonné au technicien courant
12     * @param Technician $subordinate ajoute un item de type Technician à la
13     * collection
14     */
15    public function addSubordinate(Technician $subordinate): bool
16    {
17
18        if (!in_array($subordinate, $this->subordinates, true)) {
19            $this->subordinates[] = $subordinate;
20
21            return true;
22        }
23
24        return false;
25    }
26
27    /**
28     * Retire un subordonné au technicien courant
29     * @param Technician $subordinate retire l'item de la collection
30     */
31    public function removeSubordinate(Technician $subordinate): bool
32    {
33        $key = array_search($subordinate, $this->subordinates, true);
34
35        if ($key !== false) {
36            unset($this->subordinates[$key]);
37        }
```

```

38         return true;
39     }
40
41     return false;
42 }
43
44 /**
45  * Initialise la collection de subordonnés du technicien courant
46  *
47  * @param array $subordinates collection d'objets de type Technician
48  *
49  * @return $this
50  */
51 public function setSubordinates(array $subordinates): self
52 {
53
54     foreach ($subordinates as $subordinate) {
55         $this->addSubordinate($subordinate);
56     }
57
58     return $this;
59 }
60
61 /**
62  * Retourne la collection de subordonnés du technicien courant
63  * @return Technician[]
64  */
65 public function getSubordinates(): array
66 {
67     return $this->subordinates;
68 }
69
70
71 }

```

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

```

1 //Seule les mutateurs et accesseurs concernés par la navigabilité vers le
  technicien supérieur sont repris ici. Ce code est à ajouter au code précédent.
2 <?php
3
4 class Technician
5 {
6     public function __construct(
7         private array $subordinates = [],
8         private ?Technician $superior = null,
9     ) {
10    }
11

```



```

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 se 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 $subordinates = [],
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 technicien 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"

```

```
2 <?php
3
4     //contrôle que le subordonné du technicien courant n'est pas lui-même
5     if ($subordinate === $this) {
6         throw new Exception(
7             "Un technicien ne peut pas être son propre subordonné"
8         );
9     }
```

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

```
1 <?php
2
3 //Mise en oeuvre du contrôle qu'un technicien ne peut être ni son propre
  supérieur, ni son propre subordonné
4 $t = new Technician();
5 $t->setSuperior($t); //lève une exception : Un technicien ne peut pas être son
  propre supérieur !
6 $t->addSubordinate($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.

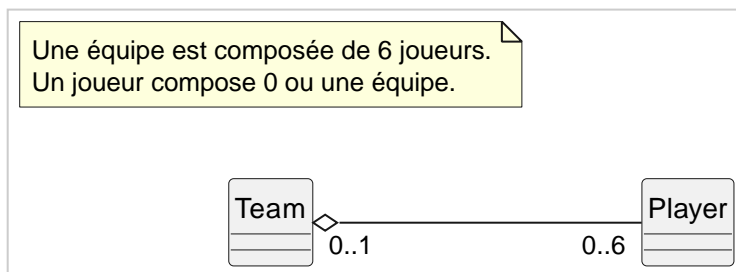
L'**agrégation** exprime la construction d'un objet à partir d'autres objets. Cela se distingue de la notion d'association que nous avons abordée jusqu'à maintenant. Effectivement, l'association traduit un lien entre deux objets alors que l'agrégation traduit le "regroupement" ou "l'assemblage" de plusieurs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

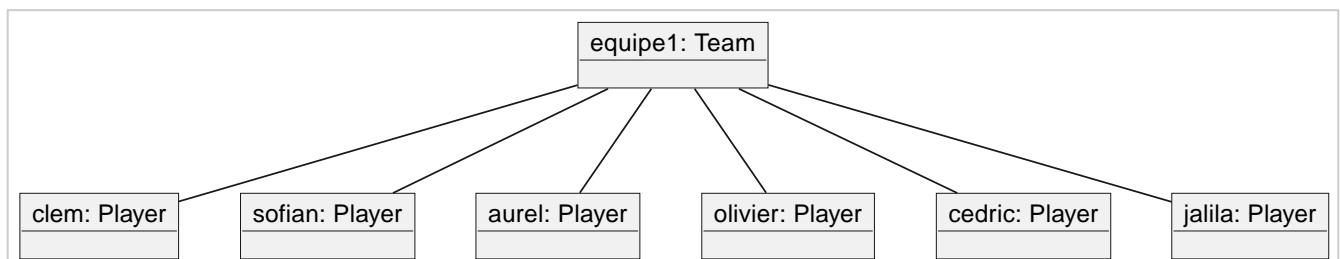
Représentons le lien entre une équipe et les joueurs qui composent celle-ci (ici une équipe de volley) :



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





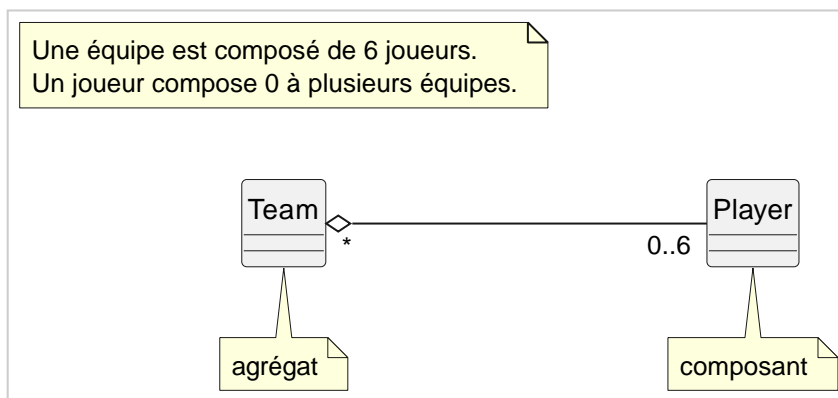
Une agrégation présente les caractéristiques suivantes :

- 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**)

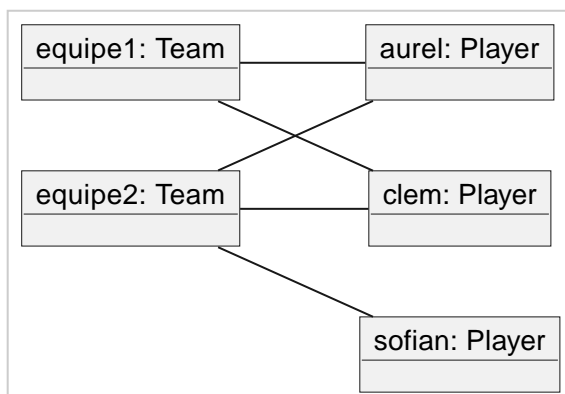
Les composants sont partageables :

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 supérieure 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.

Index

A

agrégat, [18](#)

agrégation, [18](#)

C

Caractéristiques d'une agrégation, [19](#)

composant, [18](#)