UML

Baptiste Bauer

Version v0.0.4, 2022-11-21 19:31:21

Table des matières

1. 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.

Q1) Pour chaque diagramme, exprimez la relation en prenant en compte les cardinalités.

a.)

b.)

c.)

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

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

Q2) 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.)

Q3)

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.

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

2. 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.

Q4) 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).

Q5) Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?

Q6) Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?

Q7) Dans le diagramme ci-dessous, y a-t-il des cardinalités à contrôler ?

Q8) Implémentez le diagramme suivant :

Q9) Implémentez le diagramme suivant :

Q10) Implémentez le diagramme suivant :

3. 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 :

Q11) 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.

4. L'agrégation

4.1. 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 :

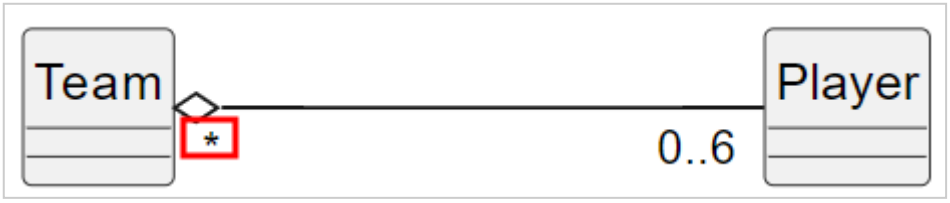
Voici un diagramme objet pour illustrer ce partage :

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

4.2. 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`.

4.3. 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.

5. La composition

5.1. 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)

> **Q12)** La modélisation précédente serait-elle la même pour une casse automobile ?

Q13) A votre avis, peut-on modéliser le diagramme suivant ?

5.2. 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`.

5.3. 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

Q14) 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 Q14 (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";
```

```
    }

}


$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