

## Introducción a agregados - Problemas de El Mundo Real™

Vamos a continuar con el ejemplo real de esta plataforma para introducir el concepto de Agregados. Recordamos que partíamos de una entidad base `Course`, desde el punto de vista del usuario final un curso se compone de:

- Rating del curso
- Resumen
- Descripción
- Total de steps
- Total de videos
- Lecciones
- Steps

Por tanto, nuestro `Course` a nivel de Entidad va a necesitar una serie de elementos que hemos definido como Value Objects: `Id`, `Rating`, `Summary` y `Description`, así como un conjunto de Entidades `Lessons`

Todo esto podría quedar de la siguiente forma a nivel de código:

Clase `Course`:

```
final class Course
{
    private $id;
    private $rating;
    private $summary;
    private $description;
    private $lessons;

    public function __construct(
        CourseId $id,
        CourseRating $rating,
        CourseSummary $summary,
        CourseDescription $description,
        Lessons ...$lessons)
    {
        $this->id = $id;
        $this->rating = $rating;
        $this->summary = $summary;
        $this->description = $description;
        $this->lessons = $lessons;
    }

    public function totalSteps(): CourseTotalSteps
    {
        $totalSteps = new CourseTotalSteps();

        foreach ($this->lessons as $lesson)
        {
            $totalSteps = $totalSteps->add($lesson->totalSteps())
        }
    }
}
```

Clase `Lessons`:

```
public class Lesson
{
    // ...
    public function totalSteps(): LessonTotalSteps
    {
        return new LessonTotalSteps(count($this->steps));
    }
}
```

A parte de recibir los Value Objects por parámetro en el constructor y un conjunto de `Lessons`, una cuestión importante en esta clase es que estamos haciendo uso del principio **Tell don't ask** en esta función `totalSteps()`. De este modo, evitamos exponer `Lessons` utilizando un Getter y los clientes no se verán afectados si por cualquier motivo decidiésemos cambiar el modo en que se calcula este `totalSteps()`. Además, esta primera iteración de cambio nos estaría llevando a

evitar modelos anémicos aplicando un modelado del dominio rico

Una siguiente mejora respecto a éste cálculo podría ser llevármolo al propio constructor de la clase, con sus propias cláusulas de guarda (Tal como vimos en el video anterior), asegurándonos de que no habrá una instanciación inválida del modelo.

## Relación entre entidades

Con lo visto hasta ahora, podríamos decir que `Course` no deja de ser una entidad más que se está **relacionando de forma directa con otras entidades**, lo que habitualmente vendría definiéndose en los ORM como un *oneToMany* o *manyToMany* dentro de nuestra clase. Esta relación al final implicará que cuando hagamos una consulta `select course...` nos acabe explotando la BD por la query tan grande que le estaríamos lanzando

Una alternativa a esto sería la definición de un **Aggregate**, que no sería más que un elemento conceptual que engloba distintas entidades, de tal modo que siempre que queramos interactuar con `Lessons`, por ejemplo, lo haremos a través de `Course`.

- Si queremos acceder al nombre de la lección no haremos una llamada del tipo `"course->lesson->name"` sino que lo haremos a través de un método `"lessonName()"` (no nos acoplaremos a cada uno de estos elementos encadenados)

Dentro de este **Aggregate** tendremos un **Aggregate Root**, que en este caso será `Course`

## Agregado Course tradicional + reviews

Clase `Course`:

```
final public class Course extends AggregateRoot {
    private final CourseId id;
    private final CourseName name;
    private final List<Review> reviews;

    public Course(CourseId id, CourseName name, List<Review> reviews){
        this.id = id;
        this.name = name;
        this.reviews = reviews;
    }

    public CourseId id() {
        return id;
    }

    public CourseName name() {
        return name
    }
}
```

Aquí podemos ver nuestro `Aggregate Course`, y cómo en el constructor de clase instanciamos sus atributos (para simplificar en esta ocasión no hemos añadido `Lessons`), de modo que estaríamos levantando todas las `Review` de un curso en el momento de levantar el propio curso.

El problema que nos surge aquí es que si las entidades `Lessons` y `Review` crecen y las mantenemos dentro del `aggregate`, el consumo en BD será tremendo y lo mismo nos sucederá a la hora de cargarlos en memoria cuando los recuperamos

No obstante, aunque hemos hablado de los problemas de usarlos, recordemos que el **Aggregate** y el **Aggregate Root** nos van a ofrecer unos beneficios

1. Encapsulación, siguiendo Tell Don't Ask, nos va a permitir que los clientes no se vean afectados si en algún momento quisiéramos cambiar cómo funciona todo de forma interna
2. Mantener las restricciones de integridad síncronas que se establezcan dentro del agregado

## ¿Alguna Duda?

En el siguiente video daremos un pasito más realizando un refactor para solucionar los problemas que nos han ido surgiendo, pero si tienes cualquier duda o quieres sugerirnos alguna mejora sobre este contenido ¡no dudes en abrir una nueva discusión aquí abajo y contarnos!


Nos vemos en el siguiente video! 🍌🍌

Complete Step

100pts

Group Discussion

Start a discussion

-  **Servicios de dominio y agregados**  
Santiago D. · 21 days ago · 👁️ +0 · +0 [Answer >](#)
-  **Relaciones NxM en Agregados**  
Juan A. · 6 months ago · 👁️ +2 · +1 [Answer >](#)
-  **Caso de uso que implica el update de 2 aggregates**  
Marcos S. · a year ago · 👁️ +5 · +3 [Answer >](#)