**Herencia VS composición**

|  |
| --- |
| Herencia VS composición……………………………………………………………………………………………………..1  Herencia……………………………………………………………………………………………………………………………….1  Composición…………………………………………………………………………………………………………………………4  Herencia Práctica………………………………………....………………………………………………………………………5 Composición Práctica………………………………………………………………………………………………………….8 Conclusiones Herencia VS Composición………..…………………………………………………………………….14  Extra relaciones OneToMany y dirección de la relación en Doctrine…………………………………….16 |

**Herencia VS Composición**

Las dos técnicas más comunes para reutilizar funcionalidad en sistemas orientados a objetos son la herencia de clases y la composición de objetos. Como ya hemos explicado, la herencia de clases permite definir una implementación en términos de otra. A esta forma de reutilización mediante herencia se la denomina frecuentemente reutilización de caja blanca. El término “caja blanca" se refiere a la visibilidad: con la herencia, las interioridades de las clases padres suelen hacerse visibles a las subclases.

La composición de objetos es una alternativa a la herencia de clases. Ahora, la nueva funcionali*dad se obtiene ensamblando o componiendo objetos para obtener funcionalidad más compleja. La* composición de objetos requiere que los objetos a componer tengan interfaces bien definidas. Este estilo de reutilización se denomina reutilización de caja negra, porque los detalles internos de los objetos no son visibles. Los objetos aparecen solo como “cajas negras”.

Tanto la herencia como la composición tienen sus ventajas e inconvenientes**.**

**Herencia:**

La herencia de clases **se define estáticamente en tiempo de compilación** y es sencilla de usar, al estar permitida directamente por el lenguaje de programación. La herencia de clases también hace que sea más fácil modificar la implementación que está siendo reutilizada. Cuando una subclase redefine alguna de las operaciones, puede afectar también a las operaciones de las que hereda, suponiendo que éstas llamen a alguna de las operaciones redefinidas.

\* La herencia es atractiva porque permite reutilizar código de una manera rápida, sencilla y evidente.

\* La herencia es poderosa porque me permite jugar con el polimorfismo con facilidad.

\* La herencia me permite sobrescribir métodos, así que si algo no es exactamente como espero siempre puedo cambiarlo si fuera necesario, por lo que es versátil.

Pero la herencia de clases también tiene **inconvenientes. En primer lugar, no se pueden cambiar las implementaciones heredadas de las clases padre en tiempo de ejecución, porque la herencia se define en tiempo de compilación. En segundo lugar, y lo que generalmente es peor, las clases padre suelen definir al menos parte de la representación física de sus subclases.** Como la herencia expone a una subclase los detalles de la implementación de su padre, suele decirse que “la herencia rompe la encapsulación” [Sny86]. La implementación de una subclase se liga de tal forma a la implementación de su clase padre que cualquier cambio en la implementación del padre obligará a cambiar la subclase.

Las dependencias de implementación pueden causar problemas al tratar de reutilizar una subclase. Cuando cualquier aspecto de la implementación heredada no sea apropiado para nuevos dominios de problemas, la clase padre deberá ser escrita de nuevo o reemplazada por otra más adecuada. Esta dependencia limita la flexibilidad y la reutilización. Una solución a esto es heredar sólo de clases abstractas, ya que éstas normalmente tienen poca o ninguna implementación.

La principal razón es bastante obvia aunque no lo parezca. En un proyecto que se enmarca en un proceso de desarrollo ágil, donde los requisitos cambian constantemente y no están para nada bajo control, es imposible hacer suposiciones del tipo: “esta pantalla va a tener que realizar siempre esta acción”, “esta respuesta de servidor siempre tiene 4 listados”. **Borra de tu mente la palabra siempre, siempre guía de forma equivocada en nuestra mente a “es un” y eso guía a herencia**.

Por lo tanto en etapas iniciales del proyecto, que es donde precisamente más incertidumbre tenemos, **es prácticamente imposible intentar modelar comportamiento basado en pautas que podamos considerar constantes**, porque nos guste o no, todo está sujeto a cambio.

Por ese motivo, si empezamos a usar comportamientos heredados, aunque podamos a priori obtener resultados rápidos y robustos, en algún momento empezarán a sucederse cosas que yo categorizo con nombres como estos: “la fiesta de los override”, “el aluvión de métodos heredados inservibles”, “la herencia de 7 niveles”, “el quiero heredar y no puedo porque estoy sujeto a otra herencia” y un montón de inconvenientes muy muy divertidos que van a ensuciar nuestro código con el cambio, y el cambio es cuestión de tiempo.

Pues bien, el problema de los problemas, es mal usar la herencia, es decir, utilizar herencia en el lugar y momento inadecuados. De hecho hacer eso es lo más parecido que hay a hipotecar tu software: al principio todo es muy bonito y todo funciona, pero cuando empieces a pagar intereses… ¡Ay! pobre de ti.

Ten en cuenta lo siguiente: **abstraer comportamiento de piezas del sistema en etapas tempranas del diseño es una de las tareas más complejas de la ingeniería del software**, y crear una relación de herencia no es más que crear una abstracción de algo que siempre va a ser de ese modo. ¿En serio te merece la pena arriesgar tanto?  
 **Cuando la herencia tiene sentido**

* Si tenemos dos clases directamente relacionadas que están basadas una en la otra y pertenecen al mismo dominio lógico, y **estás seguro de que no van a cruzar fronteras no deseadas** (por ejemplo ni ellas ni ninguna extensión saldrá del paquete), puedes optar por una relación de herencia.
* Si **la subclase es y será siempre algo basado en la superclase** y además la implementación de la superclase es apropiada e incluso necesaria para la subclase , puedes aplicar herencia sin miedo a equivocarte.
* Si además de lo que acabas de leer, **la subclase es candidata a sólo añadir nueva funcionalidad y no a sobrescribir nada**, sigues por el buen camino y la herencia es bienvenida.

## Cómo detectar que estás usando mal la herencia

* Cuando los **Overrides** empiezan a crecer y multiplicarse por tu código de forma exponencial y a veces de forma obligada y sin mucho sentido.
* Si **la relación “Es un” se ha roto** por algún motivo
* Si tu código, cuando crece por extensión, **requiere modificaciones de clases que son supertipos**, y además siempre suelen ser las mismas clases base las que estas tocando
* Si **los niveles de herencia empiezan a ser muchos**, te pierdes al tirar del hilo y los efectos de lado empiezan a manifestarse

**Composición:**

**La composición de objetos se define dinámicamente en tiempo de ejecución a través de objetos que tienen referencias a otros objetos**. La composición requiere que los objetos tengan en cuenta las interfaces de los otros, lo que a su vez requiere interfaces cuidadosamente diseñadas que no impidan que un objeto sea utilizado por otros. Pero hay una ventaja en esto: puesto que a los objetos se accede sólo a través de sus interfaces no se rompe su encapsulación. Cualquier objeto puede ser reemplazado en tiempo de ejecución por otro siempre que sean del mismo tipo. Además, como la implementación de un objeto se escribirá en términos de interfaces de objetos, las dependencias de implementación son notablemente menores.

La composición de objetos produce otro efecto en el diseño del sistema**. Optar por la composición de objetos frente a la herencia de clases ayuda a mantener cada clase encapsulada y centrada en una sola tarea. De esta manera, nuestras clases y jerarquías de clases permanecerán pequeñas y será menos probable que se conviertan en monstruos inmanejables**. Por otro lado, un diseño basado en la composición de objetos tendrá más objetos (al tener menos clases), y el comportamiento del sistema dependerá de sus relaciones en vez de estar definido en una clase.

Esto nos lleva a nuestro segundo principio del diseño orientado a objetos:

*Favorecer la composición de objetos frente a la herencia de clases.*

En la Práctica

Artículo, Video pueden recibir comentarios.

**Opción 1: Mediante herencia**

Se crea una clase base abstracta llamada CommentableObject de la cual heredan articulo y video.

abstract class CommentableObject  
{  
 private Uuid $uuid;  
 public function \_\_construct(Uuid $uuid)  
 {  
 $this->uuid = $uuid;  
 $this->comments = [];  
 }  
  
 */\*\*  
 \** ***@return*** *Uuid  
 \*/* public function getUuid(): Uuid  
 {  
 return $this->uuid;  
 }  
  
}

Siguiendo los principios de DDD al ser el objeto que contiene los comentarios y los propios comentarios en si agregados distintos no existe aquí un listado/ array de comentarios.  
  
Su implementación en doctrine:

<?xml version="1.0" encoding="utf-8"?>  
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping https://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd">  
 <entity name="App\Domain\OneToMany\InheritanceMode\CommentableObject\Entity\CommentableObject" table="commentable\_objects" inheritance-type="JOINED">  
  
 <id name="uuid" type="uuid" column="uuid" length="36" />  
  
 <discriminator-column name="discr" type="string" />  
 <discriminator-map>  
 <discriminator-mapping value="article\_in" class="App\Domain\OneToMany\InheritanceMode\ArticleInheritance\Entity\ArticleInheritance" />  
 <discriminator-mapping value="video\_in" class="App\Domain\OneToMany\InheritanceMode\VideoInheritance\Entity\VideoInheritance" />  
 </discriminator-map>  
 </entity>  
</doctrine-mapping>

Una clase hija quedaría tal que:

class ArticleInheritance extends CommentableObject  
{  
 private string $title;  
 private string $text;  
  
 */\*\*  
 \** ***@param*** *Uuid $uuid  
 \** ***@param*** *string $title  
 \** ***@param*** *string $text  
 \*/* public function \_\_construct(Uuid $uuid,string $title, string $text)  
 {  
 $this->title = $title;  
 $this->text = $text;  
 parent::*\_\_construct*($uuid);  
 }

En Doctrine:

<?xml version="1.0" encoding="utf-8"?>  
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping https://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd">  
 <entity name="App\Domain\InheritanceMode\ArticleInheritance\Entity\ArticleInheritance" table="articles">  
 <field name="title" column="title" />  
 <field name="text" column="text" type="text"/>  
 </entity>  
</doctrine-mapping>

La implementación de la clase comentario es una clase en cuya definición se referencia a un Uuid perteneciente a la clase padre de Articulo y Video (commentableObject):

class CommentInheritance  
{  
 private Uuid $commentableObjectParent;  
 private string $comment;  
 private \DateTime $commentDate;  
 private Uuid $uuid;  
  
 private function \_\_construct(Uuid $commentableObjectParent, Uuid $uuid, DateTimeProvider $dateTimeProvider, string $comment)  
 {  
 $this->commentableObjectParent = $commentableObjectParent;  
 $this->uuid = $uuid;  
 $this->commentDate = $dateTimeProvider->getCurrentDate();  
 $this->comment = $comment;  
 }  
  
 public static function create(CommentableObject $commentableObjectParent, DateTimeProvider $dateTimeProvider, string $comment):CommentInheritance  
 {  
 return new self($commentableObjectParent->getUuid(),Uuid::*random*(),$dateTimeProvider,$comment);  
 }

Dado DDD dos agregados distintos solo se referencia por Uuid, en DDD relajado podríamos haber puesto un “private CommentableObject $commentableObject” en lugar de su uuid

Para guardar comentarios o traerme los comentarios de un Commentable Object lo hacemos via repositorio en la capa de aplicación:

class AddCommentToCommentableObjectService  
{  
 …etc…  
  
 public function \_\_invoke(Uuid $commentableObjectUuid, string $comment)  
 {  
 //to prevent a fake or not store $commentable object here.  
 $commentableObject = $this->commentableObjectRepository->findOneById($commentableObjectUuid);  
 $newComment = CommentInheritance::*create*($commentableObject, $this->dateTimeProvider, $comment);  
 $this->commentInheritanceRepository->save($newComment);  
 }  
}

class GetCommentsOfCommentableObjectService  
{  
 …etc…  
  
 */\*\*  
 \** ***@param*** *Uuid $commentableObjectUuid  
 \** ***@return*** *iterable | CommentInheritance[]  
 \*/* public function \_\_invoke(Uuid $commentableObjectUuid): iterable  
 {  
 return $this->commentInheritanceRepository->findAllByCriteria(['commentableObjectParent' => $commentableObjectUuid]);  
 }  
}

**Herencia conclusión**

La lógica de la relación siempre cae del lado del padre de la herencia, es decir, la clase comentario hace referencia al padre de la herencia como posible “tenedor” de comentarios. Si el día de mañana sale otra relación para con… votes, he de ir al padre a añadir la implementación y si uno de sus hijos no la comparte => se jode y/o hace nula la implementación o hace una ñapa para no usarla.

Otro problema es que cada vez que una clase nueva quiere ser comentable he de añadir un nuevo “discriminator” en el mapping de doctrine de la clase padre.

**Opción 2: mediante Composición**

1.- Definimos un value object que encapsula el uuid de un objeto y su tipo:

final class CommentParentId  
{  
 private string $identifier;  
 private string $commentable;  
  
 public function \_\_construct(string $identifier, string $commentable)  
 {  
 $this->identifier = $identifier;  
 $this->commentable = $commentable;  
 }  
  
 public function identifier(): string  
 {  
 return $this->identifier;  
 }  
  
 public function commentable(): string  
 {  
 return $this->commentable;  
 }  
}

2.- Definimos una interfaz comentable que obliga a las clases que lo usen a implementar el siguiente método:

*/\*\*  
 \* Every class that want to receive comments by composition should implement this method, that return its  
 \* identifier to be referenced in the comments table  
 \*/*interface CommentableCompositionInterface  
{  
 public function commentParentId(): CommentParentId;  
}

3.- La implementación de las clases que usan dicha composición quedan tal que:

class ArticleComposition implements CommentableCompositionInterface  
{  
 private string $title;  
 private string $text;  
 private Uuid $uuid;  
  
 */\*\*  
 \** ***@param*** *Uuid $uuid  
 \** ***@param*** *string $title  
 \** ***@param*** *string $text  
 \*/* public function \_\_construct(Uuid $uuid,string $title, string $text)  
 {  
 $this->title = $title;  
 $this->text = $text;  
 $this->uuid = $uuid;  
 }  
  
 */\*\*  
 \* This is the only method that interface CommentableCompositionInterface forces you to implement if you want to be able to  
 \* receive comments  
 \** ***@return*** *CommentParentId  
 \*/* public function commentParentId(): CommentParentId  
 {  
 return new CommentParentId($this->getUuid()->value(), ArticleComposition::class);  
 }  
  
 */\*\*  
 \** ***@return*** *Uuid  
 \*/* public function getUuid(): Uuid  
 {  
 return $this->uuid;  
 }  
  
  
 */\*\*  
 \** ***@return*** *string  
 \*/* public function getTitle(): string  
 {  
 return $this->title;  
 }  
  
 */\*\*  
 \** ***@param*** *string $title  
 \*/* public function setTitle(string $title): void  
 {  
 $this->title = $title;  
 }  
  
 */\*\*  
 \** ***@return*** *string  
 \*/* public function getText(): string  
 {  
 return $this->text;  
 }  
  
 */\*\*  
 \** ***@param*** *string $text  
 \*/* public function setText(string $text): void  
 {  
 $this->text = $text;  
 }  
}

4.- La clase que guardará los comentarios para todos los objetos que implementen la interfaz será:

class CommentComposition  
{  
 private Uuid $uuid;  
 private CommentParentId $commentParentId;  
 private string $comment;  
 private \DateTime $commentDate;  
  
 public function \_\_construct(CommentableCompositionInterface $commentableObject, DateTimeProvider $dateTimeProvider,string $comment)  
 {  
 $this->commentDate = $dateTimeProvider->getCurrentDate();  
 $this->comment = $comment;  
 $this->uuid = Uuid::*random*();  
 $this->commentParentId = $commentableObject->commentParentId();  
 }  
  
  
 */\*\*  
 \** ***@return*** *Uuid  
 \*/* public function getUuid(): Uuid  
 {  
 return $this->uuid;  
 }  
  
  
 */\*\*  
 \** ***@return*** *string  
 \*/* public function getText(): string  
 {  
 return $this->comment;  
 }  
  
 */\*\*  
 \** ***@param*** *string $comment  
 \*/* public function setText(string $comment): void  
 {  
 $this->comment = $comment;  
 }  
  
 */\*\*  
 \** ***@return*** *\DateTime  
 \*/* public function getCommentDate(): \DateTime  
 {  
 return $this->commentDate;  
 }  
  
 */\*\*  
 \** ***@param*** *\DateTime $commentDate  
 \*/* public function setCommentDate(\DateTime $commentDate): void  
 {  
 $this->commentDate = $commentDate;  
 }  
  
 */\*\*  
 \** ***@return*** *CommentParentId  
 \*/* public function getCommentParentId(): CommentParentId  
 {  
 return $this->commentParentId;  
 }  
  
}

5.- Vemos como el mapeo de la BD NO se ve afectado ( no hay ni rastro de comentarios):

<?xml version="1.0" encoding="utf-8"?>  
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping https://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd">  
 <entity name="App\Domain\CompositionMode\ArticleComposition\Entity\ArticleComposition" table="articlecomposition\_objects">  
 <id name="uuid" type="uuid" column="uuid" length="36" />  
 <field name="title" column="title" />  
 <field name="text" column="text"/>  
 </entity>  
</doctrine-mapping>

A la hora de operar, guardar un comentario y recuperarlo sería tal que:

public function it\_should\_save\_a\_comment\_inside\_the\_articleComposition\_object(): void  
{  
 //GIVEN  
 $dateTimeProvider = $this->createMock(DateTimeProvider::class);  
 $dateTimeProvider->method('getCurrentDate')->willReturn(new \DateTime('2022-02-02 02:02:02'));  
 $saveCommentFromACommentableCompositionService = new SaveCommentFromACommentableCompositionService($this->commentCompositionRepository);  
 $retrieveCommentsFromACommentableService = new RetrieveCommentsFromACommentableCompositionService($this->commentCompositionRepository);  
 $articleComposition = $this->fakerFactory->getValidArticleComposition();  
 //save a comment  
 $comment1 = "blalblabl1";  
 $saveCommentFromACommentableCompositionService->\_\_invoke($comment1, $articleComposition,$dateTimeProvider);  
 $this->clearUnitOfWork();  
 //WHEN  
 $comments = $retrieveCommentsFromACommentableService->\_\_invoke($articleComposition);  
  
 //THEN  
 $this->assertCount(1,$comments);  
 $this->assertEquals($comments[0]->getText(),$comment1);  
}

Ojo estamos guardando la relación entre dos agregados distintos unidos por composición por lo que ok.

Usando un servicio de aplicacion que inyecta un repositorio para el guardado:

*/\*\*  
 \* Yes, a domain service that injects a repository...  
 \*/*class SaveCommentFromACommentableCompositionService  
{  
 private CommentCompositionRepositoryInterface $commentCompositionRepository;  
  
 public function \_\_construct(CommentCompositionRepositoryInterface $commentCompositionRepository)  
 {  
 $this->commentCompositionRepository = $commentCompositionRepository;  
 }  
  
 */\*\*  
 \** ***@param*** *string $comment  
 \** ***@param*** *CommentableCompositionInterface $commentableCompositionObject  
 \** ***@param*** *DateTimeProvider $dateTimeProvider  
 \** ***@return*** *iterable | CommentComposition  
 \*/* public function \_\_invoke(string $comment, CommentableCompositionInterface $commentableCompositionObject,DateTimeProvider $dateTimeProvider): void{  
 $commentComposition = new CommentComposition($commentableCompositionObject,$dateTimeProvider,$comment);  
 $this->commentCompositionRepository->save($commentComposition);  
 }  
}

La recuperación se hace a través del repositorio, en esta ocasión encapsulado en un “servicio de aplicacion”

class RetrieveCommentsFromACommentableComposition  
{  
 private CommentCompositionRepositoryInterface $commentCompositionRepository;  
  
 public function \_\_construct(CommentCompositionRepositoryInterface $commentCompositionRepository)  
 {  
 $this->commentCompositionRepository = $commentCompositionRepository;  
 }  
  
 */\*\*  
 \** ***@param*** *CommentableCompositionInterface $commentableCompositionObject  
 \** ***@return*** *iterable | CommentComposition  
 \*/* public function \_\_invoke(CommentableCompositionInterface $commentableCompositionObject): iterable{  
 return $this->commentCompositionRepository->findLatestByCommentable($commentableCompositionObject);  
 }  
}

Y el método del repositorio será:

public function findLatestByCommentable(CommentableCompositionInterface $commentable, int $numberOfComments = 10): iterable  
{  
 $commentParentId = $commentable->commentParentId();  
  
 $criteria = [  
 'commentParentId.identifier' => $commentParentId->identifier(),  
 'commentParentId.commentable' => $commentParentId->commentable(),  
 ];  
  
 $comments = [];  
 foreach ($this->findBy($criteria, [], $numberOfComments) as $comment) {  
 $comments[] = $comment;  
 }  
 return $comments   
}

**Conclusiones composición**

Creo que es la forma perfecta para:

1.- Descargar la clase base de ruido, no hay que añadir más discriminators cuando una nueva clase llega al mapping

2.- Te fuerza a usar un repo para traer la relación y con ello tienes la base para una hipotética paginación cuando el volumen suba.

3.- Las clases son más independientes

4.- Es implementar la interfaz y listo.

**Conclusiones Herencia VS Composición**

Diría que la composición es siempre mejor que la herencia, a ver, es más elaborada/molesta en cuanto has de usar un repositorio para traer los elementos relacionados, pero, si ambos son agregados sienta las bases correctas de la relación entre agregados al hacerlos via uuid y no objeto entero y, además, garantiza, o al menos, sienta las bases para un futuro en el que la relación se va de madre y hay que traer los objetos relacionados de forma paginada. Es decir, con respecto a DDD la composición promueve que la relación entre agregados sea via Uuid al imposibilitar que sea via referencia directa a los objetos. ( Insertar aquí el conocido debate de las ventajas e inconvenientes de eso)

**Un inconveniente es que es difícil examinar el código en tiempo de compliacion, es decir, la relación explicita entre articulo y comentarios no se ve reflejada directamente en el código lo que lo hace más complicado de interpretar y manejar, PERO realmente si se implementa la herencia siguiendo las bases del DDD esa inferencia tampoco es tan directa. Ahora si hubiésemos optado por implementar la herencia añadiendo un array de Comments en el padre si hubiese sido más fácil en tiempo de compilación observar el código y traer los comentarios desde dominio con un simple Articulo->/Video->getComments()**

La realidad es que no se puede optar por una solución que use la composición para esta relación OneToMany que permita tener en cada clase un listado de comentarios.

Otro problema de usar herencia es que todo objeto que quiera tener comentarios ha de extender de comentable por fuerza; esto trae a la luz los problemas clásicos de la herencia; en este caso: y si necesito añadir otra cosa a comentarios? Por fuerza se lo llevan todos los demás y si necesito hereda de otra clase? Y si necesito para articulo funciones de numero de caracteres en los comentarios pero no en video? Los otros herederos van incluyendo funciones que no necesita por el mero hecho de heredar del padre. Etc…

|  |
| --- |
| Go to service to make this due to relationship between aggregates that could be huge or store a list of uuids...  Why are we returning a Wish and not just adding the new Wish to an internal collection as we would do with Doctrine. To sum up, in this scenario, User and Wish do not conform an aggregate because there is no true business invariant to protect. A User can add and remove as many Wishes as she wants. Wishes and their User can be updated independently to the database in different transactions, if needed.  Following the rules about Aggregate design explained before, we should struggle to have Aggregates as small as possible, and that’s the result here. Each Entity has its own Repository and they are linked using their Identities, UserId in this case. Getting all the wishes of a user can be performed by a finder in the WishRepository, paginated easily without any performance issues. |

En verdad tb depende de lo controlada que esté la herencia, si es una herencia estable y la relación no es de muchos elementos casi que merece la pena ir por la herencia de cara a la claridad en el código no? Imagina en el ejemplo de los comentables anterior, si puede tener un máximo de no se 200 comentarios, son entidades simples y el padre es comentario no va a tener muchos atributos extras y de tenerlos van a ser comunes si o si a todas las clases que tienen comentarios… **para eso la herencia esta de lujo** el problema es:

|  |
| --- |
| **Borra de tu mente la palabra siempre, siempre guía de forma equivocada en nuestra mente a “es un” y eso guía a herencia**.  En etapas iniciales del proyecto, que es donde precisamente más incertidumbre tenemos, **es prácticamente imposible intentar modelar comportamiento basado en pautas que podamos considerar constantes**, porque nos guste o no, todo está sujeto a cambio. |

**Detalle interesante sobre las relaciones one to many, doctrine y la dirección de la relación.**

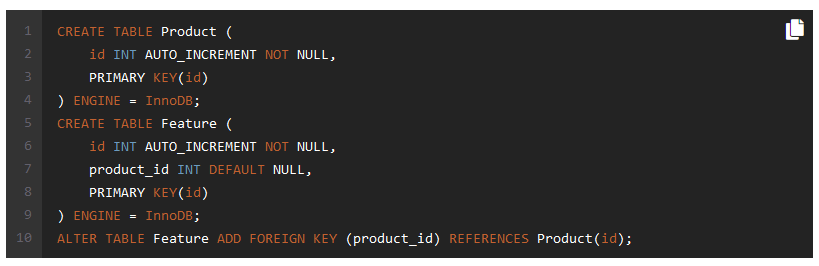
En este caso un articulo pueden tener muchos comentarios.

**Bidireccional**  
Si la relación es bidireccional tal que, desde articulo obtenemos los comentarios y viceversa, tendremos en articulo un array de comentarios y en comentario un atributo parentArticle del tipo Article, esto a la hora de llevarlo a infraestructura usando Doctrine se reflejará tal que:

Una relación one to many pura bidireccional:

|  |
| --- |
| A one-to-many association has to be bidirectional, unless you are using a join table. This is because the many side in a one-to-many association holds the foreign key, making it the owning side. Doctrine needs the many side defined in order to understand the association.  This bidirectional mapping requires the mappedBy attribute on the one side and the inversedBy attribute on the many side.  This means there is no difference between a bidirectional one-to-many and a bidirectional many-to-one.    <doctrine-mapping>  <entity name="Product">  <one-to-many field="features" target-entity="Feature" mapped-by="product" />  </entity>  <entity name="Feature">  <many-to-one field="product" target-entity="Product" inversed-by="features">  <join-column name="product\_id" referenced-column-name="id" />  </many-to-one>  </entity>  </doctrine-mapping>  Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same. |

Que en la base de datos se verá como:



Es decir, la relación se almacenará en la tabla hijo con un campo que hace referencia al id del padre sobre el cual recae la condición de clave foránea.

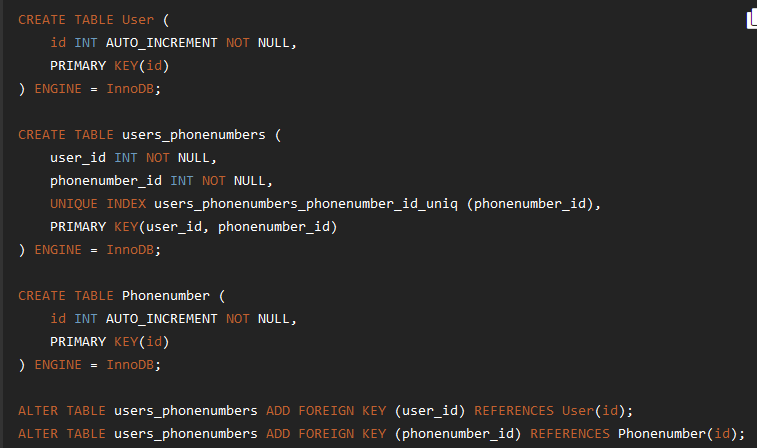
**Unidireccional**

Si la relación entre articulo y comentarios es unidireccional, es decir, desde artículos obtenemos el listado de comentarios y no del revés, es decir, en ningún momento nos interesa, a nivel de dominio, obtener el articulo asociado a un comentario partiendo de un comentario, (siempre podemos hacerlo vía método del repositorio consultando una tabla intermedia).

El hecho de que sea unidireccional hace que en doctrine se refleje tal que:

|  |
| --- |
| A unidirectional one-to-many association can be mapped through a join table. From Doctrine's point of view, it is simply mapped as a unidirectional many-to-many whereby a unique constraint on one of the join columns enforces the one-to-many cardinality.  The following example sets up such a unidirectional one-to-many association:    <doctrine-mapping>  <entity name="User">  <many-to-many field="phonenumbers" target-entity="Phonenumber">  <join-table name="users\_phonenumbers">  <join-columns>  <join-column name="user\_id" referenced-column-name="id" />  </join-columns>  <inverse-join-columns>  <join-column name="phonenumber\_id" referenced-column-name="id" unique="true" />  </inverse-join-columns>  </join-table>  </many-to-many>  </entity>  </doctrine-mapping> |

Que en la base de datos generará la siguiente estructura:



**Conclusión**

Hay que tener cuidado con el razonamiento en el sentido inverso, es decir, podemos planear la relación desde nuestro dominio en plan unidireccional, es decir, solo desde Articulo hacia Comentarios y cuando llegamos a Infraestructura, en este caso con Doctrine, nos empeñamos en hacer un One To Many, porque es un One To Many, y Doctrine para ese caso no exige un campo Articulo en la clase comentario, **NO DEBEMOS CAER EN LA TRAMPA y debemos implementarlo como un MANY TO MANY!!**