Modelo de dominio y mapeo objeto-relacional (ORM)

Diseño de Sistemas Software
Curso 2020/2021

Carlos Pérez Sancho

Universitat d'Alacant
Universidad de Alicante
Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Modelos de dominio

Implementación de modelos de dominio

- Las funcionalidades de un sistema requieren la ejecución de distintos tipos de acciones
  - Validación de los datos de entrada
  - Ejecución de las reglas de negocio
  - Comprobación de restricciones en los datos
    (p.ej. integridad referencial, normalmente se realiza en la base de datos)

Implementación de modelos de dominio

- Ejemplo: actualizar stock de producto
  - Validación: si la cantidad es negativa (reducción de stock) debe haber suficiente stock en el almacén
  - Reglas de negocio: si la cantidad restante es menor que un umbral, crear una alerta para reponer

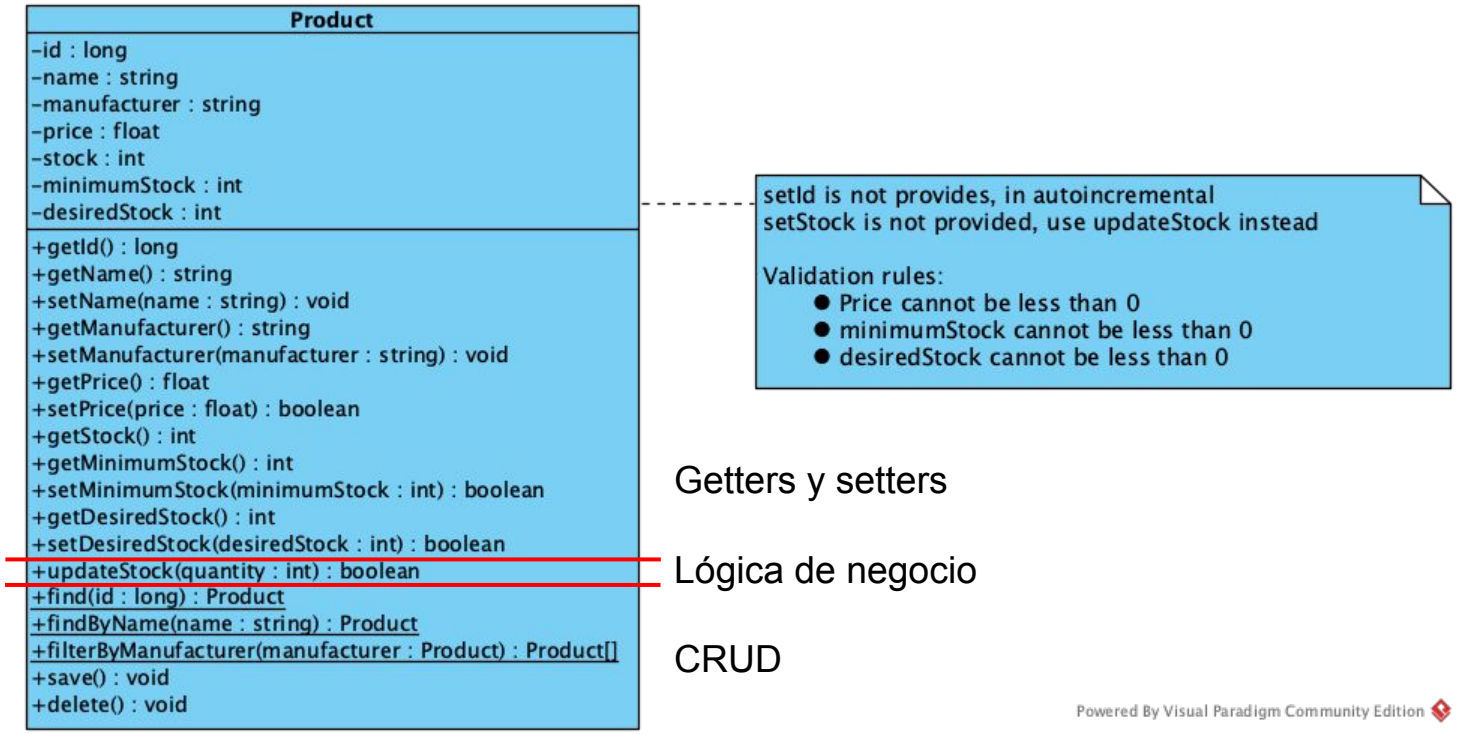
Tipos de modelos de dominio

- Al implementar la lógica de negocio con un modelo de dominio se puede optar entre 2 tipos distintos, dependiendo de cómo se distribuyan estas funcionalidades
  - Modelos de dominio **ricos**
  - Modelos de dominio **anémicos**

Modelos de dominio ricos

Modelos de dominio ricos

- En un modelo rico todos los métodos están en los modelos (CRUD + lógica de negocio)



Modelos de dominio ricos

- Implementación de las reglas de validación
  - Los métodos set (*setters*) deben comprobar la validez de los valores de entrada
  - Dos estilos de implementación
    - Devolver valores booleanos
    - Lanzar excepciones

Modelos de dominio ricos

- Implementación con Laravel
  - Los atributos son públicos y NO HAY QUE DECLARARLOS EN EL MODELO, Eloquent los coge automáticamente de la estructura de la tabla
  - No son necesarios getters ni setters, aunque se pueden implementar
    - **Accessors**: permiten crear valores calculados
    - **Mutators**: permiten realizar validaciones o transformaciones sobre los valores de entrada
  - Se pueden crear métodos CRUD para realizar consultas usando **Scopes**

Modelos de dominio ricos

- Migración

```php
public function up()
{
    Schema::create('products', function (Blueprint $table) {
        $table->bigIncrements('id');
        $table->timestamps();
        $table->string('name');
        $table->string('manufacturer');
        $table->float('price');
        $table->unsignedInteger('stock');
        $table->unsignedInteger('minimumStock');
        $table->unsignedInteger('desiredStock');
    });
}
```

Modelos de dominio ricos

- Modelo (1/2)

¡Los atributos no se declaran aquí!

```php
class Product extends Model
{
    public function getDescriptionAttribute() {
        return "{$this->name} ({$this->manufacturer})";
    }

    public function setPriceAttribute($value) {
        if ($value >= 0)
            $this->attributes['price'] = $value;
        else
            throw new \Exception('Invalid price');
    }

    public function scopeManufacturer($query, $manufacturer) {
        return $query->where('manufacturer', $manufacturer);
    }
```

Accessor

Mutator

Scope

Modelos de dominio ricos

- Modelo (2/2)

```
    public function updateStock($quantity) {
        $newStock = $this->stock - $quantity;

        if ($newStock < 0)
            throw new \Exception('There is not enough stock');

        $this->stock = $newStock;
        if ($newStock < $this->minimumStock)
            StockService::createAlert($this);
    }
}
```

Lógica de negocio

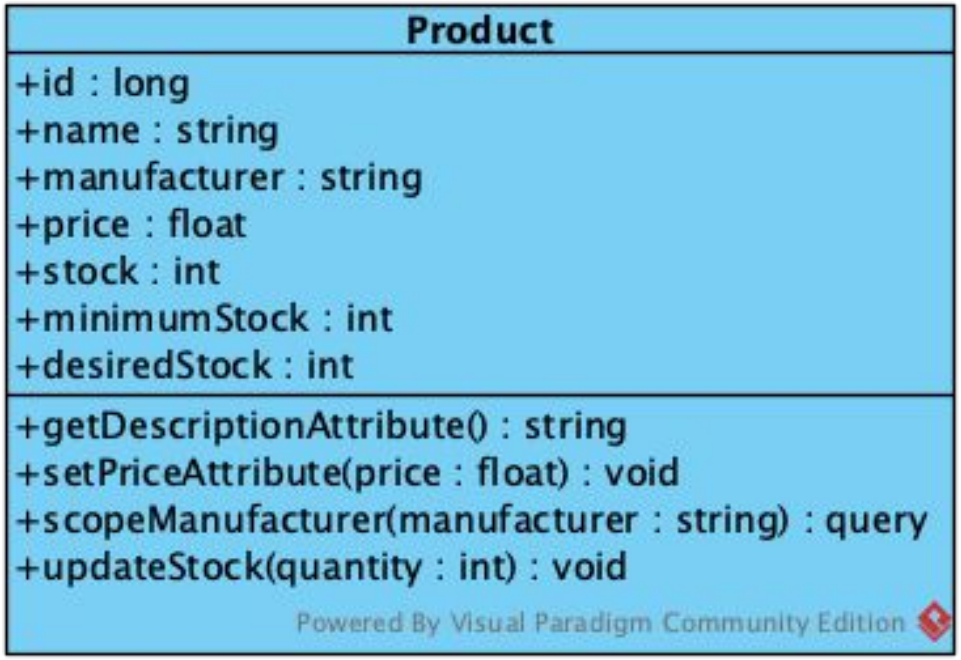
Modelos de dominio ricos

- Uso

```
>>> $p->description
=> "Monitor (Asus)"
>>> $p->price = -1
Exception with message 'Invalid price'
>>> $p->manufacturer('Asus')->get()
=> Illuminate\Database\Eloquent\Collection {#3016
     all: [
       App\Product {#3007
         id: 1,
         created_at: "2020-03-08 17:46:06",
         updated_at: "2020-03-08 17:46:06",
         name: "Monitor",
         manufacturer: "Asus",
         price: 150.0,
         stock: 20,
         minimumStock: 5,
         desiredStock: 20,
       },
     ],
   }
>>>
```

Modelos de dominio ricos

- Diagrama de implementación



Modelos de dominio anémicos

Modelos de dominio anémicos

- Un modelo de dominio **anémico** contiene objetos ligeros que solamente almacena datos y sus relaciones
- Estos objetos se pueden usar como objetos de transferencia de datos (Data Transfer Objects, DTO) para pasar información entre capas
- La validación de datos y la lógica de negocio se implementan fuera de estos objetos

Modelos de dominio anémicos

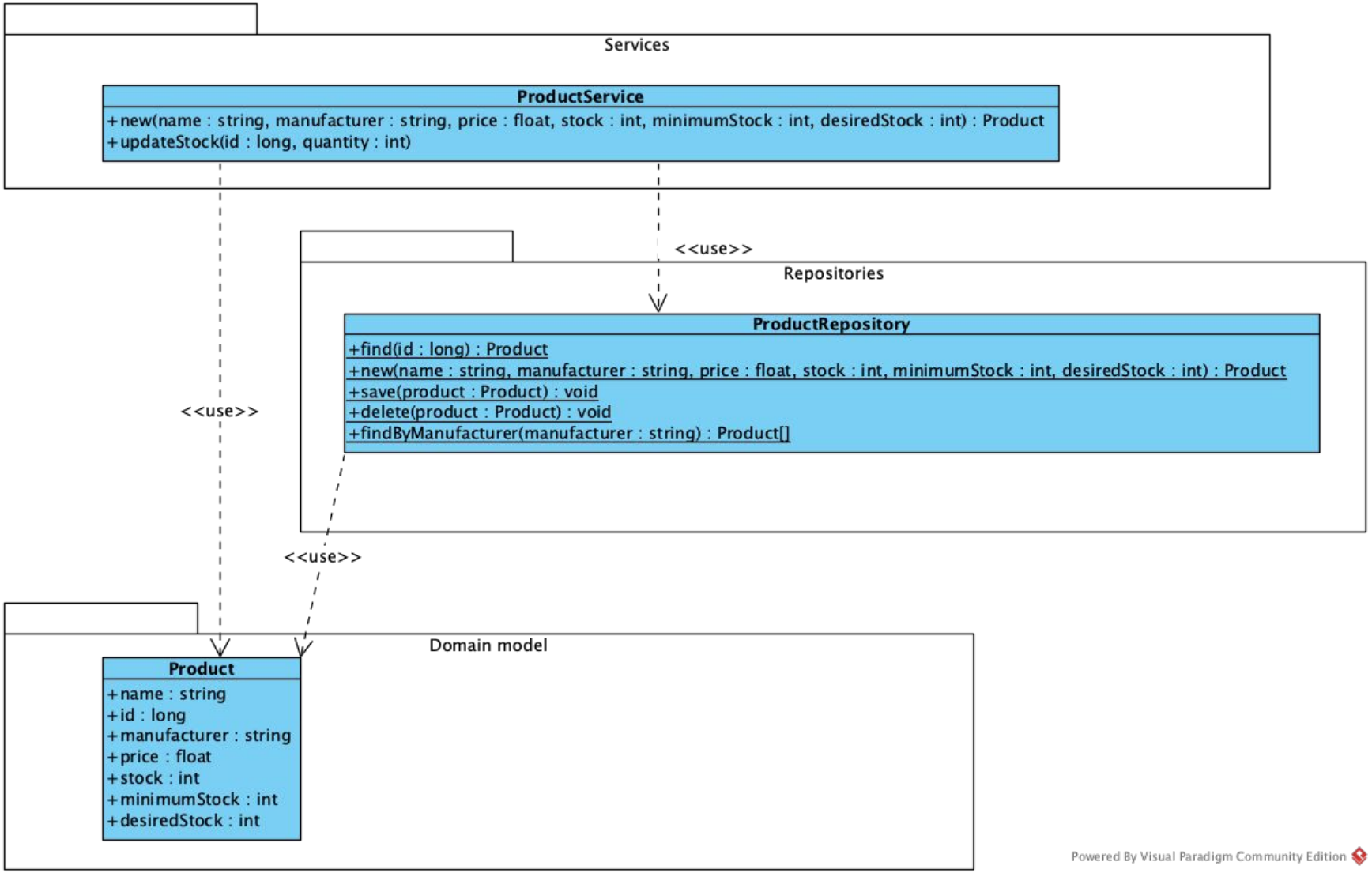
- ¿Dónde ubicamos entonces la validación de datos y la lógica de negocio?
  - La validación de los datos de entrada se suele hacer en los controladores de la capa de presentación
  - La lógica de negocio puede situarse en 2 capas distintas
    - Capa de presentación: ubicarla en los controladores hace perder la encapsulación y favorece la aparición de código duplicado (no recomendado)
    - Capa de servicios: objetos que implementan la lógica de negocio y usan el modelo de dominio para representar los datos

Modelos de dominio anémicos

- Es conveniente encapsular los métodos del CRUD para no depender de una implementación concreta
  - p.ej. podríamos cambiar de Eloquent (ActiveRecord) a Doctrine (Data Mapper)

- Para desacoplar se suele usar el **Patrón Repositorio**

Patrón repositorio



Modelos de dominio anémicos

- Modelo (si tuviera relaciones con otros objetos, habría que añadirlas aquí)

```
class Product extends Model
{
    //
}
```

Modelos de dominio anémicos

- Repositorio

```php
use App\Product;

class ProductRepository {
    public static function find($id) {
        return Product::find($id);
    }

    public static function new($name, $manufacturer, $price,
                               $stock, $minimumStock, $desiredStock) {
        $product = new Product();
        $product->name = $name;
        $product->manufacturer = $manufacturer;
        $product->price = $price;
        $product->stock = $stock;
        $product->minimumStock = $minimumStock;
        $product->desiredStock = $desiredStock;
        $product->save();
    }
}
```

Modelos de dominio anémicos

- Servicio

```php
use App\Repositories\ProductRepository;

class ProductService {
    public static function new($name, $manufacturer, $price,
                               $stock, $minimumStock, $desiredStock) {
        if ($price < 0)
            throw new \Exception('Invalid price');
        if ($stock < 0)
            throw new \Exception('Invalid stock');

        return ProductRepository::new($name, $manufacturer, $price,
                                      $stock, $minimumStock, $desiredStock);
    }

    public static function updateStock($id, $quantity) {
        $product = ProductRepository::find($id);
        if ($product === null)
            throw new \Exception('Product not found');

        $newStock = $product->stock - $quantity;
        if ($newStock < 0)
            throw new \Exception('There is not enough stock');

        $product->stock = $newStock;
        if ($newStock < $minimumStock)
            StockService::createAlert($product);
    }
}
```

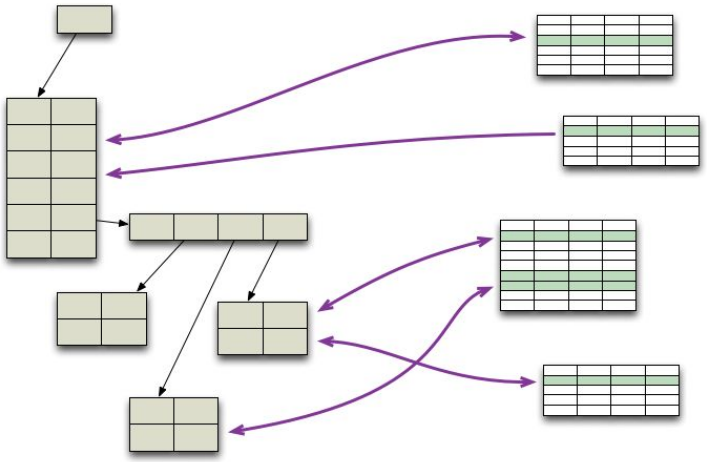
Modelos de dominio

- El patrón Repositorio también se puede usar con modelos de dominio ricos para desacoplar los controladores del uso de Eloquent

Mapeo objeto-relacional

Mapeo objeto-relacional

- Para implementar un modelo de dominio hay que encontrar la manera de adaptar la estructura del modelo a la estructura de la base de datos



https://martinfowler.com/bliki/OrmHate.html

Mapeo objeto-relacional

- Las **bases de datos orientadas a objetos** son una forma natural de persistir objetos, pero no son muy comunes
- Las **bases de datos relacionales** son mucho más comunes, pero tienen una estructura distinta a los modelos orientados a objetos (*impedance mismatch*)
- Las **bases de datos NoSQL** son más populares que las orientadas a objetos y permiten expresar la estructura de objetos complejos (p.ej. MongoDB almacena documentos JSON con una estructura de árbol), pero no tienen la potencia de las relacionales para trabajar con relaciones

Mapeo objeto-relacional

- Los sistemas de **mapeo objeto-relacional** (Object-Relational Mapping, ORM) se encargan de almacenar y recuperar los objetos en **bases de datos relacionales**
- Estos sistemas permiten crear una **capa de acceso a datos** sin necesidad de implementarla
- Ejemplos de sistemas ORM
  - Java: Hibernate, MyBatis, …
  - C#: Entity Framework, Nhibernate, …
  - PHP: Doctrine, Eloquent, …
  - Python: SQLAlquemy, Django ORM, …
  - …

Mapeo objeto-relacional

- Los sistemas ORM existentes normalmente implementan uno de los dos patrones para persistir modelos de dominio
  - **ActiveRecord**

    (+) es más sencillo de implementar

    (-) incrementa el acoplamiento con la base de datos y dificulta la refactorización

    (-) es difícil optimizar el SQL y puede perjudicar el rendimiento
  - **DataMapper**

    (+) permite realizar transformaciones complejas

    (+) facilita la optimización del acceso a la base de datos

    (-) es más difícil de entender y configurar

Mapeo objeto-relacional

- Dos estrategias
  - **Model first**: primero se diseña el modelo de dominio, y luego se crea la base de datos con la estructura necesaria. Situación ideal para usar el patrón ActiveRecord.
  - **Database first**: se parte de una base de datos ya existente, por lo que hay que adaptar el modelo de dominio a su estructura. En estos casos puede ser más conveniente usar un Data Mapper.

Patrones de comportamiento

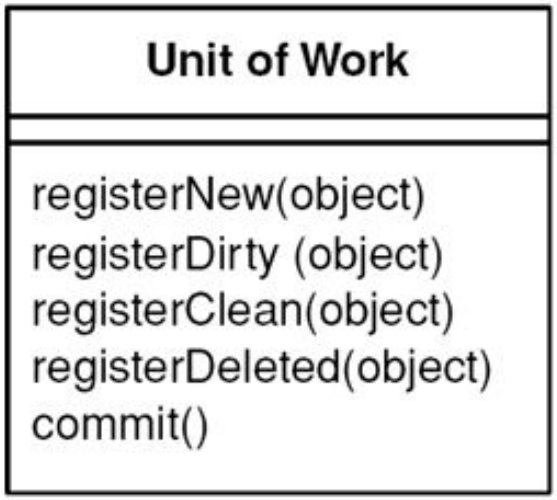
Mapeo objeto-relacional

Patrones de comportamiento

- Los patrones de comportamiento gestionan cómo se almacenan y recuperan los objetos de una base de datos relacional
- Problemas a resolver
  - Se debe mantener la integridad referencial cuando se crean y modifican múltiples objetos (**Unit of Work**)
  - No debe haber múltiples copias del mismo objeto en memoria (**Identity Map**)
  - Cargar los objetos relacionados en un modelo de dominio puede acabar recuperando la base de datos completa (**Lazy Load**)

Unit of Work

“Mantiene una lista de objetos afectados por una transacción y coordina la escritura de los cambios y la resolución de problemas de concurrencia.”



Unit of Work

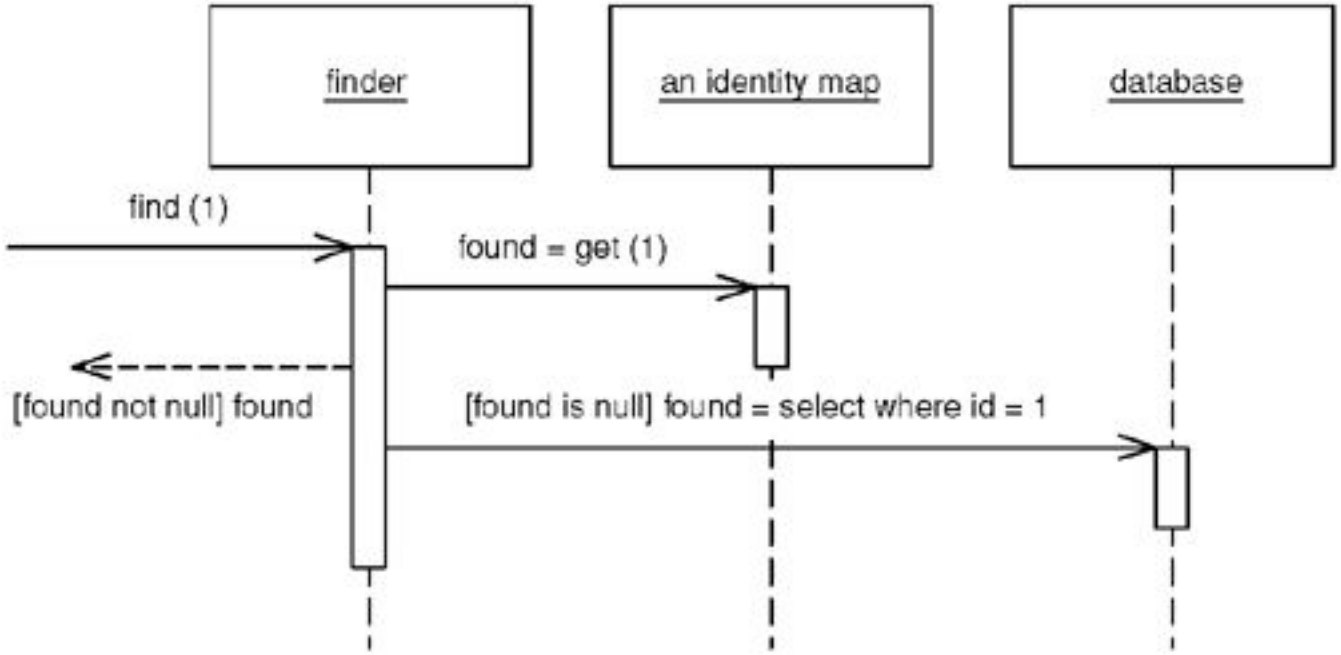
- Llamar a la base de datos para cada cambio en el modelo de dominio puede afectar al rendimiento
- En lugar de llamar directamente a la base de datos, el sistema notifica al objeto *Unit of Work*, de manera que pueda llevar un registro de los objetos nuevos, recuperados de la base de datos, modificados y borrados
- Cuando es necesario, abre una transacción con la base de datos y realiza todos los cambios en orden

Unit of work

- Laravel no proporciona este patrón

Identity Map

"Asegura que cada objeto se carga una única vez llevando un registro en un mapa de cada objeto. Busca los objetos en el mapa antes de recuperarlos de la base de datos."



Identity Map

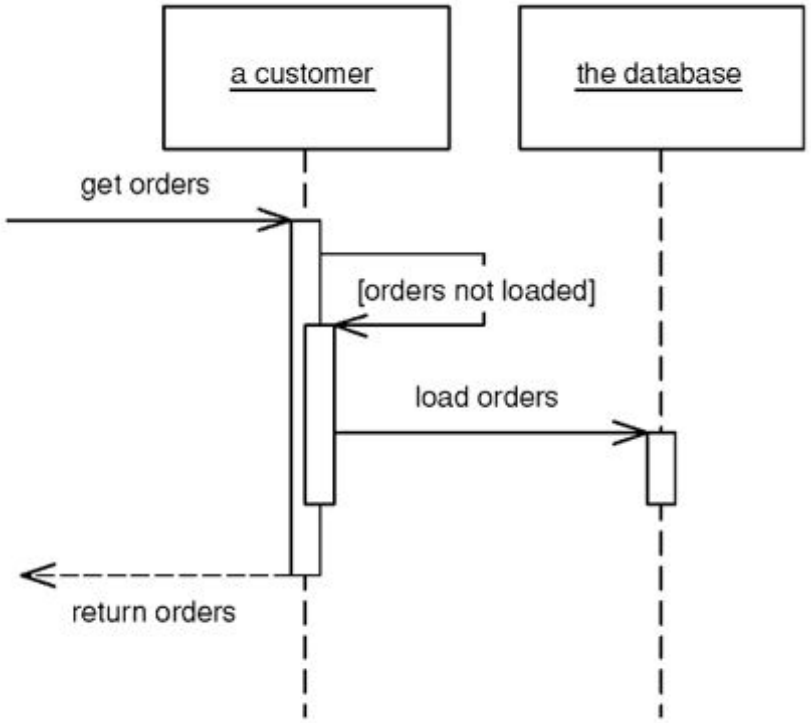
- Cargar un registro varias veces de la base de datos puede dar lugar a múltiples objetos con los mismos datos
- Problemas
  - Gasto excesivo de recursos
  - Inconsistencia cuando uno de ellos se modifica
- El objeto *Identity Map* lleva un registro de los objetos cargados y devuelve referencias para los que ya existe una instancia en memoria
- También actúa de caché para la base de datos

Identity Map

- Laravel no proporciona este patrón

Lazy Load

“Un objeto que no contiene datos, pero sabe cómo obtenerlos.”



Lazy Load

- Cuando se carga un objeto del modelo de dominio puede ser útil cargar sus objetos relacionados
- Sin embargo, esto puede degradar el rendimiento si el número de objetos relacionados es elevado, o incluso acabar cargando la base de datos completa en una reacción en cadena
- *Lazy Load* sólo carga un objeto cuando se usa

Lazy Load

- Alternativas de implementación
  - **Lazy initialization**: usa un valor nulo para los objetos hasta que se cargan, necesita que las propiedades estén encapsuladas en métodos get
  - **Virtual proxy**: los objetos se sustituyen por un objeto vacío que carga los datos cuando es necesario, permite separar la lógica necesaria para cargar los datos de los objetos del modelo de dominio

Lazy Load

- Laravel usa lazy-loading por defecto
- Se pueden cargar modelos relacionados para mejorar el rendimiento (eager loading)

```php
$books = App\Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

Patrones estructurales

Mapeo objeto-relacional

Mapeo de objetos pequeños

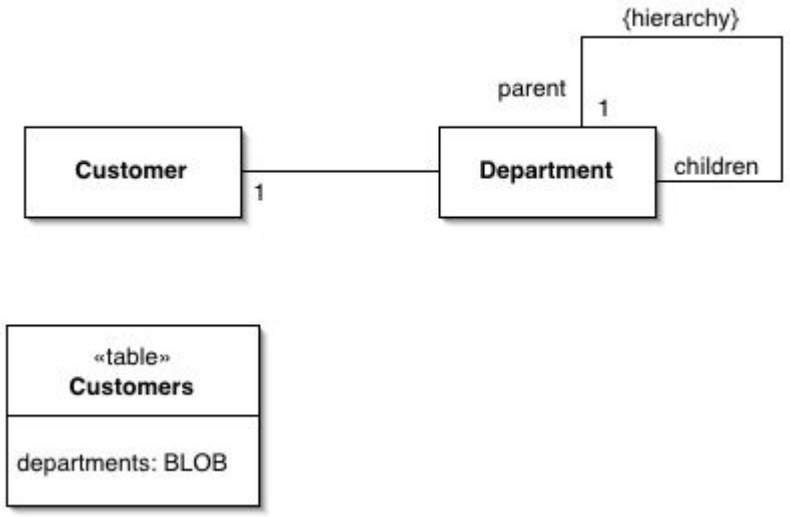
- A veces no es necesario mapear todos los objetos de dominio a tablas en la BBDD
  - Los objetos pequeños se pueden guardar junto a su contendor
  - Las colecciones o jerarquías de pequeños objetos se pueden guardar juntas, de manera que se pueda acceder a ellas en una sola operación

Embedded Value

"Mapea un objeto en varios campos de otra tabla."

Serialized LOB

- En ocasiones hay objetos que tienen una propiedad que contiene una estructura jerárquica de pequeños objetos
- Una forma eficiente de almacenarlos es transformarlos en un único objeto grande (LOB), ya sea en forma binaria (BLOB) o textual (CLOB), y almacenarlo en una única columna



Serialized LOB

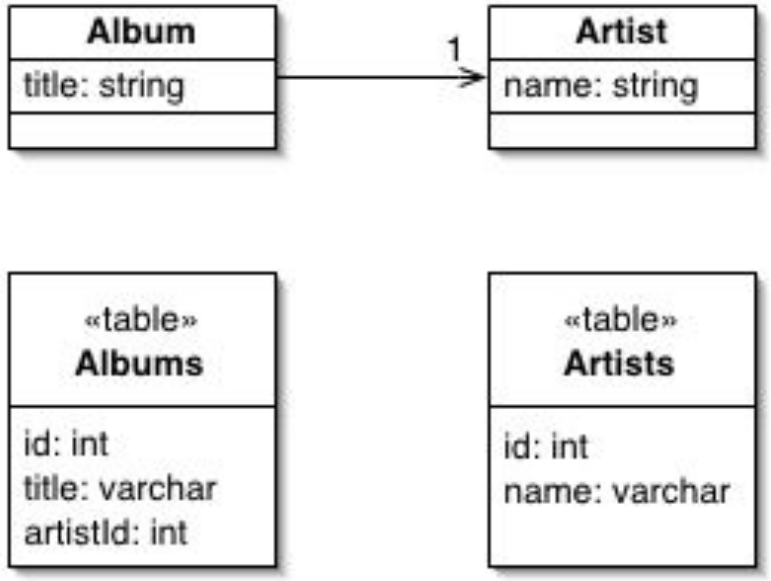
- ¡Atención! Los campos BLOB no deberían utilizarse para almacenar imágenes
- Es preferible almacenar las imágenes en una carpeta y guardar su ruta en la base de datos

Mapeo de relaciones

- Cuando un objeto contiene colecciones o referencias (compartidas) a otros objetos, no deben guardarse como valores en la misma tabla
- Necesitamos representar esas referencias para mantener la base de datos en forma normal

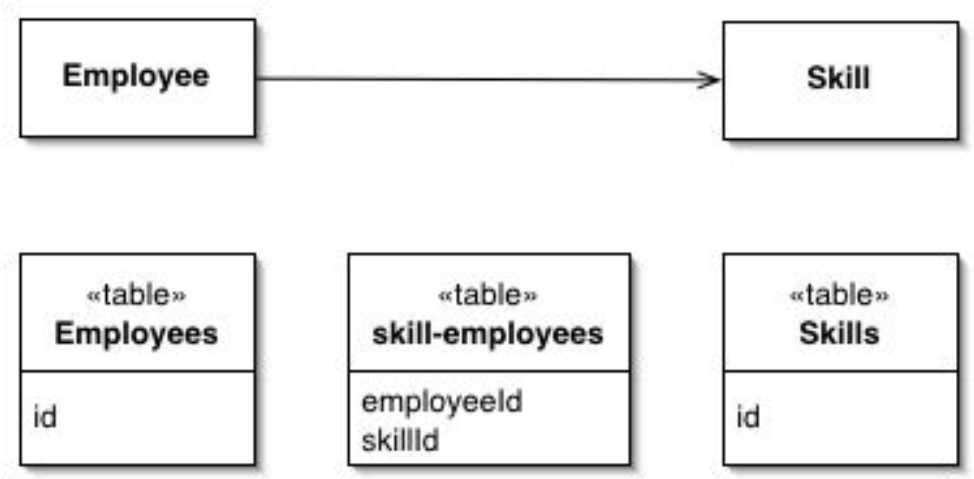
Mapeo de clave ajena

“Mapea una asociación entre objetos a una referencia de clave ajena entre tablas.”



Mapeo de tabla de asociación

"Guarda una asociación como una tabla con claves ajenas a las tablas que están vinculadas por la asociación".



Mapeo de la herencia

- Las bases de datos relacionales no soportan la herencia
- Si decidimos implementar una jerarquía de herencia en el modelo de dominio necesitamos:
  - Poder usar las clases hijas como si se tratase de la clase padre (polimorfismo)
  - Que todas las clases de la jerarquía compartan un campo identificador común

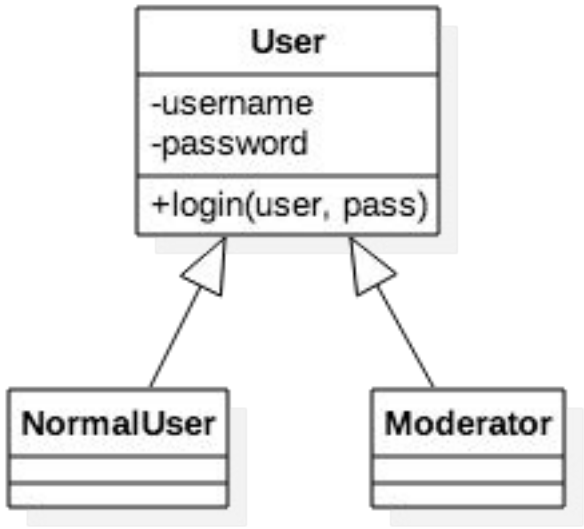
Mapeo de la herencia

Supongamos que cada objeto se almacena en una tabla diferente

```
$user = new NormalUser();
$user->save(); // Obtiene el id 1

$user = new Moderator();
$user->save(); // Obtiene el id 1

$user = User::find(1);
```



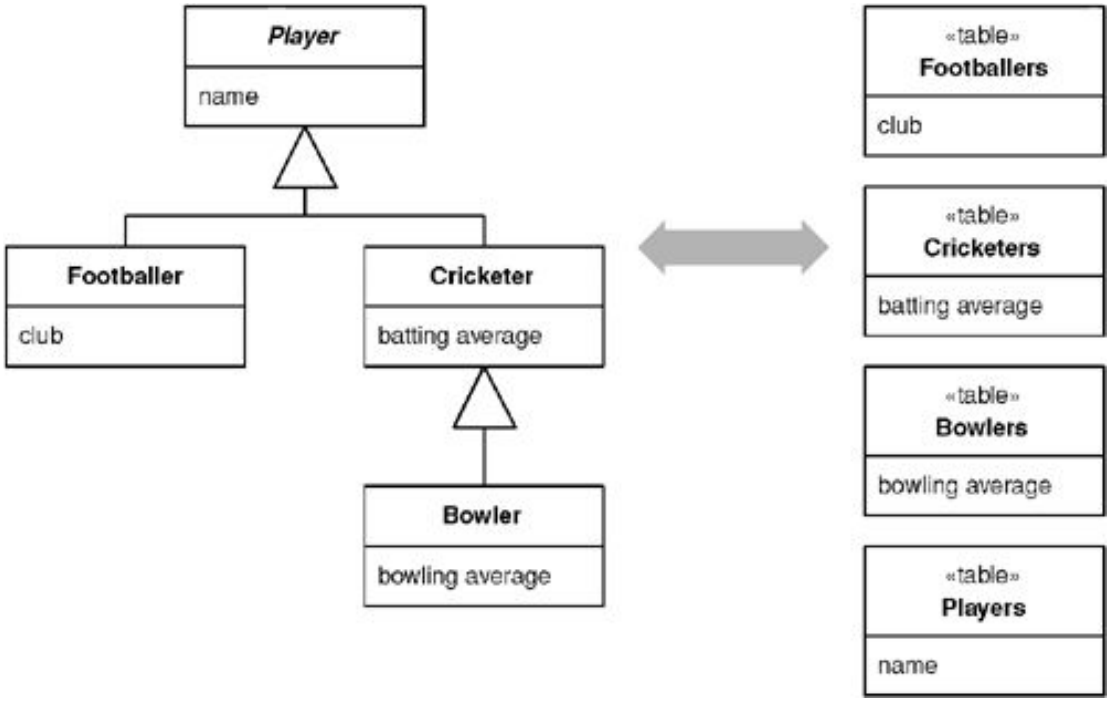
¿Cuál devolverá?

Patrón Class table inheritance

- Se usa una única tabla para cada clase en la jerarquía
- Cada tabla almacena únicamente los atributos nuevos

(-) Necesita hacer join para cada consulta, es un problema para el rendimiento

(+) No hay columnas irrelevantes



Patrón Concrete table inheritance

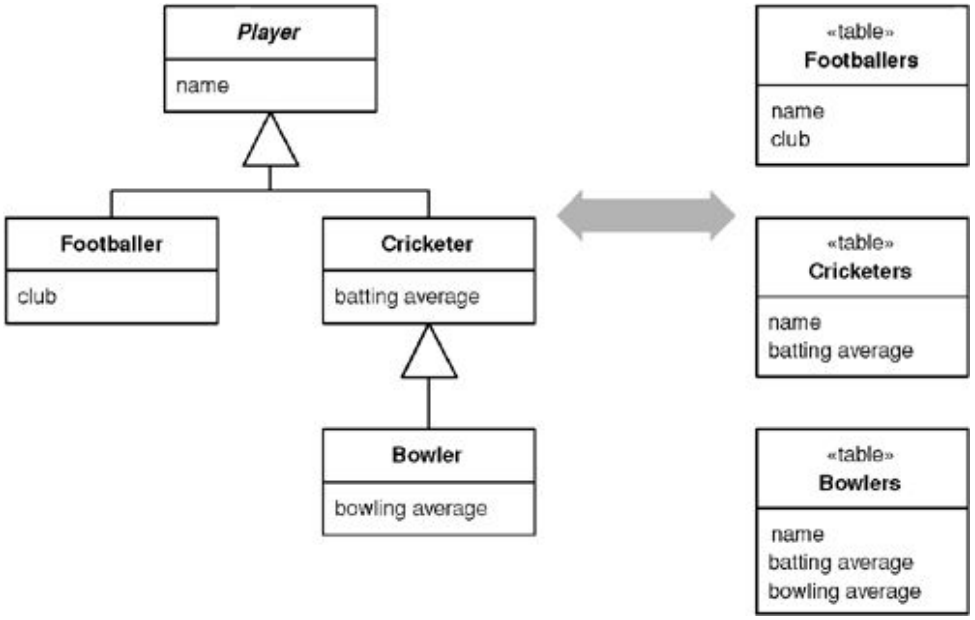
- Se usa una única tabla para cada clase “hoja”
- Cada tabla almacena todos los atributos (heredados + nuevos)

(-) Es complicado gestionar los identificadores si queremos que todas las subclases compartan campo identificador

(-) No se pueden representar relaciones con las clases abstractas

(+) No hay columnas irrelevantes

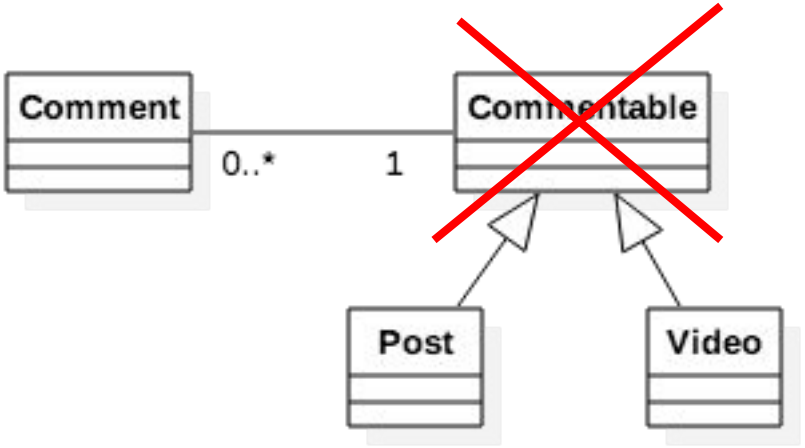
(+) No necesita hacer join



Patrón Concrete table inheritance

- Implementación en Laravel mediante relaciones polimórficas
  - Permite acceder a objetos de distinto tipo a través de una relación (`$comment->commentable` devolverá un objeto que podrá ser un `Post` o un `Video`)
  - La clase base no se implementa

https://laravel.com/docs/6.x/eloquent-relationships#polymorphic-relationships



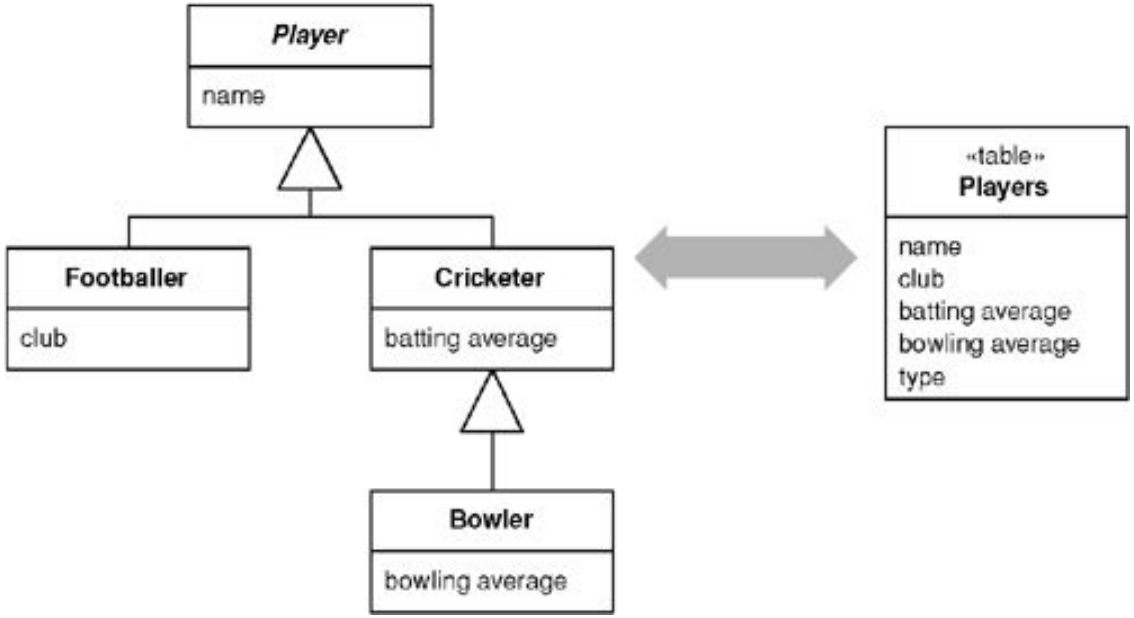
Patrón Single table inheritance

- Se usa una única tabla para todas las clases
- Almacena la unión de todos los atributos de las clases hijas

(-) Las columnas que no usan todas las clases gastan espacio

(+) Evita joins innecesarios

(+) Mismo campo identificador para todas las subclases



Patrón Single table inheritance

- Laravel tiene una extensión que permite implementar este patrón manteniendo la jerarquía de clases en el código
  - No permite instanciar objetos de la clase padre→ no podemos usar el polimorfismo `$user = User::find(1)`

https://github.com/Nanigans/single-table-inheritance

- En la práctica es más sencillo juntar todas las clases hijas en una sola con la union de todos los atributos y métodos, y un atributo adicional indicando el tipo de cada objeto

Pero entonces...

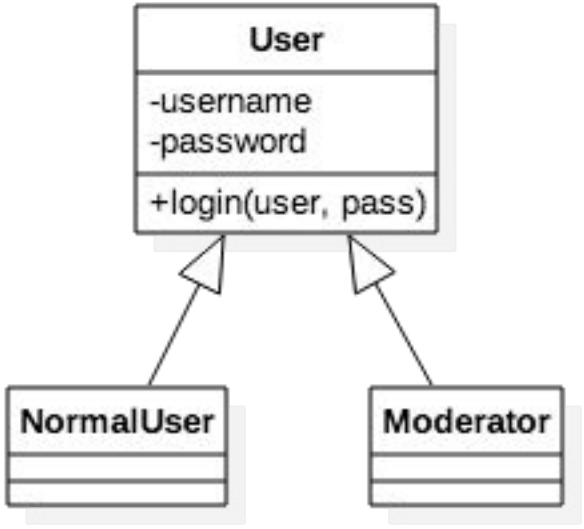
¿qué sucede si queremos tener distintos tipos de usuarios?

Autentificación de usuarios

Mapeo objeto-relacional

Autentificación de usuarios

- Requisitos:
  - En un foro podemos tener usuarios normales y moderadores
  - Los usuarios normales se pueden convertir en moderadores a propuesta de otros moderadores



¿PROBLEMAS?

- No podemos cambiar el tipo de un usuario, hay que destruirlo y crear uno nuevo
- Tendríamos que hacer comprobación de tipos (`instanceof`) para comprobar el tipo de usuario cada vez que se ejecuta una funcionalidad

Autentificación de usuarios

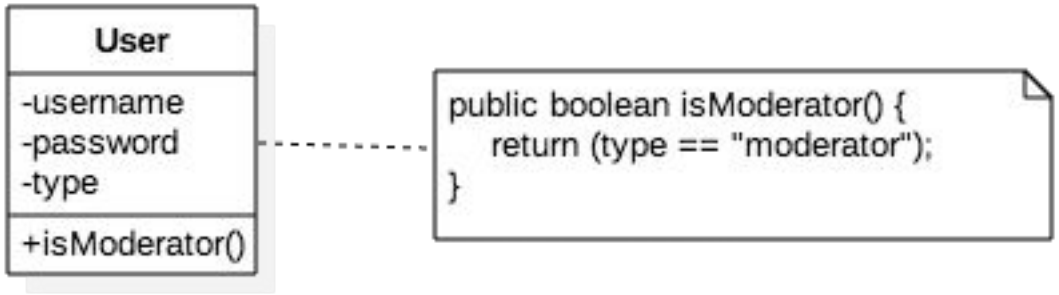
- Cuando el usuario pide acceso a una funcionalidad se hacen dos comprobaciones:
  - **Autentificación**: comprueba la identidad del usuario (recuperando el objeto `User` de la sesión activa)
  - **Autorización**: comprueba si el usuario identificado tiene permisos para acceder a la funcionalidad solicitada
- Al dibujar una pantalla se muestran únicamente los elementos a los que el usuario tiene acceso

Autentificación de usuarios

- Si la autentificación falla se redirige al usuario al formulario de login
- Cuando el usuario introduce las credenciales correctas se crea una instancia de la clase `User` y se almacena en la sesión (objeto global que almacena información compartida entre todas las pantallas)

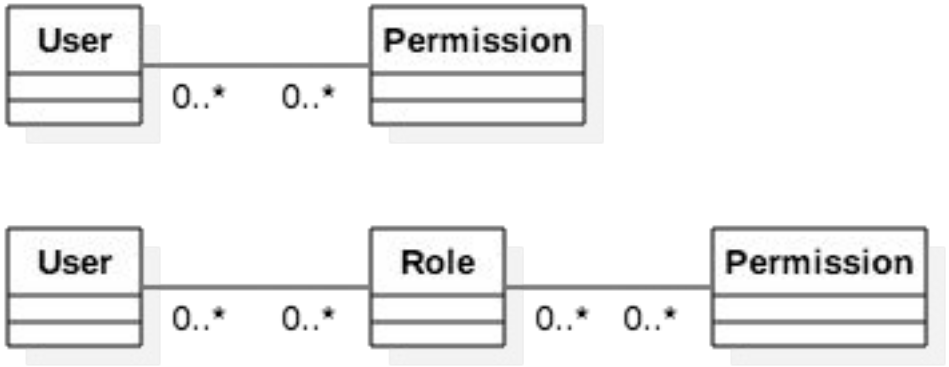
Autentificación de usuarios

- La clase `User` no implementa las funcionalidades, se usa para otorgar acceso a las pantallas que las ofertan
- Se puede simplificar el diseño usando una única clase



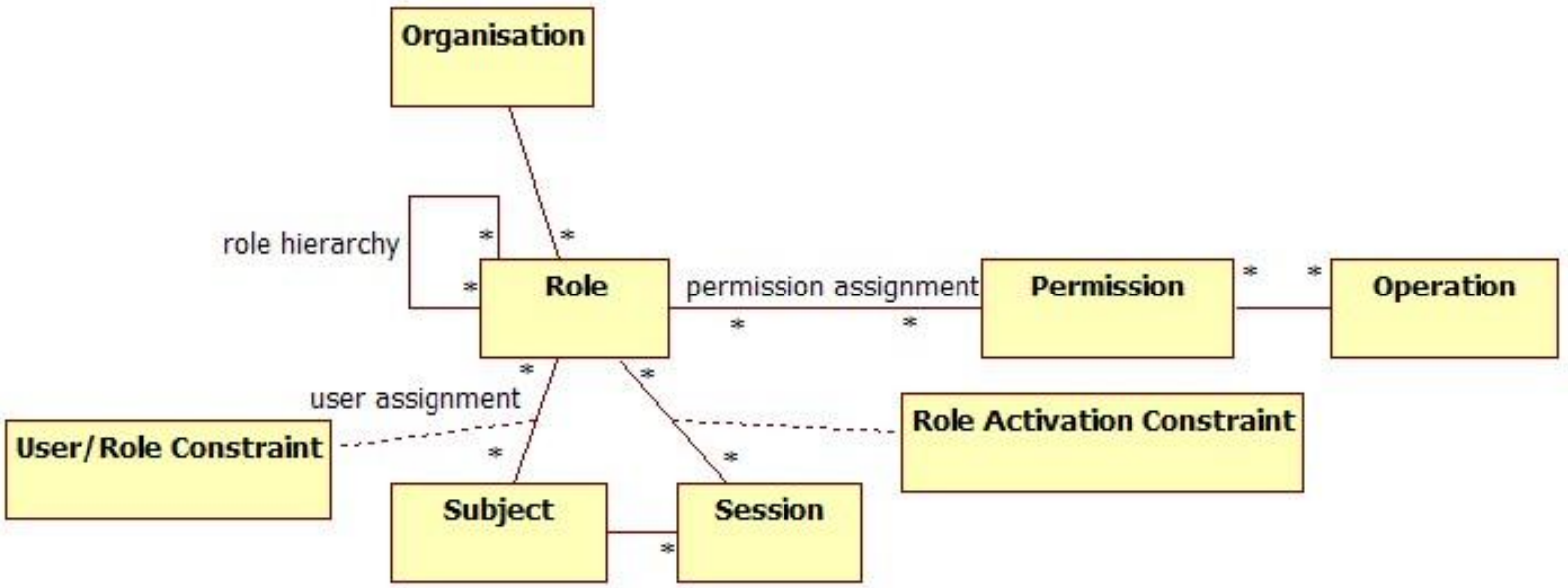
Autentificación de usuarios

- Si queremos tener mayor control sobre lo que pueden hacer los usuarios podemos añadir permisos



Autentificación de usuarios

- Generalización: patrón Role-Based Access Control (RBAC) (estándar NIST)



Fuente: Wikipedia

¿Preguntas?

Bibliografía

- Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional.
  Leer en Safari Books Online