Watch: **Refactorizando a Value Objects**

Refactorizando a Value Objects

Seguimos trabajando pasito a pasito en nuestro caso de uso "Registrar un nuevo estudiante", y en este video vamos a refactorizar a **Value Objects** el código que vimos en la lección anterior

Clase `StudentSignUpper`:

```
final class StudentSignUpper
{
    private static $students;

    public function __invoke(string $studentId, string
$studentName, string $studentPassword){
        $existentStudent = get($studentId, self::$students);

        if(null !== $existentStudent)
        {
            throw new StudentAlreadyExist($studentId)
        }

        $student = new
Student($studentId,$studentName,$studentPassword);

        self::$students[$studentId] = $studentId;
    }
}
```

Partimos del caso de uso `StudentSignUpper`, que simplemente recibía los escalares. Sin embargo, nos encontramos con que tenemos las siguientes reglas de negocio:

- La password debe tener al menos 5 caracteres
- La password debe contener al menos un valor numérico

```
final class StudentSignUpper
{
    private static $students;
```

```
  public function __invoke(string $studentId, string
$studentName, string $studentPassword)
  {
    if(\strlen($studentPassword) > 5)
    {
      if(1 === preg_match('~[0-9]~'))
      {
        $existentStudent = get($studentId, self::$students);

        if(null !== $existentStudent)
        {
          throw new StudentAlreadyExist($studentId);
        }

        $student = new
Student($studentId,$studentName,$studentPassword);

        self::$students[$studentId] = $studentId;
      } else {
        throw new \RuntimeException('The password should have
at least a digit');
      }

    } else {
      throw new \RuntimeException('The password should have
more than 5 characters');
    }

    }
}
```

Cláusulas de guarda

Por supuesto, cada nueva condición a validar supondría un nuevo bloque `if-else` que añadiría otro nivel más indentación, llegando a ser una auténtica locura en términos de cláusulas de guarda

Este planteamiento de código nos va a suponer enfrentarnos a una *pila de ejecución mental* 🤯: Para evaluar las condiciones bajo las cuales se va a realizar la acción en nuestro caso de uso, tendremos que tener en cuenta todas y cada una de las condiciones anidadas. Además con este código nos encontraríamos con que el tratamiento de los errores estará muy alejado de la comprobación de dichos errores

Por ello antes de dar el paso a los Value Objects vamos a refactorizar este código. Invertiremos el código dentro de los `if-else` con idea de extraer posteriormente cláusulas de guarda:

```
final class StudentSignUpper
{
    private static $students;

  public function __invoke(string $studentId, string
$studentName, string $studentPassword)
  {
    if(\strlen($studentPassword) > 5)
    {
      throw new \RuntimeException('The password should have
more than 5 characters');
    }
    {
    if(1 !== preg_match('~[0-9]~'))
    {
      throw new \RuntimeException('The password should have at
least a digit');
    }

    $existentStudent = get($studentId, self::$students);

    if(null !== $existentStudent)
    {
      throw new StudentAlreadyExist($studentId);
    }

    $student = new
Student($studentId,$studentName,$studentPassword);

    self::$students[$studentId] = $studentId;

    }
}
```

Como vemos, no sólo estamos acercando el tratamiento de los casos excepcionales a la detección de estos casos, sino que también hemos llevado al primer nivel de indentación la acción por la cual existe el caso de uso (En este caso crear un nuevo estudiante)

El siguiente paso que llevamos a cabo es como decíamos extraer las cláusulas de guarda

```
final class StudentSignUpper
{
  // ...

  private function ensurePasswordHasAtLeast5Characters(string
$studentPassword)
  {
    if(\strlen($studentPassword) < 5)
    {
      throw new \RuntimeException()
    }
  }

  private function ensurePasswordHasAtLeast1Numer(string
$studentPassword)
  {
    if(1 !== preg_match('~[0-9]~'))
    {
      throw new \RuntimeException('The password should have at
least a digit');
    }
  }

  private function ensureStudentDoesntExist(string $studentId)
  {
    $existentStudent = get($studentId, self::$students);
    if(null !== $existentStudent)
    {
      throw new StudentAlreadyExist($studentId);
    }
  }
}
```

Estos métodos nos van a permitir ganar en calidad de código (*reveal intention*). Además, el hecho de definirlos como *void* nos estaría indicando que van a producir un side efect como lanzar una excepción o guardar algo en BD

Value Objects

Una vez que tenemos un mínimo de semántica, llevaremos estas responsabilidades a donde le toca y crear nuestros Value Objects

Clase `StudentPassword`:

```
final class StudentPassword
{
  private $value;

  public function __construct(string $value)
  {
    ensurePasswordHasAtLeast5Characters($value)
    ensurePasswordHasAtLeast1Numer($value)

    $this->value = $value;
  }
  private function ensurePasswordHasAtLeast5Characters(string
$studentPassword){
    // ...
  }
  private function ensurePasswordHasAtLeast1Numer(string
$studentPassword){
    // ...
  }
}
```

Encapsulando estas responsabilidades dentro del Value Object, no sólo conseguimos evitar la duplicidad de código al hacer estas comprobaciones de validación en un sólo punto de nuestro código, sino que además, lo estaremos haciendo dentro del propio constructor, por lo que estaremos cortando el flujo en el momento en que tengamos un candidato para ese valor y no esté cumpliendo con nuestras restricciones.

Ahora, en todos los clientes de `StudentPassword` nos estamos asegurando de que se cumplen esas condiciones de validación sin tener que repetir la misma lógica para respetar las reglas de integridad establecidas.

```
final class StudentSignUpper
{
    private static $students;

  public function __invoke(string $studentId, StudentName
$name, StudentPassword $password)
  {
    $student = new Student($studentId,$name,$password);

    self::$students[$studentId] = $studentId;
```

```
    }
    // ...
```

Nuestro caso de uso ahora no solo está mucho más limpio, sino que en la firma del `invoke` ya podemos ver de que clase son los parámetros que espera recibir y podemos prescindir del prefijo *student*

Resumen

A modo de breve resumen podemos concluir que los Value Objects:

1. Nos van permitir no tener que duplicar las validaciones en nuestra aplicación
2. Añaden semántica en las firmas de métodos
3. Podremos encapsular fragmentos de lógica y empujarla hacia nuestro Dominio

¿Alguna Duda?

Si tienes cualquier duda sobre este video, sugerencias o te gustaría compartir tu opinión, puedes abrir una nueva discusión justo aquí abajo 👇 👇

Nos vemos en el siguiente video: ¿Dónde poner validaciones?