Informe Práctica Refactoring

MPWAR 2018

Desarrollo Eficiente del Software

Adrià Velardos Palomar

Code smells

Dispensables

Comments

En el código original se pueden encontrar comentarios que explican lo que hacen las líneas de código posteriores. El hecho de que se añadan comentarios puede ser derivado de que el nombre de los métodos, variables y demás código no es lo suficientemente explícito como para entender a qué funcionalidad o variable hacen referencia. A partir de esta premisa haremos un refactor eliminando los comentarios y dotando de significado los nombres de variables, métodos y clases tal y como promueve DDD.

```
/*
 * -prepare password to be saved
 * -concatinate the salt and entered password
 */
```

Duplicate code

La preparación de los parámetros de la consulta a la base de datos está duplicada en el registro y en el login. Si en un futuro se añade un campo nuevo, como podría ser la validación de un captcha, deberíamos de añadirlo en ambos casos de uso. Esto puede generar problemas en el futuro. El refactor consistirá en unificar el código duplicado mediante la extracción de métodos o clases.

```
// prepare query
$query = "select email, password from users where email = ? limit 0,1";
$stmt  = $con->prepare($query);

// this will represent the first question mark
$stmt->bindParam( parameter: 1,  &variable: $_POST['email']);

// execute our query
$stmt->execute();

// count the rows returned
$num = $stmt->rowCount();
```

Bloaters

Long method

En el registro y en el login nos encontramos con más de 60 líneas de código para hacer ambos casos de uso cuando el método de envío del formulario ha sido mediante post. El refactor consistirá en extraer métodos y/o pasar parte de la lógica al objeto pertinente.

```php
try {
    // load database connection and password hasher library
    require 'libs/DbConnect.php';
    require 'libs/PasswordHash.php';

    // prepare query
    $query = "select email, password from users where email = ? limit 0,1";
    $stmt  = $con->prepare($query);

    // this will represent the first question mark
    $stmt->bindParam( parameter: 1,  &variable: $_POST['email']);

    // execute our query
    $stmt->execute();

    // count the rows returned
    $num = $stmt->rowCount();

    if ($num == 1) {

        //store retrieved row to a 'row' variable
        $row = $stmt->fetch( fetch_style: PDO::FETCH_ASSOC);

        // hashed password saved in the database
        $storedPassword = $row['password'];

        // salt and entered password by the user
        $salt                = "ilovecodeofaninjabymikedalisay";
        $postedPassword      = $_POST['password'];
        $saltedPostedPassword = $salt . $postedPassword;

        // instantiate PasswordHash to check if it is a valid password
        $hasher = new PasswordHash( iteration_count_log2: 8,  portable_hashes: false);
        $check  = $hasher->CheckPassword($saltedPostedPassword, $storedPassword);

        /*
         * access granted, for the next steps,
         * you may use my php login script with php sessions tutorial :)
         */
        if ($check) {
            echo "<div>Access granted.</div>";
        } // $check variable is false, access denied.
        else {
            echo "<div>Access denied. <a href='login.php'>Back.</a></div>";
        }
    } // no rows returned, access denied
```

Refactorings

Extract Method

Se han aplicado varios Extract Method para trozos de código que o bien estaban duplicados, o su lectura resultaba complicada dentro de un método con muchas líneas. De esta manera mejoramos la legibilidad del código, aislamos partes independientes del código y ahorramos duplicidad. Un ejemplo de este refactor es la consulta para encontrar usuarios registrados a partir del email, en la primera captura vemos como esta funcionalidad está integrada con el resto del código, y en la segunda como está extradio a el método *findUserByEmail*.

```
// prepare query
$query = "select email, password from users where email = ? limit 0,1";
$stmt  = $con->prepare($query);

// this will represent the first question mark
$stmt->bindParam( parameter: 1,  &variable: $_POST['email']);

// execute our query
$stmt->execute();

// count the rows returned
$num = $stmt->rowCount();
```

```
public function findUserByEmail(Email $email)
{
    $query = "select email, password from users where email = :email limit 0,1";
    $statement  = $this->connection->prepare($query);
    $email = filter_var($_POST['email'], filter: FILTER_SANITIZE_EMAIL);

    $statement->bindParam( parameter: ':email',  &variable: $email,  data_type: \PDO::PARAM_STR);

    $statement->execute();

    return $statement;
}
```

Replace data with object

Se ha creado un Value Object para Email, de esta manera tanto la información del correo electrónico, que en este caso es una cadena de texto, como el comportamiento están en el mismo objeto.

```php
final class Email
{
    private $email;

    public function __construct($email)
    {
        $this->email = $email;
    }

    public function getEmail()
    {
        return $this->email;
    }

    public function __toString()
    {
        return $this->email;
    }

    public function validate()
    {
        if (!filter_var($this->email, filter: FILTER_VALIDATE_EMAIL))
        {
            throw new \InvalidArgumentException('Please, introduce a valid
        }
    }
}
```

```php
final class User
{
    private $email;
    private $hashPassword;

    public function __construct(Emai

    public static function register(

    public function email()
    {
        return $this->email;
    }

    public function hashCode()
    {
        return $this->hashPassword;
    }
```

Encapsulated Field

Se han usado campos encapsulados para todas las clases generadas de nuevo. De esta manera los datos y los comportamientos que los modifican estan agrupados dentro de la misma clase haciendo que sean más mantenibles.

En la captura anterior podemos ver como los campos *$email* y *$hashPassword* son privados y para acceder a ellos se devuelven mediante dos métodos públicos.

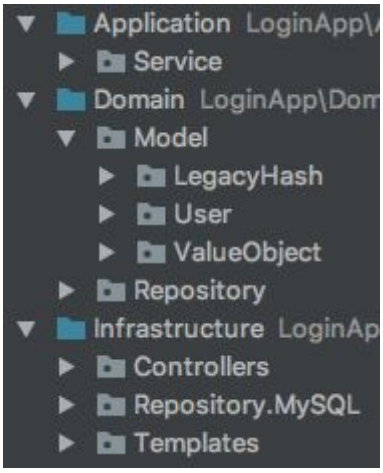
Replace Nested Conditional with Guard Clauses

Se han usado cláusulas de guarda para condicionales anidados en el proceso de validación del login. Una vez hecho el refactor el código quedaría como en la siguiente captura:

```php
public function __invoke(Email $email, $password):bool
{
    $response = $this->repository->findUserByEmail($email);

    if($response->rowCount() !== $this::ONE_USER){
        return false;
    }

    $row = $response->fetch(\PDO::FETCH_ASSOC);
    $dbEmail = new Email($row['email']);
    $dbUser = new User($dbEmail,$row['password']);

    if($dbUser->validate($password)){
        return true;
    }

    return $dbUser->validateLegacy($password);
}
```

En la imagen anterior se puede observar como los *return* son devueltos si se cumple la condición, con lo que solo se ejecuta el código por orden de anidamiento.

Arquitectura de Software



La arquitectura de software implementada en el código se basa en Arquitectura Hexagonal y siguiendo los principios SOLID y DDD.

Observando la estructuración por carpeta se pueden discernir las tres capas que conforman la base de la Arquitectura Hexagonal. Se ha dividido el dominio y la logica de negocio de la infraestructura.

En la capa de Dominio se encuentran los modelos de datos y los repositorios que definen los puertos mediante interfaces. Estas serán implementadas en la capa de infraestructuras con adaptadores para la tecnología pertinente. En está misma capa se hallan los controladores de cada vista y las plantillas para la UI.

Sobre el DDD aplicado a este refactor, se puede destacar a nivel táctico el hecho de basarse en Arquitectura Hexagonal. Por otro lado, a nivel estratégico se ha intentado usar un lenguaje ubicuo propio del dominio.

Los principios SOLID se han seguido para hacer que cada clase tenga el mínimo de responsabilidades, el código esté abierto a extensión pero el mínimo a modificación, sobretodo en el caso de dependencias de infraestructura.