DP2-G3-14

REPORT DE REFACTORIZACIÓN

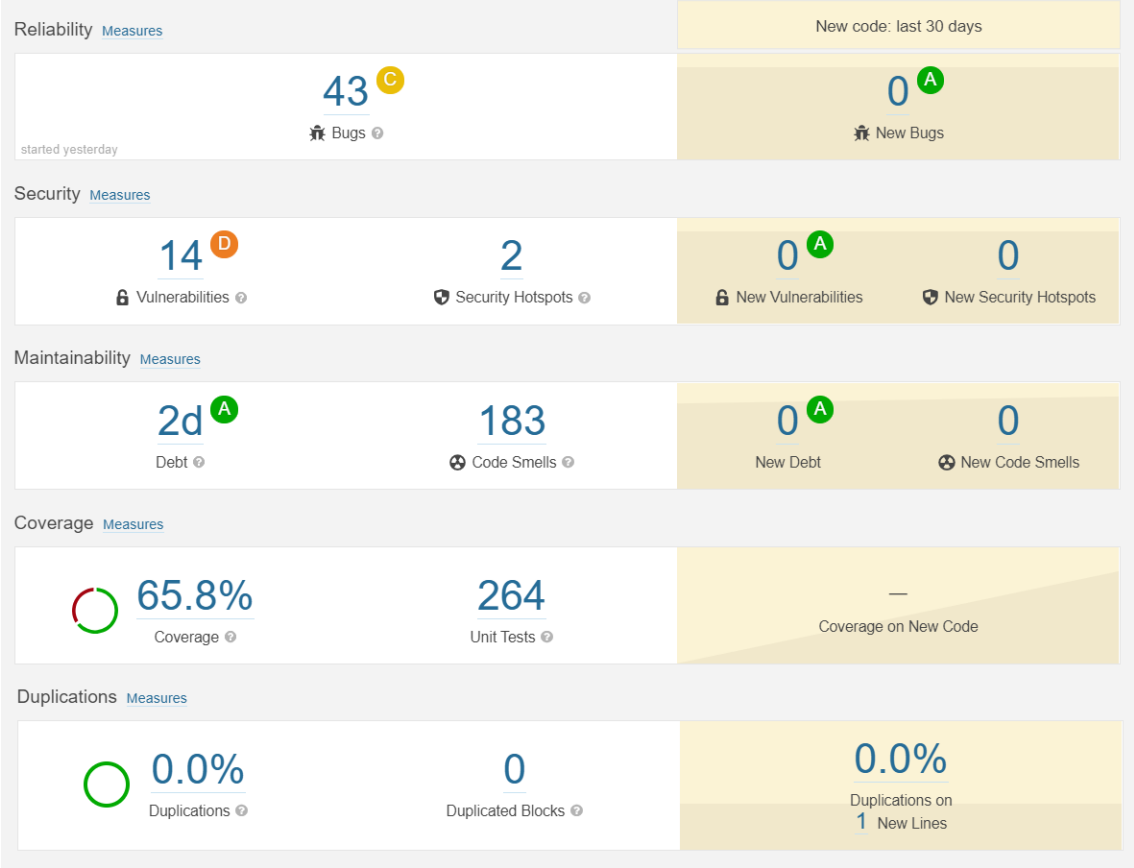
https://github.com/fersolesp/DP2-G3-14

Reyes Blasco Cuadrado
Pablo Cardenal Gamito
José María Cornac Fisas
Vanessa Pradas Fernández
Fernando Luis Sola Espinosa

Contenido

1. Análisis previo a la refactorización

Tras realizar un análisis con “SonarCloud”, hemos obtenido los siguientes resultados:



Al acceder a la información sobre los “code smells” obtenemos la siguiente distribución respecto a su prioridad:

Prioridad	Número de “code smells”	Prioridad	Número de Code Smells
Blocker	2	**Critical**	17
Major	22	**Minor**	108
Info	34		

Tras analizar los resultados decidimos solucionar los “code smells” pertenecientes a “blocker” y “critical”, y reducir los respectivos a “major” y “minor”.

2. Code smells

2.1. Blockers

A continuación, se muestran los “code smells” de tipo “blocker”:

En esta imagen, puede observarse que las excepciones en los “asserts” se encuentran en los test de servicio de los “Appointments”, ya que se ha realizado un uso incorrecto, y no está funcionando como se espera.

Se pretende que el primer “assertThat” se aplique únicamente a la existencia o no del “appointment”, donde el papel que jugaría el método “.isPresent()” se llevaría a cabo sobre el propio “assert”, no sobre dicho “appointment”.

Por otro lado, el segundo “assertThat” no está cumpliendo ninguna función, por lo que será reemplazado por “assertThrows”, que lanzará una excepción “NoSuchElementException” ya que no se encuentra el “appointment” que se ha eliminado anteriormente.

Situación inicial:

```
org.assertj.core.api.Assertions.assertThat(appointment.isPresent());
this.appointmentService.deleteAppointment(appointment.get());
org.assertj.core.api.Assertions.assertThat(!appointment.isPresent());
```

Solución:

```
org.assertj.core.api.Assertions.assertThat(appointment).isPresent();
this.appointmentService.deleteAppointment(appointment.get());
Assertions.assertThrows(NoSuchElementException.class, () -> {
    this.appointmentService.findAppointmentById(id);
});
```

2.2. Criticals

Nos disponemos a solventar un conjunto de “code smells” del mismo tipo: se trata del uso de una misma cadena “string” repetidas veces en la misma clase. Estaríamos entonces ante “code smells” del tipo “change preventers” que nos harán tener que cambiar el mismo valor en diferentes lugares. Son los siguientes:

Para resolver, por ejemplo, los de la clase “AppointmentController”, definimos las cadenas de caracteres que usaremos varias veces:

```
private static final String MESSAGE = "message";
private static final String EXCEPTION = "exception";
private static final String APPOINTMENT = "appointment";
```

Una vez hecho esto, las sustituimos en todos los lugares necesarios:

```
    try {
        appointment = this.appointmentService.findAppointmentById(appointmentId).get();
        if (!authentication.getName().equals(appointment.getOwner().getUser().getUsername())) {
            modelMap.addAttribute(AppointmentController.MESSAGE, "You cannot access another user's appointment");
            return AppointmentController.EXCEPTION;
        } else {
            String vista = "appointments/appointmentDetails";
            modelMap.addAttribute(AppointmentController.APPOINTMENT, appointment);
            return vista;
        }
    } catch (NoSuchElementException e) {
        modelMap.addAttribute(AppointmentController.MESSAGE, "Appointment not found");
        return AppointmentController.EXCEPTION;
    }
}
```

Hemos aplicado una refactorización del tipo “simplifying method calls” puesto que creamos una variable que reutilizamos en lugar de repetir el mismo valor en varios lugares. Repetimos este proceso para todas las clases donde sea procedente realizarlo y, de esta forma, mejoramos la calidad de nuestro código de tal forma, que si, por ejemplo, tenemos que modificar alguna de estas cadenas de caracteres, no tendremos que hacerlo en todas sus apariciones, sino únicamente donde la definimos.

2.3. Majors

Nos disponemos a solucionar los siguientes code smells que pertenencen a una prioridad “Major”.

Como podemos ver, el “code smell” “Either remove or fill this block of code” se repite en varios casos. Este se resuelve de la siguiente forma:

- En el caso de que el “code smell” se encuentre en un controlador se añade un atributo en el modelo que nos indique que no se ha podido validar los datos correctamente. Este cambio se repite en “AnswerController”, “AppointmentController” e “InscriptionController”.

```
try {
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();
    owner = this.ownerService.findOwnerByUserName(auth.getName());
} catch (NoSuchElementException e) {
    modelMap.addAttribute("message", "There are errors validating data");
}
```

- Si el “code smell” se encuentra en un servicio, añadimos una línea con “System.out.println(e)”;

```
try {
    inscriptions = this.findInscriptionsByCourse(inscription.getCourse());
} catch (NoSuchElementException e) {
    System.out.println(e);
}
```

En el controlador de “Pet” encontramos un método comentado, procedemos a eliminarlo:

```
/*
 * @ModelAttribute("pet")
 * public Pet findPet(@PathVariable("petId") Integer petId) {
 * Pet result=null;
 * if(petId!=null)
 * result=this.clinicService.findPetById(petId);
 * else
 * result=new Pet();
 * return result;
 * }
 */
```

Este “code smell” es del tipo “dispensable”, código que no aporta nada y ocupa espacio. Su eliminación, lógicamente aporta un mayor nivel de limpieza y claridad, a la vez que logra disminuir la complejidad cognitiva.

2.4. Minors

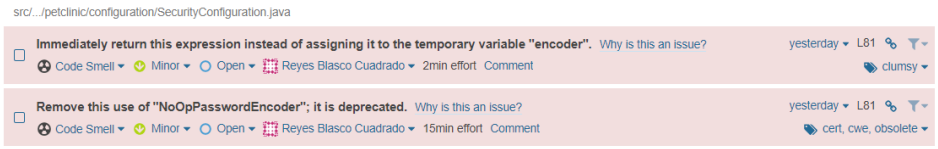
Encontramos 8 tipos de “code smells” de los cuales vamos a cubrir 6 de ellos. Incluiremos algunos ejemplos de cada tipo:

- **“Remove this unused imports”:** “code smell” del tipo “dispensable” que indica que se está importando una clase la cual no está siendo utilizada, por lo que no es necesaria. La solución trata de eliminar todos estos “imports” no utilizados.

Ejemplo de líneas a eliminar:

```
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
```

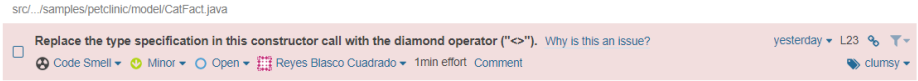
- **“Inmediately return”:** Indica que se ha inicializado una variable para un método de la cuál podemos prescindir debido a la complejidad de este. La solución vendrá dada por la eliminación de esta variable:



Ejemplo: Se eliminaría la variable “encoder” y sólo quedaría la línea “return NoOpPasswordEnconder.getInstance();”dentro del método

```
@Bean
public PasswordEncoder passwordEncoder() {
    PasswordEncoder encoder = NoOpPasswordEncoder.getInstance();
    return encoder;
}
```

- **“Replace the type specification in this constructor call with the diamond operator ("<>")”:** Indica que el tipo que se indica dentro de los símbolos de “< >” son inferidos en la compilación con lo cual se pueden dejar vacíos sin ningún tipo de problema.



Situación inicial:

```
private Map<String, Object> additionalProperties    = new HashMap<String, Object>();
```

Solución:

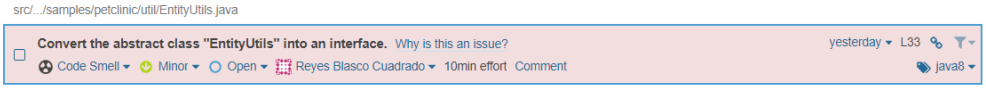
```
private Map<String, Object> additionalProperties    = new HashMap<>();
```

- **“Use the primitive boolean expression here”:** Indica que estamos comparando un “boolean” con un “boolean” primitivo dentro de una sentencia “if”.

- **“Remove the literal "true" boolean value”:** La solución de este problema está acompañada por la correspondiente al apartado anterior.

Situación inicial: if (inscription.getPet().getIsVaccinated() == false)
Solución: if (Boolean.FALSE.equals(inscription.getPet().getIsVaccinated()))

- **“Convert the abstract class "EntityUtils" into an interface”:** entendemos que se trata de un “code smell” del tipo “object orientation abusers”, ya que se hace un mal uso de los principios de “OO”, que indica que cualquier clase que no contenga atributos propios ni heredados, debe convertirse en una interfaz.



Situación Inicial:

```
33 public abstract class EntityUtils {
```

Solución:

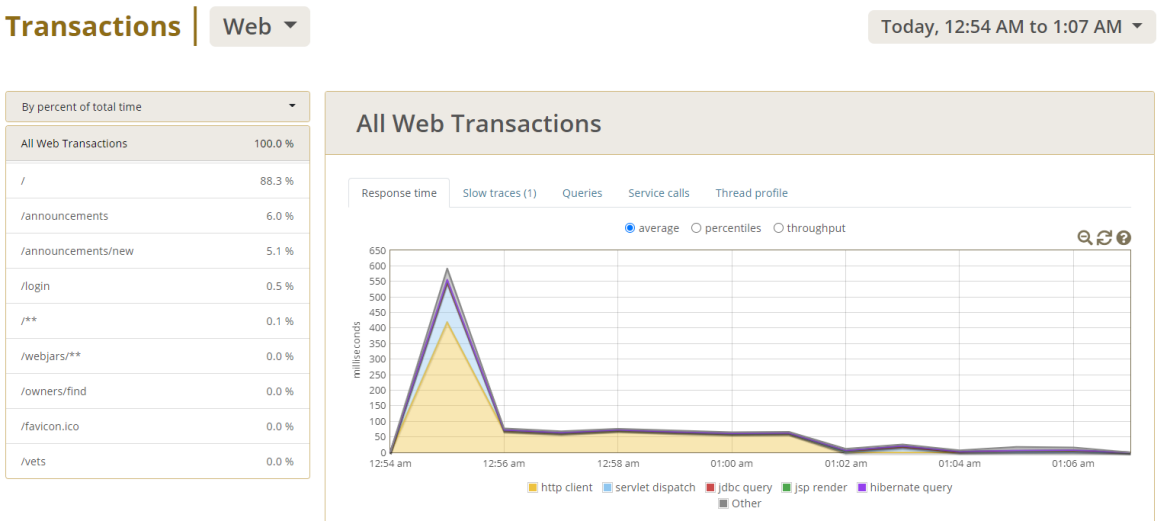
```
33 public interface EntityUtils {
```

3. Refactorización adicional (nivel 9 de acabado) de la HU01

Como hemos explicado en el apartado “Cambios y consideraciones” del “sprint” 4, hemos decidido incluir para el apartado del nivel 9 de acabado, a modo de refactorización del código que mejore el rendimiento, una optimización aplicada a una historia de usuario (la HU01), previo estudio de “profiling”, y ejecutada a través del método de “proyección”.

3.1. PRIMERAS PRUEBAS

Al hacer el análisis en “GrowRoot” de los escenarios definidos en formato “scala”, ejecutados con la ayuda de “Gatling” obtenemos los siguientes resultados:



Podemos ver que, tras la página principal, que supone el 88,3% de las peticiones, es el acceso a “/announcements” (listado de “announcements”) lo que se realiza más veces, con los accesos a base de datos que supone, trayéndonos para cada uno de los “annoucements” la entidad en sí, como el “owner” que lo publicó y el “pet type” de la mascota:

Tiempo total invertido en las “queries” = 2.809,8 + 1.164,8 + 996,2 = 4.970,8 ms

Aunque realmente no obtenemos malos resultados, como podemos ver, estamos realizando “queries” de más. Es por ello, que vamos a aplicar una optimización de tipo “proyección”, de

forma que seleccionemos concretamente los atributos que se muestran en el listado de "announcements" evitando de esta forma traernos de la base de datos entidades completas como el "owner" cuando realmente no lo mostraremos ni usaremos.

3.2. CAMBIOS REALIZADOS (PROYECCIÓN)

Para realizar la proyección, en primer lugar, debemos crear una interfaz que pueda contener los atributos que realmente vamos a mostrar:

```java
package org.springframework.samples.petclinic.projections;


public interface PetAnnouncement {
    String getId();
    String getName();
    String getPetName();
    String getCanBeAdopted();
    String getType();
}
```

Una vez hecho esto, añadimos al repositorio de "Pet" la consulta que nos permita acceder a dichos datos:

```java
    Collection<PetAnnouncement> findAllPetAnnouncements() throws DataAccessException;
}
```

```java
@Override
@Query("SELECT a.id as id, a.name AS name, a.petName AS petName, a.canBeAdopted AS canBeAdopted, t.name AS type FROM Announcement a INNER JOIN a.type t")
Collection<PetAnnouncement> findAllPetAnnouncements() throws DataAccessException;
```

Obviamente, tendría más lógica añadir este método de consulta a la base de datos en el repositorio de "Announcement", pero nos hemos visto obligados a añadirlo a este de "Pet", porque si intentábamos crearlo en un repositorio que nosotros hayamos añadido, es decir, que no viniera en la base del proyecto "PetClinic", lanzaba un error alegando que no encontraba el atributo "findAllPetAnnouncements" en la clase "Announcement", por ejemplo. Esto viene dado porque "Spring Data", por defecto, genera las "queries" a partir del nombre del método, y claro, en este caso no lo lograba porque "PetAnnouncements" no pertenece a "Announcement". Un ejemplo de esto es que para el método de repositorio:

```java
User findByEmailAddress(String emailAddress);
```

Spring generaría la "query":

```
"SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1"
```

Existen formas de desactivar este comportamiento de "Spring" pero no hemos logrado conseguirlo. Sin embargo, como hemos dicho, trasladando el método a una clase de repositorio que ya venía en el proyecto base, logramos que "Spring" no genere ese error al no intentar formar la "query" a partir del nombre del método y ejecutar la "query" que nosotros hemos indicado. Finalmente, descubrimos que la razón de este comportamiento en los repositorios, es que nuestros repositorios extienden de "CrudRepository".

Una vez aclarado esto, debemos añadir un método al servicio de "Announcements" para poder traernos el resultado de nuestra nueva "query":

```java
@Transactional(readOnly=true)
public Iterable<PetAnnouncement> findAllAnnouncements() {
    Iterable<PetAnnouncement> res = this.petRepo.findAllPetAnnouncements();
    if (StreamSupport.stream(res.spliterator(), false).count() == 0) {
        throw new NoSuchElementException();
    }
    return res;
}
```

Y, a continuación, indicamos al controlador que, para mostrar el listado de “Announcements” invoque al nuevo método del servicio:

```java
@GetMapping()
public String mostrarAnnouncements(final ModelMap modelMap) {

    String vista = "announcements/announcementsList";
    boolean isempty = false;
    try {
        Iterable<PetAnnouncement> announcements = this.announcementService.findAllAnnouncements();
        modelMap.addAttribute("announcements", announcements);
        modelMap.addAttribute("isanonymoususer", SecurityContextHolder.getContext().getAuthentication().getName().equals("anonymousUser"));

    } catch (NoSuchElementException e) {
        isempty = true;
        modelMap.addAttribute("isempty", isempty);
    }

    return vista;
}
```

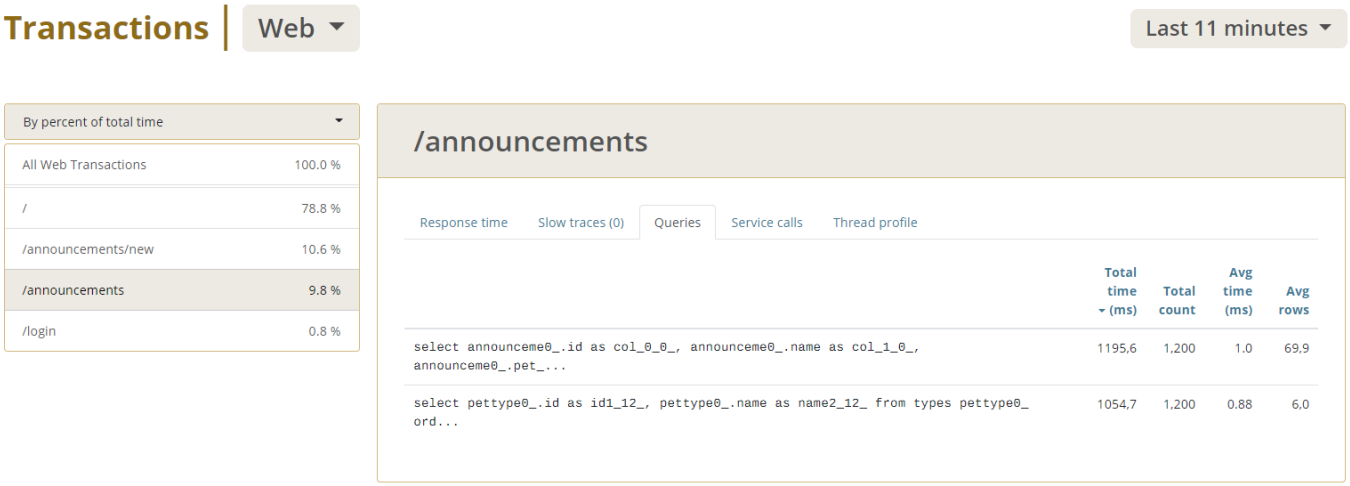
Por último, algo de menor relevancia, es que debemos actualizar la llamada que hace el “Mock” en nuestras pruebas de controlador para que invoque a este nuevo método:

```java
@WithMockUser(value = "spring")
@Test
void shouldNotShowAnnouncements() throws Exception {
    Mockito.when(this.announcementService.findAllAnnouncements()).thenThrow(NoSuchElementException.class);

    this.mockMvc.perform(MockMvcRequestBuilders.get("/announcements")).andExpect(MockMvcResultMatchers.status().isOk()).andExpect(MockMv
    .andExpect(MockMvcResultMatchers.model().attributeDoesNotExist("announcements")).andExpect(MockMvcResultMatchers.view().name("announ
}
```

3.3. RESULTADOS

Una vez hemos realizado estos cambios, lanzaremos de nuevo la aplicación, junto con los scripts de prueba de “Gatling” que usamos anteriormente, y comprobaremos si ha habido alguna mejoría en cuanto al rendimiento con la ayuda de “GlowRoot”. Estos son los resultados que obtenemos ahora:



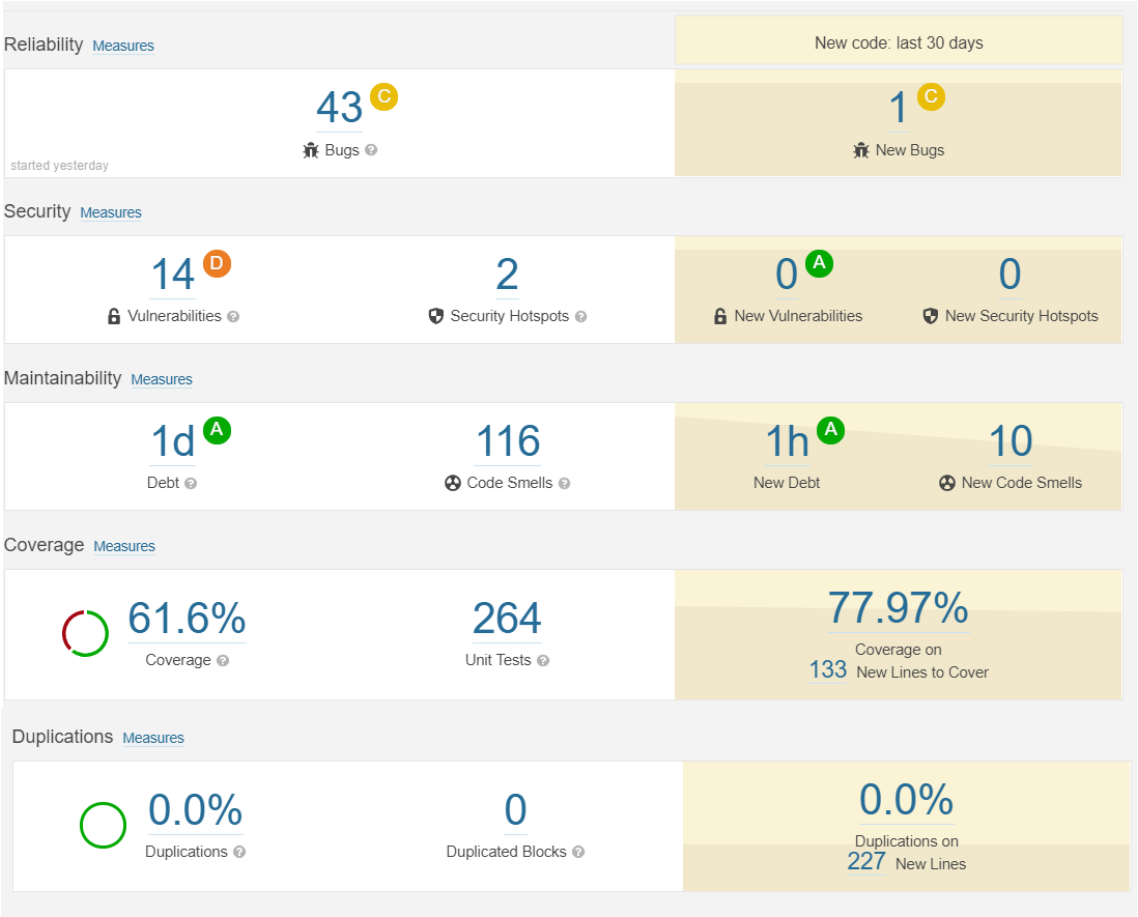
Tiempo total invertido en las “queries” = 1195,6 + 1054,7 = 2250,3 ms

Como podemos ver, ahora hemos invertido menos de la mitad del tiempo total en la ejecución de las “queries” necesarias para el listado de “Announcements” y esto se debe, en gran parte, porque ahora evitamos traernos toda la información de “Owners” que antes sí que hacíamos, ahorrándonos una “query” bastante costosa temporalmente.

Como conclusión, podemos decir que este método de optimización denominado proyección, es útil y eficaz, que tiene un mayor rendimiento en situaciones donde mostremos por pantalla datos de diversas entidades, pero muy pocos de cada una, y que, por lo tanto, debe seleccionarse bien cuándo usarlo.

4. Análisis después de la refactorización

Tras realizar un segundo análisis con SonarCloud tras solucionar los "Code Smell", hemos obtenido los siguientes resultados:



Obteniendo así la siguiente información sobre los Code Smells actuales:

Prioridad	Número de "code smells"	Prioridad	Número de "code smells"
Blocker	0	**Critical**	6
Major	12	**Minor**	64
Info	34		

Aunque se hayan solucionado ciertos "code smells" con prioridad "critical", han surgido nuevos tras realizar este segundo análisis. Finamente, podemos concluir que el número de "code smells" se ha visto reducido en:

- **Blocker**: 2, eliminando por completo este tipo de "code smells".
- **Critical**: 11.
- **Major**: 10.
- **Minor**: 44.