DP2-G3-14

REPORT DE REFACTORIZACIÓN

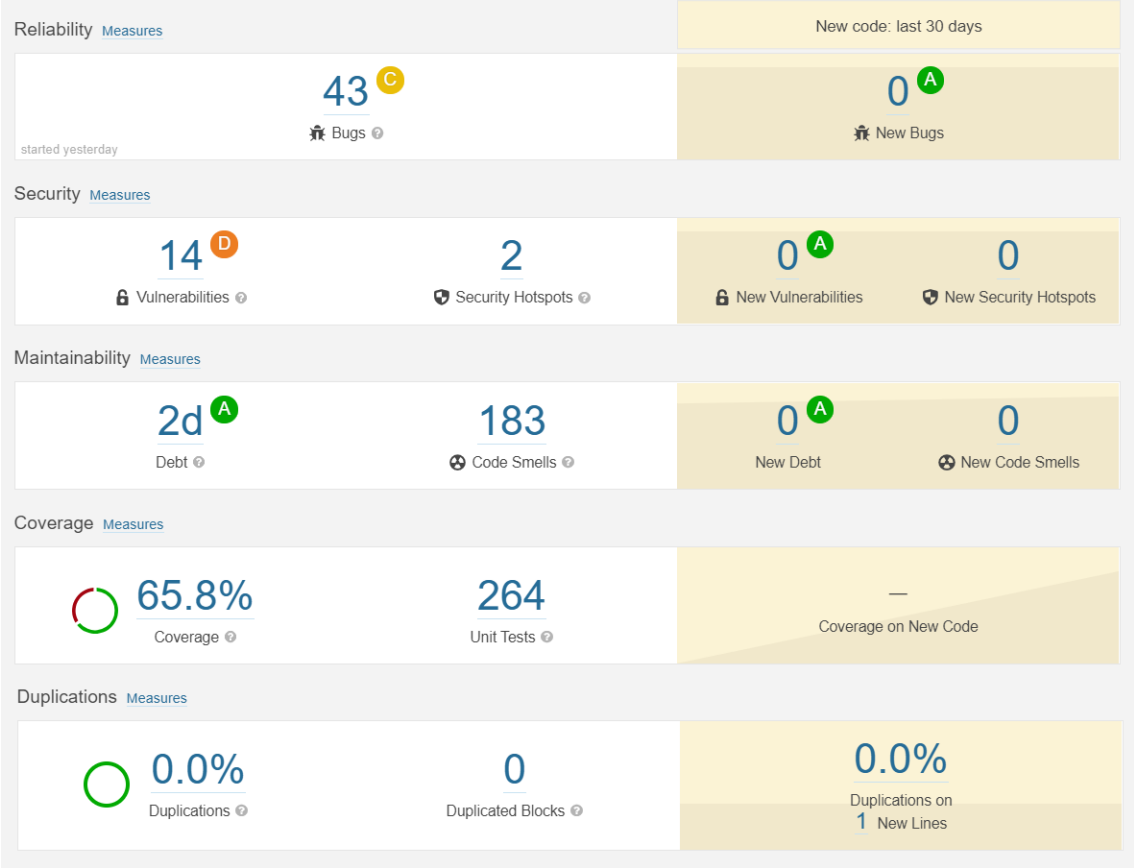
https://github.com/fersolesp/DP2-G3-14

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. 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;
    }
}
```

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;
 * }
 */
```

2.4. Minors

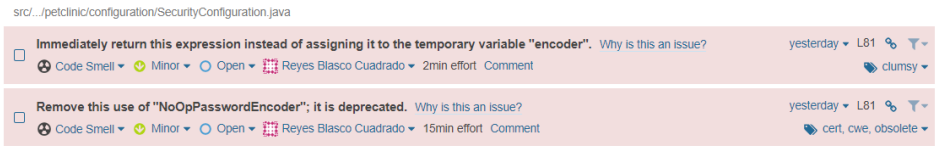
Encontramos 8 tipos de “code smells” de los cuales vamos a cubrir 6 de ellos. Incluiremos algunos ejemplos de cada tipo:

- **“Remove this unused imports”:** 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;
```

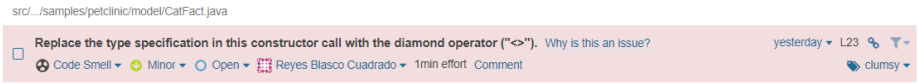
- **“Inmediately return”:** Indica que se ha inicializado una variable para un método de la cuál podemos prescindir debido a la complejidad de este. La solución vendrá dada por la eliminación de esta variable:



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:

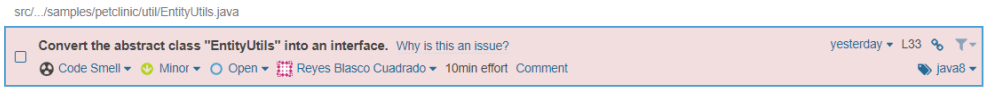
```
private Map<String, Object> additionalProperties    = new HashMap<>();
```

- **“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”:** Indica que cualquier clase que no contenga atributos propios ni heredados, debe convertirse en una interfaz.



Situación Inicial:

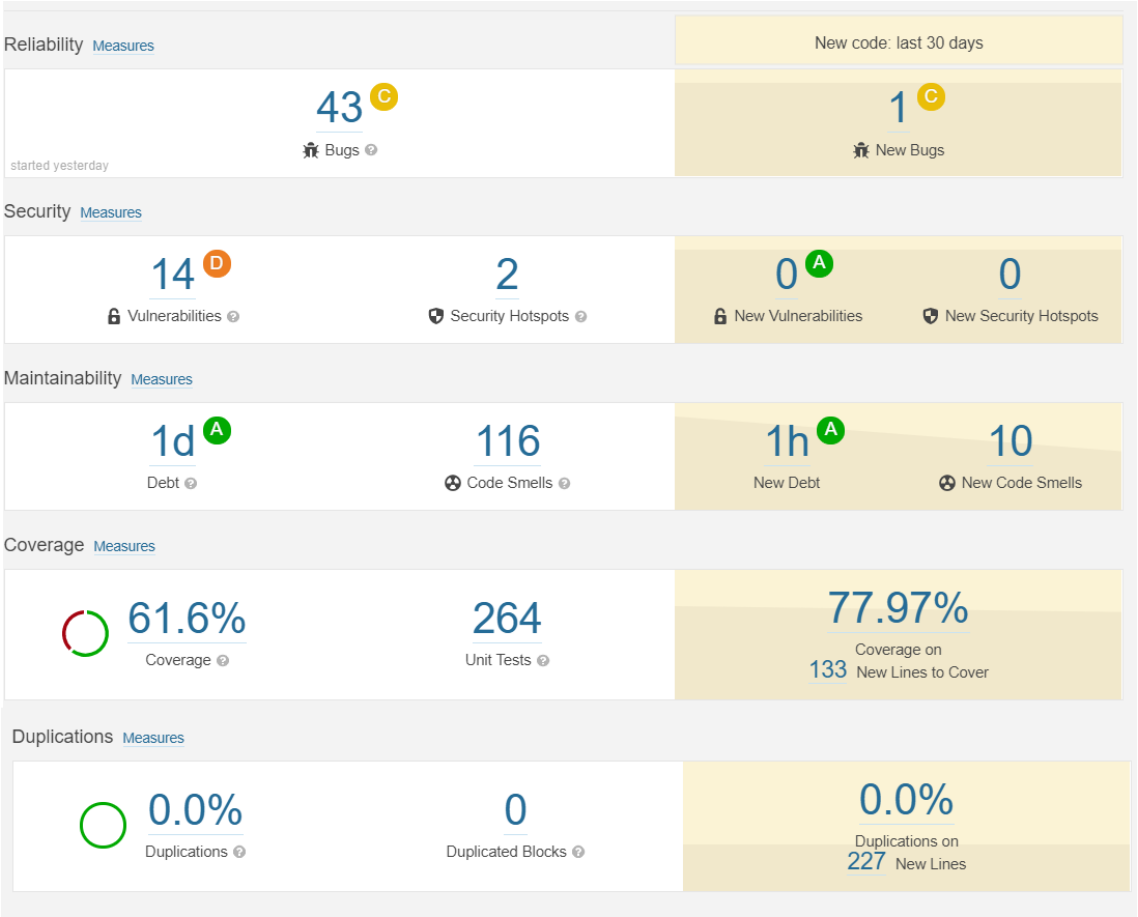
```
33 public abstract class EntityUtils {
```

Solución:

```
33 public interface EntityUtils {
```

3. 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.