DP2 - REFACTORING

GI - 01

Diana Bukša

Manuel Cañizares Juan

Yoana Dimitrova Penkova

Iván Menacho Rubia

Álvaro Rubia Tapia

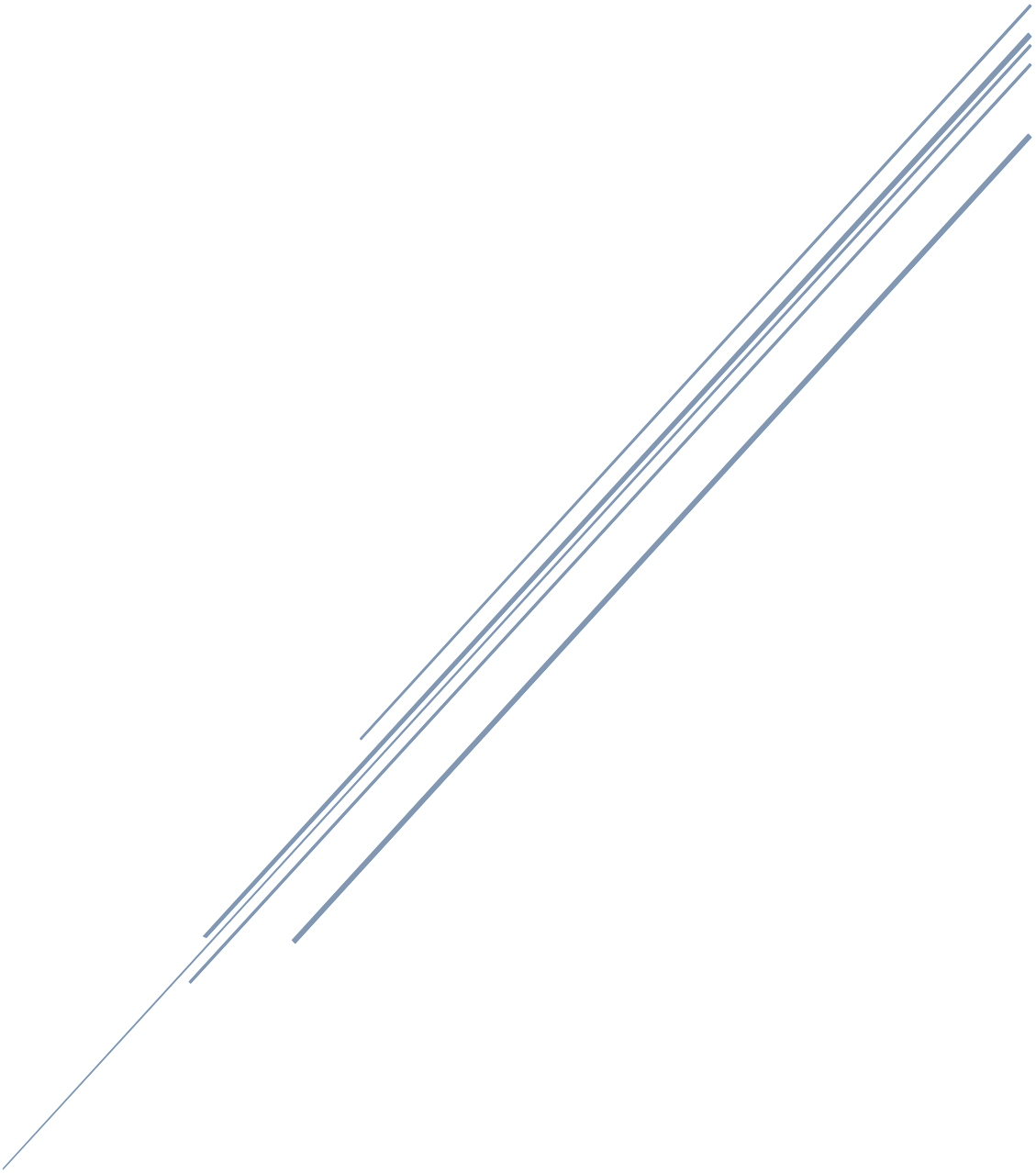

Table of contents

Table of figures

Introduction

This document will study Refactorization done by the group, where there are defined the bad smells we have encountered, how we have solved them, the US which code we have refactored, the methodology we have applied and last the result of the refactoring.

Every member of the group has participated in the refactoring task, each one delivering its own document. Here they are all merged and compared to share the results and a conclusion.

The target of this refactoring is the petclinic application, where each member of the group have perform a refactorization of its own code focusing in a specific US (nevertheless every one of us also has extended the refactoring as much as possible to achieve a better result).

The refactoring has been the last issue to work with in the group, so no more code has been implemented since we started working on it, thus, the result shown in this document can be considered the final results.

SonarCloud

For the refactoring tasks for this deliverable, we had to use some analysis tool. In this case, we used sonar cloud which was very easy to configure and sync with the GitHub accounts. We haven't synced it with Travis CI because I didn't see it that necessary as we're in the final stages of the project. By making manual analysis I thought it suffices.

Therefore, after performing the first analysis of the project, the situation was a little chaotic, as you can observe in the following screenshot:

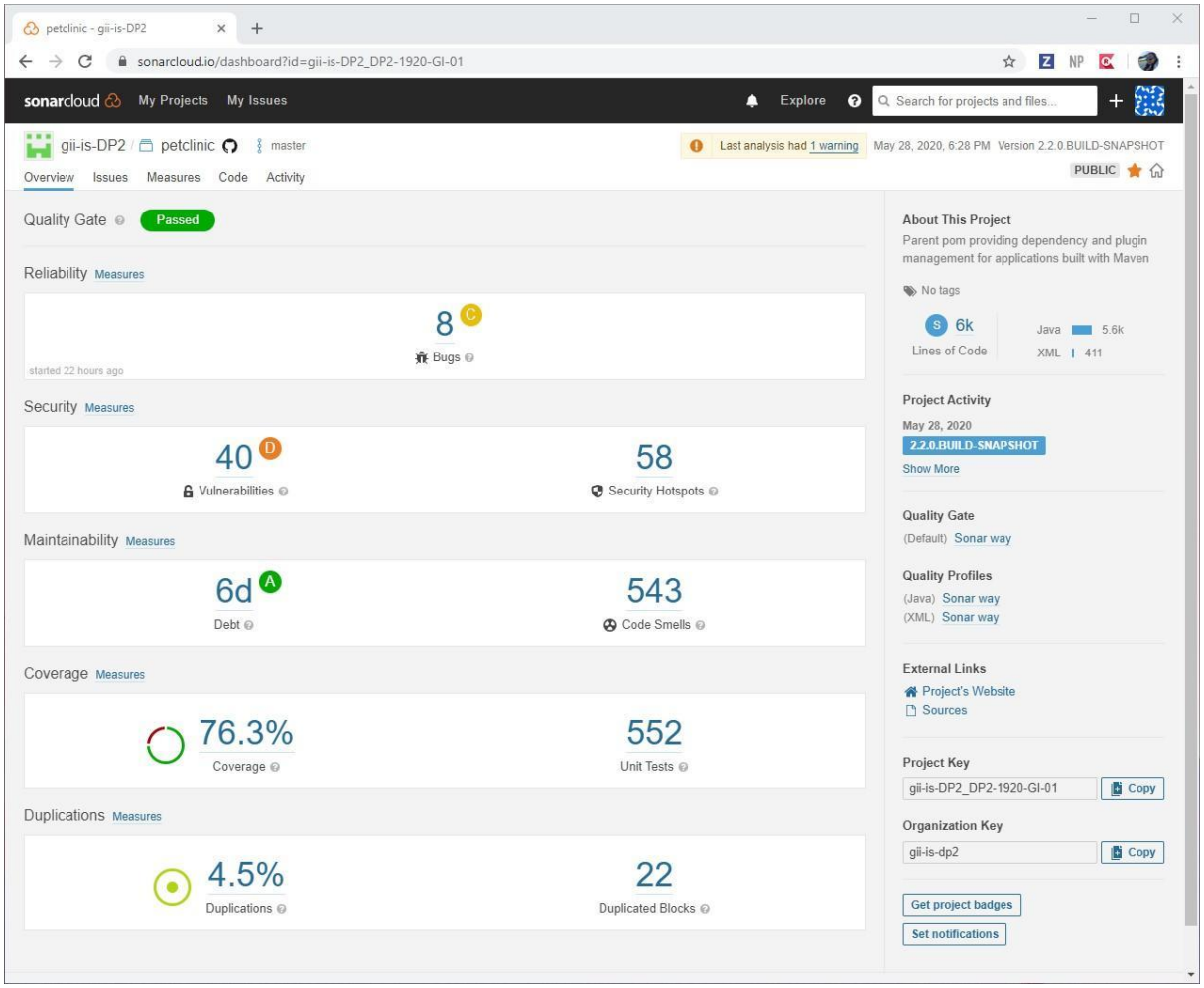

Figure 1 Starting state of the project

The first thing that can catch your eye is those 6 days of technical debt and those 543 code smells. Most of the code smells were minor or just information but they had to be fixed as well.

Every time a refactoring was done, each member of the team had the task to perform another analysis, updating the results and giving the rest of the group more information about the state of the project.

To perform manually the analysis, we followed the videos in EV, but we encountered a problem. When performing the analysis from a terminal, the buffer will ran out of space and return us an error saying the build failed. This error was fixed by adding a new parameter: **mvn verify sonar:sonar -Dsonar.login=<tuTokenDeSonar> > log-file.log**

With the last part, we made the terminal save the results in a log file. This method worked and allowed us to perform the analysis.

Bad Smells

Here are defined every bad smell the group encountered during the refactorization process. Here are defined the causes of the bad smells and how, as a group, we managed to solve them.

Each bad smell has associated the reason why they are considered bad practices (following the documentation of SonarCloud), and two screenshots, one showing how the error appear in the analysis of SonarCloud and the other showing how they were solved.

Some of the error were encountered many times, but for the simplicity of this document in this section only one error and its resolution are shown.

Define constant instead of duplicating literal

Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences.

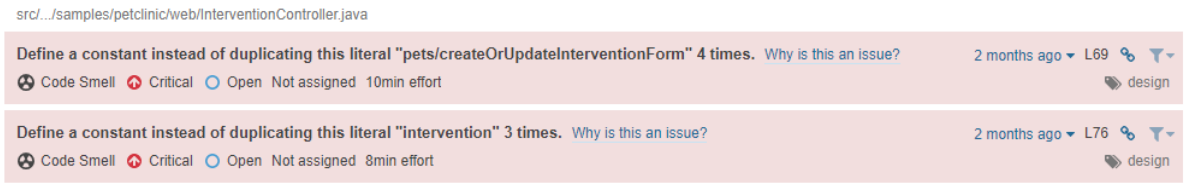

Figure 2 Duplicate literal

On the other hand, constants can be referenced from many places, but only need to be updated in a single place.

This block of commented out lines should be removed

Programmers should not comment out code as it bloats programs and reduces readability.

Unused code should be deleted and can be retrieved from source control history if required.

Figure 3 Commented out lines

Remove this unused import

The imports part of a file should be handled by the Integrated Development Environment (IDE), not manually by the developer.

Unused and useless imports should not occur if that is the case.

Leaving them in reduces the code's readability, since their presence can be confusing.

Figure 4 Unused import

They can be easily fixed removing those imports. In eclipse, an efficient way to do this is using the Ctrl + Shift + O key combination to reorganize them.

Replace the type specification in the constructor call with the literal (<>)

Java 7 introduced the diamond operator (`<>`) to reduce the verbosity of generics code. For instance, instead of having to declare a `List`'s type in both its declaration and its constructor, you can now simplify the constructor declaration with `<>`, and the compiler will infer the type.

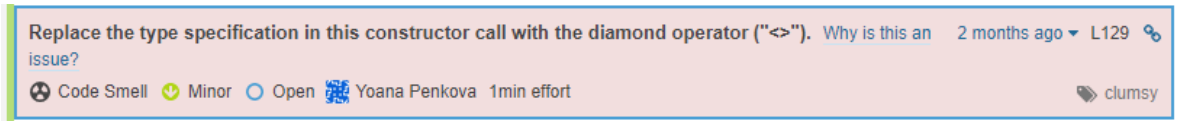

Figure 5 Replace type specification

Use the primitive expression boolean here

When boxed type `java.lang.Boolean` is used as an expression it will throw `NullPointerException` if the value is `null` as defined in Java Language Specification §5.1.8 Unboxing Conversion.

It is safer to avoid such conversion altogether and handle the `null` value explicitly.

Figure 6 Primitive boolean expression

Remove the literal “true” boolean value

Redundant Boolean literals should be removed from expressions to improve readability.

Figure 7 Remove literal true

Remove this useless local variable

A dead store happens when a local variable is assigned a value that is not read by any subsequent instruction. Calculating or retrieving a value only to then overwrite it or throw it away, could indicate a serious error in the code. Even if it's not an error, it is at best a waste of resources. Therefore all calculated values should be used.

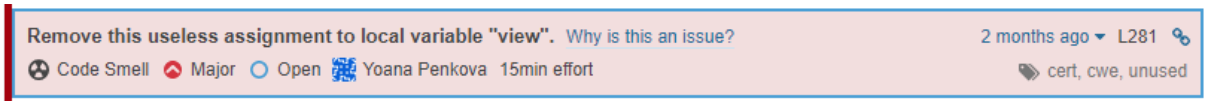

Figure 8 Useless local variable

Remove this unused private field

If a `private` field is declared but not used in the program, it can be considered dead code and should therefore be removed. This will improve maintainability because developers will not wonder what the variable is used for.

Figure 9 Remove unused private field

Either remove or fill this block of code

Most of the time a block of code is empty when a piece of code is really missing. So such empty block must be either filled or removed.

Figure 10 Remove or fill this block of code

Remove this “public” modifier

JUnit5 is more tolerant regarding the visibilities of Test classes than JUnit4, which required everything to be `public`.

In this context, JUnit5 test classes can have any visibility but `private`, however, it is recommended to use the default package visibility, which improves readability of code.

Figure 11 Remove public modifier

Add at least one assertion to this use case

A test case without assertions ensures only that no exceptions are thrown. Beyond basic run ability, it ensures nothing about the behavior of the code under test.

Figure 12 Add at least one assertion to this use case

Remove the declaration of thrown exception “DataAccesException” that is a runtime exception

An exception in a `throws` declaration in Java is superfluous if it is:

- listed multiple times
- a subclass of another listed exception
- a `RuntimeException`, or one of its descendants
- completely unnecessary because the declared exception type cannot actually be thrown

Figure 13 Remove runtime exception

Complete the task associated with this TODO comment

`TODO` tags are commonly used to mark places where some more code is required, but which the developer wants to implement later.

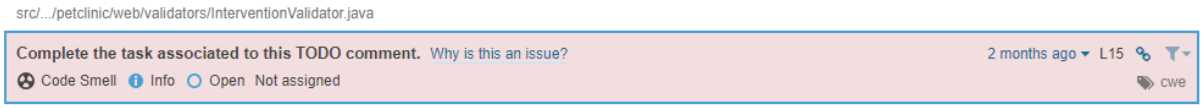

Figure 14 TODO comment

Immediately return this expression instead of assigning it to the temporary variable

Declaring a variable only to immediately return or throw it is a bad practice.

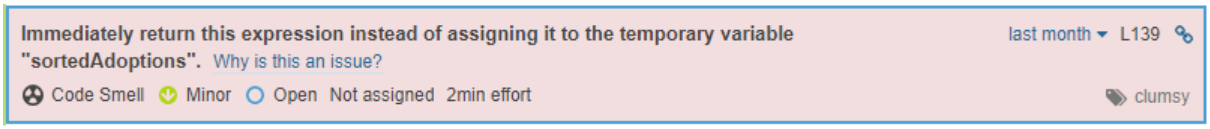

Figure 15 Immediatelly return expression

Methodology

To perform the refactoring, each member of the group opened its own branch (Refactoring-Jonh Doe). We first made a screenshot of the current state when we started working on every one branches. After finish the refactoring, we would merge the branches into the master and then performed another analysis to study the result of the refactoring.

In total, 7 analyses were made. He had some small step backs:

- We were working in parallel, and sometimes, another person would finish the refactoring faster and making the analysis earlier than other person, so for the second person, comparing his first screenshot and the last one had no sense, because he was comparing his results plus the ones of his teammates, so we had to change and the two screenshots were made just before and after the analysis, to have data to work with.

- The second step back was a small mistake of one of the members of the group that forgot to make the analysis in master and made it from his own branch. That raised the bad smells instead of lowering it because some of the bad smells that were already solved in master were unfixed in his branch. This was fixed quickly by performing this time the analysis from master and the issue was closed.

In their reports, every member took notes of the bad smells they encountered and explain of they were solved/or the reason they were not solved.

US-001

This US (Vet adds a new medicine) consistent on the creation of a new medicine and involves every class related to medicine and vet. In addition, when the refactoring of this Us was performed, several US share the same code, so they were all fixed even though the aim was this US (US 001, 002, 007, 008 and 0012).

- **Remove unused imports**

This bad smell is given when we have imports that are not used in the specified class.

They can be easily fixed removing those imports. In eclipse, an efficient way to do this is using the Ctrl + Shift + O key combination to reorganize them.

- **Rename field**

This smell is given when a field is called the same as the class. One example is:

It can be fixed by renaming the field with a more descriptive name. In this case, *vetList* for example.

- **Remove the declaration of a thrown exception which is a runtime exception**

This smell is given when a method throws an exception that can cause a runtime exception and lead in an unexpected system behavior. Some examples are:

It can be fixed by removing the explicit thrown of an exception, like in this method:

```
@Transactional(readOnly = true)
public Owner findOwnerById(final int id) throws DataAccessException {
    return this.ownerRepository.findById(id);
}
```

Which change will result in:

```
@Transactional(readOnly = true)
public Owner findOwnerById(final int id) {
    return this.ownerRepository.findById(id);
}
```

- **Define a constant instead of duplicating literals n times**

This code smell is given when there are duplicated strings with the same goal distributed all over the class, because if an update is needed, you must be sure to change all the occurrences. Some examples are:

This can be fixed adding some constants to the class and use them where the literals were.

```
private static final String MEDICINE_LITERAL = "medicine";
private static final String MEDICINE_NOT_FOUND_MESSAGE = "Medicine not found";
```

- **Remove public modifier**

This smell is given when we set the visibility to public in the test methods and classes using Junit5, which is not necessary since this version. Some examples are:

This can be fixed removing the visibility of all the testing methods and classes and let them to be by default:

```
@Test
void getOrganizationsListTest() {
    given()
        .auth()
            .oauth2(token)
    .when()
        .get("https://api.petfinder.com/v2/organizations")
    .then()
        .statusCode(200)
    .and()
        .assertThat()
            .body("organizations.size()", equalTo(20))
            .body("pagination.count_per_page", equalTo(20))
            .body("pagination.current_page", equalTo(1))
    .and()
        .time(lessThan(20L), TimeUnit.SECONDS);
}
```

US-003

This US has the description (Medical record Creation), and involves every class related to MedicalRecord. As we analyzed the code smells related to the US 003 Creation of a Medical Record we encountered very few, so the group decided to try and solve as many code smells as possible related to Medical Record.
The bad smells encountered were:

- **Duplication of literals**
- **Complete Assertions**
- **Remove public modifier**
- **Remove declaration of thrown exception**

We decided to focus in duplication of literals and the removal of thrown exception, that is because the bad smells gave by the incompletion of assertion worked were more a recommendation that a proper error, and the public modifier, while important and interesting from a security perspective (It might be possible to access some class due to injections of code) they gave us a lot of problems when were changed to private.

Once the code smells were localized, we made the following changes in the code to solve them. There will be only show one example for each change:

Firstly, several string were created which will help to reduce repetition in the code.

```
private static final String     VIEW                = "/owners/*/pets/*/visits/{visitId}/medical-record";
private static final String     MEDICAL_RECORD      = "medicalRecord";
private static final String     PET_QUERY           = "/pets/";
private static final String     REDIRECT_QUERY      = "redirect:/owners/";
private static final String     NOT_FOUND_ERROR     = "MedicalRecord not found";
```

Then, they were implemented in the places where that repetitions were done.

```
@GetMapping(VIEW + "/show")
public String showMedicalRecord(@RequestParam(value = "id", required = true) final Integer medicalRecordId, final ModelMap model) {
    MedicalRecord medicalRecord;
    Collection<Prescription> prescriptions;

    medicalRecord = this.medicalRecordService.findMedicalRecordById(medicalRecordId);
    prescriptions = this.prescriptionService.findManyByMedicalRecord(medicalRecord);

    if (medicalRecord == null) {
        throw new NullPointerException(NOT_FOUND_ERROR);
    }

    model.put(MEDICAL_RECORD, medicalRecord);
    model.put("prescriptions", prescriptions);

    return MedicalRecordController.SHOW_VIEW;
}
```

```
redirection = REDIRECT_QUERY + ownerId + PET_QUERY
```

Finally, thrown declaration were removed from medical record service.

```
@Cacheable("medicalHistory")
@Transactional(readOnly = true)
public Collection<MedicalRecord> findMedicalHistory() {
    return this.repository.findAll();
}
```

US-011

Here we will study the code smells related to the US 011 – Homeless Pet Management. This user story had to simple scenarios of creating a pet with good and wrong values. But, we wanted to go a bit further than that and try to fix the code smells related to not only, creating the homeless pet, but dealing with its interventions, visits, rehabs, etc. So, for this reason, in our project, the main files were most of the code smells (regarding this US) were centered in the controllers and some of the services.

Hereunder, there will be show the code smells related to the HomelessPetController.java file in our project just because the other homeless pet controllers have the same code smells and seems unnecessary to add them too (they are fixed in the same way though).

- **Removing unused imports.**
- **Removing unused fields.**
- **Duplication of literals instead of defining a constant for them.**
- **Replacing things like ArrayList<String> for just ArrayList<>.**
- **Removing useless assignment to variables.**
- **Removing the "true" from code pieces like (someBooleanValue == true).**

All of these were very easy to fix. Regarding the removing part there is not much to comment about, I just had to remove them because they weren't used in any part of this class. Concerning the duplication of literals, we defined constants like the following one:

```
private static final String EDIT_FORM = "homelessPets/editPet";
```

And this constant is used every time we want that view, instead of writing "homelessPets/editPet".

The replacement in regard to "ArrayList<String> -> ArrayList<String>" was because of something that we dealt with in the first deliverable. We recall in the unit testing, at first we had to be able to check the security of the system as well but that wasn't quite possible having these annotations in the test classes:

```
@WebMvcTest(value = HomelessPetController.class,
        includeFilters = @ComponentScan.Filter(value = PetTypeFormatter.class,
                                                type = FilterType.ASSIGNABLE_TYPE),
        excludeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE,
                                                classes =
WebSecurityConfigurer.class),
        excludeAutoConfiguration= SecurityConfiguration.class)
```

These annotations specify that the security configuration of our system shouldn't be taken into account while performing the unit testing. Therefore, we couldn't really check that it was functioning properly. We managed to think of a way in which we could check it and it was by adding some lines in each method of the controller. Let's see the listHomelessPets() method as an example of this:

```
@GetMapping()
public String listHomelessPets (ModelMap model) {

    String view;
    Boolean hasAuthorities;

    Collection<SimpleGrantedAuthority> authorities = new ArrayList<SimpleGrantedAuthority>();
    SimpleGrantedAuthority authorityVeterinarian = new SimpleGrantedAuthority("veterinarian");
    SimpleGrantedAuthority authorityTrainer = new SimpleGrantedAuthority("trainer");
    authorities.add(authorityVeterinarian);
    authorities.add(authorityTrainer);

    hasAuthorities = userHasAuthorities(authorities);

    if(hasAuthorities == true) {
        view = "homelessPets/listPets";
        List<Pet> homelessPets = new ArrayList<Pet>();
        homelessPets = this.petService.findHomelessPets();
        model.put("homelessPets", homelessPets);
    } else {
        model.addAttribute("message", "You are not authorised.");
        view = "redirect:/oups";
    }

    return view;
}
```

We can see that the first lines of it are there to check whether the user trying to access that has the permissions to do it. There's an additional method used here which is userHasAuthorities() which you can check in the next screenshot:

```
//This method will let us check security
public boolean userHasAuthorities(Collection<SimpleGrantedAuthority> authorities) {
    Boolean res = false;
    Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    if(principal instanceof UserDetails) {
        Collection<? extends GrantedAuthority> principalAuthorities = ((UserDetails)principal).getAuthorities();
        if(authorities.containsAll(principalAuthorities)) {
            res = true;
        }
    }
    return res;
}
```

This method basically gets the security context and the user's authorities and sees if she/he has permissions to access the URL. If it does, it returns true, otherwise it returns false. It hasn't anything to do with the code smell but we had to explain where were those lines coming from.

The refactoring of that method in particular ended up looking like this:

```
@GetMapping()
    public String listHomelessPets (ModelMap model) {

        String view;
        List<String> authorities = new ArrayList<>();
        Boolean hasAuthorities;

        authorities.add(VETERINARIAN);
        authorities.add(TRAINER);

        hasAuthorities = userHasAuthorities(makeAuthorities(authorities));

        if(Boolean.TRUE.equals(hasAuthorities)) {
            view = "homelessPets/listPets";
            List<Pet> homelessPets = this.petService.findHomelessPets();
            model.put("homelessPets", homelessPets);
        } else {
```

```
                model.addAttribute(MESSAGE, "You are not authorised.");
                view = REDIRECT_URL;
            }

            return view;
        }
```

And since this had to be done in each method we extracted some of the code in a new method which was makeAuthorities():

```
public Collection<SimpleGrantedAuthority> makeAuthorities(List<String> authoritiesString) {
            Collection<SimpleGrantedAuthority> authorities = new ArrayList<>();
            for (String s: authoritiesString) {
                SimpleGrantedAuthority authority = new SimpleGrantedAuthority(s);
                authorities.add(authority);
            }
            return authorities;
        }
```

Now in each one of those controller methods the only thing we have to pass is a list of strings which will be transformed in a collection of SimpleGrantedAuthority objects which will be processed by userHasAuthorities().

US-015

Although this refactoring started with the debugging of the classes related to US-015 (Vet plans a new intervention) due to the amount of bad smells we faced at the start of the process, this refactoring not only has covered the classes related to US-015 but also to the tests that shown some bad smells that could be fixe and the classes that were already defined in the template of the project (Visit/Pet/Owner… Controller/Service/Repository…).

The bad smells encountered were:

- **Define constant instead of duplicating literal**

  We defined a private final static String that will substitute the literals

```
@Controller
public class InterventionController {

    private final PetService    petService;

    private static final String CREATE_FORM     = "pets/createOrUpdateInterventionForm";
    private static final String INTERVENTION    = "intervention";
```

  This bad smell although easy to avoid and fix it was very common in the code.

- **This block of commented out lines should be removed**

  Commented lines in the code were abundant but very easy to solve. The solution simply consisted on deleting those lines.

```
// This is a comment
System.out.println("Hello World");
```

- **Remove this unused import**

  This bad smell is given when we have imports that are not used in the specified class.

  They can be easily fixed removing those imports. In eclipse, an efficient way to do this is using the Ctrl + Shift + O key combination to reorganize them.

- **Remove this useless local variable**

  Local variables that are not used simply make the code more difficult to understand for a new developer and the solution for them are deleting it.

- **Remove this unused private field**

  Like unused local variables, this private fields have no purpose to be in the code, thus the optimal solution is to delete them.

- **Remove this “public” modifier**

  This can be fixed removing the visibility of all the testing methods and classes and let them to be by default:

```
@Test
void getOrganizationsListTest() {
    given()
        .auth()
            .oauth2(token)
    .when()
        .get("https://api.petfinder.com/v2/organizations")
    .then()
        .statusCode(200)
    .and()
        .assertThat()
            .body("organizations.size()", equalTo(20))
            .body("pagination.count_per_page", equalTo(20))
            .body("pagination.current_page", equalTo(1))
    .and()
        .time(lessThan(20L), TimeUnit.SECONDS);
}
```

  This bad smell arise in Junit tests also and were fixed.

- **Add at least one assertion to this use case**

  This bad smell only appeared in Junit test, but they were not fixed. The reason why is because the methods SonarCloud attributed this bad smell, although the had not “assertions”, they had methods related to assert data already implemented, so the method had an assertion function.

```
103     private boolean isElementPresent(final By by) {
104         try {
105             this.driver.findElement(by);
106             return true;
107         } catch (NoSuchElementException e) {
108             return false;
109         }
110     }
```

- **Remove the declaration of thrown exception “DataAccesException” that is a runtime exception**

  This smell is given when a method throws an exception that can cause a runtime exception and lead in an unexpected system behavior.

  It can be fixed by removing the explicit thrown of an exception, like in this method:

```java
@Transactional(readOnly = true)
public Owner findOwnerById(final int id) throws DataAccessException {
    return this.ownerRepository.findById(id);
}
```

Which change will result in:

```java
@Transactional(readOnly = true)
public Owner findOwnerById(final int id) {
    return this.ownerRepository.findById(id);
}
```

The smell was found mostly in repositories.

- **Complete the task associated with this TODO comment**

As the bad smell related to commented lines, this one was solve the same way, by removing the TODO comment after we were sure he had done whatever the TODO comment was asking.

- **Immediately return this expression instead of assigning it to the temporary variable**

Instead of defining an Object and then sending it to the result value, we simply return immediately the value as shown in the picture.

```java
@Transactional(readOnly = true)
public Owner findOwnerById(final int id) {
    return this.ownerRepository.findById(id);
}
```

US-022

During refactorization we focused on the rehab entity, since one of my user stories is rehab management.

Therefore we checked the code smells regarding rehab:

After taking a look at the code smells it is possible to group them. The code smell types that has to be solved:

- **Remove the unused import**
- **Remove the literal “True” boolean value**
- **Define a constant instead of duplicating literal**
- **Remove the declaration of thrown exception**

As for the first step, we removed the unused imports and declaration of thrown exceptions. It consisted of just manually going through files and deleting unused imports and declaration of thrown exceptions, where necessary.

Now we started solving bug smells regarding RehabController, because there were more serious ones, or bug smells that require something more than just a removal or deletion of

```
String view;
List<String> authorities = new ArrayList<>();
Boolean hasAuthorities;


authorities.add(TRAINER);

hasAuthorities = userHasAuthorities(makeAuthorities(authorities));
```

unused code.

One of the code smells that was repeating was a duplication of literals. To solve this code smell we assigned variables to those literals, so therefore we solved the duplication form.

As for the next 2 bug smells some changes inside the code as well were required:

In order to solve the second bug smell. We had to change declaration of hasAuthorities as below, so that in the if part (which is located in every Post and GetMapping) the check if user has right authorities is not

hasAuthorities == true, how it was before and that was causing bug smells, but it has been changed to primitive boolean expression.

After fixing bug smells in the RehabController, we checked what is the situation with the tests and bug smells. It turned out that almost all bug smells were the same. It applied also to UI tests, E2E tests and controller tests.

After revision of the bug smell, I checked the RehabController again to see the situation now. Now we can see that the biggest apart of the bug smells have been removed. Bug smells from RehabService and RehabRepository have been all removed, therefore those files do not even appear in the list. Bug smells have been also removed from the tests.

Results

This screenshot shown below is the last analysis performed to the master branch after every refactoring was done, testing and finished:

Figure 16 Final analysis

Here we have the first analysis just to be easier to compare those two:

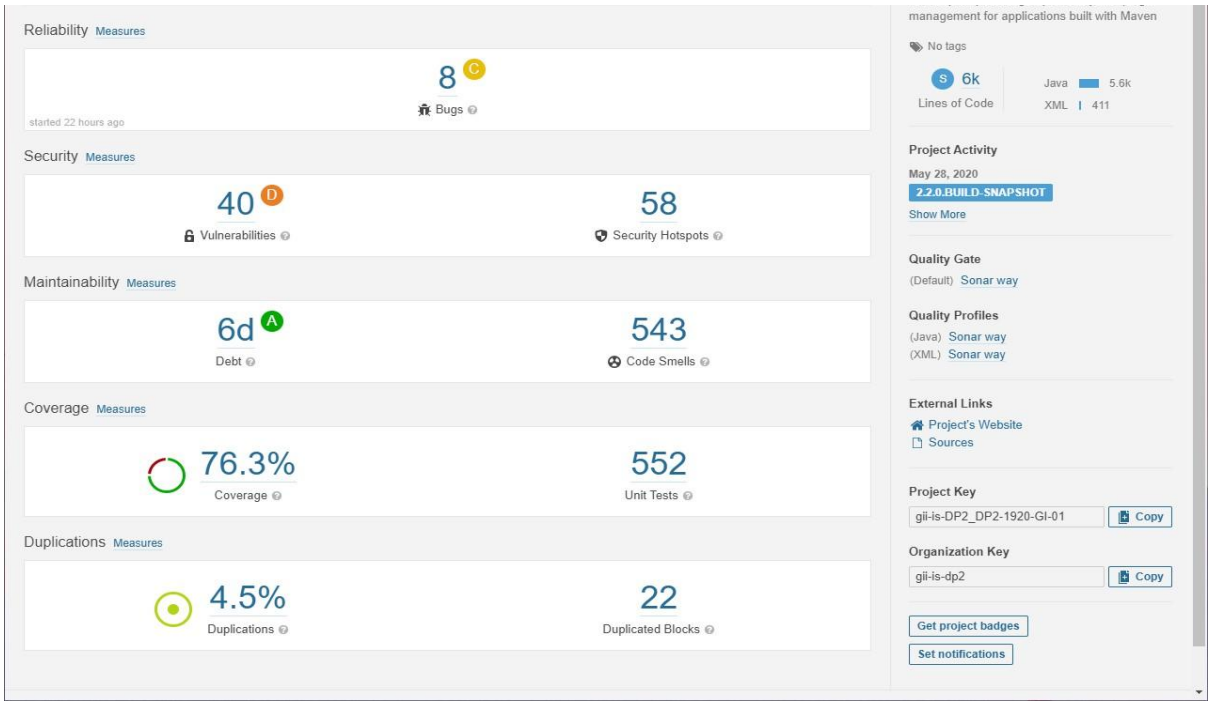

Figure 17 Comparison

The summary is the next:

- Code smells: From 543 to 198: **63.53%** decreased
- Security Hotspots: From 58 to 28: **51.72%** decreased

The result are outstanding keeping in mind that the refactoring was finished in an interval of one week (Since the theory was given until the refactoring was finish). As a group we managed to higher the quality of our code and reduce the smells, keeping the code clean, understandable and unprono to bugs or failures.

Here is the activity of our project during the week were the refactoring took place:

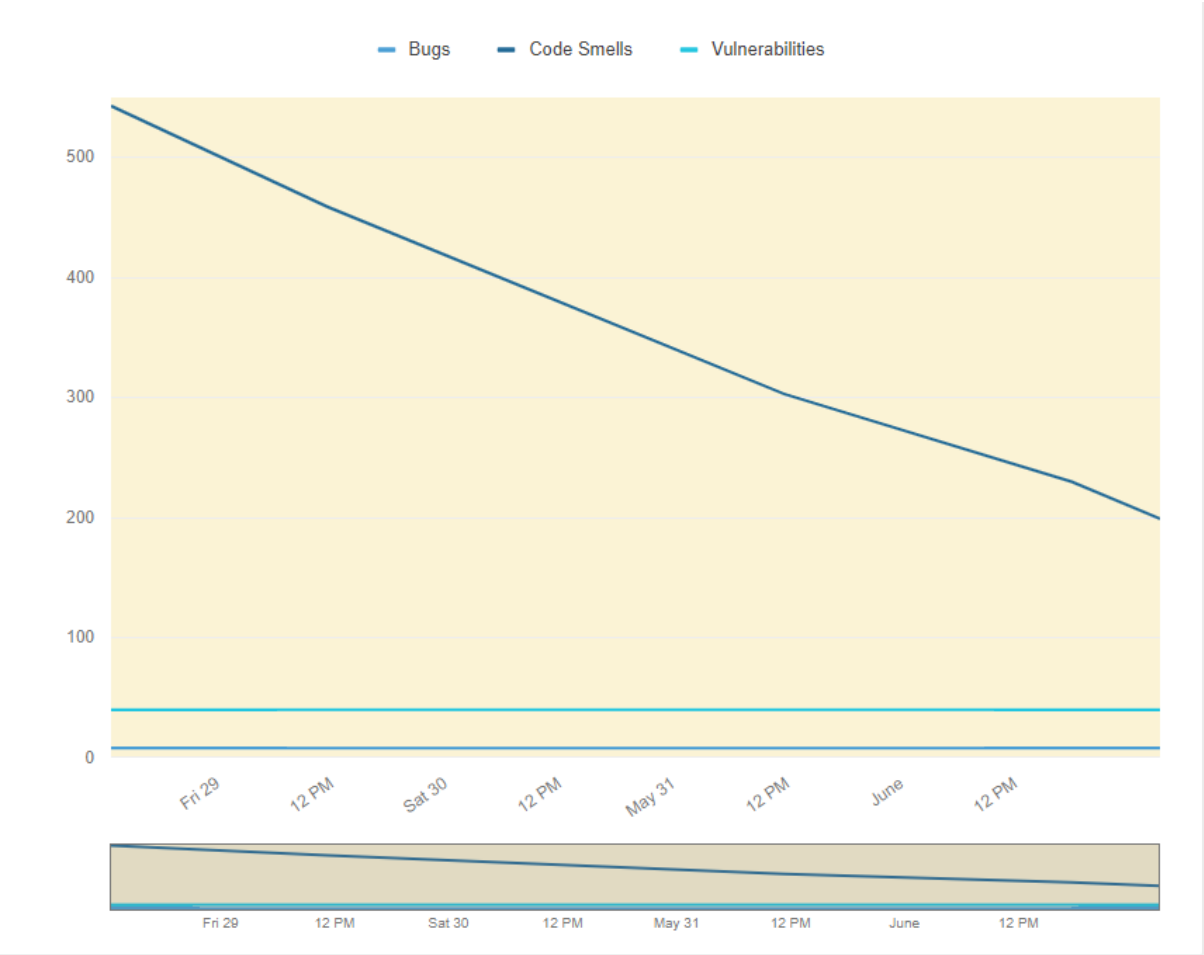

Figure 18 Activity graph

As a conclusion, we as a group are contented with our refactorization of the petclinic-application code. If there were to be more sprints in a hypothetical professional environment, this situation should not be understood as the final solution.

The correct practice would be performing a refactoring for each sprint, keeping the code clean and ensuring the quality of the deliverables/releases of the product.

With the lessons learned, we have understood the meaning of the refactoring and the problems and difficulties that can arise if a frequently revision of the code and, the use of good practices, are avoided.