École Polytechnique de Montréal

POLYTECHNIQUE MONTRÉAL

LOG8371E: Software Quality Engineering

First travaux pratique (TP1)

Professor: Rhouma Naceur

Team members

Student ID	Names
2313663	Xi-Zhen Wang
2246556	Abid Ullah Khan
2316903	Shitian Li
2313674	Fábio Akira Yonamine
2317239	Roberto Molina

Montréal, Quebec
2023

Table of Contents

I. Abstract **2**
II. Introduction **2**
III. Quality Plan **3**
A. Quality Goals 3
B. Quality Assurance Strategies 3
a. Functional suitability 3
1. Functional appropriateness 3
2. Functional correctness 4
b. Maintainability 4
1. Modifiability 4
2. Testability 4
IV. Verification **5**
A. Verification Methods 5
B. Description for Tests 6
a. Functional appropriateness 6
b. Functional correctness 6
c. Modifiability 6
d. Testability 7
C. Verification Results 7
a. Functional suitability 7
1. Functional appropriateness 7
2. Functional correctness 8
b. Maintainability 9
1. Modifiability 9
2. Testability 9
V. Improvement **12**
Functional suitability 12
1. Functional appropriateness 12
2. Functional correctness 13
VI. Comparison **13**
Functional suitability 13
1. Functional appropriateness 13
2. Functional correctness 14
VII. Conclusion **15**
Reference **16**

I. Abstract

This document presents the evaluation results of the PetClinic system for the Software Quality Engineering course, with a focus on analyzing the PetClinic application. PetClinic is a web application that serves as a practical case study for web development, emphasizing REST API functionality. The analysis centers on functional suitability (appropriateness and correctness) and maintainability (modifiability and testability). Quality goals and assurance strategies are outlined, emphasizing rigorous testing and code evaluation. The document underscores the importance of software quality assessment in adhering to specific requirements. The complexity of the PetClinic application necessitates a robust testability strategy, involving unit tests, error detection, code coverage, and complexity analysis. Overall, this document lays the foundation for a comprehensive examination of PetClinic's software quality.

II. Introduction

This document is the first travaux pratique (TP1) for the Software Quality Engineering subject. This semester, students were invited to analyze an application called PetClinic, which is a basic application suited to perform studies in web application development with different frameworks and languages, as well as using central concepts of object-oriented programming in a more practical manner.

One main point to notice is that the application doesn’t include any graphical user interface (GUI), so its focus is mainly on the REST API that functions when running it in a container, and also study the code itself. Additionally, only basic functionalities were indeed implemented, so most of the ones suited for usage in pet clinics were not included, such as billing for clients and suppliers.

Analyzing a software’s quality is paramount because it attests to one application being conformant with different specified requirements, that might include a comprehensive list of items, such as conformity to governmental laws applicable to the business using it, financial constraints as to which technologies will be used to deploy the application in order to be under the project budget, or even data protection needs, depending if there are sensitive information being stored or treated, for instance, among other important factors.

Given the simplicity of the application in study, the REST API feature will be the one under scrutiny in this first TP. As for the two quality characteristics required, our group will be analyzing the functional suitability and maintainability. For the former, the chosen sub-characteristics are functional appropriateness and functional correctness; for the latter, modifiability and testability were the chosen ones.

III. Quality Plan

A. Quality Goals

The following table summarizes the quality characteristics, sub-characteristics, quality measures, and specific objectives.

Quality Characteristics	Sub-Characteristics	Quality Measure	Goals
Functional Suitability	**Functional Appropriateness**	The degree to which the set of functions required by users are implemented and produce the correct output	Find the percentage of usage functions that each feature has implemented and calculate the total functional appropriateness of the system.
	Functional Correctness	The degree to which a product or system provides the correct results with the needed degree of precision	Determines if the program has errors and tests the coverage of the test function.
Functional Maintainability	**Modifiability**	The degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality	Calculate the average code complexity and make sure the results don't indicate a high risk program.
	Testability	The degree of effectiveness and efficiency with which test criteria can be established for a system, product, or component and tests can be performed to determine whether those criteria have been met	1. The code coverage must be equal to or greater than 90%. 2. The cyclomatic complexity of functions must be below 10. 3. We aim for at least 90% unit test coverage for each individual class or unit

B. Quality Assurance Strategies

a. Functional suitability

1. Functional appropriateness

To study the functional appropriateness of the system, first we are going to analyze the whole system and we are gonna determine what functional functions the user requires in order to accomplish all their needs.

Then we are going to observe each feature, to see if they have implemented and declared all the usage functions. With that data, we would extract the percentage of functional appropriateness each feature has, and calculate the functional appropriateness of the whole system (average of all the features).

2. Functional correctness

Functional correctness in software testing is the degree to which a software application or system performs its intended functions correctly and accurately according to its specifications and requirements. It is a critical aspect of software quality assurance and testing, ensuring that the software operates as intended and meets its functional requirements.

b. Maintainability

1. Modifiability

To determine the modifiability of the system we decided to use static analysis to obtain the specific metric value. One of the insights we focus on is cyclomatic complexity which evaluates the complexity of a program's control flow. Another method to improve modifiability is code review and reviewers can check whether the code follows SOLID principles.

We can set the scope by assessing the code complexity and potential vulnerabilities. It can be evaluated by developers and reviewers during the development stage.

2. Testability

Testability refers to the extent of how easy or challenging it is to test a system or software artifact. It is often measured by how many tests can be executed on a given system. Software testing is necessary to determine if the product meets the customer’s needs and expectations. Testability can also be described as the ease with which testing criteria for a system, product, or component can be defined and tests done to assess whether those criteria have been met.

Objective:

PetClinic is a web application that defines a wide range of functionalities with various data types and outcomes. As a result, Petclinic requires a wide range of tests to cover as many of these aspects as possible.

Measures and Methods to Validate the Objectives for Testability:

Testability metrics are quantitative measures used to assess how easy or difficult it is to test a software application. While there are various testability metrics, the number of unit tests for each method is one metric that can provide insights into the testability of the codebase. Here's how the number of unit tests for each method can be a criterion for the testability of a PetClinic application:

Metrics Used	Achievement Criteria	Tool Used
Number of Unit Tests	At least one test must be provided for each implemented method.	JUnit

Number of Fatal Errors	The software must not contain any fatal errors that block the proper functioning of the PetClinic application	SonarQube
Code Coverage	This indicates the percentage of code that is executed during testing. We are planning to reach code coverage to 90%	Jacoco
Cyclomatic Complexity	We assess the complexity of the code by counting the number of linearly independent paths through it. Lower complexity can make the code easier to test.	SonarQube

▲ Table: Testability Validation Metrics

Quality plan: 24/30

IV. Verification

A. Verification Methods

Characteristics	Sub-characteristics	Objectives	Tool Used
Functional suitability	**Functional appropriateness**	The system has implemented 95% of the functions required by the users and produces the correct output.	Manual testing and SwaggerUI
	Functional correctness	The feature provides expected results 90% of the time.	JUnit for unit testing SonarQube for line coverage
Maintainability	**Modifiability**	The system can accommodate 100% of changes without causing errors and the cyclomatic complexity score needs to be between 1 and 20.	SonarQube for analyzing Cyclomatic Complexity
	Testability	95% of unit tests must be run successfully to achieve an error rate of less than 5%.	JUnit
		The unit tests must cover 90% of the code, i.e., the code coverage must be equal to or above 90%.	Jacoco

B. Description for Tests

a. Functional appropriateness

To find out the percentage of usage functions that have been implemented, we are gonna follow the next methodology:

Firstly we need to determine what usage functions the users need to make an appropriate use of the system. We have arrived to the conclusion that at least each feature such allow the user to:

1. Create a new object
2. See the object
3. Edit an object
4. See all the objects that exist
5. Delete a concrete object

So for each feature we will obtain the degree of appropriateness by observing the number of these principal functions that have been implemented. We are gonna do it manually with the help of swaggerUI.

After getting the degree of appropriateness of each feature we are gone calculate the degree of appropriation of the whole system (which is the average of all the features) that in order to achieve our objective it needs to be greater than 95%.

b. Functional correctness

We prefer to use **JUnit** as the testing tool so that we can observe the test results and easily identify any parts that may have gone wrong. For programs that pass the test, Junit will automatically green-label the class, reducing the workload of engineers.

For the coverage problem of test function, we chose **Sonarque** as our detection tool. It will give us a specific value and percentage of missing coverage, as well as specific methods that do not have test function.

c. Modifiability

One of the maintainability sub-characteristics we selected is modifiability. We decided to use SonarQube to analyze the code complexity of the system and ensure that test coverage adequately covers the 90% of the code.

Cyclomatic complexity is one of the measurements generated from the SonarQube report, which is a software metric used to indicate the complexity of a program. We obtain the score by calculating the number of linearly independent cycles that exist in a control-flow graph. In general, the recommended score for average cyclomatic complexity is between 1 and 10. Lower complexity scores represent that the code is easier to modify and more straightforward.

d. Testability

For the testability of PetClinic, we use a number of tools and procedures. The following **TEST PLAN** summarizes the tools used and the quality goals that are set to be met.

Metrics Used	**Achievement Criteria**	**Tool Used**
Number of Unit Tests	95% tests must run successfully	JUnit
Number of Fatal Errors	The PetClinic software must not contain two fatal errors that prevent it from running.	SonarQube
Code Coverage	This indicates the percentage of code that is executed during testing. We are planning to reach code coverage to 90%	Jacoco

Cyclomatic Complexity	The cyclomatic complexity must not be greater than 20.	SonarQube

▲ Table: Test Plan

C. Verification Results

a. Functional suitability

1. Functional appropriateness

After observing the appropriateness of each feature and with the help of SwaggerUI we obtained the next table:

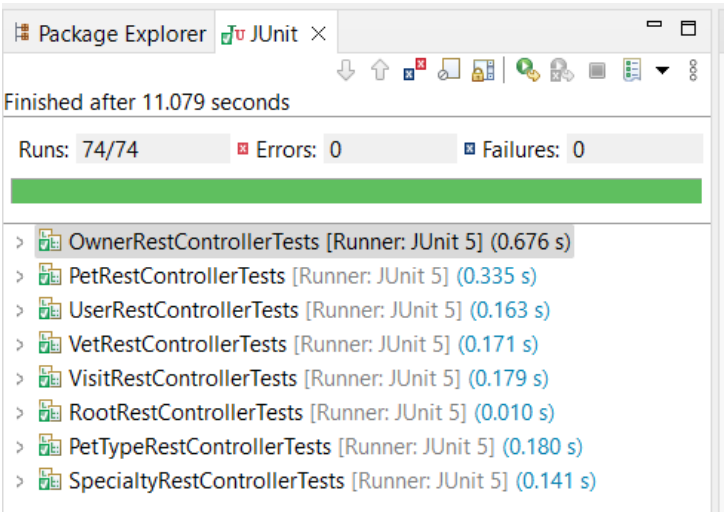
Features	Number of usage functions implemented (1-5)	Percentage (%)
pettypes	5	100
users	1	20
vets	5	100
owners	5	100
pets	5	100
specialties	5	100
visits	5	100
Total		88.57

▲ Table: functional appropriateness of each feature

After observing all the features we have concluded that the total appropriateness of the system is 88.57%.

Our objective was that 95% of the usage functions were implemented, so we can confirm that the system has a insufficient degree of functional appropriateness. Another conclusion we can extract is that the feature that needs to improve significantly is users that only have one of the five essential functions implemented.

2. Functional correctness



The figure is the result of the JUnit test, the green color indicates the pass.

Obviously, all the classes passed the test, but that does not mean we can say that our code is correct, because we do not know if all the functions to be tested have their corresponding test functions. So in the next step, we'll use SonarQube to test whether all of our functions have their corresponding test functions.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
rest	730	0	0	5	11	88.0%	0.0%
advice	51	0	0	0	1	86.4%	0.0%
controller	678	0	0	5	10	88.1%	0.0%
package-info.java	1	0	0	0	0	—	0.0%

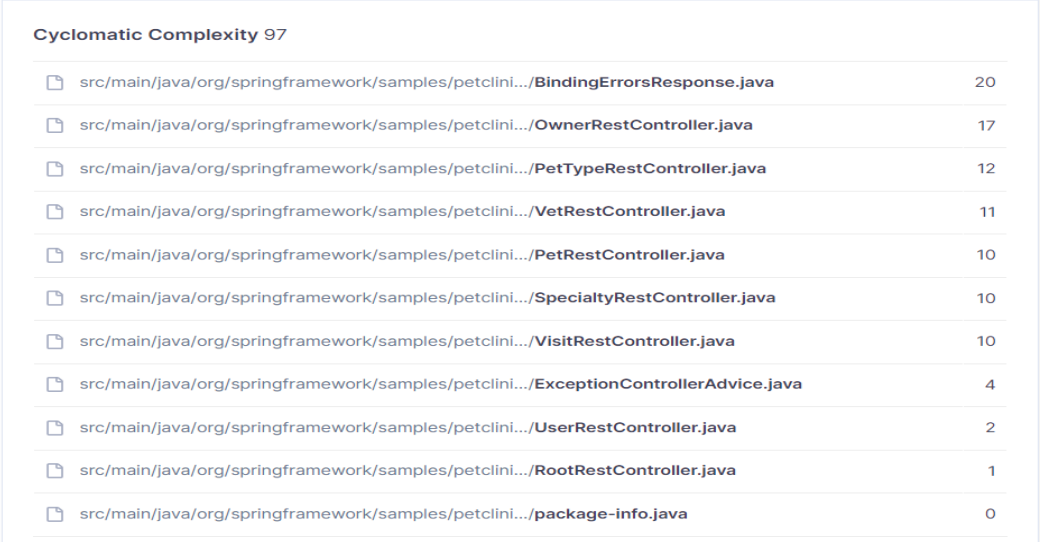
	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
controller	678	0	0	5	10	88.1%	0.0%
BindingErrorsResponse.java	93	0	0	2	1	69.0%	0.0%
OwnerRestController.java	143	0	0	1	1	93.8%	0.0%
PetRestController.java	72	0	0	0	1	94.3%	0.0%
PetTypeRestController.java	80	0	0	0	1	92.9%	0.0%
RootRestController.java	18	0	0	0	2	33.3%	0.0%
SpecialtyRestController.java	76	0	0	0	1	94.6%	0.0%
UserRestController.java	34	0	0	2	1	100%	0.0%
VetRestController.java	85	0	0	0	1	88.9%	0.0%
VisitRestController.java	77	0	0	0	1	94.7%	0.0%

From the figure above, we can see that there are 26 lines of code that are not covered, which means that there is no test function for these codes to test them. However, the coverage is 88.0%, which is lower than the 90% threshold we set for function correctness. Hence, we will make the improvements in the next section.

b. Maintainability

1. Modifiability

Through the report generated by SonarQube, we obtained the following results:



We found that the cyclomatic complexity score in each class file located in the "rest" folder is between 1 to 20.

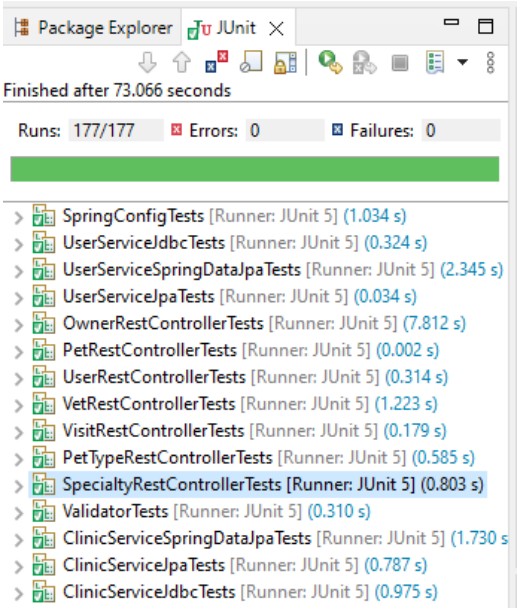
According to the classification in "Software Quality Indicators for Identifying Risks", risk can be divided into four levels: a score of 1 to 10 means that the program is a little risky; a score of 11 to 20 means moderate risk; a score of 21 to 50 represents high risk; and a score over 50 indicates very high risk.

We can see that all files have cyclomatic complexity less than or equal to 20, which means the program is at or below medium risk.

2. Testability

Test Results

In this section, we are presenting our results generated by launching the specified tools mentioned above in the TEST PLAN table. First, we launched all the unit tests in the Spring Tool Suit IDE to see if there was any possible test failure. The below figure shows the results of all the unit tests that ran successfully.

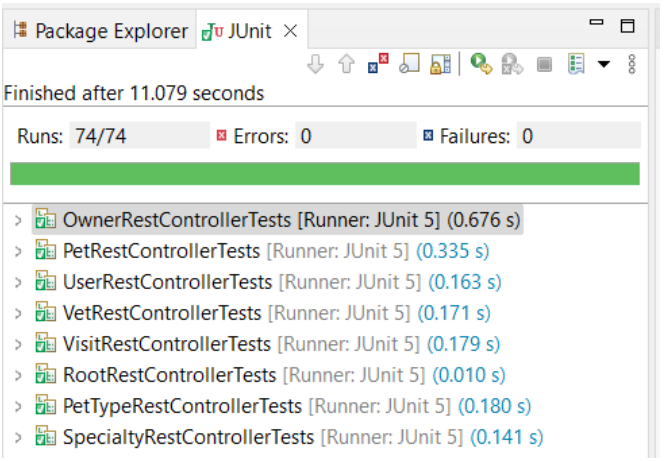


We also analyzed the results using SonarQube. Below are the results of the SonarQube.

Tests	
Unit Tests	177
Errors	0
Failures	0
Skipped	0
Success	100%
Duration	17s

Test Cases for the Controller:

Now we also run test cases for the controller. We have 73 original test cases for the controller. All the test cases ran successfully. We improved our code and wrote some test cases for uncovered methods. Therefore, our result is 74/74 which means all test cases run successfully with a 0% error rate.



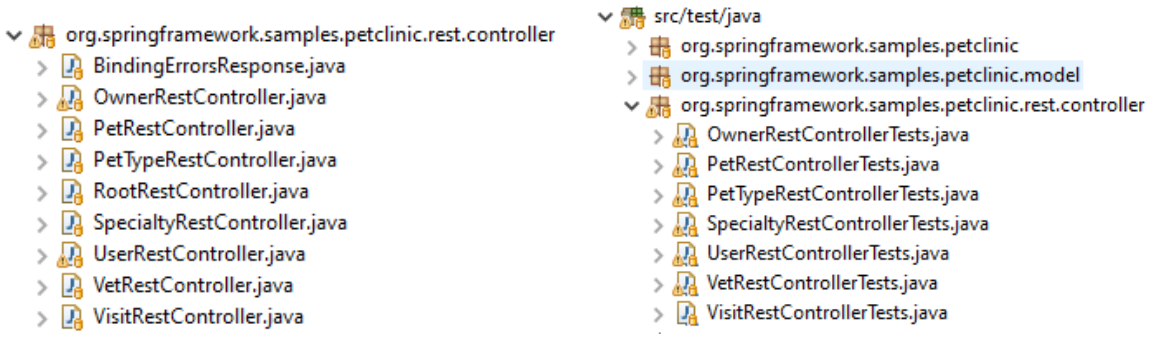
Successful Test Run for the Advice:

Now we run Junit tests for the advice folder. All the test cases run successfully, which means that their testability goals and objectives are met. We use SonarQube to see the Advice code coverage, which is 86.4%

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
advice	51	0	0	0	1	86.4%	0.0%
ExceptionControllerAdvice.java	51	0	0	0	1	86.4%	0.0%

Code Coverage:

In this section, we are looking deeper into the code coverage, i.e., how much the tests cover the code (methods). The ***rest controller*** contains **nine classes** with a number of methods or functions. There are only seven test classes to test the controller. We want to check how much these tests cover the controller classes.



We launch Jacoco and SonarQube to see code coverability for the controller and advice results shown below:

org.springframework.samples.petclinic.rest.controller

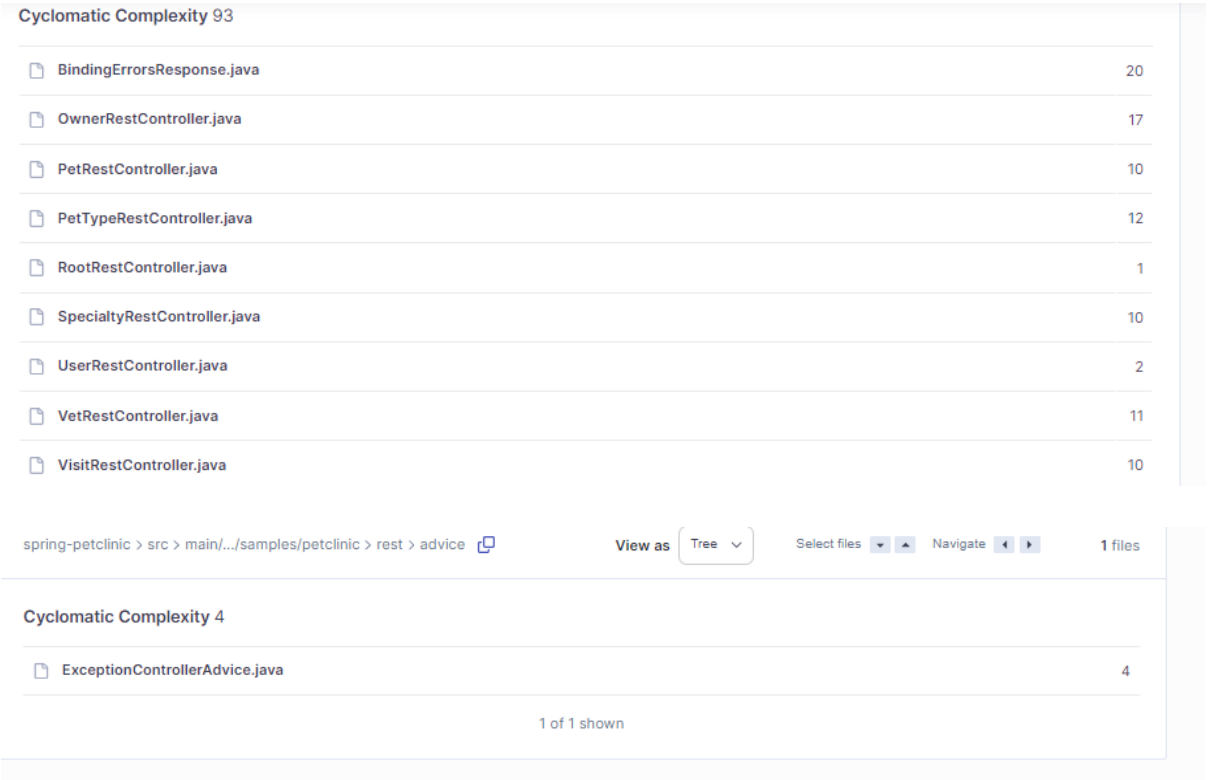
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BindingErrorsResponse		63%		43%	9	16	12	40	2	8	0	1
VetRestController		92%		80%	2	11	3	35	0	6	0	1
OwnerRestController		96%		81%	3	17	2	65	0	9	0	1
BindingErrorsResponse.BindingError		75%		n/a	1	6	1	15	1	6	0	1
RootRestController		33%		n/a	1	2	2	3	1	2	0	1
PetTypeRestController		96%		83%	2	12	1	30	0	6	0	1
VisitRestController		96%		87%	1	10	1	30	0	6	0	1
SpecialtyRestController		96%		87%	1	10	1	29	0	6	0	1
PetRestController		96%		87%	1	10	1	27	0	6	0	1
UserRestController		100%		n/a	0	2	0	8	0	2	0	1
Total	112 of 1,303	91%	19 of 78	75%	21	96	24	282	4	57	0	10

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
rest	730	0	0	5	11	88.0%	0.0%
advice	51	0	0	0	1	86.4%	0.0%
controller	678	0	0	5	10	88.1%	0.0%
package-info.java	1	0	0	0	0	—	0.0%

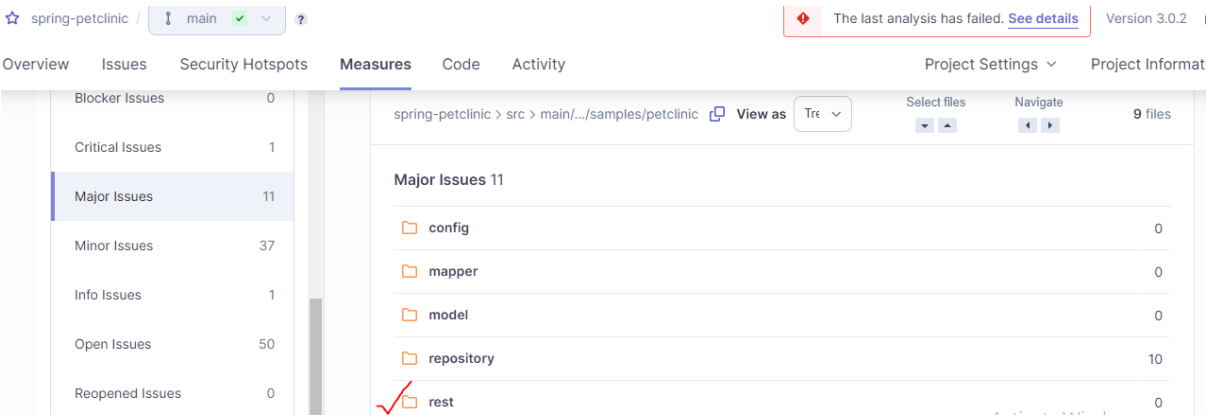
We can see in the above figure that the test cases for the controller cover 91% of the code, which successfully met our requirements/criteria.

We will now use cyclomatic complexity to demonstrate testability. Based on the defined quality goal for CC, we achieved the desired criteria, which are shown below. Our goal ranges from 1 to 20. Improvement.

We can see in the following figure that the cyclomatic complexity for the rest API is less than 20. The tool we used is SonarQube. Note that we are calculating the CC of individual class in the controller folder. Similarly, the CC for the advice class/folder is 4. From these results, we claim that PetClinic is testable.



From the analysis of the PetClinic, we identified using SonarQube that there are no fatal errors that could potentially prevent it from operating. We can see there are no major or blocker issues in the rest folder.



Therefore, based on our test plan, we conclude that all our quality goals are met and the PetClinic is declared as testable or the Testability of the system is high.

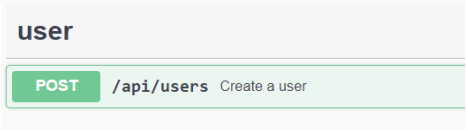
V. Improvement

Quality verification: 50/50

Functional suitability

1. Functional appropriateness

So as we observed in the previous table of functional appropriateness, the degree of appropriateness is insufficient, and the feature that needs to be improved is users. This feature only has one of the essential functions implemented:



▲ Image: SwaggerUI of users

As it can be observed in the Swagger UI user has only implemented the function of creating a new object. To improve this feature, we have implemented the other functions that were required: see a user, edit a user, see all the users and delete a user.

In order to implement each usage function, we changed the user controller, user service, and user repository. Here it can be observed an example of the functions implemented to delete a user (to see all the code visit https://github.com/Rmolina2002/TP1-Software-Quality-Engineering):

```java
@PreAuthorize("hasRole(@roles.OWNER_ADMIN)")
@RequestMapping(method = RequestMethod.DELETE, value = "/users")
public ResponseEntity<UserDto> deleteUser(String username) {
    User user = this.userService.findUserByUsername(username);
    if (user == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    this.userService.deleteUser(user);
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
```

▲ Image: User Controller

```java
@Override
public void deleteUser(User user) {
    userRepository.delete(user);
}
```

▲ Image: User service

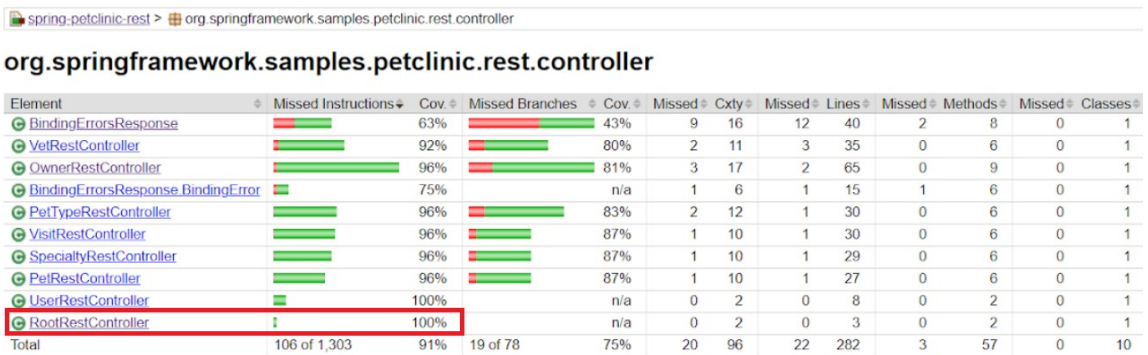
```java
@Override
public void delete(User user) throws DataAccessException {
    Map<String, Object> params = new HashMap<>();
    params.put("username", user.getUsername());
    this.namedParameterJdbcTemplate.update( sql: "DELETE FROM users WHERE username=:username", params);

}
```

▲ Image: User Repository

2. Functional correctness

Below are the improvement results. Specifically we made an improvement in the RootRestController class. Before, it was **33%**, then we wrote some test cases for this specific class, and now the code coverage for this class is **100%** which can be shown in the following screenshot.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BindingErrorsResponse		63%		43%	9	16	12	40	2	8	0	1
VetRestController		92%		80%	2	11	3	35	0	6	0	1
OwnerRestController		96%		81%	3	17	2	65	0	9	0	1
BindingErrorsResponse.BindingError		75%		n/a	1	6	1	15	1	6	0	1
PetTypeRestController		96%		83%	2	12	1	30	0	6	0	1
VisitRestController		96%		87%	1	10	1	30	0	6	0	1
SpecialtyRestController		96%		87%	1	10	1	29	0	6	0	1
PetRestController		96%		87%	1	10	1	27	0	6	0	1
UserRestController		100%		n/a	0	2	0	8	0	2	0	1
RootRestController		100%		n/a	0	2	0	3	0	2	0	1
Total	106 of 1,303	91%	19 of 78	75%	20	96	22	282	3	57	0	10

▲ The code coverage result shows RootRestController reaches 100%

```java
@RestController
@CrossOrigin(exposedHeaders = "errors, content-type")
@RequestMapping("/")
public class RootRestController {

    @Value("#{servletContext.contextPath}")
    private String servletContextPath;

    @RequestMapping(value = "/")
    public void redirectToSwagger(HttpServletResponse response) throws IOException {
        response.sendRedirect(this.servletContextPath + "/swagger-ui/index.html");
    }

    }
```

▲ Code snippet for RootRestController and green background represents coverage of the test

```java
@SpringBootTest
@ContextConfiguration(classes=ApplicationTestConfig.class)
@WebAppConfiguration
class RootRestControllerTests {

    @Autowired
    private RootRestController rootRestController;

    private MockMvc mockMvc;

    @BeforeEach
    void initRoot() {
        this.mockMvc = MockMvcBuilders.standaloneSetup(rootRestController)
            .setControllerAdvice(new ExceptionControllerAdvice())
            .build();
    }

    @Test
    @WithMockUser(roles="OWNER_ADMIN")
    void testSwaggerRedirect() throws Exception {
        this.mockMvc.perform(get("/")
            .accept(MediaType.APPLICATION_JSON_VALUE))
            .andExpect(status().is3xxRedirection());
    }
}
```

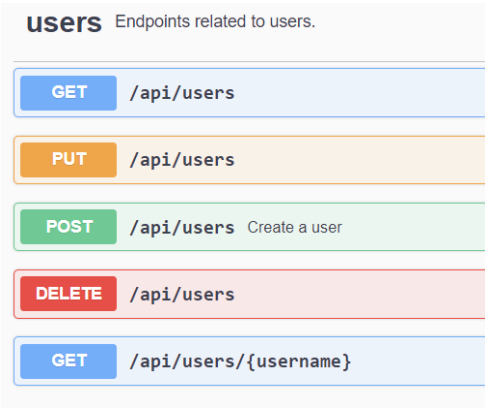
▲ Code snippet for RootRestController test case

VI. Comparison

Functional suitability

1. Functional appropriateness

After implementing the four usable function missing and making sure that each of them gives the correct output, the swaggerUI looked like this:



▲ Image: SwaggerUI of users after the improvement

So according to our description the feature users have all the usable functions implemented correctly, passing from a 20% to a 100% of functional appropriateness.

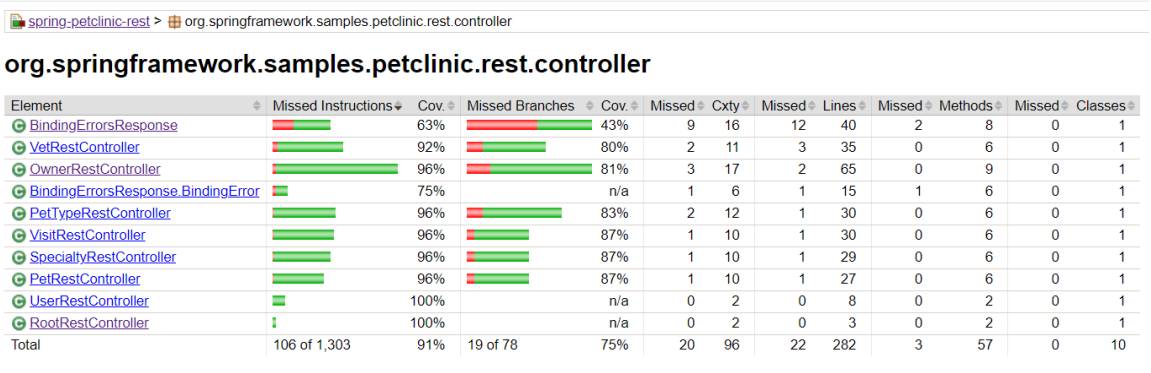
Features	Number of usage functions implemented (1-5)	Percentage (%)
pettypes	5	100
users	5	100
vets	5	100
owners	5	100
pets	5	100
specialties	5	100
visits	5	100
Total		100

As it can be observed in this table the increment of functional appropriateness of users makes that the total percentage is 100%, accomplishing our objective that required more than 95%.

2. Functional correctness

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
rest	730	0	0	5	11	88.0%	0.0%
advice	51	0	0	0	1	86.4%	0.0%
controller	678	0	0	5	10	88.1%	0.0%
package-info.java	1	0	0	0	0	—	0.0%

Before the improvement, the coverage percentage was **88%**, which was obviously lower than the target percentage we had set. After making improvements by adding more test functions, the coverage percentage increased to **91%**, surpassing our goal.



spring-petclinic-rest > org.springframework.samples.petclinic.rest.controller

org.springframework.samples.petclinic.rest.controller

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BindingErrorsResponse		63%		43%	9	16	12	40	2	8	0	1
VetRestController		92%		80%	2	11	3	35	0	6	0	1
OwnerRestController		96%		81%	3	17	2	65	0	9	0	1
BindingErrorsResponse.BindingError		75%		n/a	1	6	1	15	1	6	0	1
PetTypeRestController		96%		83%	2	12	1	30	0	6	0	1
VisitRestController		96%		87%	1	10	1	30	0	6	0	1
SpecialtyRestController		96%		87%	1	10	1	29	0	6	0	1
PetRestController		96%		87%	1	10	1	27	0	6	0	1
UserRestController		100%		n/a	0	2	0	8	0	2	0	1
RootRestController		100%		n/a	0	2	0	3	0	2	0	1
Total	106 of 1,303	91%	19 of 78	75%	20	96	22	282	3	57	0	10

Improvements: 20/20

VII. Conclusion

For the first chosen quality characteristic (functional suitability), our group has analyzed the functional appropriateness and functional correctness sub-characteristics. For the former, we defined a goal of 95%, taking into account that all features in the app were supposed to have five functions. When first analyzing the code, we realized that all features had five functions, except for one that had only one function implemented, taking our evaluation to 88.57%, below our target. To fix that, we implemented the four missing functions on one of the features, taking the evaluation to 100% and therefore meeting our goal.

For the latter, we defined a goal of 90%, taking into account the unit testing with JUnit, and also tests run with SonarQube. All classes passed the JUnit testing, but as it doesn't tell us if all tested functions actually have their corresponding test functions, we also ran SonarQube, which got us 88% of coverage, below our target, because of tests missing for some of the lines of code. To fix that, we implemented improvements by adding more test functions, taking the evaluation to 91% and therefore meeting our goal.

As for the second quality characteristic (maintainability), our group has analyzed the modifiability and testability sub-characteristics. For the former, we defined the goal of falling within the definition of medium or less risk, according to the "Software Quality Indicators for Identifying Risks". When running tests on SonarQube for cyclomatic complexity score, the application managed to fall exactly on the medium or less risk category, so no further actions were taken to improve it.

For the latter, based on the aforementioned test plan, we can also affirm that all the quality goals are met, so the app is declared as testable and no actions to further improve them are needed.

With this, we can finally conclude that all quality sub-characteristics have been successfully met, even though there were some adjustments needed.

Reference

[1] ISO/IEC 25010. Link: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[2] C. Ebert, J. Cain, G. Antoniol, S. Counsell and P. Laplante, "Cyclomatic Complexity," in IEEE Software, vol. 33, no. 6, pp. 27-29, Nov.-Dec. 2016, doi: 10.1109/MS.2016.147.

https://ieeexplore.ieee.org/abstract/document/7725232?casa_token=vdf516REVfIAAAAA:On1AK48bEv3JNa47QuS4b_FWTis3e-dAp7M4qzSwrDizrrKQp_3gzoAUxPJbdUa3QasK0D3oYQ

Quality of work: 94/100
Quality of presentation: 100/100

Grade: 18.8/20