Ilir Tagani

CS-320

12/8/2023

Project Two

Summary:

a. In Project One, I developed six distinct classes to meet the specific software requirements outlined by the client. Each class incorporated various variables, each with precise specifications as outlined in the software requirements document. To adhere to these requirements, I implemented thorough checks for each variable, ensuring that all inputs met the specified criteria.

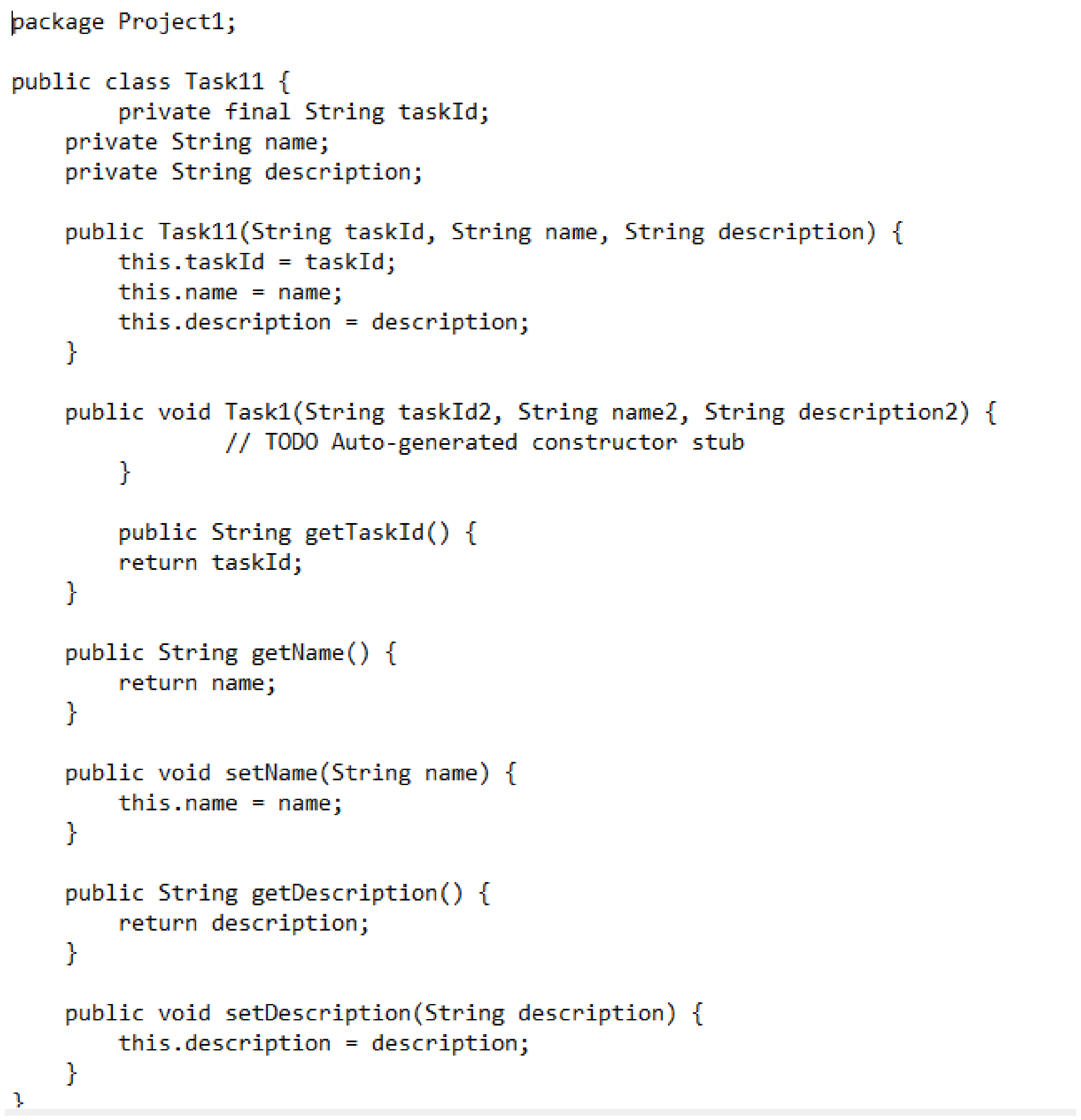
   For instance, in the Contact class, there existed a variable named "contact ID" with constraints such as a maximum length of 10 characters, non-emptiness, and non-updatability. To satisfy these requirements, I instantiated a string variable for the contact ID and implemented an if-check to verify its length and non-emptiness. Additionally, to prevent updates to the contact ID, I deliberately refrained from creating a setter method, allowing only a getter method. This deliberate design choice ensured that no method existed to alter the contact ID value.

   Another illustration pertains to the Appointment Service class, which featured a delete appointment function. This implementation was devised to meet the software requirement of facilitating appointment deletion. Through careful consideration of the requirements, I designed and implemented this functionality within the class.

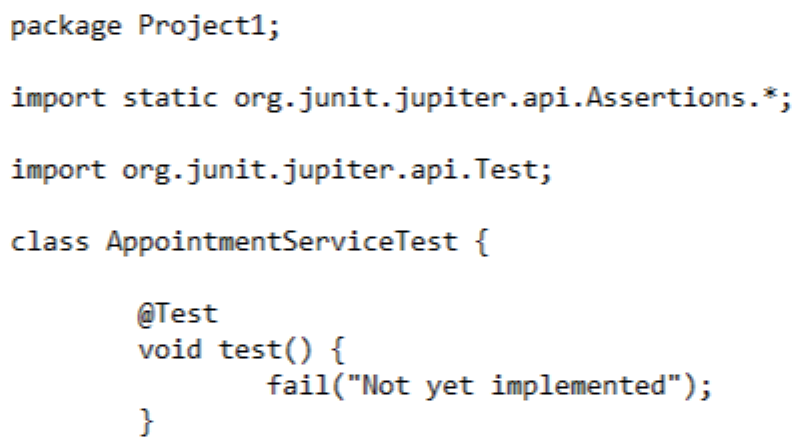
b. To validate the correct handling of inputs in my classes, I conducted comprehensive JUnit tests for the Client, Task, and Appointment classes. Two distinct types of tests were executed for each class. Firstly, the tests aimed to confirm that the class effectively processed appropriate inputs. For instance, in the Client class, a test was designed to input a valid user ID, first and last name, phone number, and address. This verification ensured the absence of unexpected errors arising from inputting data that adhered to all specified criteria. Moreover, it validated the proper functioning of getter methods for each variable.

   Secondly, I subjected each variable to tests with inputs that did not meet the specified criteria. For instance, in testing the phone number, an input exceeding 10 digits was utilized. This test was structured to anticipate the occurrence of an error, and the test would fail if no error was thrown. To enhance the quality of the tests, particular attention was given to achieving a minimum coverage percentage of 80% for all tested scenarios.

Also to ensure that my code was sufficient in meeting the requirements:



```java
package Project1;

public class Task11 {
        private final String taskId;
    private String name;
    private String description;

    public Task11(String taskId, String name, String description) {
        this.taskId = taskId;
        this.name = name;
        this.description = description;
    }

    public void Task1(String taskId2, String name2, String description2) {
                // TODO Auto-generated constructor stub
        }

        public String getTaskId() {
        return taskId;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```



```
package Project1;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class AppointmentServiceTest {

	@Test
	void test() {
		fail("Not yet implemented");
	}
```

<u>Reflection:</u>

a. I employed two primary software testing methodologies, with the first being static testing. In this type of testing, there is no need to execute the code. Instead, I systematically examined the code, scrutinizing it line by line to verify its correctness and identify any potential errors. Subsequently, I transitioned to my second testing approach, which involved JUnit testing. This entailed the creation of a distinct test file designed for JUnit testing. Within this file, I crafted tests that input specific data into the class, validating that the observed output aligned with the expected outcomes.

    i. One testing approach omitted from my project was security testing. Security testing focuses on evaluating the security aspects of the code to identify any potential errors that could lead to security vulnerabilities. An illustration of a security test involves utilizing tools such as Maven tests or dependency reports. A dependency report, for instance, reveals all known vulnerabilities within the code and provides guidance on addressing them. As an illustration, it might highlight a vulnerability where an attacker could exploit an error handling mechanism, and the recommended fix would involve upgrading to a newer version of Java that addresses this specific vulnerability.

Mindset:

As a software tester, I exercised a considerable degree of caution, recognizing that if I mishandled the tests or failed to conduct thorough and high-quality assessments, the code could potentially harbor unnoticed errors. These overlooked errors might then become vulnerabilities exploited by others, posing a risk to the client for whom the software was developed. A crucial aspect of my role was comprehending the intended functionality of the code and ensuring a clear understanding of the program's operations, especially in terms of the interconnectedness between classes and their corresponding JUnit tests.

For instance, when evaluating coverage in the JUnit tests, it was imperative to grasp that sections of the code shaded in red were not invoked in the JUnit test and therefore remained unexecuted (refer to Figure 1). In the specific example provided, the segment "this.name = name" was not called upon, as an error was triggered instead. This distinction holds significance as it contributes to the reduction of overall coverage in the JUnit test. To enhance the overall coverage, incorporating a test that exercises the "setName" function for a value that does not prompt an error would be necessary.