7-2 Project Two Submission

Natalia A. Santiago

Southern New Hampshire University

CS320: Software Test, Automation QA

Angel Cross

October 20, 2024

7-2 Project Two Submission

For the Contact feature, I conducted unit tests regarding the CRUD operation validations-addition, retrieving, updating, and deletion of contact records-by each function. In the Tasks feature, I have placed more emphasis on task creation, status updates, and deletion. Ensured edge cases were covered, such as the dates of the task deadlines being in the past. In testing for the Appointments feature, I have checked for scheduling, modification, and cancellation to handle dates and time accurately.

The testing approach has taken a close alignment to the requirements of coverage for the expected behaviors and edge cases. Example tests created to handle the boundary conditions, such as the maximum length contact names.

As proof, I present some detailed test cases to include:

*@Test*

*public void testAddContact() {*

*Contact contact = new Contact("John", "Doe", "john.doe@example.com");*

*contactService.addContact(contact);*

*assertNotNull(contactService.getContact("john.doe@example.com"));*

*}*

The above example hereby gives evidence that indeed the contact service meets the requirement of adding and retrieving contacts.

This example confirms that the contact service adheres to the requirement of adding and retrieving contacts accurately.

High percentages of code coverage can defend the quality of the JUnit tests. Tools like JaCoCo reported code coverage above 85%, hence a sign of high coverage of testing. High coverage means most of the lines of code and branches were covered; therefore, there is a low chance that undiscovered bugs exist.

Tests were written in an iterative manner with continuous refinement based on test outcomes. Best practices ensure that code is technically sound, such as meaningful assertions, no hard-coded values. Example:

*@Test*

*public void testUpdateContact() {*

*Contact contact = new Contact("John", "Doe", "john.doe@example.com");*

*contactService.addContact(contact);*

*contact.setLastName("Smith");*

*contactService.updateContact(contact);*

*assertEquals("Smith", contactService.getContact("john.doe@example.com").getLastName());*

*}*

Efficient code is assured by having a minimum of unnecessary computations besides making use of efficient data structures. Example:

*@Test*

*public void testEfficientTaskRetrieval() {*

*List<Task> tasks = taskService.getTasks();*

*assertTrue(tasks instanceof ArrayList); // Using ArrayList for efficient random access*

*}*

Firstly, I applied the black-box testing technique, whose basis of attention to input/output analysis did not involve the structure of internal code. This is helpful in providing validation for functional requirements. The second was the white-box testing technique, which called for the analysis of the internal operation of methods; this is particularly helpful in finding out logic errors, as well as path coverage.

I also employed Integration Testing, which, by definition, works in testing pieces that are combined to ensure that they work together. This was not deeply explored in the project. Lastly, I employed System Testing, which is usually a comprehensive evaluation of the whole system's functionality. This would be more applicable in later stages when the whole application is integrated.

The kind of black-box testing is most suitable for user acceptance because it consists of the work factor from the user's point of view. White-box testing also offers much useful information while the system is still in the development phase. Integration testing detects the proper functioning of interfaces between components. In general, a multi-component system goes through integration testing. System testing checks that the whole application meets expectations. This is indispensable on the eve of the release.

My approach was that caution was the better part of valor since small blind spots could make for massive bugs. The complexity of the code and the interrelationships have been understood in depth, as highlighted with thorough boundary testing. The conflicts of appointments need to be tested with a prudent design, keeping in view the conflictive time slots. Customploy peer reviews along with automated tools to reduce personal biases. Peer code reviews point out many edge cases which may have been overlooked. Quite often when one tests his code there is a confirmation bias. Instead, another team member ensures validation without any bias.

Avoiding shortcuts is important to maintain software quality in a sustainable manner. Technical debt, after all, ultimately produces problems in applications during future maintenance. My Plan to Avoid Technical Debt: besides establishing integration practices, regular refactoring of the code, good documentation, check if every new feature comes with associated unit tests, maintaining high coverage, avoiding regression.

The rigor in testing and the disciplined attitude towards software development averted not only robust code quality but laid a base for scalable and maintainable systems. This kind of commitment to quality is of utmost importance at the time of reliability and user trust in a software product.

**References**

Musings, T. (2009, March 19). *10 historical software bugs with extreme consequences*. Pingdom.

https://www.pingdom.com/blog/10-historical-software-bugs-with-extreme-consequences/

Saxena, A. (2024, February 1). *What is Defect Escape | Importance & How to Measure?*

Testsigma Blog.

https://testsigma.com/blog/defect-escape/

Tricentis Staff. (2018, December 5). *Real Life Examples of Software Development Failures*.

Tricentis.

https://www.tricentis.com/blog/real-life-examples-of-software-development-failures