Andres Trujillo

10/20/2024

Southern New Hampshire University

CS320 Software Testing, Automation, and QA

Prof. Parul Hirpara

In this project, I developed Contact, Task, and Appointment objects in Java, ensuring comprehensive unit test coverage with JUnit to align with the specified software requirements. The primary focus was to validate each feature and requirement, confirming that they met high-quality standards through varied test scenarios. For the Contact class, I implemented a testing approach based on key requirements: accepting a unique ID, validating contact information, and integrating effectively with the Contact Service class. This strategy required coverage of null test cases and the validation of input constraints, including ID length and null checks, to ensure compliance with client specifications. Helper methods were created to validate the Contact identification string, enforcing a maximum length of ten characters and disallowing null values. Additionally, I verified other fields, such as the first and last names, phone numbers, and addresses, meeting the criteria for the Contact class as stipulated for the mobile application.

public String validateContactId(String contactId) {

if (contactId == null || contactId.length() > 10) {

throw new IllegalArgumentException("Contact ID must not be null and must be less than or equal to 10 characters.");}

return contactId;

}

Similarly, the Task and Appointment objects underwent testing using a consistent methodology. Each object included validations within its class, enabling me to confirm that each was functioning as required. This defensive coding approach focused on fulfilling requirements rather than introducing bias. Consequently, errors were flagged promptly, preventing the application from accepting or storing inaccurate data in the database. For the Appointment object, I used setter methods to assign values to member variables directly, without additional helper methods. These setters-maintained data integrity by applying defensive checks during object creation, adhering strictly to requirements, and raising errors at runtime when invalid data was encountered. This robust testing strategy facilitated a clear focus on requirements while reinforcing data accuracy across all objects, allowing the application to meet the standards in both functionality and reliability. The testing came with each objective where I validated with assertions using testing methods from the Junit testing suite.

In the Contact Test class, I ensured robust testing of each requirement by focusing on boundary values and invalid data handling. For example, the ‘testLastName’ method asserts that the ‘setLastName’ function correctly throws an ‘IllegalArgumentException’ when attempting to set an excessively long last name, which aligns with the requirement for valid name lengths.

@Test

void testLastName() {

Contact settingLastName = new Contact("1", "firstn", "latn", "1234567890", "123 address correct, ok 123457");

assertThrows(IllegalArgumentException.class, () -> {

settingLastName.setLastName("toolongofaname-fail");

});

}

This test code snippet demonstrates how I directly applied boundary testing by exceeding the maximum character limit. By systematically checking each method of the Contact class—such as name, phone number, and address fields—for invalid and null values, I achieved a well-rounded test coverage that validates the class according to specifications, ensuring quality and robustness. For the Task Test class, I utilized boundary testing to verify the instantiation of a Task object. An example is the ‘testValidTask’ method, which confirms that a valid Task object is properly created with a non-null `ID`.

@Test

public void testValidTask() {

Task validTask = new Task("validId", "name", "description");

assertNotNull(validTask);

}

The preceding code illustrates how I confirmed that required fields such as `ID` were initialized correctly and remained non-null, as per the requirements. This approach provided clear validation for the presence of essential fields. Furthermore, by isolating individual test methods to address specific boundaries (such as valid and invalid IDs), I ensured complete branch coverage for each method, facilitating reliable and bias-free testing that aligns with task specifications.

In the Appointment Test class, I focused on ensuring that appointment descriptions and dates met the specifications through boundary testing and invalid data handling. For instance, the ‘testValidDescription’ method checks the creation of an Appointment object with a future date and valid description:

@Test

public void testValidDescription() {

assertDoesNotThrow(() -> {

new Appointment("123456789", new Date(System.currentTimeMillis() + TWENTY\_FOUR\_HOURS), "Good description.");

});

}

This test effectively validates that an exception is not thrown when the date and description meet all specified requirements. To comprehensively cover each class, I conducted boundary and null testing for all relevant methods, aiming for 100% branch and method coverage. By verifying each conditional branch with dedicated test functions, I removed bias and reinforced test reliability.

The overall testing approach maintained a professional focus on quality and alignment with requirements by ensuring all tests directly correlated with the functional specifications. Additionally, I implemented defensive coding principles within each class, allowing straightforward verification through test cases that involved creating objects with invalid data. This demonstrated that the class constructors and setters properly enforced requirements, enhancing the integrity and completeness of each feature’s unit testing.

During the development of this project, I relied heavily on two testing methods: unit testing and boundary value testing. These techniques were carefully chosen to align with the project’s needs, focusing on quality assurance for individual methods and validating input constraints.

For unit testing, the emphasis was on verifying those individual methods within each class—namely `Contact`, `Task`, and `Appointment`—functioned correctly on their own. Each test case was designed to isolate and examine the behavior of specific methods, such as `setLastName`, to ensure that they performed as expected for both valid and invalid inputs. This approach offered several benefits. First, it provided quick feedback on potential issues within isolated code segments, which made it easier to catch and address bugs early in the development process. Additionally, unit testing improved the quality of the code by ensuring that each component adhered to its functional requirements. This technique is particularly valuable in projects like this one, where individual functions carry specific responsibilities, and issues need to be quickly pinpointed to streamline the debugging process.

In addition to unit testing, boundary value testing was employed to validate input limits and constraints, especially for methods like ‘setName’ and ‘setDescription’. I used boundary testing to check that the system accepted names and descriptions up to their maximum allowed length, while rejecting those that exceeded these boundaries. For example, in the ‘testLastName’ method, I confirmed that the `Contact` class could manage names up to a certain length but would raise an exception for names that were too long. This testing technique is especially relevant in situations where data integrity and strict input validation are required, as it verifies that the system can handle extreme or edge-case inputs without compromising functionality. By identifying potential boundary issues early, boundary value testing helps prevent errors that could otherwise lead to system crashes or data corruption in production environments.

While unit and boundary value testing were essential to this project, I also considered the practical applications of other testing methods, such as integration testing, system testing, and acceptance testing, even though they were not directly applicable. Integration testing focuses on verifying that different components work together as expected. Given that my project’s scope was primarily on individual class functionality, the need for integration testing was minimal. This technique is typically useful for larger, more complex projects, such as web applications that involve interactions between frontend and backend components. By examining how separate modules interact, integration testing helps to reveal potential issues related to mismatches between component interfaces or data exchange.

System testing evaluates the entire software system’s functionality in a unified environment. Since my project focused on standalone classes rather than an integrated system, system testing was not necessary. However, it is worth noting that system testing is valuable in scenarios where the complete application must be verified against specified requirements. This technique is commonly applied to projects with multiple features or complex interactions, ensuring that the entire system operates cohesively and meets expected standards. For example, enterprise software or embedded systems that must function reliably under production-like conditions often undergo system testing to ensure end-to-end quality.

Finally, acceptance testing involves evaluating whether the software meets business requirements and user expectations. I did not use acceptance testing, as this project did not involve stakeholders or specific business requirements beyond the initial specifications. Acceptance testing is generally performed by clients or end-users and is essential in projects that emphasize customer satisfaction and requirement validation. This type of testing is especially useful in custom software development or consumer-facing applications, where ensuring that the final product aligns with client expectations is critical to overall project success.

Reflecting on these techniques, each testing method has unique applications depending on the project’s scope, development phase, and objectives. Unit testing plays a vital role during the development phase, particularly in projects with modular components that require frequent testing to catch bugs early. Its isolated nature enables developers to detect and correct issues before they propagate through the system, resulting in a more stable codebase. Similarly, boundary value testing is indispensable for projects with input validation and constraints, as it ensures that extreme values are handled safely and helps prevent data integrity issues. On the other hand, integration testing becomes relevant in projects with multiple interacting components, such as applications that rely on third-party APIs. This technique is well-suited to agile development environments, where different teams may develop individual components that require regular integration checks.

System testing is appropriate for projects nearing completion, where end-to-end functionality must be verified. It is commonly conducted by quality assurance teams to ensure the system’s reliability in production-like environments, making it ideal for large-scale applications. Finally, acceptance testing is essential for projects where user validation and client satisfaction are primary goals. This technique confirms that the final product meets all requirements, aligning it with client expectations. By selecting the appropriate testing methods, developers can tailor their quality assurance processes to the project’s specific needs, ultimately enhancing software quality and reliability.