Howard Gilmore

SNHU CS 320

6/19/22

Project 2

By taking into account both class and service requirements for each project, I ensured that my testing strategy was adapted to the software's specific demands. The contact class, for example, mandates that an object's contact ID should not exceed 10 characters. The protected final Strings "validID = 123456789;" and "badID = 1234567890a" indicate in code valid and invalid IDs. There are just too many characters in the second string for anybody to make use of it (11 in all). It is possible to use JUnit tests to defend the quality of the code. These tests analyze the requirements in great detail. All of our JUnit tests passed with a high percentage of coverage, indicating that the conditions were met. An additional effort is needed to assure the project's appropriate coding, according to this information.

// Verify that the contact object we created has the specified parameters

@Test

public void verifyContact() {

// Just assert that our goodContact meets all of our fields

System.out.println("Checking Contact constructor");

assertEquals(goodContact.getId(), validID);

assertEquals(goodContact.getFirstName(), validFirstName);

assertEquals(goodContact.getLastName(), validLastName);

assertEquals(goodContact.getPhoneNumber(), validPhoneNumber);

assertEquals(goodContact.getAddress(), validAddress);

As a last step, I made certain that testing detected any incorrect IDs as errors. Observe the following code and comment lines:.

// Verify that setting the ID to an invalid one raises an AssertionError

@Test

public void verifyBadId() {

System.out.println("verifying id...");

rule.expect(AssertionError.class);

new Contact(badID, validFirstName, validLastName, validPhoneNumber, validAddress);

JUnits was my go-to software testing strategy while I was designing the service milestones for Modules three, four, and five, and I depended on it heavily. In each milestone, input sections were not allowed to be empty, and ID string lengths were limited to a maximum number of allowed characters. According to the findings of the testing, the one-of-a-kind identification that is assigned to each task, contact, and appointment may be used to add or remove that information from the system. The vast majority of these JUnit tests consisted of functional testing, which was done to ensure that the product fully satisfies the criteria of each milestone. Integration testing was carried out in the same manner as data addition and deletion was so that the individual elements of the project could be properly written.

System testing has not yet been used since it needs the three milestones and integrated system to guarantee all project criteria are satisfied. Acceptance testing is another level of functional testing that ensures the finished product meets criteria and the end-user experience is as planned. External beta testers are needed for this sort of software testing. Performance testing is necessary for most big projects to guarantee the system doesn't exceed the user's computer's capabilities. Security testing is a non-functional testing approach that helps uncover gaps attackers might exploit to steal personal data. These extra testing procedures guarantee that the completed product provides the end user with the desired experience without stressing their machine and that corporate and individual information is safe from hackers. Smaller projects may not need all of the aforementioned testing methodologies, but a developer should know what testing to perform and when. Testing the system's security after just one project unit would be ineffective.

As a result, I made it a practice to go back and double-check each line of code after it was written in order to make sure the Java triggers were correctly implemented and the output was accurate. Code and test cases were thoroughly tested to ensure that every criterion was met. It was necessary for me to ensure that all of the potential inputs, outputs, and tests were clearly specified in order to see how they interacted with one another throughout programming and testing. Here are some ways to make sure the appointment object has correct fields:

@Test

public void verifyAppointment() {

System.out.println("Checking Appointment constructor");

System.out.printf("\tChecking that the id is %s\n", validID);

assertEquals(goodAppointment.getId(), validID);

System.out.println("\tChecking that the date has the correct fields");

Calendar calendar = Calendar.getInstance();

calendar.set(validYear, validMonth, validDay);

Date date = calendar.getTime();

These inputs were tested to determine whether they met the requirements by ensuring that the character length was not exceeded, there were no nulls in the inputs, and that they could be added, deleted, or updated. Testing would have been biased if the JUnit tests were constructed in a way that was consistent with the coding of the contact, task, and appointment objects. When building objects or tests, a developer should adhere to a strong writing discipline as well as prevent from producing biased tests. An precise assessment of all project requirements is crucial to avoiding mistakes, hacking, and compromised functioning.. Detailed comments on the specific functions and tests provided by each line or piece of code may assist avoid technical debt. Another member of the team may use someone else's work as a guide to evaluate what needs testing and how it should be included into the code.