Mobile Application Summary and Reflection

Lewis Quick

Department of Computer Science, Southern New Hampshire University

Software Testing, Automation, and Quality Assurance

Professor Angelo Luo M.S

October 16, 2022

A client for Grand Strand Systems requested an application to manage back-end services for a mobile application. The product backlog for the mobile application includes Contact, Task, and Appointment services. I developed code for the components of the mobile application and unit tests to ensure that the back-end services meet the client's and end-user's needs. The structure and functionality of the Contact, Task, and Appointment components are similar. Each component has an object class and a service class to manage the object and its functions. The design approach for developing the unit tests for the three components is similar and utilizes specification-based techniques.

The design approach for the unit tests heavily relies on the software requirements provided by the client. The components' characteristics make up the test conditions for the unit tests. For example, The Task object class contains a unique ID that cannot be: longer than ten characters, null, or updatable. The Task object shall have a required name that cannot be null and cannot be longer than twenty characters. The Task object shall have a required description that cannot be null and cannot be longer than fifty characters. I generated unit tests for the individual requirements of the component's object class. The unit tests for the Task component test the Task's unique ID, name, and description. The component's specifications are tested with valid, invalid, and null inputs.

Equivalence partitioning and boundary value analysis are specification-based techniques utilized to design the unit tests for the back-end services of the mobile application. The unit tests tested three groups valid, invalid, and null inputs. The test inputs are the most extreme ends of the specification boundaries. For example, a requirement for the Task name is it can not have

more than ten characters, so the valid test input is ten characters, and the invalid test input is eleven characters. The valid test input boundaries are one through ten, where one and ten are the extreme ends. The invalid test input boundaries would be eleven to infinity, where eleven and infinity are the extreme ends. The null test input would be zero. The three components' service classes utilize and depend on their respective object class. If the unit tests for the component's object class are not effective and efficient, then the unit tests for the component's service class will not be effective. The Junit tests for the back-end services of the mobile application are exceptionally effective and efficient.

I utilized the IntelliJ IDE to develop the code and unit tests for the components of the mobile application. The quality of the JUnit tests for the back-end services of the mobile application is outstanding based on the test coverage results from the IntelliJ IDE. The coverage results showed Class %: 100%, Method %: 87%, and Line %: 83%. Methods not utilized by the application impact the coverage test results for the method element, such as clearContactService(). The Contact class significantly impacts the line percentage of the coverage test because there are multiple methods with multiple lines not utilized by the application. The coverage test results for the Contact class line percentage is 71 percent.

Along with specification-based techniques, I utilize the test lifecycle, annotations, and assertions to ensure that the code is technically sound. The design approach follows the four stages of the test lifecycle: setup, exercise, verify, and teardown. The code utilizes conventional annotations and naming techniques. For example,

```
@Test
@DisplayName("Test task service, constructor")
void taskServiceClassTest() {
    TaskService actual = new TaskService();
    HashMap<String, Task> expected = new HashMap<String, Task>();
    assertEquals(expected, actual.getContactService());
}
```

This example shows the @Test annotation followed by the @DisplayName annotation describing the executed test.

The mobile application did not utilize specific and defined structure-based software testing techniques. Such as flow charts, sequential analysis, and control flow graphs. However, it did use obscure structure-based methods to design the unit tests. The coverage tests are a structure-based process for the unit tests to ensure the execution of all functions and elements in the application. The unit tests for the constructors of the three components is another component structure tested using specification-based test case design techniques. For example, an empty HashMap data structure initializes the AppointmentService constructor. The unit test for the AppointmentService asserts that the AppointmentService constructor equals an empty HashMap.

Since this was my first project developing unit tests for a system and its functions, I anticipated errors and defects. I am conscious of biases and constantly catch myself fighting with compilers about how code should work. I am becoming more conscious about my biases when developing and testing code and trying to perceive with different mindsets. To minimize and avoid bias, I should treat future projects and assignments as if it was my first assignment and anticipate errors and defects. Being flexible and accepting change is an essential skill for project development. Something I kept in mind while developing the mobile application and Junit tests is that I could be wrong, and it is more likely that I am wrong.

Another concept I adopted is to develop and test code with extreme caution with the notion that if an error or defect goes undetected, the failure can be catastrophic. It is vital not to cut corners because a small error or defect can potentially have severe consequences. For example, I read an article about how a simple coding error in Boeing's software in their CST-100 Starliner spaceship nearly turned catastrophic. The spaceship's clock was eleven hours ahead, started executing phases of the code early, and burned through twenty-five percent of its fuel early in the flight. Luckily, the attempt was a test run without passengers because the spaceship would've never made it to the space station. Hypothetically, your boss asks you to make a clock component for a program, maybe the assignment is classified, and all you know is that you are making a clock component. You think to yourself, "easy enough, a clock component is a standard component in software," so you don't use extreme caution. Next thing you know, a spaceship crashes into the International Spacestation because a defect in the clock component caused the code that lands the spaceship to execute late.

The mobile application assignment made me realize how various pressures can contribute to undetected errors and defects. The assignment also made me realize the importance of an effective and efficient test design approach. With only a week to develop the code and unit tests for the individual components of the mobile application, I had to move swiftly and with great caution. From my past experiences, I have learned that developing code for a program without an effective design can be costly, requiring more time and resources.

References

Catastrophic software errors doomed Boeing's airplanes and nearly destroyed its NASA spaceship. Experts blame the leadership's "lack of engineering culture." (2020, February 29). Business Insider. Retrieved October 21, 2022, from https://www.businessinsider.com/boeing-software-errors-jeopardized-starliner-spaceship-737-max-planes-2020-2?international=true&r=US&IR=T

Garcia, B. (2017, October 27). *Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications*. Packt Publishing.

Software Testing: An ISTQB-BCS Certified Tester Foundation guide. (2015, June 22). BCS, The Chartered Institute for IT.