Teigeler, Alexandrea

alexandrea.teigeler@snhu.edu

Abstract

A mobile application was developed for Grand Strand Systems, three services were assigned: Contact, Task, Appointment, and their correlating JUNIT Testing. The process of development and testing for these services will be discussed and evaluated.

Grand Strand Systems:  
Mobile Application report

JUNIT TESTING

**Application Requirements**

The mobile application under development had three major services that were developed separately and integrated together: ContactService, TaskService, and AppointmentService. Each service had two .java files that would meet the specified requirements and two corresponding JUNIT tests.

The Contact class needed a unique contact ID that could not be updateable nor be null. To create this specific ID, within the ContactService class, a hash function was used against the phone number of the contact to create this unique ID. This automation ensured that each ID would be unique as well as ensuring that the ID would fit within the constraints of memory and the length of the digit. The contact object attributes fell under constraints of length and type of values, disallowing the null value, which had to be tested against these constraints both within the class itself and when creating the corresponding JUNIT.

The ContactService class was used to send the contact information into the Contact class to create a new contact, which also meant that this was the class which held the hash function for the ID creation. It also needed to delete contacts and update certain fields called in this class but handled in the Contact class. A mapping list was also designed to store the information of the contact to verify if the contact was new or if it existed before the creation of the object was done, as well as testing the deletion method in the JUNIT file.

The Task class required a unique task ID, which was automated through the TaskService class in a similar manner as the ContactService. It had a task name and description attribute with constraints that limited the number of characters the container could hold as well as the value type, as such, could not be null. It tested these values before creating the object, which used the in-memory data structures that java used for storing the task objects. It used getter and setter functions for corresponding attributes to aid in testing and validating these values.

The TaskService class was used to send the values to the Task class, as well as creating the unique ID for each task. In a similar fashion, a hash function was used, parsing the character value of the description into a Long integer, and using it to create the ID. In this, the TaskService could add Tasks objects, map them on a list, delete tasks, and update the attributes as needed.

The Appointment class took values supplied by the AppointmentService class, that would create the appointment object. It validated that these values were correct before the creation of the object.

The AppointmentService class created the unique ID for the new appointment by parsing the name of the appointment as well as the date into Long integers, adding these values together, then applying a hash function to them. It communicated with the Appointment class, sending these values to create the specific attributes for the appointment object. These appointments were mapped to a list if found to be a new appointment. This class could also delete appointment objects.

Each class validated the values of the prospective object attributes, considering the constraints of what each one needed to be, and used these validation methods during the operation of the program and were utilized when designing the JUNIT tests.

**JUNIT Approach**

The JUNIT tests for each service were designed in a way to cover the most critical system functionality first. For each test file, asserting that the class object was initiated when correct values for the attributes were given, was the first use case test. If this failed, then the system application was fundamentally egregious, and the logic would need to be evaluated. This was done one of two ways, depending on the class being tested. The JUNIT Test classes ContactTest, TaskTest, and AppointmentTest were structured in very similar ways as shown in Figure A. As shown, a new task object was created with generic values that were within the scope of acceptable values. Using the assertTrue function for JUNIT testing, the object is tested using getter functions from the class to check if the attributes value corresponds with the expected value. The assertEquals is used for the ID as it is an integer and not a string value.

However, a different format for asserting that these objects had been initiated correctly was required for ContactServiceTest, TaskServiceTest, AppointmentServiceTest JUNIT tests as shown in Figure B. As shown, an empty TaskService object “service” is initiated, followed by a task object “task”. While it follows the similar format as Figure A, a call on the service object validates the task object, ensuring that the method within the class is still able to communicate as intended with the Task class.

After ensuring that the basic logic of the application functioned as intended, runtime exceptions and illegal argument exceptions were designed to make sure that the program could handle these problems appropriately. A modular approach was used, working through small iterations of the application development process, so when implementing further features and behaviors, it was not difficult to find and correct and issue once it had occurred. This approach to JUNIT testing, in an iterative manner, and testing early and often, allowed for accurate and appropriate detection of defects and holes in the program. A structured plan, testing individual components at a time against user requirements, that followed specific goals for each test and what needed to be covered in these tests to be considered a pass, determined when testing of service file was complete.

**Testing Techniques**

Designing the tests for each class required the identification of what type of tests needed to be conducted to apply full coverage as is possible when testing software applications. To identify the types of tests, the input and output (I/O) were analyzed, and the restrictions for this input was considered. What the software needed to do with these I/O, also provided insight of what should be tested. State transition testing focused on these I/O, and changes that may occur depending on different conditions, attribute values, and methods employed when transforming said object attribute data.

An understanding of where potential problems could arise, what exceptions to anticipate and design within the program, and how to test for it appropriately and thoroughly, was the initial base for the testing format. The scope determined what features each class within the program should be tested, and how other features would be required and used to help with this testing.

**Testing Considerations**

During the design and development process, precautions were taken to find and cover all faults within the system that may cause critical failure, where “a failure represents the inability of a code to perform its required function within specified performance requirements; while a fault is an incorrect step, process, or data definition in a computer program” (Favaro, F., 2013). While complete coverage and exhaustive testing is impossible in software testing, it is necessary to understand how software and its failure could lead to adverse and hazardous events. This meant to be thoughtful and considerate in the complexity of the system, and to define testing goals that would eliminate catastrophic errors in so much as it is possible to do so. Caution must be taken when considering the areas of where a system may fail, where unconsidered events may lead to such events, even when software itself complies with requirements, it can still “lead the system to a hazardous state or adverse event” (Favaro, F., 2013), which argues that if the application meets these requirements and still has faults, then there is an error in a different phase of the SDLC (Software Development Life Cycle) that created the problem.

These cautions must be taken in kind with the understanding that when testing software that has been designed and written by the same individual, there will always be some level of bias that could prevent the amount of coverage and testing needed. Limitations are imposed on a system when these biases occur, so biases in testing must be limited when possible.

**Conclusion**

The purpose of designing and implementing tests along side a program application is to ensure that the quality of the program is being monitored closely, that defects would be found in a timely manner, and that failures are minimized by keeping both testing analysts and developers do not cut corners in the SDLC process. When the quality of a system is maintained, then errors can be avoided, and serious, catastrophic events can be prevented. It is the duty of those developing software and testing it, to ensure that they account for instances to arise that might give rise to malfunctions. When precautions are taken, and the design process throughout is meticulous and thoughtful, then quality can be assured.

**Resources**

Favarò, F. M., Jackson, D. W., Saleh, J. H., & Mavris, D. N. (2013). Software contributions to aircraft adverse events: Case studies and analyses of recurrent accident patterns and failure mechanisms. *Reliability Engineering and System Safety*, *113*, 131–142. https://doi-org.ezproxy.snhu.edu/10.1016/j.ress.2012.12.018

**Appendix**

Figure (A)

**Graphical user interface, text, email

Description automatically generated**

Figure (B)

Graphical user interface, text, application, email

Description automatically generated