Final Project 2

Gregory Dionisio

Alex Pettit

Software Test Automation

October 11, 2022

In this mobile application we applied testing procedures to ensure that the program worked as expected to the requirements. To do this after the code was written we tested inputs that sometimes were true and sometimes false to ensure that the proper output or errors were being received based on the input. The requirements to the contact and task classes were that they were all strings, that they fall inside certain number of characters criteria and that they were not null. To test that these requirements were met in the program we inputted objects that fit all the criteria (pass) and ensured that the results were accepted. Then we input criteria knowingly over the length limit, with null values or a phone number not having enough digits. We expected these results to not be input and return an error to the user as was the case.

An example of this is when we input a task name that was above the 10-character maximum and asked the JUNIT test to ensure that a IllegalArgumentException was thrown as we expected from our Task.java file.

Task.Java

Text

Description automatically generated with medium confidence

TaskTest.Java

Graphical user interface, text

Description automatically generated

We also ran JUNIT tests on the service functionality of the task and contact modules. The requirements from the client in regard to the service functionality was that new records could be added with a uniqueID, objects could be deleted based on id, and objects could be updated based on id. To test these requirements were met with the code written we input valid response expecting a pass response. We also input strings that would fail such as attempting to add the same id twice, deleting an object that doesn’t exist and updating an id that doesn’t exist. An example of this is when trying to add the same record id twice such as:

TaskService.Java

Text

Description automatically generated

TaskServiceTest.Java

Text

Description automatically generated

When building the JUNIT test, we expected and attempted to find errors. Each statement of the methods was run at least once and a couple of them multiple times. There are way too many possible inputs to be able to test every possible string. We tested outlier variables and assumed that similar strings will behave the same way inside of the application to save time. The fact that all the requirements were checked is good. Although we will never get one hundred percent coverage the application has passed all test and without spending inordinate resources and time to test every input the application can be moved on to the next phase.

Through the dynamic testing done on the code we determined that the functionality was technically sound. Areas containing loops and multiple if statement can be common culprit of coding mistakes. One area in this project that needed to be checked was whether multiple ids could be inserted into the program because the requirements clearly stated that the id needs to be unique. The functionality can be seen in above example where we made sure the contact id was not in the data structure using a loop and if it was sending back a false response. This was tested by attempting to input the same id multiple times in JUNIT and expecting a false response which was received.

The code was built efficiently by containing class objects with setters and getters that could be called many times without repeated code. The functionality to check if an id was already in the system was built directly into the service classes to ensure all input to the program would be met and this would not be needed at the interface level. Finally, the functionality to test null and the string length was built directly into the class at the top so if there was an issue with the inputs very little code will be executed and the program will let the user know to change their responses.

To accomplish the ContactService, TaskService, and Appointment Service milestones we used testing techniques for specification-based testing, equivalence partitioning, boundary value analysis. Specification based testing is also known as black box testing and relies upon deriving test cases from specification documents built before the coding started. This is an integral testing technique and basically asks the question, Does the program accomplish what the client requested? The specification documents “do not define how a system should achieve the specified behavior when it is built” (Hambling et al., 2015) thus specification-based testing checks if the logic the developer implemented properly accomplished the task it set out to do. This type of testing was used often in our testing. For example, the test service requirements stated that the program should add tasks with unique id, delete, and updated tasks using the unique id. We tested this by checking to see if the task was added properly, if the program failed by allowing multiple same ids, and if the delete and update services properly performed functionality based on the unique id.

The next technique that we employed was equivalence partitioning which is used as a technique to counteract the limitless possibilities of user inputs. Instead of testing the potentially millions of user inputs into a program input are grouped into similar categories. For example, if the input is any real number the number could be positive, negative, or 0. It is not necessary to test every negative number as it would take a long time and will not likely yield an undesired result. It is a technique to make the testing cycle more efficient. This technique was used for example in our program when testing the Task class. The requirement stated that the id could not be longer than ten characters. So, when testing we attempted one id larger than ten. There was no need to test every possible greater number of integers to id, this is expected to be covered thanks to equivalence partitioning. The next technique we employed follows closely with equivalence partitioning and that is boundary value analysis. Boundary analysis agrees with equivalence partitioning, but it specifies that the chosen item to test from the set should be one that falls on the boundary. As stated in Software Testing “errors tend to cluster around boundaries” (Hambling et al., 2015), thus making this unit of a set the ideal to test. We used this technique in the Task program as stated above. When testing whether a unique id larger that ten could be created we tested 11 characters, as this was the boundary and the most susceptible to errors.

To accomplish the milestones in this and previous modules not all testing techniques were used. The ones not used were decision table testing, use case testing, and structure based testing such as control flow graphs, and experienced based techniques. Decision table testing looks at the overall decision structure of the program and how this affects the business rules. An example could be if a company offered two different types of discounts that needed certain criteria to be true for the user to be able to access them. One rule could be free shipping if a gold customer, spent more than $500 this year and live withing a certain demographic. Another one could be that to receive the 10% off discount a certain amount needed to be spent in the cart or a coupon code needed to be entered. It is important to test that each scenario results in the proper business rule being implemented before the site is launched and either doesn’t give the proper rule to the customer or gives it to them when it shouldn’t of.

Use case testing is testing to ensure that the program behaves differently as defined depending on the user. An example could be a web application displaying different functionality to a user depending on if they are signed in or not, or an application giving users more options if they are paying customer as opposed to free subscriber.

Another type of testing not done in this project is structure-based testing. As opposed to black box testing that builds test cases dependent on the requirements documentation white box techniques look at the code itself. One way to test this is by reading pseudo code and checking that it is logical. As well tools like flow charts, and control flow graphs can be helpful to test the logic in the pseudo code. A great tool for a tester is experienced to know where to look and find the errors that are in the program.

`1

**References**

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) – 4 est Design Techniques.* BCS The Chartered Institute for IT. Retrieved from  
https://app.knovel.com/hotlink/pdf/id:kt00UC2GJ2/software-testing-an-istqb/why-software-fails