# CS-320 7-2 Project Two

James Furman

james.furman@snhu.edu

Southern New Hampshire University

CS-320-T5520 Software Test Automation & QA

Prof. Ramsey Kraya

June 18, 2023

The unit tests we will discuss were designed for six components of a mobile app programmed in Java: Contact, Task, and Appointment classes as well as three Service classes, each to handle one of the first three. Contact, Task, and Appointment objects require unique, immutable ID string fields as well as other String fields, each of which has a length constraint, and the appointment class has a required Date field which may not be set in the past. Additionally, none of these fields may be null. Each Service class must be able to create new objects of its respective type, find objects by ID, and reassign the objects’ fields (other than ID).

Unit testing is the process of writing software that runs the software being tested and compares its output against pre-designated valid or invalid values. The tests’ approach was well-aligned with the software requirements. For each requirement, I wrote one test to verify correct function by making correct calls with valid arguments, and then implemented tests for each kind of invalid argument or invalid call. This included too-long arguments and null string arguments for all tests, as well as searches of nonexistent Contact, Task, or Appointment objects when testing the functionality of updating class fields from the Service classes. This handful of paradigms ensured that the classes’ requirements were fully met.

I ensured my code was technically sound mainly by static testing, using Eclipse’s syntax checking function. But in terms of JUnit tests, I ensured technical correctness by functional coverage. Since, as mentioned, these classes are relatively simple, code coverage accounts for most functional coverage, but some function testing must be approached with greater nuance. For example, in lines 183-207 of the TaskServiceTest JUnit test, I tested whether serial deletion worked correctly; this exemplifies functional coverage testing in that it ensures that correct deletion of a class object is not impeded by the prior deletion of a class object changing their containing list.

I ensured my code was efficient by multiple approaches. For one example, when the Contact, Task, or Appointment classes are passed invalid arguments, they throw exceptions (e.g. lines 17-21 of Appointment.java), so that the integrated application will be able to respond properly to bad inputs. I also wrote the Service classes’ delete functions to return Boolean values representing whether the searched ID was found and deleted, such as in lines 59-67 of ContactService.java; again, this allows the application to respond more dynamically and efficiently when calling the delete functions.

I used input equivalence partitioning to reduce valid and invalid String field inputs to a two-value boundary analysis, passing either a value on the boundary of valid String length or a value one character too long. Date values cannot be precisely boundary tested so I partitioned inputs into past (the invalid partition) or future (valid). These tests concisely verified that the objects’ constructors and setters accepted valid String and Date values and throw errors in response to invalid values.

I also used state transition testing, which verifies that the unit properly changes its behavior according to state changes from previous behavior. In these cases, the service classes retain lists of their respective objects and can add to and delete from those lists. They must be able to access list items and receive proper responses from attempts to access nonexistent list items, and these functions must not be impeded by prior changes to the lists. Although these are not explicitly defined as states in the requirements, they do represent a stateful aspect of the Service classes; therefore, I used the functions to add and delete list item objects in the service class tests to test the state transitions caused by list changes.

I did not use decision table testing to test these modules. This technique involves using a table to map decisions made in a unit and interactions between those decisions. A decision table can become highly complex and require curated attention to the relevant decision interactions, but in these cases the opposite was true. The decisions in any of these units do not interact with sufficient complexity to warrant testing according to a decision table; rather, proper decision functionality can be resolved simply by reviewing the code statically. The overall mobile app, however, is likely to be sufficiently complex that decision table testing will become useful.

I also did not apply use case testing, which involves modeling distinct types of users and examining the different ways they will interact with the software. Because these units are limited in scope and their functionality is defined very clearly by their requirements, it is not necessary to conceptualize their relations to various users. Like decision table testing, use case testing is likely to be valuable for the integrated mobile app because it will be more complex.

Although these code units were relatively simple, they do involve a certain degree of integration since calls are made to the Contact, Task, and Appointment classes from their respective Service classes. They also exhibit a degree of complexity, as any program beyond a “Hello World” is bound to. For instance, as previously mentioned, the state transitions caused by modifying the Service classes’ list members could potentially cause unintended interactions, so it is reasonable that adding and deleting from these lists, as well as attempting to delete nonexistent objects from them, be tested. While this approach may take longer to implement, that time is a worthy expenditure to exercise proper caution.

It is also critical to limit bias in testing code. The most direct way to do so is for the tester to be a different person from the developer; this approach is effective because developers tend to have a sense of possessiveness toward their code and their entire demeanor is that they want their code to work. Testing necessitates trying to prove that code does *not* work, which is harder for the code’s creator to manage. In this case, I both developed and tested the code, so I needed to actively work to prevent the bias I had as a developer from coloring my testing approach. The main technique I used to accomplish this aim was designing tests based on the software requirements, rather than on the code modules I was testing. I put the Contact, Task, and Appointment units out of my head and instead, looking at their respective requirements lists, asked, “How can this requirement fail?” Then, having conceptualized how the requirement could be shown as unmet, I went back to the modules under test and ascertained what, technically, those failures would look like; this then served as the basis for my test classes.

Because of the complexities in the program, problems can arise unnoticed, and minimizing buggy and vulnerable code requires a disciplined approach to testing. For example, when I first wrote the Contact class it replaced too-long and null String arguments with default, valid (i.e., non-null and not too-long) values. While this approach technically satisfied the class’s requirements, it created a technical debt because String values are not standardized and integrated with exception handling in the way that thrown exceptions are. This debt could create unpredictable behavior in integration and even release or, in the best case, it could require extra work parsing the default values when re-prompting end users for valid inputs. Refusing to cut corners and instead replacing this approach with throwing exceptions in response to invalid arguments is a disciplined approach that makes the program’s behavior more consistent and predictable, preventing problems and expenses down the road.