For this project, we were employees of Grand Strand Systems, given a general overview of what one of our clients was looking to accomplish. We were also provided a list of requirements for each of the three main components of the mobile application: the Contact, Appointment, and Task services. These specifications helped guide the process developing the tests for the codebase, serving as documentation on how the program should behave, as well as the parameters by which the inputs should fall under. Inasmuch, these requirements were what I used to go by when I was planning out my tests for the three components and their associated service files.

As an example, the id fields for the Contact, Task, and Appointment classes were required to be non-null, and under a certain number of characters. I translated this requirement into code by writing a test for each that would throw an exception when a user attempted to enter an input that was longer than the maximum acceptable length, and another test for when a user failed to submit an id for the creation of a task, contact, or appointment. Another piece of functionality the client requested was the ability for the classes’ associated services to be able to create new objects, and to edit those objects. To accomplish this, an array of class objects was used to store them upon creation, this handled by the ContactService.java, TaskService.java, and AppointmentService.java files. Each had a callable method for getting a specific id from the user, iterating through the array for a match, and performing the necessary action of adding a new object, deleting it from the list, or changing its parameters. Specified ids for which there was no match found, a message could optionally be displayed to the user, but no other action need be taken.

While the functional requirements help us programmers determine what needs to be coded and the expected results, we can use JUnit’s features to see how well our tests cover the codebase. You can use JUnit to check which lines of code are covered by appropriate tests, highlighting lines in green or red. This makes it easy to tell quickly at a glance which areas are included, and where the programmer should focus more attention on. It is at this point where I was somewhat confused. For the Contact.java, Task.java, and Appointment.java files, JUnit reports very high coverage for these classes and their attributes and methods. However, for the associated ContactService.java, TaskService.java, and AppointmentService.java files, coverage seemed markedly lower. While I am certain I coded the tests properly, I am wondering if these files were not correctly calling their counterparts; the Contact.java, Task.java, and Appointment.java classes.

One of the primary types of software testing techniques used for this project was unit tests. These assess the functionality of small, individual bits of code, before they start interacting together in larger systems. This was done individually on each of the three main components and is useful for finding logical errors in our code. These were done separately from one another as it is often much easier to discover defects in code when examining it one piece at a time. Later, one might consider that integration testing was implemented, as, for example, the Contact.java file was tested alongside the ContactService.java. Here, once the individual parts are tested, we can start building the system up and assessing its functionality at a higher level.

As the project progresses further in development, other testing techniques can be utilized to further assess it for defects. Regression testing could be conducted to ascertain whether newer versions or additions to the system ‘play nice’ with the current state. It is natural to assume that an application will need continued support after it hits production, this by way of updates and patches. Regression testing is important in this regard as these updates may be installed on live environments, where errors can have serious ramifications. Another type of testing typically done in this field is user acceptance. This would usually be done near the very end of the project, before the client signs off for its completion. Acceptance testing is when the development team showcases a near-finalized version of the application, and put it into the hands of the end users for their input. Granted, this could be done in early stages of development, where smaller working portions of the app are tested by the users, giving the opportunity for any miscommunication or changes to come to light sooner in the project’s life. Ultimately, acceptance testing is used to gauge whether the app performs as the client needs, and the necessity of its inclusion cannot be overstated.

As a software developer and tester, it is paramount to always code with caution in mind, as failure to detect and remove bugs from our code can have dire consequences given the right settings. One such thing to be mindful of is bias toward our own code. It can be easy to mistakenly be hurt or offended when errors are found in our code and, when in situations where we may be writing the tests to assess our own programs, may be tempted to not create tests that would highlight these defects. This may arise from the belief that our code can undoubtedly handle whatever input it is handed, or that the user will think the same manner as us when interacting with the program and so, willingly or inadvertently skip proper testing of those features. But proper edge-case testing should always be conducted, to rule out any oddities input may bring about.

Reiterating the above that defects in our programs could have dire consequences provided an unfortunate environment, we developers cannot see all methods in which our code might go awry, resulting in catastrophe of loss of life. The article I read for this week’s discussion showcases the innumerable errors evident in the military’s F-35 fighter jet program. The project has seen numerous setbacks, costing the government millions of dollars, but this is not the worst case scenario. Any number of malfunctions could occur due to software issues that end in people dying. But this is merely a hypothetical; a simple Google search can net you many instances of similar occurrences where programmatical defects resulted in accidental deaths. There truly can be no higher reason than this to ensure our applications are fully tested before they reach deployment.