**Module 7-2 Project Two**

**CS-320 Software Test Automation**

**Michael Reynolds**

When developing the three features for the Software Application (Contact, Task, Appointment), several steps were taken to ensure that the Unit Testing was aligned to the software requirements. For each of the described requirements, a Unit Test was made to consider both existing and instantiated objects with various members. E.g. a Task would be constructed with specific parameters expected to pass and expected to fail (Null, Length/Type differences) and then the objects had proceeded to be tested through a full life-cycle with involving methods with both invalid and valid data to ensure errors were handled properly after construction as well.

Utilizing and ensuring full coverage and checking for both passing and expectation of failure Unit Tests, I was able to ensure the overall quality of my JUnit tests to be sufficient/effective. It’s likely there are approaches/inputs not tested for, but within the realm of the requirements and defects expected in the Software Requirements, all bases are covered, and extra testing was also done.

My experience with writing the JUnit tests and ensuring that the code was technically sound and efficient was pretty smooth. In order to ensure that the code was technically sound and efficient, I followed a few rules. Using specific and descriptive test names ensured that I was not only being clear on what each test interacted with/tested for, but also that should someone else review my code, it would be easily understood and the actual pattern can be followed. Next I made sure to follow proper JUnit Assertions. Instead of having repetitive code or uncertainty of the call stack sequence, the utilized JUnit Assertions also helped to ensure it was clear which methods were called first and when they were repeated throughout the testing lifecycle.

Data was both unique to the Tests and also utilized shared passing data to ensure that both the standardization of input data and the isolation of each unit test. A good example of this is the following code:

Passing variables to share between all Units expecting passing data.



The “newDate” Date variable is an isolated variable specific to the Unit Test to ensure that there isn’t unexpected complications.



Throughout the software testing, there were a variety of techniques employed in the project all within White Box Testing. To start, all of the tests in our Unit Testing were done with Acceptance Testing at the forefront, in order to ensure that the Unit Tests included all client requirements. System testing was done by testing the dependencies and interoperability of the classes, ensuring they cooperate as expected, maximizing module to module engagement. Integration testing was included in this as it was two semi-coupled systems interacting with one another, these two testing methods usually came hand-in-hand. As far as Unit Testing, each of the Units covered the classes in both coupled and decoupled scenarios, to ensure standalone isolated functionality.

There were several techniques not utilized which included mostly Black Box Testing scenarios. All the Unit Tests involved running knowledge of the System and testing with it. Exploratory Testing or Ad-hoc Testing was not done, this can be considered manually extensive testing, such as testing for database interoperability to the front-end. This would be used for something like testing for valid table coverage etc. Security Testing was also not done, which security was at most considered in member requirements/input validation. This didn’t take into consideration security well, and wouldn’t provide testing against things like malicious payloads. There was also no utilization of Regression testing or Performance testing, things like revisions to the system and stress testing the system was not of concern in the Unit Tests.

When in the mindset of tester and developer, I found that there were a lot of shortcomings in my initial approach. I never really understood the importance and the cataclysmic outcomes that could be from not having proper testing. Originally I thought “testers” were always just a role that the developers will normally take themselves, but I see that it can be a position all in its own right, and in most cases would actually be beneficial this way. As a tester, I reviewed my code with extreme caution, knowing how systems were interoperable and how they need to be exhaustively tested in isolation is paramount in employing our tests correctly. Without an appreciation for the complexity and interrelationships, it’s likely we would miss use cases even with JUnit showing 100% coverage. It’s even more important to always assume we missed cases and expect the unexpected, but to give it the respect to ensure we exhausted everything to our knowledge is a very sure bet when testing.

As a developer, there was definitely a bit of bias at first. When I began testing I found various bugs even though I believe myself to have coded the systems “flawlessly”. I was humbled by my tests failing and even some breaking as they showed the flaw in my bias. I imagine it would be a concern if I was responsible for testing my own code though as well, although the client requirements seemed to cover pretty extensively the way variables should be initialized and classes should be limited etc. I can imagine myself construing client requirements from a client with less terminology/software development knowledge. When testing my own code, had I not known the length requirements etc. I would’ve probably never tested for ridiculous inputs that could put a major load on a back-end. With this knowledge, I’ve grown to respect testing in its own right, and to carry a grain of salt with my developing and testing to ensure I’m not getting too ahead of myself in regards to patience and care when testing. In the future, cutting corners or taking short cuts is something no individual and/or company can afford, and with malicious actors in the world, it’s an absolute disservice to not take writing and testing code seriously and with extreme caution. As a practitioner in the field, I plan to avoid technical debt by staying up to date, staying edge on testing techniques and practices, to ensure I’m knowledgeable on the status quo and am keeping my development and testing as secure as possible. Keeping up with publications, academic write-ups and industry experts on everything from optimal test development and risks/incidents to learn from.

I plan to break into the industry with a newfound respect, curiosity, and cautionary approach to testing and development, with huge consideration to the implications of neglect, the budget costs due to such neglect, and the dangeours potential for progress loss.