CS-320-R4835 Software Test Automation& QA 24EW4

Peter Thoman

Project 2 Submission

Unit testing is key in stable software development aiding the process in its overall success. Without testing software before integration developers run extreme risk of failure and put lots of resources at unnecessary risk. Using JUnit5 throughout this course I have been assigned to write test harnesses in tandem with my development of the required Java code. These tests were all geared to the functionality of my code making sure the processes within the class execute effectively. There are also boundary tests that attempt to break my code by executing functions beyond their given boundaries. Specifically in each test class there are tests that consider the valid creation of the objects created by the object class. Testing each service class creates a valid version of the class container object before each test case following in the JUnit script and utilizing the @BeforeEach function. All the tests I created were requirement focused because they focused on attributes of the application that were created based on the requirements given by the assignments and some were my own perspective on errors that could possibly occur within the program.

Throughout these projects I followed the guidelines provided within the assignments in order to produce successful solutions for the whole application. My coding was done on my local PC and not through the schools online Apporto portal allowing me to more efficiently edit and create the necessary code. Unfortunately, the JUnit library in my version of Eclipse would not attach itself to my code source libraries which made testing my tests ineffective. I assume they are covering a meaningful percentage of the application and its requirements but have no real data or percentage to argue my case. In the future when I have issues with nonfunctioning plugins or references, I will copy my code over to the virtual IDE in the Apporto portal in order to witness my code react in the intended environment enabling more accurate data regarding testing coverage percentages. Creating JUnit tests was rather straight forward and seemingly very logical when focusing on the functioning attributes of the classes created for the assignments. By working through these assignments, I have developed a better understanding of what the components of unit testing are and how they can be implemented into the development lifecycle. The importance of clear and well covered testing is paramount in successful system implementation minimizing system failure and user error by validating structure and function before use. Some tests are even designed to mimic forms of implementation like the tests attempting to addContact/Task/Appointment in the unit tests that effectively create the objects as if a user were entering the attributes into the I/O stream.

Writing these codes in Eclipse incorporate all of the built-in testing functionality an IDE provides. Behind the scenes the IDE is unit testing my code to guarantee it follows standard syntax and logic constraints in Java. Forms of Whitebox testing are simultaneously testing my code, and the tests I write regarding my class objects to ensure proper syntax use and logic structure. Otherwise, when writing the code, I formatted each with the same identifier between lines 1-5, and commented each major element of the codes with the standard “//” double slash ignore identifier. Each comment is meaningful and helps guide the future reader through the scripts and what they are doing. Being maintainable is a major priority regarding a code being technically sound and by following standard development practices I have written straightforward code that is technically sound and easy to read. All of the services for each object container were designed as binary search trees to promote code efficiency and processor usage. BST’s are very effective in managing, referencing, and editing data because they limit system complexity and better manage processor resources in design.

The tests I created for this application focus mostly on the requirements given from the assignments prompts and the creation of the object classes and their functionality. I have implemented some boundary tests regarding attribute input size along with some state transition style tests focusing on how the class constructs and modulates its created objects. The add, and delete function tests focus on state transition and how the objects communicate with their storage services. White box testing was not a focus of my tests mainly because I was unsure how to write JUnit tests that focus on syntax and code logic. The Eclipse IDE has a lot of built in Whitebox testing systems that test the code I write for syntax and logic simultaneous to the script being written in real time. All of my tests are subject to the inherent bias I have creating this code. Theoretically an outside tester could use the tests I created or even create other tests that I do not see because of my bias.

Testing code is equally important to the development of code. Without effective testing procedures, time and resources are needlessly lost to replace haphazard development of code created without this methodical approach. It is obvious that testing is an ongoing activity maintaining stable code and without it most systems would not be a success. Whilst adopting a tester’s mindset throughout this course and its assignments my sense of awareness to the notion of too much testing and its detriment to effective code creation was glaringly apparent. Theoretically a tester could endlessly create test cases for every aspect of a code cycling through redundant and irreverent data that has no real bearing on system effectiveness. Deciding where to draw the proverbial line between effective testing and needless testing can sometimes not be a very straightforward concept. The assignments I worked on in this course tested my novice coding capabilities but lent a new perspective regarding the level of complexity a true application warrants when undergoing any form of testing in the development lifecycle. The classes had some functions and variables while communicating with coordinated class services within one package. When programs adopt more encapsulation, polymorphism, and inheritance than what we programmed in our assignments, unit testing can become very complex and take a lot of time to create intuitively. Bias adds another level of intricacy too that can cripple development because of its inherent lack of perspective. It is understandable why firms would want to outsource their testing to limit inherent bias.

There was not a lot of bias truly when talking about theses assignments but I understand now by completing them, how bias plays a major role in test development. What one developer sees as important, another my find inconsequential and this concept being applied to large scale projects is error prone. While some teams may outsource testing to avoid bias others may use bias to their advantage and play one developer against another effectively opening perspective by harnessing weakness as strength. Having pride in one’s work is essential to success but the downfall of confidence is arrogance where obvious issue is overlooked to avoid contradiction to concept. Avoiding inefficiency by swallowing pride is essential to creating effective and efficient systems created to assist opposed to desisting users.

As a trades professional I have an inner quality focused on having pride in my own creation. This translates to my future development as a software programmer and I realize the importance of creating quality sustainable systems utilized by many as opposed to few. Developers need to remain errant in creating code that others can use by implementing standard development practices like comment formatting and descriptive documentation. All kinds of pseudo code, flow charts, and UML diagrams are effective tools at preserving a programmers work while mapping out schematics to the code for effective testing implementations. Avoiding using these tools along with implementing efficient testing methods is a way to create a code only one person will use, and most likely ineffectively overall. JUnit, Cucumber, and all of the unit testing addons for API development only aid the coder in creating a maintainable system which is the legacy of the programs existence. Poor program development is error prone without testing and not successful for problem solving making proper development methods and program testing methods the obvious choice when constructing a new system of any kind.