Project Two:

The JUnit testing framework allowed me to test each software requirement for Grand Strand Systems. Based on each requirement, I could create a test case that verified the functionality of my code. For example, in the task service project, the requirement for task ID was that it was not more than ten characters, it was not null, and it was not updatable. I ensured that these guidelines were met through my test case using the assertThrows() function. This tells JUnit that the program should throw an illegal argument exception when the task created contains an invalid value. One of the lines in my code that demonstrates this functionality is “Assertions.assertThrows(IllegalArgumentException.class, () -> {new Task("Task6537", "NamingTask", null);.” The task created has a null value where the description is expected. Since the description cannot be null, an illegal argument exception error will trigger, and the JUnit test with confirm that it has.

My code is technically sound due to verifying each listed requirement. For values that could not be null or surpass a set number of characters, I had individual functions testing for errors in each use case. For example, if the classes name value could not be more than 20 characters and could not be null I utilized the functions testNameTooLong() and testNameNull() to verify this functionality. The adding, deleting, and updating requirements were all checked for. For the adding and deleting functionalities, both passing and failing use cases were verified. This along with a passing full coverage run from JUnit is a testament to the quality of my code.

I ensured my code was efficient through the use of ArrayList. Using an array made it simple to dynamically add and delete contacts and tasks. It also made it simple to loop through to check if an ID was already being utilized. An example of the ArrayList in action can be seen in these lines, “tasks = new ArrayList<>();…. for (Task taskList : tasks) {if(taskList.equals(task)) {taskPresent = true;}.” The array, "tasks", could be called in this for loop and searched through to verify the ID for this task isn’t already present. If it is present, the function would return false, and the task would not be added, otherwise, the function would return true, and the task would be added.

Throughout development for Grand Strand Systems, we have exercised the use of white box testing techniques. White box testing is used to verify the inner workings of a software system. This can be done through statement testing and decision testing. Statement testing exercises the executable statements in the code. This testing forces the program to execute statements to ensure that what is required in the source code runs as expected. An example of this in my work with the contacts, task, and appointment projects is with error handling. In my tests, I verified that an error was thrown if an invalid input was submitted for each of the specifications we were given before development. Decision testing also forces statements to run, but unlike statement testing, it focuses on true or false results. This helps verify that both exits are met when the expected triggers are provided. I utilized this in my work through the use of assertEquals() functions. For example in my AppointmentServiceTest class, I added an appointment and verified that it returned true in the addAppointment() function, I then tried the same appointment again and verified that it returned false. It returned false because an appointment cannot be added when an appointment with the same appointment ID already exists. This helped me as a tester understand that the conditions for both valid and invalid inputs worked as expected for adding an appointment.

The projects we have developed haven’t utilized many black box testing techniques. These techniques include equivalence partitioning, boundary value analysis, decision table testing, state transition testing, and use case testing. While white box testing is focused on the internal structure of a system, black box testing is focused on functionality. We did have a limit on the number of characters for numerous variables. In those cases, we did practice equivalence partitioning by providing both a valid and invalid number of characters to verify if the requirement was being met. Boundary value analysis can go together with equivalence partitioning in our case since partitions must have boundaries. An example of this in the appointment project was checking if errors were thrown when an appointment description was null or more than 50 characters. Decision tables and state transitioning aren’t currently applicable to what we have developed so far. The decision table technique is used to demonstrate business rules. These rules list out the conditions and actions of a system to demonstrate possible interactions. State transitioning is important when user input can change. In some systems, current and past values can be important to track and consider especially if it triggers additional actions. Use cases are used to specify the functionality of a system. In the appointment project, you could consider us having use cases for adding appointments and deleting appointments. While we tested the functionality of those capabilities, they aren’t fully complete without the user interface. Since a use case is between the user and the system, the interface is an important feature to facilitate communication between the two.

In the development of user-facing applications, it is important to consider the number of ways the user can interact with a field. While there is an expectation for how a user will input a set of data, it is more than likely that some users will not follow that expectation. Since this is the case, I wrote my JUnit test cases with caution in mind. For example, when a user adds a task, the task requirements ID, name, and description all have character limits. This is an important safety feature as some users take advantage of input fields to pass malicious code through the server. My JUnit tests check to make sure these character limits are adhered to through IllegalArgumentException throws. Protection from attacks should always be a priority when creating and testing requirements.

Working as a developer, bias is important to keep in check, especially during testing. I know I have experienced times when I have received a requirement from a client and have spent a considerable amount of time trying to complete it. It can become frustrating after a while if you cannot seem to see an end to research and debugging, it is common at this point to want to implement shortcuts. Code that isn’t efficient for the system and incomplete/unstructured code are some examples of how that could be executed. This is especially prominent if the requirement passes the client's expectations but as a developer, you know your code is prone to vulnerabilities. Within this project, bias could have been implemented by not fully testing each class. Testing what you know as a developer works as expected and not accounting for what doesn’t is an easy way to avoid addressing issues. In my project, I made sure every capability was accounted for in my testing through JUnits coverage capability. Utilizing this tool showed me line by line what I had and had not tested.

Quality code is a necessary standard to uphold as a software developer. As I mentioned previously, it is easy to want to cut corners on the quality of code, especially if the client’s requirements are met but that isn’t acceptable no matter the scenario. Not only do you want the code you have developed to run on the system efficiently and be simple for other developers to read and edit, but you also want to prevent defects. Defects that are logged but not resolved, or defects that were never tested for can create a lot of issues and increase a company’s technical debt. There are multiple examples of this happening throughout history. Defects have caused machine failures, accidents, millions of dollars lost, the spread of misinformation across massive platforms, etc. Specific examples include the Chinook helicopter disaster, Microsoft Tay’s AI experiment, and Hitomi’s satellite. Technical debt can be avoided by testing for any possible scenario and if a defect is found that was not accounted for it should be dealt with as soon as possible. No defect should be left unaddressed, especially for machinery that could kill if in a wreckage or interfered with. The article “Clear and present danger: why we refused to give up” by Karl Schneider was a real eye-opener on how operators are almost always blamed for wreckages potentially caused by software issues especially if they are dead. This tells companies they are free from consequences if their software is to fail and is a dangerous mindset to adhere to as a developer. Ensuring the clients and users of a product are utilizing your application as expected and safely is an ethical and essential way to go about development.

**Resources**

Academy, C. (2018, January 31). *When Coding Goes Wrong - Coder Academy*. Medium. <https://medium.com/@coderacademy/when-coding-goes-wrong-e46d84c6565f>

Hambling, B. (2019). *Software Testing: An ISTQB-BCS Certified Tester Foundation guide - 4th edition* (4th edition). BCS, The Chartered Institute for IT.

Schneider, K. S. (2002). *Clear and present danger: why we refused to give up*. https://login.microsoftonline.com/2baef15b-b8de-423f-9d8a-46f3686d8848/wsfed?wa=wsignin1.0&wtrealm=https%3a%2f%2fsnhuweb.snhu.edu%2fRedirect%2f&wctx=rm%3d0%26id%3dpassive%26ru%3d%252fRedirect%252fEZPRedirect%252fIndex%253furl%253dezp.2aHR0cHM6Ly9lZHMucy5lYnNjb2hvc3QuY29tL2Vkcy9kZXRhaWwvZGV0YWlsP3ZpZD0w&wct=2023-02-18T00%3a24%3a09Z&wreply=https%3a%2f%2fsnhuweb.snhu.edu%2fRedirect%2f

*The Explosion of the Ariane 5*. (n.d.). https://www-users.cse.umn.edu/%7Earnold/disasters/ariane.html