Jerris English

CS-320 Software Test, Automation QA

December 22, 2024

**Project Two: Summary & Reflections Report**

1. **Summary**

As a Grand Strand Systems software engineer, I was tasked with developing the contact, task, and appointment services for a mobile application. After reviewing the client requirements for the application, I determined that I would be developing CRUD code functionality for each of the application services. In other words, I would be developing code that allows for the creation, reading, updating, and deletion of contact, task, and appointment objects. In order to ensure that each of the three services aligns with client requirements, I would also be implementing unit testing. The unit testing approach I chose to implement for this application utilizes logic checks, boundary checks, and error handling. Logic checks are used to verify that each of the functions performs as expected with valid inputs; boundary checks are used to test how the system handles normal, edge case, and invalid inputs; and error handling checks that the system properly handles errors.

* **Contact Service**

For the Contact Service, in order to verify that functionality aligned with system requirements, the ability to create, read, update, and delete contact objects were tested. Contact object restraints were coded into the Contact class constructor and dictate acceptable inputs for Contact parameters including the contact ID, First Name, Last Name, Phone number, and Address. Logic tests were utilized to validate the successful creation of a new contact with valid contact parameters for the Contact class and to verify that a new contact could be added to a contact list in the ContactService class. For example, the testClassConstructor() method first instantiates a new contact object and then validates the instantiation by reading and comparing each contact parameter return value against the initial input parameter, where assertTrue passes when the parameters match. Similarly, the testContactServiceAddContact() method instantiates a new contact list and then attempts to add a new contact to the list where the assertTrue assertion passes as long as the contact list was successfully updated with the contact. The assertFalse assertion within testContactServiceAddContact() validates, per system requirements, that the addContact() method within ContactService returns false when a contact without a unique ID or null ID is passed as parameter.

In order to test system requirement contact parameter constraints, boundary checks were utilized to either validate acceptable parameters or invalidate unacceptable parameters. In the case of invalidated parameter constraints, error handling checks were used in conjunction with the boundary checks, as throwInvalidArgumentException error handling was incorporated into the Contact constructor and setter methods within application code. For example, system requirements dictate that a contact’s name should not be null and should not be longer than 10 characters. The testClassConstructor() validates acceptable parameters, while the testContactIdTooLong() and testContactIdIsNull() methods use invalid name input parameters to validate that the invalid argument exception is thrown within the constructor and setter methods. Similar parameter test methods are used to validate error checking for other required contact parameters including the contact ID, last name, phone number, and address.

Another system requirement that is validated through unit testing is the ability to delete a contact. The testContactServiceDeleteContact() method adds a new contact to the contact list and then assertTrue passes when a contact is successfully deleted using the contact id as deleteContact parameter. The method also checks additional paths within the deleteContact() method, validating that assertFalse passes when a non-existent contact ID or null ID is passed to the method.

The final system requirement for the Contact Service is the ability to update contact parameters. The updateContact() method in ContactService allows a search ID parameter for locating the contact, as well as either null or parameters to be changed values. The testContactServiceUpdateContact() method tests the updateContact() method, and assertTrue passes when a contact has been successfully updated. The assertion assertEquals validates that the contact parameters have been successfully updated. An additional assertTrue is used to validate the updateContact() paths where contact parameters, aside from the contact ID, are null and thus no parameters are updated for the contact. Two additional boundary tests are used to test updateContact() with a non-existent contact ID search parameter and null contact ID search parameter, both evaluating to false in the assertFalse assertion.

* **Task Service**

The unit testing implemented for the Task Service is very similar to the unit testing for the Contact Service. System requirement constraints and functionality dictate the ability to tasks to the task list with a unique ID, the ability to delete tasks per task ID, and the ability to update task name and description parameters per task ID. Much like the Contact constructor, the Task constructor handles the creation of task objects, and successful creation of Task objects is Logic tested through testClassConstructor() class using assertTrue to validate each parameter of a newly created Task object. Boundary tests in conjunction with error handling checks are also used within testClassConstructor(). These assertThrows checks handle paths for illegal arguments within the constructor method and validates that an exception is returned when invalid Task parameters are passed to the constructor. For example, system requirements dictate that Task name parameters cannot be null or exceed a length of 20 characters. The following snippet outlines how boundary and error handling checks are utilized to invalidate illegal task name parameter values:

A screen shot of a computer code

Description automatically generated

Also similar to the Contact Service, the Task Service adds tasks to a task list using unique IDs, delete tasks per ID, and updates tasks per ID. The unit testing methods used to check those services are virtually identical to the Contact test methods, albeit different parameters. The unique and not null task ID is still validated when adding, deleting, or updating a task through the testAddTask(), testDeleteTask(), and testUpdateTask() methods. Similar boundary checks are used where assertFalse in those test methods handles invalid ID parameters.

* **Appointment Service**

The Appointment Service is similar to the Contact and Task Services as each appointment object must have a unique, non-null ID; a non-null description that cannot exceed 50 characters; and the ability to add and delete appointments per appointment ID. However, the Appointment Service requires additional focus because system requirements dictate that each appointment date must also have a non-null appointment date field that cannot be in the past. Thus, while unit testing the appointment constructor in the testAppointmentConstructor() method, successful creation of an appointment object is validated not only by a valid appointment ID and description, but also by a valid future date. The assertTrue paths of the Logic test pass when valid parameters are passed to the constructor as shown in the snippet below:

A screen shot of a computer code

Description automatically generated

In addition to the boundary and error handling checks that test for null or invalid ID or description parameters, additional checks were added to test that exceptions are thrown when in invalid past date or null date are passed as parameters to the constructor. A snippet below shows the boundary/error handling check for date parameter exceptions:

A computer screen with text

Description automatically generated

The units tests for adding an appointment to an appointment list and deleting an appointment are relatively identical to those used previously for the Contact and Task Services.

* **JUnit Test Quality**

The overall quality of the JUnit tests utilized can be directly linked to the high test coverage percentage. Each test class and the overall test coverage for the entire project was at roughly 90%, which anything above 80% is considered strong. This overall high coverage percentage is tied to my thorough use of Logic, Boundary, and error handling checks for each class. For each isolated Logic and Boundary test, I tested all possible paths for each method. Additionally, for any error handling I featured in my code for invalid input validation, I made sure to add error handling unit tests for those exceptions.

**Personal Coding Experience**

One way that I ensured my code was technically sound was that I added assertions to validate vector container and object instantiation for all classes. For example, when I instantiated a new vector, I unit tested to make sure that the vector size was zero, but still created. Additionally, when I added a new object I made sure to include assertion checks to make sure that the vector size had increased/decreased with the new object additions or deletions. Examples below:

A computer screen with text on it

Description automatically generated

A screen shot of a computer program

Description automatically generated

Another way I ensured technical soundness within my code is by including Object-oriented checks. Whenever a method call was invoked that changed an object, I made sure to use assertions to check the status of all object parameters. For example, in the testUpdateTask() method, I tested adding and updating an existing contact. To ensure that the contact parameters had been altered correctly, I used assertTrue to compare the updated parameter values to the expected parameter values:

A screen shot of a computer program

Description automatically generated

In order to ensure that my code was efficient, I made sure to limit any code redundancies in my testing. Rather than have separate test methods for each assertion, I combined them into one testing method. This allowed me to write one testing method that incorporated Logic tests, Boundary tests, and error handling checks without the need for additional testing methods. Additionally, by utilizing the assertAll assertion method, I was able to further reduce redundant code. An example of this can be seen in the snippet of the Task constructor test:

A screen shot of a computer program

Description automatically generated

1. **Reflection**

* **Testing Techniques**

The testing techniques that I utilized for this project include unit testing: Logic testing, Boundary testing, error handling checks, and Object-oriented checks. Unit testing was the basis for this project, ensuring that each class and its methods were validated using isolation testing. Unit testing is beneficial for identifying issues incrementally at each component level without the need to debug the entire application at runtime. Logic testing is a type of unit testing used in my project to validate methods and all possible paths using valid inputs. Boundary testing was used in my project to test behavior related to how the system responded to invalid parameter inputs for methods. Error handling checks were used to validate that expected exceptions were thrown by methods when invalid parameter inputs were used. The last testing technique I used was Object-oriented checks which validated an object’s status and parameters after a method had altered the object.

The testing techniques that were not utilized in this project include integration testing, system testing, performance testing, and user-acceptance testing. Integration testing was not used because it is a type of testing that tests how the Contact, Task, and Appointment Services would interact with each other as a combined entity. System testing is a type of testing that takes place after Integration testing that examines the entire application ensuring they work as a complete package. Performance testing is a non-functional technique that tests application stability, speed, scalability, and responsiveness under a simulated workload. The last phase of testing that was not utilized in this project is user-acceptance testing which is the phase of testing that occurs in the real-world where end-users test the functionality of the application.

The practical uses and implications for each of these techniques is substantial and can really benefit an entire project. Unit testing is great for early defect detection, improving code quality, and facilitates code maintenance which can greatly reduce time and cost budgets for a project. Integration testing is great for projects that have large code bases with multiple modules. Testing how these modules integrate and behave with each other is extremely important identifying compatibility issues and performance problems. This sort of testing is really only necessary when multiple modules are closely dependent on each other. System testing is necessary for testing the overall system functionality and quality of the application. This type of testing reduces post-launch issues and ensures that the application meets user expectations. Performance testing is necessary for applications that rely on speed and optimization. Early performance testing allows the identification and mitigation of bottlenecks and vulnerabilities before product launch. User acceptance testing is necessary for client-facing applications as it ensures that the application is working as intended and meets client expectation.

* **Mindset**

The mindset that I developed during work on this project revolved around caution and precision. Since I was acting as both the code developer and tester for this project, as I developed the code I incorporated caution for the expected behavior based on user inputs. This caution led me to include error handling for illegal arguments passed as method parameters and then test paths where the illegal arguments were thrown. As I progressed further into the project I began appreciating the interrelationships of the code that I was testing. For example, when testing the addTask() method, I had to remember that the addTask() method depended on the Task constructor method to create a new Task object. Since most of the error handling was included in the Task constructor method, I didn’t need to include additional error handling checks for the addTask() method. I just passed the Task object as parameter to the addTask() method and then the object would be validated through the constructor. By understanding the interconnectedness of the different classes within the project, I was able to limit code redundancies while still ensuring a high test coverage for the project.

Testing one’s own code can be challenging to not introduce bias. This is because it is easy to overlook your own coding errors and run tests that sidestep the errors. Throughout this project, I tried to keep an external mindset or a mindset view of how a different developer or user would see or use the code. This led me to write code that followed industry best standards that was clear and well-commented. My code also factored in all expected and unexpected behaviors and the tests were written to explore every possible path of my code. By validating every path of each method including possible errors, I ensured that my code and tests were free from bias.

As a developer it is a very important commitment to quality that I must make. By writing quality code and unit testing early and incrementally, I can reduce technical debt for the project. It is also very important for me not to cut corners as a developer. Every single method and all possible paths should be tested and accounted for, effectively reducing project costs that otherwise would require significant refactoring if major issues were ignored during the unit testing and code development phase. In the field as a developer, I can reduce technical debt by writing clear, well-commented, reuseable code that factors in software requirements and proper error handling. Additionally, utilizing thorough unit testing such as Logic, Boundary, Error handling, and Object-oriented checks will help me uncover bugs and unexpected errors early on in the development cycle. These methods will help me hand-off quality work that will not need expensive refactoring in the future.