Summary and Reflections Report

1. Summary
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.

The software requirements for each class were already predefined prior to creating each class. For the first milestone we were required to create a “Contact class” that had objects with unique string ID’s that couldn’t be null, or longer than 10 characters. It required a phone string that couldn’t be null or longer than 10 characters, and an address field that couldn’t be null or longer than 30 characters. The “Contact service” class had similar requirements. The ability to add contacts with a unique ID, ability to update first name, last name, number, and address fields. With the aid of J Unit testing, I was able to verify that all these requirements were met.

|  |
| --- |
| This J Unit validates that we can add a new contact by creating a new contact. Then using assertTrue, it checks to see if all the new contacts information is stored. We tested each aspect to include contact ID, first Name, and last Name.  @Test // Tests to see if we can create a new contact void testContact(){  Contact contact = new Contact("222333444", "Amelia", "Majesto", "1111111111", "450 Bubblegum Rd");  *assertTrue*(contact.getContactId().equals("222333444"));  *assertTrue*(contact.getFirstName().equals("Amelia"));  *assertTrue*(contact.getLastName().equals("Majesto"));  *assertTrue*(contact.getPhone().equals("1111111111"));  *assertTrue*(contact.getAddress().equals("450 Bubblegum Rd")); } |
| This test checks to verify if an ID is no longer than 10 characters. The newly added contact has an ID that is 12 characters long. Using assertThrows(IllegalArgumentExceeption), we are testing to verify there is an error with the new contact, as the ID is longer than 10 characters. Similar testing was used to verify that the addresses and phone numbers were under their designated character count.  @Test // Test to see if it throws an error if Id is too long void testIdLength(){  Assertions.*assertThrows*(IllegalArgumentException.class, ()-> {  new Contact("454545454545", "Amelia", "Majesto", "1111111111", "450 Bubblegum Rd");  }); } |
| This test checks to validate that a field is not null, using assertThrows(IllegalArgumentException). In this example, a new contact was created and the first name field was left “null”. The test verifies that there is an error with the first name field. Similar testing was used for, last name, phone number, and address.  @Test  // Test to see if it throws an error if first name is null void testFirstNameNull() {  Assertions.*assertThrows*(IllegalArgumentException.class, () -> {  new Contact("45454", null, "Majesto", "1111111111", "450 Bubblegum Rd");  }); } |
| This test verifies that we can delete contacts by creating a new contact, then deleting it. The test uses assertNull to verify that there is no contact under that ID stored.  @Test // Test to see we can delete contact public void testDelContact(){  Contact contact = new Contact ("405", "Bruce", "Banner", "1888555444", "1 Stark Tower Pl");  contactService.addContact(contact);  contactService.delContact("405");   // Test to confirm contact deleted  Contact searchDelContact = contactService.getContact("405");  Assertions.*assertNull*(searchDelContact); } |
| This test verifies that we can update the address field by updating the address. Then assert.Equals was used to verify that the field had been updated. Similar testing was used to check if first name, last name, and phone number could be updated.  @Test // Test to see if we can update address public void testUpdateAddress(){  Contact contact = new Contact("410", "Black", "Widow", "1888888888", "45 North Shore Rd");  contactService.addContact(contact);  contactService.updateAddress("410","18 red dr");   Contact searchUpdateAddress = contactService.getContact("410");  Assertions.*assertEquals*("18 red dr", searchUpdateAddress.getAddress()); |

* + 1. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?

To test the quality of the Junit tests, I ran a coverage test. All classes came back with 93% or higher. This high percentage confirms that the Junit tests were effective.

A screenshot of a computer

Description automatically generated

* 1. Describe your experience writing the JUnit tests.
     1. How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.

I was able to ensure that my code was technically sound by following the requirements closely. All specifications were met, such as using fields that were within the character limits. Employing Junit testing to confirm that all entered fields were within the specifications helped to ensure the code was technically sound. For instance in my contact class I set the string ID’s to be less than 10 characters, and not allow them to be null. This method was employed with each specification.

public Contact(String contactId, String firstName, String lastName, String phone, String address){  
 // Create Null and String length conditions  
 if (contactId == null || contactId.length() > 10) {  
 throw new IllegalArgumentException("Invalid Contact Id");  
 }

* + 1. How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.

My code was demonstrated to be efficient as I kept it neatly formatted using inline comments and easy to follow identifiers to allow for a better overall understanding of the code. Using identifiers such as:

setFirstName(firstName);  
setLastName(lastName);  
setPhone(phone);  
setAddress(address);

1. Reflection
   1. Testing Techniques
      1. What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.

The software testing techniques I used majorly included JUnit testing. It uses small segments of code and tests each small section to ensure that all components are working properly. Due to the technique of JUnit testing, this made is easy to incorporate regression testing into this project as well. Regression testing is essentially testing new poritions of code to ensure it does not change the outcome of prior code. Each method that was added did not impact any of the prior methods already incorporated.

* + 1. What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.

There are many other software testing techniques that were not used in this project. We did not use integration testing which finds defects in the interface of an application. This would test how components work together. We also didn’t use system testing, which tests to ensure the system meets requirements.

* + 1. For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.

Junit testing breaks down code into smaller units to be individually tested. Regression testing ensures that each additional portion of code does not render prior code obsolete. Integration testing validates that all the methods are integrated and work together. System testing validates that software meets the requirements.

* 1. Mindset
     1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

Software testing was a concept that I struggled with to begin with. Acting as a software testing really helped me build confidence and gain an understanding of how to write tests to ensure software is functioning correctly. I needed to employ caution and test every speficication to ensure that there was no room for potential data entry errors. When working on each test, I followed each class like a rubric and went down the line to check that each test was covered. Such as adding, deleting, updating, and staying within the character limits. Setting up a constructor helped to ensure the data stayed organized, such as:

public Contact(String contactId, String firstName, String lastName, String phone, String address){

* + 1. Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

I think the best way to eliminate or limit bias is to be diligent in testing all portions of code, and to maintain an open mind for critique and learning opportunities. With the rate at which the tech field advances, we need to always be open to learning ways to improve our existing knowledge.

* + 1. Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

It is important to not cut corners when it comes to software testing, as missing the tiniest fragment of an issue can potentially lead to catastrophic results. Coding itself is meant to be very detail oriented, and you need to establish a keen eye for every little detail. To avoid technical debt, I will always test code in smaller portions and ensure all functionalities are met before the software is turned over to the client. Being ahead of the game will avoid costly fixes later. One example would be the Airbus A400M. Accidental deleting of a much-needed package and not checking before the product was handed over to the client caused a fatal crash of a test crew. This costly mistake could have easily been avoided if they had thoroughly tested all the software prior to sending the product out.