Hayden Gentry

Dr. Jonathan Norman

CS 320

30 June 2024

CS 320 Project Two

My unit testing approach began by understanding the software requirements for the Contact and Contact Service, Task and Task Service, and Appointment and Appointment Service classes. To start with Contact, Task, and Appointment classes; the software requirements had constraints on the variables within the object. The first part of our unit testing approach is testing the constraints of the variables of the initialized object. Our testing would include both positive (expect to pass) and negative (expect to fail) tests to validate the variables of the initialized object. The positive tests would consist of an initialized object with all its variables within the constraints. The negative tests would consist of an initialized object with one of its variables outside of the constraints, either the variable length is too short or too long, the variable is null when it should not be null, or the variable is a past date when it should be a future date. I tested the variables one at a time to confirm that the tested variable is responsible for failing the test. This requires multiple negative test instances to check each of the variables within the initialized object.

The next step is to unit test the Contact Service, Task Service, and Appointment Service classes. This begins by first developing the required methods for each of the classes outlined in the software requirements. The methods for these classes include adding new objects, deleting created objects, and updating created objects. The first thing we test in each of the classes is if the object ID is unique. I conducted a negative test by first creating a validated object and adding it to the hash map that we use to store objects, then creating a second validated object with the same object ID and attempting to add it to the hash map. This functionality for checking if the object ID is unique makes up the add new object method in our class, I also conducted a positive test to ensure that an object with a unique ID is added to the hash map. For the delete object method, I conducted a positive test by creating a validated object and adding it to the hash map, then removing the object from the hash map, and testing that the object no longer exists. I also conducted a negative test by attempting to remove an object that does not exist in the hash map. For the classes that include an update object method, I conducted a positive test by creating a validated object and adding it to the hash map, then updating the variables within the constraints previously outlined and testing that the object’s variables have been updated within the hash map. Like the delete object method, I conducted a negative test by attempting to update an object that does not exist in the hash map. I also conducted multiple negative test instances for the update object method by attempting to update an object variable outside of that variable’s constraints.

Through these JUnit tests, I was able to achieve a coverage percentage of 100% for all the six classes that were tested. This was accomplished by ensuring that the code was efficient and technically sound before testing. One example of efficient code was the update method in the Contact Service class. I was able to keep the code concise by passing our updated object as an argument into the method, then using the setter methods to both validate and update the variables of the object.

public void updateContact(String contactId, String firstName, String lastName, String phoneNumber, String address) {

Contact contact = contacts.get(contactId);

if (contact != null) {

contact.setFirstName(firstName);

contact.setLastName(lastName);

contact.setPhoneNumber(phoneNumber);

contact.setAddress(address);

}

else {

throw new IllegalArgumentException("Must add a contact before you can update it.");

}

One example of technically sound code was the setter methods in the Contact class. I must use the same code that validates our initial object variables to validate the updated object variables. This is to avoid the possibility of initializing a valid object, then later changing the variables outside of their constraints. Shown below is one of the four methods used in the update contact method from the Contact Service class.

public void setFirstName(String value) {

if (value == null || value.length() > 10) {

throw new IllegalArgumentException("First name cannot be null or exceed 10 characters.");

}

else {

firstName = value;

}

}

For this project I implemented many software testing techniques to achieve a coverage percentage of 100%. For the sake of brevity, I will describe the three main techniques used for this project. The first technique is functional testing, which will test the functional requirements of the software to ensure they are met. All projects would require at least some functional testing, as every project is functionally trying to achieve something. The second technique is unit testing, which will test individual units of the software to ensure they are functioning as intended. Unit testing is important for larger projects, to ensure each individual unit behaves correctly before integrating them with other units. The last technique is boundary value testing, which will test the software at the boundaries of input values to identify any defects. Boundary value testing is needed whenever a project is taking input from the user. This is encountered most often with password creation, prompting the user for a password of a minimum size and special character usage. One technique I did not employ during this project is non-functional testing, which will test non-functional requirements of the software such as security, performance, and usability. Non-functional testing is often employed in video games, where the non-functional requirements such as performance and usability are important for the user experience.

Throughout this project, I employed great caution when developing the code for the various classes. This can be seen with the update methods, which not only have a check to see if the object exists, but also the setter methods that validate each of the updateable variables. The best way I was able to avoid bias when developing code was to test often and monitor the coverage percentage. Testing the boundary values for the variables would confirm or deny that our validators are working as intended, and the coverage percentage would reflect how well covered the code is through the JUnit tests. I was very disciplined throughout the project by holding myself to a higher standard of testing, a coverage percentage of 80% was required though I was able to achieve a coverage percentage of 100% for all the tested classes. Cutting corners when developing an application for others can result in real and tangible consequences, it has been seen time and time again from space launch disasters to massive data breaches. To avoid being responsible for any similar mistakes, I intend to remain committed to rigorously testing my projects in the future.

Source:

neeru360. (2023, December 6). *Software testing techniques*. GeeksforGeeks. https://www.geeksforgeeks.org/software-testing-techniques/