# **CS 320 Project 2**

Andrew Emilio DiStefano

SNHU

Professor Omar Toledo Lopez

June 19, 2024

**Summary**

1. **Describe your unit testing approach for each of the three features.**

* **To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.**

My approach aligned completely with the software requirements for each of the three features. In order to make sure that each input string representing the components each feature had the appropriate number or range of characters or digits as per the listed requirements, and that none of them were left null as per the listed requirements, I implemented a ‘validateInput()’ method (as per Professor Omar Toledo Lopez’s recommendations) . This method allowed input to be verified before being saved as the value of a variable.

Additionally, functions were included in order to allow objects to be added, edited, or removed as per the stated requirements for each of the features. For example, the ‘ContactService.java’ file belonging to the ‘ContactService’ feature included a ‘createContact’ function to create classes, an ‘addContact()’ function to add created contact objects to the ‘contacts’ map, a ‘removeContact()’ function to remove contact objects from the map. Additional functions were included in order to allow the first name, last name, phone number and home address of each contact to be updated as per the listed requirements for the ‘ContactService’ feature. Analogous functions were included in the appropriate field for each of the other features in order to allow objects to be added, removed, or have components edited as per the listed requirements.

* **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?**

My writing of each test involved carefully evaluating each line of my code which comprehensively addressed each of the listed requirements for each feature, and my coverage percentage for this project was 100% for almost every file of which the three features consisted. The overall coverage percentage for the entire project was 99%. “As a general rule, a test coverage rate of 80% or above is considered reasonable” (Garcia, 2017, p. 284). This of course means that my JUnit tests were incredibly effective in achieving their purpose of testing the files, functions and programs of which this project consisted.

1. **Describe your experience writing the JUnit tests.**

* **How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.**

I took several measures in order to ensure that my code was technically sound. I created tests to make sure that my objects had initialized correctly as per the listed requirements as shown in the code snippet from my ‘AppointmentTest.java’ file shown below:  
  
@Test // Test if any one of the attributes has a null value.

void testAppointmentNull() {

assertThrows(NullPointerException.class, () -> {

new Appointment(null, tomorrow, "Default description.");

});

assertThrows(IllegalArgumentException.class, () -> {

new Appointment("8675309", null, "Default Description.");

});

assertThrows(NullPointerException.class, () -> {

new Appointment("8675309", tomorrow, null);

});

}

@Test // Test if any one of the attributes does not meet the criteria detailed in our requirements.

void testAppointmentInvalidInput() {

assertThrows(IllegalArgumentException.class, () ->{

new Appointment("12345678901", tomorrow, "Default description");

});

assertThrows(IllegalArgumentException.class, () ->{

new Appointment("8675309", yesterday, "Default description");

});

assertThrows(IllegalArgumentException.class, () ->{

new Appointment("8675309", tomorrow, "Default description default description default description default description default description");

});

}

These tests assure that when an appointment object is initialized, an error will be thrown if the object fails to meet any of the listed requirements. The ‘testAppointmentNull()’ test function assures that an error is thrown if any of the three components (whether ‘String’ or ‘LocalDate’ type) within an appointment object are set to null. The ‘testAppointmentInvalidInput()’ test function assures that an error is thrown if any of the three components (whether ‘String’ or ‘LocalDate’ type) do not meet the listed requirements. If the appointment ID (string) is longer than 10 characters, an error is thrown. If the appointment date (‘DateTime’) is set in the past, an error is thrown. If the appointment description has more than 50 characters, an error is thrown.

I also made sure to add comments explaining the tests and functions and what each of them is meant to do as shown in the example below from my ‘AppointmentTest.java’ file:

@Test // Test that a setter function setting an attribute to null throws an error.

void testSettersToNull() {

Appointment appointment = new Appointment("1234567", tomorrow, "Default description");

assertThrows(IllegalArgumentException.class, () ->{

appointment.setAppointmentDate(null);

});

assertThrows(NullPointerException.class, () ->{

appointment.setAppointmentDescription(null);

});

}

To the right of ‘@Test’ at the top, my comment explains exactly what this test does in sufficient detail. This way the test comment explanations can be compared to the listed requirements and to the code in the files which make up the feature in order to ensure that my tests address the functionality detailed in the requirements, and also to ensure a high level of test coverage.  
 I also made sure to avoid complex code, and I kept redundant code and code repetition to a minimum by adopting the principle of code reuse. One example of where I did this was in my ContactTest.java file, where I created a function to validate input rather than using repetitive code in each function for adding string components to contact objects. Here is my ‘validateInput()’ function:   
  
 private boolean validateInput(String item, int itemLength) {

if (item.length() > itemLength) {

return false;

} else {

return true;

}

}

As can be seen above, the function can be used to keep track of the contents of an input string and to keep track of how many characters make up the input string. This way I can confirm that the input string is not null and that the input string has the required number of characters as per the listed requirements. Here is an example of that function being used within another function rather than repetitive input validation code:  
  
 if (!validateInput(contactID, 10)) {

throw new IllegalArgumentException("Contact ID must contain 10 characters or less.");

} else {

this.contactID = contactID;

}

The code above validates input when a contact is created so that the contact ID meets the requirements that the contact ID shall not be null and shall consist of ten characters or less.

I used the camelcase naming convention to name my functions and the upper camelcase naming convention to name my classes and files. As could be deduced by the lack of unused variable alerts upon running the programs, all variables declared were used. All of my variables were initialized correctly with the appropriate data type.

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

In writing my test files, I used several techniques in order to ensure the efficiency of my code. One example can be found in my use of the variables ‘yesterday’ and ‘tomorrow’ rather than my use in the previous iteration (in Milestone 3) of fixed dates before and after when the expected tests would take place.

Lines 21 and 22:

LocalDate tomorrow = today.plusDays(1);

LocalDate yesterday = today.minusDays(1);

Lines 40 through 54:

@Test // Test if any one of the attributes does not meet the criteria detailed in our requirements.

void testAppointmentInvalidInput() {

assertThrows(IllegalArgumentException.class, () ->{

new Appointment("12345678901", tomorrow, "Default description");

});

assertThrows(IllegalArgumentException.class, () ->{

new Appointment("8675309", yesterday, "Default description");

});

assertThrows(IllegalArgumentException.class, () ->{

new Appointment("8675309", tomorrow, "Default description default description default description default description default description");

});

}

As seen above, the ‘yesterday’ variable was set to the day before the current date and the ‘tomorrow’ variable was set to the day after the current day using the methods included in the ‘LocalDate’ class. This makes it so that the testing can be performed on any day and yield the same results. It also makes it so that ‘today.plusDays(1)’ and ‘today,minusDays(1)’ does not need to be repeated throughout the test file, mitigating against repetitive code and making code more readable to anyone who might view it in the future.

I also used several techniques to ensure that my ContactService and TaskService code was technically sound. As per the guidance I received in the form of the recommendations of Professor Omar Toledo Lopez, I created an input validation function to assure that input strings met the requirements as specified in the listed requirements for each feature as depicted in the example below from the ‘Task.java’ file of the ‘TaskService’ feature:

private boolean validateInput(String item, int itemLength) {

if (item.length() > itemLength) {

return false;

} else {

return true;

}

}

This function simplified the code involved in assuring that input strings met the listed requirements by specifying the logic for input validation once in each file rather than repeating similar logic in each of my other functions involved in input validation. The code can be easily edited according to any possible future changes to the requirements for each input string.

Another way in which I ensured the efficiency of my code (also as per the recommendations of Professor Omar Toledo Lopez) was my use of maps rather than array lists in order to store contact, task, and appointment objects as depicted in the following code:

In ContactService:  
public Map<String, Contact> contacts = new HashMap<String, Contact>();  
In TaskService:  
public Map<String, Task> tasks = new HashMap<String, Task>();  
In AppointmentService:  
public Map<String, Appointment> appointments = new HashMap<String, Appointment>();

The use of maps rather than array lists made my code more efficient since this meant that no ‘for’ loop would be necessary in order to iterate through the array list to find a specific object. Instead, these objects would be called via their corresponding key as is the case with Java maps.

**Reflection**

1. **Testing Techniques**

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

Equivalence partitioning was one testing technique which I employed in this project. Equivalence partitioning involves dividing “all possible input test data…[into]...a set of values…which we assume to be processed in the same way by the SUT…The idea is that testing one representative value within the equivalence class is consider [sic] sufficient” (Garcia, 2017, p. 283). Since the number of test cases is potentially infinite, the designation of equivalence classes according to requirements makes it easier to make sure that at least one representative value from each equivalence class is tested for. This technique was employed in all three of the services which made up this project.

Boundary analysis was another testing practice which I employed in this project. “Faults often appear at the boundary of an equivalence class…the maximum value for a given range, and so on” (Garcia, 2017, p. 284). The boundaries of an equivalence class also include ‘edge cases’ which are scenarios which occur at the extreme ends of an equivalence class. If a function is meant to check if an input number lies between ‘1’ and ‘100’ then a user’s entry of the number ‘99.9999999’ is one edge case and a user’s entry of the number ‘0.0000001’ is another edge case. I employed this technique in all three of the services which made up this project.

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

Error guessing is one technique which I did not use in this project. Looking back, I suppose that there were certain areas where I could have used error guessing. In the ‘AppointmentService’ feature I could have tested for a situation where an appointment is set much further in the past, like 1000 years ago for example. Since the ‘LocalTime’ class seems more than capable of handling such a scenario, I did not perform error guessing for this potential error in my tests. I suppose I also could have performed error guessing for an appointment set 1000 years in the future since such an appointment would obviously have been designated due to error, but would be scheduled successfully without throwing an error by the system.

User acceptance testing is one software testing technique which I did not use in this project. The scenarios specified in this project reflect a system in its earlier stages of development. Since communication with databases and interfaces is not specified in the prompt, user acceptance testing is likely expected to occur later on in the development process. “User Acceptance Testing (UAT), or application testing, is the final stage of any software development or change request lifecycle before go-live. UAT meaning [sic] the final stage of any development process to determine that the software does what it was designed to do in real-world situations” (Elazar, 2018). User acceptance testing involves users testing the system in the real world, so I did not implement this technique in this project.

Exploratory testing is another technique which I did not use in this project. “Exploratory testing…focuses on discovery and relies on the guidance of the individual tester to uncover defects that are not easily covered in the scope of other tests” (Parmar, 2024). “Exploratory testing opens testing to all key stakeholders and not just trained testers…one can capture screenshots, record voice memos and annotate feedback during sessions…[which]...enables faster and more efficient review, by people beyond the traditional software tester” (Parmar, 2024). This type of testing also of course lies outside of the scope of this project.

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

“Error guessing is a type of testing method in which prior experience in testing is used to uncover the defects in software…the tester uses…experience or intuition to gauge the problematic areas of a software application. This technique is helpful in finding bugs which are not easily detected during formal testing methods” (The Economic Times, 2024). While perhaps less comprehensive than other testing methods, error guessing is a great way for developers to test for situations which are harder to reach through other testing methods. My example above of error guessing for the ‘AppointmentService’ which would have involved testing for a date so far in the future that it definitely would have constituted an error by the person setting the date of the appointment comes from my experience using applications like Google Calendar where I have accidentally set an appointment for a few years in the future. This is understandable since it is conceivable that one might set an appointment up to or even more than one year in the future (birthdays, graduations, etc.), but those applications likely have a fail-safe so that appointments set 100 years in the future throw an error. Looking back, my ‘AppointmentService’ feature did not contain any such fail-safe.

One example of my use of equivalence partitioning can be seen in the ‘AppointmentService’ feature. The project’s requirements specified that the appointment date field should not contain a date from the past, and that the date field should not be null. One partition of values consists of dates which occurred before the current date, and testing for the incorrect entry of such dates is implemented by testing for a date which has already occurred. I chose a variable called ‘yesterday’ (to which I set the value of the date before the current date) as the date which I would test in my ‘testSettersInvalid’ test within my ‘AppointmentTest.java’ file. The test can be considered sufficient for the aforementioned partition of values, which would throw a customized error. This practice could be used in a multitude of different situations in order to assure test efficiency while also reasonably assuring that values belonging to a partition of valid or invalid values are effectively being tested for. For example if an input integer must be greater than 100 but less than 500, testing for situations where the input integer is set to 99 or 501 would constitute equivalence partitioning for the (infinite) partition of values below 100 and above 500.

One example of where I performed boundary analysis testing can be found in my ‘ContactTest.java’ test file where I test for situations where input strings contain one character more than the limits described in the requirements (contact ID must contain 10 characters or less, first name must contain 10 characters or less, etc). I also confirmed that input strings with valid numbers of characters as per the requirements resulted in successful creation of contact objects or modification of the modifiable input strings of each contact. Since I also tested for null values for each input string and since a null value would represent an input string with 0 characters, both the upper and lower boundaries of the equivalence classes were tested for. I took similar measures in the ‘TaskTest.java’ file within the ‘TaskService’ feature and in my ‘AppointmentTest.java’ file within the ‘AppointmentService’ feature. “Boundary value analysis is a method, which complements equivalence partitioning by looking at the boundaries of the test input” (Garcia, 2017, p. 248). The use of boundary value analysis to complement equivalence partitioning is of course incredibly useful in a variety of scenarios. For example if input which is to be saved into a ‘LocalDate’ variable must reflect a date within the range of October 31 1999 and January 25 2029, testing each of these dates as input to make sure that such input will not throw an error would allow developers to be confident that correct input could be successfully saved into the ‘LocalDate’ variable. Testing for input reflecting October 30 1999 and testing for input reflecting January 26 2029 would allow developers to be confident that incorrect input would throw an error in such a scenario.

1. **Mindset**

* **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.**

The mindset which I adopted while working on this project involved employing caution and addressing the complexities and interrelationships of the code in a number of important ways. One example which reflects how I was able to achieve both of these goals involves the fact that many of the functions created in my ‘Task.java’ file were used extensively in my ‘TaskService.java’ file as depicted below:

In ‘Task.java’:

// This is our getter function for the task's unique ID.

public String getTaskID() {

return this.taskID;

}

// This is our getter function for the task name.

public String getTaskName() {

return this.taskName;

}

// This is our getter function for the task description.

public String getTaskDescription() {

return this.taskDescription;

}

}

In ‘TaskService.java’:

// This function changes a task name.

public void updateTaskName(String newTaskName, String taskID) {

Task task = new Task("1234567", "Default name", "Default description.");

if (!tasks.containsKey(task.getTaskID())) {

throw new IllegalArgumentException("There is no task with that ID.");

} else {

task.setTaskName(newTaskName);

tasks.put(taskID, task);

}

}

// This function changes a task description.

public void updateTaskDescription(String newTaskDescription, String taskID) {

Task task = new Task("1234567", "Default name", "Default description.");

if (!tasks.containsKey(task.getTaskID())) {

throw new IllegalArgumentException("There is no task with that ID.");

} else {

task.setTaskDescription(newTaskDescription);

tasks.put(taskID, task);

}

}

As can be seen in above, the ‘getTaskID()’, in the ‘getTaskName()’, and in the ‘getTaskDescription’ found within the ‘Task.java’ file are used in the ‘updateTaskName()’ and in the ‘updateTaskDescription()’ functions with in the ‘TaskService.java’ file. This means that any defects within the functions in one file are likely to affect the functions in another. Similar relationships exist between functions in the ‘Appointment.java’ and ‘AppointmentService.java’ files within the ‘AppointmentService’ feature, and in the ‘Contact.java’ and ‘ContactService.java’ files within the ‘ContactService’ feature. In order to address these complexities I would first write the code within the files and corresponding test files which contained the most basic of functions, and then write the code within the files and corresponding test files afterward. I would then check test coverage and make sure that my test files covered as much of the system as possible, achieving an almost 100% coverage rate throughout the project.

* **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 can imagine how bias would be a concern if one were responsible for testing one’s own code. Confirmation bias tends to make it hard for developers to find problems in their own code (Hambing et al, 2019, pp. 36). While having developers test their own code has some advantages such as problems being fixed as soon as they are discovered, since different people think in different ways it may be more beneficial to either have specific professionals responsible for testing or to have different developers test code which was written by a given developer after that developer’s own review of their code. Some companies, whether due to limited personnel or to the establishment of a fixed way of accomplishing tasks which is built into company culture, will still expect developers to be responsible for testing their own code. In such situations, one effective method for mitigating bias in testing one’s own code would be to refrain from performing review and testing immediately after development and allow a determined period of time between software development and the review and testing of the software (perhaps a few days).

One example of when confirmation bias may have affected my own code can be found in this project itself. As I mentioned above regarding my ‘AppointmentTest.java’ file within the ‘AppointmentService’ feature, the idea to perform error guessing for the potential user error of entering a date which exists in the far future (hundreds or thousands of years) would have been a good addition to my ‘Appointment.java’ file and my ‘AppointmentTest.java’ test file. My own confirmation bias kept me from thinking of this at the time of development, but upon review of my code a few days later this potential error made itself clear to me. The review and testing of one’s own code could be done after a determined period of time has passed after the current iteration of a system’s development in order to mitigate confirmation bias.

* **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.**

Discipline and persistence is of the utmost importance in software engineering and testing. If a developer feels a low level of passion for the project or the languages or disciplines involved in the project, it may be tempting to cut corners to finish quickly and proceed with a different project. It is extremely important that this temptation be overcome since haphazard development and testing will inevitably result in an inferior software product with bugs and defects, especially in the early stages of development where the overlooking of such bugs and defects could cause the entire project to incur technical debt which becomes more and more expensive and time-consuming to fix as development proceeds.

Youtube channel Akamai Developer describes technical debt as the idea that “since main code repos are constantly being built upon over time…a shortcut [or ineffective] fix…[implemented]...on day one…[becomes]...an exponentially more expensive fix years later…because that bandaid…[or ineffective]...fix would be so deeply intertwined…into all the codebases and logic” (Akamai Developer, 2021). In other words, mistakes made in early iterations of the development process are caught before they have a chance to propagate throughout the system as it is developed and made more complex.

As a software developer I will make sure to maintain discipline even in situations where I might rather be working on something else. For example, at the beginning of this course I had little interest in Java programming, preferring to refine my skills in Python or C++ which will likely be more applicable to my planned career path as a professional. I did not, however, allow this perspective to hinder my ability to complete the assignments and put 100% effort into every assignment submission. As it turned out this class actually enhanced my interest in Java as a coding language, which likely would not have been the case if I had not put my all into each milestone and project.

**Sources:**

Akamai Developer. (2021, March 5). *CI/CD Explained | How DevOps Use*

*Pipelines for Automation* [Video]. YouTube. <https://www.youtube.com/watch?v=M4CXOocovZ4>

Elazar, E. (2018, April 23). *User Acceptance Testing (UAT) Process Explained.*

Payana. <https://www.panaya.com/blog/testing/what-is-uat-testing>

García, Boni. (2017). *Mastering software testing with JUnit 5*. Packt Publishing.

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019).   
 *Software Testing: An ISTQB-BCS Certified Tester Foundation guide -   
 4th edition*. BCS Learning & Development Limited.

Parmar, D. (2024). *Exploratory Testing*. Atlassian.

<https://www.atlassian.com/continuous-delivery/software-testing/exploratory-testing>

The Economic Times. (2024, June 18). *What is ‘Error Guessing’*.

<https://economictimes.indiatimes.com/definition/error-guessing>