Project 2: Summary and Reflections Report

David S. Harmor

Southern New Hampshire University

CS-320

Professor Tuft

August 12, 2024

As part of the Grand Strand Systems mobile application, my team was assigned to create six components: appointment, contact, and task services modules along with the basic object for each one. In addition, we made test unit cases to ensure that the component classes functioned as intended. To ensure that the components and the unit test cases were accurate, we took the requirement documents and placed those requirements for each section above the method being written as a comment as well as with the unit test case for that method. This helped in avoiding overlooking any requirements that can be caused by going back and forth from the requirements documents to the code. For example in the Appointment Service module starting on line 20 :

// Requirements

// The appointment service shall be able to add appointments with a unique appointment ID

public void addUniqueAppointment(Appointment appt) {

if(!appointments.containsKey(appt.getUniqueID())) // Test if appointment already exists

{

appointments.put(appt.getUniqueID(), appt);

}

else

{

throw new IllegalArgumentException("ID exists or invalid");

}

}

This carried over to the Unit Test for the Appointment Service Unit test case:

// Requirements

// The appointment service shall be able to add appointments with a unique appointment ID.

@DisplayName("Add duplicate Appointments")

@Test

void testAddDuplicateAppt() {

String id = "ID568";

String description = "Valid description";

Calendar date = Calendar.getInstance();

date.set(Calendar.MONTH, 11);

date.set(Calendar.DATE, 05);

date.set(Calendar.YEAR, 2025);

Appointment appt = new Appointment(id,date.getTime(),description);

apptService.addUniqueAppointment(appt);

assertThrows(IllegalArgumentException.class, () -> {

apptService.addUniqueAppointment(appt);});

}

This clearly states that each appointment ID should be unique, this also aided when Static testing was performed as the requirements that need to be met were stated in the comments. In tests where there were multiple possible failures the assertAll function was used, this allowed all the tests to run and not stop if only one case failed. If one case did fail it would note it with the given description and highlight it in red while continuing to test the remaining cases. For example in the AppoinmentTest line 64:

assertAll("Null test Appointment creation",

// Test each field was correctly set

()-> assertThrows(IllegalArgumentException.class, () -> {new Appointment(null,

date.getTime(), description);},"UniqueID null test"),

()-> assertThrows(IllegalArgumentException.class, () -> {new Appointment(id, null,

description);},"Date null test"),

()-> assertThrows(IllegalArgumentException.class, () -> {new Appointment(id,

date.getTime(), null);},"Description null test"));

This shows multiple tests being performed and the exception that should be thrown if the test fails. This also condenses the code for similar tests, each of these tests could be written as a separate case but keeping track of similar test cases would be difficult and prone to errors.

After the Static Testing phase was complete we moved on to Automatic Tests. This is where each method in the three services created was given specific inputs and tested against the known expected outputs in the Unit tests. If the test failed it would be highlighted in red and Green if it passed. Since all of the Unit Test cases in our code passed, they were highlighted in green showing that all the test cases worked as designed. The next phase was to run Code Coverage tests. When running tests written in JUnit, code coverage refers to how much code is exercised. The acceptable range is 80% or higher and if the code being tested receives 80% or higher means that most of the code was tested and attempting to archive a higher percentage has little value. If the code coverage is below 80%, the tests need to be improved as most of the code is not being tested. In my team's code, we achieved 89.9% overall and individually: AppoinmentClass – 86.9%, ContactsClass – 91.8% and TaskClass 90.6%.which we can conclude that the majority of the code has been tested and there is little benefit to try and achieve a higher percentage as the majority of the code has been tested.

**Test Coverage**:

A screenshot of a computer

Description automatically generated

Validating expected outcomes and checking exceptions in the Unit test cases can ensure that the code is technically sound. In the TaskServiceTest, exception handling is evident in the starting line at 57. Here 3 conditions are being tested 1) An invalid ID is entered, 2) An Invalid name is entered, and 3) an invalid description has been entered. So that all the tests are performed each test is an entry in the assertAll command. Without the assertAll command, if an exception is encountered in any one of the 3 tests then the function will exit at the first failure and it will not evaluate the tests that haven’t been reached yet. This would result in fixing the issue identified and rerunning the test as a result another test could fail in the same group. Requiring the process to repeat until all cases passed. Since we are using the assertAll command, similar tests can be grouped saving debugging time as the results for all the tests are performed regardless if there is a failure in one or more.

In order for code to be efficient, it needs to be easy to follow, update, and understand. This code was optimized to avoid redundant operations and ensure clear and concise validations were performed to achieve efficiency. Primarily this was done by using assertAlll, Lambda expressions, and encapsulations. For example in the Contact class, starting at line 16:

public Contact(String contactID, String firstName, String lastName, String phone, String address)

{

setContactID(contactID);

setFirstName(firstName);

setLastName(lastName);

setPhone(phone);

setAddress(address);

}

The member methods were used to set the various attributes. In doing this the various checks if any can be done without duplicating code. For example, setContactID is the method used to set the contactID information, by calling this function in the constructor the tests that need to be performed do not have to be duplicated and make it easier to read.

// ContactID

// Requirements:

// The contact object shall have a required unique contact ID string that cannot be longer than

// 10 characters.

// The contact ID shall not be null and shall not be updatable.

private void setContactID(String contactID)

{

if (contactID == null ||contactID.length() > 10)

{

throw new IllegalArgumentException("Invalid Contact ID");

}

this.contactID = contactID;

}

If the class method was not called an oversite such as forgetting to check for null when rewriting the code segment:

i.e)

if (contactID.length() > 10)

{

throw new IllegalArgumentException("Invalid Contact ID");

}

This oversight can cause odd behavior which may make it difficult to track down. Using the object method setContactID in the constructor reduces code and the possibility of introducing a bug, Another example of efficiency is the use of HashMap in all the Classes, this allows quick lookups based on a contacts ID and can quickly determine if a contact ID already exists. These are just a few items that demonstrate efficiency in coding of this application.

The software techniques I employed for this project are:

**Static testing**: Involves examining code, requirements, and design documents without executing the code. It is a cost-effective method that ensures the software meets the standards and requirements before dynamic testing.

**Automated testing**: Tests individual components using frameworks such as JUnit and verifies that they function correctly. These tests are useful because they are repeatable, yielding the same results every time. Additionally, these tests can be automated and used for regression testing.

The software techniques I did not use for this project are:

**Integration testing**: Tests that the individual modules work together. The different components or modules can be done by other programmers or the same programmer over time, this test verifies that the various components work together and operate successfully.

**System testing**: Analyzes the application as a whole to determine if the specified requirements have been met. During this type of test, the code is executed and tested in various ways within a closed run-time environment.

**Security testing**: Identifies vulnerabilities, threats, and risks in a software application to prevent malicious attacks

The practical uses and implications for each different software development technique that can be used in projects and a given situation:

**Static testing:** This is a way to ensure that the code meets the requirements. It is done manually, usually by another programmer, so a “fresh set of eyes” can spot anything that was overlooked. This can be caused by a programmer being too close to the code for a long time. It also helps with errors that won’t get flagged by the compiler or some software testing tools. Such as using “=’ instead of ‘==’ in an if-then statement, the first is an assignment and the other is an equality test. Both are valid statements, but improper usage can cause unexpected results.

**Automated testing:** This testing method is very useful, it allows for the execution of the code and testing against the expected results. In a large application, this can save a lot of time and money because the tests are repeatable and can be automated. The reports are usually color-coded (Green for pass and Red for failure) so that large amounts of code can be reviewed in a short amount of time in a single report. Also, depending on the size of the project and the number of tests that need to be run, this process can be automated to run off hours and the programmers can review the results and address any issues found the next day.

**Integration testing:** The purpose of this test is to determine if the various modules developed by many, or one programmer interact correctly. In a given module, it can pass all of the tests but fails when communicating with other modules. This is where this test is critical in determining if the system as a whole will work or if the requirements were met.

**System testing:** This test involves testing the entire system from start to end to ensure that it meets the requirements. Usually, this test is performed by testers, but can be automated. Having actual testers, test the application is very beneficial as they often find issues with the software that the development team did not address, or a test case made for it.

**Security testing:** This is a very important aspect and is gaining more attention as we become more reliant on computers. Its purpose is to ensure that the software is secure from malicious attacks, data breaches, and unauthorized access. Also, security tests ensure compliance with relevant security standards and regulations, such as HIPAA.

Throughout this project, I adopted a cautious and detail-oriented approach to software testing considering various scenarios and potential edge cases. Which is crucial for ensuring the quality and reliability of the software being tested. For example, I preceded each test case with the requirements from the documentation of a given requirement in the comments above the test method. This was extremely helpful in ensuring that the test case met all requirements. Additionally, I added test cases that should pass and cases that should fail. If the method failed an exception would occur and the code would handle the exception and not have the application terminate unexpectedly. For example, in the ContactServiceTest class, I had a method called “testAddContactFailure”. In that method, the requirement stated that each contact needs to have a unique ID. So in this test case, I added a contact and then added the same contact again. In this case, adding the contact initially passed, as it should, but when the contact was added again with the same ID an exception was thrown and the test case was handled gracefully.

Having a good understanding of the complexity and interrelationships of the code allows you to test more thoroughly, identify and mitigate risks, and ultimately deliver a better product. In addition to ensuring that features work, it ensures that the whole system functions together and reliably. Although in this project it wasn’t required to have interactions between the Contacts, Tasks, and Appointment services, There was interaction between the base object such as Contact with its service class, ContactService. This interaction played a role in which class would handle a given method or validation. For example, it is not the role of the ContactService to determine what constitutes a valid contact name but it is for the Contact class. Just as it would be in testing if a unique ID for a contact was unique which is a task for the ContactService class. This understanding of what role a given class has is indispensable, as it allows you to create better code and test cases.

The goal of code review is to ensure that the assessment is objective, thorough, and fair by limiting bias. The system I use the most is after having written a fair amount of code I would take a break, work on, or examine another part of the code. Reviewing your code after having just finished it tends for someone to gloss over some sections because they know in their mind that it was “easy” or there can’t be a problem in a certain portion of code. In stepping back for a while you tend to look at it with a “fresh set of eyes” and find simple mistakes For example a common logic error that may occur is using “>” instead of the ‘<’ sign when doing a comparison of greater than or less than. This type of logic error will compile without any issues or warnings as both are valid statements, but by stepping away from that section of code for a while and then reviewing it later you stand a better chance of catching the error than if review it immediately after you write it. Worse the programmer may be overconfident about the code and not review certain sections at all or find a small bug that they consider insignificant and leave it as it is but in taking a fresh look at the code you will most likely catch small errors, syntactical, logic or otherwise and fix it.

Software engineers should be disciplined in their commitment to quality as it creates stable software, reduces risks, and ensures long-term success. Additionally, it is important not to cut corners such as not fully documenting your code, not fully testing your code, creating confusing code, or cutting and pasting source code without understanding how the code works can lead to problems if your code needs to be updated at a later date or by somebody else. For example, in all of the classes we were assigned to write for this project, I copied all the requirements for a given requirement above in the comments preceding the method as part of the documentation for that method. I then verified that everything was covered by making a new complete list of all the requirements and one by one checking off that each one was covered. This greatly assisted in not overlooking a requirement. In avoiding technical debt I plan to use agile methodologies, communicate regularly with the stakeholders, and avoid “quick fixes” that could cost more later in the life cycle of the application instead of fixing it when it is found. Committing to these factors will ensure customer satisfaction, reduce the amount of errors, and you’ll have a good reputation.

References

Junit - Test Framework. Tutorialspoint. (n.d.). https://www.tutorialspoint.com/junit/junit\_test\_framework.htm

*Tutorial Using JUnit.* Southern New Hampshire University. <https://learn.snhu.edu/content/enforced/1615486-CS-320-11262.202451-1/course_documents/CS%20320%20Module%20Four%20Tutorial%20Using%20JUnit.pdf?ou=1615486>