CS 320

Mike Prasad

Nick Hahn

April 15th, 2023

**Summary**

The testing approach for the three features of this application were very similar. In order to ensure that all required features were verified and validated to be functional and meet the requirements of the customer, an elemental approach was employed. Each entity of the project was assessed individually. Following initial coding of the functions, the functional requirements of each element of each entity were checked on a case-by-case basis. As some functions had more stringent requirements than others, there were additional tests performed on those to ensure all criteria were satisfied.

In order to ensure that the approach I used aligned with the requirements, I utilized the list of requirements as an outline for testing. This afforded me a pre-built checklist to go through each function and build the tests piecemeal for each line item. For instance, the requirements for ID elements of each module (length requirement, not a null value, and not updatable) were satisfied by an individual test for each requirement. This, of course, precluded the not updatable requirement; however, since no function was included in the system to allow an update to be performed, no testing was required to ensure this requirement was satisfied.

The overall quality of the unit tests was acceptable and quite thorough. This conclusion can be arrived at upon examination of the results of the testing process: coverage of the programs was above 80% in all cases, all individual tests were valid and passed, and all required functionality was included within the program. With these results achieved, I can say that the experience of writing unit tests has left me with a high opinion of the process itself. Before conducting these tasks, I was rather opposed to the idea of unit testing based on the time required to develop the code for testing and the tedium of the process. However, comparing the process to previous coding endeavors shows a stark decline in debugging time and a much higher confidence in the strength of the final product.

Technical soundness and efficiency of the code were ensured in the program and testing code through a systematic approach. Each function was constructed through a separate function call defined to match requirements. In parallel, the tests to ensure the functions met the requirements were also systematically written and encapsulated in function calls for clean, concise operation.

Examples illustrating this are shown below. As you can see, the requirements for each component parameter of the Appointment object are checked upon creation, and errors are thrown if conditions are not adhered to:

public Appointment(String appointmentID, String description, Date appointmentDate)

{

if ( appointmentID == null || appointmentID.length() > 10) throw new IllegalArgumentException("Invalid appointment ID");

if ( description == null || description.length() > 50) throw new IllegalArgumentException("Invalid description");

if ( appointmentDate == null || appointmentDate.before(new Date())) throw new IllegalArgumentException("Invalid date");

this.appointmentID = appointmentID;

this.description = description;

this.appointmentDate = appointmentDate;

}

Tests for this creation were accomplished similarly. Each criteria for each parameter was tested in sequence. The code below shows the specific example of ensuring that the Appointment ID field does not exceed the maximum character length:

@Test

void testAppointmentIDTooLong()

{

long milliseconds = System.currentTimeMillis();

milliseconds += twoDays;

Date newDate = new Date(milliseconds);

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Appointment("0111111111111", "This is a description", newDate);

});

}

**Reflection**

The software testing techniques employed in this project were chiefly unit testing and acceptance testing.

As the primary purpose of the project, unit testing was the core element I utilized. This approach is a systematic one which goes through each function and element of the program in tandem with code development for those functions and elements, resulting in a thorough assessment of the functionality of the final product. As previously mentioned, this is exemplified in the construction of each function as a unit of code in tandem with test code in the same structural format.

This approach meshes well with acceptance testing, which defines the process of ensuring that all required functions are included in the product code and that the functionality has been tested before declaring that the product is feature-complete. As each element and function is constructed and tested before moving on to the next, the framework for acceptance is inherently built. With a simple review of all features and tests at the conclusion of development ,the product can be validly declared to be within acceptable conditions.

In contrast to the methods employed, the most significant methods omitted from this process were static testing and integration testing. The requirements and business elements of the project were broadly omitted, essentially treating the developed code as an isolated system. In a real environment, such thinking would almost certainly be catastrophic for the success of the project, since any system that cannot integrate with the rest of an enterprise is wasted development time and money. The same can be said for a system that does not integrate with the business model and other components of the overall enterprise; this is something that is avoided through static assessment of the requirements early on in the development cycle. Such applications add more work for the end user than they do help. These quiet failures are doomed to obscurity at best, likely providing unnecessary headaches for the business moving forward.

**Mindset**

As stated frequently throughout this summary, the process utilized in development of the product features could be succinctly defined as “systematic”. Development progressed one function at a time with each element tested in sequence; this afforded a clear development path with straightforward milestones. The declaration and definition of a clear roadmap from start to finish allowed for cognitive focus to be dedicated to coding.

In order to limit bias, the focus was given to the requirements of each function and all possible variations of input as well as all declared criteria. This was done in contrast to a focus on completion of the work; as the individual responsible for the coding as well as the testing, it would have been easy to optimize the tests and code to produce favorable results that may or may not have provided meaningful verification and validation. By focusing on the needs of the program and not the needs of the self, bias was minimized.

As with most endeavors in life, discipline is an essential element for success. The quality of a software product relies on a quality development strategy and adherence to its declarations. Motivation in the small moments and general tedium of development can be challenging; however, the production of clean, quality, tested code is in the best interests of all developers. In many cases, You will be the first person that is called when a piece of software you developed malfunctions or exhibits edge-case bug behavior. By building a system that is easy to review and test in the modular fashion promoted by unit testing, you are saving yourself a lot of headaches in the future.

In all other cases, it will be someone else who needs to reverse-engineer your code and overall system, a position that any developer will eventually find themselves in. By building code that is organized, testable, readable, and understandable, we set a stage that the following developer will be thankful for.