Cameron Winningham

Southern New Hampshire University

2/21/2021

Summary and Reflections Report

I approached unit testing by first creating the properly named test methods for each class to ensure I covered all aspects of the three feature requirements. My tests were directly aligned to the software requirements as I created a total of six test classes for all six classes noted in the design document. The quality of my JUnit tests was ensured by running coverage tests to make sure a majority of the written code was tested.

My experience with JUnit was overall very pleasant, as it was fun to learn how to ensure code quality by writing easily integrated unit tests in Java. My code was determined to be technically sound by first automatically performing static tests of the written code using the Eclipse IDE’s built-in static code analysis. Next, dynamic tests were created and run using JUnit. The tests were all written to ensure the design document’s requirements were met such as writing a specific test to ensure a contact’s first name was not too long:

@Test

void testContactIdTooLong() {

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

new Contact("12345678901", "1", "0", "8885551234", "0");

});

}

Inside the classes themselves, I implemented checks that would throw exceptions if values were null, or too long, to be utilized by the unit tests as the design document required:

public Contact(String id, String first, String last, String phone, String address) {

// id cannot be null or longer than 10 characters:

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

throw new IllegalArgumentException("Invalid id");

}

My code was also very efficient as the tests ran and passed all tests instantly, and test coverage was about 80% for the entire project.

The two main testing techniques I utilized in this project are both static and dynamic testing. Dynamic testing, specifically White Box Testing (WBT), was performed using the technique of unit testing by utilizing JUnit. Static testing, as previously mentioned, was also utilized during the development process automatically by the Eclipse IDE which I was using to develop the code.

White Box Testing was performed as I knew the internal details, such as design and structure, so the tests could be easily and quickly implemented at the same time of application development. One example of static testing that was particularly useful to me was during development of the JUnit tests. I needed to run assertions, and in order to run assertions, the associated JUnit assertion modules need to be imported first in order for them to be used. The automatic static testing performed by the IDE automatically added the required import statements below my other import statements as I was writing the assertions which saved me time by preventing errors.

Software testing techniques which are useful that I did not use for this particular project is integration testing. Although the project was a combination of three modules together, none of the classes communicated with each other, other than the specific class’s service class. Integration testing may be useful for other projects where separate modules are combined together in a way where they communicate with each other, such as if a Task and an Appointment was assigned to a Contact. The Contact class in this case would have associations with the Task and Appointment classes and could encounter integration bugs that would otherwise not exist without the associations.

I believe all projects big and small can always benefit from utilizing static testing. I believe this fact is proven true as all major IDEs have some sort of built-in static code analysis. This form of static testing is seamless, does not interrupt your work, and in most cases it saves you time finding typos, syntax errors, and logic errors you may have missed before you even attempt to build or run the application. For smaller projects, it may not be ideal to spend the time writing JUnit test cases if the code base is small. However, medium to large projects especially projects coded by multiple developers can benefit greatly by writing unit tests. Unit tests ensure that the application behaves as the developer intended. This is especially useful if changes are made and suddenly you get completely unexpected results, a test would fail in this case quickly showing the developer exactly what went wrong. This saves a tremendous amount of time for multi-developer projects since they won’t have to painstakingly search through someone else’s code base to find the mysterious error.

The mindset I adopted when working on this project is a blend of both bias and caution. I can jokingly say my code is flawless the first time I write it, which is my implicit bias, but in reality I often find myself combing through my code to find places where I can improve. One way I can limit bias is by writing comments and tests to give other developers who may work on it, future me included, an idea of what the code is supposed to do. It is always a good idea to have code looked at by a “fresh set of eyes” if you are looking for ways to improve it, or to find a stubborn bug. Unit testing is of course a sure-way to future-proof your code by not only limiting your bias, but also laying on the side of caution.

Lastly, as a future software development professional, having discipline is an absolute must. Being a professional software developer means you are writing code to fulfill your client’s needs, so it is important to not cut any corners that would lessen the quality of the final product. I’m sure nobody would appreciate an auto maker to cut corners by not including airbags or other safety features to speed up the development and manufacturing process while also cutting costs. Doing so may benefit the auto maker saving time and money, but the end user would not be happy at all with a sub-par product with a zero-star crash rating. In the same sense, it is important not to cut any corners when developing software by thoroughly testing it, avoiding technical debt, and ensuring the client gets usable software that does what the client needs. It is also equally important to ensure the software is high-quality which performs well and does not slow down the user’s computer, use up excessive resources, suddenly crash, or have bugs or other quality control issues.