Robert DiMaio

Southern New Hampshire University

CS-320 Software Test, Automation QA

Professor Samuel Haney

April 20, 2025

Last week we were tasked with and completed Project One, which involved creating Java classes and JUnit tests for several services that will be employed in a mobile application we are creating for our employer, Grand Strand Systems, to be presented to a client. This paper will serve as a summary and reflection report analyzing the process used to create this app. The key points in this summary have been addressed in previous submissions and will be referenced accordingly.

My goal with this assignment was to make sure that my classes and testing aligned with the software requirements. There were various requirements given to us that each element of the classes needed to adhere to. Most of the elements had character count requirements, could not be null, and could not be blank. In addition, each class had to have a unique ID field that could not be changed. I kept these requirements in mind when creating the JUnit tests for each corresponding Java class. “There are more ways that each method can fail instead of being successful. I provided each setter method with various options that would result in a failed result: for example, a null result, blank result, or a result that exceeds the character limit for each field.” (DiMaio, 2025)

I made sure that the JUnit tests were effective on the basis of coverage percentage. I ran each test in the Eclipse IDE under “Coverage As” so I could see how much of each class was being tested (by percentage). “When running ContactServiceTest, I can see that the ContactService class is tested at 100% in the main and test folders, and the Contact class is tested at 81.2% in the main folder. When running the TaskServiceTest class under “Coverage As” the TaskService class is tested at 100% and Task is tested at 81.5% in main, and TaskServiceTest is tested at 100% in the test folder. This means that 100% of the ContactService and TaskService branches are being tested, and over 80% of the Contact and Task classes are being tested using the same methodology.” (DiMaio, 2025)

Another aspect of the assignment was ensuring that my code was technically sound. I ensured this by including exceptions to be thrown by each method in my main Java classes. This made testing simpler by checking to see if an exception was thrown in the method being tested, which would show that the method failed. “An example of this is within the Contact class, we can look at the setFirstName method.

public void setFirstName(String firstName) throws Exception {

if (firstName == null || firstName.trim().length() < 1 || firstName.length() > 10) {

throw new Exception("firstName is invalid");

}

this.firstName = firstName;

}

Through the use of if conditional statements, this method will only set the firstName field if we do not encounter any of the conditions that do not satisfy the software requirements, such as being null, blank (less than 1 character after removing whitespace), or if the length is greater than 10.” (DiMaio, 2025)

On top of being technically sound, I also wanted my code to be efficient. One way that I accomplished this was focusing on the setters instead of the constructor. In the past I had not coded this way, but for the purposes of testing, rather than having the constructor method instantiate the object using the “this” keyword, I instead utilized the various setters. This allowed me to include thrown exceptions in the setters, and by calling the setters directly within the constructor, I was able to cut down on repeating myself and including more exceptions than were necessary. “This is evident in the constructor methods for both Contact and Task, as seen below:

public Contact(String id, String firstName, String lastName, String phone, String address) throws Exception {

super();

setId(id);

setFirstName(firstName);

setLastName(lastName);

setPhone(phone);

setAddress(address);

}

public Task(String id, String name, String description) throws Exception {

super();

setId(id);

setName(name);

setDescription(description);

}

“ (DiMaio, 2025)

The software testing techniques we employed in Project One were various white-box testing techniques. “Instead of exercising a component or system to see if it functions correctly, using a specification or model of the system to determine correct functioning, white-box tests focus on ensuring that elements of the structure of a component, sub-system or system are correctly exercised.” (Hambling et al., 2019) As stated above, there are more ways for the tests to fail than there are to be successful. “Most of the guidelines for each module specify each object cannot be null and must be within some sort of bounds of a character limit. Therefore, we have written tests that try to create or modify the object purposefully using faulty logic and “breaking the rules” of each element.” (DiMaio, 2025)

The other significant testing techniques that have been discussed that we did not use are black-box techniques and experience-based techniques. “The category known as ‘black-box’ techniques is so-called because the techniques in it take a view of the system that does not need to know what is going on ‘inside the box’.” (Hambling et al., 2019) There are many similarities between white-box and black-box testing techniques but there are some subtle but key differences. Our focus was on unit testing, using the JUnit library to test our Java classes. “Unit testing is a form of white box testing in which test cases are based on internal structure.” (Jakubiak, 2022) “Experience-based techniques derive test cases from the tester’s experience. This can be a good way to focus on the most important or most common issues that may arise with the code, due to the tester’s previous experience and knowledge working with similar systems, though it is generally viewed as less methodical and thus less effective than black-box or white-box testing and is discouraged from being used alone.” (DiMaio, 2025) Experience-based techniques are sometimes viewed as less thorough than the more methodical white-box and black-box techniques so they were not included, and furthermore, we simply don’t have the experience yet to employ experience-based techniques.

There are various practical uses and implications that come along with the above-discussed testing techniques. It is hard to give specifics on these without knowing what the needs of the software would be. “All three have their own place within testing as a whole and should be used as a complement to each other. Which techniques may be focused on will be dictated by the requirements of the program and specific use cases.” (DiMaio, 2025) The more experience we get with writing software and tests to go along with it, the clearer the strengths and weaknesses of each testing technique will become. Though they each have their own pros and cons, all three will generally be used in tandem in most testing situations to ensure the most coverage possible.

An important yet intangible element of creating software is our mindset while doing so. We must always be mindful of our own mindset while creating software and the associated tests. This includes utilizing caution, assessing our own bias, as well as remaining disciplined. Much more so than in my previous coding experience, my mindset for this project was focused on the tests, and in this way, I followed a test-driven development. Being focused on the tests forced me to be more cautious of what I wanted the code to accomplish as well as all the ways each element could fail. Being mindful of potential failure points and testing for them helps create better code overall. Creating parameterized tests was one way to remain focused on employing caution in our code. These gave us a way to easily view and think about all the ways that methods can fail, according to our software requirements. An example of this is shown below.

*@CsvSource* ({

",'2025-12-12','test description'", // id is null

"' ','2025-12-12','test description'", // id is blank

"123456789123456789','2025-12-12','test description'", // id is too long

"'12345',,'test description'", // date is null

"'12345','2024-12-12','test description'", // date is in the past

"'12345','2025-12-12',", // description is null

"'12345','2025-12-12',' '", // description is blank

"'12345','2025-12-12','testdescriptiontestdescriptiontestdescriptiontestdescriptiontestdescription'", // description is too long

})

*@ParameterizedTest*

void testFailedCreation(String id, LocalDate date, String description) throws Exception {

*assertThatThrownBy*(() -> new Appointment(id, date, description))

.isNotNull();

Though it can be difficult to remain unbiased towards our own code, we must make every attempt to test as objectively as possible. Ideally there would be a different person or team doing the testing to avoid these biases, but in this case it was not possible. We must try to remain as objective as possible and not take testing as an attack on our code or abilities. The goal of testing is to uncover errors and sometimes it can be difficult to find errors in our own work, especially for something we have been working on for so long that we lose sight of how it could be perceived to others. It can sometimes take a fresh set of eyes to uncover some errors that we may have been overlooking for a long time. Perhaps there is a poorly optimized part of code we are writing that we may have told ourselves we would come back to later on, since it works “good enough” for now that we may unconsciously write off and not place importance on.

The above example can also be viewed as what to avoid when trying to remain disciplined. We mustn’t cut corners when writing code, especially when we are beholden to stakeholders other than ourselves. We may be able to get away with doing things in an imperfect fashion for our own personal projects, but this is a dangerous habit to get into. We should be focusing on doing every step as efficiently as possible and adhere to best practices as much as possible. Doing this can help us avoid technical debt and set ourselves up for success in the future.

References:

DiMaio, R. E. (2025). *Module Four Journal* [Unpublished paper]. Computer Science Department, Southern New Hampshire University.

DiMaio, R. E. (2025). *Module Five Journal* [Unpublished paper]. Computer Science Department, Southern New Hampshire University.

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.

Jakubiak, N. (2022, December 6). *JUnit tutorial: Setting up, writing, and running Java unit tests*. Parasoft. https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/