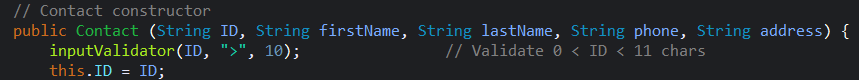
# CS320 Project Two: Summary and Reflections Report

# Daniel Gorelkin - April 12, 2025

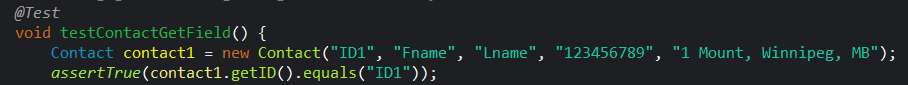
My implementation fully meets the specified requirements for the *Contact*, *ContactService*, *Appointment*, *AppointmentService*, *Task*, and *TaskService* classes, as it fully covers all declared requirements and the test cases for each class. For example, the requirements for the contract class that the contact object shall have a required *firstName*, *lastName*, phone, and address string fields of specific length, and they shall not be null achieved by using the *inputValidator()* method that compares the input length to the preset length and makes sure it is not empty string and it is not null. In the case of correct user input, it allows the constructor to create the object; otherwise, it throws an *IllegalArgumentException* error and does not proceed to execute the code. Such a mechanism is applied to each object field considering the desired field length.



A screen shot of a computer program

AI-generated content may be incorrect.

Similarly, the *ContactTest* class may use J-Unit testing techniques to test all correct and incorrect user input cases by trying to create a class object with several edge case parameters and comparing whether the object was created or the expected exception was thrown, for example, by calling the class constructor with valid arguments and testing the return value through the getters methods, or by trying to instantiate an object with incorrect arguments while expecting to get an exception error back from the constructor:



**A screen shot of a computer code

AI-generated content may be incorrect.**

In such small increments and by validating both the correct and expected and the unpredicted and null arguments input, I made sure my tests were effective and covered all defined requirements while building a comprehensive test coverage of greater than 94% for each of the examined classes ensuring they will be executed as expected while the code runs in live environment.

To ensure that my code is technically sound, I used strict validation, test-driven development, and solid design principles. I ensured that the *ContactId* field is immutable, thereby preserving its uniqueness while using no setter method, and is only accessible through its getter method so data can't be modified accidentally or intentionally. Additionally, I kept the Contact class separated from the service logic, *ContactService*, making it easier to test, maintain, and reuse. Similarly, I used defensive programmingto implement exception handling and prevent unwanted data from being stored in the memory.

To ensure that my code is efficient, I reused existing methods to perform various operations. For example, my *searchContact()* method is used both when adding a new contact by its *contactId* field to the array list and when modifying or deleting an existing contact from it. In this efficient manner, I can both support the requirement for a unique contact ID and rapidly remove a contact from a list if one is found by using the same method.

A screen shot of a computer code

AI-generated content may be incorrect.

Similarly, to prevent redundant data storage and excessive execution, I ensured that my code validated all fields and added the object to memory only if all the arguments met the defined requirements. For example, update the contact only if all fields are within the desired length:

A group of white and green and blue letters

AI-generated content may be incorrect.

Therefore, by ensuring my JUnit tests are effective and thorough, providing great testing coverage while being efficient and technically sound, I can be confident that all requirements will align with best practices and satisfy stakeholder demands.

To make sure that all requirements were met, I employed several software testing techniques, including Black-Box testing, Unit testing, Boundary and Negative testing, and regression testing. Initially, I employed the Black-Box testing methodology, where I conducted requirements elicitation and outlined my test cases, which would later serve as the basis for my code. For example, the requirement that an appointment object will require a unique ID, be in a specific length, shall not be null, and cannot be in the past allows me to construct several test cases for each one of these criteria and create a list of expected input and output values that would be tested later as the code is written. The Black Box testing allows me to plan the external behavior and plan my test cases for the system without knowing the internal code. During the next testing phase, while writing my code, I focused on White Box testing, which involves testing individual components (methods) in isolation, with a particular emphasis on boundary testing. This approach aims to catch common threshold errors and verify expected values. The following Negative Testing technique code test attempts to catch an edge error and tests the system’s behavior under invalid input, specifically when attempting to create an appointment in the past.

A screen shot of a computer code

AI-generated content may be incorrect.

The *assertThrows()* implemented in the JUnit confirms that the error message and exception were raised properly as the appointment due is in the past, which contributes to the system requirements. Ultimately, by incrementally adding and building upon the requirements, I run Regression Testing to ensure that new code changes do not break existing functionality, and if such a scenario is encountered, it could be fixed immediately without adding complex debugging and significant costs to the development process. For instance, by executing a test runner that includes all existing unit tests, we can be assured that all existing tests were passed successfully:

A screen shot of a computer program

AI-generated content may be incorrect.

However, some testing considerations were not employed. Some Integration testing gaps exist in testing the interaction between the components *AppointmentService* and *ContactService*, which should work together and interact with each other. Additionally, Cross-Platform and environment Testing were not conducted. This means that some potential failures and inconsistencies may be present due to date and time handling, as well as time zone differences. Likewise, since the classes had no UI, no user feedback or interface interaction could be tested just yet. To conclude, each one of the tests and testing techniques could be implemented in other software development projects and situations. For example, when working in a development team, Regression Testing can be a powerful tool to consistently verify that each code increment does not break or negatively affect the already completed portion of the application. Similarly, the implementation of Black Box and White Box testing can be executed by two different developers, which can significantly speed up the development process. Likewise, the definition of “done" could be easily defined by setting and reaching a desired JUnit test coverage. The existence of unit testing is a valuable tool for testing the performance of each class individually, enabling testing and maintenance of the code even in large-scale projects.

With that said, working on this project and acting as a software tester, I employed a decent amount of caution by thoroughly testing each method based on the pre-defined test cases and executing all the tests, including testing boundary conditions like min and max-length strings, integer input, null input, duplication, and negative input. For example, the following test case is trying to add a new task with the duplicated *taskId* field, which contradicts the requirements; hence, the *assertThrows()* method in the test should confirm the system will not fail when given an invalid input:

A computer screen shot of text

AI-generated content may be incorrect.

Hence, by understanding the complexity and relationships between different pieces of code and classes, I managed to test not only the main objects such as Contact, Task, and Appointment but also how their service classes instantiate and interact with those objects, for example, how an existing *Task* object is updated and verified.

**A computer screen with colorful text

AI-generated content may be incorrect.**

And despite some tests like integration and UI testing were skipped, it allowed me to prepare myself to a potential errors that would emerge in a later project development phase. Taking the tester role, I tried to limit my developer bias by expecting an alphabetic input instead of a numeric input, such as the *Phone* field. For this reason, and despite the requirements not strictly specifying it, I implemented a regular expression validation method that checks that the entire string consists of digits only and tested this case, expecting to receive an exception error.

A screen shot of a computer code

AI-generated content may be incorrect.A screen shot of a computer screen

AI-generated content may be incorrect.

For this reason, on the software developer side, bias can be a major concern if developers are responsible for testing their own code. As if the attitude of one developer is “string input only”, it does not mean that a user will not input an integer in the string field one day. Therefore, a tester, including the developer themselves, should expect any possible problems with their code and implement preventive measures to prevent the software from crashing as it is deployed and in use. For this particular reason, being disciplined in our commitment to quality as software engineering professionals, cutting corners can lead to deadly consequences, as seen in the use of the Therac-25 radiation therapy machine manufactured by AECL in the 1980s (Leveson, N. G., & Turner, C. S., 1993) where the impacts of the released escapements caused three human to lose their lives during meditational procedure. Therefore, as a practitioner in the field, I will note and declare any potential risks, failures, and possible defects that might arise in my own code or that of others, which I may encounter or cause.

**References:**

Leveson, N. G., & Turner, C. S. (1993). An investigation of the Therac-25 accidents. In *https://www.cs.columbia.edu/*. University of California, Irvine. University of Washington. <https://www.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/therac25.pdf>

*JUNiT - Test Framework*. (n.d.). <https://www.tutorialspoint.com/junit/junit_test_framework.htm>

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.