Grand Strand Systems: Summary and Reflections Report

David Vega

Southern New Hampshire University

CS-320: Software Test Automation and QA

Dr. Karl Lewis

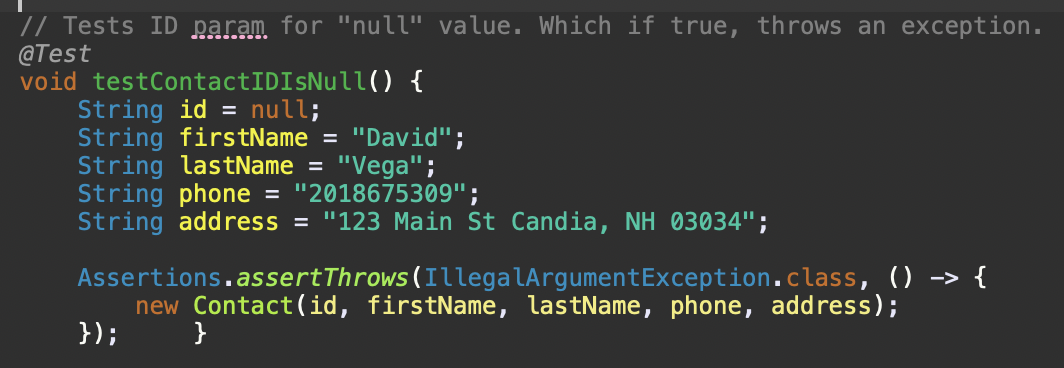
April 16, 2023

# Summary

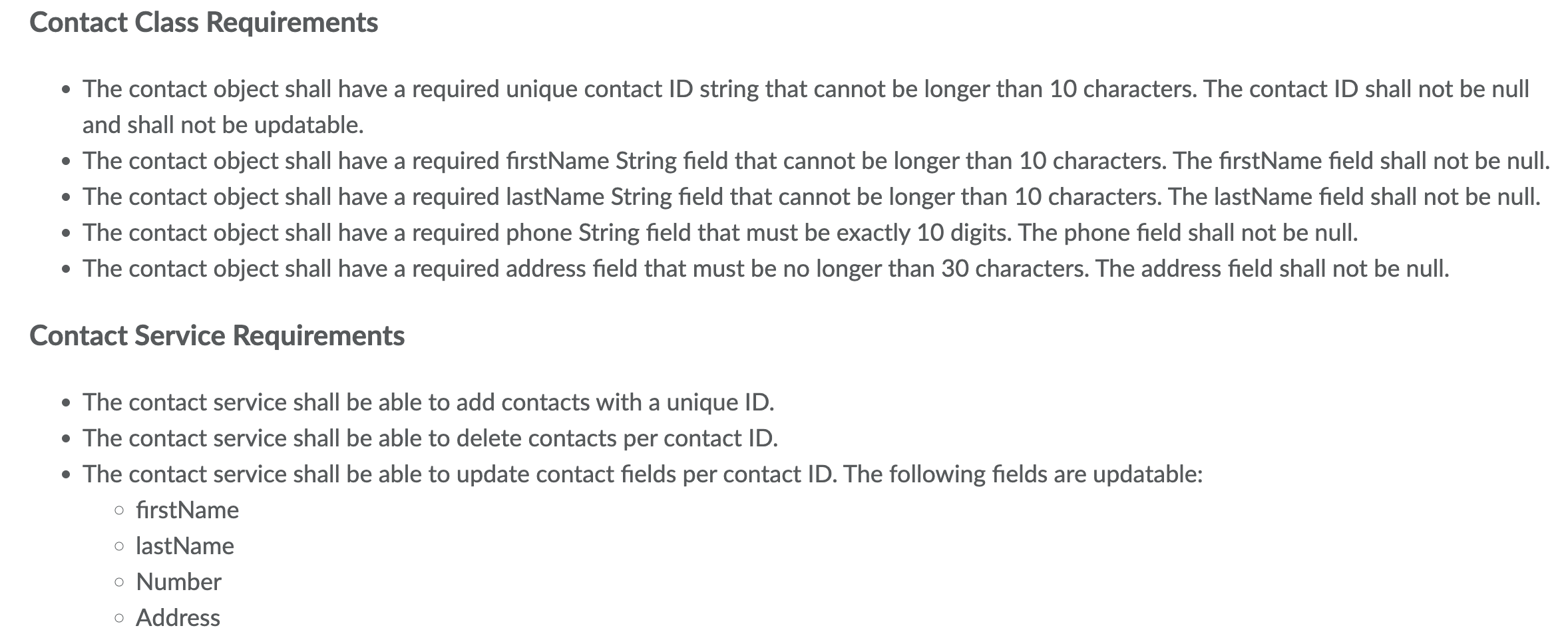
My unit testing approach for all three features were identical in terms of structure. Each feature tested consisted of instantiating an object (i.e. – Contact, Appointment, or Task) and running a series of tests based on the requirements provided by the business. The first test was to test for the valid inputs. For example, in the ContactTest I created a testContact method that hardcoded the valid input values to variables. I then instantiated a new Contact object and passed in these values. I then used an assertTrue method that does exactly what it says, it asserts that the condition is true and, if not, then it throws an AssertionError.



Once I tested the valid inputs, I created a series of tests that tested for invalid inputs. For example, in the Contact feature, the requirements specified that the ContactID could not be “null.” To test this, I reset the hardcoded values and assigned the id variable a “null” value and instantiated a new Contact. I passed in these values as parameters to the object. I wrapped this instance of a new Contact in a lambda expression that would throw an illegal exception if the id input value was “null”. In this case it would throw an exception because I intentionally set the value of the id to “null.”

 }); }

I followed the same pattern to test the other Contact object requirements which stated:



I would say that the quality of my code was sound. Of course, there is always room for improvement as tests are only as good as the requirements provided. This was a much simpler implementation, but other applications may have additional conditions to test with a higher level of complexity. The point of testing isn’t to capture all bugs but to identify and debug the more obvious ones. Over time as the system evolves and the uses for the application become more demanding, testing might reveal additional bugs or possible unforeseen limitations that could be addressed. I strived for 80% coverage on my tests which is considered ideal. Test coverage provides a quantitative measurement of the quality of tests that I am running. ­­­­­­­­­­­

In terms of efficacy, I reviewed my code again to see if there was any code that didn’t need to be there. I made the decision to remove some redundant code that, in my opinion, wasn’t relevant. For example, in the @Test for the testUpdateTask method, I only wanted to test that the method was executing and not throwing any exceptions. I wasn’t really interested in the values that I was updating in the array list so I removed the assertTrue methods that were checking to see if the task id updated was in fact the correct id that was modified. I suppose I could have added a separate test to validate that it was modifying the appropriate task but I didn’t think it was relevant for the purpose of this test. Commenting out the assertTrue methods also increased my code coverage.

Application

Description automatically generated with low confidence

# Reflection

## Techniques Employed

I used a combination of testing techniques of which involved black-box and white-box techniques. An example of black-box testing technique is equivalence partitioning which is meant to divide the input data into partitions from which test cases are derived. An example of equivalence partitioning would be one of the requirements of the Contact object states that the id could not be null and must be no more than ten characters in length. A benefit to this technique is that it helps the developer

A form of white-box testing technique involves static and dynamic testing. Static testing is the process of testing software without executing the code. An example of static testing would be an informal review of a business document that identifies the requirements or in a technical specification document that provides a written solution to the coding requirements (Hambling, et al, 2019).

Dynamic testing identifies defects from a functionality perspective. An example of dynamic testing could be found in the form of field input validation. Test cases are created to “test” an applications behavior under certain conditions (Hambling, et al, 2019).

Text

Description automatically generated

## Other Techniques

Software technique(s) that I did not use was experience-based techniques. This technique relies on the tester’s experience when no specification or “test basis” is provided to generate test cases from (Hambling, et al, 2019). The business provided requirements for each of the objects and services for this project.

## Uses and Implications of Techniques

I could see that the practical uses for equivalence partitioning in other projects would be in data analytics. This is especially true in the field of bioinformatics where mathematics is used to analyze biological data. There are extremely complex data sets involved with bioinformatics that would need to be segmented to extrapolate data to be consumed by another component or API in the system.

Static and dynamic testing techniques can apply to every project. A developer can create test cases based on a business document. Developers can also use statement and decision testing to force an exception through a non-valid input such as “null” or a value greater than the threshold provided by the business requirements. In the case of the Contact object, one of the requirements was that the id was not to be greater than ten characters in length.

## Caution

I think as a developer you want to be able to create the perfect application, free from all defects. While this is an ambitious goal, it sets an expectation that no one can meet. I am reminded that the job of a tester is different than the developer who designed the code. When writing the code, I try to write what was given and not what makes sense to me. As a tester, this is where I can essentially push the limits of the code that was written and identify any defects.

## Bias

For me, the bias I tried to remove was the need for perfection right out of the gate. The feeling that writing code is either black or white, right or wrong, perfect or imperfect. I realize that there might be 101 ways to code something and that I have to be open to refactoring if needed. I also would have to be careful not to write my code with the testing as my guide rather than the provided requirements. The adage “Teach to the test” comes to mind because when I write code, I tend to over think and try to write for the “if” rather than what is in front of me. Sometimes, it’s just easier to write what’s given and revise later.

## Discipline

As a developer, you should always consider the quality of the code you write. The code should be readable and should provide clear intent. The importance of discipline in code serves an important role in testing as well. The more transparent the requirements, the easier it is to develop the code. There shouldn’t be any ambiguity around what is being asked of the developer. The requirements provided in this project were clear, concise and helped to determine how to write the code. For example, one of the requirements specified was that one of the fields was not null and no more than 10 characters in length. I knew that I was writing an if statement that would compare the input to these conditions and if the conditions were true, an exception was thrown, otherwise, set the declared variables equal to the inputs.

# References

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.