# 

# CS 320 Summary and Reflections Report

Ahmet Mujanovic

ahmet.mujanovic@snhu.edu

Southern New Hampshire University

## 

## Summary

**Aligning with Software Requirements**

While I feel the few requirements that were given were tested for thoroughly, I cannot help but feel there are a few particular test cases which I did not account for that may not be quite as obvious to an inexperienced tester such as myself. As far as fulfilling the requirements goes; however, I feel they have been accounted for well. The contact, appointment, and task classes throw appropriate exceptions if they are initialized as null or with an out of bounds field length. The respective test classes for those classes have cases for each of the illegal arguments given in the requirements when initializing an object as well as a test to ensure a legal object can be initialized properly. Additionally, ID fields were designated as final so they could not be updated and a set method was not included for these variables. The appointment class presented a more interesting challenge regarding the implementation of logic that threw an exception if a date for the appointment was created in the past. This requirement was fulfilled by utilizing the Date class from the Java standard library in order to both create this date, as well as perform the necessary check to ensure the date was not created for a date that has already passed. The service class itself contained all the appropriate methods to update, add, and remove objects, and the test class for the service objects tested these methods with multiple test cases, such as ID collisions or missing object IDs. I feel that the service test class is where there may be a test case or two missing. The basic tests seem to account for all the requirements, but adding and updating to a data structure is always more complex than just creating a simple object, so that may be the reason why I feel there may be a few test cases that are missing in the service test class.

**Quality of JUnit Testing**

Based strictly on coverage percentage, the JUnit tests that I had written had 80-100% coverage in all the relevant areas that the particular test was suited for. For example, the task test class, which tested if any of the fields of a Task object were out of bounds when initialized, covered all of the relevant methods and the constructor to ensure there were no exceptions thrown during initialization, while ignoring any code that was not relevant. While some JUnit tests had less coverage, inspecting the coverage closely revealed that relevant areas seemed to be properly accounted for, despite not showing 100%.

**Writing Sound Code**

I believe that the very nature of writing test cases lends itself to ensuring the code is technically sound. Writing exception statements and detailing what the test case does to deal with the anticipated exception is important in writing technically sound code. As far as the code for the test cases themselves go, beginning with viewing the documentation of JUnit was an important first step in writing technically sound test cases. Knowing what methods are available to write an appropriate assertion and finding the correct annotations in the documentation to help ensure the test case is running the way that you want it to is paramount for sound code.

**Writing Efficient Code**

There are a few examples I could point to regarding efficiency, with the first being the lack of constructor overloading. Because the requirements stated no argument can be null, there is no need for multiple constructors to be tested but rather just a single constructor that takes every field as an argument. I feel one of the most efficient parts of the test case code was where I included a “@BeforeEach” annotation in the service test classes. The service class contains methods that add, remove, and update the respective class objects which means the service class itself first needs to be initialized as an object before its methods can be accessed to properly test. The annotation mentioned allows a service object to be created in one place in the code which is then called before each test case, rather than needing to be explicitly initialized in each test case. Additionally, utilizing useful functions present in the standard libraries for many popular programming languages, Java in this particular case, is always helpful in saving time and writing more efficient code. Rather than coding up a custom solution to handling something like the dates for the appointment class and checking to ensure a date is not created in the past, using the date class included with Java trivialized the process and likely led to a more accurate and quality code than something that may have been a less elegant, custom solution.

**Reflection**

**Milestone Technique**

The milestones involving the contact, task, and appointment classes with all of their respective service classes and requirements, were primarily tested with JUnit, which is a type of unit testing. The process involved coding the required functionality for each of the classes, making sure to include exception statements in the event that an out of bounds argument was used. Test cases were then designed and test classes were implemented to ensure that these exceptions were thrown and handled properly. In other words, the safeguards in place to limit the creation of objects or to limit the functionality of methods to certain conditions were triggering for the tests that were written.

Functional testing refers to testing specific, functional features based on a particular specification that was outlined, while non-functional testing refers to the testing of how well the software is working regarding non-functional aspects, such as performance. Unit testing is most often grouped as a subset of functional testing, among many other types, such integration or system testing. However, unit testing can be used to test non-functional requirements as well. Being the most focused and lowest level form of testing, unit testing is often carried out in isolation, where each component or unit is created and tested in small parts. By performing these smaller unit tests, issues in the functionality of the software can be found earlier and easier during the development stage.

**Unused Techniques**

As mentioned previously, functional testing can refer to a multitude of different types of testing that occur at different scopes in a development project. Integration testing, for instance, tests multiple modules of code that were tested individually in the unit testing phase to ensure that these modules function properly when integrated with one another. System testing is yet another focus level above integration testing, where testing targets cases across the system as a whole. The focused and specific integration and unit tests fail to account for end-to-end usage, so the true behavior of a system cannot be determined until this system testing phase occurs.

The test types mentioned previously are all part of a subset of testing known as dynamic testing, which seeks to test the behavior of the code by executing it. Another significant test type that was not used explicitly in these milestones is a form of testing known as static testing. Code is not executed in this testing style but rather other methods of testing occur, such as requirements validation or code inspection. Among the many useful benefits that static testing provides, such as catching errors early before the development phase which can save on time and costs, an example of a particularly useful test is checking the state of dependencies within an application. Many software products rely on third party dependencies which can often go out of date or contain vulnerabilities that may render them unsafe to use or cause an application to behave unexpectedly. Static testing to ensure dependencies are updated or used appropriately is an important step in creating and maintaining a strong foundation for a software application to begin the development phase strong and secure.

**Employing Caution**

Maintaining a level of caution and care when approaching testing code is paramount to achieving accurate and consistent results that reflect the true state of the code being tested. Are the tests designed accurately and do they account for many potential inputs a user might have? Are a series of tests properly considering and utilizing all the code that is interrelated or is there a portion of the code that is not being taken into account? These are some of the questions one might ask themselves when trying to design accurate tests for a codebase. Regarding the project specifically, the service classes for each major package handle the logic which adds, removes, or updates objects into a particular data structure. It was important that the actual creation of the objects was tested first, so that a valid object was known to be able to be successfully created. This is important to handle before testing was carried out on adding objects with the service classes, as an improperly created object may have had an impact on the algorithms used to add or update them.

**Limiting Bias**

The bias that is inherent in developing code as a single develop is a dangerous and destructive inclination that can often result in poor code quality and extended development time. Approaching code development from a provincial mindset can lead to a cascading chain of poor decisions that are further built upon as one views the code from only their perspective in later development. Sometimes, simple fixes for simple problems can often take a significant amount of time to correct because the developer’s mind is fixated on certain portions of the code or design process. It is important to collaborate with others so that they may provide a fresh perspective on the code that one has written, whereupon novel ideas, often ones that serve to improve the quality of code, can be discussed. In the case of this particular project, many of the packages and the classes within had closely related requirements. All service objects were to add, remove, and update their respective class objects. The classes used to create an appointment, task, or contact object also had similar constraints regarding character length or updatability. Thus, it would not be implausible for one to write a series of tests or exception catches for one class incorrectly due to their bias, and subsequently corrupt all other similar classes with poor code by copying the same code used for the first.

**Staying Disciplined**

It can be frustrating when testing for a codebase fails. Accounting for all the ways an input or a function could potentially break a segment of code, particularly when many packages are interrelated and interdependent on one another can be an exhausting process. However, maintaining discipline during these testing phases and avoiding cutting corners will, in the long run, benefit practically every aspect of a software application project. As a developer, code that is sound and quality will serve as an excellent foundation for further development of code that relies on this foundation, leading to smoother development in the future. As a business, a cascading build-up of poorly tested code can cost a significant amount of time and funds to fix, particularly if poorly tested code is already causing problems when it goes live. Testing code thoroughly and catching bugs early on can help save on monetary and time costs by increasing the chances that future issues either do not exist or can be patched promptly.

**References**

García, Boni. (2017). Mastering software testing with JUnit 5. Packt Publishing.

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.

Manico J. & Detlefsen A. (2015). *Iron-clad java : building secure web applications*. Mc Graw

Hill Education.