# **Project Two**

Glenn Bacon

CS-320-T2994

Professor Bryant Moscon

Southern New Hampshire University

**Summary**

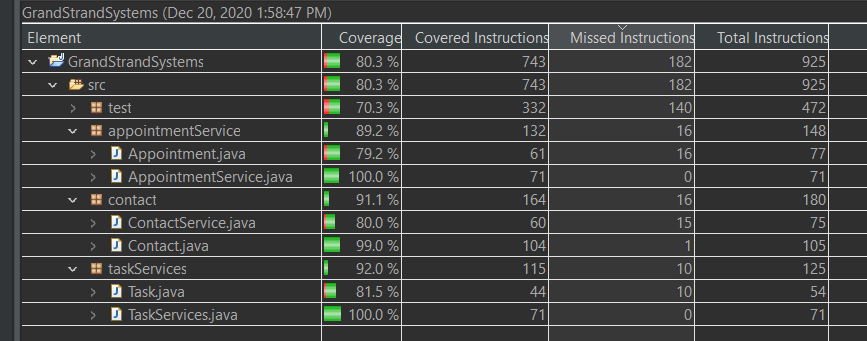
In this project, the objective was to demonstrate mastery in analyzing various approaches to software testing based on requirements and apply appropriate testing strategies to meet those requirements. To meet the objectives a fictitious company named Grand Strand Systems was the client. Grand Strand Systems requires a mobile application with three back-end service features. The features are contact, task, and appointment services.

**Software Requirements**

The Contact Service feature was aligned with software requirements. Testing using JUnit the software requirements of able to add contacts with a unique ID, delete contacts per contact ID, update contact fields per contact ID, and the following variables firstName, lastName, Number, and Address. The Task Service feature initially was notaligned with software requirements. Feedback revealed that TaskService.java needed a lot of work with data structure, methods, and tests. After review of code and the addition of hashmap, methods, and JUnit tests for TaskService.java all software requirements were met. The Appointment Service feature was aligned with software requirements. Testing using JUnit the software requirements of able to add appointments with a unique ID, date, and description. Also, the ability to delete appointments per contact ID and delete appointments.

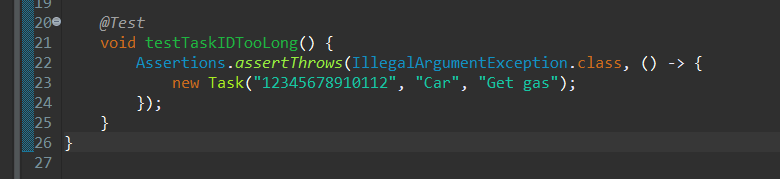
**Effective** **JUnit** **Coverage**

Contact Service coverage JUnit test result was 99% for Contact.java which shows good test coverage for the code in the file. More importantly the requirements are tested, have zero errors and failures. Initially Task Service JUnit coverage was below 80% because of excess lines of print code which brought the percentage down. After commenting out and removing print statements the JUnit coverage jumped above 80%. Task Service met requirements with testing and had zero errors and failures. Appointment Service coverage JUnit test result was 100% for AppointmentService.java which shows great test coverage for the code in the file. Requirements that were tested, have zero errors and failures.



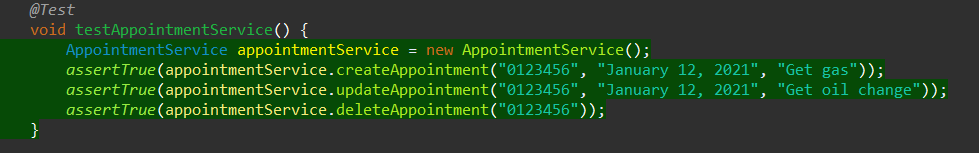
**Technically Sound**

To make sure Contact Service code is sound and to test that Contact.java throws an Illegal Argument Exception for a task with an id that is too long we test for it with JUnit. Using JUnit and Assertions.assertThrows() method we can test that with an id that is more than 10 characters there is an exception thrown. If an exception is thrown the test will pass. If we test with an id that is 10 or less characters, then the test will fail because no exception was thrown which is also what we want to happen as well.



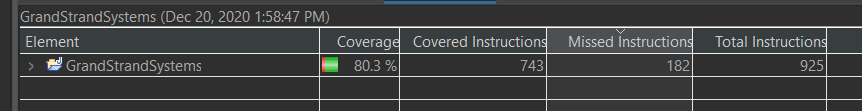
In module four Task Service I created methods and used a hashmap to store in-memory data structures. One of the new testing techniques is to run JUnit test coverage. A high percentage of coverage shows that much of the code is tested and there is very little code that is unnecessary to meet the requirements. For example, I use print statements to also debug and test code but leaving print statements in the source causes coverage percentage to drop drastically because the code is not being unit tested and is unnecessary.

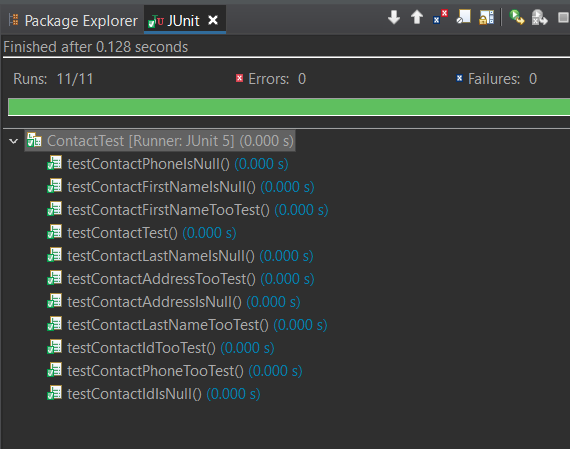
Another technically sound test was Appointment Service, and the other two backend features were tested to verify a true result was returned when each feature was created and deleted or updated. Tests would pass if a true value was returned to the assertTrue() function. If the tests failed, then the class and specific line(s) of code would need to be investigated and fixed.



**Efficient Code**

Test code is efficient below we can see each test method takes very little time to run and as a total only took 0.128 seconds to test. Another way to examine how efficient code is to look at the coverage. Grand Strand Systems complete test coverage is 80.3%. This means that over 80% of the instructions that are written for the company backend features have been tested. It the coverage percentage was much lower than this would reveal that there might be code that is not necessary to meet the requirements which would make the program less efficient.



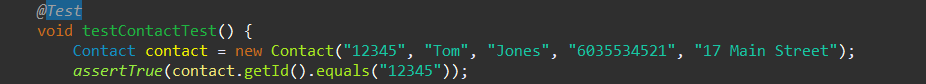


**Reflection**

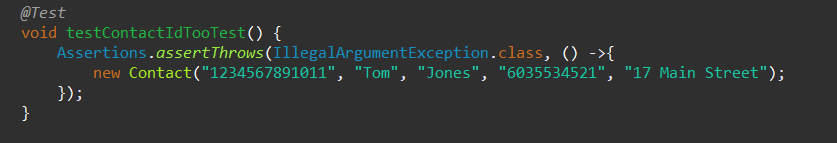
**Testing Techniques**

The testing technique used in this class with JUnit is called dynamic testing. Dynamic testing executes program code with some test data to finds defects, failures, and bugs. Dynamic testing is carried out by executing actual code, and static testing is not. The main difference between static testing and dynamic testing is static techniques can find the cause of a failure rather than the failure itself. With dynamic testing failure is found first then the cause must be found.

In module three Contact Service of the testing techniques used was assertion. One example of using assertion is to check for a true value by using assertTrue(). Below is an example of using assertTrue, the test passes if the id matches the expected id of “12345”.



Another testing technique is to use assertThrows, assertThrows passes a test when the test throws an expected exception due an incorrect argument. The figure below expects an id that is more than 10 characters to throw an exception. If an exception was not thrown then the test does not pass. If the test did not pass, then the tester would need to investigate the Contact class for the defect in the code or missing code to check and throw an exception for the condition.



**Other Software Testing Techniques**

Static testing includes techniques that test software and products without running the code (Hambling, 2015, p.62). One static test technique is to review requirements or specification documents. The review technique is a manual technique that is used to find and remove errors in documents before development. Another static testing technique is known as static analysis. Static analysis uses an automated software tool that enables code to be analyzed for structural defects or systematic programming weaknesses that may lead to defects (Hambling, 2015, p.62).

**Practical Uses and Implications**

It is important to use both static and dynamic testing to find as many defects as possible and failures in both requirements documentation, code and executing program code. An important difference in static testing is the cost to fix a defect early with static testing is much cheaper than fixing a defect or failure during the code development process. Defects that have been missed by static testing could possibly be found in dynamic testing. Using both testing techniques can help reduce the number of costly to fix later.

**Mindset**

**Employ Caution**

The mindset that I adopted working on this project was to be a professional Software Engineer working for a software engineering company that focuses on developing and testing back-end services. Acting as a software tester, I employed **caution** as if the software code if written poorly could cost my job and or the client millions of dollars in damage. It is important to appreciate the complexity and interrelationships of the code that was tested because it can reduce bugs and increase quality of the software. Specifically, if a test were not written or poorly written such as a feature could not update properly then a defect could cause great damage for everyone. Below is a line of test code for testing to see if appointment update works properly. If the line was not written or poorly written, then the feature might have a significant problem.



**Limit Bias**

Bias could be a concern if a developer were responsible for testing their own code because the tests could be manipulated to pass quality tests. Another concern would be if the developer decided some lines that should be tested would not need to be tested because of bias. To limit bias using JUnit tests and coverage to effectively pass/fail quality and efficiency tests.

**Professional Discipline**

It is very important to be disciplined and committed to the quality of software as a Software Engineer. Not cutting corners and writing code that is of great efficiency and quality can save a company millions of dollars in bugs and damage. One way to achieve good quality and efficiency is through testing. Writing good test by using a testing framework like JUnit can help improve efficiency and quality of a program and its features. Below is an example of what can happen to a company for not testing software for a negative sign or negative value.

Due to the pandemic on April 20, 2020 oil prices started falling below $0.00 for the first time in history oil prices were negative. Interactive Brokers Group Inc. software had bugs in it that didn't account for prices to be negative. "Crude was actually around negative $3.70 a barrel when Shah's screen had it at 1 cent. Interactive Brokers never displayed a subzero price to him as oil kept diving to end the day at minus $37.63 a barrel" (Leising, 2020). Because of the bugs in the software and traders could not see actual prices and bought long positions when the price was negative Interactive Brokers promises to rebate from their own funds to their customers. The rebates will cost the company a maximum loss estimate to $109.3 million.

References

Chartered Institute for IT. Retrieved from   
https://app.knovel.com/hotlink/pdf/id:kt00UC2I91/software-testing-an-istqb/static-techniques-test

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 3.1.1.1 Static Techniques and the Test Process (K2). (pp. 62). BCS The

Leising, M. (2020). Oil Crash Busted Broker’s Computers and Inflicted Big Losses. *Bloomberg.Com*, N.PAG.