JUnit Testing: Summary and Reflections Report

Craig O'Loughlin

Computer Science, SNHU

CS320: Software Test & Automation

Matthew McHann

June 19, 2022

JUnit Testing: Summary and Reflections Report

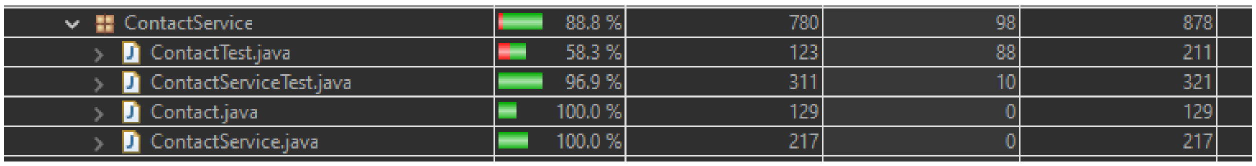
In this course we explored software testing in Java with the help of JUnit, a framework that assists in both writing and executing automated functionality tests of software. We wrote the code under test and then developed and ran test cases to ensure the quality of the code.

Summary

How did I decide if the test cases that were written were appropriate, or that they covered enough functionality of the code to be considered 'good' tests? The approach started with examining the software requirements. The software requirements of any project should define what the expected functionality of the software should be. Using this, we can develop test cases that cover functional requirements even without examining the actual code under test. For example, one of the requirements in our milestones was that the date field of an appointment object cannot be set to a date prior to the current date. This was directly verified using a specific test case. The main outline of the testing script was set in this way, by using the requirements as a framework.

But an approach to comprehensive testing should go further. The requirements describe what the software should do, and testing should verify this. However, testing should go further, and verify what the code *is not* supposed to do. The tester should consider what is known as 'edge cases', areas where the software is given unexpected input parameters, and subsequently how the software should respond. For example, what happens if an appointment object is deleted, but the object ID does not exist? I considered this and other edge cases in the testing script.

The next question is: how do we know when to *stop* testing? JUnit allows us to measure the effectiveness of the testing we have done in a straightforward way, by measuring how many lines of code that have been tested by the script. This is known as code coverage. Arguelles et. al. at Google notes that there is not always a hard target for the percentage of code covered by a given test suite, and that it is only one way to measure testing effectiveness. The coverage requirement may vary from project to project (2020). For this project, we set a target of at least 80%, and that target was reached by my testing, as verified through JUnit (Fig. 1).

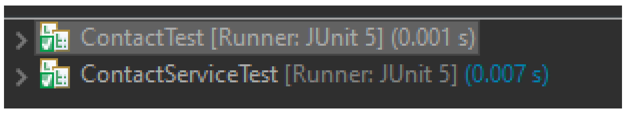


ContactService	88.8 %	780	98	878
ContactTest.java	58.3 %	123	88	211
ContactServiceTest.java	96.9 %	311	10	321
Contact.java	100.0 %	129	0	129
ContactService.java	100.0 %	217	0	217

Figure 1: Code coverage in JUnit 5 in Eclipse

Just as importantly, I ensured that my testing covered every requirement properly and tried to incorporate as many edge cases as made sense.

Finally, in order to determine if my testing was effective, I examined if it was both efficient and technically sound. Unit tests are designed to be small in scope, focusing on a single object or set of methods and verifying functionality (Feathers, 2005). It is sometimes recommended that the entirety of the test script should run in 1 second or less (Fig 2.). JUnit provides an easy way to measure the execution speed, and I was able to ensure that the test execution did not exceed this target.

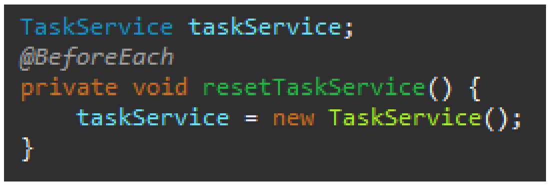


```
ContactTest [Runner: JUnit 5] (0.001 s)
ContactServiceTest [Runner: JUnit 5] (0.007 s)
```

Fig 2: unit test running time (x.xxx s)

I was also able to keep the test script compact both in terms of scope, which did not leave the functionality of the object under test, and in terms of code efficiency, by using the DRY concept (Don’t Repeat Yourself).

I verified that the test script was technically sound by examining the way the tests were built. In JUnit, any annotated test can be executed in any order. For this reason, I ensured that the test cases were written in a way that no prior state of the code under test was assumed. This was done by resetting the test object state before each test by using the @BeforeEach annotation of JUnit 5 (Fig. 3).



```
TaskService taskService;
@BeforeEach
private void resetTaskService() {
    taskService = new TaskService();
}
```

Figure 3: resetting test conditions

Reflection

Testing Techniques

There are many ways to approach software testing. The approach that I took outlined above can be described as a “white box” technique, in which the code under test is visible to the tester (Chacon, 2018). This technique has the advantage of being able to write tests specifically suited to the branches written into the test object and being able to directly measure code coverage.

The SoftwareTestingHelp group also describes other techniques that can be used, including a checklist-based technique which was loosely followed in my approach (2022). In this technique a checklist of testing requirements is built and then written.

Some techniques were not used in my approach but can be equally or more valid depending on the context of the project. There exists, for example, in contrast to white box testing, “black box” testing, in which the contents of the code under test are not known to the tester. This can be an effective approach to checking software output purely from the standpoint of a user. Another technique, “Risk-Based” testing, can be used in cases where the code is too large to measure testing effectiveness with functionality or code coverage alone. In this technique resources are allocated on priority based on overall importance of each test case to the function of the software project. “Exploratory Testing” can be used in cases where the requirements are not well-defined and relies in part of the experience of the tester (Hambling et. al., 2015, p. 127).

Mindset

The mindset of a software tester must be different than that of a software coder, even in the case where they are the same person. The software tester must be completely focused on finding software defects. The tester must also use caution in ensuring that the test cases themselves are not defective. It would be simple to write a test case that passes even without actually testing anything (example: AssertTrue(true)). Writing test cases for similar objects can also be fairly repetitive, as in the case of this project where three similar objects are built and tested, and caution is also needed to stay vigilant in the testing approach. To manage this, I found that spending some time between writing the test cases for each object can allow a mindset ‘reset’ that better allowed me to stay focused.

To limit bias, I felt that a similar distance was necessary between the writing and testing processes. It is easy, especially when testing software that you yourself wrote, to overlook some things that you may assume to be obvious. It may also be easy to ‘go easy’ on your own code. With time between processes, I was able to approach testing fresh and more objectively. This allowed me to write the mundane test cases properly, like ensuring each object field length limit was working properly, instead of glazing over or trying to lump these tests together and possibly missing something.

This leads into the mental discipline needed as a software developer. Rico Mariani at Facebook mentions that the 1,000th unit test is not any less likely to find issues that the 100th and may in fact be *more* likely to (2020). This highlights the importance of remaining disciplined in your approach to software testing. I think the similarities between the objects in the milestones of this project allowed us a chance to test that discipline ourselves in writing similar but different test cases that remain effective.

References

Arguelles, C., Ivankovic, M., Bender, A. (2020, Aug 7). *Code coverage best practices.* Google Testing Blog. https://testing.googleblog.com/2020/08/code-coverage-best-practices.html

Feathers, M. (2005, Sep. 9). *A set of unit testing rules.* Artima. https://www.artima.com/weblogs/viewpost.jsp?thread=126923

Chacon, D. (2018, Aug 2). *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/

SoftwareTestingHelp. (2022, May 4). *Popular software testing techniques with examples.* https://www.softwaretestinghelp.com/software-testing-techniques-2/#Check_List_Based_Testing

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., Williams, P. (2015). *Software testing - An ISTQB-BCS certified tester foundation guide (3rd edition).* BCS Learning & Development.

Mariani, R. (2020, Nov. 19). *Unit tests: Power and caution.* Medium. https://medium.com/swlh/unit-tests-power-and-caution-191e72f47afd