11/20/2019

Team Project

Automated Testing Framework

By Team 6 - TBD

Alan Arsiniega
Cameron Reuschel
Jacob Ballou

Chapter 1 - Checkout or clone project from Repo

In this stage of the project, our team has compiled the OpenMRS software and has run the included snapshot tests. The goal of this stage is to run the testing portion of the software, confirm that it passes its built-in tests so that it will be stable for future custom testing. Below are the steps we went through to prepare, compile, and test the program.

Preparation for the Deployment of OpenMRS

For the compilation and deployment of OpenMRS on Ubuntu, a few programs were required. A version of Java 8 was required for the program to function. For this task, we installed OpenJDK8, as it provided us with the level of functionality we needed for the project.

Git is also required for pulling the repository on github. There are multiple versions and extensions for OpenMRS. We pulled the core version, as this was the only part of OpenMRS 2.3.0 pertinent to our use of the webapp.

For the compilation of OpenMRS, the program required Apache Maven, a software management tool. For this project, the latest version of Maven, 3.6.0, was utilized. This allowed us to successfully build the program and launch the webapp.

Compiling of OpenMRS

The repository was pulled from Github and compiled using Maven. Depending on the machine and how it handled failures in testing, the compilation took between 2 to 20 minutes. Upon the first compilation, Maven initiated tests immediately afterwards, giving off errors within the openmrs-api component of the software. After not finding an immediate solution, it was decided to recompile the software, and retest it after its completion.

Testing of OpenMRS

Several tests were run using the provided test cases within OpenMRS. Through the use of Maven, the tests were iterated for OpenMRS, and each of its components. The tests were broken into 6 categories:

- OpenMRS
- openmrs-tools
- openmrs-test
- openmrs-api
- openmrs-web
- openmrs-webapp

Upon the testing of the most recent release of OpenMRS core, version 2.3.0-alpha, there were some test failures experienced during the run. Maven encountered two failures while testing the openmrs-api, and refused to test further.

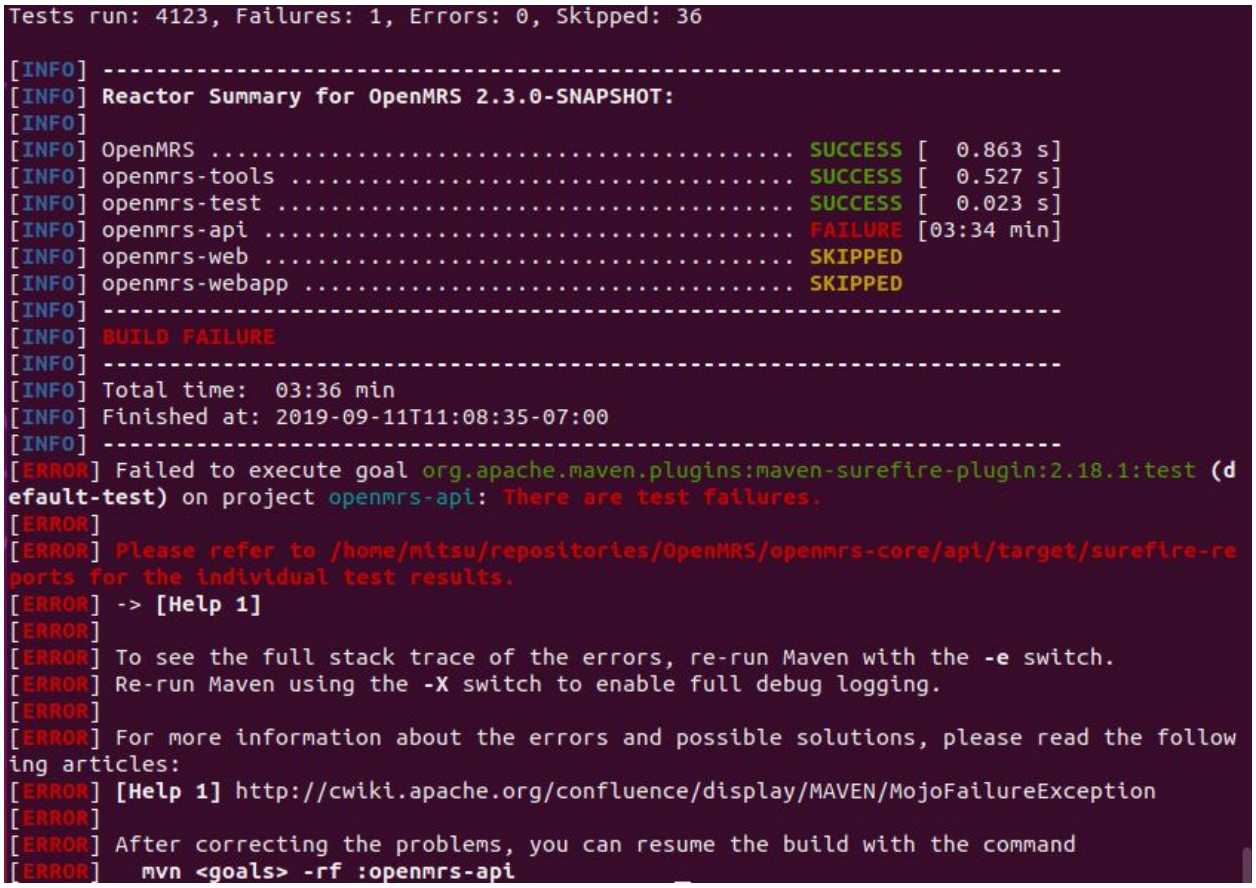


Figure 1: Test Run of OpenMRS core 2.3.0 with Maven. The test was unsuccessful

Resetting to last working version

Every decent version control system should be able to revert back to any point of the software's development life cycle. Because the code has been tracked by Github since 2012 we were able to see that the release being compiled had just been merged to master on August 29, 2019. By issuing the git reset command to the last working version, 2.2.0 from March 21, 2019, we were able to run the tests without so many errors. In fact, instead of skipping entire modules of the OpenMRS project like the web and webapp modules, we were able to successfully execute all tests in 11 minutes as shown below.

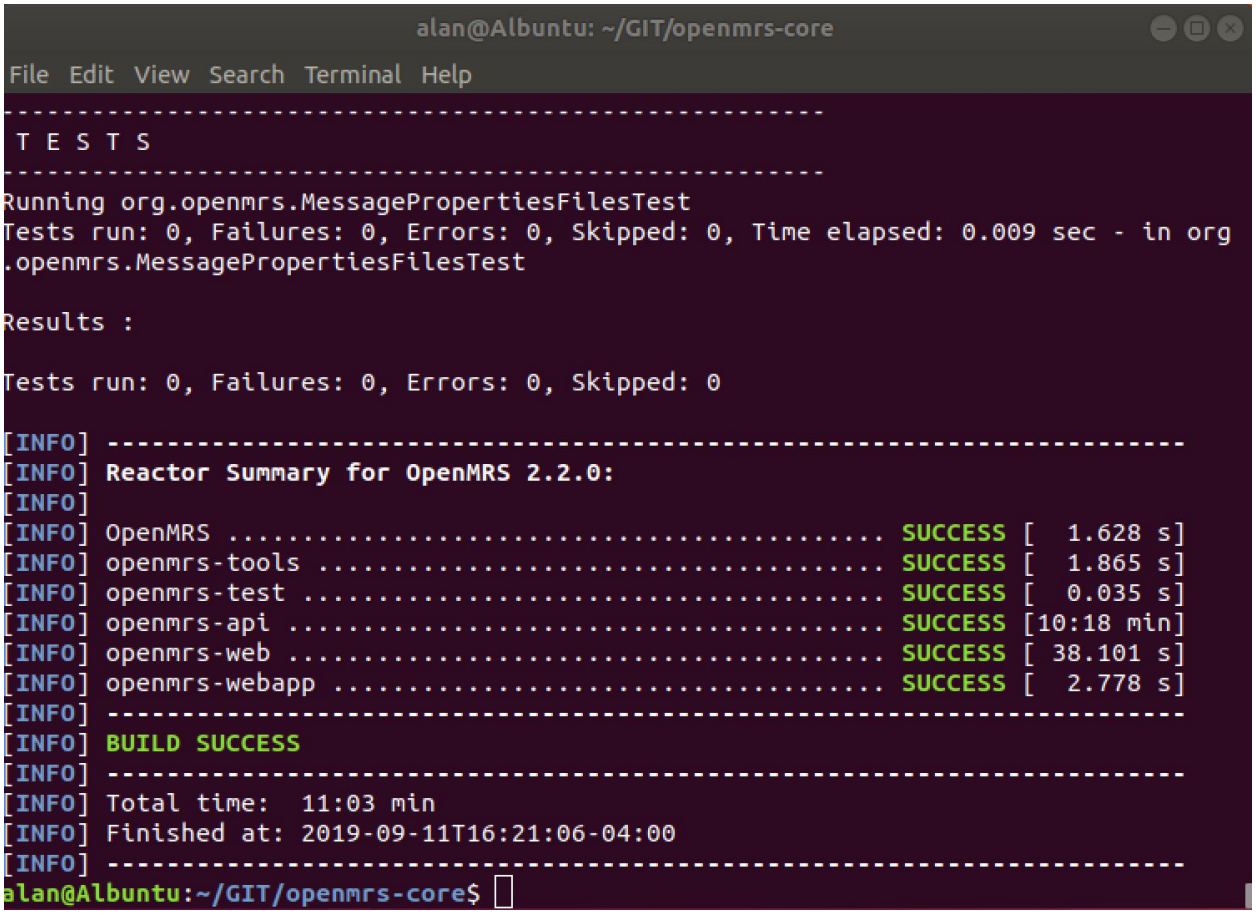


Figure 2: Test Run of OpenMRS core 2.2.0 with Maven. The tests are successful

The installation of OpenMRS and testing has been more-or-less a straightforward process. Because there are currently some issues with the newest version of OpenMRS we have opted to continue working on the last known working version which happens to be 2.2.0. This is also the reason many companies choose not to upgrade to the latest and greatest systems, sometimes in a rush to release newer versions, not all the bugs and errors are really captured and understood.

Chapter 2 - Produce a detailed test plan

In this stage of the project, our team wrote a BASH testing script that takes an input file (input.txt), then enters a while loop. During the loop, it parses the text file line by line, calls the java file and puts the parsed line as the argument during each iteration, calculates and compares the output value with the expected value, and finally prints the answer to the terminal. In later versions, the answer will be added to the output file.

Per test, after the information is read from the file, the executable driver file is compiled and then run with command line arguments specified in the test file. Upon completion of the test, the executable will output data to a file.

Test Recording Procedure

The master script will compare the output file with the oracle corresponding to the same test. A report will be processed to show how many cases failed, and which specific inputs caused the failure.

The team will review the reports and manage the failures. We will decide if the failure was an issue with the testing executable, or the program itself. If it is determined to be an issue with OpenMRS, this will be reported to them.

Hardware and Software Requirements

The software must be able to run with full functionality on Ubuntu 18.04. For our tests, we will be testing the software on an Ubuntu virtual machine. The master script must be able to be called from the top-level directory of our test repository.

Per test, after the information is read from the file, the executable driver file is compiled and then run with command line arguments specified in the test file. Upon completion of the test, the executable will output data to a file.

Test Recording Procedure

The master script will compare the output file with the oracle corresponding to the same test. A report will be processed to show how many cases failed, and which specific inputs caused the failure.

The team will review the reports and manage the failures. We will decide if the failure was an issue with the testing executable, or the program itself. If it is determined to be an issue with OpenMRS, this will be reported to them.

Hardware and Software Requirements

The software must be able to run with full functionality on Ubuntu 18.04. For our tests, we will be testing the software on an Ubuntu virtual machine. The master script must be able to be called from the top-level directory of our test repository.

This will be our plan as we continue to develop our test cases, and then ultimately perform them on OpenMRS. Our goal is to test for the functionality of the API of the system, which is largely connected to the database components of the system. From there, we should be able to discern any faults, if found, with the system's operation.

Chapter 3 - Re-work test plans

The overall framework for the automated testing of OpenMRS's API has been built. It is encompassed mostly through the master script. The script successfully compiles the necessary dependencies, takes in test cases, executes the drivers with command line arguments, takes the output, and compares it with the corresponding oracle to create a comprehensive test report.

The majority of the script, and testing is complete. Apart from the addition of more test cases, the testing framework must adapt to the dependencies being placed in the project folder, as well as moving the drivers to the appropriate location.

Framework Architecture

The script runs through several phases to prepare for the testing, initialize testing, and produce a report. This process goes as follows:

Script Structure:

Preparation: The script compiles the OpenMRS with Maven and prepares the dependencies and drivers for the test.

Load: The script takes in a list of test cases available from testing from the *testCases* directory. These will be all the tests that the framework will loop through to complete its complete API testing.

Selection: The script selects an individual test, reads the test name, what it's testing, the information needed to locate and call the oracle and driver, and retrieves the input needed to pass to the driver via command-line arguments.

Testing: The driver (already compiled) is found within the *testCaseExecutables* directory and is called using the inputs from the test file. The dependencies are located within the *src* directory and are imported by the driver, as well as the class being tested. The output is returned from the driver and stored by the script.

Writeback: The driver writes back the calculated output from the driver to an output file in the *temp* directory. The file is named based on the test case name.

Comparison: From the home directory, the script locates the output file from *temp* and the oracle from *oracles* and compares the files directly. The way the oracle and output are formatted allows for the file contents to be easily compared. The success status of the test is then written to the "testReport.html" in the *reports* directory.

Repetition: The steps from **Load** to **Comparison** are repeated for all available tests until a complete report is developed.

Production: The report is finalized by the script, with correct HTML formatting and is then displayed to the user's web browser.

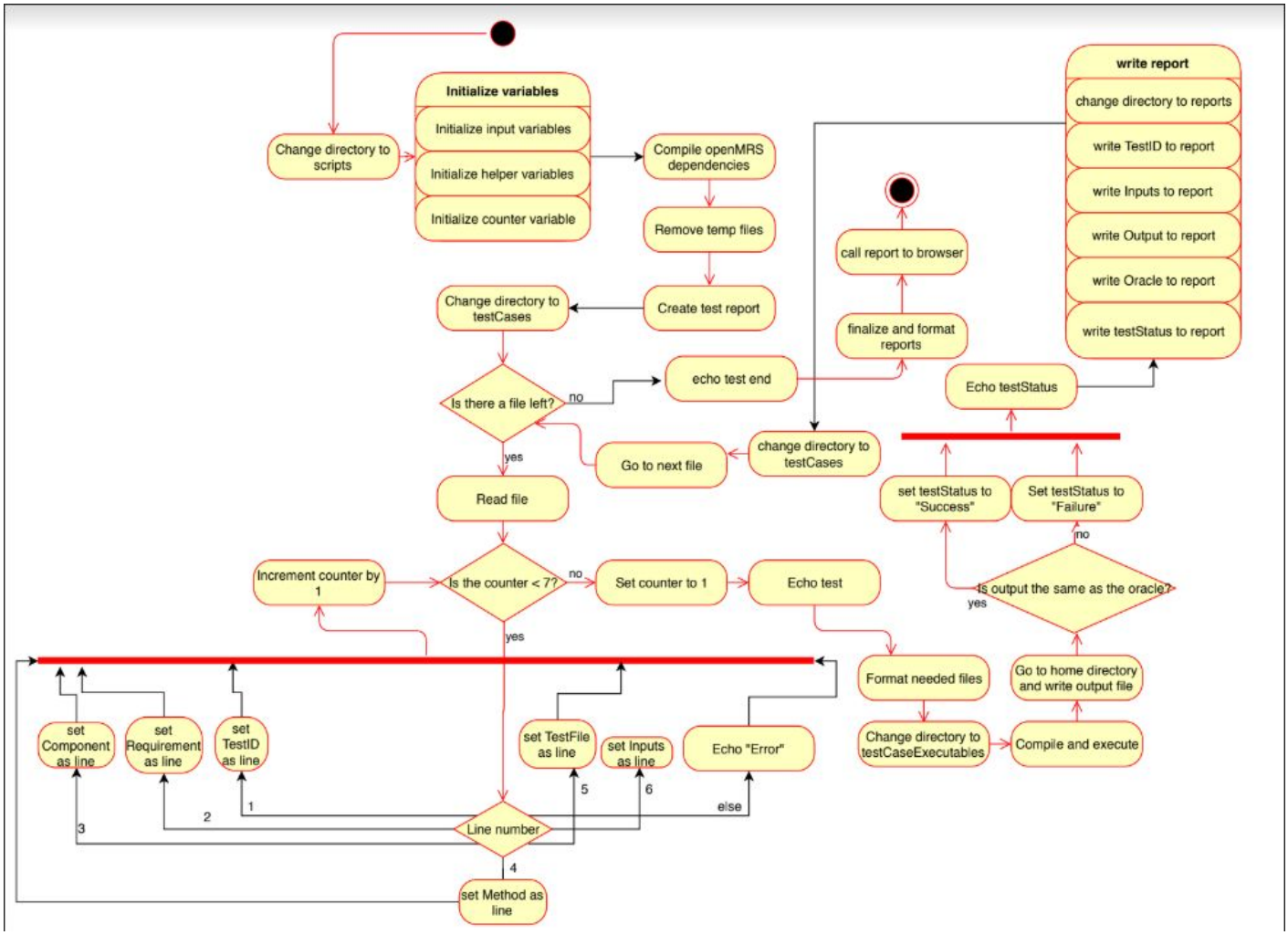
Test Cases
Example:

TEST BEGIN
==========
Test Being Run:
DoubleRangeTest1
Testing Range Function for within bounds
OpenMRS Util Component
contains()
DoubleRangeTest
1000.02 9999.1 3000
Success
==========

The script compares the oracle with the resulting output of the function being tested. In these five cases that would be the boolean from the contains() method. A Success means the oracle matched the boolean in our oracle, a Failure would mean the booleans didn't match. This will be expanded to test other methods with oracles likely being something other than a boolean for some variety in our tests.

How to Test
Pull the code from Github into an empty directory. From the home directory *TestAutomation/scripts*, the command './runAllTests.sh' will initialize the test. From that point on, the test will be fully automated, from preparation to completion. The conclusion of the test will result in the test report being opened in a web browser. This report will list each test, what is being tested, the input, and if test produced the correct output.

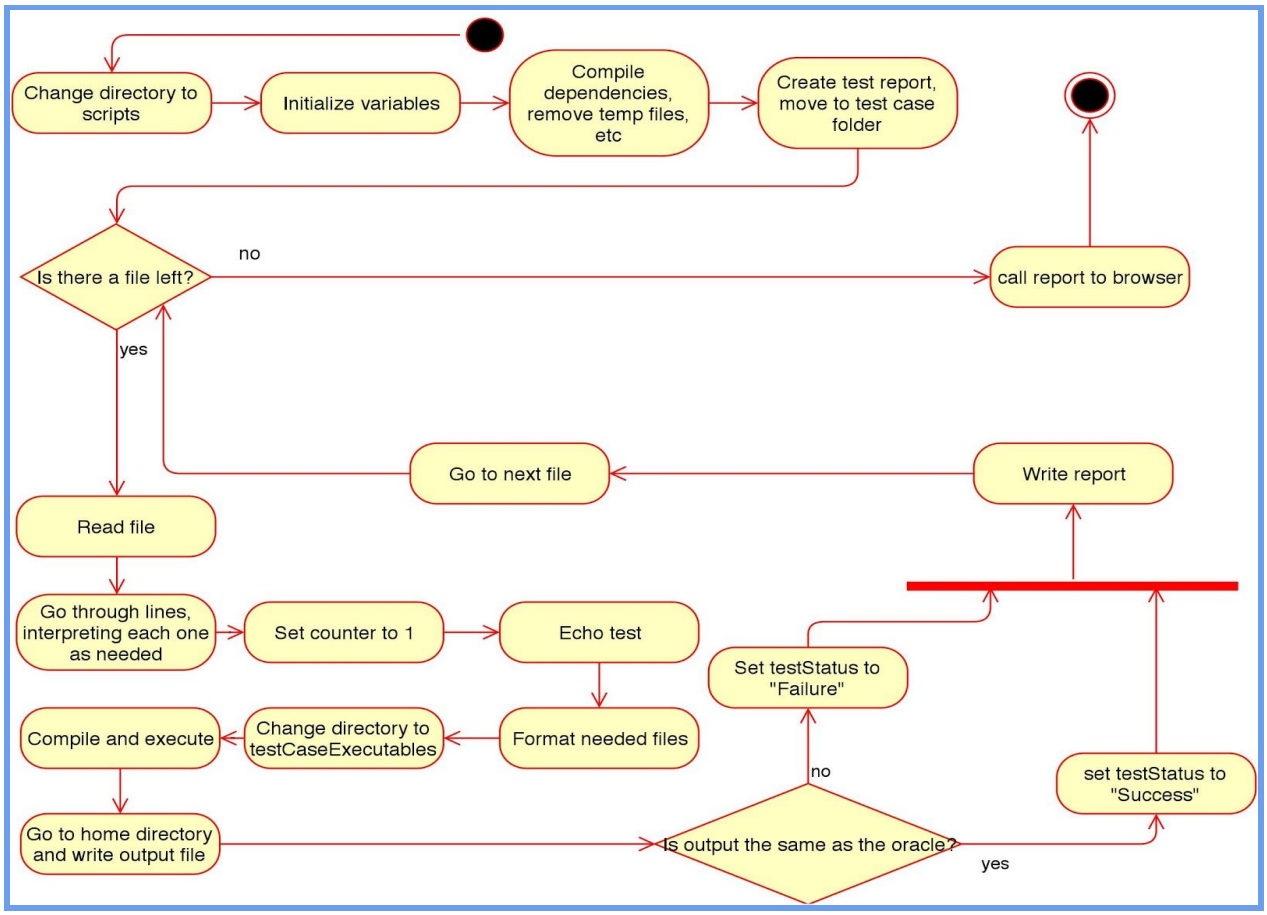
Initial Architecture



Chapter 4 - Complete the design and implementation

Final Architecture

The design is now refactored to allow for test cases to be ran in any sequential order, 1 thru 5, 10 thru 25, it's all possible by naming all the test cases with the same name and just adding the sequential number at the end thanks for Bash scripting



Final Report

We have decided to not include the inputs as are read by the script and passed to the program but to show a visual representation that helps us understand what the test driver is actually doing with the inputs. We've added the actual Java method being called and cleaned up the requirements to accurately reflect what the method should be doing.

Here are columns 1 thru 4 with accurate requirements for the methods being tested:

Test Name	Requirement	Component	Method
test1	The greater than method will return all results greater than the given value	org.openmrs.logic.result	Result gt(Integer value)
test2	The greater than method will return all results greater than the given value	org.openmrs.logic.result	Result gt(Integer value)
test3	The greater than method will return all results greater than the given value	org.openmrs.logic.result	Result gt(Integer value)
test4	The greater than method will return all results greater than the given value	org.openmrs.logic.result	Result gt(Integer value)
test5	The greater than method will return all results greater than the given value	org.openmrs.logic.result	Result gt(Integer value)

And here are columns 5 thru 8 with an input representation to help explain the tests:

Input Representation	Output	Oracle	Result
[10].gt(5)	10.0	10.0	**Pass**
[-10].gt(-5)	empty	empty	**Pass**
[0].gt(0)	empty	empty	**Pass**
[-10].gt(-15)	-10.0	-10.0	**Pass**
[1].gt(-1)	1.0	1.0	**Pass**

Chapter 5 - Design and Inject 5 faults

As of now we have 25 tests. The first 10 (test1-test10) deal with the org.openmrs.logic.result component. The next 15 (test11-test25) deal with the org.openmrs.util component. However, only test11-test20 deal with the contains method from the DoubleRange class and only ***test11-test15*** deal with inclusivity and exclusivity.

What we are testing is simple, we create a DoubleRange object with the first two input arguments and then call the contains method with the third input argument. This can be seen in our testCaseExecutables/DoubleRangeContainsTest.java file.

The requirement is: Return true if double is in range and false otherwise, **low end is closed or inclusive**, **high end is open or exclusive.** A way to represent this mathematically is with a closed low end [and an open high end)

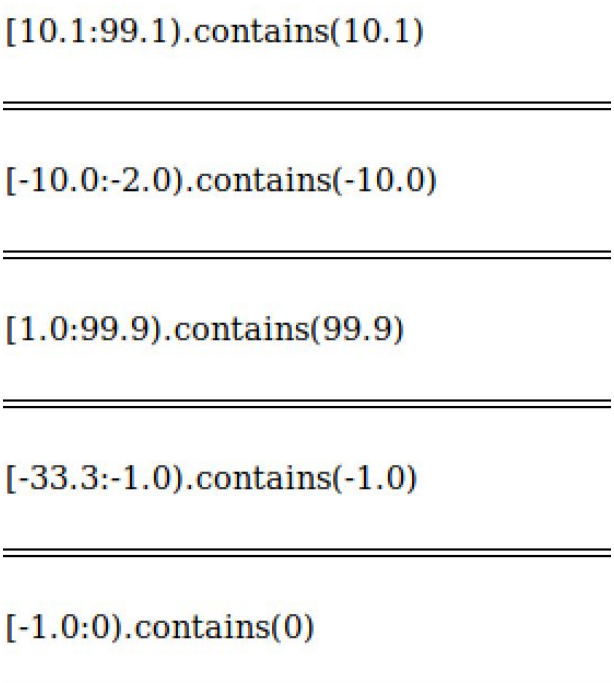


Fig 1.1 Cases test11-test15 all test closed low end and open high end

The code without changes will always say a DoubleRange of say [10.1:99.1) will contain the closed-inclusive low end of 10.1 but will not contain the open-exclusive high end of 99.1. Some classes like the Apache common similarly-named classed treat both ends as inclusive and were thus not used.

[10.1:99.1).contains(10.1)	true	true	**Pass**
[-10.0:-2.0).contains(-10.0)	true	true	**Pass**
[1.0:99.9).contains(99.9)	false	false	**Pass**
[-33.3:-1.0).contains(-1.0)	false	false	**Pass**
[-1.0:0).contains(0)	false	false	**Pass**

Fig 1.2 Cases test11-test15 all PASS low and high end inclusivity and exclusivity, respectively

Changing the Code and Recompiling

The file we will need to change to *inject faults* can be found here from the root of the project:

TestAutomation/openmrs-core/api/src/main/java/org/openmrs/util/DoubleRange.java

There are two instance boolean variables named closedLow and closedHigh each set to true and false, respectively.

```java
/**
 * This Source Code Form is subject to the terms of the Mozilla Public License,
 * v. 2.0. If a copy of the MPL was not distributed with this file, You can
 * obtain one at http://mozilla.org/MPL/2.0/. OpenMRS is also distributed under
 * the terms of the Healthcare Disclaimer located at http://openmrs.org/license.
 *
 * Copyright (C) OpenMRS Inc. OpenMRS is a registered trademark and the OpenMRS
 * graphic logo is a trademark of OpenMRS Inc.
 */
package org.openmrs.util;

import org.apache.commons.lang3.builder.HashCodeBuilder;

/**
 * Represents a bounded or unbounded numeric range. By default the range is closed (ake inclusive)
 * on the low end and open (aka exclusive) on the high end: mathematically "[low, high)". (I'm not
 * using the similarly-named class from Apache commons because it doesn't implement comparable, and
 * because it only allows inclusive bounds.)
 */
public class DoubleRange implements Comparable<DoubleRange> {

    private Double low;

    private Double high;

    private boolean closedLow = true; //TODO: add setters and getters for these

    private boolean closedHigh = false;
```

Fig1.3 To inject faults to the code we simply need to flip the true and false booleans

Once we've flipped the true and false in the source code we need to recompile the package by going to Team6/TestAutomation/openmrs-core/api
And issue commands:
Team6/TestAutomation/openmrs-core/api $ mvn clean
Team6/TestAutomation/openmrs-core/api $ mvn compile

We run the tests again by calling our script:
Team6/TestAutomation $ ./scripts/runAllTests.sh

And the outputs are now:

[10.1:99.1).contains(10.1)	false	true	**Failure**
[-10.0:-2.0).contains(-10.0)	false	true	**Failure**
[1.0:99.9).contains(99.9)	true	false	**Failure**
[-33.3:-1.0).contains(-1.0)	true	false	**Failure**
[-1.0:0).contains(0)	true	false	**Failure**

Fig 1.4 Just switching where the ends are closed with the boolean variables is enough to inject faults

Chapter 6 - Overall Experiences

One of the overall lessons was on preparedness and thinking ahead of what will be the challenges. When you're starting off it's easy to just zero in on the next deliverable or the task at hand. However, this proves to be a really bad strategy as you may box yourself into a corner by doing big term projects in this way. I tried to think forward on a lot of issues but it's impossible to think of all the issues.

We had some problems dividing up the work, we also had a few problems just communicating. In this day in age with collaboration tools like Slack and Github you would think this is made easier. My personal experience is that this wasn't the case and often times things were misinterpreted and tempers flared. I think as the more senior person on the team I should have handled things better. Unfortunately, working for the University as I do currently is a whole personal life challenge unto itself. Add all the homework and readings and the exam, the course work is quite a lot for someone with a full-time job and many other responsibilities.

Oftentimes I would feel that I should be able to take a break and let others carry some of the weight but when someone is having a lot of issues just getting started you have to make a choice. Spend the time catching them up after every deliverable or just do it with just the help of one other. After one deliverable I was kind of done spending the energy to catch teammates up after all, in the real world it's very easy to get left behind, especially in programming. In my full-time job I feel there are many that have been left behind by the times and just do the minimum to get by. That's unfortunate because this can be such an exciting field and full of innovation. I will not be part of teams that aren't forward thinking, I've had to reconsider a lot of what I want to do with my career because of experiences working with others and being put in difficult positions.

Overall, the work got done but not without sacrifices. This is a tough course. I am very surprised we covered as much as we did from the book AND had this huge project to do. I am a few homework's behind but I will catch up soon. Being able to see the tests fail after the fault injections was rewarding in itself and a revelation of how powerful programming these unit tests can be in software engineering.