CSCI5308 – Fall 2021 – Assignment 1

Topics Covered: Test-Driven Development, Continuous Integration / Continuous Deployment.

Assessment Objective

The purpose of this assignment is to assess your understanding of the concepts taught in the Test-Driven Development and Continuous Integration and Deployment learning modules and lectures.

We can assess these modules easily by:

- Asking you to design and program classes using test-driven development and JUnit
- Asking you to implement CI/CD in your individual assignment git repositories to build your project and execute your tests, uploading the results of your individual tests to Gitlab.

These two tasks combined fully test every aspect of what you have learned so far.

Instructions

This assignment will significantly test your ability to follow instructions as well. All good programmers must be extremely diligent in reading and understanding the requirements of the work they've been asked to do. Do the following:

1. Read through all of these instructions before doing them, then read the marking rubric to understand how you will be assessed. Then proceed through the instructions.
2. Log in to git.cs.dal.ca, and locate the individual assignment repository that was automatically created for you for this course. These repositories have the following structure on the faculty gitlab server: https://git.cs.dal.ca/courses/2021-Fall/CSCI5308/assignments/<some_combination_of_characters_like_your_name>. You might be getting an invitation email for the same.
3. Clone your individual repository to your local computer.
4. In your individual repository on your local computer create a folder labeled **A1**
5. This is the most common step students ignore, they name their directory "assignment1", "assignment_1".
    a. In your **A1** directory, create a gradle or maven project that uses Junit to build and execute unit tests defined in the project. If you don't know how to do this, well here's the demo you were shown in the CI/CD lab: (https://git.cs.dal.ca/sukhchand1/java-gradle-ci-sample). You can base your project on this so long as you credit the project in your PDFdocument.
    b. Another common mistake students make when I give sample repositories ininstructional documents is to confuse these repos with their individual

assignment repositories. Make sure you do your work in the right place!

6. Create a document that includes the following:
    a. Your name, CSID and Banner #
    b. A link to your individual assignment repository on Gitlab
    c. If necessary, credit for any project resources you have used as starting point for your assignment
    d. Number of Tasks are listed below. There must be commit after each step. Proper comments with naming convention must be followed while making comments.
    e. In this document, write down the description of what you have implemented for each task. Here you should not write any code, just give the brief explanation of what you have done in order to complete that task.
7. Submit your document in **PDF** format to Brightspace in the Assignment 1 folder on Brightspace. This PDF document and your individual assignment repository commit times must be submitted before the due date defined for assignment.
8. Push your code to the **master** branch of your individual assignment repository.
9. Implement CI pipeline in which there will be only two phases: Build and Test as shown in example listed above.
10. Configure your individual assignment repository, through the use of a .gitlab-ci.yml file, to build and execute your unit test results, including uploading the results of individual test executions to gitlab. You can use the project linked above as an exemplar for how to do this. If you do so, you must credit the project in your PDF document.
11. Push all your changes and ensure your individual assignment repository is building your project and executing the tests written to develop the classes. Your TAs will verify this.

Tasks to be done

Step 1 - Create a calculator class from a test perspective

Create a calculator class from a test perspective. Create a test class i.e. create CalculatorTest class and add a not null test to assert the Calculator is not null (means Calculator class exist).

Step 2 - Provide an add method which accepts two integers

Provide an add method which accepts two integers and returns the result of adding both together. You should cover two test scenarios: One for both negative numbers and second for both positive numbers.

Step 3 - Provide subtract, multiply and divide methods which accepts two integers

Provide subtract, multiply and divide methods which accepts two integers (first - second), and returns the results for respective operations. In case of subract(),you should cover two test scenarios: One for both negative numbers and second for both positive numbers.

Step 4: FizzBuzz Problem

Search FizzBuss Problem and write down the fizzbuzz( ) method in Calculator class and write the suitable test cases for its acceptance criteria. Note: All test cases must be covered.

Step 5: Add another variant of add() method i.e addstrings() method to take two Strings which can contain decimals upto two places and returns a String after addition operation.

Step 6: Based on step 5, addstrings() method should throw an exception when Strings with more than two decimal values are passed in as either argument. Try writing the test which expects an exception to be thrown, run the test and see it fail, and then implement the new code to see the test pass.

Step 7: Repeat step 5 and 6 for subtract, divide and multiply functionality.

Step 8: Throw exception when second value is zero in case of division

We'll need to use TDD to throw an exception if second value is zero as you'll get a divide by zero exception. You'll need to check the values passed in and throw an exception if they fail to meet the requirements. Try writing the test to expect the exception to be thrown, run the test and see it fail, and then implement the new code to see the test pass.

Step 9 : Create a Validation class

Create a class named Validation using TDD and also write the Not Null Test for the same. Try writing the test for the Validation class, run the test and see it fail, and then implement the new code to see the test pass. [Hint: Same as you have done in Step 1]

Step 10: Create a validate method

Follow the Acceptance Criteria(AC) below to implement a validate method in the Validation class using TDD.

Create a validate method which returns a List<String> and accepts a vargs array of Strings .

Signature will be public List<String> validate(String... values).

For example if you are passing ("1.11", "2.22"….) to validate(), it should return an empty list and a test case must be written to verify this acceptance criteria.

Try writing the test for the validate method, run the test and see it fail, and then implement the new code to see the test pass.

Step 11: Test the validate method by passing a single three-digit decimal

Follow the AC's below:

Using TDD Pass a single String with decimal value 1.111 in to the validate method. Assert that a List<String> of size 1 is returned. In this case we are passing “1.111” then it should contain the message 'Too many decimal places for value 1.111.' Try writing the test for too many decimals being passed in, run the test, and see it fail, and then implement the new code to see the test pass.

Step 12: Test the validate method with multiple three digit decimal Strings

Follow the AC's below:

Using TDD Pass a two Strings with three decimal places with values 1.111, 2.222 in to the validate method. Assert that a List<String> of size 2 is returned containing the messages 'Too many decimal places for value 1.111.', 'Too many decimal places for value 2.222.'

Try writing the test for too many decimals being passed in for both values, run the test and see it fail, and then implement the new code to see the test pass.

Step 13: Pass the Validator into the Calculator class

Follow the acceptance criteria below working from a TDD perspective.

Create new variants of add(), multiply(), divide() and subtract(). For example, create a function add_revised() which will the same functionality as addstrings() implemented in step 6. But there will no duplication of code. Instead it will utilize the functionality of Validate class for validation checks. Similarly do the same process for other functionalities like multiply(), divide() and subtract().

AC: Pass an instance of the Validator class into the Calculator constructor so that the Add, Subtract, Multiply and Divide methods can use it to validate their values (for two decimal places) rather than use the duplicate code we currently have.

NOTE – REGARDING ACADEMIC INTEGRITY:

All code you write for this assignment should be original, designed from your brain, and written by you. If you use the templates from the sample projects, get rid of the old sample code and tests. Make sure there is just **your** code left in the project.

All code submitted to Gitlab for CSCI5308 is automatically processed by MOSS and other plagiarism detection software. Do not plagiarize from any other students (current or previous) from this course or any other 3rd party source.

Marking Rubric

When we say "your CI/CD system is functioning" we mean that your system detects commits and pushes to your **master** branch, that it gets those changes, builds them, and then executes your Junit unit tests, and finally uploads the results of those tests back to Gitlab. If any of those steps do not function, the system does not function.

Your TAs will examine your code and written response and rate you on the following scale:

Exceptional (A+)

The criteria for an **A**, and in addition, you added some additional test cases apart from the given scenarios.

Very Good (A)

Your CI/CD system is functioning, you have no issues or incorrect code, you have no issues with your test coverage or the quality of your unit tests. Your written document properly describes the scenarios and how your code make failing test to “pass”. All the steps are implemented properly.

Good (B+ to A-)

Your CI/CD system is functioning, one of your **test cases** has small issues or isincorrect, or your tests lack coverage or have poor quality. Your written document properly describes the test coverage and how your code is achieving the same. At least first 10 steps are implemented properly.

Minimal (B-)

Either your CI/CD system is not functioning (but you tried to get it working) and your tests lack coverage or have poor quality. There may be issues in your written document that make your explanations confusing. At least first 8 tests are running properly.

Unacceptable (F)

Your CI/CD system is not functioning, and your **test cases** were incorrect (or missing), and/or your written documentdoes not meet sufficiently describe your code. Less than 8 steps are implemented.