CS 405 Module Four Milestone Guidelines and Rubric

Overview

You are a senior software developer on a team of software developers who are responsible for a large banking web application. Your manager has recently learned about the best practice of creating unit tests for software and wants to see a full example of how it works. Another developer on the team started creating a unit test for the std::vector class, and he managed to get the TextFixture and a few tests completed before his vacation started. Your manager is impatient and has asked you to complete the task. He wants to see a number of tests, including positive and negative tests.

Key Concepts

- Positive tests prove that functionality works when tested. For example, `Test(1+1) = 2`.
- Negative tests prove that an error or exception happens when provided bad data. For example, `Test(5/0) => Divide By Zero`.

The following are some essential notes:

- The source code has been commented using TODO to explain the detailed rules you must follow.
- You are provided the situation to create 13 unit tests, of which 11 are defined for you and 2 you will need to define.
- You only need a minimum of 2 negative tests, but can do more.
- Test names should reflect the purpose of a test. For example, to test that a number is positive, the test name would be something like `EnsureNumberIsPositive`.
- Do not confuse the `C++ assert / static_assert` with the Google Test `ASSERT` and `EXCEPT`.
    - When should you use the `EXPECT_xxx` or `ASSERT_xxx` macros?
        - Use `ASSERT` when failure should terminate processing, such as the reason for the test case.
        - Use `EXPECT` when failure should notify, but processing should continue.
- There are multiple ways to validate the test results, but each test must explicitly prove the defined condition of the test.
- Do not forget that you can leverage capabilities provided by the standard C++ library to help you achieve success.

You will learn to do the following:

- Name unit tests correctly
- Compile and run a Google unit test fixture
- Extend Google unit test fixture with a number of positive and negative tests
- Prove the test results

Prompt

Test the existing source code in the test.cpp file using the Google unit testing framework. Include a brief written summary of the process you used, the issues you found, and how you managed the issues.

Specifically, address the following in a static testing summary:

- **Unit Test Names**: Define all unit test names to appropriately reflect the test condition.
- **Unit Testing**: Successfully implement the 13 unit tests, as part of the Google Test fixture; run Google Test `ASSERT` and `EXPECT` functionality to prove the tests. Each test you run must explicitly prove the defined condition of the test.
- **Negative Unit Tests**: Complete at least 2 of the unit tests as negative tests that demonstrate capturing the appropriate unit test result based on an expected negative result of the test code.
- **C/C++ Program Functionality and Best Practices**: Demonstrate industry standard best practices, including in-line comments and appropriate naming conventions to enhance readability of code. Develop functional C/C++ code that illustrates a software design pattern approach.
- **Process Summary**: Provide a summary of the debugging that is thorough and systematic, including specific types of bugs, and that accurately describes the corrections.

To complete this assignment, download the test.cpp file and use the Google Test Guide PDF as guidance as you move through the activity. You will use your development environment to complete this activity. Google Test has a number of resources online, including the following:

- Googletest Primer
- How to Use Google Test for C++ in Visual Studio

What to Submit

Submit the completed source code as a ZIP file. Also include a Word document that contains a screenshot of the unit test run results (console application or the Visual Studio test explorer) and your brief process summary.

Module Four Milestone Rubric

Criteria	Proficient (100%)	Needs Improvement (75%)	Not Evident (0%)	Value
Unit Test Names	Defines all unit test names to reflect the test condition appropriately	Defines most names well, but some do not accurately reflect what is being tested	Does not attempt criterion	20
Unit Testing	Successfully implements the 11 defined and 2 custom unit tests as part of the Google Test Fixture, runs Google Test ASSERT and EXPECT functionality to prove the tests, and each test explicitly proves the defined condition of the test.	Completes most of the coding activity, but needs to use the provided test fixture or complete 11 defined units and 2 custom units; additional testing is needed to explicitly prove the defined condition of the test	Does not attempt criterion	40
Negative Unit Tests	Completes at least 2 negative tests that demonstrate capturing the appropriate unit test result based on an expected negative result of the test code	Completes negative tests, but does not properly implement the unit tests or has an incorrect result, or only implements 1 negative test	Does not attempt criterion	15
C/C++ Program Functionality and Best Practices	Demonstrates industry standard best practices, including in-line comments and appropriate naming conventions to enhance readability of code; develops functional C/C++ code that illustrates a software design pattern approach	Shows progress toward proficiency, but with errors or omissions; areas for improvement may include implementation of software design patterns in code	Does not attempt criterion	15
Process Summary	Provides a summary of the debugging that is thorough and systematic, including specific types of bugs, and accurately describes the corrections	Provides a summary of the debugging, but the summary lacks detail, the approach is unsystematic, specific types of bugs are not differentiated, or the corrections are not fully described	Does not attempt criterion	10
			Total:	100%