

**SET-221**

**Software Testing Technologies LAB # 05**

**LAB Title**

Writing Test Fixtures in Google Test

Assessment of CLO: 04, PLO: 03

|  |  |  |  |
| --- | --- | --- | --- |
| Student Name: |  | | |
| Roll No. |  | | |
| Semester |  | Session |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S.**  **No.** | **Perf. Level**  **Criteria** | **Excellent (2.5)** | **Good (2)** | **Satisfactory (1.5)** | **Needs Improvement (0 ~ 1)** | **Marks Obtained** |
| **1** | Project Execution & Implementation | Fully functional, optimized, and well-structured. | Minor errors, mostly functional. | Some errors, requires guidance. | Major errors, non-functional,  or not Performed. |  |
| **2** | Results & Debugging Or  Troubleshooting | Accurate results with effective debugging  Or Troubleshooting. | Mostly correct, some debugging Or Troubleshooting needed. | Partial results, minimal debugging  Or Troubleshooting. | Incorrect results, no debugging Or Troubleshooting, or not attempted. |  |
| **3** | Problem- Solving & Adaptability (VIVA) | Creative approach, efficiently solves challenges. | Adapts well, minor struggles. | Some adaptability, needs guidance. | Lacks innovation or no innovation, unable to solve problems. |  |
| **4** | Report Quality & Documentation | Clear, structured, with detailed visuals. | Mostly clear, minor gaps. | Some clarity issues, missing details. | Poorly structured, lacks  clarity, or not submitted. |  |
| **Total Marks Obtained Out of 10** | | | | | |  |

Experiment evaluated by

|  |  |  |  |
| --- | --- | --- | --- |
| Instructor’s Name | Engr.Bushra Aziz | | |
| Date |  | Signature |  |

***Copyright © Department of Engineering & Technology – UIT University Karachi***

**Objective:** This lab will guide you through creating and using test fixtures in Google Test.

# Theory:

**Introduction to Test Fixtures:**

In Google Test, a **test fixture** is a class that sets up a common environment for multiple tests. It's a way to avoid code duplication and make tests more maintainable. The test fixture class contains member variables and setup/teardown methods that are shared by all tests within the same test suite.

# Why Use Test Fixtures?

* **Reduced Code Duplication:** Avoid repeating setup and teardown logic in each test.
* **Improved Maintainability:** Centralize setup and teardown, making changes easier.
* **Enhanced Readability:** Separate setup/teardown from test logic.
* **Consistent Test Environment:** Ensure each test runs in a predictable state.

# Setup and TearDown Methods:

* **SetUp():** This method is called *before* each test case within the fixture. It's used to initialize resources, create objects, or set up any other necessary preconditions for the tests.
* **TearDown():** This method is called *after* each test case within the fixture. It's used to clean up resources, delete objects, or restore any state that was modified by the tests.

**Example:**

# Calculator Class Files:

1. **cal.h (Header File):**

## C++

#ifndef CAL\_H #define CAL\_H

class Calculator { public:

double Add(double a, double b); double Subtract(double a, double b); double Multiply(double a, double b); double Divide(double a, double b);

};

#endif // CAL\_H

1. **cal.cpp (Implementation File):**

## C++

#include "cal.h" #include <stdexcept>

double Calculator::Add(double a, double b) { return a + b;

}

double Calculator::Subtract(double a, double b) { return a - b;

}

double Calculator::Multiply(double a, double b) { return a \* b;

}

double Calculator::Divide(double a, double b) { if (b == 0) {

throw std::invalid\_argument("Division by zero");

}

return a / b;

}

1. **main.cpp (Console Application File):**

## C++

#include "cal.h" #include <iostream> #include <stdexcept>

int main() { Calculator calc;

double num1, num2; char operation;

std::cout << "Enter first number: "; std::cin >> num1;

std::cout << "Enter operation (+, -, \*, /): "; std::cin >> operation;

std::cout << "Enter second number: "; std::cin >> num2;

try {

double result; switch (operation) {

case '+':

result = calc.Add(num1, num2); break;

case '-':

result = calc.Subtract(num1, num2); break;

case '\*':

result = calc.Multiply(num1, num2); break;

case '/':

result = calc.Divide(num1, num2); break;

default:

std::cout << "Invalid operation." << std::endl; return 1;

}

std::cout << "Result: " << result << std::endl;

} catch (const std::invalid\_argument& e) { std::cerr << "Error: " << e.what() << std::endl; return 1;

}

return 0;

}

1. **calculator\_test.cpp (Google Test File):**

## C++

#include "gtest/gtest.h" #include "cal.h" #include <stdexcept>

class CalculatorTest : public ::testing::Test { protected:

void SetUp() override { calc = new Calculator();

}

void TearDown() override { delete calc;

}

Calculator\* calc;

};

TEST\_F(CalculatorTest, AddPositiveNumbers) { EXPECT\_EQ(5.0, calc->Add(2.0, 3.0));

}

TEST\_F(CalculatorTest, AddNegativeNumbers) { EXPECT\_EQ(-5.0, calc->Add(-2.0, -3.0));

}

TEST\_F(CalculatorTest, SubtractPositiveNumbers) { EXPECT\_EQ(1.0, calc->Subtract(3.0, 2.0));

}

TEST\_F(CalculatorTest, SubtractNegativeNumbers) { EXPECT\_EQ(-1.0, calc->Subtract(-3.0, -2.0));

}

TEST\_F(CalculatorTest, MultiplyPositiveNumbers) { EXPECT\_EQ(6.0, calc->Multiply(2.0, 3.0));

}

TEST\_F(CalculatorTest, MultiplyNegativeNumbers) { EXPECT\_EQ(6.0, calc->Multiply(-2.0, -3.0));

}

TEST\_F(CalculatorTest, DividePositiveNumbers) { EXPECT\_EQ(2.0, calc->Divide(6.0, 3.0));

}

TEST\_F(CalculatorTest, DivideByZero) {

EXPECT\_THROW(calc->Divide(6.0, 0.0), std::invalid\_argument);

}

Task:

Implement a test case that increments variable by 10, 20,100 and checks if the updated values are correct.