"Heaven's Light is Our Guide"

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Open-Ended Project

Course Code: CSE 3206

Course Title: Software Engineering Sessional

Submitted By-	Submitted To-
Name: Sajidur Rahman Tarafder	Farzana Parvin
Department: CSE	Lecturer
Roll No: 2003154	Department of CSE,
Section: C	RUET
Session: 2020-21	

Project Topic:

Imagine an application that models interactions with a country's government (e.g., issuing laws, policies, budgets). In a real-world system, a country has only one central government at any time. No matter how many ministries, departments, or citizens interact with it, they all refer to one single Government authority.

Chosen Design Pattern:

We can model the Government as a Singleton, ensuring there's only one instance throughout the application lifecycle.

Introduction:

Singleton is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It is one of the simplest design patterns.

This pattern ensures a class has only one instance during the application's lifecycle. It prevents creating new instances by always returning the same one. This pattern allows easy access to the instance from anywhere in the program and protects it from being overwritten. It also supports extending functionality through subclassing and helps manage shared resources like logging and database connections.

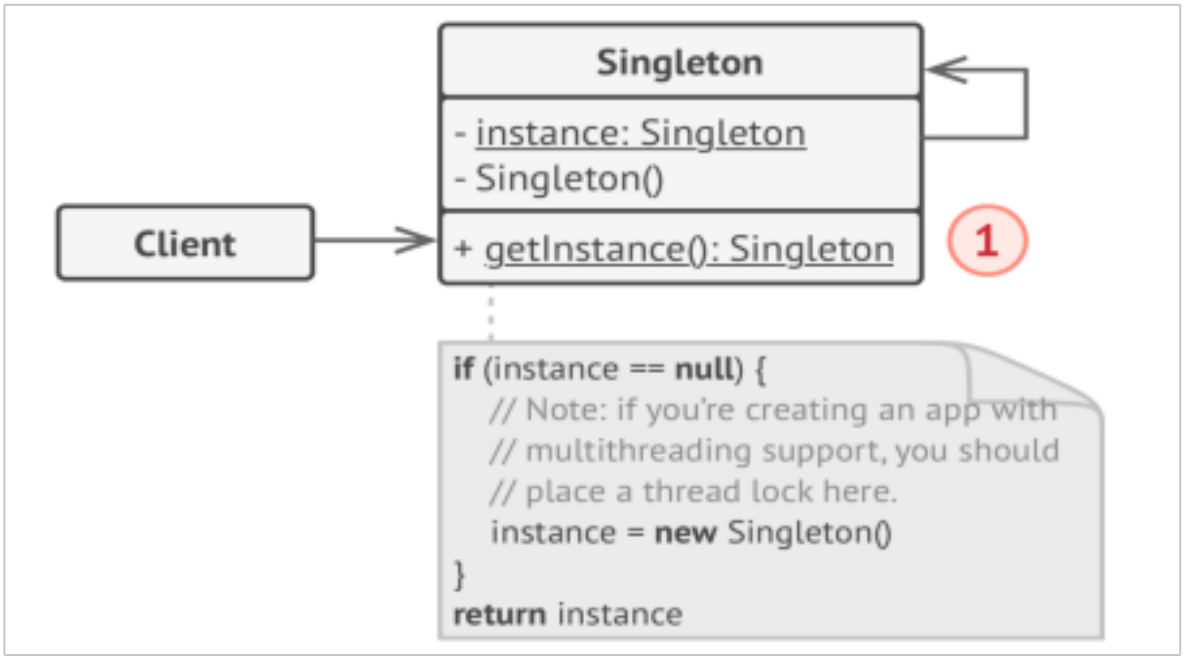
Components of Singleton Pattern:

Private Constructor: A regular constructor always returns a new object by design. This pattern solves this issue by making the default constructor as private constructor. It prevents creating fresh objects by using 'new' operator of that class.

Static Instance Variable: A static variable that holds the single instance of the class.

Public Static Method: Provides a global access point to the instance. It calls the private constructor internally and returns the same instance for all subsequent calls.

UML Diagram:



```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

Explanation:

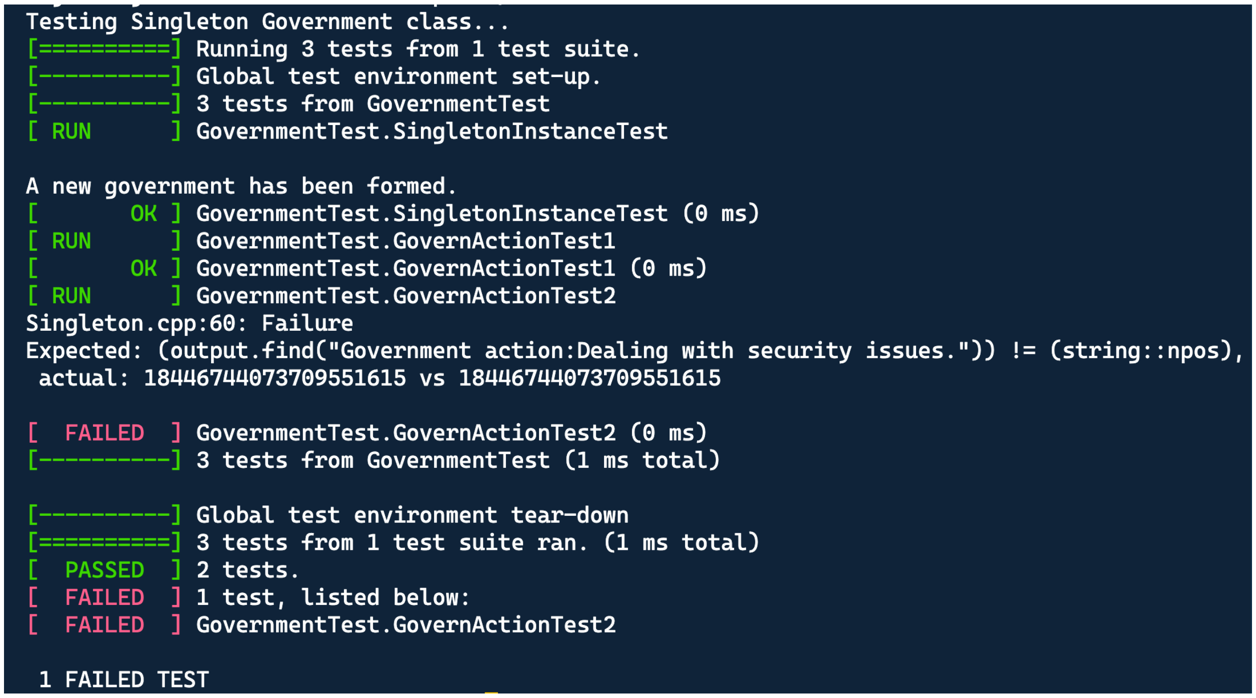
The Singleton class declares the static method getInstance that returns the same instance of its own class. The Singleton's constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the Singleton object.

Code:

```
#include <bits/stdc++.h>
#include <gtest/gtest.h>
using namespace std;

class Government {
private:
    static Government* instance;

// Private constructor ensuring only one instance can be
created in the program
    Government() {
       cout << "\nA new government has been formed." << endl;
    }

public:
    static Government* getInstance()
    {
        if (instance == nullptr) {
            instance = new Government();
        }
        return instance;
    }
    void Govern(const string& message)
    {
        cout << "Government action: " << message << endl;
    }
};
```

```

// Initializing static member
Government* Government::instance = nullptr;

int main(int argc, char** argv)
{
    cout << "Testing Singleton Government class..." << endl;
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

// Google Test Cases
TEST(GovernmentTest, SingletonInstanceTest)
{
    Government* gov1 = Government::getInstance();
    Government* gov2 = Government::getInstance();

    EXPECT_EQ(gov1, gov2);
}
TEST(GovernmentTest, GovernActionTest1)
{
    Government* gov = Government::getInstance();

    testing::internal::CaptureStdout();
    gov->Govern("Implementing new policies.");
    string output = testing::internal::GetCapturedStdout();

    EXPECT_EQ(output.find("Government action:Implementing new
policies."), string::npos);
}
TEST(GovernmentTest, GovernActionTest2) {
    Government* gov = Government::getInstance();

    testing::internal::CaptureStdout();
    gov->Govern("Dealing with security issues.");
    string output = testing::internal::GetCapturedStdout();

    EXPECT_NE(output.find("Government action:Dealing with
security issues."), string::npos);
}
```

Terminal Output:



```
Testing Singleton Government class...
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from GovernmentTest
[ RUN      ] GovernmentTest.SingletonInstanceTest

A new government has been formed.
[       OK ] GovernmentTest.SingletonInstanceTest (0 ms)
[ RUN      ] GovernmentTest.GovernActionTest1
[       OK ] GovernmentTest.GovernActionTest1 (0 ms)
[ RUN      ] GovernmentTest.GovernActionTest2
Singleton.cpp:60: Failure
Expected: (output.find("Government action:Dealing with security issues.")) != (string::npos),
 actual: 18446744073709551615 vs 18446744073709551615

[  FAILED  ] GovernmentTest.GovernActionTest2 (0 ms)
[----------] 3 tests from GovernmentTest (1 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (1 ms total)
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] GovernmentTest.GovernActionTest2

 1 FAILED TEST
```

Code Structure:

This C++ code demonstrates the Singleton design pattern through a Government class, which ensures that only one instance of the class can exist by using a private static pointer (instance), a private constructor, and a public static method (getInstance()) that creates the instance if it doesn't exist and always returns the same pointer. The class also provides a Govern method that outputs a message to standard output, simulating a government action. The code integrates Google Test for unit testing, initializing the test framework in the main() function and running all tests. Three test cases are defined: the first (SingletonInstanceTest) checks that multiple calls to getInstance() return the same pointer, confirming the singleton property; the second (GovernActionTest1) captures the output of the Govern method when called with "Implementing new policies." and expects that the output does contain the expected message, which would cause the test to success if the message is present; the third (GovernActionTest2) captures the output when called with "Dealing with security issues.” and expects that the output doesn’t contain the expected message which would cause the test to fail if the message is present. The tests use Google Test's EXPECT_EQ and EXPECT_NE assertions. This structure not only enforces the singleton pattern but also demonstrates how to use Google Test to verify both object behavior and console output in C++.