The Importance of Tests

David Sackstein

Agenda

- Why do we need tests?
- What are good tests?
- Types of tests
- How do we make code testable?
- Introducing GoogleTest
- Dependency Injection
- Mocks and stubs
- Testability challenges and the options you have

Why do we need tests?

- To verify that the code works as expected
- To step the code for debugging
- To document usage of the code
- To lock in certain functionality so we can make changes safely
- To force us to use good design principles, in particularly
  - Decoupling
  - Clear interfaces

What are good tests?

- Test should be F-I-R-S-T
  - Fast
  - Independent
  - Repeatable
  - Self-validating
  - Timely

Types of tests

- Unit test
  - Test a function of set of functions usually of one class.
- Component test
  - Tests an entire component, mocking interactions with other components
- Integration test
  - Tests the interaction between components.
- Functional test (or end-to-end)
  - Tests the entire functionality of flow as used by the user.
- Characteristic Tests: Coming up in Legacy Code

How do we make code testable?

- Apply the principles we have learned so far:
  - Components should be small
  - Components should have one responsibility
  - Components should have a clear interface
- Implement Dependency Injection
  - Dependencies should be provided as arguments to the constructor and not modified.
  - They should preferably be references or const pointers so they cannot be reassigned.

Test driven development

- TDD takes testing a step further.
  1. Write failing tests (Red)
  2. Write the code to make the tests pass (Green)
  3. Refactor, if necessary (keep Green)
  4. Go to 1
- TDD ensures good test coverage
- Sometimes the rigid ritual it is not justified
- It is another tool in your toolbox

Google Test

- A C++ unit testing framework by Google (gtest)
- It does not require any changes to your code
- Can also be used for other test type
- This depends a lot on your design
- It provides:
  - Integrated test discovery and execution
  - Assertions for comparing values
  - Stand alone tests without context
  - Test fixtures with setup/teardown
- Google Mock provides simple mocks and stubs
  - I recommend you write your own!

The only include you need

The TEST macro defines a suite name and a test name

Assertions abort the test if they fail

Expectations allow the test to continue if they fail

```cpp
#include <gtest/gtest.h>

#include <string>
#include <vector>

TEST(TestSuite, Addition) {
    constexpr int sum = 2 + 3;
    EXPECT_EQ(sum, 5);
}

TEST(TestSuite, StringConcatenation) {
    const std::string hello = "Hello, ";
    const std::string world = "world!";
    ASSERT_EQ(hello + world, "Hello, world!");
}

TEST(TestSuite, VectorSize) {
    const std::vector<int> nums = {1, 2, 3, 4};
    EXPECT_EQ(nums.size(), 4);
}

TEST(TestSuite, BooleanCondition) {
    constexpr bool condition = (10 > 5);
    EXPECT_TRUE(condition);
}
```

Define the fixture as a class in the header

The TEST_F macro defines a test name in a fixture

SetupSuite/TearDownSuite are called before and after the entire fixture

Setup/TearDown are called before and after each test

```cpp
#pragma once

#include <gtest/gtest.h>

class MathTestFixture : public ::testing::Test {
protected:
    int a = 0;
    int b = 0;

    static void SetUpTestSuite();
    static void TearDownTestSuite();

    void SetUp() override;
    void TearDown() override;
};
```

```cpp
#include "math_test_fixture.h"

#include <iostream>

TEST_F(MathTestFixture, AdditionTest) {
    std::cout << "AdditionTest" << std::endl;
    EXPECT_EQ(a + b, 15);
}

TEST_F(MathTestFixture, DivisionTest) {
    std::cout << "DivisionTest" << std::endl;
    ASSERT_NE(b, 0); // Prevent division by zero
    EXPECT_EQ(a / b, 2);
}

void MathTestFixture::SetUpTestSuite() {
    std::cout << "SetUpTestSuite before all tests" << std::endl;
}

void MathTestFixture::TearDownTestSuite() {
    std::cout << "TearDownTestSuite after all tests" << std::endl;
}

void MathTestFixture::SetUp() {
    std::cout << "SetUp" << std::endl;
    a = 10;
    b = 5;
}

void MathTestFixture::TearDown() {
    std::cout << "TearDown" << std::endl;
}
```

Dependency Injection

- Dependency Injection is a key development technique that improves testability and extensibility.
- Rather than depending on concrete classes, depend on interfaces.
- The interfaces are passed into the ctor and stored as immutable fields.
- This is how we implemented Composition in Prefer Composition over Inheritance

Dependency Injection: Benefits

- Implementations can be changed without changing the class that uses them.
- This allows you to extend the code easily.
- It also allows you to replace real implementations with mocks and stubs.
- This makes your class testable.
- Note: testability is often a motivation for improving design

Mocks and Stubs: The Problem

- Testing requires the ability to run a class in an isolated environment.
- This requires separation between the class and its environment.
- But if the class is coupled with databases, files, and other classes, this is very difficult.
- As a result, you need to set up all the dependencies in order to test the class.
- Apart from the setup complexity it also makes the tests run slowly.

Mocks and Stubs: The Solution

- If the dependencies are interfaces, you can supply a special implementation that does the minimum required to allow the test to pass.
- This is ideal for dependencies that do not change the behavior of the class (a logger, for instance).
- These are stubs:
  - A stub is a minimal implementation of an interface that is sufficient to allow a class to run in a test environment.

Mocks and Stubs: Another Problem

- Tests call the public API of a class and verify functional behavior.
- But how do we test the private methods? Do we need to?
- One solution:
  - Extract a class in which the private methods become the public methods of the extracted class. These methods can be tested.
  - Note: Tests are again a motivation to improve design
- Another solution is to write mocks.

Mocks are like spies

- A mock is an implementation of a dependency interface that maintains a state.
- It can record arguments that are passed to its methods, react in different ways to those methods.
- Moreover, it is useful to program a mock to exercise certain behaviors. (Error injection)
- And we can query it after the test to verify how it was invoked during the test.

Testability challenges and solutions

- How can I test private methods?
  - Use a mock or extract a class (discussed above)
- Methods that call sleep are not fast.
  - Avoid using sleep where possible
  - Stub the sleep function to do nothing
  - Or increment a time variable, if you need to know the time elsewhere

Testability challenges and solutions

- Methods that depend on time may not be repeatable.
  - Try not to depend on time
  - Instead of requesting the time, accept time as a parameter. Push the problem up in the call stack.
  - If you must request the time, provide the clock as an interface and mock it.
  - You may want to define a mock that provides immediate responses to sleep(), gettime(), wait() and so on.

Testability challenges and solutions

- Classes that run concurrent tasks may not be repeatable.
  - Do you really need concurrency? (I have to ask)
  - Push concurrency as high up as possible in the call stack, keeping the core logic single threaded and repeatable.
  - Focus on the non-concurrent parts and test different possible orderings that are repeatable.

Agenda

- Why do we need tests?
- What are good tests?
- Types of tests
- How do we make code testable?
- Introducing GoogleTest
- Dependency Injection
- Mocks and stubs
- Testability challenges and the options you have

Key Takeaways

Why test?	Verification, debugging, documentation and an motivator of good design
Which framework?	Choose a framework that does not force you to change your code (e.g. gtest)
Testability	Use good design principles and use dependency injection, mocks and stubs
Private methods?	Must be tested too. Use class extraction to make them public or use mocks.
Good tests	Should be F.I.R.S.T. Test Driven Development is not a must