A Review of Unit Test Patterns for C++

Overview

In this article we'll review some unit testing patterns and outline the main patterns found in tested C++ code. Additionally, we'll go over common problems that you may encounter with each pattern(the examples we discuss here, were written and runin a GTest unit test framework). Enjoy!

Introduction

Introducing unit tests to our work flow helps us, as developers, produce cleaner and faster code. Adding them assists us in setting up a safety net that will be useful for refactoring and detecting regressions early on. This simple shift in our routine can immediately transform our work to be more Agile in nature, which can be useful, as many of us work in companies that are striving to adopt some level of Agile or DevOps methodology.

However, some developers complain that writing unit tests requires too much effort for too little reward. The result is either a rejection of unit testing altogether, or the "illusion" of writing tests, (i.e. writing the wrong tests or ignoring the test results and missing the point, entirely). Therefore we decided to demonstrate some basic testing patterns that will empower you to get started writing good tests.

What does a good test look like? We have compiled here a list that showcases the common characteristics::

- **Isolated**:
  A unit test should only test the logic we need to have tested. It shouldn't test other components. Failure to test isolated code will lead to a high coupling between tests, and changing a class may cause many unrelated tests to fail. The normal scope is a class, although sometimes it makes more sense to test a cluster of classes that are closely coupled (functionality-wise).
- **Fast:**
  Unit tests need to be fast in order to give the feedback to the developers as soon as possible. As a rule of thumb, if you notice that a test takes more than half a second to run, then it's time to check your test again.
- **Self-contained:**
  Unit tests that rely on external information are prone to fail and require configurations and setups. This leads to test discarding. To make sure that this doesn't happen, all the information required for the tests should be included in the tests. For example, instead of relying on an environment variable to be set, we must set it ourselves in the test.
- **Independent:**
  A unit test should not rely on other tests to be run before or after. They must be able to be run individually so that a developer can run a single test at a time.
- **Readable:**
  A unit test should be easy to understand and self-explanatory - from its name to its assertion.
  It should be clear what is being tested, and what we expect from the test.
  It helps us understand the production code and if the test fails, we can trace the issue faster.
- **Maintainable:**
  Tests should be seen as part of the software code even though it doesn't make it to production. As such, it must be maintainable and refactored when needed.

Patterns, Patterns, Patterns

Throughout this article, we will use an example of a Deposit class. As the name suggests, this class is responsible for depositing *funds into the bank account* of our system. Each pattern description is divided into three parts:

- When to use the patterns? What are the cases in which we will use the pattern?
- How to use the pattern? What are the recommended steps of the pattern?

- Examples- what do these patterns look like in practice?

Let's delve into the patterns now.

1. Arrange Act Assert - AAA Pattern

When should it be used?

A common way of writing unit tests is applying the AAA (Arrange-Act-Assert) pattern when setting up your tests. Not only does this concept help you understand your tests better and easier, it structures your tests in a simple way.

How should be it used?

1. Arrange: setup everything needed for running the tested code. This includes any initialization of dependencies, mocks, and data needed for the test to run (This can also be done inside the test setup methods).
2. Act: Invoke the code under test.
3. Assert: Specify the pass criteria for the test, which fails it if not met.

Example

The following code snippet shows an example: test the Deposit Method

```
void DepositAccount::Deposit(const double t_amountToDeposit);
```

This is what our test might look like:

```
TEST_F(UnitTestPatterns, ArrangeActAssertPattern_DepositFundsInEmptyAccount_AccountBalanceWillGrow)
{
    // Arrange
    DepositAccount depositAcc;

    // Act
    depositAcc.deposit(100.2);

    // Assert
    EXPECT_THAT(depositAcc.getBalance(), Eq(100.2));
}
```

Note: For future reference, by clearly naming the test we will be able to easily understand the test's purpose.

2. Test-Exception Pattern

When should it be used?

When we want to test that our code raises an exception in the proper conditions

How should be it used?

There are 2 ways:

1. Use the EXCEPT_THROW or equivalent
2. Catch the exception in the test, fail if the expectation wasn't thrown or if the wrong exception was thrown

Example

```
bool DepositAccount::Withdraw(double t_amount);
```

Let's

assume that our DepositAccount::Withdraw throws an OverdraftException when we exceed the withdrawal limit.

This is how a test would look like:

```
TEST_F(UnitTestPatterns, ExceptionTestPattern_WithdrawTooMuchFunds_ExceptionWillBeThrown)
{
    DepositAccount depositAcc;

    EXPECT_THROW(depositAcc.Withdraw(10) , OverdraftException);
}
Or
TEST_F(UnitTestPatterns, ExceptionTestPattern2_WithdrawTooMuchFunds_ExceptionWillBeThrown)
{
    // Arrange
    DepositAccount depositAcc;

    try {
       // Act
       depositAcc.Withdraw(10);
       // Assert
       FAIL() << " Withdraw() should throw an error\n";
    }
    catch (OverdraftException& exception) {
       EXPECT_THAT(std::string(exception.what()), Eq("overdrawn”));
       EXPECT_THAT(exception.errorCode, Eq(2));
    }
}
```

3. Nested-Class Test Pattern

When should it be used?

When a protected entity (field or method) needs to be accessed for the test. This can occur in two scenarios:

1. When we need to test a protected method or access a protected field.
2. When we need to override a public or protected method (can be done only if the method is virtual).

How should it be used?

Create a new class that extends our tested class.To test a protected method, add a method in our extended class that calls the protected method (delegation). To override a method, override this method in our extended class.

Example - Testing a protected method

The following code snippet shows an example of how to test the protected Withdraw method and use the protected m_balance field

```
class DepositeAccount {
protected:
       double m_balance;
       bool Withdraw(double t_amount);
```

This is how our test should appear:

```
class TestDepositeAccount : public DepositeAccount
{
public:
	using DepositeAccount::m_balance; // way 1 – make protected public
	bool Withdraw(double t_amount)    // way 2 – override call protected public
	{
		return DepositeAccount::Withdraw(t_amount);
	};

};

TEST_F(UnitTestPatterns, NestedClassPattern_WithdrawFunds_AccountBalanceWillShrink)
{
    // Arrange
    TestDepositeAccount depositAcc;
    depositAcc.m_balance = 200.4;

    // Act
    depositAcc.Withdraw(100.2);

    // Assert
    EXPECT_THAT(depositAcc.m_balance, Eq(100.2));
}
```

4. Interaction test pattern

Fake objects are objects that replace the real objects and return hard-coded values. This helps test the class in isolation.

Fakes can be used to validate that the interactions between objects behave as expected.

Fake and mock frameworks are frameworks that build mock objects on the fly. There are quite a few frameworks; the difference between them is in the way the developer writes the interactions with the object.

When should it be used?

Fakes can be used in the following cases:

- The real object has a nondeterministic behavior (it produces unpredictable results, like a date or the current weather temperature)
- The real object is difficult to set up
- The behavior of the real object is hard to trigger (for example through a network error)
- The real object is slow
- The real object has (or is) a user interface
- The test needs to ask the real object how it was used (for example, a test may need to confirm that a callback function was actually called)
- The real object does not yet exist (a common problem when interfacing with other teams or new hardware systems)

How should it be used?

The three key steps to using mock objects for testing are:

1. Use virtual or pure virtual methods to describe the object.
2. Implement these in one call for production code.
3. Implement these as a fake object for testing.

Note: The only way to use this is if the methods are virtual and the object can be passed to the code under-test

Example

Suppose we don't want to call the real Withdraw as that connects to a database – we can fake it.

```
virtual bool IDepositeAccount::Withdraw(double t_amount) = 0;

bool Teller::GiveMeFunds(DepositeAccount &account, double t_amount);
```

This is how our test might look like:

```
class FakeDepositeAccount : public IDepositeAccount
{
public:

	bool withdrawWasCalled;
	bool Withdraw(double t_amount)
	{
		withdrawWasCalled = true;
		return true;
	};
	FakeDepositeAccount() : withdrawWasCalled (false) {};

};

TEST_F(UnitTestPatterns, FakePattern_WithdrawFundsWithFake_MakeSureAccountWithdrawWasCalled)
{
    // Arrage
    Teller teller;
    FakeDepositeAccount depositAcc;

    // Act
    bool result = teller.GiveMeFunds(depositAcc, 10.2);

    // Asert
    EXPECT_THAT(depositAcc.withdrawWasCalled, Eq(true));
}
```

5. Using a Mock/Fake Framework

Using a Framework will save you time, writing the tests and maintaining them as the fakes are created dynamically and there is no need to rewrite the fake-class for every change of the code:

```
TEST_F(UnitTestPatterns, FakePattern_WithdrawFundsWithFake_MakeSureAccountWithdrawWasCalled)
{
    // Arrange
    Teller teller;
    auto depositAcc = FAKE<IDepositeAccount>(); // no need to rewrite class

    // Act
    bool result = teller.GiveMeFunds(depositAcc, 10.2);

    // Assert
    ASSERT_WAS_CALLED(depositAcc.Withdraw(EQ(10.2))); // here we are also validating the args
}
```

6. Interaction test pattern – Template redefinition

When should it be used?

If you have a template and this templated type is a Composite aggregation that you need to fake.

How should it be used?

When creating your object, swap the templated type with your fake type.

Example -

```
template<typename ACCOUNT> class Teller
{
private:
       ACCOUNT m_account;
};
```

And the test would look like this:

```
TEST_F(UnitTestPatterns,
RedefineTemplatePattern_WithdrawFundsWithFake_MakeSureAccountWithdrawWasCalled)
{
    // Arrange
    Teller<FakeDepositeAccount> teller;

    // Act
    bool result = teller.GiveMeFunds(depositAcc, 10.2);

    // Assert
    EXPECT_THAT(depositAcc.withdrawWasCalled, Eq(true));
}
```

7. Isolation test pattern

When should it be used?

There are many cases in which the above unit test patterns are not enough. As a result, developers have to refactor their code to make them 'testable'. Examples of code that needs to change include:

- Singleton classes
- Calls to static members
- Calls to non-virtual members
- Free C-Style functions
- Composite aggregation classes
- Objects that are instantiated in the code being tested
- Objects that are not passed to the method

The main problem is that refactoring to make the code testable, e.g. for the benefit of writing tests for it, when you don't have the safety net of existing unit tests, does not make sense. It's risky and costly.

How should it be used?

Use an isolation framework and work through these codes:

Example –

We'll use Typemock Isolator++ API to fake a static DepositAccount::Withdraw method.

```
static void CDepositeAccount::Withdraw(double t_amount);

bool Teller::GiveMeFunds(double t_amount);
```

And here is the test:

```
TEST_F(UnitTestPatterns, IsolationPattern_WithdrawFundsWithFake_MakeSureAccountWithdrawWasCalled)
{
    // Arrange
    Teller teller;
    WHEN_CALLED(DepositAccount::Withdraw(_)).Ignore(); // fake static method

    // Act
    bool result = teller.GiveMeFunds(10.2);

    // Assert
    ASSERT_WAS_CALLED(DepositAccount::Withdraw(EQ(10.2))); // make sure it was called

    ISOLATOR_CLEANUP()
}
```

Here is an example with composite aggregation:

```
class Teller
{
private:
	DepositeAccount m_account;
}
```

Here is the test:

```
TEST_F(UnitTestPatterns, IsolationPattern_WithdrawFundsWithFake_MakeSureAccountWithdrawWasCalled)
{
    // Arrange
    auto allAccounts = FAKE_ALL<DepositeAccount>(); // all accounts are faked = even future instances
    Teller teller;
    WHEN_CALLED(allAccounts->Withdraw(_)).Ignore(); // fake method for all instances

    // Act
    bool result = teller.GiveMeFunds(10.2);

    // Arrange
    ASSERT_WAS_CALLED(allAccounts->Withdraw(EQ(10.2))); // make sure it was called at least once
    ISOLATOR_CLEANUP()
}
```

Another example calls a private method:

```
class DepositeAccount {
private:
	double m_balance;

	bool Withdraw(double t_amount);
```

Here is the test:

```
TEST_F(UnitTestPatterns, IsolationPattern_CallPrivateWithdraw_MakeSureBalanceIsOk)
{
    // Arrange
    DepositeAccount account;
    ISOLATOR_SET_VARIABLE(account,m_balance,100.0);

    bool res = false;
    // Act
    ISOLATOR_INVOKE_FUNCTION(res, account, m_ Withdraw ,50.0);

    // Assert
    double balance = 0;
    ISOLATOR_GET_VARIABLE(account, m_balance, balance);

    EXPECT_THAT(balance, Eq(50.0));
    ISOLATOR_CLEANUP()
}
```

Summary

In this article, we reviewed the characteristics that make up a good unit test and the impact that the quality of the tests may actually have on the adoption of Unit Testing as a practice. Once we established these basics, we introduced the most common unit test patterns. For each pattern we discussed when to use them, how to use them and we reviewed examples for how the tests should look. It is important to note that these patterns can be used alone or in various combinations. For example, when you want to fake a database connection but it is created in a protected virtual method, the test can use the Nested Class Pattern to return the fake database.

And finally, we reviewed the isolation pattern, and the benefits of using it over other patterns (we also took the opportunity to demonstrate the Typemock Isolator++ Professional, a powerful isolation framework). One of the main challenges that prevent successful implementations of unit testing is the challenge of adoption. When writing the tests becomes too complex, too time-consuming and too error-prone, many developers come to the conclusion that the risks of unit testing outweigh the benefits. That is why choosing the right testing patterns and the right framework is so important, since it can help you to create the good kind of unit tests, those that will make your life easier, and will lead the way to better unit test adoption, wider test coverage and, ultimately, more time for coding.