Unit test là gì

Unit Test là một loại kiểm thử phần mềm trong đó các đơn vị hay thành phần riêng lẻ của phần mềm được kiểm thử. Kiểm thử đơn vị được thực hiện trong quá trình phát triển ứng dụng. Mục tiêu của Kiểm thử đơn vị là cô lập một phần code và xác minh tính chính xác của đơn vị đó.

Đặc tính của Unit test

- Nhanh: Thời gian để chạy 1 unit test rất ngắn
- Cô lập: Unit test có thể chạy riêng lẻ và không phụ thuộc vào bất kỳ yếu tố bên ngoài
- Lặp lại: Kết quả của một bài unit test luôn không đổi nếu bạn không thay đổi gì giữa các lần chạy
- Tự kiểm tra: Unit test có thể tự phát hiện nếu nó chạy sai hoặc đúng mà không cần sự tương tác của người viết
- Đúng lúc: Thời gian viết unit test không nên tốn quá lâu so với thời gian viết đoạn code đang được kiểm tra

Các loại Unit Test

Black Box Testing: Kiểm thử giao diện người dùng cùng với đầu vào và đầu ra của phần mềm

- Functional testing: toàn bộ các unit hoạt động đúng với design (Design detail)
- Bussiness testing: toàn bộ các chương trình phần mềm đúng với yêu cầu người dùng (User requirement)

White Box Testing: Kiểm thử hành vi chức năng của sản phẩm và đánh giá hoạt đông

- Kiểm tra cú pháp bởi compiler để tránh lỗi cú pháp.
- Chạy code bằng chế độ debug, chạy từng dòng để đảm bảo rằng tất cả các câu lệnh được chạy ít nhất một lần.
- Kiểm tra cấu trúc dữ liệu cục bộ để bảo đảm toàn bộ dữ liệu được lưu trữ tạm thời duy trì tính toàn vẹn của nó trong tất cả các bước chạy code.
- Kiểm tra các điều kiện để đảm bảo rằng toàn bộ code chạy đúng tại các ranh giới được thiết lập theo yêu cầu.
- Review lại tất cả các đường dẫn xử lý lỗi

Gray Box Testing: Được sử dụng để thực hiện các bộ test, trường hợp test, đánh giá rủi ro.

Cấu trúc của Unit Test

Cấu trúc 3A

- Arrange: Thiết lập môi trường test, thiết lập các thông số cần thiết
- Act: Đưa hệ thống vào test bằng cách gọi các method, truyền các tham số cần thiết và lấy giá trị đầu ra
- Assert: Đưa ra nhận định về kết quả cuối cùng

```
int Factorial(int n)
{
    if(n <= 1)
    {
        return 1;
    }
    return n*(Factorial(n-1));
}
TEST(SimpleTest, HandlePositiveInput)
{
    //Arrange
    int range = 5;
    int result;
    //Act
    result = Factorial(range);
```

```
    //Assert
    EXPECT_EQ(Result, 120);
}
```

GTest

Assertion

Assertion là các macro giống gọi hàm. Để kiểm tra một class hoặc function thì phải đưa ra các assertion về hành vi của nó. Khi assertion không thành công, googletest sẽ in vị trí, nguồn của Assertion, cùng với thông báo lỗi.

Danh sách các Assertion của Google Test: [Assertions Reference | GoogleTest]

Test Fixture

Nếu các bài test sử dụng chung một dữ liệu, sử dụng test fixture để thiết lập môi trường chung cho các bài test

- Đây là object cần được test. Object này gồm 2 method để check xem 1 số có là nguyên tố và tính tổng các số nguyên tố

```
class PrimeNumber
{
    public:
    PrimeNumber(){};
    ~PrimeNumber(){};
    bool isPrimeNumber(int x)
    {
        for(int i = 2; i <= x/2; ++i)
        {
            if(x % i == 0)
            {
                std::cout << "not prime" << std::endl;
                return false;
            }
        }
        std::cout << "is prime" << std::endl;
        return true;
    }
    int sumOfPrimeNumber(int n)
    {
        if(isPrimeNumber(n))
        {
            return n + sumOfPrimeNumber(n - 1);
        }
        else
        {
            return 0;
        }
    }
};
```

- Cách khởi tạo Fixture

1. Tạo ra một Class từ testing::Test. Phần thân của class sẽ là protected :, vì chúng ta muốn truy cập các thành viên các lớp con.

2. Bên trong class, khai báo bất kỳ object nào muốn sử dụng.
3. Nếu cần, hãy viết default constructor (hàm SetUp ()) để khởi tạo object cho mỗi lần test
4. Nếu cần, hãy viết destructor (hàm TearDown ()) để giải phóng bất kỳ tài nguyên nào đã phân bổ trong SetUp ().
5. Nếu cần, hãy xác định các chương trình con để có thể sử dụng trong bài test

```
class PrimeNumberFixture : public :: testing :: Test
{
    protected:
    PrimeNumber *prime;
    void SetUp() override
    {
        prime = new PrimeNumber();
    }
    void TearDown()
    {
        delete prime;
    }
};
```

- Bài test có thể sử dụng các object và method đã khởi tạo trong Fixture

```
TEST_F(PrimeNumberFixture, isPrime)
{
    bool result;
    result = prime->isPrimeNumber(17);
    EXPECT_EQ(result, true);
}
TEST_F(PrimeNumberFixture, sum)
{
    int result;
    result = prime->sumOfPrimeNumber(7);
    EXPECT_EQ(result, 18);
}
```

Parameterized Tests

Parameterized Tests cho phép test với các tham số khác nhau mà không cần viết nhiều bản sao của cùng một test

- Object cần được test

```
class PrimeNumber
{
    public:
    PrimeNumber(){};
    ~PrimeNumber(){};
    bool isPrimeNumber(int x)
    {
        for(int i = 2; i <= x/2; ++i)
        {
            if(x % i == 0)
            {
                std::cout << "not prime" << std::endl;
                return false;
            }
        }
```

```
        std::cout << "is prime" << std::endl;
        return true;
    }
};
```

- Có thể sử dụng Fixture nếu muốn

```
class PrimeNumberFixture : public :: testing :: Test
{
    protected:
    PrimeNumber *prime;
    void SetUp() override
    {
        prime = new PrimeNumber();
    }
    void TearDown()
    {
        delete prime;
    }
};
```

- Để viết Parameterized Tests, trước tiên khởi tạo một class cố định, kế thừa từ cả testing::Test và testing::WithParamInterface<T>, trong đó T là loại giá trị tham số của bạn.
- Khởi tạo bài test với `INSTANTIATE_TEST_SUITE_P`

```
class PrimeNumberParameterTest : public PrimeNumberFixture, public
::testing::WithParamInterface<int>{};
TEST_P(PrimeNumberParameterTest, IsPrimeCheck)
{
    int number = GetParam();
    EXPECT_TRUE(prime->isPrimeNumber(number));
}
INSTANTIATE_TEST_SUITE_P(PrimeTesting,
                         PrimeNumberParameterTest,
                         testing::Values(5, 9, 13));
```

Ví dụ: Viết test cho class Nguyên tố

- Class Số nguyên tố, gồm 1 method kiểm trasố nguyên tố

```
class PrimeNumber
{
    public:
    PrimeNumber(){};
    ~PrimeNumber(){};
    bool isPrimeNumber(int x)
    {
        for(int i = 2; i <= x/2; ++i)
        {
            if(x % i == 0)
            {
                std::cout << "not prime" << std::endl;
                return false;
            }
        }
```

```
        std::cout << "is prime" << std::endl;
        return true;
    }
};
```

- Fixture để khởi tạo object

```
class PrimeNumberFixture : public :: testing :: Test
{
    protected:
    PrimeNumber *prime;
    void SetUp() override
    {
        prime = new PrimeNumber();
    }
    void TearDown()
    {
        delete prime;
    }
};
```

```
class PrimeNumberParameterTest : public PrimeNumberFixture, public
::testing::WithParamInterface<int>{};
TEST_P(PrimeNumberParameterTest, IsPrimeCheck)
{
    int number = GetParam();
    EXPECT_TRUE(prime->isPrimeNumber(number));
}
```

- Test lần lượt với các giá trị 5, 7, 13, 15, 18. Trong đó 15 18 sẽ fail, thêm vào để check output

```
INSTANTIATE_TEST_SUITE_P(PrimeTesting,
                         PrimeNumberParameterTest,
                         testing::Values(5, 7, 13, 15, 18));
```

Kết quả:

Có thể thấy 15, 18 fail

```
[ctest]    Site: TruongVanHuy
[ctest]    Build name: Win32-mingw32-make
[ctest] Test project D:/C++/build
[ctest]     Start 1: ParametterizedTest
[ctest] 1/1 Test #1: ParametterizedTest .............***Failed    0.04 sec
[ctest] Running main() from D:\C++\GoogleTest\googletest\googletest\src\gtest_main.cc
[ctest] [==========] Running 5 tests from 1 test suite.
[ctest] [----------] Global test environment set-up.
[ctest] [----------] 5 tests from PrimeTesting/PrimeNumberParameterTest
[ctest] [ RUN      ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/0
[ctest] [       OK ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/0 (0 ms)
[ctest] [ RUN      ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/1
[ctest] [       OK ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/1 (0 ms)
[ctest] [ RUN      ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/2
[ctest] [       OK ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/2 (0 ms)
[ctest] [ RUN      ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/3
[ctest] D:\C++\GoogleTest\test\ParametterizedTest.cpp:38: Failure
[ctest] Value of: prime->isPrimeNumber(number)
[ctest]   Actual: false
[ctest] Expected: true
[ctest] [  FAILED  ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/3, where GetParam() = 15 (0 ms)
[ctest] [ RUN      ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/4
[ctest] D:\C++\GoogleTest\test\ParametterizedTest.cpp:38: Failure
[ctest] Value of: prime->isPrimeNumber(number)
[ctest]   Actual: false
[ctest] Expected: true
[ctest] [  FAILED  ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/4, where GetParam() = 18 (0 ms)
[ctest] [----------] 5 tests from PrimeTesting/PrimeNumberParameterTest (0 ms total)
[ctest]
[ctest] [----------] Global test environment tear-down
[ctest] [==========] 5 tests from 1 test suite ran. (0 ms total)
[ctest] [  PASSED  ] 3 tests.
[ctest] [  FAILED  ] 2 tests, listed below:
[ctest] [  FAILED  ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/3, where GetParam() = 15
[ctest] [  FAILED  ] PrimeTesting/PrimeNumberParameterTest.IsPrimeCheck/4, where GetParam() = 18
[ctest]
[ctest]  2 FAILED TESTS
[ctest]
[ctest]
[ctest] 0% tests passed, 1 tests failed out of 1
[ctest]
[ctest] Total Test time (real) =   0.05 sec
[ctest]
[ctest] The following tests FAILED:
[ctest]       1 - ParametterizedTest (Failed)
[ctest] Errors while running CTest
[ctest] CTest finished with return code 8
```

Gmock

Khi viết prototype hoặc test, dựa hoàn toàn vào các đối tượng thực sẽ thường không khả thi. Mock object sử dụng một interface như một đối tượng thực, nhưng cho phép bạn chỉ định tại runtime nó sẽ được sử dụng như thế nào và nó phải làm gì (các phương thức nào sẽ được gọi? Theo thứ tự nào? Như thế nào? nhiều lần? với những đối số nào? chúng sẽ trả về những gì? vv).

- Object interface sẽ được thử nghiệm sau đây là Gumball Machine

```
class GumballMachineInterface
{
    public:
        virtual void insertQuarter() = 0;
        virtual void ejectQuarter() = 0;
        virtual void turnCrank() = 0;
        virtual void dispenseGumball() = 0;
        virtual bool isWin() = 0;
        virtual int getGumballs() = 0;
};
```

- Khởi tạo MockClass, MockClass sẽ kế thừa Interface trên
- Trong marco MOCK_METHOD, tham số đầu tiên là kiểu dữ liệu trả về, tiếp đến là tên method, kiểu tham số truyền vào, và cuối cùng là đặc tính method. Bởi vì kế thừa 1 interface, các MockGumball bắt buộc phải có đủ các method của interface

```
class MockGumball : public GumballMachineInterface
{
    public:
        MOCK_METHOD(void, insertQuarter, (), (override));
        MOCK_METHOD(void, ejectQuarter, (), (override));
        MOCK_METHOD(void, turnCrank, (), (override));
        MOCK_METHOD(void, dispenseGumball, (), (override));
        MOCK_METHOD(bool, isWin, (), (override));
        MOCK_METHOD(int, getGumballs, (), (override));
};
```

Uninteresting và Unexpected Call

A call `x.Y(...)` is **uninteresting** if there's ***not even a single*** `EXPECT_CALL(x, Y(...))`.

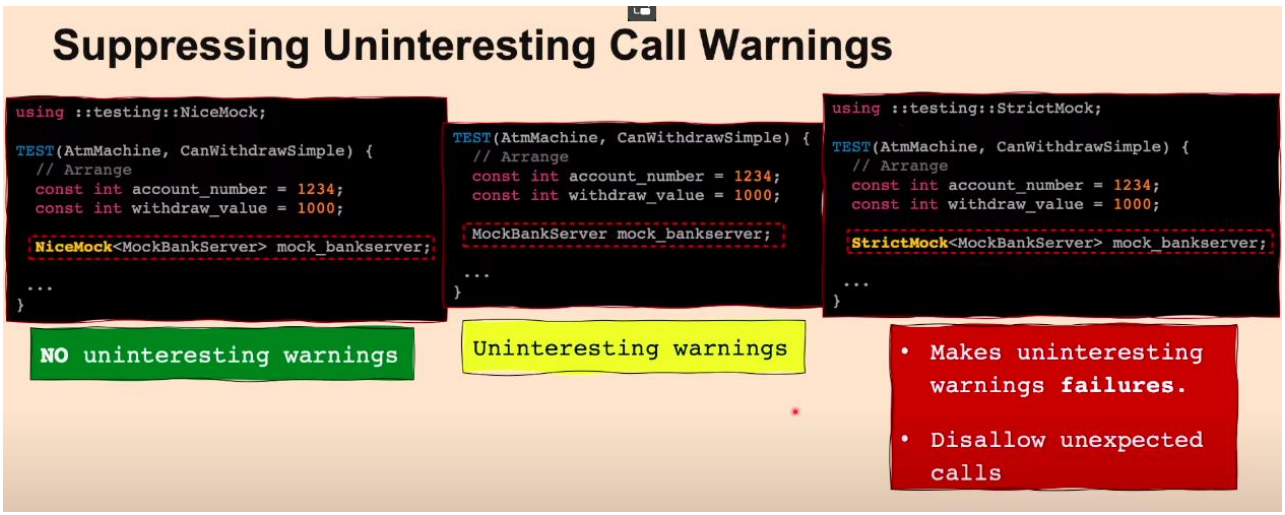
A call `x.Y(...)` is **unexpected** if there are ***some*** `EXPECT_CALL(x, Y(...))` set, but none of them matches the call.

Ví dụ cho 1 uninteresting call

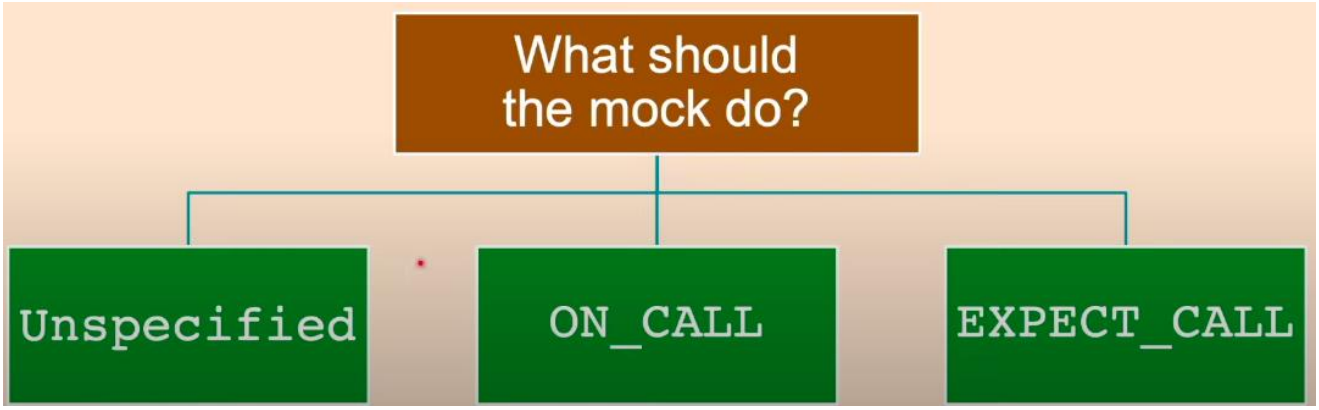
```
GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: insertQuarter()
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: turnCrank()
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress it by blindly adding an EXPECT_CALL() if you don't mean to enforce the call.
```

NiceMock hoặc StrictMock



Các hành động của MOCK



- Nếu unspecified thì mock sẽ không verify gì cả, và sẽ trả về các giá trị mặc định nếu được call
- Với ON_CALL thì mock sẽ thực hiện các hành động được đặt ra nhưng sẽ không verify

```
ON_CALL(mockGumball, getGumballs()).WillByDefault(Return(10));
```

- Với EXPECT_CALL, mock sẽ thực hiện các hành động được đặt ra và sẽ verify

```
    EXPECT_CALL(mockGumball, turnCrank()).Times(AtLeast(10));
    EXPECT_CALL(mockGumball, dispenseGumball()).Times(AtLeast(10));
    EXPECT_CALL(mockGumball,
getGumballs()).Times(AtLeast(10)).WillRepeatedly(ReturnRoundRobin({7,6,5,4,3,2,1,0}));
    EXPECT_CALL(mockGumball,
isWin()).Times(AtLeast(10)).WillRepeatedly(ReturnRoundRobin({false, true, false, false}));
    EXPECT_CALL(mockGumball, dispensePrize()).Times(AtLeast(1));
    EXPECT_CALL(mockGumball, ejectQuarter()).Times(AtLeast(1));
```

Setting Expectations and Behaviour

```
EXPECT_CALL(mockGumball, getGumballs()).Times(AtLeast(10)).WillRepeatedly(Return(0));
```

Expectation là những điều mà mong muốn sẽ diễn ra khi mock, Times(AtLeast(10)) nghĩa là mong đợi rằng method này sẽ được gọi ít nhất 10 lần. Behaviour sẽ là những hành động sẽ xảy ra khi method ấy được gọi, WillRepeatedly(Return(0)) sẽ liên tục trả về giá trị 0

Times

Specifies how many times the mock function call is expected.

The parameter *`cardinality`* represents the number of expected calls and can be one of the following, all defined in the `::testing` namespace:

Cardinality	Meaning
`AnyNumber()`	The function can be called any number of times.
`AtLeast(n)`	The function call is expected at least *n* times.
`AtMost(n)`	The function call is expected at most *n* times.
`Between(m, n)`	The function call is expected between *m* and *n* times, inclusive.
`Exactly(n)` or `n`	The function call is expected exactly *n* times. If *n* is 0, the call should never happen.

```
EXPECT_CALL(mockGumball, dispensePrize()).Times(AtLeast(1));
EXPECT_CALL(mockGumball, ejectQuarter()).Times(AtLeast(1));
```

Kết quả Test cho Times

```
[ctest] Actual function call count doesn't match EXPECT_CALL(mockGumball, getGumballs())...
[ctest]          Expected: to be called at least 10 times
[ctest]            Actual: called 8 times - unsatisfied and active
```

Action

Action chỉ định những gì Mock Function phải làm khi được gọi.

[Actions Reference | GoogleTest]

Returning a Value

Chỉ định giá trị sẽ được trả về khi Mock Function ấy được gọi

Action	Description
`Return()`	Return from a `void` mock function.
`Return(value)`	Return `value`. If the type of `value` is different to the mock function's return type, `value` is converted to the latter type *at the time the expectation is set*, not when the action is executed.
`ReturnArg<N>()`	Return the `N`-th (0-based) argument.
`ReturnNew<T>(a1, ..., ak)`	Return `new T(a1, ..., ak)`; a different object is created each time.
`ReturnNull()`	Return a null pointer.

Action	Description
`ReturnPointee(ptr)`	Return the value pointed to by `ptr`.
`ReturnRef(variable)`	Return a reference to `variable`.
`ReturnRefOfCopy(value)`	Return a reference to a copy of `value`; the copy lives as long as the action.
`ReturnRoundRobin({a1, ..., ak})`	Each call will return the next `ai` in the list, starting at the beginning when the end of the list is reached.

```
    EXPECT_CALL(mockGumball,
getGumballs()).Times(AtLeast(10)).WillRepeatedly(ReturnRoundRobin({7,6,5,4,3,2,1,0}));
    EXPECT_CALL(mockGumball,
isWin()).Times(AtLeast(10)).WillRepeatedly(ReturnRoundRobin({false, true, false, false}));
```

Invoke a function

Action	Description
`f`	Invoke `f` with the arguments passed to the mock function, where `f` is a callable.
`Invoke(f)`	Invoke `f` with the arguments passed to the mock function, where `f` can be a global/static function or a functor.
`Invoke(object_pointer, &class::method)`	Invoke the method on the object with the arguments passed to the mock function.
`InvokeWithoutArgs(f)`	Invoke `f`, which can be a global/static function or a functor. `f` must take no arguments.
`InvokeWithoutArgs(object_pointer, &class::method)`	Invoke the method on the object, which takes no arguments.
`InvokeArgument<N>(arg1, arg2, ..., argk)`	Invoke the mock function's `N`-th (0-based) argument, which must be a function or a functor, with the `k` arguments.

Giá trị trả về của Mock Function sẽ là giá trị trả về của function được invoke

```
EXPECT_CALL(mockGumball,
isWin()).Times(AtLeast(10)).WillRepeatedly(InvokeWithoutArgs(randomGenerator));
```

Giá trị trả về của mock method này sẽ là giá trị trả về của hàm randomGenerator

```
bool randomGenerator()
{
    int mRandom;
    mRandom = rand() % 2 + 0;
```

```
    std::cout << mRandom;
    if(mRandom == 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Matchers

Matcher dùng để so sánh đối số. Có thể sử dụng bên trong `ON_CALL()` hoặc `EXPECT_CALL()`, hoặc dùng để kiểm tra dữ liệu trực tiếp sử dụng các Marco dưới:

Macro	Description
`EXPECT_THAT(actual_value, matcher)`	Asserts that `actual_value` matches `matcher`.
`ASSERT_THAT(actual_value, matcher)`	The same as `EXPECT_THAT(actual_value, matcher)`, exc

Danh sách các Matchers:

[Matchers Reference | GoogleTest]

```
EXPECT_CALL(mockBank1, GetBalance(Gt(0))).Times(1).WillOnce(Return(5000));
    EXPECT_CALL(mockBank1, Debit(Gt(0), Gt(0))).Times(1);
    testUnit1.Withdraw(-10, -5);
```

```
Unexpected mock function call - returning default value.
    Function call: GetBalance(-10)
          Returns: 0
Google Mock tried the following 1 expectation, but it didn't match:

D:\C++\GoogleTest\test\Gmock2.cpp:57: EXPECT_CALL(mockBank1, GetBalance(Gt(0)))...
  Expected arg #0: is > 0
           Actual: -10
         Expected: to be called once
           Actual: never called - unsatisfied and active
unknown file: Failure

Unexpected mock function call - returning directly.
    Function call: Debit(-10, -5)
Google Mock tried the following 1 expectation, but it didn't match:

D:\C++\GoogleTest\test\Gmock2.cpp:58: EXPECT_CALL(mockBank1, Debit(Gt(0), Gt(0)))...
  Expected arg #0: is > 0
           Actual: -10
  Expected arg #1: is > 0
           Actual: -5
         Expected: to be called once
           Actual: never called - unsatisfied and active
```

Ví dụ:

Tạo MockObject cho máy nhả kẹo Gumball

Object cho máy nhả kẹo gumball

```
class GumballMachineInterface
{
    public:
        virtual void insertQuarter() = 0; //Nhét xu
        virtual int ejectQuarter() = 0; //Nhả xu
        virtual void turnCrank() = 0; //Vặn núm
        virtual void dispenseGumball() = 0; //Nhả kẹo
        virtual void dispensePrize() = 0; //Nhả phần thưởng
        virtual bool isWin() = 0; //Thắng?
        virtual int getGumballs() = 0; //Lấy số kẹo còn lại
};
```

```
class MockGumball : public GumballMachineInterface
{
    public:
        MOCK_METHOD(void, insertQuarter, (), (override));
        MOCK_METHOD(int, ejectQuarter, (), (override));
        MOCK_METHOD(void, turnCrank, (), (override));
        MOCK_METHOD(void, dispenseGumball, (), (override));
        MOCK_METHOD(void, dispensePrize, (), (override));
        MOCK_METHOD(bool, isWin, (), (override));
        MOCK_METHOD(int, getGumballs, (), (override));
};
```

```
class TestUnit
{
    GumballMachineInterface* mGumballMachine;
    int mGumballAmount;
    public:
    TestUnit(GumballMachineInterface* gumballMachine, int amount)  :
mGumballMachine(gumballMachine), mGumballAmount(amount){};
    ~TestUnit(){}
    bool beginTest() //Bắt đầu bài test
    {
        mGumballMachine->insertQuarter(); //Nhét xu vào
        mGumballMachine->turnCrank(); // Vặn tay cầm
        if(mGumballMachine->getGumballs() == 0) //Kiểm tra số kẹo còn lại
        {
            mGumballMachine->ejectQuarter(); //Nếu không còn thì nhả lại xu
            return false;
        }
        if(mGumballMachine->isWin()) //Kiểm tra xem có thắng không
        {
            mGumballMachine->dispensePrize(); //Nhả phần quà
        }
        mGumballMachine->dispenseGumball(); //Nhả kẹo
        return true;
```

```
    }
    bool randomGenerator() //Hàm khởi tạo random true false
    {
        int mRandom;
        mRandom = rand() % 2 + 0;
        if(mRandom == 1)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    int decreasingOrder() //Hàm tạo số kẹo giảm dần
    {
        mGumballAmount--;
        return mGumballAmount;
    }
};
```

```
TEST(GumballTest, runGood)
{
    MockGumball mockGumball;
    TestUnit testUnit1(&mockGumball, 9);
    int count = 9;
    bool result;
    EXPECT_CALL(mockGumball, insertQuarter()).Times(AtLeast(10));
    EXPECT_CALL(mockGumball, turnCrank()).Times(AtLeast(10));
    EXPECT_CALL(mockGumball, dispenseGumball()).Times(AtLeast(10));
    EXPECT_CALL(mockGumball,
getGumballs()).Times(AtLeast(10)).WillRepeatedly(Invoke(&testUnit1,
&TestUnit::decreasingOrder));
    EXPECT_CALL(mockGumball,
isWin()).Times(AtLeast(10)).WillRepeatedly(InvokeWithoutArgs(&testUnit1,
&TestUnit::randomGenerator));
    EXPECT_CALL(mockGumball, dispensePrize()).Times(2);
    EXPECT_CALL(mockGumball, ejectQuarter()).Times(1);
    while(count > 0)
    {
        result = testUnit1.beginTest();
        count--;
        EXPECT_TRUE(result);
    }

}
```

Kết quả:

```
[ctest] 1/1 Test #1: Gmock1 .........................***Failed    0.06 sec
[ctest] Running main() from D:\C++\GoogleTest\googletest\googletest\src\gtest_main.cc
[ctest] [==========] Running 1 test from 1 test suite.
[ctest] [----------] Global test environment set-up.
[ctest] [----------] 1 test from GumballTest
[ctest] [ RUN      ] GumballTest.runGood
[ctest] D:\C++\GoogleTest\test\Gmock1.cpp:90: Failure
[ctest] Mock function called more times than expected - returning directly.
[ctest]     Function call: dispensePrize()
[ctest]          Expected: to be called twice
[ctest]            Actual: called 3 times - over-saturated and active
[ctest] D:\C++\GoogleTest\test\Gmock1.cpp:97: Failure
[ctest] Value of: result
[ctest]   Actual: false
[ctest] Expected: true
[ctest] D:\C++\GoogleTest\test\Gmock1.cpp:88: Failure
[ctest] Actual function call count doesn't match EXPECT_CALL(mockGumball, getGumballs())...
[ctest]          Expected: to be called at least 10 times
[ctest]            Actual: called 9 times - unsatisfied and active
[ctest] D:\C++\GoogleTest\test\Gmock1.cpp:89: Failure
[ctest] Actual function call count doesn't match EXPECT_CALL(mockGumball, isWin())...
[ctest]          Expected: to be called at least 10 times
[ctest]            Actual: called 8 times - unsatisfied and active
[ctest] D:\C++\GoogleTest\test\Gmock1.cpp:87: Failure
[ctest] Actual function call count doesn't match EXPECT_CALL(mockGumball, dispenseGumball())...
[ctest]          Expected: to be called at least 10 times
[ctest]            Actual: called 8 times - unsatisfied and active
[ctest] D:\C++\GoogleTest\test\Gmock1.cpp:86: Failure
[ctest] Actual function call count doesn't match EXPECT_CALL(mockGumball, turnCrank())...
[ctest]          Expected: to be called at least 10 times
[ctest]            Actual: called 9 times - unsatisfied and active
[ctest] D:\C++\GoogleTest\test\Gmock1.cpp:85: Failure
[ctest] Actual function call count doesn't match EXPECT_CALL(mockGumball, insertQuarter())...
[ctest]          Expected: to be called at least 10 times
[ctest]            Actual: called 9 times - unsatisfied and active
[ctest] [  FAILED  ] GumballTest.runGood (0 ms)
[ctest] [----------] 1 test from GumballTest (0 ms total)
[ctest]
[ctest] [----------] Global test environment tear-down
[ctest] [==========] 1 test from 1 test suite ran. (0 ms total)
[ctest] [  PASSED  ] 0 tests.
[ctest] [  FAILED  ] 1 test, listed below:
[ctest] [  FAILED  ] GumballTest.runGood
[ctest]
[ctest]  1 FAILED TEST
[ctest]
[ctest]
[ctest] 0% tests passed, 1 tests failed out of 1
[ctest]
[ctest] Total Test time (real) =   0.07 sec
```

VÍ dụ 2: MockObject cho Ngân hàng

- Interface cho BankServer

```
class BankServer
{
    public:
    virtual bool Connect() = 0; //Kết nối đến server
    virtual void Disconnect() = 0; //Ngắt kết nối
    virtual int GetBalance(int accountNumber) = 0; //Kiểm tra số tiền còn lại
    virtual void Debit(int accountNumber, int amount) = 0; //Trừ tiền trong tài khoản
};
```

- Mock Object

```
class MockBankServer : public BankServer
{
    public:
        MOCK_METHOD(bool, Connect, (), (override));
        MOCK_METHOD(void, Disconnect, (), (override));
        MOCK_METHOD(int, GetBalance, (int), (override));
        MOCK_METHOD(void, Debit, (int, int), (override));
};
```

```
class AtmMachine
{
```

```
    BankServer *mBank;
    public:
    AtmMachine(BankServer *bank)
    {
        mBank = bank;
    }
    bool Withdraw(int accountNumber, int amount) //Hàm rút tiền
    {
        bool result = false;
        mBank->Connect(); //Kết nối đến server
        if(amount <= mBank->GetBalance(accountNumber)) Kiểm tra số tiền còn lại
        {
            mBank ->Debit(accountNumber, amount); .//trừ tiền
            result = true;
        }
        mBank -> Disconnect();//ngắt kết nối
        return result;
    }
};
```

```
TEST(ATMTest, withDrawMoney)
{
    MockBankServer mockBank1;
    AtmMachine testUnit1(&mockBank1);
    EXPECT_CALL(mockBank1, Connect()).Times(1);
    EXPECT_CALL(mockBank1, Disconnect()).Times(1);
    EXPECT_CALL(mockBank1, GetBalance(Gt(0))).Times(1).WillOnce(Return(5000));
    EXPECT_CALL(mockBank1, Debit(Gt(0), Lt(10000))).Times(1);
    bool result1 = testUnit1.Withdraw(4511, 200);
    bool result2 = testUnit1.Withdraw(4511, 200);
    EXPECT_TRUE(result1);
    EXPECT_TRUE(result2);
}
```

Kết quả chạy:

```
[ctest] Test project D:/C++/build
[ctest]     Start 1: Gmock2
[ctest] 1/1 Test #1: Gmock2 ..........................***Failed    0.05 sec
[ctest] Running main() from D:\C++\GoogleTest\googletest\googletest\src\gtest_main.cc
[ctest] [==========] Running 1 test from 1 test suite.
[ctest] [----------] Global test environment set-up.
[ctest] [----------] 1 test from ATMTest
[ctest] [ RUN      ] ATMTest.withDrawMoney
[ctest] D:\C++\GoogleTest\test\Gmock2.cpp:56: Failure
[ctest] Mock function called more times than expected - returning default value.
[ctest]     Function call: Connect()
[ctest]           Returns: false
[ctest]          Expected: to be called once
[ctest]            Actual: called twice - over-saturated and active
[ctest] D:\C++\GoogleTest\test\Gmock2.cpp:58: Failure
[ctest] Mock function called more times than expected - returning default value.
[ctest]     Function call: GetBalance(4511)
[ctest]           Returns: 0
[ctest]          Expected: to be called once
[ctest]            Actual: called twice - over-saturated and active
[ctest] D:\C++\GoogleTest\test\Gmock2.cpp:57: Failure
[ctest] Mock function called more times than expected - returning directly.
[ctest]     Function call: Disconnect()
[ctest]          Expected: to be called once
[ctest]            Actual: called twice - over-saturated and active
[ctest] D:\C++\GoogleTest\test\Gmock2.cpp:63: Failure
[ctest] Value of: result2
[ctest]   Actual: false
[ctest] Expected: true
[ctest] [  FAILED  ] ATMTest.withDrawMoney (0 ms)
[ctest] [----------] 1 test from ATMTest (0 ms total)
[ctest]
[ctest] [----------] Global test environment tear-down
[ctest] [==========] 1 test from 1 test suite ran. (0 ms total)
[ctest] [  PASSED  ] 0 tests.
[ctest] [  FAILED  ] 1 test, listed below:
[ctest] [  FAILED  ] ATMTest.withDrawMoney
[ctest]
[ctest]  1 FAILED TEST
[ctest]
[ctest]
[ctest] 0% tests passed, 1 tests failed out of 1
[ctest]
[ctest] Total Test time (real) =   0.06 sec
[ctest]
[ctest] The following tests FAILED:
[ctest]       1 - Gmock2 (Failed)
[ctest] Errors while running CTest
[ctest] CTest finished with return code 8
```