UNITTEST &
TEST-DRIVEN DEVELOPMENT

AI-T Trainning document

Date: 20th Feb, 2008

Nội dung

- Định nghĩa
- Sử dụng CppUnit
- Test-Driven Development
- Hiệu quả viết UnitTest theo TDD

I. Định nghĩa

- Software testing
  - **UnitTest** (**ProgrammerTest**): test các module (unit) để kiểm tra thiết kế chi tiết của nó là được thực hiện đúng.
  - **IntegrationTest:** test việc ghép nối giữa một nhóm các module
  - **SystemTest:** test toàn bộ hệ thống để xem đã đáp ứng được software requirement
  - **PerformanceTest:** kiểm tra các tham số QoS (còn được gọi là Non-functional requirements) được đưa ra khi xác đinh yêu cầu của sản phẩm.
  - **AcceptanceTest** (**CustomerTest**, **FunctionalTest**) tạo bởi end-user, customer để kiểm tra sản phẩm nhận được

I. Định nghĩa

- **Unit testing** là một thủ tục để xác định những module (unit) mã nguồn của phần mềm có hoạt động đúng hay không.
  - In procedural programming a unit may be an individual program, function, procedure, etc.,
  - In object-oriented programming, the smallest unit is a method; which may belong to a base/super class, abstract class or derived/child class.
- **UnitTest** được thực hiện bởi developers, ko phải bởi Software testers hoặc end-users.

I. Định nghĩa

- Lợi ích của UnitTest
  - Mục đích: phân chia các thành phần của ctrình và đảm bảo các thành phần thực hiện đúng
  - Dễ thay đổi code: programmer yên tâm khi sửa code (refactor) bằng việc test lại để xác định testcase nào bị lỗi → chỉnh sửa kịp thời
  - Dễ tích hợp: các module với interface rõ ràng và được đảm bảo unittest → dễ dàng cho intergrate test
  - Living document: developer có thể dựa vào testcase để biết được hệ thống có unit, hàm gì và sử dụng ntn.

I. Định nghĩa

- Hạn chế của UnitTest
  - Không phát hiện được hết các lỗi của ctrình
  - Khi thay đổi interface của module → phải sửa lại nhiều testcase → khi thiết kế testcase cần bỏ đi những code trùng lặp

II. CppUnit

- Giới thiệu:
  - CppUnit: C++ unit testing framework: giúp cho việc test tự động các C++ module (class) (unittest)
  - Được chuyển từ JUnit (Java)
  - Chạy trên Window/Unix/Solaris
- Source:http://downloads.sourceforge.net/cppunit/cppunit-1
- Install
  - Window: compile CppUnit project → create cppunit(d).lib for Release/Debug
  - Linux: tar → make → make install

Ví dụ: Cần UnitTest cho lớp Money

```
class Money {    //File: Money.h
public:
    Money( double amount, std::string currency )
        : m_amount( amount )
        , m_currency( currency ) {
    }
    double getAmount() const {
        return m_amount;
    }
    std::string getCurrency() const {
        return m_currency;
    }
    bool operator ==( const Money &other ) const {
        return m_amount == other.m_amount  && m_currency == other.m_currency;
    }
    bool operator !=( const Money &other ) const {
        return !(*this == other);
    }
    Money &operator +=( const Money &other ) {
        if (m_currency != other.m_currency) throw IncompatibleMoneyErr();
        m_amount += other.m_amount;
        return *this;
    }
private:
    double m_amount;
    std::string m_currency;
};
```

II. CppUnit

- Từ đặc tả của class Money → Cần test các method của class
  - Test hàm khởi tạo: → testConstructor()
  - Test các phép so sánh: ==, !=
    → testEqual()
  - Test phép cộng: → testAdd() (cùng loại tiền) và testAddThrow() (ko hợp lệ, ko cùng loại tiền)
- Từ các testcase trên → xây dựng code để thực hiện test (file **MoneyTest.h**, **MoneyTest.cpp** sử dụng CppUnit library)
- Thực hiện các testcase (MoneyApp.cpp)

Ví dụ: Testsuite cho lớp Money

```
//File MoneyTest.h
class MoneyTest : public CppUnit::TestFixture
{
    CPPUNIT_TEST_SUITE( MoneyTest );
    CPPUNIT_TEST( testConstructor );
    CPPUNIT_TEST( testEqual );
    CPPUNIT_TEST( testAdd );
    CPPUNIT_TEST_EXCEPTION(testAddThrow,
                           IncompatibleMoneyErr);
    CPPUNIT_TEST_SUITE_END();

public:
    void setUp();
    void tearDown();
    void testEqual();
    void testAdd();
    void testAddThrow();
    void testConstructor();
};
```

Ví dụ: Testsuite cho lớp Money

```
//File MoneyTest.cpp
// Registers the fixture into the 'registry'
CPPUNIT_TEST_SUITE_REGISTRATION( MoneyTest );

void MoneyTest::setUp() {
    //Code để khởi tạo các biến, module, … trước khi testsuite được chạy
}

void MoneyTest::tearDown() {
    //Code để giải phóng các biến, module, … sau khi testsuite được chạy
    xong
}

void MoneyTest::testConstructor() {                    //1st testcase
    // Set up
    const std::string currencyFF( "FF" );
    const double longNumber = 12345.90;

    // Process: hàm khởi tạo
    Money money( longNumber, currencyFF );

    // Check: kết quả của testcase
    CPPUNIT_ASSERT_EQUAL( currencyFF, money.getCurrency() );
    CPPUNIT_ASSERT_EQUAL( longNumber+1, money.getAmount() );          //FAIL
}
```

Ví dụ: Testsuite cho lớp Money

```
//File MoneyTest.cpp (cont…)

void MoneyTest::testEqual() {                    //2nd testcase
    // Set up
    const Money money123FF( 123, "FF" );
    const Money money123USD( 123, "USD" );
    const Money money12FF( 12, "FF" );
    const Money money12USD( 12, "USD" );

    // Process & Check
    CPPUNIT_ASSERT( money123FF == money123FF ); // ==
    CPPUNIT_ASSERT( money12FF != money123FF );  // !=
    amount
    CPPUNIT_ASSERT( money123USD != money123FF );// !=
    currency
    CPPUNIT_ASSERT( money12USD != money123FF ); // !=
    currency and != amount
}
```

Ví dụ: Testsuite cho lớp Money

```
//File MoneyTest.cpp (cont…)
void MoneyTest::testAdd() {       //3rd testcase
    // Set up
    const Money money12FF( 12, "FF" );
    const Money expectedMoney( 135, "FF" );

    // Process
    Money money( 123, "FF" );
    money += money12FF;

    // Check
    // += works
    CPPUNIT_ASSERT_EQUAL( expectedMoney.getAmount(), money.getAmount() );
    // += returns ref. on 'this'.
    CPPUNIT_ASSERT(&money == &(money += money12FF) );
}

void MoneyTest::testAddThrow() { //4th testcase
    // Set up
    const Money money123FF( 123, "FF" );

    // Process
    Money money( 123, "USD" );
    printf("before test");
    money += money123FF;          // should throw an exception
    printf("after test");
}
```

Ví dụ: Tìm để chạy các testcase

```
//File MoneyApp.cpp

int main(int argc, char* argv[])
{
    // Get the top level suite from the registry
    CppUnit::Test *suite =

    CppUnit::TestFactoryRegistry::getRegistry().makeTest();

    // Adds the test to the list of test to run
    CppUnit::TextUi::TestRunner runner;
    runner.addTest( suite );

    // Change the default outputter to a compiler error format
    outputter
    runner.setOutputter( new CppUnit::CompilerOutputter(
    &runner.result(), std::cerr ) );
    // Run the tests.
    bool wasSucessful = runner.run();

    // Return error code 1 if the one of test failed.
    return wasSucessful ? 0 : 1;
}
```

II. CppUnit

- Configure:
  - VC++ project: link to cppunit.lib and run app when post-compile
  - Linux: compile objects and link with flags:

    -ldl –lcppunit

    (có thể cần

    export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH)
- **Note**: có thể dùng CppUnit để test C module. Khi đó phải dùng C++ compiler để dịch cho C module.
- Kết quả test:
  - Linux: chạy file để thực hiện test
  - VC++: tự chạy sau khi dịch (post-compile) hoặc chạy trực tiếp chương trình

II. CppUnit

- Ví dụ: Kết quả test (trên Linux và Window đều giống nhau)

```
1>.F...before test
1>..\..\..\src\MoneyTest.cpp(25) : error :
   Assertion
1>Test name: MoneyTest::testConstructor
1>equality assertion failed
1>- Expected: 12346.9
1>- Actual  : 12345.9
1>Failures !!!
1>Run: 4   Failure total: 1   Failures: 1
   Errors: 0
```

II. CppUnit

- Một số macro được sử dụng
  - CPPUNIT_ASSERT(condition)
    Assertions that a condition is true.
  - CPPUNIT_ASSERT_MESSAGE(message, condition)
    Assertion with a user specified message.
  - CPPUNIT_FAIL(message)
    Fails with the specified message
  - CPPUNIT_ASSERT_EQUAL(expected, actual)
    Asserts that two values are equals.
  - CPPUNIT_ASSERT_EQUAL_MESSAGE(message, expected, actual)
    Asserts that two values are equals, provides additional message on failure
  - CPPUNIT_ASSERT_DOUBLES_EQUAL(expected, actual, delta)
    Macro for primitive value comparisons.

III. Test-Driven Development

- Khái niệm
  - CppUnit là ví dụ một framework hỗ trợ cho việc xây dựng unittest
  - Quy trình áp dụng cho việc xây dựng unittest cho module:
    - Truyền thống: xây dựng xong module rồi viết unittest
    - TDD: viết test trước rồi viết code song song
  - Test-Driven Development/Design (TDD):

    (Beck 2003; Astels 2003), is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring.

III. Test-Driven Development

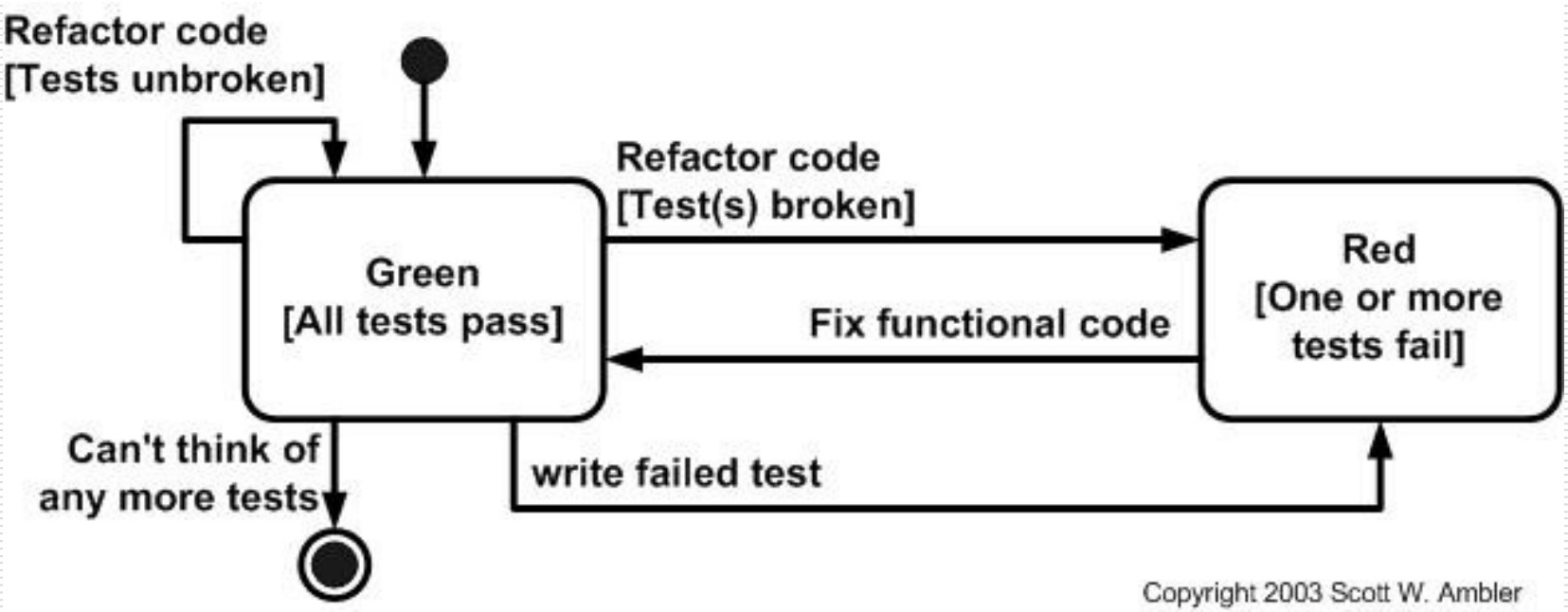
- Yêu cầu: phải hỗ trợ automatic unittest (sử dụng xUnit – unit test framework – ví dụ CppUnit) → cần xác định rõ module requirement trước
- Test-Driven Development Cycle
  - **Add a test**
  - **Run all tests and see the new one fail**
  - **Write some Code (for module)**
  - **Run the automated tests and see them succeed**
  - **Refactor code**
  - **Repeat cycle**

III. Test-Driven Development

- Development style
  - principles of "Keep It Simple, Stupid" (KISS) →viết nhưng function của module và code của testcase càng đơn giản càng tốt
  - Yêu cầu programmer “first fail the test cases” để đảm bảo testcase thực sự chạy
  - repeats the steps of adding test cases that fail, passing them, and refactoring → **incremental changes**

III. Test-Driven Development

- Development style



III. Test-Driven Development

- Ưu điểm
  - Thời gian viết test thì nhiều, tương đương thời gian code module nhưng sẽ ít hơn thời gian phải debug theo các test truyền thống
  - Mặc dù theo phương pháp này mọi thứ được đơn giản hoá và phát triển dần dần nhưng thực tế vẫn có thể áp dụng cho nhiều hệ thống lớn
  - Đảm bảo hệ thống luôn chạy, luôn hoạt động tại mọi thời điểm → gần như ít phải debug
  - Dễ dàng phát hiện được lỗi (ko đáp ứng đúng requirement) mỗi khi thay đổi code của module
  - Làm nền tảng cho việc thiết kế theo E**x**treme **P**rogramming
- Nhược điểm: khó trong một số tình huống như GUI, DB,… mà hệ thống với đầu vào và ra phức tạp không được thiết kế cho việc isolated unit testing và refactoring

IV. Hiệu quả viết UnitTest theo TDD

- Thiết kế các module để dễ test → dễ với C++ vì module nhỏ nhất là class
- Luôn viết test trước khi viết function của module

  TFD - Test First Development;

  TDD=TFD + refactoring
- Viết testcode và module function dạng empty để tạm thời pass test. Sau đó viết dần dần code cho function (refactor) rồi lại test ngay → khoảng thời gian giữa 2 hành động này là không quá 10phút → quá trình này phải lặp đi lặp lại
- Nên viết testcase mới mỗi khi thêm vài dòng code
- Với mỗi hàm mới update (hoặc clean-up), cần viết ngay testcase và test nhanh nhất có thể.
- Không nên test 1 chuỗi các hàm của 1 object trong cùng 1 testcase
- Test trường hợp bình thường, đặc biệt và trường hợp cố tình gây lỗi

IV. Hiệu quả viết UnitTest theo TDD

- Refactor (sửa code và bỏ đi những code trùng lặp): cần áp dụng cho cả module function và testcase code → tối ưu và dễ bảo trì
- Không được xoá và sửa những testcase đã pass
- Cố gắng viết đủ testcase để test được 100% các dòng code của module (→ if, else, …)
- Nguyên tắc KISS trong viết UnitTest: trong unittest không nên có if, switch hoặc lệnh so sánh logic vì như thế là ta đã test nhiều trường hợp trong 1 testcase → khó đọc, khó bảo trì, dễ gây bug ngay trong testcase
- Mỗi 1 class nên có 1 testsuite riêng (ko nên dùng chung cho các class → dễ sang intergrate test)

IV. Hiệu quả viết UnitTest theo TDD

- Các testsuite (file) nên đặt riêng ra, độc lập code với module
- Nên sử dụng 25-50% thời gian để viết test với lượng code viết testcase khoảng 50% lượng code module
- Có thể lấy code của testcase làm example trong tài liệu kỹ thuật của module
- Sau thời gian dài, số testcase sẽ nhiều → thời gian chạy lớn → chia ra làm các testcase mới và cũ. Testcase cũ sẽ chạy với tần suất ít hơn
- Make Test Easy to Run: testcase nên chia làm 2 loại:
  - Loại1: đơn giản, ko có yêu cầu đặc biệt để chạy
  - Loại2: cần phải thiết lập môi trường hoặc có yêu cầu gì đo

  Không nên đặt 2 loại trong cung 1 testsuite. Loại 2 nên đặt riêng ra để developer khác khi có thời gian họ sẽ đọc kỹ tài liệu để test loại này

IV. Hiệu quả viết UnitTest theo TDD

- Viết unittest để dễ bảo trì
  - Không nên test các thành phần private/protected → sử dụng black box hoặc gray box
  - Không nên để code trùng lặp giữa các testcase vì khi thay đổi interface của 1 module function thì lại phải thay đổi hết trong các testcase → kỹ thuật factory: viết riêng function cho đoạn code trùng lặp này
  - Các testcase không nên phụ thuộc vào nhau cả về data và thứ tự thực hiện.
  - Không được gọi 1 testcase khác trong 1 testcase
  - Không nên có nhiều Assert trong 1 testcase vì khi 1 điều kiện không thảo mãn thì các Assert khác sẽ bị bỏ qua → nên viết thành nhiều testcase riêng

IV. Hiệu quả viết UnitTest theo TDD

- Viết unittest để dễ đọc, hiểu
  - Có comment khi Assert (ktra kq test)
  - Tên testcase cần đặt dễ hiểu nhất (có thể tên dài cũng được) → UnitTest như là tài liệu kỹ thuật API của module
  - Sử dụng macro Assert có message đi kèm
- Trong phần init của testsuite, chỉ sử dụng các biến global cũng như khởi tại những object được sử dụng bởi tất cả các testcase. Ko nên tạo biến global mà chỉ sử dụng bởi một số testcase → sử dụng kỹ thuật factory để khắc phục

IV. Hiệu quả viết UnitTest theo TDD

- Những điều cần tránh (TDD Anti-Pattern)
  - Chỉ test những dữ liệu phù hợp nhất
  - Hạn chế những testcase mà cần phải tạo phức tạp về môi trường → dễ gây khó hiểu, ko biết test cái gì
  - Hạn chế nhiều testcase ko cần thiết → chạy lâu.
  - Không nên viết quá nhiều dòng code cho 1 testcase (ko nên quá 10 dòng)
  - Việc test 100% module → cần hiểu được code bên trong module (white box) → dễ dẫn đến việc phải sửa lại testcase khi refactoring → TDD ko thể dùng với write box mà sử dụng gray box thay thế
  - Các testcase có thể bị ảnh hưởng bởi nhau do sử dụng static data trong module
  - Không nên viết tên testcase như test1, test2, … vì khiến cho developer khác phải dựa vào testcase code để biết nó làm gì
  - Ko nên tích hợp nhiều testcase vào 1 → testcase càng đơn giản càng tôt

IV. Hiệu quả viết UnitTest theo TDD

- Kết luận: Thế nào là một UnitTest tốt
  - Chạy nhanh
  - Chạy độc lập giữa các testcase (ko phụ thuộc thứ tự test)
  - Sử dụng data dễ đọc, dễ hiểu
  - Sử dụng dữ liệu thực tế có thể
  - Testcase đơn giản, dễ đọc, dễ bảo trì
  - Phản ánh đúng hoạt động của module

Tham khảo

- **Test Driven Development with Visual Studio 2005 Team System**

  http://www.dotnetjunkies.com/Tutorial/9709CE8B-0986-46D2-AE3B-598
- **Test-driven development**
  http://en.wikipedia.org/wiki/Test-driven_development
- **Unit Test**
  http://en.wikipedia.org/wiki/Unit_testing
- **Instruction to test driven development**
  http://www.agiledata.org/essays/tdd.html
- **Simple Smalltalk Testing:With Patterns**
  http://www.xprogramming.com/testfram.htm
- **The Three Rules of TDD**
  http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd
- **TTD Anti-pattern**
  http://blog.james-carr.org/?p=44
- **Unit Testing Tips: Write Maintainable Unit Tests That Will Save You Time And Tears -- MSDN Magazine, January 2006:**

  http://msdn.microsoft.com/msdnmag/issues/06/01/UnitTesting