Unit Test

Investigation

Project Team

Team 3

Date

2016-10-30

Team Information

201211335김윤식

201311281송종원

201411317조민규

201511282이재승

1 UnitTest란 무엇인가?

1.1 UnitTest란?

C 언어에서의 단위 테스트란 함수 수준에서 정확히 원하는 목적으로 함수가 만들어졌는지를 확인하는 테스트를 말한다. 즉, 코딩된 함수가 제대로 실행되는지를 확인하는 것이다. 원시 코드를 시험대상으로 하며 단위 시험을 수행한다. UnitTest 는 크게 BlackBox Test 와 WhiteBox Test 를 포함한다. BlackBox Test 는 말 그대로 내부 구현에는 관심을 두지 않으며, 있어야 할 기능이 정확하게 모두 구현되어 있는지에 초점을 두며, 모든 기능을 테스트 입력으로 만들어야 한다. WhiteBox Test 는 개발대상의 가능한 모든 경우를 시험해 보았는지, CFG(Control Flow Graph) 나 DFG(Data Flow Graph)를 따라서 올바르게 구현하였는지에 중점을 두며, 특정 Converage Criteria 를 만족하는 테스트 입력을 만들어야 하며 BlackBox Test 와는 달리 기능의 구현 여부는 관심에 두지 않는다. BlackBox Test 와 WhiteBox Test 를 정리하면 아래의 그림과 같다.

블랙박스 테스트

기능 테스트 (Functional Test)

있어야 할 기능이 정확하게 구현되어 있는가?
기능명세서, 요구사항명세서, 사용자설명서
모든 기능을 테스트 입력으로 만들어야
내부 구현은 관심 無

화이트박스 테스트

구조 테스트 (Structural Test)

구현 프로그램의 가능한 모든 경우(Execution Path)를 시험해 봤는가?
CFG (Control Flow Graph) , DFG (Data Flow Graph)
특정 Coverage Criteria를 만족하는 테스트 입력을 만들어야
기능의 구현 여부는 관심 無

1.2 UnitTest의 현주소

오늘날에 대부분의 개발자는 단위 테스트가 필요하다는 것에 동의하지만, 실제로 단위 테스트를 실행하는 개발자는 드물다. 프로젝트의 요구사항을 분석하여 그를 토대로 개발한다고 하여도 절대 완벽한 요구사항분석을 할 수 없으며, UnitTest를 수행하기 위해서는 TestCode를 작성해야 하는데 TestCode까지 개발할 정도로 시간이 넉넉하지 않으며, 개발이 진행중인 상황에서도 잦은 요구사항의 변경이 요청되기 때문이다. 그로 인해 현재의 개발 방식은 과거와 많이 달라졌으며, 코딩이 일어나는 시점과 테스트 시점 사이의 간격을 가능한 짧게 가져가는 것이 최근 추세이다. 즉, 오류가 발생하는 시점에 수정을 한다는 것이 원칙이다. UnitTest를 시행할 때에는 BlackBox Test보다는 주로 WhiteBox Test로 시행한다.

1.3 UnitTest의 사용방법

기본적인 테스트 단위는 함수이다. 즉, 하나의 함수를 다른 함수가 호출해서 그 함수가 올바른지 검증하는 것을 단위 테스트라고 말한다. 호출하는 함수에서는 호출되는 함수의 입력 값과 출력 값을 예상하게 되며, 예측된 값이 제대로 출력되지 않는다면 테스트는 실패했다고 본다. 테스트의 대상이 되는 코드를 SUT(Software Under Test or Source Under Test)라고 하며, 호출하고 검사하는 측을 Test Case라고 한다. 단위 테스트를 실행하기 위해서는 단위 테스트를 동일한 특정 기능에 한정된 묶음으로 테스트 케이스들을 모을 수 있는데, 이를 Test Group이라고 한다. 하나의 Test Group에는 여러 개의 단위 테스트 케이스들이 들어가며, 각각 다른 이름으로 표현된다. 하나의 단위 테스트 그룹에는 설정(Setup)과 복구(Tear-Down)이 각각의 테스트 케이스에 대해서 반복적으로 실행된다. 테스트 케이스의 실행에 필요한 초기 조건을 생성하고, 케이스 간의 의존성을 없애기 위해서 복구 절차를 만든다. 또한 대체 함수 역할을 하는 Mock에 호출해야할 함수의 이름과 그 파라미터, 리턴 값들을 기록해놓고, 대체 함수의 호출이 발생하면 기록된 기대값들을 불러와서 테스트 중인 함수로 돌려주게 된다. 테스트 케이스에서 Mock의 기대값을 설정하고, 이후에 원하는 기대값들이 나왔는지 확인하는 과정을 거친다.

1.4 Testing의 종류

블랙 박스 테스팅(Black box testing) – 시스템의 내부 설계(Internal system design)는 이 테스팅 유형에서 고려할 대상이 아니다. 테스트는 요구사항(Requirement) 및 기능성(Functionality)에 기반해서 이루어진다.

화이트 박스 테스팅(White box testing) – 이 테스팅은 애플리케이션의 코드 내부의 로직(Logic)에 대한 지식을 기반으로 수행된다. 글래스 박스(유리상자 혹은 투명상자) 테스팅으로도 알려져 있다. 이 유형의 테스팅을 수행하기 위해서는 내부적으로 소프트웨어와 코드가 어떻게 동작하는지를 알고 있어야만 한다. 화이트 박스 테스트는 코드구문(Statements), 분기(Branches) 경로(Paths), 조건(Conditions) 커버리지 등으로 분류할 수 있다.

유닛 테스팅(Unit testing) – 각각의 소프트웨어 컴포넌트나 모듈 대상 테스팅을 의미한다. 일반적으로 테스터가 아니라 프로그래머에 의해 수행되며, 이를 수행하기 위해서는 프로그램 내부에서 수행되는 코드와 프로그램 설계에 대해 매우 해박한 지식을 가지고 있어야 한다. 테스트 드라이브 모듈(Test drive modules)이나 테스트 하네스(Test harnesses) 개발이 필요할 수도 있다.

점진적인 통합 테스팅(Incremental integration testing) – 바텀업(Bottom up) 방식의 테스팅. 예를 들어, 애플리케이션에 새로운 기능이 추가되는 것에 대해 지속적으로 이어지는 테스팅과 같은 것이다. 애플리케이션의 기능성과 모듈은 이미 각각 충분히 테스트 되어있는 상태여야 한다. 프로그래머 혹은 테스터에 의해 수행된다.

통합 테스팅(Integration testing) – 통합 이후에 결합된 기능들을 검증하기 위한 통합 모듈 테스팅. 여기서 모듈은 일반적으로 코드 모듈, 개별 애플리케이션, 네트워크 상의 클라이언트와 서버 애플리케이션 등이 될 수 있다. 이 유형의 테스팅은 특히 클라이언트/서버 및 분산 환경 시스템에 적절하다.

기능 테스팅(Functional testing) – 이 유형의 테스팅은 내부적인 부분을 무시하고 결과값이 요구사항대로 나왔는지, 혹은 그렇지 않은지에 초점을 맞춘다. 블랙박스 타입의 테스팅이 애플리케이션의 기능 요구사항 검증에 적합하다.

시스템 테스팅(System testing) – 각각의 요구사항에 대해 전체 시스템이 테스트된다. 전체 요구사항 명세에 기반한 블랙 박스 타입의 테스팅으로 모든 조합 가능한 시스템의 부분들을 커버한다.

엔드-투-엔드 테스팅(End-to-end testing) – 시스템 테스팅과 유사하며, 데이터베이스와 네트워크 커뮤니케이션의 사용,혹은 다른 종류의 하드웨어, 애플리케이션, 혹은 시스템에 대한 상호작용과 같은 실제 사용자 환경을 모방한 환경에서 사용되는 모든 애플리케이션에 대한 테스팅을 포함한다.

새너티 테스팅(Sanity testing) – 새로운 소프트웨어 버전이 주요 테스팅 업무를 수행하기에 충분히 적합한가를 판단하기 위해 수행하는 테스트. 만약 애플리케이션에서 사용 초기에 크래시(Crash)가 발생한다면, 시스템은 더 이상의 테스팅을 수행할 정도로 충분히 안정적이라고 말할 수 없으며, 빌드 혹은 애플리케이션은 이 부분을 수정해야 한다.

리그레션 테스팅(Regression testing) – 애플리케이션의 모든 모듈 및 기능에 대한 수정 사항을 테스팅 하는 것. 리그레션 테스팅에서 모든 시스템을 커버하는 것은 무척 어려운 일이므로 일반적으로 이러한 유형의 테스팅에는 자동화 테스팅이 사용된다.

인수 테스팅(Acceptance testing) – 일반적으로 이 유형의 테스팅은 시스템이 고객이 명세한 요구사항을 충족했는지를 검증하기 위해 사용된다. 사용자 혹은 고객이 애플리케이션을 인수(Accept)할 것인지를 결정하기 위해 수행한다.

부하 테스팅(Load testing) – 어느 지점에서부터 시스템의 반응 시간이 지연되거나(Degrades), 혹은 반응이 실패하는지를 알아보기 위해 부하의 범위 안에서 웹 사이트를 테스트 하는 것과 같은, 부하가 걸리는 상황 하에서 시스템의 동작을 검사하기 위해 수행하는 일종의 퍼포먼스 테스트.

스트레스 테스팅(Stress testing) – 명세에서 허용된 것 이상의 스트레스를 가해서 어떻게 그리고 언제 시스템에서 장애가 발생하는지를 체크하기 위한 테스트. 저장 용량을 초과하는 데이터를 저장하거나, 복잡한 데이터베이스 쿼리를 입력하거나, 시스템에 지속적으로 입력값을 입력하거나 혹은 데이터베이스에 부하를 거는 것과 같은 심각한 부하를 주는 테스트를 수행한다.

퍼포먼스 테스팅(Performance testing) – '스트레스' 혹은 '부하' 테스팅과 종종 혼용되어 사용되는 단어. 시스템이 퍼포먼스 요구사항을 충족하는지 검증하는 행위이다. 이를 위해 각기 다른 퍼포먼스와 부하 툴을 사용한다.

사용성 테스팅(Usability testing) – 사용자 친화적(User-friendliness)인지를 점검하는 것. 애플리케이션의 플로우와 신규 사용자들이 쉽게 애플리케이션을 이해할 수 있는지, 사용자가 원하는 어떤 시점에서도 적합한 도움말이 제공되는지 등이 테스트된다.

설치/삭제 테스팅(Install/uninstall testing) – 각기 다른 하드웨어와 소프트웨어 환경 및 다른 OS 하에서 전체, 부분, 혹은 업그레이드 설치/삭제 프로세스를 테스트한다.

회복 테스팅(Recovery testing) – 크래쉬, 하드웨어 장애 혹은 다른 심각한 문제들로부터 시스템이 어떻게 복구되는지를 테스트 하는 것

보안 테스팅(Security testing) – 해킹이 시스템을 뚫고 들어갈 수 있는지를 검증하는 것. 인가받지 않은 내부 혹은 외부의 액세스로부터 시스템이 어떻게 스스로를 방어하는지에 대해 테스트한다. 외부 공격으로부터 시스템, 데이터베이스가 안전한지를 체크한다.

호환성 테스팅(Compatibility testing) – 특정한 하드웨어/소프트웨어/OS/네트워크 환경 및 각기 다른 조합 하에서 소프트웨어가 어떻게 동작하는지를 테스트한다.

비교 테스팅(Comparison testing) – 앞서 출시된 제품 혹은 유사한 제품과 비교해 제품의 장단점을 비교함

알파 테스팅(Alpha testing) – 이 유형의 테스팅을 위해 사내에서 가상 유저 환경이 조성될 수 있다. 개발의 마지막 부분에서 이 테스트가 수행된다. 이 테스팅의 결과로 사소한 디자인 변경 등이 이루어 질 수 있다.

베타 테스팅(Beta testing) – 일반적으로 엔드 유저에 의해 완료되는 테스팅. 상용화를 위한 애플리케이션 릴리즈 이전의 최종 테스팅이다.

1.5 Smoke Testing VS Sanity Testing

1.5.1 Sanity Testing

Smoke Testing은 Software를 빌드한 후에 프로그램의 핵심 기능들이 잘 수행되는지 확인 하기 위해 수행하는 테스트이다. 이것은 모든 상세한 기능이나 회귀 테스트가 SoftWare 빌드에서 실행 되기 전에 실행된다. Smoke Testing을 하는 이유는 심하게 손상된 App을 불량으로 처리하기 위함이다. 그로 인해 Quality Assurance 팀이 소프트웨어 App을 설치하고 테스팅하는데 시간을 낭비하지 않도록 한다. Smoke Testing에서, 선택된 테스트 케이스들은 가장 중요한 기능이나 시스템의 구성요소를 포함한다. 이것은 엄격한 테스팅은 수행하지 않지만, 시스템의 핵심 기능들이 잘 수행되는 것을 확인하기 위한 목적이 있다. Ex) 어플리케이션이 성공적으로 시작되는지 확인하는 것, GUI가 즉각 반응하는지 체크(check)하는 것

1.5.2 Sanity Testing

Sanity Testing은 코드나 기능의 사소한 변경에 대하여 소프트웨어 빌드를 한 후에 버그가 수정되었고 이러한 변경이 문제를 유입하지 않음을 확인하기 위해 수행하는 활동이다. 이것의 목표는 제안된 기능이 어느정도 예상대로 동작하는지 확정하는 것이다. 만약 Sanity Testing이 실패하는 경우, 더 엄격한 테스팅과 관련된 시간과 비용을 절약하기 위해 그 빌드는 거부된다. Sanity Testing의 목적은 새로운 기능을 철저하게 입증하기 위함이 아니라, 개발자가 Software를 생산하는 동안 일부 합리적인 행동으로 적용했는지를 확정하는 것이다. Ex) 공학용 계산기를 테스팅 할 때 sin 30 + cos 50 같은 고급 기능보다는 2 + 2 = 5라는 결과를 제공하는가 체크

1.5.3 Smoke Testing VS Sanity Testing

Smoke Testing vs Sanity Testing - 핵심 차이점

Smoke Testing	Sanity Testing
Smoke Testing은 프로그램의 핵심 기능이 잘 동작하는 것을 확인하기 위해 수행한다.	Sanity Testing은 새로운 기능/버그가 수정되었는지 확인하기 위해서 한다.
Smoke Testing의 목적은 더 엄격한 테스팅을 진행하기 위해 시스템의 "안정성(stability)"을 검증하기 위함이다.	Sanity Testing의 목적은 더 엄격한 테스팅을 진행하기 위해 시스템의 "합리성(rationality)"을 검증하기 위함이다.
Smoke Testing은 개발자나 테스터에 의해 수행된다.	Sanity Testing은 보통 테스터에 의해 수행된다.
Smoke Testing은 보통 문서화(documented)되거나 절차화(scripted)된다.	Sanity Testing은 보통 문서화되지 않고, 즉흥적(unscripted)다.
Smoke Testing은 회귀 테스팅(Regression Testing)의 부분집합이다.	Sanity Testing은 인수 테스팅(Acceptance Testing)의 부분집합이다.
Smoke Testing은 처음부터 끝까지 전체 시스템에 대해 수행한다.	Sanity Testing은 전체 시스템 구성요소들 중 특정 부분에 대해 수행한다.
Smoke Testing은 일반적인 건강 검진과 같다.	Sanity Testing은 전문적인 건강 검진과 같다.

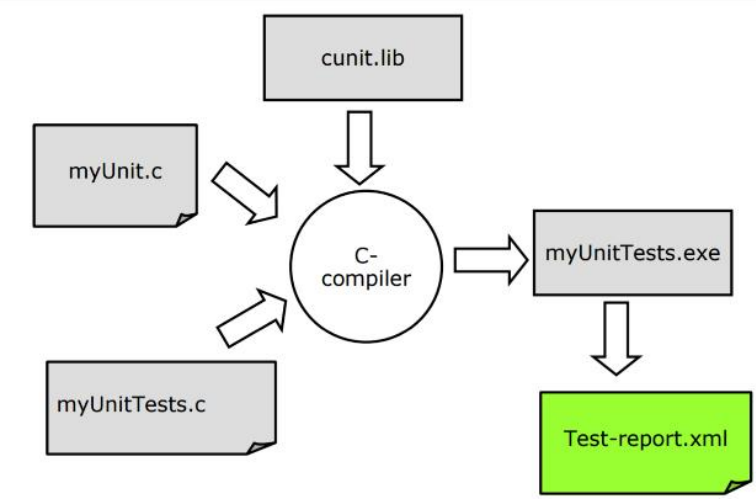
2 UnitTest의 종류

2.1 API Sanity AutoTest

C/C++ 라이브러리들을 위해 기본 유닛 테스트를 자동 생성한다. 이는 헤더 파일의 선언을 분석하여 API의 모든 함수를 위해 입력데이터 파라미터와 간단한 조합의 테스트 케이스를 생성할 수 있다. API Sanity AutoTest는 생성된 테스트는 빌드 및 실행할 수 있으며, 이를 통해 간단하게 중요한 오류의 유무를 분석할 수 있다. craches, aborts, 모든 종류의 emitted signal, non-zero 프로그램 return 코드, 프로그램 행깅(hanging) 감지가 가능하다.

2.2 C Unit Test System(CUT)

CUnit은 C언어에서 Unit test를 writing하고, administering, running하기 위한 간단한 to-the-point 유닛 테스트 시스템이다. CUnit은 유저 인터페이스의 유연한 다양성으로 기본적인 테스팅 기능을 제공한다. 사용자의 테스트 코드에 link되는 정적 library로 구축된다. 이 테스트 코드는 간단한 구조로 구축되어 있고, 테스팅을 위한 다양한 assertions으로서 common data type을 제공한다. 다른 유닛 테스트와 가장 큰 차이는 KISS 원리에 따른다는 것이다.
아래의 그림은 일반적으로 사용되는 CUnit의 testing Framework이다.
(testing framework란 Unit-test를 writing, running하기 위한 software tool이다.)



2.3 Check

C언어를 위한 유닛 테스트 라이브러리이다. 유닛 테스트 정의를 위해 간단한 인터페이스를 제공한다. autotools를 이용한 빌드 환경에서 테스트에 관련된 make target의 이름이 check인 것에서 유래했다. Check의 경우 Test는 별도의 주소 공간에서 실행된다. 그래서 Segmentation Fault 혹은 Signal Catch 같은 코드 에러를 확인할 수 있다. Check를 사용할 때 가장 큰 특징으로 볼 수 있

는 것은 각 테스트 케이스를 실행할 때, fork() 시스템 콜을 통해 별도의 프로세스를 생성하여 실행한다는 것이다. C 언어에서는 정상적인 실행 이외의 상황 (segmentation fault, exit()호출과 같이 비정상 종료) 발생 시 테스트 케이스 실행 도중 프로그램 전체가 종료되어 버릴 수 있다.
이 경우 테스트에 관련된 아무런 결과를 보고받을 수 없으므로, 이러한 문제를 극복하기 위해 분리된 별도의 프로세스에서 테스트를 수행하고, 부모 프로세스가 그 결과 및 종료 상태를 감지하여 안전하게 테스트 결과를 보고해 준다.

2.4 CMock

Testing에 있어서 테스트의 생산인 코드만이 검증 대상이지, 그것이 다른 함수를 의존하고 있어서 그 의존하고 있는 함수들까지 모두 테스트 할 수 없다. 이를 해결하기 위해서 테스트 대상이 의존하고 있는 함수들을 대체해 주어야 하는데, 이 때 사용하는 것이 테스트 더블의 일종인 Mock이다. Mock Object 는 검사하고자 하는 코드와 맞물려 동작하는 객체들을 대신하여 동작하기 위해 만들어진 객체이다. 검사하고자 하는 코드는 Mock Object의 메서드를 부를 수 있고, 이 때 Mock Object는 미리 정의된 결과 값을 전달한다. Mock Object는 자신에게 전달된 인자를 검사할 수 있으며, 이를 테스트 코드로 전달할 수도 있다. CMock의 경우에는 헤더 파일을 해석해서 필요한 Mock들을 자동으로 생성하는 기능과 그렇게 생성된 코드를 이용해 예상된 값을 테스트하고 있는 함수가 제대로 사용하고 있는지 확인하는 방식으로 코드를 검증한다. Cmock은 Ruby라는 스크립트 언어를 이용한다. 호환성에 맞춰서 받아야 하기 때문에 Github 접근을 위한 Git를 설치해야 한다 (링크 : http://github.com/throwtheswitch/cmock.git)
CMock을 다운받으면, "cmock/vendor" 디렉토리에 이미 Unity가 있기 때문에, Unity 프레임워크를 그대로 사용하면 된다. Unity를 실행하기 위해서는 main 함수와 테스트 그룹 등을 만들어야 한다. 자세한 설명은 아래의 블로그를 참고하여 진행하도록 한다.
(링크: http://blog.naver.com/knix008/220658970595)

3 C-Unit을 이용한 Testing

3.1 Source Code

3.1.1 Main.c

```
#include "CUnit/Automated.h"
#include "CUnit/Basic.h"
#include "fibonacci.h"

void simpleTest00(void) {
    CU_ASSERT_EQUAL(fibonacci(10), 55);
}

void simpleTest01(void) {
    CU_ASSERT_EQUAL(fibonacci(5), 5);
}

int main() {
    CU_pSuite pSuite = NULL;

    if(CUE_SUCCESS != CU_initialize_registry())
        return CU_get_error();
```

```
    pSuite = CU_add_suite("testing a suite", NULL, NULL);

    if(NULL == pSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    if(NULL == CU_add_test(pSuite, "simpleTest0", simpleTest00)
        || NULL == CU_add_test(pSuite, "simpleTest1", simpleTest01)) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    CU_set_output_filename("CUnit");
    CU_list_tests_to_file();
    CU_automated_run_tests();
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
}
```

3.1.2 Fibonacci.h

```
int fibonacci(int index) {
    if (index < 0) {
        return 0;
    }
    if (index > 20) {
        return -1;
    }
    if ((index == 0) || (index == 1)) {
        return 1;
    }
    return fibonacci(index - 2) + fibonacci(index - 1);
}
```

3.2 Testing Screen

test.c (~) - VIM

```c
#include "CUnit/Automated.h"
#include "CUnit/Basic.h"

#include "fibonacci.h"

void simpleTest00(void) {
    //fibonacci(10)과 55가 일치하는지 확인
    CU_ASSERT_EQUAL(fibonacci(10), 55);
}

void simpleTest01(void) {
    CU_ASSERT_EQUAL(fibonacci(5), 5);
}

int main() {
    CU_pSuite pSuite = NULL;
    //registry 초기화
    if(CUE_SUCCESS != CU_initialize_registry())
        return CU_get_error();
    //registry에 test suite 추가
    pSuite = CU_add_suite("testing a suite", NULL, NULL);

    if(NULL == pSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }
    //suite에 test cases add
    if(NULL == CU_add_test(pSuite, "simpleTest0", simpleTest00)
        || NULL == CU_add_test(pSuite, "simpleTest1", simpleTest01)) {
        CU_cleanup_registry();
        return CU_get_error();
    }
    //make mxl
    CU_set_output_filename("CUnit");
    CU_list_tests_to_file();
    CU_automated_run_tests();
    //basic run
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();

    CU_cleanup_registry();

    return CU_get_error();
}
```

test.c

```
Jong_Won@SongJongWon ~
$ gcc test.c -o test.exe -L/usr/local/lib -lcunit

Jong_Won@SongJongWon ~
$ ./test


    CUnit - A unit testing framework for C - Version 2.1-3
    http://cunit.sourceforge.net/


Suite: testing a suite
  Test: simpleTest0 ...passed
  Test: simpleTest1 ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      2      2      2      0        0
             asserts      2      2      2      0      n/a

Elapsed time =    0.000 seconds

Jong_Won@SongJongWon ~
$
```