Software Engineering
Assignment #3 (C Unit Test)

Project Team
Team 1
201211333 김영호
201211347 박성근
201211364 이경민
201211376 임제현

Date
2016-10-30

Team Information

1 Introduction

1.1 Purpose

본 문서는 2016년 건국대학교의 소프트웨어공학 개론 강의의 Assignment#3을 설명한다. Assignment#3은 Unit Test에 대하여 조사하고, C unit Test 도구를 조사하여 사용할 도구를 선정한다.

2 Unit Test

2.1 What is Unit Test

유닛 테스트(Unit Test)는 실행 가능한 소스 코드의 특정 모듈이 의도된 대로 정확히 작동하는지 검증하기위해 코드에 대한 테스트 코드를 작성하고 이를 실행하는 것이다. 즉, 함수 단위로 테스트 코드를 작성해서, 코딩된 함수와 메소드가 제대로 실행되는지 확인하는 것이다. 이를 통해 코드 변경으로 인해 문제가 발생할 경우, 단시간 내에 이를 파악하고 바로 잡을 수 있도록 해준다. 또한, 복잡한 코드를 구성하는 기본 단위들에 대해서 복잡도를 낮추어 테스트 코드를 만들어서 검증할 수 있고, 복잡한 코드에서 문제를 찾기보다는 구성하는 가장 기본 단위에서부터 검증을 통해 품질이 높은 코드를 만들어 낼 수 있다는 점에서 유닛 테스트는 중요하다. 이상적으로, 각 테스트 케이스는 서로 분리되어야 한다. 유닛 테스트는 개발자 뿐만 아니라 보다 더 심도있는 테스트를 위해 테스터에 의해 수행되기도 한다.

2.2 Advantages of Unit Test

2.2.1 문제점 발견

유닛 테스트의 목적은 프로그램의 각 부분을 고립 시켜서 각각의 부분이 정확하게 동작하는지 확인하는 것이다. 즉, 프로그램을 작은 단위로 쪼개서 각 단위가 정확하게 동작하는지 검사하고 이를 통해 문제 발생 시 정확하게 어느 부분이 잘못되었는지를 재빨리 확인할 수 있게 해준다. 따라서 프로그램의 안정성이 높아진다. 유닛 테스트는 일견 개발 시간을 증가 시키는 것처럼 보이지만 개발 기간 중 대부분을 차지하는 디버깅 시간을 단축시킴으로써 여유로운 프로그래밍을 가능케 한다.

2.2.2 용이한 변경

프로그래머는 언제라도 유닛 테스트를 믿고 리팩토링(코드의 동작이나 의도는 유지하면서 코드의 구조, 재사용성, 가독성 등을 개선해 code smell을 제거하고 전체 디자인을 개선하는 방법)을 할 수 있다. 리팩토링 후에도 해당 모듈이 의도대로 작동하고 있음을 유닛 테스트를 통해서 확신할 수 있다. 이를 회귀 테스트라 한다. 어떻게 코드를 고치더라도 문제점을 금방 파악할 수 있고 수정된 코드가 정확하게 동작하는지 쉽게 알 수 있게 되므로 프로그래머들은 더욱 더 의욕적으로 코드를 변경할 수 있게 된다. 좋은 유닛 테스트 디자인은 그 유닛이 사용되는 모든 경로를 커버할

수 있는 테스트 케이스를 만들어 준다. 지속적인 유닛 테스트 환경을 구축하면 어떠한 변화가 있더라도 코드와 그 실행이 의도대로 인지를 확인하고 검증 할 수 있게 된다. 확립된 개발 방법과 유닛 테스트의 범위에 따라서 프로그램의 정확성이 좌우된다.

2.2.3 간단한 통합

유닛 테스트는 유닛 자체의 불확실성을 제거해주므로 상향식 테스트 방식에서 유용하다. 먼저 프로그램의 각 부분을 검증하고 그 부분들은 합쳐서 다시 검증하는 통합 테스트에서 더욱 더 빛을 발한다.

2.2.4 언어 지원

D와 루비, 파이썬은 유닛 테스트를 기본으로 지원한다. 유닛 테스트 프로세스를 단순화 시켜주는 유닛 테스트 프레임워크는 여러 언어로 개발되고 있다. 구글에서도 재능 기부 활동의 하나로 구글 테스트라는 C++를 위한 유닛 테스트 라이브러리를 제공하고 있다.

2.3 How to Unit Test

2.3.1 Black box testing

블랙박스 검사는 소프트웨어 검사 방법 중 하나로 어떤 소프트웨어를 내부 구조나 작동 원리를 모르는 상태에서 소프트웨어의 동작을 검사하는 방법을 이르는 말이다. 주로 올바른 입력과 올바르지 않은 입력을 일일이 다 동원하여 올바른 출력을 판별하는 방식으로 검사가 이루어지기 때문에 검사의 진행에 있어 대상이 되는 소프트웨어의 코드나 내부 구조 및 개발 노하우에 대한 정보는 기본적으로 필요로 하지 않는다. 필요한 것은 특징, 요구 사항, 검사를 위해 공개된 설계도 등 대외적으로 공개된 사항들이며 '이 소프트웨어는 무슨 역할을 수행해야 되는가'와 같이 대상이 되는 소프트웨어의 특징이나 요구 사항 등에 초점을 맞춰 검사가 이루어진다. 검사 자체는 기능에 관한 것일 수도 있고 기능 외의 것에 관한 것일 수도 있다.

2.3.2 White box testing

화이트박스 검사는 소프트웨어 내부 소스 코드를 테스트하는 기법이다. 소프트웨어를 테스트하는 방법은 크게 블랙박스 검사(Black-Box Test) 기법과 화이트박스 검사(White-Box Test) 기법이 있다. 블랙박스 검사 기법은 소프트웨어의 내부를 보지 않고, 입력과 출력 값을 확인하여, 기능의 유효성을 판단하는 테스트 기법이며, 화이트박스 검사 기법은 소프트웨어 내부 소스코드를 확인하는 기법이다. 화이트박스 테스트를 하는 이유는 내부 소스코드의 동작을 개발자가 추적 할 수 있기 때문에, 동작의 유효성뿐만 아니라 실행 되는 과정을 살펴봄으로써, 코드가 어떤 경로로 실행되며, 불필요한 코드 혹은 테스트 되지 못한 부분을 살펴볼 수 있다.

화이트박스 테스트를 하는 부분은 대개 코드의 실행 경로를 확인해야 하기때문에 시중에 나와 있는 커버리지 분석도구를 많이 활용한다. 화이트박스 검사 기법은 블랙박스 검사 기법에 비해 많은 시간과 분석을 필요로 하지만 오류가 발생 되는 결함의 위치 등을 파악하는데 매우 유용하게 사용 할 수 있다.

3 Unit Test in C

3.1 Frameworks for C Unit Test

3.1.1 CMOCK

CMock는 C헤더로부터 자동으로 Mock와 Stub를 생성한다. 테스트 시 중요하지 않은 함수 호출은 예외로 처리할 수 있으며 호출되기를 기대하는 특정 함수와 호출 시 기대되는 인자값을 기술할 때 CMock를 사용할 수 있다.

3.1.2 BMOCK

BMock는 Mock Objects를 지원하는 C++라이브러리이며 C함수와 멤버함수를 위한 Mocks를 만들 수 있다 BMock는 Boost테스트 라이브러리와 함께 사용되며 Boost라이브러리들을 필요로 한다.

3.1.3 CuTest

Cutest는 C언어를 위한 단위테스트 라이브러리다. 이건 XP(Extreme Programming), 초기개발에 사용될 수 있다. 결함이 적고 .c , .h 파일 하나로 구성되어있다.

3.1.4 Cfix

Cfix는 C/C++ xUnit과 같은 테스트 케이스들을 작성하고 실행하는 프레임워크이다. JUnit , NUnit 처럼 개발과 매우 비슷한 프레임워크를 제공하기 위한 도구이다

3.1.5 Check

C언어를 위한 유닛테스트 라이브러리이고 유닛테스트 정의를 위해 간단한 인터페이스를 제공한다. 테스트는 별도의 주소공간에서 실행되며 세그먼테이션 오류, 시그널 캐치와 같은 코드에러를 확인할 수 있다.

3.1.6 API Sanity AutoTest

API Sanity AutoTest는 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 테스트를 빌드 및 실행할 수 있으

며, craches, aborts, 모든 종류의 emitted signal, non-zero 프로그램 return 코드, 프로그램 행깅(hanging) 감지가 가능하다.

3.1.7 CUnit

CUnit은 C에서 단위테스트를 작성, 관리, 실행을 위한 경량화된 시스템이며 C언어를 사용하는 프로그래머에게 사용자 인터페이스의 다양한 기본적인 테스트 기능들을 제공한다.

3.1.8 AceUnit

편리한 C코드 단위 테스트 프레임워크다. AceUnit은 임베디드 소프트웨어 개발, 워크스테이션 및 서버 같은 곳에 사용될 수 있으며 모든 운영체제에서 실행할 수 있다

4 Framework for Coffee Machine

4.1 CUnit

4.1.1 설치 방법

(1) CUnit 소스 : 아래 경로에서 CUnit 소스 코드를 다운로드 받을 수 있다.
http://sourceforge.net/projects/cunit/

(2) 압축 해제
tar -nxvf CUnit-2.1-2-src.tar.bz2

(3). configure 파일 존재 여부 확인
cd CUnit-2.1-2
ls -l configure

>> 다음 에러 발생시 configure 파일이 미존재. (4)를 진행한다. <<
ls: configure에 접근할 수 없습니다: 그런 파일이나 디렉터리가 없습니다

>> 다음과 같이 출력되면 configure 파일이 존재. (5)를 진행한다. <<
-rwxrwxr-x 1 core core 429635 11월 27 00:03 configure

(4). configure 파일이 없는 경우. automake 로 configure 파일을 생성
cd CUnit-2.1-2

aclocal
autoconf
autoreconf --install ((8) 에러 발생시 사용)
automake ((7) 에러 발생 시 automake --add-missing를 사용)
chmod u+x configure (실행 권한이 없는 경우 사용한다)

(5). 빌드 및 설치
cd CUnit-2.1-2
./configure
make
sudo make install

(6). 만약 (4) 진행 중 다음과 같은 에러가 발생한 경우. libtool를 설치 한다.
에러> possibly undefined macro: AC_PROG_LIBTOOL
If this token and others are legitimate, please use m4_pattern_allow.
See the Autoconf documentation.

해결> 아래 명령 실행 후 (4) 다시 수행
sudo apt-get install libtool

(7). 만약 (4) 진행 중 다음과 같은 에러가 발생한 경우.
에러> required file `./install-sh' not found
`automake --add-missing' can install `./install-sh`

해결> (4) 에서 "automake" 대신 "automake --add-missing"를 사용한다.
automake --add-missing

(8). 만약, automake 시 `config.h.in' not found 에러가 발생하는 경우.
에러> required file `config.h.in' not found

해결> automake 전에 autoreconf를 사용한다.
autoreconf --install
automake ((7) 에러 발생 시 automake --add-missing를 사용)

4.1.2 Test code 작성법

Cunit 사용방법

1) 테스트를 위한 함수를 작성.
2) 테스트 레지스트리를 초기화
3) 테스트 레지스트리에 스위트를 추가
4) 스위트에 테스트 추가
5) 적절한 인터페이스 예를 사용하여 실행 테스트
   - Automated : xml 파일 출력
   - Basic : 표준 출력
   - Console : 콘솔입력으로 실행 선택/표준출력
   - Curses : 그래픽 인터페이스(UNIX 계열)
6) 테스트 레지스터 정리

```
#include "CUnit/CUnit.h"

#include "CUnit/Console.h"

#include "CUnit/Basic.h"

#include "테스트할대상.h"


/**

 * 테스트 코드 작성


 */

void test01(void) {

   // 예상값과 결과값이 같다고 예상함

   CU_ASSERT_EQUAL("테스트할 함수(파라미터)", "예상값");

}

void init(void) {

   // 테스트전 처리작업

}
```

```
void end(void) {
// 테스트후 처리작업
}
int main() {
 CU_pSuite suite = NULL;
 // 2)테스트 레지스트리를 초기화
 CU_initialize_registry();

 // 3)테스트 레지스트리에 스위트를 추가
 suite = CU_add_suite("스위트 타이틀", "테스트 시작전 세팅 작업함수", "테스트후 처리할 함수");
 // 예) 테스트 전후로 처리할 일이 없으면 NULL을 넣음.
 suite = CU_add_suite("테스트 스위트1",  init,  end);
 // 4)스위트에 테스트 추가
 // CU_add_test(suite, "타이틀", 테스트함수);
 // 예)
 CU_add_test(suite, "test01", test01);
 // 5) 적절한 인터페이스 예를 사용하여 실행 테스트
 CU_console_run_tests(); // 콘솔
 // 6) 테스트 레지스터 정리
 CU_cleanup_registry();
 return 0;
        }
```