



>> Special Report 05

단위 테스트와 코드 커버리지를 위한 진화

비주얼 스튜디오 2012 테스트 익스플로러 활용

예전에는 상상도 못했던 일들이 소프트웨어를 통해 제어되고 있다. 소프트웨어가 화성에 로봇을 보내기도 하고, 자동차를 운전하기도 하고, 사람을 수술하기도 하고, 내 은행 잔고를 관리하기도 한다. 소프트웨어가 점점 더 많은 일을 할수록 개발자의 어깨는 무거워진다. 사소한 버그로 인해 개발 당시에는 상상도 못했던 일들이 벌어질 수도 있기 때문이다. 비주얼 스튜디오 2012에 포함된 테스트 익스플로러 기능을 사용해서 어떻게 버그 없는 소프트웨어를 만들 수 있는지 살펴보도록 하자.

필자가 지난 5년간 게임 보안 제품을 개발하면서 보낸 시간은 하루도 빠지지 않고 전쟁터 같은 일상이었다. 항상 새로 추가해야 할 기능들이 있었고 그런 일정을 비웃기라도 하듯 해킹툴은 하루도 거르지 않고 새롭게 나타났다. 이런 연유로 빌드를 밥 먹듯이 하는 생활이 이어졌다.

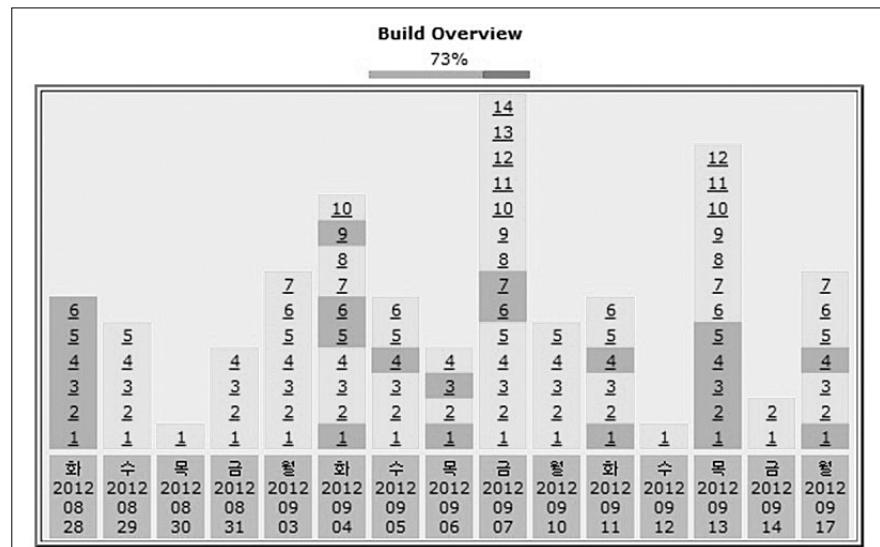
빌드 서버 로그 기록을 살펴보니 릴리즈 빌드 서버에서 지난 1년 동안 엔진 빌드를 214번, 차단 코드 빌드를 400번, 패턴을 포함한 전체 릴리즈 빌드를 무려 1,078회나 진행했다. 릴리즈 빌드는 거의 매일 3~4회씩은 있었다는 이야기고, 차단 코드를 변경하는 일도 거의 매

일 한 번 이상은 있었다는 의미다. <그림 1>에 나와 있는 최근 빌드 내역을 살펴보면 정말 매일같이 빌드가 있다는 말이 거짓말이 아님을 알 수 있다.

이렇게 매일 빌드하고 릴리즈를 하다 보니 해킹툴 만큼 버그와도 전쟁을 치려야 하는 처지가 돼 버렸고, 가장 큰 고민 중에 하나가 ‘어떻게 하면 버그 없는 소프트웨어를 개발할 수 있을까?’라는 다소 원초적인 문제였다. 다행스럽게도 앞서서 이런 문제를 고민했던 수많은 책들이 있었고, 먼저 고민한 선배들은 다양한 실천적인 해법을 제시해 줬다. 그런 다양한 조언과 실천적인 해법을 여기다 조금 옮겨보면 다음과 같다.



신영진 pop@jiniya.net, <http://www.jiniya.net> | 웨비아닷컴을 창업해 XIGNCODE3라는 게임 보안 솔루션을 개발하고 있다. 시스템 프로그래밍에 관심이 많고 다수의 PC 보안 프로그램 개발에 참여했다. 월간 마이크로소프트 웨어의 오랜 필자이면서 Microsoft MVP, 데브피아 Visual C++ 섹션 시삽으로 활동하고 있다. Steve Barakatt과 Grey's Anatomy의 광팬이며, 한때는 WoW에 미쳤었다. 한마디로 말하면 괴짜다.





- 똑똑한 프로그래머를 뽑아서 애초에 버그가 없는 코드를 만들도록 해야 한다.
- 모니터 1대를 두고 2명의 프로그래머가 같이 개발하면 버그가 혼저하게 낮은 코드를 생산할 수 있다.
- 전문 프로그래머 1명에게 적어도 1명 이상의 테스트 프로그래머를 배정해서 전문 프로그래머가 작성한 코드를 집중적으로 테스트하도록 만들어야 한다.
- 코드 리뷰를 철저하게 수행해서 리뷰를 통과하지 않은 코드는 애초에 커밋조차 할 수 없도록 만들어야 한다.
- 코드를 작성하기 전에 먼저 테스트 프로그램부터 만들어야 한다.
- 컴파일러 경고 레벨을 올리고 정적 분석기를 확실하게 활용해서 그 어떤 경고도 용납하지 않아야 한다.
- 코드 커버리지를 철저하게 측정해서 릴리즈 코드는 적어도 일정 수준 이상의 커버리지를 확보하도록 매 순간 확인해야 한다.
- 버그가 없도록 만드는 출렁한 컨벤션을 만들고 모두가 그 방식을 따르도록 강요한다.
- 전문 테스터를 고용해서 매 순간 테스터들이 무작위적으로 제품을 테스트할 수 있도록 한다.
- 구조화된 체크 리스트를 만들고 제품 릴리즈 시에는 해당 리스트를 모두 만족시키는지 빼먹지 않고 테스트하도록 한다.
- 같은 제품을 개발하는 프로그램 팀 회식을 매주 한 번씩 한다.
- 약수 물을 떠놓고 산신령께 빈다. 우리 제품에 버그가 없기를.

버그 없는 개발을 위한 이런 수많은 방법들 중에서도 단위 테스트, 코드 커버리지, 정적 분석기와 같은 것들은 투입 대비 효과가 매우 높은 편에 속하는 방법들이다. 왜냐하면 이것들을 수행하는 데에는 추가적인 리소스가 많이 필요하지 않기 때문이다. 프로그래머가 조금만 신경 쓴다면 이런 방법을 이용해서 손쉽게 다수의 버그를 줄일 수 있다.

이런 방법의 중요성을 충분히 깨닫고도 그것들을 충분히 활용하는 것에는 늘 주저했다. 왜일까? 우리가 사용하는 도구가 단위 테스트나 코드 커버리지 측정을 편리하게 실행할 수 있도록 만들어주지 않았기 때문이다. 단위 테스트를 한 번 수행하기 위해서는 별도로 복잡한 과정을 거쳐야 하고, 코드 커버리지나 프로파일링 데이터라도 측정하려면 무슨 화성에 로봇을 보내는 것과 같은 복잡한 과정을 거쳐야 했다. 효과는 있지만 충분히 쉽지 않았던 것이다. 하지만 이번에 출시된 비주얼 스튜디오 2012는 다르다. 이 모든 것들을 새롭게 추가된 Test Explorer를 통해 손쉽게 수행할 수 있도록 만들었다. 클릭 한 번으로 단위 테스트를 수행할 수 있고, 클릭 한 번으로 코드 커버리지를 실시간으로 확인할 수 있도록 만들었다. 물론 결과 데이터도 번잡하지 않다. 우리가

확인하고 싶은 것만 빠르게 확인할 수 있도록 만들어 준다. ‘진화’라는 단어가 전혀 어색하지 않을 만큼 엄청난 이 기능을 사용해서 어떻게 버그 없는 소프트웨어 개발을 할 수 있는지 살펴보도록 하자.

단위 테스트

최근에 필자는 리포팅 클래스에 캐시를 하나 더 추가할 일이 있었다. 이 과정에서 조건문을 잘못 추가하는 사소한 실수 때문에 치명적인 버그를 만들어낸 경험이 있다. 이 클래스를 가지고 단위 테스트와 코드 커버리지를 측정하는 방법을 알아보도록 하자.

내가 만들어야 했던 클래스 명세를 해보면 다음과 같다. 나는 이 클래스를 <리스트 1>과 <리스트 2>에 나온 것과 같이 만들었다.

- Notifier 클래스는 전달된 메시지를 화면에 출력하는 기능을 한다.
- 메시지는 악성(CODE_BLACK), 의심(CODE_SUSPICIOUS), 기타(CODE_EXTRA)로 구분된다.
- Notifier 클래스는 각각의 메시지를 필터링해서 이미 출력한 메시지는 두 번 출력하지 않도록 한다.
- 필터링은 악성 메시지와 나머지 메시지를 구분해서 한다. 즉 이미 의심으로 추가된 메시지가 기타나 의심으로 보고된다면 출력하지 않지만 악성으로 보고되면 출력한다는 의미다.

<리스트 1> notifier.hpp 소스 코드

```
#ifndef NOTIFIER_HPP
#define NOTIFIER_HPP

#include <set>
#include <string>

class Notifier
{
public:
    enum NotifyCode
    {
        CODE_BLACK = 13
        , CODE_SUSPICIOUS
        , CODE_EXTRA
    };

    typedef std::set<std::wstring> StrSet;
    typedef std::set<std::wstring>::iterator StrSIt;

    StrSet b_cache_;
    StrSet s_cache_;
}
```

```

    BOOL Notify(NotifyCode code, LPCWSTR msg);
    BOOL IsCacheItem(StrSet &s, LPCWSTR msg);
};

#endif

```

〈리스트 2〉 notifier.cpp 소스 코드

```

#include "stdafx.h"
#include <Windows.h>
#include "notifier.hpp"

BOOL
Notifier::IsCacheItem(StrSet &s, LPCWSTR msg)
{
    StrSet it = s.find(msg);
    if(it == s.end())
    {
        s.insert(msg);
        return FALSE;
    }

    return TRUE;
}

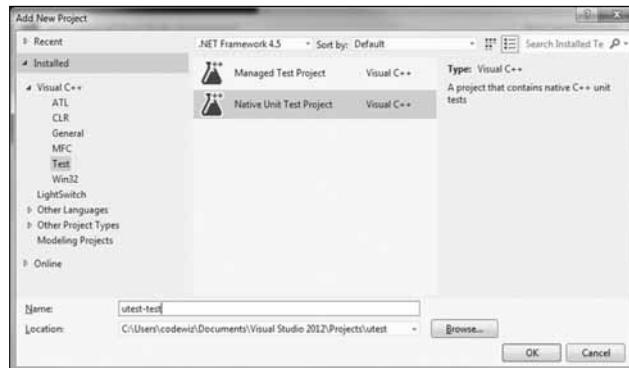
BOOL
Notifier::Notify(NotifyCode code, LPCWSTR msg)
{
    if(code == CODE_BLACK && IsCacheItem(b_cache_, msg))
    {
        return FALSE;
    }
    else if(IsCacheItem(s_cache_, msg))
    {
        return FALSE;
    }

    printf("%d %ws\n", code, msg);
    return TRUE;
}

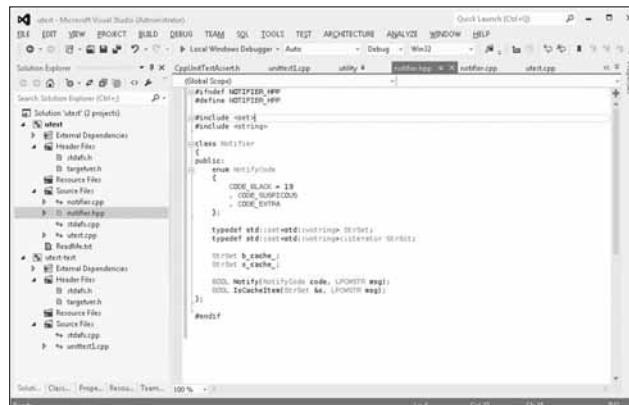
```

그림 이제 프로젝트 구성을 해보자. 비주얼 스튜디오 2012에서 utest라는 콘솔 프로젝트를 솔루션 구조로 생성한다. 프로젝트를 생성했으면 이 솔루션에 utest-test라는 네이티브 단위 테스트 프로젝트를 추가하자. 이 프로젝트 타입은 〈그림 2〉에 나타난 것과 같이 Test 탭에 포함돼 있다. 프로젝트 생성이 끝나면 utest 프로젝트에 notifier.hpp와 notifier.cpp를 추가한다. 각각의 소스 코드는 〈리스트 2〉와 〈리스트 4〉에 나와 있는 코드대로 입력하도록 하자. 끝으로 notifier.cpp의 설정 항목에 들어가서 미리 컴파일된 헤더를 사용하지 않도록 만들어준다. 여기까지 모든 과정을 정상적으로 수행했다면 〈그림 3〉에 나타난 것과 같은 솔루션 구조를 가지게 되고 탐색기에서 해당 솔루션 폴더를 찾아

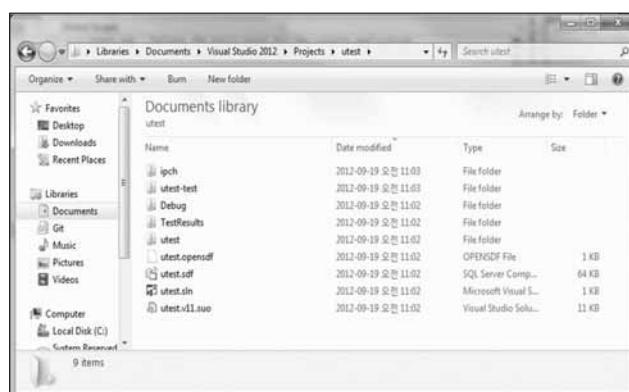
가보면 〈그림 4〉에 나타난 것과 같이 디렉터리 구조가 생성된 것을 볼 수 있다.



〈그림 2〉 네이티브 단위 테스트 프로젝트 선택 화면



〈그림 3〉 utest와 utest-test 프로젝트가 추가된 화면



〈그림 4〉 utest 솔루션 디렉터리 구조

프로젝트 구성이 끝났으면 utest-test 단위 테스트 프로젝트에서 utest 프로젝트를 참조할 수 있도록 디렉터리 설정을 변경해야 한다. 다음 순서에 따라 설정을 변경하도록 하자.

- 1 utest-test 프로젝트 설정 화면에 들어간 다음 C++ 탭의 General 페이지에 있는 Additional Include Directories 항목



에 “..\utest”를 추가한다. 이 설정은 단위 테스트 프로젝트에서 utest 프로젝트에 포함된 헤더 파일을 바로 추가할 수 있도록 만들어 주는 기능을 한다.

2 Link 탭의 General 페이지에 Additional Library Directories 항목에 “..\utest\\$(Configuration)”을 추가한다. utest에서 빌드한 오브젝트를 참조할 수 있도록 해준다. 이 설정은 단위 테스트 프로젝트에서 utest 오브젝트 파일을 바로 포함할 수 있도록 만드는 기능을 한다.

3 끝으로 Link 탭의 Input 페이지의 Additional Dependency 부분에 notify.obj 항목을 추가한다. 이 설정은 단위 테스트 프로젝트에서 notifier.cpp의 컴파일된 코드를 사용할 수 있도록 만드는 기능을 한다.

여기까지 마쳤으면 utest-test 단위 테스트 프로젝트에 테스트를 추가할 모든 준비가 끝났다. <리스트 3>에 나와 있는 것과 같이 테스트 케이스를 추가해 보도록 하자. Assert::ArrEqual 함수는 n.Notify(Notifier::CODE_BLACK, L"BITEM")의 실행 결과가 TRUE와 같은지 체크하는 기능을 한다. TRUE와 같으면 테스트가 성공하고, TRUE가 아닌 다른 값이 반환되면 테스트가 실패했음을 알려준다.

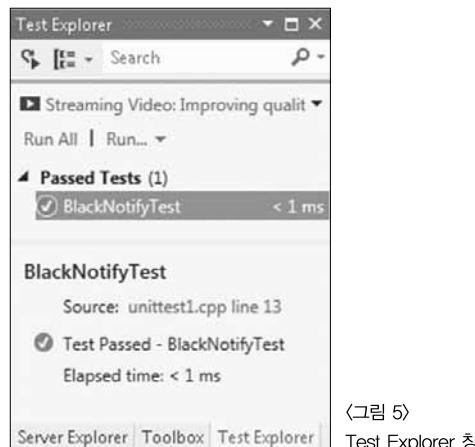
<리스트 3> unittest1.cpp 소스 코드

```
#include "stdafx.h"
#include "CppUnitTest.h"
#include <Windows.h>
#include "notifier.hpp"
using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace utesttest
{
    TEST_CLASS(NotifyTester)
    {
        public:

            TEST_METHOD(BlackNotifyTest)
            {
                Notifier n;
                Assert::AreEqual(TRUE,
n.Notify(Notifier::CODE_BLACK, L"BITEM"));
            }
    };
}
```

코드 추가가 끝났으면 각 프로젝트를 빌드하고 비주얼 스튜디오 2012 메뉴에서 Test, Window, Test Explorer 항목을 선택한다. 그러면 <그림 5>에 나타난 것과 같이 테스트 익스플로러가 화면에 표시된다. 우리가 앞으로 진행할 단위 테스트와 코드 커버리지는 모두 이 창을 통해서 손쉽게 제어할 수 있다. Black NotifyTest를 수행하기 위해서는 해당 테스트 이름에서 오른쪽을 클릭한 후 메뉴에서 “Run Selected Tests” 항목을 선택하면 된다. 그러면 비주얼 스튜디오가 알아서 테스트를 진행하고 결과를 우리에게 알려준다.



<그림 5>
Test Explorer 창

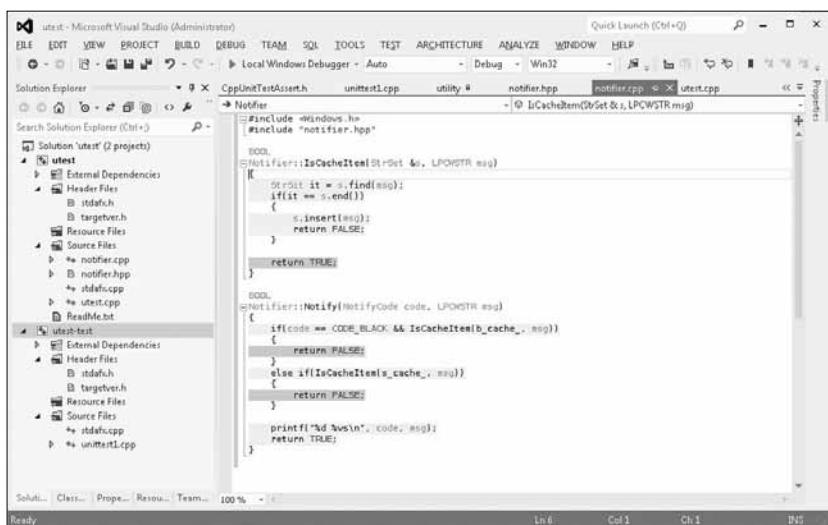
코드 커버리지

코드 커버리지는 테스트를 통해 코드의 얼마나 많은 부분이 실행됐는지를 나타내는 값이다. 코드 커버리지가 100%라는 의미는 코드의 모든 부분이 테스트를 통해서 실행됐다는 이야기를 나타낸다. 앞서 우리가 만든 BlackNotifyTest 코드가 Notifier 클래스를 얼마만큼 테스트하는지 측정해 보도록 하자. Test Explorer에서 BlackNotifyTest에 오른쪽 버튼을 누른 다음 표시되는 메뉴에서 “Analyze Code Coverage for Selected Tests” 항목을 선택한다. 그러면 단위 테스트를 수행하는 것과 똑같이 테스트가 진행되고 <그림 6>에 나타난 것처럼 코드 커버리지 측정 결과가 나타난다.

<그림 6>의 측정 결과를 살펴보면 우리가 작성한 테스트는 Notifier 클래스의 87.5%를 실행하고 있음을 알 수 있다. 세부적으로는 IsCacheItem 함수를 90%, Notify 함수를 83.33% 커버 한다. 여기서 중요한 것은 어떤 부분이 테스트되지 않았는지 살펴보는 것이다. IsCacheItem 함수 이름이 있는 줄을 더블 클릭하면 <그림 7>에 나타난 것과 같이 실제 코드에서 테스트가 진행된 부분과 진행되지 않은 부분을 명확하게 나눠서 표시해 준다. 붉은색으로 표시된 부분이 BlackNotifyTest를 통해서 테스트되지 않은 부분이다.

| Code Coverage Results | | | | |
|--|----------------------|------------------------|------------------|--------------------|
| Hierarchy | Not Covered (Blocks) | Not Covered (% Blocks) | Covered (Blocks) | Covered (% Blocks) |
| codewiz_CODEWIZ-PC 2012-09-20 11_51_09 | 4 | 10.26 % | 35 | 89.74 % |
| utest-test.dll | 4 | 10.26 % | 35 | 89.74 % |
| Global Classes | 4 | 12.50 % | 28 | 87.50 % |
| Notifier | 4 | 12.50 % | 28 | 87.50 % |
| IsCacheItem | 2 | 10.00 % | 18 | 90.00 % |
| Notify | 2 | 16.67 % | 10 | 83.33 % |
| utesttest | 0 | 0.00 % | 7 | 100.00 % |

(그림 6) 코드 커버리지 결과 화면



(그림 7) 코드 커버리지 테스트 결과 화면

〈리스트 4〉 BlackNotifyTest, SuspNotifyTest 테스트 케이스 소스 코드

```
TEST_METHOD(BlackNotifyTest)
{
    Notifier n;

    Assert::AreEqual(TRUE, n.Notify(Notifier::CODE_BLACK,
L"BITEM"));

    Assert::AreEqual(FALSE, n.Notify(Notifier::CODE_BLACK,
L"BITEM"));
}

TEST_METHOD(SuspNotifyTest)
{
    Notifier n;

    Assert::AreEqual(TRUE,
n.Notify(Notifier::CODE_SUSPICIOUS, L"SITEM"));

    Assert::AreEqual(FALSE,
n.Notify(Notifier::CODE_SUSPICIOUS, L"SITEM"));
}
```

이제 코드의 어떤 부분들이 테스트를 통해 실행되지 않았는지 알 수 있으니 그 부분들을 실행할 수 있는 테스트 케이스를 추가

해 보자. 〈리스트 4〉에는 테스트되지 않은 부분을 모두 실행할 수 있도록 만들어진 테스트 케이스가 나와 있다. 이 테스트 케이스를 추가한 후에 빌드하고 다시 코드 커버리지를 측정해 보면 〈그림 8〉에 나타난 것과 같이 코드 커버리지가 100%로 출력되는 것을 확인할 수 있다.

코드 커버리지 100%를 충족시켰으니 과연 완벽한 테스트 케이스이고 우리가 만든 Notifier 클래스가 우리의 의도를 100% 만족한다고 자신할 수 있을까? 안타깝게도 아니다. 이 수치는 우리가 만든 테스트가 실제 코드의 어떤 부분을 실행했는지 나타내는 기계적인 값이다. 코드 커버리지의 함정은 여기에

있다. 이것은 신택스만(syntax) 확인할 수 있을 뿐이지 시맨틱(semantic)은 검증할 수 없다는 것을 의미한다.

앞서 Notifier 클래스 코드 명세에서 이 클래스가 가지고 있는 2개의 캐시는 독립적으로 동작해야 한다고 했다. 그렇다면 명세대로 과연 각각의 캐시가 독립적으로 동작하는지 살펴보도록 하자. 〈리스트 5〉에는 캐시의 독립성을 검증할 수 있는 테스트 케이스가 나와 있다. 이 테스트를 추가한 다음 Test Explorer를 통해서 테스트를 진행해 보면 〈그림 9〉에 나타난 것과 같이 테스트가 실패하는 것을 볼 수 있다.

〈리스트 5〉 SBTest 코드

```
TEST_METHOD(SBTest)
{
    Notifier n;

    Assert::AreEqual(TRUE,
n.Notify(Notifier::CODE_SUSPICIOUS, L"ITEM"));

    Assert::AreEqual(FALSE,
n.Notify(Notifier::CODE_BLACK, L"ITEM"));
}
```



Code Coverage Results

codewiz_CODEWIZ-PC 2012-09-20 12_37_36

| Hierarchy | Not Covered (Blocks) | Not Covered (% Bloc...) | Covered (Blocks) | Covered (% Blocks) |
|--|----------------------|-------------------------|------------------|--------------------|
| codewiz_CODEWIZ-PC 2012-09-20 12_37_36 | 0 | 0.00 % | 50 | 100.00 % |
| + utes-test.dll | 0 | 0.00 % | 50 | 100.00 % |
| + {} Global Classes | 0 | 0.00 % | 32 | 100.00 % |
| + Notifier | 0 | 0.00 % | 32 | 100.00 % |
| + IsCacheItem | 0 | 0.00 % | 20 | 100.00 % |
| + Notify | 0 | 0.00 % | 12 | 100.00 % |
| + {} utes-test | 0 | 0.00 % | 18 | 100.00 % |

〈그림 8〉 수정된 테스트 케이스 커버리지 측정 결과



```

        break;
    }

    printf("%d %ws\n", code, msg);
    return TRUE;
}

```

앞서 살펴보았던 것과 같이 코드 커버리지는 보조적인 지표 역할을 해줄 뿐 커버리지가 높다고 테스트 품질이 높다는 것을 의미하지는 않는다. 따라서 테스트 케이스는 명세를 바탕으로 꼼꼼하게 먼저 작성하는 습관을 들이는 것이 중요하다. 코드 커버리지는 작성한 테스트를 통해서 어떤 부분이 실행되지 않는지를 확인하고 추가적으로 어떤 테스트 케이스를 만들어야 하는지를 참고하는 지표 정도로 활용하는 것이 좋다. +

문제는 if 문장에 숨어 있다. CODE_BLACK 아이템에 대해서는 else if 줄이 검사되지 않아야 한다. 하지만 〈리스트 2〉에 나타난 코드의 else if 구문에는 'code != CODE_BLACK' 과 같은 조건이 없었기 때문에 해당 else if 문이 검사되면서 이런 결과가 나타나게 된 것이다. 이와 같은 문제를 미연에 방지하기 위해서는 〈리스트 6〉과 같이 코드를 명확하게 분리해서 작성하면 된다.

〈리스트 6〉 수정된 Notify 함수

```

BOOL
Notifier::Notify(NotifyCode code, LPCWSTR msg)
{
    switch(code)
    {
    case CODE_BLACK:
        if (IsCacheItem(b_cache_, msg))
            return FALSE;

        break;

    default:
        if (IsCacheItem(s_cache_, msg))
            return FALSE;
    }
}

```

〈월간〉마소는
늘 개발자의 곁에 서 있습니다

micro 1년 후에도 내용이 살아있는 집지
Software