단위테스트를 위한 레거시소프트웨어시스템의 재구성 기법

문 중 희[†] · 이 남 용[††]

요 약

레거시소프트웨어시스템을 유지 및 보수하는 작업은 소프트웨어 공학 분야에서 중요한 화두이다. 그리고 유지 및 보수 과정에 있어 회귀 테스트는 소프트웨어의 변경에 따른 기능적 동작이 올바른지 확인한다. 그러나 기존의 회귀 테스트는 대부분 시스템 레벨에서 접근이 되었으며 단위테스트 레벨에서는 준비된 테스트 케이스가 없어서 적용이 어려웠다. 본 논문에서는 단위테스트 케이스들을 구현하고 자산화하기 위해서 기존의 레거시소프트웨어시스템을 재구성하는 기법을 제안한다. 그리고 이를 실제 개발 과제의 특정 모듈에 적용하고 그 테스트 커버리지 결과를 분석하였다. 향후 본 논문에서 제시하는 방안을 기반으로 재구성 자동화 기법 및 테스트 케이스 자동화 생성에 대한 연구가 지속된다면 레거시소프트웨어시스템의 유지 및 보수에 큰 발전을 기대할 수 있을 것이다.

키워드 : 레거시 소프트웨어, 단위테스트, 소프트웨어 유지 및 보수, 회귀테스트, 재공학

A Restructuring Technique of Legacy Software Systems for Unit Testing

Joong Hee Moon[†] · Nam Yong Lee[††]

ABSTRACT

The maintenance of legacy software systems is very important in the field of a software engineering. In the maintenance, a regression test confirms the behavior preserving of the software which has been changed but most of regression tests are done in a system level and rarely done in a unit test level because there is no test case. This paper proposes how to modify legacy software systems and make unit test cases as an asset. It uses a technique with a specific module of a real software development project and analyzes test coverage results. After this, if a study about automatic restructuring techniques and a test case generation proceeds continuously, we can expect the big advance of legacy software systems maintenance.

Key Words : Legacy Software, Unit Test, Software Maintenance, Regression Test, Restructuring

1. 서 론

20세기 중반 이후부터 최근까지 많은 소프트웨어들이 개발되었다. 이와 함께 개발된 소프트웨어를 유지하고 관리하는 방법도 중요한 문제로 대두되고 있다. 소프트웨어 유지보수[1]란 소프트웨어 시스템이 완성된 이후에 행해지는 모든 활동을 가리키며 소스코드에 변화가 생길 때에는 수행해야 할 기능적인 동작들이 올바르게 작동하는지(semantic-preserving)[2] 항시 확인할 수 있어야 한다. 이에 대한 방안으로 회귀 테스트[9]를 수행할 수 있을 것이다. 회귀 테스트는 두 가지 방법으로 접근할 수 있다. 첫째는 소프트웨어 시스템 레벨에서의 테스트이다. 두 번째는 소프트웨어 단위 모듈 레벨에서의 테스트이다. 본 논문에서는 후자의 경우에 대해서 연구를 진행하였다. 단위테스트[3]란 개발단계 중에 개발자 개개인이 직접 만든 모듈의 기능적인 동작을 확인하기 위해서 개발자 스스로가 테스트 케이스를 작성하고 수행하는 활동을 가리킨다. 그리고 작성한 단위테스트 케이스들을 이후 회귀 테스트를 위해서 재사용할 수 있다. 그러나 기존 대부분의 소프트웨어 개발단계에서는 단위 테스팅이 효과적으로 적용되고 있지 않다. 이에 대해서는 단위테스트 케이스 작성 및 수행에 따른 비용 등 여러 가지 원인들을 찾을 수 있을 것이다. 하지만 소프트웨어 유지 및 보수의 비용이 점점 증가하고 있는 요즘의 상황에서는 단위테스트 케이스들을 재사용하는 것이 점점 중요하고 필요하다. 일반적인 개발단계에서 단위테스트 케이스를 작성할 때에는 테스트 대상 모듈을 호출하는 테스트 드라이버(Test Driver)[4]와 테스트 대상 모듈이 호출하는 테스트 스텁(Test Stub)[4]을 자유롭게 작성을 할 수가 있다. 하지만 레거시 코드(Legacy Code)에서는 모든 모듈이 이미 개발되어 있어 테스트 드라이버와 테스트 스텁을 적용하는 것이 어렵거나 불가능하다. 본 논문에서는 이미 완성되어 사용되고 있는 레거시소프트웨어시스템 (Legacy Software)[5]에 대해서 단위테스트 케이스들을 작성하고 관리하는 방안을 제시하고자 한다. 본 연구에

† 정 회 원 : 삼성전자 S/W연구소 선임연구원
†† 정 회 원 : 숭실대학교 컴퓨터학과 교수
논문접수 : 2007년 5월 27일, 심사완료 : 2007년 10월 13일

DOI: 10.3745/KIPSTD.2008.15-D.1.107

서는 스텁을 테스트를 위해서 기능적인 동작을 제어할 수 있는 Mock[7]이라는 용어로 대체하여 사용하고자 한다. 연구는 다음과 같은 순서로 진행이 되었다.

2장에서는 기존 방법의 문제점을 세부적으로 설명한다. 3장에서는 기존의 관련 연구를 소개하고 본 논문과의 차이점을 설명한다. 그리고 본 논문이 가지는 의의를 설명한다. 4장에서는 본 논문에서 제시하는 방안을 설명하고 5장에서는 실제 개발 코드를 대상으로 재구성 작업을 수행한다. 그리고 재구성에 따른 개발 코드의 동작이 그대로 보존될 수 있다는 것을 설명한다. 6장에서는 적용 이전의 것과 적용 이후의 것을 단위 테스트 수행에 따른 테스트 커버리지 측정값으로 비교한다. 7장에서는 본 논문의 내용을 요약하고 논문이 가지는 의의를 설명한다.

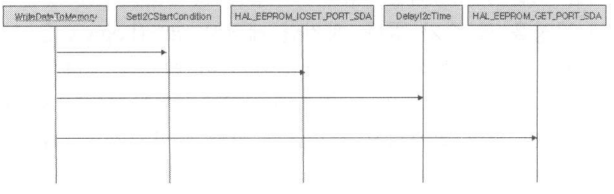
2. 기존 방법의 문제점

(그림 1)은 예제 API(Application Programming Interface) 함수를 나타낸다. 두 번째 인자가 가리키는 버퍼(buffer)의 내용을 첫 번째 인자가 가리키는 메모리(memory)의 주소로 복사하는 기능을 수행한다. 세 번째 인자는 복사할 버퍼의 크기를 가리킨다. API함수의 동작이 올바르게 수행되었을 경우에는 1이라는 반환 값을 나타내게 된다. 그렇지 않을 경우에는 0의 값을 반환한다. 이 함수에 대해서 테스트 케이스들을 작성할 경우 제어할 수 있는 요소들은 API함수의 세 가지 인자들 및 함수에서 사용되는 전역 변수들이다. 그러나 효과적인 단위테스트를 위해서는 다른 함수들간의 호출 관계 또한 고려해야 할 것이다. UML을 사용한 이 API 함수 내부의 흐름(Flow)를 살펴보면 (그림 2)과 같다. (그림 3)는 (그림 1)에 대한 Legacy 소스 코드의 일부 내용을 나타낸다. 테스트 대상 함수는 SetI2CStartCondition함수를 호출하여 EEPROM의 Start Mode를 보낸다. 또한 Hardware Abstraction Layer의 HAL_EEPROM_IOSET_PORT_SDA함수를 호출하여 하드웨어의 특정 포트 값을 설정한다. 그리고 DelayI2cTime 함수를 호출하여 일정 시간 동안 Time delay를 설정한다. 그리고 (그림 3)의 가장 우측에 위치한 HAL_EEPROM_GET_PORT_SDA함수를 호출하여 그 반환 값이 HI라는 매크로 상수와 같은지를 비교하여 그 결과에 따라 각기 다른 루틴을 수행하도록 한다. 이 경우 이미 테스트 대상 함수 혹은 모듈이 다른 모듈과 결합(Integration)된 상태에서는 (그림 2)에서 보여지는 함수들에 대한 제어가 불가능하다. 왜냐하면 이들 함수들을 제어하기 위해서는 이들 함수들과 동일한 프로토타입(Prototype)을 가지는 함수들을 스텁 혹은 Mock으로 만들어야 하는데 이 때 기존의 이미 구현된 레거시 함수 및 모듈들과 충돌하게 되기 때문이다. 때문에 테스터는 테스트 대상 함수의 입력 파라미터 및 일부 전역 변수들에 대한 제어만을 할 수 있다. 하지만 이러한 제약을 가지고 단위 테스팅을 하게 되면 내부 구조의 흐름에 대한 제어가 어렵거나 불가능하기 때문에 단위 테스팅에 따른 테스트 커버리지(Test Coverage)가 매우 낮아지게 되는 문제점이 따른다. 결국 올바른 단위 테스팅을 할 수 없게 되는 것이다. 그리고 이를 해결하기 위해서 만일 테스트 대상 개발 모듈만을 별도로 떼어내어 테스트를 수행하거나 혹은 테스트를 위해서 개발 코드를 다시 수정한다면 오히려 부가적인 비용에 큰 부담을 가지게 될 것이다.

```
/**
 * writeDataToMemory
 * Write to nonvolatile memory.
 * @param   U16 u16MemoryAddress, void* pSourceBuffer, U16 u16DataSizeToWrite
 * @return  BOOL
 * @see     N/A
 * @version
 * @author
 */
PUBLIC BOOL WriteDataToMemory(U16 u16MemoryAddress, void* pSoureBuffer, U16 u16DataSizeToWrite)
{
    U08 u08ErrorCount;
    U08 u08RetryCount;
    U08 u08DataBlock;
    u16 u16WriteByteCount;
    U16 u16AddressToWrite;

    if (u16DataSizeToWrite == 0) return OK;
}
```

(그림 1) 테스트 API 함수



(그림 2) UML Sequence Diagram

```
// EEPROM START MODE SEND
SETI2CStartCondition();
HAL_EEPROM_IOSET_PORT_SDA(PORT_IN);
DelayI2CTime(2);
if (HAL_EEPROM_GET_PORT_SDA() == HI)
{
    u08DataBlock = (U08)(((*((U08*)&u16AddressToWrite + 1)) & 0x07) << 1);
    if (WriteI2cByteData( 0xa0 | U08DataBlock) == OK)       // Control Byte
}
```

(그림 3) Legacy Source Code

3. 관련 연구 및 본 연구의 의의

3.1 Design for Testability [13]

논문에서 소프트웨어의 Testability는 소프트웨어의 동작 상태를 얼마나 용이하고 정확하게 관찰하여 이와 관련된 동작을 제어할 수 있는지에 기반한다고 설명한다. 그리고 이에 대한 세부적인 내용으로 verbose output, event logging, assertions, diagnostics, resource monitoring등 다양한 방법들을 디자인에 반영할 것을 제안한다. 이들 방법들을 적용하여 레거시 코드의 Testability를 향상시키고 단위, 통합, 시스템 테스트를 용이하게 할 수 있을 것이다. 본 논문에서는 단위 테스트를 위한 Testability향상 방안을 보다 심층적으로 연구하였고 그 결과를 소개한다.

3.2 Reengineering for Testability [14]

논문에서는 리엔지니어링을 하는 요인들 가운데에는 Testability를 높이고자 하는 목적이 있다고 설명한다. 그리고 이를 위해서 내부적인 경로를 단순화하고 개발 코드 구조의 복잡도를 낮추며 사용자 인터페이스 등을 단순화 하는 방안들을 제시한다. 이들 방법을 적용하여 레거시 코드의 Testability

는 크게 향상될 수 있을 것이다. 그러나 레거시 코드에 단위 테스트 케이스를 새로 작성하여 적용하기 위해서는 보다 세부적인 연구가 필요하다. 본 논문에서는 이에 대한 연구 결과를 설명한다.

3.3 A Survey of Software Refactoring [8]

논문에서는 소프트웨어의 리팩토링 혹은 재구성(Restructuring)의 목적을 소프트웨어 유지/보수를 용이하게 하기 위함이라고 설명한다. 그리고 유지/보수를 위해서는 확장성(Extensibility), 모듈화 정도(Modularity), 재사용성(Reusability), 복잡도(Complexity) 등의 개선이 필요하며 이를 위해서 수행해야 하는 활동과 사용하는 기법 및 관리해야하는 산출물 등을 설명한다. 그러나 ISO 9126[15]에서는 소프트웨어 유지/보수를 위해서 고려해야하는 주된 사항들 가운데 Testability를 부가적으로 제시한다. 그리고 본 논문에서는 Testability를 향상시키기 위한 소프트웨어 재구성의 단계별 수행 과정 및 기법을 설명하면서 심층적인 연구 내용을 소개한다.

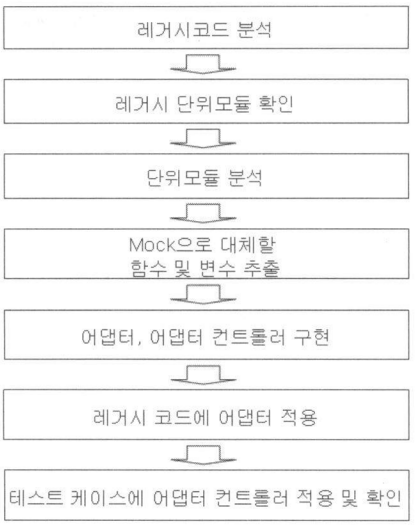
3.4 본 연구의 의의

소개한 논문의 내용을 기반으로 단위 테스트 케이스 수행에 따른 외부적인 결과만이 아니라 내부적인 상태 또한 보다 정확하고 쉽게 확인할 수 있을 것이다. 그리고 개발 코드의 구조를 단순화하고 입력 인터페이스를 단일화 하면서 단위 테스트 케이스의 수를 최소한으로 작성하여 적용할 수 있을 것이다. 그러나 그러함에도 2장에서 지적한 단위 테스트 케이스 작성에 따른 스텁과 Mock을 생성해야 하는 문제는 여전히 해결되지 않는다. 이는 아직 단위 테스트 케이스 적용을 위한 소프트웨어 재공학에 대해서는 많은 연구가 진행되지 않았기 때문으로 판단된다. 본 논문에서의 연구는 기존 연구된 내용들을 바탕으로 단위 테스트 케이스 작성을 위한 보다 심층적인 재공학 방안을 제시하고 실제 개발 과제에 이를 적용하였다는데 큰 의의가 있을 것이다.

4. 어댑터를 적용한 재구성 기법

(그림 4)은 소개될 4.1, 4.2 그리고 4.3의 단계에 대한 레거시 코드의 재구성 흐름도를 나타낸다.

먼저 대상 레거시 코드를 분석하고 단위 테스트를 적용할 모듈을 확인하고 선택한다. 그리고 선택된 단위 모듈의 내부를 분석하여 의존관계가 있는 모듈 및 변수를 확인한다. 이들 가운데 Mock으로 대체할 필요가 있는 함수 및 변수들을 추출한다. 그리고 선택적으로 Mock혹은 레거시 모듈을 적용할 수 있도록 테스트 어댑터를 구현하고 적용한다. 그리고 단위 테스트 케이스에서 이를 제어할 수 있도록 어댑터 컨트롤러를 구현한다 마지막으로 단위 테스트 케이스를 어댑터 컨트롤러를 사용하여 구현하고 수행하여 재구성이 올바르게 되었는지 결과를 확인할 수 있을 것이다. 테스트 어댑터와 테스트 컨트롤러는 4.2 그리고 4.3에서 자세하게 설명된다.



(그림 4) 레거시코드 재구성 흐름도

4.1 함수 및 변수 추출

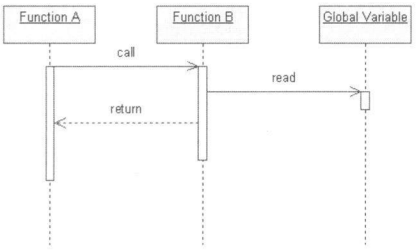
먼저 단위 모듈을 테스트 하기 위해서는 관계된 주변 모듈들을 제어할 수 있어야 한다. 즉, 이 모듈들을 Mock으로 대체하여 대상 단위 모듈이 이들을 실제 모듈인 것으로 간주할 수 있는 환경을 갖추어야 한다. 이를 위해서 제어할 함수 및 변수들을 추출하는 작업을 수행해야 한다. 직접적으로 혹은 우회적으로 제어할 수 있는 함수 혹은 변수는 선택의 순위가 낮다. 반면 단위 모듈을 테스트 하기 위해서 반드시 제어할 필요가 있으며 레거시 코드 기반으로 제어하기가 불가능하거나 어려울 경우에는 선택의 우선순위가 높아질 것이다.

(그림 5)에서 함수A를 테스트 하기 위해서는 함수B는 인위적으로 제어할 필요가 있을 것이다. 이 경우 함수B를 Mock으로 대체하여 함수A로의 반환 값을 제어할 수 있다. 그러나 다른 방법으로 함수B가 참조하는 전역변수를 제어하여 함수B가 반환하는 값을 제어할 수도 있을 것이다. 때문에 이 경우에는 순위를 낮추고 가능한 우회적인 방법으로 함수B를 제어하는 것이 효과적일 것이다.

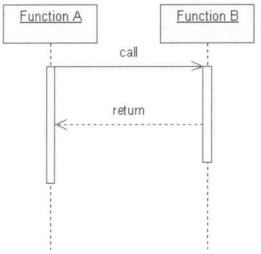
그러나 (그림 6)에서는 함수B를 제어할 방안을 찾기가 어렵다. 때문에 이 경우에는 어댑터 적용을 위한 추출 순위를 높이고 함수B를 Mock으로 대체할 필요가 있을 것이다.

4.2 어댑터 구현 및 적용

추출된 함수 혹은 전역 변수들을 Mock으로 대체하기 위한 어댑터를 구현해야 한다. 그리고 해당 함수 및 전역 변수와 테스트 대상 단위 모듈들을 어댑터를 적용하여 연결한다. 이 때 어댑터는 세 가지 기능을 지원할 수 있어야 한다. 첫째, 기존의 레거시 모듈을 가능한 적게 수정할 수 있게끔 구현되어야 한다. 그래서 재구성에 따른 비용을 최소화할 수 있어야 할 것이다. 본 논문에서는 5장에서 간단한 매크로 함수를 사용한 예를 보인다. 둘째, 개발코드와 구현한 어댑터를 함께 빌드하거나 혹은 순수 개발 코드만을 따로 빌드할 수 있어야 한다. 즉, 재구성은 단위 테스팅을 위해서만



(그림 5) 함수제어가 가능한 경우



(그림 6) 함수제어가 어려운 경우

개발 코드를 변경해야 할 것이고 순수 개발 코드만을 따로 빌드할 수 있는 환경을 제공해야 할 것이다. 그리고 셋째로는 어댑터를 적용하여 시스템의 기능적 동작에 변화가 있어서는 안 될 것이다.

4.3 어댑터 컨트롤러 설계 및 구현

단위 테스트 케이스를 구현하면서 Mock을 제어할 수 있는 어댑터 컨트롤러를 설계하고 구현해야 한다. 컨트롤러는 다음 두 가지 기능을 지원할 수 있어야 한다. 첫째, 컨트롤러를 생성하는 기능을 제공해야 한다. 이 기능은 해당 Mock을 활성화시켜서 사용할 수 있게끔 한다. 또한 반대로 Mock을 비활성화하는 기능이 필요하다. 둘째, Mock을 생성하여 적용할 수 있는 기능을 제공해야 한다. 즉, 테스트 케이스 개발자는 Mock을 구현하고 컨트롤러가 제공하는 인터페이스를 사용하여 해당 Mock을 적용할 수가 있어야 한다.

5. 레거시 코드에 대한 기법 적용 사례

본 논문에서는 세탁기에 탑재되는 실제 레거시 코드를 대상으로 기법을 적용하였다. 적용 범위는 세탁기에 탑재되는 EEPROM 디바이스 드라이버의 기존 작성된 단위 테스트 케이스로 선택하였다.

5.1 제어할 함수 및 변수 추출

실험에서는 HAL(Hardware Abstraction Layer)에서 제공하는 함수 및 변수와 조건 문에 영향을 주는 함수 및 변수를 우선적으로 선택하였다. HAL에서 제공하는 함수는 하드웨어의 동작에 대한 결과를 던져주는 역할을 가진다. 그러나 실제 하드웨어의 동작을 제어하는 것은 매우 어려운 일이기 때문에 이들 함수를 우선적으로 선택하였다. 또한 조건문에 영향을 주는 함수를 Mock으로 제어하여 테스트 대상 모듈의 테스트 커버리지를 높이려고 하였다. 이들 함수의 일부를 (그림 1)에서 확인할 수 있다.

```
int FLAG;
int (*mock)(void);

#ifndef _MOCK_ADAPTOR
    #define _(a)    a
#else
    #define _(a)    (FLAG==1) ? a : mock()
#endif // _MOCK_ADAPTOR
```

(그림 7) 어댑터 매크로 함수

```
nRet = HAL_EEPROM_GET_PORT_SDA();
nRet = _(nRet);
if ( nRet == HI)
{
    u08DataBlock =(U08)(((*((U08*)&u16AddressToWrite + 1)) & 0x07) << 1);
    if (_(WriteI2c1ByteData( 0xa0 | u08DataBlock)) == OK)
```

(그림 8) 재구성 코드

```
#define _DEV_EEPROM_C_

/*********************************
 *  [Include File]
 *********************************/
#include    "Config/DevHeader.h"
#include    "DevEEPROM.h"
#include    "Mock.h"
```

(그림 9) Mock.h를 include한 레거시 코드

5.2 어댑터 구현 및 적용

(그림 7)는 (그림 8)에서 보여지는 어댑터의 기능을 제공하는 매크로 함수가 정의된 것을 나타낸다. 전처리 기능을 적용하여 순수 개발 코드만을 빌드할 수 있게끔 한 것을 확인할 수 있다. (그림 9)는 레거시 코드에 (그림 7)의 헤더 함수를 include한 것을 나타낸다.

(그림 8)는 (그림 3)의 코드를 재구성한 결과의 일부를 나타낸다. 간단한 형태의'_()'라는 매크로 함수를 어댑터로 적용한 것을 확인할 수 있다. 즉, HAL에서 제공하는 HAL_EEPROM_GET_PORT_SDA 함수와 또 다른 디바이스 드라이버 함수 WriteI2c1ByteData은 어댑터를 적용하여 제어가 Mock으로 제어가 가능하게 하였다.

5.3 어댑터 컨트롤러 설계 및 구현

(그림 10)은 테스트 케이스의 일부를 나타낸다. CMock은 Mock을 적용하기 이전에 초기화 기능을 제공한다. 즉, CMock을 호출하면서 Mock이 활성화된다. DMock은 CMock과 반대로 Mock을 비활성화 시킨다. (그림 10)에서는 테스트 케이스 실행 이전 그리고 이후의 단계에서 각각 Cmock과 Dmock을 호출하는 내용을 보여준다. RMock은 Mock을 생성하여 적용하는 기능을 제공한다. (그림 10)에서는 CBMock_WriteDataToMemory라는 함수를 Mock으로 구현하여 함수 포인터의 형태로 전달하는 내용을 보여준다.

6. 어댑터 적용 전, 후의 단위테스트 케이스 비교

본 절에서는 레거시 코드에 어댑터를 적용하기 전과 이후의 단위테스트 케이스를 비교하고 각각에 대한 condition/decision

```
int SETUP(UTsDevEEPRom)
{
    CMock();

    return 0;
}

int TEARDOWN(UTsDevEEPRom)
{
    DMock();

    return 0;
}

void UTcWriteDataToMemory1()
{
    BOOL bRet;
    int i= 0;

    RMock(&CBMock_WriteDataToMemory);

    for(; i<1000; i++){
        bRet = WriteDataToMemory(0, "test", 4);
        MT_ASSERT(bRet == OK);
    }
}
```

(그림 10) 어댑터 컨트롤러 사용 예

```
void UTcWriteDataToMemory1()
{
    BOOL bRet = WriteDataToMemory(0, "test", 4);
    MT_ASSERT(bRet == OK);
}

void UTcWriteDataToMemory2()
{
    BOOL bRet = WriteDataToMemory(1, "test", 4);
    MT_ASSERT(bRet == OK);
}

void UTcWriteDataToMemory3()
{
    BOOL bRet = WriteDataToMemory(100, "test", 4);
    MT_ASSERT(bRet == OK);
}

void UTcWriteDataToMemory4()
{
    BOOL bRet = WriteDataToMemory(EEPROM_DATASIZE - 1, "test", 4);
    MT_ASSERT(bRet == OK);
}

void UTcWriteDataToMemory5()
{
    BOOL bRet = WriteDataToMemory(EEPROM_DATASIZE, "test", 4);
    MT_ASSERT(bRet == OK);
}
```

(그림 11) 어댑터 적용전의 단위테스트 케이스

테스트 커버리지 결과를 살펴본다. 단위 테스트 케이스는 EEPROM 디바이스 드라이버에서 제공하는 Public함수에 대해서 작성되었다. 함수의 입력 값은 Boundary Value Analysis[11]에 기반하여 선정하였으며 함수 인자들 간의 조합방법으로는 Default Combination[12]을 사용하였다.

6.1 어댑터 적용 전의 단위테스트 케이스

(그림 11)은 어댑터 적용 전의 단위테스트 케이스들의 일부를 나타낸다. 테스트 케이스는 CUnit[6]기반으로 자체 개발한 Test Framework를 사용하여 구현되었다. UTcWrite DataToMemory1()라는 첫 번째 테스트 케이스에서 Write DataToMemory함수의 첫 번째 인자는 메모리 주소(Memory Address)를 0으로 설정하는 것을 나타낸다. 두 번째 인자는 메모리 주소에 기입할 문자열을 나타낸다. 세 번째 인자는 기입할 문자열의 바이트 크기를 나타낸다. 즉, 테스트 케이스는 테스트 대상 함수를 사용하여 메모리 주소 0번지에 "test"라는 문자열 4byte를 기입한다. 그리고 함수의 반환 값이 OK라는 매크로 상수의 값과 일치할 경우에는 테스트 케이스의 결과를 'PASS'로 처리하게 된다. 이 경우 2장에서

Name	Function coverage	Uncovered functions	Condition/decision coverage
d:\?????쑫?\UnitTest_MoreTest\Source\Device\EEPRom_01\DevEEPRom.c	62%	8 - 5 = 3	36%
WriteI2c1ByteData	0%	1 - 0 = 1	0%
ReadI2c1ByteData	0%	1 - 0 = 1	0%
SendI2cAckSignal	0%	1 - 0 = 1	0%
ReadDataFromMemory	100%	1 - 1 = 0	35%
WriteDataToMemory	100%	1 - 1 = 0	47%

(그림 12) condition/decision 테스트 커버리지

Name	Function coverage	Uncovered functions	Condition/decision coverage
d:\?????쑫?\UnitTest_MoreTest\Source\Device\EEPRom_01\DevEEPRom.c	87%	8 - 7 = 1	78%
WriteI2c1ByteData	0%	1 - 0 = 1	0%
SendI2cAckSignal	100%	1 - 1 = 0	50%
ReadDataFromMemory	100%	1 - 1 = 0	75%
ReadI2c1ByteData	100%	1 - 1 = 0	75%
WriteDataToMemory	100%	1 - 1 = 0	97%

(그림 13) condition/decision 테스트 커버리지

설명하였듯이 테스트 케이스는 테스트 대상 함수가 호출하는 함수의 결과값에 대해서는 어떤 제어도 할 수 없게 되고 때문에 테스트 케이스의 모든 흐름(Flow)을 수행하는 것은 결과적으로 어렵다. 논문의 실험에서는 EEPROM을 제어하는 디바이스 드라이버에 대해서 기존의 레거시 소프트웨어를 기반으로 단위 테스트를 적용하였고 이에 대한 결과를 bullseye[10]라는 상용 테스트 커버리지 측정 도구를 사용하여 확인하였다. 그 결과는 (그림 12)과 같다.

테스트 대상 함수인 ReadDataFromMemory와 WriteData ToMemory의 condition/decision 커버리지는 각각 35%, 47%를 나타내었다.

6.2 어댑터 적용 후의 단위테스트 케이스

어댑터를 적용한 후의 단위 테스트 케이스의 일부는 (그림 10)에서 보였다. WriteDataToMemory함수를 1000번이나 호출하게 된 것은 각각의 호출에 대해서 Mock의 반응을 다양하게 하였기 때문이다. 적용한 테스트 케이스에 대해서 테스트 커버리지를 측정한 결과는 (그림 13)와 같다.

(그림 13)에서는 어댑터 적용 이전과 비교하여 condition / decision 커버리지가 각각 75%, 97%로 크게 향상된 것을 확인할 수 있다.

7. 결 론

단위 테스팅이란 개발자가 자신이 작성한 단위 모듈을 다른 개발자가 작성한 모듈과의 통합 단계로 릴리즈 하기에 앞서 스스로 오류가 없는지 검증하는 작업을 가리킨다. 그리고 이를 위해서 테스트 대상인 단위 모듈을 호출하는 모듈인 테스트 드라이버와 단위 모듈이 참조하는 모듈인 Mock 혹은 스텁을 구현하고 이들을 제어하여 단위 모듈의 동작 상태를 다각도로 검증하게 된다. 그런데 많은 레거시 소프트웨어시스템은 단위 테스팅이 수행되지 않고 구현되어

단위 테스트 케이스를 자산으로 가지고 있지 못하다. 때문에 유지 보수에 매우 중요한 역할을 하는 회귀테스트에 있어서 단위 테스트를 재사용할 수 없는 문제를 야기하게 된다. 이에 본 논문에서는 레거시소프트웨어시스템에 대한 단위테스트 케이스를 만들기 위해서 어떻게 재구성 과정을 수행할 수 있는지 설명하였다. 기존의 레거시소프트웨어시스템의 각각의 단위 모듈들은 이미 구현된 다른 단위 모듈들과 밀접하게 통합되어 있기 때문에 이들을 따로 떼어내지 않고 별도의 Mock 혹은 테스트 스텁을 구성하는 것은 어려운 일이다. 본 논문에서는 이를 해결하기 위해서 모듈간의 통합 접점에 어댑터라는 구조를 적용하였다. 그리고 어댑터를 제어하는 어댑터 컨트롤러를 사용하여 기존 연결된 레거시 모듈로 연결을 하거나 혹은 테스트를 위해서 Mock 그리고 테스트 스텁으로의 연결을 가능하게 하였다. 그리고 이러한 방법을 적용하여 단위 모듈에 대한 테스트 커버리지가 크게 올라갔다는 것을 제시하였다. 또한 재구성에 따른 순수 개발 코드의 기능적인 동작이 그대로 보존될 수 있다는 것을 보였다. 단위 테스트 케이스들은 레거시 소프트웨어의 회귀 테스트 때마다 계속적으로 사용될 수가 있다. 때문에 점차적으로 유지 및 보수가 어려워지는 상황에서 기존의 레거시소프트웨어에 대한 단위 테스트 케이스들을 준비하고 관리하는 것은 매우 중요하고 시급한 과제이다. 본 논문에서는 이를 위한 기법을 크게 세 단계로서 제시하였고 이에 대한 세부적인 내용에 대해서는 사례를 기반으로 자세하게 설명하였다. 실제 다양한 레거시소프트웨어 시스템에 대해서 재구성 작업을 수행할 때 본 논문에서 제시하는 세 가지 기법을 범용적으로 적용할 수 있을 것이다. 그러나 세부적인 방법으로는 논문에서 설명한 사례와 같이 간단한 매크로 함수를 사용할 수도 있을 것이며 이 외의 다른 방법을 사용할 수도 있을 것이다. 이에 대한 많은 사례 연구가 나올 수 있기를 기대해 본다. 그리고 본 논문이 제시하는 내용을 기반으로 차후 재구성의 자동화 방향과 Mock을 비롯한 테스트 케이스의 자동 생성 방안에 대한 연구가 함께 병행될 수 있다면 레거시 소프트웨어를 유지 및 보수하는데 큰 발전을 기대할 수 있을 것이다.

참 고 문 헌

[1] Girish Parikh, Independent Consultant, "Exploring the world of software maintenance: what is software maintenance?," ACM SIGSOFT Software Engineering Notes, Volume 11, Issue 2(April 1998), Page: 49-52.

[2] David Saff, Michael D. Ernst, M.I.T., "Test factoring: focusing test suites for the task at hand," Proceedings of the 27th international conference on Software engineering, Pages: 656-656.

[3] Wikipedia Dictionary Available at URL: http://en. wikipedia.org/wiki/Unit_test, 2007년 6월

[4] David Bernstein Available at URL: https://users.cs.jmu.edu/bernstdh/web/common/help/stubs-and-drivers.php, 2007년 5월.

[5] Wikipedia Dictionary Available at URL: http://en.wikipedia.org/wiki/Legacy_software, 2007년 6월.

[6] SF.net Available at URL: http://cunit.sourceforge.net/, 2007년 6월.

[7] Wikipedia Dictionary Available at URL: http://en.wikipedia.org/wiki/Mock_Object, 2007년 6월.

[8] Tom Mens, "A Survey of Software Refactoring," IEEE Transactions On Software Engineering, month 2004

[9] Wikipedia Dictionary Available at URL: http://en.wikipedia.org/wiki/Regression_test, 2007년 6월.

[10] Bullseye Testing Technology Available at URL: http//www.bullseye.com, 2007년 6월.

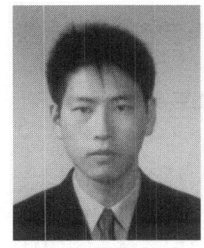
[11] Wikipedia Dictionary Available at URL: http://en.wikipedia.org/wiki/Boundary_value_analysis, 2007년 6월.

[12] Yong Rae Kwon, Department of Computer Science Korea Advanced Institute of Science and Technology, "Test Case Evaluation."

[13] Bret Pettichord, "Design for Testability," Pacific Northwest Software Quality Conference, Portland, Oregon, October, 2002.

[14] Harry M. Sneed, "Reengineering for Testability," Workshop on Software Reengineering(WSR), Bad Honnef, May, 2006.

[15] Wikipedia Dictionary Available at URL: http://en.wikipedia.org/wiki/ISO_9126, 2007년 9월.



문 중 희

e-mail : joonghee.moon@samsung.com

2002년 인하대학교 컴퓨터공학(학사)

2006년 숭실대학교 정보과학대학원(석사)

2006년~현 재 숭실대학교 전자계산학과 (공학박사) 대학원(박사과정)

2002년~현 재 삼성전자 S/W연구소 선임연구원

관심분야 : 소프트웨어 시험, 레거시소프트웨어 재공학, 리팩토링

이 남 용

e-mail : nylee@computing.soongsil.ac.kr

1979년 숭실대학교 전자계산학(학사)

1983년 고려대학교 경영대학원 경영정보학(석사)

1993년 Mississippi State University 경영정보학(박사)

1999년~현 재 숭실대학교 컴퓨터학과 교수

관심분야 : 소프트웨어 품질 시험 / 인증