컴포넌트의 테스트가능성 향상을 위한 래퍼 설계와 구현

송호진° 최은만
동국대학교
nemoz@dongguk.edu°, emchoi@dgu.ac.kr

A Design and Implementation of Wrapper for Improving Component Testability

Ho Jin Song°, Eun Man Choi
Dongguk University

요 약

컴포넌트는 서드파티(third-party)소스코드 형태로 배포되지 않는 등 여러 가지 요인으로 인해 테스트가능성(testability)이 낮아지게 된다. 이렇게 낮은 테스트가능성으로 인하여 개발된 컴포넌트가 실제로 재사용되었을 때 테스트에 많은 어려움이 따르게 된다. 이러한 테스트가능성을 향상시키기 위한 방법으로서 컴포넌트에 테스트를 위한 래퍼(wrapper)를 적용할 수 있다. 본 연구에서는 테스트가능성을 향상시키기 위한 방법인 래퍼를 설계하고 구현하는 방법에 대한 연구를 수행하였다.

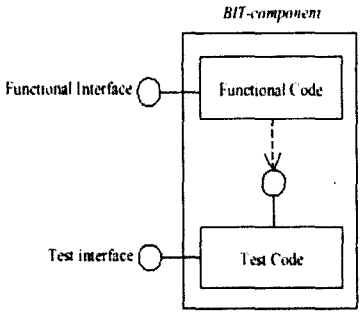
1. 서 론

최근 컴포넌트 기반 개발(component-based development)이 소프트웨어 개발의 많은 분야에서 관심을 가지고 개발이나 연구를 수행하고 있다. 이는 컴포넌트 기반 개발을 통해 개발 시간과 비용의 절감 등을 유도할 수 있는 장점을 얻을 수 있기 때문이다. 신뢰성 있는 컴포넌트의 사용 여부는 곧 소프트웨어의 품질과 직결되는 중요한 사항이라 할 수 있다. 컴포넌트의 신뢰성을 보증하기 위한 검증활동은 반드시 필요하며 컴포넌트 사용자는 검증을 위해 컴포넌트에 대한 테스트를 수행함으로써 컴포넌트의 신뢰성을 검증하고, 오류를 발견해 낼 수 있다. 개발된 컴포넌트는 어떠한 환경에서 다시 사용될지 알 수 없다. 서드파티를 통해 배포된 컴포넌트나 재사용을 위해 개발한 컴포넌트인 경우 실제 전개될 환경에서의 테스트는 반드시 필요하다. 하지만 컴포넌트는 자신이 아닌 서드파티에 의해 개발되어 배포되었을 경우 소스코드가 배포되지 않는 등의 문제로 컴포넌트 내부의 동작이나 구조를 알 수 없어 낮은 테스트 가능성을 나타낸다. 낮은 테스트 가능성은 사용자가 실제 환경에서 사용한 컴포넌트의 테스트를 어렵게 만든다. 이러한 낮은 테스트 가능성을 향상 시킬 수 있는 방법으로서 컴포넌트의 내부에 테스트를 수행하는 테스트 함수를 내장하는 BIT(Built In Test)방법이 있다[1]. 하지만 BIT는 컴포넌트의 개발자가 컴포넌트를 개발 할 때 같이 개발해야 하는 부담과 컴포넌트에 내장된 테스트 함수가 실제 컴포넌트의 전개 시에는 불필요한 부분으로서 전개될 시스템에 불필요한 부담을 초래할 수 있는 단점이 있다.

이러한 BIT의 단점을 개선하고 컴포넌트의 테스트 가능성을 향상시킬 수 있는 방안으로서 컴포넌트에 테스트를 위한 래퍼를 이용하는 방법을 제안한다.

2. 컴포넌트의 테스트

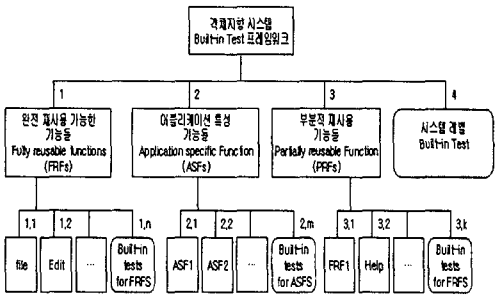
컴포넌트의 낮은 테스트가능성을 향상시키기 위한 여러 가지 방법 중 컴포넌트의 셀프테스트를 위한 BIT 방법은 컴포넌트 내에 테스트를 수행하는 함수를 내장함으로써 컴포넌트의 테스트가능성을 향상시킬 수 있다[1].



(그림 1) BIT 컴포넌트

BIT 컴포넌트는 컴포넌트의 본래 기능을 수행하는 기능 인터페이스와 하나 이상의 테스트 인터페이스가 합쳐진 형태이다. BIT방법을 사용함으로써 테스트 사용자는 블

랙박스(기능) 또는 화이트박스(구조)에 의해 생성된 테스트 케이스를 모두 시험해 볼 수 있다. 하지만 이러한 BIT 컴포넌트는 실제로 테스트가 끝난 후 시스템에 전개되었을 때 전개될 시스템에 불필요한 부담을 초래할 수 있는 요소로 작용할 수 있게 되는 단점을 가질 수 있다. 시스템은 단순히 하나의 컴포넌트만으로 구성되는 것은 아니다. 시스템을 구성하고 있는 다수의 컴포넌트가 모두 BIT 함수를 내장하고 있는 컴포넌트라면 이러한 컴포넌트들은 시스템에 있어 사용할 기회가 적은 부담이 될 수 있는 요소로 작용할 수 있다. 다음 그림 2는 객체지향 소프트웨어 시스템의 BIT 프레임워크를 나타내고 있다[2].

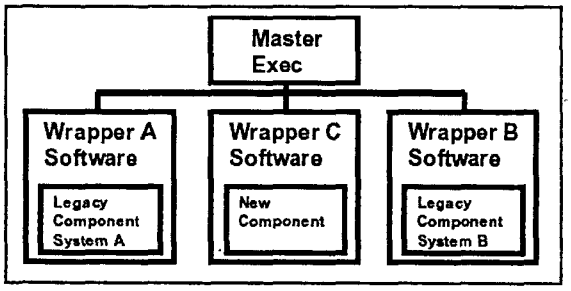


(그림 2) 시스템 Built-in Test 프레임워크

이러한 단점을 해결할 수 있는 방안으로써 본 연구에서는 컴포넌트의 래퍼를 이용하여 컴포넌트의 테스트가능성능을 향상시킬 수 있는 방법을 제안한다.

3 컴포넌트 테스트를 위한 래퍼

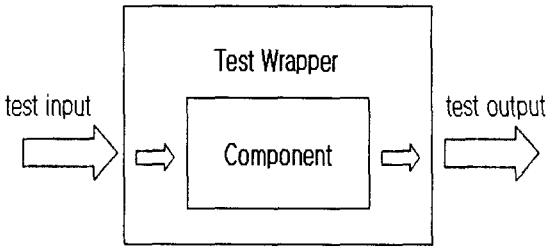
컴포넌트의 래퍼란 컴포넌트를 둘러 싸고 있는 형태의 또 다른 컴포넌트나 클래스라 말할 수 있다. 이러한 래퍼를 통하여 기존의 컴포넌트를 직접 수정 없이 재 사용하거나 기능의 변경, 성능의 개선 등을 이룰 수 있다[3]. 다음 그림3은 기존의 소프트웨어와 새로운 소프트웨어를 소프트웨어 래퍼를 통해 구성한 시스템을 보여주고 있다[4].



(그림 3) 래퍼를 이용한 시스템의 구성

3.1 래퍼의 설계

BIT와 같이 컴포넌트의 내부에 테스트 함수를 내장하는 것이 아니라, 본래의 컴포넌트 외부에 테스트 수행을 돕는 래퍼를 통해 테스트를 하게 된다. 즉 BIT컴포넌트의 내장된 테스트 함수 대신 컴포넌트 외부의 래퍼가 그 역할을 대신하게 된다. 그림 4는 테스트를 위한 소프트웨어 래퍼에 대해 잘 나타내고 있다.



(그림 4) 컴포넌트 테스트를 위한 래퍼

본래 기능을 수행하는 컴포넌트의 외부에 그림 4와 같이 테스트를 위한 래퍼를 구현한다. 래퍼는 컴포넌트의 주위를 감싼 형태로써 오직 테스트를 위한 입력과 출력 인터페이스를 가지고 있다. 이러한 래퍼를 통해 컴포넌트의 내부와 외부의 행위를 검사할 수 있으며 본래의 컴포넌트로부터 테스트를 위한 검증 코드를 완전히 분리 할 수 있다. 테스트 입력은 테스트를 위하여 개발된 테스트 드라이버를 통해 래퍼를 구동하여 테스트를 수행한다. 래퍼는 테스트 하고자 하는 컴포넌트의 인터페이스와 직접적으로 연결되어 테스트 드라이버를 통해 입력된 테스트 데이터를 컴포넌트의 인터페이스를 통해 테스트를 수행하게 된다. 래퍼가 테스트 데이터를 통해 컴포넌트를 구동하고 컴포넌트는 다시 수행 결과를 래퍼에 되돌려주게 되면 래퍼는 이러한 결과를 테스트를 수행하는 테스트 드라이버에게 돌려준다. 테스트 드라이버는 이러한 테스트 결과를 출력하게 되며 테스트를 위한 래퍼는 이러한 테스트를 통해 발견된 결함을 보고하게 된다. 테스터는 단지 테스트 하고자 하는 컴포넌트의 래퍼에 테스트 케이스 등을 입력하고 래퍼에서 가공된 결함 보고서(defect report)와 테스트 드라이버의 테스트 결과만을 주시하면 된다.

3.2 래퍼의 구현

테스트를 위한 래퍼를 구현하기 위한 여러 가지 방법 중 상속을 통한 래퍼 구현의 예가 다음 그림 4에 나타나 있다[5]. 예제의 래퍼는 연결리스트를 구현한 List클래스의 인터페이스를 이용하여 Wrapper 클래스를 구현하고 이를 확장하여 테스트 어써션(assertion)을 래퍼에 내장한 자바 코드의 일부분을 나타낸 것이다. Wrapper 클래스는 wrappedObject와 isEnable 두 개의 필드로 구성되어 있다. wrappedObject는 래핑 될 객체를 참조하는 역할을 하게 되며 isEnable는 수행 시 어써션을 구성하는 쿼리를 구성하는 역할을 하게 된다.

```
Public class Wrapper
{
    public Wrappable wrappedObject = null;
    public static CheckingPrefs isEnabled = null;
}
public class  $chx_Wrap_List extends Wrapper implements
List
{
 //...
    public int $chx_get_elementCount() {
       return wrappedObject.elementCount();
}
 public Object removeFirst() {
 //...
  if ( isEnabled.precondition() ) {
   // checkPre performs the actual
   // precondition check.
   checkPre$RemoveFirst$List();
  }

  return (($chx_Orig_List)wrappedObject).removeFirst();
 }
}
```

(그림 5) 상속을 이용한 래퍼의 구현 예

3.2 래퍼의 효과 분석

테스트를 위한 래퍼를 통해 기대할 수 있는 효과를 다음 표1에 기능, 성능, 재사용성의 항목으로 분류하여 정리하였다.

(표 1) 테스트를 위한 래퍼의 기대 효과

항목	기대 효과
기능	· 테스트 코드와 기능 코드의 분리 · 테스트 종류별 래퍼의 개발을 통한 선택적 래핑의 적용성
성능	· 컴포넌트의 실제 사용시 래퍼로 적용된 테스트 코드를 분리하여 전개함으로써 운용상에 불필요한 부담을 줄일 수 있음
재사용성	· 래퍼로 분리된 테스트 코드는 소프트웨어의 개발 생명주기 전체에 걸쳐 재사용 될 수 있음

이렇게 긍정적인 기대 효과에도 불구하고 발생할 수 있는 단점도 있다. 컴포넌트가 변경될 때 마다 래퍼가 다시 작성되거나 어떠한 목적으로 인해 래퍼가 변경되어야 할 경우 컴포넌트의 개발자나 사용자가 래퍼를 매번 직접 변경해야 하는 문제가 발생할 수 있다. 또 다른 문제로써 컴포넌트 개발자가 컴포넌트를 제작 할 때마다 테스트를 위한 래퍼를 따로 제작해야 하는 부분은 부담스러운 요소로 작용할 수 있다. 이러한 문제점을 개선하기 위해 테스트를 위한 래퍼를 자동 또는 반 자동으로 생성할 수 있는 방법이 필요하다.

4. 결론 및 향후 연구

소프트웨어의 결함을 찾기 위한 테스트는 매우 중요한 작업이다. 컴포넌트는 언제, 어떠한 환경에서, 어떻게 사용될 지 알 수 없는 경우가 많다.이미 테스트되어 검증된 컴포넌트라 하더라도 컴포넌트 기반 소프트웨어의 개발에는 실제 컴포넌트가 다른 컴포넌트와 통합되어 운용되기 전에 컴포넌트에 어떠한 장애나 결함이 있는지를 찾아내는 것이 매우 중요하다. 컴포넌트는 자체적으로 개발된 컴포넌트도 있을 수 있지만 대부분 여러 곳에서 개발된 컴포넌트들을 통합하여 사용하게 된다 하지만 여러 곳에서 개발된 컴포넌트들은 서로 다른 언어로 작성되거나 그 밖에 다른 요소들로 인해 상호 이질성을 나타낼 수 있으며 이러한 특성들은 컴포넌트의 테스트 능력을 저해하는 요소로서 작용하게 된다. 즉, 여러 상호간에 이질적인 요소를 가지고 있는 소프트웨어 컴포넌트의 통합이나 소프트웨어의 직접적인 변경을 하지 않고 기능의 확장이나 변경 등을 위해 사용되는 래퍼를 컴포넌트의 사용자 입장에서 컴포넌트의 통합 측면의 테스트에 이용함으로써 컴포넌트의 낮은 테스트가능성을 개선하고 래퍼의 재사용을 통해 어떠한 시점에서도 테스트를 원활이 수행할 수 있다는 장점을 얻을 수 있다.

향후 연구할 과제로써 컴포넌트에 포함된 최소 정보로써 컴포넌트 테스트를 위한 래퍼를 자동으로 생성할 수 있는 데이터를 구성하는 것에 대한 연구와 이를 토대로 실제 구현을 통한 실험과 성능에 대한 평가를 수행하여야 할 것이다.

참고 문헌

[1] J Vincent," Built-In-Test Vade Mecum □ Part I A Common BIT Architecture" , IST 1999-20162 Component+ European project, 1999.

[2] Yingxu Wang, " A Method for Built-in Tests in Component-Based Software" , IEEE International Conference on Software Maintenance and Reengineering(CSMR' 99), March 1999. pp.186-189.

[3] Timithy Fraser, " Hardening COTS Software with Generic Software Wrappers" , In IEEE, Symposium on Security and Privacy, May. 1999. pp. 2-16.

[4] Stephen H. Edwards " A Framework For Practical, Automated Black-box Testing of Component-Based Software" , Proceedings of 1st International Workshop on Automated Program Analysis, Testing and verification, June. 2000. pp.97-111.

[5] Roy Patric Tan," An Assertion Checking Wrapper Design for Java" , Specification and Verification of Component-Based Systems Workshop, September. 2003.