컴포넌트 테스트를 위한 래퍼의 자동 생성에 관한 연구

(Automated Generation of Wrapper to Test Components)

송 호 진 † 최 은 만 ††
(Hojin Song) (Eun Man Choi)

요 약 미리 만들어진 컴포넌트를 조립하여 새로운 소프트웨어를 개발하는 방법은 개발비용과 기간을 획기적으로 줄일 수 있다는 장점으로 전통적인 방법의 대안이 되고 있다. 하지만 컴포넌트를 통합 조립하면서 신뢰도를 확인하고 새 환경에 맞는지 테스트하는 과정이 복잡하고 비용이 많이 소요된다면 효과적인 방법이 될 수 없다. 대규모 시스템에 효율적으로 사용, 조립될 컴포넌트들은 원시코드 형태로 배포되지 않아서 새 환경에 적합한지 시험하고 관찰하는 철저한 테스트가 어렵다. 이러한 문제점을 해결하기 위하여 컴포넌트가 재사용되었을 때 쉽게 검증되도록 미리 테스트 모듈을 내장한 Built-In 테스트 방법이 있지만 컴포넌트의 기능이 다양하고 복잡하게 되면 컴포넌트에 포함된 테스트 모듈의 규모가 커지고 다양하고 융통성 있는 테스트가 어려워진다. 이 논문에서는 컴포넌트의 Built-In 테스트 기능을 대체할만한 컴포넌트 테스트를 위한 래퍼(wrapper)를 제안하고 이를 설계, 구현하여 실용성을 보였다. 래퍼를 자동 생성하여 테스트하면 컴포넌트의 테스트 준비 과정에 드는 노력이 줄어들고 테스트를 다양한 측면에서 테스트해 볼 수 있다.

키워드 : 컴포넌트 기반 소프트웨어 개발, 소프트웨어 테스트, 테스트 자동화, 테스트 도구

Abstract Assembling new software systems from prepared components is an attractive alternative to traditional software development method to reduce development cost and schedule dramatically. However, if separately developed components are tested, integrated and verified with unreasonable effort and high cost, it would not be an effective way to software development. Components are not distributed in the shape of white-box source code so that should be hard to validate and test in new application environment. For solving this problem, built-in tester components are suggested to check the contract-compliance of their server components. If components have various and complex function, built-in tester should be heavy and unflexible to test in composition of components. This paper suggests enhancing automated wrapper technique which substitutes with built-in tester components and shows the usability of the wrapper by design and implementation. Component testing in this way reduces the cost and effort associated with preparation of component testing and makes the various test experiments in components assembly.

Key words : Component-Based Software Development, Software Test, Test Automation, Test Tools

1. 서 론

컴포넌트 기반 개발(Component-Based Development)의 비전은 과거 전통적인 개발 방법에서 지적된 오버헤드, 즉 되풀이 되는 유사한 소프트웨어의 반복적인 개발, 높은 신뢰도를 유지하기 위하여 여러 단계에 걸쳐 진행되는 테스트 작업 등을 피하기 위하여 널리 사용될 컴포넌트를 미리 준비하고 이를 효과적으로 조립하는 것이다. 대규모의 소프트웨어를 이루는 부분을 미리 개발된 컴포넌트를 조립하여 개발한다면 개발 시간과 비용은 상당히 줄일 수 있고 어플리케이션의 품질은 높일 수 있다. 이러한 기대는 개발된 컴포넌트를 검증하고 통합하여 새로운 시스템을 구성하는 과정이 전통으로 사용된 처음부터 새로 개발하고 테스트하는 방

· 본 연구는 동국대학교 논문게재장려금 지원으로 이루어졌음
† 비 회 원 : 한국 NCR 테라데이타
nemoz@hanmail.net
†† 종신회원 : 동국대학교 컴퓨터멀티미디어공학과 교수
(Corresponding author임)
논문접수 : emchoi@edu.ac.kr
심사완료 : 2004년 3월 5일
2005년 6월 13일

법보다는 수월하다는 전제가 깔려 있다. 따라서 컴포넌트를 사용한 개발은 조립 과정뿐만 아니라 이미 개발된 컴포넌트가 새로운 어플리케이션의 환경에 맞게 제대로 작동하는지 테스트하고 검증하는 작업이 효과적으로 이루어져야 한다.

신뢰성 있는 컴포넌트의 사용 여부는 곧 소프트웨어의 품질과 직결되는 중요한 사항이라 할 수 있다. 컴포넌트의 신뢰성을 보증하기 위한 검증활동은 단순히 문법과 기능적인 측면만이 아니라 배치 환경과 의미 수준의 호환성 검증 작업을 수행함으로써 컴포넌트로 조립된 시스템의 신뢰성을 보증하고, 오류를 발견해 낼 수 있다. 재사용을 위하여 개발된 컴포넌트는 어떠한 환경에서 다시 사용될지 알 수 없다. 즉 서드파티를 통해 배포된 컴포넌트는 실제 전개될 환경에서의 테스트가 반드시 필요하다.

하지만 서드파티에 의해 개발되어 배포된 소프트웨어 컴포넌트들은 소스코드가 공개되지 않는 블랙박스 형태로 재사용되므로 컴포넌트 내부의 동작이나 구조를 알 수 없어 낮은 테스트 가능성을 지닌다. 단순히 컴포넌트가 제공하는 또는 요구하는 기능의 API를 점검하는 수준의 낮은 테스트 가능성(testability)은 사용자가 실제 환경에서 사용한 컴포넌트를 철저히 테스트하는 것을 어렵게 한다.

블랙박스로 제공되는 컴포넌트의 낮은 테스트 가능성을 향상시킬 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 부분으로써 전개될 시스템에 부담을 초래할 수 있는 단점이 있다. 이러한 BIT의 단점을 개선하고 컴포넌트의 테스트 가능성을 향상시킬 수 있는 방안으로서 컴포넌트 테스트를 위한 래퍼(Wrapper)를 제안하고 이를 자동화할 수 있는 방안을 실험 적용하였다. 보편적으로 래퍼를 적용하는 이유는 기존 코드의 변경 없이 부가적으로 기능을 추가하거나 변경 또는 다른 플랫폼으로의 이식을 위해 사용한다. 즉 수행 코드의 변경 없이 적용할 수 있는 장점을 테스트에 이용할 수 있다.

논문의 구성은 2장에 여러 가지 컴포넌트 테스트 방법에 대한 소개와 이 연구에서 다루는 컴포넌트 테스트 부분을 명확히 구획 지었고, 컴포넌트 배치 및 시스템 테스트 가능성 향상을 위한 방법들과 이를 위한 BIT 방법을 다루었다. 3장은 이 논문에서 제시한 BIT를 대치하는 래핑 방법이 어떤 것인지 구체적인 예를 들어 설명하였고, 4장에서는 테스트 래퍼를 자동으로 생성하는 도구를 설계하고 구현하는 과정을 보였다. 5장은 실험 결과를 나타내고 이를 평가 분석하였다.

2. 관련 연구

2.1 컴포넌트 테스트 방법

컴포넌트 테스트는 컴포넌트 개발할 때, 개발자에 의해 배포 이전에 테스트하는 단계와 컴포넌트를 구입하여 사용하는 단계에서의 테스트, 두 단계로 구분할 수 있다. 보통 컴포넌트 단위 테스트가 컴포넌트 개발자에 의해 테스트 되는 단계라 말할 수 있다. 테스트 대상 컴포넌트를 고립시키고 주로 컴포넌트 내부의 기능과 로직에 대한 시험을 하는 것이다.

컴포넌트의 통합 테스트와 시스템 테스트는 컴포넌트의 사용자에 의해 테스트되는 단계라 할 수 있으며 이를 기존의 구조적 프로그램의 테스트 절차와 비교하여 각 모듈의 단위 테스트는 컴포넌트 테스트, 모듈의 통합 테스팅은 컴포넌트의 배치(deployment) 테스트의 절차로 나타낼 수 있다[2].

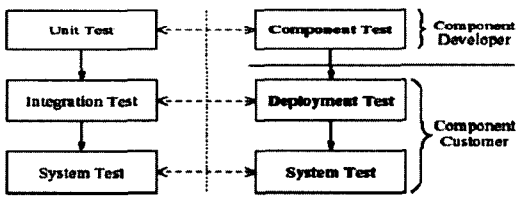

그림 1 컴포넌트 테스트 절차

실제로 컴포넌트는 독립적으로 개발되거나 상업적으로 판매된 컴포넌트를 시스템 통합 시간 전에 테스트를 통해 결함을 발견해 내는 것이 매우 중요하다. 컴포넌트 복잡성의 증가로 인하여 컴포넌트의 품질을 검증하는 문제는 매우 중요하고 시급하다. 국내외적으로 컴포넌트 품질을 검증하기 위해 많은 연구들이 수행되어져 왔다. 그러나 컴포넌트 개발 표준이 정의되어 사용되고 있는 반면 컴포넌트 시험을 위한 표준은 아직 명확히 정의되지 않았다. 대표적인 컴포넌트 테스트 연구는 NIST (National Institute of Standard and Technology/IT Laboratory)로써 컴포넌트의 상태 명세를 유한 상태로 명세하여 데이타 흐름과 제어 흐름을 나타내고 이 흐름 정보를 이용하여 테스트 케이스를 자동 생성하여 컴포넌트가 상호작용하는 다른 컴포넌트와 정상적으로 기능을 수행하는지를 검증한다. 또 호주에서는 컴포넌트를 잘 정의된 인터페이스를 갖는 소프트웨어로 정의하고, 제3의 기관에서 컴포넌트를 시험하기 위해 개발사가 제공하는 문서 및 도구 등을 제안하였다. 또한 한국에서도 한국소프트웨어 컴포넌트 컨소시엄 및 한국전자통신 연

구원에서도 컴포넌트 테스트를 위한 품질 요소들을 정의하고 효율적인 시험을 위한 자동화 도구들을 개발하고 있다. 이밖에도 여러 가지 소프트웨어 컴포넌트 테스트를 위한 기법과 도구들이 연구되고 있다.

컴포넌트를 조립하여 사용할 것을 고려하면 다음과 같은 두 가지 측면의 테스트 관점이 필요하다. 하나는 서비스를 제공하는 측면, 서버 입장과 서비스를 요청하는 측면, 즉 클라이언트 입장이다. 그림 2에서 사용되는 주체인 deployed 컴포넌트는 새로운 어플리케이션에서 사용될 때, 클라이언트 컴포넌트가 요청하는 서비스를 제공하는 역할도 하고, 반대로 다른 서버 컴포넌트에게 서비스를 요청하기도 한다. 따라서 컴포넌트를 사용하여 시스템을 구성하는 단계에서는 이러한 두 가지 역할이 잘 수행되고 있는지 체크하여야 한다.

일반적으로 컴포넌트를 조립 재구성하는 형태의 개발에서 자주 발생할 수 있는 오류는 다음 두 가지 경우이다[3]. 첫째는 컴포넌트 조립과정에 컴포넌트가 요구하는 서비스가 명시된 것과 실제 런타임에 수행되어 제공하는 것이 다른 경우이다. 둘째는 컴포넌트의 클라이언트가 조립하려는 대상 컴포넌트의 제공 서비스를 잘못 요구하거나 오해하는 오류이다. 서버 컴포넌트로부터 원하는 서비스를 받았는지, 클라이언트가 컴포넌트의 서비스를 요청하였을 때 명시된 대로 작동하는지 두 가지 측면의 테스트가 컴포넌트 조립 과정에서 관건이 된다.

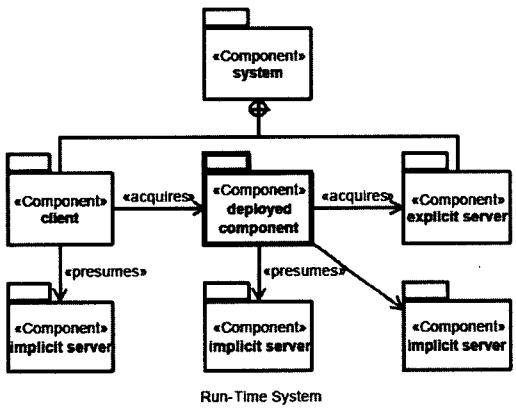

그림 2 컴포넌트 테스트의 주요 대상

그림 2에 표시한 acquire와 presume 두 관계가 컴포넌트가 조립되는 새 환경에서 잘 작동하는지 테스트하여야 한다. 따라서 단순히 컴포넌트가 포함하고 있는 메소드의 구문적인 확인만으로는 부족하다. 조립과정에서 일어날 수 있는 여러 가지 서비스의 요구와 제공의 다양한 컴비네이션을 테스트 하려면 체계적인 접근으로서 자동화를 하나의 가능한 해결책이라 볼 수 있다. 이 논문에서는 컴포넌트가 실제로 조립, 통합되는 단계의 테스트를 목적으로 하는 테스트 래퍼의 생성 방안에 대한 연구를 진행하였다.

2.2 테스트 가능성 향상

컴포넌트 기반 소프트웨어 개발은 전통적인 소프트웨어 개발 조직과는 다르게 여러 개발 조직에 의해 개발되는 특징이 있다. 이러한 점으로 인해 각 조직에 의해 개발된 컴포넌트가 실제 통합되었을 때 올바르게 동작하는지에 대한 테스트가 반드시 필요하다. 실제 COTS 컴포넌트와 같이 완전히 다른 개발 조직에서 개발되어진 컴포넌트를 사용하여 시스템을 구성하고자 할 때 이를 통합하여 시험하는 것은 완성된 시스템의 신뢰도를 높이기 위한 필수 단계라 할 수 있다.

그러나 재사용이 활발한 COTS 컴포넌트는 원시코드가 제공되지 않아 테스트에 필요한 정보를 얻기 어렵고 이러한 문제점으로 인해 신뢰성을 검증하기 위한 테스트가 어렵게 된다. 블랙박스 형태로 주어진 컴포넌트들은 제공하는 기능을 사용하는 간단한 API나 요구하는 인터페이스에 대한 점검 등이 고작이다. 이렇게 컴포넌트는 낮은 테스트 가능성을 나타내게 되며 이러한 테스트 가능성을 향상시키기 위한 방안이 필요하다.

컴포넌트의 테스트 가능성을 향상시키기 위한 방법으로서 컴포넌트 행위에 대한 부가적인 정보를 필요로 한다. 컴포넌트의 사용자에게 컴포넌트에 대한 부가적인 정보의 제공을 통해 컴포넌트의 사용 범위와 정보에 대한 이해를 높일 수 있는 방안이 연구되고 있다[4]. 컴포넌트의 기능과 컴포넌트가 포함하고 있는 함수에 대한 정보, 이들 컴포넌트에 대한 연관 관계를 XML의 형태로 제공함으로써 사용자는 컴포넌트의 소스코드 분석이 없어도 이를 참고함으로써 컴포넌트를 올바르게 사용할 수 있으며 이를 테스트에 이용함으로써 컴포넌트의 테스트 가능성을 향상시킬 수 있다. 컴포넌트에 대한 연관 관계를 XML의 형태로 나타냄으로써 자동화 도구는 이를 해당하는 플랫폼에 적용할 수 있는 래퍼를 생성할 수 있는 기반 정보로써 이용할 수 있다. 자동화 도구는 이를 통해 해당 컴포넌트의 플랫폼(Java, .NET 등)에 해당하는 래퍼 코드를 자동으로 생성하게 된다.

이와 같은 부가적인 정보를 통하여 테스트 가능성을 향상시킬 수 있지만 이 방법으로는 궁극적으로 중요한 컴포넌트의 런타임 동작에 대한 시험은 한계가 있다. 런타임에 조립될 수 있는 다른 컴포넌트들과의 연관 정보를 통하여 컴포넌트의 테스트 가능성 향상을 위한 테스트 래퍼를 자동생성 할 수 있다.

2.3 테스트 내장형(Built-in Test) 컴포넌트

BIT[3,4]는 컴포넌트의 테스트가능성을 높이기 위하여 테스트 슈트를 컴포넌트 안에 미리 내장하는 개념이

다. 대단위 규모의 효과적인 재사용을 위해서는 컴포넌트가 블랙박스 형태를 취하게 되며 이렇게 되면 테스트 슈트를 작성하는 일은 큰 부담이 된다. 따라서 테스트 슈트를 작성하는 일을 미리 컴포넌트 개발 과정에 앞당겨 미리 내장시키고 컴포넌트의 사용자는 BIT 컴포넌트의 BIT 인터페이스를 통해 테스트 오퍼레이션을 호출하여 테스트를 수행하고 수행 결과를 전달 받는다. 객체지향 소프트웨어 컴포넌트는 캡슐화와 같은 특성으로 인해 외부에서 내부의 행위나 구조를 알 수 없어 테스트에 어려움이 따른다. 이러한 문제점을 BIT를 이용하여 해결할 수 있다.

BIT 컴포넌트는 컴포넌트 본래 기능을 수행하는 기능 인터페이스와(functional interface)와 하나 이상의 테스트 인터페이스(test interface)가 합쳐진 형태로 되어 있다. 컴포넌트 사용자는 이러한 인터페이스의 접근을 통해 컴포넌트의 테스트를 비롯한 계속적인 검증 작업을 수행하게 된다.

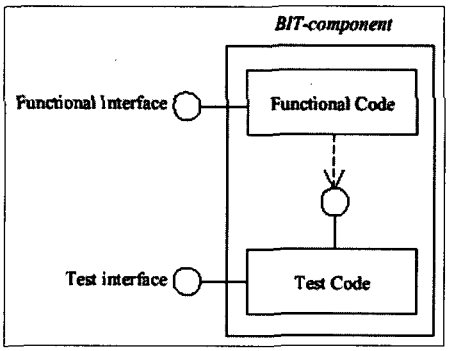

그림 3 BIT 컴포넌트의 개념

BIT는 재사용의 측면에서 소프트웨어 테스트에 소요되는 비용과 수고를 줄일 수 있으며, 소프트웨어 품질을 높일 수 있다. 또한 안정적인 테스트 모델을 제공하고 서드파티 컴포넌트들의 행위를 모니터링 할 수 있으며 유연성 있는 테스트 제어 능력을 제공하게 된다. 하지만 BIT는 컴포넌트 개발자가 컴포넌트 개발 시 테스트 코드도 같이 개발해야 하는 부담이 있다. 또한 컴포넌트가 변경될 때마다 이를 테스트하는 코드도 같이 변경해야 하며 테스트 코드가 컴포넌트에 포함되어 있기 때문에 실제 컴포넌트가 전개될 시스템에 부담을 초래할 수도 있는 단점이 있다. 이러한 문제점을 테스트 래퍼를 이용함으로써 좀 더 효율적인 테스트를 수행할 수 있도록 하려는 것이 이 연구의 목표이다.

3. 컴포넌트 테스트를 위한 래핑 방법

컴포넌트의 래퍼란 컴포넌트를 둘러싸고 있는 형태의 또 다른 컴포넌트나 클래스라고 말할 수 있다. 이러한 래퍼를 통하여 기존의 컴포넌트를 직접 수정 없이 재사용 하거나 기능의 변경, 성능의 개선 등을 이룰 수 있다[6]. 다음 그림 4는 오래전부터 개발되어 사용되고 있는 레거시 컴포넌트와 새로운 컴포넌트를 소프트웨어 래퍼를 통해 구성한 시스템을 보여주고 있다[7]. 레거시 컴포넌트와 새로운 컴포넌트는 서로 다른 언어 또는 기술을 통해 개발되었을 확률이 높다. 그림 4의 Master Exec는 기존의 컴포넌트 시스템 A와 B와 새롭개 개발된 컴포넌트 C의 서브 시스템으로 구성된 전체 시스템이다. 컴포넌트 시스템 A와 B는 기존에 개발되었던 레거시 컴포넌트이다. 새로운 컴포넌트인 C는 기존의 A,B와는 다른 언어 또는 기술로써 개발된 컴포넌트이다. 이러한 이질적인 요소로 인해 컴포넌트 A,B,C를 통합하여 시스템을 구성하는 것은 쉽지 않다. 이러한 컴포넌트에 대하여 공통적인 언어나 기술을 통한 래퍼를 이용하여 시스템을 구성하게 된다. 컴포넌트 상호간의 호출 시 컴포넌트는 컴포넌트를 직접 호출하는 것이 아니라 컴포넌트를 감싸고 있는 래퍼에게 호출을 요청하게 되고 래퍼는 컴포넌트를 대신하여 호출하고자 하는 컴포넌트의 래퍼에게 호출을 요청한다. 호출을 받은 컴포넌트의 래퍼는 실제의 컴포넌트에게 수행을 요청하고 이에 대한 응답을 받아 요청한 컴포넌트의 래퍼에게 이러한 결과를 되돌려 주며 래퍼는 호출을 요청한 컴포넌트가 인식할 수 있는 결과로 다시 가공하여 컴포넌트에게 결과를 알려주게 된다. 이러한 형태로써 컴포넌트 상호간의 이질적인 요소를 없애 기존의 컴포넌트와 새로운 컴포넌트간의 통합 시스템을 구성할 수 있는 것이다.

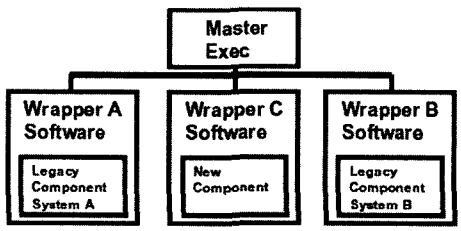

그림 4 래퍼를 이용한 시스템의 구성

이러한 소프트웨어 래퍼를 컴포넌트의 테스트에 이용할 수 있다. 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. 이렇게 래핑을 통하여 런타임에 점검해야할 다양한 형태의 서비스 호

출과 요구 서비스를 확인할 수 있다. 다음 그림 5는 컴포넌트 테스트를 위한 래퍼에 대해 나타내고 있다.

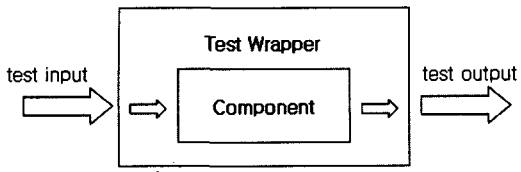

그림 5 컴포넌트 테스트를 위한 래퍼

본래 기능을 수행하는 컴포넌트의 외부에 그림 5와 같이 테스트를 위한 래퍼를 구현한다. 래퍼는 컴포넌트의 주위를 감싼 형태로써 오직 테스트를 위한 입력과 출력 인터페이스를 가지고 있다. 이러한 래퍼를 통해 본래의 컴포넌트로부터 테스트를 위한 검증 코드를 완전히 분리해 낼 수 있다. 래퍼는 테스트 하고자 하는 컴포넌트의 인터페이스와 직접적으로 연결되어 입력된 테스트 데이타를 래퍼의 테스트 인터페이스를 통해 테스트를 수행하고 테스트 결과를 출력해 주게 된다. 테스트를 위한 래퍼를 구현하는 여러 가지 방법 중 테스트 하고자 하는 구현 클래스의 인터페이스를 구현하여 테스트 목적으로 이용하게 하는 방법이 있다. 그림 6에 나타낸 예제의 래퍼는 연결리스트를 구현한 List클래스의 인터페이스를 이용하여 Wrapper 클래스를 구현하고 이를 확장하여 테스트 assertion을 래퍼에 내장한 자바 코드의 일부분을 나타낸 것이다[8]. Wrapper 클래스는 wrappedObject와 isEnable의 두개의 필드로 구성되어

```
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 6 테스트 래퍼의 구현 예

있다. wrappedObject는 래핑 될 객체를 참조하는 역할을 하게 되며 isEnable는 수행할 때 assertion을 구성하는 쿼리를 담당하는 역할을 하게 된다. 즉 isEbable필드는 테스트를 수행하기 위해 필요한 테스트 컴포넌트나 컴포넌트에 포함되어 있는 각각의 메소드 테스트에 필요한 condition을 체크하기 위해 작성된 필드이다. 예를 들어 단방향 리스트 컴포넌트를 검사하고자 할 때 테스트 수행 전 필요한 precondition이 만족되었는지를 알아 볼 필요가 있다고 하면 위의 isEnable의 필드를 이용하여 검사를 수행하게 된다. wrappedObject 객체 필드는 실제적으로 테스트하고자 하는 컴포넌트의 객체를 참조하는 필드로서 이를 통하여 실제적인 컴포넌트의 테스트가 이루어지게 된다.

위 그림 6에 요약되어 작성된 테스트 래퍼는 Java언어로 작성된 단방향 리스트의 인터페이스를 구현하도록 되어 있다. 여기에 구현된 removeFirst() 메소드는 단방향 리스트의 첫 번째 요소를 삭제하는 메소드를 테스트하는 메소드를 구현한 것이다. 먼저 removeFirst() 메소드를 테스트 하기 위한 precondition을 검사하여 이를 만족한다면 실제적인 List 객체를 참조하는 wrapped-Object의 removeFirst() 메소드를 호출하여 결과를 되돌려주게 된다. 이러한 결과를 테스트 어써션을 통해 비교하여 최종적으로 기대하는 결과와 일치하는지의 여부를 판단하여 테스트의 성공 실패 여부를 알려주게 되는 것이다.

3.1 래퍼의 효과 분석

래퍼를 이용해 컴포넌트의 변경 없이 시스템을 구성할 수 있는 장점은 컴포넌트의 테스트에도 긍정적인 요소로 작용할 수 있을 것으로 생각되었다. 또한 BIT의 테스트 코드가 기존의 컴포넌트에 포함되어 있음으로 인해서 발생할 수 있는 문제점을 개선하고자 하는 방안으로써 컴포넌트 테스트를 위한 테스트 래퍼가 사용될 수 있을 것으로 생각되어 테스트를 위한 래퍼로 얻을 수 있는 기대 효과를 분석하였다.

테스트를 위한 래퍼를 통해 기대할 수 있는 효과를 다음 표 1에 기능, 성능, 재사용성의 항목으로 분류하여 정리하였다.

위에서 언급한 테스트 래퍼와 기능 코드의 분리는 앞으로 언급할 컴포넌트 테스트의 자동화에 있어서 반드시 필요한 요소로 작용한다. 기존의 BIT가 컴포넌트의 내부에 테스트 코드와 기능 코드가 같이 포함되어 있기 때문에 컴포넌트의 변경으로 인해 테스트 코드가 변경될 필요가 있을 경우 이를 자동화 하여 포함시키기 어려운 문제가 있을 수 있다. 이러한 문제점도 테스트 래퍼를 이용하면 테스트 코드와 기능 코드가 완전히 분리되어 있는 상태이기 때문에 자동화 된 테스트 코드를

표 1 테스트를 위한 래퍼의 기대효과

항목	기대효과
기능	• 테스트 코드와 기능 코드의 분리 • 테스트 종류별 래퍼의 개발을 통한 선택적 래핑의 적용성
성능	• 컴포넌트의 실제 사용 시 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

생성하여 쉽게 적용할 수 있는 장점이 있다.

이렇게 긍정적인 기대 효과에도 불구하고 발생할 수 있는 문제점도 있다. 컴포넌트가 변경되거나 어떠한 목적으로 인해 래퍼가 변경되어야 할 경우 컴포넌트의 개발자나 사용자가 래퍼를 매번 직접 변경해야 하는 문제가 발생할 수 있다. 또 다른 문제로서 컴포넌트 개발자가 컴포넌트를 제작할 때마다 테스트를 위한 래퍼를 따로 제작해야 하는 부분은 부담스러운 요소로 작용할 수 있다. 이러한 문제점을 개선하기 위해 테스트를 위한 래퍼를 자동 또는 반자동으로 생성할 수 있는 방법이 필요하다. 예를 들어 컴포넌트간의 호출 순서나 기존의 컴포넌트와는 다른 새로운 컴포넌트가 포함되어 시스템이 구성되는 상황에서 테스트 래퍼가 자동화 되지 않았을 경우를 생각해 볼 수 있다. 컴포넌트로 구성된 시스템의 크기가 작을 경우에는 수동으로 작성하는 테스트 래퍼를 사용할 수 있을 것이다. 하지만 시스템의 규모가 방대하고 이를 조금씩 변경해 가며 여러 번 테스트를 수행하여야 할 경우가 발생할 수 있다. 컴포넌트간의 메소드 호출 순서를 조금씩 바꿔가며 테스트를 수행하여야 할 경우 이에 대해 매번 테스트에 필요한 래퍼의 코드를 변경시켜야 하는 작업은 쉽지 않다. 이를 위한 테스트 래퍼의 자동화는 반드시 필요하다 할 수 있다.

4. 테스트 래퍼의 자동 생성

4.1 자동화 도구의 구조 및 분석

서브 시스템을 구성하는 컴포넌트 간의 통합 테스팅을 수행하기 위해 테스트 래퍼를 자동 생성하는 테스트 래퍼 자동화 도구의 테스트 수행 구조는 다음과 같다. 여기서 자동화 할 부분은 컴포넌트 통합 묘사(Component Integration Description)을 통하여 컴포넌트의 앞의 그림 6과 같은 컴포넌트의 메소드나 호출 관계를 시험할 수 있는 테스트 래퍼를 자동 생성하는 부분이다. 테스트 래퍼 자동화 도구의 구조는 서브시스템을 구성하는데 필요한 후보 컴포넌트들을(Candidate Component) 수집하여 이를 통합하여 테스트 하고자 하는데 목적이 있다. 본 연구에서 테스트 자동화에 대한 요구사

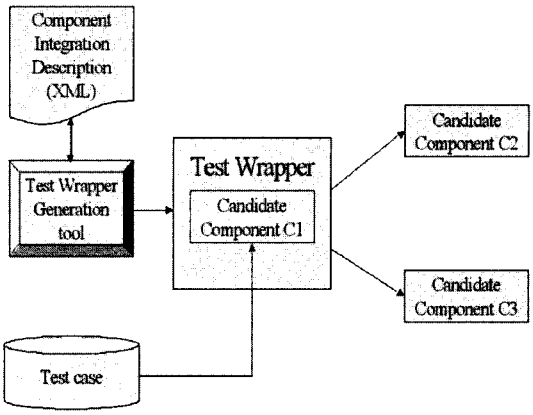

그림 7 테스트 래퍼 자동화 도구를 이용한 통합 테스트 수행 구조

항으로써 컴포넌트는 정형적으로 명세된 컴포넌트의 기능 묘사를 반드시 포함하고 있어야 한다. 즉 테스트 자동화를 위한 입력으로써 자동화 도구의 사용자는 정형화 된 형태의 컴포넌트 통합 묘사를 작성하여야 한다. 즉 사용자는 직접적으로 래퍼를 작성하는 대신 컴포넌트의 기능과 호출관계를 포함하고 있는 컴포넌트 통합 묘사를 작성함으로써 이를 테스트의 자동화에 이용할 수 있다.

후보 컴포넌트 C2와 C3를 이용하여 기능을 수행하게 되는 컴포넌트 C1을 테스트 하기 위한 래퍼를 테스트 래퍼 생성 도구가 각각의 컴포넌트의 기능과 호출 관계를 표현한 XML 형태의 컴포넌트 통합 묘사(Component Integration Description)를 통해 테스트 래퍼를 자동 생성해 내게 된다.

그림 8에 나타낸 컴포넌트 통합 묘사에서는 우리가 시험하고자 하는 컴포넌트 C1에 포함되어 테스트 하려 하는 메소드가 VirtualMethod라는 이름으로 묘사되어 있으며 여기에는 해당 메소드의 타입과 매개변수에 대한 정보를 포함하고 있다. ViutualMethod의 내부에는 해당 메소드가 다른 컴포넌트의 메소드를 호출하는 호출 정보가 execMethod로 명시되어 있다. 첫 번째로 나타난 VirtualMethod의 이름은 deposit이며 이 메소드의 반환형은 int, 매개 변수는 String형의 accNum과 Integer형의 amount임을 알 수 있다. 이러한 deposit메소드를 테스트하기 위해 필요한 다른 컴포넌트와의 호출관계는 execMethod에 포함되어 있다. 그림 8에 나타낸 deposit를 테스트 하기 위해 필요한 컴포넌트 호출 순서는 첫 번째로 CastingClass의 par2Couple메소드를 호출하여 결과를 반환받고 두 번째로 bank 객체의 put라는 메소드를 호출하는 것을 알 수 있다. 이렇게 컴포넌트의 이름과 형, 매개변수 및 호출관계를 포함한 CID는

```
<CompontIntegrationDescription>

        <testPackage name="test.bank"/>
        <testClass name="BankTest"/>
        <realPackage name="real"/>
        <createObject class="RealComp" objectName="bank"/>


        <VirtualMethod type = "int" name="deposit" parameter_type="String" parameters="accNum"
         parameter_type2="Integer" parameters2="amount">


                <execMethod class="CastingClass" name="par2Couple" type="Couple" putResultIn="c">
                        <parameter value="accNum"/>
                        <parameter value="amount"/>
                </execMethod>
                <execMethod object = "bank" name = "put">
                        <parameter value="c"/>
                </execMethod>
                <execMethod object = "bank" name = "howMuch" type="Couple" putResultIn="aCouple">
                        <parameter value="accNum"/>
                </execMethod>
                <execMethod class="CastingClass" name="Couple2Int" type="int" putResultIn="output">
                        <parameter value = "aCouple"/>
                </execMethod>
        </VirtualMethod>

        <VirtualMethod type = "int" name="withdraw" parameter_type="String" parameters="accNum"
          parameter_type2="Integer" parameters2="amount">
                                                .
                                                .
                                                .
        </VirtualMethod>

        <VirtualMethod type = "int" name="balance" parameter_type="String" parameters="accNumber">
                                                .
                                                .
                                                .


        </VirtualMethod>

</CompontIntegrationDescription>
```

그림 8 컴포넌트 통합 묘사(Component Integration Description)의 예

원하는 형태의 테스트 코드 형태로 쉽게 변환될 수 있음을 알 수 있다. 이러한 컴포넌트 통합 묘사를 테스트 래퍼 생성도구에서 파싱하여 테스트 래퍼를 자동 생성하게 된다. 만약 어떠한 이유로 인해 컴포넌트의 내용이나 호출관계에 변경이 생겼을 경우 사용자는 컴포넌트 통합 묘사만을 간단히 변경함으로써 변경된 테스트 래퍼를 자동으로 생성해 낼 수 있다. 또한 이러한 묘사는 컴포넌트 테스트 래퍼의 자동 생성의 목적만이 아니라 사용자가 사용할 컴포넌트에 대한 이해를 돕는 목적으로써도 사용할 수 있는 장점이 있다. 즉 소스코드를 사용자가 직접 분석하는 것 보다는 테스트를 목적으로 개발된 XML형태의 CID를 통해 사용자가 컴포넌트의 기능 및 호출관계를 이해하고 분석하여 컴포넌트를 사용하는데 있어 좀더 유용한 정보로 사용할 수 있다. 이렇게 자동 생성된 테스트 래퍼는 테스트를 위해 개발된 테스트 케이스를 입력으로 테스트를 수행하게 된다. 테스트 케이스는 여러 가지 형태가 있을 수 있지만 본 논문에서는 JUnit 프레임워크를 이용해 작성된 테스트 케이스를 자동 생성된 테스트 래퍼의 입력으로 사용하였다. 테스트를 위해 개발된 여러개의 테스트 케이스가 테스트 래퍼를 통해 실행되며 이를 통해 오류를 찾아내어 분석하는 형태의 구조를 나타낸다. 보편적으로 소프트웨

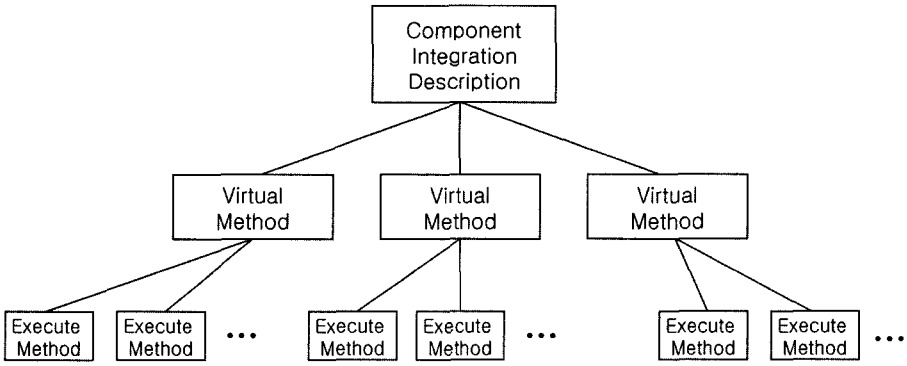

그림 9 컴포넌트 통합 묘사의 트리 구성

```
if(qName == "VirtualMethod"){
    execMethodCnt = 0;
    virtualMethodCnt++;
    virtualMethod.put("index", new Integer(virtualMethodCnt));
    virtualMethod.put("length"+ new Integer(virtualMethodCnt), new Integer(atts.getLength()));

    for (int i = 0; i < atts.getLength(); i++) {
      virtualMethod.put("parameters" + virtualMethodCnt+""+ i, atts.getValue(i));
    } // end for


 }// end if

if(qName == "execMethod"){

   execMethodCnt++;
   execMethod.put("index" + new Integer(virtualMethodCnt) , new Integer(execMethodCnt));
   execMethod.put("length"+ new Integer(execMethodCnt), new Integer(atts.getLength()));
   execMethod.put("execMethodObjectType"+ new Integer(virtualMethodCnt)+ ""+ new
   Integer(execMethodCnt), atts.getQName(0));

   for(int i=0; i < atts.getLength(); i++){
     execMethod.put("parameters" + virtualMethodCnt+""+execMethodCnt+""+i, atts.getValue(i));

     if(atts.getQName(i) == "putResultIn"){
       execMethod.put("putResultIn"+ new Integer(virtualMethodCnt) + "" + new
       Integer(execMethodCnt),atts.getValue(i));
     }
     else{
       execMethod.put("putResultIn"+ new Integer(virtualMethodCnt) + "" + new
       Integer(execMethodCnt),"null");
     }

                                          .
                                          .
   }
}

                                          .
                                          .
```

그림 10 트리를 구성하는 구현 코드의 일부분

어 컴포넌트는 단순한 각각의 클래스, 모듈, Java Bean 이나 COM과 같은 객체로 구성되어 있다. 본 논문에서는 Java로 구현된 객체지향 클래스에 대해 테스트를 적용하였다. 하지만 자동화 도구의 구현 여부에 따라 Java 외에 .NET 및 그 외의 컴포넌트 객체에 대해 적용할 수 있는 래퍼를 생성할 수 있다. 이러한 테스트 방법은 좀 더 큰 규모의 컴포넌트에도 쉽게 일반화 할 수 있다.

4.2 테스트 래퍼 자동생성 도구의 구현

테스트 래퍼의 자동생성 도구의 구현은 XML을 파싱할 수 있는 SAX(Simple API for XML)를 이용한 자바 코드로 구현하였다. SAX는 이벤트 기반 방식의 XML파서들에 구현되어 있는 API로써 파서가 XML 도큐먼트를 읽어나갈 때 요소(element)의 시작이나 끝을 만났다는 등의 이벤트를 해당 이벤트 핸들러에게 알려주는 방식을 말한다. 파서가 도큐먼트를 읽어가면서 의미 있는 이벤트가 발생할 때마다 등록된 이벤트 핸들러를 불러들여 발생한 이벤트를 전달해 준다[9]. 이렇게 한 요소를 읽어 들이기 시작할 때 발생하는 이벤트를 처리할 때 자바의 기본 유틸리티 API에 포함되어 있는 해쉬테이블 함수를 이용하여 XML로 구성된 컴포넌트 통합 묘사를 다음과 같은 트리 형태로 구성한다.

이렇게 트리 형태로 구성함으로써 각 메소드의 호출 관계와 순서를 정의하고 이를 실제 해당하는 자바 코드로 변환할 수 있는 형태로 저장하게 된다. 해쉬 테이블을 이용하여 트리를 구성하는 구현 코드의 일부분은 다음과 같다. 메소드의 호출순서는 구성된 트리의 좌측으로부터 우측으로 수행되게 된다. 현재 CID의 기술 순서에 의해 트리가 구성되지만 CID내에 호출 우선순위를 부여함으로써 트리를 구성할 수 있도록 기술할 수도 있다.

XML 파싱을 통하여 최초에 VirtualMethod가 발견되면 VirtualMethod를 카운트하게 된다. 그림 8에 나타낸 CID를 예로 들어 설명하면 최초 deposit의 이름을 가지고 있는 VirtualMethod를 파싱하게 되면 이를 VirtualMethod1이라는 스트링으로 변환하여 해쉬테이블에 인덱스로써 사용하게 된다. 이를 이용하여 해쉬테이블에 VirtualMethod의 타입 및 매개변수에 대한 정보들을 함께 저장한다. 다음 execMethod를 파싱하게 되면 이는 execMethod의 순서와 개수를 카운트하여 다음과 같이 execMethod11, execMethod12, execMethod-13과 같은 형태의 스트링으로 변환하게 된다. exec-Method의 뒤에 붙은 숫자 중 앞에 나타나는 숫자 1은 VirtualMethod를 의미하는 것이며 뒤에 나타난 숫자 1,2,3은 exeMethod의 호출 순서를 나타낸 것이다. 그 외에 테스트 코드의 생성을 위해 부가적으로 필요한 정보들은 기본적으로 virtualMethod와 execMethod로 구성된 트리구조를 바탕으로 구성된 해쉬테이블에 저장하게 되고 이렇게 저장된 정보를 바탕으로 각각의 컴포넌트의 호출관계를 시험하는 테스트 래퍼의 코드를 자동생성하게 된다.

4.3 자동화 도구를 이용한 테스트 래퍼의 생성

작성된 자동화 도구를 이용해 자동 생성해 낸 테스트 래퍼의 코드는 다음과 같다. testXXX로 명명된 각각의 메소드는 구현 시에 필요한 테스트를 수행하여 결과를 돌려주는 역할을 하게 된다. 즉 하나의 테스트 함수는 최소단위의 기능을 시험하는 하나의 단위로 볼 수 있다.

위에 자동생성된 테스트 래퍼의 코드는 해쉬테이블로 구성된 트리 구조로부터 생성된 것이다. 해쉬 테이블 내에 저장된 VirtualMethod의 이름과 리턴타입, 매개변수를 통하여 필요한 테스트 코드를 생성해 낸 것이다. 이에 포함되는 호출 시험은 execMethod로부터 생성된 것이다. execMethod는 두 가지 종류로 구분할 수 있다. 그림 8의 CID 예제에서 살펴보면 execMethod의 속성에는 object와 class의 두 종류로 나누어 진 것을 알 수 있다. class의 어트리뷰트를 포함하는 execMethod는 해당 class의 객체 생성을 필요로 하는 것이다. object 어트리뷰트를 포함하는 execMethod는 CID의 앞부분에 나타난 createObject 엘리먼트에 포함된 내용을 바탕으로 객체를 생성한다. 이렇게 해당 클래스의 객체 생성은 자동 생성된 테스트 래퍼 코드에 생성자를 통해 객체를 생성하게 된다. 그림 8의 CID와 그림 11의 자동생성된 코드를 비교해보면 VirtualMethod는 하나의 테스트 함수, execMethod는 테스트 함수에 포함된 컴포넌트 호출 나타낸 것으로서 직접 매칭될 수 있음을 볼 수 있다. 자동생성된 테스트 래퍼의 수행을 위해 필요한 입력으로써 컴포넌트의 호출관계를 시험할 수 있는 테스트 케이스가 필요하다. 자동 생성된 테스트 케이스 코드는 컴파일 된 후 JUnit로 작성된 테스트 케이스의 실행 요청

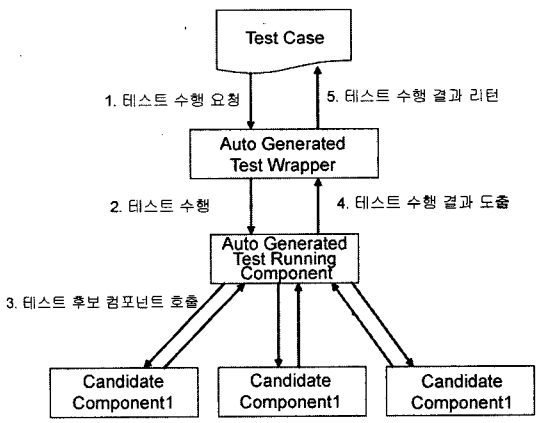

그림 11 테스트 수행구조

```
package test.bank;
import test.bank.*;
import real.*;

public class testImpWrapper{

   RealComp bank;
   CastingClass testObject1;

   public testImpWrapper{
          bank = new RealComp();
          testObject1 = new CastingClass();
   }

public int testdeposit (String accNum ,Integer amount ) {
          Couple c = testObject1.par2Couple(accNum,amount);
          bank.put(c);
          Couple aCouple = bank.howMuch(accNum);
          int output = testObject1.Couple2Int(aCouple);
          return output;
          }

public int testwithdraw (String accNum ,Integer amount ) {
          Couple c = testObject1.par2Couple(accNum,amount);
           bank.get(c);
           Couple aCouple = bank.howMuch(accNum);
          int output = testObject1.Couple2Int(aCouple);
          return output;
          }

public int testbalance (String accNumber ) {
          Couple aCouple = bank.howMuch(accNumber);
          int output = testObject1.Couple2Int(aCouple);
          return output;
          }

public int testbalance (String accNumber ) {
          Couple aCouple = bank.howMuch(accNumber);
          int output = testObject1.Couple2Int(aCouple);
          return output;
          }

}
```

그림 12 자동 생성된 테스트 레퍼의 코드

을 받아 자동으로 생성된 테스트 래퍼의 실행결과를 받아 요청한 테스트 케이스에 실행결과를 돌려주는 역할을 하게 된다. 돌려받은 결과는 테스트 케이스에 포함되어 있는 테스트 어써션을 통해 결과를 비교하여 테스트의 성공, 실패 여부를 알려주게 된다.

먼저 테스트 케이스로부터 실행 요청을 받아 실제적인 테스트 래퍼의 코드를 동작시키는 인터페이스의 역할을 하는 테스트 수행 코드의 작성이 필요하다. 이는 단순히 테스트 케이스와 테스트 래퍼간의 인터페이스 역할만을 수행하는 코드로써 작성되게 된다.

5. 실험 및 결과

컴포넌트 통합 묘사를 통해 자동 생성된 테스트 래퍼가 정상적으로 수행되어 올바른 결과를 나타내는지에 대한 실험을 실시하였다. 이를 위해 JUnit 프레임워크를

```
package test.bank;

public class testWrapper extends TestInformation implements Bank{

        public int deposit(String accNum, Integer amount){
                int ris = testDriver.testdeposit(accNum, amount);
                return ris;
        }

        public int withdraw(String accNum, Integer amount){
                int ris = testDriver.testwithdraw(accNum, amount);
                return ris;
        }

        public int balance(String accountNumber){
                int ris = testDriver.testbalance(accountNumber);
                return ris;
        }
}
```

그림 13 테스트 수행 코드

이용한 테스트 케이스를 작성하고 이를 실제적으로 수행한 결과로써 실험을 수행하였다. 테스트 케이스 코드는 테스트 시나리오에 맞추어 자동 생성된 테스트 래퍼가 이를 읽어들여 수행할 수 있는 형태로 작성해야 한다. 테스트 케이스 코드의 생성자에서 자동 생성된 테스트 래퍼 객체를 생성하여 생성된 테스트 래퍼 객체에 테스트에 필요한 값을 매개 변수를 통해 래퍼에 넘겨주게 된다. 테스트를 위해 작성된 테스트 케이스 코드는 다음과 같다. 테스트 케이스 코드는 테스트를 위해 기 작성된 것으로서 본 논문에서의 자동화 도구를 통한 자동 생성의 범주 외의 내용으로 기술하였다. 추후 CID에 추가적인 테스트 케이스 생성을 위한 정보를 추가하여 자동화 하는 방안을 모색할 필요가 있다. 아래 작성된 테스트 케이스 코드는 오류가 발생하지 않을 것을 전제로 하여 작성된 테스트 케이스 코드로서 은행 계좌 123에 500을 입금한 후 다시 499를 출금하는 시나리오를 바탕으로 작성된 JUnit 테스트 케이스 코드이다. 테스트 코드의 생성자를 통해 테스트 수행 코드의 객체를 생성하고 이를 통해 테스트 래퍼 코드의 수행 코드를 호출하여 테스트 한 결과를 테스트 어써션을 통해 비교하고 테스트 수행 결과를 JUnit 프레임워크를 통해 통보하게 된다.

결과와 같이 자동 생성된 테스트 래퍼 코드는 정상적으로 수행되었음을 알 수 있다. 위의 그림 14의 테스트 결과는 예금되어 있는 500에서 499를 출금한 후 잔고가 1이 남았음을 알려주는 테스트 수행 성공 결과가 JUnit 프레임워크를 통해 통보된 것을 나타낸 그림이다. 다음

```
package test.bank;
import junit.framework.*;
public class BankTest extends TestCase{
        test.bank.testWrapper testBank;
          public BankTest(){
                  testBank = new test.bank.testWrapper();
                  }
          public void testCase_A(){
          int init=testBank.balance("123");
          int afterDeposit=testBank.deposit("123",new Integer(500));
          assertEquals((init+500),afterDeposit);
          int afterWithdraw=testBank.withdraw("123",new Integer(init+499));
          assertEquals(1,afterWithdraw);
          }

}
```

그림 14 테스트 케이스 코드

그림 15의 테스트 수행 결과 예는 계좌 123에 500을 입금한 후 다시 500을 인출하는 테스트 케이스 시나리오를 수행한 결과를 나타낸 것이다. 수행 결과는 기대되는 결과값이 0이지만 리턴된 결과값이 500이라는 메시지를 출력하고 있다. 이는 출금과 관련되어 사용된 컴포넌트의 조건식이 예상과는 다르게 작성되었음을 알려준 것이다. 즉 계좌에서 예금을 출금하기 위한 조건식은 출금액이 예금액 보다 적거나 같아야 한다는 조건을 만족해야 출금이 이루어지게 된다. 하지만 출금을 위해서 작성된 코드는 다음과 같은 조건식을 나타내고 있다.

```
if((sum-imp) > 0){
        sum = sum - imp;
}
```

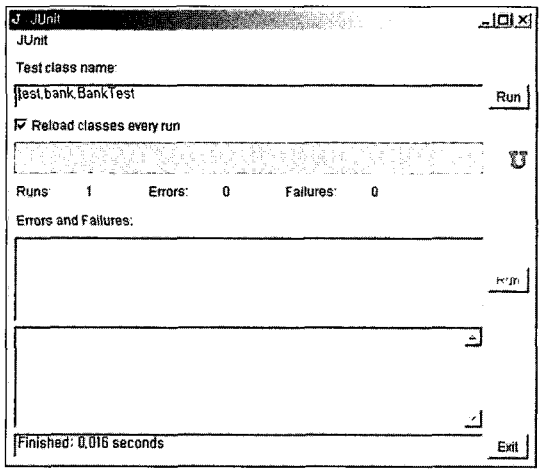

그림 15 테스트 수행 결과

즉 예금계좌보다 출금계좌가 반드시 적어야 하기 때문에 예금액 500에서 500을 출금하는 부분에서의 오류 메시지를 출력하게 된다.

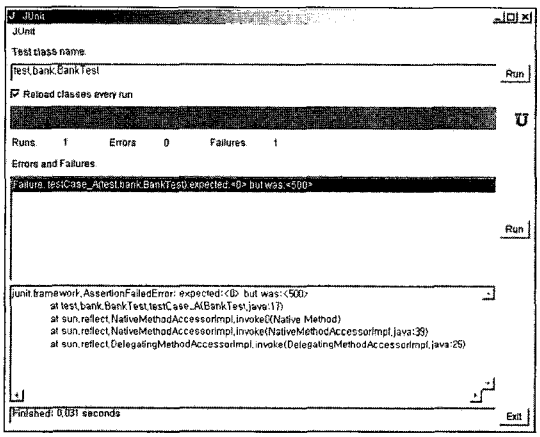

그림 16 오류를 발견한 테스트 수행 결과

또한 여러 컴포넌트간의 호출 순서를 변경함으로써 오류를 발생시켰을 경우에도 오류를 쉽게 찾아낼 수 있었다. 이 실험에서 자동 생성된 테스트 코드가 정상적으로 동작하여 오류를 찾아내었음을 알 수 있었다. 이러한 자동화 생성도구를 이용하여 테스트 수행에 필요한 테스트 코드를 빠르고 쉽게 만들어 낼 수 있음을 알 수 있다.

6. 결론 및 향후과제

소프트웨어의 결함을 찾기 위한 테스트는 매우 중요한 작업이다. 컴포넌트는 언제, 어떠한 환경에서, 어떻게 사용될지 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. 상호간에 이질적인 요소를 가지고 있는 소프트웨어 컴포넌트의 통합이나 소프트웨어의 직접적인 변경 없이 테스트 래퍼를 사용하여 테스트를 원활히 수행 할 수 있는 장점이 있다. 테스트는 같은 시간에 되도록 많이 수행해 볼수록 좋다. 위의 실험 결과를 통해 컴포넌트의 변경이나 호출 관계의 변경에 따른 테스트 래퍼의 변경이 직접 작성하여 변경한 것에 비해 변경된 내용에 대해 CID의 변경만으로도 쉽게 새로운 테스트 래퍼 코드를 생성해 낼 수 있었다. 시스템의 변경이 발생했을 때 이에 대한 테스트가 충분히 필요하다는 점을 감안할 때 이러한 변경에 빠르게 대응할 수 있는 테스트 코드의 작성은 매우 중요하다. 본 논문에서는 이러한 부분을 CID를 통한 테스트 래퍼 코드의 자동화 방안을 통해 좀 더 효율적이고 효과적인 테스트를 수행할 수 있음을 알 수 있었다. 하지만 CID를 기술하기 위해서는 CID를 충분히 이해하고 필요한 형태에 맞춰 기술하는 것이 실제로 테스트 래퍼를 작성하는 것 이상으로 테스트 사용자가 어려움을 느낄 수 있다. 이를 해결하기 위한 방법으로는 사용자가 각 컴포넌트의 호출 관계를 CASE 도구를 이용하여 UML의 시퀀스 다이어그램이나 협력다이어그램을 작성하듯이 각 호출관계를 기술토록 하여 좀 더 쉽게 CID를 작성할 수 있도록 유도할 수 있는 방법을 모색할 필요가 있다. 즉 자동화 도구의 범위 내에 컴포넌트의 정보를 표시하고 이러한 호출 관계를 시퀀스 다이어그램의 형태로 기술 할 수 있는 작성 도구를 개발하거나, 실제 널리 사용되고 있는 CASE 도구로부터 CID에 필요한 정보를 추출하여 CID의 형태로 적용할 수 있는 방법이 필요하다. 현재 대부분의 CASE 도구는 UML명세를 해당 CASE의 도구에 적용할 수 있는 XML로 묘사하고 있으며 이를 통해 해당하는 프로그램의 소스코드를 생성하거나 역으로 다른 형태의 다이어그램으로 생성하는 것이 가능하기 때문에 이를 CID의 형태로 적용할 수 있다. 이를 이용한다면 CID 자체의 복잡한 기술을 좀 더 쉽게 사용자가 작성할 수 있게 하는데 도움을 받을 수 있다. 본 논문에서는 테스트 효율성을 증대시키기 위한 방법으로서 본 연구에서는 컴포넌트의 통합 테스트를 위한 테스트 래퍼의 자동 생성 방법에 대해 연구하고 결과를 제시하였다. 향후 연구할 과제로써 좀 더 복잡한 컴포넌트 통합 묘사를 쉽게 작성할 수 있도록 UML과 같은 설계 도면으로부터 필요한 컴포넌트 통합 묘사를 추출해 내고 이를 통해 테스트 코드를 자동 생

성하는 방안에 대한 연구가 필요하고 실제 EJB나 .NET과 같은 분산 환경의 대규모 비즈니스 컴포넌트의 통합 테스트에 적용하는 방안에 대한 연구를 수행하여야 할 것이다.

참 고 문 헌

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

[2] Y. Wang, G. King, et al., "On Built-in Test Reuse on Object-Oriented Framework Design," ACM Journal of Computing Surveys, Vol. 32, No. 1, March 2000.

[3] H. Grob, "Built-in Contract Testing in Component-based Application Engineering," COLOGNET Joint Workshop on Component-based Software Development and Implementation Technology for Computational Logic Systems Affiliated with LOPSTR, Madrid, Spain, 2002.

[4] Gannon JD, McMullin PR, Hamlet R. Data-avstraction implementation, specification, and Object-Oriented Programming, Nov./Dec. 1995, 8*7): 35-41.

[5] E. Martin, C. Toyota, R. Yanagawa, "Constructing Self-Testable Software Components," Proceedings of 2001 International Conference on Dependable Software Systems and Networks, Goteborg, pp.151-160, Sweden, July 2001.

[6] A. Bertolino, A. Polini, "A Framework for Component Deployment Testing," Proc. ACM/IEEE 25yh International Comference on Software Engineering ICSE 2003, Portland, Oregon, USA, pp.221-231, May 3-10, 2003.

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

[8] Stephen G. 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, pp.97-111, June, 2000.

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

[10] Kal Ahmed, "Professional JavaXML," Wrox, 2002.

[11] A. Bertolino. E. Marchetti. and A. Polini, "Integeration of "Components" to Test Software Components," to appear in *Proceedings of TACos 2003 Workshop at ETAPS 2003*, Warsaw, Poland, April 13, 2003.

[12] A. Bertolino. E. Marchetti. and A. Polini, "WCT: a Wrapper for Component Testing," in *Proceedings of Fidji '2002*, Luxembourg, Novermember 28-29, 2002, to appear in LNCS

[13] R. V. Binder, *Testing Object-Oriented System: Models, Patterns, and Tools*, Addison-wesley, 2000.

[14] J. Cheeman and J. Daniels, UML Component - a Simple Process for Specifying *Component-Based Software*, Addison-Wesley, 2000.

[15] K. Gao, K. Gupta, S. Gupta, and s. Shim, "On Building Testable Software Components," in J. Dean and A. Gravel(Eds) *Proc. ICCBSS, LNCS 2255*, pp.108-121, 2002.

[16] E. Gamma, and K. Beck, "JUnit a Cook's Tour" abilible at: http://www.junit.org

[17] Wang, Y., King, G., Court, i., Ross, M. and Staples, G. "On Built-in Tests in Object-Oriented Reengineering," Proceedings of 5th ACM Symposium on FSE/6yh European Conference on Software Engineering/workshop on Object-Oriented Reengineering(FSE/ESEC/WOOR'97), pp.361-365, 1997.

[18] D. Kung, P. Hsia, and Jerry Gao, "An Overview of Object-Oriented Software Testing," Proceeding of 3rd IEEE High-Assurance Systems Engineering Symposium, Nivember 13-14, 1998, Wasingtin, DC.

[19] Lionel Briand and Yvan Labiche. A UML-based approch to system testing. In *Fourth International Conference o the Unified Modeling Language* (UML '01), pp.194-208, Toronto, canada, October 2001.

송 호 진

1995년 2월~2002년 2월 호서대학교 컴퓨터공학과(학사). 2002년 3월~2005년 2월 동국대학교 컴퓨터공학과(석사). 2004년 4월~현재 한국 NCR 테라데이타 재직 중. 관심분야는 컴포넌트 설계, 소프트웨어 테스트, 데이타베이스 구축

최 은 만

1982년 동국대학교 전산학과(학사). 1985년 한국과학기술원 전산학과(공학석사) 1993년 일리노이 공대 전산학과(공학박사). 1985년~1988년 한국표준연구소 연구원. 1988년~1989년 데이콤 주임연구원. 1998년~2004년 한국정보과학회 소프트웨어공학연구회 운영위원. 2000년~2001년 콜로라도 주립대 전산학과 방문교수. 2002년 카네기멜론대학 소프트웨어공학 과정 연수. 1993년~현재 동국대학교 컴퓨터멀티미디어공학과 교수. 관심분야는 객체지향 설계, 소프트웨어 테스트, 프로세스와 메트릭, Program Comprehension