사용자 화면 중심의 블랙 박스 테스트와 웹 인터페이스 테스트 커버리지를 통한 웹 어플리케이션 테스트 방법

(A Method for Testing Web Applications by Using Black-box Tests based on User Screens and Web Interface Test Coverage)

임 정 희 † 이 시 현 † 장 진 아 † 최 병 주 †† 황 상 철 †††
(Jeunghee Lim) (Sihyun Lee) (Jina Jang) (Byoungju Choi) (SangCheol Hwang)

요 약 웹 어플리케이션은 프레임워크가 제공하는 라이브러리를 재사용해서 구현하며, 다계층 아키텍처를 갖는다. 또한 사용자 요청을 처리하기 위해, 화면에서 시작하여 해당 웹 어플리케이션 구성 컴포넌트를 실행하고 데이터베이스를 경유하여 다시 화면으로 돌아오는 실행 흐름, 즉 비즈니스 로직을 갖는다. 웹 어플리케이션을 효과적으로 테스트하기 위해서는 이러한 웹 어플리케이션의 특징을 반영하는 테스트 방안이 필요하다.

본 논문은 웹 어플리케이션 테스트를 위해 사용자 화면으로 테스트를 수행하되, 웹 비즈니스 로직 상의 인터페이스 테스트 커버리지로 검증하는 방법을 제안한다. 이를 자동화한 테스트 도구인 Testopiacov 를 통해 웹 어플리케이션을 테스트하고 그 결과를 통하여 제안하는 웹 어플리케이션 테스트 방법을 분석한다.

키워드 : 웹 어플리케이션 테스트, 웹 비즈니스 로직, 웹 인터페이스, 블랙박스 테스트, 화이트박스 테스트

Abstract A web application is implemented by reusing the library provided by framework and has hierarchical architecture. Also, to deal with the user request from a screen, the web application has an execution flow, called 'Business Logic', which starts with a screen, executes its composed component and comes back to the screen via database. To test web application effectively, it should reflect the characteristics of web application.

In this paper we propose to test web applications via user screens with the black-box testing approach and verify its source codes with the web interface white-box test coverage that covers all the business logics of the test target and their corresponding interfaces. We analyze the proposing testing method through its tool: Testopiacov .

Key words : Web Application Testing, Web Business logic, Web Interface, Black-box Test, White-box Test

· 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원 사업의 연구결과로 수행되었음(NIPA-2009-(C1090-0903-0004))

† 학생회원 : 이화여자대학교 컴퓨터공학과
jeunghee@ewhain.net
nadalsh@ewhain.net
jajang@ewhain.net
†† 종신회원 : 이화여자대학교 컴퓨터정보통신공학과 교수
bjchoi@ewha.ac.kr
††† 정 회 원 : 삼성 SDS Engineering Methodology Team 책임
hwangsc@samsung.com

논문접수 : 2009년 1월 12일
심사완료 : 2009년 8월 3일

정보과학회논문지: 소프트웨어 및 응용 제36권 제9호(2009.9)

1. 서 론

최근 인터넷 사용 인구와 통신 기술의 급격한 성장으로 웹 어플리케이션은 발전과 변화를 거듭하고 있다. 웹 기반 어플리케이션은 시간과 공간의 제약이 없는 사용성(anytime anywhere)이라는 웹의 강력한 장점을 바탕으로 산업 전반에서 활발하게 사용되고 있다. 웹 어플리케이션의 수요가 늘어나고 웹 어플리케이션이 비즈니스의 핵심 역할을 하면서, 웹 어플리케이션은 갈수록 다양한 기능을 갖는 복잡한 시스템이 되고 있다[1].

웹 어플리케이션의 규모가 커지고 복잡해지면서 품질의 중요성은 더욱 높아지고 있다[1]. 그러나 부족한 개발 비용, 촉박한 개발 일정, 테스트 전문 인력 부족, 개발체계에 대한 인식 부족 등으로 인해, 단위, 통합, 시스

템 테스트를 순차적으로 면밀히 수행하지 못하는 경우가 대부분이다. 일반적으로 웹 어플리케이션을 테스트하는 실무에서는 메소드나 클래스를 단위로 하는 단위 테스트 단계를 건너 뛰고, 웹 페이지, 사용자 요구사항 처리에 관계된 함수들의 집합, 그리고 데이터베이스를 통합한 실행 가능한 단위의 결과물이 개발 되었을 때, 이를 테스트 대상으로 사용자 화면 중심의 블랙박스 기법을 사용해서 테스트를 수행한다[2].

이것은 테스트 드라이버(test driver) 또는 스텁(stub) 없이도 테스트 대상 자체만으로 실행이 가능하기 때문에 테스트를 위한 추가적인 코드를 생성하는데 소요되는 시간과 비용을 절약할 수 있다. 또한, 화면 상에서 테스트를 수행할 수 있으므로 테스트와 개발 산출물에 대한 전문 지식이 없는 테스터라도 비교적 쉽게 테스트를 수행할 수 있다는 장점이 있다. 즉, 실행 가능한 규모의 개발물을 하나의 테스트 단위로 하여 화면 중심의 블랙박스 테스트를 수행하는 것이 테스트 수준이 취약한 개발현장에서 택할 수 있는 매우 적합한 웹 어플리케이션 테스트 방법이라 할 수 있다. 이는 시장 점유율 50%를 차지하는 웹 어플리케이션 테스트 자동화 도구들이 모두 이러한 테스트 방법을 지원함을 볼 때 알 수 있다[3,4-8].

그러나 문제는 테스트 대상 개발물에 대한 면밀한 단위 테스트 과정 없이 바로 사용자 화면 중심의 블랙박스 테스트를 한다는 점에 있다. 블랙박스 테스트는 프로그램 내부 구조에 대한 이해 없이 화면상으로 보여지는 출력만으로 테스트 성공 여부를 판정한다. 이 때문에 실제로 테스트가 수행된 부분과 결함이 발견된 부분을 파악하기 매우 힘들다. 본 논문에서는 이러한 화면 중심의 블랙박스 테스트 문제를 보완하기 위해서 테스트 케이스 선정과 테스트 수행은 사용자 화면 중심의 블랙박스 테스트 기법으로 하되, 수행된 부분과 테스트 결과의 판정은 웹 어플리케이션 구조를 고려한 화이트박스 테스트 커버리지로써 검증하는 방법을 제안한다. 이는 테스트 케이스 선정과 테스트 수행이 비교적 사용자 친화적인 화면 중심의 블랙박스 테스트의 장점과 테스트 종료 시점 정량화 및 결함 위치 추적에 유리한 화이트박스 기법이 가진 장점을 극대화한 테스트 방안이다.

화이트박스 기법으로 테스트 수행 결과를 검증하기 위해서는 반드시 웹 어플리케이션 테스트에 적합한 테스트 커버리지 기준을 적용해야 한다. 본 논문에서는 웹 어플리케이션 계층 사이의 웹 인터페이스와, 테스트 실행 단위가 되는 웹 비즈니스 로직을 정의한다. 웹 인터페이스란, 웹 어플리케이션의 이질적인 계층 간 상호 협력하는 부분으로, 웹 어플리케이션의 테스트 수행 시 반드시 커버해야 되는 위치가 되며, 계층의 통합 시 발생하는 결함 존재 여부를 식별할 수 있게 하는 위치로 활용이 된다[9]. 웹 비즈니스 로직이란, 사용자 요청을 처리하기 위해, 화면에서 시작하여 해당 웹 어플리케이션 구성 컴포넌트가 실행되고 데이터베이스를 경유하여 다시 화면으로 돌아오는 실행 흐름이다. 본 논문은 테스트 대상의 모든 비즈니스 로직과 각 비즈니스 로직에 대한 인터페이스를 모두 테스트 하도록 하는 웹 비즈니스 로직 상의 인터페이스 테스트 커버리지를 제안한다. 효과적인 웹 어플리케이션 테스트를 위해 사용자 화면으로 테스트를 수행하되, 웹 비즈니스 로직 상의 인터페이스 테스트 커버리지로 테스트를 검증하는 방법을 제안하고, 이를 자동화한 테스트 도구인 Testopiacov를 소개하며, 실험을 통해 제안하는 웹 어플리케이션 테스트 방법의 우수성을 분석한다.

본 논문의 구성은 다음과 같다. 2장에서는 웹 어플리케이션 기능 테스트에 대한 기존 연구에 대해 기술하고, 3장에서는 웹 어플리케이션 테스트 방법, 4장은 제안하는 테스트 방안을 웹 어플리케이션 테스트에 적용한 실험 결과를 분석하고, 마지막으로 5장에서는 결론 및 향후 과제를 기술한다.

2. 관련 연구

웹 어플리케이션 테스트를 위한 기존 연구는 사용자 화면 중심의 블랙박스 테스트와 화이트박스로 나눌 수 있다. 화면 중심의 블랙박스 테스트는 사용자와의 모든 상호작용(interaction)이 화면을 기반으로 하는 웹의 특성에 가장 적합한 테스트 방법이며 사용하기 쉽고 직관적인 테스트가 가능하다는 장점이 있다. 반면, 테스트 수행 후 프로그램 내부의 실행 흐름을 알 수 없고 테스트 커버리지 측정이 불가능하므로 화면상으로 보여지는 출력만으로 테스트 결과를 판정해야 한다. 이 때문에 테스트해야 할 모든 영역을 테스트하였는지, 결함이 어디에서 발생하였는지 파악하기 힘들다는 단점이 있다.

웹 어플리케이션 기능 테스트 자동화 도구 중 시장 점유율 50%를 차지하는 상위 다섯 개 제품(제조업체)은 QuickTestPro(HP), Robot(IBM), SilkTest(Borland), TestPartner(Compuware), e-Tester(Empirix)[3,4-8]으로써, 이 도구들은 모두 기능/회귀 테스트 자동화 도구로써, GUI 화면 중심의 블랙박스 테스트 기법을 적용한 레코드/리플레이 방식을 지원한다. 레코드/리플레이 방식이란 테스터가 사용 시나리오에 테스트를 수행하는 순서를 녹화(record)하고, 스크립트로 변환한 뒤, 이를 재생(replay)해서 자동화된 기능 및 회귀 테스트를 수행하는 것을 뜻한다.

기존의 웹 어플리케이션 화이트박스 테스트는 대부분 소스 코드를 기반으로 웹 어플리케이션 테스트 모델

(WATM)을 만들고, 이를 웹 어플리케이션의 분석과 테스트 케이스 생성에 활용한다.

Kung[10]과 Liu 등[11]은 객체 지향 테스트 모델을 웹 어플리케이션에 적용한 웹 어플리케이션 테스트 모델(WATM: Web Application Test Model)을 소개하였다. 이들은 HTML, XML 등의 웹 페이지를 객체로 보고, 객체들 간의 데이터 흐름을 분석하여 테스트 케이스를 생성한다. 웹 어플리케이션을 분석하는 방법으로 Kung은 객체 관점(object perspective), 행위 관점(behavior perspective), 구조 관점(structure perspective)으로, Liu 등은 함수 레벨부터 어플리케이션 레벨까지 총 다섯 단계로 분석을 하였다. 그러나 분석 과정에서 많은 오버헤드를 발생시키며, 이 모델을 토대로 테스트 케이스를 도출하는 것 역시 어려운 문제로 남는다. 또한, 웹 페이지와 페이지 간의 링크만을 분석 대상으로 삼고 테스트를 수행하기 때문에 웹 어플리케이션의 내부 이벤트 처리를 담당하는 로직 흐름은 확인할 수 없다.

Ricca와 Tonella[12]는 UML(Unified Modeling Language)을 이용한 웹 어플리케이션의 분석 모델과 테스트 케이스 생성을 제안하고 이를 지원하는 도구인 ReWeb과 TestWeb을 소개하였다. 모델을 활용해서 페이지 테스트, 하이퍼링크(hyperlink) 테스트, 정의-사용(Def-Use) 테스트, 총 사용(All-Uses) 테스트, 총 경로(All-Paths) 테스트를 수행하는 5가지 방안을 제안하였다. HTML 페이지 내에서의 데이터 흐름(data-flow)만을 고려하므로 서버 측의 제어 흐름(control-flow)과 데이터 흐름을 포함한 모든 흐름을 만족하는 커버리지를 측정할 수 없는 단점은 있으나, 일반적인 화이트 박스 기법을 웹 어플리케이션 구조에 적용한 가장 대표적인 기법이다. 이것은 본 논문에서 제안하는 기법과 가장 비교가 되는 것으로써 실험을 통하여 상세히 비교 분석하고자 한다.

Di Lucca 등[13]은 객체 지향 테스트 모델을 사용해서 웹 어플리케이션의 단위 테스트와 통합 테스트 전략을 제시하고 이를 지원하는 자동화된 도구를 소개하였다. 통합 테스트 전략으로 각각의 유즈 케이스 또는 요구사항에 해당하는 기능을 구현한 페이지들을 함께 테스트 해야 한다고 주장하며, 테스트 케이스 생성에 결정 테이블(decision table)을 활용해서 보다 효율적인 테스트 케이스 생성이 가능하게 하였다. 그러나 결정 테이블을 활용한 테스트 케이스 생성 역시 상당부분 사용자의 개입이 필요하다. 또한 앞서 Ricca와 Tonella에서 지적하였듯이 모델 기반의 테스트의 문제점은 여전히 존재한다.

Elbaum 등[14]은 기존의 화이트박스 웹 어플리케이션 테스트 기술의 제한된 요소는 입력 값을 선택하는 것이 힘들고 자동화할 수 없는 부분이 많기 때문에, 사용자 세션을 이용한 테스트 케이스 선정 방법을 웹 어플리케이션 테스트 모델과 함께 사용하는 방안을 제안하였다. 사용자 세션을 이용한 테스트는 웹 서버 측의 사용자 세션 데이터를 분석하여 테스트 케이스를 생성하고 자동으로 테스트를 수행하므로 테스트에 드는 시간과 노력을 줄일 수 있다. 그러나 웹 어플리케이션의 개발이 완료된 후 실질적인 사용자들의 세션을 이용하는 것이므로 개발 단계에서의 테스트는 불가능하다는 단점이 있다.

이처럼 기존의 화이트박스 기법은 테스트 케이스 선정이 어렵고, 웹 어플리케이션 테스트에 적합한 테스트 커버리지 기준을 제공하지 못하고 있다. 본 논문에서는 이와 같은 어려움을 해결하기 위해 화면 중심의 테스트 수행 및 웹 비즈니스 로직 상의 인터페이스 테스트 커버리지 측정 방안을 제안한다.

3. 웹 어플리케이션 테스트

웹 어플리케이션은 프로그램의 일관성을 유지하고, 성능 및 품질을 향상시키기 위해 웹 어플리케이션 프레임워크(Web Application Framework)를 기반으로 개발하는 것이 일반적 추세이다[15]. 이러한 환경에서 웹 어플리케이션은 다음과 같은 특징을 가진다.

첫째, 웹 어플리케이션은 전통적인 소프트웨어와 달리 프레임워크가 제공하는 기본 골격과 품질이 보증된 코드를 재사용해서 구현한다. 개발자는 소스코드가 제공되지 않는 라이브러리 형태로 프레임워크를 재사용한다.

둘째, 웹 어플리케이션은 다 계층 아키텍처(Multi-layer Architecture)를 가진다. 프레임워크는 웹 어플리케이션을 각각의 역할 별로 계층화 한다. 이 때, 각 계층은 서로 다른 개발자에 의해 구현될 수 있으며, 서버와 클라이언트로 분리되어 지리적으로 떨어진 곳에 위치할 수도 있다. 또한, 서로 다른 구현 기술, 예를 들어 다른 언어를 사용해서 구현되었을 수도 있다. 이렇듯 이질적인 계층들이 서로 연동하며 기능을 수행한다.

셋째, 웹 어플리케이션은 하나의 기능을 처리하는 일련의 실행 흐름을 가진다. 프레임워크가 제공하는 기본 구조와 라이브러리를 재사용해서 웹 어플리케이션을 구현하기 때문에 계층 간 연동에 의해 실행되는 처리 경로는 일정한 패턴을 가진다. 즉, 클라이언트 측의 웹 페이지에서 시작한 사용자 입력은 이질적인 계층 간 연동을 거쳐 서버 측의 데이터베이스를 경유하고 다시 웹 페이지로 돌아오는 일정한 실행 흐름에 따라 동작하고, 이러한 실행 흐름이 하나의 요구사항을 처리하는 실행 단위가 된다.

따라서, 웹 어플리케이션을 효과적으로 테스트하기 위

해서는 이러한 웹 어플리케이션의 특징을 반영하는 테스트 방안이 필요하다. 웹 어플리케이션을 구성하는 이질적인 계층이 상호작용하며 올바르게 원하는 기능을 수행하는지를 반드시 테스트해야 하며, 하나의 기능을 처리하는 실행 흐름이 테스트 실행 단위가 되어야 한다. 이질적인 계층이란, 개발자가 구현한 영역뿐만 아니라 외부 라이브러리까지 포함한다.

본 장에서는 웹 어플리케이션의 구조에 기반해서 웹 어플리케이션을 구성하는 이질적인 계층을 분석하고, 이들 사이의 상호작용하는 위치를 웹 어플리케이션의 인터페이스(이하 웹 인터페이스)로 제안한다. 또한, 테스트 실행 단위인 웹 비즈니스 로직을 정의하고, 각 비즈니스 로직에 대한 웹 인터페이스를 모두 테스트 하도록 하는 "웹 비즈니스 로직 상의 인터페이스 테스트 커버리지"를 제안한다.

3.1 웹 인터페이스

웹 어플리케이션은 일반적으로 그림 1과 같이 개발자가 구현하는 사용자 어플리케이션 영역과 라이브러리 형태의 웹 프레임워크 영역으로 구성된다. 웹 프레임워크 영역은 품질이 어느 정도 보증된 블랙박스로 간주하고 전체를 하나의 라이브러리 계층으로 식별한다. 반면, 사용자 개발 영역은 테스트를 집중해서 수행해야 하는 대상이므로 역할에 따라 계층을 세분화하여 테스트 특성을 분석하고자 한다.

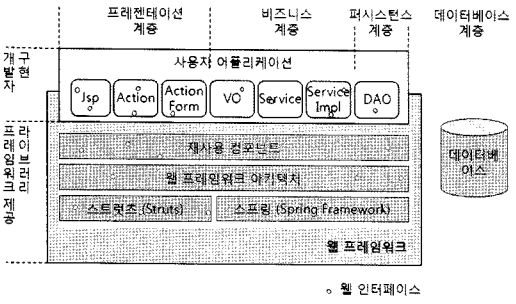

그림 1 웹 어플리케이션 계층 및 인터페이스

일반적으로 웹 어플리케이션은 프레젠테이션 계층(Presentation layer), 비즈니스 계층(Business layer), 퍼시스턴스 계층(Persistence layer)과 같은 이질적인 계층으로 구성된다[16]. 프레젠테이션 계층은 사용자와의 인터랙션을 담당하는 부분으로, 사용자가 실제 접하는 화면에 해당하는 JSP와 사용자의 요청을 비즈니스 계층으로 전달하는 액션(Action), 화면 데이터를 저장하는 객체인 폼(Action Form)으로 구성된다. 비즈니스 계층은 실제 사용자의 요청에 해당하는 서비스를 제공하는 부분이며, 퍼시스턴스 계층은 데이터베이스와의 연동을 담당한다. 비즈니스 계층은 사용자의 요청을 처리하는데 필요한 비즈니스 서비스를 구현한 각종 서비스(Service, ServiceImpl)와 요청을 처리하는데 필요한 데이터를 나타내는 객체인 VO(Value Object)로 이루어진다. 퍼시스턴스 계층은 데이터베이스에 정보를 읽고 쓰는데 필요한 서비스를 구현한 DAO(Data Access Object)가 존재한다.

본 논문에서는 웹 어플리케이션의 구성요소는 아니지만 퍼시스턴스 계층과 직접적으로 연동하는 자원인 데이터베이스를 간접적인 웹 어플리케이션 구성요소로 인정하여 웹 어플리케이션의 계층으로 식별하였다. 결론적으로 웹 어플리케이션의 계층은 테스트 관점에서 볼 때, 총 9개의 계층으로 분류할 수 있다.

> **<정의 1> 웹 어플리케이션의 계층**
> 웹 어플리케이션은 사용자 어플리케이션 영역에 존재하는 JSP, Action, Form, ServiceInterface, ServiceImpl, VO, DAO의 7개 계층, 라이브러리 계층, 데이터베이스 계층으로 구성한다.

일반적으로 인터페이스는 웹 어플리케이션의 계층 사이의 상호작용을 의미한다. 본 논문에서는 웹 인터페이스를 다음과 같이 정의한다.

> **<정의 2> 웹 인터페이스**
> 웹 인터페이스는 웹 어플리케이션의 이질적인 계층 사이에서 서로 상호작용하는 위치이다. 상호작용하는 위치는 표 1의 각 계층 간의 연결 유형에 따라 결정된다.

웹 어플리케이션의 성공적인 개발을 위하여 이를 구성하는 다양한 계층의 통합에서 비롯될 수 있는 오류를 테스트를 하는 것이 필수이다. 웹 인터페이스는 웹 어플리케이션의 이질적인 계층 간 상호 협력하는 부분으로, 웹 어플리케이션의 테스트 수행 시 반드시 커버해야 하는 위치가 되며, 계층의 통합 시 발생하는 결함 존재 여부를 식별할 수 있게 하는 위치로 활용이 된다.

한편 인터페이스를 어떻게 정의하느냐, 커버해야 하는 위치를 문장(statement)으로 할 것인가, 브랜치, 클래스 내에서의 함수 호출 위치로 하느냐, 클래스 간 함수 호출 위치, 혹은 다른 파일에 있는 함수 호출 위치 등 확대 적용할 수 있다. 그러나 본 논문은 웹 어플리케이션 테스트를 하는 현실적인 문제에 대한 솔루션으로써 화면으로 떨어지는 독립적으로 실행 가능한 정도를 최소 대상으로 한다. 이 단계에서는 내부 구조를 확인하는 것은 비교적 큰 단위의 인터페이스 즉 계층간의 인터페이스를 점검, 즉 테스트하는 것이 합리적이다. 계층 내의 인터페이스 확인은 단위테스트에 해당한다.

웹 어플리케이션은 계층 간 역할이 명확하게 구분되고, 다양한 기술을 사용해서 구현하기 때문에 다양한 상호작용 유형을 가진다. 전통적인 소프트웨어가 함수 호출과 리턴, 전역 변수를 통한 데이터 공유 등의 단순 상호작용 유형을 가진 것과 달리, 웹 어플리케이션은 표 1에서처럼 함수 호출을 비롯하여, 계층 간의 연결 유형에 따라 다양한 인터페이스 유형을 가진다.

- 액션매핑(ActionMapping): ActionServlet과 RequestProcessor의 요청처리절차 지칭하며, JSP 화면에서 사용자가 선택한 이벤트를 처리하기 위해 요청 파라메터를 ActionForm에 설정하고, 사용자의 요청(request)에 해당하는 Action을 실행하는 동작을 수행하는 위치이다.
- 자바빈 - 팩토리빈(JavaBean-FactoryBean: getBean): 사용자가 원하는 동작을 실제로 처리하는 비즈니스 계층의 서비스에 접근하기 위해 getBean 함수를 통해 bean id로 서비스 빈클래스를 참조하는 위치이다.
- 자바빈 - 세터 인젝션(JavaBean - Setter Injection: setter method): 너무 많은 수의 파라메터를 가지는 생성자를 만들지 않기 위해 setter 함수를 통해 bean id로 클래스의 객체를 나중에 설정하는 위치이다.
- 데이터베이스 연동: 프레임워크가 제공하는 데이터베이스 연동 라이브러리를 사용하는 위치이다.
- 함수 호출: 다른 클래스에서 제공하는 함수를 호출하는 위치이다.

3.2 웹 비즈니스 로직

웹 어플리케이션은 일반적인 소프트웨어와는 달리 사용자가 어플리케이션을 접하는 환경은 JSP화면에 해당하는 웹 페이지로 국한되며, 개발자가 사전에 사용자의 입력을 제한해 어플리케이션이 정확한 동작을 하는 흐름만을 선택하도록 제약을 두고 있다.

앞서 살펴 본 웹 어플리케이션의 9개 계층 간 연동을 그림 2의 화살표로 나타냈다. 라이브러리 계층은 프레임워크가 제공하는 재사용 영역이므로 개발자 개발 영역과의 차별화를 위해 음영으로 표시하였으며, 라이브러리 계층과의 연동은 프레임워크가 제공하는 간접적인 제어 방식으로 이루어지기 때문에 직접 호출 과의 차이를 두기 위해 대시 화살표로 나타냈다.

표 1 계층 간의 연결 유형에 따른 인터페이스 위치

계층	계층간 연결 유형	웹 인터페이스 위치	
JSP-Action	- JSP와 Action의 연결	- 액션매핑	①
Action-ServiceImpl	- Action과 ServiceImpl의 연결	- 자바빈-팩토리빈	②
	- Form 객체를 VO 객체로 변환	- 함수 호출	③
	- ServiceImpl의 함수 사용	- 함수 호출	④
ServiceImpl-DAO	- ServiceImpl과 DAO의 연결	- 자바빈-세터 인젝션	⑤
	- DAO의 데이터베이스 접근 함수 사용	- 함수 호출	⑥
DAO-Database	- DAO와 프레임워크가 제공하는 라이브러리와의 연결 (PropertiesService, QueryService, HibernateService)	- 자바빈-세터 인젝션	⑦
	- 프레임워크가 제공하는 데이터베이스 연동 라이브러리 사용	- 데이트베이스 연동	⑧

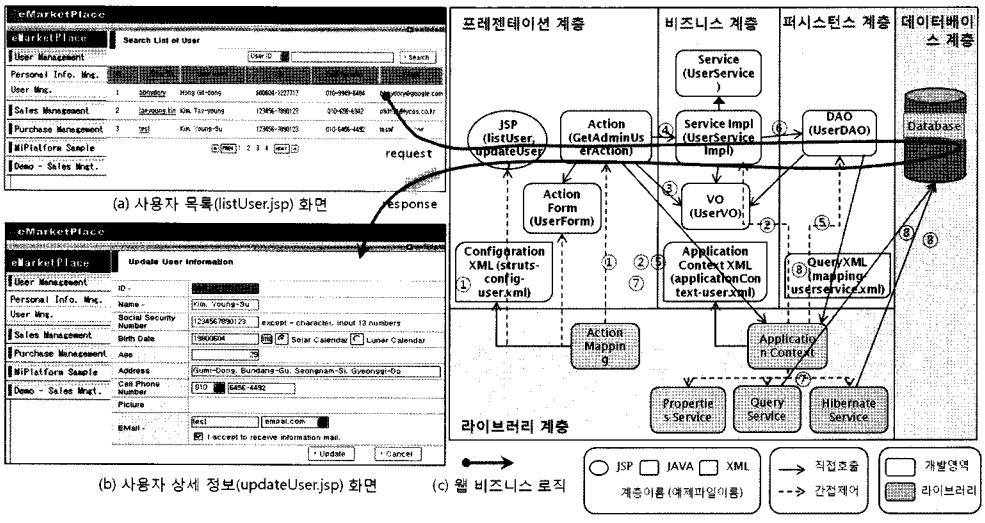

그림 2 웹 어플리케이션의 웹 비즈니스 로직과 예제 화면(괄호숫자는 표 1의 인터페이스를 뜻함)

그림 2에서 JSP 화면으로 들어온 사용자 요청은 프레임워크에서 제공하는 라이브러리 계층과 함께 연동하며, 프레젠테이션 계층과 비즈니스 계층, 퍼시스턴스 계층을 거쳐 데이터베이스를 경유하고 다시 JSP화면으로 돌아오는 일련의 실행 흐름을 갖는다. 이 때, 웹 어플리케이션의 7가지 계층 중, JSP 계층, Action계층, ServiceImpl 계층, DAO 계층, 데이터베이스 계층은 직접적으로 실행 흐름에 관여하며, ActionForm 계층과 VO 계층은 데이터를 저장하는 객체로써 계층 간에 비즈니스 데이터의 전달을 담당하므로 비즈니스 로직에 포함한다. Service 계층은 ServiceImpl 계층이 상속받아 사용하는 클래스로 비즈니스 로직에서 제외한다.

웹 인터페이스 유형 중, 액션매핑은 라이브러리 계층의ActionMapping에서 ConfigurationXML파일을 참조해서 JSP계층과 Action계층을 연결하는 부분에 해당하고, 3가지 종류의 자바빈은 라이브러리 계층의 ApplicationContex에서 ApplicationContextXML 파일을 참조해서 각각 Action계층과 ServiceImpl 계층, ServiceImpl과 DAO 계층, DAO계층과 데이터베이스 계층을 연결하는 부분에 해당한다. 라이브러리 연동은 라이브러리 계층의 QueryService와 HibernateService 서비스를 재사용하는 부분에 해당한다. 이처럼 라이브러리 계층은 웹 비즈니스 로직의 흐름이 원활하도록 지원할 뿐, 직접 실행 흐름에 관여하지 않으므로 비즈니스 로직에서 제외한다.

이처럼 JSP 화면에서 시작하여 웹 어플리케이션의 여러 계층을 거쳐, 웹 인터페이스 유형을 따라 동작하는 실행 흐름을 하나의 비즈니스 로직이라 정의한다.

<정의 3> 웹 비즈니스 로직
JSP 화면에서 시작하여 다시 JSP 화면으로 돌아오기까지 실행되는JSP 계층, Action계층, ServiceImpl 계층, DAO 계층, 데이터베이스 계층을 거치는 실행 흐름이다. 이 때, 이들 계층 간 비즈니스 데이터 전달을 담당하는 ActionForm 계층과 VO 계층을 포함한다.

JSP 화면에서 시작하여 다시 JSP 화면으로 돌아올 때 경우에 따라서는 정의 3에 기술된 모든 계층을 항상 포함할 필요는 없다. 사용자의 요구사항은 항상 웹 비즈니스 로직의 실행 흐름에 따라 수행된다. 즉, 하나의 요구 사항은 하나의 웹 비즈니스 로직과 일대일 대응 관계를 가지며, 이는 최소 실행단위가 된다. 웹 비즈니스 로직의 의미를 예제를 통해 알아본다.

표 2는 5장 테스트 대상 이마켓 프로젝트의, '사용자 정보 조회' 기능에 해당하는 요구사항이다. '사용자 정보 조회' 기능을 수행하는 화면은 그림 2(a)와 같다. 사용자 전체 목록을 보여주는 화면(listUser.jsp)에서 첫 번째 사용자의 UserID 항목(bbnydory)을 선택하면 그림 2(b)와 같이 해당 사용자의 상세 정보가 보여진다.

이 예제에서 화면상으로는 하나의 기능을 처리하는 과정이 JSP 화면 전환으로 보여지는 것이 전부이나, 내부적으로 기능을 처리하는 과정은 하나의 웹 비즈니스 로직의 흐름에 따라 이루어진다. 표 2의 '사용자 정보 조회' 기능에 대응하는 웹 비즈니스 로직에 해당하는 인터페이스는 그림 2의 각 계층에서 괄호 안에 표시된 부분이다.

3.3 웹 인터페이스 커버리지

본 논문에서는 테스트 케이스 선정과 테스트 수행은 사용자 화면 중심의 블랙박스 테스트 기법으로 하되, 테스트 종료의 판정은 웹 어플리케이션 구조를 기반으로 한 화이트박스 테스트 커버리지로써 검증하는 방법을 제안한다. 웹 어플리케이션 테스트에 적합한 화이트박스 테스트 커버리지 기준으로써 테스트 대상의 모든 비즈니스 로직과 각 비즈니스 로직에 포함된 웹 인터페이스를 모두 테스트 하도록 하는 웹 인터페이스 테스트 커버리지를 다음과 같이 정의한다.

<정의 4> 웹 인터페이스 커버리지(Web_Interface Coverage)
$B = \{b_1, b_2, \cdots, b_n\}$: 테스트 대상에 존재하는 모든 비즈니스 로직
$Interface_{bi}$: 비즈니스 로직 b_i에 존재하는 모든 웹 인터페이스
$I_{Total} = \sum_{i=1}^{n} Interface_{bi}$: 테스트 대상 웹 어플리케이션에서 커버해야 할 모든 테스트 위치
$I_{Covered}$: 테스트된 부분

$$Web_{Interface}\,Coverage = \frac{I_{Covered}}{I_{Total}}$$

4. 적용 및 검증

본 논문에서 제안한 방법에 기반하여 개발된 Testopia^{cov}를 실제 웹 어플리케이션에 적용하여, 실제 테스트는 블랙박스 테스트로 실시하되 실시결과에 대한 검토는 화이트 박스 인터페이스 기법으로 테스트 커버리

표 2 요구사항 정의서의 예

요구사항 ID	요구사항 명	처리 내용	화면 명
USERMNG-F002	사용자 정보 조회	전체 사용자 목록에서 한 건 선택하면, 선택한 사용자의 상세정보를 보여준다.	listUser.jsp - updateUser.jsp

지를 측정하는 방법의 효과를 검증하고자 한다.

4.1 Testopia^{cov}

본 논문에서는 웹 어플리케이션 테스트를 위해 사용자 화면으로 테스트를 수행하되, 웹 비즈니스 로직 상의 인터페이스 테스트 커버리지로 테스트를 검증하는 방법을 제안한다. 우리는 제안하는 테스트 방안을 자동화한 테스트 도구인 Testopia^{cov}를 개발하였다. 그림 4(a)는 테스트 대상 웹 어플리케이션 배포 파일로부터 전체 계층 구조와 인터페이스 및 비즈니스 로직을 가시화한 화면이다. 사용자는 그림 4(b)와 같이 웹 어플리케이션을 실행할 때 Testopia웹 브라우저를 띄워서 테스트를 수행한다. 테스트 커버리지와 함께 테스트된 웹 비즈니스 로직 및 웹 인터페이스 영역과 테스트 되지 않은 영역이 분석된다(그림 4(c)).

4.2 적용

테스트 대상 웹 어플리케이션은 온라인 쇼핑몰을 구축한 e-Market Place이다. 이것은 Spring과 Struts, Hibernate를 하나로 합쳐서 오픈 프레임워크로 제공하는 Anyframe JAVA[15]의 공식 홈페이지에 있는 샘플 프로젝트(Anyframe Samples 3.0.0-JDK 1.5-Struts)이다. 이마켓(e-Market Place)샘플 프로젝트(이후 이마켓 프로젝트라 함)는 일반 사용자와 관리자가 사용자 관리, 구매 관리, 판매 관리 등과 같은 메뉴를 이용할 수 있으며, 각각의 메뉴는 등록하기, 목록보기, 세부정보 보기, 수정하기의 세부 메뉴를 가진다. 이마켓 프로젝트는 총 23개의 화면과 95개의 클래스와 12개의 설정파일 문서로 구현되어 있다. Testopia^{cov}로 파악한 웹 비즈니스 로직 및 웹 인터페이스 현황은 표 3과 같다. 이마켓 프로젝트는 총 21개 비즈니스 로직이 존재하며, 21개의 비즈니스 로직으로 커버되어야 하는 인터페이스는 총 239개이다.

4.3 실험 및 분석

본 적용 사례를 통하여 본 논문에서 제안하는 블랙박스 테스트와 화이트박스 테스트를 통한 웹 어플리케이션 테스트 방법의 우수성을 다음과 같은 측면에서 분석하고자 한다.

- 화면을 기준으로 하는 블랙박스 테스트와 화이트박스 테스트의 관계
- 웹 어플리케이션을 구성하는 계층 간 상호작용에 따라 발생할 수 있는 결함
- 웹 인터페이스 테스트 기준의 우수성

4.3.1 화면을 기준으로 하는 블랙박스 테스트와 화이트박스 테스트의 관계

웹 어플리케이션을 테스트 할 때 현장에서 많이 쓰이

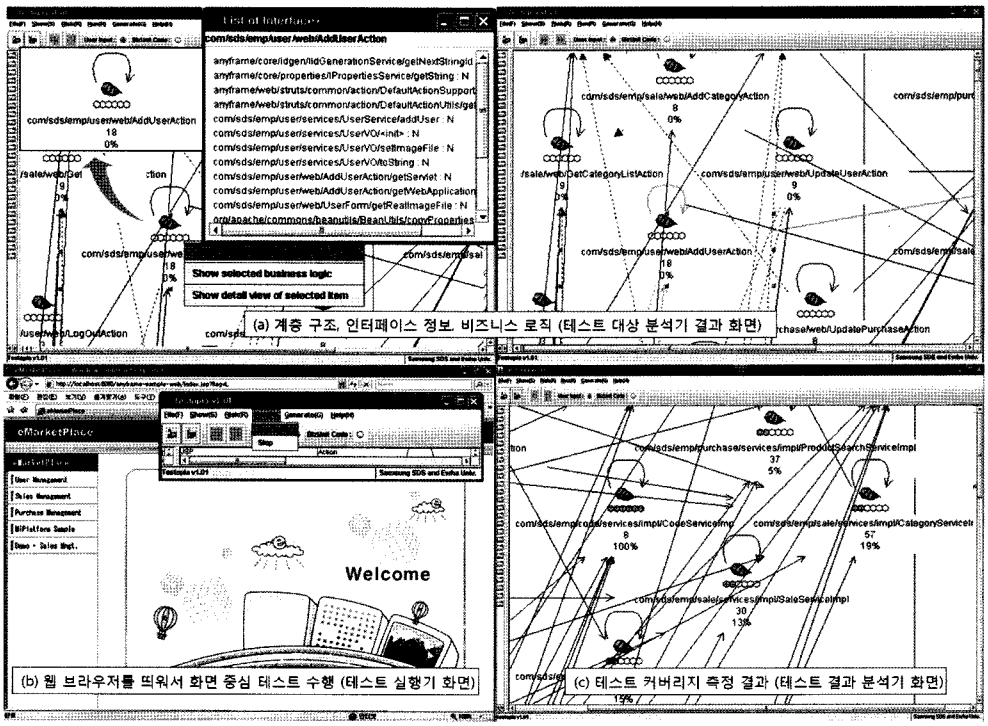

그림 3 Testopiacov 실행 화면

표 3 이마켓 샘플 프로젝트의 웹 비즈니스 로직, 인터페이스 개수

주요 기능	화면 개수	웹 비즈니스 로직 개수	웹 인터페이스 개수								
			JSP-Action	Action-ServiceImpl			ServiceImpl-DAO		DAO-Database		Total
			①	②	③	④	⑤	⑥	⑦	⑧	
공통 영역	10						1(1)	1(1)	1(1)	1(1)	4(4)
사용자 관리	4	9(6)	9(6)	21(13)	14(8)	17(10)	1(1)	5(4)	3(3)	8(6)	78(51)
판매 관리	7	10(10)	10(10)	28(28)	23(23)	20(20)	3(3)	10(10)	5(5)	15(15)	114(114)
구매 관리	2	2(2)	2(2)	11(11)	7(7)	6(6)	3(3)	3(3)	6(6)	5(5)	43(43)
총합	23	21(18)	21(18)	60(52)	44(38)	43(36)	8(8)	19(18)	15(15)	29(27)	239(212)

( ): 경계값-모든 필드 입력데이터에 의해 커버된 인터페이스 수 → 실험(2)참조

는 방법은 사용자 화면을 토대로 블랙박스 기법을 적용하여 테스트케이스를 결정한다. 문제는 블랙박스 테스트만으로 충분한 가에 있다. 이마켓 프로젝트의 총 23개의 화면에 대하여 동등분할(Equivalence Partition)과 경계값분석(Boundary Value Analysis) 블랙박스 테스트 기법을 적용하여 테스트 케이스를 구성하였다. 동등분할의 경우 각 화면의 필수 필드에 대한 테스트 데이터를 적용한 경우(최소필드)와 모든 필드를 적용한 경우(최대필드)로 구분하여 나타내었다. 또한 예외사항 값을 필드에 적용했을 경우를 나타내었다. 이 모든 블랙박스 테스트 방법 각각을 만족하는 테스트 케이스로써 e-Market Place을 실행하였을 때 커버된 인터페이스를 표 4에 나타내었다.

화면 필드 입력에 필요한 테스트 데이터를 랜덤 프로그램을 이용하여 얻은 임의의 값 R1, R2, R3을 테스트 데이터로 사용하였다. 랜덤 프로그램으로 구한 서로 다른 테스트데이터 R1, R2, R3에 대하여 모두 경과 값이 같음을 알 수 있다. 이것은 필드에 입력한 값에 따라 액션이 변하는 것이 아니기 때문이다. 그러나 필드 값에 따라 다른 서비스가 제공되도록 구현된다면, 예를 들어 필드에 입력한 국가별로 다른 서비스가 개발된다면, 커버되는 인터페이스는 달라질 것이다.

동등분할 기법의 경우 각 화면의 필수 필드만 테스트 데이터를 입력한 경우와 모든 필드에 테스트 데이터를 적용하여 테스트한 결과, 모든 필드 적용의 경우가 11개 더 커버되었다. 더 커버된11개의 인터페이스는 필드의 수 증가에 의한 것으로 필드의 수 증가는 액션 클래스, 서비스 클래스, DAO 클래스 인터페이스에 영향을 미친다. 데이터를 입력해 주는 필드의 수가 증가하면 액션 클래스에서 커버되는 인터페이스의 수가 증가하고 액션 클래스와 연결된 서비스 클래스에서도 커버되는 인터페이스의 수가 증가한다. 또한, 서비스 클래스와 연결된 DAO 클래스에서 커버되는 인터페이스의 수도 증가하게 된다. 결국, 필드 입력에 필요한 테스트 데이터의 다양성이 아니라 입력해 주는 필드의 수에 따라 커버되는 인터페이스의 수가 달라지는 것을 알 수 있다.

경계값의 경우 동등분할의 모든 필드 적용과 동일한 결과가 나왔음을 알 수 있다. 이것은 각 필드에 정해진 길이만큼의 값만 필드에 입력할 수 있기 때문이다. 예를 들어 아이디 중복을 확인하는 경우, 아이디 필드에는 데이터가 20자 이하로만 입력되도록 제한한다. 그러므로 경계 값 분석 기법에서 필드의 입력으로는 아이디 20자가 테스트 데이터로 사용된다. 예를 통해서 알 수 있듯이, 경계 값 분석 기법에서는 유효한 데이터 값 중 길이가 가장 길거나 짧은 데이터를 테스트 데이터로 사용한다. 만약 필드에 입력해주는 데이터가 숫자일 경우 유효한 데이터 값 중 가장 큰 수와 작은 수가 테스트 데이터가 된다. 결국 경계 값 분석 기법에서 사용한 테스트 데이터는 데이터의 다양성에 해당되며 위의 설명처럼 인터페이스 수에는 영향을 주지 않는다.

표 4 이마켓 샘플 프로젝트의 블랙박스테스트 vs 화이트박스테스트

Black-box Test / 주요기능 (인터페이스 개수)	동등분할						경계값			예외사항		
	필수필드			모든필드			모든필드			모든필드		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
공통영역(4)	4	4	4	4	4	4	4	4	4	4	4	4
사용자관리(78)	51	51	51	51	51	51	51	51	51	42	42	42
판매관리(114)	103	103	103	114	114	114	114	114	114	52	52	52
구매관리(43)	43	43	43	43	43	43	43	43	43	22	22	22
전체(239)	201	201	201	212	212	212	212	212	212	120	120	120
인터페이스 커버리지(평균)	84%			89%			89%			50%		

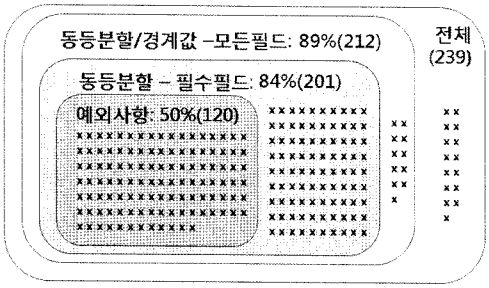

그림 4 커버된 인터페이스 상황

예외사항의 경우, 표 4에서 50%의 인터페이스를 커버하였는데, 커버된 인터페이스를 살펴보면 그림 5에서 나타내었듯이 예외사항에 대하여 커버된 인터페이스는 필수필드에 동등분할값을 적용한 경우와 모든 필드에 동등 및 경계값을 적용한 경우에 포함되었다. 이는 예외사항으로만 별도로 커버되는 인터페이스는 본 실험에서는 없었다. 그 이유는 예외사항에 대한 서비스 클래스를 따로 구현되어 있지 않기 때문이다. 따라서 만일 예외사항 처리 클래스를 별도의 서비스로 구현한다면 정상 테스트 시나리오 이외의 예외사항에 관한 테스트에 의해 커버해야 할 인터페이스가 증가하게 된다.

결론적으로 본 웹 인터페이스 커버리지는 단위 테스트보다는 웹 비즈니스 로직을 커버하는 통합 테스트에 적합함을 알 수 있다. 즉 실행할 수 있는 정도의 규모의 웹 프로그램을 화면 중심의 블랙 박스로 테스트 하되 여기서 끝내지 말고, 실행된 비즈니스 로직의 실행 경로와 이질적 계층 사이의 웹 인터페이스를 점검하게 되면, 이때 발생할 수 있는 결함을 파악할 수 있도록 도와 줄 수 있다는 점이다. 다음은 웹 어플리케이션에서의 결함과 웹 인터페이스와의 관계에 대하여 분석하고자 한다.

4.3.2 웹 어플리케이션을 구성하는 계층 간 상호작용에 따라 발생할 수 있는 결함

웹 인터페이스 유형에 따른 결함 종류[17]를 기반으로 웹 어플리케이션을 구성하는 계층 간 상호작용에 따라 발생할 수 있는 결함을 분류하면 표 5와 같다. 웹 비즈니스 로직은 JSP 화면에서 시작하여 웹 어플리케이션의 여러 계층을 거쳐, 웹 인터페이스 유형에 따라 동작하는 실행 흐름이며, 웹 비즈니스 로직에 따른 모든 웹 인터페이스를 커버하도록 웹 인터페이스 커버리지로 테스트한다는 것은 표 5에 분류된 결함을 발견할 수 있도록 유도함을 뜻한다.

실제 이마켓 프로젝트의 경우에서, "사용자 등록" 기능에서 다음과 같은 결함을 발견하였다. 사용자 등록을 수행하면, 그림 6(a)와 같은 화면 흐름에 따라 동작하고, 사용자 등록 완료 후 그림 6(b)의 로그인 화면으로 전환한다. 화면상으로는 이러한 흐름이 정상적으로 보이기 때문에 화면 테스트만으로는 사용자 등록 기능에 결함이 있음을 인지할 수 없으나, 실제 사용자 등록 여부를 확인해보면, 그림 6(c)와 같이 해당 사용자가 존재하지 않는다. 즉, 결함이 있음에도 화면에 의한 블랙박스 테스트로는 알 수 없는 경우다.

표 3에서도 알 수 있듯이 사용자 관리에 해당하는 웹 인터페이스를 확인해보면, 사용자 관리에 해당하는 웹 인터페이스의 수(( ) 안의 수)는 51개이며, 웹 비즈니스 로직 12개 중, 3개의 비즈니스 로직이 수행되지 않았으며, 이것은 ①번 유형의 인터페이스부터 커버가 되지 않았음을 알 수 있다. 수행되지 않은 비즈니스 로직의 ①번 웹 인터페이스 유형에서 그림 7과 같은 결함을 발

표 5 웹 어플리케이션 결함

위치	결함 종류	[17]의 분류 기준
①	**F1 이벤트처리연결 결함** 이벤트 처리를 담당하는 액션클래스가 잘못 연결됨	**Link faults** 링크 등을 통해 연결된 이벤트 처리 코드 내 오류
②	**F2 서비스연결 결함** 액션 클래스가 사용하는 서비스 클래스가 잘못 연결됨	**Logic faults** 비즈니스 계층의 서비스 또는 서비스 수행 흐름을 구현한 코드 내 오류
③	**F3 폼객체전환 결함** 폼 객체와 VO객체의 불일치	**Form faults** 폼에 사용자가 입력한 데이터 변환 코드 내 오류
④	**F4 서비스사용 결함** 서비스 클래스가 제공하는 서비스의 잘못된 사용	**Logic faults** 비즈니스 계층의 서비스 또는 서비스 수행 흐름을 구현한 코드 내 오류
⑤	**F5데이터베이스객체 연결 결함** DAO 클래스의 세터함수 오류	**Data store faults** 데이터베이스와 연동하는 코드 내 오류
⑥	**F6 데이터베이스객체 사용 결함** DAO 클래스가 제공하는 서비스의 잘못된 사용	
⑦	**F7 데이터베이스연결 결함** 데이터베이스 연동 라이브러리의 세터함수 오류	

그림 5 사용자 등록 기능 수행 화면

결함 수정 전 ConfigurationXML

```
<action input="/empadduserview.do"
path="/empadduser"
type="com.sds.emp.user.web.AddUserViewAction"
<forward name="success" path="/emploginview.do"
```

결함 수정 후 ConfigurationXML

```
<action input="/empadduserview.do"
path="/empadduser"
type="com.sds.emp.user.web.AddUserAction"
<forward name="success" path="/emploginview.do" />
```

그림 6 사용자 등록 기능에서 발견한 결함

견했다. 이것은 표 3의 이벤트처리연결 결함으로, JSP화면과 Action 클래스가 액션매핑 라이브러리를 통해 서로 연결될 때, 연결정보에 해당하는 ConfigurationXML 파일이 잘못되어서 발생하는 결함이다. (3.2절의 그림 2를 참조). 결함 수정 전, ConfigurationXML파일은 사용자 등록 이벤트인 "/empadduser"를 처리하는 클래스를 AddUserViewAction 클래스와 연결한다. 그러나 이는 잘못된 연결로써 올바르게 기능을 수행하기 위해서는 AddUserAction 클래스와 연결해야 한다.

4.3.3 Ricca& Tonella 웹 테스트와의 비교

실험(2)에서 보았듯이 웹 상호 계층간의 결함과 웹 인터페이스와의 상당한 관련이 있음을 알 수 있다. 그렇다면, 화면 중심의 블랙박스 테스트를 하되, 웹 인터페이스 커버리지로 점검하지 말고, 웹에 적용할 만한 다른 화이트 박스 테스트 커버리지로 하면 안 되는가 하는 점이다. 관련연구에 기술하였듯이 자바 기반의 웹 어플리케이션 테스트로 Ricca& Tonella[12]가 정리한 Page testing, Hyperlink, Definition-use, All-use, All-path를 적용하여 이마켓 프로젝트 테스트를 수행하였다.

- Page testing: 사이트의 모든 페이지를 적어도 한번은 방문
- Hyperlink testing: 모든 페이지의 모든 하이퍼링크를 적어도 한번은 실행
- Definition-use testing: 변수가 정의되고 그 변수가 사용되는 모든 탐색패스를 지나감
- All-uses testing: 변수가 정의되고 그 변수가 사용되는 탐색패스를 적어도 한번은 지나감
- All-paths testing: 사이트의 모든 패스를 적어도 한번은 지나감(루프는 한번으로 제한함)

표 6은 Page testing, Hyperlink, Definition-use, All-use, All-path로 실행했을 때 커버되는 웹 인터페이스를 보여준다. 특히 All-paths의 경우 90%의 인터페이스만을 커버했다는 의미는 표 4의 총 239개의 인터페이스 가운데 215개를 제외한 나머지는 테스트하지 못한 채 종료되었다는 것이다. 그 이유는 Ricca&Tonella 방법은 화면 간의 실행 경로만이 분석 대상이므로 아래와 같은 경우의 실행 경로를 테스트 케이스로 선정하지 못한다는 점에 있다.

예를 들어, 그림 8은 판매 관리 메뉴의 "판매 상세 정보 보기"기능에 해당하는 화면이다. 판매 상세 정보 화면은 정보를 보여주는 대상의 Transaction 상태에 따라, 그림 8(a), (b)와 같이 Update 버튼을 갖거나 갖지 않는다. 이와 같은 정보는 화면 상의 이벤트 처리와 관계된 부분으로 전체 화면의 실행 경로만을 파악해서는 알 수 없으며, 이벤트를 처리하는 내부 흐름이 드러나야 한다.

Ricca와 Tonella의 테스트 방법도 웹 인터페이스 테스트와 마찬가지로 웹 어플리케이션의 구조를 기반으로 하는 화이트 박스 기법이다. 다만, Ricca와 Tonella는 전체 화면 실행 경로만이 분석 대상이므로, 웹 화면 이외의 웹 어플리케이션 수행 흐름을 구성하는 부분을 제대로 테스트 하지 못한다.

표 6 다른 웹 화이트박스 테스트 기법과 커버된 인터페이스 관계

테스팅 종류	Page	Hyperlink	Definition-use	All-uses	All-paths
커버된 인터페이스의 개수	194	215	113	113	215
퍼센트	81%	90%	47%	47%	90%

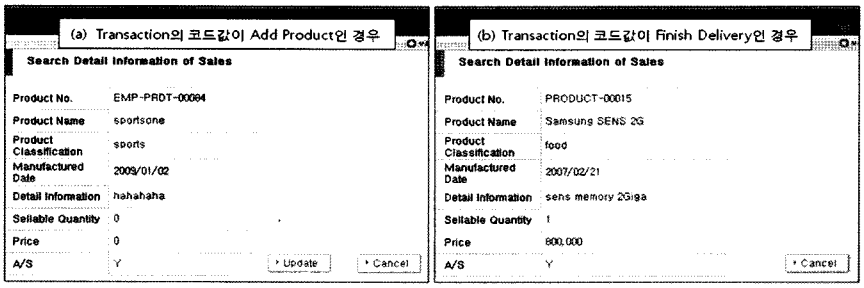

그림 7 Transaction의 코드값에 따라 다른 형태를 갖는 "판매 상세 정보 보기" 화면

5. 결론 및 향후 연구

웹 어플리케이션의 규모가 커지고 복잡해지면서 품질의 중요성은 더욱 높아지고 있다. 신뢰도 높은 웹 어플리케이션을 개발하기 위해 웹 프레임워크(Web Application Framework)를 기반으로 개발하는 것이 일반적 추세이다. 이러한 환경에서는 웹 어플리케이션을 구성하는 이질적인 계층 간 상호작용하는 위치인 웹 인터페이스가 바로 웹 어플리케이션 테스트에서 반드시 커버해야 할 중요한 테스트 위치이며, 이 때 테스트 실행은 JSP 화면에서 시작하여 Action계층, ServiceImpl 계층, DAO 계층, 데이터베이스 계층을 거쳐 다시 JSP 화면으로 돌아오는 수행 흐름인 웹 비즈니스 로직을 하나의 단위임을 제안하였다.

그러나 테스트 수준이 취약한 개발현장에서는 테스트 케이스 입력의 편의를 위해 웹 어플리케이션의 내부 구조에 대한 고려 없이 바로 사용자 화면 중심의 블랙박스 테스트를 수행한다. 이는 웹 어플리케이션의 전체 실행 흐름을 모두 수행했는지 확인할 수 없기에 테스트를 수행하지 못하는 영역이 존재할 위험이 있다. 또한, 기존의 웹 어플리케이션 화이트박스 테스트 연구는 테스트 케이스 선정과 입력 방식이 어려워 실무에서 잘 사용되지 않으며, 웹 비즈니스 로직의 흐름을 고려하지 않아서 실제 웹 어플리케이션의 기능을 처리하는 내부 컴포넌트의 흐름에 결함이 발생한 경우 이를 발견하지 못한다.

이에 본 논문은 효과적인 웹 어플리케이션 테스트 방안으로 사용자 화면으로 테스트를 수행하되, 웹 비즈니스 로직 상의 인터페이스 테스트 커버리지로 테스트를 검증하는 방법을 제안하였고, 이를 자동화한 테스트 도구인 Testopia^{cov}를 개발하였다. 또한, Testopia^{cov}를 기존의 웹 어플리케이션 블랙박스 및 화이트박스 테스트 연구와 비교 분석하여 웹 어플리케이션 구조를 기반으로 보다 면밀하게 테스트를 수행함을 보였다. 향후 Testopia^{cov}를 산업에서의 프로젝트 개발에 실제 적용할 계획이며, 이를 통해 Testopia^{cov}가 웹 어플리케이션 테스트에의 기여를 분석할 예정이다.

참 고 문 헌

[1] Chien-Hung Liu et al., "Object-Based Data Flow Testing of Web Applications," *Proceedings. First Asia-Pacific Conference* on, 1(1), pp.7-16, 2000.

[2] Filippo Ricca, Paolo Tonella, "Testing Processes of Web Applications," *Annals of Software Engineering*, 14(1-4), pp.93-114, 2002.

[3] Forrester Research, "Forrester Wave™, : Functional Testing Solutions," Tech Choices The Forrester Wave™: Functional Testing Solutions Q2 2006, 2006.

[4] HP, "QuickTestPro," https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24%5E1352_4000_100_

[5] IBM, "Robot," http://www-01.ibm.com/software/awdtools/tester/robot/index.html

[6] Borland, "SilkTest," http://www.borland.com

[7] Compuware, "TestPartner," http://www.compuware.com

[8] Empirix, "e-Tester," http://www.empirix.com

[9] Ahyoung Sung, Byoungju Choi, Seokkoo Shin, "An Interface Test Model for Hardware-dependent Software and Embedded OS API of the Embedded System," the Computer Standard & Interface journal, 29(4), pp.430-443, 2007.

[10] David Chenho Kung, "An Object-Oriented Web Test Model for Testing Web Applications," *24 th International Computer Software and Applications Conference*, 1(1), 537-542, 2000.

[11] Chien-Hung Liu, "Structural testing of Web applications," *Proceeding of 11th International Symposium on Software Reliability Engineering*, pp.84-96, 2000.

[12] Filippo Ricca, Paolo Tonella, "Analysis and Testing of Web Applications," *Proceedings of the 23rd International Conference on Software Engineering*, pp.25-34, 2001.

[13] GA Di Lucca, A Fasolino, F Faralli, "Testing Web Applications," *Proceeding of the International Conference on Software Maintenance*, pp.310-319,

2002.

[14] Sebastian Elbaum et al, "Improving web application testing with user session data," *Proceedings of International Conference on Software Engineering*, pp.49-59, 2003.

[15] Samsung SDS, "Anyframe JAVA," http://anyframejava.org, 2008.

[16] Ezra Ebner, Weiguang Shao and Wei-Tek Tsai, "The five-module framework for Internet application development," *ACM Computing Surveys (CSUR)*, 32(1es), 2000.

[17] S.Sampath, S.Sprenkle, E.Gibson, L.Pollock, and A.S.Greenwald, "Applying Concept Analysis to User Session-based Testing of Web Application," *IEEE Trans. On Software Engineering*, vol.33, pp. 643-657, 2007.

임 정 희

2005년~2009년 삼육대 컴퓨터과학과 학사. 2009년~현재 이화여대 컴퓨터공학과 석사과정. 관심분야는 소프트웨어공학, 소프트웨어 테스트, 웹 어플리케이션 테스트

이 시 현

2005년~2009년 이화여대 컴퓨터공학과 학사. 2009년~현재 이화여대 컴퓨터공학과 석사과정. 관심분야는 소프트웨어공학, 소프트웨어 테스트, 웹 어플리케이션 테스트

장 진 아

2002년~2006년 이화여대 컴퓨터공학과 학사. 2006년~2009년 이화여대 컴퓨터공학과 석사. 2009년~현재 SK텔레콤 정보기술원. 관심분야는 소프트웨어 공학, 소프트웨어 테스트

최 병 주

1979년~1983년 이화여대 수학과 학사 1986년~1987년 Purdue Univ. Computer Science 석사. 1987년~1990년 Purdue Univ. Computer Science 박사. 1991년~1992년 삼성종합기술원 1992년~1995년 용인대 전산통계학과 조교수 1995년~현재 이화여대 컴퓨터학과 교수. 관심분야는 소프트웨어공학, 소프트웨어 테스트, 소프트웨어 및 데이터 품질 측정

황 상 철

1992년~1999년 경희대 산업공학과 학사. 1999년~2001년 경희대 산업공학과 석사. 2001년~현재 삼성 SDS SW Eng 팀 근무. 관심분야는 J2EE 개발 방법론 및 아키텍처, 소프트웨어 테스트 자동화