J2EE 애플리케이션의 흐름분석을 통한 커버리지 기반 테스트 방법

이정규*, 국승학**, 김현수***
*충남대학교 전기정보통신공학부 컴퓨터전공
jjangqwer@gmail.com, {triple888, hskim401}@cnu.ac.kr

A coverage driven test method for J2EE Applications through flow analysis

Junggyw Lee*, Seunghak Kuk**, Hyeonsoo Kim***
*Dept, of Computer Sc. & Eng., Chungnam National University

요 약

최근 엔터프라이즈 애플리케이션은 J2EE 컴포넌트로 개발된다. J2EE 플랫폼은 애플리케이션을 개발하는데 편리한 기능을 제공한다. 하지만 J2EE 애플리케이션의 테스트는 J2EE 서버 내부에서 자원을 관리하는 컨테이너의 접근 통제 때문에 기존의 테스트 방법으로 수행하기 어렵다. 본 논문에서는 J2EE 환경에 맞는 J2EE 애플리케이션의 테스트 방법을 제안한다. 이 방법은 애플리케이션을 구성하는 EJB 컴포넌트를 정적 및 동적 분석하여 획득한 정보로 애플리케이션의 메소드 커버리지를 분석하고, 메소드 커버리지를 향상시키기 위한 테스트 데이터를 생성하여 J2EE 애플리케이션을 테스트한다.

1. 서 론

최근 엔터프라이즈 애플리케이션의 구축은 애플리케이션의 구성 요소들을 여러 부분으로 나누어 개발된다. 그리고 엔터프라이즈 환경에서는 이해관계자들의 요구 사항과 관련된 기술이 빠르게 변화한다. 엔터프라이즈 애플리케이션 개발은 이러한 요구들을 충족하면서도 보다 적은 비용으로 설계하고 개발할 수 있어야 한다. 또한 애플리케이션의 서비스 속도 향상보다 기존 어플리케이션이 보다 더 자원을 적게 소비하고 안정성이 보장되어야 한다.

Sun Microsystems사가 발표한 J2EE(java 2 Enterprise Edition)은 엔터프라이즈 애플리케이션을 구성하는 컴포넌트를 개발을 지원하는 작업 환경이자 개발 도구이다. J2EE 엔터프라이즈 애플리케이션은 비즈니스 솔루션을 빠르게 개발할 수 있으며, J2EE 컴포넌트를 특정 애플리케이션 서버나 운영 체제에 종속적이지 않도록 구축할 수도 있다. 이로 인해서 개발자는 비즈니스에 관한 요구 사항에 맞는 기술을 충족시키기 위해 프로그래밍에 필요한 컴포넌트를 보다 자유롭게 선택할 수 있게 되었다. 현재 국내 SI 업체에서 개발하는 대규모 소프트웨어의 90%이상이 J2EE 환경에서 이루어진다[1][2][3].

하지만 J2EE 애플리케이션의 테스트는 아래와 같은 이유에서 어렵다.

J2EE 애플리케이션은 컴포넌트 기반 소프트웨어 개발(Component-Based Software Development : CBSD)로 만들어진다. CBSD는 이미 개발된 컴포넌트들을 재사용함으로써, 소프트웨어 개발 비용을 절감시킨다. 그러나 CBSD는 새로 개발된 컴포넌트의 테스트뿐만 아니라 기존 컴포넌트와의 조립에 대한 테스트도 이루어져야 한다[4]. 이러한 멀티 티어 테스트는 컴포넌트의 각 특성 때문에 예상하지 못한 오류가 발생할 수 있다.

J2EE 애플리케이션의 테스트는 J2EE 서버 내부에서 자원을 관리하는 컨테이너의 접근 통제 때문에 기존 테스팅 기법은 J2EE 환경에서 적합하지 않다[5][6][7].

이에 본 논문에서는 J2EE 애플리케이션의 테스트 방법을 제안한다. 이 방법은 애플리케이션을 구성하는 EJB 컴포넌트를 대상으로 수행된다. EJB 컴포넌트들을 정적 및 동적 분석하여 얻은 정보로 메소드 커버리지를 분석한다. 그리고 메소드 커버리지를 향상시키기 위한 테스트 데이터를 생성한다. 이렇게 생성된 테스트 데이터로 J2EE 애플리케이션을 테스트하여 보다 높은 메소드 커버리지를 분석한다.

본 논문의 2장에서는 J2EE 애플리케이션의 테스트 방법을 기술하고 3장에서는 2장에서 소개한 테스트 방법을 이용하여 EJB 컴포넌트의 메소드 커버리지 분석 방법을 기술한다. 그리고 4장에서는 결론과 향후 연구에 대해 기술한다.

2. 관련 연구

EJB 컴포넌트 테스트 전략은 EJB가 컨테이너 내부에 동작한다는 특수한 환경을 고려한 것으로 테스트 환경과 관련된다. 전략은 크게 세 가지로 나누어 볼 수 있다[5][6][7].

2.1 Testing in Isolation

컨테이너나 EJB의 스텁을 생성하여 독립적인 환경에서 테스트를 수행한다. EJB는 다른 EJB 나 데이터베이스와 같은 자원에 의존적이기 때문에 완벽한 단위 테스트는 어렵다. Mock Objet는 이러한 어려움을 극복한다. MockEJB 프레임워크는 컨테이너나 다른 EJB들의 스텁을 손쉽게 만들어 주는 기능으로 독립적인 환경에서 EJB의 단위 테스트를 쉽게 수행한다. 하지만 MockEJB 플레임워크는 컨테이너나 EJB, DB와 같은 자원에 대한 많은 Mock Object를 생성해 주어야 하며, 독립적인 환경에서 성공적으로 테스트를 수행하여도 실제 컨테이너에서 동작한다고 확신할 수 없다.

2.2 In-Container Testing

컨테이너나 애플리케이션 서버 내부의 테스트 방법은 애플리케이션 서버 내에서 실행하는 테스트를 작성하고 배

치한다. 이는 로컬 인터페이스를 소유한 EJB 테스트에 적합하며, Cactus와 JUnitEE에서 이용한다. In-Container Testing은 컨테이너 내에서 테스트를 수행할 수 있도록 하는 테스트 프레임워크가 요구되며, 구현, 배치 그리고 테스트를 위한 호출이 더 복잡해진다.

2.3 Out Side the Container Testing

빈이 원격 인터페이스를 가질 때 원격 클라이언트를 생성하여 테스트한다. 이 방법은 테스트 작성과 실행이 용의하고 JUnit처럼 테스트 인프라를 쉽게 사용한다. 이 방법은 원격 인터페이스의 기능 테스트와 분산 환경에서 원격 인터페이스를 통한 비즈니스 로직을 테스트 하는데 적합하다. 하지만 로컬 인터페이스만을 갖는 컴포넌트의 경우 테스트가 불가능하다.

본 논문에서는 In-Container Testing 방법을 이용한다. 애플리케이션의 소스 코드 레벨에서 애플리케이션의 동적 정보를 추출할 수 있는 탐침코드를 삽입한 후 J2EE 애플리케이션에 대한 테스트를 수행한다.

3. 애플리케이션의 메소드 커버리지 분석 방법

이 장은 애플리케이션의 메소드 커버리지 분석 방법에 대해 설명한다. 이 방법은 첫 번째로 애플리케이션을 정적 분석한 후 그 정보로 정적 메소드 경로 그래프를 생성한다. 두 번째로 정적 분석 정보를 이용하여 애플리케이션의 소스 코드에 탐침 코드를 삽입한다. 이 탐침 코드는 애플리케이션의 동적 정보를 추출하는데 사용되며, 이 동적 정보의 분석으로 동적 메소드 경로 그래프를 생성한다. 세 번째로 이렇게 생성된 정적/동적 메소드 경로 그래프의 비교분석으로 메소드 커버리지를 측정한다. 마지막으로 메소드 커버리지를 높이기 위한 테스트 데이터를 생성한다.

3.1 애플리케이션 정적 분석

J2EE 애플리케이션을 구성하는 EJB 컴포넌트들을 분석한다. EJB 컴포넌트 분석은 EJB 컴포넌트의 소스 파일을 변환한 XML 문서와 XML로 기술된 배치문서를 분석하여 이루어진다. EJB 컴포넌트들의 분석은 아래와 같은 차례로 분석된다.

①**클래스 분석:** EJB 컴포넌트를 구성하는 클래스를 대상으로 클래스의 이름과 클래스의 종류(일반 자바/빈)를 분석한다.

②**인터페이스 분석:** 빈 인터페이스의 경우, 인터페이스의 이름과 종류를 분석한다. 그리고 인터페이스가 소유한 메소드를 대상으로 메소드의 기본속성(접근 변경자/자료형/이름)과 매개변수의 기본속성(자료형/이름)을 분석한다. 인터페이스는 EJB 컴포넌트의 구성요소로 EJB 컴포넌트가 외부와 상호작용을 한다. 즉 인터페이스는 외부로부터 처음으로 호출되는 메소드로 메소드 흐름 분석에 중요한 정보를 제공한다.

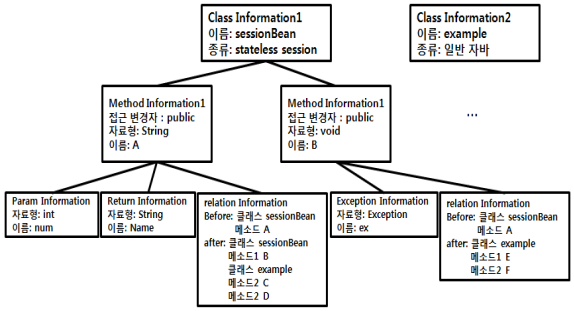
③**참조 클래스 분석:** 참조 클래스는 분석 대상 클래스에서 참조하는 클래스를 의미하며. 테스트 대상 EJB 컴포넌트에 속한다. 참조 클래스의 클래스 이름과 인스턴스 변수 이름을 분석하다. 이 인스턴스 변수로 분석 클래스에서 호출하는 메소드의 이름과 매개변수의 기본속성을 분석한다. 이러한 분석은 EJB 컴포넌트 내부에서 호출되는 메소드를 분석하는데 사용된다.

④**내부 메소드 분석:** 내부 메소드는 분석 대상 클래스에 의해 정의되고 호출되는 메소드를 의미한다. 분석중인 클래스의 내부 메소드 중 자신의 메소드에 의해 호출되는 메소드의 기본속성과 매개변수의 기본속성 정보를 분석한다.

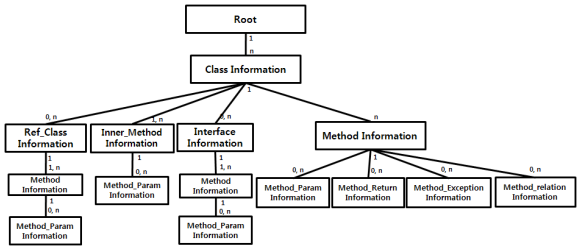
⑤**메소드 분석:** 단계 ③, ④에서의 메소드 분석은 메소드를 호출한 자바 구문를 이용하여 분석된다. 이 구문은 메소드를 정의하는 구문보다 명확한 정보를 분석할 수 없다. 따라서 단계 ③, ④에서 분석한 정보를 이용하여 메소드가 정의된 자바 구문 찾아 메소드의 정보를 보다 명확하게 분석한다. 또한 분석 대상 메소드를 호출하는 메소드도 분석 대상이 된다.

⑥**메소드 호출 관계 분석:** 메소드의 호출 관계는 분석 대상 메소드를 기준으로 분석 대상 메소드를 호출하는 메소드와 분석 대상 메소드가 호출하는 메소드가 분석된다. 그림 1은 메소드 분석 정보의 사례이다.

이 절에서 분석한 분석 정보는 그림 2와 같은 구조체에 기록된다.



(그림 1) 메소드 분석 정보



(그림 2) EJB 컴포넌트 분석 정보

3.2 정적 메소드 경로 분석

3.2.1 정적 메소드 경로 그래프

정적 메소드 경로 그래프는 애플리케이션에서 생성될 수 있는 메소드의 모든 흐름을 나타내는 그래프이다.

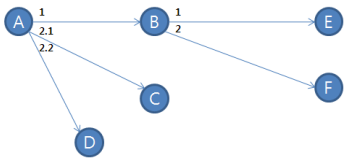
3.2.2 정적 메소드 경로 그래프 생성 방법

분석 정보의 메소드 관계 정보는 자신을 호출한 메소드와 자신이 호출한 메소드를 기술한다. 이는 메소드들의 호출되는 시퀀스 정보로 볼 수 있다. 이러한 시퀀스 정보로 메소드 경로를 추적한다.

그림 1의 메소드 분석 정보는 메소드 A, B, C, D, E, F에 대한 메소드의 관계 정보를 가진다. 만약 메소드 A가 EJB 컴포넌트의 인터페이스를 구현한다면, 우리는 인터페이스 분석 정보에 의해 메소드 A는 EJB 컴포넌트의 인터페이스를 구현한 메소드인 것을 알 수 있다. 즉 EJB 컴포넌트에서 메소드 A는 외부와 상호작용하는 메소드이다. 따라서 메소드 A는 EJB 컴포넌트에서 생성되는 메소드 경로 중 가장 첫 단에 위치하게 된다. 우리는 메소드 A와 같은 메소드의 관계 정보를 추적하여 정적 메소드 경로

그래프를 생성할 수 있다. 메소드 A는 메소드 B, C, D를 호출한다. 여기서 메소드 A는 메소드 B를 먼저 호출하고, 메소드 C, D를 특정 조건에 따라 호출한다. 그리고 메소드 B는 메소드 E, F를 호출한다. 메소드 E가 먼저 호출되고 메소드 F가 다음에 호출된다. 메소드 C, D, E, F는 호출하는 메소드가 존재하지 않는다. 그림 3은 정적 메소드 경로 그래프의 사례이며, 정적 메소드 경로 그래프는 다음과 같은 특징을 가진다.

- 노드는 메소드를 나타낸다.
- 노드는 방향성 있고 번호가 있는 에지를 갖는다.
- 에지는 숫자는 메소드의 호출하는 순서이다.
- 에지의 번호가 소수점을 가지면 현재 메소드가 조건문에 따라 다른 매소드를 호출 할 수 있다는 의미이다.
- 그림 3에서 메소드 A는 3개의 메소드를 호출한다. 메소드 A는 메소드 B를 호출하고 메소드 C, D 중 조건에 맞는 메소드 하나를 선택하여 호출한다.



(그림 3) 정적 메소드 경로 그래프

3.3 동적 메소드 경로 분석

정적 분석 정보로 메소드 A가 메소드 C를 호출하는 것을 알 수 있지만 애플리케이션이 작동할 때 메소드 A가 메소드 C를 호출한다고 확신할 수 없다. 따라서 애플리케이션의 동적 정보를 획득하고 이 정보를 이용하여 메소드의 동적 흐름을 분석하는 방법을 기술한다.

3.3.1 동적 메소드 경로 그래프

동적 메소드 경로 그래프는 애플리케이션이 동작했을 때 생성되는 메소드의 동적 흐름을 보여주는 그래프이다.

3.3.2 동적 메소드 경로 그래프 생성 방법

①**테스트 코드 삽입:** 애플리케이션의 분석 정보를 이용하여 테스트 대상 클래스에 그림 4와 같이 탐침 코드를 삽입한다. 탐침 코드는 애플리케이션의 동작 시 애플리케이션의 메소드 정보를 추출한다. 이 정보는 메소드의 기본 속성, 메소드의 매개변수, 메소드의 리턴 등이다. 그리고 탐침 코드는 메소드의 시퀀스 정보를 식별하는 특정 식별 값을 저장한다.

```
...
public String A(int num){
...
Example ex = new Example();
ex.C(num);
...
}
```

```
public Class Example{
...
public int C(int num){
...
}
}
```

테스트 코드 삽입 전

```
Import Monitor
...
Monitor test = new Monitor()
...
public String A(int num){
String sequence =
        test.getSequence();
test .extractMethod();
test .extractSequence();
test .extractParam();
Example ex = new Example();
ex.C(num, sequence);
...
}
```

```
Import Monitor
public Class Example{
Monitor test = new Monitor()
...
public int C(int num, String sequence)
test .extractMethod();
test .extractSequence();
test .extractParam();
...
test .extractReturn();
return num;
}
}
```

테스트 코드 삽입 후

(그림 4) 테스트 코드 삽입

탐침 코드의 삽입 위치는 다음과 같다.

애플리케이션의 런 타임 시 호출한 메소드와 호출된 메소드의 시퀀스 관계를 식별해 주는 sequence 식별 값을 메소드 A 내부에서 생성하는 구문과 그 정보를 메소드 B의 매개변수로 넘겨주는 구문을 삽입한다. 메소드 C를 정의하는 구문에 sequence 매개 변수를 추가하고 메소드 정보를 추출하는 탐침 코드를 삽입한다. 마지막으로 테스트 삽입 코드를 임포트 구문과 인스턴스 생성 구문을 삽입한다. 다음은 삽입된 테스트 코드 설명이다.

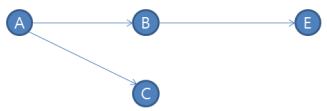
- Import Monitor : 정보 추출 클래스를 임포트한다.
- Monitor test = new Monitor : 정보 추출 클래스의 인스턴스를 생성한다.
- String sequence = test.getSequence() : 시퀀스 정보를 생성하여 변수에 저장한다.
- test.extractMethod() : 메소드 정보를 저장한다.
- test.extractSequence() : 시퀀스 정보 저장한다.
- test.extractParam() : 매개변수 정보를 저장한다.
- test.extractReturn() : 리턴 정보를 저장한다.

②**동적 정보 추출:** 애플리케이션을 작동시켜 표 1과 같은 정보를 추출하여 저장한다.

<표 1> 애플리케이션의 테스트 수행동안 축적된 정보

ID	메소드			시퀀스	매개변수			예외	시간	
	접근변경자	자료형	이름	ID	자료형	이름	객체		분	초
1	public	String	A	abcd111	int	num	ob0	-	12	12
2	public	void	B	abcd111	-			-	12	13
3	public	String	A	adfe322	int	num	ob1	-	12	20
4	public	void	B	adfe322	-			-	12	21
5	public	void	E	adfe322	-			-	12	22
6	public	int	G	defe943	int	num	ob2	ex	12	20
7	public	String	A	ertg458	int	num	ob3	-	12	40
8	public	int	C	ertg458	int	num	ob4	-	12	41

③**동적 메소드 그래프 생성:** 표 1에서 메소드 A가 처음으로 기록되고 그 다음 메소드 B가 기록되었다. 메소드 A, B의 시퀀스 아이디가 같고, 메소드 A가 메소드 B보다 앞선 시간에 기록되었으므로 메소드 B는 메소드 A에 의해 호출된 걸 알 수 있다. 이처럼 메소드의 시퀀스 정보와 메소드의 기록 시간으로 그림 5와 같은 동적 메소드 경로 그래프를 생성한다.



(그림 5) 동적 메소드 경로 그래프

3.4 메소드 경로 커버리지 분석

정적/동적 메소드 경로 그래프를 비교하여 메소드의 전체 경로 커버리지를 분석할 수 있다.

정적 메소드 경로 그래프에서 메소드 경로의 수를 분모로 놓고 동적 메소드 경로 그래프에서 메소드 경로의 수를 분자로 놓는다. 그러면 전체 커버리지는 “(동적 메소드 경로 그래프의 경로 수/ 정적 메소드 경로 그래프의 경로 수) * 100“ 이 된다. 그림 3과 5로 예를 들면 ”2/4 * 100 “으로 메소드 커버리지는 50%가 된다.

3.5 메소드 커버리지 향상을 위한 테스트 데이터 생성

경로 커버리지 분석은 테스트 데이터의 의존성이 강하다 [8]. 그림 3의 메소드 A, C, D에 대해 살펴보자. 메소드

경로는 A->C 혹은 A->D의 두 가지 경로가 생성된다. 메소드 A는 테스트 데이터에 의존하여 메소드 C 와 메소드 D 중 하나를 선택한다. 만약 테스터가 테스트 데이터를 생성할 때 메소드 A가 메소드 C만을 선택하는 데이터를 생성하였다면 테스트 결과는 메소드 D의 경로 커버리지를 구하지 못할 것이다. 이에 우리는 테스트 수행으로 획득한 메소드 매개변수의 정보와 조건문에 의해 분기가 생성되는 메소드의 경로 커버리지를 이용하여 이러한 문제를 해결하는 방법을 제안한다.

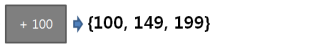
①조건문에 의한 분기를 가진 메소드 중에서 메소드의 한 구간의 경로 커버리지가 1미만인 메소드를 추출한다. 그리고 커버리지가 조건문에 의해 생성되었는지 분석한다. 예를 들어, 그림 3에서 메소드 A는 조건문에 의한 메소드 분기를 가지고 있고 2/3의 커버리지를 나타낸다. 또한 이 커버리지는 조건문에 의한 메소드 분기에 영향을 받았다.

②분석 대상 메소드 매개변수의 정보를 이용하여 매개변수의 객체에 대해 분석한다. 예를 들어, 표 1에서 메소드 A는 자료형이 int이고 변수 이름이 num인 매개변수를 소유하며 이 매개변수의 객체를 세 개를 가지고 있다. 이 정보로 매개변수의 객체를 분석한 결과 매개변수의 객체는 {0, 49, 99}이다.

③단계 ②의 분석을 바탕으로 새로운 매개변수를 생성하여 이 매개변수를 테스트 데이터로 사용한다. 만약 커버되지 않은 경로의 메소드를 호출한다면 이 매개변수를 기준으로 새로운 매개변수를 생성한다. 이러한 매개변수를 다수 생성하여 집합을 만든다. 예를 들어, 단계 ②에 객체 집합은 {0, 49, 99}이다. 일반적으로 int 자료형의 조건문에서 수의 범위에 의해 분기가 결정 된다. 따라서 객체 집합에서 가장 작은 수와 가장 큰 수를 찾아 그 범위를 정한다. 그 범위는 0~99가 된다. 우리는 '0'을 기분으로 반대 범위(-99~1)에 데이터를 랜덤하게 생성한다. 또는 '99'를 기준으로 데이터를 생성한다. 이렇게 생성한 데이터 {-98, -34, -23}이 커버되지 않은 메소드를 호출하였다.

④분석 대상 메소드가 다른 메소드에 호출이 되지 않았다면 단계 ③에서 생성한 데이터는 새로운 테스트 데이터가 된다. 하지만 추출 메소드가 다른 메소드에 의해 호출되었다면 추출 메소드를 호출하는 메소드에 대해 단계 ②와 같이 분석한다. 예를 들어, 메소드 A는 다른 메소드에 의해 호출되지 않는다. 따라서 {-98, -34, -23}은 새로운 테스트 데이터가 되는 것이다. 하지만 메소드 A가 메소드 H에 의해 호출된다고 가정하자. 메소드 H는 자료형이 int 이고 변수 이름이 num인 매개변수와 자료형이 String 이고 변수 이름이 name인 매개변수를 가지고 있다. 테스트 수행에서 추출한 객체는 {(lee, 100), (kim, 149), (cho, 199)}이다.

⑤단계 ②의 객체 집합과 단계 ④의 객체 집합을 분석한다. 분석 방법은 두 집합에서 같은 자료형의 데이터를 분석하여 특정 패턴을 알아낸다. 예를 들어, 단계 ②의 {0, 49, 99}와 단계 ④의 {(lee, 100), (kim, 149), (cho, 199)}를 분석한다. 여기서 두 데이터 집합이 공통을 갖는 자료형은 int 형이고 비교 대상이 없는 String 형을 제거한 집합{100, 149, 199}와 {0, 49, 99}를 비교 분석한다. 그림 6와 같이 {0, 49, 99}의 집합을 입력하고 {100, 149, 199}의 집합을 출력한다. 이러한 데이터의 입출력을 생성한다면 박스 안에 연산은 + 100 된다. 즉 메소드 H의 매개변수 {100, 149, 199}는 - 100 연산 후 메소드 A의 매개변수 {0, 49, 99}로 전달된다.



(그림 6) 매개변수 집합의 분석 사례

⑥단계 ③에서 추출한 데이터 집합을 단계 ⑤에서 분석한 박스 정보를 이용하여 단계 ④에서의 매개변수 정보로 생성한다. 그 후 단계 ③을 수행한다. 예를 들어, 단계 ③에서 생성한 데이터 {-98, -34, -23}은 단계 ⑤의 그림 6의 박스에 적용하면 데이터 {2, 64, 77}이 된다. 이를 통해 얻은 데이터 집합을 메소드 H의 매개변수에 대한 정보를 이용하여 복원하면 {(lee, 2), (kim, 64), (cho, 77)}이 된다.

기존의 경로 커버리지를 위한 데이터 생성은 데이터 흐름 분석을 통해 이루어져 왔다. 하지만 이 방법은 개념적으로 명확한 방법이지만 실제로 이 방법을 구현하기엔 기술적인 한계가 있다. 이 문제를 해결하기 위해 우리는 이 절에서 제안한 방법과 같이 데이터 흐름 분석에 의존하지 않는다.

3.6 J2EE 애플리케이션 테스트

테스터는 애플리케이션의 명세를 참조하여 테스트 데이터를 생성한다. 이 테스트 데이터를 이용하여 탐침 코드가 삽입된 애플리케이션을 테스트한다. 애플리케이션의 정적 분석 정보와 탐침 코드에 의해 추출된 동적 정보로 3.4절과 3.5절을 수행되어, 메소드 커버리지 분석과 새로운 테스트 데이터가 생성된다. 테스터는 이 테스트 데이터로 애플리케이션을 테스트를 수행하여 이전보다 높은 메소드 커버리지를 구한다.

4. 결 론

본 논문은 J2EE 애플리케이션을 테스트하는 방법을 제안하였다. 이 방법은 J2EE 애플리케이션을 구성하는 EJB 컴포넌트들을 정적 및 동적 분석하여 정보를 획득하고 이 분석 정보를 이용하여 정적/동적 메소드 경로 그래프를 생성한다. 그리고 정적/동적 메소드 경로 그래프를 비교하여 애플리케이션의 메소드 커버리지를 분석하였다. 그리고 애플리케이션의 동적 정보와 메소드 커버리지 분석 정보를 이용하여 보다 높은 메소드 커버리지를 형성하는 테스트 데이터의 생성 방법 제안하였다. 하지만 데이터 생성 방법은 간단한 로직을 가지는 메소드에 그 활용이 제한된다. 향후 우리는 메소드 커버리지를 향상시키기 위한 더욱 강력한 테스트 데이터 생성 방법에 대해 연구할 것이다.

참고 문헌

[1] Sun microsystems Inc., J2EE Adopttion, http://www.sun.com/software/products/appsrvr/wps_application1.xml

[2] 국승학, "EJB 컴포넌트 테스트 환경 자동 생성 방안 및 도구 구현", 충남대 졸업 논문 석사, 2006. 02

[3] 이성희, "EJB Programming", FREELEC, 2007, 01

[4] 윤회진, 최병주, "CDSD에서의 컴포넌트 조립 테스트 기법", 정보과학논문지: 소프트웨어 및 응용, 2002. 10

[5] Rod Johnson, "Expert one-on-One J2EE Design and Development", Wrox, 2003

[6] Scott W. ambler, "A J2EE Testing Primer", http://microsites.cmp.com, 2001.05

[7] Scott Stirling, "Testing J2EE applications", Java World, 2004.08

[8] 한국정보통신기술협회, "소프트웨어 테스트 전문기술 기초 분야", 정보통신부 S/W시험 인력양성 사업, 2005