컴포넌트 맞춤 오류를 위한 테스트 기법
(A Test Technique for the Component Customization Failure)

윤 회 진 † 최 병 주 ††
(Hoijin Yoon) (Byoungju Choi)

요 약 컴포넌트 맞춤(customization)으로 인해 변형된 '인터페이스 부분'과 '핵심기능 부분'의 상호작용에서 발생하는 오류를 효과적으로 테스트하기 위한 테스트 기법은 필요하다. 본 논문에서는 오류 삽입 기법과 뮤테이션 테스트 케이스 선정 기법을 사용하여 컴포넌트 맞춤 테스트 기법을 제안한다. 컴포넌트의 인터페이스 가운데 맞춤 오류가 일어나는 곳에만 오류를 삽입하여, 맞추어진 컴포넌트와 오류가 삽입된 컴포넌트를 차별하는 테스트 케이스를 선정한다. 따라서 본 기법은 컴포넌트 맞춤에 의한 오류를 발견할 가능성이 높은 테스트 케이스를 선정할 수 있으며, 인터페이스 가운데 맞춤 오류가 일어나는 부분만을 테스트 대상으로 함으로써, 테스트 시간을 단축할 수 있다.

Abstract *The test technique for the failure caused by interaction between the customized interface and core function is necessary. We propose a component customization test technique by using the fault injection technique and the mutation test case selection technique. Our technique injects fault into where the customization failure may take place and selects the test case that differentiates the fault-injected component from the customized-component. Therefore, our test case has a good fault-detectability and can reduce the testing time by injecting a fault only into a place where the customization failure may take place in the interface.*

1. 서 론

컴포넌트 기반 소프트웨어 개발(Component-Based Software Development : CBSD)에서 이용되는 컴포넌트는 다른 개발자에 의해 개발된 경우가 많으므로, 컴포넌트를 재사용하는 입장에서는 해당 컴포넌트에 대한 개발 산출물이나 기능에 대한 소스 코드를 알 수 없다. 이러한 컴포넌트의 특성을 고려해 볼 때, 기존의 테스트 기법을 적용하기에는 무리가 있다. 또한 Arian5[1]가 주는 교훈처럼 CBSD에서 테스트의 중요성이 더욱 커지고 있다. 따라서 본 논문에서는 컴포넌트가 재사용될 때 반드시 수행해야 하는 테스트를 위한 기법을 제안한다.

CBSD는 컴포넌트 맞춤(customization) 테스트, 컴포넌트 조립(composition) 테스트, 시스템 테스트, 컴포넌트 자격부여(qualification) 테스트등의 다양한 수준의 테스트를 요구한다[2]. 이 가운데 본 논문은 컴포넌트를 개발 도메인의 특정 요구사항에 맞출 때 요구되는 '컴포넌트 맞춤 테스트'를 대상으로 한다. 컴포넌트들은 되도록 많은 CBSD에서 재사용될 수 있도록 일반적인 요구사항을 기반으로 개발된다[3]. 즉 컴포넌트를 재사용할 때는 그림 1과 같은 맞춤 작업이 필요하다.

컴포넌트 맞춤을 통해 컴포넌트는 변형되고, 이때 오류가 발생할 수 있다. 컴포넌트의 개발 산출물들은 공개되지 않고 단지 인터페이스만이 공개되기 때문에 컴포넌트에서 발생하는 오류는 기존의 개발들에서 발생하는 오류들보다 더 테스트하기 어렵다. 본 논문에서는 이와 같은 컴포넌트의 특성을 고려하여, 컴포넌트 맞춤 테스트에 적합한 테스트 케이스를 추출할 수 있는 기법을 제안한다.

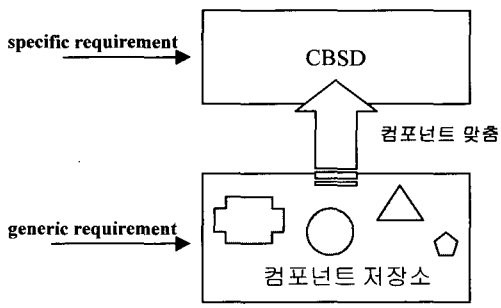


그림 1 컴포넌트 맞춤

† 학생회원 : 이화여자대학교 컴퓨터학과
hoijin@cs.ewha.ac.kr
†† 종신회원 : 이화여자대학교 컴퓨터학과 교수
bjchoi@mm.ewha.ac.kr
논문접수 : 1999년 5월 24일
심사완료 : 1999년 12월 1일

2장에서는 컴포넌트 맞춤에 대해 정의하고 컴포넌트 맞춤 테스트와 오류 삽입 기법의 기존 연구를 분석한다. 3장에서는 컴포넌트 맞춤 패턴을 정의하고 4장에서는 본 논문에서 제안하는 컴포넌트 맞춤 오류를 위한 테스트 기법을 기술한다. 5장에서는 본 논문의 기법을 분석하고, 마지막 6장에서는 결론과 향후 연구 과제를 제시한다.

2. 관련연구

2.1 컴포넌트 맞춤

Souza는 컴포넌트를 소프트웨어 구현 패키지로 정의하고, 컴포넌트가 가져야 하는 세 가지 특성을 제시하였다[4]. 첫째 컴포넌트는 독립적으로 개발되고 보급될 수 있어야 하며, 둘째 컴포넌트는 제공(provide) 인터페이스와 요구(required) 인터페이스를 가져야 한다. 셋째 컴포넌트는 컴포넌트가 지니고 있는 속성을 사용자의 요구에 따라 customize할 수 있어야 한다. 즉 컴포넌트는 컴포넌트 내의 세부 구현 사항들을 외부에 노출시키지는 않지만, 컴포넌트 사용자들이 자신의 목적에 맞도록 컴포넌트가 지니는 속성이나 메소드들을 변경할 수 있도록 지원되어야 한다. 이를 컴포넌트 맞춤이라고 한다. 컴포넌트 맞춤은 컴포넌트의 인터페이스를 통해 이루어지며, 컴포넌트의 속성을 간단하게 수정하는 차원을 넘어서 일반적인 요구사항을 기반으로 작성된 컴포넌트를 특정 도메인의 요구사항으로 확장하는 작업이다. 컴포넌트 맞춤은 컴포넌트를 다루는데 있어서 매우 중요한 작업이다. Paul Allen은 컴포넌트는 일반적인 요구사항에 따라서 만들어지며, 이들은 컴포넌트 기반 개발 과정에서 각 비즈니스 영역에 따른 특정 요구사항에 맞도록 컴포넌트 맞춤을 통해 확장되어져야 한다는 원칙에 따라 CBSD 프로세스를 정의하였다[3].

2.2 컴포넌트 맞춤 테스트

컴포넌트 테스트를 위해 기존의 화이트박스 테스트 기법 또는 블랙박스 테스트 기법 등의 단위 테스트 기법을 컴포넌트의 인터페이스에 단순히 적용할 수 있다. 그러나 이 경우 컴포넌트 맞춤으로 인해 변형된 인터페이스와 고정된 핵심기능 부분사이에서 발생할 수 있는 오류를 효과적으로 테스트할 수 없다. 따라서 '인터페이스 부분'과 '핵심기능 부분'사이에서 발생하는 오류를 위한 테스트 케이스를 선정하는 것이 컴포넌트 맞춤 테스트이다. 따라서 컴포넌트 맞춤 테스트는 단위 테스트라기 보다는 일종의 통합 테스트라 할 수 있다.

Hölzle[5]은 컴포넌트를 객체로 보고, 컴포넌트를 통합하는 방법으로써 상속과 연관 메카니즘을 통한 컴포넌트 통합의 문제점을 추출하고, 그에 대한 대안으로서, 컴포넌트의 인터페이스 모양을 바꾸는 'type adaptation'과 추가·수정·삭제의 내용만을 갖는 'extension hierarchy'를 제시하였다. 그러나 본 논문은 이처럼 컴포넌트를 객체로 보는 수준을 넘어, 컴포넌트를 구성하는 핵심기능 부분과 인터페이스 부분의 특성에 따라, 클래스를 블랙박스 클래스와 화이트박스 클래스로 구분하여, 패턴을 정의하고, 나아가 실제 테스트에 적용할 수 있는 기법을 제안함으로써, 기존 연구들 보다 더욱 구체적인 결과를 갖는다.

대부분의 객체지향 시스템을 위한 테스트에서는 시나리오를 기반으로 추출되는 무수히 많은 메소드 시퀀스 형태를 테스트 케이스로 한다. Chen[6]은 클래스 내부의 메소드들 사이에 존재하는 규칙을 표현하는 axiom을 기준으로 테스트 케이스를 추출하였다. 각 클래스의 axiom들을 통해, 메소드 시퀀스들 가운데 기본쌍(fundamental pair)을 선정하여, 그것을 테스트 케이스로 이용하였다. 하나의 기본쌍을 이루는 두개의 메소드 시퀀스의 수행 결과는 같아야 한다. 따라서 기본쌍의 수행 결과가 서로 다를 경우 해당 클래스에 오류가 존재한다는 것을 알 수 있다. 이 방법은 일단 클래스 하나를 대상으로 수행하고 있으므로, 컴포넌트의 핵심기능 부분과 인터페이스 부분 사이의 상호 작용에 대한 테스트를 하기에는 부족하고, 또한 Chen의 방법은 클래스의 소스코드를 모두 고려하여 axiom을 추출하는 절차가 필요하므로, 핵심기능 부분이 블랙 박스 형태를 띄고 있는 컴포넌트의 특성을 소화하기에는 무리가 있다.

시나리오를 기반으로 수행되는 객체지향 테스트 기법은 시나리오나 순서도 등과 같은 다양한 산출물이 요구된다. 그러나 컴포넌트의 특성을 고려해 볼 때, 하나의 컴포넌트에 대해 시나리오나 순서도 등의 많은 양의 정보가 첨부된다면, 이들 정보를 유지·관리하기 위한 비용으로 오히려 개발비용이 증가하게 될 것이다. 본 연구에서는 시나리오 기반이 아니라, 컴포넌트의 인터페이스 부분에 오류를 인위적으로 삽입함으로써 테스트 케이스를 선정한다. 인터페이스 부분은 핵심기능 부분의 행위에 변화를 줄 수 있고, 수정 가능하므로 이 부분에 실제 컴포넌트 맞춤에 의한 오류가 존재할 수 있다.

2.3 오류 삽입 기법

오류 삽입 기법은 소프트웨어 코드의 어떤 위치에 임의로 오류를 삽입한 후, 소프트웨어가 어떻게 동작하는지를 관찰하는 기술로서, 프로그램에 의해 수행되는 경로가 매우 복잡하거나 수행 경로를 쉽게 찾기 어려운 경우에 유용하다[7]. 이러한 경우 오류를 시스템에 삽입한다면, 수행 경로를 고려하지 않더라도, 시스템이 스스

로 오류를 보여줄 것이다. 컴포넌트의 경우, 공개되지 않는 핵심기능 부분으로 인해 수행 경로를 쉽게 찾기 어렵기 때문에, 컴포넌트 테스트에는 오류 삽입 기법이 적합함을 알 수 있다.

오류 삽입 기법을 테스트 케이스의 적정성(adequacy) 측정에 응용한 예가 뮤테이션 테스트[8]이다. 뮤테이션 테스트는 프로그램 코드에 문법적으로는 맞도록 프로그램을 변형하여 생성된, 즉 일종의 오류가 삽입된 뮤턴트와 원래 프로그램을 차별하는 테스트 케이스를 선정하는 기법이다. 본 논문에서는 컴포넌트의 인터페이스에 뮤테이션 기법을 단순하게 적용하지 않고, 인터페이스 가운데 핵심기능 부분에 영향을 줄 수 있는, 즉 맞춤으로 인한 오류를 일으킬 수 있는 부분만을 대상으로 오류를 삽입하는 방안을 택하였다. 이렇게 함으로써, 컴포넌트 오류를 발견할 가능성이 높은 테스트 케이스를 선정할 수 있으며, 인터페이스 가운데 맞춤으로 인한 오류가 일어나는 곳만을 대상으로 함으로써, 수많은 뮤턴트를 실행해야하는 뮤테이션 테스트의 문제점을 감소시킬 수 있다.

3. 컴포넌트 맞춤 패턴

본 장에서는 컴포넌트를 구성하는 블랙박스 클래스와 화이트박스 클래스로 정의하고, 컴포넌트 맞춤 패턴을 정의한다.

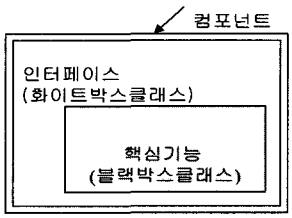


그림 2 컴포넌트의 핵심기능과 인터페이스

3.1 컴포넌트 정의

컴포넌트는 그림 2와 같이 공개되는 인터페이스와 공개되지 않는 핵심기능으로 구현된다. 현재 컴포넌트 모델로서 많이 사용되고 있는 자바빈즈[9]도 컴포넌트의 핵심기능 부분은 private나 protected로 구현하고, 인터페이스 부분은 public으로 구현하여, 각각의 특성을 반영한다. 물론 이 두 부분이 문법적으로 동일한 객체지향 클래스에 존재할 수도 있으나, 의미적으로 이 두 부분은 private나 protected로 표현된 폐쇄적인 부분과 public으로 표현되는 공개적인 부분으로 구분되어 취급된다. 따라서 본 논문은 인터페이스를 구현한 부분을 화이트박스 클래스로, 그리고 컴포넌트 핵심기능을 구현한 부분을 블랙박스 클래스로 정의한다.

[정의 1] 블랙박스클래스 (*B*)

블랙박스클래스란 컴포넌트의 핵심기능 부분을 구현한 것으로서, 클래스의 소스코드는 공개되지 않고, 단지 인터페이스에 정의된 접근 메소드를 통해서만이 접근 가능하다. 블랙박스 클래스는 *B*로 표현한다.

■ 예 : "NeedleGauge"컴포넌트의 *B*는 그림 3의 *B*와 같다.

[정의 2] 화이트박스클래스 (*W*)

화이트박스 클래스란 컴포넌트의 인터페이스 부분을 구현한 것으로서, 클래스의 소스 코드가 공개되고 수정이 가능하다. 화이트박스 클래스를 변형함으로써, 컴포넌트의 행위나 특성을 변경시킬 수 있다. 화이트박스 클래스는 W로 표현한다. 컴포넌트 맞춤으로 변형된 *W*를 *cW*라고 하고, 오류가 삽입된 *cW*를 *fW*라고 한다.

■ 예 : "NeedleGauge"컴포넌트의 W는 그림 3의 W와 같다.

[정의 3] 컴포넌트 (*BW*)

컴포넌트는 *B*와 *W*가 결합된 단위로서 *BW*로 표현한다. 컴포넌트 맞춤으로 변형된 *BW*를 *cBW*라고 하고, 오류가 삽입된 *cBW*를 *fBW*로 표현한다.

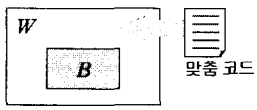
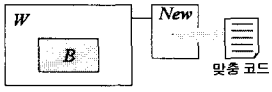
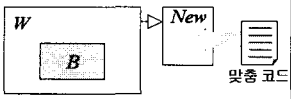
■ 예 : "NeedleGauge" BW는 그림 3과 같다.

3.2 컴포넌트 맞춤 패턴

컴포넌트 맞춤 패턴은 크게 '컴포넌트 맞춤 구문(syntactic) 패턴'과 '컴포넌트 맞춤 의미(semantic) 패턴'으로 나누어 볼 수 있다. 컴포넌트 맞춤 구문 패턴은 *W*를 구문상 '어떻게' 컴포넌트 맞춤을 하는지에 대한 패턴이고, 컴포넌트 맞춤 의미 패턴은 컴포넌트를 '왜' 컴포넌트 맞춤을 하는지에 대한 패턴이다. 컴포넌트 맞춤 구문 패턴은 *W*를 변경하는 구현상의 패턴으로 기존의 설계패턴[10]을 참조하여 표1의 세 가지로 나누어 볼 수 있다. 이때 맞춤을 위해 추가되거나 수정되는 코드를 '맞춤 코드'로 표현하고, 맞춤을 위해 새로 작성되는 클래스를 *New*로 나타낸다.

표 1의 컴포넌트 맞춤 구문 패턴들은 컴포넌트를 구성하는 *B*와 *W*의 구현 관계가 무엇으로 되어 있는가는 고려하지 않았다. 왜냐하면 *B*와 *W*사이의 구현관계는 컴포넌트 맞춤으로 변경될 수 없는 부분이기 때문이다.

표 1 컴포넌트 맞춤 구문 패턴

	컴포넌트 맞춤 구문 패턴	설 명
1		주어진 *W*를 그대로 유지하고, *W*의 애트리뷰트와 메소드에 대한 소스 코드를 수정
2		맞춤 코드를 새로운 클래스, *New*로 작성하고, 작성된 새로운 클래스를 주어진 *W*가 참조할 수 있도록 연관 관계 설정
3		맞춤 코드를 새로운 클래스, *New*로 작성하고, 작성된 새로운 클래스를 주어진 *W*가 이용할 수 있도록 새로 작성한 클래스를 주어진 화이트박스 클래스의 상위클래스로 상속 관계 설정

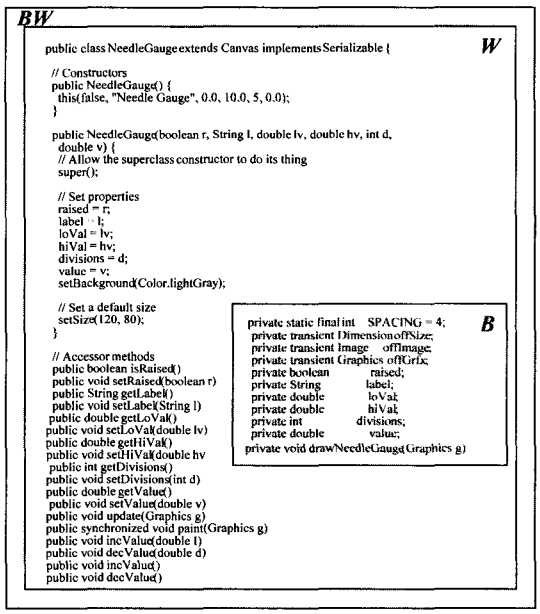


그림 3 'NeedleGauge'컴포넌트의 *B, W, BW*

컴포넌트 맞춤은 의미에 따라 컴포넌트 '속성(property) 변경 맞춤'과 '기능 추가 맞춤'으로 구분된다. 속성 변경 맞춤은 인터페이스를 통해서 컴포넌트의 속성 변수의 값을 수정하거나 조건을 부여하여 변경시키는 경우로서, NeedleGauge 컴포넌트(그림 3)의 속성 가운데 돋음효과-NeedleGauge가 3차원으로 보이도록 하는 효과-에, 범위의 최대값이 10이하일 경우에는 항상 'false'로 설정하도록 하는 경우가 속성 변경 맞춤에 속한다. 기능 추가 맞춤은, 개발 도메인에서 요구되는 특정한 기능을 컴포넌트에 플러그인을 사용하거나 새로운 코드를 작성하여 특정한 기능을 추가하는 경우이다. 비디오 대여점 시스템의 멤버쉽 컴포넌트에 대해 특정 도메인에 따른 멤버 클래스와 계정 클래스 기능을 추가하는 경우가 기능 추가 맞춤에 속한다. 이들을 컴포넌트 맞춤 의미 패턴으로 표 2와 같이 정의하였다.

표 2 컴포넌트 맞춤 의미 패턴

컴포넌트 맞춤 의미 패턴		설 명
1	속성 변경 맞춤	컴포넌트의 속성 변수의 값을 변경시키는 맞춤
2	기능 추가 맞춤	새로운 기능을 갖는 코드나 플러그인을 추가하는 맞춤

4. 컴포넌트 맞춤 오류를 위한 테스트 기법

컴포넌트 맞춤 오류는 인터페이스를 변형할 때 발생할 수 있다. 본 논문은 이러한 컴포넌트 맞춤 오류를 테스트할 수 있는 기법을 제안한다. 그림 4 (b)와 같이 *BW*의 *B*는 변하지 않고, 그림 4 (a)의 *W*에 빗금부분이 추가되어 *cBW*로 맞추어진다. 이때 컴포넌트 맞춤 오류는 *cBW*에 존재할 수 있으므로, 컴포넌트 맞춤 테스트는 *cBW*를 대상으로 한다.

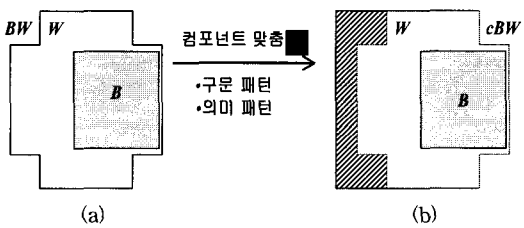


그림 4 컴포넌트 맞춤

*cBW*의 *cW* 가운데 특히 *B*와 상호작용을 하는 부분이 컴포넌트 맞춤 테스트에서 주목해야 하는 부분이다. 이 부분을 '오류 삽입 대상'이라 하고, 오류 삽입 대상에 오류를 삽입하는 연산자를 '오류 삽입 연산자'라고 한다. 본 논문은 *cBW*의 모든 부분에 오류를 삽입하지 않고,

그림 5에서처럼 컴포넌트 맞춤으로 변형된 *cBW*의 *cW* 가운데 '오류 삽입 대상'에만 오류를 삽입하여 그림 5 (b)의 *fBW*를 생성한다. 이때 오류 삽입 연산자를 이용한다. 이렇게 생성된 *fBW*와 *cBW*를 차별화할 수 있는 테스트 케이스를 선정함으로써, 컴포넌트 맞춤 오류에 적정한 테스트 케이스를 선정하는 것을 목적으로 한다.

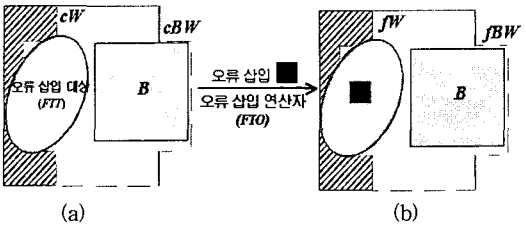


그림 5 오류 삽입 대상과 오류 삽입 연산자를 이용한 오류 삽입

4.1 오류 삽입 대상

본 기법은 2장에서 기술한대로 오류 삽입 기법을 이용한다. 이때 '오류를 어디에 삽입해야 하는가'의 문제는 선정되는 테스트 케이스의 품질을 좌우한다. 오류를 삽입할 대상으로 우선 맞춤 코드만을 고려해 볼 수 있다. 그러나 맞춤 코드가 *cW*에 대한 단위 테스팅 단계에서 모두 테스트되어 그 곳에는 오류가 존재하지 않는다고 하여도, 맞춤 코드가 그외의 *cW* 메소드 및 *B*와 상호작용을 하며 수행될 때, 오류가 발생할 수 있다. 이는 통합 테스팅에서 발견된다. 한편 *cW* 코드 전체를 대상으로 오류를 삽입하여 테스트하는 방법도 고려해 볼 수 있다. 이 방법에 의해서 선정되는 테스트 케이스들은 당연히 모든 오류를 찾아낼 가능성이 있을 것이다. 따라서 안전한 방법이긴 하지만 테스팅 시간의 지연등으로 비효율적이다. *cW* 코드 가운데 *B*와의 상호작용에 영향을 주는 부분에만 오류를 삽입할 수 있다면, 보다 효율적으로 컴포넌트 맞춤 오류를 테스트할 수 있다. 따라서 컴포넌트 맞춤 패턴에 기반하여, 오류를 삽입할 대상을 선정하는 작업이 매우 중요하다. 본 논문은 오류를 삽입할 대상을 다음과 같이 정의한다.

[정의 4] 오류 삽입 대상 (*FIT*)

오류 삽입 대상이란 *cW*의 코드들 가운데 오류가 삽입되는 곳으로서, 컴포넌트 맞춤으로 인해 수정되거나 추가된 *W*의 코드들 가운데, 특히 *B*에 영향을 주는 부분이다. 이는 *FIT*로 표현되며, *FIT*는 컴포넌트 맞춤 구문 패턴과 의미 패턴에 따라 추출된다. $FIT_{syn,sem}$은 컴포넌트 맞춤 구문 패턴 *syn*과 컴포넌트 맞춤 의미 패턴 *sem*에서의 *FIT*를 의미한다.

*FIT*는 *cW*가운데 *B*와의 상호작용에 영향을 미칠 수 있는 부분이다. 따라서 *FIT*는 *B*와 *cW*에서 일어나는 상호작용을 기반으로 추출될 수 있다. 그림 6의 α와 β는 *B*와 *cW*의 상호 작용을 표현한다. *cW*에서 맞춤으로 인해 수정되거나 추가된 코드는 그림 6의 α와 β로 컴포넌트의 *B*에 영향을 주어 컴포넌트 기능을 도메인 요구사항에 맞게 customize한다.

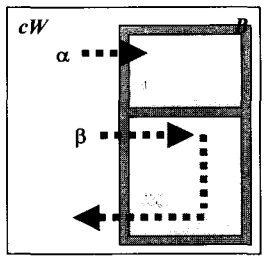


그림 6 *B*와 *cW*의 상호작용 (α, β)

컴포넌트 맞춤 의미 패턴 1에 의해 W는 컴포넌트 속성 변수를 변형시키고 *B*는 *cW*의 변형된 속성 변수를 사용하여 컴포넌트를 수행시킨다. 이것이 그림 6의 α이다. 컴포넌트 맞춤 의미 패턴 2에 의해 *W*에 새로운 기능이 추가되어 *cW*가 되고, *B*는 인터페이스를 통해서 추가된 기능을 수행한다. 이것이 그림 6의 β이다. 그림 6의 α는 컴포넌트 맞춤 의미 패턴 1에 의한 것이고, 그림 6의 β는 컴포넌트 맞춤 의미 패턴 2에 의한 것이다. 즉, 컴포넌트 맞춤 의미 패턴에 따라 *FIT*는 달라진다.

한편, 컴포넌트 맞춤 구문 패턴은 맞춤 코드와 인터페이스의 구조적 관계를 결정한다. 그에 따라 *B*와 상호작용을 하는 부분이 차별화 되므로, 컴포넌트 맞춤 구문 패턴에 따라 *FIT*가 달라진다.

각 컴포넌트 맞춤 패턴에 따라 정의한 *FIT*는 표 3과 같다.

4.2 오류 삽입 연산자

컴포넌트 맞춤 오류를 테스트하기 위해 *cBW*의 *FIT*에 오류를 삽입하는 연산자를 오류 삽입 연산자라고 하고, 다음과 같이 정의한다.

[정의5] 오류 삽입 연산자 (*FIO*)

오류 삽입 연산자는 *cBW*의 *FIT*에 오류를 삽입하는 연산자이다. *FIO*로 표현하며, *FIO*를 통해 오류가 삽입된 *cBW*를 *fBW*라고 한다. *FIO*는 *FIT*와 오류 삽입 활동(Fault Injection Action : *FIA*)으로 구성된다. *FIA*는 *FIT*에 오류를 삽입하는 행위로서 대치(Replace)와 삭제(Delete)가 이에 해당한다.

표 3 오류 삽입 대상 (*FIT*)

FIT		설명
$FIT_{1,1}$	WSP	*cW*의 set메소드가 호출되는 곳의 set메소드 매개변수 값
	WPR	*cW*의 set메소드에서 속성변수를 정의하는 rhs(right-hand-side)문장의 요소
	WSC	호출된 *cW*의 set메소드에 영향을 주는 제약문
	WPC	*cW*의 set메소드에서 속성변수를 정의하는 문장에 영향을 주는 제약문
	WNsP	*cW*의 set메소드에서 호출되는 새로운 메소드의 매개변수 값
	WNsR	*cW*의 set메소드에서 호출되는 새로운 메소드의 반환값
$FIT_{1,2}$	WOP	*cW*의 접근메소드를 제외한 나머지 메소드의 매개변수 값
	WOR	*cW*의 접근메소드를 제외한 나머지 메소드의 매개변수 값
	WNoP	*cW*의 접근메소드를 제외한 나머지 메소드에서 호출된 새로운 메소드의 매개변수 값
	WNoR	*cW*의 접근메소드를 제외한 나머지 메소드에서 호출된 새로운 메소드의 반환값
$FIT_{2,1}$	NSP	*New*에서 사용되는 *cW*의 set메소드의 매개변수 값
	NMpP	*cW*에서 속성변수를 정의하는 lhs문장에 속하는 *New*의 메소드의 매개변수 값
	NMpR	*cW*에서 속성변수를 정의하는 lhs문장에 속하는 *New*의 메소드의 반환값
	NApR	*cW*에서 속성변수를 정의하는 lhs문장에 속하는 *New*의 애트리뷰트 값
	NSC	*New*에서 사용되는 *cW*의 set메소드의 매개변수 값
	WMC	*cW*에서 사용되는 *New*의 메소드 호출에 영향을 주는 제약문
$FIT_{2,2}$	NMoR	*cW*의 접근메소드를 제외한 나머지 메소드에서 호출되는 *New*의 메소드의 반환값
	NMoP	*cW*의 접근메소드를 제외한 나머지 메소드에서 호출되는 *New*의 메소드의 매개변수 값
$FIT_{3,1}$	NMpP	*cW*에서 속성변수를 정의하는 lhs문장에 속하는 *New*의 메소드의 매개변수 값
	NMpR	*cW*에서 속성변수를 정의하는 lhs문장에 속하는 *New*의 메소드의 반환값
	NApV	*cW*에서 속성변수를 정의하는 lhs문장에 속하는 *New*의 애트리뷰트 값
	WMC	*cW*에서 사용되는 *New*의 메소드 호출에 영향을 주는 제약문
$FIT_{3,2}$	NMoR	*cW*의 접근메소드를 제외한 나머지 메소드에서 호출되는 *New*의 메소드의 반환값
	NMoP	*cW*의 접근메소드를 제외한 나머지 메소드에서 호출되는 *New*의 메소드의 매개변수 값

■ 예. *FIT*가 "WPC"이고, *FIA*가 "Replace : R"이면, 이를 표현하는 *FIO*는 "WPCR"이 된다.

각 컴포넌트 맞춤 패턴의 *FIT*에 대해, 테스트 케이스를 추출하기 위한 모든 *FIO*들은 표 4와 같다.

4.3 컴포넌트 맞춤 테스트 케이스 선정 기법

컴포넌트 맞춤 테스트의 테스트 케이스는 메소드 시퀀스 형태이다. 컴포넌트 맞춤 패턴에 따른 다양한 *FIO*들을 적용하여 생성된 *fBW*와, 테스트 대상이 되는 *cBW*를 차별화할 수 있는 메소드 시퀀스 형태의 테스트 케이스를 추출한다. 이러한 테스트 케이스를 추출하는 '컴포넌트 맞춤 테스트 케이스 선정 기법'을 아래와 같이 정의한다.

[정의 6] 컴포넌트 맞춤 테스트 케이스 선정 기법

Customize된 *BW*를 *cBW*라고 하고, *FIO*를 사용하여 *cBW*의 *FIT*에 오류가 삽입된 *cBW*를 *fBW*라고 할 때, 컴포넌트 맞춤 테스트 기법에 의해 선정되는 테스트 케이스, *TC*는 *cBW*와 *fBW*를 차별화 하는 일련의 메소드 순서들로서 아래와 같다.

$TC = \{ x \mid cBW(x) \neq fBW(x), \text{ where } x \text{ is a method sequence.} \}$

정의 6의 기법은 뮤테이션 테스트 케이스 선정 기법[8]과 유사하다. 그러나 본 기법은 *cBW*에 뮤테이션 테스트 기법을 단순히 적용한 것과는 차이가 있다. 본 기법은 컴포넌트 맞춤 패턴에 따라, *cBW* 가운데 특히 *B*와 상호작용을 하는 영역, 즉 컴포넌트 맞춤 테스트에서 반드시 테스트되어야 하는 부분을 *FIT*로 추출하고 그곳에 오류를 심음으로써 테스트 케이스를 선정한다. 이렇게 본 기법에 의해 선정된 테스트 케이스는 컴포넌트 맞춤으로 인한 오류를 발견할 가능성이 높고, 본 기법은 인터페이스 가운데 맞춤 오류가 일어나는 곳만을 대상으로 함으로써, 수많은 뮤턴트를 실행해야하는 뮤테이션 기법의 문제점을 감소시킬 수 있다.

4.4 예제

본 장에서는 예제를 통해 본 논문에서 제안한 컴포넌트 맞춤 테스트 기법의 특성을 보인다.

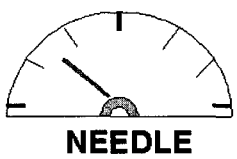


그림 7 NeedleGauge

(1) 대상 컴포넌트

그림 7처럼 자동차 속도계나 오디오 시그널 레벨을 표시하는 미터 등에 이용되는 NeedleGauge 컴포넌트[9]를 대상 컴포넌트로 한다. 그림 3은 NeedleGauge 컴포넌트의 코드이다. *W*의 각 메소드들의 소스코드는 지면관계상 생략하고 각 메소드들의 시그너처만을 표현하

표 4 오류 삽입 연산자 (*FIO*)

FIT / FIA	$FIT_{1,1}$						$FIT_{1,2}$				$FIT_{2,1}$						$FIT_{2,2}$		$FIT_{3,1}$				$FIT_{3,2}$	
	WSP	WPR	WSC	WPC	WNsP	WNsR	WOP	WOR	WNoP	WNoR	NSP	NMpP	NMpR	NApR	NSC	WMC	NMoR	NMoP	NMpP	NMpR	NApV	WMC	NMoR	NMoP
Replace	WSPR	WPRR	WSCR	WPCR	WNsPR	WNsRR	WOPR	WORR	WNoPR	WNoRR	NSPR	NMpPR	NMpRR	NApRR	NSCR	WMCR	NMoRR	NMoPR	NMpPR	NMpRR	NApVR	WMCR	NMoRR	NMoPR
Delete	WSPD	WPRD	WSCD	WPCD	WNsPD	WNsRD	WOPD	WORD	WNoPD	WNoRD	NSPD	NMpPD	NMpRD	NApRD	NSCD	WMCD	NMoRD	NMoPD	NMpPD	NMpRD	NApVD	WMCD	NMoRD	NMoPD

였다. NeedleGauge 컴포넌트는 NeedleGauge에 이름을 부여하는 라벨, 범위의 최저값, 범위의 최대값, 눈금의 수, 값, 돋음효과등을 속성으로 하고, NeedleGauge 컴포넌트의 *W*에는 각 속성의 값을 설정하고 읽어갈 수 있는 set메소드, get메소드들과 paint(), 그리고 NeedleGauge 컴포넌트를 구동시킬 수 있는 NeedleGauge()로 이루어져 있다.

(2) 컴포넌트 맞춤

NeedleGauge컴포넌트는 범위를 벗어나는 값에 대해, 최대값 또는 최소값을 임의로 설정하여 표시한다. 본 예제에서는 범위에서 벗어나는 수치가 되더라도, NeedleGauge가 보여주는 범위가 수정되어서 정확한 수치를 항상 표현할 수 있도록 컴포넌트 맞춤을 한다. 이는 컴포넌트 맞춤 의미 패턴 1에 해당하는 것으로서, 본 예제에서는 컴포넌트 맞춤 구문 패턴 1로 컴포넌트 맞춤을 한다. 그림 8 (a)는 그림 3의 *W*에 속하는 메소드들 가운데 변형될 메소드 코드만을 보이고, (b)는 *cW*의 메소드로서 (a)를 컴포넌트 맞춤을 하여 변형된 코드만을 보인다.

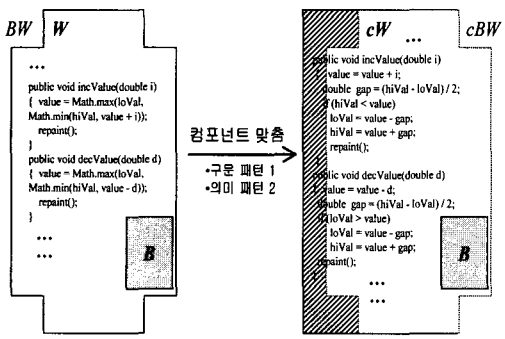


그림 8 NeedleGauge *BW*와 *cBW*

(3) 컴포넌트 맞춤 테스트

그림 8 (b)의 *cW*에 표현된 코드가 맞춤 코드이다. 맞춤 코드의 incValue()와 decValue() 메소드 내에는 오류가 없다. 즉, 그림 8 (b)의 *cW*에 속한 각 메소드들에 대한 단위 테스트를 통해 오류가 없음을 알 수 있다. 맞춤 코드 자체는 공개된 부분이므로 기존의 테스트 기법을 그대로 적용할 수 있다. 그러나 맞춤 코드에 오류가 없다는 것이 컴포넌트 맞춤으로 인한 오류가 없다는 것을 보장하지 않는다. 왜냐하면 맞춤 코드에서 수행되는 테스트는 *cW*에 속한 메소드들에 대한 단위 테스트로, 컴포넌트 맞춤로 인해 *cBW*에서 발생하는 오류는 발견할 수 없기 때문이다. *cBW*에서의 이러한 예기치 못한 오류는 *cW*와 *B*의 상호작용으로 발생할 수 있다. 이런 오류를 테스트하기 위해 본 논문에서 제안한 테스트 기법이 적용된다.

■ 오류 삽입 대상 (*FIT*)

본 예제의 컴포넌트 맞춤은 컴포넌트 맞춤 의미 패턴 1, 즉 속성 변경 맞춤에 해당되고, 컴포넌트 맞춤 구문 패턴 1로 컴포넌트 맞춤을 수행하였으므로 본 예제의 *FIT*는 표 3에서 $FIT_{1,2}$이다. 그림 9 (a) *cW*의 setHiVal()에서 'value > hiVal'과 'hiVal'(그림 9 (a)에서 이탤릭체로 표시됨)은 각각 $FIT_{1,2}$의 WPC와 WPR부분이다. 본 예제에서는 이 가운데 우선 WPC부분에 오류를 삽입하여 테스트 케이스를 선정한다.

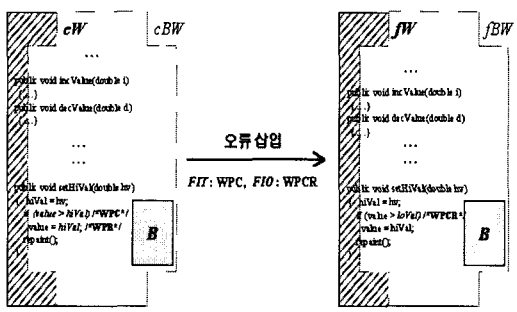


그림 9 NeedleGauge의 *cBW*와 *fBW*

■ 오류 삽입 연산자 (*FIO*)

본 예제에서는 그림 9와 같이 그림 9 (a)의 WPC부분에 WPCR을 *FIO*로 적용하여 테스트 케이스를 선정한다. WPC부분에 WPCR연산자를 적용한 결과 생성된 *fBW*는 그림 9 (b)이다. WPCR연산자에 의해 그림 9 (a)의 *cW*에서 WPC부분, 즉 'value > hiVal'이 그림 9 (b)의 *fW*에서 'value > loVal'로 대치되었다.

■테스트 케이스 ($x \in TC$)

정의 6에 따라 컴포넌트 맞춤 테스트 케이스는 $cBW(x) \neq fBW(x)$을 만족하는 x들의 집합이다. 본 예제에서 $cBW(x) \neq fBW(x)$를 만족하는 x="NeedleGauge(false, "Extensible Gauge", 0, 90, 10), incValue(80), decValue(10), incValue(50), setHival(100)"를 선정하면, $cBW(x)$는 100에 바늘이 놓여져 있고 $fBW(x)$는 바늘이 보이지 않게 되어, $cBW(x) \neq fBW(x)$를 만족하게 된다.

(4) 예제 수행 결과 분석

본 예제는 본 논문에서 제안한 기법이 갖는 특성을 다음과 같이 나타낸다.

그림 9 (a)의 WPC부분은 그림 8의 맞춤 코드에 속하지 않는 부분이다. 이는 *FIT*가 맞춤 코드이외의 곳에 존재할 수도 있음을 보인다. 이를 통해 본 논문에서 정의한 *FIT*들은 컴포넌트 맞춤 패턴에 따라 체계적으로 선정된 것으로서 단순히 맞춤 코드를 기준으로 선정된 것이 아니라는 것을 알 수 있다.

또한 본 예제는 본 기법이 단순히 *cW* 코드 전체에 오류를 삽입하는 수준을 넘어선 것임을 보인다. 본 예제의 *cW*의 맞춤 코드에는 오류가 없다. 그러나 (3)에서 선정된 테스트 케이스("NeedleGauge(false, "Extensible Gauge", 0, 90, 10), incValue(80), decValue(10), incValue(50), setHival(100)")를 *cBW*에 적용했을 때, 0부터 100까지를 표현하는 NeedleGauge컴포넌트에서 속성변수 *value*의 값이 범위를 벗어났으므로, *cBW*(*x*)에 대한 기대되는 결과는 NeedlGauge에 바늘이 보이지 않아야 한다. 그러나 실제 *cBW*(*x*)의 수행 결과로 100에 바늘이 나타나기 때문에 *cW*에 오류가 존재함을 알 수 있다. 이러한 오류는 *cW* 코드 전체에 오류를 삽입하여 선정된 테스트 케이스에 의해서도 발견될 수 있다. 그러나 이 경우 무수히 많은 *fBW*들을 생성해야 하며, 그에 대한 테스트 케이스도 무수하다. 이는 테스트 시간과 비용의 낭비이다. 그러나 본 기법은 컴포넌트 맞춤 오류를 위한 테스트 케이스를 효과적으로 선정하기 위해, 맞춤 패턴에 따른 *FIT*를 *cW*와 *B*사이의 상호작용 원리를 고려하여 추출하여 맞춤 오류에 적정한 테스트 케이스를 추출하기 위한 적절한 수의 *fBW*를 생성함으로써, 효율적인 컴포넌트 맞춤 테스트 기법을 제안하였다.

5. 분 석

본 논문의 기법은 2장에서 기술한대로 컴포넌트의 특성상 오류 삽입 기법을 사용한다. 이때 맞춤으로 변형된 *cW*의 모든 부분에 오류를 삽입한다면 컴포넌트 맞춤을 위한 오류 발견에 적정한 테스트 케이스를 효율적으로 추출할 수 없을 것이다. 즉, '적정한 테스트 케이스를 선정하려면 어디에 오류를 삽입해야 하는가'를 결정해야 한다. 이를 위해 본 기법은 *cW*가운데 특히 *B*와의 상호작용에 영향을 주는 부분을 *FIT*로 선정하고, 그곳에 오류를 삽입하여 테스트 케이스를 선정한다. 이렇게 함으로써, 선정된 테스트 케이스의 적정성을 높일 수 있다.

본 논문은 *B*와 *cW*의 상호작용을 체계적으로 분석하기 위해, 컴포넌트 맞춤에서 '무엇을' '어떻게' 변경시키는가에 대한 패턴을 추출하였다. '무엇을'에 대한 컴포넌트 맞춤 패턴은 '컴포넌트 맞춤 의미 패턴'으로 표현하였으며, '어떻게'에 대한 컴포넌트 맞춤 패턴은 '컴포넌트 맞춤 구문 패턴'으로 표현하였다. 이는 컴포넌트 맞춤과 설계패턴에 대한 지금까지의 문헌들을 기반으로 한다. 결국 본 기법의 중요한 요소인 *FIT*는 그림 6에 표현한 컴포넌트의 *B*와 *cW*사이의 상호작용을 중심으로 컴포넌트 맞춤 구문 패턴과 의미 패턴에 따라 선정되었다. 따라서 본 기법의 *FIT*는 컴포넌트 맞춤 오류를 위한 테스트 항목으로서 충분하다.

한 개의 *FIO* 연산자에 의해 한 개의 오류가 삽입된 *fBW*와 *cBW*를 차별화하는 테스트 케이스는 coupling effect[DeMillo78]에 따라 다양한 오류가 삽입된 *fBW*와 *cBW*도 차별화할 수 있다. 따라서 본 논문에서 정의한 기법은 컴포넌트 맞춤으로 인한 복잡한 오류에도 효과가 있음을 알 수 있다.

컴포넌트 맞춤 테스트 기법의 예제에서 *B*와 *W*가 하나의 클래스로 이루어진 *BW*를 예로 들었으나, *B*와 *W*가 여러 클래스들의 클러스터일 경우에도 본 논문의 기법은 적용 가능하다. *B*는 블랙박스이므로, 그 내부가 하나의 클래스이든 아니든 본 기법에서는 무관하다. 또한 *W*가 클래스들의 클러스터로 이루어졌을 경우, 컴포넌트 맞춤으로 인해 변경된 부분이 존재하는 클래스를 *cW*로 놓고, 본 논문의 기법을 적용할 수 있다. 본 논문에서 제시한 기법은 컴포넌트의 일반적인 특성과 컴포넌트 맞춤 패턴을 기반으로 하고 있으므로, 컴포넌트의 크기는 문제가 되지 않는다.

6. 결론 및 향후 연구 과제

본 논문은 다음의 특성을 갖는 컴포넌트 맞춤 테스트 기법을 제안하였다.

첫째, 제한된 정보만을 제공하는 컴포넌트를 대상으로 테스트를 수행하는 기법을 제시하였다. 컴포넌트를 대상으로 하는 테스트는 개발 산출물에서 테스트를 위한 정보를 추출하였던 기존의 방법들로는 해결할 수 없다. 만일 컴포넌트를 재사용할 때 컴포넌트의 개발 산출물들까지 획득되어진다면, 컴포넌트에 대한 정보들을 관리하기 위한 또 다른 비용이 요구될 것이다. 따라서 본 논문은 컴포넌트가 표현하는 최소한의 정보, 즉 인터페이스만을 이용하여 수행할 수 있는 테스트 기법을 제시하였다.

둘째, *B*와 *cW*사이의 상호작용 원리를 고려하여, 컴포넌트 맞춤 오류 발견 능력이 있는 테스트 케이스를 선정할 수 있도록 *FIT*를 선정하였다. *cW*의 모든 코드에 오류를 삽입하지 않고, *FIT*에만 오류를 삽입함으로써, 테스팅 시간을 줄이고 보다 적중률이 높은 테스트 케이스들을 추출할 수 있게된다.

향후, 본 기법의 컴포넌트 맞춤 오류를 발견하는 능력에 대한 실험을 수행하여 본 기법의 테스트 케이스가 갖는 컴포넌트 맞춤 오류에 대한 적정성을 평가할 계획이다. 이를 위해 본 기법의 *FIO*와 *FIT*를 적용하여 *fBW*와 그에 대한 테스트 케이스를 자동 생성하는 도구를 구현할 계획이다.

본 논문의 '컴포넌트 맞춤 테스트 기법'을 CBSD 각 단계에서 수행되는 다양한 수준의 테스트로 확장할 수 있다. CBSD의 기본적인 단계는 '컴포넌트 획득', '컴포넌트 맞춤', '컴포넌트 조립'으로 이루어지며, 개발 단계에 따라 다양한 수준의 테스트가 요구된다[2]. 본 논문은 이 가운데 컴포넌트 맞춤을 위한 테스트 기법을 제안하였고, 향후 본 기법을 컴포넌트 조립 테스트, 컴포넌트 시스템 테스트를 위한 기법으로 확장하여, CBSD 전 단계를 대상으로 하는 테스트 기법을 개발할 것이다.

참 고 문 헌

[1] J.-M.Jezequel and B.Meyer, "Design by Contract : The Lessons of Ariane," Computer, pp.129-130, Jan. 1997,

[2] 윤희진, 최병주, "컴포넌트 기반 소프트웨어 개발에서의 Testing," 소프트웨어공학회지, pp.68-74, 1999

[3] Paul Allen, "Practical Strategies for Migration to CBD," IT Journal Distributed Component Systems, 1999.

[4] Desmon F.D'Souza and A.C.Wills, *Object, Components, and Frameworks with UML*, Addison-Wesley, 1998

[5] Urs Hölzle, "Integrating Independently-Developed Components in Object-Oriented Languages," ECOOP'93 Proceedings, 1993

[6] Chen.H.Y., Tse.T.H., Chan.F.T. and Chen.T.Y., "In Black and White : An Integrated Approach to Class-Level Testing of Object-Oriented Programs," ACM Transactions on Software Engineering and Methodology, Vol. 7, No. 3, pp.250-295, July 1998

[7] Sudipto Ghosh, Aditya P. Mathur, Joseph R. Horgan, J. Jenny Li, W. Eric Wong, "Fault Injection Testing of Distributed Systems - A Case Study," Proceedings of Quality Week Europe, Nov. 1997

[8] R.A.DeMillo, R.J.Lipton, and F.G.Sayward, "Hints on Test Data Selection : Help for the Practicing Programmer," IEEE Computer, Vol.11, No.4, pp.34-41, Apr 1978

[9] Micheal Morrison, Randy Weems, Peter Coffee, and Jack Leong, *How to Program JavaBeans*, Ziff-Davis Press, 1997

[10] Wolfgang Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1994

윤 희 진

1993년 2월 이화여자대학교 전자계산학과 학사. 1998년 2월 이화여자대학교 대학원 컴퓨터학과 석사. 1998년 3월 ~ 현재 이화여자대학교 대학원 컴퓨터학과 박사과정. 관심분야는 소프트웨어공학, 분산 컴포넌트 테스팅, 테스트 프로세스, 객체지향 소프트웨어 테스팅.

최 병 주

1979년 ~ 1983년 이화여대 수학과 학사. 1986년 ~ 1988년 Purdue Univ. 전산석사. 1987년 ~ 1990년 Purdue Univ. 전산학(소프트웨어공학 전공) 박사. 1991년 ~ 1992년 삼성종합기술원 선임연구원. 1992년 ~ 1995년 용인대학교 전자계산학과 전임강사. 1995년 ~ 현재 이화여자대학교 전자계산학과 조교수. 관심분야는 소프트웨어공학, 소프트웨어 테스트, 소프트웨어 품질 측정, 테스트 케이스 적정성 측정 알고리즘임.