OCL로 기술된 객체지향 설계 명세의 테스트 케이스 생성

최 은 만†

요 약

소프트웨어의 오류에 대한 통계를 보면 구현단계보다 분석과 설계 단계에 더 중요하고 많은 오류가 유입된다. 따라서 소프트웨어의 분석 및 설계 작업의 결과인 디자인 모델이 요구에 맞게 설계되었는지, 또는 구조적으로 적합한지 잘 점검할 필요가 있다. 이 논문에서는 객체지향 설계 방법으로 사용되고 있는 UML로 표현된 설계 모델을 효과적으로 테스트할 수 있는 방법에 대해 논의하였다. UML 모델을 이루는 각 요소들에 대한 제약조건을 OCL(Object Constraint Language)로 기술하고 카테고리 분할 방법을 이용하여 UML모델을 테스트하기 위한 데이터를 생성하는 방법을 제안하였다. 생성된 테스트 데이터를 이용하여 객체지향으로 설계된 모델뿐만 아니라 개발된 시스템의 기능 테스트를 수행할 수 있다.

Generating Test Cases for Object-Oriented Design Specification

Eun Man Choi†

ABSTRACT

Statistics concerning software errors indicate that more errors are introduced in analysis and design phase than implementation phase. Therefore, it is needed to check whether the design modeling is appropriate for own function and structure. This paper discussed the effective test method for the object-oriented design model, i.e., UML. A new method was proposed for generating test data. This method consists of category partition theory by the representation each element in UML model with OCL (Object Constraint Language). Test data generated in this way can be used for testing the source code functionality as well as for checking the design model.

키워드 : UML, 소프트웨어 테스트(Software Test), OCL, 카테고리 분할(Category Partition)

1. 서 론

소프트웨어 제품이 복잡해지면서 항상 따라다니는 문제는 오류와 품질 문제이다. 규모가 커지고 기능이 복잡해지면 오류를 위한 계획적인 통제와 테스트 방법이 동원되어야 소프트웨어 품질 향상에 효과가 있다. 통계에 의하면 상품화된 소프트웨어의 경우라도 결함밀도가 2/KLOC 정도라면 좋은 품질로 간주하고 있으니 최근의 규모가 큰 소프트웨어들이 얼마나 많은 오류를 포함하고 있는지 예상할 수 있다[1].

객체지향 방법은 캡슐화, 다형성, 상속 등의 개념으로 복잡해지는 응용 문제를 잘 다룰 수 있다. 따라서 객체지향 프로그램의 경우 품질에 대한 좋은 방법이 제안되고 도입된다면 소프트웨어 개발에서 큰 과제인 복잡성과 품질 문제를 모두 해결하는 실마리가 되는 셈이다.

소프트웨어의 오류에 대한 통계를 보면 구현단계보다 분석과 설계 단계에 더 중요하고 많은 오류가 유입된다. 일반적으로 소프트웨어 테스트 작업은 명세에 대한 원시코드의 적합성만을 생각할 수 있으나 개발 전체 과정이 테스트 작업과 통합되어 조기에 오류를 찾아내는 방법이 필요하다. Extreme programming[2] 방법이 상당한 효과를 보고 있는 것도 이런 맥락에 기인한다. 테스트와 개발을 병행하면 조기에 오류를 찾아낼 수 있을 뿐만 아니라 계속되는 개발 과정에서 파생되는 오류를 방지할 수 있다. 분석 및 설계 모형을 테스트하는 방법이 이러한 목적으로 사용된다.

UML은 OMG 그룹에 의해 제안된 표준 객체지향 모델링 방법이다. UML에 의하여 표현된 설계 명세는 두 가지 수준의 테스트에 사용될 수 있다. 첫째는 원시코드가 개발된 후 시스템 수준의 기능 테스트에 사용된다는 것이다. 이 단계에서는 주로 UML의 use case들이 구현 시스템의 기능 시험에 사용된다[3]. 또 다른 수준은 설계 테스트이다. 구현되기 전 설계된 사항이 요구에 맞는지 확인하고 설계 상의 오류를 미리 찾아내려는 것이다.

† 정 회 원 : 동국대학교 컴퓨터공학과 교수
논문접수 : 2001년 4월 30일, 심사완료 : 2001년 10월 11일

UML로 표현된 설계 모델은 그래픽 표현이므로 테스트하기가 쉽지 않다. 또한 그래픽 표현에서 테스트 데이터를 찾아내는 작업 또한 쉽지 않다. 그러나 UML 안에는 그래픽 요소 외에 형식 언어인 OCL(Object Constraints Language)을 포함하고 있어 이를 이용하면 모델을 테스트하는 데이터를 구할 수 있다.

이 논문에서는 OCL로 표현된 클래스의 특성 및 제약조건을 이용하여 클래스 수준의 테스트와 클래스 사이의 협동(collaboration) 수준의 테스트에 대한 방법을 제시한다. 또한 주문 처리 시스템에서 제시한 방법으로 테스트 케이스를 생성할 수 있음을 보였다.

2. 연구 배경 및 관련 연구

UML의 중요도에 비하여 UML 기반 테스트에 이루어진 연구는 그다지 많지 않다. 크게 세 가지 연구 결과가 발표되었는데 상태 다이어그램에서 시험 데이터를 생성하는 방법을 연구한 것과 사용 사례에서 테스트 데이터를 추출하는 방법을 연구한 것, 클래스 다이어그램에서 테스트 데이터를 생성하는 방법을 연구한 것 등이 있다.

2.1 UML 상태 다이어그램에서의 테스트 사례 생성

Jeff Offutt에 의하여 소개된 이 방법은 UML 명세에서 시험 데이터를 생성하는 것으로 네 가지 수준의 기준이 있고 테스트 데이터를 자동으로 생성하는 도구를 구현하였다[4]. 상태 다이어그램도 하나의 그래프이므로 시작 상태부터 발생될 수 있는 상태 전환경로를 찾아낼 수 있다. 모든 상태 전환을 커버하는 수준(transition coverage)으로 테스트 데이터를 만들 수도 있고 모든 조건(predicate)을 만족하는 수준으로 높일 수도 있다. 또한 입력과 출력의 트랜지션의 쌍으로 그루핑하여 두 가지 조합을 시험하는 수준도 있다. 가장 완벽한 테스트 수준은 상태 다이어그램에 표현된 상태들의 순서조합을 점검해 보는 것이다. 즉 의미 있는 연속적인 상태 변환을 선택하여 검토하는 방법이다. 상태 기반으로 테스트 데이터를 생성하는 도구도 개발하였으나 이는 원시코드를 시험하기 위한 데이터이며 UML 설계를 테스트하는 기술과는 거리가 멀다. 이 방법의 중요한 결함은 클래스 수준의 테스트만을 수행할 수 있고 객체 사이의 인터랙션은 테스트할 수 없다는 점이다.

2.2 사용 사례로부터 테스트 사례 생성

Carpenter[5]는 사용 사례를 이용하여 분석 기반 테스트 케이스를 생성하는 방법을 제안하였다. 테스트 사례는 액터와 입력 이벤트, 예측된 출력 이벤트를 기초로 탐색하여 찾을 수 있다. 하지만 실제 분석 단계의 결과인 사용 사례만을 이용하여 찾아보면 자세한 정보, 예를 들면 관련된 객체 및 객체의 구체적인 상태 값들이 필요하다. 이러한 정보를 사용 사례에 추가하여 확장된 사용사례를 정의하고 이를 테스트에 이용하려는 시도가 있었다[3]. 그러나 여기서 확장된 대부분의 정보는 설계가 많이 진행된 후에 완성된 클래스 다이어그램이나 협동 다이어그램에서 찾을 수 있는 것이 대부분이다.

2.3 UML 클래스 다이어그램으로부터의 테스트 사례 생성

절차적 프로그램에 대하여 테스트 스크립트를 자동적으로 생성하려는 시도는 여러 차례의 연구[10-12]에서 있었다. 그러나 객체지향 프로그램에 대한 테스트 스크립트를 작성하려면 대부분의 정보는 클래스 다이어그램에서 얻어야 하며 이를 착안하여 자동 테스트를 위한 스크립트는 아니지만 AI 기법을 이용한 테스트를 위한 플래너에 입력을 찾는 기법을 연구한 것이 [6]이다. 즉 Scheetz[6]는 AI 기법을 이용한 테스트 플래너에게 특정한 클래스를 테스트하기 위한 목표를 UML로 표현하는 방법, 또는 UML에서 테스트 목표를 찾아내는 방법을 제안하였다. 위 연구 결과를 잘 발전시키면 클래스 다이어그램으로부터 시스템 테스트 데이터를 생성하는 방법을 찾아낼 수 있다. 클래스 다이어그램은 단일 클래스를 테스트하기 위한 정보도 가지고 있지만 클래스 사이의 관계도 표시되어 있어 시스템 수준의 테스트를 위한 정보도 가지고 있기 때문이다. 클래스 다이어그램의 정보를 이용하여 클래스의 인스턴스화, 특정 상태에 도달한 객체의 수를 알아낼 수 있고 테스트할 때 어떤 객체가 도달하여야 하는 상태를 정의할 수 있다. 즉 대부분의 연구 내용은 중요한 단일 클래스의 상태 변화를 점검하기 위한 것이며 시스템을 이루는 클래스 사이의 인터액션에 대한 테스트 사례를 만드는 것과는 거리가 있다.

객체지향 시스템의 중요한 핵심은 클래스와 클래스 사이의 상호작용에 대한 것이다. 즉 클래스 수준의 테스트뿐만 아니라 클래스 사이의 인터액션의 테스트가 필요하다.

이 논문에서는 클래스 수준의 테스트를 위하여 클래스의 속성을 카테고리 분할하고 OCL로 표현된 여러 조건을 이용하여 테스트 데이터를 생성하는 방법을 제안하였다.

3. 접근 방법

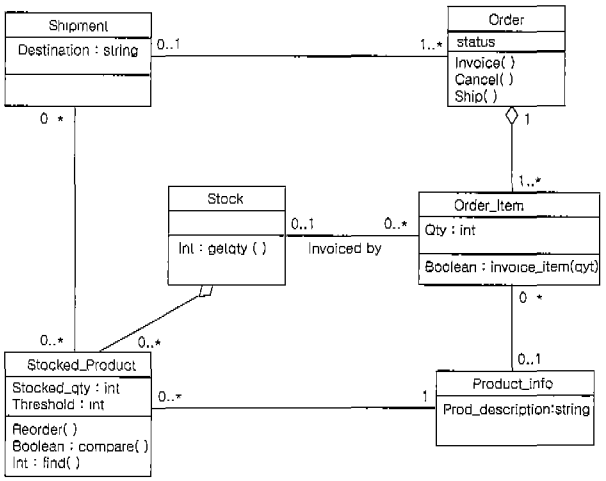
명세 기반의 테스트 방법으로 카테고리 분할 방법이 있다[7]. 이 방법은 기능 명세에서 출발하여 소프트웨어의 기능에 영향을 줄 만한 속성을 계속적으로 분할하여 찾아내는 기법이다. 즉 소프트웨어는 입력 값에 따라 분리된 다른 기능

이 수행되므로 기능에 영향을 주는 다른 입력 값들을 분할하여 찾아나간다면 명세 기반의 테스트가 가능하다는 것이다. 카테고리 분할 테스트 방법은 특히 객체지향 시스템에서 상속, 캡슐화, 초기치 설정등과 관련된 여러 가지 오류를 찾아내는데 효과적이다[8].

카테고리 분할 테스트는 테스트될 기능의 도메인을 분할하여 테스트 데이터를 선택하고 이를 이용하여 테스트하는 방법이다. 같은 카테고리에 속해 있는 데이터는 시스템의 동일한 기능을 커버하므로 대표하는 테스트 값 하나를 선택하면 된다.

클래스 다이어그램과 제한조건을 표현한 OCL을 이용하면 객체지향 시스템의 명세에서 테스트 데이터를 찾아낼 수 있다. 이를 위하여 클래스 안에 있는 속성들의 분할과 클래스 사이의 관계에서 다중도를 고려하여 분할하는 규칙이 필요하다.

연구 결과인 분할 규칙을 보이기 위하여 이 논문에서는 (그림 1)과 같은 주문 처리 시스템을 예로 채택하였다.



(그림 1) 주문처리 시스템의 클래스 다이어그램

OCL은 UML 모델의 여러 가지 요소들, 예를 들면 클래스, 관계, 속성 등의 제약사항을 정형적으로 표현하기 위한 언어이다. 오퍼레이션의 사전 조건과 종료 조건을 나타내는데 사용되기도 하며 UML 클래스 다이어그램의 네비게이션을 표현할 수도 있다. <표 1>은 UML의 여러 요소를 OCL로 표현한 예이다.

객체지향 테스트를 위한 카테고리 분할은 객체의 다중도와 관계 및 속성의 다중도에 의하여 결정된다. 예를 들어 주문 처리 시스템을 테스트하는 단계에서 주문의 발생, 인보이스, 취소, 배달 등의 오퍼레이션 시험이 중요하다. 시스템 수준의 테스트이므로 주문을 하나 내는 경우만 아니라 2개, 3개 등 다수의 주문을 내는 경우 성능을 체크할 필요가 있다. 이때 객체의 다중도에 따라 카테고리를 분할하는 기준은 최소 개수, 최대 개수, 중간 개수로 나누어 테스트 데이터를 만들 수 있다. (그림 1)의 예제에 대하여 객체의 다중도에 따라 카테고리 분할하면 <표 2>와 같이 된다.

〈표 1〉 OCL로 표현한 사례

표현 요소	OCL	설 명
클래스의 불변 조건(invariant)	Order_Item self.qty > 0 Order.status = enum{pending, suspend, invoiced, shipped, canceled}	주문 아이템의 개수는 0보다 큰 수이어야 함. 주문의 상태는 Enumeration 타입
오퍼레이션의 사전, 종료 조건	Stock : : getqty( ) pre : result : if exist sp = stocked_product.find(order_item) ! = NULL result = s.stocked_qty else result = 0 Order : : invoice( ) pre : Order.ststus = pending post : (order_item → forall(invoice_item ( ))) implies status = invoiced (order_item → exist invoice_item( ) = false) implies status = suspend	제고 수량을 찾기 위하여 product 객체에 주문 아이템이 있는지 찾아보고 있으면 그 수량을 조회 주문한 모든 상품이 invoice 되었으면 상태를 변환
네비게이션	Person self.house → includes asSet (self.ortgage.house)	Person이 소유한 집이 mortgage 관계로부터 네비게이션 될 수 있음
인터액션 다이어그램의 메시지 조건	Copy : : checkout [getfine( ) = 0] checkout	도서 대출 연체료가 없는 경우에만 대출 가능

〈표 2〉 객체의 카테고리 분할

Class	LB	LB+1, …, HB−1	HB
Order	1	2, …, 14	15
Stock	1		
Shipment	0	1, …, 4	5

카테고리 분할에는 관계의 다중도도 고려하여야 한다. 주로 일반적인 연관관계(association)와 전체-부분 관계(composition)에 적용된다. 예를 들어 (그림 1)에서 Order와 Order_Item의 관계는 1대 n 의 전체-부분 관계이다. 따라서 Order_Item 객체의 최대 인스턴스가 n 개일 때 {1}, {2..n-1}, {n}으로 분할된다. (그림 1)에 표현된 연관 관계를 다중도를 고려하여 분할하면 <표 3>과 같이 된다.

객체의 속성을 규정하는 방법은 더 복잡하다. 즉 객체의 상태는 속성의 값에 의하여 좌우되며 객체의 상태에 대한 여러 제약조건을 OCL로 나타낼 수 있다. 일반적으로 객체의 속성은 다음 네 가지로 구별될 수 있으며 각 경우마다 적용되는 카테고리 분할 규칙이 다르다.

〈표 3〉 관계의 카테고리 분할

관 계	LB	LB+1, …, HB-1	HB
Aggregation			
Order Order_Item	1 1	2, …, 19	20
Association			
Stock Order_Item	0 0	1, …, 49	50
Stocked_Product Product_Information	0 1		1
Product_Information Order_Item	0 0	1, …, 19	1 20
Shipment Order	0 1	2, …, 49	1 50
Stocked_Product Shipment	0 0	1, …, 19 1, …, 49	20 50

<규칙 1> 속성이 일정한 값만 갖는다면 초기값의 조건으로 분할한다.

규칙을 적용한 예를 들면 <표 4>와 같이 단일 속성값에 대하여 분할한다.

<규칙 2> OCL에 표현된 제약조건이 하나의 속성에만 의존하는 경우는 각 속성의 제약조건에 따라 분할한다.

예를 들어 stocked_qty ≥ 0이면 {stocked_qty = 0}, {0 < stocked_qty < HB-1}, {stocked_qty = HB}로 분할한다.

<규칙 3> 하나의 클래스 안에 여러 개의 속성이 OCL에 포함된 경우는 하나의 속성에 의하여 분할된 테스트 사례가 다른 속성에 의하여 재분할한다.

예를 들어 다음과 같은 제약 조건이 OCL로 표현되었다면

```
context Sotcked_Product : : reorder( )
pre : stocked_qty < threhold
```

이 경우 원래의 카테고리 분할이 오버랩되므로 다음과 같이 재분할된다.

{stocked_qty=0}, {0 < stocked_qty ≤ 10}, {stocked_qty=10}, {10 ≤ stocked_qty ≤ 100}, {stocked_qty=100}

결국 reorder( ) 함수를 테스트하기 위하여 필요한 분할은 <표 5>와 같이 작성된다.

〈표 5〉 reorder( ) 함수 테스트를 위한 분할

	stocked_qty의 분할				
threshold	{0}	{1, …, 9}	{10}	{11, …, 99}	{100}
{10}	true	true	false	false	false

<규칙 4> 여러 클래스에 속한 여러 개의 속성이 OCL에 포함된 경우는 복잡한 제약조건을 단순하게 하고 분할한다.

예를 들어 주문처리 시스템에서 주문 상품이 stocked_qty에 충분히 있으면 invoice_item이 된다는 제약조건은 다음과 같이 OCL로 표현된다.

$P_{invoice\_item}$ 1 :
exist sp = stocked_product and
sp.product_info = order_item.product_info,
and sp.stocked_qty ≥ itself.qty, result = True

이렇게 되면 stocked_qty와 각 Product의 qty는 다음과 같이 다시 분할되어야 한다.

P_{qty} 1 : {0}, P_{qty} 2 : {1,…,9}, P_{qty} 3 : {10}, P_{qty} 4 : {11, …, 19},
P_{qty} 5 : {20},
$P_{stocked\_qty}$ 1 : {0}, $P_{stocked\_qty}$ 2 : {1, …, 9},
$P_{stocked\_qty}$ 3 : {10}, $P_{stocked\_qty}$ 4 : {11, …, 19},
$P_{stocked\_qty}$ 5 : {20}, $P_{stocked\_qty}$ 6 : {21, …, 99},
$P_{stocked\_qty}$ 7 : {100}

4. 클래스 테스트

클래스 수준의 UML 테스트를 위하여 다음 두 가지 측면의 점검이 필요하다. 먼저 구조적 측면이다. 객체와 객체 사이의 관계, 예를 들면 일반적인 연관관계(association)와 집합연관(aggregation) 관계들에 대한 점검이 카테고리 분할에 고려되어야 한다. 또 다른 측면은 소프트웨어의 기능에 초점을 두어 테스트될 기능과 관련 있는 클래스만을 찾아내어 카테고리 분할 방법으로 점검하는 것이다.

〈표 4〉 단일 속성에 대한 카테고리 분할

속 성 이 름	OCL	카테고리 분할
Order_Item.qty	qty ≥ 0	{qty = 0},{0 < qty < 20},{qty = 20}
Stocked_Product.stocked_qty	stocked_qty ≥ 0	{stocked_qty = 0}, {0 < stocked_qty < 100}, {stocked_qty = 100}
Stocked_Product.threhold	threshold = 10	{threshold = 10}
Order.status	status : enum{pending, suspend, invoiced, shipped, candeled}	{pending}, {suspended}, {invoiced}, {shipped}, {cenceled}

4.1 클래스의 구조 점검

객체지향 방법은 모듈화가 잘 되어 있는 구조를 가지고 있다. 클래스가 하나의 빌딩 블록이며 빌딩 블록들을 여러 연관관계로 엮어 하나의 시스템을 이룬다. 이렇게 여러 가지 연관관계로 엮어진 것이 과연 잘 된 것인지 확인하는 과정은 모델 차원에서 이루어져야 하며 다음과 같은 카테고리 분할 규칙이 적용되어야 한다.

<규칙 5> 포함관계의 다중도 분할 규칙 : 일반적인 포함관계는 1대 n의 다중도를 갖는다. 클래스 B가 클래스 A를 이루는 부품이 될 때 포함관계를 점검하려면 객체 A가 하나의 객체 B와 연관되는 경우, 두 개 이상 N_B-1개의 객체 B와 연관되는 경우, 그리고 N_B개의 객체 B와 연관되는 경우 세 가지로 분할한다.

<규칙 6> 일반 연관관계의 다중도 분할 규칙 : 클래스 A와 클래스 B가 n : m의 연관관계가 있다면 A의 다중도에 따라 {1}, {2, …, n−1}, {n}, 클래스 B의 다중도에 따라 {0}, {1, …, m−1}, {m}으로 분할한다. 결국 A와 B를 고려한 분할은 (1 : 0), (1 : 3), (1 : m), (2 : 0), (2 : 3), (2 : m), (n : 0), (n : 3), (n : m) 등 9개의 다중도 관계를 점검하도록 분할되어야 한다.

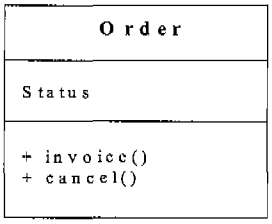
<규칙 7> 두 개의 속성과 관련된 OCL 규칙을 위한 다중도 분할 규칙 : 두 개 클래스의 속성이 참여하여 OCL로 표현된 규칙이 있을 때 만일 그 OCL 규칙이 진위값으로 표현될 수 있는 것이라면 다음과 같이 분할할 수 있다. 즉 attribute 1이 $\{A_1\}$에서 $\{A_n\}$까지 분할되고 attribute 2가 $\{B_1\}$에서 $\{B_m\}$까지 분할되었을 때 두 속성의 카디션 곱의 모든 경우에 대하여 OCL 규칙이 참 또는 거짓이 되는지 알아보고 참과 거짓 각각의 경우를 카테고리 분할한다.

4.2 기능 테스트

클래스 구조 점검은 기능 테스트를 하기 위한 기초 단계이다. 객체지향 시스템의 기능을 테스트하려면 클래스 멤버함수의 호출이 어떤 순서로 발생하며 순서 있는 호출 속에서 어떤 자료가 전달되는지 확인하여야 하므로 클래스 구조가 기능 호출을 위한 이벤트 발생의 전제 조건이 된다.

기능 테스트를 위한 카테고리 분할 방법은 다음과 같은 규칙으로 정의된다.

<규칙 8> 각 함수에 대한 사전 조건(pre-condition)이 카테고리 분할되어야 한다.



(그림 2) Order 클래스의 Cancel 함수

예를 들어 (그림 2)에 있는 order 클래스의 cancel( ) 함수를 테스트하려면 함수를 위한 OCL 정의에서 사전 조건에 해당되는 status 속성을 각각 pending, invoiced, suspend로 분할하여야 한다.

```
context Order : : cancel
  pre : status = pending, or status=invoiced
        or status = suspend
```

<규칙 9> 함수 호출의 각 경로가 적어도 한 번씩 커버되어야 한다.

객체지향 시스템은 객체들이 서로 메시지를 호출하면서 자료를 주고받는다. 이런 과정은 OCL에서도 다음과 같이 잘 표현된다.

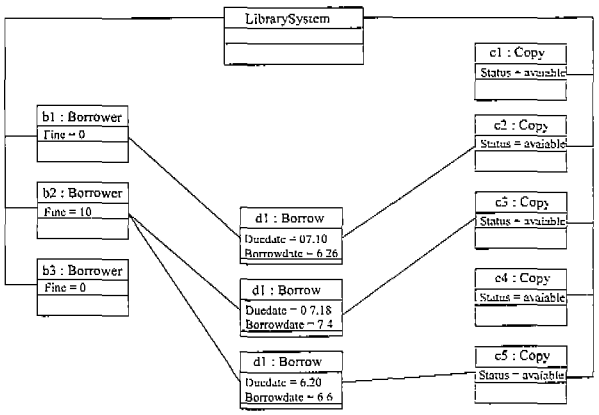
```
context A : func1
post : if state1, B : func2 and result = result1
       else B.func3 and result = result2
context B : func2
post : if state3, result = result3
       else result = result4
context B : func3
post : result = result5
```

위의 예에서 func1과 func2는 분기점을 가지고 있다. 두 개의 조건문이 이루는 세 가지 경로, 즉 state1과 state3, state1과 not state3, not state1에 대한 점검이 필요하다.

<규칙 10> 함수 안에 있는 각 이벤트는 적어도 한 번씩 수행되어야 한다.

이벤트는 함수의 호출이 될 수도 있고 단순한 계산이 될 수도 있다. 위의 예에서 func1, func2, result = result1, func3, result = result2, result = result3, result = result4, result = result5는 모두 이벤트이다. 각 이벤트는 서로 다른 경로에 속할 수 있다.

결국 기능 테스트는 다음과 같이 이루어진다. (그림 3)에 표시한 도서관 시스템의 checkout 기능은 다음과 같은 입력



(그림 3) 도서관 시스템의 객체 다이어그램

기준으로 만족하는지 테스트하여야 한다. 먼저 객체의 구조를 고려하여 카테고리 분할하면 다음과 같이 된다.

```
checkout(b1, c1)
checkout(b2, c6)
checkout(b4, c2)
checkout(b4, c6)
```

한편 함수 호출에는 다음과 같은 경로가 존재한다.

- 대출자가 연체료가 없고 빌리려는 책이 있는 경우 대출이 될 수 있음
- 대출자가 연체료가 있고 빌리려는 책이 없는 경우 대출이 될 수 없음
- 대출자가 연체료가 없고 빌리려는 책도 없는 경우 대출이 될 수 없음

Checkout 함수에 Bottower.getfine, return, Copy.available, return, Copy.checkout, Create new Borrow, data(today, today+14)가 존재한다면 다음과 같은 경로와 이벤트가 테스트에서 점검되어야 한다.

- checkout(b1, c1)은 b1.getfine, return 0, c1.available, return True, c1.checkout, create new Borrow, date(todat, today+14), return이 경로로 커버됨
- checkout(b2, c1)은 b2.getfine, return 10, return False가 커버됨
- checkout(b1, c2)는 b1.getfine, return 0, c2.available. return False를 커버함
- checkout(b2, c2)는 b2.getfine, return 10, return False를 커버함

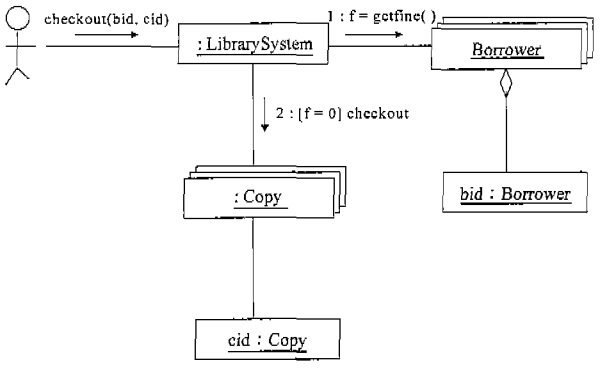
5. 협동 테스트

협동(collaboration) 다이어그램은 사용자로부터 추출된 요구를 기술한 사용 사례에 대하여 더 자세한 내부 수준, 즉 사용 사례를 만족시키기 위한 내부 객체들의 상호 작용에 대하여 작성한 명세이다. 따라서 시스템 수준의 블랙 박스 형태의 테스트를 위하여 사용 사례가 사용될 수 있다. 반면 협동 다이어그램은 시스템 수준의 그레이 박스 테스트에 사용될 수 있다. 그레이 박스 테스트는 원시 코드를 기초로 테스트하는 화이트 박스와는 다르다. 클래스 내부 코드를 참조하여 문장 단위 커버리지(statement coverage)나 조건 커버리지(condition coverage)와 같은 테스트 기준을 적용하지 않고 협동 다이어그램에 표현된 명세 정보를 이용하여 객체 사이의 메시지 전달에 초점을 둔다.

협동 다이어그램을 이용하면 그레이 박스 테스트를 위한 명세를 작성할 수 있다. 테스트 명세는 프리픽스, 입력, 예상 출력, 세 부분으로 구성된다. 프리픽스는 시스템의 초기 상태를 나타내는 조건이며 입력은 입력 파라미터의 값, 예상 출력은 시스템의 종료 상태와 출력 값을 의미한다.

5.1 프리픽스의 생성

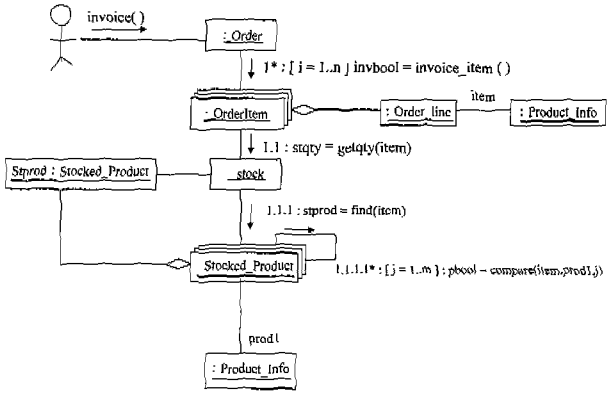
프리픽스는 시스템의 초기 상태를 나타내는 것으로 협동 다이어그램을 테스트하는 동안 인스턴스화되어야 하는 클래스와 그들 사이의 관계를 말한다. 다시 말해 4장에서 설명한 객체 사이의 관계가 특정 협력 다이어그램을 테스트하는 동안 지속적으로 만족시켜야 하는 객체 및 객체 관계의 조건이다. 프리픽스를 생성하는 단계는 다음과 같다.



(그림 4) checkout을 위한 협동 다이어그램

1) 협동 다이어그램에 있는 조건에 대한 OCL 표현을 찾아낸다. (그림 4)에 있는 조건 2 : [f=0] checkout 은 바로 전 메시지 패싱 1 : f = getfine( )의 영향을 받으므로 최종적으로 구한 조건을 OCL로 표현하면 Borrow : [getfine = 0] checkout 이 된다. OCL로 표현된 조건을 기초로 카테고리 분할하면 {getfine = 0}, {getfine > 0}, {getfine < 0}가 된다.

2) 다음은 협동 다이어그램에 참여하는 클래스를 찾아낸다. 협동 다이어그램 테스트는 참여하는 객체들의 상호 메시지 교환을 테스트하는 것이므로 참여 객체를 찾고 이를 카테고리 분할한다. 단 사용자 인터페이스를 담당하는 시스템 수준의 경계 객체는 카테고리 분할할 대상이 아니므로 제외한다. (그림 4)의 예에서는 Copy와 Borrower가 된다.
3) 협동 다이어그램을 기초로 테스트에 필요한 객체의 수를 찾아낸다.
(그림 5)에 표시한 협동 관계를 테스트하기 위하여 필요한 객체는 Order 한 개와 여러 개의 Order_Item 객체이다. Order와 Order_Item의 포함관계를 고려하여 카테고리 분할한다.
4) 협동 다이어그램에 있는 메시지 교환을 체크하여 관련된 OCL을 찾아낸다. 먼저 (그림 5)의 예에서 관련된 오퍼레이션을 찾아보면 Order::invoice, Order_Item::invoice _item, Stock::getqty, Stocked_Product::find, Stocked _Product::compare이다. 다음은 각 오퍼레이션이 수행되기 위한 전제조건을 찾는다. 예를 들어 Order::invoice를 위한 전제조건은 status = pending이다. 또한 Order::invoice 오퍼레이션을 테스트하려면 Order 안에 존재하는 모든 Order_Item이 Order_Item::invoice _item 함수를 만족하여 true가 되거나 하나 이상의 Order_Item이 만족하지 않아 false 되는 것을 체크하면 된다. qty나 stock-qty 등의 변수에 대한 구체적인 카테고리 분할 값은 6장 실험 부분에 첨부하였다.



(그림 5) invoice를 위한 협동 다이어그램

5.2 입력의 생성

테스트를 위한 입력은 협동 다이어그램에서 최초로 발생되는 메시지 교환에 필요한 파라미터이다. 즉 (그림 4)의 경우 bid, cid이며 (그림 5)의 경우는 입력이 필요하지 않다. 입력 값을 정하기 위한 카테고리 분할 방법은 4장에서 언급한 규칙이 적용된다.

5.3 예상되는 출력의 생성

협력 다이어그램으로 표현된 사용 사례에 대하여 테스트가 실행된 후의 예상 결과는 요구 추출 작업에 의하여 작성된 사용 사례를 이용하여 찾아낼 수 있다.

6. 실 험

제안한 방법에 따라 적용해 본 문제는 (그림 1)에 표시한 주문처리 시스템이다. 카테고리 분할을 위하여 작성한 OCL과 테스트를 위하여 필요한 각 객체의 개수는 다음과 같다.

6.1 테스트 객체의 분할

먼저 테스트할 대상 객체들에 대하여 카테고리 분할하면 다음과 같다. LB의 값은 OCL에 의하여 표현된 각 객체의 조건에서 추출한 것이며 HB는 연관관계의 다중도를 고려하여 적당히 선택한 것이다.

Class	LB	LB+1, …, HB-1	HB
Order	1	2, …, 14	15
Stock	1		
Shipment	0	1, …, 4	5

6.2 각 클래스를 위한 OCL과 카테고리 분할

주문처리 시스템 예에서 Order와 Order_Item 객체에 대한 제약조건들을 OCL로 작성하면 다음과 같이 된다.

6.2.1 Order

Order 객체의 상태를 변환시키는 중요한 멤버 데이터는 주문의 상태를 나타내는 status이며 invoice( )라는 멤버함수에 의하여 변환되는 status의 상태를 OCL로 나타내면 다음과 같다.

① *attribute* :

i. status : enum{pending, suspend, invoiced, shipped, canceled}

② *functions* :

ii. Order::invoice( )
A. pre : status = pending
B. post : order_item → forall(invoice_item( ))
implies status = invoiced
C. post : order_item → exist invoice_item( )
= false implies status = suspend

OCL로 나타낸 제약조건은 이런 의미이다. 어떤 주문이 처리되려면 모든 아이템이 인보이스 처리가 되어야 한다. 만일

주문한 물건이 없어 하나라도 인보이싱 되지 않는다면 보류 상태가 된다.

주어진 OCL 제약조건을 만족하는 카테고리 분할을 적용하면 다음과 같이 된다.

③ *partition* :

주문(order) 객체에 대한 status 속성을 분할하면 다섯 가지 카테고리로 나뉜다.

- partition on i(주문 객체에 대한 분할) :

  $P_{status}1$: status = pending
  $P_{status}2$: status = suspend
  $P_{status}3$: status = shipped
  $P_{status}4$: status = invoiced
  $P_{status}5$: status = canceled

주문처리를 위한 멤버 함수를 카테고리 분할하면 다음 세 가지(모두 주문처리 된 경우, 주문처리 되지 않은 아이템이 있는 경우, 모두 주문처리 되지 않은 경우)로 나눌 수 있다.

- partition on i and ii(주문 객체 및 주문처리에 대한 분할) :

  $P_{invoice}1$: order_item → forall(invoice_item)
  $P_{invoice}2$: order_item → exist(not invoice_item)
  $P_{invoice}3$: order_item → forall(not invoice_item)

다음은 주문 객체의 status 속성에 invoice( ) 함수가 적용되었을 때 카테고리 분할에 대한 결과이다. 주문 객체의 다섯 가지 상태에 invoice( ) 함수의 카테고리 분할을 적용할 때 결과는 다음과 같다.

④ *True and false table*

- for ii.B

	$P_{status}1$	$P_{status}2$	$P_{status}3$	$P_{status}4$	$P_{status}5$
$P_{invoice}1$	T	F	F	F	F
$P_{invoice}2$	F	F	F	F	F
$P_{invoice}3$	F	F	F	F	F

주문이 모두 보류(pending) 상태($P_{status}1$)이고 전제조건($P_{invoice}1$)이 만족되면 invoice( )가 적용될 수 있다.

- for ii.C

	$P_{status}1$	$P_{status}2$	$P_{status}3$	$P_{status}4$	$P_{status}5$
$P_{invoice}1$	F	F	F	F	F
$P_{invoice}2$	T	F	F	F	F
$P_{invoice}3$	T	F	F	F	F

주문이 모두 보류(pending) 상태($P_{status}1$)에서 주문 아이템이 하나라도 준비되지 않거나 하나도 준비되지 않는다면 주문처리 후에 상태는 보류(suspend)가 된다.

6.2.2 Order_Item

다음은 하나의 주문 안에 여러 개의 주문 아이템에 대하여 카테고리 분할한다. Order_Item 객체의 상태를 좌우하는 중요한 속성은 수량(qty)이며 양의 정수 타입이다. 또한 주문 아이템이 주문처리되기 위한 전제조건은 재고 수량(stock.getqty)이 주문 수량을 만족할 만큼 충분하여야 한다. 이것을 OCL로 나타내면 다음과 같다.

① *attribute* :

i . qty ≥ 0

② *functions* :

ii. invoice_item = (stock.getqty ≥ itself.qty)
iii. asSet(stock.stocked_product) → includes asSet(product_info.stocked_product)

OCL로 나타낸 조건을 이용하여 주문 아이템에 대한 카테고리 분할을 하면 다음과 같이 된다.

③ *partition* :

- partition on i(주문 아이템의 속성) :

  $P_{qty}1$: {0}
  $P_{qty}2$: {1, …, 19}
  $P_{qty}3$: {20}

주문 아이템에 대하여 카테고리 분할 LB, LB < 주문아이템 < UB, UB을 적용한 것이다.

- partition on ii(재고량이 충분한 경우) :

  - $P_{invoice\_item}1$:
    exist sp = stocked_product.find(order_item) ≠ NULL, and sp.stocked_qty ≥ itself.qty
    exist sp = stocked_product and sp.product_info = order_item.product_info, sp.stocked_qty ≥ itself.qty

주문 안에 포함된 하나의 주문 아이템에 대하여 재고가 충분히 있고 그 아이템에 대한 정보가 상품 리스트에 포함된 경우를 OCL로 표현하였다. 같은 방법으로 재고량이 부족한 경우($P_{invoice\_item}2$), 주문 상품 아이템을 재고 상품 리스트에서 찾을 수 없는 경우($P_{invoice\_item}3$), 아이템도 없고 주문 수량도 0인 경우($P_{invoice\_item}4$)를 OCL로 표현할 수 있다.

상품 아이템에 대한 주문 수량과 재고 상품 수량을 카테고리 분할하면 다음과 같다.

- $P_{invoice\_item}1$, $P_{invoice\_item}2$
  exist s = stocked_product and s.product_
  info = order_item.product_info
  result = s.stocked_qty and itself.qty
  * $P_{qty}1$: {0}
  * $P_{qty}2$: {1, …, 9}
  * $P_{qty}3$: {10}
  * $P_{qty}4$: {11, …, 19}
  * $P_{qty}5$: {20}
  * $P_{getqty}1 = P_{stocked\_qty}1$: {0}
  * $P_{getqty}2 = P_{stocked\_qty}2$: {1, …, 9}
  * $P_{getqty}3 = P_{stocked\_qty}3$: {10}
  * $P_{getqty}4 = P_{stocked\_qty}4$: {11, …, 19}
  * $P_{getqty}5 = P_{stocked\_qty}5$: {20}
  * $P_{getqty}6 = P_{stocked\_qty}6$: {21, …, 99}
  * $P_{getqty}7 = P_{stocked\_qty}7$: {100}

이제 다섯 가지로 분할된 주문 아이템 테스트 사례에 대하여 재고 상품 카테고리 분할을 적용해 보면 다음 진위표로 표현할 수 있다.

④ *True and false table*

- for $P_{invoice\_item}1$ and $P_{invoice\_item}2$
  exist s=stocked_product and s.product_info
  =order_item.product_info
  result=s.stocked_qty≥itself.qty

	$P_{qty}1$	$P_{qty}2$	$P_{qty}3$	$P_{qty}4$	$P_{qty}5$
$P_{stocked\_qty}1$	T	F	F	F	F
$P_{stocked\_qty}2$	T	T|F	F	F	F
$P_{stocked\_qty}3$	T	T	T	F	F
$P_{stocked\_qty}4$	T	T	T	T|F	F
$P_{stocked\_qty}5$	T	T	T	T	T
$P_{stocked\_qty}6$	T	T	T	T	T
$P_{stocked\_qty}7$	T	T	T	T	T

7. 결 론

소프트웨어를 이루는 모든 구성요소에 대하여 모든 가능성을 다 시험해 본다는 것은 현실적으로 매우 어려운 일이다. 변수 입력의 조합과 순서가 너무나 많은 경우의 수를 만들어 내며 이를 다 시험해 본다는 것은 불가능하기 때문이다.

여러 가지 소프트웨어 시험 방법 중에 블랙 박스 테스트는 비교적 경제성이 있는 테스트 방법으로 명세에 포함된 기능이 프로그램에 적합하게 구현되었는지를 카테고리 분할 방법으로 생성된 테스트 입력을 사용하여 확인한다. 이 논문에서는 카테고리 분할 방법을 적용하여 UML로 작성된 객체지향 설계를 테스트하는 방법을 제안하였다.

카테고리 분할 방법은 테스트할 소프트웨어를 기능적인 단위로 나누어 기능적 단위마다 테스트 입력을 분할하여 기능을 시험할 수 있다[9]. 카테고리 분할에 의하여 동치 클래스로 분류된 것은 동일한 기능에 대한 시험으로 같은 테스트 목적을 가지고 있다.

이 연구에서도 제시하였던 것처럼 카테고리 분할 방법의 중요한 장점 중의 하나는 소프트웨어의 기능을 정형적으로 표현한 명세에서 쉽게 구할 수 있다는 것이다. UML의 정형적 언어인 OCL을 사용하여 객체지향 설계를 이루는 객체, 객체의 오퍼레이션, 객체 사이의 인터액션 등 여러 요소를 표현하고 이를 기초로 테스트 입력 데이터를 구하는 방법을 제시하였다.

카테고리 분할 방법은 테스트 케이스를 찾아내고 테스트 입력값을 자동 생성해 줄 수 있지만 테스트 입력값과 함께 테스트 오러클을 이루는 예상되는 올바른 테스트 결과를 찾아주지는 않는다. 따라서 향후 연구로는 카테고리 분할에 의하여 생성된 테스트 케이스에 대하여 예상되는 결과값을 해당 OCL 부분을 보고 쉽게 찾을 수 있는 방법을 찾는 것이다.

참 고 문 헌

[1] David Chappell, "The Next Wave, Component Software Enters The Mainstream," http : //www.rational.com /product/whitepaper, 1997.

[2] K. Beck, Extreme Programming, Addison Wesley, 2000.

[3] E. Choi, A. von Mayrhauser, "Testing Object-Oriented Systems Using Extended Use-Cases," Proceedings of PDPTA 2000, Las Vegas, CSREA Press, pp.831-838, 2000.

[4] J. Offutt, A. Abdurazik, "Generating tests from UML specifications," UML '99 - The Unified Modeling Language Beyond the Standard, pp.416-430, 1999.

[5] P. B. Carpenter, "Validation of Requirements for Safety-Critical Software," Proceedings of the ACM SigAda Annual International Conference, pp.23-29, 1999.

[6] M. Scheetz, A. von Mayrhauser, R. France, "Getting Test Cases from an OO Model with an AI Planning System," Proceedings of 10th International Symposium on Software Reliability Engineering, pp.250-259, 1999.

[7] T. J. Ostrrand, M. J. Balceru, "The Category-Partition Method for Specifying and Generating Functional Tests," Communications of the ACM, Vol.31, No.6, pp.676-686, 1988.

[8] J. C. Offutt, A. Irvine, "Testing Object-Oriented Software using the Category-Partition Method," Proceeding of TOOLS USA'95 conference, 1995.

[9] E. Weyuker, B. Jeng, "Analyzing Partition Testing Strate-

gies," IEEE Transactions on Software Engineering. Vol.17, No.7, pp.703-711, 1991.

[10] D. C. Ince, "The Automatic Generation of Test Data," Computer Journal, Vol.30(1), pp.63-69. 1987.

[11] P. Maurer, "Generating Test Data with Enhanced Context Free Grammars," IEEE Software, July, pp.50-55, 1990.

[12] D. J. Kasik, H. G. George, "Toward Automatic Generation of User Test Scripts," Proceedings of the Conference on Human Factors in Computer Systems : Common Ground, ACM Press, pp.244-251, 1996.

최 은 만

e-mail : emchoi@dgu.ac.kr
1982년 동국대학교 전산학과(학사)
1985년 한국과학기술원 전산학과(공학석사)
1993년 일리노이 공대 전산학과(공학박사)
1985년~1988년 한국표준연구소 연구원
1988년~1989년 데이콤 주임연구원
1993년~현재 동국대학교 컴퓨터공학과 부교수
2000년~2001년 콜로라도 주립대 전산학과 교환교수
관심분야 : 객체지향 테스트, Program Understanding, 소프트웨어 품질 메트릭, 웹 기반 소프트웨어 테스트