제 10 장 명세 기반 테스트

10.1 개 요

블랙박스 테스트라고도 하는 명세 기반 테스트(Specification-based test)는 프로그램의 내부 논리 구조를 참조하지 않고 사용자의 요구사항이 기술된 명세나 설계 정보 등을 이용하여 테스트 케이스를 개발한다. 명세 기반 테스트는 대상 시스템의 명세 정보를 얻을 수 있는 한, 적용 대상에 제한이 없으며 컴포넌트 테스트, 통합 테스트, 시스템 테스트 및 인수 테스트 전 과정에 걸쳐 사용될 수 있다(그림 10.1 참조).

그림 10.1 명세 기반 테스트

프로그램 코드 내부 구조를 자세하게 아는 개발자보다는 프로그램 코드 내부 구조를 전혀 모르는 사람이 명세 기반 테스트를 수행하는 것이 좋을 수도 있다. 그 이유는 아무래도 개발자는 코드가 잘 동작되는 면만을 부각하는 테스트 케이스를 작성하는 경향이 있기 때문이다. 따라서 외부의 독립적인 테스터가 명세 기반 테스트를 수행하는 것이 좋다. 물론 외부 테스터라 할지라도 프로그램 기능과 도메인에 관한 이해가 충분해야 한다.

그러나 예외도 있다. 모든 명세 기반 테스트가 외부 테스터에게 맡겨지는 것은 아니다. 개발자 자신이 명세 기반 테스트를 실행할 때도 있다. XP(eXtreme Programming)의 테스트 주도 개발에서는 개발자가 먼저 테스트 케이스를 작성한 후에 코드를 구현한다. XP에서 새롭게 작성되는 테스트 케이스들은 기존 코드가 고려하지 않은 경우들을 대상으로 하기 때문에 편향적인 테스트 케이스 개발을 막을 수 있다. 그뿐만 아니라 짝 프로그래밍을

통해 테스트 케이스 개발과정을 개발자 자신 이외에 다른 작업자가 지켜보기 때문에 외부 테스터가 테스트 케이스 개발을 하는 것과 비슷한 효과가 있다.

명세 기반 테스트는 다음과 같은 여러 이점을 갖는다. 첫 번째로 프로그램 코드가 아닌 명세를 바탕으로 테스트 케이스를 설계하므로 서브 시스템이나 전체 시스템처럼 규모가 큰 단위에도 효과적으로 적용할 수 있다. 두 번째로 테스터가 구현 언어라든지 알고리즘 등 구현에 관한 지식이 없어도 테스트를 수행할 수 있으며, 사용자 관점에서 테스트를 수행하기 때문에 효과적으로 결함을 검출할 수 있는 기회가 제공된다. 세 번째로 명세 결함(일관성이나 애매모호한 점)이 드러나는 기회가 되며 명세가 완성되는 순간 테스트 케이스들을 설계할 수 있다. 즉, 코드가 구현될 때까지 기다릴 필요가 없다. 또한, 명세 기반 테스트 케이스들은 누락 결함(Missing function error)을 검출할 가능성을 높여준다. 명세에는 있지만 구현되지 않은 기능이 있는 결함을 누락 결함이라고 한다. 프로그램 코드에 누락된 기능에 관한 정보는 명세에서 얻을 수 있고 이를 바탕으로 테스트 케이스를 설계하면 누락 결함을 검출할 수 있다.

10.2 동등 분할

동등 분할(Equivalence partitioning)은 소프트웨어 테스트의 근간을 이루는 방법이라 할 수 있다. 크기가 아주 작은 프로그램이라 하더라도 모든 가능한 입력을 사용하여 프로그램을 테스트하기란 불가능하다. 일반적으로 프로그램의 입력 영역은 무한히 크기 때문에 현실적으로 모든 가능한 입력값을 사용하여 테스트 케이스를 만들 수 없다. 예를 들어, 0부터 100 사이에 있는 두 정수를 입력으로 하는 프로그램을 테스트해야 한다고 가정하자. 이때 모든 가능한 입력 조합의 수는 101*101이다. 이렇게 간단한 프로그램을 테스트하더라도 1000개가 넘는 테스트 케이스가 필요하다.

동등 분할은 테스트를 효과적으로 수행하면서도 테스트 케이스의 개수를 줄이는 방법이다. 이 방법은 기본적으로 프로그램 입력이나 출력 영역을 동등 클래스(Equivalent class)라 불리는 몇 개 영역으로 분할하여 각 클래스에서 하나의 값을 선택하여 테스트 케이스로 이용한다. 만약 동등 클래스에서 선정된 한 값에 프로그램이 올바르게 동작한다면 동등 클래스의 나머지 값들도 올바르게 동작할 것이라는 가정을 하고 있다. 한 입력 영역에 대해 여러 개의 동등 클래스로 분할할 때 주의할 점은 분할된 동등 클래스들의 합집합은 입력 영역 그 자체이고 동등 클래스들은 서로 공통된 값이 없어야 한다는 점이다.

동등 분할은 그림 10.2의 절차로 수행된다.

(1) 명세에서 입력과 출력을 식별한다.
(2) 각 입력/출력 영역을 동등 클래스들로 분할한다. 이때 프로그램이 사용되는 입력/출력에 관한 도메인 지식을 이용하거나 과거의 경험을 이용하여 입력/출력 영역을 분할할 수 있으며 아래 「심화노트」의 지침에 따라 영역을 분할할 수 있다.
(3) 각 동등 클래스에서 최소한 하나의 대푯값을 선정하여 테스트 케이스에 반영한 다음, 테스트 케이스 테이블을 작성한다.

그림 10.2 동등 분할을 수행하는 절차

심화노트 동등 클래스 입력영역 분할 규칙

일반적인 입력 영역을 분할하는 규칙은 다음과 같다.

① 입력 조건이 범위를 기술하는 경우에는 입력 조건을 만족하는 하나의 클래스와 입력 조건을 만족하지 못하는 두 개의 클래스로 분할한다. 예를 들면, 어떤 프로그램이 나이를 입력받는다고 가정하자. 이 프로그램에서 가정하고 있는 나이의 범위가 25세에서 59세까지라고 한다면 25세 이상 60세 미만, 24세 이하 60세 이상으로 분할한다.

② 입력 조건이 특정 값을 기술하는 경우에는 입력 조건을 만족하는 경우와 입력 조건을 만족하지 않는 두 개의 클래스로 분할한다. 즉, 특정 값 하나로만 이루어진 클래스와 그 값을 포함하지 않는 클래스로 분할한다.

③ 입력 조건이 어떤 집합의 원소를 기술하는 경우에는 그 집합의 원소들만으로 이루어진 클래스와 그렇지 못한 클래스로 분할한다. 예를 들면, 입력 조건이 {오렌지, 사과, 배, 포도}의 한 원소임을 요구할 때 오렌지, 사과, 배, 포도로만 구성된 클래스와 이 원소들을 하나도 갖지 않는 클래스로 분할한다.

④ 입력 조건이 어떤 개체가 존재하는지를 따지는 경우에는 있는 경우와 없는 경우 각각을 하나의 클래스로 만든다. 예를 들어, 전화번호에서 지역 번호는 경우에 따라 입력으로 받을 수도 있고 안 받을 수도 있다. 이때 지역 번호가 있는 경우를 하나의 클래스로 만들고 지역 번호가 없는 경우도 하나의 클래스로 만든다. 만약 이 예에서 입력 조건이 지역 번호 범위를 기술한다면 첫 번째 규칙도 적용할 수 있다. 즉, 지역 번호 범위가 111-555까지라면 지역 번호가 존재하는 경우에 첫 번째 분할 규칙을 적용하여 세 개의 클래스를 만들 수 있다.

Exercise 01

동등 분할을 이용한 블랙박스 테스트 개념을 이해하기 위해 놀이동산 입장권 처리와 관련된 프로그램의 명세를 살펴보자.

명세 나이가 10세 이하이면 입장을 할 수 없고 나이가 10–15세이면 보호자가 동반해야 입장이 가능하고 나이가 15세가 넘으면 혼자서도 입장이 가능하다. 또한, 80세가 넘은 경우에도 보호자가 동반해야 입장이 가능하다. 나이가 0세 이하이거나 정수가 아니면 "Invalid input" 메시지를 출력한다. 그리고 나이가 100세를 넘는 경우에는 "Too old" 메시지를 출력한다.

동등 분할 방식을 적용하기 위해 우선해야 할 일은 입력과 출력을 식별하는 일이다. 명세에서 "나이"가 유일한 입력이고 출력은 "입장 불가", "보호자 동반 입장" 및 "입장 가능"이다. 입력과 출력에 대하여 동등 분할을 수행할 때 주의할 점은 유효한 입력 및 출력만을 고려하지 않고 유효하지 않은 입력 및 출력까지도 고려해야 한다는 점이다.

표 10.1 동등 분할

	유효 분할	유효하지 않은 분할
입력	0 < 나이 ≤ 100	나이 ≤ 0 나이 > 100 나이=문자열 나이=실수
출력	"입장 불가":(0 < 나이 ≤ 10) "보호자 동반 입장": (10 < 나이 ≤ 15), (80〈나이 ≤ 100) "입장 가능": (16 < 나이 ≤ 80) "Invalid input": (나이 ≤ 0), (나이=non-integer) "Too old": (나이 > 100)	메시지 ≠ {"입장 불가", "보호자 동반 입장", "입장 가능", "Invalid input", "Too old"} =>"노인 할인 입장": (60 < 나이 ≤ 100)

예제에서는 표 10.1에서 보는 바와 같이 유효하지 않은 입력으로 나이에 문자열과 실수를 주는 경우를 고려하였다. 이와 같이 프로그램이 전혀 기대하지 않은 입력값을 주었을 때 프로그램 반응을 점검하는 것은 중요하다.

ISO/IEC/IEEE 29119에서 유효하지 않은 출력이란 명세에 명시적으로 기술되어 있지 않은 출력을 말한다. 따라서 유효하지 않은 출력의 식별은 테스터의 주관이나 과거의 비슷한 시스템에서 얻은 경험에 의존되는 경우가 많다. 표 10.1에서는 "노인 할인 입장"이라는 가상의 출력(유효하지 않은 출력)에 대한 입력 클래스를 상정한 것이다.

유효하거나 유효하지 않은 입력과 출력에 대하여 클래스들로 분할한 후에는 각 클래스에서 실제 값을 선정하여 테스트 케이스를 작성한다. 클래스에서 값을 선정할 때는 클래스의 어떤 값도 사용될 수 있다. 그 이유는 클래스에 속한 값들은 프로그램에 의해 동일하게 처리된다는 가정을 하기 때문이다. ISO/IEC/IEEE 29119에서는 입력/출력이 분할되어 나온 클래스들이 테스트 케이스에 최소한 한 번은 포함될 것을 요구한다. 표 10.2는 이와 같은 과정을 거쳐 작성한 테스트 케이스들이다.

표 10.2 테스트 케이스

테스트 케이스	입력(나이)	예상 출력	동등 클래스
1	50	"입장 가능"	0<나이≤100
2	−10	"Invalid input"	나이≤0
3	105	"Too old"	100<나이
4	"abc"	"Invalid input"	나이=문자열
5	27.5	"Invalid input"	나이=실수
6	5	"입장 불가"	0<나이≤10
7	13	"보호자 동반 입장"	10<나이≤15
8	85	"보호자 동반 입장"	80<나이≤100
9	30	"입장 가능"	16<나이≤80
10	−20	"Invalid input"	나이<0
11	23.11	"Invalid input"	나이=non−integer
12	110	"Too old"	100<나이
13	70	"입장 가능"	"노인 할인 입장": 60<나이≤100

Exercise 02 동등 분할 테스트에 대해 더 알아보기 위해 [Exercise 01]의 명세를 조금 수정하여 보자. 이제는 입력으로 남자('M')인지 여자('F')인지를 구분하고 만약 30세가 넘은 여자 관객에게 할인을 해줄 때 테스트 케이스를 설계하여 보자.

우선 유효하거나 유효하지 않은 입력과 출력에 대해 분할을 수행한다. 표 10.3에 새로 추가되거나 수정된 내용을 굵은 글씨체로 표시하였다. "성별" 입력이 추가되었으므로 이에 대해 분할을 수행한다. 분할 규칙을 적용하면 성별을 나타내는 입력에 'M', 'F' 그리고 이 두 개의 값을 가지지 않는 클래스로 분할할 수 있다. 표 10.3에서는 성별에 'M', 'F'가 아닌 값으로 'D'를 사용하였다.

표 10.3 동등 분할

	유효 분할	유효하지 않은 분할
입력	0 < 나이 ≤ 100 **성별='M'** **성별='F'**	나이 ≤ 0 나이 > 100 나이=문자열 나이=실수 **성별 ≠ {'M', 'F'}**
출력	"입장 불가": (0 < 나이 ≤ 10) "보호자 동반 입장": (10 < 나이 ≤ 15), **(80 < 나이 ≤ 100) ∧ (성별='M')** "입장 가능": (30 < 나이 ≤ 80) ∧ **(성별='M')**, (16〈나이 ≤ 30) "Invalid input": (나이 ≤ 0), (나이=non-integer), **성별 ≠ {'M', 'F'}** "Too old": (나이 > 100) **"여성 할인 입장": (30 < 나이 ≤ 80) ∧ (성별='F')** **"여성 할인 입장" ∧ "보호자 동반 입장": (80 < 나이 ≤ 100) ∧ (성별='F')**	메시지 ≠ {"입장 불가", "보호자 동반 입장", "입장 가능", "Invalid input", "Too old"} =>"노인 할인 입장": (60 < 나이 ≤ 100)

ISO/IEC/IEEE 29119에서는 입력/출력이 분할되어 나온 클래스들을 조합하는 다음과 같은 두 가지 방법을 기술하고 있다.

- One-to-One 동등 분할: 입력/출력 영역을 분할한 클래스들과 테스트 케이스 간 일 대 일 관계를 명시적으로 보여 준다. 표 10.4의 테스트 케이스들은 One-to-One 동등 분할에 의해 설계되었다. 표 10.4의 테스트 케이스 4는 "나이 ≤ 0" 클래스에 해당하는 테스트 케이스이며 이 경우 성별은 임의의 값을 사용하면 된다. 즉, 성별에 'M'이나 'F' 어떤 값을 사용해도 무방하다. 만약 성별 ≠ {'M', 'F'} 값을 사용하는 경우는 어떨까? 이렇게 만들어진 테스트 케이스는 그리 좋다고 볼 수 없다. 이 경우 나이와 성별 모두 타당하지 않은 입력값을 가지고 있다. 만약 프로그램이 이 테스트 케이스를 타당하게 입력으로 받아 처리한다면 명백하게 프로그램이 입력 필드들에 대한 검증 작업을 하지 않는다는 의미가 된다. 그러나 만약 프로그램이 이들을 입력으로 받아들이지 않는다면 테스터 입장에서는 어떤 필드가 문제가 되었는지를 알 수가 없다. 따라서 유효하지 않은 테스트 케이스를 설계하는 경우에는 한 번에 하나의 필드만 유효하지 않은 입력으로 구성하는 것이 바람직하다.

- 최소화 동등 분할(Minimized Equivalence Partitioning): 하나의 테스트 케이스와 하나의 분할된 클래스를 명시적으로 연결한 One-to-One 동등 분할과는 달리 이 방법은 하나의 테스트 케이스에 여러 개의 클래스가 포함되도록 한다. 예를 들면, 표 10.4에서 테스트 케이스 3은 One-to-One 동등 분할에서는 "100 < 나이" 클래스를 목표로 설계된 테스트 케이스이다. 그러나 최소화 동등 분할에서는 "100 < 나이"와 성별='F' 두 개의 클

래스를 모두 다루는 테스트 케이스로 간주한다. 따라서 One-to-One 동등 분할 방식보다 테스트 케이스의 수를 줄일 수 있다. 표 10.5는 최소화 동등 분할 방식을 사용하여 설계된 테스트 케이스들을 보여준다. 그러나 이 방식은 테스트 케이스 3과 같이 나이와 성별 모두 타당하지 않은 입력값들을 가지고 있다. 앞에서 이미 언급하였듯이 이러한 테스트 케이스는 가급적 피하는 것이 좋다. 이런 이유로 이 두 방식을 혼합하여 유효하지 못한 테스트 케이스는 One-to-One 방식으로 설계하고, 유효한 테스트 케이스는 최소화 동등 분할 방식을 이용하여 테스트 케이스를 설계하는 것도 고려할 만하다.

표 10.4 One-to-One 동등 분할에 의한 테스트 케이스 집합

테스트 케이스	나이	성별	예상 출력	동등 클래스
1	50	'M'	"입장 가능"	0 < 나이 ≤ 100
2	30	'M'	"입장 가능"	성별='M'
3	30	'F'	"입장 가능"	성별='F'
4	−10	'M'	"Invalid input"	나이 ≤ 0
5	105	'F'	"Too old"	100 < 나이
6	"abc"	'M'	"Invalid input"	나이=문자열
7	27.5	'F'	"Invalid input"	나이=실수
8	30	'D'	"Invalid input"	성별 ≠ {'M', 'F'}
9	5	'F'	"입장 불가"	0 < 나이 ≤ 10
10	13	'M'	"보호자 동반 입장"	10 < 나이 ≤ 15
11	90	'M'	"보호자 동반 입장"	(80 < 나이 ≤ 100) ∧ (성별='M')
12	40	'M'	"입장 가능"	(30 < 나이 ≤ 80) ∧ (성별='M')
13	20	'F'	"입장 가능"	(16 < 나이 ≤ 30)
14	−10	'M'	"Invalid input"	나이 ≤ 0
15	12.5	'F'	"Invalid input"	나이=non-integer
16	25	'D'	"Invalid input"	성별 ≠ {'M', 'F'}
17	104	'M'	"Too old"	나이 > 100
18	70	'F'	"여성 할인 입장"	(30 < 나이 ≤ 80) ∧ (성별='F')
19	85	'F'	"여성 할인 입장" ∧ "보호자 동반 입장"	80 < 나이 ≤ 100 ∧ (성별='F')
20	70	'M'	"입장 가능"	"노인 할인 입장": 60 < 나이 ≤ 100

표 10.5 최소화 동등 분할에 의한 테스트 케이스 집합

테스트 케이스	나이	성별	예상 출력	동등 클래스
1	65	'M'	"입장 가능"	0<나이≤100, 성별='M', (30<나이≤80) ∧ (성별='M') "노인할인입장": 60<나이≤100
2	40	'F'	"여성 할인 입장"	0<나이≤100, (30<나이≤80) ∧ (성별='F'), 성별='F'
3	−10	'D'	"Invalid input"	나이≤0, 성별≠{'M', 'F'}
4	110	'M'	"Too old"	100<나이, 성별='M'
5	"abc"	'M'	"Invalid input"	나이=문자열 나이=non−integer 성별='M'
6	21.5	'F'	"Invalid input"	나이=실수 나이=non−integer 성별='F'
7	5	'F'	"입장 불가"	0<나이≤100, 0<나이≤10, 성별='F'
8	13	'M'	"보호자 동반 입장"	0<나이≤100, 10<나이≤15, 성별='M'
9	90	'M'	"보호자 동반 입장"	0<나이≤100, (80<나이≤100) ∧ (성별='M') 성별='M'
10	40	'M'	"입장 가능"	0<나이≤100, (30<나이≤80) ∧ (성별='M')
11	20	'F'	"입장 가능"	0<나이≤100, 16<나이≤30, 성별='F'
12	85	'F'	"여성 할인 입장" ∧ "보호자 동반 입장"	0<나이≤100, (80<나이≤100) ∧ (성별='F'), 성별='F'

심화노트 인터페이스 기반 IDM과 기능성 기반 IDM

P. Ammann과 J. Offutt은 입력 영역을 분할하는 두 가지 방식을 제안하였다: 인터페이스 기반 IDM (Interface-based Input Domain Modeling)과 기능성 기반 IDM(Functionality-based Input Domain Modeling)이다.

인터페이스 기반 IDM은 프로그램의 기능성이나 입력 인자들 간에 존재할 수 있는 관계 등을 고려하지 않고 각 입력 인자의 영역을 기계적으로 분할한다. 반면에 기능성 기반 IDM은 프로그램의 기능성 정보 등을 바탕으로 입력 영역을 분할한다. 인터페이스 기반 IDM만으로 분할된 클래스들을 조합하여 테스트 케이스를 설계하는 경우에는 유용한 테스트 케이스가 누락될 가능성이 있으며 이 경우에는 기능성 기반 IDM으로 테스트 케이스를 추가할 수 있다. ISO/IEC/IEEE 29119에는 기능성 기반 IDM에 직접적으로 해당되는 것이 없지만, 예제에서 보인 바와 같이 출력을 기반으로 입력 영역을 분할하면 기능성 기반 IDM과 유사한 결과를 가져올 수 있다.

표 10.6

10.4 경곗값 분석

소프트웨어 결함은 보통 입력 영역의 경계에서 발생하는 경향이 있다. 경곗값 분석(Boundary Value Analysis)은 입력 영역 경계 근처에 있는 값들을 이용하여 테스트 케이스를 설계하는 테스트 방법이다. 경곗값 테스트는 동등 분할과 마찬가지로 입력/출력 영역을 여러 클래스로 분할한다. 그러나 동등 분할이 입력이나 출력을 여러 클래스로 분할하고 각 클래스에서 임의의 값을 선정하는 것과는 달리 경곗값 분석은 클래스의 경계와 경계 근처에 있는 값들을 사용하여 테스트 케이스를 설계한다.

예를 들어, 어떤 프로그램의 정수형 입력 변수 X가 10과 20 사이의 범위를 가진다고 가정할 때 동등 분할 방식과 경곗값 테스트를 통한 테스트 케이스를 구해보자. 우선 동등 분할 방식을 사용하여 입력 영역을 분할한다. 입력 영역이 범위이기 때문에 입력 조건을 만족하는 하나의 클래스와 입력 조건을 만족하지 못하는 두 개의 클래스로 분할할 수 있다(그림 10.6 참조).

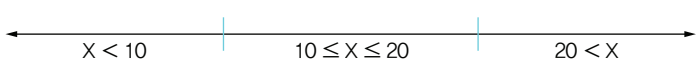

그림 10.6 동등 클래스 분할

따라서, 다음과 같은 테스트 케이스들을 추출할 수 있다:

- X=5(X < 10을 만족하는 클래스에서 추출)
- X=15(10 ≤ X ≤ 20을 만족하는 클래스에서 추출)

• X=30(X > 20을 만족하는 클래스에서 추출)

같은 예제에 대해 경곗값 테스트를 수행해 보자. 변수 X의 범위가 10 ≤ X ≤ 20으로 주어졌기 때문에 경계와 경계 근처의 값들을 선택해야 한다. ISO/IEC/IEEE 29119에서는 두 가지 방식의 경곗값 분석(BVA)에 대해 기술하고 있다: 2-value BVA와 3-value BVA. 2-value BVA는 경곗값과 경계 외부에 있는 경계와 가장 가까운 값을 선정하며, 3-value BVA는 경곗값 경계 내부와 외부에서 경계와 가장 가까운 값을 선정한다. 우선 경곗값을 식별해보자.

• X=9(X가 정수이기 때문에 X < 10은 X ≤ 9로 표현)
• X=10, 20(10 ≤ X ≤ 20)
• X=21(X가 정수이기 때문에 X > 20은 21 ≤ X로 표현)
• X=32767(만약 16비트 정수형이라면 X가 표현할 수 있는 최대 정수)
• X=-32768(만약 16비트 정수형이라면 X가 표현할 수 있는 최소 정수)

표 10.7과 표 10.8은 2-value BVA와 3-value BVA를 수행한 결과를 보여 준다.

표 10.7 2-value BVA

id	입력(X)	경계
1	9, 10	9
2	9, 10	10
3	20, 21	20
4	20, 21	21
5	32767, 32768	32767
6	-32768, -32769	-32768

표 10.8 3-value BVA

id	입력(X)	경계
1	8, 9, 10	9
2	9, 10, 11	10
3	19, 20, 21	20
4	20, 21, 22	21
5	32766, 32767, 32768	32767
6	-32767, -32768, -32769	-32768

이러한 경곗값 분석의 효용성을 알아보기 위해 다음 경우들을 생각해보자.

- 만약 프로그램이 $10 \leq X \leq 20$을 $10 < X \leq 20$으로 구현하였다면 X=10은 프로그램에서 다르게 처리되므로 이 결함을 검출할 수 있다. 그 이유는 그림 10.7에서 찾을 수 있다. 테스트로 10을 입력하면 올바르게 작성된 코드에서는 A 로직이 실행되어야 하지만 실제 구현된 프로그램(잘못된 프로그램)에서는 B 로직이 실행된다. 따라서 결함이 발견될 가능성이 커진다.
- 마찬가지로 $10 \leq X \leq 20$가 $10 \leq X < 20$으로 구현되어도 X=20에 의하여 결함이 검출된다.
- 만약 프로그램이 $10 \leq X \leq 20$을 $9 \leq X \leq 20$으로 구현하였다면 X=9가 결함을 검출할 수 있다. 그 이유는 X=9는 원래 영역에서는 외부 영역의 한 값으로 취급되었으나 프로그램에서는 영역 내부에 있기 때문이다.
- 만약 $10 \leq X \leq 20$가 $X \leq 20$으로 구현되었다면, 즉 아래쪽 경계 부분이 빠졌다면 X=9가 이제는 영역 내부에 있게 되기 때문에 결함을 검출할 수 있다. 마찬가지로 $10 \leq X \leq 20$가 $10 \leq X$로 구현되었다면 X=21이 내부의 점으로 처리되어 결함이 검출될 수 있다.

경곗값 분석의 효용성 확인을 위하여 이 프로그램에 대해 동등 클래스 분할(ECP, Equivalence Class Partitioning)과 비교해 보자. 분할 규칙에 따라 입력 영역을 분할하면 「$10 \leq X$」가 하나의 클래스를 이루고 「$X < 10$」가 또 하나의 클래스를 이룬다. 이로부터 테스트 케이스를 추출하면 {0, 50}을 추출할 수 있다. 이 두 테스트 케이스 중 어느 것을 구현된 프로그램에 실행하여도 올바른 로직을 실행하게 되어 오작동이 발생하지 않기 때문에 결함을 발견할 수 없다. 하지만 경곗값 분석은 경곗값 부근에 있는 것을 테스트로 사용하여 결함 발견 효용성을 높여준다는 것을 알 수 있다.

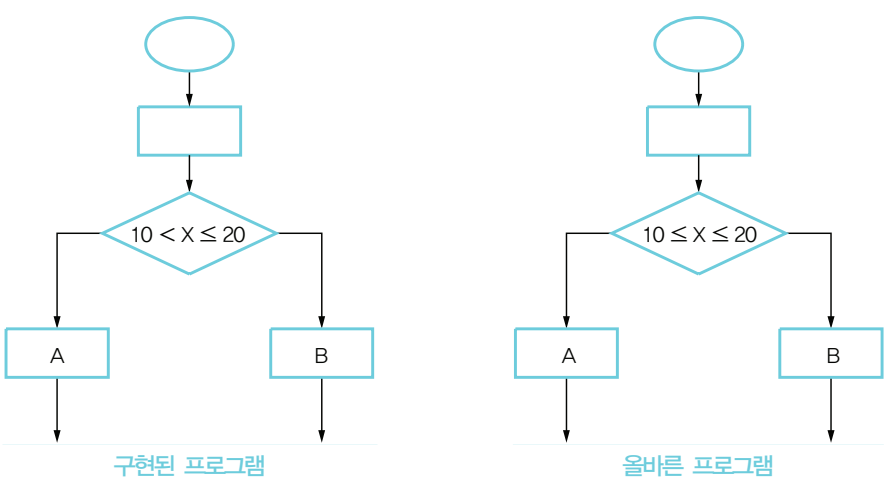

그림 10.7 경곗값 분석의 효용성

경곗값 분석은 그림 10.8의 절차로 수행된다.

(1) 명세에서 입력/출력들을 식별한다.
(2) 각 입력/출력에 대한 동등 분할을 수행한다.
(3) 각 분할된 클래스의 경곗값을 식별한다.
(4) 2-value BVA나 3-value BVA에 따라 경곗값 분석을 수행한다.
(5) (4)의 결과로 얻은 각 값에 대해 기대 출력을 명세로 구하여 테스트 케이스를 설계한다. 테스트 케이스를 구성할 때 식별된 한 경곗값에 대해 하나의 테스트 케이스를 구성하는 One-to-One 방법이나 하나의 테스트 케이스에 여러 개의 경곗값을 포함하는 최소화 방식을 사용할 수 있다.

그림 10.8 **경곗값 분석**

Exercise 04

두 개의 정수형 입력 변수 '나이'와 '신장'에 따라 처리를 달리하는 프로그램을 테스트하는 경우를 생각해 보자. 만약 각 입력 변수의 영역이 20≤나이≤50, 155≤신장≤190으로 주어졌을 때 2-value BVA를 수행하면 나이에 대해서는 경곗값으로 19, 20, 50, 51이 식별되고 신장에 대해서는 154, 155, 190, 191이 식별된다. 이들을 One-to-One 방식으로 테스트 케이스를 설계한 결과가 표 10.9이다. 예를 들어, 테스트 케이스 1부터 테스트 케이스 4는 나이 경곗값에 대한 테스트 케이스들이고, 나이 경곗값에 대해 신장 값으로 유효한 값을 선정하여 테스트 케이스들을 구성하였다. 마찬가지 방식으로 테스트 케이스 5부터 테스트 케이스 8은 신장의 경곗값을 사용하여 설계한 테스트 케이스들을 보여 준다.

표 10.9 **One-to-One 방식에 의한 테스트 케이스 설계**

테스트 케이스	1	2	3	4	5	6	7	8
나이	19	20	50	51	35	35	35	35
신장	170	170	170	170	154	155	190	191
기대 출력	reject	accept	accept	reject	reject	accept	accept	reject

표 10.10은 최소화 방식으로 테스트 케이스를 설계한 결과를 보여 준다. One-to-One 방식에 비해 테스트 케이스의 수가 반으로 줄어들었음을 알 수 있다. 그러나 테스트 케이스 1과 테스트 케이스 4처럼 모두 유효하지 않은 입력값들로만 이루어진 테스트 케이스가 생성되기 때문에 테스트 효과가 감소할 가능성이 있다.

표 10.10 최소화 방식에 의한 테스트 케이스 설계

테스트 케이스	1	2	3	4
나이	19	20	50	51
신장	154	155	190	191
기대 출력	reject	accept	accept	reject

10.5 조합 테스트

조합 테스트(Combinatorial test)는 테스트 대상 프로그램 내 여러 클래스의 각 입력 인자를 동등 분할이나 BVA 등의 방법으로 여러 클래스 또는 값으로 분할하였을 때 이들을 조합하여 테스트 케이스를 구성하는 방식이다. 예를 들어, 정수형 입력 인자가 3개 있는 int foo(int x, int y, int z)를 테스트한다고 하자. 그리고 각 입력 인자를 동등 분할을 통해 표 10.11과 같이 분할하였다고 하자. 즉, 입력 인자 x는 3개의 클래스(X1, X2, X3)로 y는 4개의 클래스(y1, y2, y3, y4)로 z는 2개의 클래스(Z1, Z2)로 분할하였다. Amman과 Offutt은 이렇게 개개의 입력 인자를 분할하는 과정을 인터페이스 기반 IDM이라 하였다(10.2절 심화노트 참조).

표 10.11 동등 분할을 통한 입력 인자 분할

입력 인자	x	y	z
동등 클래스	X1	Y1	Z1
	X2	Y2	Z2
	X3	Y3	
		Y4	

테스트 케이스를 구성하기 위해서는 이러한 입력 인자의 클래스들을 조합할 필요가 있다. 가장 간단한 조합 방식은 가능한 모든 클래스의 조합을 통해 테스트 케이스를 구성하는 것이다. 이 경우 3×4×2=24개의 테스트 케이스가 생성된다. ISO/IEC/IEEE 29119에서는 이처럼 모든 가능한 값(또는 클래스)의 조합을 생성하여 테스트 케이스를 구성하는 방법을 All combinations 테스트라 한다. 이외에도 ISO/IEC/IEEE 29119에서는 조합 테스트 방법으로 Each choice 테스트, 페어와이즈 테스트(Pairwise test) 및 Base choice 테스트를 제시한다.

- Each choice 테스트: 각 입력 인자의 분할된 클래스에서 최소한 하나의 입력값이 테스트 케이스에 포함되도록 한다.
- 페어와이즈 테스트: 각 인자의 값(또는 클래스)과 다른 인자의 값(또는 클래스)을 최소한 한 번은 조합을 하여 테스트하는 방법이다.
- All combinations 테스트: 모든 입력 인자의 모든 가능한 클래스 조합이 테스트 케이스들에 포함되도록 하는 것이다.
- Base choice 테스트: 기반이 되는 테스트 조합을 미리 선정한다. 기반 테스트는 사용자의 관점에서 선택될 빈도가 가장 높고, 일반적으로는 정상 동작할 수 있는 것을 선정하고 선정된 기반 테스트에서 하나의 인자에만 변경을 주며 나머지는 기반 테스트의 값으로 고정하여 테스트 케이스를 생성한다.

조합 테스트는 그림 10.9의 절차로 수행된다.

(1) 테스트 대상 프로그램의 입력들을 식별한다.
(2) 명세 등을 분석하여 각 입력 인자를 동등 분할이나 BVA등을 통해 여러 개의 값이나 클래스로 분할한다.
(3) 적절한 조합 테스트 방법을 선정하여 입력값(또는 클래스)들을 조합한다.
(4) 각 입력 조합에 대해 명세를 분석하여 기대 결과를 할당한다.

그림 10.9 조합 테스트 절차

Exercise 05

다음은 피자 주문과 관련된 프로그램의 명세이다.

명세 피자 도우에는 곡물, 나폴리 및 씬 크러스트가 있다. 곡물 도우는 10,000원이고 나폴리는 12,000원, 씬 크러스트는 14,000원이다. 토핑을 추가할 수 있는데 기본 토핑은 3,000원, 프리미엄 토핑은 5,000원이다. 전화로 주문하거나 온라인으로 주문할 수 있다. 온라인으로 주문하면 1,000원을 할인해준다.

표 10.12은 명세에서 입력을 식별하고 각 입력이 가질 수 있는 값들을 표기한 것이다.

표 10.12 입력 인자와 값

입력 인자	도우	토핑	주문
동등 클래스	곡물	기본	전화
	나폴리	프리미엄	온라인
	씬 크러스트		

표 10.13과 표 10.14는 각각 All combinations 테스트와 Each choice 테스트로 설계한 테스트 케이스 집합을 보여 준다.

표 10.13 All combinations 테스트에 따른 테스트 케이스 설계

테스트 케이스	입력			기대 출력
	도우	토핑	주문	
1	곡물	기본	전화	13000
2	곡물	기본	온라인	12000
3	곡물	프리미엄	전화	15000
4	곡물	프리미엄	온라인	14000
5	나폴리	기본	전화	15000
6	나폴리	기본	온라인	14000
7	나폴리	프리미엄	기본	17000
8	나폴리	프리미엄	온라인	16000
9	씬 크러스트	기본	전화	17000
10	씬 크러스트	기본	온라인	16000
11	씬 크러스트	프리미엄	전화	19000
12	씬 크러스트	프리미엄	온라인	18000

표 10.14 Each choice 테스트에 따른 테스트 케이스 설계

테스트 케이스	입력			기대 출력
	도우	토핑	주문	
1	곡물	기본	온라인	12000
2	나폴리	프리미엄	전화	17000
3	씬 크러스트	프리미엄	전화	19000

Each choice 테스트는 각 입력 인자의 클래스가 최소한 하나의 테스트 케이스에 포함되기 때문에 All combinations 테스트에 비해 테스트 케이스 수가 매우 적다.

다음 코드는 구현된 피자 주문 프로그램을 보여 준다.

```
int pizza_order(string dough, string topping, string order) {
   int pizza_price = 0;
   if (dough ==“곡물”) pizza_price = 10,000;
   if (dough ==“나폴리”) pizza_price = 12,000;
   if (dough ==“씬 크러스트”) pizza_price = 14,000;
```

```
    if (topping ==“기본”) pizza_price += 3,000;
    if (topping ==“프리미엄”) pizza_price += 5,000;
    if (order ==“온라인”) pizza_price -= 1,000;

    return pizza_price;
}
```

이 프로그램을 표 10.13의 All combinations 테스트 케이스로 테스트할 때 코드의 모든 영역이 실행됨을 볼 수 있다. 또한, 표 10.14의 Each choice 테스트 케이스도 코드의 모든 부분을 실행한다. 여기서 테스트 케이스의 수가 적더라도 테스트 효과는 유사함을 알 수 있다. 이는 코드가 어떤 결정을 내릴 때 여러 인자의 상호작용을 고려하지 않고 단 하나의 입력 인자에만 의존하기 때문이다. 이러한 형태의 프로그램은 Each choice 테스트도 효과적이다.

Exercise 06

다음은 [Exercise 05] 피자 주문과 관련된 프로그램 명세를 약간 수정한 것이다.

> 명세 피자 도우에는 곡물, 나폴리 및 씬 크러스트가 있다. 곡물 도우는 10,000원이고 나폴리는 12,000원, 씬 크러스트는 14,000원이다. 토핑을 추가할 수 있는데 기본 토핑은 3,000원, 프리미엄 토핑은 5,000원이다. 전화로 주문하거나 온라인으로 주문할 수 있다. 프리미엄 토핑을 추가하고 온라인으로 주문하면 1,500원을 할인해준다.

다음 코드는 명세를 구현한 프로그램을 보여 준다.

```
int pizza_order(string dough, string topping, string order) {
    int pizza_price=0;
    if (dough ==“곡물”) pizza_price = 10,000;
    if (dough ==“나폴리”) pizza_price = 12,000;
    if (dough ==“씬 크러스트”) pizza_price = 14,000;
    if (topping ==“기본”) pizza_price+ = 3,000;
    if (topping ==“프리미엄”) {
        pizza_price += 5,000;
        if (order ==“온라인”)
            pizza_price += 1,500;
           /*결함 pizza_price -= 1,500이 올바른 코드이다.*/
    }
    return pizza_price;
}
```

이 코드는 주석에서 볼 수 있듯이 결함이 있다. 표 10.14의 Each choice 테스트를 사용하여 구성한 테스트 케이스들은 결함을 검출할 수 없다. 그 이유는 프리미엄 토핑을 추가하고 온라인으로 주문하는 경우를 테스트하는 테스트 케이스가 없기 때문이다.

이와 같이 Each choice 테스트는 테스트 케이스를 줄일 수는 있지만, 입력 인자들의 상호작용에 따른 결함이 발생하는 경우는 테스트하지 않기 때문이다. 물론 All combinations 테스트는 입력 인자들의 상호작용을 테스트하는 테스트 케이스를 생성하지만, 입력 인자가 늘어날수록 테스트 케이스가 기하급수적으로 증가하는 단점이 있다.

표 10.15 **페어와이즈 테스트에 따른 테스트 케이스 설계**

테스트 케이스	입력			기대 출력
	도우	토핑	주문	
1	곡물	기본	전화	13000
2	곡물	프리미엄	온라인	12000
3	나폴리	기본	전화	15000
4	나폴리	프리미엄	온라인	14000
5	씬 크러스트	기본	온라인	18000
6	씬 크러스트	프리미엄	전화	19000

페어와이즈 테스트는 입력들의 모든 가능한 조합들을 테스트하는 대신 모든 입력값의 모든 짝(Pair) 조합을 테스트하는 방법이다. 즉, 모든 입력에 대해 존재할 수 있는 모든 상호작용을 고려하지 않고 모든 두 개의 입력 간에 가능한 모든 상호작용만을 고려한다. 표 10.15는 페어와이즈 테스트를 적용하여 All combinations 테스트 케이스의 개수를 12개에서 6개로 줄인 결과를 보여 준다. 좀 더 자세하게 살펴보면 도우와 토핑, 도우와 주문 및 토핑과 주문 간의 모든 가능한 조합이 포함되어 있음을 알 수 있다. 테스트 케이스 2와 테스트 케이스 4는 프리미엄 토핑을 추가했을 때 온라인으로 주문하는 경우를 테스트한다. 따라서 All combinations 테스트에 비해 테스트 케이스의 수를 줄이면서 Each choice 테스트로 검출하지 못한 결함을 검출할 수 있다.

심화노트 IPO 알고리즘

어떤 방법으로 페어와이즈 테스트 케이스들을 생성할까? 여러 방법이 존재하지만, 여기에서는 IPO(In-Parameter-Order) 방법에 대해 알아본다.

IPO 알고리즘은 프로그램이 p1, p2, …, pn개의 입력 인자를 가지고 있을 때 페어와이즈 테스트 케

이스 집합은 다음과 같은 과정을 거쳐 만든다.
우선, 입력 인자 p1과 p2로 이루어진 모든 가능한 조합들의 집합을 구성한다.
입력 인자를 하나씩 추가하면서 수평확장(Horizontal Growth, HG)과 수직확장(Vertical Growth, VG)을 반복적으로 수행하여 입력 인자 pn까지 처리한다. 수평확장은 입력 인자를 추가하는 과정이고, 수직확장은 테스트 케이스를 기존 집합에 추가하는 과정이다.
IPO 알고리즘을 통해 표 10.12의 테스트 케이스 집합을 만드는 과정을 살펴보자.
① 처음 두 인자(도우, 토핑)의 모든 쌍을 구한다.

도우	토핑
곡물	기본
곡물	프리미엄
나폴리	기본
나폴리	프리미엄
씬 크러스트	기본

[② 수평확장] 도우 인자와 주문 인자 및 토핑 인자와 주문 인자 값들의 모든 쌍을 구하고 이 쌍들의 집합을 AP라 하자. 즉, AP={(곡물, 전화), (곡물, 온라인), (나폴리, 전화), (나폴리, 온라인), (씬 크러스트, 전화), (씬 크러스트, 온라인), (기본, 전화), (기본, 온라인), (프리미엄, 전화), (프리미엄, 온라인)}. ①에서 만든 테스트 케이스들을 주문 인자 값을 사용하여 수평확장한다. 이때 AP의 짝들을 가장 많이 포함되도록 하는 주문 인자 값을 사용한다.

도우	토핑
곡물	기본
곡물	프리미엄
나폴리	기본
나폴리	프리미엄
씬 크러스트	기본

⇒

도우	토핑	주문
곡물	기본	전화
곡물	프리미엄	
나폴리	기본	
나폴리	프리미엄	
씬 크러스트	기본	

위 표는 주문으로 수평확장할 때 값으로 “전화”를 사용하는 경우이다. 이 경우는 AP 중에서 “(곡물, 전화)”, “(기본, 전화)”가 포함된다. 만약 “전화” 대신에 “온라인”을 사용했다면 “(곡물, 온라인)”, “(기본, 온라인)”을 포함한다. 이와 같이 동일한 개수를 포함한다면 “전화”나 “온라인” 어떤 값을 사용해도 무방하다. 이 예에서는 “전화”를 사용하였다. 이와 같은 과정을 AP의 모든 짝이 테스트 케이스에 포함될 때까지 반복한다. 다음은 어느 정도 수평확장이 이루어진 상태를 보여 준다.

도우	토핑	주문
곡물	기본	전화
곡물	프리미엄	온라인
나폴리	기본	전화
나폴리	프리미엄	온라인
씬 크러스트	기본	
씬 크러스트	프리미엄	

현재 테스트 케이스들에 포함된 인자 값들의 짝들을 제거한 후 AP는 다음과 같다. AP={(씬 크러스트, 전화), (씬 크러스트, 온라인), (기본, 온라인), (프리미엄, 전화)}. 수평확장을 위해 주문 인자 값으로 "전화"를 사용하면 "(씬 크러스트, 전화)"만을 포함하지만 "온라인"을 사용하면 "(씬 크러스트, 온라인)", "(기본, 온라인)" 2개의 쌍을 포함한다. 따라서, 주문 인자에 대한 수평확장의 값으로 "온라인"을 선택한다.

도우	토핑	주문
곡물	기본	전화
곡물	프리미엄	온라인
나폴리	기본	전화
나폴리	프리미엄	온라인
씬 크러스트	기본	온라인
씬 크러스트	프리미엄	

AP에 포함되지 않은 쌍은 "(씬 크러스트, 전화)"와 "(프리미엄, 전화)"이다. 이를 모두 포함하게 하는 주문 인자 값은 "전화"이다. 남은 쌍이 더 없으므로 IPO 알고리즘 과정은 종료한다. 다음은 이 같은 과정을 거쳐 구한 최종 테스트 케이스들을 보여 준다.

도우	토핑	주문
곡물	기본	전화
곡물	프리미엄	온라인
나폴리	기본	전화
나폴리	프리미엄	온라인
씬 크러스트	기본	온라인
씬 크러스트	프리미엄	전화

[③ 수직확장] 만약 여전히 남은 쌍이 있다면 나머지 쌍을 포함하도록 테스트 케이스들을 추가한다. 이 예에서는 남아 있는 쌍이 없기 때문에 수직확장이 필요 없다.

Exercise 07

다음은 [Exercise 06] 피자 주문과 관련된 프로그램 명세를 약간 수정한 것이다.

명세 피자 도우에는 곡물, 나폴리 및 씬 크러스트가 있다. 곡물 도우는 10,000원이고 나폴리는 12,000원, 씬 크러스트는 14,000원이다. 토핑을 추가할 수 있는데 기본 토핑은 3,000원, 프리미엄 토핑은 5,000원이다. 전화로 주문하거나 온라인으로 주문할 수 있다. 만약 프리미엄 토핑을 추가하고 온라인으로 주문하면 1,500원을 할인해준다. 또한, 씬 크러스트 도우에 프리미엄 토핑을 추가하여 온라인으로 주문하면 500원 추가 할인해준다.

다음 코드는 명세를 구현한 프로그램을 보여 준다.

```
int pizza_order(string dough, string topping, string order) {
   int pizza_price = 0;
   if (dough==“곡물”) pizza_price = 10,000;
   if (dough==“나폴리”) pizza_price = 12,000;
   if (dough==“씬 크러스트”) pizza_price = 14,000;
   if (topping ==“기본”) pizza_price+ = 3,000;
   if (topping ==“프리미엄”) {
       pizza_price += 5000;
       if (order==“온라인”)
            pizza_price -= 1500;
           if (dough == “씬 크러스트”)
                   pizza_price -= 700;
/*결함 pizza_price -= 500이 올바른 코드이다.*/

}
   return pizza_price;
}
```

이 코드는 주석에서 볼 수 있듯이 결함이 있다. 표 10.15의 페어와이즈 테스트 케이스들은 이 결함을 검출할 수 없다. 그 이유는 씬 크러스트 도우에 프리미엄 토핑을 추가하고 온라인으로 주문하는 경우를 테스트하는 테스트 케이스가 없기 때문이다. 이 경우는 All combinations 테스트로 검출할 수 있다. 표 10.13의 테스트 케이스 12로 결함을 검출할 수 있다.

Exercise 08	Base choice 테스트는 기반이 되는 테스트 조합을 미리 선정한다. 기반 테스트는 사용자의 관점에서 가장 선택될 빈도가 높으면서 일반적으로는 정상 동작할 수 있는 것을 선택한다. 기반 선정 조합은 우선 기반 조합을 선정하고 선정된 기반 테스트에서 하나의 인자에만 변경을 주고 나머지는 기반 테스트의 값으로 고정하여 생성한다.

표 10.12를 예를 들어, Base choice 테스트를 설명한다. (곡물, 기본, 전화)를 기반이 되는 테스트 조합이라고 하자. 표 10.16은 Base choice 테스트 케이스들을 보여 준다. 테스트 케이스 1은 기반이 되는 테스트 케이스 조합이고, 테스트 케이스 2는 도우와 토핑 인자의

기반이 되는 값은 고정하고 주문 인자의 값만 변경하여 만든 조합이다. 테스트 케이스 3은 도우와 주문 인자의 기반이 되는 값은 고정하고 토핑 인자의 값만 변경하여 만든 조합이다. 테스트 케이스 4, 테스트 케이스 5 및 테스트 케이스 6도 이러한 과정을 거쳐 만들어진다. 여기에서는 Exercise 05의 명세를 가정하여 기대 결괏값을 산출하였다.

표 10.16 Base choice에 따른 테스트 케이스 설계

테스트 케이스	입력			기대 출력
	도우	토핑	주문	
1	곡물	기본	전화	13000
2	곡물	기본	온라인	12000
3	곡물	프리미엄	전화	15000
4	나폴리	기본	전화	15000
5	씬 크러스트	기본	전화	17000

10.6 결정표 테스트

결정표 테스트(Decision table test)는 결정표를 이용하여 테스트 케이스를 설계하는 테스트 방법이다. 결정표는 조건을 기술하는 부분과 조건의 조합에 대해 취하는 행위를 기술하는 부분으로 구성된다. 그림 10.10은 결정표의 양식을 보여 준다.

		규칙(조건조합)			
조건	조건 1				
	조건 2				
	...				
	조건 n				
행위	행위 1				
	행위 2				
	...				
	행위 m				

그림 10.10 결정표 양식

Exercise 09	H 대학교에서는 성적이 C 이하인 학생들을 대상으로 여러 학습 상담 프로그램을 운영하고 있다. 성적이 B 이상인 학생들은 학습 상담 프로그램을 이수하지 않아도 된다. 성적이 C 이하인 학생들은 결석 일수가 3일 이상이면 A 학습 상담 프로그램을 받게 하고 결석 일수가 2일 이하이면 B 학습 상담 프로그램을 받게 한다. 또한, 1학년은 C 학습 프로그램을 동시에 받도록 하고 있다.

표 10.17은 위 명세를 결정표로 표현한 것이다. 결정표의 조건에서 T는 조건이 참인 경우이고, F는 거짓인 경우를 의미한다. 행위에서 T는 행위가 수행되는 경우이고, F는 행위가 수행되지 않는 경우를 의미한다.

표 10.17 **결정표 예**

		규칙							
		1	2	3	4	5	6	7	8
조건	성적 C 이하	Y	Y	Y	Y	N	N	N	N
	결석 일수 3일 이상	Y	Y	N	N	Y	Y	N	N
	1학년	Y	N	Y	N	Y	N	Y	N
행위	A 상담 프로그램	Y	Y	F	F	F	F	F	F
	B 상담 프로그램	F	F	Y	Y	F	F	F	F
	C 상담 프로그램	Y	F	Y	F	F	F	F	F

예제에서 보는 바와 같이 결정표는 문제를 분석할 때 생각할 수 있는 모든 조건과 조건들의 모든 가능한 조합에 취해야 할 행위를 열거한 표이다. 결정표를 만들면 가능한 조건 조합 중 어떤 경우가 누락되었는지 알 수 있다. 결정표 테스트는 그림 10.11의 절차로 수행된다.

(1) 명세 등을 분석하여 모든 조건을 분석한다.
(2) 모든 조건의 조합에 대한 행위를 결정한다.
(3) (1)과 (2) 단계를 통해 결정표를 만든다.
(4) 가능하지 못한 조건의 조합은 배제한다.
(5) 결정표를 축약할 수 있는지 파악한다.
(6) 결정표의 각 규칙이 최소한 한 번은 테스트될 수 있도록 테스트 케이스를 생성한다.

그림 10.11 **결정표 테스트**

Exercise 10

표 10.17에서 테스트 케이스를 생성해보자. 그림 10.11의 결정표 테스트 절차에서 (1)~(3)번 단계는 이미 수행되었으므로 (4)번 단계부터 수행한다. 결정표를 축약할 수 있는지 보기 위해서는 같은 행위를 선택하게 하는 두 가지 이상의 조건들이 있는지 살펴본다. 규칙 5-규칙 8은 성적이 B 이상이면 결석 일수와 1학년인지 상관없이 상담 프로그램을 받지 않아도 된다. 따라서 "성적이 B 이상" 조건만 성립하면 나머지 다른 조건들은 아무 의미도 없게 된다. 즉, 조건의 조합은 8개에서 5개로 줄어드는데, 이를 축약된 결정표로 나타내면 표 10.18과 같이 된다. 즉, 규칙 6, 7, 8을 하나의 규칙으로 통합하고 아무 의미가 없는 조건에는 '—' 표시로 상관없음을 나타냈다.

표 10.18 축약된 결정표

		규칙				
		1	2	3	4	5
조건	성적 C 이하	Y	Y	Y	Y	N
	결석 일수 3일 이상	Y	Y	N	N	Y
	1학년	Y	N	Y	N	Y
행위	A 상담 프로그램	Y	Y	F	F	F
	B 상담 프로그램	F	F	Y	Y	F
	C 상담 프로그램	Y	F	Y	F	F

축약된 결정표에서 테스트 케이스를 생성하는 것은 매우 간단하다. 각 규칙이 최소한 한 번은 테스트 될 수 있도록 테스트 케이스들을 생성한다. 이를 위해 규칙을 실행하는 데 요구되는 조건의 조합을 만족하는 입력 및 출력을 식별하여 테스트 케이스를 구성한다. 표 10.19는 표 10.18에서 생성되는 테스트 케이스 집합을 보여 준다.

표 10.19 결정표 테스트 케이스 설계

테스트 케이스	입력			기대 출력
	학점	결석 일수	학년	
1	D	3	1	A/C 상담
2	C+	4	3	A 상담
3	C	1	1	B/C 상담
4	D+	2	2	B 상담
5	B+	3	4	해당 없음

10.7 상태 전이 테스트

상태 전이 테스트(State-transition test)는 시스템을 상태 전이도(State-transition diagram, STD)로 모델링한 후 테스트 케이스들을 상태 전이도에서 체계적으로 선정하는 방법이다.

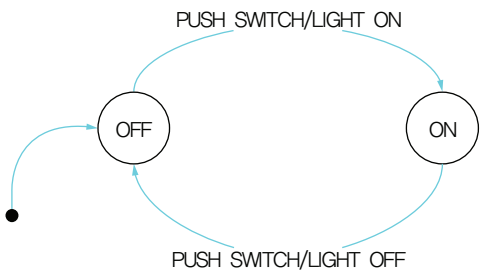

그림 10.12 형광등 상태 전이도

상태 전이도는 시스템 외부에서 들어오는 일련의 이벤트들에 대해 시스템 상태가 어떻게 전이되고 어떤 식으로 반응하는가를 나타내는 좋은 명세 수단이다. 예를 들면, 형광등은 "ON" 상태나 또는 "OFF" 상태에 있을 수 있는데 스위치 버튼을 누르는 외부 행위에 따라 "ON" 상태에서 "OFF" 상태로 전이되거나 역으로 "OFF" 상태에서 "ON" 상태로 전이된다.

그림 10.12는 형광등을 상태 전이도로 표현한 것이다. 상태 전이도에서 원은 시스템 상태를 표현하고 화살표는 상태 간의 전이를 나타낸다. "PUSH SWITCH"는 시스템 외부에서 시스템으로 들어오는 시스템의 상태 변화를 야기하는 이벤트를 나타낸다. "/" 다음에 나오는 명령은 시스템의 행위를 나타낸다. 또, 검은 원은 시스템의 시작점을 나타낸다. 이 예에서 시스템의 초기 상태는 "OFF" 상태이고, 스위치를 눌러(이벤트 "PUSH SWITCH"발생) 형광등을 켜고 끄는 과정을 모델링하였다.

ISO/IEC/IEEE 29119는 몇 가지 대표적인 상태 전이 테스트 방식을 소개한다.

- **상태 테스트(State test)**: 상태 전이도의 모든 상태를 최소한 한 번 방문하는 테스트 케이스들을 설계한다.
- **단일 전이 테스트(Single transitions test, 0-switch 테스트)**: 상태 전이도의 모든 유효한 전이들을 최소한 한 번 방문하는 테스트 케이스들을 설계한다.
- **All transitions 테스트**: 유효한 전이를 포함하여 유효하지 않은 전이들도 최소한 한 번 방문하는 테스트 케이스들을 설계한다.

• 다중 전이 테스트(Multiple transitions test, N-switch test): 상태 전이도에 있는 N+1개의 전이 시퀀스들을 최소한 한 번 방문하는 테스트 케이스들을 설계한다.

그림 10.13은 All transitions 테스트로 상태 전이 테스트를 수행하는 과정이다. 다른 테스트 방법도 비슷한 절차로 수행한다. 그림 10.13에서 단계 (4)가 없다면 단일 전이 테스트를 수행하는 절차가 된다.

(1) 테스트하려고 하는 프로그램의 명세를 상태 전이도(State transition diagram)를 사용하여 모델링한다.
(2) 상태 전이도에서 전이 트리(Transition tree)를 만든다. 이 과정은 심화노트를 참조한다.
(3) 전이 트리에서 각 전이 경로를 테스트하는 테스트 케이스들을 생성한다.
(4) 유효하지 않은 전이를 테스트하기 위한 테스트 케이스들을 생성한다.

그림 10.13 **상태 전이 테스트를 수행하는 절차**

심화노트 상태 전이도에서 전이 트리를 만드는 과정

상태 전이도에서 전이 트리를 만드는 과정은 다음과 같다.
(1) 상태 전이도의 초기 상태를 전이 트리의 루트 노드로 한다.
(3) 루트 상태에서 나오는 각 전이에 전이 목적 상태에 해당하는 노드를 추가하고 루트 노드에서 추가된 노드로 간선을 연결한다.
(3) 만약 목적 상태가 전이 트리에 이미 나와 있거나 종료 상태가 아니라면 이와 같은 과정을 각 목적 상태에 수행한다.

Exercise 11

다음은 테이프 재생기에 대한 명세이다.

명세 테이프 재생기는 재생(Play), 빠른 전진 이동(Fast forward) 및 빠른 재생(Fast play) 기능이 있다. 재생과 빠른 전진 이동은 각각 재생과 빠른 전진 이동 버튼을 통해 기능이 활성화되며 정지(Stop) 버튼을 사용하여 취소될 수 있다. 재생 모드에 있을 때 빠른 재생을 위해 빠른 전진 이동 버튼을 사용한다. 빠른 재생 모드에 있을 때는 빠른 전진 이동 버튼을 사용하여 빠른 전진 이동 모드로 전환하거나 정지 버튼을 사용하여 재생 모드로 돌아갈 수 있다. 빠른 전진 이동 모드에 있을 때는 재생 버튼을 사용하여 재생 모드로 바로 갈 수 있다.

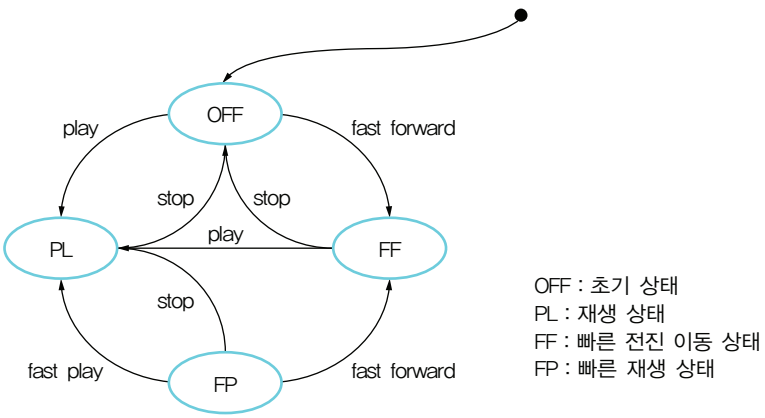

그림 10.14 테이프 재생기 상태 전이도

그림 10.14는 테이프 재생기 명세에서 구축한 상태 전이도를 보여 준다. 다음 단계에서는 상태 전이도에서 전이 트리를 만든다. 그림 10.15는 그림 10.14의 상태 전이도에서 생성된 전이 트리를 보여 준다.

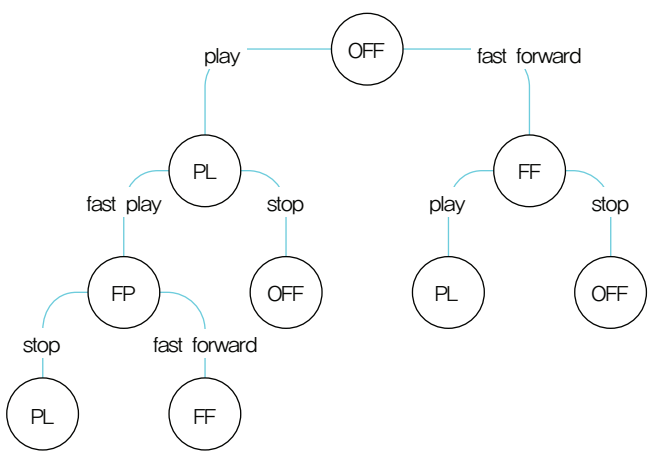

그림 10.15 전이 트리

전이 트리의 각 간선(Edge)이 하나의 테스트 케이스에 해당된다. 그림 10.15는 유효한 전이들만 테스트하는 테스트 케이스들을 보여 준다. 즉, 0-switch 테스트 또는 단일 전이 테스트를 수행한 결과이다.

표 10.20 단일 전이 테스트에 따른 테스트 케이스 집합

테스트 케이스	입력		예상 출력	
	시작 상태	이벤트	행위	목적 상태
1	OFF	play	–	PL
2	PL	fast play	–	FP
3	FP	stop	–	PL
4	FP	fast forward	–	FF
5	PL	stop	–	OFF
6	OFF	fast forward	–	FF
7	FF	play	–	PL
8	FF	stop	–	OFF

표 10.20의 테스트 케이스들은 유효한 전이만을 테스트한다. 즉, 상태 전이도에 명시적으로 기술된 상태와 이벤트의 조합만을 고려하여 만들었다. 그러나 더욱 철저한 테스트를 위해서는 이렇게 정상적인 경우만을 테스트해야 할 뿐만 아니라 유효하지 않은 경우에 대해서도 테스트할 필요가 있다. 여기에서 유효하지 않은 경우란 상태 전이도의 각 상태에서 명시되어 있지 않은 이벤트가 왔을 때를 말한다.

예를 들어, 테이프 재생기가 FP 상태에 있을 때 “Play” 이벤트에 대한 전이 정보가 상태 전이도에는 없다. 만약 현재 상태에서 기대하지 않은 이벤트를 만나면 일반적으로 시스템은 예외처리를 하고 상태는 변경되지 않는다. 표 10.21은 유효하지 않은 전이들을 테스트하는 테스트 케이스 집합이다. 표 10.20과 표 10.21의 테스트 케이스들은 All transitions 테스트 케이스 집합을 구성한다.

표 10.21 유효하지 않은 전이들을 테스트하는 테스트 케이스 집합

테스트 케이스	입력		예상 출력	
	시작 상태	이벤트	행위	목적 상태
1	OFF	stop	예외 발생	OFF
2	OFF	fast play	예외 발생	OFF
3	PL	fast forward	예외 발생	PL
3	PL	play	예외 발생	PL
4	FP	play	예외 발생	FP
5	FP	fast play	예외 발생	FP
6	FF	fast forward	예외 발생	FF
7	FF	fast play	예외 발생	FF