떤 근거로 선택할 것인가? 이것이 테스트 작업의 중요한 관심사이며 무엇을 근거로 하느냐에 따라 다른 방법이 된다.

표 9.1 테스트 케이스

고유번호	테스트 대상	테스트 조건	테스트 데이터	예상 결과
FT-1-1	로그인 기능	시스템 초기 화면	정상적인 사용자 ID('gdhong')와 패스워드('1234')	시스템 입장
FT-1-2	"	"	비정상적 사용자 ID('%$##')와 패스워드(' ')	로그인 오류 메시지
⋮	⋮	⋮	⋮	⋮

10.2 블랙박스 테스트

블랙박스 테스트는 내부 코드 구조, 구현 세부 사항 및 소프트웨어의 내부 경로에 대한 지식을 보지 않고 테스트 대상의 기능이나 성능을 테스트하는 기술이다. 블랙박스 유형의 테스트는 전적으로 소프트웨어 요구 사항 및 사양을 기반으로 한다.

블랙박스 테스트에서는 소프트웨어 프로그램의 내부 지식에 신경 쓰지 않고 소프트웨어 시스템의 입력 및 출력에 중점을 둔다.

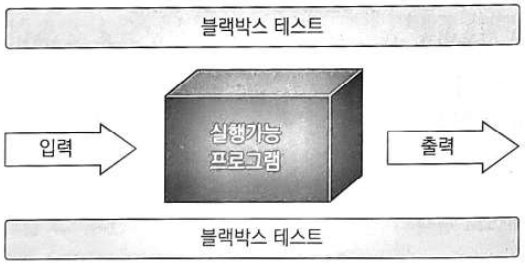

그림 10.6 블랙박스 테스트

블랙박스 테스트 방법을 사용하는 테스트 엔지니어는 시스템에 줄 수 있는 입력과 시스템이 배출하여야 하는 출력이 무엇인지를 결정하여야 한다. 테스트를 완벽하게 하기 위해서는 시스템의 모든 기능에 대하여 전부 테스트해보는 것이 좋다고 생각할 수 있다. 그러나 모듈이나 시스템이 가지는 모든 입력 자료값의 조합에 대하여 테스트 하는 것은 바람직하지 않다. 왜냐하면 프로그램 중에서 같은 부분을 구동시키고 결과를 확인하는 값들은 대푯값으로 한 번만 하는 것이 효율적이기 때문이다.

■ 장점

- 테스트하기 위하여 기술적 배경이 필요하지 않다. 사용자의 입장에서 테스트하고 사용자의 관점에서 대상 프로그램에 자극을 주고 반응을 보면 되기 때문이다.
- 코딩이 완료되어야 테스트를 시작할 수 있다. 테스터와 개발자 모두 서로가 방해하지 않고 독립적으로 작업할 수 있다.
- 크고 복잡한 응용 프로그램에 더 효과적이다.
- 테스트 초기 단계에서 결함과 불일치를 식별할 수 있다.

■ 단점

- 기술적 또는 프로그래밍 지식이 없으면 테스트할 시나리오의 가능한 조건을 무시할 가능성이 있다.
- 규정된 시간에 가능한 적은 입력을 테스트하고 가능한 모든 입력과 출력 테스트를 건너뛸 수 있다.
- 크고 복잡한 테스트 대상은 완전한 테스트 범위가 불가능하다.

10.2.1 동등 분할 기법

동등 클래스(equivalence class)는 시스템의 동작이 같을 것으로 예상되는 입력들로 구성된다. 동작이 다를 것으로 예상되는 입력은 각각 다른 동등 클래스로 분리되어야 한다.

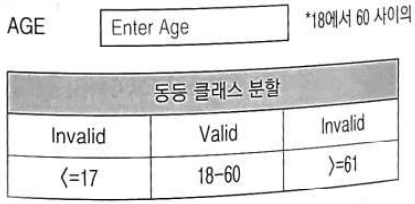

그림 10.7 동등 클래스 분할

예를 들어 프로그램에 나이를 입력받는 AGE 필드는 18에서 60까지의 숫자만을 허용한다고 하자. 결과의 유사성에 따라 입력을 나눈다면 다음과 같이 세 개의 그룹으로 나눌 수 있다.

a) AGE <= 17
b) AGE >= 61
c) 18 <= AGE <= 60

따라서 AGE가 가질 수 있는 값 중 테스트 케이스를 3개로 축소하여 모든 가능성을 커버한 것이다. 동등 분할에 의하여 같은 출력을 유발할 것으로 예상되는 입력 그룹을 찾고 대푯값을 선정하여 테스트한다면 그것으로 충분하다.

동등 클래스에서 테스트 케이스를 선택하는 방법에는 여러 가지가 있다. 정상적인 입력만 생각하는 것이 아니라 비정상적인 입력도 고려한다. 예를 들어, 길이 N인 스트링 s와 정수 n을 입력으로 받는 프로그램이 있다고 하자. 스트링이나 숫자가 아닌 여러 종류의 문자를 각각 테스트해보는 것이 좋은데 [표 10.2]에 있는 것이 동등 클래스 집합 사례이다.

표 10.2 정상 및 비정상 동등 클래스

입력	정상적인 동등 클래스	비정상적 동등 클래스
s	EQ1: 숫자를 가진 스트링 EQ2: 소문자를 가진 스트링 EQ3: 대문자를 가진 스트링 EQ4: 특수문자를 가진 스트링 EQ5: 0에서 N 사이의 길이를 가진 스트링	IEQ1: ASCII가 아닌 문자 IEQ2: 길이가 N보다 큰 스트링
n	EQ6: 정상 범위의 정수	IEQ3: 정수형 범위를 벗어난 수

이렇게 동등 클래스를 나눈 후에는 s와 n이 쌍을 이룬 테스트 케이스를 정한다. 첫 번째 테스트 케이스는 길이가 N보다 작으면서 소문자, 대문자, 숫자, 특수문자를 가진 스트링과 n을 5로 정한다. 이 테스트 케이스는 EQ1에서 EQ6까지를 커버한다. 다음으로 IEQ1, IEQ2, IEQ3를 커버하는 테스트 케이스를 하나씩 만들면 모두 4개의 테스트 케이스가 필요하다.

다른 방법으로 동등 클래스마다 다른 입력을 줄 수도 있다. 예를 들면 숫자로 이루어진 스트링과 5를 하나의 테스트 케이스로 정하여 EQ1과 EQ6를 만족시키고 EQ2에서 EQ5까지의 동등 클래스를 위한 테스트 케이스를 정한다. 또한 IEQ1에서 IEQ3까지의 테스트 케이스를 추가하는 방법이 있다.

10.2.2 경곗값 분석

경곗값 분석이란 이름 자체에서 알 수 있듯이 입력값의 경계선에 중점을 두는 테스트 방법이다. 프로그램에서 반복이나 입력의 크기, 자료의 시작과 끝 등 경계에 많은 문제를 가질 수 있고 자주 발생하는 결함의 타입이다.

경계는 시스템의 동작이 변경되는 한계 근처의 값을 의미한다. 따라서 경곗값 분석에서 유효한 입력과 유효하지 않은 입력 모두 테스트하여 문제를 확인하는 방법이다.

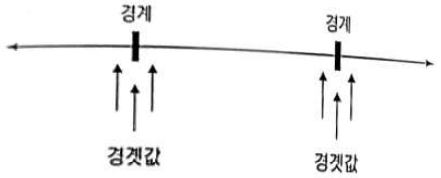

그림 10.8 경곗값 분석

예를 들어 1에서 100 사이의 값을 수용해야 하는 필드를 테스트하려면 1-1, 1, 1 + 1, 100-1, 100 및 100 + 1 경곗값을 선택한다. 즉 1에서 100까지의 모든 값을 사용하는 대신 0, 1, 2, 99, 100 및 101을 사용한다.

만일 입력이 여러 개라면 테스트 케이스들이 어떻게 경곗값을 만족시켜야 할까? 정수의 범위가 min에서 max까지라고 할 때 min-1, min, min+1, max-1, max, max+1 여섯 개가 된다. 테스트 케이스에 여러 경곗값의 조합을 선택하는 방법은 두 가지가 있다.

첫 번째 방법은 하나의 변수에 대하여 여러 경곗값들을 선택하면서 다른 변수는 정상적인 값으로 고정한다. 또한 모든 변수에 대하여 범위 안의 정상적인 값을 가진 테스트 케이스를 추가한다. 이런 경우 6n+1개의 테스트 케이스를 가진다. X와 Y 두 개의 변수라면 [그림 10.9]와 같이 13개의 테스트 케이스가 존재한다.

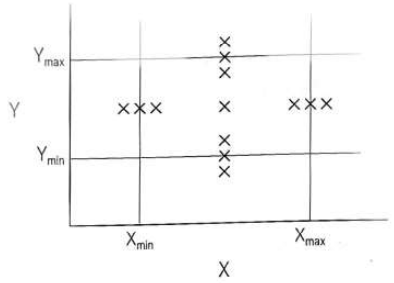

그림 10.9 두 개의 변수에 대한 경곗값 분석

10.2.3 원인 결과 그래프

동등 분할과 경곗값 분석 방법의 단점은 각각의 입력을 별도로 생각한다는 점이다. 즉 입력 조건이나 클래스의 조합을 고려하지 않는다.

여러 입력 조건의 조합을 검사하는 한 가지 방법은 입력 조건의 모든 동등치클래스의 조합을 대상으로 테스트하는 것인데 이 경우 테스트 케이스가 너무 많이 나온다. 예를 들어 서로 다른 n개의 입력 조건이 있다면 입력의 조합은 2n개의 테스트 케이스가 필요하다.

원인 결과(cause-effect) 그래프는 입력 조건의 조합을 체계적으로 선택하여 너무 많게 되지 않게 하는 기법이다.. 먼저 테스트하려는 시스템의 원인과 결과를 파악한다. 원인은 구별되는 입력 조건이며 결과는 구별되는 출력 조건이다.

예를 들어, 다음과 같은 두 가지 명령어가 있는 은행 데이터베이스가 있다.

입금 계정 번호 트랜잭션_금액
출금 계정 번호 트랜잭션_금액

명령어가 입금이고 계정 번호가 정상이면 트랜잭션 금액이 잔액에 추가된다. 만일 명령어가 출금이고 계정 번호가 정상이고 트랜젝션 금액이 잔액보다 적으면 출금된다. 만일 명령어가 정상이 아니거나 계정 번호가 정상이 아니거나 또는 출금 트랜잭션 금액이 정상이 아니면 오류 메시지가 출력된다. 원인과 결과를 정리하면 다음과 같이 된다.

원인:
c1. 명령어가 입금
c2. 명령어가 출금
c3. 계정 번호가 정상
c4. 트랜잭션 금액이 정상

결과:
e1. '명령어 오류'라고 인쇄
e2. '계정 번호 오류'라고 인쇄
e3. '출금액 오류'라고 인쇄
e4. 트랜잭션 금액 출금
e5. 트랜잭션 금액 입금

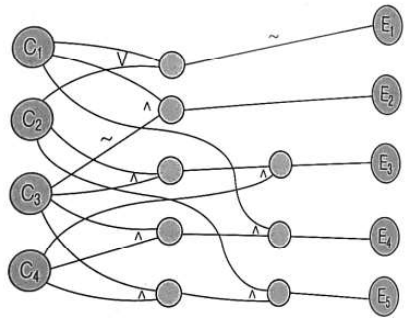

그림 10.10 원인 결과 그래프

원인 결과 관계는 [그림 10.10]과 같다. 각 결과를 결정하는 원인의 관계가 표현되어 있고 누구든 쉽게 파악할 수 있다. 예를 들어 e5 결과는 c2, c3, c4 원인에 좌우되어 c2, c3, c4가 모두 참일 경우에만 e5라는 결과가 나올 수 있다. 마찬가지로 e2 결과는 c3가 거짓일 경우에만 나올 수 있다.

▪ 결정 테이블

원인 결과 그래프로부터 테스트 케이스를 만들 때 사용하는 것이 결정 테이블(decision table)이다. 기본적인 방법은 결과 하나를 참으로 정하고 이런 조건을 가능하게 하는 원인을 찾는다. 원인들의 조건이 바로 테스트 케이스를 구성한다. 원인 조건이 참, 거짓, 또는 don't care일 수도 있다. 모든 원인 조건을 고려하기 위하여 표를 사용하는 것이 편리하다.

[표 10.3]에 있는 것처럼 각각의 결과들에 대하여 조건의 조합을 나열한다. 테이블에 있는 조건의 조합이 결과를 결정하며 이것이 테스트 케이스가 된다. 예를 들어 e3 결과를 위한 테스트는 c2, c3가 모두 참으로 세트되어야 한다. 즉 '출금액 오류 인쇄'라는 결과를 테스트하려면 테스트 케이스는 명령어가 '입금'이어야 하며(c2가 참), 계정 번호가 정상이고(c3가 참), 트랜잭션 금액이 비정상(c4가 거짓)이어야 한다.

표 10.3 결정 테이블

No.	1	2	3	4	5
c1	0	1	x	x	1
c2	0	x	1	1	x
c3	x	0	1	1	1
c4	x	x	0	1	1
e1	1				
e2		1			
e3			1		
e4				1	
e5					1

원인 결과 그래프는 좋은 테스트 케이스를 만들어 주는 이외에 시스템의 기능을 이해하는 데 도움이 된다. 왜냐하면 서로 다른 원인과 결과를 잘 구별할 수 있기 때문이다. 그래프를 적절히 방문하여 테스트 케이스를 줄이는 방법도 있다. 원인과 결과를 한 번 나열하고 관계를 표시하는 결정 테이블로 만들면 테스트 케이스는 쉽게 만들 수 있다.

10.3 화이트박스 테스트

블랙박스 테스트는 프로그램이 수행하는 기능에 초점을 가지고 테스트하는 방법으로 실제 구현한 프로그램의 내부 구조는 다루지 않는다고 하였다. 따라서 블랙박스 테스트는 프로그램의 구현보다는 기능에 관심을 갖는다. 한편 화이트박스 테스트는 모듈 안의 작동을 자세히 관찰하는 시험 방법이다. 투명한 박스와 같이 모듈의 논리적인 구조를 체계적으로 점검하기 때문에 구조적 테스트라고도 한다.

화이트박스 테스트는 프로그램의 구조를 시험하기 위하여 여러 가지 다른 구조에 대하여 테스트 케이스를 찾아내는 데 목적이 있다. 기능 테스트와는 다르게 구현된 프로그램의 구조를 기반으로 테스트하기 때문에 검증 기준이 더 정확하고 세밀하다.

화이트박스 테스트를 하려면 다음과 같은 단계를 수행하여야 한다.

1. 원시 코드를 통해 애플리케이션의 구조를 이해한다. 즉 그래프를 그려 논리흐름을 찾는다.
2. 검증 기준을 정한다. 테스트에 의하여 검증하는 범위(커버리지)를 정하고 이에 맞는 테스트 경로와 선택조건을 찾는다.
3. 각 경로를 구동시키는 테스트 데이터를 준비하여 시험 대상 프로그램을 수행시키고 결과를 비교한다.

10.3.1 논리 흐름의 표현

화이트박스 테스트를 더 자세히 이해하기 위하여 논리 흐름도(logic-flow diagram)를 이용한다. 논리 흐름도란 모듈 내의 제어 흐름을 간선으로 표시한 그래프로서 모듈 내의 모든 세그먼트가 그래프의 정점으로 표현된다. 세그먼트와 선택 구조 사이의 제어 흐름은 간선으로 표시된다.

논리 흐름도는 우리에게 익숙한 흐름도(flowchart)와는 다르다. 흐름도는 프로그램의 설계에 대한 표현이며 원시 코드를 작성하기 위하여 자세한 내용을 충분히 담고 있지 않다. 논리 흐름도는 원시 코드의 논리 흐름을 그대로 표현한 것으로 논리 흐름에 관계되지 않는 사항은 생략하여 나타낸다.

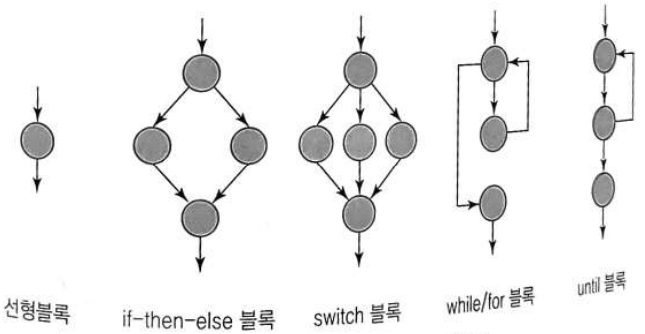

그림 10.11 논리 흐름도의 기본 요소

프로그램 논리 흐름의 기본 요소는 선형, 선택, 스위치, while 반복, until 반복이며 [그림 10.11]에 표시한 것과 같이 흐름도로 표시한다.

A, B 두 가지 값을 입력 받아 분석하는 프로그램의 논리흐름도를 그리면 [그림 10.12]와 같다.

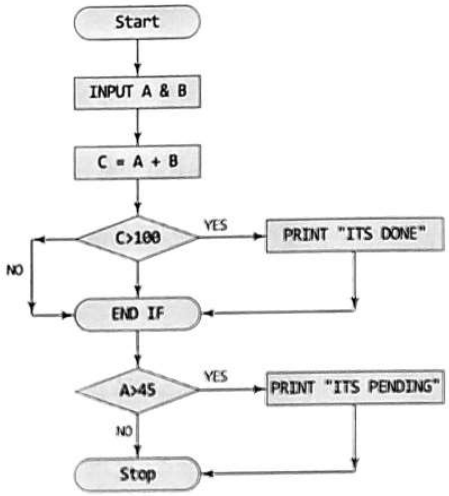

그림 10.12 논리흐름도 사례

10.3.2 검증 기준

화이트박스 테스트에서 테스트 데이터를 선택하기 위하여 테스트 실행이 프로그램의 어떤 기준을 커버하는지 먼저 결정하여야 한다. 이를 검증 기준(test coverage)이라고 하는데 일반적으로 다음 세 가지가 있다.

■ 문장 커버리지

프로그래밍 언어에서, 문장은 컴퓨터가 이해하고 행동하도록 지시하는 코드 또는 명령이다. 이름 자체에서 알 수 있듯이 문장 커버리지는 코드의 각 라인이 적어도 한 번 실행되는지를 검증하는 방법이다.

따라서 [그림 10.12] 프로그램에 대하여 문장 커버리지를 만족하는 테스트 케이스를 찾으면 다음 1개의 케이스로 모든 문장이 실행된다.

Test Case_01: A=50, B=60

하지만 한 개의 테스트 케이스로 테스트가 충분히 되었다고 할 수 있나? If 문장의 조건, 즉 C>100과 A>45라는 조건이 만족되었을 때만 테스트되고 이들이 만족되지 않을 때는 다른 결과를 보이므로 충분하다고 할 수 없다. 즉 C<=100이라는 조건과 A<=45라는 조건인 경우도 테스트 되어야 한다. 이것이 다음 설명할 분기 커버리지이다.

■ 분기 커버리지

프로그래밍 언어의 분기는 IF 문에서 이루어진다. IF 문에는 True와 False의 두 가지 분기가 있다. 따라서 분기 커버리지(branch coverage)에서는 각 분기가 한 번 이상 실행되는지 확인한다.

IF 문은 경우 두 가지 테스트 조건이 있다.

- 분기 조건이 참인 경우
- 분기 조건이 거짓인 경우

따라서 이론적으로 분기 커버리지는 실행될 때 각 결정 지점의 각 분기가 실행되도록 하는 테스트 방법이다. 따라서 A=30, B=30을 테스트 케이스에 추가 하면

Test Case_01: A=50, B=60
Test Case_02: A=30, B=30

두 개의 테스트 케이스로 3, 6번 문장의 분기에 대하여 True, False 경로를 테스트할 수 있다.

■ 경로 커버리지

경로 커버리지는 프로그램의 모든 실행 경로를 테스트하는 기준이다. 즉 프로그램의 모든 경로가 적어도 한 번씩 통과하도록 분기가 나올 때마다 누적하여 추가한다. 경로 커버리지는 분기 커버리지보다 더 강력한데 그 이유는 실행 가능한 모든 경로를 체크하기 때문이다. 따라서 경로가 복잡한 프로그램을 테스트하는 데 유용하다.

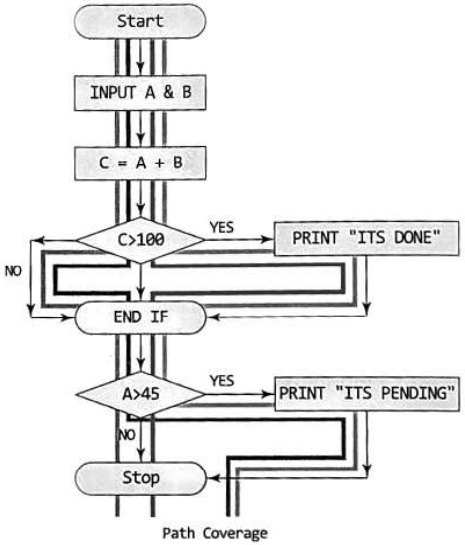

그림 10.13 경로 커버리지

[그림 10.12] 프로그램의 경로에 대하여 최대 커버리지를 보장하려면 4가지 테스트 사례가 필요하다. 2개의 분기가 있으므로 각 분기에 대해 테스트 할 2개의 조건이 필요하다. 하나는 참이고 다른 하나는 거짓 조건이다. 따라서 2개의 결정문에 대해 True를 테스트하기 위해 2개의 테스트 케이스가 필요하고 False 측을 테스트하기 위해 2 개의 테스트 케이스가 필요하므로 총 4개의 테스트 케이스가 된다.

TestCase_01: A=50, B=60(C>100: true, A>45: true)
TestCase_02: A=55, B=40(C>100: false, A>45: true)
TestCase_03: A=40, B=65(C>100: true, A>45: false)
TestCase_04: A=30, B=30(C>100: false, A>45: false)

테스트 케이스를 구하는 방법은 실행 경로에서 만나는 모든 분기 조건을 AND로 연결하면 된다. 예를 들어 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 경로를 지나가는 TestCase_01은 C>100 분기조건과 A>45 분기조건을 동시에 만족해야 하므로 A=50, B=60을 구할 수 있다.

싸이클로매틱 복잡도

화이트박스 테스트의 기본 개념은 기본 경로(basis path)라 불리는 독립적인 논리 흐름을 검사하는 테스트 케이스를 생성하는 것이다. 기본 경로라는 것은 시작(start) 노드에서 종료(exit) 노드까지의 서로 다른 경로로써 싸이클은 최대 한 번만 지나가야 간다.

기본 경로의 수는 싸이클로매틱 복잡도로 구할 수 있다. McCabe가 발견한 싸이클로매틱 복잡도를 구하는 방법은 다음 세 가지가 있다. 모두 다른 방법이지만 같은 값을 계산해 낸다.

① **폐쇄 영역의 수 + 1** 논리 흐름 그래프는 이차원 평면을 여러 영역으로 나누고 있다. 예를 들어 [그림 10.12]의 흐름도는 세 개의 폐쇄된 영역으로 나누고 있다. 따라서 여기에 1을 더하면 싸이클로매틱 수 4가 나온다.

② **노드와 간선의 수** 싸이클로매틱 복잡도 = E(간선의 수) - N(노드의 수) + 2 로 구해진다. [그림 10.12]의 예에서는 11개의 간선, 9개의 노드가 있어 싸이클로매틱 복잡도는 11-9+2=4.

③ **단일 조건의 수 + 1** 단일 조건이란 참과 거짓으로 판별되는 원자적 조건이며 AND, OR로 연결된 복합 조건이 아니라 단일 조건이란 뜻이다. 단일 조건의 개수에 1을 더하면 기본 경로의 수가 나온다.

싸이클로매틱 복잡도는 흐름 그래프의 정확성을 검증하는 데 유용하다. 즉 세 가지 방법으로 구한 싸이클로매틱 복잡도가 같아야 한다. 싸이클로메틱 복잡도는 기본 경로를 테스트하는 데 필요한 테스트 케이스의 개수를 결정하는 데 쓰인다.

▪ 루프 테스트

모든 경로 커버리지는 현실적으로 사용하기 어렵다. 대부분의 프로그램은 여러 차례 반복되는 중복된 반복문을 가지고 있어 셀 수 없이 많은 경로가 발생되기 때문이다. 그러면 어떻게 반복문을 테스트하여야 할까? Beizer는 모든 반복구조를 다음 네 가지로

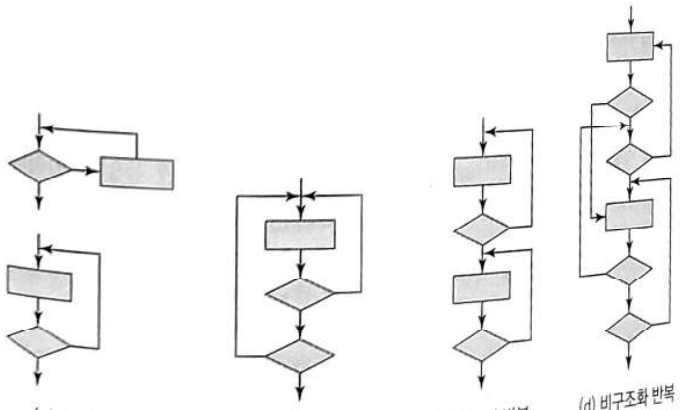

그림 10.14 반복 구조의 종류

나누었다. [그림 10.14]에 나타낸 단순 반복, 중첩된 반복, 연속된 반복, 비구조화 반복이다.

- **단순 반복문 테스트** – 단순 반복문을 테스트하기 위하여 경곗값 분석에서 사용하였던 방법을 다음과 같이 적용한다. 1) 반복 구조를 들어가지 않고 생략, 2) 반복 구조 안에서 한 번 반복, 3) 범위 내의 임의 반복 4) 반복 최대 횟수 -1 만큼 반복, 5) 반복 최대 횟수 만큼 반복
- **중첩된 반복문 테스트** – 중첩된 반복은 [그림 10.14(b)]와 같이 반복 구조가 중첩된 구조로 가장 내부에 있는 반복 구조부터 하나씩 외부로 테스트 하되 다른 외부 반복 구조는 최소 반복 횟수로 지정한다. 먼저 가장 내부에 있는 반복 구조를 단순 반복문 테스트의 다섯 가지 케이스로 나누어 테스트 한다. 다음 외부로 향하여 다음 반복구조를 테스트 한다. 다른 모든 외부 반복 구조는 최소 횟수만 반복되게 하고 모든 내부 중첩 반복구조는 임의의 횟수를 반복한다.
- **연속된 반복** – 연속된 반복 구조를 테스트할 때는 두 가지 경우를 고려한다. 반복 구조가 서로 독립적이면 단순 반복 구조와 같은 방법으로 테스트한다. 만일 하나의 반복 구조 안의 제어변수가 다른 반복 구조의 제어변수에 직접 또는 간접으로 의존하고 이런 의존 관계가 같은 경로에서 일어난다면 중첩된 반복 구조와 같은 방법으로 테스트 한다.
- **비구조적 반복** – 비구조적 반복은 이해하기도 어렵고 테스트 하기는 더욱 어렵다. 이런 경우는 제거하여 구조적 반복으로 바꾼 후 테스트하여야 한다.

10.4 상태기반 테스트

같은 입력에 대하여 언제나 같은 동작을 보이며 동일한 결과를 생성하는 프로그램이 있다. 이를 상태가 없는(state-less) 프로그램이라고 한다. 하지만 동일한 입력에 대하여 시간에 따라 다르게 동작하고 다른 결과를 출력하는 프로그램이 있다.

다르게 동작하는 이유는 상태 때문이다. 즉 프로그램의 동작과 출력은 제공되는 입력뿐만 아니라 상태에도 좌우된다. 프로그램의 상태는 프로그램이 받은 과거의 입력에 좌우된다. 다시 말해 상태는 시스템이 과거에 받은 입력의 누적된 영향을 나타낸다.

대부분의 대규모 소프트웨어 시스템은 과거의 상태가 데이터베이스나 파일에 저장되어

시스템의 동작 제어에 사용되므로 이런 범주에 속한다. 이런 시스템을 위한 테스트 케이스를 선택하는 방법이 바로 상태 기반 테스트다.

이론적으로 상태를 저장하는 소프트웨어는 상태 머신으로 모델링한다. 하지만 일반적인 프로그램에서 상태 공간은 사용되는 변수 개수의 곱에 비례하므로 거의 무한에 가깝다. 프로그램의 상태 집합을 관리할 수 있는 정도라면 상태 모델을 구축한다. 상태 모델은 네 가지 요소로 구성된다.

- 상태 – 시스템의 과거 입력에 대한 영향을 표시한다.
- 트랜지션 – 이벤트에 대한 반응으로 시스템이 하나의 상태에서 다른 상태로 어떻게 변해나가는지를 나타낸다.
- 이벤트 – 시스템에 대한 입력
- 액션 – 이벤트에 대한 출력

상태 모델은 무엇이 상태 변화를 일으키는지 이벤트에 대한 반응으로 시스템이 어떤 액션을 취하는지 나타낸 것이다. 상태 모델은 시스템의 요구를 자세히 보면 상태, 트랜지션, 액션 등을 찾아내거나 유추할 수 있고 특히 UML을 이용하여 모델링 한다면 상태 다이어그램을 작성하여 완성할 수 있다.

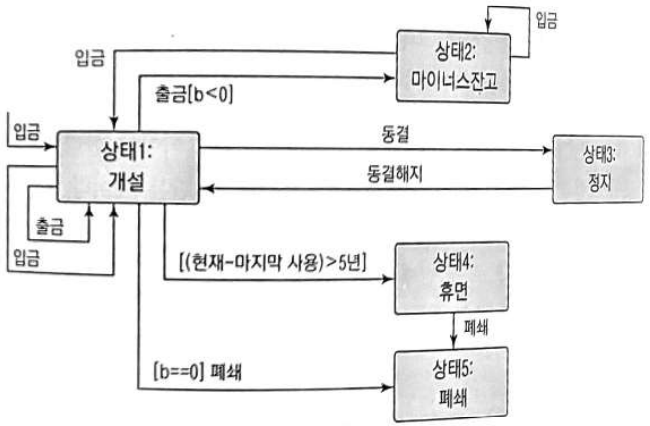

그림 10.15 예금 계좌의 상태 모델

은행 시스템을 위한 상태 머신을 만들기 위하여 계좌의 상태를 생각해 보자. 먼저 계좌에 약간의 입금으로 개설할 수 있고 통장의 잔고가 마이너스가 되지 않는 한 계속 입금

과 출금이 가능한 상태 1에 머문다. 상태 1에서 출금할 때 잔고(b)가 마이너스가 되면 상태 2, 즉 마이너스 잔고 상태로 변환된다. 마이너스 잔고 상태에서는 입금만 될 수 있다. 입금하고 잔고가 0 이상이면 다시 상태 1로 간다. 상태 1(개설)에서 사용하지 않고 5년이 경과되면 상태 4(휴면) 상태로 간다. 휴면 상태에서는 입금과 출금이 불가능하고 폐쇄 명령에 의하여 상태 5로 갈 수 있다. 상태 1에서 잔고가 0이면 폐쇄할 수 있고 그러면 상태 5(폐쇄)로 간다.

시스템의 상태 모델이 구축된 후에는 테스트 케이스를 선택할 수 있다. 아직 프로그램이 완성되지 않았더라도 설계 단계에서 작성된 상태 모델로 테스트 케이스를 찾아낼 수 있어 블랙박스 테스트에 해당된다.

시스템의 상태 모델이 주어졌을 때 테스트 케이스는 어떻게 만들어야 할까? 여러 가지 검증 기준이 있지만 많이 사용되는 세 가지는 다음과 같다.

- **모든 트랜지션** – 테스트 케이스 집합이 상태 그래프의 모든 트랜지션을 점검
- **모든 트렌지션 쌍** – 테스트 케이스 집합이 모든 이웃 트랜지션(adjacent transition)의 쌍을 점검하는 것으로 이웃 트랜지션이란 어떤 상태에 들어오는 유입(incoming) 트랜지션과 방출(outgoing) 트랜지션의 쌍을 의미한다.
- **트랜지션 트리** – 테스트 케이스의 집합이 모든 단순 경로를 만족시키는 기준. 단순 경로란 시작 상태에서 출발하여 그래프에 있는 다른 상태로 중복되지 않고 방문될 수 있는 경로를 말한다.

첫 번째 검증 기준은 테스트하는 동안 모든 트랜지션이 체크되어야 한다. [표 10.4]에 나타낸 것이 이런 기준으로 은행 시스템의 테스트 케이스를 만든 것이다. 이렇게 하면 모든 상태가 방문되어 점검된다. 트랜지션 쌍 검증기준은 각 상태의 유입과 방출 트랜지션의 모든 조합이 검사되므로 첫 번째 기준보다 더 강하다. 어떤 상태가 t1, t2 두 개의 유입 트랜지션과 두 개의 방출 트렌지션 t3, t4가 있다면 t1;t3, t2;t4도 만족되는지 점검하여야 한다.

이와 같이 상태 기반 테스트는 상태와 트랜지션에 집중한다. 간단한 경우도 많은 테스트 케이스가 나오며 상태가 늘어나면 폭발적으로 증가한다. 하지만 입력 도메인만 보고 설계하면 시스템의 상태를 고려하지 않아 테스트 시나리오가 간과되기 쉽다.

표 10.4 상태 기반 테스트를 위한 테스트 케이스

No.	트랜지션	테스트 케이스
1	start →1	입금
2	1→1	출금
3	1→2	입금
4	2→2	입금(잔고)0)
5	2→1	입금
6	1→3	출금(잔고〈0)
7	3→1	동결
8	1→4	동결해지
9	1→5	(현재-마지막사용일))5년
10	4→5	폐쇄(잔고=0)
11		폐쇄

10.5 통합 테스트

통합 테스트는 시스템을 구성하는 모듈들의 결합을 테스트하려는 것이다. 여러 개발팀에서 단위 모듈을 개발한 후 이들 사이에 인터페이스가 잘 되는지 테스트하려는 목적을 가진다.

시스템을 구성하는 여러 모듈을 어떤 순서로 결합하여 테스트할 것이냐에 따라 빅뱅(big-bang), 하향식(top-down), 상향식(bottom-up), 연쇄식(threads)이 있다. 빅뱅은 시스템을 구성하는 모듈을 각각 따로 구현하고 전체 시스템을 단번에 묶어 시험하는 방법이다. 이 방법은 초보 프로그래머들이 사용하는 방법이다. 빅뱅을 제외한 다른 방법들은 모두 점증적으로 통합한다. 시스템의 일부가 구현되고 단위 테스트된 뒤에 부분적으로 통합하여 시험한다. 단위 테스트가 끝난 모듈들을 단계적으로 추가 통합하여 테스트하는 방법이다. 점증적 방법에는 다음 세 가지가 있다.

1. 하향식 – 하향식은 명령어 처리 모듈을 먼저 구현하고 시험한다. 명령어 처리 모듈은 시스템 구조도의 최상위에 있는 모듈을 말한다. 다음 단계의 모듈이 구현, 시험 완료되었으면 추가한다. 이렇게 계속 하위층의 모듈로 내려가면서 테스트하고 추가하여 전체 시스템이 모두 결합될 때까지 계속한다.
2. 상향식 – 상향식은 시스템 구조도의 최하위층에 있는 모듈을 먼저 구현하고 테스트한다. 다음에는 바로 위층에 있는 모듈을 테스트하여 부시스템으로 만든다. 상위 모