8. 소프트웨어 테스트

학습목표

- 검증 및 확인 계획 수립
- 소프트웨어 인스펙션
- 단계별 소프트웨어 테스트
- 테스트 기법

검증 vs. 확인

❖ 검증(Verification)

"Are we building the product **right**?" ("제품을 올바르게 생성하고 있는가?")

- 소프트웨어가 명세화된 기능적 요구사항과 비기능적 요구사항을 만족하는지 검사하는 것
- 소프트웨어는 명세서와 일치해야 함.

❖ 확인(Validation)

"Are we building the **right** product?" ("올바른 제품을 생성하고 있는가?")

- 고객의 기대를 만족하도록 보장하는 것
- 요구사항 명세서가 시스템의 고객이나 사용자의 진정한 요구나 니즈를 항상 반영하는 것은 아니므로, 확인은 필수적인 프로세스
- 소프트웨어는 사용자가 실제로 원하는 것을 해야 함

V&V 프로세스

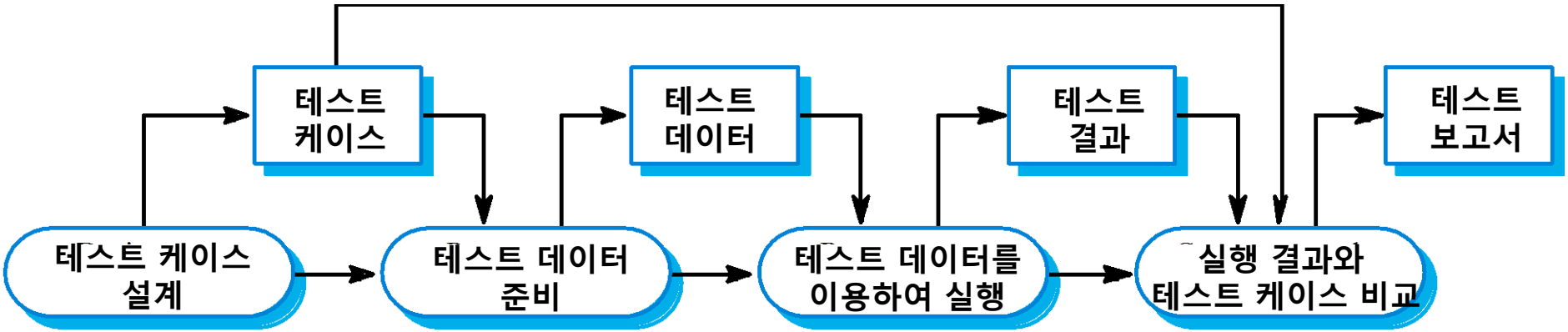
1. **소프트웨어 인스펙션** 혹은 **동료의 검토**: 요구사항 명세서, 설계 모델, 프로그램 소스 코드와 같은 시스템 표현을 분석하고 검사한다. 프로세스의 모든 단계에서 인스펙션을 실시할 수 있다. 인스펙션은 시스템이나 관련 문서의 자동 분석으로 보완될 수 있다. 소프트웨어 인스펙션은 시스템을 컴퓨터에서 실행시킬 필요가 없는 정적 V&V 기술이다.

2. **소프트웨어 테스트**: 테스트 데이터를 가지고 구현된 소프트웨어를 실행하는 것을 포함한다. 소프트웨어가 요구되는 대로 수행되는가를 점검하기 위해 소프트웨어의 출력과 운영 동작을 검토한다. 테스트는 대상 프로그램을 컴퓨터 상에서 실행시키는 동적 V&V 기술이다.

테스트 프로세스

- 단위 테스트
  - 개별적인 프로그램 컴포넌트의 시험
  - 대개 컴포넌트 개발자의 책임 (때때로 중대한 시스템의 경우는 예외)
  - 테스트는 개발자의 경험에서 유도됨
- 시스템 테스트
  - 시스템이나 서브시스템을 생성하기 위해 통합된 컴포넌트 그룹의 테스트
  - 독립적인 테스트 팀의 책임
  - 테스트는 시스템 명세서에 기반함

테스트 프로세스


테스트 케이스
테스트 데이터
테스트 결과
테스트 보고서
테스트 케이스 설계
테스트 데이터 준비
테스트 데이터를 이용하여 실행
실행 결과와 테스트 케이스 비교

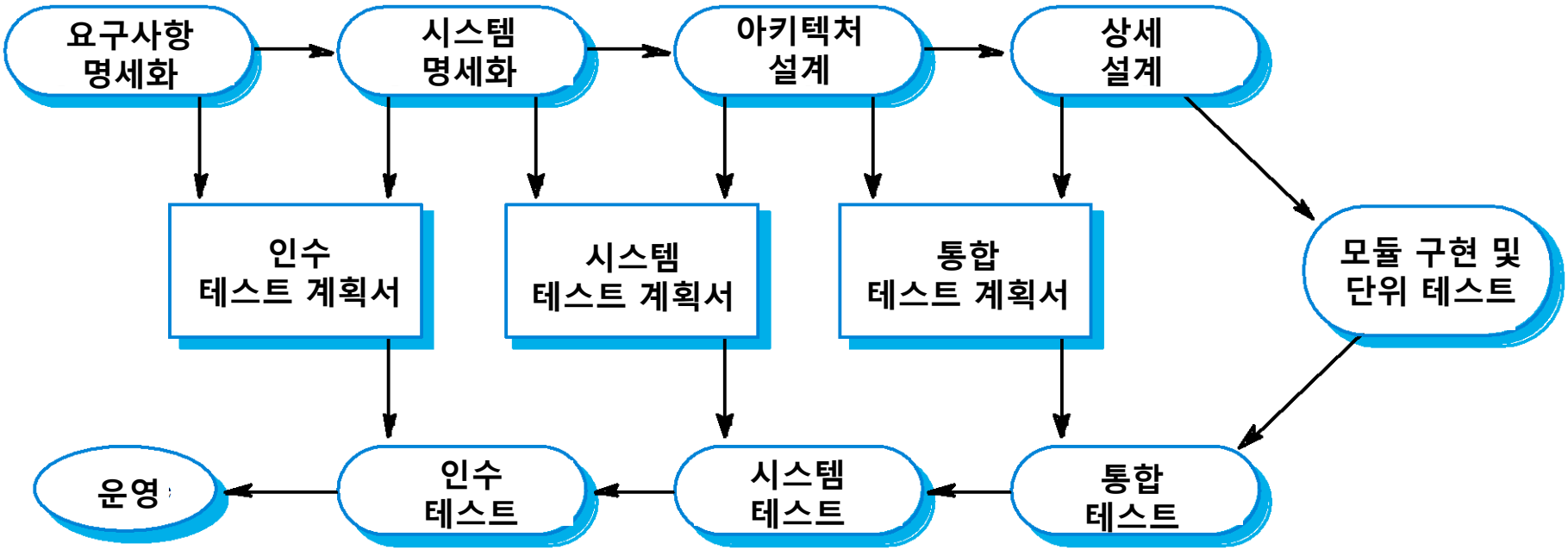
프로그램 테스트

- “테스트는 오류가 존재한다는 것을 보일 수 있는 것이지, 오류가 존재하지 않는다는 것은 보이는 것이 아니다.” by Edsger Dijkstra
- 테스트를 통해 소프트웨어에 결함이 없거나 모든 상황에서 명세서대로 동작할 것이라는 것을 입증할 수는 없다.

V&V 계획 수립

- 검증과 확인을 최대한 활용하기 위한 신중한 계획이 필요.
- 계획 수립은 개발 프로세스 초기에 시작되어야 함.

개발과 테스트의 V 모델


요구사항
명세화
시스템
명세화
아키텍처
설계
상세
설계
인수
테스트 계획서
시스템
테스트 계획서
통합
테스트 계획서
모듈 구현 및
단위 테스트
운영
인수
테스트
시스템
테스트
통합
테스트

소프트웨어 테스트 계획서의 구조

테스트 프로세스

테스트 프로세스의 주요 단계를 설명.

요구사항 추적성

사용자는 그들의 요구사항을 만족시키는 시스템에 관심을 가지고 모든 요구사항이 개별적으로 테스트되도록 계획해야 한다.

테스트 항목

테스트해야 할 소프트웨어 프로세스의 제품을 명세화해야 한다.

테스트 일정

전체적인 테스트 일정 및 자원 배정. 전반적인 프로젝트 개발 일정과 연관되어 있다.

테스트 기록 절차

테스트를 단지 수행하는 것만으로는 충분하지 않다. 테스트 결과를 체계적으로 기록해야 한다. 테스트 과정이 정확하게 수행되는지를 검토하기 위해 테스트 과정을 감사(audit)하는 것이 가능해야 한다.

하드웨어 및 소프트웨어 요구사항

필요한 소프트웨어 도구 및 추정된 하드웨어 이용을 기술해야 한다.

제약 조건

인원의 부족을 미리 예측해야 하는 것과 같은 테스트 과정에 영향을 미치는 제약 조건.

소프트웨어 인스펙션

- 소프트웨어 시스템을 검토하여 오류, 생략, 이상을 찾기 위한 정적 V&V 프로세스
- 인스펙션은 시스템의 실행을 요구하지 않으므로 구현 이전에 이용될 수 있음
- 요구사항 명세서, 설계 모델과 같은 시스템의 임의의 표현에 적용될 수 있음
- 프로그램 오류를 발견하기 위한 효과적인 기술

소프트웨어 인스펙션의 장점

① 테스트 동안에는 일단 하나의 오류가 발견되면, 다른 문제가 새로운 오류에 기인한 것인지 혹은 원래 오류의 부작용인지의 여부를 결코 확신할 수 없다. 인스펙션은 정적 프로세스이므로, 오류간의 상호 작용에 대해 관심을 가질 필요는 없다.

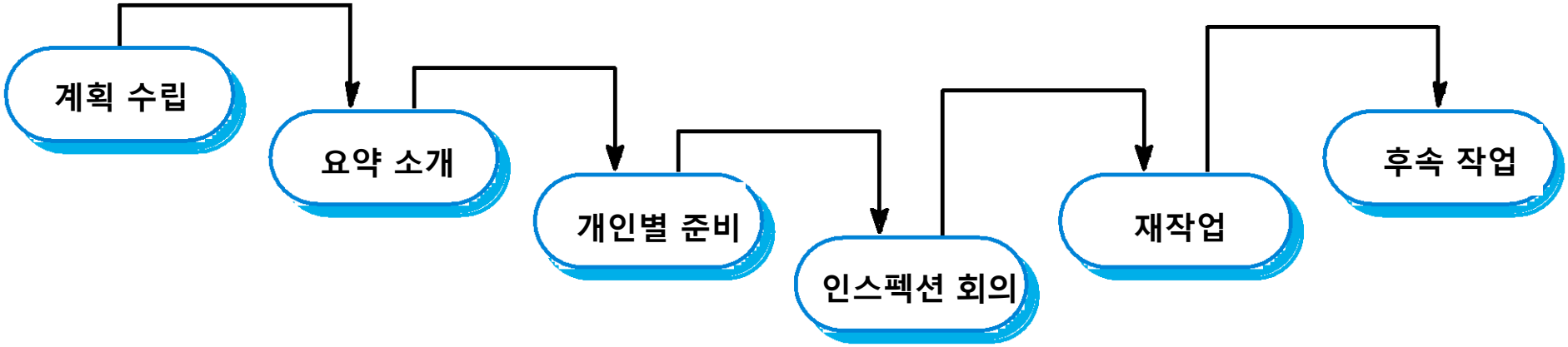
② 인스펙션은 시스템이 비록 불완전해도 추가 비용 없이 검사할 수 있다. 만일 프로그램 전체가 완성되지 않았으면, 활용 가능한 부분이라도 테스트하기 위해 특수한 테스트 프로그램이 개발되어야 하므로 시스템 개발 비용이 증가한다.

③ 인스펙션은 프로그램 결함을 찾는 것뿐만 아니라, 또한 표준의 준수, 호환성, 유지보수성과 같은 프로그램의 광범위한 품질 속성에 대해서도 검토할 수 있기 때문에, 시스템을 유지보수하기 어렵게 만드는 비효율성, 부적절한 알고리즘, 어설픈 프로그래밍 스타일을 찾을 수 있다.

프로그램 인스펙션

- 문서 검토(review)를 위한 정형 기법
- 결함의 **탐지**(수정이 아닌)가 목적
- 결함은 오류 조건(초기화되지 않은 변수)을 나타내는 논리 오류, 코드 이상 혹은 표준과의 불일치일 수도 있음

인스펙션 프로세스



인스펙션 프로세스에서의 역할

역할	설명
저자 또는 소유자	프로그램이나 문서를 작성한 프로그래머나 설계자. 인스펙션 중에 발견된 결점을 고쳐야 하는 책임이 있다.
인스펙터	프로그램이나 문서의 오류, 생략, 불일치를 찾는다. 인스펙션 팀 영역 밖의 광범위한 이슈를 다룰 수도 있다.
낭독자	인스펙션 회의 시에 코드나 문서를 낭독한다.
서기	인스펙션 회의 결과를 기록한다.
의장 또는 진행자	인스펙션 과정을 관리하고 보고서를 수석 진행자에게 제출한다.
수석 진행자	인스펙션 과정의 개선, 체크리스트 갱신, 표준 개발 등의 책임이 있다.

인스펙션 체크리스트

결함 종류	설명
데이터 결함	변수 선언 상수 이름 배열 상한 문자열에서 구분자(delimeter)의 사용 버퍼 오버플로우의 가능성
제어 결함	각 제어문의 조건 정확성 각 반복문의 종료 여부 복합문에 괄호의 적절한 사용 case 문의 모든 경우 각 case 다음의 break 문
입출력 결함	모든 입력 변수의 사용 모든 출력 변수 값 저장 예상치 못한 입력으로 인한 오류
인터페이스 결함	매개변수 개수 매개변수 타입 매개변수 순서
메모리 관리 결함	수정된 링크드 리스트의 링크 재할당 동적 메모리의 메모리 할당 명시적 메모리 해제
예외 처리 결함	모든 가능한 오류조건

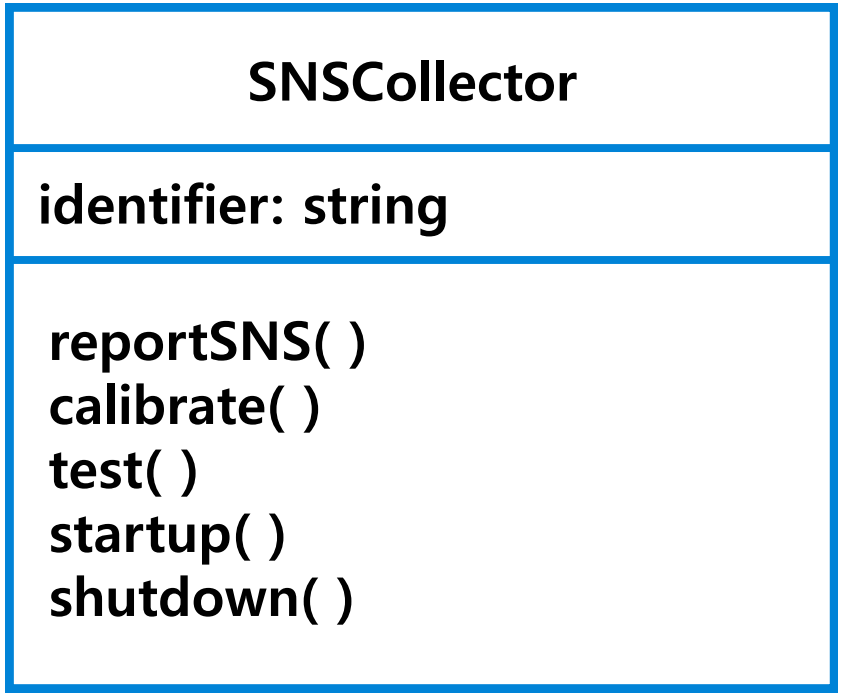
단위 테스트

- 컴포넌트 테스트 혹은 단위 테스트는 개개의 컴포넌트를 분리하여 테스트하는 프로세스
- 단위 테스트의 대상
  - 객체 내부의 개별 함수 혹은 메소드
  - 여러 속성과 메소드를 가지는 클래스
  - 여러 상이한 객체나 함수로 구성된 복합 컴포넌트

클래스 테스트

- 클래스의 완전한 테스트 범위
  - 객체와 관련된 모든 오퍼레이션들을 분리한 테스트
  - 객체와 관련된 모든 속성들의 설정과 질의
  - 가능한 모든 상태에서 객체의 실행
- 상속은 시험될 정보를 지역화하지 않으므로 클래스 테스트를 더욱 어렵게 만듦

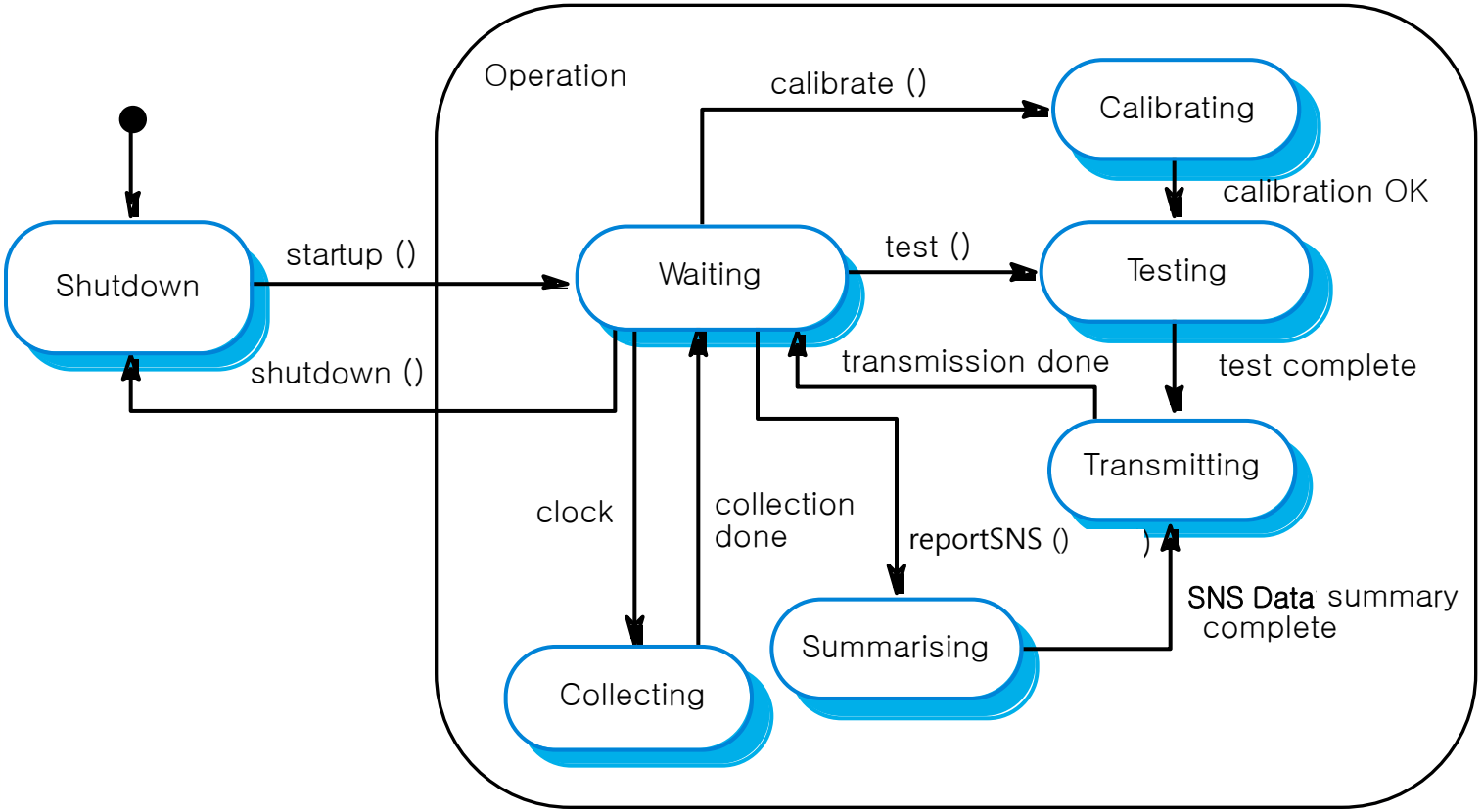
SNSCollector의 인터페이스



SNS 데이터 수집기 테스트

- reportSNS, calibrate, test, startup, shutdown에 관한 테스트 케이스 정의 필요
- 상태 모델을 이용하여 시험될 일련의 상태 전이와 이 전이를 야기하는 이벤트 순서를 인식함
- SNS 데이터 수집기에서 테스트되어야 하는 순차적인 상태의 예는 다음을 포함
  - Shutdown → Waiting → Shutdown
  - Waiting → Calibrating → Testing → Transmitting → Waiting
  - Waiting → Collecting → Waiting → Summarising → Transmitting → Waiting

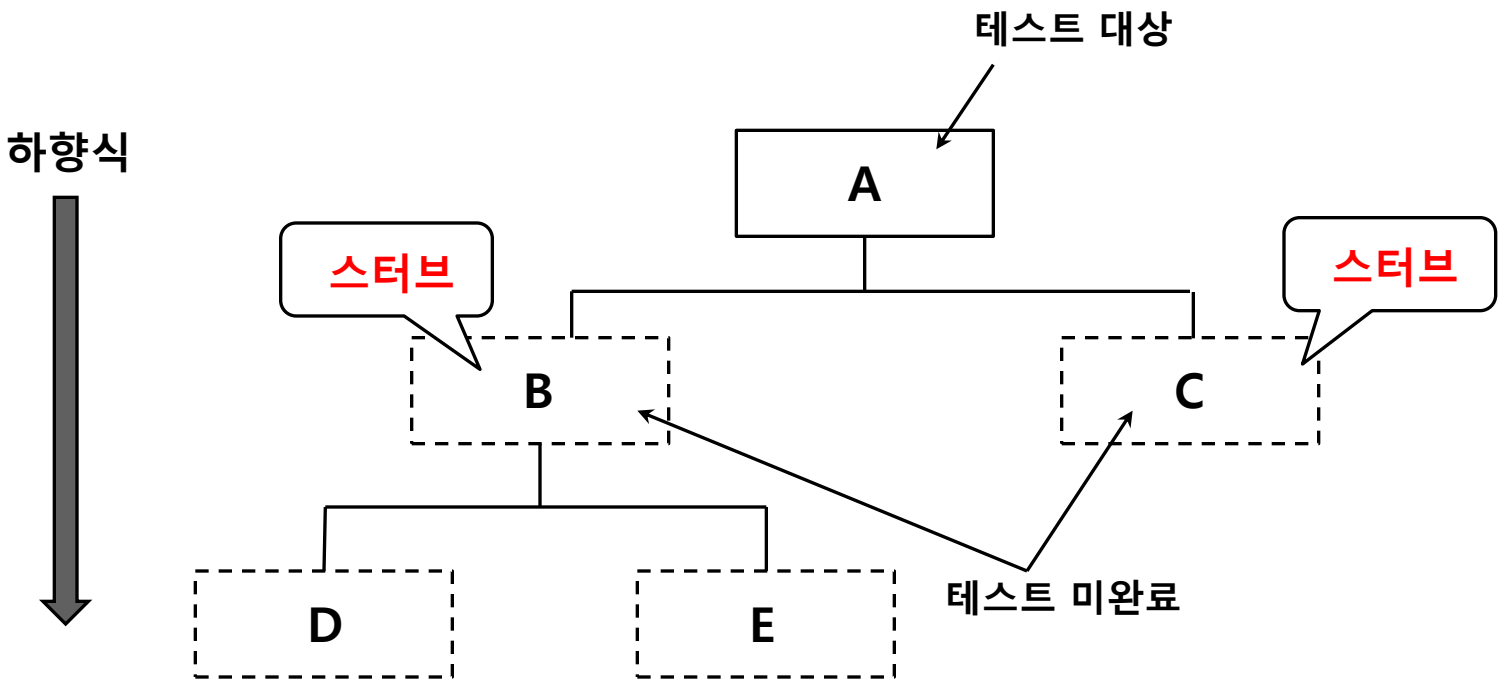
SNSCollector의 상태 다이어그램



통합 테스트

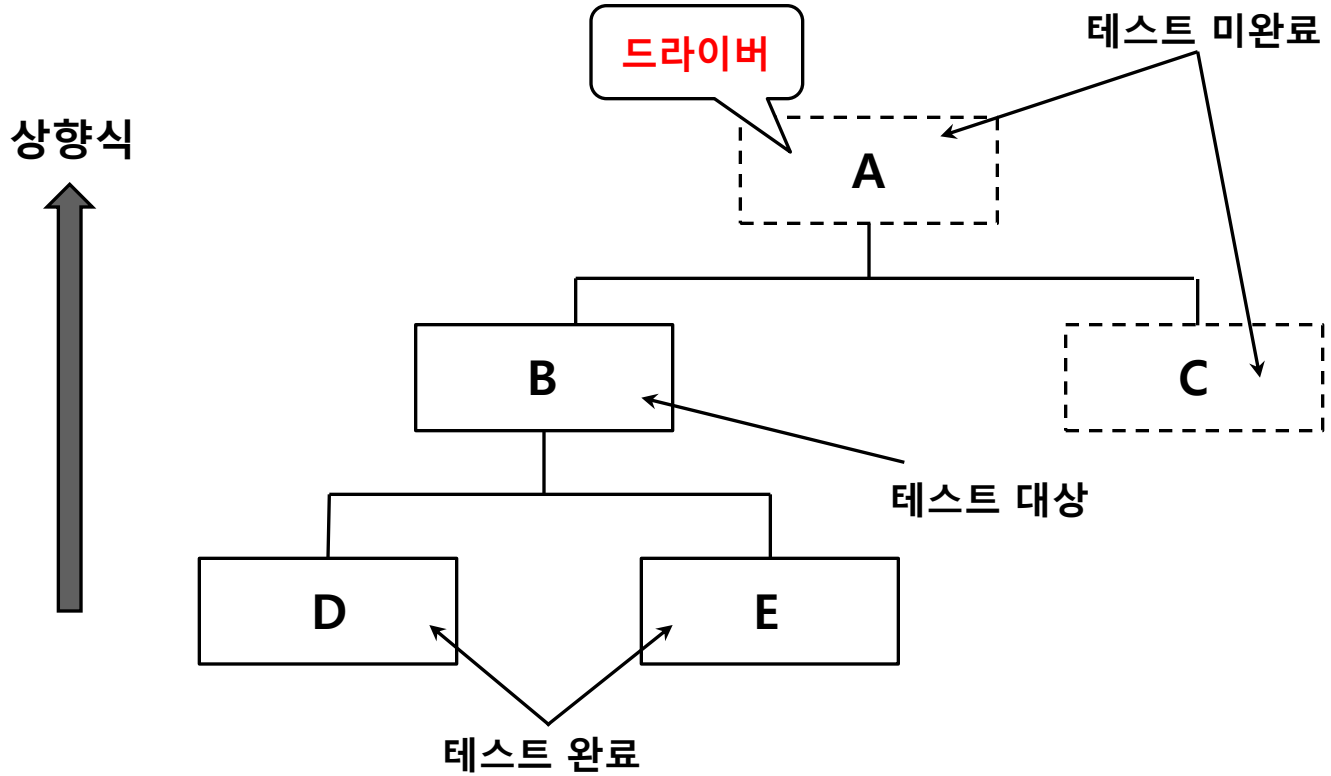
- 통합 테스트의 정의
  - 통합 후 함께 동작하는 이러한 컴포넌트들이 바르게 호출되고 인터페이스를 통해 적절한 시점에 적절하게 데이터를 전달하는지를 검사
  - 인터페이스에 관한 오류로서는 매개변수의 순서와 타입에 관한 오류, 컴포넌트 호출 시 매개변수에 관한 전제조건 모순에 관한 오류, 컴포넌트들 사이의 공유 데이터 구조가 서로 불일치하는 오류 등
- 하향식 통합
  - 시스템의 전체적인 골격이 먼저 개발되고 컴포넌트들이 추가
- 상향식 통합
  - 네트워크 및 데이터베이스 접근과 같은 공통적인 서비스를 제공하는 기반구조 컴포넌트들을 먼저 통합하고 나서 기능 컴포넌트들을 추가

하향식 테스트



프로그램 전체에 영향을 줄 가능성이 높은 상위 모듈 및 그 인터페이스의 오류를 초기에 발견하는 것이 가능하다. 또한 하위 모듈의 일부가 완성되지 않은 상태에서도 프로그램 전체의 개요를 테스트할 수 있다. 그러나 테스트의 초기 단계에서 작업의 분산화가 어렵다.

상향식 테스트



최하위 모듈은 독립적으로 테스트할 수 있기 때문에 초기 단계에서 병행적으로 여러 개의 모듈을 테스트할 수 있다. 그러나 프로그램 전체에 영향을 줄 가능성이 높은 상위 모듈 및 그 인터페이스에 관한 오류의 발견이 지연되는 단점이 존재한다. 특히 최상위에 가까운 위치에 있는 모듈에서 오류를 수정하게 되면, 해당 모듈의 하위에 속하는 모든 모듈에게 영향을 미칠 가능성이 있다.

혼합식 테스트

- 상향식 테스트와 하향식 테스트 기법을 혼합하여 각각의 장점을 결합하는 것을 목적으로 하는 테스트 기법
- 기본적으로는 하향식으로 모듈을 통합하고 스터브의 작성이 곤란한 부분이나 하위에 존재하는 중요한 부분에 대해서만 상향식으로 모듈들을 통합하여 테스트하는 방식

빅뱅 테스트

- 모든 모듈들을 개별적으로 테스트한 후에 각 모듈들을 모두 한번에 통합하여 수행하는 테스트 기법
- 소규모 프로그램이나 프로그램의 일부에 대해서 실시하는 경우가 많음

회귀 테스트(Regression test)

- 새로운 컴포넌트가 통합되고 테스트될 때 이전에 이미 테스트가 완료된 컴포넌트의 상호작용 패턴이 변경되어야 할 수도 있고, 통합 전의 간단한 시스템의 테스트에서 나타나지 않았던 오류들이 새롭게 드러날 수도 있다.
- 새로운 증분이 통합될 때, 새로운 시스템 기능을 검증하기 위해 필요한 새로운 테스트뿐만 아니라 이전의 증분들에 관한 테스트를 다시 실시하는 것이 중요하다는 의미이다.
- 기존의 테스트를 다시 실시하는 것을 회귀 테스트라고 부른다. 만일 회귀 테스트가 오류를 발견하면, 이 오류가 새로운 증분의 통합을 통해 드러난 이전 증분의 오류인지 혹은 추가된 증분의 오류인지의 여부를 검사해야 한다.
- 비용이 많이 드는 기법이라서 자동화 도구의 지원이 없으면 비실용적이다.

시스템 테스트

- 완성된 소프트웨어 시스템 전체의 동작을 확인함으로써 시스템 내부에 존재하는 오류를 찾아냄
- 실제 시스템이 운영될 환경과 가장 가까운 환경으로 테스트 환경을 구축하고 명세서에 기술된 기능과 성능 등이 구현되어 있는지를 검사
- 시스템 테스트는 크게 기능 테스트와 성능 테스트로 구분
  - 기능 테스트에서는 최종적으로 통합된 시스템이 명세서에 기술된 기능을 만족하고 있는지를 검사
  - 성능 테스트에서는 비기능적인 요구사항을 평가하기 위해 실시. 성능 테스트의 유형은 스트레스 테스트, 용량 테스트, 호환성 테스트, 보안 테스트, 타이밍 테스트, 복구 테스트, 사용편의성 테스트 등

테스트 케이스 설계

- 시스템을 테스트하는데 이용되는 테스트 케이스(입력과 출력)를 설계
- 테스트 케이스 설계의 목표는 효과적인 테스트 집합의 생성

기능 테스트

- 테스트를 담당하는 사람이 소프트웨어의 구현이 아닌 기능에 관심을 가지고, 시스템이 명세서를 만족하여 고객의 신뢰를 높이도록 하는 것이다.
- 기능 테스트는 대개 테스트 케이스가 명세서로부터 유도되는 블랙박스 테스트이다.
- 기능 테스트를 다른 이름으로 릴리스 테스트라고 하는 이유는 고객에게 배포될 시스템의 릴리스를 테스트하는 프로세스이기 때문이다.
- 요구사항을 만족하는지 검증하기 위한 최선의 방법은 시나리오 기반 테스트로, 다수의 시나리오를 고안하여 이 시나리오로부터 테스트 케이스를 유도한다. 또한 시나리오 기반 테스트를 구성하여 가장 가능성이 높은 시나리오를 먼저 테스트하고, 특이하거나 예외적인 시나리오는 나중에 고려한다.

성능 테스트

- 시스템이 완전하게 통합되면, 시스템을 성능과 신뢰도 같은 창발성에 관해 테스트를 할 수 있다. 성능 테스트는 시스템이 원하는 부하를 처리할 수 있다는 것을 보장하도록 설계되어야 한다. 이것은 시스템의 성능상 더 이상 받아들일 수 없게 될 때까지 부하를 서서히 증가시키는 일련의 테스트를 포함한다.
- 성능 테스트는 애초에 설계된 이상으로 소프트웨어 시스템에 스트레스를 주는 것을 의미하므로 **스트레스 테스트**라고 부른다.
- 스트레스 테스트는 네트워크를 기반으로 한 분산 시스템에서 특히 중요하다. 이 시스템은 부하가 심하면 심각한 기능 저하가 나타날 수 있다. 네트워크는 상이한 프로세스 간에 교환되어야 하는 데이터를 조정하느라고 꼼짝 못하게 된다.

테일러 리포트

FA·IT·통신 기술 접목한 스마트팩토리는?...'인더스트리 4...
- (5) 박종현 KEBA코리아 지사장- "자율생산은 기계 통제 고민...
- (6)박현 지브라테크놀로지코리아 이사 - "옴니채널 시대에...

정부, BMW 화재 원인 찾기 위해 '스트레스 테스트' 진행

발행일 : 2018.08.31

가 가

[AD] 로옴전자 - 출력 감시 기능 탑재, 차량용 초소형 강압 DC/DC 컨버터 「BD9S 시리즈」 개발

교통안전공단, BMW 화재 원인 규명 조사 계획 기자회견

정부가 BMW 화재 사태 명확한 원인조사를 위해 피해자 모임에서 제안한 '스트레스트 테스트' '시뮬레이션 테스트' 등을 진행하기로 결정했다. BMW 피해자 모임은 국토교통부 측에 대한 소송을 취소하고, 환경부 측에 대한 소송만 진행한다. 특히 BMW 인증 과정에서 책임을 맡은 연구원을 추가적으로 피고소인에 넣는다.

BIZPLUS

- SBA, '스무 살 SBA, 사회와 함께 커나간다'.....
- SBA-한국산업인력공단, '외국인 근로자 무역...
- SBA, 오는 23일 ' 2018 IT-제조 융·복합 기업...
- 치주염, 구취 막는데 이 방법이 최고
- 영화관을 통째로 집으로 옮겼다
- 바리스타 뺨치는 라테아트 비결은?

이향선의 테크비전 ›

오픈소스SW 라이선스, 4차 산업 혁명 경쟁력 핵심

인수 테스트

- 시스템 테스트에 의해 시스템이 요구사항대로 동작하는 것을 기대할 수 있다면, 그 다음 단계에서 인수 테스트를 실시한다. 인수 테스트는 개발자가 아닌 고객이나 사용자가 요구사항을 기반으로 테스트를 실시한다. 이와 같이 고객의 요구사항대로 시스템이 동작하는지의 여부를 고객과 사용자 스스로 확인하는 것이다.
- 시장에 출시하여 일반인을 대상으로 하는 소프트웨어처럼 사용자를 특정할 수 없는 경우의 인수 테스트는 개발자가 사용자의 입장이 되어서 수행한다. 이러한 테스트를 **알파 테스트**라고 한다.
- 잠재적 사용자들 중 일부에게 시험적으로 사용해보도록 하여 수행하는 인수 테스트를 **베타 테스트**라고 한다.

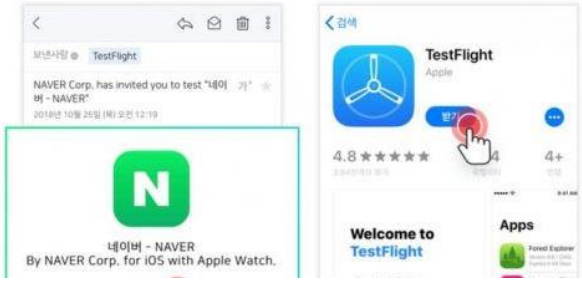
일반

바뀐 네이버 모바일, 아이폰·모바일 웹도 베타테스트

최종수정 2018.11.04 09:16 기사입력 2018.11.04 09:16 댓글 쓰기 뉴스듣기 가 가

iOS 베타테스터 5일까지 모집...1만명 대상

모바일 웹 베타버전도 조만간 선보일 예정



많이 본 뉴스 > 종합 산업/IT

1 10명중 4명이 사먹는 물 '삼다수'...이르면 다...

테스트 기법

- 테스트의 목적이 소프트웨어 내부에 존재하는 오류를 찾아내는 것이므로 좋은 테스트란 아직 찾지 못한 오류를 효율적으로 찾는 것
- 좋은 테스트 기법은 보다 적은 테스트 케이스를 사용하여 보다 많은 오류를 찾아낼 수 있는 기법
- **요구사항 명세서에 기반을 둔 블랙박스 테스트**와 **구현에 기반을 둔 화이트박스 테스트**로 구분

블랙박스 테스트

- 블랙박스 테스트는 소프트웨어 내부의 로직(프로그램의 내부 구조 및 알고리즘)을 고려하지 않고, 명세서에 기술되어 있는 소프트웨어의 동작을 토대로 실시하는 테스트이다.
- 프로그램에 입력 데이터를 제공하여 실행 결과만을 관측함으로써 오류를 찾는다.
- 대표적인 테스트 기법으로 동등 분할법, 경곗값 분석법, 원인결과 그래프 기법 등이 있다.

동등 분할(Equivalence partition)법

- 프로그램 입력값의 영역을 명세서에 있는 입력 조건에 관하여 유효 범위와 무효 범위로 분할하여 테스트 케이스를 유도하는 방법
- 동등(equivalence)이란 같은 범위에 속하는 입력 데이터에 대하여 동일한 실행 결과를 얻을 수 있는 것, 즉 프로그램의 동작이 동일하게 되는 것을 의미

"Hyeja5798"과 "YT588"

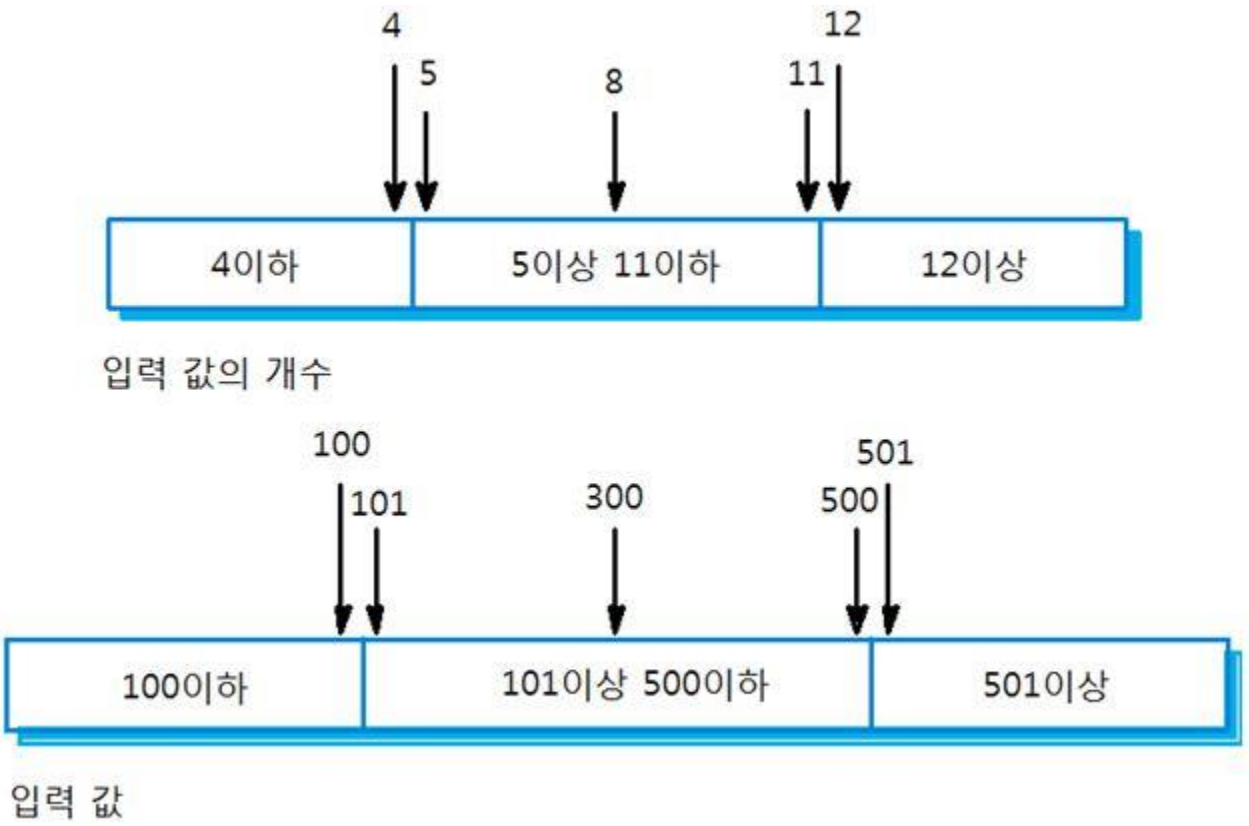
입력 조건	유효 동등 클래스	무효 동등 클래스
문자수	5 이상 9 이하	4 이하 혹은 10 이상
문자의 종류	영문자와 숫자	영문자만, 숫자만, 영문자와 특수문자, 숫자와 특수문자, 영문자와 숫자와 특수문자
선두 문자	영문자	숫자, 특수문자

"xy88"(4문자 이하), "Youngtae"(영문자만으로 구성), "88couple"(선두 문자가 숫자)

1개의 무효 동등 클래스에 속하는 입력 데이터와 나머지 유효 동등 클래스에 속하는 입력 데이터를 이용하여 테스트를 실시

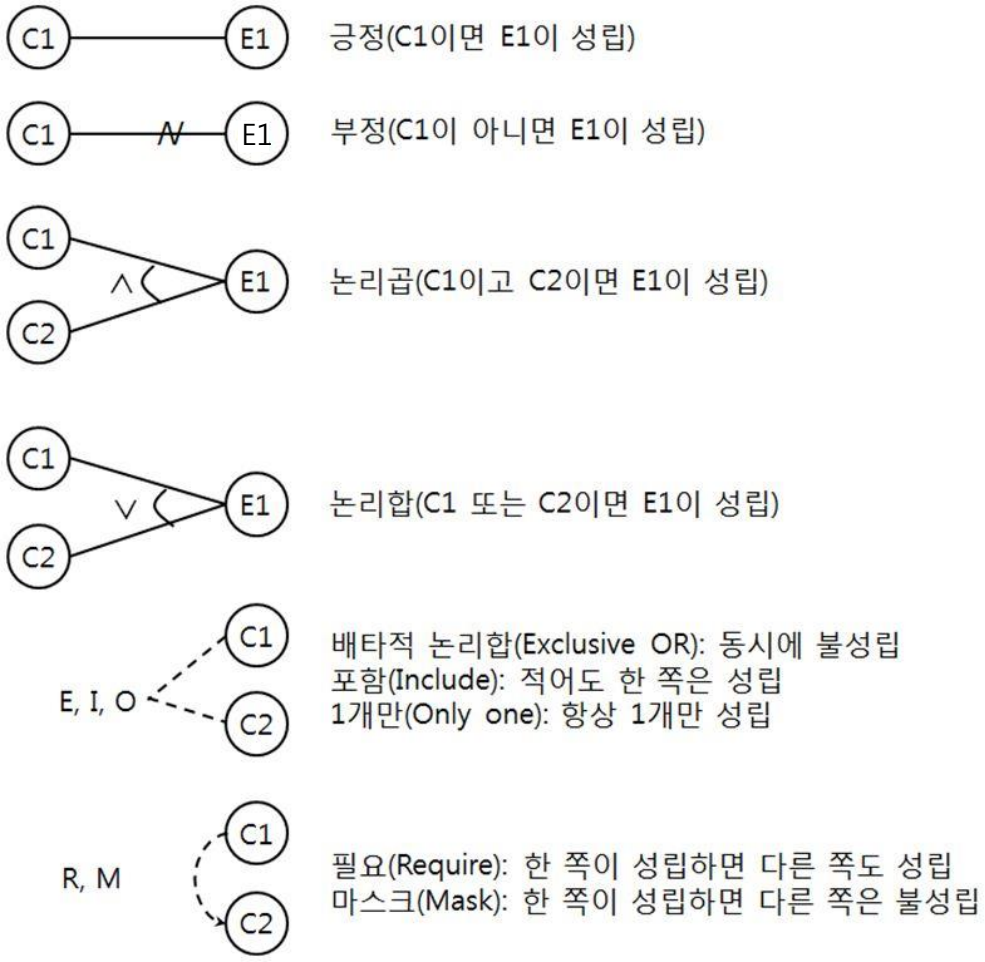
경곗값 분석(Boundary value analysis)

- 경험적으로 많은 오류가 입력 데이터의 경곗값 근처에서 발생하고 있다고 알려져 있으므로, 입력 데이터로 경곗값과 경곗값으로부터 조금 떨어진 값을 사용



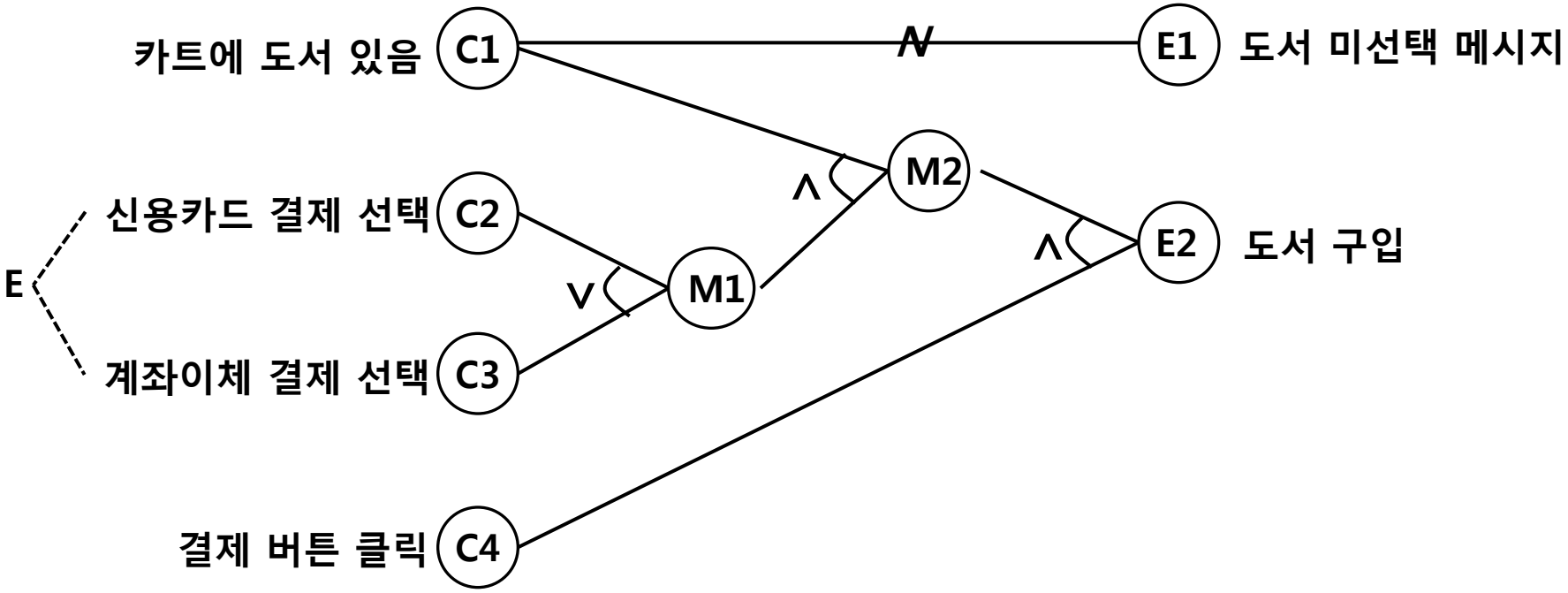
원인결과 그래프(Cause-effect graph) 기법 (1/3)

- 소프트웨어를 동작시키는 원인인 입력과 소프트웨어가 실행된 결과인 출력의 인과관계는 원인 결과 그래프로 표현. 원인과 결과는 소프트웨어의 기능 명세를 토대로 유도.



논리 기호

원인결과 그래프(Cause-effect graph) 기법 (2/3)



- C1과 E1은 "카트에 도서가 없는 경우, 도서 미선택 메시지가 표시" 라는 인과 관계가 존재한다.
- C2와 C3는 동시에 성립하지 않는다는 논리 관계에 의해 "신용카드 결제 선택 또는 계좌이체로 결제 선택" 중에서 어느 하나가 된다.

원인결과 그래프(Cause-effect graph) 기법 (3/3)

❖ 결정표

- 원인으로서 채택 가능한 모든 조합에 관하여 그 결과를 원인결과 그래프로부터 구해서 작성

		1	2	3	4	5	6	7	8	9	10	11	12
원인	C1	×	×	×	×	×	×	○	○	○	○	○	○
	C2	×	×	×	×	○	○	×	×	×	×	○	○
	C3	×	×	○	○	×	×	×	×	○	○	×	×
	C4	×	○	×	○	×	○	×	○	×	○	×	○
중간	M1	×	×	○	○	○	○	×	×	○	○	○	○
	M2	×	×	×	×	×	×	×	×	○	○	○	○
결과	E1	○	○	○	○	○	○	×	×	×	×	×	×
	E2	×	×	×	×	×	×	×	×	×	○	×	○

- 12가지의 테스트 케이스를 유도 가능
- "도서 구입"이라는 결과인 E2가 성립하는 경우에 대해서 "카트에 도서가 있고, 신용카드 결제를 선택하였으며, 결제 버튼 클릭"과 "카트에 도서가 있고, 계좌이체 결제를 선택하였으며, 결제 버튼 클릭"과 같은 두 가지 테스트 케이스를 유도

화이트 박스 테스트

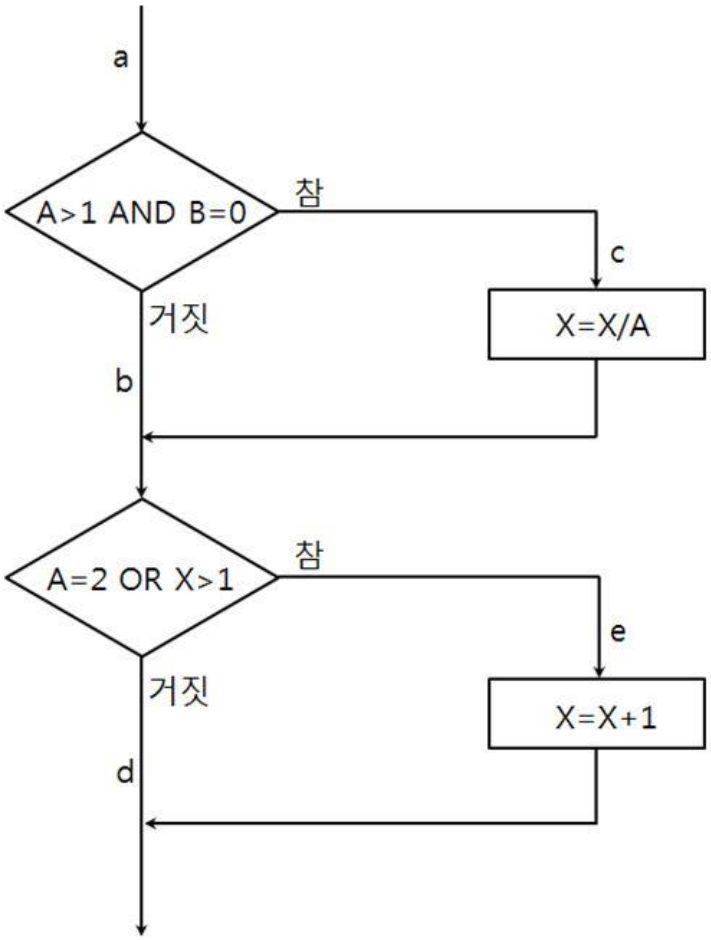
- 프로그램 소스코드의 논리적인 구조를 커버하도록 테스트 케이스를 유도하여 테스트를 실시하는 기법
- "유리상자 테스트", 혹은 "투명한 상자 테스트"
- 목표는 프로그램 내부 구조에 존재하는 모든 경로를 실행하는 것이지만 루프를 포함한 프로그램에서는 경로의 수가 대단히 많으므로 이를 모두 테스트하는 완전 경로 테스트(complete path test)는 타당성이 없을 뿐 아니라 불가능
- 현실적인 방법으로 테스트 케이스를 선정하는 기준이 필요하고, 프로그램을 커버하는 범위를 **커버리지(coverage)**라고 부름
- 다양한 커버리지 기법, 기본 경로 테스트 기법 등

문장 커버리지(Statement coverage) (1/2)

- 프로그램 내의 모든 문장들을 한번 이상 실행하도록 요구하는 기준
- 문장 커버리지를 만족하는 테스트 케이스를 결정하는 것은 수월하며 테스트를 수행하기도 쉽지만, 분기와 반복과 같은 프로그램 구조를 검사하는 경우에는 프로그램의 오류를 찾을 수 없는 경우도 많음

문장 커버리지(Statement coverage) (2/2)

- 경로 a-c-e를 지나는 테스트 케이스로 A=2, B=0, X=3을 선정한다면 모든 문장을 한 번씩 실행하는 문장 커버리지를 만족
- 그러나 첫 번째 조건문의 AND가 OR로 잘못 작성되거나 두 번째 조건문에서 X>1 대신에 X>0과 같은 오류가 발생할 때 문장 커버리지의 테스트 케이스로는 이를 발견할 수 없음
- X 값이 변하지 않는 경로 a-b-d 상에서 오류가 발생하여도 이를 발견하지 못함
- 일반적으로 사용하기에는 미흡한 기준



분기 커버리지(Branch coverage)

- 결정 커버리지(decision coverage)
  - 프로그램에 있는 각 결정문의 참과 거짓인 분기를 적어도 한 번 이상 실행시키는 것을 기준으로 하는 테스트 기법. 즉, 프로그램의 모든 분기들이 실행되도록 테스트 케이스를 선정
  - 분기 커버리지에 의한 두 개의 경로 a-c-e와 a-b-d 혹은 a-c-d와 a-b-e로 테스트 케이스를 구할 수 있음
  - 만일 a-c-d와 a-b-e의 경로를 통과하는 A=3, B=0, X=3과 A=2, B=1, X=1의 두 가지 테스트 케이스를 선택하는 것 대신에, a-c-e와 a-b-d의 경로를 선택하였을 때 X 값이 변하지 않는 a-b-d 경로 상에서 오류가 발생한다면 이 오류를 두 경로에 대한 테스트 케이스가 검출할 수 없음
  - 예를 들면, X>1의 조건이 X<1로 잘못 표기되었을 때 이와 같은 경우가 발생. 하나의 결정문이 두 개 이상의 복합 조건을 갖고 있기 때문.
  - 하나의 결정문이 여러 방향의 분기를 갖는다면 분기 커버리지는 모든 분기들을 커버하도록 테스트

조건 커버리지(Condition coverage) (1/2)

- 각 결정에 대한 분기를 커버하는 분기 커버리지에 비해서 결정 내에 존재하는 각 조건들의 **참**과 거짓을 한 번 이상 실행하도록 테스트 케이스를 작성하는 테스트 기법
  - for(k=1; i*j>m && k<=100; k++){ ... }에 대한 조건별 구분 1≤k≤100과 i*j>m
  - 조건 커버리지를 만족하는 테스트 케이스의 설정 범위는 다음의 다섯 가지이다.
    - 1≤k≤100, k<1, k>100, i*j>m, i*j≤m

조건 커버리지(Condition coverage) (2/2)

- [그림 8-10]에서는 A>1, B=0, A=2, X>1 등 4가지의 조건이 존재하는데, 이를 조건 커버리지에 맞도록 정리하면

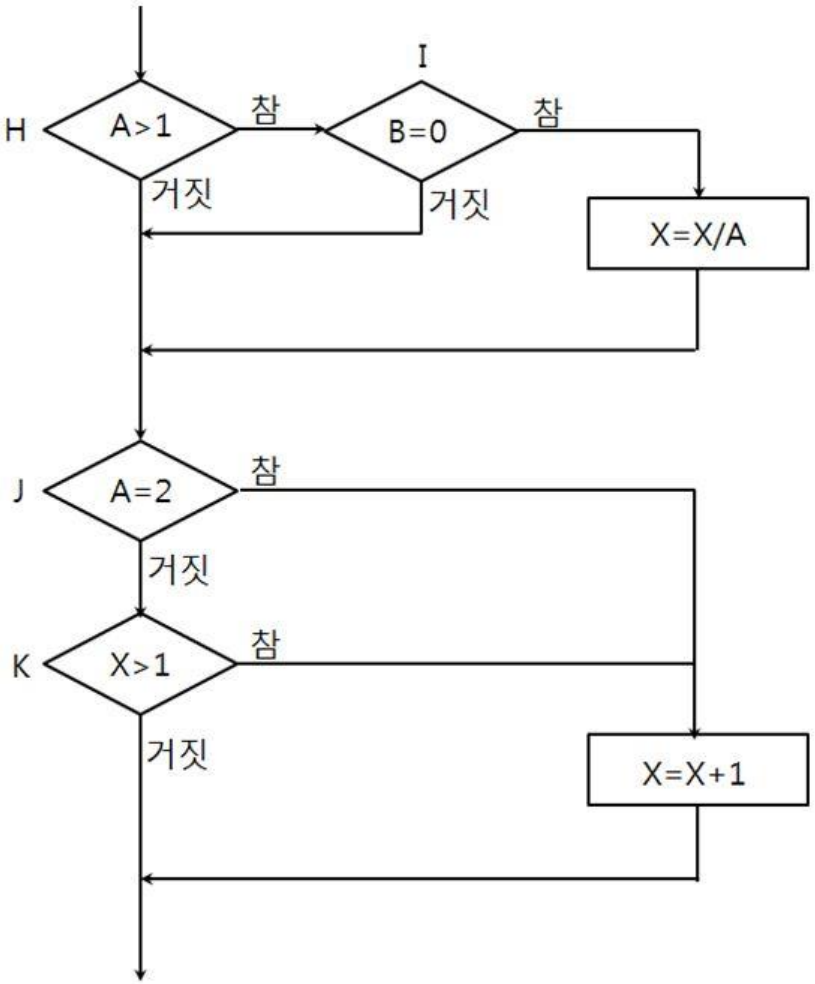
a 지점	b 지점
A>1	A=2
A≤1	A≠2
B=0	X>1
B≠0	X≤1

- 이들 8가지 조건을 커버하도록 A, B, X 값을 결정하면 여러 값들 중 다음 두 개의 테스트 케이스를 유도
  - ① A=1, B=0, X=3
  - ② A=2, B=1, X=1
- 4개의 결정 분기 중 경로 a-b-e에 해당하는 두 개의 결정 분기만 커버할 뿐 첫 번째 결정 분기의 참과 두 번째 결정 분기의 거짓에 해당되는 부분은 커버하지 못하는 문제점

결정/조건 커버리지 (1/2)

- 결정문 내에 존재하는 각 조건들의 참과 거짓을 모두 커버하고 각 결정문의 참과 거짓에 해당하는 모든 분기들이 모두 커버될 수 있도록 충분히 테스트 케이스를 만들어 나가는 기법

결정/조건 커버리지 (2/2)



- 다중 조건이 포함된 결정문을 단일 조건만으로 구성되도록 변환
- H와 I는 AND 관계이므로, H의 결과가 거짓일 때 I의 결과에 무관하게 거짓으로 판단
- J와 K는 OR 관계이므로 J 결과가 참이면 K의 결과와 관계 없이 참으로 판단
- 논리식 내부에 오류가 존재할 때 이를 발견하기 어려움
- 다른 조건들의 판단 결과에 상관 없이 미리 판단하는 것이 마스크

다중조건 커버리지(Multiple condition coverage)

- 조건과 조건 사이의 마스크 현상을 해결하기 위하여 각 결정문 내에 존재하는 조건들의 조합으로 나타날 수 있는 모든 결과를 커버할 수 있도록 테스트 데이터를 선정하는 기준

조건 번호	다중 조건
1	A>1, B=0
2	A>1, B≠0
3	A≤1, B=0
4	A≤1, B≠0
5	A=2, X>1
6	A=2, X≤1
7	A≠2, X>1
8	A≠2, X≤1

다중 조건을 모두 커버하는 테스트 케이스

① A=2, B=0, X=4 (조건 1, 5 커버)
② A=2, B=1, X=1 (조건 2, 6 커버)
③ A=1, B=0, X=2 (조건 3, 7 커버)
④ A=1, B=1, X=1 (조건 4, 8 커버)

테스트 케이스는 각각 a-c-e, a-b-d, a-b-e, a-b-d 경로를 테스트하는데, 경로 a-b-d는 중복 테스트되며, 경로 a-c-d는 테스트할 데이터가 존재하지 않음

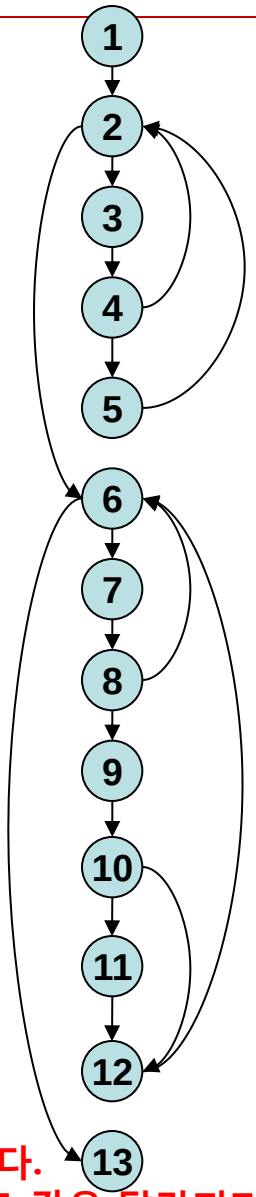
이러한 상황은 루프 구조를 갖는 경우, 다중조건 커버리지에 의한 테스트 케이스의 수가 경로의 수보다 적어진다는 것을 의미하며, 이 점이 다중조건 커버리지에서 고려해야 할 문제점

기본 경로 테스트(Basic Path test)

- 컴포넌트나 프로그램의 독립적인 경로 모두를 수행하는 것을 목적으로 하는 구조 테스트로, 화이트박스 테스트 기법
- 독립적인 모든 경로가 수행되면, 모든 문장은 적어도 한번은 모든 조건문들의 참과 거짓인 경우 모두에 대해 테스트
- 프로그램에서 경로의 수는 대개 프로그램의 크기에 비례하기 때문에, 모듈들이 시스템에 통합되면서 구조 테스트를 이용하는 것이 타당하지 않고, 단위 테스트 동안 주로 이용
- 루프를 가진 프로그램에서 가능한 경로의 조합의 수는 무한하므로 프로그램의 모든 경로의 조합을 테스트할 수는 없음
- 기본 경로 테스트의 목적은 프로그램의 각 독립적인 경로가 최소한 한번은 실행되도록 보장하는 것

프로그램과 흐름 그래프

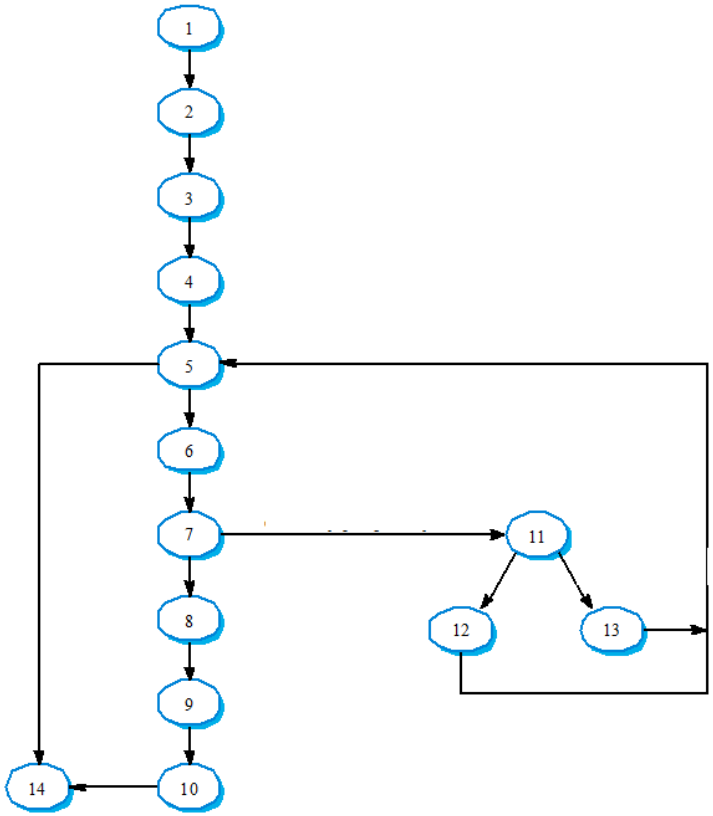
```
        void topologicalSort()
        {
1.              int top = 0, i, id;
                int inDegree[numOfTasks];
                int *stack[numOfTasks];
2.              for (id=0; id<numOfTasks; id++) {
3.                      int numOfPred = task[id].getInDegree();
                        inDegree[id] = numOfPred;
4.                      if (numOfPred == 0) {
5.                              stack[top] = id;
                                top++;
                        }
                } // end for
6.              for (i=0; i<numOfTasks; i++) {
7.                      top--;
                        id = stack[top];
                        topOrder.push_back(id);
                        Edge *e = task[id].getSucc();
8.                      while (e) {
9.                              int succId = e->getSuccTask();
                                inDegree[succId]--;
10.                             if (inDegree[succId] == 0) {
11.                                     stack[top] = succId;
                                        top++;
                                }
12.                             e = e->getSucc();
                        } // end while
                } // end for
13.     }
```



그래프 모양은 일반 문장이 노드에 포함되었는지에 따라 다를 수 있다.
일반적으로 조건, 루프가 아닌 문장은 무시해도 사이클로매틱 복잡도 값은 달라지지 않는다.

사이클로매틱 복잡도의 계산

방식	[그림 8-12]	[그림 8-13]
① V(G) = 조건문의 수 + 1	V(G) = 5 + 1 = 6	V(G) = 3 + 1 = 4
② V(G) = 에지의 수 - 노드의 수 + 2	V(G) = 17 - 13 + 2 = 6	V(G) = 16 - 14 + 2 = 4
③ V(G) = 영역의 수	V(G) = 6	V(G) = 4



경로의 수 4는 사이클로매틱 복잡도로 계산한 결과와 동일

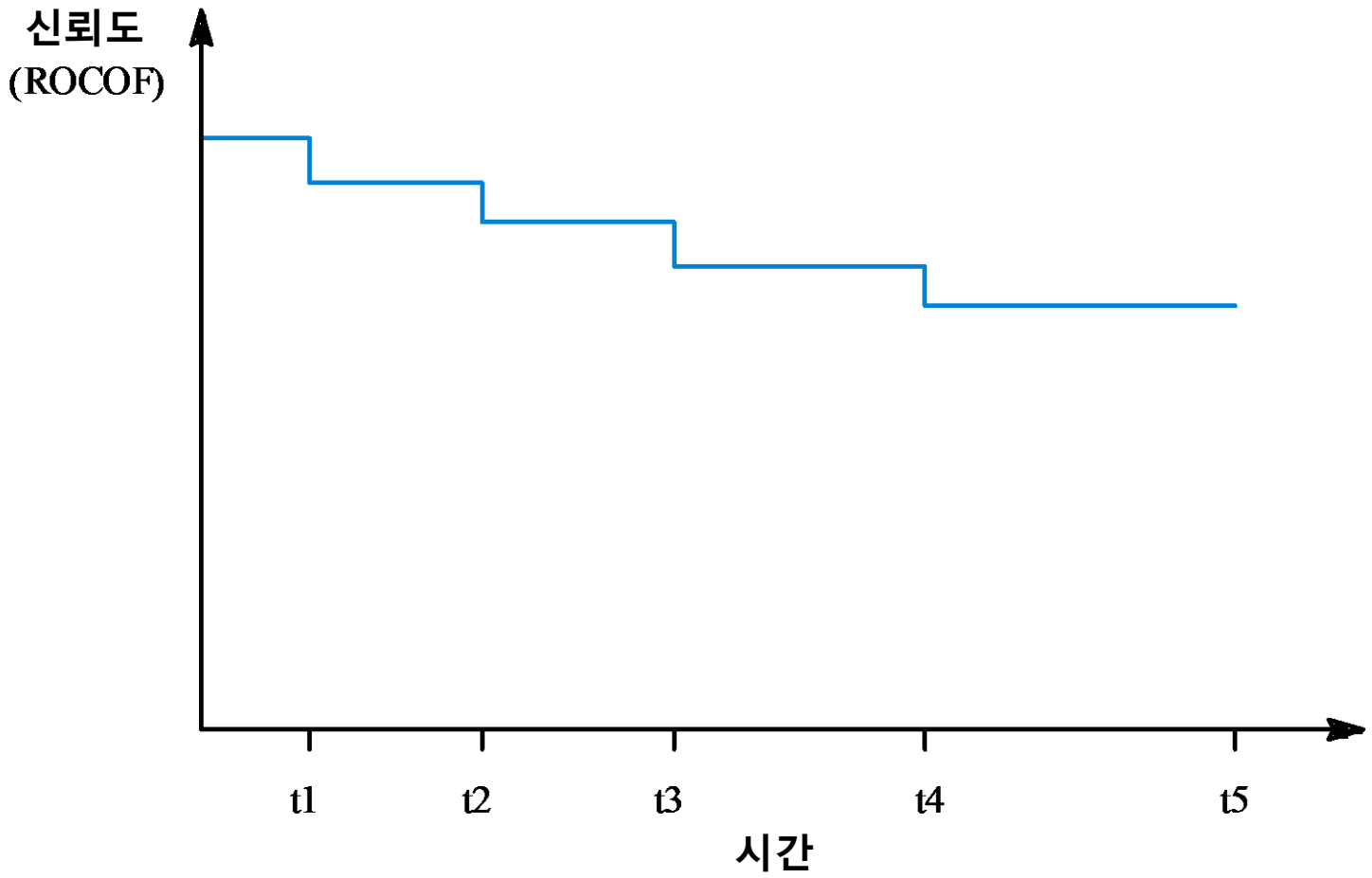
1, 2, 3, 4, **5**, **6**, 7, 8, 9, 10, 14
1, 2, 3, 4, **5**, 14
1, 2, 3, 4, **5**, 6, 7, 11, 12, **5**, ...
1, 2, 3, 4, **5**, 6, 7, 11, 13, **5**, ...

만일 위의 모든 경로가 수행되면, 프로그램 내의 모든 경로가 적어도 한번은 수행되고, 모든 분기가 참과 거짓인 조건에 대해서 수행될 수 있음

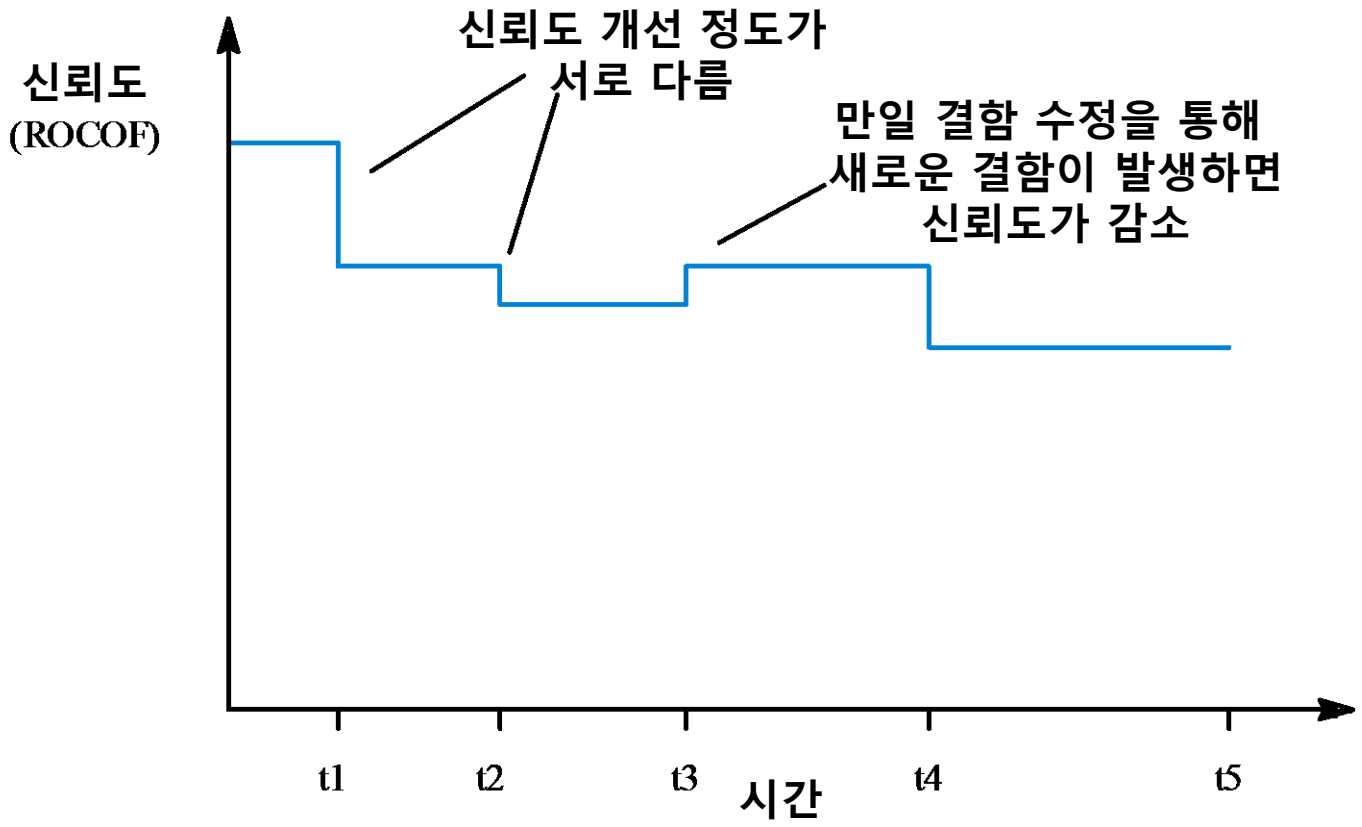
신뢰도 성장 모델

- **시스템 테스트에는 비용이 많이 들기 때문에, 가능하면 빨리 테스트를 중단하여 시스템을 지나칠 정도로 테스트하지 않는 것이 중요하다. 요구되는 수준의 시스템 신뢰도가 달성될 때 테스트를 중단**해야 하지만, 신뢰도를 예측해 보면 요구되는 **신뢰도 수준**이 결코 달성되지 못할 것이라고 예상될 수도 있다. 이 경우에 관리자는 소프트웨어 일부를 다시 작성하거나 시스템 계약을 다시 하는 어려운 결정을 내려야 한다.
- 대부분의 모델은 결함이 발견되고 제거됨에 따라 신뢰도가 빠르게 증가하는 지수형 모델이다. 신뢰도의 증가 정도는 점차 감소하다가 테스트 후반 단계에서 점점 더 적은 결함이 발견되고, 이 결함이 제거됨에 따라 안정기에 도달한다.

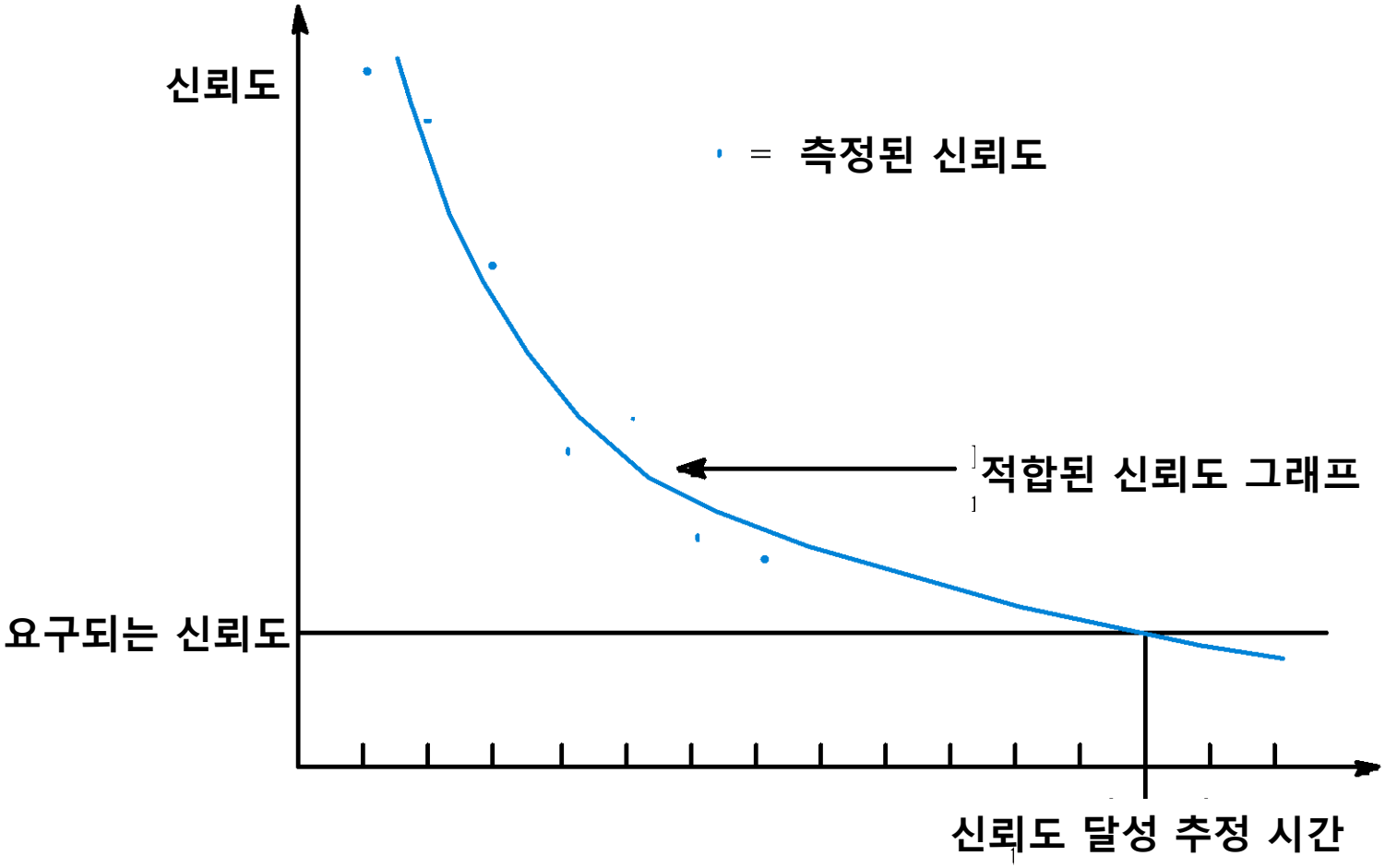
계단 함수 신뢰도 성장 모델



랜덤 계단 함수 신뢰도 성장 모델



신뢰도 추정



신뢰도와 가용도 측정

- 신뢰도: 주어진 시간 동안 오류 없이 작동할 확률
  - MTBF(Mean Time Between Failure) = MTTF + MTTR
    - MTTF(Mean Time To Failure): 평균 가동 시간 = 사용 시점부터 고장이 발생할 때까지의 가동 시간 평균, 평균 가동 시간
    - MTTR(Mean Time To Repair): 평균 수리 시간 = 고장이 발생하여 가동하지 못한 시간들의 평균
- 가용도: 주어진 시점에서 요구사항에 따라 운영될 확률
  - 총 운용 시간 중 정상적으로 가동된 시간의 비율

$$\text{가용도} = \frac{MTBF}{MTBF + MTTR} \times 100\% = \frac{MTTF}{MTTF + MTTR} \times 100\%$$

$$= \frac{MTTF}{MTBF} \times 100\%$$