Chap. 10 소프트웨어 테스팅

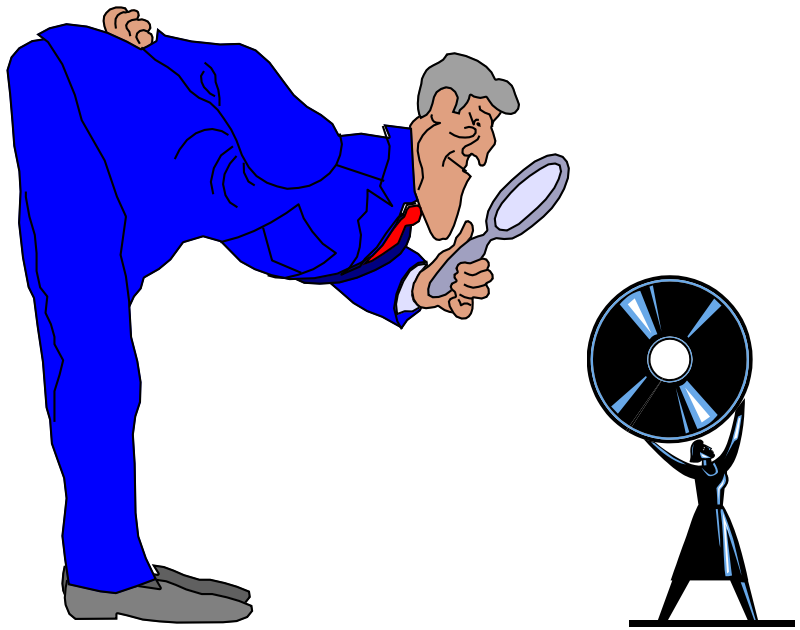
Software Engineering

©2014 SELAB

개요(1)

- 소프트웨어의 생명은 품질
  - 결함이 많은 소프트웨어는 상품 가치가 X

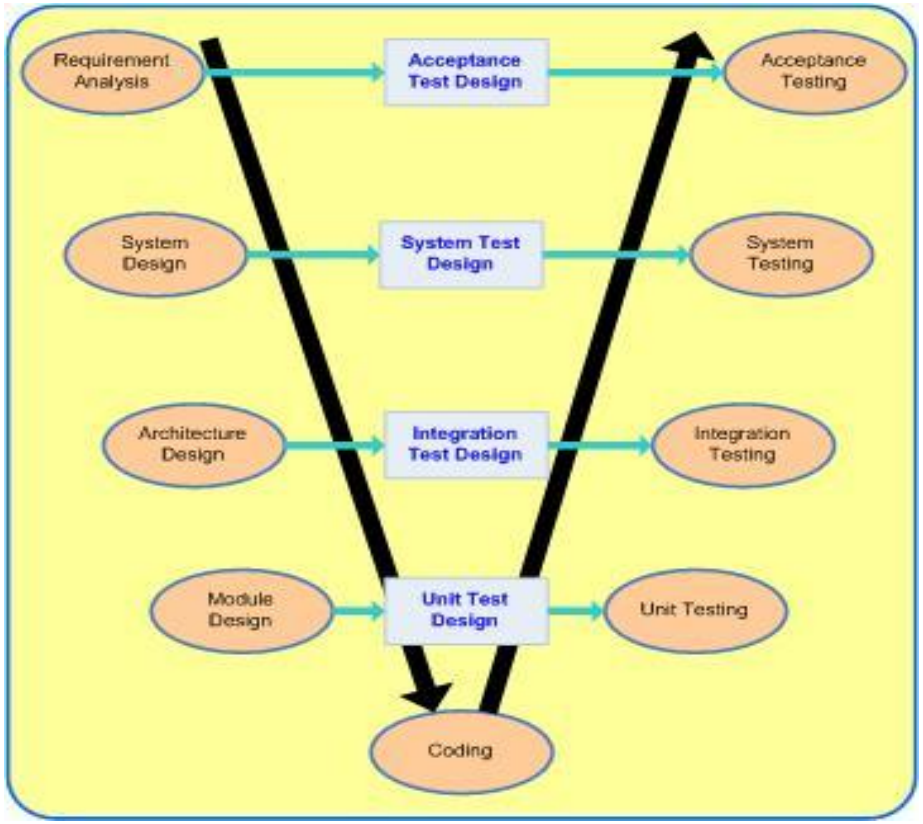
∴ 소프트웨어는 사용자에게 인도되기 전에 철저히 테스트 받아야 함



개요(2)

- 테스트 정의 및 목적
  - 정의
    - Myers : 테스트란 프로그램의 품질과 신뢰성을 향상시키기 위하여 에러를 발견한다는 예정으로 프로그램을 실행하는 과정
    - Conger : 에러를 찾으려는 의도로 프로그램을 실행하는 과정
    - Vliet : 소프트웨어에 있는 fault를 발견하기 위한 것
    - *프로그래밍이 완료된 다음에 계속해서 하는 작업이 아니라 소프트웨어 설계와 동시에 설계해야 될 작업*
    - *소프트웨어 품질 기준을 만족하는지를 확인하는 작업*
  - 목적
    - Myers : 프로그램은 에러를 포함하고 있다는 가정하에 테스트, 가능한 에러를 찾기 위해 테스트 수행
    - *정확하게 처리되지 않는 사양이나 기능 등의 결함을 발견하는 것*

개요(3)


Requirement Analysis
Acceptance Test Design
Acceptance Testing
System Design
System Test Design
System Testing
Architecture Design
Integration Test Design
Integration Testing
Module Design
Unit Test Design
Unit Testing
Coding

개요(4)

- 검증(Verification)
  - 검증은 시험자의 시각에서 시스템이 명세서대로 만들어졌는지를 점검하는 것
  - 각 단계에서 입력으로 제공된 제품들과 표준에 기준하여 정확성과 일치성을 보장하기 위한 것
- 확인(Validation)
  - 확인은 사용자의 시각에서 고객의 요구사항이 구현되었는지를 점검하는 것
  - 규정된 요구사항에 따르는지를 보장하기 위해 소프트웨어를 평가하는 과정
- 디버깅(Debugging)
  - 디버깅은 개발자의 시각에서 노출된 오류를 고치는 활동
  - 검증 및 확인 작업 결과 발견된 오류를 제거하는 것

테스트 종류

- **테스트 방법에 의한 분류**
  - 블랙 박스 테스트 : 소프트웨어 외부에서 그 기능, 성능을 테스트
  - 화이트 박스 테스트 : 소프트웨어 내면의 논리적 구조를 테스트
- **테스트 절차에 의한 분류**
  - 단위 테스트(모듈 테스트)
  - 통합 테스트
  - 시스템 테스트
  - 인수 테스트
  - 설치 테스트

1. 블랙 박스 테스트(Black-box Test) (1)

- 블랙 박스 테스트 (기능 테스트)
  - *소프트웨어의 내부는 고려하지 않고 주어진 입력에 요구되는 결과가 나오는가를 테스트*
  - 프로그램과 외부와의 인터페이스에 대해서 테스트가 이루어짐
  - 프로그램의 기능에 대해 초점을 맞추며, 주어진 입력 값들의 조합에 의해 요구되는 결과가 나오는가를 점검하는 것
  - 이 테스트는 프로그램 내부를 참조하지 않으며, 프로그램이 무슨 일을 수행하는가를 나타내는 설계 명세서 또는 요구사항 명세서를 참조하여 이루어질 수 있음
    - 프로그램의 논리 경로는 전혀 고려하지 않고 블랙 박스화
  - 기능 또는 명세서에 기초한 테스트라 불리기도 함
    - 소프트웨어가 수행할 특정 기능을 알기 위해서 각 기능이 완전히 작동되는 것을 입증하는 검사 ⇒ 기능 테스트
  - 프로그램의 what(무엇을 수행할 것인가)에 대한 실현도 조사
  - 문제점
    - 숨겨진 기능(명세에는 없으나 구현된 기능)을 테스트할 수 없음

1. 블랙 박스 테스트(Black-box Test) (2)

- **블랙 박스 테스트에 의해 발견될 수 있는 오류의 종류**
  - 부정확하거나 잘못된 기능
  - 인터페이스 오류
  - 데이터 구조나 외부 데이터베이스 접근 시 오류
  - 성능 오류
  - 초기화 또는 종료에 관한 오류

- **블랙 박스 테스트 기법 종류**
  - 동등(등가) 분할(equivalence partitioning)
  - 경계 값 분석(boundary value analysis)
  - 원인-결과 그래프 기법(cause-effect graphing)

2. 화이트 박스 테스트(White-box Test) (1)

- 화이트 박스 테스트 (구조 테스트)
  - *프로그램의 내부 구조를 참조하여 테스트를 수행하는 기법*
    - 프로그램 내부의 구성을 테스트하는 것
    - 모듈 안의 작동 상태를 자세히 관찰하는 방법
    - 프로그램 내부 구조에 대한 조사를 기초로 테스트 계획이 이루어짐
  - 프로그램을 참조하여 이루어지며, 구조 또는 코드에 기초한(code based) 테스트라 불림
    - 모듈의 원시 코드를 오픈 시킨 상태에서 원시코드의 논리적인 구조를 검사하여 테스트 케이스를 설계하는 방법 ⇒ 구조 테스트
  - 프로그램 코드를 검사하고, 프로그램을 실행시키지 않고도 프로그램의 독립적인 모든 경로의 수행, 논리적인 결정, 순환(loop)의 경계값 조사, 내부 자료 구조의 사용 등을 알 수 있도록 수행되며 이를 위한 테스트 데이터를 생성
  - 원시 코드의 모든 문장을 한번 이상 수행
  - 프로그램 리스트의 논리 경로를 보면서 프로그램의 how(어떻게 처리를 실현할 까)에 대한 실현도 조사
  - 철저한 화이트 박스 테스트는 많은 오류를 제거시켜 주지만, 모든 가능한 테스트 사례의 생성은 가능하지 않음
  - 문제점
    - 실종된 기능(명세에는 있으나 구현되지 않은 기능)을 검출할 수 없음

2. 화이트 박스 테스트(White-box Test) (2)

- 화이트 박스 테스트 기법 종류
  1) 모든 실행문의 수행 조사
     - 각 실행 가능한 문장의 실행을 검사하는 것은 코드의 품질과 이해도를 위해 요구됨
     - '실행문 적용'(statement coverage) 또는 '실행문 테스팅'(statement testing)이라 불림
     - 모든 수행 가능한 문장들이 단 한 번이라도 실행되도록 테스트 사례를 만들고, 요구되는 결과가 나오는지 검증해보는 것
  2) 분기점 조사
     - 모든 실행문에 대한 조사는 테스트에서의 가장 기본적인 요구이지만 조건문에 대한 검사는 매우 미약함
     - 충분한 테스트 사례를 설계하여 프로그램의 모든 조건을 만족하는 상황과 만족하지 않는 상황을 검증하는 것
  3) 조건 조사
     - 충분한 테스트 사례를 설계하여 조건문의 모든 조건식을 만족하는 경우와 만족하지 않는 경우를 검사하는 것

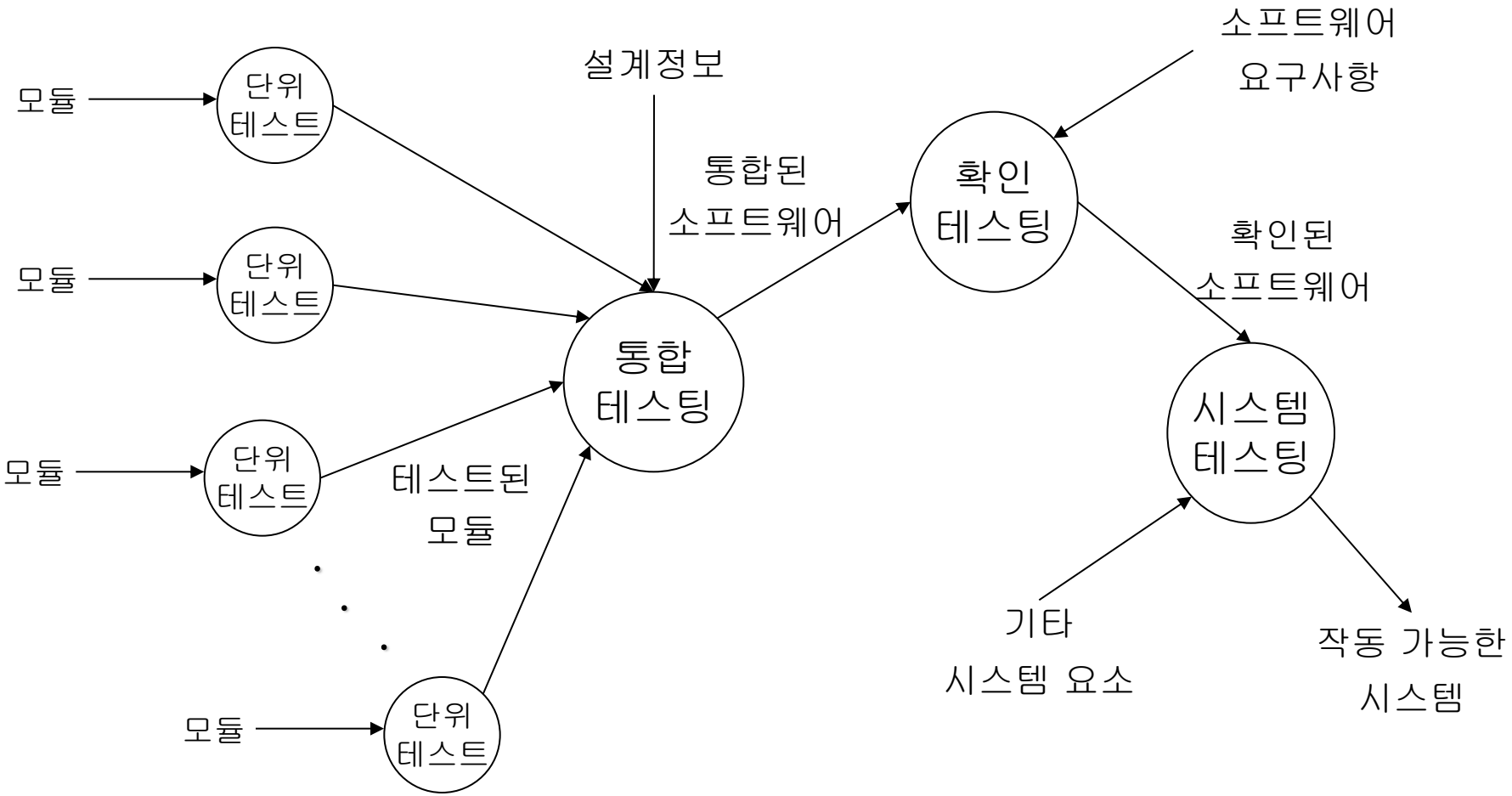
2. 화이트 박스 테스트(White-box Test) (3)

- 화이트 박스 테스트의 필요성
  - 블랙박스 테스트만 가지고는 프로그램의 모든 실행문이 실제로 실행되는지 알 수 없음
    - 모듈이 사용하고 있는 모든 변수와 상수, 서브루틴의 실행을 포함
    - 만약, 실행되지 않는 코드가 있는 경우 이를 이해하기 어려우며 유지보수하기 힘듦
    - 프로그램의 완전성에 대해 확신하기 어려움
  - 블랙박스 테스트만 가지고는 부작용(side effect)을 예측하기 어려움
    - 모듈이 실행되어 전역 변수의 값을 변하게 하는 것도 부작용의 예
    - 부작용은 일반적으로 모듈의 응집도가 약해서 발생
  - 화이트박스 테스트로 프로그램 내에 불필요한 기능이 있는 경우 이를 찾아낼 수 있음
  - 화이트박스 테스트로 프로그램에 type error가 있는 경우 이를 찾아낼 수 있음
    - 코드 검사에서도 찾아낼 수 있음

테스트 절차

- 테스트 과정
  - ① 테스트에 의하여 점검할 것이 무엇인지 결정
    - 예) 테스트의 목표 – 기능의 완벽성, 신뢰도
  - ② 테스트 방법 결정
    - 예) 검사 & 확인, 블랙 박스 테스트, 화이트 박스 테스트, 자동화 도구
  - ③ 테스트 케이스 개발
    - 테스트 자료, 시행 조건
  - ④ 테스트의 예상되는 올바른 결과 작성
  - ⑤ 테스트 케이스로 실행

테스트 절차(1)


모듈
단위 테스트
모듈
단위 테스트
모듈
단위 테스트
모듈
단위 테스트
설계정보
통합된 소프트웨어
테스트된 모듈
통합 테스팅
확인 테스팅
소프트웨어 요구사항
확인된 소프트웨어
시스템 테스팅
기타 시스템 요소
작동 가능한 시스템

테스트 절차(2)

단위 테스트	Unit Test	프로그램의 기본 단위인 각 모듈의 내부설계 명세에 맞도록 작동하는지 테스트
통합 테스트	Integration Test	복잡한 시스템의 경우 여러 모듈들이 모여서 서브시스템 형성 이들간의 인터페이스를 중심으로 테스트
확인 테스트	Validation Test	요구분석에 의해 작성된 요구를 소프트웨어가 만족하는가를 확인하는 과정
시스템 테스트	System Test	다른 시스템 요소들과 연결된 소프트웨어가 시스템 기능 및 성능 등의 사용자 요구를 만족하는가를 테스트
디버깅	Debugging	검사 단계에서 발생한 오류의 원인을 찾아 수정하는 과정
인수/설치	Acceptance/ Installation	사용자측 관점에서 소프트웨어가 요구사항을 충족하는가를 평가하고, 개발된 소프트웨어가 사용자 시스템에 설치된 후 정상적으로 작동하는가를 검사

1. 단위 테스트(Unit Test) (1)

- 단위 테스트 (Unit Test, Module Test)
  - 소프트웨어 설계의 최소 단위인 모듈의 검증에 초점
  - 상세 설계 명세서를 지침으로 중요한 제어 경로들을 테스트
  - 코드에 중점을 두어 내부 논리 흐름의 점검
  - 여러 모듈들이 병렬적으로 테스트 가능
  - 모듈 내에서 모든 문장이 한번씩 수행되는가를 보증하기 위하여 실행
  - *화이트 박스(white-box) 테스트*를 이용하여 프로그램의 제어 흐름을 확인
  - 단위 테스트에서 각 모듈에 대해 인터페이스, 지역 자료구조, 경계 조건, 제어 흐름 및 오류 처리 등 다양한 면을 확인하여야 함

1. 단위 테스트(Unit Test) (2)

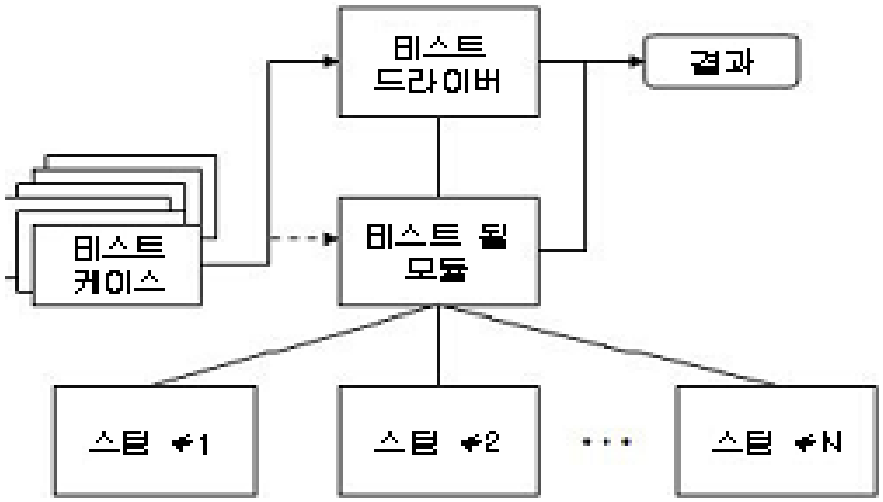
- 단위 테스트 시 고려사항 (1)
  - 모듈 인터페이스(Module interface) : 테스트되는 모듈의 내부 또는 외부로 전달되는 정보를 보증
    - 입력 변수의 개수가 정확한지 확인
    - 입력 변수의 타입이 정확한지 확인
    - 변수의 단위가 정확하고 호출 모듈과 호출되는 모듈에서 일치하는지 확인
    - 입력 변수의 순서가 정확한지 확인
    - 읽기 전용인 입력 변수가 모듈 내부에서 변경되었는지 확인
    - 모듈간 전역 변수에 대해 정확히 사용하고 있는지 확인
    - 라이브러리 모듈에 대한 호출에서 입력 변수가 정확한지 확인
    - 파일에 대한 사용 방법이 정확한지 확인
  - 지역 자료 구조(Local data structure) : 알고리즘이 실행되는 모든 단계 동안에 값이 변하지 않게 일시적으로 저장된 데이터를 보증
    - 부정확 하거나 일치하지 않는 타입이 존재하는지 확인
    - 변수에 대한 초기화가 부정확하거나 부정확한 디폴트값을 사용하였는지 확인
    - 변수 이름이 부정확하게 타이핑 되었거나 컴파일러에 의해 잘리지 않는지 확인
    - 불일치한 데이터 타입을 이용하였는지 확인
    - 수식의 계산에서 언더플로우 또는 오버플로우가 발생하지 않는지 확인

1. 단위 테스트(Unit Test) (3)

- 단위 테스트 시 고려사항 (2)
  - 독립 경로(Independent path) = 기본 경로
    - 제어구조를 통하여 기본이 되는 경로들이 모듈 안의 모든 문장들을 적어도 한번 실행하는지 테스트
  - 에러 처리(Error handling) : 에러 처리시에 발생할 수 있는 잠재적인 에러를 고려
    - 오류 메시지가 이해하기 쉬운지 확인
    - 오류 처리가 오류 클래스와 일치하지 않는지 확인
    - 오류 조건이 오류 처리 전에 원하지 않는 시스템 개입을 야기시키지 않고, 정상처리과정에 반영되는지 확인
    - 예외 처리가 정확한지 확인
    - 오류에 대한 서술이 오류 위치를 찾는데 필요한 충분한 정보를 제공하는지 확인
  - 경계 조건 테스팅
    - 경계조건에 대한 잘못을 검사
    - 단위 테스트에서 가장 나중에 테스트

1. 단위 테스트(Unit Test) (4)

- 모듈은 독립된 프로그램이 아니므로
  -> 단위 테스트를 위해 **테스트 드라이버** 또는 **스텁**이 만들어져야 함
  - 테스트 드라이버 (Test driver) : 가짜 상위 모듈
    - 테스트될 모듈을 호출하는 모듈
    - 가상의 주 프로그램 역할
    - 테스트 케이스 자료를 입력 받고, 검사를 위해 연관된 결과를 출력하는 제어 프로그램
  - 스텁 (Stub) : 가짜 하위 모듈
    - 테스트하려는 모듈이 호출하는 가상모듈
    - 가상의 부 프로그램 역할
    - 모듈간의 통합 검사를 위해 일시적으로 필요한 조건만을 가지고 임시로 제공되는 시험용 모듈
  - 테스트 드라이버와 스텁은 테스트를 위한 가상 프로그램 => ∴ 완성된 제품에는 포함되지 않음
  - 드라이버 모듈과 스텁 모듈의 설계에는 많은 인력과 비용이 요구되며 테스트 비용이 많이 든다는 단점이 있음



2. 통합 테스트(Integrated test) (1)

- 통합 테스트(Integrated test)
  - 통합 테스트의 필요성
    - 단위 시험을 충분히 수행한다 하여도 모든 설계 오류를 발견할 수는 없음
    - 모듈 각각이 완전하게 보여도 이들 모듈을 통합하면 서로 불일치하여 발생하는 오류, 전역 변수에 대한 부정확한 사용, 한 모듈에서의 작은 오차가 여러 모듈에서 축적되어 큰 문제를 유발하는 등 여러 가지 오류들이 발생할 수 있음
    - 결국 모듈들을 통합한 후 프로그램에 대한 테스트를 반드시 수행하여야 하는데 이를 **통합 테스트**라 함
  - 전체 시스템을 이루는 모듈들을 모아서 시스템이 요구된 기능을 제대로 수행하는지를 점검
  - 통합 테스트에서 우선 일 부분 모듈을 통합하여 테스트한 후 또 다른 일 부분 모듈을 통합시켜 통합 테스트를 수행하는 방법으로 점진적으로 모듈을 통합하여 전체 모듈이 모두 통합될 때까지 통합 테스트를 수행하는 것이 바람직 => **점진적(incremental) 통합 테스트**
  - 모듈 간의 인터페이스와 연관된 오류를 밝히기 위한 검사 방법
  - 기능 점검을 위해 *블랙 박스(Black-box)* 테스트케이스 설계 기법을 위주로 사용하여 검사
  - 종류
    - top-down test (하향식)
    - bottom-up test (상향식)
    - big bang test
    - sandwich test

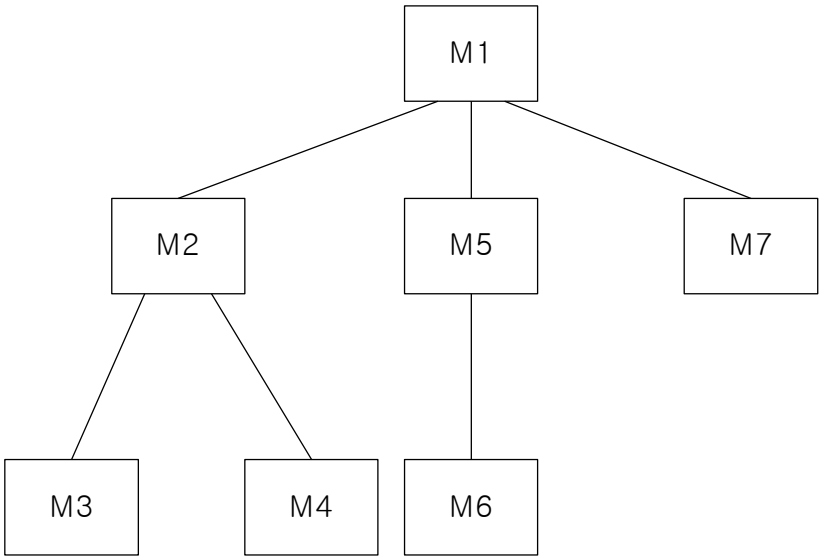
2. 통합 테스트(2)

- Top-down test (하향식 테스트)
  - 프로그램의 상위 모듈에서 하위 모듈 방향으로 통합하여 검사
    - 시스템의 최상위에 있는 주 프로그램 모듈을 먼저 테스트
    - 이 모듈이 직접 호출하는 다음 레벨의 모듈을 테스트하여 점차 하위 레벨로 테스트를 이동하는 방법
  - 스텁 모듈 필요, 드라이버 모듈 필요 없음
  - 하향식 통합은 깊이-우선(depth-first)과 넓이-우선(breadth-first) 방법으로 나눌 수 있음
    - 깊이-우선 통합의 기본 원칙은 한 모듈을 선택한 후 우선 그 모듈의 하위 모듈을 통합는 것
  - 장점
    - 하위 모듈에서 테스트가 완료된 상위 모듈을 이용

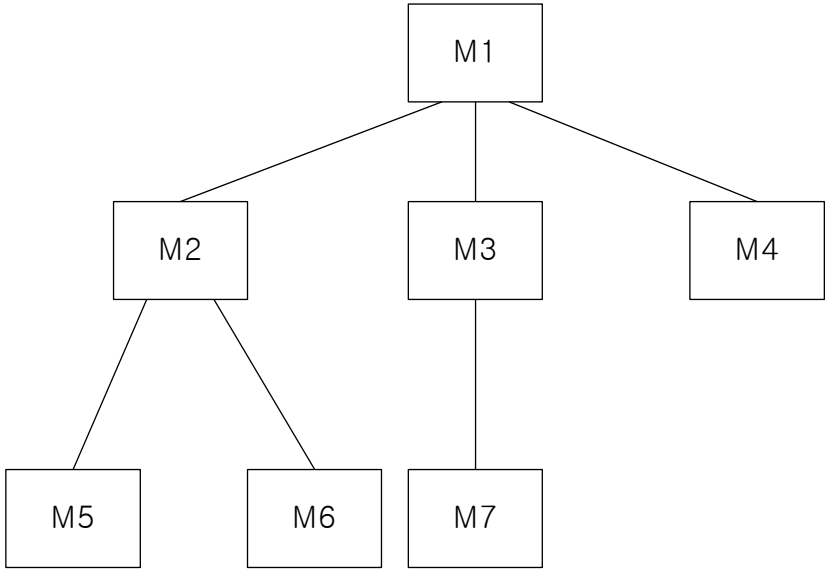
      → 실제 가동 환경과 유사한 환경에서 테스트가 이루어짐
    - 테스트의 최초 단계에서 인터페이스 검증이 가능
  - 단점
    - 테스트 초기에 병행 작업이 곤란
    - 스텁(stub)이 필요 → 스텁의 작성에 많은 노력이 필요

2. 통합 테스트(3)

- 모듈 통합 순서



(a) 깊이-우선 통합



(b) 넓이-우선 통합

2. 통합 테스트(4)

- Bottom-up test (상향식 테스트)
  - 프로그램의 하위 모듈에서 상위 모듈 방향으로 통합하여 검사
    - 하위 레벨에 있는 모듈들을 클러스터로 결합하여 특정 소프트웨어의 부분 기능을 수행
    - 테스트 케이스의 입출력을 조정하기 위해 드라이버 작성
    - 클러스터 검사
    - 드라이버를 제거하고 클러스터를 상위 모듈과 결합
  - 드라이버 모듈 필요, 스텁 모듈 필요 ×
  - 장점
    - 드라이버 모듈의 설계가 상대적으로 쉬움
    - 테스트 초기부터 병행 작업이 가능(여러 팀이 동시에 테스트를 수행할 수 있음)
    - 모듈 단위의 철저한 테스트
  - 단점
    - 프로그램 전체의 윤곽을 통합 테스트 최후 단계에서 볼 수 있음
      - 전체적 윤곽을 초기에 볼 수 없으면 고객의 요구사항과 위배되는 "큰" 오류를 일찍 발견할 수 없음
    - 마지막 순간까지 독립된 소프트웨어의 형태를 가지지 못함
    - 인터페이스의 테스트가 가정인 상태로 행해짐
  - 실제 통합 테스트에서 단순한 하향식 또는 상향식 방법을 이용하는 것이 아니라 흔히 하향식과 상향식 방법을 결합하여 테스트를 수행

2. 통합 테스트(5)

- Big bang test
  - 모든 모듈에 대하여 각각 단위 테스트를 실시하여 단위 테스트 종료 후 한꺼번에 결합
  - 상호간의 인터페이스에 대하여는 고려하지 않음
  - 대규모 시스템에서는 비효율적
    - 한꺼번에 결합하여 프로그램을 테스트하여 이상한 결과가 생기는 경우, 어느 모듈이 원인인가를 파악하기 어려움 → 모든 모듈 조사 필수
  - 소규모적인 프로그램에서 모듈 구조가 잘 설계되어 인터페이스가 번잡하지 않을 경우 사용
- Sandwich test
  - 상향식 테스트 장점 + 하향식 테스트 장점
  - 양방향에서 단위 테스트를 실시하여 중간지점에서 결합

3. 확인 테스트(Validation test)

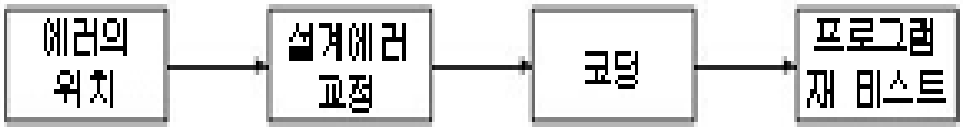
- 확인 테스트(Validation test)
  - 사용자 요구사항에 대한 만족여부를 검사하는 *블랙박스(black-box) 테스팅*을 통하여 이루어짐
  - Barry Boehm의 정의[Boehm, 1984]
    - Verification(검증) : Is the software right? (제품을 올바르게 구축했는가?)
      - 과정, 프로세스에 중점
      - 각 단계가 정확하게 수행했는지를 결정하는 프로세스(요구사항 명세대로 개발이 이루어지고 있는지 검사)
      - 각 단계가 종료된 후 수행(통합 테스트 때)
    - Validation(확인) : Is the right software? (올바른 제품을 구축하였는가?)
      - 제품에 중점
      - 사용자가 원하는 대로 동작하는지 검사(소프트웨어가 명세를 모두 만족하는지를 검사)
      - 통합 테스트 후 사용자의 요구가 제대로 반영되었는지 검사(제품이 고객에 인도되기 전에 수행)

4. 시스템 테스트(System test)

- 시스템 테스트(System test)
  - 시스템의 모든 요소들이 적합하게 통합되어 개발된 소프트웨어가 해당 시스템에서 정확하게 수행되는가를 검사하는 방법
  - 서로 다른 목적을 가진 일련의 test들로 이루어짐
    - **회복(recovering) 테스팅** = 복구 테스팅
      - 고장 발생시 미리 제안된 시간 안에 회복되는지 테스트
    - **안전(security) 테스팅** = 보안 테스팅
      - 시스템 외부로부터의 보안기능을 갖고 수행되는 테스트
    - **스트레스(Stress) 테스팅** = 강도 테스팅
      - 시스템에 과부하가 걸려있는 상황을 테스트
    - **성능(performance) 테스팅**
      - 시스템의 실행 시간, 응답시간, 처리능력 등 목표 성능을 만족하는지 테스트

5. 프로그램 디버깅(Program debugging)

- Testing
  - 프로그램 에러의 존재를 발견하는 과정
- Debugging
  - 프로그램 내 에러 발생위치를 찾아서 그 에러를 제거하는 과정
  - 테스팅 결과에 따른 연속 과정
- Debugging 단계



6. 인수/설치

- 개발팀 내에서 기본적인 테스트가 완료되면 사용자들로 하여금 테스트를 수행하게 함
  - 알파(Alpha) 테스트
    - 개발자 환경에서 사용자들이 소프트웨어를 사용하며 시험하고 문제점 점검
    - 사용자가 개발 환경에 와서 테스트하는 것
    - 개발자의 통제하에서 수행되고, 이 때 개발자는 사용자의 어깨너머로 테스트를 지켜 보게 됨
    - 오류는 사용자 또는 개발자에 의해 기록됨
    - 알파 테스트를 위한 소프트웨어를 α 버전이라 부름
  - 베타(Beta) 테스트
    - 알파 테스트가 성공적으로 이루어지면 베타 테스트가 이루어짐
    - 사용자 환경에서 사용자가 일정기간 직접 사용하면서 문제점을 기록하여 추후 반영될 수 있도록 개발자에게 통보
    - 개발자팀이 소프트웨어를 사용자에게 배포하여, 사용자가 자신의 컴퓨터 환경 또는 실제 상황에서 수행하는 테스트

테스트 케이스(Test case) (1)

- 테스팅 과정에서 효과적인 테스트 케이스 설계가 중요
  - 프로그램 내의 모든 에러를 찾는 완벽한 테스팅은 불가능

    ∴ 불완전한 테스팅을 할 수 밖에 없음

    ∴ 테스팅 과정은 가능한 불완전성을 줄이는 데 초점이 맞추어짐

    ⇒ *"가장 많은 에러를 검출할 수 있는 최소한의 테스트 케이스는?"*

테스트 케이스(Test case) (2)

- 테스트 케이스(Test case)
  - 테스트를 위해 표본(sample)으로 입력되는 데이터
    - 프로그램의 어느 동작 상태를 조사하기 위하여 여러 가지 테스트 데이터를 경우에 따라 나누어 설정한 것
    - 테스트 데이터와 함께 기대되는 출력 결과를 포함하는 것이 일반적
  - 훌륭한 테스트 케이스는 아직까지 발견하지 못한 오류를 찾을 수 있는 확률을 높여줌
  - 설계방법
    - **블랙 박스 테스트** : 외부 명세에 근거한 방법
    - **화이트 박스 테스트** : 내부 논리적 명세에 근거한 방법

테스트 방법 비교

내용 \ 종류	블랙 박스 검사	화이트 박스 검사
검사 종류	동치 분할 경계 값 분할 원인-결과 그래프 기법 비교 검사	기초 경로 검사 조건 검사 데이터 흐름 검사 루프 검사
처리	프로그램의 외부 명세에 근거 (프로그램의 기능 명세서)	프로그램의 내부 명세에 근거 (원시 프로그램 리스트)
기준	What (무엇을 수행하는가?)	How (어떻게 처리되는가?)
검사	기능 검사	구조 검사
시점	검사단계 후반부에서 수행	검사단계 전반부에서 수행
구조	제어구조 무시 (정보 영역에 초점)	정보 영역보다 제어 구조 중시

테스팅 도구

- 소프트웨어 테스트
  - 소프트웨어 개발에 소요되는 총 노력의 40% 이상을 차지하기도 함
  - 정확성을 유지하면서 테스트 시간을 줄일 수 있는 도구가 매우 중요

- 자동 테스트 도구
  - 정적 분석기(static analyzers)
  - 코드 감사기(code auditors)
  - 테스트 파일 생성기(test file generators)
  - 단정 처리기(assertion processors)
  - 테스트 검증기(test verifiers)
  - 테스트 장치(test harness)