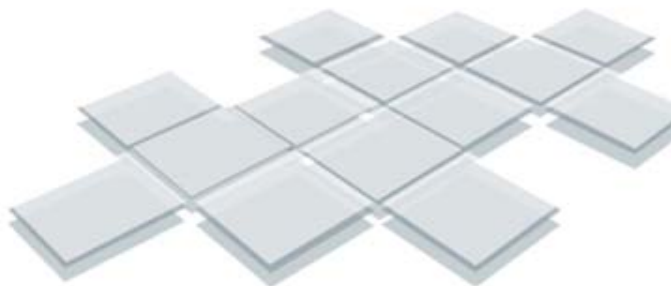
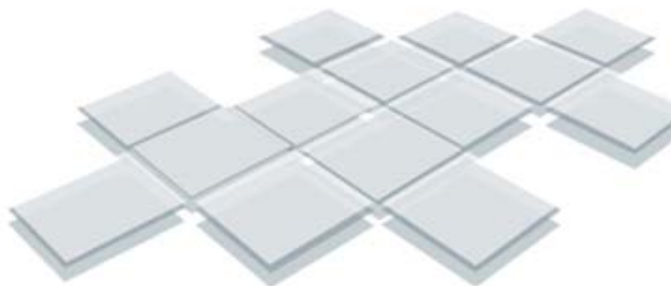
소프트웨어 테스팅 개요 및 기법



목차

- 소프트웨어 테스트 개요
- 소프트웨어 테스트 중요성
- 소프트웨어 테스트 기법
  - 명세 기반 테스트 기법
  - 코드 기반 테스트 기법
  - 오류 기반 테스트 기법
  - 상태 기반 테스트 기법
  - 객체지향 기반 테스트 기법
  - 컴포넌트 기반 테스트 기법
- 소프트웨어 테스트 충분성 기준
- 생명주기 테스트



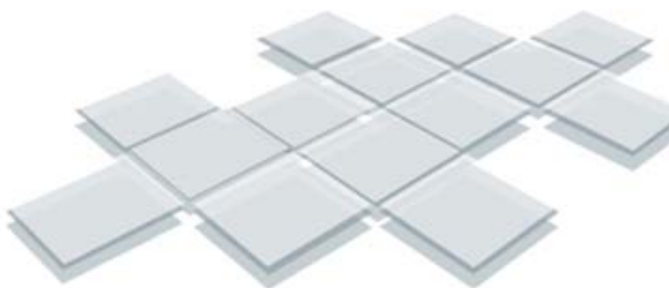
소프트웨어 테스트 개요

- 테스팅이란 응용 프로그램 또는 시스템의 동작과 성능, 안정성이 요구하는 수준을 만족하는 지 확인하기 위해 결함을 발견하는 메커니즘

  - 테스팅에 대한 오해
    - 테스팅과 디버깅은 같다
      - 테스팅 : 소프트웨어에 내재되어 있는 오류를 드러낸다
      - 디버깅 : 결함의 원인을 밝히는 개발 활동, 코드를 수정하고 올바른 작동 확인
    - 테스팅은 소프트웨어를 실행하면서 수행하는 테스트 작업만을 말한다
      - 일부분에 불과하며 테스팅 활동 전부를 나타내지 않는다
      - 테스트 활동은 테스팅 계획, 제어, 테스트 조건의 선택, 테스트 케이스 설계, 테스트 수행 결과 점검, 테스트 완료 및 통과 조건 평가, 테스트 프로세스와 테스트 중인 시스템에 대한 리포트, 마무리 또는 마감과 같은 일련의 활동을 포함

소프트웨어 테스트 개요

- 테스트의 목적
  - 남아 있는 결함 발견
  - 명세 충족 확인
  - 사용자 및 비즈니스의 요구 충족 확인
  - 결함 예방
  - 품질 수준에 대한 자신감 획득과 정보 제공
  - 비즈니스 리스크를 감소시키는 정보에 근거한 조언 제공
  - 개발 프로세스 점검, 이슈 제기
  - 논리적 설계의 구현 검증
  - 시스템과 소프트웨어가 적절히 동작함을 확인

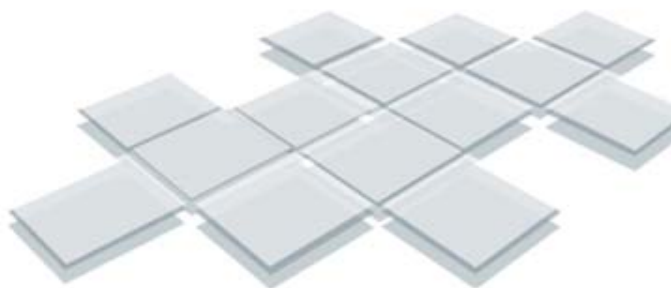


소프트웨어 테스트 개요

- 테스팅의 일반적인 원리
  - 원리 1 – 테스팅은 결함이 존재함을 밝히는 것
    - 소프트웨어의 잔존하는 결함을 간과할 가능성은 줄이지만, 결함이 전혀 발견되지 않는 경우라도 결함이 없이 완전하다는 것은 증명 불가.
  - 원리 2 – 완벽한 테스팅(Exhaustive testing)은 불가능
    - 모든 가능성을 테스팅 하는 것은 지극히 간단한 S/W를 제외하곤 불가능.
  - 원리 3 – 개발 초기에 테스팅 시작
    - 개발의 시작과 동시에 테스트를 계획, 전략적으로 접근하는 것을 고려.
    - 요구사항 분석서와 설계서 등의 개발 중간 산출물을 분석하여 테스트 케이스 도출 등 각 단계별 테스트 목표에 따라 테스팅.
  - 원리 4 – 결함 집중(Defect clustering)
    - 적은 수의 모듈에 대다수의 결함이 발견되는 등, 일정 모듈에 결함이 집중되는 현상을 발견할 수 있다.
      - E.g., 자체적으로 복잡한 구조를 가진 모듈, 크기가 큰 모듈, 난이도가 높은 모듈 등.

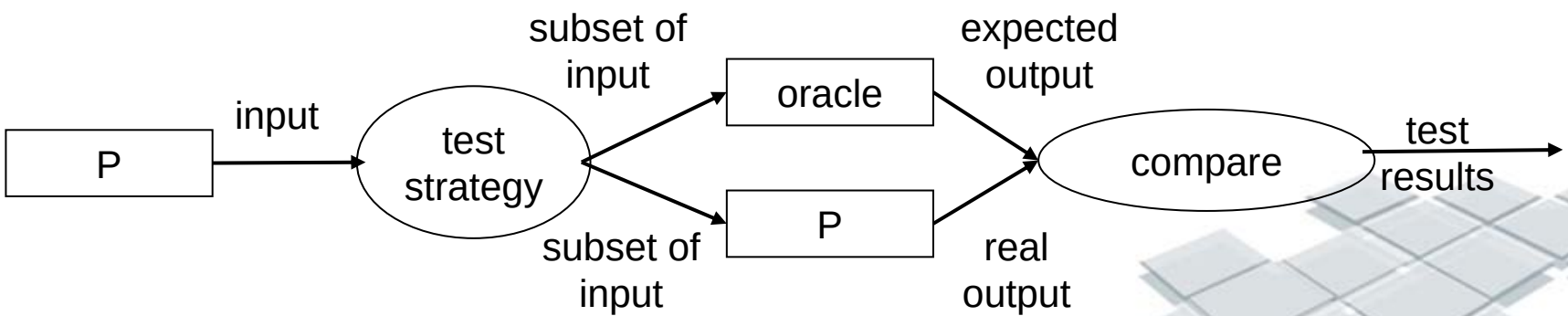
소프트웨어 테스트 개요

- 테스팅의 일반적인 원리
  - 원리 5 – 살충제 패러독스(Pesticide paradox)
    - 동일한 테스트가 반복적 수행되는 경우 결국 동일한 테스트 케이스들로는 더 이상 새로운 버그를 찾아내지 못함.
    - 잠재된 보다 많은 결함 발견을 위해 테스트 케이스를 정기적 리뷰, 개선.
  - 원리 6 – 테스팅은 정황(Context)에 의존적
    - 테스팅은 정황에 따라 다르게 진행된다.
      - 안전-최우선 S/W와 전자 상거래 사이트의 테스트는 다르다.
  - 원리 7 – 오류-부재의 궤변(Absence-of-errors fallacy)
    - 개발된 시스템이 사용자 또는 비즈니스의 요구를 충족시키지 못한다면, 결함을 찾고 결함을 제거하여도 품질이 높다고 볼 수 없다.

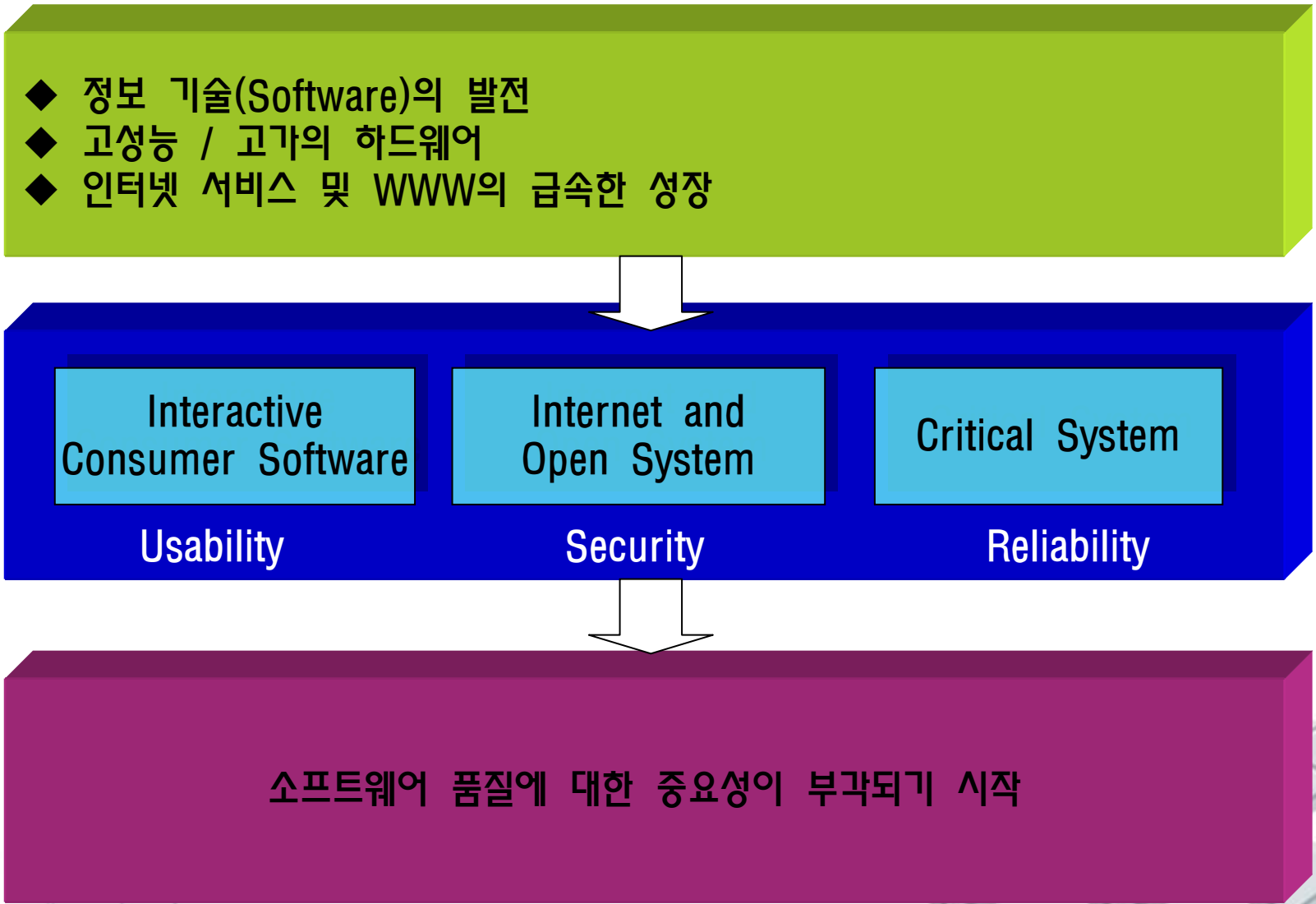


소프트웨어 테스트 개요

- 테스트 프로세스의 주요 활동
  - 계획과 제어(Planning and control)
  - 분석과 설계(Analysis and design)
  - 구현과 실행(Implementation and execution)
  - 완료 조건의 평가와 리포팅(Evaluating exit criteria and reporting)
  - 테스트 마감 활동(Test Closure activities)

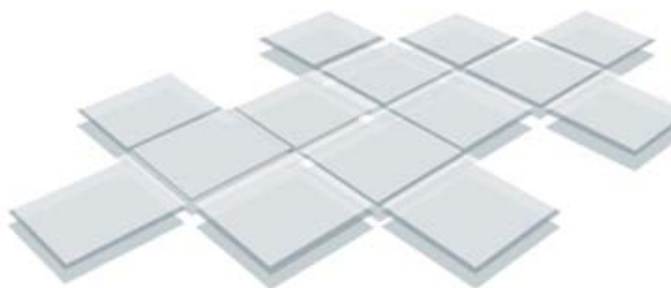


소프트웨어 테스트 중요성



소프트웨어 테스트 중요성

- 부적합한 S/W 평가 기반으로 인한 손실
  - S/W 개발비의 80%가 결함 보완 과정에 소요
    - SW의 수명단축과 복잡성 증대로 경제적 지출은 계속 증가.
  - S/W의 결함으로 매년 GDP의 0.6%의 손실 발생 (미국)
    - 소프트웨어의 신뢰도 저하에 따른 부차적 손실(브랜드 이미지 저하, 고객의 충성도 저하, 매출 및 시장 점유율 하락 등)은 고려 하지 않은 것으로 유, 무형으로 더 많은 손실이 발생하고 있다.
  - 결함의 절반 이상은 개발 중에 발견되지 않음
    - 제품 출시 이후 고장 발생으로 유지보수 비용 증대
  - SW 개발 사업자는 충분하게 검증된 품질평가기술 확보시급
    - 향상된 평가 기술도입만으로 품질문제 해결 안됨(객관적 검증)



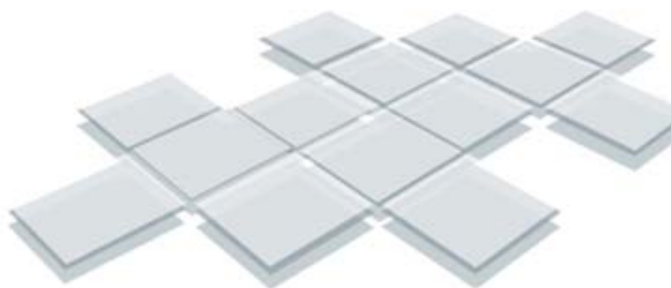
소프트웨어 테스트 중요성

- 부적합한 테스트로 인한 실제 사고 사례
  - 96년도 아리안(Ariane) 로켓 발사 30초 후 공중폭발
    - 로켓 제어 소프트웨어의 내부적 결함으로 5,000억원의 손실 발생
  - 일본 위성의 수명 단축
    - 두 번의 로켓 발사에 실패한 후 2001년도 발사에 성공한 위성이 지상시스템 SW 신뢰성이 문제되어 수명 크게 단축될 전망 보고
  - 국내에서 위성 수신 셋톱 박스 판매 후 리콜
    - 소프트웨어 내부적 결함으로 50,000여대 리콜 후 수리보상사례
  - AECL사의 방사선 치료기 테락(Therac)-25의 방사능 과다 노출로 인한 사망
    - 방사선 치료기인 테락은 이전 모델인 테락-20을 기반으로 만들어 이전 모델이 무사고였으므로 코드를 생략, 사고가 발생.
    - 근거 없는 S/W 신뢰가 가져온 엄청난 결과로, 기준치 100배의 방사능 노출

소프트웨어 테스트 기법

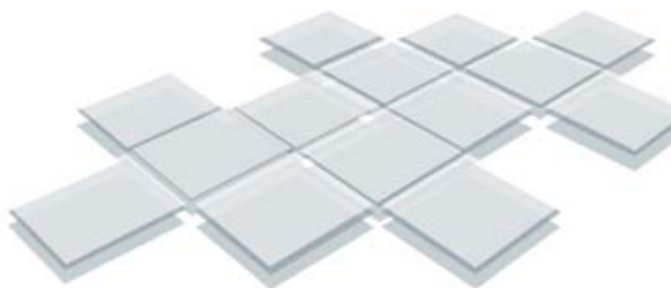
- 명세기반 테스트 기법
  - 주어진 명세(일반적으로 모델의 형태)를 빠뜨리지 않고 테스트 케이스화하는 것을 의미하고 해당 테스트 케이스를 수행해서 중대한 결함이 없음을 보장하는 것.

  - 명세기반 테스트 기법의 종류
    - 동등 분할 (Equivalence partitioning)
    - 경계 값 분석 (Boundary value analysis)
    - 결정 테이블 테스팅 (Decision table testing)
    - 유스케이스 테스팅 (Use case testing)
    - 조합 테스팅 (Pairwise testing)



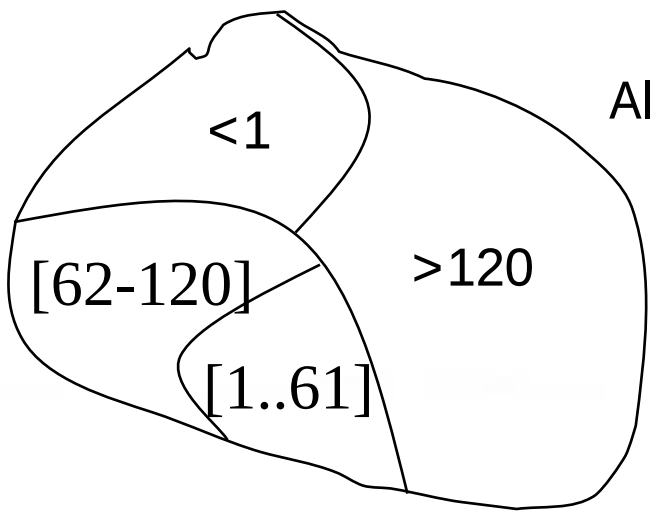
소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 동등 분할(Equivalence partitioning)
    - 입력 값/출력 값 영역(Input/output space)을 유한개의 상호 독립적인 집합(Mutual disjoint subset)으로 나누어 수학적인 등가 집합을 만든 후, 각 등가 집합의 원소 중 대표 값 하나를 선택하여 테스트 케이스를 선정.
    - 동등 분할 클래스는 유효한 입력 데이터와 유효하지 않은 입력 데이터(입력되지 말아야 할 값)를 포함할 수 있다.
    - 테스트 케이스
      - 같은 특성을 가지면서 같은 방식으로 처리된다고 판단하는 모든 등가 집합(Equivalence classes)에서 대표하는 입력 값들을 적어도 한 개씩은 사용하여 작성되었다는 것까지를 보장.
      - 경험과 필요에 따라 하나 이상의 값을 선정하여 테스트 케이스를 작성하는 경우, 하나만 선정하여 테스팅 하는 것보다 더 많은 결함을 발견할 수는 있지만 결과적으로 보장성은 동일.



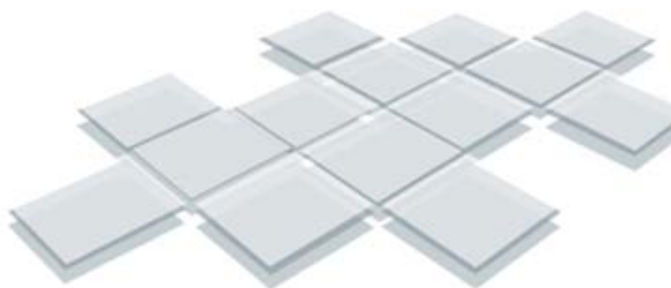
소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 동등 분할(Equivalence partitioning)
    - 동등 분할 적용 범위
      - 출력 값(Outputs)
      - 내부 값(Internal values)
      - 시간관련 값(Time-related values, 이벤트 이전과 이후)
      - 통합 테스팅에서 다루는 모듈간 인터페이스 파라미터(Interface parameters)
    - 모든 테스트 형태(Test types)에서 적용 가능.
    - 기능성 테스팅, 비기능성 테스팅, 구조적 테스팅 뿐 아니라 확인/리그레션 테스팅에서도 사용할 수 있는 "약방의 감초" 같은 테스트 기법.



소프트웨어 테스트 기법

- 명세 기반 테스트 기법: 블랙 박스 테스팅 사례
  - 식료품 점의 전산화 사례에서
    - 식료품의 무게는 1부터 48 사이의 값을 가질 수 있으며 소수점을 갖지 않는 숫자이어야 한다.
  - 동치 클래스
    - 1보다 작은 값(비정상) : 0
    - 1과 48 사이의 값(정상) : 24
    - 48보다 큰 값(비정상) : 100
    - 정수(정상) : 24
    - 실수(비정상) : 7.9
    - 숫자(정상) : 24
    - 숫자 아님(비정상) : 5%

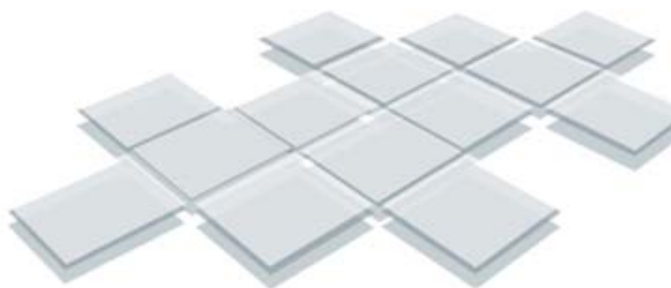


소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 경계 값 분석 (Boundary value analysis)
    - 동등분할의 경계부분에 해당되는 입력 값에서 결함이 발견될 확률이 경험적으로 높기 때문에 결함을 방지하기 위해 경계 값까지 포함하여 테스트하는 기법.
    - 경계 값 : 해당 분할영역의 최대값과 최소값
      - 유효 경계 값 : 유효한 분할영역의 경계 값
      - 비유효 경계 값 : 유효하지 않은 분할영역의 경계 값
    - 경계 값을 고려하여 테스트 케이스를 설계.
    - 경계 값 분석은 동등 분할과 마찬가지로 모든 테스트 레벨, 모든 테스트 형태, 모든 테스트 분류에 적용 가능.
    - 결함 발견 율이 높고, 적용하기 쉬운 장점이 있어 가장 많이 사용되는 테스트 기법 중 하나이다.
    - 경계 값 분석 기법은 경계 값을 명시한 자세한 명세서가 지원될 경우 적용하기가 수월하다.

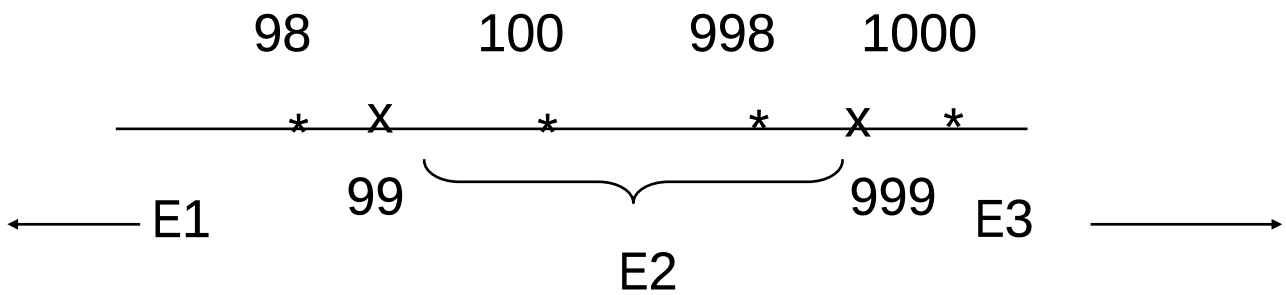
소프트웨어 테스트 기법

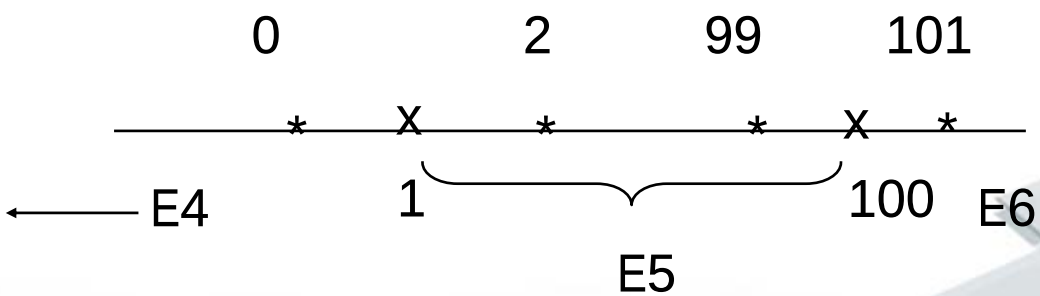
- 명세 기반 테스트 기법
  - 경계 값 분석 (Boundary value analysis)
    - 동등 분할과 경계 값 분석이 사용하기 용이하고 활용도가 높은 기법이기는 하지만 아래와 같은 한계점을 지니고 있다.
      - 일련의 동작에 대한 조합을 테스트하기에는 적합하지 않다.
      - 입력 범위를 동등 분할하여 제한하더라도 입력 값 조합의 수가 테스트 가능한 수를 넘어서는 경우가 많다.
      - 입력 조합이 상호간에 독립적(의존성이 없는)이라는 가정에서만 적합한 기법이다.
      - 출력이 입력조건이나 변수들 사이의 관계에 따라 달라지는 경우, 입력 조건을 동등 분할하는 것이 매우 어려울 수 있다.



소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 경계 값 분석 (Boundary value analysis)
    - 분할된 그룹과 경계 값의 예





소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 결정 테이블 테스팅 (Decision table testing)
    - 논리적인 조건이나 상황(Conditions)을 구현하는 시스템 요구사항을 도출하거나 내부 시스템 디자인을 문서화하는 매우 유용.
    - 시스템이 구현해야 하는 복잡한 비즈니스 규칙(Business rules)을 문서화하는데 사용.
    - 명세를 분석하고, 시스템의 조건과 동작(Actions)을 식별.
    - 입력조건과 동작은 참(True)과 거짓(False)으로 주로 표현.
    - 결정 테이블은 동작을 유발시키는 조건 또는 상황(Triggering conditions - 주로 모든 입력 조건에 대한 참과 거짓의 조합으로 나타남), 각 해당 조합에 대한 예상 결과까지 포함.
    - 테이블의 각 칼럼은 비즈니스 규칙과 대응 관계를 갖는다.
      - 해당 비즈니스 규칙(테이블의 각 칼럼)은 유일한 조건의 조합(Combination of conditions)을 정의하고, 조건의 조합은 해당 비즈니스 규칙과 연관된 동작을 수행.

소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 결정 테이블 테스팅 (Decision table testing)
    - 결정 테이블의 장, 단점 비교

장점	단점
◆논리적으로 의존적인 가능한 모든 조건들의 조합을 생성함 ◆요구사항 등 테스트 베이시스의 문제점을 드러내게 하는 효과적인 테스트 케이스 생성 가능 ◆테스트 베이시스의 불완전성과 모호함 지적 가능 ◆테스트 케이스를 만들면서 결함을 발견하는 것이 가능함	◆작성에 많은 노력과 시간이 소요될 수 있음 (특히, 테스트를 위해 결정 테이블을 만들어야 할 경우 작성 후 개발 측의 검토가 필요함) ◆복잡한 시스템을 표현하기 어려울 수 있으며, 작성 시 논리적 실수의 소지 있음

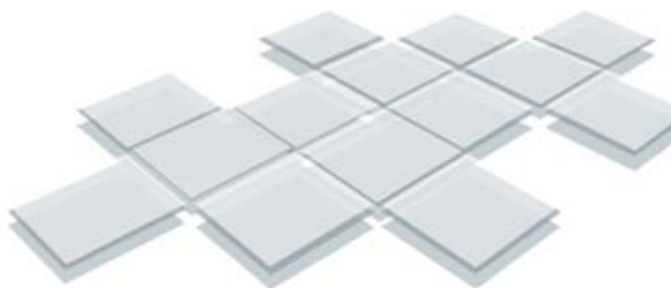
소프트웨어 테스트 기법

- **명세 기반 테스트 기법**
  - 결정 테이블 테스팅 (Decision table testing)
    - 결정 테이블의 예

테스트 케이스 번호		1	2	3	4	5
의사결정	현금 주문	Y	Y	N	N	N
	신용카드	-	-	Y	Y	N
	우수 고객	Y	N	Y	N	-
액션	주문 처리	√	√	√	√	
	주문 거부					√
	10% 할인	√		√		
	정상 가격		√		√	

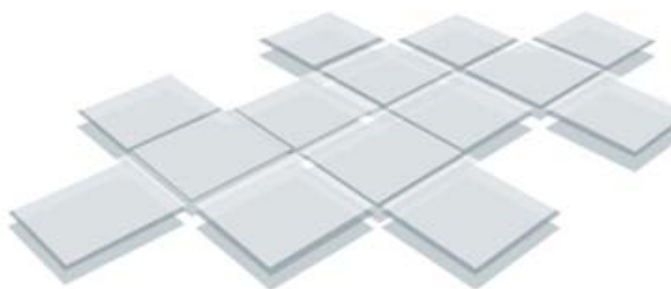
소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 결정 테이블 테스팅 (Decision table testing)
    - 결정 테이블의 예
      - 한 회사에서 다음과 같은 주문 절차 정책을 시행
        - 명령은 현금 지급 또는 신용 인증이 있을 때만 수행된다.
        - 우수 고객은 10% 할인이 가능하고 다른 모든 고객은 전액 모두 지급한다.
      - 5열 : 5개의 테스트 케이스 의미
      - "-" : "Y", "N" 둘 중 아무거나 무방.
      - 3개의 입력 의사 결정이 있으므로 열의 개수는 8(2^3)개 가능.
        - 결과가 같은 것은 제외 가능.



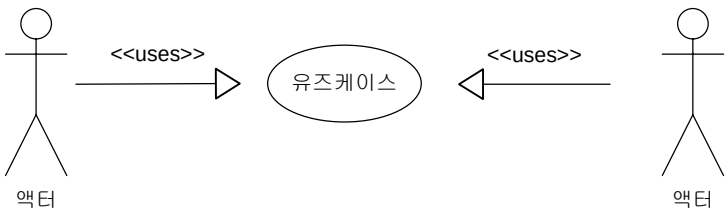
소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 유스케이스 테스팅(Use case testing)
    - 유스케이스나 비즈니스 시나리오를 기반으로 테스트를 명세화
      - 하나의 유스케이스는 액터(유저 혹은 시스템)와 액터 사이의 상호작용을 표현
      - 해당 상호작용은 시스템 유저에게 결과값을 제공.
      - 각각의 유스케이스는 그 유스케이스가 성공적으로 수행되기 위한 전제 조건(Preconditions)을 가짐.
      - 각각의 유스케이스는 임무를 완수한 후 후속조건(Post conditions - 관찰 가능한 결과와 시스템의 마지막 상태)을 가지면서 종료.
    - 대개 주류 시나리오 또는 기본 흐름(Mainstream scenario, Basic or Main flows)과 대체 흐름(Alternative branches or Alternative flows)으로 구성.
    - 각각의 유스케이스는 자세하게 표현하기 위해 유스케이스 상세(Use case description)를 가짐.

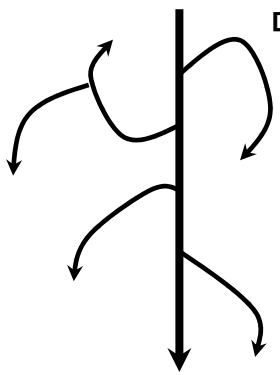


소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 유스케이스 테스팅(Use case testing)



유스케이스와 액터



대체 흐름(Alternative Flows)

기본 흐름(Basic Flows)

유스케이스의 기본 흐름과 대체 흐름

소프트웨어 테스트 기법

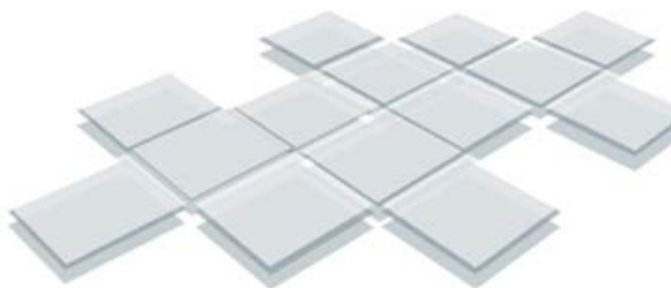
- 명세 기반 테스트 기법
  - 유스케이스 테스팅(Use case testing)
    - 유스케이스는 시스템이 실제 사용되는 방식에 기반하여 "프로세스 흐름"을 기술.
      - 유스케이스에 기반하여 생성된 테스트 케이스는 시스템이 실제 사용되는 프로세스 흐름에서 결함을 발견하는데 상당히 유용.
      - 시나리오라고도 불리는 유스케이스는 고객이나 유저 그룹을 참여시키는 인수테스트(Acceptance Test)를 디자인할 때 매우 유용.
    - 유스케이스 테스팅은 통합테스트 단계에서 서로 다른 컴포넌트 사이의 상호작용과 활동을(시스템 테스팅 레벨에서) 테스팅 하는 방법을 생각할 수 있다.
    - 테스트 케이스 도출 방법
      - 유스케이스 각각의 테스팅은 유스케이스 상세(Use case description)에서 흐름과 시나리오만을 고려하여 테스트 케이스를 도출하는 방법
      - 유스케이스 상세를 문장 별로 분석하여 테스트 케이스를 도출하는 방법

소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 조합 테스팅(pairwise testing)
    - 커버해야 할 기능적 범위에 비해 상대적으로 적은 양의 테스트 세트를 구성하여 소프트웨어의 결함을 찾고 테스트에 대한 자신감(Confidence)을 얻을 수 있는 방법.
    - 대부분의 결함이 2개 요소의 상호작용(Interactions of two factors)에 기인한다는 것에 착안, 2개 요소의 모든 조합을 다룬다.
    - 페어 와이즈 조합
      - 테스트를 하는데 필요한 각 값들이 다른 파라미터의 값과 최소한 한번씩은 조합을 이룬다는 의미.
    - 자원, 시간적으로 제한된 상황에서 테스트 대상 소프트웨어의 설정, 기능, 이벤트 등의 조합을 모두 테스트 하는 것은 일반적으로 불가능.
    - 반면, 테스트를 하지 않거나, 일부 조합을 의도적으로 누락시키는 것은 그만큼의 리스크를 동반하게 되므로 조합 테스팅은 매우 중요한 의미를 가짐.

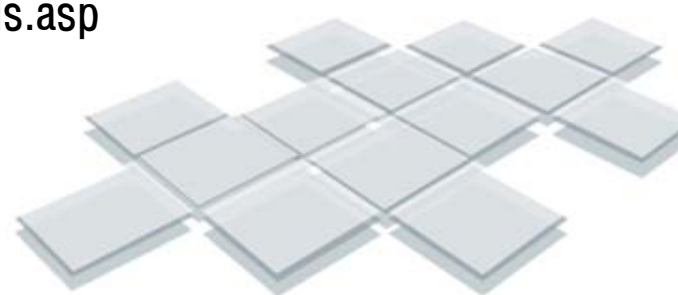
소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 조합 테스팅(pairwise testing)
    - 페어와이즈 조합 테스팅기법을 사용하여 테스팅 한 결과에 결함이 없었다는 것까지는 보장성을 제공.
    - 경험적으로 의미 있고 결함을 발견할 가능성이 높다고 판단되는 조합을 추가하여 관리 가능한 선에서 조합을 늘리는 것은 조합 테스팅의 효율을 높이는데 도움이 된다.
    - 조합 테스팅에서 파라미터(Parameters)는 조합할 값을 대표하는 요소로서 소프트웨어의 다양한 기능, 사용자 또는 하드웨어 설정, 속성, 선택옵션 등의 종류를 파악함으로써 알 수 있다.
    - 값(values)은 각 파라미터에 대한 선택 가능한 개별적인 값을 의미하는 것으로 숫자, 텍스트,또는 리스트에서의 선택된 것이 해당될 수 있다.



소프트웨어 테스트 기법

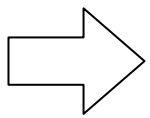
- 명세 기반 테스트 기법
  - 조합 테스팅(pairwise testing)
    - 예제
      - 3개의 파라미터, 각 파라미터가 5가지, 4가지, 5가지의 값 가짐.
      - 총 5 * 4* 5 = 100가지의 조합
      - 100가지 경로 테스트 시 100 * 100 = 10000개 테스트 케이스 수행
    - 조합할 것이 조금 더 많아지고 테스팅 하는 경로나 종류가 더 다양해진다면 테스트에 엄청난 시간과 비용이 소요.
    - 이런 경우, 합리적으로 일정 수준의 보장성을 확보 하면서 조합의 수를 줄일 필요가 있음.
    - 파라미터 수가 조금만 늘어도 페어와이즈 조합 도출이 쉽지 않은데, 자동화 툴을 이용하여 생성할 수 있다.
      - 참고 사이트 : http://www.pairwise.org/tools.asp



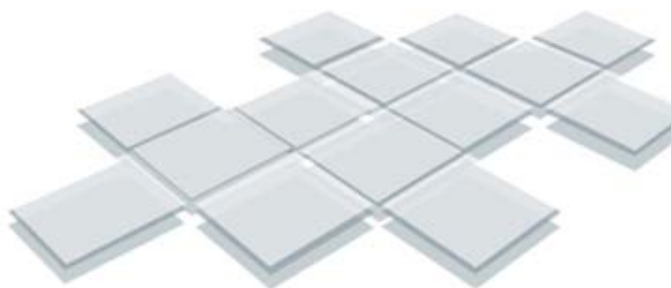
소프트웨어 테스트 기법

- 명세 기반 테스트 기법
  - 조합 테스팅(pairwise testing)
    - 예제
      - 휴대폰 MP3 플레이어 사례

동작모드	설정	이퀄라이저
순차	Hold	Off
순차	Hold	Live
순차	착신	Off
순차	착신	Live
순차반복	Hold	Off
순차반복	Hold	Live
순차반복	착신	Off
순차반복	착신	Live

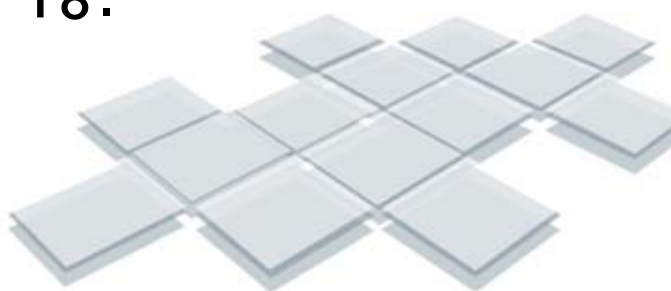


동작모드	설정	이퀄라이저
순차	Hold	Off
순차	착신	Live
순차반복	Hold	Live
순차반복	착신	Off



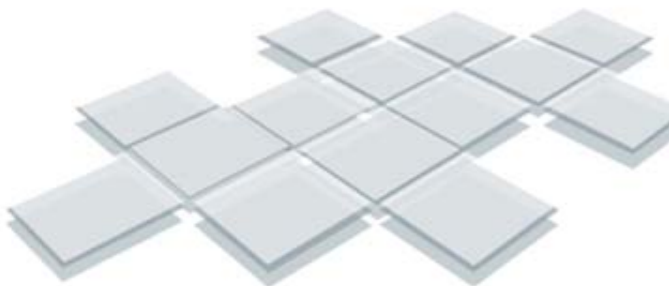
소프트웨어 테스트 기법

- 코드 기반 테스트 기법
  - 코드 기반(화이트 박스) 테스팅은 아래 예시와 같이 소프트웨어나 시스템의 코드와 구조를 중심으로 테스팅 하는 것을 말한다.
    - 컴포넌트 레벨의 구조는 구문, 결정, 분기문 등 코드 그 자체.
    - 통합 레벨의 구조는 한 모듈이 다른 모듈을 호출하는 관계를 도식화한 콜 트리(Call tree) 등이다.
    - 시스템 레벨의 구조는 메뉴 구조, 비즈니스 프로세스 혹은 웹 페이지 구조.
  - 커버리지(Coverage) : 시스템 또는 소프트웨어의 구조가 테스트 스위트(test suite)에 의해 테스트된 정도. 코드의 구조를 테스트하는 기법은 일반적으로 특정 커버리지를 달성하기 위한 테스트를 설계하고 테스트 케이스를 도출하기 위해 사용.



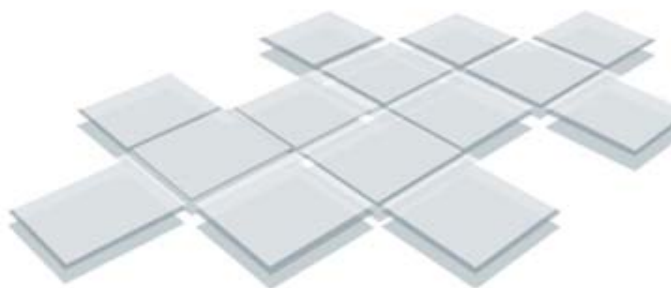
소프트웨어 테스트 기법

- 코드 기반 테스트 기법
  - 코드 기반 테스트 기법의 종류
    - 구문 테스팅과 커버리지 (Statement testing and Coverage)
    - 결정 테스팅과 커버리지 (Decision testing and coverage)
    - 조건 테스팅과 커버리지 (condition testing & coverage)
    - 변경 조건/결정 커버리지(MC/DC)
    - 다중 조건 커버리지 (Multiple Condition Coverage)



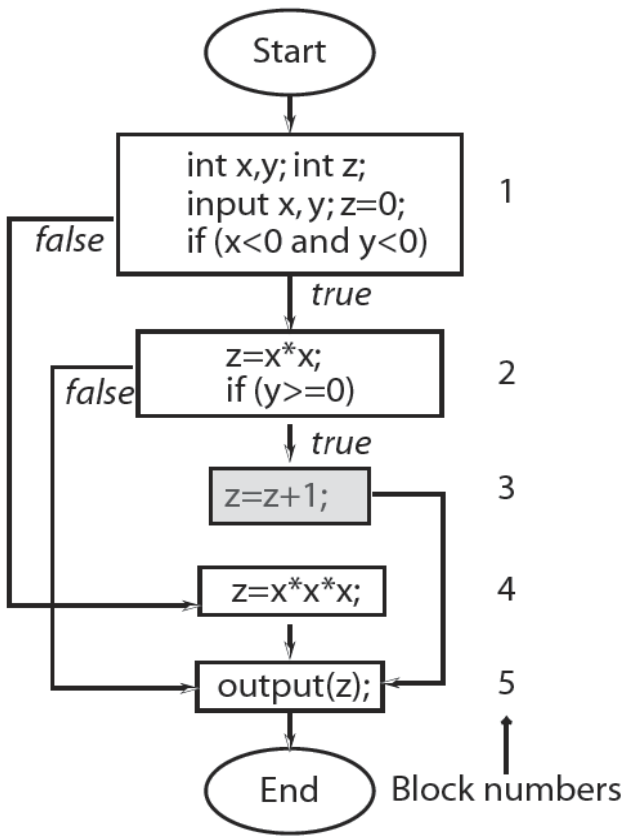
소프트웨어 테스트 기법

- 코드 기반 테스트 기법
  - 구문 테스팅과 커버리지(Statement testing and Coverage)
    - 테스트 케이스 스위트에 의해 실행된 구문이 몇 퍼센트인지를 측정하는 것.
    - 코드의 모든 구문을 실행할 수 있는 입력 값이나 이벤트 등의 테스트 데이터를 제공해 주면 달성.
    - 적은 개수의 테스트 데이터로 쉽게 달성할 수 있지만 코드 상에 존재하는 가능한 경우 중 많은 부분을 검증하지 못하는 보장성 낮은 커버리지.
    - 분기 커버리지, 다중 조건 커버리지, 경로 커버리지 등 포함관계가 더 큰 커버리지를 달성하면 저절로 달성 됨.



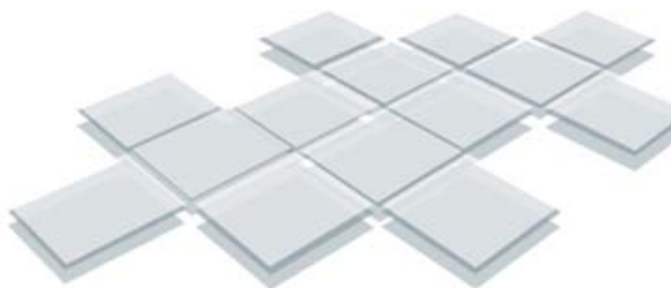
소프트웨어 테스트 기법

- 코드 기반 테스트 기법
  - 구문 테스팅과 커버리지(Statement testing and Coverage)
    - 구문 테스팅 예제



소프트웨어 테스트 기법

- 코드 기반 테스트 기법
  - 결정 테스팅과 커버리지 (Decision testing and Coverage)
    - 테스트 케이스 스위트에 의해 실행된 조건문 분기(if구문의 참 혹은 거짓)가 몇 퍼센트인지를 측정하고 평가하는 것.
    - 결정 포인트 내의 전체 조건식이 “참”과 “거짓”의 모든 값을 갖게 되어 모든 분기로 흐르게 되면 달성 됨.
    - 결정 테스팅은 결정 포인트에 해당하는 제어 흐름을 다루는 제어 흐름 테스팅의 하나이며 제어 흐름도를 이용하여 결정문의 대체 흐름을 가시화.
    - 구문 커버리지 보다 강력, 100% 포함.
    - 제어 흐름 테스트를 통해 테스트 케이스 도출 방법이 많이 쓰임.



소프트웨어 테스트 기법

- 코드 기반 테스트 기법
  - 조건 테스팅과 커버리지 (Condition testing and Coverage)
    - 결정 포인트 내에 있는 개개의 개별조건식이 “참”과 “거짓”의 모든 값을 갖게 되면 달성.
    - 조건 커버리지가 결정 커버리지를 포함한다고 생각되기 쉬우나 개별 조건식이 모두 “참”, “거짓”을 가진다고 해서 전체 조건식이 항상 “참”, “거짓”을 가지는 않는다.

```
If (x >= −2 and y < 4)
Then x= y −7, y = x + y − 5
```

조건
커버리지

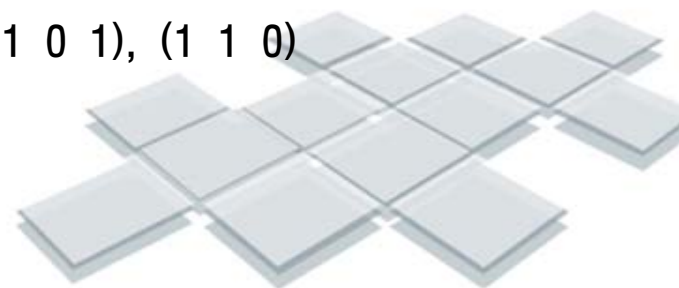
(x, y) / 조건식	(-3, -2)	(0, 6)	(2, 1)
x >= -2	F	T	T
y < 4	T	F	T
x >= -2 and y < 4	F	F	T

조건/결정
커버리지

결정
커버리지

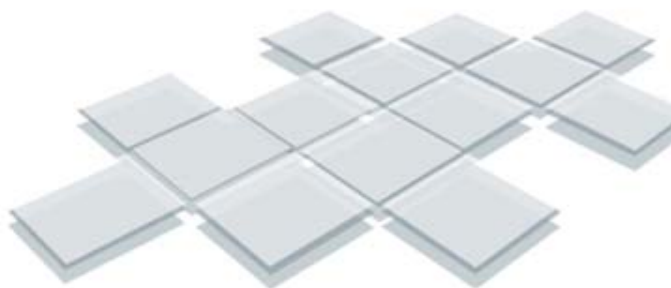
소프트웨어 테스트 기법

- 코드 기반 테스트 기법
  - 변경 조건/결정 커버리지(MC/DC)
    - 각 개별 조건식이 다른 개별 조건식에 영향을 받지 않고 전체 조건식의 결과에 독립적으로 영향을 주도록 함으로써 조건/결정 커버리지를 향상 시킨 것.
    - 결정 커버리지, 조건/결정 커버리지 보다 강력.
    - MC/DC의 결정 테이블 작성 접근법
      - 프로그램에 있는 모든 결정 포인트 내의 전체 조건식은 적어도 한번 모든 가능한 결과값(참, 거짓)을 취한다.
      - 프로그램에 있는 결정 포인트 내의 모든 개별 조건식은 적어도 한번 모든 가능한 결과 값(참, 거짓)을 취한다.
      - 결정 포인트에 있는 각각의 개별 조건식은 다른 개별 조건식에 영향을 받지 않고 그 결정 포인트의 결과값에 독립적으로 영향을 준다.
      - e.g. A OR B: (1 0), (0 1), (0 0)
        A AND B AND C: (1 1 1), (0 1 1), (1 0 1), (1 1 0)



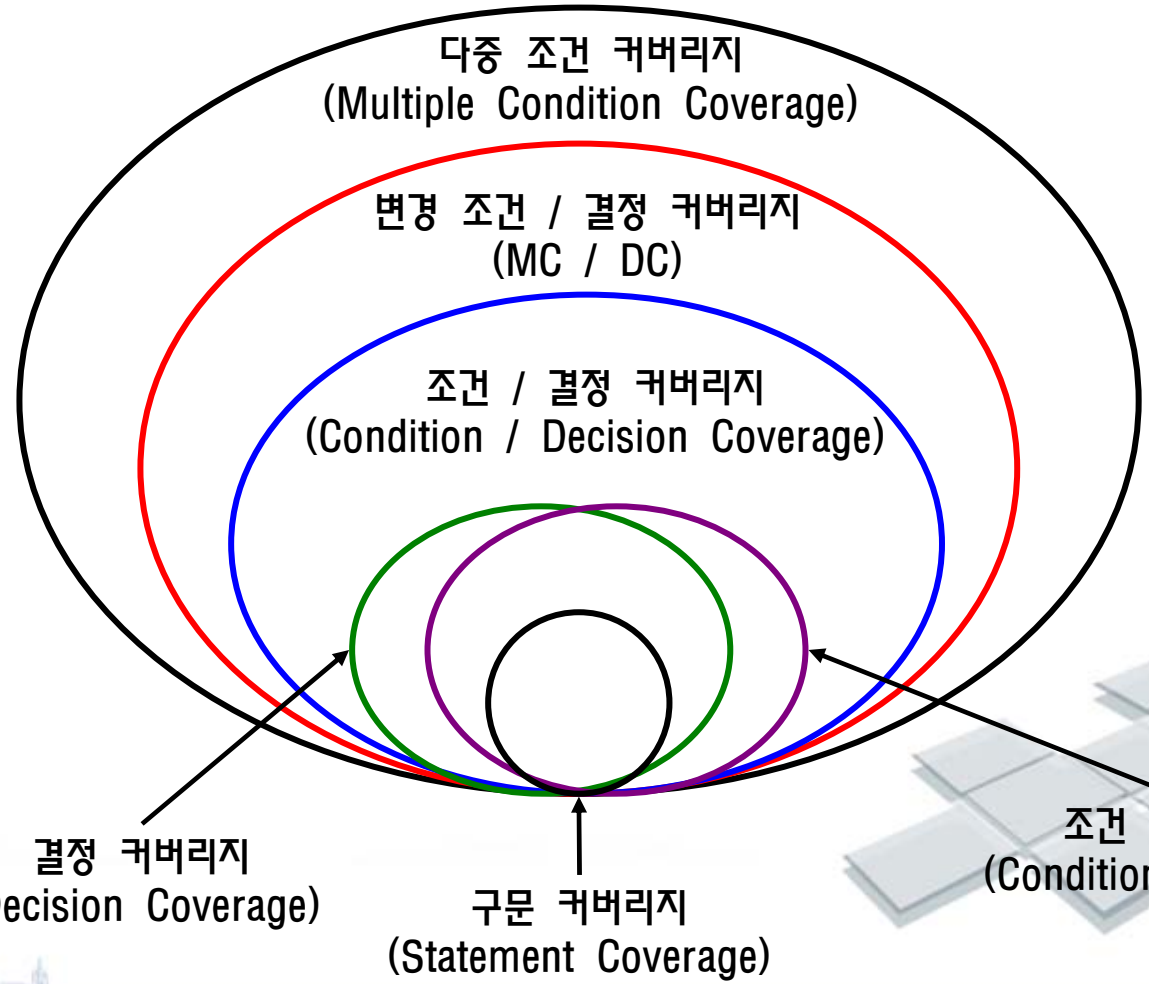
소프트웨어 테스트 기법

- 코드 기반 테스트 기법
  - 다중 조건 커버리지 (Multiple condition coverage)
    - 결정 포인트 내에 있는 모든 개별 조건식의 모든 가능한 논리적인 조합을 고려한 강력한 커버리지를 의미.
    - 모든 조합의 경우의 수를 고려하는 방법이므로 테스트 케이스의 양이 매우 방대, 출시 전에 반드시 100% 결함을 제거해야 하는 제품 테스트에서 주로 사용됨.



소프트웨어 테스트 기법

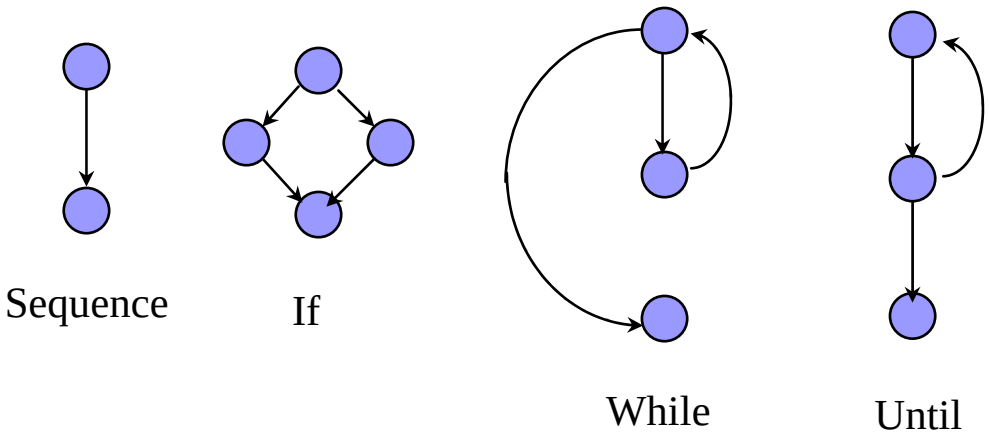
- 코드 기반 테스트 기법
  - 커버리지 간의 포함 관계



소프트웨어 테스트 기법

- 코드 기반 테스트 기법
  - 화이트 박스 테스팅 절차

1. 흐름 그래프 작성



2. 선형적으로 독립적인 경로의 기본 집합 결정
3. 기본 집합에서 각 경로를 실행시킬 수 있는 테스트 케이스 준비

소프트웨어 테스트 기법

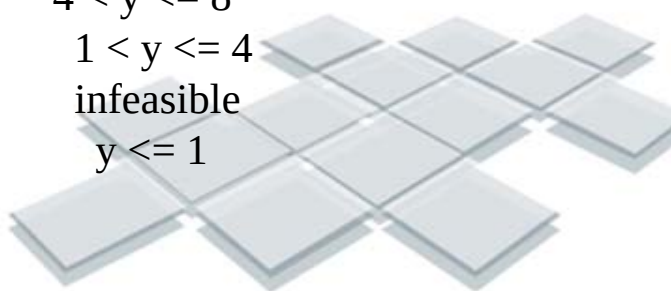
- 코드 기반 테스트 기법
  - 화이트 박스 테스팅 사례

```
if   y > 1   then
       y = y + 1
       if   y > 9   then
            y = y + 1
       else
            y = y + 3
       end
       y = y + 2
 else
       y = y + 4
 end

 if   y > 10  then
       y = y + 1
 else
       y = y - 1
 end
```

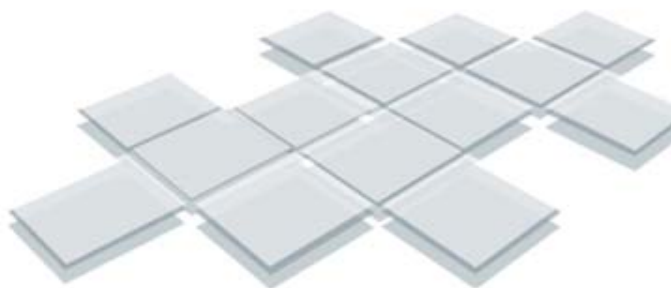
Input Domain	Program Output
y > 8	y + 5
4 < y <= 8	y + 7
1 < y <= 4	y + 5
y <= 1	y + 3

Program Path	Path Condition
T T T	y > 8
T T F	infeasible
T F T	4 < y <= 8
T F F	1 < y <= 4
F - T	infeasible
F - F	y <= 1



소프트웨어 테스트 기법

- 오류기반 테스트 기법(Mutation Test)
  - 과거에 많이 발생한 오류 타입이 현재 테스트 중인 시스템이나 시스템 컴포넌트 또는 프로그램에서도 많이 발생할 것이라는 예측적 지식에 근거하여 주로 이러한 것들의 발견에 중점을 두고 테스트 데이터를 선택하는 방법.
  - 소스 프로그램의 구문을 일정한 규칙을 적용하여 변형시킨 후, 원본 프로그램과 동일한 입력 값을 주고 수행시켰을 때 서로 다른 결과를 출력시키는 적절한 입력 값을 테스트 케이스로 선정.



소프트웨어 테스트 기법

- 오류기반 테스트 기법(Mutation Test)
  - 예제

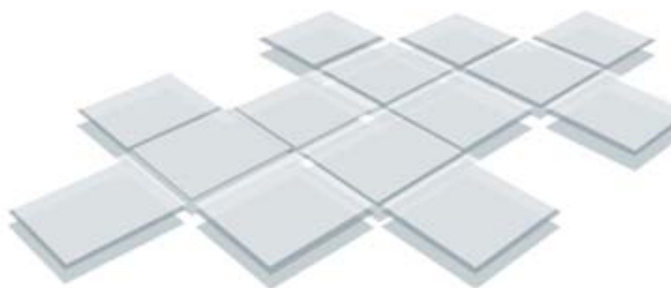
```
int foo(int x, y) {
    return (x-y);
}
```

원본 소스 코드.
이 부분을 변형 시킨다.

M1:
```
int foo(int x, y){
return (x+y);
}
```

M2:
```
int foo(int x, y){
return (x-0);
}
```

M3:
```
int foo(int x, y){
return (0+y);
}
```



소프트웨어 테스트 기법

- 오류기반 테스트 기법(Mutation Test)
  - 예제

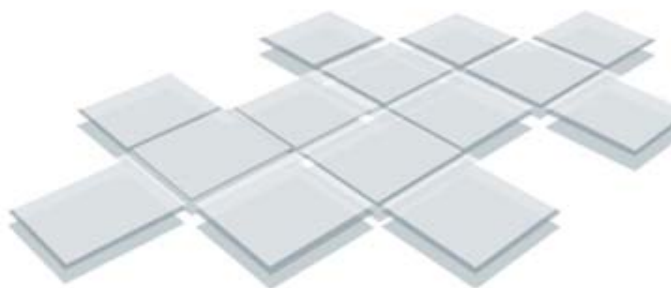
T={ t1: <x=1, y=0>, t2: <x=-1, y=0>}

Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1	1	1	1	0
t2	-1	-1	-1	0
		Live	Live	Killed

T' = T ∪ { t3: <x=1, y=1>} ?

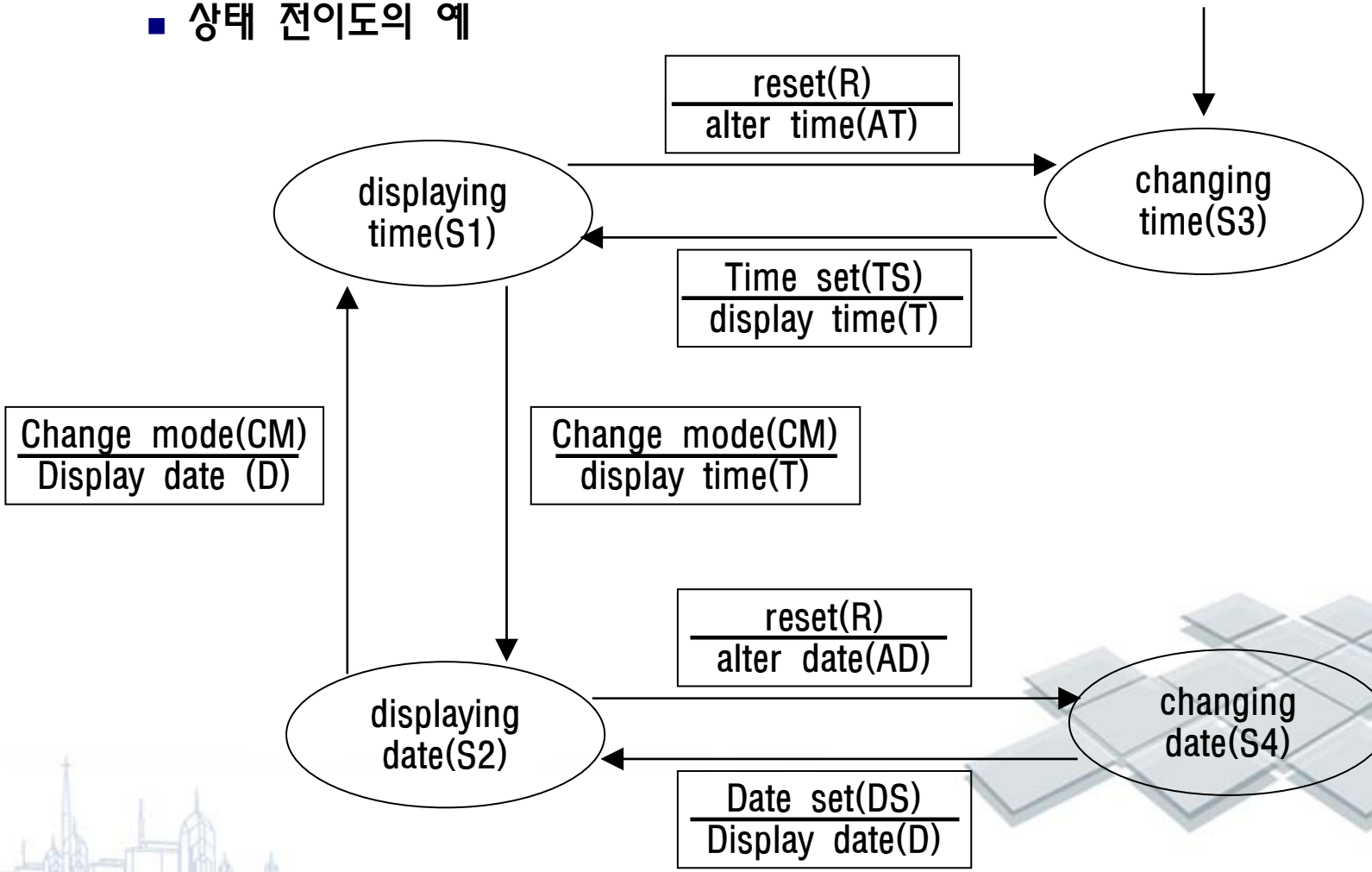
소프트웨어 테스트 기법

- 상태 기반 테스트 기법
  - 상태를 가지는 프로그램을 대상으로 하는 시험 기법으로, 입력 값과 상태를 고려하여 시험 데이터를 생성.
  - 상태 기반 테스트의 목적
    - 시험 데이터는 근원 상태와 입력 값의 쌍으로 구성, 이를 이용하여 근원 상태의 입력 값에 대해서 잘못된 결과값을 출력하는 출력 오류와 잘못된 목적 상태에 도달하는 전이 오류를 확인하는 것.
  - 프로그램 명세나 모델 : 유한 상태 기계(FSM)와 같은 상태 전이 다이어그램으로 주로 표현.
  - 프로그램의 행위 : 프로그램의 근원 상태와 입력 값의 쌍에 대해서 프로그램이 도달하는 목적 상태와 상태 전이시의 출력 값으로 정의.



소프트웨어 테스트 기법

- 상태 기반 테스트 기법
  - 상태전이 테스팅(State transition testing)
    - 상태 전이도의 예

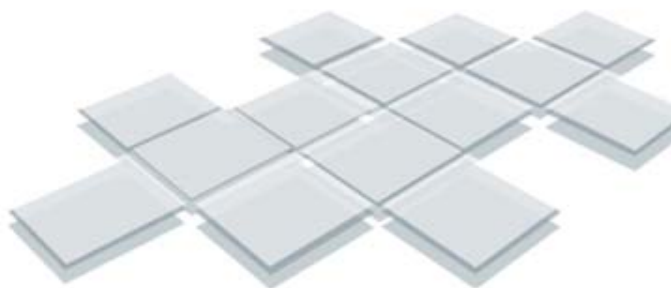


소프트웨어 테스트 기법

- 상태 기반 테스트 기법
  - 상태전이 테스팅(State transition testing)
    - 소프트웨어 또는 시스템을 상태 사이의 관계 즉, 상태 간의 전이, 상태를 변화시키는 이벤트와 입력 값, 상태의 변화로 유발되는 동작 등을 파악.
    - 시스템은 현재 상황(Conditions)이나 이전의 이력(History)을 반영하는 상태(States) 및 그 변화(Transition)에 따라 다르게 동작.
    - 시스템의 이러한 측면을 상태 전이 다이어그램(State transition diagram)으로 표현.
    - 테스트 대상 시스템이나 객체의 상태는 개별적으로 식별 가능하고 유한한 개수로 표현.
    - 상태 전이 다이어그램을 테이블 형태로 전환한 상태 테이블은 상태와 입력 값 간의 관계를 보여주고 개연성은 있지만 요구사항과 맞지 않는 전이(Transition)를 보여줄 수도 있다.

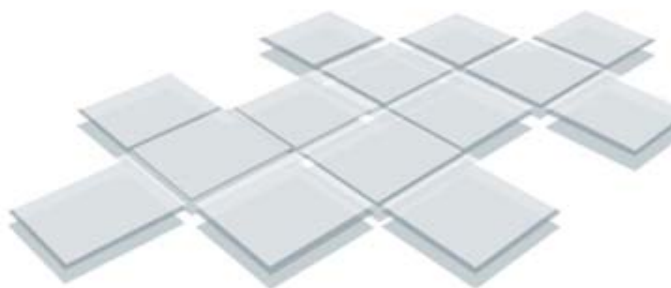
소프트웨어 테스트 기법

- 상태 기반 테스트 기법
  - 상태전이 테스팅(State transition testing)
    - 상태 전이 테스팅을 통해 다음과 같은 방식으로 테스트를 설계하는 것이 가능하다.
      - 전형적인 상태의 순서를 커버하는 방식
      - 모든 상태를 커버하는 방식
      - 모든 상태 전이를 실행하는 방식
      - 특정한 상태 전이 순서를 실행하는 방식
      - 불가능한 상태 전이를 테스트하는 방식



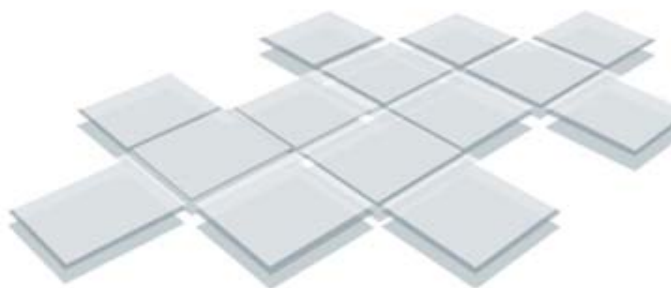
소프트웨어 테스트 기법

- 상태 기반 테스트 기법
  - 상태전이 테스팅(State transition testing)
    - 상태 다이어그램으로 시스템을 설계하는 경우, 존재하는 결함을 아래와 같이 모델상의 결함과 구현상의 결함으로 분류할 수 있다.
      - 모델상의 결함(Defects in model)
        - 초기상태 누락
        - 가드(Guards)를 "전이" 대신 상태에 표기함
        - 가드의 중복 또는 불일치
      - 구현상의 결함(Defects in implementation)
        - 여분/누락/회손 상태(Extra/missing/corrupt state)
        - 액션이 틀리거나 누락됨
        - 스니크 패스(Sneak paths), 트랩 도어(Trap doors, back doors)



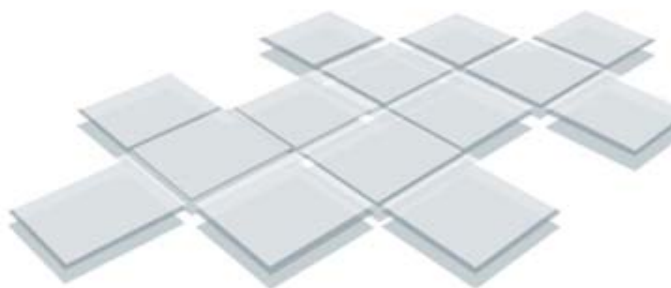
소프트웨어 테스트 기법

- 상태 기반 테스트 기법
  - 상태전이 테스팅(State transition testing)
    - 상태 다이어그램에서 모델상의 결함은 인스펙션 및 리뷰 기법과 정적 분석 툴로 발견하는 것이 가능. 반면 구현상의 결함은 테스팅을 통해 발견할 수 있다.
    - 상태 전이 테스팅은 일반적으로 임베디드 소프트웨어 산업분야나 기술적으로 자동화가 필요한 부분에서 사용된다.
    - 특정한 상태를 갖는 비즈니스 객체 모델링이나 인터넷 어플리케이션 또는 비즈니스 시나리오와 같이 화면-대화 창 흐름을 테스팅 할 때도 적절히 적용할 수 있다.



소프트웨어 테스트 기법

- 객체 지향 기반 테스트 기법
  - 전통적 방법과 객체지향 방법에서 테스트의 차이점
    - 절차적 패러다임 : 프로시저나 함수를 테스팅 단위로 함
    - 객체지향 패러다임 : 클래스 단위의 테스팅
  - 전통적 방법에서의 소프트웨어의 특징
    - 명령식(imperative) 언어로 작성
    - 기능적 분할 방법에 의해 기술
  - 객체지향 프로그램의 특성
    - 상속
    - 캡슐화
    - 다형성(동적 바인딩)

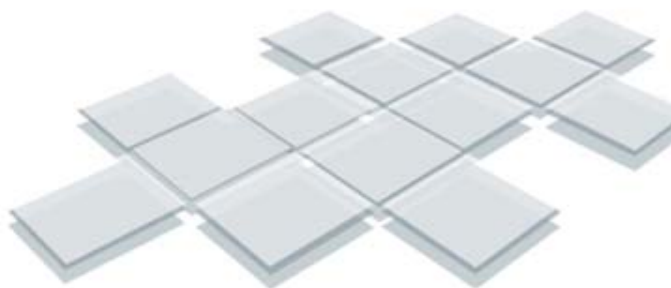


소프트웨어 테스트 기법

- 객체 지향 기반 테스트 기법
  - 테스트를 어렵게 하는 객체지향 프로그램의 특성
    - 캡슐화
      - 클래스 내부 상태 테스트하기 어려움
    - 상속된 멤버함수
      - 슈퍼 클래스에서 이미 테스트 되었지만 파생된 클래스가 사용되는 환경이 다를 경우 다시 테스트 해야 함
    - 다형성
      - 많은 테스트가 필요
    - 동적 바인딩
      - 다형성으로 인한 동적 바인딩은 잘못된 바인딩 메시지 등으로 인해 잘못된 결과를 나타내고 바인딩으로 인한 결과의 예측 어렵게 함
      - 동적 바인딩의 모든 경우를 실행하여 테스트 하는 것은 어렵다.
      - 복잡한 다형성과 그것으로 인한 동적 바인딩은 버그 발생 확률 높임

소프트웨어 테스트 기법

- 객체 지향 기반 테스트 기법
  - 테스트 단계
    - 클래스 테스트
      - 테스트 스터브(stub)와 드라이버를 이용, 클래스를 고립시켜 결함 발견
    - 통합테스트
      - 여러 클래스를 통합하여 결함 발견
    - 시스템 테스트
      - 전체 시스템의 기능 및 비기능적 요구, 목표환경에 초점을 둔 테스트
  - 객체지향 소프트웨어의 단위 테스트
    - 클래스에 대한 테스트



소프트웨어 테스트 기법

- 객체 지향 기반 테스트 기법
  - 클래스 모드에 근거한 테스트 설계 패턴
    - 클래스 모드(class modality)
      - 클래스 행위의 일반적인 패턴
    - 4가지 클래스 모드
      - 모드 없는 클래스(nonmodal class)
        - 메시지 순서가 어떤 제약도 없는 형태
        - 예) DateTime 클래스
          - 날짜나 시간을 읽어 오거나 변경하는 것에 어떤 순서 제약도 없음
      - 단일 모드 클래스(unimodal class)
        - 어떤 상태의 승인을 위한 메시지의 실행 순서에 제약이 있는 형태
        - 예) TrafficSignal 클래스
          - 녹색 신호에서 빨간 신호로 신호 변경
          - GreenLightOff → YellowLightOn → RedLightOn
          - 메시지의 승인을 위한 메시지의 실행 순서에 제약이 있는 형태

소프트웨어 테스트 기법

- 객체 지향 기반 테스트 기법
  - 클래스 모드에 근거한 테스트 설계 패턴
    - 준 모드 클래스(quasi-modal class)
      - 객체의 상태에 따라 받아들여지는 메시지가 제한되는 형태
      - 예) Stack 클래스
        - 보통의 경우, 스택에 값을 넣을 때 별다른 제약 없음
        - 스택이 가득 찼을 경우, 스택에 더 이상 값을 넣을 수 없음
    - 모드 클래스(modal class)
      - 수행 메시지와 입력 도메인 모두에 제약이 있는 형태
      - 예) Account 클래스
        - 만약 은행의 고객이 예금 지급을 요청했을 때 해당 계좌에 남아 있는 금액 보다 많은 금액을 요청하거나 계좌에 돈이 하나도 남아 있지 않다면 예금 지급을 승인하지 않음
        - 계좌가 닫혀있지 않거나 동결되지 않았을 때 해당 계좌의 고객에 의한 계좌 동결 요청을 승인할 수 있음

소프트웨어 테스트 기법

- 객체 지향 기반 테스트 기법
  - 클래스의 통합 테스트
    - 통합 테스트
      - 각각의 클래스 유닛들은 다른 클래스 유닛과 결합하기 전에 완전히 작동되는 상태로 완성되어 있어야 함
      - 통합 테스트
        - 나누어져 있는 각 클래스 유닛들을 결합하여 테스트를 수행
        - 통합 테스트의 주요 목적

          - 클래스 사이의 인터페이스들의 동작이 정확한지를 증명
      - 통합테스트가 포함하는 테스트 목적
        - 정확한 시간에 정확한 메소드가 호출되는가?
        - 올바른 파라미터를 전달하는가?
        - 다루어진 값들이 올바르게 반환되는가?
        - 예외들이 올바르게 핸들링 되는가?

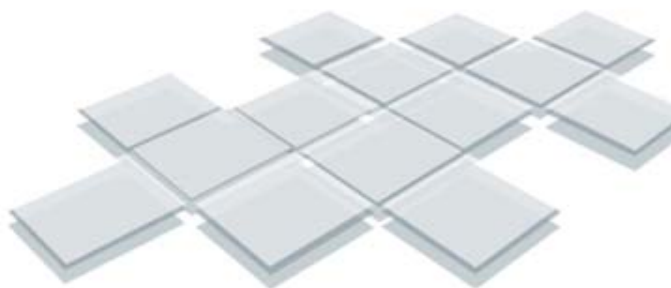
        → 이러한 것들은 유닛 테스트만으로는 발견해 낼 수 없는 문제의 유형으로써 통합 테스트 단계에서 발견하여 오류를 수정 할 수 있음

소프트웨어 테스트 기법

- 객체 지향 기반 테스트 기법
  - 시스템 테스트
    - 전체 시스템이 기능적 요구사항과 비기능적 요구사항을 만족하는지 확인
    - 시스템 테스트의 종류
      - 기능 테스트
        - 기능적 요구사항과 시스템간의 차이를 발견하기 위해 수행
        - 블랙 박스 기법으로 테스트 케이스를 추출
      - 성능 테스트
        - 시스템을 설계하는 동안 선택된 성능 목표와 시스템 사이의 차이를 발견하기 위해 수행
      - 파일럿 테스트
        - 필드 테스트라고도 하며 시스템을 설치한 후 특정 사용자가 시스템을 사용
      - 인수 테스트
        - 개발 의뢰자가 개발 시스템의 환경에서 프로젝트 계약에 명시된 기준에 대하여 사용용이성, 기능성, 성능을 테스트 하는 것
      - 설치 테스트
        - 개발 의뢰자가 목표 시스템의 환경에서 사용용이성, 기능성, 성능 등을 테스트

소프트웨어 테스트 기법

- 컴포넌트 기반 테스트 기법
  - 테스트 가능한 (최소)단위로 분리된 소프트웨어(모듈, 프로그램, 객체, 클래스 등) 내에서 결함을 찾고 그 기능을 검증하는 것.
  - 테스트 스텁(stub), 드라이버, 시뮬레이터가 사용 될 수 있음.
  - 코드 중심 수행.
  - 단위 테스트 프레임워크 또는 디버깅 툴 같은 개발 환경의 지원 필요.
  - 컴포넌트 테스팅의 일반적인 목적
    - 기본적 경로(path)를 확인
    - 모든 오류 처리 경로(Error handling path)를 확인
    - 모듈 인터페이스 확인
    - 로컬 데이터 확인, 경계값 확인

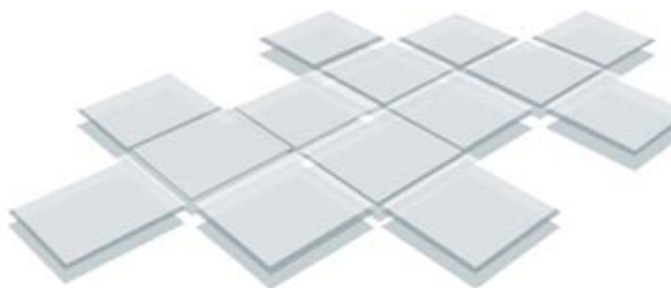


소프트웨어 테스트 충분성 기준

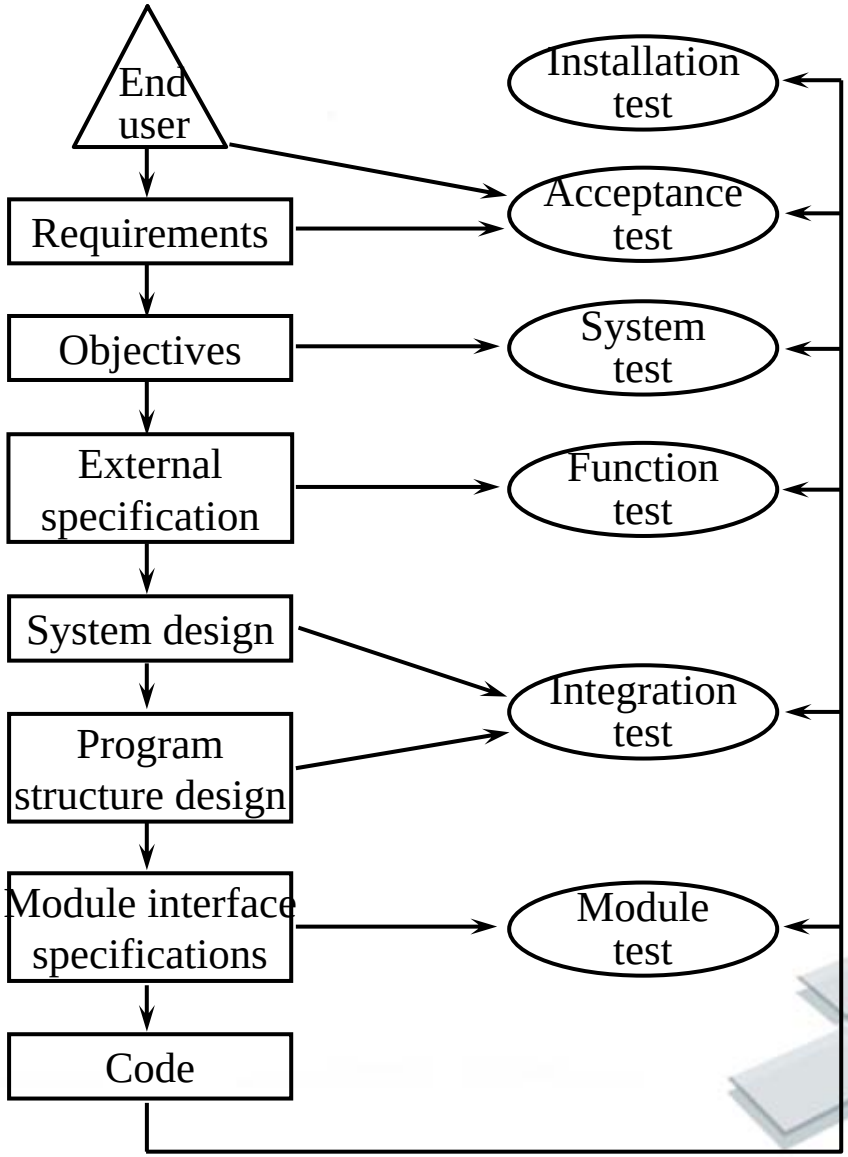
- 제어 흐름 충분성 기준
  - 문장 커버리지
    - 제어 흐름 그래프 상의 모든 basic block을 최소한 한 번씩 수행
  - Branch 커버리지
    - 제어 흐름 그래프 상의 모든 분기를 최소한 한 번씩 수행
  - Path 커버리지
    - 제어 흐름 그래프 상의 모든 경로를 최소한 한 번씩 수행
  - Full Predicate 커버리지
    - 한 결정 내의 모든 조건들이 가질 수 있는 모든 결과 값들을 적어도 한 번씩 검사
  - Modified Condition/Decision 커버리지
    - 모든 조건자 커버리지 만족시키면서 결정과 그 결정을 이루는 조건들 간의 의존성을 검사

소프트웨어 테스트 충분성 기준

- 상태 기반 테스트 충분성 기준
  - 상태 커버리지
    - 모든 상태는 최소한 한 번씩 방문
  - 전이 커버리지
    - 모든 전이는 최소한 한 번씩 방문
  - 경로 커버리지
    - 상태 기계가 갖는 경로들을 최소한 한 번씩 방문



생명주기 테스트

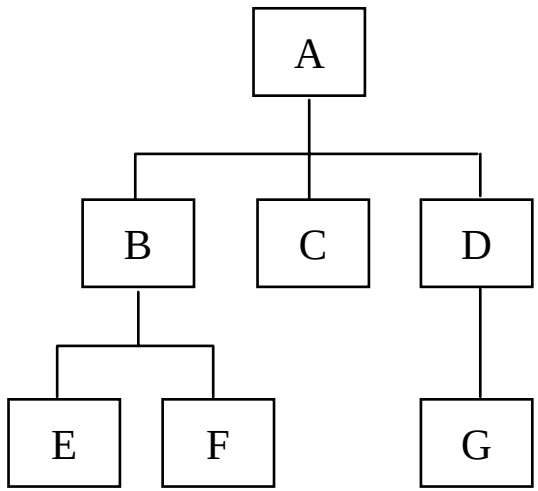

End user
Requirements
Objectives
External specification
System design
Program structure design
Module interface specifications
Code
Installation test
Acceptance test
System test
Function test
Integration test
Module test

생명주기 테스트

- 모듈 테스트(Module Test, 단위 테스트)
  - 격리된 환경(isolated environment)에서 하나의 모듈을 검증
    - 요구 사항과 관련된 특성보다는 설계 및 구현과 관련된 프로그램의 계산 특성을 집중적으로 테스트한다.
    - 테스트 데이터는 프로그램의 기능과 형태에 매우 밀접하게 관련되어야 한다. (white box test)
    - 테스트 스텁(test stubs)과 테스트 드라이버(test driver)가 필요하다.
    - 디버깅(pinpointing and correcting errors)이 쉽다.
  - 고려 사항
    - **인터페이스** : 정보 입출력의 적절성 시험
    - **지역 자료 구조** : 모듈이 실행되는 동안 자료의 무결성 유지 여부를 시험
    - **경계 조건** : 처리를 제한하거나 금지시키기 위해 설정된 경계에서 모듈의 적절한 작동 여부를 시험
    - **독립 경로** : 모듈의 모든 문장은 적어도 한번 실행됨을 확인하기 위한 시험
    - **오류 처리 경로**(error-handling paths)

생명주기 테스트

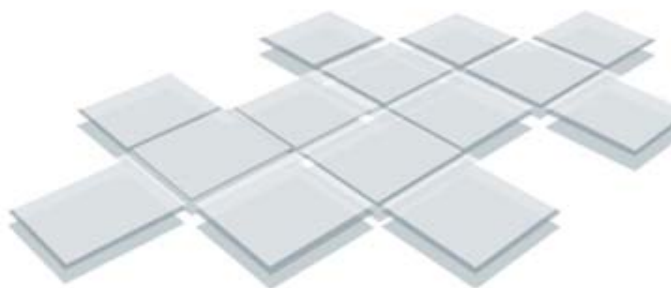
- 통합 테스트(Integration Testing)
  - 시스템을 구성하는 요소들 간의 인터페이스를 검사
    - 비 점진적 테스팅 : "big bang" 테스팅
    - 점진적 테스팅 : 하향식(top-down) 또는 상향식(bottom-up) 테스팅
  - 테스트 종결 기준
    - 모듈 호출
    - 모든 가능한 응답
    - 파라미터 전달



  - Test harness
    - 시스템을 테스트하기 위하여 작성된 별도의 프로그램, 테스팅이 끝난 후 삭제된다.
    - 테스트 드라이버 : 시험 대상 모듈을 호출, 파라미터를 전달, 모듈 수행 후의 결과 제시. 상향식 테스팅에 필요
    - 테스트 스텁 : 시험 대상 모듈이 호출하는 다른 모듈의 기능을 간략히 수행. 메시지 출력, 상수 값 반환

생명주기 테스트

- 통합 테스트(Integration Testing)
  - 하향식 테스팅
    - 개발 초기부터 시행 가능
    - 상위 층의 핵심적인 부분이나 인터페이스에 대해 조기에 검사함으로써 위험을 줄일 수 있다.
    - testing harness를 자연스럽게 제공한다.
    - 테스트 조건을 생성하기가 매우 어렵다.
  - 상향식 테스팅
    - 모듈에서 서브시스템으로, 서브시스템에서 시스템으로 점진적으로 통합해 가면서 테스팅 하기 때문에 에러를 발견하기 쉽다.
    - 하위 층 모듈을 더 많이 테스트하기 때문에 중요한 기능이 하위 층에 많은 경우 효과적이다.
    - 개발 초기에 시스템의 구조를 파악하기 어렵다.



생명주기 테스트

- 기능 테스트(Function Test)
  - 시스템의 외부 명세에 기술된 시스템의 기능을 검사
  - black-box 중심의 테스팅
  - 테스트 데이터 선택
    - 동치 분할(Equivalence Partitioning)
    - 명세 테스팅(specification based testing)

- 시스템 테스트
  - 시스템이 그것의 초기 목표에 맞게 동작하는 가를 확인하고 검증
    - 시스템의 초기 목표(objectives)로부터 외부 명세를 작성하는 동안에 발생한 에러에 대해 집중적으로 검사한다.
  - 테스트 항목
    - Usability
    - Security
    - Performance
    - Reliability
    - Maintainability
    - Compatibility/Conversion