Chapter 19/21. 정리

Chapter 19. 품질 개념

1) 품질 치수

2) 품질 계수

3) 품질 특성

4) 품질 비용

5) 품질 관리와 품질 보증

Chapter 19. 소프트웨어 품질보증

1) SQA요소

2) 소프트웨어 품질 보증요소

3) 신뢰성, 가용성, 안전성 차이

4) 소프트웨어 신뢰성과 가용성 측정

5) ISO 9001:2000 표준요소

5) SQA 계획에 대한 표준요소

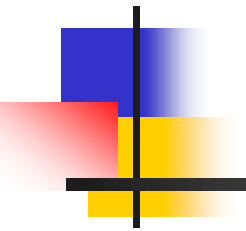
전략

전략이 있는데 전술이 없으면
이기기가 매우 지난하고 ,

전술이 있는데 전략이 없으면
패배를 자초하는 길이다 .

전략은 어느 길로 갈 것인가를
결정하는 것이고 ,
전술은 어떤 방법으로 그 길을
갈 것인가를 결정하는 것이다 .

지혜로운 자는 이로움과 해로움을 동시에 본다



제22장.소프트웨어 테스팅 전략

May. 2018
Young-gon, Kim
ykkim@kpu.ac.kr
Department of Computer Engineering
Korea Polytechnic University

Topics covered

- 소프트웨어 테스팅 전략적 접근
- 전략적 이슈
- 전통적인 소프트웨어를 위한 테스트전략
- 객체지향 소프트웨어를 위한 테스트전략
- 확인 테스팅
- 시스템 테스팅
- 디버깅 기술
- 소프트웨어 결함의 유형

1.소프트웨어 테스팅 전략적 접근

◆ 소프트웨어 테스팅 **전략**

- **로드맵** 제공
  - 테스팅 수행 단계 : 언제 **계획** , **착수** , 많은 **노력**과 시간과 자원 필요 설명
- 테스트 계획, 테스트 케이스 설계, 테스트 실행, 결과 데이터 수집과 평가
- 전략 내용
  - **상위레벨** 테스트 : 고객 요구사항에 대한 주요시스템 기능 확인
  - **하위레벨** 테스트 : 작은 **소스 코드 세그먼트**가 올바르게 구현되었는지를 확인.

* 전략

- " 접근법과 철학 ",
- 실무자를 위한 지침과 관리를 위한 일련의 **이정표 제공**

1.소프트웨어 테스팅 전략적 접근

◆ 테스팅

- 사전에 계획, 체계적으로 수행 일련의 활동
- 소프트웨어 프로세스에 정의 : 테스트케이스 설계 기법, 테스팅 방법<-템플릿

◆ 소프트웨어 테스팅 전략 : 템플릿 제공 -> 일반적인 특성

- 효과적인 테스팅을 수행하려면
  - **효과적인 기술검토** 수행 -> 많은 오류들 테스팅 시작전 에 제거
- 테스팅은 **컴포넌트 수준에서 시작** 하고, **전체 컴퓨터 – 기반 시스템** 을 통합하는 방향으로 “바깥쪽으로” 진행
- 다양한 소프트웨어공학 접근법과 시점에 따라 : 서로 **다른 테스팅 기법** 적합
- 테스팅 : 소프트웨어 개발자와 **독립적인 테스트 그룹**에 수행
- 테스팅과 디버깅 : 서로 다른 활동
  - **디버깅은 모든 테스팅 전략에 수용.**

1.소프트웨어 테스팅 전략적 접근

◆ 검증 및 확인 (V&V : Verification and Validation)

- **검증**
  - 소프트웨어 특정 기능을 올바를게 구현 하였는지를 보장하는 일련의 작업
  - “우리가 제품을 **올바르게** 만들고 있는가”
- **확인**
  - 개발된 소프트웨어가 고객의 요구사항에 맞는지 를 보장하는 일련의 작업
  - “우리가 **올바른** 제품을 만들고 있는가”
- 다양한 SQA **활동**포함
  - 기술적 검토 , 품질과 형상 감사 , 성능 모니터링 , 시뮬레이션 , 타당성 조사 , 문서 검토 , 데이터베이스 검토 , 알고리즘 분석 , 개발 테스팅 , 사용성 테스팅 , 적격성 테스팅 , 인수 테스팅 , 설치 테스팅 .
- 테스팅와 품질
  - 테스팅 : 품질이 평가될 수 있고, 좀더 실용적이며, 오류를 찾아낼 수 있는 최후의 보루
  - 품질 : 소프트웨어공학 프로세스를 거치는 동안 소프트웨어 녹아 들어감 . 테스팅을 하는 동안 방법론과 도구 , 효과적인 기술적 검토 , 그리고 확실한 관리와 측정의 올바른 적용이 입증된 품질로 이어짐.

1.소프트웨어 테스팅 전략적 접근

◆ 테스팅 오해

- 소프트웨어 개발자는 절대 테스팅을 하면 안된다
- 소프트웨어는 그것을 무자비하게 테스트할 낯선 사람에게 “벽 너머로 던져져야 ” 한다
- 테스터들은 테스팅 단계가 시작될 때쯤에서야 프로젝트에 참여한다.

◆ 소프트웨어 테스팅을 위한 조직

- 독립적인 테스트 그룹 (ITG : Independent Test Group) 역할
  - 개발자가 자신의 개발한 것을 **테스트하는 것**과 관련된 본질적인 문제를 제거
  - 이해관계자 충돌 제거
  - 오류를 찾는것에 주의 집중
  - 분석과 설계에 참여
    - 대규모 프로젝트 전반 (테스트 절차 계획및 명세화) 지속적 참여 -> 개발팀 일원
  - 소프트웨어 품질 보증 조직에 리포터 -> 독립성
- 개발자와 ITG 관계
  - 철저한 테스트가 수행될 수 있도록 소프트웨어 프로젝트 전반에 걸쳐 긴밀한 협력관계 를 유지
  - 테스팅 수행중 , 개발자는 발견되는 오류를 수정 하는 것이 가능.

1.소프트웨어 테스팅 전략적 접근

◆ 소프트웨어 테스팅 전략 – 큰 그림

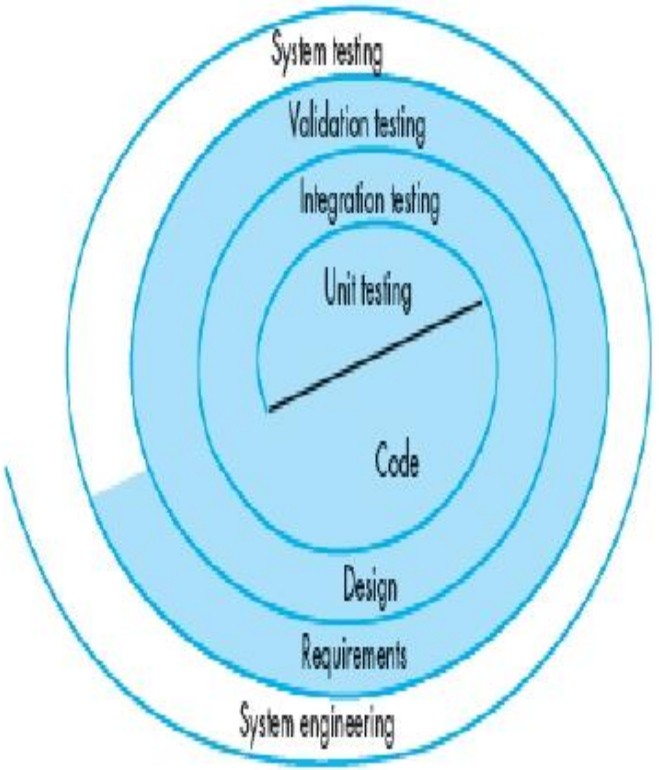
- **나선형** 전략

1) 단위 테스팅 (Unit Testing)
   - 나선 중심에서 시작
   - 소스 코드로 구현되는 단위
     . 컴포넌트 클래스

2) 통합 테스팅 (Integration Testing)
   - 설계와 아키텍처 구축이 초점

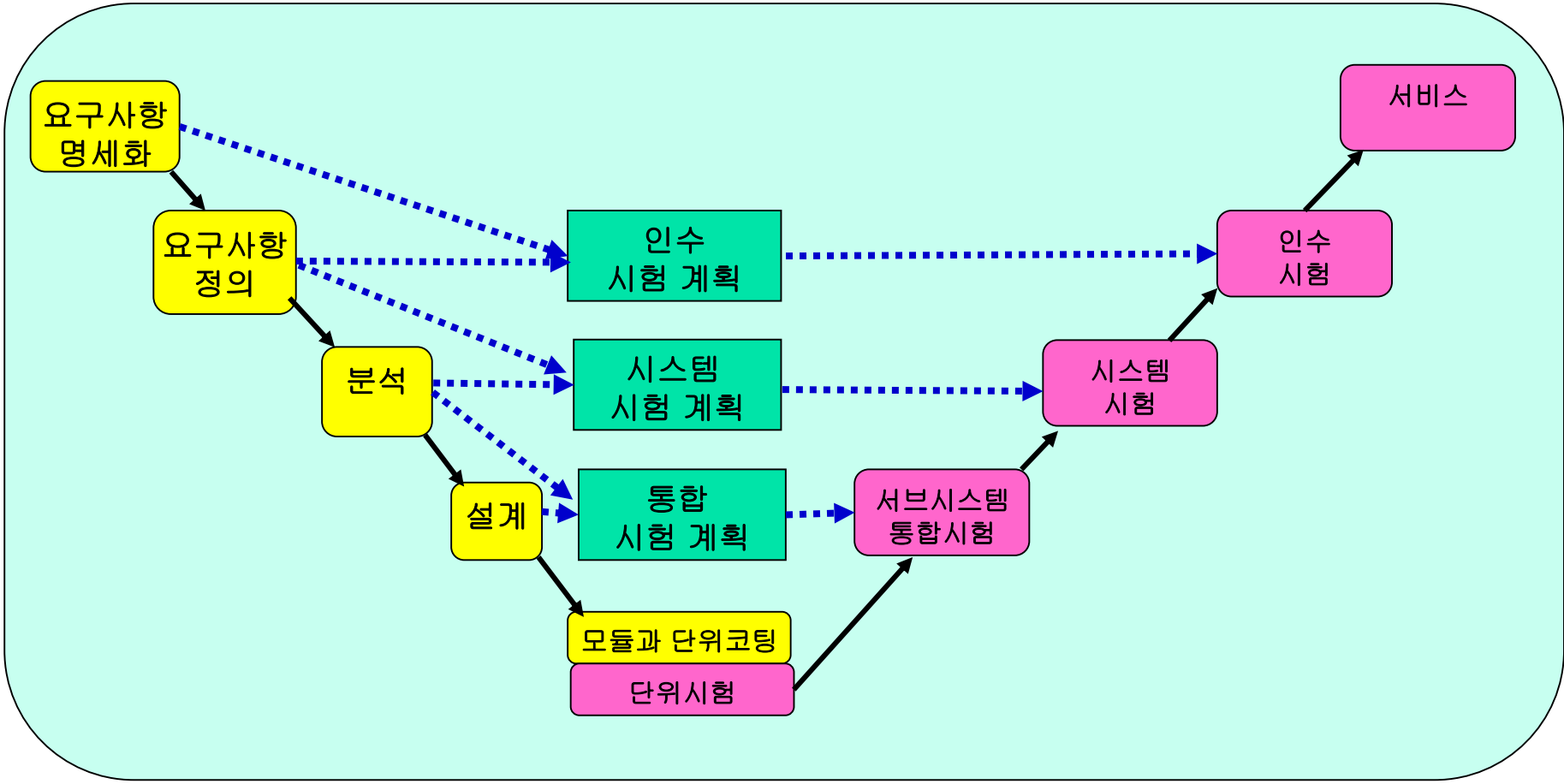
3) 확인 테스팅 (Validation Testing)
   - 요구사항 설계 요구사항 확인

4) 시스템 테스팅 (System Testing)
   - 시스템 요소 테스트.



1.소프트웨어 테스팅 전략적 접근

◆ 개발과 시험을 연결하는 시험 계획



1.소프트웨어 테스팅 전략적 접근

◆ 소프트웨어 테스팅 전략 – 큰 그림

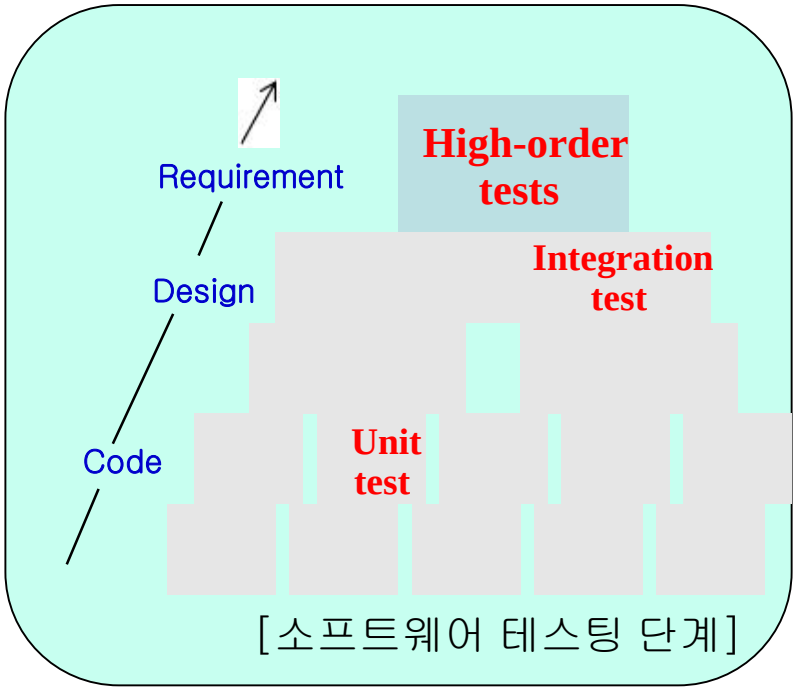
- **절차적 관점에서 프로세스 단계**

1) **단위 테스팅**
   - 완벽한 **커버리지**와 최대한 **오류 검출** 보증
     . 컴포넌트 **제어구조에서의 특정 경로 검사** 테스팅 기법

2) **통합 테스팅**
   - **검증과 프로그램 구축**이라는 이중문제
   - 통합시 **입/출력 초점**을 맞추는 테스트 케이스 설계 기법

3) **확인 테스팅**
   - 소프트웨어가 모든 **기능 , 동작 , 성능 요구사항**을 충족하는 최종 보증 제공

4) **상위레벨 테스팅**
   - **컴퓨터 시스템 공학(SE벗어남)**의 넓은개념
   - 다른 시스템요소와 결합
     . **하드웨어, 사용자, 데이터베이스**
   - 달성 유무 검증
     . 시스템 **요소**가 적절히 **조화**
     . 모든 **시스템 기능/성능** 달성.



[소프트웨어 테스팅 단계]

2.전략적 이슈

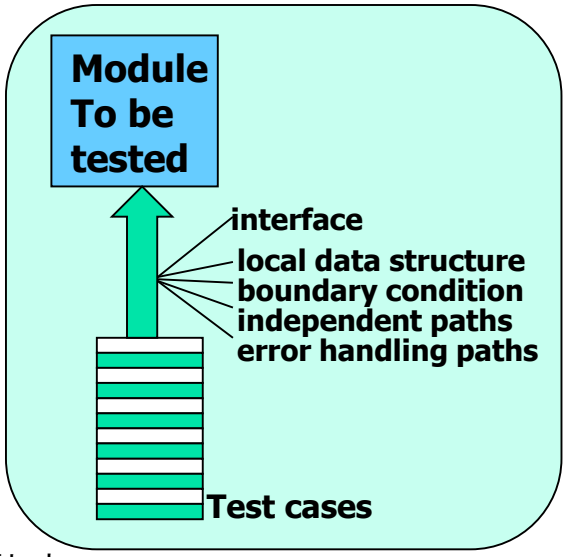
- 소프트웨어 테스팅 전략 성공할 것 주장 : 테스터
  - 테스트 개시 훨씬 이전에 정량적으로 제품 요구사항을 명세화 할 것
  - 테스팅의 목적을 명확하게 서술 할 것
  - 소프트웨어의 사용자를 이해 하고 각 사용자 분류 별 프로파일을 개발 할 것
  - “ 신속한 테스팅 주기 (rapid cycle testing) ”을 강조하는 테스팅 계획을 수립 할것
  - 자기 자신을 테스트하도록 설계된 견고한 소프트웨어를 개발 할 것 (버그방지 : antibugging)
  - 테스팅 이전에 필터링할 수 있도록 효과적인 기술적 검토 를 활용할 것
  - 테스트전략과 테스트케이스 자체를 평가하기 위해 **기술적 검토**를 수행할 것
  - 테스팅 프로세스를 지속적으로 개선하는 방안을 개발할 것.

3.전통적인 소프트웨어를 위한 테스트전략

◆ 1) 단위 테스팅

- 단위테스팅
  - 소프트웨어 **설계의 가장 작은 단위** : 소프트웨어 **컴포넌트** 또는 **모듈** 을 검증 초점
  - **컴포넌트 수준 설계 기술서**를 가이드
  - 모듈의 경계 내부에 있는 오류를 찾아내기 위해 **주요 제어 경로**가 테스트
  - 테스트의 상대적 복잡성과 테스트가 찾아내는 오류 들은 단위테스팅을 하기 위해 설정된 **제한된 범위**로 한정
  - 컴포넌트의 경계 내에서 **내부 처리 로직과 데이터 구조**에 초점
- 단위테스트의 **고려사항**

  1) **모듈인터페이스**
     - 정보가 프로그램 **내/외 부 흐름** 확인
  2) **로컬데이터 구조**
     - **임시 저장 데이터**가 알고리즘 실행동안 무결성 유지
  3) **경계조건**
     - 모듈 한정 , 제한적인 처리위해 **설정된 경계 작동** 확인
  4) **독립 경로**
     - **제어구조**를 통과하는 **모든 경로는 적어도 한번** 실행
  5) **오류 처리 경로**
     - **잘못된 계산 , 맞지않는 비교 , 부적절한 제어 흐름** 오류 확인.

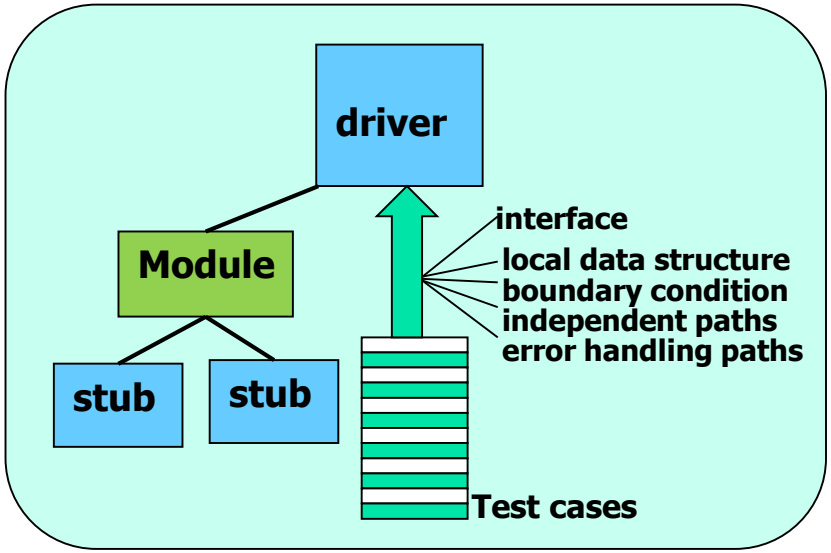


3.전통적인 소프트웨어를 위한 테스트전략

◆ 1) 단위 테스팅

● 단위테스트 **절차**

- 컴포넌트는 단독적인 절차가 아님 -> **드라이버/스텁** 소프트웨어 개발
- **드라이버 (diver)** : " 메인프로그램 "
  - 테스트케이스 입력
  - 컴포넌트로 전달
  - 결과를 출력
- **스텁 (stubs)**
  - 테스트할 컴포넌트에 의해 호출되는 **하위모듈을 대체**하는 역할
  - **더미 서브 프로그램** (스텁)
    . 하위모듈의 **인터페이스** 사용
    . 최소한의 **데이터 조작** 가능
    . 들어온 값을 확인하기위한 **출력**
    . 테스트 중인 모듈로 **제어를 되돌림**



* driver/stub : overhead, 인도물이 아님
  - 단위테스트 미완벽-> 통합테스트까지 사용.

3.전통적인 소프트웨어를 위한 테스트전략

◆2) 통합 테스팅

- 통합 테스팅
  - 소프트웨어 아키텍처를 구축하는 동시에 **인터페이스와 관련된 오류를** 찾아내기 위한 테스트를 수행하는 체계적인 기법
  - 목적 : 단위 테스트가 끝난 컴포넌트들을 가지고 **설계에서 지시** 된 **프로그램 구조**를 만드는 것
  - " 서로 결합 하는" (인터페이스) -> **점증적인 접근방식**
  - 통합 분류
    - **점증적인 통합**
      - 프로그램은 오류를 분리하고 수정하기 쉽도록 작은 단위로 구축되고 테스트
      - 인터페이스: 완전하게 테스트할 가능성 더 높음
      - **체계적인 테스트 접근 방식** 적용
    - 비점증적인 통합 : 빅뱅 방식
      - 모든 컴포넌트들은 사전에 결합되며 **전체 프로그램이 한꺼번에 테스트**
      - **일반적인 혼돈** : 오류발생하면 전체 프로그램이 너무 커서 원인의 분리가 어렵기 때문에 **수정이 어려움.**

3.전통적인 소프트웨어를 위한 테스트전략

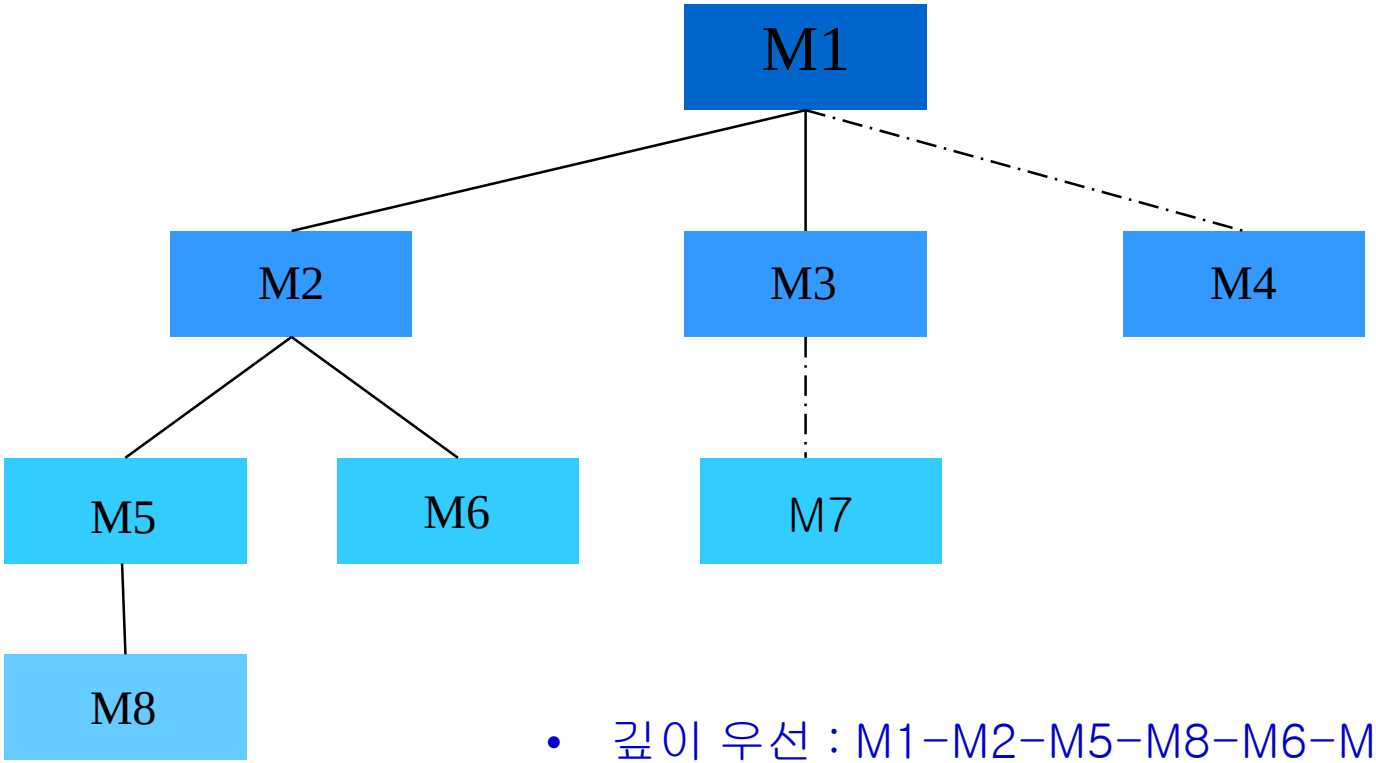
◆ 2) 통합 테스팅: 점증적인 통합 -> 하향식 통합

- 하향식 통합 테스팅 (Top-down integration testing)
- 모듈들은 메인 제어 모듈 (메인 프로그램) 부터 시작 하여 계층구조를 따라 아래 방향 으로 이동하면서 통합
- 메인 제어 모듈에 **종속되는 모듈**들(최하위모듈) : 깊이-우선, 넓이-우선 중 한가지 방식으로 구조에 통합
  - **깊이 – 우선 (depth-first integration) 통합**
    - 프로그램 구조의 **주요 제어 경로**에 있는 모든 컴포넌트를 통합
    - 주요 경로 선택 : 임의적 (주로 : **왼쪽에서 오른쪽**)
  - **넓이 – 우선 (Breadth-first integration) 통합**
    - **구조를 수평적**으로 따라 이동하면서 각 레벨에서 **바로 하위 컴포넌트** 통합
- **통합 과정 단계**

1) **메인 제어 모듈** : 테스트 드라이브 로 사용, 스텁 이 메인제어모듈의 바로 하위에 있는 모든 컴포넌트를 대신
2) **선택된 통합방식 (깊이/넓이 우선) 에 따라** , 하위 스텁이 한번에 하나씩 실제 컴포넌트 대체
3) 각 컴포넌트가 **통합 될 때마다 테스트가 수행**
4) 각 테스트 세트가 완료되면 **다른 스텁이 실제 컴포넌트로 대체**
5) 새로운 오류가 발생되지 않았음을 보증하기 위해 **회귀테스팅이 수행**될 수 있음 .

3.전통적인 소프트웨어를 위한 테스트전략

◆ 2) 통합 테스팅: 점증적인 통합 : 하향식 통합



- 깊이 우선 : M1-M2-M5-M8-M6-M3-M7-M4
- 넓이 우선 : M1-M2-M3-M4-M5-M6-M7-M8

3.전통적인 소프트웨어를 위한 테스트전략

◆ 2) 통합 테스팅: 점증적인 통합 -> 상향식 통합

- 상향식 통합 테스팅 (Bottom-up integration testing)
- **원자 모듈**(프로그램 구조의 최하위 레벨에 있는 컴포넌트) 로부터 구축과 테스팅 시작
- 컴포넌트들이 아래에서 **위로 통합**되기 때문 : 스텁 필요없음(하위모델 사용)
- 통합이 위쪽으로 이동하기 때문
  - 별도의 테스트 드라이브에 대한 필요성 줄어듬
  - **상위 두레벨이 하향식으로 통합** 되었다면
    - **드라이버의 수**는 실질적으로 줄일 수 있고 , **클러스터의 통합**도 간소화
- 통합 과정 단계

  1) 하위 레벨 컴포넌트들이 특정 소프트웨어 **서브 기능을 수행하는 클러스터로 결합**
  (빌더:기능 구현을 위한것[데이터파일,라이브러리,재사용 가능 모듈,컴포넌트)

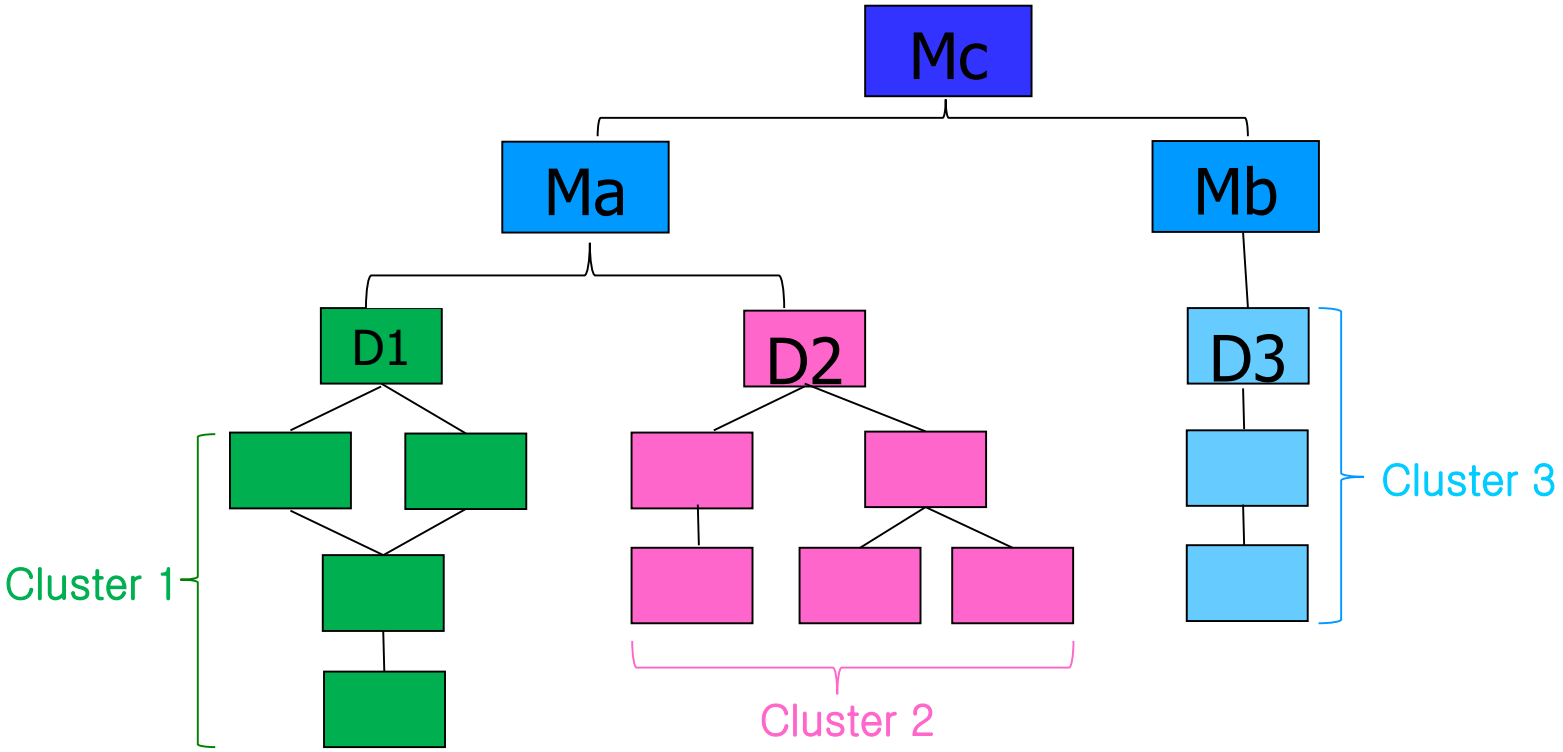
  2) 테스트 케이스의 입력과 출력을 관장하기 위해 **드라이버 작성**
  (테스팅을 위한 제어프로그램)

  3) **클러스터가 테스트**

  4) **드라이버들이 제거**되고 클러스터들은 프로그램 구조의 위쪽으로 이동하며 결합됨.

3.전통적인 소프트웨어를 위한 테스트전략

◆ 2) 통합 테스팅: 점증적인 통합 : 상향식 통합



- 클러스터 1,2,3 형성되도록 결합
- D 1 과 클러스터 1, D 2 와 클러스터 2,D 3 와 클러스터 3
- D 1 과 D 2 제거후 Ma 로 통합 , D 3 제거후 Mb 로 통합
- Ma 와 Mb ->Mc 로 통합

3.전통적인 소프트웨어를 위한 테스트전략

◆ 2) 통합 테스팅: 회귀 테스트

- 회귀테스팅 (regression testing)
  - 변화로 인해 의도하지 않은 **부작용이 전파되지 않았다는 것을 보증**하기 위해 이미 수행된 **테스트의 일부분 다시 – 실행하는 것**
    – 통합테스트 중 **모듈 추가** : 소프트웨어 **변경** (**데이터 흐름 변경 /IO/ 제어로직**)
  - 변경이 **의도하지 않은 동작** 이나 **추가적인 오류** 가 나타나지 않도록 도움
  - 테스트 케이스의 일부를 다시– 실행함으로써 수동/자동화된 캡처 / 플레이백도구
    – Capture/playback: 테스트케이스와 연속적인 플레이백 및 **비교 결과** 제공

- 회귀테스트 **모음** (regression test suite : 실행될 테스트 일부분)의 다른 부류의 테스트케이스 포함
  1) 모든 소프트웨어 기능을 실행할 **테스트의 대표적인 샘플**
  2) **변경에 의해 영향을 받을 가능성이 높은** 소프트웨어 기능에 초점을 맞춘 추가적인 테스트
  3) **변경이 일어난 소프트웨어 컴포넌트** 에 초점을 맞춘 테스트.

3.전통적인 소프트웨어를 위한 테스트전략

◆ 2) 통합 테스팅: 스모킹 테스팅

- 스모크 테스팅 (Smoke testing)
  - 시간이 중요한 프로젝트에서 시간을 조절 하기 위한 메커니즘으로 설계
  - 복잡하고 시간이 중요한 소프트웨어 프로젝트에 적용 : 제품 소프트웨어
- 스모크 테스팅 방법의 활동

  1) 코드로 변환된 소프트웨어 컴포넌트는 빌드로 통합

  - 빌드 : 하나 또는 그 이상의 제품 기능의 구현이 요구되는 모든 데이터 파일 , 라이브러리 , 재사용 가능한 모듈 , 컴포넌트 포함

  2) 일련의 테스트는 빌드의 기능을 제대로 수행하지 못하게하는 에러들을 찾아내기 위해 설계

  - 목적 : 소프트웨어 프로젝트가 계획된 일정을 맞추지 못하도록 가능성이 가장 큰오류 (show-stopper:HW/SW 못쓰게하는 Bug)를 밝혀 내는데 있음

  3) 빌드는 다른 빌드와 통합 되고, 전체 제품(현재 상태에서)은 매일 스모크 테스팅

  - 통합 방식은 하향식 또는 상향식이 될 수 있음.

3.전통적인 소프트웨어를 위한 테스트전략

◆ 2) 통합 테스팅: 스모킹 테스팅

- 스모크 테스트 **이점**
  - **통합 위험의 최소화**
    - 스모크 테스트는 **매일 수행** : 비호환성과 다른 **중대한 오류가 조기에 발견** 되고 , 따라서 오류가 발견되었을 때 **일정상의 심각한 영향을 줄일 수 있음** .
  - **최종 제품의 품질이 향상됨**
    - **통합 지향** : 스모크 테스팅은 **구조적인 오류와 컴포넌트 수준 설계 오류와 기능 오류** 를 찾아낼 가능성이 높음 .
  - **오류 진단과 수정이 간단**해짐
    - 스모크 테스팅 중에 발견되는 오류는 "**소프트웨어의 새로운 증분**"과 관련될 가능성이 크다 .
    - 빌드에 추가되는 소프트웨어는 **새롭게 발견되는 오류의 원인**일 가능성이 크다 .
  - **진행 상황을 평가하기 쉬움**
    - 더 많은 소프트웨어가 통합되고 동작하는 것을 **입증** -> **팀의 사기**를 높여주고 , 관리자에게 **진행 상황 진전되고 있다는 좋은 지표** 제공.

4.객체지향 소프트웨어를 위한 테스트전략

◆ 객체지향에서의 단위 테스팅

- 단위 (Unit) : 캡슐화 (클래스와 객체)
  - 가장 작은 단위 : 클래스 내부의 연산 (메소드) 이 테스트 가능 .
    - 하나의 클래스 : 많은 연산들을 포함하여 하나의 **특정 연산은 많은 클래스들에 속**할 수 있기 때문에 , 단위 테스트에 적용되는 **전술은 변화**.
  - 초점
    - **전통적인 소프트웨어 단위테스팅**
      . 모듈의 상세한 **알고리즘** , 모듈 인터페이스를 통과하는 **데이터 흐름**
    - 객체지향 소프트웨어
      . 클래스 테스팅은 클래스 내에 캡슐화된 **연산과 상태 행위**를 중요시

◆ 객체지향에서의 통합 테스팅의 전략

- 1) 스레드 기반 테스팅 (thread-based testing)
  - 시스템의 **입력과 이벤트에 응답**해야 하는 일련의 클래스 통합,스레드개별적시험
- 2) 사용 기반 테스팅 (use-based testing)
  - 독립적인 클래스 테스트후, 종속적인 클래스 테스팅 -> 전체시스템으로.

7.확인 테스팅

◆ 확인 테스팅

- 통합테스팅의 정점에서 , 소프트웨어가 **완전히 패키지로 조립**되었을 경우
- **인터페이스 오류가 발견되고 수정** 되었을 때 시작
- **초점** : 시스템에서 사용자가 볼 수 있는 행동과 인식 할 수 있는 **출력**
- 확인 테스팅 접근 방식의 근거기반을 형성하는 **확인 기준 부분**을 포함

◆ 확인 테스트 기준

- 설계된 특정 **테스트 케이스 정의**
  - 모든 **기능적인 요구사항** 만족 , 모든 **성능요구사항** 도달 , **문서는 올바른지**
  - **사용성** 과 **기타요구사항** (이식성 , 호환성 , 오류 복구 , 유지보수성)
- 형상검토 (Configuration review)
  - 소프트웨어 **형상의 모든 요소**가 적절히 개발 , 분류 , 지원 활동 강화 하기 위해 필요한 세부사항이 있는 지 확인
- 알파와 베타 테스팅
  - **알파 테스트** : **최종사용자대표 , 개발자 측면** , 사용자의 “어깨너머로 보면서”, 통제된 환경 에서 수행
  - **베타 테스트** : **최종 사용자** , 개발자 미참석 , “생생한”환경 , 고객인수테스팅.

8.시스템 테스팅

◆ 복구 테스팅 (Recovery testing)

- 다양한 방법으로 **장애 발생 후 복구가 적절히 수행** 검증 ,
- **자동 복구** 경우 : 재 초기화 , 체크 포인트형 메커니즘 , 데이터 복구 , 재시작
- **수동**(사람 개입) 복구 : **평균수리시간이 허용 한도** 내 여부 결정 평가

◆ 보안 테스팅 (Security testing)

- 시스템에 **보호 메커너즘 내장**되어 실제적으로 **침투로부터 시스템 보호** 확인
- **전면** 공격 , **측면** 공격 /**후방** 공격에 대하여 손상 테스트

◆ 스트레스 테스팅 (Stress testing)

- **비정상 상황** 에 프로그램 직면, "얼마나 **많이 자극을 주어야 장애 발생** "
- 비정상적인 **수량 , 횟수 , 크기**의 자원 -> 프로그램 파괴 : **민감도 테스팅 기법**

◆ 성능 테스팅 (Performance testing)

- 시스템 요소 완전히 통합 후 , **런타임 성능** 테스트 , 하드웨어와 소프트웨어 모두 필요
- 스트레스 시험과 수시로 결합 : **성능 저화와 시스템 장애** 로 이어질 상황 발견

◆ 배치 테스팅 (Deployment/Configuration testing)

- **동작할 환경** 에서 소프트웨어 검사
- 모든 설치 절차와 고객이 이용하게 되는 **특별한 소프트웨어/소개하는 모든 문서** 검사.

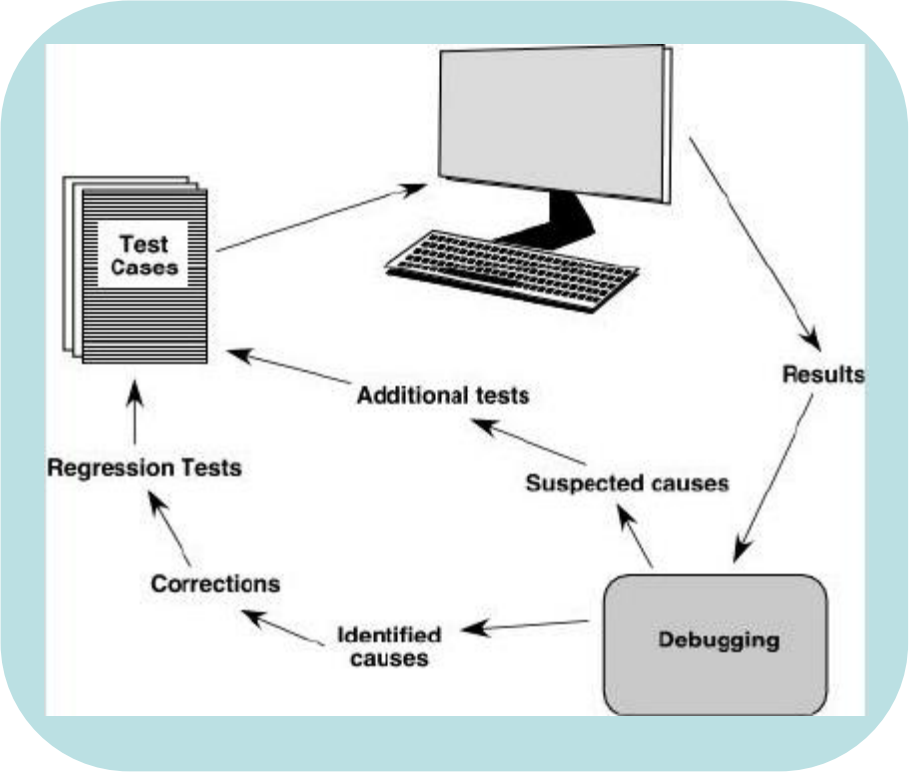
9.디버깅 기술

◆디버깅 (Debugging)

- 성공적인 테스트 결과
- 테스트 케이스 (오류 찾아냄), 디버깅 (**오류 제거하는 과정**)

◆디버깅 프로세스

- 프로세스
  - 시작 : 테스트케이스 실행
  - 결과 판단 : **기대과값과 차이**
  - **원인 증상 미확인**
- 보통 2 가지 결과
  - **원인을 찾아 수정** 하거나
  - **원인을 찾지 못하는** 경우
    - 원인을 의심 하여 :
      의심 확인 을 하기 위해
      테스트 케이스 설계
      -> **반복적인 방법**으로
      오류 수정



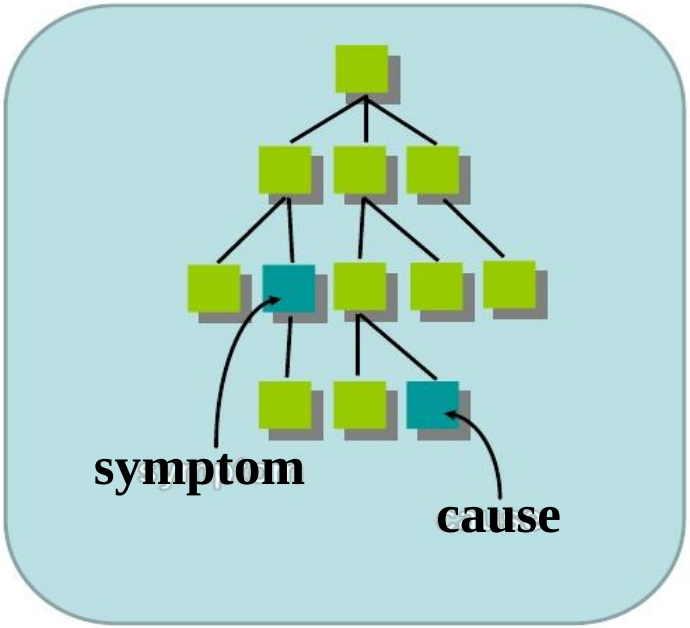
9.디버깅 기술

◆ 버그의 특성

- 증상과 그 원인 : 지리적으로 떨어져 있을 수 있음
  - 결합도 높은 컴포넌트 상황을 더 악화 시킬 수 있음
- 입력조건을 정확하게 재현 하는 것이 어려울 수도 있음(real time)
- 증상 :
  - 다른 오류가 수정되면 (일시적으로) 사라 질 수도 있음.
  - 오류가 아닌 것 때문에 발생 할 수도 있음(반올림 부정확)
  - 쉽게 추적되지 않는 인간의 오류에 의해 발생될 수도 있음
  - 프로세싱 문제보다 타이밍 문제의 결과 로 나타날 수 있음
  - 간헐적 일 수도 있음.
    - 하드웨어와 소프트웨어가 복잡하게 얽힌 임베디드 시스템이 특히 일반적
  - 서로 다른 프로세서에서 동작하는 몇몇 작업들 사이에 분산 되어 있는 원인 에 기인할 수 있음
    .

9.디버깅 기술

◆ 버그의 특성



9.디버깅 기술

◆ 디버깅 전략

- **디버깅 전술**
  - **무작위 (Brute force)**
    - **다른 모든 방법이 실패** 하면 무작위 디버깅 방법 적용 -> **비효율적**인 방법
    - 컴퓨터가 오류를 찾도록 함 : 메모리 덤프 , 런타임 추적 가동 , 출력 로드 <-단지 단서
  - **역추적 (Backtracking)**
    - 증상이 발견된 위치에서 시작하여 원인이 발견될 때까지 **소스코드 거꾸로추적**
    - **작은 프로그램 성공적** 사용
  - **원인 제거 (cause elimination)**
    - 귀납법과 연역법을 이용해 수행 -> 이진 분할 개념 도입
    - 오류 발생 과 관련된 데이터 : **잠재적 원인** 을 분류하여 구성
      . " **원인가설** "이 만들어지고 , **가설을 증명하거나 반증** 하기 위해 데이터 사용
- **자동화된 디버깅**
  - 통합개발환경:컴파일할 필요없이 특정 언어별로 미리 정의된 오류 발견(종료기호,미정의변수
    - 여러 종류의 **디버깅 컴파일러 , 동적 디버깅 도구 , 자동 테스트 케이스 생성기, 상호참조 매핑도구** 사용
- **인적 요인**
  - 좌절의 시간에 이어 밝혀지는 경이로움->"모든 것이 실패로 돌아가면 , **도움을 청하라** ".

10.소프트웨어 결함의 유형

◆ 1) 알고리즘 결함 (algorithmic faults)

- 컴포넌트의 알고리즘 또는 논리 가 처리되는 단계 중에 무엇인가 잘못되었기 때문에 주어진 입력에 대해서 적절한 출력을 보여주지 않을 때 발생함
  - 너무 이른 (늦은) 분기
  - 잘못된 조건에 대한 테스팅
  - 변수 초기화 혹은 루프 불변량 (loop invariants) 설정에 대한 망각
  - 특정한 조건 (0 으로 나누는 연산) 에 대한 테스트 망각
  - 부적절한 자료형을 가진 변수 비교.

◆ 2) 계산 및 정밀도 결함

- 공식 구현의 오류나 요구된 정도의 정확성에 대한 결과를 계산 하지 못할 때 발생함
  - 표현식에서 정수 및 고정 혹은 부동소수점 변수들을 같이 사용 시
  - 부동 소수점 자료의 부적절한 사용 , 예기치 않은 절사 (truncation) 또는 연산들의 순서 가 적합한 정밀도보다 낮은 결과를 가지게 할 수 있음.

10.소프트웨어 결함의 유형

◆3) 문서화 결함

- 문서화가 프로그램이 실제 수행하는 것 과 일치하지 않을 경우
  - 대부분의 우리들은 수정하기 위해 코드를 검토할 때 문서화를 믿는 경향이 있기 때문에 결함.
  - 발생 시 프로그램의 수명 후반부에 다른 결함들의 급증을 초래.

◆4) 스트레스 또는 과부하 결함 (stress or overload faults)

- 자료 구조들이 지정한 용량을 초과 할 때 발생 (큐의 길이 , 버퍼의 크기).

◆5) 용량 또는 경계 결함 (capacity or boundary faults)

- 시스템 활동이 명세된 한계까지 도달하지 못해 시스템의 성능이 받아들여질 수 없을 때 발생함.

◆6) 타이밍 또는 조정 결함 (timing or coordination faults)

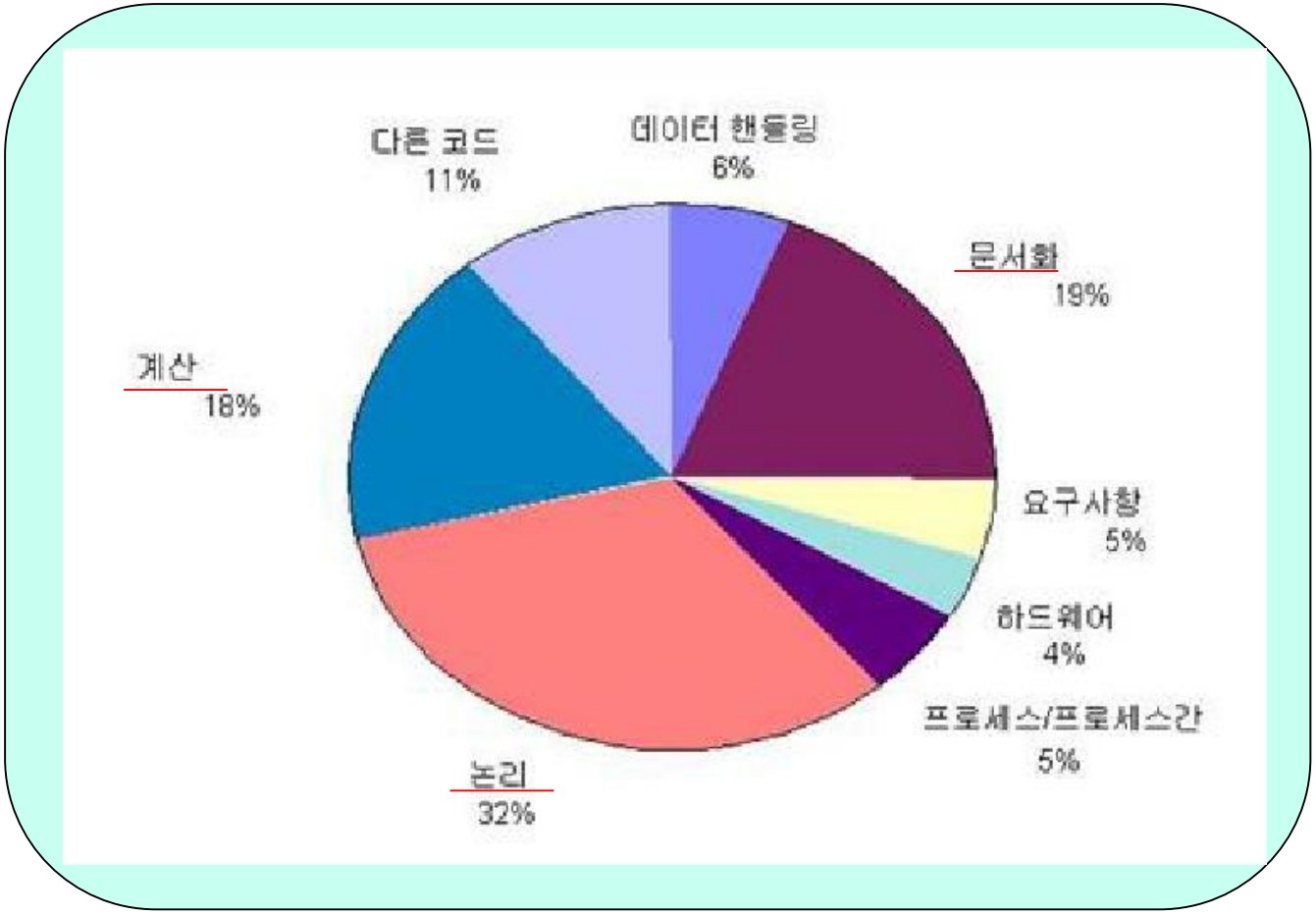
- 이벤트를 조정하는 코드 가 정확하지 않을 때 발생함.

10.소프트웨어 결함의 유형

- **7) 처리능력 또는 성능 결함** (throughput or performance faults)
  - 시스템이 요구사항에 미리 정해진 속도 로 수행되지 않을 때 발생함
- **8) 복구 결함** (recovery faults)
  - 실패가 우연히 발생 했을 때와 시스템이 설계자가 원하고 고객이 요구한대로 작동되지 않을 때 발생함
    - (예) 시스템 처리 중에 전원 고장 발생 - 복구의 문제
- **9) 하드웨어 및 시스템 소프트웨어 결함** (hardware and system software faults)
  - 공급된 하드웨어 및 시스템 소프트웨어가 문서화된 작동 조건과 프로시저 에 따라서 실제로 작동되지 않을 때 발생함
- **10) 표준 및 프로시저 결함** (standards and procedures faults)
  - 시스템이 테스트되고 수정되기 때문에 결함이 생성되는 환경에서는 발생 할 수도 있음.

10.소프트웨어 결함의 유형

◆Hp의 한 부서에서의 결함



정리 및 Homework

1) 소프트웨어 테스팅 전략의 일반적인 특성

2) 검증 및 확인의 정의 및 차이점

3) 테스팅 전략과 나선형 전략, 개발과 시험을 연결하는 시험 계획

4) 테스팅전략의 절차적 관점에서 프로세스 단계

5) 단위테스팅, 통합테스팅, 시스템 테스팅 종류및 차이점

6) 소프트웨어 결함 유형

Project

1장. 프로젝트 개요
- 1.1 프로젝트 제목
- 1.2 선정 이유
- 1.3 팀 운영 방법

2장 시스템 정의
- 2.1 시스템 간략한 설명
- 2.2 유사 사례 간략한 설명

3장 프로세스 모델
- 3.1 규범적인 프로세스 모델 선정 및 이유
- 3.2 특수한 프로세스 모델 선정 및 이유

4장. 실무 가이드 원칙
- 4.1 각 프레임워크 원칙에서 중요한 3 개 정의
- 4.2 프로젝트 계획 보고서

5장. 요구사항 획득
- 5.1 기능 요구사항과 비기능 요구사항 정의
- 5.2 표준 양식을 사용한 시스템 요구사항 명세 3개 작성
- 5.3 정형적인 형식에 따른 유스케이스 작성

6장. 시스템 설계
- 6.1 설계 개념의 중요한 개념을 적용
- 6.2 설계 모델에 따른 요소별 설계

7장. 아키텍처 개념
- 7.1 아키텍처 스타일 선정및 이유
- 7.2 아키텍처 설계 프로세스 정의 및 설계

8장. 품질
- 7.1 시스템 품질 속성 정의
- 7.2 비즈니스 품질 속성 정의
- 7.3 아키텍처 품질 속성 정의
- 7.4 소프트웨어 품질 목표, 속성과 척도
- 7.4 통계적 방법 이용 사례 선정

9장 . 테스팅 전략
- **8.1 테스팅 전략적 이슈 순위정의 (전략 성공)**
  - **– 개인별로 3개, 팀별로 5개 정의**
- **8.2 통합테스팅 방법 및 순서**
  - **– 방법 선정, 프로세스 정의[팀별 1개]**
- **8.3 시스템 테스팅 중요이슈 정의**
  - **– 개인별 2개, 팀별 3개 선정후 정의**
- **8.4 결함 유형 정의**
  - **– 개인별 3개, 팀별 4개 정의.**