협성대학교 소프트웨어공학과

소프트웨어공학

기말고사 대비 문서. (9 주차 ~ 14 주차 강의 요약)

이시헌

2024-6-9

6장 정리 시작.

6장: 아키텍처 설계와 클래스 설계.

1. 아키텍처: 소프트웨어의 골격이 되는 기본 구조.
   A. 시스템 전체에 대한 밑그림 -> 대규모 소프트웨어의 복잡성 문제 해결.
   B. 특징
      ① 추상화된 전체 구조를 제공. (골격)
      ② 소프트웨어의 구성요소들을 다룬다.
      ③ 인터페이스를 통해 구성요소가 어떻게 상호작용하는지 정의.
   C. 효과
      ① 개발에 참여하는 사람들의 이해의 폭 넓어짐.
         -> 구현상의 문제점 도출 가능.
      ② 구조화를 위한 구체적인 방안 도출 가능.
      ③ 설계의 원칙과 가이드 제공.
         -> 설계를 재사용 가능.
      ④ 품질 특성에 대한 평가 방법 결정 가능.

2. 아키텍처의 품질 속성.
   A. 품질 속성: 이해관계자의 요구사항과 관심사를 반영한 속성.
      -> 아키텍처는 이해관계자들이 요구하는 수준의 품질 속성을 달성해야 함.
      -> 이때 요구 수준은 가급적 정확한 수치로 제시하는 것이 좋다.

3. 아키텍처 설계 시 품질 속성을 반영하는 방법.
   A. 품질 속성 반영 과정.
      ① 프로젝트에 중요한 품질 속성 결정.
      ② 품질 속성을 어느 정도로 설계할지 목표 설정.
      ③ 목표 달성 방법 서술.
      ④ 품질 속성을 평가할 방법 서술.
   B. 예시
      ① 보안성을 중요한 품질 속성으로 설정.
      ② 계층 구조 아키텍처 선택.
      ③ 보호해야 할 요소를 가장 깊숙한 내부 계층에 두고, 높은 등급의 보안 인증을 요구함.

4. 아키텍처의 품질 속성 목록.

A. 시스템 품질 속성.

① 가용성 – 실패 없이 운용될 수 있는 확률.
② 변경 용이성 – 얼마나 쉽게 변경 가능한지. 변경을 위한 추적 가능성.
③ 성능 – 초당 처리 가능한 트랜잭션 수.
④ 보안성 – 허용되지 않은 접근 차단, 네트워크 공격 실시간 탐지 등.
⑤ 사용성 – 오류 되돌리기, 도움말, 친숙한 UI….
⑥ 테스트 용이성 – 테스트 비용 감소를 위한 설계 포함.

B. 비즈니스 품질 속성

① 시장 적시성 – 구매한 컴포넌트들의 연결 가능성 + 적절한 출시일.
② 비용과 이익 – 고비용 쉬운 유지보수 VS 저비용 어려운 유지보수.
③ 예상 시스템 수명 – 사용 기간 예측 -> 확장성 이식성 수준 설정.
④ 목표 시장 – 경쟁 제품 고려 + 이식성 수준 설정.
⑤ 신규 발매 일정 – 유연성과 확장성을 기능 공개 일정에 맞춘다.
⑥ 기존 시스템과의 통합 – 기존 시스템과의 통합을 고려하는지 설정.

C. 아키텍처 품질 속성

① 개념적 무결성(일관성) – 전체 시스템과 구성요소가 통합 가능해야 함.
② 정확성과 완전성 – 요구분석명세서와의 일치 정도.
③ 개발 용이성 – 시스템을 모듈로 분할 -> 기한 내에 개발 완료.

5. 아키텍처의 4+1 관점

   A. 아키텍처를 파악하기 위해 제시된 관점 중 4+1 관점을 알아보자.

   B. 해법 영역의 4개 관점 + 문제 영역의 1개 관점으로 구성되어 있다.

   C. 해법 영역

      ① 논리적 관점 – 분석가, 설계자를 위함. 시스템 내부의 클래스나 컴포넌트들의 관계에 초점을 맞춘다. 다양한 다이어그램으로 표현.

      ② 프로세스 관점 – 개발자와 시스템 통합자를 위한 것. 논리적 관점과 유사하나, 시스템의 동시성과 동기화를 중시하여 보는 관점이다.

      ③ 개발 관점 – 프로그래머를 위함. 서브시스템의 모듈(원시 코드, 데이터 파일, 컴포넌트, 실행 파일 등) 구조와 관계들에 초점을 맞춘다.

      ④ 물리적 관점 – 시스템 엔지니어를 위함. 하드웨어를 포함하여, 서브시스템이 물리적 환경에서 어떻게 배치, 연결, 실행되는지를 나타낸다.

   D. 문제 영역 = 시나리오 관점. (유스케이스 관점)

      = 최종 사용자가 원하는 **시스템의 기능**으로 아키텍처를 바라보는 것.

      = 해법 영역의 4개 관점에서 시나리오 관점을 기반으로 시스템을 구성함.

      (설계자와 사용자의 대화에도 사용된다.)

6. 아키텍처 스타일

   A. SW 아키텍처에는 몇 가지 스타일이 있다. 각 스타일만의 구성과 모양이 존재한다.

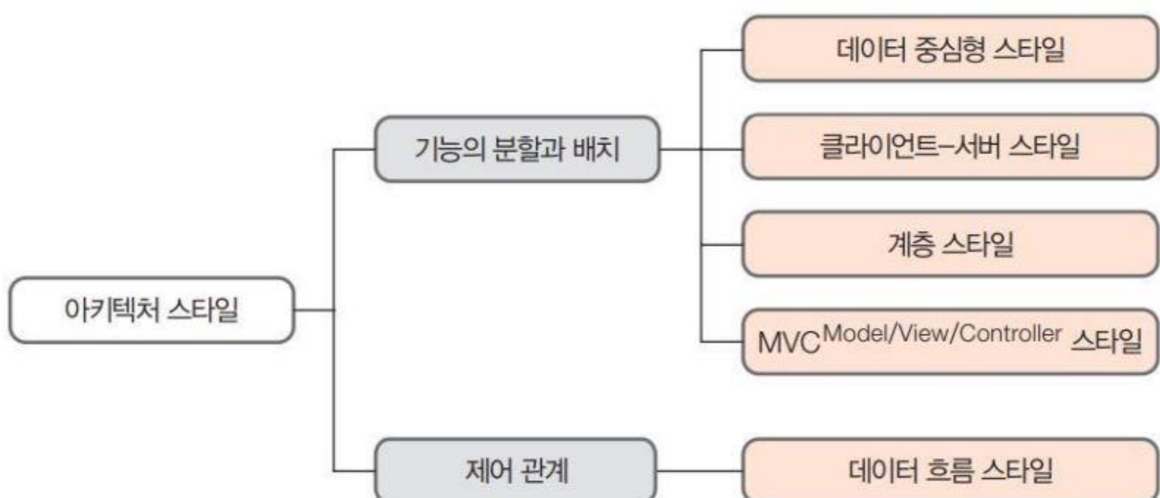


그림 6-3 아키텍처 스타일의 분류

B. 스타일 목록

① 데이터 중심형

- 주요 데이터를 리포지토리에서 중앙 관리하는 시스템.
- 예: 은행 시스템.

② 클라이언트-서버

- 클라이언트와 서버의 기능이 분리되어 있는 시스템.
- 씬 스타일: 클라이언트의 처리 요구량 낮음. (최소: UI만 있음)
- 팻 스타일: 클라이언트에 더 많은 부하. (최대: DB의 일부도 있음)

③ 계층

- 기능을 계층으로 나누어 구현한 것. 일반적으로 운영체제가 계층 구조로 구현되어 있다.
- DB를 많이 사용하는 시스템의 경우 일반적으로 다음과 같이 구성한다.

그림 6-7 3-계층 스타일

④ MVC (중요!)

- SW 개발에 보편적으로 많이 사용되는 스타일이다. 사용자 인터페이스가 자주 변경되어도 모델에는 영향을 주지 않기 때문에, 구조 변경에 대한 요청을 반영하기 쉽다.
- M: Model – 데이터 처리를 실제로 하는 부분.
- V: View – 화면 출력.
- C: Controller – 사용자로부터 입력을 받고, 모델에 연산을 요구하고, 뷰로 결과를 보내는 등, 시스템 전체를 관리한다.

⑤ 데이터 흐름

- 일종의 파이프이다. 하나의 데이터 입력을 처리하여, 다음 시스템에게 결과를 넘겨주는 스타일을 칭한다.
- 예: 유닉스 셸

C. 아키텍처 스타일의 장점

아키텍처 스타일을 사용해 시스템을 설계하면 몇 가지 이점이 있다.

① 개발 기간 단축 <- 시행착오를 줄여준다.

② 수월한 의사소통 <- 공통 아키텍처 공유로 달성.

③ 용이한 유지보수 <- 개발에 참여하지 않은 사람의 이해도 증진.

④ 검증된 아키텍처 -> 안정적인 개발 일정.

⑤ 구축 전 시뮬레이션 -> 시스템 개발 전에 특성을 알아볼 수 있다.

⑥ 기존 시스템에 대한 빠른 이해 가능.

7. 객체지향 설계

A. 객체지향으로 설계를 한다는 것은, 클래스를 정의하고, 클래스들 사이의 관계를 정의하는 것이다. 이제 객체간 관계의 종류를 알아보자.

B. 연관 관계

= 클래스 간에 서로 메시지를 주고받는 관계.

① 양방향 / 단방향

그림 6-10 연관 관계

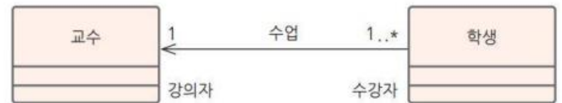


그림 6-15 단방향 연관 관계

② 다중 연관 관계

표 6-2 다중성 표기

표기	의미
1	오직 하나, 즉 1
0..*(또는 *)	다수(0 포함), 즉 0(없거나) 또는 1 이상
1..*	다수(0 제외), 즉 1 이상
0..1	0(없거나) 아니면 1, 즉 0 또는 1
3..6	3에서 6까지 중 하나, 즉 3 또는 4 또는 5 또는 6
2, 4, 6	2 또는 4 또는 6
1, 4..6	1 또는 4에서 6까지 중 하나, 즉 1 또는 4 또는 5 또는 6

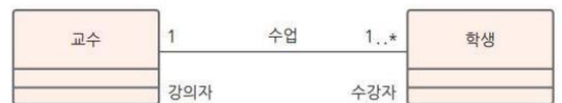


그림 6-12 일 대 다(1 : n) 관계

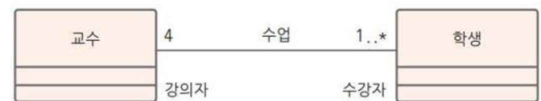


그림 6-13 4 대 다(4 : n) 관계

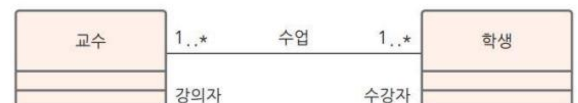


그림 6-14 다 대 다(n:n) 관계

C. 일반화 관계

= 여러 클래스의 공통점을 모아서 상위 클래스로 만드는 작업을 '일반화'라고 한다. (추상화이기도 함)

D. 집합 관계

= 독립적인 클래스들을 모아야만 구성되는 상위 클래스가 있을 때, 이 상위 클래스와 하위 클래스들의 관계를 '집합 관계'라고 한다.

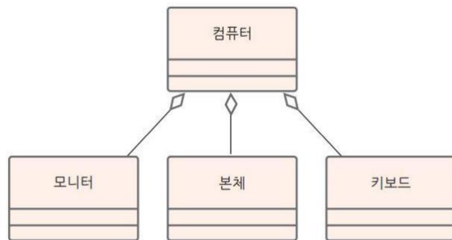


그림 6-18 집합 관계의 예

컴퓨터는 모니터, 본체, 키보드로 이루어진다.

E. 합성 관계

= 상위 클래스에 완전히 종속되는 하위 클래스들이 있을 때, 이 하위 클래스와 상위 클래스의 관계를 '합성 관계'라고 한다. (집합 관계와 유사하지만, 이 경우 하위 클래스들은 독립적으로 존속할 수 없다.)

F. 의존 관계

= 두 클래스가 서로 상대를 사용(참조)할 때의 관계를 말한다.

(자연스럽게 한쪽의 수정은 반대쪽의 수정이 된다. 즉, 의존적이다.)

G. 실체화 관계

= 일반화 관계에 속하는 하위 클래스들의 일부에 새로운 기능을 추가해야 한다고 하자. 이때 추가하는 기능을 상위 클래스로 넣어도 좋지만, 모든 하위 클래스의 공통점이 아니므로, 새로운 상위 클래스를 만들어서 상속시키는 것이 좋다. 이때 인터페이스 클래스를 정의하여 사용하면, 인터페이스와 하위 클래스는 '실체화 관계'를 맺는다.

-참고. 9주차 강의 끝. 10주차 강의 시작.

8. 클래스 설계 원칙

A. 클래스 사이의 관계에 대하여 7번 항목에서 알아보았으니, 이제 클래스를 어떻게 설계해야 하는지 알아보자. (SOLID 원칙)

B. 단일 책임 원칙 S

① 클래스는 단 하나의 책임만 진다.

② 즉, 클래스를 변경해야 하는 이유는 항상 하나이다.

C. 개방 폐쇄 원칙 O

① 이미 작성한 클래스의 변경 없이, 새로운 기능을 추가할 수 있어야 한다. (기능이 추가된 하위 클래스를 새로 작성한다.)

② 즉, 변경에는 닫혀 있고, 확장에는 열려 있어야 한다.

D. 리스코프 교체 원칙 L

① 상위 클래스의 객체는 언제나 하위 클래스의 객체로 교체할 수 있어야 한다. (상위 클래스의 객체가 사용되던 장소에, 하위 클래스의 객체를 넣어도 동작해야 함.)

② 상위 클래스의 메서드를 재정의 한다면, 상위 클래스를 추상 클래스로 작성해야만 달성되는 원칙이다.
(**추상 메서드가 아니라면, 재정의를 하지 않아야 한다.**
즉, 상위 클래스의 기능을 하위 클래스에서 변경하지 않아야 한다는 말이다. 오로지 확장이 전제된 메서드에 대해서만 수정이 허용되어야, 상위 클래스 대신 하위 클래스를 사용해도 문제가 없을 수 있다. 상위 클래스의 기능을 침범하지 마라.)

E. 인터페이스 분리 원칙 I

① 클라이언트는 오로지 자신이 사용하는 메서드와 의존관계를 맺을 수 있다. (사용하지 않는 메서드를 상속받으면 안된다.)

② 각각의 클라이언트들이 필요로 하는 메서드 집단의 개수만큼 인터페이스를 정의해야 달성할 수 있는 원칙이다.

③ 예: 학생 / 교수 / 직원 집단 -> 3개의 인터페이스 클래스 작성.

F. 의존 관계 역전 원칙 D

① 클라이언트는 오로지 추상 클래스와 의존관계를 맺는다.
(구체 클래스와 의존관계를 맺으면 안된다. 자주 바뀌기 때문에!)

② 상위 클래스를 가급적 추상 클래스로 작성하여, 의존관계의 방향을 역전시키라는 뜻이다.

6장 정리 끝.

7장 정리 시작.

7장: 디자인 패턴

1. 디자인 패턴
   A. 여러 설계 사례를 일반화하여 만든, 유형별 설계 패턴.
      (SW 설계에 대한 지식과 노하우를 문제 유형별로 정리한 것.)
   B. GoF: SW공학에서 자주 사용되는 23가지 디자인 패턴을 정의함.
      <- 쉽게 재사용할 수 있도록 객체지향 개념에 따른 설계만을 패턴화.

2. GoF <- 아주 긴 시간동안 중요하게 다뤄졌다. 잘 읽을 것.
   A. GoF는 세 부류로 나뉜다.
      ① 행위 패턴 (11가지)
      ② 구조 패턴 (7가지)
      ③ 생성 패턴 (5가지)

   B. 행위 패턴
      = 객체가 제공하는 기능(행위)의 패턴.
      = 반복적으로 사용되는 객체의 상호작용을 패턴화 한 것.
      = 클래스나 객체가 상호작용하는 방법과 책임을 분산하는 방법을 정의.

      ① strategy
         = **자주 바뀌는 메서드**(기능, 행동)를 뽑아내어, 인터페이스 클래스로 만들자.
         = 유사한 행위들을 캡슐화 하는 인터페이스를 정의하자.

      ② state
         = **자주 바뀌는 상태**를 뽑아내어, (인터페이스)클래스로 변환하자!
         = 상태를 정의하는 인터페이스 클래스에 의해, 그것을 상속받은 클래스의 행동이 변화하도록 설계하자는 말.

-참고. 10주차 강의(strategy 패턴 실습함) 끝. 11주차 강의 시작.

C. 구조 패턴

= 프로그램 내의 데이터나 인터페이스 구조를 설계하는데 활용하는 패턴.

= 클래스나 객체들로 더 큰 구조를 만들어야 할 때 유용한 패턴.

= 다수의 객체를 갖는 큰 규모의 SW에서 새로운 기능을 가진 복합 객체를 손쉽게 추가할 수 있도록 설계하는 방법. (상속 구조에서는 이런 방식으로 객체를 쉽게 추가할 수 없다.)

① decorator

= 객체를 프로그램의 실행 시간에 구성요소를 동적으로 확장하는 패턴을 말한다. (상속의 경우 동적인 확장이 아니다.)

<- 이 같은 '동적 확장'은 '객체의 구성' 혹은 '객체의 합성'을 통해 달성된다.

<- 왜 합성이라 하는가? 동적으로 생성한 객체를, 목표 클래스의 구성요소로 포함시키기 때문이다.

-참고. 11주차 강의(state, decorator 패턴 실습함) 끝. 12주차 강의 시작.

② adapter

= 호환되지 않는 복수의 객체가 서로 상호작용 할 수 있도록 설계하는 패턴을 말한다.

= 기존에 이미 제공되고 있는 클래스를 가져온 경우, 필요한 형태로 클래스를 변환하기 위한 패턴이다.

<- 두 가지 방법: 클래스 adapter 패턴 / 인스턴스 adapter 패턴

- 클래스 adapter: 기준이 되는 객체와 동일한 이름의 메서드를 갖도록 어댑터 클래스를 인터페이스로 정의한 경우.
- 인스턴스 adapter: 인터페이스 작성 후 클래스에 위임한다.
  즉, 모든 하위 클래스의 공통 기능을 뽑아내어 인터페이스 클래스로 만든 후, 해당 인터페이스를 구체화 한 하위 클래스 객체들을 사용하도록 설계한다는 말.
  (위임: 클래스가 다른 클래스 타입의 인스턴스를 가지고 있는 것.)

D. 생성 패턴

= 객체의 생성과 참조 과정을 추상화 하여, 특정 객체의 생성 과정을 패턴화 한 것.

= 언제든 기존 시스템에 영향을 최소화 하며 새로운 객체를 생성 가능하도록 하는 설계 패턴.

① factory method

= 객체 생성만 수행하는 객체를 만들어서, 객체간 의존관계를 관리하도록 설계하는 패턴.

과정 1. factory 클래스 정의 -> 생성.

과정 2. factory 클래스 객체 삭제. (factory 인스턴스가 생성한 객체들은 남아있다.)

= 모든 객체의 동적 생성을 factory 클래스에게 위임하는 것.

= factory클래스가 필요한 모든 객체를 동적으로 생성하고, 생성한 객체들을 여러 다른 객체에게 전달한 후, 자신은 소멸하는 것.

② singleton

= 프로그램 전체에 단 하나만 존재해야 하는 객체를 설계하는 패턴.

= 두 규칙으로 달성할 수 있다.

- 생성자를 private로 선언한다. <- 클래스의 인스턴스 생성 불가.
- 객체를 static으로 선언한다. <- 컴파일 시간동안 전역공간에 객체가 생성된다.

➔ 즉, 다시는 더 생성할 수 없는 객체 하나가, 프로그램 실행 시작 시점에 이미 하나 존재하도록 한다는 것.

- 이렇게 만들어진 singleton 객체는, (static으로 선언된)자기 자신에 대한 참조를 반환하는 게터를 가지고 있어야 한다. (외부로부터의 접근이 차단되어 있기 때문에, 원본을 참조하여 접근해야 한다.)
- 참조로 접근을 한다면, 복수의 객체에서 singleton 객체에 접근할 수 있다. 이때 일관성 유지를 위해 내부 메서드를 동기함수로 작성해야 한다.

  (교재에서는 명시적인 동기함수를 작성한다.

  public synchronized void Brew(){}; <- 이런 방식으로 메소드에 접근 가능한 함수를 하나로 제한한다. (synchronized를 적용하면 한 번에 하나의 스레드만이 이 메소드 실행 가능.)

-참고. 12주차 강의(adapter, factory method, singleton 패턴 실습함) 끝. 13주차 강의 시작.

8장: 구현. 〈- 교재 p.386 ~ 402를 가볍게 읽어보라. 익히 잘 알려진 내용들이라 요약은 생략하겠다.

9장 요약 시작.

9장: 테스트.

1. SW 테스트
   A. 정의
      ① 많은 학자들이 저마다 SW 테스트가 무엇인지 정의하였다. 교재에서 요약한 바를 옮기면 다음과 같다.
      ② SW 테스트
      = SW 내부에 존재하지만 드러나지 않고 숨어 있는 오류를 발견할 목적으로 개발 과정에서 생성되는 문서나 프로그램에 있는 오류를 여러 기술을 이용해 검출하는 작업.
      = SW 내부의 오류 검출 + SW 개발 과정의 오류 검출.

   B. SW 테스트의 목표
      = SW의 신뢰성 향상.
      = 고객의 요구 만족 + 오류 없음

   C. SW 테스트가 어려운 이유
      ① 테스트 케이스가 적다
      ② 완벽한 테스트 케이스 도출이 어렵다
      ③ 테스트를 위한 실제 사용 환경 구축 어려움
      ④ 작은 실수 발견 어려움
      ⑤ 테스트 중요성에 대한 인식 부족
      ⑥ 고객의 요구사항을 충족시켜야 함
      ⑦ 테스트 단계에서만 수행되는 활동이 아님 (개발 단계 전체와 함께함)
      ⑧ 완벽한 테스트 불가능 (오류 없음 보장 불가)
      ⑨ 살충제 패러독스(테스트 내성) 문제 해결을 위한 테스트 케이스 업데이트 요구됨

D. 특징

① 파레토 원리 적용 가능.
(일반적으로 오류의 80%는 SW 모듈의 20% 영역에서 발견된다.)

② 모듈 단위를 점차 확대해 나가며 테스트 진행 가능
(단위 – 통합 – 시스템 – 인수)

E. 용어

① 오류: 개발자의 실수. 결함의 원인.

② 결함: 오류에 의해 SW가 완전치 못한 것. (SW에 필요 없는 정보가 포함되거나, 필요한 정보가 없는 경우가 많다.)

③ 고장, 실패, 문제, 장애: 시스템이 요구사항대로 작동하지 않는 것.

- 요구분석 명세서가 잘못되었거나, 명세서에 요구사항이 충분히 반영되지 못했을 수 있다. 혹은, 기술적으로 불가능한 요구사항이 포함되었을 수도 있다.
- 모든 결함이 실패를 유발하지는 않는다.

2. SW 테스트 절차

= 측정 도구나 장비를 이용해 SW가 사용자의 요구를 만족하는지 테스트하는 절차이다. SW 개발 단계와 연결되어 있는 작업이다.

A. 테스트 절차는 다음과 같다.

① 테스트 계획

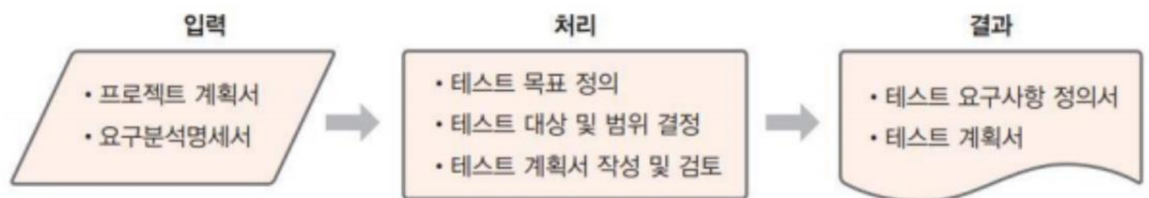


그림 9–3 소프트웨어 테스트 계획 단계

- 그림의 '처리' 부분이 '테스트 계획'단계의 작업이다.
- 테스트 계획서를 도출해야 한다.

② 테스트 케이스 설계

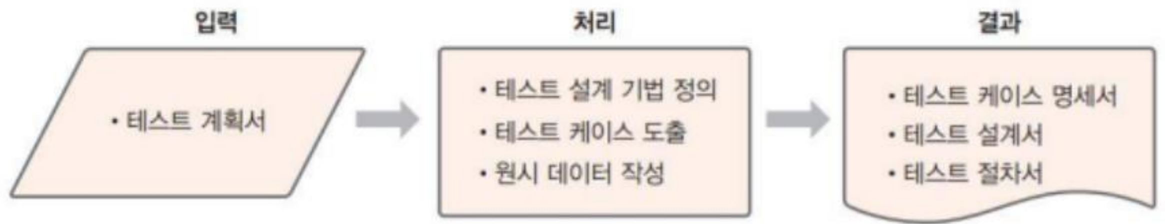


그림 9-4 소프트웨어 테스트 케이스 설계 단계

● 계획서의 내용을 구체화 하는 단계이다.

③ 테스트 실행 및 측정

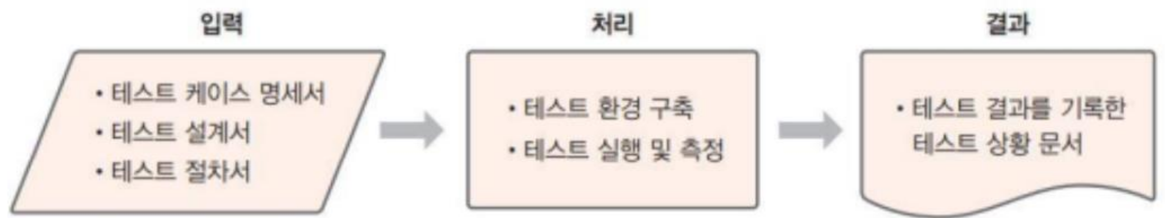


그림 9-5 소프트웨어 테스트 실행 및 측정 단계

● 테스트의 결과물을 산출해야 한다.

④ 테스트 결과 분석

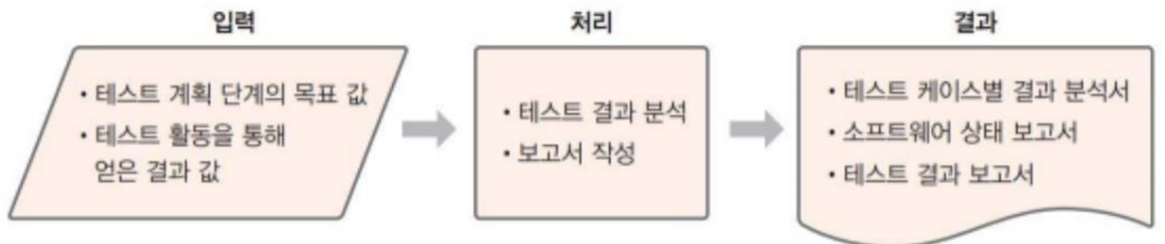


그림 9-6 소프트웨어 테스트 결과 분석 및 보고서 작성 단계

● 테스트 결과를 분석한 보고서를 산출해야 한다.

⑤ 오류 추적 및 수정

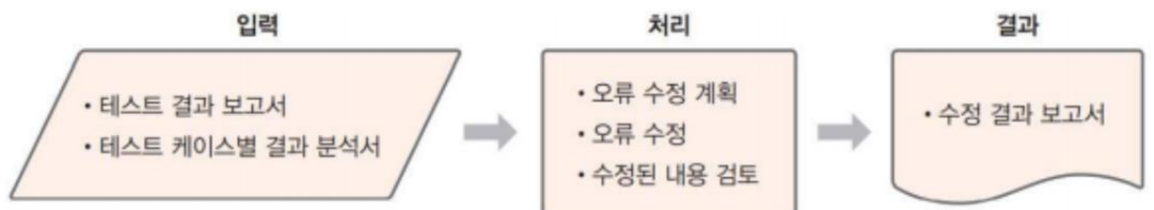


그림 9-7 오류 추적 및 수정 단계

● 분석 보고서를 기반으로, 오류를 수정하고, 또 보고서를 작성한다.

3. 테스트의 분류

A. 시각 기반

① 확인 테스트

- 개발자의 시각에서 설계서대로 제작되었는지 테스트
- 단, 설계서의 문제는 발견 불가능.

② 검증 테스트

- 사용자의 요구사항대로 제작되었는지 테스트.
- 일반적으로 확인 테스트와 함께 진행한다.

B. 목적 기반

① 성능 테스트

- 효율성 측정.
- 부하 -> 실행 시간, 응답 시간, 처리 능력, 자원 사용량 등 평가.

② 스트레스 테스트

- 비정상적으로 높은 부하에서의 시스템 반응 확인.

③ 보안 테스트

- 부당하고 불법적인 침입, 침투를 잘 막아내는지 테스트.

④ 안정성 테스트

- 긴 시간의 부하에서 어떻게 반응하는지 확인. (메모리 누수 등)

⑤ 복원 가능성 테스트

- SW를 고장낸다 -> 복구가 잘 되는지 확인.

C. 실행 여부

① 정적 테스트

- 실행 X
- 코드를(+개발 과정의 모든 산출 문서들을) 보며 오류를 찾는다.

② 동적 테스트

- 실행 O
- SW의 작동 모습을 보면서 오류를 찾는다.

4. 정적 테스트

= 실패보다는 결함을 찾는 방법.

= 전문가들이 모여서 나름의 방법으로 온갖 문서들을 검토하는 방법.

그림 9-10 정적 테스트 방법

A. 가볍게 이해하라. 결국 사람이 읽고 검토하는 것이다.

-참고. 13주차 강의 끝. 14주차 강의 시작.

5. 동적 테스트

A. 두 종류가 있다. 명세 기반 / 구현 기반

6. 명세 기반 테스트

= 블랙박스 테스트.

= 요구분석명세서 혹은 설계사양서에서 테스트 케이스를 추출하여 테스트 함.

= SW 내부를 직접 들여다보지 않는다. 즉, 현실의 사물로 치자면, 검사 장비를 동원한 비파괴검사가 블랙박스 테스트에 속한다.

<- SW 가 기능을 어떻게 수행하는지는 관심이 없다. 내부를 열어 볼 이유가 없는 것이다. '**사용자가 원하는 기능을 수행하는가?**'를 테스트 한다는 말.

A. 신택스 기법

① 문법을 정해 두고, 적합/부적합 입력 값에 따른 예상 결과가 제대로 나오는지를 테스트한다.

B. 동등 분할 기법

① 출력이 동일한 입력 범위들을 구하고, 각 범위에 대하여 실제 출력이 맞는지 검사한다.

② 다만 엄밀하게 입력 가능한 모든 값을 넣어본다는 말은 아니다. 적당한 값을 범위 별로 하나씩 준비하여 넣어본다는 말이다.

표 9-6 동등 분할 기법의 예

테스트 케이스	1	2	3	4
점수 범위	0~69점	70~79점	80~89점	90~100점
입력 값	60점	73점	86점	94점
예상 결과 값	0원	200만 원	400만 원	600만 원
실제 결과 값	0원	200만 원	400만 원	600만 원

C. 경계 값 분석 기법

① 동등 분할 기법과 유사하지만, 입력할 데이터를 각 범위의 경계 값으로 정하는 기법이다.

② 경계 값, 경계 이전 값, 경계 이후 값으로 검사를 수행한다.

표 9-7 경계 값 분석 기법의 테스트 예

테스트 케이스	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
입력 값	−1	0	1	69	70	71	79	80	81	89	90	91	99	100	101
예상 결과	오류	0	0	0	200	200	200	400	400	400	600	600	600	600	오류
실제 결과	오류	0	0	0	200	200	200	400	400	400	600	600	600	600	오류

D. 원인-결과 그래프 기법

= 가능한 모든 입력에 따른 출력 결과를 그래프로 만들고, 해당 그래프로 '의사 결정 테이블'을 만들어서 테스트를 수행한다.

= 즉, 가능한 모든 출력 값에 대하여, 올바른 값과 아닌 값을 지정하기 위한 수단이 '원인-결과'그래프인 것이다. (의사 결정 테이블은 일종의 진리표이다.)

<- 여러 입력 조건들을 조합하여, 최종적으로 SW가 출력 가능한 모든 경우의 수를 계산해볼 수 있다는 장점이 있다.

테스트 과정은 다음과 같다.

① 프로그램을 '원인-결과 그래프'로 변환할 만한 크기로 분할.

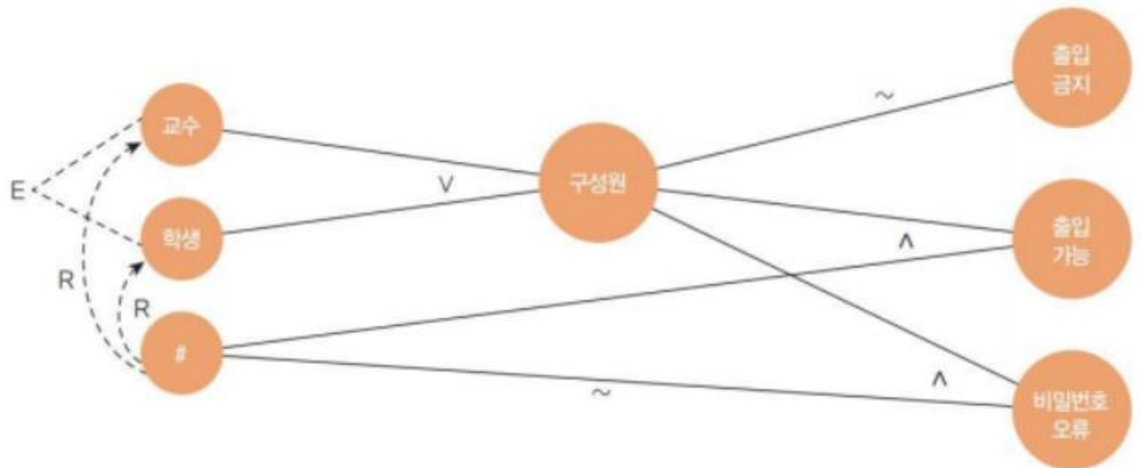
② 원인과 결과를 찾는다. <- 요구분석명세서, 설계서, 프로그램에서 원인과 결과를 모두 찾아낸다. (즉, 모든 제어문을 찾고, 입력과 출력을 분석한다는 말이다.)

③ 원인-결과 그래프 작성. 〈- 원인과 결과를 연결하기 위한 수단으로 그래프를 사용하는 것이다. 원인도 노드, 결과도 노드이다.

④ 그래프에 제한 조건을 표시한다. 〈- 논리식의 조합을 '제한 조건'이라 칭하고 있다. (기본적인 논리 연산은 원인-결과 그래프를 처음 작성할 때 함께 포함한다.)

표 9-9 원인-결과 그래프의 제한 조건 기호

명칭	기호	설명	
배타적exclusive 관계	E (A, B)	2개가 동시에 존재할 수 없다.	A=1이면 B=0 A=0이면 B=1 (A=B=0은 가능, A=B=1은 불가능)
포함inclusive 관계	I (A, B)	적어도 둘 중 한쪽은 성립한다.	A=1일 때 B=1 또는 0 B=1일 때 A=1 또는 0 (A=B=0은 불가능)
선택only one 관계	O (A, B)	항상 한쪽만 성립한다.	A=1이면 B=0 A=0이면 B=1 (A=B=0은 불가능)
필요require 관계	R (A, B)	한쪽이 성립하면 다른 쪽도 성립한다(B가 성립하기 위해 A가 반드시 필요하다).	A=1이면 B=1 또는 0 A=0이면 B=0 (B=1은 불가능)
강요mask 관계	M (A, B)	한쪽이 성립하면 다른 쪽은 성립하지 않는다.	A=1이면 B=0



⑤ 의사 결정 테이블로 변환한다. 〈- 모든 입력 항목에 대한, 테스트 케이스별 나와야 하는 값들을 표시한 진리표 작성.

표 9-11 성적 처리에 대한 의사결정 테이블

입력 항목 \ 테스트 케이스		1	2	3	4	5	6	7	8
총점	90 이상	T	T	F	F	F	F	F	F
	80 이상	F	F	T	T	F	F	F	F
	70 이상	F	F	F	F	T	T	F	F
	70 미만	F	F	F	F	F	F	T	T
리포트	제출	T	F	T	F	T	F	T	F
결과 값	A^+	T	F	F	F	F	F	F	F
	A^0	F	T	F	F	F	F	F	F
	B^+	F	F	T	F	F	F	F	F
	B^0	F	F	F	T	F	F	F	F
	C^+	F	F	F	F	T	F	F	F
	C^0	F	F	F	F	F	T	F	F
	D	F	F	F	F	F	F	T	F
	F	F	F	F	F	F	F	F	T

⑥ 의사 결정 테이블대로 SW가 동작하는지 검사.

7. 구현 기반 테스트

= 화이트박스 테스트와 유사.

= 코드 기반 테스트. (SW 코드를 테스트 설계의 기반으로 사용. 다만, 입력 데이터를 투입하고, 실행 상태를 확인하며 테스트를 진행하기에 동적 테스트로 분류된다.)

A. 테스트 절차

① 테스트 데이터 적합성 기준 설정.

- 가능한 모든 입력을 시도해 본다면 좋겠지만, 그럴 수 없다. 시간과 비용은 한정되어 있으니, 우리는 적절한 기준으로, 어디까지, 얼마나, 어디를 검사할지를 설정해야 한다.

② 테스트 데이터를 생성한다.

- 테스트 데이터 적합성을 기준을 만족하는 테스트 데이터를 만든다.
- 각종 명세와 원시 코드를 분석하여 기준을 만족하는 테스트 데이터를 생성한다는 뜻.

③ 테스트 실행.

- 정말 다양한 방법이 있다. 방법별로 복잡도와 소요 시간이 다르기에, 테스트의 목적과 조건에 맞는 방법을 선택해야 한다.
- 직후의 항목 B에서 설명한다.

B. 화이트박스 테스트의 종류

① 문장 검증 기준

- 원시 코드 내의 모든 문장이 최소 1회 이상 실행될 수 있는 테스트 케이스를 선정하여 검사.
- 조건문의 조합 오류를 검사하지 못한다는 문제 있음.

② 분기 검증 기준 (결정 검증 기준)

- 모든 조건문에 대하여 '참이 되는 경우와 거짓이 되는 경우'가 최소 1회 이상 실행되는 테스트 케이스를 선정하여 검사.
- 모든 조건문에 대하여 검사를 한다는 점에서 '문장 검증'보다는 강화되었다.
- 그러나 아직도 조건문 자체의 오류를 찾아낼 수는 없다. 이는 조건문이 여러 논리연산을 결합하여 만들어졌기 때문이다. 소프트웨어

에서 복합 논리식은 수학과 달리 왼쪽에서부터 차례로 연산되기에, or로 연결된 논리식의 우측 항의 연산이 무시될 수 있다.

- 이런 문제를 해결하기 위하여 다음의 '조건 검증 기준' 등장.

③ 조건 검증 기준

- 이전의 '분기 검증 기준'에서 잡아내지 못한 개별 조건식의 오류를 잡아낼 수 있다.
- 모든 복합 조건식을 개별 조건식으로 분해하여, 모든 개별 조건식에 대한 T와 F인 경우를 검사한다.
- 다만 개별 조건식은 자신과 연결된 다른 논리 연산과 한 덩어리이기 때문에, 모든 논리 값을 테스트 케이스로 설정하여도 and와 or 연산의 오류를 잡아낼 수 없다.
- or는 이미 설명하였으니, and를 예로 들자.
  and로 연결된 복합 조건식에서 첫 조건식이 F인 경우, 우측 연산은 무시된다.
  〈- 이것을 '마스크 문제'라고 한다.

④ 다중 조건 검증 기준

- 개별 조건식의 오류를 '조건 검증 기준'에서 다 잡아냈지만, 마스크 문제를 해결할 수 없었다. 그러므로 우리는 마스크 문제가 발생 가능한 모든 조건식에 대하여, 가려지는 부분 조건식을 검사해야 한다.
- and 연결에서 F -〉 우측 조건식에 대하여 T와 F 검사.
- or 연결에서 T -〉 우측 조건식에 대하여 T와 F검사.
- 위와 같이 데이터를 생성하는 기준이 '다중 조건 검증' 기준인 것.

⑤ 기본 경로 테스트

- 원시 코드의 ‘독립적인 경로’가 최소한 한 번씩은 실행되는 테스트 케이스를 찾아서 검사.
- 모든 ‘독립적인 경로’를 검사하는 것이 목표.
- 검사 방법
  = (순서도를 기반으로) 흐름 그래프를 작성하여 독립적인 경로를 찾아내어, 순환 복잡도(CC)를 구한다.
  -〉 순환 복잡도 개수만큼의 테스트 케이스 생성.
  -〉 검사.
- 순환 복잡도(Cyclomatic Complexity)
  = 원시 코드의 복잡도를 정량적으로 평가하는 방법.
  = ‘독립적인 경로’ 개수. (다른 계산 방법도 둘 있으나, 결과는 같다. 독립 경로의 개수가 곧 CC 값이다.)
- ‘독립적인 경로’
  = 코드의 최종 위치에 도달하는 모든 경로의 수.
  = 화살표와 노드로 둘러싸인 구역.

C. 소프트웨어 개발 단계에 따른 테스트

= 소프트웨어 개발 단계의 순서와 짝을 이루어 테스트를 함.
V 테스트의 경우 다음의 다섯 단계를 거친다.

① 단위 테스트 (개발자가 수행)

= 모듈 테스트.
= 구현 단계에서 각 모듈의 개발이 완료된 후, 요구분석명세서 대로 잘 구현되었는지를 테스트한다. 〈- 일반적으로 화이트박스 테스트 같은 구조적 테스트를 주로 시행한다.

- 모듈의 테스트에 아직 개발되지 않은 상위, 하위 모듈이 필요한 경우가 있다. 이때 가상의 모듈을 만들어서 테스트를 할 수 있고, 각각은 다음과 같이 표현된다.
- 가상 상위 모듈: 테스트 드라이버 〈- 테스트 할 모듈 호출함.
- 가상 하위 모듈: 스텁 모듈 〈- 테스트 대상이 스텁 모듈을 호출함.

② 통합 테스트 (개발자가 수행)

= 단위 테스트가 끝난 모듈들을 통합하는 과정에서 발생할 수 있는 오류를 찾는 테스트.

<- 통합 방식은 세가지가 있다.

- Big-bang 테스트: 단위 테스트가 끝난 모듈들을 한꺼번에 결합하여 테스트를 수행하는 것이다.
  <- 소규모 프로그램에서나 할 수 있다.
  <- 어디서 오류가 났는지 확인하기 어렵다.

- 하향식 기법: 시스템을 구성하는 맨 위의 모듈부터, 점차 하위 모듈 방향으로 통합하며 테스트.
  <- 맨 위부터 테스트 하므로, 적절한 스텁 모듈을 만들어서 테스트를 해야 한다.
  <- 깊이우선 혹은 너비우선 방식으로 통합해가며 테스트 한다.
  <- 상위 모듈부터 테스트 하므로, 시스템 전체에 영향을 주는 오류를 일찍 발견할 수 있다는 장점이 있다.
  <- 다수의 스텁 모듈을 만들어야 한다면, 비용이 많이 들 수 있다.

- 상향식 기법: 가장 아래에 있는 모듈부터 통합하며 테스트.
  <- 맨 아래부터 테스트를 하므로, 적절한 테스트 드라이버를 만들어서 테스트 해야 한다.
  <- 최하위 모듈을 개별적으로 병행하여 테스트할 수 있다는 장점이 있다. 그러나 상위 모듈에서 오류가 발견된다면, 그 모듈과 관련된 모든 하위 모듈을 다시 테스트 해야 한다는 단점이 있다.

③ 시스템 테스트 (개발자가 수행)

= 모듈이 모두 통합된 후, 사용자의 요구사항을 만족하는지 확인하는 테스트이다.

= 실제 사용 환경과 유사한 테스트 환경을 만들고, 요구분석명세서에 명시한 기능 요구사항과 비기능 요구사항을 충족하는지 검사한다.

<- 주로 부하를 많이 주며, 비기능적 테스트(성능, 안전성, 확장성 등 등)를 중심으로 수행한다.

④ 인수 테스트 (사용자가 수행)

= 시스템을 인수하기 전, 요구분석명세서에 명시된 사항을 충족시켰는지 검사.

<- 시스템이 사용자에게 정상적으로 인수되면 프로젝트는 종료된다.

- 알파 테스트: 내부 테스트. 회사 내의 다른 직원에게 개발자 환경에서 테스트를 시킨다.
- 베타 테스트: 상품으로 내어놓기 전에, 특정 사용자에게 미리 사용해 보도록 배포하여 테스트 하는 것.

⑤ 회귀 테스트 (개발자와 사용자가 함께 수행)

= 수정된 모듈과 관련된 모듈들을 문제가 없는지 다시 테스트 하는 것.

- 만일 인수 테스트에서 문제가 발견된 경우, 그 문제를 해결하고 나서 제대로 해결되었는지를 사용자에게 확인시키는 테스트를 '확정 테스트'라고 한다.
- 확정 테스트를 수행하고 나서 '회귀 테스트'를 해야 한다.
- 수정을 위한 회귀 테스트: 사용자에게 전달하기 직전 새로운 오류가 발견된 경우, 그 오류를 수정하고 나서 다시 수행하는 회귀 테스트.
- 점진적 회귀 테스트: 시스템을 사용하던 중에 일부 기능을 추가한 경우, 새로운 버전을 만들고 다시 테스트 하는 것이다.

9장 요약 끝.

10장 요약 시작.

10장: 품질.

1. SW 품질의 정의
   A. 많은 학자들이 정의한 내용을 적절히 요약하면 다음과 같다.
      ① ‘요구분석명세서에 서술된 기능과 성능을 만족하는 정도.’
      ② ‘사용자의 요구와 부합하는 정도.’

2. 관점에 따른 SW 품질
   A. 다양한 관점에서 SW 품질을 인식할 수 있다.
      ① PM 관점 – 계획 이상의 비용 부담이 발생하지 않는 SW.
      ② 개발자 관점 – 개발하기 쉽고, 수정이 쉬운 SW.
      ③ 유지보수자 관점 – 가독성이 높고 이해하기 쉬운 SW.
      ④ 구매 담당자 관점 – 가격이 저렴하고 필요한 기능과 성능을 지닌 SW.
      ⑤ 사용자 관점 – 배우기 쉽고, 사용이 편하고, 빠르고, 기능이 많은 SW.

3. 품질 목표
   A. 미국 품질보증연구소에서 정의한 품질 목표는 다음과 같다.
      ① 정확성 – 요구분석명세서와 일치하는 정도.
      ② 신뢰성 – 기능이 일관되고 정확한 정도.
      ③ 효율성 – 기능이 시간과 기억 용량을 사용하는 정도.
      ④ 무결성 – 허가받지 않은 사용자의 접근을 차단할 수 있는 정도.
      ⑤ 사용성 – 사용하기 편리한 정도.
      ⑥ 유지보수 용이성 – 오류를 찾아 수정하기 쉬운 정도.
      ⑦ 테스트 용이성 – 쉽고 철저히 테스트할 수 있는 정도.
      ⑧ 유연성 – 새로운 기능을 추가하기 쉬운 정도.
      ⑨ 이식성 – 다른 하드웨어 환경에 이식하기 쉬운 정도.
      ⑩ 재사용성 – 시스템을 다른 애플리케이션에서 사용하기 쉬운 정도.
      ⑪ 상호운용성 – 다른 SW와 쉽게 정보를 교환할 수 있는 정도. (연계성)

4. 품질 특성
    A. 다양한 학자들이 위의 품질 목표를 기반으로 하여, 저마다의 품질 요소를 정의한 바 있다.
    B. 교재 p.476 참고.

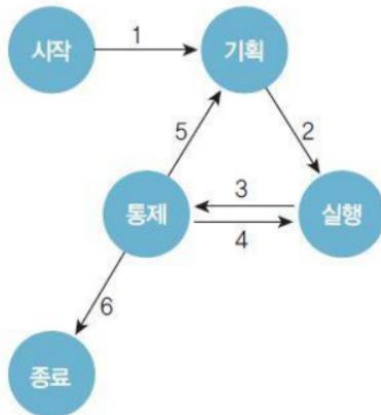
-참고. 10장의 후반부('제품 품질 특성 평가 모델'부터 끝까지의 내용)는 넘어가셨다.

10장 요약 끝.

11장 요약 시작.

11장: 프로젝트 관리

1. 프로젝트
   = 유일한 제품이나 서비스를 만들기 위해, 일정한 기간을 정해놓고 수행하는 작업.
   A. 프로젝트의 특징
      ① 한시성 – 일의 시작과 끝이 명확히 정해져 있다.
      ② 유일성 – 기간이 종료되어 만들어 내는 인도물은 유일하다.
      ③ 참여자의 일시성 – 참여 인력은 종료되면 해체된다.
      ④ 한정성 – 프로젝트가 종료되면 사용된 자원은 원래의 위치로 돌아가거나 없앤다.
   B. 프로젝트 관리
      = 프로젝트 완수를 위한 모든 작업을 수행해야 한다. 진행 상황 관리, 예상을 벗어난 일에 대한 대응, 인력 배치, 자원 계획 및 공급, 활동에 필요한 기술 기법 도구의 배치 등등….
      <- 프로젝트 관리법에 대한 표준이 제시되어 있다. (PMBOK)
2. 프로젝트관리지식체계: PMBOK (Project Management Body of Knowledge)
   A. 다섯 가지 프로세스 그룹의 상호작용을 정의하고 있다.



프로젝트의 프로세스 그룹 관계도

      ① 시작 그룹: 프로젝트를 구성하는 단계를 정의하고 승인한다.
      ② 기획 그룹: 프로젝트 목표 설정 + 목표 달성을 위한 계획 수립.
      ③ 실행 그룹: 계획대로 실제 프로젝트 수행.
      ④ 통제 그룹: 계획 대비 목표의 진척 상황 감시 및 측정 수행.
      ⑤ 종료 그룹: 프로젝트를 공식적으로 종결. (계약 종료, 관리 종료….)
   B. 이상으로 깊게 PMBOK을 파고들지는 않는다. (교재 내용 일부 스킵)

3. 형상 관리

= 특정 항목의 변화에 대해 관리하는 것.
(시스템의 통합과 일치를 보장하기 위해서!)
= 개발 중 발생하는 모든 산출물을 체계적으로 관리하여, 소프트웨어의 형상을 관리하고 유지하는 기법.

A. 소프트웨어는 항상 변한다. 개발 생명주기 전체에서 계속 시스템은 변화하고, 모든 작업에서 다양한 종류의 산출물이 쌓인다. 이것들을 잘 관리해야 한다!

B. 형상 관리에는 다양한 행위가 속한다. 대표적인 형상 관리는 다음과 같다.

① 변경 관리: 업무 환경의 변화 기록 + 기술 환경의 변화 기록.
- 변경되는 항목 뿐만 아니라 주변도 기록해야 한다는 말이다.
- 업무 환경 변화: 고객의 요구, 시장 상황, 예산이나 일정 변화….
- 기술 환경 변화: 하드웨어나 운영체제의 변화….

② 버전 관리
- 언제라도 과거에 릴리즈한 파일을 가지고 작업할 수 있도록, 각 버전의 정보를 데이터베이스화 하여 관리해야 한다.
- 언제 어느 버전의 정보가 필요할 지 알 수 없기 때문이기도 하며, 파일의 이력과 차이점을 관리하여 각 원시 파일과 문서를 유용하게 활용하기 위해서이기도 하다.

C. 형상 관리의 필요성

① 변경되어가는 상태에 대한 가시성 확보 -> 가장 안정적인 버전의 SW 유지.
② 변경에 관한 질문에 답변 가능 -> 누가 언제 무엇을 왜 변경했는지 추적.
③ 생산성과 안정성 향상 -> 좋은 품질의 SW 생산과 유지보수 용이.
④ 적절한 변경 관리 -> 무절제한 변경의 사전 예방 + 변경에 따른 부작용 최소화.

D. 형상 관리 효과

① 프로젝트의 적절한 통제 가능 -> 체계적이고 효율적인 관리 가능해짐.

② 가시성과 추적성 확보 -> 소프트웨어의 생산성과 품질 향상 가능.

-참고. 유지보수 파트(p.545부터)는 강의하지 않으셨다.

11장 요약 끝.

이상으로 〈쉽게 배우는 소프트웨어 공학〉(김치수 지음, 한빛아카데미)의 6장 ~ 11장 내용 요약 및 정리가 끝났다.