<1장. 서론>

*** 소프트웨어 공학** (p.4)

-소프트웨어 중요성과 특징

1. 국가 기반 시설, 공익 사업을 컴퓨터 기반 시스템으로 제어
2. 대부분의 전기 시스템은 컴퓨터와 제어 소프트웨어를 탑재
3. 제조, 유통, 금융 등 분야의 컴퓨터화
4. 소프트웨어 시스템은 추상적이며 무형이다.
   4-1. 제품의 특성으로 인한 제약이 없으며, 물리적 법칙이나 제조공정으로부터 영향이 없다.
   4-2. 물리적 제약이 없기 때문에, 소프트웨어 시스템이 극도로 복잡해진다.
   이로 인해 이해하기 어려우며 변경하는데 많은 비용이 들게 된다.

- 다양한 종류의 소프트웨어 시스템

1. 단순한 임베디드 시스템부터 복잡한 정보 시스템까지 다양한 소프트웨어 시스템이 존재한다.
2. 소프트웨어 시스템의 종류에 따라 다른 접근법이 필요하다.
3. 소프트웨어 공학에는 보편적인 표기법이나 방법, 기법이 존재하지 않는다.

*** 소프트웨어 실패** (p.4-5)

- 소프트웨어 장애의 2가지 요인 (소프트웨어 실패)

1. 시스템 복잡도 증가
   : 시스템을 더 신속하게 공급해야 하며 더 크고 더 복잡한 시스템이 필요해졌기 때문이다.

2. 소프트웨어 공학 방법론 사용의 실패
   : 소프트웨어 공학 방법과 기술을 사용하지 않을 시 소프트웨어가 안정성, 신뢰성이 부족하게 된다.

- 소프트웨어 FAQ

소프트웨어란?	컴퓨터 프로그램과 관련된 문서들, 특정 고객 및 일반 시장을 고려해서 소프트웨어 개발 가능
좋은 소프트웨어 특징	요구되는 기능과 성능을 사용자에게 제공해야 하고, 유지보수성, 확실성, 사용성이 좋아야 한다.
소프트웨어 공학이란?	초기 구상 단계부터 운영과 유지보수까지 포함하는 소프트웨어 생산의 모든 관점과 연관된 공학 분야이다.
소프트웨어 공학 기본활동	명세화 (Specification), 개발 (Development), 검증 (Validation), 진화 (Evolution)
소프트웨어 공학이 직면한 주요 도전과제	증가하는 다양성, 인도기간 단축 요구, 신뢰할 수 있는 소프트웨어 개발에 대응하는 것
소프트웨어 공학 비용	60% 가 개발비용, 40% 테스트 비용. 맞춤형의 경우 종종 진화 비용이 개발비용을 넘어선다.
최선의 소프트웨어 공학 기법과 방법은 무엇인가?	.모든 소프트웨어 프로젝트는 전문적으로 관리되고 개발되어야 하며 시스템 유형에 따라 적합한 기법이 다르기 때문에 우열을 가를 수 없다.

*** 소프트웨어 제품** (p.7)

- 소프트웨어 제품
    1. 일반 제품 (Generic)
    1-1. 수평적 제품 (Horizontal) : 응용 분야에 관계 없이 사용되는 애플리케이션
    1-2. 수직적 제품 (Vertical) : 특정 응용 분야를 위한 애플리케이션

    2. 맞춤식 제품 (Customized, Bespoke)
    : 특정 고객을 위해 해당 고객에 맞추어 개발하는 시스템

    3. ERP (Enterprise Resource Planning) 시스템
    : 전사적 자원 관리 시스템
    : 일반 제품으로 개발하여 특정 고객의 요구사항을 반영

- 제품 명세
    1. 일반 제품
    : 소프트웨어 명세를 개발 조직이 소유하고 변경 사항을 결정

    2. 맞춤형 제품
    : 소프트웨어 명세를 고객이 소유하고 변경 사항을 결정

*** 좋은 소프트웨어의 필수적 특성** (p.8)

1. 수용성 (Acceptability)
: 소프트웨어는 해당 유형의 사용자에 의해 수용될 수 있어야 한다.
: 이해하기 쉽고, 사용하기 쉽고, 기존 시스템과 호환성 있어야 한다.
: Understandable, Usable, Compatible

2. 확실성, 보안성 (Dependability and Security)
: (확실성) 시스템 장애시에 물리적 또는 경제적 손실이 없어야 한다.
: (보안성) 악의적 사용자가 시스템에 접근하거나 피해를 줄 수 없어야 한다.
: 신뢰성, 보안성, 안전성을 포함
: Reliability, Security, Safety

3. 효율성 (Efficiency)
: 메모리나 CPU 타임 등의 시스템 자원을 낭비해서는 안된다.
: 응답성과 처리시간 메모리 활용 등을 포함한다.
: Responsiveness, Processing Time, Memory Utilization

4. 유지보수성 (Maintainability)
: 소프트웨어는 고객의 변경 요구에 맞도록 진화할 수 있게 작성되어야 한다.

*** 소프트웨어 공학** (p.8)

- 소프트웨어 공학이란
  : 초기 시스템 명세를 작성하는 것부터 시스템이 사용되고 유지보수 되는 것까지
  소프트웨어 제품화와 관련된 모든 관점을 다루는 공학 분야다.

  공학적 학문 분야
  : 적절한 이론, 방법, 도구 등을 적용하여 조직, 재정 등 제약 하에서 해결책을 찾는다.
  : 적용 가능한 이론이나 방법이 없는 경우에도 문제 해결을 위해 노력해야 한다.

  소프트웨어 생산의 모든 관점
  : 기술적인 소프트웨어 개발 과정 뿐만 아니라 프로젝트 관리와 개발 지원 도구,방법,이론의 개발 등을 포함

- 소프트웨어 공학의 중요성
  : 개인과 사회가 소프트웨어 시스템에 많이 의존적이다.
  안정적이고 신뢰할 수 있는 시스템을 경제적이고 빠르게 개발할 필요가 있다.

  : 전문적인 소프트웨어 개발을 위해 소프트웨어 공학 방법 및 기법을 사용하는 것이 보통 비용이 적게 든다.
  소프트웨어 공학 방법을 사용하지 않을 시 테스팅, 품질보증, 장기적 유지보수 등을 위한 비용이 증가한다.

- 소프트웨어 프로세스
  : 소프트웨어 제품을 만드는 데 필요한 일련의 활동들이다.

- 소프트웨어 프로세스의 공통적인 4가지 기본사항
  1. 소프트웨어 명세화 (Specification)
  : 고객과 개발자가 소프트웨어의 기능과 운영 제약조건을 정의

  2. 소프트웨어 개발 (Development)
  : 소프트웨어를 설계하고 프로그램을 작성

  3. 소프트웨어 검증 (Validation)
  : 소프트웨어가 고객이 요구하는 것이 맞는지 확인

  4. 소프트웨어 진화 (Evolution)
  : 고객과 시장의 요구에 따라 소프트웨어를 수정

* 애플리케이션 유형 (p.11-12)

- 애플리케이션 유형
    1. 독립형 애플리케이션
    2. 대화형 트랜잭션 기반 애플리케이션
    3. 임베디드 제어 시스템
    4. 일관처리 시스템
    5. 오락 시스템
    6. 모델링 및 모의실험 시스템
    7. 데이터 수집 및 분석 시스템
    8. 복합 시스템

- 소프트웨어 공학 기본 사항
    1. 관리되고 이해할 수 있는 개발 프로세스를 이용하여 개발해야 한다.
    2. 모든 유형에 시스템에서 확실성 (Dependability) 과 성능 (Performance) 는 중요하다.
       장애 없이 의도대로 작동해야 하며, 동작 중엔 안전해야 하며, 자원낭비 없이 동작해야 한다.
    3. 소프트웨어 명세와 요구사항을 이해하고 관리하는 것이 중요하다.
    4. 기존의 자원을 활용 -> 이미 개발되어 있는 소프트웨어를 재사용(Reuse) 해야 한다.

<2장 소프트웨어 프로세스>

*** 소프트웨어 프로세스** (p.32)

- 소프트웨어 프로세스
  : 소프트웨어 시스템을 제품화하기 위한 활동들의 집합
  : 다양한 유형의 시스템 개발에 적용 가능한 보편적 소프트웨어 공학 기법은 존재하지 않는다.

- 기본적인 소프트웨어 공학 활동
  1. 소프트웨어 명세화 (Specification)
  : 소프트웨어의 기능과 운영 제약조건을 정의해야 한다.

  2. 소프트웨어 개발 (Development)
  : 시스템의 구조를 정의하고 구현, 명세와 일치하는 소프트웨어를 개발해야 한다.

  3. 소프트웨어 검증 (Validation)
  : 시스템이 고객이 요구하는 것인지 점검해야 한다.

  4. 소프트웨어 진화 (Evolution)
  : 고객의 요구 변화에 따라 시스템을 변경해야 한다.

- 프로세스 설명에 포함되어야 하는 것
  1. 제품과 산출물 (Product and Deliverable)
  : 프로세스 활동의 결과물
  : 명세화, 설계, 구현, 테스팅 등

  2. 역할 (Role)
  : 프로세스에 참여하는 사람들의 책임
  : 프로젝트 관리자, 형상 관리자, 프로그래머 등

  3. 사전/사후 조건 (Pre/Post Condition)
  : 프로세스 활동이 이루어지거나 제품이 만들어지는 전후에 만족되어야 하는 조건들

*** 계획 주도 프로세스와 애자일 프로세스** (p.33)

- 계획 주도 프로세스 (Plan-driven Process)
  : 모든 프로세스 활동(Activities)을 미리 계획하고 계획 대비 실적을 측정한다.

- 애자일 프로세스 (Agile Process)
  : 계획을 점증적으로 세우고 고객의 요구를 반영하여 프로세스를 간단히 변경한다.

=> 대부분의 실제 프로세스는 계획 주도와 애자일 프로세스를 둘 다 포함한다.

*** 소프트웨어 프로세스 모델** (p.34)

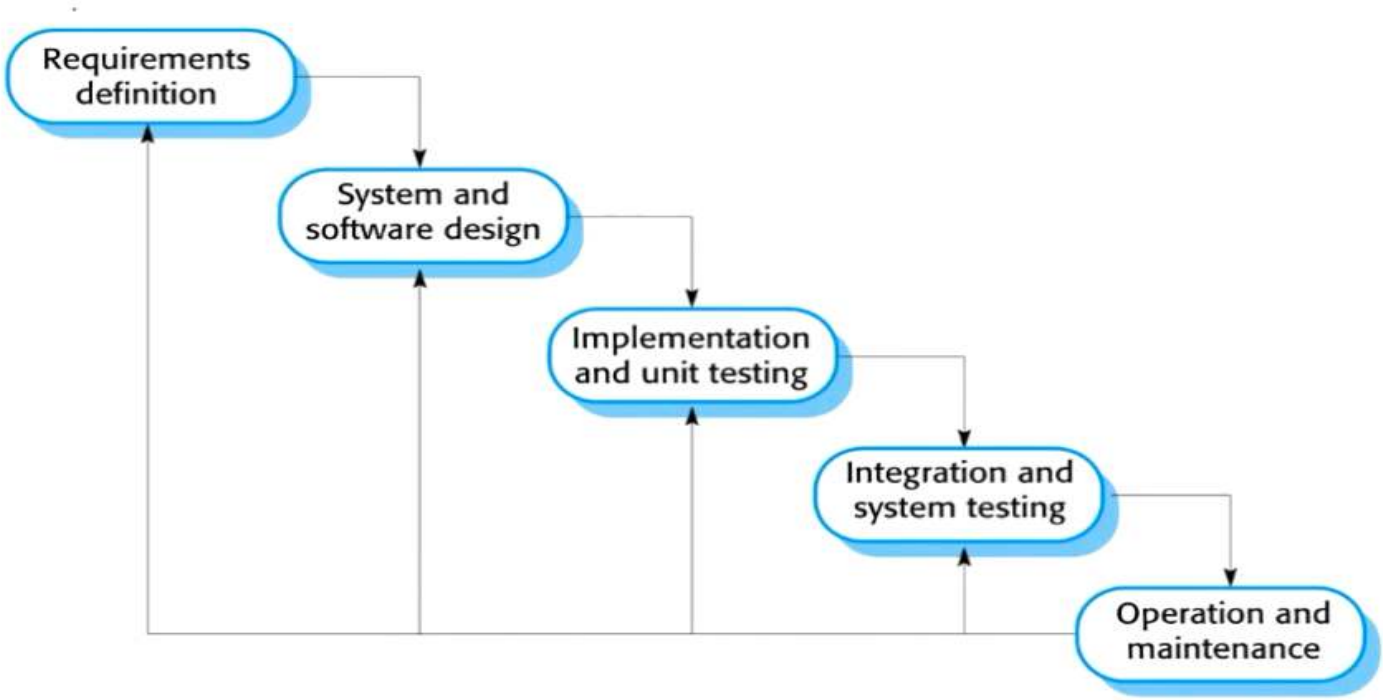
- 일반적인 프로세스 모델

  1. 폭포수 모델 (Waterfall)
  : 계획 주도
  : 명세화, 개발, 검증, 진화를 개별적인 프로세스 단계로 구분한다.

  2. 점증적 개발 (Incremental)
  : 계획 주도 또는 애자일
  : 명세화, 개발, 검증 활동이 중첩된다.
  : 시스템은 일련의 증가분 (Increment) 으로 개발된다.
  : 증가분이랑 이전 버전과 비교하여 추가된 기능을 의미한다.

  3. 통합 및 설정 (Integration and Configuration)
  : 계획 주도 또는 애자일
  : 재사용할 수 있는 컴포넌트나 시스템을 설정하고 통합하는데 초점을 맞춘다.

* **폭포수 모델** (p.35)



- 폭포수 모델의 단계

1. 요구사항 분석 및 정의
 : 시스템 사용자와의 면담을 통해 시스템의 서비스, 제약조건과 목표를 설정한다.
 : 설정한 내용은 더 구체화 시킨 후 시스템 명세서로 사용한다.

2. 시스템 및 소프트웨어 설계
 : 요구사항을 하드웨어와 소프트웨어 시스템으로 나누어 할당하고, 전체 시스템 아키텍처를 작성한다.
 : 소프트웨어 설계 = 소프트웨어 시스템 추상화와 이들 간의 관계를 파악하고 기술하는 활동을 포함한다.

3. 구현 및 단위 테스팅
 : 소프트웨어 설계를 프로그램으로 실체화한다.
 : 단위 테스팅 = 각각의 단위가 주어진 명세를 만족하는지 검증하는 활동이다.

4. 통합 및 시스템 테스팅
 : 개별 프로그램 단위나 프로그램을 통합하여 소프트웨어 요구사항을 만족하는지 전체 시스템을 검사한다.
 : 테스트 후에는 소프트웨어 시스템을 고객에게 전달한다.

5. 운영 및 유지보수
 : 시스템이 서리되고 사용되는 단계로, 정상적인 경우 이 활동이 가장 긴 생명 주기 단계다.
 : 이전 단계에서 발견하지 못했던 오류를 수정, 시스템 단위의 구현을 개선, 새로운 요구사항을 반영하여 시스템 서비스를 향상시킨다.

*** 폭포수 모델의 특징과 단점** (p.36-37)

- 폭포수 모델의 특징
  : 각 단계의 결과로 하나 혹은 하나 이상의 승인된 문서가 존재해야 한다.
  : 원칙적으로 한 단계가 종료되야 다음 단게를 시작한다.

- 폭포수 모델의 단점
  : 실제 소프트웨어 개발은 각 단계가 중첩된다.
  : 한 단계 (구현, 설계)에서 다른 단계 (설계, 요구사항)로의 피드백이 일어난다.
  : 이전 단계의 문서를 수정하게 되면 프로세스 지연이 일어난다.
  : 결국 추가 변경을 반영하지 않도록 명세서를 확정하게 된다.
  => 변화하는 고객의 요구사항에 대응하기 어렵다.

- 폭포수 모델이 적합한 분야
  : 요구사항 이해도가 높으며, (설계, 구현 중) 요구사항 변경이 제한되는 분야

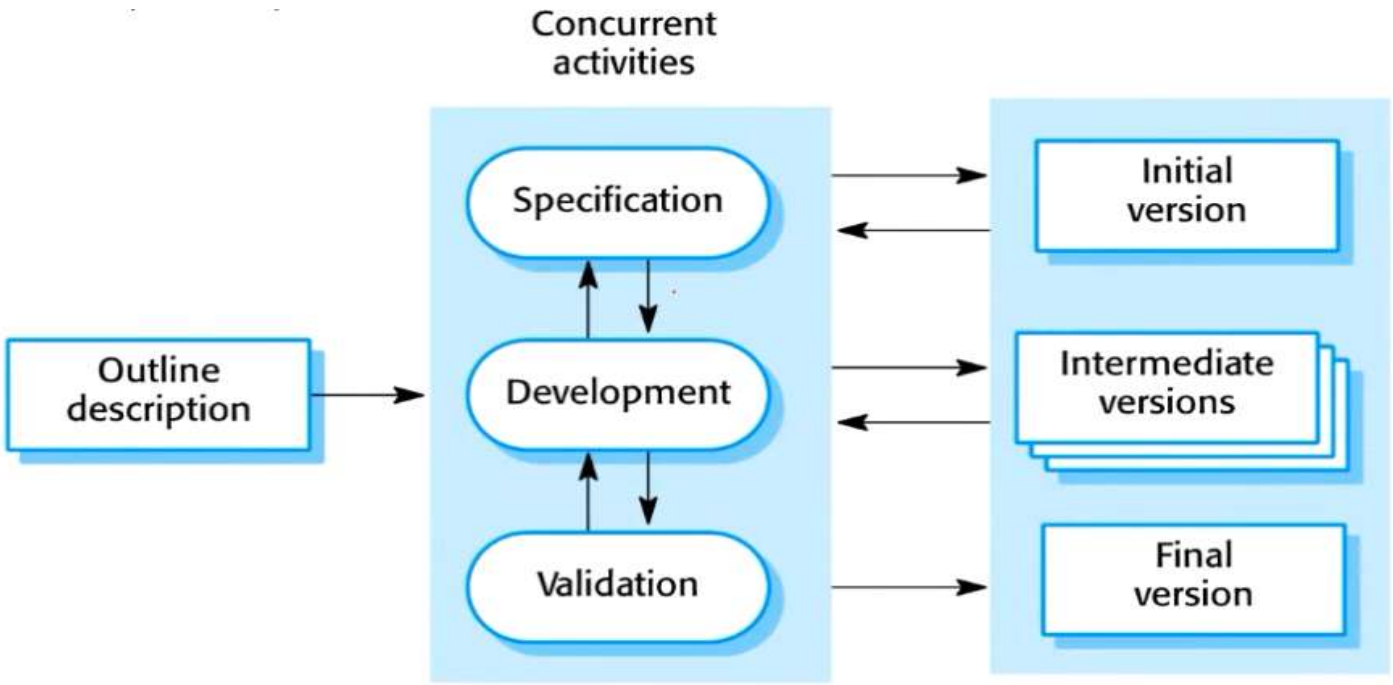
  1. 임베디드 시스템
  : 하드웨어와 연동해야 함으로 구현 시 까지 의사결정을 미룰 수 없기 때문이다.

  2. 중대한 시스템
  : 명세와 설게애 댜헌 뷴석 및 검토가 필요하기 때문이다.

  3. 대규모 소프트웨어 시스템
  : 여러 회사, 장소에서 개발하기 때문에 완성된 명세서가 필요하기 때문이다.

- 폭포수 모델이 적합하지 않은 분야
  1. 자유로온 팀 커뮤니케이션이 가능
  2. 소프트웨어 요구사항이 빨리 변경되는 상황
  => 반복적 개발과 애자일 방법론이 적합하다.

* **점증적 개발** (p.38)



- 점증적 개발 (Incremental Development)
  : 초기 구현을 개발하고, 사용자와 다른 사람들로부터 피드백을 받아서,
  여러 버전을 거쳐 소프트웨어를 진화시킴으로써 요구한 최종 시스템을 개발하는 것이다.

  : 명세, 개발, 검증은 명확히 구분되지 않으며 중첩되어 있다.

  : 요구 사항이 쉽게 변하는 시스템에 적합하다. => 대부분의 비즈니스 시스템

  : 개발하면서 발생하는 변경 사항에 대처하기 쉽다.

  : 시스템의 증가분
  (시스템의 초기 증가분) -> 가장 중요하거나 긴급하게 요구되는 기능을 포함시킨다.
  (기타 시스템 증가분) -> 진척도와 고객의 우선순위에 따라 달라진다.

- 점증적 개발 장점 (p.39)
  : 요구사항 변경을 구현하는 비용이 줄어든다.
  : 이미 개발된 내용에 대해 고객의 피드백을 받기 쉽다.
  : 모든 기능을 개발하기 전에 유용한 소프트웨어를 고객에게 전달하여 설치하고 사용하게 할 수 있다.

- 점증적 개발 단점 (p.40)

1. 프로세스가 비가시적이다.
 : 관리자는 진척도를 측정하기 위해 산출물이 필요하지만 문서를 작성하는 것은 비용 측면에서 비효율적

2. 새로운 증가분 추가됨에 따라 시스템 구조가 저하되는 경향이 존재
 : 구조 저하와 코드가 복잡해지는 것을 막기 위해 정기적 리팩토링 (Refactoring)이 필요하다

=> 대규모 시스템의 개발에는 적절하지 않다.

Tip) Refactoring : 개선과 재구성.

*** 통합과 환경설정** (p.40)

- 통합과 환경설정
 : 재사용할 수 있는 코드를 찾고 요구에 맞추어 수정한 후 이미 개발한 새로운 코드와 통합한다.

- 재사용 기반 프로세스

1. 요구사항 명세화
2. 소프트웨어 발견 및 평가
3. 요구사항 정제
4. 애플리케이션 시스템 설정
5. 컴포넌트 수정과 통합

*** 프로세스 활동** (P.42-43)

- 실제 소프트웨어 프로세스
  : 명세, 설계 구현, 테스팅 등의 활동이 중첩되어 있다.

- 기본 프로세스 활동
  1. 소프트웨어 명세화 (Specification)

  2. 소프트웨어 개발 (Development)

  3. 소프트웨어 검증 (Validation)

  4. 소프트웨어 진화 (Evolution)

- 개발 프로세스 별 기본 활동 구성
  1. 폭포수 모델 : 순차적 구성

  2. 점증적 모델 : 중첩 구성

* 소프트웨어 명세화 (P.43)

- 소프트웨어 명세화 (= 요구 공학)
  : 시스템을 개발하기 위해 어떤 서비스가 필요한지를 이해하고 정의하며,
    시스템의 운영과 개발에 대한 제약사항을 찾아내는 프로세스

  : 명세화 (요구 공학) 이전에 타당성 조사 (Feasibility Study) 나 마케팅 조사를 수행 가능하다.

- 소프트웨어 명세화 (요구 공학) 프로세스의 주요 활동 3가지
  1. 요구사항 도출과 분석 (Requirements Elicitation and Analysis)
  : 기존 시스템 관찰, 잠재적 사용자 및 구매자와 의사소통, 업무 분석을 통해
    시스템 요구사항을 얻어내는 프로세스

  2. 요구사항 명세화 (Requirements Specification)
  : 요구사항 분석 과정에서 확보한 정보를 바탕으로 요구사항을 담은 문서를 작성하는 활동
  (사용자 요구사항) -> 고객 및 사용자를 위한 것으로, 시스템 요구사항에 대한 추상적 문제들에 해당한다.
  (시스템 요구사항) -> 제공할 기능에 대한 보다 상세한 설명을 포함한다.

  3. 요구사항 검증 (Requirements Validation)
  : 요구사항에 대한 현실성, 일관성, 완전성을 검사하는 활동 (Realism, Consistency, Completeness)
  : 요구사항 문서상의 오류를 발견하고 해결한다.

*** 소프트웨어 설계 및 구현** (p.44-45)

- 소프트웨어 설계
  : 시스템 명세로부터 실행가능한 시스템으로 변환하는 프로세스

- 소프트웨어 설계 기술 목록
  1 . 구현해야 하는 소프트웨어 구조
  2 . 시스템이 사용하는 데이터 모델과 구조
  3 . 시스템 컴포넌트들 사이의 인터페이스
  4 . 사용하는 알고리즘
  => 이 과정에서 설계에 대한 상세한 내용 추가 및 이전 설계를 수정하기 위해 끊임없이 되돌아가는 작업 반복

- 소프트웨어 구현
  : 프로그램 개발
  : 설계와 구현이 중첩되는 경우가 많다.
  : 프로그래밍은 개별적 활동이기 때문에 따로 준수해야 할 일반적인 프로세스가 존재하지 않는다.

- 소프트웨어 구현 시 주의할 점
  1. 테스팅 : 제거해야 할 프로그램의 결함을 찾는다 -> 결함의 존재 확인.
  2. 디버깅 : 프로그램의 결함을 찾아서 고치는 작업 -> 결함의 위치를 찾고 수정

* **소프트웨어 검증** (p.47-48)

- Verification and Validation (V&V)
    1. Verification : 시스템이 주어진 명세를 준수하는 것

    2. Validation : 시스템이 고객의 기대를 충족하는 것

    => 실제처럼 제작한 테스트 데이터를 사용해서 시스템을 실행하는 '프로그램 테스팅'은 주요한 검증 기법이다.

- 테스팅 프로세스 단계
    1. 컴포넌트 테스팅
    : 개별 컴포넌트 (함수,객체, 또는 이들의 그룹 등)를 테스트

    2. 시스템 테스팅
    : 컴포넌트를 통합하여 시스템을 구성하고 이를 테스트
    : 컴포넌트 간 예기치 못한 상호작용과 인터페이스 문제 때문에 발생한 요류를 찾는 활동
    : 기능적 / 비기능적 요구사항 만족 여부, 창발적 (Emergent) 시스템 속성을 테스팅

    3. 고객 테스팅
    : 시스템이 실제 운영을 위한 인수가 결정되기 전, 테스팅 프로세스의 최종 단계
    : 시스템 고객의 실제 데이터를 이용하여 테스트

* **소프트웨어 진화** (p.49)

- 소프트웨어 유연성 (Rlexibility)
    : 소프트웨어 유연성이라는 주요한 이유로
    점점 더 많은 소프트웨어가 통합을 통해 더 크고 복잡한 시스템으로 변화 중

    : 소프트웨어 변경은 하드웨어 변경에 비해 쉽기 때문이다.

- 개발과 유지보수
    : 개발과 유지보수의 구분이 줄어들고 있다.
    : 완전히 새로 만든 소프트웨어 시스템은 거의 없다.
    : 요구사항 변화를 지속적으로 반영하면서 소프트웨어가 진화한다.
    => 요구사항이 변화하므로 소프트웨어도 진화하고 변경되어야 한다.

*** 변경 처리** (p.50)

- 변경을 처리하고 시스템 요구사항을 바꾸기 위한 두가지 방법

1. 시스템 프로토타이핑
 : 고객의 요구사항 및 결정한 설계의 타당성을 확인하기 위해
 시스템의 한 버전이나 부분을 빠르게 개발하는 것
 : 인도 전에 사용자가 시스템을 사용하여 미리 요구사항을 개선할 수 있도록 한다.

2. 점증적 인도
 : 의견을 받거나 사용할 수 있도록 시스템 증가분을 고객에게 전달하는 것

*** 소프트웨어 프로토타이핑** (p.51)

- 프로토타입 (Prototype)
 : 제품의 아이디어를 시연하고, 디자인 선택 사항들을 시도해보고, 문제점과 해결책을 찾아내기 위해
 사용하는 소프트웨어 시스템의 초기 버전

- 프로토타입의 용도

1. 요구 공학 프로세스
 : 요구사항 도출과 검증에 도움을 준다.
 : 프로토타입을 사용해 보고 아이디어를 얻을 수 있다.

2. 설계 프로세스
 : 설계 문제에 대한 해결책을 탐색하거나 설계의 타당성을 확인하기 위한 실험용으로 사용 가능
 : 시스템을 위한 사용자 인터페이스 개발을 위해서도 사용 가능

<3장 애자일 소프트웨어 개발>

*** 신속한 소프트웨어 개발** (p.64)

- 신속한 개발과 인도 (Delivery)
  : 대부분의 비즈니스 시스템에서 신속한 개발과 배포는 가장 중요한 요구사항이다.
  ; 변화하는 비즈니스 환경에 따라 안정적인 요구사항을 얻기가 힘들다.
  : 시스템을 경험한 후에 요구사항을 명확히 알게되지만, 요구사항은 계속 변화한다.

- 기존 계획주도 프로세스의 한계
  : 계획주도 프로세스는 신속한 소프트웨어 개발에 적합하지 않음

- 애자일 기법의 등장
  (익스트림 프로그래밍) -> 애자일 개발 방법
  (스크럼) -> 애자일 프로젝트 관리

*** 애자일 개발** (p.65)

- 애자일 기법의 특징
  1. 명세화, 설계 및 구현 프로세스가 중첩된다.
  : 설계 문서화를 최소화하거나 시스템 구현을 위한 프로그래밍 환경에서 자동 생성한다.

  2. 시스템을 증가분의 연속으로 구현한다.
  : 최종 사용자와 다른 시스템 이해당사자가 각 증가분을 명세하고 평가하는데 참여한다.

  3. 개발 프로세스를 지원하기 위해 방대한 도구를 사용하게 된다.
  : 테스트 자동화 도구, 형상 관리와 시스템 통합을 지원하는 도구, UI 를 자동으로 생성해주는 도구 등

  => 고객과의 커뮤니케이션 활성화 + 문서 최소화

- 계획 주도 개발과 애자일 개발의 차이점
  1. 계획 주도 개발 (Plan - driven)
  : 프로세스 단계별로 해당하는 산출물을 생성하도록 각 단계를 분리
  : 각 단계의 산출물이 다음 단계에서 사용된다.
  : 활동 별로 반복이 이루어진다.

  2. 애자일 개발 (Agile)
  : 설계와 구현을 프로세스 중심활동으로 두고, 요구사항, 테스팅 등을 설계와 구현에 포함시킨다.
  : 요구사항과 설계를 따로 두지 않고 함께 발전시킴

*** 애자일 기법** (p.66)

- 계획 주도 접근법 특징
  : 요구사항 및 설계 등의 문서화에 중점
  : 대규모로 오래 지속되는 소프트웨어 시스템을 개발하는데 적합
  : 요구사항이 바뀌면 명세,설계, 프로그램을 함께 변경해야 한다.
  : 중소 규모 비즈니스 시스템을 개발하기에는 오버헤드가 크단 단점이 존재

- 애자일 기법 등장'
  : 개발팀이 설계와 문서 작업보다 소프트웨어 자체에 더 집중하게 한다.
  : 요구사항이 자주 변경되는 애플리케이션을 개발할 때 적합하다.
  : 반복적인 개발
  : 고객이 작동하는 소프트웨어를 빨리 받아 이후 요구사항 도출에 기여

- 애자일 선언
  1. 프로세스 도구 < 개인과 상호작용
  2. 이해하기 좋은 문서 < 작동하는 소프트웨어
  3. 계약 협상 < 고객과의 협업
  4. 계획을 따르기 < 변화에 대응

- 애자일 기법 원칙

고객 참여	고객이 개발 프로세스 전반에 밀접하게 관여해야 한다. 시스템 요구사항 제공, 우선순위 결정, 반복적으로 개발하는 시스템을 평가
변화 수용	시스템 요구사항이 변한다고 예상하여 변화를 수용할 수 있도록 시스템 설계
점증적 인도	소프트웨어는 점증적으로 개발한다. 고객이 다음 증가분에 포함해야 할 요구사항을 명세한다.
단순성 유지	소프트웨어와 개발 프로세스의 단순성에 집중한다. 가능하면 시스템의 복잡성을 제거한다.
프로세스가 아닌 사람	개발팀의 기술을 인지하고 잘 활용할 수 있어야 한다.

- 애자일 기법이 유용하게 쓰이는 경우
  1. 소프트웨어 회사가 중소규모의 제품을 개발
  2. 고객이 개발 프로세스에 참여할 수 있어야 한다.
  3. 외부 이해 당사자나 규제가 없는 조직 내에서 이루어지는 맞춤형 개발
  4. 같은 장소에 있는 팀으로 구성

*** 애자일 개발 기법** (p.68)

- 익스트림 프로그래밍 (Extreme Programming) (XP)
  : 극단적으로 반복적인 접근법
  : 하루에 여러개의 버전, 2주 마다 증분을 시도
  : 요구사항을 사용자 스토리(User Story)라고 하는 시나리오로 표현
  + 사용자 스토리를 태스크(Task)로 나누어 구현
  : 프로그래머는 짝으로 개발하고 코드를 작성하기 전에 테스트를 먼저 작성
  : 새로운 코드를 시스템에 통합하려면 모든 테스트를 통과해야 한다.
  : 시스템을 자주 배포해야 한다.

- XP 실무
  : 자세한 표는 p.69 참조
  : 대부분의 조직에서 XP 실무의 실무 적용이 어렵다고 밝혀짐
  : 실무 원칙을 골라서 사용

- 중요한 실무 원칙 4가지
  1. 사용자 스토리
  2. 리팩토링
  3. 테스트 우선 개발
  4. 짝 프로그래밍

*** 사용자 스토리** (p.70)

- XP 의 요구사항 관리
  : XP 와 같이 애자일 기법은 요구사항 변경을 지원하기 위해 별도의 요구 공학 활동을 두지 않는다.
  : 시스템 사용자가 경험할 수 있는 일종의 사용 시나리오인 '사용자 스토리'를 만들어 관리한다.
  : 고객을 개발팀과 스토리 카드를 작성, 스토리 카드를 태스크로 나누어 개발한다.

- 사용자 스토리의 장점 / 단점

  (장점) 이해하기 쉽다.
  : 요구 사항 문서나 유스케이스보다 쉽게 이해할 수 있다

  (단점) 요구사항 완전성 문제
  : 시스템의 중요한 요구사항 전체를 다룰 수 있을 정도로 제작했는가?
  : 개별 스토리에서 누락된 내용은 없는가?

*** 리팩토링** (p.72)

- 리팩토링 (Refactoring)
  : 소프트웨어 구조와 신뢰성을 개선함으로써 소프트웨어를 변경할 때 자연스럽게 발생하는 구조적 약화 해결

- XP 가 생각하는 변경 처리
  : 기존 전통적인 소프트웨어 공학 지침의 '변경을 고려한 설계' = 시간 낭비
  : 변경이 없거나, 예상치 못한 변경요청이 이루어지기도 하므로 변경을 고려한 설계는 시간 낭비
  : 변경은 필연적으로 발생 + 점증적 개발로 인해 소프트웨어 구조가 망가지는 경향 존재
  => 지속적인 리팩토링으로 소프트웨어 구조와 가독성 개선 -> 구조적 악화를 막고 변경 처리를 쉽게 한다.

- 리팩토링의 예
  1. 중복 코드를 제거하기 위한 클래스 계층 구조 변경
  2. 속성과 메소드 이름을 적절하게 변경
  3. 비슷한 코드가 사용되는 것을 메소드 호출로 대체

*** 테스트 우선 개발** (p. 72-75)

- 점증적 개발과 계획 주도 개발의 중요한 차이점 -> 시스템을 테스트 하는 방식
  : 점증적 개발에서는 시스템 테스트를 위해 외부 테스팅팀이 사용할 수 있는 시스템 명세가 없다는 문제 존재

- XP 의 테스트
  : 테스팅을 자동화하고 개발 프로세스의 중심에 둠으로써
  모든 테스트를 성공적으로 테스트하기 전에는 개발을 진행할 수 없도록 하였다.

- XP 테스팅 특징
  1. 테스트 우선 개발
  2. 시나리오를 이용하여 점증적 테스트 개발
  3. 테스트 개발과 검증에서의 사용자 참여
  4. 테스트 자동화 프레임워크 사용
  5. 스토리 카드 -> 태스크 -> 태스크 별 테스트 작성
  6. 고객의 역할은 다음 릴리스에서 구현할 스토리에 대한 인수 테스트를 개발하는 것을 돕는 것이다.

- 테스트 주도 개발 (Test-driven Development) (TDD)
  : 코드를 작성하기 전에 테스트를 먼저 작성하는 것

- 테스트 주도 개발 특징
  : 테스트를 작성하려면 기능과 인터페이스에 대해 알아야 하므로 요구사항을 명확히 할 수 있다.
  : 테스트 지연 문제를 방지한다.
  : 테스트 자동화는 테스트 주도 개발에 필수적이다.
  : 테스트를 쉽게 작성할 수 있게 한다.
  : 테스팅을 자동으로 실행할 수 있으며, 테스팅 결과가 명세에 맞는지 확인 가능하다.

* 짝 프로그래밍 (p.75-76)

- 짝 프로그래밍이란?
  : 프로그래머가 소프트웨어를 짝으로 개발하는 것이다. (짝은 바뀔 수 있따.)

- 짝 프로그래밍의 장점
  : 시스템에 대한 공동 소유권과 책임을 지원
  : 다른 사람이 같이 코드를 보기 때문에 비공식적인 리뷰 프로세스가 진행된다.
  : 리팩토링을 장려하고 지지한다.

- 짝 프로그램의 실무 적용
  : 경험 많은 개발자 + 경험 부족한 개발자
  : 지식의 공유가 중요

*** 애자일 프로젝트 관리** (p.76)

- 개발 기법에 따른 프로젝트 관리

    1. 계획 주도 접근법
    : 무엇을 언제 인도할 것인지, 누가 프로젝트 산출물을 만들 것인지에 대한 계획을 가지고 있다.
    : 관리가 쉽다는 장점이 존재

    2. 애자일 기법
    : 프로젝트 진척도가 비가시적이다.
    : 비공식적 계획과 프로젝트 통제는 가시화에 대한 비즈니스 요구사항과 충돌했다.
    : 큰 조직에는 적합하지 않았다.

- 스크럼 (Scrum)
    : 애자일 프로젝트를 조직화하고 가시성을 제공하기 위한 프레임워크

- 스크럼 특징
    1. 반복적인 개발을 관리하는데 중점을 둔 방법
    2. 특정 애자일 실무 원칙을 요구하지 않는다.

- 스크럼 용어 (p.77)
    1. 스크럼 마스터 : 프로젝트 관리자
    2. 제품 백로그 : TO do List
    3. 스크럼 : 짧은 일일 개발 회의
    4. 스프린트 : 2~4 주 길이의 개발 반복 주기

- 스크럼 스프린트 사이클 (Scrum Spring Cycle)
    1. 2~4 주 범위의 고정 길이
    : 완성하지 못한 작업은 제품 백로그로 돌려 놓는다. (기간 연장 X)

    2. 제품 백로그 항목들의 우선 순위를 매겨서 해당 스프린트에 작업할 스프린트 백로그 선정

    3. 매일 스크럼을 진행
    : 진척도를 점검하고 우선순위를 조정하는 짧은 미팅으로, 진척 사항+문제점+계획등 정보 공유

    4. 스크럼 보드를 두어 정보 공유
    : 스프린트 백로그, 진행상황, 완료된 작업 등 표시
    : 누구든지 변경 가능

    5. 스프린트가 끝날 때 리뷰 미팅 실시

<4장 요구공학>

* 요구 사항 (Requirement) (p.96-98)

- 요구 사항 (Requirements)
  : 시스템이 제공해야 하는 서비스(Services) 와 (서비스,개발,운영 등)에 대한 제약조건(Constraints)

- 요구 사항 유형
  1. 사용자 요구사항
  : 시스템이 사용자에게 제공해야 할 서비스와 동작하면서 준수해야 할 제약사항들에 대해
  자연어와 다이어그램으로 기록한 문장
  => 고객 관리자, 시스템 최종 사용자, 고객 엔지니어, 계약 관리자, 시스템 아키텍트

  2. 시스템 요구사항
  : 소프트웨어 시스템의 기능, 서비스와 동작 중 제약사항에 대한 보다 상세한 설명
  : 무엇을 구현해야 할지에 대해 정확하게 정의
  : 구현해야 할 시스템의 서비스와 기능에 대해 보다 구체적인 정보 제공
  => 시스템 최종 사용자, 고객 엔지니어, 시스템 아키텍트, 소프트웨어 개발자

Tip)

	기능적 요구사항	비기능적 요구사항	도메인 요구사항
사용자 요구 사항			
시스템 요구 사항			

- 시스템 이해당사자
  : 어떠한 방식으로든 시스템의 영향을 받는 사람에 해당
  : 시스템에 어느정도의 관심을 가지고 있는 누구라도 시스템 이해당사자에 해당

- 요구 공학 (Requirements Engineering)
  : 요구사항을 찾고, 분석하고, 문서화하고, 점검하는 프로세스

- 타당성 조사 (Feasibility Studies)
  : 요구 공학 프로세스 초기에 이루어져야 하는 짧은 기간의 집중적 조사
  1. 시스템이 조직의 전체 목적에 기여하는가?
  2. 시스템이 현재의 기술을 이용해 일정과 예산 내로 구현이 가능한가?
  3. 시스템을 사용하는 다른 시스템과 통합 가능한가?
  => 3가지를 모두 충족해야 한다.

* 기능적 / 비기능적 요구사항 (p.99)

- 요구 사항 분류
    1. 기능적 요구사항 -> Service
    : 시스템이 제공해야 하는 서비스와 특정 입력에 대해 시스템이 반응하는 방식,
    특정 상황에 시스템이 동작해야 하는 방식을 기술한 것이다.

    2. 비기능적 요구사항 -> Constraints
    : 시스템이 제공하는 서비스나 기능에 대한 제약사항
    : 개별적인 시스템 특징이나 서비스보다는 전체 시스템에 적용되는 경우가 많다.

    3. 도메인 요구사항
    : 시스템 사용자의 특정 요구사항이 아닌 시스템의 응용 도메인에서 얻는다.
    : 소프트웨어 엔지니어가 시스템이 동작하는 도메인의 특성을 이해하지 못한다는 단점이 존재한다.

* 기능적 요구사항 (p.99-101)

- 기능적 요구사항 (Functional Requirements)
    : 시스템에 대한 기능적 요구사항은 시스템이 무엇을 해야 하는지를 설명
    : 시스템이 무엇을 해야 하는지에 초점
    : 시스템의 기능이나 서비스를 기술한다.

- 기능적 요구사항의 종류:
    1. 기능적 사용자 요구사항
    : 시스템이 무엇을 해야 하는지 고수준으로 기술

    2. 기능적 시스템 요구사항
    : 시스템의 서비스를 자세하게 기술

- 요구사항 명세의 부정확성 (Requirements Imprecision) (p.101)
    : 모호한 요구사항은 사용자와 개발자가 서로 다르게 해석할 수 있어 분쟁을 일으킬 수 있다.

- 이상적인 기능적 요구사항 명세
    1. 요구사항의 완전성
    : 필요로 하는 모든 서비스와 정보가 정의되어야 한다.

    2. 요구사항의 일관성
    : 충돌하거나 모순되는 요구사항이 없어야 한다.

    => 실수 혹은 누락, 이해당사자간 충돌로 인해 완전하고 일관적인 현실에서 요구사항을 작성하는 것은 어렵다

*** 비기능적 요구사항** (p.101-105)

- 비기능적 요구사항 (Non-Functional Requirements)
  : 시스템이 사용자에게 제공하는 특정 서비스와 직접적으로 관련되지 않은 요구사항
  : 전체 시스템에 대한 명세나 제약조건
  : 신뢰성, 응답시간, 메모리 사용량 등 창발적(Emergent) 시스템 속성과 관련된 제약
  : 입출력 장치 성능, 다른 시스템과 인터페이스에 사용되는 데이터 표현등 시스템 구현과 관련된 제약

- 비기능적 요구사항 특징
  : 비기능적 요구사항이 만족되지 않을 경우 기능적 요구사항의 경우보다 심각해질 수 있다.
  : 비기능적 요구사항은 특정 컴포넌트보다 시스템의 전체 아키텍처에 영향을 받는다.

- 비기능적 요구사항 분류
  1. 제품 요구사항 (Product Requirements)
  : 소프트웨어의 실행시간 동작에 대한 제약조건
  : 성능, 신뢰성, 보안, 사용성 등

  2. 조직 요구사항 (Organizational Requirements)
  : 고객과 개발자 조직의 정책이나 절차로부터 오는 요구사항
  : 운영 프로세스, 개발 프로세스, 개발환경, 프로세스 표준, 운영환경 등

  3. 외부 요구사항 (External Requirements)
  : 시스템과 개발 프로세스 외부로부터 오는 광범위한 요구사항
  : 규제 법률 윤리 등

- 비기능적 요구사항을 명세하기 위한 척도

속성	척도
속도 Speed	초당 처리 트랜잭션 수 / 사용자,사건 응답시간 / 스크린 리프레시 시간
크기 Size	메가바이트 / ROM 칩 개수
사용 편리성 Ease of use	교육 시간 / 도움말 프레임 수
신뢰성 Reliability	평균 고장 시간 / 고장 발생 비율 / 가용성
견고성 Robustness	고장 후 재가동 시간 / 고장을 원인 사건의 백분율 / 고장에 의한 데이터 손실 확률
이식성 Portability	타겟 시스템에 종속된 코드 비율 / 타겟 시스템의 수

*** 요구 공학 프로세스** (p.105-106)

- 요구공학 프로세스 활동
    1. 요구사항 도출 (Elicitation)
    2. 요구사항 분석 (Analysis)
    3. 요구사항 명세화 (Specification)
    4. 요구사항 검증 (Validation)

    => 실무에서는 요구공학 활동들이 중첩되어 진행된다.

*** 요구사항 도출** (p.107)

- 요구사항 도출 프로세스
    : 이해당사자들의 업무를 이해하고 시스템을 업무에 활용하는 방식을 알아낸다.
    : 이해당사자들로부터 응용 도메인, 시스템이 제공해야 하는 서비스, 요구되는 시스템 성능, 하드웨어 제약 등을 알아내야 한다.

- 요구사항 도출 프로세스 활동
    1. 요구사항 발견 및 이해 (Discovery and Understanding)
    2. 요구사항 분류 및 구성 (Classification and Organization)
    3. 요구사항 우선순위 지정 및 협상 (Prioritization and Negotiation)
    4. 요구사항 문서화 (Documentation)

- 관점 (Viewpoints)
    : 공통점을 가진 이해당사자들로부터 요구사항을 모으고 구성하는 방식이다.
    : 요구사항 분석을 위해 이해당사자 정보를 조직화
    : 이해당사자 그룹이 하나의 관점을 가졌다고 간주하고 해당 그룹의 관점으로부터 요구사항 수집
    : 도메인 요구사항이나 타 시스템 관련 제약조건을 나타내는 관점을 포함시킬 수 있다.
    : 상이한 이해당사자들은 요구사항의 중요도와 우선순위가 다르므로 정기적인 미팅이 중요하다.

- 요구사항 도출을 위한 주요 기법 (p.109)
    1. 인터뷰 (Interview)
    : 사람들이 무엇을 하는지에 대해 이야기를 나눈다.

    2. 관찰 또는 문화기술적 연구 (Observation or Ethnography)
    : 일을 하는 사람들의 모습을 지켜보고 사람들이 무엇을 사용하는지, 어떻게 사용하는지 등을 살핀다.

- 인터뷰 유형
    1. 미리 정의된 질문 목록이 있는 폐쇠적 인터뷰 (Closed)
    2. 개방적 인터뷰 (Open)

- 인터뷰의 문제점
    1. 도메인 전문 용어는 비전문가가 이해하기 어렵다.
    2. 전문가는 자신의 지식이 상식이라 간주하여 어떤 도메인 지식을 언급하지 않는 경우도 존재한다.

- 문화 기술적 연구
    : 운영 프로세스를 이해하고 이와 같은 프로세스를 지원하는 소프트웨어의 요구사항을 얻기 위해 사용하는 관찰기법

- 문화기술적 연구가 효과적인 경우
    : 실제 일하는 방식으로부터 얻을 수 있는 요구사항
    : 협력과 다른 사람의 활동을 인식하는 것으로부터 얻을 수 있는 요구사항

- 문화기술적 연구의 특징과 예시
    : 문화 기술적 연구는 프로토타입 개발과 결합 가능하다
    1. Nokia -> 문화 기술적 연구를 통해 새 제품의 요구사항 도출
    2. Apple -> 기존 사용법을 무시

Tip)
ISP (Information Strategy Planning)
    : 조직의 경영계획과 목표를 지원하기 위한 정보 시스템의 비전을 수립하는 정보전략 계획

BPR (Business Process Reengineering)
    : 조직의 작업을 개선하고 자원을 효율적으로 활용하기 위해 처음부터의 근본적인 변화를 도출하는 컨설팅 프로젝트

* 스토리와 시나리오 (p.113-115)

- 스토리와 시나리오
  : 특정 작업을 위해 어떻게 시스템을 사용하는지 기술
  : 작업 시나리오를 거치면서 무엇을 하는지, 사용하는 정보는 무엇인지, 어떤 시스템을 이용하는지에 대한 설명을 만든다.

- 시나리오
  : 사용자 상호작용이 이루어지는 일정 구간에 대한 사례를 구조적으로 설명한다.

  => 스토리와 시나리오는 본질적으로 같은 것으로,
  특정 작업을 위해 어떻게 시스템을 사용하는가에 대한 설명에 해당한다.

- 시나리오의 구성
  1. 시나리오가 시작할 때 시스템과 사용자가 기대하는 것에 대한 설명
  2. 시나리오 상의 사건들의 정상적인 흐름에 대한 설명
  3. 잘못될 경우와 그에 대한 대처 방안에 대한 정보
  4. 동시에 진행할 수 있는 다른 활동에 대한 정보
  5. 시나리오가 종료했을 때 시스템 상태에 대한 설명

*** 요구사항 명세** (p.116)

- 요구사항 명세란?
  : 사용자 요구사항 및 시스템 요구사항을 요구사항 문서로 작성하는 과정

- 이상적인 요구사항 명세
  1. 시스템의 외부행동과 운영상의 제약조건을 기술해야 한다.
  2. 시스템 설계와 구현에 대한 사항을 다루지 않아야 한다.

- 현실적인 요구사항 명세
  : 요구사항에서 설계정보를 완전히 제외하는 것은 불가능하다.
  : 초기 아키텍처 설계가 요구사항을 구성하는데 도움이 된다.
  : 대부분의 경우 시스템은 기존 시스템과 상호작용이 필요하다.
  : 비기능적 요구사항을 만족시키기 위해 특정 아키텍처를 하용해야 하는 경우도 존재한다.

- 자연어 명세 (Natural Language Specification)
  : 시스템 소프트웨어 요구사항을 명세하기 위해 가장 많이 사용되는 수단
  : 표현력이 풍부하고 직관적이며 보편적이라는 장점과 애매모호하다는 단점이 존재한다.

- 자연어 명세 작성 지침
  1. 표준 형식을 만들어 사용
  2. 필수적인 사항과 바람직한 사항을 구분하기 위해 일관성 있는 표현을 사용
  3. 중요 부분의 글자를 강조
  4. 요구사항의 독자가 기술적이고 소프트웨어 공학 용어를 이해한다고 생각하지 말 것
  5. 가능하다면 사용자 요구사항에 대한 이유를 기록하여 변경 시 활용한다.
  : 왜 포함됐는가 / 누가 제안했는가 등

- 구조적 명세 (Structured Specifications)
  : 구조적 자연어는 요구사항을 자유롭게 작성하지 않고 표준화된 방식으로 작성하는
  시스템 요구사항의 한 방법이다.
  : 자연어의 대부분의 장점을 살리면서도 어느정도의 통일성을 부여할 수 있다.
  : 변동성을 줄이고, 요구사항을 더 효과적으로 구성 가능

- 유스케이스 모델 (p.120)
    : 그래픽 모델과 구조적인 글을 이용하여 사용자와 시스템 사이의 상호작용을 설명하기 위한 방법
    : 시스템의 행동을 모델링

- 유스케이스 구성 요소
    1. 유스케이스 : 시스템이 수행하는 기능
    2. 액터 : 시스템과 직접 상호작용 하는 모든 것
    3. 연관 : 유스케이스와 액터 간의 관계

- 유스케이스 간의 관계
    1. 일반화 (Generalization)
    2. 포함 , 확장 ()Include, Extend)

- 소프트웨어 요구사항 문서 (p.122)
    : 시스템 개발자가 구현해야 하는 것에 대한 공식적인 내용
    : 시스템에 대한 사용자 요구사항과 시스템 요구사항에 대한 상세 명세를 포함할 수 있다.

    : 애자일 기법에선 공식 문서를 사용하지 않고 사용자 요수사항을 점진적으로 수집해서 카드에 작성
      + 짧은 사용자 스토리로 칠판에 기록하는 방식을 이용한다.

*** 요구사항 검증** (p.125)

- 요구사항 검증 (Requirements Validation)
    : 요구사항이 고객이 정말 원하는 시스템을 제대로 정의하고 있는지를 점검하는 과정이다.
    : 요구사항 문제를 수정하는 비용은 설계나 구현 오류를 수정하는 경우에 비해 매우 크다.

- 요구사항 점검 방식 (p.126)
    1. 유효성 점검 : 사용자의 실제 요구를 반영하는지 점검
    2. 일관성 점검 : 문서 상의 요구사항은 서로 상충되지 않아야 한다.
    3. 완전성 점검 : 시스템 사용자가 의도한 모든 기능과 제약을 정의하는 요구사항을 포함해야 한다.
    4. 실현성 점검 : 기존 지식과 기술을 사용하여 주어진 예산으로 구현이 가능한지 점검
    5. 검증 가능성 : 시스템이 각 요구사항을 만족시키는지 보여주는 테스트를 작성 가능한지 점검

- 요구사항 검증 기법
    1. 요구사항 검토 (Review)
    2. 프로토타이핑 (Prototyping)
    3. 테스트 케이스 생성 (Test case)

*** 요구사항 변경** (p.127)

- 요구사항이 변하는 주요 원인
  : 시스템의 비즈니스와 기술적 환경은 계속 변화한다.
  : 개발 비용을 지불하는 사람이 시스템의 사용자가 아니다.
  : 대규모 시스템은 서로 다른 요구사항과 우선순위를 가진 다양한 집단이 관여하기 때문이다.

- 공식적인 요구사항 관리 프로세스가 필요하다.
  : 요구사항 변경으로 인한 영향을 평가
  : 개별 요구사항을 추적하고 연관된 요구사항들 간 관계를 정리
  : 애자일 프로세스에서는 개발 과정에서 변경되는 요구사항을 처리하므로 공식적인 변경 프로세스가 없다.

- 요구사항 관리 계획
  : 어떻게 개선되어가는 요구사항을 관리할 것인지에 관한 것

- 요구사항 변경 관리 (p.130)
  1. 문제 분석 및 변경 명세
  2. 변경 분석 및 비용 산출
  3. 변경 구현

- 추적 가능성 (Tracebility) (p.131)
  : 요구사항 출처, 요구사항, 시스템 설계 간의 관계를 추적할 수 있어야 한다.
  : 변경의 이유와 출처를 관리한다.
  : 제안된 변경이 시스템의 어떤 부분에 영향을 주는지 분석한다.

<5장 시스템 모델링>

*** 시스템 모델링** (p.136)

- 시스템 모델링
  : 시스템의 추상 모델을 개발하는 프로세스
  : 각각의 모델은 시스템의 서로 다른 뷰나 관점을 나타낸다.
  : 보통 UML (Unified Modeling Language) 다이어그램 표기법을 이용하여 나타낸다.

- 모델의 용도
  1. 요구공학 프로세스
  : 시스템의 상세 요구사항을 이끌어 내기 위하여

  2. 설계 프로세스
  : 시스템을 구현할 엔지니어에게 시스템을 설명하기 위하여

  3. 구현 후
  : 시스템의 구조와 동작을 문서화하기 위해

- 시스템 모델은 완전한 표현인가?
  : 시스템 모델은 시스템의 완전한 표현이 아니다.
  : 모델은 시스템의 다른 표현이라기 보다는 검토 중인 시스템의 추상화이다.

- 시스템 관점
  : 서로 다른 관점에서 시스템을 표현하기 위해 다른 모델들을 개발

- 시스템 관점 유형
  1. 외부 관점 (External)
  : 시스템의 컨텍스트나 환경을 모델링

  2. 상호작용 관점 (Interaction)
  : 시스템과 그 환경 사이의 상호 작용을 모델링

  3. 구조 관점 (Structural)
  : 시스템의 구성이나 시스템에 의해 처리되는 데이터의 구조를 모델링

  4. 동작 관점 (Behavioral)
  : 시스템의 동적인 행동을 모델링

* 그래픽 모델 (p.137)

- 그래픽 모델을 사용하는 방법

1. 시스템에 대해 토론하고 아이디어를 도출하는 용도
 : 모델은 불완전 할 수 있으며 표기법을 약식으로 사용 가능하다
 : 주로 애자일 기법에서 사용하는 방법

2. 기존 시스템을 문서화하는 용도
 : 모델이 완전할 필요는 없으나 모델이 정확해야 한다.

3. 시스템 구현을 생성하는 모델 기반 프로세스의 입력
 : 소스코드를 생성하는데 사용되기에 모델이 완전하고 정확해야 한다.

* UML (p.137)

- UML (Unified Modeling Language)
  : 소프트웨어 시스템을 모델링하는 데 사용될 수 있는 13가지 다이어그램 유형들의 집합
  : 객체 지향 모델링의 표준 언어

- 시스템 모델 종류
  1. 컨텍스트 모델 (Context)
  2. 상호작용 모델 (Interaction)
  3. 구조 모델 (Structural)
  4. 동작 모델 (Behavioral)

- UML 다이어그램 대표적 유형 5가지
  1. 액티비티 다이어그램 (Activity) (= 순서도)
  : 프로세스나 데이터 처리와 관련된 액티비티 흐름을 보여준다.

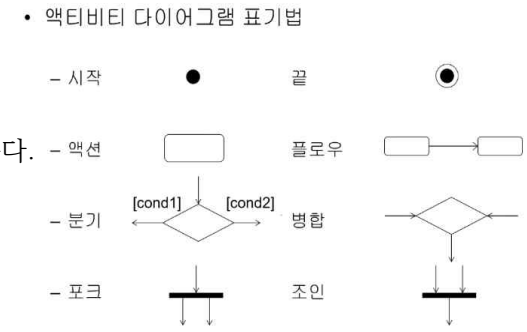
  2. 유스케이스 다이어그램 (Use case)
  : 시스템과 그 환경 간의 상호 작용을 보여준다.

  3. 시퀀스 다이어그램 (Sequence)
  : 액터와 시스템 간, 시스템 컴포넌트들 간의 상호 작용을 보여준다.

  4. 클래스 다이어그램 (Class)
  : 시스템의 객체 클래스들과 클래스들 간의 연관을 보여준다.

  5. 상태 다이어그램 (State)
  : 시스템이 내부 또는 외부 이벤트에 대해 어떻게 반응하는지 보여준다.

* 컨텍스트 모델 (p.138)



- 컨텍스트 모델
  : 시스템의 운영 환경을 기술
  : 시스템의 경계와 시스템과 연동되는 외부 시스템을 보여준다.

- 비즈니스 프로세스 모델
  : 전반적인 업무의 흐름을 보여준다.
  : 컨텍스트 모델과 함께 사용될 수 있다.
  : UML 액티비티 다이어그램 (=순서도) 으로 표현 가능하다.

*** 상호 작용 모델** (p.141)

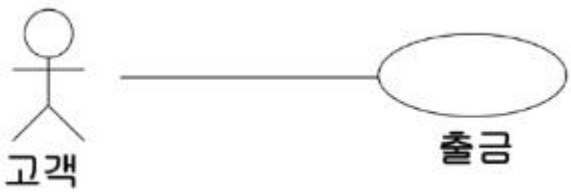
상호 작용 모델의 용도

1. 사용자와 시스템 간 상호작용 모델은 사용자 요구사항을 확인하는데 도움
2. 시스템 간 상호작용 모델은 시스템 간 커뮤니케이션 문제 발생 가능성을 해결한다.
3. 컴포넌트 간 상호작용 모델은 시스템의 구조가 성능과 확실성을 제공할 수 있는지 이해하는데 도움 준다.

- 상호 작용 모델 종류 (두가지 접근법)
  1. 유스케이스 다이어그램
     : 시스템과 외부 에이전트 (사람 사용자 or 다른 시스템들) 와의 상호 작용을 모델링하는 데 주로 사용된다.

  2. 시퀀스 다이어그램
     : 시스템 컴포넌트들 간의 상호 작용을 모델링하는 데 사용되지만 외부 에이전트도 상호 작용에 포함 가능

- 유스케이스 다이어그램 (p.142)



- 유스케이스 다이어그램
  : 상호 작용을 단순하게 개략적으로 보여주는 다이어그램

- 유스케이스 (Use case)
  : 사용자가 어떤 인터페이스를 통하여 시스템에 무엇을 기대하는지에 대한 간단한 기술
  : 시스템이 제공하는 기능
  : 공통적인 사용자 목표와 관련된 시나리오의 집합

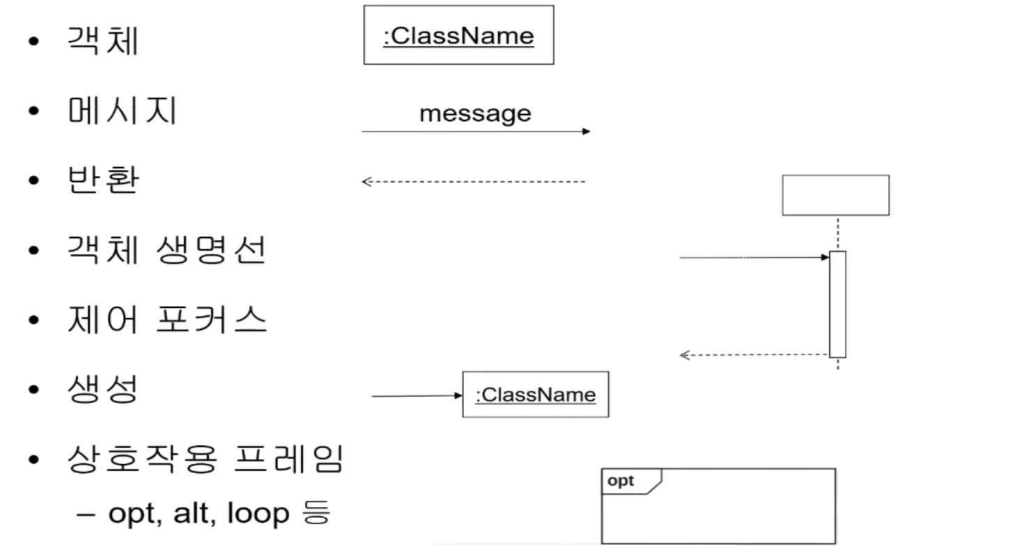
- 액터 (Actor)
  : 시스템이 외부에서 시스템과 직접 상호작용하는 모든 (사용자, 외부 시스템, 장치 등) 것

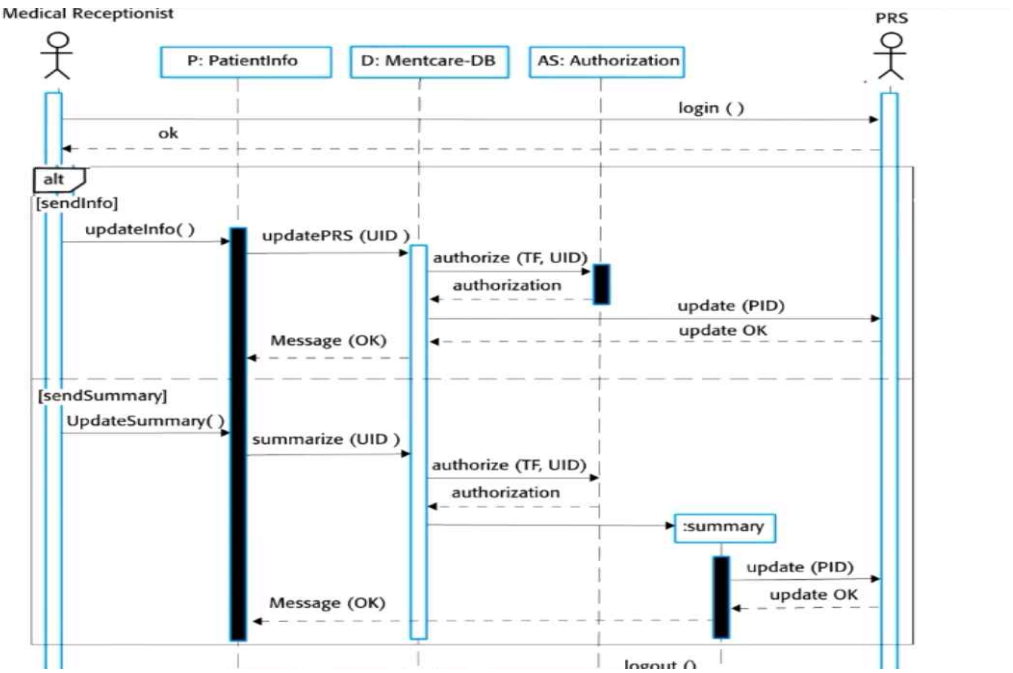
- 연관 (Axxociation)
  : 액터가 유스케이스를 수행

*** 시퀀스 다이어그램** (p.144)

- 시스템 다이어그램
  : 액터와 시스템 객체들간의 상호 작용과 객체들간의 상호작용을 모델링하기 위해 사용
  : 사용자와 시스템 간 상호작용을 나타내는 시퀀스 다이어그램

- 시스템 다이어그램 용도
  : 특정 유스케이스나 유스케이스 인스턴스에서 일어나는 상호 작용의 순서를 보여준다.
  : 코드 생성이나 상세 문서화에 사용하지 않는다면 모든 상호작용을 다이어그램에 포함시킬 필요는 없다.





* 구조 모델 (p.146)

- 구조 모델 (Structural Models)
  : 시스템을 구성하는 컴포넌트들과 그들 간의 관계를 보여준다.

- 구조 모델 유형
  1. 정적 모델 : 시스템 설계구조를 보여준다.
   -> 객체 클래스의 정적 구조를 보여주기 위한 클래스 다이어그램 사용

  2. 동적 모델 : 시스템이 실행 될 때 구성을 보여준다.

- 클래스 다이어그램 (Class Diagram)
  : 시스템의 클래스들과 그들 간의 연관을 보여주는 객체지향 시스템 모델 개발 시 사용된다.

- 클래스
  : 한 가지 종류의 시스템 개체를 일반적으로 정의한 것
  : 같은 속성, 오퍼레이션, 관계, 의미를 가지는 객체 집합에 대한 설명
  : 객체는 클래스의 인스턴스다.

- 연관
  : 클래스들 사이에 어떤 관계가 있다는 것을 표시하는 클래스 사이의 연결
  : 클래스 객체 간의 관계
  : 연결은 연관의 인스턴스다.

Tip) 연관의 속성
  : 이름, 다중성 (Multiplicity), 끝점 이름 (End Name), 도달성 (Navigability)

Tip) 클래스 구성
  1. 클래스 이름

  2. 속성
   : Java 의 데이터 필드

  3. 오퍼레이션
   : Java 의 메서드

  4. 속성과 오퍼레이션 접근 한정자
   4-1. public (+)
   4-2. private (-)
   4-3. protected (#)

- 일반화 & 집합 (p.150-151)

- 일반화 (Generalization)
    : 상속 (Inheritance)
    : Java 의 extends

- 집합 (Aggregation)
    : 한 객체(전체)가 다른 객체(부분)로 구성되는 것
    : 복합(Composition) 관계는 집합 관계의 특수한 경우다.

*** 동작 모델** (p.151-152)

- 동작 모델 (Behavioral Models)
  : 시스템이 실행될 때의 동적 행동에 대한 모델
  : 자극 (Stimulus) 에 대한 시스템의 반응을 모델링

- 자극의 종류
  1. 데이터 : 데이터가 오면 시스템에 의해 처리
  2. 이벤트 : 이벤트가 발생하면 이에 대해 반응

- 상태 다이어그램
  : 내부, 외부 이벤트에 대한 시스템의 반응을 보여준다.

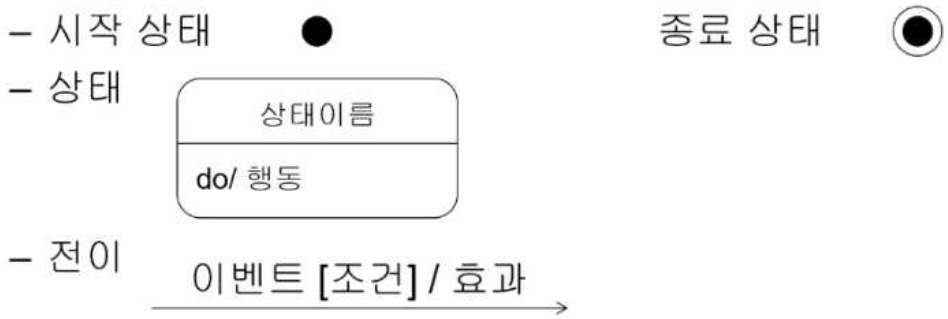
- 데이터 주도 모델링 (Data-driven Modeling)
  : 입력 데이터의 처리와 이와 연관된 출력 생성과 관련된 일련의 행동을 보여준다.
  : 입력 -> 처리 -> 출력

- 데이터 흐름도 (Data Flow Diagram) (DFD) (p.152)
  : 데이터가 처리되는 과정을 보여준다.
  : 액티비티 다이어그램을 이용한다.
  : 자료 흐름 (객체) 와 처리 (액티비티) 로 구성된다.

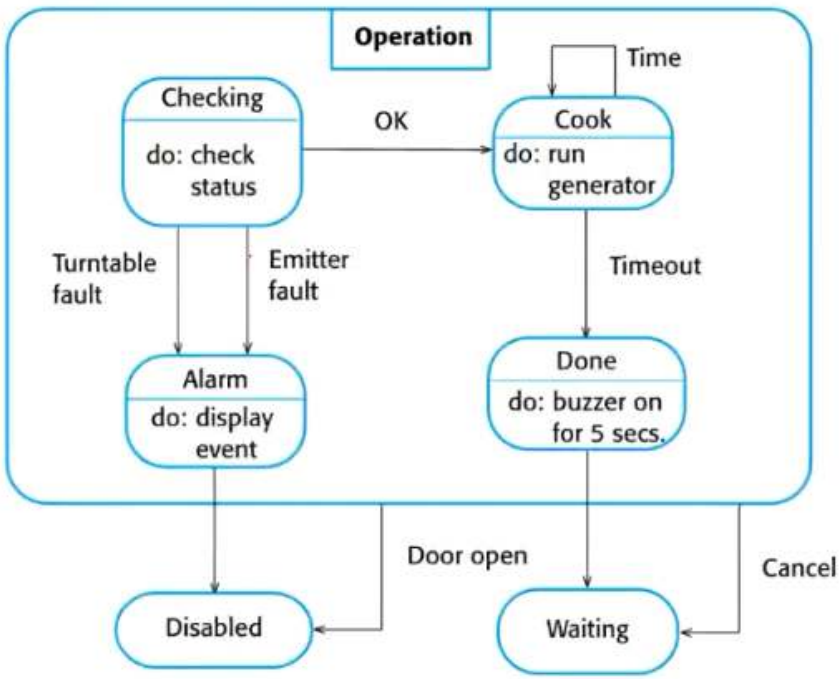
- 이벤트 주도 모델링 (Event-driven Modeling) (p.153-156)
    : 어떻게 시스템이 외부와 내부 이벤트에 반응하는지를 보여준다.
    : 시스템이 외부/내부 이벤트(자극) 에 반응하는 방식

- 이벤트 주도 모델의 구성
    1. 시스템은 유한한 개수의 **상태 (State)** 를 가진다.
    2. 이벤트(자극)가 한 상태에서 다른 상태로 **전이 (Transition)** 된다.

- 상태 다이어그램
    : 시스템의 상태와 한 상태에서 다른 상태로 전이를 일으키는 이벤트를 보여준다.



- 상위 상태와 하위 상태
    1. 상위 상태 -> 여러 개의 서로 다른 상태들을 포함



<6장 아키텍처 설계>

*** 아키텍처 설계** (p.166)

- 아키텍처 설계 (Architectural Design)
  : 시스템 전체 구조 설계
  : 시스템 주요 구조 컴퓨넌트 (Subsystem) 들과 상호작용하는 컴포넌트 간의 관계 (Interfaces)

- 아키텍처 변경
  : 아키텍처의 변경은 비용이 많이 든다.
  : 애자일 프로세스에선 애자일 개발 프로세스 초기 단계가
  전체 시스템 아키텍처 설계에 초점을 맞추어야 한다는 것을 받아들인다.
  : 변화에 대응하여 컴포넌트를 리팩토링 (Refactoring) 하는 것은 비교적 쉽다.
  그러나 시스템 아키텍처를 리팩토링하는 것은
  대부분의 시스템들을 아키텍처 변화에 따라 수정할 수 있기 때문에 비용이 많이 든다.
  => 아키텍처의 점진적인 개발은 바람직 하지 않다.

- 아키텍처 설계와 요구공학 프로세스의 중첩
  : 이상적으로는 요구사항 명세에는 설계가 포함되지 않아야 한다.
  : 그러나 주요 아키텍처 컴포넌트들이 시스템의 상위 수준 특징을 반영하기에
  컴포넌트들을 식별할 수 있어야 한다.
  : 요구 공학 프로세스의 일부로서, 시스템의 기능들과 특징들을 그룹지어
  큰 단위의 컴포넌트나 서브시스템과 연결 짓는 추상 시스템 아키텍처를 제안할 수 있어야 한다.
  => 요구공학 프로세스의 일부로서 추상 시스템 아키텍처가 제시되어야 한다.

- 아키텍처와 기능적 / 비기능적 요구사항
  1. 시스템의 개별 컴포넌트 -> 기능적 요구 사항을 구현
  2. 시스템의 아키텍처 -> 비기능적 시스템 특성에 지배적인 영향을 준다.

- 아키텍처의 명시적 설계와 문서화의 장점
  1. 이해당사자 간의 의사소통
  : 상위 수준의 시스템 표현으로 이해당사자 간의 의사소통에 도움

  2. 시스템 분석
  : 시스템의 중대한 비기능적 요구사항 (성능,신뢰성,유지보수) 등을 만족시킬 수 있는지 분석

  3. 대규모 재사용
  : 비슷한 요구사항을 가진 시스템의 아키텍처를 재사용

Tip) 시스템 아키텍처는 간단한 블록 다이어그램을 이용하여 약식으로 나타낼 수 있다.
: 컴포넌트 (네모상자) + 데이터/제어 신호 (화살표)

*** 아키텍처 설계 과정** (p.169)

- 아키텍처 설계
  : 시스템의 기능적, 비기능적 요구사항을 만족시키기 위한 시스템 구조를 설계하는 창의적인 프로세스
  : 정해진 방법은 없다.
  : 일련의 활동이라기보다 연속적인 결정이라 여기는 것이 좋다.

- 아키텍처 및 시스템 특성
  : 아키텍처 스타일과 구조의 선택은 시스템의 비기능적 요구사항에 좌우된다.

- 비기능적 요구사항 유형

성능	같은 컴퓨터에 배치된 소수의 컴포넌트 안에 중요 작업이 모이도록 한다. 컴포넌트 간 통신을 줄임, 시스템 중복, 부하 분산
보안성	아키텍처에 계층 구조가 사용되야 한다. 중요한 자산을 가장 안쪽 계층에 두는 계층 구조 사용
안전성	안전 관련 작업이 단일 컴포넌트나 소수의 컴포넌트에 같이 배치되도록 설계한다. 안전 관련 작업을 소수의 컴포넌트에 배치하여 안전 검증 및 대응을 간단하게 한다.
가용성	중복 컴포넌트를 배치, 시스템 중단 없이 컴포넌트 교체 및 갱신
유지보수성	변경이 용이한 독립적인 컴포넌트 사용

*** 아키텍처 뷰** (p.171)

- 4+1 뷰 모델 (4+1 View model of software Architecture)

논리적 뷰	객체 또는 객체 클래스로 시스템의 핵심 추상화를 보여준다. 시스템의 논리적 구성을 보여준다.
프로세스 뷰	시스템의 (운영체제)프로세스/쓰레드의 런타임 상호작용을 보여준다. 성능,가용성 등 비기능적 시스템 특성과 관련
개발 뷰	소프트웨어가 개발을 위해 어떻게 분해되는지 보여준다.
물리적 뷰	시스템 하드웨어와 소프트웨어 컴포넌트들의 배치를 보여준다.
유스케이스 뷰	시스템의 행동(유스케이스 시나리오)을 액터 관점에서 보여준다.

=> 시스템을 바라보는 관점을 위의 5가지처럼 볼 수 있다. 꼭 아키텍처 관점으로만 볼 필요가 없다.

* **아키텍처 패턴** (p.173-175)

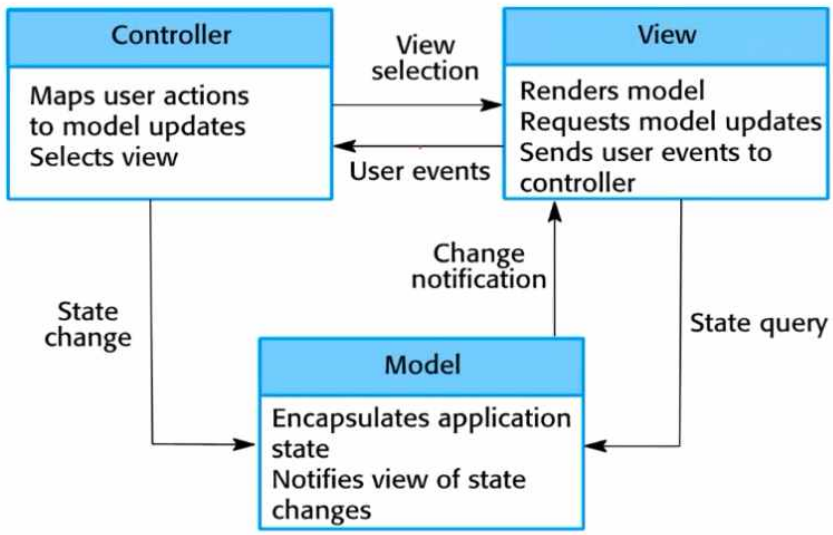
- 패턴
  : 자주 발생하는 문제 (Problem) 에 대한 해법 (Solution)
  : 지식의 공유와 재사용을 목적으로 한다.
  : 이름, 설명, (적용 가능한 경우)문제, 해법 등으로 구성된다.

- 아키텍처 패턴 (= 아키텍처 스타일)
  : 서로 다른 시스템과 환경에서 시도되고 시험된 바람직한 사례를 양식화하고 추상화한 기술
  : 해당 영역에서 성공적이고 바랍직한 시스템 구조 사례를 추상화

- 아키텍처 패턴의 예
  1. 모델 뷰 제어기 (Model View Controller) (MVC)

- MVC 패턴

설명	시스템 데이터로부터 표현과 상호작용을 분리시켜 3개의 논리적 컴포넌트로 구조화 모델 : 데이터와 이에 대한 오퍼레이션을 관리 뷰 : 사용자에게 데이터를 표현하는 것을 관리 제어기 : 사용자와 상호작용을 관리하고 이를 뷰와 모델에 전달
언제 사용되는가	1. 데이터를 보여주고 상호 작용하는 방법이 여러 가지 일 때 사용 2. 데이터 표현과 상호작용에 대한 추후 요구사항을 알 수 없을 때
장점	1. 데이터 표현과 무관하게 데이터를 변경 가능하다. 2. 동일한 데이터를 서로 다른 방법으로 표현하는 것을 지원한다. 3. 뷰의 추가, 변경이 쉽다.
단점	1. 데이터 모델과 상호작용이 단순한 경우 불필요하게 코드가 복잡해질 수 있다.

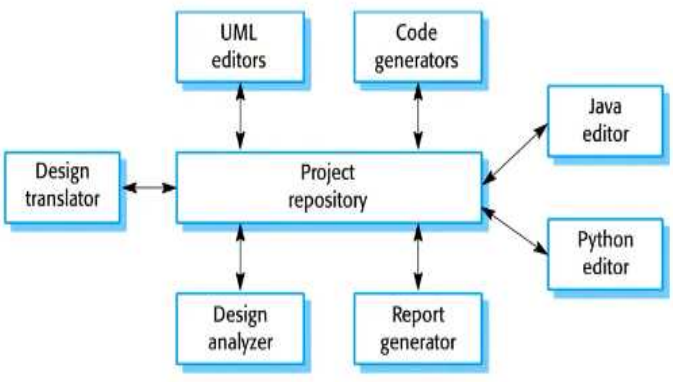


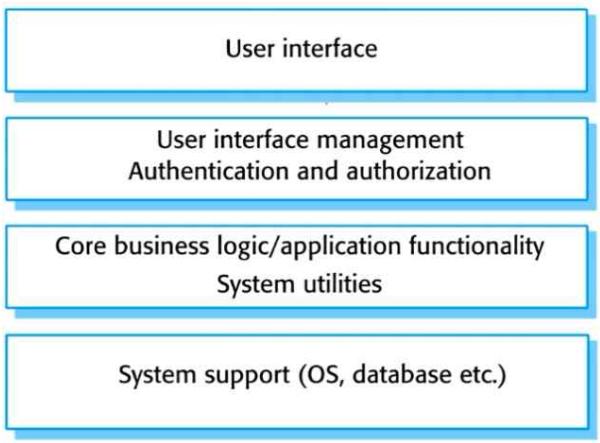
- 계층 아키텍처 (p.176)

설명	각각의 기능을 계층으로 시스템을 구성한다. 각 계층은 상위 계층에 서비스를 제공한다.
언제 사용되는가	1. 새로운 기능을 기존 시스템 상에 구축할 경우 2. 계층별로 나누어 여러 팀이 독립적으로 개발하는 경우 3. 다중 수준 보안이 필요한 경우
장점	1. 인터페이스가 동일하다면 계층을 대체하는 것이 가능 2. 시스템의 확실성을 높이기 위해 중복된 기능을 제공할 수 있다.
단점	1. 계층을 명확하게 구분하는 것이 어렵다. 2. 바로 아래 계층을 통하지 않고 하위 계층을 사용해야 하는 경우도 존재 3. 각 계층의 처리를 거치면서 성능이 저하되는 단점 존재

- 저장소 아키텍처 (p.177)

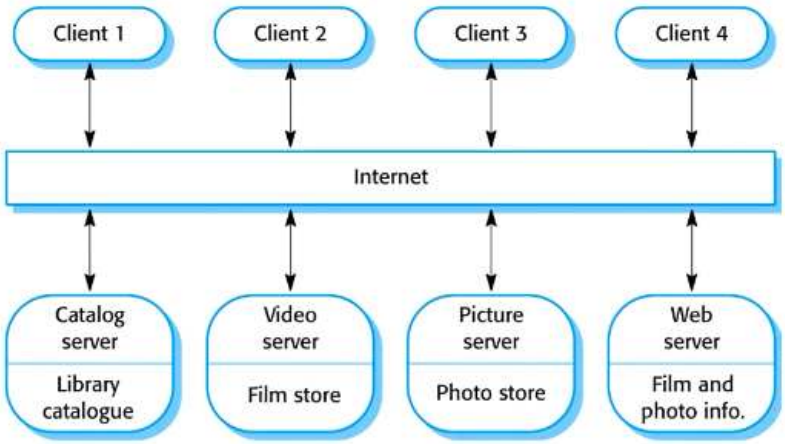
설명	시스템의 모든 데이터는 모든 시스템 컴포넌트들이 접근할 수 있는 중앙 저장소에 관리 대량의 데이터를 사용하는 시스템은 공유 데이터베이스(저장소)를 중심으로 구성 컴포넌트들을 직접 상호작용하지 않고 저장소를 통해서만 상호작용
언제 사용되는가	1. 장기간 저장되어야 하는 대량의 정보를 생성하는 시스템일 경우 2. 저장소에 데이터 추가 시 어떤 행동이나 도구의 동작이 필요한 데이터 주도 시스템일 경우
특징	컴포넌트(도구)들을 저장소를 중심으로 배치 1. 효율적으로 대량의 데이터를 공유 2. 공통적인 저장소 데이터 모델(스키마)를 따라야 한다.
장점	1. 컴포넌들이 독립적이며 다른 컴포넌트들을 알 필요가 없다. 2. 데이터가 일관성 있게 관리된다.
단점	1. 저장소가 단일 장애점 (Single point of Failure) 이기에 저장소에 문제가 생기면 시스템 전체에 영향을 준다. 2. 통신이 저장소에 집중되어 비료율적이다.





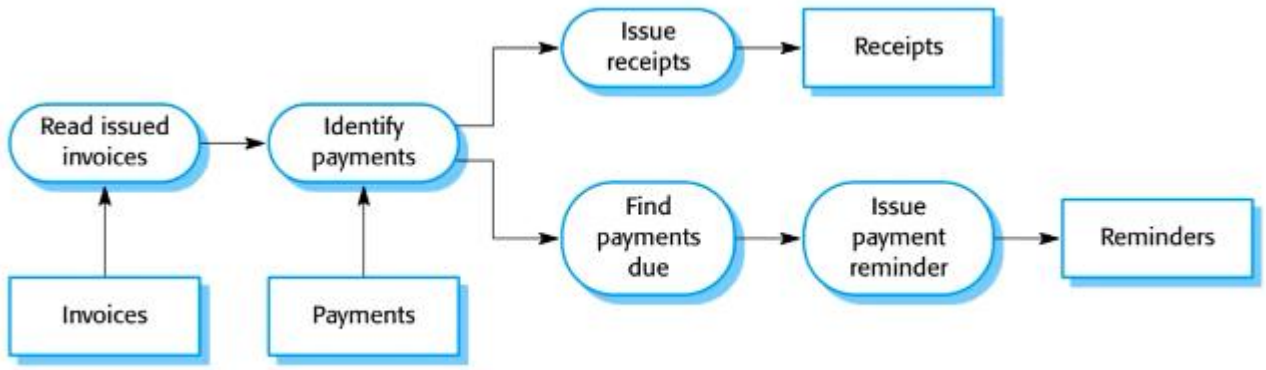
- 클라이언트-서버 아키텍처 (p.179)

설명	분산 네트워크 환경의 주요 아키텍처 시스템이 각 서비스가 독립적인 서버에 의해 제공되는 서비스들의 집합으로 표현한다.
언제 사용되는가	1. 공유 데이터베이스에 있는 데이터를 여러 지역에서 접근해야 할 경우 사용한다. 2. 서버는 중복이 가능하므로 시스템의 부하 변동이 클 경우에도 사용 가능
특징	- 클라이언트 서버 아키텍처를 구성하는 컴포넌트 1. 서비스를 제공하는 서버(프로세스) 의 집합 2. 서비스를 요청하는 클라이언트(프로세스) 의 집합 3. 클라이언트와 서버를 연결하는 네트워크 - 클라이언트와 서버 dussruf 1. 클라이언트는 서버를 찾을 수 있어야 한다. 2. 클라이언트는 HTTP, 원격 프로시저 호출 등의 프로토콜을 이용하여 서버에 서비스를 요청
장점	1. 서버가 네트워크 상에 분산이 가능하다. 2. 분산 아키텍처이므로 서버 추가, 통합, 업그레이드가 쉽다.
단점	1. 각 서비스가 단일 장애점 (Single point of Failure) 이므로 서비스 거부 공격 (DoS)에 취약하다. 2. 네트워크에도 영향을 받기 때문에 성능을 예측하기 어렵다.



- 파이프 필터 아키텍처 (p.181)

설명	시스템에서 데이터 처리는 각 처리 컴포넌트 (Filter) 가 분리, 한 가지 종류의 변환을 수행하도록 구성한다. 입력 데이터를 기능적 변환 (Transformation) 으로 처리하여 출력 데이터를 생성한다. 데이터가 변환을 통해 흘러간다. 변환은 순차적 또는 병렬적으로 실행한다.
언제 사용되는가	사용자 상호 작용이 제한되어 있는 1. 일괄처리 시스템 2. 임베디드 시스템
장점	1. 이해하기 쉽고 변환의 재사용을 지원한다. 2. 워크 플로 유형은 다수의 비즈니스 프로세스 구조와 일치하다. 3. 변환을 추가하여 기능을 변경하는 것이 쉽다. 4. 순차적 혹은 병행 시스템으로 구현 가능하다.
단점	1. 시스템 부담을 증가시키며 호환되지 않는 데이터 구조를 사용하는 아키텍처 컴포넌트는 재사용이 불가능하다. 2. 데이터를 주고 받는 변환 간에 데이터 형식이 일치해야 한다.



- 아키텍처 패턴
    1. MVC
    2. 계층 아키텍처
    3. 저장소 아키텍처
    4. 클라이언트-서버 아키텍처
    5. 파이프 라인 아키텍처

* 애플리케이션 아키텍처 (p.183)

- 애플리케이션 아키텍처
    : 비즈니스 또는 조직의 필요를 만족시키기 위한 것이다.
    : 같은 유형의 시스템을 개발할 때 공통적인 아키텍처 구조가 재사용될 수 있다.

- 트랜잭션
    : 원자성을 가지며 트랜잭션이 수행되면 트랜잭션 내 모든 작업들이 완료되어야 한다.

- 트랜잭션 처리 시스템
    : 사용자 서비스 요청을 비동기적으로 처리하는 대화식 시스템

- 정보 시스템
    : 사용자 인터페이스가 웹 브라우저로 구현된 웹 기반 시스템

- 정보 시스템 아키텍처
    : 주로 다단 클라이언트-서버 아키텍처를 사용한다.

<7장 설계와 구현>

* 설계와 구현 (p.196)

- 소프트웨어 설계와 구현
  : 실행 가능한 소프트웨어 시스템이 개발되는 소프트웨어 공학 프로세스 단계

- 설계와 구현
  1. 설계 -> 요구사항을 실현할 소프트웨어 컴포넌트들과 그들 관계를 식별하는 활동
  2. 구현 -> 설계를 프로그램으로 실체화 시키는 활동
  => 설계와 구현은 필연적으로 중첩된다.
  => 설계와 구현은 밀접하게 연결되어 있으며 설계 시 구현 이슈를 고려해야 한다.

- 개발 프로세스에 따른 차이
  1. 계획 기반 : 설계 단계가 있으며 설계가 모델링되고 문서화된다.
  2. 애자일 : 설계는 대략적인 스케치만 하고 설계 결정은 프로그래머가 한다.

* 객체 지향 설계 (p.197)

- 객체 지향 시스템
  : 객체 지향 시스템은 상호작용하는 객체로 구성된다.
  : 객체들은 자신의 내부 상태를 유지하고 그 상태에 대한 오퍼레이션을 제공한다.

- 객체 지향 설계 프로세스
  : 객체 클래스들과 이 클래스들간의 관계를 설계하는 것과 관련
  : 객체 클래스들을 찾고 클래스 간의 관계를 식별

- 객체 지향 설계 프로세스의 일반적인 절차
  1. 시스템 컨텍스트와 시스템과 외부의 상호작용을 이해하고 정의
  2. 시스템 아키텍처 설계
  3. 시스템 주요 객체 식별
  4. 설계 모델 개발
  5. 객체 인터페이스를 명시

- 시스템 컨텍스트 모델 (p.198)
  : 개발하는 시스템의 환경에 있는 다른 시스템들을 보여주는 **구조** 모델
  : 개발되는 시스템의 환경에 있는 다른 시스템들과의 관계를 보여준다.
  : 클래스 다이어그램과 연관을 이용하여 표현 가능

- 상호 작용 모델
  : 시스템이 사용될 때 어떻게 그 환경과 상호작용하는지 보여주는 **동적** 모델
  : 시스템이 사용될 때 다른 시스템들과의 상호작용을 보여준다.
  : 각 유스케이스의 내용을 기술해야 한다.

Tip) 연관
  : 단순히 개체들간에 어떤 관계가 있다는 것을 보여준다.

- 아키텍처 설계
  : 소프트웨어 시스템과 시스템의 환경 간 상호 작용이 정의되면
    이 정보를 시스템 아키텍처 설게를 위한 기반으로 사용한다.

  : 시스템을 구성하는 주요 컴포넌트(**서브시스템**)들과 그들 간의 **상호작용**을 식별
  : 아키텍처 설계의 일반 지식과 도메인 지식을 활용

- 객체 클래스 식별 지침 (p.202)
    : 관련 문서에서 객체와 속성은 명사이고 오퍼레이션이나 서비스는 동사
    : 응용 도메인의 실제 개체를 나타내는 클래스를 만든다.
    : 시나리오 기반 분석을 사용한다.

- 객체 클래스 식별은 반복적인 과정
    1. 대략적인 시스템 설명으로부터 클래스, 속성, 오퍼레이션을 식별
    2. 응용 도메인 지식과 시나리오 분석을 이용하여 초기 객체들을 정련
    3. 요구사항 문서, 사용자 인터뷰, 기존 시스템 분석 등으로부터 정보 수집

- 설계 모델 (p.204)
    : 설계 모델에서 필요한 상세함의 수준은 사용되는 설계 프로세스에 달려있다.

- 설계 프로세스에 따른 상세 수준
    1. 계획 기반 프로세스
    2. 애자일 프로세스

- 설계 모델의 종류
    1. 구조 모델
    : 시스템의 정적구조를 객체 클래스와 클래스들 간의 관계로 표현
    (문서화 가능 중요 관계) 일반화(상속) 관계, 사용/사용됨의 관계, 복합 관꼐

    2. 동적 모델
    : 실행 중에 일어나는 객체 간의 상호작용을 표현
    (문서화 관계 상호작용) 객체에 의한 서비스 요청 순서, 객체 상호 작용에 의해 유발되는 상태 변화

- 유스케이스 모델과 아키텍처 모델에 세부 사항을 추가하기 위한 **UML 모델 유형 3가지** (p.205)
    1. 서브시스템 모델 (구조 모델)
    : 객체들의 논리적인 그룹들을 일관된 서브시스템으로 보여준다.
    : 객체들을 둘러싸는 패키지로 각 서브시스템을 나타내는 클래스 다이어그램 형태로 표현

    2. 시퀀스 모델 (동적 모델)
    : 객체 상호작용의 순서를 보여준다
    : 시퀀스 다이어그램 혹은 협력 다이어그램으로 표현

    3. 상태 기계 모델 (동적 모델)
    : 객체들이 이벤트에 반응하여 어떻게 그들의 상태를 변경하는지 보여준다.
    : 상태 다이어그램으로 표현

- 인터페이스 (p.208)
    : 객체 또는 객체 그룹에 제공되는 서비스
    : 인터페이스 클래스를 이용해 서비스의 시그니처(Signature)를 정의
     : <<interface>> 스테레오 타입 이용
    : 오퍼레이션을 가지며 데이터는 가지지 않는다.

- 인터페이스 설계
    : 객체의 인터페이스 또는 객체 그룹의 인터페이스의 상세 내역을 명시하는 것과 관련
    : 객체 또는 객체 그룹에 제공되는 서비스들의 시그니처와 의미를 정의하는 것을 의미한다.

*** 오픈 소스 개발** (p.219)

- 오픈 소스 개발
  : 소프트웨어 시스템의 소스 코드가 공개되고,
  개발 프로세스에 지원자들이 참여하도록 초대되는 소프트웨어 개발 접근법

- 오픈 소스 개발 특징
  1. 자유 소프트웨어 재단으로부터 유래되어 누구든지 참여 가능하나, 실제론 핵심 개발 그룹이 주도한다.
  2. 널리 사용되는 오픈 소스 시스템은 안정적이고 버그 수정도 신속하게 해결된다.

- 대표적인 오픈 소스 시스템
  1. Linux
  2. Apache
  3. Eclipse

- **오픈 소스 라이선스** (p.221)
  1. **GPL** (GNU 일반 공증 라이선스)
  : 프로그램 GPL 소스를 (일부라도) 사용/변경한다면
  프로그램의 소스를 공개해야 하고 프로그램도 GPL 라이선스를 따른다.

  2. **LGPL** (GNU 레서 일반 공중 라이선스)
  : 완화된 GPL 로서 LGPL 라이브러리를 사용한다면 프로그램의 소스를 공개할 필요는 없다.
  : LGPL 라이브러리를 수정한 경우 소스를 공개해야 한다.

  3. **BSD** (버클리 표준 배포)
  : BSD 소스를 (변경하여) 사용할 수 있으며 변경 내역과 프로그램 소스 공개 의무 없다.
  : 저작권자의 이름과 라이선스 내용을 같이 배포해야 한다.