8. 서버 프로그램 구현

개발 환경 구축

1. 하드웨어 환경
    - Client와 Server로 구성
    - Server의 종류
        1. 웹 서버 (Web Server)

            : 클라이언트로부터 직접 요청을 받아 처리

            : 저용량의 정적 파일 제공 (HTML, CSS, Image)
        2. 웹 애플리케이션 서버 (WAS)

            : 동적 서비스 제공

            : 웹 서버 - 데이터베이스 서버, 웹 서버 - 파일 서버 사이의 인터페이스 역할 수행
        3. 데이터베이스 서버 (DB Server)

            : 데이터베이스와 관리하는 DBMS 사용
        4. 파일 서버 (File Server)

            : 데이터베이스에 저장하기에는 비효율적이거나, 서비스 제공을 목적으로 유지하는 파일 저장

2. 소프트웨어 환경
    - 시스템 소프트웨어 종류

      : 운영체제 (OS), 웹 서버 및 WAS 운용을 위한 서버 프로그램, DBMS
    - 개발 소프트웨어 종류
        1. 요구사항 관리 도구
        2. 설계/모델링 도구
        3. 구현 도구
        4. 빌드 도구
        5. 테스트 도구
        6. 형상 관리 도구

3. 웹 서버 기능

a. HTTP / HTTPS 지원
b. 통신 기록
c. 정적 파일 관리
d. 대역폭 제한
e. 가상 호스팅
f. 인증

소프트웨어 아키텍처

1. 소프트웨어 아키텍처
   - 소프트웨어를 구성하는 요소들 간의 관계를 표현하는 시스템의 구조 또는 구조체

2. 모듈화 (Modularity)
   - 시스템의 기능들을 모듈 단위로 나누는 것
   - 크기를 작게 나누면 개수가 많아 모듈 간 통합 비용이 많이 듦
   - 크기를 크게 나누면 통합 비용은 적지만 개발 비용이 많이 듦

3. 추상화 (Abstraction)
   - 전체적이고 포괄적인 개념 설계 후 차례로 세분화하여 구체화 시키는 것
   - 유형
     1. 과정 추상화
        : 전반적인 흐름만 파악할 수 있게 설계
     2. 데이터 추상화
        : 데이터 세부 속성이나 용도 정의 없이 구조를 대표하는 표현으로 대체
     3. 제어 추상화
        : 이벤트 발생의 절차나 방법 정의 없이 대표할 수 있는 표현으로 대체
4. 단계적 분해 (Stepwise Refinement)
   - 상위 중요 개념으로부터 하위의 개념으로 구체화시켜 분할

5. 정보 은닉 (Information Hiding)
   - 한 모듈 내부에 포함된 절차와 자료들의 정보가 감추어져 다른 모듈이 접근하거나 변경하지 못하도록 하는 기법
   - 정보 은닉으로 모듈을 독립적으로 수행할 수 있음
   - 하나의 모듈이 변경되어도 다른 모듈에 영향을 주지 않아, 수정, 시험, 유지보수에 용이

6. 상위 설계 & 하위 설계
   a. 상위 설계 (아키텍처 설계, 예비 설계)
      i. 설계 대상 : 시스템 전체 구조
      ii. 세부 목록 : 구조, DB, 인터페이스
   b. 하위 설계 (모듈 설계, 상세 설계)
      i. 설계 대상 : 시스템 내부 구조 및 행위

ii. 세부 목록 : 컴포넌트, 자료 구조, 알고리즘

7. 소프트웨어 아키텍처 설계 과정
    a. 설계 목표 설정
    b. 시스템 타입 결정
    c. 아키텍처 패턴 적용
    d. 서브시스템 구체화
    e. 검토

아키텍처 패턴

1. 아키텍처 패턴 (Patterns)
    - 아키텍처를 설계할 때 참조할 수 있는 전형적인 해결 방식 또는 에제
    - 서브시스템들과 그 역할이 정의되어 있음
    - 서브시스템 사이의 관계와 여러 규칙 등이 포함

2. 레이어 패턴 (Layers Pattern)
    - 시스템을 계층으로 구분해서 구성하는 고전적 패턴
    - 상위 계층은 하위 계층에 대한 서비스 제공자, 하위 계층은 상위 계층의 클라이언트
    - 서로 마주한 두 계층 사이에서만 상호작용이 이루어짐
    - OSI 참조 모델

3. 클라이언트-서버 패턴 (Client-Server Pattern)
    - 하나의 서버 컴포넌트와 다수의 클라이언트 컴포넌트로 구성
    - 사용자가 클라이언트를 통해 서버에 요청하면 응답을 받아와 사용자에 제공

4. 파이프-필터 패턴 (Pipe-Filter Pattern)
    - 데이터 스트림 절차의 각 단계를 필터로 캡슐화 하여 파이프를 통해 전송하는 패턴
    - 앞 시스템의 처리 결과를 파이프를 통해 전달받아 처리 후 다시 파이프를 통해 넘겨주는 패턴 반복
    - Shell

5. MVC 패턴 (Model-View-Controller Pattern)
    - 서브 시스템을 모델 - 뷰 - 컨트롤러로 구조화하는 패턴
    - 컨트롤러가 사용자의 요청을 받으면 핵심 기능과 데이터를 보관하는 모델을 이용하여 뷰에 정보를 출력
    - 여러 개의 뷰 생성 가능

6. 기타 패턴
    a. 마스터-슬레이브 패턴 (Master-Slave Pattern)
    b. 브로커 패턴 (Broker Pattern)
    c. 피어 투 피어 패턴 (Peer-To-Peer Pattern)

d. 이벤트-버스 패턴 (Event-Bus Pattern)
e. 블랙보드 패턴 (Blackboard Pattern)
f. 인터프리터 패턴 (Interpreter Pattern)

객체 지향 (Object-Oriented)

1. 객체 지향
   - 각 요소들을 객체(Object)로 만든 후, 객체들을 조립하여 소프트웨어를 개발하는 기법
   - 소프트웨어 재사용 및 확장이 용이, 고품질의 소프트웨어를 빠르게 개발할 수 있고 유지보수가 쉽다.

   - 구성 요소
     1. 객체 (Object)
        : 데이터와 함수를 묶어 놓은 소프트웨어 모듈
     2. 클래스 (Class)
        : 공통된 속성과 연산을 갖는 객체의 집합
        : 인스턴스 - 클래스에 속한 각각의 객체
     3. 메시지 (Message)
        : 객체들 간의 상호작용에 사용되는 수단, 객체의 동작이나 연산을 일으키는 외부의 요구사항

   - 특징
     1. 캡슐화 (Encapsulation)
        : 외부에서 접근을 제한하기 위해 인터페이스를 제외한 세부 내용을 은닉하는 것
     2. 상속 (Inheritance)
        : 상위 클래스의 모든 속성과 연산을 하위 클래스가 물려받는 것
     3. 다형성 (Polymorphism)
        : 하나의 메시지에 대해 각각의 객체가 고유한 방법으로 응답할 수 있는 능력
     4. 연관성 (Relationship)
        : 두 개 이상의 객체들이 상호 참조하는 관계
        : is member of - 연관화 (Association)
        : is instance of - 분류화 (Classification)
        : is part of - 집단화 (Aggregation)
        : is a - 일반화 (Generalization), 특수화/상세화 (Specialization)

객체지향 분석 및 설계

1. 객체지향 분석 (OOA : Object Oriented Analysis)
   - 사용자의 요구사항과 관련된 객체, 속성, 연산, 관계 등을 정의하여 모델링하는 작업

2. 객체지향 분석 방법론
   a. Rumbaugh (럼바우) 방법
      : 모든 소프트웨어 구성요소를 그래픽 표기법을 이용하여 모델링하는 기법

: 분석활동 순서 - 객체 모델링 → 동적 모델링 → 기능 모델링

b. Booch (부치) 방법

: 클래스와 객체들을 분석 및 식별하고 클래스의 속성과 연산을 정의

c. Jacobson 방법

: Use Case를 강조하여 사용

d. Coad-Yourdon 방법

: E-R 다이어그램 사용하여 객체의 행위를 모델링

e. Wirfs-Brock 방법

: 고객 명세서를 평가해서 설계 작업까지 연속적으로 수행

3. Rumbaugh 방법

a. 객체 모델링

: 시스템에서 요구되는 객체를 찾아내어 속성, 연산을 식별하고 객체들 간의 관계를 규정하여 객체 다이어그램으로 표시

b. 동적 모델링

: 상태 다이어그램 사용하여 시간의 흐름에 따른 객체들간의 동적 행위를 표현

c. 기능 모델링

: 자료 흐름도 (DFD)를 이용하여 다수의 프로세스들 간의 자료 흐름을 중심으로 모델링

4. 객체 지향 설계 원칙 (SOLID)

a. SRP - 단일 책임 원칙

: 객체는 단 하나의 책임만 가져야 함

b. OCP - 개방 폐쇄 원칙

: 기존의 코드를 변경하지 않고 기능을 추가할 수 있어야 함

c. LSP - 리스코프 치원 원칙

: 자식 클래스는 최소한 부모 클래스의 역할을 수행할 수 있어야 함

d. ISP - 인터페이스 분리 원칙

: 사용하지 않는 인터페이스와 의존 관계를 맺거나 영향 받지 않아야 함

e. DIP - 의존 역전 원칙

: 의존 관계 성립 시 추상성 높은 클래스와 의존 관계를 맺어야 함

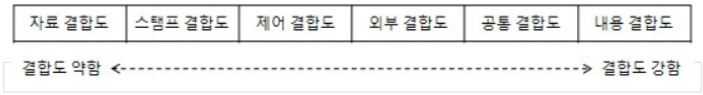
모듈

1. 모듈

- 모듈화를 통해 분리된 시스템의 각 기능
- 결합도와 응집도에 의해 측정됨

2. 결합도 (Coupling)

- 모듈 간에 상호 의존하는 정도, 두 모듈 사이의 연관 관계
- 결합도가 약할수록 품질이 높음



i. 자료 결합도

: 모듈 간의 인터페이스가 자료 요소로만 구성될 때의 결합도

ii. 스탬프 결합도

: 모듈 간의 인터페이스로 배열이나 레코드 등의 자료구조가 전달될 때의 결합도

iii. 제어 결합도

: 어떤 모듈이 다른 모듈 내부의 논리적 흐름을 제어하기 위해 제어 신호를 전달하는 결합도

iv. 외부 결합도

: 어떤 모듈에서 선언한 데이터(변수)를 외부의 다른 모듈에서 참조할 때의 결합도

v. 공통 결합도

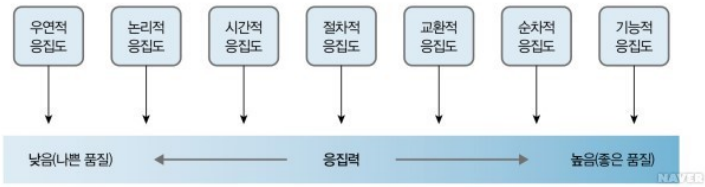
: 공유되는 공통 데이터 영역을 여러 모듈이 사용할 때의 결합도

vi. 내용 결합도

: 한 모듈이 다른 모듈의 내부 기능 및 내부 자료를 직접 참조하거나 수정할 때의 결합도

3. 응집도

- 모듈 내부 요소들이 서로 관련되어있는 정도
- 응집도가 강할수록 품질이 높음



i. 우연적 응집도

: 모듈 내부의 각 구성 요소들이 서로 관련 없는 요소로만 구성된 경우의 응집도

ii. 논리적 응집도

: 유사 성격을 갖거나 특정 형태로 분류되는 처리 요소들로 모듈이 형성되는 경우의 응집도

iii. 시간적 응집도

: 특정 시간에 처리되는 기능을 모아 하나의 모듈로 작성할 경우의 응집도

iv. 절차적 응집도

: 모듈이 다수의 관련 기능을 가질 때 모듈 안의 구성요소들이 그 기능을 순차적으로 수행할 경우의 응집도

v. 교환적 응집도

: 동일 입력과 출력을 사용하여 서로 다른 기능을 수행하는 구성 요소들이 모였을 경우의 응집도

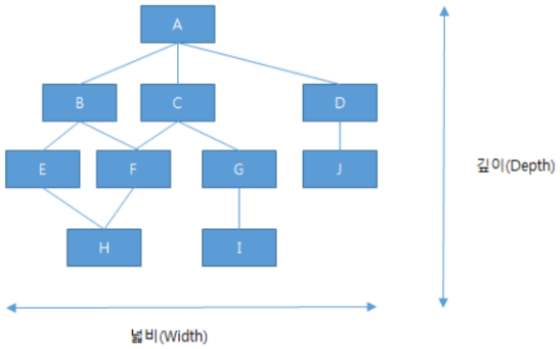
vi. 순차적 응집도

: 모듈 내 하나의 활동으로부터 나온 출력 데이터를 다음 활동의 입력 데이터로 사용할 경우의 응집도

vii. 기능적 응집도

: 모듈 내부의 모든 기능 요소들이 단일 문제와 연관되어 수행될 경우의 응집도

4. Fan-In / Fan-Out
   - Fan-In : 어떤 모듈을 제어하는 모듈의 수
     - 높을 경우 재사용 측면에서 설계가 잘 됨
     - 단일 장애점이 발생할 수 있으므로 중점적인 관리와 테스트 필요
   - Fan-Out : 어떤 모듈에 의해 제어되는 모듈의 수
   - 시스템 복잡도 최적을 위해 Fan-In은 높게, Fan-Out은 낮게 설계해야 함



   i. B의 Fan-In : 1 (A), Fan-Out : 2 (E, F)
   ii. C의 Fan-In : 1 (A), Fan-Out : 2 (F, G)

5. N-S 차트 (Nassi-Schneiderman Chart)
   - 논리의 기술에 중점을 두고 도형을 이용해 표현하는 방법
   - 조건이 복합되어 있는 곳의 처리를 시각적으로 명확히 식별하는 데 적합

단위 모듈

1. 단위 모듈 (Unit Module)
   - 한 가지 동작을 수행하는 기능을 모듈로 구현한 것
   - 독립적 컴파일 가능, 다른 모듈에 호출되거나 삽입되기도 함
   - 구현 과정

     : 단위 기능 명세서 작성 → 입 출력 기능 구현 → 알고리즘 구현

2. IPC (Inter-Process Communication)
   - 모듈 간 통신 방식을 구현하기 위해 사용되는 대표적인 프로그래밍 인터페이스 집합
   - 대표 메소드
     - Shared Memory

       : 공유 가능한 메모리를 구성하여 다수의 프로세스가 통신하는 방식
     - Socket

       : 네트워크 소켓을 이용하여 네트워크를 경유하는 프로세스간 통신하는 방식
     - Semaphores

       : 공유 자원에 대한 접근 제어를 통해 통신하는 방식

- Pipes & Named Pipes

  : 선입선출 형태의 메모리를 여러 프로세스가 공유하여 통신하는 방식

  : 하나의 프로세스가 이용 중이면 다른 프로세스는 접근할 수 없음

- Message Queueing

  : 메시지 발생 시 이를 전달하는 방식으로 통신하는 방식

공통 모듈

1. 공통 모듈
   - 여러 프로그램에서 공통으로 사용할 수 있는 모듈
   - 자주 사용되는 기능들이 공통 모듈로 구성될 수 있음
   - 해당 기능을 명확히 이해하도록 명세 기법을 준수해야 함

2. 공통 모듈 명세 기법 종류
   a. 정확성 (Correctness)

      : 시스템 구현 시 해당 기능이 필요하다는 것을 알 수 있도록 정확히 작성해야 함

   b. 명확성 (Clarity)

      : 해당 기능을 이해할 때 중의적으로 해석되지 않도록 명확하게 작성

   c. 완전성 (Completeness)

      : 시스템 구현을 위해 필요한 모든 것을 기술

   d. 일관성 (Consistency)

      : 공통 기능들 간 상호 충돌이 발생하지 않도록 기술

   e. 추적성 (Traceability)

      : 기능에 대한 요구사항의 출처 등을 파악할 수 있도록 작성

3. 재사용 (Reuse)
   - 이미 개발된 기능들을 새로운 시스템이나 기능 개발에 사용하기 적합하도록 최적화 하는 작업
   - 새로 개발에 필요한 비용과 시간 절약 가능
   - 누구나 이해하고 사용하도록 사용법 제공 필요
   - 규모에 따른 분류
     1. 함수와 객체 - 클래스나 메소드 단위의 소스 코드 재사용
     2. 컴포넌트 - 컴포넌트 자체에 대한 수정 없이 인터페이스를 통해 통신하는 방식으로 재사용
     3. 애플리케이션 - 공통된 기능들을 제공하는 애플리케이션을 공유하는 방식으로 재사용

4. 효율적인 모듈 설계 방안
   a. 결합도는 줄이고 응집도를 높여 모듈의 독립성과 재사용성 높인다
   b. 복잡도와 중복성을 줄여 일관성을 유지
   c. 모듈의 기능은 예측 가능해야 하고 지나치게 제한적이면 안됨
   d. 효과적 제어를 위해 모듈 간 계층적 관계를 정의하는 자료가 제시되어야 함

코드

1. 코드
   - 자료의 분류, 조합, 집계, 추출을 용이하게 하기 위해 사용하는 기호
   - 주요 기능
     1. 식별 기능 - 데이터 간의 성격에 따라 구분 가능
     2. 분류 기능 - 특정 기준이나 유형에 해당하는 데이터 그룹화 가능
     3. 배열 기능 - 의미를 부여하거나 나열 가능
     4. 표준화 기능 - 다양한 데이터를 기준에 맞추어 표현
     5. 간소화 기능 - 복잡한 데이터 간소화

   - 종류
     1. 순차 코드 - 차례로 일련번호 부여
     2. 블록 코드 - 공통성 있는 것 끼리 블록으로 구분하고 일련번호 부여
     3. 10진 코드 - 코드화 대상을 10진 분할
     4. 그룹 분류 코드 - 기준에 따라 그룹으로 구분하고 일련번호 부여
     5. 연상 코드 - 명칭이나 약호와 관계있는 문자, 기호를 통해 코드 부여
     6. 표의 숫자 코드 - 물리적 수치를 그대로 코드에 적용
     7. 합성 코드 - 2개 이상의 코드를 조합

디자인 패턴

1. 디자인 패턴
   - 모듈 간의 관계 및 인터페이스 설계 시 참조할 수 있는 전형적인 해결 방식 또는 예제

2. 생성 패턴 (Creational Pattern)
   - 클래스나 객체의 생성과 참조 과정을 정의하는 패턴

   a. 추상 팩토리 (Abstract Factory)
      : 구체적 클래스에 의존하지 않고 인터페이스를 통해 서로 연관 의존하는 객체들의 그룹으로 생성하여 추상적으로 표현
      : 연관된 서브 클래스를 묶어 한 번에 교체 가능

   b. 빌더 (Builder)
      : 작게 분리된 인스턴스를 건축 하듯이 조합하여 객체 생성

   c. 팩토리 메소드 (Factory Method)
      : 객체 생성을 서브 클래스에서 처리하도록 분리하여 캡슐화 한 패턴
      : 상위 클래스는 인터페이스만 정의하고 실제 생성은 서브 클래스에서 담당

   d. 프로토타입 (Prototype)
      : 원본 객체를 복제하는 방법으로 객체 생성

   e. 싱글톤 (Singleton)
      : 하나의 객체를 생성하면 생성된 객체를 어디서든 참조할 수 있지만 여러 프로세스가 동시 참조 불가
      : 클래스 내에 인스턴스가 하나임을 보장

3. 구조 패턴 (Structural Pattern)
   - 클래스나 객체들을 조합하여 더 큰 구조로 만드는 패턴

   a. 어댑터 (Adapter)

      : 호환성 없는 클래스들의 인터페이스를 다른 클래스가 이용할 수 있도록 변환하는 패턴

   b. 브리지 (Bridge)

      : 구현부에서 추상층을 분리하여 서로가 독립적으로 확장할 수 있도록 구성한 패턴

      : 기능과 구현을 별도 클래스로 구현

   c. 컴포지트 (Composite)

      : 여러 객체를 가진 복합 객체와 단일 객체를 구분 없이 다루는 패턴

      : 객체를 트리 구조로 구성

   d. 데코레이터 (Decorator)

      : 객체 간의 결합을 통해 능동적으로 기능들을 확장할 수 있는 패턴

   e. 퍼사드 (Facade)

      : 복잡한 서브 클래스들을 피해 더 상위에 인터페이스를 구성하여 서브 클래스들의 기능을 간편히 사용할 수 있도록 하는 패턴

   f. 플라이웨이트 (Flyweight)

      : 인스턴스가 필요할 때마다 매번 생성이 아닌 가능한 공유해서 사용하는 패턴

      : 메모리 절약

   g. 프록시 (Proxy)

      : 접근이 어려운 객체와 연결하려는 객체 사이에 인터페이스 역할을 수행하는 패턴

4. 행위 패턴 (Behavioral Pattern)
   - 클래스나 객체들이 서로 상호작용하는 방법이나 책임 분배 방법을 정의하는 패턴

   a. 책임 연쇄 (Chain of Responsibility)

      : 요청을 처리할 수 있는 객체가 둘 이상 존재하여 한 객체가 처리하지 못하면 다음 객체로 넘어가는 형태의 패턴

   b. 커맨드 (Command)

      : 요청을 객체 형태로 캡슐화하여 재이용하거나 취소할 수 있도록 요청에 필요한 정보를 저장하거나 로그로 남기는 패턴

   c. 인터프리터 (Interpreter)

      : 언어에 문법 표현을 정의하는 패턴

   d. 반복자 (Iterator)

      : 접근이 잦은 객체에 대해 동일한 인터페이스를 사용하도록 하는 패턴

   e. 중재자 (Mediator)

      : 수많은 객체들 간의 복잡한 상호작용을 캡슐화하여 객체로 정의하는 패턴

   f. 메멘토 (Memento)

      : 특정 시점에서 객체 내부의 상태를 객체화함으로써 이후 요청에 따라 객체를 해당 시점의 상태로 돌릴 수 있는 기능을 제공하는 패턴

   g. 옵저버 (Observer)

      : 한 객체의 상태가 변화하면 객체에 상속되어 있는 다른 객체들에게 변화된 상태를 전달하는 패턴

      : 분산된 시스템 간에 이벤트 Publish, Subscribe할 때 사용

   h. 상태 (State)

: 객체의 상태에 따라 동일한 동작을 다르게 처리할 때 사용하는 패턴

i. 전략 (Strategy)

: 동일한 계열의 알고리즘을 캡슐화하여 상호 교환할 수 있게 정의하는 패턴

j. 템플릿 메소드 (Template Method)

: 상위 클래스에서 골격을 정의하고 하위 클래스에서 세부 처리를 구체화하는 구조의 패턴

k. 방문자 (Visitor)

: 각 클래스들의 데이터 구조에서 처리 기능을 분리하여 별도의 클래스로 구성하는 패턴

빌드 도구

1. 빌드 도구
   - 소스 코드 파일들을 컴퓨터에서 실행할 수 있는 제품 소프트웨어로 변환하는 과정과 결과물
   - 전처리, 컴파일 등의 작업을 수행
   - 종류
     1. Ant (Another Neat Tool)

        : Apache에서 개발

        : 자바 프로젝트의 공식 빌드 도구

        : 정해진 규칙이나 표준 없음
     2. Maven

        : Apache에서 Ant의 대안으로 개발

        : 의존성 설정하여 라이브러리 관리

        : 규칙이나 표준이 존재해 예외 사항만 기록
     3. Gradle

        : Ant와 Maven을 보완하여 개발

        : Android Studio 공식 빌드 도구

        : Groovy 기반의 빌드 스크립트 사용

API (Application Programming Interface)

1. API
   - 응용 프로그램 개발 시 라이브러리를 이용할 수 있도록 규칙들을 정의해 놓은 인터페이스
   - 효율적 개발 가능
   - Open API - 누구나 무료로 사용

배치 프로그램

1. 배치 프로그램 (Batch Program)
   - 여러 작업들을 미리 정해진 일련의 순서에 따라 일괄 처리하도록 만든 프로그램
   - 필수 요소
     a. 대용량 데이터 - 대량의 데이터 처리 가능해야 함

     b. 자동화 - 심각한 오류 상황을 제외하고 사용자 개입없이 수행되어야 함

c. 견고성 - 데이터 이상으로 인한 중단이 없어야 함
d. 안전성 / 신뢰성 - 오류 발생 시 추적 가능해야 함
e. 성능 - 다른 응용 프로그램 수행을 방해하지 않아야 함

2. 배치 스케쥴러 (Batch Scheduler)
   - 일괄 처리 (Batch Processing) 작업이 설정된 주기에 맞춰 자동으로 수행되도록 지원하는 도구
   - = Job Scheduler
   - 종류
     a. 스프링 배치
     b. Quartz
     c. Cron