Section 2. 소프트웨어 아키텍처

1. 소프트웨어 아키텍처

(1) 소프트웨어 아키텍처(SoftWare Architecture) 개념

- 소프트웨어의 골격이 되는 기본구조
- 시스템의 컴포넌트 사이의 관계를 정의한 구조

(2) 소프트웨어 아키텍처의 특징

특징	설명
간략성	이해하고 추론할 수 있을 정도의 간결성 유지
추상화	시스템의 추상적인 표현을 사용
가시성	시스템이 포함해야 하는 것들을 가시화
관점 모형	이해당사자의 관심사에 따른 모형 제시
의사소통수단	이해당사자 간 원활한 의사소통 수단으로 이용

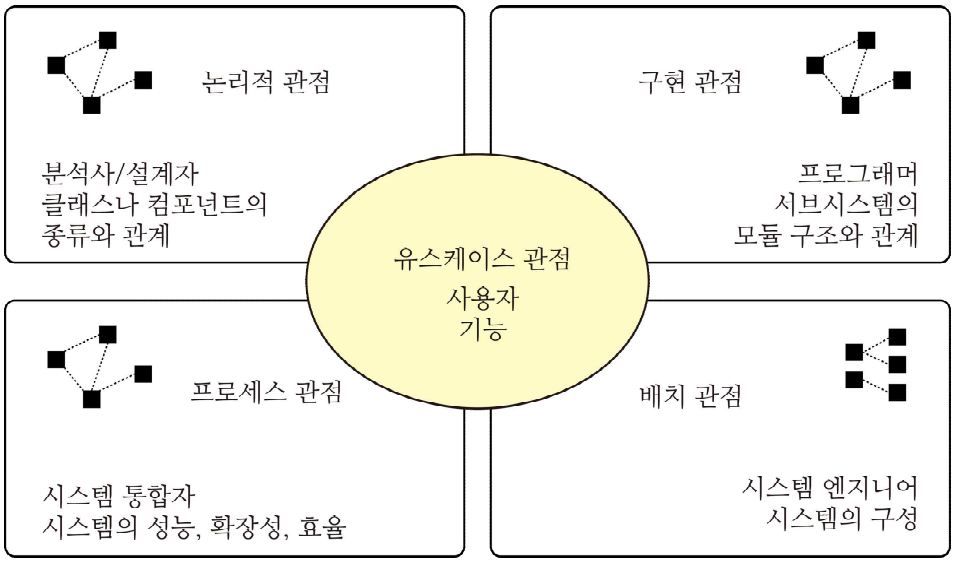
(3) 소프트웨어 아키텍처 프레임워크 구성요소

- 아키텍처 명세서 (Architecture Description)
  - 아키텍처를 기록하기 위한 산출물
- 이해관계자 (Stakeholder)
  - 소프트웨어 시스템 개발에 관련된 모든 사람과 조직
  - 고객, 개발자, 프로젝트 관리자 등 포함
- 관심사 (Concerns)
  - 동일한 시스템에 대해 서로 다른 이해관계자 의견
  - 예) 사용자 : 기본적인 기능, 신뢰성, 보안등의 요구
- 관점 (Viewpoint)
  - 서로 다른 역할이나 책임으로 시스템이나 산출물에 대한 서로 다른 관점
- 뷰(View)
  - 이해 관계자들과 이들이 가지는 생각이나 견해로부터 전체 시스템을 표현

(4) 소프트웨어 아키텍처 4+1 뷰

- 소프트웨어 아키텍처 4+1 뷰 개념
  - 고객의 요구사항을 정리해 놓은 시나리오를 4개의 관점에서 바라보는 소프트웨어적인 접근 방법
  - 복잡한 소프트웨어 아키텍처를 다양한 이해관계자들이 바라보는 관점
  - View는 시스템의 여러 가지 측면을 고려하기 위한 다양한 관점을 바탕으로 정의

- 4+1 View Model 과 구성요소



구성요소	설명
논리 뷰 (logical view)	- 시스템의 기능적인 요구사항 - 시스템이 최종 사용자를 위해 해야 하는 것을 나타낸다.
구현 뷰 (implementation view)	- 개발 환경 안에서 정적인 소프트웨어 모듈의 구성 - 개발자 관점에서 소프트웨어 구현과 관리적인 측면을 컴포넌트 다이어그램으로 표현
프로세스 뷰 (process view)	- 프로그램 실행시의 시스템 표현 - Thread와 Process에 의한 동작을 중점적으로 표현 - 동시성, 분산처리, 시스템 통합, 오류 허용 등을 표현
배치 뷰 (deployment view)	- 다양한 실행 파일과 다른 런타임 컴포넌트가 해당 플랫폼 또는 컴퓨팅 노드에 어떻게 매핑 되는가를 표현 - 가용성, 신뢰성, 성능, 확장성 등의 시스템의 비기능적인 요구사항 고려 - 물리적인 노드의 구성과 상호 연결 관계를 배포 다이어그램으로 표현
유스케이스 뷰 (use case view)	- 아키텍처를 도출하고 설계하는 작업을 주도하는 뷰 - 다른 뷰를 검증하는데 사용 - Use Case Diagram이 사용 - +1 에 해당하며 유스케이스가 나머지 4개 뷰에 모두 참여하면서 영향을 준다.

(5) 소프트웨어 아키텍처 품질속성

- 정확성 (Correctness)
  - 소프트웨어가 사용자의 요구기능을 충족시키는가?
- 신뢰성 (Reliability)
  - 기능이 오차나 오류 없이 동작하는가?
- 효율성 (Efficiency)
  - 기능을 수행하는데 적절한 자원이 소요되는가?
- 무결성 (Integrity)
  - 허용되지 않는 사용이나 자료 변경을 제어하는가?
- 사용 용이성 (Usability)
  - 쉽게 배우고 사용할 수 있는가?
- 유지보수성 (Maintainability)
  - 변경 및 오류 교정 시 쉽게 수정할 수 있는가?
- 시험 용이성 (Testability)
  - 개선, 유지보수 등에 있어서 테스트를 하기 용이하게 되어 있는가?
- 유연성 (Flexibility)
  - 새로운 요구사항에 대해서도 쉽게 개선 및 적용 가능한가?
- 이식성 (Potability)
  - 다양한 플랫폼 및 하드웨어에서 동작 하는가?
- 재사용성 (Reusability)
  - 개발된 기능을 다른 목적으로 사용하기 용이한가?
- 상호 운용성 (Interoperability)
  - 다른 소프트웨어와 상호 교류가 용이한가?

동료검토 - 멘토링 해기
워크스루 - 회의(발표자 - 나)
인스펙션 - 전문가 회의(발표자 - 전문가)

(6) 소프트웨어 아키텍처 평가

- 소프트웨어 아키텍처 평가 개념
  - 아키텍처의 접근법이 품질속성(보안, 성능, UI 등)에 미치는 영향을 판단하여 아키텍처 적합성을 판단하고 평가하는 표준 기법
- 소프트웨어 아키텍처 평가기법 유형

품질 / 비용

관점	유형	내용
가시성	가시적 평가	Inspection, Review, Validation & Verification
	비가시적 평가	SAAM, ATAM, CBAM, ARID, ADR
시점	이른 평가	아키텍처 구축과정 중 어느 때나 평가 가능
	늦은 평가	기존 시스템의 요구사항에 대한 아키텍처의 적합성을 판단할 때 사용

2. 소프트웨어 아키텍처 패턴 유형

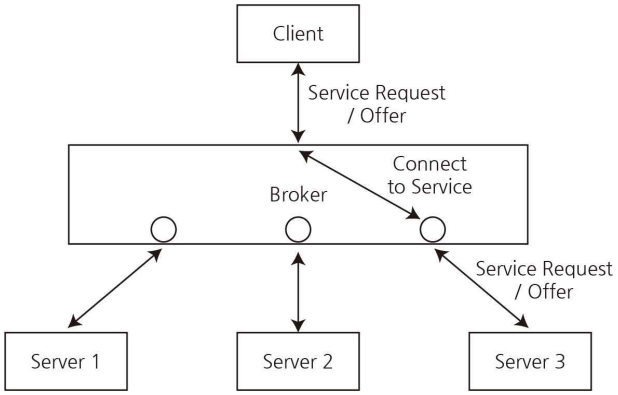
(1) 소프트웨어 아키텍처 패턴의 개념

- 주어진 문맥 안에서 소프트웨어 아키텍처의 공통적인 발생 문제에 대한 일반적인, 재사용 가능한 해결책
- 소프트웨어 공학의 다양한 문제를 해결
  - 컴퓨터 하드웨어 성능
  - 비즈니스 위험의 최소화
  - 고가용성
- 일부 아키텍처패턴은 소프트웨어 프레임워크 안에 구현되어 있다.

(2) 소프트웨어 아키텍처 패턴 종류

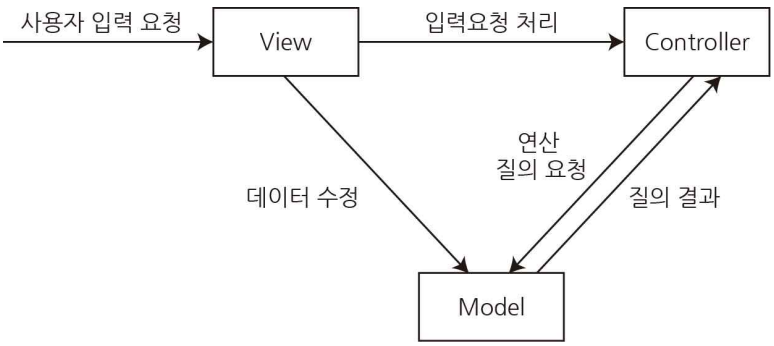
- 계층화 패턴(Layered pattern)
  - n-티어 아키텍처 패턴으로 부른다.
  - 하위 모듈을 그룹으로 나눌 수 있는 구조화된 프로그램에서 사용
  - 각 서브 시스템이 하나의 계층이 되고 하위층이 제공하는 서비스를 상위층의 서브 시스템이 이용할 수 있는 구조
  - 예) OSI 7계층, TCP/IP 4계층
- 클라이언트-서버 패턴(Client-Server Pattern)
  - 다수의 클라이언트와 하나의 서버로 구성
  - 서버는 클라이언트에게 서비스를 제공하며 데이터를 관리하는 역할
  - 예) 일반적인 웹 프로그램
- 마스터-슬레이브 패턴(Master-Slave Pattern)
  - 마스터 컴포넌트가 동등한 구조의 슬레이브 컴포넌트로 작업을 분산하고, 슬레이브가 결과값을 반환하면 최종 결과값을 계산하는 구조 마스터-연산 슬레이브-입,출력
  - 예) 컴퓨터와 주변장치
- 파이프-필터 패턴(Pipe-Filter Pattern)
  - 데이터 스트림을 생성하고 처리하는 시스템에서 사용 가능한 패턴
  - 필터 컴포넌트에서 각 처리과정을 실행하며, 처리된 데이터는 파이프를 통해 전송
  - 서브시스템이 입력데이터를 받아 처리하고 결과를 다음 서브시스템으로 넘겨주는 과정을 반복
  - 예) Unix 쉘처리

- 브로커 패턴(Broker Pattern)
  - 분리된 컴포넌트로 구성된 분산 시스템에서 사용되는 패턴
  - 각 컴포넌트들은 원격 서비스를 통해 서로 상호작용을 할 수 있으며 브로커 컴포넌트가 컴포넌트 간의 통신을 조절



- 피어 투 피어 패턴(Peer to Peer Pattern)
  - 피어라 부르는 각 컴포넌트 간에 서비스를 주고 받는 패턴
  - 피어 객체 하나가 클라이언트, 서버의 역할을 모두 수행하는 구조
  - 예) 파일 공유(P2P)
- 이벤트-버스 패턴(Event-Bus Pattern)
  - 이벤트 버스를 통해 특정 채널로 메시지를 발행
  - 리스너가 구독한 채널에 소스가 서비스를 제공하면 채널이 리스너에게 서비스를 제공
  - 예) 알림 서비스
- 모델-뷰-컨트롤러 패턴(Model-View-Controller Pattern) - MVC Pattern
  - 3개의 각 컴포넌트는 각자의 역할을 갖고 사용자에게 서비스를 제공
  - 자료의 저장, 제어, 표현 기능을 분리하여 재사용을 증진
  - 모델 : 도메인의 기능과 자료를 저장하고 보관
  - 뷰 : 사용자에게 결과를 표시
  - 컨트롤러 : 사용자로부터 입력을 받아 연산을 처리
  - 예) 일반적인 웹 어플리케이션 설계 아키텍처

Data M
C 처리
V 화면



- 블랙보드 패턴(Blackboard Pattern)
  - 명확히 정의된 해결 전략이 알려지지 않은 문제에 대해서 유용한 패턴
- 인터프리터 패턴(Interpreter Pattern)
  - 특정 언어로 작성된 프로그램을 해석하는 컴포넌트를 설계할 때 사용되는 패턴