Software Architecture

엄진영

교재 및 참고 문헌

교재

- Fundamentals of Software Architecture (2nd Edition), Mark Richards & Neal Ford [O'Reilly]

서론

소프트웨어 아키텍처를 왜 알고 싶은가?

개발자 프로젝트 매니저

여러 프로젝트 경험

“시스템의 큰 구조들이 어떻게 맞물리는지,

그 안에 어떤 트레이드오프가 존재하는지를 더 깊게 이해하고 싶다!”

소프트웨어 아키텍트

소프트웨어 아키텍처

소프트웨어 아키텍트란?

✓ 복잡하기 짝이 없는 소프트웨어 시스템을 깊이 이해하고 분석하며,

✓ 때로는 중요한 트레이드오프를 불완전한 정보에 근거해서 **결정**하는 사람

↓

“복잡하게 변하는 상황에서 여러 선택지를 저울질하여

최적의 균형점을 찾는, 바로 그런 결정”

↓ 자신이 처한 환경의 현실에 근거해서 결정을 내린다.

모든 아키텍처는 자신이 탄생한 환경의 영향을 받는다!

소프트웨어 아키텍처의 정의

아키텍처 특성
Arch. characteristic

시스템의 역량 정의
- 가용성
- 신뢰성
- 확장성
- 보안
- 회복성
- 성능
- 배포성

소프트웨어 아키텍처

아키텍처 스타일
Arch. style

아키텍처의 특징/특성 설명
- 컴포넌트 토폴로지
- 물리적 아키텍처
- 배포
- 통신 스타일
- 데이터 토폴로지

논리적 컴포넌트
Logical component

시스템의 행동방식을 규정
- 도메인
- 엔티티
- 작업흐름

* 행동방식: 조건/논리 + 행동(action)

아키텍처적 결정
Architectural decision

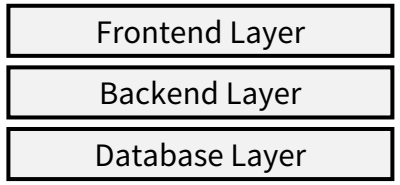
시스템을 구축할 때 지켜야 할 규칙
- 시스템에 어떠한 제약조건을 추가
- 개발 팀이 무엇을 해도 되고 무엇을 해서는 안되는지에 대한 지침을 제공

토폴로지(topology)?

"물리적 모양이나 정확한 위치를 버리고,

구성 요소들이 어떻게 **연결**되고 **의존**하는지를 설명하기 위한 개념"

Structure



- 3-tier 구조
- 레이어별 역할 구분

용도: **정적인 설계 설명**

- 레이어드 아키텍처
- 디렉토리 구조
- 클래스 구성
- DB 스키마

시스템의 뼈대

(무엇이 어떻게 만들어져 있는가?)

웹 서비스 예:

Topology

- 호출 흐름
- 의존 관계

용도: **동적인 관계 설명**

- 네트워크 연결
- 서비스 호출 관계
- 클러스터 구성
- 장애 영향 분석

그 뼈대 위에서 흐르는 **연결과 관계**

(그것들이 어떻게 연결되어 있는가?)

소프트웨어 아키텍처의 법칙

제1법칙

- 소프트웨어 아키텍처의 모든 것은 트레이드오프(trade-off)이다.
- 트레이드오프가 아닌 무언가를 발견했다고 생각한다면,
  아마도 트레이드오프를 아직 찾아내지 못했을 가능성이 높다.
- 트레이드오프 분석을 단 한 번만 하고 끝낼 수 없다.

제2법칙

- 어떻게(방법) 보다 왜(이유)가 더 중요하다.

제3법칙

- 대부분의 아키텍처적 결정은 양자택일이 아니라
  양극단 사이의 스펙트럼에 있는 한 지점이다.

소프트웨어 아키텍트의 기대 역할

아키텍처적 결정을 내린다.

아키텍트는 팀이나 부서, 또는 전체 조직에서 기술 선택의 지침이 될 아키텍처적 결정과 설계 원칙을 정의해야 한다.

“프런트엔드 웹 개발에 반응형 프레임워크를 사용하라” – 팀들이 올바른 기술을 선택하도록 이끄는 지침

“프런트엔드 개발에 React를 사용하라” – 기술 선택이지 아키텍처적 결정이 아니다.

아키텍처를 지속적으로 분석한다.

아키텍트는 아키텍처와 현재 기술 환경을 지속적으로 분석하고, 개선 방안을 제안해야 한다.

“아키텍처 활력(vitality)” – 비즈니스나 기술의 변화 속에서 현재 시점에서 어느 정도나 “**살아 있는지**”를 평가하는 개념

“기술이나 문제 도메인의 변화까지도 전체적으로 고려해서, 아키텍처가 현재 상황에서도 여전히 건전한지를 평가해야 한다.”

지속적으로 분석하는 데 충분한 노력을 기울이지 않는다면 → 아키텍처에서 구조적 붕괴 현상이 발생!

소프트웨어 아키텍트의 기대 역할(계속)

최신 트렌드를 계속 따라간다.

아키텍트는 개발자처럼 최신 기술을 꾸준히 익히고 일상에서 사용하는 기술도 항상 최신 상태로 갱신해야 한다.

↳ 아키텍처가 내리는 결정은 오랫동안 영향을 미치며 뒤집기 어렵다.
트렌드를 꿰뚫고 있으면 미래에도 유효한 결정을 내릴 수 있다.

결정 사항의 준수를 보장한다.

아키텍트는 아키텍처적 결정과 설계 원칙의 준수를 보장해야 한다.

↳ 아키텍트가 정의하고 문서화해서 전달 아키텍처적 결정과 설계 원칙을 개발 팀들이 실제로 따르는지 지속적으로 확인하는 것.
아키텍처적 결정 사항을 준수하지 않으면 → 아키텍처가 원래 의도했던 특성들을 보장하지 못할 수 있으며, 애플리케이션이나 시스템이 제대로 동작하지 않는 상태로 이어진다.

소프트웨어 아키텍트의 기대 역할(계속)

다양한 기술을 이해한다.

아키텍트는 여러 기술, 프레임워크, 플랫폼, 환경을 접해야 한다.

↳ 모든 부문에서 전문가일 필요는 없다. 하지만 다양한 기술에 어느 정도 익숙해야 한다.

기술의 깊이 보다는 너비에 중점을 둘 필요가 있다.

비즈니스 도메인을 숙지한다.

아키텍트는 비즈니스 도메인의 전문성을 일정 수준 이상으로 갖춰야 한다.

↳ 비즈니스 도메인(비즈니스 문제, 목표, 요구사항)을 이해하지 않은 채로 효과적인 아키텍처를 설계하기 어렵다.

비즈니스 도메인에 익숙하지 않으면 → 이해관계자나 비즈니스 사용자와 의사소통이 원활하지 않아서 신뢰를 잃기 쉽다.

소프트웨어 아키텍트의 기대 역할(계속)

대인 관계 스킬을 갖춘다.

아키텍트는 탁월한 대인 관계 스킬을 갖춰야 한다. 그러한 능력에는 **팀워크**, **조정력**, **리더십**이 포함된다.

아키텍트의 역할은 단순히 기술적인 지침을 제공하는 것에 그치지 않는다.
개발 팀을 이끌어 아키텍처를 실제로 구현하도록 주도하는 일까지 포함된다.

정치를 파악하고 헤쳐 나간다.

아키텍트는 기업의 정치적 환경을 파악하고 헤쳐 나갈 수 있어야 한다.

아키텍트가 내리는 결정은 제품 관리자나 프로젝트 관리자, 비즈니스 이해관계자, 개발자들에게 막대한 영향을 끼친다.
관련자들을 설득하려면, 조직 내 정치적 상황을 잘 파악하고 협상 능력을 발휘해야 한다.

기초 - 아키텍처적 사고

아키텍처적 사고(architectural thinking)

"아키텍처적 사고란,

사물을 설계자의 눈으로 보는 것, 즉 아키텍처의 관점에서 바라보는 것이다"

"**뭔가를 변경할 때**" (아키텍처적 사고의 예)

- ✓ 그것이 전반적인 확장성에 어떤 영향을 미칠지 이해해야 한다.
- ✓ 시스템의 서로 다른 부분이 어떻게 상호작용하는지 주의를 기울여야 한다.
- ✓ 주어진 상황에 적합한 서드파티 라이브러리나 프레임워크를 파악하고 식별해야 한다.

아키텍처적 사고 - 필요역량

- ✓ 소프트웨어 아키텍처가 **무엇인지 이해**해야 한다.
- ✓ 남들이 보지 못하는 해결책과 가능성을 인지할 수 있는 **폭넓고 풍부한 지식을 갖춰야** 한다.
- ✓ **비즈니스 동인의 중요성을 이해**해야 한다.
- ✓ 비즈니스 동인이 아키텍처상의 문제로 **어떻게 이어지는지 알아야** 한다.
- ✓ 다양한 솔루션과 기술의 트레이드오프를 **이해, 분석, 조율하는 능력**도 필요하다.

* 비즈니스 동인(business driver)?
조직이 '**왜**' 이 시스템을 만들고, '**무엇을 가장 중요하게**' 여기는지를 결정하는 **근본적인 요인**

기능 요구사항 vs 비기능 요구사항 vs 비즈니스 동인

기능 요구사항	✓ 로그인, 결제, 검색 기능
비기능 요구사항	✓ 성능, 보안, 확장성
비즈니스 동인	✓ 왜 이 기능/비기능이 중요한가? **“요구사항을 만들어내는 배경 원인”**

아키텍처와 비즈니스 동인의 관계

비즈니스 목표

기업 또는 조직이 중.장기적으로 달성하고자 하는 **전략적 상태**

- "무엇을 이루고 싶은가"
- 기술·시스템과 독립적, 비교적 장기적

↓

비즈니스 동인

비즈니스 목표를 향한 의사결정을 **지금 이 시점에서 강제하는 구체적인 압박, 제약, 또는 우선순위 요인**

- "왜 지금인가, 왜 반드시 고려해야 하는가"
- 실패 시 손실·위험이 명확, 아키텍처 선택을 압박함

↓

아키텍처 요구

비즈니스 동인을 시스템 관점에서 해석한 결과로 **시스템이 가져야 할 핵심 성질(품질 속성)과 제약 조건**

- "그래서 시스템은 어떤 성질을 가져야 하는가"
- 기능이 아닌 품질 중심, 기술 중립적, 여러 아키텍처 결정의 기준이 됨

↓

아키텍처 결정

아키테처 요구를 만족시키기 위해 선택한 **구체적인 구조, 패턴, 기술, 분리 방식에 대한 결정**

- "그래서 어떻게 구현할 것인가"
- 되돌리기 비용이 큼, 트레이드오프를 포함, 명시적으로 기록되어야 함

아키텍처와 비즈니스 동인의 관계 – 1) 비즈니스 목표

“무엇을 이루고 싶은가”

2년 내 월간 활성 사용자(MAU) 100만 명 달성 및
시장 선도 기업으로 자리매김
(교육 콘텐트 서비스)

아키텍처와 비즈니스 동인의 관계 – 2) 비즈니스 동인

“왜 지금인가, 왜 반드시 고려해야 하는가”

국내 교육 시장 포화로 인한 성장 한계

국내 잠재 고객 풀이 30만 명 수준에서 정체, 목표 달성을 위해 해외 시장 진출이 불가피

스트리밍 품질 저하 시 즉각적인 환불 요청과 이탈 발생

경쟁사 대비 버퍼링 발생 시 48시간 내 20% 이탈 데이터 확인, 품질 이슈는 소셜미디어에서 빠르게 확산되어 브랜드 신뢰 타격

경쟁사의 공격적인 기능 출시 주기 (월 1회)

현재 분기별 배포로는 시장 선도 이미지 유지 불가, 지난 분기 AI 추천 기능 출시 지연으로 5,000명 이탈 경험

아키텍처와 비즈니스 동인의 관계 – 2) 비즈니스 동인(계속)

"왜 지금인가, 왜 반드시 고려해야 하는가"

저녁 8-10시 트래픽 집중으로 인한 서비스 불안정

전체 트래픽의 70%가 2시간에 집중, 3개월 전 장애로 인한 보상비용 3억 원 발생

강의 수 급증(월 300개 신규 강의)으로 인한 사용자 선택 피로

콘텐츠 증가로 인해 사용자가 적합한 강의를 찾지 못하고 신규 가입자 이탈률이 급증

아키텍처와 비즈니스 동인의 관계 – 3) 아키텍처 요구

"그래서 시스템은 어떤 성질을 가져야 하는가"

국내 교육 시장 포화로 인한 성장 한계

- **확장성** → 사용자 수 증가에 따라 성능 저하 없이 확장 가능해야 함
- **글로벌 배포 가능성** → 지역별 사용자 증가에 대응 가능
- **운영 자동화** → 소수 인력으로 다지역 운영 가능해야 함

스트리밍 품질 저하 시 즉각적인 환불 요청과 이탈 발생

- **가용성** → 핵심 기능(스트리밍)은 항상 사용 가능해야 함
- **성능** → 재생 지연, 끊김 최소화
- **장애 격리** → 일부 장애가 전체 서비스 품질로 확산되지 않아야 함
- **관측 가능성** → 품질 저하를 즉시 탐지 가능해야 함

아키텍처와 비즈니스 동인의 관계 – 3) 아키텍처 요구(계속)

"그래서 시스템은 어떤 성질을 가져야 하는가"

경쟁사의 공격적인 기능 출시 주기 (월 1회)

- **변경 용이성** → 기능 변경·추가가 쉽고 빠르게 가능해야 함
- **배포 독립성** → 특정 기능 변경이 전체 시스템 배포로 이어지지 않아야 함
- **개발 생산성** → 작은 팀에서도 빠른 개발 가능

저녁 8-10시 트래픽 집중으로 인한 서비스 불안정

- **탄력적 확장성** → 피크 시간대에 자동 확장 가능해야 함
- **부하 내성** → 급격한 트래픽 증가에도 서비스 유지
- **우아한 성능 저하** → 일부 기능 제한으로 핵심 기능 보호

아키텍처와 비즈니스 동인의 관계 – 3) 아키텍처 요구(계속)

"그래서 시스템은 어떤 성질을 가져야 하는가"

강의 수 급증(월 300개 신규 강의)으로 인한 사용자 선택 피로

- **데이터 확장성** → 콘텐츠 및 사용자 데이터 증가에 대응 가능해야 함
- **분석 가능성** → 사용자 행동 분석을 통해 개선 가능해야 함
- **실험 용이성** → 추천·노출 로직을 빠르게 실험 가능해야 함

아키텍처와 비즈니스 동인의 관계 – 4) 아키텍처 결정

“그래서 어떻게 구현할 것인가”

모듈형 모놀리스 + 핵심 도메인 분리

초기에는 모듈형 모놀리스로 개발 속도/단순 운영 확보, 트래픽/리스크가 큰 영역은 별도 서비스로 분리(Strangler 패턴)

→ 개발 생산성, 변경 용이성(초기), 장애 격리, 배포 독립성(점진적)

CDN 중심의 “Edge-first” 콘텐츠 전송

동영상/이미지/정적 리소스는 CDN으로 오프로딩, 지역(국가)별 캐시 정책/프리페치 전략 적용

→ 글로벌 배포 가능성, 성능(스트리밍 품질), 부하 내성

Stateless 애플리케이션 + 수평 확장 기본값

웹/API 서버는 상태를 갖지 않게 설계(세션은 외부 저장소로 이동), 피크 트래픽은 오토스케일로 대응

→ 확장성, 탄력적 확장성, 운영 자동화

아키텍처와 비즈니스 동인의 관계 – 4) 아키텍처 결정(계속)

"그래서 어떻게 구현할 것인가"

핵심 경로 보호

피크 시간대/장애 시: 스트리밍/강의 시청(핵심)은 유지, 추천/랭킹/비필수 분석 기능은 제한 또는 캐시된 결과 제공

"읽기 우선(serve cached)" 전략

→ 우아한 성능 저하, 가용성, 부하 내성

회복 탄력성 패턴 표준화

Circuit Breaker/Retry(백오프)/Timeout/Bulkhead(격리) 적용, 외부 의존성(결제, 추천, 검색 등) 호출에 정책 통일

→ 장애 격리, 가용성, 부하 내용

관측 3종 세트(Logs/Metrics/Traces) + SLI/SLO 운영

스트리밍 품질 지표를 "비즈니스 지표"로 승격: 재생 시작 시간, rebuffering 비율, 실패율, 지역별 품질

SLO 위반 시 자동 알림/롤백/기능 제한

→ 관측 가능성, 품질 저하를 "48시간 내 이탈" 전에 탐지

아키텍처와 비즈니스 동인의 관계 – 4) 아키텍처 결정(계속)

"그래서 어떻게 구현할 것인가"

Feature Toggle + A/B 테스트를 아키텍처 기본 기능으로

추천/콘텐츠 노출/UX 변경은 토글로 분리, 실험은 서버/클라이언트에서 일관되게 적용(실험 ID 고정)

→ 변경 용이성, 실험 용이성, 빠른 신규 기능 출시(배포 부담을 줄임)

CI/CD 표준화 + 점진 배포(카나리/블루그린)

월 1회 이상 배포를 목표: 자동 테스트(단위/통합), 카나리 배포로 위험 감소, 실패 시 자동 롤백

→ 배포 독립성, 개발 생산성, 가용성(배포로 인한 장애 감소)

OLTP(서비스 DB)와 분석/이벤트 저장 분리

운영 DB는 트랜잭션(가입/수강/결제)에 집중, 사용자 행동 이벤트는 별도 파이프라인으로 수집(분석용 저장소)

→ 데이터 확장성, 분석 가능성, 운영 성능 보호(피크 시간대에도 OLTP 안정)

아키텍처와 비즈니스 동인의 관계 – 4) 아키텍처 결정(계속)

“그래서 어떻게 구현할 것인가”

추천/검색을 “독립 진화 가능한 컴포넌트”로 분리

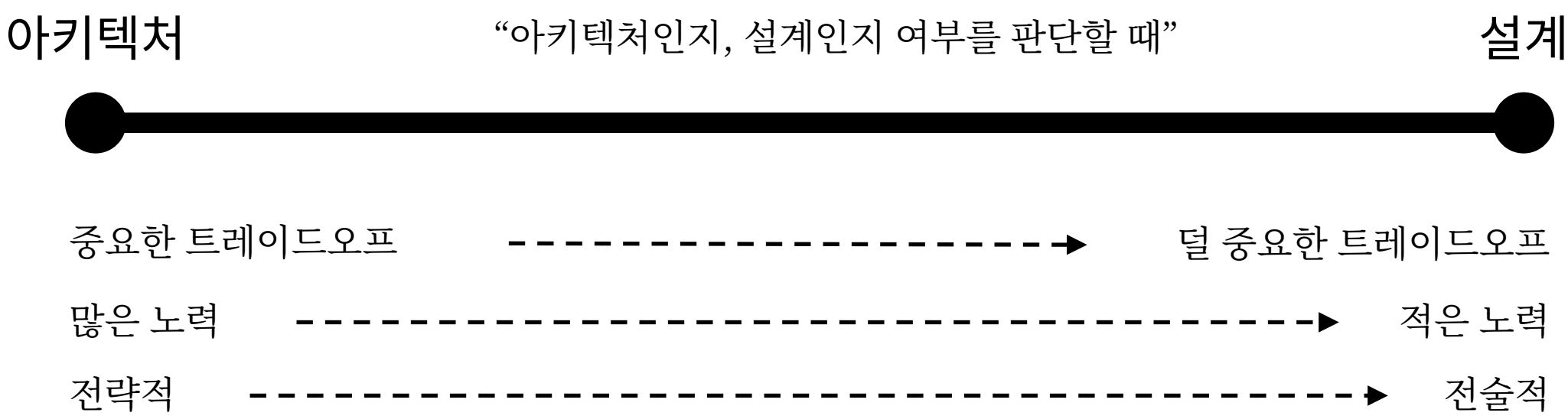
초기에는 같은 코드베이스 내 모듈로 시작 가능 → 트래픽/변경이 커지면 별도 서비스로 분리

모델/랭킹 로직은 버전 관리(롤백 가능)

→ 실험 용이성, 변경 용이성, 장애 격리(추천 장애가 시청을 죽이지 않게)

아키텍처와 설계의 차이

아키텍처	시스템의 전체 **구조**를 규정한다.
설계	개별 구성 요소를 **어떻게 구현할지**를 정하는 구체적 결정



아키텍처와 설계의 차이 – 전략적 결정과 전술적 결정

"전략적일수록 아키텍처에 가깝고, 전술적일수록 설계에 가깝다"

✓ 결정에 얼마나 많은 사고와 계획이 필요한가?

몇 분 만에 내릴 수 있는 결정 → 전술적 성격이 강함 → 설계에 가깝다.

몇 주 동안의 계획이 필요 ➔ 전략적인 결정 ➔ 아키텍처에 가깝다.

✓ 결정에 몇 명이 관여하는가?

혼자 또는 동료 한두 명과 내리는 결정 → 전술적일 가능성이 높음 → 설계에 가깝다.

다수의 이해관계자와 여러 차례 회의해야 하는 결정 ➔ 전략적일 가능성이 높음 ➔ 아키텍처에 가깝다.

✓ 결정이 장기적인 비전인가, 단기적인 행동인가?

곧 변경될 가능성이 큰 결정 → 전술적 특성을 가짐 → 설계에 가까운 주제이다.

오랜 기간 지속될 결정 ➔ 전략적임 ➔ 아키텍처 쪽에 속한다.

아키텍처와 설계의 차이 – 노력의 정도

"변경이 어려울수록 대체로 더 많은 노력이 필요하다. 노력이 많이 필요한 결정은 아키텍처에 가깝다."

아키텍처 변경의 예:

모놀리스 방식의 계층형 아키텍처 ➔ 마이크로서비스

→ 상당한 노력이 필요하다.

설계 변경의 예:

UI 화면의 입력 필드 순서를 바꾸는 일

→ 노력이나 비용이 덜 든다.

아키텍처와 설계의 차이 – 트레이드오프 중요성

"중요한 트레이드오프일수록 해당 결정은 아키텍처에 가깝다"

아키텍처

마이크로서비스 아키텍처 스타일

장점

- **확장성**, **민첩성**, **탄력성**, **내결함성**(fault tolerance) 등이 좋아진다.

단점

- 시스템이 **매우 복잡**해지고 **비용**이 많이 든다.
- 데이터 **일관성**이 낮아 진다.
- 서비스 사이에 **결합도**가 높아서 **성능**이 떨어진다.

마이크로서비스의 트레이드오프에 비해 중요도가 떨어진다.

설계

클래스 파일을 여러 조각으로 쪼개기

장점

- **유지보수성**과 **가독성**이 좋아진다.

단점

- 관리해야 할 클래스가 늘어나므로 **관리 부담**이 늘어난다.

개발자 vs 아키텍트

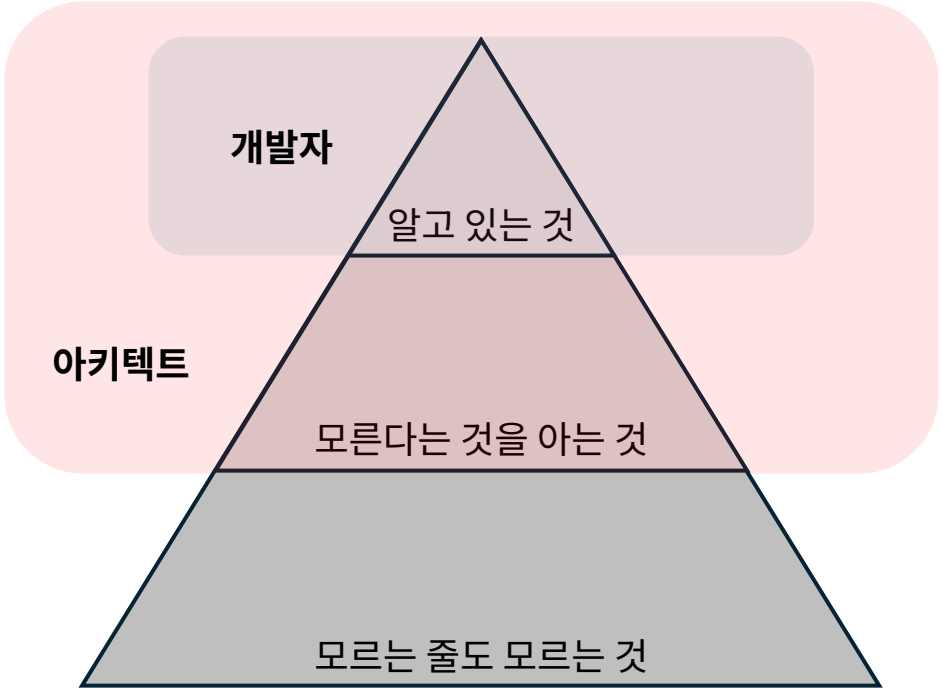
"개발자는 기술적 깊이에 중요하고, 아키텍트는 기술적 너비가 중요하다"

개발자 - 기술의 깊이(technical depth)

- 특정 프로그래밍 언어나 플랫폼, 프레임워크, 제품 등에 대해 **깊이 있게 아는 것**.
- 지식을 상단 영역에 계속 두려면 **시간을 들여서 해당 전문성을 유지**해야 한다.

아키텍트 - 기술의 너비(technical breadth)

- 많은 분야를 조금씩 아는 것. 즉, 기술들을 **폭넓게 이해**하고 특정 문제 해결에 기술을 어떻게 활용할지 아는 것.
- 힘들게 쌓은 일부 전문성을 포기하고 그 시간 만큼 자기 **기술 포트폴리오의 폭을 넓혀야** 한다.



개발자에서 아키텍트로 전환하는 과정에 나타나는 문제점

"개발자 관점에서 아키텍트 관점으로 전환하는 것은 어렵다"

첫째, 아키텍트가 많은 영역의 전문성을 유지하려다

어느 것 하나 성공하지 못하고, 그 과정에서 소진된다.

둘째, 자신의 오래된 지식이 여전히 가장 앞선 정보라고 착각하는

케케묵은 전문 지식 현상이 나타난다.

예) 꽁꽁 언 원시인 안티패턴(Frozen Caveman antipattern)

- 아키텍트가 **자신이 집착하는 비합리적인 관심사로 모든 아키텍처를 판단**하는 행동적 패턴을 가리킨다.
- 과거의 잘못된 결정이나 뜻밖의 사건에 데인 경험이 있는 아키텍트가 관련된 **모든 것에 과도하게 신중해지는 형태**로 나타난다.

* 안티패턴(antipattern)?
처음에는 좋은 생각 같았지만, 결국은 문제를 일으키는 어떤 것

아키텍트가 기술의 너비를 넓히는 방법

"개발자에서 아키텍트 역할로 전환하는 사람은 지식 습득에 대한 관점을 바꿔야 한다"

방법1 - 20분 규칙

- **매일 최소 20분을 새로운 내용을 학습**하거나 특정 주제에 대해 더 깊이 파고드는 데 사용한다.
- 처음 듣는 **유행어나 용어는 인터넷으로 검색**하여, '모르는 줄도 모르는 것'을 '모른다는 것을 아는 것'으로 바꾼다.
- **아침에 이메일을 확인하기 전, 즉 집중력이 떨어지기 전에 20분을 투자**하라. 점심 시간은 밀린 업무를 처리하느라, 퇴근 후는 피곤한 데다가 가족 등의 일정 때문에 20분을 확보하기 어렵다.
- 바쁘더라도 기술의 너비를 넓히는 데 집중할 시간을 의도적으로 확보하는 것이다.

아키텍트가 기술적 너비를 확보하는 방법(계속)

"개발자에서 아키텍트 역할로 전환하는 사람은 지식 습득에 대한 관점을 바꿔야 한다"

방법2 – 개인 레이더 개발

- 개발자나 아키텍트, 기타 기술전문가가 엄청난 시간과 노력을 투자해서 특정 기술을 공부하고 훈련하다 보면 그 기술에 과도하게 몰입해서 **밈 거품(memetic bubble)에 갇히게** 된다.
- 밈 거품 안에서는 다른 모든 사람도 자신만큼 그 기술을 중요하게 여기고 잘 안다고 느끼며, 거품 밖에서 제시되는 진솔한 평가를 제대로 보지 못하게 된다. 일종의 **반향실 효과가 발생**한다.
- **이런 기술 거품에 빠지지 않으려면, 개인 기술 레이더를 준비해야 한다. 기술 레이더란, 기존 또는 신생 기술의 위험과 보상을 평가할 수 있게 돕는 '살아 있는' 문서를 말한다.**
- 참고: 소트웍스(ThoughtWorks)의 'Build Your Own Radar(https://oreil.ly/IV7G8)' 도구

* 반향실(echo chamber)?
소리가 벽면에서 반사되어 메아리가 계속 울려 퍼지는 밀폐된 공간이다. 여기서는 **특정 정보나 신념이 닫힌 체계 안에서 반복적으로 공유되면서 강화되고, 외부의 반대 의견은 차단되는 환경**을 의미한다.

소트웍스 기술 레이더 예)

Techniques

IDE 등 개발 도구부터 엔터프라이즈급 통합 도구까지 모든 도구를 포함한다.

컴퓨터 언어, 라이브러리, 프레임워크를 다룬다. 보통 오픈소스 기반이다.

Languages & Frameworks

Hold 회피해야 할 기술과 기법, 고치고 싶은 습관. 예) 최신 소식 찾아보는 습관

Assess 유망하다고 느끼지만 아직 직접 검증하지 못한 기술. 본격적인 연구를 위한 대기 장소.

Trial 적극적인 연구와 개발의 영역. 대규모 코드베이스 안에서 스파이크 실험을 해보는 것.

Adopt 특정 문제를 해결하는 데 최고의 관행이라고 간주하는 새로운 기술이나 기법.

소프트웨어 개발 전반을 돕는 모든 관행. 프로세스, 엔지니어링 관행, 다양한 조언 등.

Tools

데이터베이스, 클라우드 제공업체, 운영체제 등 기술 플랫폼을 다룬다.

Platforms

트레이드오프 분석(아키텍처적 사고의 한 부분)

"모든 해법에서 트레이드오프(절충점)를 찾아내고, 그 장단점을 분석해서 최선의 해법을 도출하는 것"

- "REST와 메시징 중 어느 쪽이 내 시스템 더 좋은가?"
- "마이크로서비스가 새 제품을 위한 올바른 아키텍처 스타일인가?"

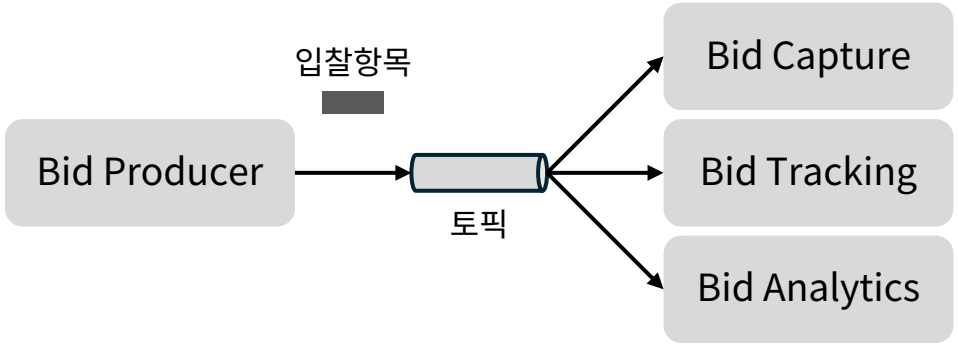
상황에 따라 다르다(It depends)

배포할 환경, 비즈니스 목표, 조직 문화, 예산, 일정, 개발자 역량 같은 **수많은 요소에 따라 달라진다**.

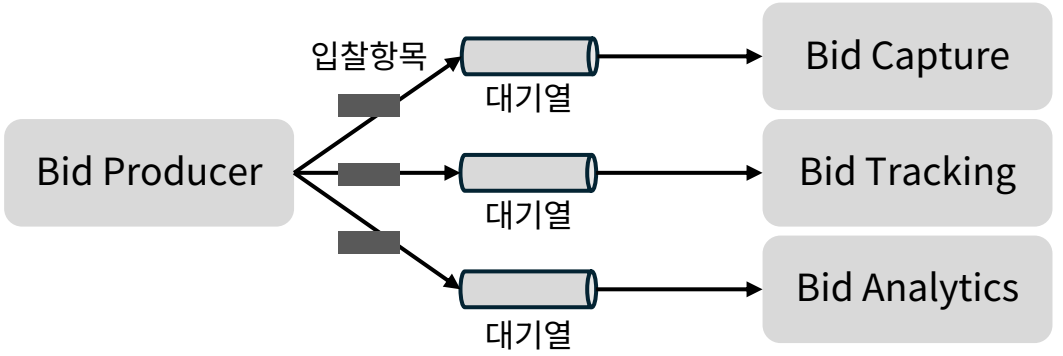
"아키텍처에는 **정답도 오답도 없다**. **트레이드오프가 있을 뿐**이다" – 닐 포드

트레이드오프 분석 예 - 토픽 vs 대기열

서비스 간 통신에 **토픽을 사용**하는 예:



서비스 간 통신에 **대기열 사용**하는 예:



	토픽 방식	**대기열 방식**
아키텍처 확장능력 예) BidHistory 추가	서비스 추가가 **쉽다**. 기존의 서비스나 인프라를 고칠 필요가 없다. BidHistory가 토픽을 구독하면 그만이다.	서비스 추가가 **어렵다**. 기존 서비스와 인프라를 수정해야 한다. 새 대기열을 만들고, BidProducer와 연결해야 한다.
결합도	시스템 결합도가 **낮다**. BidProducer는 입찰 정보가 어떻게 쓰이는지, 다른 서비스가 어떻게 사용하는지 알 필요가 없다.	시스템 결합도가 **높다**. BidProducer는 입찰 정보를 누가 어떻게 사용하는지를 구체적으로 알게 된다.
데이터 보안	접근 및 보안 **문제 발생**. 토픽 구독으로 누구나 입찰 데이터에 접근 가능하다. 도청이 쉽다.	접근 제약에 따른 **보안 강화**. 생산자가 보낸 메시지를 특정 소비자만 받을 수 있다. 도청 시에 즉시 데이터 손실 알림을 받을 수 있다.
모니터링	**개별** 모니터링이 **어렵다**.	대기열 별 **개별** 모니터링이 **가능**하다.

비즈니스 동인의 이해

"시스템 자체를 이해하는 것을 넘어서 **시스템의 성공에 필요한 비즈니스 동인들도 이해해야** 한다."

비즈니스 동인 이해 —도출→ **아키텍처 특성(확장성, 성능, 가용성 등)**

- 비즈니스 도메인 지식이 필요하다.
- 주요 비즈니스 이해관계자들과 원만하고 협력적인 관계를 맺어야 한다.

아키텍처와 코딩 실무의 균형

"아키텍트가 직면하는 난제 중 하나는 **코딩 실무와 소프트웨어 아키텍처 작업의 균형을 맞추는 일**이다."

- 아키텍트가 시스템의 핵심 경로에 위치한 코드를 소유 함 → 팀 전체의 병목이 되는 상황
- 개발 외에 다이어그램 작성, 회의 참석 등 아키텍트 역할도 수행 → 코드 작성이 늦어짐
  → 팀 전체의 발목을 잡게 된다.

"병목 함정(Bottleneck Trap)에 빠지지 말라"

실천 방안:

- 시스템의 핵심 부분을 개발 팀 내에 **다른 이들에게 위임**한다.
- 핵심 기능 구현 이후에, 소규모 **비즈니스 기능성(서비스나 UI화면)에 초점을 둔 코딩 실무**를 맡는다.

효과:

- 실제 프로덕션 코드를 작성하면서도 팀의 병목이 되지 않는다.
- 개발자들이 시스템의 좀 더 어려운 부분에 대해 소유권을 가지고 시스템을 더 깊이 이해하게 만든다.
- 아키텍트가 비즈니스 관련 코드 작성 → **개발과정, 프로세스, 개발 환경 등 팀이 겪는 고충을 좀 더 잘 이해할 수 있다.**

아키텍처가 코드 작성에 참여하지 못하는 상황

"실무 감각을 유지하고 기술적 깊이를 일정 수준 이상 보장하려면?"

PoC를 자주 진행

- PoC(proofs-of-concept; 개념 증명) 작업을 자주 진행하면 아키텍트가 직접 소스 코드를 작성하게 된다.
- 아키텍처적 결정을 실제 구현 관점에서 검증하는 데에도 도움이 된다.
- 예) 두 가지 캐싱 솔루션을 선택할 때 → 각 캐싱 솔루션을 사용하는 예제 시스템들을 실제로 개발해서 비교해 본다.
- 주의! – 아키텍트는 가능한 한 프로덕션급 품질의 코드를 작성해야 한다. → 다른 이들이 참고할 수 있기 때문이다.

기술 부채 해결

- 기술 부채가 해결되면 개발 팀은 중요한 기능적 사용자 스토리 작업에 집중할 수 있다.
- 기술 부채는 우선순위가 낮으므로 아키텍트가 특정 반복 기간(iteration) 안에 작업을 완료하지 못해도 악영향을 미치지 않는다.

버그 잡기

- 반복 동안(iteration) 버그를 고치는 것도 실무 감각을 유지하고 개발 팀에 도움을 주는 방법이다.
- 코드베이스와 아키텍처의 문제점이나 약점을 발견하는 기회가 된다.

아키텍처가 코드 작성에 참여하지 못하는 상황(계속)

"실무 감각을 유지하고 기술적 깊이를 일정 수준 이상 보장하려면?"

자동화

- 간단한 명령줄 도구나 분석기를 만들어서 개발 팀의 일상 업무를 자동화하는 것도 코딩 실무 역량을 유지하면서 개발 팀의 생산성을 높이는 방법이다.
- 개발 팀이 여러 번 되풀이하는 작업을 찾아서 자동화해보자.

코드 검토

- 코드 검토를 자주 진행하는 것도 아키텍트가 코딩 실무 감각을 유지하는 방법이다.
- 코드 검토를 진행하면,
  - 코드를 직접 작성하지는 않더라도 소스 코드에 관여하게 되며,
  - 아키텍처 준수 여부를 점검할 수 있고
  - 개발 팀에 대한 멘토링과 코칭 기회를 포착할 수도 있다.

아키텍처적 사고하려면,

✓ 시스템의 전체 구조를 이해해야 한다.

✓ 비즈니스 관점을 파악하고 이를 아키텍처 특정으로 전환할 수 있어야 한다.

✓ 시스템을 논리적 컴포넌트의 관점에서 바라보는 안목도 필요하다.

기초 - 모듈성

모듈성(modularity)

"모듈 개념은 소프트웨어 아키텍트에서 보편적으로 쓰이지만, 사실 정의하기가 상당히 까다롭다"

- ✓ 누구나 인정하는 **한 가지 정의는 없다**.
- ✓ 다만 주어진 개발 플랫폼에서 **모듈성이 구현되는 다양한 형태를 이해하는 것이 필요**하다.
- ✓ 아키텍처 분석 도구들이 모듈성 및 관련 개념들에 의존하기 때문이다.
- ✓ 모듈성은 하나의 조직화 원칙이다.
- ✓ 시스템의 **구조적 건전성을 보장하려면** 아키텍트가 **지속적으로 에너지를 투입해야** 한다.
- ✓ **구조적 건정성이 저절로 보장되는 일은 없다**.
- ✓ **'지속 가능한' 코드베이스**를 위해서는 **좋은 모듈성이 제공하는 질서와 일관성**이 꼭 필요하다.

모듈성 vs 세분도(granularity)

모듈성은 시스템을 **더 작은 조각으로 나누는** 것에 관한 것

세분도는 그런 **조각들의 크기**에 관한 것 → 즉 시스템의 특정 부분이 얼마나 커야 하는가의 문제

유지보수 하기 어려운 아키텍처 안티패턴을 피하기 위해서는 **세분도** 및 컴포넌트/서비스 **결합 수준**에 항상 주의를 기울여야 한다.

모듈성의 정의

"서로 연관된 코드를 **논리적으로 묶은 것**을 통칭하는 일반적인 용어. 예) 클래스, 함수 등"

객체 지향 언어 → **클래스**

구조적/함수형 언어 → **함수**

- 자바의 **package**
- .NET의 **namespace**

✓ 모듈성은 물리적 분리를 의미하지 않는다. **논리적 분리를 의미**한다.

✓ 프로그래밍 언어 마다 **자신만의 방식으로 모듈이라는 개념을 지원**한다.

✓ 각자 고유한 범위 규칙과 특이성(quirk)을 가지고 있다.

- 예) 자바 패키지는 클래스 파일들의 디렉토리 구조와 일치해야 한다.

모듈성 측정

"모듈성의 이해를 위한 도구로 **응집**, **결합**, **동변성**이란 언어 독립적 지표를 만들었다"

응집도(cohesion)

하나의 모듈을 구성하는 요소들이 정말로 **그 모듈 안에 들어 있어야 하는가**에 관한 것이다.

결합도(coupling)

한 모듈(클래스, 컴포넌트)이 다른 모듈에 **얼마나 강하게 의존하는지를 나타내는 정도**를 의미한다.

동변성(connascence)

한 모듈을 변경할 때 다른 모듈도 반드시 함께 변경해야 하는 정도를 의미한다.

두 모듈이 **같은 규칙, 같은 가정, 같은 형태를 공유**해야만 정상 동작한다면 그 둘은 동변 관계(connascence)에 있다.

모듈성 측정 – 응집도

- 모듈을 구성하는 요소들이 서로 얼마나 관련이 있는지를 측정하는 지표다.
- 응집 관점에서 **이상적인 모듈**은 **필요한 모든 요소가 한데 패키징된 모듈**이다.

"**응집력이 있는 모듈을 나누려**고 해 봤자
결합도가 증가하고 **가독성이 떨어질** 뿐이다"
– 래리 콘스탄틴(Larry Constantine) -

응집도 유형

최선

기능적 응집 모듈의 모든 요소가 서로 관련되어 있음. **모듈이 작동하는 데 필요한 모든 요소가 모듈에 들어 있다.**

순차적 응집 두 모듈이 데이터를 매개로 상호작용한다. **한 모듈이 데이터를 출력하면 다른 모듈이 그것을 입력으로 삼는다.**

통신적 응집 두 모듈이 통신 체인을 형성해서 각자 정보를 처리하거나 어떤 출력에 기여한다.
한 모듈이 데이터베이스에 레코드를 추가한다. 다른 모듈은 그 정보를 바탕으로 이메일을 생성한다.

절차적 응집 두 모듈이 코드를 반드시 **특정한 순서로 실행**한다.

시간적 응집 모듈들이 시간적 의존성에 기반해서 연관된다. **시스템 시작 시에 초기화에 참여하는 모듈, 그러나 서로 관계는 없다.**

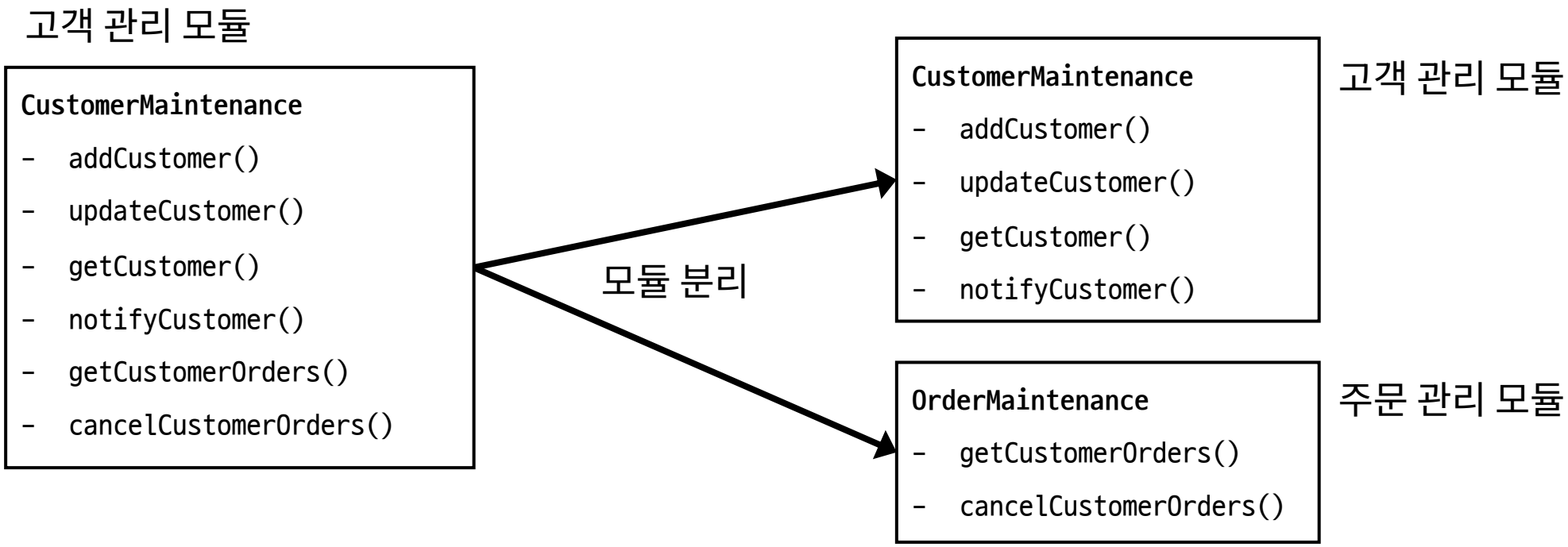
논리적 응집 데이터에 대해 논리적으로 관련되어 있지만 기능적으로는 그렇지 않다. **자바 StringUtils 패키지의 클래스들.**

우발적 응집 모듈의 요소들이 서로 관련이 없고, **그저 같은 소스 파일에 모여 있을 뿐**이다. 가장 부정적인 형태의 응집이다.

최악

모듈성

응집도와 모듈



어느 것이 **올바른 구조**일까? 언제나 그렇듯이 **답**은 "**상황에 따라 다르다!**"

- OrderMaintenance 모듈의 연산이 고객 주문 조회와 취소 둘뿐인가? 모듈이 너무 작다. **두 모듈을 통합하는 것이 합리적**일 수 있다.
- CustomerMaintenance 모듈이 지금보다 더 커질 것으로 예상되는가? 그렇다면 **두 모듈로 분리하는 것이 합리적**일 수 있다.
- OrderMaintenance 모듈에서 Customer 정보를 많이 사용하는가? 두 모듈로 나누면 **결합도가 증가하고 가독성이 떨어질 뿐**이다.

응집도 지표 – 응집 결여도(Lack of Cohesion)

LCOM 지표(Lack of Cohesion in Method; 메서드 응집 결여도)

- 치댐버와 케메러(Chidamber and Kemerer)가 만든 **모듈의 응집도를 측정하는 지표**다.
- 객체지향 설계에서 **클래스의 응집도가 얼마나 낮은지**를 수치로 나타낸다.
- 클래스 내 **메서드들**이 **인스턴스 변수를 얼마나 공유**하는지를 측정한다.
- LCOM은 여러 버전(LCOM1 ~ LCOM5)이 존재한다.

$$LCOM1 = \begin{cases} P - Q, \text{만일 } P > Q \text{이면} \\ 0, \text{그렇지 않으면} \end{cases}$$

(값이 클수록 나쁨)

P : 인스턴스 변수를 **공유하지 않는 메서드 쌍**의 개수

Q : 인스턴스 변수를 **공유하는 메서드 쌍**의 개수

$LCOM = 0$: 높은 응집도(좋음✅)

$LCOM > 0$: 낮은 응집도(나쁨❌)

필드를 공유하지 않는 메서드 쌍이 많을 수록 응집도가 낮다. 클래스를 분리해야 하는 신호다.

응집도 지표 – LCOM 계산 예

```
public class Rectangle {
  private int width; // 인스턴스 변수
  private int height; // 인스턴스 변수

  public int getArea() { // M1
    return width * height;
  }
  public int getPerimeter() { // M2
    return 2 * (width + height);
  }
  public void scale(int factor) { // M3
    width *= factor;
    height *= factor;
  }
}
```

메서드 쌍	공유 필드	공유 여부
(M1, M2)	width, height	공유 **있음** → Q
(M1, M3)	width, height	공유 **있음** → Q
(M2, M3)	width, height	공유 **있음** → Q

- P = 0 (공유 안 하는 메서드 쌍)
- Q = 3 (공유하는 메서드 쌍)
- **LCOM = 0** (P < Q 이므로) ➔ **완벽한 응집도!**

✓ 이 지표가 찾을 수 있는 것은 **구조적** 응집도 결여뿐이다.

✓ 특정 요소들이 **논리적**으로 잘 맞는지는 이 지표로 **알 수 없다**.

모듈성 측정 – 결합도

한 모듈이 다른 모듈에 얼마나 강하게 의존하는지를 나타내는 정도

의존 정보가 많을수록 → 결합도 높음

의존 정보가 적을수록 → 결합도 낮음

결합도가 높을 때 발생하는 문제:

- **변경에 취약** 예) 한 클래스를 수정하면, 다른 여러 클래스를 수정해야 한다.
- **테스트 어려움** 예) 단위 테스트가 힘들다.
- **재사용성 저하**
- **아키텍처 분리 실패** 예) 모놀리스에서 마이크로서비스 전환이 어렵다.

결합도 분석 도구

- 코드베이스의 결합도를 분석할 때 **그래프 이론을 활용**할 수 있다.
- 메서드 호출과 리턴은 호출 그래프를 형성한다. 이 **호출 그래프를 수학적으로 분석**하는 것이다.

결합도 지표

『Structured Design』에서 정의한 지표

구심 결합도(afferent coupling; 유입 결합도)

- "다른 패키지들이 **나(이 패키지)를 얼마나 의존**하는가?"
- 해당 코드 요소로 **들어오는**(incoming) 의존성의 개수를 측정한다.
  (컴포넌트, 클래스, 함수 등)

원심 결합도(efferent coupling; 유출 결합도)

- "내가 **다른 패키지들을 얼마나 의존**하는가?"
- 다른 코드 요소로 **나가는**(outgoing) 의존성의 개수를 측정한다.

* 『Structured Design』
Edward Yourdon과 Larry Constantine의 저서이다. **구조적 설계 방법론을 체계화한 고전** 중의 고전이다.
객체지향 이전 시대에 **모듈성**, **응집도**, **결합도**를 설계의 중심으로 끌어올린 책이다.

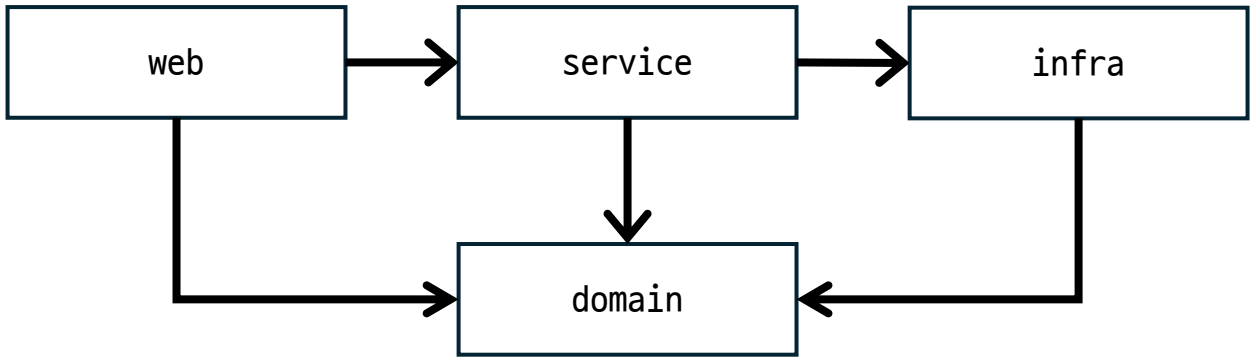
구심/원심 결합도와 패키지 의존 관계

```
package domain;

public class Order {
  private final String id;
  public Order(String id) {…}
  public String id() {…}
}
```

```
package infra;

import domain.Order;

public class OrderRepository {
    public void save(Order order) {…}
}
```

```
package service;

import domain.Order;
import infra.OrderRepository;

public class OrderService {
  OrderRepository repository = …;
  public Order create(String id) {…}
}
```

```
package web;

import domain.Order;
import service.OrderService;

public class OrderController {
  OrderService service = …;
  public String createOrder(String id) {…}
}
```



패키지 의존 관계

패키지의 구심/원심 결합도 계산

패키지	의존성	결합도
domain	domain ← infra domain ← service domain ← web	구심 = 3 원심 = 0
infra	infra ← service Infra → domain	구심 = 1 원심 = 1
service	service ← web service → infra service → domain	구심 = 1 원심 = 2
web	web → service web → domain	구심 = 0 원심 = 2

구심 결합도가 높은 패키지

- **바꾸기 어렵다. 바꾸면 영향 범위가 크다.**
- 더 안정적이고 테스트가 탄탄해야 한다.

  예) domain, core, api 등
- **핵심 패키지**일수록 **구심 결합도는 높이고** 원심 결합도는 낮춰야 한다.

원심 결합도가 높은 패키지

- **외부 변화에 취약**하다.
- 그래서 보통 **인터페이스로 차단**하거나 **어댑터 계층에서 몰아서 관리**하다.

  예) service, web 등
- **변화가 많은 패키지**는 **원심 결합도가 높다**. 단 의존 관계가 뒤엉키지 않게 **계층 방향을 단방향으로 고정**해야 한다.

  예) web → service → infra

모듈성 측정 – 핵심 지표들

추상도(abstractness) Robert C. Martin 이 제안한 지표

- 추상적인 요소와 구체적인 요소의 비이다.
- 코드베이스가 어느 정도나 **추상적/구체적**인지를 나타낸다.

추상적인 요소: **추상 클래스**, **인터페이스**
구체적인 요소: **구현 코드**

추상도 계산식: $$A = \frac{\sum m^a}{\sum m^c + \sum m^a}$$

m^a : 모듈 안의 추상 타입(추상적 요소)의 개수이다.

m^c : 모듈 안의 구체 타입(구체적 요소)의 개수이다.

A : 모듈 안의 전체 타입에서 추상 타입이 차지하는 비율을 나타낸 값이다. $(0 \leq A \leq 1)$

모듈성 측정 – 핵심 지표들

추상도와 결합도의 관계

⬆ ⬆

“구심 결합도가 높을수록 추상도가 높아야 한다.”

(많이 의존 받는 패키지일수록 추상도가 높아야 변경에 강하다)

왜냐하면:

- ✓ 많은 곳이 의존하는데
- ✓ 구체 클래스만 있으면
- ✓ 변경 시 영향이 폭발하기 때문이다.

잘못된 패턴:

구심결합도 ↑, 추상도 ↓ (고통의 영역)

- 많이 의존 받는데 구현만 가득
- 변경이 거의 불가능

구심결합도 ↓, 추상도 ↑ (무용의 영역)

- 아무도 안 쓰는데 추상만 가득
- 쓸모없는 인터페이스

모듈성 측정 – 핵심 지표들

불안정도(instability) Robert C. Martin 이 제안한 지표

- 패키지가 **외부 변화에 얼마나 취약한가**를 나타내는 지표이다.
- 코드베이스의 변동성(volatility; 또는 휘발성)을 결정한다.

(이 코드가 변경 압력에 얼마나 노출된 구조인가)

불안정도 계산식: $$I = \frac{C^e}{C^e + C^a}$$

- $I = 0$ → 매우 안정적
  - 많이 의존 받고, 거의 의존하지 않음
- $I = 1$ → 매우 불안정
  - 의존은 많이 하지만, 의존 받지 않음

C^e: 원심 결합도(나가는 결합도)이다.

C^a : 구심 결합도(들어오는 결합도)이다.

I : 전체 결합도에서 원심 결합도가 차지하는 비율이다.

"불안정도가 높은 코드베이스는 결합도가 높아서 변경 시 고장 나기 쉽다"

- 작업을 위임하기 위해 다른 클래스의 메서드를 너무 많이 호출하는 클래스는 불안정도 높으며,
- 만일 호출하는 메서드 중 하나 이상이 변경되면 호출하는 클래스 역시 잘못될 가능성이 높다.

모듈성 측정 – 주 시퀀스로부터의 거리

- 아키텍처 구조에 대한 전일적(holistic) 지표다.
- 즉, 개별 클래스나 모듈 하나의 품질이 아니라 **시스템 전체**의 **구조적 균형과 방향성을 평가**하는 지표다.
- **추상도**와 **불안정도**에서 **파생된 지표**다.

주 시퀀스로부터의 거리: $D = |A + I - 1|$
(Distance from the Main Sequence)

A: 추상도 $(0 \leq A \leq 1)$

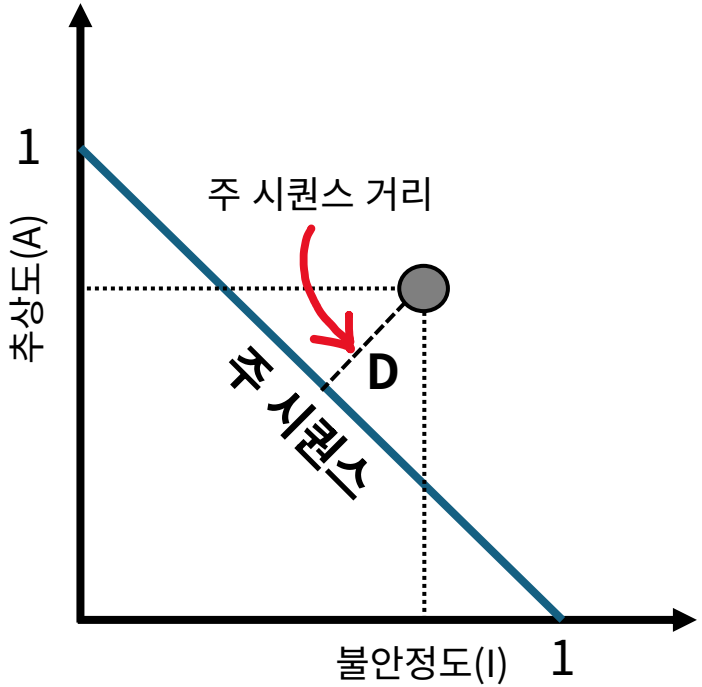
I: 불안정도 $(0 \leq I \leq 1)$

D: 주어진 클래스가 추상성과 불안정성이 이상적으로 균형을 이루는 선과 얼마나 떨어져 있는지를 표현

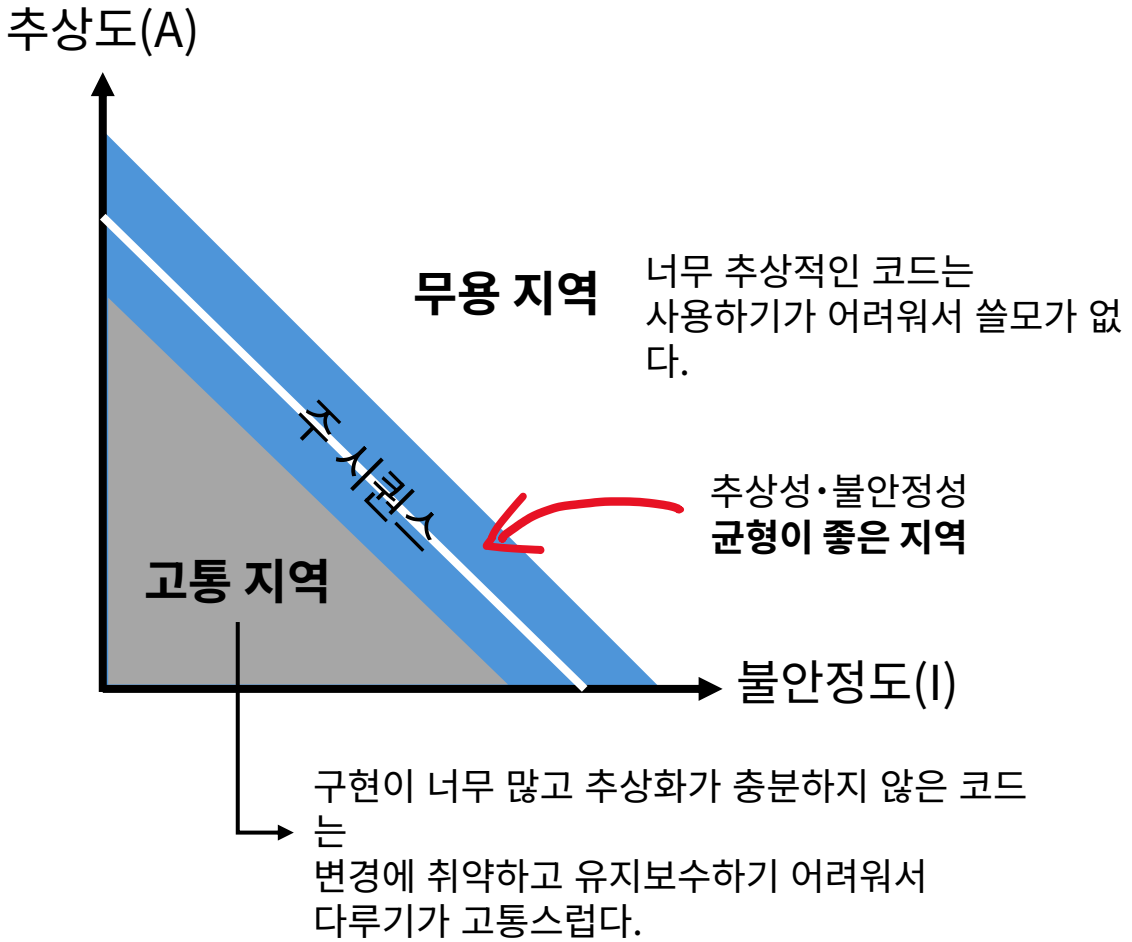
“클래스가 이상적인 균형선에 가까이 있다는 것은
추상성과 불안정성이라는 서로 경쟁하는 관심사가 잘 혼합되어 있다는 뜻이다”

모듈성 측정 – 주 시퀀스로부터의 거리 계산

<특정 클래스에 대한 정규화된 주 시퀀스 거리>



<무용 지역과 고통 지역>



모듈성 측정 – 동변성(connascence)

- Meilir Page-Jones는 저서 『What Every Programmer Should Know about Object-Oriented Design』(Dorset House, 1996)은 객체지향 설계에서 다양한 유형의 결합을 설명하기 위해 '**동변성**'이라는 개념을 제시했다.
- 동변성이란, **한 요소를 변경할 때 다른 요소도 함께 변경해야만 시스템이 올바르게 동작**하는 그런 **의존 관계를 뜻하는 말**이다.

"두 컴포넌트가 동변적(connascent)이라는 것은,
한 컴포넌트가 변했을 때 **시스템의 전체적인 정확성을 유지**하려면
다른 컴포넌트도 변해야 한다는 뜻이다"

모듈성 측정 – 동변성 유형/정적 동변성

정적 동변성(static connascence)

- 소스 코드 수준의 결합을 의미한다.
- 코드를 작성하거나 컴파일하는 단계에서 이미 드러나는 동변성이다.

종류

- 이름 동변성(connascence of name)
- 타입 동변성 (connascence of type)
- 의미 동변성 (connascence of meaning)
- 위치 동변성 (connascence of position)
- 알고리즘 동변성 (connascence of algorithm)

모듈성 측정 – 동변성 유형/정적 동변성

1) 이름 동변성

두 코드 요소가 **같은 이름을 정확히 공유해야만** 올바르게 동작하는 의존 관계를 말한다.

→ "이 이름을 바꾸면, 다른 곳도 반드시 같이 바꿔야 한다"

```
// A.java
public class User {
    public String userName;
}
```

```
// B.java
public class UserPrinter {
    public void print(User user) {
        System.out.println(user.userName);
    }
}
```

User 클래스의 userName 에 동변

- userName → name 으로 바꾸면?
  - **User와 UserPrinter를 동시에 수정해야 됨**
- 이 둘은 이름(userName)에 대한 동변성을 가짐

모듈성 측정 – 동변성 유형/정적 동변성

2) 타입 동변성

두 코드 요소가 **같은 타입을 반드시 공유해야만** 올바르게 동작하는 의존 관계를 말한다.

→ “이 타입을 바꾸면, 다른 쪽도 반드시 같이 바꿔야 한다”

```
// Producer
public class UserService {
    public void setAge(int age) {
        // ...
    }
}
```

```
// Consumer
userService.setAge(20);
```

User 클래스의 userName 에 동변

- int → String 으로 바꾸면?
  - **호출부 전부 수정해야 함**
- 호출자와 피호출자는 타입(int)에 동변

모듈성 측정 – 동변성 유형/정적 동변성

3) 의미 동변성

두 코드 요소가 **같은 값(또는 구조)에 대해 같은 의미를 암묵적으로 공유해야만** 올바르게 동작하는 의존 관계를 말한다.

→ "이 값이 무엇을 의미하는지, 양쪽이 약속하고 있다"

```
// OrderService
public void process(int status) {
    if (status == 1) {
        ship();
    }
}
```

```
// 호출부
orderService.process(1);
```

값 1의 의미를 호출자와 피호출자가 암묵적으로 공유

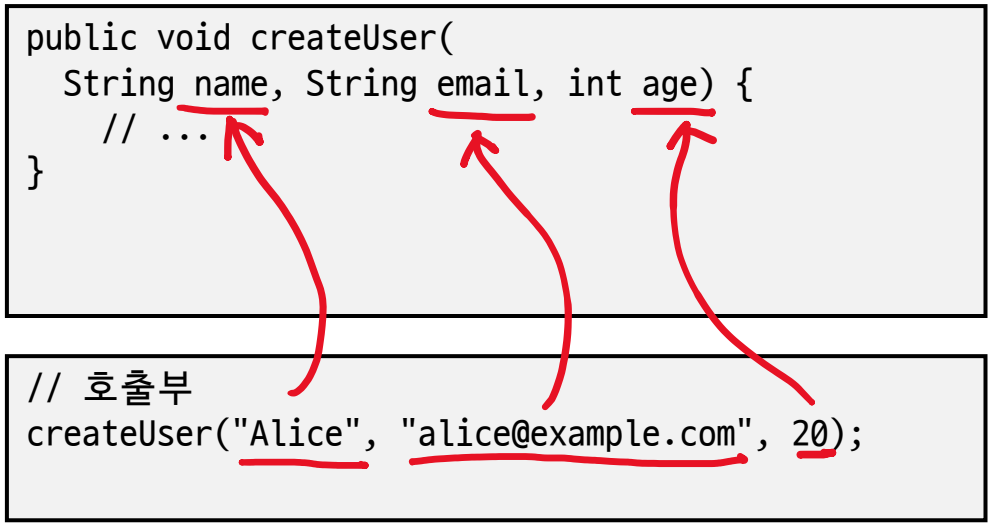
- 현재 값 1의 의미가 '배송 가능'을 의미함
- 만약 값 1의 의미를 '배송 불가'로 바꾼다면?
  - **호출부 전부 수정해야 함**

모듈성 측정 – 동변성 유형/정적 동변성

4) 위치 동변성

여러 값의 **순서**를 **호출자와 피호출자가 정확히 동일하게 알고 있어야**만 동작하는 의존 관계를 말한다.

→ “이 값들의 순서를 바꾸면, 여러 곳을 동시에 고쳐야 한다”



호출자의 파라미터의 순서를 정확히 알아야 함

- 현재 파라미터가 name, email, age 순서로 되어 있음
- 만약 파라미터의 순서를 바꾼다면?
  - **호출부 전부 수정해야 함**

모듈성 측정 – 동변성 유형

5) 알고리즘 동변성

둘 이상의 모듈이 **같은 계산 규칙/비즈니스 로직**을 각자 구현하고 있어서 그 규칙이 바뀌면 모두 함께 바꿔야 하는 의존 관계를 말한다.

→ "이 계산 규칙을 바꾸면, 서로 떨어진 여러 곳을 동시에 고쳐야 한다"

```
// OrderService
public int calculateTotal(int price, int quantity) {
    int total = price * quantity;
    if (quantity >= 10) { // 할인규칙
        total *= 0.9;
    }
    return total;
}
```

```
// InvoiceService
public int calculateInvoice(int price, int quantity) {
    int total = price * quantity;
    if (quantity >= 10) { // 할인 규칙
        total *= 0.9;
    }
    return total;
}
```

10개 이상 10% 할인 규칙을 두 군데에서 공유

- 만약 할인 정책을 바꾼다면?
  - **두 군데 모두 수정해야 함**

모듈성 측정 – 동변성 유형/동적 동변성

동적 동변성(dynamic connascence)

- 코드를 봐서는 잘 드러나지 않지만, **실행 시점에서 특정 규칙을 함께 지켜야만 정상 동작**하는 동변성이다.
- Meilir Page-Jones 가 정리한 동변성 분류에서 가장 위험하고 발견하기 어려운 형태로 강조되고 있다.
- “이 규칙을 어기면, **컴파일은 되는데 실행 중에 깨지는 상황**이다”

종류

- 실행 동변성(connascence of execution)
- 타이밍 동변성 (connascence of timing)
- 값 동변성 (connascence of values)
- 신원 동변성 (connascence of identity)

모듈성 측정 – 동변성 유형/동적 동변성

1) 실행 동변성

여러 코드 요소가 **같은 실행 규칙(호출 순서·실행 ·실행 조건)을 정확히 공유해야만** 올바르게 동작하는 의존 관계를 말한다.

→ "이 코드들이 언제, 어떤 순서로 실행되어야 하는지 호출자가 기억해야 한다"

```
class FileWriterSession {
    public void open() {…} // 파일 오픈

    public void write(String text) {
        // open()이 먼저 호출되어야 함
    }
    public void close() {…} // 자원 해제
}
```

```
// 호출부
FileWriterSession s = new FileWriterSession();
s.write("hello"); // 컴파일 OK, 런타임 오류 가능
s.open();
s.close();
```

open() → write() → close() 호출 순서를 알아야 함

- 만약 순서를 어긴다면?
  - **런타임 오류 발생**

모듈성 측정 – 동변성 유형/동적 동변성

2) 타이밍 동변성

여러 코드 요소가 **정확한 시간 간격·시점·지연을 공유해야만** 올바르게 동작하는 의존 관계를 말한다.

즉, 컴파일은 되지만 **언제 실행되느냐가 어긋나면 런타임에서 깨지는** 의존이다.

→ "이 코드가 올바르게 동작하려면 특정 시간에 맞춰 실행돼야 한다"

```
cache.put(key, value);
Thread.sleep(100);      // 100ms 뒤에만 의미 있음
String v = cache.get(key);
```

sleep()에 의미가 있는 코드

- 100ms 를 양쪽이 암묵적으로 공유한다.
- 만약 100ms 시간보다 적다면?
  - **올바른 값을 리턴하지 않음**
- 머신/부하/GC에 따라 100ms는 충분하지 않을 수 있음

```
asyncService.send(msg);
Thread.sleep(500);      // 이 정도면 처리됐겠지?
assertTrue(asyncService.isProcessed(msg));
```

비동기 처리 완료를 '기다리는 시간'으로 추정

- 처리 완료를 시간으로 추정
- 느린 환경에서는 **실패**, 빠른 환경에서는 우연히 **성공**

모듈성 측정 – 동변성 유형/동적 동변성

3) 값 동변성

여러 값이 **서로 의존**하며, 반드시 **함께 변경되어야 하는** 의존 관계를 말한다.

→ “여러 값들이 하나의 의미를 나눠서 표현하고 있고, 그 의미를 유지하려면 값들이 반드시 함께 변경되어야 한다”

```
// 사각형을 꼭지점 4개로 표현
Point p1 = new Point(0, 0);   // 좌하
Point p2 = new Point(10, 0);  // 우하
Point p3 = new Point(10, 5);  // 우상
Point p4 = new Point(0, 5);   // 좌상
```

```
p2.x = 12;  // 꼭지점 값 변경
p3.x = 12;  // 이 꼭지점 값도 변경해야 한다.
```

각 꼭지점의 값은 다음과 같은 암묵적인 규칙이 있다

- `p1.y == p2.y`
- `p3.y == p4.y`
- `p1.x == p4.x`
- `p2.x == p3.x`
- 선은 서로 직각
- 변은 서로 평행
- 만약 한 꼭지점의 값을 바꾼다면?
  - 의존 관계의 값도 바꿔야 한다.
  - `p2.x == p3.x` 이므로,
    - **`p3.x = 12`**

모듈성 측정 – 동변성 유형/동적 동변성

4) 신원 동변성

여러 **다른 코드 요소가 같은 엔티티를 가리킨다**고 **런타임에 동일하게 인식해야만** 올바르게 동작하는 의존 관계이다.

→ “이 둘이 같은 것이라고 가정하지 않으면 시스템이 깨진다”

```
// Service A
Order order = new Order("ORD-123");
// … 주문 완료(저장)
publish(new OrderPaidEvent("ORD-123"));
```

```
// Service B
public void handle(OrderPaidEvent e) {
    Order order = repository.findById(e.orderId());
    ship(order);
}
```

ID 값에 의존하는 신원 동변성의 예이다

- 두 서비스는 “**ORD-123이 같은 주문을 가리킨다**”는 가정을 실행 중에 공유한다.
- 만약 ‘ORD-123’이 다음과 같다면?
  - 아직 저장되지 않았다
  - 이미 다른 상태로 변경되었다
  - 다른 고객의 주문과 동일한 ID일 수도 있다
  - **이런 경우에 동작이 깨진다.**

모듈성 측정 – 동변성의 속성들

- 동변성은 **아키텍트와 개발자를 위한 분석의 틀**(프레임워크)이다.
- **이 틀을 현명하게 사용하는데 도움이 되는 속성**들이 있다.

강도(strength) = 세기

개발자가 시스템의 결합을 리팩토링하기가 얼마나 쉬운지를 나타낸다.

지역성(locality) = 국소성

코드베이스에서 모듈들이 서로 얼마나 가까이 있는지를 측정한다.

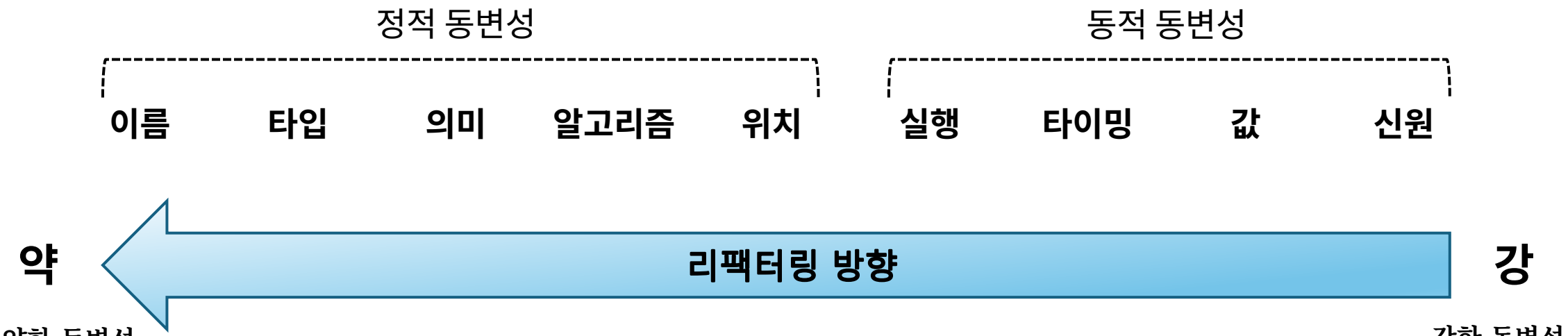
정도(degree)

특정 모듈에서 클래스를 변경했을 때 그것이 시스템에 미치는 영향의 크기와 연관된다.

모듈성 측정 – 동변성의 속성들/강도

강도

- 동변성의 유형에 따라 동변성의 세기가 다르다. 동적 동변성보다 **정적 동변성을 선호**해야 한다.
- **정적 동변성**은 개발자가 **간단한 소스 코드 분석으로 파악할 수** 있으며, 현대적인 도구들을 사용하여 **수월하게 개선**할 수 있다.



약한 동변성

변경 시 바로 드러나고, 도구로 관리 가능

강한 동변성

조금만 어겨도 시스템이 쉽게 망가짐

→ 발견이 어렵고, 영향 큼

모듈성

모듈성 측정 – 동변성의 속성들/지역성

지역성

- **근접한 코드(같은 모듈 안의 코드)**는 멀리 떨어진 코드(별도 모듈이나 코드베이스에 있는 코드)보다 **동변성이 더 크고 다양**하다.
- 즉, **같은 유형의 동변성이라도** 컴포넌트들이 **멀리 떨어져 있을 때는 바람직하지 않은 결합**을 나타내지만, **가까이 있을 때는 문제가 덜하다**.
- 동변성의 세기와 지역성은 함께 고려하는 것이 바람직하다. **강한 유형의 동변성이라도**, **같은 모듈 안에 있을 때가** 다른 모듈 안에 있을 때보다 **'코드 악취(code smell)'가 덜하다**.

"동변성을 제거할 수 없다면,
반드시 **지역화**하라"

- 변경 비용 ⬇
- 실수 가능성 ⬇
- 이해 비용 ⬇

모듈성 측정 – 동변성의 속성들/지역성

같은 클래스 안에 지역화된 동변성 예:

```
class Rectangle {
    private int left;
    private int width;

    public void moveRightEdge(int newX) {
        this.width = newX - left; // 값 동변성
    }
}
```

- `left` 와 `width`는 값 동변성 관계이다.
- 같은 클래스 안, 같은 메서드에서 함께 변경된다.
- 외부에서 보이지 않는다.
- **지역성이 높음 → 안전**

여러 파일에 흩어진 동변성

```
// A.java
int DISCOUNT_THRESHOLD = 10;

// B.java
if (quantity >= 10) { ... }

// C.java
boolean eligible = count >= 10;
```

- `10` 이라는 규칙이 여러 위치에 분산되어 있다.
- 변경 시 모든 파일을 기억해야 한다.
- **지역성 낮음 → 위험**

모듈성 측정 – 동변성의 속성들/정도

정도

- 하나의 규칙(값·알고리즘·타이밍 등)이 바뀔 때 **동시에 수정해야 하는 코드 요소의 수**를 말한다.
- **변경에 영향을 받는 클래스가 많을수**록 동변성의 **정도가 높은 것**이다.
- 동변성의 **정도가 낮으면** 다른 클래스와 모듈을 **조금만 변경하면 된다**.

“설계의 현실적인 목표” →

다음 세가지를 동시에 관리하는 것.

- **세기**(strength): 얼마나 위험한가?
- **지역성**(locality): 얼마나 가까이 모여 있는가?
- **정도**(degree): 얼마나 많이 함께 바꿔야 하는가?

정도는 **변경 비용**을 **선형/기하급수**로 키우는 요인이다.

동변성을 이용한 모듈성 개선 가이드라인

페이지존스가 제시한 세 가지 가이드라인:

『What Every Programmer Should Know about Object-Oriented Design』 저서

- 시스템을 캡슐화된 요소들로 분해해서 전체적인 동변성을 최소화한다.
- 캡슐화의 경계에 걸쳐 있는 동변성을 찾아서 최소화한다.
- 캡슐화 경계 안의 동변성을 최대화한다.

짐 웨이리치가 제시한 규칙:

'Emerging Technologies for the Enterprise' 컨퍼런스에서 Jim Weirich가 강연

- 정도의 규칙(Rule of Degree): 강한 유형의 동변성을 더 약한 유형의 동변성으로 변환하라.
- 지역성의 규칙(Rule of Locality): 소프트웨어 요소들 사이의 거리가 멀수록 더 약한 유형의 동변성을 적용하라.

아키텍트가 동변성을 알아야 하는 이유

“동변성이라는 틀로 다양한 유형의 결합을 더 간결하고 정확하게 설명할 수 있다”

예:
- “메서드 선언 중간에 마법의 문자열 리터럴을 추가하지 말고, 상수로 추출하세요”
- “**의미 동변성**이 **이름 동변성**으로 변하도록 코드를 리팩토링 하세요”

참고 - 설계 패턴 사용

일반적인 문제에 대한 맥락과 해법을 **간단한 이름 하나로 깔끔하게 캡슐화**한다.

예:
- “서비스가 하나 필요한데, 그 서비스의 인스턴스가 딱 하나여야 해”
- “**싱글턴(Singleton)** 서비스가 필요해”

기초 – 아키텍처 특성의 정의

아키텍처 특성이란?

"아키텍트는 도메인을 정의하는 과정에 참여하여 도메인 기능성과는 직접 관련되지 않으면서도 **소프트웨어가 수행해야 하는 모든 것**을 정의, 식별, 분석해야 한다. **그 모든 것이 아키텍처 특성이다.**"

아키텍트의 핵심 역할:
- ✓ **아키텍처 특성** 분석
- ✓ 논리적 컴포넌트 설계

* **도메인**(domain) **또는 문제 도메인**(problem domain)
어떤 문제를 소프트웨어를 이용해서 해결하기로 결정한 회사가 제일 먼저 그 시스템에 대한 요구사항(requirement) 목록을 수집하는 것으로 시작한다. 이 요구사항들의 집합을 **문제 도메인**이라고 부른다. 줄여서 그냥 **도메인**이라고 부르기도 한다.

소프트웨어 솔루션

소프트웨어 솔루션 = **도메인 요구사항** + **아키텍처 특성(= 시스템의 역량)**

다른 용어

비기능적 요구사항(non-functional requirement)

비판 → '비기능적'인 것에 충분한 관심을 기울이도록 팀을 설득하는 것이 힘들다.

품질 속성(quality attribute)

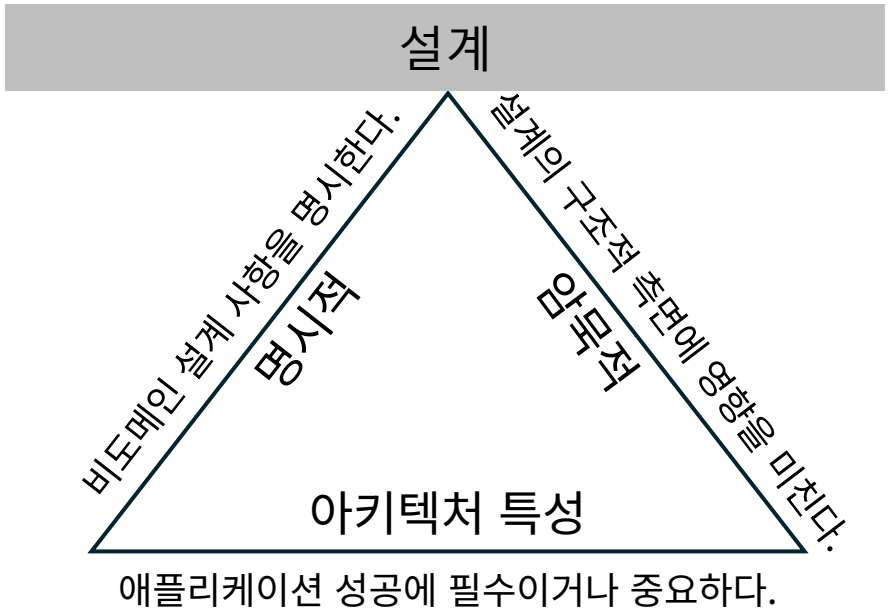
비판 → 설계보다는 사후 품질 평가를 의미하는 것 같기 때문에 관심을 집중시키기 어렵다.

아키텍처 특성의 정의

아키텍처 특성의 기준 – 아키텍처 결정을 정당화하는 핵심 근거

- 비도메인 설계 고려 사항을 명시해야 한다.
- 설계의 구조적 측면에 영향을 미쳐야 한다.
- 애플리케이션의 성공에 필수적이거나 중요해야 한다.

모두 만족해야 한다!



- 각 요소는 다른 요소들을 지탱한다
- 이 요소들은 함께 시스템 전체의 설계를 지탱한다.

아키텍처 특성의 정의

아키텍처 특성의 기준 – 명시적

1) 도메인과 무관한 설계 고려 사항을 명시한다

- 설계 요구사항
  - 애플리케이션이 **무엇**을 해야 하는지 명시한다.
- **아키텍처 특성**
  - 요구사항을 **어떻게** 구현할지, 어떤 사항을 **왜** 그렇게 결정하고 선택했는지를 명시한다.
  - 프로젝트가 성공하기 위한 **운영 및 설계의 기준**(criteria)이다.

예:
- 성능
- 확장성
- 가용성
- 보안
- 배포성
- …

✅ **명시하는 않는다면?**
- 팀마다 서로 다른 가정을 함
- 설계 판단의 기준이 사라짐
- 나중에 “왜 이렇게 만들었지” 라는 질문에 답을 못함

그래서 아키텍처 특성은 **문서·회의·결정 기록에 분명히 드러나야** 한다.

아키텍처 특성의 기준 – 암묵적

2) 설계의 구조적 측면에 영향을 미친다

- 아키텍처 특성을 서술하려는 주된 이유?
  - 그로부터 **중요한 설계 고려사항을 끌어내기 위함**이다.
- "구조를 바꾸지 않는다면 아키텍처 특성이 아니다"
  - 해당 특성을 만족시키기 위해서는 **컴포넌트 분리, 통신 방식, 배포 단위, 데이터 저장 방식**이 달라져야 한다.

예:

- **응답시간 100ms 이내** ➔ 캐시도입, 비동기 처리, CQRS 분리
- **독립 배포 가능** ➔ 모놀리스(❌) , 서비스 경계 명확화
- **고가용성** ➔ 단일 인스턴스 (❌), 이중화 / 무상태 설계

아키텍처 특성
(**명시적**으로 문서에 명시됨)

설계 구조 변경
(**암묵적**으로 설계에 반영됨)

아키텍처 특성의 정의

아키텍처 특성의 기준 – 필수적

3) 애플리케이션 성공에 필수이거나 중요해야 한다

- 시스템이 지원하는 **아키텍처 특성이 늘어날수록 설계가 더 복잡**해진다.
- 이것이 아키텍트가 아키텍처 특성을 **가능한 한 적게**(많이가 아니라) **선택하려고 노력해야** 하는 이유다.
- "있으면 좋은것"이 아니라, "**없으면 실패하는 것**"이어야 한다.

아키텍처 특성 예:

- 피크 시간대 트래픽을 감당하지 못하면 서비스가 죽는다
- 보안 사고가 나면 사업이 종료된다
- 배포가 느려서 시장에서 밀린다

아키텍처 특성이 **아닌 예:**

- 코드가 예쁘다
- 최신 기술을 쓴다
- 개발자가 재미있다

중요한 아키텍처 특성들

- **아키텍처 특성들은** 낮은 수준의 코드 특성(모듈성)부터 정교한 운영상 관심사(확장성, 탄력성 등)까지, 복잡성의 **광범위한 스펙트럼에 걸쳐 존재**한다.
- **소프트웨어 생태계가 매우 빠르게 변화**하기 때문에 새로운 개념, 용어, 측정, 검증이 지속적으로 등장하며, 따라서 **아키텍처 특성을 새로이 정의할 기회가 계속 생긴다**.

"정량화 하기 어렵지만, 분석을 위해 몇 가지 범주로 살펴 보자"

아키텍처 특성 분류:

- ✓ 운영 아키텍처 특성
- ✓ 구조적 아키텍처 특성
- ✓ 클라우드 특성
- ✓ 횡단적 아키텍처 특성

중요한 아키텍처 특성들 – 운영 아키텍처 특성

가용성(availability)	시스템이 얼마나 오랫동안 사용 가능한 상태를 유지하는가에 관한 특성이다. 예를 들어, 24/7(매일 24시간, 연중 무휴)이 요구된다면, 어떤 장애가 발생해도 시스템을 신속하게 재시동하는 조치가 필요.
연속성(continuity)	시스템의 재해 복구 역량.
성능(performance)	시스템이 얼마나 잘 수행되는지를 나타낸다. 이를 측정하는 방법에는 스트레스 테스트, 피크 분석, 기능의 사용 빈도 분석, 응답 시간 등이 있다.
복구성(recoverability)	비즈니스 연속성 요구사항. 재해 발생 시 시스템이 얼마나 빨리 다시 온라인 상태가 되어야 하는가? 여기에는 백업 전략과 하드웨어 복제(다중화) 요구사항이 포함된다.
신뢰성(reliability) **안정성**(safety)	시스템이 안전장치를 갖춰야 하는지, 또는 인명에 영향을 미칠 정도로 미션 크리티컬한지의 여부. 시스템이 실패할 경우 회사에 큰 금전적 손실을 초래할 것인가? 이 측면은 이분법적인 양자택일이기보다는 스펙트럼에 해당할 때가 많다.
견고성(robustness)	실행 중에 오류와 경계 조건을 처리하는 시스템의 역량. 예를 들어, 인터넷 연결이 끊기거나 전원이 고장 났을 때 시스템이 어떻게 대응하는가?
확장성(scalability)	사용자가 요청 수가 증가할 때 시스템의 수행 및 작동 능력.

중요한 아키텍처 특성들 – 구조적 아키텍처 특성

설정성(configurability)	최종 사용자가 인터페이스를 통해 소프트웨어 구성의 측면을 얼마나 쉽게 변경할 수 있는가에 관한 특성
확장 능력(extensibility)	아키텍처가 기존 기능성을 확장하는 변경 사항을 얼마나 잘 수용하는지를 나타낸다.
설치성(intallability)	모든 필수 플랫폼에 시스템을 설치하는 것이 얼마나 쉬운지에 관한 특성이다.
활용성(leverageability) **재사용**(reuse)	시스템의 공통 컴포넌트를 여러 제품에서 어느 정도나 활용할 수 있는지를 나타낸다.
현지화(localization)	데이터 필드의 입력/조회 화면에서 여러 언어를 지원하는 문제에 관한 것이다.
유지보수성 (maintainability)	변경 사항을 적용하고 시스템을 개선하는 것이 얼마나 쉬운지를 나타낸다.
이식성(portability)	시스템을 둘 이상의 플랫폼(오라클, SAP DB 등)에서 실행할 수 있는 능력.
업그레이드성 (upgradeability)	서버와 클라이언트에서 새 버전으로 업그레이드하는 것이 얼마나 쉽고 빠른지를 나타낸다.

아키텍처 특성의 정의

중요한 아키텍처 특성들 – 클라우드 특성

온디맨드 확장성(on-demand scalability)	수요에 따라 자원을 동적으로 확장하는 클라우드 제공업체의 능력
온디맨드 탄력성	리소스 수요가 급증할 때 클라우드 제공업체의 유연성. 확장성과 유사함.
존 기반 가용성	컴퓨팅 존(zone)으로 자원을 분리해서 좀 더 복원력이 강한 시스템을 만드는 클라우드 제공업체의 능력.
지역 기반 개인정보보호 및 보안	다양한 국가와 지역(region)에 데이터를 합법적으로 저장하는 것과 관련한 클라우드 제공업체의 능력. 자국민의 데이터 저장 위치를 규제하는(그리고 종종 해당 지역 외부에 데이터를 저장하지 못하게 하는) 법률을 가진 국가가 많다.

중요한 아키텍처 특성들 – 횡단적 아키텍처 특성

"여러 범주에 걸쳐 있거나 특정한 하나의 범주로 분류하기 어려운 아키텍처 특성"

접근성(accessibility)	색맹이나 청각 장애와 같은 장애를 가진 사용자를 비롯해 모든 사용자가 시스템에 얼마나 쉽게 접근할 수 있는지를 나타낸다.
보관성(archivability)	지정된 기간 후 데이터를 보관하거나 삭제하는 것과 관련된 시스템의 제약조건이다.
인증(authentication)	사용자가 본인이 주장하는 사람이 맞는지 확인하기 위한 보안 요구사항이다.
권한부여(authorization)	사용자가 애플리케이션 내에서 특정 기능에만 접근할 수 있도록 하는 보안 요구사항이다.
법적 요건	EU의 GDPR 같은 데이터 보호법이나 미국의 사베인스-옥슬리법(Sarbanes-Oxley) 같은 재무 기록법처럼 시스템의 운영에 중요한 영향을 미치는 법적 제약조건이다.
개인정보보호(privacy)	내부 회사 직원(개발자, DBA 등)으로부터 트랜잭션을 암호화하고 숨기는 시스템의 능력이다.
보안	데이터베이스나 내부 시스템 간 네트워크 통신에서의 암호화, 원격 사용자 접근을 위한 인증, 기타 보안 조치에 관한 규칙과 제약조건이다.
지원성(supportability)	애플리케이션이 필요로 하는 기술 지원 수준이다. 시스템의 오류를 디버깅하기 위해 로깅 및 기타 기능이 어느 정도까지 필요한지를 의미한다.
사용성(usability)/ 성취성(achievability)	사용자가 애플리케이션/솔루션으로 목표를 성취하기 위해 필요한 교육 수준이다.

아키텍처 특성의 정의

중요한 아키텍처 특성들 – ISO가 정리한 주요 특성

성능 효율성

- 알려진 조건에서 사용된 자원의 양을 기준으로 성능을 측정한 것이다.
- **시간 행동 방식**(time behavior) – 응답, 처리 시간 또는 처리량 비율의 측정.
- **자원 활용**(resource utilization) – 사용된 자원의 양과 유형.
- **용량**(capacity) – 설정된 최대 한계를 초과하는 정도.

호환성

- 어떠한 제품이나 시스템, 컴포넌트가 다른 어떤 제품, 시스템, 컴포넌트와 정보를 어느 정도나 교환할 수 있는가를 나타낸다.
- 그런 요소들이 동일한 하드웨어 또는 소프트웨어 환경을 공유하면서 필요한 기능을 어느 정도나 잘 수행할 수 있는가를 나타낸다.
- **공존성**(coexistence) – 다른 제품과 공통 환경 및 자원을 공유하면서 필요한 기능을 효율적으로 수행할 수 있음.
- **상호운용성** – 둘 이상의 시스템이 정보를 교환하고 활용할 수 있는 정도.

사용성

- 사용자가 의도된 목적을 위해 시스템을 어느 정도나 효과적이고 효율적으며 만족스럽게 사용할 수 있는가를 나타낸다.
- **적정성 인지 능력**(appropriateness recognizability) – 소프트웨어가 자신의 필요에 적합한지를 사용자가 인식할 수 있음.
- **학습성** – 사용자가 소프트웨어 사용법을 얼마나 쉽게 배울 수 있는지.
- **사용자 오류 보호**(user error protection) – 사용자의 실수로부터 보호.
- **접근성** - 아주 다양한 특성과 능력을 갖춘 사람들이 모두 소프트웨어를 사용할 수 있게 하는 것.

아키텍처 특성의 정의

중요한 아키텍처 특성들 – ISO가 정리한 주요 특성(계속)

신뢰성

- 시스템이 지정된 조건에서 지정된 기간 동안 어느 정도나 잘 동작하는지를 나타낸다.
- **성숙도**(maturity) – 소프트웨어가 정상 운영 중에 신뢰성 요구사항을 충족하는지.
- **가용성** – 소프트웨어가 운영 가능하고 접근 가능한지.
- **내결함성** – 하드웨어 고장이나 소프트웨어 장애 상황에서도 소프트웨어가 의도한 대로 작동하는지.
- **복구성** – 소프트웨어가 장애로부터 복구하여 영향 받은 데이터를 복구하고 시스템의 원하는 상태를 재설정할 수 있는지.

보안

- 사람이나 다른 제품 또는 시스템이 자신의 유형과 권한 수준에 적합한 정도로만 데이터에 접근할 수 있게 함으로서 정보와 데이터를 보호하는 능력에 관한 것이다.
- **기밀성**(confidentiality) – 접근 권한이 있는 사람만 데이터에 접근할 수 있음.
- **무결성**(integrity) – 소프트웨어나 데이터에 대한 무단 접근이나 수정을 방지.
- **부인 방지**(nonrepudiation) – 행동이나 사건이 실제로 일어났음을 증명하는 능력.
- **설명책임성**(accountability) – 사용자의 행동을 추적하는 것.
- **진정성**(authenticity) – 사용자의 신원을 증명하는 것.

중요한 아키텍처 특성들 – ISO가 정리한 주요 특성(계속)

유지보수성

- 개발자가 소프트웨어를 개선하거나, 바로잡거나, 환경 또는 요구사항의 변화에 적응시키기 위해 어느 정도나 효과적/효율적으로 소프트웨어를 수정할 수 있는지를 나타낸다.
- **모듈성** – 소프트웨어가 개별 컴포넌트로 구성된 정도.
- **재사용성** – 개발자가 하나 이상의 시스템에서 또는 다른 자산을 구축할 때 자산을 사용할 수 있는 정도.
- **분석성**(analyzability) – 개발자가 소프트웨어에 대한 구체적인 지표를 얼마나 쉽게 수집할 수 있는지.
- **수정성**(modifiability) – 개발자가 결함을 도입하거나 기존 제품 품질을 저하시키지 않고 소프트웨어를 수정할 수 있는 정도.
- **테스트성**(testability) – 개발자와 다른 사람들이 소프트웨어를 얼마나 쉽게 테스트할 수 있는지.

이식성

- 개발자가 시스템, 제품 또는 컴포넌트를 한 하드웨어, 소프트웨어 또는 기타 운영이나 사용 환경에서 다른 환경으로 어느 정도나 옮길 수 있는지를 나타낸다.
- **적응성**(adaptability) – 개발자가 다른 또는 계속 바뀌는 하드웨어, 소프트웨어, 기타 운영 환경 및 사용 환경에 맞춰 소프트웨어를 효과적이고 효율적으로 적응시킬 수 있는지.
- **설치성** – 지정된 환경에서 소프트웨어를 설치 또는 제거할 수 있는지.
- **대체성**(replaceability) – 개발자가 다른 소프트웨어로 얼마나 쉽게 대체할 수 있는지.

트레이드오프와 가장 덜 나쁜 아키텍처

> “**최고의** 아키텍처를 추구하지 말고, **가장 덜 나쁜 아키텍처**를 목표로 하라”

- ✓ 아키텍트는 시스템의 성공에 필수이거나 중요한 아키텍처 특성만 지원해야 한다.
- ✓ 시스템이 모든 아키텍처 특성을 지원할 필요는 없다.
- ✓ 이유?
  - 어떤 특성이든 지원하려면 아키텍트의 설계 노력과 개발자의 구현 및 유지보수 노력이 필요하다.
  - 아키텍처 특성들은 서로 상승작용을 일으키며, 문제 도메인과도 상승작용을 일으킨다.
    즉, 하나를 변경하면 종종 다른 것도 변경해야 한다.
  - 아키텍처 특성에 대한 표준 정의가 없다 보니, 조직이 모호성과 싸워야 한다.
  - 소프트웨어 아키텍처가 복잡해질수록 조직의 다른 부분과 얽히는 경향이 있다.
- ✓ 너무 많은 아키텍처 특성을 지원하려 하지 말고, **상충하는 관심사들 사이에 트레이드오프를 결정**하라.
- ✓ 가능한 **반복(iteration)이 용이한 아키텍처**를 설계하도록 노력하라.
- ✓ **아키텍처를 변경하기 쉬우면**, 첫 시도에서 정답을 찾아야 한다는 **부담이 줄어**든다.

기초 – 아키텍처 특성의 식별

아키텍처 특성 식별

도메인 문제 이해

- 도메인 관심사
- 프로젝트 요구사항들
- 암묵적 도메인 지식

이해 관계자들과 협력해서 도메인 관점에서
진정으로 중요한 것이 무엇인지 결정한다.

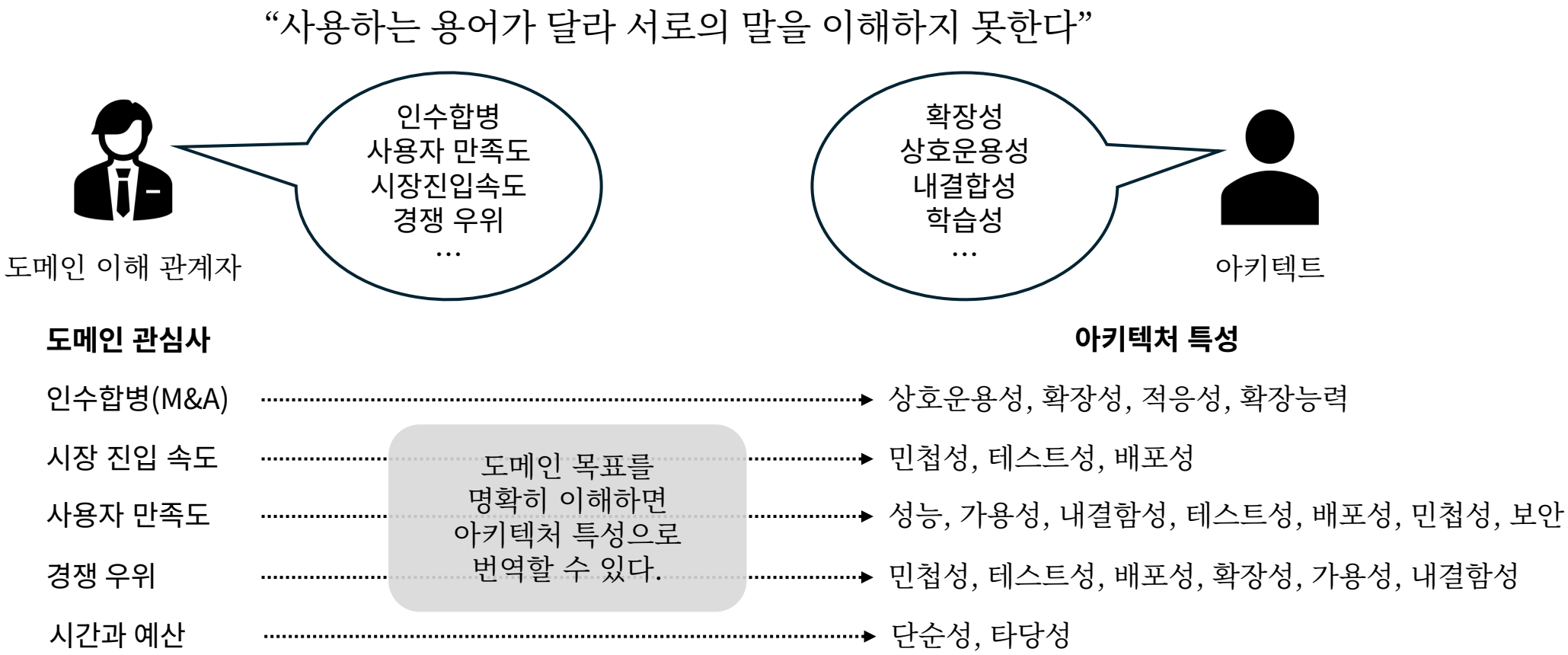
아키텍처 특성 식별

특정 문제나 애플리케이션에 적합한 아키텍처 특성

아키텍처 특성의 식별

도메인 관심사들에서 아키텍처 특성 도출하기

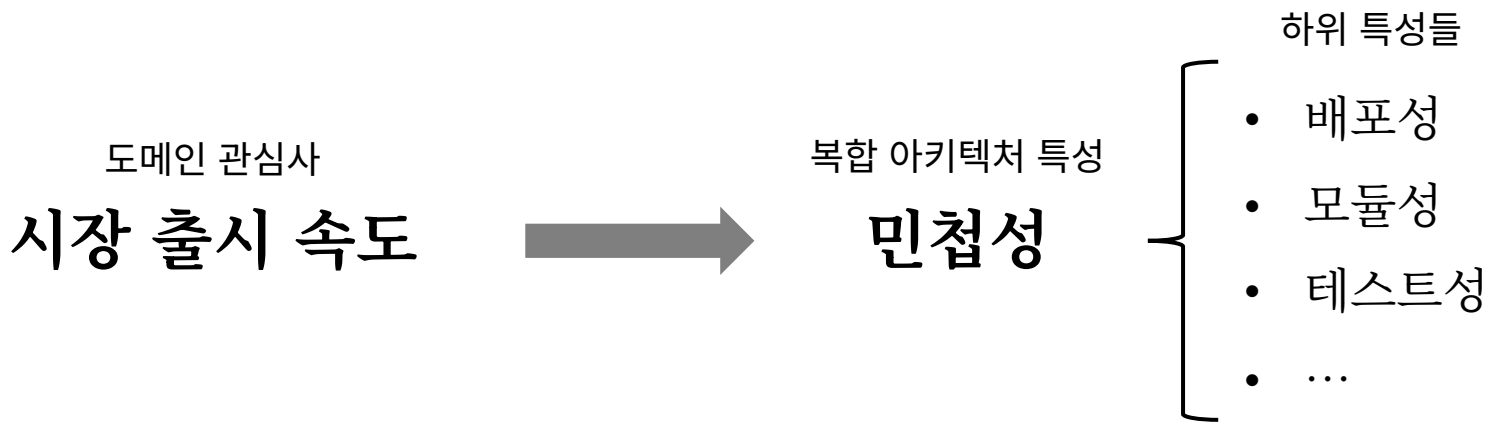
- **핵심 도메인 이해관계자들의 이야기**를 듣고 그들과 협력해서 **비즈니스의 관점**에서 **중요한 것이 무엇**인지를 알아내는 과정에서 얻어진다.



아키텍처 특성의 식별

복합 아키텍처 특성

- **도메인 관심사**를 아키텍처 특성으로 번역할 때 흔히 발생하는 실수 중 하나는 복합적인 개념을 담은 아키텍처 특성을 고려하지 않고 **단순히 하나의 특성으로 등치시키는 것**이다.
- 아키텍처 특성 중에는 **여러 요소로 이루어진 복합적인 개념을 담은 특성**이 있다.
- 이런 특성을 다룰 때는 이 특성을 **구성하고 있는 하위 특성들이 무엇인지 고민**해야 한다.

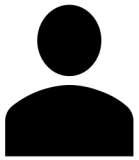


아키텍처 특성의 식별

복합 아키텍처 특성을 고려해야 하는 이유

도메인 이해 관계자

"규제 관련 요구사항 때문에, 장 마감 펀드 기준가 산정을 정해진 시간 내에 완료해야 한다" (도메인 관심사)



아키텍트

두드러진 특성에만 집착(❌) → **"성능"** (아키텍처 특성)

다른 특성에도 똑같이 주의를 기울일 수 있어야 한다.

- 시스템이 아무리 빨라도, 필요한 순간에 사용할 수 없으면 아무 소용 없다. (**가용성**)
- 시간에 지남에 따라 도메인이 커지고 펀드 수가 늘어나면, 일일 장 마감 처리 작업을 시간 내에 끝내도록 시스템의 규모를 확장할 수 있어야 한다.(**확장성**)
- 시스템은 항상 사용 가능해야 할 뿐 아니라, 신뢰성도 보장되어야 한다. 마감 시점에 펀드 가격을 계산하는 도중에 시스템이 다운되면 곤란하다.(**신뢰성**)
- 만약 시스템이 펀드 기준가 산정을 85% 끝낸 시점에 장애가 발생한다면, 중단된 시점부터 기준가 산정을 재개할 수 있어야 한다.(**복구성**)
- 시스템이 빠른 것만으로는 충분하지 않다. 기준가가 제대로 계산되는지도 검증해야 한다.(**감사성**)

아키텍처 특성의 식별

아키텍처 특성의 추출

- 대부분의 아키텍처 특성은 요구사항 문서(어떤 형식이든)에서 명시적인 문구 형태로 나타난다.

  예) 도메인 관심사들의 목록에는 **예상 사용자 수**나 **시스템 규모** 같은 명시적인 수치가 있다.
- 아키텍트가 지닌 도메인 지식에서 비롯되는 특성들도 있다. 이것이 각 아키텍트가 자신의 도메인을 잘 알아야 하는 이유 중 하나이기도 하다.

"**아키텍처 카타**(architecture kata)"

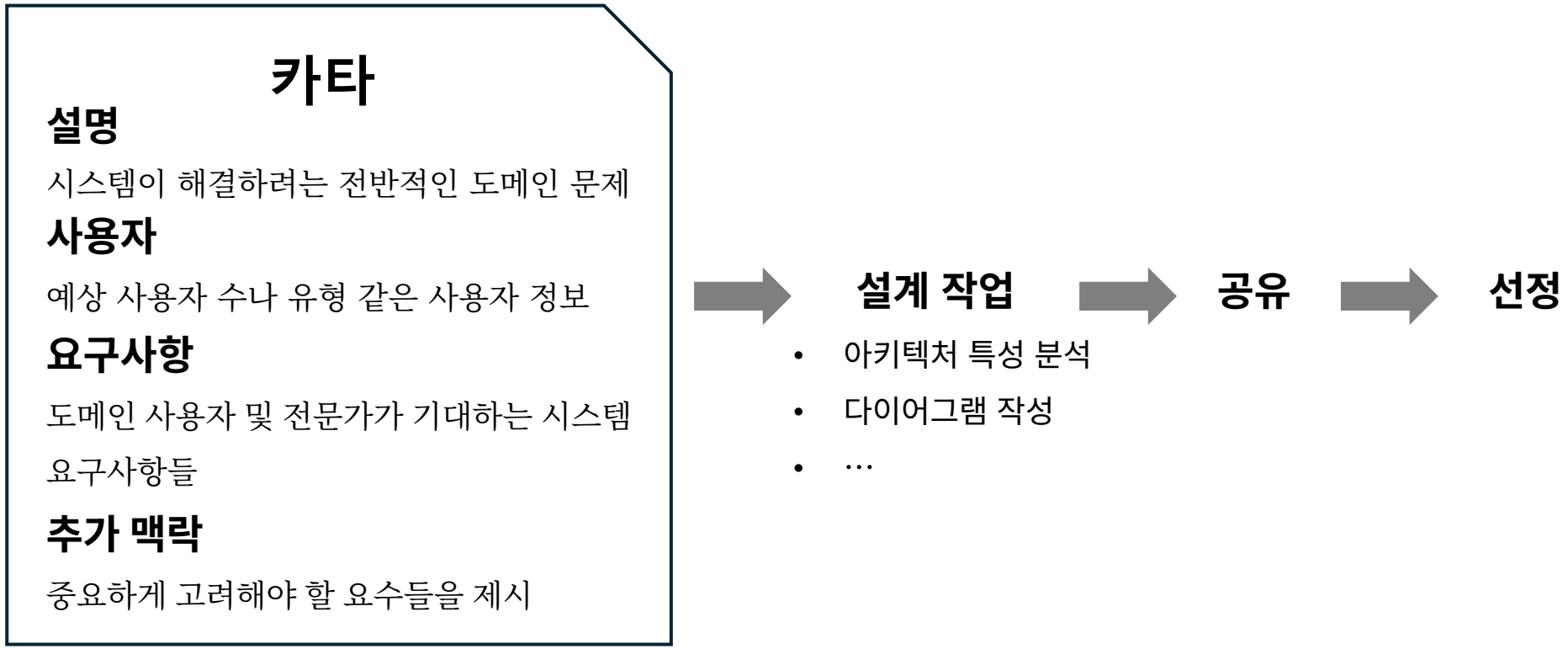
- 초보 아키텍트들이 도메인에 관한 설명에서 아키텍처 특성을 도출하는 연습을 할 수 있게 해주는 방법이다.
- 저명한 아키텍트인 테드 뉴어드(Ted Neward)가 고안했다.
- 카타 훈련 사이트
  - 테드가 만든 훈련 사이트 – https://www.architecturalkatas.com
  - 교재에서 제공하는 카타 목록 사이트 – https://fundamentalsofsoftwarearchitecture.com

*** 카타**

올바른 자세와 기술을 강조하는 개인 무술 훈련 방식을 가리키는 일본어 '가타(かた)'에서 유래했다.

아키텍처 카타

- 각각의 카타 실습은 특정 문제를 **도메인 관점에서 서술한 문장**과 **관련 요구사항들**, 그리고 **추가적인 맥락**(요구사항에 포함되지 않더라도 설계에 영향을 줄 요소들)을 제시한다.
- 이에 기반해서 소규모 팀들이 일정 시간 동안 **설계 작업(아키텍처 특성 분석 및 다이어그램 작성)을 진행**한다.
- 이후 **각 팀이 제출한 아키텍처**를 모든 팀이 **공유하고**, 투표를 통해 가장 뛰어난 아키텍처 하나를 **선정한다**.



아키텍처 특성의 식별

카타 실습 – 실리콘 샌드위치

설명

전국 규모의 샌드위치 전문점이 기존의 전화 주문 서비스 외에 온라인 주문 기능을 제공하려고 한다.

사용자

사용자는 현재는 수천 명이지만 언제가는 수백만 명에 이를 수도 있다.

요구사항

- 사용자가 음식을 주문 할 수 있도록 한다. 만야 매장이 배달 서비스를 제공한다면, 사용자가 포장 또는 배달을 선택할 수 있어야 한다.
- 포장 고객에게는 주문을 수령할 시간과 매장을 찾아오는 방법을 안내한다.(이를 위해서는 교통 정보와 외부 지도 서비스 통합이 필요)
- 배달 서비스의 경우, 주문된 음식을 배달 기사가 사용자에게 배달한다. 모바일 기기로 접근할 수 있게 한다.
- 전국 모든 매장에서 일일 판촉/할인 행사를 시행한다. 매장별로 자체 일일 판촉/할인 행사를 시행한다.
- 온라인 결제, 매장 내 결제, 착불 결제(배달 시 결제)를 모두 허용한다.

추가 맥락

- 실리콘 샌드위치 매장들은 가맹점 방식(프랜차이즈)으로 운영된다. 따라서 매장마다 소유자가 다르다.
- 본사는 조만간 해외 진출을 계획하고 있다.
- 본사의 목표는 값싼 인력을 고용해서 이윤을 극대화하는 것이다.

카타 실습 – 1) 명시적 특성

설명

전국 규모의 샌드위치 전문점이 기존의 전화 주문 서비스 외에 온라인 주문 기능을 제공하려고 한다.

사용자

사용자는 현재는 수천 명이지만 **언제가는 수백만 명에 이를 수도 있다**.

확장성

- 문제 **설명 섹션**에 확장성이라는 단어가 명시되어 있지 않지만, **사용자 섹션**의 **예상 사용자 수**가 암묵적으로 확장성을 시사한다.
- 동시 접속 사용자가 많아져도 성능 저하 없이 이를 처리할 수 있는 능력을 최우선 아키텍처 특성의 하나로 두어야 한다.
- 확장성은 시간에 따라 사용자가 꾸준히 늘어나는 상황에서 측정된다. **예) 호텔 예약 시스템**

탄력성

- 탄력성은 짧은 시간에 몰리는 요청을 처리할 수 있는 능력이다.
- 즉, 트래픽이 순간적으로 급증할 때 측정된다. **예) 콘서트 티켓 예매 시스템**
- 요구사항에 탄력성이라는 단어는 없지만, 매장의 특성 상 **식사 시간대에 트래픽이 집중**되기 때문에 탄력성을 도출하였다.

카타 실습 – 1) 명시적 특성(요구사항 분석)

사용자가 음식을 주문 할 수 있도록 한다. 만약 매장이 배달 서비스를 제공한다면, 사용자가 포장 또는 배달을 선택할 수 있어야 한다.

- 이 요구사항을 지원하기 위해 특별한 이키텍처 특성이 필요해 보이지 않는다.

포장 고객에게는 주문을 수령할 시간과 매장을 찾아오는 방법을 안내한다.(이를 위해서는 교통 정보와 외부 지도 서비스 통합이 필요)

- 외부 지도 서비스에는 통합 지점이 필요하다. 이는 **신뢰성** 같은 품질 특성에 영향을 미친다. 개발자가 서드파티 시스템에 의존하는 시스템을 구축할 경우, 서드파티 시스템이 고장 나면 그것을 호출하는 시스템의 신뢰성도 떨어진다.
- 만약 외부 교통 정보 서비스가 중단된다면? 전체 사이트가 장애를 내야 할까, 아니면 그냥 교통 정보 없이 덜 효율적으로 안내할까?
- 아키텍트는 항상 불필요한 취약성을 설계에 도입하는 것을 경계해야 한다.

배달 서비스의 경우, 주문된 음식을 배달 기사가 사용자에게 배달한다.

- 이 요구사항 역시 특별한 이키텍처 특성을 요구하지 않는다.

카타 실습 – 1) 명시적 특성(요구사항 분석 계속)

모바일 기기로 접근할 수 있게 한다.

- 이 요구사항은 주로 애플리케이션의 사용자 경험 설계에 영향을 미친다.
- 이식성이 좋은 **웹 애플리케이션 하나를 만들 것인지** 아니면 **네이티브 앱 여러 개를 만들 것인지**가 주된 고려사항이 될 것이다.
- 예산이 한정적이고 기능이 단순하다면, 여러 개의 앱을 따로 구축하는 것보다 모바일 환경에 최적화된 웹 애플리케이션이 더 낫다.
- 만약 그렇다면 **모바일에 민감한** 요소를 최적화하기 위한 **성능 관련 아키텍처 특정 몇 가지를 명확하게 정의할 필요가 있다**.
- 이 경우 아키텍트 뿐만 아니라 UX 디자이너, 도메인 관계자, 기타 이해관계자와 협업해서 결정 사항을 검증하는 것이 바람직하다.
- 예를 들어 네이티브 앱에서만 가능한 어떤 기능을 경영진이 원할 수도 있다.

전국 모든 매장에서 일일 판촉/할인 행사를 시행한다. 매장별로 자체 일일 판촉/할인 행사를 시행한다.

- 두 요구사항 모두, 판촉 및 할인 행사를 위한 고도의 **맞춤성(customizability; 커스텀화 능력)**을 암시한다.
- 즉 **맞춤성을 아키텍처 특성의 하나로 고려**해야 할 것이다.

카타 실습 – 1) 명시적 특성(요구사항 분석 계속)

온라인 결제, 매장 내 결제, 착불 결제(배달 시 결제)를 모두 허용한다.

- 일반적으로 온라인 결제는 보안이 필요함을 암시한다. 그러나 이 예제에서는 높은 수준의 보안이 요구되지는 않는다.
- 보안은 설계에서 처리할 수도 있고, 아키텍처 차원에서 처리할 수도 있으므로, 보안은 그리 중요한 아키텍처 특성이 아니다.

실리콘 샌드위치 매장들은 가맹점 방식(프랜차이즈)으로 운영된다. 따라서 매장마다 소유자가 다르다.

- 이 요구사항은 아키텍처 비용 측면에 제약을 줄 수 있다. 비용, 일정, 인력의 숙련도 같은 제약조건들을 고려해서, 좀 더 **단순한 아키텍처**를 정의하던가, 아니면 '**희생형 아키텍처**(https://martinfowler.com/bliki/SacrificialArchitecture.html)' 의 타당성을 검토해야 한다.
- 희생형 아키텍처란, "**지금 만들고 있는 아키텍처는 언젠가 반드시 버려질 것**"이라는 전제로 설계하는 접근 방식이다. 성장/확장성/운영 패턴이 아직 확정되지 않은 경우에 채택하는 방식이다. **초기 제품이나 서비스인 경우** 완전히 미래를 맞추는 아키텍처보다, **지금 빠르게 결과를 만들 수 있는 아키텍처가 더 낫다**.

카타 실습 – 1) 명시적 특성(요구사항 분석 계속)

본사는 조만간 해외 진출을 계획하고 있다.

- 이 요구사항은 **국제화**(internationalization; i18n)를 시사한다.
- 이 부분은 아키텍처 구조 변경 없이 다양한 설계 기법으로 대응할 수 있다. 다만 UX 관련 결정에는 영향을 줄 것이 분명하다.

본사의 목표는 값싼 인력을 고용해서 이윤을 극대화하는 것이다.

- 이 요구사항은 사용성이 중요함을 암시하지만, 역시 설계상의 관심사일 뿐 이키텍처 특성과는 거리가 있다.

성능

✓ 요구사항들에서 도출할 수 있는 아키텍처 특성은 **성능**이다.

✓ 성능이 나쁜 앱으로 샌드위치를 사고 싶은 사람은 없을 것이다. 특히 손님이 많은 피크 타임이라면 더욱 그렇다.

카타 실습 – 2) 암묵적 특성

- 요구사항에 명시되어 있지 않지만 설계에서 여전히 매우 중요한 요소인 것이 많다.

가용성

✓ 사용자는 언제라도 실리콘 샌드위치 사이트에 접속할 수 있어야 한다. 이를 보장하는 것이 가용성이다.

신뢰성

✓ 사용자가 사이트를 이용하는 동안 사이트가 멀쩡하게 유지되는 것을 말한다. 이용 도중에 연결이 끊겨서 다시 로그인해야 하는 사이트에서 상품을 구매하려는 사용자는 별로 없을 것이다.

보안

✓ 모든 시스템에서 기본적으로 고려해야 할 암묵적 특성이다.
✓ 보안이 설계의 구조적 측면에 영향을 미치며, 애플리케이션에 반드시 필요하거나 중요하다면 아키텍처 특성의 하나로 간주해야 한다.
✓ 결제 처리를 서드파티(소위 PG사)에 맡기면 보안을 위해 별도의 아키텍처 구조를 마련할 필요가 없다.

맞춤성

✓ 여러 요구사항의 세부 내용에 따르면 매장별로 재정의가 필요한 부분이 많다. 레시피, 지역별 판매, 매장 방문 경로 안내 등이 그렇다.
✓ 따라서 아키텍처는 앱 행동방식의 커스텀화를 지원해야 한다.

카타 실습 – 아키텍처 특성 결정

- 딱 맞는 아키텍처 특성들을 받드시 찾아내야 한다고 너무 스트레스 받지 말라.

  → 개발자는 주어진 기능성을 구현하는 다양한 방법을 알고 있다. 개발자를 믿고 부담을 덜어라.
- 하지만 **아키텍트가 중요한 구조적 요소를 제대로 파악**하면, **좀 더 단순하고 우아한 설계**를 끌어내기가 쉽다.
- 아키텍처에 **'최고' 설계는 없다**. **'가장 덜 나쁜' 트레이드오프들의 모음이 있을 뿐**이다.
- 후보 아키텍처 특성들에 우선순위를 매겨서 가장 단순한 필수 특성 집합을 만들려고 노력해야 한다.

  → 가장 덜 중요한 요소가 무엇인지 찾아 보라.

설계와 아키텍처, 그리고 트레이드오프

아키텍처로 처리할 것인가? 아니면 설계로 처리할 것인가?

- 답은 **상황에 따라 다르다**.
- 해당 **아키텍처를 도입하지 말아야 할 합당한 이유가 있는지** 생각해야 한다. (성능 저하, 강한 결합 등)
- 아키텍처 특성 중에서 **아키텍처와 설계 중에서 어느 것이 구현하기 더 어려운 것인지** 따져본다.
- 아키텍처 특성 모두에 대해 **아키텍처 레벨**에서 지원하는 것과 **설계 레벨**에서 지원하는 것 중 **어떤 쪽이 비용이 더 많이 드는지도 중요**하다.
- 개발자, 프로젝트 관리자, 운영 팀 등 소프트웨어 시스템의 공동 구축자들과 긴밀한 협력이 중요하다.
- 구현 팀과 분리된 채로 아키텍처 의사결정을 내리는 것은 위험하다. 이른바 '상아탑 아키텍처(Ivory Tower architecture)' 라는 안티패턴으로 이어진다.

아키텍처 특성의 제한과 우선순위 부여

"도메인 이해관계자들과 협력하면서 핵심 **아키텍처 특성들을 정의**할 때는

반드시 최종 목록을 최대한 짧게 유지하기 위해 노력해야 한다"

아키텍처 특성 워크시트

https://developertoarchitect.com/resources.html

주의!

- **범용** 아키텍처를 설계하려는 것.
- **모든 아키텍처 특성을 지원**하는 아키텍처를 만들려는 것.
- 아키텍처가 지원하는 특성이 많아질수록 **전체 시스템의 설계가 복잡**해진다.

기초 – 아키텍처 특성의 측정과 거버넌스

아키텍처 특성의 측정과 거버넌스

도메인 문제 이해

- 도메인 관심사
- 프로젝트 요구사항들
- 암묵적 도메인 지식

이해 관계자들과 협력해서 도메인 관점에서
진정으로 중요한 것이 무엇인지 결정한다.

아키텍처 특성 식별

특정 문제나 애플리케이션에 적합한 아키텍처 특성