



Microservices Architecture

(ver 2.1)

목 차

0. 들어가기 전에...
1. 비즈니스 서비스 이해
2. SOA 이해
3. RESTful API
4. Business Component
5. DDD
6. Microservices
7. 마이크로서비스와 조직
8. 마이크로서비스와 아키텍처
9. 사례 연구

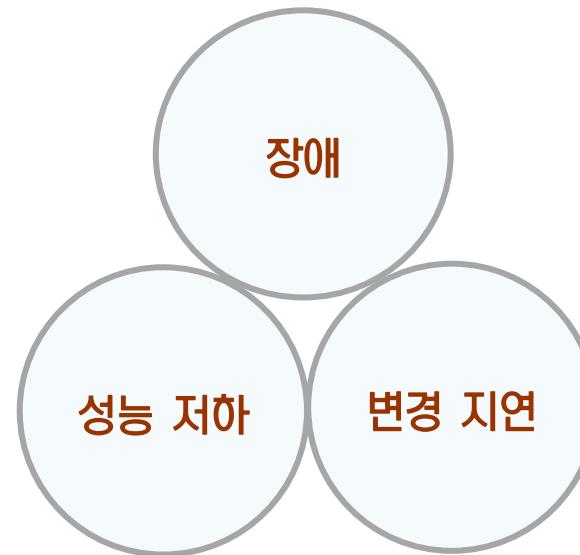


0. 들어가기 전에...

-
- 0.1 기업 IT 운영의 3대 과제
 - 0.2 기업의 관심사
 - 0.3 복잡도
 - 0.4 구조적인 대응
 - 0.5 기술 흐름
 - 0.6 Why MSA?
 - 0.7 요약

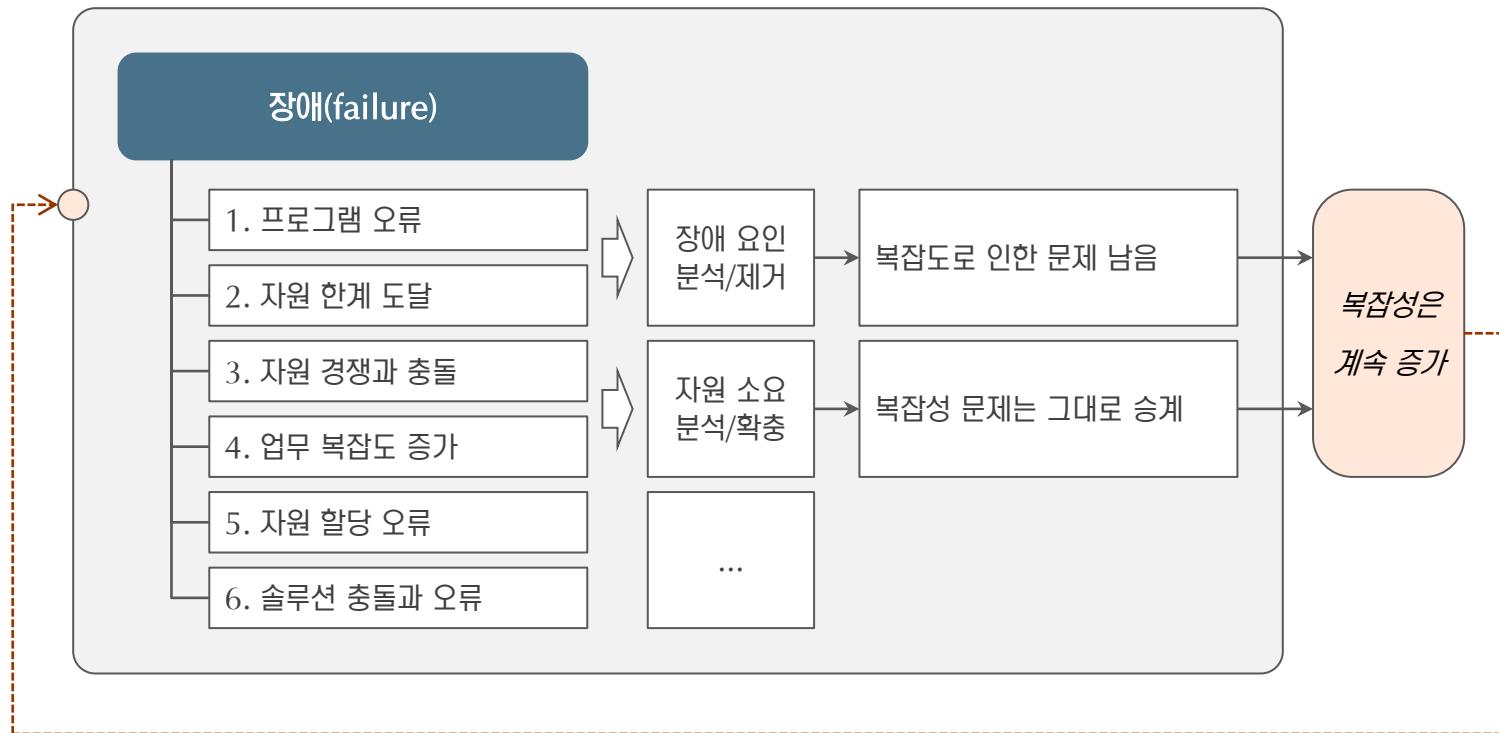
0.1 기업 IT 운영의 3대 과제

- ✓ IT 운영 조직의 최대 관심사는 운영 시스템의 장애, 성능, 변경 대응, 세 가지로 볼 수 있습니다.
- ✓ 최근 들어 보안, 규제 등도 관심사이지만, 이러한 이슈는 변경 대응으로 볼 수 있습니다.
- ✓ 세 가지 관심사는 해결할 수 있고, 해결하려는 의지는 있지만, 해결이 불가능한 “언제나 이슈”입니다.
- ✓ 결국 세 가지 이슈를 제대로 해결하지 못하면서 IT 예산을 급증하는 경향이 있습니다.



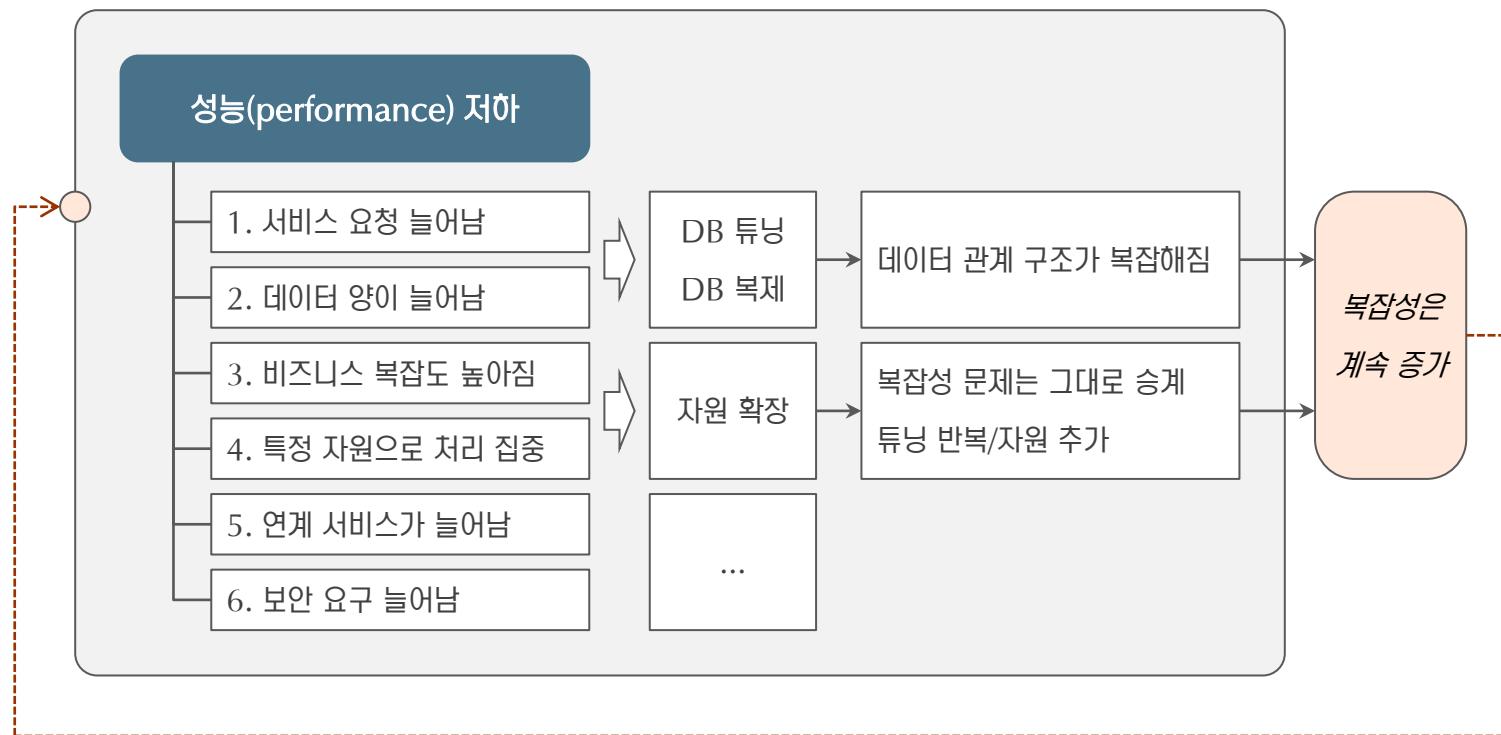
0.2 기업의 관심사 (1/3): 장애(failure)

- ✓ 컴퓨터 시스템을 구성하는 모든 요소 중에 장애로부터 완전히 자유로운 요소는 없습니다.
- ✓ 장애는 피할 수 없습니다. 따라서, 장애로 인한 피해를 줄이는 방법은 장애 대응 역량에 있습니다.
- ✓ 대응 역량은 대응 가능한 환경 구축, 대응 가능한 시스템 구조 설계, 대응 가능한 운영 방식 등을 포함합니다.



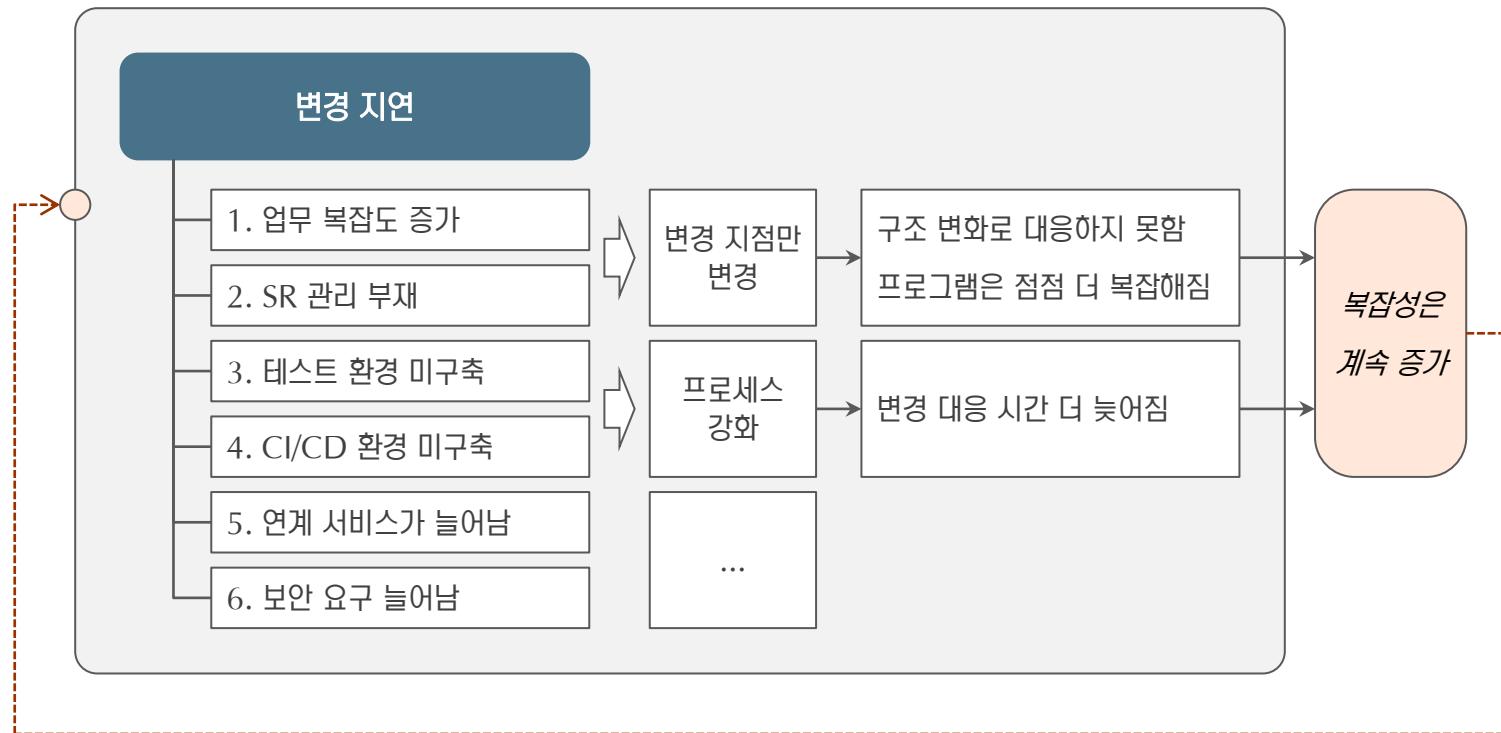
0.2 기업의 관심사 (2/3): 성능(performance)

- ✓ 어느 시스템이나 구축 후 오픈 초기에는 적절한 수준의 성능을 제공하도록 설계합니다.
- ✓ 발전하는 조직일수록 시스템 성능 문제에 빨리 부딪힙니다. 사용자와 데이터가 빨리 늘어나기 때문입니다.
- ✓ 대체로 자원을 최적화하는 방식으로 문제를 해결하지만, 어느 순간부터 scale-up 한계에 부딪히게 됩니다.



0.2 기업의 관심사 (3/3): 변경 지연

- ✓ 시스템이 복잡해 질수록 비즈니스 변경에 따른 프로그램 변경 대응이 늦어지는 경향이 있습니다.
- ✓ 변경 대응 지연 보다 더 큰 문제는 때로는 변경 자체가 불가능하며, 그에 따라 비즈니스 확장을 방해합니다.
- ✓ 업무 규모 크고 시스템 규모가 클 수록 변경 대응 시간은 지수적으로 지연되는 경향이 있습니다.



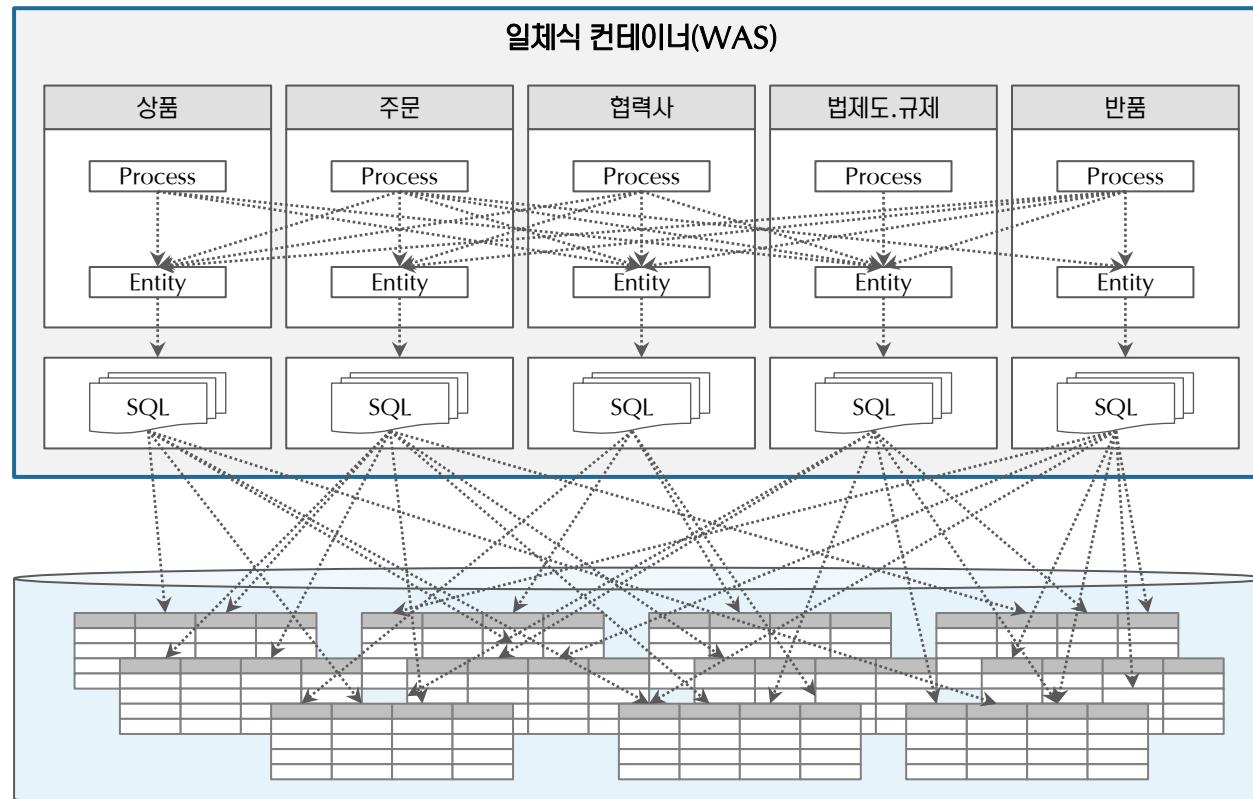
0.3 복잡도 [1/2]

- ✓ 세 가지 핵심 이슈를 해결하기 위한 활동이 활발할 수록 시스템의 복잡도는 빠른 속도로 증가합니다.
- ✓ 복잡한 시스템은 다시 장애, 성능 저하, 변경 지연의 또 다른 이유가 됩니다.
- ✓ 시스템 구조 관점에서 낮은 복잡도를 유지하도록 설계하는 것이 “해답”이 될 수 있는 이유입니다.
- ✓ 물론 이러한 설계가 모든 문제를 해결할 수는 없지만, 적어도 문제에 대한 신속한 대응 역량을 제공합니다.



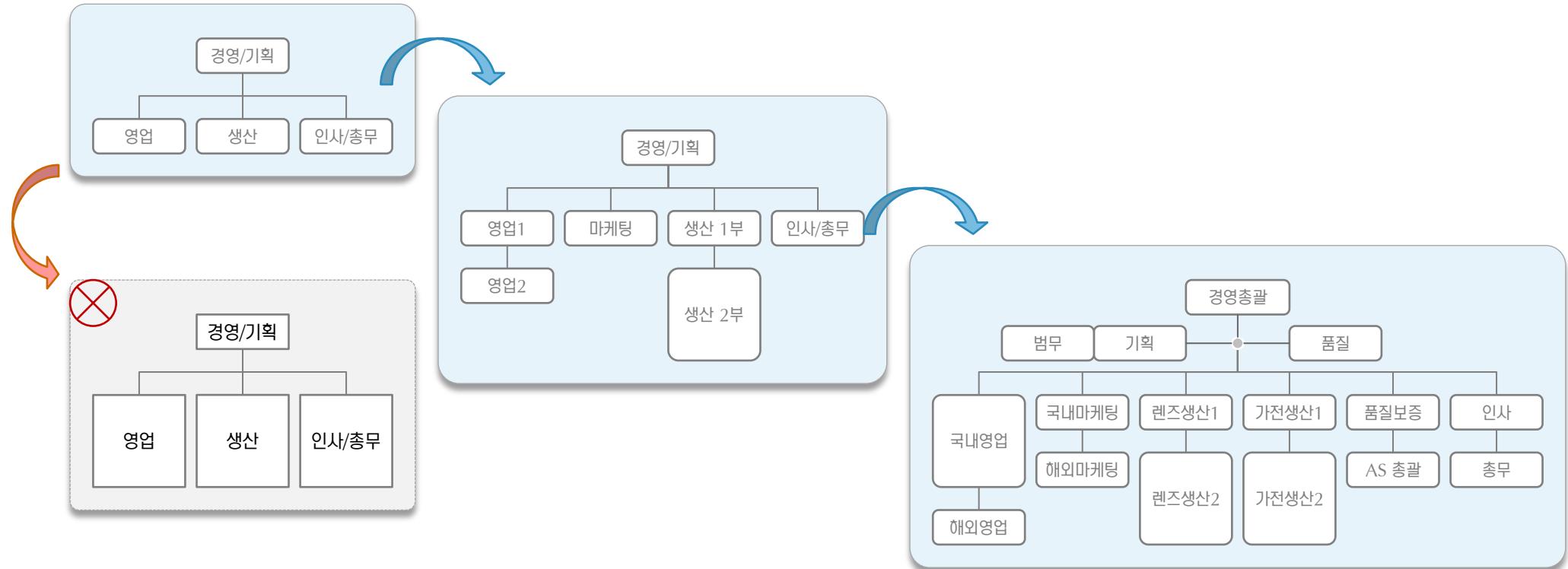
0.3 복잡도 [2/2]

- ✓ IT 기반의 서비스를 제공하는 조직의 기간계나 기간계에 해당하는 시스템은 매우 거대한 규모를 자랑합니다.
- ✓ 어떤 금융사의 경우, 기간계 배포에만 두 시간이상이 걸릴 정도로 엄청난 규모입니다.
- ✓ 어떤 컴포넌트에 한 줄이라도 변경을 한다면, 전체를 다시 배포해야 합니다. 이 거대한 시스템의 구조는 그대로 둔 채로 지속적으로 추가하거나 변경하는 방식으로 시스템을 운영하고 있습니다.



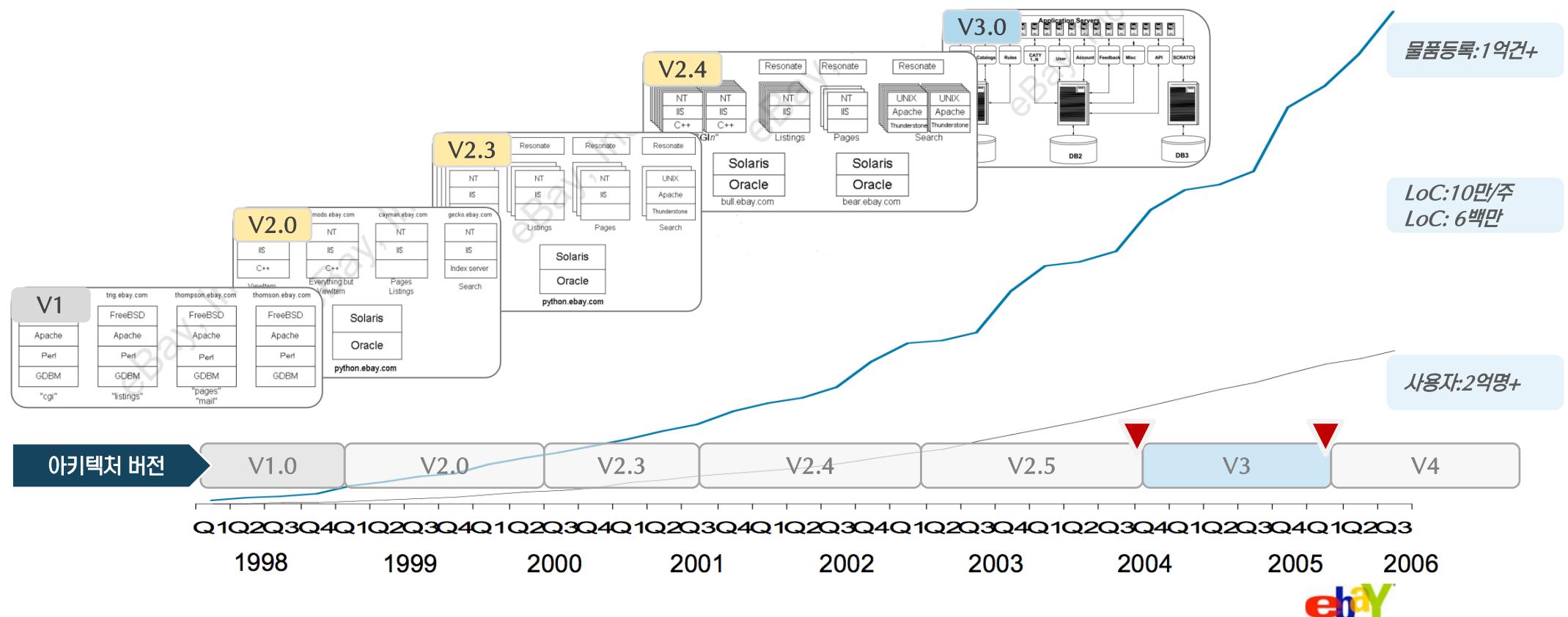
0.4 구조적인 대응 (1/2)

- ✓ 성장하는(growing) 기업은 급변하는 비즈니스 환경에 대응하기 위해 다양한 전략을 마련하고 실천합니다.
- ✓ 조직[구조]의 변화를 통한 경쟁력 확보는 무엇보다 중요한 전략 요소입니다.
- ✓ 기존 팀의 분할하거나 합치고, 새로운 팀을 추가하고, 팀의 규모를 늘리는 방식으로 조직 역량을 높여 갑니다.
- ✓ 성장하는 기업의 조직은 끝없이 성장함으로써, 기업의 경쟁력을 확보하여 성장할 수 있는 발판을 마련합니다.



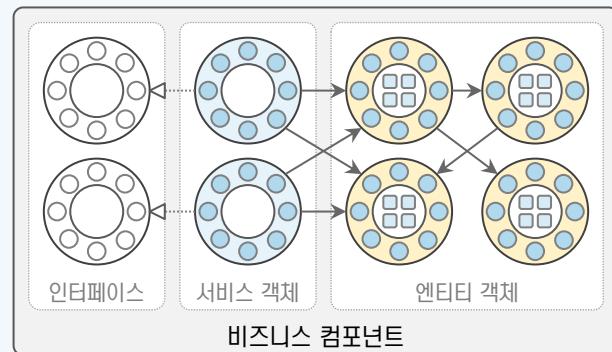
0.4 구조적인 대응 [2/2] – ebay

- ✓ ebay.com은 빠른 속도로 성장하여 온 e-commerce 서비스의 대표 기업입니다.
- ✓ 사용자와 물품 등록 건수의 증가에 따라, 아키텍처(구조)를 지속적으로 개선하는 방식으로 대응했습니다.
- ✓ 각 아키텍처(시스템 구조) 버전 별로 감당할 수 있는 사용자와 물품 등록 건수가 있습니다.



0.5 기술 흐름(1/6) – OOAD,CBD

- ✓ SW 설계 접근방법 → 절차 지향 설계와 객체 지향 개념을 바탕으로 하는 설계(OOAD, CBD, SOA, MSA)
- ✓ 두 세계는 패러다임이 전혀 다릅니다. 프로그래밍 언어 역시 두 가지로 나눌 수 있습니다.
- ✓ 객체 지향 설계 이후, SW 설계는 비약적으로 성장하였으며, 현재에 이르러서는 모든 산업의 기반이 되었습니다.
- ✓ OOAD는 많은 문제를 안고 있었지만, 그것을 극복하고 컴포넌트 기반 설계로 발전하였습니다.



질서정연한 객체들의 집합으로 구성한 비즈니스 컴포넌트

1995년 ~

OOAD (객체 지향 분석 설계)

절차 지향 분석 설계

2002년 ~

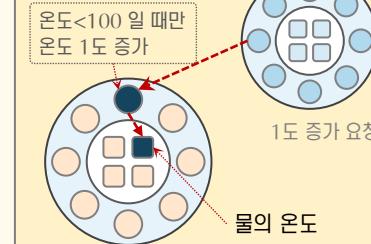
CBD (컴포넌트 기반 개발)

2008년 ~

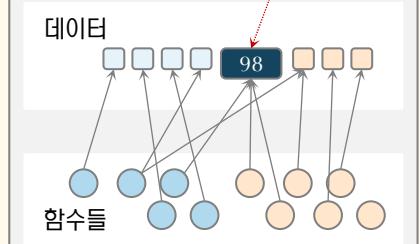
SOA (서비스 지향 아키텍처)

MSA (마이크로 서비스 아키텍처)

객체지향



절차지향



0.5 기술 흐름[2/6] – SOA, MSA

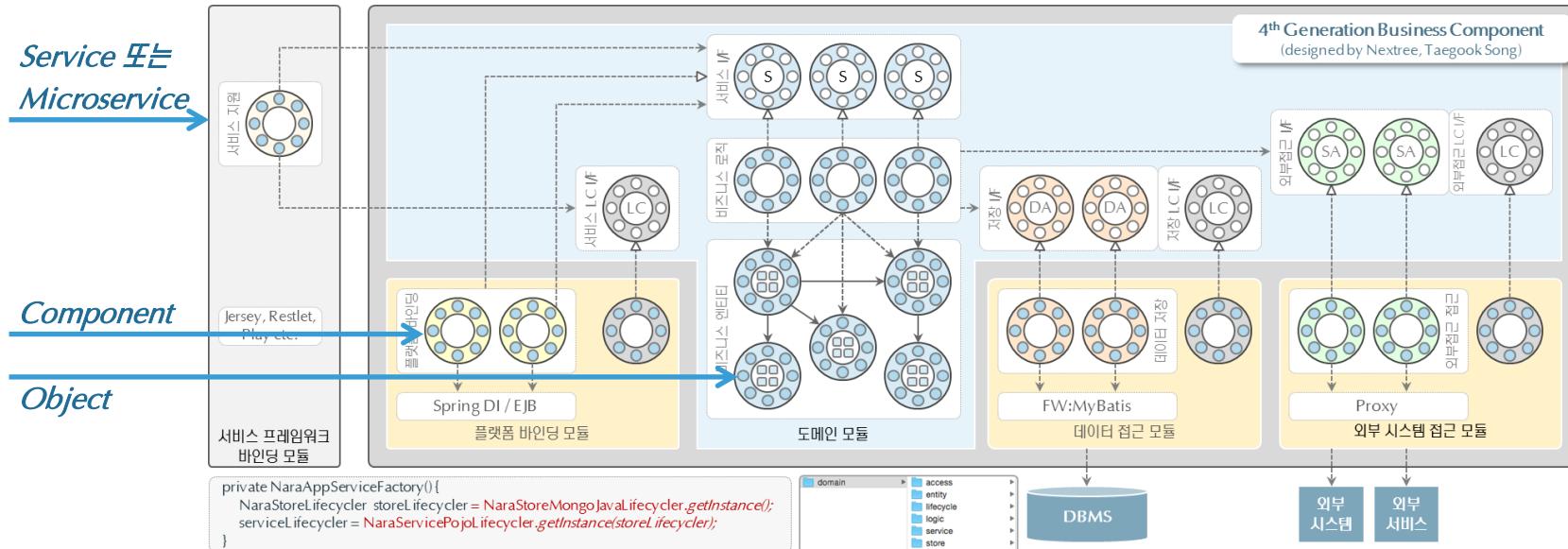
- ✓ 복잡한 컴퓨팅 환경은 기업 내부 분산을 넘어서 기업 간 분산 환경으로 발전하였습니다.
- ✓ SOA는 컴포넌트를 뛰어 넘어, 비즈니스 단위로 써의 서비스로 발전하였으며, 다양한 시련을 겪어왔습니다.
- ✓ 컴포넌트 다음 세대의 서비스는 플랫폼과 환경, 표준 등의 문제를 극복하면서 마이크로 서비스로 발전하였습니다.
- ✓ 이전 기술을 버리고 새로운 기술로 발전한 것이 아니라, 이전 기술의 축적을 바탕으로 발전하여 갔습니다.



0.5 기술 흐름(3/6) – 컴포넌트

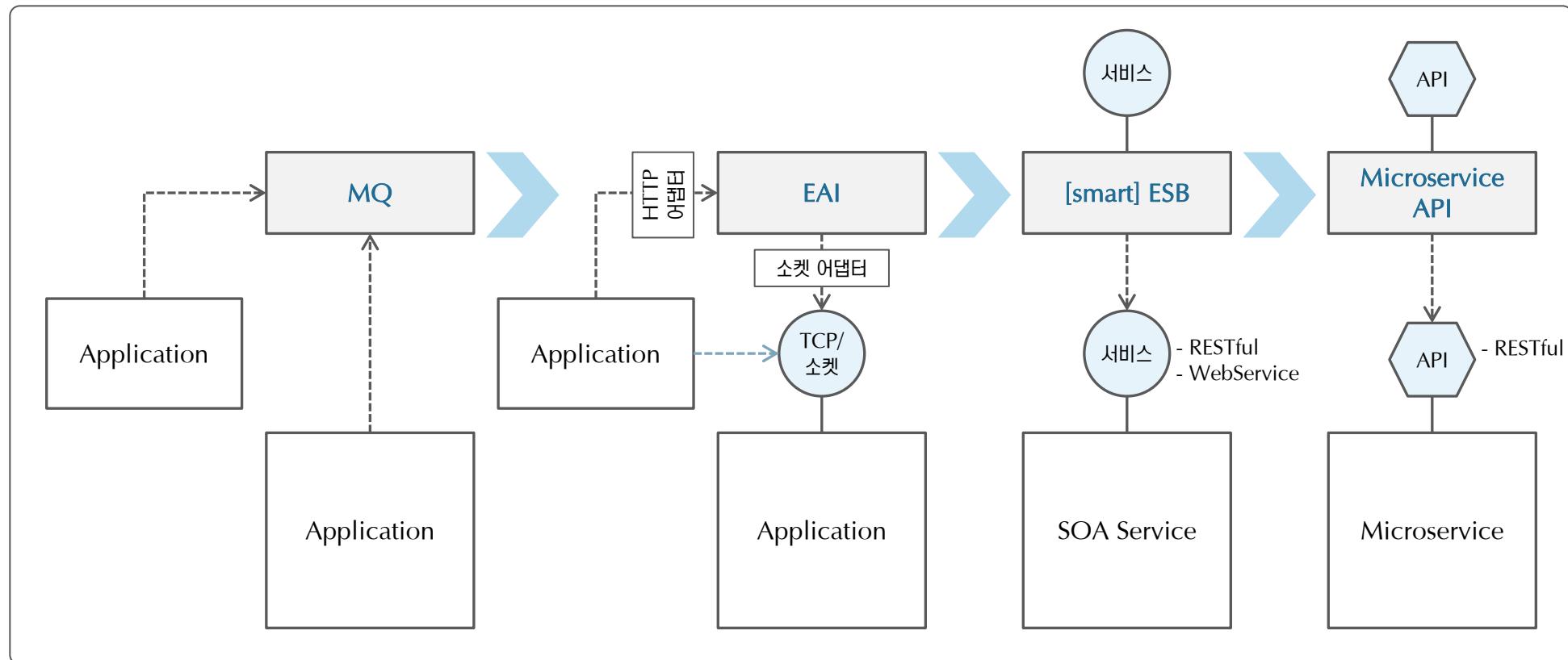
- ✓ 객체 지향 기술과 컴포넌트 기술은 퇴색되지 않고, 서비스 저 안쪽에 탄탄하게 자리잡고 있습니다.
- ✓ 객체 모델링 역량은 컴포넌트와 서비스 내부를 잘 채우는데 도움을 줍니다.
- ✓ 컴포넌트 모델링 역량이 있어야 기술적으로 또는 업무적으로 의미 있는 객체 그룹을 구성할 수 있습니다.
- ✓ 무늬만 컴포넌트로 만드는 이유는 객체 모델링 역량과 컴포넌트 모델링 역량이 부족하기 때문입니다.

[4세대 비즈니스 컴포넌트와 서비스]



0.5 기술 흐름[4/6] – 연결 기술

- ✓ 기업 시스템은 여러 개의 애플리케이션으로 구성되어 있으며, 서로 간에 연결을 통해 협업을 합니다.
- ✓ 과거의 MQ로부터 현재의 Microservice API에 이르기까지 기술이 발전해 왔습니다.
- ✓ MQ나 EAI는 여전히 많이 쓰고 있는 기술이지만, 전체 흐름은 REST 서비스 쪽으로 흘러가고 있습니다.



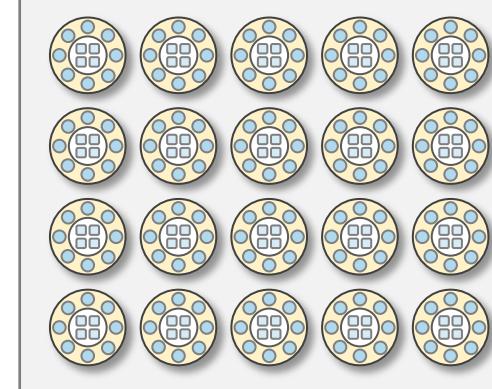
0.5 기술 흐름[5/6] – 객체로부터 서비스까지

✓ OOAD로부터 MSA로 이르는 과정을 간략하게 표현하면 다음과 같습니다.

서비스 인터페이스 →

컴포넌트 인터페이스 →

CORBA TCP 소켓

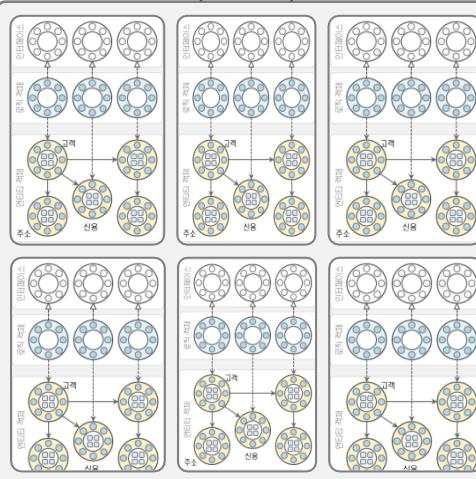


- ✓원격 인터페이스: CORBA,TCP
- ✓표현 단위: 객체
- ✓컨테이너: 단일 컨테이너

문제: 표현단위, 원격 통신

[OOAD]

Vender*



- ✓원격 인터페이스: RMI/IIOP
- ✓표현 단위: 컴포넌트
- ✓컨테이너: 단일 컨테이너

문제: 원격통신, 컨테이너 복잡도, 표준

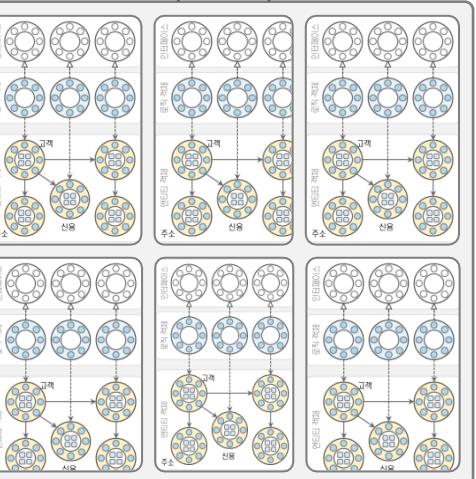
[CBD]

RESTful

Web Front

ESB

RMI/IIOP



- ✓원격 인터페이스: 웹서비스/RESTful
- ✓표현 단위: 컴포넌트/서비스
- ✓컨테이너: 단일 컨테이너

문제: [원격통신], 컨테이너 복잡도

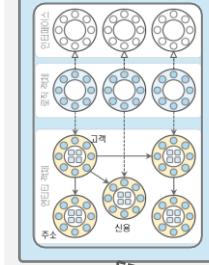
[SOA]

SOAP/WSDL

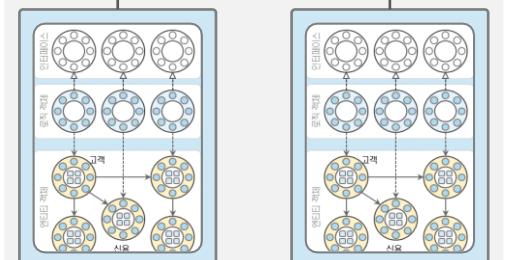
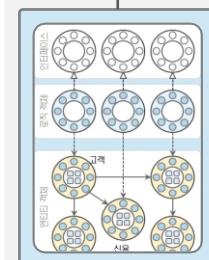
ESB

RMI/IIOP

RESTful



RESTful



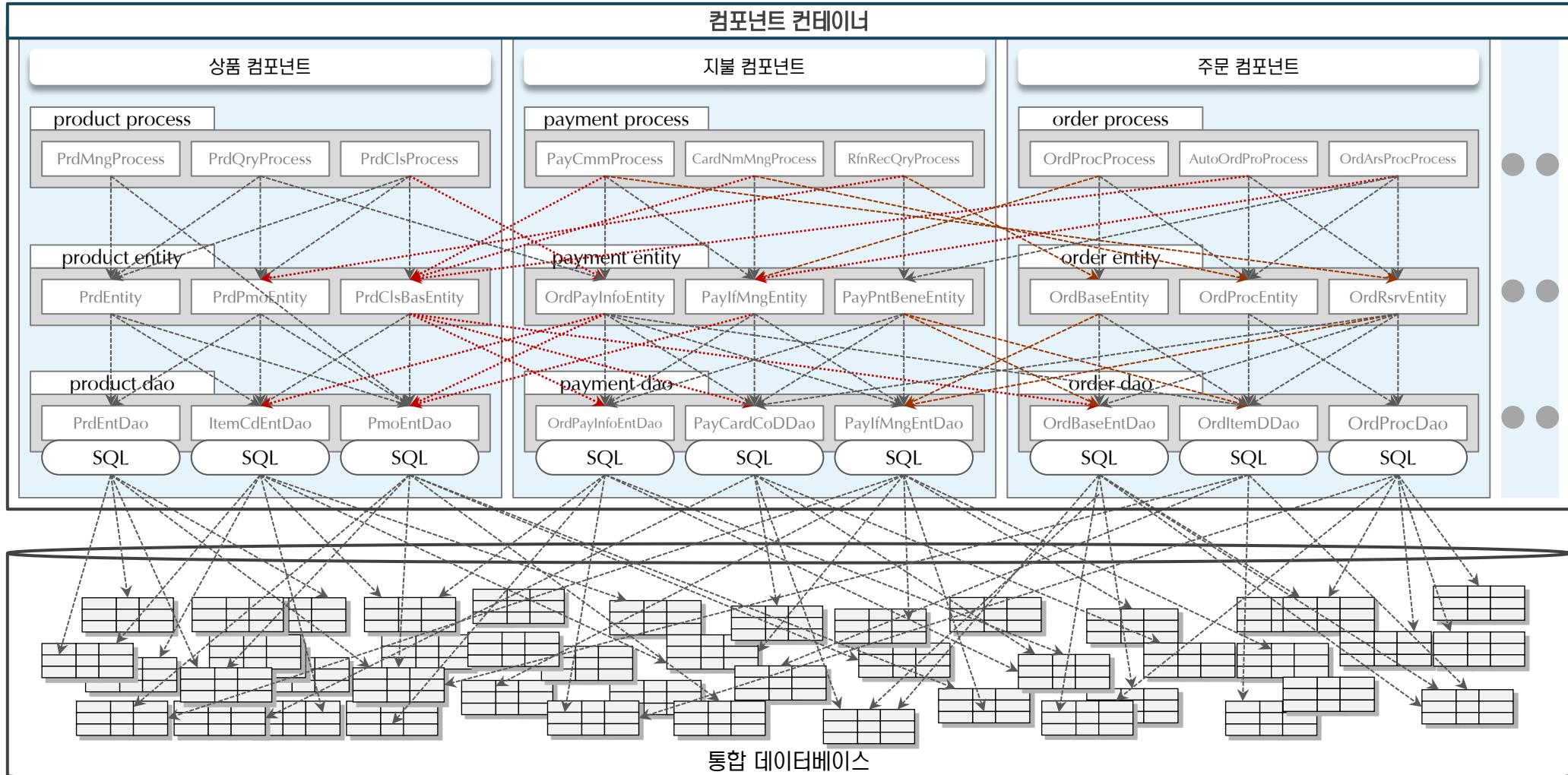
- ✓원격 인터페이스: RESTful
- ✓표현 단위: 마이크로서비스/컴포넌트
- ✓컨테이너: 마이크로서비스 컨테이너

원격통신, 복잡도 문제를 모두 해결함

[MSA]

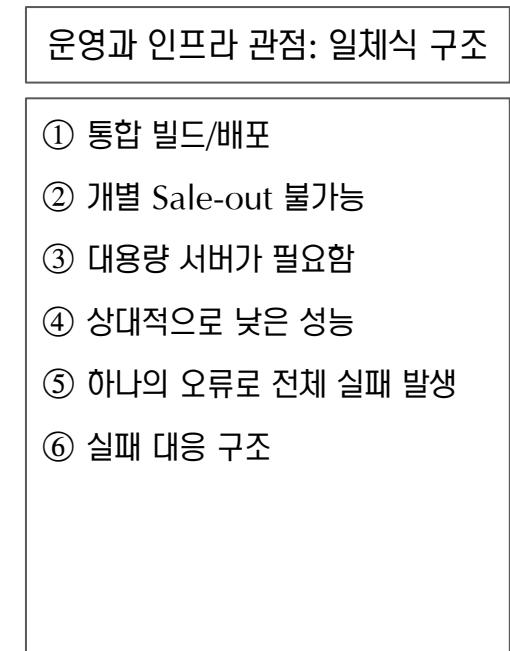
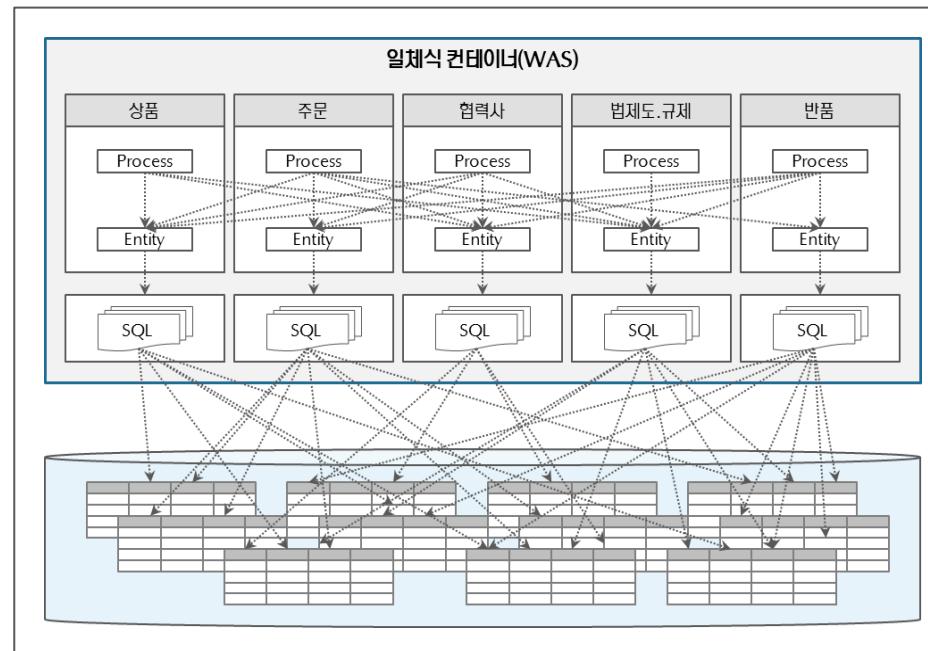
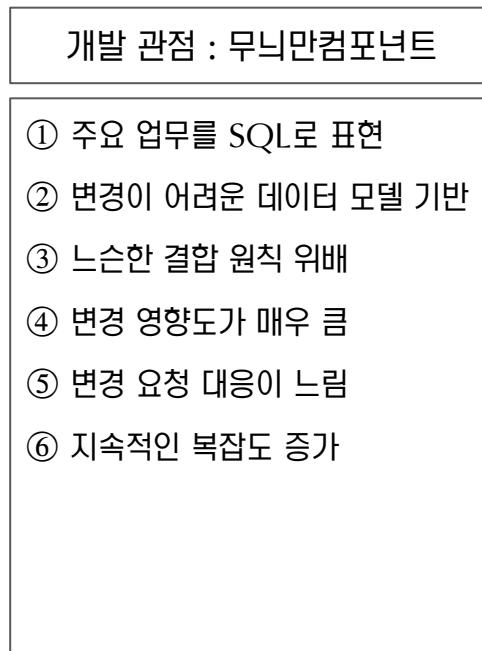
0.5 기술 흐름[6/6] – 단일 컨테이너와 통합DB의 복잡도

- ✓ 컴포넌트 의존관계를 통제하지 못하면 쉽게 통제불가능 상태가 됩니다.
- ✓ 통합 DB 구성에 따른 복잡도는 일정 수준을 넘어서면 관리가 어려운 상태가 됩니다.



0.6 Why MSA? (1/5)

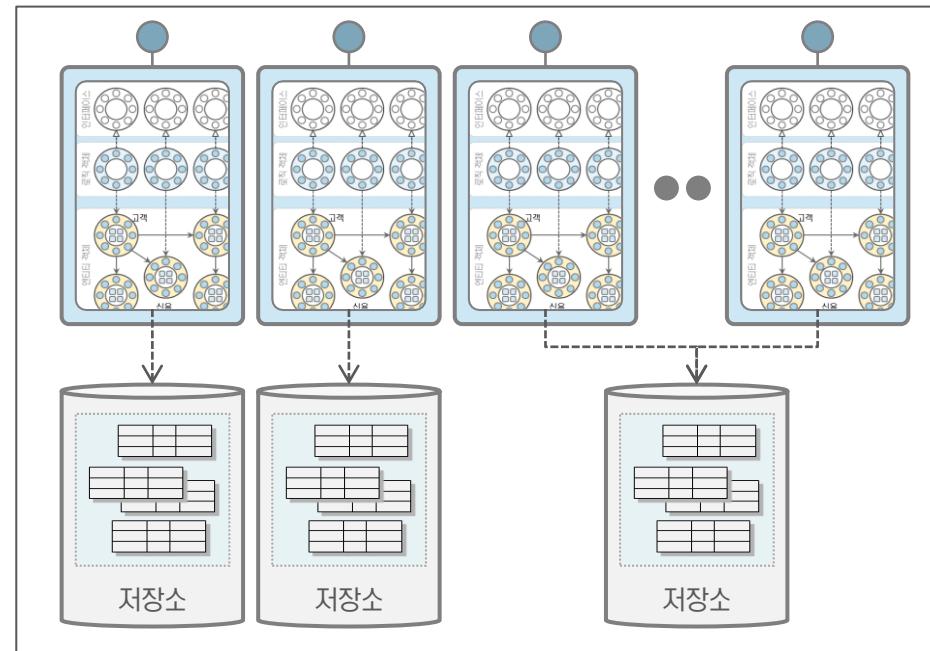
- ✓ 현재 국내에서 개발되어 운영되는 대부분의 시스템은 무늬만 컴포넌트에 일체식 배포 구조를 갖고 있습니다.
- ✓ 무늬만 컴포넌트는 내부에 로직을 갖지 않고 대부분의 로직을 SQL과 데이터 모델로 표현합니다.
- ✓ 일체식 컨테이너와 통합 DB는 운영 중에 성능, 배포, 실패, 등과 관련 여러 가지 문제를 보여 주었습니다.



0.6 Why MSA? (2/5)

- ✓ CBD는 업무를 담을 모듈을 설계할 때 컴포넌트 단위로 설계하려는 접근방법입니다.
- ✓ SOA와 MSA는 업무 컴포넌트를 배포하고 제공할 때 어떤 구조로 할 것인가를 결정하는 접근 방법입니다.
- ✓ 잘 설계된 컴포넌트를 마이크로서비스로 형식으로 구성하여 배포하고 제공합니다.

개발 관점 : 컴포넌트
① 기술 중립적인 컴포넌트 설계
② 업무 기준 컴포넌트 의존성 관리
③ 도메인-드리븐 디자인 원칙 준수
④ 무늬만 컴포넌트 경계
⑤ 컴포넌트 단위 데이터 모델
⑥ 복잡도 조절과 느슨한 결합

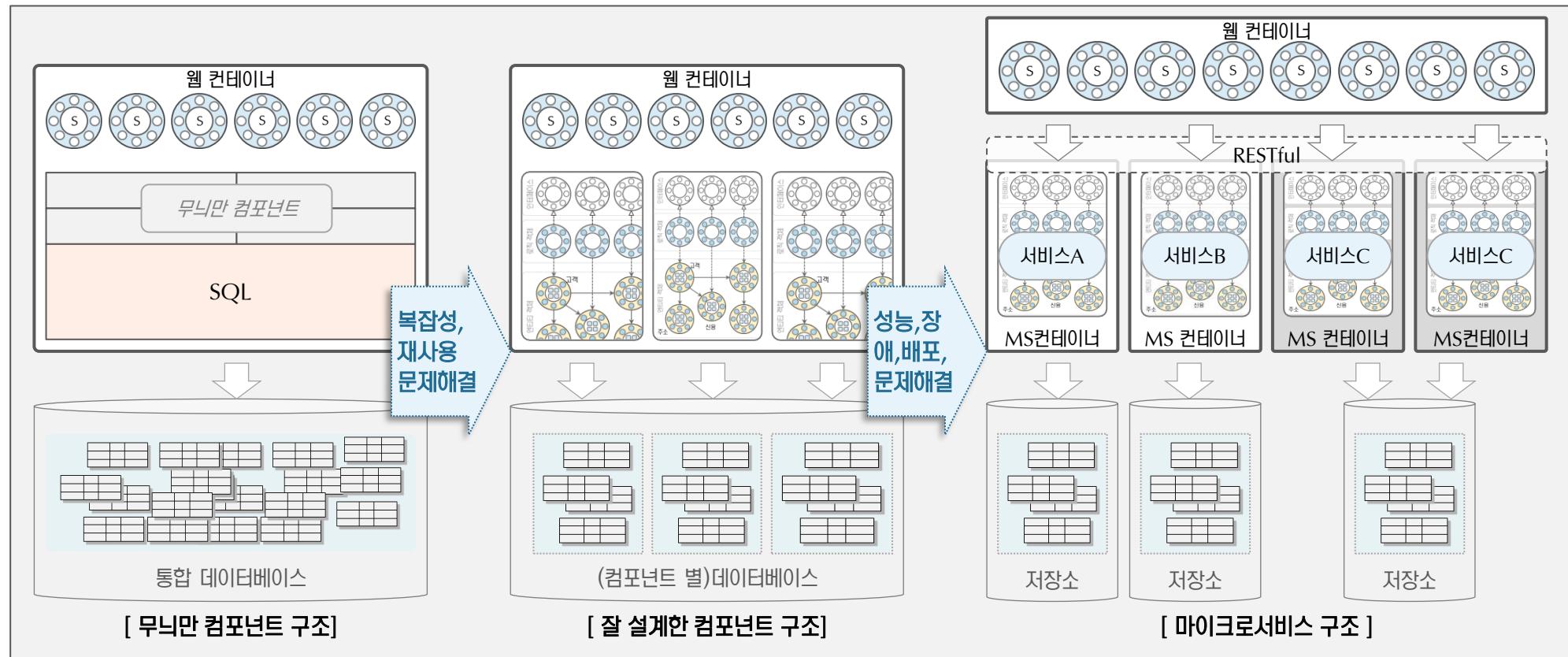


운영과 인프라 관점: MSA
① 컴포넌트를 마이크로 서비스화
② 서비스 단위로 개선/빌드/배포
③ 서비스 별 Scale-out
④ 서비스 별 성능 확장
⑤ 서비스 별 데이터 저장소 분리
⑥ 실패 대응 구조

[컴포넌트 기반 마이크로 서비스 아키텍처 구조]

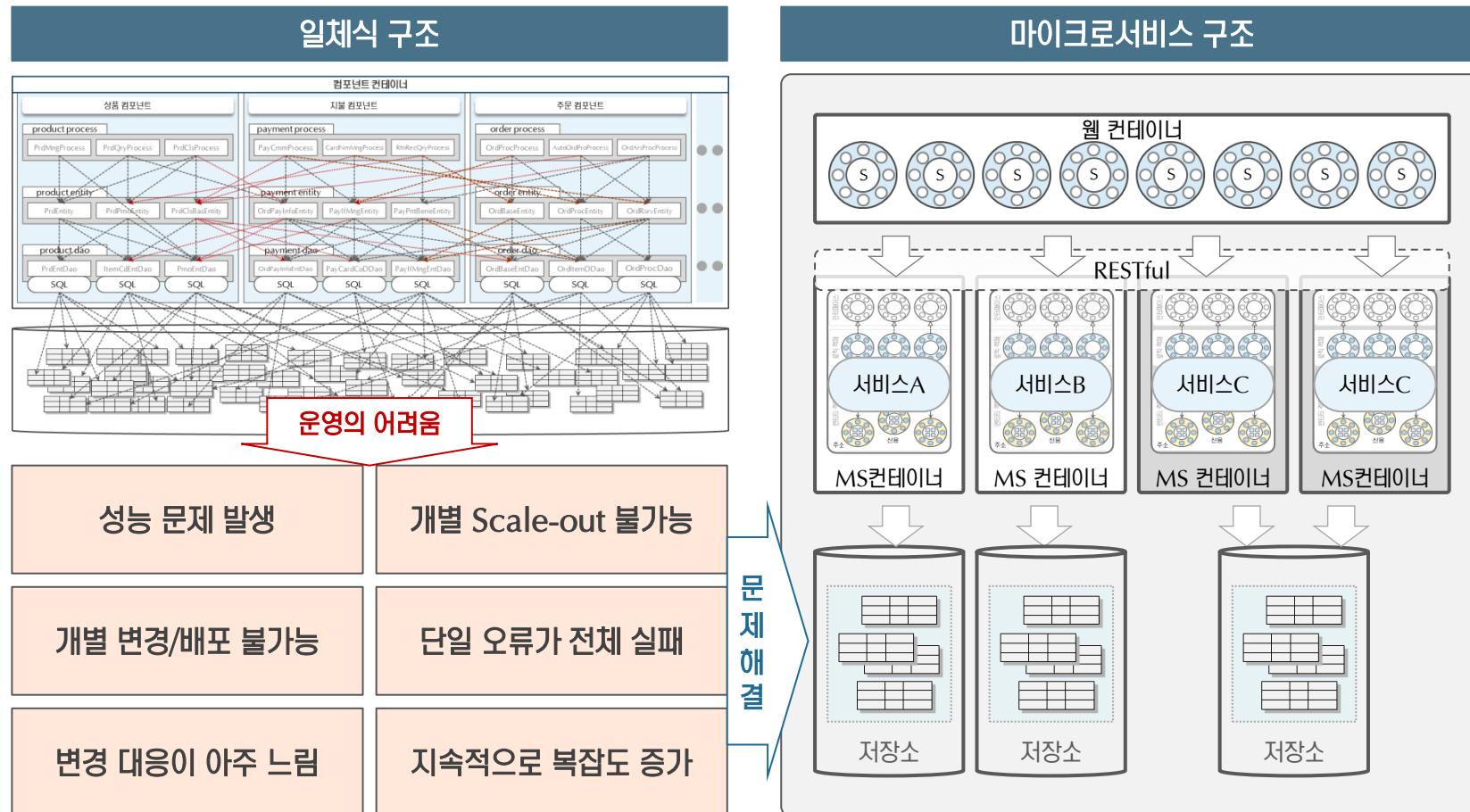
0.6 Why MSA? (3/5)

- ✓ 트랜잭션 스크립트(무늬만 컴포넌트) 방식의 시스템 구성에서 로직은 SQL 문과 DB모델에 집중되었습니다.
- ✓ 컴포넌트 방식의 시스템 구성은 SQL 문과 DB모델에 놓인 업무 로직으로 컴포넌트로 잘 나누어 배치했습니다.
- ✓ MSA는 업무 로직을 마이크로 서비스화 하고, 데이터 저장소를 서비스 단위로 분리하여 배치하였습니다.
- ✓ 그 결과, 업무 단위 독립성을 확보하여, 개별적으로 업데이트하고 배포하고, Scale-out 할 수 있게 되었습니다.



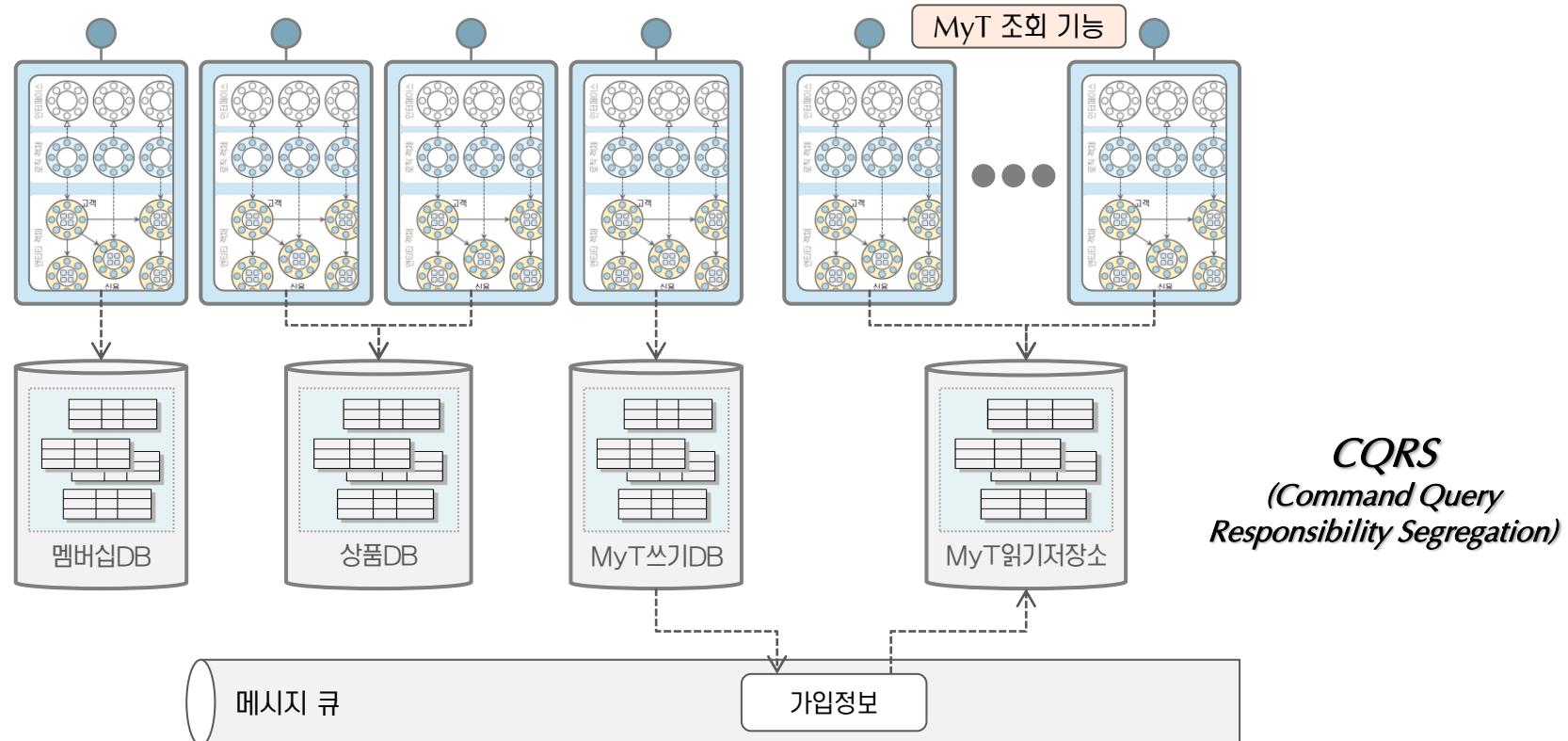
0.6 Why MSA ? (4/5)

- ✓ 대다수의 시스템이 무늬만 컴포넌트를 가진 일체식(monolithic) 구조로 설계되어 있습니다.
- ✓ 잘 설계된 마이크로 서비스 구조로 설계를 하면 기존 시스템의 문제를 모두 해결할 수 있습니다.
- ✓ MSA 기반 시스템은 언제든 필요할 경우 클라우드로 바로 이전할 수 있습니다.



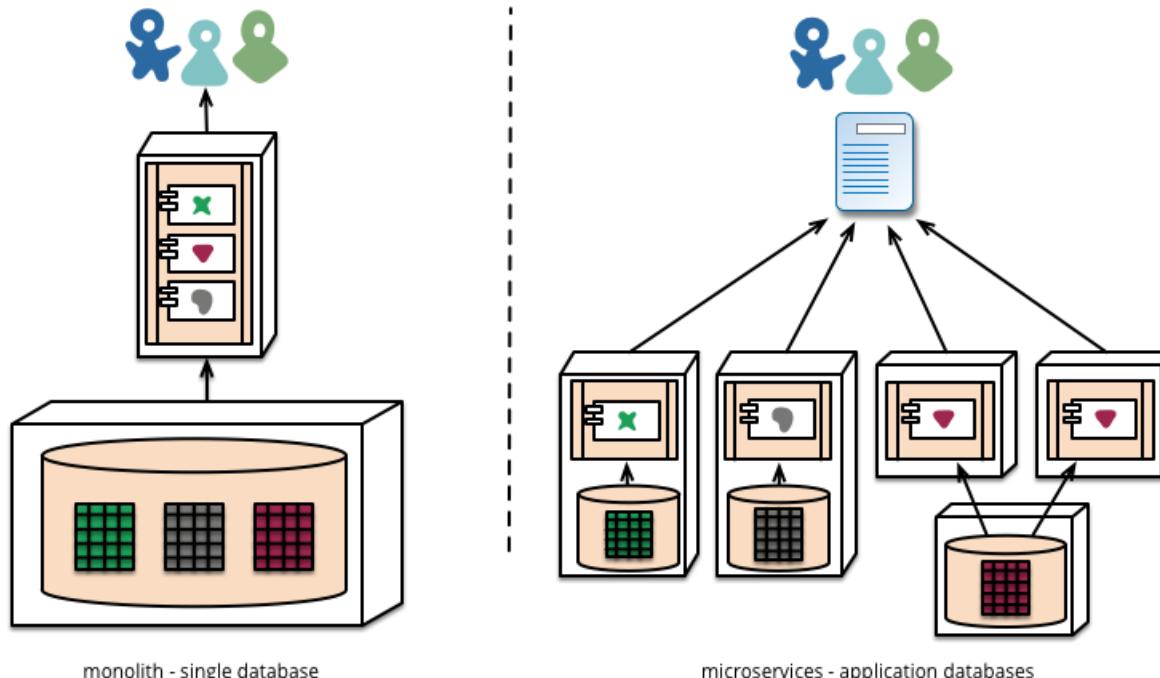
0.6 Why MSA ? (5/5)

- ✓ 마이크로 서비스는 개별 컨테이너 구조로 빠른 배포와 기능별 최적화가 가능합니다.
- ✓ CQRS 아키텍처 패턴은 읽기, 쓰기 기능을 별도의 서비스로 구성하여 상황에 적합한 스케일을 지원합니다.
- ✓ 각 서비스 별 부하에 따른 동적인 Scale-out 은 빠른 Performance를 보장하여 줍니다.



0.7 요약

- ✓ 기업 IT 시스템의 문제를 Microservice와 API로 문제를 풀어 보려 합니다.
- ✓ 2016년 현재의 기술로 문제를 풀 수 있는 방법과 사례는 충분한데, 그 길을 선택하기는 쉽지 않아 보입니다.
- ✓ API와 API를 제공하는 주체로써의 Microservice를 살펴봅니다.



[0/0/0] 출처: <http://martinfowler.com/articles/microservices.html>



1. 비즈니스 서비스 이해

-
- 1.1 서비스 동상이동
 - 1.2 기업의 IT 영역
 - 1.3 IT 기술 트랜드
 - 1.4 비즈니스 분석과 시스템 연계 – 액티비티 정의서 분석

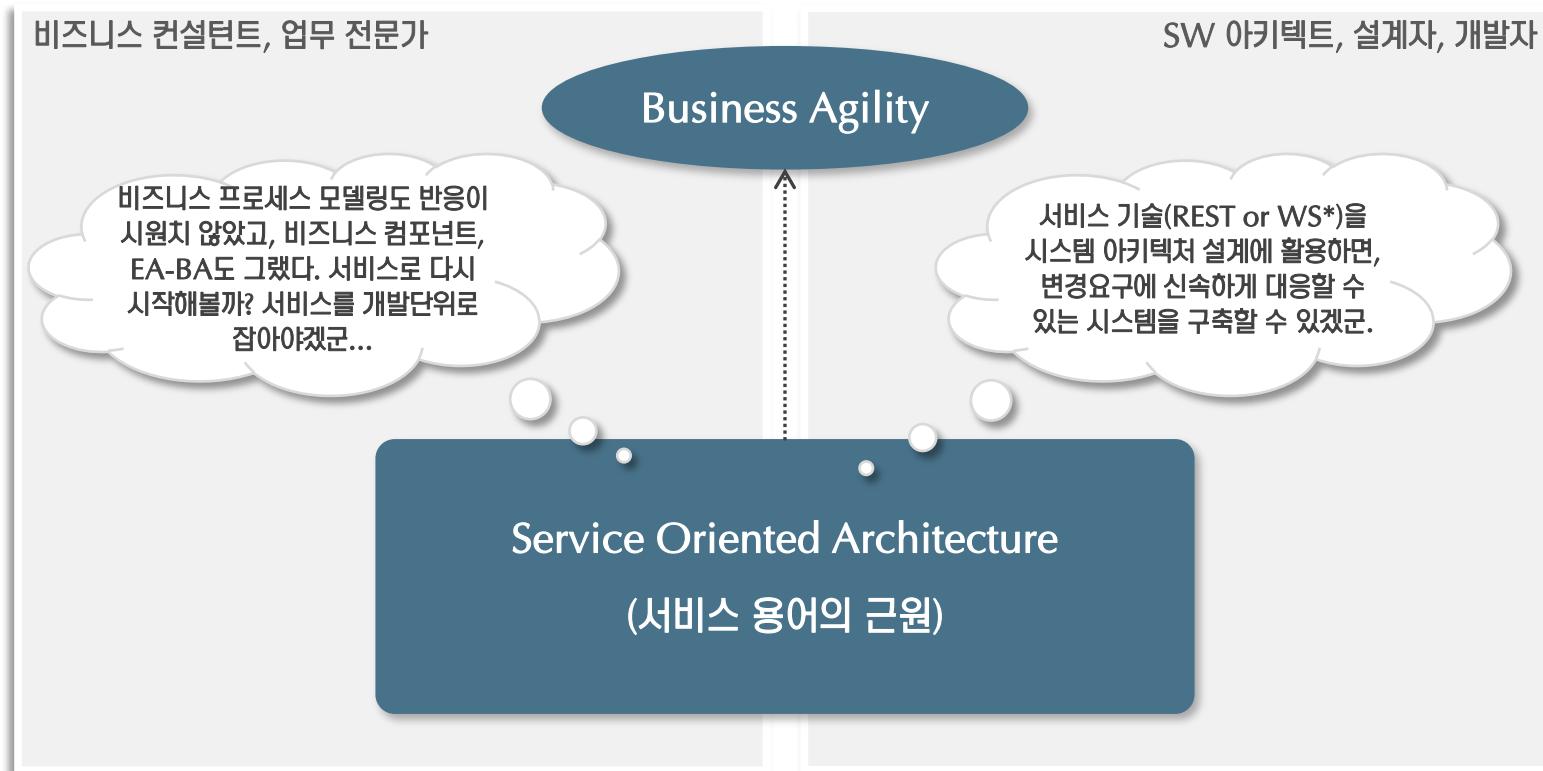
1.1 서비스 동상이동 [1/2]

- ✓ 비즈니스 분석 영역의 작업 결과가 시스템 구축 영역으로 제대로 연결되는 결과를 보기 힘들었습니다.
- ✓ 벤더와 컨설팅 회사들은 컴포넌트 이후에 새로운 키워드가 필요 했고, SOA를 키워드로 설정하였습니다.
- ✓ 비즈니스 컨설턴트는 SOA에서 “서비스”를 추출하였고, 벤더는 서비스 발행 플랫폼을 만들어서, 고객을 설득하기 시작하였습니다. 동상이동이 시작되었습니다.



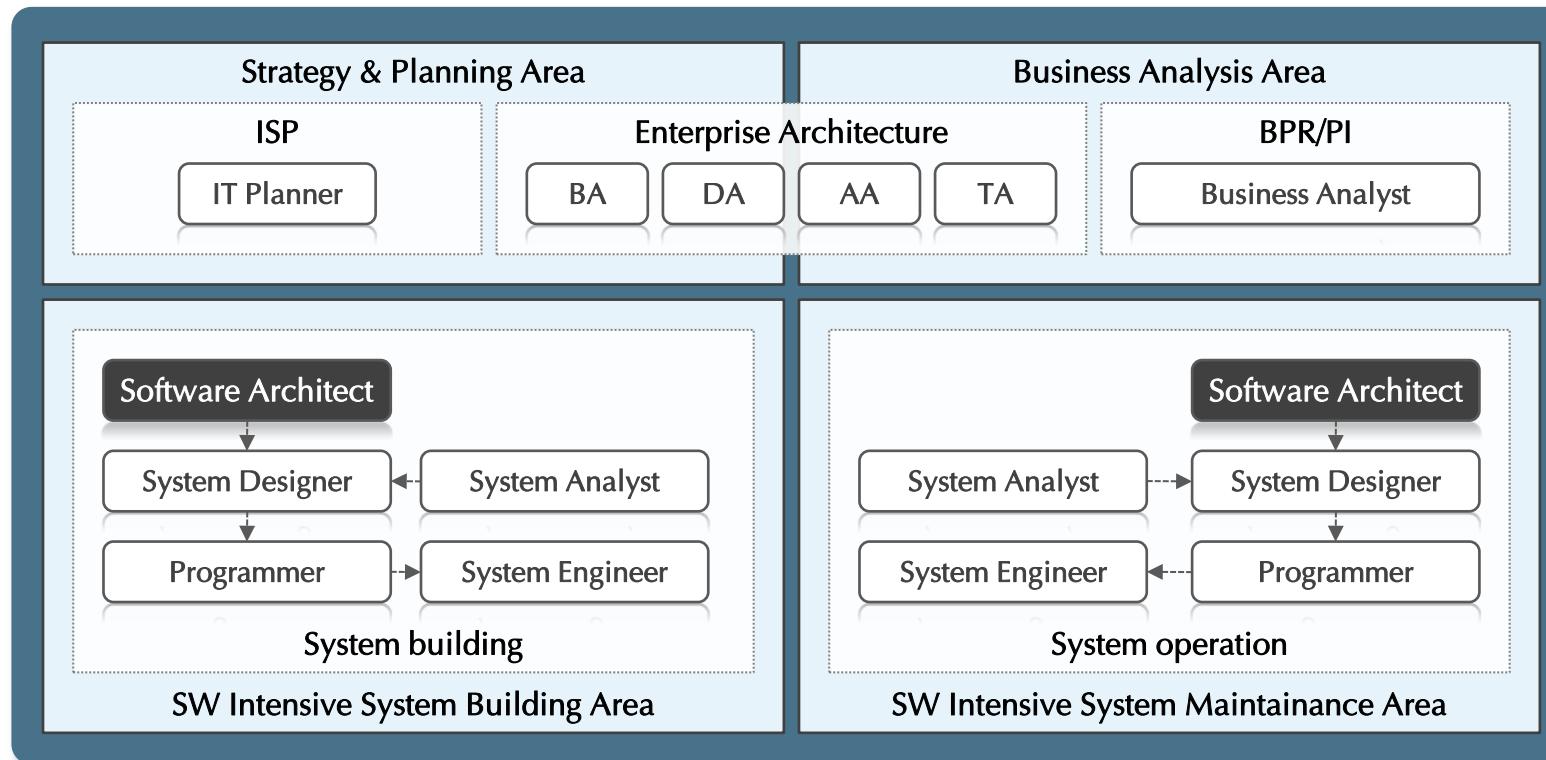
1.1 서비스 동상이동 [2/2]

- ✓ 비즈니스 분석과 시스템 구축 영역 담당자의 최대 목표는 Business Agility를 높이는 것입니다.
- ✓ 비즈니스 영역에서 여러 가지 가지 접근 방법을 시도해 왔습니다, BPR/PI → 비즈니스 컴포넌트 → EA → 서비스
- ✓ 시스템 구축 영역에서도 기술을 지속적으로 발전시켜 왔습니다, 절차지향 → 객체지향 → 컴포넌트 → 서비스



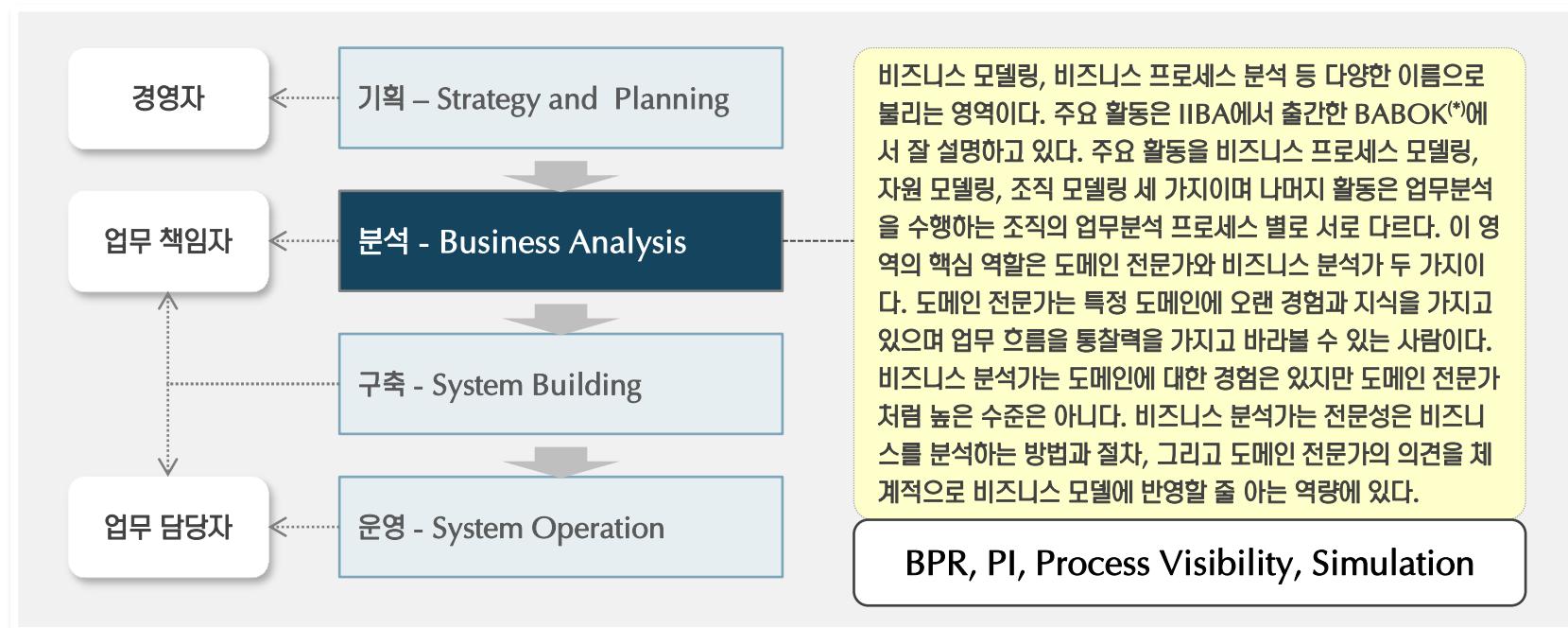
1.2 기업의 IT 영역 (1/6) – 활동과 역할

- ✓ 각 영역별로 주요 활동과 이에 따른 역할이 정의되어 있습니다.
- ✓ Enterprise Architecture 활동의 경우, 기획과 분석 두 영역에 걸쳐 있으며 ISP와 BRR/PI 활동과도 중복이 일부 존재합니다. 구축활동과 운영활동 영역에서 역할 정의는 비슷하지만, 역할 수행의 특성이 서로 다릅니다.



1.2 기업의 IT 영역 [2/6] – 분석영역

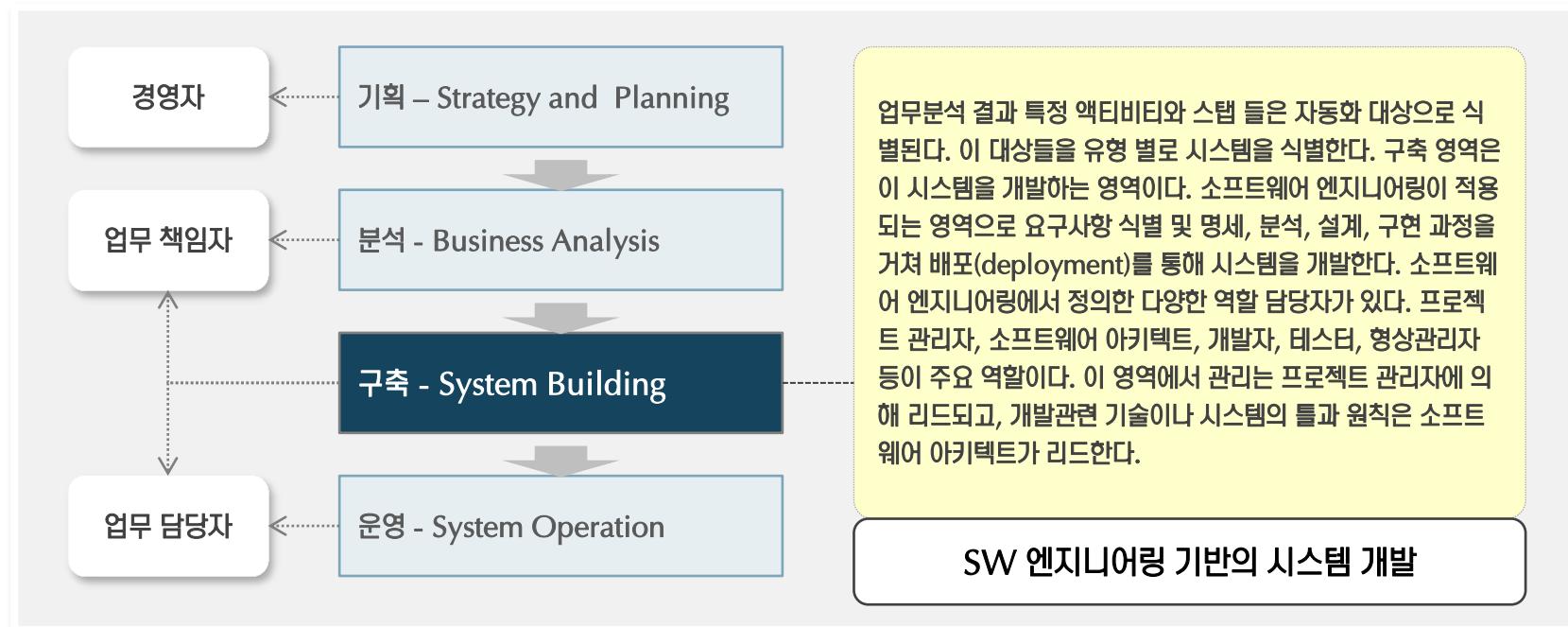
- ✓ 비즈니스 분석 활동은 기업의 비즈니스 프로세스 가시화(visibility)나 개선에 초점을 둡니다.
- ✓ SW엔지니어링처럼 엔지니어링 영역이 아니므로, 모델링에 대한 자유도가 매우 높아 조직마다 서로 다른 정의와 접근방법을 가집니다. 비즈니스 분석을 위한 모델링 활동은 대체로 비즈니스 프로세스, 비즈니스 엔티티를 비롯한 6대 모델을 중심으로 진행합니다.



- BABOK(Business Analysis Body of Knowledge) – International Institute of Business Analysis, www.theiiba.org

1.2 기업의 IT 영역 [3/6] – 구축영역

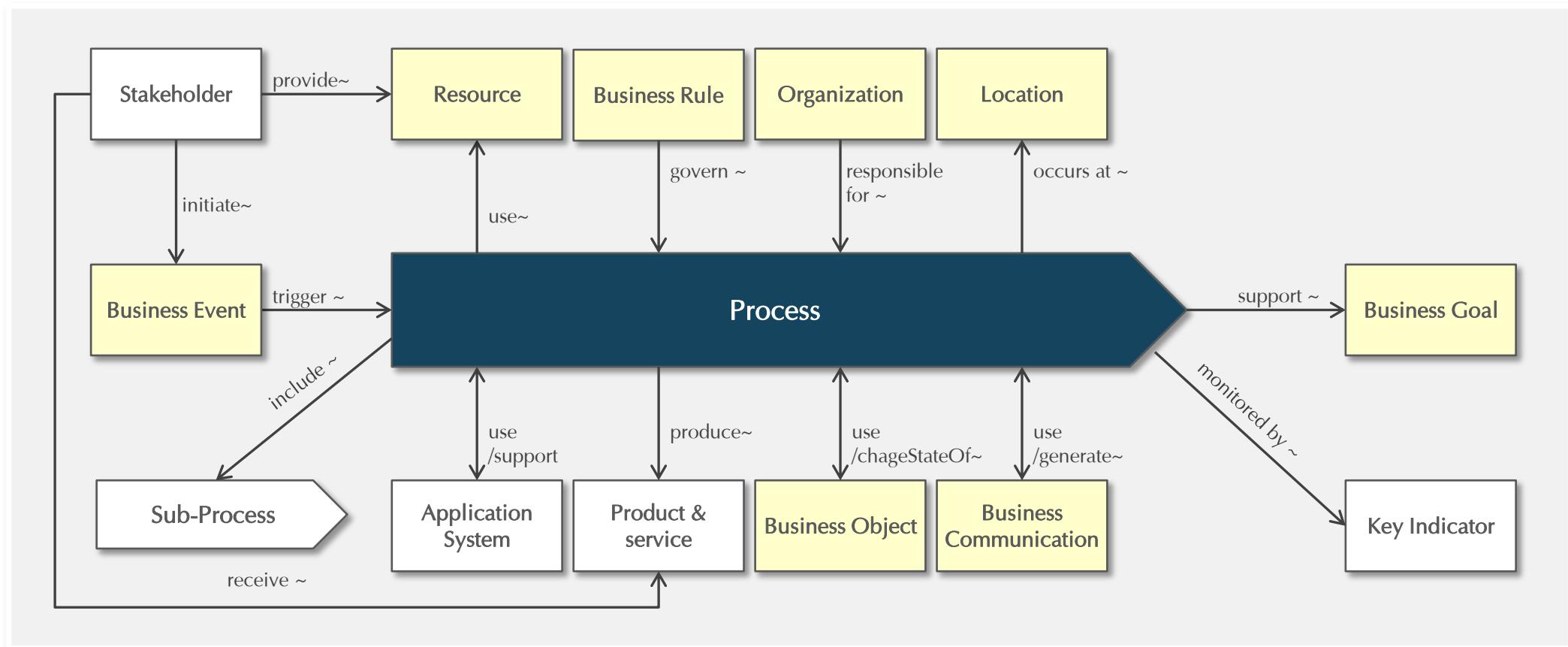
- ✓ 비즈니스 분석에서 식별된 자동화 시스템을 개발하는 영역으로써 SW 엔지니어링 역량과 올바른 기능 구현에 초점을 둡니다 . SW 엔지니어링을 기반으로 활동하며, UML, UP 등과 같은 표준을 가급적 준수하여야 하므로 활동 자유도가 높지 않은 영역입니다.
- ✓ UML 표준의 기본을 이루고 있는 객체지향 개념을 근간으로 컴포넌트 모델까지 확장이 가능한 영역입니다.



- BABOK(Business Analysis Body of Knowledge) – International Institute of Business Analysis, www.theiiba.org

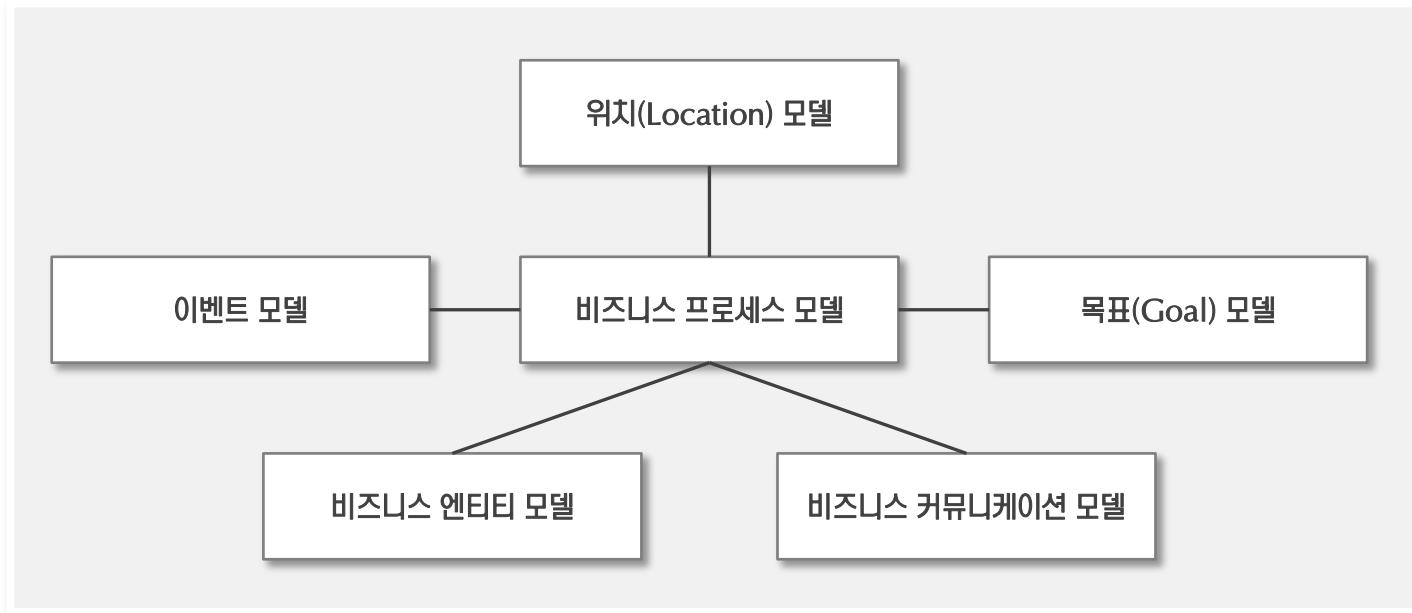
1.2 기업의 IT 영역 (4/6) – 비즈니스 분석 컨텍스트

✓ 비즈니스 분석 영역은 기업활동이 이루어지는 영역으로 다양한 역할, 자원, 개념, 시스템 등이 프로세스를 중심으로 구성됩니다. 비즈니스 모델링은 비즈니스 영역에서 주목할만한 요소들에 대한 모델링으로 접근방법에 따라 다양할 수 있습니다. 전통적인 비즈니스 모델링은 비즈니스 컨텍스트 안의 정보, 개념, 자원, 작업 등을 최적화하기 위해 다양한 방법을 제기하였습니다.



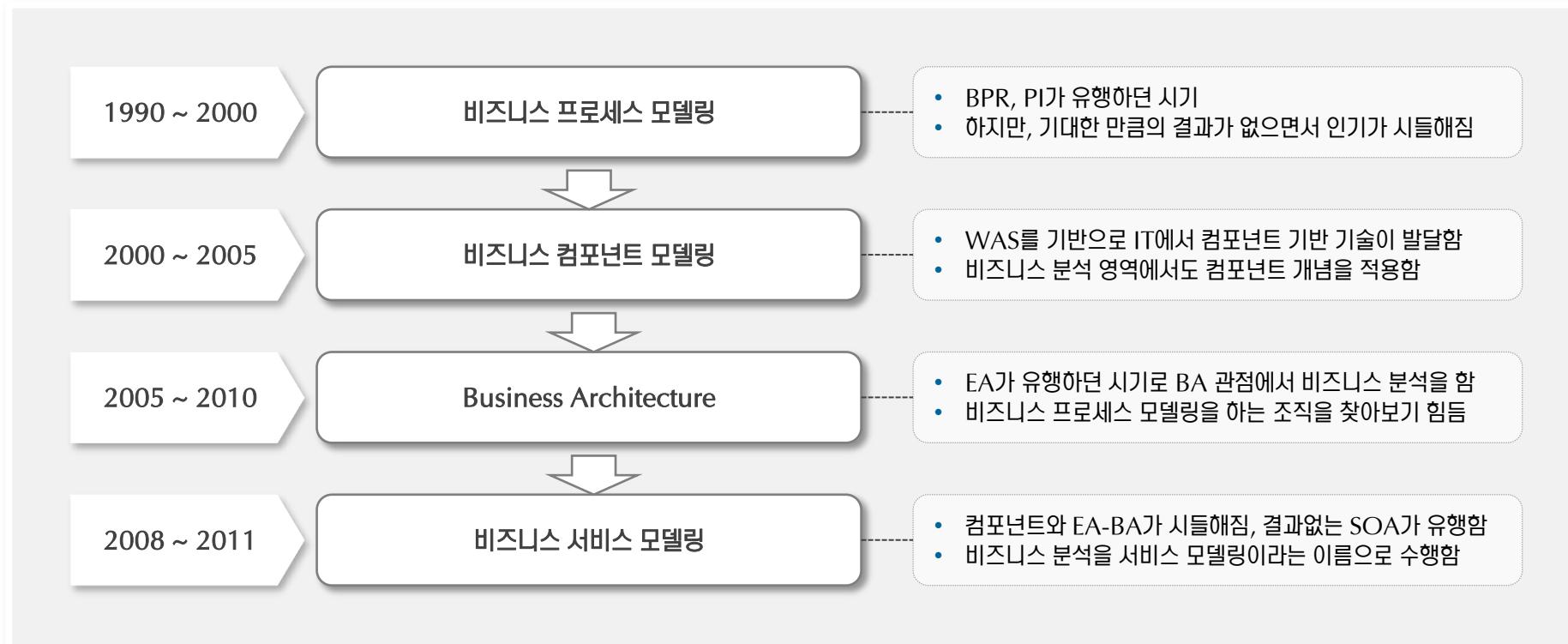
1.2 기업의 IT 영역 (5/6) – 비즈니스 모델링 리뷰 : BPR 6대 모델

- ✓ 비즈니스 컴포넌트 모델링, 비즈니스 서비스 모델링이라는 용어를 사용하기 이전에 주로 수행했던 모델링은 주로 여섯 가지 내외이며, 비즈니스 컴포넌트/서비스 모델링의 경우에도, 프로세스, 이벤트, 비즈니스 엔티티 등은 모델링 활동에 포함되어 있습니다.
- ✓ 비즈니스 분석 컨설팅 회사들은 저마다의 접근방법을 가지고 있습니다.



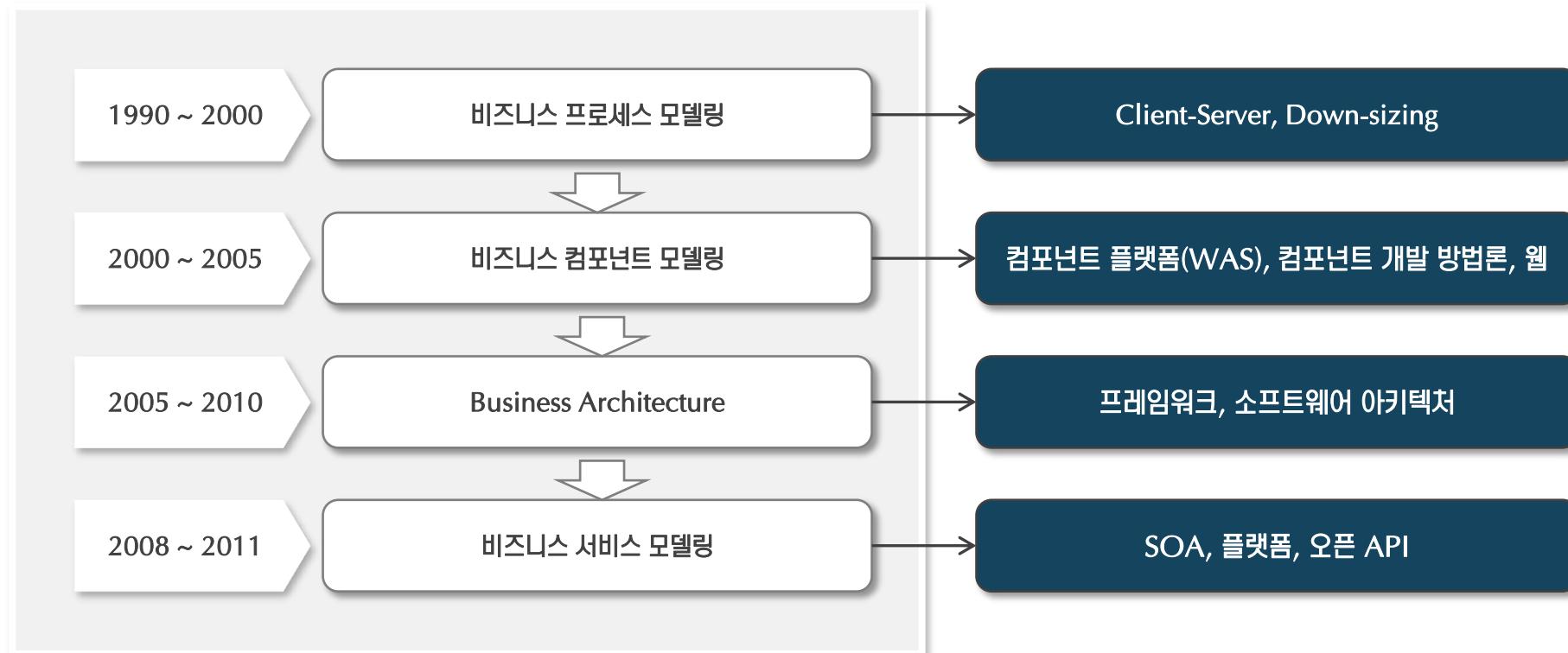
1.2 기업의 IT 영역 [6/6] – 비즈니스 모델링 트랜드

- ✓ 비즈니스 영역에서 표현하는 기업의 활동이나 개념은 예나 지금이나 큰 변화가 없습니다.
- ✓ 하지만 비즈니스 활동을 지원하는 Application System의 비중이 높아지면서, IT 기술의 발전이 모델링에 영향을 주기 시작했습니다.



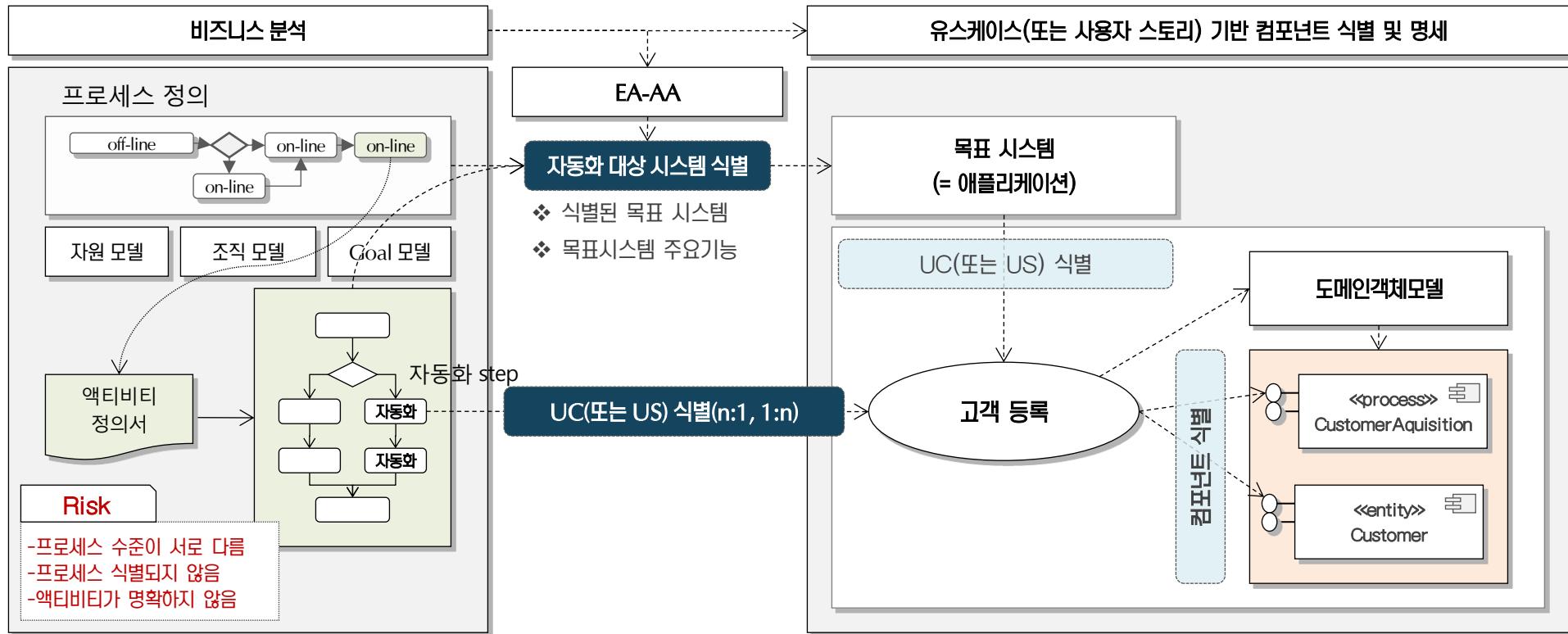
1.3 IT 기술 트랜드

- ✓ 비즈니스 영역과 시스템 구축 영역은 서로 다른 역할 담당자의 활동을 요구하는 독립적인 영역입니다.
- ✓ IT 기술의 급격한 변화는 비즈니스 분석 활동에 지속적으로 영향을 준다는 사실을 알 수 있습니다.
- ✓ 특히 SOA는 2010년 까지는 거의 영향을 미치지 못하는 상태였으나, 2011년 이후 많은 프로젝트 영향을 주고 있음을 알 수 있습니다.



1.4 비즈니스 분석과 시스템 연계 – 액티비티 정의서 분석

- ✓ 비즈니스 분석의 액티비티 정의서의 자동화 스텝은 시스템 유스케이스의 근거가 됩니다.
- ✓ 하나의 자동화 스텝은 여러 유스케이스로 매핑되고, 그 반대의 경우도 존재합니다.
- ✓ 프로세스 모델의 액티비티는 비즈니스 유스케이스(BUC)와 대체로 일치하며, 하나의 BUC는 3~11개의 시스템 UC로 매핑됩니다.



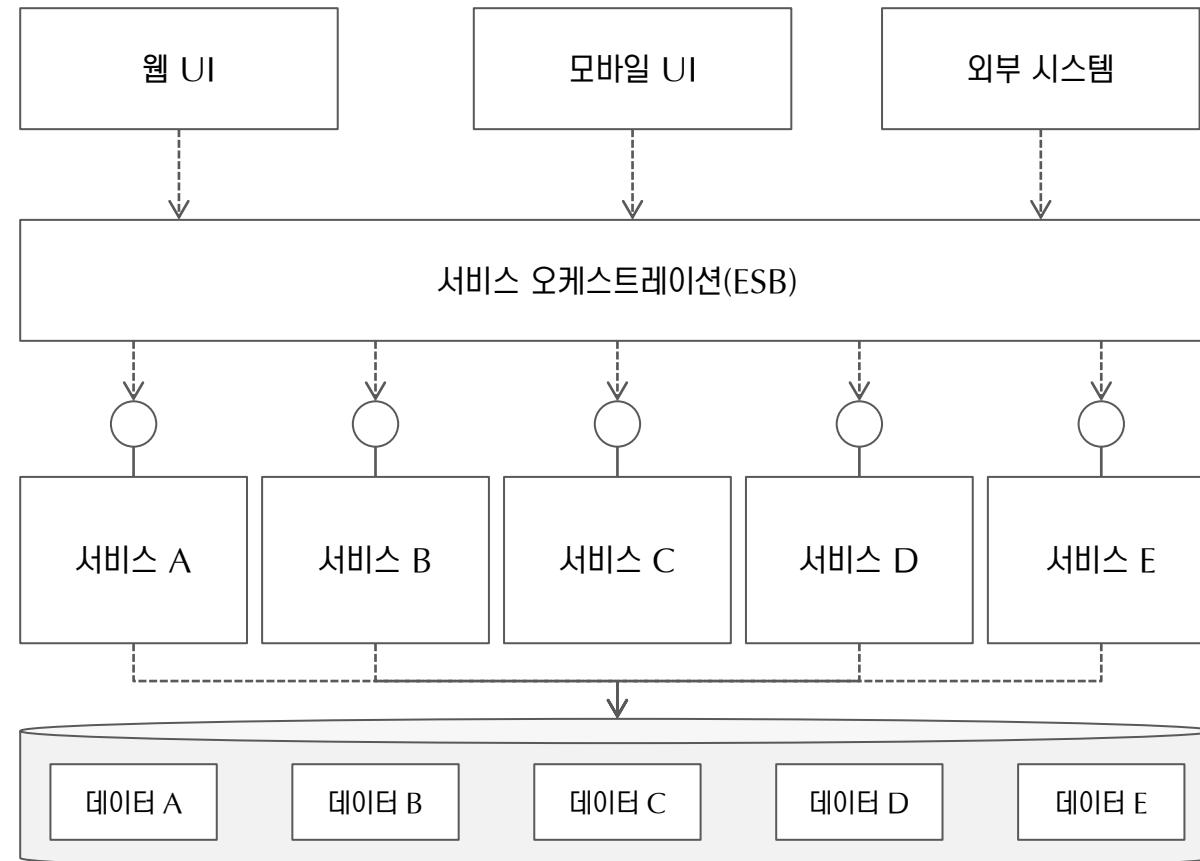


2. SOA 이해

- 2.1 SOA 이미지
- 2.2 SOA 관련 용어
- 2.3 SOA와 레이어
- 2.4 SW 설계 기술
- 2.5 서비스와 컴포넌트
- 2.6 컴포넌트의 한계

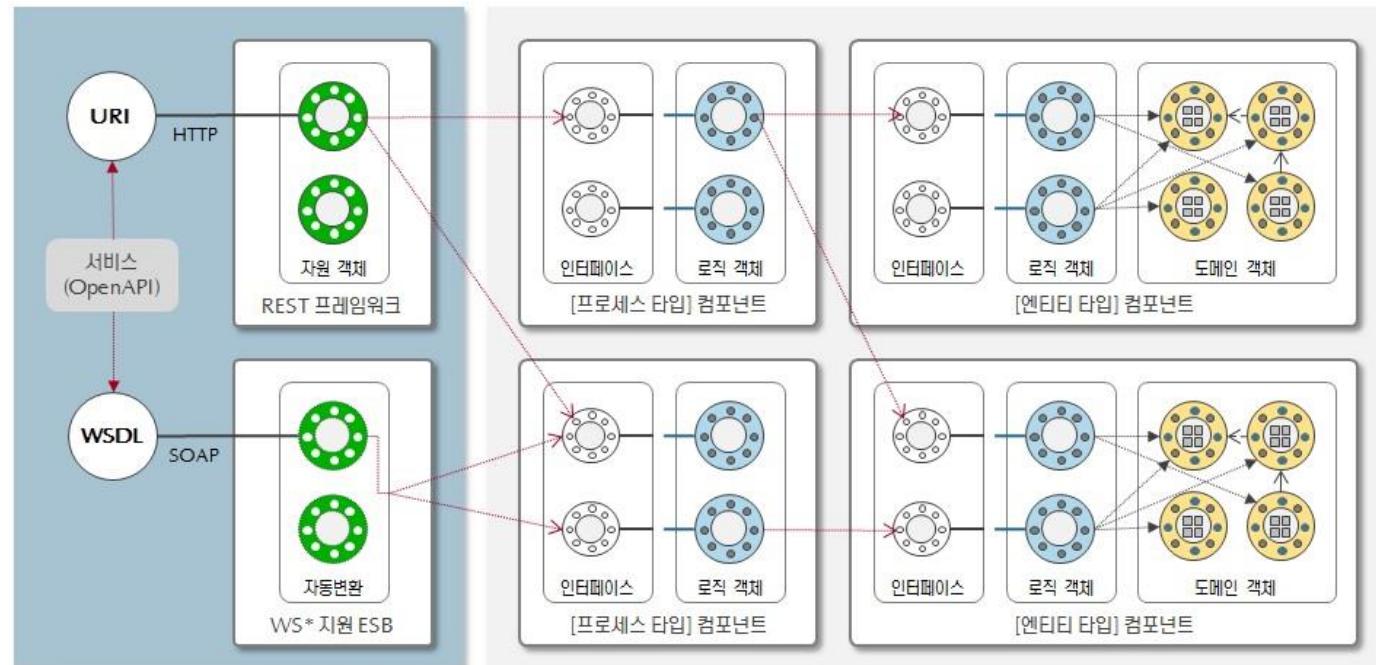
2.1 SOA 이미지 (1/3)

- ✓ 서비스는 WS-* 기술로 발행(publishing)하며, Monolith 컨테이너에 배포하여 관리합니다.
- ✓ ESB의 서비스 오케스트레이션을 이용하여 사용자나 외부시스템에 서비스를 제공합니다.
- ✓ 서비스 데이터는 물리적으로 하나의 DB 인스턴스이지만 논리적으로는 분리된 각자 서비스를 위한 테이블입니다.



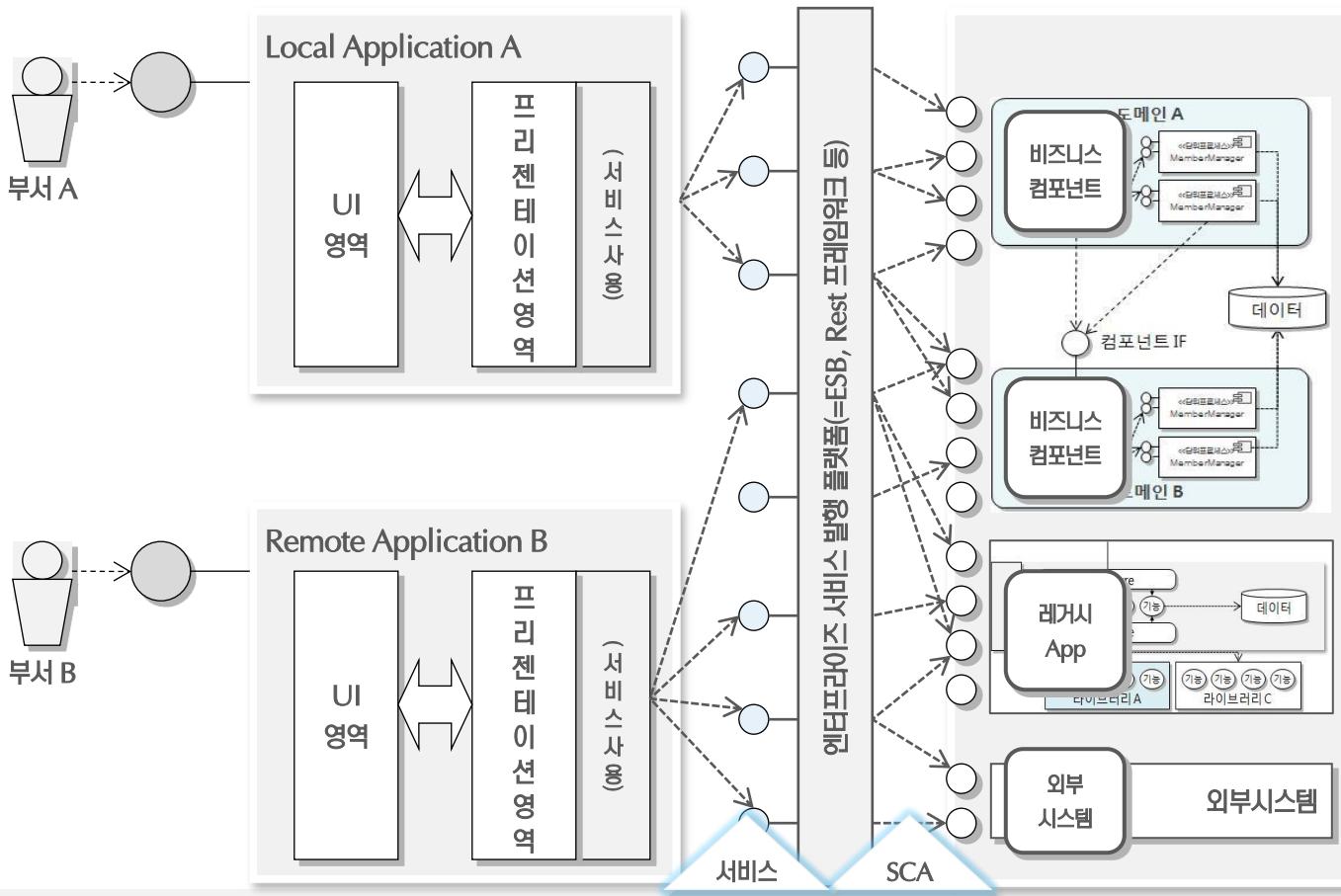
2.1 SOA 이미지 (2/3)

- ✓ SOA에서 비즈니스 로직을 표현하는 모듈은 컴포넌트이며, 비즈니스 로직을 외부로 제공할 때 표현하는 방식은 서비스입니다.
- ✓ 컴포넌트는 여러 유형의 객체들로 구성되며, 컴포넌트 간의 복잡한 관계규칙을 준수하여야 합니다.
- ✓ SOA는 컴포넌트 뿐만 아니라 기존의 레거시 애플리케이션도 연결할 수 있습니다.



2.1 SOA 이미지 (3/3)

- ✓ 이상적인 형태의 서비스 기반 애플리케이션은 서비스 발행 플랫폼을 기반으로 설계합니다.
- ✓ ESB는 전사의 서비스 자원이 되는 레거시, 컴포넌트, 외부 자원 등을 서비스화하여 외부로 발행하여 줍니다.
- ✓ 다양한 IT 자원을 엑기 위한 표준으로 SCA(Service Component Architecture) 가 있습니다.

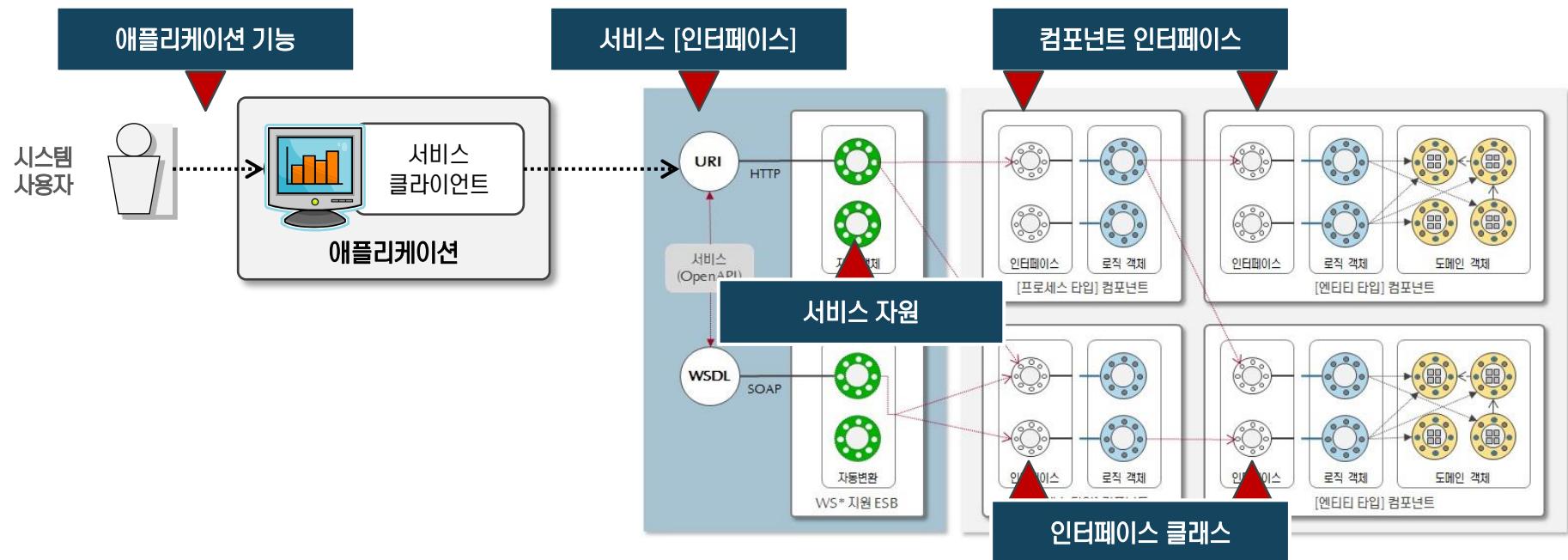


서비스 기반 애플리케이션의 특성

- ❖ 비즈니스 서비스에 상응하는 IT 영역의 서비스 도입으로 비즈니스 민첩성 목표 달성이 용이함
- ❖ 재사용: 서비스는 휘발성이 강하지만 재사용성이 높음
- ❖ 연결: W3C 웹 서비스 표준이나 REST 표준을 사용하여 연결
- ❖ 개념: 가시범위는 iter-enterprise 이므로, 서비스 제공 위치에 상관없이 서비스를 모아 애플리케이션을 구성할 수 있음
- ❖ 언어: 언어 독립적임. 일반 프로그래밍 언어뿐만 아니라 Shell 스크립트, 선언적 언어인 XPDL 등도 가능함
- ❖ 애플리케이션은 서비스를 모아서 구성하는 매우 동적인 시스템 사용단위로 의미가 변경됨

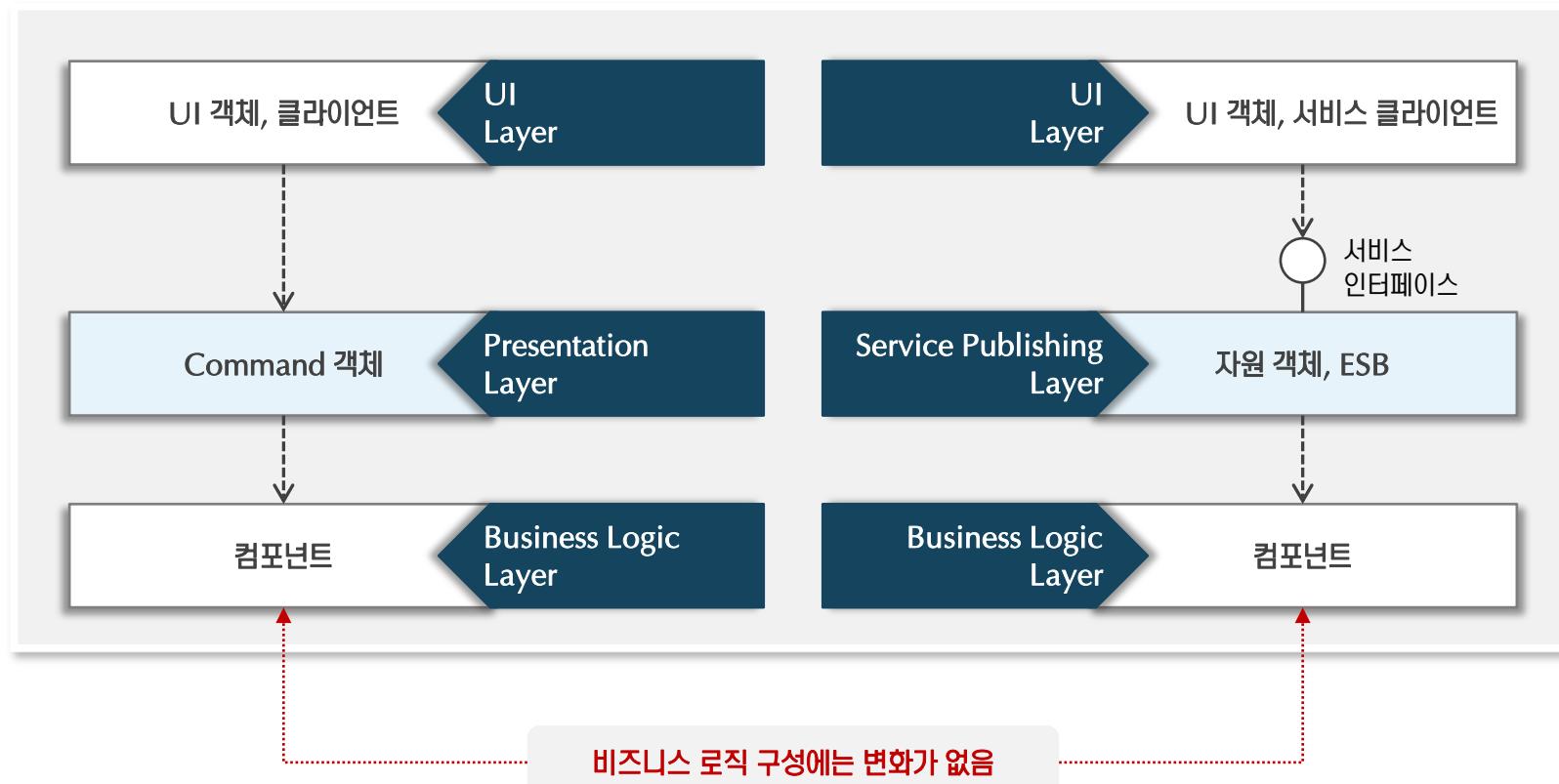
2.2 SOA 관련 용어

- ✓ 기능 – 애플리케이션(또는 SW intensive System)이 사용자들에게 제공하는 것은 기능 또는 시스템 기능이라고 한다.
- ✓ 컴포넌트 인터페이스 – 컴포넌트 내부에 존재하는 로직을 외부로 제공하는 유일한 창구 역할을 하는 장치 또는 클래스이다.
- ✓ 서비스 또는 서비스 인터페이스 – 서비스 기술(REST 또는 WebService)을 사용하여 외부로 발행(publishing)한 인터페이스
- ✓ REST – 서비스 기술 중에 하나로 기존에 검증된 웹 기술을 기반으로 하며, HTTP 명세에 정의된 오퍼레이션을 활용을 잘 활용한다.
- ✓ Web Service – WSDL, SOAP, UDDI 세 가지 표준 명세서와 관련 표준으로 구성된 서비스 기술
- ✓ Service Client – 서비스 기술을 이용하여 발행한 서비스 인터페이스를 사용하는 클라이언트
- ✓ Open API – 조직의 외부에서 누구나 쉽게 접근할 수 있도록 공개한 API로 서비스 기술 뿐만 아니라 다른 기술을 사용할 수도 있다.



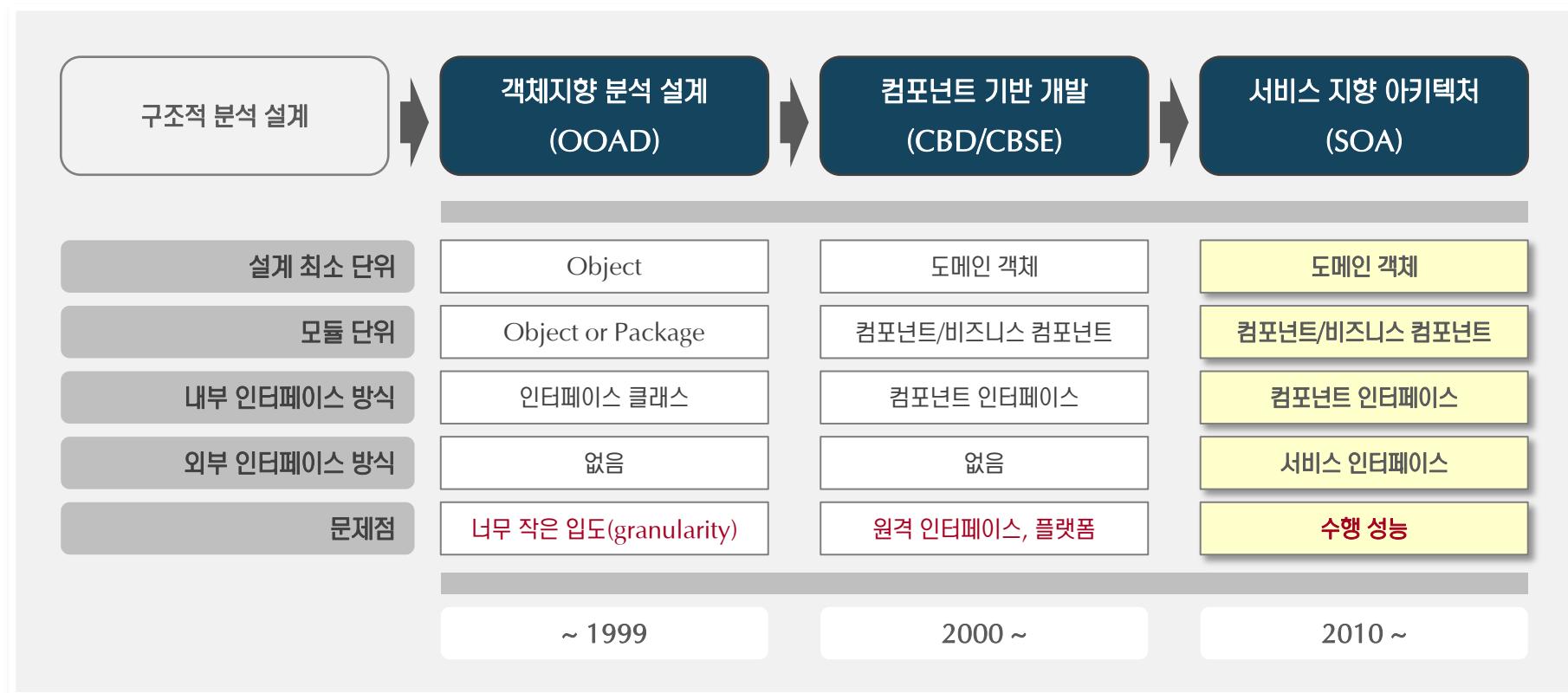
2.3 SOA와 레이어

- ✓ 시스템 구축 영역에서는 서비스라는 용어보다는 SOA라는 용어를 주로 사용하고 있습니다.
- ✓ SOA의 A가 아키텍처를 의미하듯, 주로 아키텍처 설계에 영향을 주고 있습니다.
- ✓ 기능 설계는 여전히 컴포넌트 중심이며, SOA는 인터페이스를 외부로 노출하는 방식에 영향을 줄 뿐입니다.



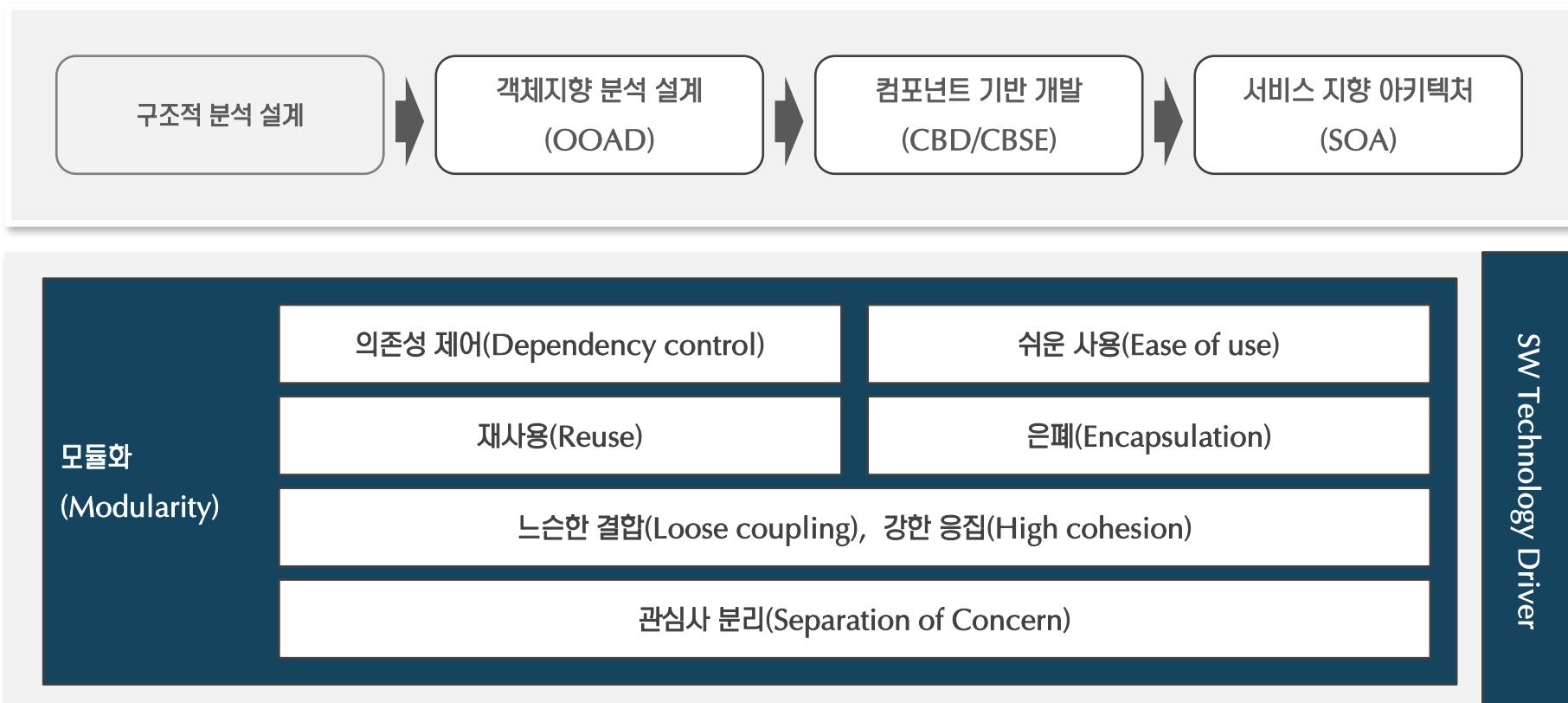
2.4 SW 설계 기술 (1/4) – 트랜드

- ✓ SOA 이전의 기술은 사라지는 것이 아니라, 충분히 성숙한 후에 후속 기술의 바탕이 되는 특성이 있습니다.
- ✓ SOA 시대에도 OOAD는 바탕이 되는 기술로써, 좋은 객체 모델링이 좋은 서비스를 보장합니다.
- ✓ SW 비즈니스 로직을 구성을 위한 단위는 컴포넌트이며, SOA의 서비스는 외부로 인터페이스를 발행(publishing) 하는 단위입니다.



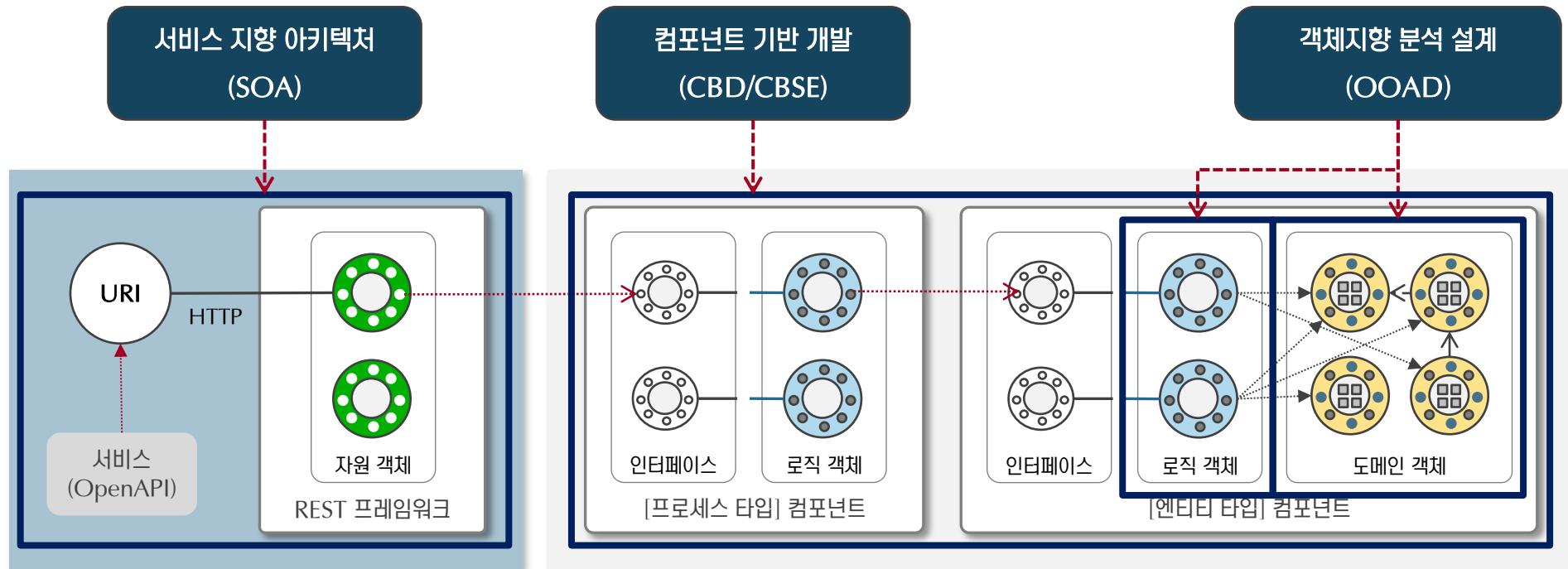
2.4 SW 설계 기술 [2/4] – 기술 발전 동인

- ✓ SW 기술은 유연하며 확장 가능한 모듈을 개발하려는, 즉 모듈화(modularity) 목표를 향해서 움직입니다.
- ✓ SoC, Loose coupling, Encapsulation, Reuse 등의 개념은 상호 연관을 가지고 있습니다.
- ✓ 예를 들면, 재사용을 극대화 했을 경우, 모듈의 크기가 작아지고, 그 결과로 모듈 간의 관계가 복잡해져, 쉬운 사용이 어려워집니다.



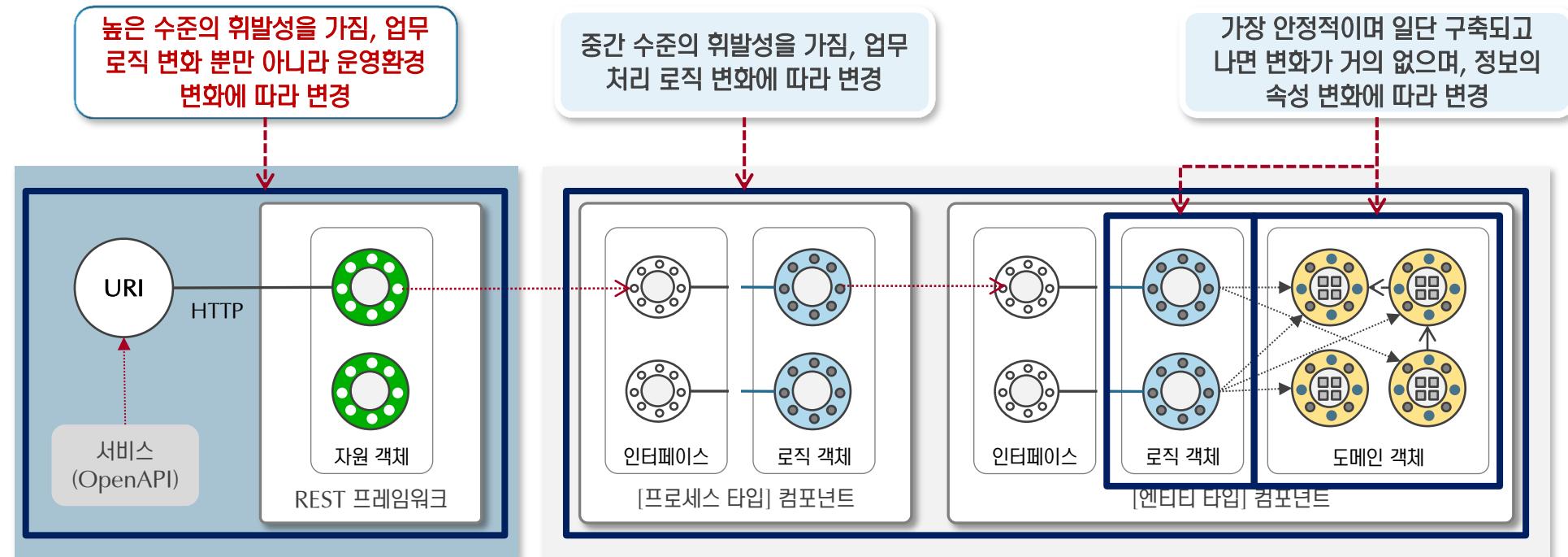
2.4 SW 설계 기술 (3/4) – 설계 기술과 대상

- ✓ 객체지향 기술은 도메인 객체와 로직 객체를 구성할 때 적용함 ← 도메인의 본질을 담아내는 영역
- ✓ 컴포넌트 기술은 객체들을 유형별로 나누고 묶을 때 적용함 ← 업무 단위로 묶어내는 꾸러미
- ✓ 서비스 기술은 컴포넌트의 인터페이스를 외부로 제공할 때 적용함 ← 외부 사용자 관점에서 내부의 인터페이스를 재구성하는 영역



2.4 SW 설계 기술 (4/4) – 휘발성(volatility)

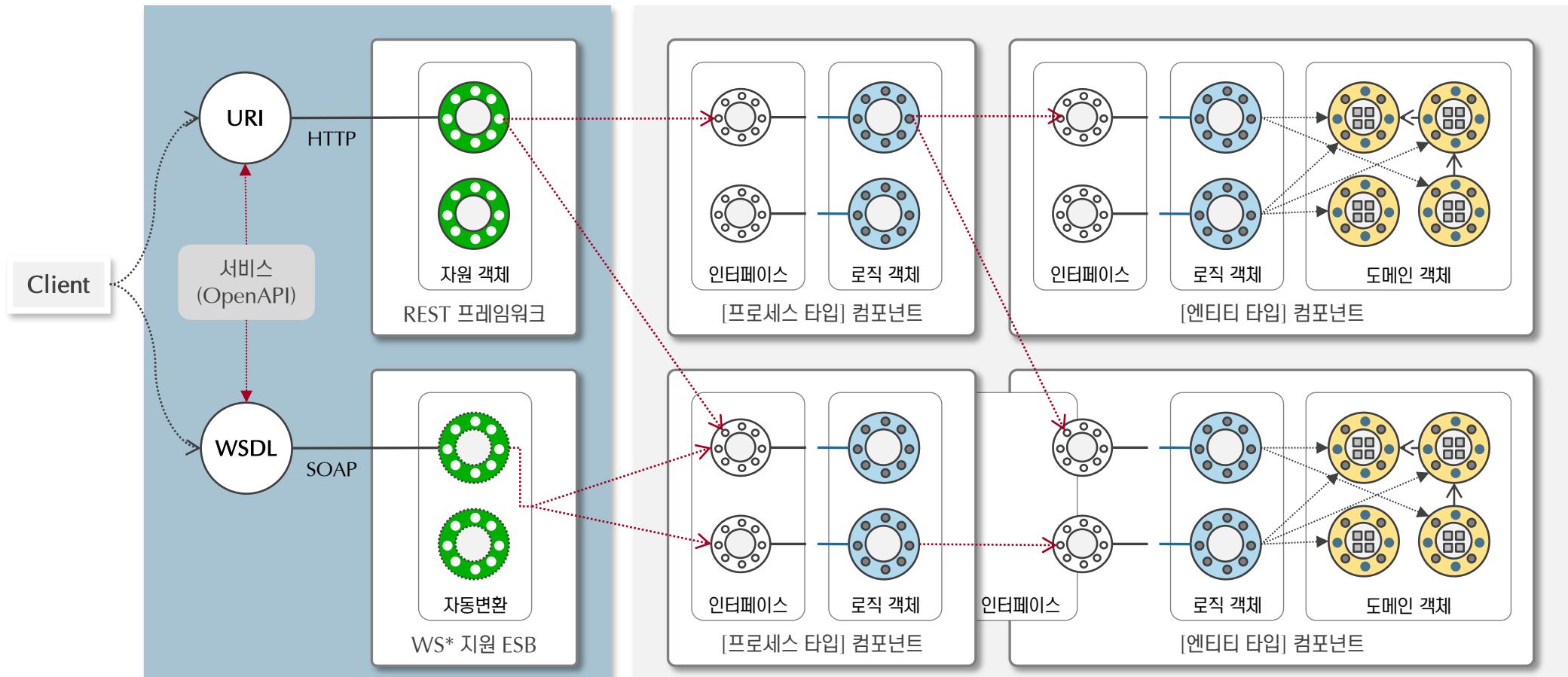
- ✓ 비즈니스 민첩성 목표를 달성하기 위해 시스템을 다양한 변경 요구를 수용하여야 합니다.
- ✓ 변경요구에도 불구하고 가장 안정적인 영역은 엔티티 타입 컴포넌트 영역이며, 프로세스 타입 컴포넌트는 중간 정도 휘발성을 가집니다.



시스템 구축 시, 휘발성이 강한 영역에 비즈니스 로직을 담으면, 변경 요구가 발생할 경우, 로직을 잃어버릴 수 있으며, 그 결과로 동일한 비즈니스 로직을 중복 개발할 수 있음, 따라서 보다 안정적인 특성을 가지는 영역에 비즈니스 로직을 두도록 설계해야 함, 엔티티 타입의 컴포넌트에 업무의 본질을 담아 제대로 설계해야 하는 이유가 여기에 있음.

2.5 서비스와 컴포넌트 (1/3)

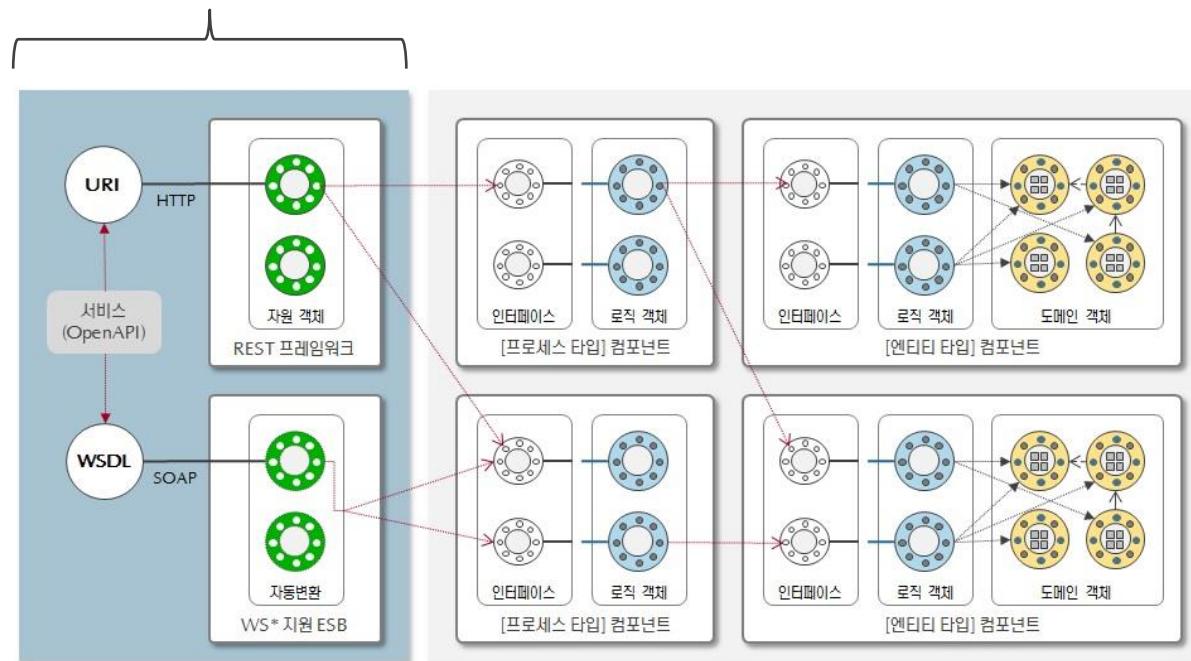
- ✓ 비즈니스 로직은 다양한 유형의 컴포넌트 단위로 구현되고, 컴포넌트 상호 간에 복잡한 의존규칙을 가지고 있 습니다. 로직 처리에 필요한 정보는 도메인 객체로 로직 자체는 로직 객체로 분리하여 표현하고, 인터페이스는 인터페이스 클래스로 처리합니다.
- ✓ 시스템 설계에서 [서비스]란 컴포넌트 형식으로 개발된 비즈니스를 외부로 노출하는 방식의 한 가지입니다.



2.5 서비스와 컴포넌트 (2/3)

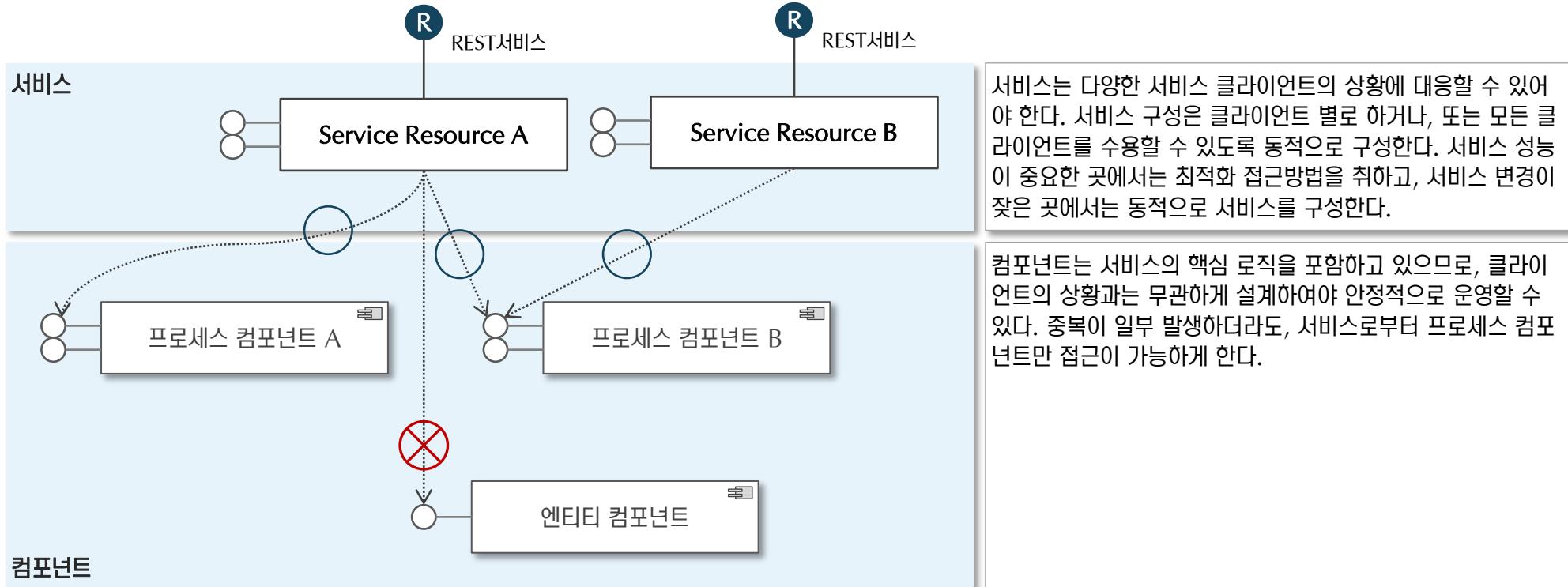
- ✓ SOA 기반 설계에서도 시스템의 확장성과 유연성은 컴포넌트 설계에 달려있습니다.
- ✓ SOA의 장점 하나는 컴포넌트 인터페이스가 가진 한계를 극복하여, 지역 호출, 원격 호출 불문하고 편안한 호출을 가능하게 하는 것입니다. SOA의 장점 들은 모듈의 단위로 용이함과 표준을 바탕으로 서비스 인터페이스 수준으로 끌어올려 기업 SW 자원의 효율성을 높여 주는 것입니다.

컴포넌트의 한계를 극복하기 위해 보완된 장치로써의
서비스 발행 프레임워크와 서비스 자원



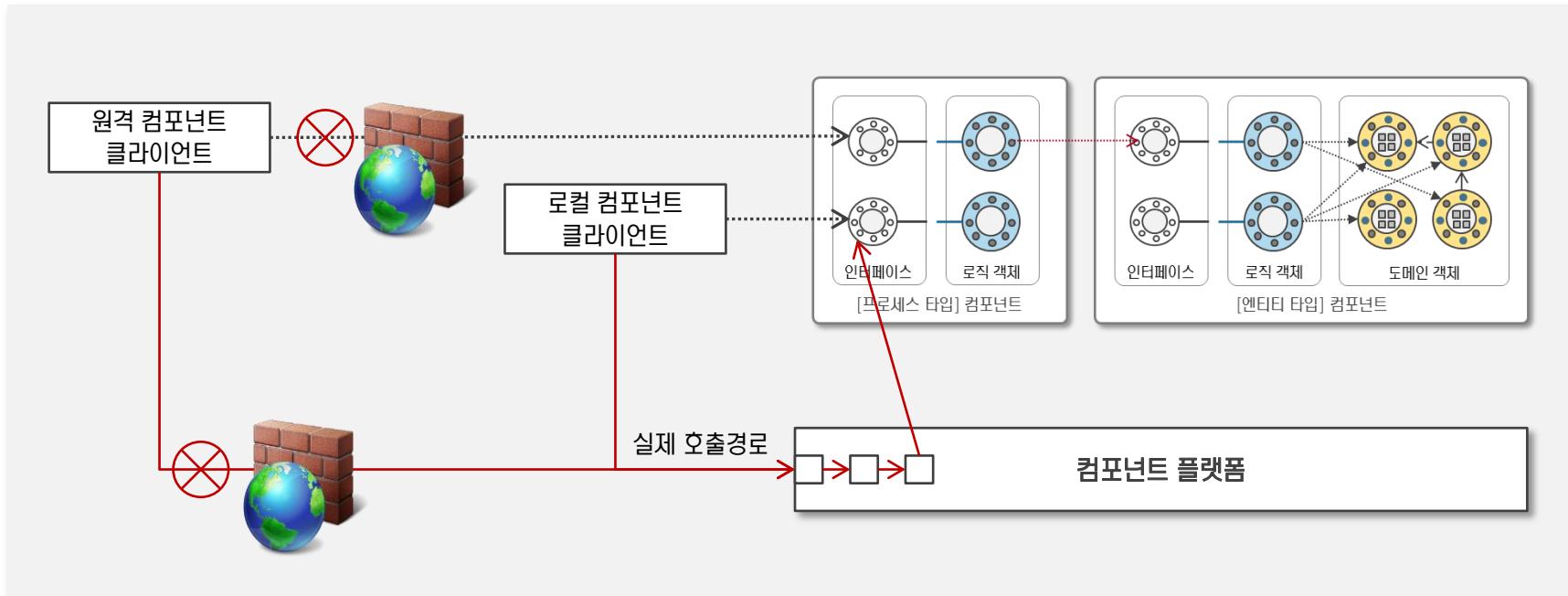
2.5 서비스와 컴포넌트 (3/3)

- ✓ 설계 시 컴포넌트 관점에서 서비스 클라이언트를 완전히 가려서 보이지 않도록 설계해야 합니다.
- ✓ 다양한 서비스 클라이언트 변형(variation)은 서비스 영역에서 담당할 수 있어야 합니다.
- ✓ 모든 클라이언트 상황에 적용할 수 있는 서비스를 제공하기에는 어려움이 있으며, 클라이언트 별 차별화 대응이 필요합니다.



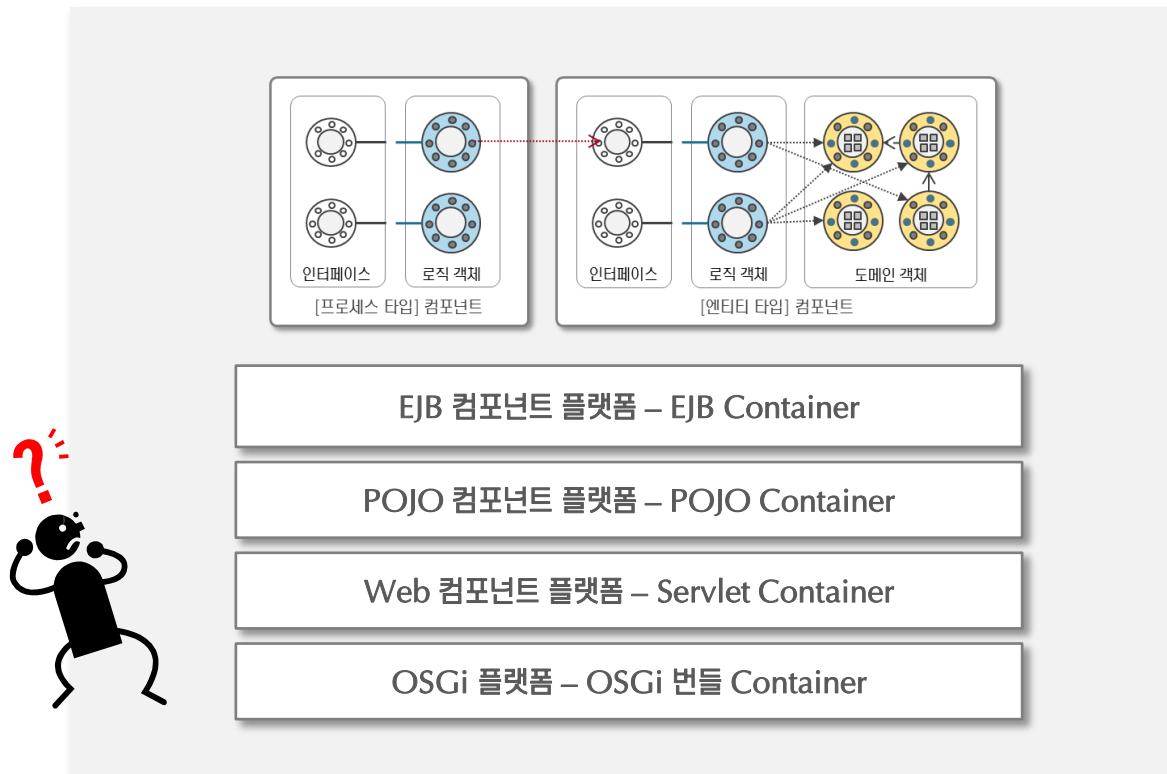
2.6 컴포넌트의 한계 (1/4) – 호출거리

- ✓ 로컬 클라이언트는 컴포넌트 플랫폼에 접근할 수 있는 환경에 있지만, 원격 클라이언트는 보안 등의 이유로 접근할 수 없습니다. 따라서, 컴포넌트 플랫폼으로의 접근을 보다 쉽게 도와주는 인프라가 필요합니다.
- ✓ 그리고, 원격 클라이언트가 로컬의 컴포넌트 플랫폼으로의 접근을 도와주는 인프라가 필요합니다.



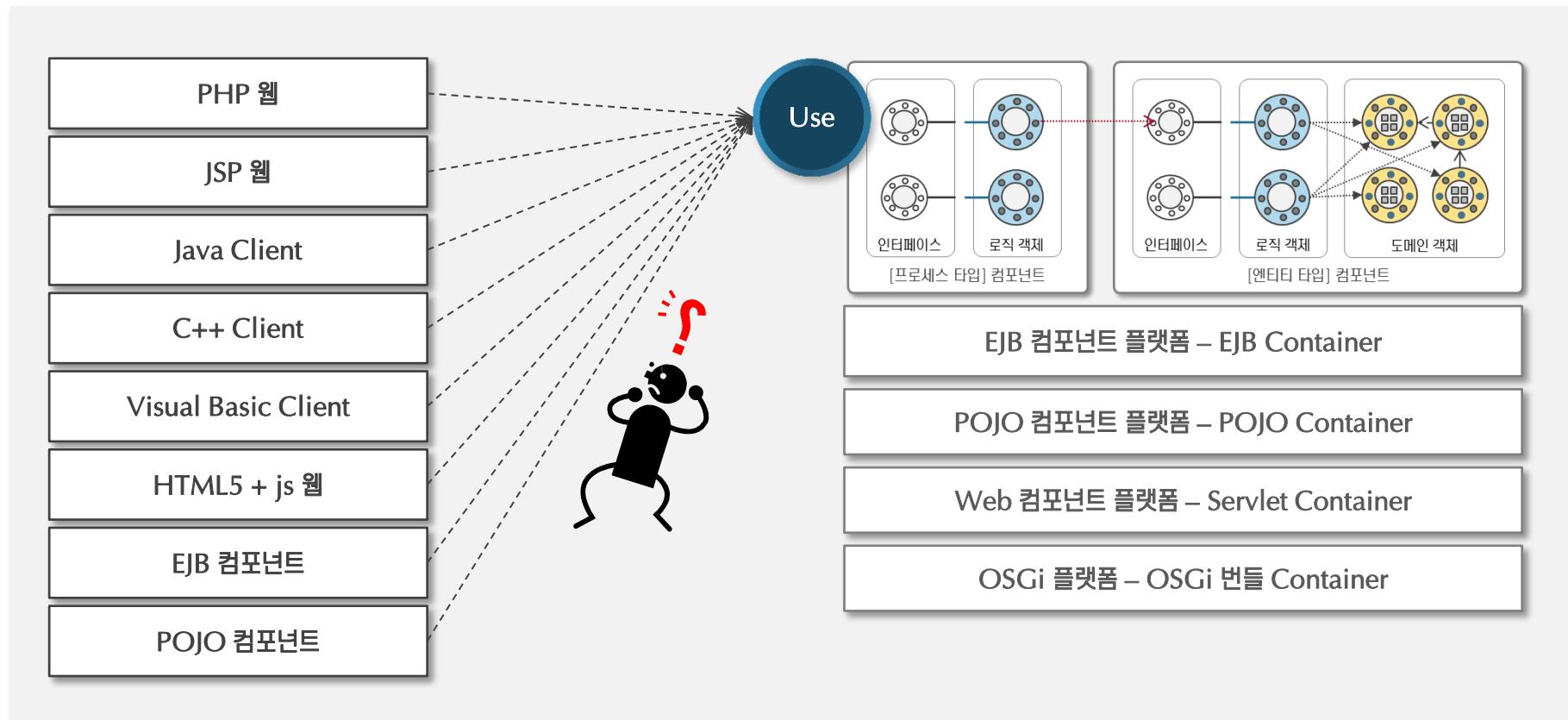
2.6 컴포넌트의 한계 (2/4) – 다양한 컴포넌트 모델

- ✓ 다양한 컴포넌트 모델이 존재하므로 목표 시스템의 특성을 잘 반영한 모델을 선택할 수 있는 장점이 있습니다.
- ✓ 하지만, 이러한 다양성은 클라이언트 입장에서 보면 혼란과 어려움을 의미합니다.
- ✓ 서버 측의 다양한 컴포넌트에 한 가지 방식으로 접근할 수 있어야 합니다. ← 서비스 발행 플랫폼의 역할



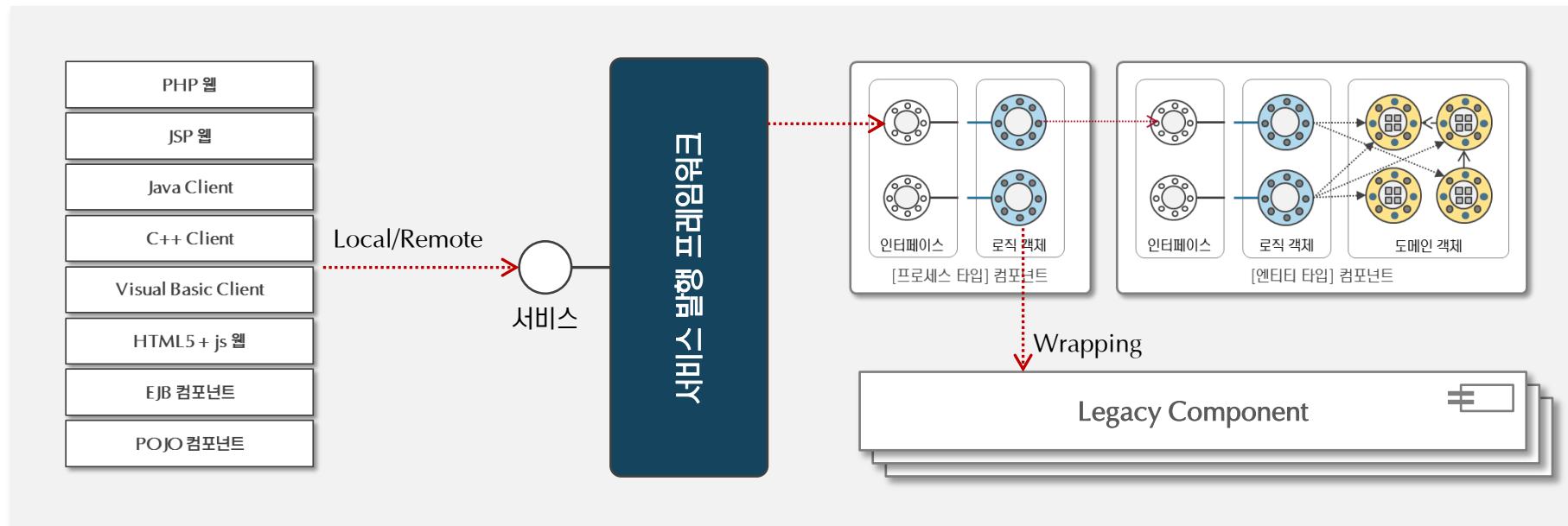
2.6 컴포넌트의 한계 (3/4) – 사용/재사용의 어려움

- ✓ 원격 접근의 어려움 외에도, 컴포넌트 인터페이스의 사용 또는 재사용의 어려움이 존재합니다.
- ✓ 어떤 클라이언트라도 서버 측에서 잘 구축된 컴포넌트를 쉽게 활용할 수 있다면 개발 생산성 및 자원 효율성을 높일 수 있습니다.
- ✓ 모든 클라이언트들이 가장 잘 알고 쉽게 접근하는 방법이 있어야 합니다. ← 서비스 발행 플랫폼의 역할



2.6 컴포넌트의 한계 (4/4) – SOA를 통한 극복

- ✓ 서비스 발행 프레임워크(Service Publishing Framework)은 컴포넌트의 제약 조건을 극복할 수 있도록 도와주는 인프라입니다. 서비스 발행 프레임워크는 사용하기 쉬운 방식으로 표준 서비스 인터페이스를 외부로 노출(publishing) 하여 줍니다.
- ✓ 서비스 발행은 REST 방식과 WS*(WSDL, SOAP, UDDI) 방식이 있습니다.



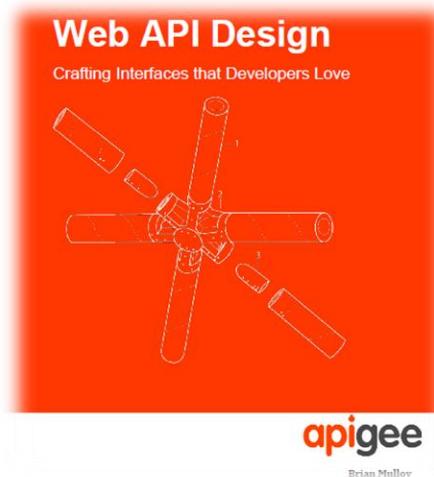


3. RESTful API

-
- 3.1 실용주의 REST API
 - 3.2 명사와 동사
 - 3.3 연관 관계 단순화
 - 3.4 에러 처리
 - 3.5 버전 팀
 - 3.6 페이지 처리와 부분 응답
 - 3.7 자원이 없는 API
 - 3.8 API 호출 예제

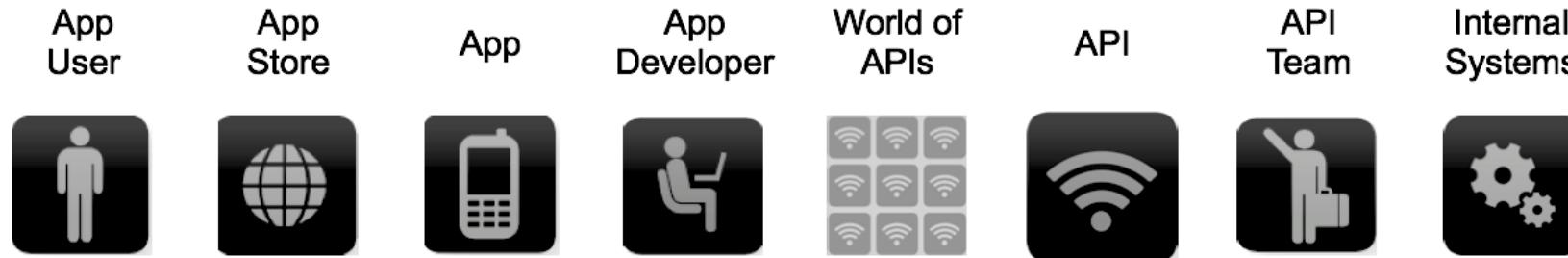
3.1 실용주의 REST API (1/3)

- ✓ 이 장은 apigee(www.apigee.com) 사의 eBook을 기반으로 진행합니다. ← 무료 다운로드 가능함
- ✓ REST API 설계는 아키텍처 스타일이지 표준이 아닙니다. 적용에 유연한 입장을 취할 수 있습니다.
- ✓ API 설계의 핵심은 실용주의 관점을 유지하는 REST입니다.
- ✓ API 설계는 얼마나 많은 개발자들이 빨리 여러분의 API를 이용하느냐로 증명됩니다.



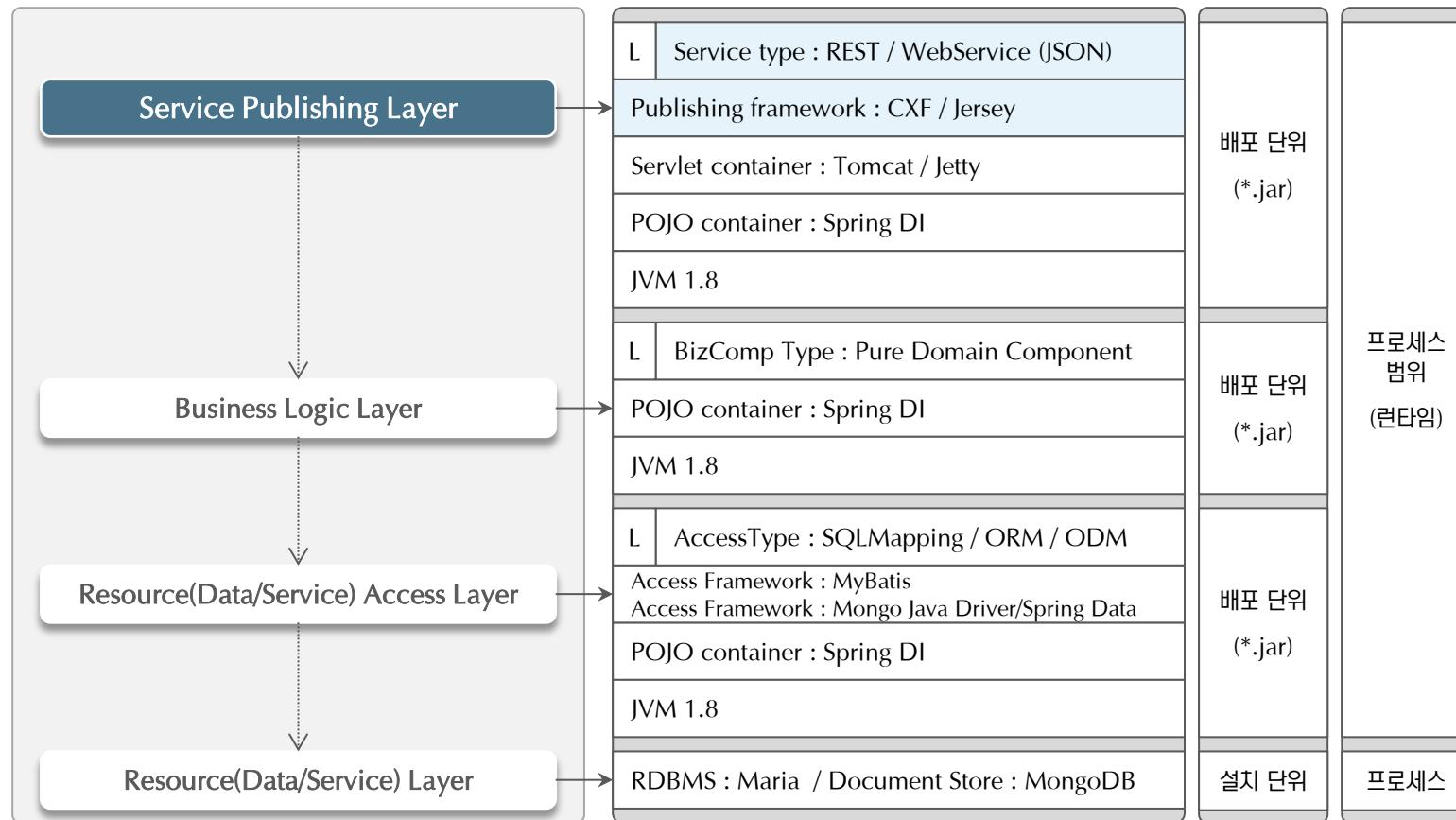
3.1 실용주의 REST API (2/3)

- ✓ API의 목표는 API를 사용하는 개발자의 성공을 돋는 것입니다.
- ✓ 따라서 API 설계의 첫번째 원칙은 개발자의 생산성과 성공을 극대화 하는 것입니다. ← 실용주의 REST
- ✓ 개발자의 관점에서 보면 많은 팁과 경험을 포함하는 훌륭한 가이드가 가장 중요합니다.



3.1 실용주의 REST API (3/3)

- ✓ RESTful API는 애플리케이션 레이어에서 보면 서비스 발행 레이어에 놓입니다.
- ✓ 아키텍처 설계 - 서비스 발행 플랫폼(또는 프레임워크)를 선택하고 RI를 개발한 후, API 가이드를 만듭니다.
- ✓ 도메인 모델(또는 서비스 명세) – API 가이드에 따라 외부로 서비스 API를 정의한 후 발행합니다.



3.2 명사와 동사 [1/3]

- ✓ 실용적인 RESTful 디자인의 첫번째 원칙은 간결하고 직관적인 기준 URL을 유지하는 것입니다.
- ✓ 기준 URL 설계는 API에서 가장 중요한 설계 행동유도성(affordance)입니다.
- ✓ 행동유도성은 가이드 문서가 필요없는 설계 재산입니다. ← 직관적인 설계



설계 행동유도성(affordance)과 문서 간의 충돌 !!

3.2 명사와 동사 (2/3)

- ✓ 자원(resource) 별로 두 개의 기준 url을 사용합니다.
- ✓ 기준 url에는 동사를 두지 않습니다.
- ✓ 컬렉션이나 요소들을 다룰 때는 HTTP 동사(??)를 사용합니다.

// 목록을 위한 URL

/dogs

// 목록 중 특정 개체를 위한 URL

/dogs/1234

Resource	POST(create)	GET(read))	PUT(update)	DELETE(delete)
/dogs	새로운 dog 생성	dogs 목록	dogs에 대한 대량 업데이트	모든 dogs 삭제
/dogs/1234	예러	특정 dog 보기	있으면 업데이트, 없으면 예러	삭제

3.2 명사와 동사 (3/3) – 복수 명사와 구체적인 이름

- ✓ 개발자가 API를 이용할 때, 예측하거나 추측할 수 있도록 일관성을 유지해야 합니다.
- ✓ 직관적인 API는 단수 명사 보다는 복수를 사용합니다.
- ✓ 좋은 API는 추상적인 이름이 아닌 구체적인 이름을 사용합니다.
- ✓ /items이나 /assets 보다는 /blogs나 /videos라는 식으로 구체적인 이름을 사용하는 것이 좋습니다.

Foursquare	GroupOn	Zappos
/checkins	/deals	/Product

3.3 연관관계 단순화

- ✓ URL을 5내지 6레벨까지 구성하는 것은 읽기 어려울 뿐만 아니라 바람직하지 않습니다.
- ✓ 자원 간의 연관관계를 단순하게 유지하는 방법으로 API의 직관성을 유지해야 합니다.
- ✓ HTTP 물음표(?) 아래 여러 패러미터를 두고 복잡성을 감추어야 합니다.

```
GET /dogs?color=red&state=running&location=park
```

3.4 에러 처리 (1/3)

- ✓ 실용주의 REST API 에서는 에러를 어떻게 처리하는가?
- ✓ Facebook은 상태 코드가 200과 #803 에러가 났다고 하지만 그 에러가 무엇인지 알려주지 않습니다.
- ✓ Twilio는 상태코드가 401이며, 에러와 에러에 해당된 내용을 확인할 수 있는 문서정보까지 제공합니다.
- ✓ SimpleGeo는 상태코드가 401이며, 에러코드 외에 다른 정보는 제공하지 않습니다.

Facebook

HTTP Status Code: 200

```
{"type": "OAuthException", "message": "#803 Some of the aliases you requested do not exist: foo.bar"}
```

Twilio

HTTP Status Code: 401

```
{"status": "401", "message": "Authenticate", "code": 20003, "more info": "http://www.twilio.com/docs/errors/20003"}
```

SimpleGeo

HTTP Status Code: 401

```
{"code": 401, "message": "Authentication Required"}
```

3.4 에러 처리 (2/3)

- ✓ HTTP 상태 코드를 사용합니다.
- ✓ 실제 상태 코드는 세 가지 정도만 제공하면 되고 필요하면 더 추가할 수 있습니다.
- ✓ 더 추가할 경우, 최대 8개를 넘지 않아야 합니다.

추천 기본 상태

200 - OK

400 - Bad Request

500 - Internal Server Error

추가상태

201 - Created

304 - Not Modified

404 - Not Found

401 - Unauthorized

403 - Forbidden

3.4 에러 처리 (3/3)

- ✓ 상태코드를 사용하더라도 사용자에게 가능한 자세한 메시지를 제공하여야 합니다.

코드를 위한 코드

200 - OK 401 - Unauthorized

사용자를 위한 메시지

```
{"developerMessage": "Verbose, plain language description of the problem for the app developer with hints about how to fix it.", "userMessage": "Pass this message on to the app user if needed.", "errorCode": 12345, "more info": "http://dev.teachdogrest.com/errors/12345"}
```

3.5 버전 팀 (1/2)

- ✓ 버전 없이 API를 릴리즈하면 안됩니다. 그리고 버전은 옵션이 아닌 필수입니다.
- ✓ 다른 서비스에서 사용하는 **versioning** 예제입니다.
 - Twilio URL에 timestamp를 사용합니다.
 - Salesforce.com은 URL의 중간 지점에 v20.0과 같은 버전 정보를 둡니다.
 - Facebook은 v. 와 같은 버전 표기법을 사용하지만 버전 정보는 선택 가능한 패러미터입니다.

Twilio

/2010-04-01/Accounts/

salesforce.com

/services/data/v20.0/sobjects/Account

Facebook

?v=1.0

3.5 버전 팀 (2/2)

✓ 실용주의적인 REST에서 버전 번호를 어떻게 처리해야 하는가?

- 단순한 서수를 사용합니다.
- 적어도 하나의 버전을 유지합니다.
- 버전을 폐기하기 전에 개발자들이 대응할 수 있는 시간을 충분히(한 사이클 정도) 주어야 합니다.

3.6 페이지 처리와 부분 응답 (1/2)

- ✓ “부분 응답”은 개발자들이 필요로 하는 것만 제공하는 것입니다.
- ✓ 아래 정보는 3가지 서비스에서 제공하고 있는 “부분 응답” 예제입니다.
- ✓ 다음은 각 서비스에서 쉼표로 구분한 옵션 필드 등을 이용하여 제공하는 “부분 응답” API 예제입니다.

LinkedIn

/people:(id,first-name,last-name,industry)

Facebook

/joe.smith/friends?fields=id,name,picture

Google

?fields=title,media:group(media:thumbnail)

3.6 페이지 처리와 부분 응답 [2/2]

- ✓ 개발자가 페이지 처리를 하기 쉽게 offset과 limit을 제공하도록 합니다.
- ✓ 다음은 offset과 limit 값의 예입니다.

Facebook

offset 50 and limit 25

Twitter

page 3 and rpp 25 (페이지 당 레코드 개수)

LinkedIn

start 50 and count 25

3.7 자원이 없는 API

- ✓ DB에 저장된 자원(resource)과 관련이 없는 응답을 처리할 때는 다음과 같이 하는 것을 추천합니다.
 - 명사가 아닌 동사를 사용합니다.
 - 비자원 처리 API는 다른 시나리오를 사용함으로 명확하게 서술합니다.
- ✓ 아래는 100유로를 중국 yen으로 바꿀 때의 예제입니다.

```
/convert?from=EUR&to=CNY&amount=100
```

3.8 API 호출 예제 (1/3)

- ✓ 이름이 AI인 갈색 강아지 객체를 생성합니다.
- ✓ AI라는 이름을 가진 강아지의 이름을 Rover로 이름을 변경합니다.

Request

```
POST /dogs  
name=AI&furColor=brown ← 바디로 전달
```

Response

```
200 OK  
{ "dog": { "id": "1234", "name": "AI", "furColor": "brown" } }
```

Request

```
PUT /dogs/1234  
name=Rover ← 바디로 전달
```

Response

```
200 OK  
{ "dog": { "id": "1234", "name": "Rover", "furColor": "brown" } }
```

3.8 API 호출 예제 (2/3)

- ✓ 특정 강아지 정보를 조회합니다.
- ✓ 모든 강아지 정보를 가져옵니다.

Request

```
GET /dogs/1234
```

Response

```
200 OK
```

```
{ "dog": { "id": "1234", "name": "Rover", "furColor": "brown" } }
```

Request

```
GET /dogs
```

Response

```
200 OK
```

```
{ "dogs": [  
    { "dog": { "id": "1233", "name": "Fido", "furColor": "white" } },  
    { "dog": { "id": "1234", "name": "Rover", "furColor": "brown" } }  
],  
"_metadata": [ { "totalCount": 327, "limit": 25, "offset": 100 } ]  
}
```

3.8 API 호출 예제 (3/3)

- ✓ 1234라는 아이디를 가진(여기서는 Rover라는 이름을 가진) 강아지 정보를 삭제합니다.

Request

```
DELETE /dogs/1234
```

Response

```
200 OK
```

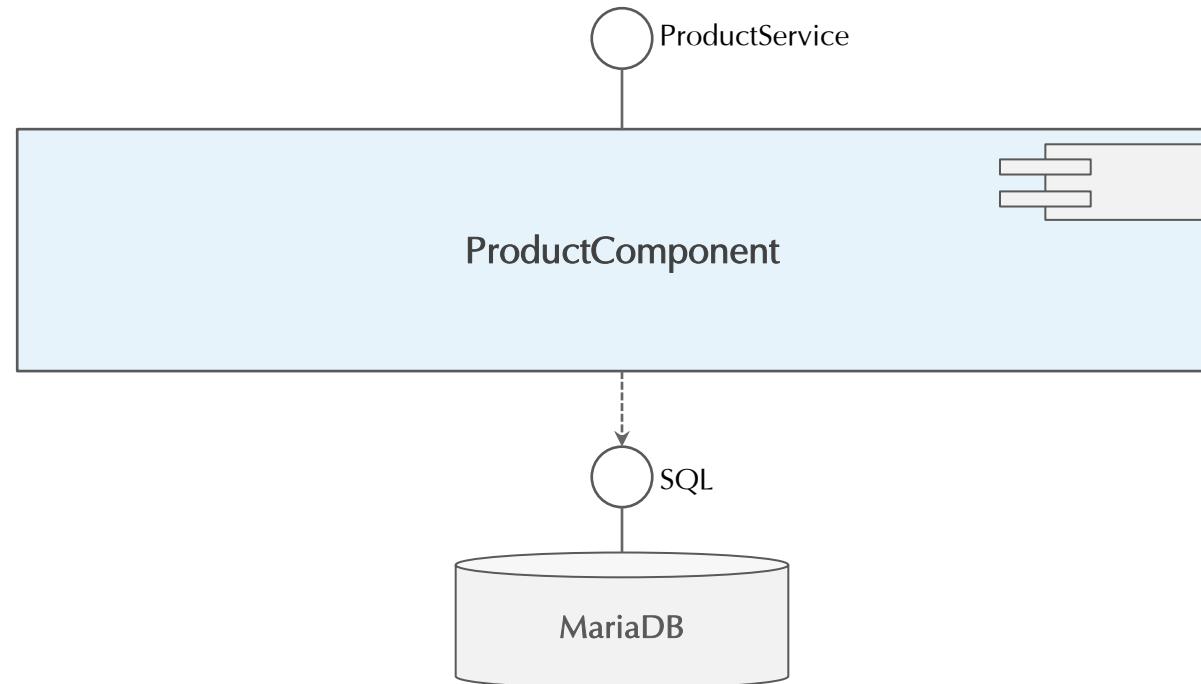


4. 비즈니스 컴포넌트

- 4.1 컴포넌트 구성
- 4.2 컴포넌트 설계
- 4.3 무늬만 컴포넌트
- 4.4 BCF 컴포넌트
- 4.5 도메인 컴포넌트
- 4.6 Pure 도메인 컴포넌트

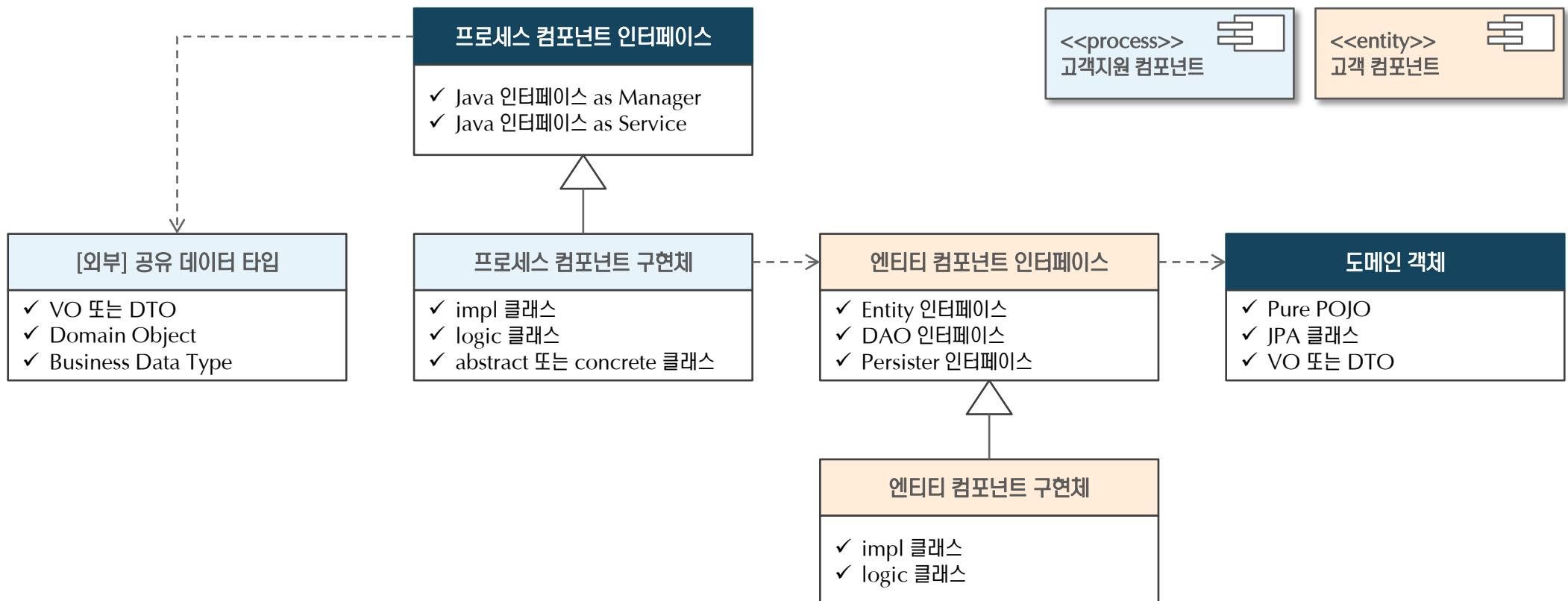
4.1 컴포넌트 구성(1/3)

- ✓ 비즈니스 컴포넌트 속에는 무엇이 있을까요?
- ✓ 정보에 해당하는 도메인 엔티티 객체, 로직에 해당하는 도메인 로직 객체, 데이터 저장소에 접근하는 객체, ...
- ✓ 여러 가지 유형의 인터페이스와 객체들이 규칙적으로 정렬되어 있을 겁니다.
- ✓ 컴포넌트 내부를 구성하는 요소(객체들의 그룹)는 어떤 것들이 있는지 생각해 봅시다.



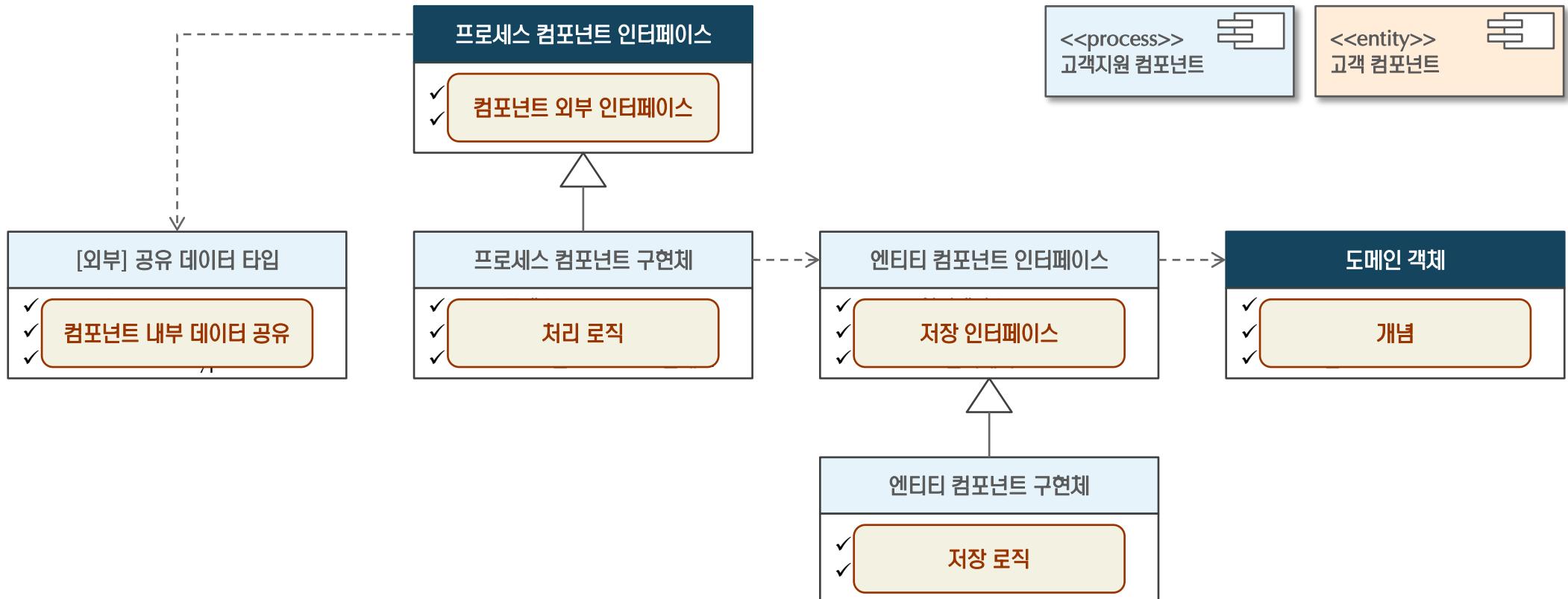
4.1 컴포넌트 구성(2/3) – 요소

- ✓ 비즈니스 컴포넌트를 구성하는 요소는 프로세스/엔티티 컴포넌트의 인터페이스와 구현체, 공유 타입과 도메인 객체입니다.
- ✓ 각 요소를 구현하는 방식에 따라 컴포넌트의 특징을 결정할 수 있습니다.
- ✓ 컴포넌트 구조 설계 시, 사용하는 요소의 이름은 서로 달라서 혼란스럽긴 하지만, 결국은 아래 여섯 가지 요소 중 하나입니다.
- ✓ 가장 이상적인 컴포넌트에서부터 무늬만 컴포넌트 까지 아래의 여섯 가지 구성 요소의 조합으로 결정됩니다.



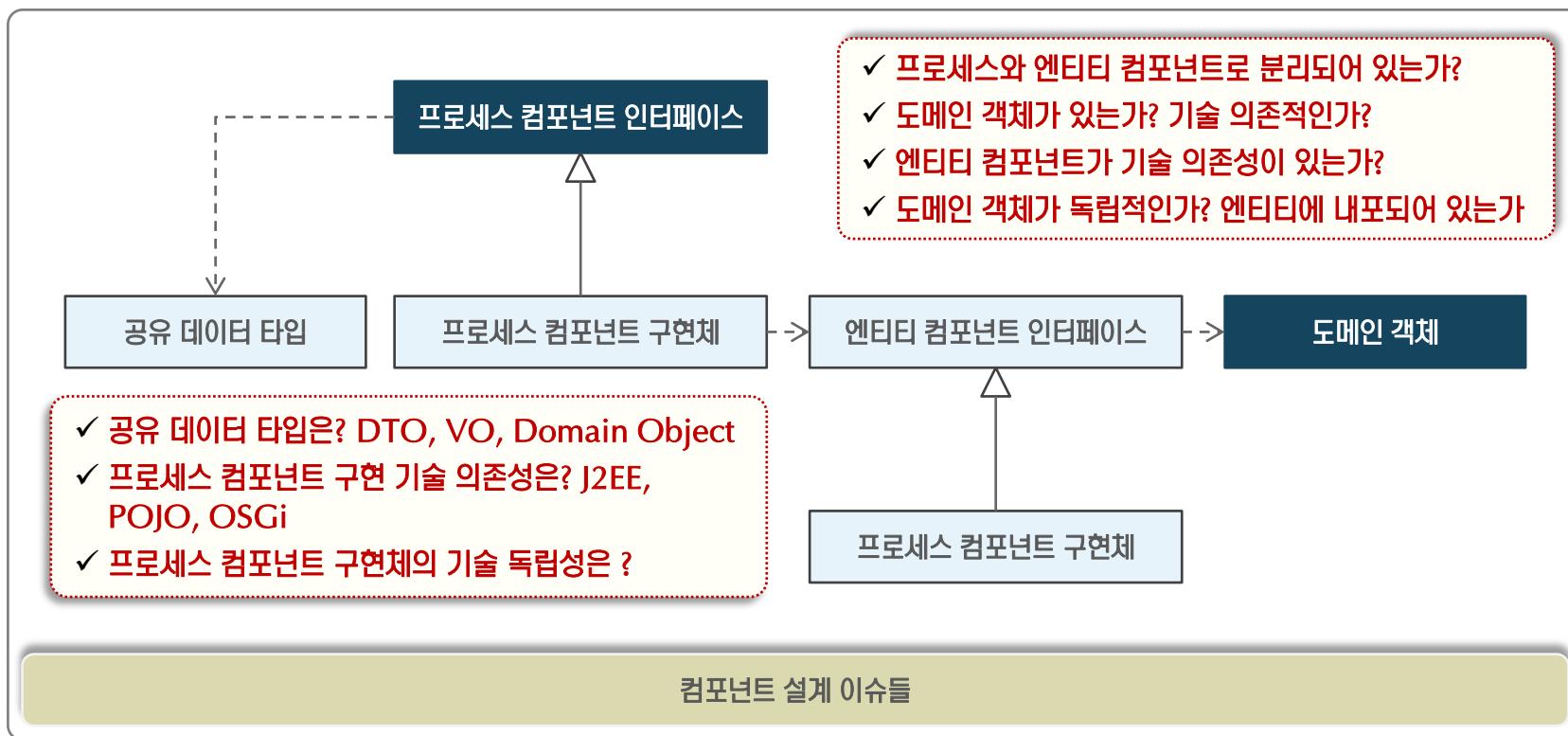
4.1 컴포넌트 구성(3/3) – 요소 별 목적

- ✓ “Loose coupling, High cohesion” 설계 목표를 달성하기 위해 컴포넌트 구성 요소는 각자의 존재 이유가 있습니다.
- ✓ 컴포넌트의 외부로 노출되는 구성요소, 비즈니스 로직을 처리하는 구성요소, 지속성을 담당하는 요소와,
- ✓ 컴포넌트가 책임지고 있는 주요 개념을 담당하는 “도메인 객체”로 구성됩니다.
- ✓ 각자의 역할을 가지고 전체 컴포넌트 그룹에 참여합니다. 이러한 요소의 역할을 조정하면 서로 다른 특성을 보여 줍니다.



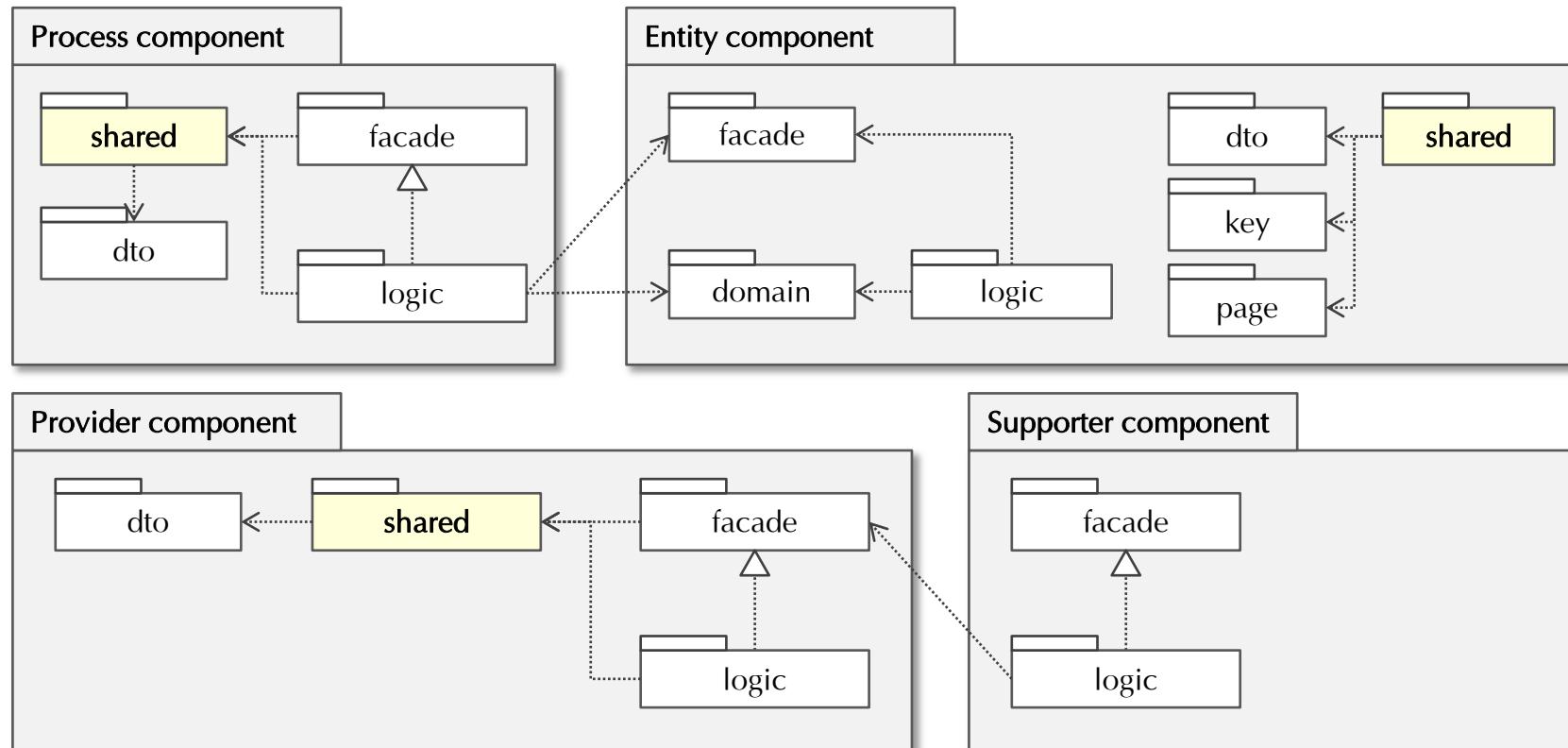
4.2 컴포넌트 설계(1/4) – 이슈

- ✓ 프로세스와 엔티티 컴포넌트로 분리되어 있는가? 분리되어 있다면, 엔티티 컴포넌트를 무엇이라 부르는가?
- ✓ 도메인 객체가 존재하는가? 존재한다면 기술 종속적인가, 아니면 PURE POJO인가?
- ✓ 도메인 객체가 엔티티 컴포넌트 내부에 있는가? 아니면 외부에 독립적으로 존재하는가?
- ✓ 프로세스 컴포넌트의 비즈니스 로직은 기술 독립적인가? 독립적이라면 추상 클래스인가, 실제 클래스인가?



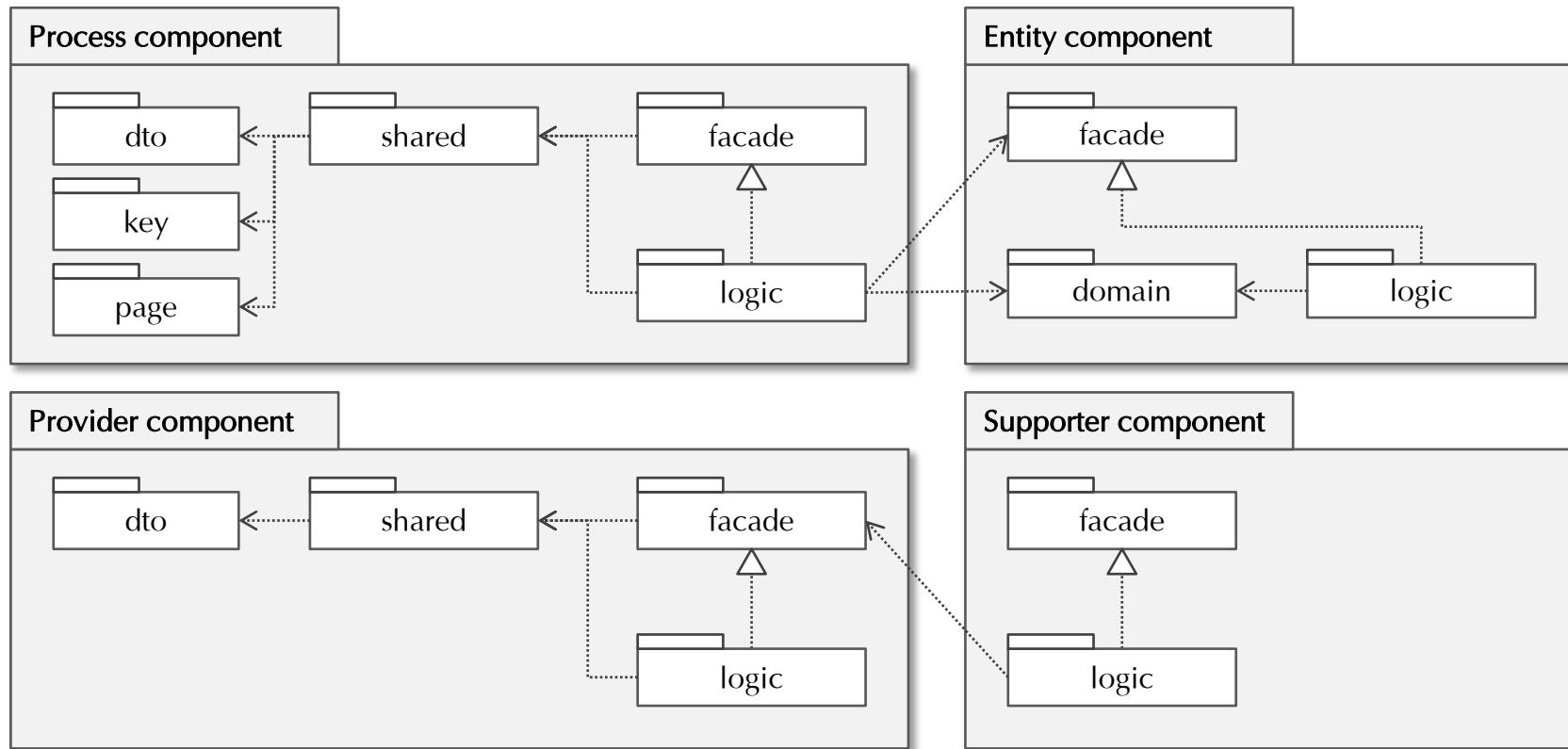
4.2 컴포넌트 설계(2/4) – 내부 구조 1

- ✓ 컴포넌트 유형별로 서로 다른 내부구조를 가지는데 이것은 컴포넌트의 역할과 관계가 있습니다.
- ✓ 프로세스 타입 컴포넌트 내부에 있는 shared 영역을 엔티티 타입 컴포넌트에도 두어야 할지 고민이 필요합니다.
- ✓ Supporter 컴포넌트는 shared 영역이 없으며 가장 단순한 구조를 가져야 합니다.



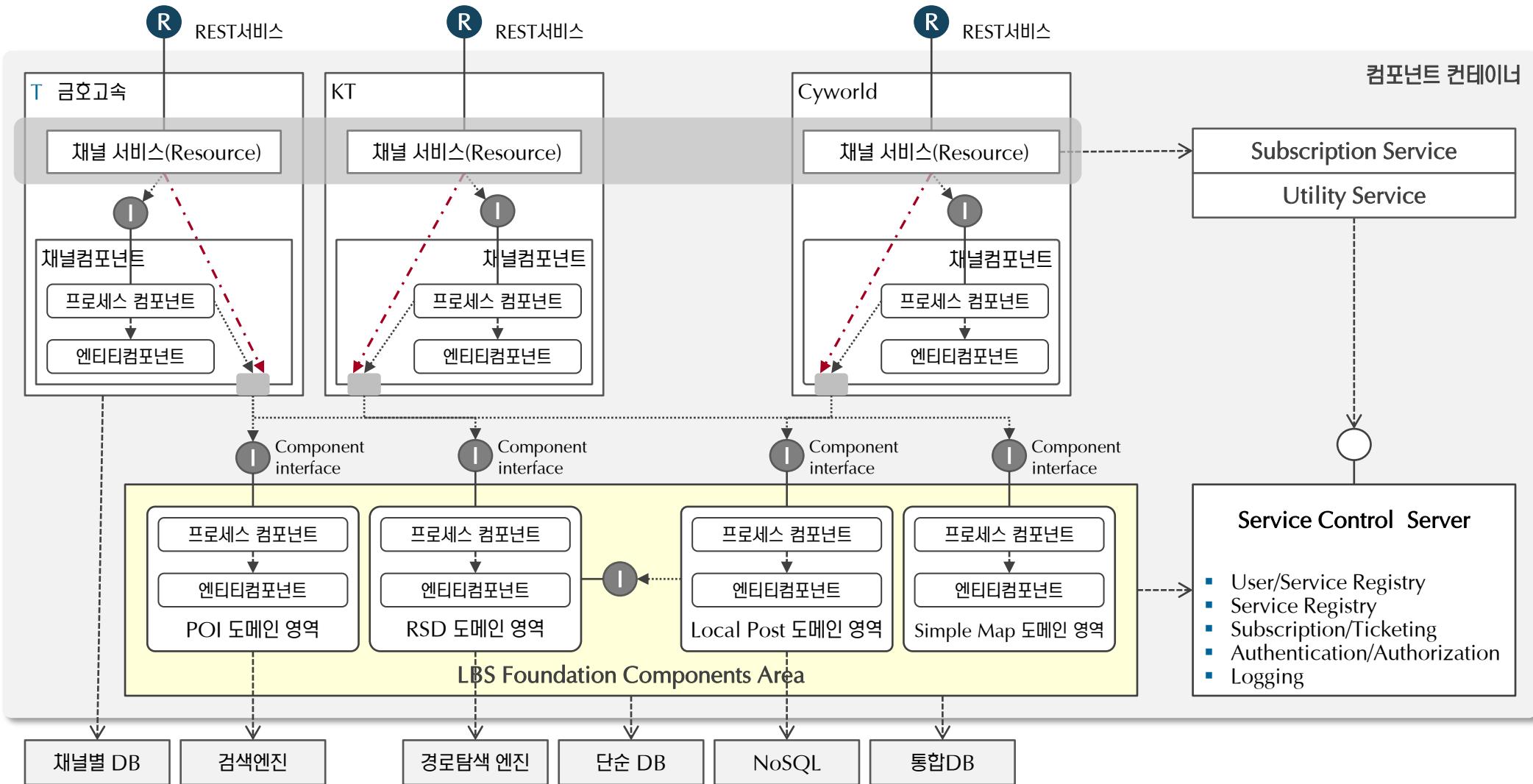
4.2 컴포넌트 설계(3/4) – 내부 구조 2

- ✓ Entity 컴포넌트는 공유정보를 가지지 않고, 모든 공유정보는 Process 컴포넌트 내부로 두는 구조입니다.
- ✓ 하나의 프로세스 컴포넌트가 여러 Entity 컴포넌트를 사용하는 경우, Process 컴포넌트의 공유영역이 복잡합니다.
- ✓ 설계의 원칙을 준수하지만 현실적인 어려움이 발생하는 구조입니다.



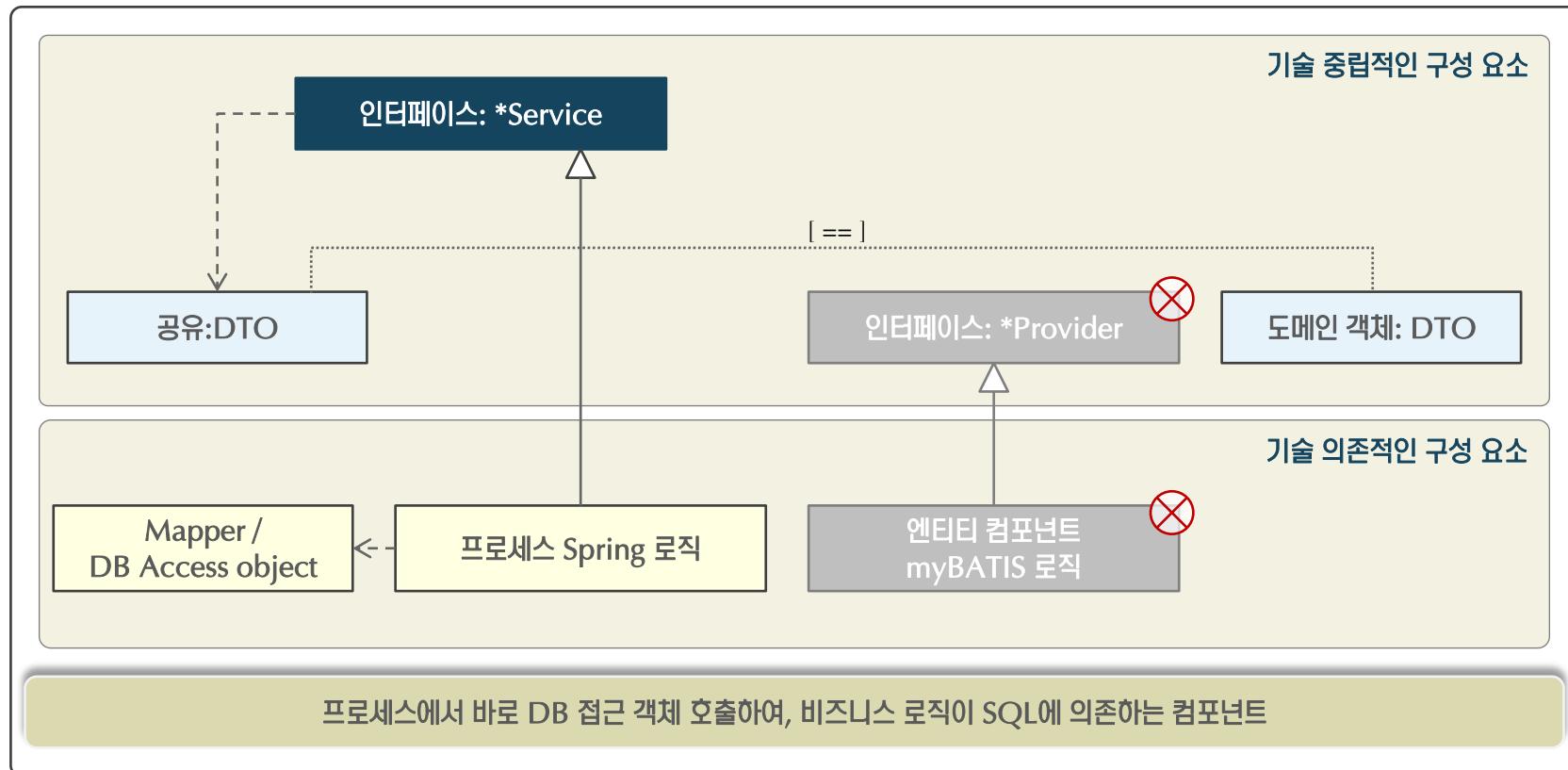
4.2 컴포넌트 설계(4/4) – 채널 구조

✓ 채널 구조가 필요한 시스템의 설계 예제입니다.



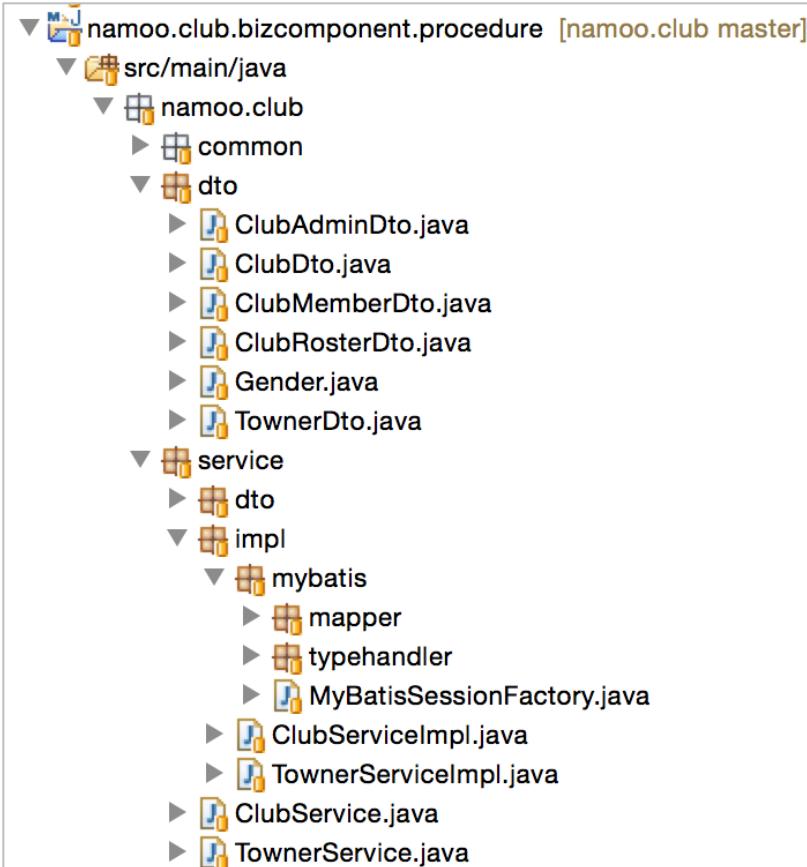
4.3 무늬만 컴포넌트(1/2)

- ✓ 컴포넌트 내부에 프로세스 로직과 엔티티 로직이 분리되어 있지 않습니다.
- ✓ 이런 구조는 기존의 절차 지향 구조를 컴포넌트 인터페이스 속으로 감춘 것에 지나지 않습니다.
- ✓ 도메인 객체가 없을 뿐만 아니라, DTO는 비즈니스 본질을 담기 보다는 UI에서 요구하는 정보를 담아 줄 뿐입니다.
- ✓ SI 프로젝트에서 이런 구조를 심심치 않게 봅니다. 어찌되었건 myBATIS는 사용을 했으니 문제 될 것이 없다고 합니다.



4.3 무늬만 컴포넌트(2/2)

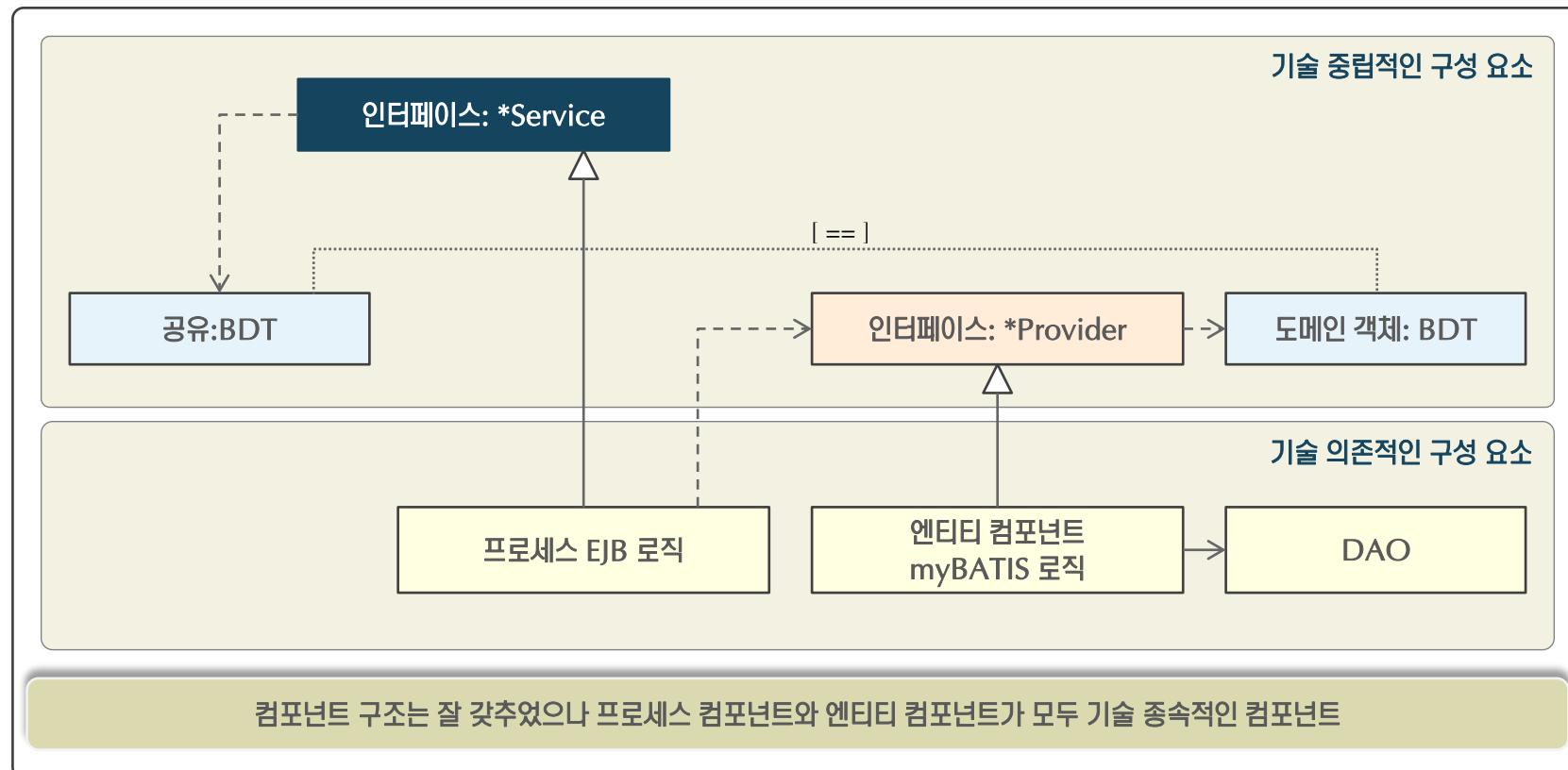
- ✓ 비즈니스 로직에서 Persistence를 책임지는 객체를 직접 다루도록 설계하였습니다.
- ✓ SQL에 많이 의지 합니다.
- ✓ 저장 방식에 의존적입니다.



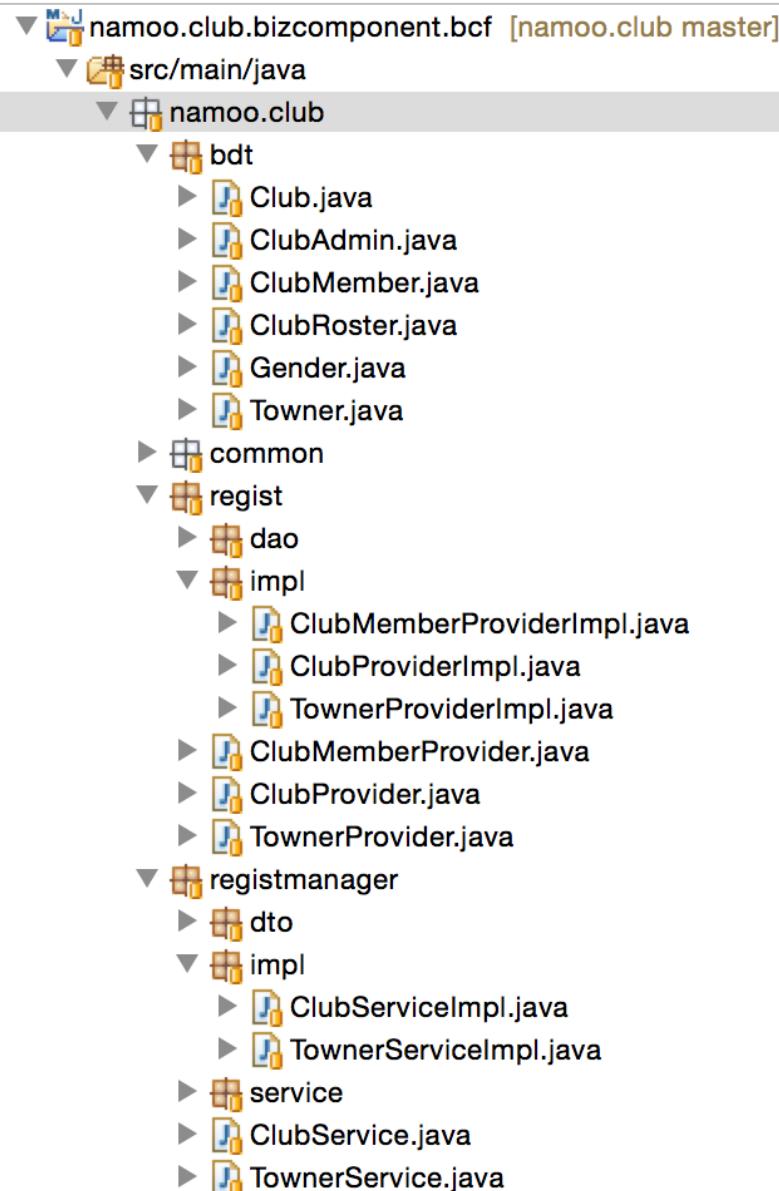
```
public class ClubServiceImpl implements ClubService {  
    //  
    @Override  
    public String registerClub(ClubCDto clubCDto) throws NamooClubException {  
        //  
        SqlSession session = MyBatisSessionFactory.getSession();  
  
        try {  
            ClubMapper clubMapper = session.getMapper(ClubMapper.class);  
            TownerMapper townnerMapper = session.getMapper(TownerMapper.class);  
            ClubMemberMapper clubMemberMapper = session.getMapper(ClubMemberMapper.class);  
  
            if (clubMapper.selectClubByName(clubCDto.getClubName()) != null) {  
                throw new NamooClubException("Already existed club --> " + clubCDto.getClubName());  
            }  
  
            TownerDto townner = null;  
            try {  
                townner = townnerMapper.selectTownerByEmail(clubCDto.getAdminEmail());  
            }  
            catch (EmptyResultException e) {  
                throw new NamooClubException(e);  
            }  
  
            ClubDto club = clubCDto.createDomain(new ClubAdminDto(townner));  
            clubMapper.insertClub(club);  
  
            ClubMemberDto clubMember = new ClubMemberDto(club, townner);  
            clubMemberMapper.insertClubMember(club.getId(), clubMember);  
            return club.getId();  
        } finally {  
            session.close();  
        }  
    }  
}
```

4.4 BCF 컴포넌트(1/2)

- ✓ 도메인 개념을 표현하는 도메인 객체 관점 보다는 외부와의 공유 관점에 초점을 둔 Business Data Type을 정의합니다.
- ✓ 프로세스 컴포넌트는 플랫폼에 종속적입니다. 이런 구조를 가질 때는, 플랫폼을 변경할 가능성이 없을 때입니다.
- ✓ 엔티티 인터페이스는 중립 영역에 있으며, 구현은 지속성 기술에 종속적입니다.
- ✓ 물론 DAO를 통해서 DB에 접근함으로써, DB 변경에 따른 유연성을 가질 수 있는 구성입니다.



4.4 BCF 컴포넌트(2/2)



```
public class ClubServiceImpl implements ClubService {
    //
    private ClubProvider clubProvider;
    private ClubMemberProvider clubMemberProvider;
    private TownerProvider townerProvider;

    /**
     * @return
     */
    public ClubServiceImpl() {
        //
        RegistManagerServiceFactory serviceFactory = RegistManagerServiceFactory.getInstance();

        this.clubProvider = serviceFactory.getClubProvider();
        this.clubMemberProvider = serviceFactory.getClubMemberProvider();
        this.townerProvider = serviceFactory.getTownerProvider();
    }

    /**
     * @Override
     */
    public String registerClub(ClubCDto clubCDto) throws NamooClubException {
        //
        if (clubProvider.isExistClubByName(clubCDto.getClubName())) {
            throw new NamooClubException("Already existed club --> " + clubCDto.getClubName());
        }

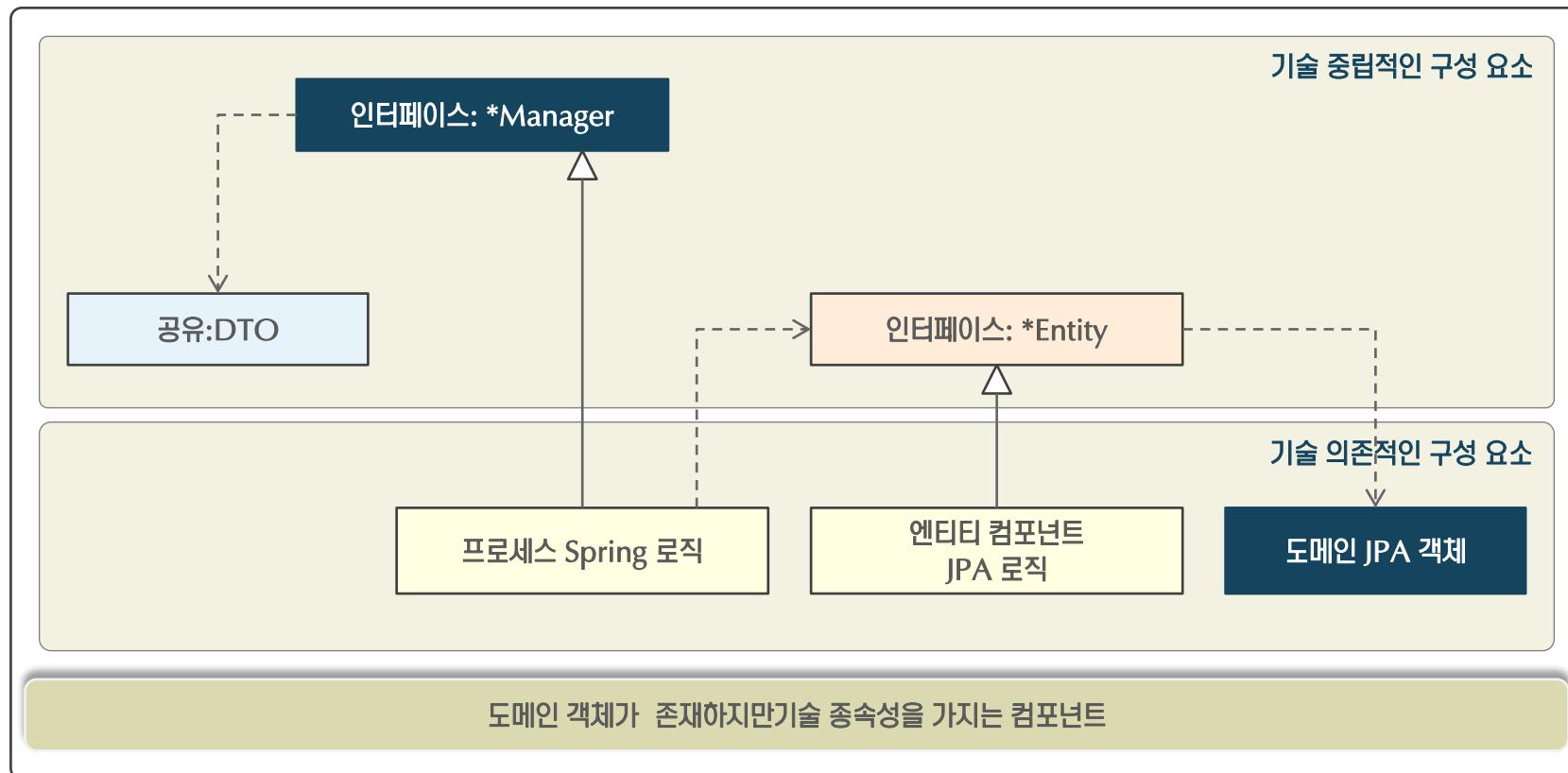
        Towner towner = null;
        try {
            towner = townerProvider.retrieveByEmail(clubCDto.getAdminEmail());
        } catch (EmptyResultException e) {
            throw new NamooClubException(e);
        }

        Club club = clubCDto.createDomain(new ClubAdmin(towner));
        clubProvider.create(club);
        ClubMember clubMember = new ClubMember(club, towner);
        clubMemberProvider.addMember(club.getId(), clubMember);

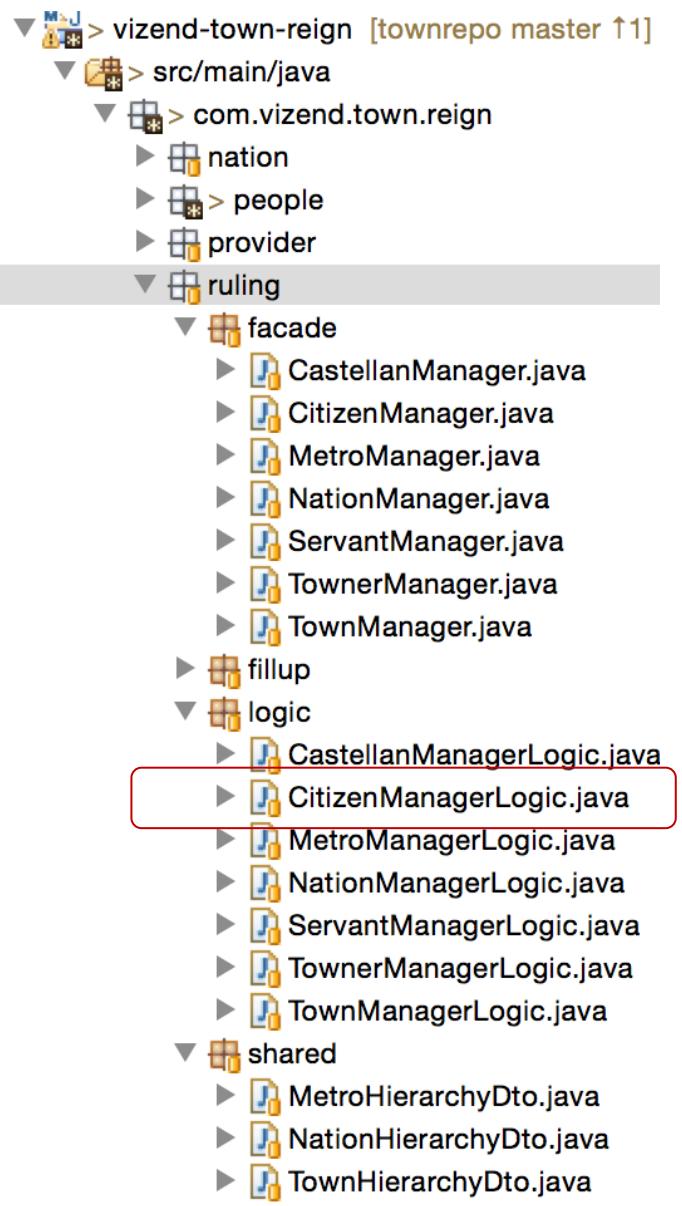
        return club.getId();
    }
}
```

4.5 도메인 컴포넌트(1/3)

- ✓ 도메인 객체가 분명하게 존재하며, 개념을 잘 담고 있습니다. 하지만, 기술에 의존적입니다.
- ✓ 예를 들면, JPA를 사용하여 지속성을 처리할 때, 도메인 객체에 JPA annotation이 포함됩니다.
- ✓ 프로세스와 엔티티 컴포넌트 구분이 명확합니다. 프로세스 로직도 별도로 존재하지만, 기술 종속적입니다.
- ✓ 컴포넌트 플랫폼과 지속성 플랫폼 또는 솔루션을 변경하지 않을 때 유용한 컴포넌트 구조입니다.

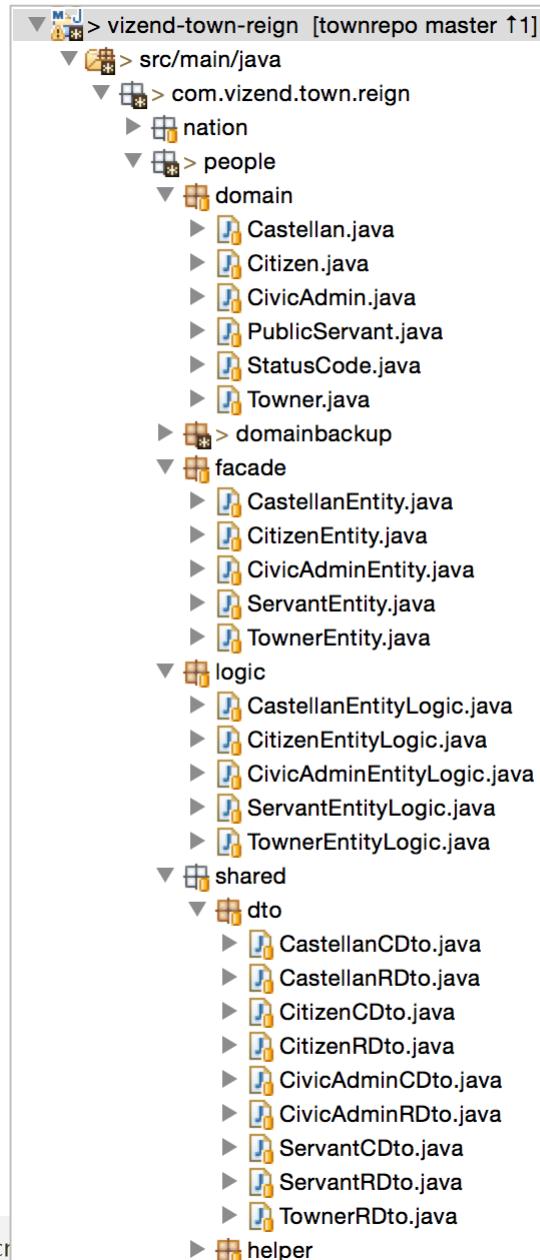


4.5 도메인 컴포넌트(2/3)



```
@Override  
@Transactional(propagation = Propagation.REQUIRED)  
public String registCitizen(String metroId, CitizenCDto citizenCDto) {  
    //  
    // validation  
    if(citizenEntity.existByLoginId(citizenCDto.getLoginId())){  
        throw new VizendTownException("loginId already exists! >> loginId : "+citizenCDto.getLoginId());  
    }  
  
    Castellan castellan = null;  
  
    if (!castellanEntity.existByAuthEmail(citizenCDto.getEmail())) {  
        Nation nation = nationEntity.retrieveByMetroId(metroId);  
        castellan = new Castellan(  
            nation.getId(),  
            townIdGenerator.generateCastellanId(nation.getId()),  
            citizenCDto.getLoginName(),  
            citizenCDto.getEmail());  
        castellanEntity.create(castellan);  
    } else {  
        castellan = castellanEntity.retrieveByAuthEmail(citizenCDto.getEmail());  
    }  
  
    String citizenId = townIdGenerator.generateCitizenId(metroId);  
    Citizen citizen = citizenCDto.createDomain(castellan, metroId, citizenId);  
    citizen.setPassword(EncryptUtils.encrypt(citizenCDto.getPassword()));  
  
    return citizenEntity.create(citizen);  
}
```

4.5 도메인 컴포넌트(3/3)



```
@Repository
public class CitizenEntityLogic implements CitizenEntity {
    //
    @PersistenceContext
    private EntityManager em;

    //-----
    // methods

    @Override
    public String create(Citizen citizen) {
        //
        em.persist(citizen);
        return citizen.getId();
    }

    @Override
    public void createBackup(CitizenBackup citizenBackup) {
        //
        em.persist(citizenBackup);
    }

    @Override
    public Citizen retrieve(String citizenId) {
        //

        String queryStr = "FROM Citizen WHERE id = :citizenId";
        Query query = em.createQuery(queryStr);
        query.setParameter("citizenId", citizenId);

        return SafeQuery.getUniqueResult(query);
    }
}
```

```
@Entity
public class Citizen {
    //
    @Id
    @Column(name = "CITIZEN_ID")
    private String id; // MetroId-ABC123

    @Column(nullable = false)
    private String metroId;

    @Column(nullable = false)
    private String loginId;

    @Column(nullable = true)
    private String loginName;

    @Column(nullable = false)
    private String password;

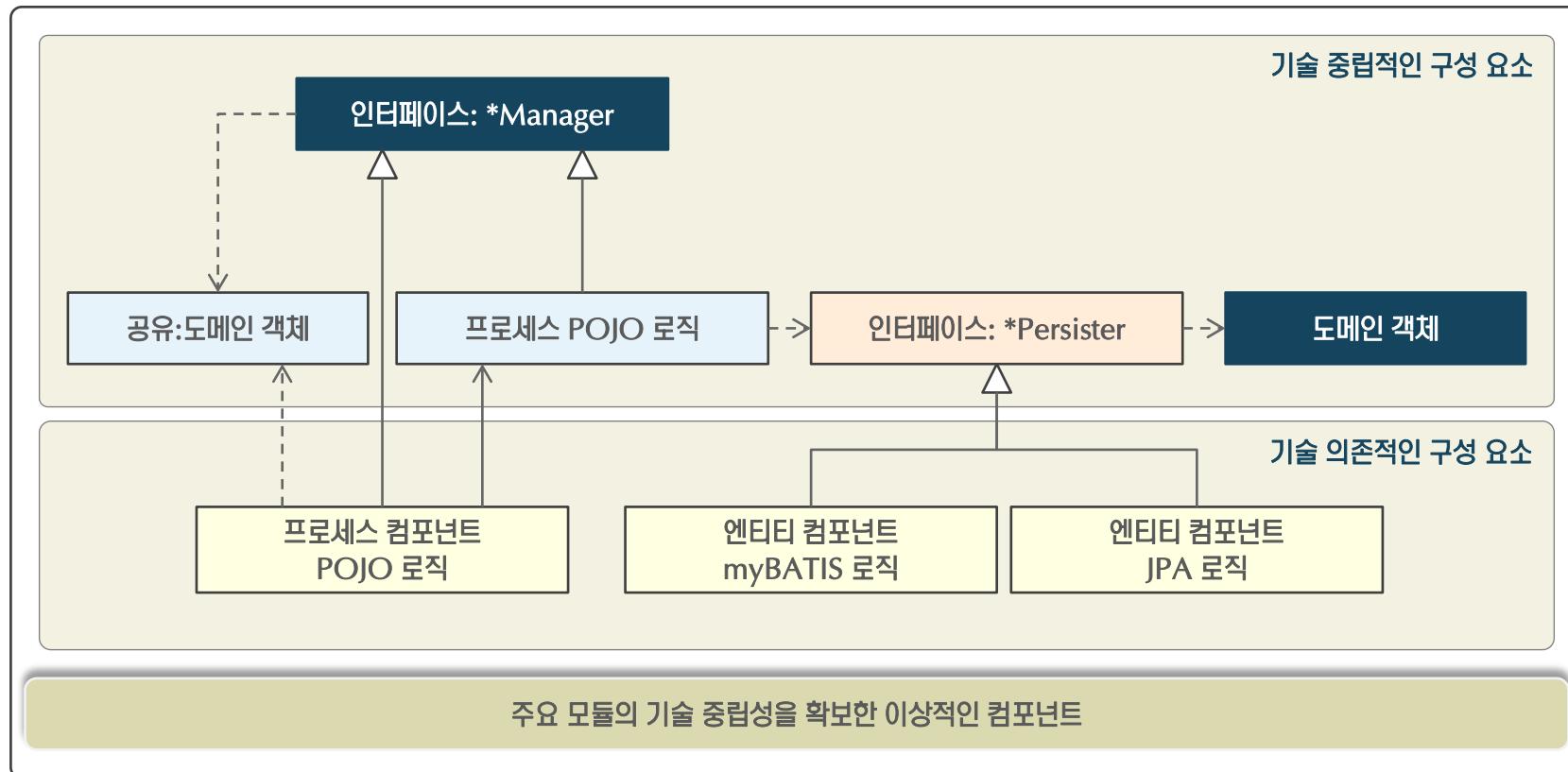
    @Column(nullable = true)
    private String email;

    @Column(nullable = true)
    private String phoneNumber;

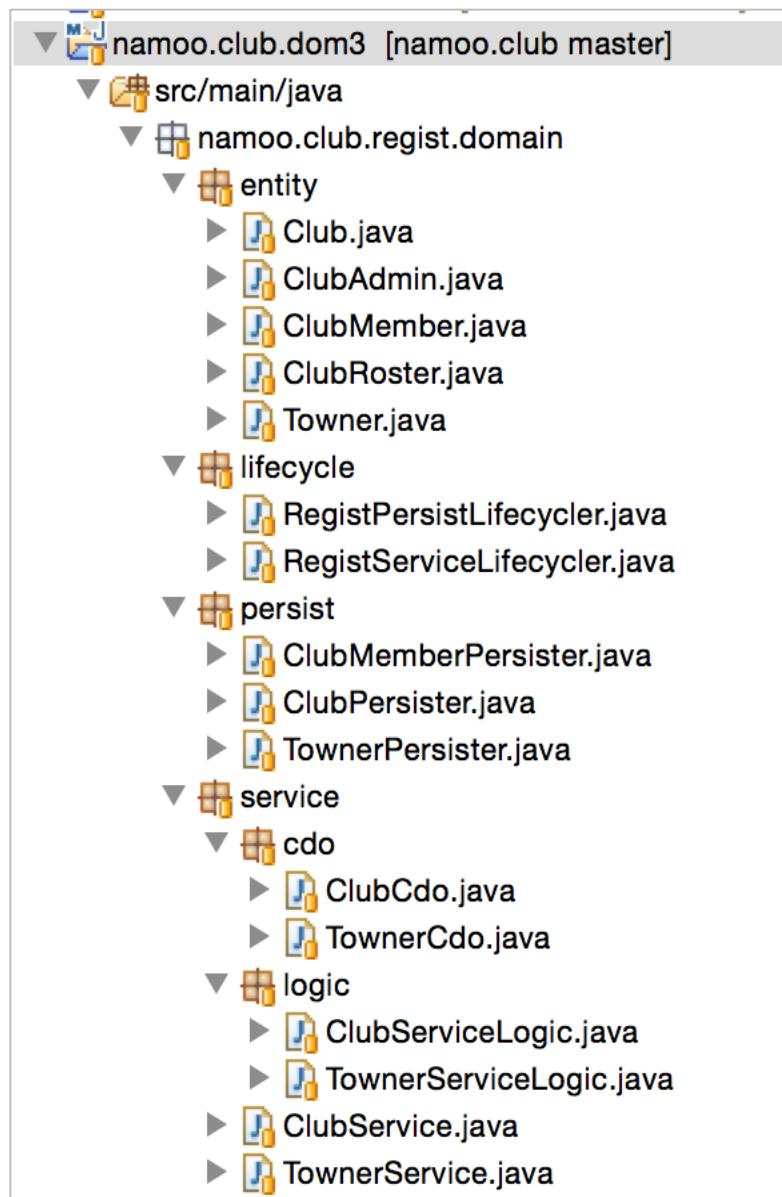
    @Column(nullable = true)
    private String photoId;
}
```

4.6 PURE 도메인 컴포넌트(1/2)

- ✓ 처리 로직이 “프로세스 POJO 로직”으로 구성되어, 어떤 플랫폼에서도 로직을 사용할 수 있습니다.
- ✓ 지속성 처리가 외부의 구현체로 설계하여, 어떤 지속성 구현체가 오든 기술 중립적인 구성 요소는 영향을 받지 않습니다.
- ✓ 가장 이상적인 형태의 컴포넌트 구성이며, 도메인 객체는 지속성이 아닌 외부로 공유되는 구조입니다.
- ✓ 도메인 객체 모델을 오픈하여 공유함으로써 DDD의 “Ubiquitous Language”로 사용합니다.



4.6 PURE 도메인 컴포넌트(2/2)



```
public class ClubServiceLogic implements ClubService {
    //
    private ClubPersister clubPersister;
    private ClubMemberPersister clubMemberPersister;
    private TownerPersister townerPersister;

    /**
     * @param RegistPersistLifecycle lifecycler
     */
    public ClubServiceLogic(RegistPersistLifecycle lifecycler) {
        //
        this.clubPersister = lifecycler.getClubPersister();
        this.clubMemberPersister = lifecycler.getClubMemberPersister();
        this.townerPersister = lifecycler.getTownerPersister();
    }

    /**
     * @Override
     */
    public String registerClub(ClubCdo clubCdo) throws NamooClubException {
        //
        if (clubPersister.isExistClubByName(clubCdo.getClubName())) {
            throw new NamooClubException("Already existed club --> " + clubCdo.getClubName());
        }

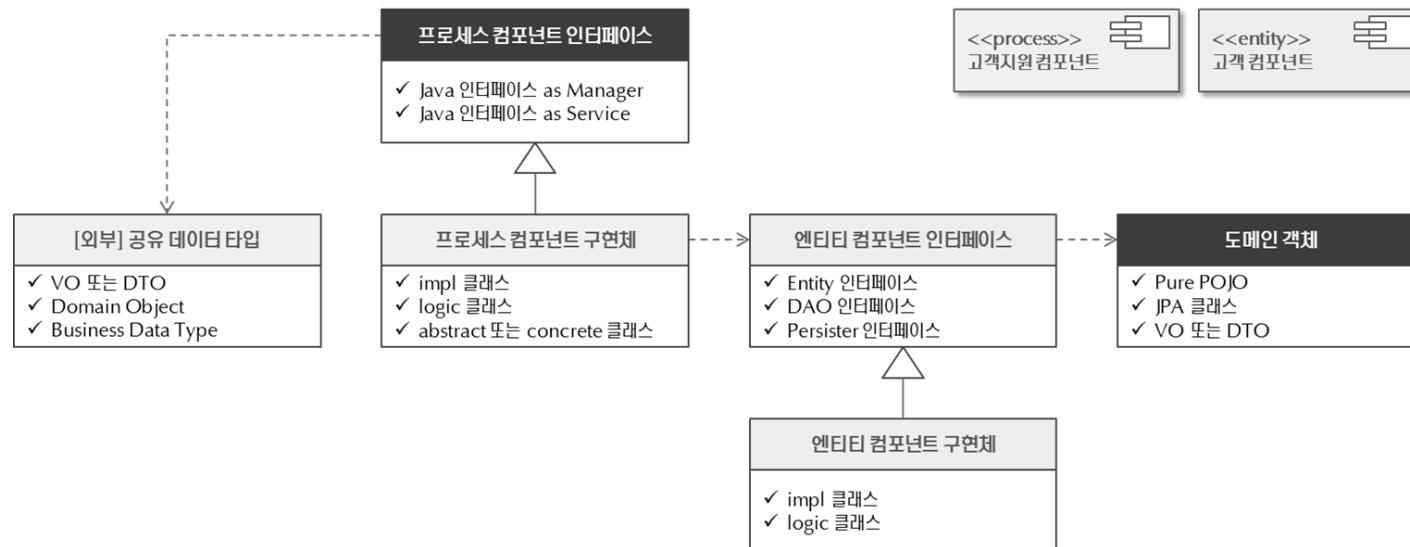
        Towner towner = null;
        try {
            towner = townerPersister.retrieveByEmail(clubCdo.getAdminEmail());
        } catch (EmptyResultException e) {
            throw new NamooClubException(e);
        }

        Club club = clubCdo.createDomain(new ClubAdmin(towner));
        clubPersister.create(club);
        ClubMember clubMember = new ClubMember(club, towner);
        clubMemberPersister.addMember(club.getId(), clubMember);

        return club.getId();
    }
}
```

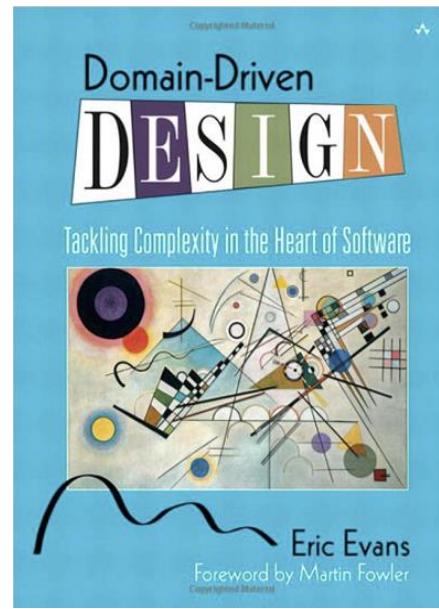
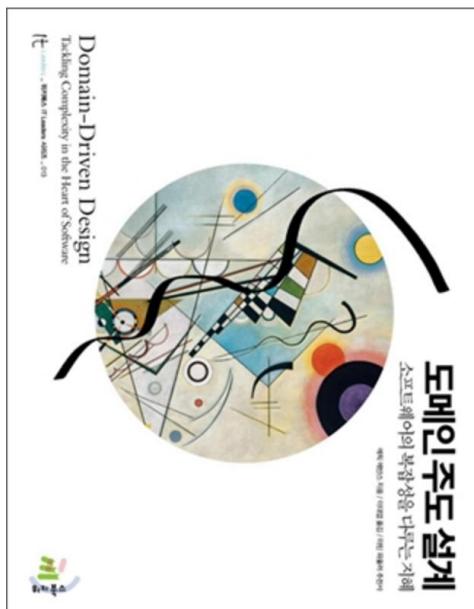
요약

- ✓ 아키텍처 설계에서 컴포넌트 구조 설계는 아주 중요합니다. 비즈니스를 담는 그릇입니다.
- ✓ 컴포넌트 구조에 따라 품질(확장성, 변경용이성, 유지보수성 등)과 개발 공수에 차이가 있습니다.
- ✓ 컴포넌트 구조는 개발 팀의 기술 수준, 개발 환경, 고객 니즈 등에 따라 서로 다릅니다.
- ✓ 다양한 컴포넌트 구조에 대한 올바른 이해가 있어야, 상황에 맞는 컴포넌트 구조를 설계할 수 있습니다.





5. Domain Driven Design 기초



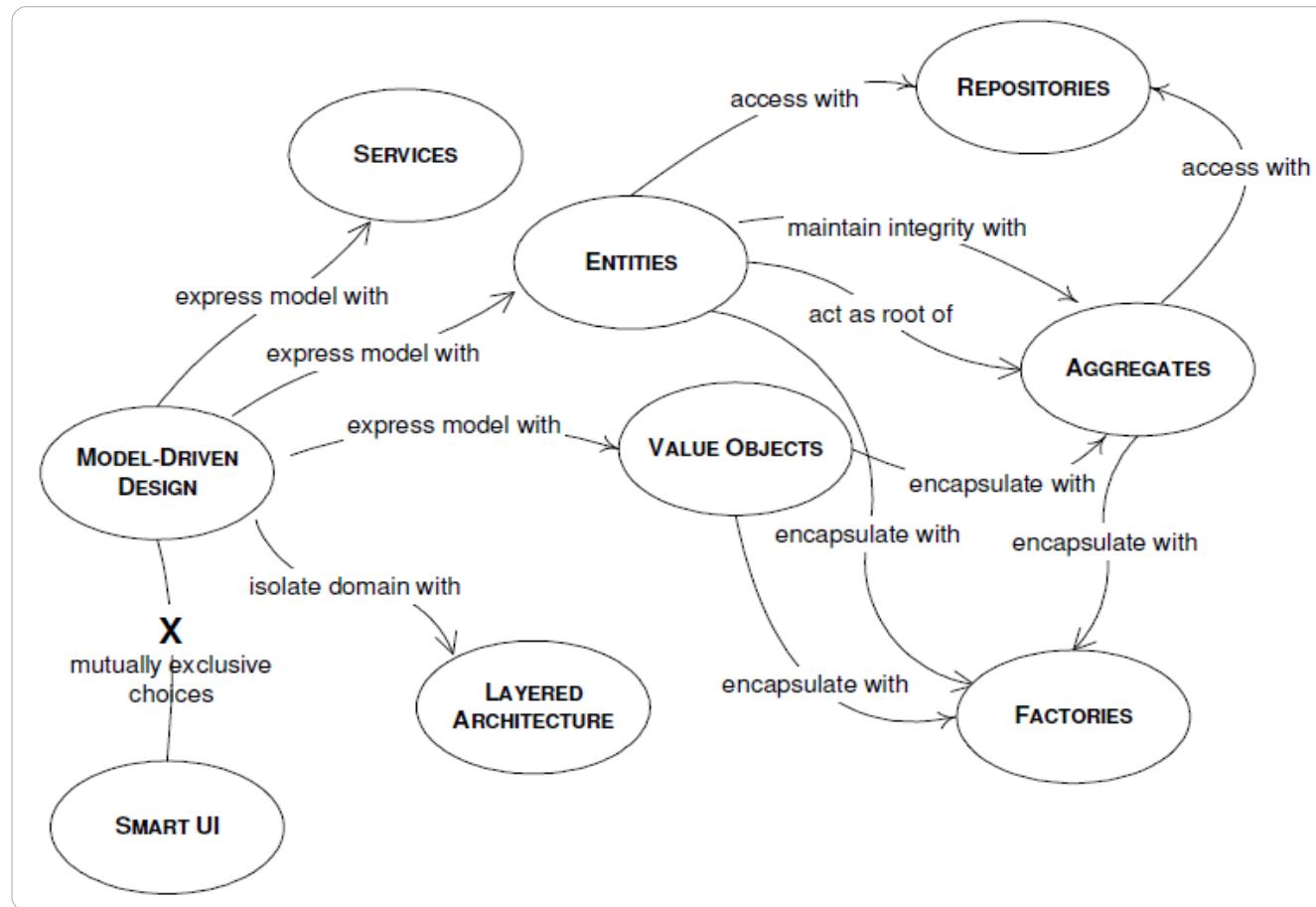
5.1 DDD 기본 블럭

5.2 모델요소 – Entity, Value Object, Service

5.3 도메인객체 생명 주기 – Aggregate, Factory, Repository

5.1 DDD 기본 블럭

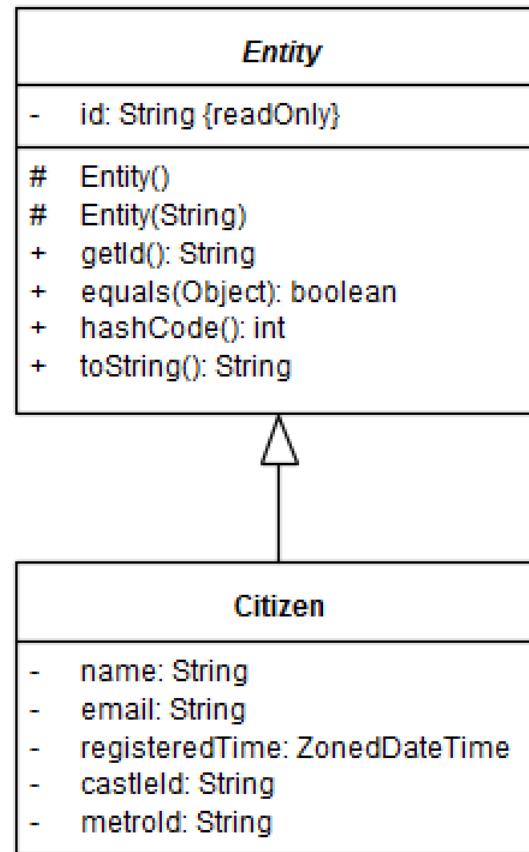
- ✓ DDD를 이끌어 가는 기본 개념(패턴, 또는 패턴 언어) 간의 관계는 navigation map에서 확인할 수 있습니다.
- ✓ 도메인을 표현하는 기본 모델 요소는 Entities, Value Objects, Services 세가지 입니다.
- ✓ 세 요소를 효율적으로 표현하기 위해 Repositories, Aggregates, Factories 세 가지 개념을 추가했습니다.



[A navigation map of the language of Model-Driven Design, Eric Vans]

5.2 모델 요소[1/6] – Entity 1

- ✓ 객체는 다른 객체와 식별이 되고, 식별 자체가 연속성을 갖는가를 기준으로 정의합니다.
- ✓ 따라서, 엔티티는 생명주기 동안 형태와 내용이 변경될 수도 있지만, 연속성은 유지해야 합니다.
- ✓ Entity는 상태를 변경할 수 있으며, Value Object 와는 다른 라이프 사이클을 가집니다.



5.2 모델 요소[2/6] – Entity 2

- ✓ Entity는 별도로 분리되어 있을 때, 자신의 책임을 가장 잘 수행합니다.
- ✓ Entity를 정의할 때, 속성이나 행위에 집중하기 보다는 본질적인 특성에 초점을 두고 정의합니다.
- ✓ 아래 abstract 클래스인 Entity는 Identity를 설정하고, 유지하고, 비교에 활용하는 기능을 제공합니다.

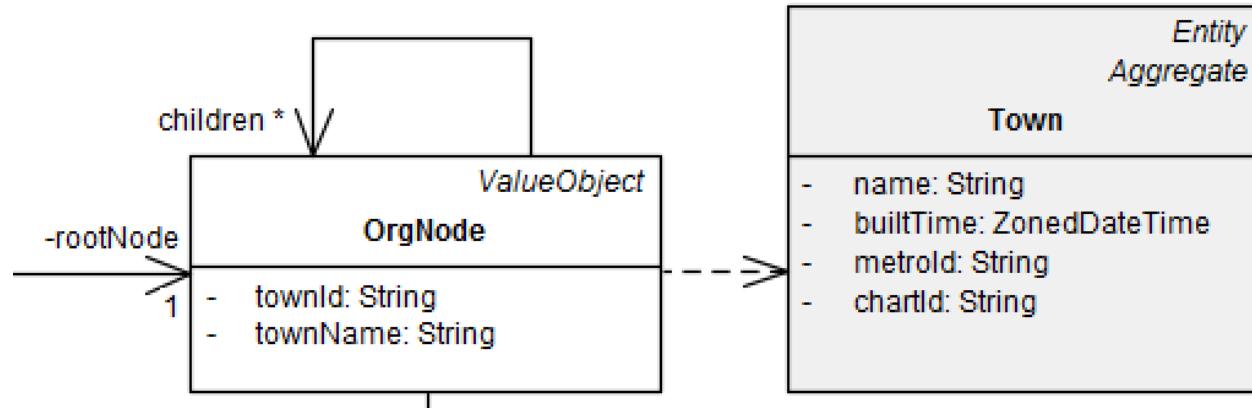
<pre>public abstract class Entity { // private final String id; protected Entity() { this.id = UUID.randomUUID().toString(); } protected Entity(String id) { this.id = id; } public String getId() { return this.id; } }</pre>	<pre>public boolean equals(Object target) { if(this == target) { return true; } else if(target != null && this.getClass() == target.getClass()) { Entity entity = (Entity)target; return Objects.equals(this.id, entity.id); } else { return false; } public int hashCode() { return Objects.hash(new Object[]{this.id}); } public String toString() { return "id:" + this.id; } }</pre>
---	--

5.2 모델 요소[3/6] – Value Object 1

- ✓ 개념적으로 식별이 필요없고, 단순히 값만을 갖고 있는 객체를 Value Object라고 합니다.
- ✓ 따라서 Value Object는 값을 변경할 수 없는, 즉 상태를 변경할 수 없는 Immutable 객체입니다.
- ✓ 한 번 만들어진 객체의 값은 불변이므로 전송이나 멀티 스레드 환경에서 안전성을 보장합니다.

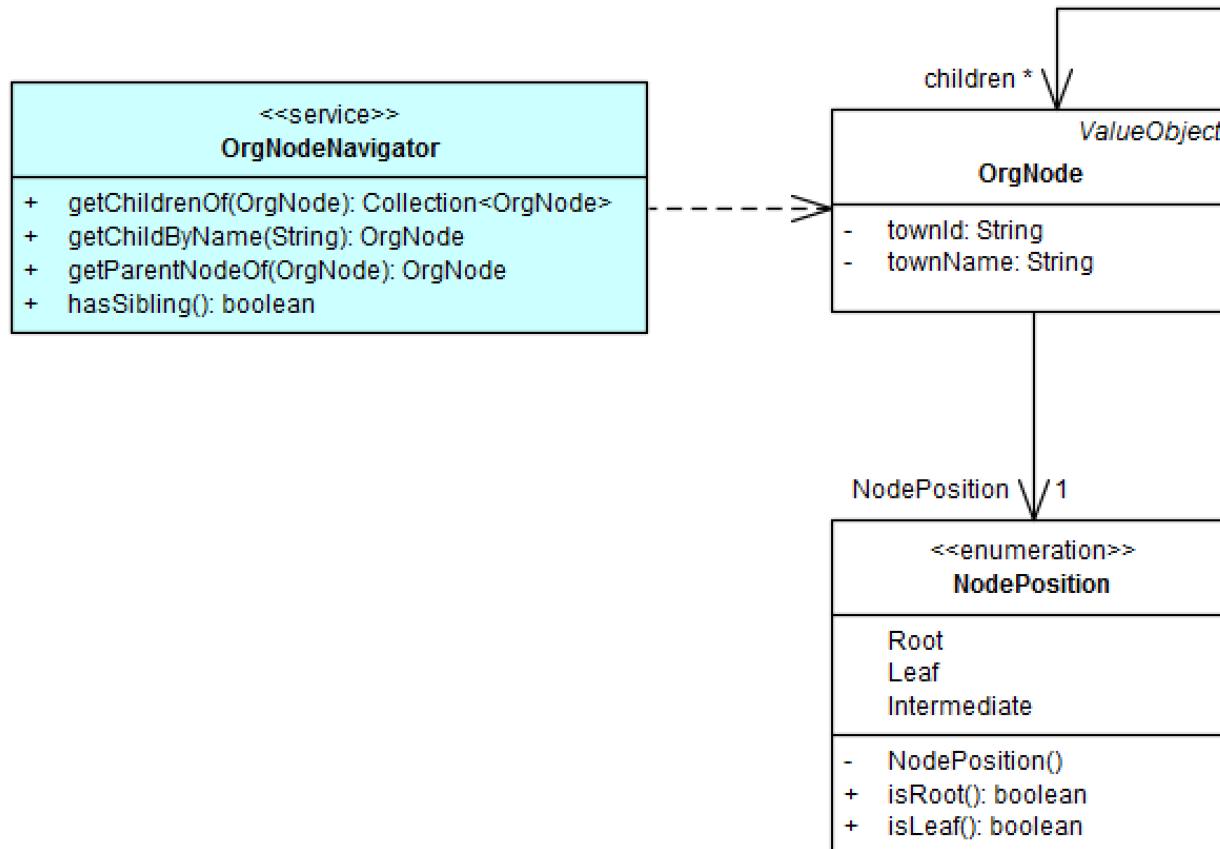
5.2 모델 요소[4/6] – Value Object 2

- ✓ Value Object 라고 반드시 단순한 것 만이 아니라, 복합 객체 형식을 가진 것도 있을 수 있습니다.
- ✓ Value Object 도 Entity를 참조할 수 있고, 통신 구간에서 값을 전달하는 용도로 사용할 수 있습니다. → DTO
- ✓ Address 라고 모두 Value Object가 아닙니다. 우체국의 주소체계를 관리 SW에서 Address는 Entity입니다.
- ✓ Value Object 가 서로 참조하는 것은, Value Object가 식별성이 없으므로 표현상의 오류입니다.



5.2 모델 요소[5/6] – Service 1

- ✓ 설계가 매우 명확하고 실용적이더라도 어떠한 객체에도 포함할 수 없는 연산을 Service로 정의합니다.
- ✓ 이러한 연산은 본질적으로 사물이 아니라 활동(activity)이나 행동(action)입니다.
- ✓ 복잡한 연산을 분리하지 못하면, 객체를 복잡하게 하거나 개념을 흩어 놓아 모델을 망칩니다.

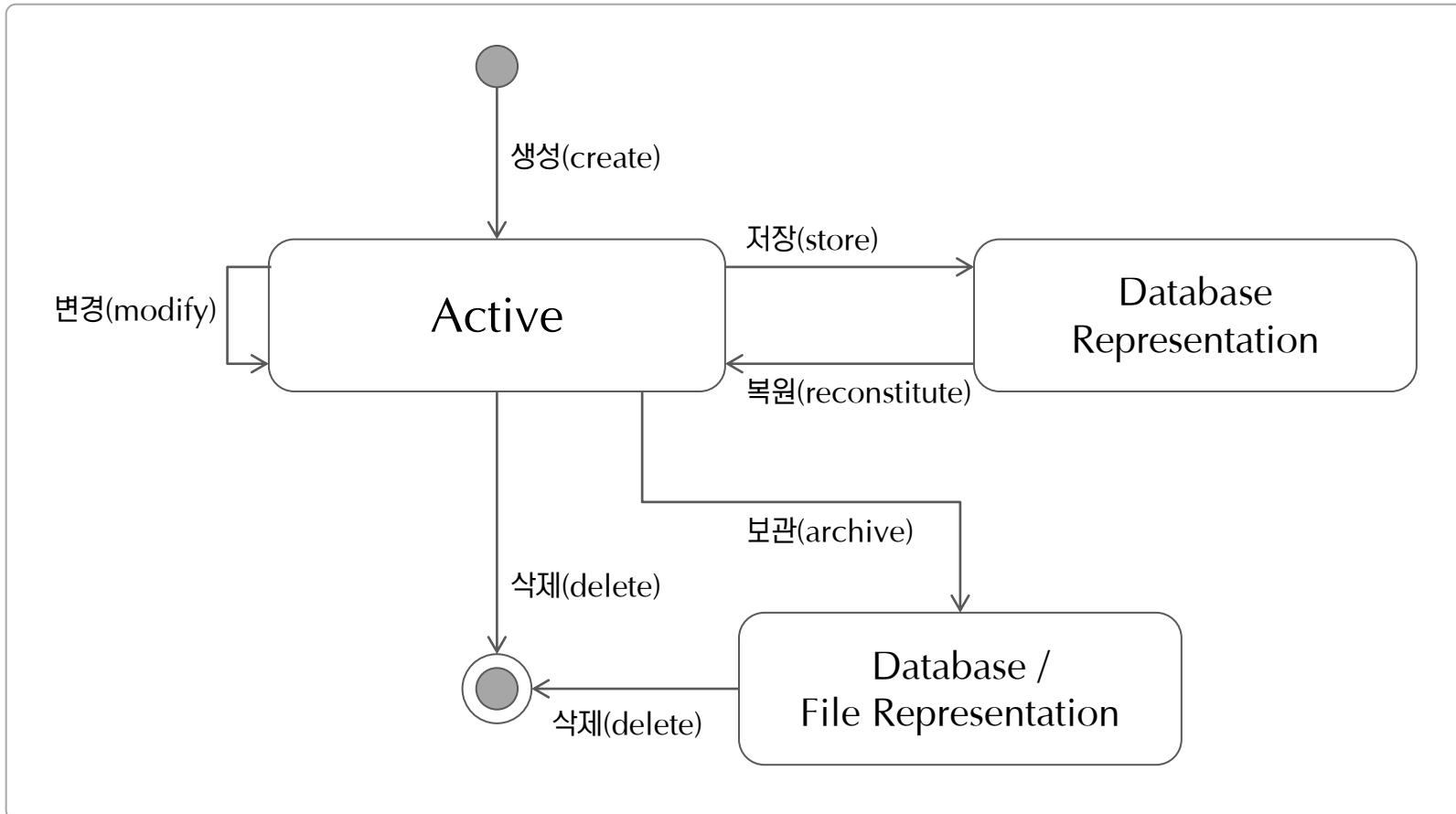


5.2 모델 요소[6/6] – Service 2

- ✓ 연산이 Entity나 Value Object의 한 부분이 아니라 도메인 개념과 관련이 있을 때, Service로 정의합니다.
- ✓ 도메인 모델의 외적인 요소에 의해 인터페이스가 결정될 때, Service로 정의합니다.
- ✓ 도메인의 프로세스나 연산이 어떤 Entity나 Object Value의 책임이 아니라면 Service로 정의합니다.
- ✓ 모델의 언어라는 측면에서 인터페이스를 정의하고, Ubiquitous Language로 이름을 정합니다.
- ✓ 기술적인 요소를 포함하고 있는 것은 응용 서비스입니다. 도메인 Service와 철저하게 분리합니다. → File, Excel

5.3 도메인 객체의 생명 주기(1/7)

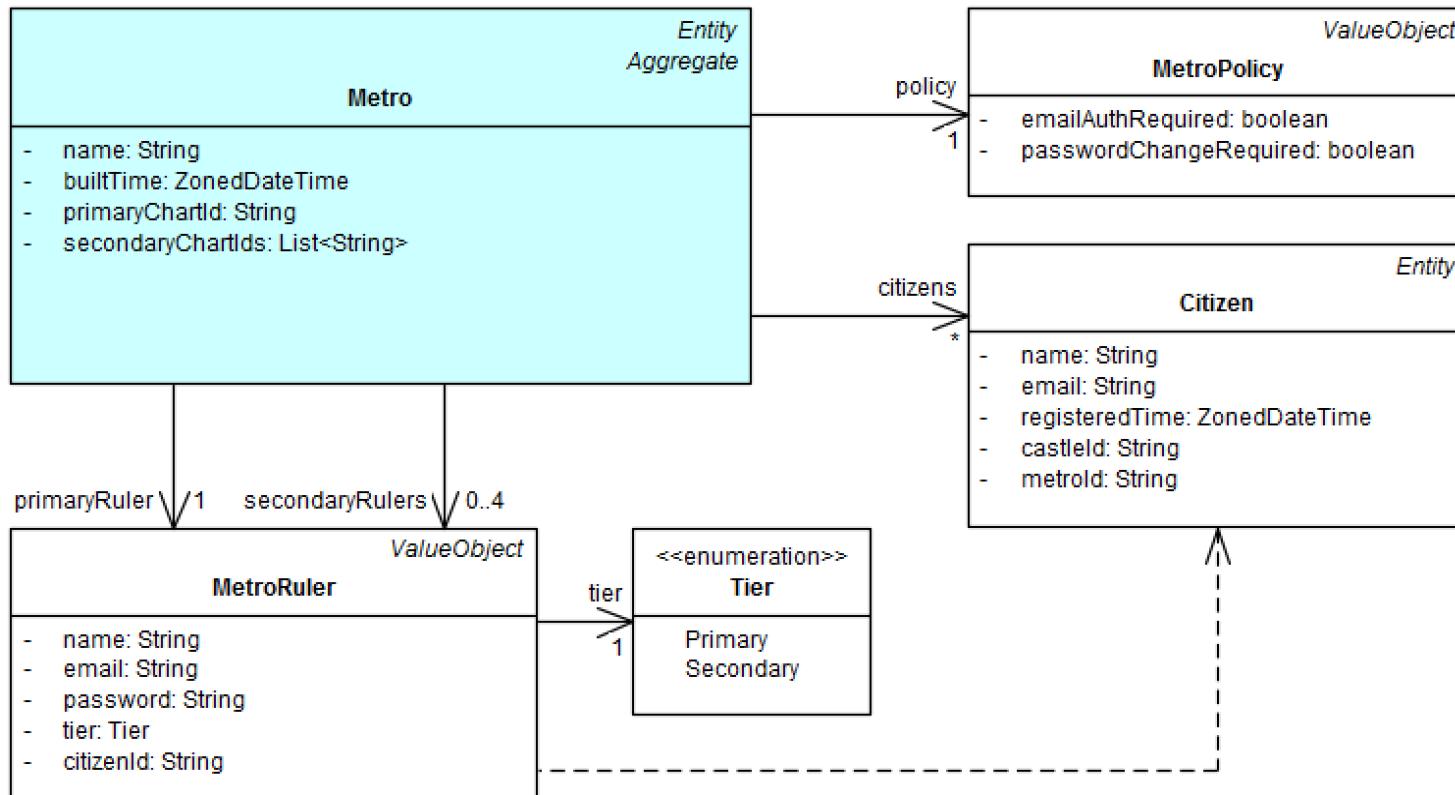
- ✓ 도메인 객체는 생성으로부터 소멸에 이르기까지 세 가지 상태를 가질 수 있습니다.
- ✓ 정보 처리를 위해 비즈니스 로직 영역(메모리 영역)으로 읽어 들였을 때를 활성(active) 상태라고 합니다.
- ✓ 지속성을 보장하고 필요할 때 사용하려면 DB에 저장하거나, 오랫동안 보관을 위해 파일/DB로 저장합니다.



[A lifecycle of a domain object, Eric Vans]

5.3 도메인 객체의 생명 주기(2/7) – Aggregate 1

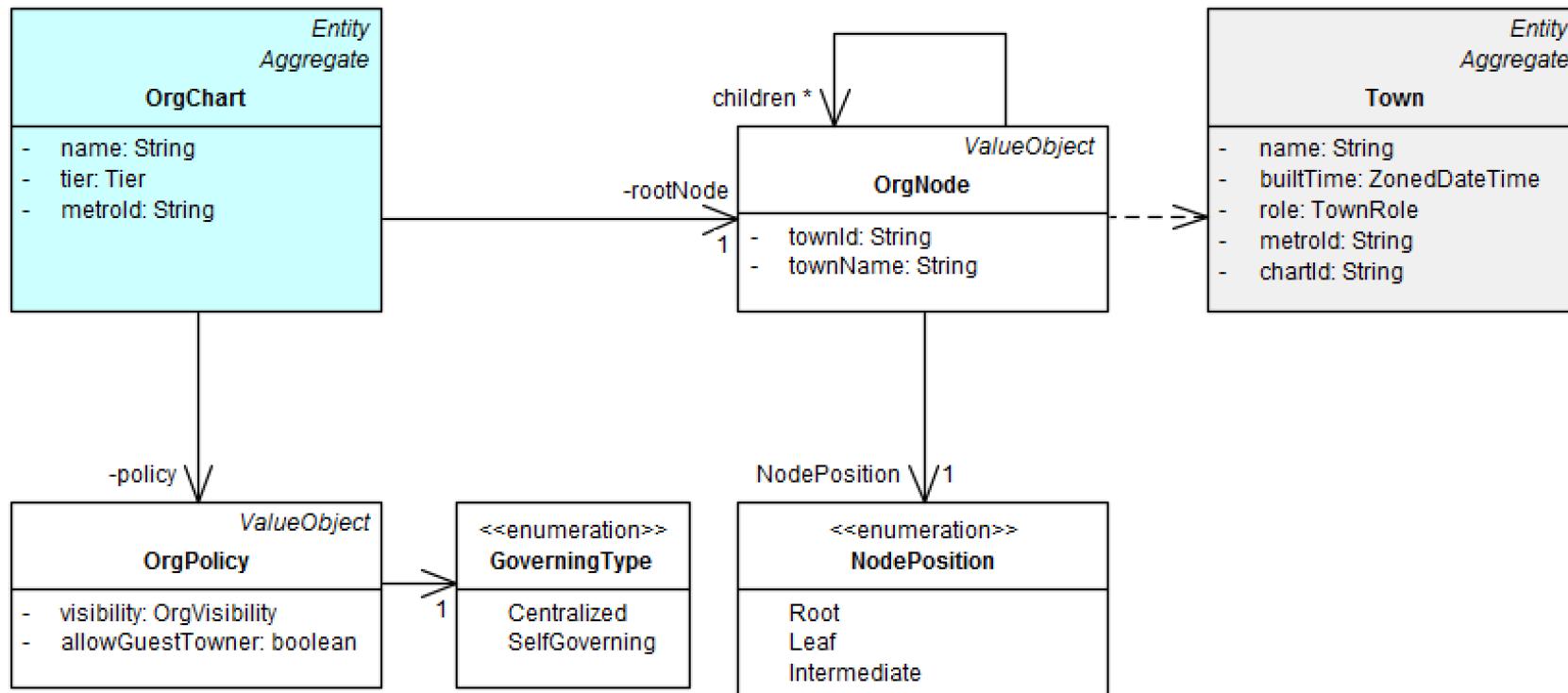
- ✓ 업무 관점의 응집도를 유지하는 선에서 연관관계를 가능한 단순하게 유지합니다.
- ✓ Aggregate은 소유권과 경계를 명확하게 정의하여, 객체 관계에 질서를 부여합니다.
- ✓ 도메인 객체를 처리하는 트랜잭션 단위이며 따라서 무결성을 유지하는 단위이기도 합니다.



[Metro as an Aggregate root]

5.3 도메인 객체의 생명 주기(3/7) – Aggregate 2

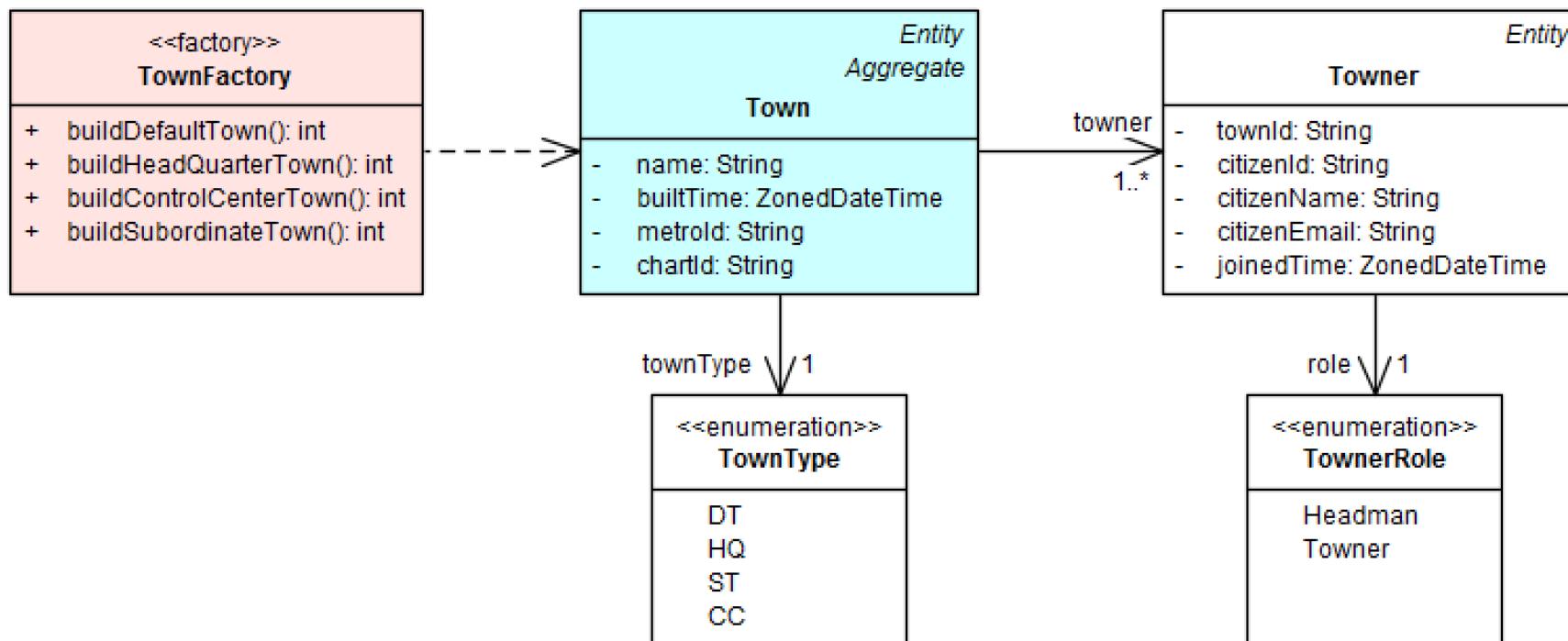
- ✓ Aggregate 루트는 Identity를 가지는 Entity이면서 동시에 내부 객체들에 대한 생명주기를 책임집니다.
- ✓ Aggregate 루트는 전역 식별 범위를 가지며, 내부의 Entity에 대한 참조를 외부로 전달할 수 있습니다.
- ✓ Aggregate 안의 객체는 서로 참조할 수 있지만, 외부 Aggregate는 루트만 참조할 수 있습니다.



[OrgChart as an Aggregate root]

5.3 도메인 객체의 생명 주기(4/7) – Factory 1

- ✓ Aggregate을 생성하는 일이 복잡하거나, 내부 구조를 많이 드러내어야 하는 경우, Factory를 사용합니다.
- ✓ 복잡한 생성 과정을 Aggregate에게 맡기면, Aggregate은 내부 구조를 관리하는 수준의 복잡한 일을 합니다.
- ✓ 외부의 클라이언트에게 생성을 넘기면, Encapsulation 원칙을 깨는 결과를 가져옵니다.

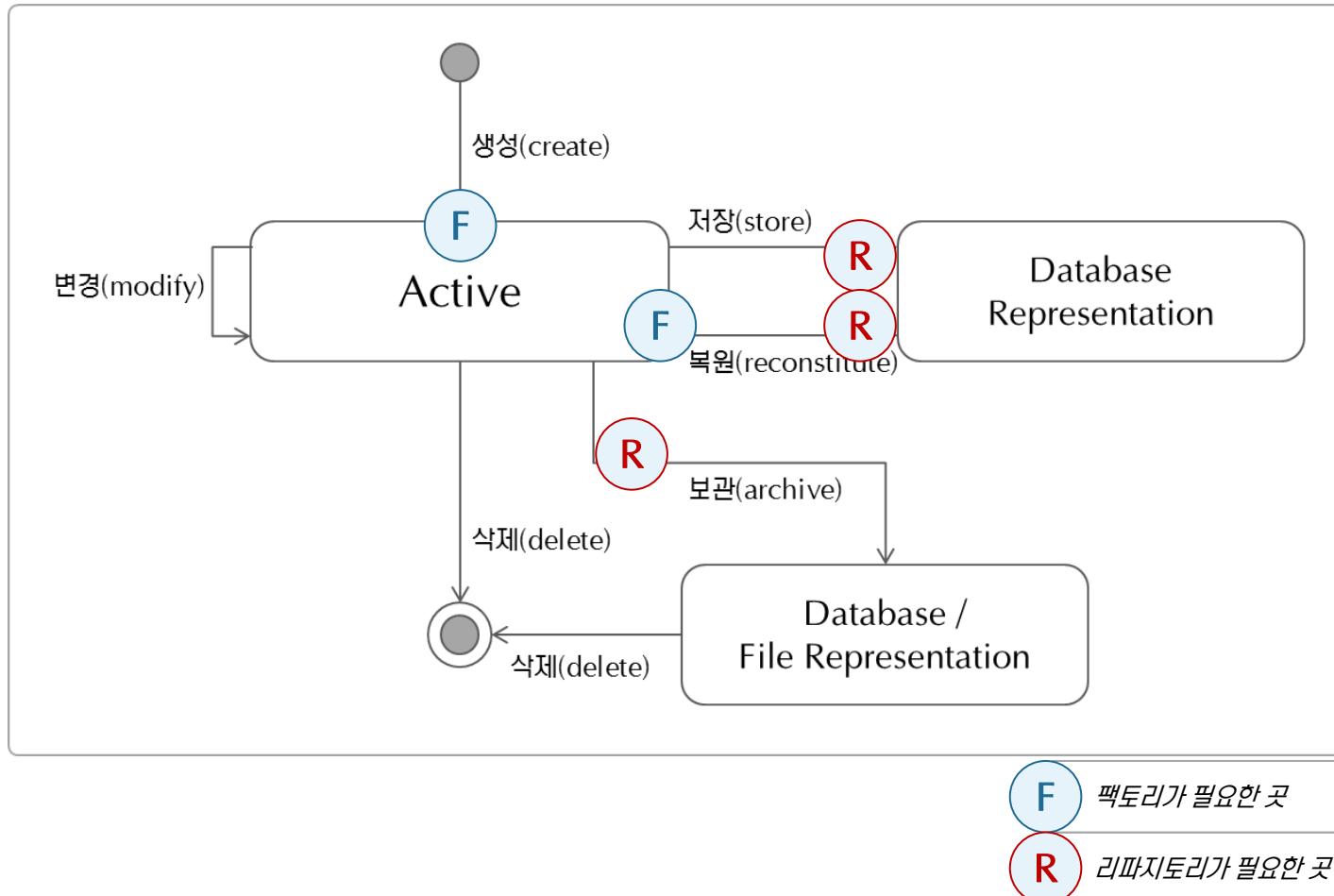


5.3 도메인 객체의 생명 주기(5/7) – Factory 2

- ✓ 생성이 상대적으로 간단한 경우, Aggregate 루트에 Factory 메소드를 둡니다.
- ✓ Aggregate 내부에 있는 어떤 Entity가 Aggregate 바깥에 있어야 하는 Entity를 만듭니다. → 정보가 있으므로
- ✓ Aggregate 외부에 별도의 Factory 객체를 두는 방식으로 설계합니다.
- ✓ Entity를 위한 Factory를 필요한 최소한의 매개변수를 넘겨 받아 Entity를 생성하고 나머지 값은 후에 받습니다.
- ✓ Entity를 위한 Factory는 식별을 위한 Identity 값을 외부로부터 넘겨 받거나 내부에서 만듭니다.
- ✓ Value object를 위한 Factory는 한번에 완전한 Value object를 만들어야 합니다. → 상태 변화가 없으므로
- ✓ Factory는 생성 뿐만 아니라 복원(reconstitute) 활동 역시 지원하여야 합니다.

5.3 도메인 객체의 생명 주기(6/7) – Repository 1

- ✓ Factory의 Repository는 생명 주기의 시작 점을 다루는 측면이 있으므로 차이점을 알기 어렵습니다.
- ✓ Factory는 생성과 복원(또는 재구성)에 사용하고, Repository는 저장과 찾아오기에 사용합니다.
- ✓ Repository는 도메인을 기술과 구분하는 경계 지점 역할을 합니다.



5.3 도메인 객체의 생명 주기(7/7) – Repository 2

- ✓ Repository를 설계할 때, 해당 타입의 객체로 구성한 컬렉션이 있다고 가정하고 인터페이스를 정의합니다.
- ✓ Repository에 객체를 추가하고 삭제하는 메소드를 제공하여, 물리적인 저장소로의 저장과 질의를 추상화합니다.
- ✓ Repository는 Aggregate 루트에만 제공하고, 모든 내부 객체 저장과 접근을 위임합니다.

❖ Repository의 장점

1. Repository는 저장된 객체를 획득하고, 객체의 생명주기를 관리하는 단순한 모델을 제시한다.
2. Repository는 여러 저장 방식으로부터 도메인을 분리하여 준다.
3. Repository 인터페이스를 활용하면 테스트를 용 Mock-up 객체를 쉽게 만들 수 있다.



6. Microservices

이 장은 마틴 폴러 사이트의 자료를 정리한 것입니다.

<http://martinfowler.com/articles/microservices.html>

6.1 Microservice 개요

6.2 Microservice 특징

6.3 기업 시스템과 Microservice

6.1 Microservices – 개요(1/4)

- ✓ “마이크로서비스” 아키텍처 스타일은 하나의 애플리케이션을 여러 개의 작은 서비스 집합으로 개발하는 접근방법입니다.
- ✓ 각 서비스는 개별 프로세스에서 실행되고 HTTP 자원 API와 같은 가벼운 수단을 사용하여 서로 통신합니다.
- ✓ 서비스는 비즈니스 기능 단위로 구성하며 자동화된 배포 방식을 이용하여 독립적으로 배포할 수 있습니다.
- ✓ 서비스들에 대한 중앙 집중적인 관리는 최소화하고, 각 서비스는 서로 다른 언어와 데이터 저장기술을 사용할 수 있습니다.

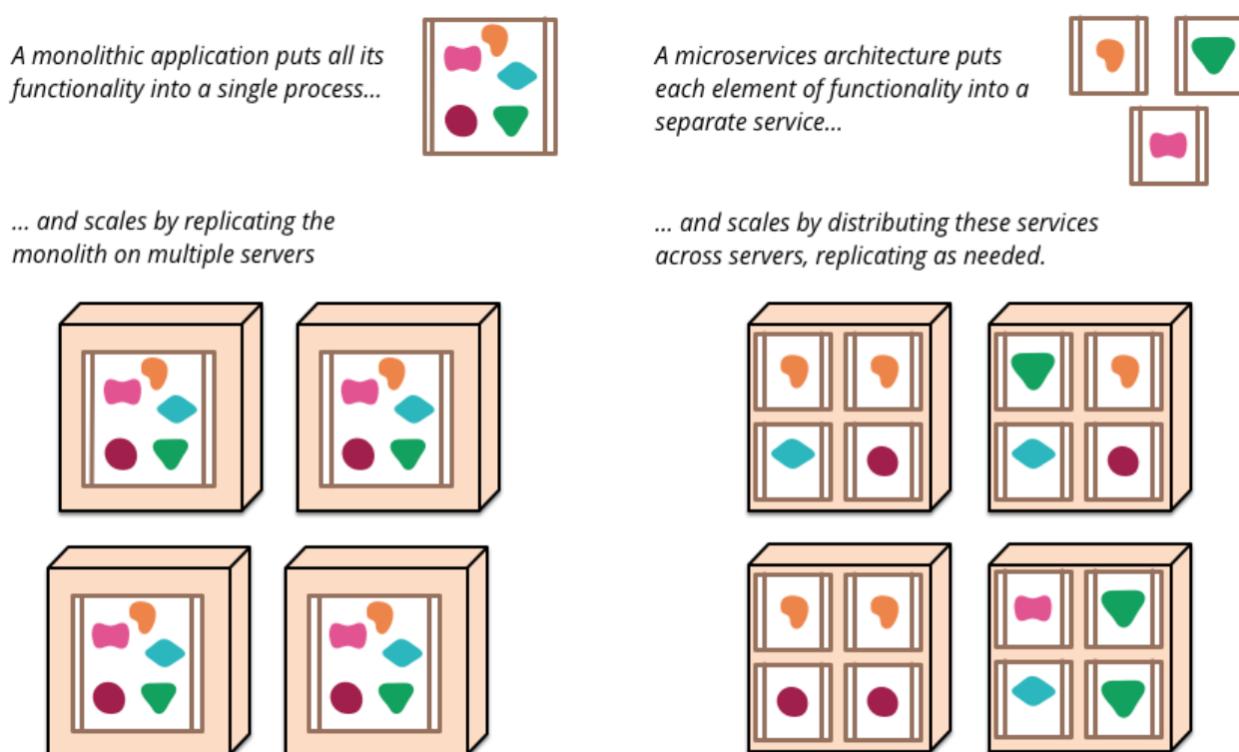


Figure 1: Monoliths and Microservices

<http://martinfowler.com/articles/microservices.html>

6.1 Microservices – 개요(2/4)

- ✓ 마이크로서비스와 반대 개념은 하나의 단위로 개발되는 일체식(monolithic) 스타일 애플리케이션 설계입니다.
- ✓ 기업 애플리케이션은 클라이언트 쪽의 사용자 인터페이스와 데이터베이스, 그리고 서버 쪽 애플리케이션 세 개로 구성합니다.
- ✓ 서버 쪽 애플리케이션은 HTTP 요청을 처리하고, 도메인 로직을 실행하고, 데이터 베이스로부터 데이터를 찾거나 갱신하고, 브라우저로 보낼 HTML 뷰를 선택하고 생성하여 전송합니다.
- ✓ 서버 쪽 애플리케이션이 일체 (monolith), 즉 논리적인 단일 실행체이며, 아무리 작은 변화에도 새로운 버전을 생성합니다.

6.1 Microservices – 개요(3/4)

- ✓ 일체식(monolithic) 서버에서 요청을 처리하는 모든 로직은 단일 프로세스에서 실행됩니다.
- ✓ 로직 개발을 위해 클래스, 기능, namespace 등과 같은 언어의 특성을 활용합니다.
- ✓ 개발자의 컴퓨터에서 애플리케이션을 개발하고 테스트 한 다음, 배포 메커니즘을 거쳐 운영 서버에 배포합니다.
- ✓ 로드 밸런서 뒤에 여러 인스턴스를 두고 그 곳에 모노리스(monolith)를 수평으로 확장합니다.

- ✓ 모노리스도 충분히 훌륭할 수 있지만, 많은 사람들이 좌절을 느끼기 시작했습니다.
- ✓ 특히 클라우드에 여러 개의 모노리스를 설치하고 사용할 때 그런 경우가 많아지기 시작했습니다.
- ✓ 작은 변화라도 생기면 모든 모노리스를 다시 빌드하고 배포해야 하는 것은 큰 골치거리였습니다.
- ✓ 시간이 흐를 수록 모듈 체계는 무너지고 변경으로 인한 파급효과를 모듈 안으로 가두는데 실패하였습니다.
- ✓ 필요한 특정 기능만 확장(scale-up)할 수 없고, 전체 애플리케이션을 동시에 확장해야 합니다.

6.1 Microservices – 개요(4/4)

- ✓ 이러한 좌절은 사람들을 마이크로서비스 아키텍처 스타일로 이끌었습니다.
- ✓ 서비스는 독립적으로 배포, 확장(scale-up)할 수 있고, 확고한 모듈 경계를 가지며, 서로 다른 언어로 개발할 수 있습니다.
- ✓ 각 서비스는 서로 다른 팀이 관리를 할 수 있습니다.
- ✓ 마이크로서비스 스타일은 혁신적이 아니라 유닉스 설계의 개념과 같으며, 이전에 사용하던 설계 방식보다 더 훌륭합니다.

6.2 MSA 특징(1/9) – 서비스를 통한 컴포넌트화

- ✓ 소프트웨어 컴포넌트를 서로 연결하여 시스템을 구축하려는 바램은 오랫동안 지속되어 왔습니다.
- ✓ 컴포넌트의 정의에 대해서 여러 의견이 있지만, 여기서는 독립적으로 대체하거나 업그레이드가 가능한 SW 단위입니다.
- ✓ 라이브러리는 프로그램에 연결되고 메모리 안에서 호출이 가능한 컴포넌트로 정의합니다.
- ✓ 서비스는 웹서비스나 RPC호출 방식으로 서로 연결하는 프로세스 밖의 컴포넌트로 정의합니다.

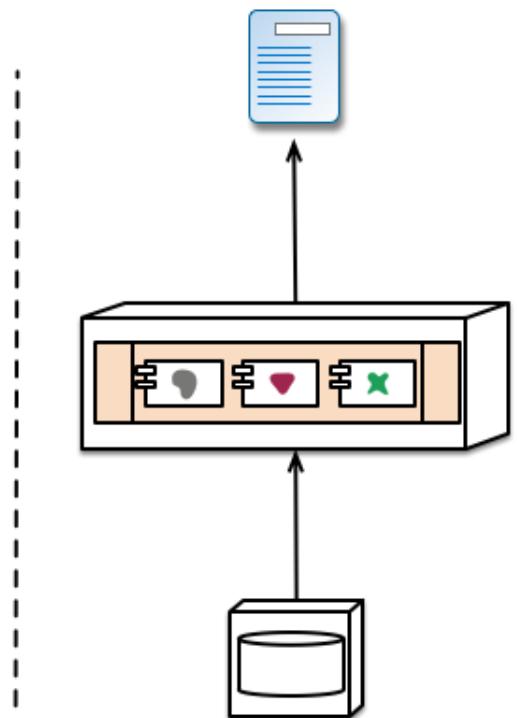
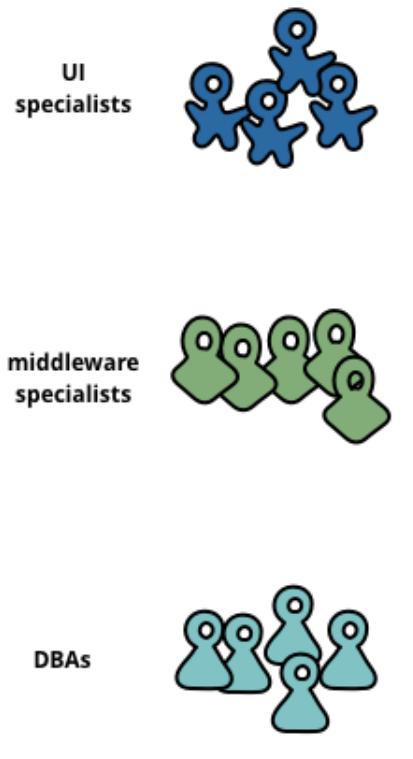
- ✓ 서비스를 (라이브러리가 아닌) 컴포넌트로 간주하는 이유는 독립적으로 배포 가능하기 때문입니다.
- ✓ 단일 프로세스 안에 여러 라이브러리로 구성한 애플리케이션이 있다면, 어떤 라이브러리의 작은 변화도 재배포를 요구합니다.
- ✓ 애플리케이션을 여러 서비스로 구성하면, 어떤 서비스가 변경되었을 때 그 서비스만 다시 배포하면 됩니다.
- ✓ 물론 인터페이스 변화는 다른 서비스에 변화를 가져올 수 있지만 응집된 서비스 경계 개념과 서비스 계약 안의 점진적인 개선을 통해 파급 효과를 최소화 할 수 있습니다.

- ✓ 보다 명확한 또는 명시적인 원격 호출 대상인 서비스 인터페이스를 정의하고 사용할 수 있습니다.
- ✓ 서비스를 원격으로 호출하는 것은 메모리 호출보다 많은 비용을 요구하며, 그 결과 거친-입자 인터페이스로 흘러갑니다.
- ✓ 거친-입자 인터페이스는 고운-입자 인터페이스 보다 사용하기 어려울 수도 있습니다.
- ✓ 컴포넌트 간에 책임을 다시 배정할 때, 프로세스 경계를 넘어서 해야 한다면 훨씬 더 어려울 수 있습니다.

- ✓ 한 서비스는 여러 프로세스로 구성될 수 있지만 함께 개발하고 배포하여야 하고 DB는 그 서비스만 사용하여야 합니다.

6.2 MSA 특징(2/9) – 비즈니스 역량 기반 팀 1

- ✓ 하나의 대규모 애플리케이션을 나눌 때, UI, 서버 로직, DB와 같은 기술을 기준으로 나누는 경향이 있습니다.
- ✓ 이런 방식의 팀 구조에서는 작은 변화라도 여러 팀의 협업이 필요하며, 시간과 예산이 많이 들어갈 수 밖에 없습니다.



... lead to siloed application architectures.
Because Conway's Law

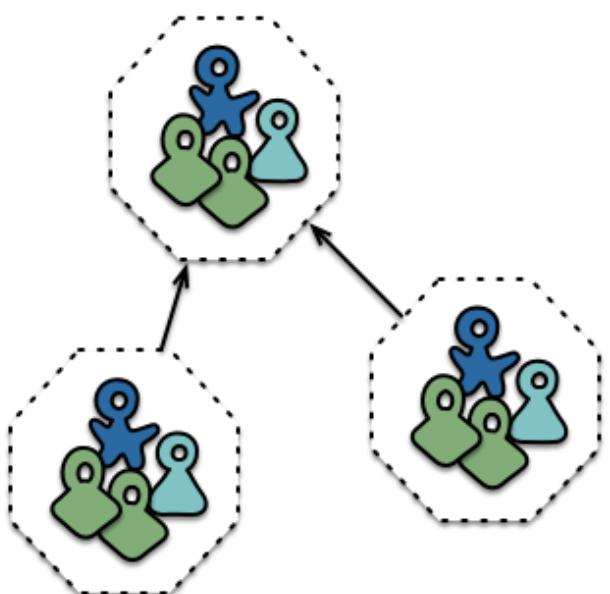
시스템을 설계하는 조직은 누구나 조직의 소통 구조를
그대로 복제하는 구조를 갖도록 시스템을 설계한다.

-- 멜빈 콘웨이, 1967

<http://martinfowler.com/articles/microservices.html>

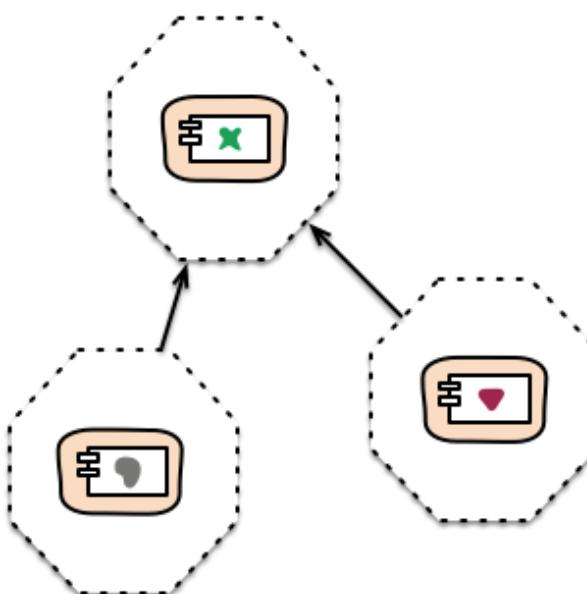
6.2 MSA 특징(3/9) – 비즈니스 역량 기반 팀 2

- ✓ 마이크로서비스 접근 방법은 기술이 아니라 비즈니스 역량을 중심으로 서비스를 나눕니다.
- ✓ 따라서 팀은 cross-functional 특성을 가져야 하며, UI, DB, 프로젝트 관리 등 필요한 모든 기술을 갖추어야 합니다.
- ✓ Cross-functional 팀은 각 제품을 개발하고 운영할 책임을 가지며, 각 제품은 메시지 버스를 이용하여 통신하는 여러 개의 서비스로 구성되어 있습니다. ← www.comparethemarket.com
- ✓ amazon.com의 마이크로서비스 팀은 피자 두 판, 즉 12명을 넘지 않도록 했고, 6명 팀은 보다 작은 업무를 할당했습니다.



Cross-functional teams...

<http://martinfowler.com/articles/microservices.html>



... organised around capabilities
Because Conway's Law

6.2 MSA 특징(4/9) – 프로젝트가 아니라 제품

- ✓ 우리가 볼 수 있는 대부분의 애플리케이션 개발은 프로젝트 모델을 따르고 있습니다.
- ✓ 마이크로서비스 팀은 제품의 전체 라이프사이클을 책임집니다. ← “you build, you run it”, amazon.com
- ✓ 소프트웨어를 완성되어야 할 기능을 세트로 보는 것이 아니라 제품, 즉 **비즈니스 역량으로의 연결 고리**로 간주합니다.
- ✓ 서비스 입도가 작을 수록 서비스 개발자와 서비스 담당자 간의 소통은 훨씬 수월합니다.

6.2 MSA 특징(5/9) – 똑똑한 끝지점 단순한 파이프

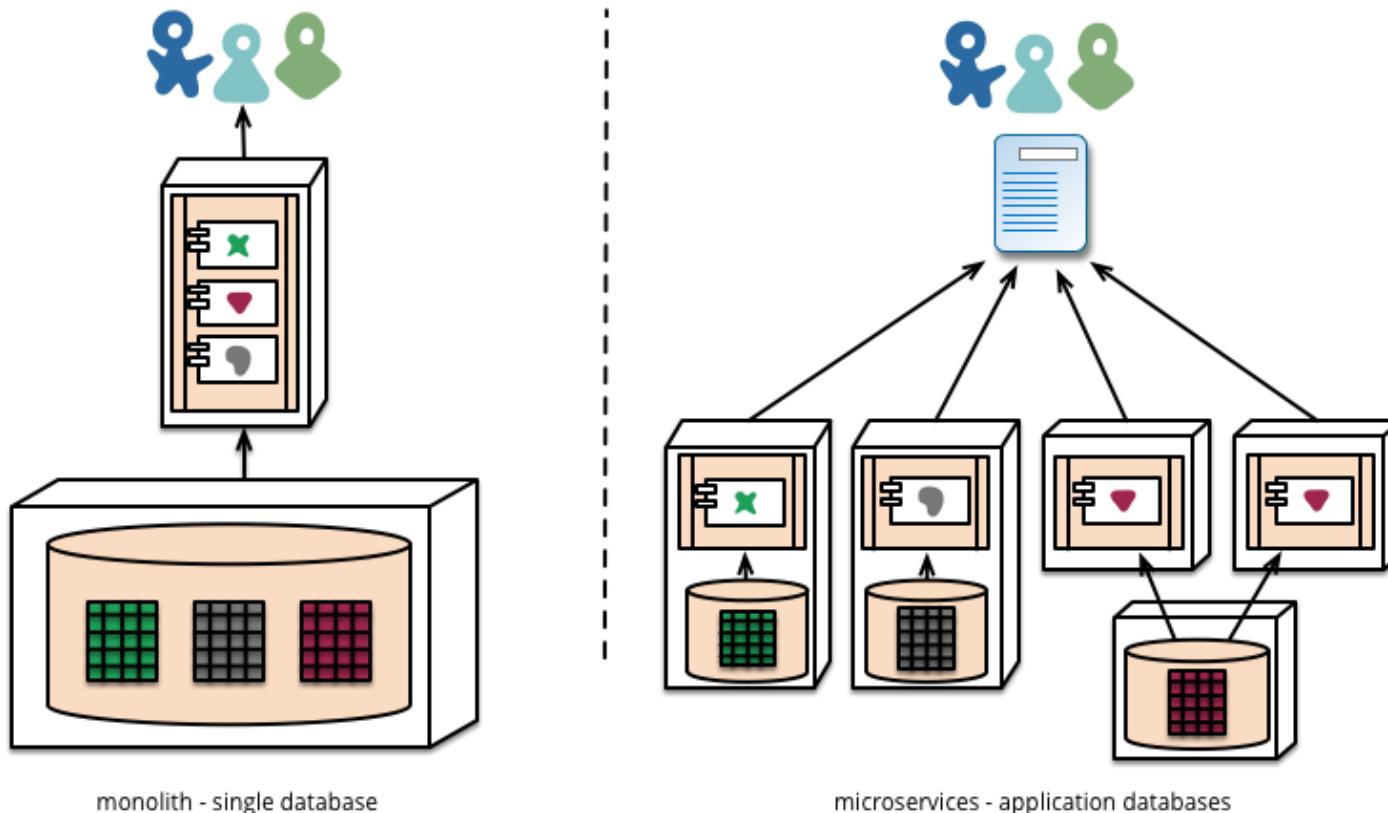
- ✓ 서로 다른 프로세스 간에 통신을 위해서 ESB 등과 같은 똑똑한 제품을 사용해 왔습니다. ← 래우팅, 변환, 비즈니스, 조율
- ✓ 마이크로서비스는 커뮤니티는 “smart endpoints and dumb pipes” 방식을 선호합니다.
- ✓ 도메인 로직은 서비스 속에 high cohesion 되어야 하고, 연결은 호출에서 loose coupling되어야 합니다.
- ✓ 따라서 REST와 같은 단순한 도구를 WebService, BPEL과 같은 복잡한 도구보다 선호합니다.
- ✓ HTTP 요청/응답을 자원API와 함께 사용하거나 경량 메시지 버스(Rabbit MQ, ZeroMQ)를 주로 사용합니다.
- ✓ 메모리 안에서 사용하던 고운 입자(fine-grained) 인터페이스를 거친 입자(coarse-grained)로 변경하여 사용합니다.

6.2 MSA 특징(6/9) – 분권 거버넌스

- ✓ “build it, run it”을 위해 필요한 모든 것을 팀이 알아서 결정한다라는 amazon.com의 정신은 유명합니다.
- ✓ 개발 언어, 개발 도구, 프레임워크, 개발 방법론 등 어는 것이든 팀에서 찾아서 활용하고 필요하면 공유합니다.
- ✓ 지금 당장 꼭 필요한 것만 개발하며, 불필요한 표준이나 문서 등으로 개발에 방해를 받아서는 안됩니다.

6.2 MSA 특징(7/9) – 분권 데이터 관리

- ✓ 단일 데이터베이스를 유지하는 방식은 벤더의 라이센스 모델과 데이터베이스의 기능 확장에 그 뿌리를 두고 있습니다.
- ✓ 마이크로서비스는 “Polyglot Persistence” 접근방법을 선택하며, 서비스 별로 데이터베이스를 갖도록 설계합니다.
- ✓ 데이터 일관성(consistency)을 유지하기 위해 두 서비스 간의 **트랜잭션이 아닌 협업을 강조합니다.**
- ✓ 데이터 일관성 목표가 좀 늦게 달성하더라도 원래대로 돌려놓는 시나리오가 비즈니스 개념에 적합할 수 있습니다.



6.2 MSA 특징(8/9) – 실패를 위한 설계

- ✓ 서비스를 컴포넌트로 사용한다는 의미는 컴포넌트는 실패할 수 있으므로 서비스 실패에 견디도록 설계함을 의미합니다.
- ✓ 서비스는 언제든 실패할 수 있으며, 실패해서 더 이상 진행할 수 없을 때, 자연스럽게 대응할 수 있도록 설계해야 합니다.
- ✓ 제품화 단계에 다양한 실패에 대비하여 자동으로 테스트할 수 있는 환경을 마련하여야 합니다. ← Netflix
- ✓ 기술 요소와 비즈니스 내용에 대한 실시간 모니터링 체계를 갖추어야 합니다. ← 인터페이스 상태와 주문 상태
- ✓ 이러한 설계는 애플리케이션이 긴급 상황에 빠르고 유연하게 대응할 수 있도록 합니다.

6.2 MSA 특징(9/9) – 진화하는 설계

- ✓ 시스템을 서비스로 분해할 때 기준은 “독립적으로 대체하거나 업그레이드 할 수 있어야”입니다.
- ✓ 모노리스로부터 시작해서 마이크로서비스를 지속적으로 추가하는 경우도 있습니다.
- ✓ 컴포넌트를 마이크로서비스화 함으로써, 보다 작은 릴리스 계획을 잡을 수 있습니다.
- ✓ 서비스 소비자는 작은 변화에 대응할 수 있도록 설계함으로써 버전 폭증 문제를 해결할 수 있습니다.



7. 마이크로서비스와 조직

7.1 기능 조직과 서비스 조직

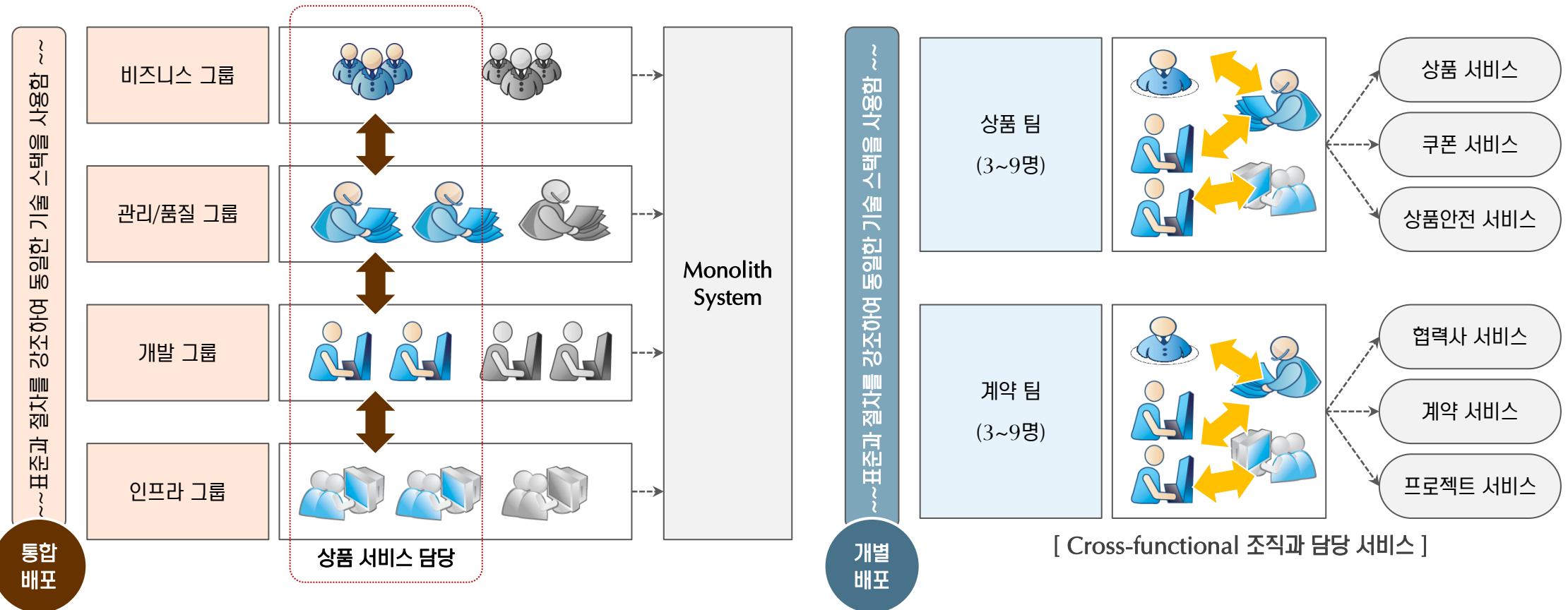
7.2 서비스 조직의 강점

7.3 조직과 설계

7.4 업무와 기술 분리

7.1 기능 조직과 서비스 조직

- ✓ 기능 중심의 팀이 겪는 소통(communication)의 문제를 극복하고, 시장에 능동적으로 대응하려면 Cross-functional, 특성과 Self-organization 특성을 갖춘 조직의 형태가 필요합니다.
- ✓ 업계의 애자일 경험을 통해, 도메인 중심의 소규모 팀(3~9명)인 스크럼 팀이 매우 효율적임을 알았습니다.
- ✓ 스크럼 스타일 팀은 기본 기술에 대해서만 표준을 따를 뿐, 마이크로서비스 내부 기술과 구조는 팀이 결정합니다.



7.2 서비스 조직의 강점

- ✓ 도메인 중심의 마이크로서비스 개발팀의 세 가지 수준에서 독립성을 갖고 있습니다.
- ✓ 먼저, 독립적인 제품화 일정을 갖고 가므로, 다른 팀과 배포 일정을 맞추기 위해 조율할 이유가 없습니다.
- ✓ 다음으로 독립적인 기술 선택과 구조를 갖추고 있으므로, 팀의 역량을 최고로 발휘할 수 있는 기술 구조를 선택하면 됩니다.
- ✓ 각 팀은 해당 도메인의 언어(ubiquitous language)로 요건을 정의하고 모델링 하고 구현하면 됩니다.



7.3 조직과 설계 [1/2]

- ✓ Conway의 법칙은 컴퓨터 프로그래머인 Melvin Conway의 1967년 아이디어에서 나왔습니다.
- ✓ 1968년 “National Symposium on Modular Programming”에서 발표하였습니다.
- ✓ 소프트웨어 개발 관점에서는 어떤 한계(limitation)나 제약조건(constraint)에 대해 이야기 합니다.
- ✓ 이 주장은 소프트웨어 뿐만 아니라 모든 유형의 시스템에 해당됩니다.

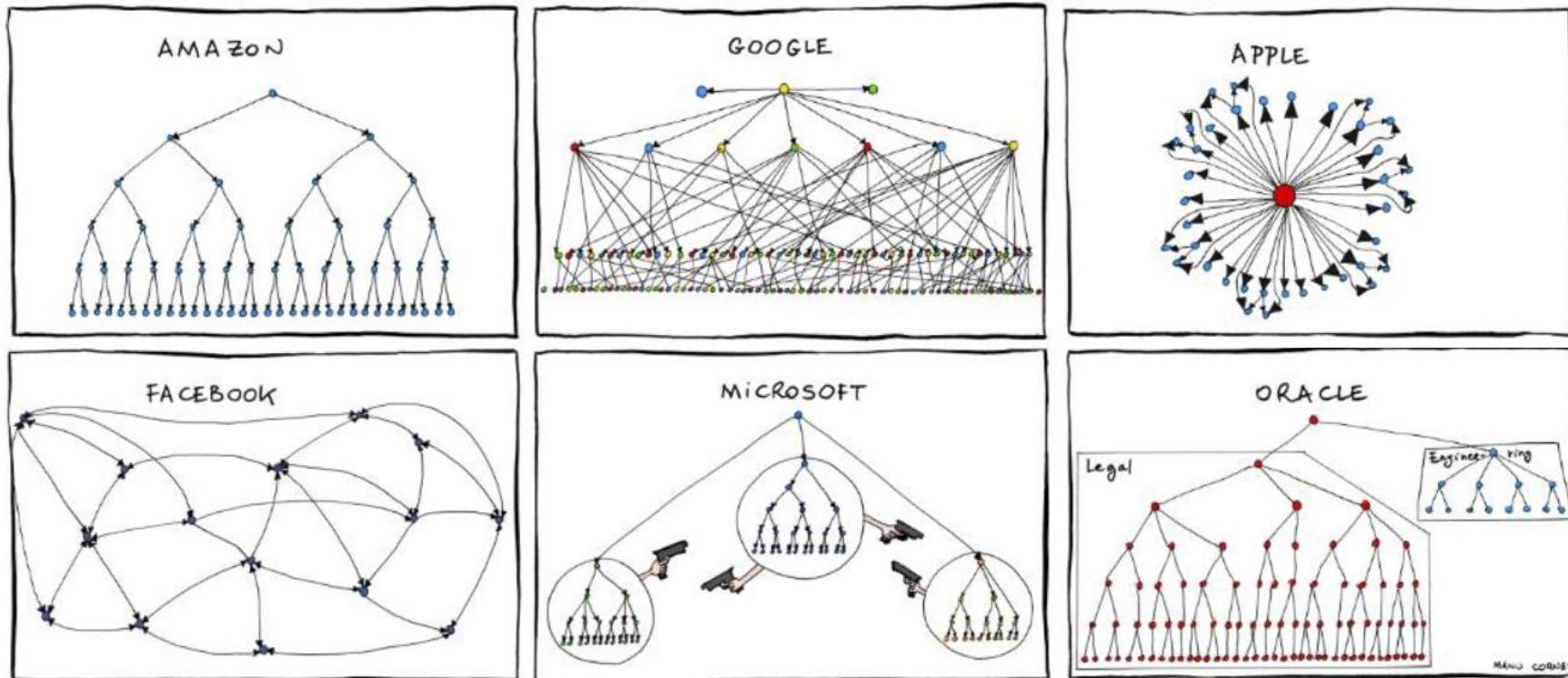
시스템을 설계하는 조직은 어떤 조직이든 그 조직의 소통 구조를 닮은 구조를 가진 시스템으로 설계할 것이다.

Any organization, that design a system (defined broadly), will inevitably produce a design whose structure is copy of the organization's communication structures.



7.3 조직과 설계 [2/2]

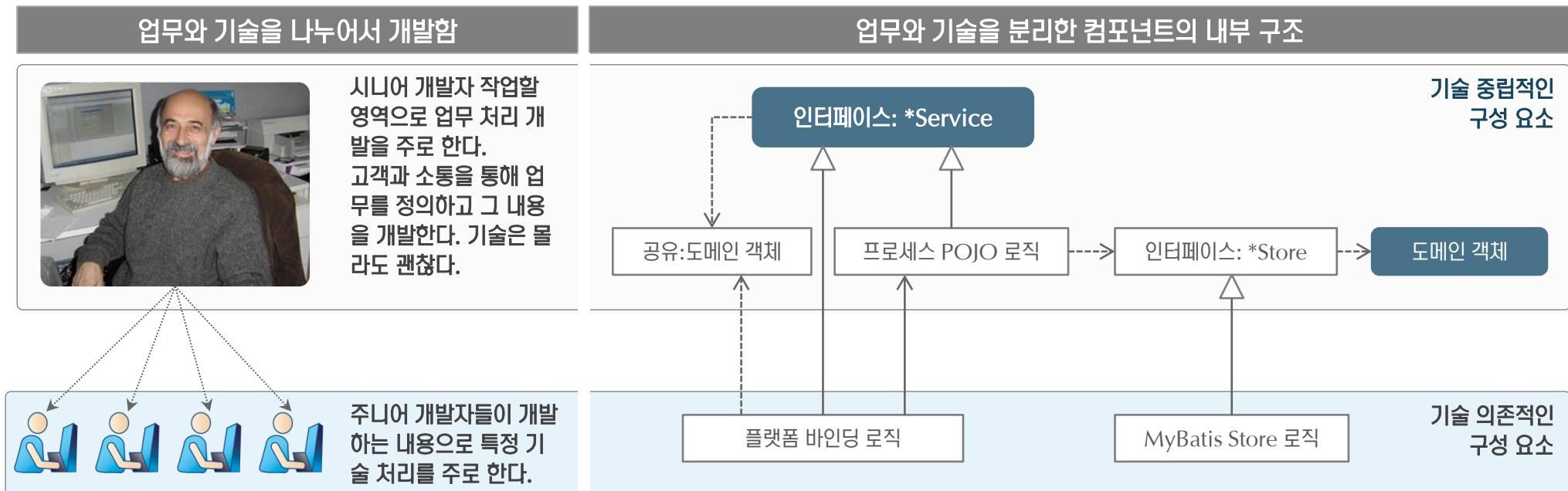
- ✓ 아래 삽화는 “Bonkers world”에 게시된 것으로 조직의 의사소통 구조를 잘 보여 줍니다.
- ✓ 우리가 잘 알고 있는 몇몇 기업은 고개가 끄덕여 집니다.



[출처] <http://www.bonkersworld.net/organizational-charts/>

7.4 업무와 기술 분리 (1/2)

- ✓ 컴포넌트 설계를 통해서 개발자의 역할 배정을 효율적으로 할 수 있습니다. ← 멋지고 경제적인 설계 아이디어
- ✓ 컴포넌트를 업무를 다루는 모듈과 기술에 종속적인 모듈로 나눈다면, 의외의 아주 큰 소득(예산/인력)이 있습니다.
- ✓ 시니어 개발자는 새로운 기술에 약해서 개발에 어려움을 겪고, 주니어 개발자는 업무에 약해서 개발에 어려움을 겪습니다.
- ✓ 결국 시니어 개발자는 개발에서 손을 떼고, 업무를 잘 모르는 주니어 개발자가 개발을 주도합니다. ← 품질 문제 발생



정확한 업무 이해를 바탕으로 하는 개발과 세밀한 기술을 바탕으로 하는 개발을 나누어서 진행할 수 있습니다.

업무와 기술 두 가지 모두에서 비약적인 품질 향상을 얻을 수 있고, 엔지니어를 활용도를 높일 수 있습니다.

7.4 업무와 기술 분리 (2/2)

- ✓ 제대로 설계한 컴포넌트는 도메인 모듈과 기술 종속적인 모듈을 잘 분리하여 서로 느슨한 관계를 갖도록 합니다.
- ✓ 기술 종속 모듈은 해당 기술이 변경될 경우, 큰 어려움 없이 교체할 수 있도록 설계합니다.
- ✓ 도메인 모듈은 프로그래밍 언어만을 사용하여 업무 로직과 엔티티 객체를 구현합니다. 시니어 개발자의 뜻입니다.
- ✓ 데이터 접근, 외부 서비스 호출, 그리고 컴포넌트 컨테이너 바인딩 등은 다른 개발자가 개발하여도 문제 없도록 설계합니다.



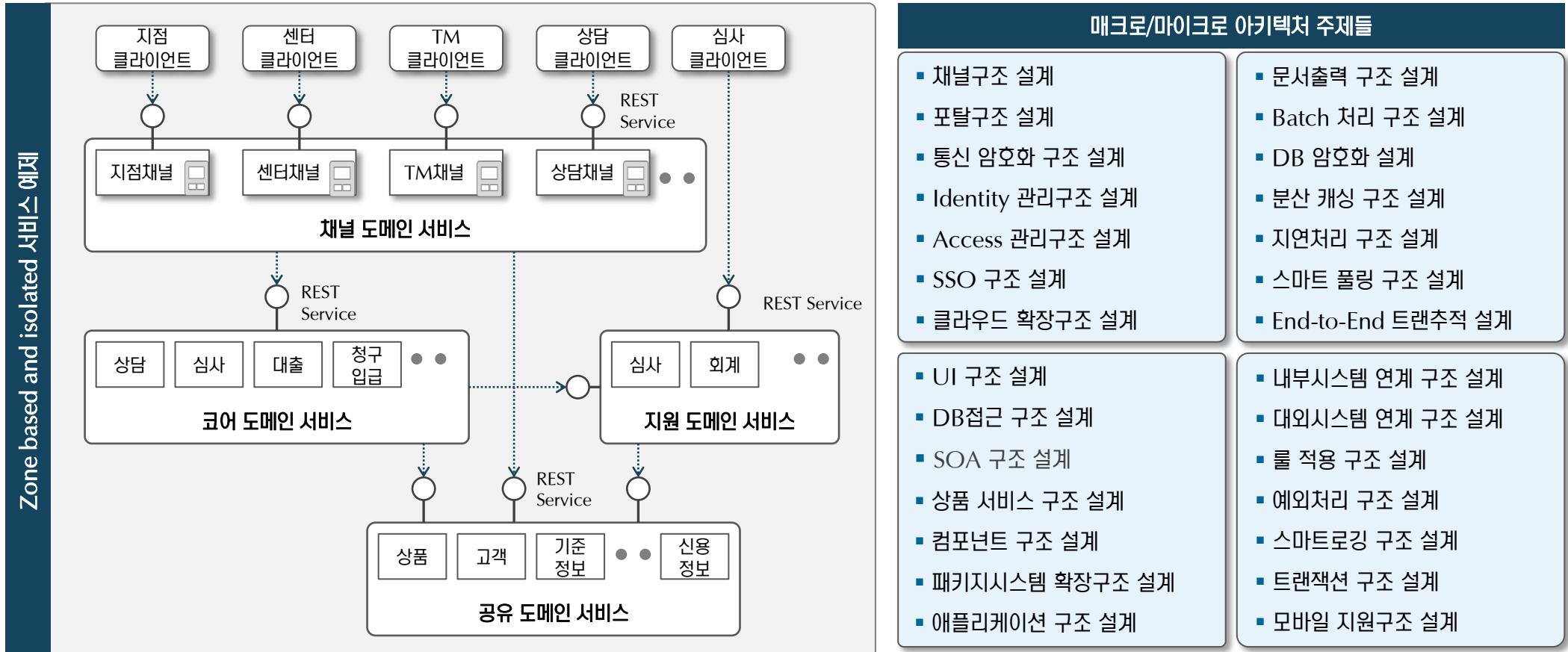


8. 마이크로서비스와 아키텍처

-
- 6.1 Microservice 개요
 - 6.2 Microservice 특징
 - 6.3 기업 시스템과 Microservice

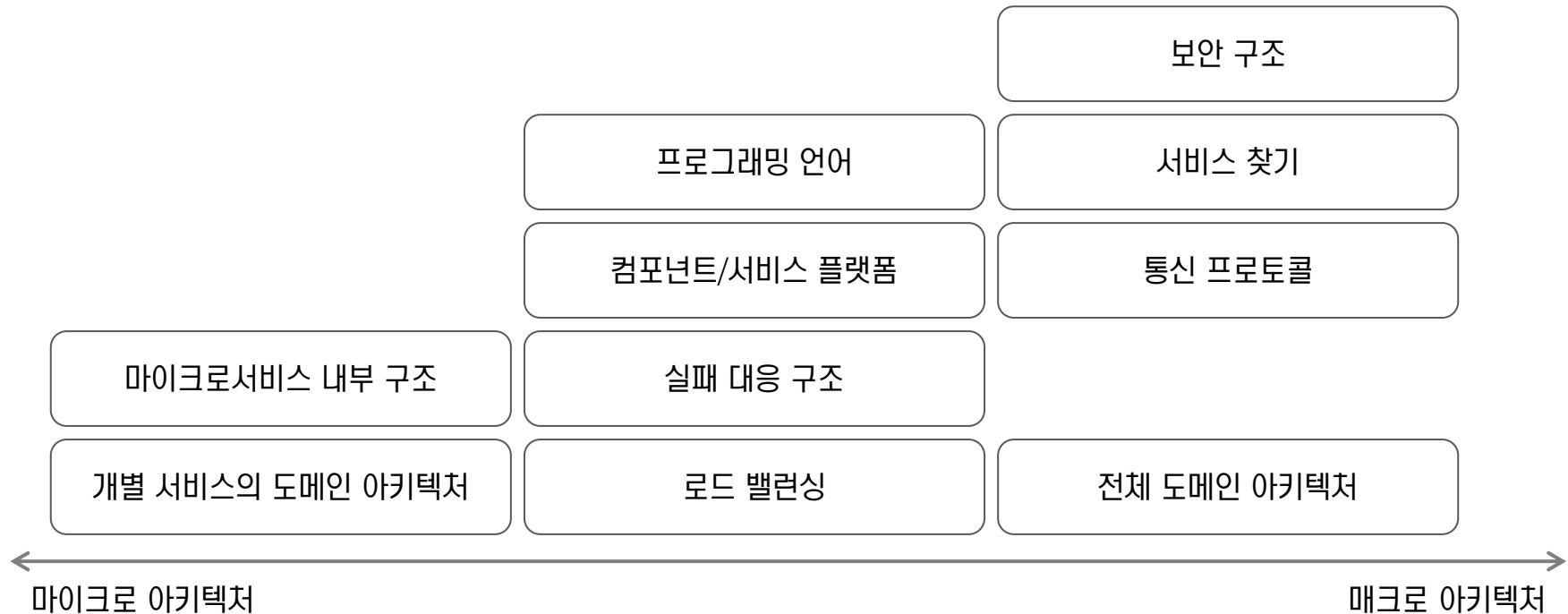
8.1 매크로 아키텍처

- ✓ 대규모 시스템에서 아키텍처 설계 주제는 매우 다양합니다.
- ✓ 어떤 주제는 전체를 가로지르는 것이고, 어떤 주제는 특정 영역을 위한 설계입니다.
- ✓ MSA 접근방법에서는 마이크로서비스 자체와 매크로 서비스 주제로 나눌 수 있습니다.



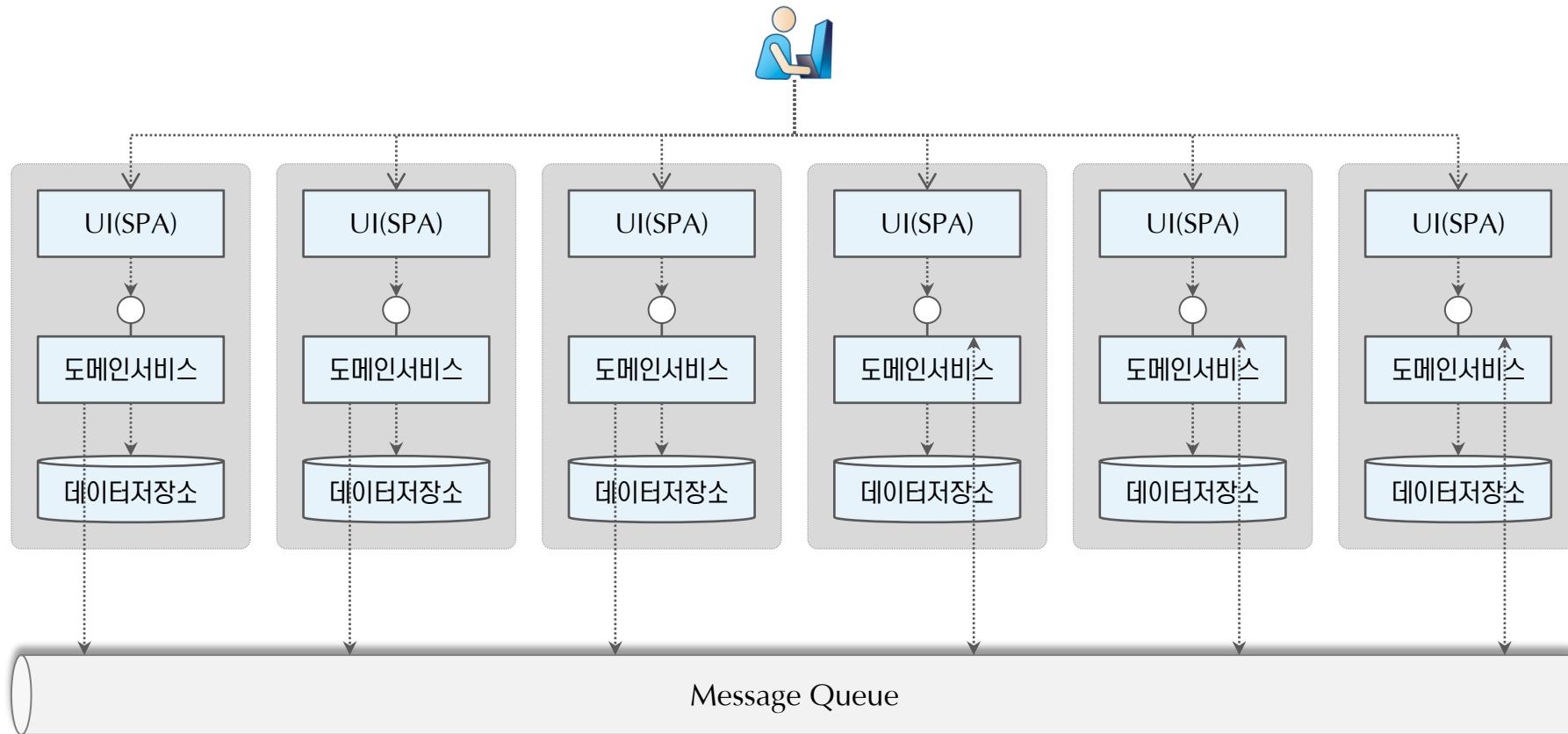
8.2 매크로와 마이크로 영역

- ✓ 여러 아키텍처 주제를 마이크로와 매크로 영역으로 나눌 수 있습니다.
- ✓ 매크로 아키텍처는 조직의 아키텍처 팀이 주도합니다.
- ✓ 마이크로 아키텍처는 서비스 개발팀 내에서 설계하고 구현합니다.



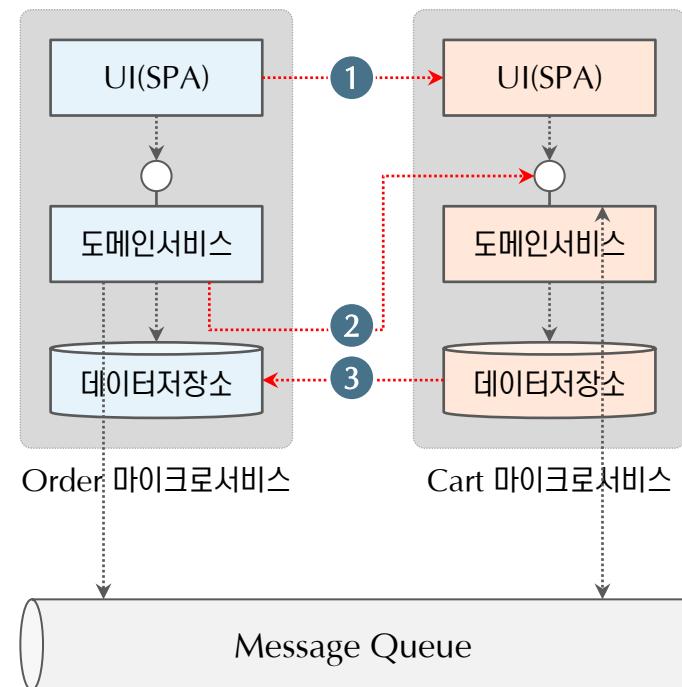
8.3 마이크로서비스 아키텍처

- ✓ 마이크로서비스는 UI, 도메인, 데이터를 모두 갖춘 아주 작은 애플리케이션입니다.
- ✓ 마이크로서비스 간의 연결은 서비스 호출이나, 메시지 전방 방식을 선택합니다.
- ✓ 마이크로서비스 UI는 주로 SPA(Single Page App) 스타일로 구성합니다.



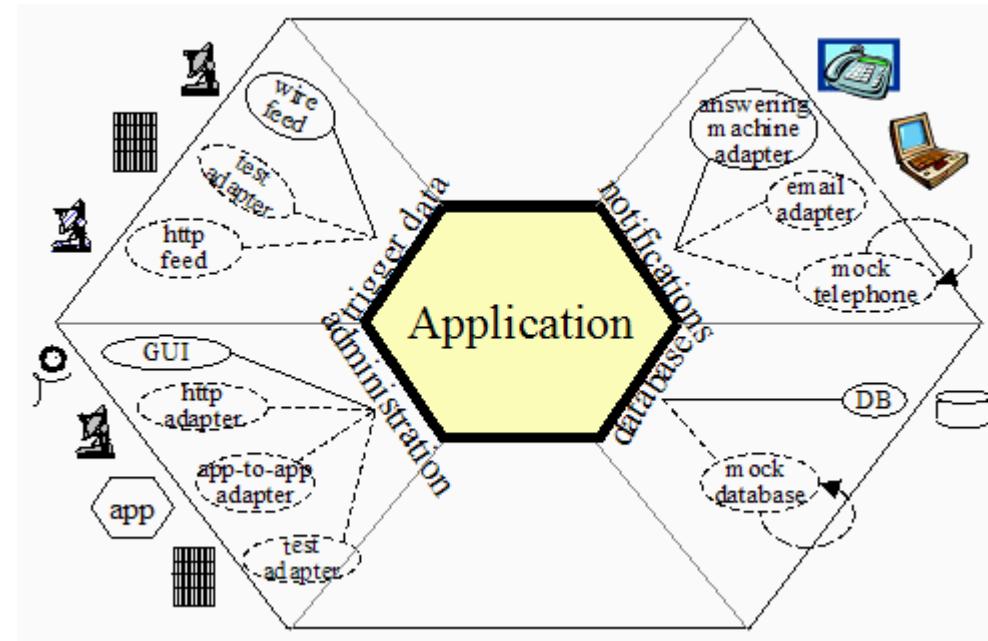
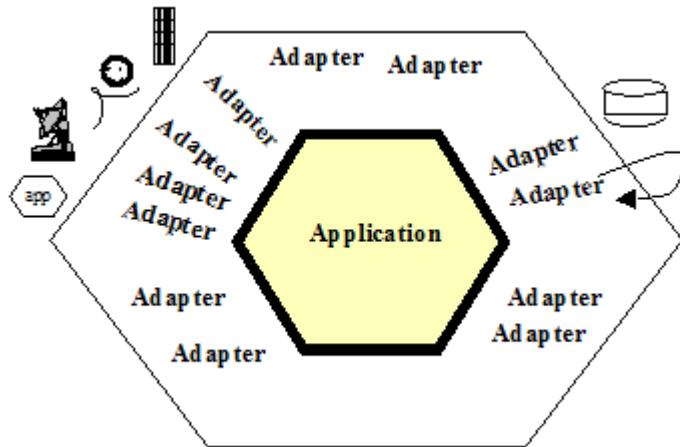
8.4 마이크로서비스 연결

- ✓ 마이크로서비스는 세 가지 수준에서 통합(integration)이 가능합니다.
- ✓ UI 수준의 통합은 다른 서비스의 UI에 URL 링크를 이용하여 연결하는 것입니다.
- ✓ 도메인 서비스 수준의 통합은 도메인 서비스가 외부 서비스를 자원으로보고 호출하는 방식으로 연결합니다.
- ✓ 데이터 수준의 통합은 다른 서비스의 데이터를 복제하여 사용하는 방식으로 연결합니다.



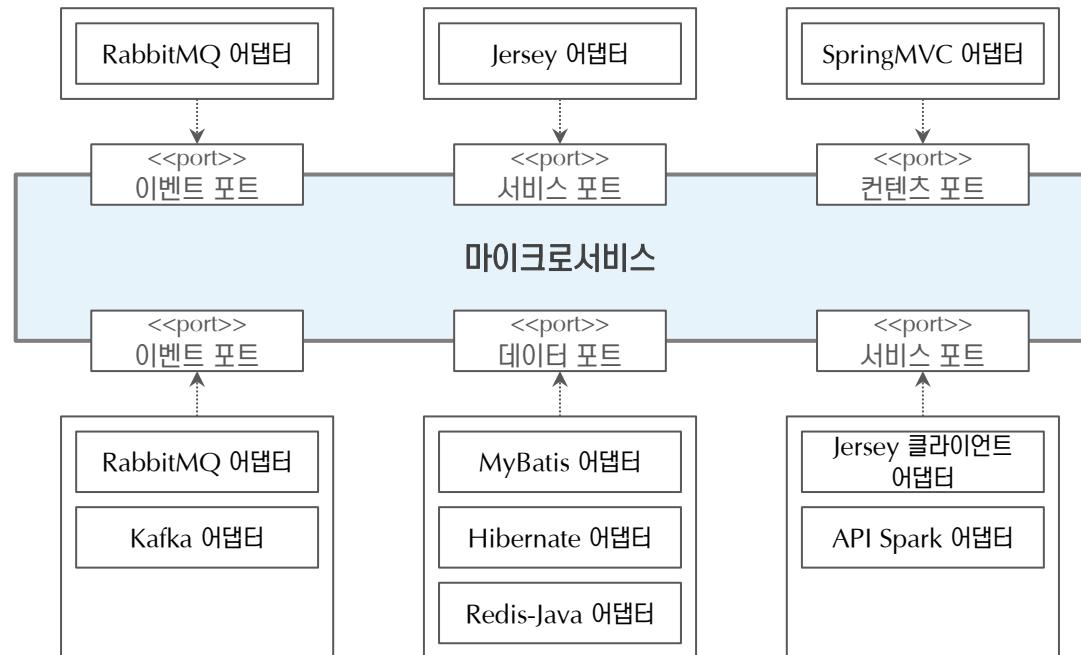
8.5 Hexagonal 아키텍처 (1/2)

- ✓ 마이크로서비스 설계에 Alistair Cockburn의 Hexagonal은 아주 훌륭한 참조자료입니다.
- ✓ 마이크로 서비스로 복잡한 기술 환경 속에서 독립성과 독자성을 갖추어야 합니다.
- ✓ 포트와 어댑터 개념을 활용하여 급변하는 환경에 적응하는 애플리케이션 개념을 제시합니다.



8.5 Hexagonal 아키텍처 (2/2) – 응용

- ✓ 마이크로서비스 내부 설계는 Hexagonal 아키텍처 패턴을 기반으로 합니다. → 사례 참조
- ✓ 빠르게 변화하는 환경 속에서 마이크로서비스가 능동적으로 대응하는 방법을 제시합니다.
- ✓ 포트와 어댑터라는 표현을 사용합니다. → eCommerce 사례 참조





9. 사례 연구

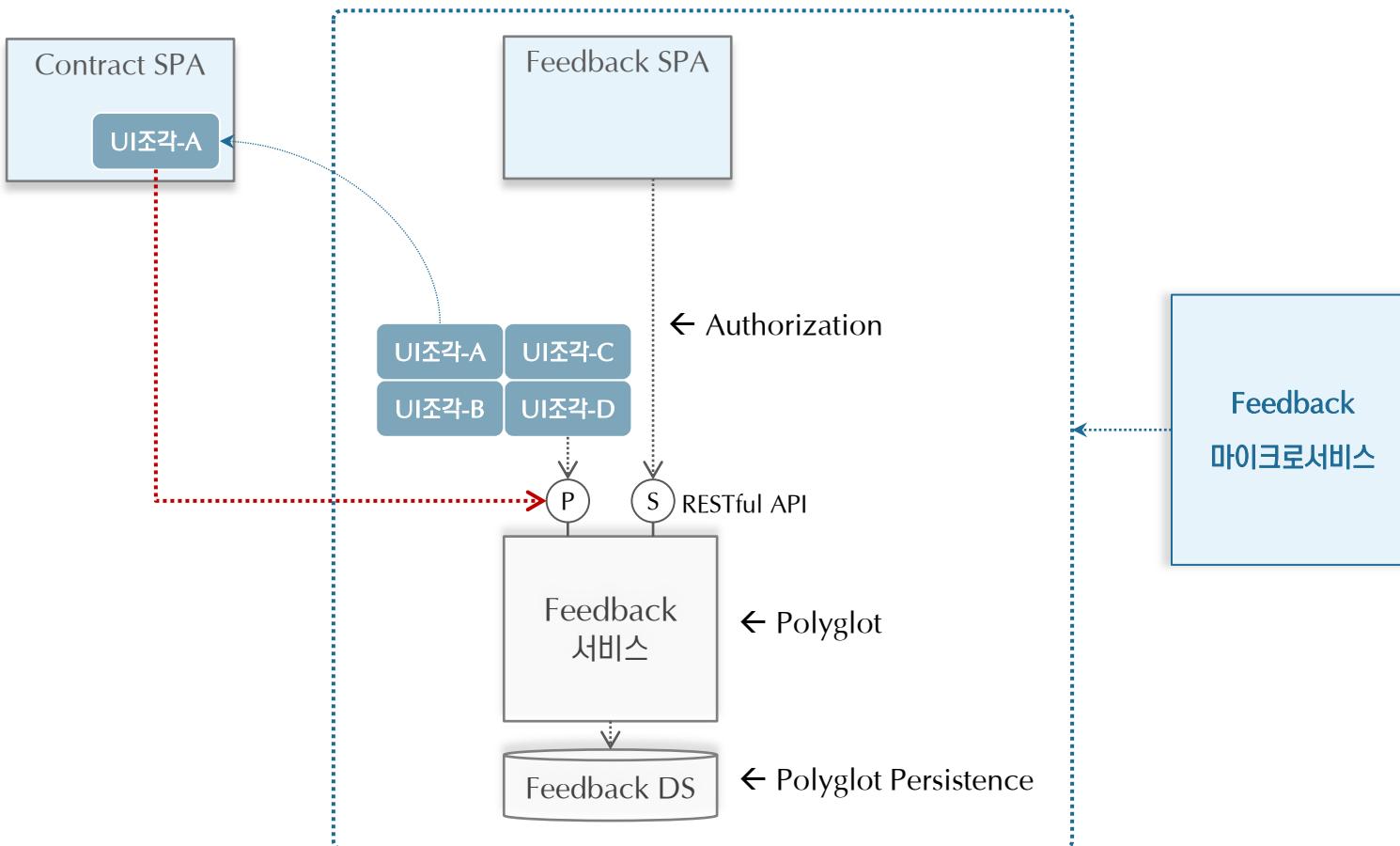
9.1 SFA 사례

9.2 gilt.com

9.3 Netflix.com

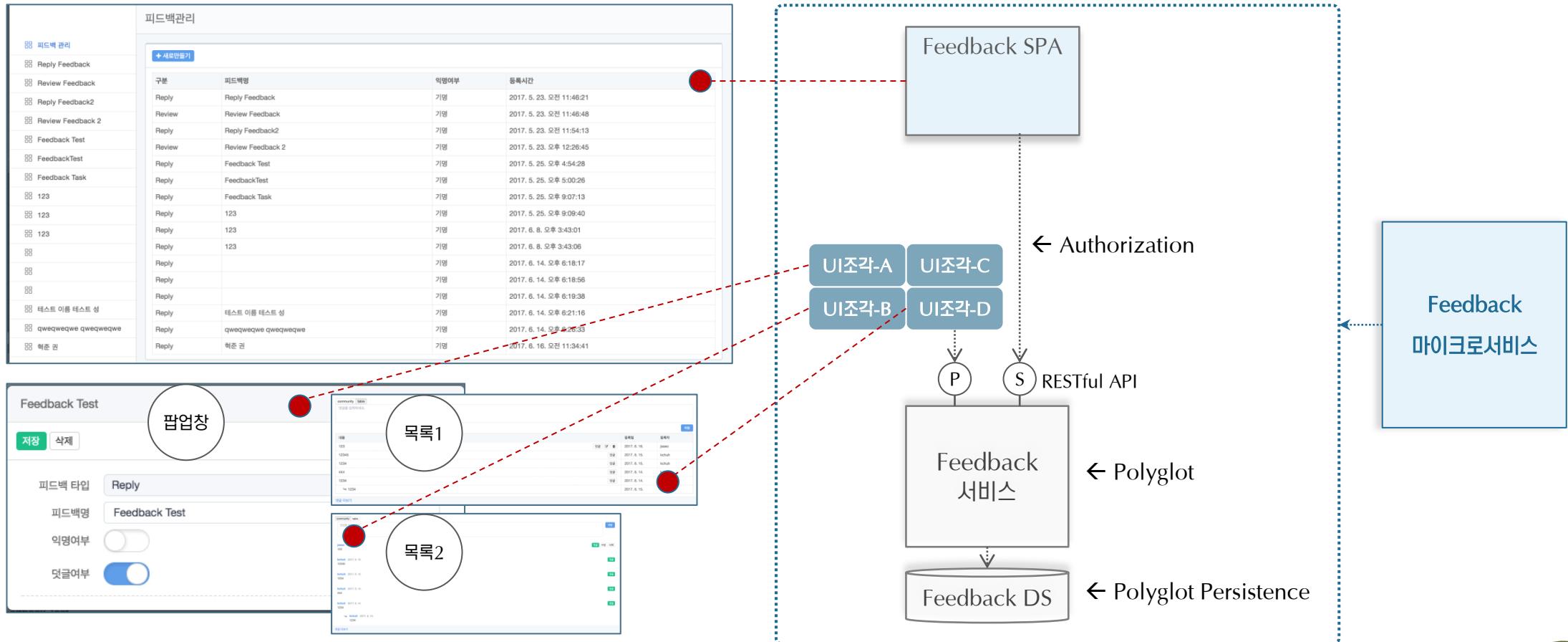
9.1 SFA 사례(1/8): 마이크로서비스 구조

- ✓ 마이크로서비스는 RESTful API를 제공하는 서비스와 SPA 스타일의 UI 클라이언트로 구성합니다.
- ✓ 다른 서비스가 Feedback 서비스를 SPA 영역에서 사용할 때 필요한 UI 조각(piece) 여러 개를 포함합니다
- ✓ UI 조각이 사용하는 인터페이스(*Provider)는 SPA에 사용하는 인터페이스(*Service)는 따로 식별하고 사용합니다.
- ✓ Feedback Data Store는 Feedback 서비스 전용 데이터 저장소여야 하며, 저장 방식에 제약조건은 없습니다.



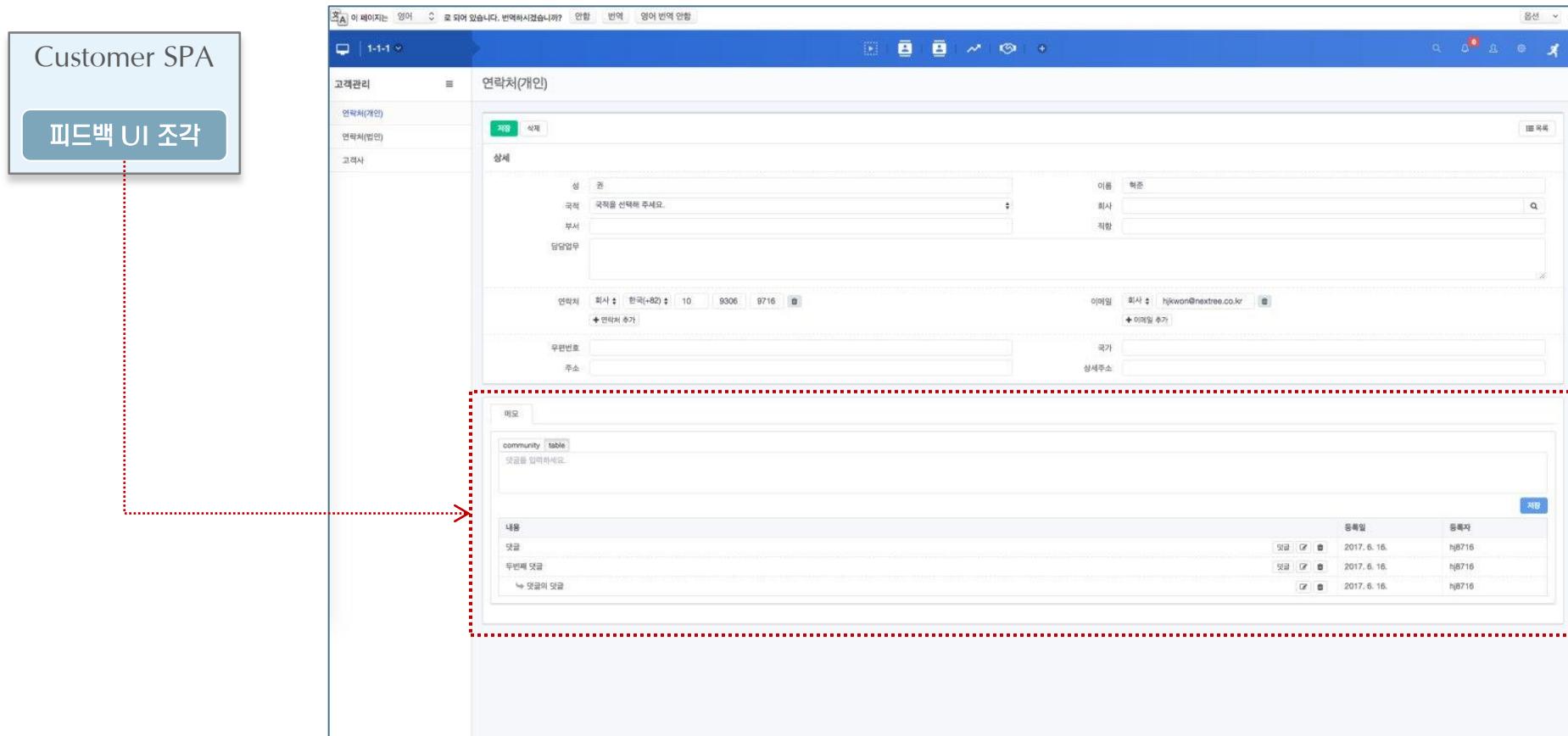
9.1 SFA 사례(2/8): 마이크로서비스 구조 – UI

- ✓ SPA UI는 Feedback 마이크로서비스의 기본 UI이며 주요 기능을 모두 포함하고 있습니다.
- ✓ UI 피스(piece)는 다른 서비스에서 사용할 UI이며, 팝업, 목록 등 다양한 조각을 갖추고 있습니다.
- ✓ SPA UI는 접근제어가 필요한 *Service 유형의 RESTful API를 호출합니다.
- ✓ UI 피스는 접근제어가 필요없는 *Provider 유형의 RESTful API를 호출합니다.



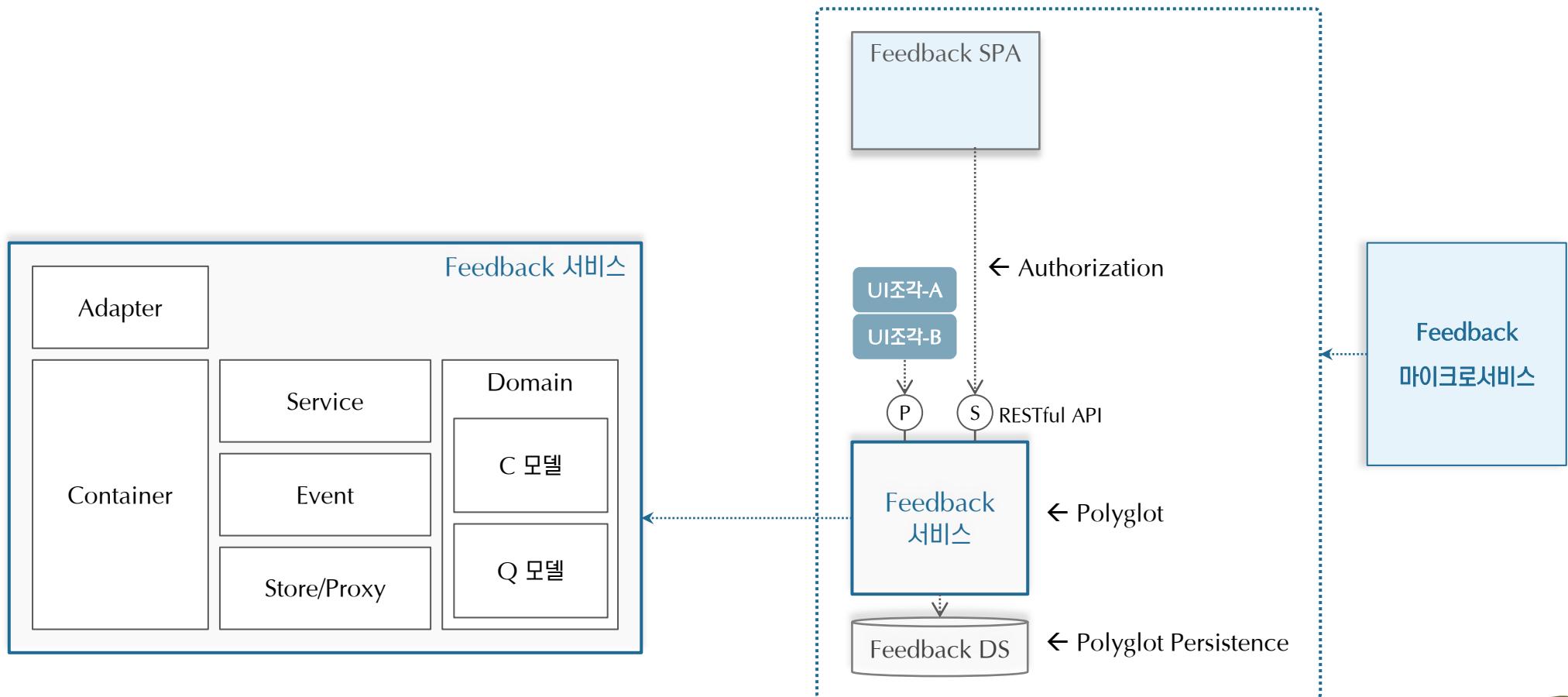
9.1 SFA 사례(3/8): 마이크로서비스 구조 – UI 조각

- ✓ Customer 마이크로서비스의 SPA는 고객 관리를 위한 기능을 제공하는 UI 화면입니다.
- ✓ 특정 고객에 대한 다양한 피드백을 보기 위해서 또는 Feedback 기능 추가를 위해서 Feedback 서비스를 조합합니다.
- ✓ Feedback 마이크로서비스가 제공하는 기능을 결합하기 위해 피드백 UI 조각 중에 필요한 것을 선택합니다.
- ✓ UI 조각은 팝업, 리스트, 디테일, 조회 등 단위 기능을 제공합니다.



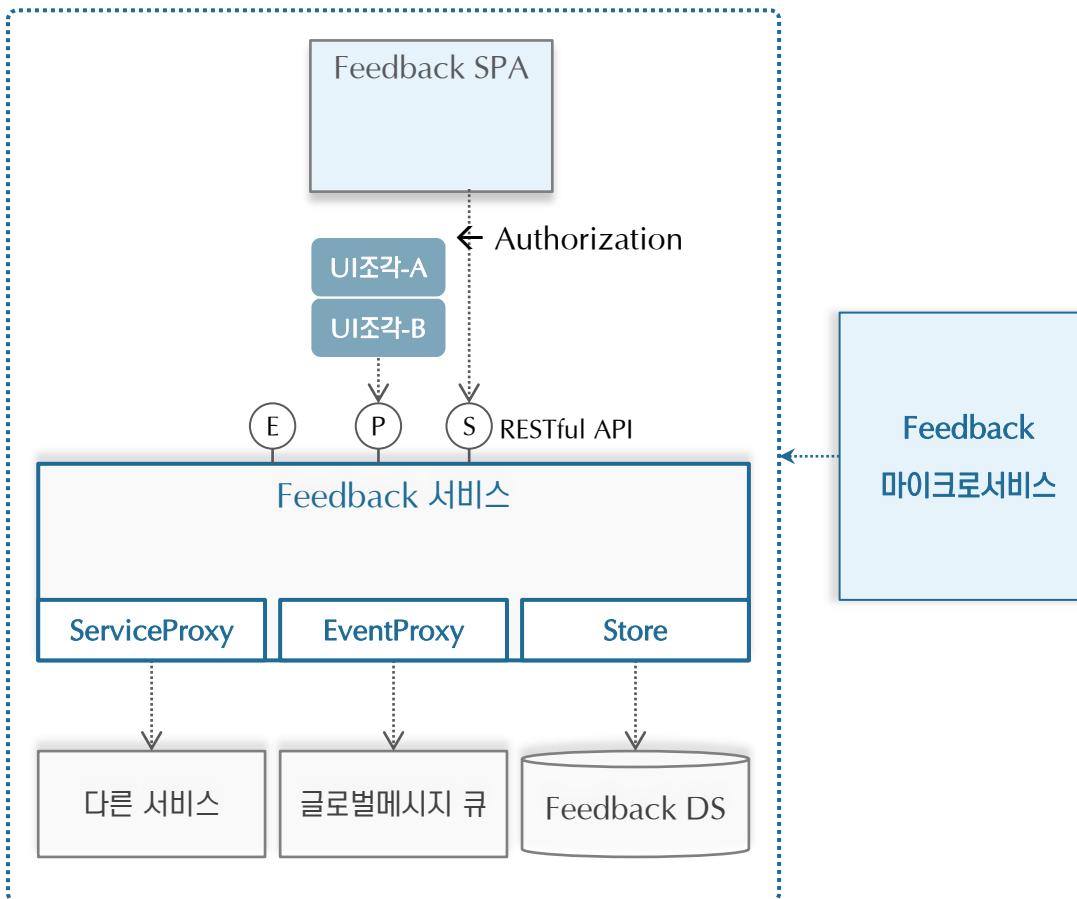
9.1 SFA 사례(4/8): 마이크로서비스 구조 – 서비스

- ✓ Feedback RESTful API를 제공하는 Feedback 서비스는 컴포넌트 기반 구조를 갖출 것을 권합니다.
- ✓ 서비스를 발행하는 부분, 이벤트 처리 부분, 도메인 모델, 저장소 바인딩 부분, 컨테이너 부분 등으로 구성합니다.
- ✓ 각 부분은 독립된 프로젝트로써 서비스 안의 다른 부분 또는 다른 서비스의 특정 부분들과 의존성을 갖게 됩니다.
- ✓ Adapter는 Feedback 서비스를 사용하는 서비스 클라이언트에게 제공하여, REST API 호출을 쉽게 합니다.



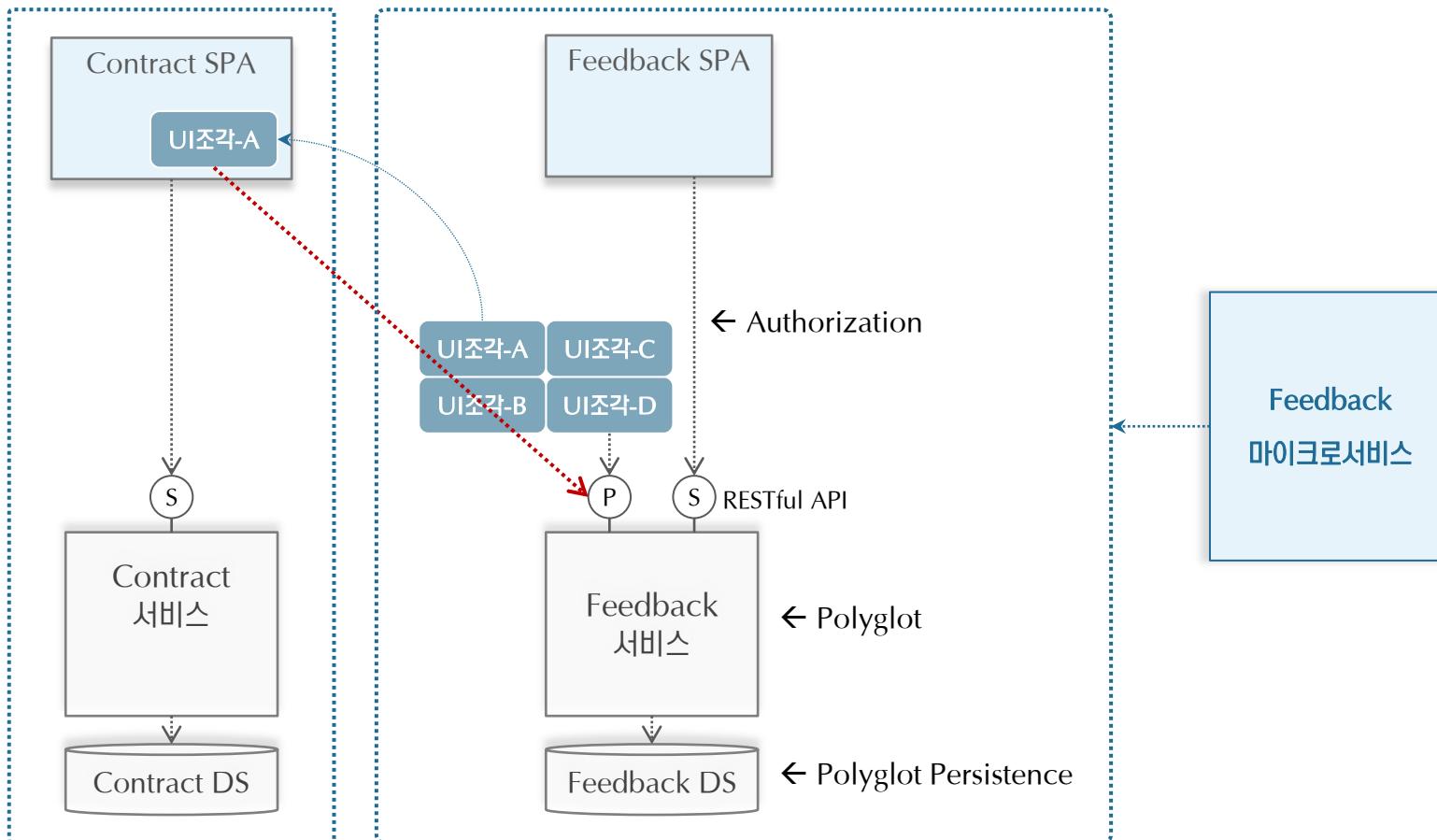
9.1 SFA 사례(5/8): 마이크로서비스 구조 – 자원

- ✓ 마이크로서비스가 사용하는 자원은 세 가지로 요약할 수 있습니다.
- ✓ 외부 서비스 자원을 호출하는 통로인 ServiceProxy는 다양한 서비스 접근을 지원합니다.
- ✓ EventProxy는 [Global] 이벤트를 발행하는 창구로 사용하며, 이벤트 리스닝은 API를 이용합니다.
- ✓ Store는 데이터 저장소로 가는 창구이며, 다양한 데이터 바인딩으로써 도메인으로부터 저장소를 분리하여 줍니다.



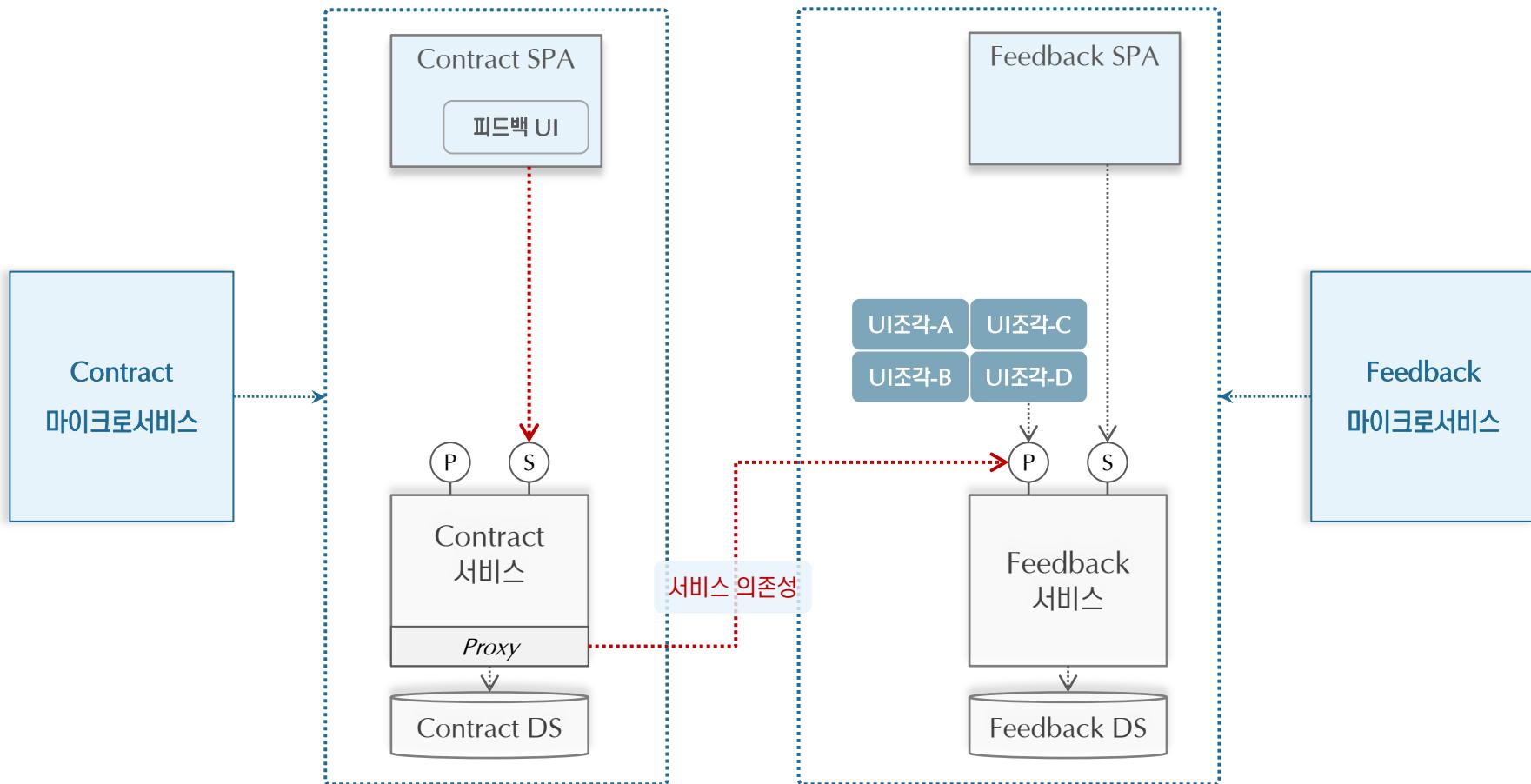
9.1 SFA 사례(6/8): 마이크로서비스 조합 (in SPA)

- ✓ Contract 서비스에서 Feedback 서비스의 특정 기능이 필요할 경우, UI 조각을 사용할 수 있습니다.
- ✓ 이 경우, Contact 서비스 API와는 상관없이 UI 수준에서 결합이 이루어집니다. ← Feedback ID만 갖고 있습니다.
- ✓ 즉, Contract SPA 내부의 Feedback UI 조각에서 직접 Feedback 서비스를 호출합니다.
- ✓ Provider API에는 호출한 서비스와 관련 사용자 정보를 건내줌으로써 Contract와 느슨한 결합을 유집합니다.



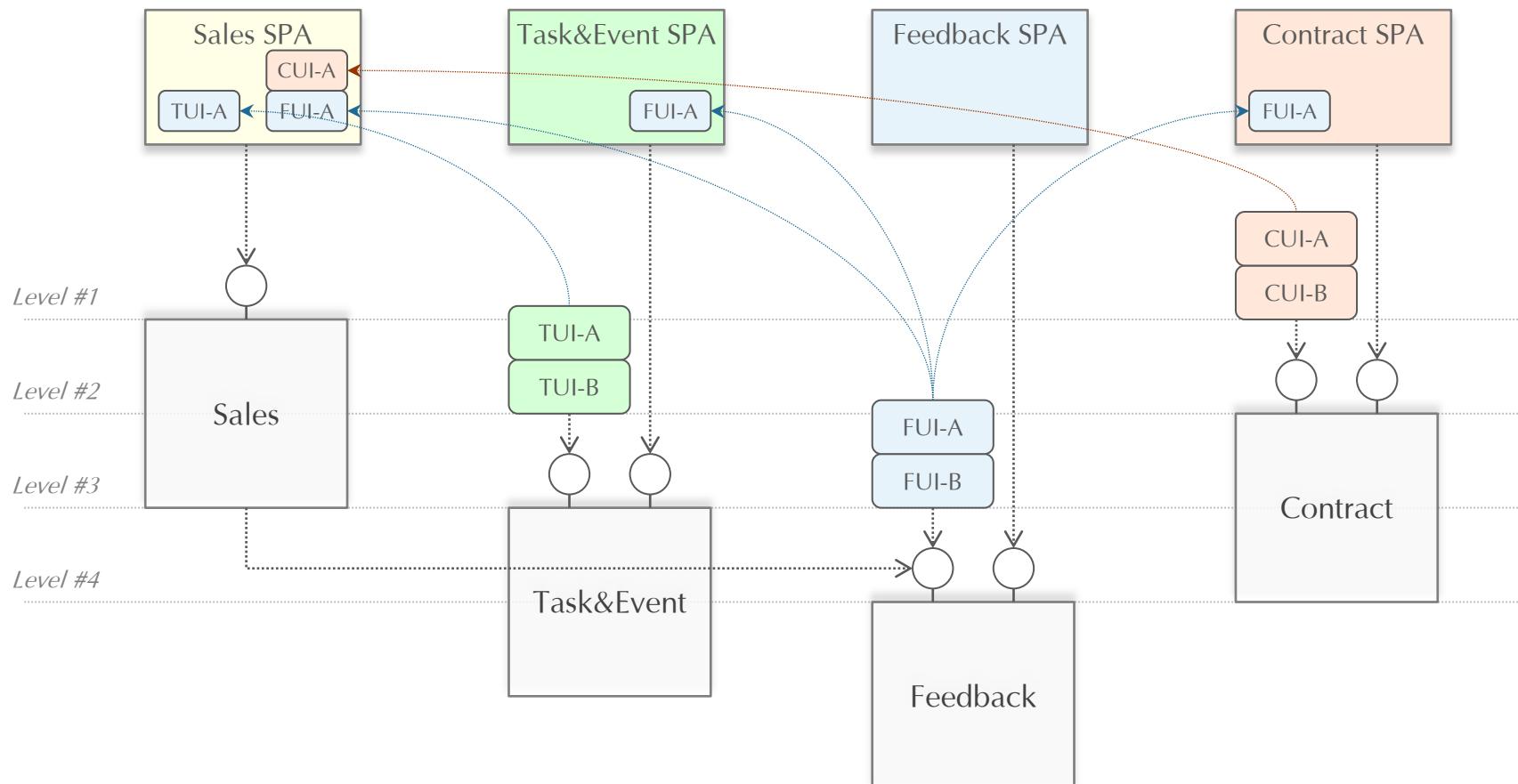
9.1 SFA 사례(7/8): 마이크로서비스 조합 (in API)

- ✓ 마이크로서비스는 자신 보다 아래 계층에 놓인 마이크로서비스를 사용할 수 있습니다.
- ✓ API 수준에서 다른 서비스를 조합할 수 있습니다. 이 경우, UI는 호출 서비스 스스로 구성한 UI입니다. ← UI조각 아님
- ✓ 아래 예제는 Contract 서비스가 Feedback 서비스를 사용하여, Contract SPA에서 보여주는 예제입니다.
- ✓ Contract는 Feedback 서비스 보다 위 계층에 위치하며, Contract 서비스는 proxy를 통해 Feedback에 접근합니다.



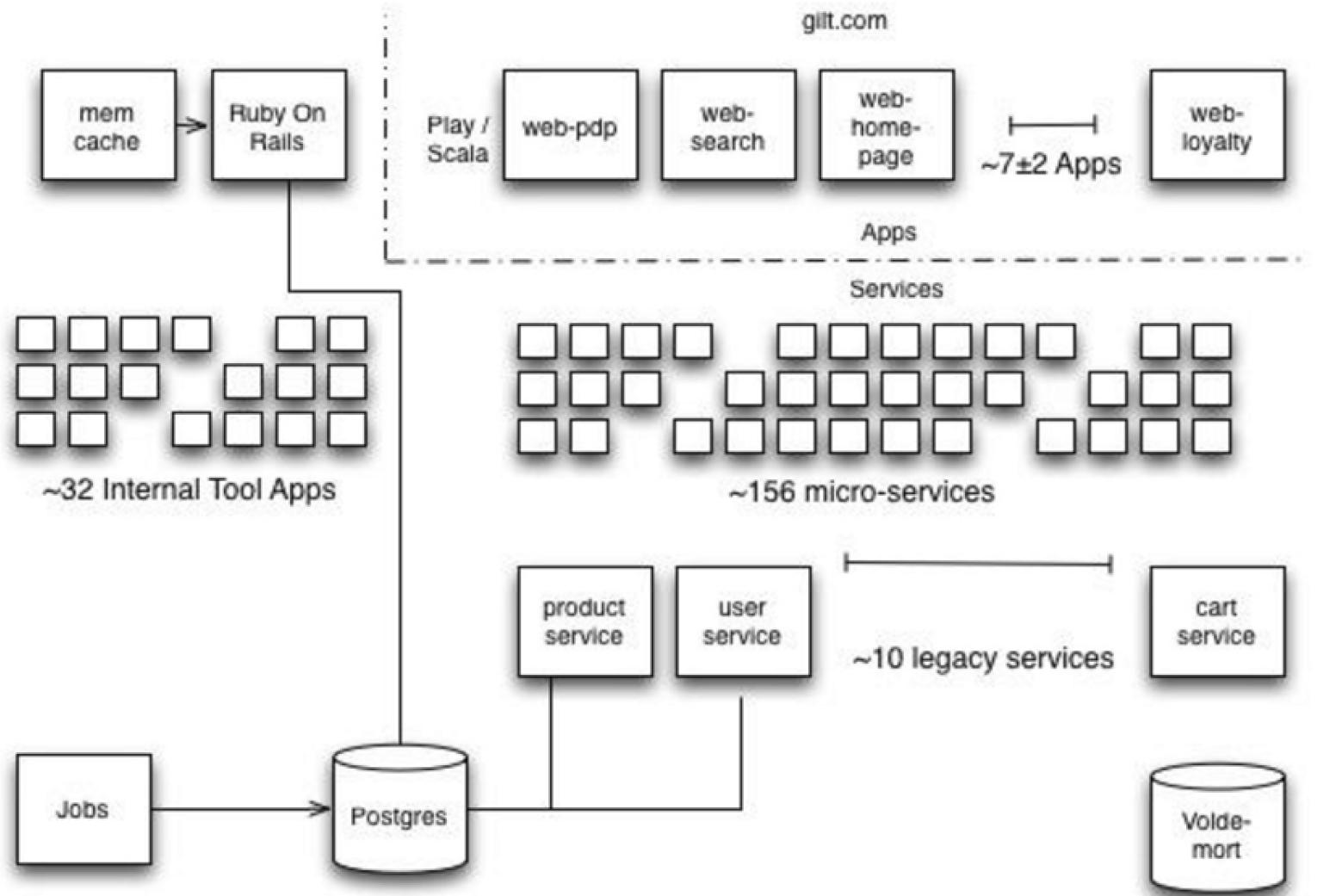
9.1 SFA 사례(8/8): 마이크로서비스 조합 개요

- ✓ 마이크로 서비스는 UI 수준 또는 API 수준에서 다른 서비스를 조합할 수 있습니다.
- ✓ UI 수준의 조합은 유일한 키 만을 사용하므로 가장 느슨한 결합 구조입니다.
- ✓ API 수준의 조합은 API 호출에 따른 의존성을 피할 수 없으며, UI 부분 중복 개발이 될 수 있습니다.
- ✓ 조합을 통한 서비스 제공은 활용 가능한 마이크로 서비스가 많을 수록 개발 속도가 빨라질 수 있습니다.



9.2 gilt.com

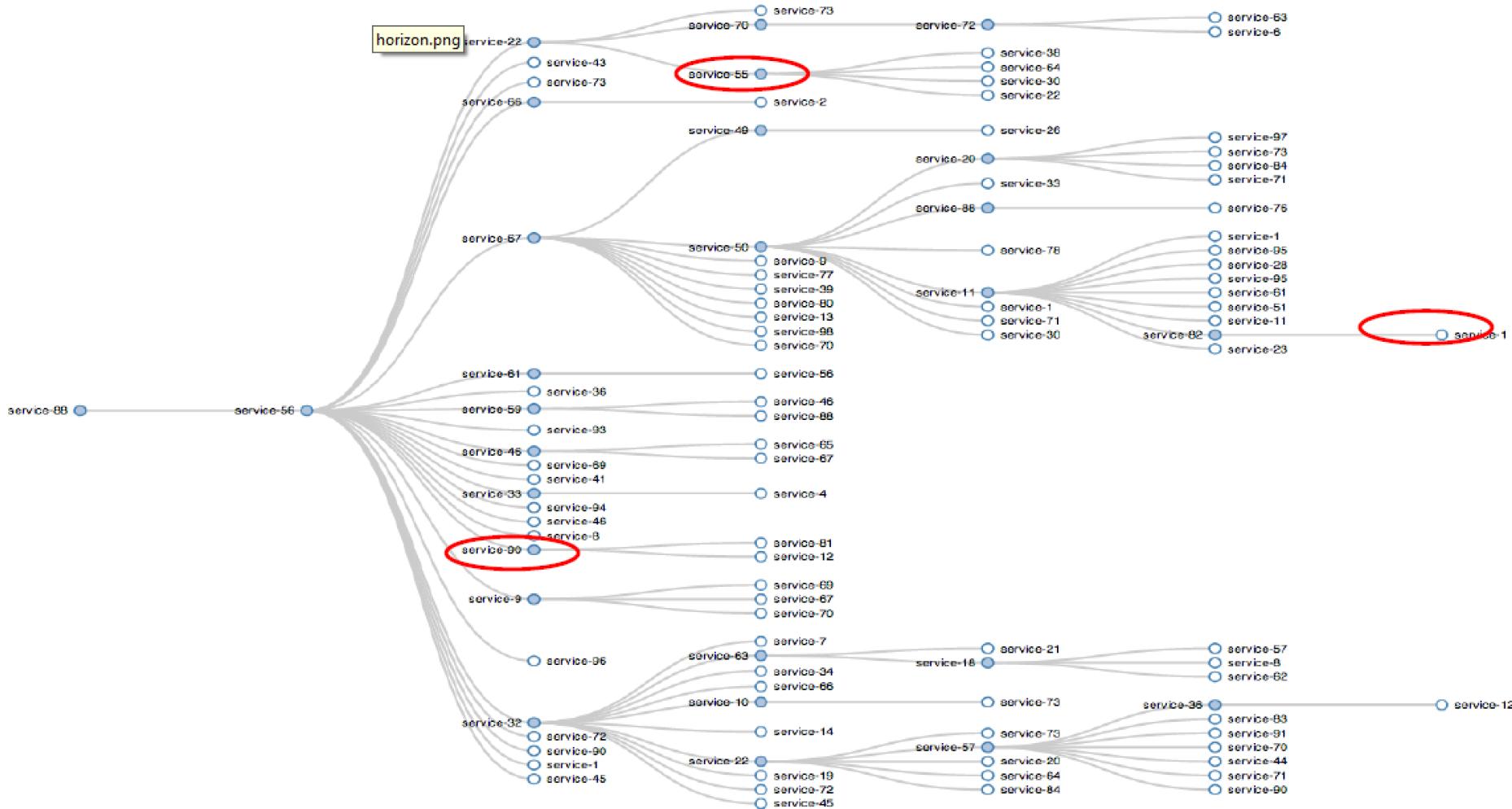
✓ 2015년 gilt.com의 마이크로 서비스



9.3 Netflix.com

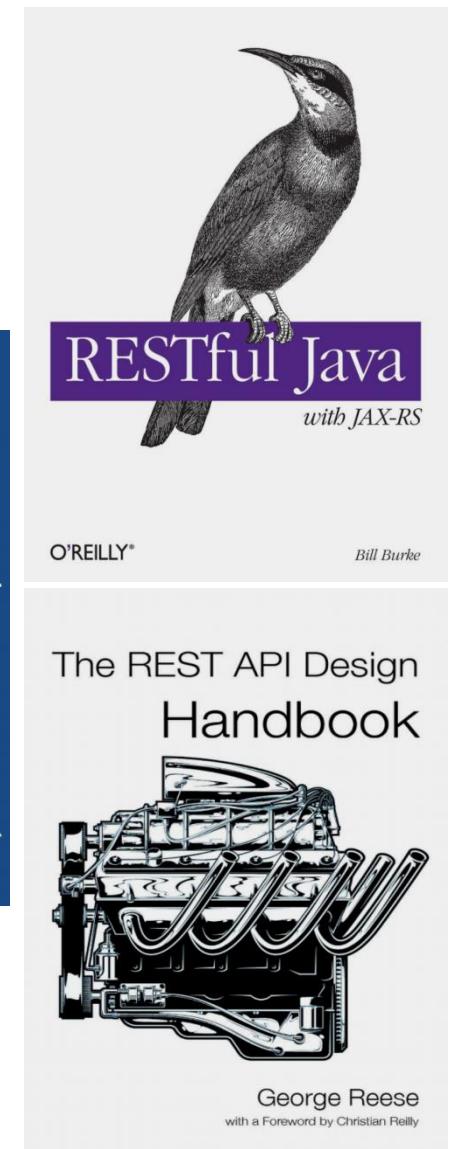
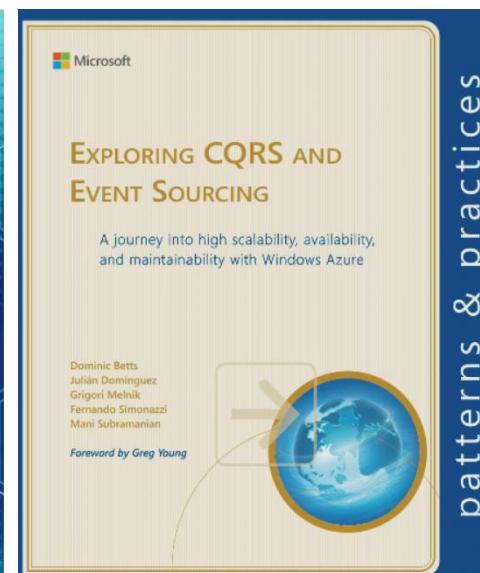
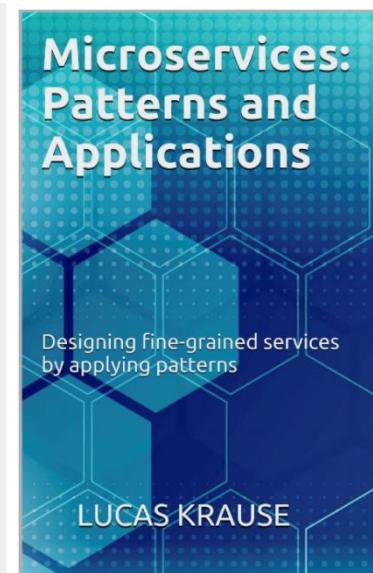
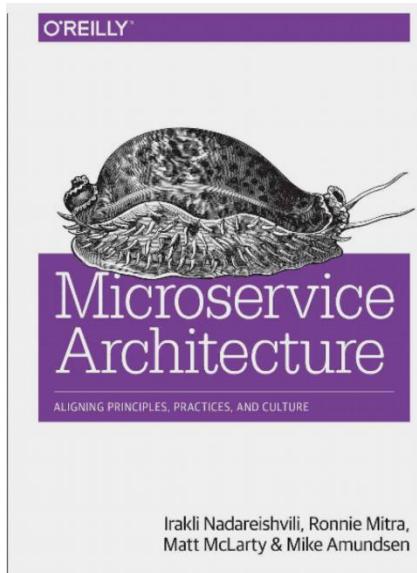
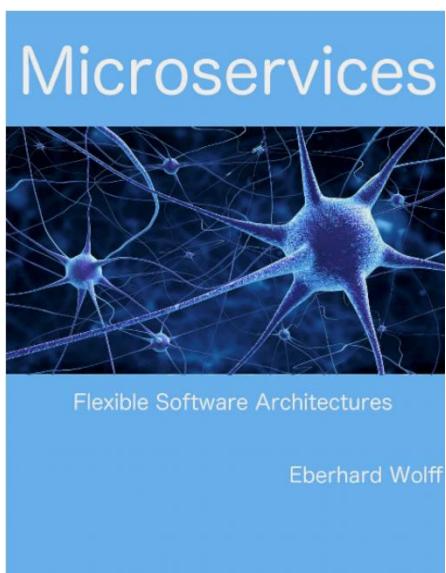
✓ Netflix의 서비스 구성을 살펴봅니다.

✓ Scalable Microservices at Netflix. Challenges and Tools of the Trade, Sudhir Tonsem, QCon2014



[참조자료]

- ✓ <http://microservices.io>
- ✓ <https://martinfowler.com/articles/microservices.html>
- ✓ <https://en.wikipedia.org/wiki/Microservices>



✓ 토론

감사합니다...

- ❖ 송태국 대표 (tsong@nextree.co.kr)
- ❖ 넥스트리컨설팅(주)
- ❖ www.nextree.co.kr